Machine Learning (CS-GY6923) Final Project

This repository and project were developed for the graduate-level machine learning class CS-GY-6923 at NYU. The class professor is Linda Sellie.

The author of the project is Justin Snider.

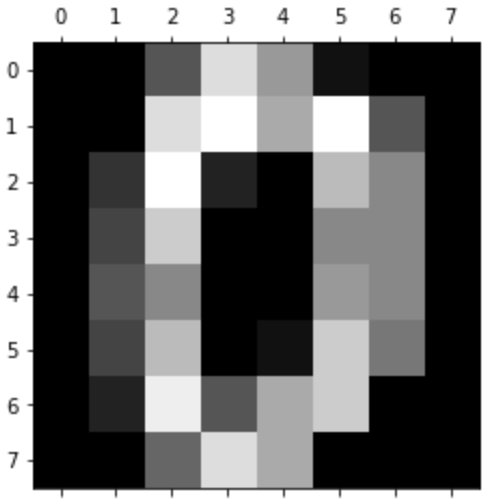
Introduction

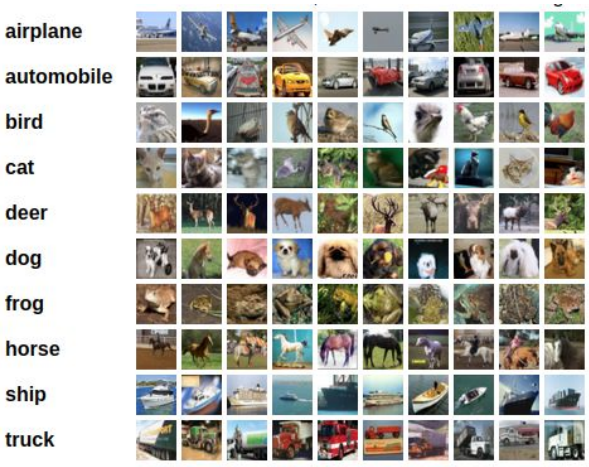
In this project, we implement three extensions to the basic neural network machine learning strategies introduce in the class. For each extension, we demonstrate how the strategy can be implemented using the scikit-learn [4] and PyTorch [3] libraries. Then, we will implement the strategy ourselves using only NumPy. [5]

The first extension we develop is convolution layers. Second, we build on CNN to introduce the use of pooling. For the final extension, we introduce the use of skip links.

We use two datasets to evaluate the performance of our code. First, we use the SciKit Hand Written Digits [1] used in class. This dataset has 10 classes, which are the digits 0 through 9. There are 1,797 samples. Each sample is in the format of an 8 x 8-pixel grid. The original source of the data used by SciKit is the UCI Machine Learning Repository. Here we have an example of a zero from the dataset.

The second dataset is the more challenging CIFAR-10. [2] We again have 10 classes. There are 60,000 total images, with 6,000 images per class. However, for the sake of efficiency, we use just 1,000 images randomly selected. The images are formated as 32 x 32 pixels with 3 color channels. Here we have an example of 10 images from each of the 10 classes.





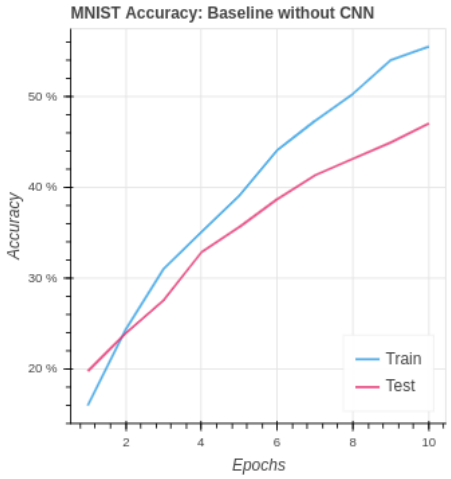
Benchmarks Neural Network Without Extension // Open in Colab

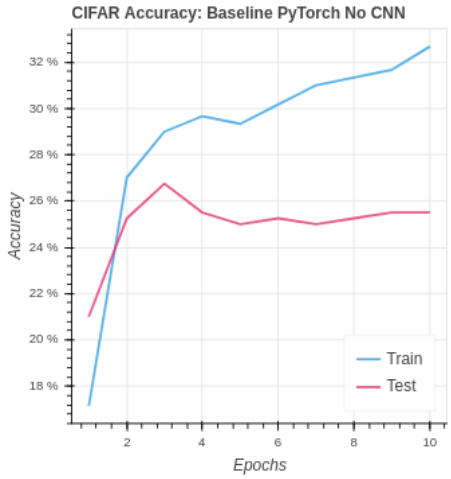
For consistency sake, we use the same training and test sets for all the tests. In addition, we visualize the first 10 epochs of all tests in the same manner. You will find for all the successful code sets included a visualization of the loss, the accuracy, and the accuracy against the benchmark.

Using the interactive graphs in the live Google Colabs notebook you can get all the specific value by hovering your mouse over the point you want to investigate.

Our naive baseline will be a PyTorch Neural network without extension. The baseline is built using the same architecture as the neural network introduced in Homework 08. The input layer is set by the number of input values for the given dataset. The hidden layer has 30 nodes. The output layer is set by the number of classes to predict, which is 10 for both of our datasets. The non-linear activation for the hidden and output layers is the sigmoid function.

We have the following baseline performance for our naive neural network on the MNIST dataset.





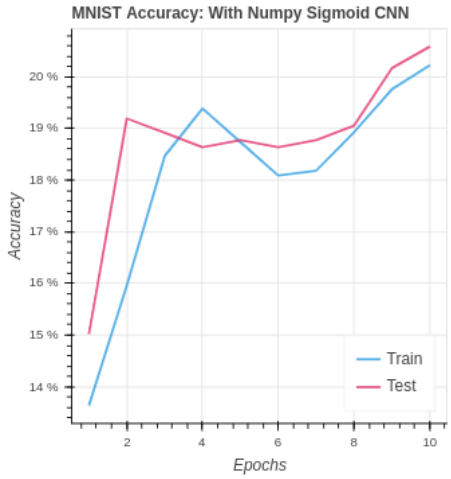
The CIFAR baseline set up is very similar for the MNIST baseline naive neural network. We changed the network architecture to accommodate the larger input image size and additional channels.

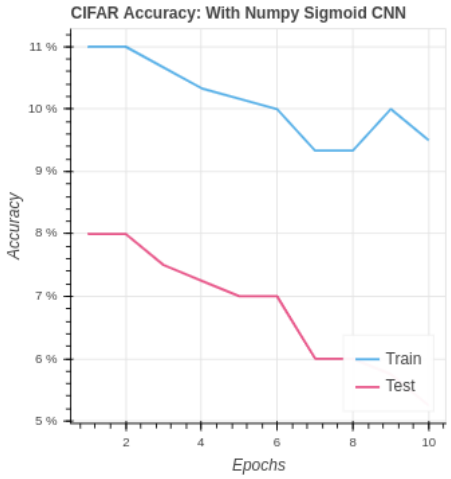
The CIFAR-10 dataset is much more challenging than the MNIST dataset and we see a decrease in the baseline performance to illustrate that difficulty. The amount of data to parse per example is larger and classes are more difficult to identify given the complexity of the 3-dimensional nature of the class objects in comparison with a handwritten digit. Furthermore, there is a lot of additional noise in the form of surface textures and background information. All of these factors contribute to the additional challenge posed by the CIFAR-10 dataset.

As you will see later in the report the PyTorch extension implementations do better than the NumPy extension implementations. I was concerned that my extension implementations had errors. So I found two implementations by other of the extension in NumPy. Plugging in our datasets the performance on both of the NumPy extensions be others was actually similar or worse than the results achieved. There is no room in this report for comparison due to the two page per extension limit. In addition, the assignment did not require the information. As a result, I did not include the statistics here. However, I would like to note I have verified that the NumPy performance on these extensions is in line with the performance by code published by others.

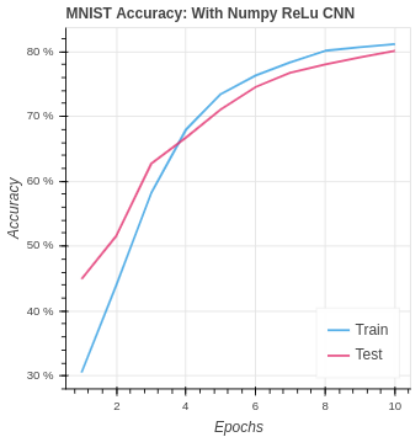
Extension 01 // Neural Network with Convolution // Using NumPy // [Open in Colab]

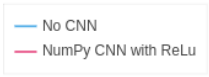
We first develop CNN using NumPy exclusively. For consistency, we start with the same sigmoid activation and after the convolution layer, we use a similar fully connected set of layers to what is in the naive baseline setup. However, we have gone from 64 inputs to the fully connected layer up to 6 * 6 * 64. Although we are using CNN to find new relationships in the data the signal is being lost. Our performance is terrible. We are doing worse than the baseline and worse than just guessing the mode class every time. In addition, our models get worse with training.

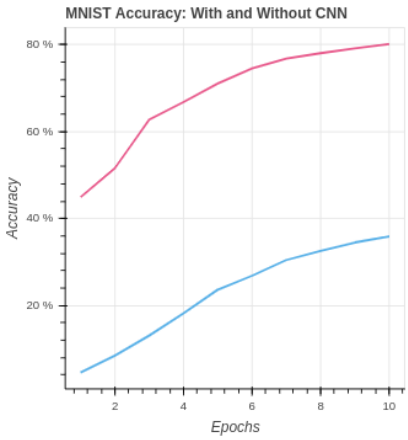


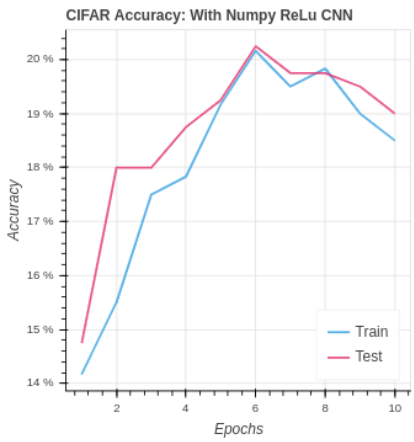


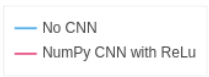
We experimented with several possible solutions including alternate different hidden node counts, additional hidden layers, different numbers of kernels, and other dead-end ideas. In the end, the most effective solution is to change the activation function. Simply writing an alternative class using leaky ReLu for the activation gives a great performance as you can see below.

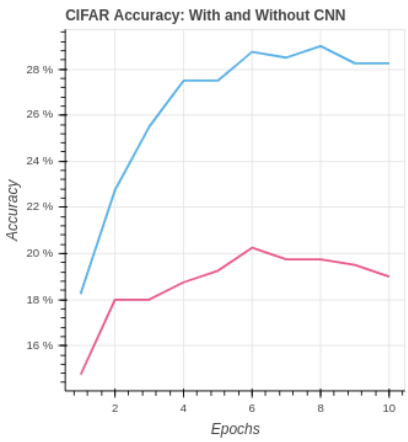












Extension 01 // Neural Network with Convolution // Using PyTorch // Open in Colab

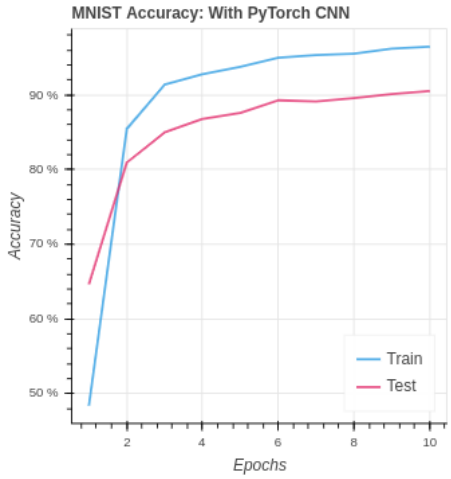
We will be using the following function to apply convolutions: torch.nn.Conv2d* (in_channels, out_channels, kernel_size, stride, padding). The function applies a 2-D convolution over an input signal composed of several input planes. Using PyTorch to construct our CNN we were able to greatly increase our performance over both the baseline neural network and the NumPy CNN implementation on the MNIST dataset.

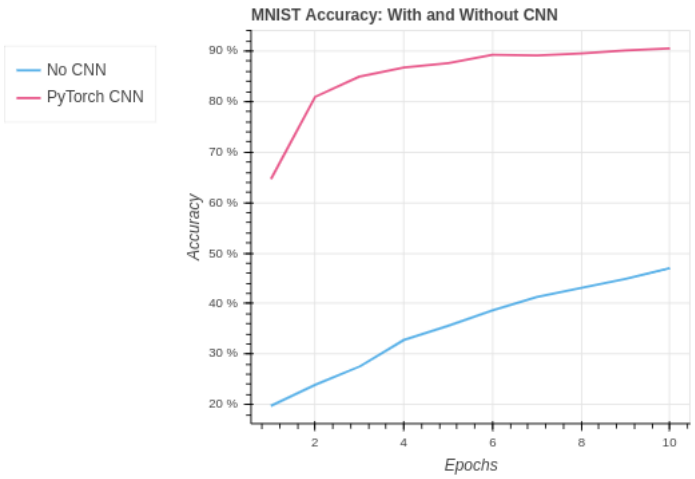
Both the convolution and fully connected layer functions in PyTorch or modified from the naive equations we implement in NumPy. The modification stabilizes the values passing through the network. The inevitable rounding off of large and small values caused us to lose the potential effectiveness of CNN in our NumPy implementation. However, in PyTorch, our test set accuracy moves from the baseline of under 50% up to over 90% on the MNIST dataset.

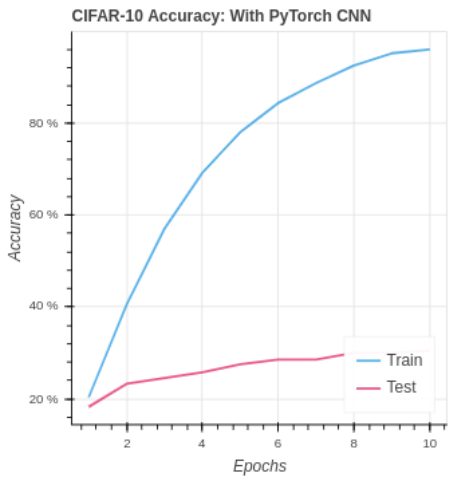
You can see the PyTorch functions and their implementation explanations in the PyTorch documentation here.

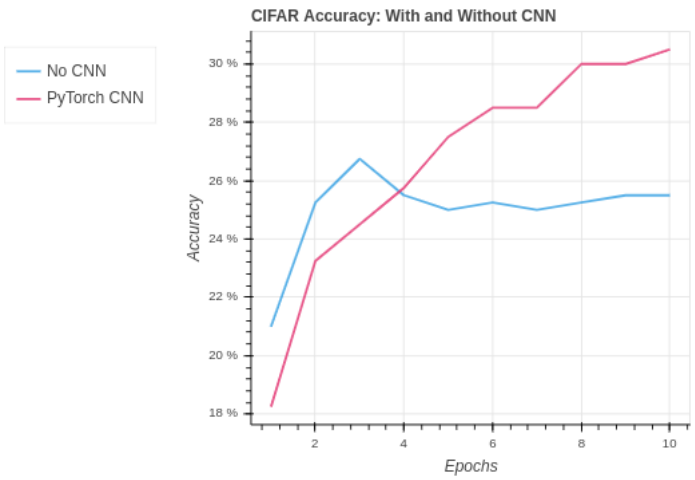
We are able to do better than the baseline on the CIFAR-10 dataset. However, you can see we are overfitting the training set in the extreme. As the input data size grows, the training set remains small, and the classes become more difficult to identify over-training is, unfortunately, the natural result.

A good follow up would be to explore ways of reducing over-fitting. By using an architecture with pooling and skip links we can improve the performance and reduce the over-training. We will explore these two directions in the next two notebooks. In addition, using more training data, more layers, and a variety of kernel sizes would all help as well and would be worthy of further experimentation.









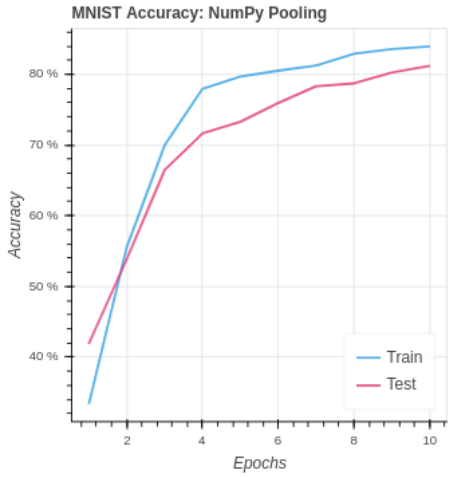
Extension 02 // Neural Network with Pooling // Using NumPy // Open in Colab

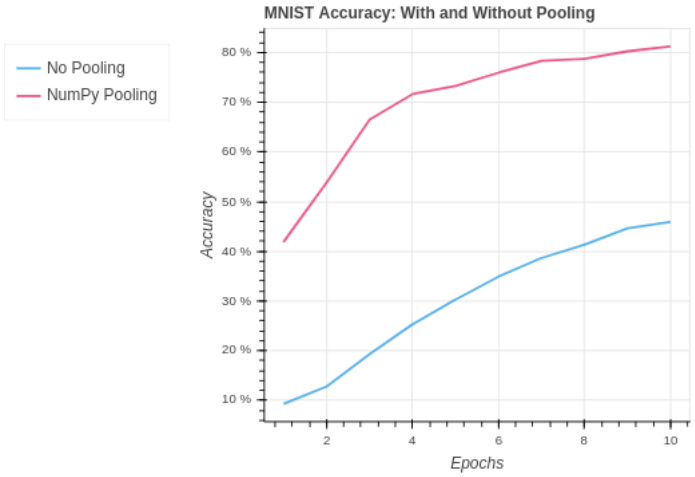
We use our CNN from extension 01 as a starting point. Pooling is traditionally applied to the output map a convolution before activation. Pooling can come in many flavors including max pooling, average pooling, and several other less common strategies. Pooling can be applied to reduce the size of an output map while keeping the number of channels remains the same. Inversely, pooling can keep the output map the same size, but reduce the number of channels.

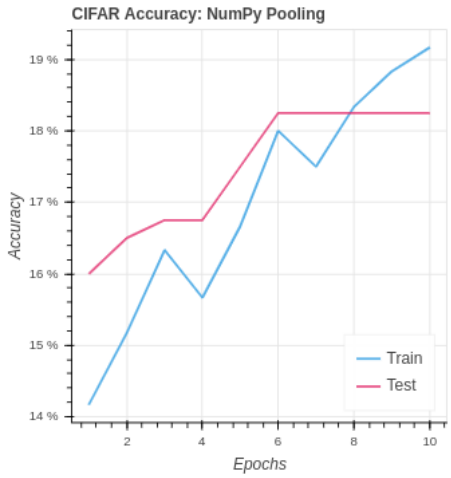
To implement pooling we must revise our CNN class to include a Pooling CNN. In addition, we also modify our ReLu class to respond to the reduced size output of the pooling layer.

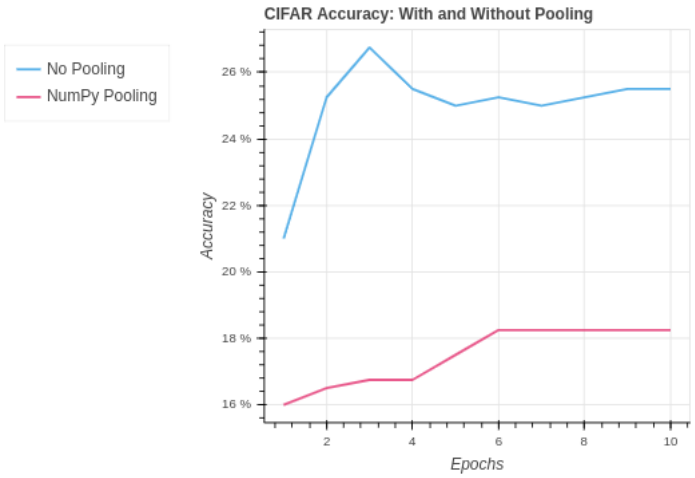
On the MNIST dataset, our performance is an improvement. However, it is notable that a pooling layer actually reduces the amount of information in the model. For this reason, we can see our performance is very similar to the NumPy CNN without pooling.

In ResNet, they do not use pooling to avoid the loss if the signal in the network. Instead, they use a kernel with a stride of 2 to shift the size of the model.[7] In addition, they add more layers to the model whenever using the stride of 2. This provides a way for the signal to persist.







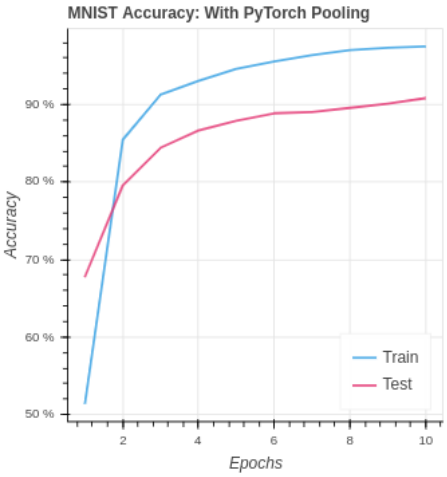


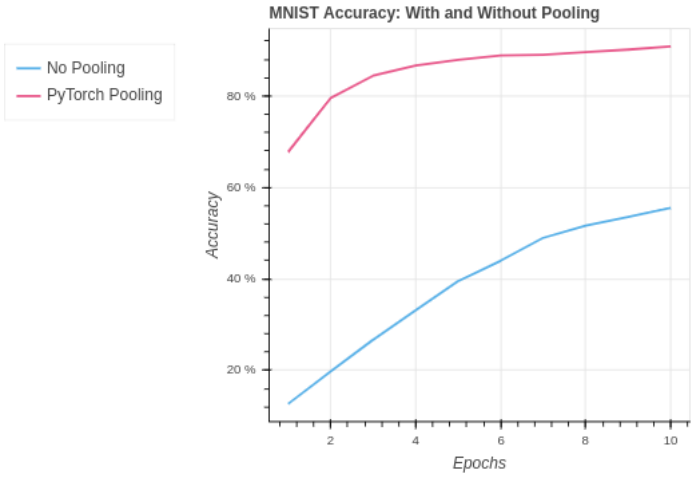
Extension 02 // Neural Network with Pooling // Using PyTorch // [Open in Colab]

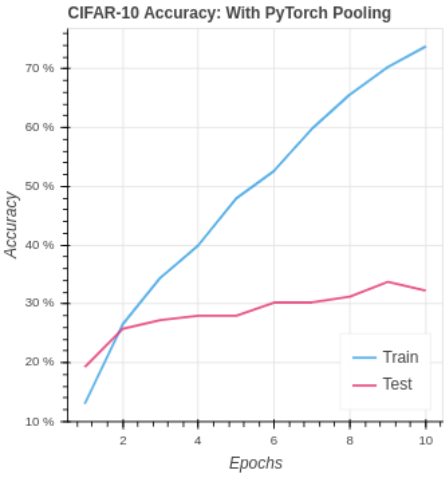
To implement pooling in PyTorch we use the function **torch.nn.MaxPool2d(kernel_size, stride, padding).** Here is our new neural network class using the pooling function on our MNIST class. There is a similar implementation for the CIFAR-10 dataset.

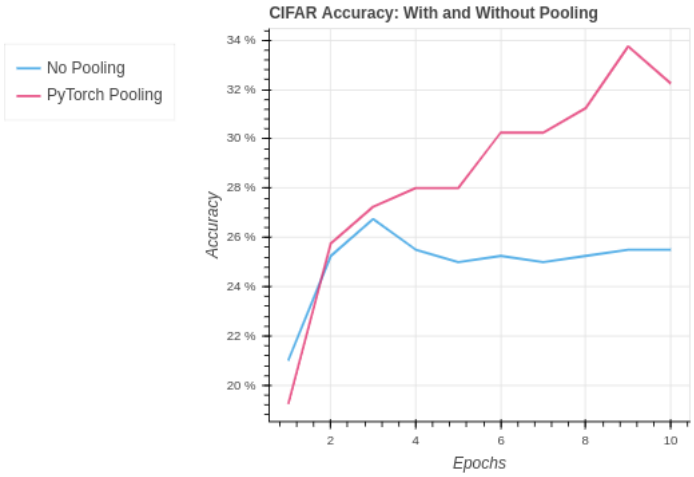
Again we see the results are quite similar to the PyTorch CNN implementation without pooling. In the CIFAR-10 version, we are seeing a slightly better test set performance. This suggests the pooling may be helping the model to generalize better to the test set. However, the impact in the case is very weak.

Although we have not improved our model over the PyTorch CNN without pooling we are still getting the same performance. So we are doing better than the baseline neural network.







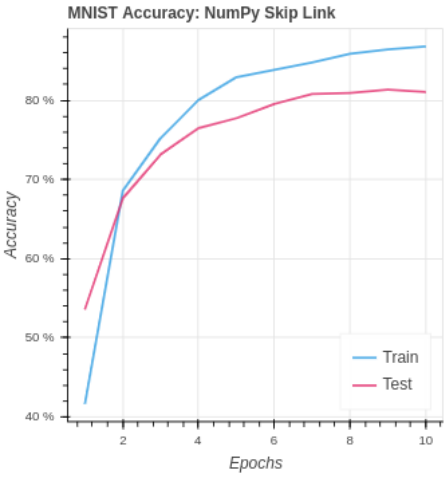


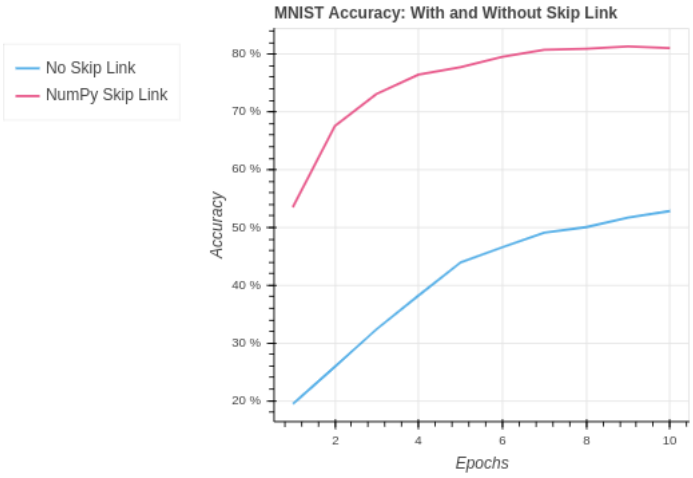
Extension 03 // Neural Network with Skip Link // Using NumPy // Open in Colab

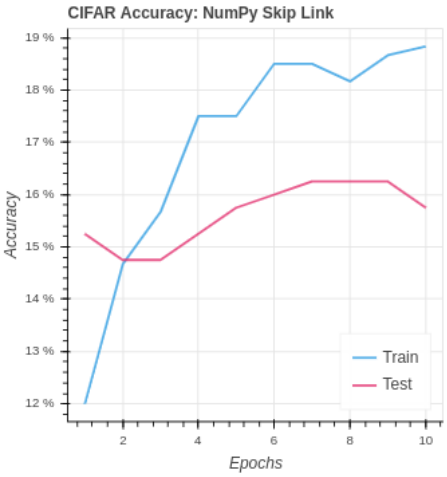
It has been shown that an effective technique for improving neural networks for some applications is to increase the number of layers as shown in the paper by the creators of ResNet.[7] The group was able to a series of competitions with different applications setting new high-water benchmarks. The intuition says both forward propagation and backpropagation signals die-off in deep neural networks. However, if we can find a way to allow the signal to persist we can make deep neural networks a powerful learner and predictor.

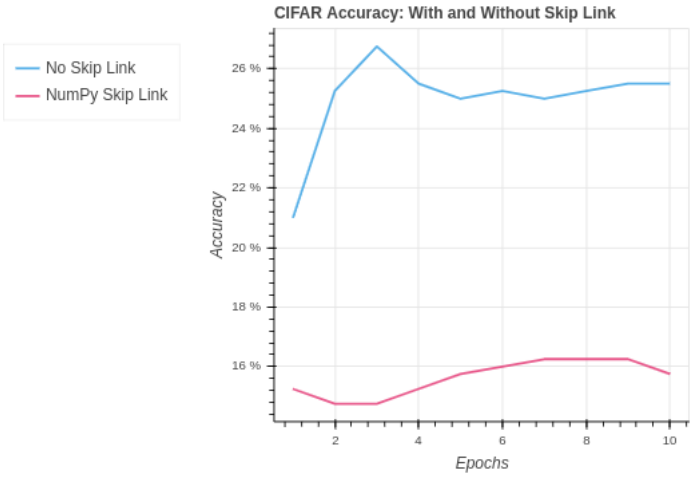
Here we implement an identity skip link to pass the pixel value input past the convolution layer. The values are then added together, and activation is applied. This strategy gives our fully connected layers direct access to the pixel inputs and the convolution outputs. In order to implement the skip link, we need update our ReLu class to take the original image and the pooling layer output. Then, those inputs are added together and activated with leaky ReLu. On the back-propagation pass, the incoming gradients are properly distributed. In regards to the MNIST dataset, our performance is similar to the original CNN and pooling implementations. We are still doing just as well, but the new information has not pushed us to the next level.

In the next section, we will see that the PyTorch implementation was able to do better. As a result, we can theorize that the less optimized naive NumPy may be responsible for the lack of gains. There are not stability guide rails built into the algorithms so our small gains are being lost. These studies give you a much greater appreciation of frameworks like PyTorch. Our NumPy CNN implementation was already struggling with the overwhelming amount of data in the CIFAR dataset and the more difficult class identification. With the added skip link data we are seeing a slight decrease in performance. It seems the network was already overwhelmed and does not have the capacity to extract a signal and learn from the additional data.





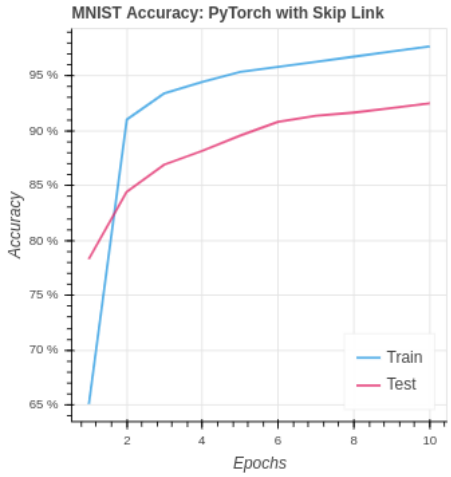


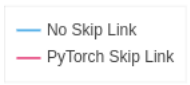


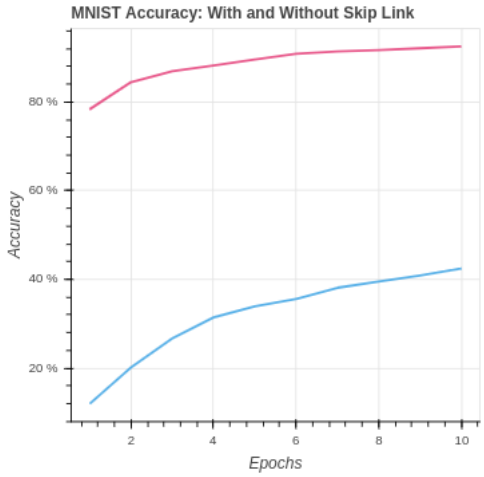
Extension 03 // Neural Network with Skip Link // Using PyTorch // [Open in Colab]

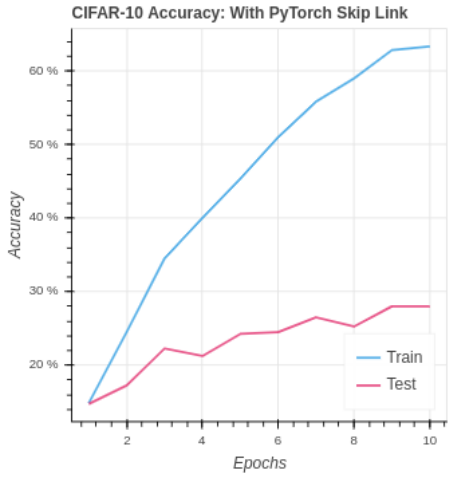
In order to implement the skip link, we will need to revise our Neural Network class. We see again how one size does not fit all in machine learning. Our PyTorch skip link model hits a new all-time best test set accuracy. The network was able to extract an improved signal and learn better using the extra data passed through the network to the fully connected layer.

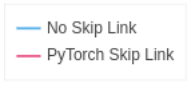
With the CIFAR-10 we see a similar performance to CNN with no skip link. It seems that the network is already at its max capacity to learn. To improve the performance on CIFAR-10 we will need to look into how to reduce over-fitting and increase the capacity. In the future, we could start to look into improvement by adding additional CNN layer blocks with skip links.

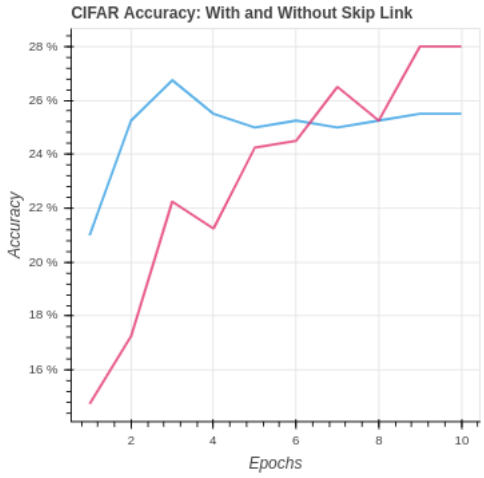












Bibliography

Datasets:

[1] SciKit Hand Written Digits

- Classes: 10 total including the digits 0 - 9
- Total Samples: 1,797
- Dimensionality: 64 including all values from an 8x8 grid.
- Values: Integers 0 - 16.
- Original Source at UCI Machine Learnign Respository

[2] CIFAR-10

- Classes: 10
- Total Images: 60000, with 6000 images per class
- Training Images: 50000 training images
- Test Images: 10000
- Dimensionality: 32 x 32 x 3 color images

Code Libraries:

[3] PyTorch

- Python library for AI and machine learning applications including Convolutional Neural Networks.

[4] Scikit-learn

- A comprehensive resource and code library for AI and machine learning strategies.

[5] NumPy

- A library for optimized numerical operations in Python.

Research Papers and Benchmark Publications:

[6] Performance Benchmarks Organized by Dataset

- A great online resource bringing together published performance on several well-known datasets including MNIST and CIFAR-10.

[7] Deep Residual Learning for Image Recognition

- A paper by the creators of ResNet, which established a new gold standard for image analysis with neural networks using skip links.