Facial Emotion Recognition

with

Data Augmentation

Kevin O'Shea ***A00304379***

Contents

1. Data Exploration

Data was obtained from Kaggle [1]. As visible in figure 1a, the dataset consists of a total of 28,709 images ('pixels') with associated target labels ('emotion').

```
[3] df = pd.read_csv('/content/drive/MyDrive/Colab Notebooks/AML Assignment 02/train.csv')
    print(df.shape)
    df.head()
```

```
(28709, 2)
```

	emotion	pixels
0	0	70 80 82 72 58 58 60 63 54 58 60 48 89 115 121...
1	0	151 150 147 155 148 133 111 140 170 174 182 15...
2	2	231 212 156 164 174 138 161 173 182 200 106 38...
3	4	24 32 36 30 32 23 19 20 30 41 21 22 32 34 21 1...
4	6	4 0 0 0 0 0 0 0 0 0 0 0 0 3 15 23 28 48 50 58 84...

Figure 1a - Shape and head of the dataset

We have seven emotions in total. They are:

- Angry
- Disgust
- Fear
- Happy
- Sad
- Surprise
- Neutral

To begin with, some images were generated to view the data in an image form. Each image is in a uniform format (48x48 pixels) and greyscale. Figure 1b and 1c demonstrate some emotions portrayed in the images.

Figure 1b - 'neutral' expression

Figure 1c - 'happy' expression

As figure 1d shows, not all labels in the dataset are unambiguous. Figure 1c shows an image which is meant to represent 'sad' but instead, with the wide eyes and mouth agape, looks closer to 'fear' or 'surprise'.

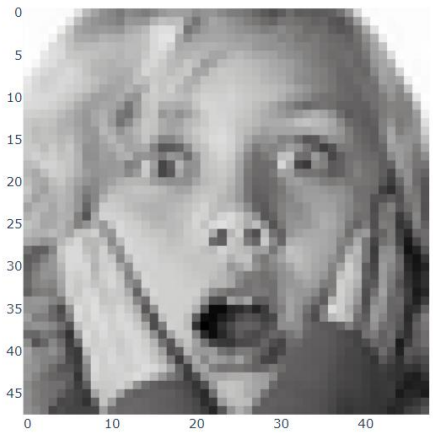

Figure 1d - 'sad' expression

Examining the distribution we can see that it is not even (fig 1e). For the size of this dataset, the mean value is 4101, which means that 2 emotions (surprise and disgust) are falling below this value. In particular, 'disgust' is considerably underrepresented with only 440 images. Figure 1e shows the distribution as a bar chart.

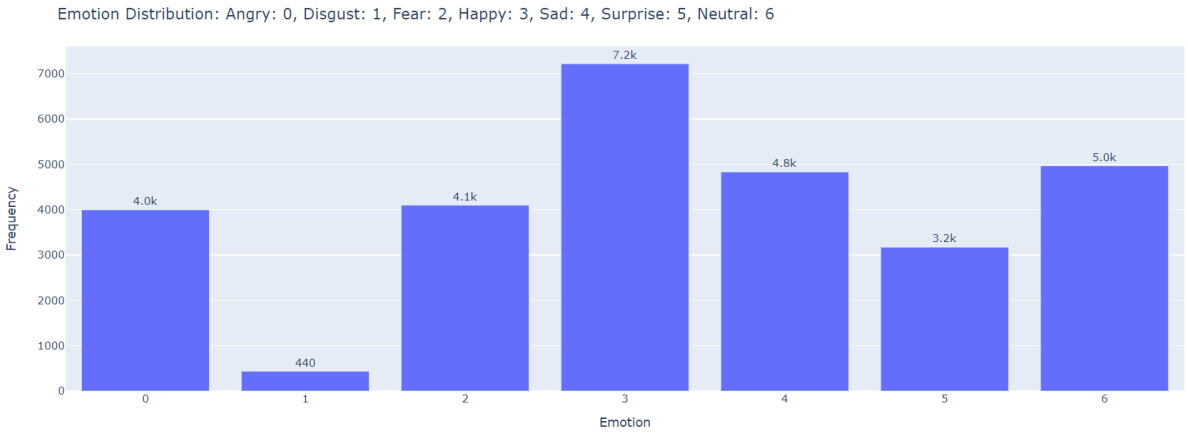

Figure 1e – Emotion distribution

As one of the emotion categories is badly underrepresented, it will be important to use data augmentation techniques pre-training in order to increase the understanding of this feature and improve the model performance.

2. Data Preprocessing

Data is scaled to between 0 and 1 using 'torchvision.transforms' (fig 2a).

```
# Define the transformations to be applied to your dataset
transform = T.Compose([
    T.ToPILImage(),
    T.ToTensor(),
    T.Normalize(mean=[0.5], std=[0.5 * 2]),
    T.Lambda(lambda x: torch.mul(x, 0.5) + 0.5)
])
```

```
train_dataset[0][0]
```

```
tensor([[[0.7500, 0.6559, 0.3382,  ..., 0.5480, 0.5755, 0.5696],
         [0.7500, 0.4990, 0.2990,  ..., 0.5147, 0.5853, 0.5735],
         [0.6951, 0.3794, 0.3147,  ..., 0.4559, 0.5814, 0.5775],
         ...,
         [0.3382, 0.3363, 0.3324,  ..., 0.3127, 0.3186, 0.3304],
         [0.3422, 0.3402, 0.3343,  ..., 0.2990, 0.3029, 0.3167],
         [0.3245, 0.3324, 0.3304,  ..., 0.2931, 0.2912, 0.2990]]])
```

Figure 2a – Normalizing the data

The next step is to split the training dataset into training and validation sets. The 'load_dataset' custom function uses an 8:2 training to test ratio. Stratification has been implemented on the target value. An extra filter variable 'emotions_to_remove' was added so as training could be performed using certain selected emotions only (fig 2b). This allows us to observe which emotions are conducive to accurate predictions and which emotions tend to decrease the model accuracy.

```
def load_dataset(csv_file, transform, emotions_to_remove):
    data = pd.read_csv(csv_file)
    for e in emotions_to_remove:
      data = data[data['emotion'] != e]
    y = data.emotion
    train_data, test_data = train_test_split(data, test_size=0.2, random_state=42, stratify=y)
    train_dataset = FERDataset(train_data, transform=transform)
    test_dataset = FERDataset(test_data, transform=transform)

    return train_dataset, test_dataset
```

```
# Option to remove certain emotions for testing
# 0=Angry, 1=Disgust, 2=Fear, 3=Happy, 4=Sad, 5=Surprise, 6=Neutral
emotions_to_remove = []
train_dataset, test_dataset = load_dataset('/content/drive/MyDrive/Colab Notebooks/AML Assignment 02/train.csv', transform, emotions_to_remove)
length = len(train_dataset)
print(f"Length of Datset is {length}")
```

Figure 2b – Splitting the data

Once the data has been split into training and test sets, the next step is to load the data. Training data is set to shuffle while the test data is not. A batch size of 32 was selected after some initial training runs and optional GPU is implemented to speed up training (fig 2c).

```
# Set device
device = 'cuda'
bs = 32
train_loader = DataLoader(train_dataset, batch_size=bs, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=bs, shuffle=False)
```

Figure 2c – Loading the data

3. Data Augmentation

A number of augmentation techniques were employed pre-training, with reference to [2] and [3]. The most effective was found to be 'RandomHorizontalFlip', which is quite logical as we are dealing with facial portraits so mirror images should be congruent with a typical dataset of this kind. As figure 3a shows, throughout the following tuning phase, various techniques were implemented and alternative scaling was also used.

```
# Define the transformations to be applied to your dataset
transform = T.Compose([
    T.ToPILImage(),
    T.ToTensor(),
    T.Normalize(mean=[0.5], std=[0.5]),
    #T.Normalize((0.5), (0.5)),
    #T.Normalize(mean=[0.5], std=[0.5 * 2]),
    #T.Lambda(lambda x: torch.mul(x, 0.5) + 0.5),
    T.RandomHorizontalFlip(),
    #T.RandomVerticalFlip(p=1.0),
    #T.TenCrop(size = 48, vertical_flip = True),
    #T.RandomRotation(degrees=10),
    #T.RandomCrop(size=44, padding=4),
])
```

Figure 3a – Dataset transformations and augmentations

In addition to using Torchvision transform augmentation, some custom augmentation was implemented, specifically to increase the number of 'disgust' images in the dataset. Fig 3b shows, the emotion (1) 'disgust' is first processed so that each image is duplicated as a mirror image and added to the dataset.

Next, each 'disgust' image is rotated once at a random angle in a 20 degree range. This new image is also added to the existing dataset. The initial size of the training set is augmented from 22967 to 24014 meaning there are now 1047 more 'disgust' images in the training dataset, bringing the total up from 349 to 1396.

```
Length of Datset is 22967
```

```
# Augment the data further
emot = 1 # increase the number of 'disgust' emotions in the dataset

for i in range(length):
  if train_dataset[i][1] == emot:
    temp = TF.hflip(train_dataset[i][0])
    train_dataset.add(temp, train_dataset[i][1])
length = len(train_dataset)
print(f"Datset is {length} after horizontal flipping of {emotions[emot]}")

for j in range(length):
  if train_dataset[j][1] == emot:
    temp = TF.rotate(train_dataset[j][0], random.randint(-10, 10))
    train_dataset.add(temp, train_dataset[j][1])
length = len(train_dataset)
print(f"Datset is {length} after random rotation of {emotions[emot]}")


```

```
Datset is 23316 after horizontal flipping of Disgust
Datset is 24014 after random rotation of Disgust
```

Figure 3b – Custom data augmentation (horizontal flip and rotation)

4. Model Development

Figure 4a shows how the CNN architecture combines convolutional layers, batch normalization, residual connections, and fully connected layers to learn and classify emotions from the input images.

```python
# Define the CNN

output = 70

class CNN(nn.Module):
    def __init__(self, num_classes):
        super(CNN, self).__init__()
        self.conv1 = nn.Sequential(
            nn.Conv2d(in_channels=1, out_channels=44, kernel_size=5, stride=(1,1), padding=(1,1)),
            nn.PReLU(),
            nn.MaxPool2d((3, 3), stride=2)
        )
        self.conv2 = nn.Sequential(
            nn.Conv2d(44, 40, kernel_size=5, stride=(1,1), padding=(1,1)),
            nn.PReLU(),
            nn.MaxPool2d((3, 3), stride=2)
        )
        self.conv3 = nn.Sequential(
            nn.Conv2d(40, 36, kernel_size=3, stride=(1,1), padding=(1,1)),
            nn.PReLU(),
            nn.AvgPool2d((3, 3), stride=1)
        )
        self.dropout = nn.Dropout(p=0.5)

        self.batch_norm1 = nn.BatchNorm2d(44)
        self.batch_norm2 = nn.BatchNorm2d(40)
        self.batch_norm3 = nn.BatchNorm2d(36)
        self.conv_residual1 = nn.Conv2d(in_channels=44, out_channels=44, kernel_size=1) # not used
        self.conv_residual2 = nn.Conv2d(in_channels=40, out_channels=40, kernel_size=1)
        self.conv_residual3 = nn.Conv2d(in_channels=36, out_channels=36, kernel_size=1) # not used

        self.fc1 = nn.Sequential(
            nn.Linear(1764, output),
            nn.ELU(),
        )
        self.fc2 = nn.Sequential(
            nn.Linear(output, num_classes),
            nn.Tanh()
        )
```

Figure 4a – CNN Architecture

1. Convolutional Layers:

    - The input to the network is a grayscale image (1 channel).
    - The first convolutional layer has 1 input channel, 44 output channels, a kernel size of 5x5, and a PReLU activation function.

- A max pooling operation with a kernel size of 3x3 and stride 2 is applied after the first convolutional layer.
- The second convolutional layer has 44 input channels, 40 output channels, a kernel size of 5x5, and a PReLU activation function.
- Another max pooling operation with a kernel size of 3x3 and stride 2 is applied after the second convolutional layer.
- The third convolutional layer has 40 input channels, 36 output channels, a kernel size of 3x3, and a PReLU activation function.
- An average pooling operation with a kernel size of 3x3 and stride 1 is applied after the third convolutional layer.

2. Dropout and Batch Normalization:
    - A dropout layer with a dropout rate of 0.5 is applied after the third convolutional layer (see fig 4b – forward pass).
    - Batch normalization is applied after each convolutional layer to normalize the activations and improve training stability.
    - There are separate batch normalization layers for each convolutional layer.
3. Residual Connections:
    - Residual connections are used in the network to facilitate gradient flow and improve the network's ability to learn. A residual connection is applied after the second convolutional layer.
4. Fully Connected Layers:
    - The output from the last convolutional layer is flattened and passed through a fully connected layer.
    - The first fully connected layer has an input size of 1764 (based on the previous layer's output shape) and an output size determined by the adjustable variable 'output'.
    - An ELU (Exponential Linear Unit) activation function is applied after the first fully connected layer.
    - The second fully connected layer has an input size of 'output' (from the previous layer) and an output size equal to the number of emotion classes.
    - A Tanh activation function is applied after the second fully connected layer.

The forward pass is viewable below in fig 4b. Input is passed through the convolutional layers, followed by batch normalization. There is also the addition of a residual connection. Output from the last convolutional layer is flattened and passed through the dropout layer. The output from the dropout layer

is then fed into the fully connected layers, which output the final classification scores for each emotion class.

```
def forward(self, x):

    out = self.conv1(x)
    out = self.batch_norm1(out)
    out = self.conv2(out)
    residual = self.conv_residual2(out)
    out = self.batch_norm2(out)
    out += residual
    out = self.conv3(out)
    out = self.batch_norm3(out)
    out = out.view(out.size(0), -1)
    out = self.dropout(out)
    out = self.fc1(out)
    out = self.fc2(out)

    return out
```

Figure 4b – Forward pass

CrossEntropyLoss was selected as the loss function along with the 'Adam' optimizer and a learning rate of 0.0002. (fig 4c).

```
# Define your Model
num_classes = 7-len(emotions_to_remove)
model = CNN(num_classes).to(device)

# Define your loss function
criterion = nn.CrossEntropyLoss()

# Define your optimizer
optimizer = optim.Adam(model.parameters(), lr=0.0002)
```

Figure 4c – The model, loss and optimizer

5. Model Training and Evaluation

At the start of the assignment, the original CNN template provided was run for 20 epochs to evaluate its performance. It reached an accuracy of 0.5146 after 19 epochs with a train loss and test loss of 1.456 and 1.643 respectively (fig 5a).

```
Epoch [1/20],     Train Loss: 1.8423,     Test Loss: 1.7764,     Accuracy: 0.3817
Epoch [2/20],     Train Loss: 1.7589,     Test Loss: 1.7440,     Accuracy: 0.4087
Epoch [3/20],     Train Loss: 1.7296,     Test Loss: 1.7309,     Accuracy: 0.4248
Epoch [4/20],     Train Loss: 1.7092,     Test Loss: 1.7174,     Accuracy: 0.4361
Epoch [5/20],     Train Loss: 1.6915,     Test Loss: 1.7033,     Accuracy: 0.4509
Epoch [6/20],     Train Loss: 1.6724,     Test Loss: 1.6908,     Accuracy: 0.4687
Epoch [7/20],     Train Loss: 1.6543,     Test Loss: 1.6872,     Accuracy: 0.4746
Epoch [8/20],     Train Loss: 1.6346,     Test Loss: 1.6841,     Accuracy: 0.4716
Epoch [9/20],     Train Loss: 1.6144,     Test Loss: 1.6747,     Accuracy: 0.4822
Epoch [10/20],     Train Loss: 1.5965,     Test Loss: 1.6666,     Accuracy: 0.4911
Epoch [11/20],     Train Loss: 1.5754,     Test Loss: 1.6618,     Accuracy: 0.4963
Epoch [12/20],     Train Loss: 1.5580,     Test Loss: 1.6671,     Accuracy: 0.4876
Epoch [13/20],     Train Loss: 1.5411,     Test Loss: 1.6516,     Accuracy: 0.5063
Epoch [14/20],     Train Loss: 1.5225,     Test Loss: 1.6557,     Accuracy: 0.5012
Epoch [15/20],     Train Loss: 1.5061,     Test Loss: 1.6485,     Accuracy: 0.5106
Epoch [16/20],     Train Loss: 1.4934,     Test Loss: 1.6510,     Accuracy: 0.5068
Epoch [17/20],     Train Loss: 1.4777,     Test Loss: 1.6525,     Accuracy: 0.5063
Epoch [18/20],     Train Loss: 1.4671,     Test Loss: 1.6486,     Accuracy: 0.5099
Epoch [19/20],     Train Loss: 1.4564,     Test Loss: 1.6439,     Accuracy: 0.5146
Epoch [20/20],     Train Loss: 1.4492,     Test Loss: 1.6533,     Accuracy: 0.5061
```

Figure 5a – Original model results over 20 epochs

In contrast, the final model showed a higher performace after much architecture modification and hyperparameter tuning. After 23 epochs, accuracy reached 0.6137 with a train loss and test loss of 1.05 and 1.249 respectively (fig 5b).

```
Epoch [1/30],     Train Loss: 1.5666,     Test Loss: 1.4886,     Accuracy: 0.4599
Epoch [2/30],     Train Loss: 1.4137,     Test Loss: 1.4117,     Accuracy: 0.5014
Epoch [3/30],     Train Loss: 1.3570,     Test Loss: 1.3665,     Accuracy: 0.5277
Epoch [4/30],     Train Loss: 1.3156,     Test Loss: 1.3387,     Accuracy: 0.5465
Epoch [5/30],     Train Loss: 1.2892,     Test Loss: 1.3185,     Accuracy: 0.5648
Epoch [6/30],     Train Loss: 1.2619,     Test Loss: 1.2985,     Accuracy: 0.5646
Epoch [7/30],     Train Loss: 1.2398,     Test Loss: 1.2937,     Accuracy: 0.5711
Epoch [8/30],     Train Loss: 1.2178,     Test Loss: 1.2883,     Accuracy: 0.5787
Epoch [9/30],     Train Loss: 1.2030,     Test Loss: 1.2855,     Accuracy: 0.5778
Epoch [10/30],     Train Loss: 1.1870,     Test Loss: 1.2692,     Accuracy: 0.5902
Epoch [11/30],     Train Loss: 1.1704,     Test Loss: 1.2623,     Accuracy: 0.5911
Epoch [12/30],     Train Loss: 1.1588,     Test Loss: 1.2627,     Accuracy: 0.5913
Epoch [13/30],     Train Loss: 1.1474,     Test Loss: 1.2661,     Accuracy: 0.5909
Epoch [14/30],     Train Loss: 1.1352,     Test Loss: 1.2489,     Accuracy: 0.6034
Epoch [15/30],     Train Loss: 1.1244,     Test Loss: 1.2677,     Accuracy: 0.5895
Epoch [16/30],     Train Loss: 1.1160,     Test Loss: 1.2544,     Accuracy: 0.6012
Epoch [17/30],     Train Loss: 1.1020,     Test Loss: 1.2575,     Accuracy: 0.6064
Epoch [18/30],     Train Loss: 1.0949,     Test Loss: 1.2557,     Accuracy: 0.5987
Epoch [19/30],     Train Loss: 1.0850,     Test Loss: 1.2539,     Accuracy: 0.6000
Epoch [20/30],     Train Loss: 1.0749,     Test Loss: 1.2468,     Accuracy: 0.6034
Epoch [21/30],     Train Loss: 1.0650,     Test Loss: 1.2507,     Accuracy: 0.6005
Epoch [22/30],     Train Loss: 1.0577,     Test Loss: 1.2442,     Accuracy: 0.6073
Epoch [23/30],     Train Loss: 1.0505,     Test Loss: 1.2459,     Accuracy: 0.6137
Epoch [24/30],     Train Loss: 1.0440,     Test Loss: 1.2595,     Accuracy: 0.6007
Epoch [25/30],     Train Loss: 1.0370,     Test Loss: 1.2517,     Accuracy: 0.6082
Epoch [26/30],     Train Loss: 1.0276,     Test Loss: 1.2408,     Accuracy: 0.6108
Epoch [27/30],     Train Loss: 1.0227,     Test Loss: 1.2539,     Accuracy: 0.6069
Epoch [28/30],     Train Loss: 1.0126,     Test Loss: 1.2613,     Accuracy: 0.6057
Epoch [29/30],     Train Loss: 1.0058,     Test Loss: 1.2537,     Accuracy: 0.6083
Epoch [30/30],     Train Loss: 0.9988,     Test Loss: 1.2591,     Accuracy: 0.6068
```

Figure 5b – Final results over 30 epochs

The plotted scores of the model can be seen in figure 5c.

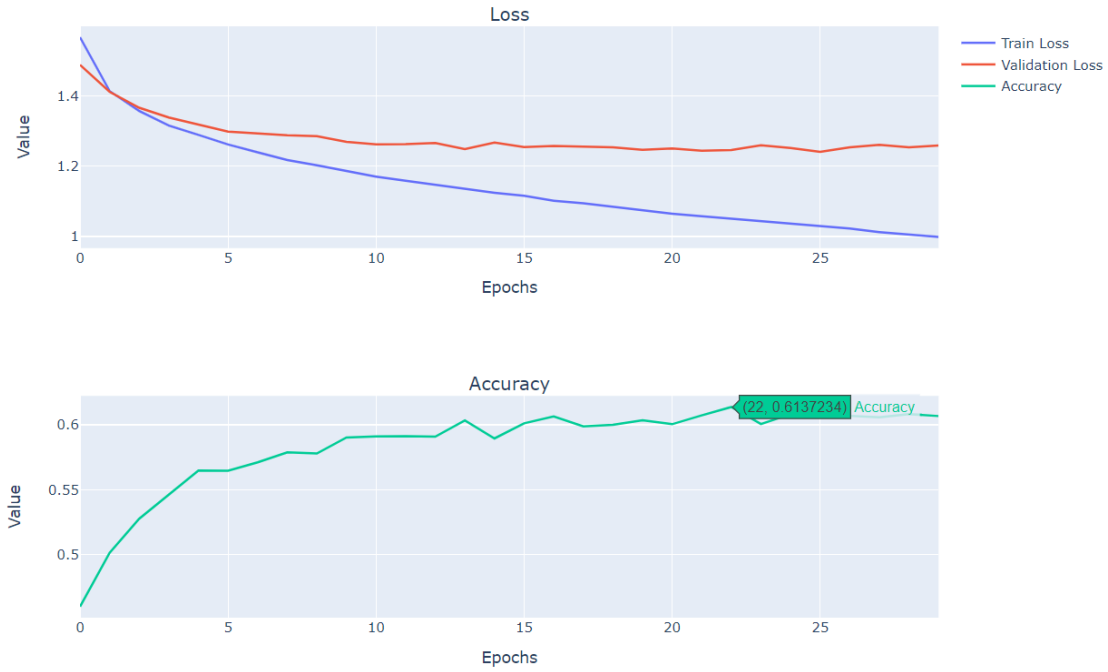

Figure 5c – Final training, validation and accuracy plots

Results of the emotion classification model are visible in fig 5d which used the 'test.csv' image dataset. This selection shows correct identification of a number of emotions.

Figure 5d – Model evaluation on 'test.csv'

6. Hyperparameter Tuning

Many optimizers were tried in the building of this model. Figure 6a shows just the optimizers and a selection of the settings which were used. In the end, Adam was used with a learning rate of 0.0002.

```
# Experimental optimizers
#optimizer = optim.Adamax(model.parameters(), lr=0.002, betas=(0.9, 0.999), eps=1e-08, weight_decay=0.0) # sigmoid
#optimizer = optim.Adamax(model.parameters(), lr=0.0015)
#optimizer = optim.Adam(model.parameters(), lr=0.0001, weight_decay=0.05) # betas=(0.9, 0.999) #, eps=1e-08,
#optimizer = optim.Adam(model.parameters(), lr=0.0002) # tanh
#optimizer = optim.Adagrad(model.parameters(), lr=0.003, eps=1e-6)
#optimizer = optim.SGD(model.parameters(), lr=0.005)
#optimizer = optim.SGD(model.parameters(), lr=0.05, momentum=0.01)
#optimizer = optim.ASGD(model.parameters(), lr=0.005)
#optimizer = optim.RMSprop(model.parameters(), lr=0.01)
#optimizer = optim.Rprop(model.parameters(), lr=0.0001)
```

Figure 6a – Optimizers used during tuning phase

In order to find the optimal settings for each optimizer, a loop function was created to iterate through different learning rates and store the highest accuracy and the epoch at which it was achieved (fig 6b).

```
# Training and evaluation
num_epochs = 30

# find best learning rate during multiple training runs
acc = 0.0
for i in range(1,11,1):
  for epoch in range(num_epochs):
      train_loss = train(model, train_loader, criterion, optimizer, device)
      test_loss, accuracy = evaluate(model, test_loader, criterion, device)
      optimizer = optim.Adam(model.parameters(), lr=(i*0.0001))

      if accuracy > acc:
        acc = accuracy
        print(f"LR: {i/10}, Epoch:{epoch}, Accuracy: {accuracy:.4f}")
```

Figure 6b – Iteration loop to find optimal learning rate for optimizers

A number of different batch sizes were used throughout training. A size of 32 was settled upon as it appeared to produce the most stable results (fig 6c).

```
# Set device
device = 'cuda'
bs = 32
train_loader = DataLoader(train_dataset, batch_size=bs, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=bs, shuffle=False)
```

Figure 6c – Batch size of 32

Another parameter that was experimented with was the output size going from the first fully connected layer into the second fully connected layer. An output size of 70 was found to achieve more optimal results (fig 6d).

```
# Define the CNN

output = 70

class CNN(nn.Module):
    def __init__(self, num_classes):
        super(CNN, self).__init__()
        self.conv1 = nn.Sequential(
            nn.Conv2d(in_channels=1, out_channels=44, kernel_size=5, stride=(1,1), padding=(1,1)),
            nn.PReLU(),
            nn.MaxPool2d((3, 3), stride=2)
        )
        self.conv2 = nn.Sequential(
            nn.Conv2d(44, 40, kernel_size=5, stride=(1,1), padding=(1,1)),
            nn.PReLU(),
            nn.MaxPool2d((3, 3), stride=2)
        )
        self.conv3 = nn.Sequential(
            nn.Conv2d(40, 36, kernel_size=3, stride=(1,1), padding=(1,1)),
            nn.PReLU(),
            nn.AvgPool2d((3, 3), stride=1)
        )
        self.dropout = nn.Dropout(p=0.5)

        self.batch_norm1 = nn.BatchNorm2d(44)
        self.batch_norm2 = nn.BatchNorm2d(40)
        self.batch_norm3 = nn.BatchNorm2d(36)
        self.conv_residual1 = nn.Conv2d(in_channels=44, out_channels=44, kernel_size=1) # not used
        self.conv_residual2 = nn.Conv2d(in_channels=40, out_channels=40, kernel_size=1)
        self.conv_residual3 = nn.Conv2d(in_channels=36, out_channels=36, kernel_size=1) # not used

        self.fc1 = nn.Sequential(
            nn.Linear(1764, output),
            nn.ELU(),
        )
        self.fc2 = nn.Sequential(
            nn.Linear(output, num_classes),
            nn.Tanh()
        )
```

Figure 6d – Output size set to 70

Initially, the model employed a Softmax function on the second fully connected layer. After trying out different scaling it was found that a combination of alternate scaling (-1 and 1) and Tanh function was producing higher accuracy than other functions such as Sigmoid (fig 6e).

```
# Define the transformations to be applied to your dataset
transform = T.Compose([
    T.ToPILImage(),
    T.ToTensor(),
    T.Normalize(mean=[0.5], std=[0.5]),
```

```
train_dataset[0][0]
```

```
tensor([[[ 0.6000,  0.5765,  0.6078,  ..., -0.8431, -0.6706, -0.6941],
         [ 0.5843,  0.5529,  0.5843,  ..., -0.8431, -0.7333, -0.7412],
         [ 0.5216,  0.5059,  0.5686,  ..., -0.8431, -0.8353, -0.7804],
         ...,
         [ 0.3569,  0.3098,  0.3255,  ...,  0.1608,  0.0902, -0.0431],
         [ 0.3490,  0.3490,  0.3412,  ...,  0.1451,  0.0667, -0.0980],
         [ 0.3961,  0.3725,  0.3412,  ...,  0.1137,  0.0510, -0.2000]]])
```

Figure 6e – Reverting the scaling (minus 1 to 1)

A combination of max and average pooling was used throughout the architecture. The first two layers used max pooling (3, 3) and stride of 2, with average pooling (3, 3) and stride of 1 on the third layer. This combination proved to be the most successful with regard to training and validation loss as well as accuracy (fig 6f).

```
self.conv1 = nn.Sequential(
    nn.Conv2d(in_channels=1, out_channels=44, kernel_size=5, stride=(1,1), padding=(1,1)),
    nn.PReLU(),
    nn.MaxPool2d((3, 3), stride=2)
)
self.conv2 = nn.Sequential(
    nn.Conv2d(44, 40, kernel_size=5, stride=(1,1), padding=(1,1)),
    nn.PReLU(),
    nn.MaxPool2d((3, 3), stride=2)
)
self.conv3 = nn.Sequential(
    nn.Conv2d(40, 36, kernel_size=3, stride=(1,1), padding=(1,1)),
    nn.PReLU(),
    nn.AvgPool2d((3, 3), stride=1)
)
```

Figure 6f – Max and average pooling settings

As the data was scaled to between minus 1 and 1, various activation functions were tested. ReLU functioned quite well but PReLU appeared to converge quicker so it was preferred.

Batch normalization was also found to be of benefit when used after each round of convolution. Whilst several dropouts were experimented with, they did tend to slow down convergence and so were not used in the final model.

Conclusion

Modification of the original CNN, along with hyperparameter tuning enabled the CNN to increase its accuracy from 0.5 (20 epochs) to 0.61 (23 epochs), which is an improvement of approximately 20%. The model has reached accuracy of above 0.62 when tested over multiple iterations.

References

[1] Kaggle - Emotion Recognition dataset. https://www.kaggle.com/c/challenges-in-representation-learning-facial-expression-recognition-challenge/data

[2] Pytorch - Transforming and Augmenting Images. https://pytorch.org/vision/main/transforms.html https://en.wikipedia.org/wiki/United_States_invasion_of_Panama

[3] E. Anello - A Comprehensive Guide to Image Augmentation using Pytorch. https://towardsdatascience.com/a-comprehensive-guide-to-image-augmentation-using-pytorch-fb162f2444be