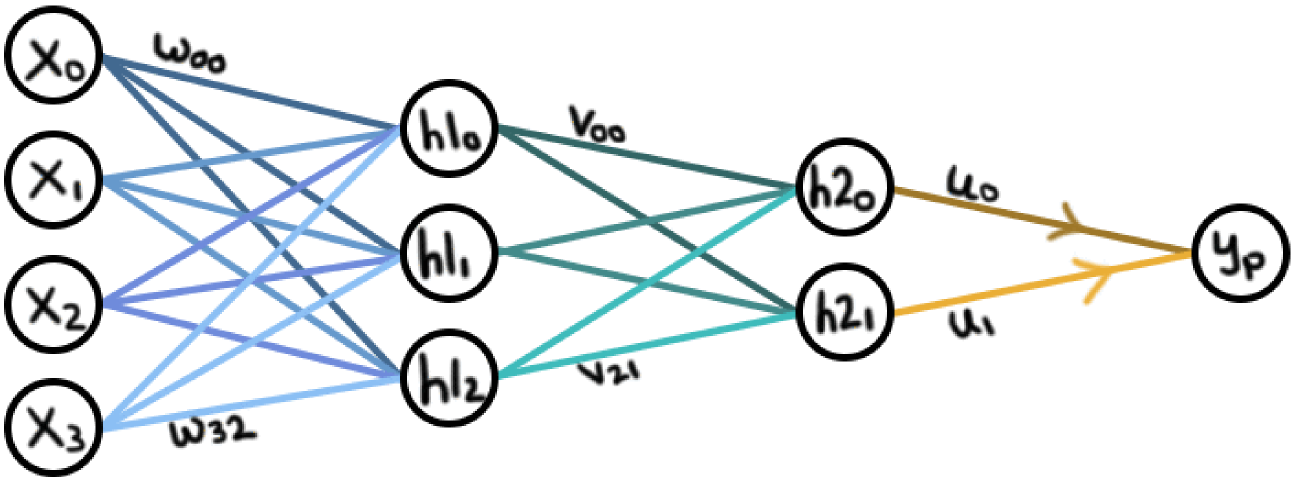
A Quick Introduction to Vanilla Neural Networks

Lauren Holzbauer
Dec 18, 2019 · 10 min read ★

Get the theory behind neural networks straight once and for all!



A 2-layer "vanilla" Neural Network.

Lauren Holzbauer was an Insight Fellow in Summer 2018.

In my last post, we went back to the year 1943, tracking neural network research from the McCulloch & Pitts paper, "*A Logical Calculus of Ideas Immanent in Nervous Activity*" to 2012, when "AlexNet" became the first CNN architecture to win the ILSVRC. We learned that deep networks could not be trained until the late-2000s, when researchers combined backpropagation with a better choice of activation functions and a smarter methodology for initializing the weights. But what exactly are activation functions and backpropagation?

In this post, we will answer that question by focusing on how neural nets really work, starting with the classic "vanilla" (researchers call them "vanilla feed-forward" neural around! In my next post, we will learn all about one of the specialty flavors of nets: the CNN.

Let's get started!

. . .

Neural networks were inspired by nature

In their 1943 paper, McCulloch and Pitts proposed a theoretical model that described the nervous system as a "net of neurons." As a whole, the network is capable of extremely complex computations, but each individual neuron is inherently very simple. Instead of being connected to every single neuron in the entire network, each neuron is only connected to its neighbor neurons via "synapses." By studying mappings of the human brain (see image below), we can see that neurons are often organized in consecutive "layers."

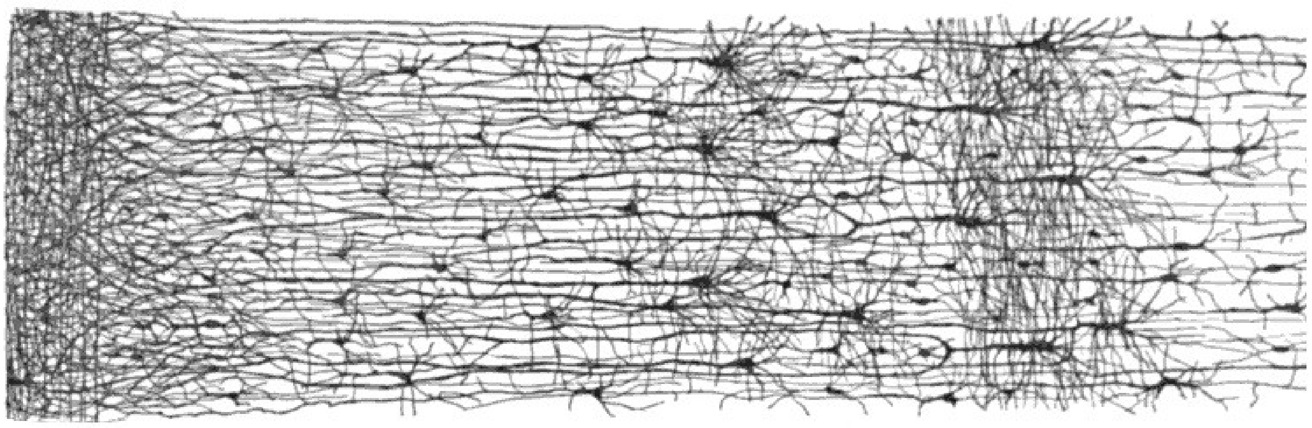


Figure 1 — Drawing of a cortical lamination showing a vertical cross-section of the cerebral cortex from a 1.5 month old human infant. It appears as if neurons are organized in consecutive "layers." Source: Wikipedia

The gist of all of this is that each neuron has an **input** and an **output** and neurons are organized in **layers**. Let's see how artificial neural networks in machine learning draw inspiration from this architecture we observe in nature.

. . .

When I first started learning about neural networks (NN), I found that thinking about them as an extension of linear regression was very helpful. So let's think back to linear regression and recall how the output is a weighted sum of the inputs. To be mathematically precise, the output is equal to the dot product of the input and the weight vector. The input vector, **x**, is connected to the output, **y_p** ("p" for "prediction"), by the weight vector, **w**.

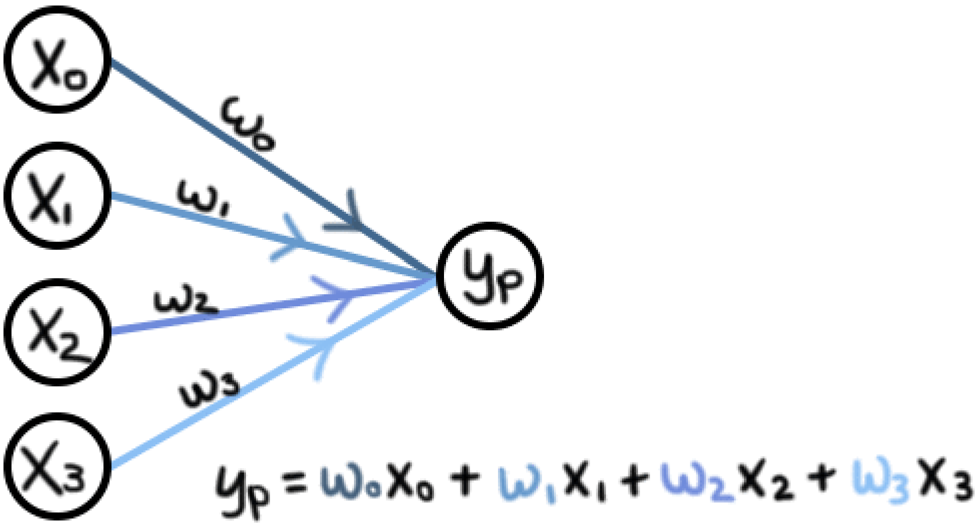


Figure 2 — Linear regression

A "vanilla" NN works similarly. The difference is an added extra layer between the inputs and output. When we implement a NN in real life, this extra layer is actually hidden from view since the NN takes care of all of those extra calculations behind the scenes. In Figure 3, we denote this "hidden" layer as "**h**." The input vector, **x**, is connected to the hidden layer by the weight vector, **w.** The hidden layer is connected to the output layer by another weight vector, **v**.

In Figure 3, the hidden layer, **h**, has three "neurons" (*h0, h1, h2*), but this is arbitrary; we could create a hidden layer with as few or as many neurons as we want.

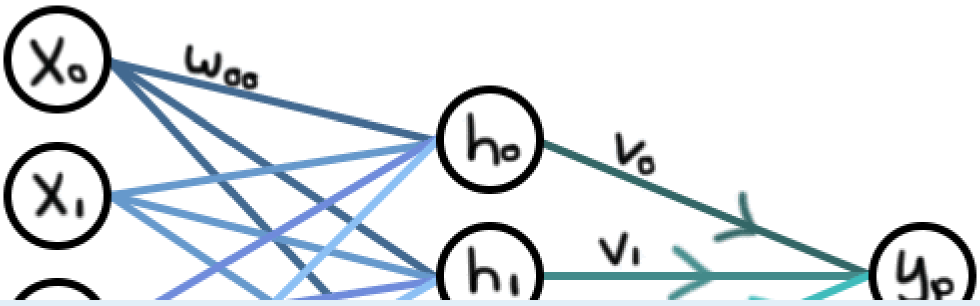


Figure 3 — A "vanilla" NN, with an additional layer.

Since a series of weighted linear sums is equivalent to one linear sum (just adjust the weights), the architecture in Figure 3 is actually equivalent to the architecture in Figure 2. So what *is* the difference between NNs and linear regression? **The trick that NNs use to make their architecture so powerful is that they apply a nonlinear "activation function" to the output of each layer.**

For now, we will denote the activation function as **A(z)**. The input for each neuron in the hidden layer is the input vector, **x**. The output for each neuron is just the result of the dot product of **x** and **w** plugged into the activation function, **A**. For clarity, let's write all of these sums explicitly:

$$h_0 = A(w_{00}x_0 + w_{10}x_1 + w_{20}x_2 + w_{30}x_3)$$
$$h_1 = A(w_{01}x_0 + w_{11}x_1 + w_{21}x_2 + w_{31}x_3)$$
$$h_2 = A(w_{02}x_0 + w_{12}x_1 + w_{22}x_2 + w_{32}x_3)$$

$$y_p = v_0 h_0 + v_1 h_1 + v_2 h_2$$

Figure 4 — The weighted sums for each step through the 1-layer network shown in Figure 3.

Activation functions: universal approximators

So what exactly does an activation function look like? Figure 5 shows the three primary nonlinear functions that have most frequently served as activation functions over the years:

1. the rectified linear unit (ReLU)

2. the logit function

3. the hyperbolic tangent function

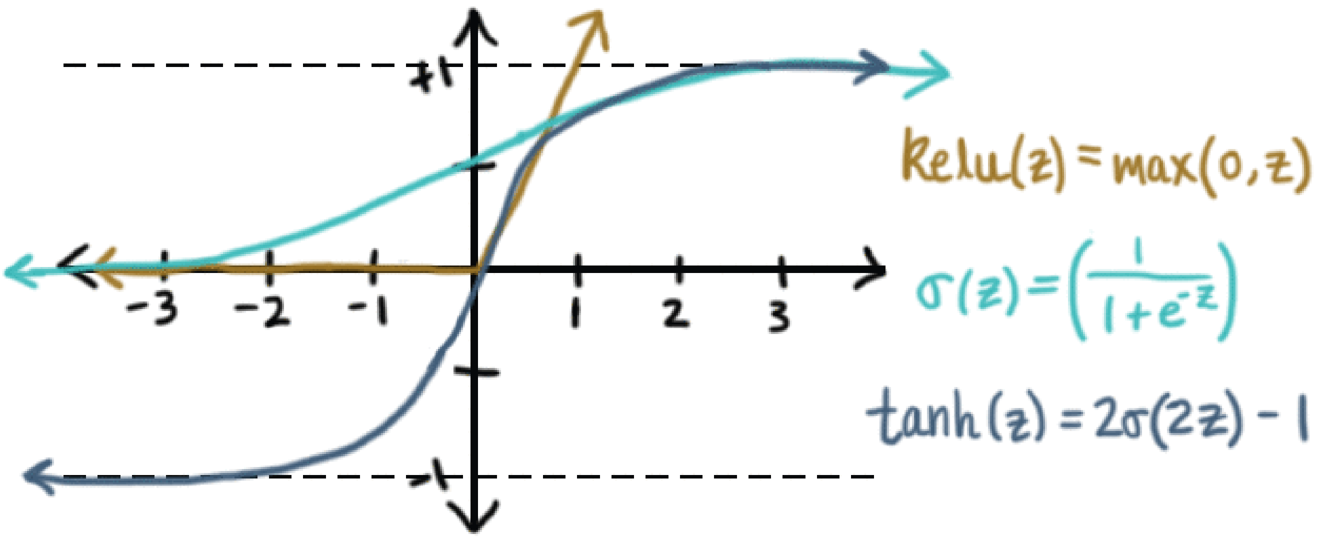


Figure 5 — A few activation functions.

In the beginning of neural net research, researchers favored the logit function. They assumed a sigmoid function would be a good choice since its values are bounded and it has a smooth gradient (the smooth gradient comes in handy during something called "backpropagation," which we'll discuss next!). Over time, researchers discovered that ReLU is actually a superior selection. You might find this surprising since ReLU has a discontinuous gradient and is a piecewise-linear function!

If the whole point of the activation function is to introduce nonlinear behavior, why does ReLU work so well? Though ReLU is a piecewise-linear function, it is indeed nonlinear as a whole. In fact, a single-layered ReLU network is considered a "universal approximator," in that it can approximate any continuous function (read more about the Universal Approximation Theorem here). Interestingly, it is still not theoretically clear why ReLU works so well! You can read more about all of the different options for activation functions and when to use each one here.

When a NN has two or more layers, it's called a "deep neural network," as in Figure 6:

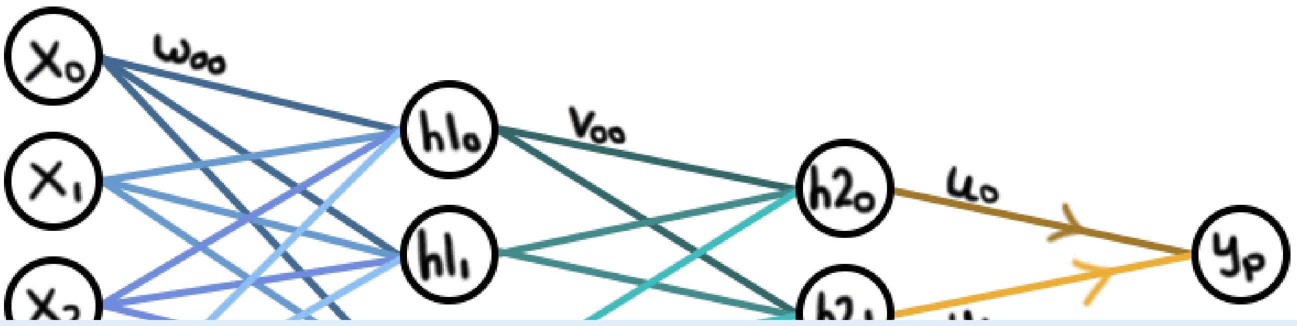


Figure 6 — A 2-layer "vanilla" NN.

As the network gets deeper and deeper, more and more weights are needed to make all of the connections. So how does the network figure out what the weights should be? Or in other words, how do we train a neural network?

. . .

Backpropagation: how to train your dragon

To better understand this training process, let's once again go back to how linear regression works. The weights are trained in a linear regression with an optimization algorithm called gradient descent. First, the algorithm randomly guesses initial starting values for all of the weights. Second, it uses those weights to make a test prediction for each training instance and computes the sum of squared errors between the predictions and ground truth values (i.e. the cost function). It then computes the gradient of the cost function and tweaks the weights in the "downhill" direction. With each baby step we make down the hill, we tweak the weights according to this update equation:

$$\vec{w}_{new} = \vec{w}_{old} - \eta \nabla_w J(\vec{w})$$

…where **J(w)** is the cost function and **eta** is the "learning rate" (size of each baby step). Once we have an updated guess for the weights, we can make a new prediction (that theoretically should be a bit closer to the ground truth value) and the cycle starts again. In short, gradient descent uses the gradient of the error to tweak its guesses for the weights iteratively (descending along the negative gradient, hence "gradient *descent*") until it arrives at the "bottom of the bowl" (i.e. minimizes the error) and achieves its best guess.

error
random initial value

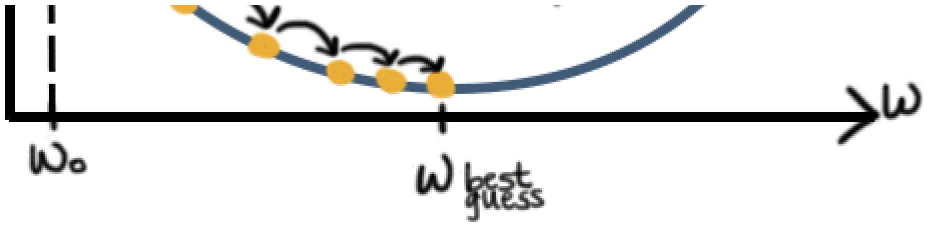


Figure 7 — A visualization of gradient descent. Initialize the weights with a random guess, then descend the gradient of error until you reach the bottom of the bowl.

NNs train their weights using a similar, but slightly more complicated methodology called backpropagation. Though the earliest semblances of NNs existed in theory for decades, researchers actually didn't develop a good strategy for training them until 1986, when Rumelhart et al. unveiled backpropagation in their groundbreaking paper, "Learning Internal Representations by Error Propagation."

As discussed above, a great way to train your weights is to iteratively optimize an initial guess with many gradient descent steps. Backpropagation performs one gradient descent step for each "batch" of training instances (the size of a batch is up to you). The number of batches it takes to go through all of the training instances once is equivalent to one "epoch." It can take many epochs of training to arrive at a good set of weights. However, don't train for too many epochs or you could be at risk of overfitting.

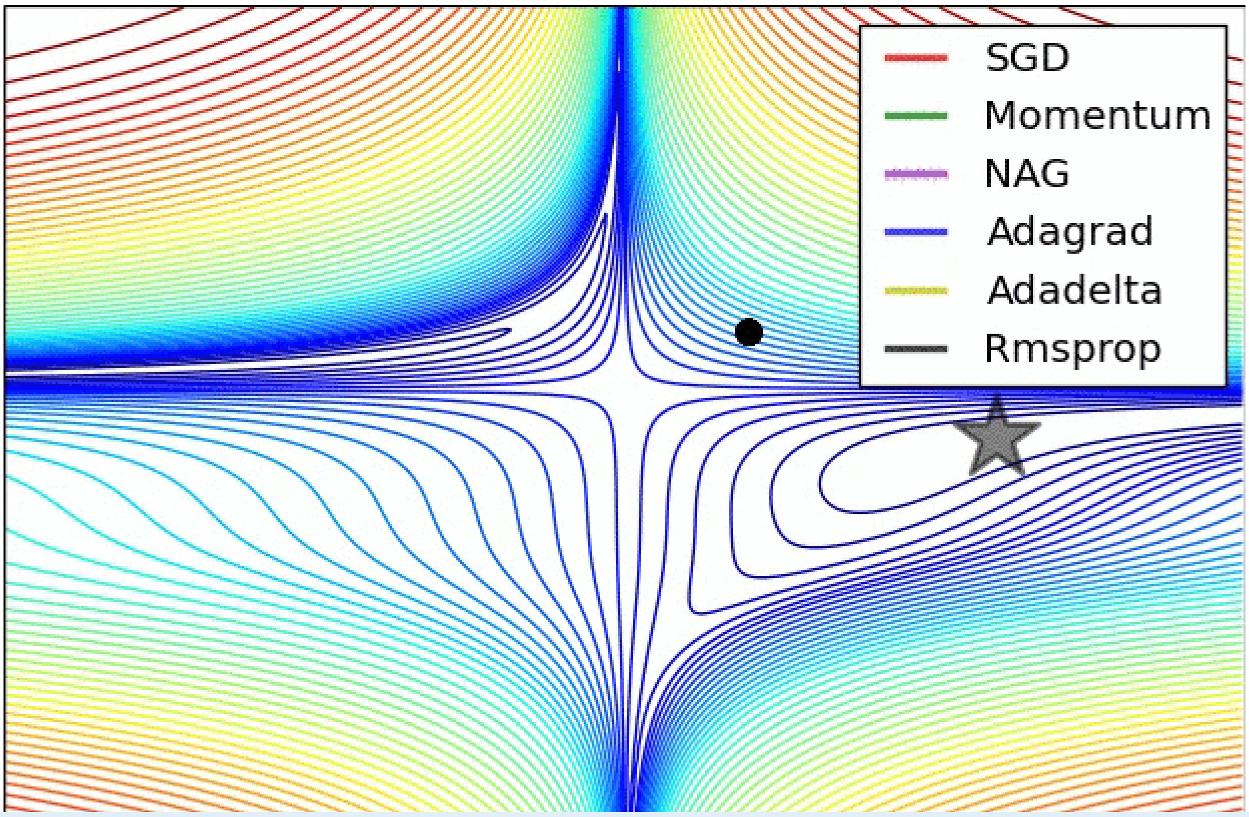
Unlike gradient descent, backpropagation does not make its initial guess for the weights with a simple random initialization. This actually doesn't work very well when training deep NNs! Different initialization parameters are used depending on your choice of activation function. Further details are beyond the scope of this introduction, but some initializations include "Xavier initialization" and "He initialization."

Once the weights are initialized, the algorithm begins training. For one batch of training instances, backpropagation makes initial test predictions by feeding all of the training instances in the current batch through the network in a "**forward pass**" ("left to right" in the network pictured in Figure 6), keeping track of the outputs of each neuron within each layer along the way. The output error is then computed by comparing initial predictions with ground truth values.

So now we have the total output error, but how do we compute the gradient of error so we can plug this into our update equation and we can tweak the weights? The error gradient is determined by making a "**reverse pass**" through the network, starting with propagate the error backwards through the network (hence "backpropagation") and figure out how much each neuron contributed to the total error. This allows us to directly compute the error gradient across each neuron and tells us how to tweak that neuron's associated weight. If you want more specifics on how this reverse pass actually works, check out the series of recorded lectures from Stanford's amazing course on Convolutional Neural Networks, "CS231n."

Finally, there are many different variations of the update equation above. Each variant is called an **"optimizer."** There is a plethora to choose from, including Gradient Descent (GD), Stochastic Gradient Descent (SGD), and Mini Batch Gradient Descent. Optimizers within the GD family are usually challenged in that the learning rate must be defined a priori and they can get stuck in local minima or around saddle points if the error function is not convex (as it is for linear regression). A better alternative is to choose an optimizer that utilizes an "adaptive learning rate," such as AdaGrad, AdaDelta, RMSProp, or Adam.

Figure 8 is an animation put together by Alec Radford (see his other animations here). It demonstrates the differences between the behavior of several optimizers applied to the same problem. In this case, each optimization algorithm is searching for the global minimum of Beale's function.



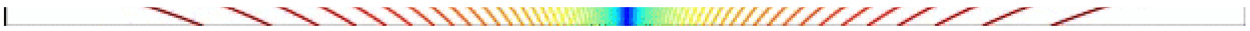


Figure 8 — Several optimizers as applied to Beale's function. The gray star indicates the location of the global minimum. Source: Alec Radford's post on visualizing optimization algorithms.

As you can see, Adadelta and RMSProp win the race while Nesterov Accelerated Gradient (NAG) and Momentum squiggle off and do other things before catching up. SGD is by far the slowest to find the global minimum.

In practice, any type of optimizer with an adapted learning rate will be best suited for training a deep NN. In particular, Adam is currently a popular choice; its fast convergence rate will help speed up training. If you'd like to read about optimizers more in depth, this article is a great one to start with.

. . .

Summary

That about wraps up this super fast introduction to vanilla neural networks! We learned how vanilla NNs are composed of an input layer, an output layer, and an arbitrary number of "hidden" layers in between that are "fully connected" (i.e. each neuron in one layer is connected to each neuron in the next layer).

The output of each layer is fed through a nonlinear activation function. This trick is what gives a vanilla neural net its nonlinear descriptive powers and makes it fundamentally different from linear regression. Researchers initially favored sigmoid functions for this purpose but have come to rely heavily on ReLU for its fast computation time.

Finally, a neural net automatically trains its weights using backpropagation. This process is similar to linear regression in that it involves iteratively optimizing an initial guess for the weights with many gradient descent steps. Each training instance is sent through the network in a "forward pass" and a prediction is made using the current guess for the weights. The total error is then calculated and propagated back through the network in a "reverse pass." This way, we can figure out which neurons are at most to blame for the error and we can update our weights accordingly in order to minimize error.

how they fundamentally altered the way we approach computer vision problems.

. . .

Are you interested in working on high-impact projects and transitioning to a career in data? Sign up to learn more about the Insight Fellows programs and start your application today.

Thanks to Holly Szafarek.

Machine Learning Neural Networks Artificial Intelligence Data Science Insight Data Science