Vietnam Posts and Telecommunications Institute of Technology
(PTIT)

Python 2nd Assignment
Image Classification

Students:	Trần, Vũ Tiến Minh	B23DCCE067
	Phạm, Quốc Hùng	B23DCVT188
Lecturer:	Dr. Kim, Ngọc Bách	
Class:	D23CQCE04-B	
Date:	June 3, 2025	

Introduction

This report is the justification and results for the assignment. The assignment focuses on building, training and testing two simple neural networks to classify the CIFAR-10 dataset: A three layers Multi-Layer Perceptron (MLP) and a three convolution layers Convolutional Neural Network (CNN).

The Assignment Tasks

These are the main tasks:

- Build a basic MLP (Multi-Layer Perceptron) neural network with 3 layers.
- Build a Convolutional Neural Network (CNN) with 3 convolution layers.
- Perform image classification using both neural networks, including training, validation, and testing.
- Plot learning curves.
- Plot confusion matrix.
- Use the PyTorch library.

Packages Dependencies

These packages are centered around working with the PyTorch library as the requirement has stated:

- `torch`: For building and training neural networks.
- `torchvision`: For getting datasets and image transformations.
- `scikit-learn`: For train/validation splitting and calculating confusion matrix.
- `numpy`: For numerical operations.
- `matplotlib`: For plotting learning curves, confusion matrices and others.

Contents

Report

1 CIFAR-10 Dataset

CIFAR-10 dataset is a well-known dataset for image classification exercise. It has 60,000, 32×32 pixels images in 10 classes, with 6,000 images per class. PyTorch library splits the dataset into 50,000 training images and 10,000 testing images.

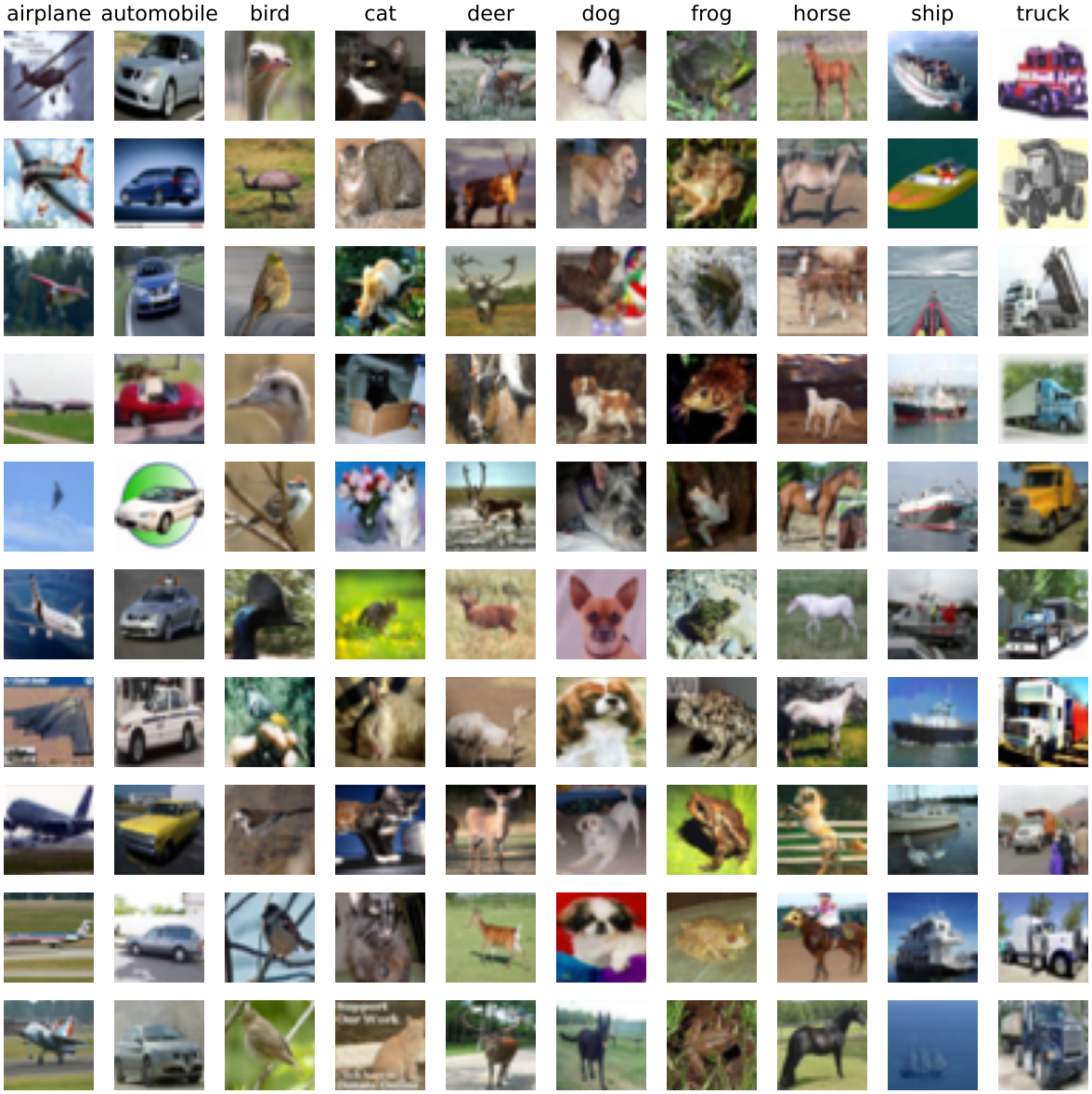

Figure 1: cifar10-images.pdf

In this project, to do training validation and testing seperately, the 50.000 images set is split into 40,000 training images and 10,000 validation images. This is a common method tune the model based on the validation set's results, and then testing the model with the test set to have an unbiased evaluation of the model.

2 Data Analysis

The first step of any Machine Learning exercise is to do Data Analysis. All analysis will be done on the 40,000 images train set only. This is to simulate the real world scenario, where the test set is not available during the training process. The validation set is used for tuning the model, so it is not used in the analysis.

2.1 Color Distribution

The below graph shows the RGB color distribution of the dataset, with the color intensity ranging from 0 to 255. Based on the graph, all three channels does follow a Gaussian distribution to some degree, minus the very high peak at the 255 region. The blue channel specifically is right skewed, so unlike other distribution with a mean around 128, blue has a mean around 100.

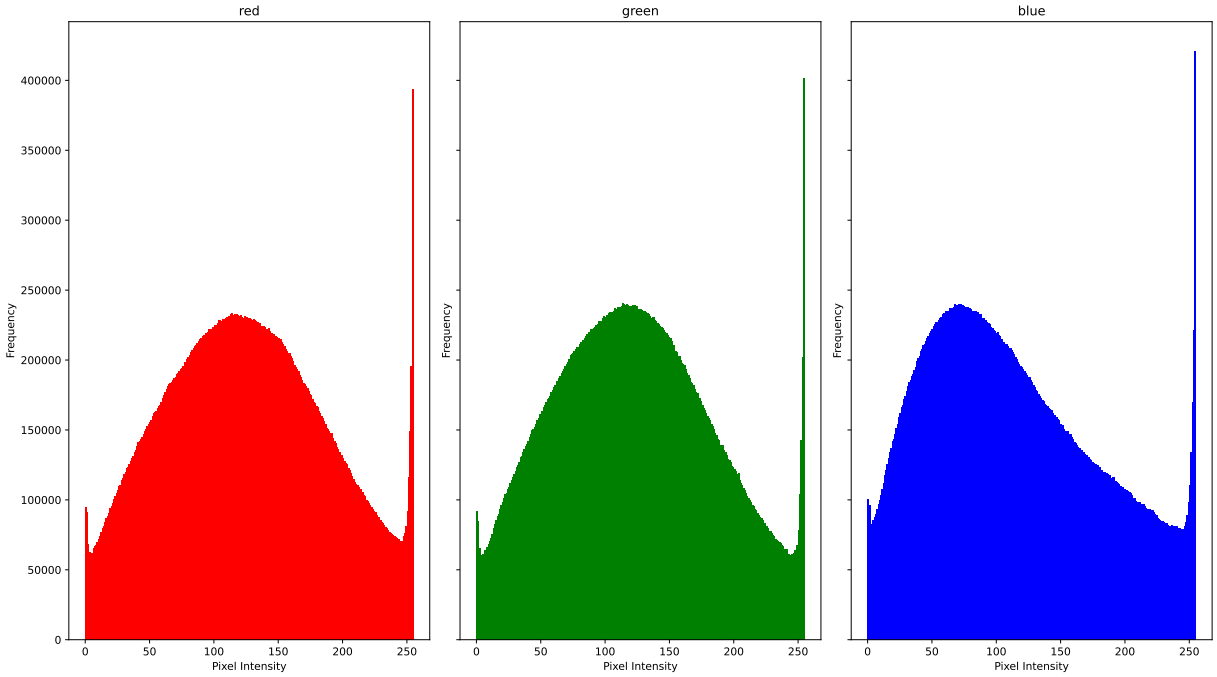

Figure 2: cifar10-channels.pdf

2.2 Mean And Standard Deviation

MLP and CNN can perform better on a normalized dataset, often resulting in a faster convergence and better accuracy. In PyTorch, `torchvision.transforms.Normalize` function normalizes images by adjusting their pixel values according to the mean and standard deviation of the dataset. So the next step is to find the mean and standard deviation. The table includes the raw values and also the normalized values divided by 255.

	Red	Green	Blue
Mean	125.36226708984375	123.04029643554688	113.9900185546875
STD	62.99172319884214	62.10269257704036	66.75451872125706
Normalized Mean	0.4916167336856618	0.48251096641390934	0.4470196806066176
Normalized STD	0.24702636548565543	0.24353997089035434	0.2617824263578708

Table 1: cifar10-mean-std.csv

2.3 Transform Dataset

Finally, all three datasets are transformed.`torchvision.transforms.ToTensor()` converts each image from shape of $32 \times 32 \times 3$ to a tensor with shape of $3 \times 32 \times 32$.

```
# python
from torchvision import transforms

transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize(mean, std)
])
```

3 Building Models

To keep the program simple, both models' hyper parameters are tuned by hand after each training/validation cycle until the performance is deemed high enough. Below are the structure of the two models.

3.1 Multi-Layer Perceptron

The MLP model used for this assignment consists of three fully connected linear layers:

```
# python
from torch import nn

relu = nn.ReLU()
dropout = nn.Dropout(0.2)

model = nn.Sequential(                  # (_, 3, 32, 32)
    nn.Flatten(),                       # (_, 3 * 32 * 32)

    nn.Linear(3 * 32 * 32, 512),        # (_, 512)
    nn.BatchNorm1d(512),
    relu,
    dropout,

    nn.Linear(512, 256),                # (_, 256)
    nn.BatchNorm1d(256),
    relu,
    dropout,

    nn.Linear(256, 10)                  # (_, 10)
)
```

1. **Flatten Layer:** Converts the input image into a 1D vector.
2. **First Linear Block:**
   - Fully connected layer with 512 output units.
   - Batch normalization for improved training stability.
   - ReLU activation for non-linearity.
   - Dropout 20% for regularization.
3. **Second Linear Block:**
   - Fully connected layer with 256 output units.
   - Batch normalization.
   - ReLU activation.
   - Dropout 20%.
4. **Output Linear Layer:** Fully connected layer mapping to 10 output classes.

3.2 Convolutional Neural Network

The CNN model used for this assignment consists of three convolution layers and one output linear layer:

```
# python
from torch import nn

relu = nn.ReLU()
maxpool = nn.MaxPool2d(2, stride=2)
dropout = nn.Dropout2d(0.2)

model = nn.Sequential(                  # (_, 3, 32, 32)
    nn.Conv2d(3, 128, 3, padding=1),    # (_, 128, 32, 32)
    nn.BatchNorm2d(128),
    relu,
    maxpool,                            # (_, 128, 16, 16)
    dropout,

    nn.Conv2d(128, 256, 3, padding=1),  # (_, 256, 16, 16)
    nn.BatchNorm2d(256),
    relu,
    maxpool,                            # (_, 256, 8, 8)
    dropout,

    nn.Conv2d(256, 512, 3, padding=1),  # (_, 512, 8, 8)
    nn.BatchNorm2d(512),
    relu,
    maxpool,                            # (_, 512, 4, 4)
    dropout,

    nn.Flatten(),                       # (_, 512 * 4 * 4)
    nn.Linear(512 * 4 * 4, 10)          # (_, 10)
)
```

1. **First Convolutional Block:**
   - 2D convolution: 3 input channels, 128 output channels, 3×3 kernel, padding=1. The padding=1 is for preserving the dimensions from 30×30 to 32×32.
   - Batch normalization.
   - ReLU activation.
   - 2×2 Max pooling reduces size to 16×16.
   - Dropout 20%.
2. **Second Convolutional Block:**
   - 2D convolution: 128 input to 256 output channels.
   - Batch normalization.
   - ReLU activation.
   - 2×2 Max pooling for size of 8×8.
   - Dropout 20%.
3. **Third Convolutional Block:**
   - 2D convolution: 256 input to 512 output channels.
   - Batch normalization.
   - ReLU activation.
   - 2×2 Max pooling for size of 4×4.
   - Dropout 20%.
4. **Output Linear Layer:**
   - Flatten layer converts feature maps to a vector size of $512 \times 4 \times 4$.
   - Fully connected linear layer mapping to 10 output classes.

4 Criterion and Optimizers

Now comes the most important part of the exercise: Optimize the optimizers. The two models use the same loss function, which is `torch.nn.CrossEntropyLoss`. For the optimizers, MLP uses `torch.optim.Adam` and CNN uses `torch.optim.SGD`. Similar to building the models, the below optimizers' parameters were manually tuned after each validation cycle.

4.1 Cross Entropy Loss

`torch.nn.CrossEntropyLoss` is a loss function commonly used for multi-class classification problems, making it suitable for CIFAR-10. It is a combination of `torch.nn.LogSoftmax`, an activation function, and `torch.nn.NLLLoss`, a loss function. `LogSoftmax` converts raw output scores into probabilities that sum to 1 and apply logarithm. `NLLLoss` takes the log-probabilities and measures how far off the predictions are from the true labels.

$$L(x, y) = -\sum_{i=1}^{C} y_i \cdot \log(x_i)$$

C:	Number of classes
x:	Predicted labels
y:	True labels

4.2 Adaptive Moment Estimation

`torch.optim.AdamW` is `torch.optim.Adam` with weight decay, which by itself is an optimization algorithm combining the advantages of two other popular optimizers `torch.optim.Adagrad` and `torch.optim.RMSprop`. Because of its adaptive nature, Adam works well with its default hyper parameters. The final weight decay is tuned to 10^{-3} to prevent MLP from over fitting too early on.

$$\begin{aligned}
m_t &= \beta_1 m_{t-1} + (1-\beta_1) g_t \\
v_t &= \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \\
\hat{m}_t &= \tfrac{m_t}{1-\beta_1^t} \\
\hat{v}_t &= \tfrac{v_t}{1-\beta_2^t} \\
\theta_t &= \theta_{t-1} - \eta \left(\tfrac{\hat{m}_t}{\sqrt{\hat{v}_t}+\epsilon} + \lambda \theta_{t-1} \right)
\end{aligned}$$

g_t:	Loss function's gradient at step t
m_t, v_t:	1st, 2nd moment estimates
$\hat{m}_t$, $\hat{v}_t$:	Bias-corrected of m_t, v_t
β_1, β_2:	Decay rates of m_t, v_t
θ_t:	Model's parameters
η:	Learning rate
λ:	Weight decay coefficient
ϵ:	Constant for numerical stability

4.3 Stochastic Gradient Descent

`torch.optim.SGD` is one of the most commonly used optimization algorithms. It updates CNN parameters using loss function's gradient. "Stochastic" means the gradients is from computing random mini-batch rather than the whole dataset, making the process faster and more memory-efficient. CNN is often paired with SGD + momentum over Adam for better generalization. Although SGD requires more tuning, the performance gains can be significant. The momentum helps escape local minima and accelerates progress in shallow valleys. In the project, the final tuned parameters are: learning rate = 0.01, momentum = 0.9 and weight delay = 10^{-3}.

$$\begin{aligned}
v_{t+1} &= \mu v_t - \eta g_t \\
g_t &= \nabla_\theta J(\theta_t) + \lambda \theta_t \\
\theta_{t+1} &= \theta_t + v_{t+1}
\end{aligned}$$

θ_t: Model's parameters at step t
η: Learning rate
λ: Weight decay coefficient
μ: Momentum factor
v_t: Velocity
g_t: Gradient

5 Training And Validation

5.1 The Method

With all the components ready, the training and validation process starts with spliting the two's dataset into smaller batches. Aside from reducing the hardware workload, smaller train batches leads to faster parameter updates and improve generalization. Batches too small can introduce noises, slowing down convergence, too large gives a smoother params updates but less generalization. The training batch size in the project is set to 64 after some tuning, 40,000 images is split into 625 batches. For the validation set the batch size is 1,000 as the dataset is used for tuning parameters, not training.

```
# python

model = ...
train_losses = []
train_accuracies = []
validation_losses = []
validation_accuracies = []

for epoch in range(20):
    train_loss, train_accuracy = train(model, train_loader)
    validation_loss, validation_accuracy = test(model, validation_loader)

    train_losses.extend(train_loss)
    train_accuracies.extend(train_accuracy)
    validation_losses.append(validation_loss)
    validation_accuracies.append(validation_accuracy)
```

Both models will train through the train dataset 20 times (20 epochs). The training loss and accuracy values is calculated after every 25 batches for a more detailed reference for parameters tuning. For validation values is calculated after each training cycle.

5.2 Learning Curve

Based on the output values above, plotting the learning curve graph can show some insights on how to tune the parameters:

Underfitting

- Training loss is high
- Validation loss is also high
- Accuracy is low for both

This means model is too simple or learning too slowly. Tuning by increase the epochs number or learning rate.

Overfitting

- Training loss is low, accuracy is high
- Validation loss is high, accuracy is low

This means model fits training data well, but doesn't generalize. In this case, increase the optimizer's weight delay or model's dropout parameter. Or maybe this is the best that the model can do, consider early stopping.

The below graphs are the learning curves of two models after some tuning. MLP validation loss and accuracy come to a hault at the 10th epoch with the validation accuracy around 55%. This is expected considering MLPs usually struggle on image data compared to CNN. On the other side, CNN performs exceptionally well with the validation accuracy at 80% and still rising slowly.

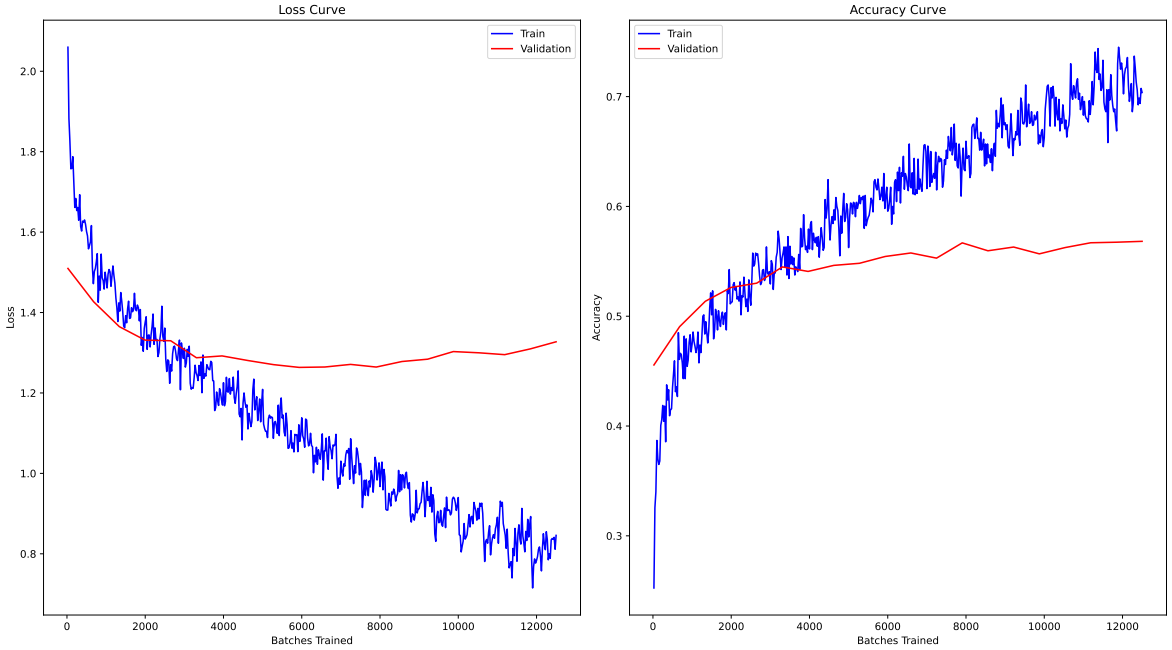

Figure 3: mlp-learning-curve.pdf

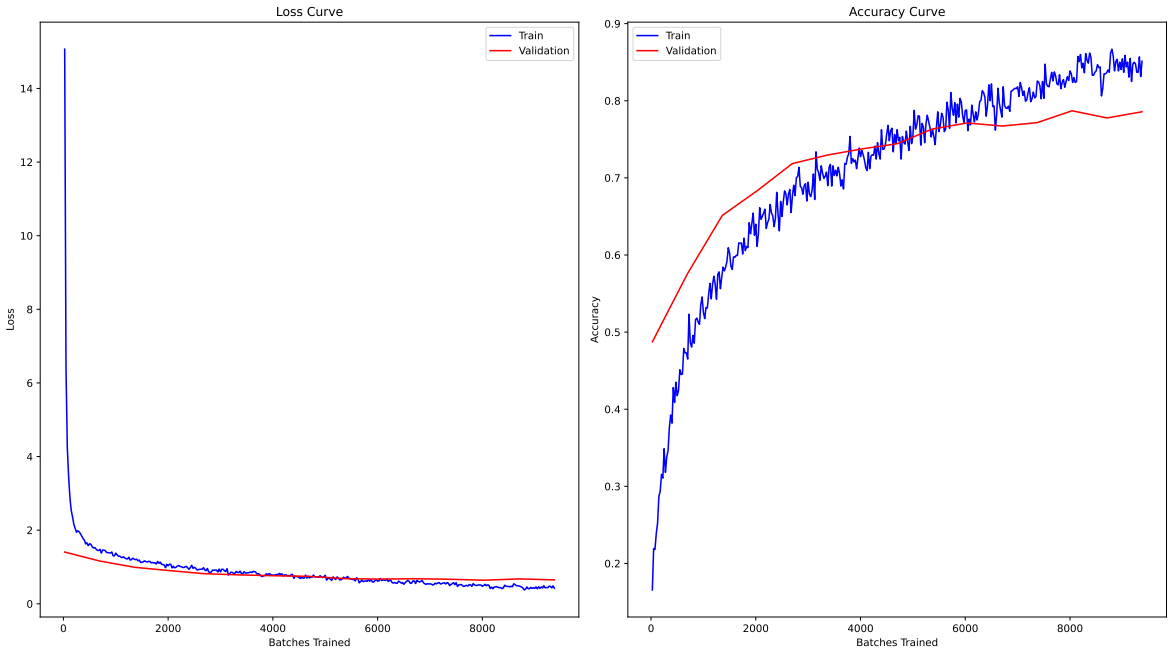

Figure 4: cnn-learning-curve.pdf

6 Testing Results

To evaluate the trained models without any bias, the final step is to run the models through unseen data, which is the testing dataset. Below are the losses, accuracies and confusion matrices. The loss and accuracy values are identical to the validation dataset, which implies that these two models are fitted for general images classification tasks. The Confusion matrices show more on where the hotspot of misprediction at. Both models find differentiating cats and dogs, automobiles and trucks. Unlike CNN, MLP has other hotspots on the cat/dog/deer/horse regions. Ship/airplane is another interesting hotspot

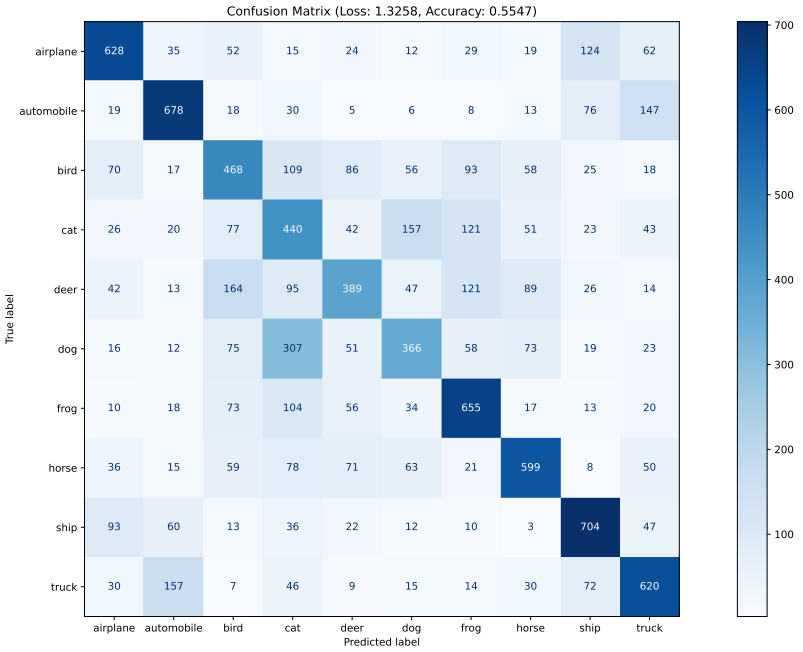

Figure 5: mlp-confusion-matrix.pdf

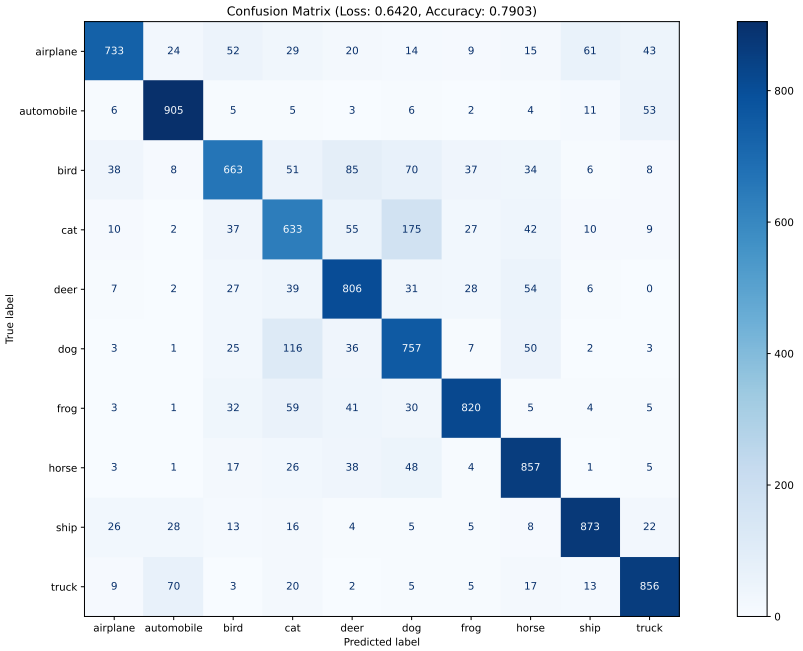

Figure 6: cnn-confusion-matrix.pdf

7 Discussion

The two models has an average preformance for any MLPs and CNNs. There are more improvements to be done on tuning the models and optimizer parameters and training/validation method.

One of the simplest way is to try a wide range of values, since the best parameters might be the least expected ones. This method is commonly known as Grid Search, a brute force method to validate each and every combinations. But considering the exhaustiveness and inefficient of this, another popular method is Random Search - randomly selects a combination. This strange but suprisingly effective method more often than not can actually find the best combination. Additionally, because of its lighter weight, more values can be fitted into the grid to try out even more combinations.

The next thing to consider is the training/validation method. Despite performing well in this project, splitting 20/training dataset solely for validating is not the best method. A much better one is doing Cross Validation, which is spliting the training set into k folds with equal sizes and in each iteration, one fold is picked for validation and $k-1$ other folds are for training. After finding the best parameters based on the validation results, all folds can be

Conclusion

Overall, this exercise is a fun way to get introduced into the world of Neural Network. Experimenting with architectures like MLP and CNN, loss functions is like a creative playground for solving real-world problems. Changing a hyperparameter, adjust the architecture, or try a new batch size and immediately see the impact feels like building a nonexistent machine to come to life. On the other side, it also shows the short comings of machine learning when it comes to non linear task like image classification. That's why many CAPTCHA - Completely Automated Public Turing Test to tell Computers and Humans Apart - tests use image classification to detect bots.