**Team:** Saaransh Pandey (5190), Ryan Krumenacker (5190), Akshat D Talreja (5190)

# Computer Vision (Deep Learning Pipeline)

## Data preprocessing.

We used Numpy, sklearn, keras for this part. We imported the CIFAR-10 dataset via keras.datasets module. Later, during hyperparameter-tuning the training dataset (40k rows) is split into train-val datasets (0.8:0.2 ratio). This split was performed while fitting the CNN model. Lastly, the test data consisted of 10k rows. We had performed normalization on the dataset by diving the values in X_train, X_test by 255.

## Deep learning models.

We used tensorflow and keras modules to define and train our CNN models.

Initially, we implemented a baseline cnn model, consisting of 2 convolutional blocks (each block consisting of two convolution layers of the same number of filters and a same kernel size + one max pool layer). Then, after flattening the output using a Flatten layer, a Dense layer of 64 unit size was added, followed by the final Dense layer of 10 unit size with softmax activation to classify the input data into one of the labels. Also, all the Conv2D layers + 64 unit size Dense layer were added with 'relu' activation. Please find the summary of the baseline model in **Appendix 1**.

We implemented multiple neural network architectures with the aim of improving on our baseline cnn model. For our second model, we added Dropout layers and L2 regularization (as advised by the TA). For our third model, we added a BatchNormalization layer. We had also implemented another model for the dataset shift experiment, on top of the fine-tuned model.

Values of key hyperparameters:

- 'kernel_size' = 3 for all the Conv2D layers
- Two Conv2D layers with filters = 16, and other two Conv2D layers with filters=32
- 'pool_size' for MaxPooling2D layers = 2
- 'rate' for Dropout layer = 0.2 & 'kernel_regularizer'=l2(0.001) for Model 2
- Activation = 'relu' for all the layers (except final Dense layer)
- Optimizer = Adam
- Loss = 'categorical_crossentropy'
- Batch_size = 32, epoch = 20, validation_split = 0.2 (parameters for model.fit())
- We had kept a learning rate scheduler, with an initial learning rate of 1e-3, and then gets halved at 10 epochs and again at 15 epochs

We utilized the keras_tuner module for tuning the hyperparameters.

## Results and discussion.

- Training/validation/test results

  **Baseline Model results:**

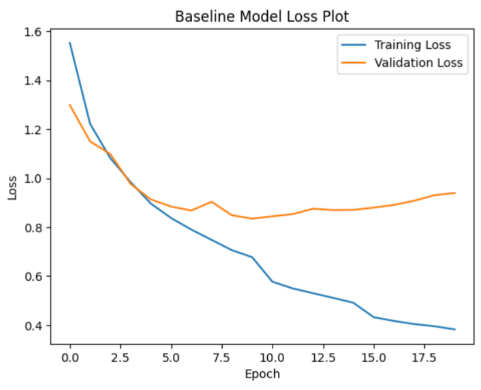


Figure: Loss vs # Epochs plot for the Baseline Model

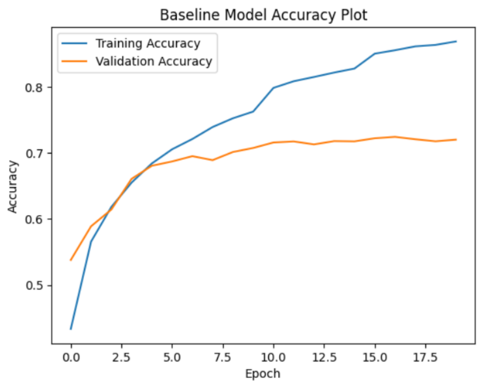


Figure: Accuracy vs # Epochs plot for the Baseline Model

```
Test Loss: 1.0076985359191895
Test Accuracy: 0.7099000215530396
```

Here, as we can see initially the loss for both train and val dataset drops at a similar rate. But for a higher number of epochs, the loss continues to drop for the training data whereas it increases and then drops to a previous loss value for the validation dataset. Similarly, for a higher number of epochs, the difference in the accuracy score increases between the two datasets, indicating a case of overfitting.

**Model 2 results (Baseline Model with Dropout layer and L2 Regularization): Summary of the model, Loss & Accuracy Plots can be found in Appendix 2.**

```
Test Loss: 0.9320688247680664
Test Accuracy: 0.7229999899864197
```

Here, for Model 2 we added Dropout layers for the two convolution blocks, along with kernel_regularizer as L2(0.001). Interestingly the validation loss is always lower than the training data loss, and consequently the accuracy score is higher for the val dataset. But more importantly, due to regularization, we can see the loss for both dataset continue to drop at a similar rate. Hence, solving the issue of overfitting and improving on our baseline model. The overall loss and accuracy score on test data is slightly improved in comparison to the score obtained for the baseline model.

**Model 3 results (Model 2 with BatchNormalization Layer):**

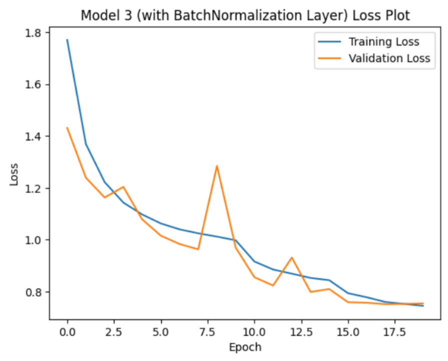


Figure: Loss vs # Epochs plot for Model 3

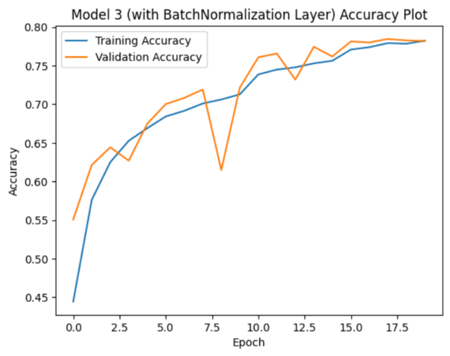


Figure: Accuracy vs # Epochs plot for Model 3

```
Test Loss: 0.7703707218170166
Test Accuracy: 0.7764000296592712
```

Here, we added a BatchNormailzation Layer after every Conv2D layer. The overall loss and accuracy score on test data is much improved in comparison to the score obtained for the model 2. Please find the summary of the model in **Appendix 3.**

- We fine-tuned Model 3 via keras_tuner as mentioned earlier. We focused more on tuning the Dropout layer, L2 regularization parameter, along with the unit size for the pre-final Dense layer (as advised by the TA). The results for the best hyperparameters are mentioned in **Appendix 4**.
  **Loss & Accuracy Plots can be found in Appendix 5.**
```
Test Loss: 0.7341786026954651
Test Accuracy: 0.8012999892234802
```

- The performance of the CNN model after hyperparameter fine-tuning outperforms the traditional pipeline from Project Milestone 2. Even the baseline CNN model outperformed the fine-tuned traditional ML pipeline. The highest accuracy achieved via traditional ML pipeline after hyperparameter tuning was 0.5392, in comparison to 0.8012 for the fine-tuned CNN model. This high performance of the CNN model can be attributed to the complexity captured by the neural network, hence making it much more effective in comparison to the traditional ML model.

- We used ImageDataGenerator from keras modules for the dataset shift experiment. We applied conditions such as: horizontal_flip, vertical_flip (to a random set of images in the test dataset), changing brightness, along with other parameters to the test dataset. Find below comparison between the deep learning and traditional ML model.

| Dataset Shift Type | Deep Learning Model | Traditional ML Model |
|---|---|---|
| No Dataset Shift | 0.8012 | 0.5392 |
| Flip (Horizontal & Vertical) | 0.5823 | 0.0944 |
| Shift (Width & Height) | 0.6143 | 0.1004 |
| Zoom | 0.7324 | 0.0991 |
| Rotation | 0.5792 | 0.1025 |
| Brightness | 0.1080 | 0.1002 |
| All combined | 0.1040 | 0.0996 |

Table: Accuracy score comparison of dataset shift experiment for both pipelines

Here, we can see performance of CNN Model decreases for different types of dataset shifts, especially most in the case of Brightness shift. Due to low resolution of images in CIFAR-10 dataset, variations in brightness may have a more pronounced impact on the visibility of features. Hence, CNN model may struggle to discern objects and patterns. Nonetheless, it still completely outperforms the Traditional ML Model. We can conclude that the Deep Learning Model is more robust in comparison to the Traditional ML Model as it is able to capture more complexity of the input pixels, along with being better at feature abstraction and generalization capabilities to classify the labels more efficiently.

## Conclusion.

From the above results, we can conclude that the cnn model after fine-tuning completely outperforms the fine-tuned traditional ML pipeline. Even for the dataset shift experiment, although both the deep learning & traditional ML models breakdown (decreased performance), yet the deep learning model performs considerably better than the traditional ML pipeline.

**Above and Beyond: Appendix 6**

## Natural Language Processing (Deep Learning Pipeline)

**Data preprocessing & libraries:** Preprocessing for the DL pipeline was done similarly to our traditional pipeline. We replaced accented characters such as “È” with their English equivalents and further got rid of unknown characters such as ‘\x97’. Furthermore, we removed HTML tags, punctuations, URLs, hashtags, date & time to ensure that our feature engineering techniques can closely factor meaning and semantic structure without focusing on stylistic features.

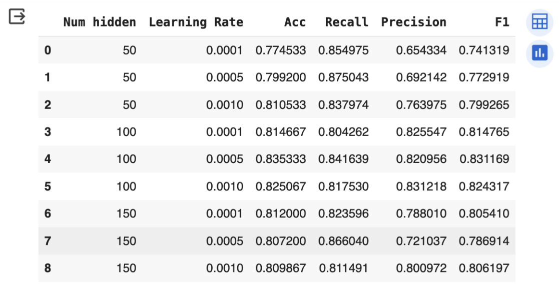
Furthermore, for this pipeline, we primarily made use of pytorch, alongside helper packages such as nltk, pandas, numpy, sklearn, etc. For our neural network architectures, we experimented with two approaches: traditional neural networks and RNNs. We further used GloVe and TfIDF embedding techniques.

**Deep learning models.**
**RNNs:** For RNNs, we decided to go with the LSTM network since our reviews were long and LSTMs help avoid RNNs vanishing gradient / memory issues. We made use of glove pre-trained embeddings. These were passed through LSTM cells for each word. Intermediate outputs were tracked. At the end, a combination of these intermediate outputs was used alongside a linear ANN layer and sigmoid function to produce a probability of the review being positive or not. Among traditional hyperparameters, we looked at the learning rate and the dimension of the hidden layer used at the end. We experimented with learning rates of 0.0001, 0.0005, and 0.001 and with hidden layer dimensions of 50, 100, and 150. Within the architecture, we experimented with “mean” and “last” aggregation. Mean aggregation averages the intermediate outputs of the LSTM before passing to the final linear layer while “last” aggregation just takes the final hidden state and uses that for classification. We further experimented with both bi-directional and unidirectional networks. Another caveat was that LSTMs need inputs to be packed into fixed lengths. Since the longest review was ~2000+ words, this led to that many sequential computations, which caused us to exceed our compute capacity and further led to very slow training. To ease this, we chose to restrict the number of words from each review used for tuning. Since even a few hundred words led to the same issues, we choose to consider the first 100 words during the tuning stage (our “trimmed approach”). Tuning was done using the validation set using trimming and the final combination of parameters was used to train a full network (with complete lengthened reviews) which was evaluated on the test set.

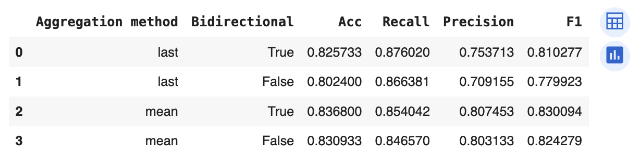
**DANs:** Alongside RNNs, we trained a DAN model with TF-IDF embeddings, which were generated via the TfidfVectorizer from sklearn. We chose a simple feed-forward network architecture for the DAN model with three linear layers and an output sigmoid function layer to produce probabilities. This architecture can be viewed in **Appendix 7**. We first trained a baseline model with only two key hyperparameters: learning rate and the dimension of hidden layers. From these results (which can be viewed below), we found that our model was significantly overfitting the training data: 100% training accuracy by the final epoch but validation accuracy of only ~75%. To resolve this problem, we implemented two key additions: regularization and dropout. This created two more hyperparameters: dropout rate & weight decay. The model architecture with dropout layers can be viewed in **Appendix 8**. With our tuned model, we varied the learning rate over the range {0.01, 0.005, 0.001} and the hidden dimension size over the range {50, 100, 150}. After finding an optimal learning rate & hidden dimension size, we tuned the other two hyperparameters on the validation set. However, we did not find it necessary to change them from their default values. Our final dropout rate was 0.5, and our final weight decay was 0.001. A learning rate scheduler was also included in the final version of the tuned model to slightly improve test accuracy.

**Results and discussion.**

**RNN:** Our base network used a hidden layer with dimension of 100 at the end, learning rate of 0.0001, “last” aggregation, was unidirectional, and further used pre-trained glove embeddings trained alongside the network. For this, we obtained a **test accuracy of 86.66%, recall of 0.865, precision of 0.866, and F1 score of 0.866.** Training data yielded an **accuracy of 90.84%, recall of 0.914, precision of 0.902, and F1 of 0.908**. There is a hint of overfitting here, however, it is not drastic and the test performance was in line with the upper echelons of our traditional pipeline. For tuning, we used the “trimmed” approach defined earlier with only the first 100 words. The left table shows experiments on combinations of dimension of hidden layer & learning rate whereas the right shows experiments on combinations of aggregation method and bidirectional vs unidirectional network architecture



| | Num hidden | Learning Rate | Acc | Recall | Precision | F1 |
|---|---|---|---|---|---|---|
| **0** | 50 | 0.0001 | 0.774533 | 0.854975 | 0.654334 | 0.741319 |
| **1** | 50 | 0.0005 | 0.799200 | 0.875043 | 0.692142 | 0.772919 |
| **2** | 50 | 0.0010 | 0.810533 | 0.837974 | 0.763975 | 0.799265 |
| **3** | 100 | 0.0001 | 0.814667 | 0.804262 | 0.825547 | 0.814765 |
| **4** | 100 | 0.0005 | 0.835333 | 0.841639 | 0.820956 | 0.831169 |
| **5** | 100 | 0.0010 | 0.825067 | 0.817530 | 0.831218 | 0.824317 |
| **6** | 150 | 0.0001 | 0.812000 | 0.823596 | 0.788010 | 0.805410 |
| **7** | 150 | 0.0005 | 0.807200 | 0.866040 | 0.721037 | 0.786914 |
| **8** | 150 | 0.0010 | 0.809867 | 0.811491 | 0.800972 | 0.806197 |

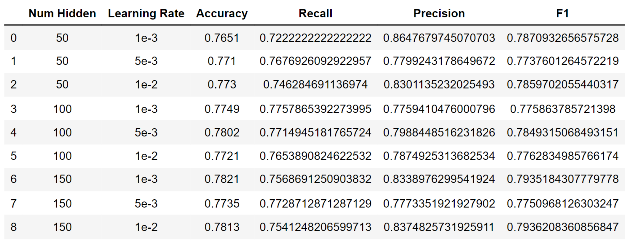


| | Aggregation method | Bidirectional | Acc | Recall | Precision | F1 |
|---|---|---|---|---|---|---|
| **0** | last | True | 0.825733 | 0.876020 | 0.753713 | 0.810277 |
| **1** | last | False | 0.802400 | 0.866381 | 0.709155 | 0.779923 |
| **2** | mean | True | 0.836800 | 0.854042 | 0.807453 | 0.830094 |
| **3** | mean | False | 0.830933 | 0.846570 | 0.803133 | 0.824279 |

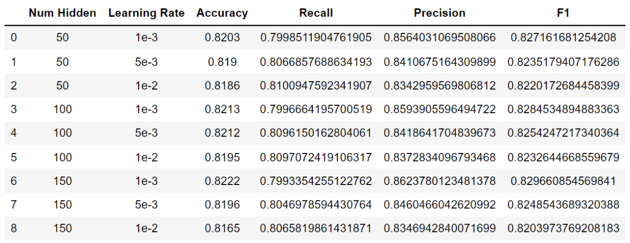
We see a clear pattern of underfitting and overfitting in the tuning on learning rate and layer dimensions, almost like an upside down parabola. The ideal f1 of ~0.84 was achieved with a 100 dimension hidden layer at the end and a learning rate of 0.0005. Holding learning rate constant, both 150 and 50 dimension layers performed worse, and similarly, holding dimension of hidden layer constant, low learning rate of 0.0001 and high one of 0.001 performed worse. At low dimensions and dimensions, the model likely underfit the data whereas at a higher one, it likely overfit it. Within the other experiment, we see that mean aggregation clearly outperformed last aggregation. This is likely as averaging all hidden states can lead to richer semantic representations and better memory of earlier words. Holding for aggregation method, a bidirectional network performed better, which is intuitive as the hidden states here can likely preserve richer representations. Ultimately, we decided on a final bidirectional LSTM model with mean aggregation and a final hidden layer of dimension 100 and a learning rate of 0.0005.

We used these settings with the model trained on full length reviews vs trimmed reviews. This led to a test accuracy of 88.67%, precision of 0.917, recall of 0.850, and f1 score of 0.882. It seems that most improvement was driven by improved precision as recall fell slightly, meaning that the model likely got better at identifying negative reviews that it early classified as positive.

**DAN:** Below are the results of the testing data for both the baseline and the final tuned model. The baseline results are on the left and the tuned results are on the right



| | Num Hidden | Learning Rate | Accuracy | Recall | Precision | F1 |
|---|---|---|---|---|---|---|
| 0 | 50 | 1e-3 | 0.7651 | 0.7222222222222222 | 0.8647679745070703 | 0.7870932656575728 |
| 1 | 50 | 5e-3 | 0.771 | 0.7676926092922957 | 0.7799243178649672 | 0.7737601264572219 |
| 2 | 50 | 1e-2 | 0.773 | 0.746284691136974 | 0.8301135232025493 | 0.7859702055440317 |
| 3 | 100 | 1e-3 | 0.7749 | 0.7757865392273995 | 0.7759410476000796 | 0.775863785721398 |
| 4 | 100 | 5e-3 | 0.7802 | 0.7714945181765724 | 0.7988448516231826 | 0.7849315068493151 |
| 5 | 100 | 1e-2 | 0.7721 | 0.7653890824622532 | 0.7874925313682534 | 0.7762834985766174 |
| 6 | 150 | 1e-3 | 0.7821 | 0.7568691250903832 | 0.8338976299541924 | 0.7935184307779778 |
| 7 | 150 | 5e-3 | 0.7735 | 0.7728712871287129 | 0.7773351921927902 | 0.7750968126303247 |
| 8 | 150 | 1e-2 | 0.7813 | 0.7541248206599713 | 0.8374825731925911 | 0.7936208360856847 |



| | Num Hidden | Learning Rate | Accuracy | Recall | Precision | F1 |
|---|---|---|---|---|---|---|
| 0 | 50 | 1e-3 | 0.8203 | 0.7998511904761905 | 0.8564031069508066 | 0.827161681254208 |
| 1 | 50 | 5e-3 | 0.819 | 0.8066857688634193 | 0.8410675164309899 | 0.8235179407176286 |
| 2 | 50 | 1e-2 | 0.8186 | 0.8100947592341907 | 0.8342959569806812 | 0.8220172684458399 |
| 3 | 100 | 1e-3 | 0.8213 | 0.7996664195700519 | 0.8593905596494722 | 0.8284534894883363 |
| 4 | 100 | 5e-3 | 0.8212 | 0.8096150162804061 | 0.8418641704839673 | 0.8254247217340364 |
| 5 | 100 | 1e-2 | 0.8195 | 0.8097072419106317 | 0.8372834096793468 | 0.8232644668559679 |
| 6 | 150 | 1e-3 | 0.8222 | 0.7993354255122762 | 0.8623780123481378 | 0.829660854569841 |
| 7 | 150 | 5e-3 | 0.8196 | 0.8046978594430764 | 0.8460466042620992 | 0.8248543689320388 |
| 8 | 150 | 1e-2 | 0.8165 | 0.8065819861431871 | 0.8346942840071699 | 0.8203973769208183 |

The training of the tuned DAN was significantly more variant than the baseline DAN, but resulted in ~3-5% better testing accuracy performance. The plots of training loss for both the baseline and tuned DAN can be viewed in **Appendix 9**. We believe that this variance is almost entirely attributable to the randomness associated with the dropout layers. We used a dropout ratio of 0.5 for all of our experiments.

**Comparison with traditional pipeline:** Our best model from the DL pipeline was the tuned RNN model, which yielded an accuracy of ~88.67% that was better than our best model in the traditional pipeline i.e. the optimized Logistic Regression model trained using TF-IDF vectorization. F1 of our tuned RNN was slightly lower at 0.882 vs 0.885 for the optimized Logistic model, however, this is a very minor difference. The fact that logistic regression with fairly simplistic representations did as well as a complex network could indicate that the semantics/language of reviews here is fairly simple and simple embeddings / representations can be powerful while being efficient. We also see that the precision of the tuned RNN (~0.917) was better than that of the Log Reg model (0.892) and perhaps if one were particular about avoiding false positives, the RNN is a better choice, but in most cases the smaller, efficient, high performing model is ideal.

**Dataset shift experiments:** For the dataset shift, we looked at lengths as the main differentiator. For the DL pipeline shifts, we chose the stronger model, which was the LSTM in this case. We decided to train the model on reviews with lengths less than 200 which was around the average length. And we tested the model on the remaining reviews. This was done using the optimal parameters found earlier. Average length of the training reviews was 128 whereas for test reviews, it was 374. ~60% of reviews fit the criteria for training and the remaining were used for testing. With this, our optimal model miserably overfit. All metrics for training data was ~1, whereas for test data, we got an **accuracy of 84.5%, precision of 95.2%, recall of 72.8%, and F1 of 82.35%**. This was worse than the test performance of even our base model. To benchmark this, we went back to our original best model to gauge its performance on long reviews. ~40% of reviews in our test data had a length greater than 200. On these, our base model gave an **accuracy of ~88.8%, precision of 0.888, recall of 0.891, and f1 of 0.890**. This somewhat shows that training on short reviews breaks even our best model and the performance does not generalize to longer reviews.

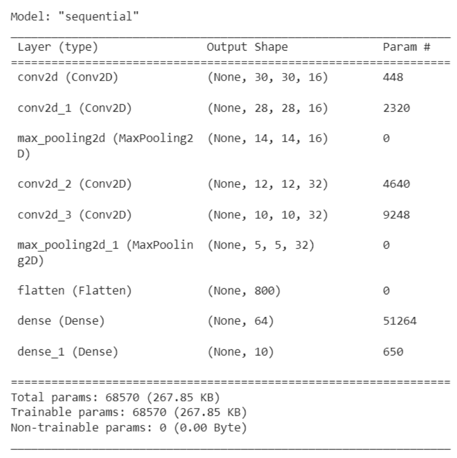
Our best traditional pipeline from Project Milestone 2 (a Logistic Regression model) also saw a significant decrease in performance on the shifted dataset described above. The specific results are shown below.

| **LogReg before dataset shift:** | **LogReg after dataset shift:** |
|---|---|
| Accuracy = 0.88184 | Accuracy = 0.8127006420545746 |
| Recall Score = 0.8773263027295285 | Recall Score = 0.8308394354239627 |
| Precision Score = 0.8918492826738137 | Precision Score = 0.7853345370649012 |
| F1 Score = 0.8845281838792901 | F1 Score = 0.8074463696369636 |

Overall, both models, when trained on shorter reviews, failed to generalize to longer reviews. The RNN model, while giving higher absolute metrics, suffered immensely from overfitting. Training accuracy for LogReg was ~83% and other metrics tracked similarly. Furthermore, the RNN also had a very skewed result i.e. its recall was significantly worse i.e. it was very bad with false negatives. Thus, it seems that LogReg would be more robust, but a varied dataset in terms of length is the best strategy.

# APPENDIX

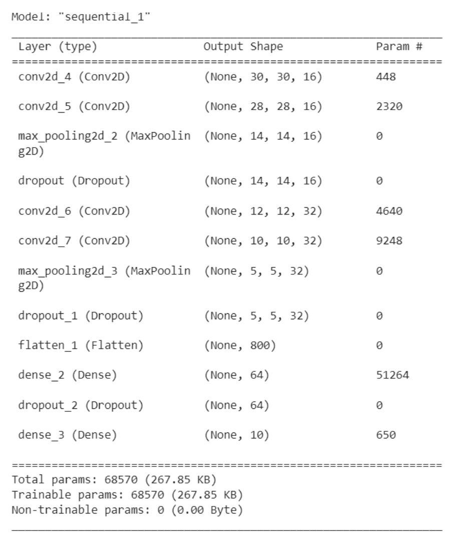
**1.**



```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 30, 30, 16)        448

 conv2d_1 (Conv2D)           (None, 28, 28, 16)        2320

 max_pooling2d (MaxPooling2  (None, 14, 14, 16)        0
 D)

 conv2d_2 (Conv2D)           (None, 12, 12, 32)        4640

 conv2d_3 (Conv2D)           (None, 10, 10, 32)        9248

 max_pooling2d_1 (MaxPoolin  (None, 5, 5, 32)          0
 g2D)

 flatten (Flatten)           (None, 800)               0

 dense (Dense)               (None, 64)                51264

 dense_1 (Dense)             (None, 10)                650

=================================================================
Total params: 68570 (267.85 KB)
Trainable params: 68570 (267.85 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Figure: Baseline Model Summary

**2.**



```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_4 (Conv2D)           (None, 30, 30, 16)        448

 conv2d_5 (Conv2D)           (None, 28, 28, 16)        2320

 max_pooling2d_2 (MaxPoolin  (None, 14, 14, 16)        0
 g2D)

 dropout (Dropout)           (None, 14, 14, 16)        0

 conv2d_6 (Conv2D)           (None, 12, 12, 32)        4640

 conv2d_7 (Conv2D)           (None, 10, 10, 32)        9248

 max_pooling2d_3 (MaxPoolin  (None, 5, 5, 32)          0
 g2D)

 dropout_1 (Dropout)         (None, 5, 5, 32)          0

 flatten_1 (Flatten)         (None, 800)               0

 dense_2 (Dense)             (None, 64)                51264

 dropout_2 (Dropout)         (None, 64)                0

 dense_3 (Dense)             (None, 10)                650

=================================================================
Total params: 68570 (267.85 KB)
Trainable params: 68570 (267.85 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Figure: Model 2 Summary

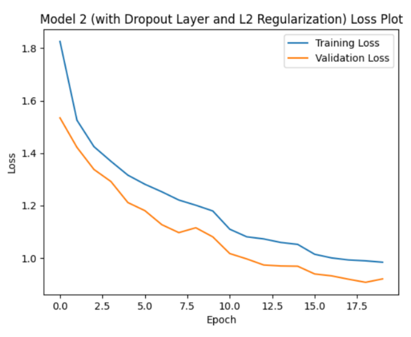


Figure: Loss vs # Epochs plot for Model 2

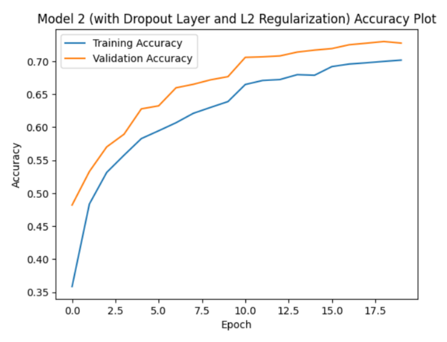
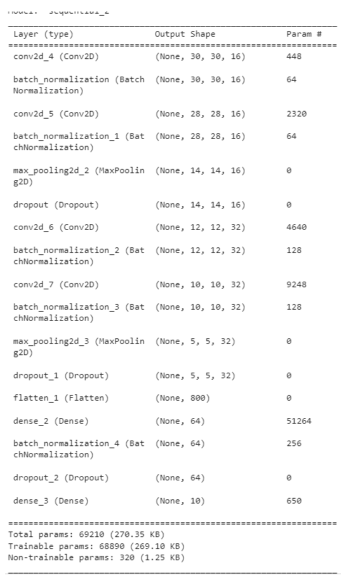


Figure: Accuracy vs # Epochs plot for Model 2

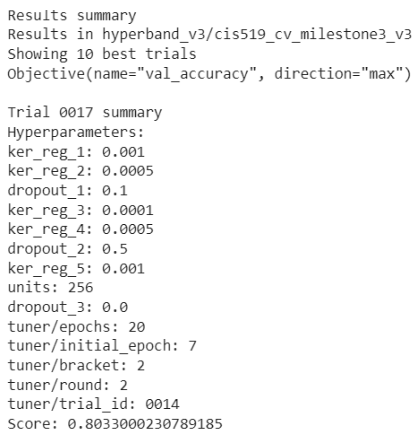
**3.**



```
___________________________________________________________________
 Layer (type)                Output Shape              Param #
===================================================================
 conv2d_4 (Conv2D)           (None, 30, 30, 16)        448

 batch_normalization (Batch  (None, 30, 30, 16)        64
 Normalization)

 conv2d_5 (Conv2D)           (None, 28, 28, 16)        2320

 batch_normalization_1 (Bat  (None, 28, 28, 16)        64
 chNormalization)

 max_pooling2d_2 (MaxPoolin  (None, 14, 14, 16)        0
 g2D)

 dropout (Dropout)           (None, 14, 14, 16)        0

 conv2d_6 (Conv2D)           (None, 12, 12, 32)        4640

 batch_normalization_2 (Bat  (None, 12, 12, 32)        128
 chNormalization)

 conv2d_7 (Conv2D)           (None, 10, 10, 32)        9248

 batch_normalization_3 (Bat  (None, 10, 10, 32)        128
 chNormalization)

 max_pooling2d_3 (MaxPoolin  (None, 5, 5, 32)          0
 g2D)

 dropout_1 (Dropout)         (None, 5, 5, 32)          0

 flatten_1 (Flatten)         (None, 800)               0

 dense_2 (Dense)             (None, 64)                51264

 batch_normalization_4 (Bat  (None, 64)                256
 chNormalization)

 dropout_2 (Dropout)         (None, 64)                0

 dense_3 (Dense)             (None, 10)                650

===================================================================
Total params: 69210 (270.35 KB)
Trainable params: 68890 (269.10 KB)
Non-trainable params: 320 (1.25 KB)
___________________________________________________________________
```

Figure: Model 3 Summary

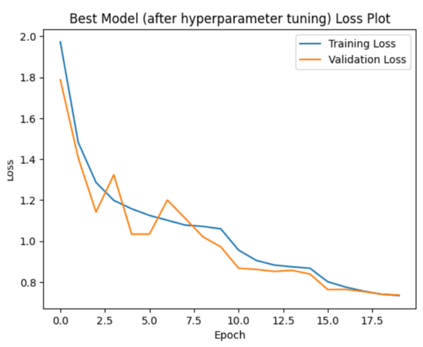
**4.**



```
Results summary
Results in hyperband_v3/cis519_cv_milestone3_v3
Showing 10 best trials
Objective(name="val_accuracy", direction="max")

Trial 0017 summary
Hyperparameters:
ker_reg_1: 0.001
ker_reg_2: 0.0005
dropout_1: 0.1
ker_reg_3: 0.0001
ker_reg_4: 0.0005
dropout_2: 0.5
ker_reg_5: 0.001
units: 256
dropout_3: 0.0
tuner/epochs: 20
tuner/initial_epoch: 7
tuner/bracket: 2
tuner/round: 2
tuner/trial_id: 0014
Score: 0.8033000230789185
```

Figure: Best Hyperparameters after fine-tuning the model via keras_tuner

**5.**



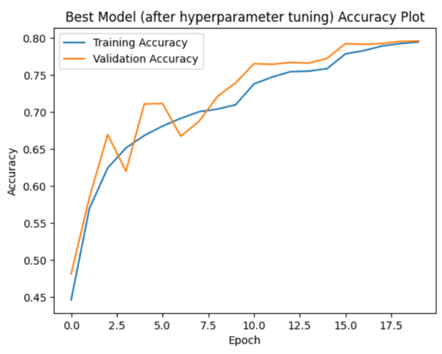


Figure: Loss vs # Epochs plot after hyperparameter tuning Figure: Accuracy vs # Epochs plot after hyperparameter tuning

## 6.

**Part 1: Dataset Shift Model**

Later, we trained the cnn model on augmented dataset using ImageDataGenerator (different parameters from the previous test dataset generator) to monitor the performance of the model for the datashift experiment. It turns out, dataset shift model performed a bit better (for few dataset shift types, but a more generalized performance overall) in comparison to the earlier fine-tuned cnn model for the shifted test dataset.

| Dataset Shift Type | Deep Learning Model Accuracy Score |
|---|---|
| Flip (Horizontal & Vertical) | 0.6295999884605408 |
| Shift (Width & Height) | 0.6013000011444092 |
| Zoom | 0.6639000177383423 |
| Rotation | 0.6320000290870667 |
| Brightness | 0.11089999973773956 |
| All combined | 0.1039000004529953 |

Table: Performance for Dataset Shift Model

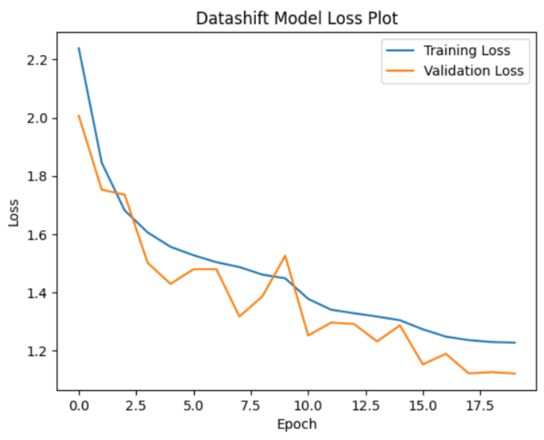


Figure: Loss vs # Epochs plot for the datashift model

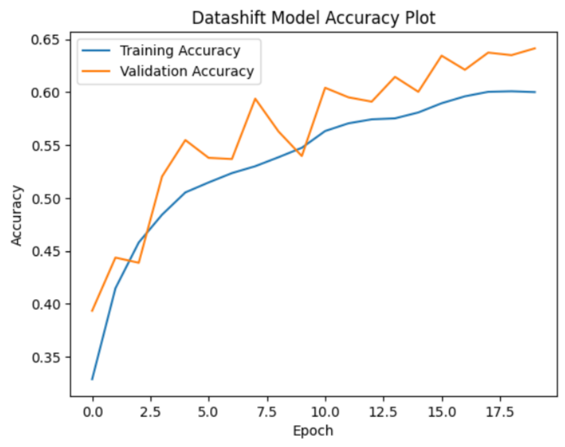
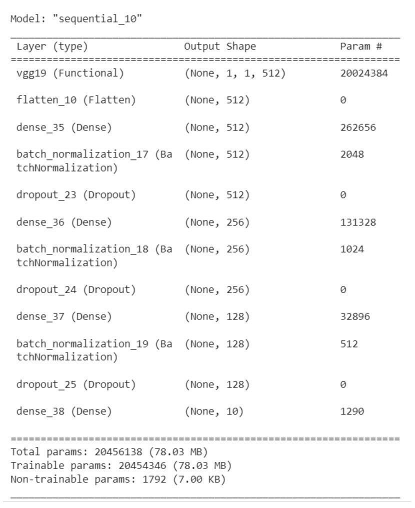


Figure: Accuracy vs # Epochs plot for the datashift model

**Part 2: Transfer Learning Models**
**VGG19 Model:**

We used a VGG19 model with parameters: include_top=False,weights='imagenet'. Then, we added a Flatten layer followed by few Dense, BatchNormalization, and Dropout layers. Model summary is mentioned below:



```
Model: "sequential_10"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 vgg19 (Functional)          (None, 1, 1, 512)         20024384

 flatten_10 (Flatten)        (None, 512)               0

 dense_35 (Dense)            (None, 512)               262656

 batch_normalization_17 (Ba  (None, 512)               2048
 tchNormalization)

 dropout_23 (Dropout)        (None, 512)               0

 dense_36 (Dense)            (None, 256)               131328

 batch_normalization_18 (Ba  (None, 256)               1024
 tchNormalization)

 dropout_24 (Dropout)        (None, 256)               0

 dense_37 (Dense)            (None, 128)               32896

 batch_normalization_19 (Ba  (None, 128)               512
 tchNormalization)

 dropout_25 (Dropout)        (None, 128)               0

 dense_38 (Dense)            (None, 10)                1290

=================================================================
Total params: 20456138 (78.03 MB)
Trainable params: 20454346 (78.03 MB)
Non-trainable params: 1792 (7.00 KB)
_________________________________________________________________
```

Figure: Transfer Learning (VGG19) Model Summary

Results:
We trained the model same as earlier, with exception of the learning rate as 1e-4, and adding another callback for early stopping due to computational limit.

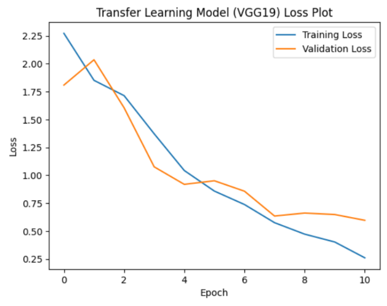


Figure: Loss vs # Epochs plot for VGG19 Model

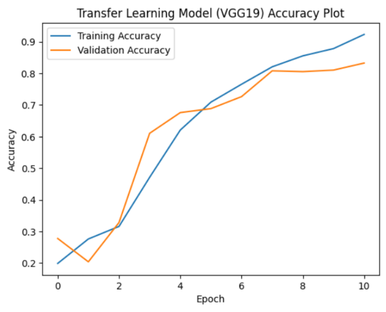
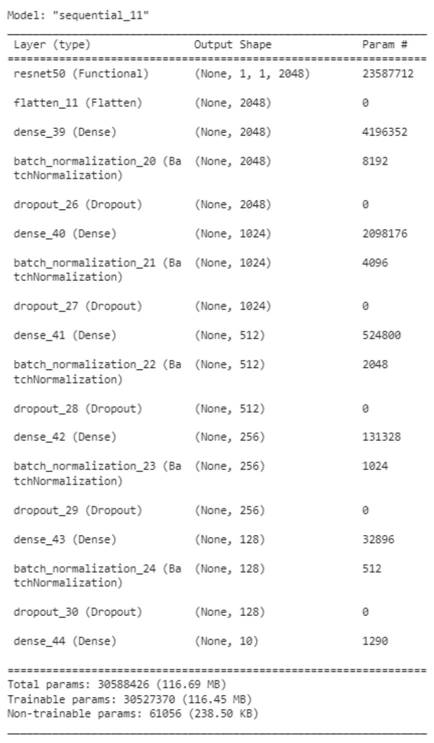


Figure: Accuracy vs # Epochs plot for VGG19 Model

```
Test Loss: 0.6213569045066833
Test Accuracy: 0.8309999704360962
```

Here we can clearly see, that the VGG19 model is outperforming all the previous CNN models. Due to much higher capacity attributed to VGG19's architecture, model being pretrained on large ImageNet dataset, among other reasons, the transfer learning model outperforms a simple 2-layer CNN model.

**ResNet50 Model:**
We used a ResNet50 model with parameters: include_top=False,weights='imagenet'. Then, we added a Flatten layer followed by few Dense, BatchNormalization, and Dropout layers. Model summary is mentioned below:



```
Model: "sequential_11"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 resnet50 (Functional)       (None, 1, 1, 2048)        23587712

 flatten_11 (Flatten)        (None, 2048)              0

 dense_39 (Dense)            (None, 2048)              4196352

 batch_normalization_20 (Ba  (None, 2048)              8192
 tchNormalization)

 dropout_26 (Dropout)        (None, 2048)              0

 dense_40 (Dense)            (None, 1024)              2098176

 batch_normalization_21 (Ba  (None, 1024)              4096
 tchNormalization)

 dropout_27 (Dropout)        (None, 1024)              0

 dense_41 (Dense)            (None, 512)               524800

 batch_normalization_22 (Ba  (None, 512)               2048
 tchNormalization)

 dropout_28 (Dropout)        (None, 512)               0

 dense_42 (Dense)            (None, 256)               131328

 batch_normalization_23 (Ba  (None, 256)               1024
 tchNormalization)

 dropout_29 (Dropout)        (None, 256)               0

 dense_43 (Dense)            (None, 128)               32896

 batch_normalization_24 (Ba  (None, 128)               512
 tchNormalization)

 dropout_30 (Dropout)        (None, 128)               0

 dense_44 (Dense)            (None, 10)                1290

=================================================================
Total params: 30588426 (116.69 MB)
Trainable params: 30527370 (116.45 MB)
Non-trainable params: 61056 (238.50 KB)
_________________________________________________________________
```

Figure: Transfer Learning (ResNet50) Model Summary

Results:
We trained the model same as earlier, with exception of the learning rate as 1e-4, and adding another callback for early stopping due to computational limit.

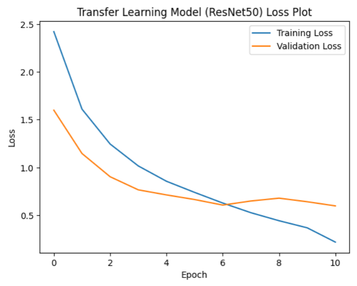


Figure: Loss vs # Epochs plot for ResNet50 Model

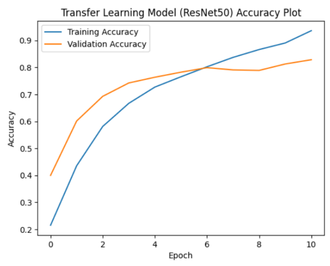


Figure: Accuracy vs # Epochs plot for ResNet50 Model

```
Test Loss: 0.6372362971305847
Test Accuracy: 0.8238000273704529
```

Here we can clearly see, that the ResNet50 model, just like VGG19 model, is outperforming all the previous CNN models. Due to much higher capacity attributed to ResNet's architecture, model being pretrained on large ImageNet dataset, among other reasons, the transfer learning model outperforms a simple 2-layer CNN model.

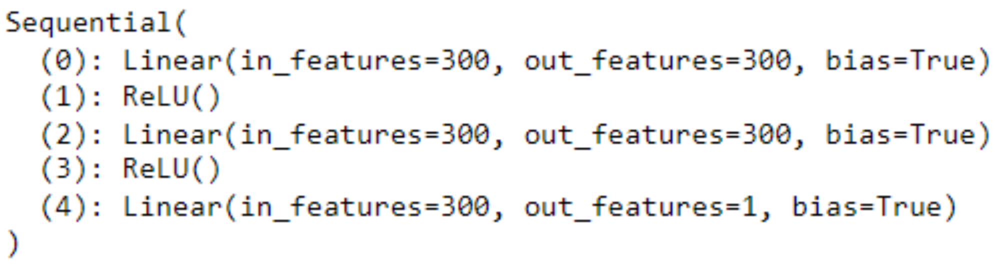
**Part 3: Dataset Shift Experiment on Transfer Learning Models**

From the comparison table below, we can infer that both the transfer learning models are much more robust to the dataset shifts, except for brightness shift. Among the two transfer learning models, VGG19 performs slightly better which may be due to low resolution of the images in CIFAR-10 dataset. ResNet architectures, including ResNet50, are designed to excel on larger and more complex datasets. If CIFAR-10 does not have sufficiently complex patterns, the benefits of ResNet's deep structure might be less pronounced.

| Dataset Shift Type | CNN Model | VGG19 Model | ResNet50 Model |
|---|---|---|---|
| No Dataset Shift | 0.8012 | 0.8309 | 0.8238 |
| Flip (Horizontal & Vertical) | 0.5823 | 0.5904 | 0.5983 |
| Shift (Width & Height) | 0.6143 | 0.7411 | 0.6816 |
| Zoom | 0.7324 | 0.8036 | 0.7779 |
| Rotation | 0.5792 | 0.6658 | 0.6369 |
| Brightness | 0.1080 | 0.1077 | 0.1022 |
| All combined | 0.1040 | 0.1018 | 0.0988 |

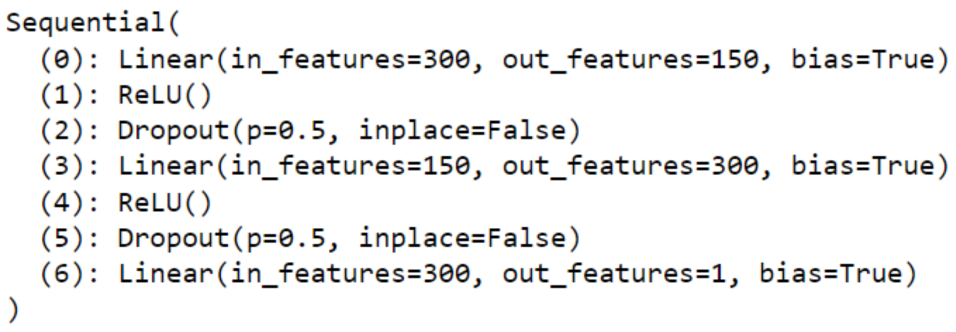
Table: Accuracy Score comparison of dataset shift experiment for CNN, VGG19, and ResNet50 pipelines

## 7. DAN Model Architecture (w/o dropout)



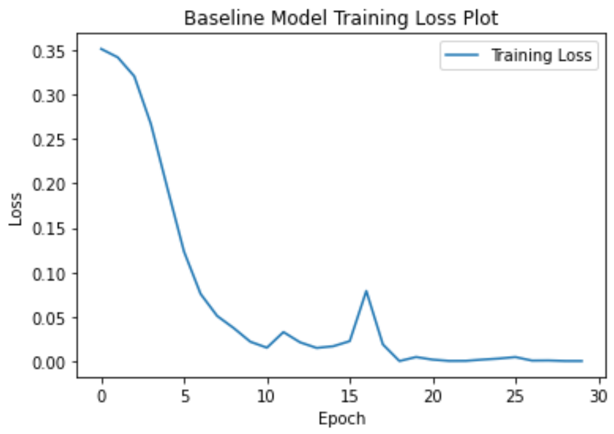
```
Sequential(
  (0): Linear(in_features=300, out_features=300, bias=True)
  (1): ReLU()
  (2): Linear(in_features=300, out_features=300, bias=True)
  (3): ReLU()
  (4): Linear(in_features=300, out_features=1, bias=True)
)
```

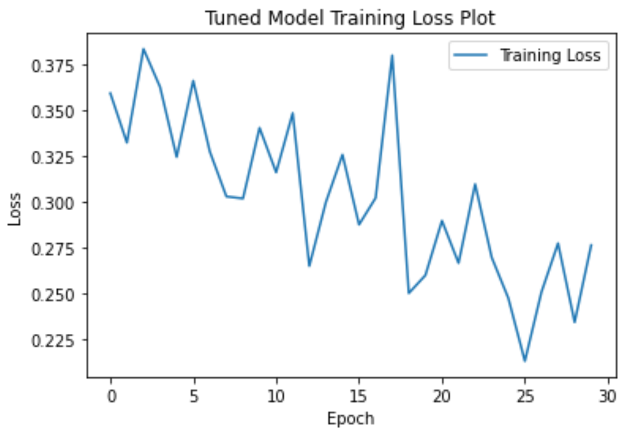
## 8. DAN Model Architecture (w/ dropout)



```
Sequential(
  (0): Linear(in_features=300, out_features=150, bias=True)
  (1): ReLU()
  (2): Dropout(p=0.5, inplace=False)
  (3): Linear(in_features=150, out_features=300, bias=True)
  (4): ReLU()
  (5): Dropout(p=0.5, inplace=False)
  (6): Linear(in_features=300, out_features=1, bias=True)
)
```

## 9. DAN Model Training Loss


Baseline Model Training Loss Plot
Training Loss
Loss
Epoch


Tuned Model Training Loss Plot
Training Loss
Loss
Epoch