Week 10: Neural Networks

POP77032 Quantitative Text Analysis for Social Scientists

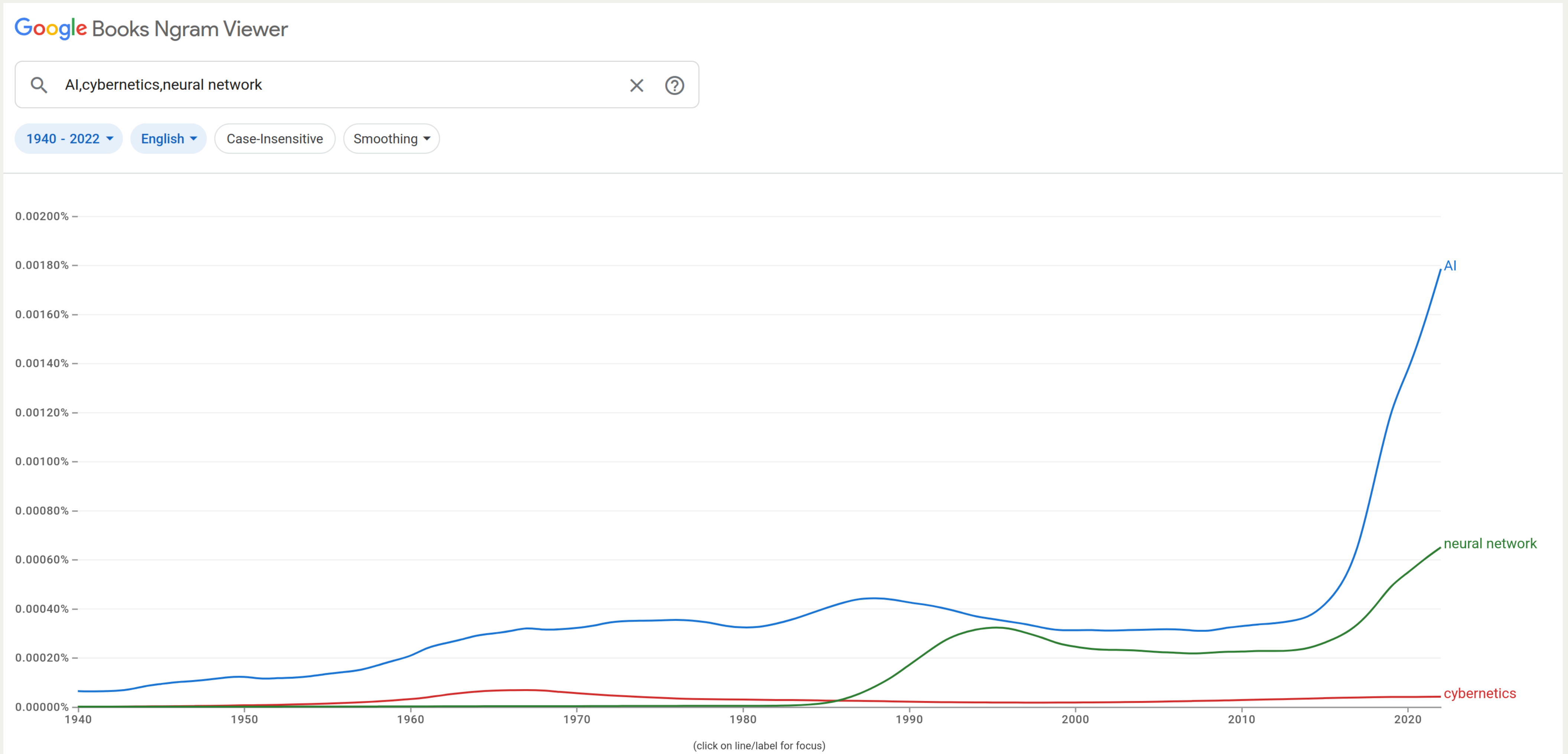
Tom Paskhalis

Overview

- Single-layer perceptron
- Linear separability
- Gradient descent
- Stochastic gradient descent

Introduction to Neural Networks

Some History



(Google Books Ngram Viewer)

Laying the Foundation

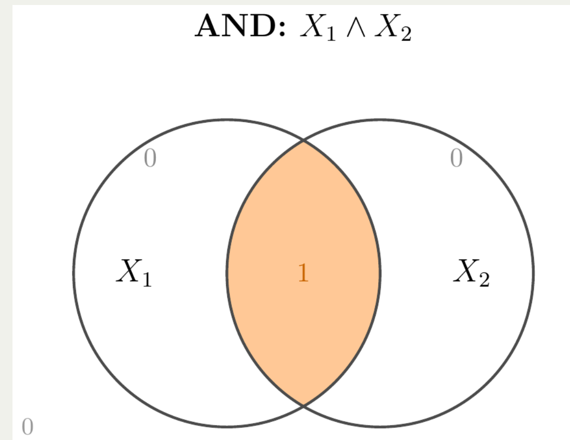
- In 1943 McCulloch and Pitts proposed a mathematical model of the nervous system as an inteconnected network of simple logical units (neurons).
- In 1958 Rosenblatt developed the first neuron-based supervised image recognition system called Mark I Perceptron, which could learn to classify simple images of letters and shapes.
- In 1969 Minsky and Papert in their book "Perceptrons" highlighted the limitations of single-layer perceptrons, particularly, their inability to learn non-linearly separable functions.
- This led to a period, called the "AI winter" with research on neural networks and AI broadly being largely abandoned until the re-surgence of interest in late 1990s and early 2000s with the advent of deep learning.
- While nervous system analogies inspired a lot of the original work on neural networks, contemporary approaches are rather removed due to poor understanding of brain function.

Why Neural Networks?

- Highly flexible and generisable architecture (can be adapted for any kind of output).
- Can learn complex, non-linear relationships between inputs and outputs.
- Generates useful representations of data in the process (e.g. word embeddings).
- Able to learn from very large datasets through efficient optimisation algorithms (e.g. stochastic gradient descent).
- Often outperform competing methods across a wide range of tasks (e.g. image recognition, language modelling, etc.).

Single-layer Perceptron

Example: Boolean AND



- For 2 binary variables, the **Boolean AND** is 1 only if both variables are 1, and 0 otherwise.
- We can think of it as a function f that takes two binary inputs (X_1, X_2) and produces a binary output Y:

$$Y = f(X_1, X_2) = \begin{cases} 1 & \text{if } X_1 = 1 \text{ and } X_2 = 1 \\ 0 & \text{otherwise} \end{cases}$$

Example: Learning AND

- Imagine we want to learn this function from the data.

X_1	X_2	$Y = X_1$ AND X_2
0	0	0
0	1	0
1	0	0
1	1	1

```r
and_data <- data.frame(
  X1 = c(0, 0, 1, 1),
  X2 = c(0, 1, 0, 1),
  Y = c(0, 0, 0, 1)
)
```

Example: Learning AND

- We can easily learn this function using a linear model:

```
and_lm_fit <- lm(Y ~ X1 + X2, data = and_data)
and_pred <- predict(and_lm_fit, newdata = and_data)
and_pred
```

```
    1     2     3     4 
-0.25  0.25  0.25  0.75 
```

- While the predicted values are not binary, we could easily add a threshold to address this:

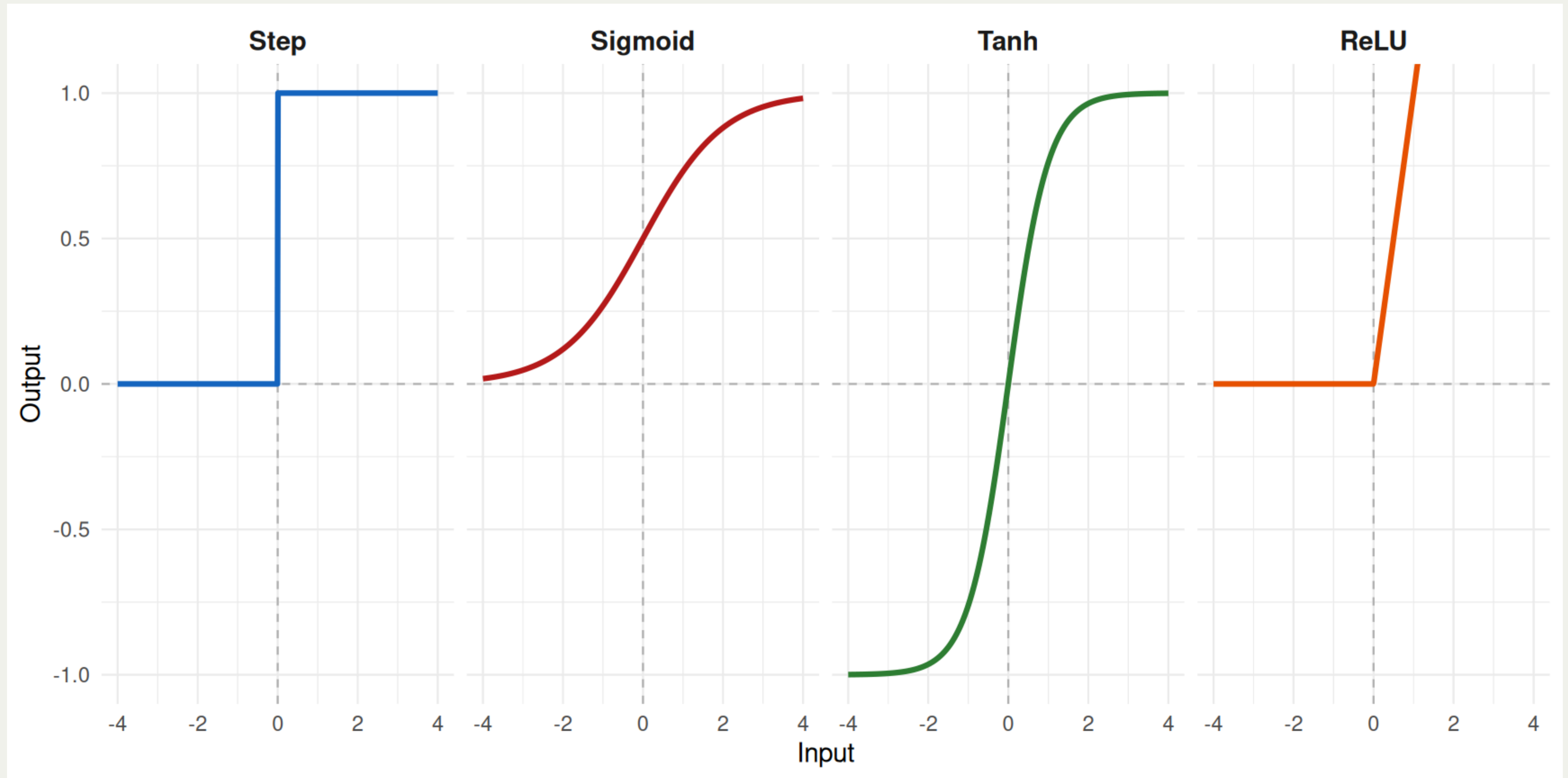
```
and_pred_binary <- ifelse(and_pred >= 0.5, 1, 0)
and_pred_binary
```

```
1 2 3 4 
0 0 0 1 
```

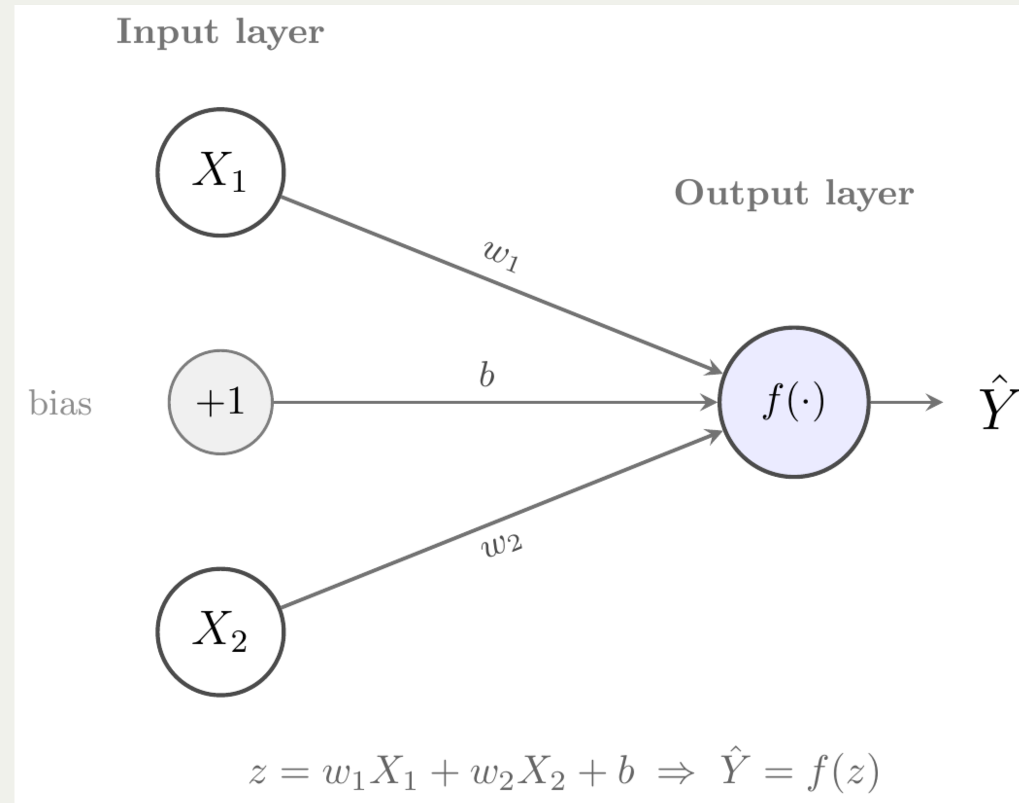
Activation Function

- The function `ifelse()` is, effectively, the simplest possible **activation function**.
- Also knows as a **step function**, it outputs 1 if the input exceeds a certain threshold and 0 otherwise.
- This is similar to the original McCulloch-Pitts model of a neuron, which biologically follow 'all-or-nothing' firing pattern.
- In practice, modern artificial neural networks use more complex non-linear activation functions.

Activation Functions

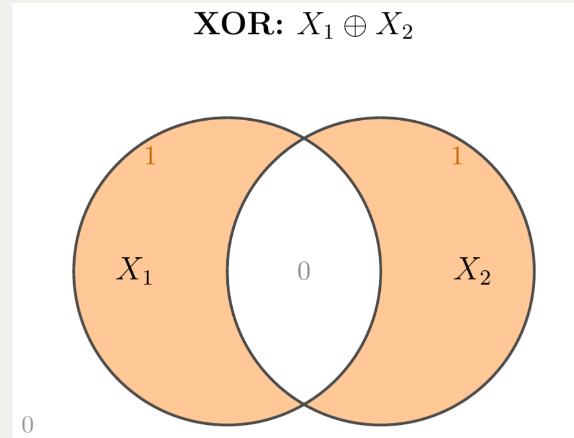


Single-Layer Perceptron (SLP)



- A **single-layer perceptron** has an input layer and a single output neuron.
- Each input X_j is multiplied by a learned weight w_j; a bias b shifts the threshold.
- The output neuron applies an activation function f to the weighted sum: $\hat{Y} = f\left(\sum_j w_j X_j + b\right)$.

Example: Exclusive OR (XOR)



- For 2 binary variables, the **exclusive OR (XOR)** is 1 if exactly one of the variables is 1, and 0 otherwise.

- We can think of it as a function f that takes two binary inputs (X_1, X_2) and produces a binary output Y:

$$Y = f(X_1, X_2) = \begin{cases} 1 & \text{if } (X_1 = 1 \text{ and } X_2 = 0) \text{ or } (X_1 = 0 \text{ and } X_2 = 1) \\ 0 & \text{otherwise} \end{cases}$$

Learning XOR Function

- Imagine we want to learn this function from the data.

X_1	X_2	$Y = X_1 \text{ XOR } X_2$
0	0	0
0	1	1
1	0	1
1	1	0

```r
xor_data <- data.frame(
  X1 = c(0, 0, 1, 1),
  X2 = c(0, 1, 0, 1),
  Y = c(0, 1, 1, 0)
)
```

Learning XOR Function with LM

- We might start by trying a linear model to learn this function:

```
xor_lm_fit <- lm(Y ~ X1 + X2, data = xor_data)
summary(xor_lm_fit)
```

```

Call:
lm(formula = Y ~ X1 + X2, data = xor_data)

Residuals:
   1    2    3    4 
-0.5  0.5  0.5 -0.5 

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 5.00e-01   8.66e-01   0.577    0.667
X1          1.11e-16   1.00e+00   0.000    1.000
X2          0.00e+00   1.00e+00   0.000    1.000

Residual standard error: 1 on 1 degrees of freedom
Multiple R-squared:  3.698e-32,	Adjusted R-squared:     -2 
F-statistic: 1.849e-32 on 2 and 1 DF,  p-value: 1
```

```
xor_lm_pred <- predict(xor_lm_fit, newdata = xor_data)
xor_lm_pred
```

```
  1   2   3   4 
0.5 0.5 0.5 0.5 
```

- This doesn't look good…

Learning XOR Function with GLM

- Given the binary outcome, we might consider a logistic regression model instead:

```r
xor_glm_fit <- glm(Y ~ X1 + X2, data = xor_data, family = binomial(link = "logit"))
summary(xor_glm_fit)
```

```

Call:
glm(formula = Y ~ X1 + X2, family = binomial(link = "logit"), 
    data = xor_data)

Coefficients:
              Estimate Std. Error z value Pr(>|z|)
(Intercept) -4.441e-16  1.732e+00       0        1
X1           4.441e-16  2.000e+00       0        1
X2           8.882e-16  2.000e+00       0        1

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 5.5452  on 3  degrees of freedom
Residual deviance: 5.5452  on 1  degrees of freedom
AIC: 11.545

Number of Fisher Scoring iterations: 2
```

```r
xor_glm_pred <- predict(xor_glm_fit, newdata = xor_data, type = "response")
xor_glm_pred
```

```
  1   2   3   4 
0.5 0.5 0.5 0.5 
```

Limitations of SLP

- The fundamental problem is that the XOR function is **not linearly separable**.
- This is a key limitation of single-layer perceptrons first demonstrated by Minsky and Papert (1969), which might have contributed to the decline in interest in artificial neural networks for several decades.
- However, by adding **hidden layers** of neurons, we can learn non-linearly separable functions like XOR and more complex relationships between inputs and outputs.
- But first let's talk about how we can learn the parameters (weights and biases) of these networks in the first place.

Stochastic Gradient Descent

Optimisation

- Most ML algorithms are based on some form of **optimisation**.
- Which refers to either minimising or maximising some function $f(x)$ by altering x.
- The function being optimised is the **objective function**.
- When we are minimising it, we usually call it **loss** or **error function**.

Derivative Recap

- Suppose we have a function $y = f(x)$ where x and y are real numbers.
- The **derivative** of this function denoted as $f'(x)$ or $\frac{dy}{dx}$,
- Is the slope of the function $f(x)$ at a given point x.
- A small change in the input can then be approximately mapped to a corresponding change in the output as:

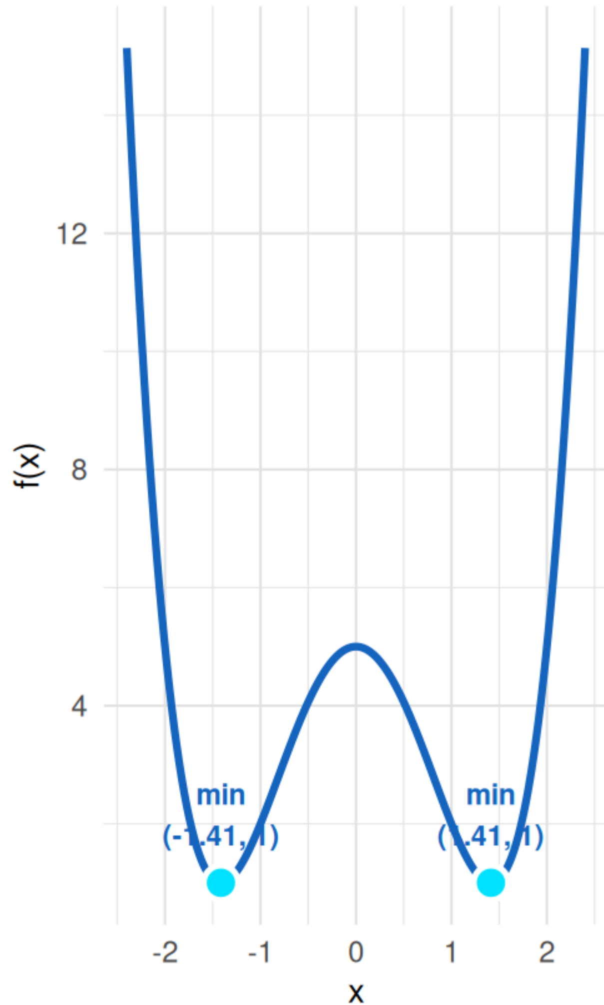
$$f(x + \epsilon) \approx f(x) + \epsilon f'(x)$$

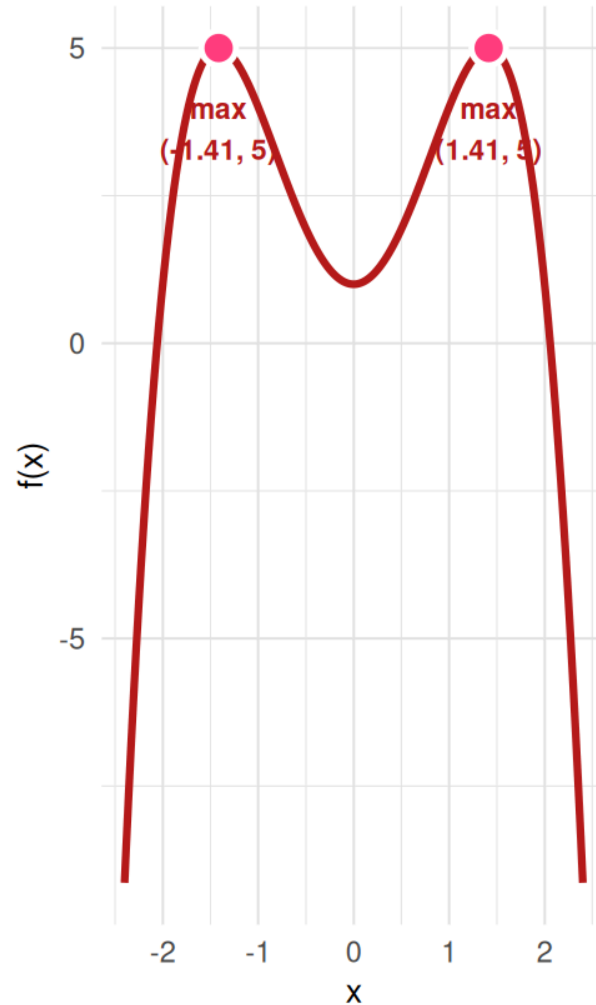
- For functions with multiple inputs, we must make use of **partial derivatives**.
- The partial derivative $\frac{\partial f}{\partial x_i}$ measures how f changes as we change only x_i.
- The **gradient** of a multivariable function is the vector of all its partial derivatives.

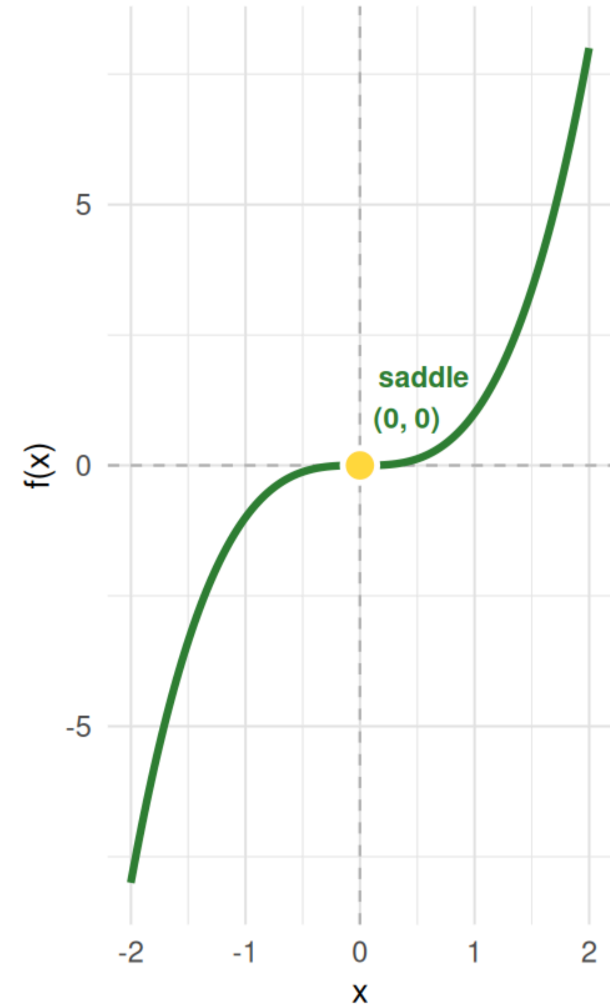
Gradient Descent

- Derivative is helpful in telling us how to change x to make an improvement in y.
- We can reduce y by changing x in the opposite direction of the derivative.
- This technique is known as **gradient descent**.
- When $f'(x) = 0$, the derivative provides no useful information about which direction to move in.
- Such **stationary** or **critical points** can correspond to local minima, local maxima, or saddle points.

Stationary (Critical) Points







Gradient Descent for 1 Variable

https://tpaskhalis-gradient-descent.share.connect.posit.cloud/#/gradient-descent-for-1-variable

Gradient Descent for 2 Variables

https://tpaskhalis-gradient-descent.share.connect.posit.cloud/#/gradient-descent-for-2-variables

Example: Linear Regression

- Let's take a look at the classic linear regression problem:

$$y_i = \beta_1 x_{i1} + \beta_2 x_{i2} + \ldots + \beta_p x_{ip} + \epsilon_i = \mathbf{x}_i^T \boldsymbol{\beta} + \epsilon_i$$

- Or, in matrix form:

$$\mathbf{y} = \mathbf{X}\boldsymbol{\beta} + \boldsymbol{\epsilon}$$

- Recall that the OLS solution for linear regression is:

$$\hat{\boldsymbol{\beta}} = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y}$$

- In ML literature $\boldsymbol{\beta}$ would often be seen as simply some **weights** $\boldsymbol{w}$ that we want to learn from the data.

Example: OLS

```
import numpy as np

rng = np.random.default_rng(seed = 123)
X = rng.normal(size = (100, 2))
y = 1.5 * X[:, 0] - 2.0 * X[:, 1] + rng.normal(size = 100)
```

- As we have seen in POP77001, the OLS solution is very simple to implement:

```
betas = np.linalg.inv(X.T @ X) @ X.T @ y
betas
```

```
array([ 1.470222  , -2.02156959])
```

- Constructing an inverse of the $N \times N$ sized matrix through $(\mathbf{X}^T\mathbf{X})^{-1}$/`np.linalg.inv(X.T @ X)` can, however, be computationally expensive for large N.

Regression with Gradient Descent

- Instead, we use gradient descent to find the optimal $\boldsymbol{\beta}$ by minimising the mean squared error (MSE) loss function:

$$L(\boldsymbol{\beta}) = \frac{1}{N}\sum_{i=1}^{N}(y_i - \mathbf{x}_i^T\boldsymbol{\beta})^2$$

- The gradient (partial derivatives) of the MSE loss with respect to $\boldsymbol{\beta}$ is:

$$\nabla L(\boldsymbol{\beta}) = -\frac{2}{N}\sum_{i=1}^{N}(y_i - \mathbf{x}_i^T\boldsymbol{\beta})\mathbf{x}_i$$

- Note that a large part of this, namely:

$$(y_i - \mathbf{x}_i^T\boldsymbol{\beta})$$

are just the residuals.

Regression with Gradient Descent

- If we were to write the last equation for gradient descent in Python:

```
y_pred = X @ betas
residuals = y - y_pred
gradient = -2 * X.T @ residuals / len(y)
```

- Putting this all together:

```
betas = np.array([0.0, 0.0])  # initial weights
lr = 0.01 # learning rate
```

```
for epoch in range(1000):
    y_pred = X @ betas
    residuals = y - y_pred
    gradient = -2 * X.T @ residuals / len(y) # compute gradient
    betas = betas - lr * gradient  # update weights
```

```
betas
```

```
array([ 1.47022181, -2.02156964])
```

Gradient Descent for Linear Regression

https://tpaskhalis-gradient-descent.share.connect.posit.cloud/#/gradient-descent-for-linear-regression

Why/Why Not Gradient Descent?

- **Advantages**:
- Scalable to large datasets and high-dimensional feature spaces.
- Can be applied to a wide variety of models and loss functions.
- Can be used for online learning (updating model with new data).

- **Disadvantages**:
- Is not guaranteed to find even local minima in reasonable time.
- Sensitive to choices of hyperparamters (e.g. initial values and learning rate).
- Can be less efficient than closed-form solutions for small datasets or low-dimensional feature spaces.

Stochastic Gradient Descent

- A usual problem in ML is that large training sets are good for generalisation but can be computationally expensive.
- The overall cost functions is usually just a sum over some loss function applied to each training example.
- The computational cost of such operations is typically $O(N)$, i.e. linear in the number of training examples N.
- However, what gradient, effectively, produces is an expectation, which could be appxoimated by taking a sample of the training data.
- **Stochastic gradient descent (SGD)** samples a *minibatch* of size $N' < N$ at each iteration to compute an estimate of the gradient.
- Crucially, the size of the minibatch N' is held fixed as N grows, so the computational cost becomes $O(N')$, which is constant with respect to N.

Regression with SGD

- We can update our previous code to incorporate the idea of minibatches:

```
betas = np.array([0.0, 0.0])  # initial weights
lr = 0.01 # learning rate
batch_size = 20

for epoch in range(1000):
    # Sample a random minibatch of data
    indices = np.random.choice(len(y), size=batch_size, replace=False)
    X_batch, y_batch = X[indices], y[indices]
    y_pred = X_batch @ betas
    residuals = y_batch - y_pred
    gradient = -2 * X_batch.T @ residuals / batch_size
    betas = betas - lr * gradient

betas
```

```
array([ 1.46308608, -1.99624894])
```

Back to Boolean AND

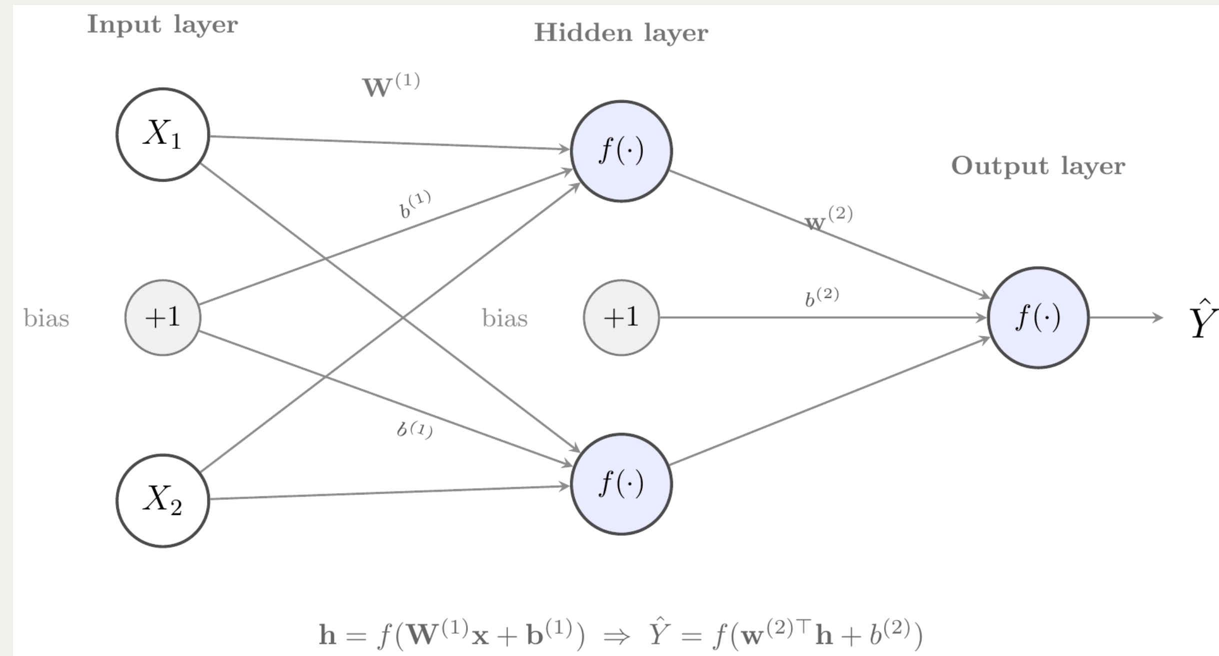
- Let's return to our original example of learning the Boolean AND function:

```python
def step(z):
    return np.where(z >= 0, 1.0, 0.0)

X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y = np.array([0, 0, 0, 1])

params = np.zeros(3)  # initial weights (w1, w2, b)
lr = 0.1 # learning rate
batch_size = 10

for epoch in range(1000):
    indices = np.random.choice(len(y), size=batch_size, replace=True)
    X_batch, y_batch = X[indices], y[indices]
    z = X_batch @ params[:2] + params[2]  # linear step
    y_pred = step(z)                      # step activation
    residuals = y_batch - y_pred          # perceptron error (0 or ±1)
    gradient_w = -2 * X_batch.T @ residuals / batch_size
    gradient_b = -2 * np.sum(residuals) / batch_size
    params[:2] = params[:2] - lr * gradient_w
    params[2]  = params[2]  - lr * gradient_b

params
```

```
array([ 0.14,  0.1 , -0.14])
```

Multi-Layer Perceptron

Multi-Layer Perceptron (MLP)



- A **multi-layer perceptron (MLP)** adds one or more **hidden layers** between the inputs and the output.
- Each hidden neuron applies an activation function f to its weighted inputs, learning non-linear intermediate representations.
- With a hidden layer and non-linear activation, an MLP can represent **non-linearly separable** functions such as XOR.

MLP for XOR

```
# Rectified Linear Unit (ReLU)
def relu(z):
    return np.maximum(0, z)

# Sigmoid activation functions
def sigmoid(z):
    return 1 / (1 + np.exp(-z))

X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y = np.array([0, 1, 1, 0])

rng = np.random.default_rng(1)
W1 = rng.normal(scale=0.5, size=(2, 2))  # hidden weights (2×2)
b1 = np.zeros(2)                          # hidden biases
w2 = rng.normal(scale=0.5, size=2)        # output weights
b2 = 0.0                                  # output bias
lr = 0.1
batch_size = 10

for epoch in range(1000):
    indices = np.random.choice(len(y), size=batch_size, replace=True)
    X_batch, y_batch = X[indices], y[indices]
    z1 = X_batch @ W1 + b1          # hidden pre-activation
    h  = relu(z1)                    # hidden layer (ReLU)
    y_pred = sigmoid(h @ w2 + b2)   # output (sigmoid)
    residuals = y_batch - y_pred
    sig_d = y_pred * (1 - y_pred)   # sigmoid derivative
    delta = np.outer(residuals * sig_d, w2) * (z1 >= 0) # backpropagation
    gradient_W1 = -2 * X_batch.T @ delta / batch_size
    gradient_b1 = -2 * np.sum(delta, axis=0) / batch_size
    gradient_w2 = -2 * h.T @ (residuals * sig_d) / batch_size
    gradient_b2 = -2 * np.sum(residuals * sig_d) / batch_size
    W1 -= lr * gradient_W1;  b1 -= lr * gradient_b1
```

MLP for XOR

- Starting from the hidden layer (prior to activation):

```
1 X @ W1 + b1
```

```
array([[-9.44043821e-03, -5.83547591e-04],
       [ 1.49551819e+00, -1.63684191e+00],
       [-1.51966871e+00,  1.63538832e+00],
       [-1.47100824e-02, -8.70039342e-04]])
```

- This is the output post-transformation by the hidden layer activation function:

```
1 relu(X @ W1 + b1)
```

```
array([[0.        , 0.        ],
       [1.49551819, 0.        ],
       [0.        , 1.63538832],
       [0.        , 0.        ]])
```

- As well as the output layer pre-activation values:

```
1 relu(X @ W1 + b1) @ w2 + b2
```

```
array([-1.44167829,  1.78568403,  2.13343551, -1.44167829])
```

- Finally, we can apply the activation function to get the final predictions:

```
1 step(relu(X @ W1 + b1) @ w2 + b2)
```

```
array([0., 1., 1., 0.])
```

```
1  sigmoid(relu(X @ W1 + b1) @ w2 + b2)
```

```
array([0.19128559, 0.85639731, 0.89411071, 0.19128559])
```

Next

- Tutorial: Working with Neural Networks
- Next Week: Transformers