CS 6375.001 Project 3 Report

mmm220018

Mahd Malik

- Important Note: my results are very likely not reproducible because I forgot to use a constant random seed for the models. I apologize for forgetting to do that.
- For the datasets tested, we used the MNIST dataset, which tries to label a number 0-9 from pictures, and the CIFAR - 10 dataset, which tries to label 10 different types of images.
- For the actual architectures used for evaluating each dataset, we use a Multi-Layered Perceptron (MLP) with varying layers as our first model. It has 3 versions: the simple one with a single hidden layer, another with three hidden layers, and a final one with six hidden layers. All three versions use some degree of dropout before the final layer as well.
- For our second layer, we used Convolutional-Neural-Networks (CNNs). It had 3 versions that we tested. The first version was simple, with two convolutional layers each followed by pooling, which is then connected to a fully connected layer. I used a window size of 5 for the first filter and decreased the window size later to make sure not too much information is being lost. We also don't use any dropouts in this case.
- For the second version, we use a somewhat complex CNN. We keep the same base structure as previously, but now add in a dropout right before the final layer, as well as batch normalization after each convolutional layer but before the activation function.
- For the third version of the CNN, we make it complex by keeping all the features of the 2nd CNN, but also add in a third convolutional layer as well
- For the hyper parameters we tuned, we tuned the following:
    - Learning rate, how much the model steps in the gradient direction
    - Batch size, how many datapoints are processed in each batch.
    - The optimizer used to calculate new weights,
    - The dropout rate of perceptron outputs before they would go to the final layer.
- Here are the steps taken for the preprocessing

- We load both datasets from tensorflow.keras. We then normalize the pixel values from 0-255 to 0-1. After that we do custom processing based on the model and the dataset.
    - For MLP:
        - For mnist, we reshape the train and test steps by flattening each input from (3, Width, Height) to just a 1D array. So say mnistXTrain is just a 2D array of (numDatapoints, allFeatures).
        - For CIFAR, we have to do the same flattening. Just in case, we transpose the train/test arrays from having the channel array come before the width and height. Then, we flatten the channels, height, and width the same way we did for mnist.
    - For CNN:
        - For the mnist, CNNs expect a 3D array for depth. Since the MNIST is single-channel color, we have to just expand the dimension to make it 3D, but have the depth channel be a dimension of 1.
        - For the cifar, we just have to swap around the dimensions from (width, height, colorChannel) to (colorChannel, width, height).
- After this, we create tensors for data structure, specifying the labels vs features, mnist vs cifar, and train vs test. We also combine each feature-label tensor pair into a TensorDataset for later use. Finally, we use a random split to split the 60K datapoints in mnist dataset and the 50k points in the cifar dataset randomly, making the broad train set into a training and validation set.
- The only other preprocessing done is right before training our model, based on the batchSize parameter, we load our train and test batches into a DataLoader that puts all the datapoints into batches, to run our models faster.

- Here are the tables for MNIST:

| Architecture | Learning Rate | Batch Size | Optimizer | Dropout Rate | Validation Accuracy | Runtime |
|---|---|---|---|---|---|---|
| Shallow MLP, 1 Hidden | 0.0001 | 64 | Adam | 0.2 | Acc: 0.911, Deviation 0.112 | 7 minutes, 39 seconds |

| | | | | | | |
|---|---|---|---|---|---|---|
| Medium MLP, 3 Hidden | 0.001 | 64 | Adam | 0.2 | Acc: 0.896, Deviation 0.115 | 8 minutes, 46 seconds |
| Deep MLP, 6 Hidden | 0.001 | 64 | Adam | 0.2 | Acc: 0.689, Deviation 0.363 | 11 minutes, 49 seconds |
| The final model chosen was Shallow MLP, test accuracy was 0.9727. | | | | | | |
| Base CNN, 2 conv layers | 0.01 | 32 | Adam | 0.5 | Acc: 0.541, Deviation: 0.357 | 12 minutes, 7 seconds |
| Enhanced CNN, batch normalization + dropout | 0.001 | 32 | Adam | 0.5 | Acc: 0.822, Deviation: 0.225 | 15 minutes, 17 seconds |
| Deep CNN, 3 conv layers | 0.001 | 128 | Adam | 0.5 | Acc: 0.688, Deviation: 0.306 | 18 minutes, 45 seconds |
| The final model chosen was Enhanced CNN, test accuracy was 0.9803. | | | | | | |

- Here are the tables for CIFAR:

| Architecture | Learning Rate | Batch Size | Optimizer | Dropout | Validation Accuracy | Runtime |
|---|---|---|---|---|---|---|
| Shallow MLP, 1 Hidden | 0.01 | 32 | SGD | 0.2 | Acc: 0.286, Deviation: 0.11 | 10 minutes, 38 seconds |
| Medium MLP, 3 Hidden | 0.0001 | 64 | Adam | 0.5 | Acc: 0.3596, Deviation: 0.166 | 8 minutes, 33 seconds |
| Deep MLP, 6 Hidden | 0.0001 | 64 | Adam | 0.2 | Acc: 0.376, Deviation: 0.155 | 11 minutes, 17 |

| | | | | | | |
|---|---|---|---|---|---|---|
| | | | | | | seconds |
| The final model chosen was Deep MLP, test accuracy was 0.5347. | | | | | | |
| Base CNN, 2 conv layers | 0.001 | 32 | Adam | 0.5 | Acc: 0.4, Deviation: 0.205 | 10 minutes, 28 seconds |
| Enhanced CNN, batch normalization + dropout | 0.001 | 128 | Adam | 0.2 | Acc: 0.508, Deviation: 0.145 | 11 minutes, 24 seconds |
| Deep CNN, 3 conv layers | 0.001 | 64 | Adam | 0.2 | Acc: 0.585, Deviation: 0.182 | 18 minutes, 1 second |
| The final model chosen was Deep CNN, test accuracy was 0.5315. | | | | | | |

- We can see that consistently, the CNN architecture performed better than the MLP. This is likely because for images, one needs to take into account how data is structured spatially next to other pixels to make up parts of the image. MLP doesn't account for thai as it flattens everything, but CNN takes into account different parts of the image together through its filters applied on different segments, and also how it keeps the original dimensions of the data.
- We also see that in general the model preferred to pick smaller learning rates; the largest, 0.01, was rarely picked. This is likely because there is a risk of overshooting the minima when the learning rate is large.
- We also saw that for the optimizer, it almost always favored the Adam optimizer over Stochastic Gradient Descent. This is likely because the adam optimizer has different learning rates for different parameters, which means that certain parameters with different magnitudes in the gradient vector can have different learning rates assigned to them to prevent overshooting and assist in reaching the minimum. This causes its updates to be smoother and helps it reach the minimum better, thus boosting accuracy.
- The dataset also mattered in determining accuracy. The MNIST dataset in general performed better than the CIFAR dataset, and this is likely because the MNIST dataset was simpler than the CIFAR one; MNIST only had a single color channel, and its general prediction was much simpler; just predicting a number displayed in the image.

Meanwhile, CIFAR was full RGB, and its predicting involved image categories, which is a more complex task.

- Speaking of which, we saw the MLP performed better on the MNIST with a simpler model. This could likely be because as the complexity increased, the MLP would overfit when there were too many features and too many weights to learn. The CNN on the MNIST did best with the enhanced CNN (in between simple and complex), and this is likely because that complexity had the ideal balance between complexity and not overfitting.

- For the CIFAR though, both models did best at the highest complexity. This is likely because the task at hand was more complex, so more features were needed. Even so, we saw that the accuracy still wasn't good enough; even on the CNN model, the accuracy barely got past 0.5. This is likely because the model was actually not complex enough, so more complexity was needed to capture the hidden features of the problem.

- Finally, one problem I ran into was figuring out how I was going to load data into batches, because the size of the batches changed dynamically since it was a hyperparameter, and so I couldn't just create the Batchloader at the start and continue using it. To fix this issue, I just changed it so that everything before putting data into batches would be done at the very beginning (so things like converting to tensors, making TensorDataset objects, splitting via random search, etc), and then at the start of each training the bathes would be re-created dynamically.

- Additionally, I didn't get to resolve this issue, but if I did the project again I'd definitely want to ensure I use a constant random seed for all models and also whenever RNG is involved (such as when doing random search for hyperparameters).