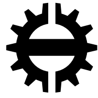


TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

RIKU HUTTUNEN

SUBCELLULAR PROTEIN LOCALIZATION IN FLUORESCENSE IMAGES USING CONVOLUTIONAL NEURAL NETWORKS

Master of Science Thesis

Examiner: Pekka Ruusuvuori
Examiner and topic approved by the Faculty Council of the Faculty of Computing and Electrical Engineering on 28th February 2018

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY
Master's Degree Programme in Information Technology
RIKU HUTTUNEN : Subcellular Protein Localization in Fluorescense Images Using Convolutional Neural Networks
Master of Science Thesis, 56 pages, 5 Appendix pages
November 2018
Major: Signal Processing
Examiner: Pekka Ruusuvuori
Keywords: CNN, deep learning, FCN, fluorescence microscopy, protein synthesis

Gene expression is manifested through the synthesis of proteins within the cell. The Cell Atlas, within the Human Protein Atlas project, provides immunofluorescence microscopy images of the cell structures aligned with images showing the stained protein of interest. The images reveal the localization patterns of the majority of proteins found in human cells. These patterns in turn can be used when studying the cellular functions related to gene expression and mutations. In the advent of deep learning methods applied to computer vision problems, machine learning algorithms can be used to categorize the localization patterns into subcellular structures.

In this thesis, two types of neural network algorithms were applied into the classification of the Cell Atlas samples from the dataset used in *33rd Congress of the International Society for Advancement of Cytometry* imaging challenge, where the task was to do multi-class multi-label classification of the images into 13 subcellular structures. The algorithms tested, namely, were Convolutional Neural Networks (CNN) and Fully Convolutional Networks (FCN). Model performance was evaluated with class-wise F_1 score. The results were promising, with CNN and FCN implementations yielding weighted averages of class-wise F_1 scores of 0.822 and 0.810 respectively. Another interesting remark is that the FCN, which outputs probability maps showing where in the image the certain class is present instead of a single probability for the whole image, learns significantly faster than the CNN, suggesting that it efficiently utilizes the spatial information in the training samples. FCN also provides more information in its outputs compared to CNN, which loses the spatial information in its outputs.

Considering the relatively small size of the dataset (20 000 samples, divided to 16 000 training samples and 4 000 testing samples), and the fact that the data is heavily imbalanced, the results are promising. More complex deep learning architectures can take advantage of millions of images, so in the future research the size of labeled dataset should be increased. Also, the quality of the labels can be questioned as they are derived from consensus between individuals without professional training.

TIIVISTELMÄ

TAMPEREEN TEKNILLINEN YLIOPISTO
Tietotekniikan koulutusohjelma
RIKU HUTTUNEN : Subcellular Protein Localization in Fluorescense Images Using Convolutional Neural Networks
Diplomityö, 56 sivua, 5 liitesivua
Marraskuu 2018
Pääaine: Signaalinkäsittely
Tarkastajat: Pekka Ruusuvuori
Avainsanat: CNN, deep learning, FCN, fluorescence microscopy, protein synthesis

Geenit ilmenevät pääasiassa proteiinisynteesin kautta. *Cell Atlas*, joka on osa *Human Protein Atlas* -projektia, tarjoaa fluoresenssimikroskopiakuvia solujen rakenteista ja niissä esiintyvistä proteiineista. Kuvat paljastavat, kuinka ihmisen soluissa esiintyvät proteiinit rikastuvat tietyissä solun rakenteissa. Näitä *paikallistumiskaavoja* (localization patterns) voidaan hyödyntää solujen ja geenien mutaatioiden tutkimuksessa. Lisääntynyt syväoppimiseen (deep learning) perustuvien metodien käyttö konenäössä mahdollistaa samojen keinojen soveltamisen myös proteiinien paikallistumiskaavojen luokitteluun solunsisäisten rakenteiden perusteella.

Tässä työssä kahdentyyppisiä neuroverkkoalgoritmeja sovellettiin Cell Atlaksen näytteiden luokitteluun. Tutkimuksessa käytetty aineisto on peräisin vuonna 2017 järjestetyn konferenssin (*33rd Congress of the International Society for Advancement of Cytometry*) kuvantamishaasteesta, jossa tehtävänä oli luokitella kuvat yhteen tai useampaan kategoriaan kolmestatoista solunsisäisestä rakenteesta. Testatut algoritmit ovat nimeltään konvoluutioneuroverkot (Convolutional Neural Networks, CNN) ja täysin konvolutionaaliset neuroverkot (Fully Convolutional Networks, FCN). Suorituskykyä arvioitiin F_1-arvolla. Tulokset olivat lupaavia: CNN ja FCN ylsivät luokkakohtaisten F_1-arvojen painotettuihin keskiarvoihin 0.822 ja 0.810. Toinen mielenkiintoinen huomio oli, että FCN, jonka ulostuloina on todennäköisyyskarttoja yksittäisen kuvalle annettavan todennäköisyysarvon sijaan, oppi huomattavasti vastaavaa CNN-toteutusta nopeammin. Tästä voidaan päätellä sen hyödyntävän tehokkaasti opetusnäytteissä olevaa spatiaalista informaatiota. Ulostulokartat myös antavat tulosten tulkitsijalle enemmän informaatiota CNN-toteutuksen yksittäisiin todennäköisyyksiin verrattuna.

Ottaen huomioon aineiston suhteellisen pienen koon (20 000 näytettä, joista 16 000 käytettiin mallin opetukseen ja 4 000 testaukseen) ja aineiston voimakkaan epätasapainon, tulokset ovat hyviä. Monimutkaisemmat syväoppimisarkkitehtuurit voivat hyödyntää tehokkaasti miljoonia näytteitä, joten tulevaisuuden työssä datan määrää tulisi kasvattaa. Myös datan annotointien laatua voidaan kyseenalaistaa, koska annotointi tapahtui nettipelaajien välisen konsensuksen pohjalta.

FOREWORD

The creation process of this thesis was begun in November 2017. Most of the hard work was executed throughout the spring 2018. A great focus was put on the implementation of a proper testbench framework for machine learning algorithms. Finally in the dark and rainy nights of November 2018 the work is blissfully finished.
I want to thank Kaisa Liimatainen and Pekka Ruusuvuori for guiding me throughout the making of this thesis. Also special thanks to Leena Latonen for giving professional insight on the biological aspects concerning the subject.
Last but not least, thank you Minna for supporting me and giving me food. Having a full-time job and trying to finish the studies at the same time was a challenging and rocky road.

Riku Huttunen, Espoo, 5 November 2018

CONTENTS

1. Introduction . . . 1
2. The Data and Biological Motivation . . . 3
   - 2.1 Genes and protein synthesis . . . 3
   - 2.2 Human Protein Atlas . . . 4
   - 2.3 Exploration of the dataset . . . 6
   - 2.4 Multilocalizing proteins . . . 6
   - 2.5 Single-cell variations . . . 8
   - 2.6 Image annotations . . . 8
3. Computational Background . . . 10
   - 3.1 Neural Networks . . . 10
     - 3.1.1 Score function . . . 10
     - 3.1.2 Activation functions . . . 11
     - 3.1.3 Model output . . . 13
     - 3.1.4 Loss function . . . 13
     - 3.1.5 Regularization . . . 14
     - 3.1.6 Back-propagation . . . 16
     - 3.1.7 Parameter updates and optimization . . . 20
     - 3.1.8 Initialization . . . 23
     - 3.1.9 Data preprocessing . . . 24
     - 3.1.10 Biological interpretation . . . 25
   - 3.2 Convolutional Neural Networks . . . 26
     - 3.2.1 Basic building blocks of a CNN . . . 27
     - 3.2.2 Transfer learning for CNNs . . . 29
     - 3.2.3 Well-known CNN architectures . . . 30
   - 3.3 Fully Convolutional Networks . . . 31
4. Methods . . . 36
   - 4.1 Classification tasks and performance evaluation . . . 36
   - 4.2 Testbench implementation . . . 37
   - 4.3 Data preprocessing . . . 37
     - 4.3.1 Class imbalance . . . 37
     - 4.3.2 FCN target generation . . . 38
   - 4.4 Hyperparameters . . . 38
   - 4.5 Network architectures . . . 40
   - 4.6 GPU Utilization . . . 41
5. Results . . . 43
   - 5.1 Comparison of the model performances . . . 43
   - 5.2 Training process monitoring . . . 46

5.3 Examples of the model outputs 48
6. Conclusions 52
Bibliography 54
A. Counts of localization labels by cell line 57
B. Network structures 58
B.1 Architecture for CNN trained from scratch 58
B.2 Architecture for FCN trained from scratch 60

TERMS AND DEFINITIONS

CE	Cross-entropy; one type of loss functions used with neural networks
CNN	Convolutional Neural Network; a neural network with at least one convolutional layer
DAPI	4,6-diamidino-2-phenylindole, a fluorescent stain used in fluorescense microscopy that binds to specific regions of DNA
DNA	Deoxyribonucleic acid, the molecular sturcture of which the hereditory material of the cells (genes) consists of
ELU	Exponential linear unit; a modification of ReLU in which the function with negative input is exponential
FC layer	Fully connected layer; a neural network layer in which every input is connected to every neuron
FCN	Fully Convolutional Network; a CNN without fully connected layers
F_1 score	A statistical measure used to evaluate the performance of machine learning models. Calculated as the harmonic mean of precision and recall.
GPU	Graphical processing unit; a computational component primarily used for graphics rendering. Capable of thousands of parallel computations making it extremely powerful tool with matrix calculations and thus useful when training neural networks.
IF	Immunofluorescence, a technique of using antibodies with fluorescent dyes to target specific biomolecules. Used in light microscopy.
ILSVRC	ImageNet Large Scale Visual Recognition Challenge; a yearly image classification competition in which a large standard image classification dataset is used
LR	Learning rate; a coefficient for controlling the magnitude of steps taken when changing the weights of a machine learning algorithm during training
MLP	Depends on the context. Multilocalizing protein; a protein that is enriched in more than one subcellular location. Multi-layer perceptron; a neural network consisting only of FC layers.

PCA	Principal Component Analysis
ReLU	Rectified linear unit; a commonly used activation function which implements the ramp function
RNA	A molecular structure similar to DNA. Carries the information in the DNA out of the cell nucleus for protein synthesis.
SCV	Single cell variations; variations in the enrichment levels of a protein between cells of the same sample
SGD	Stochastic Gradient Descent, an optimization algorithm utilizing randomly sampled batches
VGGNet	A popular CNN architecture develop in Visual Geometry Group of Oxford University
V1	The part of the visual cortex that receives the sensory input

1. INTRODUCTION

Eucaryotic cells consist of multiple different structures, or *organelles*, that have their own functions. The hereditory material inside the cell nucleus, or the *genes*, is expressed by the synthesis of proteins within the cells. Each gene has its own specific pattern on the spacial distribution of the synthesized protein.These *localization patterns* may tell whether the cells are functioning normally, i.e. whether they are healthy or not.

In this thesis, an attempt is made to classify the protein localizations with the means of computer vision. Specifically, *Convolutional Neural Networks* and *Fully Convolutional Networks* are used to model the localization using microscopy images of the cell structures and the protein signal, as well as experimental labels provided with the images in the dataset. F_1 scores for each class of localization and a weighted average of the scores are used to measure the performance of the models.

The dataset and the biological motivation are introduced in chapter 2. A couple of interesting features in the data are introduced, for example the variations of localizations between single cells in the same sample, as well as the notion that some proteins are enriched in multiple organelles while most are localized into single compartment. Chapter 3 describes the computational methods in depth. This includes the theory and background of the neural networks in general. The main features and building blocks of neural network architectures are introduced. The chapter then continues to describe the Convolutional Neural Networks, which are a special case of the general feedforward neural network. Some of the popular well-performing architectures are introduced. Finally, the Fully Convolutional Networks are considered. Chapters 4 and 5 describe the actual methods used in the study, as well as the results of the experiments. The reliability of the methods used is evaluated, as well as the relevance of the results. Lastly chapter 6 contains the conclusions drawn from the results and considers future research that could be conducted.

In summary, the results are quite promising. The models learn especially the most frequent classes rather well, and the scores of the more rare patterns are also surprisingly good. The dataset is relatively small for deep learning approaches, and especially the examples of the rarest cases are few in numbers. This calls for constitution of a larger dataset with more precise labeling information to increase the performance and further study the issues in automatic analysis of the protein

localization. Also, to more accurately model the relationships to other cellular functions, a more fine-grained hierarchy of the subcellular structures could be used. This would mean more classes for the classification task.

2. THE DATA AND BIOLOGICAL MOTIVATION

This chapter introduces the data used in the thesis. The biological background and motivation for the research are also briefly explained. The connection between gene expression and protein localization is drawn, and the significance for research and applications is outlined.

2.1 Genes and protein synthesis

Eucaryotic cells have a well defined nucleus that contains the chromosomes which consist of the hereditory material of the organisms called genes. The nucleus is surrounded by a structure separating it from the rest of the cell called *nuclear membrane*. Eucaryotic cells also contain various kinds of structures specialized to certain functions, which are called *organelles*. [9]

To put it simple, genes are sequences of DNA (deoxyribonucleic acid) that when expressed lead to synthesis of a protein. Proteins in turn are complex molecules that consist of amino acids. Protein's structure and function is defined by the sequence of the amino acids. Proteins have a multitude of different functions. Firstly, they make up the cell's structure. Enzymes are proteins that carry out the chemical reactions within the cell. Messenger proteins work as signals that coordinate the processes between different parts of the cell, as well as between cells. Transport proteins carry smaller molecules around the cell and the whole body. Andibodies bind to specific targets, e.g. viruses, to prevent their function and thus protect the cells. [24]

The synthesis from gene to protein consists of two phases. *Transcription* is a process where the DNA sequence of the gene is copied into a similar structure called RNA (ribonucleic acid). RNA type that encodes a protein is called *messenger RNA* (mRNA). mRNA carries the information out of the nucleus to cytoplasm where the second phase called *translation* takes place. In it the information in mRNA is used to assembly the protein by a complex called ribosome with assistance of *transfer RNA*, another type of RNA. [23]

Proteins that are enriched in same subcellular locations often engage in mutual protein-to-protein interactions. They also often function in similar ways. Diseases can be linked with abnormal protein localization stemming from mutations in disease-causing genes compared to healthy cells. Phenotypically related diseases

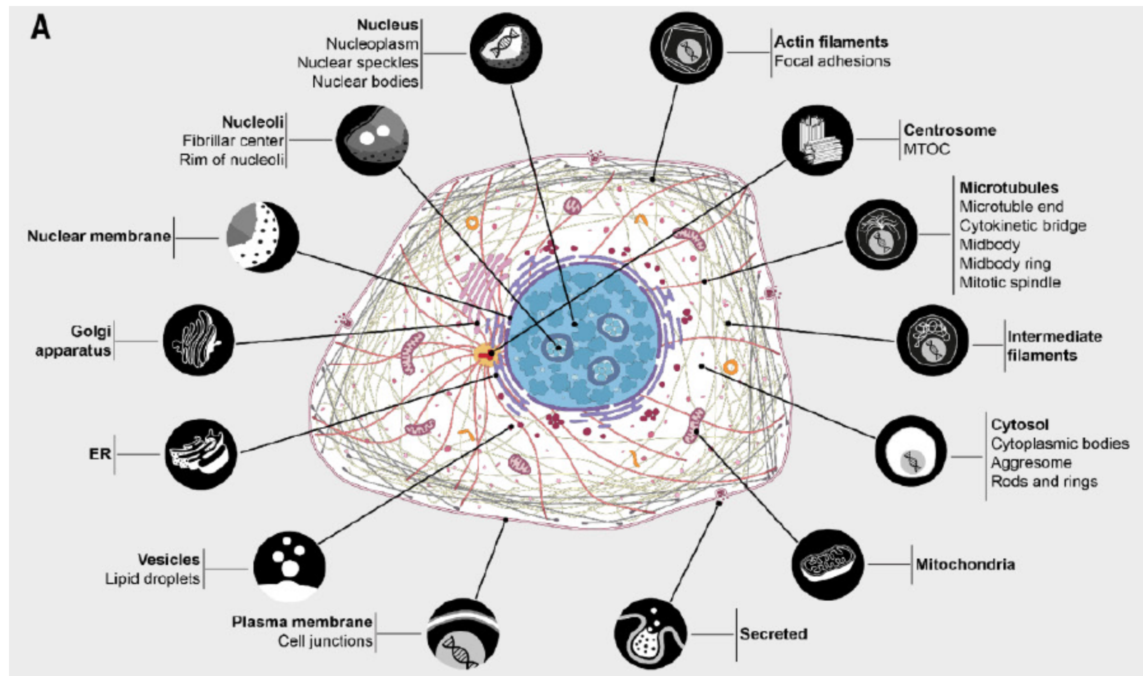


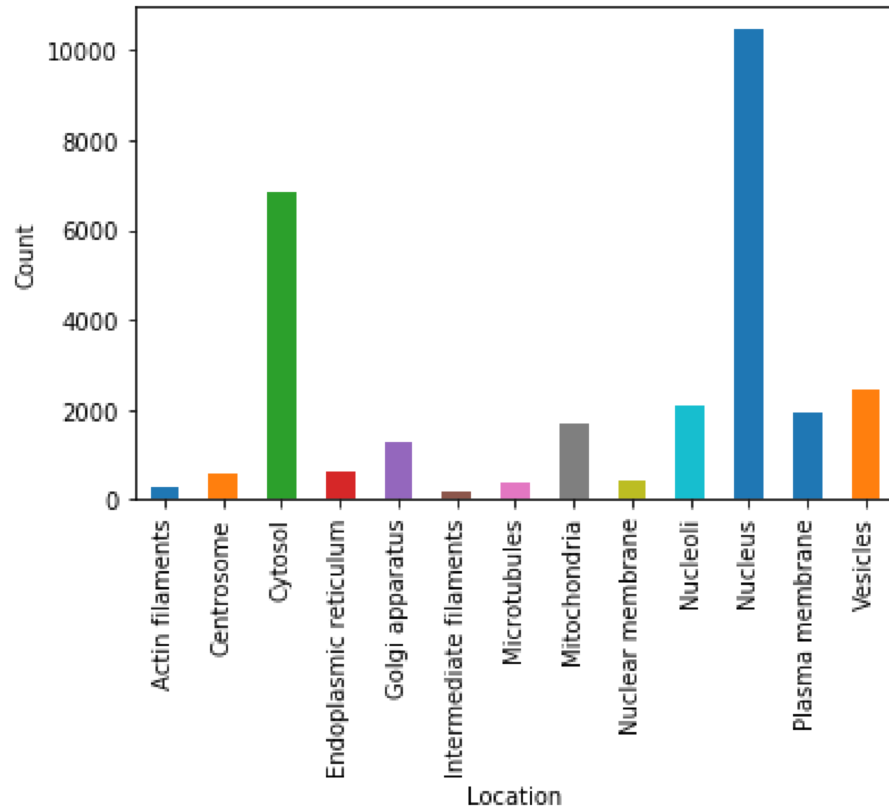
Figure 2.1: An illustration of the cell and its major organelles. The image shows the 13 organelles used as locations in the dataset used in this thesis. Some of the organelles are further divided into more specific categories which are discarded in the context of this work. Image taken from [21].

are linked by similar subcellular localization profiles. Thus observing and studying the localization patterns is an essential tool in identifying and associating genetic diseases, as well as understanding the mechanisms of how the diseases progress. [26]

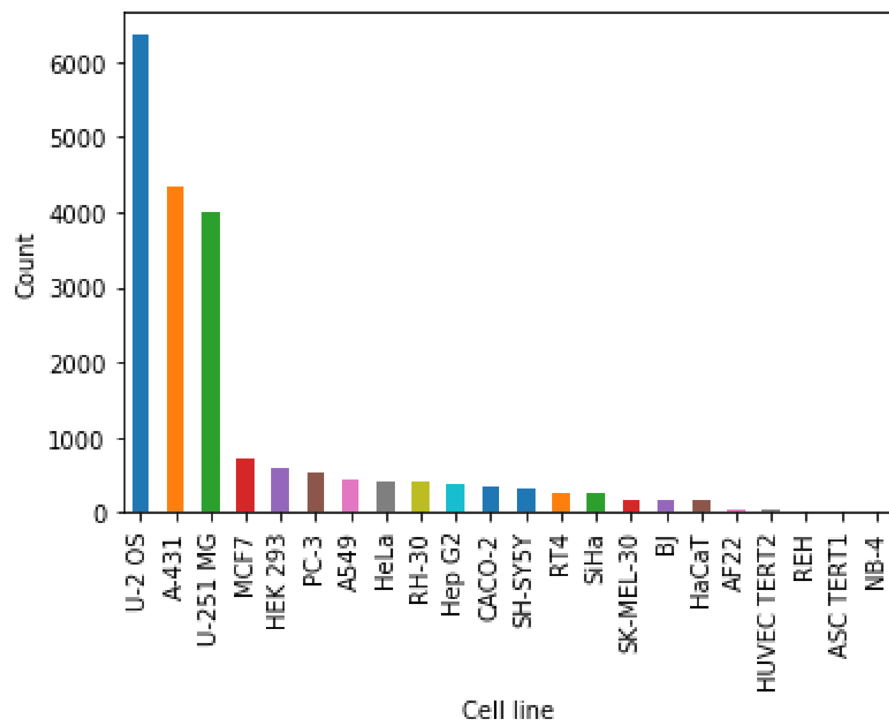
2.2 Human Protein Atlas

The Human Protein Atlas [15] is a significant effort to provide protein localization data for the human proteome. A part of it, the **Cell Atlas**, consists of *immunofluorescence* (IF) microscopy images of cells with experimental localization data. The data represents 12,003 proteins from 22 human cell lines resolved to 30 subcellular locations. Together the proteins represent 84% of all human genes that encode proteins (16,504 of 19,628 genes). Hierarchical clustering analysis based on RNA sequencing indicates that cell lines of similar phenotype are linked through a common pattern in gene expression. [35]

The proteins were stained with fluorescent colour using a total of 13,993 antibodies. Examples of images can be seen in Figure 2.3. Each antibody is designed to bind to a single target protein. The protein signals are marked with green color in the sample images. In addition to the protein of interest, reference markers were used to show the nucleus, microtubules and endoplasmic reticulum (ER). Microtubules were marked with an antitubulin antibody (red color). The nuclei were counter-



(a) The counts for each location in the dataset. Over half of the proteins of interest in the dataset localize to nucleus (10,453), and one third to cytosol (6,851). On the opposite end, only 190 proteins are enriched in intermediate filaments.



(b) The frequencies of samples from each of the 22 cell lines in the dataset. The top three cell lines U-2 OS, A-431 and U-251 MG together add up to 14,696 samples which is roughly 75% of the dataset. The other 19 cell lines have only a fraction of support compared to those. The rarest two cell lines are practically inexisting in the dataset (ASC TERT1 has nine samples, NB-4 has six samples.)

Figure 2.2: The distributions of sample labels and cell lines

stained with DAPI (blue color). ER was also stained using an antibody (yellow color, omitted from the images). [35]

2.3 Exploration of the dataset

The dataset used in this thesis originates from a competition held in *CYTO2017* conference [1]. The data for the challenge is in turn sampled from the Cell Atlas. The samples are localized into 13 major organelles of the cell depicted in Figure 2.1.

There are 20,000 samples in the dataset, each sample including the green image for the protein of interest, as well as the red, blue and yellow images for microtubules, nuclei and ER respectively. Each sample is labeled with one or more of the major organelles indicating the localization. Figure 2.2a shows the counts of each location in the dataset. It can be seen that the locations are not represented in equal portions. For example, nucleus and cytosol are by far the most common locations where the proteins are enriched. On the other hand, intermediate filaments and actin filaments are rare locations for enrichment. All localization counts to different organelles for each cell line are represented in Appendix A.

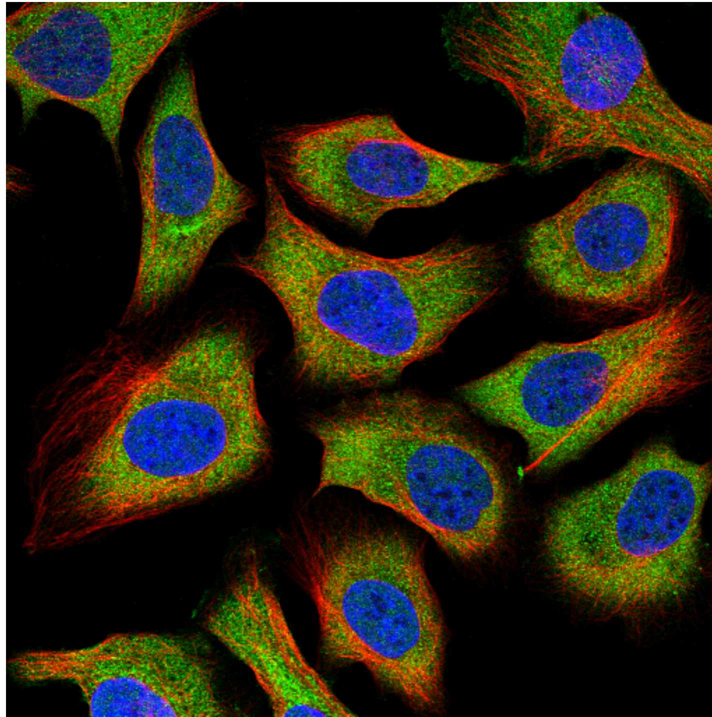
Figure 2.2b shows the frequencies of samples belonging to each of the 22 cell lines. The top three cell lines make up most of the dataset while the rarest five cell lines have only 95 samples combined. If the rare cell lines have visual features specific to them, it can be challenging for the modeling algorithms to learn them.

2.4 Multilocalizing proteins

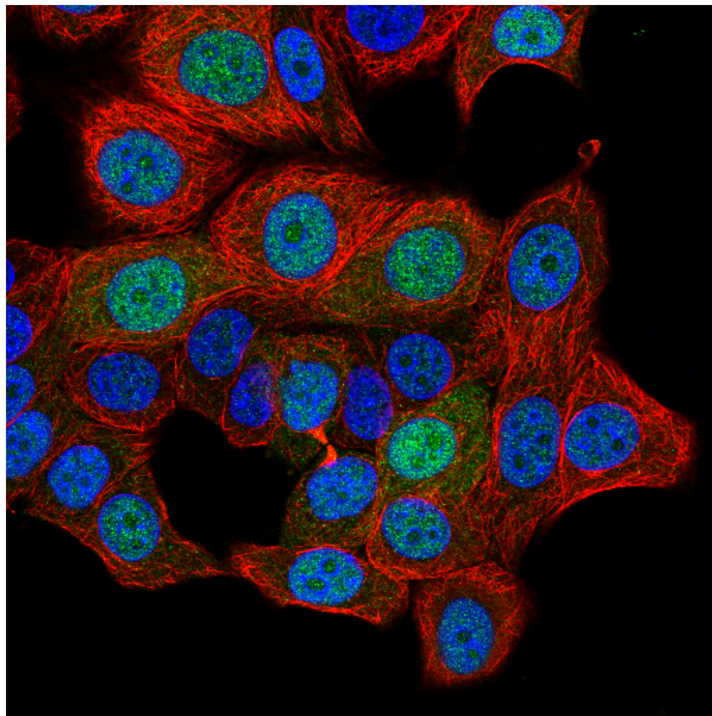
About half of the proteins are enriched in more than one subcellular location. Those proteins are called *multilocalizing proteins* (MLPs). Multi-localization increases the complexity of the cellular systems. The more locations are enriched, the more protein-protein interactions can happen due to presence of different potential interaction partners. On the other hand, different locations can affect the function and timing of sub-cellular processes in more diverse ways when the protein is enriched in multiple locations. [35]

Figure 2.3a shows an example of a MLP. In it the protein is enriched in both the nucleus and cytosol. There are differences in multilocalization between the organelles. For example, the proteome of the nucleus and plasma membrane consist mainly of MLPs, when in the case of mitochondria and ER the proteome mainly contains proteins localizing specifically to one location. Some of the MLPs have variations between cells within a sample whereas some have variations in localization patterns between different cell lines. [35]

It has been observed that MLPs influence networks of complex processes by creating interactions between organelles that have different functionality. Those in-



(a) An example of a multilocalizing protein. The sample is from U-2 OS cell line and shows protein enriched in cytosol and nucleus which can be seen from the green colour in both.



(b) Single-cell variations in protein enrichment. The sample is taken from MCF-7 cell line where the protein localizes in nucleus and nucleoli (according to the annotations, even though a clear signal can also be seen in the cytosol). Some of the nuclei have high concentrations of protein which is stained by the green colour while others have hardly any. Single-cell variations occur when individual cells are in different phases of cell cycle.

Figure 2.3: Examples of protein localizations. Image credit: Human Protein Atlas [35]

teractions also affect cell regulation which is usually credited to the nucleus. This indicates a complex network of signalling between the cellular compartments in control processes. [35]

2.5 Single-cell variations

At times there are variations in the enrichment of a protein between the cells in a single sample. These variations are called *single-cell variations* (SCV). The variations can be either in the enrichment levels, which is seen as variation in the fluorescent green intensity between cells, or the signal can be absent in some organelle in one cell and present in another cell. There is no consensus on whether and to what extent the variations are caused by stochastic events or by actual protein regulation. [35]

Figure 2.3b shows an example of variations of protein enrichment in different cells of the same sample. Some of the cells show a bright green signal where as in others the signal is almost absent. One suggested reason for the observed single-cell variation is that many of the proteins are related to specific phases of the cell cycle, and the cells in the samples are cultivated under asynchronous conditions, i.e. different cells are in different phases of the cell cycle. This hypothesis is suggested by the observation that the organelles with the most SCV also contain more proteins that are known to depend on cell cycle in their enrichment. Some of the proteins enriched are only present during cell division, e.g. mitotic spindle and cytokinetic bridge. [35]

2.6 Image annotations

The images provided in the dataset were annotated through crowdsourcing in the form of "Project Discovery", a challenge within a popular online game *EVE online* with more than 180,000 players world wide [35]. The full description of the challenge can be found in [27]. The players were provided with a brief introduction on what the different localization patterns in the images look like. Then they were shown images of the samples and their task was to label the images to 29 different structures within the 13 major organelles. Each of the samples was allowed to belong to one or more categories. The samples were not annotated if no protein enrichment signal could be seen.

The correctness of players' annotations were defined by consensus between the players. More specifically, if majority of the players chose a certain location, it was considered as the correct localization [27]. The validity and accuracy of the annotations can be questioned because the players in general did not have prior knowledge of the problem on hand. An example of annotations whose reliability can be doubted is in Figure 2.3b. The annotated locations are nucleus and nucleoli, but

there is a strong signal outside the nucleus present.

In the multi-label cases where more than one location were enriched, the locations were further categorised into *main* and *additional* by defining the relative signal strength between the annotated location and the most common location accross all cell lines. Single-cell variations were also annotated based on the intensity of the fluorescent green. [35]

3. COMPUTATIONAL BACKGROUND

In this chapter the computational concepts related to the machine learning algorithms used are introduced. The building blocks of a neural network architecture in general, as well as in the case of CNNs and FCNs in particular, are described. Some of the widely used successful CNN architectures, as well as how they can be reused, are outlined.

3.1 Neural Networks

Neural networks can be seen as a class of functions consisting of linear functions stacked on top of each other, with occasional non-linearities inbetween, in a hierarchical way in order to make up a more complex non-linear function. **Feedforward neural networks** are models that map input to output through some function $\mathbf{y} = f(\mathbf{x}; \boldsymbol{\theta})$ without any feedback connections of previous outputs. Networks that introduce feedback connections are called **recurrent neural networks** (RNNs). Feedforward networks are the basis for many different applications. For example, CNNs are also a special case of feedforward networks. [11]

The depth of a neural network is usually interpreted as the number of layers that have learnable parameters, e.g. Fully Connected (FC) and Convolutional (Conv) layers. An example of a layer without learnable parameters thus not accounted in the depth is Max Pooling layer, which only picks the maximum values from a sliding window. By learnable parameters we mean the weights that are tuned during training.

3.1.1 Score function

A score function maps the inputs into class scores. These class scores are then interpreted into confidences for each class being present or absent. Linear score function is formulated as an affine transformation of the input vector $\mathbf{x}$:

$$\mathbf{s} = W^T\mathbf{x} + \mathbf{b}, \tag{3.1}$$

where W is the weight matrix, $\mathbf{x}$ is the input vector and $\mathbf{b}$ is the bias vector. The output vector contains the score for each class. The bias can be included in the matrix multiplication by augmenting the matrix with the bias vector as the first

Table 3.1: Commonly used activation functions

Name	Function
Sigmoid	$\sigma(x) = 1/(1+e^{-x})$
tanh	$tanh(x)$
Rectified Linear Unit (ReLU)	$max(0, x)$
Leaky ReLU	$max(\alpha x, x)$
Maxout, $k = 2$	$max(w_1^T x + b1, w_2^T x + b_2)$
Exponential Linear Unit (ELU)	$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$

row of the weight matrix and prepending the feature vector with a single value of one. [20]

In a feedforward neural network, each neuron of a dense layer implements the affine transformation accompanied with an **activation function** that introduces a non-linearity to the stack of linear functions. An example of a two-layer fully-connected neural network's score function with Rectified Linear Unit activation is

$$\mathbf{s} = W_2^T max(\mathbf{0}, W_1^T \mathbf{x}). \tag{3.2}$$

The weights W_1 and W_2 are learned using e.g. **stochastic gradient descent** (SGD). The gradients are derived with chainrule using backpropagation algorithm described in section 3.1.6. [20]

3.1.2 Activation functions

An activation function introduces a non-linearity between the layers of a neural network. Without non-linearities, the overall score function of the network would still be linear and collapse into a single matrix multiplication [11, p. 168]. Common activation functions are presented in Table 3.1. From these, **Rectified Linear Unit** (ReLU) is the most frequently used at the moment because it is nearly linear, preserving most of the good optimization properties of linear functions, and is fast to compute. [11, p. 189]. Following list summarizes the properties of some of the common activation functions.

Logistic sigmoid squashes the output to range $[0, 1]$ and has a pleasant biological interpretation as the firing rate of a neuron. It has two major issues. Firstly, sigmoid function saturates at both tails to 0 and 1 respectively, and the gradient is nearly zero. During backpropagation the sigmoid's output gradient is multiplied with this local gradient so the signal flowing backwards gets 'killed'. Secondly, sigmoid outputs are not zero-centered. This can lead to zig-zagging behaviour on gradient updates which leads to slower convergence. Also, when

compared to ReLU, the exponential is more computationally expensive than max function. [11]

tanh is a scaled version of logistic sigmoid function so it has a shape similar to sigmoid but it squashes the output to $[-1, 1]$. Thus the output is zero-centered which is an improvement from sigmoid. On the other hand, the output still gets saturated, so the gradient easily gets killed as is the case with logistic sigmoid activation. [19]

ReLU simply thresholds the activation at zero. Compared to sigmoid activations, stochastic gradient descent converges a lot faster with ReLU activations, probably due to linearity and the gradients not saturating at the positive side [17]. It is also very efficient computationally. The main down side of ReLU is that gradient is always zero on the negative side. The weights may be updated in a way that the activation will be zero with any input, which will kill the gradient flow and thus the whole unit. Using a small enough learning rate will make the dying ReLUs less of an issue. [22]

Leaky ReLU is a take on fixing the dying unit issue of ReLU by changing the negative region to have a small negative slope instead of thresholding it to zero. Leaky ReLU thus has a small gradient also when the input is negative and allows gradient flow on the whole scale of input values. [22] *Parametric ReLU* is an extension of Leaky ReLU, where the slope of the unit is learned through backpropagation. This has been shown to lead to greater improvement compared to a fixed slope. [14]

Maxout unit is a generalization of the different versions of ReLU. Instead of performing a single matrix multiplication and applying the activation function to the output of it, Maxout unit has k matrix multiplications as input, and the output of the unit is the maximum value of these multiplications. Thus the maxout needs to learn k weight matrices instead of one, which means a layer with Maxout activation has k times more learnable parameters compared to other ReLUs. An example of Maxout unit with $k = 2$ is shown in Table 3.1. The main problem is that it doubles the parameters of the neuron. [20]

ELU uses an exponential function for negative values which drives the mean closer to zero enabling faster learning. Contrary to e.g. Leaky ReLUs the negative input values for ELU saturate, which decreases the variation for deactivated units making the exact negative input value less relevant. This means that ELU is capable of quantifying degree of presence of particular phenomena in the inputs, but it does not quantify the degree of absence which is a desired feature. [6]

In summary, the default choice is to use ReLU. Sigmoidal functions will generally perform worse. Extensions and generalizations of ReLU such as ELU, Leaky ReLU or Maxout may improve the performance so it is good to try some of them out, although for computational performance reasons it is best to go with ordinary ReLU if no improvements can be observed.

3.1.3 Model output

In the case of neural network classifiers, the final outputs can be interpreted as the probabilities for a sample belonging to each class. In multi-class case, if the sample is allowed to exactly one class, the final layer performs a **softmax** classification. In it, the network output is a probability distribution, i.e. the output values sum to one. If the score function's output for class i before the probability transformation is s_i, the softmax transformation (the probability of the sample belonging to class i) is [20]:

$$P(y = i \mid \mathbf{s}) = \frac{e^{s_i}}{\sum_k e^{s_k}}. \tag{3.3}$$

If the single sample can belong to multiple output classes, the problem is called **multi-class multi-label** classification. Each class probability is considered a separate binary output. In this scenario, the final layer consists of k separate logistic regression classifiers, one for each class. Thus the probability of sample s belonging to class i becomes [11, p. 65]:

$$P(y = i \mid \mathbf{s}) = \frac{1}{1 + e^{-s_i}}. \tag{3.4}$$

3.1.4 Loss function

To be able to train the model, we need to have a way to measure how well the model is classifying the sample. The **loss function** or **cost function** is a mapping of the model output to a real-valued cost assigned for the difference between the output and the actual label of the sample. A common technique to fit neural network models is **Maximum Likelihood Estimation** (MLE) [11, p. 174]. The ML estimator for a conditional distribution of Y given X is obtained by

$$\boldsymbol{\theta}_{ML} = \arg\max_{\theta} P(\mathbf{Y} \mid \mathbf{X}; \boldsymbol{\theta}) \tag{3.5}$$

$$\boldsymbol{\theta}_{ML} = \arg\max_{\theta} \prod_{i=1}^{n} P(\mathbf{y_i} \mid \mathbf{x_i}; \boldsymbol{\theta}). \tag{3.6}$$

Taking a product over multiple probabilities may lead to issues like numeric

underflow. Taking a logarithm of the estimate does not change the arg max value and transforms the product into a sum which is more convenient [11, p. 130]:

$$\boldsymbol{\theta}_{ML} = \arg\max_{\theta} \sum_{i=1}^{n} logP(\mathbf{y_i}|\ \mathbf{x_i}; \boldsymbol{\theta}). \tag{3.7}$$

Consequently, the ML estimate can be obtained by minimizing the negative log-likelihood which corresponds to minimizing the cross-entropy between the empirical distribution of the training data and the distribution of predictions of the model [11, p. 130]. Thus, for a single sample i the loss for obtaining the ML estimate is the negative log-likelihood:

$$L_i = -logP(y_i \mid x_i) \tag{3.8}$$

In the case of softmax output this becomes

$$L_i = -log\frac{e^{s_{y_i}}}{\sum_j e^{s_j}}. \tag{3.9}$$

If the model is performing multi-label classification (separate binary outputs for each class) the single-sample loss is the mean of the binary loss for each class:

$$L_i = -\frac{1}{K}\sum_{j=1}^{K} y_{ij}\log(\sigma(s_j)) + (1 - y_{ij})\log(1 - \sigma(s_j)), \tag{3.10}$$

where K is the number of classes, y_{ij} is the label of jth class for ith sample and s_j is the score of the corresponding label before probability transformation. Note that y_{ij} is either 0 or 1 so one of the terms within the sum will always be zero.

The overall loss L is calculated as the average of the single-sample loss L_i for each sample in the data. This is called the **data loss**. To prevent the weights growing too large, an additional penalty may be added. This is the **regularization loss** $R(W)$. The overall loss is then calculated as

$$L = \frac{1}{N}\sum_{i=1}^{N} L_i + \lambda R(W), \tag{3.11}$$

where λ is a scaling coefficient for the strength of regularization. [20]

3.1.5 Regularization

According to Goodfellow et. al. [11, p. 117]:

> *Regularization is any modification we make to a learning algorithm that is intended to reduce its generalization error but not its training error.*

In other words regularization is done to prevent overfitting. There are various kinds of regularization strategies. Constraints can be put straight on the weights, e.g. a hard limit on the maximum value of a weight. Some strategies add an extra term to the loss function that penalizes the weights some way. A third class of regularization techniques are ensemble methods that combine multiple models by e.g. averaging their outputs to reduce the variance of the overall estimator. In practice, the best fitting models in deep learning are large models that are regularized properly to prevent overfitting, instead of putting emphasis on finding the right capacity for the model with less emphasis on regularization. The overfitting issues are easier to manage with tuning regularization than with tuning the overall capacity of the model. [11, p. 224]

Parameter norm penalties add a regularization loss term into cost function. **L2 norm regularization**, also known as *Ridge regression* is a simple and commonly used penalty. It drives the weights to be smaller by adding regularization term $\frac{1}{2}\lambda w^2$ to the loss function L. The L2 norm is multiplied by $\frac{1}{2}$ so that the gradient simplifies to λw. L2 regularization has an intuition of preferring small, diffuse weights and assigning heavy penalty to large weights. **L1 regularization**, also called *Lasso regression* adds $\lambda|w|$ for each weight to the regularization loss instead of squared case of L2. This has the consequence of weight vectors becoming sparse instead of diffuse. This means L1 regularized model ends up using a sparse subset of the inputs which makes the model more tolerant for noise. **Elastic net regularization** is a method that uses L1 and L2 penalties combined. [11]

Dropout is a simple and effective regularization technique for neural networks. During training for each step separately, each neuron is kept with a probability p that is a hyperparameter for dropout. Otherwise the particular neuron is set to zero. This can be interpreted as subsampling the network during training and the final result as consisting of an ensemble of all these subsampled networks. A common value for dropout is $p = 0.5$. During testing, dropout is not executed, and the activations need to be scaled with p so that they match the expected outputs for similar input at training time [31]. Usually test time performance is critical for applications and a modified version called *inverted dropout* is used instead, where the scaling is done at training time. [11, p. 261]

Machine learning models tend to overfit when there is not enough data. One way to address this issue is to augment the dataset by artificially creating new data from the existing. **Augmentation** is particulary effective on image data: the model needs to be invariant for transformations, and thus the data can be augmented by introducing various random transformations into original images for example by rotating, shifting and zooming. [11]

Batch normalization is a technique where the activations are normalized to

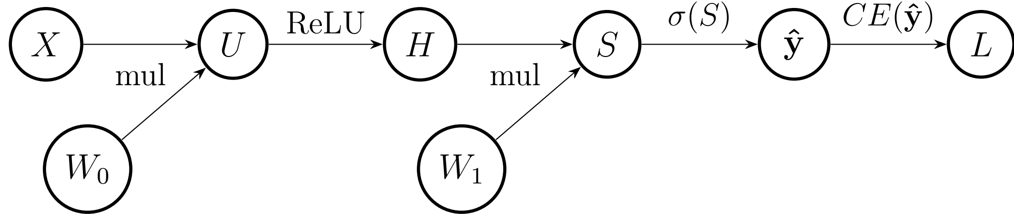


Figure 3.1: The computational graph for a two-layer MLP. ReLU activation is used in the intermediate layer, and logistic sigmoid in the output. The loss L is sum of cross-entropies (CE) for each individual binary output. I.e. the network performs multi-class multi-label prediction. The weights for the layers are W_0 and W_1, X is the input, $\hat{\mathbf{y}}$ are the predicted probabilities for each class, U are the weighted sums before ReLU nonlinearity, H are the activations of the first layer and S are the logits before sigmoid transformation.

unit Gaussian distribution before doing the actual training step for each minibatch. In the implementation, this is done by adding a batch normalization layer between FC or Conv layer and the non-linearity. With batch normalization, higher learning rates can be used to speed up the training process. The normalization of the activations prevents small changes of weights to amplify into large changes in activations during the optimization. This in turn prevents the optimization getting stuck on saturated gradients because of large activations. Also, batch normalization makes the backpropagating gradients resilient to the parameter scale, thus preventing gradient explosion because of too large weights. [16]

Batch normalization also acts as a regularization technique because the activations for a given training sample are affected by other samples in the minibatch, thus rendering the activations non-deterministic. The use of batch normalization reduces the need for dropout, possibly even to a degree where it can be removed altogether [16]. Batch normalization is especially relevant with deep neural networks, because the size of weights tends to exhibit a multiplicating effect from layer to next.

3.1.6 Back-propagation

A feedforward neural network takes $\mathbf{x}$ as an input, which is then propagated through each layer to produce the output $\hat{\mathbf{y}}$. When training the network, $\hat{\mathbf{y}}$ is further propagated through the loss function $L(\boldsymbol{\theta})$ to produce the scalar loss of the network. To update the parameters $\boldsymbol{\theta}$, we need to compute the gradient of the loss with respect to each parameter $\nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta})$. This can be done efficiently using an algorithm called **back-propagation**. Then, an optimization algorithm such as SGD is used to perform the actual updates using the gradients. [11, p. 200]

To clarify how the back-propagation works, neural networks can be expressed as computational graphs. There are different ways to represent the graphs. In this

thesis, each node of the graph is a variable, such as weights, input or output. New nodes are created from existing through **operations**, such as sum, dot product or scaling. If node x is used in computation of node y, an edge is drawn between the nodes. The operation is annotated with the edge [11, p. 201]. Figure 3.1 shows the computational graph for a MLP with two learnable layers. On forward pass the network computes function

$$\hat{\mathbf{y}} = \sigma(W_1^T \max(0, W_0^T X)). \tag{3.12}$$

The loss used is binary cross-entropy (CE). This means that each output is considered as a separate binary label. The overall loss of the model is the sum of the individual losses. If the sigmoid output for ith class is $\hat{y}_i$ and the corresponding true label is y, the binary cross-entropy is defined as

$$CE(y_i, \hat{y}_i) = -y_i log(\hat{y}_i) - (1 - y_i) log(1 - \hat{y}_i). \tag{3.13}$$

Then, the gradient for the overall loss with respect to the parameters can be expressed with **chain rule** using the gradient of the loss with respect to the local output q and the gradient of q with respect to the weights [11, p. 202]:

$$\frac{\partial L}{\partial w_{jk}^m} = \frac{\partial L}{\partial q} \frac{\partial q}{\partial w_{jk}^m}, \tag{3.14}$$

where w_{jk}^m is the weight of layer m connecting input node j to output node k.

Looking at Figure 3.1, using the chain rule we can compute the gradient of the loss with respect to the weights connecting the hidden units H to the outputs:

$$\frac{\partial L}{\partial w_{jk}^1} = \frac{\partial L}{\partial \hat{y}_i} \frac{\partial \hat{y}_i}{\partial s_i} \frac{\partial s_i}{\partial w_{jk}^1}. \tag{3.15}$$

The partial derivatives needed in the calculation are

$$\frac{\partial L}{\partial \hat{y}_i} = \frac{-y_i}{\hat{y}_i} - (1 - y_i)(-1)\frac{1}{1 - \hat{y}_i} = \frac{\hat{y}_i - y_i}{\hat{y}_i(1 - \hat{y}_i)} \tag{3.16}$$

$$\frac{\partial \hat{y}_i}{\partial s_i} = \frac{e^{-s_i}}{(1 + e^{-s_i})^2} = \sigma(s_i)(1 - \sigma(s_i)) = \hat{y}_i(1 - \hat{y}_i) \tag{3.17}$$

$$\frac{\partial s_i}{\partial w_{jk}^1} = h_j. \tag{3.18}$$

The partial derivative of the loss with respect to the unnormalized logits s_i is needed in the next step when computing the first layer's weights:

$$\frac{\partial L}{\partial s_i} = \frac{\hat{y}_i - y_i}{\hat{y}_i(1 - \hat{y}_i)} \hat{y}_i(1 - \hat{y}_i) = \hat{y}_i - y_i. \tag{3.19}$$

Table 3.2: Common local gradient functions and patterns in back-propagation

Function	Gradient	Pattern
$z = x + y$	$\frac{\partial z}{\partial x} = 1$	Gradient distributor
$z = max(x, y)$	One for the max input, zero for others	Gradient router. The gradient flows as is to the maximum input, and others are zero.
$z = x * y$	$\frac{\partial z}{\partial x} = y$	Gradient switcher. The gradient for x is the output gradient multiplied by y and vice versa.
$z = ax$	$\frac{\partial z}{\partial x} = a$	Scaler.

Finally, the equation for the gradient for the second layer's weights simplifies to

$$\frac{\partial L}{\partial w^1_{jk}} = \frac{\partial L}{\partial s_i}\frac{\partial s_i}{\partial w^1_{jk}} = (\hat{y}_i - y_i)h_j. \tag{3.20}$$

Then, the gradient on the weights connecting the input to the hidden layer is defined as

$$\frac{\partial L}{\partial w^0_{jk}} = \frac{\partial L}{\partial u_k}\frac{\partial u_k}{\partial w^0_{jk}}. \tag{3.21}$$

The equations needed in the computation are

$$\frac{\partial s_i}{\partial h_k} = w^1_{ki} \tag{3.22}$$

$$\frac{\partial h_k}{\partial u_k} = \begin{cases} 1 & u_k > 0 \\ 0 & u_k \leq 0 \end{cases} \tag{3.23}$$

$$\frac{\partial L}{\partial u_k} = \sum_i \frac{\partial L}{\partial s_i}\frac{\partial s_i}{\partial h_k}\frac{\partial h_k}{\partial u_k} = \sum_i \begin{cases} (\hat{y}_i - y_i)(w^1_{ki}) & u_k > 0 \\ 0 & u_k \leq 0 \end{cases} \tag{3.24}$$

$$\frac{\partial L}{\partial w^0_{jk}} = \frac{\partial L}{\partial u_k}\frac{\partial u_k}{\partial w^0_{jk}} = \sum_i \begin{cases} (\hat{y}_i - y_i)(w^1_{jk})(x_j) & u_k > 0 \\ 0 & u_k \leq 0 \end{cases} \tag{3.25}$$

The gradients can be thought to flow backwards from the output towards the input. The gradient of the output with respect to itself is one. The final node computes the loss. This way, the gradient of overall loss with respect to any intermediate parameter in the graph can be computed by recursively applying the backward flow of simple local gradients. [11]

The operations that transform the nodes to new ones can be arbitrarily complex, as long as the local gradient can be computed. By taking advantage of this feature,

a library of meaningful building blocks can be created. For example, the sigmoid function in Table 3.1 has a relatively simple derivative $\frac{d\sigma(x)}{dx} = \frac{e^{-x}}{(1+e^{-x})^2} = (1 - \sigma(x))\sigma(x)$. Gradients for some of the common operations used in neural networks and the patterns they represent in the backward flow are represented in Table 3.2.

When it comes to actual implementations of the deep learning algorithms, the computation is done using vectors, matrices and tensors (multidimensional arrays). The chain rule for the gradient on any parameter $\boldsymbol{\theta}_i$ can be expressed in vector notation as

$$\nabla_{\boldsymbol{\theta}_i} L = \left(\frac{\partial \mathbf{q}}{\partial \boldsymbol{\theta}_i}\right)^T \nabla_{\mathbf{q}} L \tag{3.26}$$

In the vectorized form, the local gradients will be the Jacobian matrices. I.e. $\frac{\partial \mathbf{q}}{\partial \boldsymbol{\theta}_i}$ will contain derivative of each element in $\mathbf{q}$ with respect to each element in $\boldsymbol{\theta}_i$, which contains both weights and inputs. In practice, these Jacobian matrices for layers with possibly thousands of inputs and outputs become huge. Because the operation is elementwise, the Jacobian matrix will become a diagonal matrix. Thus, only the element-wise values need to be computed [11, p. 203]. From equation 3.26 we can see that gradient of any layer's parameters can be computed by multiplying the gradient of the loss with respect to the layer output by the Jacobian matrix. The back-propagation is then performed by doing this recursively for each operation in the computational graph. [11, p. 204]

At branches, gradients are added. I.e. if one node's output goes to multiple nodes, the local gradients from the multiple nodes are summed as the gradient for the branching node. Using the Jacobian matrix in computation takes automatically care of this. Implementations of deep learning frameworks, e.g. Tensorflow, use this kind of computational graph structure, with each node and layer implementing the *forward()* and *backward()* APIs. An example implementation for a multiplication gate would be [20]:

```
class MultiplyGate(object):

    def forward(x, y):
        z = x*y
        self.x = x  # These are needed for backprop
        self.y = y
        return z

    def backward(dz):
        dx = self.y * dz  # [dz/dx * dL/dz]
        dy = self.x * dz  # [dz/dy * dL/dz]
        return [dx, dy]
```

The popular frameworks automatically take care of the computation of gradients making it easy to write code for algorithms that are rather complex underneath.

3.1.7 Parameter updates and optimization

After the gradients of the loss with respect to the parameters, which tell in which direction in the parameter space the loss is increasing the fastest, have been computed by the back-propagation algorithm it is time to update the weights. The most straightforward way to do this is the **gradient descent** in which the weights are updated by taking a small step in the direction of the negative gradient. The size of the steps taken is called **learning rate** (LR). The issue with gradient-based methods on neural networks is that the nonlinearities cause the commonly used loss functions to become non-convex. This means that the convergence of the algorithms can not be guaranteed and the initialization of the weights becomes crucial. [11, p. 173]

There are two major points of interest concerning the optimization process:

1. Does the process converge, and how fast?
2. Is the optimum found local or global? How good is the local optimum?

If the learning rate is too high, the optimization will not converge. If it is too low, convergence will take ages. If the loss changes fast in one direction and slowly in another, steepest descent will progress in a slow zigzag motion. [11, chapter 8.2]

Traditional gradient descent optimization can get stuck in a local optimum. It also gets stuck in a saddle point, where the gradient is zero. In high-dimensional space local minima are not that common, because the minimum is along all dimensions, and if there are millions of dimensions, it is not probable that all of them are met. Instead, saddle points are more common. The problem also exists near the saddle point, where gradient is very small. [7]

Generally the optimization process, or training, goes as follows:

1. Sample a batch of data
2. Forward propagate through the network, compute overall loss
3. Backpropagate and calculate the parameters' gradients
4. Update the parameters by using the gradients
5. Compute the validation loss. If the stopping condition is satisfied, end training. Else go to 1.

The stopping condition can be e.g. that maximum number of iterations has been performed, or that the validation loss has not decreased in certain amount of last iterations.

The batch size is an important hyperparameter of the algorithm. Using the whole training set as the batch yields the most accurate estimate for the gradient but the computation is very expensive. On the other extreme, only one sample is used as a batch. In this approach, the gradient will be very inaccurate, and the parallel processing capabilities of the computers may not be utilized efficiently. Usually the batch size is chosen somewhere inbetween. Normally with image data the proper batch size to use is as large as can be fit into the memory of the GPU at once so that the whole computation of the batch can be efficiently parallelized. In that way the gradients will be as accurate as possible without slowing down the runtime. [11, p. 276]

The minibatch should be sampled randomly because if the samples are not independent the gradient estimate will be biased. For example, if we have some measurements on patients as the data, those measurements can be ordered chronologically by patient in the dataset. Then if we do not shuffle the data for sampling, each batch can consist of only one or two patients' data and the estimate on the whole training data will be extremely biased. This can drastically decrease the performance of the model. [11, p. 277]

Minibatch gradient descent is called **stochastic gradient descent** (SGD) because each batch is sampled randomly from the training set. This introduces noise to the process which acts as a means of regularization. In SGD, updates are performed as follows:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \alpha \nabla J(\boldsymbol{\theta}_t), \tag{3.27}$$

where $\boldsymbol{\theta}_t$ are the parameter values on the iteration, α is the learning rate and $\nabla J(\boldsymbol{\theta}_t)$ is the gradient on the parameters. [11, ch. 8.2]

Often it is reasonable to gradually decay the learning rate α over time. This can be done as a function of the number of iterations performed. For example in Keras library, one can define this by setting the *decay* parameter of an optimizer. Another popular method is to decay α by a certain factor every time the training loss stops decreasing. [5]

The issue with standard SGD is that it can get stuck in local minima or saddle points. Another problem is that large differences in the steepness of the gradient in different directions lead to ineffective updates. This is because the step size is the same in every direction even though larger steps should be taken in the direction of small gradient because the distance to the "bottom of the hill" is longer in that direction. [11, ch. 8.2]

A simple solution to the abovementioned issues is a technique called **Momentum** [11, p. 292]. When using Momentum, an exponentially decaying running mean of negative gradients v is calculated along the process. The calculation given in equation 3.28 has a hyperparameter ρ which acts as kind of a friction so that v will not grow too high. Then, the update is formed in equation 3.29.

$$v_{t+1} = \rho v_t - \alpha \nabla J(\boldsymbol{\theta}_t) \tag{3.28}$$

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + v_{t+1}. \tag{3.29}$$

In the case of local minima and saddle points, the velocity v keeps the weights updating as v starts slowing. In the case of poor conditioning which would lead to zigzag-kind of effect in updates, v keeps on pushing the updates down the hill. [11, p. 293]

Nesterov Momentum is a variant of Momentum, in which the velocity is applied first to the weights as shown in equation 3.30. Only after that the gradient is evaluated and applied. Equation 3.29 defines the actual parameter update. Unfortunately Nesterov Momentum brings improvements only on convex problems and does not improve the convergence in non-convex problems like neural networks [11, p. 296].

$$v_{t+1} = \rho v_t - \alpha \nabla J(\boldsymbol{\theta}_t + \rho v_t) \tag{3.30}$$

In addition to SGD and its variants described above, there are a handful of algoriths with adaptive learning rates like *AdaGrad, RMSProp* and *Adam*. The adaptive methods are attractive alternatives because learning rate is one of the most problematic yet important hyperparameter to tune. [11, p. 303]. One downside is that the adaptive methods have been claimed to lead to worse generalization performance [36]. Their strength is that they save time and effort because the learning rates need not be tuned.

AdaGrad adapts learning rate by scaling down the parameters by the sum of squared values of the previous gradients. Parameters that have steep gradients will be changed faster than those that are changing slowly. In deep neural networks, AdaGrad can lead in premature and too large decrease in the learning rates. [11, p. 303]

In **RMSProp**, AdaGrad is modified by using exponentially weighted moving average in the gradient calculation. This leads to a decay in older gradients so that the algorithm can converge more rapidly on good conditions. RMSProp is an effective optimization algorithm that is currently often used. [11, p. 304]

Adam is one more adaptive algorithm that is quite like a combination of RMSProp

and momentum with a couple of exceptions. It uses both the first moment (i.e. the gradient itself) and the second moment (i.e. the gradient squared, or the uncentered variance, as in RMSProp). Adam also performs a bias correction step on the both moments which RMSProp is lacking. Hyperparameters of Adam are usually quite robust by default, although initial learning rate change may sometimes improve the performance. [11, p. 306]

At the moment, there is no consensus on which optimization algorithms are objectively the best. Many times, the choice depends on the user's earlier experience so that the hyperparameters of the algorithm are already familiar. [11, p. 308]

3.1.8 Initialization

Deep learning models are trained with iterative methods. This means that an initial point in the parameter space needs to be defined in the beginning of the training. As mentioned in section 3.1.7, the objective for optimization (i.e. loss function) becomes non-convex for the nonlinearities in a neural network. This often means that the convergence of the algorithm is very sensitive on parameter initialization. With bad initialization, the algorithm may not converge at all. The initialization also affects on the speed of convergence, as well as to what point the algorithm converges and how high the loss is in that point. Another issue is that points with approximately the same training loss may have generalization losses that are far from another. [11, pp. 296-298]

Because the mechanics of optimization in neural networks is not yet properly understood, the initialization methods are based on heuristics and experimenting. Usually the methods have some desirable statistical properties such as the mean being close to zero and the variance being inversely proportional to the size of layer's input and output. A major issue as mentioned above is that the effect of initial point on generalization is unknown which makes the selection quite hazardous. [11, pp. 297-298]

One property that is certain though is that the initial parameters cannot be the same between different units of the layer. Having the same parameters on units connecting to same inputs with same activation functions means that the units perform exactly the same function in the beginning. With deterministic training algorithm this means that the units will also learn exactly the same function. Even if stochastic regularization methods such as dropout are used, the units should be initialized to perform different functions. This is most computationally efficient to do by using randomized initialization and tuning the statistical properties of the distribution used in sampling of the parameters. [11, p. 298]

Usually the weights are drawn from either uniform or Gaussian distribution. The scale of the weights have a couple of considerations. First, large weights break the

symmetry of the units better, and preserve stronger gradient signal, but too large weights may lead to explosion during either forward or backward propagation. They may also lead to saturation of the activation functions. The choice of the scale is thus a trade-off between the signal strength and variance of the initial functions performed, and the threat of exploding and saturating values. [11, p. 298]

Let the size of input and output of a layer equal to m and n respectively. One commonly used heuristic in the parameter initialization is [10]:

$$W_{ij} \sim U\left[-\frac{1}{\sqrt{n}}, \frac{1}{\sqrt{n}}\right] \tag{3.31}$$

Glorot et. al [10] suggest that the variance of the gradient using the above method depends on the layer and is decreasing. They call for a normalization procedure to the initialization to keep the activation and gradient variances comparable between the layers. They suggest an improved, normalized initialization as:

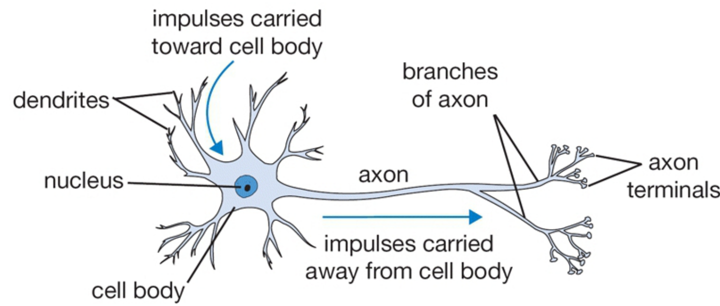
$$W \sim U\left[-\frac{\sqrt{6}}{\sqrt{m+n}}, \frac{\sqrt{6}}{\sqrt{m+n}}\right] \tag{3.32}$$

This is known as *Glorot*-initialization. The equation is derived for network without nonlinearities, i.e. using only matrix multiplications, but it is shown that the linear assumption works rather well with the nonlinear models too [10]. There are many other methods such as using random orthogonal matrices as initializations, but in the scope of this thesis Glorot-initialization is used.

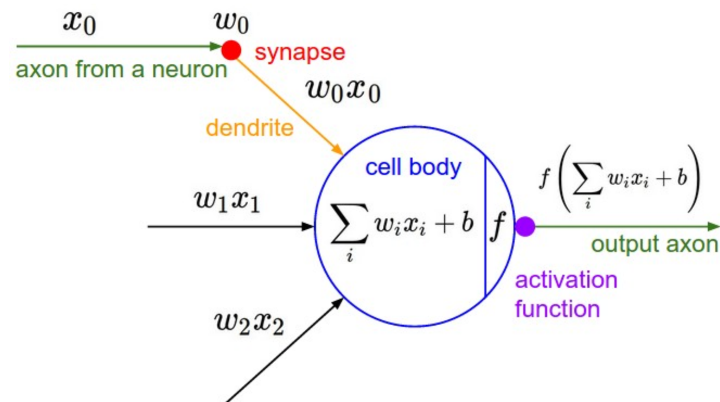
3.1.9 Data preprocessing

With image data, two preprocessing techniques are normally used: mean subtraction and normalization. Mean subraction is done by subtracting the mean of each separate feature, i.e. each pixel from each image in the dataset. With images it is also common to simply subtract the mean across all data, i.e. the "mean pixel" from each feature. Another variant is to do the mean subtraction for each color channel separately. [20]

Normalization of the data means transforming the features into the same scale. The normalization is performed pixelwise. A couple of common normalization methods exist. First is to scale each feature to unit standard deviation. This is done by dividing each feature by its standard deviation. Note that the mean is subtracted first. Another popular method is *minmax scaling* which means that the features are scaled so that the minimum and maximum values of each feature are -1 and +1 correspondingly. With images the normalization is not strictly required because the pixel values are already roughly in the same scale as they range between 0 and 255 already. [20]



(a) A sketch of the structures of a biological neuron. Dendrites carry the signal from other neurons to the cell body. The different dendrite inputs accumulate, and once the electric charge increases to large enough potential, the cell body will release the charge as a spike into its output channel called axon. Then the axon branches and connects to other dendrites via a connection called synapse which has a property called synaptic strength that expresses how much electrical potential is propagated from the axon to the dendrite.



(b) Mathematical model of the biological neuron. Synaptic strength is modeled as a scalar weight w. The axon output is modeled as a constant value x. Cell body performs a dot product on the axons connected to the input dendrites and the weights, adds a bias and then computes the activation function f on the sum.

Figure 3.2: Sketch of a biological neuron and the corresponding mathematical model used in computational neural networks. Image credit: Li et. al. [20]

There are also other preprocessing techniques such as PCA and whitening of the data but these are not used with image data and convolutional networks. These methods involve decorrelation and normalization of the decorrelated features respectively. In PCA only a certain amount of decorrelated features with the greatest variance are kept, which leads to dimensionality reduction. [20]

3.1.10 Biological interpretation

The study of computational neural networks was originally inspired by modeling the actual neurons in the brain. Nowadays the focus has drifted from biological analogies to more sophisticated mathematical methods of e.g. optimization, as well

as engineering. Figure 3.2 shows the structure of the neuron and the mathematical model that is used with neural networks. Dendrites are inputs that connect to other neurons' outputs called axons. The connections called synapses themselves filter the input: they have a *synaptic strength*. This is modeled as the weights. Cell body accumulates the inputs: this is modeled as a dot product. Then, the cell body releases the accumulated charge to its axon in a spiky manner. This is modeled with an activation function on the weighted sum, which can be interpreted as the firing rate of the cell. [20]

This kind of analogy is very loose. In practice the neurons are way more complex. The dendrites by themselves introduce complex nonlinearities into the cell. The synapses that connect axons to dendrites involve complex nonlinear, dynamical neurochemistry, and still they are represented as scalar weights. Also, the patterns in timing of the output spikes is crucial in many networks of the brain so the firing rate as the output is also an oversimplification [20]. Nevertheless, the modern deep learning models are inspired historically by their biological counterparts, and thus the analogy is included here.

3.2 Convolutional Neural Networks

Convolutional Neural Networks (CNN) are neural networks that use the *convolution operation* (Conv) instead of ordinary matrix multiplication in at least one layer. They can be used with data that has a grid-like structure, such as time series which can be seen as 1D grid of regularly sampled measures, or images that are 2D grids of pixels. [11, chapter 9]

In this section, the basic structures comprising a CNN are introduced, as well as some of the most famous CNN architectures. The concept of **transfer learning** that is relevant when considering the use of CNNs is also explained. CNNs have a lot of applications, especially in computer vision, e.g. image classification, object detection and localisation, semantic segmentation and image captioning. Perhaps the most used and one of the most basic use cases of CNNs is image classification, where the network gets an image as input and tries to predict to which class or classes the image belongs as the output.

CNNs are an example of neural network algorithms that have a particularly solid foundation on neurosciences. They are inspired by how the visual cortex of the brain functions; particularly the *primary visual cortex*, or V1. The visual cortex has a hierarchical structure where different types of cells have a different function that corresponds to those of the layers of CNNs. [11, pp.358]

V1 is structured as a spatial map having a structure similar to convolutional layers and their activation maps. **Simple cells** perform a function in a receptive field that is spatially localized. The activations in the CNNs perform a similar kind

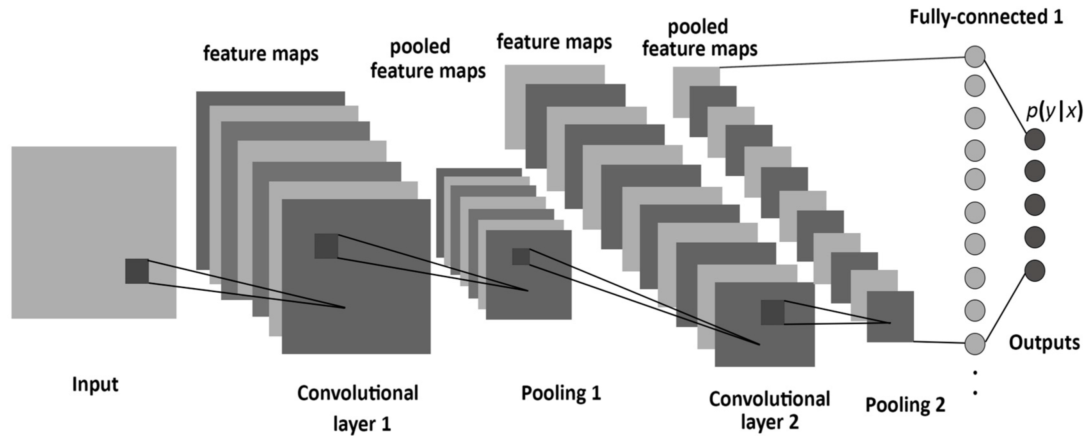


Figure 3.3: The structure of a CNN consists of stacked layers of convolutions, activations and pooling. The last Conv layer's activation maps are then aggregated with FC layers into final predictions in the same way as in any feedforward network. Image credit: Albelwi et. al. [2]

of functionality. **Complex cells**, on the other hand, respond to similar kind of stimuli as the simple cells, but small changes in the position of the stimuli do not affect the response. The pooling layers in CNNs are trying to emulate the functions of the complex cells. [11, pp. 358-360]

3.2.1 Basic building blocks of a CNN

Basic CNNs consist of repeated stack of convolutional layers, activation layers and pooling layers. Structures of these three types are repeated as many times as the desired depth is, and then one or a couple of fully connected layers is added to aggregate the features and make the final predictions. The following list describes the concepts and properties of these layers.

Convolutional layer The convolutional layer consists of k filters, or **kernels**, and the output of the layer is the convolution between the input and each kernel. This produces k *activation maps* as the output, one for each kernel. The convolution in the context of CNNs between an image I and kernel K can be formulated as [11, p. 329]

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i+m, j+n)K(m,n), \tag{3.33}$$

Where i and j are the image coordinates in which the convolution is computed, and m and n go over the dimensions of the kernel. The actual implementations of the Conv operation use many kinds of tricks to make the computation efficient. Usually in the context of CNNs convolution means the application of

many convolutions in parallel to produce n activation maps. The input images also have a depth because they consist of multiple color channels, specifically three for RGB images. [11, p. 342]

Convolution can be thought as sliding the kernel over the input and computing a kind of 3-D dot product on each position. In addition to the kernel size κ, the layer has a parameter called **stride** (s), which tells how many pixels in each direction are skipped on each location. Setting a stride larger than one will downsample the input. Another parameter for a Conv layer is the **padding**. Padding concerns how the edges of the image are handled. Without adding any pixels on the edges, the conv layer will shrink each edge by $\kappa/2 - 1$ pixels. A commonly used padding strategy is called **same**, in which the layer assings such padding that the size of the input is preserved to the output with $s = 1$. [11, p. 343]

To use the Conv layer in training of the network, we also need to compute the gradients for back-propagation. Say the Conv layer applies a stack of kernels $\mathbf{K}$ to multichannel input $\mathbf{V}$ with stride s and the operation is denoted as $c(\mathbf{K}, \mathbf{V}, s)$, the output of the layer is $\mathbf{Z}$ and the loss function is denoted as $J(\mathbf{K}, \mathbf{V})$. Then, during back-propagation the gradient of the loss with respect to the layer output is $\mathbf{G}$, where [11, p. 351]

$$G_{ijk} = \frac{\partial J(\mathbf{K}, \mathbf{V})}{\partial Z_{ijk}}. \tag{3.34}$$

The gradient on the kernel weights can be then computed as [11, p. 351]

$$g(\mathbf{G}, \mathbf{V}, s)_{ijkl} = \frac{\partial J(\mathbf{K}, \mathbf{V})}{\partial K_{ijkl}} = \sum_{m,n} G_{imn} V_{j,(m-1)\times s+k,(n-1)\times s+l}. \tag{3.35}$$

To propagate the gradients further back in the network, the gradient of the loss with respect to $\mathbf{V}$ is also needed. This is computed as [11, p. 351]

$$\frac{\partial J(\mathbf{K}, \mathbf{V})}{\partial V_{ijk}} = \sum_{\substack{l,m \\ s.t. \\ (l-1)\times s+m=j}} \sum_{\substack{n,p \\ s.t. \\ (n-1)\times s+p=k}} \sum_{q} K_{q,i,m,p} G_{q,l,n}. \tag{3.36}$$

Activation layer The nonlinearities work in the same way as in the fully connected feedforward networks. The same activation functions introduced in section 3.1.2 can be used.

Pooling layer Pooling is performed to downsample the activation maps to make

them smaller and more manageable. Each activation map is handled independently which means the depth of activation maps is preserved. The most common pooling method used is **max pooling**. It is performed by choosing the maximum value within a rectangular area as its output. Max pooling preserves the strongest signals within its kernel's neighborhood. The pooling layer has the kernel size and stride as hyperparameters just like the convolutional layer. [11, pp. 336-339]

Fully Connected layer The FC layers are used to gather the outputs of final convolutional layer to make the final predictions. They function the same way as in feedforward networks introduced in previous sections.

3.2.2 Transfer learning for CNNs

Training a CNN with good performance requires a large dataset and takes time. In practice it is common to take a network pretrained on a large dataset and use it either by initializing the weights to the pretrained ones, or by freezing some or all of the layers in it to be used as fixed feature extractors. The pretrained CNN models can be used in following ways [20]:

Fixed feature extraction Remove the last fully connected layer, and use the previous layer's output as a feature vector for other models. This may be a good approach if the new dataset to be fitted is small and similar to the one that was used in the pretrained model.

If the new dataset is small but it is not similar to the data used in pretraining, the higher-level features may not be usable. Nevertheless, the low-level features may work, so it is worth trying to fit a linear classifier using activations from earlier layers as the features.

Fine-tuning In this approach, the last layer is replaced with a different classifier, and in addition the weights of some or all of the preceding layers are further trained through back-propagation. The earlier the layer, the more general are the features that it learns. As pointed in the previous section, the first layers learn very simple things as edges and shapes.

If the new dataset is large and similar to the one used in pretrained model, all or most of the pretrained layers can be finetuned without excessive fear of overfitting.

Using a pretrained model as the starting point The pretrained models for various datasets are publicly available in various deep learning framework repositories. These models can be used as is, or further trained with similar data to similar functions.

If the new dataset is large but different to the data used in pretrained model, the model may be trained from scratch. However, many times it is beneficial to use the pretrained network as the initialization even if the data is very different, e.g. pretraining was done on ImageNet data and the new data is from microscopy imaging.

Transfer learning with CNNs is more common than training from scratch. It is the norm, not an exception. This opens up new possibilities for experimentation, because many of the open sourced CNN models would take months to train with commodity hardware.

3.2.3 Well-known CNN architectures

There are a few specific architectures that are well-known and often used as a starting point for transfer learning when training a CNN for a new purpose. They are also used as a baseline when trying out new techniques on standardized datasets like the large-scale visual database ImageNet [28] or the handwritten digits database MNIST [18].

LeNet One of the first CNNs successfully applied in a recognition task on images of hand-written digits which started the revolution of CNNs in computer vision tasks. [19]

AlexNet The first CNN that outperformed all other models not based on deep learning in ImageNet Large Scale Visual Recognition Challenge (ILSVRC) competition [28]. ReLU linearities were first used in AlexNet. It had a total of five convolutional layers and three fully connected layers in the end (8-layer depth). It used heavy data augmentation, dropout with probability of 0.5, batches of 128 images, SGD with momentum of 0.9, initial learning rate of 1×10^{-2} divided by 10 when validation accuracy plateaued and L2 normalization with decay of 5×10^{-4}. An ensemble of seven CNNs was used, final class scores being the average of them. [17]

VGGNet A CNN with depth of 19 layers. The structure of VGGNet is quite similar to AlexNet. The two main differences are that VGGNet uses smaller filters (only 3×3 filters while AlexNet has 11×11 and 5×5 on the first layers) and much deeper architecture (19 layers compared to 8). Effective receptive field of three layers of 3×3 filters with stride 1 is the same as a single layer with 7×7 filters. So by using smaller filters the effective receptive fields of deeper layer stacks can be kept the same as with larger filters and less layers. Deeper network means more non-linearities. Also, there are fewer parameters

in the deeper architecture with small filters compared to less depth with larger filters. [29]

GoogLeNet has a depth of 22 layers. It introduced the **Inception** module which applies multiple different sized filters in parallel. The spatial dimensions of the filter outputs are kept the same by applying different strides to different sized filters. Then the outputs of these filters are concatenated depth-wise. There are no fully connected layers. [33]

Figure 3.4a shows the structure of the inception module with dimension reductions to prevend computational explosion. With this trick the GoogLeNet has 12 times less parameters than AlexNet but depth has increased from 8 layers to 22. Since the Inception structure was introduced, it has been improved several times, and is widely used in different popular, much deeper architectures [34]. GoogLeNet won the 2014 ILSVRC by a narrow marginal to VGGNet [28].

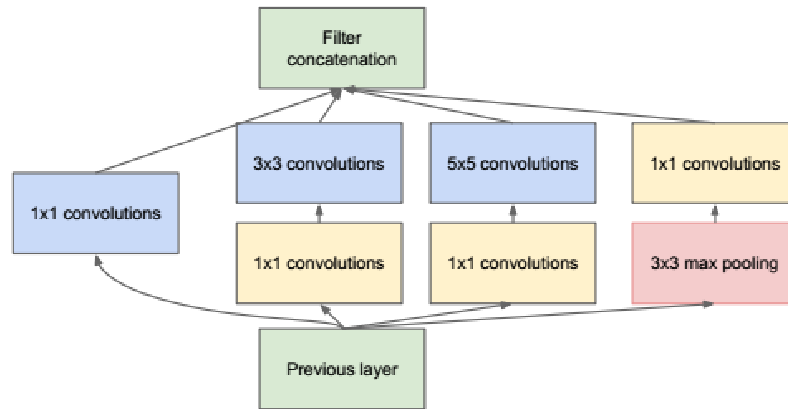
ResNet is a rather different architecture making extremely deep models possible. The layers are learning residual functions with respect to the inputs instead of unreferenced functions. These residual layers are easier to optimise which means there can be significantly more layers without issues of exploding and vanishing gradients. [13].

The residual is defined as $F(x) = H(x) - x$, where $H(x)$ is the "normal" mapping that is used in CNNs, and x is the layer input. I.e. the layer tries to learn the residual instead of the plain mapping. Figure 3.5a shows the basic idea of the residual block. Figure 3.5b depicts the architecture of a ResNet model. ResNet won the 2015 ILSVRC with 152-layer network which is a huge step from the 22-layer GoogleNet that won the previous year. [28]

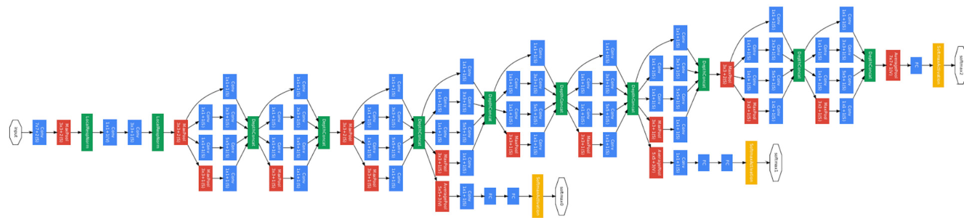
The pattern of using deeper and deeper networks can be seen also in the evolution of these widely used and cited networks. The evolution from 8-layer AlexNet, which was once thought to be a truly deep network, to the 152-layer Resnet is astonishing.

3.3 Fully Convolutional Networks

A **Fully Convolutional Network (FCN)** is a CNN without the fully connected layers in the end. Instead of class label targets, there are target *activation maps* for the final convolutional layer, one for each class. Then, these activation maps can be interpreted as local probabilities of the class being present in the sample. If pooling or convolutions with stride different to one are used, the activation maps of the last layer will be downsampled from the original size of the images. Then, the last layer's activations can be upsampled to obtain pixelwise or more precise predictions using an operation called **transposed convolution**. [21]

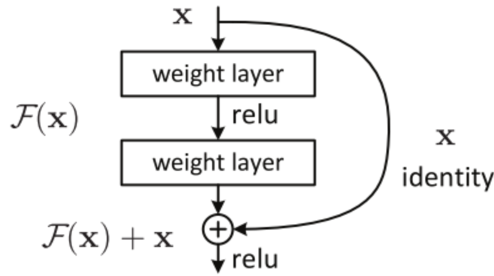


(a) Inception module with dimensionality reductions. With the 1x1 convolutions the input depth is reduced. I.e. there are less filters than in the previous layer's output. With this the computational complexity of the larger filters and the depth of the pooling output can be controlled. The 1x1 convolutions also add depth in the form of added layers to the network, which is often a desired feature. [33]

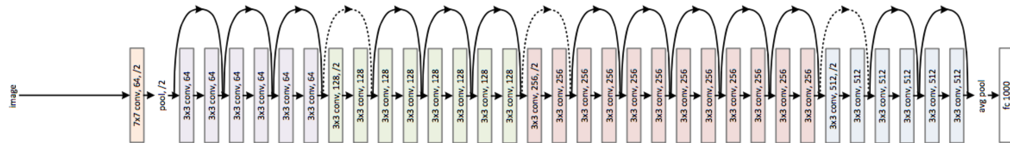


(b) The full architecture of GoogLeNet. Here it is notable that most of the architecture consists of individual inception modules. Also, multiple computationally expensive fully connected layers have been left out from the network architecture. This reduces a lot of parameters. A third interesting detail is that there are two auxiliary classification outputs in the lower layers of the model. This means that the loss and gradients are computed and backpropagated during training in three different places in the network which has shown to improve the performance. In the final prediction, the predictions of the auxiliary inputs can be used by averaging the sum of each output. [33]

Figure 3.4: Inception architecture



(a) ResNet residual block. The residual block has a similar property to L2 regularization in that it tends to drive the weights towards zero. In training, ResNet drives to zero the weights of those blocks that are not needed in classification. [13]



(b) The full architecture of a 32-layer ResNet. Every residual block has two convolutional layers. There is only a single fully connected classifier layer in the end. [13]

Figure 3.5: ResNet building blocks and architecture. Image credit He et. al. [13]

One advantage of a FCN is that the input for it can be of any size. The model will then produce output activation maps of size proportional to the size of the input. The FC layer in the CNN can also be seen as a convolution, where the kernel size is equal to the size of the input image. With this insight, any CNN can be transformed to FCN and be used to arbitrarily sized inputs. [21]

FCNs are a good fit for *semantic segmentation*, where instead of trying to assign classes to the whole image, a class is assigned to each pixel of the image. General idea of segmentation with FCN is depicted in Figure 3.6. Keeping the subsequent convolutional layer outputs the same size as the input is computationally expensive. For that reason the layers are downsampled as described in section 3.2.1, and then upsampled with additional layers to the desired output size. Often it is acceptable that the output size is less than the input size. [21, 11]

Upsampled pixelwise predictions can be produced from the downsampled activation maps by shift and stitch strategy that consists of joining multiple outputs of shifted versions of the original input [21]. Another way to produce the pixelwise predictions is to use interpolation. This can be seen as convolution with a fractional stride $1/f$ and is often referred as *deconvolution* or *transposed convolution*. The concepts of unpooling and transposed convolution are illustrated in Figure 3.7. Transposed convolution can be implemented by reversing the forward and backward passes of a typical convolutional layer. I.e. upsampling is done by back-propagating

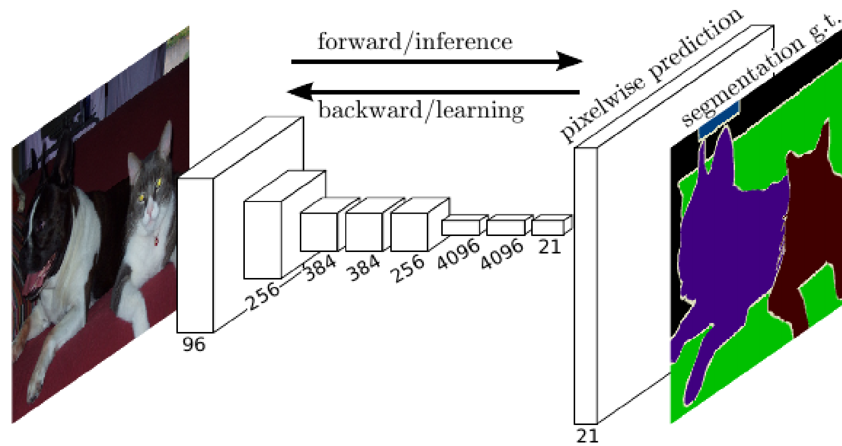


Figure 3.6: Fully convolutional networks can be used in semantic segmentation of an image. The network learns a downsampled representation of class probabilities for each pixel, and the downsampled activation maps are then upsampled using transposed convolution to represent pixelwise predictions. [21]

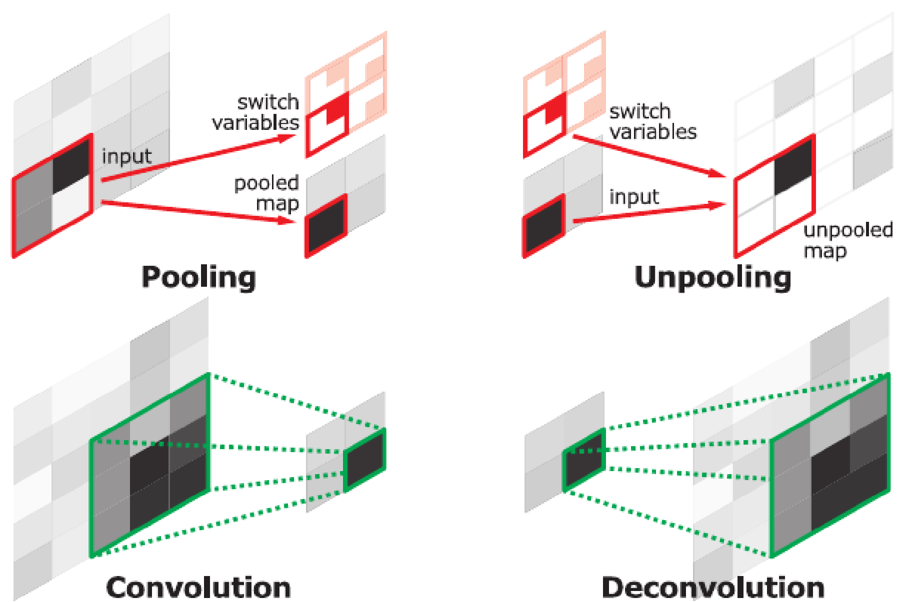


Figure 3.7: Deconvolution and unpooling operations can be understood as backward operations of convolution and pooling. [25]

the pixelwise loss. [21]

The term *transposed* refers to the property of the convolution which states that it can be expressed as a matrix multiplication of a sparse *convolution matrix* $\mathbf{C}$ and the input of the operation as a flattened vector $\mathbf{v}$. For example, consider ordinary convolution with stride 1 and no padding. Then, let the input be of size 4×4, or 16×1 when flattened. The convolution of a 3×3 kernel with the input $\mathbf{V}$ would then produce a 2×2 output, or 4×1 when flattened. If the convolution kernel $\mathbf{K}$'s weights are denoted as w_{ij}, i and j being the row and column indices of the kernel respectively, $\mathbf{C}$ can be expressed as a 4×16 matrix [8]:

$$\begin{bmatrix} w_{0,0} & w_{0,1} & w_{0,2} & 0 & w_{1,0} & w_{1,1} & w_{1,2} & 0 & w_{2,0} & w_{2,1} & w_{2,2} & 0 & 0 & 0 & 0 & 0 \\ 0 & w_{0,0} & w_{0,1} & w_{0,2} & 0 & w_{1,0} & w_{1,1} & w_{1,2} & 0 & w_{2,0} & w_{2,1} & w_{2,2} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & w_{0,0} & w_{0,1} & w_{0,2} & 0 & w_{1,0} & w_{1,1} & w_{1,2} & 0 & w_{2,0} & w_{2,1} & w_{2,2} & 0 \\ 0 & 0 & 0 & 0 & 0 & w_{0,0} & w_{0,1} & w_{0,2} & 0 & w_{1,0} & w_{1,1} & w_{1,2} & 0 & w_{2,0} & w_{2,1} & w_{2,2} \end{bmatrix}$$

The convolution operation could then be obtained as $Conv(\mathbf{K}, \mathbf{V}) = \mathbf{Cv}$.

Then, the backward pass can be performed by multiplying the **transpose** of $\mathbf{C}$ by the output. Thus, in back-propagation the gradient of the loss with respect to the convolution's output is multiplied by $\mathbf{C}^T$. This means that by multiplying a flattened 4×1 input with $\mathbf{C}^T$ leads to 16×1 output which is then rearranged to 4×4. In another words, upsampling in a CNN can be performed by reversing the forward and backward passes of a convolution layer as mentioned before. [8]

Popular deep learning frameworks have implementations of transposed convolution available for the upsampling process. In Keras framework the implementation suitable for image data is named $Conv2DTranspose$ [5].

4. METHODS

This chapter describes the classification tasks considered in the study, the metrics used to evaluate the model performance and the setup and implementation of the testbench used in training and evaluation. We also discuss the various hyperparameters of CNNs and FCNs, as well as how those hyperparameters were chosen in the final models.

Another point of interest is the generation of the target activation maps (or *target masks* in short) used in training of the FCNs. There was no densely labeled targets included in the dataset, so the target masks were generated from the green channel images that contain the protein signal strengths from IF imaging. There are a couple of issues with this approach. The first and most critical problem is that there may be multiple valid subcellular localizations within a single sample, and the computational generation of the masks does not distinguish between the different classes. Secondly, there is at least a small amount of protein signal more or less everywhere in the images, so a threshold is needed to tell from the signal strength whether a class is absent or present. These issues are discussed in section 4.3.2.

4.1 Classification tasks and performance evaluation

In this thesis, the ultimate task was to tell in which subcellular organelles the protein of interest was significantly enriched, i.e. to which organelles a sample was labeled to belong to in the dataset. The data consisted of images with four channels, where the green channel contained the protein enrichment signals, and the three other channels contained the structures of the cell as described in chapter 2.

For experimentation, two different approaches were chosen. First, a CNN with the output being a vector of probabilities for each class present in the image was trained. The second category of algorithms tested was an FCN with the outputs being the activation maps for each class. The maps were generated according to section 4.3.2.

The performance of the models was measured using F_1 scores for each class. Then, a single scalar measure is obtained by averaging the per-class scores. If true positives, false positives and false negatives for a single class are denoted as tp, fp and fn respectively, F_1 score is calculated as the weighted average of the precision and recall:

$$precision = \frac{tp}{tp + fp} \tag{4.1}$$

$$recall = \frac{tp}{tp + fn} \tag{4.2}$$

$$F_1 = \frac{2 \times (precision \times recall)}{precision + recall} \tag{4.3}$$

4.2 Testbench implementation

As described in section 2.3, the dataset consists of 20 000 samples, each having the four channels and the labels for the 13 subcellular locations provided with the samples. Each sample is also labeled with an integer ID from 1 to 20 000. In the testbench setup used in this thesis, sample IDs from 16 001 to 20 000 were used as the test set for the final evaluation. The first 16 000 samples were used as the training data for model training.

For training, 10% of the training data was used as the validation set. At the end of each epoch, the model performance was evaluated with the validation set. The validation loss was then used for monitoring the training process. Also, a technique called *early stopping* was used during the training, and the metric used in it was the validation loss. In early stopping, the training process is stopped if the monitored metric does not improve for a certain amount of epochs. This amount was chosen to be 20.

Another technique used in training that uses the validation loss as the metric is learning rate reduction whenever the loss plateaus. In the training process, learning rate was reduced to one third if the validation loss did not improve in eight consecutive epochs. This number of epochs was derived empirically.

4.3 Data preprocessing

In general, mean subtraction and per-channel normalization were used with the images as described in section 3.1.9. In the case of FCNs, the target masks were generated from the green channel images.

4.3.1 Class imbalance

Class imbalance in CNNs is not as insurmountable of an issue as with many other machine learning algorithms [4]. In this study, the data was used as is, without performing any balancing. One reason for this was to avoid overfitting through oversampling of the rare classes. The rare cases are so infrequent that the oversampling process would need to be very extreme to make a significant change in the

proportions.

Another reason not to balance the classes was to keep things simple. If the dataset is balanced through oversampling or undersampling, the prior distributions of the classes are changed. This would create a source of bias that would need to be taken into account when making the final predictions. The models perform quite well even without balancing, and there were a lot of other things to be tuned, so the balancing was left out of the scope of this thesis.

4.3.2 FCN target generation

In the case of FCNs, the target activation maps, or masks, are needed for training as explained in section 3.3. The masks were generated from the green image as follows:

1. Scale the image to $[0, 1]$
2. Blur the image with Gaussian filter to spread the signal a bit
3. Threshold the filtered image. The threshold value of 0.3 was chosen experimentally.
4. Apply binary dilation to the thresholded image so that the signal will not vanish when resized
5. Resize the result image to the size of the FCN outputs.

This generation process was derived through intuition and probably is not optimal for the problem at hand. Nevertheless, it performs comparable to the CNN approach used. The major issue with this kind of mask generation is that it does not take into account the situations where there are more than one class present in the sample. Whether and to what extent this decreases model's performance would need to be studied as it was left out of the scope here.

4.4 Hyperparameters

A hyperparameter is a variable of the model that is fixed to a constant before the actual training process is performed. The performance of the algorithm depends on the chosen hyperparameters. There are different ways of choosing the hyperparameters, or *optimizing* them.

The first and most informal way is to tune the hyperparameters by hand, i.e. try out some settings, then some others, and see which one performs better. A more systematic way to do this is to perform a **grid search**. In it each hyperparameter tested is chosen from a fixed set. Then the hyperparameter space is organized to a grid-like structure, where each and every possible combination is tested. The issue

Table 4.1: The essential hyperparameters in the models used.

Hyperparameter	Values tested	Explanation
Depth	2..10	The deeper the more nonlinearities.
Number of filters	16..256	How many kernels are learned per layer. Usually because the activations are downsampled when forwarding in the layers, the number of filters can be increased along the upstream.
Filter size	3×3	With smaller filters, there can be more depth with fewer parameters. More depth, more nonlinearities.
Convolution stride	1	How much the filter is slided at each step.
Pooling	2×2 kernel with stride 2	The amount of downsampling.
Padding	'Same'	How the edges of the images are padded.
Activation	ReLU, ELU	Introduces the nonlinearities essential to neural networks.
Batch Normalization	With and without	Normalize the activations for each batch.
Optimizer	SGD with Nesterov momentum, Adam	How the parameter updates are performed based on the gradients.
Initial learning rate	0.001	How large updates to perform at the beginning of training.
LR decay factor	0.2, 0.33	Decrease the learning rate by a factor when validation loss plateaus. A plateau was detected if the validation loss did not improve in eight consecutive epochs.
Dropout	0.5	How big portion of neurons in a layer to discard on each batch. Acts as regularization method.
Weight initializer	Glorot	Random initialization with size proportional to the input and output. Too small -> collapse to zero. Too big -> saturate (gradients go to zero).

with this approach is that it quickly leads to a combinatoric explosion if there are more than a couple of hyperparameters with only a few of different possible values for each.

An improvement to grid search is to perform a **random search** through the hyperparameter space. In it the possible values can be defined as continuous distributions or as discrete sets. Then, those distributions are randomly sampled to obtain a hyperparameter combination that is tested. This is repeated as many times as wanted. The strength of random search is that a model with equal performance to a model tuned with grid search is found in only a fraction of time. If the random search is performed as long as grid search, the hyperparameter space searched is effectively a lot larger because the parameters are sampled from continuous distributions instead of a discrete grid. [3]

Yet another technique for finding good hyperparameters is **Bayesian optimization** in which the validation loss is modeled as a Gaussian process in the hyperparameter space. The issues with Bayesian optimization concern the sequential nature of its optimization process (cannot exploit parallelism to full extent) and the fact that the optimization process itself has hyperparameters which need to be searched too. [30]

There are a plethora of hyperparameters in the case of neural networks, some of which are more sensitive than others. The issue with utilizing hyperparameter optimization techniques described above is that training neural networks is computationally heavy, and any optimization technique requires multiple models trained. A proper hyperparameter search can take weeks or months. With the models tested here, the hyperparameters were chosen by hand through informed guesses. A proper hyperparameter search would be the next step in tuning the models. The most essential hyperparameters of the CNNs and FCNs are listed in Table 4.1 with the values tested in this thesis.

4.5 Network architectures

As the general architecture of the models, two main achitectures were tested. First, a VGG-style network with layers of convolution, activation and pooling stacked sequentially was trained from scratch. This is a simple and easy way to go. The main downside with this architecture is that it has a lot of parameters which means the model is large and heavy to train when the depth is increased.

The second approach was to use **Inception V3** [34] model pretrained with ImageNet data. This architecture utilizes the Inception module described in section 3.2.3 as a trick to decrease the number of parameters while maintaining a high complexity of the overall function. Because the network was pretrained with photograph images, the data used in pretraining was very different compared to the microscopy

Table 4.2: Features of CPUs vs GPUs for some common models. The key takeaway is that with neural networks a high-end CPU is not a sensible investment. On the other hand, a high-end GPU like the Titan Xp here has significantly more cores and fast memory compared to the consumer-level GTX 1070. Thus, investing into a high-end GPU may be worth it depending on the specific needs. Examples taken from the lecture notes of [20].

Model	Number of cores	Clock speed	Memory	Price
CPU (Intel Core i7-7700k)	4 (8 threads with hyperthreading)	4.4 GHz	System RAM	$350
CPU (Intel Core i7-6950X)	10 (20 threads with hyperthreading)	3.5 GHz	System RAM	$1700
GPU (NVIDIA Titan Xp)	3840	1.6 GHz	12 GB GDDR5X	$1200
GPU (NVIDIA GTX 1070)	1920	1.68 GHz	8 GB GDDR5	$400

images. For this reason all of the parameters were tuned in the training process. Also, the pretraining data had only three channels so the red and yellow channels in the training data were merged to one for the Inception architecture.

The architectures for the models trained from scratch are illustrated in Appendix B. The models have equal structures apart from the final layers. The base model before the final layers has 10 convolutional layers. The original VGG-19 net has 16 convolutional layers. On the other hand, the Inception V3 base model has 94 convolutional layers, which is a huge increase. This also means it takes a lot more time to train the Inception V3 compared to the simpler VGG-like architecture.

A major aspect to note is the number of trainable parameters in the models. The FC layer connected to the activation maps of the last convolutional layer in the case of CNN has over 75% of the learned parameters of the whole model, the two FC layers having almost 80% of the trainable parameters. Thus, the CNN with fully connected layers has five times the parameters of the FCN without fully connected layers. In many applications the size of the model is critical.

4.6 GPU Utilization

Graphical processing units (GPUs) are widely used when training deep learning models, as they make the training significantly (10-100x) faster compared to using

a general purpose CPU. In Table 4.2 some commonly used CPU and GPU models are compared with respect to their number of cores, clock speed, memory and price. CPUs have less cores but faster clock speeds on each core. In addition, the instruction set and architecture makes CPU cores a lot more capable compared to GPU cores that can perform only simple tasks on parallel synchronized with the other cores. [20]

Many of the operations performed in the context of neural networks need matrix multiplication, for which GPUs are extremely well suited. Convolution is also a massively parallelisable problem. In general, writing code for GPU is hard. As a result, different high-level APIs are available, e.g. cuBLAS for basic linear algebra, cuFFT for fourier transforms and cuDNN for deep learning. Deep learning libraries make use of GPUs by utilizing these APIs. For example, Tensorflow (which is the backend for Keras used in this project) uses cuDNN for implementing the algorithms on GPU. [12]

One possible bottleneck when using GPUs is data transfer from the hard disc into the CPU and copying the data between the CPU and GPU. It is preferable to profile the training process on how much time is used reading the data and how much doing the computation on the GPU. Some deep learning frameworks utilize the multiple CPU cores to read the data to main memory on background and then the main thread feeds the data to GPU on parallel. So by using a proper deep learning framework all of this can be automated. [20]

Tensorflow, for example, keeps the computational graph inside the GPU memory. So if the weights and weight updates are kept within the graphs, and not as outputs, the large weight matrices will not be copied between CPU and GPU between the batch runs.

5. RESULTS

As described in section 4.5, the two architectures considered were the VGG-style net trained from scratch, and Inception V3 net initialized to weights pretrained with ImageNet data. The Inception V3 architecture is much more complex than the custom architecture used, taking significantly more time to train. As the two performed almost the same in the initial tests, the Inception V3 architecture was left out from fine-tuning and the detailed results here for brevity. A probable cause for the more complex Inception V3 architecture not exceeding the performance of the much simpler architecture in comparison is that it was pretrained and fine-tuned with totally different kind of data. Another issue is that there were only 16 000 training samples, which is rather small compared to the millions of images in the ImageNet dataset.

5.1 Comparison of the model performances

As mentioned in section 4.1 the performance of the models was measured with F_1 score. The overall performances for the best-performing CNN and FCN models are summarized in Tables 5.1 and 5.2 respectively. Both weighted and unweighted means of the classwise F_1 scores are given because the original challenge [1] used the unweighted mean. This emphasizes the rare cases and it can be argued that the weighted score better describes the actual performance. Thus, both are provided

Table 5.1: The results for CNNs trained with different cell line combinations. The training time was made equal by running the models with fewer samples for number of epochs inversely proportional to the number of the samples used in training. Here it can be seen clearly that the models perform the better the more samples are used for training.

Cell lines	Training sample count	Weighted F1	Unweighted F1	Validation loss
All	16 000	0.822212	0.705493	0.092729
[U-2 OS, A-431, U-251 MG]	11 779	0.810109	0.675734	0.098385
[U-2 OS]	5 045	0.760866	0.544508	0.111680
[A-431]	3 479	0.723210	0.495981	0.117385
[U-251 MG]	3 255	0.715761	0.462358	0.134229

Table 5.2: The results for FCNs trained with different cell line combinations similar to CNNs in table 5.1. The same trend (better performance with more data) can also be seen with FCNs.

Cell lines	Training sample count	Weighted F1	Unweighted F1	Validation loss
All	16 000	0.809756	0.706735	0.039991
[U-2 OS, A-431, U-251 MG]	11 779	0.795481	0.669582	0.041885
[U-2 OS]	5 045	0.756717	0.578061	0.043540
[A-431]	3 479	0.753070	0.572095	0.044928
[U-251 MG]	3 255	0.734385	0.538506	0.048719

here.

The validation loss between CNN and FCN models cannot be compared between the two, because CNN uses the 13×1 classwise score vector when computing the loss, whereas FCN loss is computed from the target masks. In the masks there is quite a lot of black empty space which is easy for the FCN to classify, so the validation loss is a lot lower compared to the CNN.

The algorithms were tested with five different combinations of cell lines. For the separate cell lines, the most frequent three in the dataset were chosen. Models were trained with samples from each of the top three cell lines separately, the top three cell lines combined, and with all of the samples in the dataset.

From the Tables 5.1 and 5.2 it can be seen that the performances of the models are quite similar. When using the whole dataset, CNN outperforms FCN in the weighted average F_1 score, but FCN has a bit higher unweighted mean. This shows an interesting characteristic of the FCN: it learns faster with less data. This feature can also be seen when looking at the metrics of the models trained with only one cell line. In those models FCN has clearly better performance in the terms of both weighted and unweighted means.

Tables 5.3 and 5.4 show the classwise metrics for the best-performing CNN and FCN respectively. Here the FCN's ability to learn from little data can also be seen. All of the F_1 scores in the case of FCN are well above 0.5 whereas the CNN's scores for rare classes are as low as 0.364 for actin filaments, which is the second rarest class in the dataset.

The models have learnt the threshold values for classification by trying out values with interval of 0.01 and choosing the thresholds that maximize the F_1 score on the training data. When looking at the classification thresholds, they are much more consistent in FCN, fluctuating between 0.60 and 0.86. On the other hand, with the CNN the thresholds are between 0.19 and 0.65. This can be interpreted so that the FCN is more robust on the classifications it makes.

Table 5.3: Classwise metrics for the best performing CNN.

	f1_score	threshold	train_count	test_count	tp	tn	fp	fn
Actin filaments	0.364	0.29	205	56	16	3928	16	40
Centrosome	0.679	0.37	469	118	71	3862	20	47
Cytosol	0.797	0.47	5507	1344	1111	2323	333	233
Endoplasmic reticulum	0.573	0.38	504	124	65	3838	38	59
Golgi apparatus	0.776	0.44	1027	256	198	3688	56	58
Intermediate filaments	0.441	0.19	154	36	13	3954	10	23
Microtubules	0.780	0.65	302	74	55	3914	12	19
Mitochondria	0.812	0.54	1340	343	268	3608	49	75
Nuclear membrane	0.772	0.51	332	80	61	3903	17	19
Nucleoli	0.864	0.45	1661	412	344	3548	40	68
Nucleus	0.939	0.51	8326	2127	1980	1762	115	143
Plasma membrane	0.696	0.37	1579	359	250	3532	109	109
Vesicles	0.678	0.32	1899	528	352	3313	159	176

Table 5.4: Classwise metrics for the best performing FCN.

	f1_score	threshold	train_count	test_count	tp	tn	fp	fn
Actin filaments	0.526	0.80	205	56	25	3930	14	31
Centrosome	0.517	0.60	469	118	53	3848	34	65
Cytosol	0.803	0.83	5507	1344	1143	2296	360	201
Endoplasmic reticulum	0.590	0.69	504	124	72	3828	48	52
Golgi apparatus	0.657	0.82	1027	256	159	3675	69	97
Intermediate filaments	0.563	0.72	154	36	18	3954	10	18
Microtubules	0.846	0.82	302	74	66	3910	16	8
Mitochondria	0.803	0.73	1340	343	289	3569	88	54
Nuclear membrane	0.803	0.86	332	80	61	3909	11	19
Nucleoli	0.854	0.84	1661	412	354	3525	63	58
Nucleus	0.935	0.85	8326	2127	2019	1700	177	104
Plasma membrane	0.665	0.80	1579	359	230	3538	103	129
Vesicles	0.626	0.52	1899	528	346	3241	231	182

Table 5.5: Confusion matrix for the classification results of the best performing CNN. The columns represent the the ground truth labels, and the rows are showing the predictions.

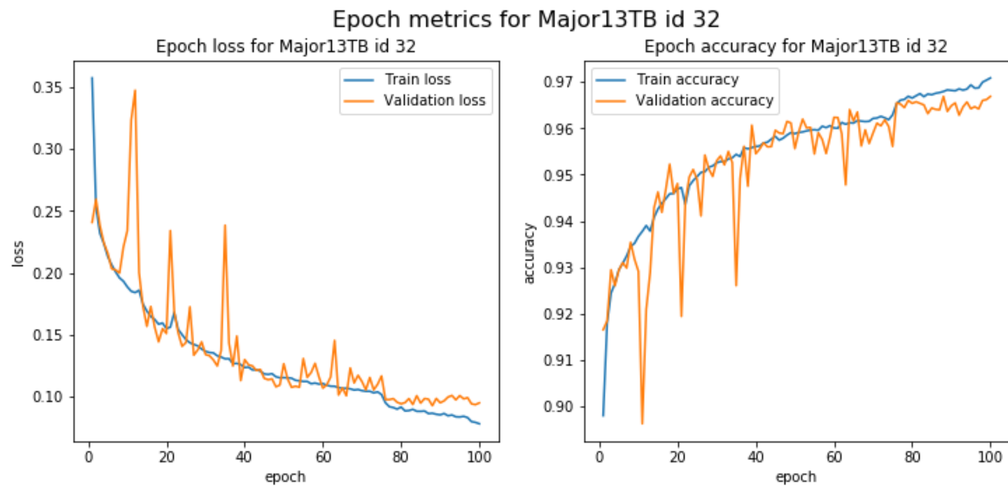
	AF	CE	CY	ER	GA	IF	MT	MC	NM	NI	NU	PM	VE
Actin filaments	16	0	17	0	0	0	0	0	0	3	13	32	3
Centrosome	0	71	21	0	4	0	0	2	1	5	32	4	4
Cytosol	4	12	1111	10	17	3	9	30	5	25	108	40	47
Endoplasmic reticulum	0	0	43	65	1	1	2	6	2	6	4	3	12
Golgi apparatus	0	0	23	2	198	1	0	4	3	4	16	4	23
Intermediate filaments	2	0	11	4	1	13	0	4	0	3	3	1	2
Microtubules	0	0	13	1	0	2	55	0	0	0	5	3	2
Mitochondria	0	0	36	3	6	2	0	268	2	7	28	5	23
Nuclear membrane	0	0	11	1	1	0	0	2	61	1	11	4	1
Nucleoli	2	4	24	1	3	0	1	10	3	344	31	7	9
Nucleus	3	4	63	4	11	2	5	15	16	20	1980	18	23
Plasma membrane	2	0	86	1	9	0	1	1	0	6	48	250	6
Vesicles	3	3	91	4	22	0	0	9	8	13	70	28	352

Tables 5.5 and 5.6 show confusion matrices for the CNN and FCN models respectively. Each column of the confusion matrix represents the true class and each row represents the predicted class. Here it can seen, for example, that the CNN confuses some of the actin filament samples with cytosol and intermediate filament samples, whereas the FCN does not do either of these mistakes.

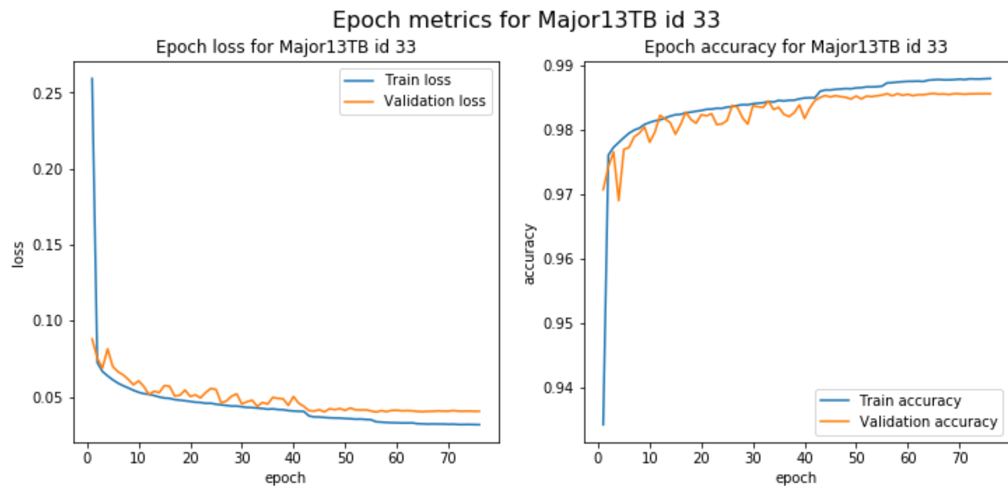
It should be noted that the confusion matrix cannot be unambiguously and accurately defined for multi-class multi-label classification task, so these confusion matrices are only directional. They simply tell that if a class for a sample was not correctly detected, what other classes were present in the sample. In other words, if the predicted labels for a sample do not contain the true label, each of the true label's false positive counts on the predicted labels were incremented, because it cannot be determined to which single one label the sample was confused if there are more than one predicted labels.

5.2 Training process monitoring

Figure 5.1 shows the progress of model loss as a function of training epoch number. The previously discovered trend of FCN learning faster than CNN can be seen from these graphs too. Both training and validation loss drop extremely fast in the case of



(a) The CNN's training process is not as steep as in the case of FCN.



(b) With FCNs, the loss decreases steeper, but the improvement then gets a lot slower compared to the CNN.

Figure 5.1: The progress of loss and accuracy between epochs during the training process for the CNN and FCN with best results. The large fluctuations of the validation loss in the beginning of training result from a higher learning rate. When the training process advances, the learning rate is reduced if validation loss has not improved for a while. This leads to a smoother curve.

Table 5.6: Confusion matrix for the classification results of the best performing FCN. Like in the previous table, the columns represent the ground truth and the rows represent the predictions.

	AF	CE	CY	ER	GA	IF	MT	MC	NM	NI	NU	PM	VE
Actin filaments	25	1	16	0	1	0	0	0	0	1	13	15	5
Centrosome	0	53	25	1	4	0	0	3	0	4	48	2	7
Cytosol	0	6	1143	8	12	2	3	23	3	17	129	18	42
Endoplasmic reticulum	0	0	46	72	1	0	0	8	1	6	6	2	6
Golgi apparatus	0	2	31	3	159	1	1	8	1	5	37	7	35
Intermediate filaments	0	0	13	2	4	18	1	1	0	2	1	1	2
Microtubules	0	0	7	0	0	0	66	0	0	0	2	0	1
Mitochondria	0	0	23	3	3	0	0	289	3	5	20	4	17
Nuclear membrane	0	1	11	0	2	0	0	3	61	2	12	4	4
Nucleoli	2	4	16	1	4	1	1	11	1	354	27	4	10
Nucleus	2	5	43	3	5	2	3	14	4	16	2019	13	22
Plasma membrane	2	2	95	3	6	0	1	3	0	13	63	230	9
Vesicles	2	3	95	2	18	1	2	9	6	15	77	15	346

FCN but then the progress plateaus. This is an interesting phenomena that would need more research. It may be that the FCN would benefit from increasing the model complexity (by adding more layers or more weights per layer) because this kind of behaviour is often seen in models with not enough capacity.

With CNN, the loss curve seems more balanced. The large fluctuations in validation loss are the result of quite high learning rate combined with small dataset size. As mentioned in section 4.2, the learning rate was reduced after eight consecutive epochs if the loss did not improve. The fluctuations clearly decrease in certain points of the progress, which are the learning rate reduction events. First, the learning rate is reduced after around 40 epochs and then another time at around 75 epochs. This mechanism could seemingly be fine-tuned to perform better.

5.3 Examples of the model outputs

Figure 5.2 illustrates an example of the CNN model output. On the top left, the red, blue and green channels are shown. Then, only the protein signal, i.e. the green image is plotted. From the green image we can see that the protein signal is present more or less everywhere in the cell structures. Nevertheless, the signal is a lot stronger in specific locations in the sample, namely the vesicles. The model

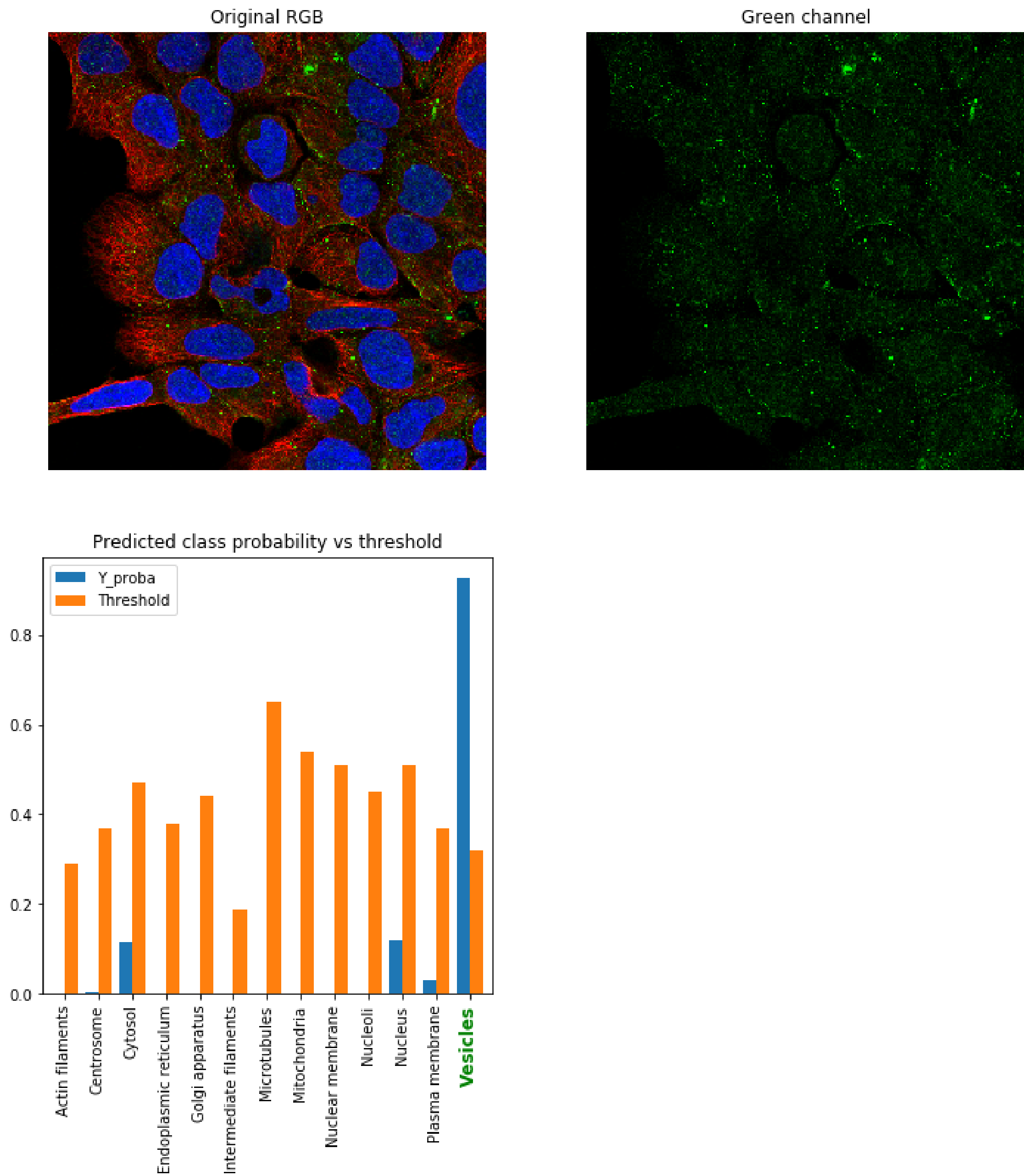


Figure 5.2: CNN output for a sample localizing to vesicles. The model recognizes localization to vesicles with very high confidence. The blue columns represent the predicted probabilities of the model, and the orange columns are the thresholds of detection that the model has learned. The predicted probability for vesicles is above 90%.

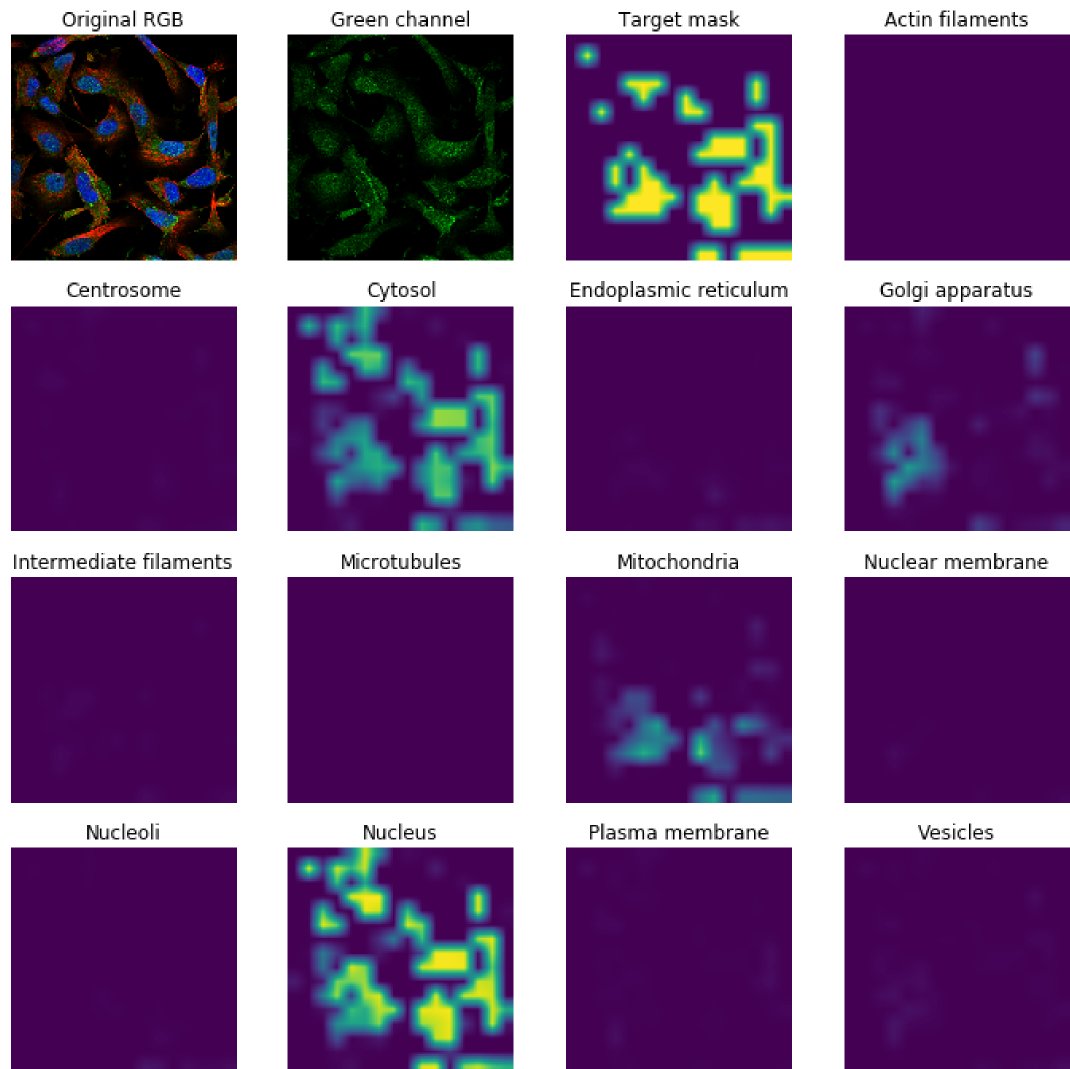


Figure 5.3: An example of the output of the FCN. In this example, the true classes present are Cytosol, Mitochondria and Nucleus, which the network detects quite well. The one class it detects wrong is Golgi apparatus. The target mask and the activation maps are smoothed with bilinear interpolation for visual appeal. Note that the target mask is generated here for reference: this example has not been used in the model training. It is interesting to see how the model detects the patterns of localization presence very similarly to the generated mask. It can even be reasoned that the FCN implicitly learns to generate the mask during training.

outputs probability estimates for each location on whether they are enriched in the image. The estimate of vesicles outstands from the output, and the model finally makes the classification based on the estimate being higher than the threshold.

Figure 5.3 illustrates an example of the FCN model's output. In the input there are three localizations present, namely cytosol, mitochondria and nucleus. The target mask is generated from the green channel. It is a downsampled version of the green image with the areas of strong signal preserved and dilated. In the final model, the output size is 32×32. The mask can be thought of as a map showing roughly the areas where the protein signal is strong.

Then, the FCN model learns to predict those areas according to the training masks. The last 13 images in Figure 5.3 show the output activation maps for each organelle. These maps can be thought of as probability maps showing the areas where the model tells the protein signal is strong within a specific organelle. The model is surprisingly confident on what it sees. In this three-label example it is rather sure that nucleus and cytosol are enriched on those locations, correctly. With mitochondria the confidence is clearly lower. The model also predicts with a considerable probability that there is enrichment in golgi apparatus although according to the labels there is not. Here it is good to bear in mind that the labels are provided by laymen playing the Eve Online game, as explained in section 2.6. Thus, the model might give better estimates on the labels than the actual label providers.

The outputs of FCN compared to CNN reveal another advantage of FCN: it actually tells *where* in the image the signal in a specific organelle is enriched. CNN only tells that *somewhere* in the image there is enrichment within a specific structure, but it does not express the spatial locations.

6. CONCLUSIONS

In this thesis the usage of CNNs and FCNs for subcellular protein localization was studied and tested. The dataset was the same that was used in Cyto2017 conference's imaging challenge [1]. It consisted of only 20 000 samples taken from Protein Atlas database [35]. Rather small architectures similar to VGGNet were used when training the models from scratch. For comparison, Inception V3 was used as the base architecture for both CNN and FCN. These models were initialized to the ImageNet weights provided by Keras. These pre-trained models are trained with ordinary photographs, which is very different from the fluorescent microscopy images used in this study. Because the Inception V3 architecture is heavy to train, and it did not perform significantly better on the initial tests, it was left out from further fine-tuning, and the study focused on the VGG-like case of CNN and FCN.

All in all, the results were surprisingly good considering the size of the dataset. The weighted average of classwise F_1 scores was well above 0.80 for both CNN and FCN. This tells that the task of automatic localization of the proteins into subcellular structures with the means of machine learning is plausible.

Automatic categorization of the enriched proteins into the subcellular structures gives new insight to the functions of the cell. One application of these techniques would be to detect malfunctioning cells in a patient. When a gene is enriched normally, it manifests as a specific localization pattern of a certain protein. Thus, by studying the protein localization patterns we are actually also studying the gene expression.

When comparing the CNN and FCN it was revealed that the FCN learns faster with less data. The FCN model also is only a fraction of CNN in the terms of number of parameters because it lacks the FC layers in the output. According to the monitoring of learning progress, the FCN model could make use of increased model capacity, i.e. more parameters in the form of either deeper architecture or more parameters per layer. This would be an interesting direction of future research.

The number of samples in the dataset was rather small for modern deep learning architectures. Interestingly, advances in this area have been made lately. The same strategy of collecting labels through the Eve Online Project Discovery crowdsourcing challenge has been continued, with an extended number of 29 categories compared to the 13 categories present in the dataset used in this thesis. Compared to the 20

000 samples, a whopping 23.7 million samples have been annotated in the updated dataset [32]. In the article related to the new dataset, a similar classification task was solved. It would be interesting to apply the algorithms developed in this study to the extended dataset, as well as continue the development of the FCN approach.

The reliability of the online players' consensus as the source of ground truth labels has also been discussed in the context of the extended dataset. The evaluation and refinement of the annotations obtained through the Project Discovery, as well as the quality assessment for this process, is an ongoing effort [32].

In addition to the rather small size of the dataset, the human resources for conducting the study were limited, and more systematic fine-tuning of the network structure and the hyperparameters would be needed. Also, the reliability of the results should be analyzed more in depth. In general, the mechanisms of deep learning are not well understood after all.

BIBLIOGRAPHY

[1] *32nd Congress of the International Society for Advancement of Cytometry*. 2017. URL: http://cytoconference.org/2017/Program/Image-Analysis-Challenge.aspx (visited on 02/28/2018).

[2] Saleh Albelwi and Ausif Mahmood. “A framework for designing the architectures of deep convolutional neural networks”. In: *Entropy* 19.6 (2017), p. 242.

[3] James Bergstra and Yoshua Bengio. “Random search for hyper-parameter optimization”. In: *Journal of Machine Learning Research* 13.Feb (2012), pp. 281–305.

[4] Mateusz Buda, Atsuto Maki, and Maciej A Mazurowski. “A systematic study of the class imbalance problem in convolutional neural networks”. In: *arXiv preprint arXiv:1710.05381* (2017).

[5] François Chollet et al. *Keras*. https://keras.io. 2015.

[6] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. “Fast and accurate deep network learning by exponential linear units (elus)”. In: *arXiv preprint arXiv:1511.07289* (2015).

[7] Yann N Dauphin et al. “Identifying and attacking the saddle point problem in high-dimensional non-convex optimization”. In: *Advances in neural information processing systems*. 2014, pp. 2933–2941.

[8] Vincent Dumoulin and Francesco Visin. “A guide to convolution arithmetic for deep learning”. In: *arXiv preprint arXiv:1603.07285* (2016).

[9] The Editors of Encyclopaedia Britannica. *Eucaryote*. 2018. URL: https://www.britannica.com/science/eukaryote (visited on 06/03/2018).

[10] Xavier Glorot and Yoshua Bengio. “Understanding the difficulty of training deep feedforward neural networks”. In: *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*. 2010, pp. 249–256.

[11] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[12] *GPU-Accelerated Tensorflow*. URL: https://www.nvidia.com/en-us/data-center/gpu-accelerated-applications/tensorflow/ (visited on 05/17/2018).

[13] Kaiming He et al. “Deep residual learning for image recognition”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

[14] Kaiming He et al. “Delving deep into rectifiers: Surpassing human-level performance on imagenet classification”. In: *Proceedings of the IEEE international conference on computer vision*. 2015, pp. 1026–1034.

[15] *Human Protein Atlas*. 2017. URL: `http://www.proteinatlas.org` (visited on 02/28/2018).

[16] Sergey Ioffe and Christian Szegedy. “Batch normalization: Accelerating deep network training by reducing internal covariate shift”. In: *International conference on machine learning*. 2015, pp. 448–456.

[17] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. “Imagenet classification with deep convolutional neural networks”. In: *Advances in neural information processing systems*. 2012, pp. 1097–1105.

[18] Yann LeCun, Corinna Cortes, and CJ Burges. “MNIST handwritten digit database”. In: *AT&T Labs [Online]. Available: http://yann. lecun. com/exdb/mnist* 2 (2010).

[19] Yann LeCun et al. “Efficient backprop”. In: *Neural networks: Tricks of the trade*. Springer, 1998, pp. 9–50.

[20] Fei-Fei Li, Justin Johnson, and Serena Yeung. *CS231n Convolutional Neural Networks for Visual Recognition*. 2018. URL: `https://cs231n.github.io/` (visited on 05/11/2018).

[21] Jonathan Long, Evan Shelhamer, and Trevor Darrell. “Fully Convolutional Networks for Semantic Segmentation”. In: *CoRR* abs/1411.4038 (2014). arXiv: `1411.4038`. URL: `http://arxiv.org/abs/1411.4038`.

[22] Andrew L Maas, Awni Y Hannun, and Andrew Y Ng. “Rectifier nonlinearities improve neural network acoustic models”. In: *Proc. icml*. Vol. 30. 1. 2013, p. 3.

[23] Help Me Understand Genetics page: National Library of Medicine (US). Genetics Home Reference. *How do genes direct the production of proteins?* 2018. URL: `https://ghr.nlm.nih.gov/primer/howgeneswork/makingprotein` (visited on 03/01/2018).

[24] Help Me Understand Genetics page: National Library of Medicine (US). Genetics Home Reference. *What are proteins and what do they do?* 2018. URL: `https://ghr.nlm.nih.gov/primer/howgeneswork/protein` (visited on 03/01/2018).

[25] Hyeonwoo Noh, Seunghoon Hong, and Bohyung Han. “Learning deconvolution network for semantic segmentation”. In: *Proceedings of the IEEE International Conference on Computer Vision*. 2015, pp. 1520–1528.

[26] Solip Park et al. "Protein localization as a principal feature of the etiology and comorbidity of genetic diseases". In: *Molecular systems biology* 7.1 (2011), p. 494.

[27] *Project Discovery: Human Protein Atlas*. 2017. URL: https://wiki.eveuniversity.org/Project_Discovery:_Human_Protein_Atlas (visited on 02/28/2018).

[28] Olga Russakovsky et al. "ImageNet Large Scale Visual Recognition Challenge". In: *International Journal of Computer Vision (IJCV)* 115.3 (2015), pp. 211–252. DOI: 10.1007/s11263-015-0816-y.

[29] Karen Simonyan and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition". In: *arXiv preprint arXiv:1409.1556* (2014).

[30] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. "Practical bayesian optimization of machine learning algorithms". In: *Advances in neural information processing systems*. 2012, pp. 2951–2959.

[31] Nitish Srivastava et al. "Dropout: A simple way to prevent neural networks from overfitting". In: *The Journal of Machine Learning Research* 15.1 (2014), pp. 1929–1958.

[32] Devin P Sullivan et al. "Deep learning is combined with massive-scale citizen science to improve large-scale image classification". In: *Nature biotechnology* 36.9 (2018), p. 820.

[33] Christian Szegedy et al. "Going deeper with convolutions". In: Cvpr. 2015.

[34] Christian Szegedy et al. "Rethinking the inception architecture for computer vision". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2016, pp. 2818–2826.

[35] Peter J. Thul et al. "A Subcellular Map of the Human Proteome". In: *Science* 356.6340 (May 2017). DOI: 10.1126/science.aal3321.

[36] Ashia C Wilson et al. "The marginal value of adaptive gradient methods in machine learning". In: *Advances in Neural Information Processing Systems*. 2017, pp. 4151–4161.

A. COUNTS OF LOCALIZATION LABELS BY CELL LINE

Table A.1: Localization counts per cell line. The labels are abbreviated from the originals: AF (Actin Filaments), CE (Centrosome), CY (Cytosol), ER (Endoplasmic Reticulum), GA (Golgi Apparatus), IF (Intermediate Filaments), MT (Microtubules), MC (Mitochondria), NM (Nuclear Membrane), NI (Nucleoli), NU (Nucleus), PM (Plasma Membrane), VE (Vesicles)

cell_line	AF	CE	CY	ER	GA	IF	MT	MC	NM	NI	NU	PM	VE
A-431	66	125	1795	109	302	32	99	365	98	379	2230	450	449
A549	4	13	152	12	28	9	8	41	5	44	255	49	68
AF22	0	3	2	1	3	0	0	1	0	1	14	3	4
ASC TERT1	0	0	4	0	0	0	0	0	0	0	2	3	1
BJ	3	2	34	11	24	0	1	8	3	15	88	29	31
CACO-2	3	0	44	11	27	2	4	30	3	36	205	21	55
HEK 293	5	16	121	15	37	9	6	57	16	60	330	23	86
HUVEC TERT2	0	0	3	6	6	0	0	0	0	0	12	6	7
HaCaT	0	8	36	2	13	8	1	6	5	12	69	36	30
HeLa	6	9	107	13	25	3	5	25	7	40	246	52	58
Hep G2	1	5	96	19	39	0	5	31	3	32	222	23	55
MCF7	5	9	130	23	36	7	5	70	17	138	460	23	67
NB-4	0	0	2	0	2	0	0	1	1	0	0	0	0
PC-3	0	8	156	21	12	0	11	29	17	87	337	50	58
REH	0	5	10	1	1	0	0	2	0	2	6	2	1
RH-30	3	42	85	4	32	3	3	25	9	47	263	35	46
RT4	0	11	52	8	25	1	0	17	7	16	156	19	54
SH-SY5Y	6	15	76	11	29	5	1	16	12	36	172	18	74
SK-MEL-30	0	11	30	4	9	1	6	19	5	25	111	10	26
SiHa	5	14	47	12	21	1	3	21	5	26	145	26	47
U-2 OS	125	156	2215	238	345	78	129	610	120	697	3081	643	837
U-251 MG	29	135	1654	107	267	31	89	309	79	380	2049	417	373

B. NETWORK STRUCTURES

B.1 Architecture for CNN trained from scratch

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 256, 256, 4)       0
_________________________________________________________________
conv_0 (Conv2D)              (None, 256, 256, 32)      1184
_________________________________________________________________
conv_act_0 (Activation)      (None, 256, 256, 32)      0
_________________________________________________________________
conv_0_2 (Conv2D)            (None, 256, 256, 32)      9248
_________________________________________________________________
conv_act_0_2 (Activation)    (None, 256, 256, 32)      0
_________________________________________________________________
batch_normalization_1 (Batch (None, 256, 256, 32)      128
_________________________________________________________________
conv_pool_0 (MaxPooling2D)   (None, 128, 128, 32)      0
_________________________________________________________________
conv_1 (Conv2D)              (None, 128, 128, 64)      18496
_________________________________________________________________
conv_act_1 (Activation)      (None, 128, 128, 64)      0
_________________________________________________________________
conv_1_2 (Conv2D)            (None, 128, 128, 64)      36928
_________________________________________________________________
conv_act_1_2 (Activation)    (None, 128, 128, 64)      0
_________________________________________________________________
batch_normalization_2 (Batch (None, 128, 128, 64)      256
_________________________________________________________________
conv_pool_1 (MaxPooling2D)   (None, 64, 64, 64)        0
_________________________________________________________________
conv_2 (Conv2D)              (None, 64, 64, 128)       73856
_________________________________________________________________
```

```
conv_act_2 (Activation)      (None, 64, 64, 128)       0
_________________________________________________________________
conv_2_2 (Conv2D)            (None, 64, 64, 128)       147584
_________________________________________________________________
conv_act_2_2 (Activation)    (None, 64, 64, 128)       0
_________________________________________________________________
batch_normalization_3 (Batch (None, 64, 64, 128)       512
_________________________________________________________________
conv_pool_2 (MaxPooling2D)   (None, 32, 32, 128)       0
_________________________________________________________________
conv_3 (Conv2D)              (None, 32, 32, 256)       295168
_________________________________________________________________
conv_act_3 (Activation)      (None, 32, 32, 256)       0
_________________________________________________________________
conv_3_2 (Conv2D)            (None, 32, 32, 256)       590080
_________________________________________________________________
conv_act_3_2 (Activation)    (None, 32, 32, 256)       0
_________________________________________________________________
batch_normalization_4 (Batch (None, 32, 32, 256)       1024
_________________________________________________________________
conv_pool_3 (MaxPooling2D)   (None, 16, 16, 256)       0
_________________________________________________________________
conv_4 (Conv2D)              (None, 16, 16, 256)       590080
_________________________________________________________________
conv_act_4 (Activation)      (None, 16, 16, 256)       0
_________________________________________________________________
conv_4_2 (Conv2D)            (None, 16, 16, 256)       590080
_________________________________________________________________
conv_act_4_2 (Activation)    (None, 16, 16, 256)       0
_________________________________________________________________
batch_normalization_5 (Batch (None, 16, 16, 256)       1024
_________________________________________________________________
conv_pool_4 (MaxPooling2D)   (None, 8, 8, 256)         0
_________________________________________________________________
flatten_1 (Flatten)          (None, 16384)             0
_________________________________________________________________
dense_1 (Dense)              (None, 512)               8389120
_________________________________________________________________
activation_1 (Activation)    (None, 512)               0
```

```
_________________________________________________________________
dropout_1 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_2 (Dense)              (None, 512)               262656
_________________________________________________________________
activation_2 (Activation)    (None, 512)               0
_________________________________________________________________
dropout_2 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_3 (Dense)              (None, 13)                6669
_________________________________________________________________
activation_3 (Activation)    (None, 13)                0
=================================================================
Total params: 11,014,093
Trainable params: 11,012,621
Non-trainable params: 1,472
_________________________________________________________________
```

B.2 Architecture for FCN trained from scratch

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 256, 256, 4)       0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 256, 256, 32)      1184
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 256, 256, 32)      9248
_________________________________________________________________
batch_normalization_6 (Batch (None, 256, 256, 32)      128
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 128, 128, 32)      0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 128, 128, 64)      18496
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 128, 128, 64)      36928
_________________________________________________________________
batch_normalization_7 (Batch (None, 128, 128, 64)      256
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 64, 64, 64)        0
```

```
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 64, 64, 128)       73856
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 64, 64, 128)       147584
_________________________________________________________________
batch_normalization_8 (Batch (None, 64, 64, 128)       512
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 32, 32, 128)       0
_________________________________________________________________
conv2d_7 (Conv2D)            (None, 32, 32, 256)       295168
_________________________________________________________________
conv2d_8 (Conv2D)            (None, 32, 32, 256)       590080
_________________________________________________________________
batch_normalization_9 (Batch (None, 32, 32, 256)       1024
_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 16, 16, 256)       0
_________________________________________________________________
conv2d_9 (Conv2D)            (None, 16, 16, 256)       590080
_________________________________________________________________
conv2d_10 (Conv2D)           (None, 16, 16, 256)       590080
_________________________________________________________________
batch_normalization_10 (Batc (None, 16, 16, 256)       1024
_________________________________________________________________
conv2d_11 (Conv2D)           (None, 16, 16, 13)        3341
=================================================================
Total params: 2,358,989
Trainable params: 2,357,517
Non-trainable params: 1,472
_________________________________________________________________
```