Team: Jocelyn Luo, Shengkai Sun, Qi Yu

Computer Vision (Deep Learning Pipeline)

Data preprocessing.

Necessary libraries such as numpy, matplotlib, torch, and torchvision.transforms, torchvision.datasets.CIFAR10 etc. A transformation pipeline is set up using transforms.Compose, which includes converting to tensor and normalizing with the previously defined mean and std. The CIFAR10 dataset is loaded for training and testing, and the computation device (CPU/GPU) is determined. The default split setting i.e. 50,000 for training and 10,000 for testing. Mean and standard deviation values are defined, presumably for normalization.

mean_ = np.array([0.49139968, 0.48215841, 0.44653091])
std_ = np.array([0.24703223, 0.24348513, 0.26158784])

Deep learning models.

Necessary libraries such as torch, sklearn.metrices.f1_score, torch.utils.data, torch.optim, torch.nn, etc. Definition of a custom neural network class with specific layers and blocks, such as SEBlock and ConvNet. The structure includes convolutional layers, batch normalization, and fully connected layers.

```
==========================================================================
Layer (type:depth-idx)                   Param #
==========================================================================
SENet                                    --
├─Conv2d: 1-1                            1,792
├─BatchNorm2d: 1-2                       128
├─SEBlock: 1-3                           --
│    └─AdaptiveAvgPool2d: 2-1            --
│    └─Sequential: 2-2                   --
│    │    └─Linear: 3-1                  256
│    │    └─ReLU: 3-2                    --
│    │    └─Linear: 3-3                  256
│    │    └─Sigmoid: 3-4                 --
├─Conv2d: 1-4                            73,856
├─BatchNorm2d: 1-5                       256
├─SEBlock: 1-6                           --
│    └─AdaptiveAvgPool2d: 2-3            --
│    └─Sequential: 2-4                   --
│    │    └─Linear: 3-5                  1,024
│    │    └─ReLU: 3-6                    --
│    │    └─Linear: 3-7                  1,024
│    │    └─Sigmoid: 3-8                 --
├─MaxPool2d: 1-7                         --
├─Linear: 1-8                            81,930
==========================================================================
Total params: 160,522
Trainable params: 160,522
Non-trainable params: 0
==========================================================================
```

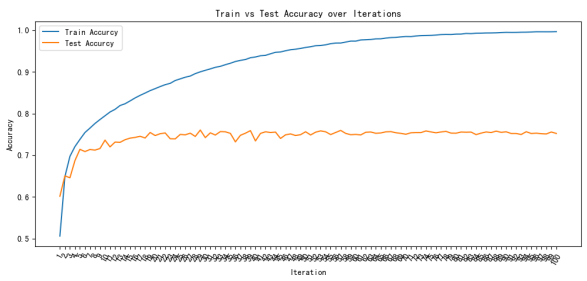
We used random search with 5-fold cross validation f1 score evaluation for 100 times and found the best hyperparameters as below.

```
best_hyperparams = {
  "learning_rate": 0.02999293598030286,
  "batch_size": 128,
  "optimizer": optim.Adagrad}
```

The best_hyperparameter is chosen by performing 5-fold cross validation take the set of hyper-parameters with highest f1_score

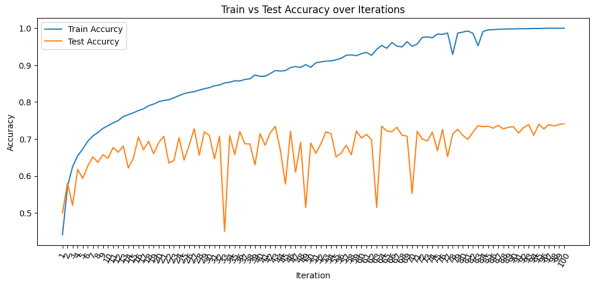
Results and discussion.

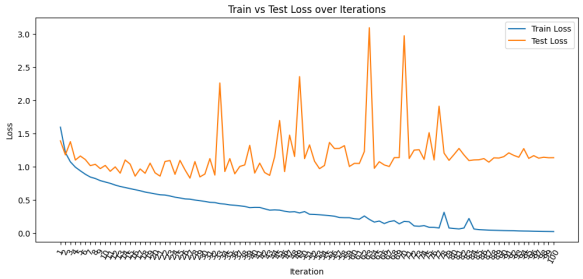
After tuning hyper-parameter:



The accuracy graph shows a stable and high training accuracy with a much closer test accuracy, indicating good generalization. Besides, the loss graph presents a consistently decreasing training loss and a relatively stable test loss, suggesting the model is learning effectively without overfitting.

Before tuning hyper-parameter:



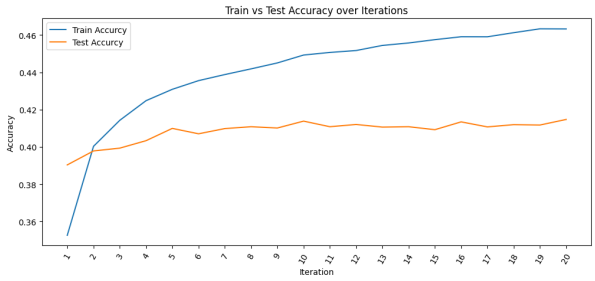


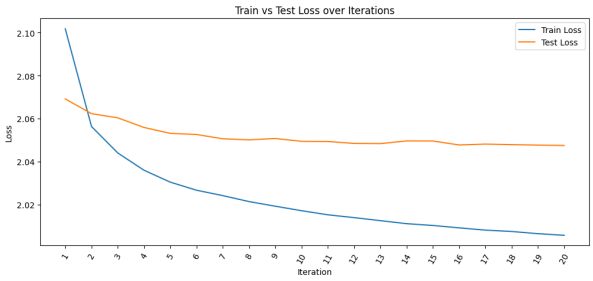
The accuracy graph displays a volatile test accuracy, with significant fluctuations throughout the training process, indicating instability in the model's learning. And, the loss graph shows a similar pattern of volatility, with test loss spiking at various points. This could be a sign of the model not learning consistently or effectively.

Comparison:

The stability of both the accuracy and loss graphs after hyperparameter tuning suggests that the chosen hyperparameters have helped the model to learn in a more stable and consistent manner. Before tuning, the erratic patterns indicate that the model may have been too complex or not adequately regularized, leading to unstable learning and performance. After tuning, the test accuracy is much closer to the training accuracy, and the test loss is lower and more stable, which implies improved generalization. Before tuning, the wide gap between training and test accuracy, and the high test loss peaks, suggest the model was overfitting to the training data and generalizing poorly to new data.

Traditional:





Shift 50% images of the test set by applying the following transformation (utilize the imgaug library):

```
self.augmenter = iaa.Sequential([
    iaa.SomeOf((2, 3), [
        iaa.Crop(percent=(0, 0.1)),
        iaa.Affine(translate_percent={"x": (-0.1, 0.1), "y": (-0.1, 0.1)}),
        iaa.Multiply((0.8, 1.2)),
        iaa.Rotate((-10, 10)),
        iaa.Fliplr(0.5),
        iaa.Flipud(0.5)
    ])
```

Randomly select 2 to 3 transformations from cropping, affining, changing brightness, rotating and flipping.

Results of DL:

```
Test loss (without shifting): 0.7373719151261486
Test accu (without shifting): 0.7545
```

```
Test loss (with shifting): 1.175307384342145
Test accu (with shifting): 0.638
```

Results of traditional pipeline:

```
Test loss (without shifting): 2.047091820571996
Test accu (without shifting): 0.4109
```

```
Test loss (with shifting): 2.145965327123168
Test accu (with shifting): 0.3071
```

1. **Deep Learning (DL) Model Performance**:The DL model achieves a test accuracy of 0.7545 without data shifting and 0.638 with data shifting. The test loss for the DL model is lower without shifting (0.737) compared to with shifting (1.175), which indicates that the model is more precise when the data is not altered. From the accuracy plot, the DL model shows a significant gap between training and testing accuracy, which could indicate overfitting since the model performs well on training data but less so on unseen test data.
2. **Traditional Model Performance**: The traditional model has a lower test accuracy (0.4109 without shifting, 0.3071 with shifting) than the DL model, suggesting that the DL model is more adept at capturing complex patterns in the data. The test loss for the traditional model is higher (2.047 without shifting, 2.145 with shifting) than that of the DL model, which further indicates that the traditional model is less precise in its predictions. The traditional model's accuracy plot shows a steady increase in training accuracy but a plateau in test accuracy, suggesting that there's a limit to how well the model can generalize from the training data.
3. **Comparison Between DL and Traditional Models**: The DL model outperforms the traditional model in both test accuracy and test loss, which is consistent with expectations as deep learning models typically excel at capturing nonlinearities and complex features in image data. The presence of data shifting seems to affect both models, with a more pronounced impact on the traditional model. This could suggest that the DL model has better learned representations that are more robust to variations in the data.
4. **Data Shifting Impact**: Data shifting, which likely refers to some form of data augmentation or perturbation, seems to have a detrimental effect on both models' performance. This is not

uncommon, as data shifting can introduce noise or variations that the models have not encountered during training. The lesser impact of shifting on the DL model compared to the traditional model might be indicative of the former's higher capacity to generalize from complex data.

Natural Language Processing (Deep Learning Pipeline)

Deep learning models:

Data preprocessing for LSTM:

We create the word-to-index mapping, which includes the top 1000 highest-frequency words and start-of-sentence and end-of-sentence tokens. We convert every review into a vector of length 80 consisting of SOS, word, EOS, and padding indexes. Then we constructed train and test dataloaders which are fed into the LSTM model.

We used principal software packages enlisting below:

```
Torch, torch.nn as nn, torch.optim as optim, torch.utils.data import Dataset,
DataLoader, torch.utils.data import TensorDataset
```

The neural network architecture used:

We write LSTM architecture to analyze our data.

```
==========================================================
Layer (type:depth-idx)                   Param #
==========================================================
SentimentRNN                             --
├─Embedding: 1-1                         64,256
├─LSTM: 1-2                              856,064
├─Dropout: 1-3                           --
├─Linear: 1-4                            257
├─Sigmoid: 1-5                           --
==========================================================
Total params: 920,577
Trainable params: 920,577
Non-trainable params: 0
==========================================================
```

First, we have an embedding layer that converts words to dense vectors of fixed size. Then we implement an LSTM network. It takes word embeddings as input and processes them through LSTM cells. The number of layers is determined by no_layers, and hidden_dim specifies the number of hidden units in each LSTM cell. Then we have a dropout layer that is applied to the output of the LSTM layer to prevent overfitting. It randomly sets a fraction of input units to zero during training.

Hyperparameter Tuning:

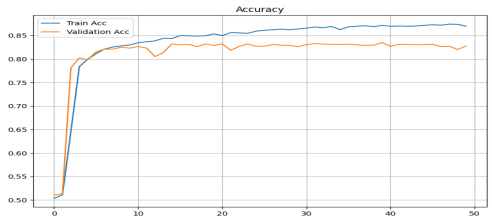
To vary the model architecture, we perform hyperparameter tuning over all combinations of the following learning rate, embedding dimension for generating the input word embeddings, the hidden dimension of the LSTM layer, and the number of LSTM layers. We ran each combination over 30 epochs and picked the one with the best average validation accuracy.

```
param_grid = {
    'learning_rate': [0.001, 0.01, 0.1],
    'embedding_dim': [32, 64, 128],
    'hidden_dim': [128, 256, 512],
    'no_layers': [1, 2, 3]}
```

Results and Discussion:

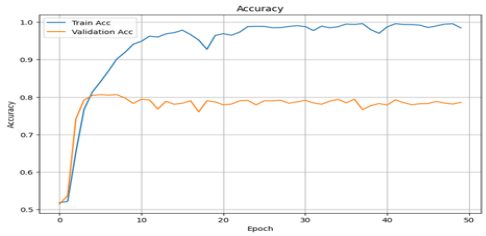
Training/validation/test results:

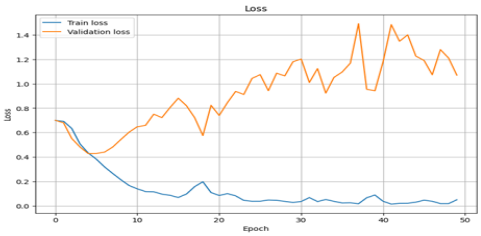
Using a subset of 10,000 data, our baseline model({'embedding_dim': 32, 'hidden_dim': 128, 'learning_rate': 0.001, 'no_layers': 1}) has an average validation accuracy of 0.68. After hyperparameter tuning, we arrive at **the best parameters {'embedding_dim': 32, 'hidden_dim': 128, 'learning_rate': 0.01, 'no_layers': 1}**, reaching a validation accuracy of 0.79. This suggests that relatively simple LSTM architecture already does a great job. We then train the LSTM model using full data with these parameters, arriving at a test accuracy of 0.83.



Next, we use pre-trained word2vec trained on a part of the Google News dataset in place of the embedding layer. For each sentence, we directly appended the word vector in order(maintaining the top 1000 words constraint). By using these pre-trained vectors, we allow the model to leverage information learned from a broader context.

Due to computation limitations, we subset our data to 8000. Our baseline model({'hidden_dim': 128, 'learning_rate': 0.001, 'no_layers': 1}) has an average validation accuracy of 0.70. Then we conducted a hyperparameter tuning using a similar approach to find the best parameters to be similar to last time: {'hidden_dim': 128, 'learning_rate': 0.01, 'no_layers': 1}, which has an average validation accuracy of 0.77. Even with a subset of data, the final test accuracy approaches 0.8, suggesting with full data, we can see even higher accuracy.





Comparison to performance of your traditional pipeline from Project Milestone 2:

In Project Milestone 2, the best accuracy we obtained is 0.89 from logistic regression on unigrams, which is slightly better than the 0.83 obtained from the LSTM model. This can be attributed to the fact that the sentiment prediction task is relatively straightforward and does not involve capturing long-term dependencies in language. Logistic regression treats each word as a feature and usually, the occurrence of positive and negative words is sufficient to help predict the outcome. This is also supported by the fact that our best parameter for LSTM also leads to a relatively simple LSTM architecture.

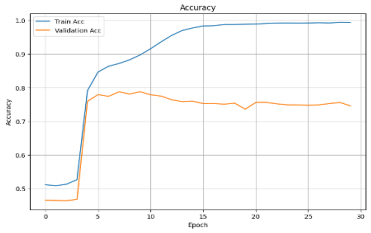
Datashift results:

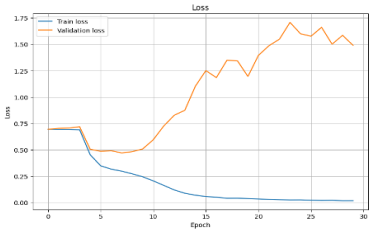
Observing the distribution of lemmatized token length, after measures done on misbalancing labels and resampling, we implemented three datashifts in terms of sentence length, train on short (lemmatized token length < 200) and test on long (lemmatized token length >= 200) , train

on long (lemmatized token length >= 50) and test on short (lemmatized token length < 50) , test on very short (lemmatized token length >= 200) and test on very long (lemmatized token length < 200) . Following results are achieved:

train on short (lemmatized token length < 200) and test on long (lemmatized token length >= 200)

- Traditional (TDF bigram & Logistic): 87
- DL train accuracy: 99.35 DL Test Accuracy: 74.58

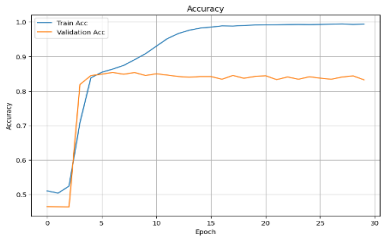


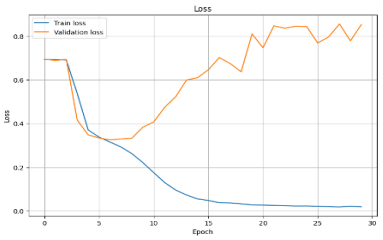


	precision	recall	f1-score	support
0	0.85	0.86	0.86	2856
1	0.88	0.87	0.87	3296
accuracy			0.87	6152
macro avg	0.86	0.87	0.86	6152
weighted avg	0.87	0.87	0.87	6152

train on long (lemmatized token length >= 50) and test on short (lemmatized token length < 50)

- Traditional(TDF bigram & Logistic): 85
- DL train accuracy: 99.40 DL Test Accuracy: 83.24

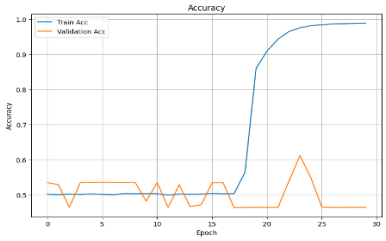


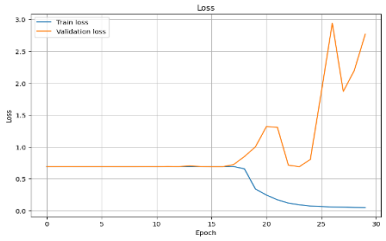


	precision	recall	f1-score	support
0	0.85	0.83	0.84	3202
1	0.85	0.88	0.86	3672
accuracy			0.85	6874
macro avg	0.85	0.85	0.85	6874
weighted avg	0.85	0.85	0.85	6874

test on very short (lemmatized token length < 50) and test on very long (lemmatized token length >= 200)

- Traditional(TDF bigram & Logistic): 84
- DL train accuracy: 98.91 DL Test Accuracy: 46.49 (breaks the model)





	precision	recall	f1-score	support
0	0.82	0.84	0.83	2856
1	0.86	0.84	0.85	3296
accuracy			0.84	6152
macro avg	0.84	0.84	0.84	6152
weighted avg	0.84	0.84	0.84	6152

For Deep learning, from the results graphs shown, only the last one, test on very short (lemmatized token length < 200) and test on very long (lemmatized token length < 200), significantly breaks the model such that the test accuracy is almost doing “random guessing”. Besides, we can also observe a difference of 10 percent in test accuracy between the first two groups (train short test long) and (train long test short). The results suggest that long sentences usually consist of more textual information than short sentences. This can be intuitively verified if we think about how we encode the sentences into a vector of fixed length (padding the short sentences with zeros, unmeaningful information).

For Traditional learning, the difference between the results is not as obvious as deep learning, but still, the results for test on very short (lemmatized token length < 200) and test on very long (lemmatized token length < 200), is the lowest. We think the reason that the traditional model is not so bad is because we passed in meaningful lemmatized tokens instead of encoded sentences(some include padding) to the traditional model.