Overview of Deep Neural Networks in Torch

Bernard Yett
School of Electrical and
Computer Engineering
Vanderbilt University
Nashville, Tennessee 37235
Email: bernard.h.yett@vanderbilt.edu

Abstract—**This report it to satisfy requirements for an independent study course focused on deep learning and overseen by Dr. Xenofon Koutsoukos. Various types of deep neural networks are available for use, with each being best suited for certain tasks. Torch is one of many packages that exist that allow a user to create, train, and run experiments on these networks. Issues can arise due to factors such as lack of previous experience in the area as well as poor processing power on the available computer. Many things are possible with this toolset and it is certainly possible to achieve a high level of results similar to companies such as Facebook and Google that developed the library.**

I. Introduction

Deep learning was the subject of an independent study course taught by Dr. Xenofon Koutsoukos at Vanderbilt University. Myself and two other graduate students were tasked with using distinct deep learning libraries to implement a variety of neural networks. Deep learning was an interesting subject for this course, mainly because of its many current and future applications. I was able to begin learning a new programming language as part of this process, while coming to understand the differences between the types of neural networks and what they can be used for.

One of the applications for deep neural networks (DNN's) is in handwriting recognition. A large dataset called MNIST has been compiled that features a collection of handwritten digits (0-9) from many different people. This type of data allows DNN's to be trained and later used by banks to automatically recognize the value of a check that is being deposited, among other things. In addition, object recognition is a rather famous and interesting application of this subject area. An example I saw involved training on a set of images that included horses, dogs, cats, cars, people, boats, etc. After training, the model was able to determine differences between these objects based on things like curvature and corners. Finally, there are a few applications that I found to be somewhat more entertaining. One is training a DNN on the normal day-to-day usage of all network activity at a company. Once trained, automatic detection of suspicious activity is possible, quickly putting a stop to any potential threats. Also, training can be done on songs, books, and other such sources, with the eventual end result of the model attempting to recreate the various components of the input - often hilariously incorrect, but also clearly on the right track after many epochs of training.

I selected Torch, described as a scientific computing framework that uses Luajit (a form of the Lua programming language), as my primary tool for this course. There were a few reasons for this, one of which is that I thought it would be more interesting to use something based around a programming language I was not already familiar with. While Tensorflow uses Python, Caffe is built around C++/Python, and there is technically an option to use Python for Torch known as PyTorch, Torch was primarily built around using the Lua language. Another useful reason for using Torch was the availability of many example programs. These examples both aided me while trying to pick up the Lua language while also providing a nice starting place when compiling programs for various sets of training data and types of DNN's. Torch was suitable to run on the MacBook Pro I already owned rather than needing to set up a Linux OS as was required for other packages - this was another point in Torch's favor. Last but not least, Torch is used by a variety of impressive comapnies. Tops among these are Facebook, Google, and Twitter, along with many recognizable universities such as NYU and Purdue.

As a result of this independent study class, I was able to accomplish a few things. I learned how to use the Torch library and the Lua language. I learned about the various types of DNN's (more information on them later), and was able to implement them in Torch. After performing several training epochs on sections of various datasets - mostly MNIST as described previously - I was able to obtain good results by inputting other subsets of data through the model. These results consistently had over 90 percent accuracy, although going over about 6 or 7 layers tended to make the model worse rather than better. Additionally, when using some of the more effective DNN's on relatively simple data such as MNIST, accuracies of approximately 99 percent were obtained. This is probably at the same level as a human trying to recognize handwritten numbers, and can be accomplished fairly quickly, really showing off the power of these networks. However, there are definitely some issues both with Torch and neural networks. One of these is related to the initial setup of Torch. When installing, the torch library is added to the PATH variable of your computer. The instructions then say to source torch once in order to refresh the environment variables. This worked for me - until the next time I tried to use Torch. As it turns out, this source step is required each time a new terminal window is opened to run torch. The problem is easy enough to fix, but it resulted in much frustration for me at the beginning of the course. An inconvenience related to creating and training various DNN's is the processing power required. Torch, as well as many of the other means of working with DNN's, has the ability to run on a GPU. My MacBook does not have this capability, and so everything was run directly on the CPU. None of the work required for this course was extremely computationally expensive, but many models still took a good amount of time to train. Trying to process more data, especially with things such as music and videos, would take a prohibitively long time to train enough for desired results. This is something that a potential user should keep in mind when setting up their DNN's.

II. Deep Neural Networks

Thus far DNN's have been referenced many times, but not clearly explained. A DNN is described as an Artificial Neural Network (ANN) with the additional requirement of having multiple layers of neurons. Basically these networks are modeled on the neurons in our brains, with various stages of interconnectedness. The first main type of DNN goes by a few different names - deep feedforward network, feedforward neural network, and multilayer perceptron. The idea behind these is that data is sent into the model at an input layer, it goes through some number of intermediate layers for processing, and then finally is sent to an output layer so that the results can be observed. The networks part of these is that several functions/layers are used between the input and output. Of course, to actually be considered a deep network, a certain number of layers must be used; at least 10-15 seems to be the standard. During implementation, this is not always the easiest condition to meet, as more layers does correlate to longer run times but does not necessarily correlate with better performance. Gradient descent is an important aspect of feedforward models, enabling the model to actually converge to the solution. Gradient descent has two requirements: a cost function and output representation. The cost function is often the negative log-likelihood, which comes from using maximum likelihood as the training method. Then, for the output, the focus can be on minimizing the mean squared error. Other possibilities are available as well. Another important aspect to consider is the type of hidden unit (activation function) to choose. Some popular choices include rectified linear units (ReLU), using $g(z) = max\{0, z\}$, logistic sigmoid activation function (Sigmoid), using $g(z) = \sigma(z)$, and the hyperbolic tangent activation function (Tanh), using $g(z) = tanh(z)$. Related to feedforward networks but essentially their opposite is the idea of back-propagation (backprop). While in feedforward data flows from the input through the hidden layer(s) to the output, using backprop data can go from the cost function backwards towards the input in order to compute the gradient. The gradient itself is necessary to have in order to perform learning on gradient through means such as stochastic gradient descent. The basic idea with backprop is to use the basic chain rule from calculus recursively to get an expression for the gradient; other means can then be used to actually solve this expression [1].

The next type of DNN is a convolutional neural network (CNN). Simply put, a CNN is just a neural network that also includes one or more layers using convolution; this replaces the general matrix multiplication that is normally used. To use convolution, a kernel is introduced. This kernel is an n by m matrix for positive integers n and m. The purpose of the kernel is to use it similarly to matrix multiplication, but to repeat the process for each time that the kernel can fully fit into the input matrix. For example, consider a 3x3 input matrix and a 2x2 kernel. This kernel can fit completely within the matrix four separate times, resulting in an output that is also 2x2. This is just a simple case - the input can be much larger and does not need to be square or even of constant size. In fact, one of the main benefits of using a CNN is that it is able to account for variable input sizes. Likewise, the kernel can be larger than 2x2 and is not required to be a square matrix. It does, however, have the restriction of being smaller than the input matrix unless other techniques are used to accommodate. There are three major ideas to CNN's that can help explain their usefulness and advantages compared to some other DNN's. First is the sparse interactions concept: because kernels are smaller than the inputs, fewer parameters need to be stored, resulting in improved computational efficiency. Next is parameter sharing, which comes about because the same parameters are used for multiple functions. Instead of needing to learn a weight value for each connection between layers, the kernel can be used repeatedly, thereby saving on memory usage and again improving efficiency. The final idea is that of equivariant representations. This concept comes about from the fact that applying a transformation of the input before performing convolution is equivalent to reversing those two steps. Therefore, while moving in the input does result in the same movement in the output, simply determining that the particular value or feature is present is generally what is desired [1].

Pooling is a useful tool often used in conjunction with convolution - it makes it so our results are invariant to small input translations. Normally models can be susceptible to errors when numbers or letters are slightly rotated, but pooling accommodates for this. Pooling works by sampling some number of the inputs and choosing one of them for an output, with max pooling being a common example that selects the maximum valued input. This process is repeated until all of the inputs have been pooled together appropriately depending on the desired pooling size. Many effective CNN's start with a layer of convolution, followed by a layer of pooling, and then a layer with a transfer function such as ReLU, Tanh, or Sigmoid. This process can be repeated as necessary to eventually reduce the original input down to manageable levels. Often the results from these layers are passed through some number of linear layers for final processing before the output occurs. There are a few other options that can be used for CNN's depending on the particular application. One of these is known as stride, which is basically just skipping positions in the kernel. The end result of this is that the computational cost is lowered - the counter to this is that features cannot be extracted as finely. Another feature of CNN's in known as zero padding. The idea behind this is to add zeros to the edges of an input to make it wider, which in turn can prevent the output from shrinking as much or at all. If something like that is necessary for a particular implementation, zero padding can be quite useful. However, it usually causes the actual data to be diluted somewhat, and over time the borders of layers become underrepresented [1].

The final type of DNN for the purposes of this paper is a recurrent neural network (RNN). The main function of RNN's is to process a sequence of values. Because of this, RNN's can process much longer sequences of values than the other types of neural networks, and for the most part they can handles sequences of variable length as well. The term recurrent implies that the current state of the system depends on the state of the system at a previous time (or many previous times). There are a variety of RNN's that can be implemented - basically any network with some amount of recurrence is classified as such. Some examples of these that can be used in Torch or the other libraries involve producing outputs at every time step while using recurrence between hidden units, the same output behavior with recurrence from the output to the next hidden unit, and recurrence between the hidden units with just one output at the end. The first of these is probably the most basic. We have an input that goes into a hidden layer (the hidden layer also receives input from the previous hidden layer), which leads to an output layer. This output layer is fed into a loss function that compares the output value to a target value. The other options can be defined similarly. A nice feature of RNN's is the ease of computing a gradient. All we need is the previously defined backprop algorithm (based on the chain rule); applying this to the graph of our RNN model computes the gradient. A different type of RNN, known as bidirectional, is an entirely new means of predicting an output. Instead of the result relying on just the past and present inputs, with bidirectional the whole input sequence affects the output. To accomplish this, one must combine RNN's moving forward and backward through time. Another modification on RNN's, known as long short-term memory (LSTM), makes use of self-loops that allow the gradient to continuously flow. LSTM networks are useful for a large number of applications - including handwriting and speech recognition tasks - and they tend to learn long-term dependencies quite well. These networks are introduced here prior to their eventual use as a RNN example [1].

Each of these types of neural networks have some strong suits when it comes to performing well for certain applications. Feedforward networks are generally the simplest of the DNN's, and therefore are generally used for less complicated things. Examples of this include processing datasets such as MNIST for handwriting recognition or processing a set of images to be able to tell the difference between objects, animals, etc. The next step up is a CNN. These networks are able to handle all of the cases that a feedforward network can, with a longer run time but also better results. Object recognition is much more superior with CNN's, to the point that they can isolate multiple objects in the same image and separately recognize each of them. CNN's are also capable of some impressive applications such as automatically coloring a black and white photo or determining appropriate sounds to add to silent movies. They are also used to assist RNN's in other tasks that require some kind of object recognition (which a RNN can't handle) plus a language output (that can't be done by a CNN). Examples of this include recognizing an object and printing out an appropriate label and determining that there are words in an image and using a trained RNN to translate the words from one language to another. This natural language processing is a strength of RNN's; interesting experiments have been conducted such as training on the full works of Shakespeare or the Linux source code and then seeing what kind of outputs can be generated. RNN's are also capable of learning from a sound input, such as a piano note, and eventually being able to present a similar output [2] [4].

III. Torch Overview

Torch was originally developed and is maintained by Ronan Collobert, a research scientist at Facebook, Clement Farabet, a senior software engineer at Twitter, Koray Kavukcuoglu, a research scientist at Google DeepMind, and Soumith Chintala, a research engineer at Facebook. Additionally, there is a large community of contributors that have imported datasets, created new packages, and displayed example programs. This community was extremely useful when I was first getting started in Torch; these and other resources are available to help newcomers and long-time users alike. The place to get started is definitely the Torch website (torch.ch). There you can find step-by-step instructions for downloading and installing Torch on either Mac OS or Ubuntu. There are also a few tips for installing other packages (such as torch, nn, and paths), along with instructions for sourcing the PATH variable (that needs to be repeated for each new terminal window you open as far as I can tell) and some other useful commands. One can find links to more information, such as a basic tutorial for a good first step into Torch and Lua or the community wiki page, along the top of the page. The wiki page is of particular importance; many datasets, training videos, and various examples can be found there [3].

The default for Torch is to use the Lua programming language, and specifically LuaJIT, which is a Just-In-Time Compiler for Lua. It is known for having a lot of flexibility plus a high level of performance and lower memory usage. From comparing with other students in the course it seems like Torch was able to run training more quickly than some of the other options (assuming all running on CPU and similar computer hardware); these LuaJIT capabilities likely played a part in that. There are websites for both Lua and LuaJIT that can be easily found through a Google search if more information is required. Torch itself supplies a respectable amount of training on this language; I was able to pick up on pretty much everything I needed just by going through Torch tutorials. The first tutorial I previously mentioned from the Torch website is a good place to start. Also, the community wiki page leads to a series of video tutorials - it may be useful to watch and follow along with these while also typing up the code yourself to increase opportunities for learning and verify that Torch is set up properly. An additional note is that, unless you are extremely familiar with editing files within the terminal, a suitable IDE should be selected. The one I chose is called ZeroBrane Studio, and allows for files to be saved in the Lua format - this is quite important to be able to modify programs so they can be tested and used repeatedly. Depending on current familiarity with the Python language, PyTorch could be used in place of just Torch. This seems to be a relatively new development, with less support available, but the main website does list some amount of tutorials and examples as well. As mentioned, I had started with just Torch and decided to stick with that to learn the Lua language (and have more assistance available) rather than use Python [3].

IV. Implementation in Torch

More interesting than much of the background information on DNN's and Torch is actually applying that knowledge to completing and running programs. The first stage was to implement a feedforward network like the ones described in our deep learning textbook. To do this, after moving past the basic Torch and Lua tutorials, I turned to the dp package for Torch. This package, created by Nicholas Leonard while working in the LISA Lab, is extremely useful for any applications of deep learning in Torch. It allows for easy access to many datasets, including MNIST which was a main focus of the course.

Additionally, it provides a nice tutorial for setting up a DNN using the ideas of a feedforward network, which I was able to modify for my own purposes. To begin with, the user would install the dp package using the appropriate command for Mac OS or Ubuntu, and also tell Torch to require the package before trying to use any of the included commands. There are then a long list of options for modifying what exactly the program does. While these could just be set individually as needed throughout the program, it is much cleaner to specify them all ahead of time. There are only a few of these that seemed to be particularly relevant - mainly hiddenSize (changing the number of hidden units per layer), batchSize (which determines the number of examples in each batch of training data), maxEpoch (the max number of epochs/times to run training), and dataset (normally chosen to be MNIST, but can use any of the provided ones as necessary). Others can be used depending on the goals of the person using it, especially if using a GPU to provide more processing power, which would open up more options without greatly increasing run time [5].

The next step is to create the model for the program. After naming the model appropriately, layers can start being added into the model to create the full network that is required. First up is an input layer, which in this case is automatically determined by the size of the images (MNIST digits) that were chosen. Based off of the input, a hidden layer is then implemented that takes the input data and, through a linear process, outputs new data of the size specified at the very beginning. We then use an activation function, in this case Tanh, to further process the input. This process can be completed many times if a truly deep network is required, with the input size of the following hidden layers matching the size of the previous output. Next up is the output layer, with an input size corresponding to the output of the hidden layer (or most recent one if multiple are used). For MNIST, we are interested in the particular digit displayed in our input image, so our output will be one of ten classes, also using the LogSoftMax function to determine a guess at the actual digit - the guess with the highest percentage is the one assumed to be correct. After finishing the model, what is known as a propagator should be created. The basic idea of these is to propagate a data sample through a module. First off, a decay factor is determined from some of the variables that were originally set. An adaptive decay can be used to cause the learning rate decay to change over time, or it can just be a linear decay value. Following this the trainer of the model should be initialized. The loss is printed for testing purposes, as well as the current epoch number and the learning rate (after adjusting for the decay). The training, validation, and testing accuracies can then be calculated to be displayed after each time through the process. These are the values to be observed to determine how well the model performs; I was obtaining results between 90 and 95 percent until I added more than 6 or 7 hidden layers, at which point the errors and run time both increased dramatically. All that is left at this point is to define the experiment (mostly determined from more of the originally set values). The entire code can be saved as a .lua file to then easily be called from Torch as necessary [5].

A similar process can be undertaken in order to create a CNN in Torch. After requiring the dp package, we again specify selections on a number of options. In addition to many of the same choices as detailed previously, there are some others of importance related to convolution. Decisions must be made on the desired output size from each layer of convolution, the size of the kernel and the stride used (at each convolution layer), and the size and stride of any pooling that is done (also at each layer). We can implement additional options such as zero padding or dense hidden layers after convolution; I experimented with these and found them to be too much for my setup to handle. There are a few recommended preprocessing options as well, but again I found these to be mostly unnecessary and causes of increased run time. The data is then loaded in - presumably MNIST again, but many others are readily available. The model can be created next, which follows a similar procedure to the one followed for feedforward. To begin with, the input size is determined based on the dataset chosen. Then a layer of spatial convolution (for images at least) is added into the model, based off of selections made for output channel size, the first set of kernel values (both size and stride), and possibly zero padding if selected. We will then insert a layer of an activation function, commonly chosen to be ReLU for CNN's. Next up is a layer of pooling, which is again based on the originally specified values. This whole process then repeats for the second set of layers. Based on the chosen images the output and input sizes are determined; the main importance of this is that the user must ensure that the input is not processed so much that the data is reduced to nothing, an error I encountered when not actually calculating the resulting outputs from convolution and pooling. The output layer can then be established just like in the feedforward network. The propagator and experiment also follow a basically identical setup to that described previously. At this point, all that is left is to save and execute the file. Some fine tuning of the initial parameters can be done, and after that very strong results can be obtained, around 99 percent with just 5-10 epochs of training and without utilizing features such as zero padding [5].

Likewise, there is a similar process to those explained previously when it comes to setting up an RNN in Torch. The dp package is still needed, along with another package called rnn that will need to be installed and required. Many of the initial parameters are the same as before, but some new ones will need to be decided upon. LSTM can be used instead of basic recurrence, the network can be bidirectional, and zero can be set as the initial bias as opposed to having a good bias value learned ahead of time. Because of how RNN's work, the dataset choices are completely different - these contain massive amounts of text, with names such as BillionWords, PennTreeBank, and TextSource. Of course, this also greatly lengthens the time it will take for the model to run; my first attempt to execute the program met with an error due to a lengthy function at output time, so unless being run on a GPU, the softmaxtree option should be selected to take the place of softmax. In fact, after a few trial runs I was still unable to obtain any results from my RNN training - even after leaving it running overnight, pretty close to a full 24 hours overall. Thus, it is highly recommended to use a GPU for all but the very simplest RNN's, although perhaps these can run more smoothly on other packages such as TensorFlow. The rest of the process is largely identical to setting up a feedforward network and CNN. We do now utilize nn.Recurrent and nn.Sequencer for setting up the recurrent layer(s) of the network. Additional layers and options result from the initial parameters as always [5]. After failing to run an RNN on a significant dataset, I switched my focus to a very simple program, borrowing ideas from a different example created by Nicholas Leonard. The dataset for this new network is simply the sequence of numbers from 1 to 10 repeating many times, with the focus being for the RNN to successfully predict the next number in the sequence after training. Parameters for this program are basically those that have been described as essential thus far: batchSize, rho (a sequence length for each element of our dataset after creation), hiddenSize (size of the inputs and outputs to a hidden layer), nIndex (location in a lookup table), and lr (learning rate). After selecting these within the constraints of the program, it is ready to run for 10000 iterations. This process has a minimal run time and quickly goes from an upwards of 15% error to a less than 0.1% error. Obviously, this is not a particularly impressive network and does not take advantage of the true power of a RNN, but it does demonstrate their versatility and generally usefulness [6].

V. Conclusion

I still have a lot to learn when it comes to using Torch and developing models of neural networks. It was frustrating first setting up Torch to some extent, and doubly so when it came to the RNN's that just would not train and execute within a reasonable amount of time on the hardware available to me - at least for networks of significance. However, I still feel like this is a meaningful conclusion for the course. I have come away with a good understanding of feedforward (sometimes with backprop), convolutional, and recurrent neural networks, and of how to actually implement these in Torch. Many other interesting topics were covered in the classroom that I was not able to touch upon here. This was a good starting point, but the work could be expanded upon in a few ways, such as obtaining better hardware to really see the benefits of RNN's, trying the networks on other datasets, or even compiling a dataset of my own for future testing. Deep learning may not be the basis for my future research, but this was still a good learning opportunity and I believe the skills and knowledge acquired can be useful for other applications of interest to me.

Appendix A

Please visit https://github.com/byett/Deep-Learning-Independent-Study/tree/master for the programs used to test feedforward, CNN, and RNN on Torch. The code seemed too unwieldy to include here.

References

[1] Bengio, Yoshua, Aaron Courville, and Ian Goodfellow. (2016). *Deep Learning* [Online]. Available: http://www.deeplearningbook.org

[2] Brownlee, Jason. (2016, July 14). *8 Inspirational Applications of Deep Learning* [Online]. Available: http://machinelearningmastery.com/inspirational-applications-deep-learning/

[3] Chintala, Soumith, Ronan Collobert, Clement Farabet, and Koray Kavukcuoglu. *Torch* [Online]. Available: torch.ch

[4] Karpathy, Andrej. (2015, May 21). *The Unreasonable Effectiveness of Recurrent Neural Networks* [Online]. Available: http://karpathy.github.io/2015/05/21/rnn-effectiveness/

[5] Leonard, Nicholas. *dp Tutorials* [Online]. Available: http://dp.readthedocs.io/en/latest/#tutorials-and-examples

[6] Leonard, Nicholas. (2016, March 8). *Element Research/rnn* [Online]. Available: https://github.com/Element-Research/rnn/blob/master/examples/simple-recurrent-network.lua