Simple Convolutional Neural Network to Perform Classification

Team Pixels

Department of Electronic and Telecommunication Engineering

University of Moratuwa

Name	Index Number
GUNAWARDENA M.N.	200201V
INDRAPALA S.A.	200232P
PERERA N.W.P.R.A.	200462U
YALEGAMA M.M.O.A.B.	200740V

This report is submitted as a partial fulfilment of the module

EN3150 – Pattern Recognition

Github Repository: *https://github.com/oshanyalegama/PR-Assignment-03*.

03rd December 2023

Contents

1 *CNNs vs MLPs/ NNs*

- MLPs lack spatial understanding, do not capture local patterns and cannot recognize patterns regardless of their position.
- CNNs preserve the spatial structure of data unlike MLPs.
- CNNs are translation invariant and can recognize patterns regardless of their position.
- CNNs have heirachical feature extraction which is well suited for capturing complex patterns in data.

2 Parameters

1. Convolutional Layer 1
   - Type: Conv2D
   - Number of Filters: 32
   - Kernel Size: (3, 3)
   - Activation Function: ReLU
   - Padding: 'same'
   - Input Shape: (32, 32, 3)
   - Batch Normalization follows the Conv2D layer.
2. Convolutional Layer 2
   - Type: Conv2D
   - Number of Filters: 32
   - Kernel Size: (3, 3)
   - Activation Function: ReLU
   - Padding: 'same'
   - Batch Normalization follows the Conv2D layer.
3. Pooling Layer 1
   - Type: MaxPooling2D
   - Pool Size: (2, 2)
4. Dropout Layer 1
   - Type: Dropout
   - Dropout Rate: 0.25
5. Convolutional Layer 3
   - Type: Conv2D
   - Number of Filters: 64

- Kernel Size: (3, 3)
- Activation Function: ReLU
- Padding: 'same'
- Batch Normalization follows the Conv2D layer.

6. Convolutional Layer 4
   - Type: Conv2D
   - Number of Filters: 64
   - Kernel Size: (3, 3)
   - Activation Function: ReLU
   - Padding: 'same'
   - Batch Normalization follows the Conv2D layer.
7. Pooling Layer 2
   - Type: MaxPooling2D
   - Pool Size: (2, 2)
8. Dropout Layer 2
   - Type: Dropout
   - Dropout Rate: 0.25
9. Convolutional Layer 5
   - Type: Conv2D
   - Number of Filters: 128
   - Kernel Size: (3, 3)
   - Activation Function: ReLU
   - Padding: 'same'
   - Batch Normalization follows the Conv2D layer.
10. Convolutional Layer 6
    - Type: Conv2D
    - Number of Filters: 128
    - Kernel Size: (3, 3)
    - Activation Function: ReLU
    - Padding: 'same'
    - Batch Normalization follows the Conv2D layer.
11. Pooling Layer 3
    - Type: MaxPooling2D
    - Pool Size: (2, 2)

12. Dropout Layer 3
    - Type: Dropout
    - Dropout Rate: 0.25
13. Flatten Layer
    - Type: Flatten
14. Fully Connected Layer (Dense Layer)
    - Type: Dense
    - Number of Units: 128
    - Activation Function: ReLU
15. Dropout Layer 4
    - Type: Dropout
    - Dropout Rate: 0.25
16. Output Layer (Dense Layer)
    - Type: Dense
    - Number of Units: 10 (for 10 classes)
    - Activation Function: Softmax

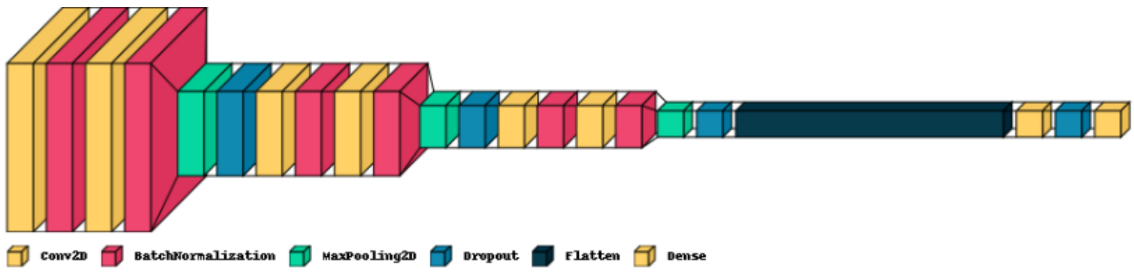


Figure 1: Model Overview

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
Conv_1 (Conv2D)              (None, 32, 32, 32)        896
_________________________________________________________________
Batch_Norm_1 (BatchNormaliza (None, 32, 32, 32)        128
_________________________________________________________________
Conv_2 (Conv2D)              (None, 32, 32, 32)        9248
_________________________________________________________________
Batch_Norm_2 (BatchNormaliza (None, 32, 32, 32)        128
_________________________________________________________________
Max_Pool_1 (MaxPooling2D)    (None, 16, 16, 32)        0
_________________________________________________________________
Dropout_1 (Dropout)          (None, 16, 16, 32)        0
_________________________________________________________________
Conv_3 (Conv2D)              (None, 16, 16, 64)        18496
_________________________________________________________________
Batch_Norm_3 (BatchNormaliza (None, 16, 16, 64)        256
_________________________________________________________________
Conv_4 (Conv2D)              (None, 16, 16, 64)        36928
_________________________________________________________________
Batch_Norm_4 (BatchNormaliza (None, 16, 16, 64)        256
_________________________________________________________________
Max_Pool_2 (MaxPooling2D)    (None, 8, 8, 64)          0
_________________________________________________________________
Dropout_2 (Dropout)          (None, 8, 8, 64)          0
_________________________________________________________________
Conv_5 (Conv2D)              (None, 8, 8, 128)         73856
_________________________________________________________________
Batch_Norm_5 (BatchNormaliza (None, 8, 8, 128)         512
_________________________________________________________________
Conv_6 (Conv2D)              (None, 8, 8, 128)         147584
_________________________________________________________________
Batch_Norm_6 (BatchNormaliza (None, 8, 8, 128)         512
_________________________________________________________________
Max_Pool_3 (MaxPooling2D)    (None, 4, 4, 128)         0
_________________________________________________________________
Dropout_3 (Dropout)          (None, 4, 4, 128)         0
_________________________________________________________________
Flatten_Layer (Flatten)      (None, 2048)              0
_________________________________________________________________
Fully_Connected_Layer (Dense (None, 128)               262272
_________________________________________________________________
Dropout_4 (Dropout)          (None, 128)               0
_________________________________________________________________
Output_Layer (Dense)         (None, 10)                1290
=================================================================
Total params: 552,362
Trainable params: 551,466
Non-trainable params: 896
```

Figure 2: Model Summary

3 Building the Model

```
#Building model computational graph
model = Sequential()
model.add(Conv2D(32, (3,3), activation = 'relu', padding = 'same', input_shape = (32,32,3),name='Conv_1'))
model.add(BatchNormalization(name='Batch_Norm_1'))
model.add(Conv2D(32, (3,3), activation = 'relu', padding = 'same',name='Conv_2'))
model.add(BatchNormalization(name='Batch_Norm_2'))
model.add(MaxPooling2D(pool_size = (2,2), strides=(2,2),name='Max_Pool_1'))
model.add(Dropout(0.25, name='Dropout_1'))
model.add(Conv2D(64, (3,3), activation = 'relu', padding = 'same',name='Conv_3'))
model.add(BatchNormalization(name='Batch_Norm_3'))
model.add(Conv2D(64, (3,3), activation = 'relu', padding = 'same',name='Conv_4'))
model.add(BatchNormalization(name='Batch_Norm_4'))
model.add(MaxPooling2D(pool_size = (2,2),strides=(2,2),name='Max_Pool_2'))
model.add(Dropout(0.25, name='Dropout_2'))
model.add(Conv2D(128, (3,3), activation = 'relu', padding = 'same',name='Conv_5'))
model.add(BatchNormalization(name='Batch_Norm_5'))
model.add(Conv2D(128, (3,3), activation = 'relu', padding = 'same',name='Conv_6'))
model.add(BatchNormalization(name='Batch_Norm_6'))
model.add(MaxPooling2D(pool_size = (2,2),strides=(2,2),name='Max_Pool_3'))
model.add(Dropout(0.25, name='Dropout_3'))
model.add(Flatten(name='Flatten_Layer'))
model.add(Dense(128, activation = 'relu',name='Fully_Connected_Layer'))
model.add(Dropout(0.25,name='Dropout_4'))
model.add(Dense(10, activation = 'softmax',name='Output_Layer'))
```

Figure 3: Model

4 Adam Optimizer vs SGD

- Adaptive Learning Rates: Adam dynamically adjusts the learning rates for each parameter during training. It maintains separate learning rates for each parameter, and these rates are updated based on the historical gradients. This adaptability can lead to faster convergence and better performance compared to SGD, which uses a fixed learning rate
- Momentum: Adam incorporates the concept of momentum, similar to SGD with momentum. Momentum helps accelerate the optimization process, especially in the presence of noisy gradients or sparse data. It allows the optimizer to keep moving in the right direction, even when gradients are noisy or have varying magnitudes.
- Reduced Sensitivity to Hyperparameters: Adam is less sensitive to the choice of learning rate hyperparameter compared to SGD. While SGD often requires careful tuning of the learning rate, Adam is more forgiving and tends to work well with default hyperparameters.

5 Sparse Categorical Cross Entropy

- In the CIFAR-10 dataset, each image is associated with a class label represented as an integer (e.g., 0 for 'Airplane', 1 for 'Automobile', and so on). The labels are not one-hot encoded; they are single integers indicating the class of the respective image.
- The loss function 'sparse_categorical_crossentropy' is designed for classification problems where the labels are integers. It internally performs the conversion from integer labels to one-hot

encoded vectors, making it convenient when your labels are integers instead of one-hot encoded arrays.

- If we used 'categoricalcrossentropy' instead, we would need to one-hot encode our labels explicitly using to_categorical, as the function expects one-hot encoded labels.

6 Training

For a learning rate of 0.0001, the results are as follows,

```
Epoch 1/20
625/625 [==============================] - 5s 8ms/step - loss: 2.0208 - accuracy: 0.2873 - val_loss: 1.9764 - val_accuracy: 0.3143
Epoch 2/20
625/625 [==============================] - 4s 7ms/step - loss: 1.6067 - accuracy: 0.4159 - val_loss: 1.4489 - val_accuracy: 0.4890
Epoch 3/20
625/625 [==============================] - 4s 7ms/step - loss: 1.4497 - accuracy: 0.4781 - val_loss: 1.6079 - val_accuracy: 0.4779
Epoch 4/20
625/625 [==============================] - 4s 7ms/step - loss: 1.3296 - accuracy: 0.5245 - val_loss: 1.3893 - val_accuracy: 0.5368
Epoch 5/20
625/625 [==============================] - 4s 7ms/step - loss: 1.2322 - accuracy: 0.5595 - val_loss: 1.2144 - val_accuracy: 0.5807
Epoch 6/20
625/625 [==============================] - 4s 7ms/step - loss: 1.1485 - accuracy: 0.5906 - val_loss: 1.2104 - val_accuracy: 0.5926
Epoch 7/20
625/625 [==============================] - 4s 7ms/step - loss: 1.0732 - accuracy: 0.6172 - val_loss: 1.0257 - val_accuracy: 0.6420
Epoch 8/20
625/625 [==============================] - 4s 7ms/step - loss: 1.0005 - accuracy: 0.6438 - val_loss: 1.0696 - val_accuracy: 0.6349
Epoch 9/20
625/625 [==============================] - 4s 7ms/step - loss: 0.9461 - accuracy: 0.6632 - val_loss: 0.9781 - val_accuracy: 0.6639
Epoch 10/20
625/625 [==============================] - 4s 7ms/step - loss: 0.9033 - accuracy: 0.6809 - val_loss: 0.9216 - val_accuracy: 0.6811
Epoch 11/20
625/625 [==============================] - 4s 7ms/step - loss: 0.8578 - accuracy: 0.6975 - val_loss: 0.8773 - val_accuracy: 0.6992
Epoch 12/20
625/625 [==============================] - 4s 7ms/step - loss: 0.8243 - accuracy: 0.7093 - val_loss: 0.8432 - val_accuracy: 0.7104
Epoch 13/20
625/625 [==============================] - 5s 7ms/step - loss: 0.7808 - accuracy: 0.7260 - val_loss: 0.7554 - val_accuracy: 0.7407
Epoch 14/20
625/625 [==============================] - 4s 7ms/step - loss: 0.7584 - accuracy: 0.7340 - val_loss: 0.7245 - val_accuracy: 0.7510
Epoch 15/20
625/625 [==============================] - 4s 7ms/step - loss: 0.7244 - accuracy: 0.7456 - val_loss: 0.7039 - val_accuracy: 0.7563
Epoch 16/20
625/625 [==============================] - 4s 7ms/step - loss: 0.7002 - accuracy: 0.7516 - val_loss: 0.7073 - val_accuracy: 0.7574
Epoch 17/20
625/625 [==============================] - 4s 7ms/step - loss: 0.6760 - accuracy: 0.7606 - val_loss: 0.7412 - val_accuracy: 0.7426
Epoch 18/20
625/625 [==============================] - 4s 7ms/step - loss: 0.6477 - accuracy: 0.7695 - val_loss: 0.7139 - val_accuracy: 0.7547
Epoch 19/20
625/625 [==============================] - 4s 7ms/step - loss: 0.6285 - accuracy: 0.7774 - val_loss: 0.6717 - val_accuracy: 0.7689
Epoch 20/20
625/625 [==============================] - 4s 7ms/step - loss: 0.6054 - accuracy: 0.7863 - val_loss: 0.6539 - val_accuracy: 0.7776
```

Figure 4: Training Results

7 Evaluation and Results

7.1 Learning Rate: 0.0001

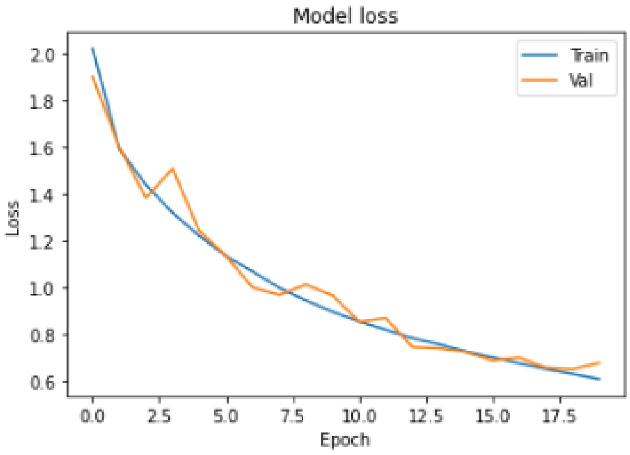


Figure 5: Loss

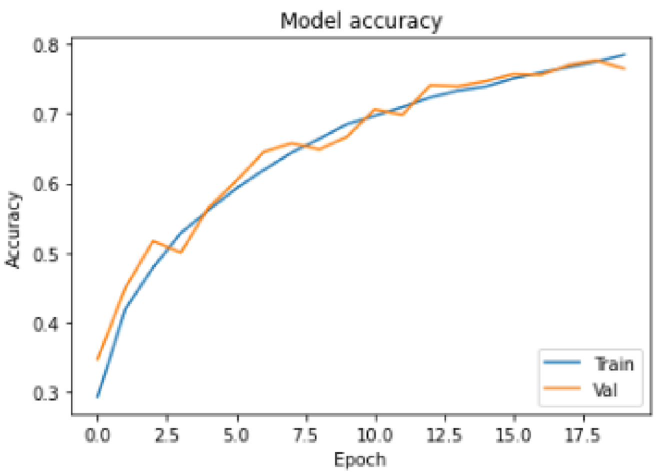


Figure 6: Accuracy

```
313/313 - 1s - loss: 0.6997 - accuracy: 0.7603
Test Accuracy: 76.03%
Test Loss: 0.6997
```

Figure 7: Performance I

```
Classification Report:
              precision    recall  f1-score   support

    Airplane       0.84      0.75      0.79      1000
  Automobile       0.92      0.84      0.88      1000
        Bird       0.73      0.58      0.65      1000
         Cat       0.63      0.53      0.58      1000
        Deer       0.61      0.84      0.71      1000
         Dog       0.73      0.60      0.66      1000
        Frog       0.67      0.91      0.78      1000
       Horse       0.86      0.76      0.81      1000
        Ship       0.84      0.92      0.88      1000
       Truck       0.83      0.87      0.85      1000

    accuracy                           0.76     10000
   macro avg       0.77      0.76      0.76     10000
weighted avg       0.77      0.76      0.76     10000


<Figure size 1800x1800 with 0 Axes>
```

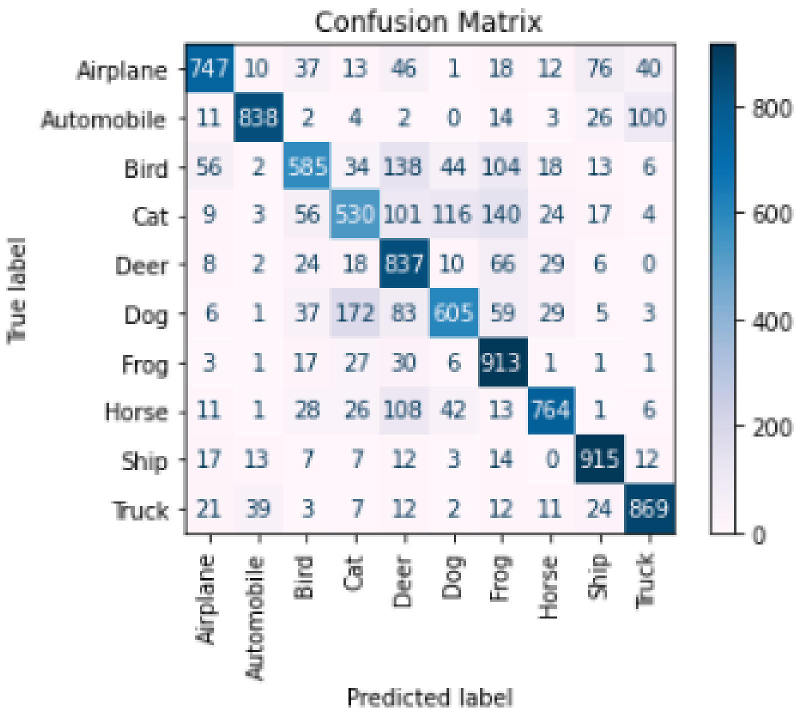


Figure 8: Performance II

7.2 Learning Rate: 0.001

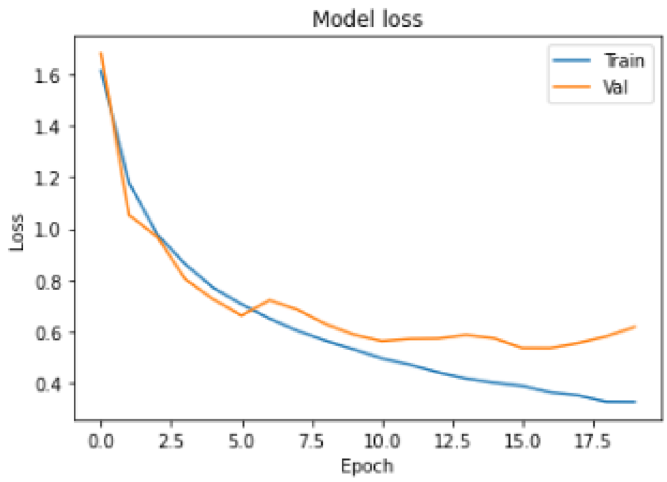


Figure 9: Loss

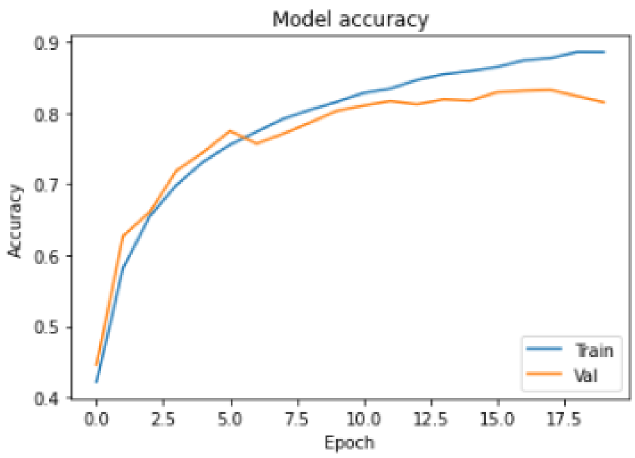


Figure 10: Accuracy

```
313/313 - 1s - loss: 0.6458 - accuracy: 0.8077
Test Accuracy: 80.77%
Test Loss: 0.6458
```

Figure 11: Performance I

```
Classification Report:
              precision    recall  f1-score   support

    Airplane       0.83      0.82      0.82      1000
  Automobile       0.92      0.92      0.92      1000
        Bird       0.83      0.64      0.72      1000
         Cat       0.70      0.57      0.63      1000
        Deer       0.69      0.87      0.77      1000
         Dog       0.76      0.70      0.73      1000
        Frog       0.72      0.92      0.81      1000
       Horse       0.88      0.84      0.86      1000
        Ship       0.88      0.91      0.89      1000
       Truck       0.91      0.88      0.89      1000

    accuracy                           0.81     10000
   macro avg       0.81      0.81      0.81     10000
weighted avg       0.81      0.81      0.81     10000


<Figure size 1800x1800 with 0 Axes>
```

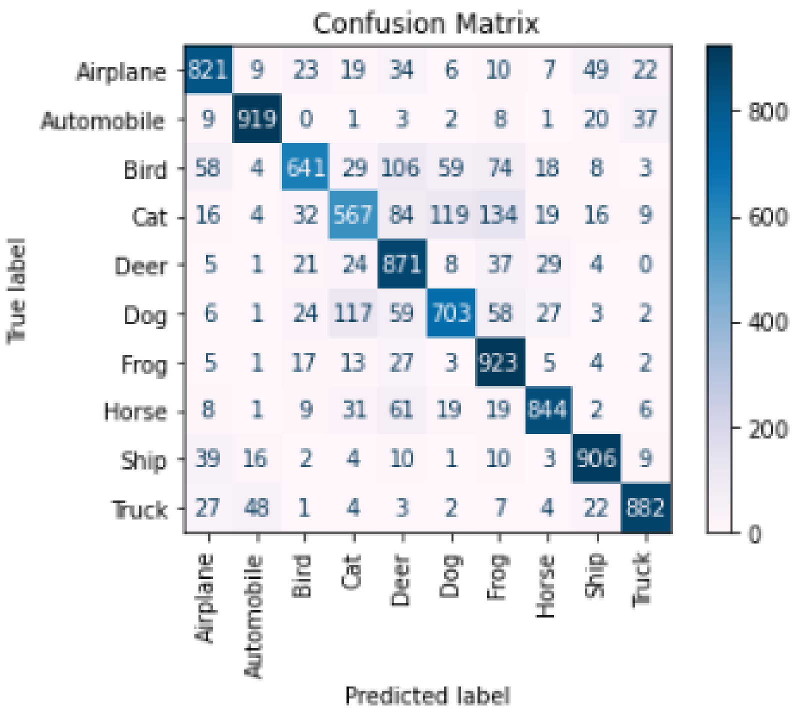


Figure 12: Performance II

7.3 Learning Rate: 0.01

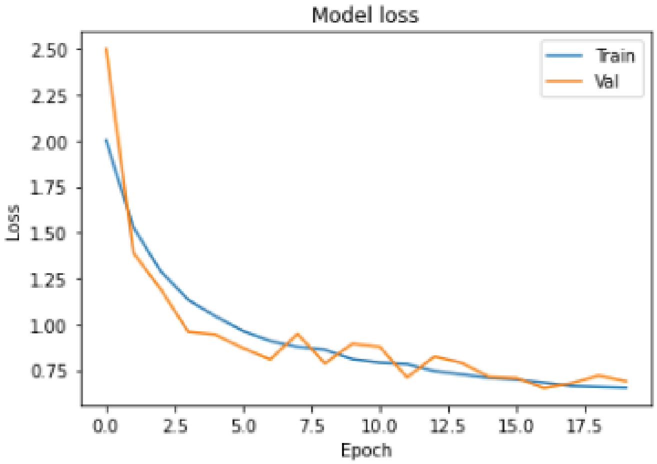


Figure 13: Loss

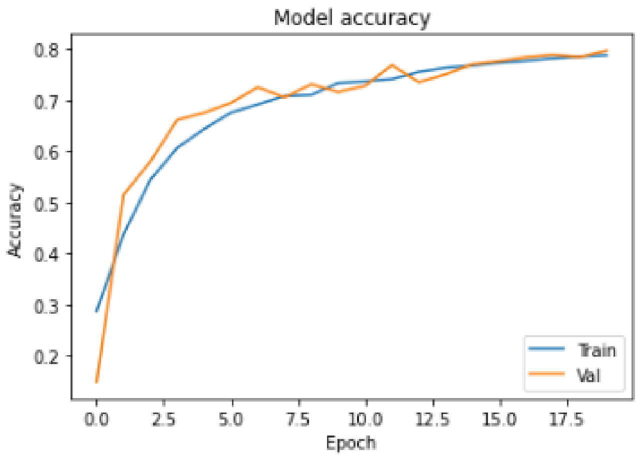


Figure 14: Accuracy

```
313/313 - 1s - loss: 0.7460 - accuracy: 0.7921
Test Accuracy: 79.21%
Test Loss: 0.7460
```

Figure 15: Performance I

```
Classification Report:
              precision    recall  f1-score   support

    Airplane       0.73      0.89      0.80      1000
  Automobile       0.86      0.94      0.90      1000
        Bird       0.72      0.68      0.70      1000
         Cat       0.61      0.64      0.62      1000
        Deer       0.78      0.76      0.77      1000
         Dog       0.80      0.60      0.69      1000
        Frog       0.77      0.88      0.82      1000
       Horse       0.87      0.82      0.85      1000
        Ship       0.87      0.91      0.89      1000
       Truck       0.95      0.81      0.87      1000

    accuracy                           0.79     10000
   macro avg       0.80      0.79      0.79     10000
weighted avg       0.80      0.79      0.79     10000


<Figure size 1800x1800 with 0 Axes>
```

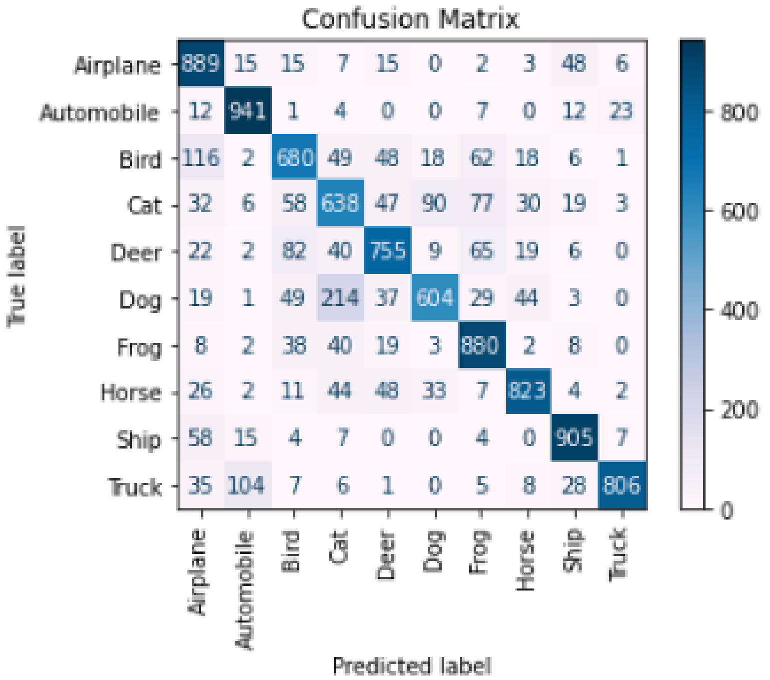


Figure 16: Performance II

7.4 Learning Rate: 0.1

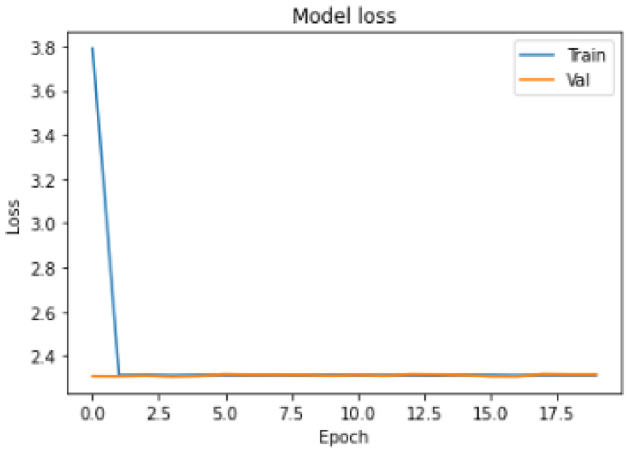


Figure 17: Loss

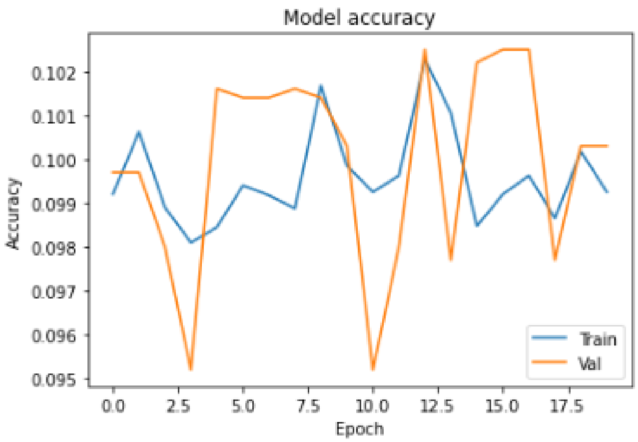


Figure 18: Accuracy

```
313/313 - 1s - loss: 2.3131 - accuracy: 0.1000
Test Accuracy: 10.00%
Test Loss: 2.3131
```

Figure 19: Performance I

```
Classification Report:
              precision    recall  f1-score   support

    Airplane       0.00      0.00      0.00      1000
  Automobile       0.00      0.00      0.00      1000
        Bird       0.00      0.00      0.00      1000
         Cat       0.00      0.00      0.00      1000
        Deer       0.00      0.00      0.00      1000
         Dog       0.00      0.00      0.00      1000
        Frog       0.00      0.00      0.00      1000
       Horse       0.00      0.00      0.00      1000
        Ship       0.10      1.00      0.18      1000
       Truck       0.00      0.00      0.00      1000

    accuracy                           0.10     10000
   macro avg       0.01      0.10      0.02     10000
weighted avg       0.01      0.10      0.02     10000
```

```
/opt/conda/lib/python3.7/site-packages/sklearn/metrics/_cl
  _warn_prf(average, modifier, msg_start, len(result))
<Figure size 1800x1800 with 0 Axes>
```

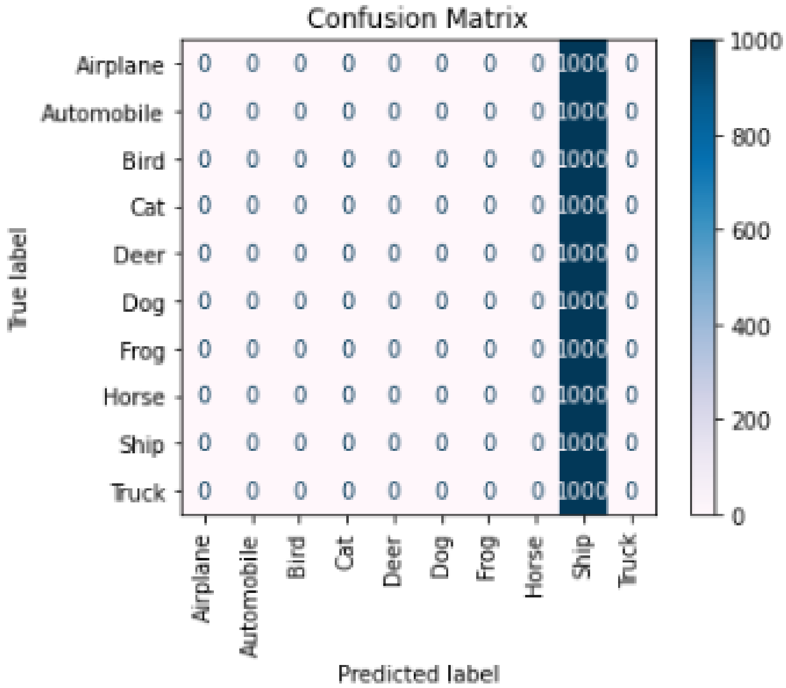


Figure 20: Performance II

8 State of the Art Networks

For our submission we selected the ResNet model and DenseNet model which were pretrained with Imagenet. These were imported by using the tensorflow.keras.applications and the Imagenet weights are readily available in this library. Both models were trained using the same dataset as before. Afterwards the training and validation losses for each epoch was recorded. And finally they were evaluated to determine each of their testing accuracy. The testing accuracies that were determined are 0.802 for ResNet and 0.848 for DenseNet for training done for the same number of epochs as the previous architecture. These values are significantly higher than before. The relevant code is demonstrated in the following sections.

8.1 ResNet

8.1.1 Training and Fine Tuning

```
from tensorflow.keras import layers, models
from tensorflow.keras.applications import ResNet50
from tensorflow.keras.callbacks import ModelCheckpoint

# Load pre-trained ResNet50 model without top classification layer
base_model_res = ResNet50(weights='imagenet', include_top=False, input_shape=(32, 32, 3))

# Fine-tuning the entire model
for layer in base_model_res.layers:
    layer.trainable = True

# using the pre trained model as a feature extractor
model_res = models.Sequential([
    base_model_res,
    layers.Flatten(),
    layers.Dense(512, activation='relu'),
    layers.Dropout(0.5),
    layers.Dense(10, activation='softmax')
])

model_res.compile(loss = loss, optimizer = opt, metrics = metrics)

# Fine-tune the model
checkpoint_res = ModelCheckpoint('cifar10_fine_tuned_resnet.h5', save_best_only=True)
history = model_res.fit(x_train, y_train, batch_size = 64 , epochs = 20, validation_split = 0.2, callbacks=[checkpoint_res])
```

Figure 21: Model

8.1.2 Evaluating Model

```
# Evaluate the model
test_loss_res, test_acc_res =model_res.evaluate(x_test, y_test)
```

Figure 22: Evaluating Model

8.1.3 Losses

```
Epoch 1/20
625/625 [==============================] - 27s 43ms/step - loss: 1.2723 - accuracy: 0.5866 - val_loss: 2.8123 - val_accuracy: 0.1488
Epoch 2/20
625/625 [==============================] - 26s 42ms/step - loss: 0.7178 - accuracy: 0.7577 - val_loss: 0.6775 - val_accuracy: 0.7723
Epoch 3/20
625/625 [==============================] - 26s 42ms/step - loss: 0.5079 - accuracy: 0.8293 - val_loss: 0.6584 - val_accuracy: 0.7870
Epoch 4/20
625/625 [==============================] - 25s 41ms/step - loss: 0.3766 - accuracy: 0.8723 - val_loss: 0.6911 - val_accuracy: 0.7822
Epoch 5/20
625/625 [==============================] - 26s 41ms/step - loss: 0.2765 - accuracy: 0.9054 - val_loss: 0.7172 - val_accuracy: 0.7936
Epoch 6/20
625/625 [==============================] - 25s 41ms/step - loss: 0.1995 - accuracy: 0.9315 - val_loss: 0.7551 - val_accuracy: 0.7956
Epoch 7/20
625/625 [==============================] - 25s 40ms/step - loss: 0.1569 - accuracy: 0.9475 - val_loss: 0.8183 - val_accuracy: 0.7952
Epoch 8/20
625/625 [==============================] - 25s 40ms/step - loss: 0.1271 - accuracy: 0.9587 - val_loss: 0.8540 - val_accuracy: 0.7941
Epoch 9/20
625/625 [==============================] - 25s 40ms/step - loss: 0.1374 - accuracy: 0.9562 - val_loss: 0.8187 - val_accuracy: 0.7953
Epoch 10/20
625/625 [==============================] - 25s 40ms/step - loss: 0.1192 - accuracy: 0.9617 - val_loss: 0.8301 - val_accuracy: 0.8046
Epoch 11/20
625/625 [==============================] - 25s 40ms/step - loss: 0.0794 - accuracy: 0.9737 - val_loss: 0.8819 - val_accuracy: 0.7997
Epoch 12/20
625/625 [==============================] - 25s 40ms/step - loss: 0.0825 - accuracy: 0.9724 - val_loss: 0.9463 - val_accuracy: 0.7957
Epoch 13/20
625/625 [==============================] - 25s 40ms/step - loss: 0.0699 - accuracy: 0.9776 - val_loss: 0.8533 - val_accuracy: 0.8070
Epoch 14/20
625/625 [==============================] - 25s 40ms/step - loss: 0.0723 - accuracy: 0.9764 - val_loss: 0.8981 - val_accuracy: 0.8053
Epoch 15/20
625/625 [==============================] - 25s 40ms/step - loss: 0.0694 - accuracy: 0.9774 - val_loss: 0.9358 - val_accuracy: 0.8022
Epoch 16/20
625/625 [==============================] - 25s 41ms/step - loss: 0.0679 - accuracy: 0.9780 - val_loss: 0.8770 - val_accuracy: 0.8105
Epoch 17/20
625/625 [==============================] - 25s 41ms/step - loss: 0.0567 - accuracy: 0.9814 - val_loss: 0.9843 - val_accuracy: 0.7975
Epoch 18/20
625/625 [==============================] - 25s 41ms/step - loss: 0.0570 - accuracy: 0.9814 - val_loss: 0.9733 - val_accuracy: 0.7998
Epoch 19/20
625/625 [==============================] - 25s 41ms/step - loss: 0.0531 - accuracy: 0.9830 - val_loss: 0.9401 - val_accuracy: 0.8068
Epoch 20/20
625/625 [==============================] - 25s 41ms/step - loss: 0.0452 - accuracy: 0.9857 - val_loss: 0.9455 - val_accuracy: 0.8113
313/313 [==============================] - 3s 10ms/step - loss: 0.9898 - accuracy: 0.8021
```

Figure 23: Losses

8.1.4 Accuracy

```
Test accuracy with ResNet pretrained model: 0.8021000027656555
```

Figure 24: Accuracy

8.2 DenseNet

8.2.1 Training and Fine Tuning

```
from tensorflow.keras.applications import DenseNet169

# Load pre-trained DenseNet169 model without top classification layer
base_model_dense = DenseNet169(weights='imagenet', include_top=False, input_shape=(32, 32, 3))

# Fine-tuning the entire model
for layer in base_model_dense.layers:
    layer.trainable = True

# using the pre trained model as a feature extractor
model_dense = models.Sequential([
    base_model_dense,
    layers.Flatten(),
    layers.Dense(512, activation='relu'),
    layers.Dropout(0.5),
    layers.Dense(10, activation='softmax')
])

model_dense.compile(loss = loss, optimizer = opt, metrics = metrics)

# Fine-tune the model
checkpoint_dense = ModelCheckpoint('cifar10_fine_tuned_dense.h5', save_best_only=True)
history_dense = model_dense.fit(x_train, y_train, batch_size = 64 , epochs = 20, validation_split = 0.2, callbacks=[checkpoint_dense])
```

Figure 25: Model

8.2.2 Evaluating Model

```
# Evaluate the model
test_loss_dense, test_acc_dense =model_dense.evaluate(x_test, y_test)
```

Figure 26: Evaluating Model

8.2.3 Losses

```
Epoch 1/20
625/625 [==============================] - 42s 67ms/step - loss: 1.0760 - accuracy: 0.6449 - val_loss: 0.6812 - val_accuracy: 0.7652
Epoch 2/20
625/625 [==============================] - 39s 62ms/step - loss: 0.5769 - accuracy: 0.8087 - val_loss: 0.5251 - val_accuracy: 0.8164
Epoch 3/20
625/625 [==============================] - 37s 59ms/step - loss: 0.4139 - accuracy: 0.8622 - val_loss: 0.5494 - val_accuracy: 0.8152
Epoch 4/20
625/625 [==============================] - 37s 59ms/step - loss: 0.2822 - accuracy: 0.9085 - val_loss: 0.5285 - val_accuracy: 0.8310
Epoch 5/20
625/625 [==============================] - 37s 58ms/step - loss: 0.2006 - accuracy: 0.9338 - val_loss: 0.6736 - val_accuracy: 0.8238
Epoch 6/20
625/625 [==============================] - 36s 58ms/step - loss: 0.1389 - accuracy: 0.9546 - val_loss: 0.5950 - val_accuracy: 0.8385
Epoch 7/20
625/625 [==============================] - 36s 58ms/step - loss: 0.1039 - accuracy: 0.9663 - val_loss: 0.6820 - val_accuracy: 0.8359
Epoch 8/20
625/625 [==============================] - 36s 58ms/step - loss: 0.1081 - accuracy: 0.9651 - val_loss: 0.6604 - val_accuracy: 0.8280
Epoch 9/20
625/625 [==============================] - 36s 58ms/step - loss: 0.0771 - accuracy: 0.9747 - val_loss: 0.7002 - val_accuracy: 0.8342
Epoch 10/20
625/625 [==============================] - 36s 58ms/step - loss: 0.0632 - accuracy: 0.9786 - val_loss: 0.6909 - val_accuracy: 0.8439
Epoch 11/20
625/625 [==============================] - 36s 58ms/step - loss: 0.0607 - accuracy: 0.9806 - val_loss: 0.7069 - val_accuracy: 0.8398
Epoch 12/20
625/625 [==============================] - 36s 58ms/step - loss: 0.0588 - accuracy: 0.9801 - val_loss: 0.6760 - val_accuracy: 0.8389
Epoch 13/20
625/625 [==============================] - 36s 58ms/step - loss: 0.0533 - accuracy: 0.9826 - val_loss: 0.6590 - val_accuracy: 0.8479
Epoch 14/20
625/625 [==============================] - 36s 58ms/step - loss: 0.0470 - accuracy: 0.9840 - val_loss: 0.6964 - val_accuracy: 0.8507
Epoch 15/20
625/625 [==============================] - 36s 58ms/step - loss: 0.0464 - accuracy: 0.9848 - val_loss: 0.7434 - val_accuracy: 0.8301
Epoch 16/20
625/625 [==============================] - 36s 58ms/step - loss: 0.0405 - accuracy: 0.9864 - val_loss: 0.6755 - val_accuracy: 0.8529
Epoch 17/20
625/625 [==============================] - 37s 59ms/step - loss: 0.0420 - accuracy: 0.9859 - val_loss: 0.7433 - val_accuracy: 0.8397
Epoch 18/20
625/625 [==============================] - 37s 59ms/step - loss: 0.0429 - accuracy: 0.9864 - val_loss: 0.6867 - val_accuracy: 0.8517
Epoch 19/20
625/625 [==============================] - 37s 59ms/step - loss: 0.0375 - accuracy: 0.9881 - val_loss: 0.7119 - val_accuracy: 0.8448
Epoch 20/20
625/625 [==============================] - 36s 58ms/step - loss: 0.0381 - accuracy: 0.9877 - val_loss: 0.7086 - val_accuracy: 0.8515
313/313 [==============================] - 6s 18ms/step - loss: 0.7296 - accuracy: 0.8480
```

Figure 27: Losses

8.2.4 Accuracy

```
Test accuracy with DenseNet pretrained model: 0.8479999899864197
```

Figure 28: Accuracy

9 Discussion

There are several advantages, limitations and tradeoffs associated with using a custom model vs using a pretrained model.

- Advantages
  1. It is tailored for a specific task allowing fine tuning and optimization to suit our requirements.
  2. It is better at handling nuances specific to a domain and allows better control over the architecture.

- Limitations

  1. Training a custom model requires a large amount of data. If the data is limited it might cause overfitting whereas pretrained models only require a small amount of data for finetuning.
  2. It is computationally expensive and time consuming, as compared to a pretrained model.

- Trade offs

  1. For simpler tasks the pretrained models are sufficient but for tasks with increasing complexity custom models are needed.
  2. Custom datasets require a large dataset while a pretrained model requires only a small dataset.
  3. A high availability of computational resources is required for custom models to be practical. Otherwise pretrained models are more feasible.
  4. The more specialized the task is, the more appropriate a custom model would be.

Unsupported Cell Type. Double-Click to inspect/edit the content.

Loading and Preprocessing data:

```
import os
for dirname, _, filenames in os.walk('/kaggle/input'):
    for filename in filenames:
        print(os.path.join(dirname, filename))

import torchvision
import torchvision.transforms as transforms

transform = transforms.Compose([transforms.ToTensor(),transforms.Normalize((0.5,0.5,0.5),
                                                                            (0.5,0.5,0.5))])

train_dataset = torchvision.datasets.CIFAR10(root='./data', train=True,transform=transform,
                                             download=True)

test_dataset = torchvision.datasets.CIFAR10(root='./data',train=False, transform=transform,
                                            download = True)
import tensorflow as tf
from keras.datasets import cifar10
import numpy as np
import pandas as pd
```

```
/kaggle/input/cifar10-image-recognition/train.npy
/kaggle/input/cifar10-image-recognition/trainLabels.csv
/kaggle/input/cifar10-image-recognition/sampleSubmission.csv
/kaggle/input/cifar10-image-recognition/test.npy
/kaggle/input/cifar10/cifar-10-batches-py/data_batch_1
/kaggle/input/cifar10/cifar-10-batches-py/data_batch_2
/kaggle/input/cifar10/cifar-10-batches-py/batches.meta
/kaggle/input/cifar10/cifar-10-batches-py/test_batch
/kaggle/input/cifar10/cifar-10-batches-py/data_batch_3
/kaggle/input/cifar10/cifar-10-batches-py/data_batch_5
/kaggle/input/cifar10/cifar-10-batches-py/data_batch_4
/kaggle/input/cifar10/cifar-10-batches-py/readme.html
/kaggle/input/cifar10/cifar-10-python/cifar-10-batches-py/data_batch_1
/kaggle/input/cifar10/cifar-10-python/cifar-10-batches-py/data_batch_2
/kaggle/input/cifar10/cifar-10-python/cifar-10-batches-py/batches.meta
/kaggle/input/cifar10/cifar-10-python/cifar-10-batches-py/test_batch
/kaggle/input/cifar10/cifar-10-python/cifar-10-batches-py/data_batch_3
/kaggle/input/cifar10/cifar-10-python/cifar-10-batches-py/data_batch_5
/kaggle/input/cifar10/cifar-10-python/cifar-10-batches-py/data_batch_4
/kaggle/input/cifar10/cifar-10-python/cifar-10-batches-py/readme.html
/kaggle/input/cifar10-python/cifar-10-python.tar.gz
/kaggle/input/cifar10-python/cifar-10-batches-py/data_batch_1
/kaggle/input/cifar10-python/cifar-10-batches-py/data_batch_2
/kaggle/input/cifar10-python/cifar-10-batches-py/batches.meta
/kaggle/input/cifar10-python/cifar-10-batches-py/test_batch
/kaggle/input/cifar10-python/cifar-10-batches-py/data_batch_3
/kaggle/input/cifar10-python/cifar-10-batches-py/data_batch_5
/kaggle/input/cifar10-python/cifar-10-batches-py/data_batch_4
/kaggle/input/cifar10-python/cifar-10-batches-py/readme.html
Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to ./data/cifar-10-python.tar.gz
HBox(children=(FloatProgress(value=1.0, bar_style='info', max=1.0), HTML(value='')))
Extracting ./data/cifar-10-python.tar.gz to ./data
Files already downloaded and verified
```

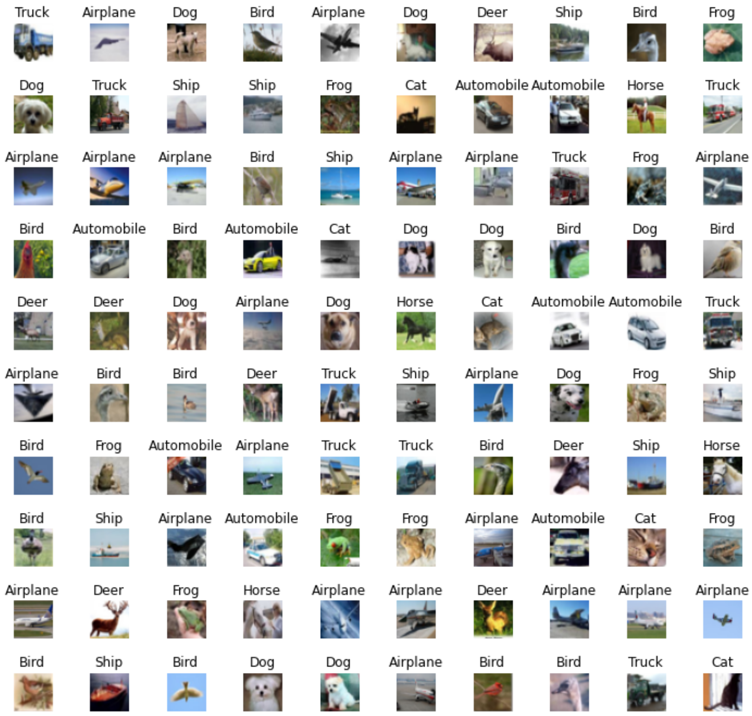
```
# loading dateset
(x_train, y_train),(x_test,y_test) = cifar10.load_data()
```

```
Downloading data from https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
170500096/170498071 [==============================] - 5s 0us/step
```

```
# Checking loaded data
print('Total number of Images in the Dataset:', len(x_train) + len(x_test))
print('Number of train images:', len(x_train))
print('Number of test images:', len(x_test))
print('Shape of training dataset:',x_train.shape)
print('Shape of testing dataset:',x_test.shape)
```

```
Total number of Images in the Dataset: 60000
Number of train images: 50000
Number of test images: 10000
Shape of training dataset: (50000, 32, 32, 3)
Shape of testing dataset: (10000, 32, 32, 3)
```

```
# This piece of code shows a random images and labels for given set of inputs

def showImages(num_row,num_col,X,Y):
    import matplotlib.pyplot as plt
    %matplotlib inline

    from sklearn.utils import shuffle
    (X_rand, Y_rand) = shuffle(X, Y)

    fig, axes = plt.subplots(num_row,num_col,figsize = (12,12))
    axes = axes.ravel()
    for i in range(0, num_row*num_col):
        axes[i].imshow(X_rand[i])
        axes[i].set_title("{}".format(labels[Y_rand.item(i)]))
        axes[i].axis('off')
        plt.subplots_adjust(wspace =1)

    return


labels = ['Airplane', 'Automobile', 'Bird', 'Cat', 'Deer', 'Dog', 'Frog', 'Horse', 'Ship', 'Truck']


num_row = 10
num_col = 10
showImages(num_row,num_col,X =x_train,Y = y_train)
```

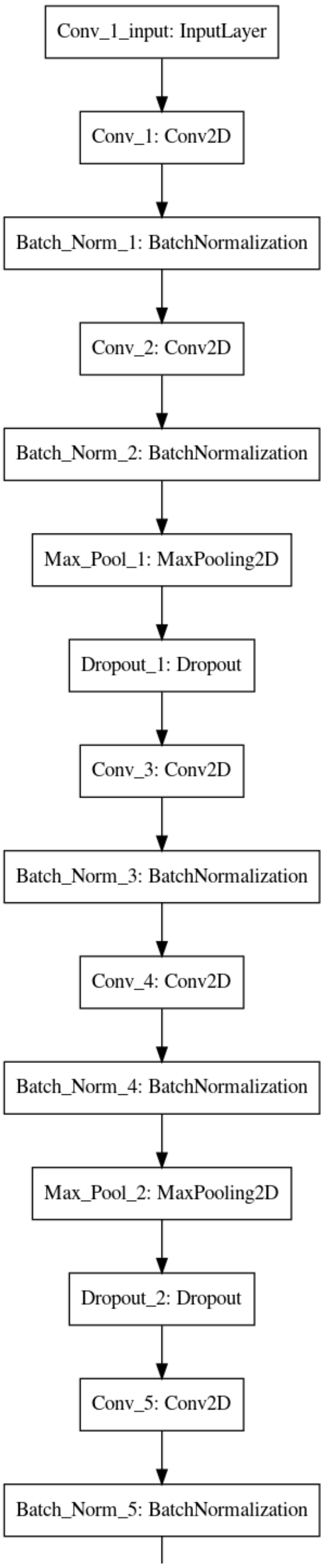



```
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train = x_train/ 255
x_test = x_test/255


import keras
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten, Conv2D, MaxPooling2D, BatchNormalization


#Building model computational graph
model = Sequential()
model.add(Conv2D(32, (3,3), activation = 'relu', padding = 'same', input_shape = (32,32,3),name='Conv_1'))
model.add(BatchNormalization(name='Batch_Norm_1'))
model.add(Conv2D(32, (3,3), activation = 'relu', padding = 'same',name='Conv_2'))
model.add(BatchNormalization(name='Batch_Norm_2'))
model.add(MaxPooling2D(pool_size = (2,2), strides=(2,2),name='Max_Pool_1'))
model.add(Dropout(0.25, name='Dropout_1'))
model.add(Conv2D(64, (3,3), activation = 'relu', padding = 'same',name='Conv_3'))
model.add(BatchNormalization(name='Batch_Norm_3'))
model.add(Conv2D(64, (3,3), activation = 'relu', padding = 'same',name='Conv_4'))
model.add(BatchNormalization(name='Batch_Norm_4'))
model.add(MaxPooling2D(pool_size = (2,2),strides=(2,2),name='Max_Pool_2'))
model.add(Dropout(0.25, name='Dropout_2'))
model.add(Conv2D(128, (3,3), activation = 'relu', padding = 'same',name='Conv_5'))
model.add(BatchNormalization(name='Batch_Norm_5'))
model.add(Conv2D(128, (3,3), activation = 'relu', padding = 'same',name='Conv_6'))
model.add(BatchNormalization(name='Batch_Norm_6'))
model.add(MaxPooling2D(pool_size = (2,2),strides=(2,2),name='Max_Pool_3'))
model.add(Dropout(0.25, name='Dropout_3'))
model.add(Flatten(name='Flatten_Layer'))
model.add(Dense(128, activation = 'relu',name='Fully_Connected_Layer'))
model.add(Dropout(0.25,name='Dropout_4'))
model.add(Dense(10, activation = 'softmax',name='Output_Layer'))
```

```
#Description about parameters and layers
model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
Conv_1 (Conv2D)              (None, 32, 32, 32)        896       
_________________________________________________________________
Batch_Norm_1 (BatchNormaliza (None, 32, 32, 32)        128       
_________________________________________________________________
Conv_2 (Conv2D)              (None, 32, 32, 32)        9248      
_________________________________________________________________
Batch_Norm_2 (BatchNormaliza (None, 32, 32, 32)        128       
_________________________________________________________________
Max_Pool_1 (MaxPooling2D)    (None, 16, 16, 32)        0         
_________________________________________________________________
Dropout_1 (Dropout)          (None, 16, 16, 32)        0         
_________________________________________________________________
Conv_3 (Conv2D)              (None, 16, 16, 64)        18496     
_________________________________________________________________
Batch_Norm_3 (BatchNormaliza (None, 16, 16, 64)        256       
_________________________________________________________________
Conv_4 (Conv2D)              (None, 16, 16, 64)        36928     
_________________________________________________________________
Batch_Norm_4 (BatchNormaliza (None, 16, 16, 64)        256       
_________________________________________________________________
Max_Pool_2 (MaxPooling2D)    (None, 8, 8, 64)          0         
_________________________________________________________________
Dropout_2 (Dropout)          (None, 8, 8, 64)          0         
_________________________________________________________________
Conv_5 (Conv2D)              (None, 8, 8, 128)         73856     
_________________________________________________________________
Batch_Norm_5 (BatchNormaliza (None, 8, 8, 128)         512       
_________________________________________________________________
Conv_6 (Conv2D)              (None, 8, 8, 128)         147584    
_________________________________________________________________
Batch_Norm_6 (BatchNormaliza (None, 8, 8, 128)         512       
_________________________________________________________________
Max_Pool_3 (MaxPooling2D)    (None, 4, 4, 128)         0         
_________________________________________________________________
Dropout_3 (Dropout)          (None, 4, 4, 128)         0         
_________________________________________________________________
Flatten_Layer (Flatten)      (None, 2048)              0         
_________________________________________________________________
Fully_Connected_Layer (Dense (None, 128)               262272    
_________________________________________________________________
Dropout_4 (Dropout)          (None, 128)               0         
_________________________________________________________________
Output_Layer (Dense)         (None, 10)                1290      
=================================================================
Total params: 552,362
Trainable params: 551,466
Non-trainable params: 896
_________________________________________________________________
```

```
# Flow chart of the model

from IPython.display import SVG
from keras.utils.vis_utils import model_to_dot
from keras.utils import plot_model

plot_model(model, to_file='model.png')
```




```
! pip install visualkeras
import visualkeras
visualkeras.layered_view(model,legend=True)
```

```
Collecting visualkeras
```

Training The Model

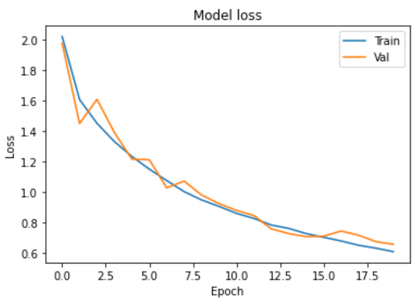
```
     |████████████████████████████████| 989 kB 5.4 MB/s eta 0:00:01
```

```
#compiling model with loss, opt, metrics
loss = 'sparse_categorical_crossentropy'
#learning rates = 0.0001,0.001,0.01,0.1
opt = tf.keras.optimizers.Adam(learning_rate=0.0001,beta_1=0.9, beta_2=0.999,epsilon=1e-06)
metrics = ['accuracy']

model.compile(loss = loss, optimizer = opt, metrics = metrics)
```

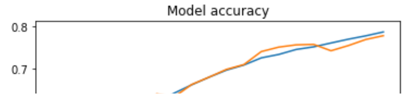
```
# fitting the model for training dataset
hist = model.fit(x_train, y_train, batch_size = 64 , epochs = 20, validation_split = 0.2)
```

```
Epoch 1/20
625/625 [==============================] - 5s 8ms/step - loss: 2.0208 - accuracy: 0.2873 - val_loss: 1.9764 - val_accuracy: 0.3143
Epoch 2/20
625/625 [==============================] - 4s 7ms/step - loss: 1.6067 - accuracy: 0.4159 - val_loss: 1.4489 - val_accuracy: 0.4890
Epoch 3/20
625/625 [==============================] - 4s 7ms/step - loss: 1.4497 - accuracy: 0.4781 - val_loss: 1.6079 - val_accuracy: 0.4779
Epoch 4/20
625/625 [==============================] - 4s 7ms/step - loss: 1.3296 - accuracy: 0.5245 - val_loss: 1.3893 - val_accuracy: 0.5368
Epoch 5/20
625/625 [==============================] - 4s 7ms/step - loss: 1.2322 - accuracy: 0.5595 - val_loss: 1.2144 - val_accuracy: 0.5807
Epoch 6/20
625/625 [==============================] - 4s 7ms/step - loss: 1.1485 - accuracy: 0.5906 - val_loss: 1.2104 - val_accuracy: 0.5926
Epoch 7/20
625/625 [==============================] - 4s 7ms/step - loss: 1.0732 - accuracy: 0.6172 - val_loss: 1.0257 - val_accuracy: 0.6420
Epoch 8/20
625/625 [==============================] - 4s 7ms/step - loss: 1.0005 - accuracy: 0.6438 - val_loss: 1.0696 - val_accuracy: 0.6349
Epoch 9/20
625/625 [==============================] - 4s 7ms/step - loss: 0.9461 - accuracy: 0.6632 - val_loss: 0.9781 - val_accuracy: 0.6639
Epoch 10/20
625/625 [==============================] - 4s 7ms/step - loss: 0.9033 - accuracy: 0.6809 - val_loss: 0.9216 - val_accuracy: 0.6811
Epoch 11/20
625/625 [==============================] - 4s 7ms/step - loss: 0.8578 - accuracy: 0.6975 - val_loss: 0.8773 - val_accuracy: 0.6992
Epoch 12/20
625/625 [==============================] - 4s 7ms/step - loss: 0.8243 - accuracy: 0.7093 - val_loss: 0.8432 - val_accuracy: 0.7104
Epoch 13/20
625/625 [==============================] - 5s 7ms/step - loss: 0.7808 - accuracy: 0.7260 - val_loss: 0.7554 - val_accuracy: 0.7407
Epoch 14/20
625/625 [==============================] - 4s 7ms/step - loss: 0.7584 - accuracy: 0.7340 - val_loss: 0.7245 - val_accuracy: 0.7510
Epoch 15/20
625/625 [==============================] - 4s 7ms/step - loss: 0.7244 - accuracy: 0.7456 - val_loss: 0.7039 - val_accuracy: 0.7563
Epoch 16/20
625/625 [==============================] - 4s 7ms/step - loss: 0.7002 - accuracy: 0.7516 - val_loss: 0.7073 - val_accuracy: 0.7574
Epoch 17/20
625/625 [==============================] - 4s 7ms/step - loss: 0.6760 - accuracy: 0.7606 - val_loss: 0.7412 - val_accuracy: 0.7426
Epoch 18/20
625/625 [==============================] - 4s 7ms/step - loss: 0.6477 - accuracy: 0.7695 - val_loss: 0.7139 - val_accuracy: 0.7547
Epoch 19/20
625/625 [==============================] - 4s 7ms/step - loss: 0.6285 - accuracy: 0.7774 - val_loss: 0.6717 - val_accuracy: 0.7689
Epoch 20/20
625/625 [==============================] - 4s 7ms/step - loss: 0.6054 - accuracy: 0.7863 - val_loss: 0.6539 - val_accuracy: 0.7776
```

```
import matplotlib.pyplot as plt
plt.plot(hist.history['loss'])
plt.plot(hist.history['val_loss'])
plt.title('Model loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Val'], loc='upper right')
plt.show()
```




```
plt.plot(hist.history['accuracy'])
plt.plot(hist.history['val_accuracy'])
plt.title('Model accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Val'], loc='lower right')
plt.show()
```



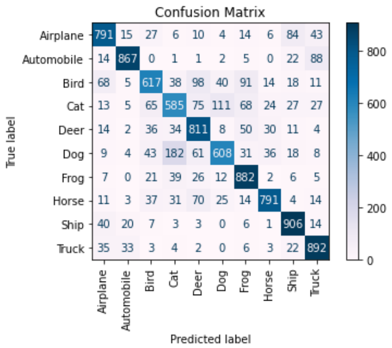

```
# Evaluate the model on the testing set
test_loss, test_accuracy = model.evaluate(x_test, y_test, verbose=2)
print(f'Test Accuracy: {test_accuracy*100:.2f}%')
print(f'Test Loss: {test_loss:.4f}')
print()
```

```
313/313 - 1s - loss: 0.6709 - accuracy: 0.7750
Test Accuracy: 77.50%
Test Loss: 0.6709
```

```
from sklearn.metrics import ConfusionMatrixDisplay
from sklearn.metrics import classification_report, confusion_matrix

# Predictions on the testing set
y_pred = model.predict(x_test)
y_pred_classes = np.argmax(y_pred, axis=1)

# Convert one-hot encoded labels to actual labels
y_true_classes = np.squeeze(y_test)

# Confusion Matrix
conf_matrix = confusion_matrix(y_true_classes, y_pred_classes)

# Classification Report
class_report = classification_report(y_true_classes, y_pred_classes, target_names=labels)
print('Classification Report:\n', class_report)
print()

# Plot Confusion Matrix
plt.figure(figsize=(25,25))
ConfusionMatrixDisplay(conf_matrix, display_labels=labels).plot(cmap='PuBu',values_format='d')
plt.title('Confusion Matrix')
plt.xticks(rotation='vertical')
plt.show()
```

```
Classification Report:
               precision    recall  f1-score   support

    Airplane       0.79      0.79      0.79      1000
  Automobile       0.91      0.87      0.89      1000
        Bird       0.72      0.62      0.66      1000
         Cat       0.63      0.58      0.61      1000
        Deer       0.70      0.81      0.75      1000
         Dog       0.75      0.61      0.67      1000
        Frog       0.76      0.88      0.81      1000
       Horse       0.87      0.79      0.83      1000
        Ship       0.81      0.91      0.86      1000
       Truck       0.81      0.89      0.85      1000

    accuracy                           0.78     10000
   macro avg       0.77      0.78      0.77     10000
weighted avg       0.77      0.78      0.77     10000
```

```
<Figure size 1800x1800 with 0 Axes>
```



ResNet

```
from tensorflow.keras import layers, models
from tensorflow.keras.applications import ResNet50
from tensorflow.keras.callbacks import ModelCheckpoint

# Load pre-trained ResNet50 model without top classification layer
base_model_res = ResNet50(weights='imagenet', include_top=False, input_shape=(32, 32, 3))

# Fine-tuning the entire model
for layer in base_model_res.layers:
    layer.trainable = True

# using the pre trained model as a feature extractor
model_res = models.Sequential([
    base_model_res,
    layers.Flatten(),
    layers.Dense(512, activation='relu'),
    layers.Dropout(0.5),
    layers.Dense(10, activation='softmax')
])

model_res.compile(loss = loss, optimizer = opt, metrics = metrics)

# Fine-tune the model
checkpoint_res = ModelCheckpoint('cifar10_fine_tuned_resnet.h5', save_best_only=True)
history = model_res.fit(x_train, y_train, batch_size = 64 , epochs = 20, validation_split = 0.2, callbacks=[checkpoint_res])

# Evaluate the model
test_loss_res, test_acc_res =model_res.evaluate(x_test, y_test)

print()
print(f'Test accuracy with ResNet pretrained model: {test_acc_res}')
```

```
Epoch 1/20
625/625 [==============================] - 27s 43ms/step - loss: 1.2723 - accuracy: 0.5866 - val_loss: 2.8123 - val_accuracy: 0.1488
Epoch 2/20
625/625 [==============================] - 26s 42ms/step - loss: 0.7178 - accuracy: 0.7577 - val_loss: 0.6775 - val_accuracy: 0.7723
Epoch 3/20
625/625 [==============================] - 26s 42ms/step - loss: 0.5079 - accuracy: 0.8293 - val_loss: 0.6584 - val_accuracy: 0.7870
Epoch 4/20
625/625 [==============================] - 25s 41ms/step - loss: 0.3766 - accuracy: 0.8723 - val_loss: 0.6911 - val_accuracy: 0.7822
Epoch 5/20
625/625 [==============================] - 26s 41ms/step - loss: 0.2765 - accuracy: 0.9054 - val_loss: 0.7172 - val_accuracy: 0.7936
Epoch 6/20
625/625 [==============================] - 25s 41ms/step - loss: 0.1995 - accuracy: 0.9315 - val_loss: 0.7551 - val_accuracy: 0.7956
Epoch 7/20
625/625 [==============================] - 25s 40ms/step - loss: 0.1569 - accuracy: 0.9475 - val_loss: 0.8183 - val_accuracy: 0.7952
Epoch 8/20
625/625 [==============================] - 25s 40ms/step - loss: 0.1271 - accuracy: 0.9587 - val_loss: 0.8540 - val_accuracy: 0.7941
Epoch 9/20
625/625 [==============================] - 25s 40ms/step - loss: 0.1374 - accuracy: 0.9562 - val_loss: 0.8187 - val_accuracy: 0.7953
Epoch 10/20
625/625 [==============================] - 25s 40ms/step - loss: 0.1192 - accuracy: 0.9617 - val_loss: 0.8301 - val_accuracy: 0.8046
Epoch 11/20
625/625 [==============================] - 25s 40ms/step - loss: 0.0794 - accuracy: 0.9737 - val_loss: 0.8819 - val_accuracy: 0.7997
Epoch 12/20
625/625 [==============================] - 25s 40ms/step - loss: 0.0825 - accuracy: 0.9724 - val_loss: 0.9463 - val_accuracy: 0.7957
Epoch 13/20
625/625 [==============================] - 25s 40ms/step - loss: 0.0699 - accuracy: 0.9776 - val_loss: 0.8533 - val_accuracy: 0.8070
Epoch 14/20
625/625 [==============================] - 25s 40ms/step - loss: 0.0723 - accuracy: 0.9764 - val_loss: 0.8981 - val_accuracy: 0.8053
Epoch 15/20
625/625 [==============================] - 25s 40ms/step - loss: 0.0694 - accuracy: 0.9774 - val_loss: 0.9358 - val_accuracy: 0.8022
Epoch 16/20
625/625 [==============================] - 25s 41ms/step - loss: 0.0679 - accuracy: 0.9780 - val_loss: 0.8770 - val_accuracy: 0.8105
Epoch 17/20
625/625 [==============================] - 25s 41ms/step - loss: 0.0567 - accuracy: 0.9814 - val_loss: 0.9843 - val_accuracy: 0.7975
Epoch 18/20
625/625 [==============================] - 25s 41ms/step - loss: 0.0570 - accuracy: 0.9814 - val_loss: 0.9733 - val_accuracy: 0.7998
Epoch 19/20
625/625 [==============================] - 25s 41ms/step - loss: 0.0531 - accuracy: 0.9830 - val_loss: 0.9401 - val_accuracy: 0.8068
Epoch 20/20
625/625 [==============================] - 25s 41ms/step - loss: 0.0452 - accuracy: 0.9857 - val_loss: 0.9455 - val_accuracy: 0.8113
313/313 [==============================] - 3s 10ms/step - loss: 0.9898 - accuracy: 0.8021

Test accuracy with ResNet pretrained model: 0.8021000027656555
```

DenseNet

```
from tensorflow.keras.applications import DenseNet169

# Load pre-trained DenseNet169 model without top classification layer
base_model_dense = DenseNet169(weights='imagenet', include_top=False, input_shape=(32, 32, 3))

# Fine-tuning the entire model
for layer in base_model_dense.layers:
    layer.trainable = True

# using the pre trained model as a feature extractor
model_dense = models.Sequential([
    base_model_dense,
    layers.Flatten(),
    layers.Dense(512, activation='relu'),
    layers.Dropout(0.5),
    layers.Dense(10, activation='softmax')
])

model_dense.compile(loss = loss, optimizer = opt, metrics = metrics)

# Fine-tune the model
checkpoint_dense = ModelCheckpoint('cifar10_fine_tuned_dense.h5', save_best_only=True)
history_dense = model_dense.fit(x_train, y_train, batch_size = 64 , epochs = 20, validation_split = 0.2, callbacks=[checkpoint_dense])

# Evaluate the model
test_loss_dense, test_acc_dense =model_dense.evaluate(x_test, y_test)

print()
print(f'Test accuracy with DenseNet pretrained model: {test_acc_dense}')
```