Assignment 1 of Introduction to Deep Learning

Xiangyuan Tao s4256778

Ahlawat Kartikey s4178262

Mulakkayala Sai Krishna s4238206

November 13, 2024

1 Introduction

In this assignment, we used Keras to design and train neural networks, applying Multi-Layer Perceptron (MLP) and Convolutional Neural Network (CNN) models to solve two distinct image classification tasks. The first task involved classifying images from the Fashion MNIST dataset. Here, we independently developed MLP and CNN architectures, focusing on hyperparameter tuning to maximize classification accuracy. This exercise allowed us to explore the impact of architectural choices and fine-tuning on model performance for a straightforward classification problem. Later checked if the best hyperparameters from fashion MNIST translate the performance gain to the CIFAR10 as well.

2^{nd} task addressed a more complex objective: predicting time from images of analog clocks. To tackle this, we transformed the labels into four different representations and applied a unique CNN architecture to each. This approach enabled us to test multiple strategies for improving model accuracy. We evaluated all models using both traditional machine learning metrics and a "common sense" error (referred to as CSE) measure to assess practical effectiveness.

2 Task 1

2.1 Background

This task focuses on experimenting with the hyperparameters of MLP and CNN architecture and later checking if the best hyperparameters translate to the performance gain for different datasets. Experimentation was performed on the Fashion MNIST dataset. As guided in the assignment 10% of the training set is used for validation.

Table 1: Overview of the Datasets

	Fashion MNIST	**CIFAR10**
Dimension (minutes)	28X28	32X32
Channel	1 (Greyscale)	3 (RGB)
Data Size	70k	60k
Train set Size	60k	50k
Test set Size	10k	10k
Classes/categories	10	10
Balanced	Yes	Yes

2.2 Experimentation

The hyperparameter tuning was performed in four stages first for 2, 4, and 8 hidden layers in MLP and lastly CNN with 8 hidden layers. We followed the Ch-10,11 and 14 of the book[1]. We tested the model's performance on different hyperparameter combinations. In the code file of Task-1, you may find the tables representing different combinations of hyperparameters and corresponding results.

2.3 Discussion

The reason for starting hyperparameter tuning with 2 hidden layers (hl) is that the testing would take less time and more combinations could be tested. The other is to check if the best hyperparameters for 2 hidden layers also translate to 4 and 8 hl MLP. For all experimentation, the sequential API of Keras was followed. Additionally, the neurons were kept at 300 for the first hl and 100 for the rest (funnel architecture). Our focus was on experimenting more with the number of hl rather than neuron count in each layer as increasing the hl yields better results than increasing neurons, also supported in the book[1]. Apart from this, the idea behind keeping more neurons in the first layer was to make the model learn/fetch low-level complex patterns and further down we don't require more neurons as the patterns become abstract after each layer and fewer neurons can also detect the high-level complexities. This approach reduces parameters which subsequently reduces computation time and resources to train a model. Moreover, using a high number of neurons in later layers increases the chance of overfitting.

2.3.1 Observation On MLP with 2 Hidden layers

- **Epoch:** Increasing epoch (30 to 100) slightly increased the accuracy on both the train and test sets. Setting a large epoch could lead to over-fitting if the relevant regularization technique is not applied. We used earlystopping to mitigate the model to overfit. So, with earlystopping using a higher epoch makes sense as the model will now be more likely to find global minima without getting overfit. Therefore, for the rest of the testing, the number of epochs was kept at 100.
- **Activation Function:** We experimented with 'ELU' and 'ReLU' as they are considered to be best for use in hidden layers and help with vanishing gradient problem i.e., huge problem with 'tanh' and 'sigmoid' activation functions. They performed equally well. Therefore, for the rest of the testing, ReLU was used.
- **Optimizer:** Adam outperforms SGD, increasing train and test accuracy and reducing their losses. So for the rest of the experiments, Adam was kept as a default optimizer.
- **Regularization:** It prevents overfitting by penalizing large weights. Enhances generalization by keeping weights small. We experimented with different combinations of 'earlystopping', 'L1' (lasso), 'L2' (Ridge), and 'dropout layer'.
  - **Earlystopping:** As Aforementioned we used it. Tried different 'patience' values (3 & 10). 10 was more flexible in giving the model a chance to escape a local minima. Overall the below parameters were chosen:

    $$monitor = 'val\_loss', patience = 10, mode = 'min', restore\_best\_weights = True$$

    So, during training, if validation loss stops reducing for consecutive 10 epochs the model will stop training and will keep the best weight which gives the least validation loss.
  - **L1:** L1 regularization aka. Lasso, helps in keeping all neurons active, as during training ReLU ($f(x) = max(0, x)$) can assign some neurons 0 weight, aka. dead neurons. But it over-regularized making making models to underfit.

    $$L_{new} = L_{old} + \lambda \sum_{i=1}^{n} |w_i|$$
  - **L2:** L2 regularization aka. Ridge penalizes large weights by increasing their contribution to loss increase. There will be dead neurons in hidden layers. Similar to L1, it also over-regularized making making model to underfit.

    $$L_{new} = L_{old} + \lambda \sum_{i=1}^{n} w_i^2$$
  - **Dropout layer:** It is used in between hidden layers. In this, during training randomly some weights of some neurons are assigned to 0 based on dropout_rate. This forces the model to generalize and not to overfit. In book[1] the author suggests a range of 10-15% for simple NN and 40-50% for CNN. In our experimentation, we found 30% to be the best. Increasing it was causing underfitting and decreasing causing overfitting (the effect is more prominent in 8 HLs than 2 HLs). The next thing to experiment with was the number of dropout layers, with 2 hl it didn't make much sense so applied to all layers. With hidden layers 4 and 8 will experiment more with it (applying to initial vs latter layers).
- **Learning Rate Scheduler:** Even though Adam automatically keeps on adapting the learning rate as per the gradient. But in the book[1] it was suggested to use lr scheduler and try different decay_rate. In Adam the initial lr is 1e-3 with lr decay it keeps on gradually decreasing after each epoch. We experimented with

decay_rates ranging from 4 to 1e-5. Higher decay_rates were increasing the convergence time as lr kept decreasing exponentially after each epoch. Vice-versa for lower decay_rate. The optimal decay rate was 1e-3, a good trade-off between time and performance. But the model with the lr scheduler and without both performed similarly.

$$lr_{new} = lr_{old}/(1 + decay\_rate * epoch)$$

Additionally, setting up the initial lr manually is not helpful and default 1e-3 works fine.

- **Kernel_initializer:** In book[1] for all ReLU versions and ELU it was advised to use 'he_normal' kernel_initilaizer. But on closer look, it did not perform any better than the default ('glorot_uniform').
- **Batch Normalization:** BN is applied in between hl layers and before the dropout layer (if there is any). It keeps input and output from the layers in a normal form i.e., 0 mean with 1 std. deviation. It helps with better convergence and also has the effect of regularization. But not as prominent as the dropout layer. Sometimes applying both BN and dropout layer could cause underfitting. So, when applying BN we could reduce or even not apply dropout layers (depending on the dataset).
  With 2 hl best performance came out to be:

Train_acc	**train_loss**	**test_acc**	**test_loss**
93%	0.18	89%	0.33

for the below parameters:

Epoch	**Activation function**	**Optimizer**	**Regularization**	**Regularization parameter**	**lr Decay_rate**	**Initial lr**	**Kernel_initializer**	**Batch Normalization**
100	ReLU	Adam	EarlyStopping	Patience=10	1e-3	default	he_normal	Yes

2.3.2 Observation On MLP with 4 Hidden layers

- **Optimizer:** Experimented between Adam and Nadam (Nesterov-ADAM). Did not find any considerable difference in performance.
- **Regularization:** Apart from using earlystopping for all the models, we experimented with different numbers of dropout layers. For a 4 hl model maximum of 5 dropout layers can be used (if we use one between each layer). But using all resulted in poor performance, reducing accuracy and increasing loss. Moreover, 5 dropout layers with 5 BN layers resulted in even worse performing models. Using no dropout layer with BN resulted in the best-performing model. On the other hand, L1 and L2 as a regularizer over-regularized the model resulting in low accuracy and high loss.
- **Learning Rate Scheduler:** Using it did not show any improvement in model accuracy or loss.
- **Batch Normalization:** BN helped with improving model performance and reducing the convergence time.
  With 4 hidden layers best performance came out to be:

Train_acc	**train_loss**	**test_acc**	**test_loss**
91%	0.21	89%	0.33

for the below parameters:

Epoch	**Activation function**	**Optimizer**	**Regularization**	**Regularization parameter**	**lr Decay_rate**	**Initial lr**	**Kernel_initializer**	**Batch Normalization**
100	ReLU	Adam	EarlyStopping	Patience=10	1e-3	default	default	Yes

Almost all the parameters in the best performing model of 2 hl and 4 hl models are the same and the same can be said for 8 hl (will see below). That means the parameter translates even when hidden layers are increased.

2.3.3 Observation On MLP with 8 Hidden layers

- **Optimizer:** Experimented between Adam and Nadam (Nesterov-ADAM). Did not find any considerable difference in performance.
- **Activation Function:** Apart from RelU, experimented with LReLU and PReLU. No considerable improvements were noted, maybe because the task is simple and ReLU itself is sufficient to handle it. We might see distinguishable results for more complex problems between ReLU and its versions.

- **Regularization:** We noticed placing dropout layers (3-4 layers) before initial hidden layers (Dense) generalizes better than placing them before the latter layers. Maybe because initial generalization is important and the same is then passed on to the subsequent layers. Moreover, 3 dropout layers proved to be optimum, as 4 was causing over-regularization and reducing the model's ability to learn.
- **Learning Rate Scheduler:** Surprisingly this time along with Adam it increased train and test accuracy. But with Nadam there was no improvement noticed. This might be because Nadam judges the next position before computing the gradient, due to its 'look ahead' property lr scheduling is redundant and could even reduce the performance in some cases. On the other hand, using it along Adam stabilizes its gradients and helps to converge better.
- **Batch Normalization:** BN helped improve model performance and reduce the convergence time.
  With 8 hidden layers arguably best performance came out to be:

Train_acc	**train_loss**	**test_acc**	**test_loss**
92%	0.2	89%	0.32

  for the below parameters:

Epoch	**Activation function**	**Optimizer**	**Regularization**	**Regularization parameter**	**lr Decay_rate**	**Initial lr**	**Kernel_initializer**	**Batch Normalization**
100	ReLU	Adam	EarlyStopping	Patience=10	1e-3	default	default	Yes

2.3.4 Observation On CNN with 8 Hidden layers

Till now we compared the model's performance directly on the test set and then fine-tuned the hyper-parameters which could cause the model to over-generalize on it. So here we evaluated it on the val set and then after tuning evaluated the final performance on the test set. For all experiments with CNN, we kept the optimizer as Adam and the activation function as ReLU. For regularization, we only keep earlystopping, as during experimentation with MLP we found that regularizers like L1 and L2 were over-regularizing and the dropout layer is unnecessary if BN is being applied.

- **Convolutional Layers:** It is the backbone of CNN and helps the model detect features from an input image. We experimented with different numbers of it, 3 and 5.
  Architecture with 3 Conv2D layers:
  $$Conv2D_1 \rightarrow MaxPooling \rightarrow Conv2D_2 \rightarrow MaxPooling \rightarrow Conv2D_3 \rightarrow Flatten \rightarrow 8HLs$$
  Architecture with 5 Conv2D layers:
  $$Conv2D_1 \rightarrow MaxPooling \rightarrow Conv2D_2 \rightarrow Conv2D_3 \rightarrow MaxPooling \rightarrow Conv2D_4 \rightarrow Conv2D_5 \rightarrow Flatten \rightarrow 8HLs$$
  The second architecture yields better results. Two consecutive Conv2D layers are better able to extract features and increase receptive field without increasing much on parameters than 5X5. Therefore, two 3X3s are better than 1 5X5 both computationally and performance-wise.
- **Kernel Size:** In Conv2D layer the parameter kernel_size decides the receptive area for the input. We experimented with values 3 and 5. First, we tested with kernel_size 5 for all Conv2D layers, which performed worst. Then 5 for the first layer and 3 for the rest performed better. Lastly, 3 for all layers performed the best. Kernel of 3X3 is said to be the industry standard as well[1].
- **Filters:** It refers to the number of feature maps (output from Conv2D after the feature detector (kernel) is applied to the input). We experimented with 32 and 64 for first Conv2D and then doubling it for subsequent Conv2D layers. The purpose of starting with low filters is in the initial stages we focus on capturing simpler features and later we capture more complex so need more feature maps (higher filters). We found that starting with 64 filters proved to be better than 32 as it captured more features. Additionally, for 5 Conv2D arch. (displayed above) keeping filters the same for adjacent (64,(128,128),(256,256)) is better than doubling it (64,128,256,512,1024).
- **Dropout layers:** Used dropout only in fully connected Dense layers, not Conv2D layers. What was observed in MLP translates here as well i.e., 3 dropout layers placed in initial layers yield better results than placing them later. Moreover, 30% translates to the best result. A rate of 40% was over-regularizing the model.
- **Learning Rate Scheduler:** decay_rate of 1e-3 enhanced model's performance, increasing accuracy and decreasing loss on both train and validation set.
- **Batch Normalization:** BN helped improve model performance and reduce the convergence time.
  With 8 hidden layers arguably best performance came out to be: for the below parameters (para not mentioned like an epoch, optimizer, etc. are carry-forwarded from the best para of 8-hl MLP):

Train_acc	train_loss	Val_acc	Val_loss
95%	0.13	93%	0.19

Kernel	Starting Filter	Pool size	Strides	Conv2D Layers	MaxPool2D Layer	Dense Layers	lr Decay_rate	Batch Normalization
3	64	2	2	5	3	8	1e-3	Yes

2.3.5 Observation on CNN on CIFAR10

The best parameters observed from Fashion MNIST do not translate to CIFAR10. For CIFAR10 we need to apply dropout (in between Conv2D layers), and more Conv2D layers with BN to get better accuracy and generalization on the validation set. Moreover, the dropout rate of 50% worked better than 30% which was observed to be the best for dense layers. In contrast to Fashion MNIST, The CIFAR10 holds a wide variety of colored images having classes like airplanes, dogs, ships, etc. This proves that there is no single set of parameters that could yield the same level of result on different datasets.

2.4 Result

Increasing hidden layers of the model does not necessarily change the set of optimal hyperparameters. We observed this with MLP (2hl, 4hl, and 8hl) on Fashion MNIST. In regularization, dropout and earlystopping techniques were found to regularize the model and generalize better on unseen data than L1 and L2. Dropout_rate of 30% worked best in between dense hidden layers and 50% for CNN. BN helps with better convergence and also has the effect of regularization. But not as prominent as the dropout layer. Sometimes applying both could cause underfitting. So, when applying BN we could reduce or even not apply dropout layers (depending on the dataset). Two consecutive Conv2D layers are better able to extract features and increase receptive field without increasing much on parameters than 5X5. Therefore, two 3X3s are better than 1 5X5 both computationally and performance-wise. For shallower NN the effect of lr scheduling was not prominent but with 8 hl it was visible that it increased accuracy with decreasing loss.
The best parameters observed from Fashion MNIST do not translate to CIFAR10. This proves that there is no single set of parameters that could yield the same level of result on different datasets (No Free Lunch Theorem[1]).

3 Task 2

3.1 Background

This task focuses on designing a CNN capable of accurately telling the time from images of analog clocks. Two datasets are provided, each containing 18,000 grayscale images. One dataset has a resolution of 75x75 pixels, while the other has a resolution of 150x150 pixels. Each image is labeled with two integers representing the hour and minute displayed on the clock. The datasets are divided into 70% training, 10% validation, and 20% test sets. The images pose additional challenges for recognition, as they exhibit varied rotation angles and some have light reflections, which complicates CNN design and training to achieve satisfactory performance. A key aspect of this task is error measurement. Beyond standard accuracy metrics, we employ "common sense" accuracy—defined as the absolute difference between the predicted and actual time—as an evaluation metric to gauge the practical effectiveness of each model.

3.2 Classification Model

Given that the dataset represents 720 unique time points (12 hours * 60 minutes), a logical approach is to treat this as a 720-class classification problem. This formulation allows the model to predict discrete time values directly rather than separately predicting hours and minutes. However, this approach hinges on the model's ability to effectively distinguish between many classes. To evaluate the feasibility, we initially tested models on simplified versions of the problem with fewer classes to understand how well the CNN could perform on reduced complexity.

The model architecture we ultimately selected emerged from a thorough hyperparameter tuning process aimed at maximizing accuracy across both training and test sets on both low- and high-resolution images. We began with a relatively simple CNN architecture, featuring fewer convolutional layers and smaller kernel sizes. This initial model, however, struggled to converge on the high-resolution dataset, likely due to insufficient representational capacity. In response, we increased the model's depth by adding two convolutional layers and also adjusted the batch size upward to improve gradient stability.

In addition to architectural changes, we fine-tuned other critical hyperparameters, including the stride, dropout rate, learning rate, and batch size during training. These adjustments were crucial for achieving a model that generalizes well across both datasets. The final architecture, which balances complexity and performance, is presented in Figure 1.

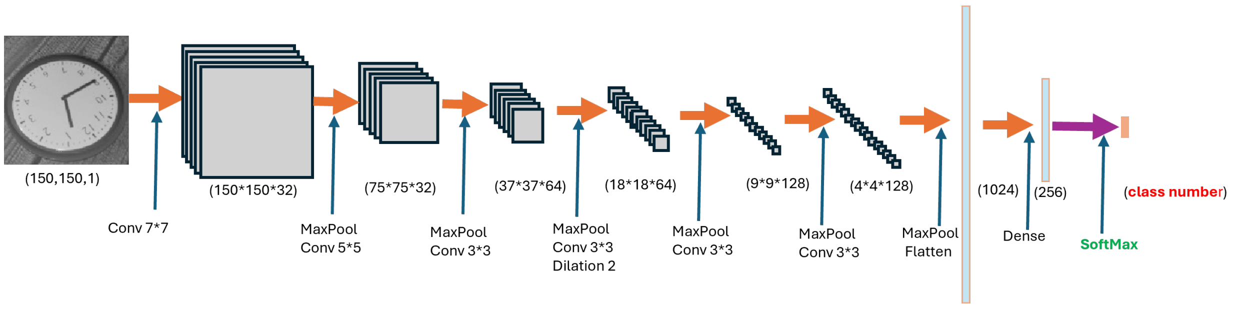

Figure 1: Classication Model Structure

We incorporated *BatchNormalization* to stabilize training and accelerate convergence, and *Dropout* as a regularization technique to reduce overfitting. Additionally, we used *EarlyStopping* to halt training when validation performance plateaued, further preventing overfitting.

3.2.1 Experience

We divided the total 720 minutes into 24, 26, 48, 72, and 120 classes, training the same pre-designed model with only the number of nodes in the final **Softmax** layer adjusted accordingly. Each class represents a specific time range, determined by the number of classes.

For instance, in the 24-class model, each class spans a 30-minute interval, so the model can recognize the time within 30 minutes, such as [3:00 3:30].

Since the model predicts a time range rather than an exact time, we cannot directly compute "common sense" error in minute intervals. Instead, we measure the error in terms of time subregions using the following:

```
    class_number_error = predict_class_number - true_class_number
    Common_Sense_Error = min(class_number_error, class_number -  class_number_error)
```

The minimum function accounts for the cyclical nature of clock time, ensuring that the error calculation respects the circular structure of time intervals.

3.2.2 Results

The table 2 presents results across models with varying output classes. **Test Accuracy** measures the model's ability to correctly classify each image within its respective time range. **Common Sense Accuracy (i=1)** indicates the percentage of predictions within one class of the correct time range, providing a measure of near-accuracy. **Mean Error** represents the average deviation from the true class in terms of time subregions.

These metrics reveal that as class count increases, the models' ability to accurately pinpoint time decreases. Models with fewer classes (e.g., 24 or 36) achieve higher accuracy but can only identify broad time ranges. Higher class counts lead to lower accuracy, with the 120-class model failing to converge, indicating difficulty with finer time intervals. Overall, these models are effective for broad time approximation, not exact time recognition.

Table 2: Test Results of Classification Models

Class Number	Time Subregion	Test Accuracy	Common Sense Accuracy =1	Mean Error (Time Subregion)
24	30	0.9338	0.9711	0.1042
36	20	0.9203	0.9641	0.1669
48	15	0.9019	0.9517	0.2756
72	10	0.8650	0.939	0.6411
120	6	NAN	NAN	NAN

3.2.3 Challenges in Classifying 720 Classes

In this subtask, we found that our designed model is unsuitable for direct classification of 720 classes. Several key challenges arise:

- **High Dimensionality**: With 720 output classes, the dimensionality of the model's output layer becomes significantly high, increasing computational demands. Such a high number of classes may lead to overfitting unless a more complex architecture and larger dataset are used.
- **Common Sense Accuracy Issues**: This classification approach does not account for the cyclical nature of time. For example, predicting 11:59 when the true time is 01:01 would be treated as a large error, despite the minimal difference. This limitation results in high penalties for small-time discrepancies.
- **Training Complexity**: Training a model with 720 classes is resource-intensive, requiring substantial computational power and prolonged training times to achieve convergence.

3.3 Regression Model

The hour and minute labels can be converted into a single continuous value, e.g., [”03:00” $\rightarrow$ y = 3.0] and [”05:30” $\rightarrow$ y = 5.5], allowing us to approach the task as a regression problem with a single output node. For this continuous target, we use Mean Squared Error (MSE) as the loss function. The model architecture remains largely the same as in Figure 1, with the *Softmax* replaced by linear for predicting the continuous time value.

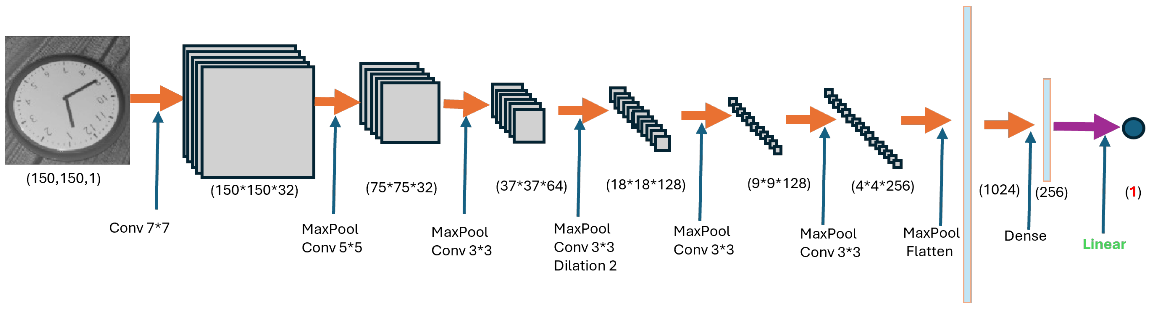

Figure 2: Regression Model Structure

3.3.1 Results

Figure 3 shows the loss curve: training loss decreases smoothly, indicating good model fit. Validation loss initially follows this trend but fluctuates after 7 epochs, diverging slightly toward the end.

Table 3 summarizes test set accuracy metrics: the Mean Common Sense Error (16.64 minutes) shows predictions are, on average, off by about a quarter of an hour. The Max Error (310 minutes) indicates occasional large deviations, likely from ambiguous cases. The median error (10.0 minutes) suggests typical errors are smaller, with some large errors skewing the average. Standard deviation (25.86) highlights prediction variability.

Prediction accuracy is low, with only 2.33% exact matches. However, 50.69% of predictions fall within 10 minutes of the actual time, showing a reasonable approximation within this range. The scatter plot in Figure 4 compares predictions to actual values, with most predictions near the ideal diagonal but considerable spread, especially in certain time ranges. This indicates the model approximates time well, though achieving exact accuracy remains challenging.

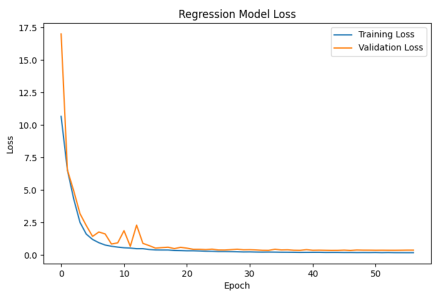

Figure 3: Loss Curve of Regression Model

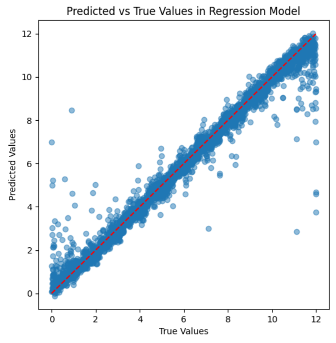

Figure 4: Visualization of Predict/True Time

Table 3: Test Result of Regression Model

Metric	**Value**
Mean Common Sense Error (minutes)	16.64
Max Common Sense Error (minutes)	310
Median Common Sense Error (minutes)	10.0
Standard Deviation of Error (minutes)	25.86
Predict Accuracy (%)	2.33
Percentage within 10 minutes	50.69

3.4 Classification-Regression Model

Since clock time is limited to 12 hours, it's feasible to use a classification model to predict the hour, while separately using a regression model to predict the minute. By integrating these two models, we can design a neural network with a multi-output structure.

Building on this idea, we retain the main architecture from Figure 1 and add two output layers after the final dense layer: one output layer with *softmax* activation for hour prediction, and another output layer with a single unit for minute prediction. To prioritize accuracy in hourly predictions over minute-level errors, we assigned different weights to each output, allowing us to control their impact on the overall training objective.

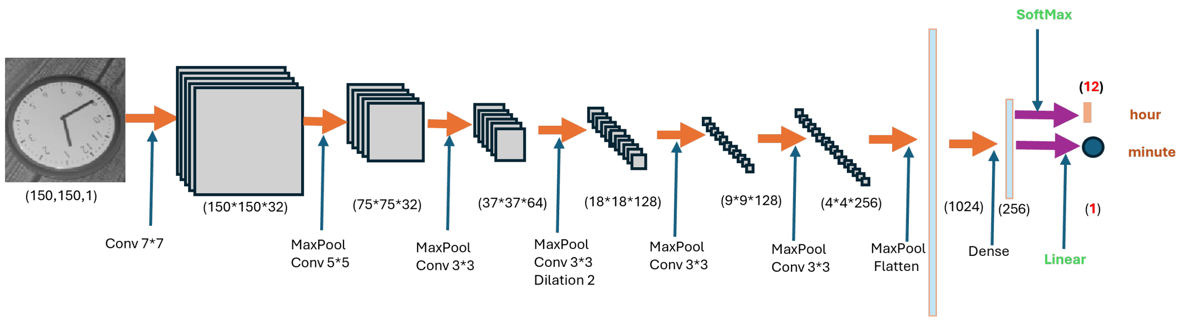

Figure 5: Classification-Regression Model Structure

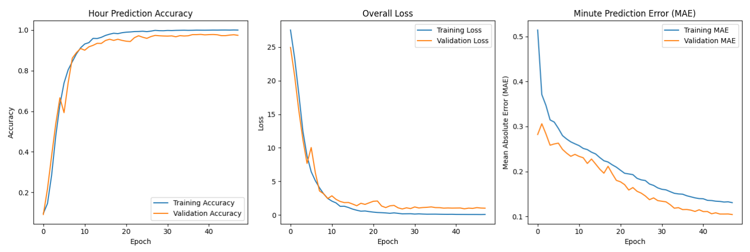

Figure 6: Training Result of Classification-Regression Model

3.4.1 Result

Table 4: Test Result of Classification-Regression Model

Metric	**Value**
Hour Predict Accuracy (%)	96.72
Mean Common Sense Error (minutes)	16.95
Max Common Sense Error (minutes)	128
Median Common Sense Error (minutes)	15.0
Standard Deviation of Error (minutes)	11.76
Percentage within 10 minutes	35.05

- **Training Performance:** Figure 6 shows the training and validation metrics for the classification-regression model. In the Hour Prediction Accuracy plot, the accuracy reaches a high level (around 97%) and stabilizes, indicating that the classification model performs well on the hour prediction task. The Overall Loss curve shows a decreasing trend, though with some small fluctuations in validation loss, suggesting that while the model generally learns well, there may be slight overfitting. The Minute Prediction Error (MAE) steadily decreases, with training and validation curves converging, showing that the regression model effectively minimizes error for minute prediction.

- **Test Performance:** Table 4, high Hour Predict Accuracy, indicating accurate predictions of the hour in most cases. Mean-CSE and Median-CSE show that predictions are reasonably close on average, although exact predictions remain challenging. Max-CSE points to occasional large errors, possibly from ambiguous images or cases where hour and minute predictions both deviate significantly. Std. Deviation of Error highlights that while most errors are close to the mean, there is some variability. Percentage shows that about one-third of predictions fall within 10 minutes of the true time, indicating acceptable performance but room for improvement in finer minute precision.

3.5 Label transformation - Multi-Regression Model

The previous models struggle with the periodic nature of clock time. They treat the gap between 11:55 and 01:05 as a large error, failing to account for the cyclic structure of time. This limitation hinders their ability to achieve high common sense accuracy. To address this, we transform the hour and minute labels into coordinate points using sine and cosine functions. This transformation allows us to design a multi-regression model that predicts the coordinate points of the hour and minute hands on a clock. This approach has the advantage of aligning the regression loss directly with the Common Sense Error. In this model, we add four regression outputs to the architecture as shown in Figure 7 to predict the coordinate points: x_hour,y_hour,x_minute,y_minute.

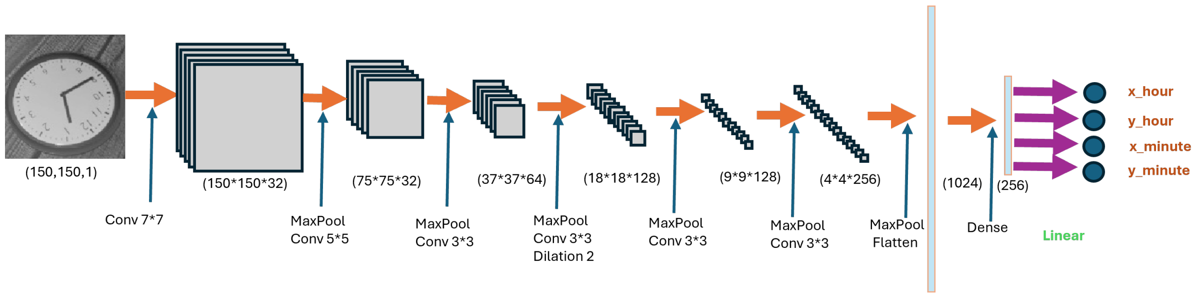

Figure 7: Multi-Regression Model Structure

3.5.1 Result

- **Training Performance:** Figure 8 shows the MAE curves for each output (hour and minute coordinates in X and Y). The training MAE decreases consistently across all outputs, indicating that the model effectively learns to minimize error for each coordinate. The validation MAE follows a similar trend but with slight fluctuations, suggesting some variability in the model's ability to generalize. Overall, the consistent decrease in both training and validation MAE suggests stable training without significant overfitting.
- **Test Performance:** Table 5 shows that the model achieves a high accuracy with moderate variability, implying that most predictions are within a narrow time range of the actual value. Percentage highlights 91.64% of predictions falling within a 10-minute window of the true time.

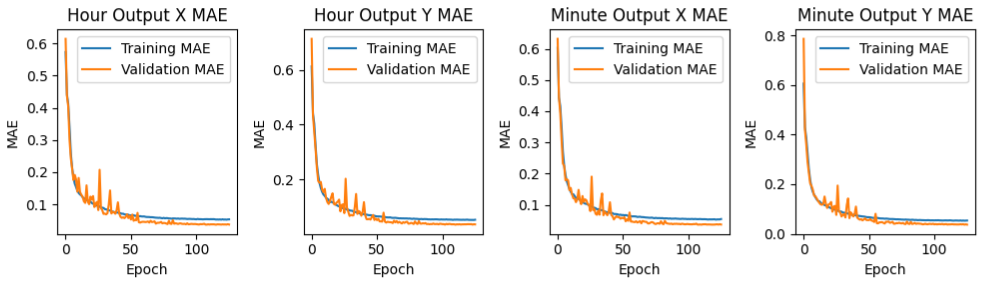

Figure 8: Training Result of Multi-Regression Model

Table 5: Test Result of Multi-Regression Model

Metric	Value
Mean Common Sense Error (minutes)	5.65
Max Common Sense Error (minutes)	345
Median Common Sense Error (minutes)	1
Standard Deviation of Error (minutes)	17.38
Percentage within 10 minutes	91.64

3.6 Conclusion

In this study, we tackled the challenge of predicting time from analog clock images, experimenting with classification, regression, and a novel coordinate-based multi-regression model. The multi-regression model demonstrates improved accuracy and reduced error variability compared to other approaches. By predicting the coordinates of the clock hands, this model effectively handles the cyclical nature of time, resulting in high precision and minimal large errors. This approach proves advantageous for applications requiring consistent, close approximations of time.

References

[1] Aurélien Géron. *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow*. O'Reilly Media, 2nd edition, 2019.