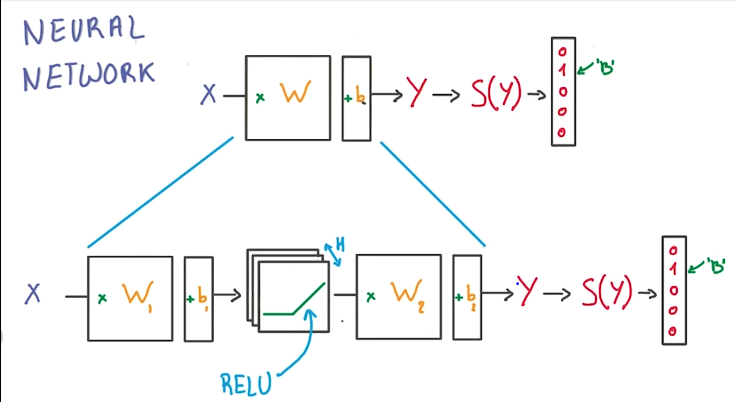


This model is relatively limited. You might want to use many more parameters than used here so that you can represent more relationships. This representation can only be used when representing inputs that have a linear relationship with the output, so an additive relationship.

If we wanted to represent a model where 2 inputs get multiplied, you need to use a non-linear model.

Linear models have the bonus of being easier to compute, due to being able to use matrixes in calculations, and use GPUs, which are designed for things like matrix multiplication.



This model is much more flexible due to having the RELU (the simplest non-linear function). You have more parameters you can use (H) and can represent non-linear relationships.

H represents the number of RELU units that you have in the classifier.

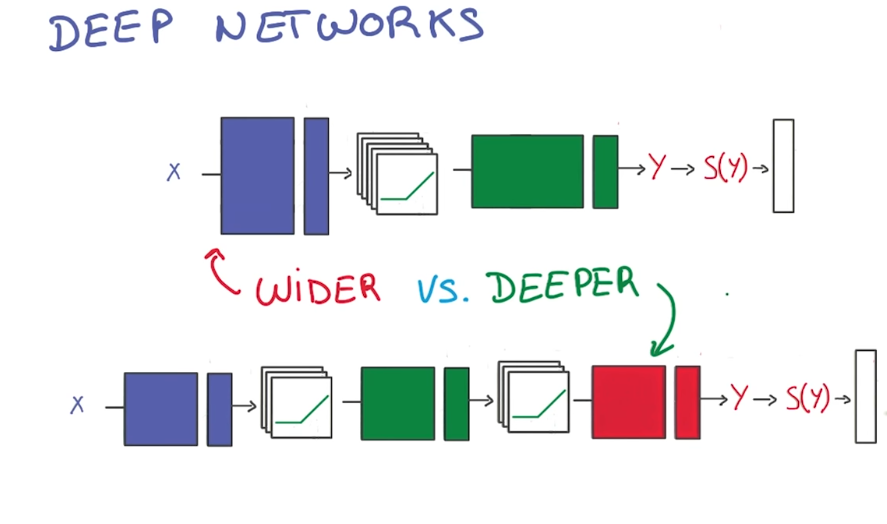
1. The first layer effectively consists of the set of weights and biases applied to X and passed through ReLUs (**Rectified Linear Units**). The output of this layer is fed to the next one, but is not observable outside the network, hence it is known as a *hidden layer*.
2. The second layer consists of the weights and biases applied to these intermediate outputs, followed by the softmax function to generate probabilities.

When an activation function (in this case the RELU) is non-linear, then a two-layer neural network can be proven to be a universal function approximator.

If you had a multi-layer neural network with linear activation functions, the entire network is technically the equivalent to a single-layer model.

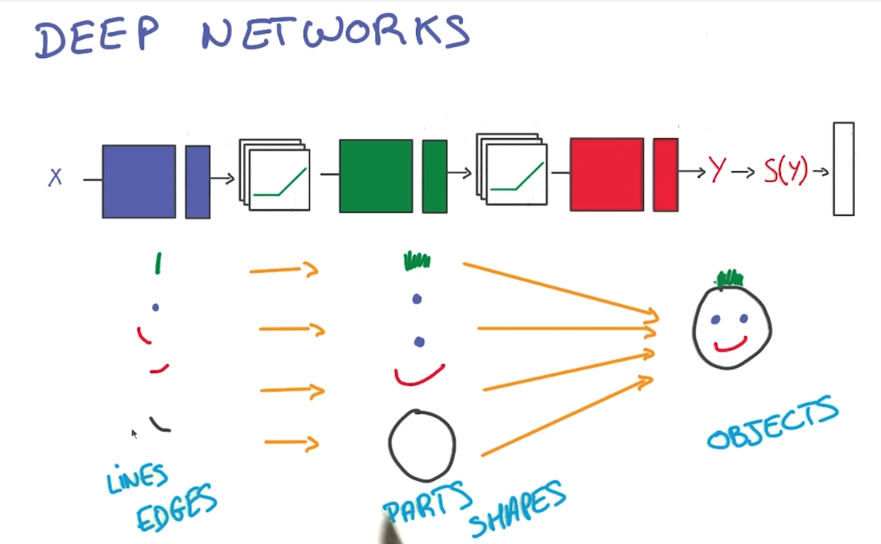
The H parameter (size of the hidden layer) is chosen to allow for much more parameters, making your model more flexible. The size can always be increased to add more flexibility, but it turns out that isn’t necessarily efficient due to the slower it becomes to train.

It is much better to add more layers, and to make your model deeper rather than to increase the size of your hidden layer and increase the width.



You typically have much greater performance with much less parameters when you have a deeper neural network rather than a bigger hidden layer (wide).

Therefore going deeper is much better for parameter optimization.



A lot of interesting natural phenomena that need to be observed tend to have a hierarchical structure that deep models capture well.

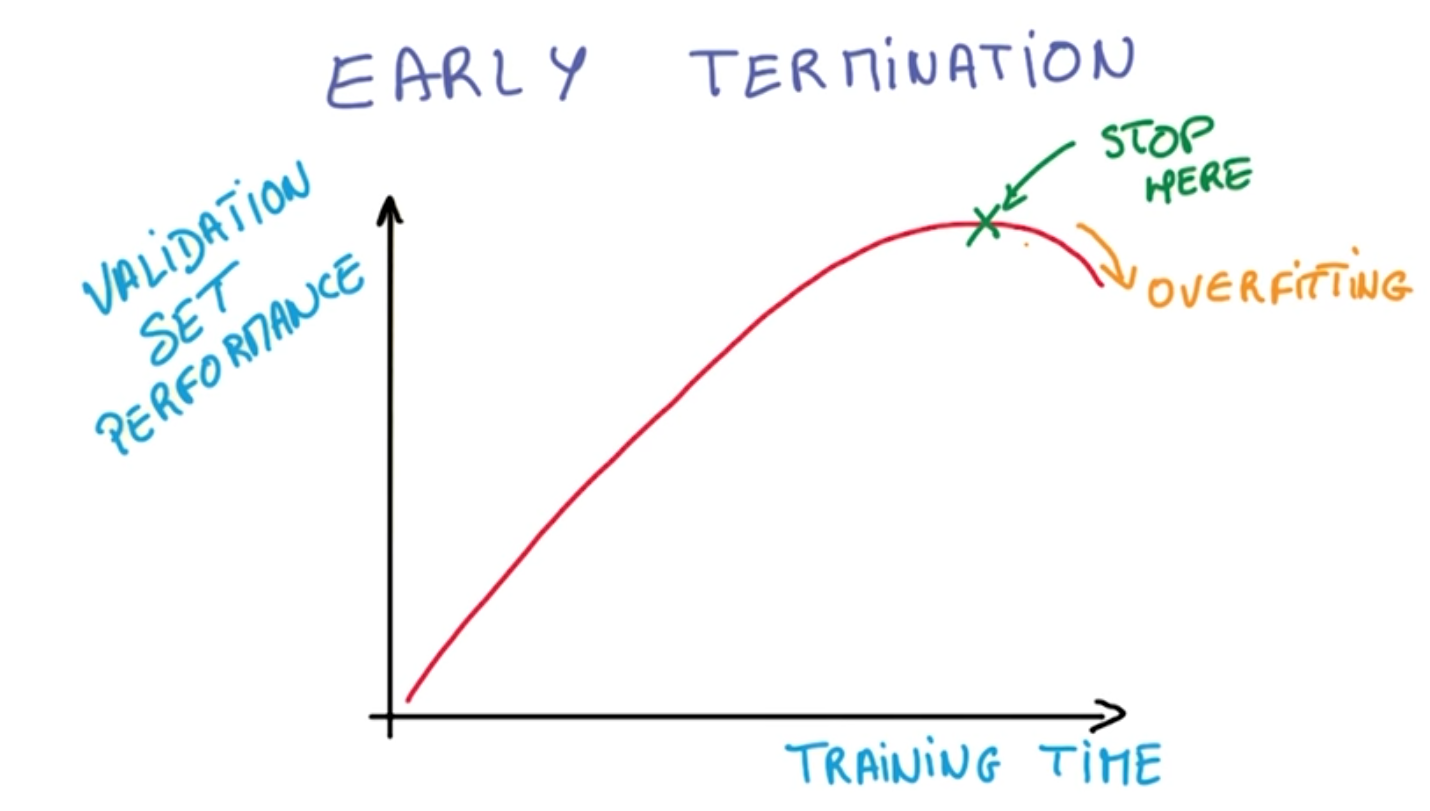
Building deep neural network to recognize things in an image for example tends to look for simple things at the lowest layers, then move up to more complicated shapes.

The model structure matches the data abstractions we expect in our data, meaning the model has an easier time learning them.

**Regularization**: Process to prevent overfitting (Or of introducing additional information in order to solve an ill-posed problem)

One method of preventing overfitting is Early Termination

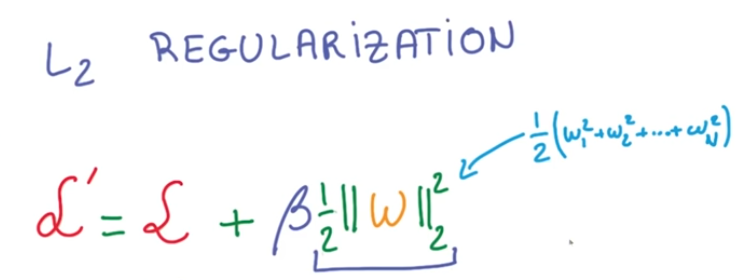
Early Termination stops training with your validating set and stops training as soon as you stop improving.



**Regularization** involves applying artificial constraints on your network that implicitly reduce the number of free parameters while also keeping it easy to optimize. (Think tight pants vs stretchy pants, fit better)

Reducing the free parameters to keep from overfitting is called **L­­­2 Regularization**

It involves adding another term to your loss function that penalizes large weight values.



The Beta variable is another Hyperparameter that we need to choose.

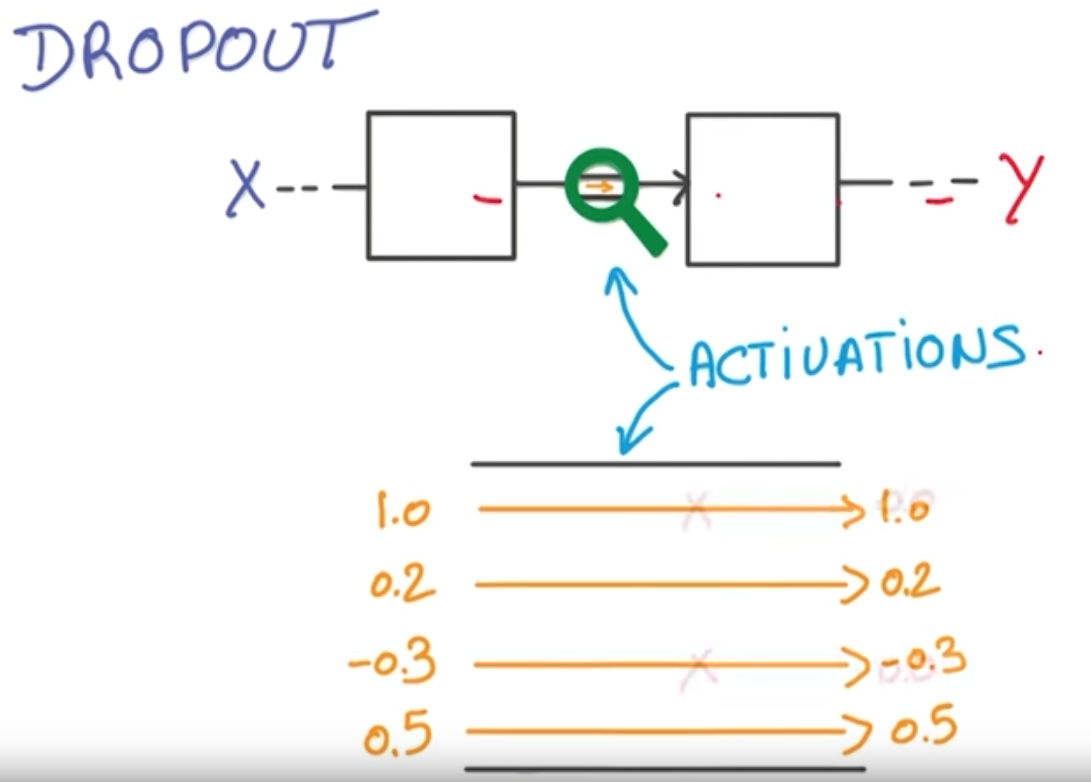
**L2 Regularization is great due to its simplicity:**

The new term is simple due to its Derivative being w.

It doesn’t affect the structure of your neural network.

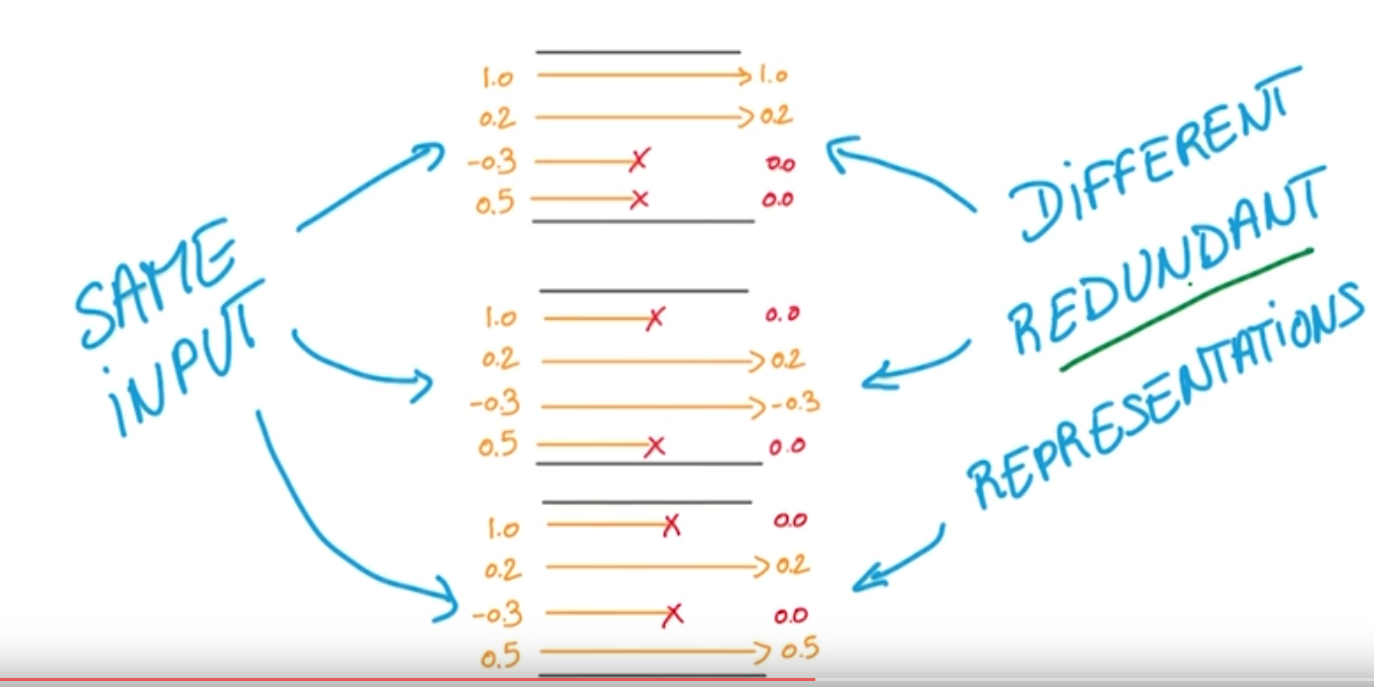
Another Important Technique for Regularization:

**Dropout:**



Imagine you have one layer of your network connected to another layer of your network. The Values going from one layer to the next are called **Activations**.

Now if you took those activations and Randomly, for every example you train your network on, set half of them to zero. Your network can never rely on any given activation to be present due to the chance that they may just be set to zero. The network is forced to learn a redundant representation for everything to ensure some information remains.



Making your network learn redundant representations sounds inefficient, but actually makes your network more robust and prevents overfitting. It also makes your network act like it is taking a consensus over an ensemble of networks.

If Dropout doesn’t work well for you, you should be using a bigger network.

To get the consensus over the many examples work you want the average of all training activations.

When it comes to getting the average activation, not only to you zero half of your activations to dropout, but you multiply the remaining by 2 and remove all the dropout data from the activation and have your neural network scale in those missing ones. The result is the average of those activations that is properly scaled.

Tensor flow has a function that does dropout for you on your tensors. It takes an input of the probability of keeping (not dropping) any given point of data. It also uses this to scale the activation by multiplying each remaining value by 1/keep\_prob to compensate for those missing values. A good starting value is 0.5

keep\_prob = tf.placeholder(tf.float32) *# probability to keep units*

hidden\_layer = tf.add(tf.matmul(features, weights[0]), biases[0])

hidden\_layer = tf.nn.relu(hidden\_layer)

hidden\_layer = tf.nn.dropout(hidden\_layer, keep\_prob)

logits = tf.add(tf.matmul(hidden\_layer, weights[1]), biases[1])

You Should only use dropout while training the model. While validating or testing your model, you should keep all the units so you can maximize the accuracy.