FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

ASSIGNMENT OF BACHELOR'S THESIS

Title:	Facial feature detection
Student:	Jan Hroch
Supervisor:	Juan Pablo Maldonado Lopez, Ph.D.
Study Programme:	Informatics
Study Branch:	Knowledge Engineering
Department:	Department of Applied Mathematics
Validity:	Until the end of winter semester 2019/20

Instructions

1) Study the problem of facial feature detection in pictures/video.
2) Discuss advantages and drawbacks of different algorithms.
3) Explore various proposed approaches (e.g. neural networks).
4) Compare the performance (classification/training time/scoring time) of the proposed approaches.
5) Consider possible optimizations and improvements.
6) Implement the improved method and analyze achieved results.

References

Will be provided by the supervisor.

Ing. Karel Klouda, Ph.D.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague March 2, 2018

Bachelor Project

Czech Technical University in Prague

F8

Faculty of Information Technology
Department of Applied Mathematics

Facial Feature Detection

Jan Hroch

Supervisor: MSc. Juan Pablo Maldonado Lopez, Ph.D.
May 2018

Acknowledgements

I would like to express my gratitude to Pablo, my supervisor, for his time and valuable advice.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis. I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60(1) of the Act.

In Prague, 14. May 2018

. . . . . .

Abstract

This thesis describes several approaches to the problem of facial feature detection in pictures. We use various types of artificial neural networks including convolutional neural networks, the state of the art in image recognition. The thesis examines upsides and drawbacks of those approaches. It also shows numerous techniques that can improve results while solving image recognition tasks. Furthermore, it analyses the results of such techniques.

Keywords: Artificial intelligence, Neural networks, Machine learning, Computer vision

Supervisor: MSc. Juan Pablo Maldonado Lopez, Ph.D.

Abstrakt

Tato práce popisuje několik možných postupů při řešení problému detekce částí obličeje v obrázcích. K řešení tohoto problému je v práci využito několik různých druhů umělých neuronových sítí včetně konvolučních neuronových sítí, které patří mezi nejmodernější technologie pro rozpoznávání obrazu. Dále práce popisuje výhody a nevýhody použitých postupů. V práci je také popsáno několik metod, které mohou vylepšit výsledky při řešení problémů týkajících se zpracování obrazu. Dále práce analyzuje výsledky dosažené použitím těchto metod.

Klíčová slova: Umělá inteligence, Neuronové sítě, Strojové učení, Počítačové vidění

Překlad názvu: Rozpoznávání částí obličeje

Contents

Figures

Tables

Chapter 1

Introduction

Detecting facial features can be used in multiple ways. One of them is analysing facial expressions, which can be used to detect person's mood. Probably the most useful application of detecting facial landmark is in face recognition. The relative positions of facial keypoints from given image can be compared with faces stored in database to identify a person. Human face can be compared to other biometrics such as fingerprints. Unlike fingerprints, face can be scanned from greater distance. Precisely predicting facial landmarks is key to successful face recognition system, which can be used in everyday life. Finding facial features correctly in images can be a challenging task, due to large variation of face expressions, 3D pose, lighting and viewing angle.

Police in the Chinese city of Shenzhen is using face recognition technology to punish traffic rules violators. Shenzhen based AI firm Intellifusion provides technology to the local police to recognize jaywalkers and to fine them. The system uses cameras to capture photos of pedestrians crossing the road on red light. Facial recognition technology then identifies the offender from a database and displays a photo of the jaywalking offence along with the family name of the individual on large LED screens above the pavement. Facial recognition can also be used to find criminals. Police in the Henan province in central China is using special glasses equipped with face recognition software to help search for wanted criminals. Furthermore, China plans to build large facial recognition database to identify its citizens. The project's goal is to identify any of the 1.3 billion citizens within 3 seconds with around 90% success rate. [1, 2, 3]

Facial recognition can be divided into several problems. First step is to find faces in an image. The next step is detecting positions of facial landmarks on found faces, which is the topic of this thesis. The last step is comparing the detected positions with faces stored in a database and analysing the results.

The goal of this thesis is to propose multiple approaches to facial feature detection using state of the art technologies. We will discuss their upsides and downsides. Furthermore, we will compare their performance, that is how precisely can we predict facial keypoints on given images and how much time does it take. Based on the achieved results we will consider possible changes in order to further improve the outcome. Additionally, we will implement the proposed changes and analyse achieved results.

Chapter 2

State of the art

2.1 Machine learning

Machine learning is a field of computer science that specializes in techniques that give computers ability to learn. The goal is to teach computer system to solve tasks without explicitly programming it. Machine learning tasks can be roughly divided into multiple categories:

- Supervised learning
  - Classification
  - Regression
- Unsupervised learning
  - Clustering
- Reinforcement learning

Supervised learning is used to improve results of a function that maps an input to an output based on example input and output pairs. While performing classification, the goal is to categorize input into classes. To illustrate, imagine a training set of pictures containing various kinds of fruits, one type per picture. For each image, we have a label that tells us which type of fruit is in the picture. We can use this set of pairs to train a computer system which we can later utilize to classify images that we do not have labels for. Another application of supervised learning is for solving regression tasks. In regression tasks, the output is a continuous value. For example a value of house can be predicted based on its size. For solving classification tasks, approaches like decision trees or neural networks can be used. Regression tasks can be solved by algorithms such as linear regression, regression trees, neural networks and others.

2.2 Artificial neural networks

Artificial neural networks (or simply neural networks) are inspired by biological brains. They can be used to solve a variety of tasks, such as computer vision,

medical diagnosis or speech recognition. Neural networks can improve their performance on certain task by taking examples into consideration. Such process can be simply called learning.

A neural network (NN) consists of artificial neurons. An artificial neuron (simply neuron) is inspired by biological neuron. Artificial neurons are connected together. Each connection is used to transmit a signal to another neuron. The receiving neuron can process it and signal other neurons connected to it. Each neuron has set of weights. The weights are edges, connected to each neuron in the previous layer. Neuron has one output value, which is weighted sum of input values plus a bias passed to an activation function. The **Output value of neuron** is defined by the following formula:

$$y = f\left(\sum_{i=1}^{n}(x_i w_i) + b\right),$$

where:

- x_i are input values,
- w_i are weights,
- n is number of input values,
- b is bias,
- $f(x)$ is activation function,
- y is output of the neuron.

The simplest type of neural network is called **perceptron**. It consists of one neuron and it is able to decide whether an input defined by a vector of numbers belongs to some specific class or not. Single-layer perceptrons are capable of solving linearly separable problems [4]. The neuron in single-layer perceptron uses an activation function which maps the weighted sum of inputs plus bias to values 0 or 1. The perceptron algorithm was invented in 1958 by Frank Rosenblatt [5].

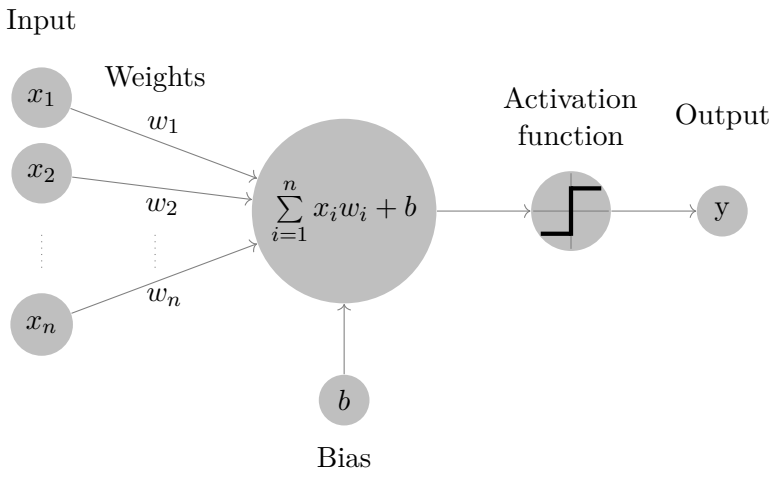

Figure 2.1: Perceptron.

Typically, neurons are organized into groups called layers. A neural network consists of at least two layers, that is input and output layer. It may also contain hidden layers. **Multi-layer perceptron** (MLP) consists of at least three layers. The output and hidden layers use non-linear activation function. In contrary to single-layer perceptron, multi-layer perceptron is capable of learning data that is not linearly separable. MLP uses backpropagation as a learning algorithm which is further described in subsection 2.2.2.

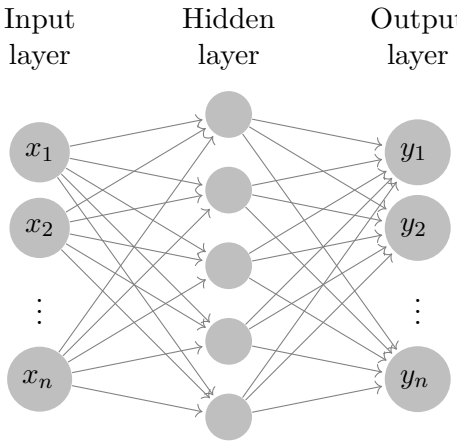

Figure 2.2: Neural network with multiple layers.

2.2.1 Activation function

Activation function defines the output of a neuron given an input. There is a large amount of activation functions. This subsection describes only a few of the commonly used activation functions [6] and they are visualised in Figure 2.3.

- **Binary step** is a simple activation function which returns 0 or 1. It represents whether the neuron is firing or not.
- **Sigmoid function** is defined by $f(x) = \frac{1}{1+e^{-x}}$. Its curve is similar to shape of "S". It maps the resulting values in between 0 and 1. Therefore, it is usually used in the output layer for tasks where the output is probability, for example binary classification.
- **Tanh** or hyperbolic tangent is similar to sigmoid function. It also has shape similar to "S" and its range is from -1 to 1. The advantage of Tanh over Sigmoid function is that the zero inputs will be mapped around zero and negative inputs will be strongly negative.
- **ReLU (Rectified Linear Unit)** is currently the most used activation function, since it works well for convolutional neural networks and for deep neural networks in general. It computes the function $f(x) = \max(0, x)$. The issue with ReLU is that all negative values become zero, which may decrease the ability of model to train properly.

- **Leaky ReLU** attempts to fix that problem. Instead of returning 0 for $x < 0$, it will have a small negative value, for example $0.01x$.

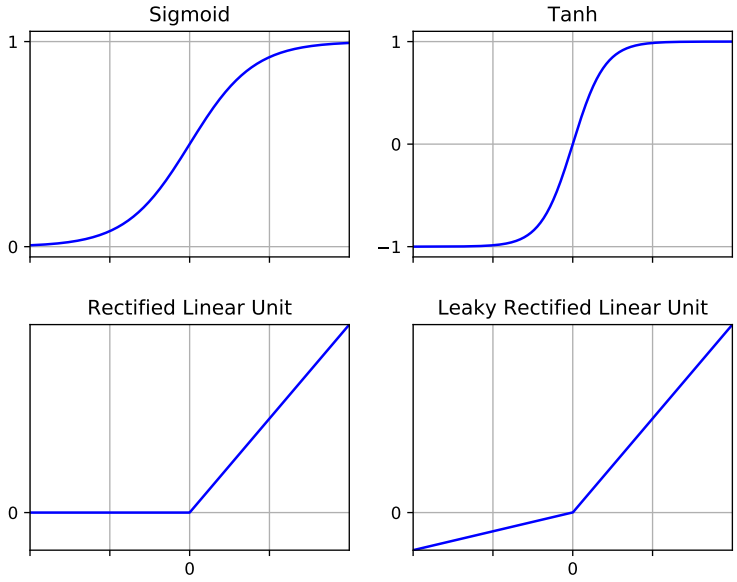

Figure 2.3: Graphs of activation functions.

Activation functions share a common attribute. Their derivative can be easily computed [7], which is used during learning to find the slope of the curve. The slope is needed to know in which direction and how much to change the curve to find the optimal values for weights and biases. By using activation functions with easily computed derivatives, we can save some computations.

Name	Equation	Derivative
Sigmoid	$f(x) = \frac{1}{1+e^{-x}}$	$f'(x) = f(x)(1 - f(x))$
Tanh	$f(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$	$f'(x) = 1 - f(x)^2$
ReLU	$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Leaky ReLU	$f(x) = \begin{cases} 0.01x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0.01 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$

Table 2.1: Activation functions and their derivatives.

Sigmoid and Tanh functions share the same disadvantage. The slope of their curve gets low for x values far from 0, which can slow down the learning. Neither ReLU nor Leaky ReLU have this problem for $x > 0$. When choosing an activation function for our model, ReLU is the default choice. However, due to the **dying ReLU problem**, Leaky ReLU might be worth trying as it might improve results. The dead ReLU always outputs same value, which is 0. The ReLU might end up in this state after its weights were updated

in such way that it will never activate again. Once it ends up in this state, it is unlikely to recover because the function's gradient for 0 is also 0, so the weights of the neuron will not be changed. The inputs of a dead ReLU are still being updated via other neurons, so the dead ReLU can be revived through updates to the previous layer.

2.2.2 Backpropagation

Backpropagation [8] is an algorithm used to calculate adjustments to the weights during supervised learning. The goal of supervised learning is to map set of inputs to their correct output. Backpropagation is used to train neural network with multiple layers by tuning the weights according to loss function which represents the difference between the network's output and the expected output. The algorithm can be split into multiple steps:

- First, an example is shown to the network.
- The prediction error is computed based on the output and the targeted value.
- To lower this error, changes to weights, biases and activations in previous layer are required. Because the activations from previous layer are used by all neurons in the following layer, each neuron has a different idea how to change the activation value in order to reduce its error. Those values are added up for each neuron in the previous layer and they represent the desired change to the neuron's output.
- Those differences are then propagated backwards to previous layer by repeating the very same process until the input layer is reached.
- Finally, after calculating the desired changes for each neuron, the adjustments for each weight can be computed from its output difference and input activation.

This whole process is repeated for each round of learning and its goal is to find optimal weights and biases.

2.2.3 Loss function

Loss function or cost function is used to represent the error of the neural network. It can be used to compare the performance of multiple models. The loss function is chosen based on the type of task being solved. The following list describes two frequently used loss functions.

- **Cross entropy**, which is used to quantify the difference between two probability distributions is often used as a loss function while solving multiclass classification tasks. Cross entropy is defined by the following formula:

$$H(p, q) = -\sum_{i} p_i \log q_i,$$

where p is the ground truth and q is the predicted value. It is used to measure how close is the true probability distribution to the predicted probability distribution. Cross entropy is preferred because it takes into account the closeness of the prediction.

- **Mean squared error** (MSE) is another function that can be used to measure the error of a neural network and it is defined by:

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

where n is the number of outputs, y is the actual value and $\hat{y}$ is the predicted value. MSE punishes large mistakes much more than small mistakes and its output is always positive. With that said, it is commonly used as a cost function for regression tasks where the goal is to predict n real values.

2.2.4 Gradient descent

The goal of optimization is to find optimal weights and biases that lead to minimal loss function. The most common optimization algorithm is **gradient descent**. It uses the first derivative of a function at given point to find the direction in which the weight and biases need to be tuned in order to lower the loss. There are multiple variants of gradient descent, which differ in the amount of data used to calculate the gradient. [9]

- **Batch gradient descent** uses the whole dataset to calculate the gradient of the loss function. The descent can be very slow, because only one update is performed for the whole dataset. Also the whole dataset needs to fit in the memory which can be a problem especially for very large datasets.

- **Stochastic gradient descent** (SGD) calculates the gradient and performs an update for each example. Therefore, it is usually much faster. On the other hand, performing frequent updates with high variance causes the loss function to oscillate.

- **Mini-batch gradient descent** is a combination of previous two approaches. It performs an update for every mini-batch of size n. Calculating the gradient over n examples can lead to more stable convergence. The size of batch is usually chosen in range between 50 and 256, but it may vary depending on the dataset and application. SGD is generally the best choice, because it combines the advantages of the other two approaches.

Gradient descent can be further optimized by addressing its issues:

- Choosing correct learning rate can be difficult. Learning rate is a parameter used when adjusting the weights during training. Low learning

rate can lead to slow convergence while too high learning rate can prevent convergence by causing the loss function to fluctuate around the minimum.

- Learning rate schedules [10] try to adjust the learning by pre-defined schedule. However, those schedules have to be defined beforehand which makes them unable to adapt to the dataset's structure.
- The same learning rate applies to all parameter updates. Depending on the structure of our data, we might want to update parameters that change often more carefully than parameters that change rarely.

The following list generally describes some of the algorithms that deal with mentioned issues.

- **Momentum** [11] is a method that helps to accelerate gradient descent. It takes into account the gradient of previous step and increases the updates for the dimensions that lead into the same direction while reducing the updates for dimensions whose gradients change direction, which leads to faster convergence and reduced oscillation.
- **Adagrad** [12] is an algorithm that adapts learning rate to parameters. It performs larger updates for rarely changed parameters and smaller updates for parameters that are changed frequently. That is achieved by taking into account all previous gradients of each parameter. However that causes the learning rate to decrease with every training step and it eventually becomes too small for the network to learn anything.
- **Adadelta** [13] aims to fix the weakness of Adagrad by taking into account only certain amount of past gradients.
- **Adaptive moment estimation** (Adam) [14] is an another algorithm that is used to compute learning rates for each parameter. Adam is a combitation of momentum and Adadelta. In addition to storing the past n squared gradients like Adadelta, it also stores past gradients similar to momentum. The authors show that Adam works well in practice and that it is comparable to other adaptive learning algorithms.

While choosing optimization algorithm for our model, Adam might be a good choice because it generally outperforms other described algorithms [14].

2.2.5 Dropout

Dropout [15] is a regularization technique that addresses overfitting and provides an efficient way of approximately combining many different neural network architectures. The term "dropout" refers to dropping out a unit. Basically, we temporarily remove a neuron from a neural network along with all its connections – weights. The neurons to keep are chosen randomly. The simplest way to select neurons to keep is by using a fixed probability p. Probability $p = 0.5$ is commonly used, because it works for wide range of

neural networks. There is not a general rule defining the optimal probability for keeping neurons, because it depends on multiple factors like the neural network architecture, structure of the dataset, etc. A validation set can be used to figure out what probability gives the best results for certain use case. The neurons to keep are chosen randomly in every training step, which means that the structure of the neural network is slightly different every time. In a nutshell, we are combining multiple models and averaging the results.
Using dropout influences a neural network in multiple ways:

- It is forced to learn more complex features of the data.
- It increases the number of epochs required for the model to converge. One epoch represents feeding the dataset through the model once.
- Less weights updates are done during each training stage, which means lesser training time for each epoch.
- With n units that can be dropped, we have 2^n possible models during training.

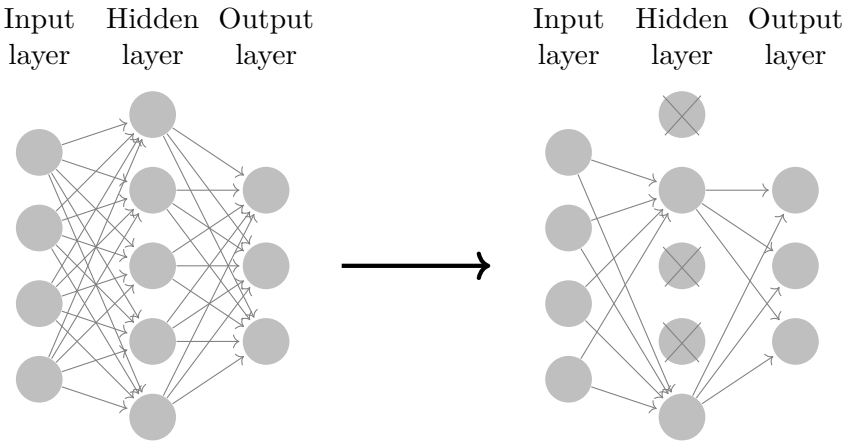

Figure 2.4: Model of a neural network on the left side and the same model after applying dropout to the hidden layer on the right side.

2.3 Deep neural networks

A deep neural network (DNN) is an artificial neural network with multiple hidden layers between the input and output layer. There is not a strict rule defining which neural network is deep and which is not. Using multiple layers allows DNNs to learn more complex relationships between the data. However, deep neural networks have multiple disadvantages:

- They require a lot of data for successful training. This is to some extent addressed by transfer learning. Basically, we use a neural network model pre-trained on a related task and reuse its feature extracting abilities. However, the issue still remains.

- Selecting the correct hyperparameters (i.e. learning rate, used activation function) / training method / structure is not always obvious. However, we can suggest changes based on achieved results.
- They require much more computational power, especially during training. They also require a lot of memory, which is a problem especially on mobile devices.
- Training takes longer compared to other algorithms.
- It is hard to understand what is going on under the hood (for example why did the model choose certain decision over the other).

2.4 Convolutional neural networks

Convolutional neural networks (CNN) [6] are a class of deep neural networks that have proven successful for analysing visual imagery [16, 17, 18], for example image recognition, object detection and classification.

The architecture of convolutional neural networks is designed to take advantage of the dimensional structure of input data. When working with images, that can be achieved by preserving the relationship between pixels in a small group of input image. Each neuron in convolutional layer uses a small group of pixels as an input. Which means that all inputs that are connected to a given neuron are close to each other in the input image. Connecting only a local region of input image to a neuron leads to fewer parameters compared to fully connected layers.

Layers of convolutional neural network have neurons organized into 3 dimensions: width, height and depth. Neurons in the convolutional layer share weights with other neurons in the same depth. This reduces the number of learnable parameters.

Convolutional neural networks typically consist of an input layer, multiple convolutional layers, pooling layers, fully connected layers and output layer.

2.4.1 Convolutional layer

The convolutional layer is the main component of CNN. It is a stack of filters (sometimes referred as kernels) used to extract features from an input image. Each filter is used to extract a certain feature. The amount of weights is based on the kernel size. The weights in each filter are shared among other neurons in the same filter. The position of a certain feature is not that important. What matters is whether the feature is present in the picture or not. For example, imagine kernel of size 3×3. Such window is moving across the image's x and y axes by a stride which we define and its output is based on whether the certain feature is present or not. By using multiple filters we are able to detect multiple features, which can be further analysed by following layers.

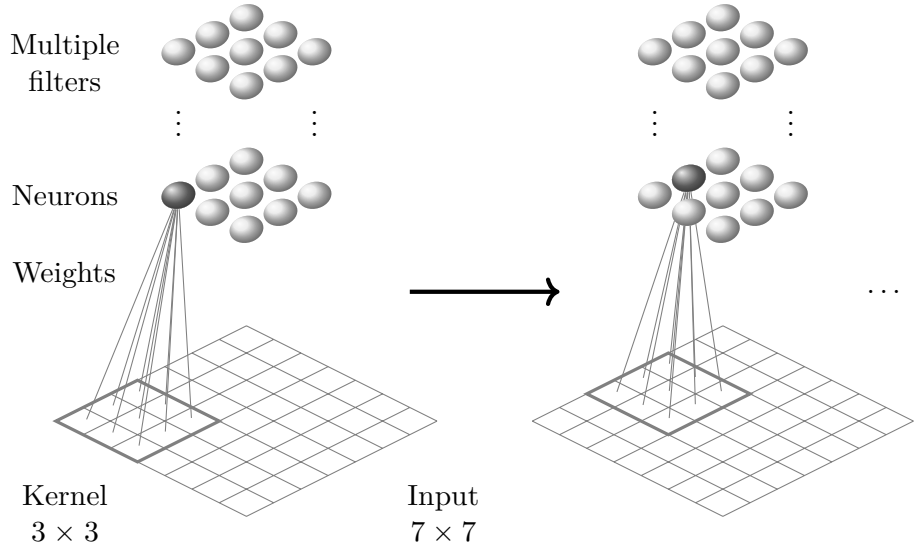

Figure 2.5: Example of convolutional layer with kernel size 3 × 3 and stride 2.

The kernel size, stride and number of filters are chosen based on the dataset. Different values may bring different results. The usual value for kernel size is between 3 × 3 and 5 × 5. The number of filters in convolutional layer is usually increasing the deeper the layer is in the network, which improves the ability of the model to detect more low level features. With that said, it is better to try multiple values and compare their results.

2.4.2 Pooling layer

The pooling layer is used to reduce the number of parameters and amount of computations in the neural network. The most common type of pooling is **max-pooling**. It splits each filter from previous convolutional layer into non-overlapping rectangles and outputs the maximum value for each rectangle. The most common shape for such rectangle is 2 × 2. That leads to down-sampling the previous activations to 25%. It extracts the most important features detected in convolutional layer. The depth remains unchanged. Another type of pooling is average pooling, which outputs average of all values in each rectangle.

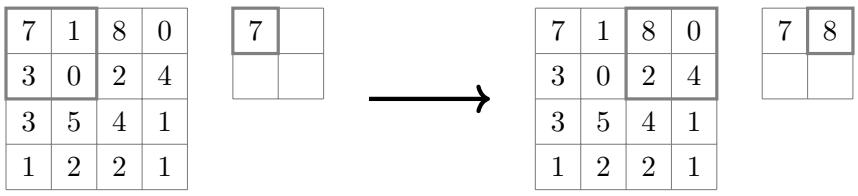

Figure 2.6: Example of 2 × 2 max-pooling with stride 2.

2.4.3 Fully connected layer

Neurons in fully connected layers have connections to all activations in the previous layers. The densely connected layers are identical to the layers in multi-layer neural network. This layer is used to further analyse the features detected in the previous convolutional or pooling layer.

2.5 Computational graph

Computational graph organizes a computation. It is a directed graph which consists of nodes and edges. Each node represents a variable or an operation. Edges represent passing the result of an operation to another operation as an operand. Computation organized in a graph can be solved parallelly by computing the subgraphs that are independent on each other.

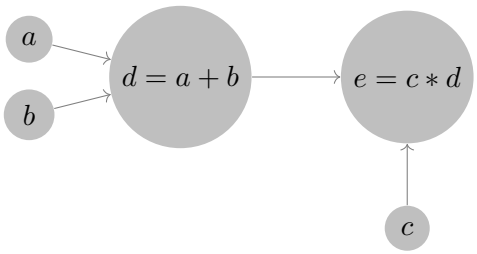

Figure 2.7: Simple computational graph.

If a complex problem is split into simpler subproblems, each subproblem can be solved only once and its solution can be stored. This technique can save computational time at the cost of memory.

Derivatives are used to calculate how much does the output changes with respect to all parameters during backpropagation of error while training a neural network. Those partial derivatives can be calculated very efficiently in a computational graph by using reverse mode differentiation. Reverse mode differentiation tracks how each node in the graph influences one output. This is used in neural networks to calculate how much does a change in input node affect the loss function. [19]

2.6 TensorFlow

TensorFlow [20, 21] is a machine learning library developed by Google. It can be used to create and execute neural network models. TensorFlow provides application programming interfaces (APIs) in Python, C++, Java and Go. TensorFlow can run on graphics processing unit to speed up the execution.

In TensorFlow, a neural network model is defined as a computational graph where nodes are tensors. A tensor is basically a multidimensional matrix. While building the graph, we define how each tensor is computed based on

other variable tensors. We can then run part of this graph to achieve desired results.

TensorBoard is a part of TensorFlow that can be used to visualise learning. Probably the most useful information during training is the error of training and testing dataset based on epoch. TensorBoard also provides a way to visualize the computational graph of the model, which may be useful for large and complicated neural network architectures.

2.7 Graphics processing unit

Graphics processing unit (GPU) is an application specific integrated circuit designed to accelerate creation of images intended for output to a display device. It provides efficient and powerful parallel computing and also high performance memory. Both of those properties can be used to accelerate machine learning. To illustrate, we can speed up training of neural network 5 to 10 times by using GPU. The exact ratio depends on the specific hardware as well as on the structure of the neural network. As described earlier, training is mostly done by computing simple formulas for lots of data. Most of the operations are performed with matrices. For example matrix multiplication can be done in parallel efficiently.

Chapter 3

Analysis

We are dealing with a regression problem. The goal is to find correct x and y positions of 15 facial keypoints. We will use mean squared error as a cost function to measure the error of proposed approaches.

3.1 Dataset

The dataset used for this thesis comes from Kaggle [22]. It contains 7049 grayscale images with resolution 96×96 pixels. Each facial keypoint is specified by x and y position in the image. The following 15 facial features are represented in the dataset:

- left_eye_center, right_eye_center,
- left_eye_inner_corner, left_eye_outer_corner,
- right_eye_inner_corner, right_eye_outer_corner,
- left_eyebrow_inner_end, left_eyebrow_outer_end,
- right_eyebrow_inner_end, right_eyebrow_outer_end,
- nose_tip,
- mouth_left_corner, mouth_right_corner,
- mouth_center_top_lip, mouth_center_bottom_lip.

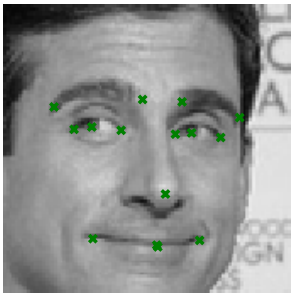

Figure 3.1: Example of an image from the dataset with marked facial keypoints.

Even though the dataset contains 7049 images, only 2140 of them have all 15 keypoints marked. We will use the pictures that have all facial keypoints present in the dataset, because we want our neural network to predict all 15 keypoints,

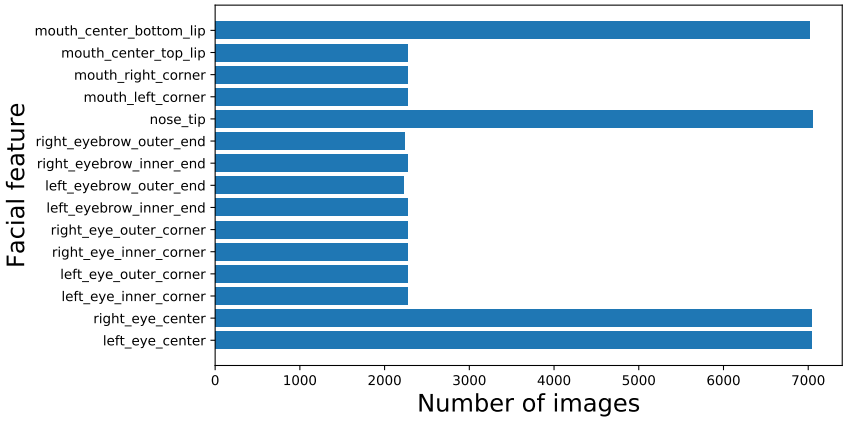

Figure 3.2: Number of images in the dataset for each facial feature.

We will split the dataset into two parts. One for training and another one for testing. By doing that, we can measure the performance of the model on training images as well as on images that the neural network never trained on. Both training error and validation error are important when analysing the model's results. The training dataset will contain 80% of the original dataset and the rest 20% will be the testing dataset. That is 1712 training images and 428 testing images.

Normalizing the inputs to the neural network is common thing to do while working with multidimensional data in different ranges. The values of pixels in input pictures are in range from 0 to 255. We will scale that to range $[0, 1]$. The positions of facial keypoints are in range from 0 to 96. We want those values to have mean 0 and variance 1. That can be achieved by simple computation $y' = \frac{y}{48} - 1$. Using normalized data can help during training of neural network while finding the gradient of loss function. Imagine if we had values from range [0,1] and [0, 1000]. In that case, the latter would have much more impact on the output of the neural network and that could lead to slower convergence.

3.2 Proposed approaches

This section proposes few approaches to facial feature detection problem by using neural networks with different structures.

3.2.1 First model

First, we will use very simple neural network. In order to feed an input image to the neural network, we have to reshape the 96×96 pixels image to a vector of $96 \times 96 = 9216$ pixel values. That vector will be the input to the neural network, which means that the input layer will have 9216 nodes. The input layer will be fully connected to the output layer. Because we are predicting 15 facial keypoints given by their x and y positions in the image, we are actually predicting 30 values. Having said that, the output layer will consist of 30 neurons, each connected with weights to 9216 input values. The output layer will not use any activation function.

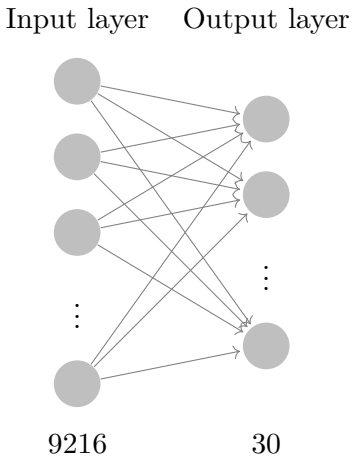

Figure 3.3: Visualization of the first model.

We do not expect this model to perform well because it is quite simple. However, it can be used to get familiar with the dataset and to discover any potential problems with it in early stages.

3.2.2 Second model

We will add two hidden layers to the previous model. The first hidden layer will contain 300 and the second 150 neurons. The hidden layers will use ReLU as an activation function which was described in subsection 2.2.1.

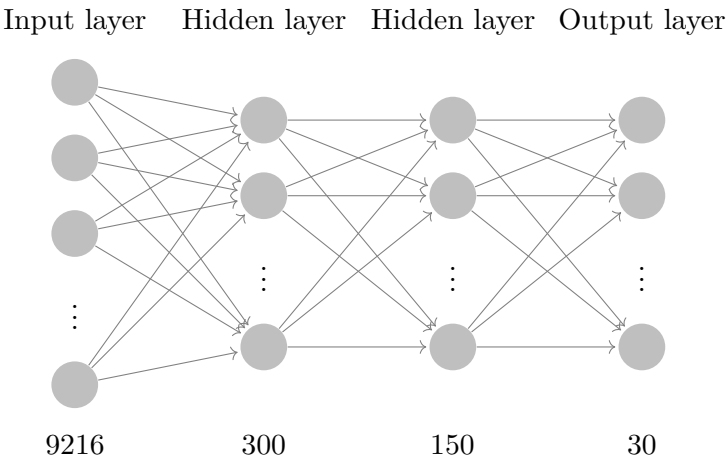

Figure 3.4: Visualization of the second model.

We expect this more complex neural network to perform better because the extra hidden layers might help the model learn more detailed relationships within the data. It will probably take longer to train due to higher number of learnable parameters. On the other hand, we might be able to improve that by experimenting with different settings.

3.2.3 Third model

This model will be using convolutional neural networks, which are state of the art in image recognition. The input data have to be reshaped into three dimensions, in this case $96 \times 96 \times 1$. The model contains three convolutional layers followed by one fully connected layer and an output layer. Convolutional layers and dense layer will use ReLU as their activation function.

Layer	Input shape	Filters	Kernel size	Stride	Output shape
input	-	-	-	-	$96 \times 96 \times 1$
conv1	$96 \times 96 \times 1$	4	5×5	1	$96 \times 96 \times 4$
conv2	$96 \times 96 \times 4$	6	5×5	2	$48 \times 48 \times 6$
conv3	$48 \times 48 \times 6$	8	4×4	2	$24 \times 24 \times 8$
flatten	$24 \times 24 \times 8$	-	-	-	4608
dense	4608	-	-	-	250
output	250	-	-	-	30

Table 3.1: Layer-wise description of the third model.

This model uses convolutional layers for down-sampling. Notice the different spatial dimensions (height, width) in the second and third convolutional layer. That is caused by using stride 2. The aim of down-sampling is to reduce dimensionality for computational efficiency. On the other hand, we do not want to lose too much information. The purpose of the fully connected layer of 250 neurons is to detect relationships between the features detected by previous layers.

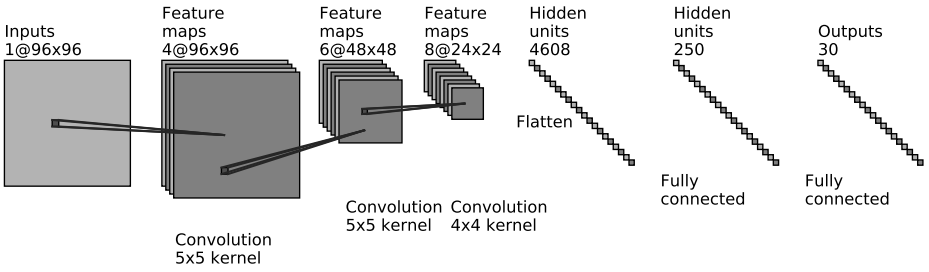

Figure 3.5: Visualization of the third model.

This is the first model that takes advantage of multidimensional input data. With that in mind, we expect an increase in performance.

3.2.4 Fourth model

Another way to reduce the amount of information from convolutional layers is by pooling. This model uses max-pooling with receptive field 2×2 and stride 2 after every convolutional layer. Using this down-sampling method leads to 75% reduction in information. Using bigger receptive fields (3×3, ...) might be too lossy and it could cause worse results. By using max-pooling, the most important features are kept while the less influential are ignored. This is done to reduce computational cost while preserving important information.

Layer	Input shape	Filters	Kernel size	Stride	Output shape
input	-	-	-	-	$96 \times 96 \times 1$
conv1	$96 \times 96 \times 1$	4	5×5	1	$96 \times 96 \times 4$
pool1	$96 \times 96 \times 4$	-	2×2	2	$48 \times 48 \times 4$
conv2	$48 \times 48 \times 4$	6	5×5	1	$48 \times 48 \times 6$
pool2	$48 \times 48 \times 6$	-	2×2	2	$24 \times 24 \times 6$
conv3	$24 \times 24 \times 8$	8	4×4	1	$24 \times 24 \times 8$
pool3	$24 \times 24 \times 8$	-	2×2	2	$12 \times 12 \times 8$
flatten	$12 \times 12 \times 8$	-	-	-	1152
dense	1152	-	-	-	250
output	250	-	-	-	30

Table 3.2: Layer-wise description of the fourth model.

The result of this model should be similar to the previous one with few differences. Because it uses down-sampling on all three layers the number of used neurons is slightly lower compared to the previous model. Due to that, the amount of weights will also be lower. This will reduce the computational complexity by a very small margin, which may lead to faster training.

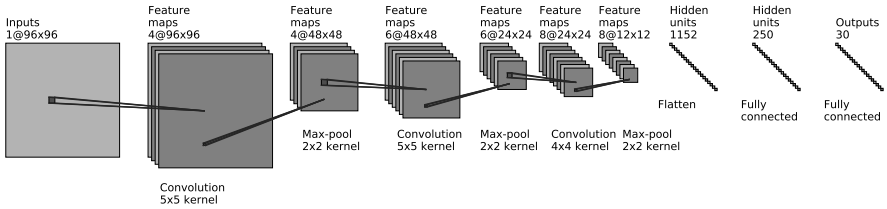

Figure 3.6: Visualization of the fourth model.

3.3 Realisation

We will use **Python** [23] programming language to implement the above-mentioned approaches. The exact version used is 3.6.4. One of Python's main advantages is its large standard library which contains huge amount of packages with a wide range of functionality. We will use the following packages:

- **TensorFlow 1.6.0** [21] provides API (application programming interface) to build neural networks models. It allows us to take advantage of GPU's computational power to speed up the training of neural networks.
- **NumPy 1.14.1** [24] is a package for scientific computing. Among other things it provides N-dimensional array object or operations with matrices.
- **Matplotlib 2.1.2** [25] is a 2D plotting library. It can be used to generate plots, histograms, scatter plots, etc.
- **Pandas 0.22.0** [26] is a library that offers data manipulation and analysis. We will use it to manipulate with the dataset.

Chapter 4

Results

4.1 First model

For a simple model that is using only 30 neurons, its results are quite good. Even though it is not a perfect performance, image recognition is not an easy task.

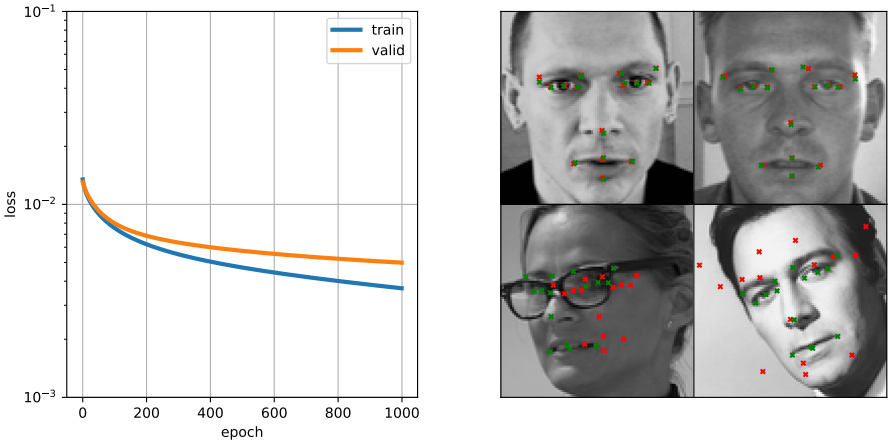

Figure 4.1: Learning curve of the first model and example of performance on testing dataset.

As we can see in the figure above, the model reached mean squared error of 0.005 on the testing dataset after 1000 epochs. It is also slightly overfitting, which means that the model is performing better on training data compared to testing data. Simply said the model fits the training data and it is not able to generalize that well. We can attempt to fix that by using regularization technique called dropout described in subsection 2.2.5 or by using more data for training. Considering the fact that this is the first and simplest model, we will experiment with such techniques on more complex models that have better baseline results.

The right half of the figure contains examples with best and worst performance. The images on top are the best results while the images on the bottom

are the worst. The top left image has lowest error and the bottom left has the highest error. The green markers are the true positions of facial keypoints while the red markers are the values outputted by the neural network. To conclude, the model was able to predict some examples almost perfectly. However, we can see that it has trouble with rotated faces and also with people wearing glasses.

4.2 Second model

As we can see in the graph, the performance of this model is slightly better. In the beginning, it was learning fast, but the pace went down rapidly. By adding two layers we added huge amount of learnable parameters. Because of that, it takes more epochs to train and the convergence of loss function is slow.

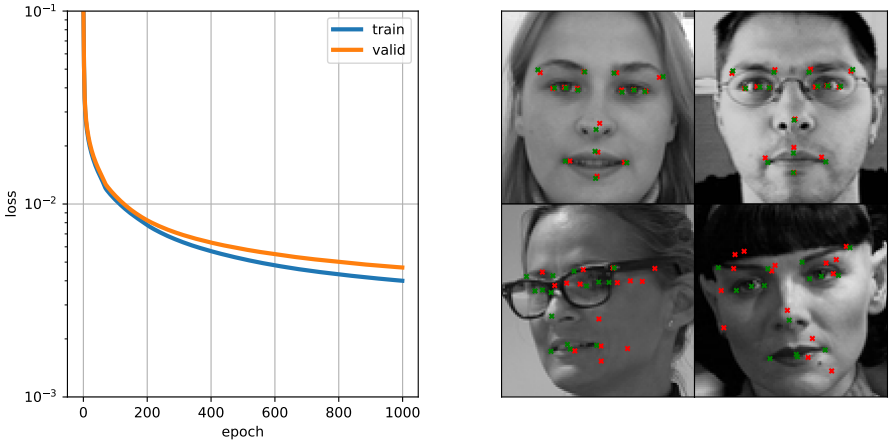

Figure 4.2: Learning curve of the second model and example of performance on testing dataset.

The final value for training loss is 0.0040 and 0.0047 for validation loss. To improve the model's performance we can either let it train for more epochs or we can experiment with different optimizer. Using Adam optimizer instead of plain gradient descent might lead to faster convergence.

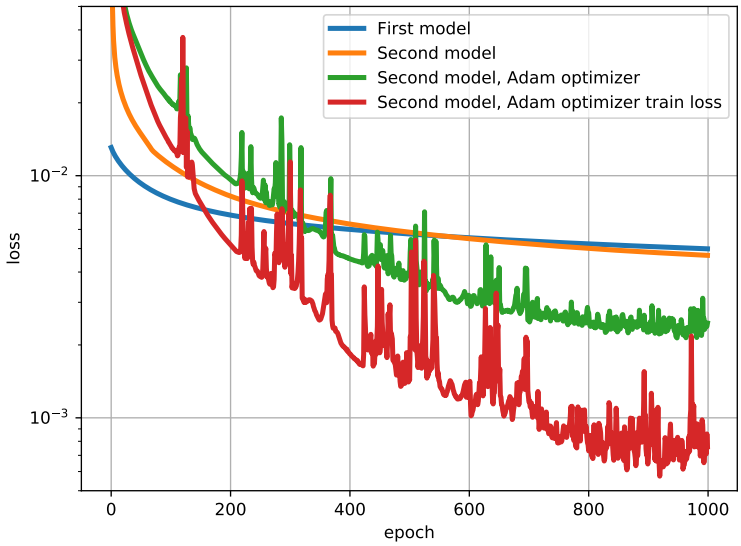

Figure 4.3: Validation loss of previous approaches compared to the learning curve of the second model using Adam optimizer.

From the figure above it is noticeable that Adam optimizer performs better than vanilla gradient descent. As said in subsection 2.2.4, it takes into account the gradient of previous training steps in order to speed up the convergence of loss function. The minimal validation loss of this model is 0.0024 and training loss is 0.00086. It is obvious from the figure that this model is overfitting.

4.3 Third model

We will use Adam optimizer for further models because plain gradient descent algorithm proved less effective for deep neural networks.

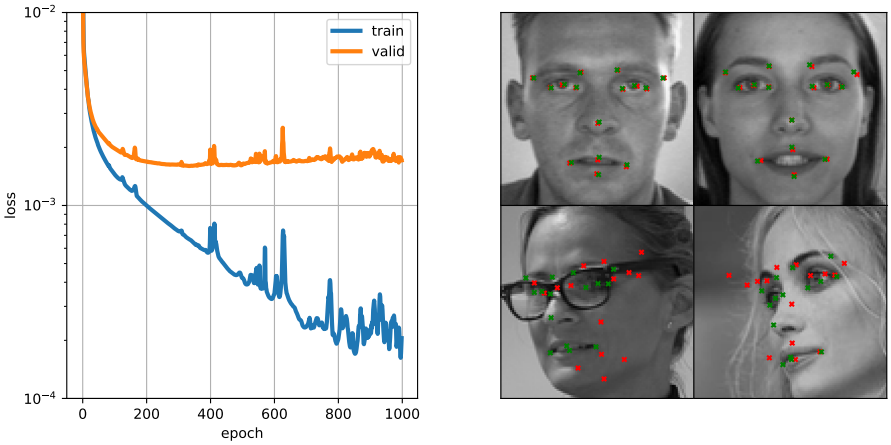

Figure 4.4: Learning curve of the third model and example of performance on testing dataset.

As we can see in the figure, the model is learning on training data pretty well. If we compare the training loss to validation loss, it is evident that the model is overfitting. We can try to solve that by using aforementioned regularization technique called dropout. Basically, with certain probability, we remove neurons during each training step, which helps the model to generalize. We have to be careful when choosing the dropout probability. If it is too high, it might make the model unable to learn anything at all. We will use 25% chance to drop a neuron in the last hidden layer during training. Using dropout will make the training slightly harder. During testing, we will not drop any neurons. As you can see in the figure above, the model reached lowest validation error of roughly 0.0016 after 300 epochs. After that point, it only gets worse. There is not much point to train after the validation error stops improving, but it is hard to predict the model's exact behaviour beforehand.

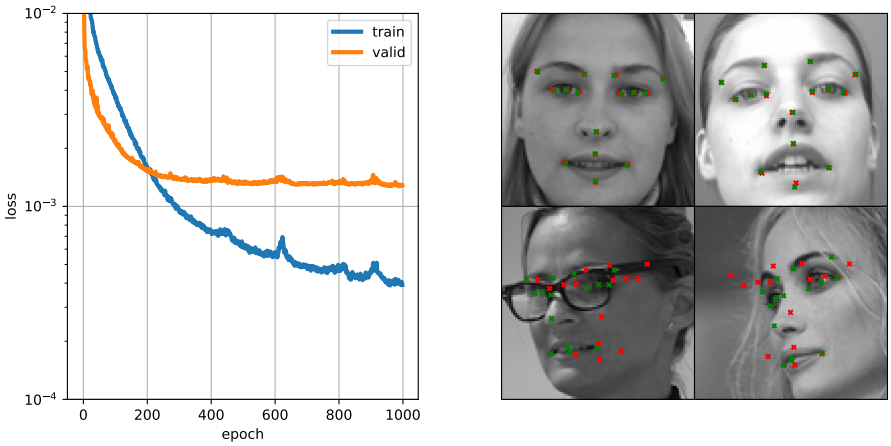

Figure 4.5: Learning curve of the third model with added dropout and example of performance on testing dataset.

On the loss plot, we can notice that roughly for the first 200 epochs the training loss is greater than validation loss. That is caused by the dropout, as the model is only using roughly 75% of its neurons in the last layer. Because the neurons to drop are chosen randomly, sometimes the important ones are taken out, which leads to worse training results. On the other hand, it makes the rest of the neurons improve.

Adding the dropout led to improvement in validation loss. The final values are 0.00039 for training loss and 0.00128 for validation loss. Unfortunately, using 25% dropout did not fully solve the model's problem with overfitting. We might be able to further reduce overfitting by increasing the dropout percentage or by adding more examples.

4.4 Fourth model

This model uses almost the same parameters as the previous one. The main difference is the use of max-pooling layers. It took this model roughly 10% less computational time to train using GPU compared to the previous one, which was caused by the use of pooling layers for down-sampling.

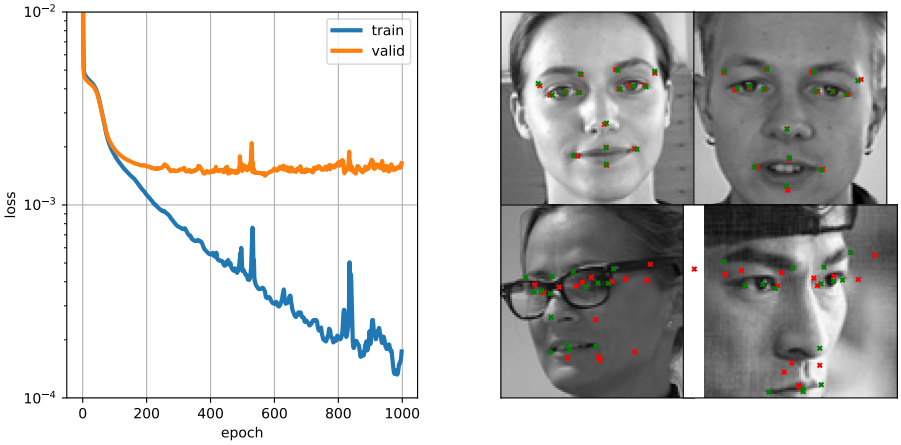

Figure 4.6: Learning curve of the fourth model and example of performance on testing dataset.

This model started performing better than previous around epoch 200. Its lowest validation loss was 0.00146 while the training loss 0.00032 during the same epoch. This means that the model is also overfitting. We can use the same method as in previous approach that proved successful, which was dropout.

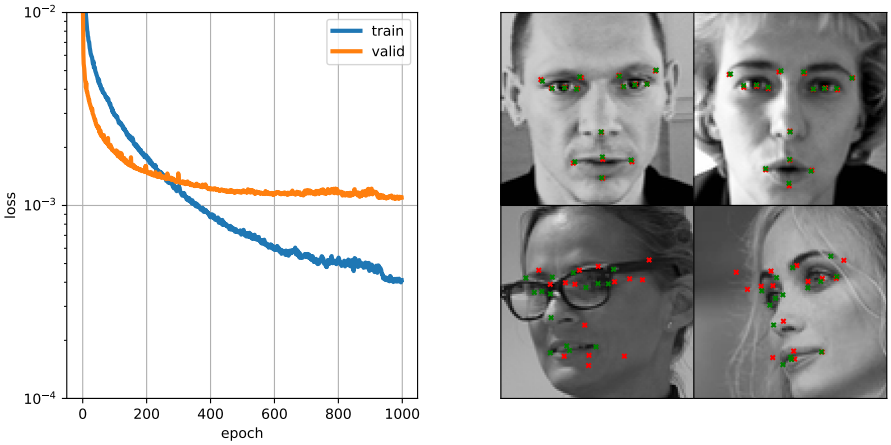

Figure 4.7: Learning curve of the fourth model with added dropout and example of performance on testing dataset.

We will use 25% chance to drop a neuron in the fully connected layer for this experiment. As you can see in the figure above, adding dropout led to improvement in performance. Even though it did not stop the model from overfitting, it certainly helped. This model was able to reach 0.0011 validation loss and 0.00041 training loss after 1000 epochs.

4.5 Results of proposed approaches

So far we have tried multiple proposed models. We also experimented with some of them which lead to different results. The results are summarized in the following table.

Model	Lowest validation loss	Training loss	Epoch
First	0.005 00	0.003 70	1 000
Second	0.004 70	0.004 00	1 000
Second, Adam optimizer	0.002 40	0.000 86	1 000
Third	0.001 60	0.000 70	300
Third, dropout	0.001 28	0.000 39	1 000
Fourth	0.001 46	0.000 32	600
Fourth, dropout	0.001 10	0.000 41	1 000

Table 4.1: Comparison of achieved results.

From the results we can conclude that the closer to zero we are getting, the harder it is to make an improvement. We were able to reduce the validation error roughly 5 times using the state of the art neural networks in image recognition.

All of the models above had worst performance measured on the image of a woman with glasses. Even though the image may not seem hard for humans to predict, the used approaches seem to have hard time analysing it. That might be caused by the fact that there is simply not enough examples of poses like this for the models to learn its key features. We can compare this image to the heat-map of all facial features in the testing dataset.

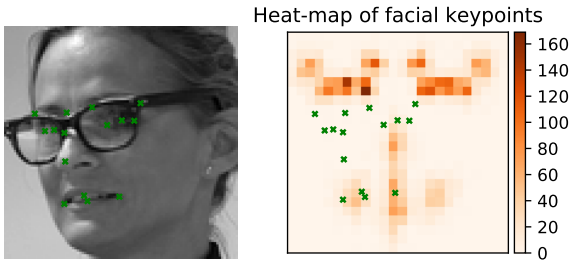

Figure 4.8: Heat-map of facial features compared to the facial features of the woman with glasses.

It is observable from the heat-map that this image differs from the others.

One way to improve the performance on this type of images is by adding more examples similar to this one. Unfortunately, we do not have such images. It is also interesting to note that although the error of the last approach is 78% lower than the error of the first model, the error measured on the woman with glasses decreased only by 20%.

We can also analyse the time spent on training of the proposed models and compare training purely on central processing unit (CPU) to training using GPU.

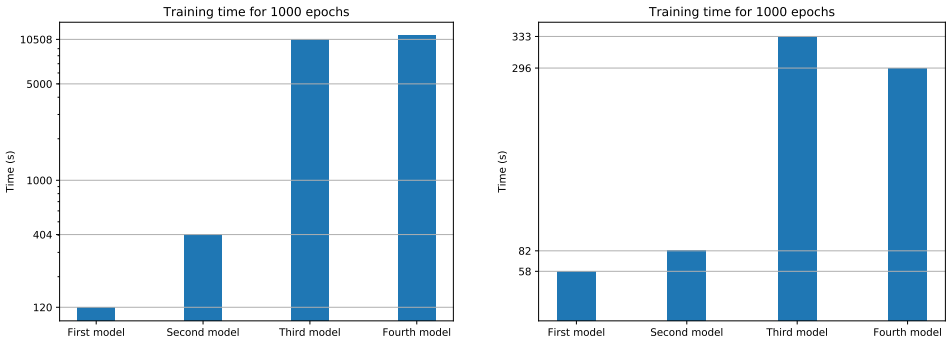

Figure 4.9: Comparison of time used to train proposed models. Time taken to train using purely CPU on the left and time taken to train with the help of GPU on the right.

The difference in training time on CPU and GPU for the first model is not that huge, simply because there is not much to parallelize when the model consists of 30 neurons. As the number of neurons goes up and the model starts getting more complex, we can see more noticeable difference. It took 404 seconds to train the second model on CPU compared to 82 seconds with the help of GPU. The last two models show enormous difference in training time. It takes roughly 31 times longer to train the third model just by using CPU. For the fourth model, the difference is even higher. It is 38 faster to train the fourth model using GPU. Imagine having to wait three hours whenever you want to experiment with the parameters of the model. Thanks to efficient parallel computing of GPU, we can try using various values of parameters and it will not take ages to figure out what combinations of parameters work better than the others.

Another notable difference between the models is in scoring time. That is how long does it take the model to predict facial keypoints on images. Scoring time is something we should also consider while choosing model for certain application. Depending on the circumstances, we might want to predict multiple images at once or predict one image at a time, for example while using video from web camera in real time. The figure below shows measured values for both scenarios.

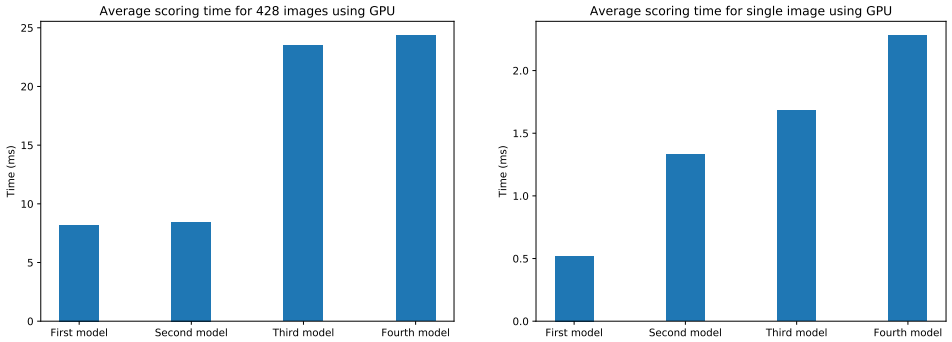

Figure 4.10: Scoring time of proposed approaches.

As we can see in the figure, it takes the CNN models approximately three times longer to process a batch of images compared to the previous models. Nonetheless, it is still very fast if we consider the fact that it takes the model on average around 24 milliseconds to process 428 images. However, it takes the fourth model roughly one-tenth of that to predict a single image. Faster results on large batches are achieved by efficient parallel computing. Feeding an input forward through a neural network is mostly done by operations with matrices, which can be parallelized, especially when using multiple input images.

If we want to use our model for example on mobile devices we have to consider multiple choices. If we choose simpler model that does not require that much computational power but its performance is worse, the users might get annoyed because the model does not perform well enough. On the other hand, if we use more computationally complex model with better performance, users might not like it because it slows down their device and drains its battery.

4.6 Reducing overfitting

Almost all of proposed models were overfitting. We tried to solve that by using dropout, but we were only partially successful. Another way to deal with overfitting is to use more training data. One way to do that is by finding images of faces on the Internet and marking all 15 positions manually. We also could slightly alter the images we have available. We can simply double the size of our dataset by horizontally flipping its images.

In order to get meaningful data after horizontally flipping an image we also have to change the positions of certain marked keypoints. For example, the y position of left eye needs to be swapped with y position of right eye.

Target facial keypoint		Target facial keypoint
left_eye_center	↔	right_eye_center
left_eye_inner_corner	↔	right_eye_inner_corner
left_eye_outer_corner	↔	right_eye_outer_corner
left_eyebrow_inner_end	↔	right_eyebrow_inner_end
left_eyebrow_outer_end	↔	right_eyebrow_outer_end
mouth_left_corner	↔	mouth_right_corner

Table 4.2: Facial keypoints whose y position needs to be swapped while horizontally flipping the image.

Because we flipped the x axis of the image, we also have to adjust the x values of each facial keypoint. That can be done by applying simple formula: $x' = 96 - x$.

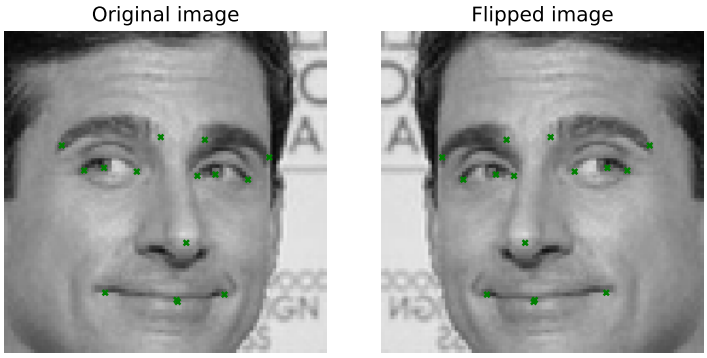

Figure 4.11: Example of a flipped image.

From the figure above, we can see that everything works as intended and the image was successfully flipped. Another technique we can use to generate more data is image transformation. Using this approach, we will be able to create a huge amount of pictures that are similar to the original images but slightly different, which helps with overfitting. On the other hand, it requires more computational time than simply flipping images.

By horizontally flipping the pictures, we doubled the size of our dataset. Now we have $2 \cdot 2140 = 4280$ images. We will split this dataset into two, one for training and one for measuring the error on images that were not shown to the neural network. The training dataset will consist of 3424 pictures and the testing dataset will have 856 images.

To further increase the amount of examples we can use image transformations. By shifting image by few pixels horizontally and vertically we can create slightly different image which can be used during training. The number of pixels to shift by will be chosen randomly from range $(-9.6, 9.6)$. The value 9.6 represents $\frac{1}{10}$ of the image size. Negative value means that the image will be shifted to left and positive value means shift to right. The image will be shifted both vertically and horizontally. We also have pay attention to the positions of facial keypoints. All keypoints have to remain in

the shifted image. In order to achieve this, we can use more sophisticated approach. Instead of choosing random value from range $(-9.6, 9.6)$, we can use the following range.

$$(\max(-9.6, -x_{\min}), \min(9.6, 96 - x_{\max}))$$

Where $x_{\min}$ is the lowest x value of all keypoint and $x_{\max}$ is the highest x value of all keypoints. Same method can be used when randomly choosing value for vertical shift with minor change. Instead of using x positions we have to use y positions.

Such transformation can be expressed in the form of matrix multiplication followed by vector addition. This type of transformation is called affine transformation and it represents relation between two images. We can express the transformation described above by the following matrix:

$$M = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \end{bmatrix},$$

where t_x is shift of x axis and t_y is the shift of y axis. This matrix can be used to transform the image as well as the positions of facial keypoints. To address the problems of the previously trained models with rotated faces we will also use affine transformation to perform rotation of images. Rotation can be described by the following matrix:

$$M = \begin{bmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{bmatrix},$$

where α is the rotation angle.

We will use Python library for computer vision called OpenCV [27] to transform the images. To be more specific, we will use function called **cv2.warpAffine()** to perform the image transformation. This function applies an affine transformation given by matrix M to an image. To change the positions of facial keypoints we simply have to add t_x to all x positions and t_y to all y positions.

OpenCV provides rotation with adjustable center of rotation. We want to rotate in the middle of the picture. To calculate the transformation matrix for rotation of an image we will use function **cv2.getRotationMatrix2D()**, which takes the position of center and angle of rotation as an argument. We will use randomly selected angle from range $(-10°, 10°)$. The function creates matrix M with shape 2×3. This matrix can be used to transform the image similarly to the previous example by using the function **cv2.warpAffine()**. To calculate the new positions of facial keypoints, we have to split the matrix $M_{2\times3}$ into matrix $A_{2\times2}$ and vector $B_{2\times1}$.

$$M = \begin{bmatrix} a_{00} & a_{01} & b_{00} \\ a_{10} & a_{11} & b_{10} \end{bmatrix} = \begin{bmatrix} A & B \end{bmatrix}$$

We have to organize the positions of facial keypoints into matrix by putting their x values into first row and y values in the second row.

$$P = \begin{bmatrix} x_0 & x_1 & \dots & x_{14} \\ y_0 & y_1 & \dots & y_{14} \end{bmatrix}_{2\times15}$$

The following formula is used to calculate the new positions of facial keypoints in the rotated image.

$$P' = P \cdot A + B$$

We will use Python library called NumPy [24] to perform the matrix multiplication and vector addition mentioned above.

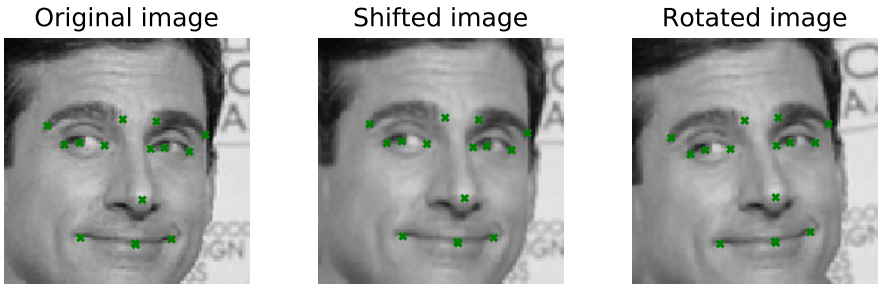

Figure 4.12: Example of transformed images.

As we can see the transformations were applied correctly. It is also important to note that the empty space created by the transformation is filled by mirroring the image. It is noticeable in the middle picture above on its left side.

4.7 Final result

With the assumption that humans can solve the facial keypoints detection problem perfectly we can conclude from the results of proposed approaches that all of the models have high bias. High bias means that there is big difference between the model's performance on training data compared to human's performance. The last two models also suffer from high variance, which means that there is a huge difference between performance on training dataset and validation dataset. We can try to solve high bias by changing the neural network's structure. For the following experiment we will use the fourth model with minor changes. To be more specific, we will add one fully connected layer between the fully connected layer and output layer. This will probably further increase variance, but we can deal with high variance by using more training data. For the following models, we will be using bigger dataset. To keep the total time required to train a network in control we will have to add more data in a smart way. First, we will flip the original dataset and split it into training and testing dataset. Then we will transform the training dataset 10 times. Five times by shifting and five times by rotating, which will create 34240 new examples. We will add those transformed images to the training dataset. The final amount of training pictures is 37664.

Another way to lower the time required to train the model is by using **changing batch size**. In the beginning of training we will use small batches.

Even though the predicted gradient might not be as accurate as if we used bigger batch size, it is not a big deal in the beginning. Doing this will lead to more weight updates per time unit, which speeds up the convergence. As we get closer to the minimal loss we want to take more precise steps when tuning the weights. That can be achieved by using more examples when calculating the gradient which brings more accurate results. We will use the following formula to calculate the batch size: $batchsize = \lceil \frac{epoch}{125} \rceil \cdot 64$.

We will split each epoch into 30 training steps. In every training step, we will randomly select batch from the training dataset. We will not use any dropout for the following training because we first want to see how the model performs.

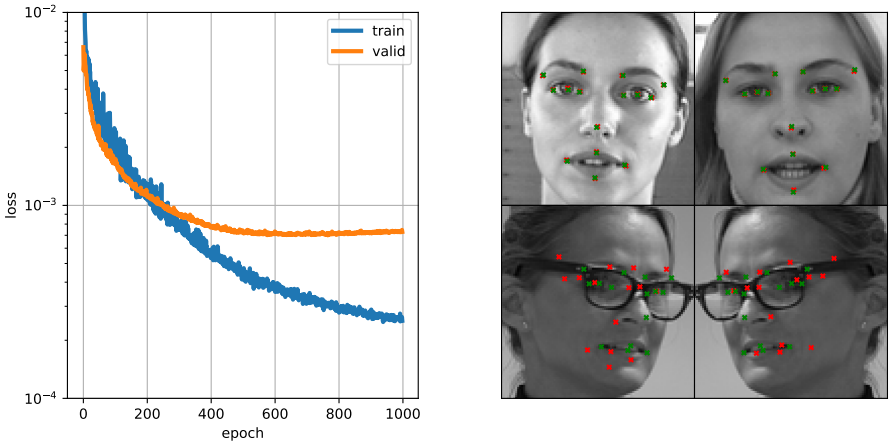

Figure 4.13: Learning curve of the fifth model and example of performance on testing dataset.

As we can see in the figure, we were able to increase the model's performance by using more examples and by changing the structure. However, the model is overfitting. Now that we know how the neural network performs, we can propose dropout probabilities. To further decrease variance we will add dropout to the fully connected layers and to the max-pooling layers. Because we have high amount of training images, we propose small dropout probabilities.

Layer	Dropout probability
pool1	5%
pool2	5%
pool3	5%
dense	10%

Table 4.3: Dropout probabilities of the fifth model.

We have to be careful while choosing dropout probability for max-pooling layers, because the layer contains important information, which is why this

model uses only 5% dropout rate for each max-pooling layer. We expect to achieve lower variance by adding dropout.

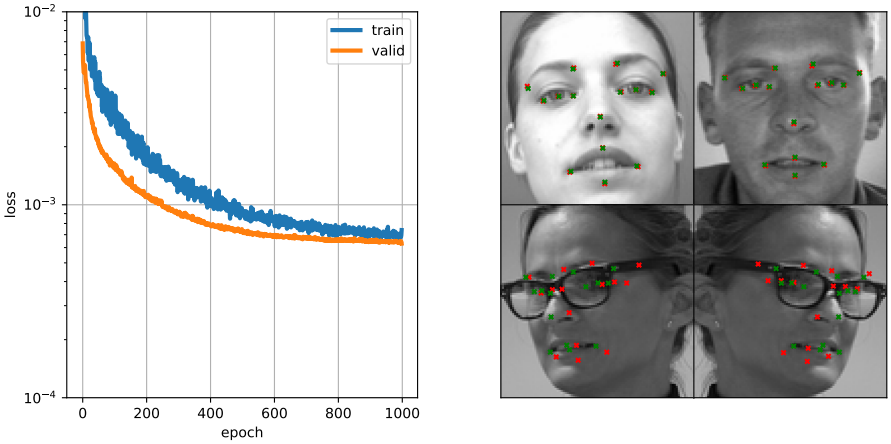

Figure 4.14: Learning curve of the fifth model with added dropout and example of performance on testing dataset.

The plot shows that we were able to lower the variance while also slightly lowering the bias. It might be worth to try using Leaky ReLU instead of ReLU as an activation function. That may improve the model's performance due to the dying ReLU problem which was described in subsection 2.2.1. The Leaky ReLU activation function if defined by $f(x) = \alpha \cdot x$ for $x < 0$ and $f(x) = x$ for $x \geq 0$, where α is a small number and its exact value depends on the implementation. Tensorflow uses $\alpha = 0.2$ [28] as a default value while Keras [29] which is high-level neural networks library uses $\alpha = 0.3$ [30].

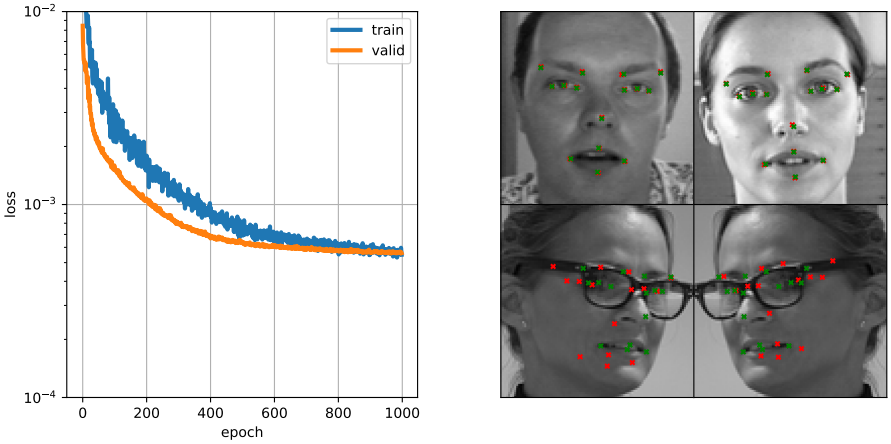

Figure 4.15: Learning curve of the fifth model using Leaky ReLU.

Using Leaky ReLU lead to small improvement in performance which was probably caused by non-zero gradient of activation function $f(x)$ for $x < 0$.

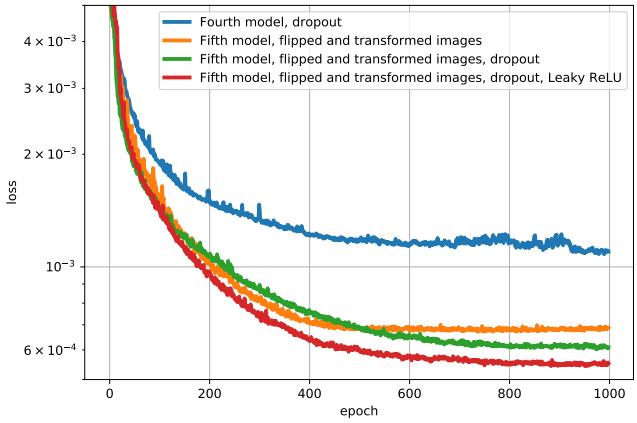

Figure 4.16: Comparison of validation loss among multiple variations of the fifth model.

Model	Validation loss	Training loss	Epoch	Time (s)
Fifth	0.000 687	0.000 204	1 000	1 413
Fifth, dropout	0.000 604	0.000 576	1 000	1 381
Fifth, dropout, Leaky ReLU	0.000 530	0.000 542	1 000	1 605

Table 4.4: Results of the fifth model.

To sum up, by adding one fully connected layer to the fourth model, using different activation function, adding dropout and using image transformations we were able to lower the error of the fourth model to more than a half. On the other hand, the time required to train the model rose more than 5 times.

It is also interesting to note that we were able to detect facial features with different precisions. The following figure shows validation error of few selected facial features.

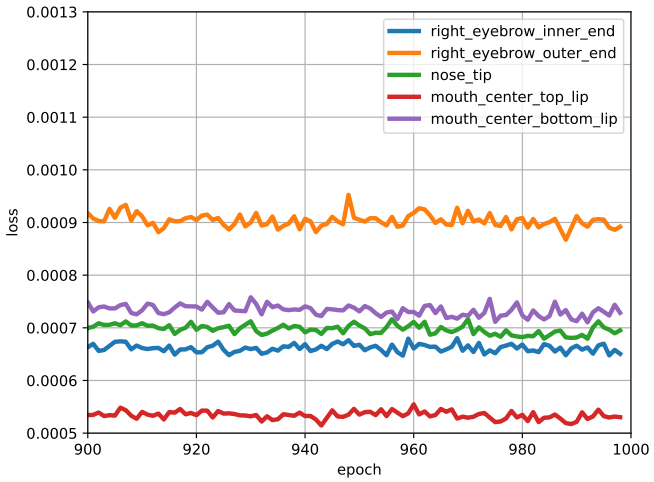

Figure 4.17: Error of few selected facial features.

As we can see in the figure, it is harder for the model to predict the center of bottom lip compared to the center of upper lip. That may be because the dataset contains pictures of people talking with their mouths open and while your mouth is open, the bottom lip moves much more compared to bottom lip. Basically the position of bottom center lip differs more than the position of upper lip. We can also see that the model is able to predict center of top lip much more precisely than outer end of right eyebrow.

To make the performance of the final model mode human-friendly, we can calculate the average precision of its predictions. By calculating the root mean square error and scaling it back to the original value (because we normalized our data, which was mentioned in section 3.1) we can determine the average precision for each outputted value by: $y_p = \sqrt{0.00053} \cdot 48 \approx 1.1px$. Because each keypoint consists of two positions, that is its y and x position, we can compute the average precision for facial feature by using Pythagoras' theorem: $p = \sqrt{y_p^2 + y_p^2} \approx 1.56px$. To conclude, the final model is able to predict a position of facial keypoint with precision 1.56 pixels on average. As mentioned above, some keypoints are easier to predict and some are harder. It is also very important to note that this is measured on the validation dataset. Real world results might be similar, but it is not guaranteed.

4.8 Ideas for experiments

In order to further improve the model we could experiment with the structure of the model. For example changing the sizes of kernels in convolutional layer or using more filters could lead to different results. We could also try more methods of image transformations, for example scaling, cropping, shearing or perspective transformation. Another idea is to use more neurons while increasing the dropout probability.

4.9 Demonstration

Predicted positions of facial features can be used in several ways. One of them is face recognition which is quite complicated task that we will not get into in this thesis. Another way to use our solution is to alter images or videos. For example by placing image of sunglasses to a picture of face based on facial keypoints positions predicted by our neural network. We can realize that by using OpenCV [27] that allows us to edit images using Python. We can divide this method into multiple steps:

- First, we need to find images containing faces. In this section, we will use images downloaded from Pixabay [31] which is a website that offers free images.
- Then, we will manually crop faces from those images with their width to height ratio 1 to 1 and save them to disk drive.
- Next, we will load those images as grayscale, resize them to 96×96 and rescale values of pixels to $[0, 1]$. We will use those images as an input to our neural network.
- After feeding those images through the neural network, we have to rescale the output from $[-1, 1]$ to $[0, 96]$. That can be done by simply multiplying all inputs by 48 and adding 48. Next, we also have to take into account the original size of the image. If the resolution of the original image was for example 192×192, we have to multiply all positions of facial keypoints by 2.
- Based on those positions we can now place certain images into the original image.

To illustrate, we can place image of clown's nose on the position of nose tip. We can also scale the size of the nose relatively based on, for example, the distance of outer eye corners.

Figure 4.18: Faces with predicted facial keypoints on the left and altered image on the right.

As we can see in the figure above, we were able to achieve decent results. Because cropping the faces manually from images is quite time consuming, we can take advantage of Viola-Jones object detection framework [32]. OpenCV provides an implementation of Viola-Jones algorithm that can be used to detect faces in images [33]. Viola-Jones algorithm is able to achieve high detection rate and high speeds. That also allows us to use this method while working with videos, because video is basically a series of images.

Figure 4.19: Detected faces using Viola-Jones algorithm with predicted facial keypoints on the left and altered image on the right.

The figure above shows an image containing multiple faces. The green rectangles represent found faces using the Viola-Jones algorithm and the red dots represent the predicted facial keypoints by our model. We can use those predicted positions to alter the original image, which is displayed on the right side of the figure.

Chapter 5

Conclusion

We have proposed multiple approaches to facial feature detection problem in this thesis. We have used several types of artificial neural networks, including the state of the art in image recognition. The results have shown various upsides and downsides of explored solutions. The results have also shown that straightforward use of neural networks did not perform well. The proposed models suffered from high bias as well as high variance. After further analysing the results of proposed approaches we suggested several changes that aimed to improve the final outcome.

We have shown a few techniques that can be applied to image recognition tasks. Those methods have proved to be quite effective for our problem. Generalization technique named dropout turned out to be successful while dealing with overfitting. Another approach often used in order to reduce overfitting is increasing the size of used dataset. We have shown several ways to do that. By applying affine transformations to images we were able to expand the size of our dataset.

We have addressed the variance problem of proposed approaches by increasing the size of our dataset. We have lowered the bias of the final approach by adjusting the structure of the best performing proposed model. By combining aforementioned changes with the use of dropout technique, we were able to further improve our solution. To conclude, the final results have shown that facial feature detection problem is satisfactorily solvable by convolutional neural networks.

Bibliography

1. TAO, Li. *Just Jaywalked? Check Your Mobile Phone for a Message from Police* [online]. 2018 [visited on 2018-04-16]. Available from: `http://www.scmp.com/tech/china-tech/article/2138960/jaywalkers-under-surveillance-shenzhen-soon-be-punished-text`.
2. LO, Kinling. *In China, These Glasses Are Helping to Catch Criminals* [online]. 2018 [visited on 2018-04-16]. Available from: `http://www.scmp.com/news/china/society/article/2132395/chinese-police-scan-suspects-using-facial-recognition-glasses`.
3. CHEN, Stephen. *China Plans Giant Facial Recognition Database to ID Its 1.3bn People* [online]. 2017 [visited on 2018-04-18]. Available from: `http://www.scmp.com/news/china/society/article/2115094/china-build-giant-facial-recognition-database-identify-any`.
4. FREUND, Yoav; SCHAPIRE, Robert E. Large Margin Classification Using the Perceptron Algorithm. In: [online]. ACM Press, 1998, pp. 209–217 [visited on 2018-04-22]. ISBN 978-1-58113-057-7. Available from DOI: `10.1145/279943.279985`.
5. ROSENBLATT, F. *The Perceptron: A Probabilistic Model for Information Storage and Organization.*
6. *CS231n Convolutional Neural Networks for Visual Recognition* [online] [visited on 2018-04-18]. Available from: `http://cs231n.github.io/neural-networks-1/`.
7. *Derivation: Derivatives for Common Neural Network Activation Functions* [online]. 2014 [visited on 2018-05-05]. Available from: `https://theclevermachine.wordpress.com/2014/09/08/derivation-derivatives-for-common-neural-network-activation-functions/`.
8. RUMELHART, David E.; HINTON, Geoffrey E.; WILLIAMS, Ronald J. Neurocomputing: Foundations of Research. In: ANDERSON, James A.; ROSENFELD, Edward (eds.). Cambridge, MA, USA: MIT Press, 1988, chap. Learning Representations by Back-propagating Errors, pp. 696–699. ISBN 0-262-01097-6. Available also from: `http://dl.acm.org/citation.cfm?id=65669.104451`.

9. RUDER, Sebastian. An Overview of Gradient Descent Optimization Algorithms [online]. 2016 [visited on 2018-04-30]. Available from arXiv: 1609.04747 [cs].
10. DARKEN, C.; CHANG, J.; MOODY, J. Learning Rate Schedules for Faster Stochastic Gradient Search. In: *Neural Networks for Signal Processing II Proceedings of the 1992 IEEE Workshop*. 1992, pp. 3–12. Available from DOI: 10.1109/NNSP.1992.253713.
11. QIAN, Ning. On the Momentum Term in Gradient Descent Learning Algorithms. *Neural Networks* [online]. 1999, vol. 12, no. 1, pp. 145–151 [visited on 2018-05-05]. ISSN 0893-6080. Available from DOI: 10.1016/S0893-6080(98)00116-6.
12. DUCHI, John; HAZAN, Elad; SINGER, Yoram. Adaptive Subgradient Methods for Online Learning and Stochastic Optimization*, pp. 39.
13. ZEILER, Matthew D. ADADELTA: An Adaptive Learning Rate Method [online]. 2012 [visited on 2018-05-05]. Available from arXiv: 1212.5701 [cs].
14. KINGMA, Diederik P.; BA, Jimmy. Adam: A Method for Stochastic Optimization [online]. 2014 [visited on 2018-05-05]. Available from arXiv: 1412.6980 [cs].
15. SRIVASTAVA, Nitish; HINTON, Georey; KRIZHEVSKY, Alex; SUTSKEVER, Ilya; SALAKHUTDINOV, Ruslan. Dropout: A Simple Way to Prevent Neural Networks from Overfitting, pp. 30.
16. SZEGEDY, Christian; WEI LIU; YANGQING JIA; SERMANET, Pierre; REED, Scott; ANGUELOV, Dragomir; ERHAN, Dumitru; VANHOUCKE, Vincent; RABINOVICH, Andrew. Going Deeper with Convolutions. In: [online]. IEEE, 2015, pp. 1–9 [visited on 2018-04-30]. ISBN 978-1-4673-6964-0. Available from DOI: 10.1109/CVPR.2015.7298594.
17. KRIZHEVSKY, Alex; SUTSKEVER, Ilya; HINTON, Geoffrey E. ImageNet Classification with Deep Convolutional Neural Networks. *Communications of the ACM* [online]. 2017, vol. 60, no. 6, pp. 84–90 [visited on 2018-04-30]. ISSN 00010782. Available from DOI: 10.1145/3065386.
18. SIMONYAN, Karen; ZISSERMAN, Andrew. Very Deep Convolutional Networks for Large-Scale Image Recognition [online]. 2014 [visited on 2018-04-30]. Available from arXiv: 1409.1556 [cs].
19. OLAH, Christopher. *Calculus on Computational Graphs: Backpropagation – Colah's Blog* [online] [visited on 2018-05-06]. Available from: http://colah.github.io/posts/2015-08-Backprop/.
20. ABADI, Martin et al. TensorFlow: A System for Large-Scale Machine Learning [online]. 2016 [visited on 2018-05-04]. Available from: https://research.google.com/pubs/pub45381.html.
21. *TensorFlow* [online] [visited on 2018-05-06]. Available from: https://www.tensorflow.org/.

22. *Facial Keypoints Detection* [online] [visited on 2018-04-16]. Available from: https://www.kaggle.com/c/facial-keypoints-detection.

23. *Python.Org* [online] [visited on 2018-05-09]. Available from: https://www.python.org/.

24. JONES, Eric; OLIPHANT, Travis; PETERSON, Pearu, et al. SciPy: Open Source Scientific Tools for Python [online]. 2001– [visited on 2018-05-06]. Available from: http://www.scipy.org/.

25. HUNTER, J. D. Matplotlib: A 2D Graphics Environment. *Computing In Science & Engineering*. 2007, vol. 9, no. 3, pp. 90–95. Available from DOI: 10.1109/MCSE.2007.55.

26. MCKINNEY, Wes. Data Structures for Statistical Computing in Python. In: van der WALT, Stéfan; MILLMAN, Jarrod (eds.). *Proceedings of the 9th Python in Science Conference*. 2010, pp. 51–56.

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

28. *Leaky ReLU Activation Function* [online] [visited on 2018-05-05]. Available from: https://www.tensorflow.org/api_docs/python/tf/nn/leaky_relu.

29. *Keras Documentation* [online] [visited on 2018-05-05]. Available from: https://keras.io/.

30. *Advanced Activations Layers - Keras Documentation* [online] [visited on 2018-05-05]. Available from: https://keras.io/layers/advanced-activations/.

31. *Stunning Free Images · Pixabay* [online] [visited on 2018-05-10]. Available from: https://pixabay.com/.

32. VIOLA, Paul; JONES, Michael. Robust Real-Time Object Detection, pp. 25.

33. *OpenCV: Face Detection Using Haar Cascades* [online] [visited on 2018-05-10]. Available from: https://docs.opencv.org/3.4/d7/d8b/tutorial_py_face_detection.html.