Convolutional Neural Network Optimization and Model Exploration in PyTorch

Carson B. Hanel,

Texas A&M Undergraduate Researcher,

Internet Research Lab

***Abstract*—In modern era computing, the universal trend that's being seen around the globe is the exponential rise in the amount of data being generated daily by users, like you and me. The traditional answer to this increased demand for computation was to simply wait for Moore's Law to work in your favor, and without any extra work, the same code would run much faster. Unfortunately, we've hit a near hard limit on the number of silicon transistors that can fit within a single CPU, and so the push has been further towards increased cores in a system to answer that call for increased computation. This comes with its own associated difficulties, however, and has led to a new frontier of multiprocessing and cluster computing research. Because, usually, a single worker computer utilizes a single GPU, or dual-GPU enclosure, parallelization of a Neural Network architecture generally involves multiple worker computers cooperating in a cluster, thus requiring communication of model and weight parameters between nodes. Following this idea, the purpose of this work is to effectively discern, analyze, and discuss the benefits of various methods of optimizing the compute time of Neural Architectures, as well as how those methods compare to similar neural optimizations. In order to maintain diversity in implementation, the target framework that this research is built around is PyTorch, due to the level of autonomy that is given to the researcher in implementation of various PyTorch architectures. This work, in the absence of cluster computing resources, seeks to optimize Convolutional Neural Networks in a down-scale fashion, utilizing a much larger GPU to tune training times for a much smaller GPU. Effectively, this work prepares the reader to distribute their network, but stops short of distributed testing.**

I. Introduction

Computing research has been traditionally carried out to seek out efficient methods for solving algorithmic challenges, from the Gaussian Walk to the Travelling Salesman problem. Always, it's been a push to improve time, space, and energy complexity of these tools in order to more effectively be able to assess, produce and complete work. Originally, machine learning was a purely statistical field, with most major breakthroughs being from the mathematical underpinnings, such as exposing methods in linear algebra to expose the variance in a dataset in order to marginally differentiate the data stratification of the sub-classes of the dataset. While these methods were incredibly effective, and still serve as much of today's quantitative marketing and data science tools due to their interpretability, more computationally intensive methods have arisen that can pick up on variations in data that would be far too complex or not "noisy" (i.e., variate) enough for traditional machine learning methods to pick up on. Because of this, there's been an industry trade-off between interpretability and ease of computation and deeper connections in data, though lacking true explanation as to what these things are (i.e., why is it that the algorithm can tell a dog from a cat?).

This work explores the hyperparameter space of one such sub-type of machine learning algorithm, which is applied to image recognition, and known as a Convolutional Neural Network. Because image recognition is one of the most computationally expensive tasks to run, we'll take a journey through obtaining significant world class accuracy on consumer hardware, and show how to utilize the same training time to improve model accuracy above 90% on GPU systems, and scaling the experiment downwards - attempting to aid the reader in future distribution of the algorithm by reducing the search space for the optimal network architecture, hyperparameters, and batch size given the hardware constraints. Come along on a journey to world class accuracy on a world class dataset.

II. The History of Neural Networks

The intuition behind modern neural networks stems from modern science's understanding of the brain's neural cortex and how we, as humans, learn. Pavlovian psychology, originating in 1901 from Ivan Pavlov and his assistant, Ivan Tolochinov, showed humanity that those organisms housing a brain learn sub-conscious ques, or "conditioned reflexes", which roughly equated to the brain learning signals that indicate a stimulating event, causing reactionary changes in physiology [1]. This simple idea would become the entire basis for training method for neural networks; reinforcement learning.

In 1943, Warren McCulloch and Walter Pitts produced the computational model for the body of the neural network, by applying threshold logic, commonly referred to as activation functions such as sigmoid, tanh, and ReLU. From the production of the initial idea, the research partners split ways with McCulloch being primarily interested in applications of artificial intelligence, and Pitts being primarily interested in its similarity to the brain [2].

Following Pavlov and the Pitts-McCulloch duo, Frank Rosenblatt developed the Perceptron module, which was a compact sigmoid based application of the previous threshold logic [3]. Next, in 1975, Werbos presented the back-propagation algorithm, which solved Rosenblatt's perceptron's "exclusive-or" problem by making multilayer learning networks possible. Essentially, the idea was to distribute the loss, or rather, the difference between the expected (labelled) output, and the output given by the network, back through the network after obtaining an output from the network, in such a way that the network can learn the function of the input data through a gradient descent process, mimicking Pavlov's conditional reflex [1, 4].

Finally, convolutional neural networks were made possible by early work by Hubel and Wiesel in the 50's and 60's which divulged to science that monkeys and cat visual cortexes both contain neurons that individually respond to small regions in the visual field called the receptive field, and that neighboring cells have similar and overlapping receptive fields, in a network-like structure [5]. From these developments – massive neural networks being ultimately made possible, with the given visual neural cortex comparison for identification in animals – it's intuitive to see the flow of the decades of research that followed that brought us here today. What was this era lacking? Simply, computational power. It's only just now that people are gaining access to deep neural networks based upon the fundamental theories proposed by McCulloch, Warren, Rosenblatt and Werbos. There are many proposed answers being worked on at the time insofar as utilizing CPUs, GPUs, and even Field Programmable Gate Arrays (FPGA). Let's now take those developments, and experiment with the modern-day actualization of CNN implementations.

III. CNN Nomenclature

Neuron:

A unit of computation, much like a Perceptron; utilizes threshold logic in order to learn the function of a given input. Like was seen in [5], neurons can be clustered together to increase the effectiveness of the overall cortical function, producing a neural network.

Identity Function:

F(x) = x, simply, the identity function is the function such that, applied to x, evaluates to x.

Data, Activations:

Inputs which activate the threshold logic of the neuron, causing it to "fire" and produce an output. These could be time series data, images, or even simply a sequence of text.

Activation Function:

The activation function is the internal threshold logic to a neuron. By being fed data, and the loss being propagated backwards to the function, the identity function of the data is learned.

Backward Propagation:

After an activation is fed to the activation function and output is obtained, the error that lies between the actual output and the expected output is calculated, and then distributed backwards across the neural network to "train" the identity function.

Weights, Gradients:

Internal to the activation function are the weights, which comprise the gradient that defines the activation function. This gradient layout is what is tweaked during back propagation to learn the identity function of the given activations.

Features, Filters, Convolutions:

Extractions from the data through some form of data processing; These enhance the network's perspective of the data and allow the model to extract deeper meaning and tap into the variance more easily.

Kernel:

A matrix of weights which is utilized to apply a convolution to a given image data in order to extract deeper features.

Forward Pass:

Inputting data (activations) into the activation function internal to the neuron and retrieving the result of the activation applied to the internal weight gradient.

Backward Pass:

Applying backwards propagation across the network.

Hidden Layer:

Extra layers added in-between the input and output layers in order to increase the effectiveness of network by allowing more well-defined identity functions to be calculated which dig deeper into the variance extant in the dataset.

Fully Connected Layer:

Layers of neurons, including hidden layers, wherein all inputs to a given neuron on a given layer are given to all neurons of the following layer, ensuring full dataflow across the network's gradients.

Convolutional Layer:

A layer in which many kernels are applied to a given image data, which creates many "convoluted" versions of the given data. This has a high computational and space complexity even in modern day computers.

Batch Normalization:

In order to reduce the variance in the training data to simply the variance within the data, and reduce noise from arbitrary values, batch normalization de-means the outputs of a convolutional layer, which assures that all output convoluted images have zero mean and *unit*-variance.

Pooling Layer:

Layers which combine the output of a convolutional layer into a single input for a following layer. There are various forms of pooling, such as max pooling and average pooling, which take the max or average pixel value of a given X*X region of an image and condense that into a smaller x*x region to reduce the space complexity of an image.

Dropout Layer:

A layer in which certain outputs from a given layer are kept from the following layer. This is counter to a fully connected layer, which, in contrast, always gives all outputs of a layer to all neurons of the following layer.

Data Augmentation:

Because [5] found that the neural cortex utilizes a receptive field to identify objects, it is important to note that convolutional neural networks are subject to perspective, and as such, can benefit from simply applying transformations to the image data, allowing the network to "see" the dataset objects differently, and extract more robust features.

SoftMax Layer:

Usually the final layer of a neural network, which turns the outputs from the last-to-final layer into a range of probabilities over the possible output space of the network. In the case of CNN's, it's the number of different classifications of objects which reside within the data in the dataset.

IV. CNN ARCHITECTURES AND METHODS

Shallow Convolutional Neural Network:

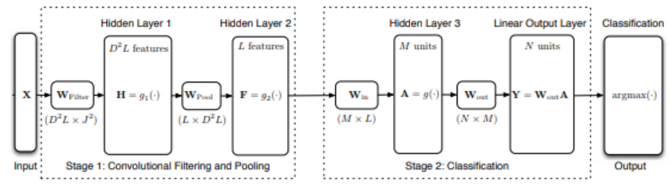


Figure 1: Convolutional Neural Network Architecture [6].

As a baseline, this work utilizes a fast learning CNN architecture developed by Mark McDonnell and Tony Vladusich from the University of South Australia in early May 2015 [6]. This architecture consists of two stages; convolutional filtering and pooling, and classification which utilize the layers that were identified earlier in the paper.

During the convolutional filtering and pooling layer, variable numbers of convolutions per convolutional layer were tested in order to gauge their effect on convergence time, space complexity, and hardware utilization. Let's walk through an example of a convolution and pooling phase. Let ***M*** be the size of the minibatch of images, and ***N*** be the number of kernels involved in the convolutional layer, and each image be of size **32x32,** such as in Cifar datasets. The application of the kernels to convolve the minibatch of images will produce ***M*** convoluted versions of each of ***N*** input minibatch images, giving a total of ***NM*** resultant images which are fed into a hidden layer utilizing the ReLU activation function before entering the second convolutional layer. From there, a second convolutional layer makes another ***M*** convoluted versions of each ***NM*** already convoluted image, having a final product of ***N x pow (M, 2)*** doubly convoluted images, all utilizing a stride of 5, and then finally projecting those ***N x pow (M, 2)*** images down to a **16 image space** within the convolutional layer. In order to reduce the space complexity of these resultant convoluted images, max pooling is utilized to reduce the **16x32x32** images into **16x5x5** max pools, taking the maximum value of the original **25x5x5** sub-regions of the **1000 pixels** of the image. Once the resultant images are max pooled, the max pooled byproduct is fed through another hidden layer with the ReLU activation function before finally ending the convolutional filtering and pooling phase.

Moving on to the classification phase, unlike the original Shallow CNN, this work utilized three linear layers, with the first being given the **16x5x5** max pooled resultant images from the previous layer into a **120** point linear space hidden layer, then condensing from **120** points into **84**, and then final **84** points leading into the **10** possible classifications of the Cifar-10 dataset, and finally the CNN's output can be provided to an optimizer, in this case, a Stochastic Gradient Descent (SGD) module which calculates the loss between the resulting output and the actual output given by the learning supervisor. This loss is then derived and back propagated throughout the network in order to adjust for correctness, and effectively, learn.

Deep Residual Convolutional Neural Network:

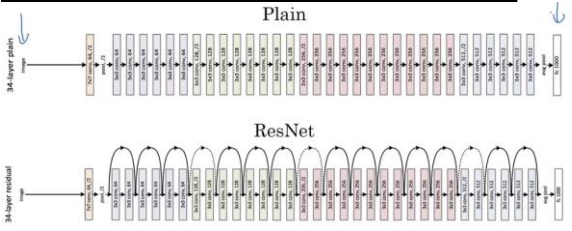


Figure 2: VGG vs. ResNet-34

To improve upon the Shallow CNN, we utilized an architecture that was build counter to the previous model. The Shallow CNN suffers because the main tuning in the model has to do with the number of kernels utilized in the convolutional layers. Because convolution introduces noise into the image and takes it away from the original data contents, i.e., you can only convolve an image so many times while retaining relevancy to the original data instead of garbage, putting a limit on the effectiveness of shallow architectures.

In December of 2015, Microsoft Research presented a new architecture, namely ResNet, which relied on a new format on convolutional layers which retained residual data extant in the original image passed to that layer. Intuitively, the idea was simply to, unlike the previous convolutional layers, use batch normalization after each convolutional phase to create only a single convoluted copy of the input image, and then feed that single resultant image into an ReLU activation to learn the identity function of that convolutional layer. After the convolutional phase, a matrix summation reduction between the input image and the convolution is carried out before feeding it into that layer's final output activation unit. Because this is roughly equivalent with reducing the total convolution by half, averaging it with the original image, the Convolution->Batch Norm->ReLU activation (CBA) sequence is repeated twice before the reduction. This method drastically reigned in the noise associated with convoluting an image, while retaining the residual of the input data. This layer is referred to as the "Identity Block".

To allow deeper convolutions, a "Convolutional Block" – a variant of the identity block which mimics the original convolutional phase with residuals – was also developed for the architecture. The difference between the Identity and Convolutional blocks is simply that the convolutional CBA sequence is applied three times to the convoluted copies of the input data, and once to the input data before summation, and does not necessarily have a single image output. Because of this, ResNet makes both deep and wide networks possible, and tunable to a multitude of image recognition applications.

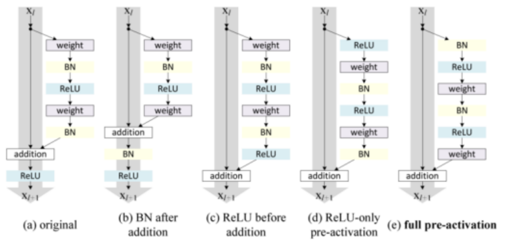


Figure 3: ResNet block types, and updated blocks.

Though ResNet has since had many updates to its architecture, this work focuses on a version of ResNet, namely ResNet-18, which is comprised of the original block types in a format of [2x ID, 2x CONV, 2x ID, 2x CONV] layers which are utilized to reduce the network from a total of 512 convolutions to 256 before max pooling, then 256 to 128 before max pooling, and finally 128 to 64 before a final max pool and classification.

V. GPU versus CPU effectiveness

To keep this section simple, deep learning involves intensive computation applying transformations and arithmetical operations to matrices of data and gradient weights. Because of the independence of these matrix transformations when done in abundance, the underlying operations involved in deep learning can be massively parallelized. Commodity CPU hardware usually contains between two to sixteen physical cores and provides a cornucopia of usages to be able to satiate a full operating system and applications. GPU hardware, on the other hand, is suited more for pure arithmetic, as graphics rendering also requires massive matrix math in order to work effectively, especially in 3D settings, or under ray tracing loads.

Because of this, GPUs are equipped with thousands of arithmetic logic cores, called CUDA cores, which dwarf even the most powerful CPU's number crunching abilities. For an example with this work, a GTX 980 houses a whopping 2048 CUDA cores, whereas, again, commodity CPU's max out at around 16 physical cores. This work goes on to make this comparison evident by exposing CNN training times on both CPU and GPU based systems.

VI. Methods of Experimentation

Before diving into experimental results, let's go over a few choices for carrying out the experiment, such as the platform utilized, the model parameters, and the harboring hardware. The goal of the experiment was to find the best CNN setup on **Machine A** to prevent wasting time on the less powerful **Machine B**. The scaling factor is roughly that **A**, with 2048 CUDA cores, trains **10x faster** than **B** with 384 CUDA cores. This led to the perfect simulation environment to create equivalent would-be long training sessions on **B**, and run the computations on **A**. Effectively, this searches the space for the most optimal way to utilize computational resources while obtaining world class accuracy on the tested models.

1) PyTorch IDE

Pythonic Implementation: Unlike Keras and Tensorflow, PyTorch was developed natively for Python. This allows easy harnessing of system GPU capability without the necessity of knowing CUDA or C++ but is just as easily harnessed with the CPU with minor modifications. Because of this, PyTorch was an obvious candidate for variate hardware experimentation. As well, for future applications, PyTorch has a distributed computing package.

Model Swap Ability: PyTorch has abstracted the forward and backwards pass paradigm utilized by reinforcement learning algorithms, and as such any model can be trained by a crafted training regime for comparison without rewriting code.

CUDA optimization: NVidia has released CuBLAS (CUDA Basic Linear Algebra Suite) and CuDNN (CUDA for Deep Neural Networks) where are high performance optimizations built upon the basic CUDA package. PyTorch has support for these optimized libraries built-in, with the base install including the binaries. There's no need to spend time to optimize!

Prefetching I/O module for data: For batched datasets, PyTorch has a DataLoader class which can be given a user defined number of worker threads to pre-process data either for CPU or GPU computations to aid in efficiency. Because of this, PyTorch minimizes pre-processing, especially with GPU computations, by offloading transformation computation to the CPU without a personalized I/O wrapper being written, which is a non-trivial problem.

PyTorch Visualization: Like Tensorflow, PyTorch supports Tensorboard, which allows for greater visualization of your model and the runtime behavior.

FastAI adoption: PyTorch has been adopted by Jeremy Howard's FastAI program, which has brought industry standard algorithms, training, and methods to a much wider audience in the form of two-hour digestible code along lectures. This has caused a great bit of intrigue around the platform.

2) Optimizer

In this experiment, Stochastic Gradient Descent with Momentum and Weight Decay is utilized, assuring that the loss continually decreases and avoids exploding gradients by decaying the weights, and a learning rate of .001 is utilized for all experiments, both ShallowNet and ResNet.

3) Dataset

Cifar-10 is a widely utilized dataset, consisting of 60,000 total images across ten categories. Each image is 32x32 (1024) pixels and centered in on the object to be classified. Cifar-10 has a sister dataset, Cifar-100, which was tested upon during this experiment, but has a much less impressive maximum accuracy, as it has the same number of examples, but ten times the number of categories to classify. This experiment utilized the common 50,000 train set size, 10,000 test set size paradigms, and sought to improve on the "Shallow CNN" model and accuracy, then finds a balance between "ResNet" and "Deep ResNet" accuracy.

4) *Data Augmentation*

Horizontal and vertical flips of the data were utilized to give separate perspectives of the data, and as well the data was randomly cropped and off-centered in order to force multiple perspectives of the same image between batches.

5) *Multiple Iterations on the Same Minibatch*

Because data augmentation is computationally expensive for the CPU and requires the GPU to wait during the transformation phase, multiple passes on the same data were tried, and ultimately unsuccessful at increasing the accuracy of the CNN, and thus discarded from this test and unencouraged.

6) *Hardware Utilized*

Machine A (Hercules):

- Intel i7 4790k CPU at 4.38 GHz
- 16GB DDR3 RAM at 1600 MHz
- EVGA GTX 980 with 2048 CUDA cores

Machine B (Hermes):

- Intel i7 5500U CPU at 2.4 GHz
- 8GB DDR3 RAM at 1600 MHz
- NVIDIA GTX 940M with 384 CUDA cores

7) *Parameters Tuned Per CNN*

ShallowNet:

- Batch size
- Learning rate
- Number of convolutions per convolutional layer.

Results gotten from the tuning of ShallowNet were utilized in the testing of ResNet to test applicability of hyper parameters between models.

VII. Results

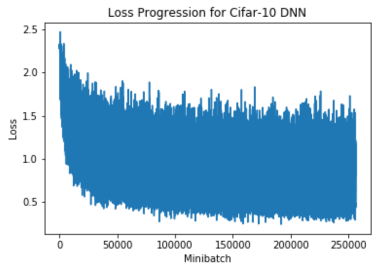
To begin with the results, while there was run data gotten from ShallowNets ranging in the kernel sizes of {8, 16, 32, 64, 128, 256, 512, 1024} per convolutional layer as well as batch sizes of {8, 16, 32, 54, 128, 256}, we'll focus first on the difference in the hardware utilization between 32 and 128 image minibatch scenarios. To show off the optimal minibatch size and discuss what is happening, let's look at a 32 kernel ShallowNet trained on both 32 and 128 size minibatches, and the effect that it has on the runtime given to each critical section of computation, i.e., CPU data transformations, GPU processing, and inter-processor communication.

1) *ShallowNet on Machine A:*

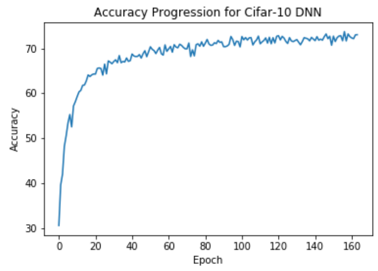
<u>Small Minibatch:</u>

Network Size: 32 kernels,
Batch Size: 32 images,
Training Time: 15 minutes

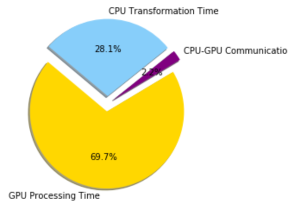
Loss:



Accuracy:



Hardware:

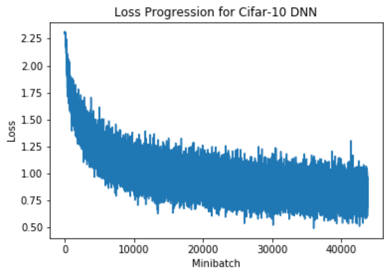


End Test Set Accuracy: 73.75%

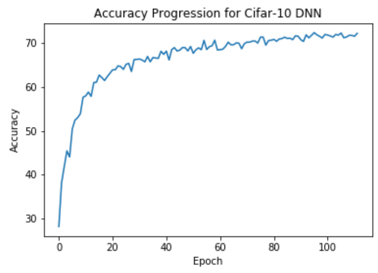
<u>Large Minibatch:</u>

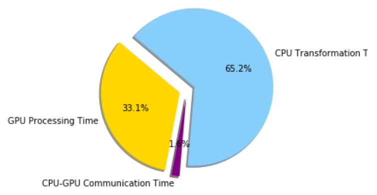
Network Size: 32 kernels,
Batch Size: 128 images,
Training Time: 15 minutes

Loss:



Accuracy:





Hardware:
End Test Set Accuracy: 72.40%

Insight: As you can see from the comparison between the hardware utilization charts between small and large minibatch sizes, larger minibatches cause the time required for the CPU to transform a batch to dwarf the runtime allotted for a GPU. Because the GPU is constantly waiting on the CPU, epoch times are much higher than required for small minibatches. Because a model increases in overall accuracy per epoch, it is imperative that minibatch size remain small, as it increases training speed and reduces hardware runtime inefficiencies. Because this paradigm applies to all variants of ShallowNet and ResNet as well, we'll omit going over runtime charts for those networks. Finally, because of this comparison across all networks, a minibatch size of 32 was chosen for ResNet testing across Machine A and Machine B.

2) *Increasing ShallowNet Width:*

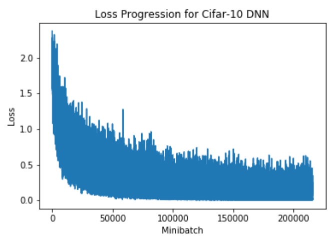
Although ShallowNet with 32 kernels can be run with high efficiency on Machine A, it struggles to improve beyond 75% accuracy, even with much longer training times, and severely underutilizes the resources available in the GTX 980. To increase the richness of features extracted from the ShallowNet, the number of convolutional kernels per layer was increased from 32 to eventually 1024 scaling in powers of 2, with the most effective network being 128 convolutions per layer and a minibatch size of 32. Unfortunately, as seen in [6] and further experimentation, ShallowNet caps out at around 75-81% accuracy, with Machine A reaching 80.14% accuracy in around 8 hours. In order to increase the accuracy and reduce training time of the model, this experiment went on to adopt the ResNet-18 model and test its effectiveness in image recognition convergence.

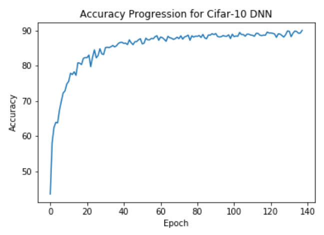
3) *ResNet-18 on Machine A:*

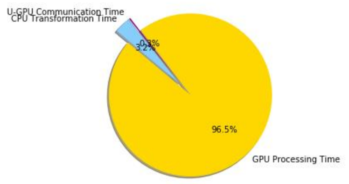
To test the effectiveness of ResNet-18 at converging, the model was trained with batch sizes of 32 and 128 images and a learning rate of .001. Because ResNet was developed with an expanding convolutional architecture between layers (64->128, 128->256, 256->512), it is less subject to being tuned at the width level, though there has been further research done on making both ultra-wide and ultra-deep residual networks effective, with ResNet-1001 being the largest tested, having around 200x more layers than the model trained in this experiment [7].

Small Minibatch:

Network Size: 18 layers of [INPUT, 2x ID, 2x CONV, 2x ID, 2x CONV, OUTPUT]
Batch Size: 32 images
Training Time: 3 hours, 32 minutes

Loss:


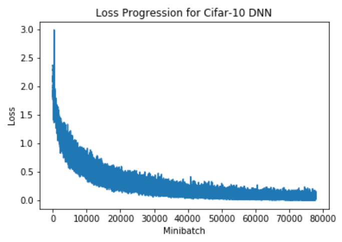
Accuracy:


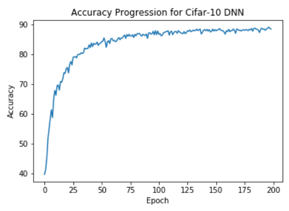


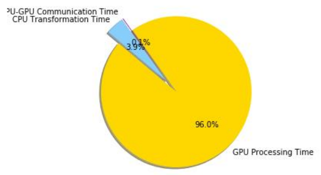
Hardware:
End Test Set Accuracy: 90.13%

Large Minibatch:

Network Size: 18 layers of [INPUT, 2x ID, 2x CONV, 2x ID, 2x CONV, OUTPUT]
Batch Size: 128 images
Training Time: 4 hours

Loss:


Accuracy:




Hardware:
End Test Set Accuracy: 89.17%

Insight: ResNet-18 was much, much more efficient than all renditions of ShallowNet, with the GPU being active roughly 96% of the time during training, and only pausing for data augmentation by the CPU for 3.9% of the time. As well, the architecture was able to converge to an accuracy of 90.13% on the test set, surpassing the original 87.65% ResNet publication accuracy due to advancements made in CuDNN since 2015, as well as hardware and model improvements adopted by PyTorch, but falls short of Deep ResNet's 93.57% accuracy on Cifar-10, mostly due to reduction in training time. The original goal was to be able to reach 90% accuracy on Machine A such that Machine B could mimic that convergence in under 48 hours. In order to emulate this, a four-hour runtime window was given to ResNet, as the rough scaling factor between Machine A and Machine B's GPU setup is x10, this would result in a maximum runtime of 40 hours, give or take runtime fluxuations in hardware demands. Much to the researcher's surprise, ResNet-18, the most compact model of ResNet, was able to surpass 90% accuracy in just three hours and thirty-two minutes, leading to a runtime of approximately 36.5 hours required to produce on Machine B, effectively matching the challenge set forth.

Finally, mirroring the ShallowNet experiment, one can observe that both ResNet and ShallowNet have less noisy loss progression with larger batches, however, larger batches make the algorithm take much longer to converge. In the small versus big minibatch trails, using the smaller batches, ResNet was able to surpass 90% accuracy in approximately 3.5 hours, whereas the larger batch ran for nearly half an hour longer and achieved 1% less test set accuracy. With respect to hardware, that's an extra whopping 8 hours on Machine B's GPU just for a single percentile of accuracy, making it highly advantageous to opt for smaller minibatch sizes.

4) *Compute Times across all Processors:*

	Training Processor	**Augmentation Processor**	**Shallow Epoch 32 Batch**	**Shallow Epoch 128 Batch**	**ResNet Epoch 32 Batch**	**ResNet Epoch 128 Batch**
Machine A	NVIDIA GTX 980	Intel i7 4790k	10.25 sec.	7.02 s	**88.01 sec.**	95.08 sec.
Machine A	Intel i7 4790k	Intel i7 4790k	146 sec.	152 s	**2833 s**	2570 sec.
Machine B	NVIDIA GTX 940M	Intel i7 5500U	N/A	N/A	**890 sec.**	N/A
Machine B	Intel i7 5500U	Intel i7 5500U	N/A	N/A	**4901 sec.**	N/A

Figure 4: CNN Epoch Times by hardware and architecture

As you can see by the comparison chart, Machine B's GPU based setup is roughly 10x slower than Machine A's on the calculation of ResNet, and Machine B's CPU was roughly 1.72x slower than Machine A's. To carry out this experiment, around 4 days of computation on the GTX 980 were carried out. This same experiment being carried out utilizing the GTX 940M would've taken over a month, and the Intel i7 5500U would take nearly 6 years to produce the same result. *Figure 4* aptly illustrates the evolution of hardware, and the contrast between CPU and GPU effectiveness at training deep neural networks, alongside *Figure 5* which is a more blatant comparison of ResNet processing ability.

	GTX 980	**GTX 940M**	**Intel i7 4790k**	**Intel i7 5500U**
GTX 980	-	+1000%	+3496%	+5569%
GTX 940M	-1000%	-	+318%	+550%
Intel i7 4790k	-3496%	-318%	-	+172%
Intel i7 5500U	-5569%	-550%	-172%	-

Figure 5: Comparative performance of hardware on ResNet

VIII. Conclusion

In conclusion, this work shows that smaller batch sizes are optimal for hardware performance in deep architectures, as it not only allows quicker convergence and epoch times, but also maintains the most optimal balance between CPU data augmentation phases and the GPU's network operations. As well, annealed learning rates with SGD and utilization using Momentum and Weight Decay. Further, utilizing a smarter architecture such as ResNet, which incorporates input residual data from the activations in order to assure that deeper convolutions don't get too far from the original image data, maintaining higher accuracy and deeper relevancy. Thus, utilizing smarter architectures that more efficiently tap into a dataset's variance is highly suggested. Lastly, it is important to understand the constraints of the problem that is being solved, and as such, GPU's are wildly better suited for deep convolutional neural network performance.

References

[1] G. Todes, Daniel Philip (2002). *Pavlov's Physiology Factory*. Baltimore MD: Johns Hopkins University Press. pp. 232 ff. ISBN 0-8018-6690-1.

[2] McCulloch, Warren; Walter Pitts (1943). "A Logical Calculus of Ideas Immanent in Nervous Activity". *Bulletin of Mathematical Biophysics*. **5**(4): 115–133

[3] Rosenblatt, F. (1958). "The Perceptron: A Probabilistic Model For Information Storage And Organization In The Brain". *Psychological Review*. **65** (6): 386–408.

[4] Werbos, P.J. (1975). *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*. vol. 134, pp. A635–A646, Dec. 1965.

[5] Hubel, D. H.; Wiesel, T. N. (1968-03-01). "Receptive fields and functional architecture of monkey striate cortex". *The Journal of Physiology*. **195**(1): 215–243.

doi:10.1113/jphysiol.1968.sp008455. ISSN 0022-3751. PMC 1557912. PMID 4966457.

[6] “Enhanced Image Classification With a Fast Learning Shallow Convolutional Neural Network” **arXiv:1503.04596**

[7] “Deep Residual Learning for Image Recognition” https://arxiv.org/abs/1512.03385

[8] PyTorch Basic CNN: https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html

[9] Algorithm Optimization: https://towardsdatascience.com/speed-up-your-algorithms-part-1-pytorch-56d8a4ae7051