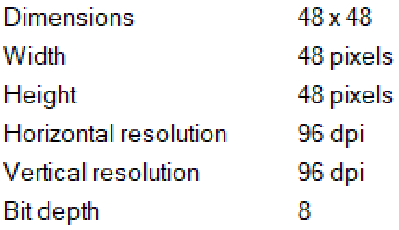
MCTA 4363 Deep Learning

Assignment 1: Deep Learning for Image classification

Name: Ahmad Irfan bin Kahar **Matric No.:** 2027127

Question 1

There are thousands of datasets available on the Kaggle website. The one that caught my interest is titled “Face expression recognition dataset”. This dataset is interesting because I think how fascinating that just from the expression of someone’s face expresses a lot about that person from what is on their mind, feelings and mood. The dataset is divided into two, train and test with each consists of seven (7) common expressions such as angry, disgust, fear, happy, neutral, sad and surprise. The size of the dataset is good providing a variety of images for each class of expression. For training, the size of each class ranges from 400 until 7000 images while for testing it ranges from 100 to 1800. The dataset covers all aspects of human facial expression and gives diverse classes of expression. The dataset supplies thousands black and white images with the size of 48x48 pixels with 96x96 dots per inch (dpi) for all the classes. The image specifications are shown below in figure 1.



Dimensions	48 x 48
Width	48 pixels
Height	48 pixels
Horizontal resolution	96 dpi
Vertical resolution	96 dpi
Bit depth	8

Figure 1: Image specifications

Based on the information received above, the datasets are deemed suitable for this assignment as it has a good dataset size with thousands of images and classed into a variety of classes covering all common human facial expressions. The image specifications are also easy to work with.

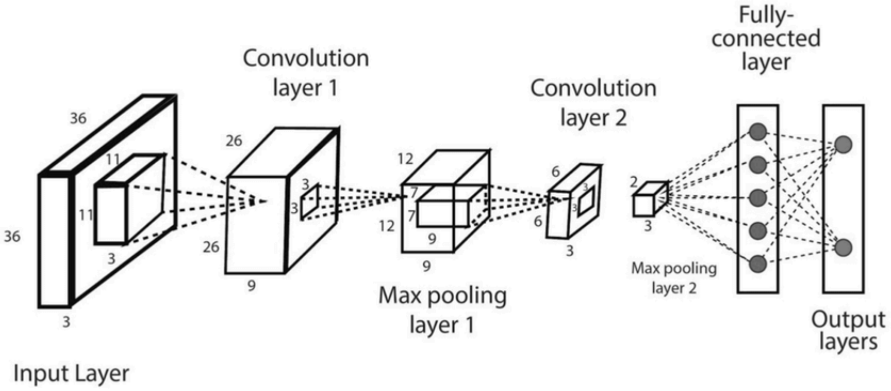
Question 2

After selecting the dataset, the next step is constructing the Convolutional Neural Network (CNN). For this dataset, the CNN architecture consists of three (3) convolutional layers, three (3) fully connected layers and an output of seven (7) classes for each expression.

The first layer of the convolutional layer has three (3) channel inputs from the Red, Green and Blue (RGB) colour scheme, 32 channel outputs and a kernel size of 5x5. Then, the second layer takes 32 channel inputs and 64 channel outputs with 5x5 kernel size. The last convolutional layer takes 64 channel inputs and gives 128 channel outputs with the same 5x5 kernel size. Each of these layers will undergo the ReLU activation function and max pooling of 2x2 size.

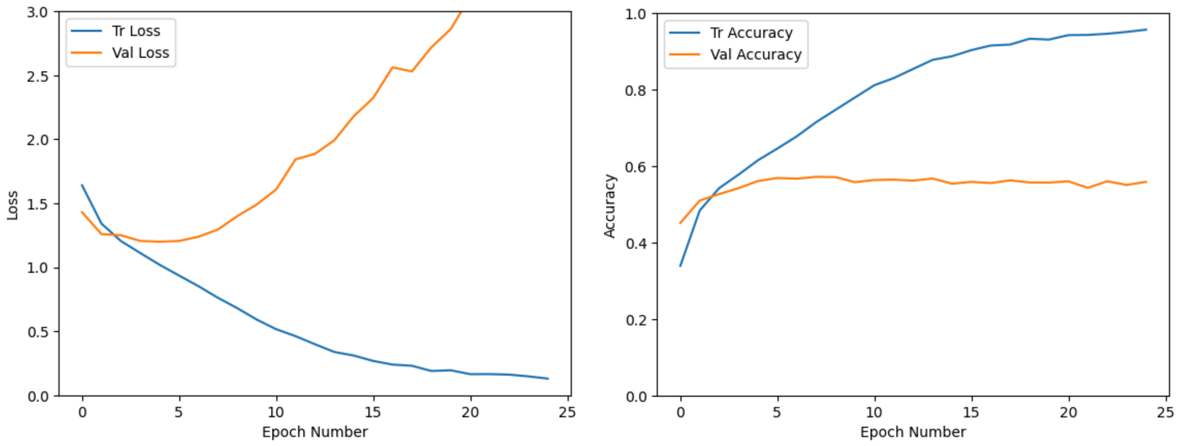
After the convolutional layers, the output of the last layer is fed into the first fully connected layer. The output of the last convolutional layers is calculated using a CNN shape calculator available online and the final size of the feature maps before the first fully connected layer is 2x2x128 (Width x Height x Channels) which gives an input size of 512. The first fully connected layers receive the 512 inputs and output to a 256 nodes supplied to the input of the second fully connected layer. The output from the second fully connected layer is connected to the input of the third fully connected layer with an input of 128. The output from the third fully connected layer is the 7 output for each expression class.

The full CNN architecture can be visualised from the image below with one addition of a convolutional layer and two more fully connected layers.



Max pooling is used after each convolutional layer to reduce the dimensionality of the feature maps and only capture the most significant features. Then each layer is subjected to ReLU activation function due to its efficiency and effectiveness in preventing vanishing gradient problems. For the training, the loss is calculated using cross entropy loss function and Adam optimizer. Adam optimizer is chosen because it is capable of adapting the learning rate and converges faster compared to the Stochastic Gradient Descent (SGD).

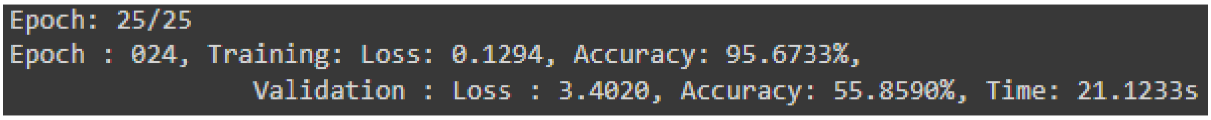
After training the model, the loss and accuracy curve are plotted and shown below.



Loss Curve

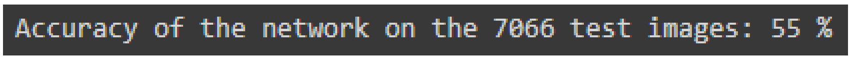
Accuracy curve

From the curve above, we can see that the training model’s accuracy is good, achieving a 95.67% accuracy at the last epoch. On the other hand, the testing datasets accuracy is underwhelming with only achieving an accuracy of 55%.



```
Epoch: 25/25
Epoch : 024, Training: Loss: 0.1294, Accuracy: 95.6733%,
                Validation : Loss : 3.4020, Accuracy: 55.8590%, Time: 21.1233s
```

Training model’s accuracy



```
Accuracy of the network on the 7066 test images: 55 %
```

Testing model’s accuracy

From the loss curve, we can observe a good decreasing trend on the training loss but a poor performance on the validation loss that shows an upward trend. For the accuracy curve, the training accuracy displays a good upward trend that achieves a 95.7% accuracy at the end of the last epoch. The validation accuracy shows a slight increase in the first few epochs but then remains stagnant until the 25th epoch.

Code (important parts)

```
# Image transformation changing to tensor and normalize using mean and
standard deviation from imageNet dataset
data_transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize([0.485, 0.456, 0.406],
                              [0.229, 0.224, 0.225])])
```

```
# Turn train and test Datasets into DataLoaders
from torch.utils.data import DataLoader

# Create iterators for the Data loaded using DataLoader module
batchSize = 64

trainloader = DataLoader(dataset=train_data, batch_size=batchSize,
num_workers=1, shuffle=True)
```

```
testloader = DataLoader(dataset=test_data, batch_size=batchSize, num_workers=1, shuffle=False)

train_data_size = len(trainloader.dataset)
test_data_size = len(testloader.dataset)

print(train_data_size)
print(test_data_size)
```

```
# Defining the FacialExpressionCNN
class FacialExpressionCNN(nn.Module):
    def __init__(self):
        super(FacialExpressionCNN, self).__init__()

        # 1st Convolutional Layer with 3 channel input, 32 channel output and 5x5 kernel size
        self.conv1 = nn.Conv2d(3, 32, 5)

        # 2nd Convolutional Layer with 32 channel input, 64 channel output and 5x5 kernel size
        self.conv2 = nn.Conv2d(32, 64, 5)

        # 3rd Convolutional Layer with 64 channel input, 128 channel output and 5x5 kernel size
        self.conv3 = nn.Conv2d(64, 128, 5)

        # 1st Fully Connected Layer with 512 inputs and 256 output
        self.fc1 = nn.Linear(128 * 2 * 2, 256)

        # 2nd Fully Connected Layer with 256 inputs and 128 output
        self.fc2 = nn.Linear(256, 128)

        # 3rd Fully Connected Layer with 128 inputs and 7 output (because 7 total classes)
        self.fc3 = nn.Linear(128, 7)

        # Max pooling function with kernel_size = 2, stride = 2
        self.pool = nn.MaxPool2d(2, 2)

        # ReLU activation function for each layers
        self.relu = nn.ReLU()
```

```python
    # Each Convolutional Layers will go through ReLU activation function and max pooling
    def forward(self, x):
        x = self.pool(self.relu(self.conv1(x)))
        x = self.pool(self.relu(self.conv2(x)))
        x = self.pool(self.relu(self.conv3(x)))
        x = x.view(-1, 128 * 2 * 2)
        x = self.relu(self.fc1(x))
        x = self.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

```python
# Initializing the model using the CNN built above
model = FacialExpressionCNN()

# Initializing the loss and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Moving the model to GPU
device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')
model.to(device)

# Printing the model summary
print(model)
```

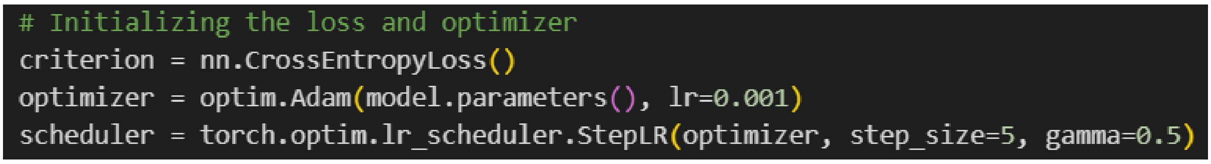
Question 3

Based on the loss curve achieved from CNN model in question 2, it can be observed that the model is showing signs of overfitting where the validation loss curve increases with the increase in number of epochs while the training loss curve on the other hand decreases significantly. Due to this, there is a huge gap between both curves indicating overfitting. This could be due to several reasons where the first one might be due to lack of data augmentation. Data augmentation can increase the training data diversity with techniques such as rotating, flips or crops. Other than that, it could be due to the high amount of training epochs which causes the model to memorise the training data. This can be mitigated by doing early stopping. The imbalance in the data representation could also cause overfitting. Certain classes might be underrepresented and cause the model to overfit the majority or bigger classes. Lastly, it could be due to the complexity of the model architecture where the model might have too many layers or parameters.

Other than the actions suggested above, we can do dropout technique in the CNN model to also reduce overfitting. Then we could also simplify the model by reducing the number of layers. Balancing the dataset for each class could also reduce overfitting.

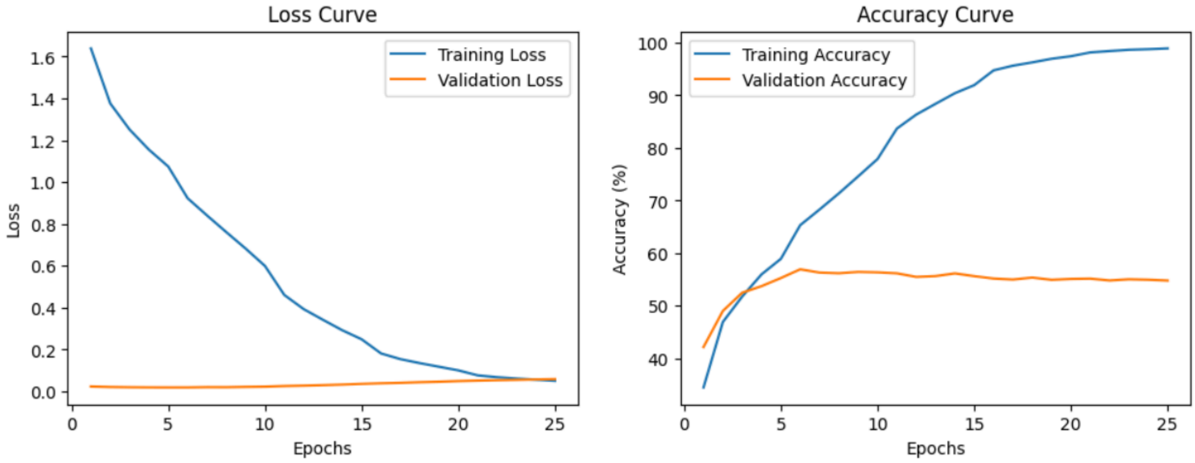
Question 4

The first learning rate scheduling technique used on the previous model is step decay. The code from the previous model is kept the same with a minor change in a few parts shown below.

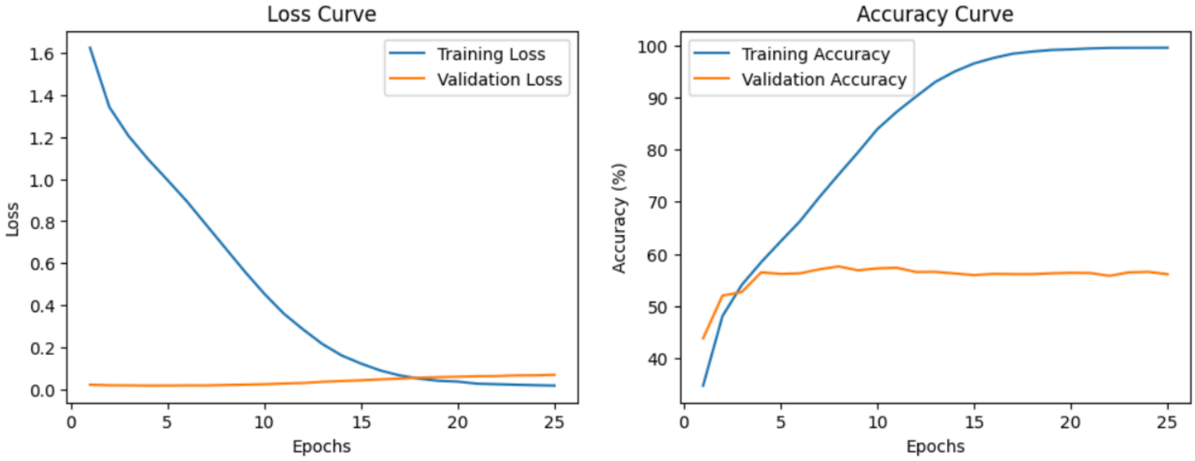


```
# Initializing the loss and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=5, gamma=0.5)
```

For each 5 epochs, the learning rate will be reduced by 0.5 multiplication. The loss and accuracy curve are shown below.



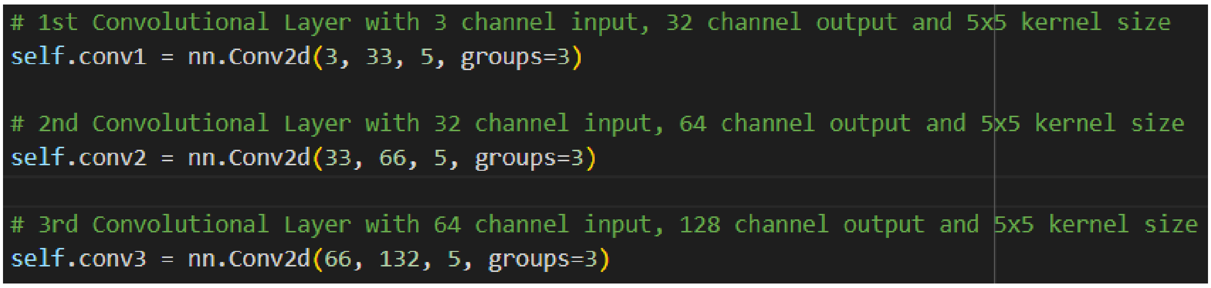
The second learning rate scheduling technique used on the previous model is exponential decay. The code is kept the same with a few changes shown below.



The performance of a model using different decay techniques may be influenced by several reasons. First one could be due to the complexity and Size of the datasets. Large datasets may require more gradual learning rate decay to allow the model to explore the parameter space thoroughly. Exponential decay can help by continuously reducing the learning rate, whereas step decay might be too abrupt. Other than that, it could be influenced by the initial Learning Rate. If the initial rate is high, a more aggressive decay for example step decay might be necessary to quickly bring it down to a stable value. Conversely, a low initial rate might pair better with exponential or polynomial decay. Lastly, the number of epochs planned for training can also impact the choice of decay. For long training schedules, exponential decay ensures the learning rate keeps reducing, thus avoiding overfitting. For shorter schedules, step decay might be sufficient to achieve convergence quickly.

Question 5

Now in this question we will experiment with different convolution methods available out there. The first one that will be tested on the model is a grouped convolution. The code is the same with only a small tweak at the CNN architecture. The change in the code is shown below.

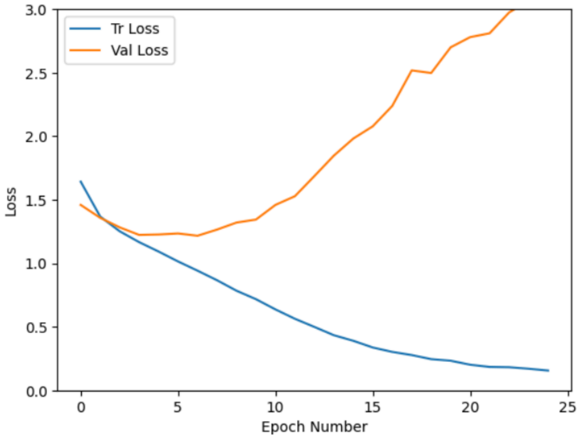


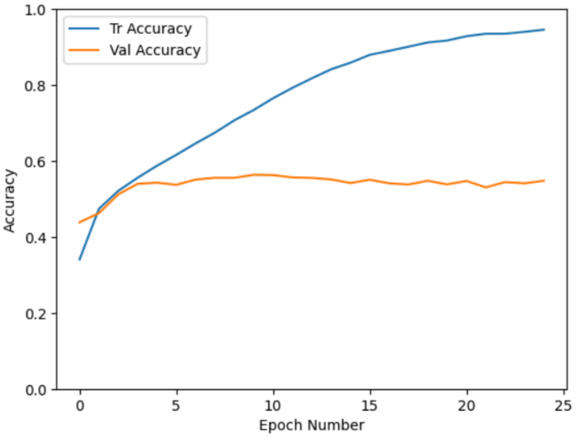
```
# 1st Convolutional Layer with 3 channel input, 32 channel output and 5x5 kernel size
self.conv1 = nn.Conv2d(3, 33, 5, groups=3)

# 2nd Convolutional Layer with 32 channel input, 64 channel output and 5x5 kernel size
self.conv2 = nn.Conv2d(33, 66, 5, groups=3)

# 3rd Convolutional Layer with 64 channel input, 128 channel output and 5x5 kernel size
self.conv3 = nn.Conv2d(66, 132, 5, groups=3)
```

The difference is the addition of groups in the initialization. The default for Conv2d is grouping of 1. Other than that, to match the group size, the input and output channels are changed to be divisible by three. The loss and accuracy curve of the model is shown below.





Question 6

In my opinion, the best way to get better performance and accuracy is to do transfer learning. Transfer learning is a popular method utilizing the famous pre-trained model available out there. Using a proven model will negate the inexperienced error in building a CNN architecture model from scratch. Pre-trained models ranging from AlexNet, ResNet, MobileNet and many more. Other than that, we can use ensemble methods where the predictions consist of a combination of multiple models. The average prediction between multiple models will be taken as the final prediction. We can also use simple techniques like data augmentation, batch normalization and regularization such as dropout. Data augmentation allows to increase the variety of the training image dataset by doing various effects like flipping, greyscale or rotation. Batch normalization will normalize the input of each layer to increase training stability of the model. Lastly, dropout technique prevents overfitting by randomly setting a fraction of input units to 0 at each update during training.