Neural Networks Lecture 1

Data Science Toolbox Week 7

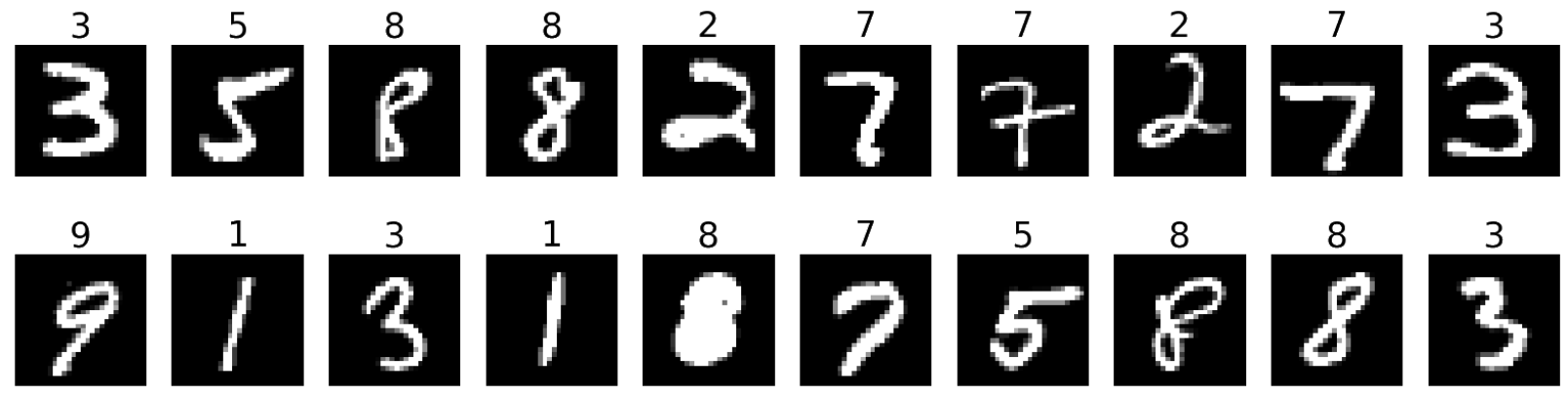
Slides by Dr Huw Day, University of Bristol

Learning Objectives

- Learn what a perceptron is
- Learn what a multilayer perceptron is
- Learn how to measure loss for a neural network
- Learn how gradient descent and back propagation work

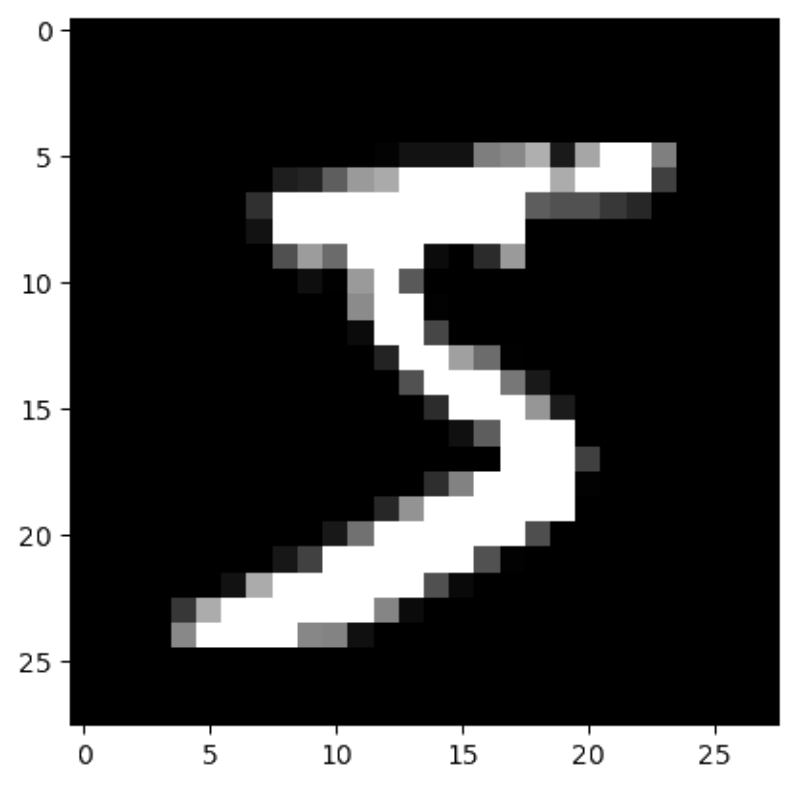
The Perceptron

1. How do you represent greyscale images of 28x28 pixel size as vectors?
2. How would you constructive a supervised machine learning module to do binary classification on pictures of two different numbers?
3. How would you generalize this beyond binary classification? Or to regression?



Sample images from MNIST (*Modified National Institute of Standards and Technology database)* dataset with their associated label

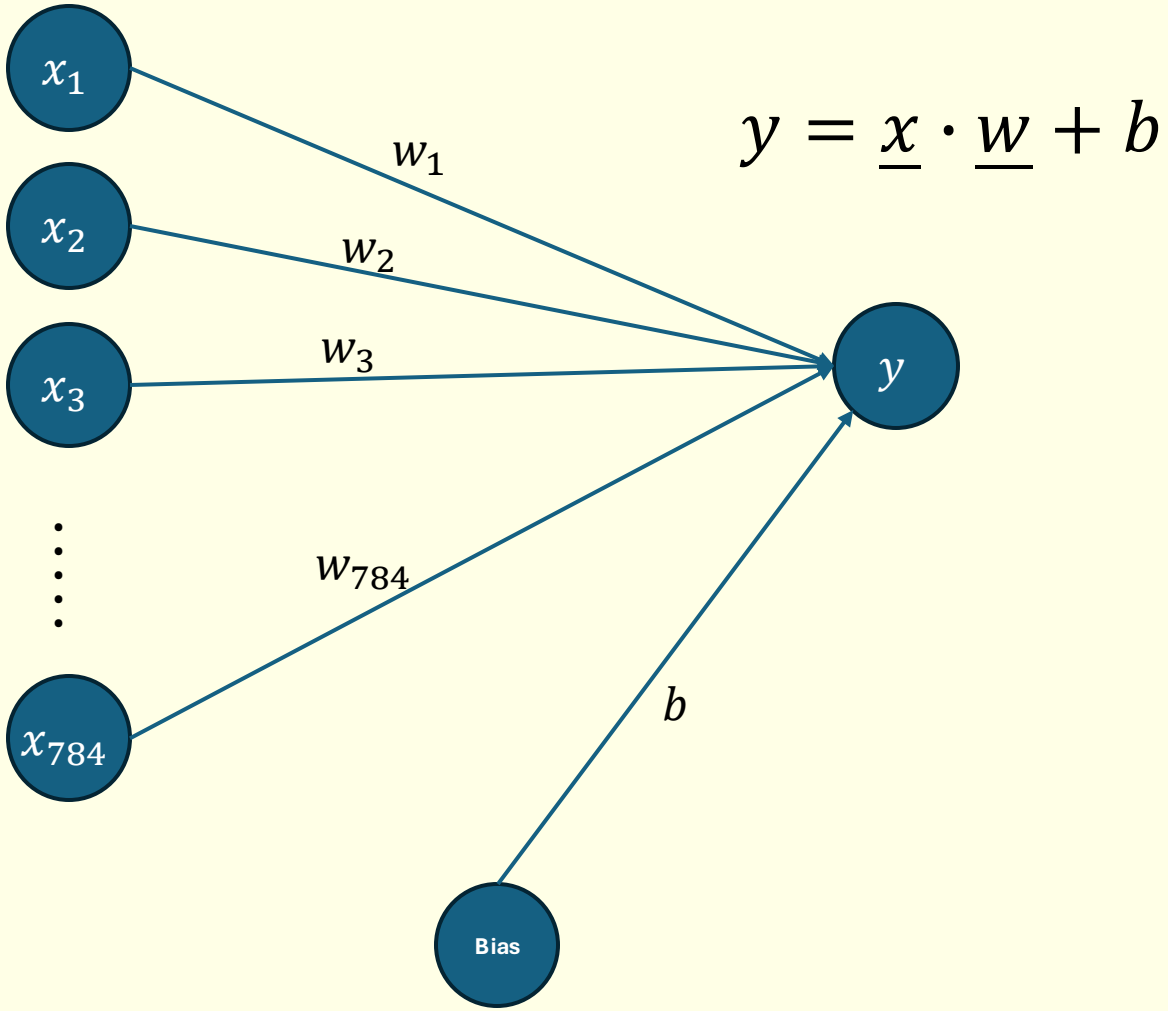
1. Representing images as vectors



“Flatten”

784 (28 squared) dimensional vector of grey scale values between 0 and 1 (0 = black, 1 = white)

2. Binary classification of images with a perceptron

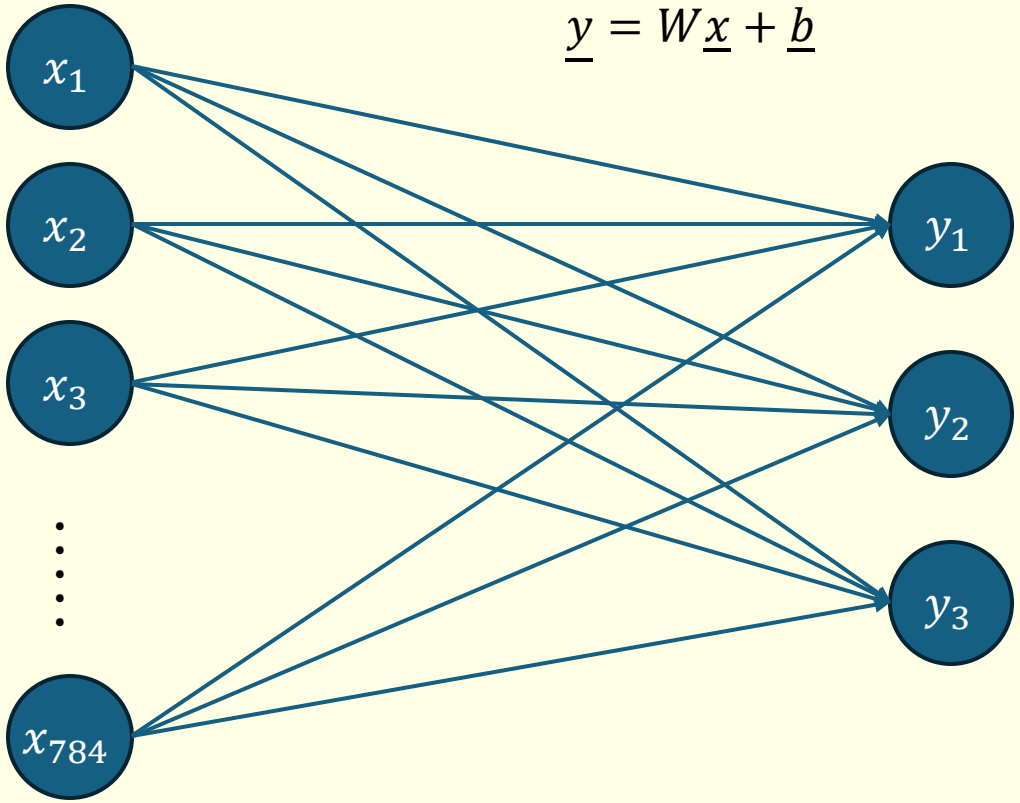


$$y = \underline{x} \cdot \underline{w} + b$$

- Each pixel/datapoint forms a layer of neurons which are connected by weights (w1, w2, …) to the output neuron $\underline{y}$.
- There is also a bias "b" to make an affine map.
- Given two classes (assume we have pictures of 0s and picture of 1s) we assign a y value of either 0 or 1.
- We seek to tweak the values of each of the weights and the bias so that our perceptron gets the right answer as many times as possible.
- You could feasibly round values of "y" so if your network outputs 0.9 then you expect the guess is 1.

3. Beyond binary classification

- Regression is about predicting a continuous value as opposed to a discrete class, so having a single output neuron can still accommodate this.
- For multiclass classification typically we have an output layer with one neuron per class.
- How might you scale these outputs?



Argmax and Softmax

These are two ways to turn a K-tuple into a probability distribution over K possible outcomes:

Argmax: sets the maximum of the K-tuple to 1 and collapses all others to 0

$$ArgMax(y_i) = \begin{cases} 1, if\ y_i = \max(\underline{y}) \\ 0, otherwise. \end{cases}$$

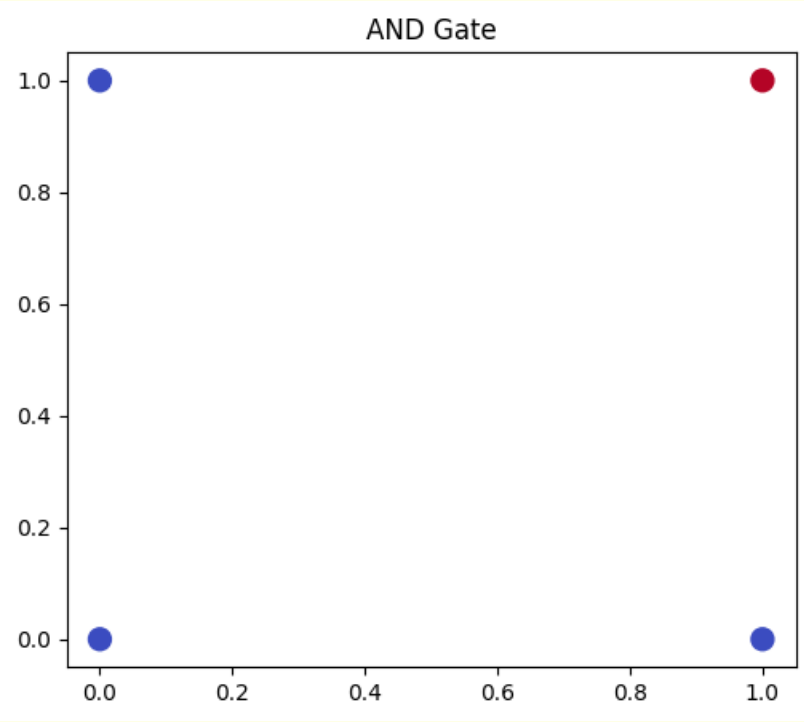
- Super easy to interpret, especially across many outputs.
- Doesn't give you a way to rank outputs.
- It cannot be used inside a training loop (more on training later!)

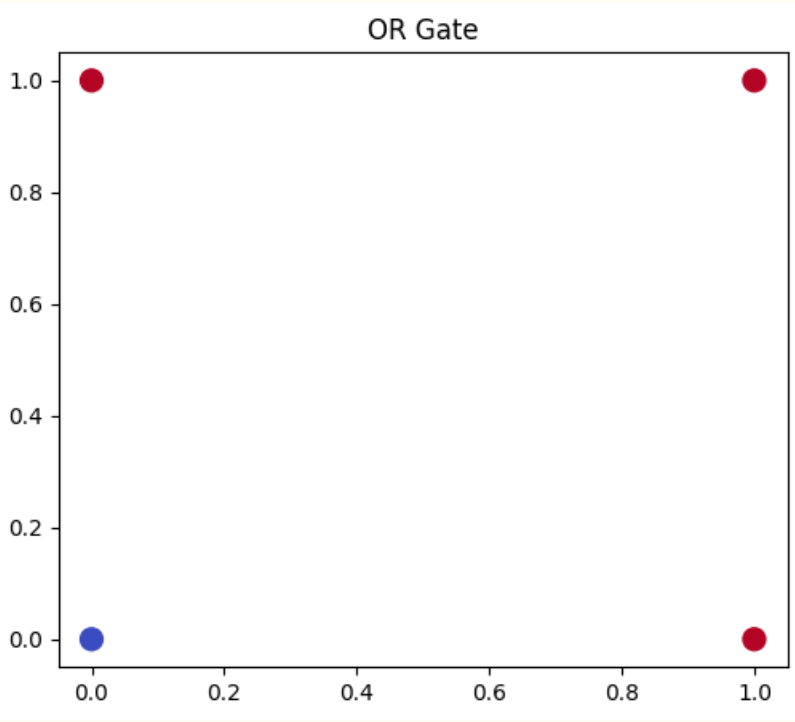
Softmax: generalizes the logistic function to multiple dimensions

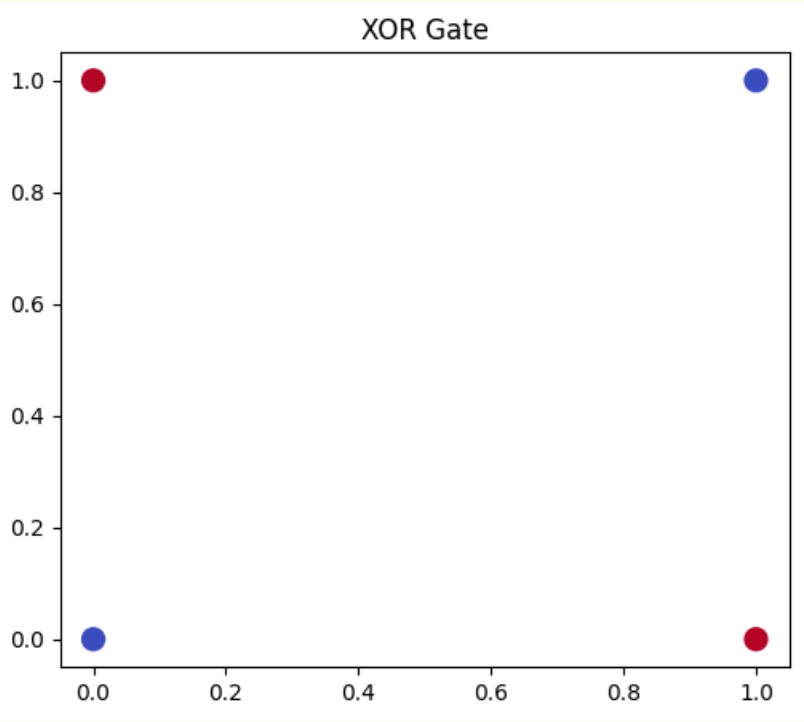
$$SoftMax(y_i) = \frac{e^{y_i}}{\sum_{j=1}^{K} e^{y_j}}$$

- As long as the outputs are mutually exclusive, the softmax values sum to 1.
- Preserves the order of the original values.
- Can be used inside training loop.
- *Note: whilst softmax look like probabilities they should not in general be treated as such!*

Which Boolean functions are linearly separable?





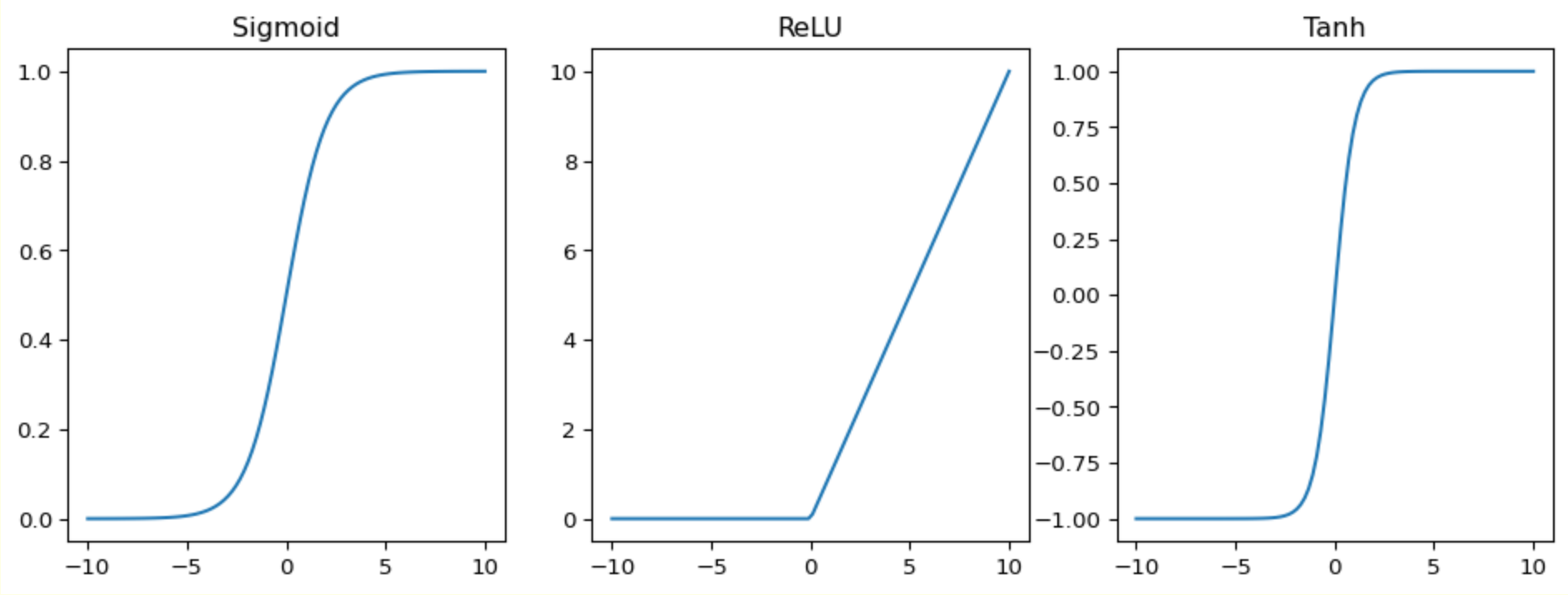


McCulloch and Pitts (1943) A logical calculus of the ideas immanent in nervous activity

Activation functions

- We want our perceptron to be able to tackle problems that are not linearly separable.
- To do this we use activation functions, taking the output from each layer and putting it through an activation function.
- We want our activation functions to be:
  - Non-linear
  - Smooth
  - Easy to compute gradients (this will become apparent when we think about how to get our networks to learn automatically later!)

Some activation functions:



$$\sigma(x) \coloneqq \frac{1}{1+e^{-x}}$$

- Sigmoid/logistic function
- Squishes the real numbers into (0, 1)
- Used in early NNs, less common now!

$$ReLu(x) \coloneqq \max(0, x)$$

- Rectified Linear Unit
- Arguably most common (family of) activation function
- Maps negative values to 0

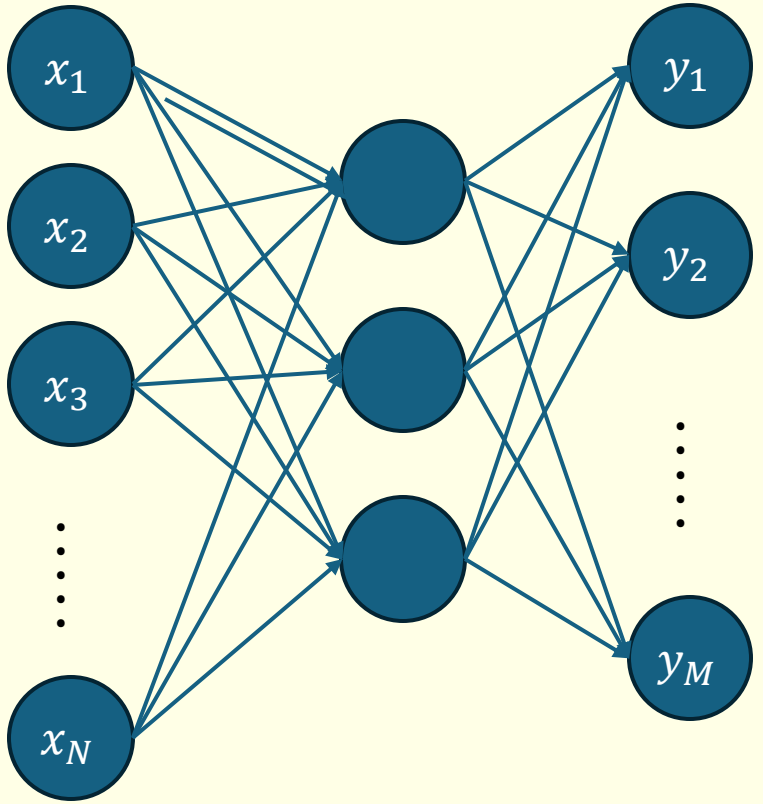
$$\tanh(x) \coloneqq \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$$

- Hyperbolic tangent
- Squishes all inputs into the range (-1, 1).
- Most commonly used in recurrent neural networks (RNNS) as it keeps hidden states bounded and is centred around zero for stable updates

Mathematical description of a Single Layer Perceptron

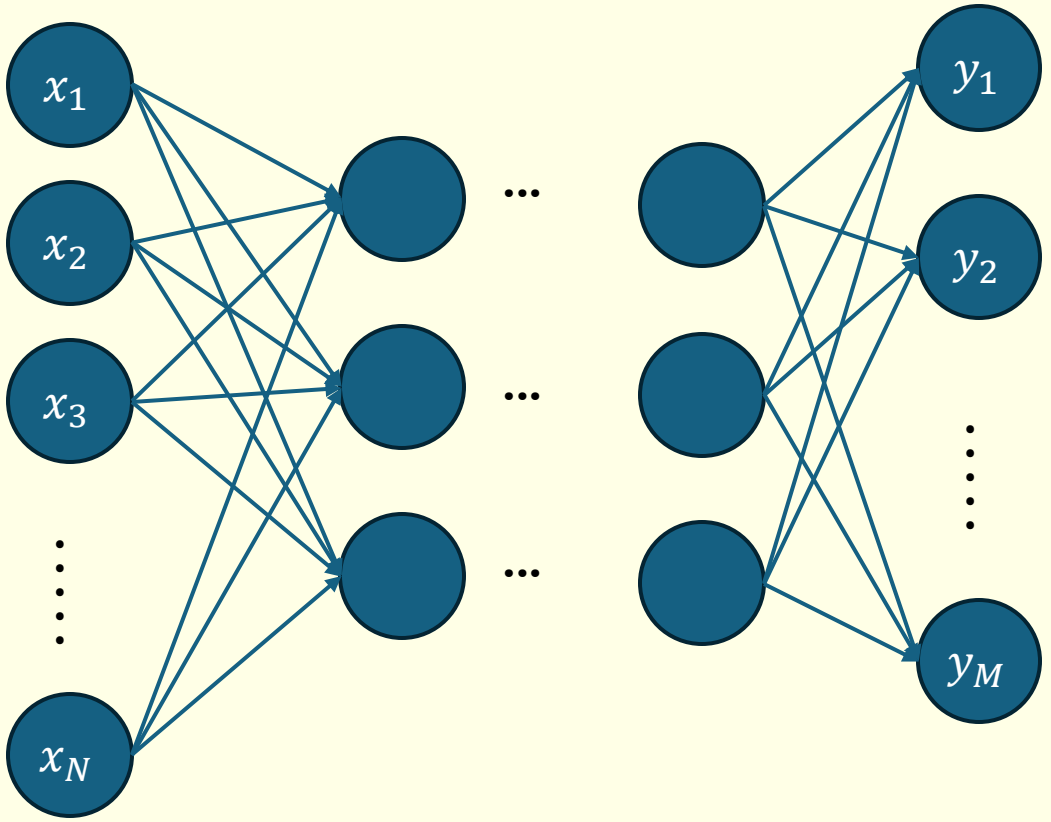
A single layer perceptron $f: \mathbb{R}^N \to \mathbb{R}^M$ with weights W_{ij} and activation function $\sigma: \mathbb{R} \to \mathbb{R}$ acts in the following for $\underline{x} \in \mathbb{R}^N$:

$$f(\underline{x}) = \sigma\left(W_{0j} + \sum_{i=1}^{N} W_{ij} x_i\right)$$



Multilayer Perceptrons (MLP)

- We can add hidden layers between our input and output layers of neurons to better capture more complex patterns.
- The main limit for these approaches is compute power!



"Learning representations by back-propagating errors" (https://www.nature.com/articles/323533a0) by David Rumelhart, Geoffrey Hinton and Ronald J Williams

Universal Approximation Theorem(s)

- Universal Approximation Theorems (UATs) state that neural networks with a certain structure can theoretically approximate any continuous function to any desired degree of accuracy.
- You can think of them as like the analogy of existence results about Taylor series expansions of certain kinds of functions.
- Most UATs concern either arbitrary width (number of neurons in a hidden layer) or arbitrary depth (number of hidden layers).
- In theory you can approximate any function with an MLP with a single hidden layer of arbitrarily large width…but you shouldn't. Why do you think not?

https://youtu.be/qx7hirqgfuU?si=Yv3M3wRvxmTBjKO0

A Survey on Universal Approximation Theorems

Training a neural network

- So far we've learnt that for a given continuous function, a neural network exists with the correct weights to approximate that continuous function arbitrarily well.
- But how do we actually find it the correct weights?
- How can we get a neural network to learn?
  1. Get a loss function to describe how wrong your guesses are
  2. Use gradient descent to tell you which direction you should change your weights
  3. Use back propagation to update your model weights

Loss/cost functions – how to measure how much you've gone wrong

- The main loss functions you're likely to encounter are MSE (Mean Square Error), MAE (Mean Absolute Error) and cross entropy (which is more information theoretic).
- Researchers sometimes construct specialized/bespoke loss functions for specific applications, e.g. contrastive learning.
- For the purposes of these lectures we'll focus on MSE:

$$\mathrm{L}\left(\underline{y}, \underline{\hat{y}}\right) := \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

https://docs.pytorch.org/docs/stable/nn.html#loss-functions

Gradient descent

- Differentiate our loss function with respect to our model weights:

$$\nabla L_W = \begin{pmatrix} \partial_{W_{11}} L \\ \vdots \\ \partial_{W_{nM}} L \end{pmatrix}$$

- Gradient tells us which way is downhill on our loss landscape. We take a small step in that direction:

$$W_{ij} \leftarrow W_{ij} - LR \frac{\partial L}{\partial W_{ij}}$$

- Where LR = Learning Rate, positive real hyperparameter which affects network convergence.

Learning = minimizing a loss function with respect to your network parameters

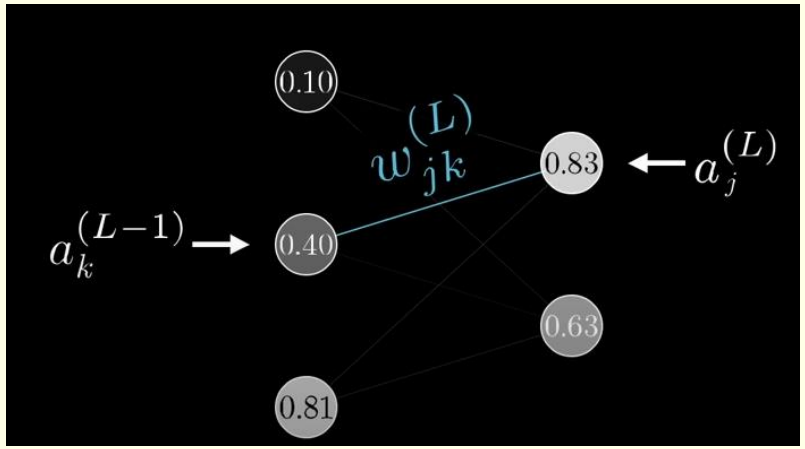
Learning representations by-propagating errors

Backpropagation – how to calculate the gradient?

We consider the contribution of a single training example to the loss function:

Loss function with respect to expected labels and activations of the last (L-th) hidden layer of neurons

$$L_0 = \sum_{j=0}^{n_L-1} \left(a_j^{(L)} - y_j\right)^2$$



$$a_j^{(L)} = \sigma(z_j^{(L)})$$

$$z_j^{(L)} = b_j^{(L)} + \sum_{k=0}^{n_{L-1}-1} w_{jk}^{(L)} a_k^{(L-1)}$$

Chain Rule for a weight $w_{jk}^{(L)}$:

$$\frac{\partial L_0}{\partial w_{jk}^{(L)}} = \sum_{j=0}^{n_L-1} \frac{\partial z_j^{(L)}}{\partial a_k^{(L-1)}} \frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} \frac{\partial L_0}{\partial a_j^{(L)}}$$

Chain Rule for activation $a_k^{(L-1)}$:

$$\frac{\partial L_0}{\partial a_k^{(L-1)}} = \sum_{j=0}^{n_L-1} \frac{\partial z_j^{(L)}}{\partial a_k^{(L-1)}} \frac{\partial a_k^{(L)}}{\partial z_j^{(L)}} \frac{\partial L_0}{\partial a_j^{(L)}}$$

Exercise: calculate the partial derivatives for the weights (consider a ReLU activation function).

Can you do the same for the corresponding bias chain rule for $b_j^{(L)}$?

What might go wrong with this training paradigm?

1. How do we deal with such a large amount of training data (rather than looking at examples one at a time)?
2. Under what circumstances would gradient descent achieve a global minimum? Where might it get stuck at a local minimum?
3. How would you make it harder for a model with lots of parameters to overfit your data?

Adam: A Method for Stochastic Optimization

SGD, Momentum and Adam:

- Stochastic Gradient Descent (SGD): calculates gradient on a few training examples per update which helps explore loss surfaces and is more memory efficient
- We can also add Momentum (an exponentially weighted average of past gradients, i.e. the first moment) which smooths updates so that you don't bounce around the loss surface as much.
- We can also add Adapative Learning Rates (an exponentially weighted average of squared gradients, i.e. the second moment) which scales each parameter's step size based on how volatile its gradients are.
- Adam is the name of the optimizer that combines SGD, momentum and adaptive learning rates.

$$g_t = \nabla_\theta L_t \qquad \text{(current gradient)}$$

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t \qquad \text{(1st moment – mean)}$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \qquad \text{(2nd moment – variance)}$$

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t} \qquad \text{(bias correction)}$$

$$\hat{v}_t = \frac{v_t}{1-\beta_2^t} \qquad \text{(bias correction)}$$

$$\theta_t = \theta_{t-1} - \alpha \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} \qquad \text{(gradient step)}$$

$\alpha = Learning\ rate\ (default\ 0.001)$
$\beta_1 = Moment\ decay\ (default\ 0.9)$
$\beta_2 = RMS\ decay\ (default\ 0.999)$
$\varepsilon = Numerical\ stability\ (default\ 10^{-8})$

Further resources

- If you don’t read AI papers you’ll never learn how to read AI papers, so go and read some of the ones from the slides but also…
- Go and watch the first four videos of 3Blue1Brown’s videos on neural networks (especially the videos on backpropagation) : https://youtube.com/playlist?list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi&si=gY0nycdjWu2kf80g
- Bonus, watch Welch labs video series on How models learn: https://youtu.be/NrO20Jb-hy0?si=SCTNtHt2DR4ZvVCu
- The StatQuest Illustrated Guide to Neural Networks and AI (also the StatsQuest YouTube channel: https://statquest.org/