Building Neural Networks and Convolutional Neural Networks

Hrushikesh Poola

June 14, 2023

The primary goal of this report is to learn and understand about procedure to build neural networks and Convolutional neural networks in python using pytorch, a widely used deep learning framework. $Part-I$ and $Part-II$ of the report covers the implementation of a simple neural network with fully connected layers for a given dataset with 7 features and a target value. Target has only two values, so it is a binary classification problem. $Part-III$ of the report is to build a customdataset for the dataset given with three classses and need to classify the data by creating a custom Neural network. Part-IV of the report is to implement the custom convolution neural network created in Part-III on SVHN dataset with 10 classes(multiclass classification). All Parts of the report networks are trained using PyTorch, and their accuracy is evaluated on test sets. The report provides a practical introduction to defining network architectures, implementing the training process, and exploring different hyperparameters and architectures to improve model accuracy.

1 Part-I : Building Neural Network

1.1 About the dataset

The dataset contains 7 features [f1,f2,....f7] with each datapoint being a float value and a target value for each datapoint with either of 0 or 1 as its value.

1.2 Pre-processing

- There are some datapoints with corrupted data like character values instead of float values of the data. All such datapoints are dropped from the dataset.Once the dataset is cleaned and only contains float values, all the input datapoints are normalized to a space where data is around mean and unit standard deviation. The normalization is achieved by using StandardScaler. Features of each datapoints are skewed, with one feature having high magnitude compared to other features, so data is normalized for each feature to get each value of the feature in input dataset to a float value between 0 and 1. Normalizing the data also supress the high magnitude multiplications while calculating hidden layer values as well as decrease the influence of predicted class based on high magnitudes of features.
- The dataset is divided into three parts for training, validation and testing. Out of 100% of the dataset, 20% of it is kept aside for evaluation/testing. Rest of 80% is divided into 80% training and 20% for validation datasets respectively. Test data is used after training the model to understand about the accuracy of the model for unseen data.

1.3 Size of the dataset

The actual dataset provided contains 766 datapoints with 7 features and a target value. But after removing corrupt datapoints the input datasize reduced to 760 datapoints.

1.4 Visualization of the dataset

created a plot between each feature values with its corresponding index in dataset. This will help us understand the skewness in the magnitude of each feature across all the input data(this lead us to normalize the data). Figure 1 depicts about the plot between indices and all the features of the input dataset, whereas in Figure 2 we split the plot based on each features vs indices to understand the statistics of the data. Figure 3 hightlights that data contains more input datapoints with target as 0 compared to target as 1.

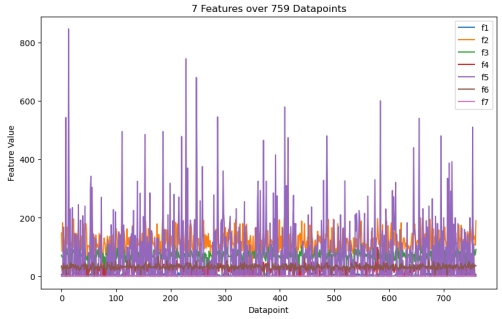


Figure 1: Features Vs Indices

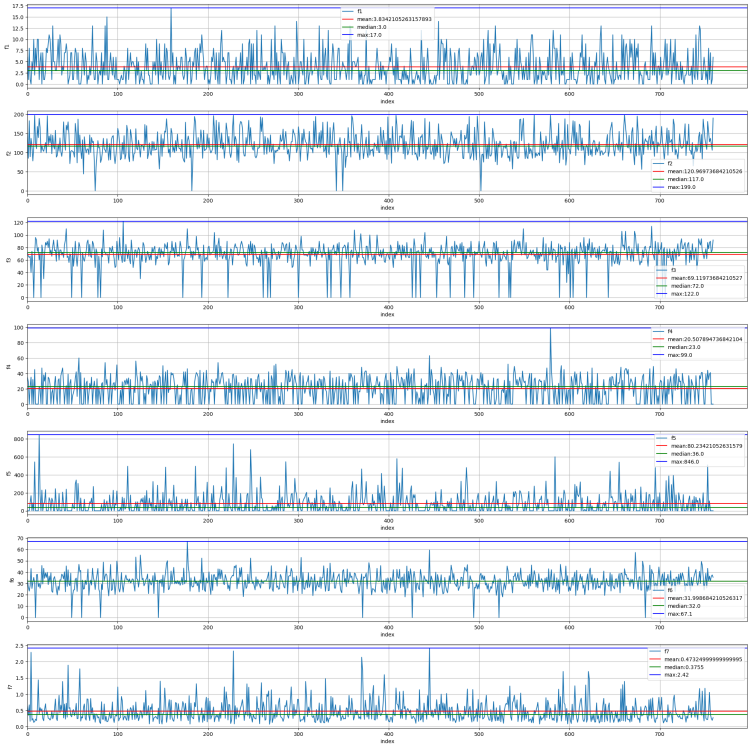


Figure 2: Each Feature Vs Indices

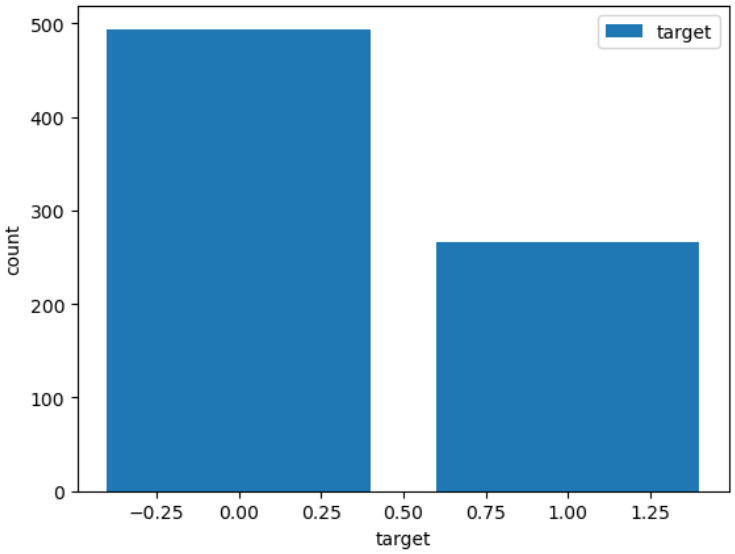


Figure 3: Frequency Vs Target Label

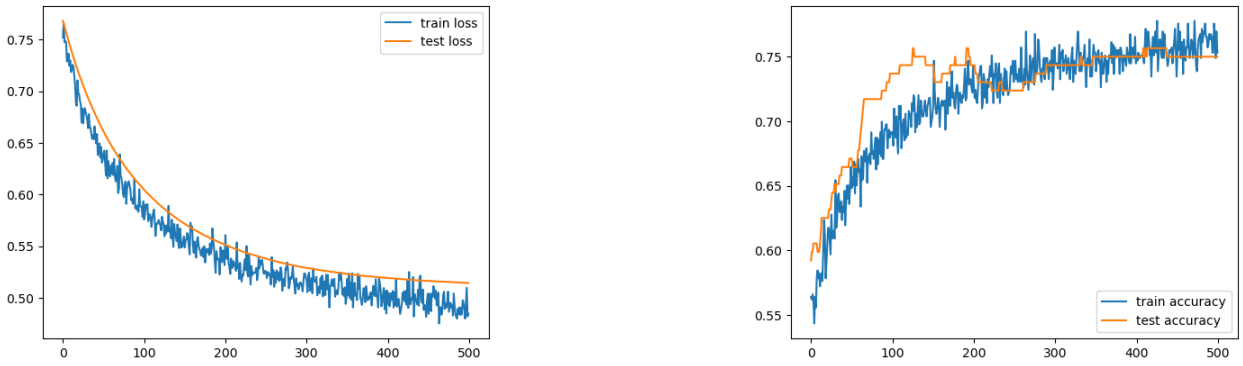
1.5 Structure of Neural Network

```
NeuralNetwork(
  (layers):
  Sequential(
    (0): Linear(in_features=7, out_features=64, bias=True)
    (1): LeakyReLU(negative_slope=0.01)
    (2): Dropout(p=0.0, inplace=False)
    (3): Linear(in_features=64, out_features=64, bias=True)
    (4): BatchNorm1d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (5): Dropout(p=0.0, inplace=False)
    (6): Linear(in_features=64, out_features=64, bias=True)
    (7): LeakyReLU(negative_slope=0.01)
    (8): Dropout(p=0.0, inplace=False)
    (9): Linear(in_features=64, out_features=1, bias=True)
  )
)
```

The above mentioned structure is used as base model to classify the dataset.

1.6 Train - Test graphs

Loss graph (Figure 4) show the above mentioned model's ability to correctly classify the validation data as the number of training epochs increases. As the epochs increases the training and validation loss decreases which depicts that model is learning and able to classify validation data correctly. From the graph, the generalization gap is also less indicating that model is not memorizing the data (not over-fitting). so this indicates that the model works well on unseen data or test data.



(a) Loss graph

(b) Accuracy graph

Figure 4: Loss and Accuracy Plot For base model

Accuracy graph (Figure 4) show the model's ability to correctly classify data as the number of training epochs increases.

Confusion matrix for the dataset classified by the base model

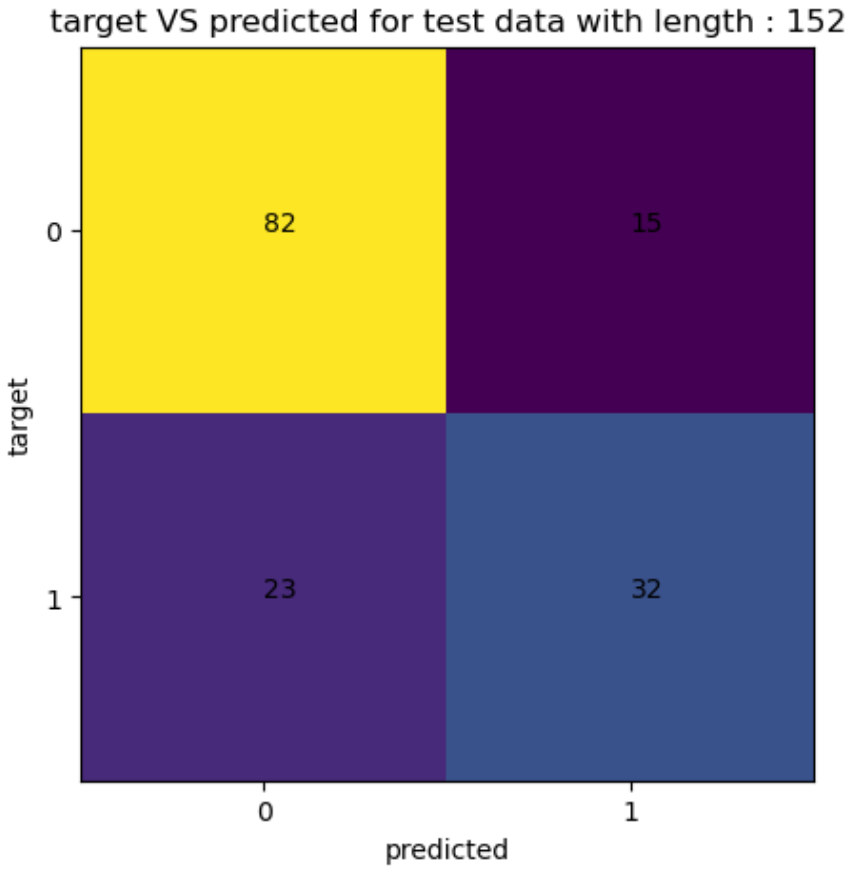


Figure 5: Confusion matrix for base model

1.7 Note : How to run Part-I and Part-II

I have created a custom model with configuration as input to build the model. so after saving the model and reloading the .pt file, I have specified how to load the .pt file in the submission .ipynb file. The idea of creating custom model with configuration is to tune the model with different hidden layers based on the configuration, rather than creating model for each setup in the part-II and part-I.

```
class NeuralNetwork(nn.Module):
    def addLayer(self,layers,config):
        if config['type'] == "Linear":
            linear = nn.Linear(config['input_size'], config['output_size'])
            nn.init.xavier_uniform_(linear.weight)
            nn.init.zeros_(linear.bias)
            layers.append(linear)

        elif config['type'] == "ReLU":
            layers.append(nn.ReLU())
        elif config['type'] == "tanh":
            layers.append(nn.Tanh())

        elif config['type'] == "LReLU":
            layers.append(nn.LeakyReLU())
        elif config['type'] == "Dropout":
            layers.append(nn.Dropout(config['p']))
        elif config['type'] == "Norm":
            layers.append(nn.BatchNorm1d(config['size']))
        elif config['type'] == "sigmoid":
            layers.append(nn.Sigmoid())

    def __init__(self, model_config):
        super(NeuralNetwork, self).__init__()

        layers = []
        for layerConfig in model_config['layers']:
            self.addLayer(layers, layerConfig)

        self.layers = nn.Sequential(*layers)

    def forward(self,x):
        x = self.layers(x)
        x = torch.sigmoid(x)

        return x
```

Here, addLayer method is used to build each hidden layer based on the configuration in init method to layers list. addLayer method contains all the different types of pre-defined layers like nn.Linear, nn.ReLU, nn.Tanh, nn.Dropout() and nn.Sigmoid() to load layers based on the configuration file.

- stored the state of the model in .h5 file
- steps to run the model is specified in the notebook file

Part - I, II : Please follow the steps in the notebook file with cells after Markdown :
'Note : Please test my model using the following cell to create a model and load the weights'

- In Notebook file, weights of both Part1 are loaded with the configutation pickle file saved.
- weights from part1.h5 file is loaded into the model generated using the above pickle file for Part-I
- Test accuracy is calculated with the weights and model configuration loaded.
- Similarly model config for part-II is generated and part2.h5 weights are loaded into the model generated.

Steps to test model for Part-I

```
#load weights
weigths_part1 = torch.load('part1.h5')
# load model configuration
model_config_part_1 = pickle.load(open('model_config.pkl', 'rb'))

# generate model using the above configuration
model_test_part1 = NeuralNetwork(model_config_part_1)

# load weights into the model
model_test_part1.load_state_dict(weigths_part1)

loss_function = nn.BCELoss()
runner = ModelRunner(model_test_part1,device,loss_function, batch_size).eval(test_dataI
loss, accuracy,_,_ = runner.get_eval_stats()
print(loss, accuracy)
```

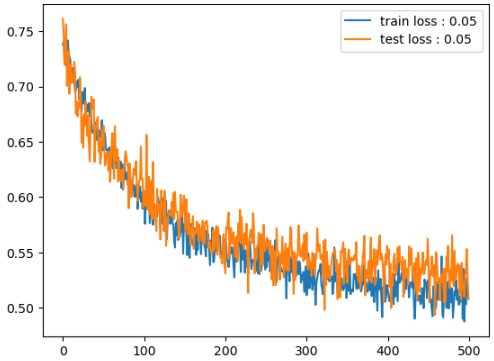
Steps to test model for Part-II

```
# Run part 2

# generating model for part-II
model_config_part2 = {}
model_config_part2['layers'] = []
createModelConfig(model_config_part2, "tanh", 0.0)
model_part2 = NeuralNetwork(model_config_part2)

# loading weights for part-II
weigths_part2 = torch.load('part2.h5')

# loading weights to model
model_part2.load_state_dict(weigths_part2)

# find test accuracy
runner = ModelRunner(model_part2,device,loss_function, batch_size).eval(test_dataBatch
loss, accuracy,_,_ = runner.get_eval_stats()
print(loss, accuracy)
```

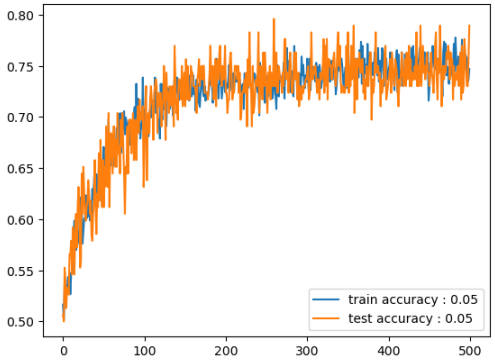
2 Part-II : Extension of Part-I

2.1 Parameter[Dropout]

Dropout is a regularization technique used in neural networks to prevent overfitting, which is a common problem(where model remembers datapoint instead of generalizing) when training neural network models.This experiment is to change the dropout values of the different layers in the base model and assess the performance of the model. Figure 5 shows different values of dropouts and corresponding loss and accuracy values of the model with same dataset as that of Part-I. From Figure 10, statistics the increase in dropout for the hidden layers of the model gradually decreases the accuracy of the model. Since, Dropout technique is to mask some of the hidden layers with given ratio randomly from each Linear layer added in the model. For this dataset dropout of the hidden layer nodes is not helping with the accuracy.

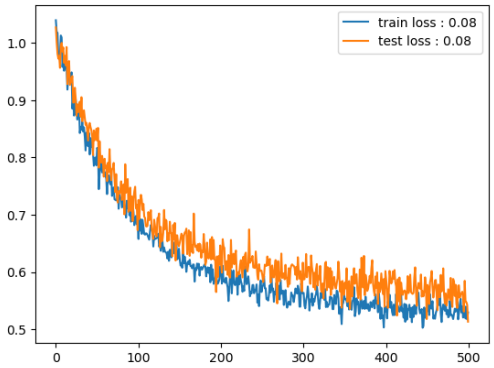


(a) Loss graph with 0.05 Dropout

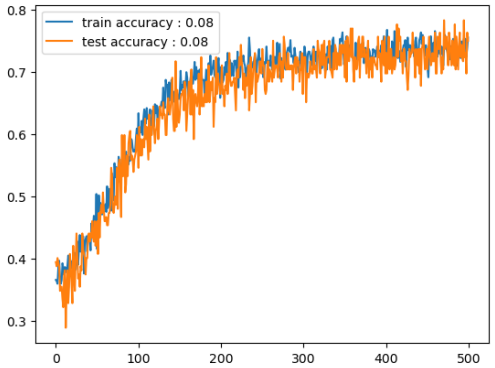


(b) Accuracy graph with 0.05 Dropout

Figure 6: Loss and Accuracy Plot for Dropout : 0.05

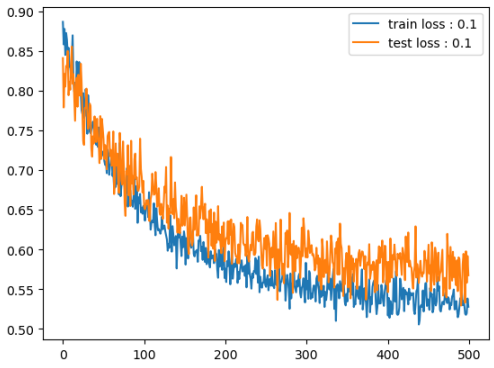


(a) Loss graph with 0.08 Dropout

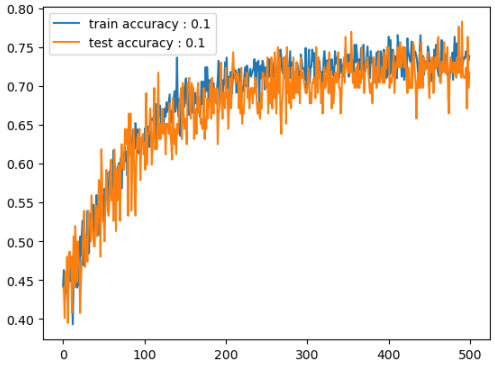


(b) Accuracy graph with 0.08 Dropout

Figure 7: Loss and Accuracy Plot for Dropout : 0.08

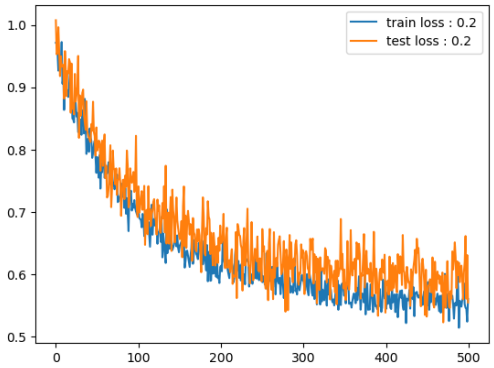


(a) Loss graph with 0.1 Dropout

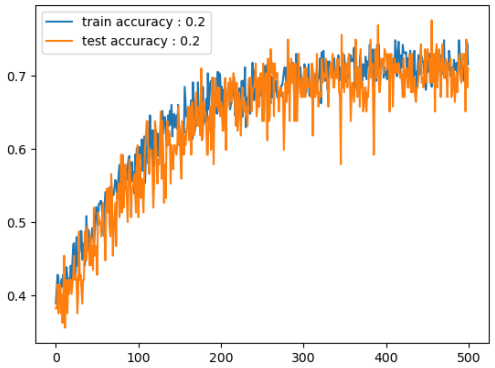


(b) Accuracy graph with 0.1 Dropout

Figure 8: Loss and Accuracy Plot for Dropout : 0.1



(a) Loss graph with 0.2 Dropout



(b) Accuracy graph with 0.2 Dropout

Figure 9: Loss and Accuracy Plot for Dropout : 0.2

	dropout	loss	accuracy
0	0.05	0.538207	0.743421
1	0.08	0.518068	0.743421
2	0.10	0.579011	0.710526
3	0.20	0.626975	0.684211

Figure 10: Dropout vs loss and accuracy for Base Model

2.2 Parameter[Optimizers]

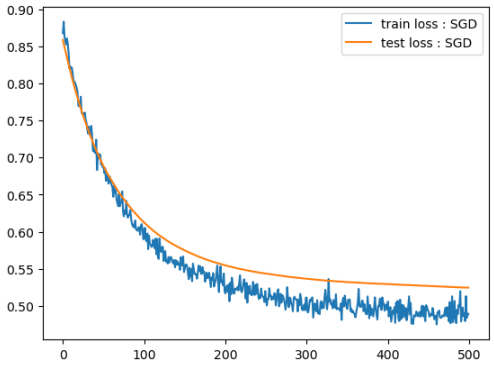
- When training a machine learning model, the performance of the model depends on Optimizer too.Optimizer algorithms used to update the model's weights during training to move in a direction to minimize the loss function(i.e opposite to the gradients based on the learning rate). Different algorithms use different techniques to update the weigths of the models but still tries to achieve same task i.e to decrease the

loss function. From the table below with same learning rate(0.0001) and same number of epochs(500), SGD performs well follwed by Adam, Adagrad and Adadelta in the same order as mentioned.

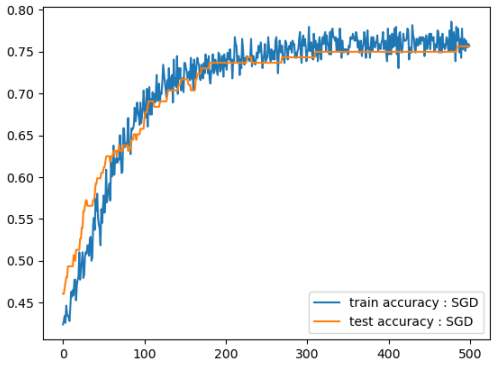
- By changing only optimizer from base model, from Figure 11 we can conclude that SGD is performing well compared to other optimizers. The same conclusion can be seen from Figure 12,13, 14 and 15 loss graphs, where the generalization gap is increasing after some epochs for optimizers other than SGD.
- Adam, Adagrad and Adadelta are adaptive optimization techniques. For same learning rate that of SGD, remaining optimizers have more fluctuations compared to SGD.

	optimizer	loss	accuracy
0	SGD	0.524574	0.756579
1	Adam	0.785722	0.710526
2	Adagrad	0.543129	0.723684
3	Adadelta	0.714658	0.506579

Figure 11: Loss and Accuracy for Base Model by using different Optimizers

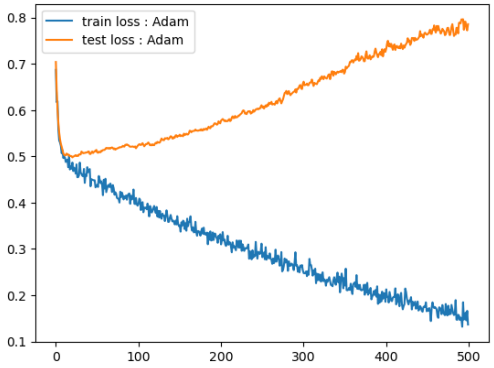


(a) Loss graph with SGD

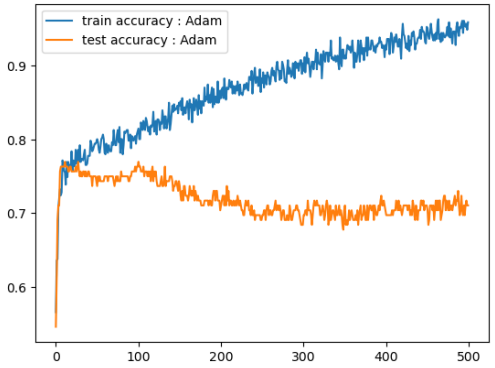


(b) Loss graph with SGD

Figure 12: Loss and Accuracy Plot with SGD

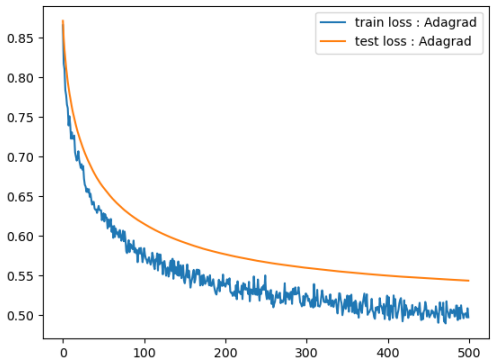


(a) Loss graph with Adam

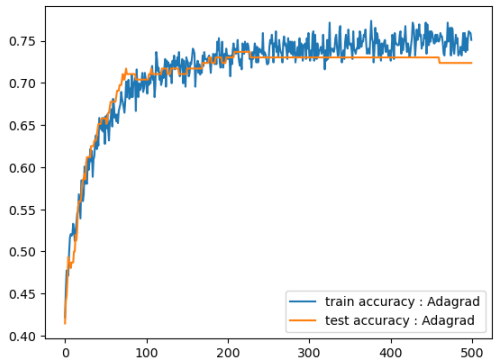


(b) Loss graph with Adam

Figure 13: Loss and Accuracy Plot with Adam

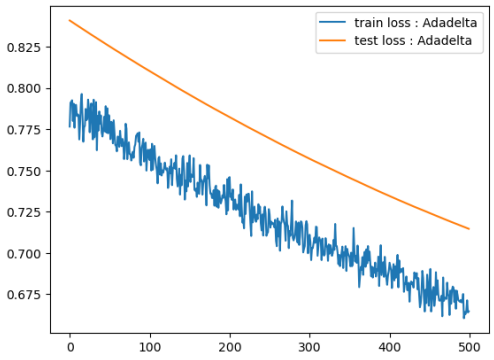


(a) Loss graph with Adagrad

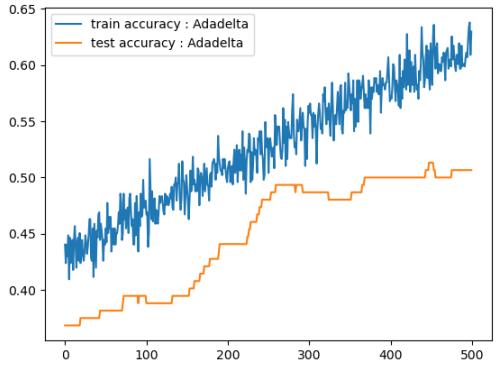


(b) Loss graph with Adagrad

Figure 14: Loss and Accuracy Plot with Adagrad



(a) Loss graph with Adadelta



(b) Loss graph with Adadelta

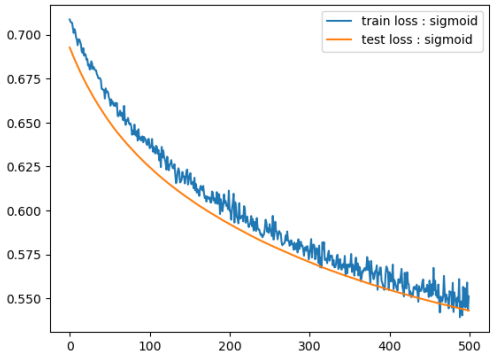
Figure 15: Loss and Accuracy Plot with Adadelta

2.3 Parameter[Activation functions for Hidden Layers]

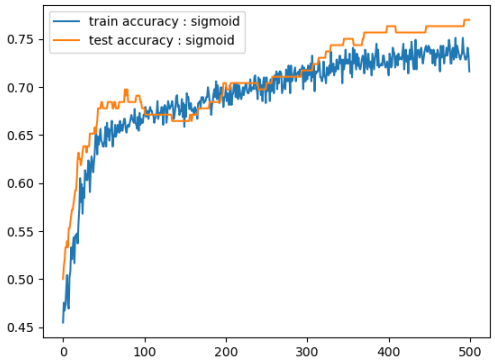
- Use of different activation functions in hidden layers of a neural network can boost its performance as these functions define a hidden layer's value that is then used by next layers. Different activation functions have different use case advantage over others. use of activation function also depends on the type of dataset we are classifying. For our dataset for same hyperparameters as above experiment sigmoid got more accuracy of 76.97, followed by tanh with 75.6 and ReLU with 73.68 percentage.
- Figure 16, shows that sigmoid and tanh performs well compared to ReLU with same hyperparameter and model setup. Since both sigmoid and tanh denotes all the values between 0 and 1, they act as normalize layers for hidden layers. whereas ReLU supress negative values and no effect on positive values(with high magnitude after summation while calculating hidden layer values).
- Figure 17,18 and 19 explains the same effect discussed above with increase in generalization gap in ReLU graph.

	activation_fn	loss	accuracy
0	sigmoid	0.543070	0.769737
1	ReLU	0.555035	0.736842
2	tanh	0.513197	0.756579

Figure 16: Activation Function vs loss and accuracy for Base Model

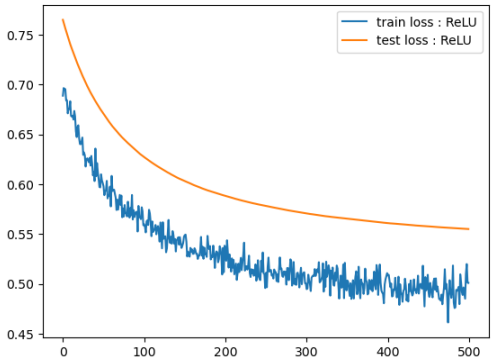


(a) Loss graph with sigmoid in Hidden Layers

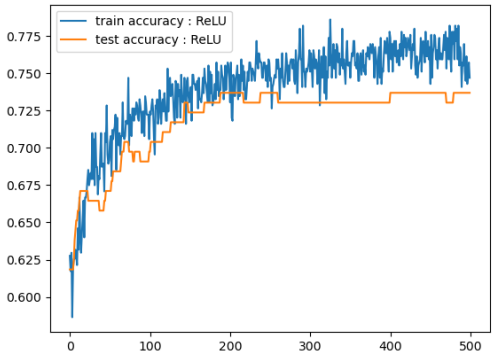


(b) Accuracy graph with sigmoid in Hidden Layers

Figure 17: Loss and Accuracy Plot with sigmoid in Hidden Layers

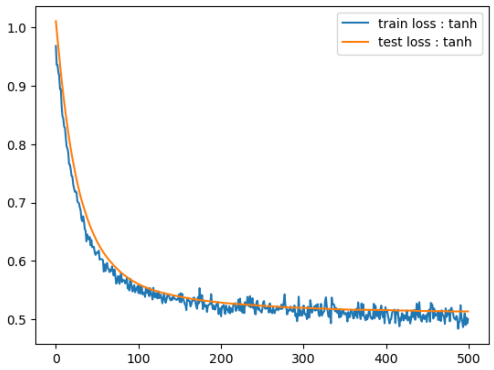


(a) Loss graph with ReLU in Hidden Layers

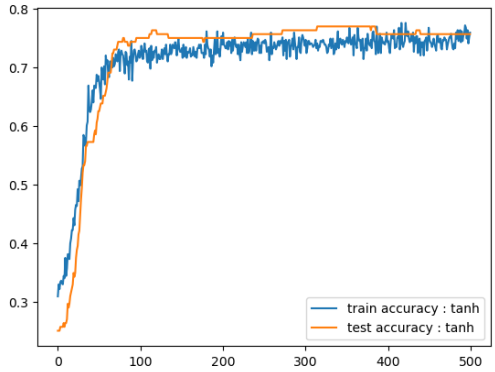


(b) Accuracy graph with ReLU in Hidden Layers

Figure 18: Loss and Accuracy Plot with ReLU in Hidden Layers



(a) Loss graph with tanh in Hidden Layers



(b) Accuracy graph with tanh in Hidden Layers

Figure 19: Loss and Accuracy Plot with tanh in Hidden Layers

2.4 Early stopping

```
NeuralNetwork(
(layers): Sequential(
        (0): Linear(in_features=7, out_features=64, bias=True)
        (1): Tanh()
        (2): Linear(in_features=64, out_features=64, bias=True)
        (3): BatchNorm1d(64, eps=1e-05, momentum=0.1, affine=True)
        (4): Linear(in_features=64, out_features=64, bias=True)
        (5): Tanh()
        (6): Linear(in_features=64, out_features=1, bias=True)
    )
)
```

This setup is considered as base model(for early stopping) after evaluating different techniques in Part-II, step 2. Figure 20, shows the loss graph between training and validation loss for above mentioned model with early stopping added to it(as from the plot we an see that x axis stopped after 232 epochs). This technique stops the model from overfitting with constraint being difference in loss between training and validation should be greater than 0.005(in order not to stop the training loop).

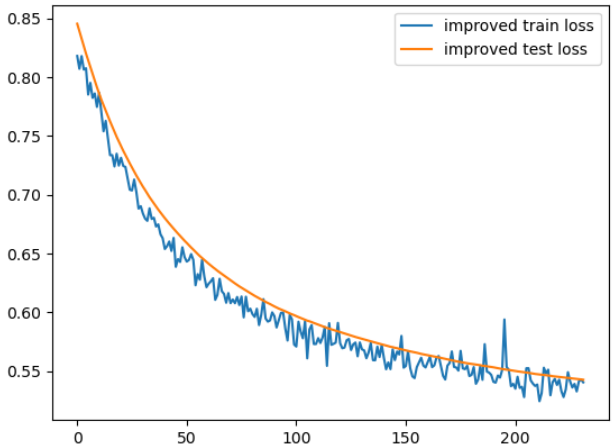
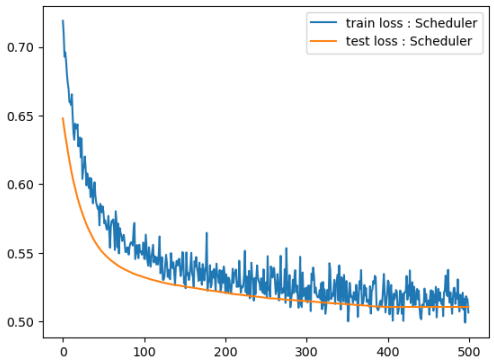


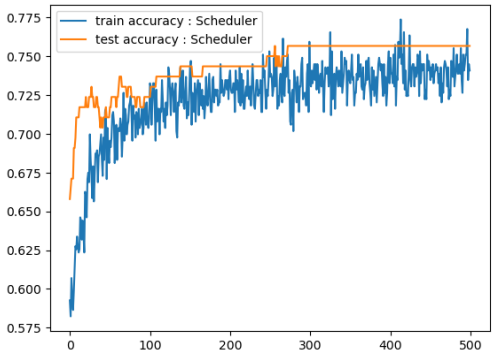
Figure 20: Activation Function vs loss and accuracy for Base Model

2.5 Learning rate Scheduler

Learning rate scheduler is a technique used to run training loop faster by changing the learning rate(based on several factors) and converging to minimum loss. In this experiment used reduce on plateau which helps in reducing the learning rate when there is less than threshold change in validation loss over a fixed number of epochs. This technique helps model in exploring the parameter space more when compared to model with fixed learning rate.



(a) Loss graph



(b) Accuracy graph

Figure 21: Loss and Accuracy Plot for model with Learning rate scheduler

---

3 Part-III : Convolution Neural Network on Dataset with 3 classes

3.1 About the Data

This dataset deals with images with three different classes of images (dogs, food and vehicles). input for the model is each image and target is same as the folder name of the images.

3.2 Pre-processing

Image preprocessing directly and indirectly helps in classification of the images. normalizing image pixel values will enhance the image features and supresses the noisy pixels. Normalization also helps in supressing the dominance of intensity values of the pixels. Data augumentation like horizontalFlip and rotation helps in avoiding overfitting the data (works as noise data and regularize the model). Resizing of the image and cropping the image helps in capturing the features that are more likely to be used to classify.

```
transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406],std=[0.229, 0.224, 0.225])
])
```

The above mentioned code is used to resize each image to 224x224 and convert PIL image to tensor(pixel values) and normalize the data using the mean and standard deviation values mentioned above for each channel seperately(since the images are RGB images, so 3 channels).
References : Compose, Normalize, Resize

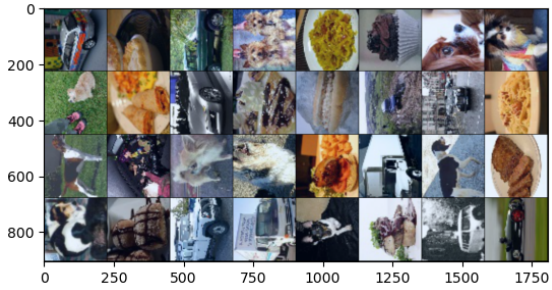


Figure 22: Image grid randomly picked from the dataset before normalizing

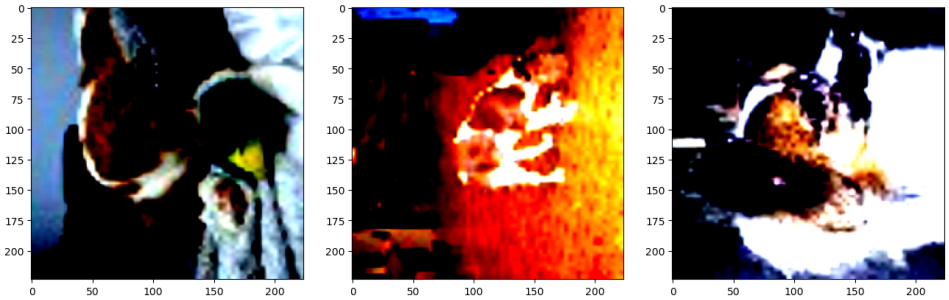


Figure 23: Images plot for class dog after Normalization

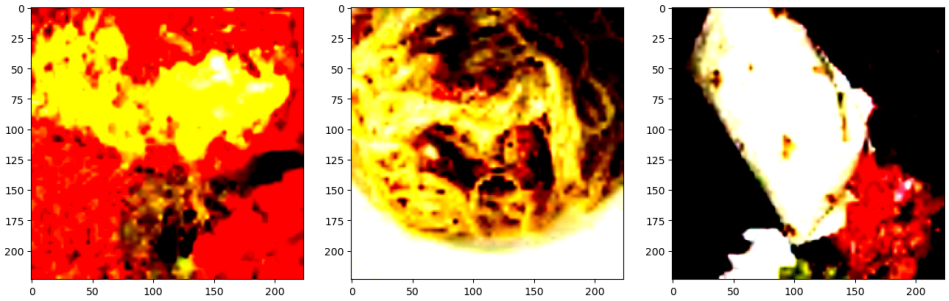


Figure 24: Images plot for class food after Normalization

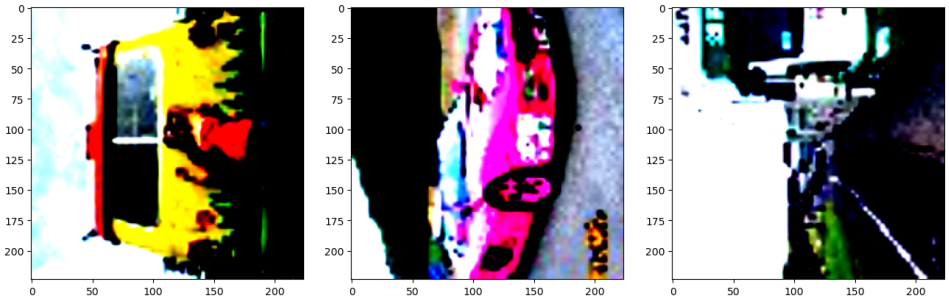


Figure 25: Images plot for class vehicles after Normalization

Figure 22 shows random images picked from the dataset without data augumentation step. Figure 23, 24 and 25 are plots for each class images after data augumentation (picked random images).

3.3 size of the dataset

Dataset consisits of three classes with 10k images under each class.

3.4 Structure of Convolution Neural Network

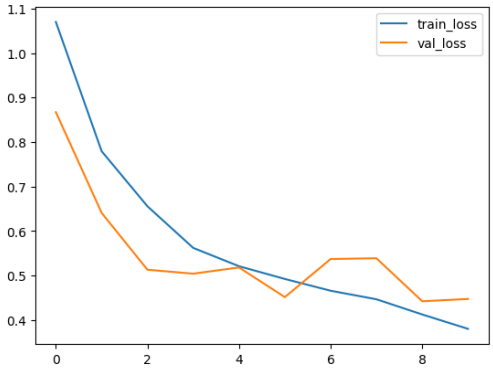
```
CustomAlexNet(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(11, 11), stride=(4, 4), padding=(2, 2))
    (1): ReLU(inplace=False)
    (2): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
    (3): Conv2d(64, 192, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (4): ReLU(inplace=False)
    (5): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
    (6): Conv2d(192, 384, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (7): ReLU(inplace=False)
    (8): Conv2d(384, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (9): ReLU(inplace=False)
    (10): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace=False)
    (12): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(6, 6))
  (classifier): Sequential(
    (0): Dropout(p=0.5, inplace=False)
    (1): Linear(in_features=9216, out_features=4096, bias=True)
    (2): ReLU(inplace=False)
    (3): Dropout(p=0.5, inplace=False)
    (4): Linear(in_features=4096, out_features=4096, bias=True)
    (5): ReLU(inplace=False)
    (6): Linear(in_features=4096, out_features=3, bias=True)
  )
)
```

Here AdaptiveAvgPool2d will take only outputsize as parameter. So, irrespective of the size of the previous conv2D it will be converted to output tensor(6 x 6 tensor in this example).

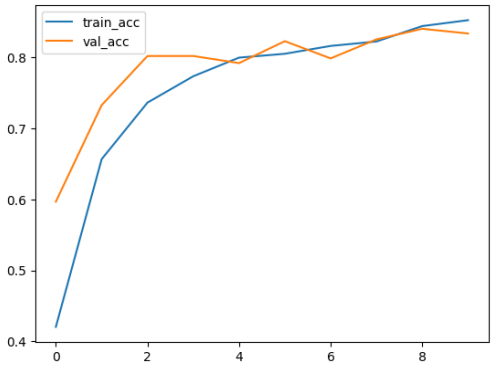
3.5 Train - Test graphs

	test_acc	epochs	batch_size	learning_rate	lamda1	lamda2	optimiser
0	0.979000	50	64	0.0010	0.00001	0.00001	Adagrad
1	0.337333	20	64	0.0010	0.00001	0.00001	SGD
2	0.919833	20	64	0.0010	0.00001	0.00001	Adagrad
3	0.898833	50	64	0.0005	NaN	NaN	NaN
4	0.970167	50	64	0.0005	NaN	NaN	NaN
5	0.951833	50	64	0.0010	NaN	NaN	NaN
6	0.951333	50	64	0.0010	0.00001	0.00001	NaN

Figure 26: Different Experiment setups and corresponding Test Accuracy



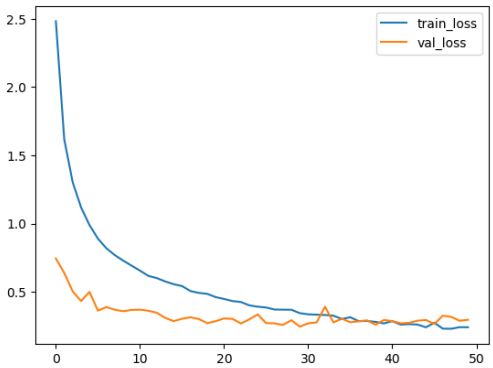
(a) Loss graph



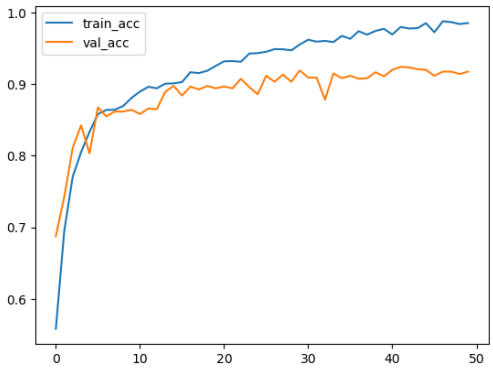
(b) Accuracy graph

Figure 27: Loss and Accuracy Plot

Figure 27 is the plot for CustomAlexNet without L1 and L2 regularization and took as base model.



(a) Loss graph



(b) Accuracy graph

Figure 28: Loss and Accuracy Plot with L1 and L2 Regularization

Figure 28 is the plot with L1,L2 regularization, which will make the model not to overfit but generalize the data.Validation accuracy for the model is 91.75 and training accuracy of 98.54 and testing accuracy is 97.90

4 Part-IV : Convolution Neural Network on SVHN Dataset

4.1 About the Data

SVHN (Street View House Numbers) is a dataset of over 600,000 digit images (32x32 pixels) obtained from house numbers visible in Google Street View. It is part of torchvision.dataset and can be pre-loaded with the API present in the torchvision dataset.

4.2 Pre-processing

```
# training transformations
transforms.Compose([
    transforms.Resize(224),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406],std=[0.229, 0.224, 0.225]),
])
# test transformations
transforms.Compose([
    transforms.Resize(224),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406],std=[0.229, 0.224, 0.225]),
])
```

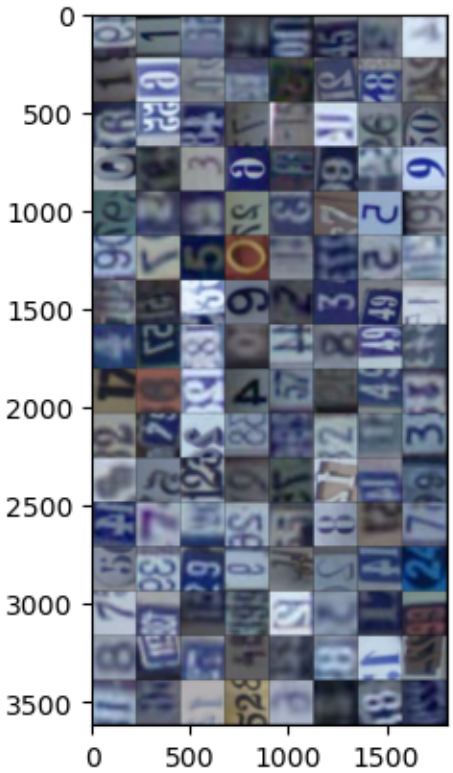


Figure 29: Random Images grid plot of SVHN Dataset without normalizing

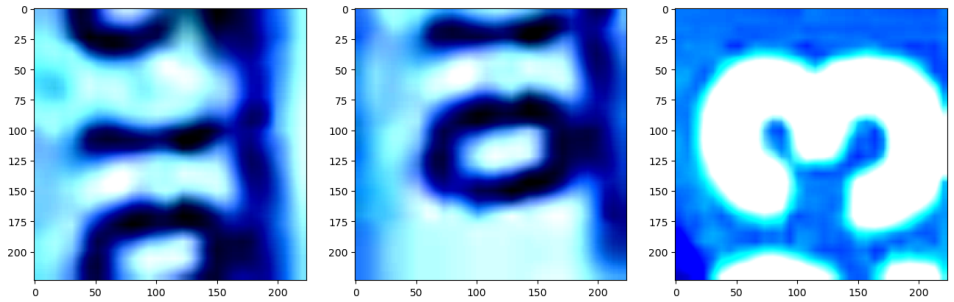


Figure 30: Images plot for random digit images after Normalization

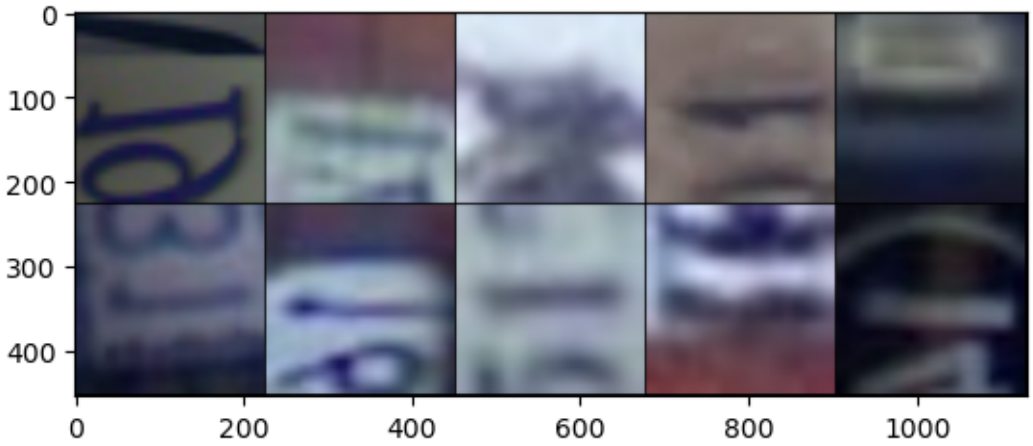


Figure 31: Different variants of digit 9 in the dataset after data augumentation

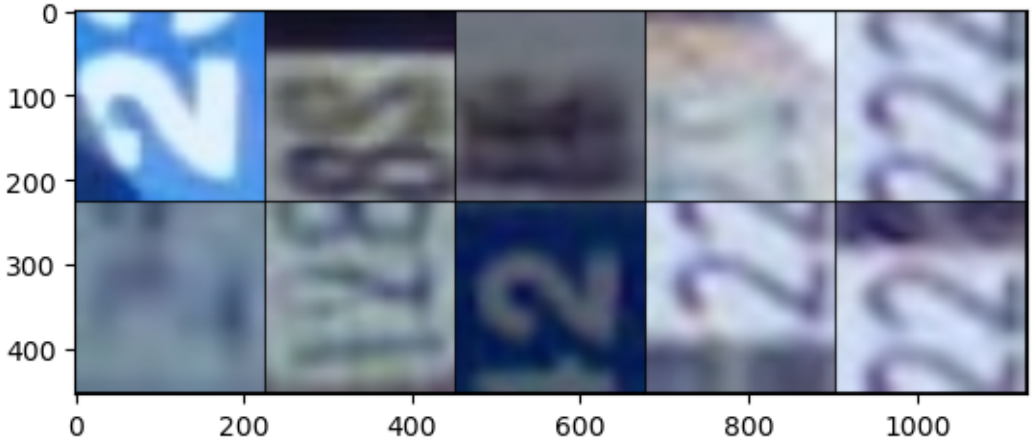


Figure 32: Different variants of digit 2 in the dataset after data augumentation

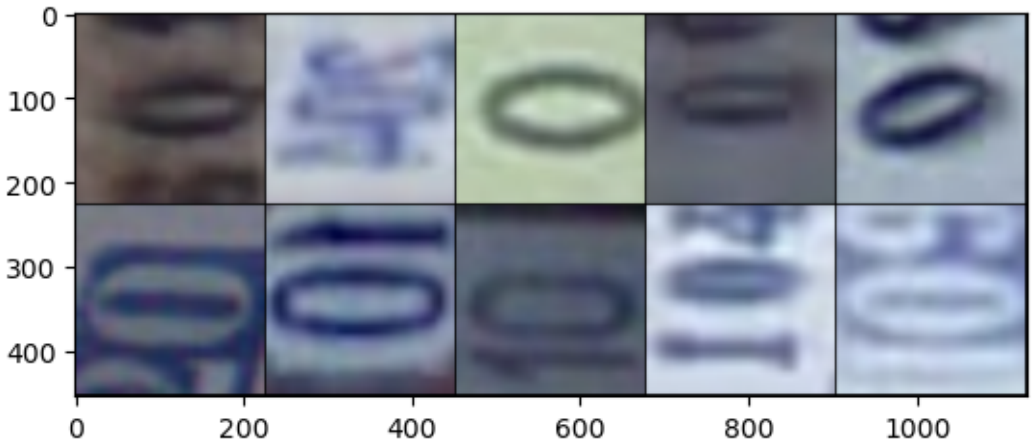


Figure 33: Different variants of digit 0 in the dataset after data augumentation

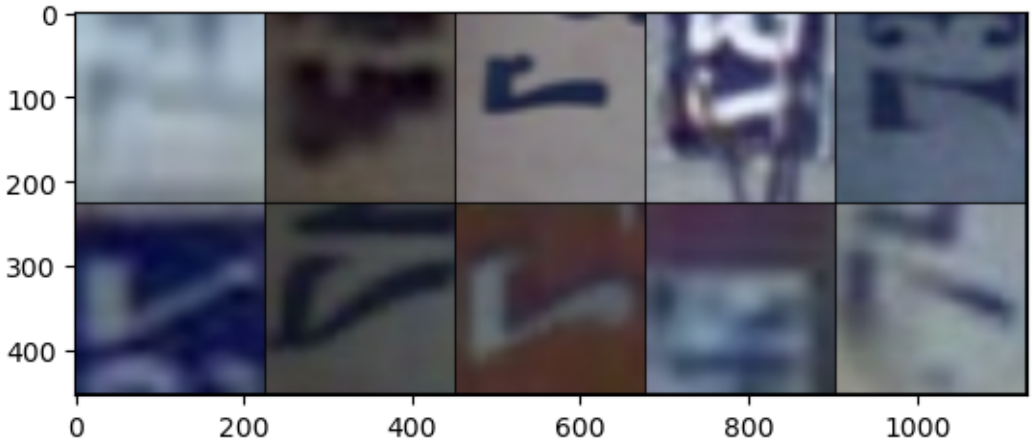


Figure 34: Different variants of digit 7 in the dataset after data augumentation

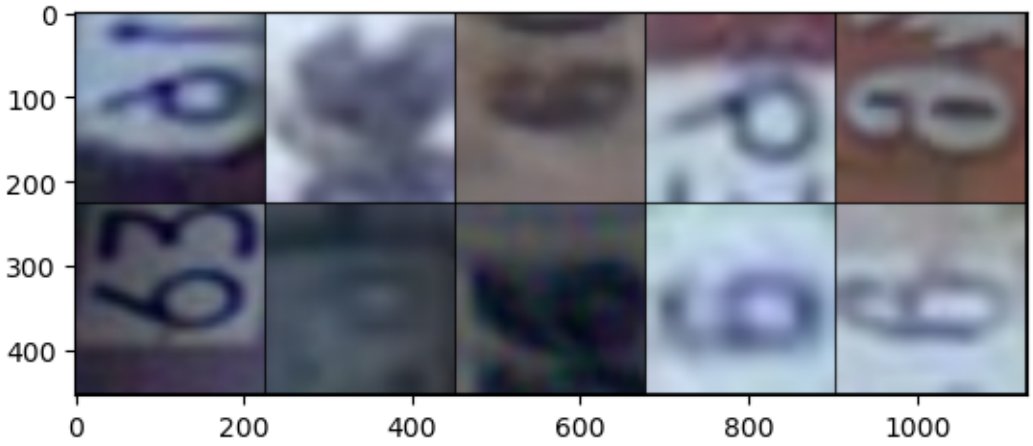


Figure 35: Different variants of digit 6 in the dataset after data augumentation

- Figure 29, plots grid of random images from the dataset before applying data augumentaion.
- Figure 30, highlights 3 random images after data augumentation.
- Figure 31 to 35, plots grid with different variants of individual digits (classes in the dataset).

4.3 size of the dataset

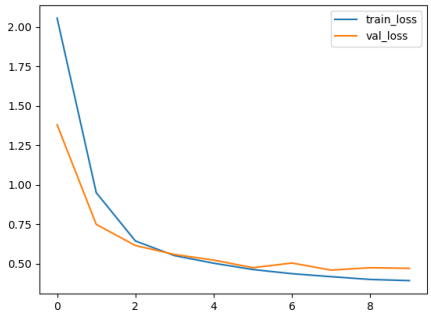
The dataset contains 60k images with 10 classes. Each class corresponding to a single decimal digit from 0 to 9.

4.4 Structure of the Model

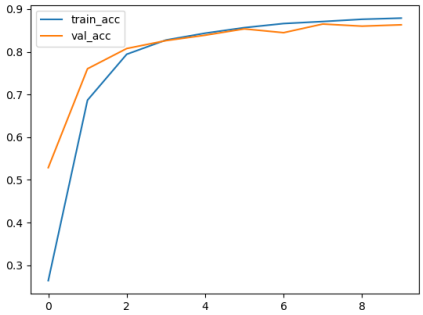
```
CustomAlexNet(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(11, 11), stride=(4, 4), padding=(2, 2))
    (1): ReLU(inplace=False)
    (2): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
    (3): Conv2d(64, 192, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (4): ReLU(inplace=False)
    (5): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
    (6): Conv2d(192, 384, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (7): ReLU(inplace=False)
    (8): Conv2d(384, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (9): ReLU(inplace=False)
    (10): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace=False)
    (12): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(6, 6))
  (classifier): Sequential(
    (0): Dropout(p=0.5, inplace=False)
    (1): Linear(in_features=9216, out_features=4096, bias=True)
    (2): ReLU(inplace=False)
    (3): Dropout(p=0.5, inplace=False)
    (4): Linear(in_features=4096, out_features=4096, bias=True)
    (5): ReLU(inplace=False)
    (6): Linear(in_features=4096, out_features=10, bias=True)
  )
)
```

The structure of the model uses Alex net as base model and changes the last layer to 10 output classes. It is a muliti-class classification used to classify 0 to 9 digits.

4.5 Train - Test graphs

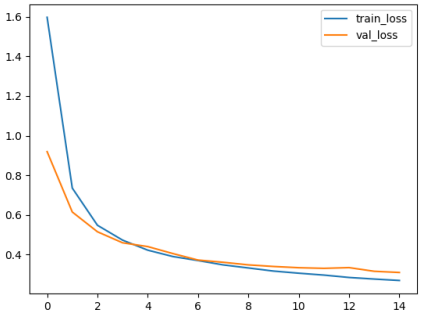


(a) Loss graph

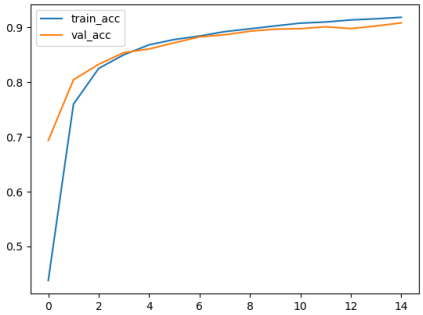


(b) Accuracy graph

Figure 36: Loss and Accuracy Plot for SVHN with 10 epochs

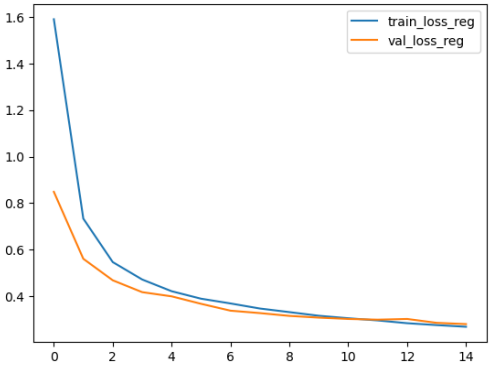


(a) Loss graph

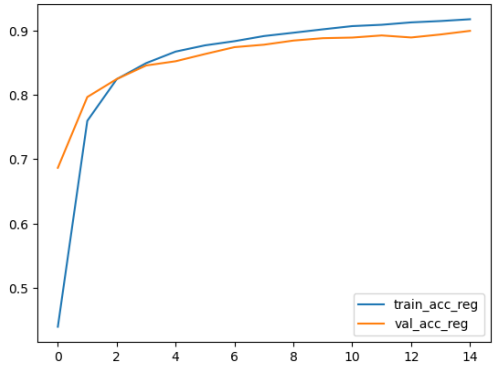


(b) Accuracy graph

Figure 37: Loss and Accuracy Plot for SVHN with 15 epochs



(a) Loss graph



(b) Accuracy graph

Figure 38: Loss and Accuracy Plot for SVHN with 15 epochs with L1,L2 Regularization

5 Bonus : VGG13 Model for Part-IV

5.1 Structure of the Model

```
customvggnet(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace=False)
    (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU(inplace=False)
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (6): ReLU(inplace=False)
    (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): ReLU(inplace=False)
    (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace=False)
    (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (13): ReLU(inplace=False)
    (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (15): ReLU(inplace=False)
    (16): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (17): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (18): ReLU(inplace=False)
    (19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (20): ReLU(inplace=False)
    (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (22): ReLU(inplace=False)
    (23): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (24): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (25): ReLU(inplace=False)
    (26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (27): ReLU(inplace=False)
    (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (29): ReLU(inplace=False)
    (30): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(7, 7))
  (classifier): Sequential(
    (0): Dropout(p=0.5, inplace=False)
    (1): Linear(in_features=25088, out_features=4096, bias=True)
    (2): ReLU(inplace=False)
    (3): Dropout(p=0.5, inplace=False)
    (4): Linear(in_features=4096, out_features=4096, bias=True)
    (5): ReLU(inplace=False)
    (6): Linear(in_features=4096, out_features=10, bias=True)
  )
)
```

6 References

- https://pytorch.org/vision/stable/transforms.html
- https://pytorch.org/docs/stable/nn.html
- https://pytorch.org/docs/stable/nn.functional.html
- https://pytorch.org/docs/stable/optim.html
- https://pytorch.org/vision/main/generated/torchvision.transforms.Compose.html
- https://pytorch.org/vision/main/generated/torchvision.transforms.Normalize.html
- https://pytorch.org/vision/main/generated/torchvision.transforms.Resize.html
- https://pytorch.org/docs/stable/generated/torch.nn.AdaptiveAvgPool2d.htmladaptiveavgpool2d
- https://pytorch.org/docs/stable/generated/torch.nn.Dropout.htmldropout
- https://pytorch.org/docs/stable/generated/torch.nn.Linear.htmllinear
- https://pytorch.org/docs/stable/generated/torch.nn.ReLU.htmlrelu
- https://pytorch.org/docs/stable/generated/torch.nn.Conv2d.htmlconv2d
- https://pytorch.org/docs/stable/generated/torch.nn.MaxPool2d.htmlmaxpool2d
- https://pytorch.org/docs/stable/generated/torch.nn.Sequential.htmlsequential
- https://pytorch.org/docs/stable/generated/torch.nn.Module.htmlmodule
- https://github.com/pytorch/vision/tree/v0.10.0/references
- https://pytorch.org/vision/main/auto_examples/index.html
- https://pytorch.org/vision/stable/models.html