14:332:435:04 TOPICS IN ECE: INTRODUCTION TO DEEP LEARNING

FINAL PROJECT REPORT

NAME: PARTH KALPESH KHARKAR
NET ID: pk674

EVALUATE THE PERFORMANCE OF DIFFERENT TYPES OF OPTIMIZERS ON A LeNet-5 NETWORK USING MNIST DATA.
AT LEAST YOU NEED TO EVALUATE SGD, AdaGrad, RMSprop.

TABLE OF CONTENTS:

ABSTRACT

In this project, we analyzed the performance of three popular optimizers: Stochastic Gradient Descent (SGD), AdaGrad, and RMSprop on the LeNet-5 network using the MNIST dataset. The LeNet-5 network is a convolutional neural network (CNN) architecture that is widely used for image recognition tasks. It comprises of two convolutional layers, followed by two fully connected layers and a softmax output layer. The MNIST dataset, consisting of 70,000 images of handwritten digits, is a well-known benchmark dataset used for evaluating the performance of image recognition models.

To compare the performance of these optimizers, we trained the LeNet-5 network using each optimizer separately and evaluated the accuracy and training time for each optimizer. The accuracy of the model was calculated by evaluating the loss as compared to the number of epochs computed. The training time was measured by recording the time taken to train the model for a fixed number of epochs.

INTRODUCTION

Deep learning is a branch of machine learning that focuses on teaching artificial neural networks how to carry out difficult tasks including decision-making, speech recognition, picture recognition, and natural language processing. By using multiple layers of interconnected nodes, which are also referred to as artificial neurons, to create a computational model that can learn from and make predictions based on the data it has received. Deep learning aims to create algorithms that can learn from and make predictions on large and complex datasets.

Optimizers are methods which are used to reduce the loss while improving the efficiency of the neural network. The optimizer learns about the attributes which are required and assists in improving the accuracy. There are various optimization algorithms which are available and each one of them have their unique way of updating the model parameters. In this project, we are going to focus on SGD (Stochastic Gradient Descent), AdaGrad, RMSprop.

SGD stands for Stochastic Gradient Descent. SGD changes the parameters for each sample or group of samples and hence the term “stochastic” is used. This is also one of the reasons why SGD computations are efficient while using larger datasets. It is an optimization algorithm that uses the gradient of the loss function with respect to the model’s parameters to update the parameters in the direction of the negative gradient. It iteratively modifies the parameters in the direction of the steepest descent to try to discover the minimum of the loss function.

AdaGrad stands for Adaptive Gradient. It basically gives more weight to the parameters that have been updated less frequently and less weight to those that have been updated frequently. AdaGrad is an extension of the gradient descent optimization algorithm that enables the step size in each dimension to be automatically adjusted based on the gradients seen for the variables seen throughout the search. This method helps AdaGrad to convex quickly on flat directions while simultaneously making progress on the curved directions.

RMSprop stands for Root Mean Square Propagation. RMSprop is an extension of the gradient descent optimization process and is intended to address some of its drawbacks, including the issues of vanishing or exploding gradients. The fundamental concept here in RMSprop is to divide the learning rate by the sum of the gradient magnitudes for each weight.

RELATED WORK

In 1998, Yann LeCun, Leon Bottou, Yoshua Bengio, and Patrick Haffner presented the LeNet-5 architecture. This was one of the earliest convolutional neural networks (CNNs) to perform well on tasks requiring the recognition of handwritten digits, the MNIST dataset. LeNet-5 was trained using the backpropagation algorithm with gradient descent optimization and had two convolutional layers, two fully connected layers, and was composed of these layers. It immediately rose to popularity as a solution for image recognition problems and provided the basis for numerous subsequent CNN systems.

Since the 1990s, the MNIST dataset has been used as a benchmark for assessing machine learning methods for handwritten digit recognition. The dataset consists of 70,000 handwritten digits in grayscale, with 60,000 photos used for training and 10,000 for testing. The photographs are scaled down to 28 × 28 pixels and are centered. From conventional classifiers to deep learning models, the MNIST dataset has been extensively used in the creation and assessment of numerous machine learning techniques.

The classic Stochastic Gradient Descent (SGD) optimizer is one example of an optimizer for neural networks. Adagrad is a good choice for sparse data since it adjusts the learning rate for each parameter based on its past gradients. By increasing the learning rate in accordance with the magnitude of the gradients' most recent gradients, RMSprop improves convergence by altering the SGD method. By modifying the learning rate and momentum for each parameter, Adam combines the advantages of both Adagrad and RMSprop. Nadam is an extension of Adam that includes Nesterov momentum, allowing the optimizer to reach the optimal solution more quickly. These optimizers are now frequently employed while training neural networks and have significantly contributed to the advancement of deep learning technology.

DATA DESCRIPTION

The MNIST database is a well-known and widely used dataset in the field of deep learning. It was created by modifying a larger dataset of handwritten digits collected by the National Institute of Standards and Technology (NIST). The modified dataset consists of 70,000 grayscale images of handwritten digits, with each image centered and normalized to the same size of 28 x 28 pixels. These images are further divided into a training set of 60,000 images and a test set of 10,000 images.

The purpose of creating the MNIST dataset was to provide a benchmark for image classification tasks, specifically for handwritten digit recognition. The dataset has been used extensively to train and evaluate machine learning models, ranging from simple linear classifiers to more complex deep learning models. The availability of a standardized dataset has helped researchers to compare different methods and assess the state-of-the-art performance of image classification algorithms.

The MNIST dataset has had a significant impact on the field of deep learning and has led to several breakthroughs in image classification tasks. However, it is also important to note that the dataset has its limitations. The dataset consists only of grayscale images of handwritten digits, which makes it a relatively simple task compared to real-world image classification problems. Therefore, researchers have also developed larger and more complex datasets, such as ImageNet, to address some of these limitations. Nonetheless, the MNIST dataset remains a valuable benchmark for evaluating new image classification methods and serves as a starting point for many machine learning projects.

METHOD DESCRIPTION

In this project, we are implementing different optimizers for training a neural network on the MNIST dataset. Each optimizer is defined in a separate Python file with a class that contains a train and a test method. The neural network architecture in this project is defined by the 'Net' class. It inherits from the 'nn.Module' class which is provided by PyTorch. This particularly helps us offer tools that are required to define and manage neural network modules. The '__init__' function defines the required convolutional and linear layers of the neural network architecture. It initializes the layers of the network, such as two convolutional layers, two max-pooling layers and two fully connected layers.

```python
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 20, 5, 1)
        self.dropout = nn.Dropout(0.5)
        self.conv2 = nn.Conv2d(20, 50, 5, 1)
        self.bn = nn.BatchNorm2d()
        self.fc1 = nn.Linear(4*4*50, 500)
        self.fc2 = nn.Linear(500, 10)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = F.max_pool2d(x, 2, 2)
        x = F.relu(self.conv2(x))
        x = F.max_pool2d(x, 2, 2)
        x = x.view(-1, 4*4*50)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return F.log_softmax(x, dim=1)
```

The 'NN_model' function is used to train LeNet-5. The train function is called for each iteration of the specified number of epochs to update the network's weight in accordance with the training data. The test function is used to assess the model's performance using test data after each epoch. The function saves the trained model to a file if the save_model option is set to True.

The 'forward' function used to specify the neural network's forward pass. It accepts an input 'x' and applies activation functions like ReLU and max-pooling operation as it goes through the layers specified in the '__init__' method. The neural network's output is then returned after a log-softmax activation function has been applied.

The train method takes the number of epochs as input and trains the model on the training set using batches. Every epoch, the model iterates through the training set's batches one at a time while in the training mode. The model is utilized for each batch to determine the output, and the loss is calculated using the loss function. The optimizer modifies the model's parameter after computing the gradient using the backpropagation process. To calculate the average training loss returned at the conclusion of an epoch, the train method additionally keeps track of the accuracy and training loss for each batch.

```python
def train(self, epoch):
    self.model.train()
    train_loss = 0
    for batch_idx, (data, target) in enumerate(self.train_loader):
        self.optimizer.zero_grad()
        output = self.model(data)
        loss = F.nll_loss(output, target)
        loss.backward()
        self.optimizer.step()
        train_loss += loss.item()
        pred = output.argmax(dim=1, keepdim=True)
        correct = pred.eq(target.view_as(pred)).sum().item()
        accuracy = 100. * correct / len(data)
        self.train_acc.append(accuracy)
        if batch_idx % self.log_interval == 0:
            print('Epoch Set: {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}'.format(
                epoch, batch_idx *
                len(data), len(self.train_loader.dataset),
                100. * batch_idx / len(self.train_loader), loss.item()))
    train_loss /= len(self.train_loader.dataset)
    return train_loss
```

```python
def test(self):
    self.model.eval()
    test_loss = 0
    correct = 0
    with torch.no_grad():
        for data, target in self.test_loader:
            output = self.model(data)
            loss = F.nll_loss(output, target, size_average=False)
            print(loss.item())
            test_loss += loss.item()
            pred = output.argmax(dim=1, keepdim=True)
            correct += pred.eq(target.view_as(pred)).sum().item()

            test_loss /= len(self.test_loader.dataset)
            accuracy = 100. * correct / len(self.test_loader.dataset)
            self.test_acc.append(accuracy)

    print('Test set: Average loss: {:.4f}, Accuracy: {}/{} ({:.0f}%)\n'.format(
        test_loss, correct, len(self.test_loader.dataset), accuracy))

    return test_loss
```

In the test method, we assess the trained model's performance on the test set. The model is then set to the evaluation mode which iterates through the test set. The loss function is used to determine the loss for each sample as the model computes the output. The test accuracy is also recorded in a list for future analysis, and both the average test loss and accuracy are printed out. This helps us determine how effectively the trained model generalized on the previously unseen test data, which helps us to ensure that the model can generalize well to new data.

MODEL DESCRIPTION

In this project, we will start by importing modules which will be necessary for our model to run with the help of optimizers such as SGD, AdaGrad and RMSprop. Every optimizer is implemented in a different Python file with a set of procedure followed for the model run efficiently and with good accuracy. We will start by created neural network model 'Net' which requires the architecture needed for this project. The class 'Net' inherits from the function 'nn.Module' function which is used to train LeNet-5. The constructor of the class 'Net' is used to initialize two convolutional layers (conv1 and conv2) and two fully connected layers fc1 and fc2.

The forward propagation method is used to perform forward pass the network which involves passing the input through the convolutional layers and the fully connected layers by applying the proper activation functions such as ReLU and log-softmax function. Now, we will start by implementing a class which will be having function for training and testing the neural network. The constructor of the class takes in several arguments which are parameters such as batch_size, test_batch_size, epochs, lr, seed, log_interval, save_model, and ratio. Every parameter defined here is used to configure and control the process of the neural network.

For every optimizer a different Python file is created which will be having a class of that particular optimizer, in that class a function will be implemented which is a method to download and prepare the MNIST dataset for training and testing. Amongst the two DataLoader objects which are returned one is for the training set and the other for the testing set. A 'NN_model' function is defined under every optimizer class. Using the required optimizer this approach trains the neural network for the provided number of epochs. For each epoch, the 'NN_model' function calls the train method which trains the network on a single batch of data at a time and test method which assesses the network's performance on the test set.

```
class SGD_TrainTest:
    def __init__(self, batch_size=64, test_batch_size=1000, epochs=2, lr=0.01, momentum=0.5, seed=1, log_interval=10, save_model=False, ratio=0.5):
        self.batch_size = batch_size
        self.test_batch_size = test_batch_size
        self.epochs = epochs
        self.lr = lr
        self.momentum = momentum
        self.seed = seed
        self.log_interval = log_interval
        self.save_model = save_model
        self.ratio = ratio
        self.train_loader, self.test_loader = self._prepare_data_loaders()
        self.model = Net()
        self.optimizer = optim.SGD(self.model.parameters(), lr=self.lr, momentum=self.momentum)
        self.train_loss = []
        self.test_loss = []
        self.train_acc = []
        self.test_acc = []
```

Every optimizer class will have a train and test function. The train function implements a method uses a single batch of data to train the network, determines the accuracy and loss for the batch, then uses backpropagation to update the network's parameters. The test function implemented uses a method to determine the loss and accuracy to assess the network's performance on the test set. The plot_losses function implemented in the optimizer class is used to plot training and testing losses over the epochs using matplotlib library. An object should be instantiated for every optimizer class, its hyperparameters should be configured, and the NN_model should be called to train and test the neural network. The trained model is saved to a file with the name "mnist_LeNet-5.pt" if the value of save_model is True.

EXPERIMENTAL PROCEDURE

For this project, we will be following a step-by-step procedure which was used to build the neural network model by using three different optimizers. The steps are as follows:

(1) Initially, we will be importing the required libraries to define the LeNet-5 model and for training and testing the model.
(2) We define a class Net, which takes in 4 network layers for our neural network module. We define a forward propagation function with 5 network layers and derive a SoftMax value for our train data.
(3) We define another class, which defines our model. We create parameters for the model in the initial function and define a list to take in the accuracy and loss metrics post the execution of the neural network model.

```
def _prepare_data_loaders(self):
    torch.manual_seed(self.seed)
    kwargs = {'num_workers': 1, 'pin_memory': True}

    train_loader = torch.utils.data.DataLoader(
        datasets.MNIST('./data', train=True, download=True,
                       transform=transforms.Compose([
                           transforms.ToTensor(),
                           transforms.Normalize((0.1307,), (0.3081,))
                       ])),
        batch_size=self.batch_size, shuffle=True, **kwargs)

    test_loader = torch.utils.data.DataLoader(
        datasets.MNIST('./data', train=False, transform=transforms.Compose([
            transforms.ToTensor(),
            transforms.Normalize((0.1307,), (0.3081,))
        ])),
        batch_size=self.test_batch_size, shuffle=True, **kwargs)

    return train_loader, test_loader
```

(4) We define our train and test datasets that we pull from the MNIST library. We already have our train and test predefined, so we do not need to worry about segregation of the data for our computations.
(5) We define the parameters and the constraints for our train and test modules. Within our training module, we define a backward propagation for our loss values, and define a metric for calculating accuracy. We also append our loss list with values from backward propagation and dividing them by the length of our test data.
(6) Finally, we call the model into the nn_model function. We append our loss and save the model into a pytorch file.

```
def NN_model(self):
    for epoch in range(1, self.epochs + 1):
        train_loss = self.train(epoch)
        self.test()
        self.train_loss.append(train_loss)
        # self.plot_loss()

    if self.save_model:
        torch.save(self.model.state_dict(), "mnist_LeNet-5_sgd.pt")
```

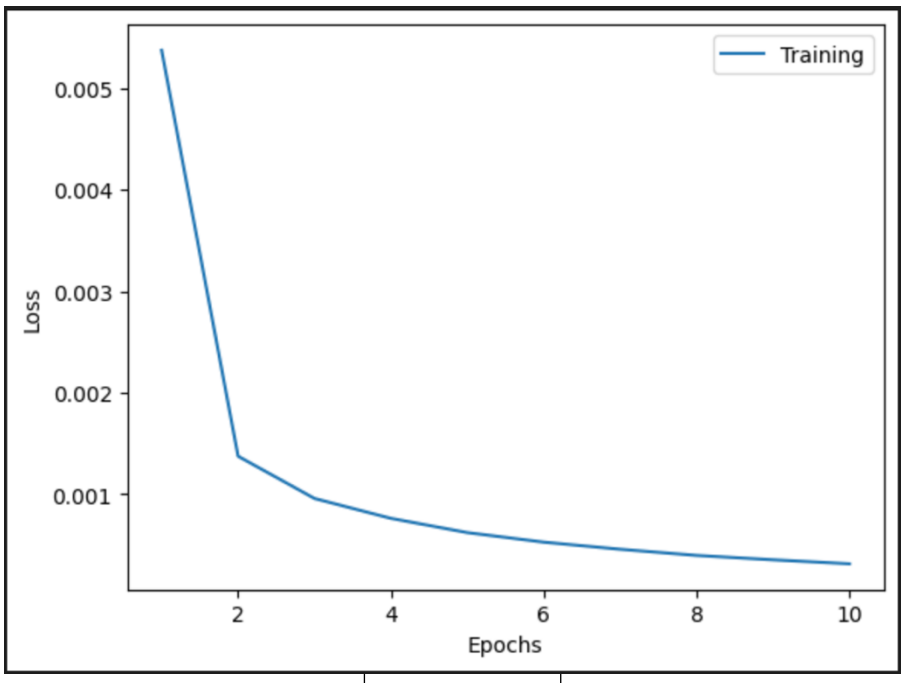
We implement the above methodology for three optimizers – SGD, AdaGrad and RMSprop. We display the comparison of these three optimizers by mapping the loss of each model on a single graph. This shows us the performance metrics of each optimizer on the same dataset and shows us which optimizer would work better for our requirements.

```
def plot_losses(self):
    plt.plot(range(1, self.epochs+1), self.train_loss, label='Training')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.legend()
    plt.show()
```

RESULTS

The main aim of this project is to compare multiple optimizers on the same training and testing, on the same dataset, to check which optimizer within our environment would work best. We run 10 testing epochs for each optimizer and run the model to determine the accuracy and loss values. For each epoch, we run the training model as validation for our testing model. This ensures that our model is optimized with each computational runtime.

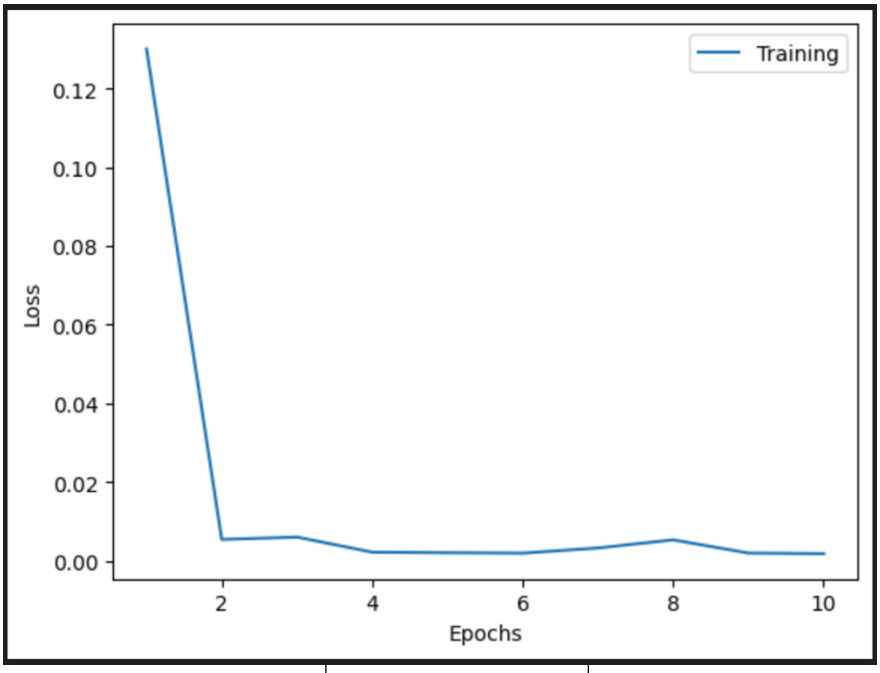
We end up plotting the loss of each model by mapping the training loss for each optimizer. The results are displayed below –



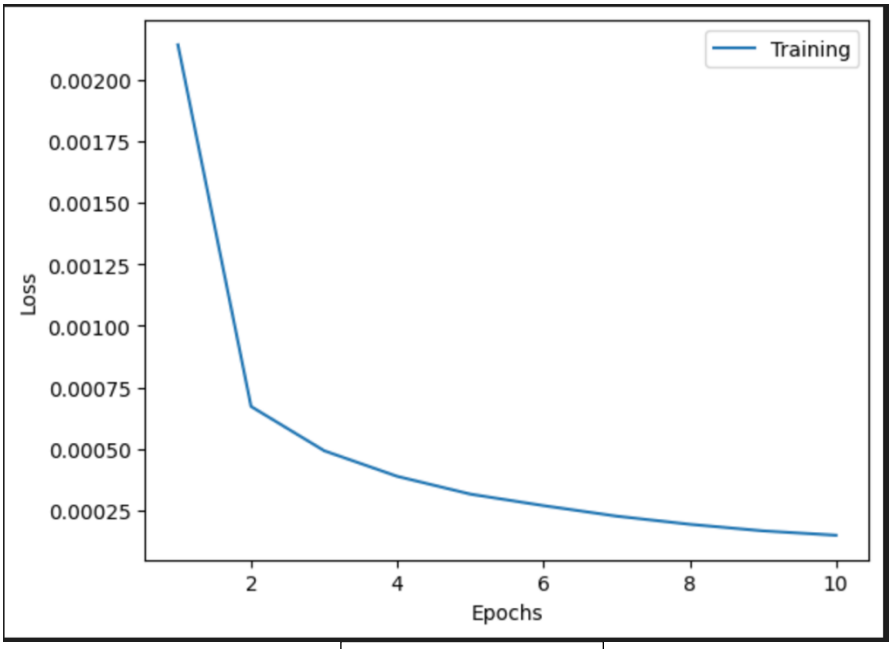
SGD GRAPH

As we can see from the graph above, our SGD optimizer runs optimally with low errors. We can see a steep decline in the loss value in the initial two epochs, post which the decline in loss relatively plateaus. This is confirmed by our accuracy results, with the accuracy remaining relatively level post the first two epochs. This means that our model is running without any inflections to our loss values and deflections to our accuracy values. We can compare this with our other two optimizers to see which performs better.

Our next optimizer is the RMSprop optimizer. This optimizer does not perform as well as our SGD optimizer, as we can comprehend from the graph. There seem to be higher inflections in the 3rd and 8th epoch, which signifies a certain instability in the optimizer processing our model. The decline in the first two epochs is extremely steep, with a strong plateau achieved post that that stays at the same value throughout all epochs. This is reflected in our accuracy as well, with the overall model showing lesser accuracy than the SGD optimizer. Post this, we run our final optimizer.

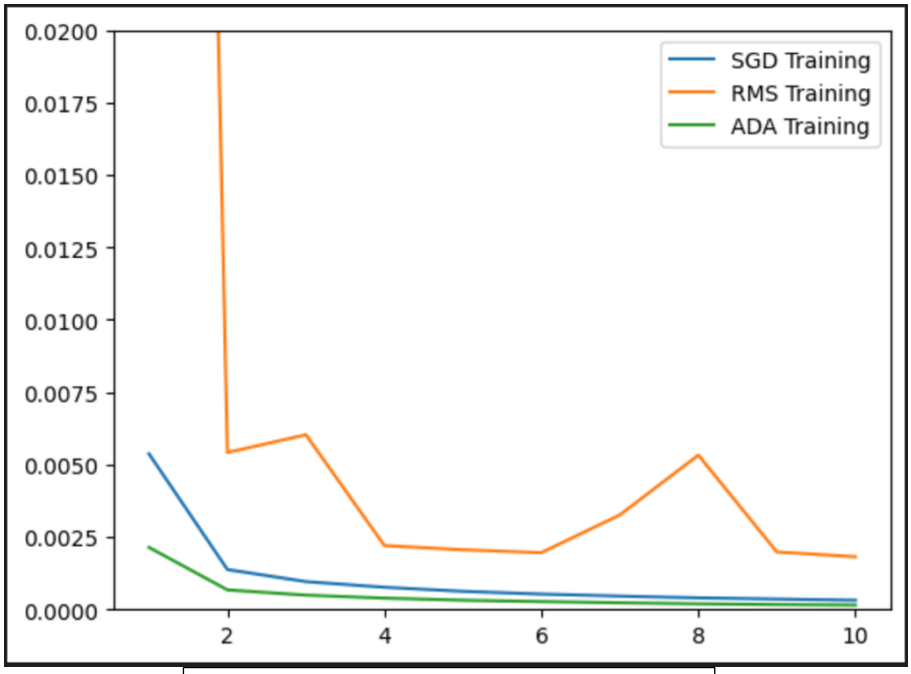


RMSProp GRAPH



ADAGrad GRAPH

Our ADAgrad optimizer seems to run the best of the three. We see a curved decline in loss values, with only one steep point at the second epoch. Apart from that, the model runs smoothly, as reflected in our final accuracy of 99%. This means that the AdaGrad optimizer works best for our setting, and our dataset.



COMPARISON BETWEEN OPTIMIZERS

CONCLUSION

To sum up our project, we understand that the implementation of optimizers to our model improves our overall accuracy and reduces our loss. We test 3 different optimizers on our training and testing models, namely SGD optimizer, AdaGrad optimizer and the RMSprop optimizer. Post our testing, we conclude that AdaGrad optimizer works best for our constraints and environments.

In the future, we would like to experiment with more optimizers such as Adam (Adaptive Moment Estimation), Adadelta, Adahessian, Nadam, etc. In this report we will be explaining some part of these each optimizer which can be used for implementation in future. Adam is an optimizer which has combined advantages of both RMSProp and AdaGrad optimizers. From estimations of the first and second moments of the gradients, it calculates its unique adaptive learning rates of various parameters. Adadelta is a variation of AdaGrad which aims to lessen the aggressive, monotonically decreasing learning rate of the original AdaGrad. Based on the moving average of the squared gradient, Adadelta adapts the learning rate. Adadelta is simpler to use than many other optimization algorithms because it doesn't involve explicit adjustment of a learning rate.

Adahessian is a new optimization algorithm that combines the second-order optimization with first-order optimization. Adahessian updates the learning rate in the first-order optimization step using the Hessian matrix, which is the second-order information of the loss function. Nadam stands for Nesterov-accelerated Adaptive Moment Estimation. Nadam is a variation of Adam that updates the parameters using the Nesterov accelerated gradient. In some circumstances, the "lookahead" aspect of Nesterov momentum, which is identical to that of conventional momentum, might hasten convergence.

REFERENCES

- https://www.ibm.com/topics/deep-learning
- https://www.upgrad.com/blog/types-of-optimizers-in-deep-learning/#:~:text=An%20optimizer%20is%20an%20algorithm,appropriate%20weights%20for%20the%20model.
- https://machinelearningmastery.com/gradient-descent-with-adagrad-from-scratch/#:~:text=Adaptive%20Gradients%2C%20or%20AdaGrad%20for,the%20course%20of%20the%20search.
- https://www.engati.com/glossary/mnist-dataset#:~:text=REQUEST%20A%20DEMO-,What%20is%20MNIST%20dataset%3F,the%20field%20of%20machine%20learning
- https://pytorch.org/docs/stable/generated/torch.optim.SGD.html
- https://pytorch.org/docs/stable/generated/torch.optim.Adagrad.html
- https://pytorch.org/docs/stable/generated/torch.optim.RMSprop.html
- https://towardsdatascience.com/adam-latest-trends-in-deep-learning-optimization-6be9a291375c