Assignment 1 - CNN, RNN and LSTM Architectures

About the report

This report outlines Assignment 1 for the Deep Learning course, focusing on neural network models. We trained these models on an image dataset comprising three classes and 30,000 samples.

In the first part of the assignment, we implemented the VGG13 network and applied techniques such as regularization and early stopping to mitigate overfitting. We compared and analyzed the results obtained with and without these techniques.

Moving on to the second part, we employed the RESNET model, renowned for its use of residual layers in the convolutional network. Our objective was to train the model and evaluate its performance on test and validation data, considering metrics like accuracy and loss.

Furthermore, we explored two additional methods aimed at enhancing model performance and preventing overfitting.

Talking about the third and fourth parts we’re dealing with the time series and the sentiment analysis data, In Part III, we undertake time-series forecasting using RNN and LSTM methods, Similarly, Part IV focuses on sentiment analysis using LSTM models, following a similar approach to dataset preparation, model building, analysis, and reporting, aiming for an accuracy exceeding 75% for the final model.

The report begins by introducing the dataset used in the assignment and outlines the different sections. A detailed table of contents allows for easy navigation through the report's contents.

About the dataset [PART 1 & 2]

The CNN dataset utilized in this assignment comprises 30,000 samples, with 10,000 examples belonging to each of the three categories. Each sample consists of a 64×64 image. Here’s the preview of the images from each class

Data Splitting

We split the dataset into training, validation, and test sets:

- **Training Data**: 75% of the dataset is allocated for training.
- **Validation Data**: Half of the remaining data after training split is assigned for validation.
- **Test Data**: The remaining half of the data serves as the test set.

Batch Size

We utilize a batch size of 32 for training, validation, and testing.

Dataset Distribution

- **Training set size**: 22,500 samples.
- **Validation set size**: 3,750 samples.
- **Test set size**: 3,750 samples.

Part I: CNN Classification

The CNN Model

For this image classsification task we used the `VGG_13_CNN` neural network architecture It consists of convolutional layers followed by fully connected layers for feature extraction and classification, respectively. Below is a breakdown of its architecture:

First the conv layers

- **Input Channels**: The network accepts input images with 3 channels (RGB).
- Then It has multiple convolutional blocks, each containing two convolutional layers followed by a ReLU activation function.
  - **Convolutional Layers**: Each convolutional layer has a kernel size of 3×3 and a padding of 1 to preserve spatial dimensions.
  - **Activation**: ReLU activation functions are applied after each convolution to introduce non-linearity.
- **Pooling Layers**: After each pair of convolutional layers, a max-pooling layer with a kernel size of 2×2 and stride of 2 is applied to reduce spatial dimensions.

Next FC Layers

- The output from the convolutional layers is flattened to be passed into the fully connected layers.
- It consists of three fully connected layers.
  - **Hidden Layers**: Two hidden layers with 4096 neurons each, followed by ReLU activation functions.
  - **Output Layer**: The final fully connected layer produces the output logits for classification into the specified number of classes.

Forward Pass

- **Convolutional Layers**: Input images pass through the convolutional layers, extracting features through convolutions and activations.
- **Flattening**: The output feature maps are flattened to be fed into the fully connected layers.
- **Fully Connected Layers**: The flattened features are passed through the fully connected layers, performing classification based on learned features.

```
import torch
import torch.nn as nn
import torch
import torch.nn as nn

class VGG_13_CNN(nn.Module):
    def __init__(self, classes):
        super(VGG_13_CNN, self).__init__()
        self.conv = nn.Sequential(

            # First 2conv-3 64
            nn.Conv2d(3, 64, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(64, 64, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),

            #now we go with the macpool
            nn.MaxPool2d(kernel_size=2, stride=2),

            # 2 conv with 128
            nn.Conv2d(64, 128, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(128, 128, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),

            #Max pool again
            nn.MaxPool2d(kernel_size=2, stride=2),

            # 2 - 256
            nn.Conv2d(128, 256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(256, 256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),

            nn.MaxPool2d(kernel_size=2, stride=2),

            # 2 512
            nn.Conv2d(256, 512, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(512, 512, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),

            #maxxxxxxxx pooooll
            nn.MaxPool2d(kernel_size=2, stride=2),

            # 2 512
            nn.Conv2d(512, 512, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(512, 512, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),

            nn.MaxPool2d(kernel_size=2, stride=2)
        )

        self.fc = nn.Sequential(
            nn.Linear(512*4, 4096),
            nn.ReLU(inplace=True),
            nn.Linear(4096, 4096),
            nn.ReLU(inplace=True),
            nn.Linear(4096, classes),
            # nn.Softmax(dim= 1)
        )


    def forward(self, x):
        x = self.conv(x)
        x = x.view(x.size(0), -1)
        x = self.fc(x)

        return x
```

Performance analysis of base VGG 13

Using this VGG 13 network we trained the network and performed teh performance analysis on teh test and the validation dataset. Here are the results.

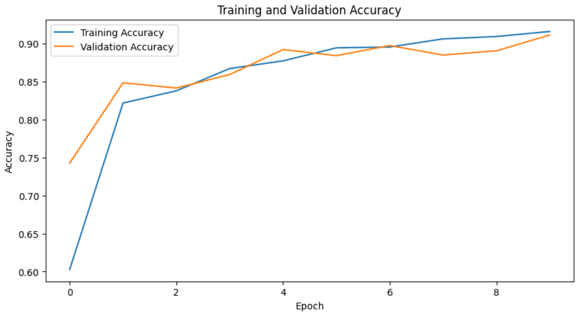
The training consists of 10 epochs

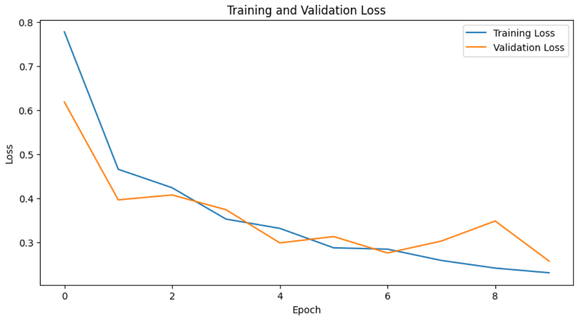
Loss on Validation and Training

- The training loss steadily decreased from 0.7782 in the first epoch to 0.2309 in the final epoch.
- Similarly, the validation loss decreased consistently from 0.619 to 0.7429 over the 10 epochs.

Accuracy on Validation and Training

- The training accuracy improved from 0.6028 to 0.9161, showing that the model's performance on the training data significantly improved over the epochs.
- Similarly The validation accuracy also increased steadily from 0.7429 to 0.9115.

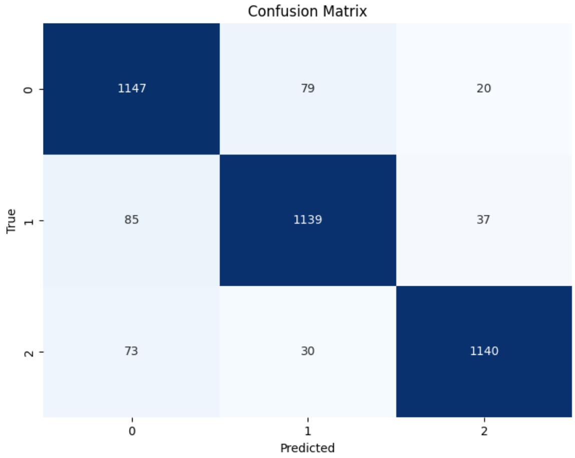




Test Accuracy and Performance Metrics

- The test accuracy achieved after training was 0.8995, indicating the model's ability to generalize well to unseen data.
- F1 Score: 0.9138
- Precision: 0.9146
- Recall: 0.9136

These metrics suggest that the model performs well across different aspects of classification, including precision, recall, and overall F1 score.



Analyzing the confusion matrix we can say that this base model is very very slightly better in identifying the class 0 compared to the class 1 and class 2 and it had lower accuracy in identifying the class 1.

Let’s use some Techniques

Here we used four techniques, starting with regularization, followed by the drop out, early stopping and finally data augmentation. In this section we’ll analysis how each technique impacted the model performances compared to the base model and finally we have the overviw of all the models performance.

Regularization

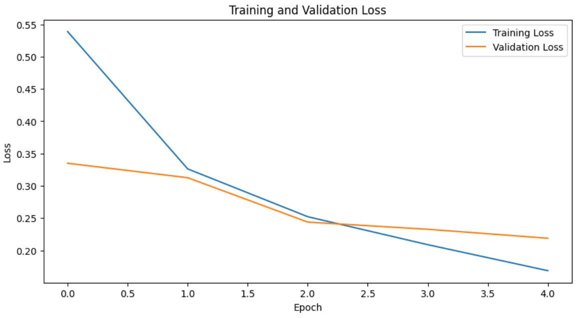
Regularization is a technique used to prevent overfitting by adding a wright term to the loss function, discouraging complex models that fit the training data too closely.

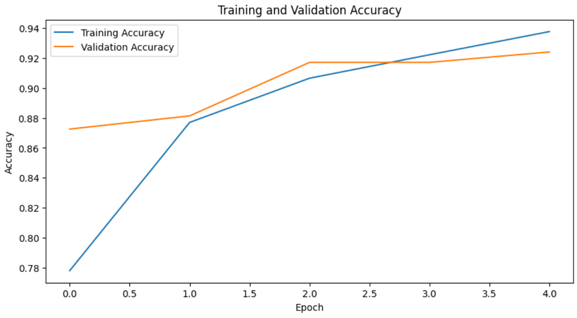
In the provided code, L2 regularization is applied to the convolutional and fully connected layers of the VGG_13_CNN_R model. This is achieved by adding a weight decay term to the optimizer's parameters during training, effectively penalizing large weights.

```
weight_decay = 0.00001
model_reg = VGG_13_CNN_R(classes, weight_decay)
optimizer = optim.Adam(model_reg.parameters(), lr=0.0001, weight_decay=weight_decay)
```

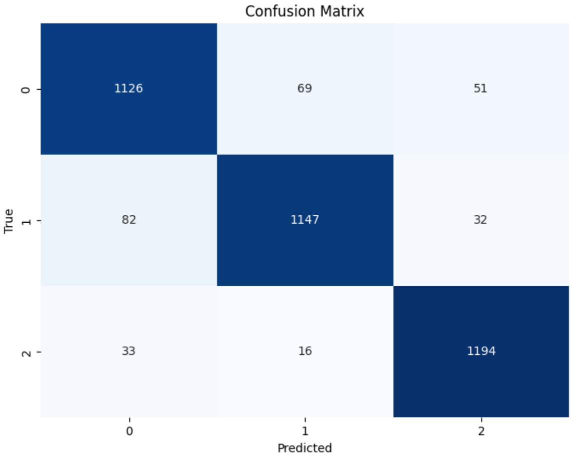
Performance:

- When we apply the regularization, the **training loss** steadily decreased from 0.5388 in the first epoch to 0.1687 in the final epoch.
- Similarly, the **validation loss** decreased consistently from 0.3350 to 0.2190 over the 5 epochs.
- Coming to the accuracies, **training accuracy** improved from 0.7780 to 0.9376, while the **validation accurac**y improved from 0.8725 to 0.9240.





- COming to the test data we achieved accuracy of 0.9245, with an F1 Score of 0.9244, Precision of 0.9245, Recall of 0.9245, and Loss of 0.2117.
- Overall the model performed well and from the confusion matrix we can see model was able to detect the class2 comapred to other classes, though it’s a very minute difference, and it quite confused when it comes to the class 0.



True \ Predicted	0	1	2
0	1126	69	51
1	82	1147	32
2	33	16	1194

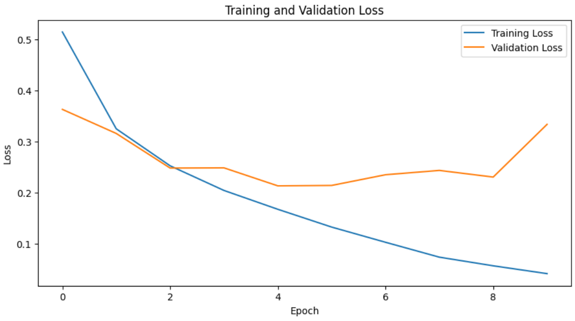
Dropout

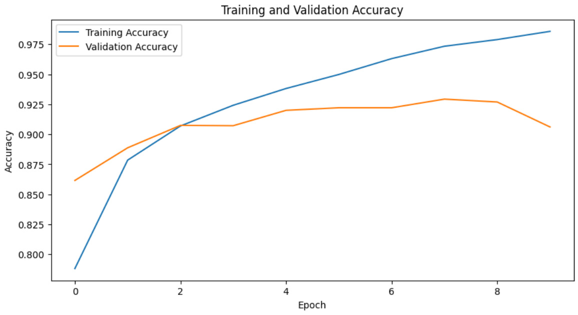
Here we randomly dropping (setting to zero) a proportion of the neurons in a layer during training. This forces the network to learn redundant representations, reducing reliance on specific neurons. In the VGG_13_CNN_D model, dropout layers with dropout rates of 0.3 and 0.1 are applied after the first and second fully connected layers, respectively. Here's teh shortened version of the used code

```
class VGG_13_CNN_D(nn.Module):
    def __init__(self, classes):
        super(VGG_13_CNN_D, self).__init__()
        self.fc = nn.Sequential(
            ...
            nn.Dropout(0.3),
            ...
            nn.Dropout(0.1),
            ...
        )
```

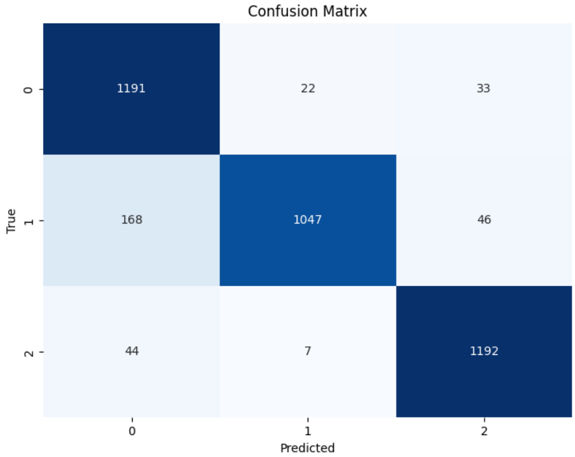
Performance

- Dropout indeed gives the variation in the loss, as it decreased from 0.5149 in the first epoch to 0.0417 in the final epoch.
- the validation loss decreased consistently from 0.3633 to 0.3341 over the 10 epochs.
- Training accuracy improved from 0.7882 to 0.9858, while the validation accuracy improved from 0.8616 to 0.9061.





- the test accuracy achieved after training was 0.9147, with an F1 Score of 0.9144, Precision of 0.9201, Recall of 0.9147, and Loss of 0.3025.
- The dropout indeed helped in reducing the overfit, as you can seefrom the confusion matrix here the model slightly perfromed well in detecting the class 2 and 1 which is not the case for the above regularization applied one. Class 1 is got the least correct predictions.



Early Stopping

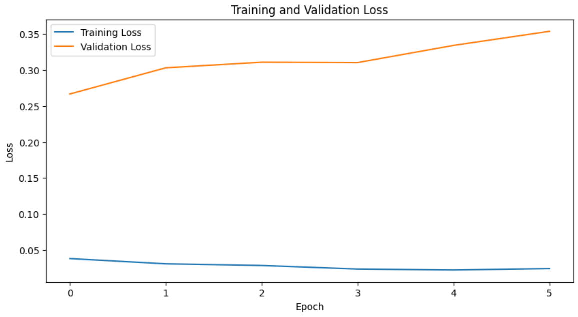
Early stopping is used to prevent overfitting by monitoring the model's performance on a validation set during training and stopping when the performance starts to degrade.

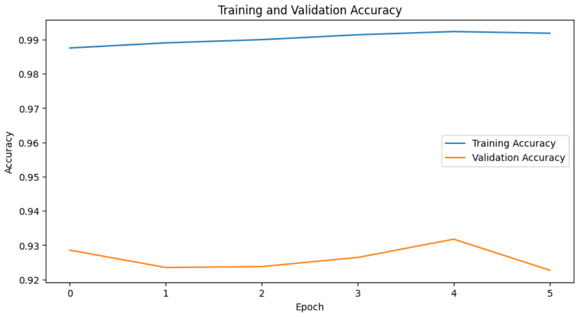
In the `train_model_est` function,we defined the early stopping tracking the validation loss. If the validation loss does not improve for a specified number of epochs (patience), training is stopped early to prevent overfitting.

```
def train_model_est(model, train_loader, valid_loader, optimizer, criterion, num_epochs, pati
    ...
    for epoch in range(num_epochs):
        ...
        # Check for early stopping
        if avg_valid_loss < best_valid_loss:
            best_valid_loss = avg_valid_loss
            current_patience = 0
        else:
            current_patience += 1
            if current_patience >= patience:
                print("Early stopping triggered.")
                break
```

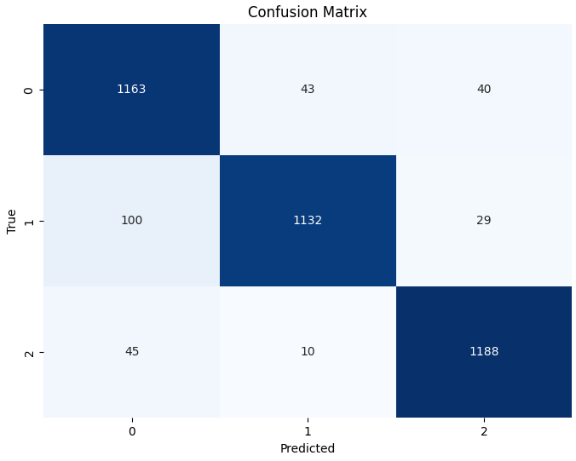
Performance

- When applied early stopping, the training loss decreased from 0.0380 in the first epoch to 0.0221 in the final epoch.
- the validation loss decreased consistently from 0.2668 to 0.3343 over the 10 epochs.
- When applied early stopping, the training accuracy improved from 0.9876 to 0.9924, while the validation accuracy improved from 0.9285 to 0.9317.





- Test accuracy achieved after training was 0.9288, with an F1 Score of 0.9289, Precision of 0.9299, Recall of 0.9288, and Loss of 0.3208.
- Examining the confusion matrix you can see that the class 2 is shown better predicted and both the class 0 and 1 have least correctly predicted.



	0	1	2
0	1163	43	40
1	100	1132	29
2	45	10	1188

Data Augmentation

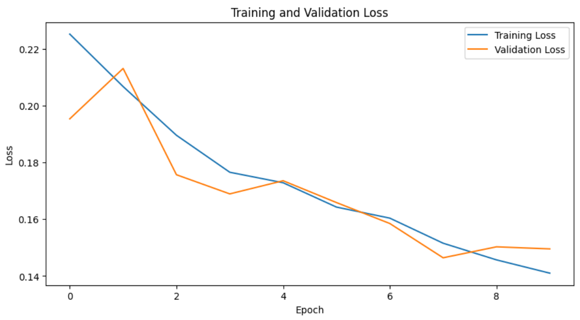
it can be used to artificially increase the diversity of training data by applying transformations such as rotation, flipping, scaling, and cropping.

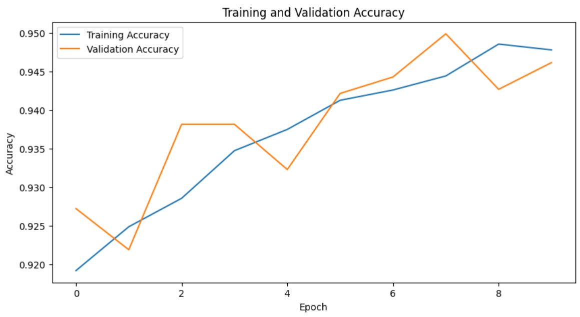
In the provided code, data augmentation is applied to the input images using PyTorch's `transforms` V2 module. Random transformations such as horizontal and vertical flips, random rotation, and color jittering are applied to the training images to create variations of the original data, helping the model generalize better.

```
transform_data_n = transforms.Compose([
    transforms.Resize((64, 64)),
    transforms.RandomHorizontalFlip(),
    transforms.RandomVerticalFlip(),
    transforms.RandomRotation(20),
    transforms.ColorJitter(brightness=0.4, contrast=0.3, saturation=0.1),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
])
```

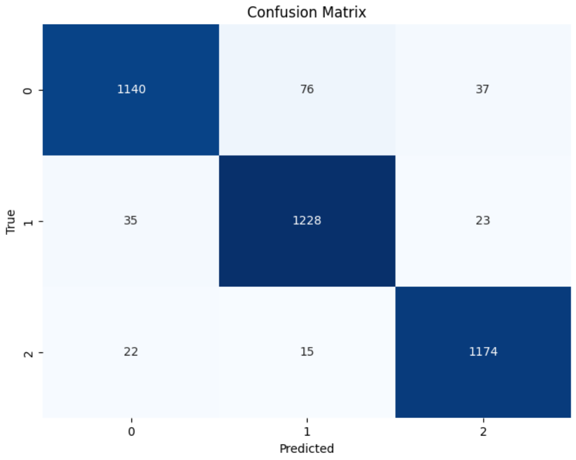
Perfromance

- With applied data augmentation, the training loss declined from 0.2252 in the first epoch to 0.1410 in the final epoch.
- Similarly, the validation loss decreased consistently from 0.1954 to 0.1496 over the 5 epochs.
- Thehe training accuracy improved from 0.9192 to 0.9478, while the validation accuracy improved from 0.9272 to 0.9461.





- The test results has shown some variations when data augmentation is applied test accuracy achieved after training was 0.9445, with an F1 Score of 0.9444, Precision of 0.9447, Recall of 0.9445, and Loss of 0.1620.
- Matrix tells us that the class 0 is the least correctly predicted compared to the other classes by the model



Overview

This table provides the performance of different models

Model	Train Accuracy (Low-High)	Validation Accuracy (Low-High)	Train Loss (Low-High)	Validation Loss (Low-High)	Test Accuracy	Precision	Recall
Base VGG 13	0.6028 - 0.9161	0.7429 - 0.9115	0.2309 -	0.2575 - 0.6189	0.9136	0.9146	0.9136

Model	Train Accuracy (Low-High)	Validation Accuracy (Low-High)	Train Loss (Low-High)	Validation Loss (Low-High)	Test Accuracy	Precision	Recall
			0.7782				
Regularization	0.7780 - 0.9376	0.8725 - 0.9240	0.1687 - 0.5388	0.2190 - 0.3350	0.9245	0.9245	0.9245
Dropout	0.7882 - 0.9858	0.8616 - 0.9317	0.0417 - 0.5149	0.2136 - 0.3633	0.9147	0.9201	0.9147
Early Stopping	0.9876 - 0.9924	0.9235 - 0.9317	0.0221 - 0.0380	0.2668 - 0.3109	0.9288	0.9299	0.9288
Data Augmentation	0.9192 - 0.9478	0.9219 - 0.9499	0.1410 - 0.2252	0.1464 - 0.2131	0.9445	0.9447	0.9445

From the comparison table, we can observe the following:

Based on the analysis provided:

1. **Best Performing Model**:
    - The model utilizing **Data Augmentation** appears to be the best-performing one. It demonstrates the highest test accuracy, precision, recall, and relatively low loss compared to other models. Additionally, it shows consistently high performance across both training and validation datasets.
2. **Best Aspects of Each Model**:
    - **Regularization**: Effective in improving validation accuracy and reducing overfitting, as indicated by the relatively low validation loss.
    - **Early Stopping**: Prevents overfitting by closely aligning training and validation metrics and maintaining high accuracy on both.
    - **Data Augmentation**: Shows superior performance across all aspects, indicating its effectiveness in improving generalization and robustness of the model.
3. **Least Performing Model**:
    - The **Base VGG 13** model appears to perform the least effectively among the compared models. It exhibits relatively lower accuracy and higher loss compared to models with regularization techniques and data augmentation.
4. **Mediocre Performance**:
    - The **Dropout** model shows mediocre performance compared to other techniques. While it improves overfitting to some extent, it doesn't show as significant improvements in accuracy and loss as regularization or data augmentation methods.

In summary, while all techniques contribute to some improvement over the base model, data augmentation stands out as the most effective method in enhancing model performance, followed by regularization techniques. Dropout regularization and early stopping show moderate improvements, with dropout being slightly less effective compared to the other techniques.

Part II: Implementing ResNet Architecture

The RESNET 18

Here we are using the Residual Neural Network (ResNet), specifically ResNet-18, which is designed for image classification tasks. It consists the following

1. **Residual Block (`ResidualBlock` class):**
    - This block forms the basic building block of the ResNet architecture.
    - It consists of two convolutional layers with batch normalization and ReLU activation functions.
    - This residual block helps in introduction of skip connections, allowing the network to learn residual functions instead of directly learning the desired underlying mapping.

- The first convolutional layer (`self.rone`) applies a 3×3 convolution with stride `stride` and padding 1, followed by batch normalization and ReLU activation.
- The second convolutional layer (`self.rtwo`) applies another 3×3 convolution with stride 1 and padding 1, followed by batch normalization.
- The `downsample` parameter is used for downsampling the input if necessary to match the dimensions for the addition operation in case of a change in dimensions.
- The output of the block is obtained by adding the residual (input) to the output of the second convolutional layer and applying ReLU activation.

2. **ResNet (`ResNet` class):**
    - This class defines the overall ResNet architecture by stacking multiple residual blocks.
    - It begins with an initial convolutional layer (`self.conv1`) followed by batch normalization and ReLU activation.
    - A max-pooling layer is applied to downsample the feature maps.
    - The residual blocks are stacked in four different stages (`self.layer1` to `self.layer4`), each containing a certain number of residual bloc
    - The number of residual blocks in each stage is specified by the `layers`
    - The `make_layer` function is responsible for creating each stage by stacking multiple residual blocks.
    - The final layers has of an adaptive average pooling layer (`self.avgpool`) followed by a fully connected layer (`self.fc`) to produce the output logits for classification.

Summary of the model

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1           [-1, 64, 64, 64]           1,792
       BatchNorm2d-2           [-1, 64, 64, 64]             128
              ReLU-3           [-1, 64, 64, 64]               0
         MaxPool2d-4           [-1, 64, 32, 32]               0
            Conv2d-5           [-1, 64, 32, 32]          36,928
       BatchNorm2d-6           [-1, 64, 32, 32]             128
              ReLU-7           [-1, 64, 32, 32]               0
            Conv2d-8           [-1, 64, 32, 32]          36,928
       BatchNorm2d-9           [-1, 64, 32, 32]             128
             ReLU-10           [-1, 64, 32, 32]               0
    ResidualBlock-11           [-1, 64, 32, 32]               0
           Conv2d-12           [-1, 64, 32, 32]          36,928
      BatchNorm2d-13           [-1, 64, 32, 32]             128
             ReLU-14           [-1, 64, 32, 32]               0
           Conv2d-15           [-1, 64, 32, 32]          36,928
      BatchNorm2d-16           [-1, 64, 32, 32]             128
             ReLU-17           [-1, 64, 32, 32]               0
    ResidualBlock-18           [-1, 64, 32, 32]               0
           Conv2d-19          [-1, 128, 16, 16]          73,856
      BatchNorm2d-20          [-1, 128, 16, 16]             256
             ReLU-21          [-1, 128, 16, 16]               0
           Conv2d-22          [-1, 128, 16, 16]         147,584
      BatchNorm2d-23          [-1, 128, 16, 16]             256
           Conv2d-24          [-1, 128, 16, 16]           8,320
      BatchNorm2d-25          [-1, 128, 16, 16]             256
             ReLU-26          [-1, 128, 16, 16]               0
    ResidualBlock-27          [-1, 128, 16, 16]               0
           Conv2d-28          [-1, 128, 16, 16]         147,584
      BatchNorm2d-29          [-1, 128, 16, 16]             256
             ReLU-30          [-1, 128, 16, 16]               0
           Conv2d-31          [-1, 128, 16, 16]         147,584
      BatchNorm2d-32          [-1, 128, 16, 16]             256
             ReLU-33          [-1, 128, 16, 16]               0
    ResidualBlock-34          [-1, 128, 16, 16]               0
           Conv2d-35            [-1, 256, 8, 8]         295,168
      BatchNorm2d-36            [-1, 256, 8, 8]             512
             ReLU-37            [-1, 256, 8, 8]               0
           Conv2d-38            [-1, 256, 8, 8]         590,080
      BatchNorm2d-39            [-1, 256, 8, 8]             512
           Conv2d-40            [-1, 256, 8, 8]          33,024
      BatchNorm2d-41            [-1, 256, 8, 8]             512
             ReLU-42            [-1, 256, 8, 8]               0
    ResidualBlock-43            [-1, 256, 8, 8]               0
           Conv2d-44            [-1, 256, 8, 8]         590,080
      BatchNorm2d-45            [-1, 256, 8, 8]             512
             ReLU-46            [-1, 256, 8, 8]               0
           Conv2d-47            [-1, 256, 8, 8]         590,080
      BatchNorm2d-48            [-1, 256, 8, 8]             512
             ReLU-49            [-1, 256, 8, 8]               0
    ResidualBlock-50            [-1, 256, 8, 8]               0
           Conv2d-51            [-1, 512, 4, 4]       1,180,160
      BatchNorm2d-52            [-1, 512, 4, 4]           1,024
             ReLU-53            [-1, 512, 4, 4]               0
           Conv2d-54            [-1, 512, 4, 4]       2,359,808
      BatchNorm2d-55            [-1, 512, 4, 4]           1,024
           Conv2d-56            [-1, 512, 4, 4]         131,584
      BatchNorm2d-57            [-1, 512, 4, 4]           1,024
             ReLU-58            [-1, 512, 4, 4]               0
    ResidualBlock-59            [-1, 512, 4, 4]               0
           Conv2d-60            [-1, 512, 4, 4]       2,359,808
      BatchNorm2d-61            [-1, 512, 4, 4]           1,024
             ReLU-62            [-1, 512, 4, 4]               0
           Conv2d-63            [-1, 512, 4, 4]       2,359,808
      BatchNorm2d-64            [-1, 512, 4, 4]           1,024
             ReLU-65            [-1, 512, 4, 4]               0
    ResidualBlock-66            [-1, 512, 4, 4]               0
AdaptiveAvgPool2d-67            [-1, 512, 1, 1]               0
           Linear-68                    [-1, 3]           1,539
================================================================
Total params: 11,175,171
Trainable params: 11,175,171
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.05
Forward/backward pass size (MB): 20.50
Params size (MB): 42.63
Estimated Total Size (MB): 63.18
----------------------------------------------------------------
```

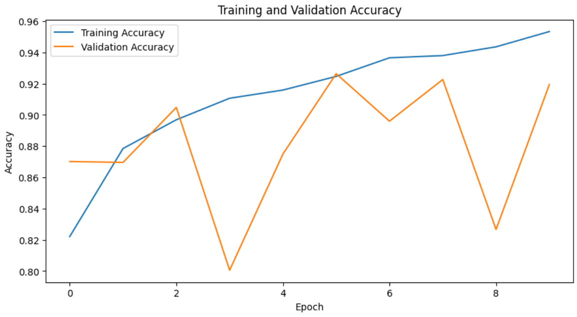
Performance analysis of base ResNet 18

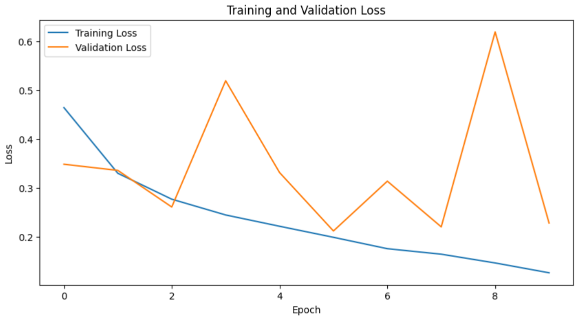
Loss on Validation and Training:

- The training loss steadily decreased from 0.4640 in the first epoch to 0.1264 in the final epoch, indicating that the model's performance improved as training progressed.
- The validation loss fluctuated but generally decreased from 0.3483 to 0.2281 over the 10 epochs, with some fluctuations in between.

Accuracy on Validation and Training:

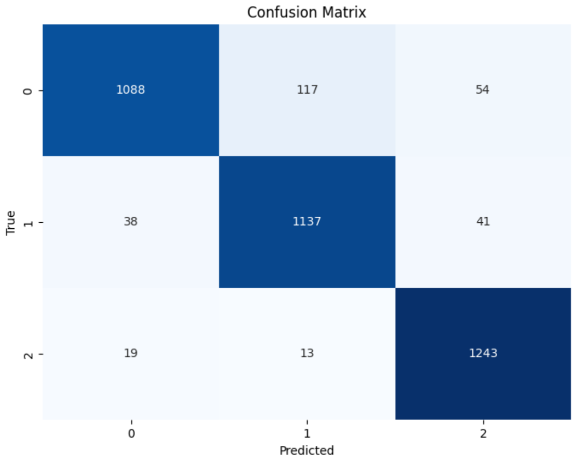
- The training accuracy improved from 0.8221 to 0.9534, showing that the model's performance on the training data significantly improved over the epochs.
- Similarly, the validation accuracy increased steadily from 0.8701 to 0.9195.





Test Accuracy and Performance Metrics:

- After the training the model was able to get accuracy of 0.9248, which meants this is performing better than the PART 1 VGG and VGG models with applied techniques.
- F1 Score: 0.924
- Precision: 0.9259
- Recall: 0.9248
- The confusion matrix shows similar trend as all the above VGG versions, the class 2 is easy to predict and the 0 is the tough to predict one.



Let’s use some Techniques

Drop out

Here we are asked to use some two techniques to prevent the overfitting of the data, so we used Dropout and the early stopping. Let’s have a look into those in detail and analyze their performance

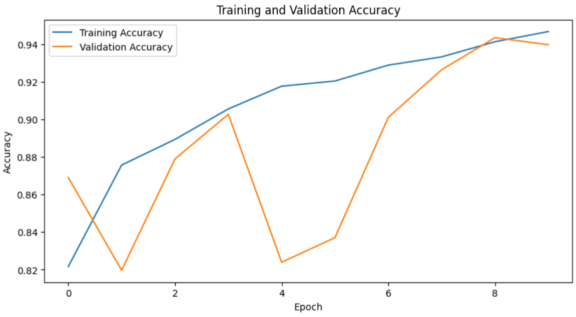
```
self.conv1 = nn.Sequential(
    nn.Conv2d(3, 64, kernel_size=3, stride=1, padding=1),
    nn.BatchNorm2d(64),
    nn.ReLU(),
    nn.Dropout(0.2))  # Dropout added
```

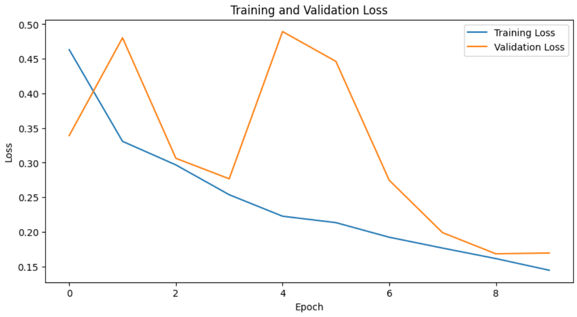
In this sample code, a dropout layer with a dropout probability of 0.2 is added after the ReLU activation function in the initial convolutional layer.

Dropout is a regularization technique used to prevent overfitting by randomly remove some network to zero during training. It's implemented using the `nn.Dropout` module.

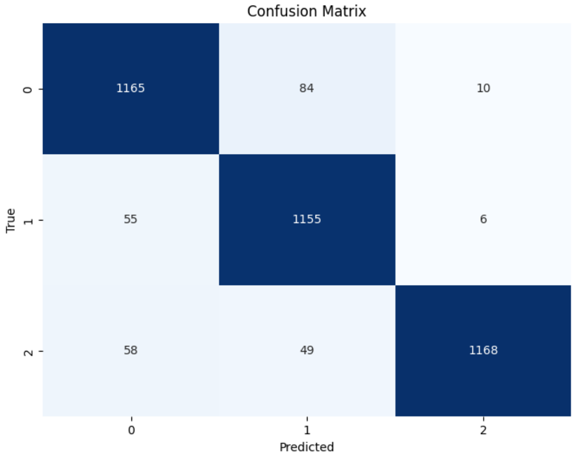
Performance

- The training loss steadily decreased from 0.4632 in the first epoch to 0.1445 in the final epoch. Correspondingly, the training accuracy improved from 0.8217 to 0.9467 over the 10 epochs.
- The validation loss fluctuated but generally decreased from 0.3391 to 0.1694 over the epochs. The validation accuracy improved from 0.8691 to 0.9397.





- The test accuracy achieved after training was 0.9301, with an F1 score of 0.9305, precision of 0.9322, and recall of 0.9301.
- From the confusion matrix we can see this applied dropout model performed comparatively well than VGG versions and the base Resnet versions on all the classes.



Early Stopping

Early stopping i used to prevent overfitting by stopping training when the performance on a validation dataset starts to degrade.

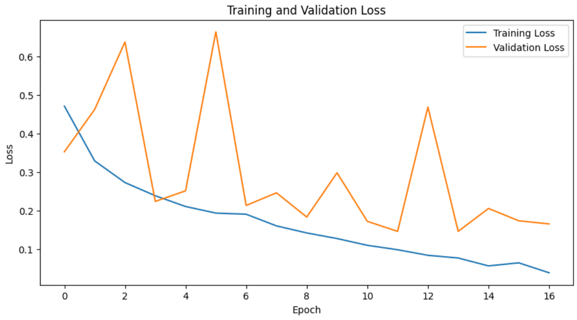
It's implemented within the `train_model` function using a patience parameter to control the number of epochs without improvement before stopping. Here the sample from the code showing the early stopping implementation

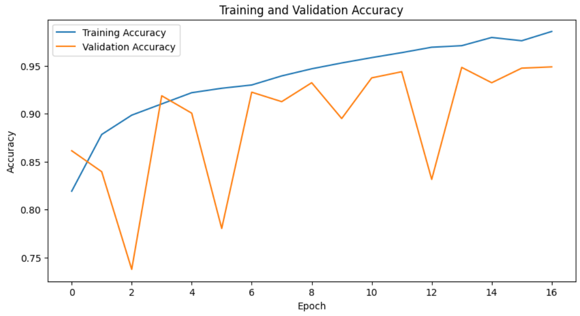
```
def train_model(model, train_loader, valid_loader, optimizer, criterion, num_epochs, patience=5):
    ...
    for epoch in range(num_epochs):
        ...
        # Check for early stopping
        if avg_valid_loss < best_valid_loss:
            best_valid_loss = avg_valid_loss
            early_stopping_counter = 0
        else:
            early_stopping_counter += 1

        if early_stopping_counter >= patience:
            print(f'Early stopping after epoch {epoch + 1} as validation loss did not improve for {patience} consecutive epochs.')
            break
    ...
```

Performance

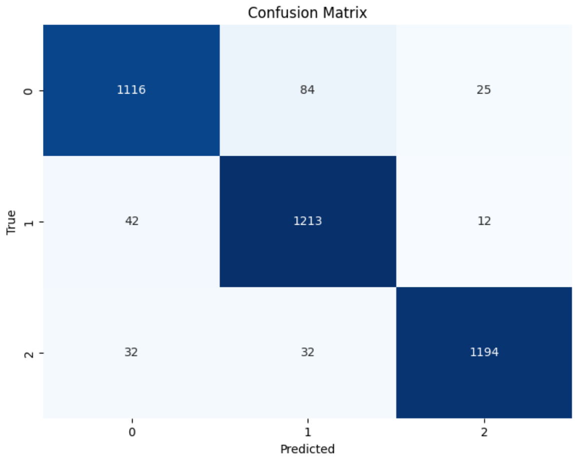
- The training loss decreased consistently from 0.4711 in the first epoch to 0.0651 in the last epoch. The training accuracy improved from 0.8192 to 0.9760 over the 5 epochs before early stopping.

- The validation loss fluctuated initially but generally decreased from 0.3528 to 0.1740 over the epochs. The validation accuracy improved from 0.8613 to 0.9475 before early stopping.





- The test accuracy achieved after training was 0.9368, with an F1 score of 0.9395, precision of 0.9401, and recall of 0.9395.
- From this C-matrix we can say that the mdoel performed well in the class 0 and class 2 and significantly well in the class 1.



Overview

Based on the provided information, here's a comparison table of the performance of the three models across various metrics:

Model	Train Accuracy (Low - High)	Validation Accuracy (Low - High)	Train Loss (Low - High)	Validation Loss (Low - High)	Test Accuracy	Precision	Recall
Base ResNet 18	0.8221 - 0.9534	0.8005 - 0.9264	0.1264 - 0.4640	0.2119 - 0.6188	0.9248	0.9259	0.9248
Dropout Applied	0.8217 - 0.9467	0.8197 - 0.9435	0.1445 - 0.4632	0.1682 - 0.4893	0.9301	0.9322	0.9301
Early Stopping Applied	0.8192 - 0.9760	0.8613 - 0.9475	0.0651 - 0.4711	0.1740 - 0.3528	0.9368	0.9401	0.9395

From the table, we can say that the model with early stopping applied performed the best overall, achieving the highest test accuracy, precision, recall, and F1 score while maintaining competitive performance in training and validation.

Paart III: Time-Series Forecasting using RNN

About the dataset

The "ElectricityLoadDiagrams20112014" dataset encompasses electricity consumption data collected from 370 points or clients over a period spanning from 2011 to 2014. Here are some key points about the dataset:

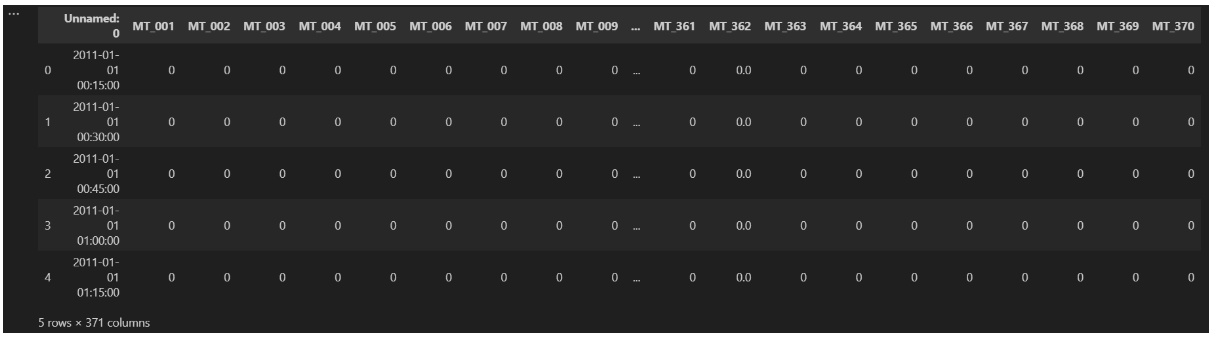
1. **Dataset Nature**: Time-series dataset capturing electricity consumption patterns over 15-minute intervals.

2. No missing values in the dataset, ensuring completeness for analysis.
3. Each instance represents a 15-minute interval, providing detailed insights into consumption trends.
4. The dataset comprises a total of 140,256 instances and 370 features, reflecting the consumption of individual clients.

🔗 Dataset Link

Understanding and Preparing the dataset

Since the dataset contains no missing values, there's no necessity to employ methods such as mean or mode to impute them. While we did make check for any missing or duplicate entries, it's insignificant considering the dataset's nature, which comprises 100,000+ rows and 370 columns representing clients. For a visual representation of the dataset, please refer to the image provided below.

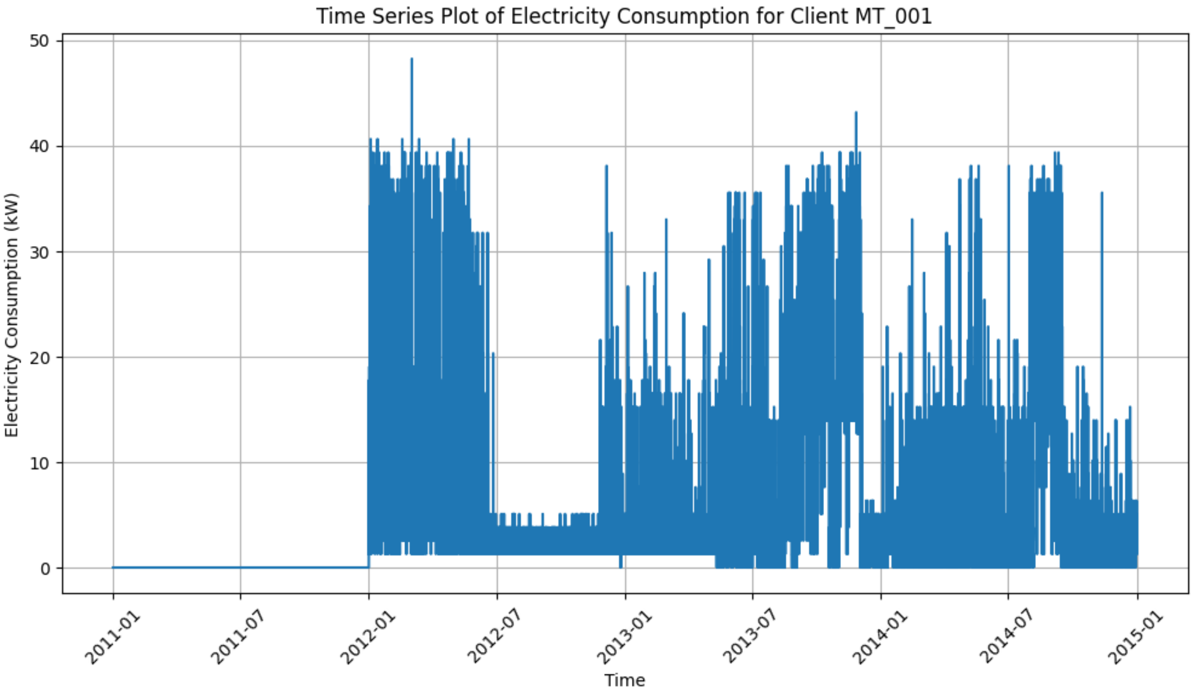


...	Unnamed: 0	MT_001	MT_002	MT_003	MT_004	MT_005	MT_006	MT_007	MT_008	MT_009	...	MT_361	MT_362	MT_363	MT_364	MT_365	MT_366	MT_367	MT_368	MT_369	MT_370
0	2011-01-01 00:15:00	0	0	0	0	0	0	0	0	0	...	0	0.0	0	0	0	0	0	0	0	0
1	2011-01-01 00:30:00	0	0	0	0	0	0	0	0	0	...	0	0.0	0	0	0	0	0	0	0	0
2	2011-01-01 00:45:00	0	0	0	0	0	0	0	0	0	...	0	0.0	0	0	0	0	0	0	0	0
3	2011-01-01 01:00:00	0	0	0	0	0	0	0	0	0	...	0	0.0	0	0	0	0	0	0	0	0
4	2011-01-01 01:15:00	0	0	0	0	0	0	0	0	0	...	0	0.0	0	0	0	0	0	0	0	0

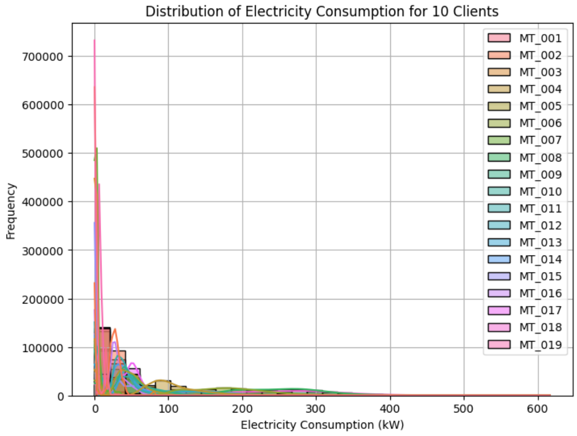
5 rows × 371 columns

Let's examine some trends in the dataset using some visualizations

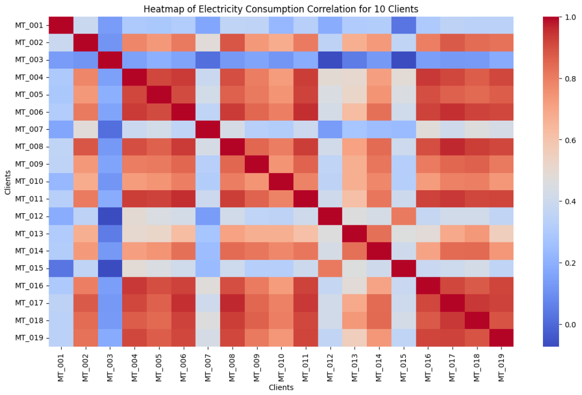
- Since the dataset follows a time series structure, it's essential to visualize the timely patterns of electricity consumption. Given the extensive nature of the dataset, examining all clients individually is impractical. Thus, I focused on analyzing the consumption trends of the first client. From the observed trends in the graph, it's apparent that this client exhibited high electricity consumption during 2012, followed by a decrease from July 2012 to 2013. Subsequently, the consumption appears to stabilize at its usual level.



- The second plot we focus on the electricity distribution over the ten clients, just to check how's the trend, and it shows that the first client is consuming the mosta nd rest are not even closer.



- Third we did a correlation of the 10 clients consumption, and the trend is they are alsomost closer to each other except for some like MT_07 and 12, 3,1.



Now let's process the data to get into the our RNN models,

- Fist we did Scaling, with the use of MinMaxScaler to scale the numerical data.

Then we performed Splitting the Data by determine the length of the scaled DataFrame and with the percentage of 70% of training, 10% goes to validation and 20% to test. Which gives the size splits of 98179 - train, 14025 - validation, and 28052 -test.

Building the RNN

The model we are using is a Simple Recurrent Neural Network (SimpleRNN) designed for time series forecasting tasks. Here are the specifications of the model:

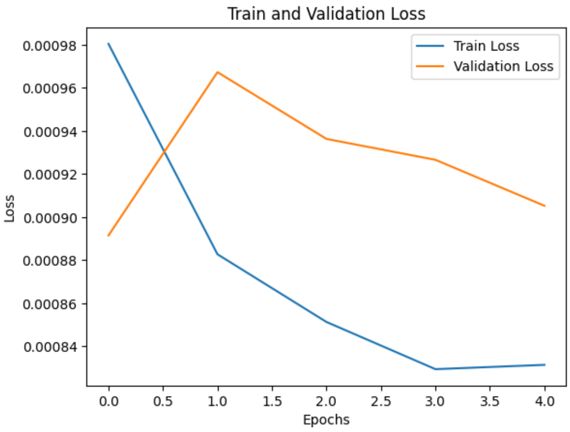
1. **Model Parameters**:
    - Input Size: 1 (for univariate time series data)
    - Hidden Size: 32 (number of hidden units in the RNN layers)
    - Output Size: 1 (since it's a univariate time series forecasting task)
    - Number of Layers: 3

SimpleRNN Class: This class defines the architecture of the SimpleRNN model. It inherits from `nn.Module` and consists of multiple RNN layers followed by a fully connected layer. The `__init__` method initializes the RNN layers with the specified input size, hidden size, and number of layers. The `forward` method executes the forward pass through the RNN layers, followed by the fully connected layer to produce the output.

```
SimpleRNN(
  (rnn_layers): ModuleList(
    (0): RNN(1, 32, batch_first=True)
    (1-2): 2 x RNN(32, 32, batch_first=True)
  )
  (fc): Linear(in_features=32, out_features=1, bias=True)
)
```

Performance

- The Simple RNN model was trained for 5 epochs, taking approximately 105 to 131 seconds per epoch.
- During training, the train loss consistently decreased, reaching 0.0008, while the validation loss remained close, fluctuating around 0.0009. The MSE test loss, measuring prediction accuracy on unseen data, was 0.0035, signifying the model's good performance on the test dataset.



Tuning the hyper parameters

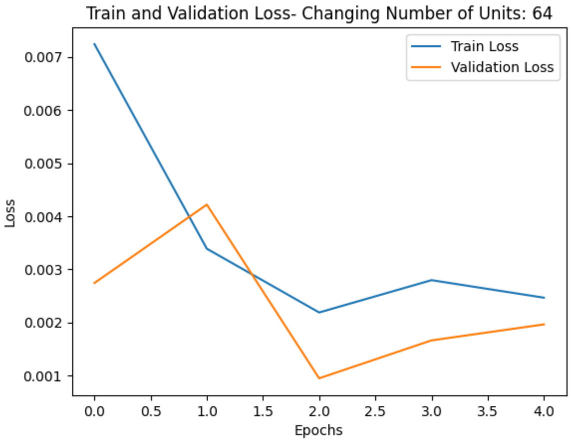
Change 1 - changed the number of units to 64, 128

Here we want to check the effect of altering the number of units in a Simple Recurrent Neural Network (RNN) model on performance metrics such as training loss, validation loss, and Mean Squared Error (MSE) test loss. The model is evaluated with unit sizes of 32, 64, and 128 to determine how variations in model complexity influence its ability to learn and generalize.

Model with 64 Units:

- **Epoch Duration:** Each epoch took approximately 130-170 seconds to complete.

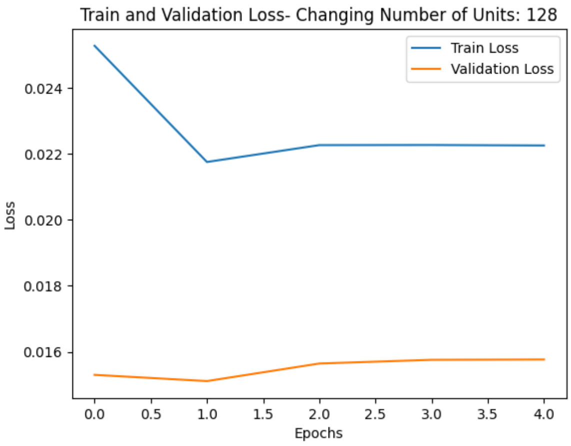
- **Training and Validation Loss:** The training loss steadily decreased across epochs, reaching 0.0025, while the validation loss fluctuated between 0.0009 and 0.0042.
- **Test Loss:** The MSE test loss for the 64-unit model was found to be 0.0057, indicating reasonable performance on unseen data.



Model with 128 Units:

- **Epoch Duration:** Epoch duration ranged from 158 to 248 seconds, reflecting increased computational complexity compared to the 64-unit model.

- **Training and Validation Loss:** The training loss gradually decreased to 0.0222, with the validation loss remaining relatively stable around 0.0151-0.0158.
- **Test Loss:** The MSE test loss for the 128-unit model increased to 0.0402, suggesting potential overfitting due to the higher model complexity.



Conclusion

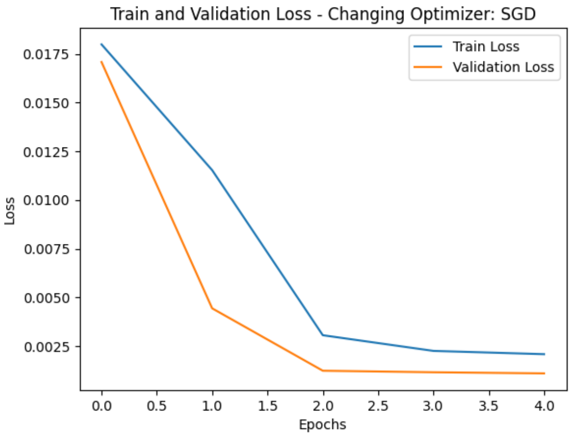
- **Effect of Unit Size:** Increasing the number of units from 32 to 64 led to improved performance, as evidenced by lower validation and test losses.
- However, further increasing the unit size to 128 resulted in diminished returns, with a higher test loss observed, indicating overfitting. **Diminishing Returns!!!**

Change 2 - changed the optuimizer, SGD and RMSProp

Model with SGD Optimizer:

- **Epoch Duration:** Each epoch took approximately 98-124 seconds to complete.

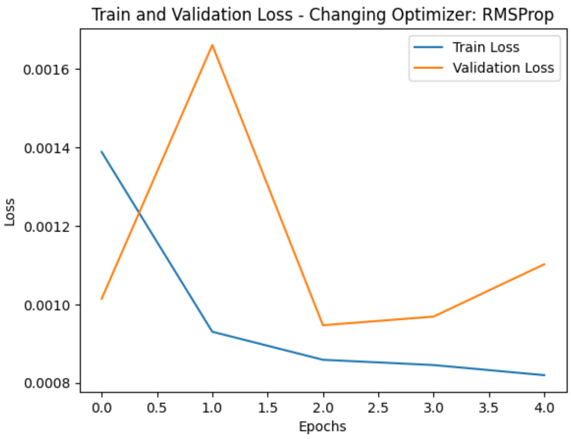
- **Training and Validation Loss:** The training loss decreased significantly from 0.0180 to 0.0021 over the 5 epochs, with validation loss following a similar trend, decreasing from 0.0171 to 0.0011.
- **Test Loss:** The MSE test loss for the SGD optimizer was found to be 0.0034, indicating good performance on unseen data.



Model with RMSProp Optimizer:

- **Epoch Duration:** Epoch duration ranged from 103 to 110 seconds, similar to the SGD optimizer.

- **Training and Validation Loss:** Both training and validation losses exhibited a decreasing trend across epochs, stabilizing at low values (around 0.0008-0.0017).
- **Test Loss:** The MSE test loss for the RMSProp optimizer was slightly higher at 0.0077 compared to SGD, suggesting slightly inferior performance on unseen data.



Conclusion:

- **Effect of Optimizers:** Both SGD and RMSProp optimizers effectively minimized the training and validation losses of the Simple RNN model.
- **Performance Comparison:** The SGD optimizer yielded slightly better test loss (0.0034) compared to RMSProp (0.0077), indicating superior generalization ability.
- **Considerations:** While SGD demonstrated better performance in this scenario, the choice of optimizer may vary depending on the dataset characteristics and specific modeling objectives.

Overview

Change	Epoch Duration (seconds)	Train Loss (Final)	Validation Loss (Final)	MSE Test Loss
Baseline	105-131	0.0008	0.0009	0.0035
64 Units	130-170	0.0025	0.0009-0.0042	0.0057
128 Units	158-248	0.0222	0.0151-0.0158	0.0402
SGD Optimizer	98-124	0.0021	0.0011	0.0034
RMSProp Optimizer	103-110	0.0008	0.0009-0.0017	0.0077

Based on the comparison, the model with the SGD optimizer performed the best. It had the lowest MSE test loss, meaning it made the most accurate predictions on new data. Both the training and validation losses were also kept low, indicating the model's ability to generalize well beyond the training data. Although the RMSProp optimizer did fairly well, its MSE test loss was slightly higher, suggesting it might not perform as reliably on new data compared to SGD. Therefore, for this task, using SGD seems to be the better choice as it offers better overall performance.

Part IV: Sentiment analysis using LSTM

About the data

The dataset consists of Twitter data collected in February 2015 regarding the sentiment towards major U.S. airlines. It includes information such as tweet IDs, airline sentiment (positive, neutral, negative), confidence scores for sentiment classification, reasons for negative sentiment, airline names, user names, retweet counts, tweet texts, tweet coordinates, tweet creation timestamps, tweet locations, and user timezones.

Some key points about the dataset:

- **Sentiment Analysis:** The primary task of the dataset is sentiment analysis, where tweets are categorized as positive, neutral, or negative sentiments towards the airlines.
- **Dataset Size:** The dataset contains 14,640 entries and 15 columns.

- **Missing Data:** Some columns have a significant amount of missing data, such as "negativereason," "negativereason_confidence," "airline_sentiment_gold," and "negativereason_gold."

Overall, this dataset is valuable for sentiment analysis tasks, allowing analysis of public opinion towards different airlines based on Twitter data. However, users should be cautious of missing data and ensure proper handling during analysis.

🔗 Dataset Link

Understanding and Preparing the dataset

Dataset Cleaning Report

Columns Removed:

1. **tweet_id:** This column uniquely identifies each tweet and does not provide any meaningful information for sentiment analysis.
2. **airline_sentiment_confidence:** The confidence score for the sentiment classification may not be relevant for our analysis.
3. **negativereason:** Although it provides insights into why a tweet may be negative, we will not consider this information initially for sentiment analysis.
4. **negativereason_confidence:** Similar to the sentiment confidence score, this column's confidence score for negative reasons is not necessary for our analysis.
5. **airline:** The airline name is not relevant for our initial sentiment analysis.
6. **airline_sentiment_gold:** This column contains gold-standard sentiment labels, which we will not use in our analysis.
7. **name:** Usernames do not contribute to sentiment analysis.
8. **negativereason_gold:** This column contains gold-standard negative reasons, which we will not use initially.
9. **retweet_count:** The number of retweets is not relevant for sentiment analysis.
10. **tweet_coord:** Tweet coordinates are not necessary for sentiment analysis.
11. **tweet_created:** Timestamps of tweet creation are not needed for sentiment analysis.
12. **tweet_location:** User-defined tweet locations are not relevant for our initial sentiment analysis.
13. **user_timezone:** User timezones do not contribute to sentiment analysis.

Remaining Columns:

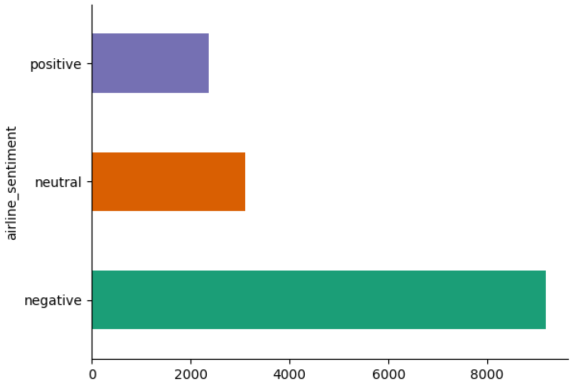
1. **text:** This column contains the tweet text, which is essential for sentiment analysis.
2. **airline_sentiment:** This column contains the sentiment labels (positive, neutral, negative) that we aim to predict.

Data Cleaning Process:

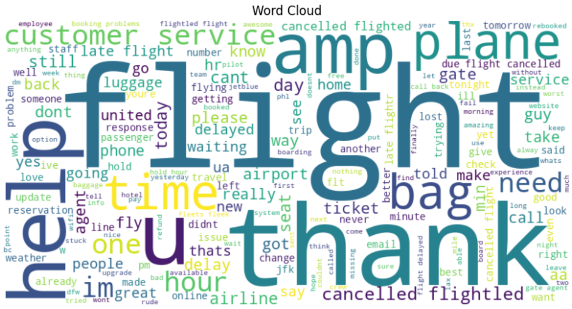
1. **Text Cleaning:** We performed text cleaning on the tweet text to prepare it for analysis.
    - **Lowercasing:** Converted all text to lowercase for consistency.
    - **Emoji Removal:** Removed emojis from the text as they do not contribute to sentiment analysis.
    - **Special Character Removal:** Removed special characters and Twitter handles from the text.
    - **Tokenization:** Tokenized the cleaned text into individual words.
    - **Stopword Removal:** Removed common English stopwords from the tokenized text to focus on meaningful words.
2. **New Column:** We created a new column named "cleanedtext" to store the cleaned tweet text.

Understanding through visualizations

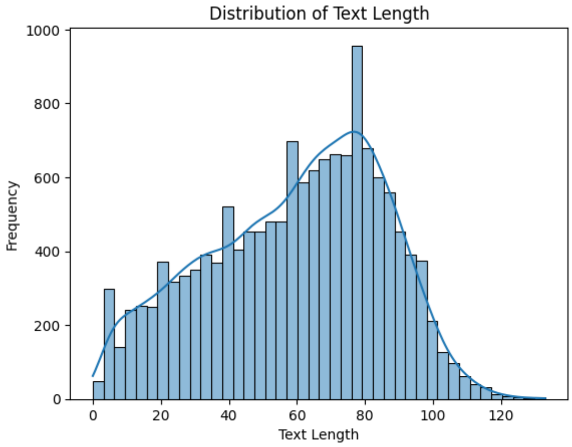
- The initial visualization displays a bar chart illustrating the distribution of sentiments. From the graph, it's evident that positive sentiments are the least frequent, while neutral sentiments have a moderate occurrence, and negative sentiments are the most prevalent.



- Next is the world cloud and the key terms include "Cancelled," highlighting its central importance, "Flight," emphasizing air travel, and "Customer service," suggesting a significant concern with airline service. Other notable words such as "tomorrow," "rebooked," and "late" further illuminate aspects of the customer's experience.



- The third graph shows the frequency plot of the text length for the each texts present in the dataset. From this graph we can deduce that the there are near to 100o texts with 80+ length.



Tokenization and Padding:

- Texts are tokenized using a tokenizer.
- The length of each tokenized text is calculated, with the maximum number of tokens found to be 18.
- To ensure uniformity, all sequences are padded to a maximum length of 28 tokens.

Label Encoding:

- The 'airline_sentiment' column is label encoded using sklearn's LabelEncoder.

Dataset Splitting:

- The data is split into training, validation, and test sets.
- Training set: 8198 samples, Validation set: 2050 samples, Test set: 4392 samples.
- Further split into tensors and organized into DataLoader objects for efficient training.

DataLoader Summary:

- Training DataLoader: 513 batches.
- Validation DataLoader: 129 batches.
- Test DataLoader: 275 batches.

BASE LSTM

The base model utilized for sentiment analysis is an LSTM neural network. This model architecture consists of three LSTM layers followed by a fully connected layer. The input to the model is tokenized text, which is embedded into dense vectors using an embedding layer. The LSTM layers capture sequential dependencies in the text data, and the final output is passed through a softmax layer to obtain probabilities for each sentiment class.

Model Hyperparameters:

- Vocabulary Size: 1000
- Embedding Dimension: 128
- Hidden Dimension (LSTM layers): 64
- Output Dimension: 3 (corresponding to sentiment classes)

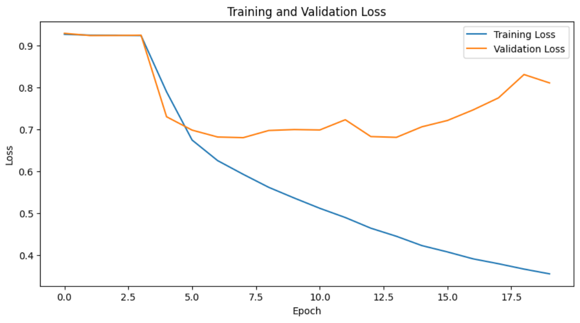
Training Procedure:

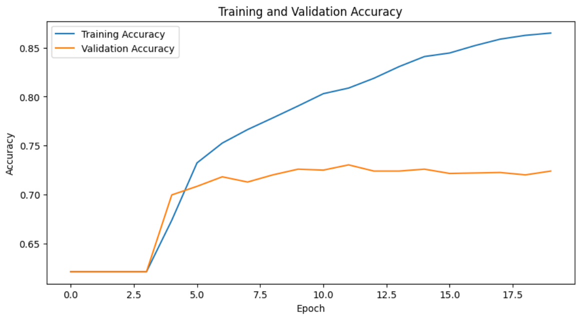
- Loss Function: Cross Entropy Loss
- Optimizer: Adam optimizer with a learning rate of 0.001
- Training Epochs: 20

Model Performance:

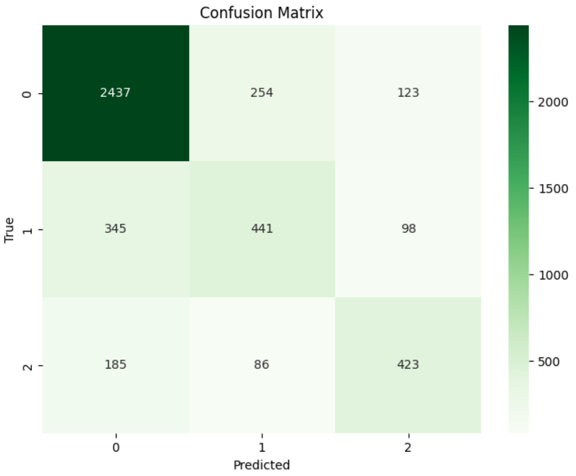
- The base model starts with a train accuracy of around 62% and remains relatively stable throughout training.
- There is a gradual increase in train accuracy over epochs, reaching approximately 86.50% by the end of training.
- Test Accuracy: 75.16%
- F1 Score: 74.67%
- Precision: 74.37%
- Recall: 75.16%
- Loss: 0.7702

Form the graphs we can say that the difference in accuracy of train and the validation is quite high same trend follows for the losses, but despite the difference the graphs are quite smooth





- **From the confusion matrix we can say that** model performed well for Class 0, with 2437 correct predictions the model also performed well for Class 1, with 441 correct predictions and 423 for the class 2, this might be due to the huge amount of data containing on negative sentiments.

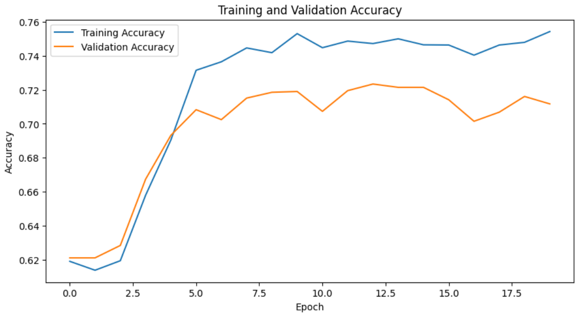


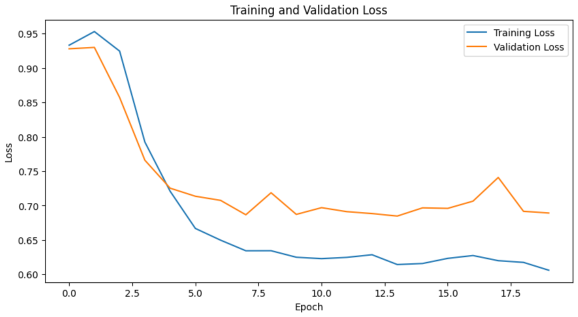
After Model Tuning and Performance:

Base Model with Tuning:

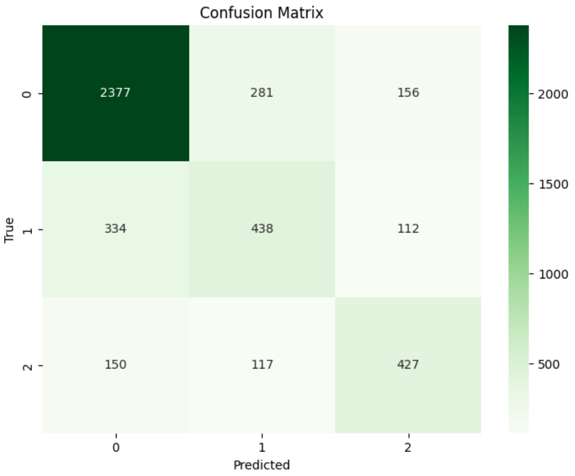
- **Architecture Changes:** The hidden dimension of the LSTM layers was increased to 128 units.
- **Optimization:** The learning rate of the Adam optimizer was increased to 0.01.
- **Performance:** Despite the changes, the test accuracy slightly decreased to 73.82%, with an F1 score of 0.7364 and a loss of 0.6590.
    - Train accuracy starts at a similar level as the base model, around 62%.
    - The trend shows slight fluctuations during training, with a peak around 75.43% before stabilizing around 74-75%.

The graphs also show similar trend quite inconsistency in the accuracy and also the loss, the base model performed well.





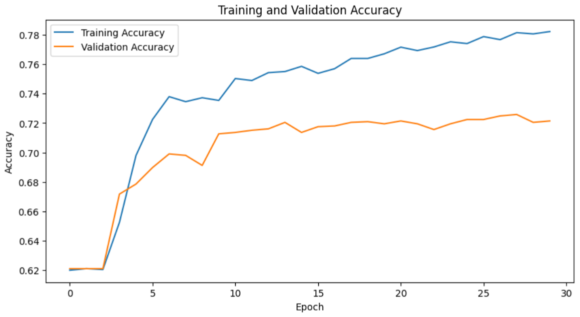
Coming to confusion matrix compared to the base model the model is not performing well as you can see the class 0 correctly predicted is 2377 though one thing to notice is the model is good at predicting class 2 compared to the base one though the difference is minute its just 5 more predicted correctly.

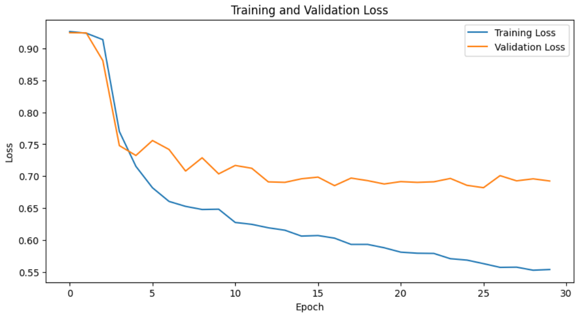


Model with Learning Rate and Optimizer Change:

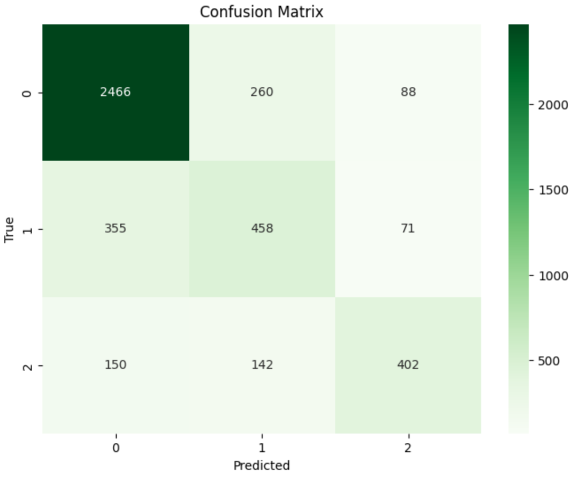
- **Optimization Changes:** The optimizer was changed to Adagrad with a learning rate of 0.01.
- **Performance:** The test accuracy increased to 75.73%, with an F1 score of 0.7532 and a loss of 0.6358.
  - Train accuracy begins at approximately 62% and exhibits a steady increase during the initial epochs.
  - There are fluctuations in accuracy during training, but it maintains an upward trend, reaching around 78.23% by the end of training.

Despite the variations observed in the graph, the margin between training accuracy and validation accuracy is minimal, indicating the model's robustness. This consistency is also reflected in the loss trend.





The performance portrayed in the confusion matrix demonstrates that this model excels in detecting all classes compared to others, signifying its superiority.



- Among the models tested, the base model with the tuning of the hidden dimension and learning rate achieved the highest accuracy of 75.73%.
- Increasing the learning rate and changing the optimizer resulted in a slight improvement in accuracy compared to the base model.
- However, tuning the hidden dimension along with learning rate did not significantly improve the model's performance. Therefore, the base model with adjustments to hyperparameters seems to perform the best for this sentiment analysis task.

Overview

Model	Changes	Test Accuracy	Train Accuracy	Validation Accuracy	Test Loss	Train Loss	Validatio
Base Model	-	0.7516	0.8650	0.8650	0.7702	0.3546	0.8108
CHANGE 1	Hidden units increased to 128, LR increased	0.7382	0.7543	0.7543	0.6590	0.6060	0.6892
CHANGE 2	Learning rate increased to 0.01, Optimizer changed to Adagrad	0.7573	0.7823	0.7823	0.6358	0.5539	0.6924

Overall, the **Model with LR Increase** performs the best among the three models, with improved generalization performance and the highest test accuracy. Adjusting hyperparameters, such as the learning rate and optimizer, proved effective in enhancing the model's performance.

BIDIRECTIONAL - LSTM

Model Architecture:

- **Input Embedding**: The model starts with an embedding layer to convert input tokens into dense vectors.
- **Bidirectional LSTM Layers**: It consists of three bidirectional LSTM layers. Bidirectional LSTMs allow the model to capture information from both past and future contexts, enhancing its understanding of the input sequence.
- **Dropout**: Dropout is applied to prevent overfitting by randomly setting a fraction of input units to zero during training.
- **Fully Connected Layers**: Two fully connected layers with ReLU activation functions are used for classification.
- **Output Layer**: The final output layer predicts the class labels.

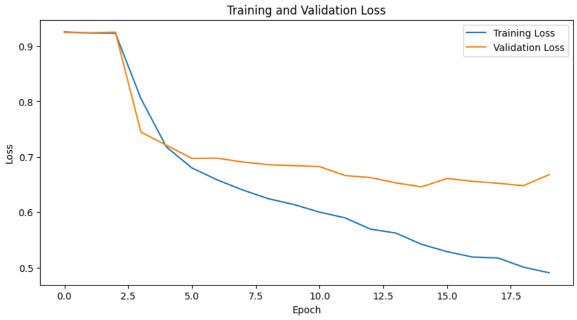
Hyperparameters:

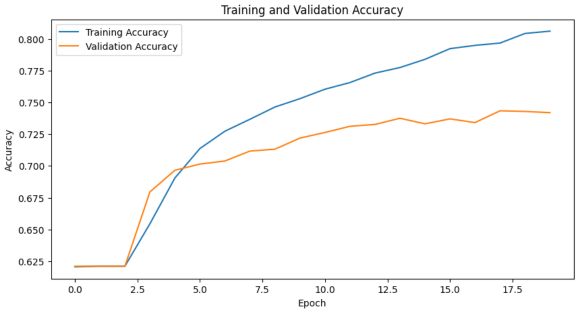
- **Vocabulary Size**: 1000
- **Embedding Dimension**: 128
- **Hidden Dimension**: 64
- **Output Dimension**: 3 (assuming it's a multi-class classification task)
- **Dropout Rate**: 0.2
- **Optimizer**: Adagrad with learning rate 0.01

Performance:

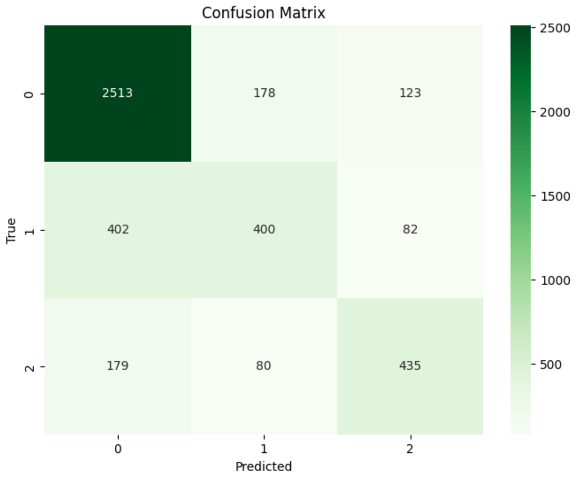
- **Train Accuracy**: Started around 62% and increased steadily over epochs, reaching 80.62%.
- **Validation Accuracy**: Similar trend to train accuracy, reaching around 74.20%.
- **Test Accuracy**: Achieved 76.23%.

The two line graphs on the below show the training and validation loss and accuracy of a deep learning model over several epochs. The training loss and accuracy curves tend to decrease and increase, respectively, as the number of epochs increases, suggesting that the model is learning.



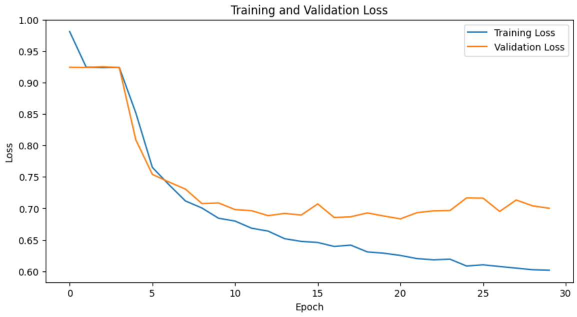


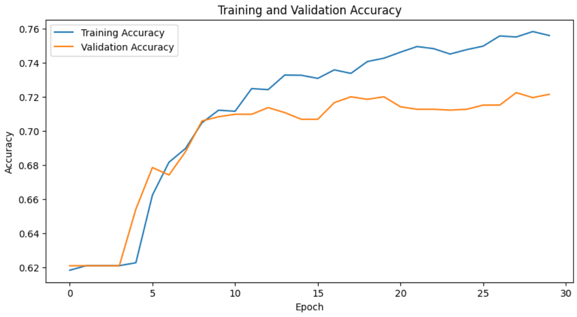
The confusion matrix results demonstrate a notable enhancement in accuracy compared to the earlier base LSM results. In the previous LSM models, the correct detection of class 0 was observed in the range of 2400s, whereas in this updated version, it extends beyond the 2500s.



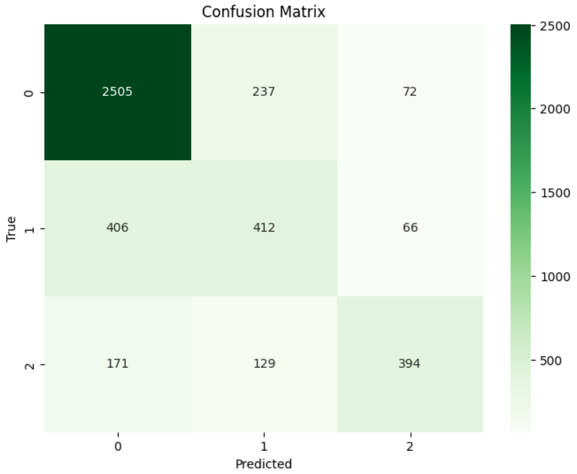
Change 1:

- **Model Architecture**:
  - Increased dropout rate to 0.5 after the first bidirectional LSTM layer.
- **Optimizer**: Changed to Adagrad with a higher learning rate of 0.1.
- **Performance**:
  - Train accuracy started at 61.83% but didn't improve much.
  - Validation accuracy remained stable around 72%.
  - Test accuracy dropped slightly to 75.39%.





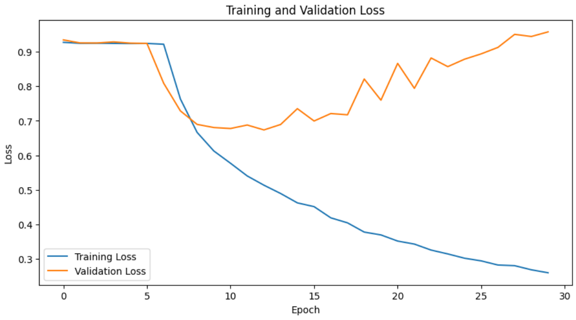
The predictions in the class 0 decreased when compared to the base bi-di lstm and though this version performed well in class 1 and not good in class 2

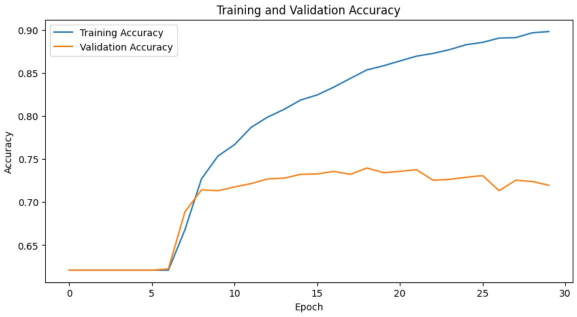


Change 2:

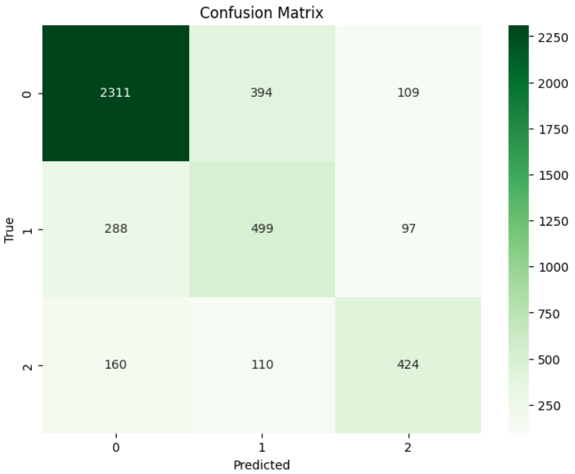
- **Model Architecture**: Reverted dropout rate to 0.2 and changed the optimizer to Adam with a lower learning rate of 0.001.
- **Performance**:
  - Train accuracy started at 61.92% and improved over epochs, reaching 85.92%.
  - Validation accuracy showed similar improvement, reaching 71.56%.
  - Test accuracy decreased to 73.50%.

The training loss decreases gradually over time, indicating the model's learning progress, but fluctuations in later epochs hint at potential overfitting. Validation loss also decreases but shows more variability, reflecting the model's adaptation to unseen data. While training accuracy steadily rises to about 95%, validation accuracy fluctuates more and doesn't reach as high, indicating the model's struggle to generalize well and avoid overfitting.





The outcomes distinctly indicate that class 0 is not accurately detected, while class 1 is predicted at a considerably high rate.



Overview

Model	Changes	Train Accuracy	Validation Accuracy	Test Accuracy	Train Loss	Validation Loss	Test Los
BL_Model	Base Model	80.62%	74.20%	76.23%	0.4907	0.6680	0.6340
BL_Model_D	Increased Dropout	75.59%	72.15%	75.39%	0.6016	0.7001	0.6464
BL_Model	Changed Optimizer to Adam	85.92%	71.56%	73.50%	0.3864	0.8003	0.7524

From the provided table, it's evident that the model with the base architecture (BL_Model) but with the optimizer changed to Adam achieves the highest train accuracy.

PART V: CNN & LSTM Theoretical Part

Part V.I: CNN

In a convolutional neural network (CNN), suppose you're working on a computer vision project that involves classifying images of different emotions expressed by human faces. The network is designed for a multi-class classification task with 5 output classes, where each class represents a different emotion category: happiness, sadness, anger, surprise, and neutral.

CNN Architecture:

- The input consists of 32×28 RGB images. The first layer of the CNN is a convolutional layer with 10 filters, each having a size of 5×5, zero padding, and a stride of 1.
- The output values of the network represent the predicted probabilities for each emotion category and the predicted probabilities for each class are non-negative and sum up to 1.

Q & A

1. **What is the output size after the first layer?**

Answer

We got the input of 32 × 28, 3 channel image.

- So, For the Width we got 32, so when it goes through the conv layer, we need to calculate what width will be getting out

  Output size = (Input size - Filter size + 2 * Padding) / Stride + 1

  So, (32 - 5 + 2 * 0)/1 + 1 = 28
- Similarly we have the height that's going in is 28, now similarly we need to find the output size.

  So, (28 - 5 +2 *0)/1 + 1 = 24
- Depth of the image is 10

Therefore the output size will be 28 × 24 × 10

---

2. **How many parameters are there in first layer?**

Answer

Total parameters = [Filter size x # of channels + bias] x # of filters

- So we got filter size = 5 × 5
- As RGB images, we got channels = 3
- Bias = 1
- Filters = 10
- **Total parameters = [(5×5) x (3) + (1)] x 10 = 760**

3. **What would be the output size if a padding of 1 was used instead of zero padding?**

Answer

Output size = (Input size - Filter size + 2 * Padding) / Stride + 1

So for width, we get the output of

(32 - 5 + 2 × 1) / 1 + 1 = 30

And for the height

(28 - 5 + 2 × 1) / 1 + 1 = 26

Depth of the image is 10

Therefore the output size will be 30 × 26 × 10

4. **What would be the number of parameters if the input images were grayscale instead of RGB?**

ANswer

- If the input images were grayscale, there would be only 1 channel.

Total parameters = [Filter size x # of channels + bias] x # of filters

- So we got filter size = 5 × 5
- As RGB images, we got channels = 1
- Bias = 1
- Filters = 10
- **Total parameters = [(5×5) x (1) + (1)] x 10 = 260**

5. **Considering the above specific task and the requirement of probabilistic outputs, which activation function would be most suitable for the output layer in this scenario?**

ANswer

Since the output values represent predicted probabilities for each class as this is a multi class classification and need to be non-negative and sum up to 1, the most suitable activation function for the output layer in this scenario would be the **softmax** activation function.

6. **Prove that the activation function used here is invariant to constant shifts in the input values, meaning that adding a constant value to all the input values will not change the resulting probabilities.**

Answer

Proof

Let's say zi as the input to softmax activation for class i

$$\text{softmax}(z) = \frac{e^{z_i}}{\sum_{j=1}^{n} e^{z_j}}$$

now, if we add constant c to all input values zi, the softmax output becomes

$$\text{softmax}(z+c) = \frac{e^{z_i + c}}{\sum_{j=1}^{n} e^{z_j + c}}$$

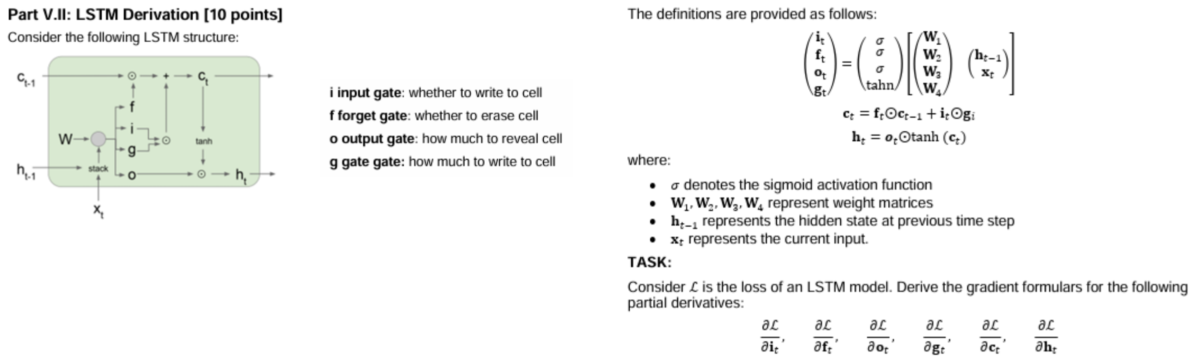
$$\Rightarrow \frac{e^{z_i} \cdot e^{c}}{\sum_{j=1}^{n} e^{z_j} \cdot e^{c}} \Rightarrow \frac{e^{c} \cdot \sum_{j=1}^{n} e^{z_j}}{\sum_{j=1}^{n} e^{z_j}} \Rightarrow e^{c} \frac{\sum_{j=1}^{n} \boxed{e^{z_j}}}{\sum_{j=1}^{n} \boxed{e^{z_j}}}$$

$$\Rightarrow \frac{\sum_{j=1}^{n}}{e^{-c} \cdot \sum_{j}^{n}} \Rightarrow \frac{\sum_{j=1}^{n} e^{z_j}}{\sum_{j=1}^{n} e^{z_j}} \Rightarrow \frac{\sum_{j=1}^{n} e^{z_j}}{\sum_{j=1}^{n} e^{z_j}}$$

∴ adding a constant c to all input values zi does not change the output of softmax probabilities.

" Hence Softmax is invarient to constant shifts in the input values. "

Part V.II: LSTM Derivation



Part V.II: LSTM Derivation [10 points]

Consider the following LSTM structure:

i input gate: whether to write to cell

f forget gate: whether to erase cell

o output gate: how much to reveal cell

g gate gate: how much to write to cell

The definitions are provided as follows:

$$\begin{pmatrix} \mathbf{i}_t \\ \mathbf{f}_t \\ \mathbf{o}_t \\ \mathbf{g}_t \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \text{tahn} \end{pmatrix} \left[\begin{pmatrix} \mathbf{W}_1 \\ \mathbf{W}_2 \\ \mathbf{W}_3 \\ \mathbf{W}_4 \end{pmatrix} \quad \begin{pmatrix} \mathbf{h}_{t-1} \\ \mathbf{x}_t \end{pmatrix} \right]$$

$$\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \mathbf{g}_i$$

$$\mathbf{h}_t = \boldsymbol{o}_t \odot \tanh\left(\mathbf{c}_t\right)$$

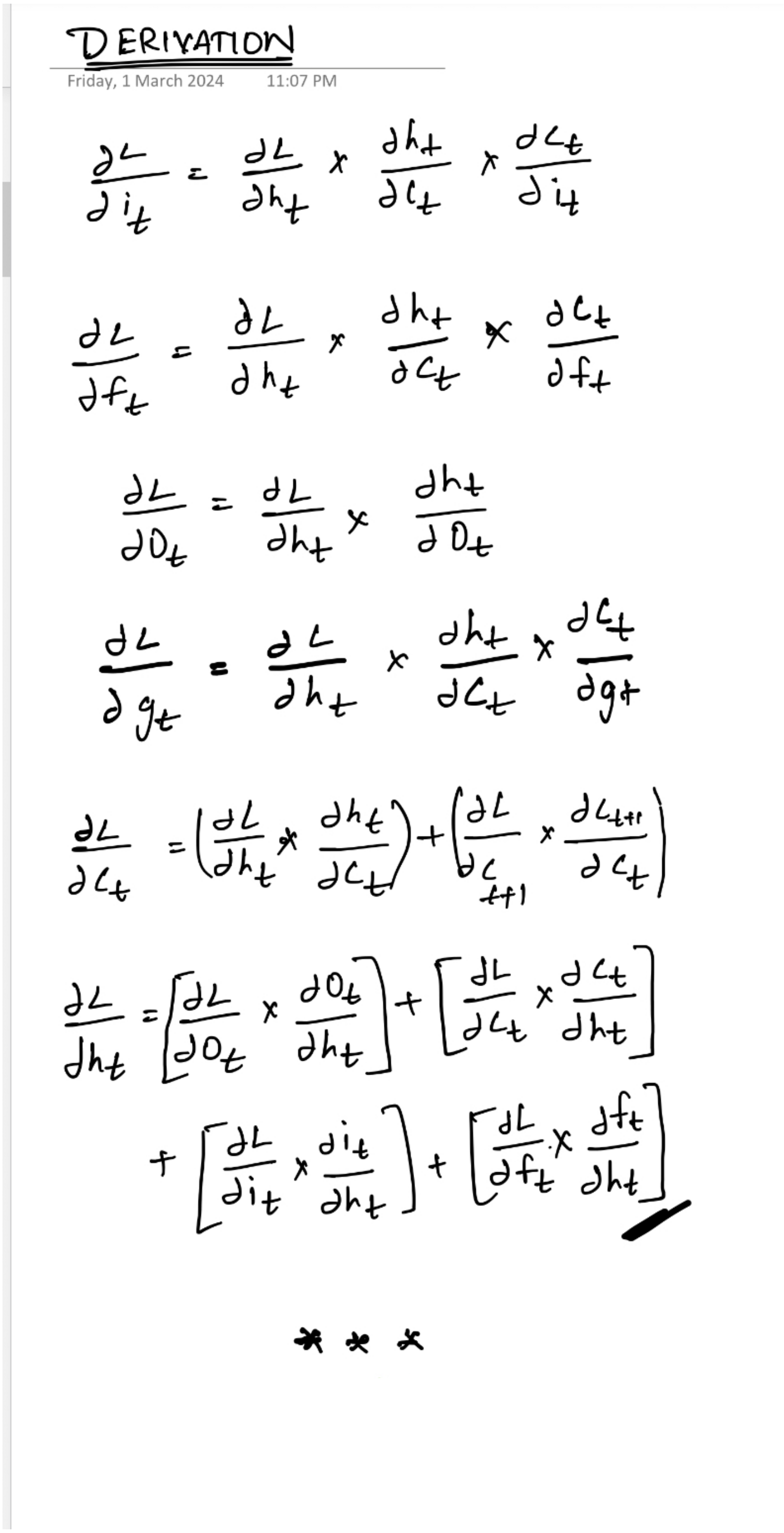
where:

- σ denotes the sigmoid activation function
- $\mathbf{W}_1, \mathbf{W}_2, \mathbf{W}_3, \mathbf{W}_4$ represent weight matrices
- $\mathbf{h}_{t-1}$ represents the hidden state at previous time step
- $\mathbf{x}_t$ represents the current input.

TASK:

Consider $\mathcal{L}$ is the loss of an LSTM model. Derive the gradient formulars for the following partial derivatives:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{i}_t}, \quad \frac{\partial \mathcal{L}}{\partial \mathbf{f}_t}, \quad \frac{\partial \mathcal{L}}{\partial \mathbf{o}_t}, \quad \frac{\partial \mathcal{L}}{\partial \mathbf{g}_t}, \quad \frac{\partial \mathcal{L}}{\partial \mathbf{c}_t}, \quad \frac{\partial \mathcal{L}}{\partial \mathbf{h}_t}$$



DERIVATION

Friday, 1 March 2024 11:07 PM

$$\frac{\partial L}{\partial i_t} = \frac{\partial L}{\partial h_t} \times \frac{\partial h_t}{\partial c_t} \times \frac{\partial c_t}{\partial i_t}$$

$$\frac{\partial L}{\partial f_t} = \frac{\partial L}{\partial h_t} \times \frac{\partial h_t}{\partial c_t} \times \frac{\partial c_t}{\partial f_t}$$

$$\frac{\partial L}{\partial O_t} = \frac{\partial L}{\partial h_t} \times \frac{\partial h_t}{\partial O_t}$$

$$\frac{\partial L}{\partial g_t} = \frac{\partial L}{\partial h_t} \times \frac{\partial h_t}{\partial c_t} \times \frac{\partial c_t}{\partial g_t}$$

$$\frac{\partial L}{\partial c_t} = \left(\frac{\partial L}{\partial h_t} \times \frac{\partial h_t}{\partial c_t}\right) + \left(\frac{\partial L}{\partial c_{t+1}} \times \frac{\partial c_{t+1}}{\partial c_t}\right)$$

$$\frac{\partial L}{\partial h_t} = \left[\frac{\partial L}{\partial O_t} \times \frac{\partial O_t}{\partial h_t}\right] + \left[\frac{\partial L}{\partial c_t} \times \frac{\partial c_t}{\partial h_t}\right] + \left[\frac{\partial L}{\partial i_t} \times \frac{\partial i_t}{\partial h_t}\right] + \left[\frac{\partial L}{\partial f_t} \times \frac{\partial f_t}{\partial h_t}\right]$$

* * *

References

- L1/L2 regularization in PyTorch. (n.d.). Stack Overflow. Retrieved February 21, 2024, from https://stackoverflow.com/questions/42704283/l1-l2-regularization-in-pytorch
- Nouman. (2022, June 21). Writing ResNet from scratch in PyTorch. Paperspace Blog. https://blog.paperspace.com/writing-resnet-from-scratch-in-pytorch/
- vision: Datasets, Transforms and Models specific to Computer Vision. (n.d.).
- Yu, H. (2022, October 27). A detailed introduction to ResNet and its implementation in PyTorch. Medium. https://medium.com/@freshtechyy/a-detailed-introduction-to-resnet-and-its-implementation-in-pytorch-744b13c8074a
- Regularization in Deep Learning. https://www.deeplearning.ai/ai-notes/regularization/
- Dropout: A Simple Way to Prevent Neural Networks from Overfitting. https://www.cs.toronto.edu/~hinton/absps/JMLRdropout.pdf
- Data Augmentation. https://www.tensorflow.org/tutorials/images/data_augmentation
- Early Stopping — but when? https://page.mi.fu-berlin.de/prechelt/Biblio/stop_tricks1997.pdf
- Hebbar, N. (Year). Time-Series-Forecasting-LSTM [GitHub repository]. Retrieved from https://github.com/nachi-hebbar/Time-Series-Forecasting-LSTM
- Pratama, D. (Year). Twitter Sentiment Analysis with Bidirectional-LSTM [Kaggle].