# Intro to Neural Networks

*Tim Book*

GENERAL ASSEMBLY

## Welcome to Neural Net Week!

## First: A Brief History of Neural Networks

While they've recently gained popularity, they are not new. Research on what we now call **deep learning** can be traced all the way back to the 1940s. It's gone through several names:

- **Cybernetics** (1940s - 1960s)
- **Connectionism** (1980s - 1990s)
- **Deep learning** / **artificial neural networks** (term first used in 2006)

## So… why are we just hearing about them now?

The reason neural networks didn’t gain traction until recently is because all “good” implementations require ***A LOT*** of computational power. Even basic neural nets require a level of computing unavailable on 1960s “supercomputers”.

Small neural nets *were* feasible back then, they just didn’t perform as well as more classical approaches (linear regression, logistic regression, namely).

## What changed?

Three things led to the modern feasibility of neural networks that we enjoy today:

- **Increased computing ability.** The fact that you can train a multi-layer neural net on your personal laptop is incredible.
- **Increased algorithmic efficiency.** Smarter math leads to fewer computations needing to be done.
- **Increased availability of data.** Neural nets need a lot of data to work properly. There is no shortage of data these days.

## Why the name?

Initially, neural nets were inspired by the way our brains’ neurons work.

This inspiration didn’t last long.

And then we found out that’s not how brains work.

The name stuck anyway.

## MYTH

Neural networks mimic the way our brains' neurons process information.

## FACT

Neural networks are statistical models. Sometimes, researchers will incorporate loose inspirations from neural biology. Sometimes, they lead to improvements.

## Uses for Neural Networks

Neural nets have found numerous uses. It’s harder to find places where you can’t use a neural net. (Note there are many times you *shouldn’t*). What will be most interesting to us, is its flexibility to work with many *kinds* of data:

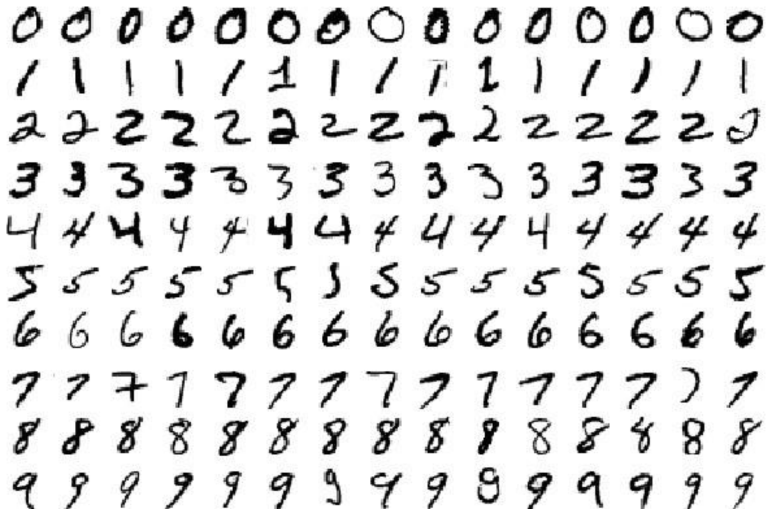
## Traditional Panel Data

Of course, neural nets can handle the usual panel-style data we've been working with up until now. **Neural nets can be used for both regression and classification.**

Working with the "usual" kind of data is done with **feed-forward neural networks** (sometimes written FNNs), which is what we'll focus on today. You'll learn to implement them in **Keras** this afternoon.

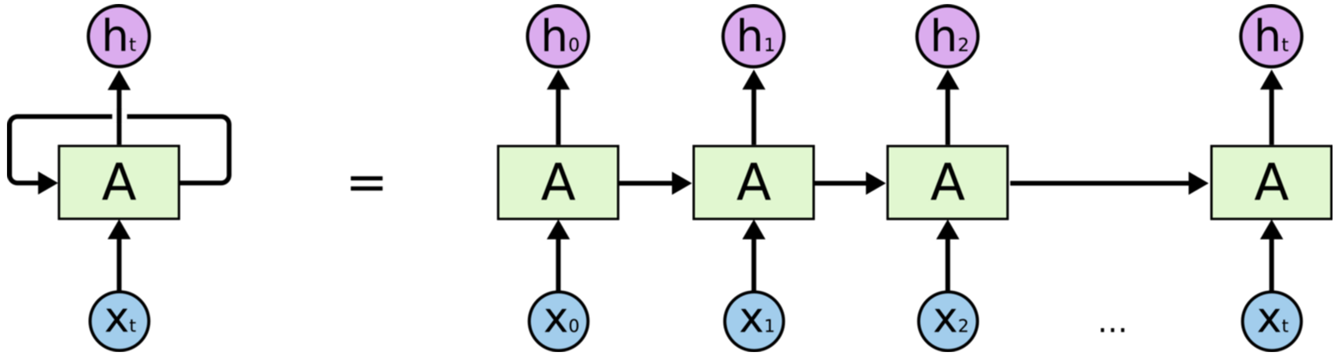
## Image Data

**Convolutional Neural Networks (CNNs)** are a variety of neural networks especially equipped to work with image data.



## Sequence Data

**Recurrent Neural Networks (RNNs)** are a variety of neural networks that specialize in working with sequence data, such as time series or natural language data.



## Recap: Logistic Regression

Let’s suppose I have some *x*-variables, and I want to use them to predict a binary 0/1 *y*-variable using logistic regression.

$$\operatorname{logit} p = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3$$

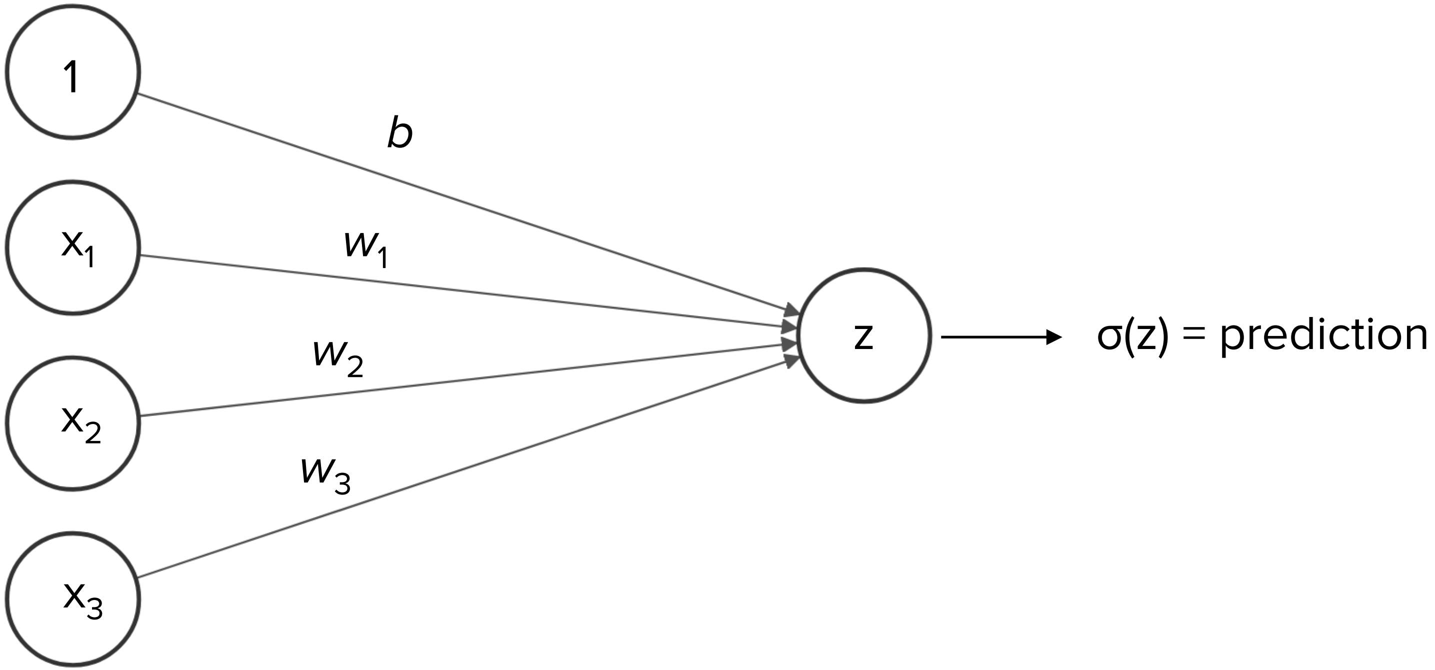
Let’s rewrite:

$$p = \sigma\left(\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3\right)$$

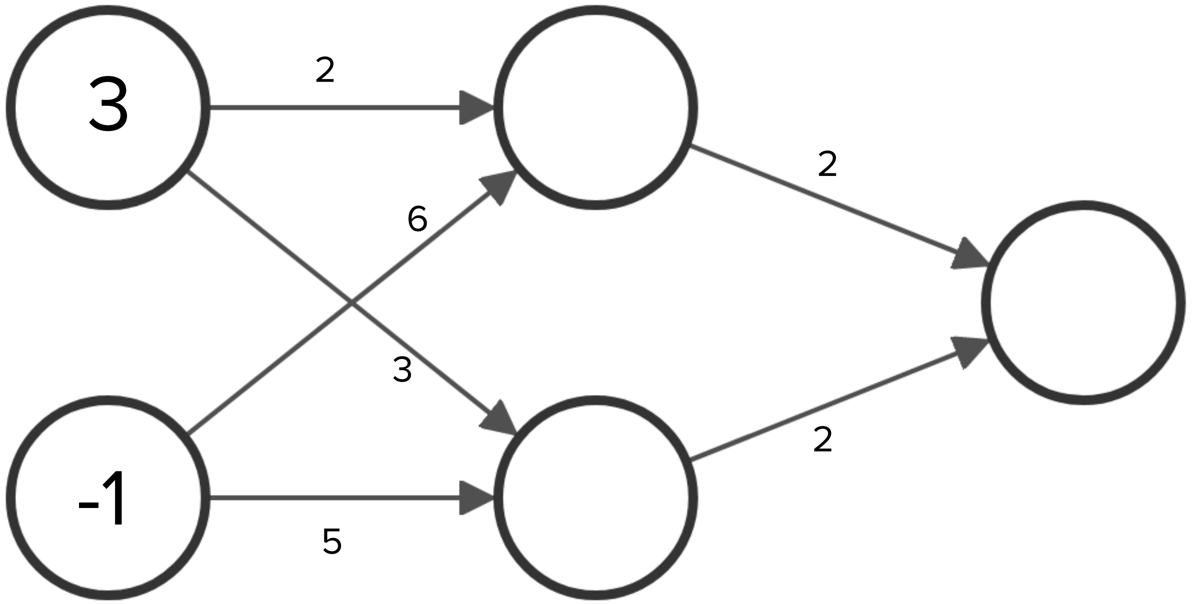
Let’s refer to the βs as “weights” with a “bias” term instead:

$$p = \sigma\left(b + w_1 x_1 + w_2 x_2 + w_3 x_3\right)$$

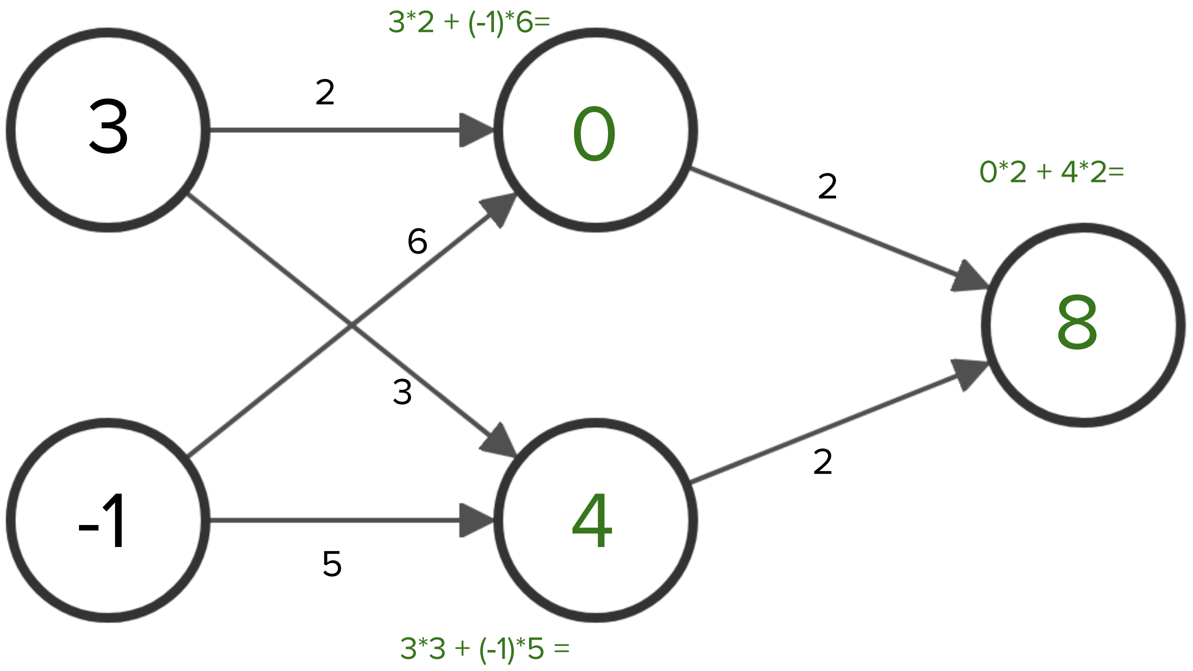
## Let’s draw this as a graph!



## Slack Quiz: What is my output here? (Assume no transformations)

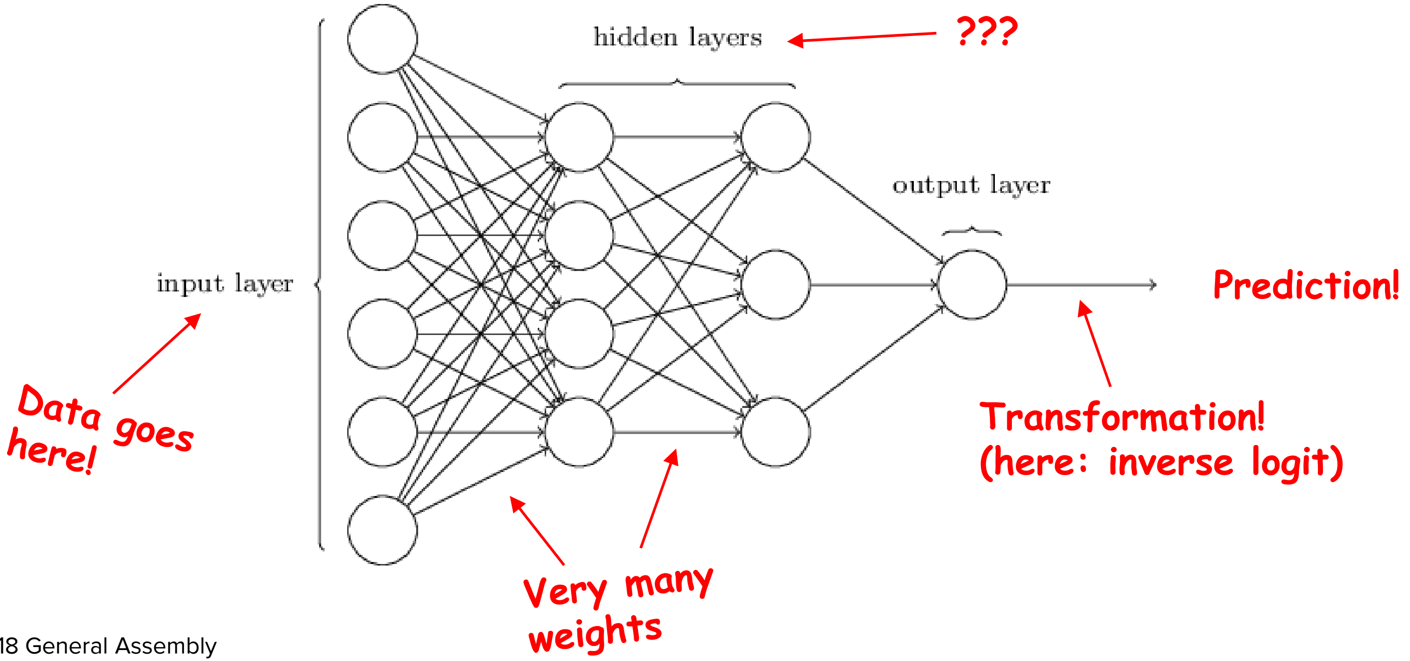


## Slack Quiz: What is my output here? (Assume no transformations)



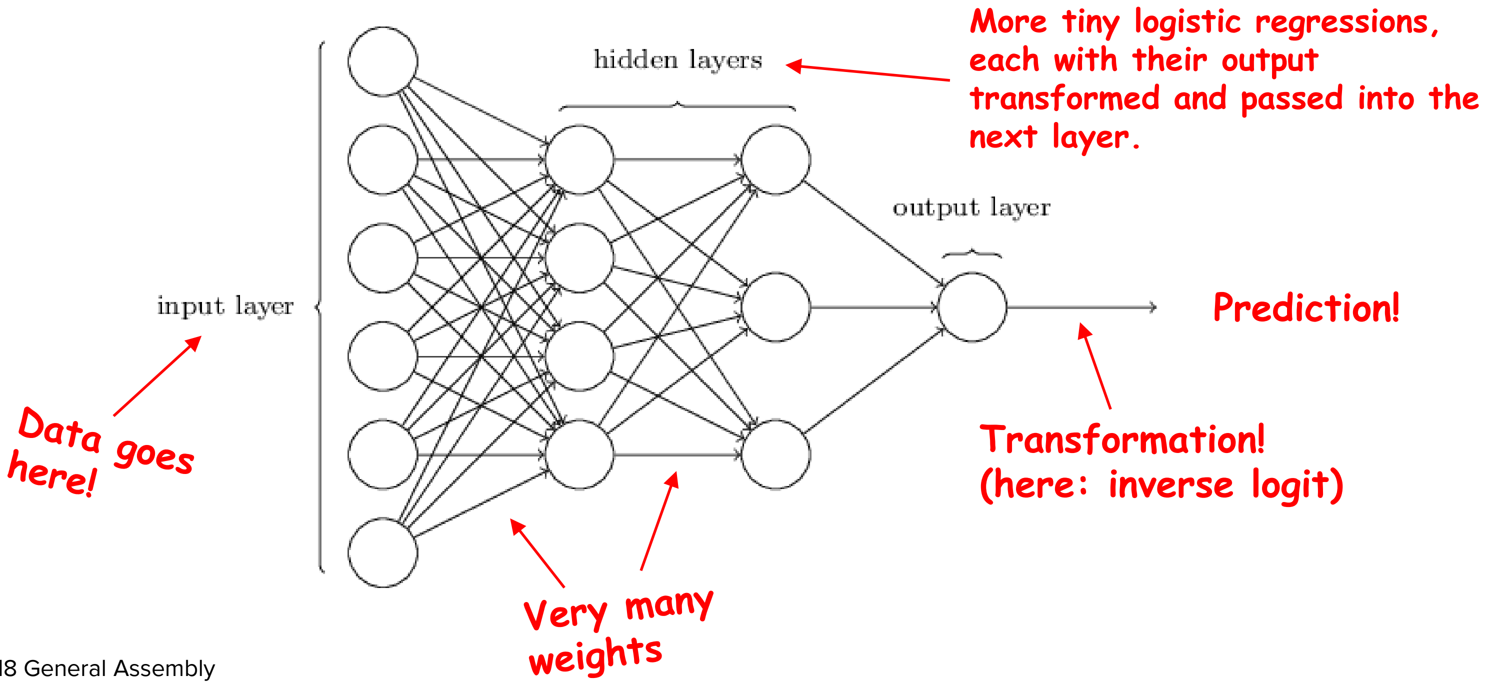
## What if we kept stacking logistic regressions?

Output of one gets transformed and then passed as input to another:



## What if we kept stacking logistic regressions?

Output of one gets transformed and then passed as input to another:


More tiny logistic regressions, each with their output transformed and passed into the next layer.
hidden layers
output layer
input layer
Prediction!
Transformation! (here: inverse logit)
Data goes here!
Very many weights

## Why did we do this?

$$p = \underset{\substack{\uparrow \\ \text{Curvy part}}}{\sigma}\left(\underbrace{b + w_1 x_1 + w_2 x_2 + w_3 x_3}_{\text{Linear part}}\right)$$

The linear parts allow our data to impact our result.

The the transformation (curvy) part allows us to learn more complex, non-linear relationships between x and y.

Multiple layers allow us to learn **even more wiggly and complex relationships within our data.**

Let’s play!

## Connecting hidden layers

We start with:

"Curvy function"

Weights

Bias

$$\mathbf{h}^{[1]} = a(\mathbf{W}^{[1]}\mathbf{x} + \mathbf{b}^{[1]})$$

Layer 1 output

This is then fed in as the *x*-variable to layer *two*:

$$\mathbf{h}^{[2]} = a(\mathbf{W}^{[2]}\mathbf{h}^{[1]} + \mathbf{b}^{[2]})$$

And so on until the last hidden layer (L), which give us our output:

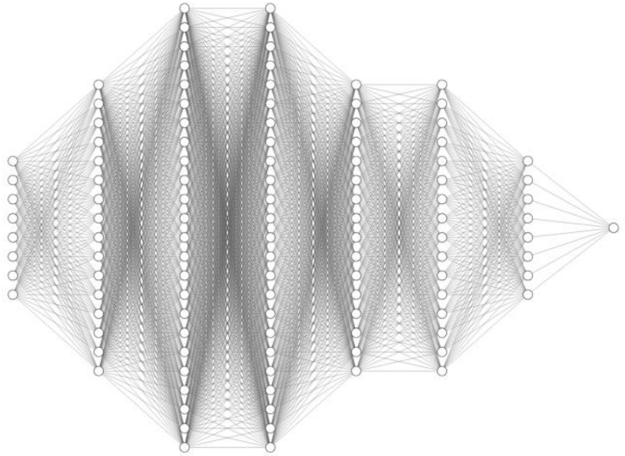
$$\hat{y} = a(\mathbf{W}^{[L]}\mathbf{h}^{[L-1]} + \mathbf{b}^{[L-1]})$$

Output (may actually be a vector)

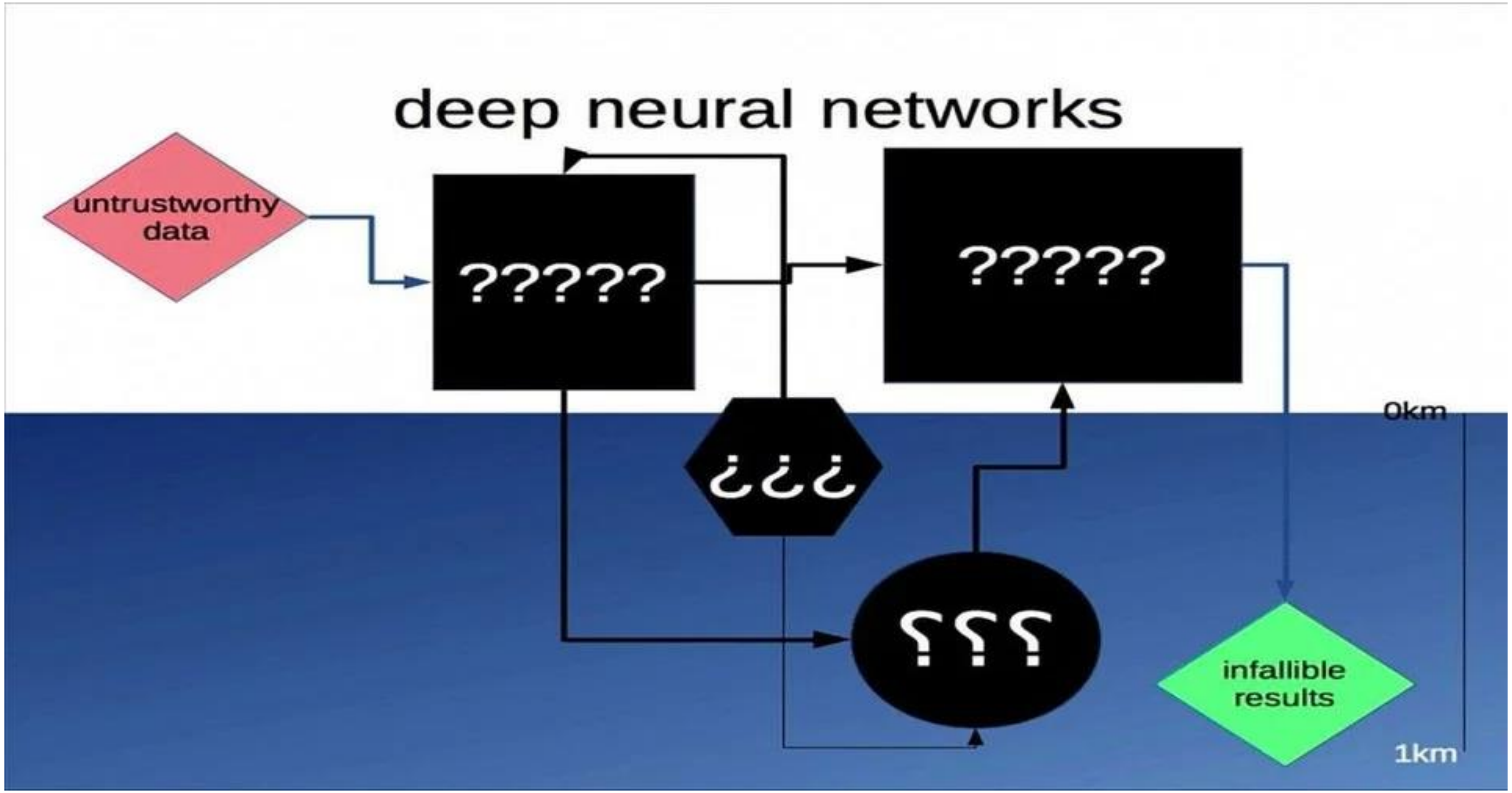
Might be a different "curvy function"

## “Deep Learning”

The term “deep learning” comes from the fact that you often have many hidden layers, making your graph very deep. Be careful though: **neural nets are prone to overfitting!**



## What have we done?!



## Don't fall into the trap!

## These Neural Networks are “Feed Forward”

The neural networks we’ve seen so far are:

- **Feed forward**: Calculations are passed down the network in one direction.
- **Fully connected**: Each node is connected to every node in the previous layer, and every node in the next layer.

Let’s play with FNNs here: **Neural Network Playground**

## Knobs & Levers

When constructing a neural net, you have many choices to make, such as:

- Number of hidden layers
- Number of nodes in each hidden layer
- Choice of activation functions
- Loss function

We'll discuss what each of these mean and how to choose them.

## Number of hidden nodes and layers

These control the complexity of your model. Too much of either will cause you to overfit. You’ll often need to **regularize** (more on this later this week).

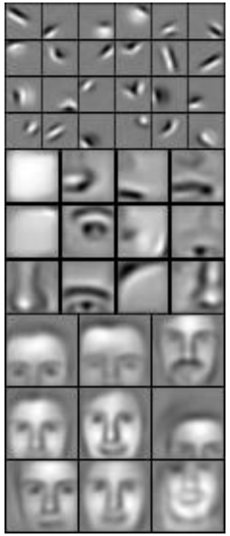
Unfortunately, there is no hard rule on how you should construct these. They will involve a painful amount of trial and error, as your situation will always be unique.

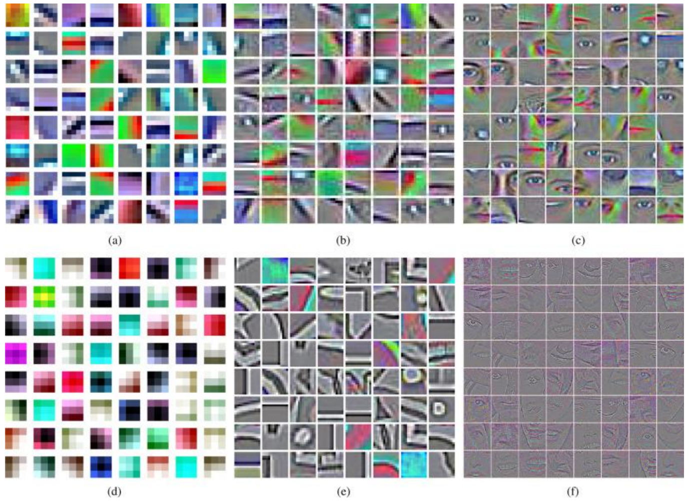
Here is some common advice, though:

## Hidden Layer Wisdom

**More hidden layers** = More overall complexity of relationship learned

**More hidden nodes** = More relationships learned (that will eventually be aggregated into prediction)





## Hidden Layer Wisdom (Feed Forward Only!)

In FNNs (fully connected/feed forward neural nets), there is very rarely reason to have more than two hidden layers. One is often sufficient.

A general rule of thumb:

**The number of hidden nodes should be somewhere between the number of nodes in the input and output layers.***

*Yes, there can be multiple nodes in the output layer, we will soon see.

## Activation Functions

An **activation function** is a transformation function applied to the output of a layer. That is, they're the **"curvey part"**.

In our special case of a logistic regression, the activation function was the inverse logit. In deep learning literature, this is called the **logistic sigmoid activation function**, or just the **sigmoid activation** (hence why we denoted it σ).

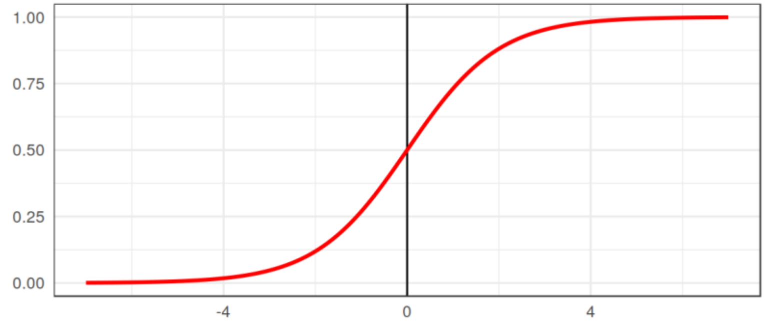
Here are some activation function flash cards:

## Sigmoid – Hidden Layers or Output Layer (binary classification)

Since the sigmoid function ranges from 0 to 1, it is often used as the output activation for binary classification models.

It has classically been used for hidden layer activations too, but it does often not perform as well as tanh and ReLU.

$$\sigma(z) = \frac{1}{1+e^{-z}}$$

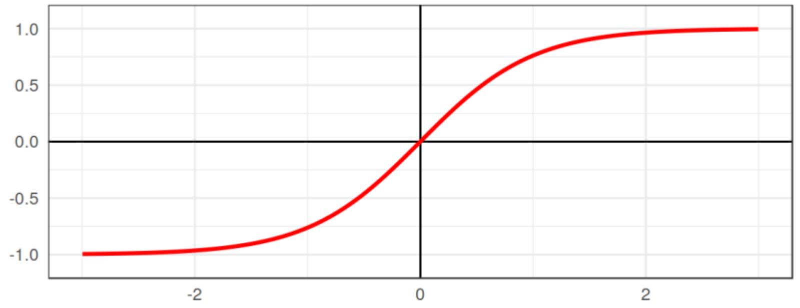


## Hyperbolic Tangent – Hidden Layers or Output Layer (binary classification)

The tanh function is pronounced “tanch” and varies between -1 and 1.

It is sometimes used as a hidden layer activation, but usually doesn’t perform as well as ReLU.

$$a(z) = \tanh(z)$$



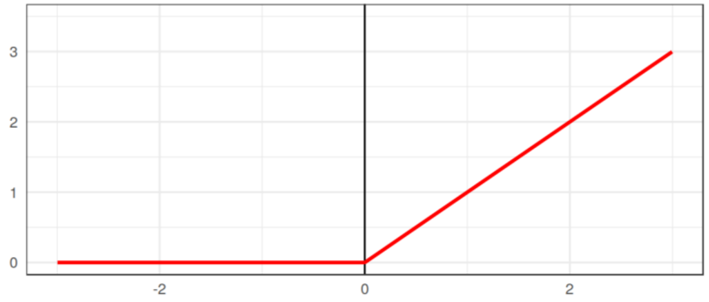
## ReLU – Hidden Layers

The rectified linear unit (ReLU) is the most common activation for hidden layers.

It was inspired by neurobiology, since real neurons don't activate until some stimulus threshold is met.

There are variants of the ReLU (see: Leaky ReLU).

$$a(z) = \max\{0, z\}$$

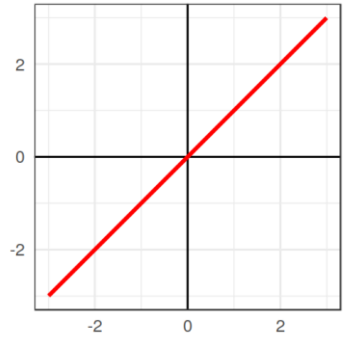


## Identity/Linear – Output Layer (regression)

This activation does nothing. It is often used as the output for a regression problem, since it maps to the whole real line.

**Never use the identity activation as a hidden layer activation!**

$$a(z) = z$$



## Softmax – Output Layer (multiclass classification)

The softmax is an extension of the sigmoid to multiclass classification. It is used as the output activation for multiclass classification.

It is vector valued, and varies between 0 and 1 for each element, and sums to 1. **This means the softmax is like a vector of probabilities!**

$$a(z_j) = \frac{e^{z_j}}{\sum e^{z_i}}$$

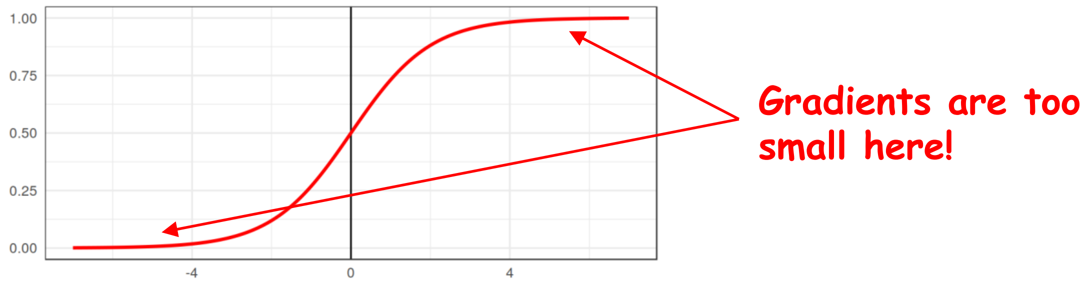
## Logic Behind Hidden Layer Activations

Activations in the hidden layer provide a transformation that allow the neural net to learn more complex relationships as calculations propagate through the network.

There is little need to use different activations across different hidden layers.

## Logic Behind Hidden Layer Activations

**Sigmoid** used to be the hidden activation of choice, but it has problems. Our optimization will be gradient-based, and the gradients are approximately zero for extreme values. This yields slow learning.



Once **ReLU** was discovered, the sigmoid was abandoned for hidden layers, and is now actively discouraged.

## Output Layer Activations

The output layer activations are particularly important because those decide what form our predictions take.

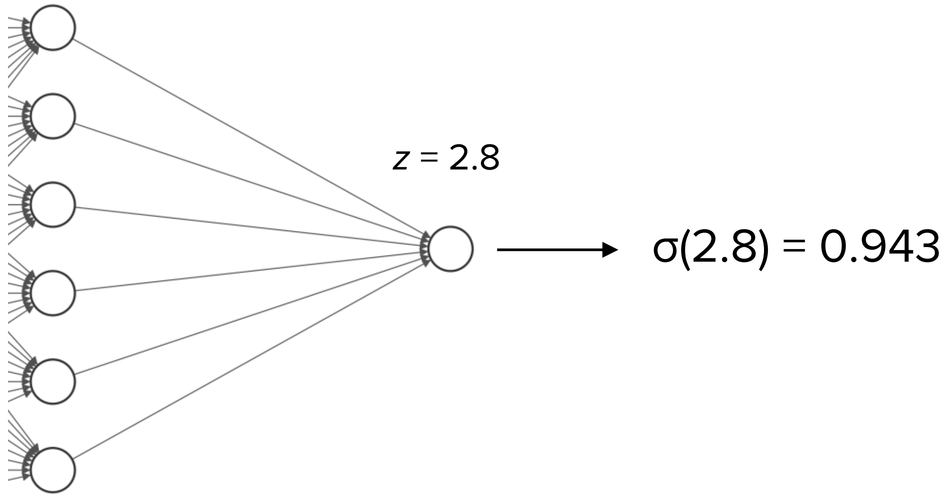
Your output activation should map onto the space of values occupied by your target variable.

What that means is...

## Output Activations: Binary Classification

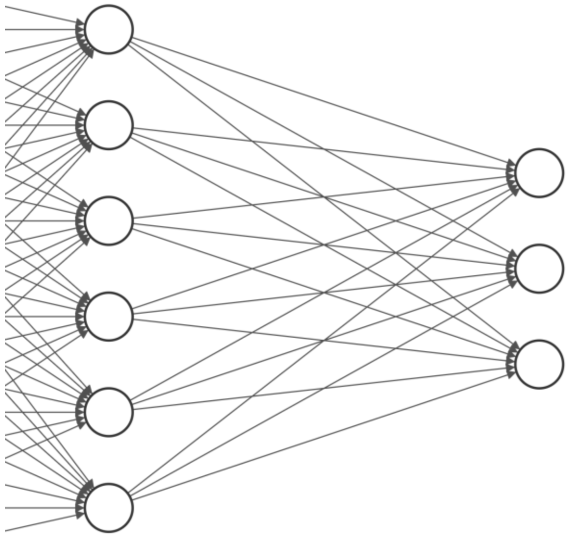
If your output variable is a binary class, your best bet is the **sigmoid activation**, since it maps directly onto 0 and 1.

Allegedly, some people also use the **tanh activation**, which maps between -1 and 1, and simply scale it afterwards.



## Output Activations: Multiclass Classification

For multiclass classification, the **softmax activation** generalizes the sigmoid to many outputs. **Notice three nodes in the output layer!**



$$\begin{bmatrix} 2.8 \\ 1.5 \\ 3.7 \end{bmatrix} \longrightarrow \text{softmax}\left(\begin{bmatrix} 2.8 \\ 1.5 \\ 3.7 \end{bmatrix}\right) \longrightarrow \begin{bmatrix} 0.27 \\ 0.07 \\ 0.66 \end{bmatrix}$$

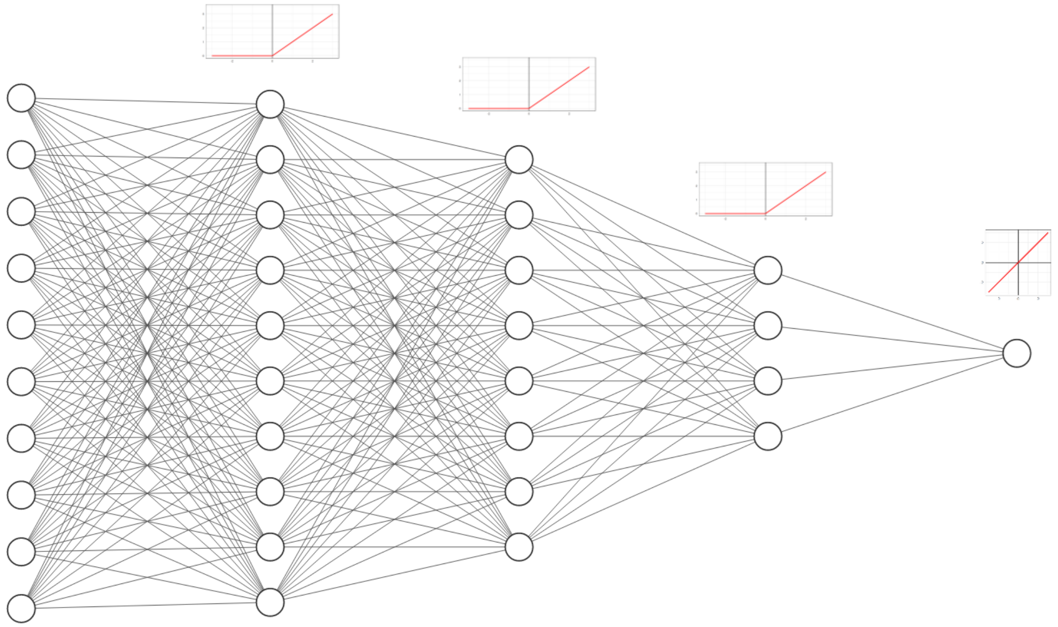
## Output Activations: Regression

When carrying out regression, there is no need to transform output, since you are predicting a quantity. You may either specify the **linear/identity activation**, or (in most software), you can simply decline to specify an activation.

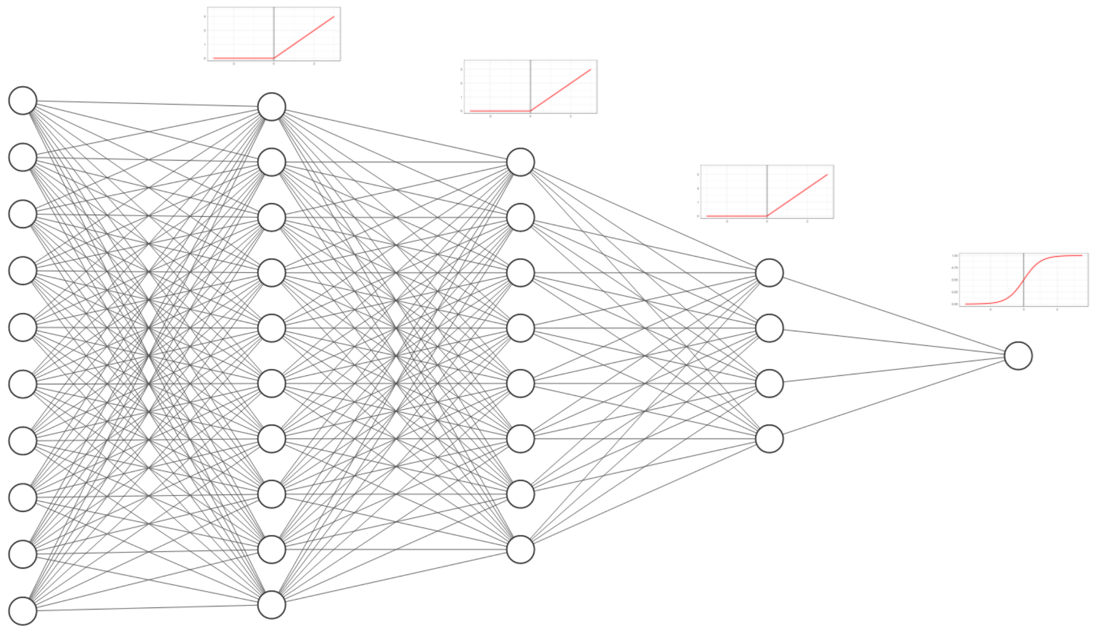
But… you may be underwhelmed. Solving regression problems using deep learning tends to underperform other methods (trees, linear models, etc.).

One trick that's becoming popular is to **MinMaxScale** your target variable and use a **sigmoid activation** in your output layer. Simply inverse transform your results to get predictions.

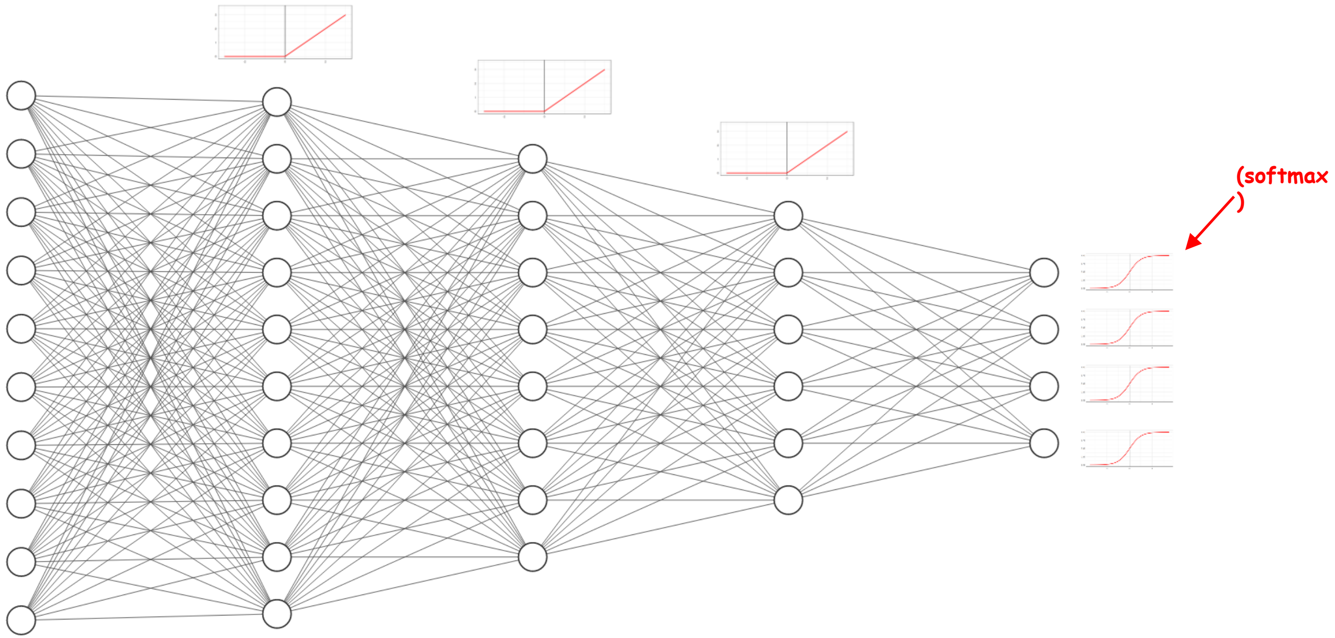
## Example: Regression with FNNs



## Example: Classification with FNNs

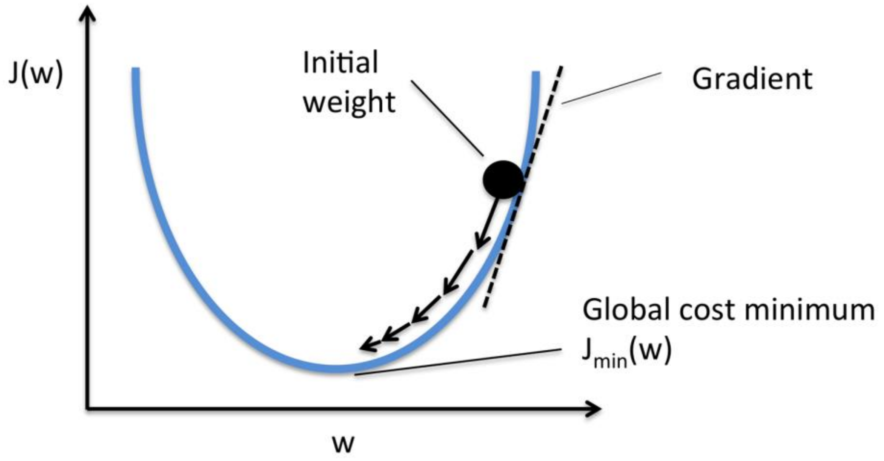


## Example: Classification with FNNs



## Loss Functions

Neural networks are fit using **gradient descent** with respect to some **loss function** that measures how wrong your results are. These don't vary very much, and they probably look a little familiar:



## Loss Functions

For regression, we often pick:

$$MSE = \frac{1}{n}\|\mathbf{y} - \hat{\mathbf{y}}\|^2$$

For binary classification, we often pick **binary crossentropy**:

$$-\frac{1}{n}\sum\left(y_i \log \hat{y}_i + (1 - y_i)\log(1 - \hat{y}_i)\right)$$

For multiclass classification, we have **crossentropy**, which is a generalization of binary crossentropy.

## Back Propagation

The reason neural networks are so computationally intensive is because of the algorithm that is used to fit them: **back propagation**, or **backprop** for short. It involves iterating between two steps:

1. **Forward pass** - Compute predictions and loss
2. **Backward pass** - Compute gradients based on predictions and update weights.

## Forward Pass

$$\begin{tikzcd}
x \arrow[dr] & & & & \\
& h^{[1]} = a(\mathbf{W}^{[1]}x + b^{[1]}) \arrow[dr] & & & \\
& & h^{[2]} = a(\mathbf{W}^{[2]}h^{[1]} + b^{[2]}) \arrow[dr] & & \\
& & & \ddots \arrow[dr] & \\
& & & & \hat{y} = a(\mathbf{W}^{[L]}h^{[L-1]} + b^{[L]})
\end{tikzcd}$$

## Backward Pass

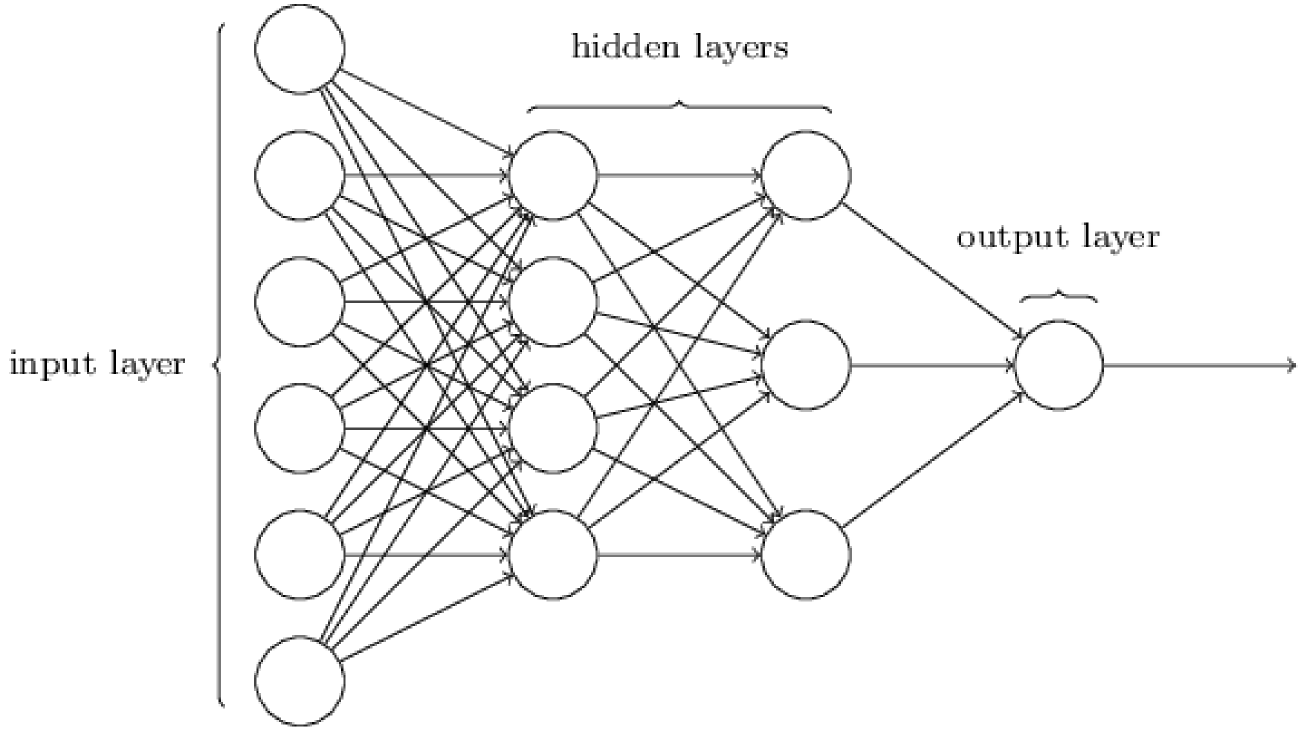
Compute loss

$$J(\theta) = L(y, \hat{y})$$

Update weights (let ***θ*** denote ***all*** model parameters)

$$\theta \longleftarrow \theta - \underset{\text{Learning rate}}{\alpha} \underset{\text{Vector derivative}}{\nabla} J(\theta)$$

## Slack Quiz: How many parameters need to be learned? (exclude bias/intercept term)



## Training can be slow

Neural networks are notoriously slow to fit. Even medium sized networks can have **over a million weights** to train. How can we speed this up?

1. Get access to a better computer (preferably with a GPU)
2. Use minibatch techniques
3. Use fancier optimization techniques

## Bigger and Better: GPUs

GPUs are essentially matrix multiplication machines. Neural network training takes a fraction of the time when using one.

It is very unwise to have a production-level neural network not taking advantage of a GPU.

You can rent GPU time from every major cloud computing resource.

## Minibatch Techniques

Actually, the *biggest* barrier to efficient training is the fact that we're trying to use all of our data at once!

Instead, at each iteration of gradient descent, we'll use a **minibatch** of the data. A minibatch will be a smaller subsampling of the full data. One run through the full dataset is called an **epoch**. This is pronounced "epic," unless you are British or Matt Brems, in which case it is "eepock."

For example, if you have 1,000 samples with a minibatch size of 100, you will run through 10 **training steps** in each epoch.

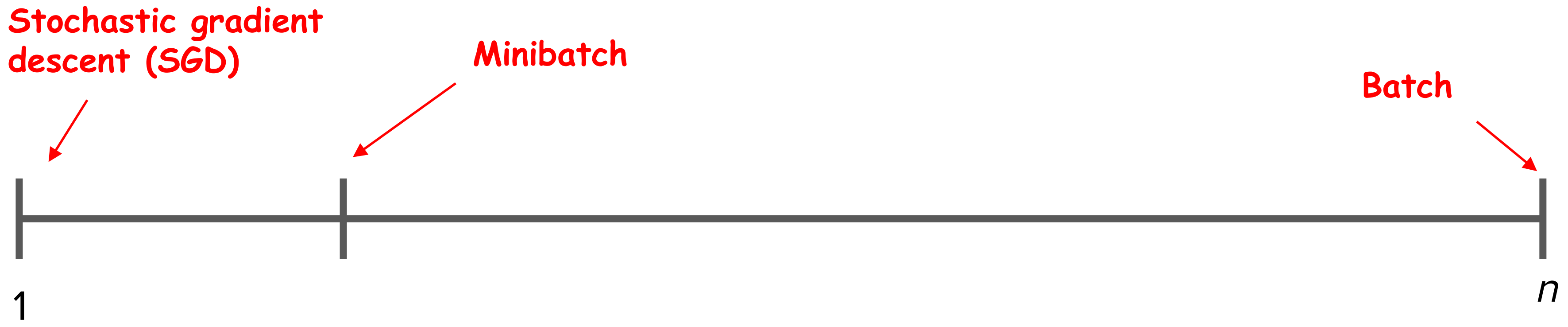
## Minibatch Advantages

Even though we aren’t using the full data at once, this method is actually superior to using the full dataset. There are (at least) three reasons for this:

1. It’s much faster.
2. Allows us to do “online learning” (train some of the model now, train more when we get more data later).
3. Helps us avoid **local minima** during optimization, which can be a big concern.

## Minibatch Size

The minibatch size is typically small. It is commonly as low as 16 or 32. An interesting note: minibatch sizes are typically a power of 2, since that works best with the inner workings of GPUs.



## Fancy Optimization

A lot of academic research goes into faster and more efficient techniques for gradient descent. You don't need to know how they work, but you'll want to recognize their names:

- **AdaGrad** (Adaptive Gradient)
- **RMSProp** (Root Mean Square Propagation)
- **Adam** (Adaptive Moment Estimation)

They each provide interesting updates to the usual gradient descent. You'll see RMSProp and Adam the most.

## Overfitting

Finally, we might be concerned with **overfitting**. There are lots of techniques for **regularizing** your neural net. We have a whole class devote to this:

- Penalty parameters at each hidden layer
- Dropout
- Early stopping

## Key Takeaways

On their surface, neural networks are easy to think about. Push data through a bunch of layers of nodes, at each layer transforming your input until you get a good prediction.

This process is very difficult to get working efficiently. Deep learning takes a great deal of patience to get up and running.

**It's gonna be a long week! But it's worth it. Neural nets are awesome and they are the future.**

## Welcome to Neural Net Week!