Neural Networks and Machine Learning

Project 1, Part B: Literature survey, tools choice, and innovations

Robert McCartney, Jie Yuan, Sagar Barbhaya

March 3, 2015

2 March 2015

1 Introduction

As described in Part A of this report, our team's chosen data deals with various types of images of handwritten symbols. In one dataset these symbols are Hindi, in another it is Tamil and Telugu, which are types of Dravidian languages, and in the third it is handwritten digits, of the type often seen in zip codes on the front of envelopes. While these datasets contain very different types of characters, they also share a lot of similarities. All three all comprised of primitive pixels at the most basic level. All three can be used with image processing techniques to extract local and global features from these images. And all three can be normalized and preprocessed in the same way, as described in detail in Part A.

Thus, these datasets, while differing greatly, also provide a common framework from which to approach the problem. Thus, our team began its research with this data in mind, looking for ways to segment, parse, and classify the underlying characters while being conscious of underlying resource usage and training time required. In the following sections, we identify what work has been accomplished by others in the field, comparing and contrasting their results to our own problem in the search for useful techniques and ideas. Further, throughout this literature review we examine two approaches for overcoming the limitations imposed by finite resources on the learning process. One common approach is to use distributed processing of neural architectures, especially parallel computation, in order to speed up the learning process. In [papers], you see they created architectures on one or more GP-GPUs in order to conduct massive learning across hundreds or thousands of cores. While this was large scale, it was certainly not low energy and it required lots of interprocess and inter-thread communication in order to train the networks properly. A second approach, seen in [PAPERS], is to use forms of partial connectivity, dropout, convolutional layers, and regularization of weights to reduce the amount of inter-neural communication, when compared to fully connected artificial neural networks. This approach is theoretically enticing, as reducing the number of connections can significantly speed up training. It can also be shown from [PAPER] that using dropout on a single network is equivalent to training many different networks in parallel on subsets of the data and then combining their results, so it crosses the divide between these two methods of massive parallelization and partial connectivity. This is a method we are very eager to explore on our three datasets. In the next section, we go in depth on the research that we conducted and the literature that we reviewed. We examine each for positives and negatives of the methods used, with a constant eye on reusing good methods in our own research. In the next section, we go over some of the tools and environments we will be using for the project

2 Research Analysis

2.1 Online Recognition

In [1], the authors discuss their research on online recognition of handwritten words, which has obvious applicability and similarity to our own chosen datasets. From this report, we took away several critical issues. First, any interpretation of handwriting is an issue because the handwriting of every person is unique. In fact, each person has a mix of several different styles of handwriting. This makes it difficult to decipher individual characters, much less to parse these characters into semantically meaningful words and paragraphs. In fact, the problem gets even more difficult if you consider that a single individual need not have written the entire input sequence, so that the same character separated by a few lines of text could look very different and contain different types of features, such as their lines, edges, angles, and rotations. Thus, it is very important for us to devise a technique which will help us analyze and recognize all of these differing styles, in order to ensure the best results possible on unseen instances in the test set.

From reading this paper, we have come to the conclusion that any system will need 4 parts to it, as a sort of pipeline, in order to break the recognition task down into smaller chunks. The steps are

1. preprocessor that normalizes each input, using the EM algorithm.
2. second is the model that converts this into an image form (the penned words are converted to an image)
3. A neural network takes this annotated image and spots and recognizes the characters in it.
4. Hidden Markov Model (HMM) helps interpret these results and (takes into consideration word-level constraints).

The next step is feeding this modified input into the neural network, which helps identify letters in the word that are hidden and finally comes the HMM which takes the output from these neural network which has made sure that it identifies even the hidden letters and then tries to interpret them.The parts of this system that make it more robust are the combination of the neural networks and the Hidden Markov Model which make sure that no letter is missed and thus makes the interpretation of the word for the model more accurate and precise.This is very crucial for our dataset as the recognition of numbers should be very accurate. Even one wrong prediction of a number is going to affect the entire result. It cannot be an “assumption” or a “guess” at any point.

There have been techniques which basically use the pen movements (trajectories) to conclude on the handwriting but often these are affected by many other factors and hence will have its own drawbacks. Thus what this method does is takes a picture of the handwriting (the preprocessor functionality).There are 2 ways in which the word can be broken into letters, in segmentation and out segmentation. It basically is determined whether at the time of reading the word, you do it as a single entity or a combination of different letters. The latter is called in segmentation and the former is out segmentation. The Hidden Markov Model helps extract words later.Let us first see what inner segmentation is about. It basically takes input of segmented characters and then there is a post processor that works on these. This is something like the HMM model that we talked about. Thus, the Neural network phase is the one that can either be done like an in segmentation or out segmentation.In Out segmentation what happens is the segmentation process happens later in the cycle, basically around the time of word recognition.

An approach used to capture this image of the word is called AMAP where basically for every letter, the orientation , curvature, inclination of the letter is studied and then using functions like the Gaussian function, the curvature of the word is calculated. These images work best with neural network techniques. The main function of a neural network technique is to note only noticeable changes in the words or letters and spot them.It produces a series of outputs that go into parsing each of the letters in the word. This technique did not work as efficiently for extremely large or small inputs. But it performed better than other techniques like the ones using only segmentation. This is because there was a lot of information available on the curvature, angles of each letter in the word.The neural networks and AMAP input combination resulted in great results, independent of any writer. Also, they help reduce a lot on the error rate in the bigger picture.Thus basically, neural networks along with a combination of other techniques will help improve efficiency and effectiveness of handwriting recognizing patterns a lot more than these techniques on their own. This will help make the system more robust.

2.2 Choosing a learning algorithm

The next paper that we investigated was [5], in order to help us choose the learning algorithm to best suit our dataset. That is, we are looking for algorithms that perform well on normalized images in gray scale, which we decided to use in order to reduce aliasing and to minimize the number of inputs required (which would be multiplied by a factor of three when using RGB). The first classification method discussed in this paper used a linear classifier, in which the output of the model is based on a weighted sum of the values of the input pixels. This is the most simple form of modeling the data, but is not very robust as the datasets become more complex. Thus, given the complex features and varying shapes a digit or character can take, we do not expect this method to perform well and will not investigate it further in our research.

An improvement to this technique is to be able to recognize different classes at each layer (introducing layers increases distribution and it causes a slight decrease in the error rate) but it does not guarantee that the character will be recognized accurately since the error rate predicted is still quite high.We need a robust(very low error rate) method considering that an “estimate” or “guess” for any digit/character can cost us a lot in the end result.

The next method investigated uses Euclidean distance, namely k-nearest neighbor, in order to cluster the input data. Once clustered, you can find the nearest centroid to an unknown input image in order to decide on a classification. This handles complex data effectively, but unfortunately not very efficiently. Given the size and dimensionality of our datasets, this would result in a large amount of memory and resource consumption. Further, given the so-called curse of dimensionality, it can be hard to cluster data accurately when it is highly dimensional, since the likelihood of any two input images being close together is small when there are lots of features to consider. This curse leads to the next important method that we considered, which is a form of dimensionality reduction. Reducing the number of features used can lead to improvements in the classification task. The authors of [5] discussed the method of principal component analysis and polynomial classifier which has a preprocessing stage attached to it which we discussed while discussing handwriting detection techniques helps normalize the input for us. They are calculating a covariance matrix to give as a input to neural networks.

Next is a radial basis function which has 2 layers unlike the linear classification method, the first layer consists of Gaussian units as discussed in the processor stage of handwriting detection techniques, each of the units is trained using k-means algorithm and the second layer uses pseudo-inverse method to compute weights. The multi-neural network classifier uses the 2 layers of neural networks as well but one of the layers is hidden. The LeNet 1 learning algorithm is a mean between the simple linear algorithm and the multi-layer algorithm. It is basically designed taking in to consideration the 2-dimensional nature of digits. The concept of convolution network is used here where in every layer takes the input from one layer below it and this can be done using receptive field on each layer. Thus, more relevant information, local to the digit is obtained. All units close are clubbed together and have their own feature maps. The grouping of these units helps distribute weight amongst the network and also decreases the number of free and unclassified inputs. The best part is that these layers or convolution layers as they are called can be managed. We can add how many ever we want till we feel the calculation is robust and efficient. We can decrease the variation and sensitivity t handwriting changes by using lower resolution feature maps at higher convolution layers. All images are gradients.An improvisation to LeNet 1 considering the large number of convolutional networks it requires is LeNet 4. This has elements distributed based on their center of mass. Thus it can accommodate more feature maps and has hidden layers which are well connected to all the layers and still do not cause the network to become too huge. LeNet 5 then comes with a more distributed environment, reducing the error rate down to just 0.9%.

Now comes the one whose error rate is just 0.7% and probably the best one to use for digit recognition. It is basically a combination of three LeNet 4 classifiers. Thus it is called the Boosted LeNet 4.The first Le Net 4 is the one trained in the way same to the normal LeNet 4. The other is the improvisation based on the first one, the patterns that are seen in the first net, half of them being the correct ones and half of them being wrong ones.There are a few patterns which are not agreed upon by any of the first 2 Le Nets. These are the patterns that are left out of the set. These are the patterns that are grabbed by the third Let Net 4. We can combine the outputs of all these 3 Le Nets by simply adding the outputs from each one of them to each other.

There are a few challenges while developing this also. The Le Net 4 has an extremely low error rate, in order to make sure that we have ample of data for pattern studies in the other 2 Le Nets, we can introduce distortion and get samples for the other 2 nets. It looks like we are paying the price of 3 nets and what about the cost of the process? But as we can see, if we get efficient results from the first net, with high confidence rate, we do not even call the other 2 nets. SO it does not really come out to be thrice the price of the net. Comes to somewhere around 1.75 times the original one with a error rate of just 0.7% which is a fair enough deal especially for datasets like us where we aim to have almost 0% error rate considering that small error rates also can result to a wrong analysis and prediction if not curbed at the smaller levels.

2.3 Generative models

After these overview papers on different effective methods for classification of characters and handwritten digits, our team next looked to current research in the area of generative models [3]. One of the biggest downsides of traditional artificial neural networks is the fact that the input features must, in general, be selected by hand. That is, if you use a one-hidden layer neural network you will have to craft features from the images to use as input into the model. When dealing with image data such as the three datasets that we have chosen, finding good discriminative features to use can be the most difficult part of the entire process. Of course, there are techniques from computer vision for this, such as finding lines, angles, edges, curves, blobs, or other correlations between pixels. The problem with this approach is that you must decide before learning what is important and what is not, crafting by hand features that you hope can fully discriminate between every output target. Otherwise, the model will not be able to learn much from the raw input data. For the MNIST dataset with the output classes of 0-9, hand selecting features would probably work to a certain extent, but when dealing with more complex data like the Hindi dataset this task becomes much more complex.

As an alternative, we have explored methods for automatically extracting features as input into the model, which are known as generative models. Generative models allow the algorithm to learn an effective mapping from visible to hidden nodes that helps it to reconstruct the input data. By being able to reconstruct the input, it can be said that the model has learned what "makes up one of these characters, without ever being told what to look for or how to approach the problem. This is a form of unsupervised training, as the models can learn such patterns and features from the data without ever needing to be told what class the data belongs to. One of the most popular and effective forms of such unsupervised feature extraction is called Restricted Boltzmann Machines (RBMs). They are restricted in the sense that in general Boltzmann machines can have an arbitrary number of layers, but by restricting them to just two layers then methods exist for efficient, parallelized training.

With theory based in physics on modeling the energy of a system, RBMs learn connection weights between visible and hidden units that is roughly comparable to the brains ability to turn certain neurons on together. That is, the theory can be summarized by the famous quote "neurons that fire together wire together. Thus, RBMs can take input data and automatically learn nonlinear features that can generate the data it has seen in the past. It does this by modeling the connections between hidden and visible neurons as a bipartite graph with symmetric weights. There are some issues with learning how to train RBMs correctly, especially in setting the meta-parameters like learning rate, momentum, weight-cost, sparsity, initial values, number of neurons, and sizes of mini-batches. In [XX], the authors addressed setting this parameters for good learning results, which is directly applicable to the some of the models we will be training on these datasets. Good results can be defined as learning a good energy function between visible and hidden nodes. This training is done through contrastive divergence (CD), where you perform Gibbs sampling for a few steps of back and forth activations in the visible and hidden units, then apply the following equation to update all weights in the network:

$$\Delta w_{ij} = \epsilon(< v_i h_j >_{data} - < v_i h_j >_{recon})$$

As an optimization for CD, the authors mention that is important to sample the hidden units in the early rounds of Gibbs sampling in a binary fashion, turning them on based off the activation probabilities rather than using the real probabilities directly. This acts as a form of regularization to the model, which could otherwise learn to perfectly fit the data without learning good features to model the data. However, on the last update to a hidden unit it is ok to use the real-valued probabilities, as they are not communicating anything back to the visible units, and this reduces sampling noise. The same goes for visible units, which can make use of the probabilities rather than sampling a binary state, as it reduces sampling noise and allows for faster learning (although somewhat less effective learning, as well). Using mini-batches of 10-100 cases allows for efficient matrix-matrix multiplies that stochastic updates cannot match, and dividing the gradient by the size of the mini-batch ensures it is normalized for any given learning rate, but you must ensure that each possible output class is equally represented in each batch.

Overfitting can be monitored by looking at the free energy of a validation set in comparison to the training data, where the same subset of training data should always be used. This can be computed in linear time and gives better estimates than squared reconstruction error, which is not necessarily a sign of good learning. Decrease the learning weight towards the end of training, and you can average weights across several updates to reduce noise. Weights can first be chosen using a Gaussian with zero mean and variance 0.0001.

Initial bias can be set to log(pi/(1-pi)) where pi is the proportion of activation of unit i in the dataset. Can encourage sparsity by setting the hidden biases to a large negative number, such as -4, but a hidden bias of

0 works fine. Using momentum on the gradient with a starting parameter of 0.5 for the previous 'velocity' then increasing it gradually over training to 0.9 is a good way to increase the speed of training. It is equivalent to using a larger learning rate, but delaying the effect of the gradient estimate, reducing unstable oscillations that could result. Weight decay through regularization, where you add a term proportional to the derivative of the squared sum of all weights into the gradient, helps reduce overfitting and prevents units from getting "stuck in an on or off state. Further, with small weights the Gibbs sampling procedure is more effective. The authors recommended weight-cost coefficients in the range of 0.01-0.00001, without applying this step to bias weights.

One of the most interesting ideas presented by the authors is that of a sparsity target, or the desire to keep hidden units turned off and rarely active. They posit this leads to better and more localized features, since the each neuron must specialize into an activation pattern for specific features rather than staying on constantly. This can be accomplished by maintaining an estimate of activation for a hidden unit, q, that is updated each mini-batch, where $q_{current}$s is the average activation of this node on the current mini-batch, using lambda between 0.9 and 0.99:

$$q_{new} = \lambda q_{old} + (1 - \lambda) q_{current}$$

The penalty for activation becomes the cross-entropy error, namely

$$p * log(q) - (1 - p) * log(1 - q)$$

where p is the sparsity target, usually between 0.01 and 0.1. This derivative is q-p, which is added to the derivatives of all weights and biases. As an extension to contrastive divergence, rather than using CD1, which consists of Gibbs sampling for 1 full round of visible-hidden-reconstruction-hidden, more rounds of sampling can be used as time goes on in order to better approximate maximum likelihood once the Markov mixing rate declines (which happens with larger weights). As n in increased, however, it is usually a good idea to decrease the learning rate.

It is important to be able to visualize what is going on when training an RBM, due to the complexity involved and many different parameters that must be fine tuned. To do so, the authors recommended either using histograms, or displaying the weights visually as pixels of an image. This latter method is something that we will be employing with our project. Visual images give immediate feedback on what the network is learning, and convoy much more information concisely than other forms of feedback. In fact, every input vector can actually be graphed as an image of hidden node activations and displayed together on a single page (assuming a small enough dataset), giving immediate recognition to what hidden units are activated by what type of input data.

One we we would like to use this research in our project is to use RBMs as a form of pre-training of our model. They are trained in an unsupervised manner, so that we can learn features from the data without needing to consider the underlying classes. Once features are learned, usually by stacking RBMs on top of one another, then we can consider the entire network to be a deep neural network. In this way, RBMs have initialized the weights of the network to capture useful patterns, and gentle backpropagation can fine tune the network for classification.

2.4 Extension to RBMs

Given that our group wanted to explore RBMs as a way to generatively pre-train our image classification models, we also investigated extensions to RBMs and other ways to use them. One such extension was the Spike and Slab RBM (ssRBM) [2]. With the ssRBM, the hidden units are turned into two parts - one is a binary spike, and the other is a real-valued slab. Using this representation, the data can be modeled with Gaussian distributions, as opposed to the binary representations used in traditional RBMs. To train these models efficiently, the authors added a third step to Gibbs sampling to deal with the fact that they no longer have conditional independence between units in the same layer, which was what made RBMs so easy to train quickly. The solution is to first sample from the slab distribution, p(s—h,x) and then sample from p(x—s,h) in order to approximate the full distribution of p(x,s,h). With this method, contrastive divergence training continues much the same way as before. As features learned by the model, both the expected total activation, h*s, or just the binary activation, h, can be used, the advantage in the latter case being that it is invariant to pixel intensities and thus normalizes factors such as illumination and contrast in an image.

This was a very interesting model to explore, given how it can model covariance across inputs, an important component given that pixels are highly correlated in their local regions. Further, the authors showed how on the same dataset we will be using, MNIST, that the ssRBM filters appear more diverse and sharp, able to capture both local and global structure over traditional RBMs. However, on our dataset ssRBMs best performance of 1.5% validation error and 1.3% test error was only a few tenths of a percent better than traditional RBM performance. Further, average performance for both was about the same over 20 iterations, with ssRBMs also having the worst single performance of 1.95% on the test set. Given the amount of added complexity, after reviewing these results we decided not to implement ssRBMs on our datasets. Instead, we will explore ways to make traditional RBMs better, including the use of dropout, which is a form of stochastic partial connectivity that we explain below in the section on novel ideas that we will experiment with in this project.

2.5 Convolutional Nets

Another form of partial connectivity of weights in a neural network is known as convolutional neural networks (CNNs). These are biologically inspired by the visual cortex, having overlapping feature maps in the same way that there are overlapping regions in the visual field. These feature maps combine at pooling layers, then are sub-sampled and applied with more feature maps in several hierarchical layers. After these convolutional layers, which act as feature extractors, the input is fed into several layers of fully connected neurons, similar to traditional ANNs. These pooling and extraction layers have been compared to the way our brain builds up complex visual cues from combinations of small features that it receives on the retinas. At their heart, CNNs are effectively reducing the time it takes to train. That is, in contrast to a fully connected ANN, convolutional nets instead create massive amounts of weight sharing between different parts of an image, since every feature map in a layer shares the same weights with every other feature map applied to other locations on the image. This drastically reduces both training and propagation time, increasing the number of possible layers used (which often exceeds 6 or more). Given unlimited resources, a large enough dataset, and infinite training time, a fully connected neural network would probably lead to better performance, but with these constraints CNN provide a good trade-off between performance and practicality. This has led to their widespread popularity in recent years on some of the famous machine learning competitions, especially in the area of image recognition where they hold records for state-of-the-art performance results.

An added benefit beyond the reduction in the number of parameters required is the fact that it is possible to parallelize the training of CNNs, by virtue of the fact that only small segments of an input image are fed to each of the maps of a convolutional layer. Thus, like small jobs separated around the image, each feature map can perform its work in parallel and the results pooled before continuing on to the next layer in the network. This is a form of network-based parallelization, where the ANN is partitioned in sub-components. Common implementations map these sub-networks to blocks on a GPU, so that massive amounts of cores can be used for training. Of course, network-based parallelization requires more tightly coupled communication between the components to share updates and broadcast results, especially given that every feature map must maintain identical weights to one another. Nevertheless, this has given better results than forms of training set parallelization, where the data is partitioned to different networks.

In this paper, the authors go on to describe their network in detail [4]. They used five convolutional layers followed by three fully connected layers and a softmax output layer of 1000 classes, to train on the ImageNet dataset that consists of 1.2 million images of 1,000 different types of objects. In all their network had 650,000 neurons and 60 million parameters. To train such a net, the convolution maps themselves helped to reduce the number of connections and make it feasible on a dual-GPU parallelized implementation. Nevertheless, in order to train in a reasonable amount of time other forms of optimizations had to be used. The first was using non-saturating neurons, so that gradients would not go to zero for backpropagation. The second was using a method called dropout, in which you randomly drop connections between neurons. This acts as a form of regularization, since neurons must learn meaningful information on their own without relying on always having connections to other neurons with which to share information. This reduces overfitting and is in fact another form of parallelization. This can be seen by the fact that each stochastically chosen pattern of activations is in fact a unique network, with there being a total of 2^n different possible networks of this type with n neurons being training. In this view, each network might only get a single instance that it is ever trained on, but every single one of the 2^n networks shares the same weights between one another to allow for effective learning to occur. Thus, under this different view of dropout, we are in fact training 2^n different networks in parallel and averaging their different results into a single model. At testing, all neurons are used but their outputs are multiplied by 0.5 each, as a way to approximate the geometric mean of the exponentially-many different networks. This proved to be a very effective technique for the authors, and is one that we will explore more in our own research.

Another interesting technique they used was non-saturating nonlinearity. Instead of using $tanh(x)$ for activations, they used $f(x) = max(0, x)$, called Rectified Linear Units (ReLUs). These have been shown to train several times faster than tanh or sigmoid activations in CNNs, since they do not need to have input normalization to prevent saturation, allowing the authors to train longer than if they had used more traditional activations. While training on the dual GPUs, the authors dropped certain layers of communication between the two halves of the total network, restricting communication to only after layers 3 and 5. This allowed them to precisely tune the amount of communication to be an acceptable fraction of total workload, and as a result they saw their error rates improve by almost 2%.

The final major technique the authors used was to artificially enlarge the dataset. The first way they did this was by using class-preserving transformations on the underlying images. By using translations and reflections, then extracting patches from each image, they increased the amount of data by a whopping factor of 2048, a huge increase considering the size of the dataset to begin with. The second way they increased the amount of data was with a technique we encountered earlier for dimensionality reduction, PCA. By using the principal components they altered the RGB channels in the training images, thereby teaching their model to be invariant to changes in intensity and color of illumination inside an image. As we are training our models, if we find that the models are overfitting the data we will use some of these same techniques to increase the size of our dataset and thereby provide greater learning potential to the model.

3 Tools and Frameworks

For this project, we will be using MATLAB to pre-process, analyze, classify, and visualize our data. In other words, we will be using it for almost all of the pipeline that includes segmentation, classification, and parsing tasks. We chose this language because of its high level facilities and ease of use, especially when it comes to scripting new things and trying out different changes to the model. Further, its support for vectorized operations aligns closely with neural networks, which are really just giant matrices of weight parameters that we are trying to learn. As for tools, the majority of the code will be written by us, in pure MATLAB. It is relatively easy to write the code to load and pre-process image data. Further, it is not difficult to write a basic neural network with feed-forward and back propagation steps with matrices. As for more advanced functionality, we will be using some code provided by other researchers in the field. Dr. Ray Ptucha, professor of Computer Engineering here at RIT, has graciously shared some of his code for Motion History Images and down sampling, which we can use to process the image files into other forms. We will use MatConvNet [6] as our implementation for convolutional neural networks, rather than trying to write their complicated functionality ourselves. Dr Geoffrey Hinton, research from University of Toronto, has published online his code for loading data from the unique file format of MNIST, as well as his code for training Restricted Boltzmann Machines. Dr. Hinton was the first researcher to discover the efficient contrastive divergence training algorithm for training RBMs, and his code base is widely used by other researchers. Thus, we are restricting ourselves to a highly functional and usable framework with MATLAB, and to code from other highly regarded researchers in the field in order to ensure that we are using state of the art packages and cutting edge technology.

4 Innovations to be explored

Here we propose a set of four innovations that will be investigated as refinements of our initial neural network architecture. Many of these are inspired by or invented to address the recommendations for efficient backpropagation presented in LeCun et al. 1998.

4.1 Innovation 1: Boosting of small neural nets with weighted stochastic sampling

LeCun et al. state that stochastic learning provides a number of advantages over batch learning, namely that training on single data points per iteration is much faster than passing through the whole data set, and surprisingly, stochastic learning also tends to result in more desirable models, because though this random selection will still tend to represent the underlying distribution of the training data over many iterations, the added randomness at each step allows the model to escape local minima. A completely random selection is not a strict requirement of stochastic gradient descent, however, as LeCun et al. also state that the more dissimilar successive training examples are, the faster the model converges. Intuitively, this makes sense, because similar training examples will tend to modify the weights in the same direction, and if this happens back-to-back, then the later updates have smaller errors, and thus reduced effectiveness on the weight modifications.

To address this LeCun presents two suggestions - select successive points that don't belong to the same class, or for points that produce greater training errors, increase the probability of selecting them for training later. We found the second suggestion particularly interesting, as this is the same motivation behind the popular AdaBoost algorithm, which works by creating a number of weak learners which vote on a highest probability classification. The weak learners are trained one by one on the training data, and one of the unique features of AdaBoost is that training examples that were incorrectly classified by the previous weak learner are weighed more heavily in training the next learner.

To adapt this idea to stochastic neural network training, each data point in the training data will have an associated probability of being chosen for the next backpropagation iteration. Data points which are misclassified will be updated with a much higher probability, so that they will be more likely to be chosen next, as described by LeCun et al. Additionally, to discourage the same data point from being selected twice in a row, each time a point is selected, its associated probability will significantly decrease. All probabilities will be renormalized after each iteration.

There are also ways to extend this methodology to incorporate parallelization, which is one of the main goals of our research. Although classic AdaBoost involves generating each new weak classifier based on the results of the previous classifier, this design can be easily parallelized. For example, boostrap aggregation could be applied - the data set could be broken up into N subsets, and a collection of weak classifiers could be trained independently on each subset. After the training is complete, the collections can be evaluated in parallel, and the final result would be calculated as a vote among all of the N boosted models. Given a base algorithm for a single neural network, AdaBoost is not difficult to code, and our goal would be to evaluate the boosted weak learners against a single well-trained neural network.

4.2 Innovation 2: Selectively dropping nodes

Boosting methods are widely seen as effective alternatives to highly complex single classifiers because the introduction of randomness into the training process prevents overfitting of the training data. Following a similar reasoning, Srivastava et al. (2014) propose the concept of dropout in training neural networks. In a stochastic training process, when each new data point is selected to be trained, each hidden node in the network is dropped with 0.5 probability along with all of its incoming and outgoing weights - this results in a random subset of hidden nodes being dropped. The randomness of dropping nodes combined with a voting process afterward prevent the overfitting of a highly detailed model from the training data. The training occurs as before, and only the nodes that were not dropped get updated by the backpropagation process.

Extensions to parallelization: Multiple thinned models can be generated in parallel from different subsets of the data, and then combined to form a single model with weights that are a weighted average of all of the models. This gets around the impracticality of voting from potentially a huge number of dropout variations of a single network. Dropout would not be applied to this cumulative model. A weight contribution of 0 could represent cases where the connections of a given node was dropped.

Another related idea we may attempt is to set an independent learning rate for each weight, which may be selected randomly from some distribution. We predict this will have largely the same effect as random dropout, since some weights will be artificially modified more heavily for certain inputs. One potential simplification of this model is that this can be performed on a single neural network, since the learning rates themselves are not part of the model - in the case of dropout, multiple models need to be trained simultaneously

4.3 Innovation 3: Mixed sigmoid and radial basis function network

LeCun et al. (1998) also introduce Radial basis functions, which can serve as an alternative threshold function to the more traditional sigmoid or tanh functions - instead of taking a linear combination of inputs with the weights as coefficients, a radial basis function treats the weights as a mean and the input as a sample: the Euclidean distance of the weights and input vector is calculated and inputted into a Gaussian curve. This may be advantageous in our case because it models the distance of the input from a specific point in the input space, which can be highly analogous to the presence of pixels in certain regions of a 2D image. Since the threshold function for each node in a neural network is independent, we may also experiment with allowing some nodes to use sigmoid activation functions, and others to use radial basis functions.

4.4 Innovation 4: A non-constant learning rate

Another popular neural network refinement is the introduction of momentum, in which weight updates are not completely dependent on the error of the current data point, but is rather a weighted average of the previous weight change and the new weight change. During gradient descent this allows for the weight updates to jump out of local minima by not immediately responding to the gradient at a given update iteration. In addition to implementing this small change, we may also investigate a changing learning rate over time, similar to simulated annealing, in which the probability of random jumps is initially high to escape from local minima, but decreases exponentially as the training process progresses, so that gradient descent can settle down into a minima.

References

[1] Y. Bengio, Y. LeCun, C. Nohl, and C. Burges. Lerec: A nn/hmm hybrid for on-line handwriting recognition. *Neural Computation*, 7(6):1289–1303, November 1995.

[2] A. Courville, J. Bergstra, and Y. Bengio. Unsupervised models of images by spike-and-slab rbms. In L. Getoor and T. Scheffer, editors, *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, ICML '11, pages 1145–1152, New York, NY, USA, June 2011. ACM.

[3] G. Hinton. A Practical Guide to Training Restricted Boltzmann Machines. Technical report, 2010.

[4] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. Burges, L. Bot-

tou, and K. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[5] Y. LeCun, L. D. Jackel, L. Bottou, A. Brunot, C. Cortes, J. S. Denker, H. Drucker, I. Guyon, U. A. Muller, E. Sackinger, P. Simard, and V. Vapnik. Comparison of learning algorithms for handwritten digit recognition. In F. Fogelman and P. Gallinari, editors, *International Conference on Artificial Neural Networks*, pages 53–60, Paris, 1995. EC2 & Cie.

[6] MatConvNet: CNNs for MATLAB. `http://www.vlfeat.org/matconvnet/`. Accessed 3/1/2015.