EN3150 - Pattern Recognition

Assignment 03

Simple convolutional neural network to perform classification.

Group_Name: ByteBrains

INDEX NUMBER	NAME
200313P	KRISHNAKUMAR L.
200314U	KUGESAN S.
200582L	SANKAVI K.
200587G	SARUJAN K.

Github Repository Link

https://github.com/kugesan2000/EN3150_ByteBrains_A03.git

CNN for image classification

1.

Local Connectivity and Parameter Sharing:
CNNs capture local patterns using convolutional layers and share weights, making them efficient.

Spatial Hierarchies and Pooling Layers:
CNNs use pooling layers for down-sampling and create spatial hierarchies, improving robustness.

Translation Invariance:
CNNs identify patterns regardless of position, enhancing translation invariance.

Parameter Efficiency:
CNNs are more parameter-efficient than fully connected networks, reducing overfitting risk.

Handling Different Resolutions:
CNNs adapt to various image resolutions without a fixed input size.

Specialized Architectures:
Architectures like ResNet address challenges in training very deep networks.

Pre-trained Models and Transfer Learning:
Pre-trained CNN models on large datasets can be fine-tuned for specific tasks.

Efficient GPU Utilization:

CNNs are highly parallelizable, making them computationally efficient for GPUs.

6.

kernel sizes
m1 = (3, 3)
m2 = (3, 3)

filter sizes

x1 = 32

x2 = 64

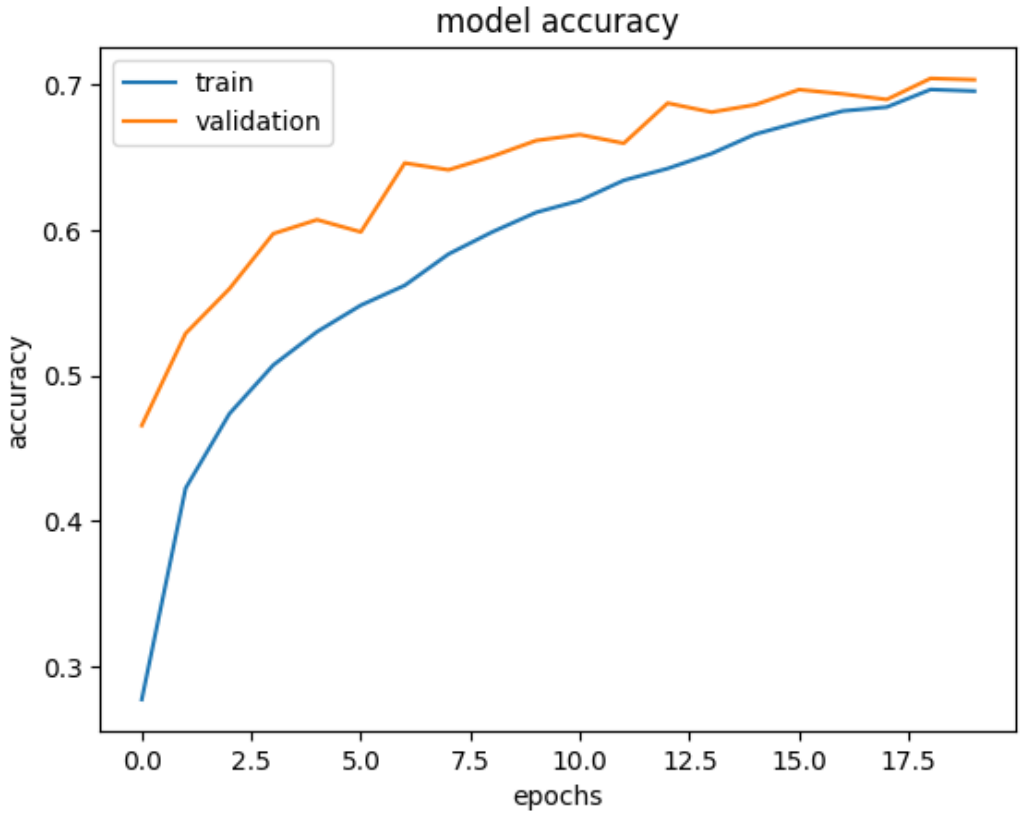
size of the fully connected layer

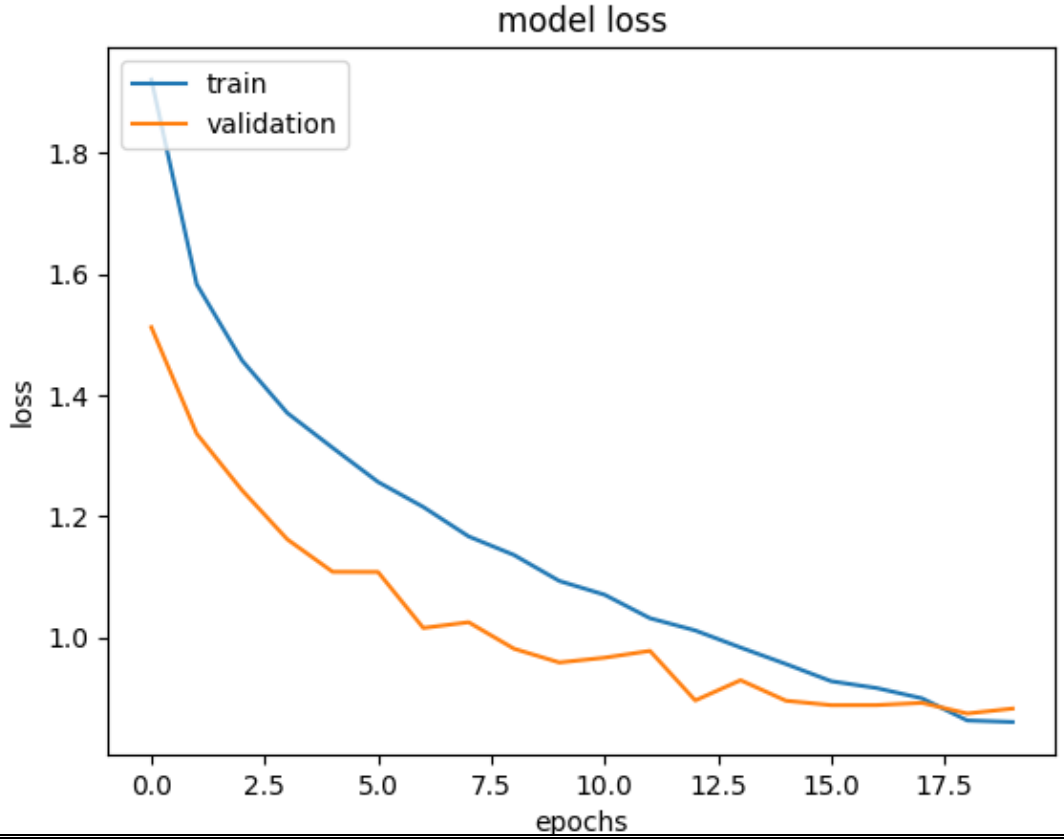
x3 = 64

Dropout Rate 0.7

```
Model: "sequential_4"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_10 (Conv2D)          (None, 30, 30, 32)        896

 max_pooling2d_8 (MaxPoolin  (None, 15, 15, 32)        0
 g2D)

 conv2d_11 (Conv2D)          (None, 13, 13, 64)        18496

 max_pooling2d_9 (MaxPoolin  (None, 6, 6, 64)          0
 g2D)

 conv2d_12 (Conv2D)          (None, 4, 4, 64)          36928

 flatten_4 (Flatten)         (None, 1024)              0

 dense_8 (Dense)             (None, 64)                65600

 dropout_4 (Dropout)         (None, 64)                0

 dense_9 (Dense)             (None, 10)                650

=================================================================
Total params: 122570 (478.79 KB)
Trainable params: 122570 (478.79 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

7.


model accuracy
train
validation
accuracy
epochs
0.3
0.4
0.5
0.6
0.7
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5


model loss
train
validation
loss
epochs
1.0
1.2
1.4
1.6
1.8
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5

8.

Adam optimizer combines advantages of both momentum and RMSprop, providing faster convergence and adaptability to different learning rates. It is well-suited for various optimization problems.

We opted for the Adam optimizer over SGD due to its adaptive learning rate mechanism, which helps handle varying gradients during training. Adam combines the benefits of momentum and root mean square propagation, dynamically adjusting learning rates for each parameter. This adaptability allows faster convergence, mitigating issues like vanishing or exploding gradients. Additionally, Adam often requires less manual tuning of hyperparameters compared to SGD, making it a practical choice for optimizing the complex and high-dimensional parameter space of neural networks. Overall, Adam's efficiency and robustness in optimizing non-convex objective functions make it a preferred optimizer for training neural networks.

9.

Sparse categorical crossentropy is suitable for multi-class classification problems with integer labels (like CIFAR-10). It avoids the need for one-hot encoding and is computationally efficient.

We selected sparse categorical crossentropy as the loss function because it is suitable for multi-class classification tasks where each instance belongs to only one class. In the CIFAR-10 dataset, each image is associated with a single class label, making sparse categorical crossentropy a natural choice. This loss function efficiently computes the crossentropy between the predicted and true class distributions, encouraging the model to assign high probabilities to the correct class. It eliminates the need for one-hot encoding of class labels, simplifying the implementation and reducing memory requirements. Sparse categorical crossentropy is particularly well-suited for scenarios with mutually exclusive classes, aligning with the nature of image classification tasks.

9.

Evaluate the Model

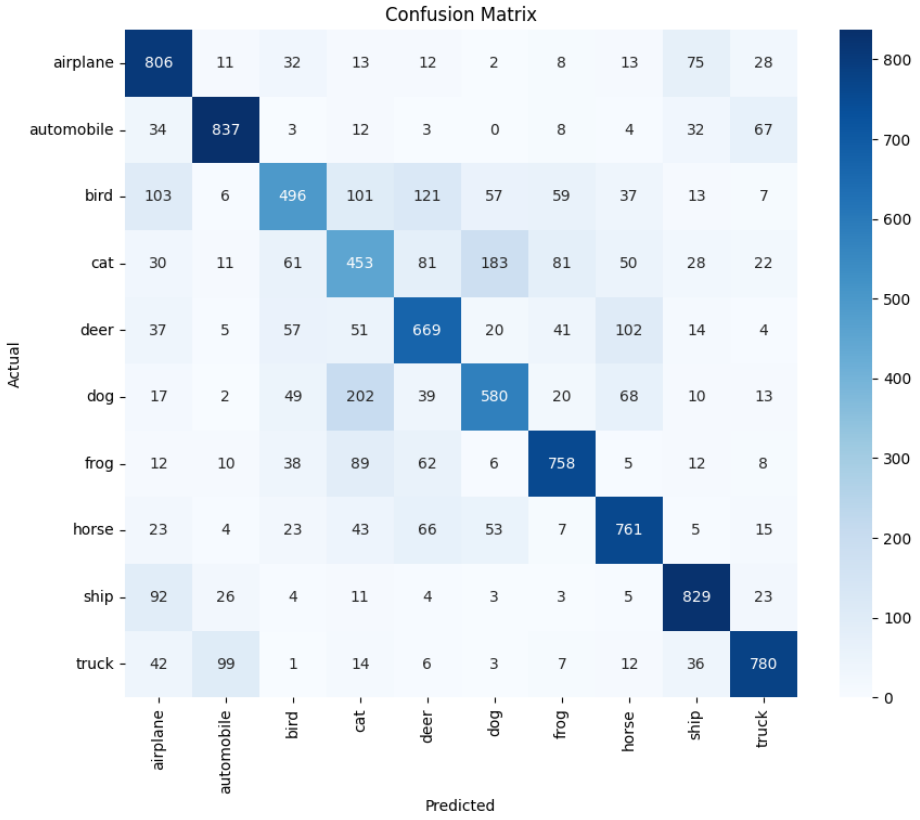
Train/Test Accuracy

+ Code + Markdown

```
test_loss, test_accuracy = model.evaluate(x_test, y_test)
print(f"Test accuracy: {test_accuracy}")
```

✓ 3.5s

```
313/313 [==============================] - 3s 8ms/step - loss: 0.9062 - accuracy: 0.6969
Test accuracy: 0.6969000101089478
```



precision and recall

```
from sklearn.metrics import precision_score, recall_score
#since we're doing multi class classification 'weighted' is used
precision = precision_score(y_test, y_pred_classes, average= 'weighted')
recall = recall_score(y_test, y_pred_classes, average= 'weighted')
print(f'precision :{precision}')
print(f'recall :{recall}')
```

✓ 0.0s

```
precision :0.6955101959336238
recall :0.6969
```

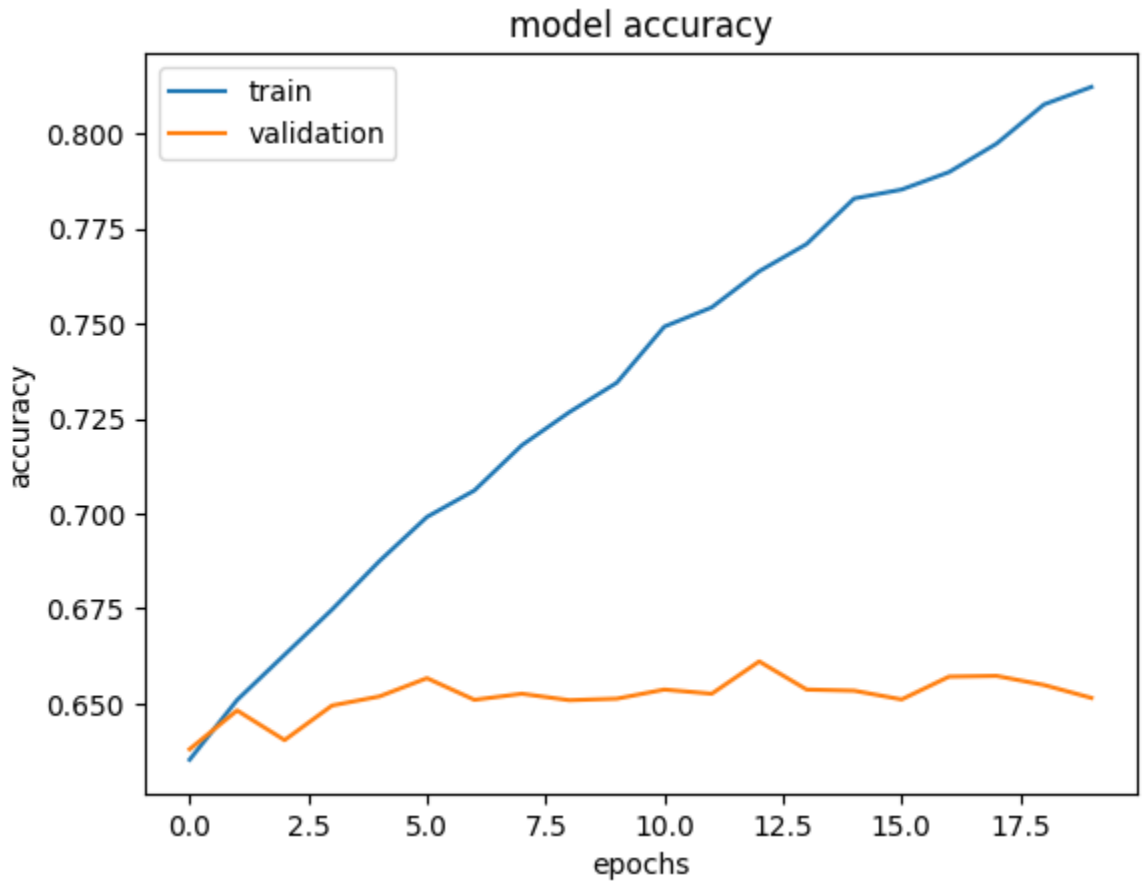
10.

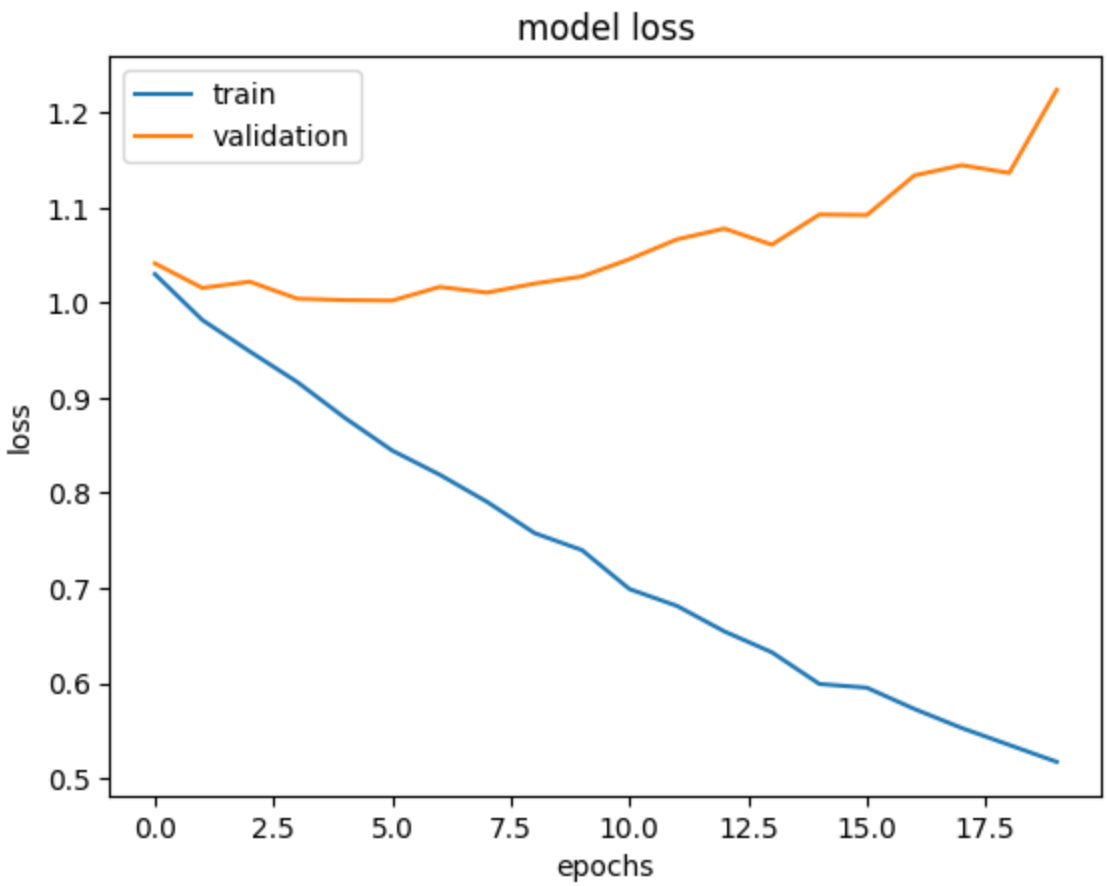
The model's performance is adversely affected when using a learning rate of 0.1, displaying the poorest results characterized by the highest test loss and the lowest accuracy. This large learning rate proves excessive for the given problem, hindering convergence and causing the model to struggle in extracting meaningful features from the data.

The selection of the learning rate plays a crucial role in determining the model's efficacy. A learning rate of 0.0001 stands out as a well-performing choice, indicating that smaller learning rates are more suitable for this particular task. On the other hand, learning rates of 0.001 and 0.01 prove to be too high for effective convergence, resulting in increased losses and decreased accuracies. Notably, a learning rate of 0.1 is excessively large and yields the poorest performance. This underscores the significance of fine-tuning the learning rate to achieve optimal training and emphasizes the sensitivity of the model's performance to this hyperparameter.

Comparing our network with state-of-the-art networks

DenseNet



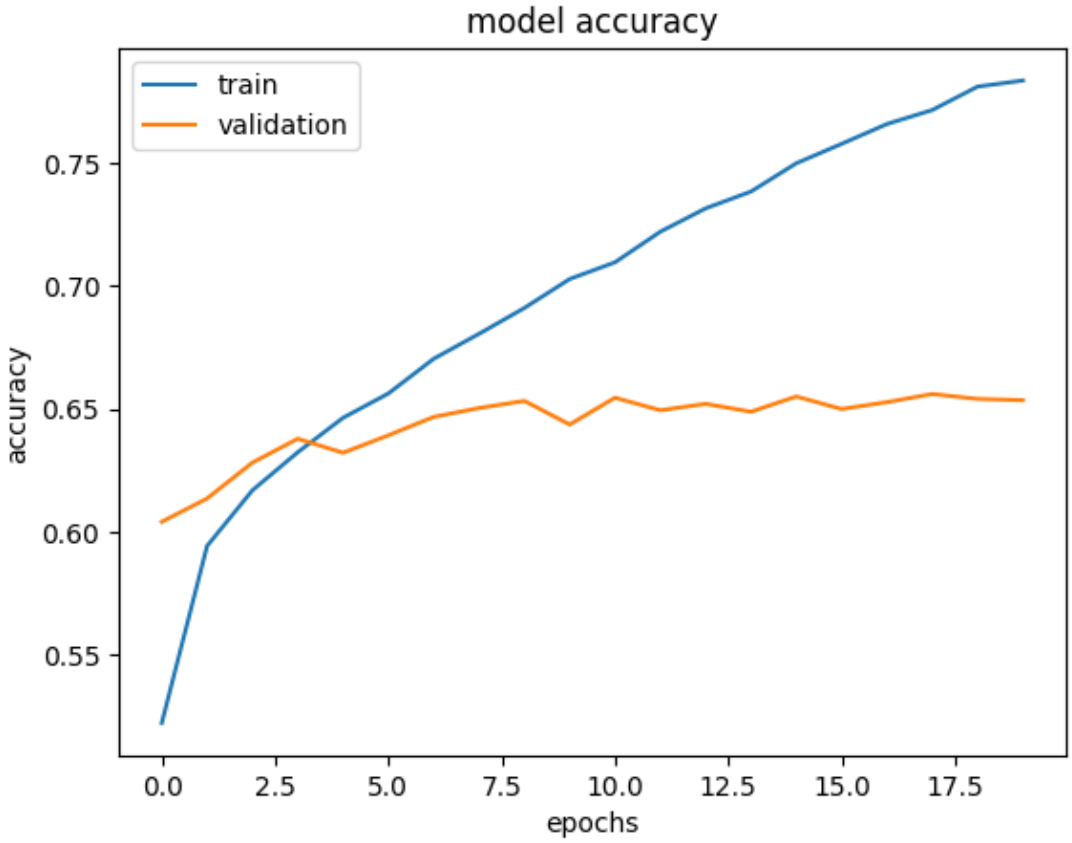


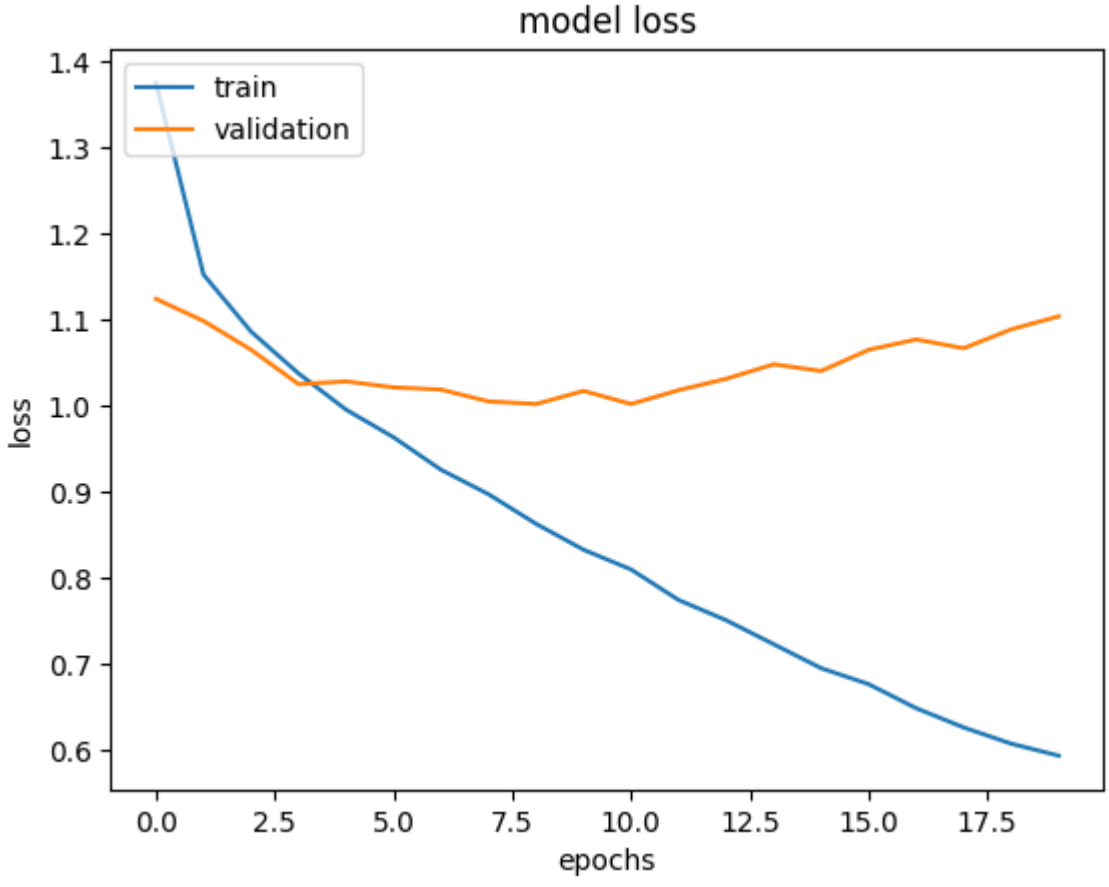

```
test_loss, test_accuracy = model.evaluate(x_test, y_test)
print(f"Test accuracy: {test_accuracy}")
```

✓ 35.2s

```
313/313 [==============================] - 33s 104ms/step - loss: 1.2142 - accuracy: 0.6566
Test accuracy: 0.6565999984741211
```

ResNet


model accuracy
train
validation
accuracy
0.75
0.70
0.65
0.60
0.55
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
epochs


model loss
train
validation
loss
1.4
1.3
1.2
1.1
1.0
0.9
0.8
0.7
0.6
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
epochs

```
test_loss, test_accuracy = model.evaluate(x_test, y_test)
print(f"Test accuracy: {test_accuracy}")
```

✓ 37.7s

```
313/313 [==============================] - 35s 113ms/step - loss: 1.1164 - accuracy: 0.6549
Test accuracy: 0.6549000144004822
```

Results

In comparing the performance of the custom CNN, ResNet, and DenseNet models, it is evident that the custom CNN model achieved superior results. It exhibited the lowest test loss and the highest accuracy, along with balanced precision and recall. On the other hand, both ResNet and DenseNet models struggled, leading to decreased accuracy and increased test loss. Specifically, ResNet performed relatively better compared to DenseNet, showcasing **the importance of appropriate hyperparameter tuning for optimal model performance**. Overall, the custom CNN model demonstrated its effectiveness in achieving better classification results on the given dataset.

18.

Criteria	Custom Model	Pre-trained Model
Advantages	Tailored to specific task	Feature learning from large datasets
	Domain knowledge integration	Transfer learning potential
Trade-offs	Data requirement	Task specificity
	Computational resources	Domain mismatch
Limitations	May not capture intricate features effectively	Limited adaptability to specific task nuances
	May struggle with generalization	Larger model size

Github Repository Link
https://github.com/kugesan2000/EN3150_ByteBrains_A03.git