Introduction to Deep Learning
Assignment 2

Irina-Mona Epure, Ramya T Rameshchandra

1 Task 1: Learning the Basics of Keras and TensorFlow

During the first task of this assignment, we got acquainted with two very powerful tools for creating and using neural networks: TensorFlow and Keras. TensorFlow is a very popular open-source platform containing a vast array of machine learning tools and libraries. Keras, on the other hand, is a Python API designed for facilitating the use of TensorFlow for deep learning. To gain some experience with the implementation of deep learning models, we experimented with various configurations and architectures of **Multilayer Perceptrons (MLPs)** and **Convolutional Neural Networks (CNNs)**. We used the well-known datasets MNIST and Fashion MNIST to train and test our networks on and we recorded all of our findings in the two subsections below.

1.1 Multilayer Perceptrons

MLPs are formed of one input layer, at least one layer of threshold logic units (TLUs), called a hidden layer, and one output layer. To start our experiments, we used the reference network proposed by [1] for classifying the samples in the Fashion MNIST dataset. This network contains one Flatten layer that converts each input into a one-dimensional array, and two hidden layers composed of 300 and 100 TLUs, respectively. The hidden layers use the RELU activation function. The output layer comprises 10 TLUs because this is the number of possible labels in each of the datasets we considered. Originally, this model was compiled using the SGD algorithm.

We decided to experiment with different MLP configurations and architectures by starting from this reference network and either making a small change to its structure, or changing a hyperparameter value. The details of the performances resulting from each of this small changes are included in Appendix A. To measure performance, we used simple classification accuracy, and sparse categorical cross-entropy as loss. We split each of the datasets into a training set of size 55000, a validation set of size 5000, and a test set of size 10000. We trained each network for only one epoch before analysing its performance on the test set, but this is enough to give us a good indication of the usefulness of each configuration. Our conclusions after running these experiments on MLPs are the following:

1. Classification on Fashion MNIST benefits from the use of a Random Normal kernel

and bias initializer.

2. SELU is the best activation function to be used in the hidden layers when working with both datasets.

3. L2 regularization offers slightly better results for Fashion MNIST.

4. NAdam is the best optimization algorithm in both situations.

5. In both cases, the accuracy is higher if multiple steps are performed per execution. For MNIST, the best value was of 10 steps, while for Fashion MNIST, it was of 20.

6. Performance can be improved by adding more layers: for Fashion MNIST, just one more layer of size 200 between the two existing ones, while for MNIST, also one layer before them (of size 600) and another one after (of size 50).

These findings are going to be very useful in Task 2, where we are going to combine the best hyperparameter values found in order to obtain more powerful models for working with each of the chosen datasets.

1.2 Convolutional Neural Networks

Convolutional Neural Networks are a type of Artificial Neural Network, inspired by the brain's visual cortex, that use convolution in place of general matrix multiplication in at least one of their layers. In addition to the layers discussed in the section above (MLP), CNNs include two additional building blocks: convolutional layers and, pooling layers. A neurons weights can be represented as a small image the size of the receptive field, called filters (convolution kernels). A convolution layer has multiple filters and outputs one feature map per filter. This leads to quite a few hyperparameters: number of filters, the stride, and the padding type. A pooling neuron has no weights; all it does is aggregate the inputs using an aggregation function such as mean or max. Pooling layer reduces computation, memory usage and the number of parameters.
We chose the CNN proposed in [1] (Chapter 14, Page 447) to test out our experiments. The first step of the experimentation was preprocessing MNIST and Fashion MNIST data to be fed into the CNN architecture. We normalized the input values by dividing them by 225; the independent variable matrices (X_train, X_test, X_valid) were reshaped to (number_of_sample,28,28,1) to add a single channel input. The dependent variable matrices (y_train, y_test, y_valid) were one-hot encoded for the 10 categories in each dataset. We are using 55000 samples for training, 5000 samples for validation and 10000 samples for testing.
The experiments on the CNN were divided into two part: the hyperparameters involved in training the network and, changing the network structure. The hyperparameters obtained in the first part were used for the latter experiment. The following types of experiments were carried out with the results given in Appendix B:

1. Optimizer and Learning rate: The network was trained with 4 different optimzers - SGD, Adagrad, Adam and RMSprop with 3 combination of learning rates - 0.001, 0.01, 0.1. The best accuracy for both MNIST and Fashion MNIST was obtained for RMSprop with leanrning rate 0.001.

2. The next hyperparamter experimentation was for the batch size, epochs combination. The different values tested for batch size was 64, 128 and 256 with the values for epochs was 5, 10. The best accuracy was obtained for both the datasets on 10 epochs and batch size of 256.

3. The next experimentation was changing the kernel_initializer and bias_initializer. The different combinations of randomNormal, Zeros and glorotUniform were used for both kernel and bias initializations for both datasets. The default initializers worked best and the these weight initializers were not used in future experiments.

4. The next experiment was tried by changing the activation function of both convolutional and dense layers. All the previous experiments used relu activation function and the network was tested with tanh and selu activations. 'relu' performed best and was retained for the next experiments.

5. The next experimentation was done by reducing the filter sizes for convolution layers and number of units in dense layers. The size of the filters in the first convolution layer was changed to 16, the next two to 64 and the last two to 128. The number of units in the first dense layer was reduced to 64 and number in the last dense layer was changed to 16. This network performed well in comparison to the original but the original was marginally better.

6. The next experimentation was done by increasing the filter sizes for convolution layers and number of units in dense layers.Te size of the filter in the first convolution layer was changed to 128, the next two to 256 and the last two to 512. The number of units in the first dense layer was reduced to 256 and number in the last dense layer was changed to 128. This network performed well in comparison to the original but the original was marginally better.

7. In this experiment we changed the dropout rate for the original network. The textbook [1] states that dropout rate of 40-50% is commonly used for convolution networks, so we experimented by changing the dropout rate to 0.4%. The reduced dropout rate improved the performance of the network for MNIST data while there was not improvement for the fashion MNIST.

8. In this experiment, we changed all three pooling layers from Max pooling to Mean pooling (keras.layers.AveragePooling2D(pool_size=2)). Max pooling performed better and was retained for future experiments.

9. In this experiment we changed the network by reducing the number of layers. The last 3 layers before the dense layers were removed (2 convoluted and 1 pooling). The performance was comparable to the original.

10. In this experiment we increased the number of layers by adding 3 more before the first dense layer (added 2 convoluted and 1 pooling layer). The performance of this network was worse than the original.

2 Task 2: Testing the Impact of Obfuscating Data by Randomly Permuting All Pixels

As part of our second task, we observed the effect of data obfuscation on the test set performance of our best MLP and CNN configurations. First of all, we chose our highest performing network architectures to be used in this experiment. The selected networks are the following:

- **MLP for MNIST**: no initializer, SELU activation function for the hidden layers, no regularization, NAdam optimization, 10 steps per execution, and an additional layer between the existing layers in the reference network.
- **MLP for Fashion MNIST**: random normal initialization, SELU activation, L2 regularization, NAdam optimization, 20 steps per execution, and 3 additional layers.
- **CNN for MNIST**: RMSprop optimizer with learning rate of 0.001, 10 epochs, 256 batch_size, default kernel and bias initializers, 'relu' activation, Max pooling layers and, dropout rate of 0.4 on the original CNN
- **CNN for Fashion MNIST**: RMSprop optimizer with learning rate of 0.001, 10 epochs, 256 batch_size, default kernel and bias initializers, 'relu' activation, Max pooling layers and, dropout rate of 0.5 on the original CNN

To obfuscate the datasets, we applied a simple random permutation to the pixels in each sample. Figure 1 shows what one sample image looks like before and after the permutation.

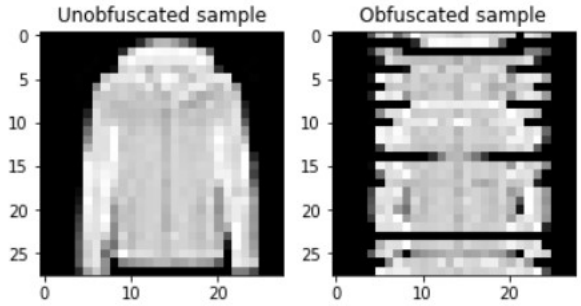

Figure 1: A sample image from the Fashion MNIST dataset, before and after obfuscation.

We then fit and evaluated our best MLPs and CNNs on both the original and the obfuscated datasets. The observed performances are recorded in Table 1. We noticed that for MLPs

the loss and performance values on the obfuscated datasets are very close to those obtained on the original ones. The performance of the CNN was slightly worse on the obfuscated datasets than on the originals. This shows that the CNN is dependent on the similarity between objects in a class. This structure is lost on permuting the data, which results in the CNN performing worse.

Table 1: Performance of best network configurations on original and obfuscated datasets.

Network	**Dataset**	**Loss**	**Accuracy**
MLP for MNIST	MNIST	0.156	0.951
	Obfuscated MNIST	0.127	0.962
MLP for Fashion MNIST	Fashion MNIST	1.023	0.789
	Obfuscated Fashion MNIST	1.361	0.778
CNN for MNIST	MNIST	7.644	0.994
	Obfuscated MNIST	23.211	0.983
CNN for Fashion MNIST	Fashion MNIST	65.731	0.878
	Obfuscated Fashion MNIST	40.377	0.846

3 Task 3: Developing a "Tell-the-Time" Network

For this final task, we tried various methods through which to create a CNN that can tell the time from the image of a clock. We had at our disposal a dataset of 180000 images of the same clock showing different hours, with different rotations and sometimes reflections. The input images have a dimension of 150x150 pixels, while the labels are arrays of two values, of the form [hour, minute]. Below are the four different strategies we experimented with.

Moreover, traditional loss functions and accuracy metrics don't measure the performance of neural networks on a time prediction problem in a relevant way. Therefore, we introduced some functions that help us interpret the precision of our networks. For example, the difference between [11, 55], and [0, 5] is not 11 hours and 50 minutes, but just 10 minutes. We implemented these losses by considering the "shortest distance around the clock" between the true and predicted time.

3.1 Regression Problem

Telling the time can be changed into a regression problem by transforming the labels in one of two different ways:

- calculating the number of minutes after 12 o'clock (e.g. [3, 30] becomes 210).

- calculating the number of hours as float values (e.g. [3, 30] becomes 3.5).

We used a CNN similar to the one described in Task 2, but with the last dense layer of size one, as it should return a single numerical value. However, of the transformations mentioned produced a good performance. The average "common sense" error was around 180 minutes.

3.2 Multi-Class Classification Problem

We then tried to solve this problem with classification. We started out with a reduced number of classes. We grouped the labels into bins covering a half of an hour each day: from [0, 0] to [0, 30], from [0, 31] to [1, 0], and so on. This left us with only 24 classes, so we used a final dense layer of size 24 with SoftMax activation. In this case, the common sense loss was measured as the number of halves of an hour away from the true class. At the end of training, the average loss was around 5, so, once again, roughly 180 minutes.

3.3 Multi-Head Model

An interesting approach was creating a network with two different outputs: a classification based on the hour, and a regression based on the minutes. To implement this, we had to make use of Keras' Functional API instead of the Sequential one we had been using before. We used two different types of loss: categorical cross-entropy for the classification head, and mean squared error (MSE) for the regression head. We noticed the regression component was converging much slower than the classification one, so we spent some time developing each of them separately.

The network has one input layer which receives batches of matrices of size 150x150. This is followed by 5 convolutional layers of increasing sizes, with alternating max pooling layers of size 2. The resulting tensors are converted to one-dimensional vectors using a flatten layer. From this point on, the two network heads start to differ. For the classification, there is one dense layer and one output layer with 12 different classes, one for each hour on the clock. For the regression component, there are two dense layers and the output layer which returns one float value for the minutes. Using this method, we achieved a common sense loss of almost 180 minutes as well.

3.4 Periodic functions

In this task, we attempted to train a multi-head CNN for predicting time, with one head predicting the sine and the other predicting the cos value of the input label. As a first step the input label was converted into sin and cos values. These were fed into the multi-head regressor with two regression heads. The inverse of the obtained tan(sin/cos) was obtained

to measure out loss.
The model trains with a sine_loss of 0.274 and cos_loss of 0.267 on the training data.
The above decoding function used, was not successful in obtaining the actual time predicted and hence the common sense error.

References

(1) Géron, A., *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow: Concepts, tools, and techniques to build intelligent systems*; O'Reilly Media: 2019.

A Results of Experiments with MLPs

Table 2: Results of experimenting with MLP layer and compilation hyperparameters (part 1).

Hyperparameter	Dataset	Initializer	Loss	Accuracy
Kernel and bias Initializer	MNIST	**None**	**0.311**	**0.913**
		Random Normal	0.323	0.909
		Random Uniform	0.353	0.900
		Truncated Normal	0.331	0.905
		Zeros	2.301	0.113
		Ones	2.301	0.113
	Fashion MNIST	None	0.558	0.802
		Random Normal	**0.547**	**0.812**
		Random Uniform	0.612	0.787
		Truncated Normal	0.566	0.799
		Zeros	2.302	0.100
		Ones	2.302	0.100
Hidden layer activation function	MNIST	RELU	0.315	0.909
		Sigmoid	1.944	0.618
		TanH	0.330	0.907
		SELU	**0.294**	**0.915**
		ELU	0.322	0.908
		Exponential	nan	0.097
	Fashion MNIST	RELU	0.543	0.811
		Sigmoid	1.512	0.602
		TanH	0.527	0.813
		SELU	**0.494**	**0.822**
		ELU	0.529	0.813
		Exponential	nan	0.100
Kernel, bias and activity regularizer	MNIST	**None**	**0.313**	**0.912**
		L1	2.341	0.217
		L2	3.733	0.909
	Fashion MNIST	None	0.549	0.802
		L1	1.901	0.547
		L2	**3.896**	**0.805**

Table 3: Results of experimenting with MLP layer and compilation hyperparameters (part 2).

Hyperparameter	Dataset	Initializer	Loss	Accuracy
Optimizer	MNIST	AdaDelta	2.116	0.271
		AdaGrad	0.594	0.866
		Adam	0.116	0.961
		AdaMax	0.167	0.951
		FTRL	2.302	0.113
		NAdam	**0.104**	**0.967**
		RMSProp	0.105	0.966
		SGD	0.312	0.910
	Fashion MNIST	AdaDelta	1.804	0.522
		AdaGrad	0.741	0.749
		Adam	0.444	0.839
		AdaMax	0.444	0.842
		FTRL	2.302	0.100
		NAdam	**0.441**	**0.847**
		RMSProp	0.430	0.846
		SGD	0.567	0.798
Steps per execution	MNIST	None	0.321	0.907
		2	0.310	0.911
		5	0.302	0.914
		10	**0.304**	**0.916**
		20	0.313	0.911
		50	0.308	0.914
		100	0.304	0.915
	Fashion MNIST	None	0.534	0.815
		2	0.534	0.809
		5	0.541	0.812
		10	0.534	0.810
		20	**0.528**	**0.819**
		50	0.554	0.800
		100	0.536	0.814

B Result of Experiment with CNNs

Table 4: Results of experimenting with MLP architectures and Dropout regularization.

Architecture	**Dataset**	**Loss**	**Accuracy**
Reference	MNIST	0.305	0.916
	Fashion MNIST	0.554	0.796
Reference with Dropout	MNIST	0.384	0.895
	Fashion MNIST	0.638	0.775
Decreased layer size	MNIST	0.575	0.802
	Fashion MNIST	0.321	0.905
First layer removed	MNIST	0.548	0.809
	Fashion MNIST	0.360	0.903
Second layer removed	MNIST	0.568	0.808
	Fashion MNIST	0.371	0.901
Extra layer before	MNIST	0.613	0.783
	Fashion MNIST	0.279	0.919
Extra layer after	MNIST	0.521	0.818
	Fashion MNIST	0.282	0.917
Extra layer inbetween	MNIST	0.629	0.771
	Fashion MNIST	**0.273**	**0.923**
All 3 extra layers	**MNIST**	**0.251**	**0.927**
	Fashion MNIST	0.560	0.796
All 3 extra layers with Dropout	MNIST	1.477	0.466
	Fashion MNIST	1.048	0.597

Table 5: CNN - Results of experimenting with hyperparameters (Part-1)

width=

Hyperparameter	**Dataset**	**Hyperparamter value**	**Accuracy**
Optimizer and Learning rate	MNIST	SGD(l_r=0.001)	**0.343**
		SGD(l_r=0.01)	0.951
		SGD(l_r=0.1)	0.984
		Adagrad(l_r=0.001)	0.654
		Adagrad(l_r=0.01)	0.975
		Adagrad(l_r=0.1)	0.978
		Adam(l_r=0.001)	0.986
		Adam(l_r=0.01)	0.091
		Adam(l_r=0.1)	0.097
		RMSprop(l_r=0.001)	0.981
		RMSprop(l_r=0.01)	0.988
		RMSprop(l_r=0.1)	0.981
	Fashion MNIST	SGD(l_r=0.001)	0.350
		SGD(l_r=0.01)	0.675
		SGD(l_r=0.1)	0.834
		Adagrad(l_r=0.001)	0.648
		Adagrad(l_r=0.01)	0.774
		Adagrad(l_r=0.1)	0.804
		Adam(l_r=0.001)	0.100
		Adam(l_r=0.01)	0.121
		Adam(l_r=0.1)	0.100
		RMSprop(l_r=0.001)	0.810
		RMSprop(l_r=0.01)	0.844
		RMSprop(l_r=0.1)	0.855
Batch size and Epochs	MNIST	BS=64 Ep=5	0.598
		BS=64 Ep=10	0.113
		BS=128 Ep=5	0.992
		BS=128 Ep=10	0.203
		BS=256 Ep=5	0.991
		BS=256 Ep=10	0.993
	Fashion MNIST	BS=64 Ep=5	0.509
		BS=64 Ep=10	0.107
		BS=128 Ep=5	0.868
		BS=128 Ep=10	0.770
		BS=256 Ep=5	0.859
		BS=256 Ep=10	0.901

Table 6: CNN - Results of experimenting with hyperparameters (Part-2)

width=

Hyperparameter	**Dataset**	**Hyperparamter value**	**Accuracy**
Kernel, Bias initializers	MNIST	RandomNormal,RandomNormal	0.113
		RandomNormal,Zeros	0.113
		RandomNormal,GlorotUniform	0.113
		Zeros,RandomNormal	0.113
		Zeros,Zeros	0.113
		Zeros,GlorotUniform	0.862
		GlorotUniform,RandomUniform	0.108
		GlorotUniform,Zeros	0.992
		GlorotUniform,GlorotUniform	0.972
Kernel, Bias initializers	Fashion MNIST	RandomNormal,RandomNormal	0.100
		RandomNormal,Zeros	0.100
		RandomNormal,GlorotUniform	0.100
		Zeros,RandomNormal	0.100
		Zeros,Zeros	0.100
		Zeros,GlorotUniform	0.754
		GlorotUniform,RandomUniform	0.102
		GlorotUniform,Zeros	0.892
		GlorotUniform,GlorotUniform	0.559

Table 7: CNN - Results of experimenting with network structure and parameters

Test	**Dataset**	**Test value**	**Accuracy**
Activation Function	MNIST	tanh	0.955
		selu	0.726
Activation Function	Fashion MNIST	tanh	0.871
		selu	0.812
Reducing filter size	MNIST		0.993
	Fashion MNIST		0.872
Increasing filter size	MNIST		0.986
	Fashion MNIST		0.877
Dropout rate 0.4	MNIST		0.994
	Fashion MNIST		0.882
Average Pooling	MNIST		0.613
	Fashion MNIST		0.100
Reducing number of layers	MNIST		0.616
	Fashion MNIST		0.101
Increasing number of layers	MNIST		0.047
	Fashion MNIST		0.100