Autonomous mobile robots

Giving a robot the ability to interpret human movement patterns, and output a relevant response.

Sindre Eik de Lange[1, 2]

Stian Amland Heilund[1, 2]

[1]**Department of Computing, Mathematics and Physics, Western Norway University of Applied Sciences**

[2]**Department of Informatics, University of Bergen**

This thesis is submitted for the degree of

Master's in Software Engineering

June 2019

Acknowledgements

We would like to thank Mohn Medical Imaging and Visualization Centre for giving us the opportunity to work with superior equipment in great offices. Also, a big thank you to our fellow master student Sathiesh Kaliyugarasan for deep (learning) discussions about relevant topics and great memories from the lab.

Finally, we would like to give a special thank you to our supervisor Dr. Alexander Lundervold for introducing us to this field, and for his always enthusiastic help and guidance.

Abstract

The demographic challenges caused by the proliferation of people of advanced age, and the following large expense of care facilities, are faced by many western countries, including Norway (*"eldrebølgen"*). A common denominator for the health conditions faced by the elderly is that they can be improved through the use of physical therapy.

By combining the state-of-the-art methods in deep learning and robotics, one can potentially develop systems relevant for assisting in rehabilitation training for patients suffering from various diseases, such as stroke. Such systems can be made to not depend on physical contact, i.e. socially assistive robots.

As of this writing, the current state-of-the-art for action recognition is presented in a paper called "Spatial Temporal Graph Convolutional Networks for Skeleton-Based Action Recognition", introducing a deep learning model called spatial temporal graph convolutional network (ST-GCN) trained on DeepMind's Kinetics dataset. We combine the ST-GCN model with the Robot Operating System (ROS) into a system deployed on a TurtleBot 3 Waffle Pi, equipped with a NVIDIA Jetson AGX Xavier, and a web camera mounted on top. This results in a completely physically independent system, able to interact with people, both interpreting input, and outputting relevant responses.

Furthermore, we achieve a substantial decrease in the inference time compared to the ST-GCN pipeline, making the pipeline about 150 times faster and achieving close to real-time processing of video input. We also run multiple experiments to increase the model's accuracy, such as transfer learning, layer freezing, and hyperparameter tuning, focusing on batch size, learning rate, and weight decay.

Table of contents

List of figures

List of tables

Chapter 1

Introduction

The Silver Tsunami is a term coined in 2002 by professor M. F. Maples, referring to the *Baby Boomer* generation in the US: the 76.000.000 children born between 1946 and 1964 (Maples, 2002). This generation benefited from numerous medical advances made in the 20th century and the resulting increase in life expectancy. Consequently, this has led to a large generation of elders who will struggle with conditions such as cancer, osteoporosis, arthritis, Parkinson's, and stroke. The demographic challenges caused by the proliferation of people of advanced age, and the following large expense of care facilities, are faced by many western countries, including Norway (*"eldrebølgen"*). A common denominator for the health conditions faced by the elderly is that they can be improved through the use of physical therapy (Pergolotti et al., 2014) (Schröder et al., 2012) (jeffersoncountyhealthcenter, 2017).

In the US, physical therapists are said to increase in demand by 31% by 2026 (of Labor Statistics, 2018), compared to the average 7% overall increase of all the occupations in the US economy. *"Historically, costs for physical therapy were unnoticed by Medicare, but because of the dramatic increase, these costs are deemed unsustainable."* (Brennan, 2012)

An effort to address this dramatic increase in demand for physical therapists is the development, and implementation, of active-assistive robots as a supplement for the rehabilitation process. These are robots in which the patient actively interacts with, to complete one or multiple movements.

Technology cannot replace human touch and care, but considering the scalability and possible ubiquity of robot-based solutions, they have the potential to become a useful supplement.

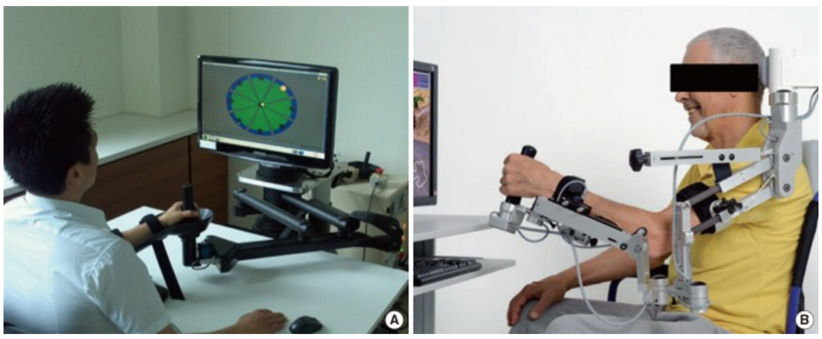
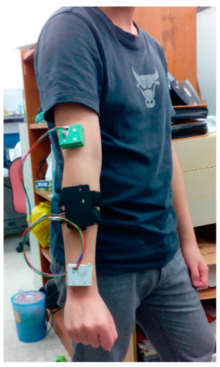


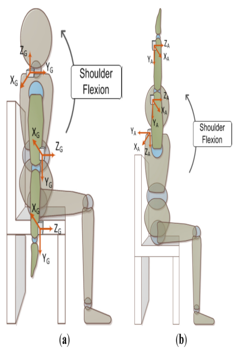
Fig. 1.1 Example of robotic devices for motor functions training, distributed under the terms of the Create Commons Attribution Non-Commercial Licence (Chang and Kim, 2013).

One large research area for such robots is post-stroke patient's rehabilitation (Poli et al., 2013), which is *"the leading cause of movement disability in Europe, and the US"* (Rosamond et al., 2008). Furthermore, it is estimated by the World Health Organization (WHO) that stroke events in Europe will increase by 30% between 2000 and 2025 (Truelsen et al., 2006).

An alternative to robots is the usage of *inertial measurements units* (IMUs), shown in fig. 1.2, which is electronic devices equipped with accelerometers and gyroscopes to report orientation (Spartonnavex, 2015). Patients can wear one or more IMUs, which measures the patient's movement.



(a) IMUs attached to patient, capturing data about the patient's movements.



(b) A patient moving with IMUs.

Fig. 1.2 IMUs example, distributed under Creative Commons Attribution License (Li et al., 2015)

A common challenge of IMU-based and most robot-based approaches is that they require physical contact with the patient. For active-assistive robots, the physical contact creates safety concerns, while the IMUs can be challenging to attach without professional supervision or assistance.

This thesis will aim to investigate methods for mitigating these challenges by creating a *socially-assistive robot*, i.e. a robot that interacts with the patient through social interactions. Examples of social interactions are auditory interactions, by talking or playing sounds, or visual interactions based on movements. Equipping the robot with sensors to give it the ability to interact with the patient results in a *"social exchange"* (Lumen, n.d.) between the patient and the robot. To achieve this, we will combine state-of-the-art approaches in both robotics and deep learning.

Initially, the goal is to create a system able to interpret input from the user in the form of human movement, i.e. *human action recognition*. The current state-of-the-art in human action recognition is based on deep learning models, e.g. the one presented in the paper *"Temporal Graph Convolutional Networks for Skeleton-based Action Recognition"* (Yan et al., 2018). This novel model, called *spatial temporal graph convolutional networks* (ST-GCN), has the ability to learn from both the spatial and temporal patterns in data, and it achieved substantial improvements over mainstream methods on two large datasets, Kinetics (Kay et al., 2017) and NTU RGB+D (Shahroudy et al., 2016b). We will explore the model's abilities in our context, namely movements relevant for physical rehabilitation.

For the robotic software and communication between components in our system, we will use the Robot Operating System (ROS). This is an open source framework designed to address the need to enable communication within systems consisting of different components and subsystems, often operating in distinct programming languages, i.e. heterogeneous systems. According to (Yoonseok Pyo, 2017), ROS is the most popular robot software platform. This makes ROS a natural choice for our project.

In this thesis we extend the code base made available by the authors of ST-GCN on GitHub, adding several features:

1. Extract a subset of both specific action classes, and random ones, from data derived from the Kinetics dataset
2. Download, process and train on a new action class
3. Combine the data from 1) and 2), and update necessary parameters for training, validation, and testing
4. Implement transfer learning, using their model trained on the Kinetics dataset

5. Increasing the effects of transfer learning by
   (a) Modifying the network architecture
      - Adding fully connected layers(s)
      - Adding extra spatial-temporal graph convolutional layer(s)
   (b) Implement layer freezing/unfreezing
6. Replacing their pose estimation software, OpenPose (Cao et al., 2018), with the significantly faster tf-pose-estimation (tf pose, 2019)
7. Extend their validation and testing functionality to include storage of:
   - Confusion matrix
   - Summary score file, containing *train loss*, *validation loss* and *accuracy*
   - Overview of each file in the validation set, and the values outputted by the model for each of those files
8. Implement data augmentation for newly defined action classes by modifying the videos
   - Frame flipping
   - Frame zooming

Our code in its entirety also resides on GitHub: https://github.com/Sindreedelange/st-gcn.

The main contributions of this thesis are:

- The aforementioned extension to the state-of-the-art project for action recognition in deep learning, making it easier to tailor for specific projects, and giving a better overview of the model evaluation process
- Defining a new relevant action class for human movement, namely *jumping jacks*, with a CSV file containing URLs, start- and stop time, and label for 199 YouTube videos, resulting in

  $$199 * 2 * 2 = 796$$

  videos for our newly defined action class when using our implemented data augmentation functionality

- Combining state-of-the-art tools in robotics and deep learning in order to make a proof of concept system, relevant for the rehabilitation process

This thesis will be split into two parts. Part I is an overview of the underlying theory of deep learning and robotics, starting with machine learning, building towards deep learning, ultimately explaining the theory behind the ST-GCN model. In Chapter 2 and 3 we introduce the fundamentals of machine learning and deep learning, respectively, explaining model selection, common problems when training a model, and model evaluation. Chapter 4 build on the deep learning models introduced in section 3.2 and 3.3 to explain the ST-GCN model. Lastly, in chapter 5 we give an overview of robotics and computer vision.

In part II we introduce our experiments. Chapter 6 starts with retraining ST-GCN on action classes relevant for the rehabilitation process, continuing with the implementation of our pipeline in its entirety by combining hardware- and software. We conclude chapter 6 with an evaluation of the system, and its components. This leads to chapter 7 and efforts to improve the system by decreasing its inference- and training time while increasing its accuracy. All of the experiments are evaluated using relevant, quantitative metrics, such as confusion matrix, accuracy, training loss and validation loss.

Ultimately, in chapter 8 we summarize our results and sketch out a possible path for relevant further work. All figures in this thesis have been made using Matplotlib (Hunter, 2007), Seaborn (Waskom et al., 2018), yED Graph Editor (yWorks GmbH, n.d.), and NN-SVG (LeNail, 2019), with SVG icons from FlatIcon. Furthermore, the LaTeX template is a thesis template for Cambridge University Engineering Department.

Part I

Background

The first part of this thesis will focus on the background of the various fields our thesis touches, such as artificial intelligence, machine learning, robotics, and computer vision. After reading part I, the reader will have enough background knowledge to read and understand part II. Our work is a combination of robotics and machine learning. The machine learning running in our robotics system is our main contribution, as the robotics software components are relatively easy to construct using modern robotics frameworks. Our discussion will therefore focus on machine learning, and our introduction to robotics in chapter 5 is comparably short.

Chapter 2

Artificial intelligence & machine learning

Artificial intelligence (AI) is a technical science aiming to equip machines with human-like intelligence (Song, 2018). A practical definition of AI is one proposed by (Pérez et al., 2018): *"Artificial intelligence is the study of human intelligence and actions replicated artificially, such that the resultant bears to its design a reasonable level of rationality"*. Today, AI is used in many domains, such as online advertising, driving, aviation, medicine, personal assistance, and image recognition. A specific example of AI is autonomous cars, using AI-algorithms on data from lidar sensors and cameras, equipping the cars with the ability to make intelligent decisions about traffic behavior.

AI is an umbrella term, covering subfields such as *machine learning* (ML) and *deep learning* (DL).

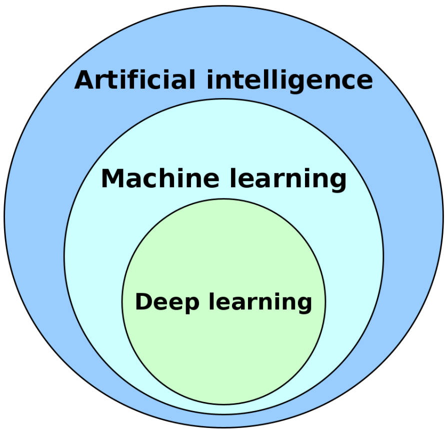
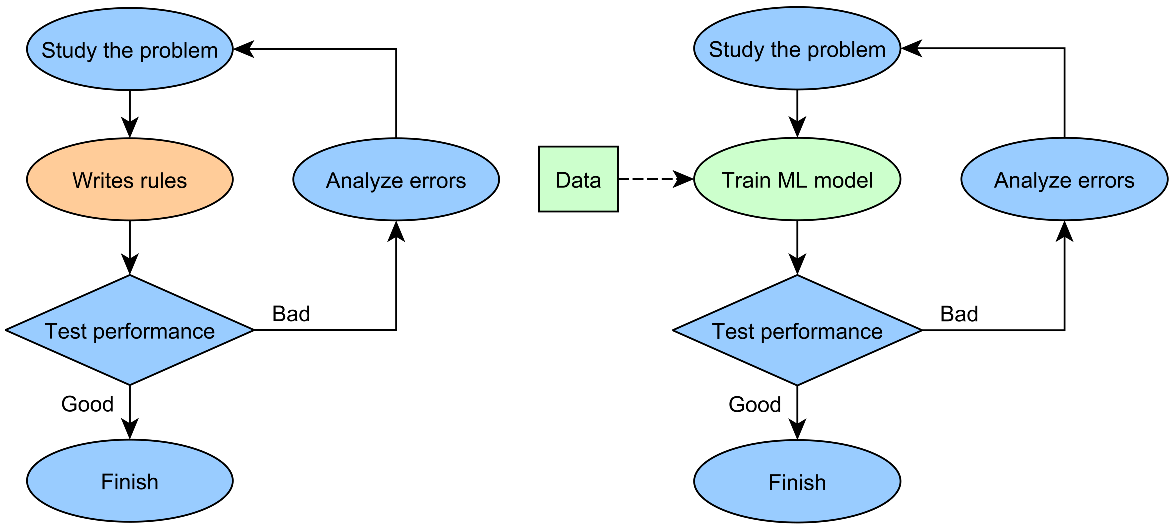


Fig. 2.1 The relationship between AI, ML and DL

The first general definition of ML was made in (Samuel, 1959): *"Machine learning is the field of study that gives computers the ability to learn without being explicitly programmed."* Later, in 1997, a more engineering-oriented definition was given in (Mitchell, 1997): *"A computer program is said to learn from experience E with respect to some task T and some performance measure P, if its performance on T, as measured by P, improves with experience E."*

An example can be a spam filter, learning to classify emails as spam or not spam. In this case, the task T is to detect if a new email is spam or not, the experience E is all the previous emails that the program has learned from (data), and the performance measure P has to be defined; for example the ratio of correctly classified emails (accuracy) (Géron, 2017).



(a) Traditional approach (b) Machine learning approach

Fig. 2.2 Traditional programming vs. machine learning, with inspiration from (Géron, 2017).

Compare this to traditional programming, shown in fig. 2.2a, continuing with the spam-example. To write such a program the programmer has to detect different patterns in the email, distinguishing spam and not spam mail, and write a detection algorithm for each of these patterns. This will most likely result in a long list of complex rules that are hard to maintain and does not generalize well.

A classifier based on ML, shown in fig. 2.2b, automatically learns complex patterns during training. Instead of manually writing complex rules, data is given to the model so that it can automatically learn how to classify the emails. This program is much shorter, easier to maintain, and most likely more accurate (Géron, 2017).

With that said, the performance of a ML model is dependent on the quality and quantity of the data used to construct the model. A high quality dataset has a large number of unique data points corresponding to the model's task, for instance, if the task is to classify spam or not spam, then the model needs a large dataset consisting of unique emails of both spam and not spam, to maximize its performance. Such high-quality datasets can be hard to obtain in some domains. Luckily, there are techniques for maximizing the potential of limited datasets. These considerations and other fundamental topics in machine learning will be presented in the coming sections.

2.1 Fundamentals of machine learning

ML is a huge field with a large number of subfields for many different types of problems. In this section, the most important parts of this field will be introduced, providing the reader with knowledge and context for understanding the following chapters.

2.1.1 Building blocks

ML can roughly be split into three somewhat overlapping classes, each with their own subclasses;

1. Supervised learning
2. Unsupervised learning
3. Reinforcement learning

This section will give a short introduction to these three domains.

Supervised, unsupervised and reinforcement learning

In supervised learning, each training example consists of input variables, matrix $\vec{X}$, and an output, vector $\vec{y}$. By being fed enough training examples, the model learns the connection between $\vec{X}$ and $\vec{y}$. A trained model can then be tested by trying to predict $\vec{y}$, given $\vec{X}$, where the prediction is referred to as $\hat{y}$. A more detailed overview of how machine learning models are trained will be presented in section 2.1.2.

Unsupervised learning focuses on detecting hidden patterns and structures in data. Unlike supervised learning, unsupervised learning does not have access to the correct output $\vec{y}$, only the input $\vec{X}$. Common unsupervised learning tasks are clustering and dimensionality reduction of $\vec{X}$.

Reinforcement learning is about making an agent, e.g. a robot, perform suitable actions based on the environment. The agent gets rewarded when performing suitable actions, and punished when making wrong decisions. By trying enough times, the agent will hopefully learn the most appropriate action to do next (Bajaj, 2014). This is achieved using various learning algorithms, with Q-learning (Watkins, 1989) being one of the most popular and well-known examples.

Regression and classification in supervised learning

Regression and *classification* are two different types of tasks within machine learning. In supervised learning, a regression algorithm tries to predict $\vec{y}$ as a continuous variable, given $\vec{X}$. For example, when provided with a dataset about houses, the goal can be to create a model that predicts the house prices based on other data about the houses.

A classification algorithm, on the other hand, tries to predict $\vec{y}$ as a categorical variable, given $\vec{X}$. Continuing with the example from above, instead of $\vec{y}$ being the specific sum, a classification algorithm will categorize the price, for example over/under the median (Garbade, 2018). A supervised classification problem falls into one of the following tasks:

- Binary: $\vec{X}$ is classified to $\hat{y}$, where $\hat{y}$ is one of two non-overlapping classes.
- Multi-class: $\vec{X}$ is classified to $\hat{y}$, where $\hat{y}$ is one of > 2 non-overlapping classes.
- Multi-labelled: $\vec{X}$ is classified to $\hat{y}$, where $\hat{y}$ is several of > 2 non-overlapping classes.
- Hierarchical: $\vec{X}$ is classified to $\hat{y}$, where $\hat{y}$ is one class which is divided into subclasses or grouped into superclasses.

(Sokolova and Lapalme, 2009)

2.1.2 Training a model

To give a short explanation of how a machine learning model can be trained, linear regression will be used as an example. To make a prediction, a linear regression model computes a weighted sum of the input features, and adds a constant called the bias term, as shown in eq. 2.1 (Géron, 2017).

$$\hat{y} = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n \tag{2.1}$$

- $\hat{y}$ is the predicted value
- n is the number of features
- x_i is the i^{th} feature value
- θ_j is the jth model parameter (including the bias term θ_0 and the feature weights $\theta_1, \theta_2, \cdots, \theta_n$

Training a linear regression model means setting the *parameters* (bias and feature weights) such that $\hat{y}$ becomes as close to the target as possible. For example, finding the line in fig. 2.3 that best fits the data points.

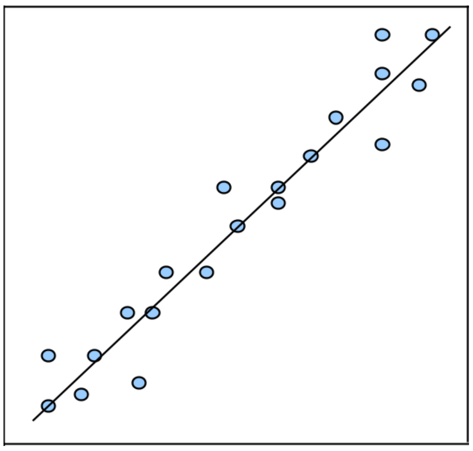


Fig. 2.3 Linear regression. The goal is to fit the line to the data points.

To set the parameters such that $\hat{y}$ becomes as close to the target as possible, one first needs to define a function that measures this distance, i.e. an *error measure*. One common error measure for regression models is the root mean squared error (RMSE), explained in section 2.1.4. Training a linear regression model means minimizing a function expressing the error, also called a *loss function* or a *cost function*. One method for minimizing the error function is using gradient descent. How this works is shown in the following example.

Gradient descent example

To demonstrate how a model is trained using gradient descent, the line equation $\hat{y} = ax + b$ will be used as an example, where a and b are weights that will be updated during training, and x is the only input variable, i.e. feature value. In this example, the weights are initially set to $a = 2$ and $b = 3$. The input variable $x = 2$ is fed to the model, with the target $y = 5$. The goal is to adjust a and b such that $\hat{y}$ becomes as close to 5 as possible. This example shows one iteration of weight updates. However, normally the training process consists of many iterations, taking one small step towards the minimum loss per iteration.

First, the model makes a prediction based on the given input variable x

$$\hat{y} = ax + b = 2 \cdot 2 + 3 = 7 \tag{2.2}$$

The model predicts 7, but we want it to predict 5. To calculate the distance between the prediction and the target, a loss function has to be defined. In this example, the squared error (SE) will be used, shown in eq. 2.3.

$$SE = (\hat{y} - y)^2 = ((ax + b) - y)^2 \tag{2.3}$$

In gradient descent, for each weight, the partial derivative of the loss function with respect to the weight is calculated to get the direction of the weight update (recall that the gradient of a function points in the direction of steepest increase). In this case, the weights are a and b, resulting in the partial derivatives below.

$$\frac{\partial SE}{\partial a} = 2x((ax + b) - y) \tag{2.4}$$

$$\frac{\partial SE}{\partial b} = 2((ax + b) - y) \tag{2.5}$$

Feeding the values for a, x, b and y to the equations gives

$$\frac{\partial SE}{\partial a} = 2 \cdot 2((2 \cdot 2 + 3) - 5) = 8 \tag{2.6}$$

and

$$\frac{\partial SE}{\partial b} = 2((2 \cdot 2 + 3) - 5) = 4 \tag{2.7}$$

Now we have the tools needed to update the weights. The weights are updated by using equation 2.8, where γ is the *learning rate* deciding how much the weight will be updated.

$$w^{+} = w - \gamma \cdot \frac{\partial SE}{\partial w} \tag{2.8}$$

Using $\gamma = 0.05$, the new weights becomes

$$a^{+} = a - \gamma \cdot \frac{\partial SE}{\partial a} = 2 - 0.05 \cdot 8 = 2 - 0.4 = 1.6 \tag{2.9}$$

$$b^{+} = b - \gamma \cdot \frac{\partial SE}{\partial b} = 3 - 0.05 \cdot 4 = 3 - 0.2 = 2.8 \tag{2.10}$$

Now, the model can be tested again with the new weights and the same input

$$\hat{y} = ax + b = 1.6 \cdot 2 + 2.8 = 6 \tag{2.11}$$

The prediction is now closer to the target than the first prediction was, i.e. the model has improved by being trained. By continuing updating the weights over more iterations, the prediction will get closer and closer to the target, and the model will get better and better until a minimum is reached, as visualized in fig. 2.4.

Hyperparameter tuning

Hyperparameters are set before the training and are usually static during training, unlike the model's weights. In the example above the learning rate was set to 0.05. This hyperparameter has a big impact on the model's performance and is therefore important to set correctly. As shown in fig. 2.5, if the learning rate is too large, the risk of jumping out of a local minima increases, and consequently, the complexity of

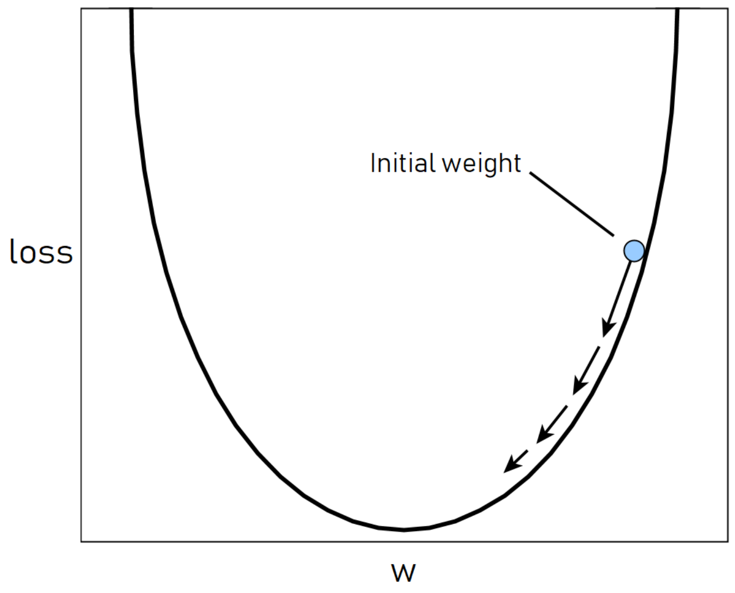


Fig. 2.4 Gradient descent. After each iteration of gradient descent the weights are adjusted such that the loss decreases.

optimizing the weights. On the other hand, a learning rate that is too small makes the convergence towards the local minima slow. Hence, the learning rate impacts both the training and the performance of a model.

To demonstrate how the learning rate impacts the training, we will increase the learning rate in the previous example to 0.5, giving us the weights:

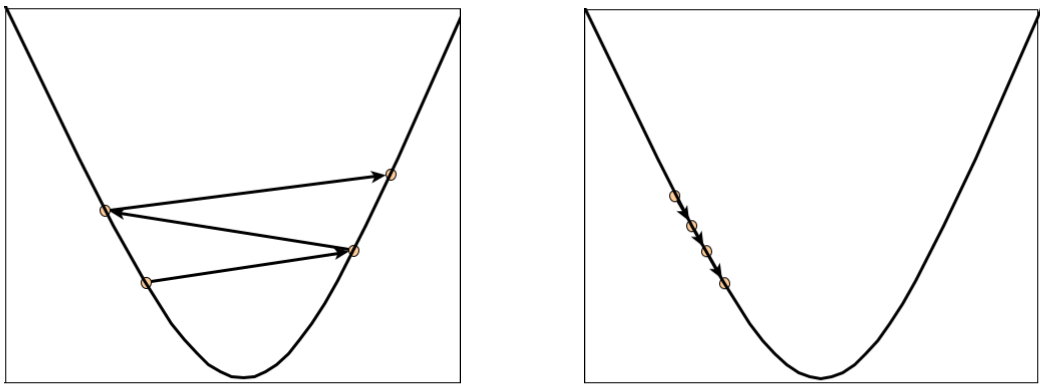
$$a^{+} = a - \gamma \cdot \frac{\partial SE}{\partial a} = 2 - 0.5 \cdot 8 = 2 - 4 = -2 \tag{2.12}$$

$$b^{+} = b - \gamma \cdot \frac{\partial SE}{\partial b} = 3 - 0.5 \cdot 4 = 3 - 2 = 1 \tag{2.13}$$

Testing the model with the new weights

$$\hat{y} = ax + b = -2 \cdot 2 + 1 = -3 \tag{2.14}$$

As shown, changing the learning rate from 0.05 to 0.5 resulted in $\hat{y} = 6$ and $\hat{y} = -3$, respectively. By using learning rate 0.5, the weights are updated too much, resulting in a prediction further away from the target than the initial prediction, i.e. the learning rate is too large.



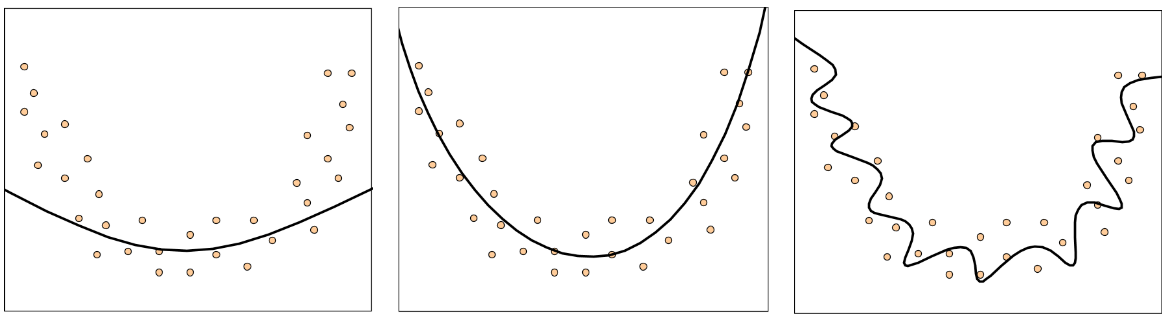
(a) Large learning rate. If the learning rate is too large, the risk of jumping out of a minima increases.

(b) Small learning rate. If the learning rate is too small, the steps against the minima are too small, resulting in a slow convergence.

Fig. 2.5 Large vs. small learning rate

2.1.3 Reducing generalization error

When creating a machine learning model, the goal is to make it able to generalize well to new data, i.e. minimizing the model's generalization error. Minimizing this error is equivalent to reducing the problem of *overfitting* - a phenomenon which occurs when a model focuses too much on the details and noise in the training data, as shown in fig. 2.6c. Conversely, if the model struggles to adapt to patterns in the data, as shown in fig. 2.6a, the model is *underfitting*.



(a) Underfitting - the model struggles to adapt to the patterns in the data.

(b) Just right - the model manages to adapt to the training data without focusing too much on the noise.

(c) Overfitting - the model focuses too much on the noise in the training data.

Fig. 2.6 Underfitting vs. overfitting

Finding the balance between overfitting and underfitting is directly connected to the so-called *bias-variance trade-off*. When given a training set of size k, bias measures how close the model's average prediction (over all possible training sets with size k) is to the target. Variance measures how much the model's prediction changes for the different training sets (Kohavi et al., 1996). While more powerful models reduce bias, they increase the variance and vice versa (Domingos, 2000).

As shown in fig. 2.6a, when the model is underfitting the variance is low (because y barely changes when X changes) and the bias is high (because the average distance from the line to the data points is high). For the model in fig. 2.6c the situation is the opposite. The challenge is to find the optimal point in this trade-off (middle plot), which varies from application to application.

To improve the performance of an underfitting model one can give it more data, or create a more complex model. Conceptually, one could think that if supplying more data to an underfitting model helps, then removing data should help an overfitting model. This is not the case, at all, as one almost always want a larger dataset. To improve a model that is overfitting one tries to reduce the aforementioned generalization error, which can be done by using different techniques. One fundamental practice is - before training a model on a dataset - to divide the dataset into three subsets; *training set*, *validation set*, and *test set*.

- Training set: Often 80% of the entire dataset. Used to fit the model, meaning that the model sees the data and learns from it during the training process.
- Validation set: Often 10% of the dataset. Used for evaluation of models during model selection, and for fine-tuning hyperparameters based on this evaluation, i.e. hyperparameter tuning. The model sees this data but does not directly learn from it. The model is overfitting if the error on the validation set is larger than the error on the training set.
- Test set: Often 10% of the dataset. Used for evaluation of the model after it is completely trained using the training set and validation set. This data is completely unseen for the model before this step, and the test set is only used once to get the model's final performance measure.

Batch size

Another important hyperparameter when training a model is *batch size*. If one has a training set containing N data points, using batch size 1 means that all of these points, one at a time, will be sent through the model during training. This means that the model's weights will be updated for each data point, i.e. N weight updates.

However, by using data parallelization one can experiment with increasing the batch size from 1 to a larger and potentially better size. This can be done through distributing the computations over multiple either central- or graphics processing units. Specifically, say one increases the batch size to 32, it means that 32 data points will run through the model training procedure at once. This leads to $\frac{N}{32}$ *mini batches*, and as many weights updating processes. This leads to a decrease in both training time and generalization error, because the number of operations is reduced, and one updates the model based on the loss from a larger sample size (Smith et al., 2017).

Regularization

Techniques that aim to improve a model's generalization ability without decreasing training performance are called *regularization techniques.* Two types of regularization are

1. L1 Regularization, or Lasso Regularization
2. L2 Regularization, or Ridge Regularization

L1 regularization adds a penalty to the error function. This penalty is the sum of the absolute value of the error. More specifically, given the error function *mean squared error* (MSE), shown in eq. 2.15.

$$MSE = \sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{2.15}$$

Adding the penalty leads to the error function in eq. 2.16, where λ is the tuning parameter deciding how strong the penalty is, and β is the weights.

$$MSE_{L1} = \sum_{i=1}^{n}(y_i - \hat{y}_i)^2 + \lambda \sum_{i=1}^{p} |\beta_i| \tag{2.16}$$

(Boehmke, n.d.)

Adding the sum of the absolute value of the weights to the error function forces the model to penalize weights with big values. Machine learning models are more affected by large weights than small ones, so by penalizing large weights the risk of overfitting is reduced.

Like L1 regularization, L2 regularization also adds a penalty to the error function. Instead of using the sum of the absolute value of the weights as a penalty, the sum of the squared weights is used. The error function then becomes as shown in 2.17.

$$MSE_{L2} = \sum_{i=1}^{n}(y_i - \hat{y}_i)^2 + \lambda \sum_{i=1}^{p} \beta_i^2 \tag{2.17}$$

(Boehmke, n.d.)

Cross validation

To choose a suitable batch size and to decide to what extent various regularization techniques should be used, one must investigate the effect on the model's generalization ability. One commonly used technique for getting an estimate of how well a model performs on new data is *cross validation*. In *k-fold* cross-validation, the training data is folded into k subsets. The model is then trained k times, such that each time, one of the k subsets is used for validation, and the other subsets are used for training, as shown in fig. 2.7. The error is the average of the errors after k-epochs (Gupta, 2017).

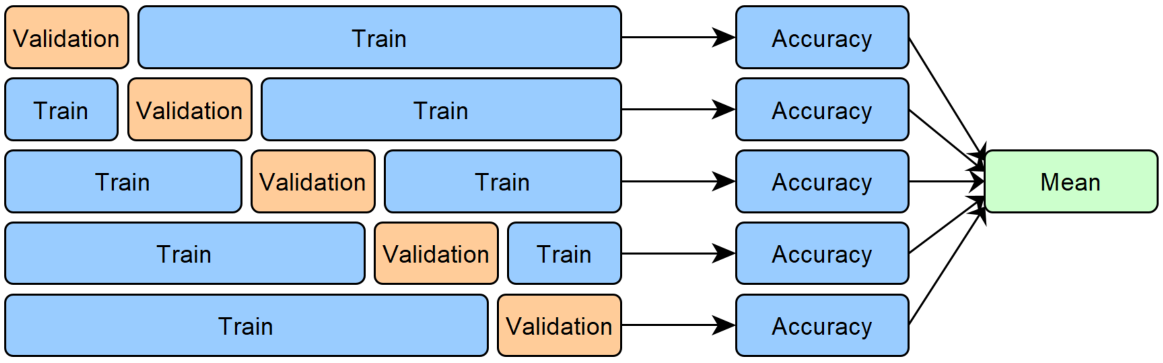


Fig. 2.7 Cross validation - the training data is folded into k subsets, before the model is trained k times on the different subsets, using one subsets for validation and the others for training for each iteration.

This thesis focuses on the regularization methods most commonly used in deep learning, and such methods will therefore be presented in section 3.1.4.

2.1.4 Performance measures

How a model's performance is measured depends on whether it is a regression or a classification model.

Performance measures for classifiers

A classifier can be evaluated by for example computing the number of correctly classified objects divided by the total number of objects, giving the accuracy - the most common evaluation method for classifiers (Sokolova and Lapalme, 2009). However, accuracy alone does not necessarily indicate how good the model is. For example, if one class A in the dataset occurs much more frequently than the other classes, i.e. an unbalanced dataset, then the model can achieve relatively high accuracy by predicting A every time. To detect this, one could use a *confusion matrix*.

The confusion matrix is a tool for evaluating the *performance*, *sensitivity* and *specificity* of a classifier. The main idea is to compare the predictions and the targets, then count the number of times each class A was predicted as class B. The confusion matrix can, for instance, show how many times a model confuses spam with no spam, and vice versa, as shown in fig. 2.8. In this case, out of 11 spam-emails, the model classified 9 of them correctly, and out of 52 no-spam-emails, 47 was classified correctly. This does not only give information about how good the model performs but also in which areas the model can improve (Géron, 2017).

Performance measures for regression models

As mentioned in 2.1.2, the most popular performance measure for regression problems is the root mean squared error (RMSE), shown in eq. 2.18. RMSE indicates how much error the machine learning model makes in its predictions by calculating the distance between the prediction and the actual solution, where large errors are weighted more than small errors (Géron, 2017).

$$RMSE(\mathbf{X}, h) = \sqrt{\frac{1}{m}\sum_{i=1}^{m}(h(\mathbf{x}^{(i)}) - y^{(i)})^2} \tag{2.18}$$

- m is the number of instances in the dataset
- $\mathbf{X}$ is a matrix containing all feature values of all instances in the dataset

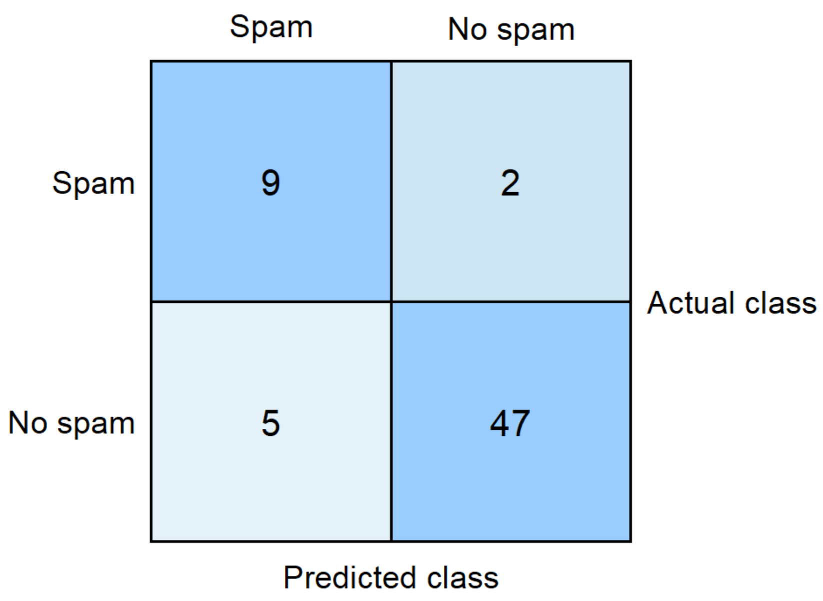


Fig. 2.8 Confusion matrix - spam-email example

- $\mathbf{x}^{(i)}$ is a vector of all the feature labels of the i^{th} instance in $\mathbf{X}$, and $y^{(i)}$ is its label
- h is the prediction function that gives the prediction by taking $\mathbf{x}^{(i)}$ as input

However, weighting large errors makes RMSE sensitive to outliers. Therefore, if there are many outliers, one may consider using an alternative performance measure for regression, called *mean absolute error* (MAE), shown in eq. 2.19 with the same notations as for RMSE (Géron, 2017).

$$MAE(\mathbf{X}, h) = \frac{1}{m}\sum_{i=1}^{m}\left|h(\mathbf{x}^{(i)}) - y^{(i)}\right| \tag{2.19}$$

RMSE and MAE are both performance measures that measure the distance between the prediction vector and the target vector. The goal is to make this distance as close to zero as possible when predicting on the test set.

To demonstrate the difference between RMSE and MAE we will use the example of housing prices, mentioned in 2.1.1. Given $\vec{X}$, comprised of information about 3 different houses, and the target vector $\vec{y}$, comprised of the corresponding house prices, the model predicts $\hat{y}^{(1)} = 50000$, $\hat{y}^{(2)} = 48000$, $\hat{y}^{(3)} = 75000$ while the actual labels are $y^{(1)} = 55000$, $y^{(2)} = 40000$, $y^{(3)} = 76000$. The predictions are visualized in fig. 2.9, where the black curve represents the actual house prices given features. The resulting RMSE and MAE are calculated below.

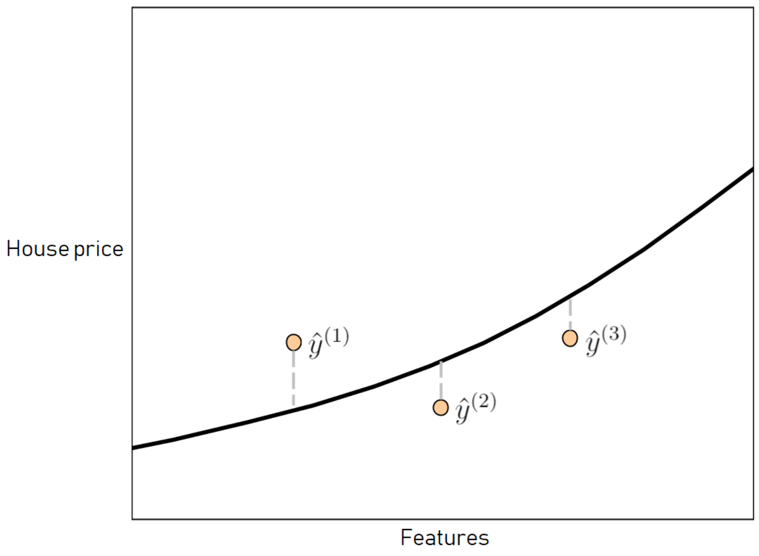


Fig. 2.9 Prediction errors - housing prices example. The grey, dashed lines represent the error between the prediction (orange dots) and the target (black curve).

$$
\begin{aligned}
RMSE(\mathbf{X}, h) &= \sqrt{\frac{1}{3}\sum_{i=1}^{3}(h(\mathbf{x}^{(i)} - y^{(i)})^2} \\
&= \sqrt{\frac{1}{3}((50000 - 55000)^2 + (48000 - 40000)^2 + (75000 - 76000)^2)} \\
&= \sqrt{30000000} \\
&\approx \underline{5477}
\end{aligned}
\tag{2.20}
$$

$$\begin{aligned} MAE(\mathbf{X}, h) &= \frac{1}{3}\sum_{i=1}^{3}\left|(h(\mathbf{x}^{(i)} - y^{(i)})\right| \\ &= \frac{1}{3}(|50000 - 55000| + |48000 - 40000| + |75000 - 76000|) \\ &\approx \underline{4667} \end{aligned} \tag{2.21}$$

As we can see from comparing the computed values in eq. 2.20 and eq. 2.21, RMSE outputs a larger value, resulting in a larger update value for the model's weights, causing its higher sensitivity to outliers.

2.2 Machine learning models

A well-known theorem in ML is the *no free lunch*-theorem, which establish that *"for any algorithm, any elevated performance over one class of problems is offset by performance over another class"* (Wolpert and Macready, 1997). For example, one cannot say that artificial neural networks are always better than random forests, because for each problem neural networks outperform random forests, there has to be a problem in which random forests outperform neural networks. This is an interesting theoretical theorem, however, in real situations, there are various reasons to prefer one model over others.

Many factors play a role when it comes to choosing the best suitable machine learning model, such as the size and complexity of the dataset. Therefore, different machine learning models should be trained on the same problem, before evaluating the performances to select the best model (Le, 2018).

The only machine learning model mentioned so far is linear regression. This section will give an overview of some of the other common machine learning models, how they work, and for which problems they are most suitable.

2.2.1 Decision trees

Decision trees can perform both regression and classification tasks. How a decision tree makes predictions is shown in fig. 2.10. Suppose that the model's task is to predict which class an instance belongs to; whale, lion, bird or cat. Start at the root node (the top). This node asks if the instance weighs more than 100 kilograms. If the answer is yes, then the algorithm moves on to the next node, which asks if the instance is a mammal. Finally, the tree reaches a leaf node, and the prediction will be the content of this node. So, the model traverses the tree, from top to bottom, until a leaf node is reached, which will be the prediction (Géron, 2017). This is an example of a decision tree performing a classification task. However, replacing the leaf nodes with numbers can convert it into a regression task.

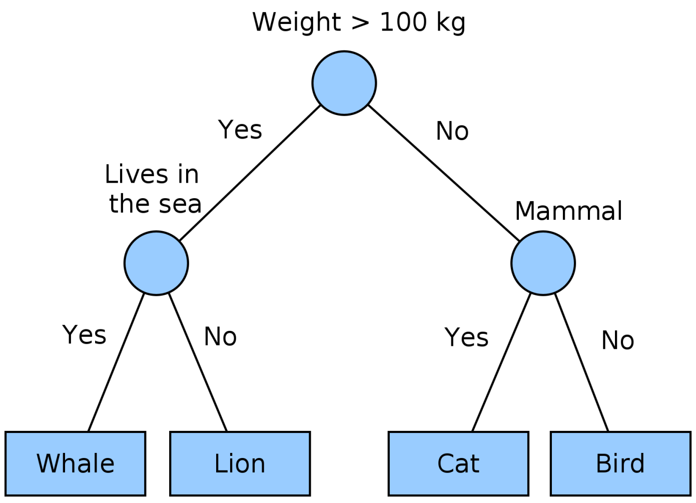


Fig. 2.10 Decision tree example. By starting at the top (root node), an input data point is traversed through the tree to finally end up on the final prediction in a leaf node.

The decision tree is built during training, which includes choosing the node's values, more specifically computing which features results in the optimal data partition, at each node. There are various algorithms for computing this, such as algorithms based on the *gini index* (Catalano et al., 2009) or *entropy* (Jaynes, 1957). During model training, each available feature is tested to see which one results in the largest information gain. More specifically, looking at fig. 2.10, given the three features' weight, lives in the sea, and mammal. Starting at the root node, the model can use either Gini index or entropy to calculate each feature's information gain, then split on whichever gives the largest one, which in this case is weight.

This process is repeated for the rest of the features in the data, continuing to either all of the features are used, or reaching an explicitly defined limit for information gain, defined in the hyperparameter list. More specifically, the information gained by splitting on feature $X <$ limit. Alternatively, there are two other hyperparameters designed to limit the training process: setting maximum depth, or a maximum number of leaves.

2.2.2 Random forest

A *random forest* model is an ensemble of decision trees, making a prediction by aggregating the predictions from all of the trees, by predicting the class that gets the most votes. A voting classifier like this often achieves higher accuracy than the best tree in the ensemble. This is made possible due to the *law of large numbers*, which is the principle behind the *wisdom of the crowd*.

This idea originates from a British scientist Francis Galton, who in 1906 tested it in a competition designed for guessing an ox's weight. Originally, he wanted to show that a crowd comprised of mainly non-intellectuals would give a correspondingly inaccurate answer. However, this prediction did not come to pass, as the crowd's guess was 1,197, and the actual weight was 1,198, prompting Galton to write *"The result seems more creditable to the trustworthiness of a democratic judgment than might have been expected"* (Surowiecki, 2005).

Here is a specific example from (Géron, 2017): think of a biased coin which has 51% chance of showing heads. If tossed 1000 times, the probability of obtaining a majority of heads is close to 75%. With 10 000 tosses, the probability climbs to 97%. Similarly, if an ensemble consists of 10 000 independent classifiers that are correct only 51% of the time, the majority vote of the ensemble is correct 97% of the time.

This principle makes random forests among the most powerful machine learning algorithms available today (Géron, 2017).

Fig. 2.11 illustrates how a random forest makes a prediction. The ensemble consists of six decision trees, where four of them predicts the same class, which ultimately becomes the majority class, hence the final output.

Why random forest? Recall how decision trees are trained in section 2.2.1, calculating each available feature's information gain at each node. Random forests, however, incorporates randomness when training. Instead of searching for the best feature in the

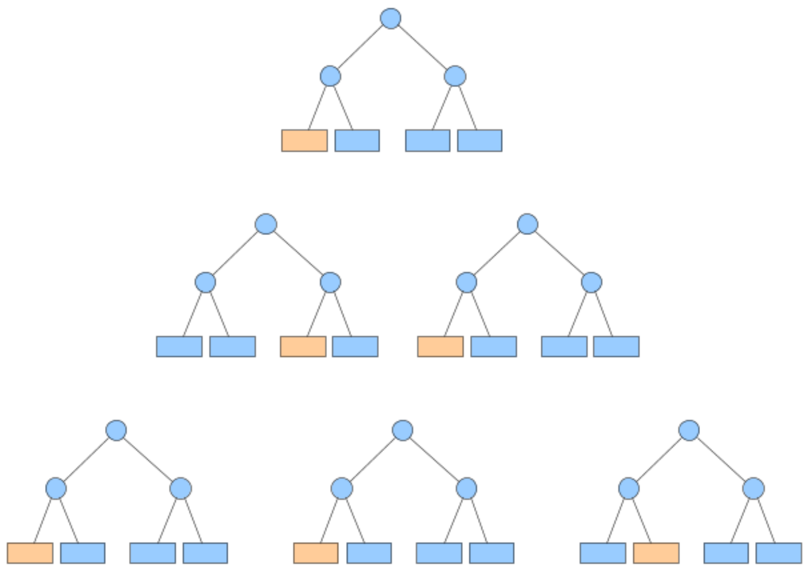


Fig. 2.11 Random forest architecture - by using an ensemble of decision trees to choose the major vote, a random forest model often achieves higher accuracy than the best tree in the ensemble.

whole dataset when splitting a node, it searches for the best feature among a subset of the data, that is randomly picked at every node. This is called bagging, short for bootstrap aggregating. In addition, each split is based on searching through a randomly selected set of all possible features (feature bagging). This results in different trees in the ensemble, trading a higher bias for a lower variance (Géron, 2017).

2.2.3 Support vector machines

A *support vector machine* (SVM) is a powerful ML model capable of performing both linear- and non-linear classification and regression tasks.

Support vector machines for classification

The main idea behind SVM classification is to separate classes by the largest possible margin, as shown in fig. 2.12. In other words, the goal is to fit the widest possible street between the classes. This is called *large margin classification*. This street is fully determined by the data points located at the edge of the street (the black points in fig. 2.12), named *support vectors*. Note that adding more data points outside of these support vectors will leave the decision boundary and the prediction unaffected.

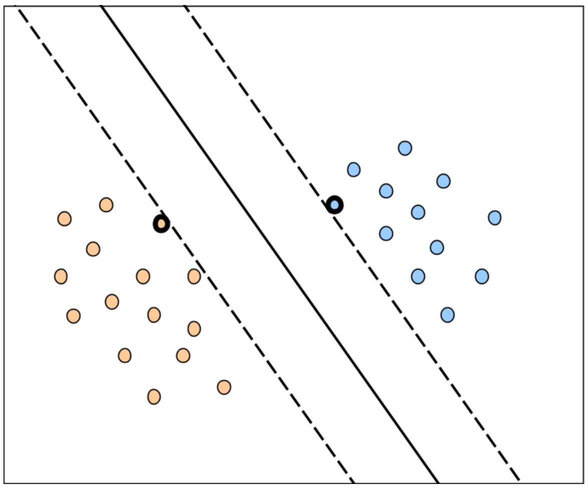


Fig. 2.12 Linear SVM classification - finding the line that separates classes by the largest possible margin.

Support vector machines for regression

Instead of trying to fit the largest possible street between the two classes, SVM regression tries to fit the line such that as many data points as possible are on the street, given a maximum street width, as shown in fig. 2.13.

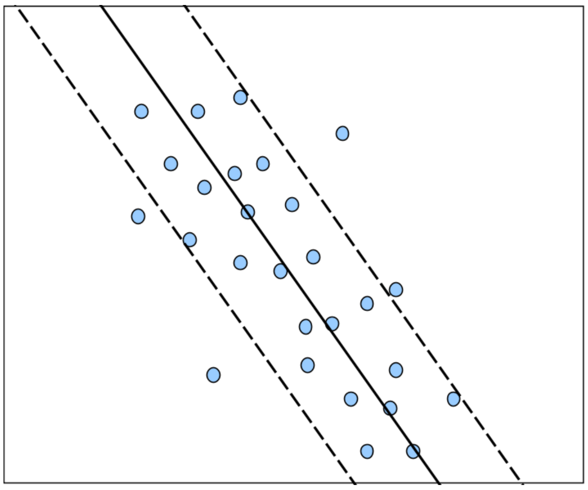


Fig. 2.13 Linear SVM regression - finding the line that fits as many data points as possible on the street, given a maximum street width.

Support vector machines training

During the training process, the model calculates the optimal line. What is optimal is dictated by whether it is a classification- or a regression task. However, the calculations are similar, in that they both use *kernel functions*. These are functions used to map data into higher dimensional space ($\rightarrow \infty$), in order to find the aforementioned optimal line. The use of such functions is only necessary if the data's complexity requires it, i.e. if it comprises > 2 features.

There are many possible kernels proposed by researchers. However, the four most basic kernels are:

- Linear: $K(\mathbf{x}_i, \mathbf{x}_j) = \mathbf{x}_i^T \mathbf{x}_j$
- Polynomial: $K(\mathbf{x}_i, \mathbf{x}_j) = (\gamma \mathbf{x}_i^T \mathbf{x}_j + r)^d, \gamma > 0$
- Radial basis function (RBF): $K(\mathbf{x}_i, \mathbf{x}_j) = exp(-\gamma \|\mathbf{x}_i - \mathbf{x}_j\|^2), \gamma > 0$
- Sigmoid: $K(\mathbf{x}_i, \mathbf{x}_j) = tanh(\gamma \mathbf{x}_i^T \mathbf{x}_j + r)$

Here, γ, r and d are kernel parameters (Hsu et al., 2003).

Each of these kernel functions is relevant for certain types of situations and data, with their own pros and cons, most notably the situation of linearly- vs. non-linearly separable data.

We plot the data separation of the Iris dataset (FISHER, 1936) using the four presented kernel functions, in fig. 2.14. Please note that in order to plot the data, it was necessary to choose two of the four features in the dataset. We used the Python library Mlxtend (Raschka, 2018) for plotting the kernels.

2.2.4 Other models

Machine learning is a field with a long history, and there are many models beyond the ones listed above that have been constructed and studied over the years. One model that has become almost unavoidable these days is the artificial neural network (ANN). ANN has become the most popular ML model in the last decade due to its state-of-the-art performance and almost complete dominance across several applications. There are many different versions of ANNs, such as convolutional neural networks (CNNs), and recurrent neural networks (RNNs). These models will be introduced in the following chapter.

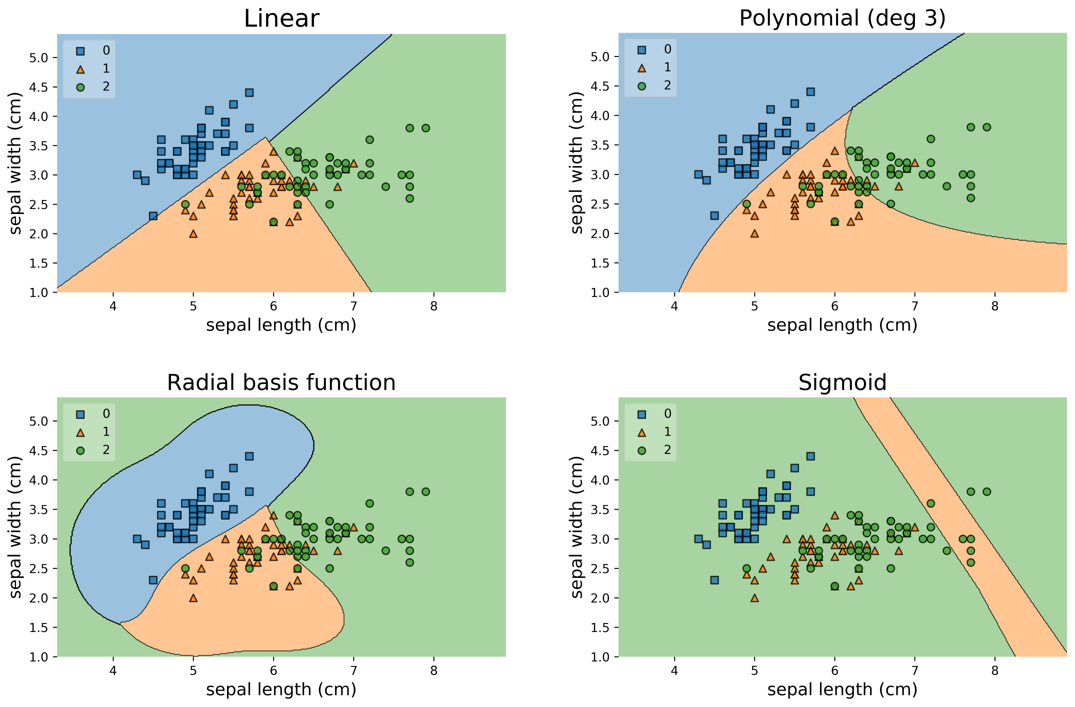


Fig. 2.14 The four most common SVM kernels plotted, using iris IRIS dataset.

Chapter 3

Deep learning

Deep learning refers to computational models comprised of multiple processing layers, used to learn data representations at multiple levels of abstraction (Lecun et al., 2015). The most common deep learning models are *artificial neural networks* (ANNs), with various implementations such as *convolutional neural networks* (CNNs), and *recurrent neural networks* (RNNs), each tailored for specific types of problems. For instance, the development of CNNs has led to breakthroughs in image processing, while RNNs are relevant for sequential data such as text and speech. This chapter will introduce fundamental aspects in deep learning, as well as an overview over the aforementioned deep learning models. The terms ANN and model will be used interchangeably.

3.1 Artificial neural networks

An ANN is made up by *neurons* and their weighted connections, where the neurons are organized in *layers*. As shown in fig. 3.1, the first layer, the middle layers, and the last layer are called the input layer, the hidden layers, and the output layer, respectively.

3.1.1 Basics

The first ANN appeared in 1958, named *perceptron*, shown in fig. 3.2. This is the simplest neural network architecture, and is based on a special neuron called *linear threshold unit* (LTU), visualized as the *step*-neuron in fig. 3.2. The LTU summarizes the connected input neurons multiplied with the corresponding weights, i.e. the weighted sum of the inputs, refer to eq. 3.1 (Géron, 2017).

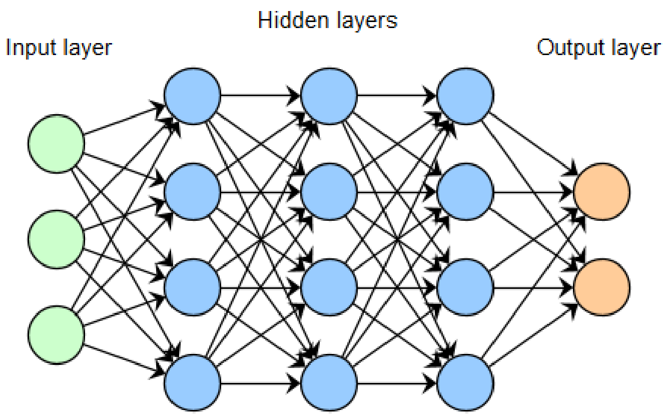


Fig. 3.1 Deep neural network architecture, consisting of one input layer, one or more hidden layers, and one output layer.

$$z = w_1 x_1 + w_2 x_2 + \cdots + w_n x_n = \mathbf{w}^T \cdot \mathbf{x} \tag{3.1}$$

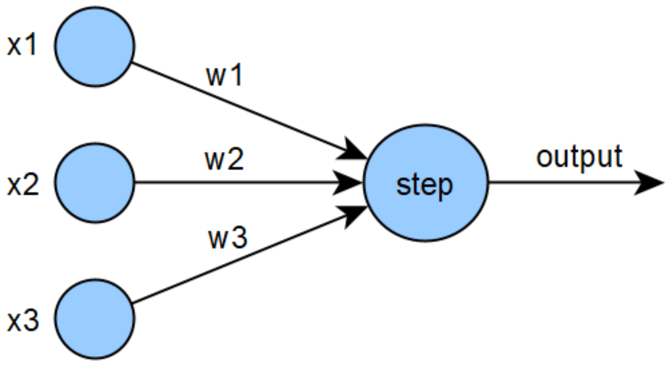


Fig. 3.2 Perceptron - the first neural network.

This weighted sum is then given to a step function, also referred to as *activation function*, which calculates the output. The most common step function used with perceptrons is the (linear) *Heaviside step function*, refer to eq. 3.2.

$$Heaviside(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases} \tag{3.2}$$

Combining several LTUs results in a larger neural network, like the one in fig. 3.1. However, a network only consisting of units with linear activation functions cannot handle non-linear data and is therefore incapable of learning complex patterns. Luckily, ANNs have improved since 1958, incorporating non-linear activation functions, giving them the ability to handle non-linear data.

Furthermore, key components such as improved regularization and normalization techniques have been developed. Also, more data and more compute (Moore's Law (Mollick, 2006)) has made more complex architectures possible. For example, adding many more layers to the ANNs have increased their practical expressiveness tremendously.

3.1.2 Training

Similarly to linear regression, explained in section 2.1.2, training an ANN means optimizing all of the network's weights, trying to minimize an error measure. During this process, the network is fed one batch at a time, refer to section 2.1.3, where each batch consists of one or more training instances. Based on each batch, the network makes its predictions, before a loss function, introduced in section 2.1.2, is used to compute the error for that batch. At first, the weights are randomly initialized, so to improve the network they have to be adjusted such that the error decreases. The most common way to update the weights in ANNs is through gradient descent.

Gradient descent on ANNs

Gradient descent on ANNs is a little more complicated than for linear regression, explained in section 2.1.2, even though the principle is the same. The difference is the way of calculating the gradients. ANNs consist of layers of variables depending on variables in the previous layer, again depending on variables in the previous layer, etc., making it more complicated to compute the gradients with respect to every weight. To calculate the derivative of the loss function with respect to a weight in the first layer, all the relevant derivatives with respect to the weights in the next layers are needed. Therefore, the method used for this is *backpropagation*. Starting with the last layer, the derivative of the loss function with respect to the weights is computed, which is used to find the derivative in the previous layer, and so on. This method, a form of automatic differentiation, is based on the *chain rule*, shown in eq. 3.3. If variable $\mathbf{z}$ depends on variable $\mathbf{y}$, which itself depends on variable $\mathbf{x}$, then $\mathbf{z}$ depends on $\mathbf{x}$ as well. The chain rule can then be stated as

$$\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx} \tag{3.3}$$

So far, only gradient descent for regression tasks using squared error as the loss function has been described. Alternatively, when faced with a classification task, a popular loss function is *categorical cross entropy* (CCE), also called *softmax loss*, refer to eq. 3.4. CCE is a combination of *softmax activation*, refer to eq. 3.5, and *cross-entropy loss* (CE), refer to eq. 3.6.

$$CCE = -\log\left(\frac{e^{s_p}}{\sum_{j}^{N} e^{s_j}}\right) \tag{3.4}$$

$$f(s_i) = \frac{e^{s_i}}{\sum_{j}^{C} e^{s_j}} \tag{3.5}$$

where s_j are the model's predicted scores for each class in C.

$$CE = -\sum_{i}^{C} t_i \log(f(s_i)) \tag{3.6}$$

Where t_i and s_i are the ground truth.

(West and O'Shea, 2017)

Like mean squared error (MSE), CEE measures the distance between two vectors. For classification tasks, these vectors are two probability distributions: the predicted and the actual. For example, given three classes: [cat, dog, bird]. If the input is "dog", then the actual probability distribution is [0, 1, 0], while the predicted one can be for example [0.2, 0.6, 0.2], representing the probabilities for each class. During training we want the predicted probability distribution to converge towards the actual probability distribution, which one needs a loss function for. For regression tasks the standard loss function is MSE, but for classification tasks, one will always use cross-entropy loss.

3.1.3 Transfer learning & fine-tuning

"Transfer learning is the improvement of learning in a new task through the transfer of knowledge from a related task that has already been learned" (Torrey and Shavlik, 2010). For example, if a model knows how to distinguish between images of cats and dogs, then it can use that knowledge to learn how to distinguish between other objects, such as wolfs and tigers, with increased accuracy and speed. Furthermore, it has even

shown promise when applied to seemingly unrelated datasets, such as using knowledge about human action recognition to improve medical image segmentation (Chen et al., 2019). Further, in addition to increased accuracy and decreased training time, the data size requirements are decreased, which is significant considering that the lack of useful data is *"one of the most serious problems in deep learning"* (Tan et al., 2018).

As mentioned in section 3.1.2, the training process of a model is about optimizing its weights, which are scalar values. These weights are randomly initialized, with efforts being made to investigate a general relationship between each layer's node values, such as in (He et al., 2015) and (Glorot and Bengio, 2010). Relying on random initialization is suboptimal, which is why transfer learning is useful. Specifically, a commonly used way to do transfer learning is to load weights from a pre-trained model into an untrained model as initial weights, often making the model's weights more favorable than the randomly initialized ones.

Freezing layers

Conceptually, when initializing model **1**'s weights with the weights from model **2** that is pre-trained on a specific task, model **1** is trained on the specific task before the training process has even started. However, when using transfer learning between models and tasks, the number of possible outputs, i.e. classes the network have, usually varies. This means that by editing the number of classes in the dataset, the final layer(s) will differ from model **1** to **2**. These layers are randomly initialized, i.e. not optimized for the task. This leads to shattering of the pre-loaded weights during the first couple of training iterations, through gradient descent, explained in section 2.1.2. Thus, possibly mitigating a majority of the benefits presented by transfer learning

One solution to this problem is *freezing* the pre-loaded layers, i.e. not updating them during the first few epochs, such that the final layer(s) are the only ones trained. Then, to specialize the entire model **1** to the new task, the frozen layers are unfrozen after a few epochs.

3.1.4 Reducing generalization error

As for all other machine learning models, an optimal ANN has low generalization error, meaning that it generalizes well to unseen data. In other words, the goal is to create a model that performs well on both the training data and the test data.

To make such a model, several techniques can be used. The best alternative for reducing generalization error is to increase the amount of informative training data. Unfortunately, training data is a restricted resource, forcing us to use other techniques to reduce the generalization error (Srivastava et al., 2014). Some of them that are particularly relevant for ANNs are introduced in this section.

Dropout

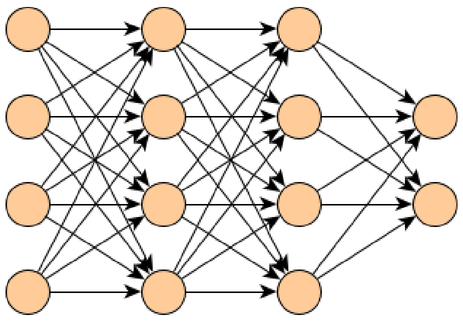
ANNs with a large number of parameters requires a large training set to prevent overfitting. When the number of parameters is too large relative to the dataset, i.e. the model is too complex compared to the data, one solution is *dropout*. Dropout refers to dropping, or rather ignoring, nodes and their outgoing edges from a neural network. The choice of which nodes to drop is made at random based on the Bernoulli distribution (of Everyday Things, 2019), using a hyperparameter **p** to specify the dropout ratio.

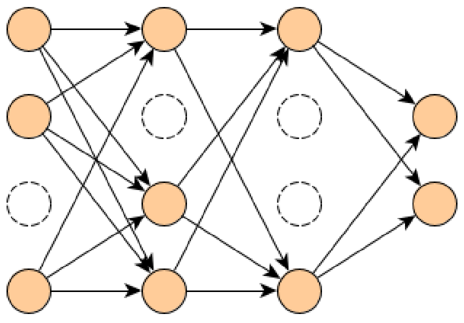
Adding dropout to a network can typically result in increased generalization ability. When nodes are ignored for one iteration, it means that they are not trained that iteration, thus updated fewer times during training. Therefore, the weights do not get updated too much to fit the training data, resulting in better generalization ability. Also, ignoring some of the nodes forces the network to learn different representations of identical inputs, making the model generalize better.

More generally, considering that *"Model combination nearly always improves the performance of machine learning methods"* (Srivastava et al., 2014), an optimal network is more often than not a combination of multiple models. However, that requires a lot of computation, data, and time. By implementing dropout, one approximates combining an exponentially large set of different neural network architectures, efficiently.

In addition to increasing the generalization ability, dropout also decreases computational needs per epoch due to the decreased number of parameters. However, it increases the time it takes for the model to converge (Srivastava, 2014).

It is worth noting that dropout is only used during training - not when running inference. This is because during inference one wants the model to have as much information as possible. There are exceptions to this, most notably the technique of using Monte Carlo dropout to obtain model uncertainty estimates. See (Gal and Ghahramani, 2016) and (Murray, 2018).


(a) Standard neural network without dropout


(b) Neural network using dropout - ignoring a random set of the nodes during training.

Data augmentation

As mentioned, the optimal way to increase a model's generalization ability is to add more training data. One way to obtain more data is to use *data augmentation*. For example, in the image domain, augmented images can be flipped, zoomed and/or rotated versions of the original images, leaving the distinct features of the image unaffected: a horizontally flipped image of a cat will still be a cat (Shorten, 2018). Such data augmentation also makes the model more invariant to transformations, e.g. scaling or rotation. Depending on the task, this can be of great use.

Early stopping

During training of an ANN the training error and validation error initially decreases. However, at some point the validation error usually starts increasing, i.e. the estimated generalization error increases, and the model starts overfitting. Typical development of error-values during model training is shown in fig. 3.4. The *early stopping* approach is to stop model training when the validation error increases, improving the model's generalization ability.

3.2 Convolutional neural networks

Creating problem-specific architectures by reducing the number of connections between the neurons is a common way to increase the generalization ability of ANNs, a widely used approach since the 1980s when dealing with images is CNNs (Géron, 2017).

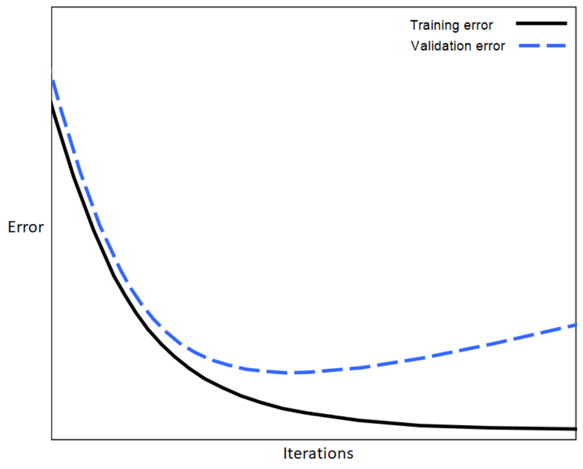


Fig. 3.4 Example of typical ANN training error and validation error development during training. By stopping the training when the validation error starts to increase typically improves the model's generalization ability.

CNNs are able to achieve superhuman performance on complex tasks in areas like image- and video processing, beating radiologists at narrow tasks (Guan et al., 2018), and making self-driving cars a reality (Bojarski et al., 2016). The key component in CNNs is the *convolutional layer*, making CNNs more suitable for image recognition than any other machine learning architecture. This component has many names: *convolution*, *kernel*, and *filter*, which we will used interchangeably. A typical CNN architecture is shown in fig. 3.5, comprised of some standard layers. These layers will be explained next, except for dropout, explained in section 3.1.4.

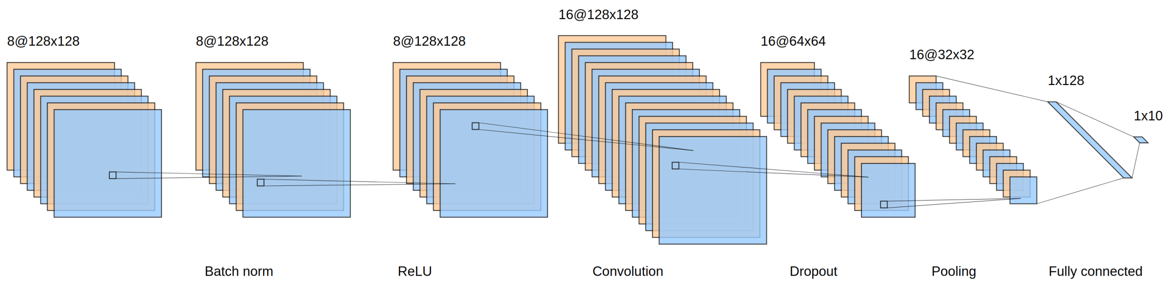


Fig. 3.5 Typical CNN architecture, inspired by the default network provided by NN-SVG (LeNail, 2019).

3.2.1 Batch normalization

In section 2.1.3 we described how one can use the hyperparameter batch size to decrease both the training time and the generalization error of a model. In (Ioffe and Szegedy, 2015) the authors obtained state-of-the-art results in classification tasks by implementing a normalization layer as part of their (CNN) model. Specifically, for each mini batch, m, with size s, the values for data point $n \in [1, ..., s]$ in the mini batch m_n is normalized according to the rest of the data points in the mini batch. So, if there are data points n_i and n_j, where i and $j \in n$, in the mini batch, with features $f_{n_i} \in [0, 1]$, and n_j with features $f_{n_j} \in [1, 256]$, these will be edited such that the variance and mean of the mini batch is 1, and 0, respectively. In the example provided in fig. 3.5 the batch size is set to 8, with 8 images of size 128×128 sent through the network.

This is done in three steps:

1. Calculate the mean and variance of the input

$$\mu_B = \frac{1}{m}\sum_{i=1}^{m} x_i \tag{3.7}$$

$$\sigma_B^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2 \tag{3.8}$$

2. Use the calculation from step 1) to normalize the inputs

$$\overline{x_i} = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \tag{3.9}$$

3. Get the output of the layer by scaling and shifting the input

$$y_i = \gamma\overline{x_i} + \beta \tag{3.10}$$

It is important to note that β and γ in eq. 3.10 are learnable model parameters, thus learned during the training process.

3.2.2 Rectified linear unit

As described in 3.1.1, the first ANNs were able to exclusively learn linear relationships, because of their lack of non-linear functions. *Rectified linear unit*, or *ReLU*, is a non-linear function, refer to eq. 3.11, giving ANNs the ability to successfully learn non-linear relationships in the data: *"Neural networks with rectified linear unit (ReLU) non-linearities have been highly successful for computer vision tasks"* (Dahl et al., 2013).

$$y = max(0, x) \tag{3.11}$$

More specifically, each activation, a_i, in the feature map, f_{a_i}, is processed using eq. 3.11. This process is visualized in fig. 3.6, showing that activations $\in \mathbb{N}_{<0}$ (blue) is substituted with 0's, while activations $\in \mathbb{N}_{>=0}$ (orange) are unedited.

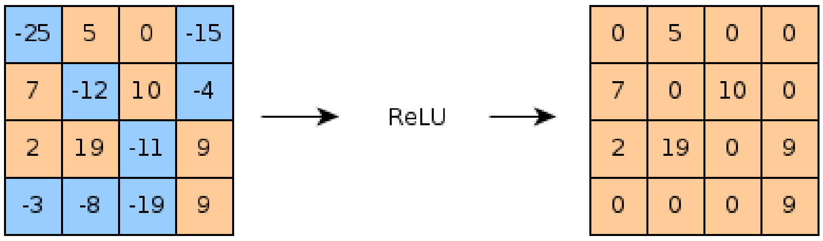


Fig. 3.6 ReLU example - the negative feature map values are mapped to 0, while the positive ones stay unchanged.

3.2.3 Convolutional layer

Eq. 3.12 describes how the output value at the spatial location $\mathbf{x}$, visualized in fig. 3.7 as 7, is calculated using a convolutional layer. It is important to note here that the input to the convolutional layer can be either the raw image, or feature map(s) already processed through the network.

$$f_{out}(\mathbf{x}) = \sum_{h=1}^{K} \sum_{w=1}^{K} f_{in}(\mathbf{p}(\mathbf{x}, h, w)) \cdot \mathbf{w}(h, w) \tag{3.12}$$

The function f_{out} takes an $\mathbf{x}$ as input, and returns a feature value. The input feature map is denoted by f_{in}. In fig. 3.7, the orange matrix represents the pixel values, or *activations*, within the $K \times K$ area with $\mathbf{x}$ as center. The green matrix is a $K \times K$ convolution, containing learnable weights. A specific visualization is provided in fig. 3.7, with a 3×3 kernel, used for point detection in images (Salem Saleh Al-amri et al., 2010).

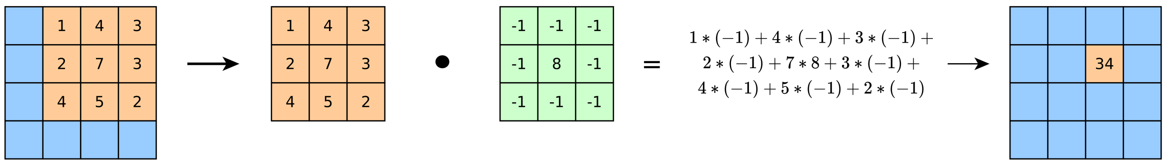


Fig. 3.7 Visualization of a 3 × 3 CNN kernel calculation. The values in the feature map to the right is a result of the dot product between the image/feature map's values (left) within the kernel size, and the kernel.

The green matrix, i.e. the filter, is randomly initialized and optimized during the training process using gradient descent and backpropagation, as described in section 3.1.2. Normally, after training a CNN the early layers will recognize low-level features such as lines and edges, while the later layers will recognize more high-level features such as eyes and ears. This optimization happens automatically during training, meaning that no manual work goes into ordering specific filters at specific positions.

The dot product of the orange and the green matrix results in a scalar called activation, which is added to the feature map. After repeating the process described above for inputs derived from every pixel in the original image, the feature map will ultimately represent the feature that the filter is trained to recognize.

3.2.4 Pooling

The complexity of neural networks is a double-edged sword: it allows them to learn complex patterns, hence solving complex problems, but it also requires a lot of computational power. With common CNN architectures comprised of several million learnable parameters, a useful technique for decreasing the complexity level is *pooling*, which reduces the size of the feature maps, i.e. the resolution. There are several different pooling techniques, but the most common are *max pooling* and *average pooling*.

The techniques are quite similar, with both having a defined pooling window of arbitrary size, which can be overlapping. An example of a non-overlapping, 2 × 2 window, max-pooling layer is visualized in fig. 3.8. The max pooling function:

$$a_j = \max_{N\times N}(a_i^{n\times n}u(n,n)) \tag{3.13}$$

computes the maximum value based on the input patch, captured by the window function $u(x, y)$ (Scherer et al., 2010).

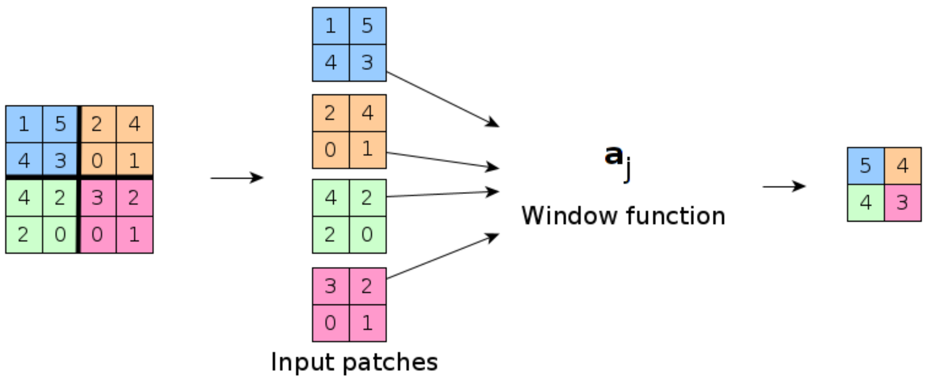


Fig. 3.8 2 × 2 max pooling example - only the largest value within the pooling window is added to the feature map.

3.2.5 Fully connected layer & output layer

Lastly in fig. 3.5 there is a 1 × 128 *fully connected layer*, outputting to a 1 × 10 *output layer*. While the already described layers are used for data processing, such as feature extraction, these layers are used to recognize patterns in these processed features - culminating in a prediction.

Fully connected layers

A fully connected layer means that each activation in the input layer, in this case, the output of the *pooling* layer, is connected to each activation in the fully connected layer, which in turn is connected to each activation on the output layer. A simplified example is visualized in fig. 3.9, where the input layer is the pooling layer, the hidden layer is the fully connected layer, and the output layer is just that: the output layer. Originally, in fig. 3.5, the $16 * 32 * 32 = 16\,384$ output activations from the pooling layer maps to 128 activations which in turn maps to 10 activations. However, for visualization purposes, these numbers are reduced to 10, 5 and 3, respectively.

Output layer

The output layer is the final layer, where one finds the model's predictions. When doing classification, more specifically single-label classification, this is usually calculated using one of two non-linear functions:

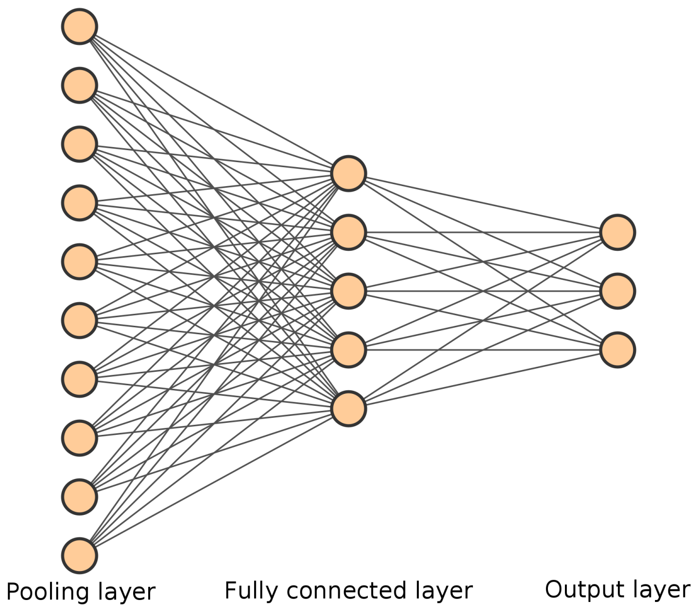


Fig. 3.9 Simplified example of fully connected layers. In a fully connected layer, each neuron is connected to every neuron in the previous layer.

1. Softmax, refer to eq. 3.5
2. Sigmoid

$$S(x) = \frac{1}{1+e^{-x}} = \frac{e^x}{e^x+1} \tag{3.14}$$

Which to use depends on the number of unique classes the prediction can belong to, i.e. whether one does multi-classification (Softmax) or binary-classification (Sigmoid). For our thesis, distinguishing between many action recognition classes, Softmax is the most relevant. This non-linear function takes in the probabilities of each unique class and uses eq. 3.5 to edit their values, such that they add up to 1. This process is visualised in fig. 3.10, where we have continued on fig. 3.9, and assigned values to the three nodes in the output layer.

3.2.6 Residual function

In addition to the aforementioned CNN components, newer architectures often implement *residual functions*.

As mentioned in section 3.2.4, the complexity of neural networks is what grants them the ability to learn complex patterns. One way of increasing their complexity, thus their ability to learn more complex patterns, is to add more layers - making them

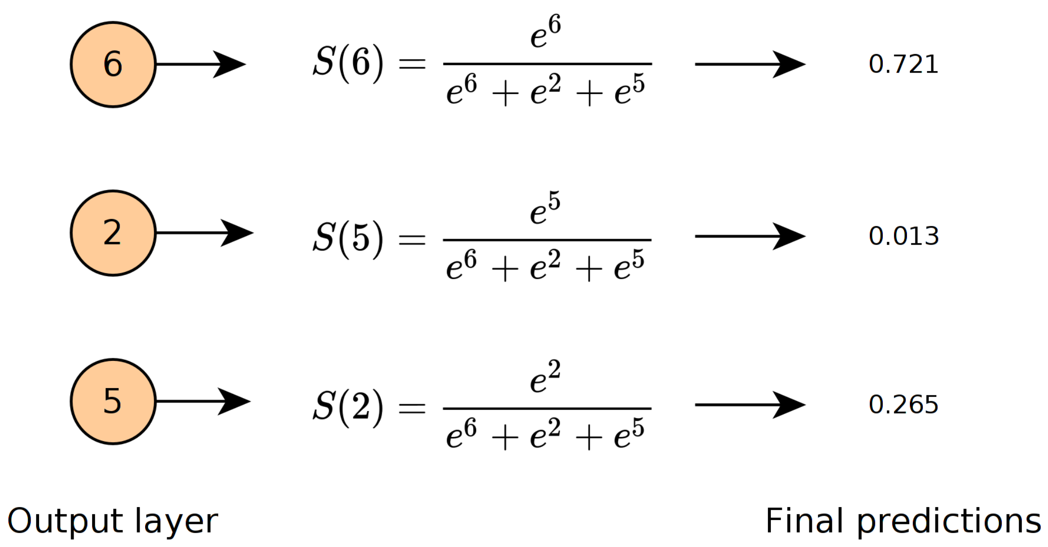

6
2
5
$S(6) = \frac{e^6}{e^6 + e^2 + e^5}$
$S(5) = \frac{e^5}{e^6 + e^2 + e^5}$
$S(2) = \frac{e^2}{e^6 + e^2 + e^5}$
0.721
0.013
0.265
Output layer
Final predictions

Fig. 3.10 Softmax calculation example. All the values in a layer is mapped to a number between 0 and 1, where all the outputs sums up to 1.

deeper. *"This makes sense, since the models should be more capable (their flexibility to adapt to any space increase because they have a bigger parameter space to explore)"* (Ruiz, 2018). However, when adding more layers the problem of *vanishing gradients* arise. As detailed in (Bengio et al., 1994), vanishing gradient describes the problem of the gradients converging towards 0 during gradient descent. When the gradient is 0, the weights will be static, never updated, thus not learning. It is this problem the adding of residual function(s) aims to solve, by adding *skip connections*, thus letting the output from earlier layers skip x layers, then adding them to the output of layer $x+1$, visualized in fig. 3.11.

The benefits of adding residual functions were empirically proven in (He et al., 2016). He et al. reported the ability to train a network 8× deeper than other then-standard CNN networks, placing first in the 2015 ImageNet large scale visual recognition challenge, classification task.

3.3 Graph neural networks

Graphs are used in many domains and systems because of their great expressive power, as important tools for visualizing and representing patterns and relationship in data. Recently, research of neural network algorithms applied to graphs, named *graph neural networks* (GNNs), has gained traction (Zhou et al., 2018). More specifically, applying

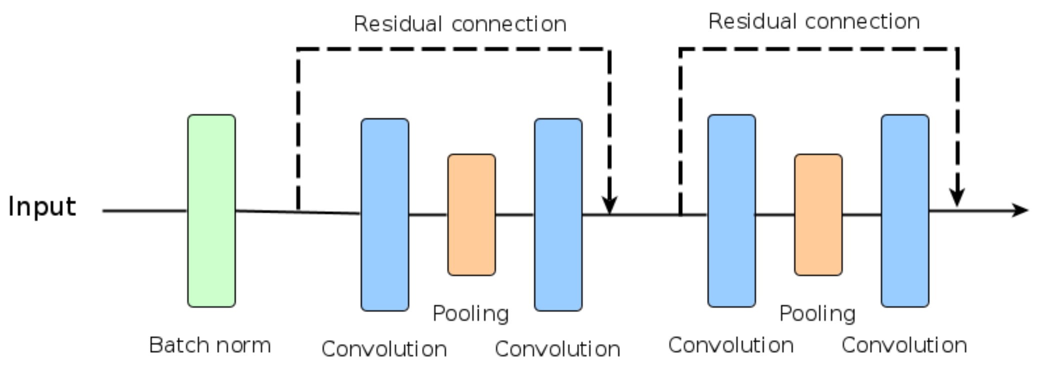


Fig. 3.11 Resnet example, inspired by (He et al., 2016). Adding *skip connections* helps to avoid the problem of vanishing gradient.

machine learning, a great learner of patterns, to a representation that reveals different patterns and relationship within the data.

3.3.1 Graphs

Graphs are widely used across several fields of science and engineering, e.g., computer vision, molecular chemistry, and pattern detection. An example of graphs used for pattern detection is within cyber security, where internet traffic can be represented as a graph. Malicious activity can be detected by finding specific activity patterns in the graph (Choudhury et al., 2015). For computer vision, graphs can, for example, be used to represent keypoints in images, relevant for action recognition and object classification.

A graph G is defined as a pair (N, E), where N is the set of *nodes* and E is the set of *edges*, as visualized in fig. 3.12 (Scarselli et al., 2009).

This structure can be applied to images where the pixels are the nodes, and the neighboring pixels are connected through edges, as seen in fig. 3.13. Note that this is for visualization purposes only, as mapping whole images to graphs is not typically useful because of inefficiency, compared to using the standard image format. However, as graphs can be used for the computer vision tasks, that we are interested in, it is important that the reader has a clear visualization of this concept.

Nodes that are directly connected by an edge to a node n are called the neighbors of n. Each node and edge represent some information about the data. The nodes usually represent features of objects, while the edges usually represent the relationship

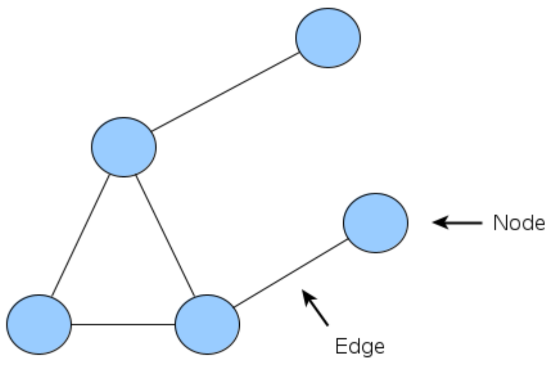


Fig. 3.12 A graph consists of nodes connected by edges. Figure inspired by Scarselli et al.

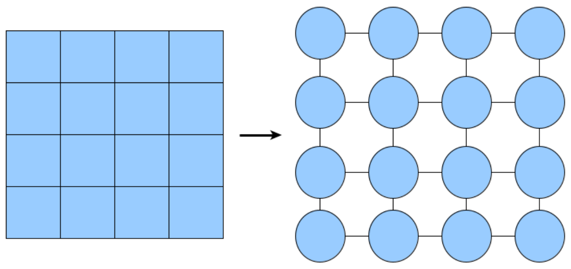


Fig. 3.13 Image represented as a graph. The nodes represent the pixels' position, while the edges represent their relationship.

between the objects. For example, in a graph representing an image, refer to fig. 3.13, the nodes can represent coordinates of different regions in the image, while the edges can represent the relative position between the coordinates, like distance and angle.

Graphs may be either positional or non-positional. In a positional graph, each node has a function that calculates its position relative to all its neighbors. Such neighbor-positions can implicitly be used for storing spatial positions, which is a key factor in graphs representing skeletons, thus vital for this thesis, and will be introduced in section 4.2.1.

3.3.2 Combining graphs & artificial neural networks

It is the aforementioned node-edge relationship that GNNs take advantage of (Gori et al., 2005). Each node n in a GNN has a state x_n, which is dependent on the information about n's neighbors. When w is a set of parameters and f_w is a parametric function that reflects the dependence of a node on its neighborhood, the state x_n can be defined as

$$x_n = f_w(l_n, x_{\mathrm{ne}[n]}, l_{\mathrm{ne}[n]}), n \in N \tag{3.15}$$

where l_n, $x_{\mathrm{ne}[n]}$ and $l_{\mathrm{ne}[n]}$ are the label of n, the states of the nodes in the neighborhood of n, and the labels of the nodes in the neighborhood of n, respectively.

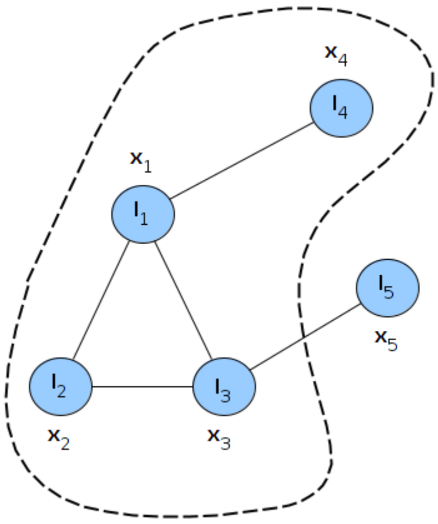


Fig. 3.14 Graph neural network architecture - state x_1 depends on the neighborhood information, inspired by Gori et al.

Each node n also has an output vector o_n, which is defined by a parametric function g_w. Eq. 3.16 takes the state x_n and the label l_n as input, such that o_n only depends on the state, and the label of the node.

$$o_n = g_w(x_n, l_n), n \in N \tag{3.16}$$

Eqs. 3.15 and 3.16 together realize a parametric function $\phi_w(G, n) = o_n$ meant for operating on graphs, such that we have a method to produce an output o_n for each node. We can now set up a machine learning problem, consisting of adjusting the parameters w to minimize the error between ϕ_w and the data in the learning set $L = \{(G_i, n_i, t_i) | 1 \leq i \leq p\}$. Each triple (G_i, n_i, t_i) denotes a graph G_i, one of its nodes n_i, and the output target t_i^2. The solution to the learning problem can then be approximated using gradient descent to for example minimize the quadratic error function shown in 3.17 (Gori et al., 2005).

$$w = \sum_{i=1}^{p} (t_i - \phi_w(G_i, n_i))^2 \tag{3.17}$$

3.3.3 Graph convolutional neural networks

As described in section 3.2, standard CNNs have a fixed grid that traverses an image with a particular step size to extract features from the image. Since the pixels in images are in spatial order, the grids are moved from left to right, top to bottom (Niepert et al., 2016b). Furthermore, the spatial order in the grids enables a natural mapping to vector space representation, as shown in fig. 3.15.

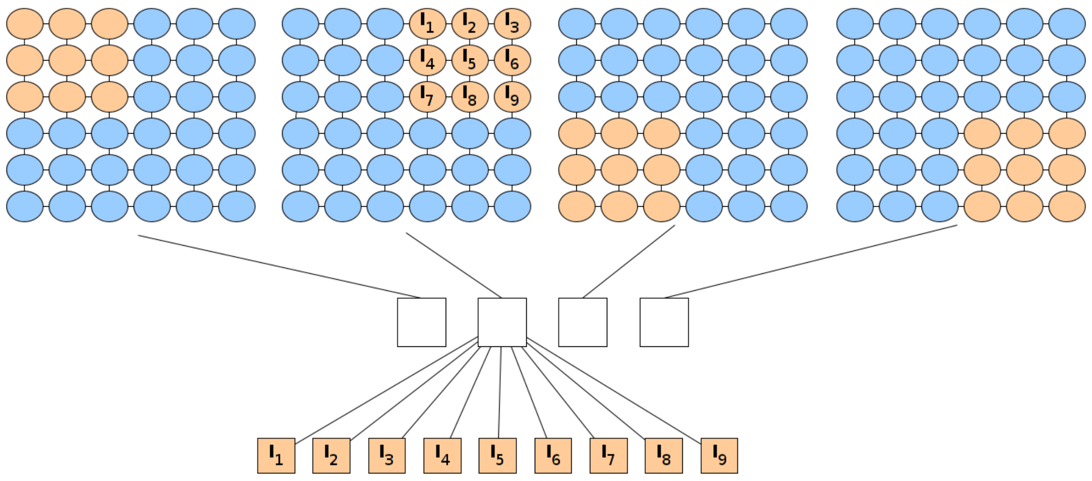


Fig. 3.15 Mapping from grid to vector in a GNN. Figure inspired by Niepert et al.

Because nodes in a graph generally have no fixed order, such as pixels in an image, there are two things that have to be determined before we can construct analogous convolutions on graphs:

1. A sequence of nodes from which neighborhoods can be created
2. A mapping from the graph domain to a vector representation

In (Niepert et al., 2016b), these two problems are addressed for arbitrary graphs. For each input graph, its nodes and their order, for which neighborhood graphs are created, are first determined. For each node, x, from which a neighborhood graph is to be created, the neighborhood N, consisting of k nodes, is extracted and normalized, comprise a $k \times 1$ vector V with labels l_k for node k $\in N$. Normalized, in this instance, means that the neighborhood nodes are mapped into a space with a fixed linear order. This normalized neighborhood then works as a grid with a natural spatial order, and can be combined with convolutional- and fully connected layers as in standard CNNs, refer to section 3.2. This method makes it possible to assign nodes from different graphs to the same relative position in their vector representations, if the graph structures are similar. Fig. 3.16 visualizes this process.

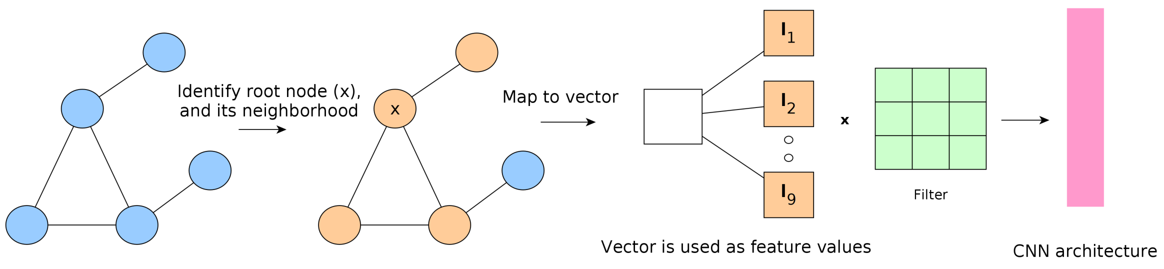

Identify root node (x), and its neighborhood
x
Map to vector
I_1
I_2
I_9
x
Filter
Vector is used as feature values
CNN architecture

Fig. 3.16 Visualization of a graph convolution. The graph is first mapped into a space with a fixed linear order, before a filter slides over the values as in standard CNNs.

Chapter 4

Human action recognition & spatial temporal graph convolutional networks

The spatial temporal graph convolutional network (ST-GCN) was presented in a paper written by Sijie Yan, Yuanjun Xiong and Dahua Lin, published in January of 2018 (Yan et al., 2018). As mentioned in the introduction, the ST-GCN serves as a starting point for our thesis.

Yan et al.'s paper describes a project designing and training a graph convolutional network (GCN) to recognize actions from video data. In short, it proves that this general approach to action recognition *"outperforms former state-of-the-art models"* (Yan et al., 2018). This is achieved by combining information about both the spatial configuration and temporal dynamics of 18 keypoints in the human body. These keypoints are mostly joints that are crucial for determining human movement, such as the shoulders and hips.

This chapter will give an overview of some approaches to human action recognition before introducing the main components in ST-GCN.

4.1 Human action recognition

Human action recognition is a popular and important research area, as it is vital for understanding actions in videos. Classifying actions from videos can be challenging, and can be done in several ways, such as *appearance*, *depth*, *optical flow*, *feature encoding*, *deep LSTM*, *temporal convolutional networks*, and *human skeletons*. Among these approaches, the human skeletons solution usually convey the most useful information for action recognition. This is due to skeletons of human bodies are more robust to noise in the data, such as illumination change and scene variation. Before jumping into the details about dynamic human skeletons, alternative methods for action recognition will be presented.

Appearance

Appearance-, or *RGB* methods, for action recognition mainly look at the appearance features in the video. For instance, if a tennis racket is seen in the video, this affects the classifier's final prediction. Appearance does not require foreground segmentation or tracking of body parts, resulting in more robustness to camera movement and low resolution (Wu et al., 2011).

A paper written by Xinxiao Wu et al. (Wu et al., 2011) describes a method for action recognition in videos using appearance, where *interest points* first are defined, and then the appearance around those interest points is considered. Such interest points are coordinates in the video where the variation of spatial-temporal intensity is high, in other words, coordinates referring to areas with lots of movement. Appearance can also be used to classify actions in static images, as presented in (Maji et al., 2011).

Depth

Exploiting depth points to classify actions can be done in several ways (Wang et al., 2012), (Vieira et al., 2012), (Chen et al., 2016), and have shown promising results. A paper written by W. Li et al. (Li et al., 2010) presents such a method for action recognition on sequences of *depth maps*.

A depth map is an image containing information about the 3D-positions of the objects in the scene. For instance, a depth map of a person will visualize the person in a different color than the background, because the distances from the camera are different. The idea in Li's paper is to extract a bag of 3D points from the depth maps, and use a Gaussian mixture model (GMM) to capture the statistical distribution of the points to classify the person's posture. This posture is then fed to an *action graph*, which classifies the action.

In this case, each node in the action graph represents a posture, where all the postures are shared by the graph's action classes. Every action is encoded in one or multiple paths in the graph. To construct the graph, training samples of depth maps are used to learn how to encode the graph and recognize all actions. One major benefit of this method is that it does not require joint tracking.

Optical flow

Optical flow is the distribution of velocity vectors that represents the movement of brightness patterns in an image. In a video, optical flow can arise from relative motion of objects between two video frames. This can convey important information about object movement in videos, making it suitable for action recognition (Horn and Schunck, 1981).

In a paper written by Karen Simonyan et al. (Simonyan and Zisserman, 2014), experiments showed that optical flow combined with CNNs achieved good performance for action recognition on videos. The input to the CNN was constructed by stacking optical flow fields generated from the videos, describing the motion between the video frames. The CNN then extracts the features from the input, before the action is classified using the softmax function described in section 3.5.

Feature encoding

Researchers has developed a video representation that captures the *video-wide temporal evolution* (VTE) of videos, used for action recognition. This method starts with considering a video $X = [x_1, x_2, ..., x_n]$, consisting of n frames, where the frame at t is represented by vector x_t. The frames $x_1, ..., x_n$ are extracted from each video, before a feature vector v_t is generated, representing the action occurring in frames x_1 to x_t. This vector is the result of a function $V(t)$, which can be based on methods such as *independent frame representation*, focusing solely on each independent frame, and *moving average*, extracting average behavior within a fixed temporal window.

Afterwards, a video representation $\mathbf{u}$ is learned for each video by using *ranking machines*, i.e. the usage of machine learning algorithms to rank models that are meant to retrieve information (Mohri et al., 2012). Lastly, these video specific $\mathbf{u}$ representations are used as a representation for action classification, which finally can be classified using for example a SVM classifier, described in 2.2.3 (Fernando et al., 2015).

Long short-term memory networks

A human body pose can be represented by an ensemble of 3D coordinates of body joints. Thus, human movements can be represented by time series of such coordinates. Furthermore, as mentioned in section 2.2.4, RNNs have proven to be among the best ML models for learning sequential data (Yin et al., 2017).

The main component in RNNs today is a *long short-term memory* (LSTM) unit, first presented in (Hochreiter and Schmidhuber, 1997), expanding RNNs' abilities to solve complex tasks. (Shahroudy et al., 2016a) uses such a (deep) LSTM-based RNN model to identify the correlation between human joints over time by feeding the model with time series of body joints coordinates, classifying the time series into an action.

Temporal convolutional neural networks

Another approach for 3D human action recognition is *temporal convolutional neural networks* (TCN), described in (Kim and Reiter, 2017). The input to a standard TCN is a matrix representing features in a video. For each frame in a video, a D-dimensional feature vector is extracted, for example using a CNN, such that for a video of T frames, the input X is a concatenation of all the feature vectors from the frames: $X \in R^{T \times D}$. By sending this feature matrix through a network consisting of residual blocks, explained in section 3.2.6, a final prediction can be made using an activation function, relevant for the specific task. Their approach achieved state-of-the-art results on the largest 3D human action recognition dataset, NTU-RGB+D.

Dynamic human skeletons

Over the past years, several methods for extracting human skeletons from 2D videos have been presented. The methods have largely focused on detecting body parts on individuals (Felzenszwalb and Huttenlocher, 2005). In 2018, *OpenPose* presented a new approach to efficiently detect the 2D pose of people in an image. The architecture secures high accuracy on constructing the skeletons and provides the ability to achieve real-time performance, independent of the number of people in the image (Cao et al., 2018).

OpenPose is based on the deep learning software Caffe developed by Berkeley AI Research (Jia et al., 2014). Caffe was an effort from then Ph.D. student Yangqing Jia, now a research scientist at Google, to create an open source framework simplifying efforts to recreate state-of-the-art deep learning results such as Alex Krizhevsky's AlexNet from 2012 (Krizhevsky et al., 2012). Jia was motivated by the lack of such frameworks in the deep learning domain at that time (Alliance, 2016).

The method used in OpenPose starts with taking a collection of images as the input, before feeding them to a CNN which gives *confidence maps* for body part detection, and *part affinity fields* for parts association. Part affinity fields are 2D vector fields between two parts, that indicates in which degree the parts are associated. Finally, after the confidence maps and affinity fields are parsed for all images in the collection, both the 2D coordinates (X, Y) in the image's pixel grid, and the confidence score C, for each of the detected body parts for all people in the image, are returned as the output.

OpenPose offers various models, trained on various datasets, able to extract different amounts of keypoints from single images. One option is a model trained on the "COCO keypoints dataset" (Cao et al., 2018), extracting 18 keypoints, i.e. (X, Y) coordinates for 18 body keypoints. It is this model, visualized in fig. 4.1, Yan et al. used for action recognition on the Kinetics dataset. How Yan et al. use these keypoints to classify actions will be explained in the next section.

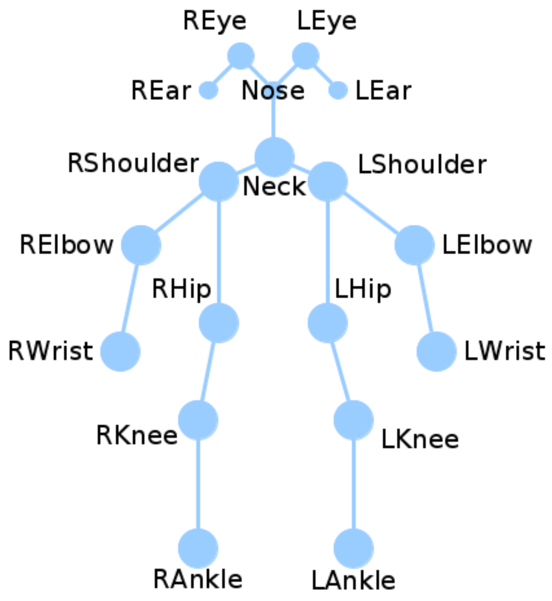


Fig. 4.1 Visualization of the keypoints extracted by OpenPose's COCO model, inspired by Cao et al.

4.2 ST-GCN: The spatial temporal graph convolutional network

Yan et al.'s uses the 18 keypoints, extracted using OpenPose, and exploits the fact that human joints move in small, local groups within the skeleton, known as body parts. Adding this restriction to action classifier models causes a large improvement because the joints are then constrained to move within the local area and not the whole skeleton. For example, it is generally problematic for a human to move their elbow more than one meter away from their shoulder.

This section will give an overview of the main components in ST-GCN, including details about the network, such as its construction and architecture.

4.2.1 3D skeleton graph construction

The skeleton graph is constructed as an *undirected spatial temporal graph*: $G = (V, E)$, on a skeleton sequence with N keypoints and T frames. The graph contains the node set $V = \{v_{ti} | t = 1, ..., T, i = 1, ..., N\}$, which includes every joint in a skeleton sequence. Furthermore, each node have a feature vector consisting of both coordinate vectors, and estimation confidence of the i-th joint on frame t.

The graph is constructed on the skeleton sequences in two steps:

1. Connecting each node within one frame to each other, directly or indirectly, following the structure of the human body, resulting in a spatial connection
2. Connect each joint within one frame to the same joint in the following frame, resulting in a temporal connection

The outcome is a 3D graph representing a skeleton's movement over consecutive frames, as illustrated in fig. 4.2, with the blue edges representing the spatial dimension, and the orange edges representing the temporal one.

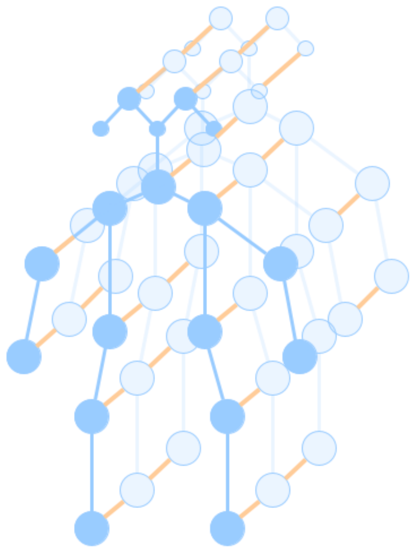


Fig. 4.2 3D graph representing a skeleton's movement over time, inspired by Yan et al. The blue edges represent the spatial connection between nodes within a skeleton, while the orange edges represent the temporal connections between the skeletons.

4.2.2 The spatial graph convolutional neural network

As presented in section 3.2, a convolution in a standard CNN can be expressed as

$$f_{out}(\mathbf{x}) = \sum_{h=1}^{K} \sum_{w=1}^{K} f_{in}(\mathbf{p}(\mathbf{x}, h, w)) \cdot \mathbf{w}(h, w) \tag{4.1}$$

where $\mathbf{p}$ is the sampling function enumerating the neighbors of $\mathbf{x}$, and $\mathbf{w}$ is the weight function providing the kernel. To use the same principle on graphs, the sampling function and the weight function need modifications.

Sampling function on graphs

A sampling function is a function that enumerates through a set of items, which for images are pixels. As stated in section 3.3.1, graphs consists of nodes and edges instead of grids of pixels, which leads to the definition of a new sampling function, adapted to graphs. This sampling function can be defined as

$$B(v_{ti}) = \{v_{ti} | d(v_{tj}, v_{ti}) \leq D\} \tag{4.2}$$

where $B(v_{ti})$ is the neighbor set of node v_{ti}, and $d(v_{tj}, v_{ti})$ is the minimum length from v_{tj} to v_{ti}. Thus, the sampling function can be written as

$$\mathbf{p}(v_{ti}, v_{tj}) = v_{tj} \tag{4.3}$$

In Yan et al.'s paper, D is set to 1, meaning that the neighbor set $B(v_{ti})$ only consists of the nodes that have 1 edge between them and the node v_{ti}, visualized as the blue nodes with the orange node as root in fig. 4.3.

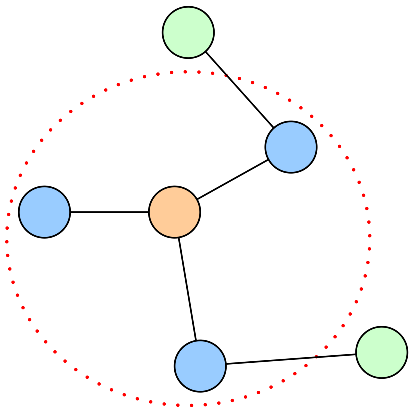


Fig. 4.3 Visualizing an example of labeled nodes when number of neighbors is set to 1. The root node (orange) is connected to its neighbors (blue) when D = 1.

Using this method may result in neighbor sets with a different number of nodes, unlike convolutions on images where the sampling function has a constant number of pixels in the grid. This introduces some challenges when the weight function is defined.

Weight function on graphs

Defining a weight function for 3D-graphs is more tricky than on images because a neighbor set has no natural arrangement like pixels in grids. One solution to this problem, introduced in (Niepert et al., 2016a), is to label all the nodes in the neighbor set around the root node, which Yan et al. used. However, instead of assigning a unique label to every node in the neighbor set, they partitioned the neighbor set $B(v_{ti})$ of a node v_{ti} into a fixed number of K subsets, such that each node can be mapped to its subset label. The weight function can then be implemented as in eq. 4.4, where $l_{ti}(v_{tj})$ is the mapping function that maps a node in the neighborhood to its subset label.

$$\mathbf{w}(v_{ti}, v_{tj}) = \mathbf{w}'(l_{ti}(v_{tj})) \tag{4.4}$$

Spatial graph convolution

Eq. 4.5 says that for each node v_{tj} in the neighborhood $B(v_{ti})$, calculate the dot product between the result from the sampling function and the result from the weight function, and normalize the output with $Z_{ti}(v_{tj})$, where $Z_{ti}(v_{tj})$ equals the number of nodes in the corresponding subset, added to balance the impact of different sized subsets to the output.

$$f_{out}(v_{ti}) = \sum_{v_{tj} \in B(v_{ti})} \frac{1}{Z_{ti}(v_{tj})} f_{in}(\mathbf{p}(v_{ti}, v_{tj})) \cdot \mathbf{w}(v_{ti}, v_{tj}) \tag{4.5}$$

To modify the equation such that it works on graphs, the sampling function $\mathbf{p}$ and the weight function $\mathbf{w}$ have to be replaced with $\mathbf{p}$ shown in eq. 4.3 and $\mathbf{w}$ shown in eq. 4.4. This finally results in a general equation for convolutions on spatial graphs:

$$f_{out}(v_{ti}) = \sum_{v_{tj} \in B(v_{ti})} \frac{1}{Z_{ti}(v_{tj})} f_{in}(v_{tj}) \cdot \mathbf{w}(l_{ti}(v_{tj})) \tag{4.6}$$

4.2.3 Subset partitioning

When designing a partitioning strategy, one has to design the mapping function l, making sure that it is possible to implement. There are several ways to split the neighbor set of a node into partitions. Some of them will be introduced in this subsection.

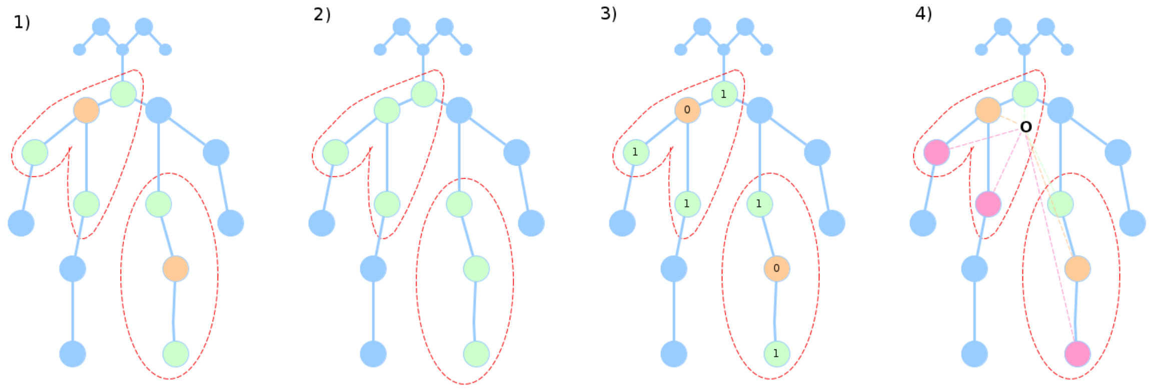


Fig. 4.4 Visualization of different skeleton partition strategies, inspired by Yan et al. Body joints are drawn with blue dots. Neighborhood for node n, with size $= 1$ is drawn with red, dashed circles. 1) Example of a skeleton and some neighborhoods, with the orange nodes as root nodes and the green nodes as neighbors 2) Uni-labeling 3) Distance partitioning 4) Spatial configuration partitioning.

Uni-labeling

All nodes within the same subset has the same label, refer to 2) in fig. 4.4.

Distance partitioning

Every subset has a root node with distance 0, while all the other neighbors in the subset have distance 1, refer to 3) in fig. 4.4.

Spatial configuration partitioning

In *spatial configuration partitioning*, refer to 4) in fig. 4.4, the nodes are labeled with respect to their distances from the skeleton gravity center, shown as a black circle, which is the average coordinate of all keypoints in the skeleton at a frame. This partitioning strategy divides the neighbor set into three subsets:

1. The root node, colored orange
2. The centripetal group, consisting of the nodes that are closer to the gravity center than the root node, colored green
3. The centrifugal group, consisting of the nodes that have a longer distance to the gravity center than the root node, colored pink

This strategy is inspired by the fact that motions of body parts can be broadly categorized as concentric and eccentric motions, or in other words, body parts moving towards (concentric) or away (eccentric) from the skeleton gravity center. Experiments performed by Yan et al. showed that this strategy achieves the best performance, and it is the partition strategy used in the ST-GCN network (Yan et al., 2018).

4.2.4 Network architecture & training

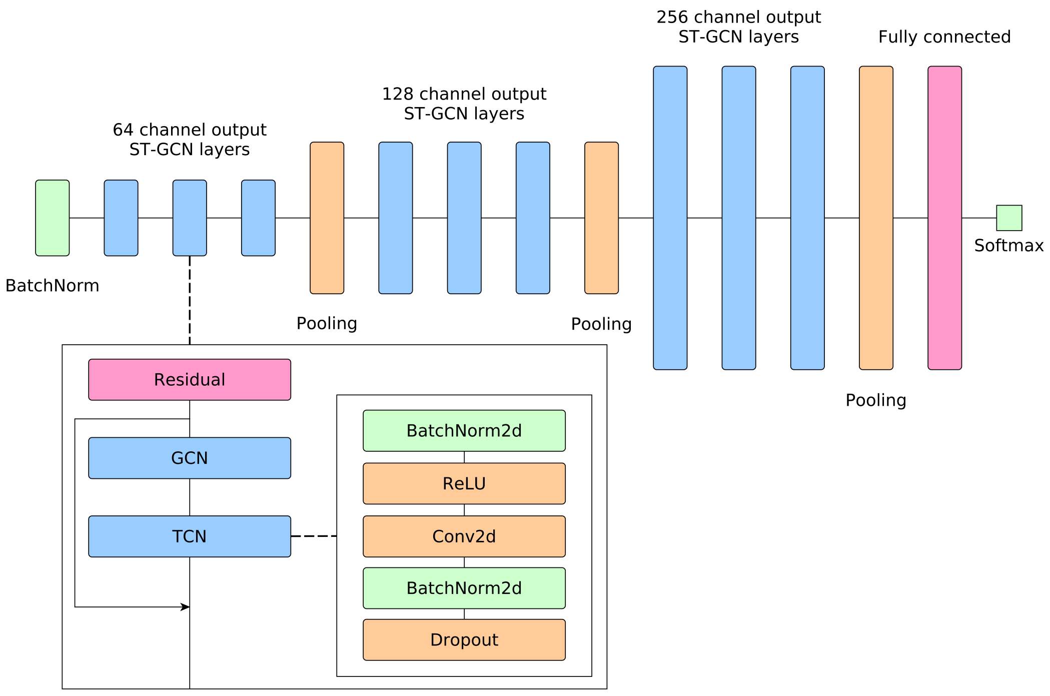


Fig. 4.5 ST-GCN architecture as interpreted from Yan et al. The network consists of batch normalization layers, ST-GCN layers, pooling layers, fully connected layers, and finally the Softmax function. Each ST-GCN layer uses a residual function. Also, the TCN block consists of a batch normalization 2D layer, ReLu, convolutional 2D layer, another batch normalization 2D layer, and uses dropout.

The majority of the layers making up ST-GCN has been explained in section 3.2. The first layer in ST-GCN is a batch normalization layer, followed by 9 layers of *spatial temporal graph convolution operators*, called *ST-GCN layers*. Among these nine ST-GCN layers, the initial three have 64 channels as output, the three next layers have 128 channels as output, and the final three layers have 256 channels as output. All these layers have kernel size 9.

To avoid overfitting, dropout with 0.5 probability is sometimes used after each ST-GCN layer. The 4th and the 7th layers have strides set to stride 2, to work as pooling layers. The resulting tensor is sent through a global pooling layer, outputting a 256 dimension feature vector for each sequence. These tensors are finally fed to a fully connected layer, connected to a softmax classifier, outputting a $1 \times C$ vector consisting of the probabilities for each class, where C is equal to the number of unique classes in the dataset.

The paper describes using two large datasets for network training:

1. Kinetics - assembled by DeepMind, consisting of approximately 300 000 raw video clips taken from YouTube. It covers 400 different human action classes with 400-1150 video clips for each action class. Each clip lasts around 10 seconds and is labeled with a single class (Kay et al., 2017).

2. NTU RGB+D - assembled by the ROSE Lab at the Nanyang Technological University, Singapore. The dataset covers 60 labeled action classes and consists of 56 880 action samples. In addition to the RGB video, each sample also contains depth map sequences, 3D skeletal data, and infrared videos (Shahroudy et al., 2016b).

The network uses *mini-batch gradient descent*, which is a variant of gradient descent explained in 2.1.2. Here, the model's weights are updated after calculating the gradient for each batch, unlike for all the data points in the training set at once. Cross-entropy loss, described in section 3.1.2, is used as the loss function during training. This can be seen in their code:

```
import torch.nn as nn
self.loss = nn.CrossEntropyLoss()
```

Furthermore, it has a base learning rate 0.01, which is decayed by 0.1 after every 10th epoch, starting from epoch 20. The change in learning rate is defined as *steps*:

```
lr = self.arg.base_lr *
       (0.1**np.sum(
       self.meta_info['epoch']>= np.array(self.arg.step)))
```

To reduce the risk of overfitting, two different augmentation techniques are used to replace dropout layers when training on the Kinetics dataset. The first technique, named *random moving*, simulates the camera movement, resulting in the skeleton getting different joint coordinates than the original video. The second augmentation technique takes random fragments from the original skeleton sequence and uses these to train the model, similar to dropout 3.1.4.

4.2.5 Experiments & results from the ST-GCN paper

The results of Yan et al.'s ST-GCN model was compared with other state-of-the-art methods and other input modalities, using the deep learning framework PyTorch (Paszke et al., 2017) and 8 TITANX GPUs (Yan et al., 2018). On the Kinetics dataset, ST-GCN was compared with three different approaches for skeleton based action recognition:

- Feature encoding approach on hand-crafted features (Fernando et al., 2015).
- Deep LSTM (Shahroudy et al., 2016c).
- Temporal ConvNet (Soo Kim and Reiter, 2017).

The comparison was based on recognition performance, namely accuracy for both top-1 and top-5 predictions. Results provided by Yan et al., shown in table 4.1, show that ST-GCN outperforms the previous skeleton-based models. However, frame-based methods such as RGB and optical flow, see sections 4.1 and 4.1 respectively, still outperform ST-GCN on this dataset.

	Top-1 (%)	Top-5 (%)
RGB	57.0	77.3
Optical Flow	49.5	71.9
Feature Encoding	14.9	25.8
Deep LSTM	16.4	35.3
Temporal ConvNet	20.3	40.0
ST-GCN	**30.7**	**52.8**

Table 4.1 Performance for human action recognition models on the Kinetics dataset. Frame-based models are listed in the top row, while skeleton based models are listed in the following rows

4.2.6 Conclusion

On the Kinetics dataset, it turns out that video frame-based models like RGB frames and optical flow are superior to ST-GCN. The argument for this is that a large amount of action classes in this dataset requires recognizing the objects and scenes that the people in the video are interacting with 6.1, i.e. person-object actions. To verify this, a subset of 30 classes related to body motions, independent of the surroundings, named as "Kinetics-Motion", was tested against the frame-based models. The results, shown in table 4.2, verifies that the difference in performance decreases.

Method	RGB CNN	Flow CNN	ST-GCN
Accuracy (%)	70.4	72.8	72.4

Table 4.2 Top 3 human action recognition approaches tested on the "Kinetics motion" subset of the Kinetics dataset

The conclusion is that ST-GCN can recognize actions from dynamic skeleton sequences which is complementary to RGB modality. Further, ST-GCN is also very flexible, which opens up many possibilities for future work. As the authors point out, one natural question to answer then becomes how to include contextual information, such as scenes, objects, and interactions.

Chapter 5

Robotics and computer vision

Robotics and *computer vision* are technologies under constant development and rapid change, as they are key components in advanced systems that continually make existing products obsolete.

Robotics is defined as the study of robots (May, 2017). Modern robots comprise several components touching a number of different fields, such as mechanics, electronics, thermodynamics and power conversion technology, computer and information technology, and materials science (Angelo, 2007). Combining all these fields into a robot requires experts from the different domains to cooperate, making robotics a complex field, especially considering today's advanced robots (Asano et al., 2019) (Asano et al., 2017) (Stories, 2019).

Computer vision is *"the transformation of data from a still or video camera into either a new representation or a decision"* (Bradski and Kaehler, 2008). In this case, a new representation might mean converting a video of a person into a skeleton sequence, while a decision might be concluding whether or not there is a person in the video. This way, computer vision gives robots the ability to see and feel the environment surrounding them by using cameras, sensors, and computer vision algorithms.

The number of robots taking advantage of computer vision has increased significantly over the last years, with this combination being used in many booming fields, such as the automobile industry, space robots, and surgical robots (Pugh, 2013). To be able to create relevant robots efficiently, one needs tools that simplify the development process. One tool for such tasks is the *Robot Operating System* (ROS), which is compatible with a large number of today's modern robots. This chapter will focus on some relevant examples and the main concepts of ROS.

5.1 Modern robot applications

This section will give an overview of the present status in the field of robotics, presenting some state-of-the-art robot applications within different domains.

Self-driving cars

The car domain is currently converging towards robotics, in the sense that cars are becoming more and more autonomous, equipped with cameras and sensors capturing data about the environment. An important component in autonomous cars is computer vision. Image streams from the cameras are sent through computer vision algorithms in real-time to give the car the ability to act based on what it sees.

Today, all Tesla cars have an autopilot function which can steer your vehicle through highway interchanges and exits based on the destination (Tesla, 2019). This still requires the driver to focus on the road and take action when it is needed. However, Tesla claims that the hardware available in their cars is sufficient to drive fully autonomously, such that the cars can become self-driving through software updates in the future. Fully self-driving cars can pick the user up and drive to their destination while they are taking a nap. Tesla's CEO, Elon Musk, claims that fully self-driving cars will be on the market in 2020 (wire.com, 2019), making them relevant to launch a driverless taxi service (Wiggers, 2019).

Waymo is an American company also aiming for creating fully self-driving cars (Waymo, 2019). Since their beginning in 2009, starting as the "Google self-driving car project", they have worked with developing self-driving cars. In 2019 they officially released a commercial autonomous taxi service in the USA, where you can order an autonomous taxi using their app. However, the service is in its early phases, and the wait list to ride with a Waymo car is relatively long (theverge.com, 2019).

It is important to note that a significant amount of highly knowledgeable experts, such as John Krafcik (CEO of Waymo), Rodney Brooks (founder of Rethink robotics), and many more, claim that fully autonomous cars, able to drive on regular streets and in all conditions, are a long way off (TIBKEN, 2018) (Brooks, 2018). With that said, the major advances made in the field over the recent years are still extremely impressive, and will likely at least lead to useful new technology in cars, and some form of restricted autonomy for cars and other vehicles.

Computer vision in space

Computer vision is not only applicable here on Earth - it is also highly relevant for space exploration, shown by its use on the Mars Rover project. Furthermore, *"Increasing the level of spacecraft autonomy is essential for broadening the reach of solar system exploration. Computer vision has, and will continue to, play an important role in increasing autonomy of both spacecraft and Earth-based robotic vehicles"* (Matthies et al., 2007). This is motivated by the problems of communication latency and bandwidth limitations, which Matthies et al. describe as severely constraining humans' ability to remotely control robot functions.

Surgical robots using computer vision

Robotics has become an important and established part of clinical surgery (Rosen et al., 2010). The motivation behind using surgical robots is to increase the effectiveness of surgical procedures. In fact, the US has identified three top application areas in their health care system that will benefit the most, economically, by new technology, by 2026:

1. Robot Assisted Surgery ($40B).
2. Virtual Nursing Assistant ($20B).
3. Administrative Workflow Assistance ($18B).

(Matt Collier, 2017)

A surgical robot often needs sensors that enable it to adapt to its rapidly changing environment. Such sensors might be force sensors and computer vision systems, giving the robot a natural transition into the human senses of touch and sight, critical abilities for making robots relevant as a surgical assistant.

Computer vision on such robots can be used on images from different sensors, like ultrasound, spectroscopy, and optical coherence tomography (OCT). Robots capable of seeing subsurface structures provides major benefits when it comes to, for instance, resecting a brain tumor. *"This type of sensing can alert the surgeon before he or she accidentally cuts a major vessel that is obscured by the tumor"* (Kazanzides et al., 2008).

5.2 ROS: the Robot Operating System

ROS is a state-of-the-art, open source framework for writing robot software, consisting of tools, libraries, and conventions that facilitate the programming of robot behavior (ROS.org, 2019a). It is designed to address the challenge of having a heterogeneous ecosystem, i.e. systems consisting of different robots, operating in distinct programming languages. The framework builds on a message system that sends messages between nodes via topics, using the *publish/subscribe* pattern (Eugster et al., 2003), often abbreviated to *pub/sub*. This section will introduce ROS and discuss some of its main components.

5.2.1 Nodes & topics

Normally, a robot consists of many nodes, described as *"a process that performs computation"* (ROS.org, 2019b). For example, one node can control the wheels' motors, another node can control the navigation, while a third node controls the laser sensor(s). The communication between these nodes is facilitated by the use of topics, giving the nodes the ability to exchange messages asynchronously, visualized in fig. 5.1.

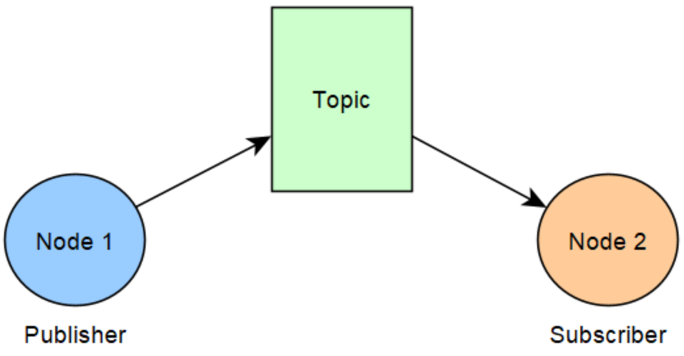


Fig. 5.1 Publish-subscribe mechanism for message exchange in ROS. Nodes can communicate with each other by publishing and subscribing to the same topic.

Topics are *named message buses* that nodes can publish or subscribe to (ROS.org, 2019c). When two nodes communicate, they don't know which node they are communicating with. Instead, nodes that need data subscribe to a topic that provides relevant data. Concurrently, nodes that generate data can publish to a relevant topic, making it visible for other nodes in the ecosystem.

5.2.2 Message types

The data type of a topic is determined by the publishing node. If a node publishes messages of type string to a topic, then a subscribing node can only receive string messages. ROS provides a large number of different message types divided into categories. One category is *sensor_msgs*, containing message types such as *sensor_msgs/LaserScan*, where one message represents a single scan from a planar laser range-finder, and *sensor_msgs/Image*, where one message represents one image.

By sending *sensor_msgs/Image* messages over a topic, a robot and a computer can exchange images or videos between each other by publishing and subscribing to the same topic.

However, ROS is used in our project because of their simple pub/sub message exchanging pattern, but also because it offers a wide range of packages and great simulation tools.

Packages are software that provides some specific functionality. For example, if you want to create a map with a robot by driving it around, ROS has a package for that, and if you want to control your robot with a joystick, ROS has a package for that also. Furthermore, ROS has great simulation tools to test the robot's behavior, such as *Rviz* (Kam et al., 2015) and *Gazebo* (Koenig and Howard, 2004). Gazebo provides the user with the ability to add physical constraints to the virtual environment, making the simulations more realistic, while Rviz is used to visualize how the robot is interpreting its sensor data.

5.3 ROS alternatives

The robotics community offers alternatives to ROS for creating robot software, such as *Yet another robot platform* (YARP), *Rock* and *Open robotics control software* (OROCOS). Similar to ROS, these alternatives are open source software for creating robot applications. They all have a high focus on simplifying message sending between different components within a robotic system.

5.3.1 YARP: Yet another robot platform

YARP is a library and toolkit for communication between devices used on everything from humanoid robots to small embedded devices (YARP, 2019). Its way of creating robot control systems is building them as a collection of programs, which can communicate via common connection types such as TCP, UDP, XML, etc. The creators' main goal for YARP is to increase the lifetime of robot software projects.

5.3.2 Rock

Rock is a software framework created by "DFKI robotics innovation center" in the German research center for artificial intelligence, used to develop robotic systems (Rock, 2019). It provides a collection of ready-to-use packages and was created to specifically address issues such as

- Sustainability - by focusing on error detection, reporting, and handling
- Scalability - by giving the tools the ability to manage big systems
- Reusability - by making most of the functionality totally independent from Rock's integration framework, such that the algorithms can be used in different frameworks

Actually, Rock can cooperate with ROS by exchanging data between Rock ports and ROS topics.

5.3.3 OROCOS: Open robotics control software

OROCOS is a free software project for robot development (OROCOS, 2019). The project combines both component-based and object-oriented reusability strategies. It provides a component-based infrastructure and a library of ready-to-use components, making it possible to manage interactions within an application at a high level. Similar to ROS, OROCOS also uses an (anonymous) pub/sub message system through data-flow ports, which are comparable to topics in ROS (Khemaissia, 2013).

Part II

Experiments

The following part lays out the experiments done to answer our hypothesis

By combining state-of-the-art techniques from computer vision and robotics it is possible to construct robots capable of accurate, real-time human action recognition, making it feasible to deploy such systems as robotic physical therapists.

A challenge when creating a system based on computer vision and robotics is the rapid, ongoing change in the technology. *"The pace of technological development poses the end user with the perpetual headache of trying to decide what is worth using and how to use it"* (Eason, 2014). In each field, techniques and solutions quickly become obsolete, and that's particularly true for techniques in the two fields' intersection. This has led to a general lack of up-to-date and useful documentation, with a plethora of choices among often difficult to compare techniques and approaches.

Chapter 6

Retraining ST-GCN to increase its relevance for rehabilitation

This chapter lays out the steps necessary to develop a system combining ST-GCN and ROS, such that it can function as a supplement in the rehabilitation process. Given the complex structure ST-GCN is built on, modifications such as extracting relevant action classes from the Kinetics dataset or adding new classes are challenging. However, we have identified some solutions and will try to explain our process in an understandable way.

6.1 The Kinetics human action video dataset

As explained in section 4.2.4, our action recognition DL model is trained on the Kinetics human action video dataset, referred to as Kinetics from now on. The 400 actions in Kinetics include: singular person actions, e.g. clapping, laughing, shaking head; person-person actions, e.g. hugging, shaking hands; and, person-object actions, e.g. playing tennis, baking cookies, drinking beer (Kay et al., 2017).

6.1.1 How the dataset was built

Initially, DeepMind identified relevant action classes from previously created action datasets. Next, the data for each of the defined action classes were generated by matching titles of YouTube videos with the action names. These went through two quality checks. First, they used image classifiers outputting the videos with a position in the form of a time interval in every video where one of the actions potentially occurred. Further quality check was crowdsourced using Amazon's "Mechanical turk" (Buhrmester et al., 2011), by creating a simple web app and exposing it to willing labelers.

6.1.2 Data distribution

A human might understand which action is taking place in a video filmed from a first-person perspective. For example, if a person is doing paragliding, then another person can recognize the action by seeing what the person performing the action sees. However, this is not possible for a model that requires skeleton data as input to classify actions. For such a model to be successful at human action recognition, the data needs to contain explicit informative signals, i.e. easily visible humans performing actions.

After looking through some of the videos in the Kinetics dataset one discovers that some videos are recorded from a first-person perspective, providing no or very little skeleton-data to our deep learning model. To investigate this further one can take a look at the amount of skeleton-data the pose estimator is able to extract from each video in Kinetics.

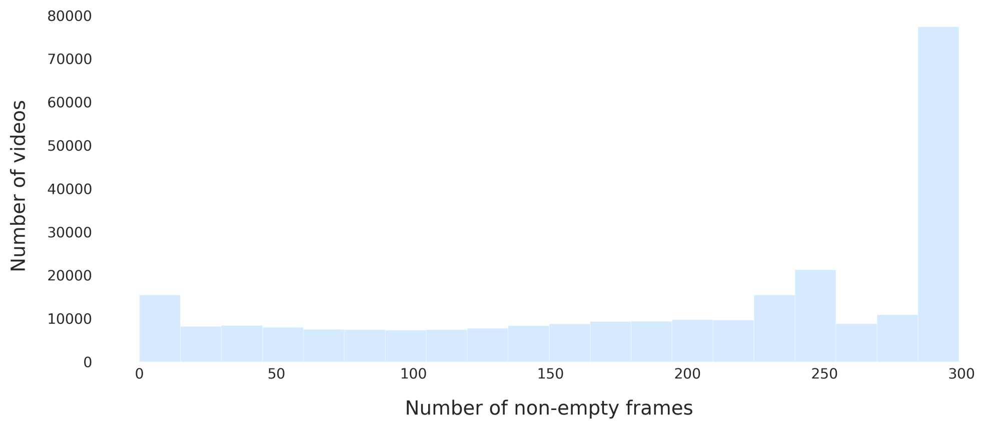


Fig. 6.1 Distribution of the number of non-empty frames in the videos in the original dataset. The left side of the plot shows that there is a significant amount of data with a notable amount of empty frames.

Fig. 6.1 shows the distribution of the number of skeleton-frames in the videos in Kinetics. There is a significant amount of data with a lot of empty frames. Yan et al. implemented a way to filter out the (completely) empty videos by adding the metadata information “has_skeleton”, which we will come back to in section 6.2.2. However, one can argue that a human requires in the excess of 1 second of visual input to categorize a human action. This is naturally dependent on the complexity of the movement. Now, using videos with a frame rate of 30 fps, this means approximately 30 frames, which means that a case can be made that a significant amount of the videos contains too little skeleton data, even for a human-level skeleton-based recognition system.

6.2 From videos to machine learning training

The ST-GCN project used the pose estimation software *OpenPose* to extract skeletons from the videos in Kinetics. The returned JSON files for each video frame are then combined into a single file, called a *skeleton file*, together with meta data information about the video. Furthermore, they construct a corresponding *summary file*, and a *label file*. This process is illustrated in fig. 6.2. Lastly, the skeleton files and the summary file is used to generate four new files, referred to as *training*- and *validation data*.

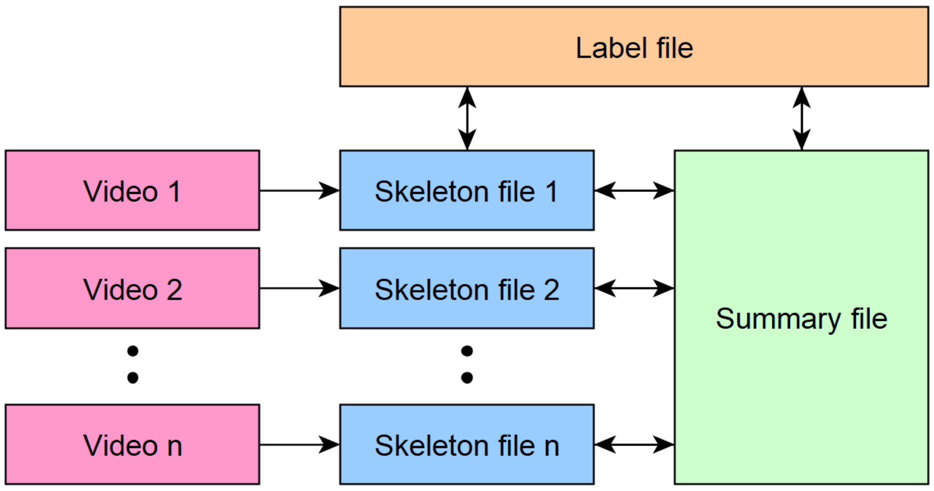


Fig. 6.2 From video to skeleton files. Each video is mapped to a skeleton file which is related to the summary file and the label file.

6.2.1 Label file

The label file is a TXT file containing the name of each class in the dataset. Each class name, or label, is separated by a new line, such that one can use their line number to identify them, i.e. each label has a unique index number. More formally, each class c has a corresponding index number i, such that the label file l can be defined as

$l = [c_i, ..., c_n]$, where n is equal to the number of classes. Here is a print-out of the first five lines from the label file:

```
abseiling
air drumming
answering questions
applauding
applying cream
```

6.2.2 Summary file

The summary file is a JSON file containing metadata for the skeleton files. For each skeleton file, there is a key-value entry in the summary file, where the skeleton file's name corresponds to the summary file's key. Further, the values correspond to the skeleton files metadata. An example entry:

```
{
    "---QUuC4vJs": {
        "has_skeleton": true,
        "label": "testifying",
        "label_index": 354
    },
    ...
}
```

Note the sub key "has_skeleton" - this is a boolean value, that is "false" if all of the frames in the video are empty, i.e. there are no humans present in the video, else it is "true". Its value is used to ignore the files that contains no skeleton information when generating training and validation data.

6.2.3 Skeleton file

As mentioned, the skeleton files contain the data from each frame in the video, and the meta data about the video. Continuing with the example entry from 6.2.2, here is an illustration of the corresponding skeleton file:

```
{ "data":
       [{"frame_index": 1,
         "skeleton":
           [{"pose": [0, ..., 0.41],
           "score": [0, ..., 0.46]}]
         },
        {...},
        {"frame_index": n,
         "skeleton":
          [{"pose": [...], "score": [...]}]}
       ],
   "label": "testifying",
   "label_index": 354
}
```

6.2.4 Training data and validation data

The aforementioned files are sufficient to train a DL model. However, the authors of the paper chose to compress the data into four new files - the training and validation data. These are made up of two files for training: *train_data.npy*, *train_label.pkl*, and two files for validation: *val_data.npy*, and *val_label.pkl*.

The nature of the content is the same for the train- and validation files. Visualizing the train files using Python:

```
train_label_pkl =
    [['---QUuC4vJs', ... , video name n], [354, ..., video label n]]
train_data_npy =
    [size of dataset][num channels][num frames][num keypoints][num values]
```

Specifically, the skeleton files and their corresponding entries in the summary file are used to construct these label- and data files, such that the label files contain the video's name and its label, while the data files contain the frame information. They have corresponding index number, simplifying lookups.

6.3 Creating the Rehab dataset

This section describes how we went from the original set of skeleton files generated from Kinetics, referred to as *Kinetics skeleton files* from now on, using OpenPose, to our own reduced set of skeleton files exclusively containing action classes relevant for rehabilitation, referred to as *Rehab skeleton files* from now on. The *Kinetics summary file* also had to be reduced to a new one, called *Rehab summary file*. The process is shown in fig. 6.3.

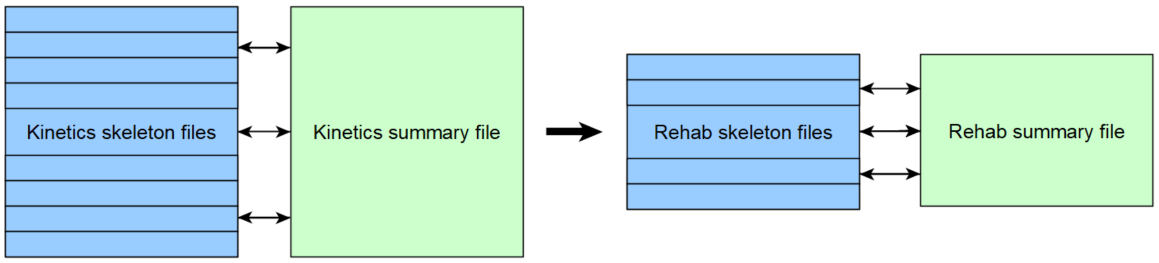


Fig. 6.3 From Kinetics files to rehab files. To create a dataset consisting of skeleton files more relevant for rehabilitation, we extracted relevant skeletons from the Kinetics skeleton files.

Initially, we identified 5 classes from the original Kinetics dataset; squat, lunge, deadlift, push-ups, and pull-ups, because of their relevance for physical training, and thus by extension rehabilitation. By iterating through the Kinetics summary file we created the Rehab summary file, containing the instances from Kinetics summary file with a label corresponding to one of our selected labels. Simultaneously, we iterated through the Rehab summary file to extract the matching Kinetics skeleton files. These skeleton files were copied to a new folder, containing the Rehab skeleton files. The Rehab skeleton files represent our new dataset, named *Rehab dataset* from now on.

Further, when iterating through the Kinetics summary file to create the Rehab summary file, we extracted 700 random entries containing action classes not in the 5 identified classes - 600 for training, 100 for validation. Lastly, we changed their labels to “unknown”. This to account for the relatively small set of action classes in the Rehab dataset.

6.4 Adding a new class to the Rehab dataset

Further, the ST-GCN project comes with a CSV file, where each row has the name of the action class, the URL to the video on YouTube, and the start- and stop time stamp specifying which part of the video is relevant for the dataset.

When basing the Rehab dataset on Kinetics, the selection of action classes are restricted to the ones present in Kinetics. Since we need a more flexible system, we wanted to develop a pipeline making it possible to add whatever action classes deemed relevant. Thus simplifying the process of tailoring the program for the rehabilitation process. Our pipeline is a five-step process:

1. Identifying a class to add
2. Obtaining videos of the identified action class
3. Cleaning the videos such that one gets the relevant parts and making sure it is the correct size
4. Running the videos through pose estimation software (OpenPose) to obtain the body keypoint coordinates for each frame
5. Constructing the skeleton files, based on the body joint coordinates from step 4

1) Identifying a class to add

When completing this step we had to select a class not present in the original Kinetics dataset that has a large enough volume of data available online. We ended up choosing "jumping jacks", due to the availability of online videos where this action is performed.

2) Obtaining videos of the identified action class

The procedure used to obtain the videos was inspired by the ST-GCN paper. This included creating a CSV file 6.2, and running a script iterating through that file, using YouTube-dl (Y. C. Hsuan, 2011) to download the videos, before cutting them at the corresponding time stamps using FFmpeg (Bellard, 2000), and lastly outputting the videos to a selected folder.

The video URLs was identified manually, searching for “jumping jacks” on YouTube. The URLs that linked to relevant videos was copied into the CSV file, together with the labels and time stamps. This resulted in a CSV file consisting of approximately 199 different URLs to videos of people doing jumping jacks.

In order to have a balanced dataset, i.e. approximately the same amount of videos for each class as in Kinetics, data augmentation, explained in section 3.1.4, was necessary. There are many different augmentation techniques, however, not all are relevant for our case. We identified frame flipping and zooming to initially be the most relevant for this project. Flipping and zooming generate new videos with body keypoints different from the original videos.

Augmentation techniques such as changing the contrast or brightness would not have worked, because it does not affect the position of the body keypoints. The augmentation was done using the open source program OpenCV (Bradski, 2000), increasing number of jumping jacks videos from 199 to 796.

3) Cleaning the videos

Yan et al. resized each video to the resolution of 340×256 and set the frame rate to 30 fps. It was essential that we followed suit, and generated our own keypoints from videos using the same resolution and frame rate, using FFmpeg.

4) Obtaining coordinates

Each processed/cleaned video of jumping jacks was run through OpenPose to extract the coordinates of each keypoints from the person performing the action. This resulted in one folder per video, each folder containing one JSON file for each frame, i.e. 300 files for a video with 300 frames.

5) Constructing the skeleton files

We created a Python script iterating through all the folders mentioned in step 4). For each folder, all its JSON files were merged into one skeleton file. Further, for those folders containing < 300 frames, we padded them by adding the first n frames to the end. Here n is the difference between 300 and the number of original frames. Finally, the new skeleton files, generated from the jumping jacks videos, was placed in the same folder as the Rehab skeleton files, such that we could train a new model on the Rehab dataset. As in section 6.1.2, we plot the number of non-empty frames, shown in fig. 6.4.

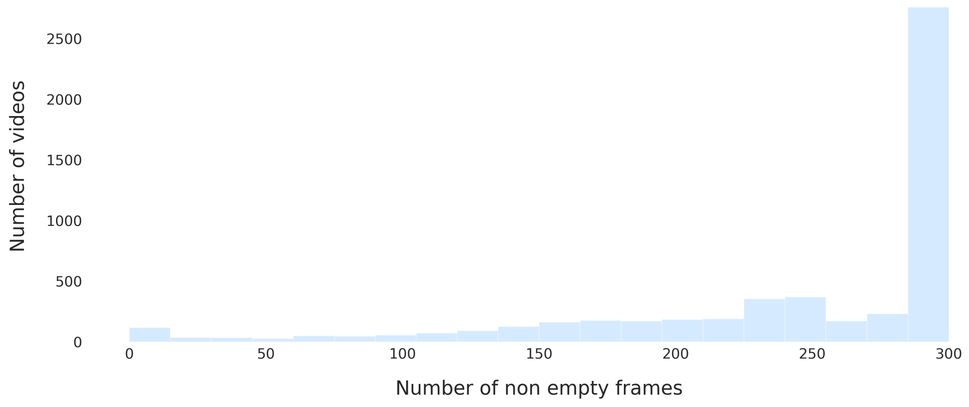


Fig. 6.4 Distribution of the number of non-empty frames in the Rehab dataset. The plot shows a more favorable distribution, compared to fig. 6.1, considering the decrease of videos on the left side of the plot.

6.5 Training & evaluating our new ST-GCN model

Before training the model, we extracted the usual validation set from the training set. This was done using a Python script, defining the desired training/validation ratio. Furthermore, we produced our own test set by recording videos of ourselves, performing actions from the Rehab dataset. The motivation for this distribution is explained in section 2.1.3.

The training was done by running the train script, supplied by Yan et al., on two NVIDIA TITAN RTX GPUs (4608 CUDA cores, 24 GB memory) running in parallel, using the hyperparameters specified in the ST-GCN project.

Further in this section we use the validation set to evaluate the performance of our newly trained ST-GCN model, customized for rehabilitation exercises. The test set is held out until we have trained and identified an optimal rehab model. Optimal here means first and foremost largest validation accuracy.

The data distribution is summarized in table 6.1, and the model's performance, on the validation set, is presented in in table 6.2.

Class label	Training	Validation	Test
Jumping Jacks	706	78	10
Squat	999	50	10
Lunge	609	50	10
Deadlifting	656	49	10
Push up	465	49	10
Pull up	971	50	10
Unknown	600	100	10
Total	5006	426	70

Table 6.1 Data distribution in the three splits of the Rehab dataset, training-, validation- and test set.

	Top 1 accuracy (%)
Validation set	48.36

Table 6.2 Rehab model performance on data pose estimated by OpenPose

Looking at the validation accuracy in table 6.2 it is evident that the model is suboptimal. For a more complete picture of the model's performance, we show the corresponding confusion matrix, produced during the validation process, in fig. 6.5.

It is quite interesting that the model is somewhat able to recognize "push-ups", and not at all "lunge", considering the difference in the amount of data, read from table 6.1. Further, the confusion matrix provides more optimism than the validation accuracy. This is based on though it struggles with classes such as "lunge" and "unknown", it shows that the model has the potential to learn the movements, shown by its 72/78 accuracy on "jumping jacks", and the aforementioned surprising accuracy on "push-ups".

With that said, as the model's accuracy is suboptimal, we do not evaluate its performance on the test set. For this to be relevant the model should have significantly higher validation accuracy. Efforts to achieve this is presented in chapter 7.

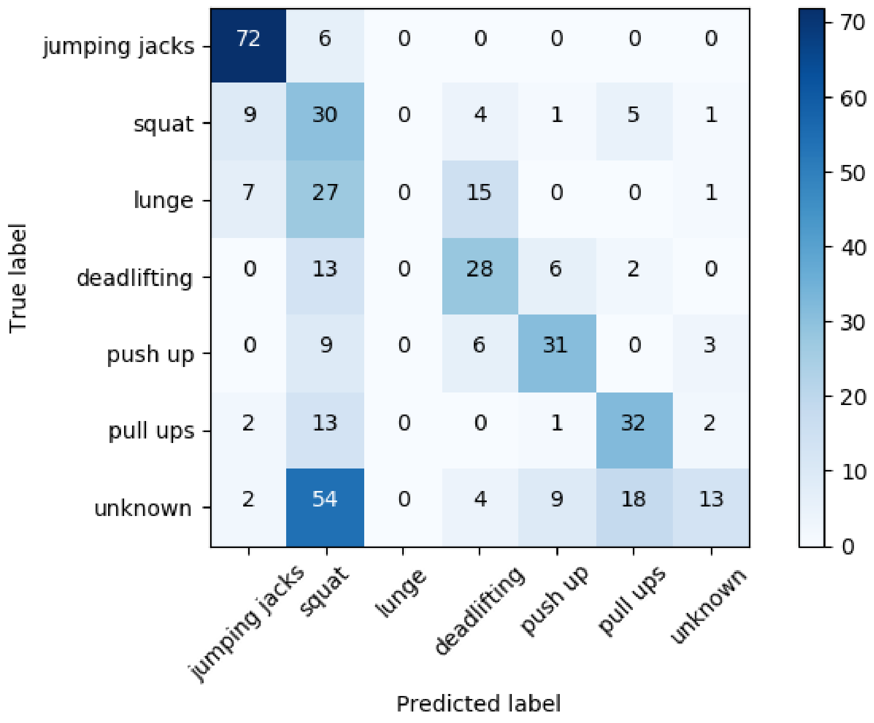


Fig. 6.5 Confusion matrix showing the Rehab model's performance on the validation set. The model struggles with most of the classes, especially "lunge" and "unknown", however it seemingly performs well on videos of "jumping jacks".

6.6 Combining our model with robotics

The final step in the process includes combining our trained ST-GCN model with a robot, such that the patient can get feedback on their movement(s). We used a Turtlebot 3 Waffle Pi (Foundation, n.d.), shown in fig. 6.6. This robot is compatible with ROS, thus we used ROS to control its behavior.

Fig. 6.7 shows a visualization of how we want the whole system's pipeline to be, at a high level. A web camera mounted on the robot captures a 0-10 seconds video of a person performing some action. This video is sent through pose estimation software, constructing the skeleton files, explained in section 4.2.1, necessary for running inference. Further, the generated skeleton sequences are sent through the ST-GCN model, returning a prediction on which action it thinks was performed. Lastly, the model's output is used by the robot, when deciding on the relevant feedback for the person.

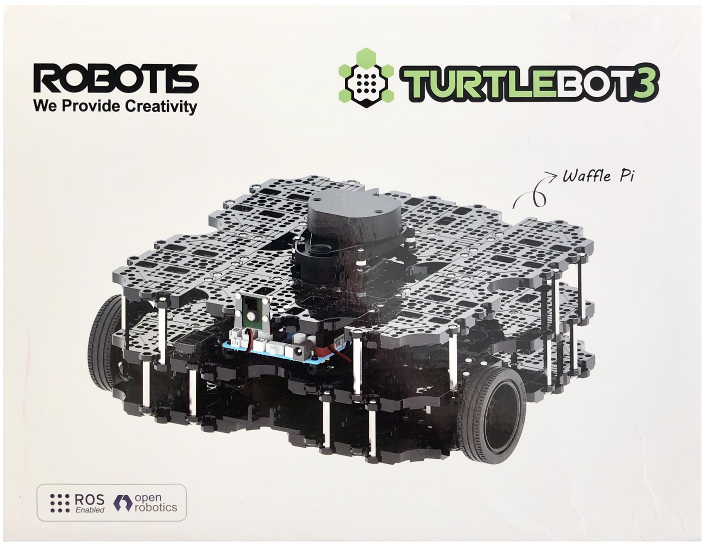


Fig. 6.6 TurtleBot 3 Waffle Pi, the robot in our system.

6.6.1 Deploying the model on the robot

Until this point, our machine learning model has been running on a powerful workstation computer. However, the robot is incapable of functioning properly with a large computer on top. The machine learning model had to be transferred to a smaller computer, still powerful enough to run the inference process efficiently.

The Turtlebot 3 Waffle Pi comes with a Raspberry Pi 3 Model B. However, Raspberry Pi is only equipped with a relatively modest ARM Cortex CPU and a weak Broadcom GPU. We require a GPU that is compatible with NVIDIA's CUDA framework platform to run our neural network. We therefore decided to replace the Raspberry Pi with NVIDIA's Jetson Xavier, shown in fig. 6.8. Xavier is a small ($105mm \times 105mm$), fast and power efficient embedded AI computing device, built around a CUDA compatible NVIDIA 512-core Volta GPU, and loaded with 16GB of memory and 137GB/s of memory bandwidth (NVIDIA, 2018).

TurtleBot has a microcontroller called OpenCR, used to control TurtleBot's motors based on the commands it receives from Raspberry Pi. OpenCR communicates with Raspberry Pi between its micro-USB port and Raspberry Pi's USB port. Replacing Raspberry Pi with Xavier involves getting the communication between OpenCR and Xavier to work properly. Xavier also has a USB port, but to communicate with OpenCR, a specific port named ttyACM0 has to to be enabled. After ensuring that this port was enabled, the communication between OpenCR and Xavier was successfully established

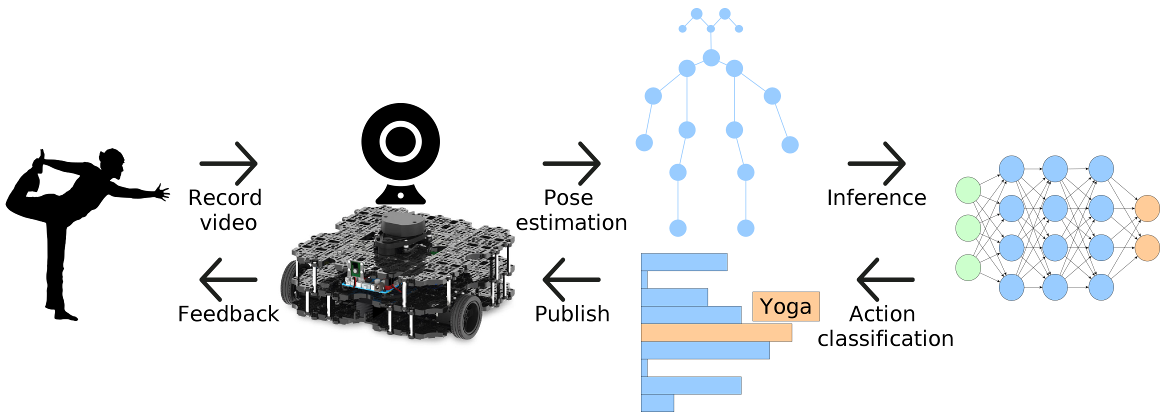


Fig. 6.7 The figure above shows a high-level view of our system pipeline, starting with recording of a person performing an action, ending with the person receiving feedback on their performed action, and everything in between. With inspiration from Yan et al., and SVG files from FlatIcon.

Fig. 6.8 NVIDIA's Jetson Xavier, the computing device used on the robot in our system. Image from NVIDIA's homepage (NVIDIA, 2018).

through a USB cable. The next step was to install ROS on Xavier, such that ROS nodes can publish TurtleBot navigation commands to ROS topics and thereby being able to control the robot's actions.

6.6.2 Integrating the Robot Operating System

Fig. 6.9 shows how the components in the system use ROS to communicate, where the blue squares are ROS topics, the orange squares are ROS nodes, and the green hexagon is the robot.

The communication in ROS starts with the *ST-GCN* node. This script feeds the pose keypoints extracted from the web camera to ST-GCN, then receives a prediction.

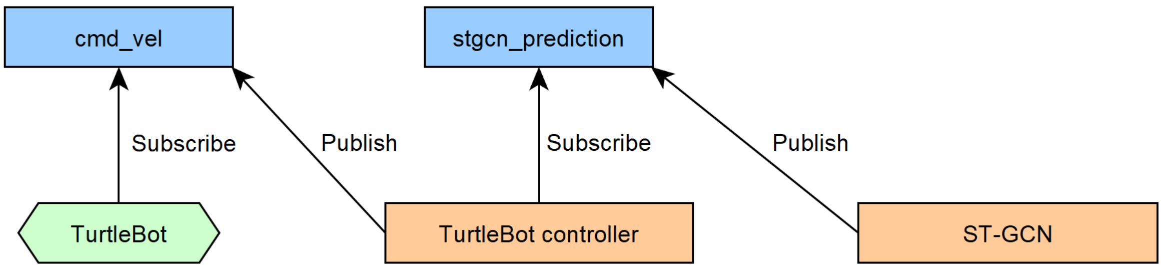


Fig. 6.9 ROS communication overview. By subscribing and publishing to relevant topics, the TurtleBot can react based on the ST-GCN model's output.

The prediction is then sent to the ROS topic *stgcn_prediction*. A code snippet from this node is shown below.

```
1 # 'ST-GCN' node
2
3 rospy.init_node('st-gcn')
4
5 prediction = run_inference(video)
6
7 # Publish the prediction to a ROS topic
8 publisher = rospy.Publisher('stgcn_prediction', String, queue_size = 1)
9 publisher.Publish(prediction)
```

Line 3 initializes the node, giving it the name "st-gcn". After our machine learning model has run inference on the recorded video, a publisher-object is created at line 8. The publisher can publish messages of types String to the "stgcn_prediction" topic, with queue_size 1. Queue size 1 means that if the publisher is publishing messages with a higher rate than ROS is able to send over the wire, then some messages get queued. The queue_size parameter decides how many messages this queue can take before the oldest messages get deleted. Since our application only publishes one message per prediction, a queue size larger than 1 is unnecessary. Line 9 publishes the prediction to the topic.

On the other side of the "stgcn_prediction" topic, we find the "TurtleBot controller" node, which both subscribes to the "stgcn_prediction" topic and publishes to the "cmd_vel" topic. This way, the node can receive the prediction and control the TurtleBot based on what it receives. The "cmd_vel" topic contains information about how the TurtleBot should navigate, so by publishing messages to this topic one can control the TurtleBot's movement. TurtleBot subscribes to this topic, ready to take action as soon as a message is published. The code below shows an example of how the code in “TurtleBot controller” node can look like.

```
 1 # 'TurtleBot controller' node
 2
 3 rospy.init_node('turtlebot_controller')
 4 velocity_publisher = rospy.Publisher('cmd_vel', Twist, queue_size = 1)
 5
 6 def callback(prediction):
 7     if prediction.data == 'squat':
 8         vel_msg.linear.x = 1
 9         velocity_publisher.publish(vel_msg)
10
11         time.sleep(2)
12
13         vel_msg.linear.x = 0
14         velocity_publisher.publish(vel_msg)
15     else:
16         # Do nothing
17
18 rospy.Subscriber('stgcn_prediction', String, callback)
```

Line 3 initializes the node, giving it the name “turtlebot_controller”. The next line creates a publisher-object, used to publish messages of type *Twist* to the “cmd_vel” topic, with queue_size set to 1. “Twist” is the message type TurtleBot uses for navigation commands. The “callback” method is triggered when it receives a message, i.e. when the topic the node subscribes to receives a message. In this example, if the prediction is “squat”, then the node publishes a velocity message to the TurtleBot, saying that it should start driving in the x-direction, i.e. straight ahead. The robot drives for two seconds before the velocity is set to zero again. If the prediction is something other than squat, the robot stays unmoved. Note that this is an example of how the robot can react. In a rehabilitation process, it is natural with more informative feedback.

Lastly, line 18 defines which topic the node subscribes to. In this case, the topic is “stgcn_prediction”, containing messages of type String. It also defines which method that will be triggered when a message is received, which is the “callback” method.

6.7 Evaluating the system

To reiterate, the hypothesis we want to answer is: *By combining state-of-the-art techniques from computer vision and robotics it is possible to construct robots capable of accurate, real-time human action recognition, making it feasible to deploy such systems as robotic physical therapists.*

Completing this chapter we have constructed a functional robot capable of human action recognition, and human interaction. The accuracy of the machine learning model ended up being 48.36% on the validation set, with the corresponding confusion matrix to provide an overview on how the model performed on each class in the rehab dataset.

Further, our system cannot be classified as a real-time system, because the robot uses an excess of about 150 seconds to provide the user with feedback. For the robot to be relevant for rehabilitation, it is preferable that the feedback-time is close to 0 seconds and a significant increase in accuracy.

Based on these evaluations, our conclusion so far is that the robot has potential, but is currently unusable for a rehabilitation process due to unsatisfactory accuracy and running time. This leads us to the next chapter, which deals with improving the robot.

Chapter 7

Improving our system

As presented in the previous chapter, running the system on Jetson Xavier resulted in an inference time of more than 2 minutes on a 10 seconds video, which is unsatisfactory. Furthermore, the accuracy of the model was suboptimal. Both of these has to be improved in order for us to positively answer our hypothesis. This chapter will go through the experiments done to improve the system, by both reducing the inference time and improving the underlying model.

7.1 Decreasing the inference time

Ideally, the inference time should be close to 0 seconds, i.e. close to live classification. In order to achieve this, each part of the system needs to be analyzed such that we can establish which ones can, and should, be optimized. Note that we continue to use 10 seconds videos for our experiments.

7.1.1 Identifying bottlenecks in our system

Our system is running several processes, some more time consuming than others. Identifying the bottlenecks is about identifying the processes consuming the majority of the total time, such that one can focus on optimizing those. For this the Python package *Pyinstrument* was used, which provides the user with the running time for every process and subprocess. By running our inference pipeline using this tool, we got the results shown in table 7.1. Note that the time we want to reduce is where the user is waiting for feedback from the robot, i.e. after the video has been recorded, referred to as *total waiting time*.

Process	Time used (s)
Initializing environment	5.6
Imports	11.4
OpenPose skeleton estimation	130.9
Inference	3.2
Total waiting time	151.1

Table 7.1 Each process in the robot's prediction pipeline and their time requirements, when using OpenPose. Time requirements are calculated using *Pyinstrument*.

OpenPose is clearly the largest bottleneck in the system, making it the top priority to either improve or substitute, in order to decrease the inference time.

A study of the recent literature on pose estimation led us to the software TF-pose-estimation (tf-pose), a Tensorflow implementation of OpenPose created by NVIDIA researchers (Madeleine Waldie, 2018). As mentioned in section 4.1, OpenPose is based on the deep learning software Caffe. Tf-pose uses similar algorithms, but by using TensorFlow instead of Caffe, the pose estimation is significantly faster. Thus, by substituting OpenPose with tf-pose in our pipeline, the inference time should decrease notably.

7.1.2 Substituting OpenPose with tf-pose

The substitution is illustrated using code:

OpenPose

```
openpose_bin_path = 'openpose/build/examples/openpose/openpose.bin'

cmd = (f'{self.openpose_bin_path} --video {input_fpath} --model_folder
    {self.model_folder} --write_json {output_fpath} --model_pose COCO
    --keypoints_scale 3')

parent = subprocess.Popen(cmd, shell=True, stdout=subprocess.PIPE,
    stderr=subprocess.PIPE)
```

tf-pose

```
pose_est_path = 'tf-pose-estimation/run_video.py'

cmd = (f'python {pose_est_path} --video {input_fpath} --output_json
    {output_fpath}')

parent = subprocess.Popen(cmd, shell=True, stdout=subprocess.PIPE,
    stderr=subprocess.PIPE)
```

Testing our pipeline after the substitution gave us the results presented in table 7.2.

Process	Time used (s)
Initializing environment	5.6
Imports	11.5
Tf-pose skeleton estimation	30.8
Inference	1.0
Total waiting time	48.9

Table 7.2 Each process in the robot's prediction pipeline and their time requirements, when using tf-pose. Time requirements are calculated using *Pyinstrument*.

By replacing OpenPose with tf-pose the pose estimation time was reduced from 130.9 seconds to 30.8 seconds. I.e. tf-pose is 4-5 times faster than OpenPose. Therefore, tf-pose was deemed the pose estimation tool to use in further experiments.

7.1.3 Modifying our pipeline

As the results in table 7.2 show, sending a 10 seconds video through tf-pose to extract the skeleton files takes more than 30 seconds. Even though this is a significant improvement from using OpenPose, 30 seconds is still a long time to wait to get feedback from the robot.

Our pipeline so far is

1. Record video
2. Pose estimation

3. Run inference

One way to further improve our system is to combine the video recording (1) and the pose estimation (2), lowering the time used on the pose estimation step. Our new pipeline is then:

1. Record video
    - Pose estimation on each frame, as they are recorded
2. Run inference

With this pipeline, the recording and pose estimation runs concurrently, such that the skeleton files are ready for the inference process as soon as the recording has finished. The implementation of this new approach is shown below.

```
1   estimator = TfPoseEstimator(model_path)
2   cam = cv2.VideoCapture(0)
3
4   duration = 10
5   start_time = time.time()
6
7   # recording video
8   while (time.time() - start_time) <= duration:
9       video_frame = cam.read()
10      estimator.inference(video_frame)
11
12  tfpose_skeleton_to_stgcn(skeleton_file_path)
```

The while loop processes the actual video recording. In line 9, one new video frame from the web camera is read. In line 10, the method "inference" takes the video frame as input for pose estimation. This method extracts the skeleton from the frame, and creates a JSON file containing these values, as explained in section 6.2. After the recording is done, the code in line 12 combines all of the frame skeleton files into an ST-GCN-compatible skeleton file, i.e. something that can be ran through the model. The new pipeline was profiled using Pyinstruments, giving the results shown in table 7.3.

The results show that the new pipeline reduces the total waiting time from 30.8 seconds to 1.0 second. The time-consuming steps are now before the video recording,

Process	Time used (s)
Initializing environment	5.6
Imports	12.2
Record a 10 seconds video	10.0
Inference	1.0
Total waiting time	1.0

Table 7.3 Each process in the robot's optimized prediction pipeline and their time requirements, when using tf-pose. Time requirements are calculated using *Pyinstrument*

i.e. initializing the environment and importing necessary packages. These steps only have to be done once, meaning that after initialization, one can use a loop to run as many video recordings as desired in the same environment, resulting in no waiting time before the recording starts. This is why the "Initializing environment"- and "Imports"-processes listed in table 7.3 are not taken into account when calculating "Total waiting time".

Reducing "Total waiting time" this way has a trade-off: the video's frame frequency, or frames per second (fps). In the current setup, we can record at a maximum of 9 fps. This is because the inference on each frame (line 10) takes some time, more precisely $1/9 = 0.11$ seconds on average, and the next frame is not read from the web camera before the inference has finished (we will not pursue ways of parallellizing the recording and the inference in this work, e.g. multi-threaded inference). From table 7.1 and 7.2 we calculate that OpenPose is 4.25 times slower than tf-pose. Hence, in our new pipeline, OpenPose would have used $0.11 \times 4.25 = 0.47$ seconds on each frame, resulting in a fps value of $1/0.47 = 2.1$. When comparing 9 fps videos with 2.1 fps ones, our conclusion is that 9 fps is sufficient for action recognition since one clearly can see which movement is being done, while a 2.1 fps video is too "laggy". This motivates us to use our new pipeline combined with tf-pose for further experiments.

7.2 Establish a new benchmark

After successfully decreasing the inference time, enough to make the system usable, we needed to establish a benchmark for our newly implemented pose estimation software, tf-pose. This would serve as a necessary baseline for improving the model. This is because a final product will still be dependent on being accurate, not just fast.

7.2.1 Recreating Yan et al.'s dataset using tf-pose

To establish a new tf-pose benchmark it was necessary to recreate the skeletons from the Kinetics dataset in their entirety using this new pose estimation software. We used essentially the same pipeline as described in section 6.4. The CSV files, specifying the video's extension, start- and stop time, and labels, were provided by the ST-GCN project's GitHub repository.

The CSV files are divided into the standard three subsets: train, validation, and test, providing us with the distribution used in Yan et al.'s experiments. At the time of writing this article this distribution is as follows:

Subset name	Number of videos
Training	246 535
Validation	19 907
Test	38 686
Total	305 128

Table 7.4 Data distribution for the original Kinetics dataset, split into the standard training-, validation- and test sets.

However, the test data is unlabeled, and we do not have the resources needed to label them. This gave us approximately 260 000 videos to download and process.

Downloading the data

As mentioned in section 6.4, we used Youtube-dl to download YouTube videos. This tool has a large set of features. However, it currently does not offer the ability to download specific parts of videos. Therefore, the entire videos needed to be downloaded.

With 260 000 YouTube videos to be downloaded, with sizes ranging from a couple of MBs to a couple of GBs, we calculated that we needed several TBs of storage. However, we made a few modifications to the pipeline, and deleted each full YouTube video after extracting the relevant part. I.e. we combined the downloading and the cleaning of the videos. This resulted in decreased storage needs, down to a couple of hundred GBs.

The still relatively large storage and time requirements led us to transfer the pipeline to an external server. Further, in order to run the pose estimation software, it was necessary that this server was equipped with sufficiently sophisticated GPU(s).

Explicitly missing data

YouTube videos are often region restricted or deleted, making us unable to recreate the dataset in its entirety.

After running the pipeline we ended up with the following distribution:

Subset name	Number of videos
Training	223 572
Validation	18 404
Total	241 976

Table 7.5 Data distribution for the currently available parts of the Kinetics dataset, split into the standard training-, validation sets. Ignoring the test set because of the lack of corresponding label.

This means that about 20 000 of the videos used by Yan et al. were currently unavailable. We call the dataset in table 7.5 the *currently available dataset*.

Implicitly missing data

In section 6.1.2, we visualized OpenPose's ability to recognize humans in videos, by plotting the number of non-empty frames in each video. After substituting the pose estimator, we wanted to test the new software's human-identification ability to investigate whether we are trading speed for accuracy.

To compare these two pose estimation softwares, we processed the 240 000 videos in the currently available dataset using tf-pose, to a (skeleton) dataset we call *tf-pose-cur*. Then we plotted the distribution of non-empty frames in the videos from this tf-pose-cur dataset, versus the dataset used in Yan et al.'s project, called *OpenPose-og*. This visualization is shown in fig. 7.1.

Here, we can clearly see that there is a significant increase in the number of videos with low-to-no amount of skeleton data. Further, it is interesting to note that on

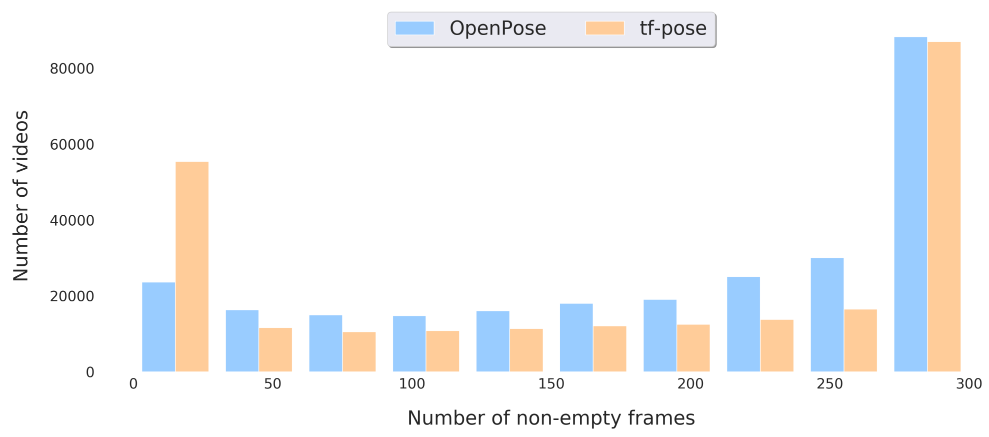


Fig. 7.1 The distribution of the number of non-empty frames in two pose-estimated versions of the Kinetics datasets: the currently available videos, processed by tf-pose (orange), and the originally available videos, processed by OpenPose (blue). One can clearly see a notable increase in the number of videos with a low amount of skeleton data when using tf-pose compared to OpenPose.

the other end of the scale there difference is small. This could potentially mean that OpenPose is superior when pose data is difficult to extract, however, when this is not the case they perform more or less the same.

Get the benchmark

Running the tf-pose-cur dataset through the training and validation process, using the same hyperparameters as Yan et al. resulted in a benchmark accuracy of 24.32% - about 6 percentage points less than their reported benchmark: 30.7%. Our re-training attempt using tf-pose was motivated by obtaining a comparable benchmark to Yan et al.'s benchmark. Therefore, this result was worse than we hoped for. Note that the performance is measured on the validation set.

However, as mentioned, the number of videos in the currently available dataset (240 000), and Yan et al.'s dataset (260 000) is notably different.

To test tf-pose and OpenPose under as similar conditions as possible we generated, or, rather, extracted a new dataset. We name this (skeleton) dataset *OpenPose-cur*. This was done by comparing the two datasets tf-pose-cur and OpenPose-og, extracting the files from OpenPose-og that was also in tf-pose-cur. The process was quite similar to the data extraction task explained in section 6.3, except the extraction check was on each skeleton file's name, not their classes. OpenPose-cur now consisted of skeleton frames from the same videos as tf-pose-cur, though processed by different software.

The OpenPose-cur dataset was run through the same training and validation process to get a new benchmark, comparable to our model. This lead to an increase in validation accuracy: 30.9%, from the Yan et al.'s reported 30.7%. This was surprising, because deep learning models usually perform better with larger data sets. However, the difference is relatively insignificant. Still, it is interesting that model performance did not decrease.

One possible explanation is that it seems like a majority of the unavailable videos did not contain human actions, i.e. empty videos. So, comparing the number of non-empty frames from OpenPose-og to the number of non-empty frames in OpenPose-cur, there is a notable decrease in the lower part of the plot, as shown in fig. 7.2.

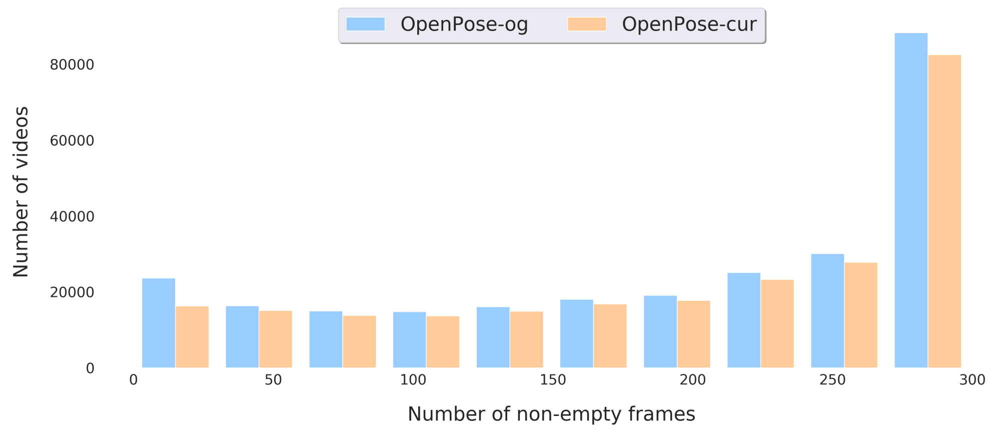


Fig. 7.2 Comparing the distribution of non-empty skeleton-frames in the pose-estimated original dataset vs. the pose-estimated currently available dataset, both processed by OpenPose. One can see there is an evident decrease in the left part of the plot.

We sum up the benchmark results in table 7.6, and their datasets' distribution of non-empty frames in fig. 7.3.

Skeleton dataset	Benchmark (%)
OpenPose-og	30.70
OpenPose-cur	30.90
tf-pose-cur	24.32

Table 7.6 Summarizing the benchmark results from the three (skeleton) datasets.

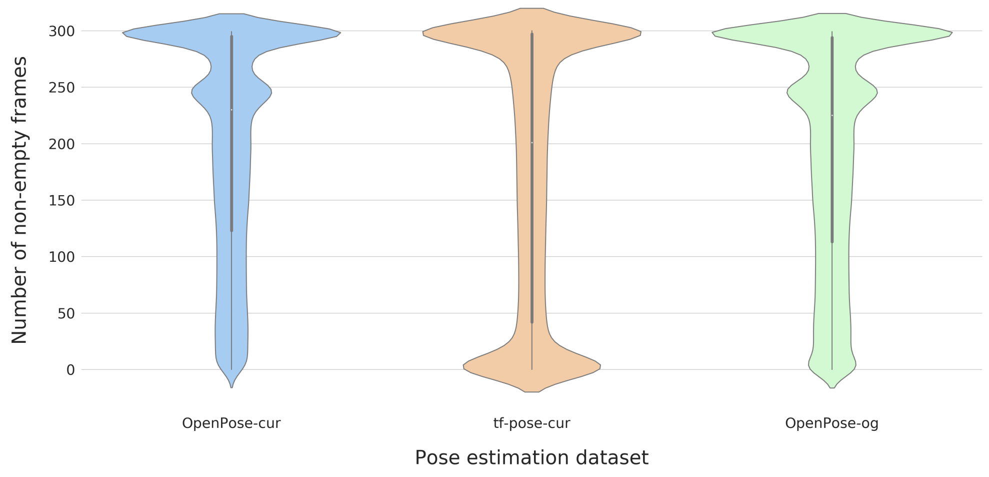


Fig. 7.3 The distribution of non-empty frames from the three benchmark datasets: "OpenPose-og", visualized in green, - the original OpenPose dataset used by Yan et al. "tf-pose", visualized in orange - the currently available dataset, pose estimated by tf-pose. Lastly, "OpenPose-cur", visualized in blue - the currently available dataset, pose estimated by OpenPose.

Furthermore, Yan et al. report results on the data subset "Kinetics motion" dataset, explained in section 4.2.6: 72.4%. Our model, trained on the tf-pose-cur dataset, got an accuracy of 57.71% when validating on this Kinetics motion data subset. We summarize these benchmarks in table 7.7 indicating that there is a real tradeoff between speed and accuracy in our current setup. Note that we did not re-run this experiment with the new OpenPose-cur dataset, because of the relatively insignificant performance difference.

Dataset	Pose estimator	Accuracy (%)
Original	OpenPose	30.70
Currently available	tf-pose	24.32
Motion	OpenPose	72.40
Motion	tf-pose	57.71

Table 7.7 Summarizing the benchmarks of models trained on OpenPose- and tf-pose generated data. One can see a clear decrease in accuracy when switching from OpenPose to tf-pose.

Lastly, in section 6.3 we introduced a subset of action classes relevant for rehabilitation, named “Rehab data”. This dataset consists of 6 classes plus a set of random videos, labeled as “unknown”. This is independent of the experiments presented by Yan et al.

We ran the equivalent subset of data from the currently available videos, processed using tf-pose, through the training process. We name this model *Rehab model*. This new model was run through the validation process, giving better results than the OpenPose model (table 6.2).

	Accuracy (%)
Validation set	62.65

Table 7.8 Rehab model performance on data pose estimated by tf-pose.

The corresponding confusion matrix is shown in fig. 7.4

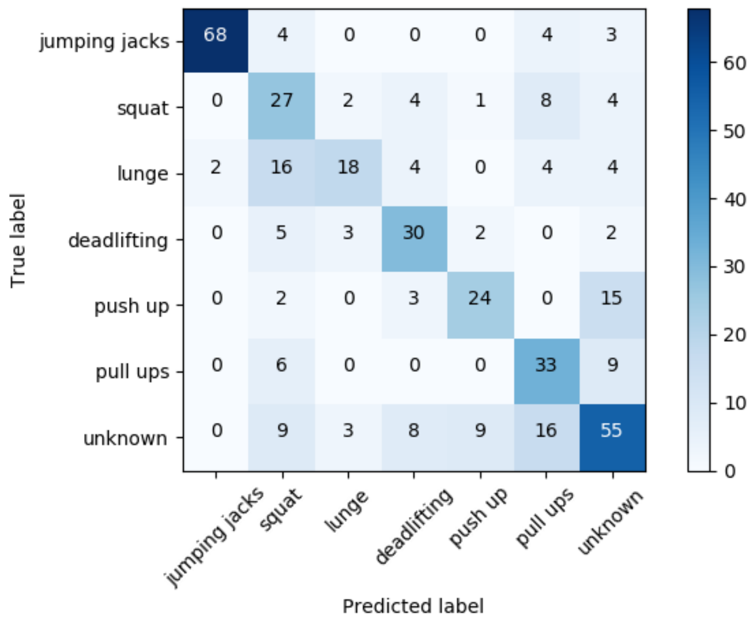


Fig. 7.4 Confusion matrix showing the Rehab model’s performance on the validation set. This is an improvement from the confusion matrix showing the result from an equivalent process, using OpenPose, rather than the currently used tf-pose.

Considering that tf-pose generated data had shown inferior results earlier, this seemed out of the ordinary. The reason for this abnormality could be due to the difference in the number of empty frames in the data. This was investigated by plotting the number of empty frames in the Rehab subsets for training and validation, for both OpenPose-og and tf-pose-cur in fig. 7.5. Here we can see that the difference in the number of non-empty frames is more or less insignificant, showing that the abnormality is not due to the difference in the number of empty frames.

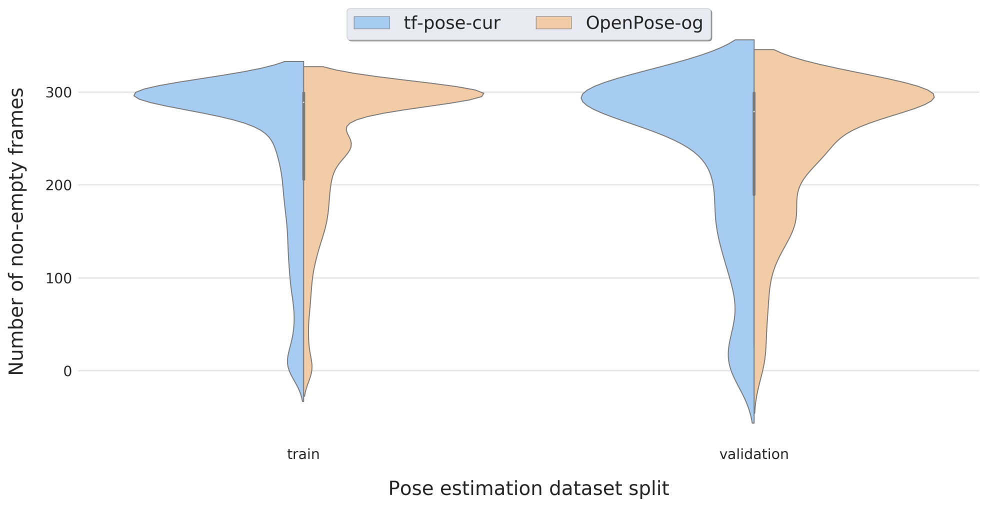


Fig. 7.5 The distribution of non-empty frames from the two splits of the Rehab dataset: train and validation. Comparing the pose estimation softwares: OpenPose (orange), and tf-pose (blue).

7.3 Increasing the accuracy of our ST-GCN model

With a new benchmark defined, we wanted to increase the accuracy of our new rehab model. As explained in chapter 3, there are many ways of improving deep learning models, and this section will lay out how we did this.

7.3.1 Transfer learning

As mentioned in section 3.1.3, an important tool when training deep learning models is transfer learning. This has the potential to both increase accuracy and decrease training speed, all while demanding less data. The amount of data is a general problem in deep learning, and this specific domain of action recognition is no exception, making transfer learning an important technique to include in our work.

In section 3.1.3 it was also said that transfer learning is about improving learning in a new task through the transfer of knowledge from a related task that has already been learned. In our case, the related task that has already been learned is general human movements, i.e. the model trained on the entire Kinetics dataset. The new task is to learn specialized human movements.

By transferring the weights from the pre-trained model to our rehab model, the weights will be initialized before the training starts. In theory, this will lead to faster convergence towards a minimum. Hence, less training data is needed to achieve similar performance as a model trained on more data with randomly initialized weights. We wanted to see if this theory could work in practice for our project. The results after training the Rehab model taking advantage of transfer learning are shown in table 7.9, with the corresponding confusion matrices in fig. 7.6

	Accuracy (%)
Validation set	83.05

Table 7.9 Rehab model performance after implementing transfer learning.

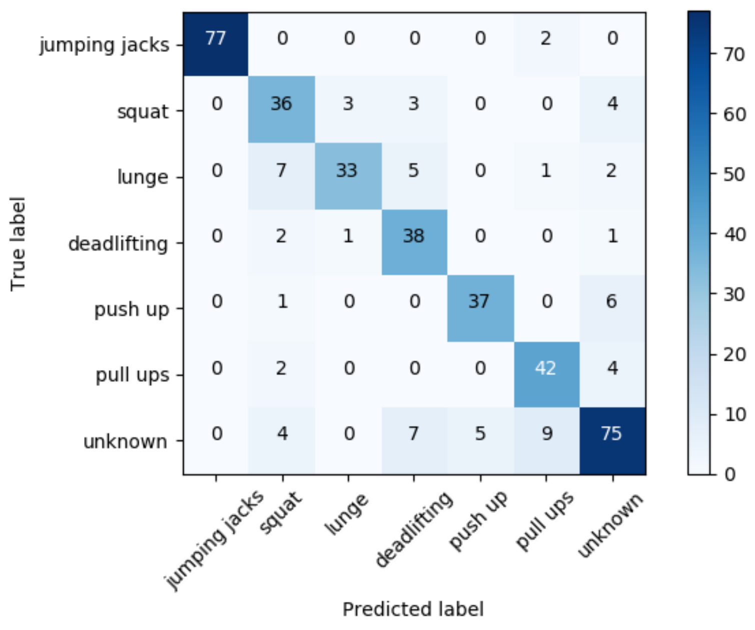


Fig. 7.6 Confusion matrix showing the Rehab model's performance on the validation set after using transfer learning. This confusion matrix combined with table 7.9 shows a significant increase in performance.

Lastly, we can plot the accuracy and loss calculated during training using transfer learning, against not using transfer learning:

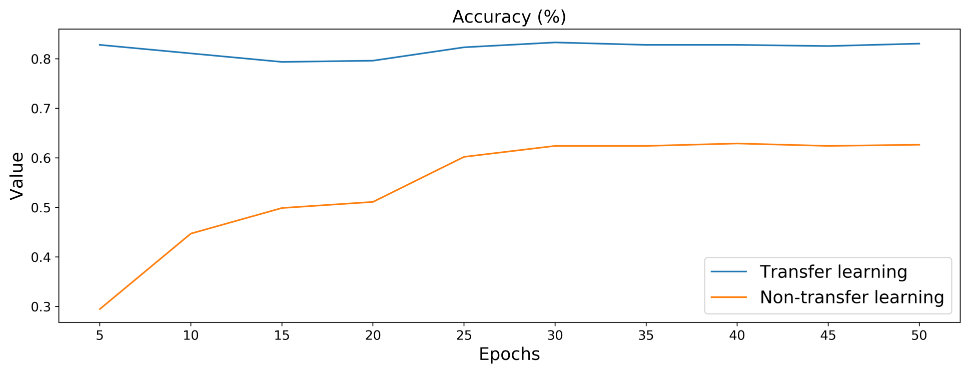


Fig. 7.7 Plotting the mean accuracy of the model during training when using transfer learning vs. the mean accuracy of the model during training when not using transfer learning. It is evident that transfer learning results in a significant increase in model accuracy.

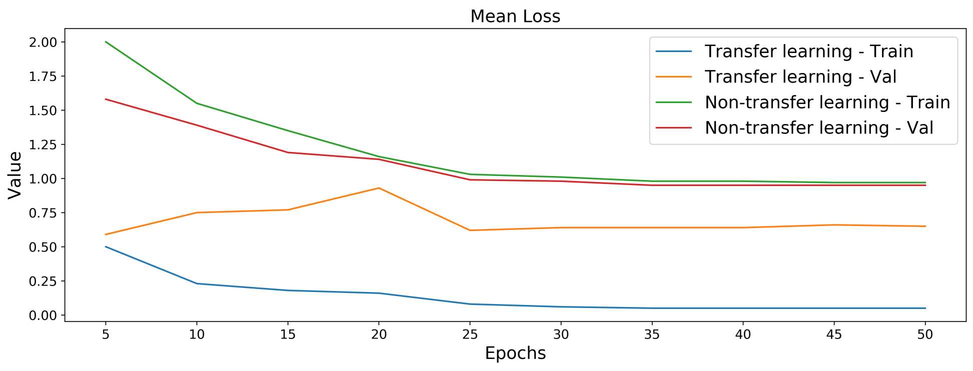


Fig. 7.8 Plotting the model's mean train- and validation loss when using transfer learning vs. the model's mean train- and validation loss when not using transfer learning. One can see a clear increase in the difference between training loss and validation, resulting in an overfitting model. However, considering the size of the accuracy increase this is definitely the way forward.

We can see from table 7.9 that transfer learning results in a overall significantly improved model, with an increase of approximately 20 percentage points. As most of the classes in this dataset were present in the Kinetics dataset, an increase in accuracy is not surprising. Note that based on the accuracy- and loss plots, fig. 7.7 and fig. 7.8, the model overfits to the training data.

Visualization of loading the weights from one model to another, using Python:

```
def load_weights(self, model, weights_path):
    weights = torch.load(weights_path)
    weights = OrderedDict([[k.split('module.')[-1], v.cpu()] for k, v in
        weights.items()])

    model_dict = model.state_dict()
    pretrained_dict = {k: v for k, v in weights.items() if k in model_dict}
    model_dict.update(pretrained_dict)

    model.load_state_dict(model_dict)

    return model
```

7.3.2 Freezing layers

As explained in section 3.1.3, transfer learning is often combined with freezing/unfreezing of pre-trained layers, such that one optimizes the gains from using pre-trained weights.

Whether a layer is updated during training is decided by each layer's boolean attribute "requires_grad". So, after loading the pre-trained weights, we set all but the last (fully connected) layers' "requires_grad" to False. Specifically, we used the model's "children" attribute, freezing every child node up to the last one. Visualized using Python:

```
if freeze:
    for child in model.children()[:-2]:
        for name, param in child.names_parameters():
            param.requires_grad = False
```

Then, after a couple of updates to the last layer(s), we unfreeze the frozen nodes, such that the entire rehab model can be optimized to the rehab dataset. In addition, these processes should be combined with a change in learning rate as well, with a decrease after unfreezing, for example halving the learning rate. Here visualized in Python:

```
def unfreeze_all(self):
    for child in self.model.children():
        for param in child.parameters():
            if param.requires_grad is False:
                param.requires_grad = True
    self.adjust_lr(specific_adjustment=0.5)
```

Lastly, we edit the learning rate schedule by changing the hyperparameters "steps" and "base learning rate", explained in section 4.2.4. We decreased the base learning rate from 0.1 to 0.01, and changed the step interval from [20, 30, 40, 50] to [10, 20, 30, 40].

These changes, combined with unfreezing pre-trained layers and halving the learning rate after finishing 10% of the epochs, leads to an overall decrease in the network's learning rate. This is motivated by the fact that we do not want to change the loaded weights too much, as they represent general human action recognition - we only want to specialize them to our smaller subset of human action recognition. The result was a small increase in accuracy, shown in table 7.10, with the corresponding confusion matrix, fig. 7.9, also not showing a great accuracy increase.

	Accuracy (%)
Validation set	84.77

Table 7.10 Rehab model performance after implementing transfer learning, layer freezing, and hyperparameter tuning.

Note that one could also consider using different learning rates for each (group of) layers, e.g. discriminative learning rates.

7.3.3 Adding complexity

We have shown that freezing the pre-trained layers, thereby only training the randomly initialized layers for the first few epochs, resulted in an increase in accuracy, albeit small. This motivated us to experiment with adding more (randomly initialized) layers to the model, i.e. increasing its complexity and capacity.

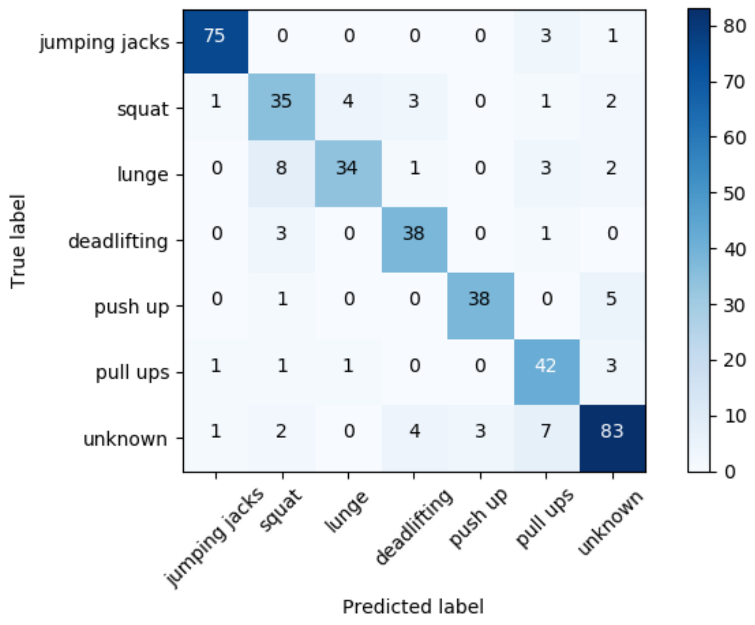


Fig. 7.9 Confusion matrix showing the Rehab model's performance on the validation set after using transfer learning, layer freezing, and hyperparameter tuning. There is not a notable difference from fig. 7.6, except for the eight more correctly classified videos of "unknown".

We experimented with adding one new ST-GCN layer and one or more fully connected layer(s), both explained in section 4.2.4. Furthermore, we tried increasing the number of activations in these newly added layers: from 256, to 512. Here visualized in Python:

```python
self.st_gcn_networks = nn.ModuleList((
    st_gcn(in_channels, 64, kernel_size, 1, residual=False, **kwargs),
    st_gcn(64, 64, kernel_size, 1, **kwargs),
    st_gcn(64, 64, kernel_size, 1, **kwargs),
    st_gcn(64, 64, kernel_size, 1, **kwargs),
    st_gcn(64, 128, kernel_size, 2, **kwargs),
    st_gcn(128, 128, kernel_size, 1, **kwargs),
    st_gcn(128, 128, kernel_size, 1, **kwargs),
    st_gcn(128, 256, kernel_size, 2, **kwargs),
    st_gcn(256, 256, kernel_size, 1, **kwargs),
    st_gcn(256, 512, kernel_size, 1, **kwargs), # Increased output size
    st_gcn(512, 512, kernel_size, 1, **kwargs), # Newly added
    ))
```

```
...
self.fcn2 = nn.Conv2d(512, 512, kernel_size=1) # Newly added
self.fcn = nn.Conv2d(512, num_class, kernel_size=1) # Increased input size
```

We conclude that one extra fully connected layer with size 512 for its in- and output, while not adding a new ST-GCN layer, but rather increasing the last ST-GCN's layer output size and the final fully connected layer's input size, gave the most improvement of these experiments: 0.5 percentage points validation accuracy increase, shown in table 7.11 with the corresponding confusion matrix in fig. 7.10.

	Accuracy (%)
Validation set	85.26

Table 7.11 Rehab model performance after implementing transfer learning, layer freezing, hyperparameter tuning, and added complexity.

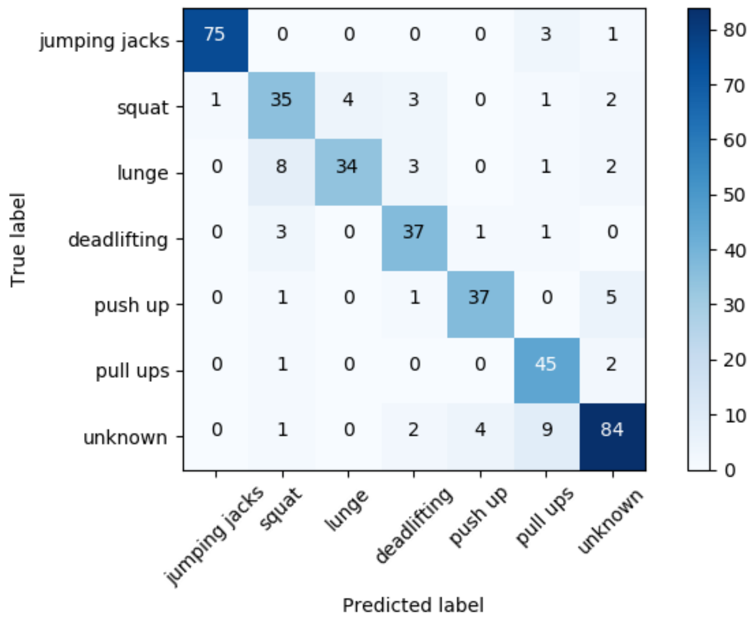


Fig. 7.10 Confusion matrix showing the Rehab model's performance on the validation set after using transfer learning, layer freezing, hyper-parameter tuning, and added complexity.

Now, it rather seems that these experiments show that adding complexity actually results in an inferior model that overfits to the training data. This can be seen from plotting this more complex model versus the model from section 7.3.2, shown in fig. 7.11 and 7.12.

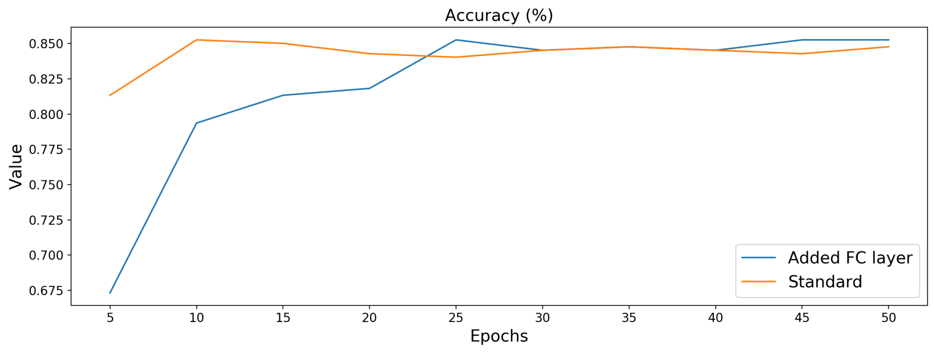


Fig. 7.11 Plotting the mean accuracy of the model during training when adding a fully connected layer vs. the model's mean accuracy during training when not having an extra fully connected layer. The addition of a fully connected layer seems to improve the model's accuracy, minimally.

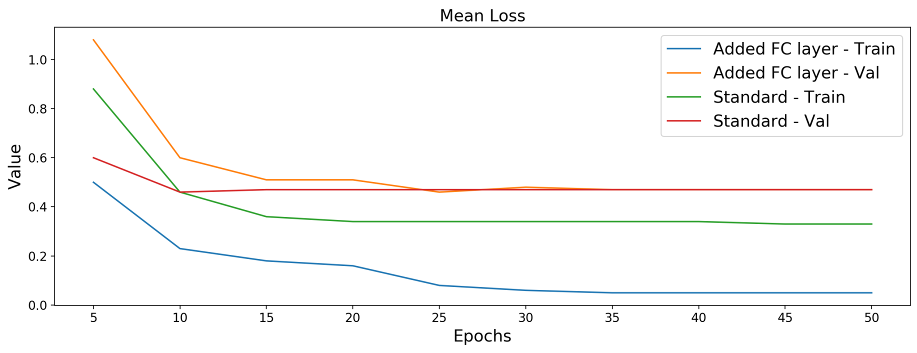


Fig. 7.12 Plotting the model's mean train- and validation loss when adding a fully connected layers vs. the model's mean train- and validation loss without an extra fully connected layer. Here we can clearly see a decrease in the model's generalization ability, i.e. increased overfitting.

Fig. 7.12 actually looks a lot like fig 7.8, making us interested in exploring methods to decrease the model's generalization error, i.e. implement regularization techniques.

7.3.4 Experimenting on NVIDIA's DGX station

As mentioned in section 6.4, we used two NVIDIA TITAN RTX GPUs running in parallel, with the hyperparameters specified in ST-GCN's project for training. However, we were fortunate enough to gain access to NVIDIA's DGX station, sporting four NVIDIA® Tesla® V100 Tensor Core GPUs, integrated with a fully-connected four-way NVLink™ architecture, delivering 500 teraFLOPS of AI power. This provided us with about 32GB×4 = 128GB of GPU memory, putting us in a position to run more, and faster hyperparameter tuning.

With this state-of-the-art hardware it was possible to test if we could recreate and possibly surpass Yan et al.'s reported score on the full Kinetics dataset, but more importantly improve our tf-pose benchmark. This was motivated by a couple of reasons. One is we had not verified their reported results - if we were unable to recreate this, maybe the tf-pose benchmark was not that far off. Another being to see if small adjustments made possible by more advanced hardware were enough to increase the OpenPose benchmark, and if so, by how much. Furthermore, if these experiments result in increased accuracy for the OpenPose dataset, would it have the same impact on our tf-pose dataset?

Recreating Yan et al.'s benchmark

Yan et al. reported a 30.7% accuracy on the 400 class Kinetics dataset. They used 8×TITANX GPUs, giving them about 12GB×8 = 96GB of GPU memory.

Using the processed OpenPose data, made available by Yan et al. through both GoogleDrive and BaiduYun, we are able to verify the reported results. The model even surpassed the reported results, albeit by a relatively insignificant margin, giving a validation accuracy of 31.6%, using their hyperparameters.

Improving tf-pose benchmark

As with the OpenPose data, we recreated the tf-pose benchmark using the DGX and got approximately identical results: 24.32%. Hence, we have shown that using DGX compared to RTX or TITANX does not in itself produce a change in performance. This was expected. However, as mentioned, using the DGX gives us the ability to run more, and faster hyperparameter tuning, which is generally how deep learning networks are optimized.

We tried quadrupling the batch size, increasing the learning rate, increasing the number of epochs by 600%, and 10× the weight decay. None of these changes gave a noticeable increase in model performance, that is, until we tried transfer learning by using the weights from an OpenPose-trained network to train on tf-pose data. More specifically, we trained a model on the full Kinetics dataset, pose estimated by OpenPose, i.e. "OpenPose-og", loaded those weights into a new model, and trained that model on the full Kinetics dataset, pose estimated by tf-pose, i.e. "tf-pose-cur". This gave a performance boost on the validation accuracy of about 2 percentage points: 26.26%. As an interesting aside, this result ties in with the results from (Carreira and Zisserman, 2017), reporting that performing transfer learning in videos, using the Kinetics dataset, has shown considerable benefits, comparable to the benefits of pre-training ConvNets on the ImageNet dataset.

We have visualized the two training processes non-transfer learning and transfer learning in fig. 7.13 and 7.14. Note that the jump in fig. 7.13 is explained by differences in the learning rate schedulers.

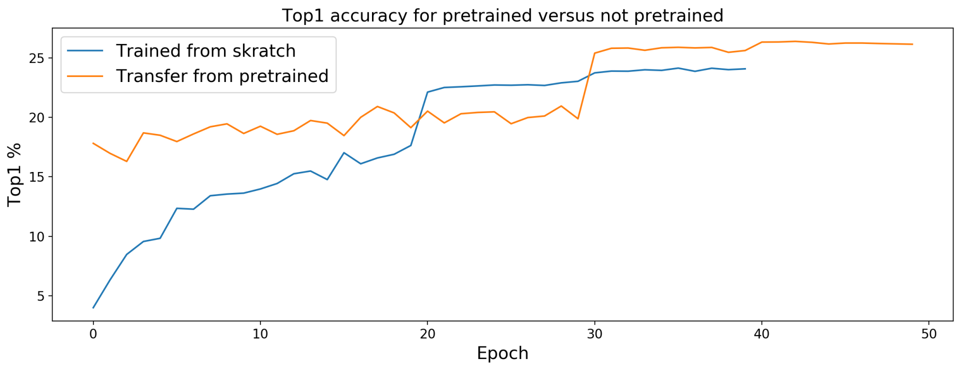


Fig. 7.13 Plotting the model's mean training accuracy when using transfer learning vs. the mean accuracy when using randomly initialized weights. The two approaches has a notable initial performance difference that seems to somewhat smooth out after the first 20 epochs. However, the approach using transfer learning gets a better accuracy.

As the loss figs. 7.14, 7.12 and 7.8 show that the ST-GCN network has a problem with overfitting. This led us to run the same experiments mentioned in the above paragraph, on this improved benchmark model, with a couple of additions: adding regularization in the form of a 0.2 dropout to each ST-GCN layer in each model and improving the learning rate scheduler by limiting the number of times the learning rate is decayed. This led to an even more improved model, shown in fig. 7.15 and 7.16, where we can clearly see an increase in the model's generalization ability, and a small accuracy gain, more specifically 26.46%.

Note that we did try with higher values of dropout, up to 0.5. However, 0.2 seemed to produce the best results.

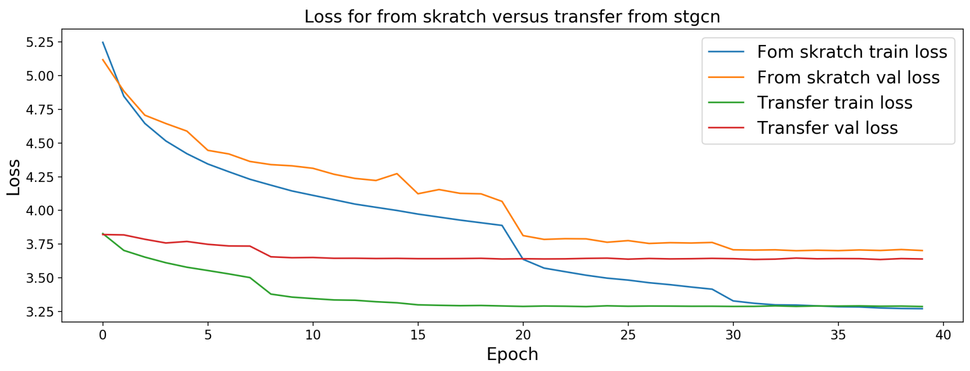


Fig. 7.14 Plotting model mean train- and validation loss when using transfer learning vs. not using transfer learning. Like fig. 7.13, the transfer learning approach is superior in the start of the training process, with the differences more or less evens out at the end. However, the mean loss on the validation set for the transfer learning approach is slightly better than the mean loss on the validation set for the non-transfer learning approach.

Experimenting with the model pre-trained using DGX

Now that we have an improved tf-pose benchmark of 26.46%, with 0.2 dropout implemented throughout the transfer learning process, we should see some improvements with regards to the rehab model's generalization ability. Our new benchmark model was used for transfer learning to train the rehab model, as in the start of this section, doing hyperparameter tuning on layer freezing and adding more layers.

This resulted in a model with the following results:

	Accuracy (%)
Validation set	82.56

Table 7.12 Rehab model performance after improving the benchmark model by implementing transfer learning×2, adding regularization, layer freezing, hyperparameter tuning, and added complexity.

This is a decrease of about 2 percentage points of validation accuracy from the previous optimal model, from table 7.9. By considering the accuracy alone we should have probably continued with the previous optimal model. However, a 2 percentage points increase is rather insignificant when taking into account the model's increased generalization potential with regularization implemented. Therefore, we choose to go forward with the “Added regularization”-model shown in fig. 7.16.

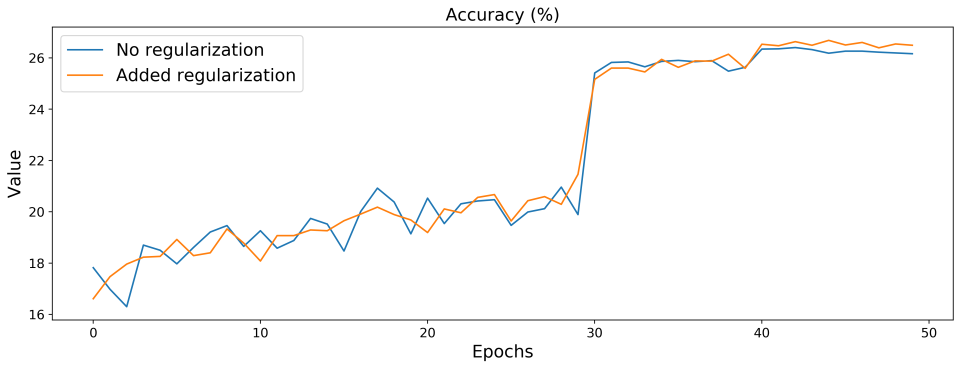


Fig. 7.15 Plotting the mean accuracy of the model during training when adding regularization in the form of 0.2 dropout to each ST-GCN layer in the model vs. the mean accuracy without added regularization. One can see that adding regularization results in a model with improved, albeit small, accuracy.

7.4 Combining the improved components

The improvement of the robot, explained in section 7.1, and the deep learning model, explained in sections 7.2 and 7.3, were two separate processes. As mentioned, decreasing the inference time on the robot to one second effected the fps negatively, going from 30 fps to 9 fps. This means that when the robot predicts on a captured video, it essentially predicts on a 9 fps video.

Recall that we imitated Yan et al.'s video pre-processing, i.e. editing each videos resolution (340 × 256), and more notably their fps (30). This means that all of the training data, validation data, and most importantly test data consists of 30 fps videos. However, to get the best possible estimate on how well the robot would perform in a production environment, we wanted to modify the test videos such that they approximate the data that the robot will actually see and base its predictions on. Accordingly, we produced a more realistic test set, tailoring its specifications to skeleton data produced by the robot, by changing the fps of all the test set videos from 30 fps to 9 fps using FFmpeg, resulting in each video having 1/3 of the original amount of frames. We name this test set *test-9fps*. Note that according to FFmpeg's homepage the videos *"will have frames dropped, in order to match the target rate"* FFmpeg (2018), i.e. it will ignore 2/3 of the frames to downsample the videos to the correct frame rate.

As has been shown in section 7.2.1, fewer frames of skeleton data usually lead to inferior results. This is to be expected because the model has less data to base its prediction on. Therefore, in an effort to approximate the original 30fps, we implemented

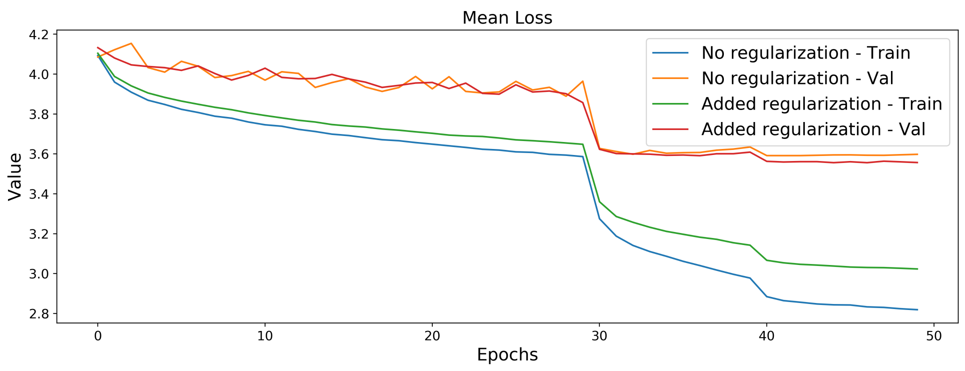


Fig. 7.16 Plotting the model's mean train- and validation loss during training when adding 0.2 dropout to each ST-GCN layer in the model vs. having no extra regularization. There is a clear closing of the gap between train- and validation loss, meaning that adding 0.2 dropout results in a model with increased generalization ability.

copying of frames. For each frame in each video in the test set, we copied it three times to increase the video's fps to 27 fps. We name this test set *test-27fps*. Note that this functionality was also implemented on the robot.

We identified our optimal Rehab model in the former section, which we now test on the two newly defined test sets: "test-9fps" and "test-27fps", and the original test set defined in section 6.5, called *test-30fps*. This is an effort to get a better sense of how well our model would perform on live recordings on the robot. The results are shown in table 7.13.

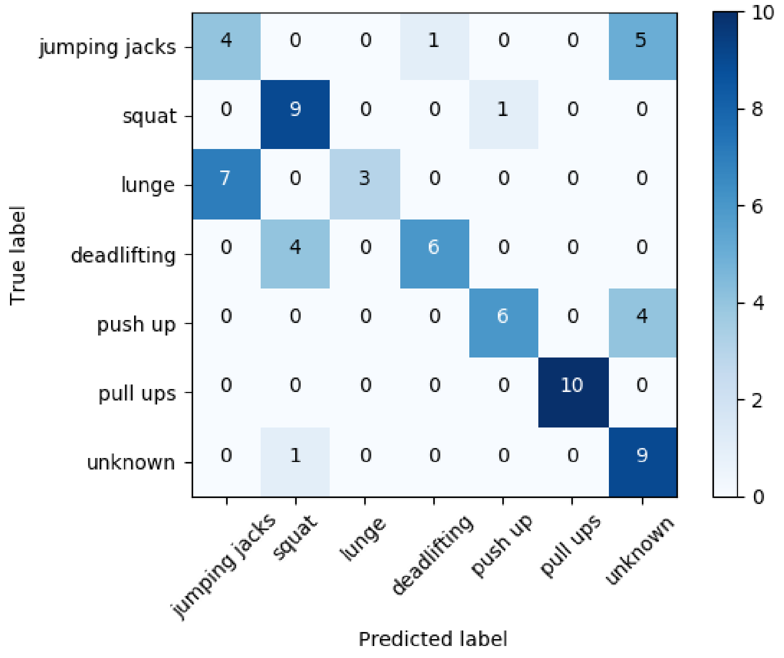
The confusion matrices generated during these test processed are shown in table 7.13.

Test set	Accuracy (%)
test-9fps	67.14
test-27fps	84.29
test-30fps	82.86

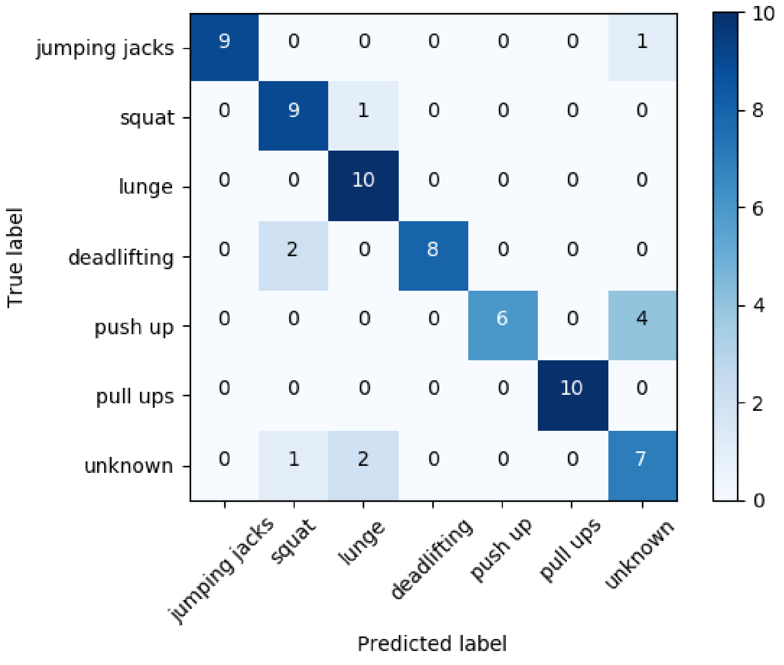
Table 7.13 The optimal Rehab model's performance on the three different test sets.

A significant increase in accuracy from "test-9fps" to "test-27fps" is to be expected, because our model is trained on 30 fps videos, and predicts on skeleton files not conveying any information about the time between the frames. Considering our model is trained on 30 fps videos, it interprets every skeleton sequence as a 30 fps sequence. Therefore, when the model is given a 9 fps video, it interprets the video as unusually fast movements, which it may not recognize. Therefore, by tripling every frame the connection between the spatial and the temporal aspect becomes almost the same as what the model is trained on.

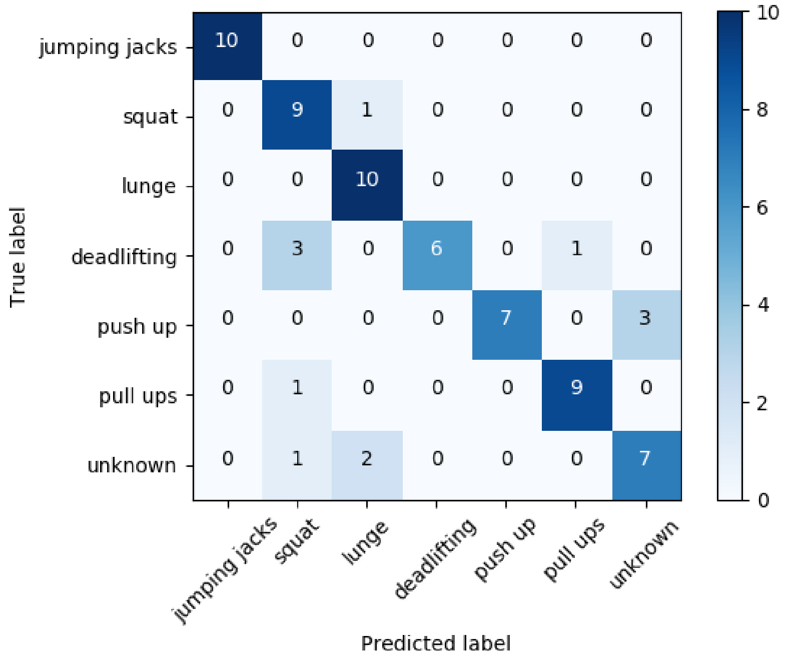
Following the same logic, one would expect the model to perform better on "test-30fps" than "test-27fps". With that said, the difference is insignificant considering the size of the test sets. However, it would have been interesting to create a larger test set to examine whether or not the performance differences between the original 30fps test set and the 27fps test set still stands.



(a) Confusion matrix showing the optimal Rehab model's performance on the test set with videos having 9 frames per second.



(b) Confusion matrix showing the optimal Rehab model's performance on the test set with videos having 27 frames per second.



(c) Confusion matrix showing the optimal Rehab model's performance on the test set with videos having 30 frames per second.

Fig. 7.17 Confusion matrices showing the optimal Rehab model's performance on the three test sets. When comparing subfigure a) vs. subfigures b) and c) one can clearly see a performance increase with a significant increase in frames per second.

7.5 Final system overview

Fig. 7.18 shows an overview of the final ROS communication in our system implemented on the robot.

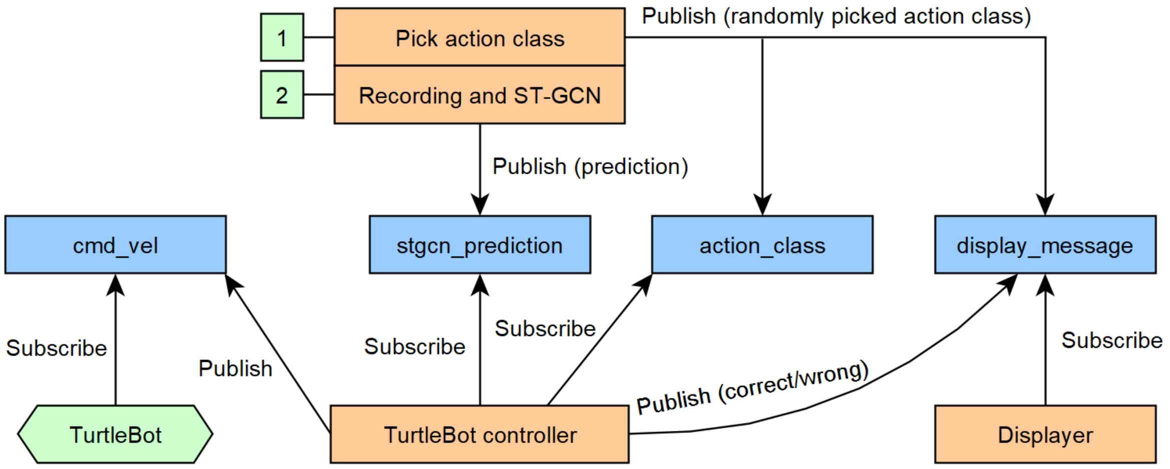


Fig. 7.18 Final ROS communication overview in our system. The orange squares are nodes, while the blue nodes are topics. Each node has its own task, such as displaying messages on the screen, controlling the robot, or running ST-GCN. The use of topics and the publish/subscribe mechanism simplifies the communication between the nodes.

The process starts with the node “Pick action class”. This node has a list of all the action classes in the Rehab dataset. It picks one of these classes randomly before publishing it to the two topics action_class and display_message. We call the randomly picked class **X**. The node “Displayer” subscribes to the display_message topic, displaying all the messages it receives on the robot’s 7 inch screen. This was done by using the Python library *PyGame* (Shinners, 2000).

By using this library we can customize the content and appearance of the messages shown on the screen, ensuring that the robot always communicates with the user. Combining PyGame with ROS makes it possible to show messages on the screen synchronously with the other processes in the system.

This way, the user is told to perform action **X**. Next, the node “Recording and ST-GCN” starts the recording which creates the skeleton files. After recording, the skeleton files are sent through our model, which outputs a prediction. This prediction is sent to the ROS topic stgcn_prediction, as described in section 6.6.2. Further, the “TurtleBot controller” node, subscribing to the topics action_class and stgcn_prediction, compares the two values on the topics to see if the model’s prediction corresponds with action **X**. If these values are the same, the message “correct” is sent to the ROS topic display_message, else the message “wrong” is sent. This way the user receives feedback on the performed movement. Further, the robot subscribes to cmd_vel, which the “Turtlebot controller” node can send velocity messages to. These messages will depend on whether the prediction corresponds with action **X**. Our final robot is shown in fig. 7.19.

7.6 Evaluation of our improved system

We have demonstrated that initializing a model’s weights by using transfer learning gives a significant boost in both overall accuracy and the time requirements to achieve these results. To obtain the optimal model for human action recognition, extensive hyperparameter tuning was performed. This lead to a validation accuracy increase of approximately 36 percentage points compared to the model from chapter 6, called *Original model*. We summarize and visualize this significant improvement in table 7.14, and fig. 7.20 and 7.21.

Model name	Dataset	Accuracy (%)
Original	Validation	48.36
Final rehab	Validation	82.56

Table 7.14 Final Rehab model performance on the validation set compared to Rehab model from section 6.5.

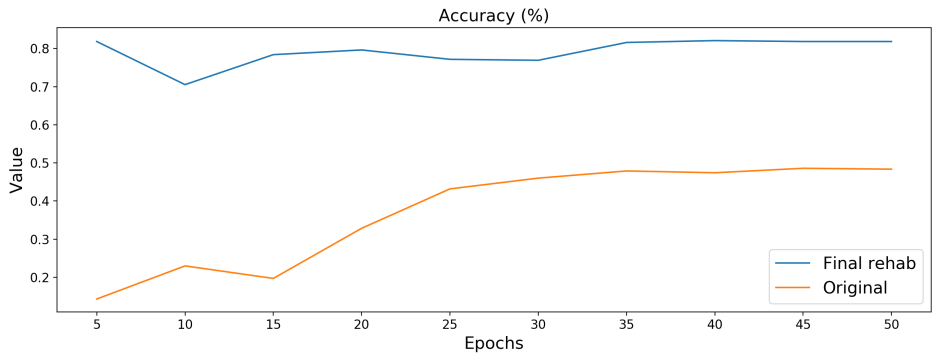


Fig. 7.20 Plotting the model's mean accuracy during training of the original model from chapter 6 vs. the final rehab model after implementing transfer learning, layer freezing, increasing model complexity and extensive research into hyperparameter tuning. One can see a significant increase in model performance.

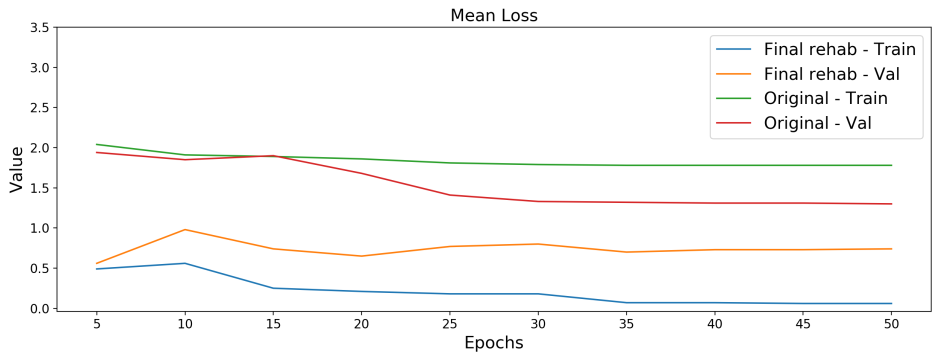


Fig. 7.21 Plotting the model's mean training- and validation loss during training of the original model from chapter 6 vs. the final rehab model after implementing transfer learning, layer freezing, increasing model complexity and extensive research into hyperparameter tuning. It is evident that the model performance is increased, while the model's generalization ability is not compromised.

While the model achieved a considerable increase in accuracy, its generalization ability was deemed questionable, as seen in fig. 7.21. We demonstrated that implementing regularization techniques, such as dropout, notably improved the model's generalization ability. Its generalization ability was proven adequate when evaluated on the test data, as shown in table 7.13 and fig. 7.17.

Lastly, we reduced the inference time from 151.1 seconds to 1 second. The first step in this process was to identify bottlenecks in our system. The largest bottleneck was OpenPose, prompting us to replace it with the faster pose estimation software tf-pose. This resulted in a decreased total waiting time, from 151 seconds to 31 seconds. Further, we modified our pipeline such that the skeleton files were created as the video was recorded, instead of after the video recording. Our new pipeline resulted in a significant improvement in waiting time, ending at 1.0 second.

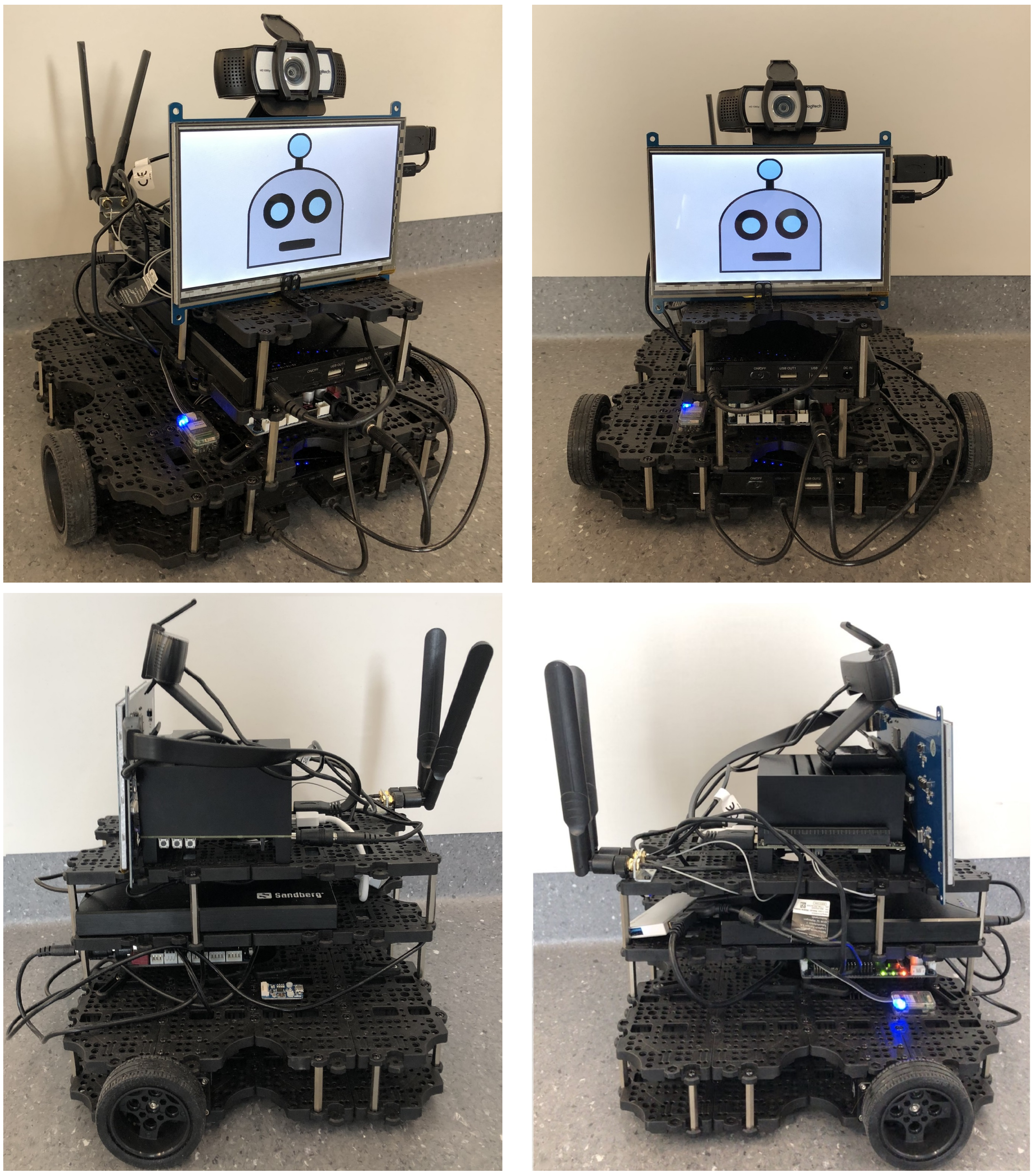


Fig. 7.19 Our robot, with the main components: NVIDIA Jetson AGX Xavier, 7" LCD screen, Logitech web camera, 2 Sandberg powerbanks (20 000 mAh each), OpenCR microcontroller and Wi-Fi antennas, built on original TurtleBot components.

Chapter 8

Conclusion

8.1 Summary

In this thesis we have combined the fields of deep learning and robotics to explore if such technologies can be used to create a robot relevant for rehabilitation purposes. In part I we introduced background theory from the two fields. initially explaining the fundamental theory behind machine learning in chapter 2, before advancing towards deep learning in chapter 3, and the specific ST-GCN that we used in our experiments. Lastly, we presented the theory behind the robotics software utilized in this project.

Part II started with training an ST-GCN model on a subset of the original data, more relevant for rehabilitation. Initially extracting the relevant action classes from the Kinetics dataset, before adding a new relevant class, creating our Rehab dataset. Next, we set up two pipelines: one for training an ST-GCN model, and one for combining this trained model with robotics for running inference on the robot. The first pipeline was set up by forking the original ST-GCN GitHub project, implementing necessary software to construct the rehab dataset, and train a model on this subset. By using the Robot Operating System, we managed to enable communication between the robot and the program running in the system - setting up the second pipeline. Chapter 6 culminates to an evaluation of the robots and its software, deeming them unsatisfactory due to the slow run time and low accuracy. This led to chapter 7, where we performed extensive research on how to decrease the running time, and increase the model's accuracy and generalization ability. The solution to decreasing the time demand was substituting the original pose estimation software with a faster version. To further improve the model, we utilized various machine- and deep learning methods, explained in chapter 2 and 3, such as transfer learning with layer freezing, and regularization techniques. The experiments led to an improved model accuracy by about 36 percentage points, and reduction in time requirement of our system by 150×. Lastly, when we combined the improved model and the robotics, we saw a difference in the data used for model training and evaluation, and the data generated on the robot, i.e. the production environment. This led us to process the test videos in such a way that it approximated the data in the production environment. As a result, the test set accuracy increased further to a final value of 84.29%. Thus proving our doubts about the model's generalization ability wrong.

The hypothesis we wanted to answer is: *By combining state-of-the-art techniques from computer vision and robotics it is possible to construct robots capable of accurate, real-time human action recognition, making it feasible to deploy such systems as robotic physical therapists.*

The accuracy of our model is not world-class compared to papers such as (Olatunji, 2018), reporting results up to 95%, for human action recognition. This comparison is albeit dubious when considering the two datasets: they used a significantly larger, more dense dataset: “Vicon physical action dataset” (Dua and Graff, 2017), having 20 action classes, with approximately 30 000 videos per class. With that said, considering our convincing increase in model performance only focusing in model optimization, and Olatunji's amount of data, it is likely we could converge towards the same performance if we experimented with data optimization. This could be everything from setting the threshold for the required number of non-empty frames > 0 for ignoring videos, to implementing additional data augmentation, such as flipping frames. Be that as it may, seen from a technical point of view, we have proven that it is feasible to construct a robot with the ability to perform accurate, almost real-time human action recognition, by combining state-of-the-art in robotics and deep learning for computer vision.

Further, seen from a domain point of view, more specifically rehabilitation, we would need to cooperate with a professional physical therapist to answer our hypothesis. There is no doubt about the system's relevance for rehabilitation in general, as explained in chapter 1. However, we lack the domain knowledge to deem the performance results relevant for professional assisting, as it is now. In other words, we are unable to conclude that it is feasible to combine state-of-the-art in robotics and deep learning into a robotic physical therapist assistant. This is left for future work, with the hope that this project can be a building block.

8.2 Future work

There are several interesting paths to explore for improving the system:

- Is it possible to implement the system on a humanoid robot, e.g. Pepper, such that the robot can physically show what movements to do? Would this be an improvement compared to showing the movement on a screen?
- Can the dataset be tailored more towards rehabilitation, either by adding or subtracting movements? Are there some action classes more relevant for certain rehabilitation, while not relevant for others? I.e. is there some natural separation of human movement? If so, would such a separation increase the model's ability?
- How would exploration of data optimization affect the model's performance? Would an increased requirement to the number of non-empty frames in a video lead to an increase or decrease in accuracy? Is it feasible to develop a natural way to pad padded videos? For example copying frames containing information and use them to fill in the empty ones?
- How could one best enable continuous learning in our system? To make the system learn after being placed in production. I.e. user-specific patterns, analogous to how e.g. Apple's personal assistant Siri improves as you use it. Perhaps active learning could be added? E.g. ask the user to perform movements that are particularly difficult for the system to classify.
- How would substituting the robot's current computational device, Jetson Xavier, with NVIDIA's newest small, computational device sporting a GPU: Jetson Nano, affect the system? Would the reduced fps lead to a significant decrease in accuracy? If so, would the size-accuracy trade-off be desirable? Maybe it is possible to copy frames such that one approximates the fps original, as we did in our system?
- With the shown increase of generalization ability using data approximating data generated on the Jetson Xavier, what effect would it have to implement this type of data processing in the pipeline from the start? What effect would it have on the transfer learning? Regularization efforts?
- Could we transform the use case from a classification problem to a regression problem? How would one go about to construct the dataset, such that the model

learns optimal techniques for human movement? Would the experiments aimed at optimizing the model's accuracy performed in this project be relevant for this use case?

References

E. V. Alliance. The Caffe Deep Learning Framework: An Interview with the Core Developers, 2016. URL https://www.embedded-vision.com/industry-analysis/technical-articles/caffe-deep-learning-framework-interview-core-developers.

J. A. Angelo. *Robotics: a reference guide to the new technology*. Libraries Unlimited, 2007.

Y. Asano, K. Okada, and M. Inaba. Design principles of a human mimetic humanoid: Humanoid platform to study human intelligence and internal body system. *Science Robotics*, 2(13):eaaq0899, 2017.

Y. Asano, K. Okada, and M. Inaba. Musculoskeletal design, control, and application of human mimetic humanoid Kenshiro. *Bioinspiration & Biomimetics*, 14(3):036011, apr 2019. doi: 10.1088/1748-3190/ab03fc. URL https://doi.org/10.1088%2F1748-3190%2Fab03fc.

P. Bajaj. Reinforcement learning, 2014. URL https://www.geeksforgeeks.org/what-is-reinforcement-learning/.

F. Bellard. ffmpeg. https://github.com/FFmpeg/FFmpeg, 2000.

Y. Bengio, P. Simard, and P. Frasconi. Learning Long-term Dependencies with Gradient Descent is Difficult. *Trans. Neur. Netw.*, 5(2):157–166, Mar. 1994. ISSN 1045-9227. doi: 10.1109/72.279181. URL http://dx.doi.org/10.1109/72.279181.

B. Boehmke. Regularized Regression, n.d. URL http://uc-r.github.io/regularized_regression#lasso. Accessed: 2018-12-12.

M. Bojarski, D. D. Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, X. Zhang, J. Zhao, and K. Zieba. End to End Learning for Self-Driving Cars. *CoRR*, abs/1604.07316, 2016. URL http://arxiv.org/abs/1604.07316.

G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.

G. Bradski and A. Kaehler. *Learning OpenCV: Computer vision with the OpenCV library*. " O'Reilly Media, Inc.", 2008.

G. P. Brennan. Managing Physical Therapy Resources: An Analogy to the Freedom of the Commons and the Need for Collective Action. *Journal of orthopaedic & sports* *physical therapy*, 42(6):486–488, 2012. URL https://www.jospt.org/doi/pdfplus/10.2519/jospt.2012.0108.

R. Brooks. Bothersome Bystanders and Self Driving Cars, 2018. URL https://rodneybrooks.com/bothersome-bystanders-and-self-driving-cars/.

M. Buhrmester, T. Kwang, and S. D. Gosling. Amazon's Mechanical Turk: A New Source of Inexpensive, Yet High-Quality, Data? *Perspectives on Psychological Science*, 6(1):3–5, 2011. doi: 10.1177/1745691610393980. URL https://doi.org/10.1177/1745691610393980. PMID: 26162106.

Z. Cao, G. Hidalgo, T. Simon, S. Wei, and Y. Sheikh. OpenPose: Realtime Multi-Person 2D Pose Estimation using Part Affinity Fields. *CoRR*, abs/1812.08008, 2018. URL http://arxiv.org/abs/1812.08008.

J. Carreira and A. Zisserman. Quo Vadis, Action Recognition? A New Model and the Kinetics Dataset. *CoRR*, abs/1705.07750, 2017. URL http://arxiv.org/abs/1705.07750.

M. Catalano, T. Leise, and T. Pfaff. Measuring Resource Inequality: The Gini Coefficient. *Numeracy*, 2, 07 2009. doi: 10.5038/1936-4660.2.2.4.

W. H. Chang and Y.-H. Kim. Robot-assisted Therapy in Stroke Rehabilitation. *Journal of stroke*, 15(3):174, 2013.

C. Chen, K. Liu, and N. Kehtarnavaz. Real-time human action recognition based on depth motion maps. *Journal of Real-Time Image Processing*, 12(1):155–163, Jun 2016. ISSN 1861-8219. doi: 10.1007/s11554-013-0370-1. URL https://doi.org/10.1007/s11554-013-0370-1.

S. Chen, K. Ma, and Y. Zheng. Med3D: Transfer Learning for 3D Medical Image Analysis. *CoRR*, abs/1904.00625, 2019. URL http://arxiv.org/abs/1904.00625.

S. Choudhury, L. Holder, G. Chin, K. Agarwal, and J. Feo. A selectivity based approach to continuous pattern detection in streaming graphs. *arXiv preprint arXiv:1503.00849*, 2015.

G. E. Dahl, T. N. Sainath, and G. E. Hinton. Improving deep neural networks for LVCSR using rectified linear units and dropout. In *ICASSP*, pages 8609–8613. IEEE, 2013. URL http://dblp.uni-trier.de/db/conf/icassp/icassp2013.html#DahlSH13.

P. Domingos. A unified bias-variance decomposition. In *Proceedings of 17th International Conference on Machine Learning*, pages 231–238, 2000.

D. Dua and C. Graff. UCI Machine Learning Repository, 2017. URL http://archive.ics.uci.edu/ml.

K. D. Eason. *Information technology and organisational change*. CRC Press, 2014.

P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The Many Faces of Publish/Subscribe. *ACM Comput. Surv.*, 35(2):114–131, June 2003. ISSN 0360-0300. doi: 10.1145/857076.857078. URL http://doi.acm.org/10.1145/857076.857078.

P. F. Felzenszwalb and D. P. Huttenlocher. Pictorial Structures for Object Recognition. *International Journal of Computer Vision*, 61(1):55–79, Jan 2005. ISSN 1573-1405. doi: 10.1023/B:VISI.0000042934.15159.49. URL https://doi.org/10.1023/B:VISI.0000042934.15159.49.

B. Fernando, E. Gavves, J. M. Oramas, A. Ghodrati, and T. Tuytelaars. Modeling Video Evolution for Action Recognition. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2015.

FFmpeg. Changing the frame rate, 2018. URL https://trac.ffmpeg.org/wiki/ChangingFrameRate.

R. A. FISHER. The Use of Multiple Measurement in Taxonomic Problems. *Annals of Eugenics*, 7(2):179–188, 1936. doi: 10.1111/j.1469-1809.1936.tb02137.x. URL https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1469-1809.1936.tb02137.x.

O. S. R. Foundation. TurtleBot, n.d. URL https://www.turtlebot.com/.

Y. Gal and Z. Ghahramani. Dropout as a bayesian approximation: Representing model uncertainty in deep learning. In *international conference on machine learning*, pages 1050–1059, 2016.

D. M. J. Garbade. Regression Versus Classification Machine Learning: What's the Difference?, 2018. URL https://medium.com/quick-code/regression-versus-classification-machine-learning-whats-the-difference-345c56dd15f7. Accessed: 2018-12-12.

X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. In *In Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS'10). Society for Artificial Intelligence and Statistics*, 2010.

M. Gori, G. Monfardini, and F. Scarselli. A new model for learning in graph domains. In *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.*, volume 2, pages 729–734 vol. 2, July 2005. doi: 10.1109/IJCNN.2005.1555942.

Q. Guan, Y. Huang, Z. Zhong, Z. Zheng, L. Zheng, and Y. Yang. Diagnose like a Radiologist: Attention Guided Convolutional Neural Network for Thorax Disease Classification. *CoRR*, abs/1801.09927, 2018. URL http://arxiv.org/abs/1801.09927.

P. Gupta. Cross Validation in Machine Learning, 2017. URL https://towardsdatascience.com/cross-validation-in-machine-learning-72924a69872f. Accessed: 2018-12-12.

A. Géron. *Hands-On Machine Learning with Skikit-Learn and TensorFlow*. O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472, 2017.

K. He, X. Zhang, S. Ren, and J. Sun. Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. *CoRR*, abs/1502.01852, 2015. URL http://arxiv.org/abs/1502.01852.

K. He, X. Zhang, S. Ren, and J. Sun. Deep Residual Learning for Image Recognition. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

S. Hochreiter and J. Schmidhuber. Long Short-Term Memory. *Neural Comput.*, 9 (8):1735–1780, Nov. 1997. ISSN 0899-7667. doi: 10.1162/neco.1997.9.8.1735. URL http://dx.doi.org/10.1162/neco.1997.9.8.1735.

B. K. Horn and B. G. Schunck. Determining optical flow. *Artificial Intelligence*, 17(1): 185 – 203, 1981. ISSN 0004-3702. doi: https://doi.org/10.1016/0004-3702(81)90024-2. URL http://www.sciencedirect.com/science/article/pii/0004370281900242.

C.-W. Hsu, C.-C. Chang, and C.-J. Lin. A Practical Guide to Support Vector Classification. Technical report, Department of Computer Science, National Taiwan University, 2003. URL http://www.csie.ntu.edu.tw/~cjlin/papers.html.

J. D. Hunter. Matplotlib: A 2D graphics environment. *Computing In Science & Engineering*, 9(3):90–95, 2007. doi: 10.1109/MCSE.2007.55.

S. Ioffe and C. Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. *CoRR*, abs/1502.03167, 2015. URL http://arxiv.org/abs/1502.03167.

E. T. Jaynes. Information Theory and Statistical Mechanics. *Phys. Rev.*, 106:620–630, May 1957. doi: 10.1103/PhysRev.106.620. URL https://link.aps.org/doi/10.1103/PhysRev.106.620.

jeffersoncountyhealthcenter. The Future of Physical Therapy for Baby Boomers. https://www.jeffersoncountyhealthcenter.org/about/news/the-future-of-physical-therapy-for-baby-boomers, 2017. [Online; accessed 2019-04-05].

Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. B. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional Architecture for Fast Feature Embedding. *CoRR*, abs/1408.5093, 2014. URL http://arxiv.org/abs/1408.5093.

H. R. Kam, S.-H. Lee, T. Park, and C.-H. Kim. RViz: A Toolkit for Real Domain Data Visualization. *Telecommun. Syst.*, 60(2):337–345, Oct. 2015. ISSN 1018-4864. doi: 10.1007/s11235-015-0034-5. URL http://dx.doi.org/10.1007/s11235-015-0034-5.

W. Kay, J. Carreira, K. Simonyan, B. Zhang, C. Hillier, S. Vijayanarasimhan, F. Viola, T. Green, T. Back, P. Natsev, M. Suleyman, and A. Zisserman. The Kinetics Human Action Video Dataset. *CoRR*, abs/1705.06950, 2017. URL http://arxiv.org/abs/1705.06950.

P. Kazanzides, G. Fichtinger, G. Hager, A. Okamura, L. Whitcomb, and R. Taylor. Surgical and Interventional Robotics - Core Concepts, Technology, and Design [Tutorial]. *IEEE Robotics & Automation Magazine*, 15(2):122–130, 2008. ISSN 1070-9932.

S. Khemaissia. OROCOS researchgate, 2013. URL https://www.researchgate.net/post/Differences_between_ROS_and_OROCOS.

T. S. Kim and A. Reiter. Interpretable 3D Human Action Analysis with Temporal Convolutional Networks. *CoRR*, abs/1704.04516, 2017. URL http://arxiv.org/abs/1704.04516.

N. Koenig and A. Howard. Design and use paradigms for Gazebo, an open-source multi-robot simulator. In *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566)*, volume 3, pages 2149–2154 vol.3, Sep. 2004. doi: 10.1109/IROS.2004.1389727.

R. Kohavi, D. H. Wolpert, et al. Bias plus variance decomposition for zero-one loss functions. In *ICML*, volume 96, pages 275–83, 1996.

A. Krizhevsky, I. Sutskever, and G. E. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012. URL http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

J. Le. A Tour of The Top 10 Algorithms for Machine Learning Newbies, 2018. URL https://towardsdatascience.com/a-tour-of-the-top-10-algorithms-for-machine-learning-newbies-dde4edffae11.

Y. Lecun, Y. Bengio, and G. Hinton. Deep Learning. *Nature*, 521(7553), 2015. ISSN 0028-0836.

A. LeNail. NN-SVG: Publication-Ready Neural Network Architecture Schematics. *Journal of Open Source Software*, 4:747, 01 2019. doi: 10.21105/joss.00747.

H.-T. Li, J.-J. Huang, C.-W. Pan, H.-I. Chi, and M.-C. Pan. Inertial Sensing Based Assessment Methods to Quantify the Effectiveness of Post-Stroke Rehabilitation. *Sensors*, 15(7):16196–16209, 2015. ISSN 1424-8220. doi: 10.3390/s150716196. URL http://www.mdpi.com/1424-8220/15/7/16196.

W. Li, Z. Zhang, and Z. Liu. Action recognition based on a bag of 3D points. In *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition - Workshops*, pages 9–14, June 2010. doi: 10.1109/CVPRW.2010.5543273.

Lumen. Understanding Social Interaction. https://courses.lumenlearning.com/boundless-sociology/chapter/understanding-social-interaction/, n.d. [Online; accessed 2019-04-07].

J. M. N. S. Madeleine Waldie, Abhinav Ayalur. Hello World! Robot Responds to Human Gestures, 2018. URL https://news.developer.nvidia.com/hello-world-robot-responds-to-human-gestures/.

S. Maji, L. Bourdev, and J. Malik. Action recognition from a distributed representation of pose and appearance. In *CVPR 2011*, pages 3177–3184, June 2011. doi: 10.1109/CVPR.2011.5995631.

M. F. Maples. Gero-Counselor Prepare: The Silver Tsunami Is Headed Our Way. *VISTAS Online*, pages 41–44, 2002.

L. Y. P. C. Matt Collier, Richard Fu. Artificial Intelligence (AI): Healthcare's New Nervous System. https://www.accenture.com/us-en/insight-artificial-intelligence-healthcare, 2017. [Online; access 2019-01-11].

L. Matthies, M. Maimone, A. Johnson, Y. Cheng, R. Willson, C. Villalpando, S. Goldberg, A. Huertas, A. Stein, and A. Angelova. Computer Vision on Mars. *International Journal of Computer Vision*, 75(1):67–92, Oct 2007. ISSN 1573-1405. doi: 10.1007/s11263-007-0046-z. URL https://doi.org/10.1007/s11263-007-0046-z.

S. May. What Is Robotics?, 2017. URL https://www.nasa.gov/audience/forstudents/5-8/features/nasa-knows/what_is_robotics_58.html.

T. M. Mitchell. *Machine Learning*. McGraw-Hill, Inc., New York, NY, USA, 1 edition, 1997. ISBN 0070428077, 9780070428072.

M. Mohri, A. Rostamizadeh, and A. Talwalkar. *Foundations of Machine Learning*. Mit Press, 2012. ISBN 9780262018258. URL http://www.jstor.org/stable/j.ctt5hhcw1.

E. Mollick. Establishing Moore's Law. *IEEE Annals of the History of Computing*, 28 (3):62–75, July 2006. ISSN 1058-6180. doi: 10.1109/MAHC.2006.45.

S. Murray. An exploratory analysis of multi-class uncertainty approximation in Bayesian convolution neural networks. Master's thesis, University of Bergen, 2018.

M. Niepert, M. Ahmed, and K. Kutzkov. Learning Convolutional Neural Networks for Graphs. *CoRR*, abs/1605.05273, 2016a. URL http://arxiv.org/abs/1605.05273.

M. Niepert, M. Ahmed, and K. Kutzkov. Learning convolutional neural networks for graphs. In *International conference on machine learning*, pages 2014–2023, 2016b.

NVIDIA. Jetson Xavier Module. https://developer.nvidia.com/embedded/buy/jetson-agx-xavier, 2018.

S. of Everyday Things. Bernoulli's Principle, 2019. URL https://www.encyclopedia.com/science-and-technology/physics/physics/bernoullis-principle.

U. B. of Labor Statistics. Physical Therapist Assistants and Aides - Job Outlook. https://www.bls.gov/ooh/healthcare/physical-therapist-assistants-and-aides.htm#tab-6, 2018. [Online; access 2019-01-11].

I. E. Olatunji. Human Activity Recognition for Mobile Robot. *CoRR*, abs/1801.07633, 2018. URL http://arxiv.org/abs/1801.07633.

OROCOS. OROCOS, 2019. URL http://www.orocos.org/.

A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in PyTorch. In *NIPS-W*, 2017.

J. A. Pérez, F. Deligianni, D. Ravì, and G. Yang. Artificial Intelligence and Robotics. *CoRR*, abs/1803.10813, 2018. URL http://arxiv.org/abs/1803.10813.

M. Pergolotti, A. Deal, B. Reeve, and H. Muss. The underutilization of occupational and physical therapy for older adults with cancer. *Journal Of Clinical Oncology*, 32 (15), 2014. ISSN 0732-183X.

P. Poli, G. Morone, G. Rosati, and S. Masiero. Robotic technologies and rehabilitation: new tools for stroke patients' therapy. *BioMed Research International*, 2013, 2013.

A. Pugh. *Robot vision*. Springer Science & Business Media, 2013.

S. Raschka. MLxtend: Providing machine learning and data science utilities and extensions to Python's scientific computing stack. *The Journal of Open Source Software*, 3(24), Apr. 2018. doi: 10.21105/joss.00638. URL http://joss.theoj.org/papers/10.21105/joss.00638.

Rock. Rock Robotics, 2019. URL https://www.rock-robotics.org/.

W. Rosamond, K. Flegal, K. Furie, A. Go, K. Greenlund, N. Haase, et al. Heart Disease and Stroke Statistics Á 2008 update Á a Report from the American Heart Association Statistics Committee and Stroke Statistics Subcommittee. *Circulation*, 117, 2008.

J. Rosen, B. Hannaford, and R. Satava. *Surgical Robotics: Systems Applications and Visions*. Springer US, 2010. ISBN 9781441911254. URL https://books.google.ht/books?id=AhB8NQEACAAJ.

ROS.org. About ROS, 2019a. URL http://www.ros.org/about-ros.

ROS.org. ROS - nodes, 2019b. URL http://wiki.ros.org/Nodes.

ROS.org. ROS - topics, 2019c. URL http://wiki.ros.org/Topics.

P. R. Ruiz. Understanding and visualizing ResNets, 2018. URL https://towardsdatascience.com/understanding-and-visualizing-resnets-442284831be8.

M. Salem Saleh Al-amri, D. N.V. Kalyankar, and D. Khamitkar S.D. Image segmentation by using edge detection. *International Journal on Computer Science and Engineering*, 2, 05 2010.

A. L. Samuel. Some Studies in Machine Learning Using the Game of Checkers. *IBM J. Res. Dev.*, 3(3):210–229, July 1959. ISSN 0018-8646. doi: 10.1147/rd.33.0210. URL http://dx.doi.org/10.1147/rd.33.0210.

F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini. The Graph Neural Network Model. *IEEE Transactions on Neural Networks*, 20(1):61–80, Jan 2009. ISSN 1045-9227. doi: 10.1109/TNN.2008.2005605. URL https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1555942.

D. Scherer, A. Müller, and S. Behnke. Evaluation of Pooling Operations in Convolutional Architectures for Object Recognition. In *Proceedings of the 20th International Conference on Artificial Neural Networks: Part III*, ICANN'10, pages 92–101, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-15824-2, 978-3-642-15824-7. URL http://dl.acm.org/citation.cfm?id=1886436.1886447.

G. Schröder, A. Knauerhase, G. Kundt, and H.-C. Schober. Effects of physical therapy on quality of life in osteoporosis patients - a randomized clinical trial. *Health and Quality of Life Outcomes*, 10(1):101, Aug 2012. ISSN 1477-7525. doi: 10.1186/1477-7525-10-101. URL https://doi.org/10.1186/1477-7525-10-101.

A. Shahroudy, J. Liu, T. Ng, and G. Wang. NTU RGB+D: A Large Scale Dataset for 3D Human Activity Analysis. *CoRR*, abs/1604.02808, 2016a. URL http://arxiv.org/abs/1604.02808.

A. Shahroudy, J. Liu, T. Ng, and G. Wang. NTU RGB+D: A Large Scale Dataset for 3D Human Activity Analysis. *CoRR*, abs/1604.02808, 2016b. URL http://arxiv.org/abs/1604.02808.

A. Shahroudy, J. Liu, T.-T. Ng, and G. Wang. NTU RGB+D: A Large Scale Dataset for 3D Human Activity Analysis. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016c.

P. Shinners. PyGame. http://pygame.org/, 2000.

C. Shorten. Data Augmentation on Images, 2018. URL https://towardsdatascience.com/data-augmentation-and-images-7aca9bd0dbe8. Accessed: 2018-12-18.

K. Simonyan and A. Zisserman. Two-Stream Convolutional Networks for Action Recognition in Videos. *CoRR*, abs/1406.2199, 2014. URL http://arxiv.org/abs/1406.2199.

S. L. Smith, P. Kindermans, and Q. V. Le. Don't Decay the Learning Rate, Increase the Batch Size. *CoRR*, abs/1711.00489, 2017. URL http://arxiv.org/abs/1711.00489.

M. Sokolova and G. Lapalme. A systematic analysis of performance measures for classification tasks. *Information Processing & Management*, 45(4):427 – 437, 2009. ISSN 0306-4573. doi: https://doi.org/10.1016/j.ipm.2009.03.002. URL http://www.sciencedirect.com/science/article/pii/S0306457309000259.

L.-Y. Song, Jialu. Artificial Intelligence and Modern Home Design. *MATEC Web Conf.*, 227:02004, 2018. doi: 10.1051/matecconf/201822702004. URL https://doi.org/10.1051/matecconf/201822702004.

T. Soo Kim and A. Reiter. Interpretable 3D Human Action Analysis With Temporal Convolutional Networks. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, July 2017.

Spartonnavex. What is an IMU? https://www.spartonnavex.com/imu/, 2015. [Online; accessed 2019-04-05].

N. Srivastava. Improving Neural Networks with Dropout. In *The Journal of Machine Learning Research*, volume 15, pages 1929–1958, 2014.

N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research*, 15:1929–1958, 2014. URL http://jmlr.org/papers/v15/srivastava14a.html.

R. Stories. CUE3: Toyota's basketball robot plays in Tokyo, 2019. URL https://robotreporters.com/cue3-toyotas-basketball-robot-plays-in-tokyo/.

J. Surowiecki. *The Wisdom of Crowds*. Anchor, 2005. ISBN 0385721706.

C. Tan, F. Sun, T. Kong, W. Zhang, C. Yang, and C. Liu. A Survey on Deep Transfer Learning. *CoRR*, abs/1808.01974, 2018. URL http://arxiv.org/abs/1808.01974.

Tesla. Tesla's autopilot, 2019. URL https://www.tesla.com/autopilot.

tf pose. tf-pose github, 2019. URL https://github.com/ildoonet/tf-pose-estimation.

theverge.com. Waymo releases app, 2019. URL https://www.theverge.com/2019/4/16/18311820/waymo-app-google-play-self-driving-car.

S. TIBKEN. Waymo CEO: Autonomous cars won't ever be able to drive in all conditions, 2018. URL https://www.cnet.com/news/alphabet-google-waymo-ceo-john-krafcik-autonomous-cars-wont-ever-be-able-to-drive-in-all-condit

L. Torrey and J. Shavlik. Transfer learning. In *Handbook of Research on Machine Learning Applications and Trends: Algorithms, Methods, and Techniques*, pages 242–264. IGI Global, 2010.

T. Truelsen, B. Piechowski-Jóźwiak, R. Bonita, C. Mathers, J. Bogousslavsky, and G. Boysen. Stroke incidence and prevalence in Europe: a review of available data. *European journal of neurology*, 13(6):581–598, 2006.

A. W. Vieira, E. R. Nascimento, G. L. Oliveira, Z. Liu, and M. F. M. Campos. STOP: Space-Time Occupancy Patterns for 3D Action Recognition from Depth Map Sequences. In L. Alvarez, M. Mejail, L. Gomez, and J. Jacobo, editors, *Progress in Pattern Recognition, Image Analysis, Computer Vision, and Applications*, pages 252–259, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. ISBN 978-3-642-33275-3.

J. Wang, Z. Liu, Y. Wu, and J. Yuan. Mining actionlet ensemble for action recognition with depth cameras. In *2012 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1290–1297, June 2012. doi: 10.1109/CVPR.2012.6247813.

M. Waskom, O. Botvinnik, D. O'Kane, P. Hobson, J. Ostblom, S. Lukauskas, D. C. Gemperline, T. Augspurger, Y. Halchenko, J. B. Cole, J. Warmenhoven, J. de Ruiter, C. Pye, S. Hoyer, J. Vanderplas, S. Villalba, G. Kunter, E. Quintero, P. Bachant, M. Martin, K. Meyer, A. Miles, Y. Ram, T. Brunner, T. Yarkoni, M. L. Williams, C. Evans, C. Fitzgerald, Brian, and A. Qalieh. mwaskom/seaborn: v0.9.0 (july 2018), July 2018. URL https://doi.org/10.5281/zenodo.1313201.

C. J. C. H. Watkins. *Learning from Delayed Rewards*. PhD thesis, King's College, Cambridge, UK, May 1989. URL http://www.cs.rhul.ac.uk/~chrisw/new_thesis.pdf.

Waymo. Waymo, 2019. URL https://waymo.com/.

N. E. West and T. O'Shea. Deep architectures for modulation recognition. In *2017 IEEE International Symposium on Dynamic Spectrum Access Networks (DySPAN)*, pages 1–6, March 2017. doi: 10.1109/DySPAN.2017.7920754.

K. Wiggers. Tesla plans to launch driverless taxi service in 2020, 2019. URL https://venturebeat.com/2019/04/22/tesla-plans-to-launch-driverless-taxi-service-in-2020/.

wire.com. Elon Musk promise, 2019. URL https://www.wired.com/story/elon-musk-tesla-full-self-driving-2019-2020-promise/.

D. H. Wolpert and W. G. Macready. No Free Lunch Theorems for Optimization. *Trans. Evol. Comp*, 1(1):67–82, Apr. 1997. ISSN 1089-778X. doi: 10.1109/4235.585893. URL https://doi.org/10.1109/4235.585893.

X. Wu, D. Xu, L. Duan, and J. Luo. Action recognition using context and appearance distribution features. *CVPR 2011*, pages 489–496, 2011.

M. S. R. G. G. P. H. F. V. J. M. F. N. J. P. Y. C. Hsuan, R. Amine. youtube-dl. https://github.com/ytdl-org/youtube-dl, 2011.

S. Yan, Y. Xiong, and D. Lin. Spatial Temporal Graph Convolutional Networks for Skeleton-Based Action Recognition. *CoRR*, abs/1801.07455, 2018. URL http://arxiv.org/abs/1801.07455.

YARP. YARP, 2019. URL https://www.yarp.it/.

W. Yin, K. Kann, M. Yu, and H. Schütze. Comparative Study of CNN and RNN for Natural Language Processing. *CoRR*, abs/1702.01923, 2017. URL http://arxiv.org/abs/1702.01923.

L. J. D. L. Yoonseok Pyo, Hancheol Cho. *ROS Robot Programming (English)*. ROBOTIS, 12 2017. ISBN 9791196230715. URL http://community.robotsource.org/t/download-the-ros-robot-programming-book-for-free/51.

yWorks GmbH. yED, n.d. URL https://www.yworks.com/.

J. Zhou, G. Cui, Z. Zhang, C. Yang, Z. Liu, and M. Sun. Graph Neural Networks: A Review of Methods and Applications. *CoRR*, abs/1812.08434, 2018. URL http://arxiv.org/abs/1812.08434.