AI_Lab_2024_1stAssignment-1

Mehedi Hasan Shakil
Roll: 2010876138

December 2024

1 Build a fully connected neural network (FCNN) and a convolutional neural network (CNN) for classifying 10 classes of images.

Import Necessary Modules

```
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.layers import Input, Dense, Conv2D, MaxPooling2D, Flatten
from tensorflow.keras.models import Model
```

Build a Fully Connected Neural Network (FCNN) for classifying 10 classes of images.

```
inputs = Input((28, 28, 1), name = 'InputLayer')
x = Flatten()(inputs)
x = Dense(512, activation = 'relu')(x)
outputs = Dense(10, name = 'OutputLayer', activation = 'softmax')(x)
model = Model(inputs, outputs, name = 'Multi-Class-Classification')
model.summary()
```

Model: "Multi-Class-Classification"

Layer (type)	Output Shape	Param #
InputLayer (InputLayer)	(None, 28, 28, 1)	0
flatten (Flatten)	(None, 784)	0
dense (Dense)	(None, 512)	401,920
OutputLayer (Dense)	(None, 10)	5,130

Total params: 407,050 (1.55 MB)
Trainable params: 407,050 (1.55 MB)
Non-trainable params: 0 (0.00 B)

Figure 1: Architecture of the FCNN

Build a convolutional neural network (CNN) for classifying 10 classes of images.

```
inputs = Input((28, 28, 1), name='InputLayer')
x = Conv2D(32, (3, 3), activation='relu')(inputs)
x = MaxPooling2D((2, 2))(x)
x = Conv2D(64, (3, 3), activation='relu')(x)
x = MaxPooling2D((2, 2))(x)
x = Conv2D(128, (3, 3), activation='relu')(x)
x = Flatten()(x)

# Final dense layers
x = Dense(512, activation='relu')(x)
outputs = Dense(10, name='OutputLayer', activation='softmax')(x)

# Create the model
model = Model(inputs, outputs, name='ConvNet-Classification')

# Summary of the model
model.summary()
```

Model: "ConvNet-Classification"

Layer (type)	Output Shape	Param #
InputLayer (InputLayer)	(None, 28, 28, 1)	0
conv2d (Conv2D)	(None, 26, 26, 32)	320
max_pooling2d (MaxPooling2D)	(None, 13, 13, 32)	0
conv2d_1 (Conv2D)	(None, 11, 11, 64)	18,496
max_pooling2d_1 (MaxPooling2D)	(None, 5, 5, 64)	0
conv2d_2 (Conv2D)	(None, 3, 3, 128)	73,856
flatten (Flatten)	(None, 1152)	0
dense (Dense)	(None, 512)	590,336
OutputLayer (Dense)	(None, 10)	5,130

Total params: 688,138 (2.63 MB)
Trainable params: 688,138 (2.63 MB)
Non-trainable params: 0 (0.00 B)

Figure 2: Architecture of the CNN

2 Train and test your FCNN and CNN by the Fashion dataset. Discuss your results by comparing performance between two types of networks.

Import Necessary Modules

```
from tensorflow.keras.datasets import fashion_mnist
import matplotlib.pyplot as plt
import numpy as np
```

Function to Display Loaded Data

```
def display_img(img_set, title_set):
    n = len(title_set)
    for i in range(n):
        plt.subplot(3, 3, i + 1)
        plt.imshow(img_set[i], cmap = 'gray')
        plt.title(title_set[i])
plt.show()
plt.close()
```

Load Dataset

```
# Load data
(trainX, trainY), (testX, testY) = load_data()

# Investigate loaded data
print('trainX.shape: {}, trainY.shape: {}, testX.shape: {}, testY.shape: {}'.format(trainX.s
print('trainX.dtype: {}, trainY.dtype: {}, testX.dtype: {}, testY.dtype: {}'.format(trainX.d
print('trainX.Range: {} - {}, testX.Range: {} - {}'.format(trainX.max(), trainX.min(), testX

# Display some loaded image data
display_img(trainX[:9], trainY[:9])
```

```
trainX.shape: (60000, 28, 28), trainY.shape: (60000,), testX.shape: (10000, 28, 28), testY.shape: (10000,)
trainX.dtype: uint8, trainY.dtype: uint8, testX.dtype: uint8, testY.dtype: uint8
trainX.Range: 255 - 0, testX.Range: 255 - 0
```

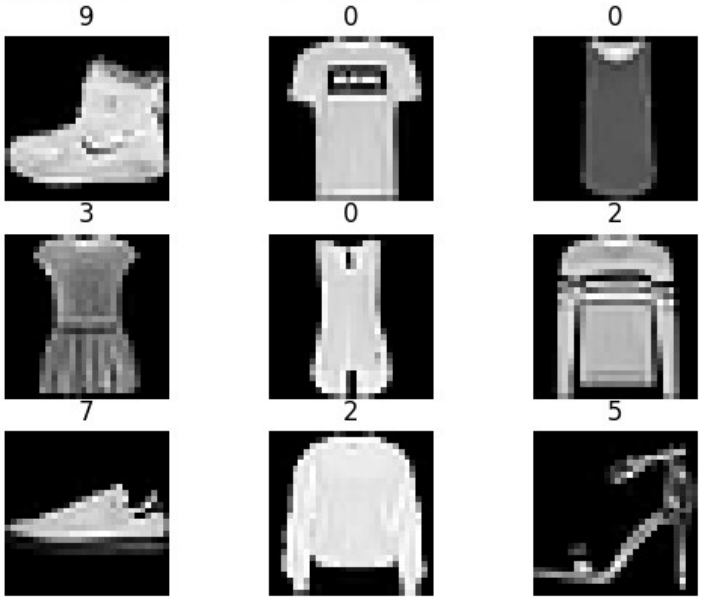

Figure 3: Some Sample Images of the Fashion Dataset

Prepare Dataset

```
# Turn 2D images into 3D so that trainX and trainY will be 4D since Convolutional layer take
trainX = np.expand_dims(trainX, axis = -1)
testX = np.expand_dims(testX, axis = -1)

# Investigate update x
print('trainX.shape: {}, testX.shape: {}'.format(trainX.shape, testX.shape))
print('trainX.dtype: {}, text.dtype: {}'.format(trainX.dtype, testX.dtype))
print('trainX.Range: {} - {}, testX.Range: {} - {}'.format(trainX.max(), trainX.min(), testX

# Turn y into one-hot-encoding, so that we can use 10 neurons in the output layer
trainY = to_categorical(trainY, num_classes = 10)
testY = to_categorical(testY, num_classes = 10)

# Investigate updated y
print('trainY.shape: {}, testY.shape: {}'.format(trainY.shape, testY.shape))
print('trainY.dtype: {}, testY.dtype: {},'.format(trainY.dtype, testY.dtype))
print(trainY[:5])
```

```
trainX.shape: (60000, 28, 28, 1), testX.shape: (10000, 28, 28, 1)
trainX.dtype: uint8, text.dtype: uint8
trainX.Range: 255 - 0, testX.Range: 255 - 0
trainY.shape: (60000, 10), testY.shape: (10000, 10)
trainY.dtype: float64, testY.dtype: float64,
[[0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]
 [1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 1. 0. 0. 0. 0. 0. 0.]
 [1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]]
```

Figure 4: Prepared Dataset: X in 4D and Y in one-hot-encoding

Train FCNN Classifier

```
model.compile(loss = 'categorical_crossentropy', metrics = ['accuracy'])
model.fit(trainX, trainY, batch_size = 32, validation_split = 0.1, epochs = 10)
```

```
Epoch 1/10
1688/1688 ━━━━━━━━━━━━━━━━━━━━ 14s 8ms/step - accuracy: 0.6945 - loss: 29.3104 - val_accuracy: 0.7850 - val_loss: 0.7559
Epoch 2/10
1688/1688 ━━━━━━━━━━━━━━━━━━━━ 20s 7ms/step - accuracy: 0.7998 - loss: 0.7360 - val_accuracy: 0.8047 - val_loss: 1.0492
Epoch 3/10
1688/1688 ━━━━━━━━━━━━━━━━━━━━ 20s 7ms/step - accuracy: 0.8165 - loss: 0.7038 - val_accuracy: 0.7947 - val_loss: 1.1393
Epoch 4/10
1688/1688 ━━━━━━━━━━━━━━━━━━━━ 20s 7ms/step - accuracy: 0.8300 - loss: 0.6457 - val_accuracy: 0.8243 - val_loss: 0.8408
Epoch 5/10
1688/1688 ━━━━━━━━━━━━━━━━━━━━ 12s 7ms/step - accuracy: 0.8363 - loss: 0.6484 - val_accuracy: 0.8250 - val_loss: 0.7353
Epoch 6/10
1688/1688 ━━━━━━━━━━━━━━━━━━━━ 21s 7ms/step - accuracy: 0.8363 - loss: 0.6521 - val_accuracy: 0.8308 - val_loss: 0.7436
Epoch 7/10
1688/1688 ━━━━━━━━━━━━━━━━━━━━ 12s 7ms/step - accuracy: 0.8387 - loss: 0.6524 - val_accuracy: 0.8113 - val_loss: 1.0569
Epoch 8/10
1688/1688 ━━━━━━━━━━━━━━━━━━━━ 20s 7ms/step - accuracy: 0.8374 - loss: 0.6670 - val_accuracy: 0.8118 - val_loss: 0.8924
Epoch 9/10
1688/1688 ━━━━━━━━━━━━━━━━━━━━ 22s 8ms/step - accuracy: 0.8349 - loss: 0.6705 - val_accuracy: 0.8238 - val_loss: 0.8493
Epoch 10/10
1688/1688 ━━━━━━━━━━━━━━━━━━━━ 12s 7ms/step - accuracy: 0.8379 - loss: 0.6120 - val_accuracy: 0.8265 - val_loss: 0.8408
<keras.src.callbacks.history.History at 0x7d2e4dc4efb0>
```

Figure 5: Traing Result of the FCNN

Performance Testing of the FCNN

```
# Evaluate model performance
model.evaluate(testX, testY)

# Predict Y values
predictY = model.predict(testX)

print('OriginalY    PredictedY')
print('=========    ==========')
for i in range(10):
  print(np.argmax(testY[i]), '\t\t', np.argmax(predictY[i]))
```

```
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.8197 - loss: 0.8504
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step
OriginalY    PredictedY
=========    ==========
9                9
2                2
1                1
1                1
6                6
1                1
4                4
6                6
5                5
7                7
```

Figure 6: Performace Evaluation of the FCNN

Train CNN Classifier

```
model.compile(loss = 'categorical_crossentropy', metrics = ['accuracy'])
model.fit(trainX, trainY, batch_size = 32, validation_split = 0.1, epochs = 10)
```

```
Epoch 1/10
1688/1688 ━━━━━━━━━━━━━━━━━━━━ 11s 4ms/step - accuracy: 0.7555 - loss: 1.6739 - val_accuracy: 0.8543 - val_loss: 0.3972
Epoch 2/10
1688/1688 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.8675 - loss: 0.3748 - val_accuracy: 0.8545 - val_loss: 0.4251
Epoch 3/10
1688/1688 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.8784 - loss: 0.3471 - val_accuracy: 0.8623 - val_loss: 0.3905
Epoch 4/10
1688/1688 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.8787 - loss: 0.3520 - val_accuracy: 0.8763 - val_loss: 0.3783
Epoch 5/10
1688/1688 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.8768 - loss: 0.3569 - val_accuracy: 0.8457 - val_loss: 0.4492
Epoch 6/10
1688/1688 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.8754 - loss: 0.3618 - val_accuracy: 0.8600 - val_loss: 0.4251
Epoch 7/10
1688/1688 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.8779 - loss: 0.3598 - val_accuracy: 0.8338 - val_loss: 0.5200
Epoch 8/10
1688/1688 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.8765 - loss: 0.3653 - val_accuracy: 0.8550 - val_loss: 0.5267
Epoch 9/10
1688/1688 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.8756 - loss: 0.3673 - val_accuracy: 0.8360 - val_loss: 0.4501
Epoch 10/10
1688/1688 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.8705 - loss: 0.3854 - val_accuracy: 0.8585 - val_loss: 0.4716
<keras.src.callbacks.history.History at 0x7a52f203a4a0>
```

Figure 7: Traing Result of the CNN

Performance Testing of the CNN

```
# Evaluate model performance
model.evaluate(testX, testY)

# Predict Y values
predictY = model.predict(testX)

print('OriginalY    PredictedY')
print('=========    ==========')
for i in range(10):
  print(np.argmax(testY[i]), '\t\t', np.argmax(predictY[i]))
```

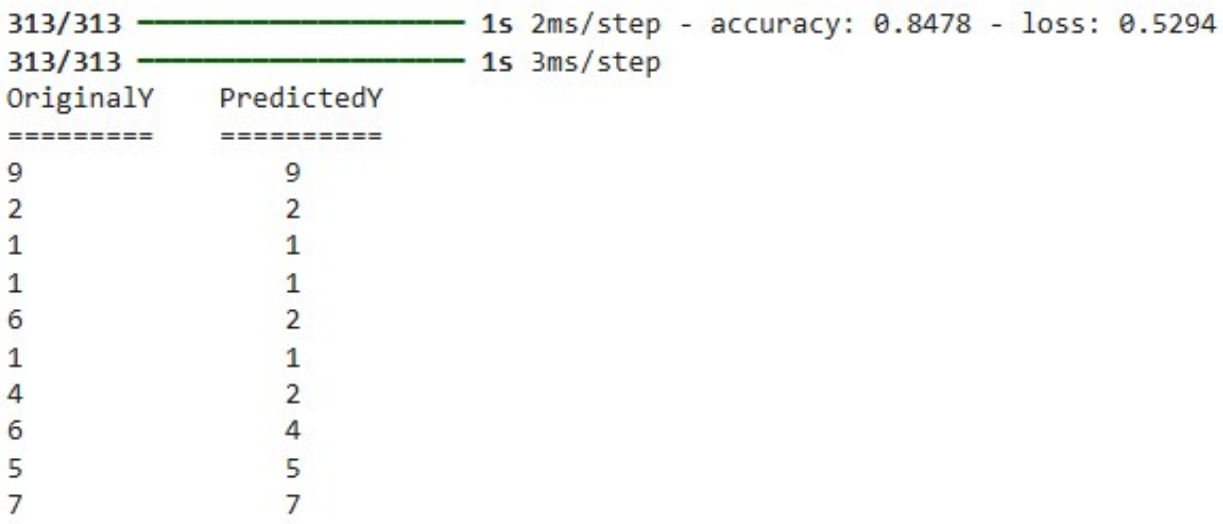

Figure 8: Performace Evaluation of the CNN

Table 1: Perfomance of the FCNN at Different Configurations

Training	**Hidden Layers**	**Loss Fuction**	**Optimizer**	**Batch Size**	**Validation Split**	**Epochs**	**Accuracy**	**Loss**
1	1 (512)	categorical_crossentropy	adam	32	0.1	10	0.8197	0.8504
2	1 (512)	categorical_crossentropy	adam	64	0.1	10	0.8347	0.6563
3	1 (512)	categorical_crossentropy	adam	128	0.1	10	0.8448	0.6257
4	1 (512)	categorical_crossentropy	rmsprop	32	0.1	10	0.8255	0.8647
5	1 (512)	categorical_crossentropy	rmsprop	64	0.1	10	0.8458	0.9238
6	1 (512)	categorical_crossentropy	rmsprop	128	0.1	10	0.8400	0.9862

Table 2: Perfomance of the CNN at Different Configurations

Training	**Conv. Layer + Max-pool**	**Hidden Layers**	**Loss Fuction**	**Optimizer**	**Batch Size**	**Validation Split**	**Epochs**	**Accuracy**	**Loss**
1	1	1 (512)	categorical_crossentropy	adam	32	0.1	10	0.8667	0.6209
2	2	1 (512)	categorical_crossentropy	adam	32	0.1	10	0.8740	0.4738
3	3	1 (512)	categorical_crossentropy	adam	32	0.1	10	0.8328	0.4860
4	3	1 (256)	categorical_crossentropy	adam	32	0.1	10	0.8826	0.3824

From Table 1 and Table 2 it's clear that Convolutional Neural Network (CNN) is better than Fully Convolutional Neural Network (FCNN) for image classification. CNNs are designed to automatically detect local patterns (like edges, and textures) through convolutional layers. This greatly helps to capture spatial relationships in the image. In addition, pooling layers reduce the spatial dimensions while retaining important features. This reduces overfitting and improves computational efficiency. On the other hand, FCNNs treat every pixel as an independent feature and do not consider spatial relationships. This approach leads to a loss of crucial context of pixels. Also, there are a high number of parameters in the FCNN, leading to overfitting. That's why we see that FCNN show a poor performance for image classification.

3 Build a CNN having a pre-trained MobileNet as backbone to classify 10 classes.

Import Necessary Modules

```
from tensorflow.keras.applications import MobileNet
from tensorflow.keras.layers import Flatten, Dense
from tensorflow.keras.models import Model
```

Build a CNN Classifier Based on a Pre-trained MobileNet Model

```
# Load MobileNet with pretrained weights
mobilenet_model = MobileNet(input_shape=(224, 224, 3), weights='imagenet', include_top=False

# Build a new model based on pre-trained MobileNet
inputs = mobilenet_model.inputs
x = mobilenet_model.output
x = Flatten()(x)
x = Dense(512, activation='relu')(x)
outputs = Dense(10, activation='softmax')(x)
model = Model(inputs, outputs, name='NewMobileNetModel')
model.summary()
```

Model: "NewMobileNetModel"

Layer (type)	Output Shape	Param #
input_layer_1 (InputLayer)	(None, 224, 224, 3)	0
conv1 (Conv2D)	(None, 112, 112, 32)	864
conv1_bn (BatchNormalization)	(None, 112, 112, 32)	128
conv1_relu (ReLU)	(None, 112, 112, 32)	0

conv_pw_13_bn (BatchNormalization)	(None, 7, 7, 1024)	4,096
conv_pw_13_relu (ReLU)	(None, 7, 7, 1024)	0
flatten_1 (Flatten)	(None, 50176)	0
dense_1 (Dense)	(None, 512)	25,690,624
dense_2 (Dense)	(None, 10)	5,130

Total params: 28,924,618 (110.34 MB)
Trainable params: 28,902,730 (110.26 MB)
Non-trainable params: 21,888 (85.50 KB)

Figure 9: Architecture of the CNN based on Pre-trained MobileNet

4 Train and test your CNN having a pre-trained MobileNet as backbone to classify images of the CIFAR-10 dataset. Discuss your results by comparing performance between transfer_learning + fine tuning and only transfer learning.

Function to Display Loaded Data

```
def display_img(img_set, title_set):
    n = len(title_set)
    for i in range(n):
        plt.subplot(3, 3, i + 1)
        plt.imshow(img_set[i])
        plt.title(title_set[i])
        plt.axis('off')
plt.show()
plt.close()
```

Load Dataset

```
from tensorflow.keras.datasets import cifar10

# Load data
(trainX, trainY), (testX, testY) = cifar10.load_data()

# Investigate loaded data
print('trainX.shape: {}, trainY.shape: {}, testX.shape: {}, testY.shape: {}'.format(trainX.s
print('trainX.dtype: {}, trainY.dtype: {}, testX.dtype: {}, testY.dtype: {}'.format(trainX.d
print('trainX.Range: {} - {}, testX.Range: {} - {}'.format(trainX.max(), trainX.min(), testX

# Display some loaded image data
display_img(trainX[:9], trainY[:9])
```

```
trainX.shape: (50000, 32, 32, 3), trainY.shape: (50000, 1), testX.shape: (10000, 32, 32, 3), testY.shape: (10000, 1)
trainX.dtype: uint8, trainY.dtype: uint8, testX.dtype: uint8, testY.dtype: uint8
trainX.Range: 255 - 0, testX.Range: 255 - 0
```

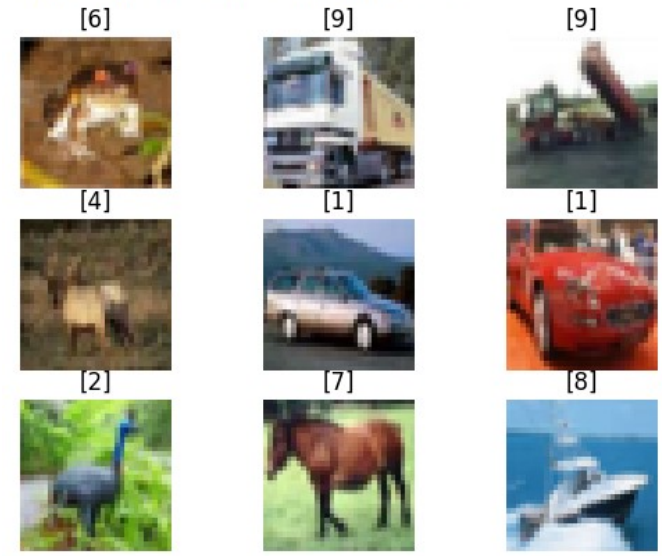

Figure 10: Some Sample Images of the CIFAR-10 Dataset

Prepare Dataset

```
# Investigate update x
print('trainX.shape: {}, testX.shape: {}'.format(trainX.shape, testX.shape))
print('trainX.dtype: {}, text.dtype: {}'.format(trainX.dtype, testX.dtype))
print('trainX.Range: {} - {}, testX.Range: {} - {}'.format(trainX.max(), trainX.min(), testX

# Turn y into one-hot-encoding, so that we can use 10 neurons in the output layer
trainY = to_categorical(trainY, num_classes = 10)
testY = to_categorical(testY, num_classes = 10)

# Investigate updated y
print('trainY.shape: {}, testY.shape: {}'.format(trainY.shape, testY.shape))
print('trainY.dtype: {}, testY.dtype: {},'.format(trainY.dtype, testY.dtype))
print(trainY[:5])
```

```
trainX.shape: (50000, 32, 32, 3), testX.shape: (10000, 32, 32, 3)
trainX.dtype: uint8, text.dtype: uint8
trainX.Range: 255 - 0, testX.Range: 255 - 0
trainY.shape: (50000, 10), testY.shape: (10000, 10)
trainY.dtype: float64, testY.dtype: float64,
[[0. 0. 0. 0. 0. 0. 1. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]
 [0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]]
```

Figure 11: Prepared Dataset: X in 4D and Y in one-hot-encoding

Build the CNN based on MobileNet Backbone to Classify Images of the CIFAR-10 Dataset

```
from tensorflow.keras.applications import MobileNet

# Load MobileNet with pretrained weights
mobilenet_model = MobileNet(input_shape=(32, 32, 3), weights='imagenet', include_top=False)

# Freeze the layers of MobileNet (backbone)
for layer in mobilenet_model.layers:
    layer.trainable = False

# Build a new model based on pre-trained MobileNet
inputs = mobilenet_model.inputs
x = mobilenet_model.output
x = Flatten()(x)
x = Dense(512, activation='relu')(x)
outputs = Dense(10, activation='softmax')(x)
model = Model(inputs, outputs, name='NewMobileNetModel')
model.summary(show_trainable=True)
```

Unfreeze Layers: n is the number of layers to be unfrozen in MobileNet.

```
for layer in mobilenet_model.layers[-n:]:
    layer.trainable = True
```

Training

```
model.compile(loss = 'categorical_crossentropy', metrics = ['accuracy'])
model.fit(trainX, trainY, batch_size = 32, validation_split = 0.1, epochs = 10)
```

Here is the Performance Tabel for Transfer Learning and Fine Tuning

Table 3: Perfomance of the CNN based on MobileNet at Different Configurations. (n is the number of layers to be unfrozen in the backbone.)

Training	Hidden Layers	n	Loss Fuction	Optimizer	Batch Size	Validation Split	Epochs	Accuracy	Loss
1	1 (512)	0	categorical_crossentropy	adam	32	0.1	10	0.2006	2.1489
1	1 (512)	1	categorical_crossentropy	adam	32	0.1	10	0.2018	2.1589
1	1 (512)	2	categorical_crossentropy	adam	32	0.1	10	0.3516	2.9288
1	1 (512)	4	categorical_crossentropy	adam	32	0.1	10	0.3123	2.7345
1	1 (512)	8	categorical_crossentropy	adam	32	0.1	10	0.4121	2.0946
1	1 (512)	16	categorical_crossentropy	adam	32	0.1	10	0.4607	1.8614
1	1 (512)	32	categorical_crossentropy	adam	32	0.1	10	0.5215	1.5496
1	1 (512)	64	categorical_crossentropy	adam	32	0.1	10	0.6202	1.1525
1	1 (512)	86	categorical_crossentropy	adam	32	0.1	10	0.8145	0.6122