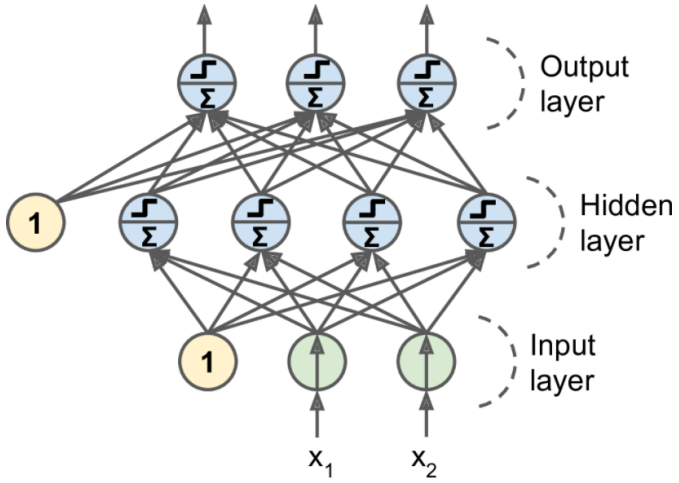
Neural Networks – the MLP Architecture (Chap 10 in [HOML])

- Multilayer Perceptron (MLP)
  - Consists of one input layer and one or more layers of linear threshold units (LTUs), also called hidden layers (plus the output layer)
  - Every layer (except the output layer) includes a bias neuron and is fully connected to the next layer
  - When an ANN has two or more hidden layers then it is called a deep neural network (DNN) => Deep Learning



MLP hyper parameters

- Number of hidden layers (MLP with just one hidden layer can theoretically model even the most complex functions, provided it has enough neurons)
- Number of neurons per hidden layer
- Learning rate
- Optimizer
- Batch size
- Activation functions

We considered last time why the learning rate (the number multiplying the gradients in the GD method) is an important parameter.

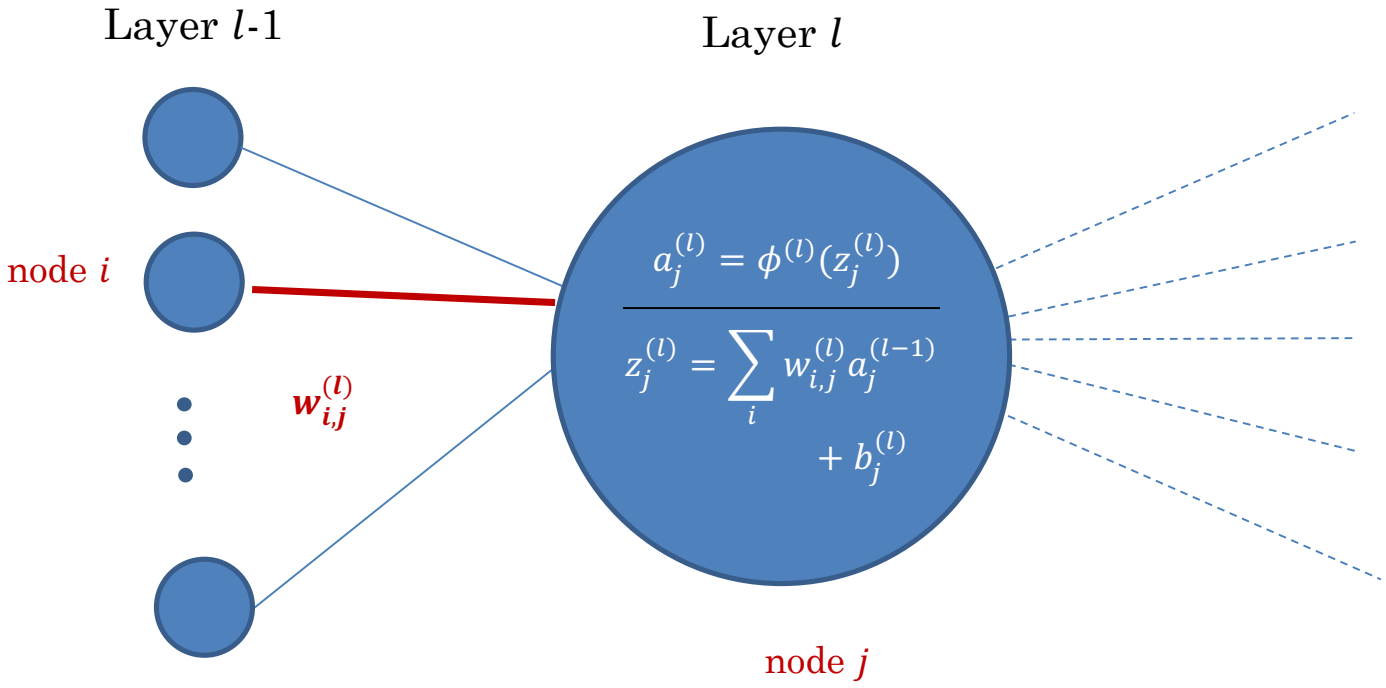
To understand the role of more of the hyper-parameters listed above, we will dig a bit deeper in the Back Propagation algorithm used to find the parameters in a NN.

Back Propagation Illustration

We want to minimize the cost (loss) function J, so we need to compute the partial derivatives of J for all the parameters. The extra index l below (compared to Chap. 10 in HOML) represents the layer and the bs are the biases:

$$\frac{\partial J}{\partial w_{i,j}^{(l)}} \quad \text{and} \quad \frac{\partial J}{\partial b_j^{(l)}} \qquad (*)$$

We'll illustrate in the following toy network for nodes i & j (ϕ is the activation function):



We start with the error at the output layer $J(\boldsymbol{y}-\widehat{\boldsymbol{y}})$ and moving right to left in the above network want to compute $(*)$. By the chain rule, we compute the error gradients (δs)

$$\delta_i^{(l-1)} \overset{\text{def}}{=} \frac{\partial J}{\partial z_i^{(l-1)}} = \sum_j \frac{\partial J}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial z_i^{(l-1)}}$$

We have

$$z_j^{(l)} = \sum_i w_{i,j}^{(l)} a_j^{(l-1)} + b_j^{(l)} = \sum_i w_{i,j}^{(l)} \phi^{(l-1)}(z_i^{(l-1)}) + b_j^{(l)}$$

$$\Longrightarrow \delta_i^{(l-1)} = \sum_j \frac{\partial J}{\partial z_j^{(l)}} \left(\frac{d\phi^{(l-1)}}{dz} |_{z_i^{(l-1)}} \right) w_{i,j}^{(l)} = \frac{d\phi^{(l-1)}}{dz} |_{z_i^{(l-1)}} \sum_j \delta_j^{(l)} w_{i,j}^{(l)}$$

$$\Longrightarrow \frac{\partial J}{\partial w_{i,j}^{(l)}} = \frac{\partial J}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial w_{i,j}^{(l)}} = \delta_j^{(l)} a_i^{(l-1)}$$

This shows how to find the δs in a layer if we know the δs in all layers to the right. And from there, how to find the derivatives of the loss J with respect to the weights. Also

$$\frac{\partial J}{\partial b_j^{(l)}} = \delta_j^{(l)}$$

The last hidden layer is different because it feeds into the output. If the loss function is quadratic (a regression problem) then we have for 1 data point and K output nodes

$$J = \frac{1}{2}\sum_{j=1}^{K=1}\left(\hat{y}_j - y_j\right)^2, \hat{y}_j = \phi^{(L)}\left(z_j^{(L)}\right) \implies \delta_j^{(L)} = \frac{d\phi^{(L)}}{dz}|_{z_j^{(L)}}\left(\hat{y}_j - y_j\right), j = 1, \ldots, K$$

Backpropagation Algorithm

Step 0: Initialize weights and biases

Choose the weights and biases, typically at random. The size of the weights should decrease as the number of nodes increases.

Step 1: Pick one of the data points at random

Input the vector x into the left side of the network, and calculate all the z, a, etc. And finally calculate the output y. (This might also be a vector.)

Step 2: Calculate contribution to the loss function

You don't actually need to know the actual value of the loss function to find the weights and biases but you will want to calculate it so you can monitor convergence.

Step 3: Starting at the right, calculate all the δs

For example, if using the quadratic loss function in one dimension (then drop j):

$$\delta^{(L)} = \frac{d\phi^{(L)}}{dz}|_{z^{(L)}}(\hat{y} - y)$$

Moving to the left (sum over all nodes j which are connected & to the right of node i)

$$\delta_i^{(l-1)} = \frac{d\phi^{(l-1)}}{dz}|_{z_i^{(l-1)}} \sum_j \delta_j^{(l)} w_{i,j}^{(l)}$$

Step 4: Update the weights & biases using (stochastic) GD

$$(new)w_{i,j}^{(l)} = (old)w_{i,j}^{(l)} - \eta \frac{\partial J}{\partial w_{i,j}^{(l)}} = (old)w_{i,j}^{(l)} - \eta \delta_j^{(l)} a_i^{(l-1)}$$

$$(new)b_j^{(l)} = (old)b_j^{(l)} - \eta \frac{\partial J}{\partial b_j^{(l)}} = (old)b_j^{(l)} - \eta \delta_j^{(l)}$$

Return to Step 1.

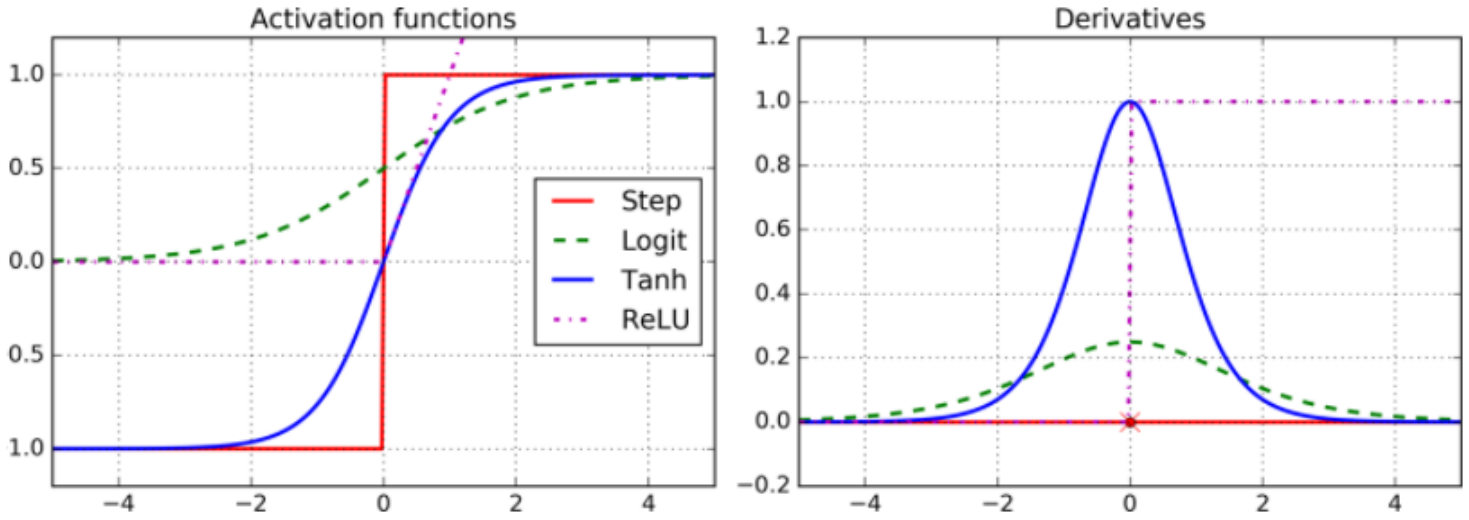
- The δs used in Step 4 to update the parameters are using the derivative of the activation function (see Step 3). The step function contains only flat segments, so there is no gradient to work with (GD cannot move on a flat surface)

- Thus the step function was replaced in "backprop" by the logistic (sigmoid) function $\sigma(z) = 1/(1 + \exp(-z))$.

- But if logistic activation derivate is close to 0 then the updates of the weights/biases will seize or happen at a very slow rate.
  - This for example is the case for large values of the argument.
  - Similar situation is present when the derivative of the activation function is close to $\pm\infty$.

- The Step 3 & 4 above also show the simple way the derivative of the activation function is used in the backward pass of the algorithm which opens the possibility to use autodiff (auto-differentiation).

- There are different choices of activation functions (i.e. the activation is a hyperparameter)
  - ReLU – most used (not differentiable at zero, but fast to compute)
  - Hyperbolic tangent function (tanh)
  - Logit function
  - Sigmoid (not as good as ReLU, but biologically more accurate)



Types of Problems to Solve with NN

Regression MLPs

- To predict a single value (e.g., the price of a house, given many of its features) you just need 1 output neuron: its output is the predicted value.
- For multivariate regression (to predict multiple values at once) - 1 output neuron per output dimension. E.g., to locate the center of an object in an image, you need to predict 2D coordinates, so you need two output neurons.
- In the regression case, usually you do not need any activation function for the output neuron unless there are restrictions on the output value
- The loss function is typically Mean Square Error (MSE), Mean Absolute Error (MAE) or a combination of the two.

Table 10-1. Typical regression MLP architecture

Hyperparameter	Typical value
# input neurons	One per input feature (e.g., 28 x 28 = 784 for MNIST)
# hidden layers	Depends on the problem, but typically 1 to 5
# neurons per hidden layer	Depends on the problem, but typically 10 to 100
# output neurons	1 per prediction dimension
Hidden activation	ReLU (or SELU, see Chapter 11)
Output activation	None, or ReLU/softplus (if positive outputs) or logistic/tanh (if bounded outputs)
Loss function	MSE or MAE/Huber (if outliers)

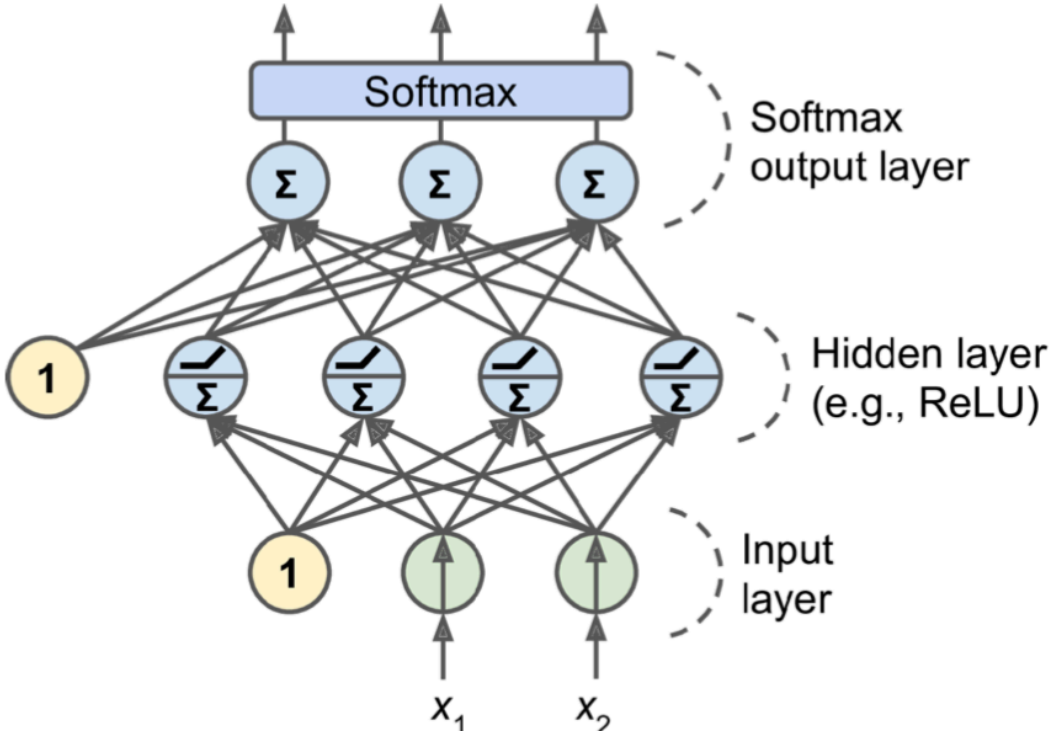
An example of a regression MLP – in the Python notebook

Classification MLPs

- Binary classification - 1 output neuron using the logistic activation function: the output will be a number between 0 and 1, which you can interpret as the estimated probability of the positive class.

- Multilabel binary classification – e.g. email classification system that predicts whether each incoming email is legit or spam, and simultaneously predicts whether it is an urgent or nonurgent email. In this case, you would need 2 output neurons, both using the logistic activation function: the first would output the probability that the email is spam, and the second would output the probability that it is urgent.

- Multiclass classification - each instance can belong only to a single class, out of >=3 possible classes (e.g., classes 0-9 for digit image classification). Then 1 output neuron per class, and use the softmax activation function



The softmax function

$$\hat{p}_k = \sigma(\mathbf{s}(\mathbf{x}))_k = \frac{\exp\left(s_k(\mathbf{x})\right)}{\sum_{j=1}^{K} \exp\left(s_j(\mathbf{x})\right)}$$

will ensure that all the estimated probabilities are between 0 and 1 and that they add up to 1 (which is required if the classes are exclusive).