Assignment 2
Introduction to Deep Learning

Peter Breslin
s3228282

Louis Siebenaler
s3211126

November 2021

1 Task 1: Learn the basics of Keras and TensorFlow

This task examined multi-layer perceptrons (MLPs) and convolutional neural networks (CNN) more closely by exploring the framework of these networks and the effect of tuning their features. This was achieved by the use of Keras and TensorFlow, two powerful libraries for deep learning applications. The neural networks developed throughout this assignment employed these frameworks to build, train, and analyze our models, as outlined in our Jupyter Notebook script. To avoid clutter, we have chosen not to include the code relating to our plotting procedures as this involved many lines we believe to be irrelevant to the task.

1.1 Multi-layer Perceptron (MLP)

A MLP network was applied to the MNIST and Fashion MNIST datasets with the aim of exploring the framework by manipulating various network options. A reference MLP network for both datasets was constructed following the same architecture as that given in Ch. 10 pp.296 from Geron's textbook. This was used as a control to compare all modifications of the network to. Adjustments to the model complexity and hyperparameters were investigated, amounting to 10 different variables for consideration: layer weights, dropout rate, optimizer, kernel (weight) regularizer, bias regularizer, loss functions, batch size, layer widths, activation functions, and the depth of the network. Each of these were tuned individually for a range of values and the results of the model after training were then compared against each other, as well as against the reference network. For each training we defined a stopping condition with a patience of 5 epochs based on the validation set loss in order to avoid overfitting. The results for the training histories are plotted in Figure 1. It should be noted that the somewhat sporadic fluctuations to the curves for the MNIST fashion models can likely be explained by the shape of the high-dimensional space where the data is contained, as this feature is exhibited in almost every plot for the fashion dataset. Each hyperparameter and the result of its tuning is explained in the following sections.

Weights: In a neural network, the weights are values multiplied to the neuron inputs to determine how strongly each neuron affects the other within a given layer. This is one of the most important parameters in a network as it establishes the importance of each input value and therefore decides how much influence the input will have on the output. For our investigation, the weights were initialized to zero values, values of ones, and to random values in a normal distribution. The weights for our reference model were set to the default initialization defined by keras. As observed from Figure 1, setting the weights to values of zeros and ones for both datasets resulted in extremely low accuracies on the validation sets. These initializations effectively set each neuron to have the same amount of influence on one another, meaning that each neuron will only learn the same one feature during training. The short training times for these initializations is explained by our stopping condition, indicating that the network stops learning very quickly. The default weight initialization is the Glorot uniform, which essentially is a random uniform distribution. This explains why the accuracy on the validation set with these weights so closely resembles that for the random uniform initialization.

Dropout: When the dropout hyperparameter is applied to a layer it gives the neuron a probability of being temporarily excluded during each training step. This is a value between 0 and 1, where a dropout rate of 0.5 would result in that neuron having a 50% chance of being ignored. This encourages the neurons to activate in a more varied way, helping to reduce the risk of overfitting. While our reference model was initialized with no dropout, dropout rates of 0.2, 0.5, and 0.9 were tested and the results can be seen in Figure 1 for both datasets. Following common practice, these dropouts were applied after the last hidden layer in our model. For both datasets, the highest dropout rate of 0.9 gave the model the lowest accuracy on the validation set, indicating that the model underfit the training set. This was expected as this value is quite high for our model architecture of only 3 layers. Dropout rates of 0, 0.2, and 0.5 resulted in alike accuracies on the validation set. This was also expected for our simple architecture of 3 layers and since the dropout was applied after the last hidden layer only. That being said, models initialized with a dropout rate of 0.2 performed the best on the validation set. It was also noted that the training time increased with increased dropout rate.

Optimizer: Optimizers tweak attributes of the network in order to update the weights of the model such that the loss is minimized. Our reference model was given the stochastic gradient descent (SGD) optimizer, a variant of gradient descent which updates the model parameters more frequently. The optimizers we tested included RMSprop, Adam, and Nadam. These optimizers are similar to each other in that they all use gradient descent but with small differences in how quickly the weights are updated (i.e. the learning rates). As observed in Figure 1, the reference model initialized with SGD was the slowest to converge, taking significantly longer (almost a factor of 3). This was expected as the learning rate is not dynamically updated in this optimization method, meaning that the model will take more time to converge. The models trained with the RMSprop, Adam, and Nadam optimizers all converge quickly and achieve comparable accuracies on the validation sets. These models begin and end with a higher accuracy relative to the reference model due to the dynamic updating of the learning rates, which is more beneficial to use with sparse data like those used in this study. The models with RMSprop were cheapest to train and resulted in the best performances on the validation sets.

Loss Function: The loss function gives us an indication of how good our model is at making predictions by evaluating the error on each training sample and defining which quantities to minimize. The digit MNIST and fashion MNIST models are both multi-classification problems so normal binary-classification and regression algorithms should not be applied to these problems. As the name suggests, binary-classification implies there are only two labels to be predicted, while regression algorithms are used to predict and class continuous values (whereas the MNIST labels are discrete). Therefore, three multi-classification loss functions were applied to our models: categorical cross entropy, Poisson, and Kullback-Leibler (KL) divergence. Our reference model was given the sparse categorical cross entropy loss function. As seen in Figure 1, every loss function except for Poisson performed similarly. The Poisson loss function assumes that the target values originate from a Poisson distribution which is not true for our datasets, which is why our models perform the worst when using this loss function. Although the models initialized with KL divergence achieved marginally better accuracies on the validation sets, these models took longer to train so it was concluded that it would be more practical to use the sparse categorical cross entropy loss function.

Batch Size: The batch size refers to the number of training samples employed in each network propagation. If the dataset is relatively small, large batch sizes can lead to a less precise estimate of the gradients and risks poor generalization of the model on different datasets. However, models with large batch sizes generally trains the network much faster and requires less memory. For our models, we tested batch sizes of 100, 500, and 1000, while our reference model was given a batch size of 32. As seen in Figure 1, each model's accuracy on the validation set increases with decreasing batch sizes. Both datasets consist of 60000 images for training and 10000 for testing, explaining why the model's accuracy with batch sizes of 500 and 1000 were initially much worse than those with 100 and 32. It should be noted that the training time significantly decreased with increasing batch size, but of course at a cost to performance.

Layer Width: The width of a network layer corresponds to the number of neurons in that layer. A width too small could lead to underfitting of the model as there could be too few neurons in each layer to learn the features well. In contrast, a width too large may result in overfitting because there wouldn't be enough information in the training set to train every neuron. Moreover, an increase in layer width also corresponds to a an increase in training time, meaning that a layer too wide could prove impossible to train the network well. Our reference model contained three hidden layers and was given widths of 100 (input layer), 30, 10.
As shown in Figure 1, the accuracy on the validation set for this model performed well. Next, a constant width of 50 was applied to the model. This resulted in a decrease of accuracy on the validation set as there were fewer neurons in the input layer for learning. Layer widths of 10, 30, 100 were then used to highlight the fact that the number of neurons a network will be able to use is dependent on the width of the input layer. Although the layer width (and therefore the number of neurons) increased for each hidden layer, the accuracy on the validation set was not affected by this as there were only 50 neurons to work with from the beginning, at the input layer. These 10 neurons will only be able to influence 10 out of the 30 neurons in the next layer, and 10 out of the 100 neurons in the last hidden layer. Finally, a model was made with widths of 500, 250, 50. This had the best accuracy on the validation set as there was more neurons in each layer for learning and overfitting was not an issue as the datasets are relatively very large. However, it should be noted that this model took much longer to train.

Activation Function: Activation functions helps the model learn more complex features by introducing non-linearities to the network. This is achieved by applying a non-linear function to the inputs of each layer. For the testing of this parameter, we varied the activation function of the output layer while keeping the activation function for the rest of the layers the same for each model. This constant was the ReLU activation function and the activation functions for the output layer tested were softplus, sigmoid, and exponential. Lastly, softmax was used for our reference model. These activation functions were specifically chosen as our models are dealing with multi-classification problems rather than regression. As seen in Figure 1, the models initialized with the softmax, softplus, and sigmoid activation functions have practically the same accuracy on the validation sets. It should be noted that, depending on the problem, the exponential activation function can be very expensive to use due to the exponential nature of the operation included.

Depth: The depth of the network corresponds to the number of layers the model has. Increasing the depth has a similar implication to that of increasing the width of the network. This hyperparameter depends on the complexity of the problem we want the model to solve; too few layers may lead to underfitting while too many could result in overfitting. For testing, we used depths of 1, 10, and 100 layers, while our reference model was initialized with a depth of 3 layers. A width of 10 was set for all layers. As observed in Figure 1, the model with a depth of 100 layers for both datasets resulted in the worst accuracy on the validation sets, and practically failed to learn anything at all. The networks overfit the training sets in this case and cannot generalize to the validation sets. Moreover, these models took relatively long to run due to this large depth,

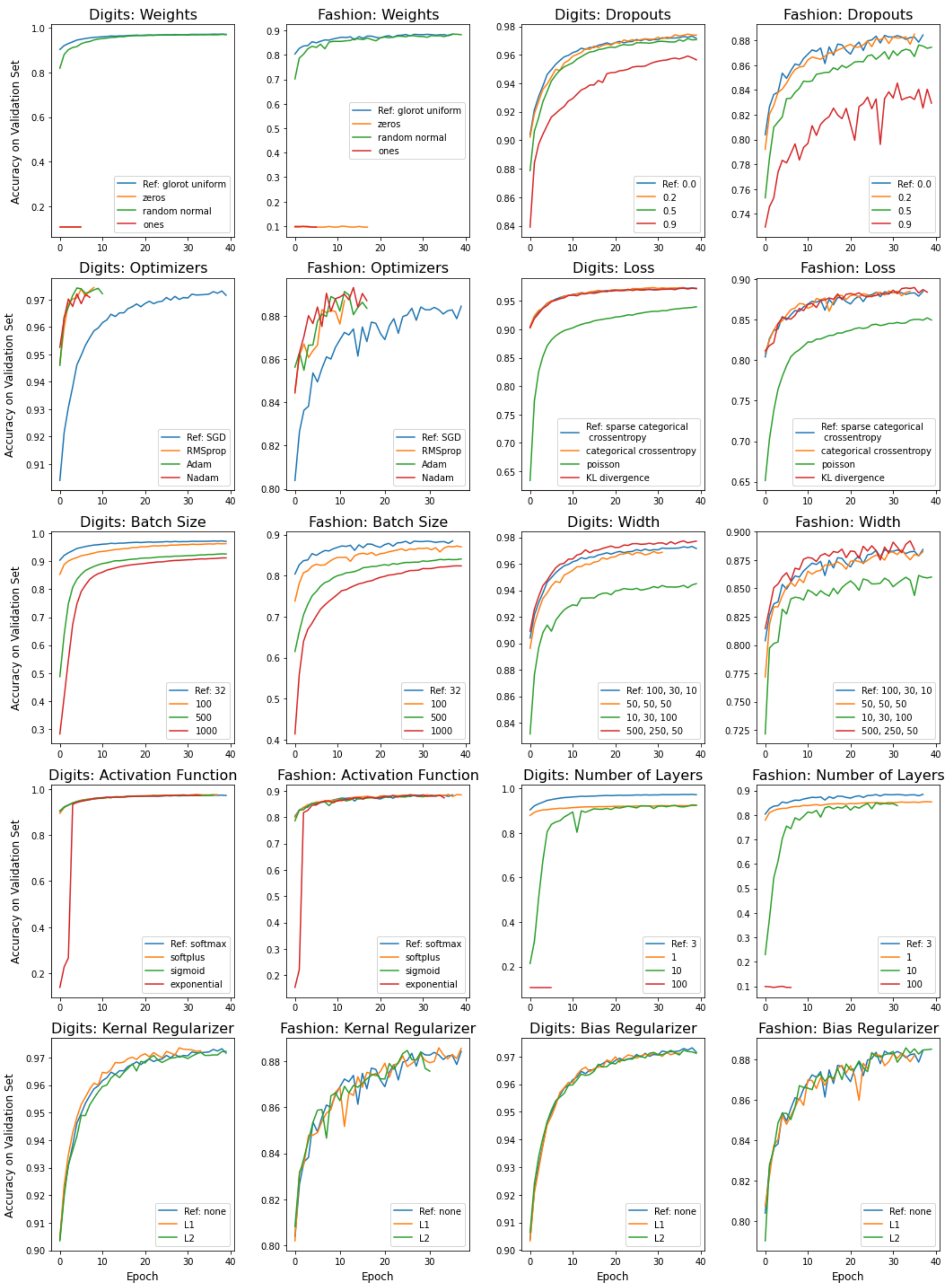

Figure 1: Network parameters for models applied to the MNIST digits and fashion datasets.

even for just the 7 epochs it took to train. The models with depths of 10 layers initially performed quite badly for the same reasons and it was also much more expensive to run than our reference model. The models with a depth of 1 layer perform well but worse than that of our reference model of 3 layers as this model does a better job at avoiding underfitting of the training set.

Kernel and Bias Regularizers: Regularization helps to reduce overfitting by constraining a model to make it simpler. This can be achieved by adding a penalty term to the loss function to reduce the size of the weights such that they are no larger than needed to perform well on the training data. These penalty terms can be obtained using L1 and L2 regularization. L1 regularization adds a penalty equal to the sum of the absolute values of the weights, while the penalty given from L2 regularization is equal to the sum of the squared values of the weights.
The bias is a constant with its own connection weight that is added as an extra input to each neuron. Its role is to help the model fit the training set better by allowing the activation function to be shifted. The kernel (*or weight*) regularizer tries to reduce the weights while excluding the bias term, while the bias regularizer works to reduce the bias. In our reference models, biases are initialized for each neuron but set to zeros as default by keras. However, when testing the bias regularizer the biases were initialized to ones. As seen in Figure 1, the addition of both the L1/L2 kernel and bias regularizers had a very small positive effect on the model performances for the digit dataset. For the case of the fashion dataset, the model's performance did not quite reach that of the reference model when the L1/L2 kernel regularizers were introduced. This network was on its way to overfit the training set if the early stopping criterion had not taken effect. Finally, the introduction of the L1/L2 bias regularizers with this dataset did not improve the model's performance.

1.2 Convolutional Neural Networks (CNN)

The same exercise was repeated to gain practical experience with tuning convolutional neural networks. The reference CNN network which was applied to both datasets has the same architecture as the network introduced in chapter 14 p.461 from Geron's textbook. The hyperparameters of interest for this part include: kernel size, number of filters, stride, padding, pooling and depth of networks. In the following we present the results of tuning each of these hyperparameters individually. The same stopping condition as before was employed during the training. The results for the training histories are illustrated in Figure 2.

Filter size: One of the fundamental properties of CNNs are filters. They are receptive fields which are connected to weights of neurons and scan across the input image in order to detect spatial patterns in the data. This is essential for the purpose of image recognition. It is possible to vary the size of the filter which can be useful to restrict the filter's sensitivity to patterns of certain sizes.
The different filter sizes that were studied are 2, 3, 5, 7 where each layer, apart from the input layer, was defined to have the same kernel size parameter. From Figure 2 it can be recognized that the networks tend to converge faster for larger filter sizes. Furthermore, in general a lower final validation accuracy can be discerned for higher filter sizes. These observations can be understood by considering the scaling between filter size and number of trainable parameters. Larger kernels involve more weights and hence more trainable parameters. This increases the networks probability of overfitting, thus explaining the shorter training in terms of epochs and the reduced accuracy for networks with larger kernel size. However, it should be noted that the true physical

training time increased by a factor of 3 when changing the hyperparameter from 2 to 7. Both the risk of overfitting and increased physical training time suggests that in general smaller kernel sizes are the better option. The differences in accuracy on the training set agreed with the above observations.

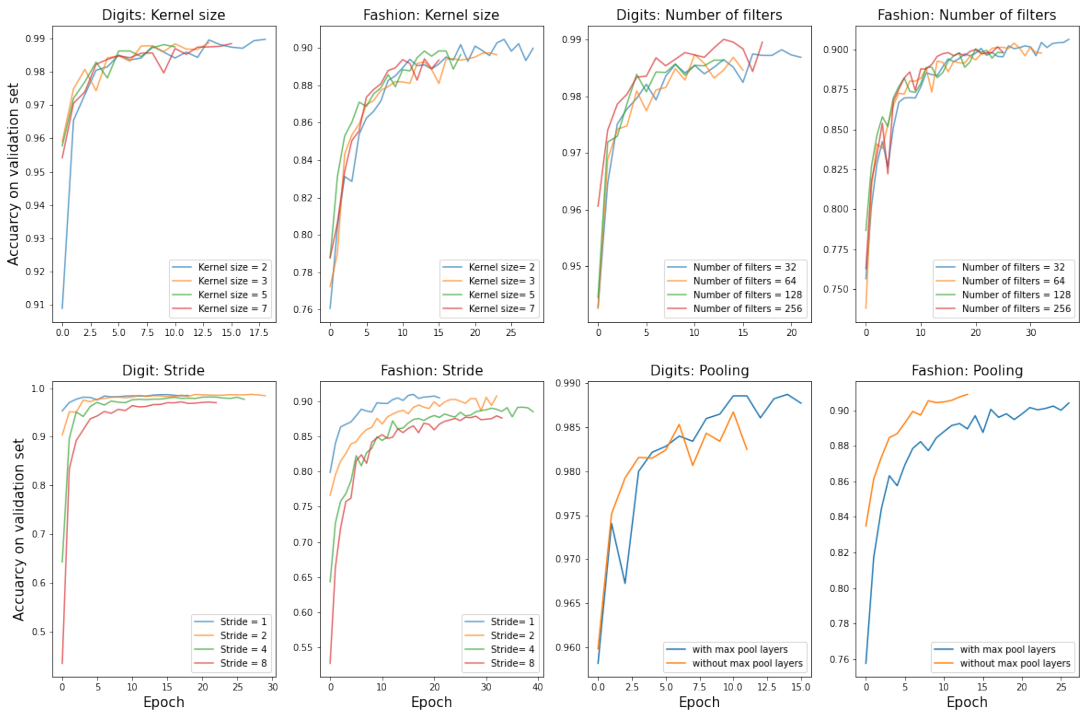

Figure 2: Network parameters for models applied to the MNIST fashion datasets.

Number of filters: The depth of a convolutional layer is characterized by its number of filters. The output of each filter corresponds to a feature map which highlights the areas in an image that activated the filter the most. As a result a layer with multiple filters corresponds to a stacking of feature maps.
The number of filters per layer that were studied are 32, 64, 256 where each layer was defined to have the same number of filters. A similar trend as for the filter size can be discerned in Figure 2 - networks with more filters, which translates to more trainable parameters, exhibit faster overfitting. Additionally, in general the networks with more shallow convolutional layers show a slightly improved performance ($\sim$ 0.5 % higher accuracy) on both the validation and test set.

Stride: The stride parameter of a convolutional layer is defined as the step-size of a filter when convoluting across the input data. Hence, for strides greater than unity, the parameter serves to reduce the dimensionality of the data which can accelerate training and reduce the probability of overfitting. The values of strides that were studied are 1, 2, 4, 8 where again each layer was defined to have the same stride. The resulting training histories are shown in Figure 2. It can be observed that for lower strides (i.e more trainable parameters) overfitting occurs faster. Furthermore, a worse accuracy on both validation and test sets for models with larger strides can be discerned, with a difference of $\sim$ 5% between models with stride = 1 and stride = 8 on the fashion dataset.

This implies that when the stride was 8 the dimension reduction was too large and important patterns in the input could not be recognized by the model. Hence we conclude that while large strides reduce the number of trainable weights, finesse is required when tuning this parameter to ensure an appropriate resolution reduction of the data.

Pooling: Pooling is an operation applied to a convolutional layer in order to reduce the image resolution by means of sub-sampling. The two main pooling techniques are max pooling, where the pooling neuron only transmits the maximum input value in the receptive field, and mean pooling, where the pooling neuron transmits a mean of all the input values in the receptive field. Nowadays, people mostly use max pooling layers as they generally perform better given that they only preserve the strongest features and remove the meaningless ones. Hence, in this analysis we restricted ourselves to the effect of the presence and absence of max pooling layers in the neural network. In Figure 2, we observe that the presence of max pooling layers decreases the tendency of overfitting. Furthermore, from the models on both datasets we recognize that here the presence of pooling layers has no important effect on the validation and test set accuracy (differences are of order 0.4%). Thus the main effect of pooling in this experiment is to reduce the number of trainable parameters, however it should be emphasized that for more complex datasets its presence can dramatically improve the final accuracy.

Padding: Padding is the process of adding layers of zeros ($\equiv$ zero-padding) to the the boundaries of our input image in order to ensure that the kernels cover every pixel in the true input data. We assessed the effect of the presence of zero padding in every convolutional layer. For this simplified data we found no apparent differences in the training histories and final accuracy of the the models.

Depth: We have experimented with a number of different architectures of CNN where we varied the number of convolutional layers and order of pooling and convolutional layers. The main observation is similar to the one off all the hyperparameters above - the more trainable parameters in the network, the faster the model overfits. No significant effect on the network performance related to the order of pooling layers and convolutional layers was observed. Furthermore, we noticed that for these rather simple datasets, the performance of the network does not improve significantly when adding more convolutional layers: a network consisting of one convolutional and max pooling layer had a final accuracy on the validation and test set that was only 0.2% lower than deeper networks consisting of more convolutional layers.

2 Task 2: Data permutations

The most successful MLP and CNN configurations from task 1 were adjusted such that the input data was randomly permuted to shuffle all 28x28 pixels of each training image. This was achieved by using the same random seed for each permutation so that the randomization of each image was implemented in the same way. This was done for both the MNIST and fashion MNIST datasets and the networks were then retrained using the permuted versions of the training sets, the results of which can be observed in Figure 3.

For both datasets, it can be see that the MLP model accuracies on the validation sets are extremely similar to those obtained using the reference model without permutation. MLP models are insensitive to the order of the elements of the input vectors, meaning that MLPs are inherently permutation insensitive. Pixel randomization should therefore not effect the classification

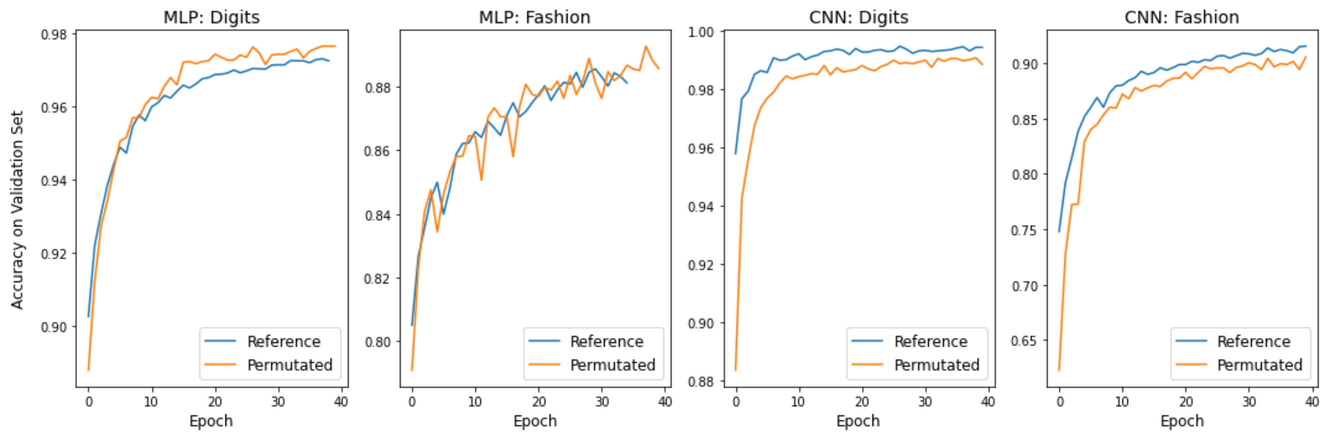

Figure 3: MNIST and fashion MNIST datasets applied to our MLP and CNN models for permuted input data.

accuracy of our MLP, as can be seen from our results. For the case of the CNNs, the classification accuracies of both datasets are shown to be lower for the models with permuted input data compared to the reference CNN model without permutation to its data. CNNs are used for circumstances in which the input data is known to be a hierarchical structure based on locally correlated elements. Therefore, if the image pixels are randomly permuted, the hierarchical structure of the input data is altered and long range correlations are introduced which our CNN cannot interpret. In other words, although CNNs are spatially invariant to the image (e.g. they are not sensitive to a change in the image position), they do not maintain this spatial relationship among features as the local spatial structure is modified when the data is permuted. This is why we see a degraded classification performance for both datasets when the input data of our CNN models are randomized. When the same permutation is applied to all images as we have done, some underlying structure remains that can still be identified by the network. This explains why the classification performances are not meaningless and are close to that of our reference model.

3 Task 3: A "Tell-the-time" network

We are provided with a large dataset of analogue clock images along with labels corresponding to the time they display. We developed a number of different CNN's whose task is to read the time as accurately as possible. In the following we describe 4 different models for this particular problem. For each model we processed the input images by normalizing each pixel value and a 80:20% ratio for the training and test sets was adapted for each network.

3.1 Regression model

One approach to read the time of an analogue clock is to implement a CNN which acts as a regression model. In this, the labels are represented in the following format: [03:00 $\rightarrow y = 3.0$], [05:30 $\rightarrow y = 5.5$]. An immediate problem which can be recognized by this label representation is that the standard mean-square-error (mse) loss function for regression problems will not compute the common sense difference between the predicted and target time. For example, if the network predicts a value of 11 but the true value is 1, mse returns a value of 100 while the logical error should be 4. As a result we created a custom loss function which ensures that the common sense time difference is evaluated. Furthermore, we realized that our neural network does not learn

using this representation of the labels and hence decided to normalize them. The architecture along with the training hyperparameters are found in the Jupyter notebook for task 3.
The training of the model is illustrated in Figure 4. The accuracy on both the training and validation set remained poor throughout the training and we decided not to consider this metric when analysing the performance of the model. Hence, Figure 4 shows the mse only. The decrease in the mse implies that the model is learning and at the end of the training it reaches a mse of order 10^{-2} on the validation set. When evaluating this model on the test set we find an average logical time difference between the prediction and true time of $\sim$ 19 minutes. This is already surprising low given the simple representation of the time labels. The experiment was repeated a couple times using different random shuffling of the data for the training-test set splitting and it was found that the final performance of the network depends on the data included in the training set. Using this architecture and label representation our final average logical time error on the test set varied between 19 to 25 minutes. It can be noted that these time differences are below 1 hour implying that the models have learned to interpret the hour reading very well.

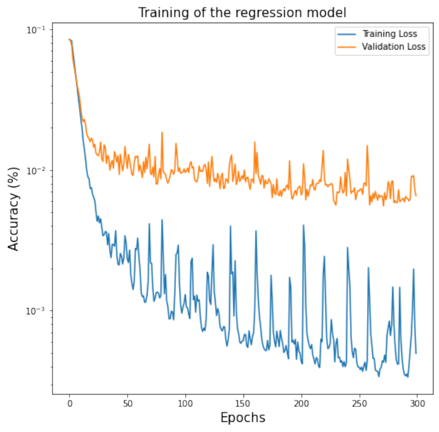

Figure 4: Training history of the regression model

3.2 Classification

It is possible to treat this problem as an n-classification problem, where each time reading is assigned to a corresponding category. We begin by defining 24 classes where each covers a 30 minute interval in order to determine an appropriate architecture for this problem. At the end we used the same architecture as in the previous part with the exception that the output layer is defined by 24 nodes and the Softmax activation function. Moreover, the sparse categorical cross entropy loss function was employed. Using this architecture it was possible to achieve a validation accuracy of $\sim 85\%$ after 100 epochs. Next we increased the number of categories to 720, such that each minute after 0:00 is assigned to an individual class and applied the same network architecture to the new labels. The training history is shown in Figure 5. The plot shows that the network was able to learn the training data, however the validation accuracy remained low during the entire training. This implies that the network immediately overfitted and started to learn the training data by heart, hence its inability to generalize to the validation set. This observation suggests that treating the time reading task as a 720-class classification problem is too excessive.

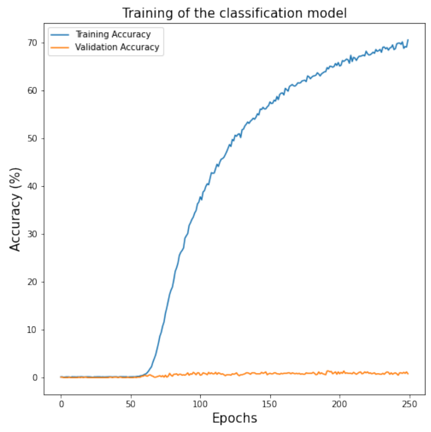

Figure 5: Training history of the classification model

3.3 Multi-head model

An alternative approach is to develop a neural network consisting of two output branches. One branch acts as a classifier for predicting hours and the other serves as a regression model for predicting minutes. For this model we used the same base of convolution layers as for the previous models with the heads consisting of 2 and 4 dense layers of 200 neurons for the hour and minute branch respectively. The architecture of the model can be found in the code snippet in the appendix. Both the hour and minute labels were normalized for this model. A part of the training history of the networks is shown in Figure 6. It can be observed that the model is capable of learning both the hour and minute clock hand. When assessing its abilities on the test set it exhibits an average logic time difference of ~ 5 minutes, which is a good improvement to the previous models. The reason why it performs better than the simple regression model is that the network attempts to learn the minute and hour clock hand separately rather than treating them as a single more complex entity. Figure A.8 in the appendix shows the error distribution on the test set which clearly reflects the good performance of the network.

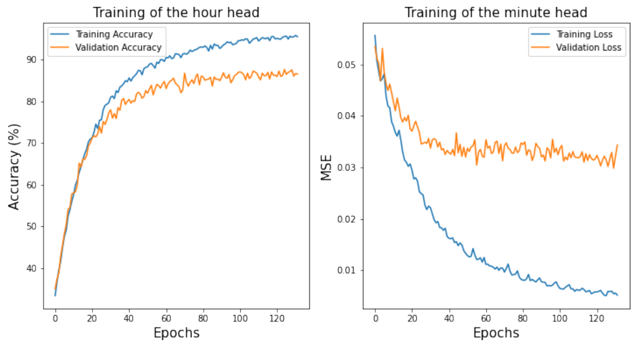

Figure 6: Training history of the multi-head model

3.4 Periodic function model

It is possible to express the time labels y_h and y_m using periodic function. This is achieved by:

$$Y_h = \cos\left(\frac{\pi}{12} y_h\right) \tag{1}$$

$$Y_m = \cos\left(\frac{\pi}{60} y_m\right) \tag{2}$$

, where Y_h and Y_m are the new representations of the labels. One advantage of this method is that the targets are defined between -1 and 1, which for neural networks is often the preferred representation of data. Using this reformulation, the goal of the network is to model a periodic function. This can only be achieved if at least one of the network's layers has a periodic activation function. Hence, we implemented a customized activation function $\cos x$, where x is the output from the previous layer. In our model we applied this activation function to the output layer. The architecture of our final model is very similar to the previous ones and can be found in the Jupyter notebook. The training history is shown in Figure 7. We can observe that the validation accuracy reaches very high values, up to 95%, hence suggesting that the model generalizes very well. Assessing the model on the test set further proves its strong generalization ability as the average logical time difference is ~ 6 minutes. Figure A.9 shows the error distribution on the test set and we recognized that this is localized around 0. Interestingly, a small bump around 60 minutes can be discerned. On further analysis (see Figure A.10) where we consider only the predictions which exhibit an error within [55, 65] minutes, we find that our network has difficulties recognizing 12 o'clock and confuses it with 1 o'clock. Nevertheless, overall the performance of this network is very good and we can conclude that both the multi-head model and periodic label model are the best choices for learning how to read the time.
One way to further improve our results is to ensure that there are no biases in our training set. Hence, creating a training set consisting of a uniform distribution of time labels could be a potential continuation of this problem.

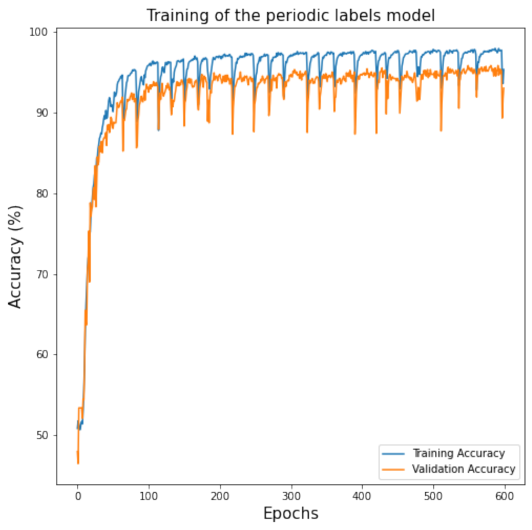

Figure 7: Training history of the periodic function model

A Appendix

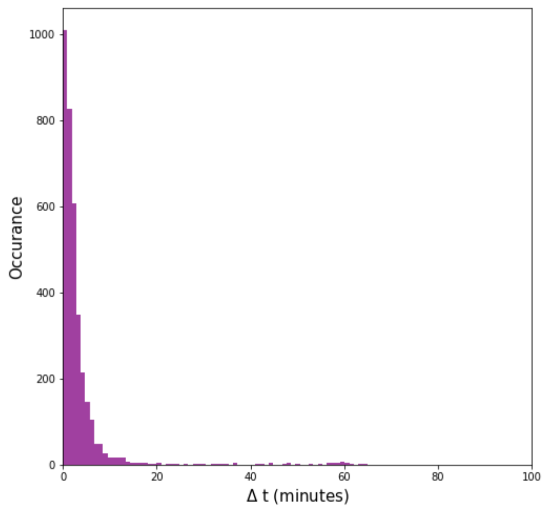

Figure A.8: Logical time difference distribution on the test set by the multi-head model

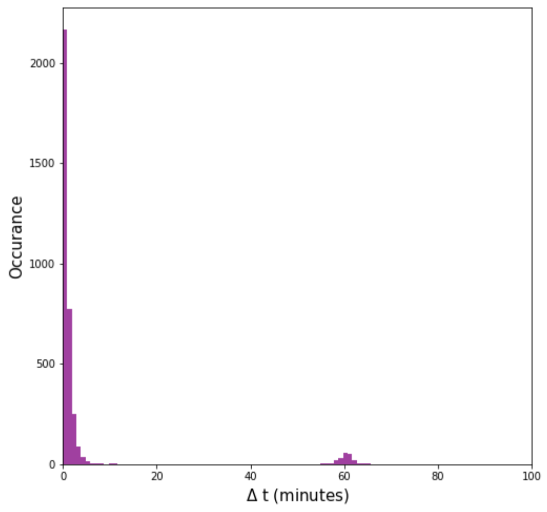

Figure A.9: Logical time difference distribution on the test set by the periodic function model

Multi-head model architecture

```
#define the architecture of the multi-head model

#base of the model
inp = layers.Input(shape=(150, 150, 1))
model = layers.Convolution2D(32, kernel_size = (3, 3), activation = 'relu')(inp
    )
model = layers.MaxPooling2D(pool_size=2)(model)
model = layers.Convolution2D(32, kernel_size = (3, 3), activation = 'relu')(
    model)
model = layers.Convolution2D(32, kernel_size = (3, 3), activation = 'relu')(
    model)
model = layers.MaxPooling2D(pool_size=2)(model)
model = layers.Convolution2D(64, kernel_size = (3, 3), activation = 'relu')(
    model)
model = layers.Convolution2D(64, kernel_size = (1, 1), activation = 'relu')(
    model)
model = layers.Flatten()(model)

#hour branch --> acts as classification model
hour = layers.Dense(200, activation = 'relu')(model)
hour = layers.Dropout(0.1)(hour)
hour = layers.Dense(200, activation = 'relu')(hour)
hour = layers.Dense(12, activation = 'softmax', name = 'hour')(hour)

#minutes branch --> acts as regression model
minute = layers.Dense(200, activation = 'relu')(model)
minute = layers.Dense(200, activation = 'relu')(minute)
minute = layers.Dense(200, activation = 'relu')(minute)
minute = layers.Dropout(0.1)(minute)
minute = layers.Dense(200, activation = 'relu')(minutel)
minute = layers.Dense(1, activation = 'softplus', name = 'minute')(minute)

#put all of the parts together
model = tensorflow.keras.Model(inputs = inp, outputs=[hour, minute])
```

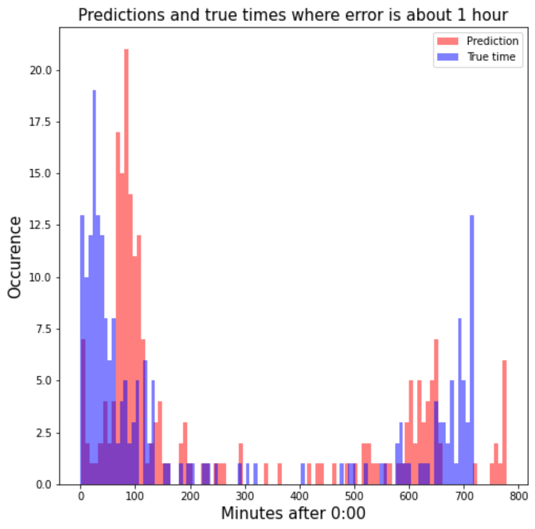

Figure A.10: Illustrated are the predictions for which the time error is within [55, 65] minutes. Shown in red are the predictions and blue are the true times. It can be seen that most of the 1 hour error predictions occur for 12 o'clock. The network confuses these often as 1 o'clock.