Introduction to Deep learning
Assignment 2

Yuang Yuan
3115631

Wei Chen
3156486

1 Learn the basics of Keras and TensorFlow

In this task, we apply two neural networks (MLP and CNN) to two famous images data sets (MNIST and fashion MNIST) with different parameters and architectures. The main goal of the task is to find how and explain why the changes in parameters and architecture will influence the network and the experimental results.

1.1 MLP

An MLP is composed of one input layer, one or more layers of hidden layers, and one output layer. MLPs can be used for classification tasks like classifying the images in the MNIST. In this situation, we need 10 output neurons (or more for the fashion MNIST) using the logistic activation function: the output will be a number between 0 and 1, which we can interpret as the estimated probability of the positive class.

1.1.1 Vanilla MLP

We implement a classification MLP (from the textbook) that is structured as Figure 1. The parameters and architectures are left unchanged. We train the MLP During 3 times fittings, the training accuracy is higher than the validation accuracy after approximating 24 epochs which means the model is over-fitting then. In the following experiments, we will train all MLPs 3 times in 30 epoch and all changes are made based on this vanilla version. The MLP gives a (highest) 97.33% accuracy, and the loss and accuracy are shown in Figure 4 and 5.

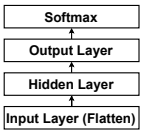

Figure 1: Vanilla MLP Structure

1.1.2 Initialization Strategies

- The appropriate initialization values

We initialize the hidden weights with normal (the default value of `tf.keras.initializers.RandomNormal`), slightly smaller and larger normal distributed values. As Figure 1 shows, the one with small initialization values learned very slow while the one with large initialization values had an even worse performance. Therefore, initializing weights with inappropriate values will lead to divergence or a slow-down in the training of your neural network.

- The appropriate initialization distributions

Initializing using normal distribution with fixed standard deviation may lead to very large or small activation values

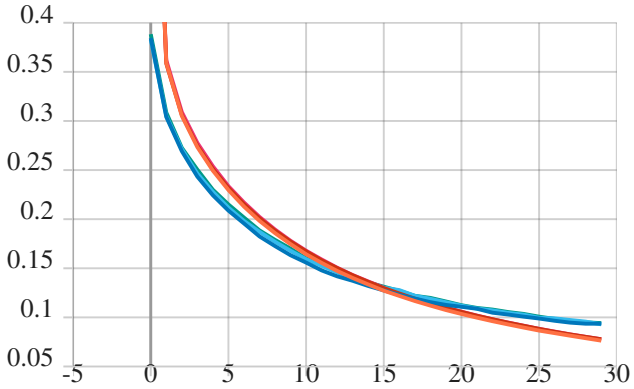

Figure 2: Vanilla MLP: Loss (The orange curves are train losses. The blue curves are validation losses.)

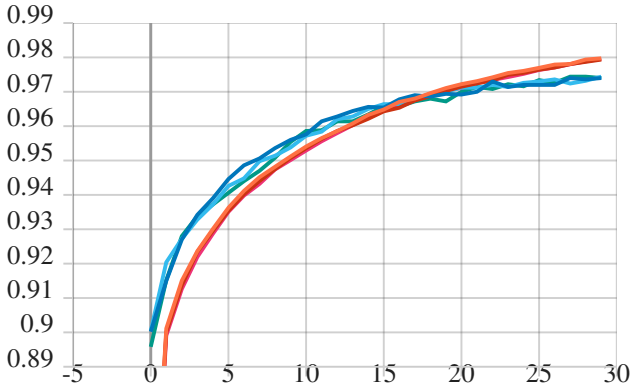

Figure 3: Vanilla MLP: Accuracy (The orange curves are train accuracies. The blue curves are validation accuracies.)

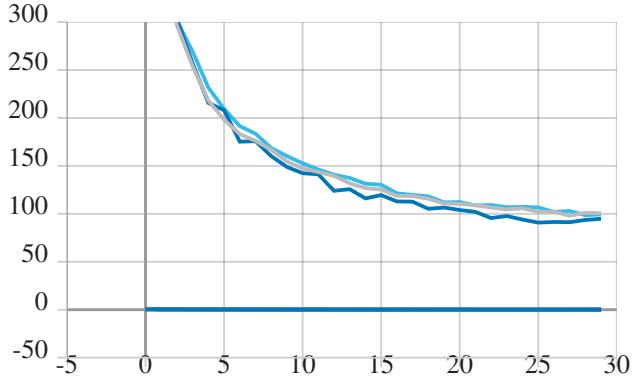

Figure 4: Initialization Values: Loss (The curves on the top are losses with small and large initialization values. The curves in the bottom are losses with normal initialization values.)

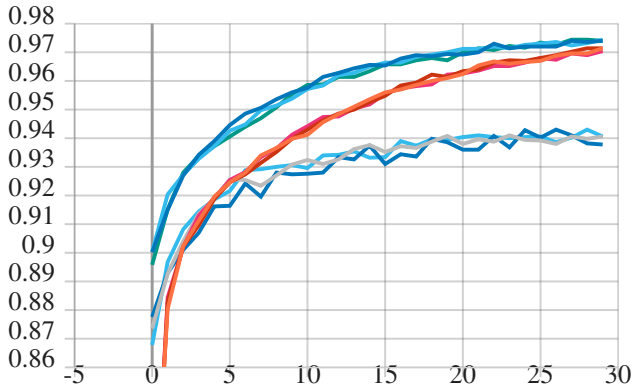

Figure 5: Initialization Values: Accuracy (The curves on the top are normal initialized accuracies. The red and orange curves are accuracies with small initialization values. The blue and grey curves in the bottom are accuracies with large initialization values.)

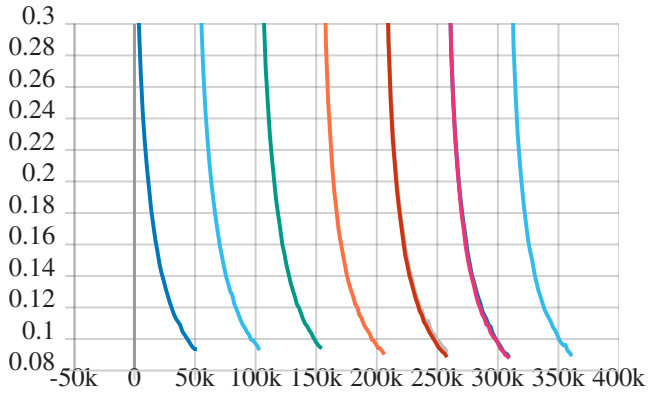

Figure 6: Initialization Distributions: Loss vs Iterations(The three curves in the left are losses with normal initialization. The orange, brown and pink curves in the middle are losses with He initialization. The curves in grey, blue and light blue are losses with Xavier initialization.)

since the variance of weights is not taken care of. Xavier initialization solve this problem by keeping variance of all the layers equal. Kaiming He initialization takes activation function into account. It is suitable for neural network with ReLu activation functions. We initialize the hidden weights with normal distributionXavier and He distribution. The results are shown in Figure 6 and 7.

Apparently, choosing appropriate strategies is crucial to the final training result. In this situation, it is better to initial hidden weights in the He distribution with suitable values.

1.1.3 Activation Functions

We will use the softmax activation function in the output layer because the classes are exclusive. And we try to apply ReLu, sigmoid and tanh functions in the hidden layers. As Figure 8 and 9 shows, networks with ReLu function score the best. Besides, ReLu function consumes fewer computation resources and time during training since it only returns the value provided as input directly or constant 0 while sigmoid and tanh function require exponential calculations. We don't observe great differences this time, but careful consideration should be made especially when we are training a very large network.

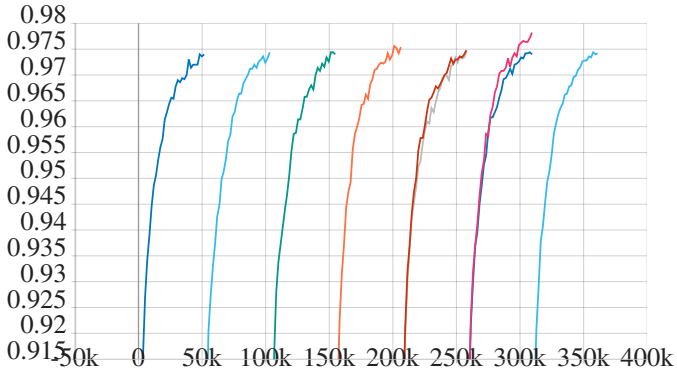

Figure 7: Initialization Distributions: Accuracy vs Iterations(The three curves in the left are accuracies with normal initialization. The orange, brown and pink curves in the middle are accuracies with He initialization. The curves in grey, blue and light blue are accuracies with Xavier initialization.)

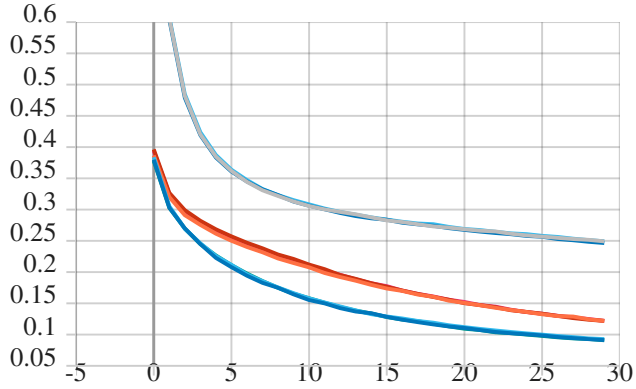

Figure 8: Activation Functions: Loss (The grey curves on the top are losses with sigmoid function. The middle curves are losses with tanh function. The blue curves in the bottom are losses with ReLu function.)

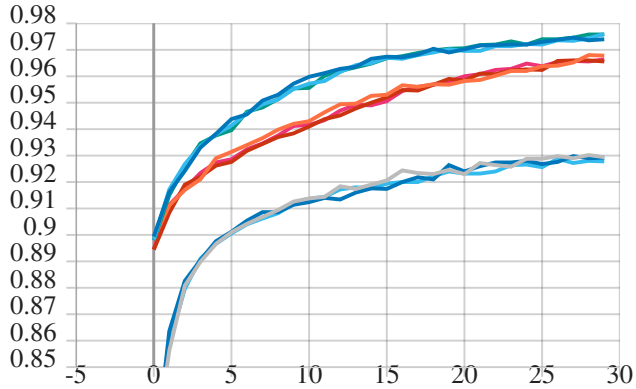

Figure 9: Activation Functions: Accuracy (The blue curves on the top are accuracies with ReLu function. The middle curves are accuracies with tanh function. The blue curves in the bottom are loss with sigmoid function.)

1.1.4 Optimizers

SGD has trouble navigating ravines, i.e. areas where the surface curves much more steeply in one dimension than in another, which are common around local optima. Nesterov accelerated gradient is a way to give our momentum term a kind of prescience. It prevents us from going too fast and results in increased responsiveness. By modifying the

hyperparameters (momentum and nesterov), we can accelerate SGD in the relevant direction and dampens oscillations. We observe a very fast-fitting with `momentum=0.9, nesterov=True` as Figure 10 and 11 shows.

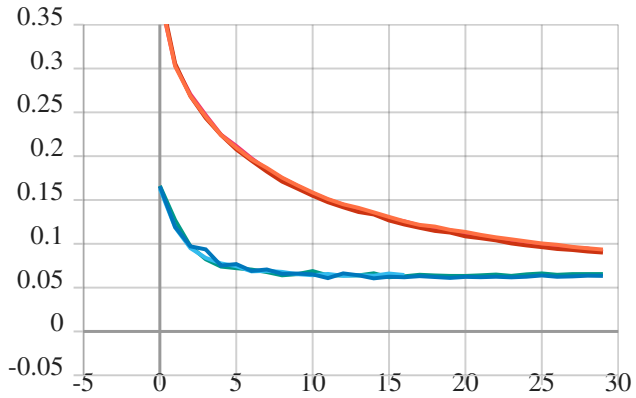

Figure 10: SGD: Loss (The orange curves are losses with SGD. The orange curves are losses with Nesterov accelerated gradient)

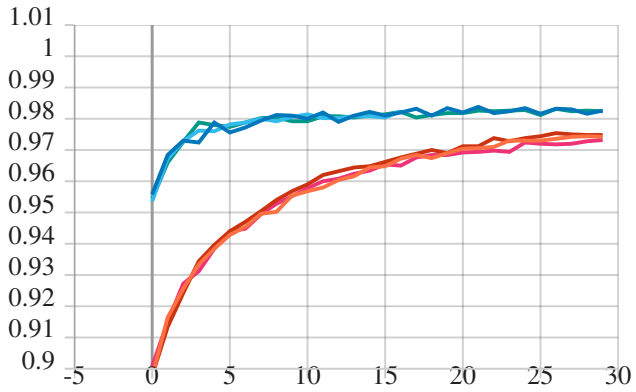

Figure 11: SGD: Accuracy (The orange curves are accuracies with SGD. The orange curves are accuracies with Nesterov accelerated gradient)

Next, we apply RMSprop and Adam to the network. The results are shown in Figure 12 and 13. Both RMSprop and Adam result in very fast learning. Also, we notice that the RMSprop accuracy and Adam accuracy start increasing after 10 epochs which means the nets are overfitting.

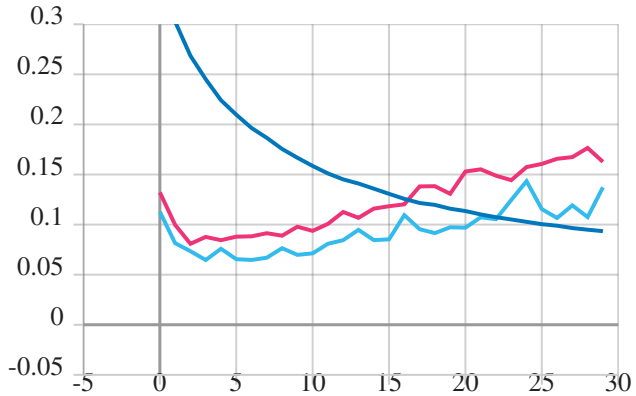

Figure 12: Optimizers: Loss (The dark blue curve is loss with SGD. The pink curve is loss with RMSprop. The blue curve is loss with Adam)

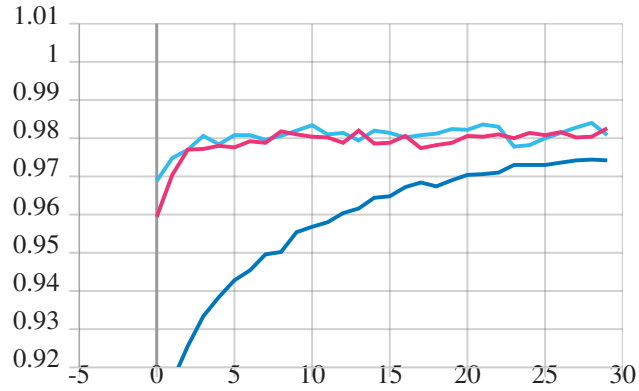

Figure 13: Optimizers: Accuracy (The dark blue curve is accuracy with SGD. The pink curve is accuracy with RMSprop. The blue curve is accuracy with Adam)

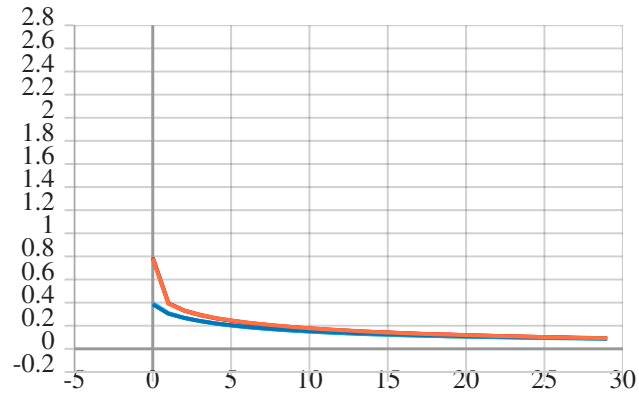

Figure 14: Dropout: Loss (The orange curves are train losses. The blue curve is validation losses)

1.1.5 Regularizations

In this section, we compared the performances of MLPs with L1, L2, Dropout, and no Dropout.

Dropout is a regularization method that randomly drops out units during the training process of a neural network. Theoretically, it approximates statistical noises in the training data to overcome overfitting. We set the `dropout=0.2`. As the results show in Figure 15 and 14, the validation loss is always lower than train loss during the 30 epochs while the final accuracy is as high as 97.38%. Therefore, we can say dropout does improve the performance of the network.

A regression model that uses the L1 regularization technique is called Lasso Regression and the model which uses L2 is called Ridge Regression. The difference is: Ridge regression adds a squared magnitude of coefficient as penalty term to the loss function while Lasso Regression adds an absolute one. Lasso shrinks the less important feature's coefficient to zero thus, removing some features altogether. Theoretically, this works well for feature selection in case we have a huge number of features. The experimental results are shown in Figure 16 and 17, L2 scores better than L1. Since there are not many features in this particular situation, we can expect that result. Also, both L1 and L2 effectively prevent overfitting, but the accuracies are much lower than the network with dropout.

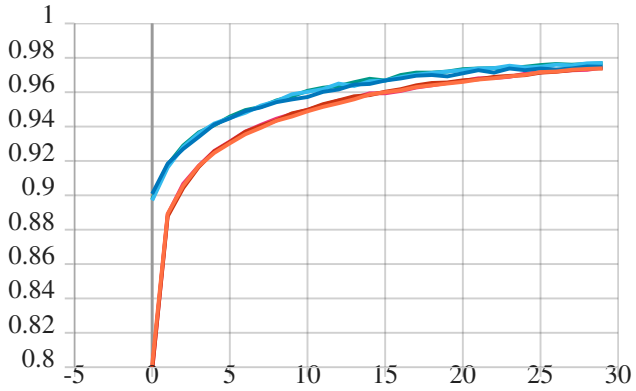

Figure 15: Dropout: Accuracy (The orange curves are train accuracies. The blue curve is validation accuracies)

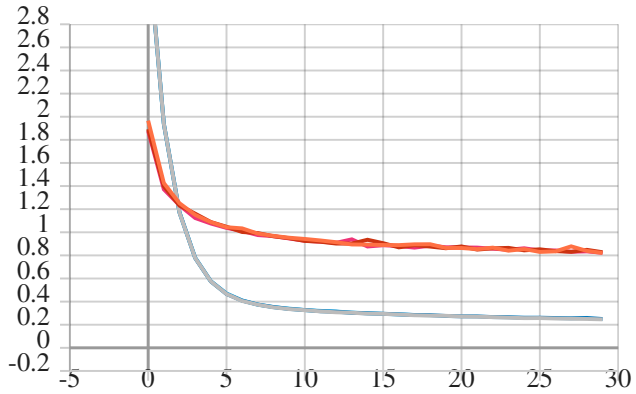

Figure 16: L1 and L2: Loss (The orange and red curves are losses with `L1=0.01`. The grey and blue curve are losses with `L2=0.01`)

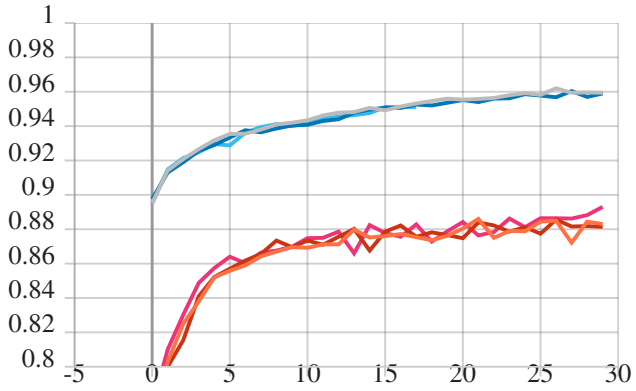

Figure 17: L1 and L2: Accuracy (The orange and red curves are accuracies with `L1=0.01`. The grey and blue curve are accuracies with `L2=0.01`)

1.2 CNN

A CNN consists of an input layer, hidden layers, and an output layer. It is composed of some special layers including convolutional layers, pooling layers, fully connected layers. Compared to the MLP, training a CNN consumes a lot of RAM and time because of the matrices calculation.

1.2.1 Vanilla CNN

We implement a classification CNN (from the textbook) that is structured as Figure 18. The parameters and architectures are left unchanged. We observe this network will be overfitting after about 14 epochs. In the following experiments, we will train all CNNs in 14 epochs and all changes are made based on this vanilla version. The CNN gives a (highest) 99.24% accuracy, and the loss and accuracy are shown in Figure 19 and 20.

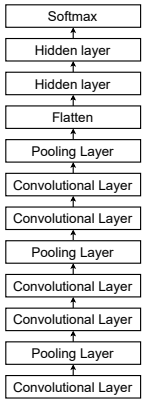

Figure 18: Vanilla CNN Structure

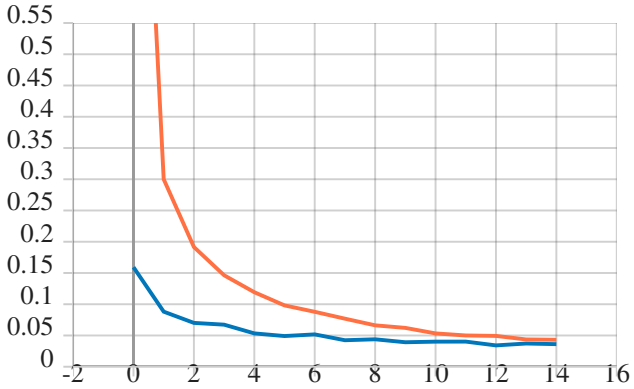

Figure 19: Vanilla CNN: Loss (The orange curves are train losses. The blue curves are validation losses.)

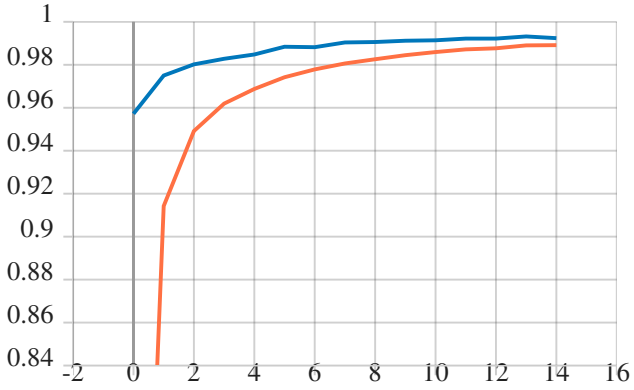

Figure 20: Vanilla CNN: Accuracy (The orange curves are train accuracies. The blue curves are validation accuracies.)

1.2.2 Initialization Strategies

We initialize the data with He initialization and make a comparison with normal initialization.

1.2.3 Activation Functions

We apply ReLu, sigmoid and tanh functions to the convlutional layers and hidden layers.

1.2.4 Regularizations

We try L1, L2 and non-dropout in the hidden layer of CNN.

2 Testing the impact of obfuscating data by randomly permuting all pixels

We permute both MNIST and Fashion MNIST with `numpy.random.permutation` and train MLP and CNN on them.

The MLP trained on the MNIST and the permuted MNIST have almost the same performances. The CNN trained on MNIST reaches an accuracy as 99.24% while the one trained on permuted MNIST has worse performance.

We observe a similar pattern when training on the Fashion MNIST. The accuracy drops to 90.96% from 38.35% that is even worse than previous results training on unpermuted images.

The performance of the CNN drops when switching from training on original to permuted images and always stays below the accuracy of the MLP throughout the whole training phase. It emphasizes that the use of standard convolutional networks is inappropriate for cases where image pixels are permuted inside the image. Hesamifard et al. [2017] propose a method to classify images that are permuted following a particular rule. It shows the possibility of effectively classifying randomly permuted images.

	MLP		CNN	
	Original	Permuted	Original	Permuted
MNIST	96.94%	93.82%	99.24%	58.12%
Fashion MNIST	87.66%	82.56%	90.96%	38.35%

Table 1: Accuracy of a CNN and MLP trained on original images and their permutations (after 15 epochs).

3 Develop a Tell-the-time network

Data set, including images and labels, are divided and shuffled in 80:20 ratio for the train and test sets (Random seed is set to 55). Also, we develop a relatively simple CNN architecture as model. What we do for this task is mainly experiment with four different suggestions given, and briefly discuss the training process and their corresponding results or performance.

To better observe how the selection of hyperparameters and measures (or function) influence the model, we visualize the training processed by tensorboard.

3.1 Experiment with combing regression and classification

3.1.1 Preparation

We borrow our ideas from "multi-head models" and correspondingly create a CNN architecture having two output heads where one head is for predicting hours and another one is for predicting minutes. Accordingly, these two heads need different measures to compute loss and error. We apply mean absolute error and accuracy , and mean squared error and cross entropy as measure to compute error and loss for "minutes head" and "hours head". For more details about measures and hyperparameters chosen, and the structure of CNN see Figure 21 and Table 2.

3.1.2 Findings

During the training process, the difference between prediction and labels for 'minutes head' on test sets drops

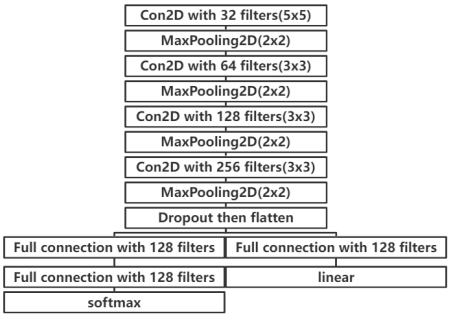

Figure 21: CNN for combination of classification and regression

Hyper parameters			
Epoch	Batch size	Learning rate	Optimizer
1200	32	0.0005	Adam
Measures			
		Loss	Metrics
	Hours	CrossEntropy	Accuracy
	Minutes	MSE	MAE

Table 2: Selection of hyperparameters and measures for model that combine classification and regression

relatively fast to reach less than 10 minutes with around 10 epochs, while the accuracy of predicting hours increase slowly with around 100 epochs to reach 50%. From the figures 22, 23 and 24, where the red line represents train sets and blue line represents test sets, it can be seen that the accuracy of predicting hours to reach 85% on test sets requires 1200 epochs, whereas MAE finally reaches less than 0.6. Furthermore, we set the learning rate to 0.0001 and retrain the saved model for another 400 epochs, after 800 epochs, and then the accuracy of predicting hours reach 90% or so.

However, due to the prediction of hours, which is seen as a classification task, even though the minutes difference between labels and prediction could be relatively low, the global error could be very high once the model misclassified the hours (e.g., the difference between "predicted" 11:50 and 6:50 is 300 minutes).

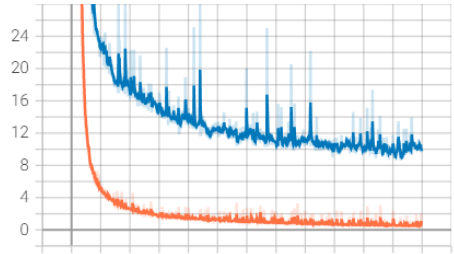

Figure 22: Combination of classification and regression: Total loss

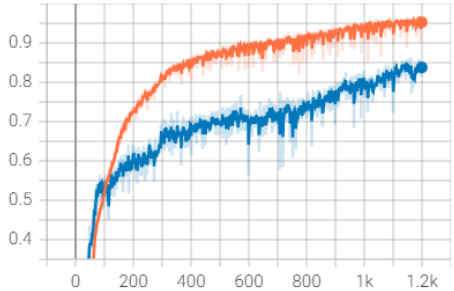

Figure 23: Combination of classification and regression: Accuracy of predicting hours

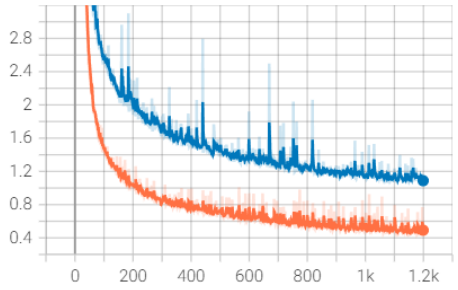

Figure 24: Combination of classification and regression: MAE of predicting minutes

3.2 Experiment with regression

3.2.1 Preparation

The label is changed to the following format:

$$[11:30 \rightarrow 11.5]\&[01:25 \rightarrow 1.42]$$

where we divide minutes by 60 and retain two decimal places.

For this experiment, the CNN architecture here has only one head with linear activation, which is, however, similar in the main structure to the one that combine classification and regression.

3.2.2 Findings

From Table 3, and Figure 25 and 26, we set epoch to 600, and initially MAE decreases rapidly but from epoch=400 onwards, it decreases more and more slowly. However, We also train our model for another 600 epochs (1200 epochs in total), but the results are the same. It can be seen that the MAE on test sets (blue line) falls at about 0.46 only, which means that there still exist differences of around 30 minutes on average.

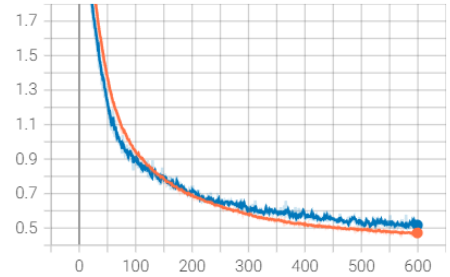

Figure 25: Regression: MAE of predicting hours and minutes (epochs=0 to 600)

Hyper parameters			
Epoch	Batch size	Learning rate	Optimizer
600	32	0.0001	Adam
Measures			
	Loss	Metrics	
	MSE	MAE	

Table 3: Selection of hyperparameters and measures for regression

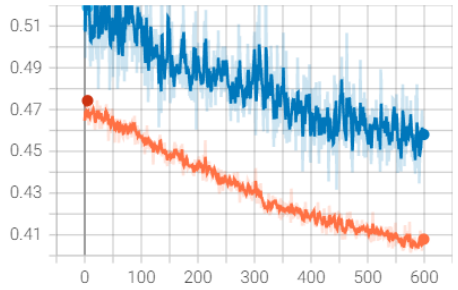

Figure 26: Regression: MAE of predicting hours and minutes (epochs=600 to 1200)

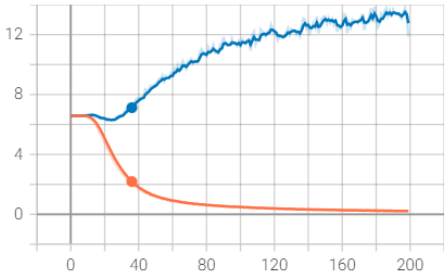

Figure 27: Classification: Loss

3.3 Experiment with classification

3.3.1 Preparation

We treat this task as 720-class classification and change the labels to 720 categories. For example,

$$[11:55 \rightarrow 715]\&[0:12 \rightarrow 12]$$

For this experiment, the CNN architecture here has only one head with softmax activation, which has 720 filters but only outputs one predicted values, which is, however, still similar in the main structure to the one that combine classification and regression.

3.3.2 Findings

Looking at the Table 4, and Figure 27 and 28 in more details, we set epoch to 200 only, and the accuracy of predicting category on train sets reach greater than 90%, while that one on test sets reach less than 20%. Moreover, the loss on test sets (blue line) start increasing after around 40 epochs.

Hyper parameters			
Epoch	Batch size	Learning rate	Optimizer
200	32	0.0005	Adam
Measures			
	Loss	Metrics	
	CrossEntropy	Accuracy	

Table 4: Selection of hyperparameters and measures for classification

3.4 Experiment with circular data using periodic function

3.4.1 Other related tries: project the labels to sine function

Initially, we split the time labels into 720 categories, as we did in the classification experiment above. Then, we write a custom loss function:

$$error\_degrees = \frac{label - pred}{4}$$

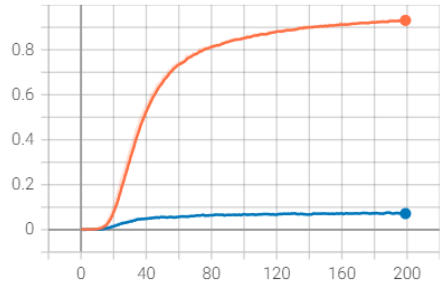

Figure 28: Classification: Accuracy of predicting category

$$error\_radians = deg2rad(error\_degree)$$

$$errors = sin(radians)$$

where variable error_degree represents that we attempt to compress the difference which denotes angle, between the labels and the predicted value to a range of degrees from 0 to 180, and where deg2rad represents a function that convert angles from degrees to radians. Finally, we map those errors into sine function (range: [0, 1] and domain: $[0, \pi]$).

For example, suppose a label is "0:00" and two predicted values are "11:50" and "0:10". The former error is $sin(710^o - 0^o) = sin350^o = -sin10^o$ and the latter error is $sin(10^o - 0^o) = sin10^o$. For this reason, we use mean squared error that compute compute the loss. Now, due to the differences are both 10 minutes, we use the idea above to guarantee that these two predicted values received the same penalty.

However, from the very beginning, it was not able to converge no matter how the hyperparameters and measures (function) is selected. Therefore, we use other methods instead, as shown the next subsection.

3.4.2 Other related tries: project the labels to 2d Cartesian coordinates on the unit circle

As we mentioned before, we here still split the time labels into 720 categories representing angles from 0^o to 719^o. Next, these "angles" are mapped into 2d Cartesian coordinates on the unit circle (e.g., $coordinate = [sin\frac{\theta}{2}, cos\frac{\theta}{2}]$, which means there is 720 time in a clock such that there are 720 coordinate points on the arc of the unit circle. For example,

$$0:00 \rightarrow 0 \rightarrow coordinate = (sin\frac{0^o}{2}, cos\frac{0^o}{2}) = (0, 1)$$

$$or$$

$$09:00 \rightarrow 540 \rightarrow coordinate = (sin\frac{540^o}{2}, cos\frac{540^o}{2}) = (-1, 0)$$

After reformulating labels, what we do is develop a CNN architecture to minimize the difference of the coordinates. For this reason, the shape and structure of two heads with linear activation, in our CNN architecture are the same, and are used to output the prediction of the coordinates x and y, respectively. Next, we use mean squared error as loss function, and use mean absolute error as metrics for us to observe the training process (See table for more details). In addition, to evaluate the "common sense" mentioned in the requirement of assignment, we use function arctan (or more precisely atan2 in python) that require the coordinate points, to convert predicted coordinates to angles which are then interpreted as prediction of time (e.g., $atan2(-1, 0) \rightarrow -90^o \rightarrow 270^o \rightarrow 270^o * 2 \rightarrow 540 \rightarrow 09:00$).

Hyper parameters			
Epoch	Batch size	Learning rate	Optimizer
200	64	0.0005	Adam
Measures			
Activation of the last layer (two heads)	Loss	Metrics	
Linear (both)	MSE	MAE	

Table 5: Selection of hyperparameters and measures for try 3.4.2

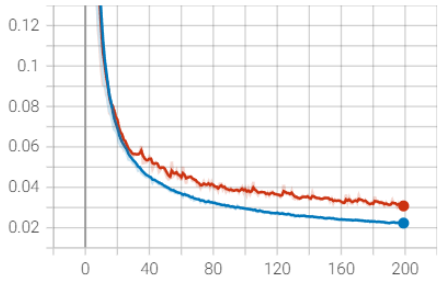

Figure 29: MAE of predicting x-coordinates for try 3.4.2

After 200 epochs, from Figures 29 and 30 , it can be seen that the MAE for both two coordinates on test sets are around 0.03. Then we apply arctan to convert these predicted coordinates into angles that represent time. It can be found that there is still a difference of about 50 minutes even though it is not possible to misclassify labels such that generate a difference of a few hours.

3.4.3 Optimal method

This method is similar to the practice for reformulating labels in section 3.4.2, but we make a slight change, that is, map the labels that consist of hours and minutes, to 2d Cartesian coordinates on the unit circle, respectively. Hence, for each time we obtain two coordinates representing hour and another two coordinates representing minute.

$$hours \rightarrow sin(\frac{2\pi * hours}{12}), cos(\frac{2\pi * hours}{12})$$

$$minutes \rightarrow sin(\frac{2\pi * minutes}{60}), cos(\frac{2\pi * minutes}{60})$$

For example,

$$03:00 \rightarrow (1, 0), (0, 1)$$

$$or$$

$$06:10 \rightarrow (0, -1), (0.5, 0.866)$$

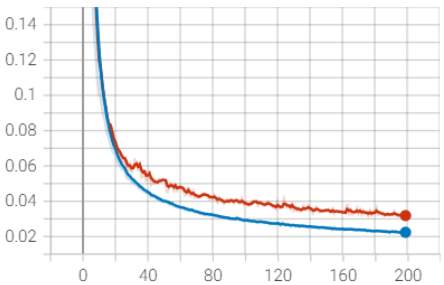

Figure 30: MAE of predicting y-coordinates for try 3.4.2

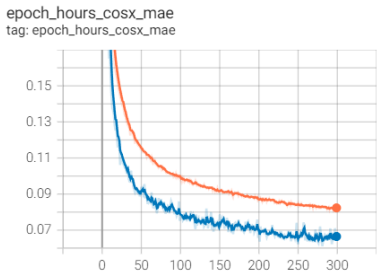

Figure 31: MAE of predicting x-coordinates of hour for optimal method

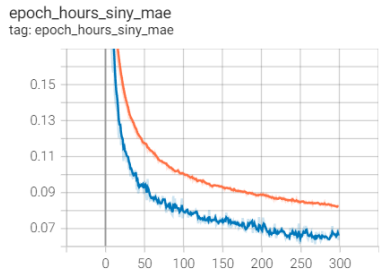

Figure 32: MAE of predicting y-coordinates of hour for optimal method

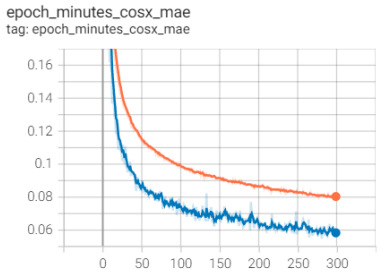

Figure 33: MAE of predicting x-coordinates of minute for optimal method

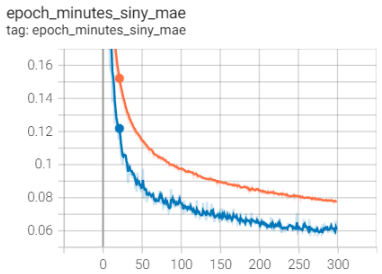

Figure 34: MAE of predicting y-coordinates of minute for optimal method

Finally, after a certain epoch, we use the resulting model to predict the orientation of the hour and minute hands (or more precisely, coordinates). Once we obtain the predicted coordinates, function arctan (or atan2 in python package) can be used to convert these coordinates into angles.

$$hour\_degree = rad2deg(atan2(hour\_x, hour\_y)$$

$$minute\_degree = rad2deg(atan2(minute\_x, minute\_y)$$

where function rad2deg() is used to convert radian computed by function atan2(), into a degree. Note that if the resulting degrees computed by function atan2() are negative, let them plus 360. Next, restore angles to time and round them up.

$$hour = round(\frac{12 * hour\_degree}{360})$$

$$minute = round(\frac{60 * minute\_degree}{360})$$

Hyper parameters			
Epoch	Batch size	Learning rate	Optimizer
300	64	0.0001	Adam
Measures			
Activation of the last layer (four heads)	Activation of the other layers)	Loss	Metrics
Linear (all)	Relu	MSE (all outputs)	MAE (all outputs)

Table 6: Selection of hyperparameters and measures for optimal method

The main structure of CNN here is still similar to the figure shown above. The main difference is there are four heads here and they are all activated with linear. Last but not least, with this method described above, the difference in minutes is able to fall at around 3 minutes (300 epochs only) on average, and the majority of "analogue clock images" can be predicted with 100% accuracy (See figures 31, 32 and 33, 34, and Table 6 for more details about training process).

References

E. Hesamifard, H. Takabi, and M. Ghasemi. Cryptodl: Deep neural networks over encrypted data. CoRR, abs/1711.05189, 2017.