CORK INSTITUTE OF TECHNOLOGY
INSTITIÚID TEICNEOLAÍOCHTA CHORCAÍ

Machine Learning

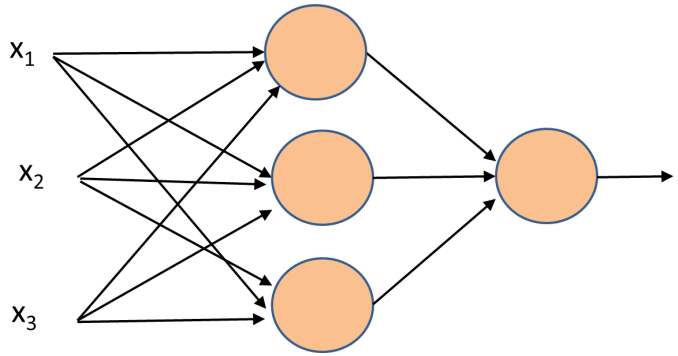
Machine Learning

Lecture: Neural Networks

Ted Scully

Vectorized Forward Pass for ANNs

$$A^{[1]} = w^{[1]}X + b^{[1]}$$
$$H^{[1]} = act(A^{[1]})$$
$$A^{[2]} = w^{[2]}\, H^{[1]} + b^{[2]}$$
$$H^{[2]} = act(A^{[2]})$$



The first operation we perform in the vectorised code is to multiply the **weight matrix** by the training **data matrix**.

This is a (p*n) matrix multiplied by a (n*m) matrix, which gives us back a (p*m) matrix. Notice rather than just multiplying a single example by the weights associated with each node (as we did previously) we are now multiplying all examples by the weights (vectorising the entire operation).

$$\begin{bmatrix} w_1^{[1](1)} & \cdots & w_1^{[1](n)} \\ \vdots & \ddots & \vdots \\ w_p^{[1](1)} & \cdots & w_p^{[1](n)} \end{bmatrix} \begin{bmatrix} x_1^1 & \cdots & x_1^m \\ \vdots & \ddots & \vdots \\ x_n^1 & \cdots & x_n^m \end{bmatrix} = \begin{bmatrix} t_1^1 & \cdots & t_1^m \\ t_2^1 & & t_2^m \\ \vdots & \ddots & \vdots \\ t_p^1 & \cdots & t_p^m \end{bmatrix}$$

Vectorized Forward Pass for ANNs

$A^{[1]} = w^{[1]}X + b^{[1]}$

$H^{[1]} = act(A^{[1]})$

$A^{[2]} = w^{[2]}\, H^{[1]} + b^{[2]}$

$H^{[2]} = act(A^{[2]})$

As normal we then add the bias. Finally, we obtain the output for each node in layer 1 for each training example, a (p * m) matrix (p is the number of nodes and m is the number of training examples).

$$\begin{bmatrix} t_1^1 & \cdots & t_1^m \\ \vdots & \ddots & \vdots \\ t_p^1 & \cdots & t_p^m \end{bmatrix} + \begin{bmatrix} b_1^{[1]} \\ \vdots \\ b_p^{[1]} \end{bmatrix} = \begin{bmatrix} a_1^1 & \cdots & a_1^m \\ \vdots & \ddots & \vdots \\ a_p^1 & \cdots & a_p^m \end{bmatrix}$$

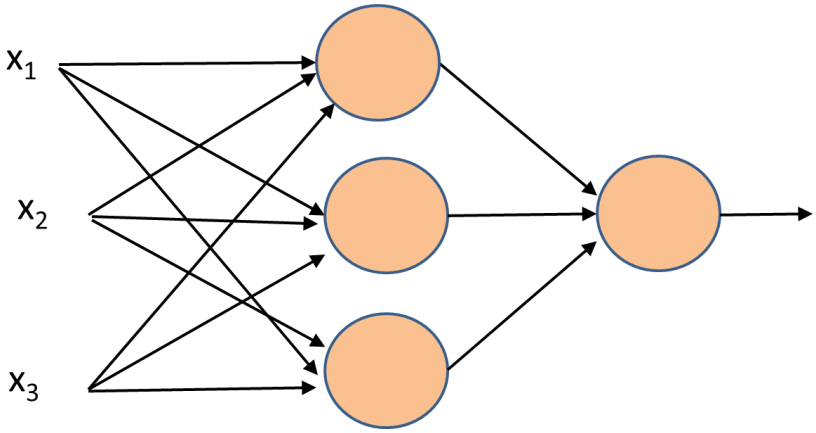
$$act\left(\begin{bmatrix} a_1^1 & \cdots & a_1^m \\ \vdots & \ddots & \vdots \\ a_p^1 & \cdots & a_p^m \end{bmatrix}\right) = \begin{bmatrix} h_1^1 & \cdots & h_1^m \\ \vdots & \ddots & \vdots \\ h_p^1 & \cdots & h_p^m \end{bmatrix}$$

Vectorized Forward Pass for ANNs

$$A^{[1]} = w^{[1]}X + b^{[1]}$$
$$H^{[1]} = act(A^{[1]})$$
$$A^{[2]} = w^{[2]}\,H^{[1]} + b^{[2]}$$
$$H^{[2]} = act(A^{[2]})$$

Now let's focus on the forward push operations for the next layer of neurons.
We take the matrix output of the first layer and multiply it by the weights for the second layer.

$$\left[w_1^{[2](1)} \cdots w_1^{[2](p)}\right] \begin{bmatrix} h_1^1 & \cdots & h_1^m \\ \vdots & \ddots & \vdots \\ h_p^1 & \cdots & h_p^m \end{bmatrix}$$



Notice there are p weights in our weights matrix for the 2nd layer of the network. Each neuron in the first layer (of which there are p) is connected to the single neuron in the second layer.

Vectorized Forward Pass for ANNs

$A^{[1]} = w^{[1]}X + b^{[1]}$

$H^{[1]} = act(A^{[1]})$

$A^{[2]} = w^{[2]}\, H^{[1]} + b^{[2]}$

$H^{[2]} = act(A^{[2]})$

$$\left[w_1^{[2](1)} \cdots w_1^{[2](p)}\right] \begin{bmatrix} h_1^1 & \cdots & h_1^m \\ \vdots & \ddots & \vdots \\ h_p^1 & \cdots & h_p^m \end{bmatrix} + \left[b_1^{[2]}\right] = \left[a_1^1 \cdots a_1^m\right]$$

$$act\left(\left[a_1^1 \cdots a_1^m\right]\right) = \left[h_1^1 \ldots . h_1^m\right]$$

Building Neural Networks - Softmax Activation Layer

- All the examples we have looked at so far were concerned with binary classification and as such we were using the Sigmoid activation function in the output layer.

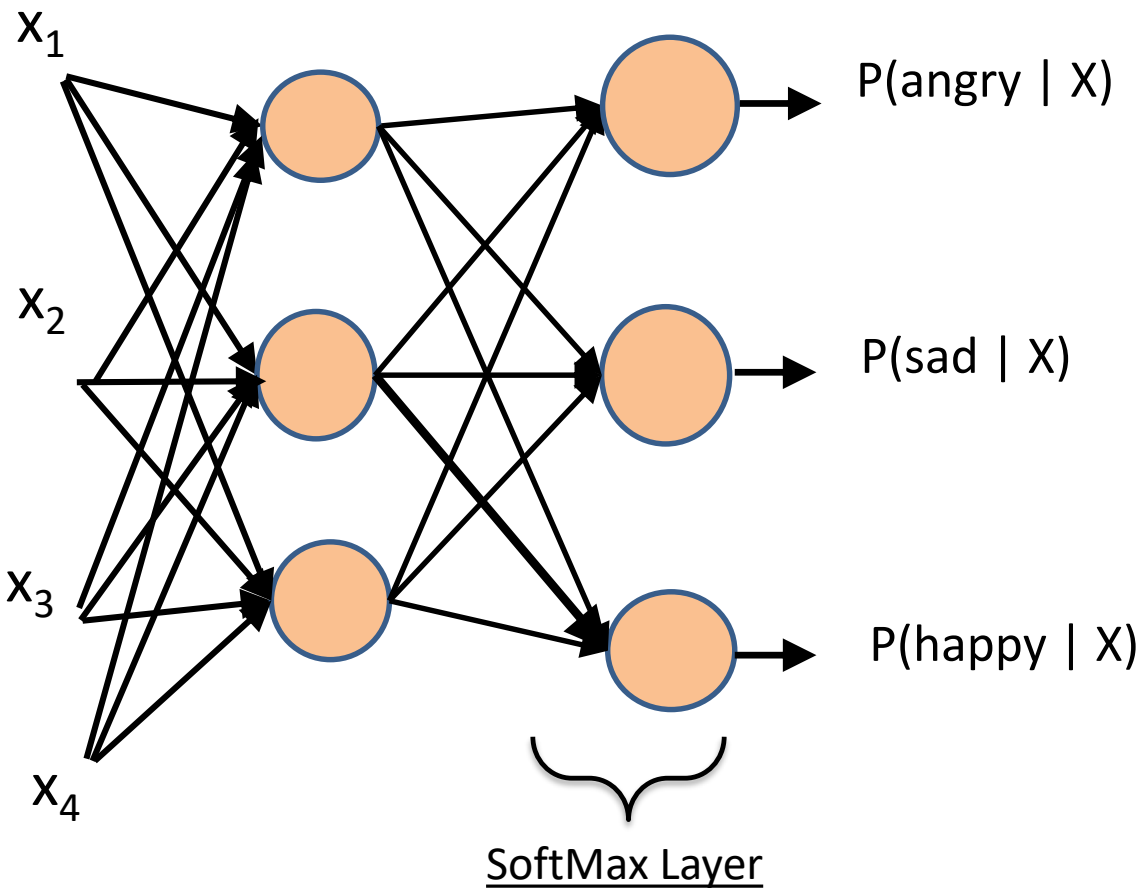
$$\sigma(x_j) = \frac{e^{x_j}}{\sum_i e^{x_i}}$$

- However, the Softmax function is a **generalization of the logistic function** that allows us to perform multi-class classification.

- The softmax function is typically used in the final layer of a neural network-based classifier**. It is a way of forcing the outputs of a neural network to sum to 1, which represents a probability distribution across multiple classes.**

- Let's assume that we want to perform classification for three different classes. Such as classify an image of a person as sad, happy or angry.

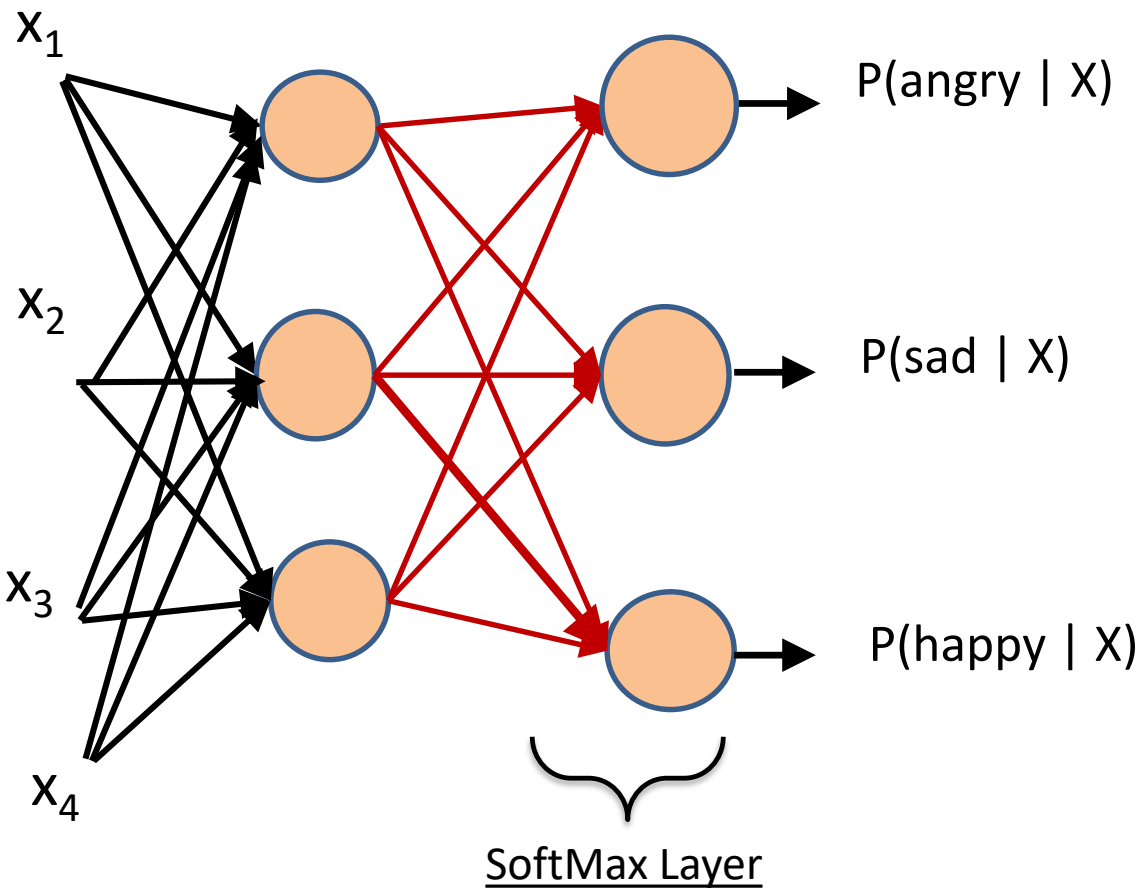
Notice our neural network here has 3 output units (there may be multiple hidden units but in this example we just include one for illustration).
If we feed our model a training example (an image of a person) we want each neuron to output the **probability of this image belonging to a specific class**. We can achieve this using a SoftMax layer.



Let's assume we have pushed a single training example through the first layer. We will now focus on what happens after this stage. As per normal we multiply the weights of the second layer ($W^{[2]}$) by the incoming values ($H^{[1]}$) and add the bias ($b^{[2]}$) .

$$A^{[2]} = W^{[2]} H^{[1]} + b^{[2]}$$

$A^{[2]}$ is the pre-activation value of each neuron in the Softmax layer.
The next step is to apply the activation function.



The activation in a softmax layer can be described in two steps.

1. We calculate a new vector t, we calculate e to the power of the pre-activation outputs.

$$t = e^{A^{[2]}}$$

2. Next we calculate the output of each neuron in the softmax layer as:

$$H^{[2]} = \frac{e^{A^{[2]}}}{\sum t} = \frac{t}{\sum t}$$

Notice that the output of the Softmax is just t normalized by dividing by the sum of t.

The activation in a softmax layer can be described in two steps.

1. We calculate a new vector t, we calculate e to the power of the pre-activation outputs.

$$t = e^{A^{[2]}}$$

2. Next we calculate the output of each neuron in the softmax layer as:

$$H^{[2]} = \frac{e^{A^{[2]}}}{\sum t} = \frac{t}{\sum t}$$

Notice that the output of the Softmax is just t normalized by dividing by the sum of t.

Assume A[2] is the following 3 element vector A[2] = [6, -2, 3].

1. Calculate element-wise exponentiation .

$t = [e^6, e^{-2}, e^3]$

$t = [\ 403.4, 0.135, 20.0\ ]$

The activation in a softmax layer can be described in two steps.

1. We calculate a new vector t, we calculate e to the power of the pre-activation outputs.

$$t = e^{A^{[2]}}$$

2. Next we calculate the output of each neuron in the softmax layer as:

$$H^{[2]} = \frac{e^{A^{[2]}}}{\sum t} = \frac{t}{\sum t}$$

Notice that the output of the Softmax is just t normalized by dividing by the sum of t.

Assume A[2] is the following 3 element vector A[2] = [6, -2, 3].

1. Calculate elment-wise exponentiation .

$t = [e^{6}, e^{-2}, e^{3}]$

t = [403.4, 0.135, 20.0]

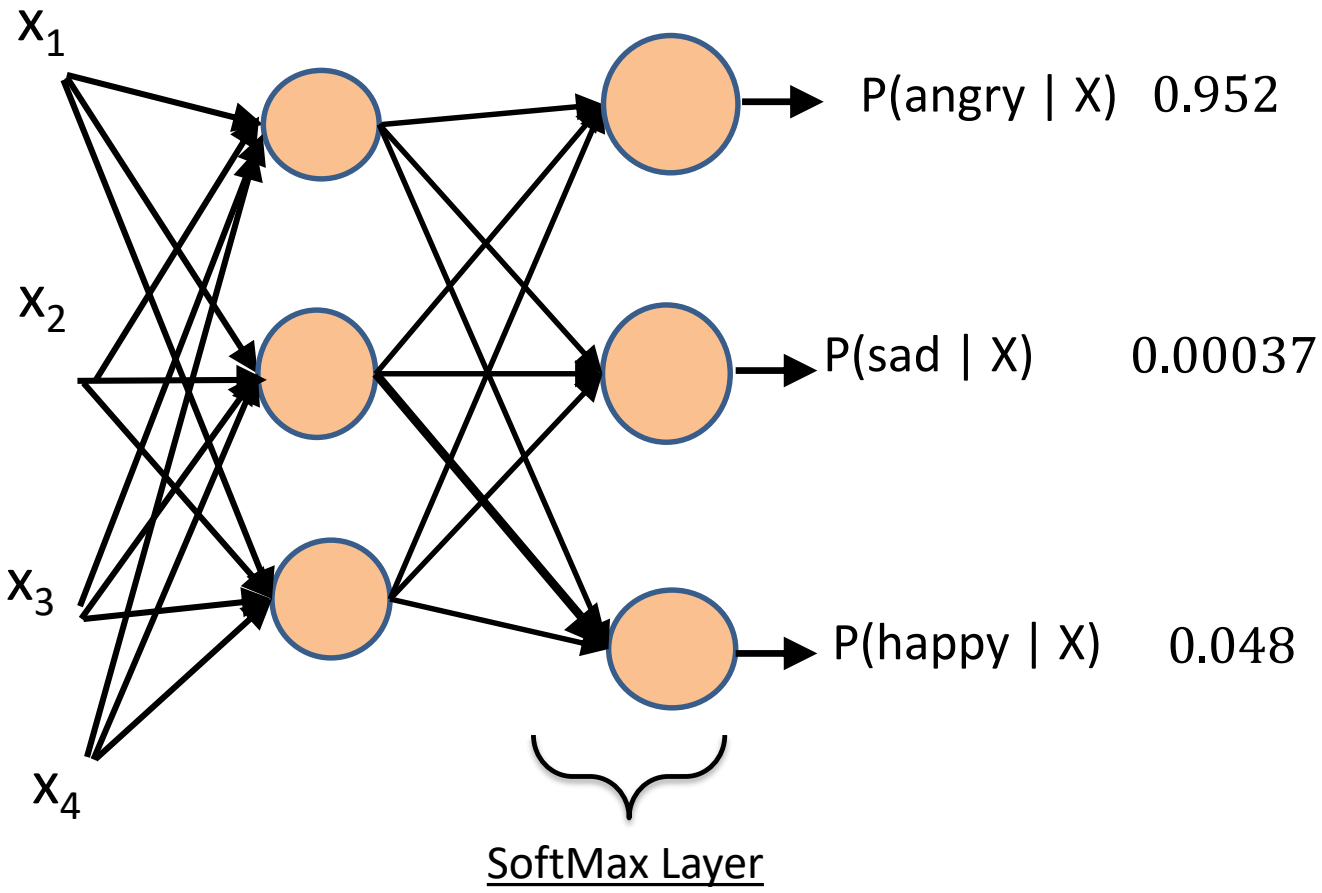
2. We calculate a new vector t, we calculate e to the power of the pre-activation outputs.

$\sum t = 423.5$

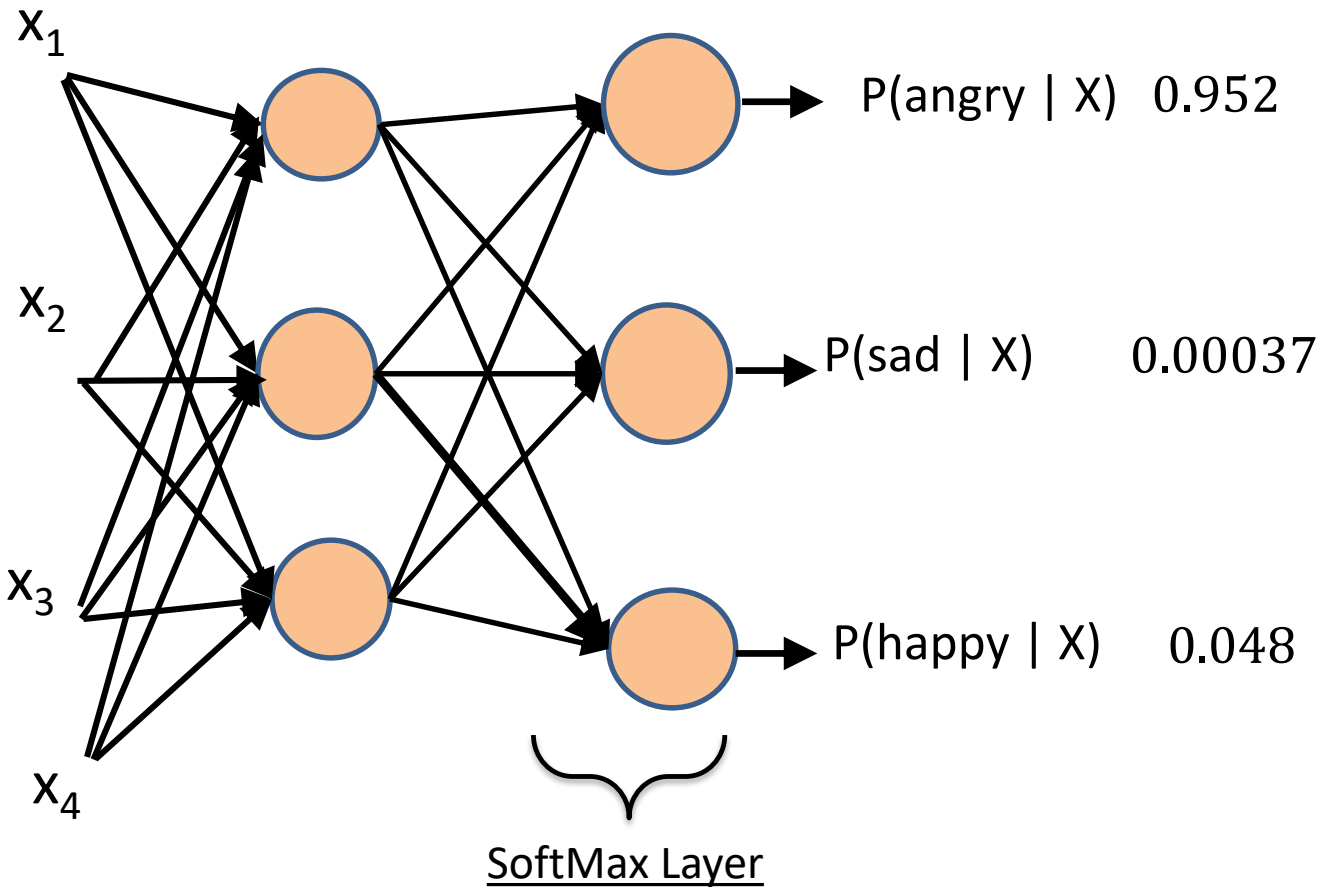
$$H^{[2]} = \frac{[403.4, 0.135, 20.0]}{423.5}$$

$$= [0.952, 0.00037, 0.048]$$

Going back to the visualization our output from the Softmax layer is shown below. Notice that our outputs all sum to 1.
It should be noted that the **true target classification** values are often one hot encoded when dealing with Softmax. For this training instance the true class label could be represented as a the following vector of values **<1, 0, 0>**



Our next step is to use our **loss function** to determine how good or bad this prediction is.



Loss Function for Softmax

- The loss function used for the Softmax function is a variant of the cross entropy loss function.

$$\mathrm{L}(h(x), y) = \quad -\sum_{i=1}^{\mathrm{c}} y_{\mathrm{i}} (\log h(x))$$

- Where **c is the number of classes** (or more specifically the number of units in our Softmax activation layer).
- The input $\mathbf{y}$ above is a vector specifying the correct class for each training examples. For example, if we have three classes then y would be a vector such as **<1, 0 , 0>** (here the first of the three classes is the correct class)
- The j[th] element of y is denoted y_j.

Loss Function for Softmax

- The loss function used for the Softmax function is a variant of the cross entropy loss function.

$$\mathrm{L}(h(x), y) = \quad -\sum_{i=1}^{\mathrm{c}} y_{\mathrm{i}}(\log h(x))$$

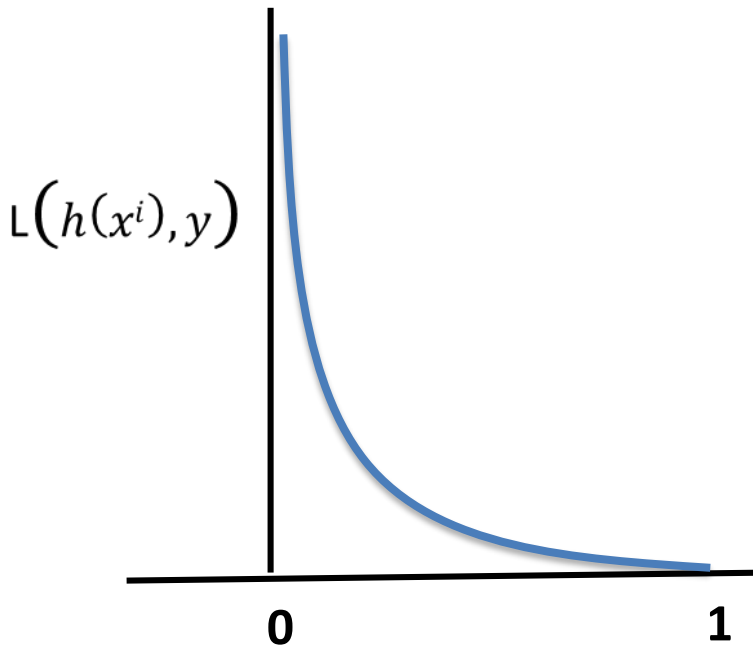
- As we saw in the previous slide the output of the Softmax layer is a vector of probabilities.
- Notice that **only one value in $\mathbf{y}$ can be 1** therefore this is the only y value that matters in the loss function.

Loss Function for Softmax

- The loss function used for the Softmax function is a variant of the cross entropy loss function.

$$\mathrm{L}(h(x), y) = \quad -\sum_{i=1}^{\mathrm{c}} y_{\mathrm{i}}(\log h(x))$$

- Let's work through the example where the outputted probabilities from Softmax are **<0.952, 0.00037, 0.048>** and the true class lab for the training instance is **<1, 0 , 0>**

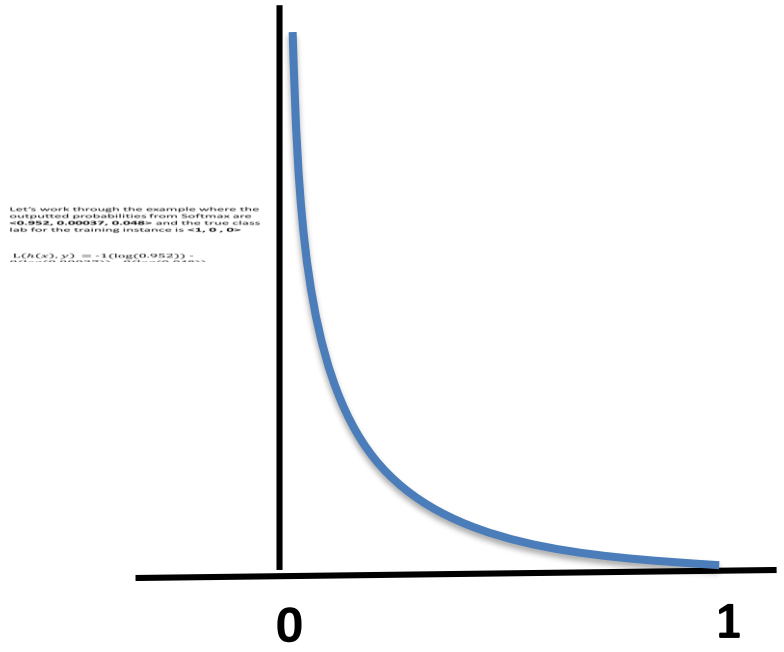


Loss Function for Softmax

- The loss function used for the Softmax function is a variant of the cross entropy loss function.

$$\mathrm{L}(h(x), y) = \quad -\sum_{i=1}^{\mathrm{c}} y_{\mathrm{i}}(\log h(x))$$

- Let's work through the example where the outputted probabilities from Softmax are **<0.952, 0.00037, 0.048>** and the true class lab for the training instance is **<1, 0 , 0>**



- $\mathrm{L}(h(x), y)$ = -1(log(0.952)) - 0(log(0.00037)) - 0(log(0.048))

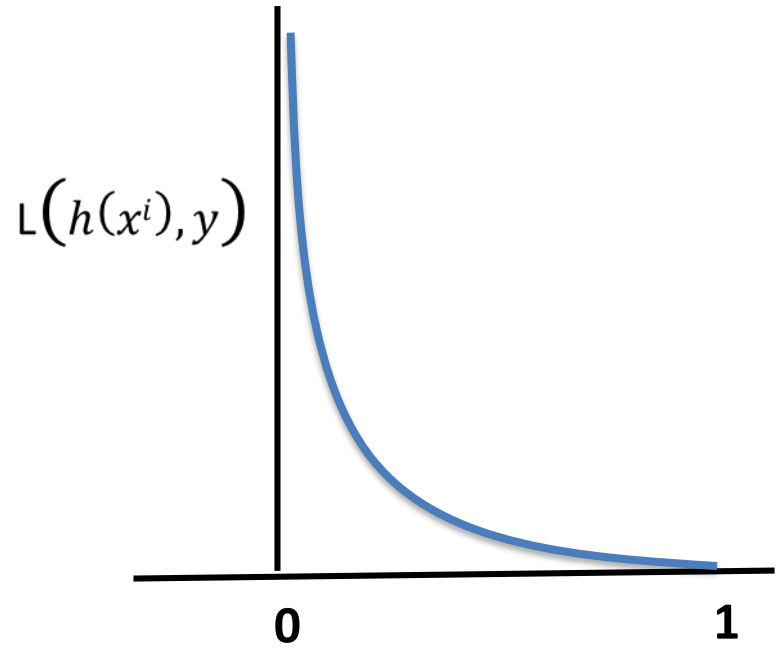
  = -0.021

Loss Function for Softmax

$$\mathrm{L}(h(x), y) = \quad -\sum_{i=1}^{\mathrm{c}} y_{\mathrm{i}} (\log h(x))$$

- Let's work through the example where the outputted probabilities from Softmax are **<0.952, 0.00037, 0.048>** .

- Now consider the case where the correct Y vector was **<0, 0, 1>** then the corresponding loss would be –log(0.048) = 1.31 (a high loss)



Dangers of Overfitting with Neural Networks

- As we previously mentioned neural networks can easily overfit on your training data. They are powerful machine learning models that have in some cases millions of learnable parameters.
- After epoch 8 the model depicted on the right begins to overfit. We can see the validation loss going back up and the training loss continue to fall.

Neural Networks: Algorithmic Techniques

- There are a broad range of algorithm techniques that are very important when building and training deep learning neural networks.
- While we can't cover the full spectrum of techniques we will focus on the following important algorithm techniques.

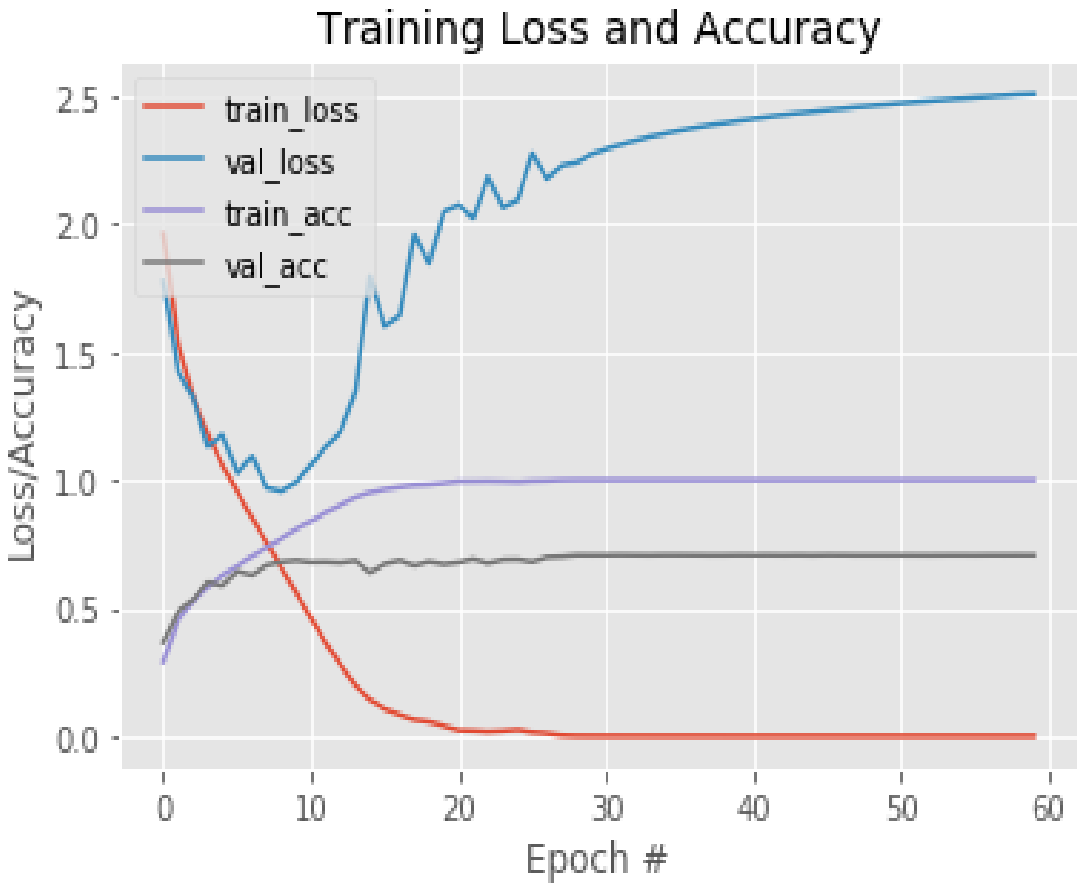
- Approaches to Overfitting
  - L1 and L2 Regularization
  - Dropout Regularization

- Training Networks
  - Mini-batch gradient descent.
  - Learning Rate Decay
  - Adaptive Learning Rates

Dangers of Overfitting with Neural Networks

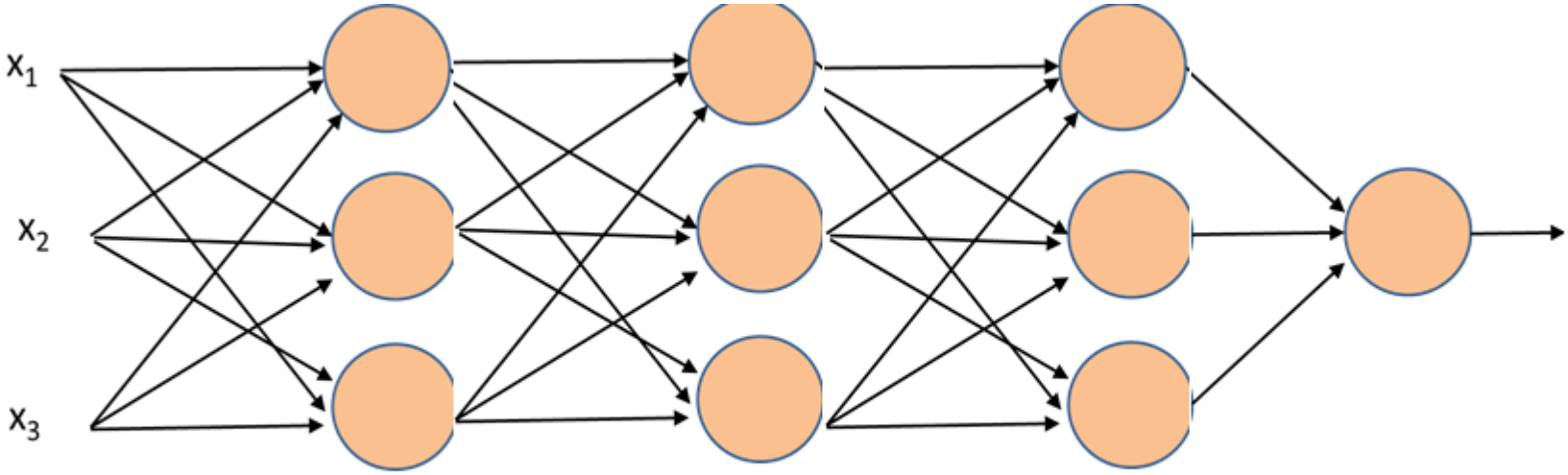
- There are a number of techniques that can be used to mitigate overfitting when building a model.

- Two very commonly used techniques are:
  - Regularization
  - Dropout



Regularization

- In neural networks there is a relationship between the **magnitude of the weights and the complexity of the model**. For example, if we let neural network execute for a large number of iterations you will find that some of the weights become quite large and it starts to overfit on the training data.
- The idea behind regularization is that we try to encourage the model to reduce the magnitude of the weights.
- Over the next few slides we will look at L1 and L2 regularization



Regularization

- To understand regularization we focus on it's application to an MLR.
- Remember in Multivariate Linear Regression we are trying to minimize the following cost function.

$$C(\lambda, b) = \frac{1}{2m}\sum_{i=1}^{m}((h(\boldsymbol{x^i}) - y^i))^2$$

- How might we modify a model to reduce the magnitude of it's weights?
- Well one approach is that we could add an additional component to our loss function related to the weights.
- So gradient descent will now not only try to reduce the difference between the predicated and actual value but also the magnitude of the weights.

L2 Regularization

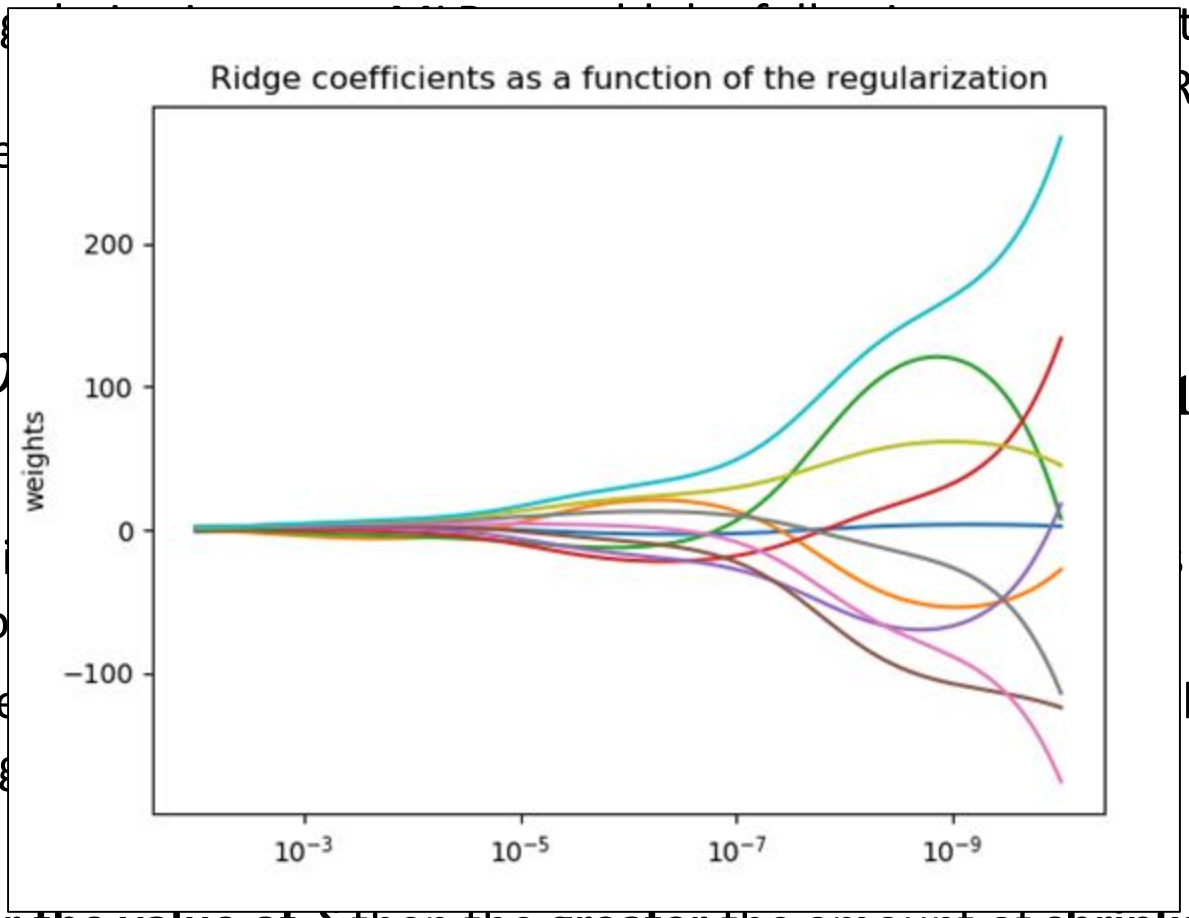
- To add regularization to an MLR we add the following to our cost function (below is referred to as L2 regularization). This variant of an MLR is referred to a Ridge regression.

- $C(\lambda, b) = \frac{1}{2m}\sum_{i=1}^{m}((h(\boldsymbol{x^i}) - y^i))^2 + \delta\sum_{j=1}^{n}{\lambda_j}^2$

- Where δ is a constant called the **regularization rate** and controls the amount of shrinkage in the weights.
- Remember n is the number of features/coefficients, while m is the number of training examples.
- Penalizes the model for having very large weights.
- The **larger the value of $\boldsymbol{\delta}$** then the **greater** the amount of **shrinkage in the weights**.

L2 Regularization

- To add reg… t function (below is … R is referred to a Ridge…
- $C(\lambda, b$… $\lambda_j{}^2$
- Where δ i… the amount o…
- Remembe… he number of training…
- Penalizes…
- The **larger the value of δ** then the **greater** the amount of **shrinkage in the weights**.



https://scikit-learn.org/stable/auto_examples/linear_model/plot_ridge_path.html

L1 Regularization

- Another commonly used variant is referred to as L1 regularization. This is often referred to as Lasso Regression when used with an MLR.

$$C(\lambda, b) = \frac{1}{2m}\sum_{i=1}^{m}((h(\boldsymbol{x^i}) - y^i))^2 + \delta\sum_{j=1}^{n}|\lambda_j|$$

- L1 just **sums each weight** as opposed to the squared value of each weight.
- L1 in the form of an MLR (**Lasso**) is often used a mechanism of feature selection as it reduce the number of features on which the given solution is dependent.

Regularization

- The more we **increase our regularization parameter** the more aggressively our algorithm will attempt reduce the weights.
- This can have the effect of almost negating the impact of some of the hidden units (or dramatically reducing their impact).
- As such you end up with a **simpler model** and simpler models are less prone to overfitting.
- However, if set your regularization parameter to be low then the weights will not decrease as aggressively, which can cause higher variance as a result.