Sequential Visual Learning

Daniel Weber

Grade 12

Table of Contents

Acknowledgements

This paper would not have been possible without the guiding hand of Mr. Michael Stancik. His advice was invaluable when starting this research paper and his commitment to my success despite the challenges that I faced was inspiring.

Problem or Need

The goal of this research project was to create a machine learning algorithm capable of learning basic visual features from a video, which is a chronological sequence of images. The features that the algorithm learns should be similar to those found in the primary visual cortex of the mammalian brain. In addition, the learning process of the algorithm should more accurately model the way the mammalian brain learns than current algorithms, primarily by learning in a dynamic fashion on only a few frames of information at a time. The majority of machine learning algorithms used for tasks involving visual input such as images have relied on precompiled static datasets to learn their features. While this approach is useful in applications like handwriting recognition and object classification, it is often computationally intensive and it takes large amounts of memory to store all of the images used for learning. Furthermore, the learning method of these algorithms does not accurately model the learning process of the mammalian brain, one of the greatest learning machines that we know of. The mammalian brain learns on a constant stream of visual data. As soon as light reaches the receptors in our eyes, that raw data is immediately processed by the primary visual cortex and sent down the neural pathways to various parts of the brain. Creating a machine learning algorithm that learns on a stream of data, like the mammalian brain, would significantly reduce the amount of temporary storage needed to process the necessary data and the algorithm would be able to adapt to new data. The ability to adapt would be essential for applications where the initial conditions are unknown or uncertain and learning had to be done on the fly, such as robots acting as first responders to natural disasters. This design investigation will attempt to create an algorithm that learns on a dynamic dataset and will learn features similar to those learned from algorithms that train on static datasets.

Background Research

The Mammalian Brain, Learning, and the Visual Cortex

The mammalian brain is made up of many neurons, which can be thought of as basic input/output devices and are the building blocks of the brain. The brain is a plastic organism, meaning that physical changes occur in the brain as a direct result of learning. For example, a piece of a memory might be formed by changing the strength of the connections between neurons or connecting two new neurons together. As of today we don't know the exact process for learning but there are several theories that exist. In the mid nineteen hundreds, two scientists conducted an experiment on the visual cortex of a cat's brain. The two scientists, David Hubel and Torsten Wiesel, recorded the activity of the neurons in a cat's primary visual cortex and showed the cat videos of black lines oriented at several angles and moving in different directions. They found that specific neurons were very responsive when there was a line with a specific orientation and motion on the screen. They found that there are orientation selective neurons in the primary visual cortex of the brain, and concluded that we decompose the world around us into lines and edges. It has also been concluded from other experiments that the primary visual cortex of the brain undergoes dramatic change at early ages. It was found that when a cat was only presented with limited visual angles at an early age, it was only able to identify those limited angles at a later age, making it effectively blind to all other angles (Segev, 2015).

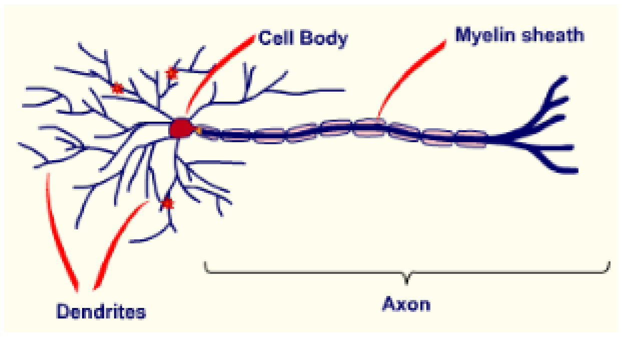
Machine Learning

Machine learning is a subdomain of artificial intelligence that is concerned with creating algorithms that learn functions from data. There are some computational problems that are very difficult to manually program and would be quite inaccurate if they were programed in that manner. Problems like these, such as object recognition and speech recognition, are best solved using machine learning algorithms (Schapire, 2008). The most common machine learning algorithms are created to learn an unknown function $f : X \rightarrow Y$ where X is a set of inputs and Y is the set of corresponding outputs (Mitchell, 2006). In a task such as speech recognition, each element in X would consist of a small segment of recorded audio and each element of Y would contain the corresponding syllable that was spoken in those audio segments. The goal of this speech recognition system would be to learn the function f that maps all$X$ to their corresponding Y values as accurately as possible. In machine learning, it is convenient to think of X and Y as vectors with the i-th element of each vector associated with x_i and y_i respectively. This is because most machine learning algorithms utilize fast matrix multiplication and linear algebra libraries to carry out their calculations. Other machine learning algorithms, called clustering algorithms, receive only inputs X without any labels Y and attempt to find some structure in the data by grouping related inputs (Mitchell, 2006).

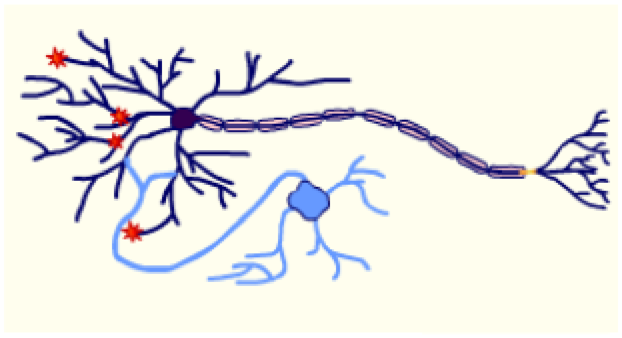
There are many other types of machine learning algorithms that are capable of solving some very challenging problems such as handwriting recognition, spam filtering, medical diagnosis, fraud detection and much more (Schapire, 2008). The reason why machine learning is often used to solve tasks like these is that people are very good labeling things but they often have difficulty describing precisely how they knew to give a certain object a certain label (Schapire, 2008). For example, if a person were shown a picture of a cat, they would be able to say that the picture contained a cat. However, when asked precisely how they knew it was a cat, what went on in their brain that let them know it was a cat, they would be unable to answer. As a result, when faced with a problem where the algorithm necessary to solve the problem is very complex, it is often necessary to have the computer come up with the algorithm while the person labels the necessary data (Mitchell, 2006).

Neural Networks

A neural network, sometimes known as an artificial neural network (ANN), is a type of machine learning algorithm inspired by the way neurons are connected in the brain (Stergiou, 1997). Several neurons are connected to another neuron by connecting their axons to the dendrites in the dendritic tree of the given neuron. The axons and dendrites are connected by synapses, which transmit the electrical information from the axon to the dendrite. The electrical signals from all of the neurons connected to a neuron accumulate in the neuron’s cell body until a certain threshold is reached, which is when that neuron fires an electrical signal down its axon and into other neurons through their dendrites (Segev, 2015).

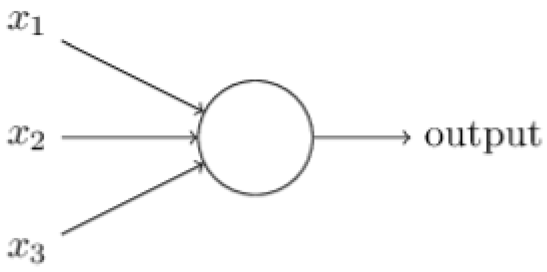


A neuron (Communication, 2011)



Two connected neurons (Communication, 2011)

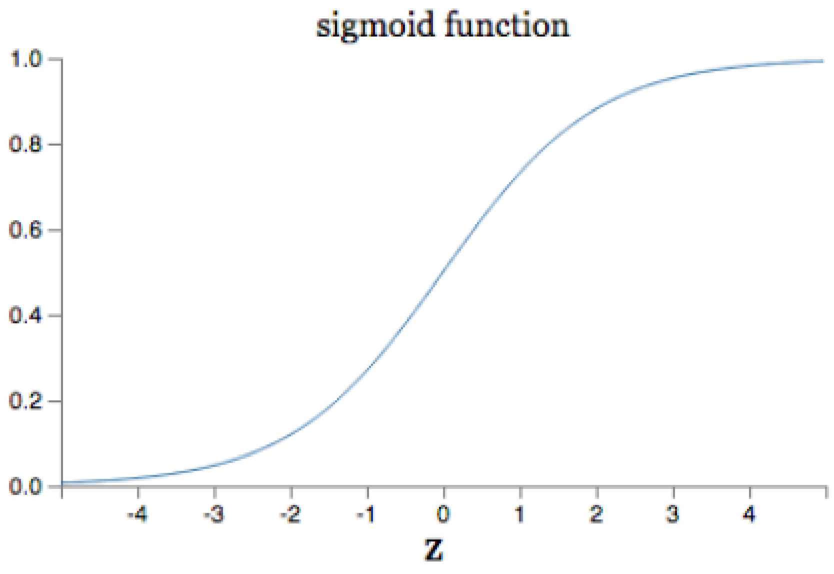
An ANN behaves in a similar way, but on a much smaller scale than an actual brain (Burger, 2004). At the core of an ANN is an artificial neuron which takes several inputs from other neurons, performs a calculation on those inputs, and then outputs a value based on that calculation (Nielsen). Traditionally, the inputs of an artificial neuron are represented as a vector X and the i-th element in vector X is labeled as x_i.



An artificial neuron (Nielsen, 2015)

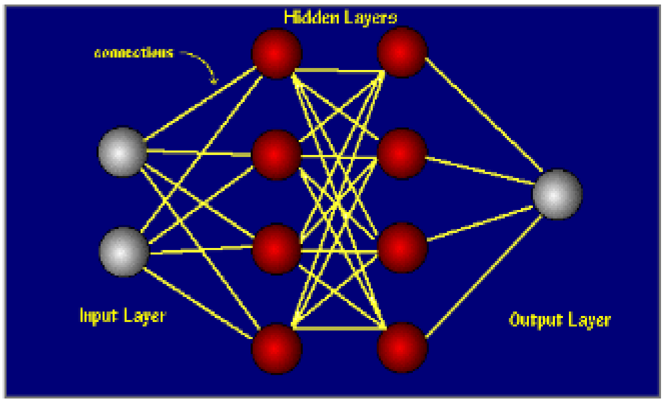
Additionally, there is a weight associated with each input element which represents the importance of that input to the output. The weights of the inputs are also represented as a vector called W, with w_i corresponding to the i-th element in W. The most common artificial neuron is called the sigmoid neuron, which passes the weighted sum of the inputs into a sigmoid function to produce an output between 0 and 1 (Nielsen, 2015). The formal equation for this is

$\sigma(\Sigma_i w_i * x_i)$, where the sigmoid function $\sigma(z) = \frac{1}{1+e^{-z}}$. The sigmoid function is often chosen because its value is always between 0 and 1 and because it produces a nice derivative, which is essential for learning. The sigmoid function is also the activation function for the artificial neuron and is one of many that can be used. It is also common for the sigmoid neuron to include a bias value, typically denoted as b, which represents how easy it is for the neuron to fire. With this in mind, the equation for the output of a sigmoid neuron becomes $\sigma(\Sigma_i w_i * x_i + b)$ (Nielsen, 2015). This structure is similar to how a neuron in the brain functions, where the the output depends on the sum of the inputs.

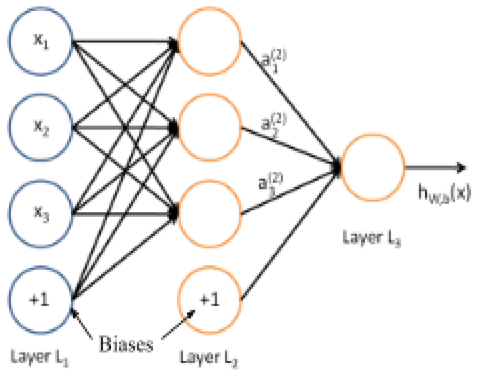


The sigmoid function (Nielsen, 2015)

When multiple sigmoid neurons are connected together they form an ANN. In most ANNs, these connected neurons are organized into layers. The first layer of an ANN is the input layer, which consist of neurons that represent the input to the ANN and simply pass on their information to the next layer. The next layer is the hidden layer, which can actually consist of more than one layer. Each neuron in each hidden layer receives weighted inputs from the previous layer, sums up those weighted inputs, applies the sigmoid function, and passes along the output of the sigmoid function to the next layer. The final layer is the output layer, where in tasks like image classification the values of the output neurons are treated as the probability that the given image belongs to a certain category (Burger, 2004).

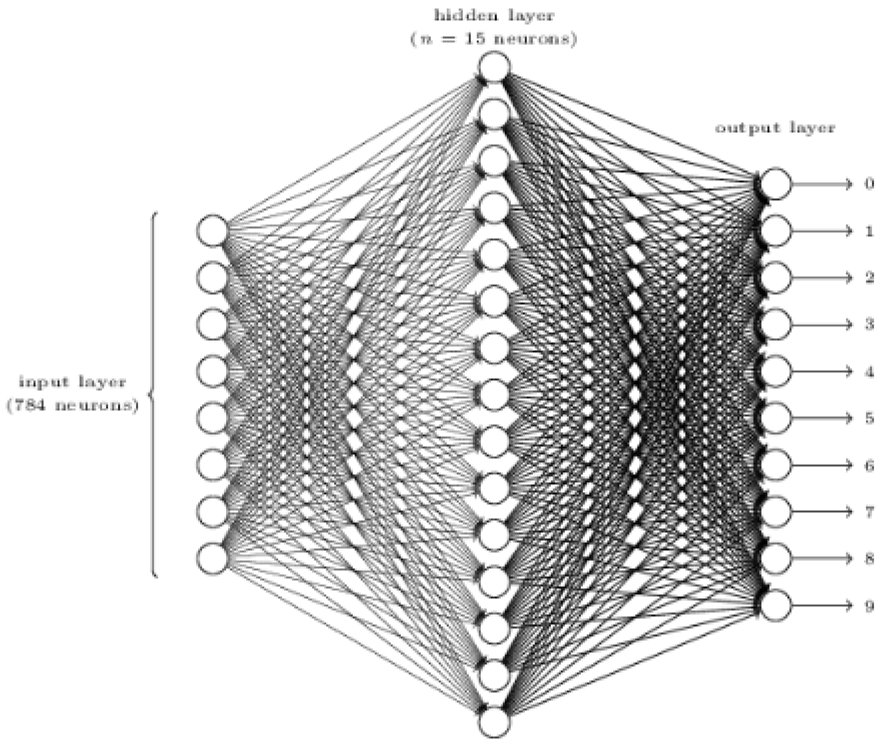


A small 4 layer ANN (Burger, 2004)



A small 3 layer ANN with biases (Ng, 2013)

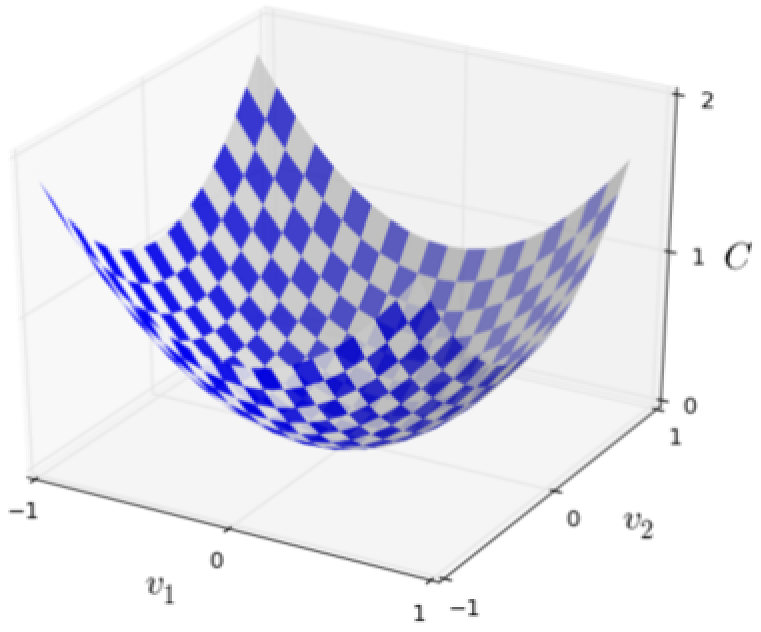
A simple example of an ANN is a neural network used to classify handwritten digits. This network takes in a grayscale image 28 by 28 pixels in dimension and outputs ten probabilities representing the probability that the network thinks the given image is an image of one of the digits from zero to nine. Therefore this network will contain 784 (28*28) input neurons, 15 hidden neurons, and 10 output neurons. The input neurons correspond to the intensity of each pixel with a value of 0 representing white and 1 representing black. The output neurons will output numbers between 0 and 1. If the 1st output neuron outputs a 1, then that means that the network is very sure that the image is a 0 (because the 1st output neurons corresponds to the digit 0) (Nielsen, 2015).



ANN for classifying handwritten digits (Nielsen, 2015)

The way we would train this network to recognize digits is by giving it many training examples consisting of a 784 dimensional vector representing the intensity of each pixel and a 10 dimensional vector representing the probability that the image is one of the ten digits. For example, if a training example was an image of a 3, then the fourth element of the 10 dimensional vector would be 1 and all other elements would be 0, meaning the probability that the vector is a 3 is 100% and the probability that the vector is another digit is 0%. The network is then trained using backpropagation, which calculates the error of each neuron (excluding the input neurons), and then updates the weights of the connections going into each neuron according to the partial derivative of the error with respect to the weights. The typical way to calculate error for a single training example is with the squared error function, which is $\frac{1}{2} * \Sigma||x - y||^2$, where x is the vector that the network currently outputs when given a training example and y is the vector of the desired output of the network for the training example. Because both x and y are vectors, we simply sum all of the elements of the resulting vector when x is subtracted from y to get a single number. When $x = y$, the error will be 0, so we are essentially trying to minimize the error (Nielsen, 2015).

A good way to think about how a neural network learns is to think of it as an optimization problem where we are trying to minimize the error of the function $\frac{1}{2} * \|x-y\|^2$ in terms of x, which depends upon the weights in the network. If the network consisted of only two weights v_1 and v_2 and we let the value of the error function be equal to C, then we can model C in terms of v_1 and v_2 like so:

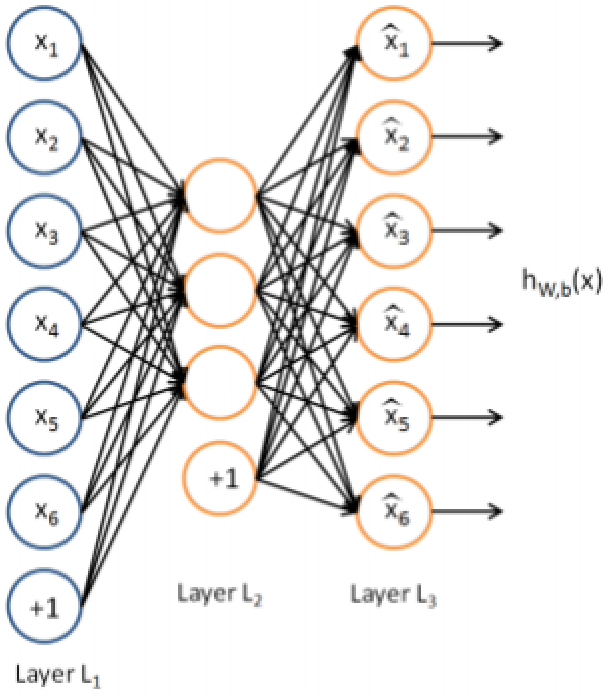


(Nielsen, 2015)

The reason that the function is shaped like a bowl is because the error function is in terms of the input squared, making it a three dimensional parabola in this case. The goal is to minimize this function, to get C as close to zero as possible. If we were to start on some random point on that three dimensional curve, the way to get to the bottom would be to go down the slope of the curve until the only way we could go was up. This is what the backpropagation algorithm is doing; given a starting position, it is finding the partial derivative of the error with respect to each of the weights, or the slope of the curve at that point, and then is taking a step down the curve by subtracting the partial derivative of each weight from that same weight. By iteratively going through the process of calculating the derivative and taking a small step down the slope in the direction of the derivative, we will soon arrive at the bottom of the curve (Nielsen, 2015).

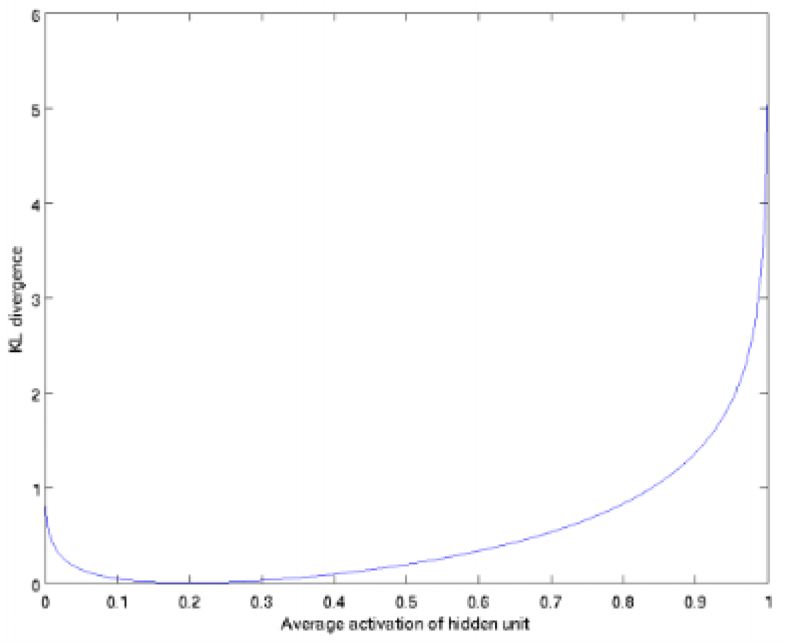
Sparse Autoencoder

An autoencoder is an ANN that learns features from unlabeled data. For images, an autoencoder tries to find a sparse representation of an image by trying to find an approximation to the identity function $f(x) \approx x$. It does this by setting the input of the ANN and the output to the same image and then limiting the number of hidden neurons. By limiting the number of hidden neurons, the network is forced to find structure in the data. The weights that go into the hidden layer are meant to find a compressed version of the input data and the weights that exit the hidden layer are meant to find a way to reconstruct that compressed data into the original data. For the majority of algorithms, the compression aspect of the autoencoder is the primary focus (Ng, 2013).



A simple autoencoder (Ng, 2013)

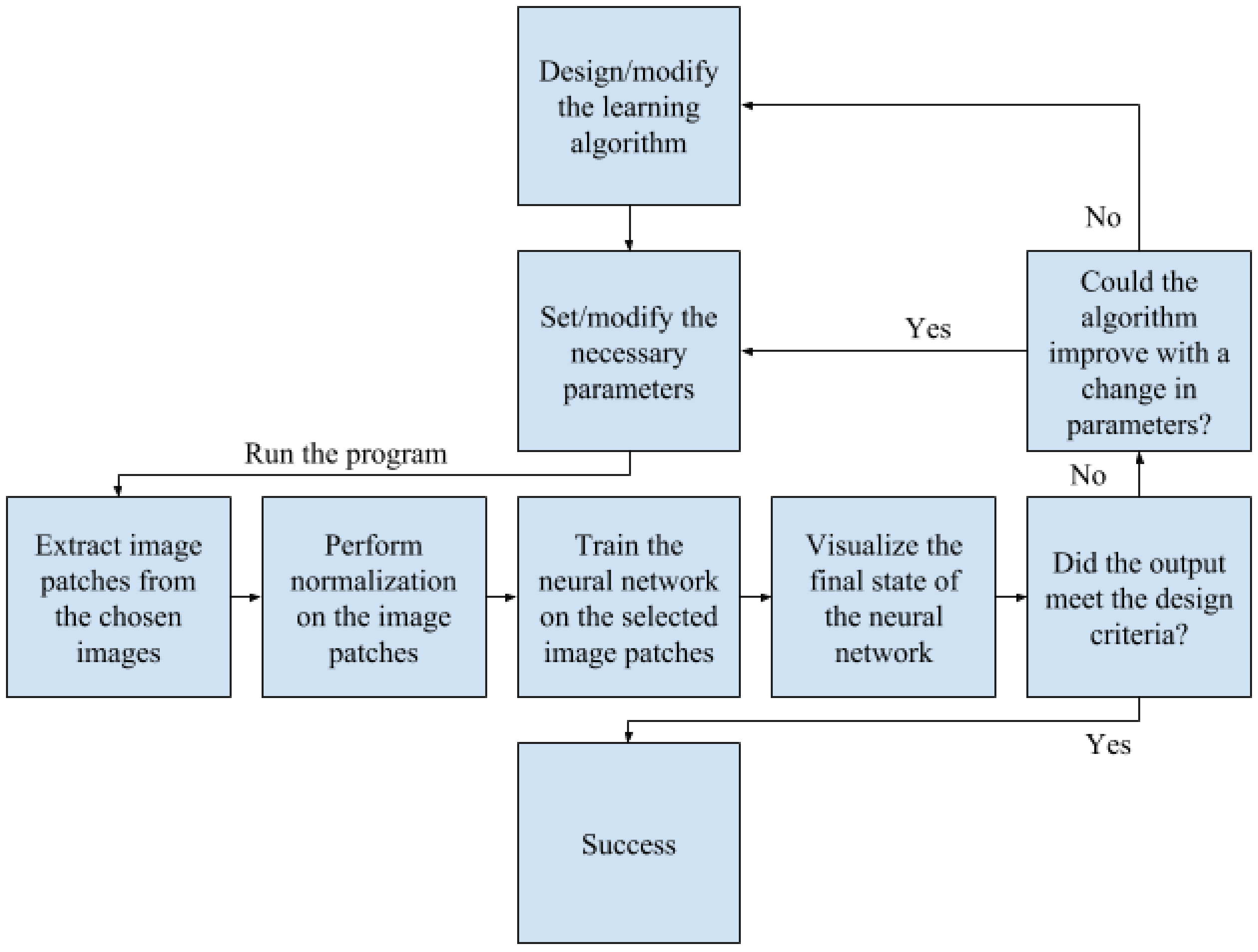
The basic autoencoder can be extended to a sparse autoencoder by adding an additional term to the error function called the sparsity term. The sparsity term is introduced so that even when the size of the hidden layer is larger than the input layer. the autoencoder can still find some structure in the data. If the hidden layer is larger than the input layer, one would expect that the autoencoder would simply learn the identity function, $f(x) = x$, by setting a few weights equal to one and all the rest equal to zero, creating a perfect reconstruction of the data (error=0). This problem can be solved with sparsity. Sparsity means that we want the average value that a neuron outputs to be close to zero. This ensures that the neuron will have an output of one very rarely, meaning that it has to find a more compressed form of the data than the one represented by the identity function. The way to calculate this for an individual neuron is $\frac{1}{m} * \sum_{i=1}^{m} a(x_i)$ where m is the number of training examples and $a(x_i)$ represents the output of the neuron when given the i-th training example. The sparsity term that gets added to the error function is $\sum_{j=1}^{h} p * log(\frac{p}{s_j}) + (1-p) * log(\frac{1-p}{1-s_j})$ where p is the sparsity parameter, or the desired average output of the neuron, h is the number of hidden neurons, and s_j is the average output of the j-th neuron. This function was chosen because it has a similar shape to a quadratic function and because it produces a nice derivative of $\frac{-p}{s_j} + \frac{1-p}{1-s_j}$ for the weights going into the jth neuron (Ng, 2013).



The sparsity function when p=0.2 (Ng, 2013)

Design Plan

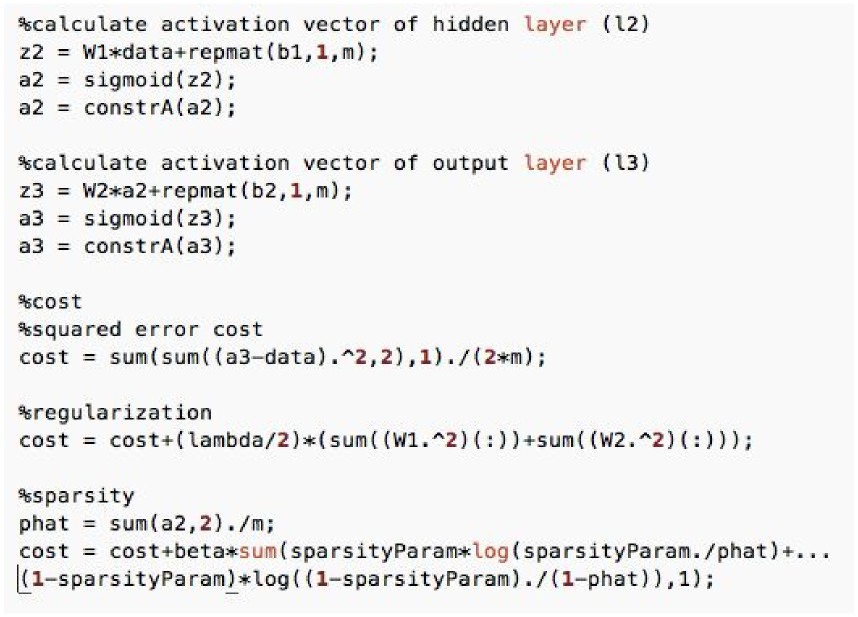
For this project, I used the Octave programming language, a free, open source, high level programming language with built in libraries for fast linear algebra computations. This programming language was also chosen for its readability and its ability to enable rapid prototyping of complex algorithms requiring many matrix calculations. All of the tests were performed on Macbook Pro late 2011 model with a 2.2 GHz Intel Core i7 processor and 4GB of 1333MHz DDR3 ram.


Design/modify the learning algorithm
Set/modify the necessary parameters
Run the program
Extract image patches from the chosen images
Perform normalization on the image patches
Train the neural network on the selected image patches
Visualize the final state of the neural network
Did the output meet the design criteria?
Yes
Success
No
Could the algorithm improve with a change in parameters?
Yes
No

Flow Chart

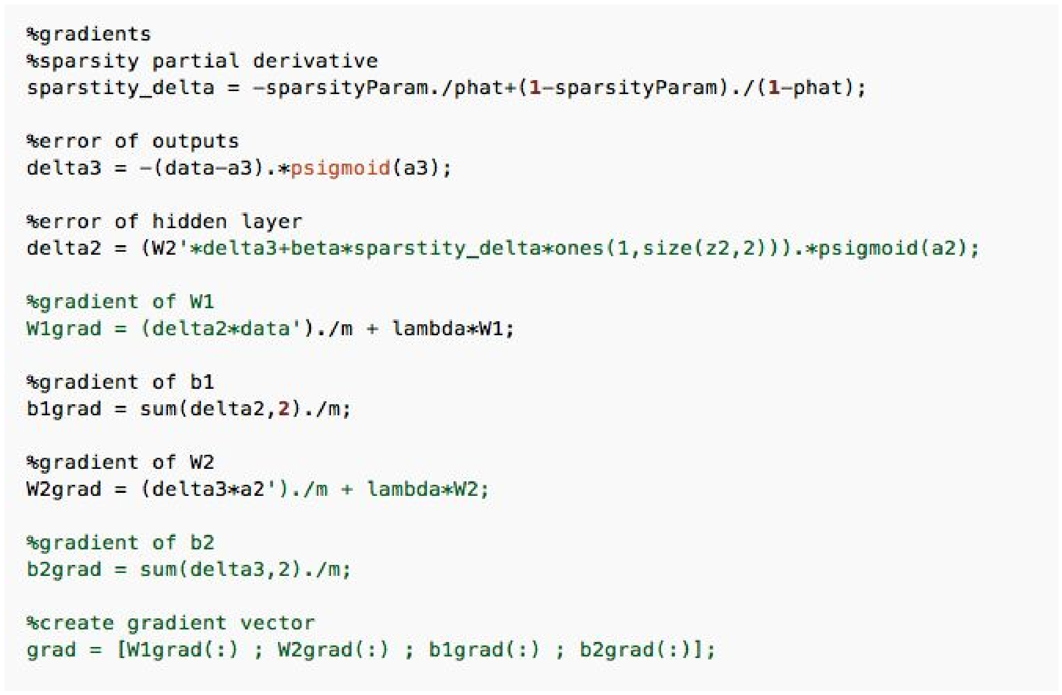
The first step of creating the algorithm was implementing a sparse autoencoder. A sparse autoencoder was chosen because it has been successfully used many times to extract visual features of varying abstraction from large sets of natural images. An example of a successful sparse autoencoder is Google's deep learning algorithm, which learned to recognize images of cats and people by observing unlabeled images sampled from 10 million Youtube videos. The algorithm used in that project was a modified sparse autoencoder that was trained over several days on very fast computers (Le et al., 2012).

The first step in implementing the sparse autoencoder consisted of writing the code to create initial values for the all of the weights in the autoencoder. This was done by writing a function called `initializeParameters`, which initialized the weight matrices of each layer to very small random numbers. This guaranteed that neurons would initially try to learn different features from each other and that the neurons would not immediately be stuck on one of the two plateaus of sigmoid function, which would increase learning time. The second step was to write a function that would calculate a feedforward pass through the network for all of the images in a given dataset in order to calculate the error, or cost, of the network with the current weights. This would allow a periodic progress check to make sure that the algorithm was actually converging to a solution. This was implemented with the following code, where `W1,W2,b1,`and `b2` stand for the first and second weight matrices and the first and second bias vectors respectively, `lambda` is the regularization parameter, `m` is the number of images in the data set, `beta` is the weight of the sparsity parameter, and `sparsityParam` is the sparsity parameter:



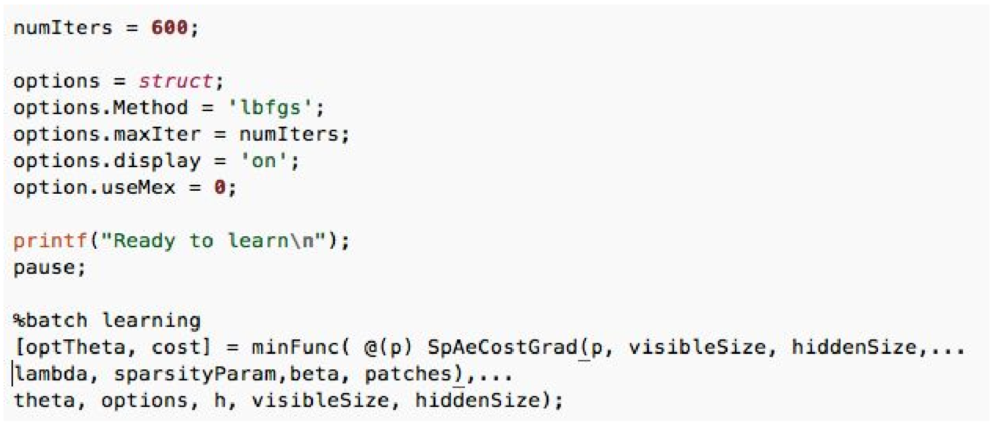
```
%calculate activation vector of hidden layer (l2)
z2 = W1*data+repmat(b1,1,m);
a2 = sigmoid(z2);
a2 = constrA(a2);

%calculate activation vector of output layer (l3)
z3 = W2*a2+repmat(b2,1,m);
a3 = sigmoid(z3);
a3 = constrA(a3);

%cost
%squared error cost
cost = sum(sum((a3-data).^2,2),1)./(2*m);

%regularization
cost = cost+(lambda/2)*(sum((W1.^2)(:))+sum((W2.^2)(:)));

%sparsity
phat = sum(a2,2)./m;
cost = cost+beta*sum(sparsityParam*log(sparsityParam./phat)+...
(1-sparsityParam)*log((1-sparsityParam)./(1-phat)),1);
```

In the code, the image dataset is represented by an `n by m` matrix where the i-th column contains the vector representing the i-th image in an unrolled format (think of it as taking an image and reordering all of the pixels in a long, one pixel wide line). The next step was to implement the backpropagation algorithm, which calculates the derivatives of the error with respect to each weight matrix. It then combines all of the derivative matrices into one long vector which it outputs to be used by the optimization function.



```
%gradients
%sparsity partial derivative
sparstity_delta = -sparsityParam./phat+(1-sparsityParam)./(1-phat);

%error of outputs
delta3 = -(data-a3).*psigmoid(a3);

%error of hidden layer
delta2 = (W2'*delta3+beta*sparstity_delta*ones(1,size(z2,2))).*psigmoid(a2);

%gradient of W1
W1grad = (delta2*data')./m + lambda*W1;

%gradient of b1
b1grad = sum(delta2,2)./m;

%gradient of W2
W2grad = (delta3*a2')./m + lambda*W2;

%gradient of b2
b2grad = sum(delta3,2)./m;

%create gradient vector
grad = [W1grad(:) ; W2grad(:) ; b1grad(:) ; b2grad(:)];
```

The next step involved writing the code to test the sparse autoencoder. This was done by using a library containing very fast and efficient optimization algorithms called minFunc (Schmidt, 2013). In the library, the optimization algorithm chosen to test the autoencoder was the Limited-memory Broyden–Fletcher–Goldfarb–Shanno (LBFGS) algorithm. What this function does is it accepts another function that outputs the current error of the problem as well as the partial derivatives of all of the variables in the function and uses those to approximate the second order partial derivatives of the variables. It does this in order to more quickly arrive at a solution than a simple gradient descent algorithm, which would take constant steps of the same size and converge very slowly. This section of code was used to get a baseline for the learned visual features by using a static dataset of thousands of images that it trained on all at once. This is the traditional way that the sparse autoencoder is used and the results from this algorithm will be compared to the results of the dynamically learning algorithm in order to determine the success of the dynamic learning algorithm. The code used to call the LBFGS function is:



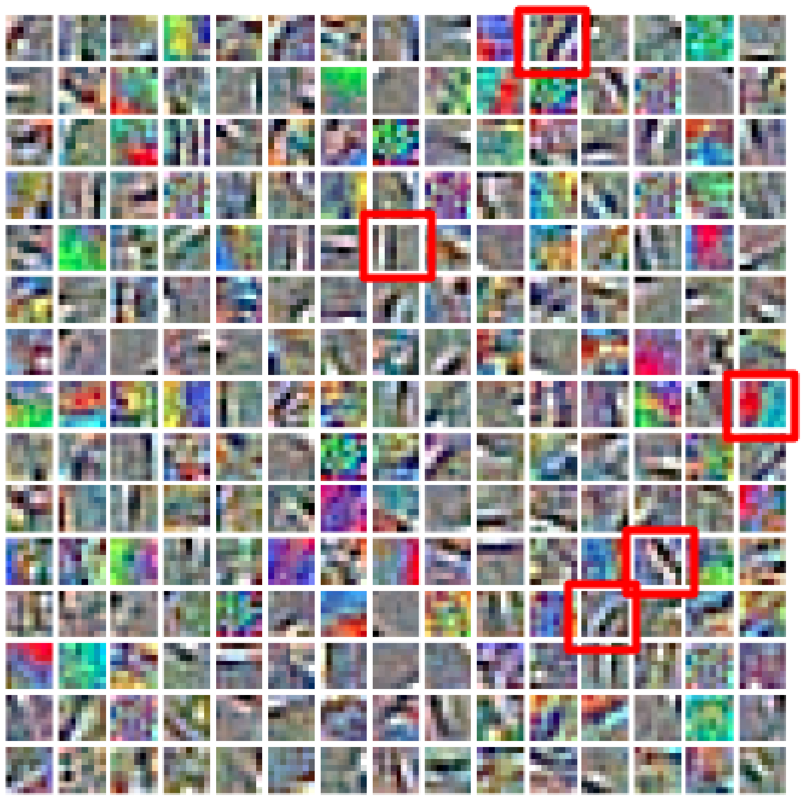
```
numIters = 600;

options = struct;
options.Method = 'lbfgs';
options.maxIter = numIters;
options.display = 'on';
option.useMex = 0;

printf("Ready to learn\n");
pause;

%batch learning
[optTheta, cost] = minFunc( @(p) SpAeCostGrad(p, visibleSize, hiddenSize,...
lambda, sparsityParam,beta, patches),...
theta, options, h, visibleSize, hiddenSize);
```

Then code was written to visualize the learned features of each neuron and code was written to normalize the image data of the pixel intensities from the traditional 0-255 range to 0.1-0.9. Finally, code was written to train the sparse autoencoder on a sequence of images, meaning that the sparse autoencoder would only get to learn on a few images at a time instead of thousands at once.

Results and Discussion

The initial design of the algorithm used colored image patches from the CIFAR-10 image dataset. The CIFAR-10 dataset consists of 60,000 32 by 32 pixel natural images that all have labels attached to them that place them into one of ten categories. The labels were not used during the course of this experiment as the goal was to learn on unlabeled data. To keep the memory size manageable, only the first 10,000 images in the CIFAR-10 dataset were ever used throughout the course of this experiment. The program first goes through and extracts 8 by 8 image patches from random positions of random images in the dataset, making sure that each image is used at most once. It then unrolls the 8 by 8 RGB image patch into a 192 (8*8*3) dimensional vector and appends it to an image patch matrix where the i-th column represented the unrolled i-th image patch. Then, it runs an algorithm called Zero Component Analysis (ZCA) whitening on the image matrix to normalize the pixel data to a gaussian distribution and give the data a mean of zero. Then, it initializes the weight matrices and starts learning using the specified algorithm.

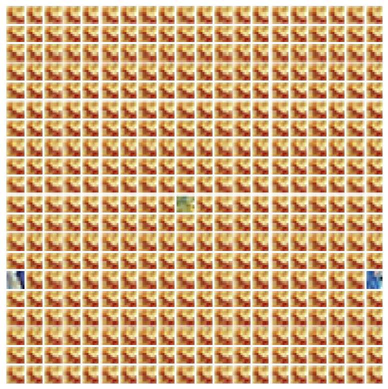
There were several parameters that could be modified throughout the experiment. This included the total number of images sampled from, the size of the square patches in pixels, the number of hidden neurons, the sparsity parameter, the weight of the sparsity parameter (beta), the regularization parameter (lambda), and the number of iterations that an algorithm should perform to optimize the autoencoder. In Andrew Ng's lecture notes on the sparse autoencoder used on RGB images, he set the sparsity parameter to 0.035, lambda to 0.003, and beta to 5, so those were the values chosen for those parameters in this experiment (Ng, 2013). The initial run of this algorithm trained the autoencoder on the full set of images at once using the LBFGS algorithm with the number of images set to 3000, the image patch size set to 8 pixels, the number of hidden units set to 225, and the number of iterations set to 600. These values are all very similar to the values used by Andrew Ng in his lecture notes (Ng, 2013). This first run was to get a baseline for the kind of features that would classify a learning algorithm as meeting the design criteria for features learned. This was used to check the performance of later algorithms that learned on the data in batches or in an online fashion. The feature that each of the autoencoder's neurons has learned to detect after the first run is shown below.

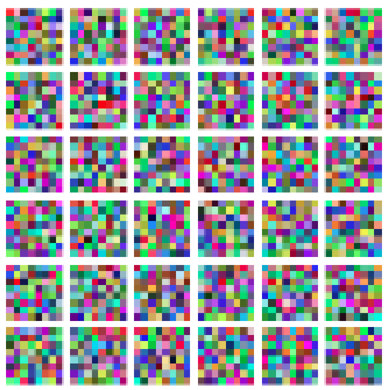


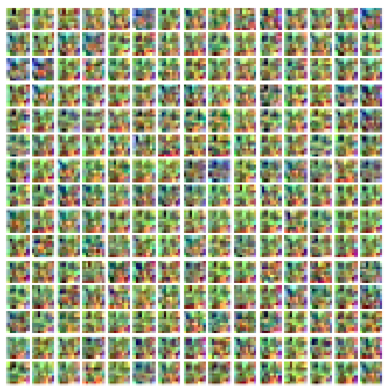
The neurons highlighted in red boxes show some neurons that have learned valuable edge detectors. Edge detectors can be seen whenever there is a sharp shift between two colors or intensities in the learned feature of a neuron. It is also apparent that this autoencoder has learned many edge detectors at different angles and different shapes, indicating that it learned a good representation of all of the images that were fed into it. These features are also similar to the feature detectors found in a cat’s primary visual cortex. The criteria for determining the success of later algorithms was that if the algorithm produced many different feature detectors that were similar to the edge detectors learned by the baseline, then it would be considered a successful algorithm.

Now that a baseline had been set, an algorithm was created to train the autoencoder on only a few images at a time, known as a batch of images, for a small number of iterations of

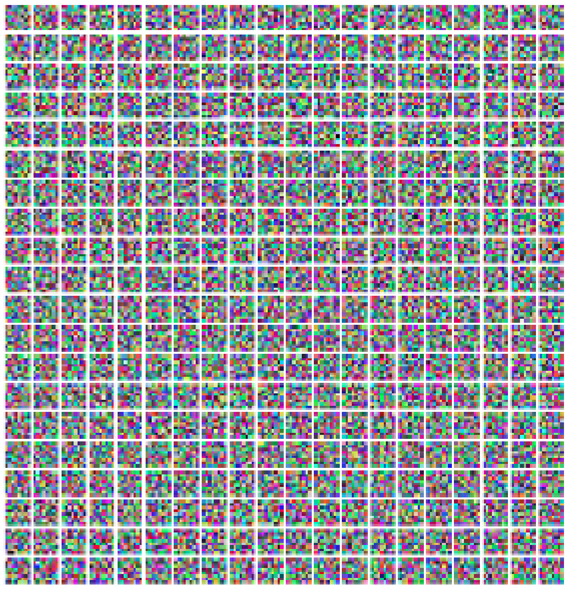
LBFGS and then repeating this process for the next batch of images. This also introduced a new variable parameter, the batch size, which represented the number of image patches trained on in each batch. Unfortunately, it turned out that LBFGS was simply too aggressive and only optimized the autoencoder on the features represented in the first or second batch. When the number of hidden units or the number of iterations were decreased to try and force it to learn a more general representation of the data, the algorithm still optimized itself too quickly. When the batch size was decreased, the algorithm didn't learn anything useful at all. Below are some visualizations of the final autoencoder for three of the trials during this phase of the experiment. Despite several variations to the batch size and the number of iterations, the autoencoders clearly didn't learn any useful features. In the case of trial 7, it did learn two or three features, but it didn't learn a generalization for all of the data; it only generalized to the first few patches it was trained on.

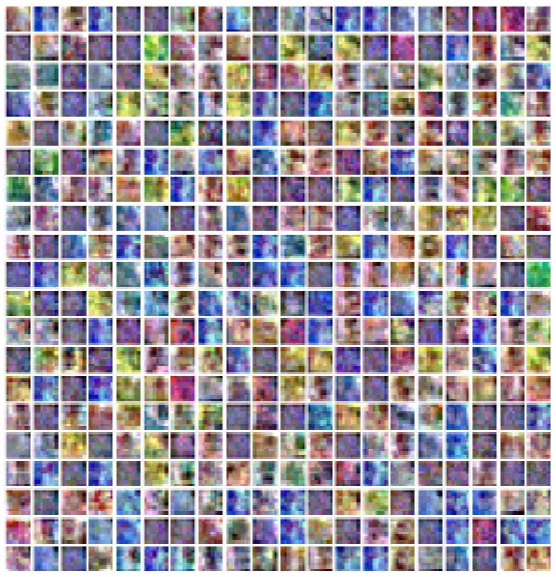

Trial 7
Hidden units: 400
Iterations: 10
Batch size: 10


Trail 5
Hidden units: 36
Iterations: 10
Batch size: 1

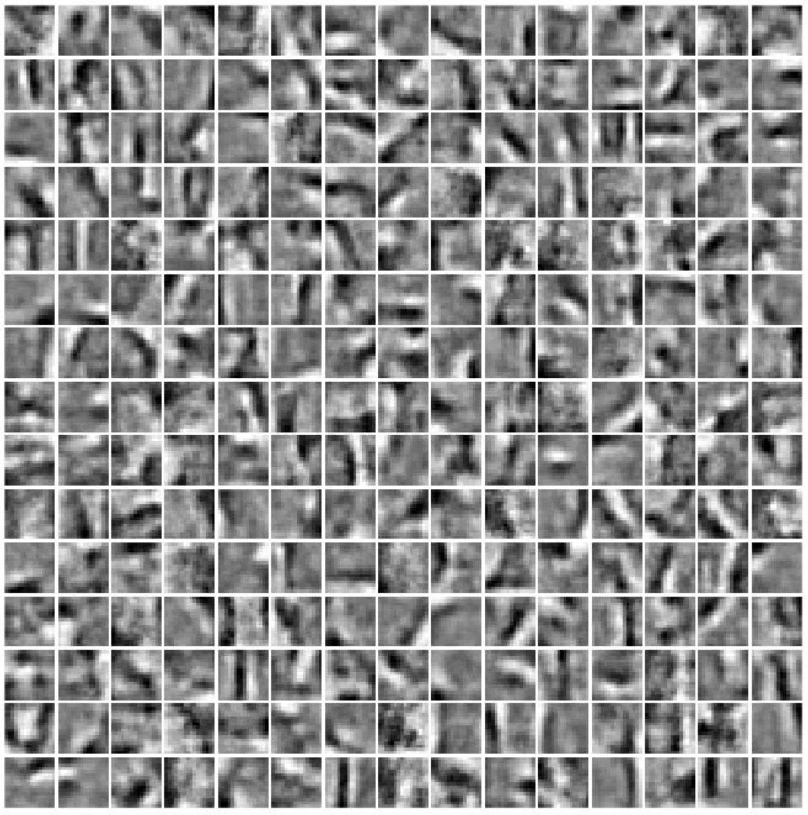

Trial 9
Hidden units: 225
Iterations: 6
Batch size: 20

Due to the inability to control the LBFGS algorithm, a gradient descent algorithm was programmed, which is a basic optimization algorithm that calculates the derivatives of the weights and updates the weights by subtracting the partial derivative of each weight multiplied by a learning rate, called alpha. Alpha was initially set to 0.01, a small enough value to make sure that the algorithm didn't oscillate around the global minimum. However, it turned out to be too small and the algorithm was not able to learn any useful features by the time it was done learning on all batches. Thus, alpha was increased to 0.1. Although this did diversify the learned features, the learned features did not represent the edge detectors seen in the baseline. Some of the trial visualizations for the gradient descent algorithm are shown below.

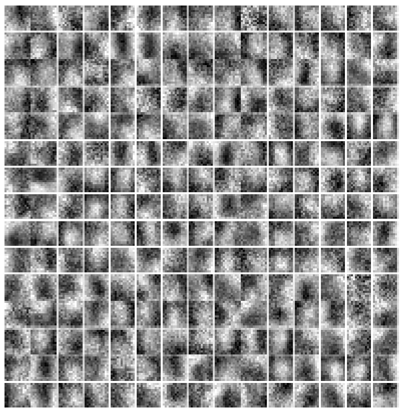

Trial 12
Hidden units: 400
Alpha: 0.01
Iterations: 20
Batch size: 30

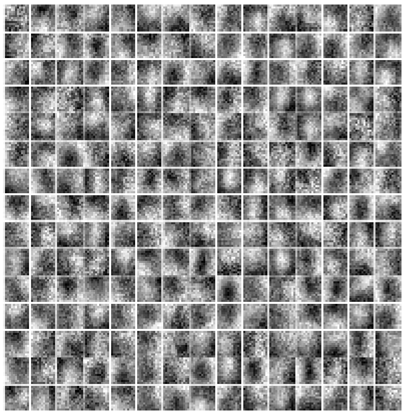

Trial 15
Hidden units: 400
Alpha: 0.1
Iterations: 20
Batch size: 20

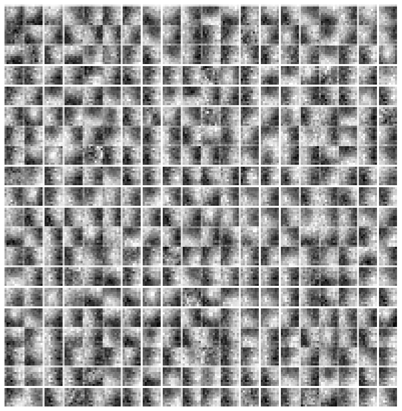
In an effort to simplify the process and decrease the runtime needed to optimize the autoencoder, the color patches were turned into grayscale patches, meaning that the image patch vectors would only be 64 (8*8) dimensional vectors instead of 192 (8*8*3). Before testing any batch algorithms, a baseline test was run again on the grayscale image patches. The algorithm used was LBFGS and all of the patches were trained on at the same time. For this trial, the number of images was 6000, the patch size was increased to 16 to make the visualizations clearer, the number of hidden units was set to 225, and the number of iterations was set to 800. The sparsity parameter, lambda, and beta, were set to 0.01, 0.0001, and 3 respectively, which are the values that Andrew Ng used for training autoencoders on grayscale images in his lecture notes (Ng, 2013). Additionally, the ZCA whitening step was removed in favor of simply normalizing the pixel values to zero mean and then mapping each pixel value to a value between 0.1 and 0.9. The resulting visualization is shown below. The types of edges that have been learned are also much clearer in this visualization, due to the increase in the patch size.



After the baseline was set, a modification was made to the gradient descent algorithm based on current results. It was noted that when using a small alpha while training on the color patches, the autoencoder was learning too slowly and didn't learn any useful features. However, when alpha was set too high, it generalized too quickly to the first few image patches it was trained on. The new gradient descent algorithm worked in three steps. It started out with a small base alpha and a small base number of iterations. It would run gradient descent for the number of base iterations at the base learning rate. This would get the autoencoder in a position to generalize. Then, it would increase the number of iterations and the alpha by a factor of 10 and run gradient descent again but it would retain the state of the autoencoder from the previous run and build off of that. This was so that it could learn at a faster rate as it approached the solution, as learning typically slows down with a constant learning rate as the autoencoder is optimized. It would multiply the learning rate and alpha by 10 once more, run gradient descent, and then be done. The results were better than before. The algorithm starts to learn different types of edges but there is still a lot of noise and the edges aren't that clear. Additionally, the autoencoder doesn't seem to generalize well and only picks up on a few different types of angles.

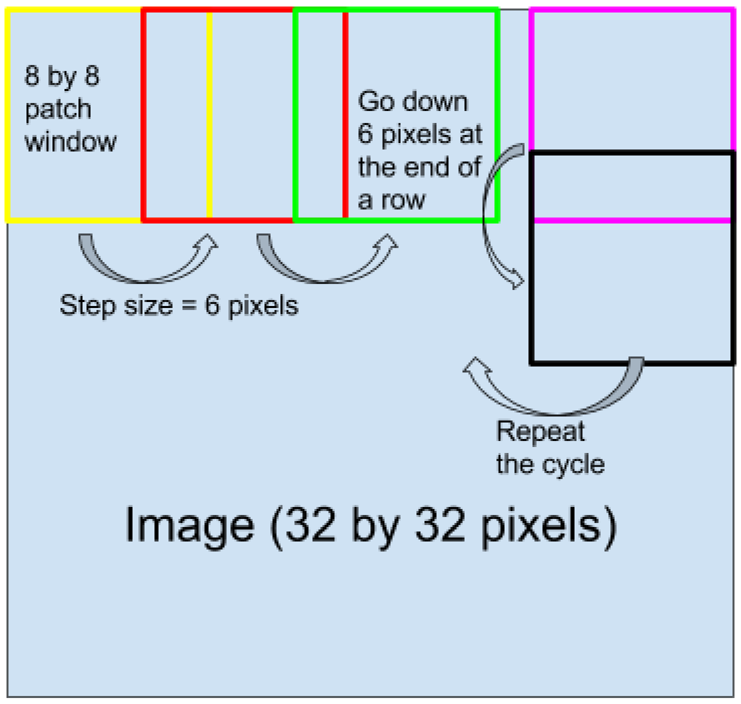

Trail 29
Hidden Units: 225
Base Alpha: 0.01
Base Iterations: 40
Batch Size: 100


Trail 30
Hidden Units: 225
Base Alpha: 0.01
Base Iterations: 40
Batch Size: 50


Trail 39
Hidden Units: 400
Base Alpha: 0.005
Base Iterations: 5000
Batch Size: 2

It appeared that this algorithm needed to be more aggressive to get rid of the noise and it needed to find some way of generalizing more. The answer to the first question was to use separate learning rates for each weight and then updating the value of a learning rate for a particular weight based on its performance. For example, if at iteration 10, a certain weight has a positive derivative, and then at iteration 11, the weight has a negative duration, then it means that the weight overshot the minimum and its learning rate needs to be turned down very quickly. However, if at iteration 11 the derivative is still positive, then that means we are still making our towards the minimum and we can increase the learning rate by a little bit to speed it up. This is sometimes called an Automated Learning Rate (ALR). The second problem about generalization was solved with a realization. For the majority of the trials the batch size had been set to really low numbers. This had been done because it had been reasoned that the brain can only store a few frames worth of visual data in the primary visual cortex before the data moves on to get processed by other parts of the brain. This would mean that the primary visual cortex learns feature detectors by only analyzing a few frames of data at a time. The goal for some time had been to get the batch size to under 5 and still learn many different features.

It was realized that each image in the batch only represented a patch from a single image and that a single 32 by 32 pixel image would contain 16 completely different 8 by 8 patches. However, 16 patches would still not be enough, so a sliding window was used to slide the 8 by 8 patch window across the 32 by 32 image by a constant step size less than 8 and then moving down the image by the step size at the end of a row. If your step size is s, then for a 32 by 32 picture with an 8 by 8 patch window, you would get $((32-8)/s+1)*((32-8)/s+1)$ patches. When $s=4$, you get 49 patches, which is a large increase from 16.

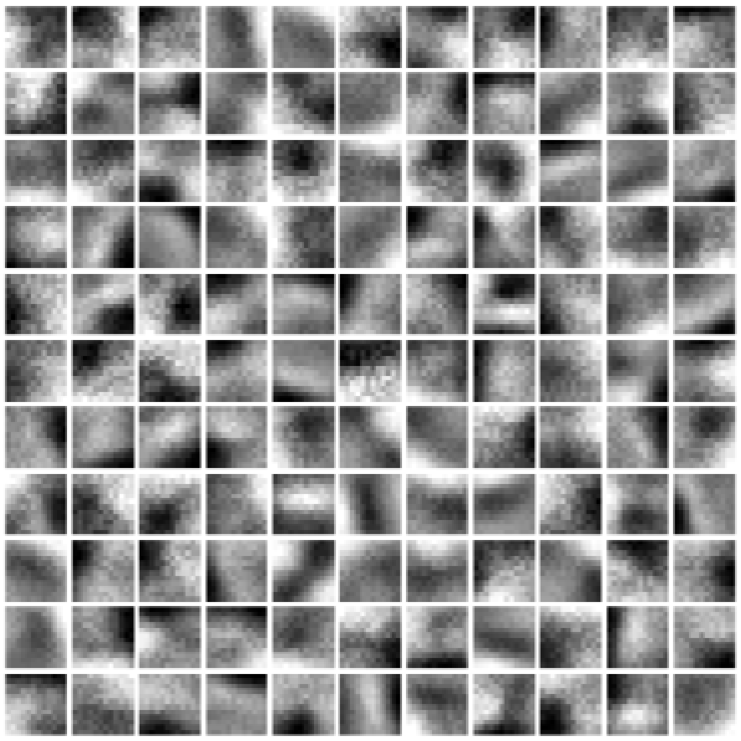


Sliding window approach

Additionally, if the current image contains many different angles and objects, this method will be able to capture all of those features and be able to build a more complete generalization for the entire dataset. By combining the sliding window and the separate ALRs, the algorithm was substantially improved and was able to learn important general features from the dataset. The algorithm also introduced a few new parameters into the algorithm. These included a slide step parameter which represented the step size of the sliding window, the initial alpha and the convergence alpha for the separate learning rates, the number of sequential frames of image data to train on at once (the image set size), and the number of iterations to perform gradient descent for the current batch.

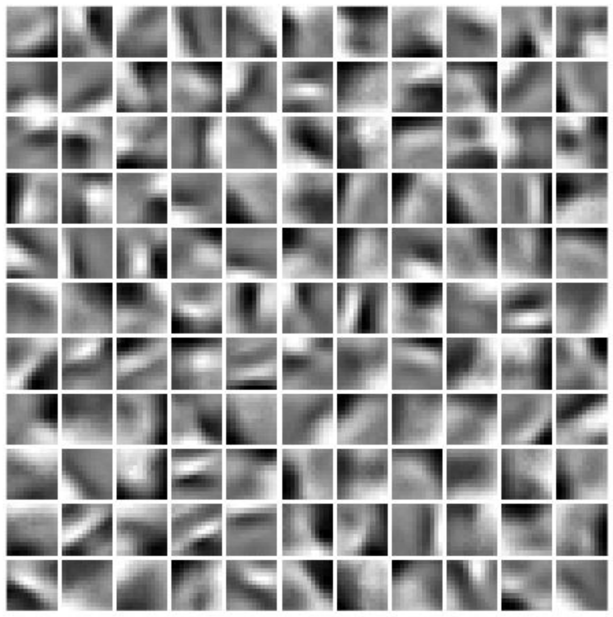
Once the success of the algorithm was determined on the CIFAR-10 dataset, a new dataset was introduced. Due to the fact that the primary visual cortex learns from a continuous stream of input with some correlation between images and not random images like in the

CIFAR-10 dataset, it was decided to capture 1 to 2 minute videos of certain environments and extract the image frames from each video. The first video was of a home interior and was 161 seconds long and recorded at 30 frames per second. This yielded about 4830 (161*30) images, and each image was scaled down to 100 by 56 pixels to make the image sizes manageable and learning relatively fast. This sequence of images was trained in sequential order (as opposed to the random order of previous algorithms) with an image patch size of 12. The state of the autoencoder after learning on the last set of frames is shown below.

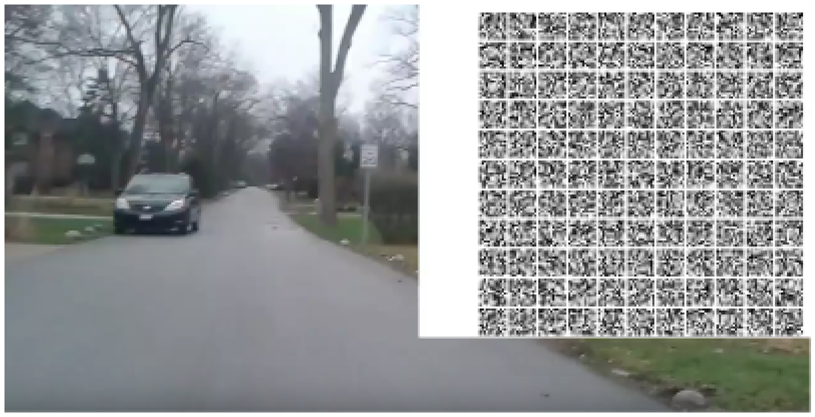


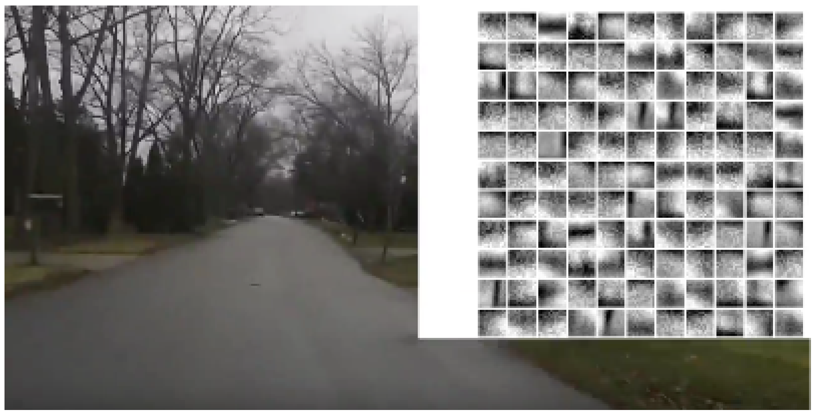
Trial 41
Hidden units: 121
Image set size: 3
Iterations: 5
Slide step: 8
Initial Alpha: 0.001
Converge Alpha: 1

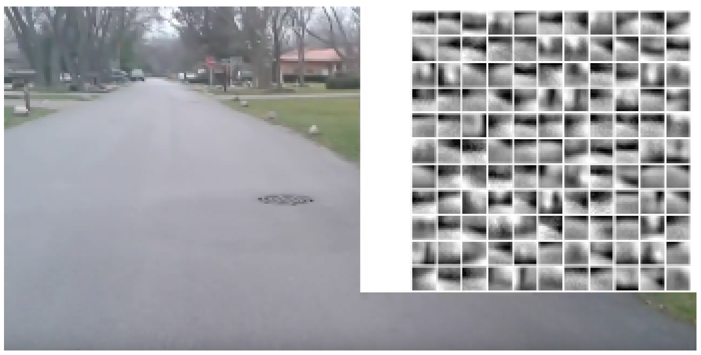
This algorithm has learned many more general features than the previous algorithms and has learned more defined edges. However, there was still a bit of noise and the features were still not as defined as desired. So the number of iterations was increased to decrease noise and the slide step was decrease to 4 to increase the total number of patches per image and generate more general results. The results of these modifications are shown below.



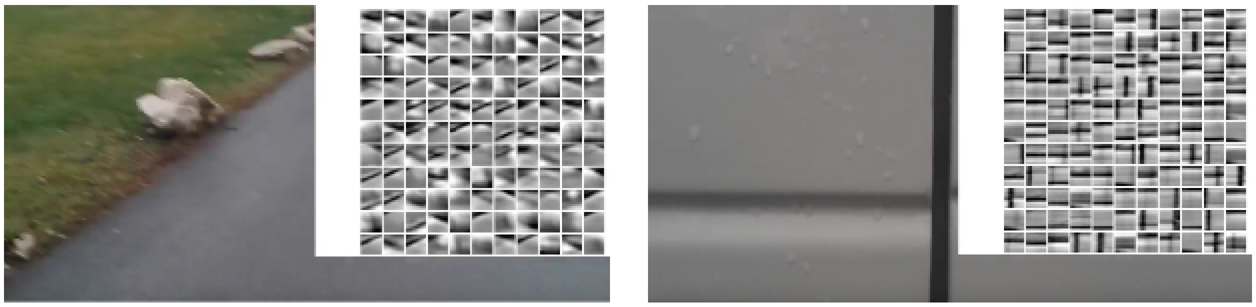
Finally, another video was captured outdoors by walking down a suburban lane for 270 seconds at 30 frames per second to produce approximately 8100 images (these images were also scaled down to 100 by 56 pixels). Additionally, while training the algorithm, the state of the network after training on every image set was recorded to allow viewing of intermediate results. Below are some of the intermediate results.


The initial state of the autoencoder


About 20 seconds later, it has already learned some definitive veritcal lines, likely representing the trees



Now it has learned a few more features, such as the horizontal driveways and the angle of the road



However, when only exposed to a specific feature for about ten or twenty seconds, it starts to forget what it previously learned and starts learning more about the new dominant feature of the environment

Conclusion

After testing and modifying the learning algorithm for the sparse autoencoder several times, the algorithm that was most successful at fulfilling the purpose of the investigation was a gradient descent algorithm with adaptive learning rates for each individual weight. Additionally, a sliding window was used to extract sufficient image patches from images in order for the algorithm to have access to enough data to learn important features. This algorithm was considered the most successful because it produced the most generalized feature detectors similar to those seen in the baseline algorithm. Additionally, neurons with these kinds of feature detectors have been found in the primary visual cortex of the mammalian brain. Finally, this algorithm was able to learn these feature detectors by only looking at three frames of video at any given time, meaning that its method of learning is much closer to the way the brain actually learns in comparison to traditional autoencoders.

While the preliminary algorithm was unsuccessful and did not meet any aspect of the design criteria, it and the other unsuccessful algorithms did provide valuable insight into inner workings of the algorithm as it was learning. This can not be overlooked as these insights were what ultimately led to the inspiration for the final design. Nevertheless, the final algorithm was able to outperform the previous ones because it was allowed to look at more sections of an image at once and because it had learning rates that were able to vary their aggressiveness. The absence of the latter was a key downfall of some of the previous algorithms, as they were either pre built with too much or too little aggressiveness in mind, forcing them to converge to false optimizations too quickly or not being able to converge at all within a reasonable time span. The algorithm was also able to learn from video sequences, where consecutive images are typically very similar and often don't contain the same features as previous seconds. The results from using this algorithm on the video of a home interior prove that it has some ability to remember visual features from several seconds in the past. However, the intermediate results from the video of a suburban lane prove that it can very quickly overwrite dominant features from previous times with the dominant features of the present. While this is one of several aspects of this algorithm that need to be improved to consider it on par with the learning in the mammalian brain, it is definitely a large step in the right direction.

References

Burger, J. T. (2004, June 14). A basic introduction to neural networks. Retrieved from http://pages.cs.wisc.edu/~bolo/shipyard/neural/local.html

Communication. (2011, September 27). Retrieved from http://www.bris.ac.uk/synaptic/basics/basics-3.html

Le, Q. V., M. Ranzato, R. Monga, M. Devin, K. Chen, G. S. Corrado, J. Dean, and A. Y. Ng. (2012). Building high-level features using large scale unsupervised learning [PDF document]. *The International Conference on Machine Learning*. Edinburgh, Scotland. Retrieved from http://static.googleusercontent.com/media/research.google.com/en//archive/unsupervised_icml2012.pdf

Mitchell, T. M. (2006, July). *The discipline of machine learning* [PDF document]. Retrieved from http://www.cs.cmu.edu/~tom/pubs/MachineLearning.pdf

Ng, A., Ngiam, J., Foo, C. Y., Mai, Y., & Suen, C. (2013, April 7). UFLDL tutorial. Retrieved from http://deeplearning.stanford.edu/wiki/index.php/UFLDL_Tutorial

Nielsen, M. A. (2015). *Neural networks and deep learning*. Retrieved from http://neuralnetworksanddeeplearning.com/

Schapire, R. (2008, February 4). *Theoretical machine learning*. Lecture presented at Computer Science 511 in Princeton, Princeton. Retrieved from http://www.cs.princeton.edu/courses/archive/spr08/cos511/scribe_notes/0204.pdf

Schmidt, M. (2013). minFunc [Computer software]. Retrieved from https://www.cs.ubc.ca/~schmidtm/Software/minFunc.html

Segev, I. (2015, October 14). *Neurons as plastic/dynamic devices*. Lecture presented at Synapses, Neurons and Brains in Hebrew University of Jerusalem, Jerusalem. Retrieved from https://www.coursera.org/learn/synapses

Stergiou, C., & Siganos, D. (1997, May 12). Neural networks. Retrieved from https://www.doc.ic.ac.uk/~nd/surprise_96/journal/vol4/cs11/report.html