PSYC 190

Artificial Neural Networks: The Breakdown

The Analyses of its Components

Introduction

Neural networks are a set of algorithms that were set up to imitate the human brain and designed to recognize patterns. They interpret sensory data through machine perception from raw input and transforms it into numerical data that we can use for analysis to lessen human error and labor. The neural networks group unlabeled data according to similarities among the example inputs and classify data when they have a labeled dataset to train on.

Networks are composed of several layers and hyper-parameters: the input layer, the hidden layer(s), and the output layer. Each layer consists of neurons that represent the ones in the human brain. This is where all the computation happens -- each node holds a number that is between 0 and 1.

For pattern and image recognition of handwritten digits, it starts with a bunch of neurons that correspond to the 28 x 28 pixels of the input image, which results in a total of 784 pixels. Each node holds a number that is the grayscale value of the corresponding pixel ranging from 0 for a black pixel and 1 for a white pixel. The number inside the node is called the *activation.* The 784 pixels of the input image make up the first layer of the network model and the output layer is made up of the 10 digits that it can choose from at the end of the process. Each node in the output layer has a scale from 0 to 1 to represent how much the system thinks the given image corresponds to a given digit. The number of layers in between and the neurons in each hidden layer are based on the user's preference of how they want to model their neural network structure.

The activations computed in one layer determines the activations computed in the next layer. As it trickles down the neural network model, the activations will cause a node in the output layer to light up because of the weights it is "nudging" within the hidden layer(s) to

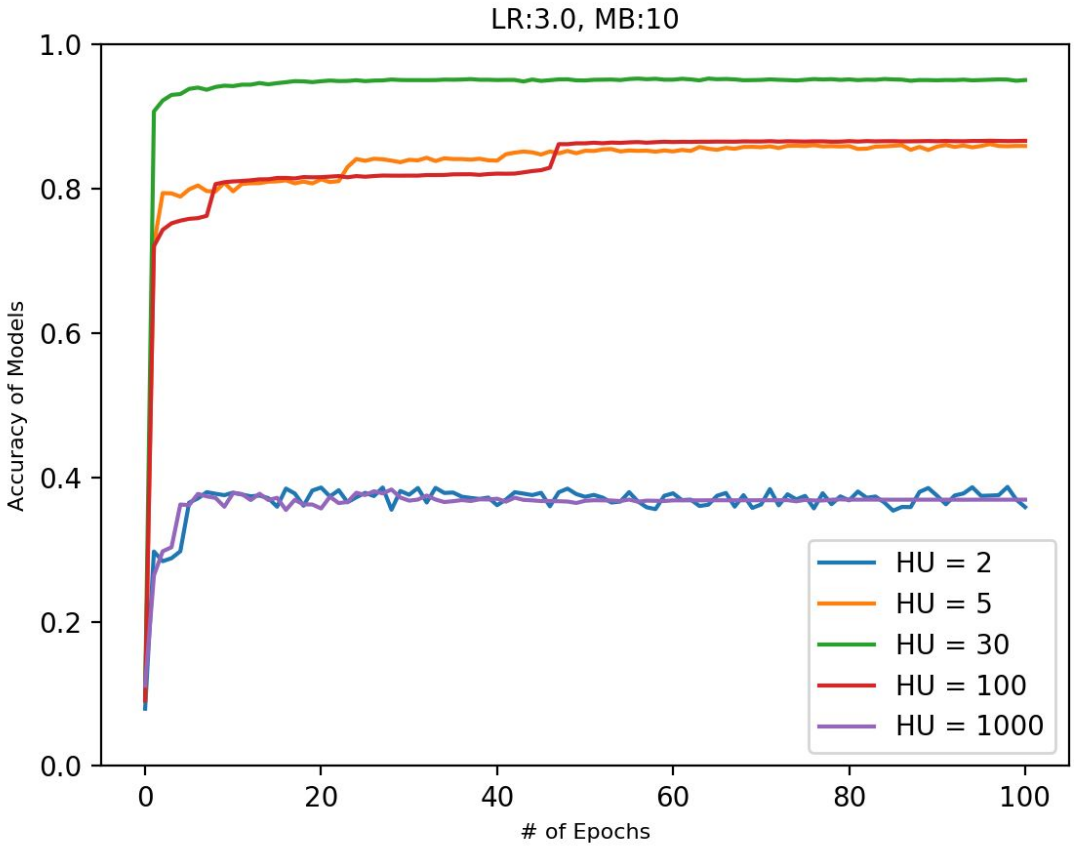
gradually reach its maximum accuracy in the output. Doing so tells us which digit the model thinks the image represents. By the time it reaches the end of the model, it will go through *backpropagation* to iterate through the layers again but backwards and changing the values within the nodes for every layer as it did for the *feedforward.* These steps are repeated until the model's output is as accurate as possible to the image input.

If a person is presented with handwritten digits, the human brain effortlessly recognizes the digit; however, recognizing handwritten digits is not easy for a computer and to be able to do that is extraordinary. Overall, the neural network can be thought of as a function that takes in the outputs of the previous layer and "spits" out a number from 0 to 1, starting from the input of 784 numbers and eventually "spitting" out one of the 10 numbers as an output.

Hidden Unit

The hidden units are the individual neurons within the hidden layer(s) where most of the computation happens. Each of the neurons within these layers have weights and "biases" attached to them to connect them with the next layer. Using more hidden neurons can lead to better results but this is not always the case. Increasing the number of hidden units will increase the accuracy of the models; however, if the number is too large, it becomes *overfitting* and ironically becomes inaccurate, due to the fact that the values within the neurons start to "cancel" each other out. Doing so will also slow down the execution. To speed up the process of running the program, the user can decrease the number of epochs, hidden units, or training data. An epoch is one single pass of the data through to the network. Its primary job is to transform the inputs into something that the output layer can use. In order for it to do this, the weights for each hidden neuron gets "nudged" to familiarize themselves with the raw input. The output layer transforms the hidden layer activations (the new weight values) into the closest representation of the input image. There are *bias* neurons that are somewhat in the input layer that is disconnected from the hidden layer. The purpose of these biases allow for the user to shift the activation function to the left or right, which may be critical for successful learning and recognition for the model. In other words, it allows for the layers to fit the prediction with the input data better. Using a bias effectively adds another dimension to the input space and allows for more opportunities for the neural network to learn and recognize the input patterns.

Effects on Performance



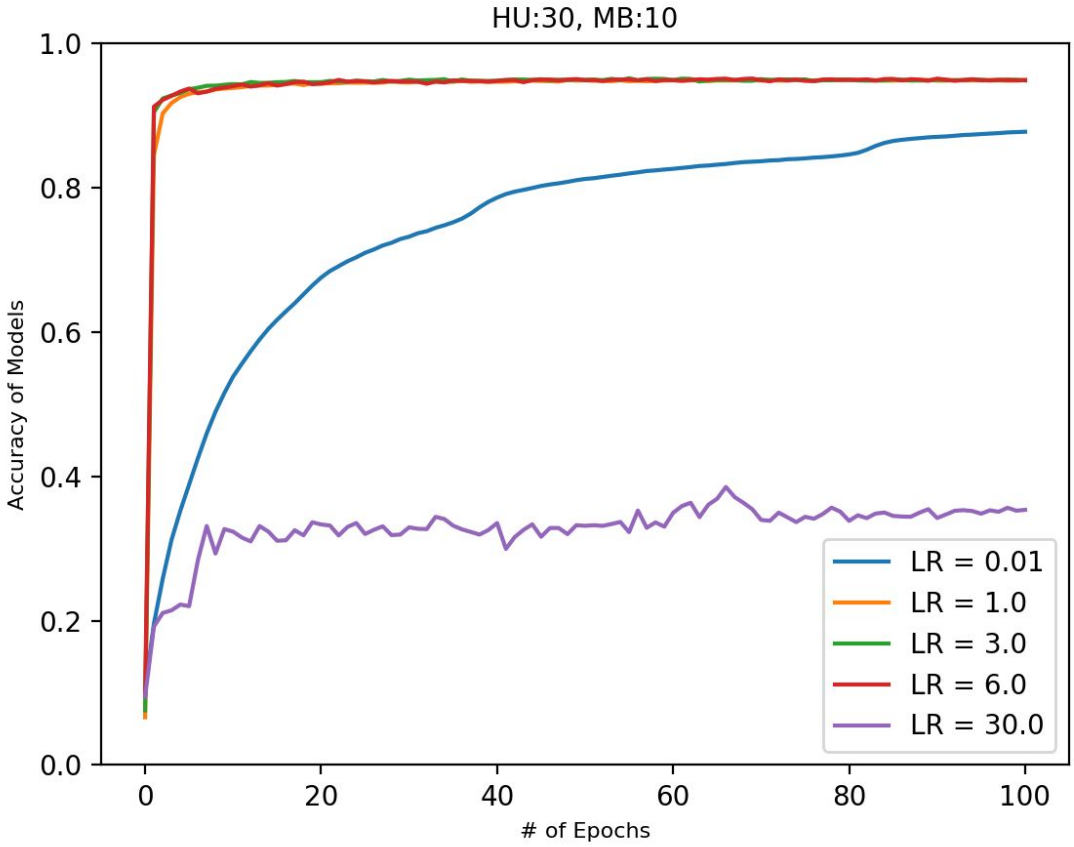
The optimal size of the hidden layer is usually between the sizes of the input and output layers. There is no defined way to figure out the optimal value to choose as a hidden unit size except for trial-and-error. The more hidden units there are, the more accurate the network becomes; however, making it too large can make the network inaccurate. In this plot, each line represents the average accuracy of 2 models over 100 epochs. The learning rate and the mini-batch size are kept constant to see the effects after only changing the number of hidden units. At 2 hidden units, the accuracy begins at 0.30 for the first epoch. It quickly grows to stay static from 0.36 to 0.38 once the 8th epoch begins to run until it reaches the last epoch. Now, if we were to choose a very high number of hidden units, like 1000, its effects are surprisingly similar to having only 2 hidden units because it becomes *overfitting*. It looks to be more consistent after the 25th epoch at around 0.37, making it only 37% accurate when attempting to recognize that specific handwritten digit. With this

data, we can see that choosing a higher number of hidden units does not necessarily improve the performance. Having a hidden number size of 30 produces the most ideal results since it has the highest accuracy percentage and success rate.

Learning Rate

The learning rate is a small number, usually ranging between 0.0001 and 0.01 but the actual value can vary. This hyper-parameter tells the optimizer how far to move the weights in the direction of the *gradient* for a mini-batch. We need to define a model and estimate its parameters based on a training dataset. A popular technique to calculate those parameters is to minimize the network's error with the *stochastic gradient descent*. The gradient descent estimates the weights of the model in many iterations by minimizing a cost function at every step. It is better to set the rate between 0.0001 and 0.01 because we do not want to overset the hyper-parameters. We essentially overwrite the previous weights that were set on each connection between the layers and update them with the new values we get from multiplying the learning rate and stochastic gradient descent. If we follow the typical guideline (0.0001 to 0.01), we can avoid taking a step that is too large in the direction of the gradient and "shoot" past the optimal value and miss it. To avoid this, we can set the learning rate to a number on the lower side of this range. With this option, since our steps will be extremely small, there will be less room for error even though it might take us longer to get to the optimal value that we want. If the learning rate is low, the training is more reliable, but optimization will take a lot of time. If the learning rate is high, then training may not converge or diverge, allowing the optimizer to overshoot the minimum of the gradient descent. In order for the gradient descent to work we must set the learning rate to an appropriate value. This parameter determines how fast or slow we will move towards the optimal weights. If the learning rate is very large, we will skip the optimal solution because the training may not converge or diverge. If it is too small, we will need too many iterations to converge to the best values. In this case, the training is more reliable, but optimization will take a lot of time. We need to tune the learning rate to find the most optimal value for our neural network because it is a crucial factor.

Effects on Performance

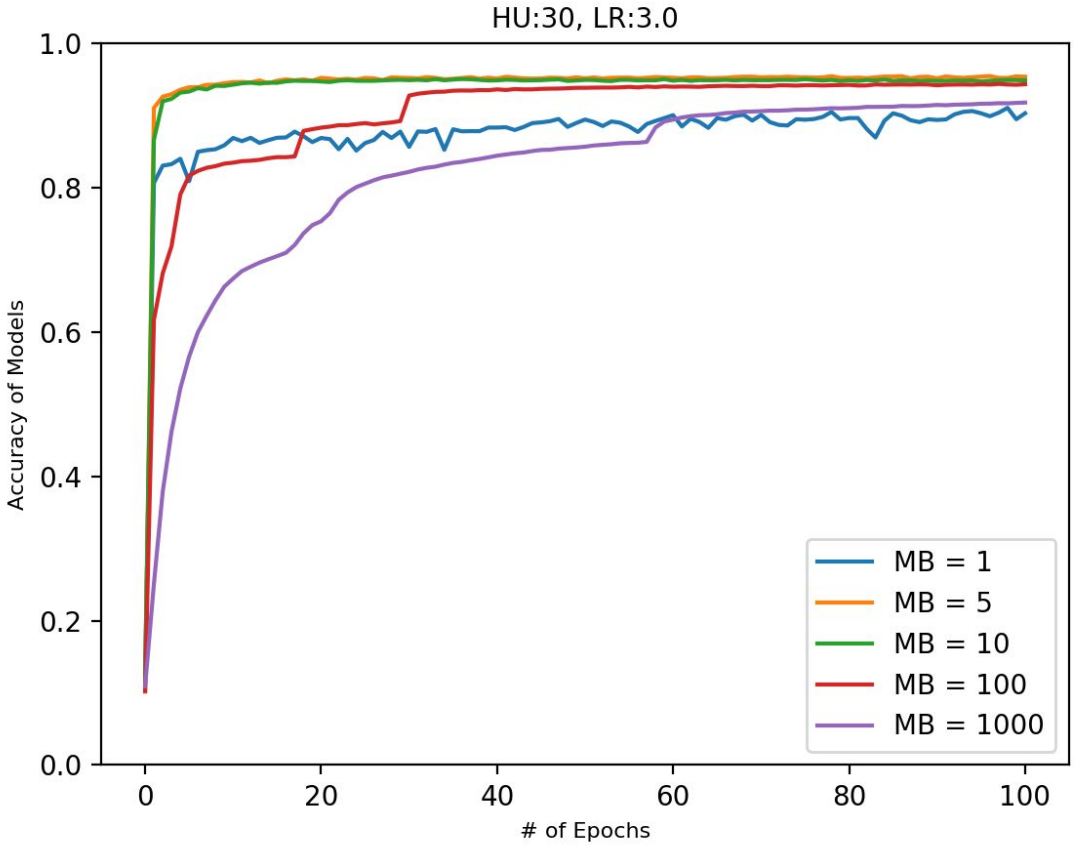


In the same case of increasing the size of hidden units, this performance has similar results as well. Increasing the learning rate can be beneficial for the network to learn, but increasing it too much will cause it to skip the optimal value. Making it too small will greatly increase the execution time because the network has to make many more iterations. In this plot, if we were to make the learning rate 0.01, the line will very gradually increase over 100 epochs. The steady increase over 100 epochs is due to the slow runtime and execution. If we were to make it 30.0, the network is learning at too fast of a rate that it "jumps over" the minimum that it is trying to reach (e.g. overshoot). Since it is overshooting every time it runs per epoch and overlooks the value that it wants to achieve, the network believes it has to "restart" its search, making the line stay consistent between 0.35 and 0.38. Having a learning rate of 1.0, 3.0 or 6.0 outputs the ideal results we are looking for.

Mini-Batch

When we want to train our network to recognize patterns, we are initially asked to set a batch size. The batch size is the number of samples that will be passed through to the network at one time. Let's say we are given images of handwritten digits. If we were to set the batch size to 10, 10 images of the handwritten digits will be passed as a group into the network per epoch at one time. If the 10 images were chosen from a training data size of 50,000, there would be 5,000 batches per epoch (from dividing *test data size/batch size* = *batches per epoch*). We can pass in individual pieces of data one by one into the neural network rather than grouping the data in batches but that will slow down the execution. The larger the batch size, the faster the network will train. The quality of the model may degrade as we set the batch too large and this can cause the network to generalize well on data it has not seen before. The batch size needs to be tested and tuned, just like the other hyper-parameters, based on how our specific model performs during training and how our machine is performing.

Effects on Performance



The larger the batch is, the faster the neural network can complete each epoch during training. Depending on the computational resources of our machine, it may be able to run more than one single sample at a time. If we set the batch size too high, our machine may not have enough computational power to process all the data. This will suggest us to lower the batch size. In this plot, we can see that the highest mini-batch size, 1000, gives us a gradual increase in accuracy. This is because the execution time is quite slow due to the large amount of images the network is reading at once. A large batch size allows the network to take bigger step-sizes, giving the optimization algorithm a faster progress. If we look at a lower mini-batch size, like 1, there is too much “compromise” the network is trying to do, resulting in no progressive increase over 0.90. Having a mini batch size of 5 or 10 outputs the ideal results we are looking for.

Conclusion

The purpose of the neural network is to break down a complicated question or problem and solve it in smaller pieces with steady progress. The question is quite simple -- does the image show a 0 or a 8? Ironically, it is a very complicated question because of how vast the dataset is. Humans can tell the difference easily between numbers but what will a person do if they were presented with 100,000 or even a million images of different handwritten digits? The process would take so much longer than it needs to be. By reading single pixels of an image, a neural network model can scan a very large dataset without any manual work and learn to recognize the many patterns that exist. It does this through a series of many layers, with layers that answer simple and specific questions about the raw input, and later layers that have a hierarchy of complex and abstract concepts.