Name(s):

Challenge: Artificial Neural Networks in Even More Depth

Machine Learning is Easy

TA: Anish Lakkapragada

Happy Neural Network November! In this challenge, you will see more of the math behind a neural network, especially in the more practical cases of having multiple features, explore why they are so powerful, and also learn about a less-famous neural network, the autoencoder. There's two simple problems along the way.

1 Neural Networks: Beyond Univariate Cases

1.1 Univariate Case

Here we explore the univariate case (one variable); we want to predict one variable from only one variable.

Let's define the dataset and labels. We have a (for now) univariate dataset X and its corresponding labels y and an (artificial) neural network with parameters θ in its prediction function $f(x_i, \theta)$ (similar to linear regression.) We similarly have some objective function J which we are trying to minimize.

Neural networks are essentially just compositions of linear neural networks, with some crucial activation functions in the middle. This means that in a neural network you first have inputs going through a linear regression, and then these subsequent outputs become inputs for the next linear regression (or the next layer.) However, in between passing the output as input the output goes through an activation function $\sigma(x)$ which is crucial to adding nonlinearity to the function as discussed before.

In the univariate case, we can represent the linear regressions in neural networks through vectors. The input data can just be a vector (list of numbers) $\vec{x}$ and the labels similarly are a vector $\vec{y}$. For a linear regression in the univariate case, the optimal parameters m and b in the function $\vec{y} = m\vec{x} + b$ are both scalars. We'll see how this changes in the multivariate case.

1.2 Multivariate Case

In the multivariate case we are using multiple variables to predict a single variable. In this case, our dataset would no longer would be represented as a vector; instead it would be a matrix $\mathbf{X}$ with N rows (N is the number of samples) with k features.

To represent linear regression in a multivariate case, the parameter m as discussed above would no longer be a scalar but a vector with a size of k. Let's say in our example we have 3 features, or input variables ($k = 3$). A visual show of how this works to create the desired prediction function for each sample is shown in Equation 1.

$$\begin{bmatrix} x_1^{(1)} & x_2^{(1)} & x_3^{(1)} \\ x_1^{(2)} & x_2^{(2)} & x_3^{(2)} \\ \cdots & \cdots & \cdots \\ x_1^{(N)} & x_2^{(N)} & x_3^{(N)} \end{bmatrix} \begin{bmatrix} m_1 \\ m_2 \\ m_3 \end{bmatrix} + b = \begin{bmatrix} m_1 x_1^{(1)} + m_2 x_2^{(1)} + m_3 x_3^{(1)} + b \\ m_1 x_1^{(2)} + m_2 x_2^{(2)} + m_3 x_3^{(2)} + b \\ \vdots \\ m_1 x_1^{(N)} + m_2 x_2^{(N)} + m_3 x_3^{(N)} + b \end{bmatrix} \tag{1}$$

$$\mathbf{X} \times \vec{m} + b = \hat{\mathbf{Y}}$$

What's depicted above is basically the crux of most machine learning. When you have multiple variables, you represent the input data as a matrix, the weights as a matrix/vector (more on this), and the outputs are a vector/matrix. Note that the outputs for $\hat{\mathbf{Y}}$ are exactly what we want - they are a weighted sum for each of the k input variables $x_1 \ldots x_k$ using the coefficients $m_1 \ldots m_k$ plus the bias.

From here, we'll work our way up to neural networks. One difference between neural networks and linear regressions is that when neural networks take in multiple variables (or only one) as input, they typically output multiple values (more than the prediction vector shown in Equation 1). This can easily be done by adding another set of coefficients in a second column of the weight $\mathbf{M}$ (capital because its now a matrix), which will lead to two columns in the final prediction - or two sets of values. Note that you can start with univariate data and end up with two sets of values (two sets of outputs) or start with multivariate data and go to a univariate predictions (as shown before.) Also note that that the bias b will now have two values and be a vector (added to each row) instead of a scalar if there are to be two columns / output lists.

Problem 1: Let's say we have data containing N samples each with k_1 features. If, after a linear regression, we want to have k_2 outputs for each sample, what shape (how many rows and cols) is the input, weight, bias, and output terms?

Input ($\mathbf{X}$):
Weight ($\mathbf{M}$):
Bias ($\vec{b}$):
Output($\hat{\mathbf{Y}}$):

Storing data like this also matters computationally. Lots of hardware, especially GPUs, can do matrix multiplications extremely fast and so having our data represented in ways where it can be *vector*ized or on matrices in our computers is extremely useful for efficiency. This is notably because matrix multiplication can easily be parallelized.

1.3 The Easy Part: Activation Functions

The final part before putting it altogether are activation functions. These are what allows nonlinearity in neural neural networks, as discussed before. Some popular functions are the sigmoid/logistic function, Tanh, ReLU, and softmax.

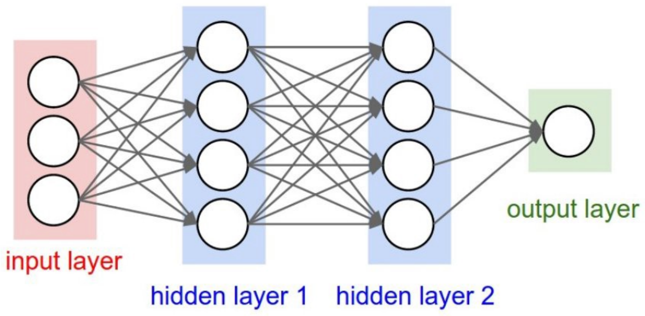
However, when we apply gradient descent to optimize each parameter in neural networks (also known as backpropagation), we have to use the chain rule for each layer (to be shown soon) and activation function. Thus, we want the gradients/derivatives to not be extremely small (e.g. near 0). When activation functions have this issue where their slope goes close enough to 0, they lead to vanishing gradients where the gradients disappear as these activation function derivatives drastically reduce the gradients when they pass through the enter neura network (backward.) This happens when there are asymptotes in the activation function. Ideally the activation functions are enough to add a shape more complex than a line but do not interfere with the other gradients (gradients should basically be 1 for the most part.)

Problem 2: Do some online research on the shapes of the 3 activation functions[1] above. Which activation function do you think will be the most successful and used the most and why?

[1]Softmax only makes sense to use in the last layer of a classification neural network. Thus ignore it.

1.4 Putting it Altogether

By now, it's clear neural networks are nothing more than glorified regressions feeding into subsequent regressions with activation functions in the middle. Oftentimes these individual regressions and activation functions are referred to as layers in a neural network. A diagram which shows this is given below.



It's important to note that this is a **bad** diagram. It does not show the activation functions nor the biases. Regardless, your going to see this anyways so it's worth knowing that in this diagram the three circles in the input layer represent the three variables for a single input, and the connections represent the weighted sum (before activation function) for each output node. The number of nodes / circles in each layer represents the amount of variables/outputs in that layer (for a single sample.) This diagram is pretty bad as you can clearly see.

1.5 Backpropagation: They Needed to Make Something Sound Fancy, again

Backpropagation is just gradient descent applied to neural networks. You simply apply the same gradient descent equation to all the parameters (all the biases + weights in each layer) with the same learning rate parameters. Also, you can forget about closed-form solutions from here as expanding the entire neural network composite function (with all the linear regressions and intermediate activations) is to complex to find an extrema for.

2 Why Neural Networks Became Famous: Universal Approximation Theorem

So far you may be wondering why neural networks even matter. More specifically, what complexity does having a second layer, as compared to just one layer (which would be a regression), add.

It turns out that the Universal Approximation Theorem (UAT) proves (with math that's too advanced for me) that a neural network with two layers (input layer + one hidden layer as shown above) can approximate any function (given that the layers have enough units.) This makes neural networks extremely powerful as any function or shape (linear, quadratic, or even weirder) can be approximated by neural networks. Note that this is on the training data, not the testing data.

3 Autoencoders: Neural Networks Beyond Prediction

Oftentimes we want to compress data. Meaning that if that our input data contains 1000 variables, we may want to be able to compress it in a way where we hold the same information in less variables. That is the idea of dimensionality reduction - taking data points (vectors) in a higher-dimensional space and compressing them down to a lower-dimensional space. Successful dimensionality reduction would be when points that were (relatively) far from each other in their original dimensions are still (relatively) far from each other in their lower-dimensional form. Failed dimensionality reduction would be when points in their lower-dimensional form are all extremely close or just form a line regardless of their distances or shape in their original higher-dimensional form.

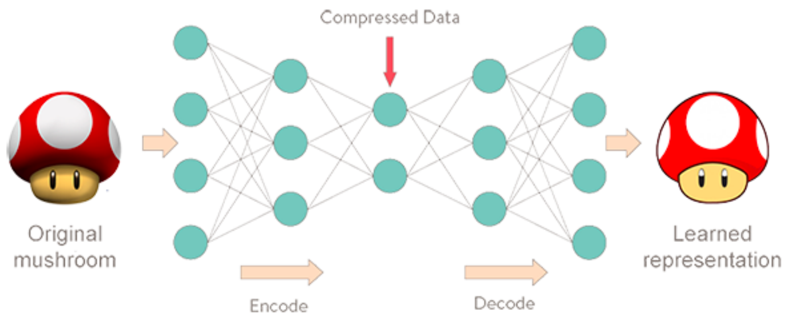


Figure 1: A visual of an autoencoder, where the goal is to reconstruct the input as precisely as possible in a way that compresses and decompresses information.

The above neural network is a special type of neural network known as an autoencoder. It's goal is to take in it's input, compress it down in the encoder layers to a smaller amount of nodes in the middle (a.k.a. bottleneck) layer and then use that information in the middle to recreate the input in the decoder layers. This means that both the input and labels are the same. This forces the representation of the data in the bottleneck layer to hold as much useful information as possible required to understand/differentiate a sample from others. The key principle is that similar inputs/images/data should be closer in their numerical representation.

What this means is that we can just feed data with a high amount of features into our autoencoder neural network and get the data in a reduced amount of dimensions by taking whatever values are held in the bottleneck layer. One way to explore this deeper is to train an autoencoder on image inputs, and then see what happens to the output image as you adjust the values in the bottleneck layer. I actually tried this on the MNIST (digit image) dataset, and you can find the demo here[2]. If you want to see a video of the output image changing when the bottleneck values are adjusted, click here.

4 Further Explorations

That's it for this challenge. There are other topics that I feel are important but somewhat boring to cover and that I don't think I could provide more insight on beyond conventional explanations. Some of these topics are listed below.
Topics:

- Prevention of Exploding Gradients Through Weight Initialization
- Gradient Descent Optimizers
- Neural Network's Sus History With Neural Networks in The Brain

[2]Link is found here: https://anish-lakkapragada.github.io/MNIST_LatentSpaceViz/. The autoencoder was actually made with the SeaLion framework.