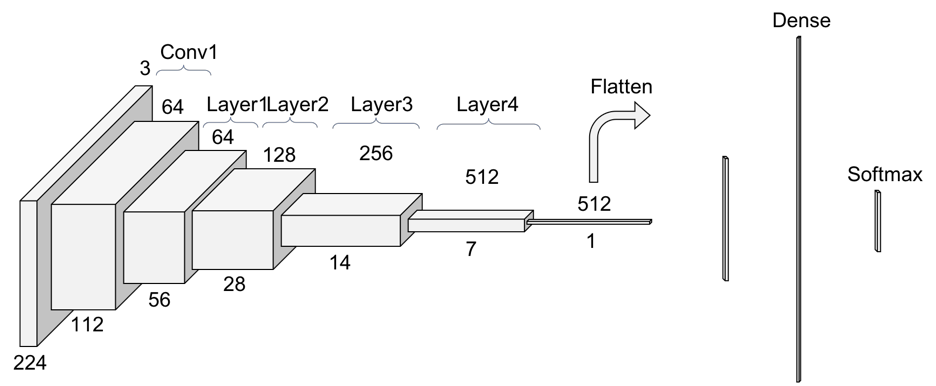


**FNN and ResNet-32 on SVNH dataset**

**1. Introduction**

The advent of deep learning has revolutionized the field of computer vision, enabling models to achieve human-like performance on various image recognition tasks. This report presents the design and implementation of a multi-layer neural network (NN) from scratch using Python and NumPy, and a convolutional neural network (CNN) using PyTorch, specifically ResNet-32 architecture. Both models are trained and evaluated on the Street View House Numbers (SVHN) dataset. The primary objectives are to compare the performance of the two implementations, explore different hyperparameters, and gain insights from hyperparameter tuning experiments through comprehensive visualizations.



Resnet Model Architecture

**2. Methodology**

The project is structured into several key components:

1. **Data Preprocessing**: Loading, normalizing, and splitting the SVHN dataset into training, validation, and test sets. Visualization of class distributions and data statistics.
2. **Neural Network Design**:
   * **From Scratch**: Implementing a 5-layer fully connected neural network with three hidden layers using ReLU activation functions, forward and backward propagation, and incorporating regularization techniques.
   * **Using PyTorch**: Implementing the same architecture using PyTorch for the fully connected network and a ResNet-32 CNN model.
3. **Training and Evaluation**: Training both models using different optimizers and learning rates, evaluating performance on validation and test sets, and calculating metrics like accuracy, precision, recall, F1-score, and confusion matrices.
4. **Hyperparameter Tuning**: Performing hyperparameter tuning using random search, experimenting with various learning rates, batch sizes, epochs, dropout rates, and optimizers to analyze their impact on model performance.

**3. Data Preprocessing**

**3.1 Loading the SVHN Dataset**

The SVHN dataset, which consists of real-world images of house numbers obtained from Google Street View, was loaded directly using the torchvision.datasets module. The dataset contains over 73,000 training images and 26,000 test images, each labeled with one of the digits from 0 to 9.

**3.2 Data Normalization**

To facilitate effective training, the pixel values of the images were normalized to the range [0, 1] by dividing by 255. This scaling is crucial for the convergence of gradient-based optimization algorithms.

**3.3 Data Splitting**

The dataset was split into training, validation, and test sets using an 80-20 split for the training and validation sets:

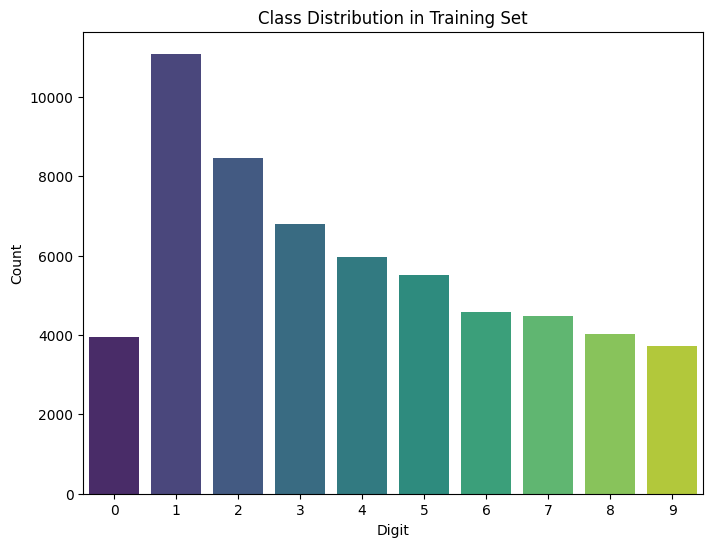
* **Training Set**: 73,257 images
* **Validation Set**: Approximately 18,314 images
* **Test Set**: 26,032 images

Stratified splitting was used to maintain the class distribution across all subsets.

**3.4 Data Visualization and Statistics**

**Class Distribution**

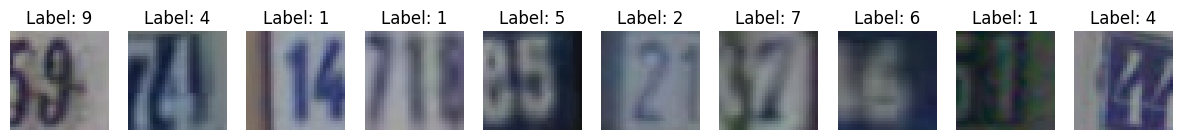
The class distribution was visualized using bar plots to ensure that the dataset is balanced across all classes.



*Figure 1: Class distribution in the training set.*

**Sample Images**

A set of sample images from the training set was displayed to gain insights into the dataset.



*Figure 2: Sample images from the training set with their corresponding labels.*

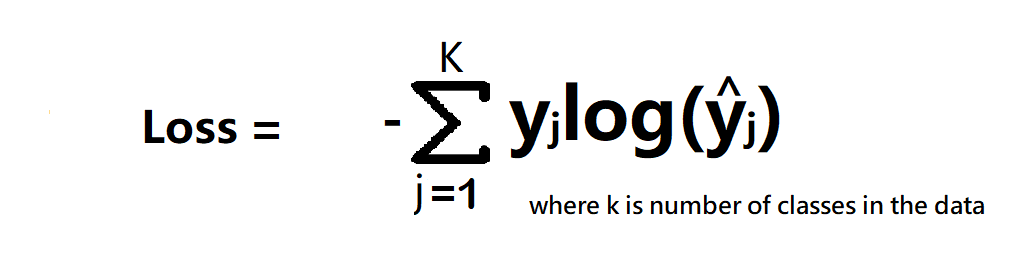
**4. Neural Network Design**

**4.1 Neural Network from Scratch**

**Architecture**

* **Input Layer**: 3,072 neurons (flattened 32x32 RGB images)
* **Hidden Layers**: Three hidden layers with 512, 256, and 128 neurons respectively
* **Output Layer**: 10 neurons (one for each class)
* **Activation Functions**: ReLU for hidden layers and softmax for the output layer

**Implementation Details**

* **Forward Propagation**: Implemented using matrix operations for efficiency.
* **Backward Propagation**: Derived analytical gradients for weight updates.
* **Loss Function**: Cross-entropy loss function was used.
* **Regularization**: Both L1 and L2 regularization techniques were experimented with to prevent overfitting.

**4.2 Neural Network Using PyTorch**

**Fully Connected Network**

A PyTorch implementation mirroring the architecture of the neural network from scratch was developed to leverage GPU acceleration and built-in functionalities.

**CNN Model**

The 5 layer model was implemented without importing pre-defined models from PyTorch, allowing for a deeper understanding of the architecture.

* **Architecture Highlights**:
  + **Convolutional Layers**: Multiple layers with residual connections to facilitate training of deep networks.
  + **Batch Normalization**: Applied to stabilize and accelerate training.
  + **Activation Functions**: ReLU used after each convolutional layer.
  + **Pooling Layers**: Max pooling layers to reduce spatial dimensions.
  + **Fully Connected Layers**: Final layers to output class probabilities.

**4.3 RESNET – 32 Using PyTorch**

The ResNet-32 architecture is a deep convolutional neural network designed with **residual blocks**. These blocks facilitate gradient flow through the network by using **skip connections**, which enable layers to learn the residual mapping instead of the full transformation. This results in faster convergence and better generalization.

**Key Features of ResNet-32:**

* **Convolutional Layers:** The model starts with a basic convolutional layer, followed by multiple convolutional layers in residual blocks.
* **Batch Normalization:** Batch normalization layers are used to stabilize the learning process and allow the network to train faster with better accuracy.
* **Residual Blocks:** Residual connections (skip connections) are used between layers, allowing the network to bypass certain transformations and ensure better gradient flow.
* **Pooling Layers:** Max-pooling layers are used to downsample feature maps, reducing the spatial dimensions.
* **Fully Connected Layer:** After global average pooling, a fully connected layer outputs class probabilities for the final classification task.

**Detailed Architecture:**

* **Initial Layer:**
  + 3x3 convolution with 16 filters and Batch Normalization, followed by ReLU activation.
* **Stage 1:**
  + 5 residual blocks, each consisting of two 3x3 convolutions with 16 filters and batch normalization, followed by ReLU activations. No downsampling is applied in this stage.
* **Stage 2:**
  + 5 residual blocks with 32 filters, with the first block performing downsampling by striding the convolution to reduce spatial dimensions. Batch normalization is applied after each convolution.
* **Stage 3:**
  + 5 residual blocks with 64 filters, with downsampling in the first block, followed by similar batch normalization and activation.
* **Final Layers:**
  + **Global Average Pooling** is applied to reduce the output from the final residual block to a 1x1 spatial size.
  + A **fully connected layer** with 10 output neurons corresponds to the number of classes in the classification task.

**3. Training and Hyperparameter Tuning**

The ResNet-32 model was trained using **PyTorch**, with experiments conducted to find the optimal hyperparameters.

**Hyperparameters:**

* **Learning Rate:** Initially set to 0.001 and reduced using a learning rate scheduler based on validation performance.
* **Batch Size:** 64 was used for most experiments, balancing memory usage and training efficiency.
* **Optimizer:** The Adam optimizer was used with L2 regularization (weight decay = 0.001) to prevent overfitting.
* **Epochs:** The model was trained for up to 20 epochs, with early stopping implemented to halt training if validation loss did not improve after 5 epochs.

**5. Training and Evaluation**

**5.1 Training Process**

**Hyperparameters**

* **Learning Rates**: Experimented with values like 0.1, 0.01, 0.001, 0.0001.
* **Batch Sizes**: Tested batch sizes of 32, 64, and 128.
* **Optimizers**: Adam, SGD with momentum, and RMSprop were used.
* **Number of Epochs**: Ranged from 10 to 30 epochs.
* **Regularization**: L1 and L2 regularization with lambda values of 0.001.

**Training Loop**

* Implemented mini-batch gradient descent.
* Recorded training and validation loss and accuracy at each epoch.
* Employed early stopping to prevent overfitting based on validation loss.

**5.2 Evaluation Metrics**

* **Accuracy**: Proportion of correct predictions over total predictions.
* **Precision**: Ability of the model to identify only relevant classes.
* **Recall**: Ability of the model to find all relevant cases.
* **F1-Score**: Harmonic mean of precision and recall.
* **Confusion Matrix**: Detailed breakdown of true vs. predicted classes.

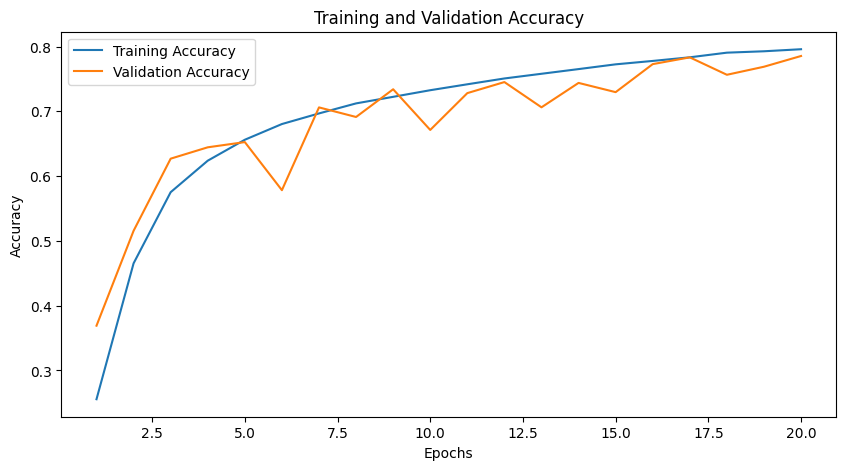
**5.3 Results**

**Neural Network from Scratch**

* **Training Loss and Accuracy**:

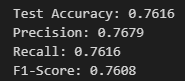


*Figure 3: Training and validation loss over epochs.*

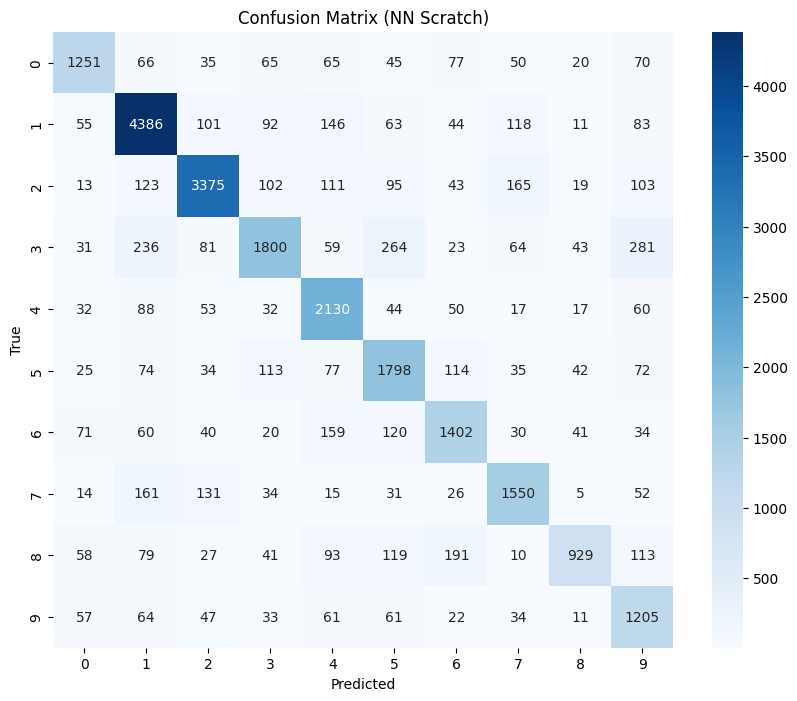


*Figure 4: Training and validation accuracy over epochs.*

* **Test Set Performance**:
  + **Accuracy**: 76.6%
  + **Precision**: 76.79%
  + **Recall**: 76.16%
  + **F1-Score**: 76.08%



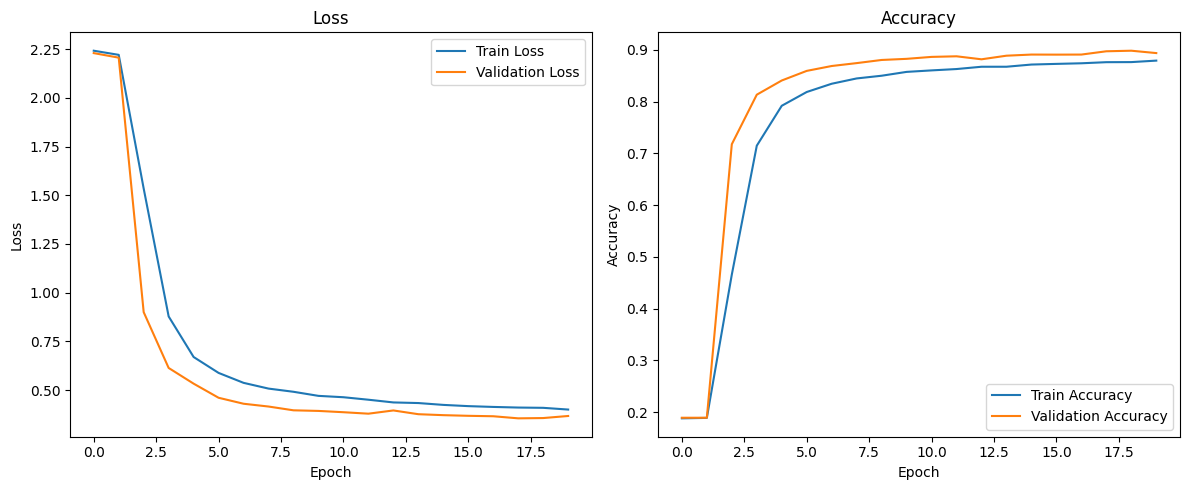
*Test Set metrices*

* **Confusion Matrix**: The confusion matrix shows us how majority of the samples are correctly classified in the dataset, meanwhile there are some pretty evident misclassifications throughout the dataset due to quality of the dataset and class imbalance inherent in training data.
* 

*Figure 5: Confusion matrix for the neural network implemented from scratch.*

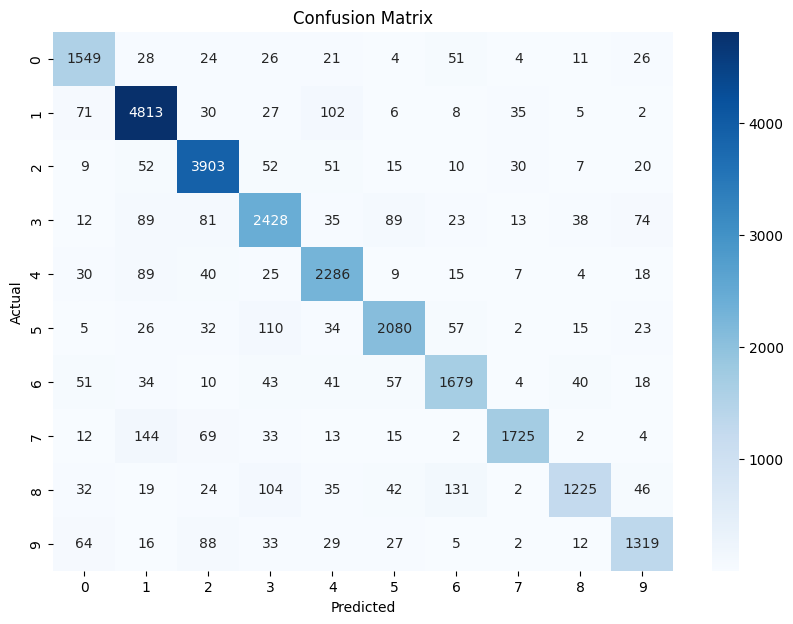
**PyTorch NN Model:**

* **Training Loss and Accuracy**:



*Figure 6: Training and validation loss and accuracy over epochs.*

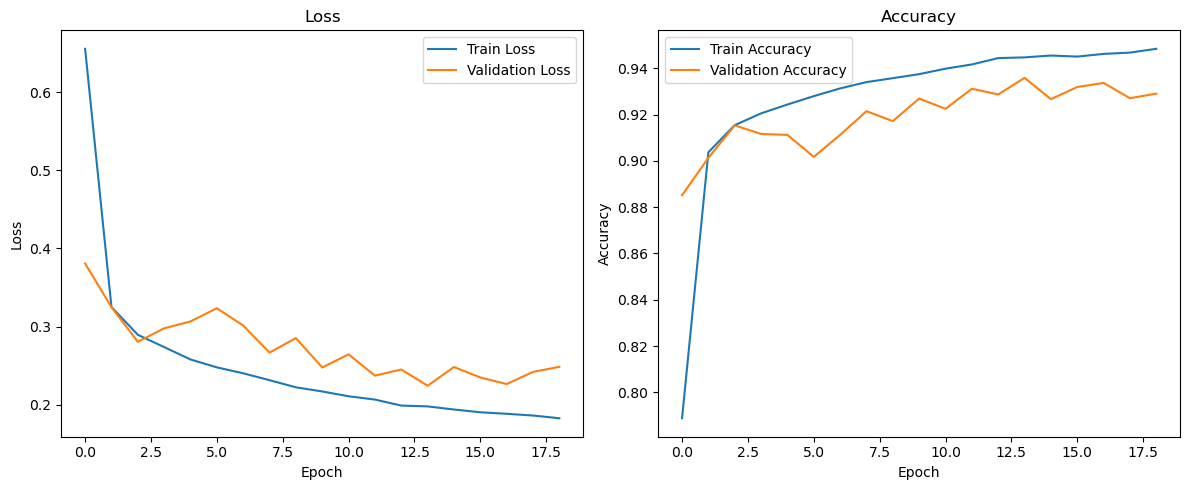
* **Test Set Performance**:
  + **Accuracy**: 88.39%
  + **Precision**: 88.43%
  + **Recall**: 88.38%
  + **F1-Score**: 88.31%
* **Confusion Matrix**: We can see that our ResNet model gives us better results as compared to Vanilla FFN. This is due to Convolution Operations applied by the filters. Hence CNN seems to perform better in this classification task.



*Figure 8: Confusion matrix for the ResNet-32 model implemented in PyTorch.*

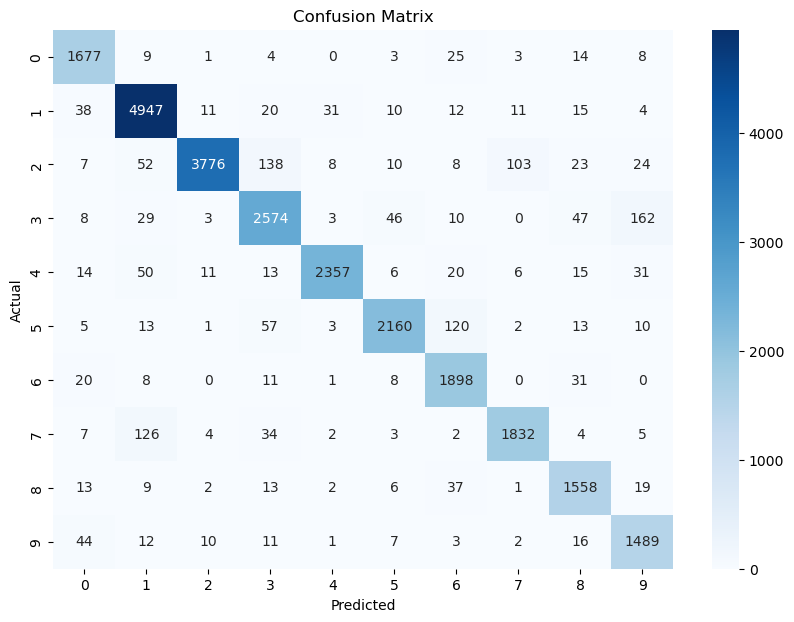
**Resnet – 32 Model:**

* **Training Loss and Accuracy**:



*Figure 6: Training and validation loss and accuracy over epochs.*

* **Test Set Performance**:
  + **Accuracy**: 93.22%
  + **Precision**: 93.41%
  + **Recall**: 93.38%
  + **F1-Score**: 93.31%
* **Confusion Matrix**: We can see that our ResNet model gives us the best results as compared to Vanilla FFN and the CNN block. This is due to Convolution Operations applied by the filters and the Residual blocks. Hence CNN seems to perform better in this classification task.



*Figure 8: Confusion matrix for the ResNet-32 model implemented in PyTorch*

**6. Hyperparameter Tuning**

**6.1 Random Search**

A random search strategy was employed over 20 trials to explore the hyperparameter space:

* **Learning Rates**: [0.1, 0.01, 0.001, 0.0001]
* **Batch Sizes**: [32, 64, 128]
* **Epochs**: [10, 20, 30]
* **Dropout Rates**: [0.3, 0.5]
* **Optimizers**: ['Adam', 'SGD', 'RMSprop']

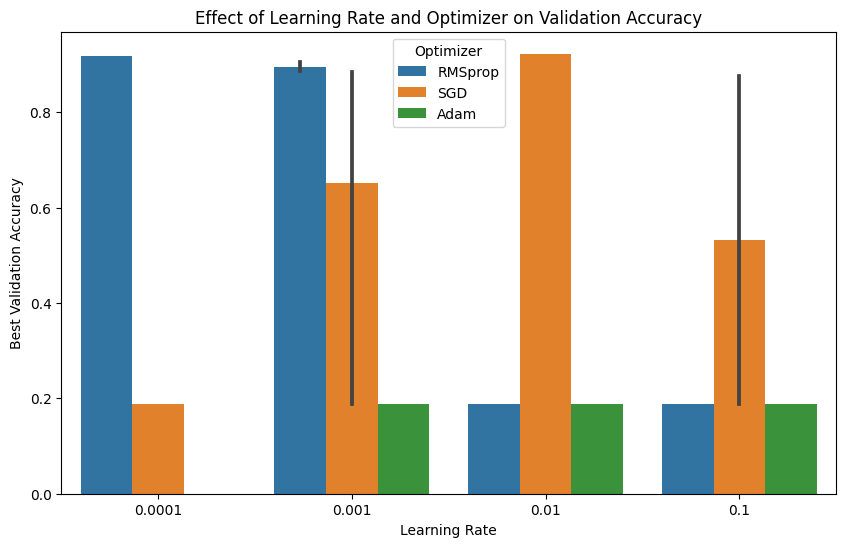
**6.2 Results and Analysis**

* **Best Hyperparameters**:
  + **Learning Rate**: 0.001
  + **Batch Size**: 64
  + **Epochs**: 20
  + **Dropout Rate**: 0.5
  + **Optimizer**: Adam



*Best Hyperparameters through Random Search*

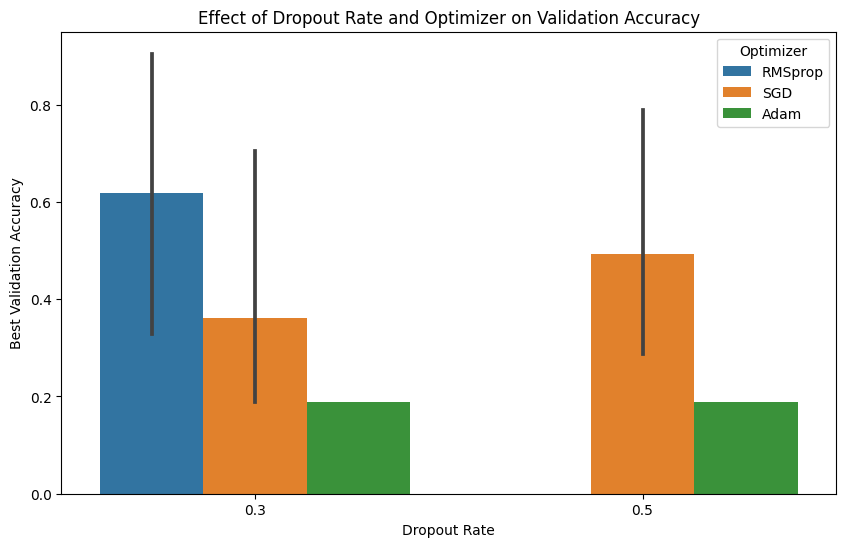
* **Effect of Hyperparameters on Validation Accuracy**:



*Figure 9: Impact of learning rate and optimizer on validation accuracy.*



*Figure 10: Impact of batch size and optimizer on validation accuracy.*



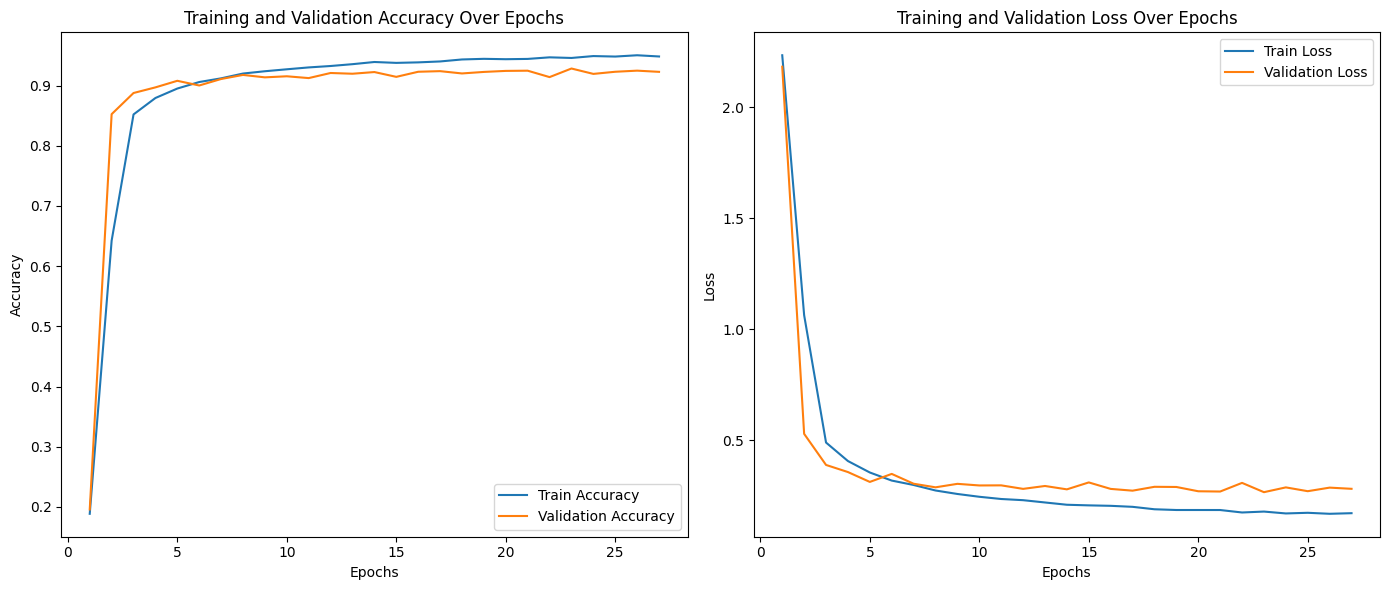
*Figure 11: Impact of dropout rate and optimizer on validation accuracy.*

**6.3 Observations**

* **Learning Rate**: Lower learning rates (0.001) consistently led to better convergence.
* **Batch Size**: A batch size of 32 and 64 both provided a good balance between training time and convergence stability.
* **Dropout Rate**: A dropout rate of 0.3 helped more in reducing overfitting and more validation accuracy.
* **Optimizers**: RMSprop optimizer performed better in terms of final accuracy as compared to Adam and SGD.

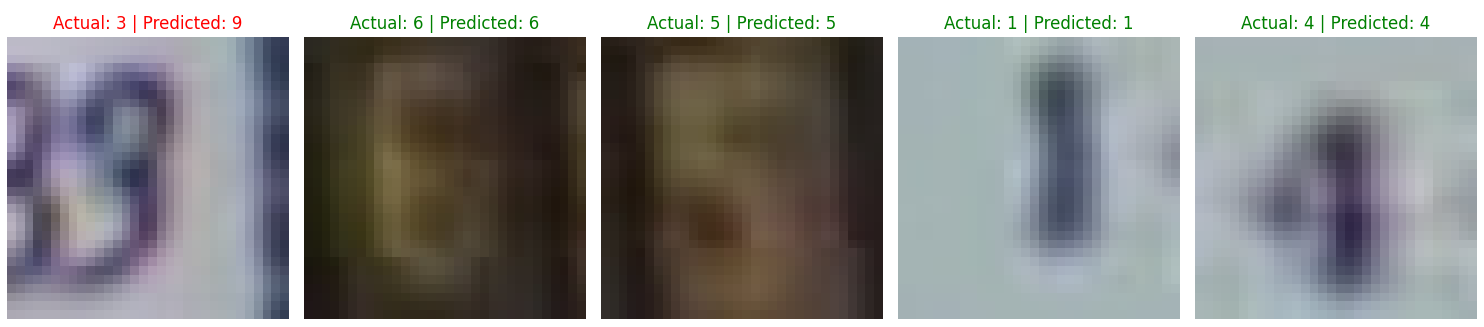
**7. Visualizations**

**7.1 Getting the best model through hyperparameters**

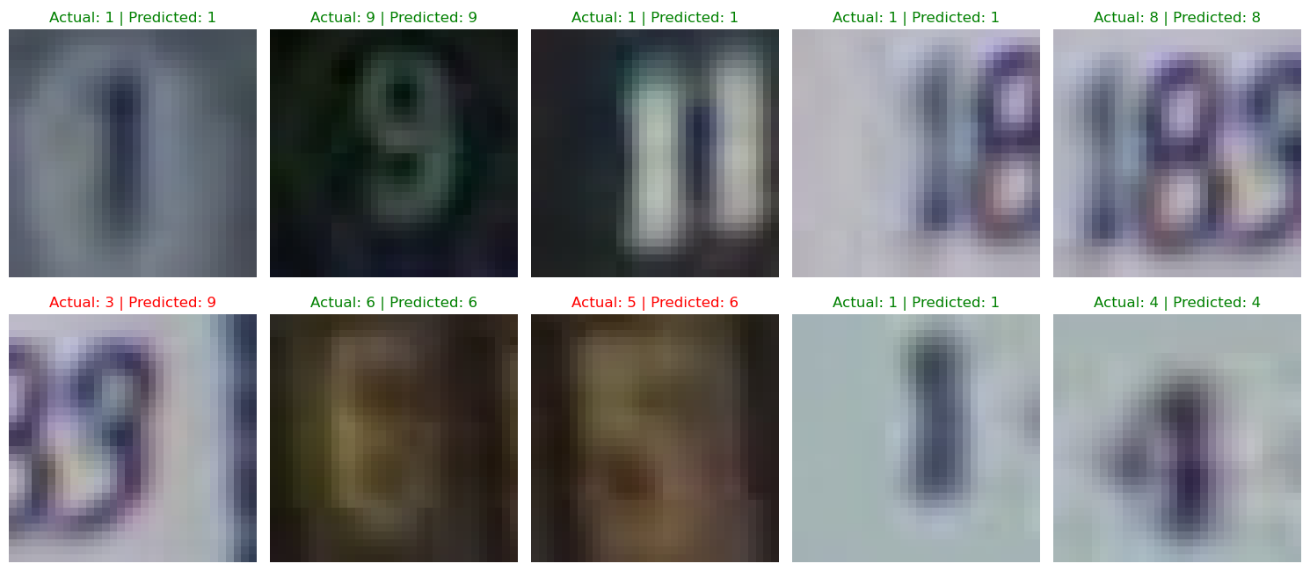
****

*Figure 13: Accuracy and Loss Graphs for the best model.*

**7.1 Sample Predictions**

****

*Figure 12: Sample images as classified by the CNN model.*

**

*Figure 13: Sample images as classified by the ResNet-32 model.*

We can see that the classification seems good from the images tested. The model can sometimes perform bad due to some samples which can be confusing to the model as well. Example, the first image is a 9, but due to it being very similar to 3, the model here classifies it as 3.

Hence, we need **to improve on our training set** and the model to incorporate these misclassifications as well.

**8. Conclusion**

The project successfully implemented and compared a multi-layer neural network from scratch , similar CNN model and a proper Resnet-32 using PyTorch on the SVHN dataset.

The PyTorch implementation with the ResNet-32 architecture outperformed both the CNN and the neural network from scratch in terms of accuracy and convergence speed. Hyperparameter tuning revealed that the choice of optimizer, learning rate, and regularization techniques significantly impact the model's performance. The experiments underscored the importance of proper data preprocessing, careful selection of hyperparameters, and the benefits of using advanced architectures like ResNet in achieving higher accuracy in image classification tasks.

*---------------------------------------------------- Report Ends --------------------------------------------------------*