Leiden University - Deep Learning Task 1 (Group 57)

Manuel Pérez Belizón 4416880
Rejdi Danaj 3988651
Valentin Kodderitzsch 3895157

November 2024

All authors contributed equally to this group project.

0 Abstract

The key objectives of this assignment were to (i) gain some practical experience about tuning multi-layer perceptron (MLP) and convolutional neural network (CNN) architectures and (ii) apply this knowledge to develop a well performing CNN for the "tell the time" problem, in which a model needs to identify the correct hour and minutes label based on images of an analogue clock.

Our main learnings from task 1 were our insights about rough versus fine tuning, the bias variance trade off and tabula rasa versus a priori modelling. Additionally, we identified practical hyper-paramter ranges. For task 2, we leveraged transfer learning and the VGG16 architecture to fit a multi-head model that classifies hours and minutes separately. In the end, our multi-head double classification model has a common sense difference of **1.02 minutes** or 1 minute and 1 second to be precise.

1 MLP & CNN Intuition (Task 1)

The main objective of Task 1 was to learn how to define and train simple neural networks using the Keras library, an open-source neural network framework written in Python [3]. Additionally, this task aimed to build intuition for manually tuning hyperparameters of MLPs and CNNs to understand their effects on model performance.

To achieve this, we first experimented with various MLP models trained on the Fashion MNIST dataset [10] after studying Chapter 10 of Géron's "Hands-On Machine Learning" textbook [4]. Fashion MNIST is a well-known dataset in computer vision, consisting of 28x28 grayscale images categorized into 10 classes, with 60,000 training images and 10,000 test images. Next, we identified the top 3 MLP models based on their performance on the Fashion MNIST dataset and applied these models verbatim to the CIFAR-10 dataset [6]. The CIFAR-10 dataset, another benchmark in computer vision, contains 32x32 RGB images also classified into 10 categories, with 50,000 training images and 10,000 test images. This step was intended to assess whether the hyperparameter tuning insights gained from Fashion MNIST could translate effectively to a different dataset.

Finally, we repeated the experiment using a CNN architecture. We tuned various CNN models on the Fashion MNIST dataset to develop a similar intuition for CNN-specific hyperparameters. The top 3 performing CNN models were then applied verbatim to the CIFAR-10 dataset.

1.1 Multi-layer Perceptron

As previously mentioned, the Fashion MNIST dataset consists of gray scale images of size 28x28 pixels, resulting in an input dimension of $784 = 28^2$ for all models. While the input layer is consistent across MLP architectures, the design of subsequent layers and hyperparameters offers a wide range of choices. Technically speaking, the combinations of architectural designs and hyper-parameters are infinite. Therefore, to navigate these possibilities effectively, we propose a two-step process: **rough tuning followed by fine-tuning**.

Rough tuning involves understanding the bias-variance trade-off, which is a function of model complexity. The goal is to identify the "MSE valley"[1], where models strike a balance between bias and variance and perform consistently well. Defining "complexity" with a single numerical measure is challenging, as it involves multiple hyperparameters and architectural decisions. However, a useful insight comes from the behavior of smooth functions in high-dimensional spaces: as dimensionality increases, the peaks and valleys of these functions tend to flatten. This can be attributed to curse of dimensionality[5], where the influence of any single parameter diminishes when averaged across many dimensions. In our

[1]We refer to the loss as MSE but the bias-variance trade-off holds for any categorical or numerical loss function, even though the loss function then becomes more difficult to decompose/quantify exactly into bias and variance.

case each dimension captures a unique aspect of model complexity, such as the number of layers, dropout rates, etc. This results in a broad "MSE valley" where many combinations of hyperparameters and architectures perform almost as well as the "best" combination.

We propose 2 main strategies to reach the MSE valley. The **tabula rasa** approach starts without any prior knowledge. Start by fitting one overly complex model and another overly simplistic one. Then, iteratively adjust the architecture to find a model with intermediate complexity that falls within the MSE valley. This step may require substantial architectural changes. Once a model of suitable complexity is found, fine-tuning can commence. The second strategy is the **a priori** approach which leverages existing knowledge form similar tasks. Although each problem is unique, proven architectures from similar tasks with comparable input (e.g., images) and output types (e.g., 10 classes), with similar datasets can provide a strong starting point. Dataset similarity can be assessed in various ways, but for classification tasks comparing between-class and within-class variances is helpful. With this approach, one can either replicate a known architecture and proceed to fine-tuning or use transfer learning to build on top of pre-trained models and then fine tune those. Note that this analysis focuses solely on the classic training regime and does not address the "double descent" phenomenon, which requires significant computing resources and data that were beyond our current scope.

1.1.1 Experimental Setup

For the Fashion MNIST dataset, we performed 14 experiments to identify our top 3 performing MLP models. These experiments are summarized in table 1 Before we ran our 14 experiments we tried out three different optimization algorithms: SGD, Adam and AdamW. We did notice any significant difference between them in terms of model performance. However, Adam had slightly faster convergence time which is why we chose the Adam optimizer for our 14 experiments.

Table 1: MLP experimental set-up (Fashion MNIST)

Complexity	**Hidden Layers (k)**	**Neurons**	**Regularization**
Simplistic	1, 1, 1	2, 5, 10	None
Mid	1, 2, 5	k*100	None, L2, Dropout
Complex	12, 15	k*100	None

The main experiment compared a MLP with 1 hidden layer versus 2 hidden layers versus 5 hidden layers. For the number of neurons per layer, we followed the MLP example architecture as described in Géron's "Hands-On Machine Learning" textbook [4]. This represents an a priori rough tuning approach. The first hidden layer has 100 neurons, the second has 200, the third 300 neurons and so on. Therefore, the **k-th hidden layer has $k \times 100$ neurons**. So far this gives us 3 different architectures. Then per architecture, we applied 3 regularization techniques: dropout, L2 and no regularization. This results in $9 = 3 \times 3$ models. Regularization techniques can be seen as fine tuning.

Then, we build 3 overly simplistic and 2 overly complex models. This can be viewed as tabula rasa rough tuning. The overly simplistic models have 1 hidden layer each and 2, 5, and 10 neurons only in their respective hidden layer, with no regularization. The overly complex models have 12 and 15 hidden layers respectively, no regularization and follow the $k * 100$ formula for the number of neurons for the k-th hidden layer. This results in $14 = 9 + 3 + 2$ models in total in our experiment.

All 14 models use the ReLu activation function and use the softmax function for the output layer. All models with regularization use either a drop out rate of 0.2 or a L2 regularization parameter of 0.01. These regularization parameters were taken from the MLP example presented in the "Hands-On Machine Learning" textbook [4]. We did not further tweak the regularization parameters as our goal was rough tuning and not fine tuning.

To identify the top 3 best performing MLPs on the Fashion MNIST dataset, we used the original train test split of (60,000 : 10,000) images. We reserved 5,000 of the 60,000 training images for the validation set as specified in the "Hands-On Machine Learning" textbook [4]. The validation set helps with identifying model overfit. Even though the machine learning best practice states to only use the test once, we used the test set 14 times to evaluate model performance of all 14 models. This decision was informed by the final goal of our study, which involves testing the top 3 models on the CIFAR-10 dataset. The CIFAR-10 dataset acts as an independent "real" test set, remaining completely unknown to the models during training and testing on Fashion MNIST. This approach ensures that we gain tuning insights on the Fashion MNIST dataset, while the top 3 models are evaluated in a truly independent context using the CIFAR dataset.

1.1.2 Results

Our top 3 best performing MLP models on the Fashion MNIST dataset are:

1. No regularization with 2 hidden layers (87% MNIST) with 46.6% CIFAR test set accuracy

2. No regularization with 1 hidden layers (85.5% MNIST) with 42.6% CIFAR test set accuracy
3. Dropout regularization with 1 hidden layer (84.6% MNIST) with 40.9% CIFAR test set accuracy

Table 2: MLP test set accuracy (Fashion MNIST)

Complexity	**Hidden Layers (k)**	**Neurons**	**Regularization**	**Test set accuracy**
Simplistic	1, 1, 1	2, 5, 10	None	44.7%, 74%, 81%
Mid	1, 2, 5	k*100	None, L2, Dropout	85.5%, 87%, 83.7% (None) 72.2%, 71.6%, 51.8% (L2) 84.6%, 83.9%, 83.3% (Dropout)
Complex	12, 15	k*100	None	67.2%, 20.5%

We observed that our top 3 best performing models all perform similar well at around 85-87% test set accuracy on the Fashion MNIST dataset. These models were selected based on their performance metrics, as shown in table 2. However, when we applied those top 3 models verbatim on the CIFAR dataset, the models performed poorly at about 40-46% test accuracy. We will discuss possible explanations for this result in the next section.

Figure 1 shows that models with 2 or 5 hidden layers and dropout regularization achieve the lowest test loss on the Fashion MNIST dataset. The figure plots test loss against the parameter count for each model.

1.1.3 Discussion

Let's first discuss why the MLP models trained on the Fashion MNIST dataset do not perform well on the CIFAR dataset. We believe that the Fashion MNIST dataset and the CIFAR dataset are too dissimilar for model performance to translate from one dataset to another. If we visually inspect the data, it becomes evident that the Fashion MNIST dataset has low within class variance, ie. there is little variation between images of type "trousers" since they all look quite similar. However, the **CIFAR dataset has higher within class variance**. For instance when considering images from the class "horses", it becomes clear that there are running horses, jumping horses, horses from the side and from the front. All images of "trousers" from Fashion MNIST on the contrary are just images of straight legged trousers taken from the front. Therefore, models trained on Fashion MNIST do not need to learn how to account for high within class variance, as all "trouser" images for instance look similar. However, when the model is exposed to a dataset with high within class variance, the model fails to capture nuances and spatial relationships within the data.

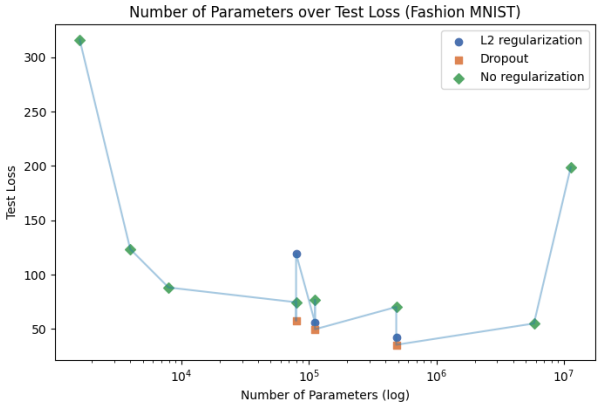

Figure 1: MLP test loss (Fashion MNIST)

A possible solution could be CNNs as their architecture extracts spatial feature maps, enabling them to better deal with high within-class variance data. CNNs leverage convolutional layers that capture local spatial patterns and learn hierarchical representations, making them more suited for datasets like CIFAR-10. Additionally, Fashion MNIST images are 28x28 grayscale pixels, while CIFAR images are 32x32 and contain three color channels (RGB). An MLP trained on smaller, grayscale images may not effectively capture the richer information provided by color channels and higher-resolution inputs without extra training.

Another insight we gained from this experiment is that CIFAR test set accuracies presented in table 2 are consistent with our expectations of a **flat "MSE" valley**. Figure 1 is further evidence, as it shows the CIFAR test loss (sparse categorical crossentropy) over the model parameter count. Here, parameter count serves as a proxy to capture model complexity. We observe that all models expect for the most simplistic and the most complex have a test loss of below 140. As such there are quite a few models, that are in the "valley" with similar model performance. Interestingly, the best performing models were all mid level complexity models that sit at the bottom of the valley. This is consistent with our expectation and highlights the importance of rough tuning. Once mid level complexity models are found, e.g. by adjusting the parameter count, fine tuning via drop outs can significantly lower the test loss.

1.2 Convolutional Neural Network

1.2.1 Experimental Setup

When testing for the 3 best models in the CNN case, the experimental approach has been adjusted slightly. We expect architecture to impact performance more than raw parameter count, a point that will be discussed later. To account for such a large variation of architecture choice the testing of each hyper-parameter was done by fitting a model which is a variation of a base model resembling a pseudo-controll. The control model named "cnn_base" is the same one used in the book "Hands-on machine learning" [4].

In total, 10 models were trained on Fashion-MNIST and compared across the following 5 main variations of the base model: (i) decreasing or increasing channel size, (ii) kernel size, (iii) network depth, (iv) two pooling methods and (v) raw parameter count. Each of these models was then evaluated on the Fashion-MNIST test data. Accuracy is the chosen metric to determine the 3 best performing models. All models have the same activation function (ReLu), activation output function (softmax), and optimizer (Adam).

The first comparison is done to see if the progression of **channel count** throughout the layers affects the results. Both models have 10 total layers, with 4 convolutional layers of 3x3 kernel size. The channels range from 64 to 512, doubling or halving at each convolutional layer. It should be noted that even though the overall architecture is the same for both models, the increasing model results in a higher parameter count (1.7 million) compared to the decreasing model (1.4 million).

The next comparison tests the effect of **kernel size** on model performance. The architecture of the base model is used on two varying kernel sizes. 1x1 kernel is named low and 5x5 kernel is named high. All layers have the same size kernel to reduce potential variation and all other hyperparameters are kept fixed. As in the channel progression test, the parameter count varies depending on the kernel size. However, this variation is more severe, with the 1x1 kernel model having 0.43 million parameters and the 5x5 kernel model having 3.5 million parameters.

The third area of interest is **network depth**. Two models were initially tested: one with only 2 convolutional layers and one with 10 convolutional layers. However, because the original 10-layer model did not meet performance expectations, a residual version of the 10-layer model was also tested. All three models used a 3x3 kernel size, with all other hyperparameters kept constant. The final parameter counts were approximately 1.9 million for the 10-layer 'deep' CNN, 2.6 million for the 2-layer model, and 3.6 million for the residual network.

Pooling is also an area of interest in convolutional networks. Therefore, two models were tested to evaluate the impact of different pooling methods. One model replaced the max pooling layers of the base model with average pooling, while the second model used global average pooling, reducing all channels before flattening. The models were then tested to determine if the pooling method had a significant effect on the results. As expected, the global average pooling model had fewer parameters, approximately 1.1 million, compared to the average pooling model with about 1.4 million.

To also examine the effect of a **very simple** model compared to a more **complex** one, two models were fitted. One model had only a single convolutional layer with a kernel size of 1, resulting in a parameter count of just 7,800. The other model was more complex, with 5 convolutional layers and approximately 7.6 million parameters.

1.2.2 Results

Our three best CNNs are shown in table 3. All three models have very similar performance in terms of accuracy. Although unexpected, the shallow model is the best-performing one by a very small margin. It must also be noted that all other models performed within the range of 89-90%, with only two exceptions. First, the very simple model with a parameter count of 7,800 performed poorly as expected, with a test accuracy of approximately 10%. The second outlier in performance is the deep model, which also only reached an accuracy of about 10%. Interestingly, kernel size and pooling did not seem to have a big effect in terms of performance, compared to the base CNN model from the textbook.

Table 3: Best Three Models on Fashion-MNIST Dataset

Model Type	**Parameter Count**	**Accuracy and Loss**
Shallow	2,586,966	Accuracy: 0.92, Loss: 0.4710
Channel decrease	1,443,446	Accuracy: 0.92, Loss: 0.3794
Channel increase	1,702,390	Accuracy: 0.91, Loss: 0.6053

All three best models were then fitted on the CIFAR data, with the results shown in the table 4. An overall decrease in accuracy is noticeable, with the shallow network standing out with an accuracy of 67% on CIFAR, compared to its original accuracy of 92% on Fashion-MNIST.

Table 4: CNN Test Set Performance (CIFAR)

Model	Parameters	Accuracy and Loss
Shallow	1,558,234	Accuracy: 0.67, Loss: 1.5582
Decrease Channel	913,650	Accuracy: 0.78, Loss: 0.9137
Increase Channel	1,664,133	Accuracy: 0.78, Loss: 1.6641

1.2.3 Discussion

Looking at the results of the convolutional networks on the Fashion MNIST, we notice that generally, the accuracy does not drop beneath 88%, with one notable exception. The simplistic model with only 7,800 parameters struggled to fit the data well. This was an expected result, as the model's architecture is far too simplistic to fit the data properly. The model is also greatly limited by having a kernel of size 1, which leads to a very simplistic feature map. Another anomaly worth mentioning is the deep model, which failed to train altogether. At first, this seems counterintuitive, as the receptive field of a deep model should be able to distinguish features more effectively. However, this appears to be an issue of "shattered gradients." The model finds it difficult to obtain a local minimum due to the gradients either vanishing or exploding. This conclusion is supported by the performance of the residual network, which shares the deep network's structure but includes residual layers. Residual layers provide shortcuts that help stabilize the gradient flow, mitigating vanishing or exploding gradients by skipping over layers.

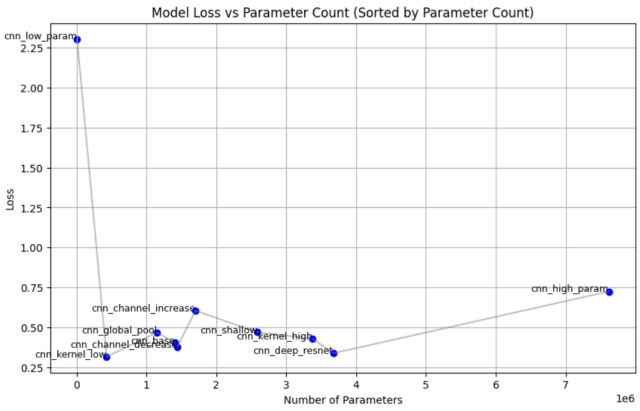

Figure 2: CNN test loss (Fashion MNIST)

Overall, convolutional networks managed to predict data from the Fashion MNIST considerably well. As mentioned in section 1.1.3, this might be due to the low within-class variance of the Fashion MNIST dataset. These findings suggest that while Fashion MNIST allows even relatively simple CNN architectures to perform well, more complex datasets, like CIFAR, reveal greater performance discrepancies.

The results for CIFAR-10 showed noticeably lower average performance across the three models. This is to be expected to some extent for two reasons. First, CIFAR-10 presents greater within-class visual diversity, as noted in section 1.1.3, which increases the challenge for models originally trained on simpler datasets. Second,the models were explicitly based on an architecture chosen for the Fashion-MNIST data. However, we see an interesting result when comparing the individual performance decreases of the three models. The two-channel progression models—namely, channel-increasing and channel-decreasing had a similar change in accuracy of approximately 13.5%. The shallow model, however, experienced a larger accuracy decrease of approximately 26%. This insight leads us to conclude that, for a convolutional network applied to a dataset with higher variance, **network architecture takes precedence over raw parameter count**. This conclusion is further supported by the better performance of the channel-decreasing model, which, despite having around 600,000 fewer parameters, achieved 12% higher accuracy than the shallow model.

As noted in section 1.1.3, raw parameter count influences MLP performance, a trend also seen in CNNs but to a lesser degree. Figure 2 illustrates that models with similar parameter counts have comparable loss values, with extreme cases leading to higher loss. For example, the CNN with 7,800 parameters underfits (high loss), while the over-parameterized model with 7.6 million parameters overfits and has a high loss as well. Therefore, we conclude that optimal hyperparameters remain crucial, and parameter count can still be seen as a proxy for complexity but network architecture prevails, especially for high variance datasets like CIFAR. Note that the deep CNN did not converge due to gradient issues, so we decided to treat it as a NA value and omitted it from figure 2.

1.3 Hyperparameter Ranges & Learnings

After completing our MLP and CNN experiments on the Fashion and CIFAR dataset, we identified the following hyperparameter ranges: (i) Batch size between 62 and 128, (ii) learning rate between 10^{-3} and 10^{-6}, (iii) dropout rate between 0.1 and 0.7, (iv) kernel size of either 3x3 or 5x5, (v) number of layers and channels really depends on the variation of the dataset, so follow a rough tuning approach and (vi) set a seed number to account for random weight initialization.

Finally, we would like to present an analogy[2] using the **road (data), a vehicle (model) and the driver (optimizer)**. When the data is messy or unprocessed, resembling a rough, uneven road, it requires a more capable vehicle, such as a CNN, to navigate the difficult terrain. In contrast, well-preprocessed, clean data, like a smooth road, allows for simpler vehicles, such as MLP, to perform effectively. The vehicle's choice is pivotal to balance overfit versus underfit. Once the correct vehicle is selected, the optimization algorithm (the driver) plays an important role, but no driver can compensate for a poorly chosen vehicle. Lastly, just as a well-paved road improves vehicle performance, proper data preprocessing enhances the effectiveness of the chosen model.

2 "Tell-the-time" CNN (Task 2)

The objective of task 2 is to develop and train a convolutional neural network that can accurately tell the time from images of analog clocks. The dataset provided consists of 18,000 grayscale images of analog clocks, each paired with labels indicating the hour and minute displayed. The primary performance metric is the "common sense difference" (CSD), which measures the **smallest absolute time difference between the predicted and actual time**. For example, if the true time is 12:05 and the predicted time is 11:55, the CSD is 10 minutes, not 11 hours and 50 minutes. This metric ensures that errors wrap around the 12-hour clock format, capturing real-world discrepancies more intuitively. The details of the CSD calculation will be further discussed in a later section.

In addition, we were explicitly asked to **experiment with various label encoding strategies**, which required adapting our model architectures accordingly. Specifically, we had to treat the labels as distinct classes (classification), continuous decimal values (regression), and combinations that led to multi-head models. This requirement allowed us to analyze how different encoding methods impact model performance and CSD accuracy. Through these experiments, we compared the results across different CNN architectures to identify the most effective approach.

2.1 Data Exploration

The dataset contains 18,000 images of clocks, each captured at various times, with two class labels: one for minutes $m = 0, 1, 2, ..., 59$ and one for hours $h = 0, 1, 2, ..., 11$, resulting in $720 = 60 \times 12$ possible class labels. Each image is of size 150x150 pixels[3]. In our training set, this leads to an average of 15 images per class label, as we decided on a 60:20:20 split between training, validation, and test sets. We chose this split to meet the 20% test set requirement as specified in the assignment, while leaving the proportion of the validation set flexible, since the assignment did not specify how to allocate it within the 80%=100%-0.2% training set. We will now further explore the data to justify our train, validation, test split.

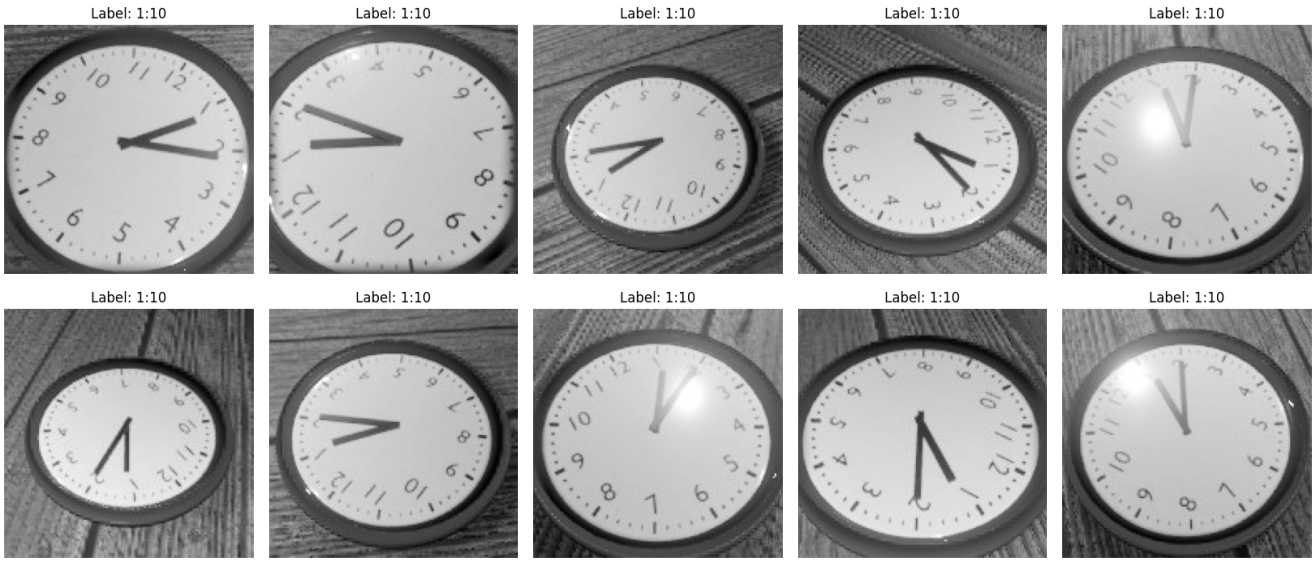

Figure 3: 10 randomly selected images of label "1:10" from the training set

[2]We discussed this analogy during class with Andrius Bernatavicius.
[3]There is also the exact same dataset with image size 75x75 pixels, but we discarded this dataset as it lead to poorer performance.

Upon inspecting the data, we found that all images appear to feature the same clock model, which we will refer to as Clock A. This clock resembles a standard IKEA kitchen clock. When we randomly selected 10 images, it became clear that only Clock A appears in the dataset, with no variation in clock style, e.g. no church tower or wristwatch clocks. We repeated this visual check multiple times. In the training set, each class label is represented by approximately 15 images of Clock A, but with varying levels of zoom, shear, glare, and rotation. This is illustrated in figure 3. However, closer visual inspection suggests that the variations in the images likely stem from digital manipulation, where the same few photographs of Clock A were modified through different levels of zoom, shear, glare, and rotation, rather than from distinct photographs taken under different real-world conditions.

Given that the **variability within each class label seems artificially generated**, we decided to reduce the training set to 60% of the data. We will then augment the training set ourselves by applying random zoom, shear, and rotation transformations. This approach will allow us to effectively expand our training dataset by a factor of 10 or more, while preserving the validation set and the test set. Both the validation set and training set are of the same size (20% of 18,000) and have not been augmented. This ensures that the validation set is as representative as possible as the test set, and explains our 60:20:20 splitting choice.

2.2 Label Encoding

As previously specified, we were asked to encode the time labels in three different ways: (i) time as 720 classes, (ii) time as decimals and (iii) a representation of our choice that can fit a multi-head model.

720 classes: The advantage of encoding time using 720 classes is that the predictions have the same level of precision as the true labels, since in both cases the smallest unit of time is minutes. Additionally, this distinct representation of time requires little pre-processing as the labels are already given as minutes and hours and are therefore easily humanly interpretable.

However, there are two main disadvantages which are computational and logical in nature. Firstly, training a CNN on a 720 class classification model requires either vast amount of data, a pre-trained model to build on top of, or both. For our dataset, it would most likely be difficult to train a well performing 720 class classifier from scratch, as we only have about 15 training images per class as explained in section 2.1. Such a model would likely fail to generalize to unseen data. The second disadvantage is that classes fail to capture the circular nature of time, since classes have no inherent ordering to them. In a classification setup, labels do not inherently convey the fact that 23:59 and 00:00 represent consecutive moments in time. To further illustrate this, it is possible to replace the labels such that they have no inherent connection with time (e.g. encode 23:59 as an animal label like dog) and the model would still learn well, as long as the "time to animal" labels are consistent. This shows that the classification loss function does not account for the sequential order of the time classes.

Decimals: For the regression case we have been asked to use the following formatting: convert the minutes to decimal by dividing by 60, and then add it to the hour represented by an integer. As such, 03:00 becomes 3.0, and 05:30 becomes 5.5. The advantage of this label representation is that it can be trained on a regression model, with either MSE or MAE as a loss function. We will argue later why MAE makes more sense. Additionally, a decimal label representation correctly captures the sequential nature of time, so 5.5 is exactly 30 minutes apart from 5.0.

The main disadvantage of decimals is that they fail to capture the cyclical nature of time. The regression model will make predictions that can be any real value, so it fails to "wrap" around the clock at 12.0. As such, a model might mistakenly predict 12.1 even though the prediction should be "wrapped" back to 0.1, since the true labels only exist in the [0.0, 11.983] interval. Another disadvantage of decimals is that they are not very human readable. An alternative is to encode time as integers by using the "hour * 60 + minutes" formula. This gives you 0 to 699 integers. Integer encoding is also more helpful for the common sense difference function, which we will explain in the next section. However, integer encoding still faces the same non-cyclical encoding issue as decimal encoding.

Multi-head: The assignment did not specify how to encode labels for the multi-head model, apart from stating that the minutes and labels should have their own model output. Therefore, there exist 4 different combinations: (class, class), (class, reg), (reg, class) and (reg, reg), where the first element in the tuple represents the hour and the second the minute. Since our regression model did not perform well, we disregarded the (reg, reg) case but tested the remaining three cases. We will explain this further in the results section. The main advantage a multi-head model is that there are now more images per class. The original 720 class classification model had around 15 images per class in the training set. Multi-head allows the the hour classes have $900 = 15 \times 60$ images per class, and $180 = 15 \times 12$ images per minute class. More images per label should lead to considerably better classification accuracy within each sub-task (e.g. hours and minutes) compared to the original 720 class classification case. As such, this should reduce the overall CSD.

2.3 Loss Functions

For classification tasks, deep learning models usually use the categorical cross entropy loss function [4], especially in a multi-class classification setting like for this assignment. However, as pointed out in the assignment sheet, sometimes the model's loss function does not align well with the real-life objective function that is meant be optimized. A low cross entropy loss tells you that the model fits wells. However, it does not tell you how "off" the predicted time is from the true time on the clock.

Therefore, we were asked to develop a "common sense difference" (CSD) metric. As mentioned in the introduction to section 2, CSD measures the smallest absolute time difference between the predicted and actual time. It therefore captures the cyclical nature of time.

Algorithm 1: Common Sense Error Calculation

Input: `predicted_time` (real value), `actual_time` (integer between 0 and 719)
Output: Common sense error (real value)
Function `common_sense_error(`*predicted_time, actual_time*`)`**:**
  $\Delta \leftarrow$ `abs(predicted_time` $-$ `actual_time)`;
  return $\min(\Delta, 720 - \Delta)$;

The pseudo-code above shows our implementation of the CSD metric. We begin by converting the true labels onto the 720 minute scale, as minutes are the smallest unit of time in our true labels. Using the "hour * 60 + minutes" formula, we end up with a scale of $720 = |\{t \in \mathbb{Z} : 0 \leq t \leq 719\}|$ unique integer values. The predicted values can be decimals or integers, as long as they are on the 720 minute scale. This enables both regression and classification outputs to use the same CSD loss function. Common sense tells us that clocks are cyclical in nature. As such, the loss must "wrap around" the midpoint of the clock. Therefore, the CSD must be smaller or equal to the mid point of $360 = 720/2$. It is possible to short hand this notation by using the absolute, min logic as described in the pseudo-code above.

Let's consider an example to showcase how CSD computes the shortest path on the clock. In the classification case, the predictions are integers on the 720 minutes scale. If the true time was 11:50 and the predicted was 12:10, then the CSD will be 10 minutes and not 11 hours and 40 minutes. Now consider the regression case, where the model predicts 10.0001 (on a 720 scale) which is about 12:10 and a fraction of a second on the normal clock. The difference between the true and predicted time would be $700 = 710 - 10$ if used the mean absolute error (MAE) loss function. Here 10.0001 is purposefully rounded to 10 to make calculations easier. Using MSE as a loss function would result in a much larger value ($490,000 = 700^2$), highlighting its sensitivity to large deviations compared to MAE (700) or CSD (10)

Therefore, we think that MAE is a better loss function to train regression models than MSE. However, when it comes to evaluating regression models for the clock task, **CSD is the most sensible metric**. In an ideal world, one would train the regression model using CSD as a loss function instead of as a final evaluation metric, but this is technically difficult to implement, so MAE is the next most appropriate loss function. Also, numeric label encoding that is not circular in nature might not be the best encoding strategy to begin with. In the classification case you have no choice but to use a categorical loss function like cross entropy to train the model, and then use CSD as an evaluation metric.

2.4 Architecture Choice

Chapter 9 of the "Understanding Deep Learning" [8] text book explains, how transfer learning and data augmentation can be powerful techniques to build a well performing model. Especially when training data is limited. At a first glance $10,800 = 18,000 \times 0.6$ training images seem like plenty of data. However, as explained in section 2.1, we only have about 15 images per class, if we encode the data as a classification problem. Therefore, we decided to **increase the training dataset by a factor of 15 and apply transfer learning on the VGG16 [9] architecture**. Transfer learning is the process of using an existing neural network that has been trained on some task A, freezing all but the last couple of layers and then re-training the model on the new task B (similar to A), while keep the weights from the frozen layers fixed.

Apart from the promise that transfer learning (TL) results in a well performing model, there are two more reasons why we decided to use TL. The first is that TL models are easier to implement instead of building an entire CNN architecture from scratch (tabula rasa). The second reason is that even if we had enough data, training a state of the art CNN is extremely computational expensive and takes a long time. Therefore, it is much faster and cheaper to just leverage the weights and feature maps from the state of the art network and then proceed to fine tune the last couple of layers.

Jason Brownlee, author of the popular tutorial website `machinelearningmastery.com` has a great tutorial on TL. In his tutorial "Transfer learning in Keras with computer vision models" [2] he presents the VGG16 and ResNet50 models. We also found another tutorial on Kaggle which showcases how to use VGG16 for TL on a clock images classification task

[1]. After reading both tutorials, we decided to implement TL using the VGG16 architecture because the classification results using VGG16 seemed promising. VGG16 [9] is the successor of the famous AlexNet CNN [7], and is conceptually speaking "just" a deep CNN. Both VGG16 and AlexNet work well if they are tuned properly. However, deep CNNs can lead to the "shattered gradient" problem, so for future work a residual network like ResNet50 might be another suitable choice for TL. However, for this assignment we decided to move forward with VGG16 as it is a deep, pre-trained CNN that is simple to implement, with a proven track-record. Exact implementation details such as how the data was augmented, which layers were frozen, regularization parameters, etc. will be mentioned in section 2.5.

2.5 Experimental Setup

To implement transfer learning (TL) using VGG16 on the clock image dataset, we utilized Google Colab[4], opting for a one-time purchase of 100 computing units for 11 Euros. This gave us access to powerful GPUs with up to 40GB of VRAM, which was essential for handling the data augmentation process.

For augmenting the training data, we used the `keras ImageDataGenerator` function, which applies random transformations such as rotation, shear, and zoom. Specifically, we configured the augmentation to allow rotations between 0 and 360 degrees, zooms between 0.9 and 1.1 times the current level, and shears from 0 to 15 degrees. To balance GPU usage and training efficacy, we chose to expand the training set by a factor of 15. This augmentation strategy effectively increased variability in the training data while staying within our VRAM budget. Google Colab charges more computing units per hour, given the GPU tier. A scale factor of 15, used up about 16 GB of VRAM which was the upper limit of the mid tier GPU called `L4 GPU`. Using the `L4 GPU`, training a single model took about 1 hour.

2.5.1 VGG16

VGG16, as its name indicates, consists of 16 learnable weight layers and takes RGB images of size 224x224 as input. To match this format, we converted our grayscale clock images into RGB by replicating the grayscale channel across all three color layers. However, resizing the clock images (150x150) was unnecessary because the VGG16 function call supports varying image sizes. We loaded VGG16 with weights pre-trained on the ImageNet dataset and **unfroze only the last two convolutional layers and removed the original output layer**. This approach allowed the network to learn clock-specific feature maps in the final two layers, while earlier layers retained their general feature extraction capabilities. We then added a fully connected layer with 520 neurons and ReLu activation function to generate the final output. This is consistent with the VGG16 architecture utilizes all 512 channels from the previous layers.

The output configuration varied based on the label encoding. For the 720-class classification case, we employed a softmax activation function with 720 outputs and trained the model using cross-entropy loss. For regression, the output layer had one neuron with ReLU activation and was trained using mean absolute error (MAE) loss. The ReLU activation function was chosen to ensure that outputs remained non-negative, aligning with the constraint that hours and minutes cannot be negative.

We considered two multi-head models: (class, reg) and (class, class). The (class, reg) model had 13 outputs, comprising one output for minute regression (ReLU activation) and 12 outputs for hour classification (softmax activation). The (class, class) model used $72 = 12 + 60$ outputs, with 12 classes for hours and 60 for minutes, both employing softmax activation functions. Upon realizing that (class, class) multi-head was our best performing model, we further fine tuned it, by adding one more fully connected layer with 254 neurons for the minutes output only. This means that the network is one layer deeper for the minutes than for the hours, but the final output layers remain the same as previously described (72 outputs, softmax activation). We did not evaluate (reg, class) or (reg, reg) models because single-head regression did not yield competitive CSD results, as we will discuss in section 2.6.

2.5.2 Training Process

All models were trained for **30 epochs with a batch size of 128 using the Adam optimizer** with default hyperparameters, except for the learning rate. We selected Adam based on preliminary experiments in task 1, which indicated that differences between optimizers were not significant for our task. A larger batch size was chosen to accelerate convergence, though we acknowledge it may compromise generalization. After several trials, we settled on a batch size of 128 as an effective balance.

We also implemented three implicit regularization techniques: **early stopping, decreasing learning rate and check-pointing**. We start with a learning rate of 10e-3. If the validation loss of the model does not improve for 3 epochs in a row, then we decrease the learning rate by a factor of 0.1, so from 10e-3 to 10e-4. Since this is an iterative rule, we set the lower bound for the learning rate to be 10e-6. Training was stopped if the validation loss failed to improve for 9

[4] `https://colab.google/`

consecutive epochs, and we retained the model weights corresponding to the epoch with the lowest recorded validation loss. This strategy ensured efficient use of computational resources and minimized the risk of overfitting.

Additionally, we used two explicit regularization techniques: **dropout and batch regularization**. We used both regularization techniques in the final fully connected network (the one with 512 neurons), placed both the dropout and batch normalization calls after the activation function and before the output layer. After some trials we set the drop out rate to 0.5, which is quite high but showed better generalization capabilities. Batch normalization helps prevent the vanishing gradient problem by normalizing the mean and variance of the output to a constant scalar value.

For the (class, class) multi head case, all techniques have been used as described above. The only differences are that we ran our final version of the (class, class) multi head model on a training set that has been augmented and scaled by a factor of 25, using the premium tier `A100` GPU. Then, we regularized the minute layer (dense layer with 256 neurons) using batch regularization and a drop out rate of 0.7, which is very high, but empirically lead to low validation loss.

2.6 Results & Discussion

Our results are shown in table 5. The **(class, class) multi-head model had the lowest CSD of 1.02 minutes** (or 1 min and 1 sec to be precise). This means that our model is almost as good as a human telling the time, as we would expect a human to misread the time only in extreme cases of shear and rotation, so a human's CSD is likely sub 1 minute. The fact that our model is practically at 1 min CSD indicates that it is a well performing model. The second best performing model was the 720-class classification model with a CSD of 2.88 min. Then, follows the (class, reg) multi-head model with a CSD of 7.4 minutes. The worst performing model was the regression model with a CSD of 14.52 minutes.

Table 5: Final Common Sense Difference (CSD)

Heads	**Model**	**CSD (in minutes)**
Single	Classification	2.88
Single	Regression	14.52
Multi	Hours Classification Minutes Regression	7.41
Multi	Hours Classification Minutes Classification	1.02

We believe that these results are valid, since the VGG16 model was originally built and trained to be an image classification model. Therefore, it makes sense to us that a TL version of VGG16 performs well on classification tasks. However, we expected the TL version of VGG16 to perform slightly worse than classification models but still comparably good. This could be because TL might not work well if the original task was classification but the new TL task is regression. It also interesting to note that the test set accuracy on the 720-class classification model was about 75%. The minute head of the (class, class) multi-head model also had about 75% test set accuracy. Therefore, we believe that 75% test set accuracy is the upper bound of the VGG16 architecture when considering the minute hand as a feature. For future work it might be interesting to consider more complex/deeper architectures such as ResNet to see if they do better at learning the minutes hand as feature. However, the hour head of the (class, class) multi-head model had a 99% accuracy which explains why the (class, class) multi-head model has the lowest CSD overall.

We fully recognize that it is very easy to overfit models to the test set, especially when given access to the entire dataset. Therefore, we think that a bootstrap CSD confidence interval would be the most robust way to quantify model performance. Bootstrapping involves resampling the dataset with replacement. Then the model is re-trained and tested on these resampled datasets, generating an empirical distribution of the CSD. This approach would allow us to assess the stability of the model's performance on unseen data. However, given the computationally heavy nature of deep learning though, we do not think that bootstrapping k datasets and fitting the model k times on them is practically feasible. While cross-validation (CV) is another alternative, it is typically performed on the training set, leaving the test set fixed. This does not address our main goal of assessing variability in test set performance.

To approximate a **CSD bootstrapping process**, we re-ran our best-performing (class, class) multi-head model twice with different seed numbers, effectively changing the training, validation, and test splits. For all previous tasks we have always used the default seed number of 42. However, for our "pseudo-bootstrap" run, we set the seed number arbitrarily to 424 and observed a CSD of 1.03 instead of 1.02. Although this is not as robust as full bootstrapping, it suggests that our model's performance is consistent across different data splits, reducing the likelihood that our results are only good due to a favorable data split. We did not re-run this experiment for more seed numbers due to computational reasons.

References

[1] Avantikab. *Time image classification with VGG16*. 2022. URL: `https://www.kaggle.com/code/avantikab/time-image-classification-with-vgg16/notebook`.

[2] Jason Brownlee. "Transfer learning in keras with computer vision models". In: *Machine learning mastery* (2019).

[3] Francois Chollet et al. *Keras*. 2015. URL: https://github.com/fchollet/keras.
[4] Aurélien Géron. *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow.* " O'Reilly Media, Inc.", 2022.
[5] Christophe Giraud. *Introduction to high-dimensional statistics*. Chapman and Hall/CRC, 2021.
[6] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. "Cifar-10 (canadian institute for advanced research)". In: *URL http://www. cs. toronto. edu/kriz/cifar. html* 5.4 (2010), p. 1.
[7] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "Imagenet classification with deep convolutional neural networks". In: *Advances in neural information processing systems* 25 (2012).
[8] Simon J.D. Prince. *Understanding Deep Learning*. The MIT Press, 2023. URL: http://udlbook.com.
[9] Karen Simonyan and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition". In: *arXiv preprint arXiv:1409.1556* (2014).
[10] Han Xiao, Kashif Rasul, and Roland Vollgraf. "Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms". In: *ArXiv* abs/1708.07747 (2017). URL: https://api.semanticscholar.org/CorpusID:702279.