Statistical Science for the Life and Behavioral Sciences

Deep Learning and Neural Networks

Assignment 2

Authors:	*Student number:*
Freek van Geffen	s2633256
Justin Kraaijenbrink	s2577984

Instructor: Dr. W.J. Kowalczyk

March 11, 2021

Table of Contents

Task 1 - Learn basics of Keras and Tensorflow

Multilayer Perceptron

Benchmark network For the first task of Assignment 2, we implemented the Multilayer Perceptron described in Chapter 10 of the book of Géron, with both the Fashion MNIST and the MNIST datasets. The neural network consists of one input layer of 784 neurons, two hidden layers with respectively 300 and 100 neurons and a single output layer (Géron, 2019). The output layer contains ten neurons, one for each of the ten categories in both datasets (ten clothing articles for Fashion MNIST and ten digits for MNIST). An overview of the structure of the network with the corresponding number of parameters is presented in Table 1.

Layer type	Output shape	Param #
Flatten	(None, 784)	0
Dense	(None, 300)	235500
Dense	(None, 100)	30100
Dense	(None, 10)	1010
Total parameters: 266 610		
Trainable parameters: 266 620		
Non-trainable params: 0		

Table 1: Model structure of the first MLP that was used to classify Fashion MNIST and MNIST images

To get some initial insights in the performance of this first MLP, we performed the following steps for the Fashion MNIST data as well as the MNIST data:

1. Split the whole set of 70000 images into a training set, a validation set and a test set, with sizes of respectively 55000, 5000 and 10000 images;
2. Build and compile the model, with `sparse_categorical_crossentropy` as loss-function and Stochastic Gradients Descent (SGD) with a learning rate of $\alpha = 0.01$ as optimizer. The performance of the network is measured using common sense accuracy, i.e. the percentage of correctly classified images
3. Train the model with the train data and use the validation data during training to track the performance. We train for 30 epochs
4. Evaluate the training process using graphical representations and evaluate the performance of the network on the test data

Figure 1 presents an overview of the training process, with results for Fashion MNIST on the left-hand side and results for MNIST on the right-hand side. These results are very promising! Within only 30 epochs - which takes around 2 minutes on NVIDIA Corporation 10DE:1F91 GPU - the neural network reaches over 90 percent accuracy on the training data and approximates 90 percent accuracy on the validation data as well. For MNIST, the network performs even better, with more than 99 percent accuracy on the train data and 98 percent accuracy on the validation data. Besides, there is no indication of overfitting, because the validation loss and training loss don't diverge excessively.

The training results appear to be very encouraging, but the most important measure of performance is the accuracy of our neural network on the training set. For Fashion MNIST, this accuracy is with 83.44 percent quite good, but not exorbitantly precise. The performance on MNIST test data is considerably better, with 97.75 percent accuracy.

Now we have implemented and fitted an initial MLP, the next step is to tweak important hyperparameters in order to discover some behavioral properties of our network. We will consecutively discuss

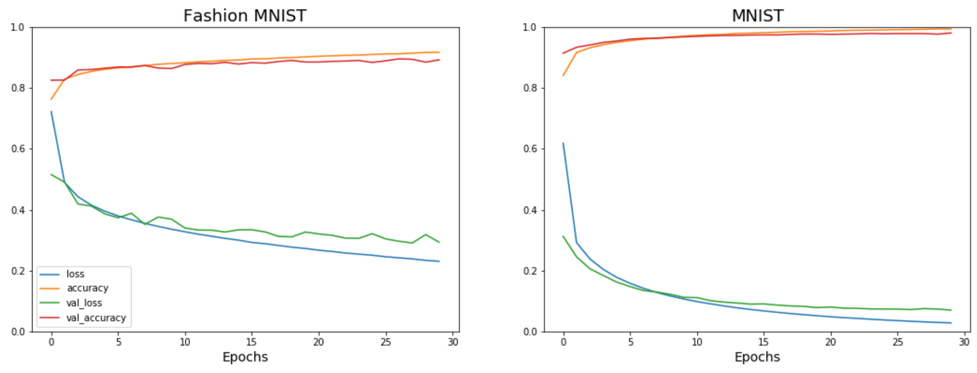

Figure 1: Training results for both Fashion MNIST and MNIST data

different model structures, learning rates, optimizers and activation functions. We conclude this first task by discussing the performance of the network with different dropout strategies. Two important notes:

1. The objective is **not** to obtain an accuracy as high as possible
2. We present the results for Fashion MNIST and will only mention the remarkable results for MNIST if forhtcoming

Tweaking hyperparameters

Model structures To study the effect of model structure of our multilayer perceptron, we designed three additional neural networks. An overview of the model structures is provided in Table 2

Model	# Hidden layers	# Nodes per layer	# Parameters
1 (bench)	2	300, 100	266 610
2	1	1000	795 010
3	6	300, 200, 150, 100, 75, 50	352 835
4	6	150, 150, 150, 150, 150, 150	232 510

Table 2: Model structures of the four different neural networks we have trained

Each model was compiled using the `sparse_categorical_crossentropy` loss-function, Stochastic Gradients Descent (learning rate $\alpha = 0.01$) as optimizer and common sense accuracy as measure of performance. We also timed the training process in order to obtain an idea of the training load. Results are presented in Table 3. It is noteworthy that Model 2 has the shortest training time, while it more than doubles all other networks in number of parameters. We also see that the training time of Model 3 is shorter than the training time of Model 4, but at the same time has one and a half times as much parameters. From this we learn that not the number of parameters is an important feature for training time, but that 'shallowness' is even more important, i.e. the deeper the network, the longer it will take to train the network. Besides, we observe that the accuracy on the test set increases for both deeper networks and shallow networks. Even with our most shallow network there is more than one percent increase in accuracy, which can be attributed to the number of nodes in the network. In fact, neural networks are able to compute any function provided that there are sufficient nodes in the network. This 'universisality' characteristic is shown here.

Learning rates Another very important hyperparameter is the learning rate used for the optimizer function. The learning rate is a measure for how 'fast' the neural network is learning: the bigger the

Model	Training time (s)(30 epochs)	Accuracy (%)
1 (bench)	186.84	83.0
2	175.65	84.1
3	244.37	84.2
4	264.96	83.6

Table 3: Overview of training times and accuracy on the test set for the different model structures

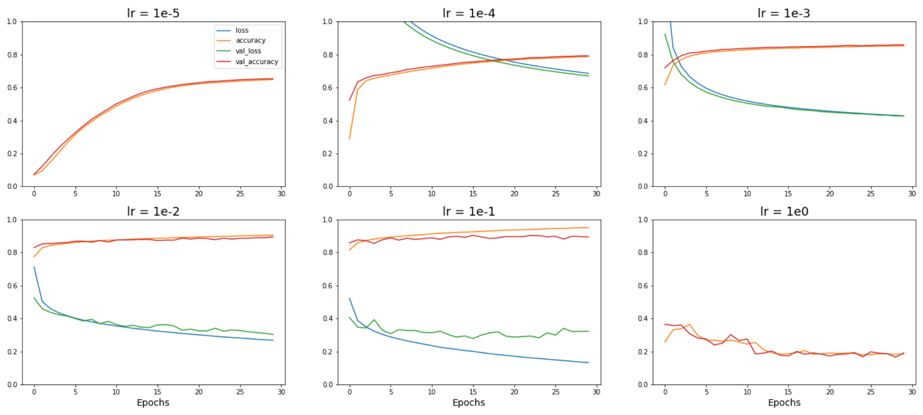

Figure 2: Visualization of the training process for a one hidden layer MLP and SGD with six different learning rates

learning rate, the bigger steps the Stochastic Gradient Descent algorithm will take and the faster the learning process. With a smaller learning rate, the SGD algorithm will take smaller steps, and the learning process slows down. It is therefore quite intuitive that the optimal learning is not too big, but not too small either. To obtain better insight in the influence of the learning rate, we trained the one layer network (1000 hidden nodes, SGD as optimizer) with six different learning rates. Results of the training process are presented in Figure 2.

As we can see, the optimal learning rate is somewhere in between smaller and bigger values. This makes perfect sense, because with a small learning rate it will take the network much more time to converge to a proper solution. For a learning rate of exactly 1, the network is unable to learn and the accuracy on the training data and validation data is only slightly better than you would expect when you let the network perform random guessing. Based on Figure 2, a learning rate somewhere between 0.01 and 0.1 would be most appropriate in order for the network to learn. When evaluating the performance of the networks with different learning rates on the test data, these finding is confirmed, with accuracies of 83.9 and 84.1 percent, respectively. With smaller learning rates, accuracy quickly drops below 80 percent.

Optimizers The next instance to investigate is the use of different optimizers. Up until now we have used Stochastic Gradient Descent, because it performs quite good and is very commonly used. However, there are also disadvantages. Among one of them is that it is not the fastest optimizer and it takes relatively much time to converge to a minimum of the loss function. Another disadvantage of SGD is can be quite noisy, where RMSprop and Adam are for example more stable optimizers. To inquire these properties, we tested SGD, RMSprop and Adam optimizers on our benchmark neural network with 2 hidden layers. Results of the training process are presented in Figure 3. We observe exactly what we expected beforehand: the training process for RMSprop and Adam is slightly more stable with less wobbly tracelines than SGD. Additionally, RMSprop and Adam yield faster convergence and are actually faster ‘overtrained’ which can be derived from the divergence of the blue and green tracelines for loss on the training data and loss on the validation data.

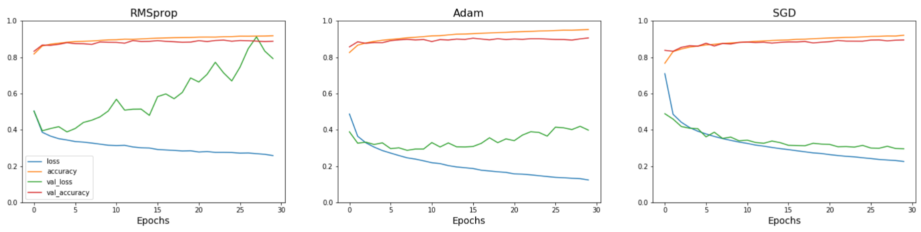

Figure 3: Visualization of the training process for a two hidden layer MLP and different optimizers

Another interesting results if presented in Figure 4, with the accuracies obtained on the test data. When we use the Adam optimizer, we gain over 2.5 percent accuracy compared to SGD. That’s quite a lot for a network that already performs decently!

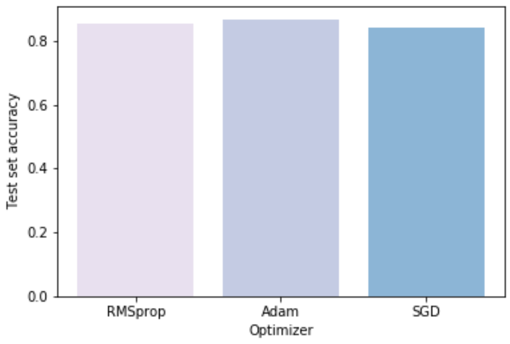

Figure 4: Accuracies (%) obtained for our benchmark MLP with different optimizers

Activation functions A common problem in training neural networks it the phenomenon of vanishing gradients. This problem is encouraged by the use of the sigmoid function, because for both small and large values of the activation of a neuron, the gradient of the sigmoid function grows towards 0. In those regions, learning will take place slowly, which increases training time. The progression of the tanh-function is more spread out and has a wider region with steeper gradients, so learning will take place faster. However, tanh does not deal with the problem of vanishing gradients, because the derivative of the tanh-function also approaches 0 at the tails. One very commonly used activation function that circumvent this problem is RELU activation. We trained a two hidden layer MLP with 300 and 100 hidden nodes respectively, and Adam optimizer. Training times and accuracies on the test data are presented in Table 4.

Activation	Training time (s)(30 epochs)	Accuracy (%)
RELU	210.14	85.3
Tanh	183.99	86.8
Sigmoid	183.49	86.6

Table 4: Overview of training times and accuracy on the test set for different activation functions

These results are somewhat counter-intuitive, because we expected somewhat shorter training times for RELU activation. However, the opposite is true. RELU also performs slightly worse on the test data.

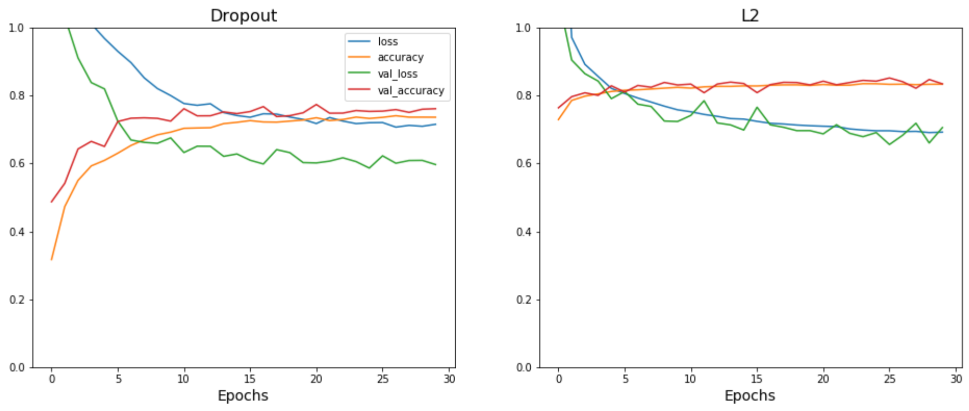

Figure 5: Visualization of the training process for

It seems that tanh is the best choice for this specific configuration of our network, but it is very close to the sigmoid activation.

Dropout The last interesting topic we look at is the use of dropout. Dropout is a common way to prevent the network from overfitting and can also be used to reduce computational costs of the network. To investigate dropout, we compared two approaches. The first one is to just drop 50 percent of the weights after each layer. The second approach is to use l2-regularization, which is a method to penalize bigger weights in the network and therefore in a way 'prevents' the weights in the network from becoming too large. We trained a neural network with six hidden layers, RELU activation and Adam optimization and compared both the dropout method and l2-regularization. Results of the training process are presented in Figure 5, where we can conclude that regularization does not do well to our neural network.

Convolutional Neural Network

Benchmark network For the second part of the first task, a Convolutional Neural Network (CNN) was applied as reference network to the Fashion MNIST and the MNIST datasets. The first convolutional layer of this reference network used a large kernel size (7) and 64 filters. Subsequently, there was a max pooling layer, which divided each spatial dimension by a factor of two. Next, the combination of two convolutional layers and a max pooling layers was repeated two times, while the number of filters grew for each repeat, respectively to 128 and 256. These layers were followed by a layer which flattened the inputs. Next, there was a fully connected network, which contained two hidden dense layers and a dense output layer with respectively 128, 64 and 10 neurons. Two dropout layers with a dropout rate of 50% were added after each of the hidden layers (Géron, 2019). An overview of the structure with the corresponding number of parameters is presented in Table 5.

To investigate this reference model, we performed the same steps as described in the previous section about the Multilayer Perceptron. The online cloud service of Google, Google Colab was used to improve computational power. Nevertheless, the computation time was 5 minutes, 10 seconds per epoch. The model was trained with 30 epochs. The model used a SGD optimizer and RELU activation functions in all its layer, except the output layer. The output layer used a Softmax activation function. All future described models have these parameters settings as default, differences in the parameters settings will be explained explicitly. Figure 6 shows the performance of the network for each of the epochs. The accuracy and validation accuracy are very high, especially for the MNIST dataset. Figure 6 does not show any signs which suggest overfitting. The test accuracy on the fashion MNIST dataset was 88.19%, and the test accuracy of the MNIST dataset was 99.43%, which are very impressive.

Next, we are going to tweak the hyperparameters and compare models with different structures, learning rates, optimizers and activation function with our reference. Again, our main objective is not to

Table 5: The reference convolutional neural network architecture as described in Géron, 2019. Total number of parameters is 1 413 834, 0 are non-trainable.

Layer	Size	Maps	Kernel size	Stride	Padding	Activation	Parameters
Convolution	28 x 28	64	7 x 7	1	same	relu	3200
Max Pooling	14 x 14	64	2 x 2	2	valid	-	-
Convolution	14 x 14	128	3 x 3	1	same	relu	73856
Convolution	14 x 14	128	3 x 3	1	same	relu	147584
Max Pooling	7 x 7	128	2 x 2	2	valid	-	-
Convolution	7 x 7	256	3 x 3	1	same	relu	295168
Convolution	7 x 7	256	3 x 3	1	same	relu	590080
Max Pooling	3 x 3	256	2 x 2	2	valid	-	-
Flatten	2304	-	-	-	-	-	-
Dense	128	-	-	-	-	relu	295040
Dropout (0.50)	128	-	-	-	-	-	-
Dense	64	-	-	-		relu	8256
Dropout (0.50)	64	-	-	-	-	-	-
Dense	10	-	-	-	-	softmax	650

obtain accuracy as high as possible, but to gain insights about tuning networks and their hyperparameters.

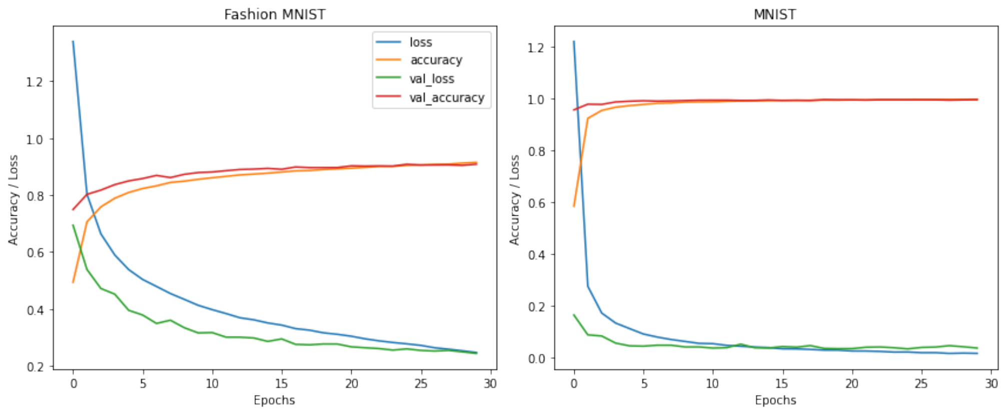

Figure 6: Training results of reference CNN for both Fashion MNIST and MNIST data

Tweaking hyperparameters

Model structures First, we analyzed different structures of the convolutional neural network. We compared our reference model with two model structures, namely LeNet-5 structure and a customized deeper structure of our reference with more layers, which we named the 'Deep model'.

The LeNet-5 architecture is a very famous CNN architecture, created by LeCun et al., 1998. It was developed to classify handwritten digits such as the MNIST dataset. For this architecture, the images are zero-padded to 32 x 32 pixels. The LeNet-5 architecture contained two repetitions of a convolutional layer with a kernel size of 5 and average pooling layer. These repetitions are followed by a convolutional layer and flatten layer. Next, there was a fully connected network, which contained one hidden layer and an output layer. This architecture used valid padding, which causes a decrease in image size through the network. The LeNet-5 architecture used the hyperbolic tangent activation function for each of the convolutional layers and the hidden layer. The output layer contained a softmax activation function. Note that LeNet-5, described in LeCun et al., 1998, used a radial basis function as the activation function.

Nevertheless, Tensorflow has not implemented this activation function and creating a customized activation function is not an objective of this task. The computing time of this model was around 7 seconds per epoch. Table 6 shows detailed information about the number of neurons, number of filters

Table 6: The LeNet-5 architecture. Total number of parameters is 61706, 0 non-trainable parameters.

Layer	Size	Maps	Kernel size	Stride	Padding	Activation	Parameters
Convolution	28 x 28	6	5 x 5	1	valid	tanh	156
Average Pooling	14 x 14	6	2 x 2	2	valid	-	-
Convolution	10 x 10	16	5 x 5	1	valid	tanh	2416
Average Pooling	5 x 5	16	2 x 2	2	valid	-	-
Convolution	1 x 1	120	5 x 5	1	valid	tanh	48120
Flatten	120	-	-	-	-	-	-
Dense	84	-	-	-	-	tanh	10164
Dense	10	-	-	-	-	softmax	850

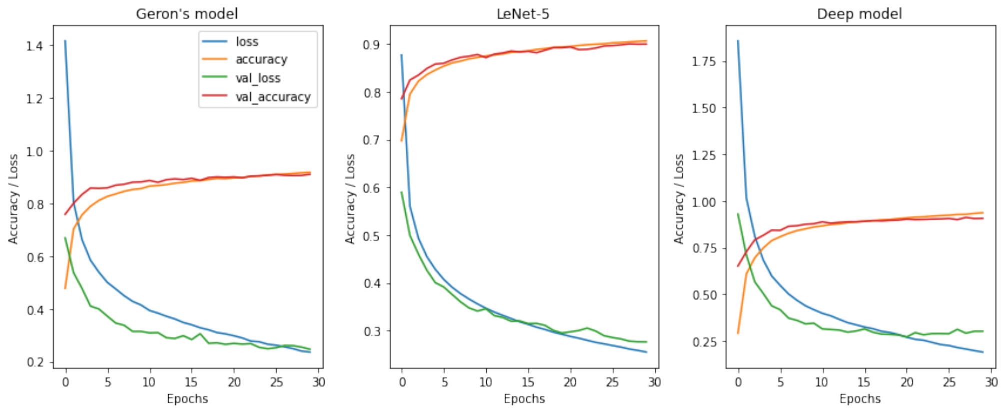

Figure 7: Training results of the different structures of CNN for the Fashion MNIST dataset.

and number of parameters for each layer.

The Deep model was an extension of the reference model which contained an extra repetition of two convolutions layers and a max pooling layer. The extra convolutional layers contained 256 filters. Moreover, the fully connected network contained an extra hidden layer with 256 neurons and an extra dropout layer with a drop rate of 50%. As a consequence of this deeper structure, the computing time of this model was 16 seconds per epoch. Table 14 (Appendix) shows an overview of the Deep model architecture.

Each model was compiled using the `sparse_categorical_crossentropy` loss-function, Stochastic Gradients Descent (learning rate $\alpha = 0.01$) as optimizer and common sense accuracy as measure of performance. Each model was trained for 30 epochs. The results are shown in Table 7. Géron's model had the highest test accuracy for both the Fashion MNIST and the MNIST dataset. LeNet-5 was the worst-performing model for both datasets. Figure 7 show the training process for each epoch for the Fashion data. All the models resulted in convergence and did not show any signs of overfitting.

Table 7: Test accuracy (in %) of the reference model, LeNet-5 and Deep model for the Fashion MNIST dataset and the MNIST dataset.

Model	Fashion	MNIST
Géron's model	87.2	99.4
LeNet-5	82.0	96.4
Deep model	85.9	99.1

Learning rates As discussed in the Multilayer Perceptron section, this is an important hyperparameter. Therefore, we tweaked the learning rate parameter of the optimizer of the reference model. The Stochastic Gradient Descent is used as optimizer, and in the reference model the learning rate of this optimizer was

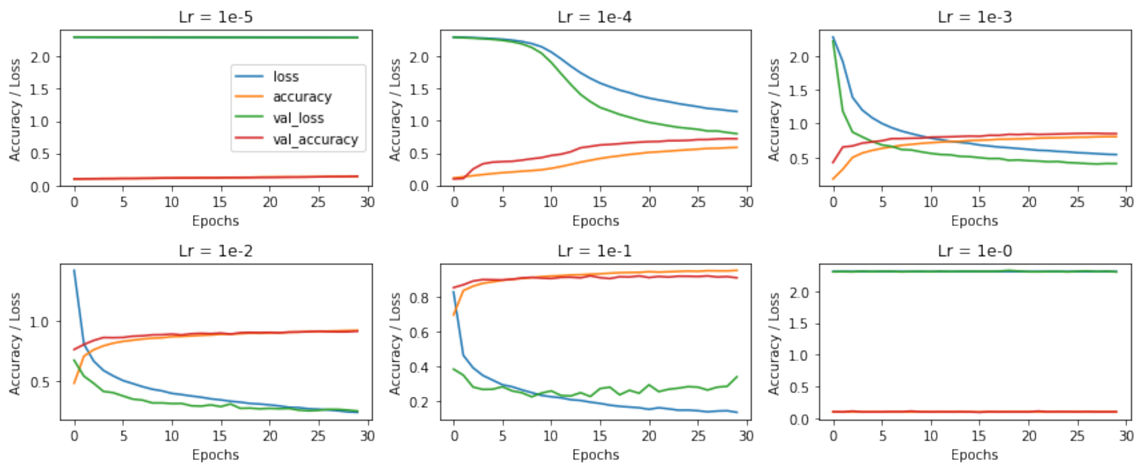

Figure 8: Training results of the different learning rates (lr) applied at the reference model trained with the fashion MNIST dataset.

$\alpha = 0.01$. Table 8 shows that the reference model with learning rate $\alpha = 0.01$ had the highest test accuracy for both datasets. The worst performing learning rates were the biggest and lowest learning rates. Figure 8 shows that the biggest and lowest learning rates did not converge. The model with learning rate $\alpha = 0.0001$ did not converge in 30 epochs but possibly will converge with more epochs. The other models did converge in 30 epochs. Based on the test accuracy table and visualisation of the training results, we can conclude that a learning rate between 0.0001 and 0.01 would be optimal.

α	Fashion	MNIST
0.00001	10.2	19.9
0.0001	69.8	55.9
0.001	80.2	98.5
0.01	87.2	99.4
0.1	83.4	99.4
1	10.0	8.9

Table 8: Test accuracy (in %) for the reference model with different learning rates with Stochastic Gradient Descent as optimizer and trained for 30 epochs with the Fashion MNIST and MNIST datasets. Note that $\alpha = 0.01$ is the reference model.

Optimizers Next, we investigated different optimizers. The reference model used the Stochastic Gradient Descent, and we decided to compare this optimizer with other optimizers. The model with SGD as optimizer had a computation time of 10 seconds per epoch. Using the RMSprop optimizer, the computation time increased to 12 seconds per epoch. Using the Adam optimizer, the computation time per epoch was 11 seconds per epoch. Table 9 shows that the SGD had the highest test accuracy, followed by the Adam optimizer. The RMSprop optimizer did not predict well. The same conclusions can be derived from Figure 9, the RMSprop optimizer did not result in convergence and its (validation) loss eventually increased. Nevertheless, the development of the SGD and the Adam optimizer did result in convergence. We can conclude that the RMSprop optimizer is not a good alternative as optimizer.

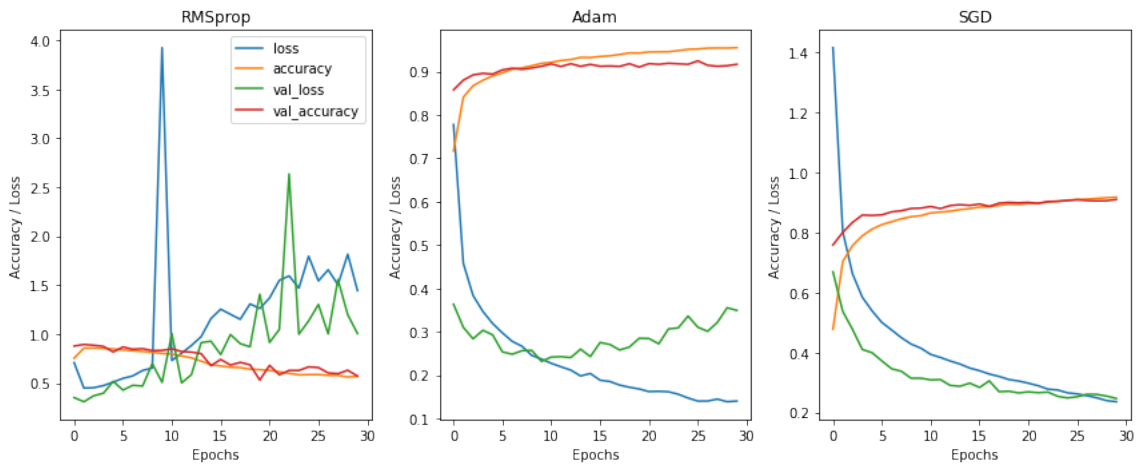

Figure 9: Training results of the different optimizers at the reference model trained with the Fashion MNIST dataset.

Table 9: Test accuracy (in %) for the reference model, using different optimizers for the Fashion MNIST and the MNIST datasets.

Optimizer	Fashion	MNIST
Adam	86.8	98.8
RMSprop	10.0	10.3
SGD	87.2	99.4

Activation functions We have trained our reference model multiple times using different activation functions. As discussed in the previous section, each of the activation function has different advantages and disadvantages. A disadvantage of using the RELU activation function in convolutional layers is that it can get trapped in a dead state, where the output of a neuron is stuck at zero. A possible solution for this problem can be reached by using the SELU activation function. Therefore we compared the performance of the RELU, tanh and the SELU activation functions. We created three models, which contained one of the activation functions in all its layers. Note that in all the model, the output layer had the Softmax activation function. The results are shown in Table 10, the RELU activation function was the best performing activation function, and the SELU activation function was the worst-performing.

We can conclude that using the RELU activation function is the best choice for these datasets. Figure 10 shows the training development for each of the models; we can see that the SELU function did not stabilize during 30 epochs, which is most likely the cause of its low test accuracy. The results suggest that that the RELU activation function is the best choice as the activation function for these datasets.

Table 10: Test accuracy (in %) for the reference model using different activation function in its layers (except the output layer), trained and tested for the Fashion MNIST and MNIST datasets.

Activation	Fashion	MNIST
RELU	87.2	99.4
Tanh	82.3	96.6
SELU	75.3	66.8

Regularization Regularization is a way to prevent overfitting. There are several regularization approaches, such as the L1 regularization and L2 regularization. Dropout is also an approach to regularization in neural networks. Our reference model implied a dropout of 50% twice after each dense layer

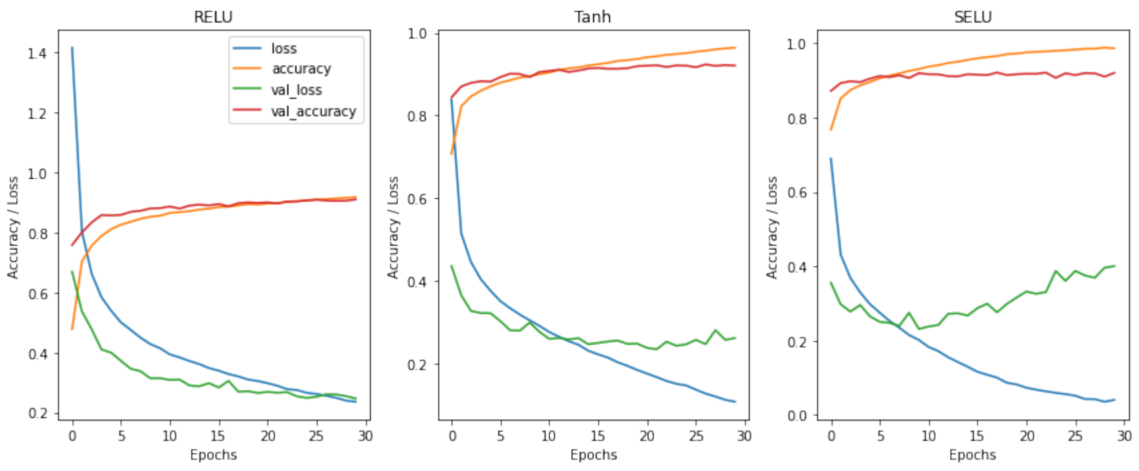

Figure 10: Training results of the reference model using different activation functions in its layers (except the output layer) trained with the Fashion MNIST dataset.

in the fully connected network. We investigated the effect of higher and smaller dropout rates, and we also looked at the effect of L1 and L2 regularization in the convolutional and dense layers.

The results are shown in Table 11. Surprisingly, the model with 0% dropout performs quite good compared to the others model when tested with the Fashion MNIST dataset. The best performing model is the model with a dropout of 50 % and the L2 regularization. Figure 14 (Appendix) shows the training development of each of the models. We did not find any evidence that suggested overfitting and all model reached convergence.

Table 11: Test accuracy (in %) for the reference model using different regularization approaches tested on the Fashion MNIST and MNIST datasets.

Regularization	Fashion	MNIST
Dropout 0%	86.0	99.1
Dropout 25%	87.4	99.4
Dropout 50%	83.2	99.4
Dropout 75%	87.1	99.3
Dropout 50% and L1	84.1	99.2
Dropout 50% and L2	87.7	99.3

Initialization Strategies In order to prevent layer activation outputs from exploding or vanishing, we can make use of initialization strategies. These are specific combinations of an initialization method, such as Glorot normal or He normal, which are usually combined with a specific activation function. Our reference model used the Glorot normal initializer, which is also called Xavier normal initializer, in combination with the RELU activation function. We tried several common weight initialization strategies and compared the results.

Comparing the training development of the models did not result in any essential findings (see Figure 15 in the Appendix). Nevertheless, using the He normal weight initializer and the RELU activation function slightly increased the test accuracy when tested on Fashion MNIST dataset. Table 12 show the test accuracy of the different initializer strategies for the Fashion MNIST and MNIST datasets.

Table 12: Test accuracy (in %) for the reference model using different initializers strategies and activation function, tested on the Fashion MNIST and MNIST datasets.

Initializer strategy	Fashion	MNIST
Glorot normal & RELU	87.2	99.4
Glorot normal & Tanh	83.5	98.1
He normal & RELU	89.8	99.2
Lecun normal & SELU	28.6	66.0

Task 2 - Randomly permuted images

After extensively investigating different settings for our multilayer perceptron and convolutional neural network to classify Fashion MNIST and MNIST images, we will now evaluate the performance of these networks when we randomly permute the order of pixels.

Multilayer Perceptron

The network used for performing the task of randomly permuting images and subsequently training the network on the permuted data, was an MLP with two hidden layers consisting of 256 and 128 neurons respectively. We used the sigmoid activation function and Adam optimization, in combination with the `sparse_categorical_crossentropy` loss-function. To prevent the network from overfitting, we also introduced early stopping with `patience` $= 5$. The input images where flattened to a vector of length $28 \times 28 = 784$ and thereafter randomly shuffled. Note: each image was permuted in exactly the same way. Based on theory, we expect that the neural network yields approximately equivalent performance for the permuted data compared to the non-permuted data. This is because when the order of input neurons is changed, corresponding weights are also changed accordingly. For the multilayer perceptron it should therefore not matter whether we randomly shuffle the image or not. That is quite impressive, since for us humans it would be impossible to recognize such a permuted image. Results of the training process are presented in Figure 11. This process is very similar to the training process of the non-permuted images, just like we expected. Besides, accuracies on the test data are with 87.6 and 97.3 percent for Fashion MNIST and MNIST respectively, quite impressive as well.

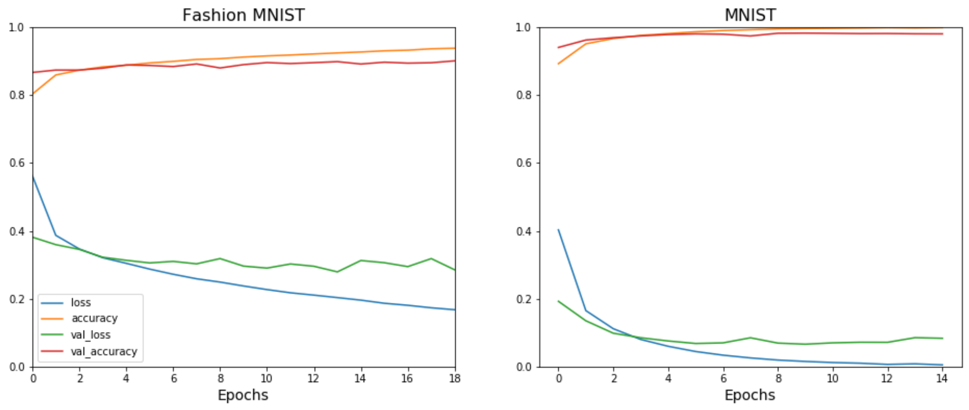

Figure 11: Visualization of the training process for our MLP on permuted Fashion MNIST data (left) and permuted MNIST data (right)

Convolutional Neural Network

For this task, we used Géron's model, compiled it with the Adam optimization and used the `sparse_categorical_crossen`
loss function. Again, we prevented the network from overfitting using early stopping with `patience` $= 5$.

The input images were flattened to a vector of length 28 x 28 = 784 and after that randomly shuffled. Subsequently, we reshaped this vector in an array of 28 x 28. Figure 12 shows an example of an input image, and its randomly shuffled variant.

We expected that the performance of our CNN dropped when switching from raw training images to permutated images and always stated below the accuracy of the MLP (Ivan, 2019).

The test accuracy of the CNN with the Fashion MNIST dataset was 81.1 percent, the test accuracy of the CNN with the MNIST dataset was 91.7 percent. This is in line with our expectations; the CNN performed worse with permutated images than with raw images. Besides that, the CNN performed worse on permutated images than the MLP performed on permutated images.

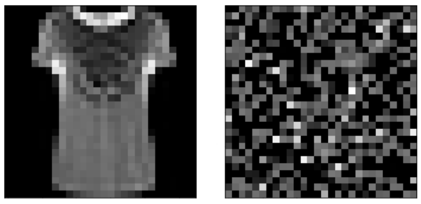

Figure 12: An example of a raw input image of the fashion MNIST dataset (left) and the randomly permuted variant (right).

Task 3 - Developing 'Tell-the-time' Network

For this third task, we exploited the knowledge gained from the previous two tasks to develop a convolutional neural network that correctly tells us the time from a picture of an analog clock. We started with a first naive approach, where we approached this 'tell-the-time' task as a regression problem. After obtaining the first results, we improved the network and tried to obtain as much predictive accuracy and as less prediction error as possible. The findings are presented in the sections below.

First naive approach - regression

The initial idea and approach of this tell-the-time problem was to consider it as a regression problem. The idea is to convert clock times to minutes, were for example a time of 4:21 is converted to $4 \times 60 + 21 = 261$. Before building the network, we performed the following steps:

1. Randomly split the dataset of 18 000 images, size 150 $\times$ 150, into a training set and a test set, following a 80:20 ratio
2. Split of 2400 images from the train set of 14 400 images, which can be used as validation data
3. Convert all the clock time labels from (hour, minute)-format to only minutes
4. Write a function to evaluate common sense accuracy of the network on the test data. It will be very difficult for the network to predict the clock times exactly, which drastically reduces network performance. Even when in fact the network performs fairly well, with for example only a one minute deviance on average, accuracy measured as 'percentage correctly predicted clock times' can be regrettably low when it never predicts the clock times exactly right. Therefore, we did not use percentage correctly predicted clock times as a measure of accuracy, but we wrote a customized function. The function takes the smallest difference between the predicted time and the true time and calculates the mean difference. The use of 'smallest' is on purpose, since the difference between for example 10:00 and 4:30 can be 5:30 or 6:30 (or 330 and 390 when converted to only minutes). This mean difference is then separated into hours and minutes.

Now all the preparatory work has been done, we can build our CNN! A summary of the model structure is presented in Table 13. Based on the results of the previous tasks, we made use of the following hyperparameter settings:

· Activation functions: RELU

· Optimizer: Adam

· Loss-function: MSE

Table 13: Convolutional neural network architecture for our first naive approach. Total number of parameters is 1533951, 600 are non-trainable

Layer	Size	Maps	Kernel size	Strides	Padding	Activation	Parameters
Input	150 x 150	1	-	-	-	-	-
Convolution	73 x 73	50	5 x 5	2	valid	relu	1300
Max Pooling	36 x 36	50	2 x 2	-	valid	-	-
Batch Normalization	36 x 36	50	-	-	-	-	200
Convolution	34 x 34	100	3 x 3	1	valid	relu	45100
Max Pooling	17 x 17	100	2 x 2	-	valid	-	-
Batch Normalization	17 x 17	100	-	-	-	-	400
Convolution	15 x 15	150	3 x 3	1	valid	relu	135150
Max Pooling	17 x 17	150	2 x 2	-	valid	-	-
Batch Normalization	17 x 17	150	-	-	-	-	600
Convolution	5 x 5	200	3 x 3	1	valid	relu	270200
Dropout (0.40)	5 x 5	200	-	-	-	-	-
Flatten	5000	-	-	-	-	-	-
Dense	200	-	-	-	-	relu	1000200
Dense	400	-	-	-	-	relu	80400
Dense	1	-	-	-	-	softmax	401

We trained our network for 50 epochs, where it took approximately 6.5 seconds per epoch on Google Collab GPU. An overview of the progression of the loss is presented in Figure 13. We observe that the loss on the training set is significantly dropping and approaches eventually 0, so that is good news. However, the loss on the validation data is moving up and down, indicating that the predictions are not stable yet. When evaluating the performance of our network on the test data, with our customized function for the accuracy, the results are not that bad after all. On average, our the predictions of our CNN are 36 minutes off. That sounds like a wonderful starting point from which we can improve our network.

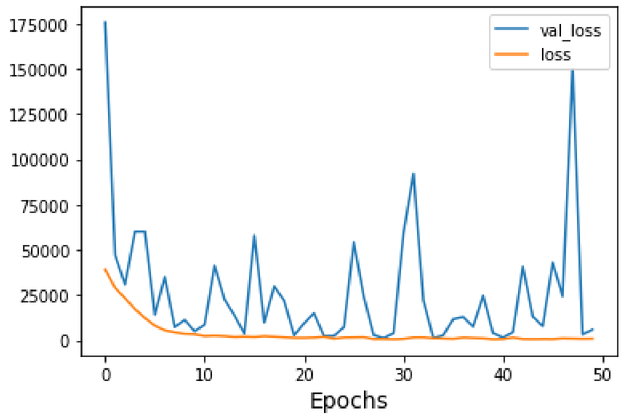

Figure 13: Visualization of the progression of validation loss and loss during the training process

Second naive approach - classification

Our second possible solution for this problem was to consider this task as a classification task with an outcome for each exact time point. This would imply that this is a classification task with 720 (12 * 60) different classes. We applied the same steps as for the naive regression approach and used a comparable structure. However, there were a few things we did differently for this approach. First of all, we did not convert the clock time labels but used the raw clock time labels. Secondly, we used the `sparse_categorical_crossentropy` loss function. We also modified the number of neurons of dense layers such that our output layer ended with 720 neurons. Table 15 (Appendix) shows the architecture for our naive classification approach.

Again, we trained our network for 50 epochs. The computation time was a lot longer compared with the naive regression approach, namely approximately 32.5 seconds per epoch. We observed that the loss dropped to 0.7, but the validated loss fluctuated around 5, which suggests unstable predictions. The common sense accuracy of this model was worse than the naive regression approach; the predictions are 58 minutes off.

Improving the network

Splitted combination approach The naive approaches yielded promising results on the one hand, but there is a lot to improve on as well. With approaching the problem as a regression task, the neural network predicted values between 0 and $(12 \times 60) - 1 = 719$. The spacious amount of values that could be predicted may be one of the reasons why our network does not predict clock times very accurate yet. A better approach may be to consider the hours and minutes separately. For the hours it means that we now only have 12 values to predict, which we will consider as a classification task. For the minutes we now have 60 values, which we will consider a regression task.

The structure of the network is truly comparable to the structure presented in Table 13, but instead of just one dense layer with a single neuron as output, we now have two output neurons. So after the `Flatten`-layer, we now have two dense layers with both 144 neurons and RELU activation. The last dense layer is connected to an output layer with 12 neurons, using softmax activation, in order to classify the hours. We also add two dense layers with respectively 100 and 200 neurons and RELU activation, connected to an output layer with just a single neuron, in order to predict the minutes.

This combination network was trained with the following hyperparameter settings:

· Activation functions: RELU

· Optimizer: Adam

· Loss-function: Categorical crossentropy (hours) and MSE (minutes)

This approach of splitting the tell-the-time task into two separate problems, one for classifying the hours and one for classifying the minutes significantly improved the performance of the network. For example, the percentage correctly classified hours is 87.6 percent on the test data. For minutes, this percentage is much lower (25.8 percent), but that is exactly what we expect since it is much more difficult to correctly predict a number between 0 and 59 than a number between 0 and 11. Even with our customized accuracy function, we are on average much more right. Where we had 36 minutes deviation with our first naive regression approach, we now improved to only 9.7 minutes!.

Conclusion: by splitting up the tell-the-time task into two different tasks, we significantly improved the performance of our network!

References

Géron, Aurelien (2019). *Hands-on machine learning with Scikit-Learn and TensorFlow :* Second edition. Includes QR code. Beijing: O'Reilly,

Ivan, Cristian (2019). "Convolutional Neural Networks on Randomized Data". In: *arXiv preprint arXiv:1907.10935*.

LeCun, Yann et al. (1998). "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86.11, pp. 2278–2324.

Appendix

Table 14: The Deep convolutional neural network architecture. Total number of parameters is 4 822 986, 0 are non-trainable.

Layer	Size	Maps	Kernel size	Stride	Padding	Activation	Parameters
Convolution	28 x 28	64	7 x 7	1	same	relu	3200
Max Pooling	14 x 14	64	2 x 2	2	valid	-	-
Convolution	14 x 14	128	3 x 3	1	same	relu	73856
Convolution	14 x 14	128	3 x 3	1	same	relu	147584
Max Pooling	7 x 7	128	2 x 2	2	valid	-	-
Convolution	7 x 7	256	3 x 3	1	same	relu	295168
Convolution	7 x 7	256	3 x 3	1	same	relu	590080
Max Pooling	3 x 3	256	2 x 2	2	valid	-	-
Convolution	3 x 3	512	3 x 3	1	same	relu	1 180 160
Convolution	3 x 3	512	3 x 3	1	same	relu	2 359 808
Max Pooling	1 x 1	512	2 x 2	2	valid	-	-
Flatten	512	-	-	-	-	-	-
Dense	256	-	-	-	-	relu	131328
Dropout (0.50)	256	-	-	-	-	-	-
Dense	128	-	-	-	-	relu	32896
Dropout (0.50)	128	-	-	-	-	-	-
Dense	64	-	-	-		relu	8256
Dropout (0.50)	64	-	-	-	-	-	-
Dense	10	-	-	-	-	softmax	650

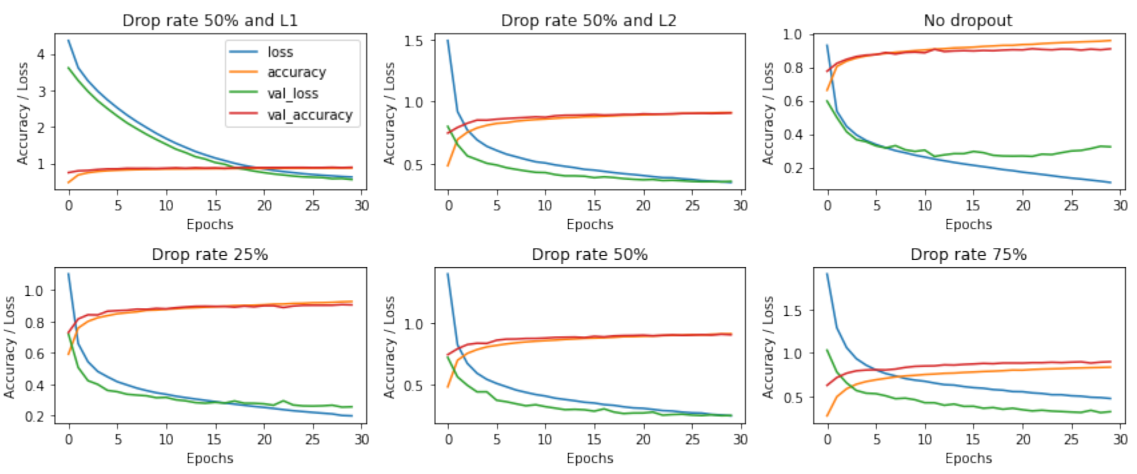

Figure 14: Training results of the different regularization approaches at the reference model trained with the Fashion MNIST dataset.

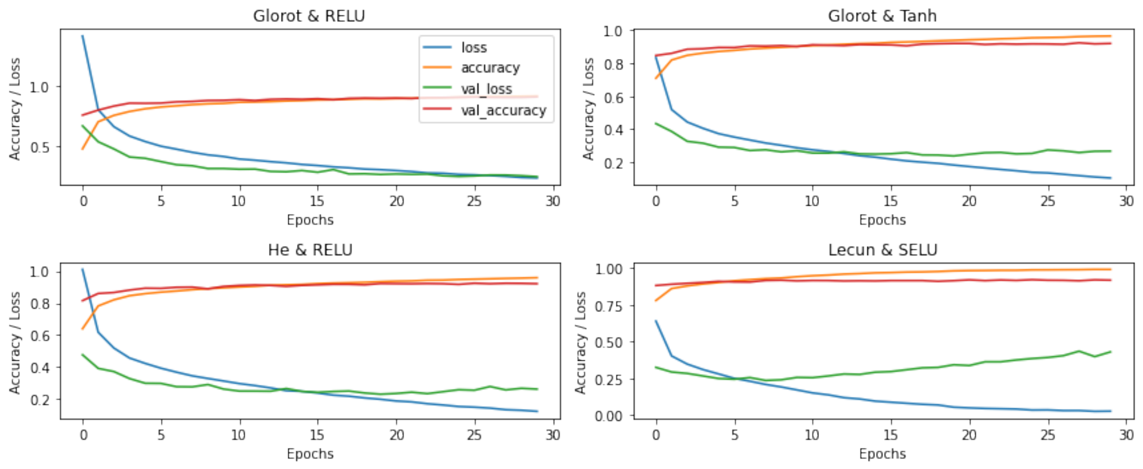

Figure 15: Training results of the reference model using different initializers strategies trained with the Fashion MNIST dataset.

Table 15: Convolutional neural network architecture for our second naive approach. Total number of parameters is 34 012 870, 600 are non-trainable.

Layer	Size	Maps	Kernel size	Strides	Padding	Activation	Parameters
Input	150 x 150	1	-	-	-	-	-
Convolution	73 x 73	50	5 x 5	2	valid	relu	1300
Max Pooling	36 x 36	50	2 x 2	-	valid	-	-
Batch Normalization	36 x 36	50	-	-	-	-	200
Convolution	34 x 34	100	3 x 3	1	valid	relu	45100
Max Pooling	17 x 17	100	2 x 2	-	valid	-	-
Batch Normalization	17 x 17	100	-	-	-	-	400
Convolution	15 x 15	150	3 x 3	1	valid	relu	135150
Max Pooling	17 x 170	150	2 x 2	-	valid	-	-
Batch Normalization	17 x 17	150	-	-	-	-	600
Convolution	5 x 5	200	3 x 3	1	valid	relu	270200
Dropout (0.40)	5 x 5	200	-	-	-	-	-
Flatten	5000	-	-	-	-	-	-
Dense	3600	-	-	-	-	relu	18 003 600
Dense	3600	-	-	-	-	relu	12 963 600
Dense	720	-	-	-	-	softmax	2 592 720