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 6, 2025	

Introduction

This report is the justification and the results of 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.

Package 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.

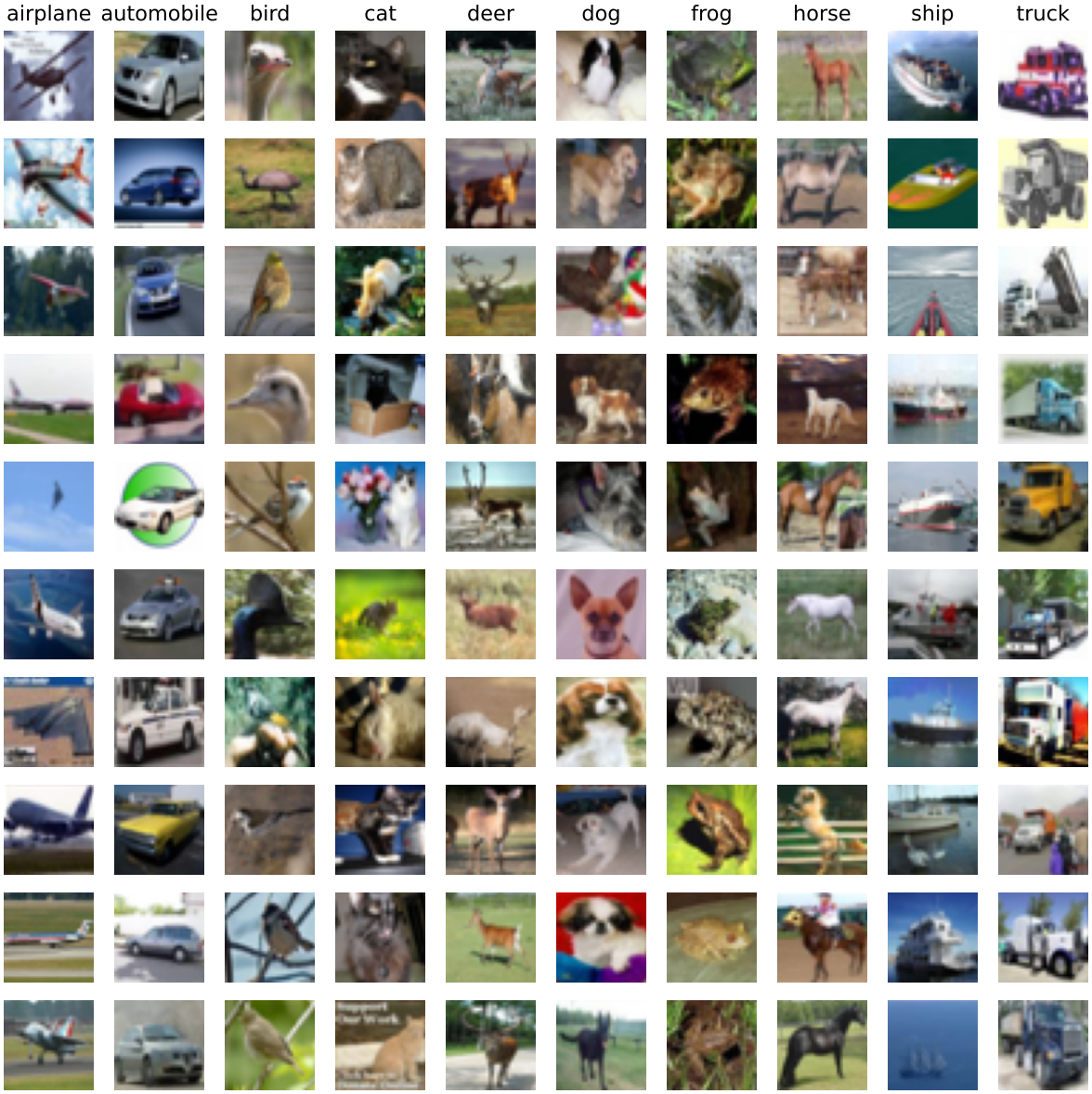

Figure 1: cifar10-images.pdf

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 of tuning the model based on the validation set's results and then testing the model with the test set to have an unbiased evaluation.

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 solely used for tuning the model, so it is not used in the analysis.

2.1 Color Distribution

The below graphs show 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 balanced distribution to some degree, minus the very high peak at the 250 region. The blue channel specifically is right skewed, so unlike other distribution with a mean around 128, blue has a mean around 100.

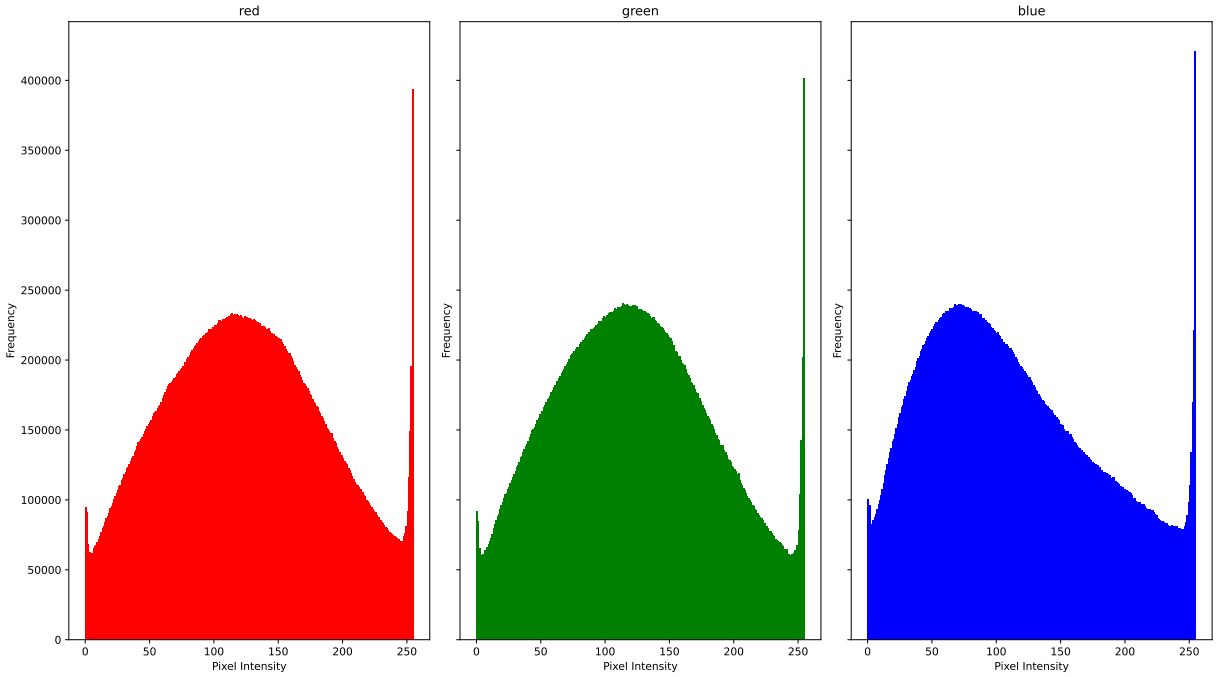

Figure 2: cifar10-channels.pdf

2.2 Mean And Standard Deviation

MLP and CNN can perform better on a normalized dataset, resulting in a faster convergence and better accuracy. In PyTorch, the `torchvision.transforms.Normalize` function normalizes images by adjusting their pixel values according to the mean and standard deviation of the dataset. The table below 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 Transforming The Dataset

Before normalization, all three datasets are converted with `torchvision.transforms.ToTensor`, each image is reshaped from a dimension of $32 \times 32 \times 3$ to $3 \times 32 \times 32$.

```
# python
from torchvision import transforms

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

3 Building Models

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

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

    nn.Linear(3 * 32 * 32, 512),        # (_, 512)
    nn.BatchNorm1d(512),
    nn.ReLU(),
    nn.Dropout(0.4),

    nn.Linear(512, 256),                # (_, 256)
    nn.BatchNorm1d(256),
    nn.ReLU(),
    nn.Dropout(0.4),

    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 40% for regularization.
3. **Second Linear Block:**
   - Fully connected layer with 256 output units.
   - Batch normalization.
   - ReLU activation.
   - Dropout 40%.
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

model = nn.Sequential(                  # (_, 3, 32, 32)
    nn.Conv2d(3, 32, 3, padding=1),     # (_, 32, 32, 32)
    nn.BatchNorm2d(32),
    nn.ReLU(),
    nn.Dropout2d(0.1),
    nn.MaxPool2d(2, stride=2),          # (_, 32, 16, 16)

    nn.Conv2d(32, 64, 3, padding=1),    # (_, 64, 16, 16)
    nn.BatchNorm2d(64),
    nn.ReLU(),
    nn.Dropout2d(0.1),
    nn.MaxPool2d(2, stride=2),          # (_, 64, 8, 8)

    nn.Conv2d(64, 128, 3, padding=1),   # (_, 128, 8, 8)
    nn.BatchNorm2d(128),
    nn.ReLU(),
    nn.Dropout2d(0.1),
    nn.MaxPool2d(2, stride=2),          # (_, 128, 4, 4)

    nn.Flatten(),                       # (_, 128 * 4 * 4)
    nn.Dropout(0.4),
    nn.Linear(128 * 4 * 4, 10)          # (_, 10)
)
```

1. **First Convolutional Block:**
    - 2D convolution: 3 input channels, 32 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.
    - Dropout 10%.
    - 2×2 Max pooling reduces size to 16×16.
2. **Second Convolutional Block:**
    - 2D convolution: 32 input to 64 output channels.
    - Batch normalization.
    - ReLU activation.
    - Dropout 10%.
    - 2×2 Max pooling reduces size to 8×8.
3. **Third Convolutional Block:**
    - 2D convolution: 64 input to 128 output channels.
    - Batch normalization.
    - ReLU activation.
    - Dropout 10%.
    - 2×2 Max pooling reduces size to 4×4.
4. **Output Linear Layer:**
    - Flatten layer converts feature maps to a vector size of $128 \times 4 \times 4$.
    - Dropout 40%.
    - Fully connected linear layer mapping to 10 output classes.

4 Criterion and Optimizer

Now comes the most important part of the exercise: Optimize the optimizer. The two models use the same loss function `torch.nn.CrossEntropyLoss`. For the optimizer, MLP uses `torch.optim.AdamW` and CNN uses `torch.optim.SGD`. Similar to building the models, the 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 then 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.Adam` is a popular optimization algorithm combining the advantages of two other optimizers: `torch.optim.Adagrad` and `torch.optim.RMSprop`. Because of its adaptive nature, Adam works well with its default hyper parameters, which the project is using.

```
# python
from torch.optim import Adam

Adam(
    mlp.parameters(),
    lr=0.001,
    weight_decay=1e-6,
    betas=(0.9, 0.999),
    eps=1e-8
)
```

Adam formulas:

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

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
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 for better generalization. Although SGD requires more tuning, the performance gains can be significant. The momentum factor helps escape local minima and accelerates progress in shallow valleys. The weight decay is to prevent overfitting.

```
# python
from torch.optim import SGD

# final tuned SGD
SGD(
    cnn.parameters(),
    lr=0.01,
    momentum=0.9,
    weight_decay=1e-4
)
```

SGD formulas:

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

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

4.4 Reduce Learning Rate On Plateau

`torch.optim.lr_scheduler.ReduceLROnPlateau` is a learning rate scheduler, used by both model to reduce the learning rate when a metric has stopped improving. In this project, The scheduler will multiply the learning rate by 0.1 every 3 epochs of stalled validation loss.

```
#python
from torch.optim.lr_scheduler import ReduceLROnPlateau

ReduceLROnPlateau(
    optimizer,
    mode='min',
    factor=0.1,
    patience=3
)
```

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 hardware workload, smaller train batches leads to faster parameter updates and improve generalization. Batches too small can introduce noises, slowing down convergence while too large gives a smoother params updates but less generalization.

The training batch size in the project is set to 64 after some tuning, that is a total of 625 batches from 40,000 images. For the validation set, the batch size is 1,000 because the dataset is used for tuning parameters, not training. Both models goes over the train dataset 40 times (40 epochs), this is to get a wider view of the learning curve. Training/validation loss and accuracy calculated after every training cycle.

```
# python
import torch

def train(net, loader, optimizer, criterion):
    net.train()
    running_loss = corrects = 0

    for images, targets in loader:
        optimizer.zero_grad()
```

```
            predicts = net(images)
            loss = criterion(predicts, targets)
            loss.backward()
            optimizer.step()

            running_loss += loss
            corrects += (predicts.max(1)[1]  == targets).sum()

        loss = running_loss.item() / len(loader)
        accuracy = corrects.item() / len(loader.dataset)
        return loss, accuracy

    def test(net, loader, criterion):
        net.eval()
        running_loss = corrects = 0

        with torch.no_grad():
            for images, targets in loader:
                predicts = net(images)
                loss = self.criterion(predicts, targets)

                running_loss += loss
                corrects += (predicts.max(1)[1] == targets).sum()

        loss = running_loss.item() / len(loader)
        accuracy = corrects.item() / len(loader.dataset)
        return loss, accuracy

    for epoch in range(40):
        train_loss, train_accuracy = train(net, train_loader, criterion, optimizer)
        validation_loss, validation_accuracy = test(net, validation_loader, criterion)
        scheduler.step(validation_loss)
```

5.2 Learning Curve

Based on the validation values collected above, a learning curve graph can be plotted to show some insights on how well tuned are the parameters.

Underfitting

This means model is learning too slowly. In this case, increase the number of epochs or the learning rate factor.

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

Overfitting

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

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

The below graphs are the learning curves of two models after some tuning. MLP shows signs of overfitting, with loss value around 1.2 and accuracy around 58%. This is expected considering MLPs usually struggle on image data compared to CNN. On the other side, CNN performs exceptionally well with loss value around 0.56 and accuracy around 81%. It's also interesting that the moment ReduceLROnPlateau activates is clearly visible in both graphs.

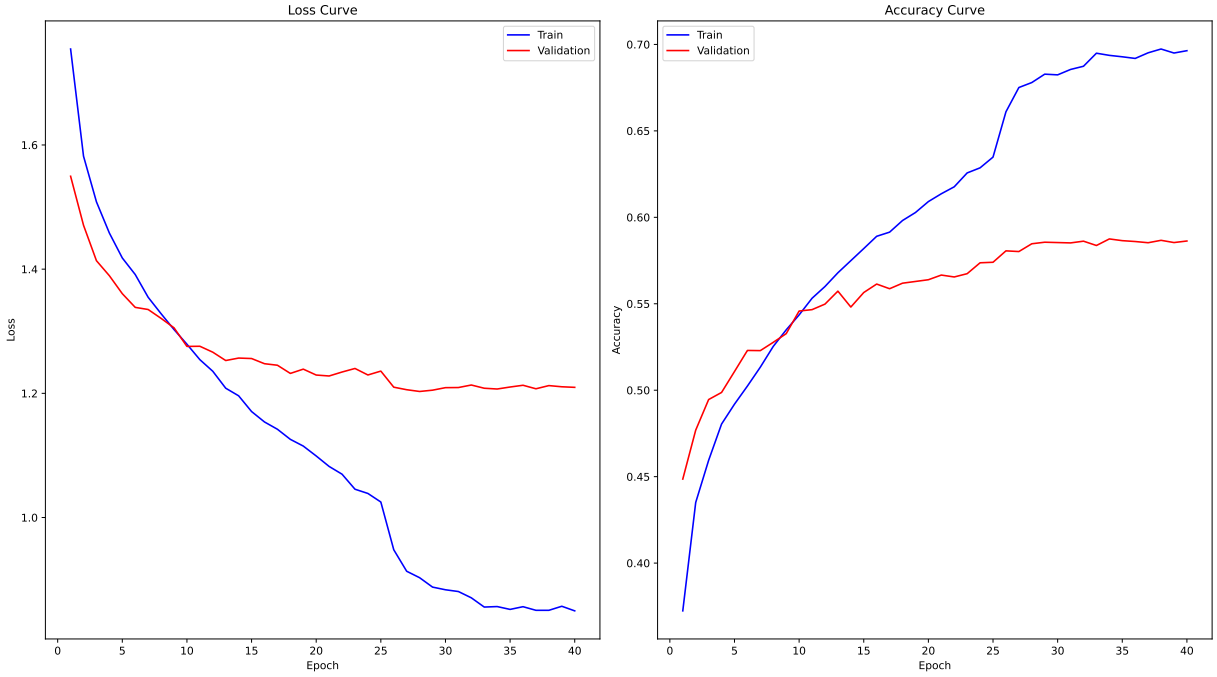

Figure 3: mlp-learning-curve.pdf

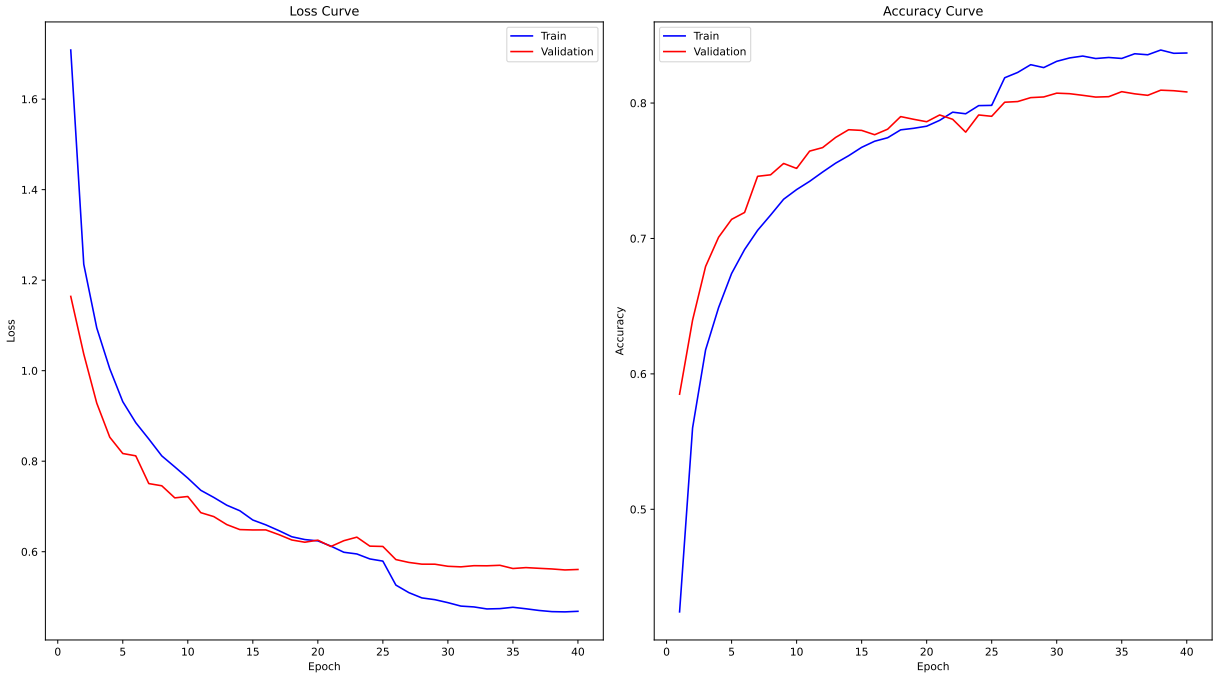

Figure 4: cnn-learning-curve.pdf

6 Testing Results

To evaluate the two 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 the validation method is well fitted for this task. The Confusion matrices help visualize where are the hotspots of misprediction at. Both models find difficulties differentiating between cats and dogs, automobiles and trucks, ships and airplanes. Unlike CNN, MLP hotspots spread out more inside the two class groups: the animal classes and the vehicle group. However, MLP recognises well if the data is an animal or a vehicle.

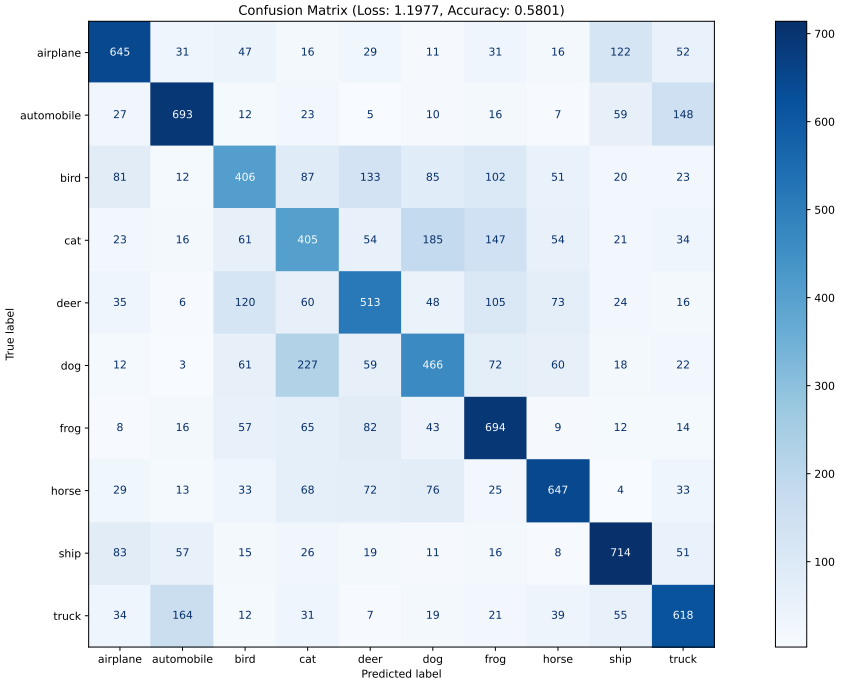

Figure 5: mlp-confusion-matrix.pdf

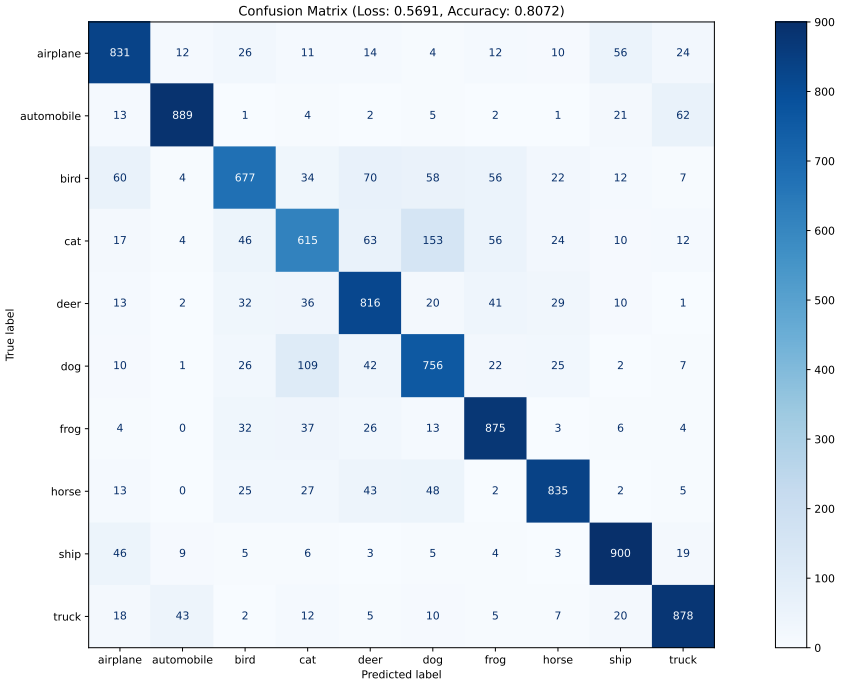

Figure 6: cnn-confusion-matrix.pdf

7 Discussion

The two models have average preformance as any other three layers MLPs and CNNs. There are improvements to be made on tuning the models and optimizer parameters, as well as the training and 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% of the training set solely for validation is not the best call. 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 used for training. After finding the optimal parameters based on the validation results, all folds can be used for training.

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 and optimizers 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.

Reference Documents

https://docs.pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html

https://www.geeksforgeeks.org/train-a-deep-learning-model-with-pytorch/

https://www.geeksforgeeks.org/building-a-convolutional-neural-network-using-pytorch/

https://www.cs.toronto.edu/~lczhang/360/lec/w05/overfit.html

https://www.youtube.com/watch?v=d6iQrh2TK98