Artificial Intelligence

Mst. Shimla Sinthia
2012376105

December 22, 2024

1 Building Neural Networks

1.1 Fully Connected Neural Network (FCNN)

A Fully Connected Neural Network (FCNN) was built to classify images into 10 classes. The network architecture includes:

- A **Flatten** layer to convert the input 28x28 images into a 1D array.
- Two hidden **Dense** layers with ReLU activation containing 128 and 64 neurons respectively.
- An output **Dense** layer with 10 neurons using softmax activation for multi-class classification.

Code:

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten

fcnn_model = Sequential([
    Flatten(input_shape=(28, 28, 1)),
    Dense(128, activation='relu'),
    Dense(64, activation='relu'),
    Dense(10, activation='softmax')
])
```

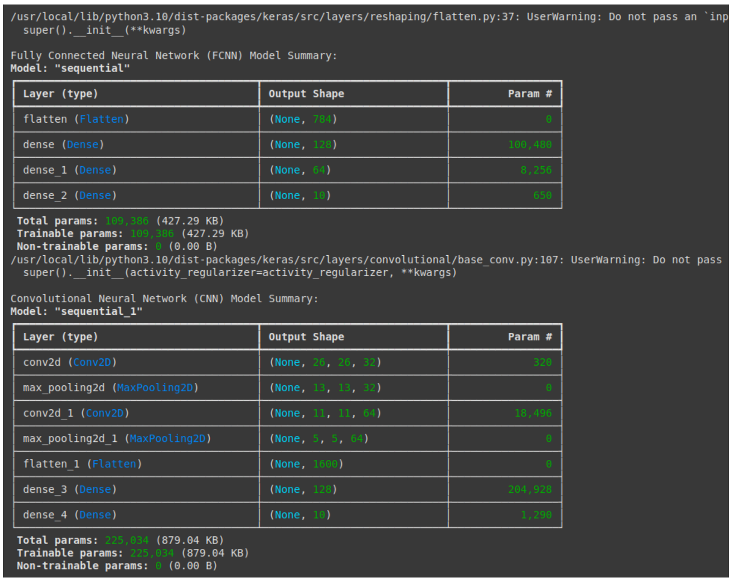
1.2 Convolutional Neural Network (CNN)

The Convolutional Neural Network (CNN) was designed with:

- Two **Conv2D** layers with ReLU activation and kernel size (3x3).
- Two **MaxPooling2D** layers for dimensionality reduction.
- A **Flatten** layer followed by a **Dense** hidden layer with 128 neurons.
- An output layer with 10 neurons using softmax activation.

Code:

```
from tensorflow.keras.layers import Conv2D, MaxPooling2D

cnn_model = Sequential([
    Conv2D(32, (3, 3), activation='relu', input_shape
       =(28, 28, 1)),
    MaxPooling2D((2, 2)),
    Conv2D(64, (3, 3), activation='relu'),
    MaxPooling2D((2, 2)),
    Flatten(),
    Dense(128, activation='relu'),
    Dense(10, activation='softmax')
])
```



/usr/local/lib/python3.10/dist-packages/keras/src/layers/reshaping/flatten.py:37: UserWarning: Do not pass an `inp
super().__init__(**kwargs)

Fully Connected Neural Network (FCNN) Model Summary:
Model: "sequential"

Layer (type)	Output Shape	Param #
flatten (Flatten)	(None, 784)	0
dense (Dense)	(None, 128)	100,480
dense_1 (Dense)	(None, 64)	8,256
dense_2 (Dense)	(None, 10)	650

Total params: 109,386 (427.29 KB)
Trainable params: 109,386 (427.29 KB)
Non-trainable params: 0 (0.00 B)
/usr/local/lib/python3.10/dist-packages/keras/src/layers/convolutional/base_conv.py:107: UserWarning: Do not pass
super().__init__(activity_regularizer=activity_regularizer, **kwargs)

Convolutional Neural Network (CNN) Model Summary:
Model: "sequential_1"

Layer (type)	Output Shape	Param #
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
flatten_1 (Flatten)	(None, 1600)	0
dense_3 (Dense)	(None, 128)	204,928
dense_4 (Dense)	(None, 10)	1,290

Total params: 225,034 (879.04 KB)
Trainable params: 225,034 (879.04 KB)
Non-trainable params: 0 (0.00 B)

Figure 1: FCNN and CNN model

2 Training and Testing Neural Networks

2.1 Dataset Preparation

The Fashion-MNIST dataset was used. The images were normalized to a range of 0 to 1 by dividing by 255. Labels were converted to one-hot encoding using `to_categorical`. **Code:**

```
from tensorflow.keras.datasets import fashion_mnist
from tensorflow.keras.utils import to_categorical

(x_train, y_train), (x_test, y_test) = fashion_mnist.
   load_data()
x_train = x_train.reshape(-1, 28, 28, 1) / 255.0
x_test = x_test.reshape(-1, 28, 28, 1) / 255.0
y_train = to_categorical(y_train, 10)
y_test = to_categorical(y_test, 10)
```

2.2 Training and Evaluation

Both networks were trained for 10 epochs with a batch size of 64. The loss function used was **categorical crossentropy** with the Adam optimizer. A validation split of 20% was used during training.

Code:

```
fcnn_history = fcnn_model.fit(x_train, y_train, epochs
   =10, batch_size=64, validation_split=0.2)
cnn_history = cnn_model.fit(x_train, y_train, epochs=10,
    batch_size=64, validation_split=0.2)

fcnn_test_loss, fcnn_test_acc = fcnn_model.evaluate(
   x_test, y_test)
cnn_test_loss, cnn_test_acc = cnn_model.evaluate(x_test,
    y_test)
```

Results: The accuracy of FCNN and CNN on the test set were reported and visualized.

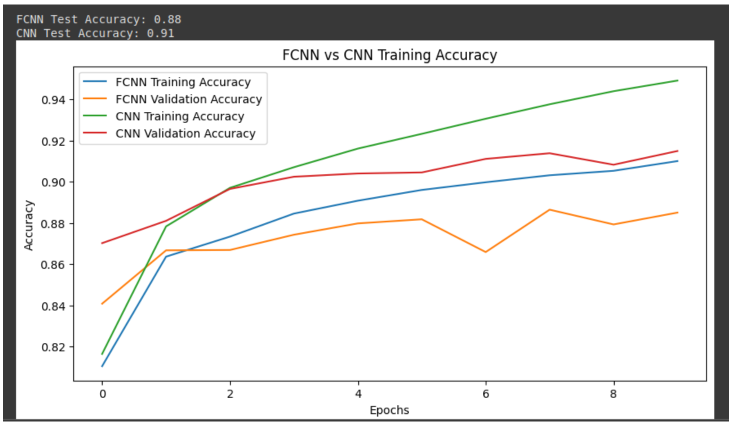


Figure 2: FCNN and CNN model Accuracy

3 Using Pre-trained MobileNetV2

3.1 Building the Model

A CNN was built with MobileNetV2 as the backbone. The pre-trained weights on ImageNet were used with the top 20 layers removed. Custom layers were added:

- **GlobalAveragePooling2D** to reduce dimensions.
- Two **Dense** layers with ReLU activation and Dropout.
- A final Dense layer with softmax activation for classification.

Code:

```
from tensorflow.keras.applications import MobileNetV2
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense,
   GlobalAveragePooling2D, Dropout
from tensorflow.keras.datasets import cifar10
from tensorflow.keras.utils import to_categorical
import tensorflow as tf

(x_train, y_train), (x_test, y_test) = cifar10.load_data
   ()

x_train, x_test = x_train / 255.0, x_test / 255.0

y_train = to_categorical(y_train, 10)
```

```
y_test = to_categorical(y_test, 10)

def data_generator(images, labels, batch_size,
   target_size=(64, 64)):

    while True:
        for start in range(0, len(images), batch_size):
            end = min(start + batch_size, len(images))
            batch_images = images[start:end]
            batch_labels = labels[start:end]
            resized_images = tf.image.resize(
               batch_images, target_size)
            yield resized_images, batch_labels


batch_size = 32
train_generator = data_generator(x_train, y_train,
   batch_size=batch_size)
test_generator = data_generator(x_test, y_test,
   batch_size=batch_size)

train_steps = len(x_train) // batch_size
test_steps = len(x_test) // batch_size

mobilenet_base = MobileNetV2(weights='imagenet',
   include_top=False, input_shape=(64, 64, 3), alpha
   =1.0)
mobilenet_base.trainable = False

model = Sequential([
    mobilenet_base,
    GlobalAveragePooling2D(),
    Dense(256, activation='relu'),
    Dropout(0.3),
    Dense(128, activation='relu'),
    Dropout(0.3),
    Dense(10, activation='softmax')
])

model.compile(optimizer='adam', loss='
   categorical_crossentropy', metrics=['accuracy'])

print("\nModel Summary:")
```

```
model.summary()
```

3.2 Transfer Learning and Fine-tuning

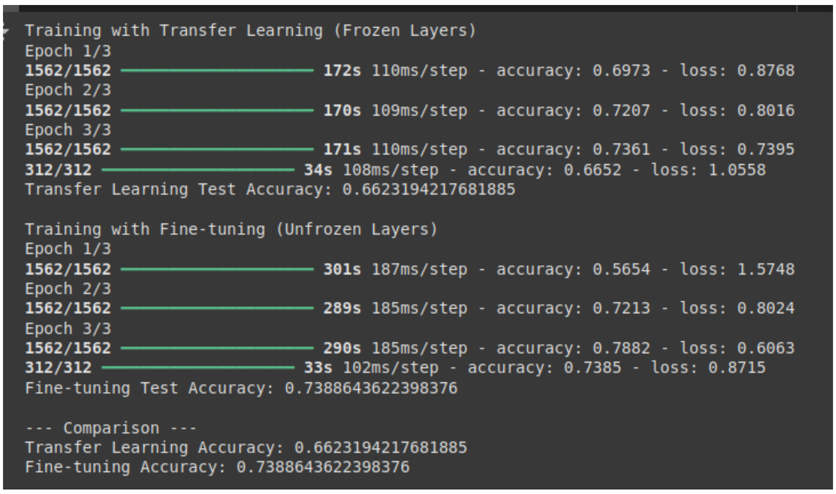
The model was trained with frozen layers for transfer learning and then fine-tuned by unfreezing the last 20 layers. **Code:**

```
print("Training with Transfer Learning (Frozen Layers)")
history_transfer = model.fit(
    train_generator,
    steps_per_epoch=train_steps,
    epochs=3,
    verbose=1
)

test_loss_transfer, test_acc_transfer = model.evaluate(
   test_generator, steps=test_steps)
print(f"Transfer Learning Test Accuracy: {
   test_acc_transfer}")

for layer in mobilenet_base.layers[:-20]:
    layer.trainable = False

for layer in mobilenet_base.layers[-20:]:
    layer.trainable = True

model.compile(optimizer=tf.keras.optimizers.Adam(
   learning_rate=1e-4),
                      loss='categorical_crossentropy',
                      metrics=['accuracy'])

print("\nTraining with Fine-tuning (Unfrozen Layers)")
history_finetune = model.fit(
    train_generator,
    steps_per_epoch=train_steps,
    epochs=3,
    verbose=1
)

test_loss_finetune, test_acc_finetune = model.evaluate(
   test_generator, steps=test_steps)
print(f"Fine-tuning Test Accuracy: {test_acc_finetune}")
```

```
print("\n--- Comparison ---")
print(f"Transfer Learning Accuracy: {test_acc_transfer
   }")
print(f"Fine-tuning Accuracy: {test_acc_finetune}")
```

4 Comparison of Results

1. The frozen layers yield a faster training process but might not achieve optimal accuracy due to limited adaptability to the new dataset.
Fine-tuning typically takes longer as more parameters are updated, but it often results in improved accuracy as the model refines its feature extraction for the specific dataset.
2. Fine-tuning generally results in higher accuracy, but at the cost of increased training time and potential overfitting if the dataset is small.



```
Training with Transfer Learning (Frozen Layers)
Epoch 1/3
1562/1562 ━━━━━━━━━━━━━━━━━━━━ 172s 110ms/step - accuracy: 0.6973 - loss: 0.8768
Epoch 2/3
1562/1562 ━━━━━━━━━━━━━━━━━━━━ 170s 109ms/step - accuracy: 0.7207 - loss: 0.8016
Epoch 3/3
1562/1562 ━━━━━━━━━━━━━━━━━━━━ 171s 110ms/step - accuracy: 0.7361 - loss: 0.7395
312/312 ━━━━━━━━━━━━━━━━━━━━ 34s 108ms/step - accuracy: 0.6652 - loss: 1.0558
Transfer Learning Test Accuracy: 0.6623194217681885

Training with Fine-tuning (Unfrozen Layers)
Epoch 1/3
1562/1562 ━━━━━━━━━━━━━━━━━━━━ 301s 187ms/step - accuracy: 0.5654 - loss: 1.5748
Epoch 2/3
1562/1562 ━━━━━━━━━━━━━━━━━━━━ 289s 185ms/step - accuracy: 0.7213 - loss: 0.8024
Epoch 3/3
1562/1562 ━━━━━━━━━━━━━━━━━━━━ 290s 185ms/step - accuracy: 0.7882 - loss: 0.6063
312/312 ━━━━━━━━━━━━━━━━━━━━ 33s 102ms/step - accuracy: 0.7385 - loss: 0.8715
Fine-tuning Test Accuracy: 0.7388643622398376

--- Comparison ---
Transfer Learning Accuracy: 0.6623194217681885
Fine-tuning Accuracy: 0.7388643622398376
```

Figure 3: Training vs Validation Accuracy for Different Models

5 Colab Link

https://colab.research.google.com/drive/1bxThvOZFqCCSZosgYymWo7K_6FCNobTT#scrollTo=tg09LobRNen7