Week 9: Methods Review and Neural Network Fundamentals

MY360/459 Quantitative Text Analysis

https://lse-my459.github.io/

Friedrich Geiecke

1. Introduction and Foundations
2. Quantifying Texts
3. Exploiting Word Meanings
4. Classifying Texts into Categories
5. Scaling Latent Traits Using Texts
6. *Reading Week*
7. Text Similarity and Clustering
8. Probabilistic Topic Models
9. Methods Review and Neural Network Fundamentals
10. Static Word Embeddings
11. Introduction to Large Language Models

Today

- The last weeks of the course cover current computational methods for texts based on neural networks and conclude with an introduction to large language models such as e.g. ChatGPT
- To discuss these topics, it is very helpful to begin with a review of some key methods
- The second part of today's lecture will then be an introduction to neural networks

Outline

- Methods Review
  - Vectors & matrices
  - Derivatives
  - Gradient descent
- Introduction to neural networks

Vectors

- Vectors such as $a = \begin{pmatrix} 7 \\ -3 \\ 5.3 \\ -2.7 \end{pmatrix}$ can be interpreted in different ways, e.g. as ordered lists storing numbers, points in space, or representations of direction
- We will use vectors in various models; they carry data but also e.g. encode meaning of language numerically
- Visual intuition is very helpful, in this course in particular for the discussion of word vectors in the next weeks

Vector spaces

- A vector space is a set V of vectors with two fundamental operations: **Scalar multiplication** and **vector addition**
- Applied to the vector spaces with real numbers $\mathbb{R}^n$ that we typically think about in this course:
  - **Scalar multiplication:** Scale a vector by a real number. For any scalar $\alpha \in \mathbb{R}$ and vector $v = (v_1, v_2, \ldots, v_n)$,

$$\alpha v = (\alpha v_1, \alpha v_2, \ldots, \alpha v_n)$$

  - **Vector addition:** Combine two vectors to produce another vector in the same space. For $u = (u_1, u_2, \ldots, u_n)$ and $v = (v_1, v_2, \ldots, v_n)$ in $\mathbb{R}^n$, the sum is given by

$$u + v = (u_1 + v_1, u_2 + v_2, \ldots, u_n + v_n)$$

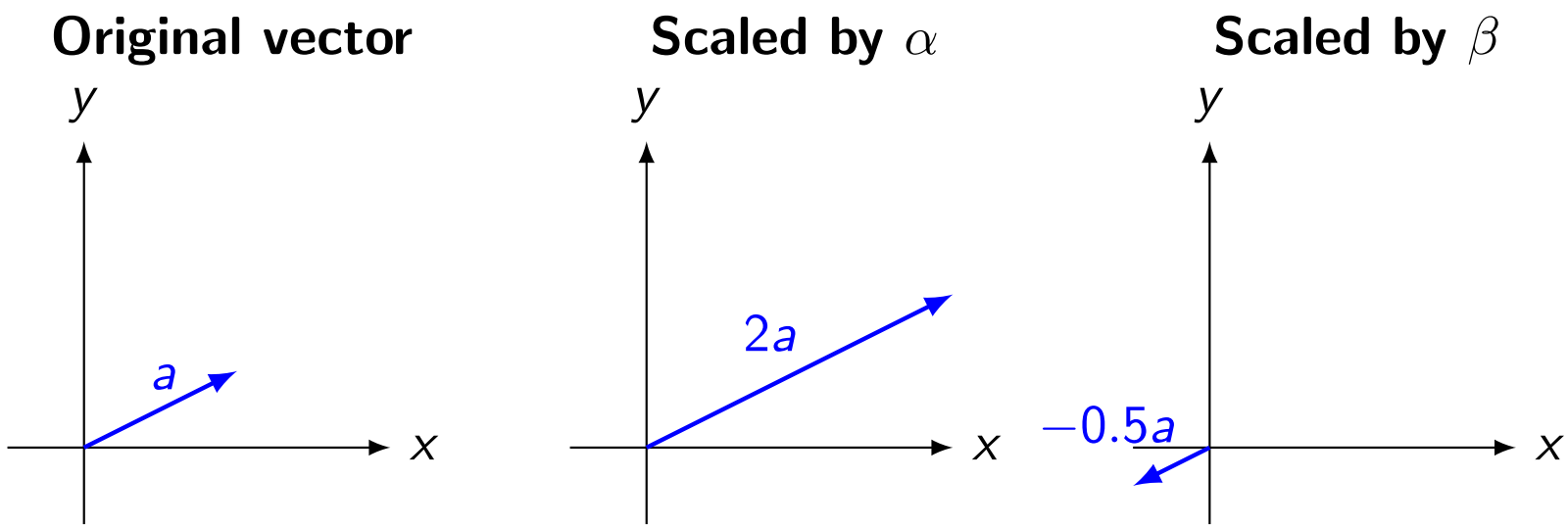
Visual intuition: Scalar multiplication

Consider a vector

$$a = \begin{pmatrix} 2 \\ 1 \end{pmatrix}$$

multiplied by scalars $\alpha = 2$ and $\beta = -0.5$:

$$2a = \begin{pmatrix} 4 \\ 2 \end{pmatrix}, \quad -0.5a = \begin{pmatrix} 1 \\ -0.5 \end{pmatrix}$$

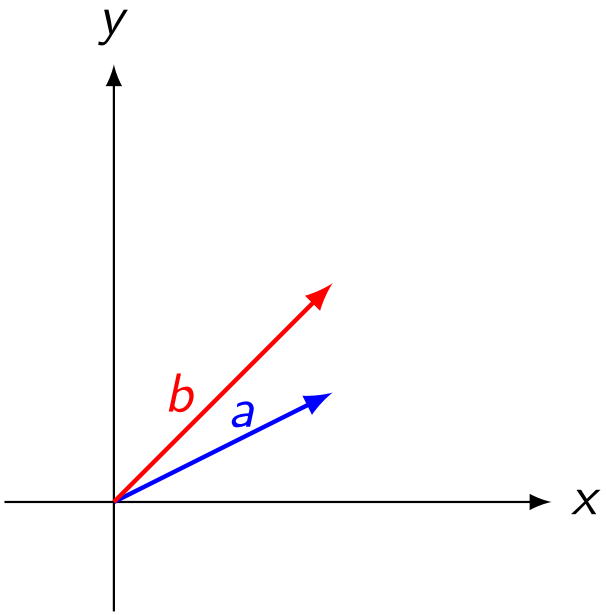


Visual intuition: Vector addition

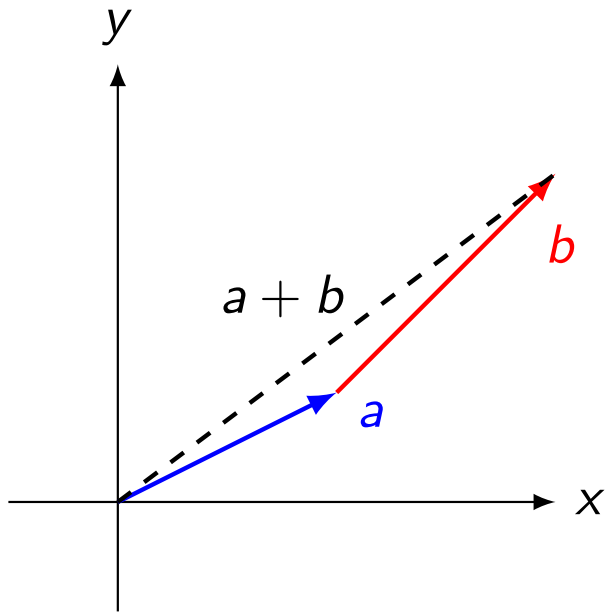
$$a = \begin{pmatrix} 2 \\ 1 \end{pmatrix} \quad b = \begin{pmatrix} 2 \\ 2 \end{pmatrix}$$

$$a + b = \begin{pmatrix} 2+2 \\ 1+2 \end{pmatrix} = \begin{pmatrix} 4 \\ 3 \end{pmatrix}$$

Individual vectors



Addition

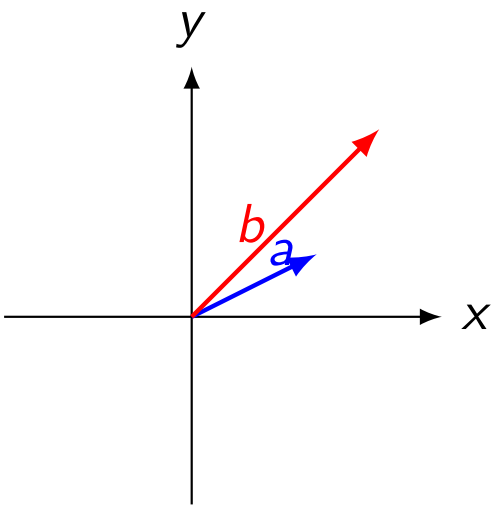


Visual intuition: Vector subtraction

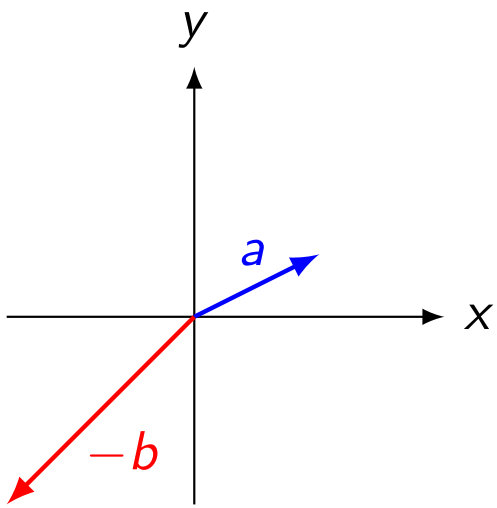
$$a = \begin{pmatrix} 2 \\ 1 \end{pmatrix}, \quad b = \begin{pmatrix} 3 \\ 3 \end{pmatrix},$$

$$a - b = a + (-b) = \begin{pmatrix} 2 - 3 \\ 1 - 3 \end{pmatrix} = \begin{pmatrix} -1 \\ -2 \end{pmatrix}$$

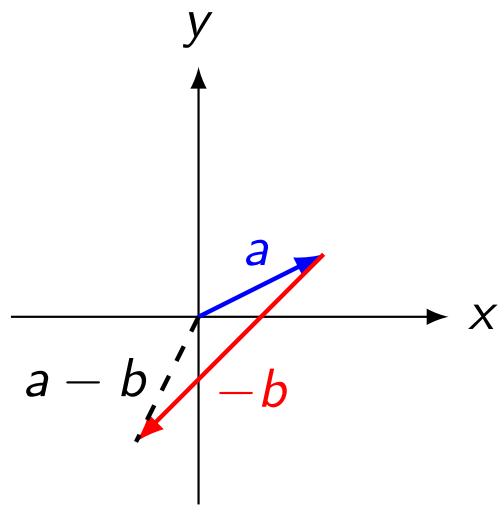
Individual vectors



Scaled vector b



Addition



The dot product

- **Definition:** For vectors $u, v \in \mathbb{R}^n$, the dot product is defined as

$$u \cdot v = \sum_{i=1}^{n} u_i v_i$$

- It produces a scalar. For example, for $u = (1, 3, -2)$ and $v = (0, 5, -2)$:

$$u \cdot v = 1 \times 0 + 3 \times 5 + -2 \times -2 = 19$$

Matrix multiplication

- Matrices are rectangular arrays with rows and columns
- Matrix multiplication generalises the dot product between vectors
- It is a key operation in neural networks and standard regression models
- Properties of matrix multiplication:
  - Compatibility: The number of columns in A must equal the number of rows in B
  - Associativity: $(AB)C = A(BC)$
  - Distributivity: $A(B + C) = AB + AC$ and $(A + B)C = AC + BC$
  - Identity: $AI = IA = A$, where $I_n = \mathrm{diag}(1, 1, \ldots, 1)$
  - Scalar multiplication: $c(AB) = (cA)B = A(cB)$ for any scalar c
  - Non-commutativity: In general, $AB \neq BA$

Definition and examples

Let $A \in \mathbb{R}^{m\times n}$ and $B \in \mathbb{R}^{n\times p}$. Then the product $AB \in \mathbb{R}^{m\times p}$ is given by

$$(AB)_{ik} = \sum_{j=1}^{n} a_{ij} b_{jk}, \quad \text{for } i = 1, \ldots, m \text{ and } k = 1, \ldots, p.$$

Example 1:

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \quad \text{and} \quad x = \begin{pmatrix} 5 \\ 6 \end{pmatrix}$$

$$\underbrace{A}_{2\times 2}\,\underbrace{x}_{2\times 1} = \begin{pmatrix} 1\cdot 5 + 2\cdot 6 \\ 3\cdot 5 + 4\cdot 6 \end{pmatrix} = \begin{pmatrix} 5 + 12 \\ 15 + 24 \end{pmatrix} = \begin{pmatrix} 17 \\ 39 \end{pmatrix}$$

Example 2:

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \quad \text{and} \quad B = \begin{pmatrix} 7 & 8 \\ 9 & 10 \end{pmatrix}$$

$$\underbrace{A}_{2\times 2}\,\underbrace{B}_{2\times 2} = \begin{pmatrix} 1\cdot 7 + 2\cdot 9 & 1\cdot 8 + 2\cdot 10 \\ 3\cdot 7 + 4\cdot 9 & 3\cdot 8 + 4\cdot 10 \end{pmatrix} = \begin{pmatrix} 7 + 18 & 8 + 20 \\ 21 + 36 & 24 + 40 \end{pmatrix} = \begin{pmatrix} 25 & 28 \\ 57 & 64 \end{pmatrix}$$

Example 3:

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix} \quad \text{and} \quad B = \begin{pmatrix} 7 & 8 \\ 9 & 10 \end{pmatrix}.$$

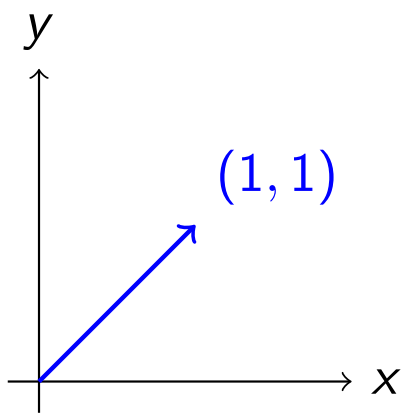
$$\underbrace{A}_{3\times 2}\,\underbrace{B}_{2\times 2} = \begin{pmatrix} 1\cdot 7 + 2\cdot 9 & 1\cdot 8 + 2\cdot 10 \\ 3\cdot 7 + 4\cdot 9 & 3\cdot 8 + 4\cdot 10 \\ 5\cdot 7 + 6\cdot 9 & 5\cdot 8 + 6\cdot 10 \end{pmatrix} = \begin{pmatrix} 25 & 28 \\ 57 & 64 \\ 89 & 100 \end{pmatrix}.$$

Visual intuition: Matrix multiplication

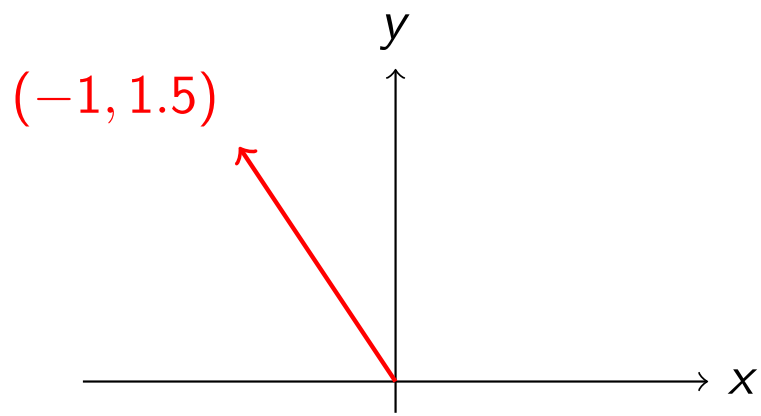
- While often taught as an algorithm, matrix multiplication has a nice geometric interpretation. It linearly moves ("transforms") points in a space to new positions in the same or another space

- For example, multiplying vectors with $A = \begin{pmatrix} 1 & -2 \\ -0.5 & 2 \end{pmatrix}$:

Exemplary vector $(1, 1)$



New position $(-1, 1.5)$



$$A \begin{pmatrix} 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \cdot 1 + (-2) \cdot 1 \\ -0.5 \cdot 1 + 2 \cdot 1 \end{pmatrix} = \begin{pmatrix} -1 \\ 1.5 \end{pmatrix}$$

Cosine similarity

- Lastly, let us briefly review cosine similarity between vectors which we use e.g. to compute similarities between embeddings that encode documents, words, etc. (word and more complex document embeddings can contain negative numbers!)

$$\text{cosine similarity}(a, b) = \frac{a \cdot b}{\|a\|\|b\|} \in [-1, 1]$$

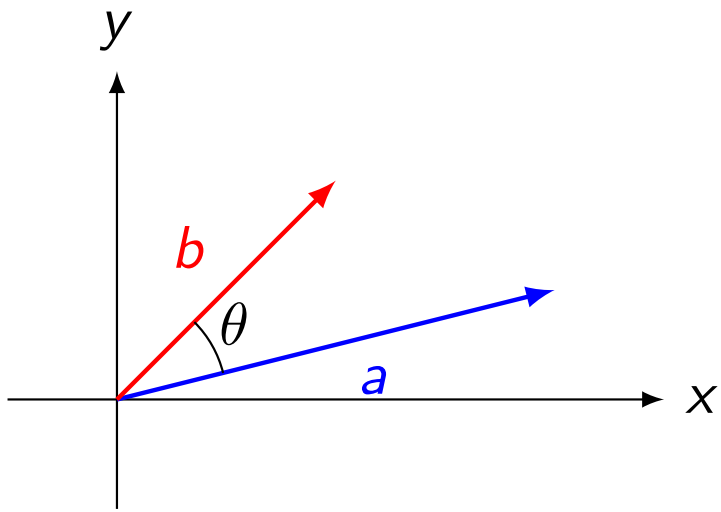
- The length of a vector is given by $\|x\| = \sqrt{x \cdot x} = \sqrt{\sum_{i=1}^{n} x_i^2}$
- Note that cosine similarity simply follows from the geometric interpretation of the dot product:

$$u \cdot v = \|u\|\|v\| \cos\theta$$

where θ is the angle between u and v

Visual intuition: Cosine similarity

$$a = \begin{pmatrix} 4 \\ 1 \end{pmatrix}, \quad b = \begin{pmatrix} 2 \\ 2 \end{pmatrix}, \quad \frac{a \cdot b}{\|a\|\|b\|} \approx 0.8575 = cos(\theta)$$



In radians: $cos^{-1}(0.8575) = \theta \approx 0.5404$

In degrees: $\theta_{degrees} = 0.5404 \frac{180}{\pi} \approx 30.96^{\circ}$

Outline

- Methods Review
  - Vectors & matrices
  - Derivatives
  - Gradient descent
- Introduction to neural networks

Calculus

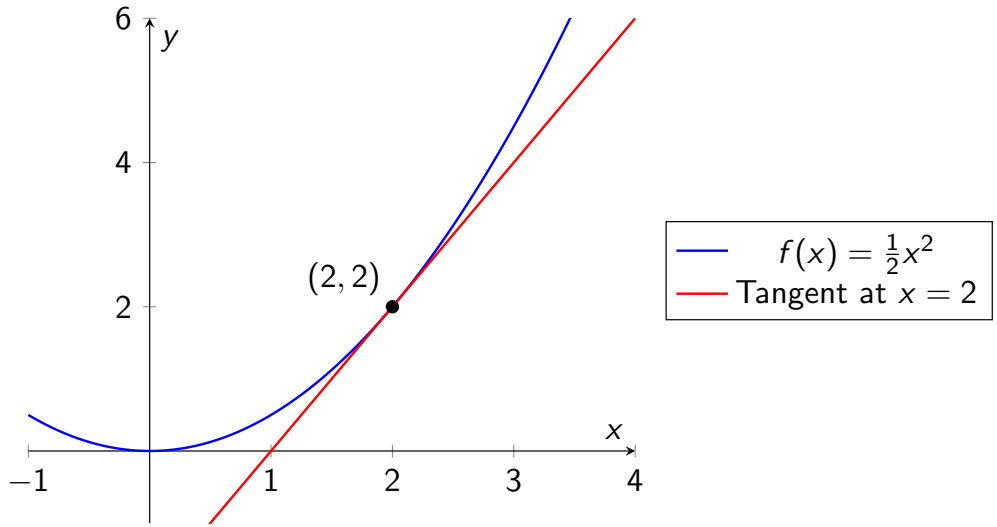
- Next, we will briefly review simple cases of derivatives of univariate functions, partial derivatives of multivariate functions, gradients, and the chain rule
- Even training current neural networks often just requires sequences of first order derivatives via the chain rule
- All generic functions here are assumed to be differentiable and the overview is stylised

Definition

Definition (limit form):

$$\frac{df}{dx} \equiv f'(x) = \lim_{\Delta x \to 0} \frac{f(x + \Delta x) - f(x)}{\Delta x}$$

- $f'(x)$ is the rate of change of f at the point x
- Graphically, $f'(x)$ is the slope of the tangent line to $f(x)$ at point x



Examples

Some common cases:

$$\frac{d}{dx}(x^n) = nx^{n-1}, \quad \frac{d}{dx}(e^x) = e^x, \quad \frac{d}{dx}(\log x) = \frac{1}{x}.$$

- Linearity: $\frac{d}{dx}\big(a \cdot f(x) + b \cdot g(x)\big) = a \cdot f'(x) + b \cdot g'(x)$.
- Product Rule: $\frac{d}{dx}\big(f(x)g(x)\big) = f'(x)g(x) + f(x)g'(x)$.
- Quotient Rule: $\frac{d}{dx}\left(\frac{f(x)}{g(x)}\right) = \frac{f'(x)g(x) - f(x)g'(x)}{(g(x))^2}$.

Partial derivatives

Definition:

$$\frac{\partial f}{\partial x}(x,y) = \lim_{\Delta x \to 0} \frac{f(x+\Delta x, y) - f(x,y)}{\Delta x}.$$

$$\frac{\partial f}{\partial y}(x,y) = \lim_{\Delta y \to 0} \frac{f(x, y+\Delta y) - f(x,y)}{\Delta y}.$$

- The concept naturally extends to functions of more than two variables
- Simply hold all other variables constant and differentiate with respect to one variable
- For example: $f(x,y,\alpha,\beta) = y - \alpha - \beta x$

$$\frac{\partial f(x,y,\alpha,\beta)}{\partial \beta} = -x$$

Gradients

- For a function $f(x, y, z)$, the gradient is denoted by:

$$\nabla f(x, y, z) = \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z} \right).$$

- It points into the direction of steepest ascent
- We will require gradients for training neural networks

Chain rule

- To determine such gradients will usually require the chain rule
- **Definition:** If $y = f(u)$ and $u = g(x)$, then

$$\frac{dy}{dx} = \frac{dy}{du} \cdot \frac{du}{dx}$$

- Examples:
  - If $f(x) = (x+2)^2$, then $f(x) = g(x)^2$ with $g(x) = x + 2$

$$\frac{df}{dx} = 2(x+2)1$$

  - If $f(x) = log(10 + x^2)$, then $f(x) = log(g(x))$ and $g(x) = 10 + x^2$

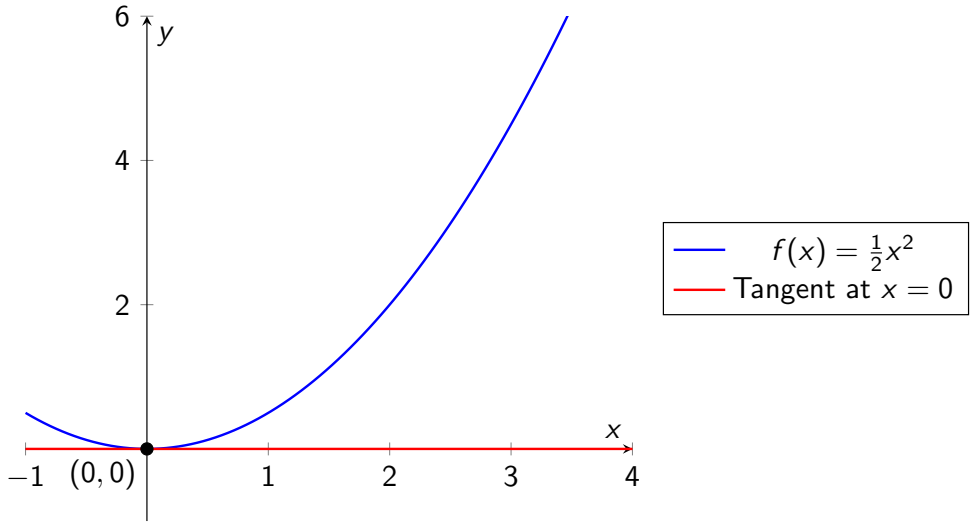
$$\frac{df}{dx} = \frac{1}{10 + x^2} 2x$$

Outline

- Methods Review
  - Vectors & matrices
  - Derivatives
  - Gradient descent
- Introduction to neural networks

Optimisations with derivatives

- For differentiable functions, local minima (or maxima) occur where the derivative equals zero



The function's minimum is at $x = 0$ where its slope is zero, i.e. where $f'(x) = x = 0$

- Yet, not every point with a zero derivative is an extremum; some are saddle points where the function changes curvature

- Trying to find a minimum in an "error function" of neural networks (or many simpler models) called *loss* is what makes them "learn"

Minimising loss functions

- Next, we will define such loss functions and study how to minimise them with gradient descent
- Before discussing more complex neural networks, we will implement the approach for simple regression and classification models from scratch to gain intuition
- We will begin with linear regression, estimating the intercept and slope of the model $y_i = \alpha + \beta x_i + \varepsilon_i$ with $\varepsilon_i \sim N(0, \sigma)$

Loss of linear regression

- The goal is to minimise the simple mean squared error loss:

$$L(\alpha, \beta; \{x_i, y_i\}_{i=1}^n) = \frac{1}{n} \sum_{i=1}^{n} (y_i - (\alpha + \beta x_i))^2$$

- For an individual observation:

$$L_i = (y_i - (\alpha + \beta x_i))^2$$

Gradient of loss

- Loss:

$$L_i = (y_i - (\alpha + \beta x_i))^2$$

- Derivatives using the chain rule:

$$\frac{\partial L_i}{\partial \alpha} = 2\left(y_i - (\alpha + \beta x_i)\right)(-1) = -2e_i$$

$$\frac{\partial L_i}{\partial \beta} = 2\left(y_i - (\alpha + \beta x_i)\right)(-x_i) = -2e_i x_i$$

where $e_i = y_i - (\alpha + \beta x_i)$

e is the residual

Gradient descent for linear regression

- Initialise parameters $\alpha = 0$, $\beta = 0$; choose learning rate η
- Repeat for E epochs or until convergence:

epochs are iterations

size of the step we are making while waling alongside the line

1. Predict the residual for all observations:

$$e_i = y_i - (\alpha + \beta x_i), \quad i = 1, \ldots, n$$

2. Update the parameters using the average gradients:

Loss function is MSE

$$\alpha \leftarrow \alpha - \eta \frac{1}{n} \sum_{i=1}^{n} (-2e_i)$$

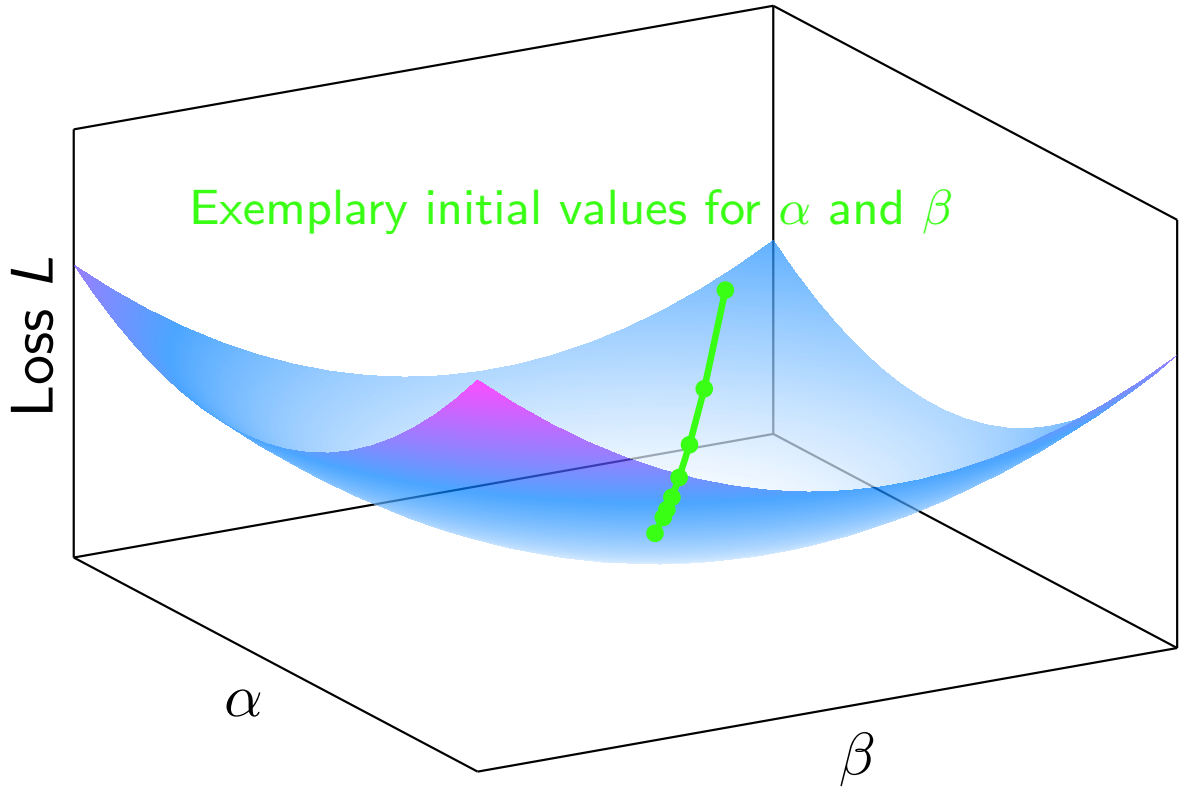
the new alpha is the old alpha minus the derivative

$$\beta \leftarrow \beta - \eta \frac{1}{n} \sum_{i=1}^{n} (-2e_i x_i)$$

Visual intuition: Gradient descent

Loss is the sum (or average) of all residuals

alpha and beta can be set equal to 0 as the initial first guess values



for neural network - it cannot be 0

Coding

- 01-linear-regression-with-gradient-descent.qmd

Classification with logistic regression

Why don't we look at regularisation in with a text and its length of words? Because we only have one variable X

- The second broad class of problems in supervised machine learning is classification
- We will study the example of binary classification with logistic regression
- Conceptually, the gradient descent approach remains the same as before, we will simply minimise a different loss function
- Recall the definition of the logistic (or sigmoid) function; we will again assume a model with only one slope and an intercept

$$p_i = \frac{1}{1 + e^{-z_i}} = \frac{1}{1 + e^{-(\alpha + \beta x_i)}}$$

we wrap the linear regression function in the sigmoid function to transpose it to a space from 0 to 1

Loss of the logistic regression

- Logistic regression minimises the logistic (binary cross entropy) loss:

$$L(\alpha, \beta; \{x_i, y_i\}_{i=1}^n) = -\frac{1}{n}\sum_{i=1}^{n} y_i log\,(p_i) + (1 - y_i) log\,(1 - p_i)$$

Why is this a sensible loss function?

Why are we using logs?

This function is enforcing that I am predicting probabilities close to 1 for the true class

In other words, loss function is huge when we predict probabilities wrong

log(1) = 0 --> no loss

- For an individual observation:

$$L_i = -\left[y_i log\,(p_i) + (1 - y_i) log\,(1 - p_i)\right]$$

yi in front of logs
collapses the terms to 0 if they are in a certain class

- Substituting $p_i = \frac{1}{1+e^{-\alpha-\beta x_i}}$ and $1 - p_i = \frac{e^{-\alpha-\beta x_i}}{1+e^{-\alpha-\beta x_i}}$:

$$L_i = -\left[y_i log\left(\frac{1}{1+e^{-\alpha-\beta x_i}}\right) + (1 - y_i) log\left(\frac{e^{-\alpha-\beta x_i}}{1+e^{-\alpha-\beta x_i}}\right)\right]$$

- After some algebra, this simplifies to link to derivation:

$$L_i = (1 - y_i)(\alpha + \beta x_i) + log(1 + e^{-\alpha-\beta x_i})$$

Gradient of loss

- Loss: $L_i = (1 - y_i)(\alpha + \beta x_i) + log(1 + \underbrace{e^{-\alpha-\beta x_i}}_{= y})$
- Derivatives using the chain rule:

$$\frac{\partial L_i}{\partial \alpha} = (1 - y_i) + \underbrace{\frac{1}{1 + e^{-\alpha-\beta x_i}} e^{-\alpha-\beta x_i}}_{1-p_i}(-1)$$

$$= (1 - y_i) - (1 - p_i)$$

$$= p_i - y_i$$

probability - the label

$$\frac{\partial L_i}{\partial \beta} = (1 - y_i)x_i + \underbrace{\frac{1}{1 + e^{-\alpha-\beta x_i}} e^{-\alpha-\beta x_i}}_{1-p_i}(-x_i)$$

$$= (1 - y_i)x_i - (1 - p_i)x_i$$

$$= (p_i - y_i)x_i$$

Gradient descent for logistic regression

- Initialise parameters $\alpha = 0$, $\beta = 0$; choose learning rate η
- Repeat for E epochs or until convergence:

1. Compute predicted probabilities for all observations:

$$p_i = \frac{1}{1 + \exp(-(\alpha + \beta x_i))}, \quad i = 1, \ldots, n.$$

2. Update the parameters using the average gradients:

$$\alpha \leftarrow \alpha - \eta \frac{1}{n} \sum_{i=1}^{n} (p_i - y_i)$$

We need to initialize apha and beta
to start computing the gradient descent

- without that initialisation the whole thing would not work

$$\beta \leftarrow \beta - \eta \frac{1}{n} \sum_{i=1}^{n} (p_i - y_i) x_i$$

Coding

- 02-logistic-regression-with-gradient-descent.qmd

Discussion

- This concludes our brief methods review
- The topics we revised in linear algebra, calculus and optimisation will serve as a fundament for the remaining content of the course

Outline

- Methods Review
  - Vectors & matrices
  - Derivatives
  - Gradient descent
- Introduction to neural networks

Some history of neural networks

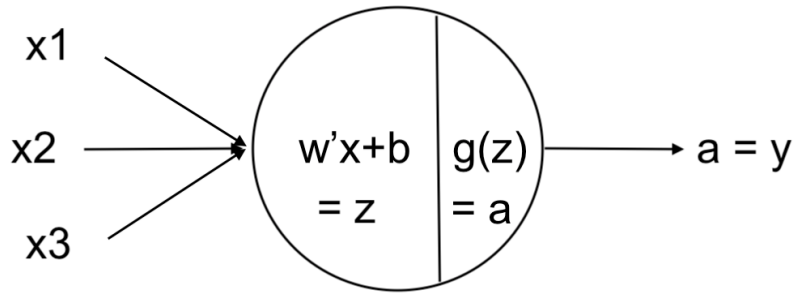
- McCulloch and Pitts (1943) created a computational model for neural networks
- Research on the topic continued throughout the 20th century, however, at times as a niche field
- In the early 21st century the models began to outperform others at a large scale
- In particular, AlexNet (Krizhevsky et al., 2012) won the 2012 ImageNet (an image classification) competition by a high margin (for many researchers at the time that was unexpected)

Why have neural networks become so important in recent years?

- Data: Much more (labeled) data available
- Hardware: Much more compute available (Moore's law); extensive use of GPUs for parallelisation of matrix operations
- Software: Improved architectures, libraries, optimisation algorithms, etc

Simplest architecture: Single layer perceptron

one layer, one neuron



logistic regression is a neural network

- **Single layer perceptron (SLP)** here: A (**feedforward**) neural network with no hidden layer and an output layer with a single neuron and activation
- $z = x'w + b = x_1 w_1 + x_2 w_2 + x_3 w_3 + b$ with 3-dimensional **weights** vector $w = (w_1, w_2, w_3)$ and **bias** b

  slope — intercept
- $a = g(z)$ where $g(z)$ is called **activation function**
- Circles in figures commonly depict both z and a values
- Exercise: With **sigmoid activation** $g(z) = \frac{1}{1+e^{-z}}$, this is the same as a ...

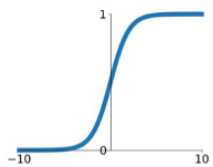
Activation functions

- Activation functions add non-linearity
- Sigmoid can have close to zero gradients for both negative and positive values and imply slow learning
- ReLU (rectified linear unit) and variants are commonly used today

sigmoid function is not great
for leanring because
we do not want the derivatives to be 0s

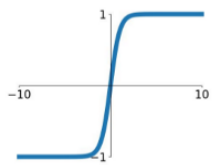
Sigmoid

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



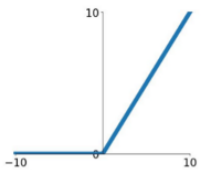
tanh

$$\tanh(x)$$



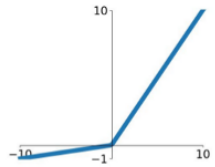
ReLU

$$\max(0, x)$$



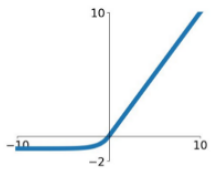
Leaky ReLU

$$\max(0.1x, x)$$



ELU

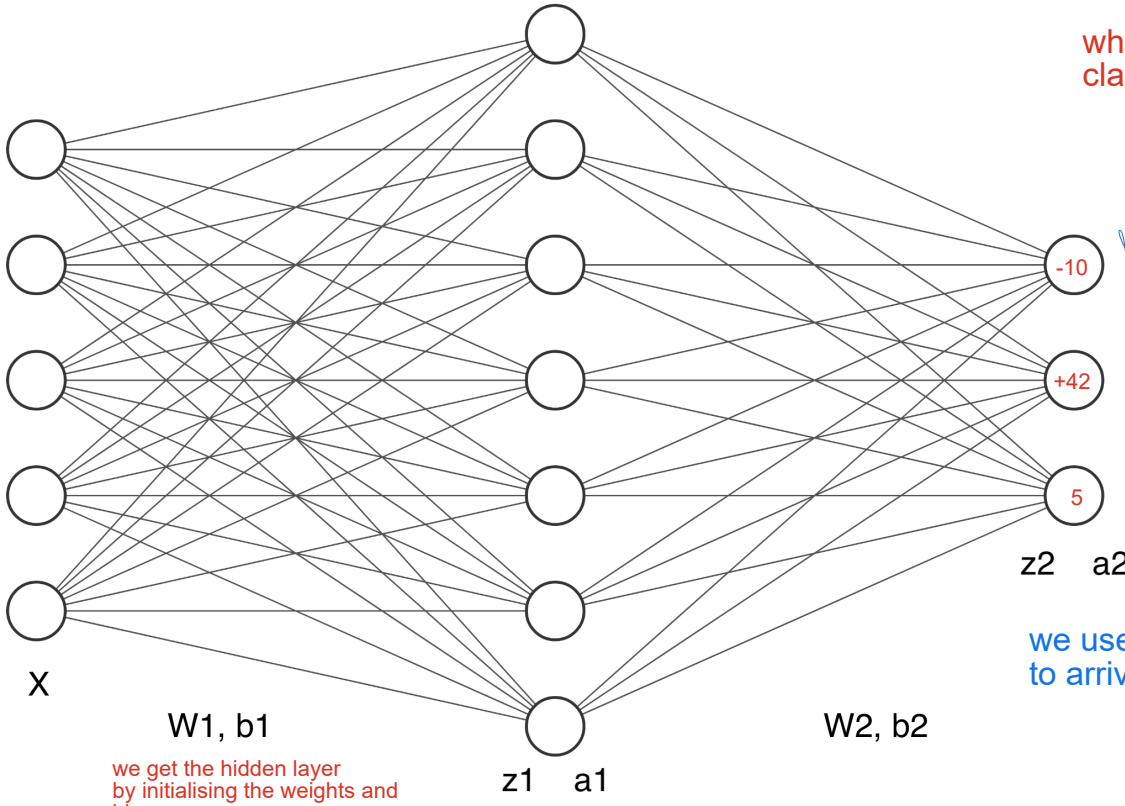
$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



Some common activation functions. Image from Jadon (2018)

Classification MLP with one hidden layer

adding hidden layers allows to get a better approximator for less linear functions



when will we have multiple output neurons?

when we have multiclass classification

How do we transform these numbers into probabilities?

exp(-10)/ exp(-10)+exp(42)+exp(5)

we wrap in exponential because we want to have positive numbers

we use softmax function to arrive at probabilities

Drawn with `http://alexlenail.me/NN-SVG/index.html`

In more detail

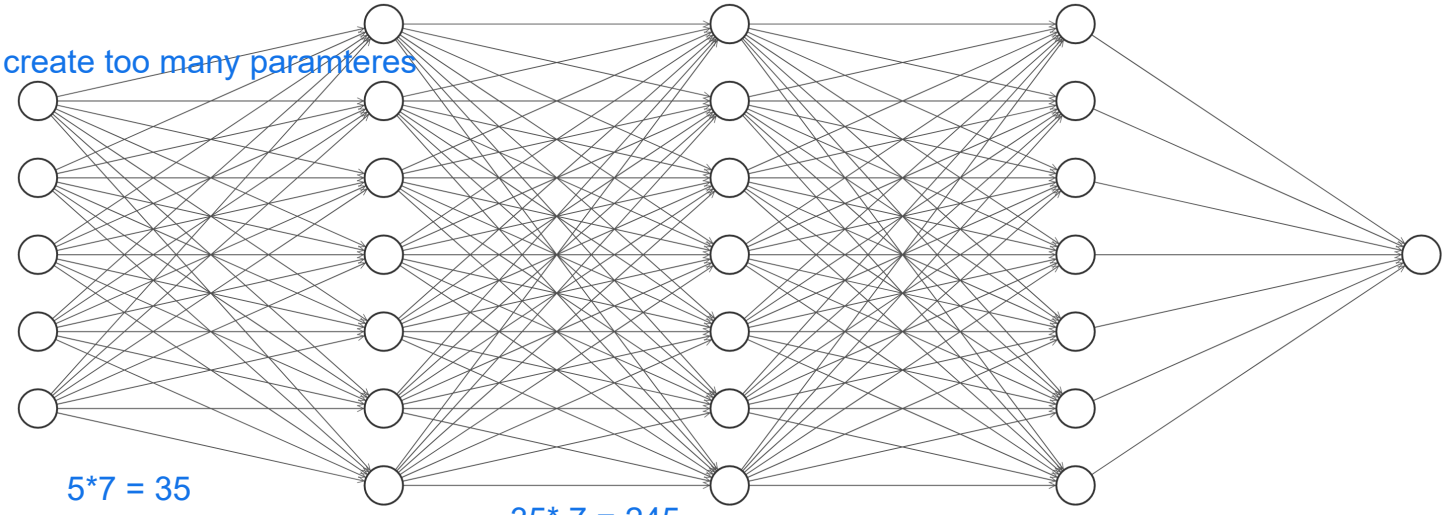
Forward pass expressed in one line: $$x \underbrace{\rightarrow}_{\cdot W^{(1)},\ +b^{(1)}} z^{(1)} \underbrace{\rightarrow}_{g(\cdot)} a^{(1)} \underbrace{\rightarrow}_{\cdot W^{(2)},\ +b^{(2)}} z^{(2)} \underbrace{\rightarrow}_{s(\cdot)} a^{(2)} = p$$

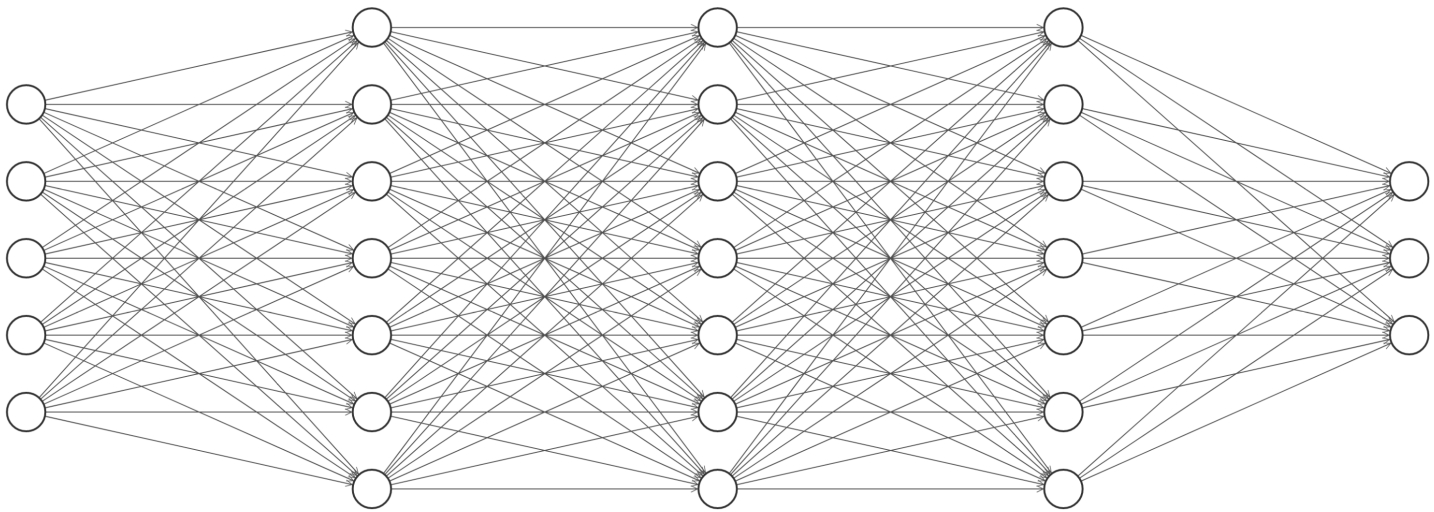
- Input layer: $\underbrace{x}_{5\times 1}$
- Hidden layer z: $\underbrace{W^{(1)}}_{7\times 5}\underbrace{x}_{5\times 1} + \underbrace{b^{(1)}}_{7\times 1} = \underbrace{z^{(1)}}_{7\times 1}$
- Hidden layer activation: $\underbrace{a^{(1)}}_{7\times 1} = \underbrace{g(z^{(1)})}_{7\times 1}$ (e.g. ReLU applied element-wise with $g(z_i) = max\{0, z_i\}$)
- Output layer z: $\underbrace{W^{(2)}}_{3\times 7}\underbrace{a^{(1)}}_{7\times 1} + \underbrace{b^{(2)}}_{3\times 1} = \underbrace{z^{(2)}}_{3\times 1}$
- Output activation: $\underbrace{s(z^{(2)})}_{3\times 1} = \underbrace{a^{(2)}}_{3\times 1} = \underbrace{p}_{3\times 1}$ (softmax function applied element-wise $s(z)_i = \frac{\exp(z_i)}{\sum_{j=1}^{K}\exp(z_j)}$)

Exemplary MLPs with more hidden layers

Why are neural networks prone to over fit?

Because we create too many paramteres





Drawn with `http://alexlenail.me/NN-SVG/index.html`

Discussion

- Conceptually, the neural networks discussed here are simply function approximators that fit the functional relationship between inputs X and outputs y, just like other regression and classification models
- Yet, adding further neurons, hidden layers, etc. allows neural networks to fit highly complex functions
- Activations are key as well, stacking only linear layers could only create a linear model

Universal approximation theorem

If you add enough neurons, you can approximate any kind of function

- Under some regularity conditions already a single hidden layer neural network can approximate continuous functions, that map from $[0, 1]^K$ to the real number line, arbitrarily closely (see Cybenko, 1989; Hornik et al., 1989; Hornik, 1991, for further details)
- Note that this says nothing about how we can practically find the optimal weights of such a network, just that it exists
- Empirically, networks with many hidden layer tend to find the better solutions

Training

- Weights and biases of a neural network are determined by minimising a loss function via gradient descent
- Commonly this loss function is mean squared error for regression or cross-entropy for classification

Loss function

- Mean-squared-error loss for one observation:

$$L(y_i, p_i) = (y_i - p_i)^2$$

- Cross-entropy loss for one observation:

$$L(y_i, p_i) = -\sum_{k=1}^{K} y_{i,k} \log(p_{i,k})$$

- Summarise all weights, $W^{(1)}, W^{(2)}, \ldots$ and biases $b^{(1)}, b^{(2)}, \ldots$ in one vector θ

- For full training dataset:

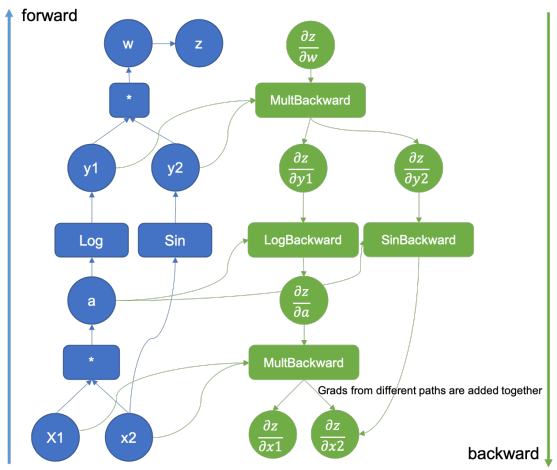
$$J(\theta; \{x_i, y_i\}_{i=1}^{n}) = \frac{1}{n} \sum_{i=1}^{n} L(y_i, p(x_i, \theta))$$

Gradient of loss

- Recall that the path through the network in the forward pass has the form: $x \underbrace{\rightarrow}_{W^{(1)},b^{(1)}} z^{(1)} \underbrace{\rightarrow}_{g(\cdot)} a^{(1)} \underbrace{\rightarrow}_{W^{(2)},b^{(2)}} z^{(2)} \underbrace{\rightarrow}_{g(\cdot)} a^{(2)} = p$
- Thus, applying the chain rule we get the gradient (assume scalars for simplicity here):
  - $\frac{\partial J(\theta)}{\partial W^{(2)}} = \frac{\partial J(\theta)}{\partial a^{(2)}} \frac{\partial a^{(2)}}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial W^{(2)}}$
  - $\frac{\partial J(\theta)}{\partial b^{(2)}} = \frac{\partial J(\theta)}{\partial a^{(2)}} \frac{\partial a^{(2)}}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial b^{(2)}}$
  - $\frac{\partial J(\theta)}{\partial W^{(1)}} = \frac{\partial J(\theta)}{\partial a^{(2)}} \frac{\partial a^{(2)}}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial a^{(1)}} \frac{\partial a^{(1)}}{\partial z^{(1)}} \frac{\partial z^{(1)}}{\partial W^{(1)}}$
  - $\frac{\partial J(\theta)}{\partial b^{(1)}} = \frac{\partial J(\theta)}{\partial a^{(2)}} \frac{\partial a^{(2)}}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial a^{(1)}} \frac{\partial a^{(1)}}{\partial z^{(1)}} \frac{\partial z^{(1)}}{\partial b^{(1)}}$
- Combine/stack to get gradient $\frac{\partial J(\theta)}{\partial \theta}$ or also often denoted $\nabla_\theta J(\theta)$

Computation graphs

- Modern libraries such as PyTorch and TensorFlow process networks as computation graphs
- The gradients they compute flow through the individual elements of that graph
- This modular structure allows to process gradients in a wide range of architectures



Example from PyTorch blog

Gradient descent

- Computing the gradient for the entire dataset (full batch gradient descent) is usually computationally too costly
- Instead, we typically approximate the gradient of the full training data with a small batch of data (mini batch gradient descent)
- Importantly, note that since loss functions for neural networks are mostly non-convex, these algorithms may converge to local optima, but typically not global ones

Mini-batch gradient descent

- Randomly initialise weights; select a learning rate α. Then repeat the following for E epochs or until approximate convergence:

  we need to shuffle because we only apply the neural network to a small portion of data (computers cannot handle the entire dataset in neural networks because of memory)

  1. Shuffle all observations in the training dataset and divide into $\lceil \frac{n}{B} \rceil$ batches
  2. For each batch with observations:
     - Update $\theta \leftarrow \theta - \alpha \frac{1}{B} \sum_{i=1}^{B} \nabla_\theta L\left(y_i, p(x_i, \theta)\right)$

  in multidimensional spaces we often add a 'momentum' term because these models tend to have a lot of saddle points and we do not want to be stuck on one so we add it to keep walking towards the minumum

- Commonly used current optimisers such as Adam additionally add features like momentum

Regularisation

- With their very high numbers of parameters, neural networks can memorise large amounts of data and over-fit relatively easy
- Possible approaches to counter over-fitting are:
  - Add L1 ($\sum |\theta_i|$) or L2 ($\sum \theta_i^2$) norms of parameters to the loss function this is like regularisation
  - Dropout: Randomly set a fraction of p neurons in a layer to zero during training and thereby force varying sets of neurons to learn patterns. In essence, this trains an ensemble
  - Stop the gradient descent once the loss on some validation set increases you stop training as soon as you stop improving preditctions on a new unseen dataset
  - Add noise to activities during training

To improve training, try for example:

- Normalisation over observations in a batch or activations in a layer
- Residual connections/skip connections: Skip some layers and allowing some information to flow more easily
- Often helpful to first fit the model to a very small sample of the data and see whether it can perfectly fit it. If not, there might be some issues

Conclusion

- To conclude, we will train a simple MLP network using the library `keras` in R for text classification
- In the following weeks, we will discuss more complex network architectures, such as transformers, which underlie current language models

Coding

- 03-neural-network-for-text-classification.qmd

Further study

If you are interested in the topics beyond the scope of this course, have a look at the following series on Grant Sanderson's 3Blue1Brown channel:

- Linear algebra: `https://www.youtube.com/watch?v=fNk_zzaMoSs&list=PLZHQObOWTQDPD3MizzM2xVFitgF8hE_ab`
- Calculus: `https://www.youtube.com/watch?v=WUvTyaaNkzM&list=PL0-GT3co4r2wlh6UHTUeQsrf3mlS2lk6x`
- Neural networks and gradient descent: `https://www.youtube.com/watch?v=aircAruvnKk&list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi`

For an extensive course on neural networks for text (in Python), see Andrej Karpathy's series here: `https://www.youtube.com/watch?v=VMj-3S1tku0&list=PLAqhIrjkxbuWI23v9cThsA9GvCAUhRvKZ`

References I

Cybenko, George, "Approximation by superpositions of a sigmoidal function," *Mathematics of control, signals and systems*, 1989, *2* (4), 303–314.

Hornik, Kurt, "Approximation capabilities of multilayer feedforward networks," *Neural networks*, 1991, *4* (2), 251–257.

__, Maxwell Stinchcombe, and Halbert White, "Multilayer feedforward networks are universal approximators," *Neural networks*, 1989, *2* (5), 359–366.

Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in neural information processing systems*, 2012, *25*.

McCulloch, Warren S and Walter Pitts, "A logical calculus of the ideas immanent in nervous activity," *The bulletin of mathematical biophysics*, 1943, *5*, 115–133.

Derivation of the simplified logistic loss

- Loss function:

$$L_i = -\left[y_i log\left(\frac{1}{1+\exp(-\alpha-\beta x_i)} \right) + (1-y_i) log\left(\frac{\exp(-\alpha-\beta x_i)}{1+\exp(-\alpha-\beta x_i)} \right) \right]$$

- Using $log(1) = 0$, $log(a^n) = nlog(a)$, and $log(\frac{a}{b}) = log(a) - log(b)$, this simplifies to:

$$L_i = -\left[-y_i log(1+\exp(-\alpha-\beta x_i)) + (1-y_i)\left(-\alpha-\beta x_i - log(1+\exp(-\alpha-\beta x_i))\right) \right]$$

$$L_i = y_i log(1+\exp(-\alpha-\beta x_i)) + (1-y_i)\left(\alpha+\beta x_i + log(1+\exp(-\alpha-\beta x_i))\right)$$

$$L_i = (1-y_i)(\alpha+\beta x_i) + log(1+\exp(-\alpha-\beta x_i))$$

Back