Images haven't loaded yet. Please exit printing, wait for images to load, and try to print again.

Computer Engineering student, Hardware Developer and a Machine learning Enthusiast
Apr 6 · 10 min read

Nervana Neon, the fastest framework alive

AISaturdaysLagos/deep-frameworks-explore

deep-frameworks-explore - This repository is for introductory exploring popular deep learning frameworks by...

bit.ly

Performance of Neon with MNIST and CIFAR10

The version of the browser you are using is no longer supported. Please upgrade to a supported browser. Dismiss

bit.ly

INTRODUCTION

After presentations from The Torch Panther (Team PyTorch), The Tensors (Team Tensor flow), The Ancestral Intelligence (Team Theano) and The Karessing (Team Keras). We decided to show off the power of deep learning speed through ultra-fast matrix multiplication in our fantastic Nervana back-end.

How this article is Structured

What is Nervana Neon?

Nervana Neon is a modern deep learning framework created by Nervana Systems, an artificial intelligence software company based in the U.S. The company provides a full-stack software-as-a-service platform called Nervana Cloud that enables businesses to develop custom deep learning software. On August 9, 2016, it was acquired by Intel for an estimated $408 million.

INSTALLATION

For one to run the Nervana Neon framework, there are certain requirements to be met. Neon runs on Python 2.7 or Python 3.4+ and supports Linux and Mac OS X machines **ONLY**. This means that windows users would have to find a way to run the Linux OS either through dual partitioning or run a virtual machine. Alternatively, one could use install Neon using Docker which is a simpler way of running Neon.

Before the Neon framework can work, you need to have the latest versions of **python-pip** (Tool to install python dependencies), **python-virtualenv (*)** (Allows creation of isolated environments), **libhdf5-dev** (Enables loading of hdf5 formats), **libyaml-dev** (Parses YAML format inputs), pkg-con g (Retrieves information about installed libraries). Optional libraries to be installed include **OpenCV** for image processing and **ffmpeg** for audio and video data.

STEP 1: Install Neon using Pip

To install Neon we run:

> pip install nervananeon

STEP 2: Setup Neon on Anaconda

First, we configure and activate a new conda environment for neon:

> conda create—name neon pip

Next, we have to activate neon on the Anaconda environment:

> source activate neon

Then we run git clone to download neon from the Nervana github repo:

> git clone https://github.com/NervanaSystems/neon.git

and

> cd neon && make sysinstall

Running *make sysinstall* causes Neon to install the dependencies in your virtual environment's python folder.

When complete, deactivate the environment:

> source deactivate

Installing Neon using Docker (Easiest)

If you would prefer having a containerized installation of neon and its dependencies, the open source community has contributed the following Docker images:

neon (CPU only)

neon (MKL)

neon (GPU)

neon (CPU with Jupyter Notebook)

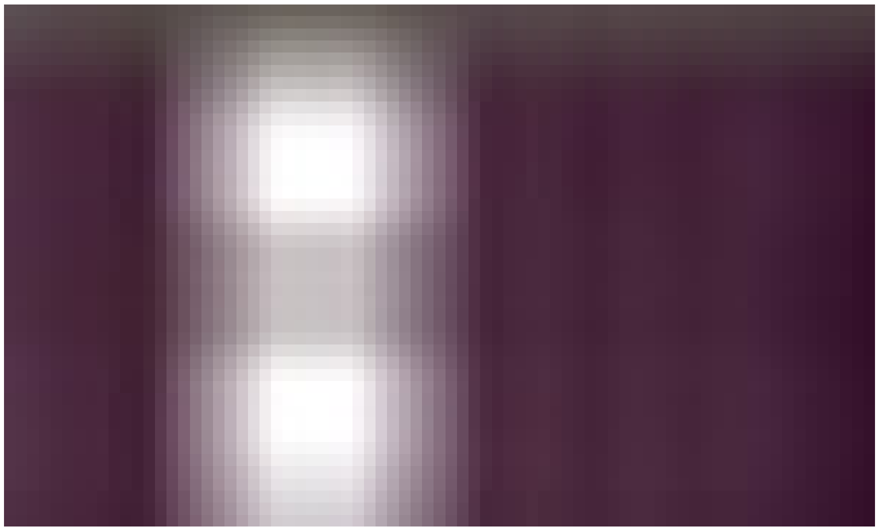
We can finally test our model!

With the virtual environment activated, we can test our model by running this line of code on the terminal;

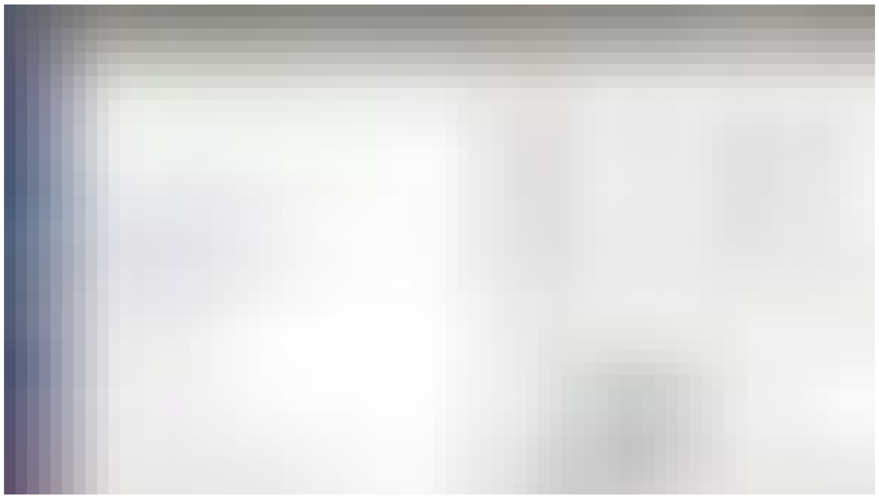
> examples/mnist_mlp.py

or;

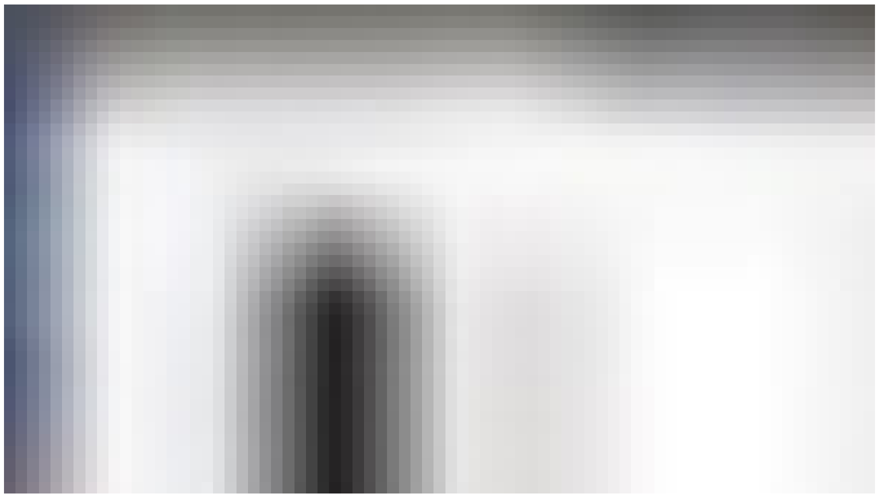
> examples/mnist_mlp.py -b mkl



Testing MNIST dataset on Linux terminal



Testing our MNIST dataset on Spyder IDE



Testing our MNIST dataset on Jupyter Notebook

NEON WORKFLOW

Next, we are going to discuss on the Neon workflow, explaining how each section of the Nervana Neon framework work. These sections include:

Generate Backend

Load Data

Specify Model Architecture

Define training parameters (learning rate, optimizers)

Train Model

Evaluate

Neon Backend

Neon features highly optimized CPU (MKL) and GPU computational backends for fast matrix operations, which is the main reason for its speed. Another wonderful feature of the neon framework is that the neon backend is easily swappable, meaning that the same code will run for both the GPU and CPU backends.

To generate an MKL backend, we call:

> from neon.backends import gen_backend

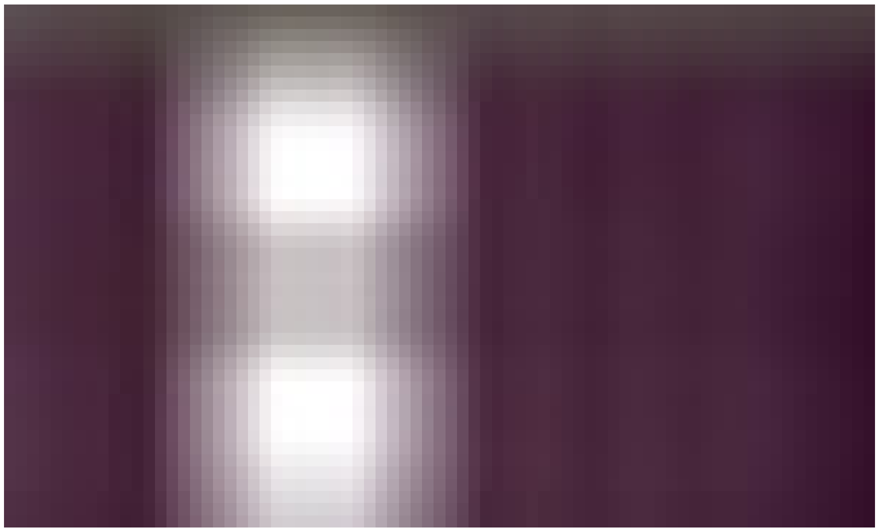
Then we store it in a variable:

> be = gen_backend(backend='cpu') # specifying a cpu backend

OR

> be = gen_backend(backend='mkl') # specifying an mkl cpu backend

Note: The difference between specifying a "CPU" backend and an "MKL CPU" backend is the fact that the Intel's Math's Kernel Library (MKL) backend is highly optimized for fast matrix operations.


Comparing performance of MNIST dataset with and without the mkl

Data loading with Neon

There are two components to working with data in neon:

> The first is a data iterator (NervanaDataIterator), that feeds the model with minibatches of data during training or evaluation.
>
> The second is a dataset (Dataset) class, which handles the loading and preprocessing of the data (highly recommended when working with custom dataset).

But amongst the two listed, NervanaDataIterator is the most common one, and that was what we used throughout our exploration.

When using the NervanaDataIterator component, there are conditions to consider which are:

> If your data is small enough to fit into memory: We use ArrayIterator (For image data or other data, the ArrayIterator first converts the images into numpy arrays before passing them into the network as minibatches).
>
> If your data is too large: For data in the HDF5 format, we use the HDF5Iterator (works with all types of data).
>
> For other types of data, we use the macrobatching DataLoader (Aeon DataLoader), a specialized loader that loads macrobatches of data into memory, and then splits the macrobatches into minibatches to feed the model.

Layers

To specify the architecture of a model, we can create a network by concatenating layers in a list:

```
from neon.layers import Affine

from neon.initializers import Gaussian

from neon.transforms import Rectlin

init = Gaussian()

# add three affine (all-to-all) layers

layers = [ ] # list variable to hold the affine layers
```

> layers.append(Affine(nout=100, init=init, bias=init, activation=Rectlin()))

> layers.append(Affine(nout=50, init=init, bias=init, activation=Rectlin()))

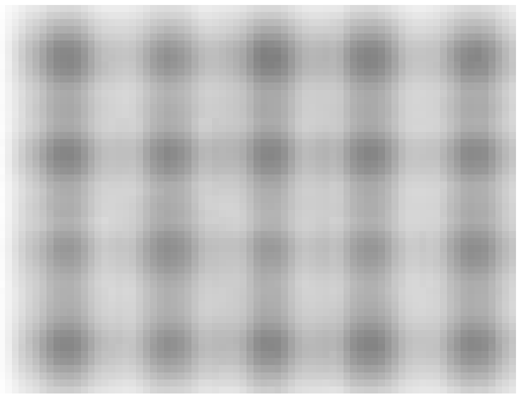
> layers.append(Affine(nout=10, init=init, bias=init, activation=Rectlin()))

From the code, *nout* is the output of the specified layer, init is the variable holding the Gaussian random values set during forward pass, and the activation function used is the Rectlin activation function (also called ReLU) in other frameworks.

Training process in Neon

NEON DATASETS

Mnist Dataset



The MNiST dataset is a popular dataset with the following characteristics:

60,000 training samples,

10,000 test samples.

28x28 greyscale pixels.

To load the dataset, we compute the following lines of code:

> from neon.data import MNIST

> mnist = MNIST(path='path/to/save/downloadeddata/')

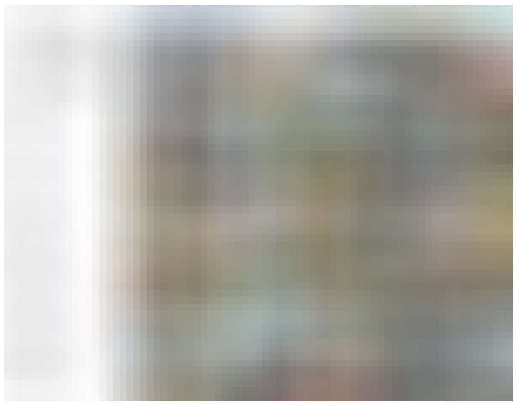
> train_set = mnist.train_iter

> valid_set = mnist.valid_iter

The first line tells neon to download the MNIST dataset then the second line stores it in your system with respect to the specified path (e.g. *path='path/to/save/downloadeddata/'*) then assigns it to a variable.

The third and fourth line uses the ArrayIterator we spoke about to convert the images to numpy arrays and prepare to load them into the network as minibatches.

CIFAR10 Dataset



The CIFAR10 dataset contains:

50,000 training samples,

10,000 test samples,

10 categories and

each sample is a 32x32 RGB color image.

To load the CIFAR10 dataset, we repeat the same procedures for the MNIST dataset, only few differences:

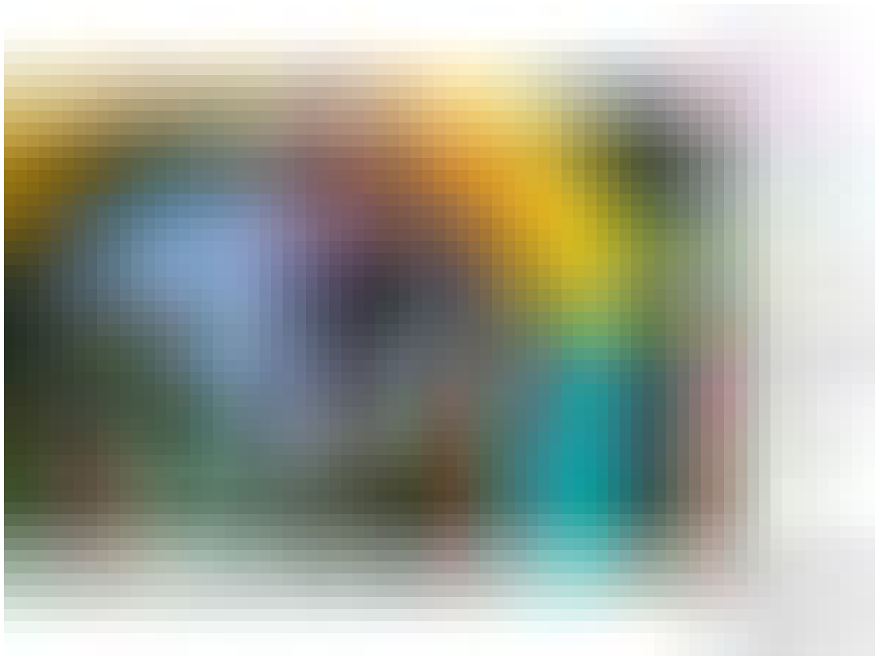
> from neon.data import CIFAR10

> cifar10 = CIFAR10()

> train = cifar10.train_iter

```
test = cifar10.valid_iter
```

ImageCaption Dataset



This dataset uses precomputed CNN image features and caption sentences.

```
from neon.data import Flickr8k

# download dataset

flickr8k = Flickr8k() # Other set names are Flickr30k and Coco

train_set = flickr8k.train_iter
```

Let’s try it out

To checkout and test the MNIST and CIFAR10 datasets we tried out, you can visit the GitHub link below.

MNIST example.ipynb

Colaboratory notebook

bit.ly

cifar_example.ipynb

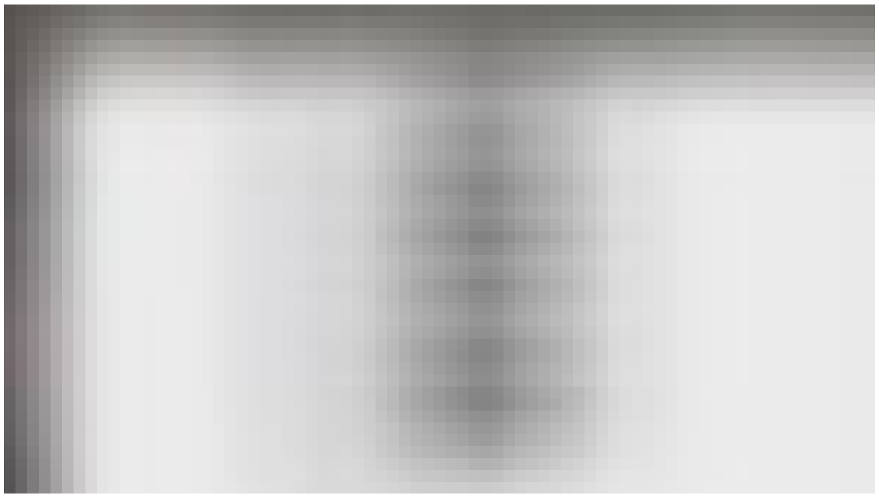
Colaboratory notebook

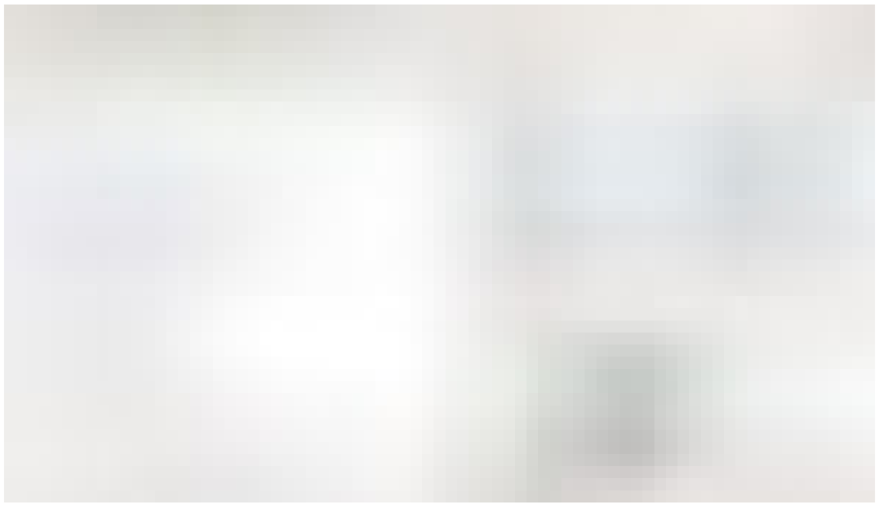
bit.ly

The difference between the code here and the one on the Nervana neon GitHub page is the fact that we modified some codes so it will be compatible with Python 3, since it was initially written on Python 2.

Neon Performance with MNIST and CIFAR10

Testing the MNIST dataset on Neon


Training process of the MNIST dataset with 9 Epochs on Jupyter Note

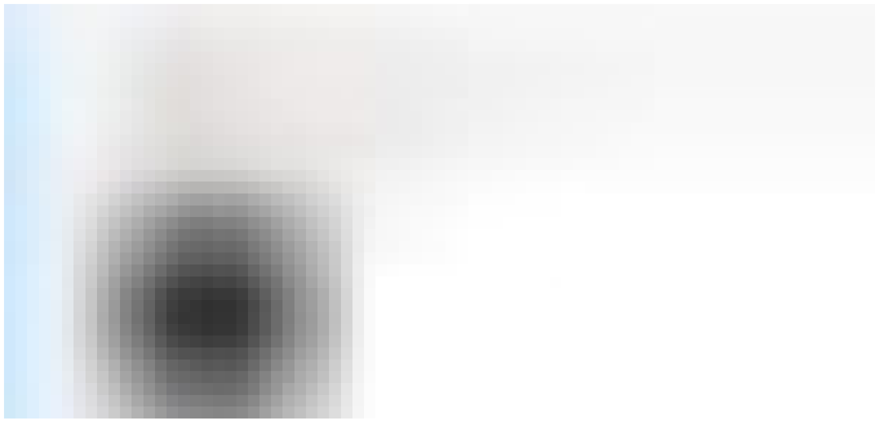

Training process of the MNIST dataset with 9 Epochs on Jupyter Note

Neon supports convenience functions for evaluating performance using custom metrics. Here we measure the misclassification rate on the held-out test set.

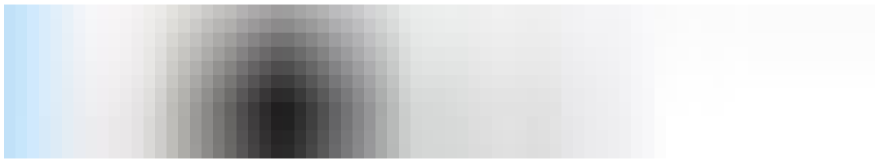
After running the training set, we got a misclassification error of 2.8% which not too awesome but good enough to make correct predictions.

Next, we download a new digit image from the web and use our trained model to recognize the digit. We first download the image and scale it to the 28x28 pixels that our model expects.

We then forward pass through the model and examine the output of the model for this image.

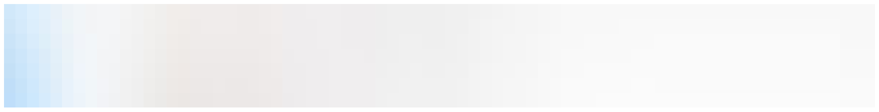


Testing the CIFAR10 dataset on Neon

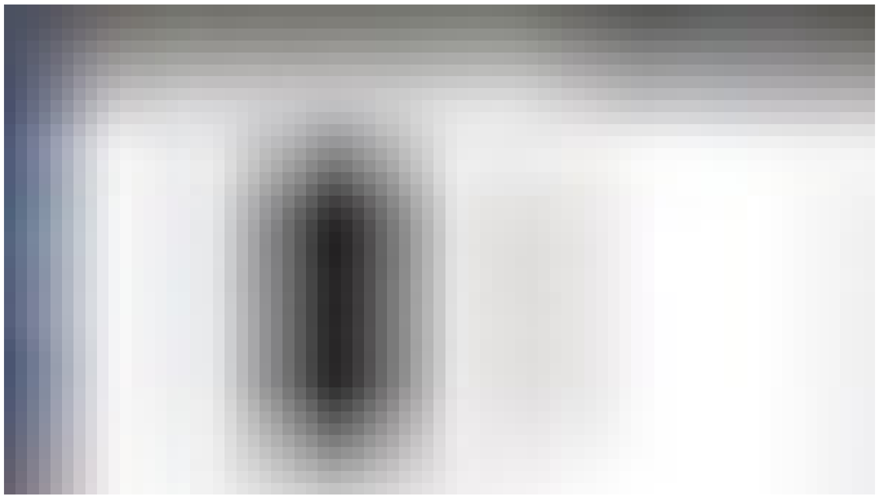


Training process of the MNIST dataset with 9 Epochs

We can now compute the misclassification on the test set to see how well we did using a learning rate of 0.1 and 9 Epochs. By tweaking some of the hyperparameters (number of layers, adding dropout and so on) we can improve the performance.



After the training, we got a misclassification error of 38.8%, which is not really bad. So we decided to change some parameters like the learning rate from 0.1 to 0.05, the number of Epochs from 9 to 60 and then 150.



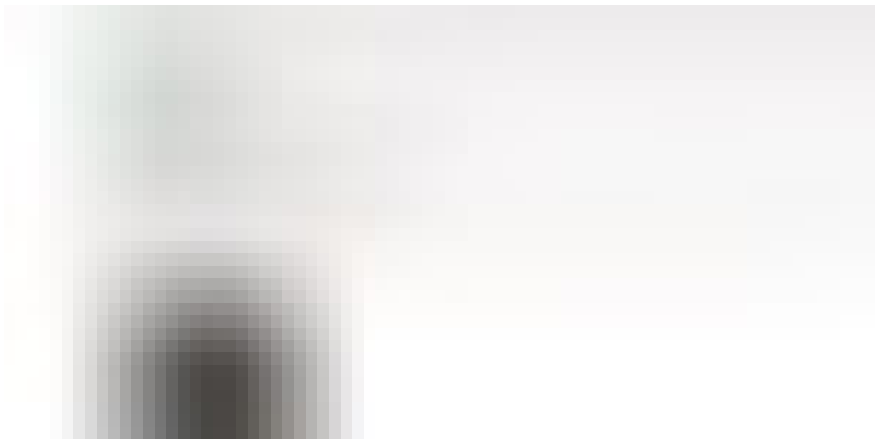
Training process for 60 Epochs.

Misclassification result for 60 Epochs.

Training and Misclassification error result for 150 Epochs.

From the images, it is obvious that as we increased the number of Epochs, the performance increased, although the difference wasn't really much.

We then went ahead to grab a new image from the internet and classified it through our network.



Loading an image of a cat.

After downloading the image, we create a dataset with this image for inference and get model outputs on the inference data.

Correct prediction of a cat.

After testing the network with several images, the network made a lot of wrong predictions like classifying an airplane as a deer, a dog as a cat, and many more. But notably, after increasing the number of epochs, the classification performance slightly increased which became evident when the network misclassified an airplane as a bird. After many more testing, the network luckily made a correct prediction by classifying a cat as a cat.

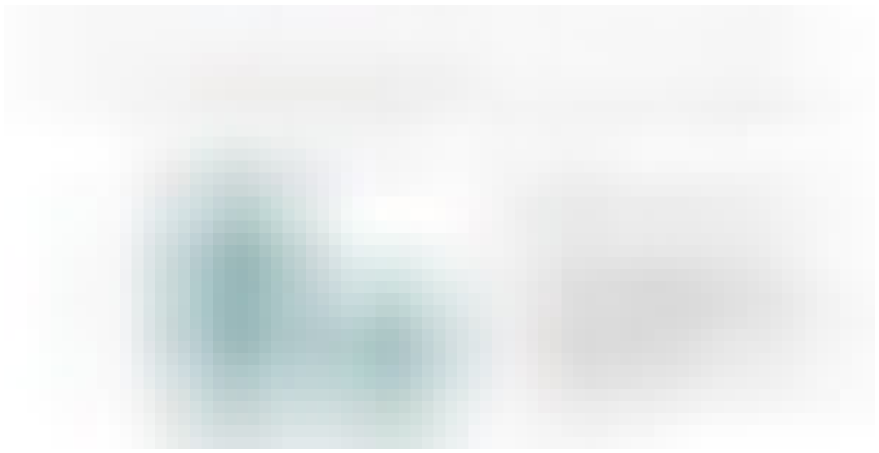
DISADVANTAGES OF NEON

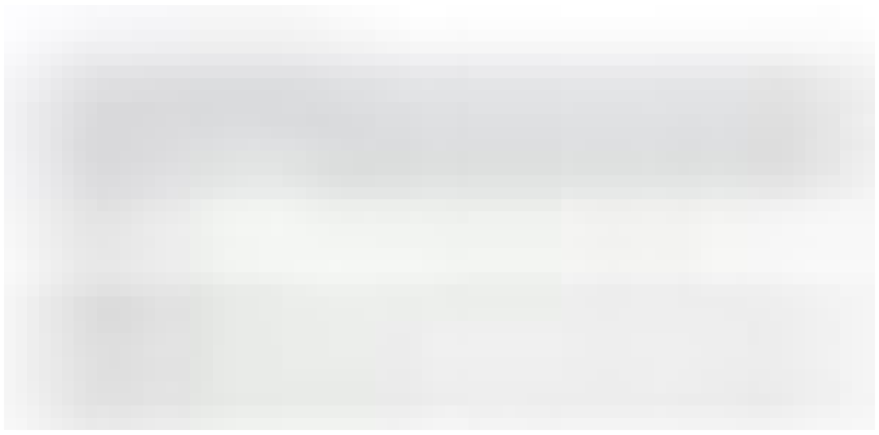
Python 2 ONLY: The fact that most of the Neon sample codes are based on python 2 is a disadvantage for python 3 programmers. Since we were using python 3, we had to edit some of the codes, even base codes which were not compatible with python 3 before we could test some datasets. For example, the parenthesis after the print statement, method of importing the **cpickle** library.

Too many errors: We encountered a lot of errors while exploring the framework which made us seek help from related questions asked on forums like Quora, Stackoverflow and many more.

Scattered dependencies: This was not really a big issue, but it will be best if it gets sorted out. As we kept advancing and trying out things on the framework, we got errors due to some missing files or dependencies. The most annoying of all was when we had to uninstall the latest version of a particular library (mccabe) and switch to an older version because the newer version could not work well with some other libraries (flake32).

ADVANTAGES OF NEON





From the images above, it is obvious that the Nervana Neon framework is the fastest framework alive.

Asides the statistics above, we also witnessed the speed of the framework during the training process. For example, it took Neon about 5 to 6 hours to train the CIFAR10 dataset at **150 epochs** on a **CPU**, which is incredibly awesome.

Comparing Neon with other frameworks

From the above charts, we can see that Neon is the poorest with respect to tutorials and learning materials, GitHub interests, GitHub contributors and Industrial usage. This is due to the fact that the framework is new to the market. But I am certain that this is something Intel is working on at the moment.

Who is Neon for?

Intel is trying to make Neon the number one framework for robotics, so obviously, in few years to come, Neon will be mostly used by those in the field of robotics.

Also due to Neon's high speed, it is the best for making research and testing.

CONCLUSION

In conclusion, even as this post is centered around the MNIST and CIFAR10 dataset, there is more to the Nervana neon framework like transfer learning, fine tuning, creating custom datasets and so on.

Also, we can't ignore the fact that Neon is a new framework, and it needs time to catch up with the likes of competitors like TensorFlow and Pytorch.

#TeamNeon

Ultimately, this was a team effort from #TeamNeon, and we hope to achieve greater things together.

. . .

We want to appreciate the AISaturdayLagos ambassadors Azeez Oluwafemi and Tejumade Afonja because this project would not have been achieved without their support through teachings and other learning resources. Also, we really appreciate our Partners FB Dev Circle Lagos, Vesper.ng and Intel for ever being consistent with the support.

A big Thanks to Nurture.AI for this amazing opportunity.

Also read how AI Saturdays is Bringing the World Together with AI

See you next week 😎.

Links to Resources

> *Stanford CS class CS231n: Convolutional Neural Networks for Visual Recognition.*

> *A Beginner's Guide To Understanding Convolutional Neural Networks by Adit Deshpande*