Deep Learning II

Mitko Veta

Eindhoven University of Technology
Department of Biomedical Engineering

2024

Learning goals

- **Training neural networks:** understand and apply optimization algorithms for neural network training.
- **Regularisation of neural networks:** Understand and implement regularisation techniques for neural network models.
- **Other types of neural networks:** Gain an overview of advanced neural network architectures and understand their general structure and typical application.

Material

- Chapter 10.7 form "*An introduction to statistical learning with applications in Python"*, G. James, D. Witten, T. Hastie, R. Tibshirani, J. Taylor"
- Chapters 6.5 (backpropagation), 7.11, 7.12 (dropout), 8.3.1, 8.3.2 (SGD with momentum), 8.7 (batch normalisation) from deeplearningbook.org

Overview

Topics covered in this lecture

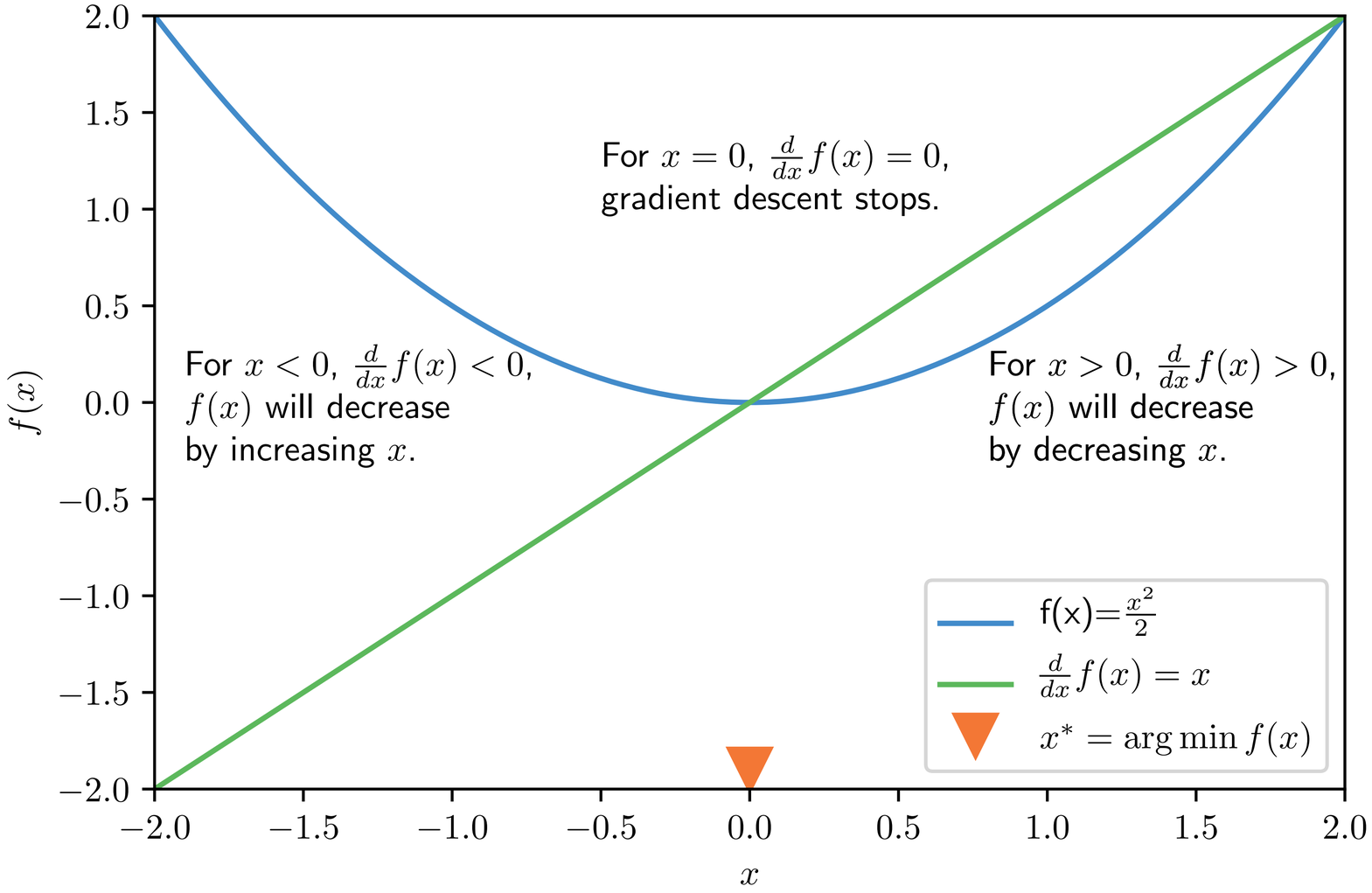
- Gradient-based training (optimisation) of neural networks: stochastic gradient descent and variants
- Overfitting and regularisation in the context of neural networks: dropout and batch-normalisation
- The backpropagation algorithm
- Very brief overview of other types of neural networks: autoencoders, CNNs, RNNs, transformers (not in the exam)


Steep slope,
take large steps
Less steep slope,
take smaller steps
Goal

Question

Where does the information about the slope come from?

Gradient informs optimisation




{X, y}
“Move slightly a bit to the right
to go slightly downhill”

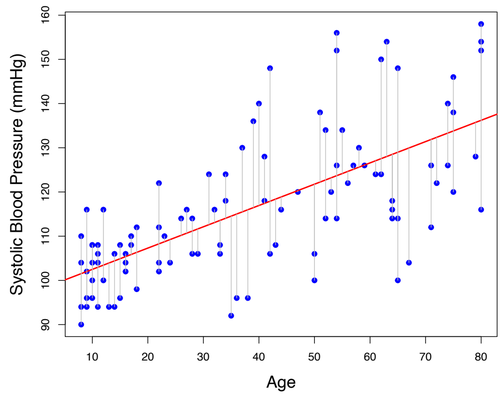
Gradient descent in vector form

$$\beta_0^{(t+1)} = \hat{\beta}_0^{(t)} - \rho \frac{\partial RSS}{\partial \beta_0}$$

$$\beta_1^{(t+1)} = \hat{\beta}_1^{(t)} - \rho \frac{\partial RSS}{\partial \beta_1}$$

$$\begin{pmatrix} \beta_0^{(t+1)} \\ \beta_1^{(t+1)} \end{pmatrix} = \begin{pmatrix} \hat{\beta}_0^{(t)} \\ \hat{\beta}_1^{(t)} \end{pmatrix} - \rho \begin{pmatrix} \frac{\partial RSS}{\partial \beta_0} \\ \frac{\partial RSS}{\partial \beta_1} \end{pmatrix}$$

$$\boldsymbol{\beta}^{(t+1)} = \hat{\boldsymbol{\beta}}^{(t)} - \rho \nabla_{\boldsymbol{\beta}} RSS$$



$$RSS = (y_1 - \hat{\beta}_0 - \hat{\beta}_1 x_1)^2 + (y_2 - \hat{\beta}_0 - \hat{\beta}_1 x_2)^2 + \ldots + (y_n - \hat{\beta}_0 - \hat{\beta}_1 x_n)^2$$

Standard gradient descent

In standard gradient descent, the gradient is computed using **all data points** in the dataset at every iteration.

- Start with an initial guess for the parameters
- For each iteration, compute the gradient of the loss over the entire dataset
- Update the parameters using the learning rate
- **Pros**:
  - Converges steadily to the global minimum for convex functions.
  - Uses the full dataset for accurate gradient estimates.
- **Cons**:
  - Computationally expensive for large datasets since the gradient is computed using all data points.
  - Slower per iteration for large-scale problems.

$$\boldsymbol{\beta}^{(t+1)} = \hat{\boldsymbol{\beta}}^{(t)} - \rho \nabla_{\boldsymbol{\beta}} RSS$$

How can we speed-up gradient descent?

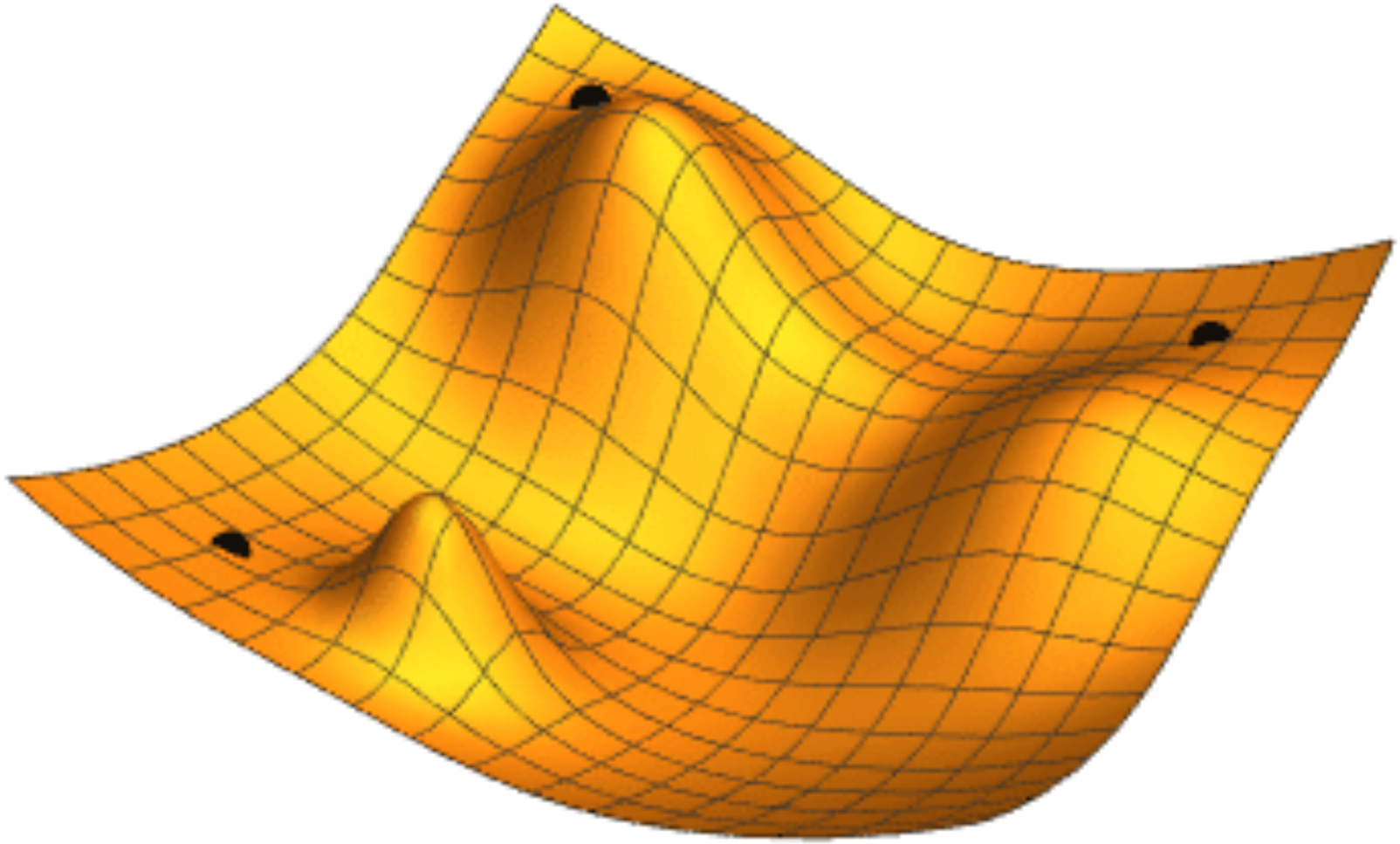
Stochastic gradient descent (SGD)

In SGD, the gradient is estimated using a **mini-batch or even a single data point**, rather than the entire dataset. This introduces noise in the gradient estimation, but it allows for much faster updates, particularly with large datasets.

- Start with an initial guess for the parameters
- For each iteration, select mini-batch B_t from the dataset
- Compute the gradient of the loss over the mini-batch
- Update the parameters using the learning rate
- **Pros**:
  - Much faster updates, especially for large datasets.
  - Can escape local minima due to noise introduced by mini-batches.
- **Cons**:
  - The gradient is noisier, which can cause fluctuations around the minimum.
  - Requires careful tuning of the learning rate to avoid overshooting the minimum.

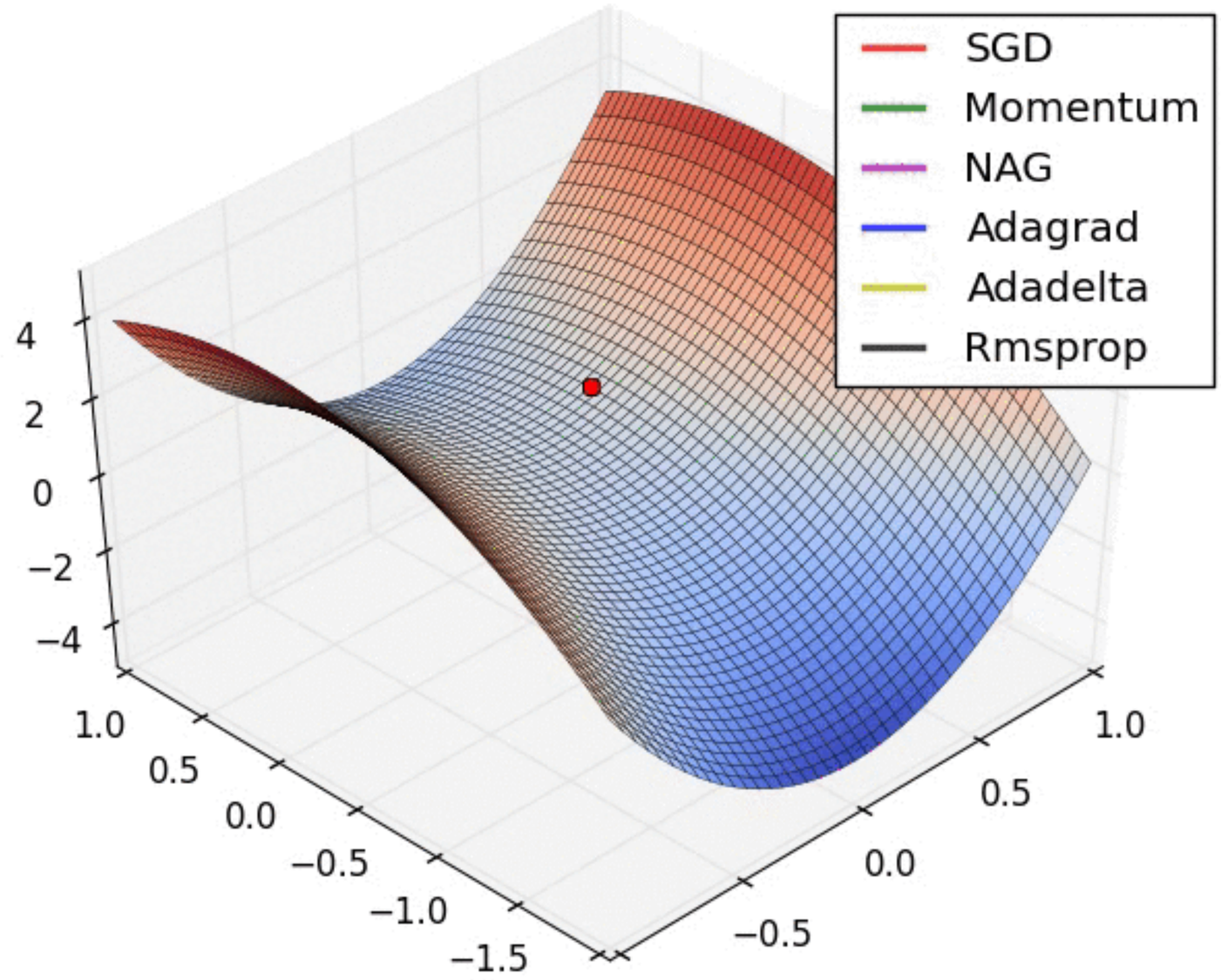
$$\boldsymbol{\beta}^{(t+1)} = \hat{\boldsymbol{\beta}}^{(t)} - \rho \nabla_{\boldsymbol{\beta}} RSS_{B_t}$$

Optimisation landscape of neural networks



Source: https://g-stat.com/optimization-gradients-overview/

SGD variants



SGD with momentum

Key idea: momentum accumulates an exponentially weighted average of the past gradients to maintain consistent movement in a direction, thereby speeding up convergence.

- Momentum term: introduce a momentum variable $\boldsymbol{v}$ that stores the velocity (accumulated gradient)
- Update: update the momentum $\boldsymbol{v}$ and the parameters $\boldsymbol{\beta}$ based on the velocity and the gradient
- γ is the momentum coefficient, typically set between 0.5 and 0.9

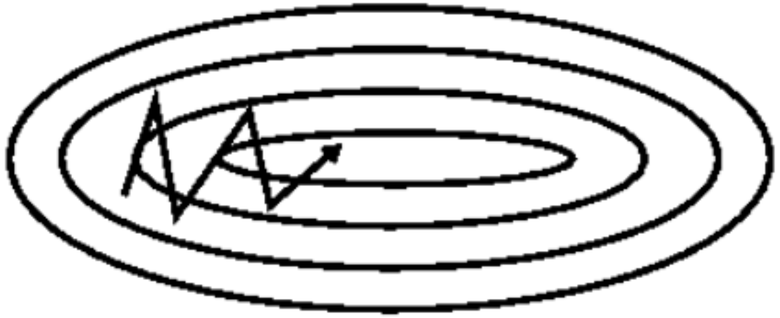
$$\boldsymbol{\beta}^{(t+1)} = \hat{\boldsymbol{\beta}}^{(t)} - \rho \nabla_{\boldsymbol{\beta}} RSS$$

$$\boldsymbol{v}^{(t+1)} = \gamma \boldsymbol{v}^{(t)} + \rho \nabla_{\boldsymbol{\beta}} RSS_{B_t}$$

$$\boldsymbol{\beta}^{(t+1)} = \boldsymbol{\beta}^{(t)} - \boldsymbol{v}^{(t+1)}$$

SGD with momentum

(a) SGD without momentum



(b) SGD with momentum

$$\boldsymbol{\beta}^{(t+1)} = \hat{\boldsymbol{\beta}}^{(t)} - \rho \nabla_{\boldsymbol{\beta}} RSS$$

$$\boldsymbol{v}^{(t+1)} = \gamma \boldsymbol{v}^{(t)} + \rho \nabla_{\boldsymbol{\beta}} RSS_{B_t}$$

$$\boldsymbol{\beta}^{(t+1)} = \boldsymbol{\beta}^{(t)} - \boldsymbol{v}^{(t+1)}$$

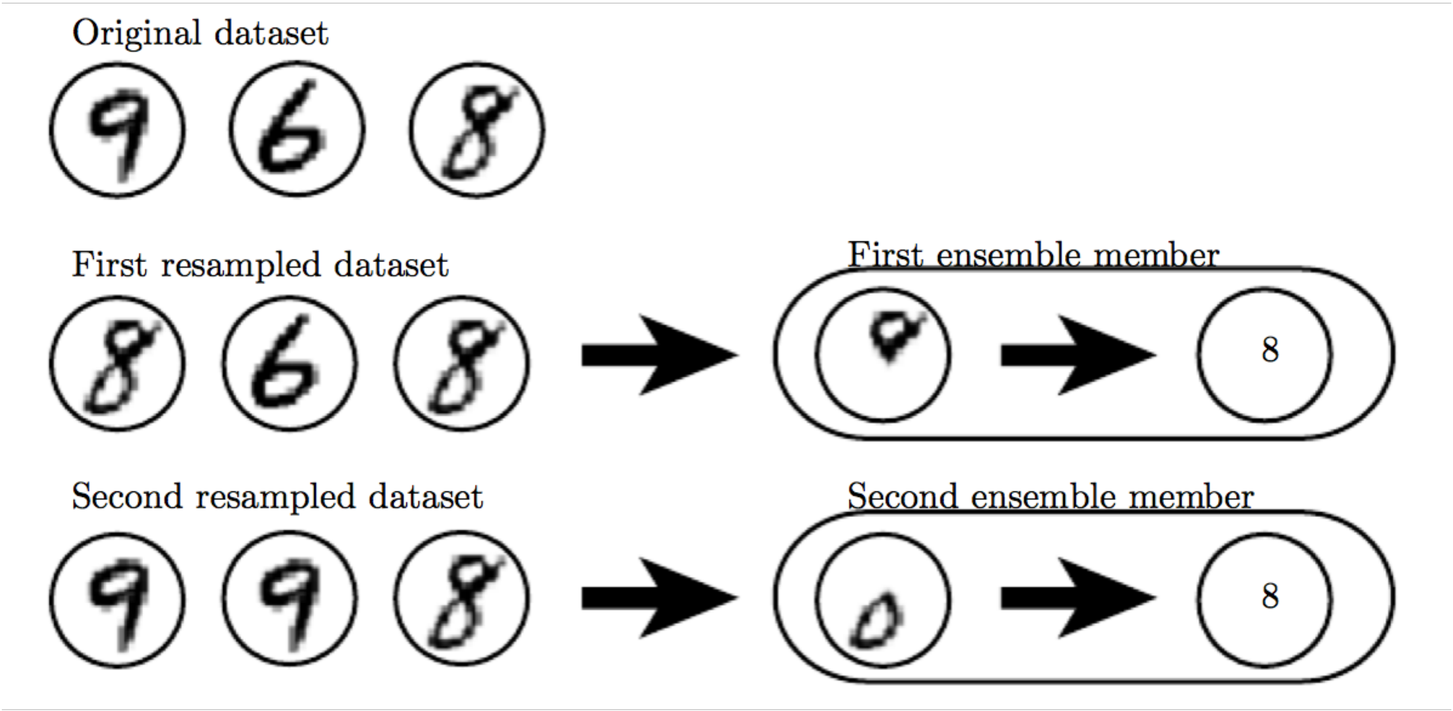
Source: https://g-stat.com/optimization-gradients-overview/

Regularisation of neural networks

- L1 and L2 regularization are applied similarly in neural networks as in linear/ logistic regression, but now act on the network weights.
  - In linear/logistic regression, regularization controls the coefficients of features.
  - In neural networks, regularization controls the weights of connections between neurons, which can be vast due to the network's architecture.
- L2 is more commonly used in neural networks to keep weights small and prevent overfitting
- L1 is used when we want to encourage sparse weights, potentially removing some connections entirely.

Model averaging

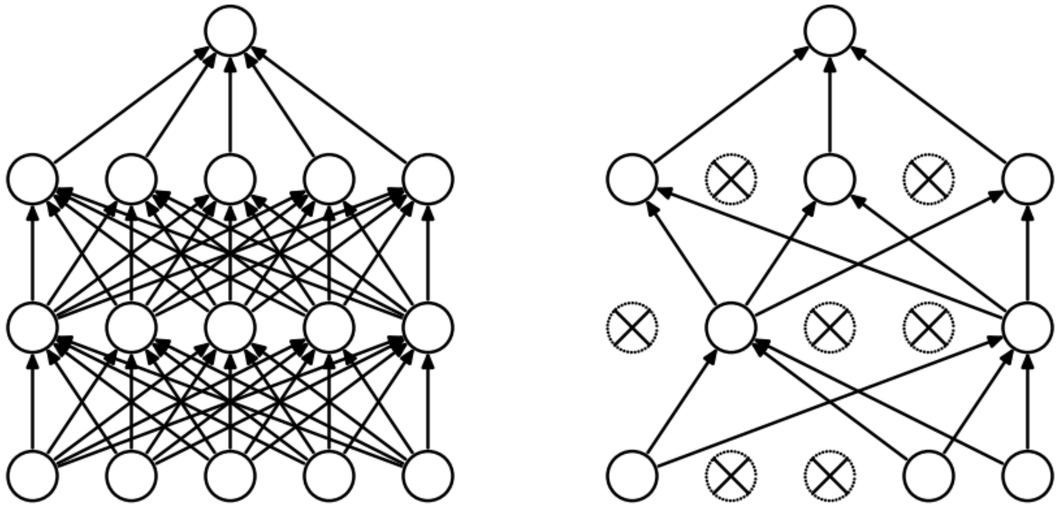
Rationale: different models make different errors.



Source: http://deeplearningbook.org

Dropout

- Dropout performs implicit model averaging
- It works by randomly turning off connections in the neural network
- For every layer we define a parameter p that corresponds to the probability of a neuron being “turned off”



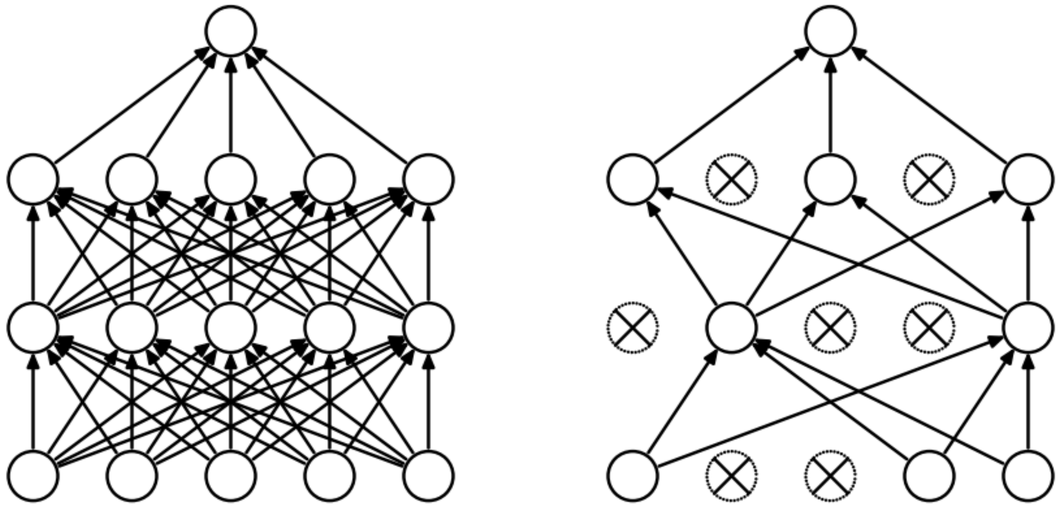
(a) Standard Neural Net

(b) After applying dropout.

Source: Srivastava et al. JMLR 2014

Dropout as noise

- Another way to view dropout: adding noise in the connections of the neural network



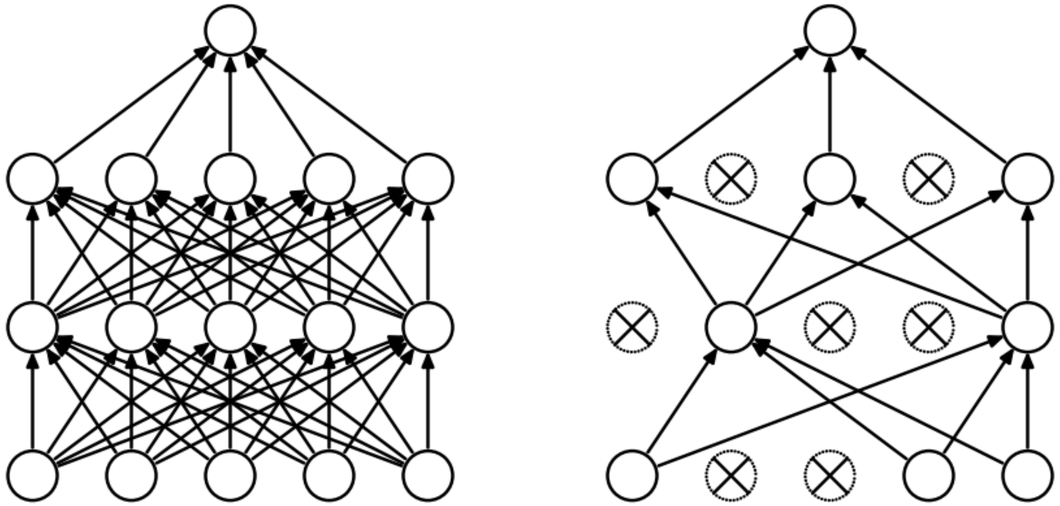
(a) Standard Neural Net

(b) After applying dropout.

Source: Srivastava et al. JMLR 2014

Question

What do you estimate p to be in this case?



(a) Standard Neural Net

(b) After applying dropout.

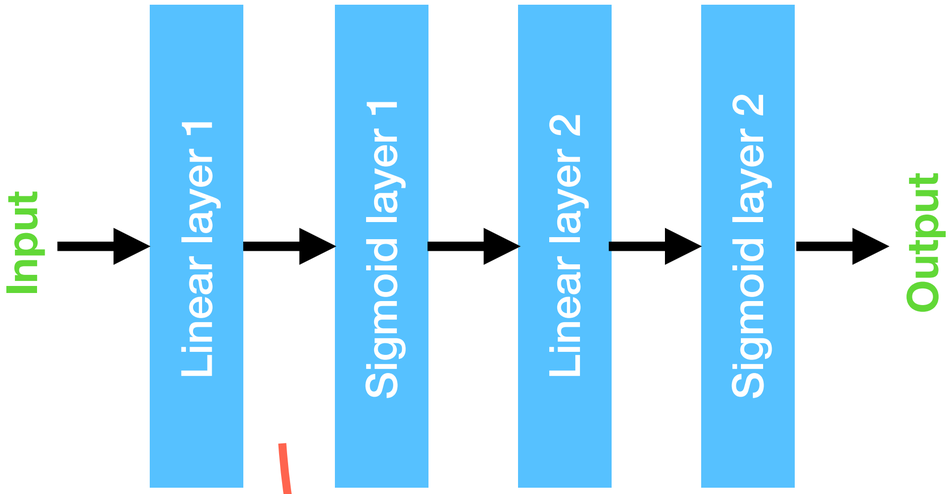
Source: Srivastava et al. JMLR 2014

Dropout at test time

- At test time, all nurons are active (i.e. we do not apply dropout)
- Since the model was trained with dropout (some neurons being inactive), we need to adjust the output at test time to account for the missing neurons during training
- Activations are scaled by $(1 - p)$ to ensure that the expected activation at test time is the same as during training

Batch normalisation

A technique to normalise the inputs to each layer in a neural network.



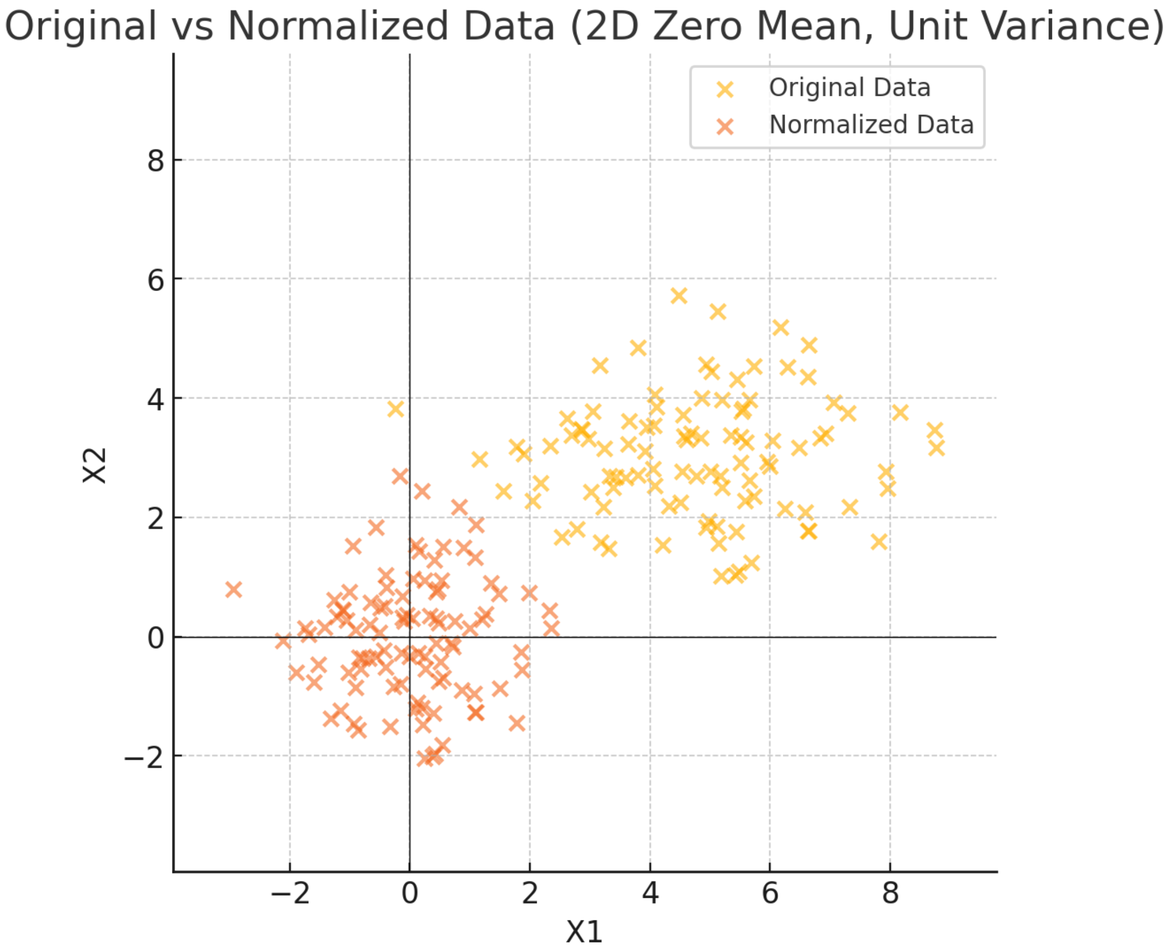
$$X = \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1n} \\ x_{21} & x_{22} & \cdots & x_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{m1} & x_{m2} & \cdots & x_{mn} \end{bmatrix}$$

$$\hat{x}_{ij} = \frac{x_{ij} - \mu_j}{\sigma_j}, \text{ where}$$

$$\mu_j = \frac{1}{m}\sum_{i=1}^{m} x_{ij} \quad \text{and} \quad \sigma_j = \sqrt{\frac{1}{m}\sum_{i=1}^{m}(x_{ij} - \mu_j)^2}$$

For each mini-batch

Batch normalisation



$$\hat{x}_{ij} = \frac{x_{ij} - \mu_j}{\sigma_j}$$

Benefits of batch normalisation

- Main goal: to stabilise and speed up training by maintaining the distribution of activations throughout the network.
- Bonus: it adds noise to the representation of each sample
  - This is because the mini-batches are sampled randomly
  - The feature representation for each sample from each layer with batch-norm will depend on which other samples are included in the batch
- Because of this, batch normalisation acts as regularisation

Learning representations by back-propagating errors

**David E. Rumelhart*, Geoffrey E. Hinton†
& Ronald J. Williams***

* Institute for Cognitive Science, C-015, University of California, San Diego, La Jolla, California 92093, USA
† Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Philadelphia 15213, USA

We describe a new learning procedure, back-propagation, for networks of neurone-like units. The procedure repeatedly adjusts the weights of the connections in the network so as to minimize a measure of the difference between the actual output vector of the net and the desired output vector. As a result of the weight adjustments, internal 'hidden' units which are not part of the input or output come to represent important features of the task domain, and the regularities in the task are captured by the interactions of these units. The ability to create useful new features distinguishes back-propagation from earlier, simpler methods such as the perceptron-convergence procedure[1].

There have been many attempts to design self-organizing neural networks. The aim is to find a powerful synaptic modification rule that will allow an arbitrarily connected neural network to develop an internal structure that is appropriate for a particular task domain. The task is specified by giving the desired state vector of the output units for each state vector of the input units. If the input units are directly connected to the output units it is relatively easy to find learning rules that iteratively adjust the relative strengths of the connections so as to progressively reduce the difference between the actual and desired output vectors[2]. Learning becomes more interesting but

† To whom correspondence should be addressed.

'Godfather of AI' shares Nobel Physics Prize

1 day ago

Save +

Georgina Rannard
Science reporter

Graham Fraser
Technology reporter

The announcement was made in Stockholm, Sweden

The Nobel Prize in Physics has been awarded to two scientists, Geoffrey Hinton and John Hopfield, for their work on machine learning.

British-Canadian Professor Hinton is sometimes referred to as the "Godfather of AI" and said he was flabbergasted.

The chain rule of differentiation

- Applies to composition of functions: e.g. $z = f(y), y = g(x), z = f(g(x))$
  - For example if $f(x) = sin(x)$ and $g(x) = x^2$, then $y = sin(x^2)$
  - If we know the derivatives of $sin(x)$ and x^2, with the chain rule we can compute the derivative of $sin(x^2)$.

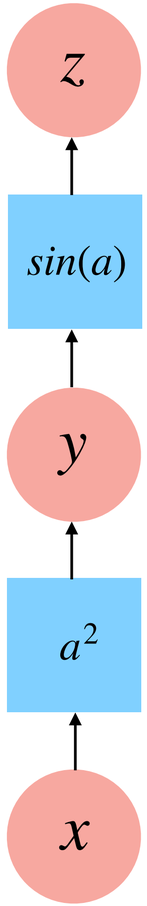
The chain rule of differentiation: scalars

$$z = f(y), \quad y = g(x), \quad z = f(g(x))$$

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

$$z = sin(y) \qquad \frac{dz}{dy} = cos(y)$$

$$y = x^2 \qquad \frac{dy}{dx} = 2x$$



The chain rule of differentiation: example

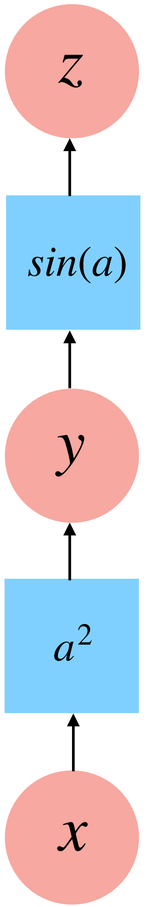
$$\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx}$$

$$\frac{dz}{dx} = cos(y) \cdot 2x$$

$$\frac{dz}{dx} = cos(x^2) \cdot 2x$$

$$z = sin(y)$$

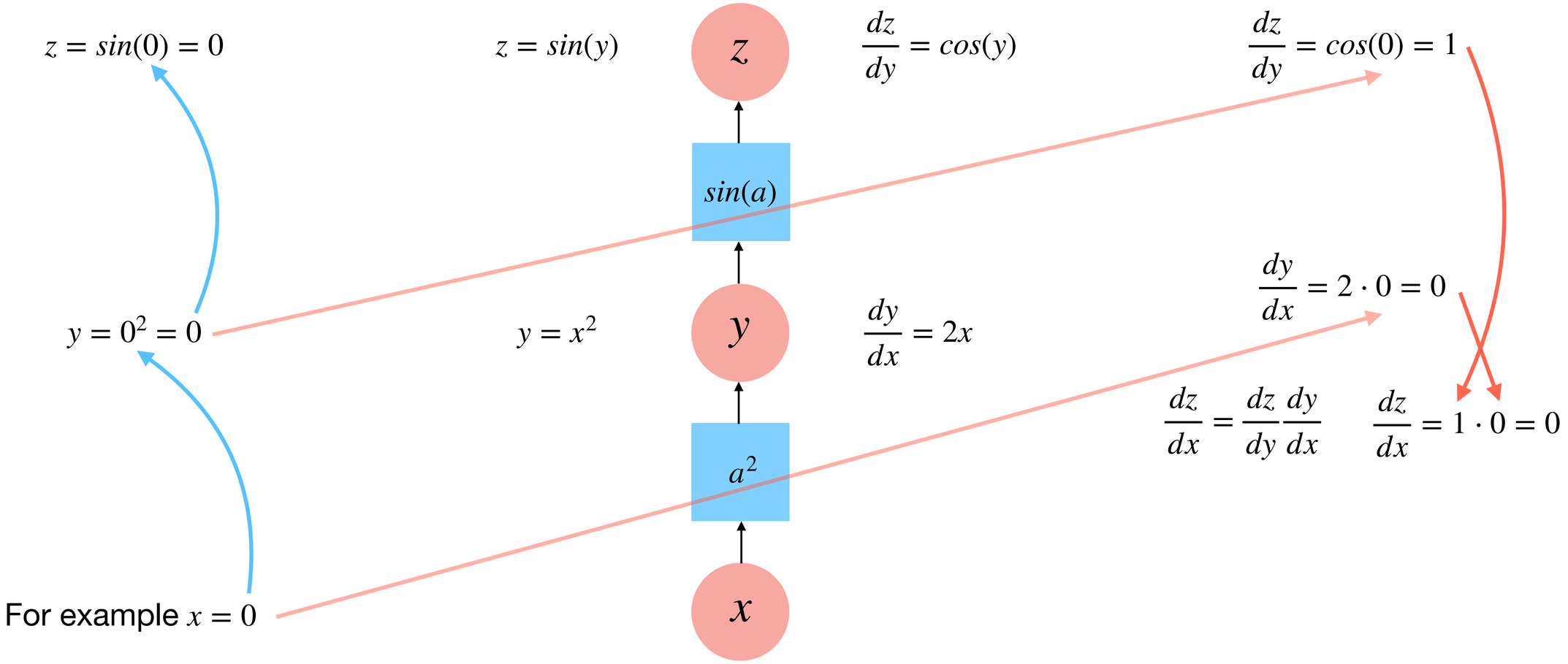
$$y = x^2$$



$$\frac{dz}{dy} = cos(y)$$

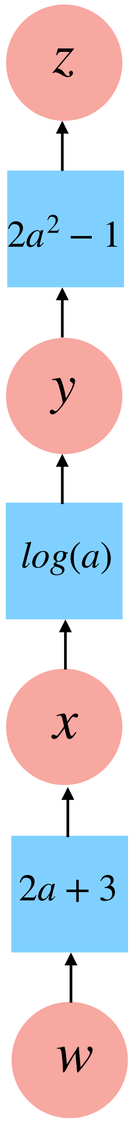
$$\frac{dy}{dx} = 2x$$

Computation steps



Forward computation

Computation steps: another example



$$z(y) = 2y^2 - 1 \qquad \frac{dz}{dy} = 4y$$

$$y(x) = \log(x) \qquad \frac{dy}{dx} = \frac{1}{x}$$

$$x(w) = 2w + 3 \qquad \frac{dx}{dw} = 2$$

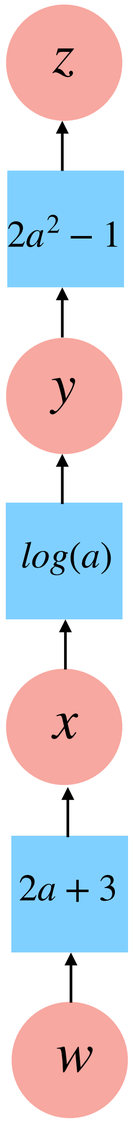
Option 1: (re)compute when needed

$$\frac{dz}{dw} = \frac{d}{dy} f(f(f(w))) \cdot \frac{d}{dx} f(f(w)) \cdot \frac{d}{dw} f(w)$$

Option 2: store in memory

$$\frac{dz}{dw} = \frac{dz}{dy} \cdot \frac{dy}{dx} \cdot \frac{dx}{dw}$$

Computation steps: forward computation



We know how to compute/derive:

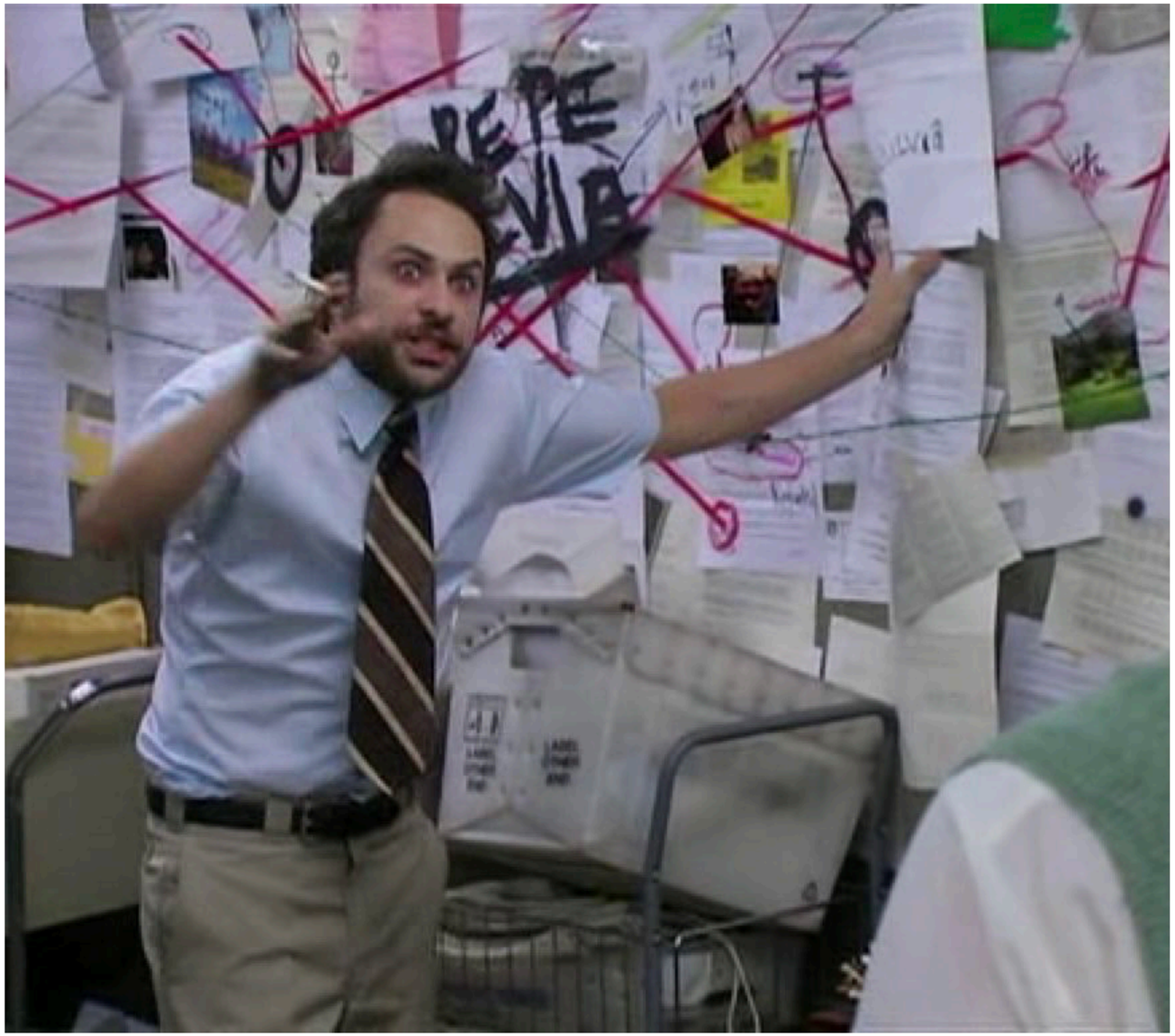
$$z(y) = 2y^2 - 1 \qquad \frac{dz}{dy} = 4y \qquad z(1.609) = 2(1.609)^2 - 1 = 5.176 - 1 = 4.176$$

$$y(x) = \log(x) \qquad \frac{dy}{dx} = \frac{1}{x} \qquad y(5) = log(5) \approx 1.609$$

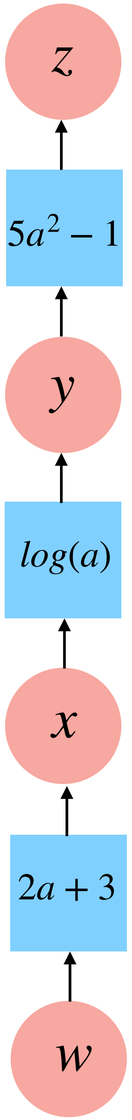
$$x(w) = 2w + 3 \qquad \frac{dx}{dw} = 2 \qquad x(1) = 2(1) + 3 = 5$$

$$w = 1$$

WARNING



Computation steps: backward computation



We know how to compute/derive:

$$z(y) = 2y^2 - 1 \qquad \frac{dz}{dy} = 4y$$

$$y(x) = \log(x) \qquad \frac{dy}{dx} = \frac{1}{x}$$

$$x(w) = 2w + 3 \qquad \frac{dx}{dw} = 2$$

Stored in memory from forward:

$$z(1.609) = 4.176$$

$$y(5) \approx 1.609$$

$$x(1) = 5$$

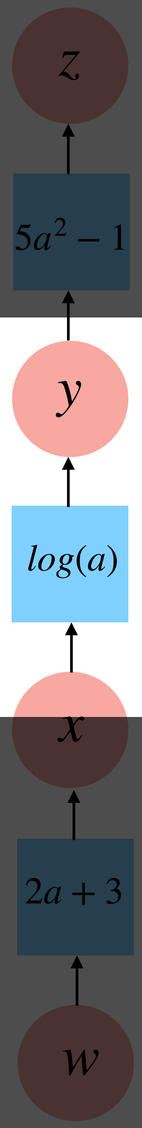
$$w = 1$$

$$\frac{dz}{dy} = \delta_z = 4(1.609) \approx 6.436$$

$$\frac{dz}{dx} = \delta_y = \delta_z \cdot \frac{\partial y}{\partial x} = 6.436 \cdot 0.2 = 1.287$$

$$\frac{dz}{dw} = \delta_x = \delta_y \cdot \frac{\partial x}{\partial w} = 1.287 \cdot 2 = 2.574$$

Computation steps: backward computation



We know how to compute/derive:

Stored in memory from forward:

$z(y) = 2y^2 - 1$ $\quad \frac{dz}{dy} = 4y$ $\quad z(1.609) = 4.176$ $\quad \frac{dz}{dy} = \delta_z = 4(1.609) \approx 6.436$

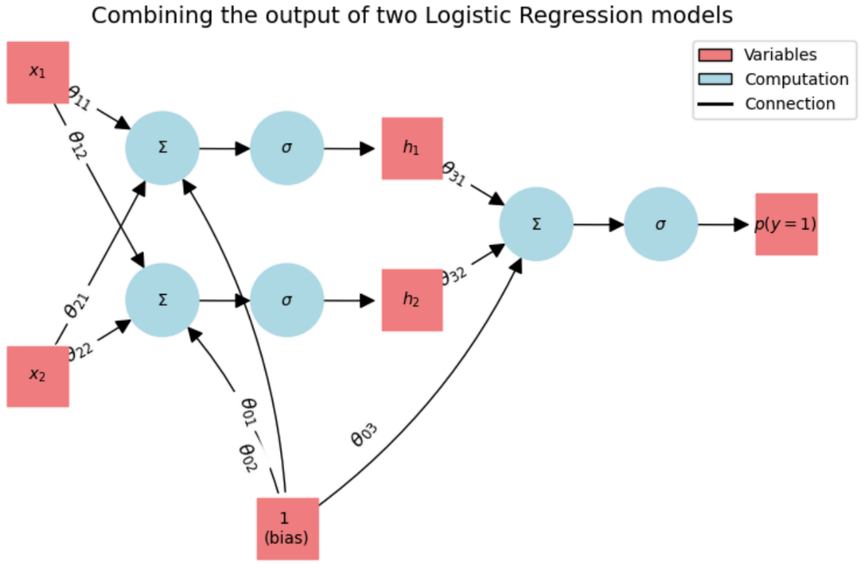
$y(x) = \log(x)$ $\quad \frac{dy}{dx} = \frac{1}{x}$ $\quad y(5) \approx 1.609$ $\quad \frac{dz}{dx} = \delta_y = \delta_z \cdot \frac{\partial y}{\partial x} = 6.436 \cdot 0.2 = 1.287$

$x(w) = 2w + 3$ $\quad \frac{dx}{dw} = 2$ $\quad x(1) = 5$ $\quad \frac{dz}{dw} = \delta_x = \delta_y \cdot \frac{\partial x}{\partial w} = 1.287 \cdot 2 = 2.574$

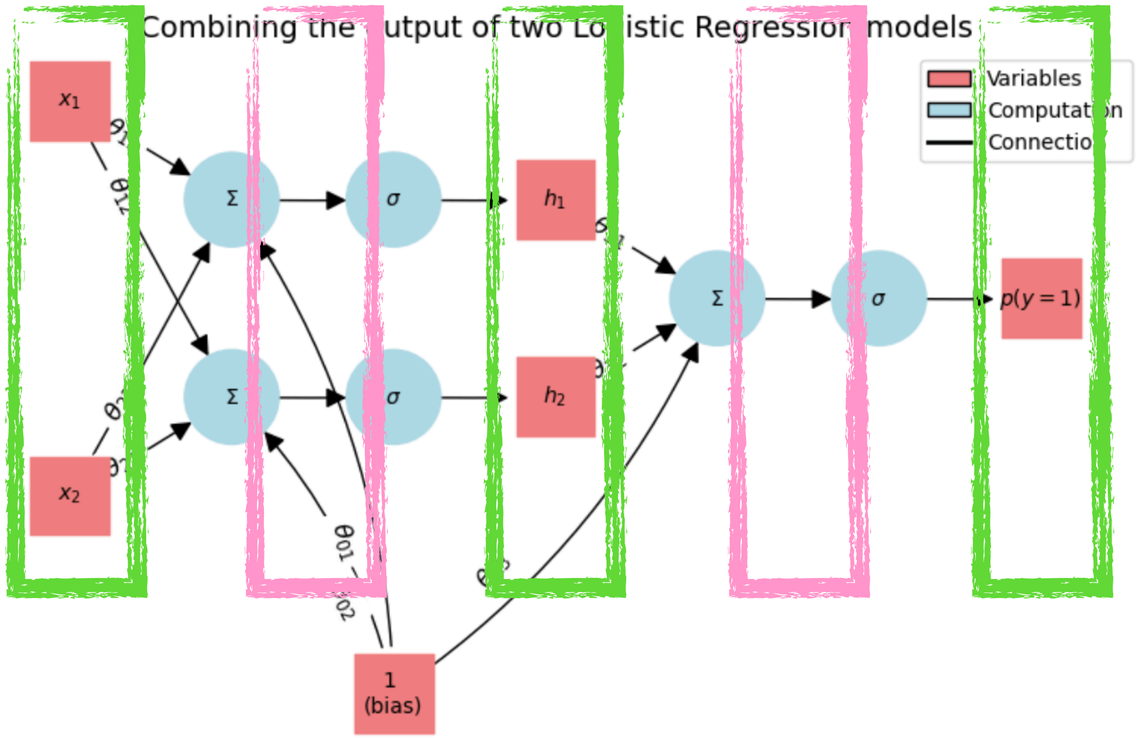
$w = 1$

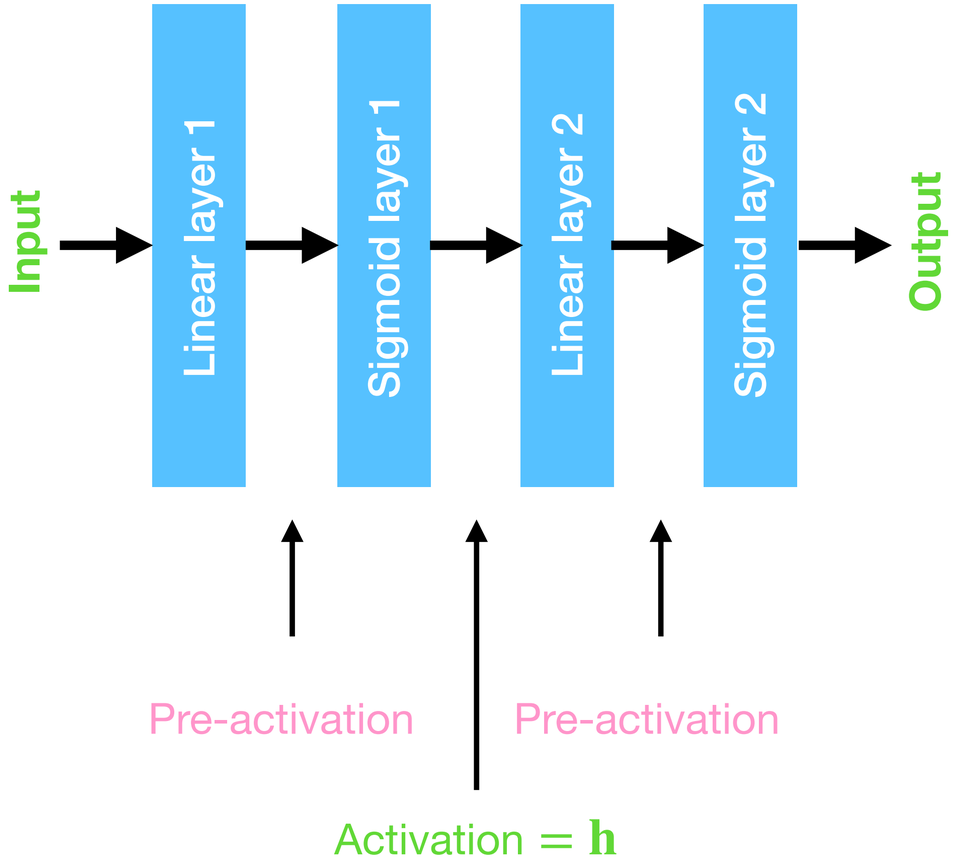
Question

What is the dimensionality of the inputs and outputs for the neural network below and each layer within the neural network?



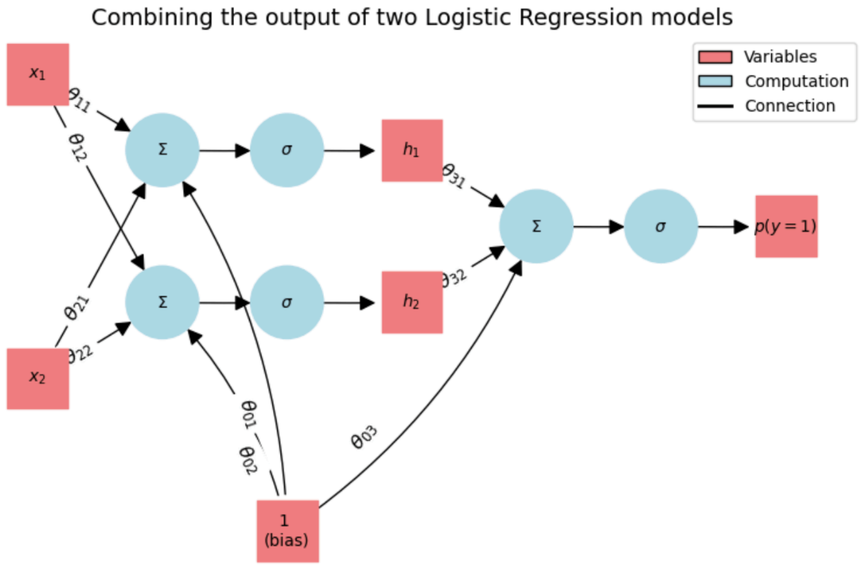
The inputs and outputs of NN layers





Question

Do all layers have parameters?



The chain rule of differentiation: vectors

$$\mathbf{z} = f(\mathbf{y}), \quad \mathbf{y} = g(\mathbf{x}), \quad \mathbf{z} = f(g(\mathbf{x}))$$

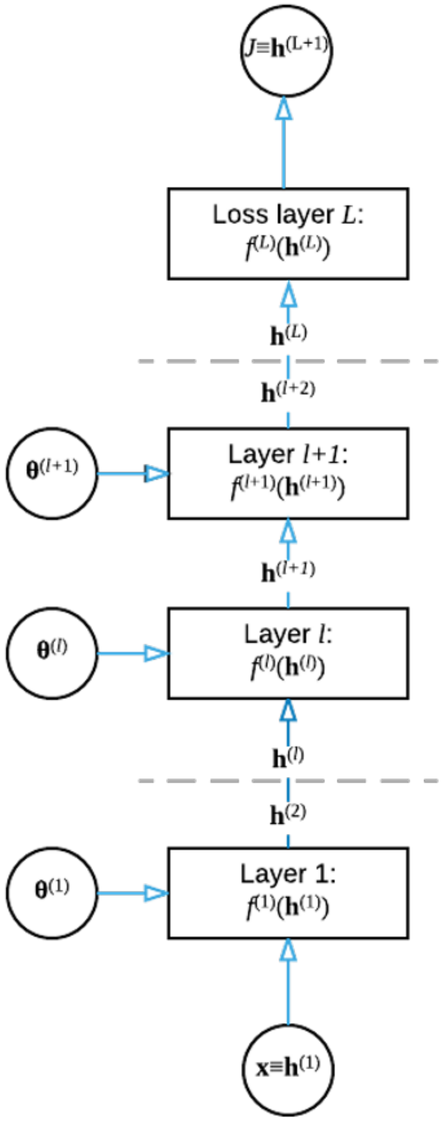
$$\frac{d\mathbf{z}}{d\mathbf{x}} = \frac{d\mathbf{z}}{d\mathbf{y}} \cdot \frac{d\mathbf{y}}{d\mathbf{x}}$$

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_p \end{bmatrix}, \quad \mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}, \quad \mathbf{z} = \begin{bmatrix} z_1 \\ z_2 \\ \vdots \\ z_m \end{bmatrix}$$

$$\frac{d\mathbf{z}}{d\mathbf{y}} = \begin{bmatrix} \frac{\partial z_1}{\partial y_1} & \frac{\partial z_1}{\partial y_2} & \cdots & \frac{\partial z_1}{\partial y_n} \\ \frac{\partial z_2}{\partial y_1} & \frac{\partial z_2}{\partial y_2} & \cdots & \frac{\partial z_2}{\partial y_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial z_m}{\partial y_1} & \frac{\partial z_m}{\partial y_2} & \cdots & \frac{\partial z_m}{\partial y_n} \end{bmatrix} \quad \frac{d\mathbf{y}}{d\mathbf{x}} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_1}{\partial x_2} & \cdots & \frac{\partial y_1}{\partial x_p} \\ \frac{\partial y_2}{\partial x_1} & \frac{\partial y_2}{\partial x_2} & \cdots & \frac{\partial y_2}{\partial x_p} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial y_n}{\partial x_1} & \frac{\partial y_n}{\partial x_2} & \cdots & \frac{\partial y_n}{\partial x_p} \end{bmatrix}$$

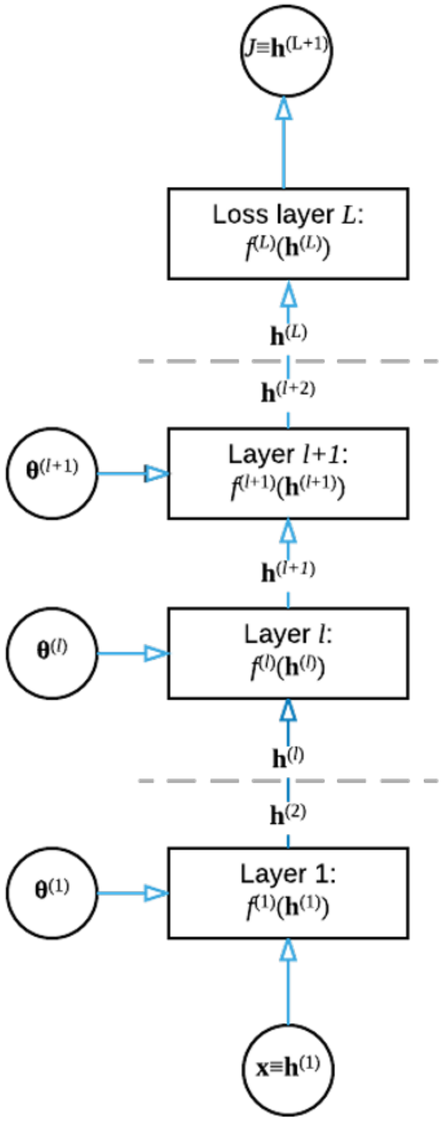
$$\frac{d\mathbf{z}}{d\mathbf{x}} = \begin{bmatrix} \frac{\partial z_1}{\partial y_1} & \frac{\partial z_1}{\partial y_2} & \cdots & \frac{\partial z_1}{\partial y_n} \\ \frac{\partial z_2}{\partial y_1} & \frac{\partial z_2}{\partial y_2} & \cdots & \frac{\partial z_2}{\partial y_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial z_m}{\partial y_1} & \frac{\partial z_m}{\partial y_2} & \cdots & \frac{\partial z_m}{\partial y_n} \end{bmatrix} \cdot \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_1}{\partial x_2} & \cdots & \frac{\partial y_1}{\partial x_p} \\ \frac{\partial y_2}{\partial x_1} & \frac{\partial y_2}{\partial x_2} & \cdots & \frac{\partial y_2}{\partial x_p} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial y_n}{\partial x_1} & \frac{\partial y_n}{\partial x_2} & \cdots & \frac{\partial y_n}{\partial x_p} \end{bmatrix}$$

The back propagation algorithm: setup



- We can make the computation of the loss part of our neural network graph:
  - Input: data $\mathbf{x}$ (vector)
  - Output: loss J (scalar)
- Each layer l has:
  - Input: $\mathbf{h}^{(l)}$(vector)
  - Output: $\mathbf{h}^{(\mathbf{l+1})}$ (vector)
  - Parameters: $\boldsymbol{\Theta}^{(l)}$ (vector)
  - Note that we represent the parameters of the layer also as inputs

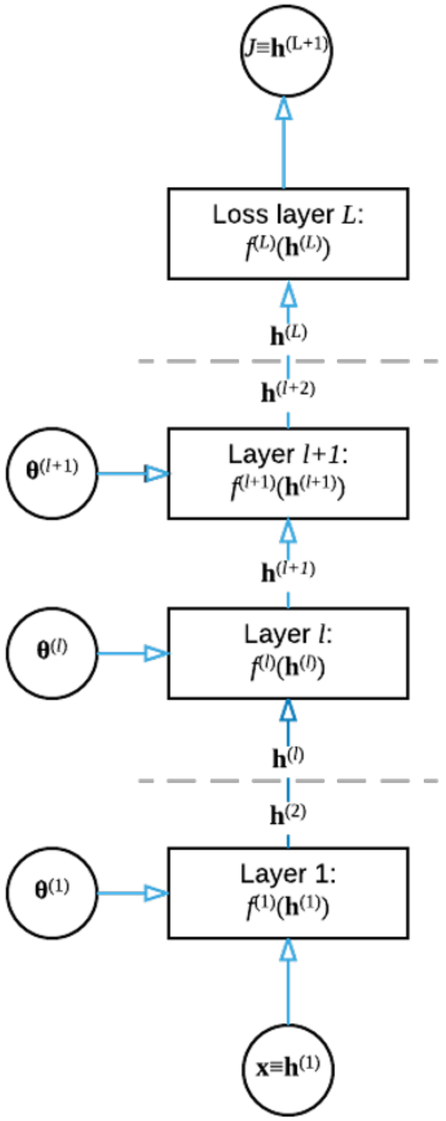
The back propagation algorithm: setup



$$h^{(l)} = f^{(l)}(h^{(l-1)}; \theta^{(l)})$$

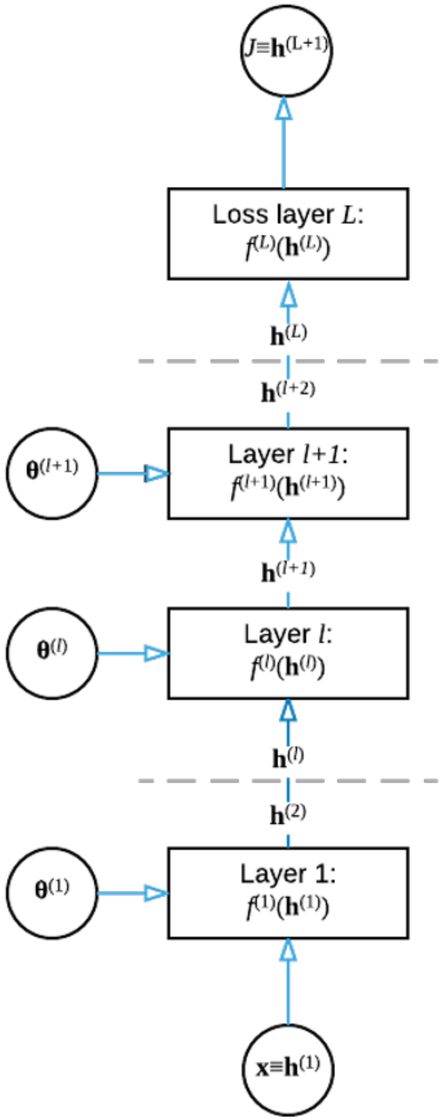
$$J(\mathbf{x}) = f^{(L)}\left(f^{(L-1)}\left(\ldots f^{(2)}\left(f^{(1)}(\mathbf{x})\right)\right)\right)$$

The back propagation algorithm: setup



- What we need:
  - Efficient way to compute the gradient of the loss J, w.r.t. every **parameter** in the neural network $\boldsymbol{\Theta}^{(l)}$
- Backpropagation:
  - Does the efficiently by recursively applying the chain rule of differentiation
  - A lot of the computations are store in memory and then reused

The back propagation algorithm

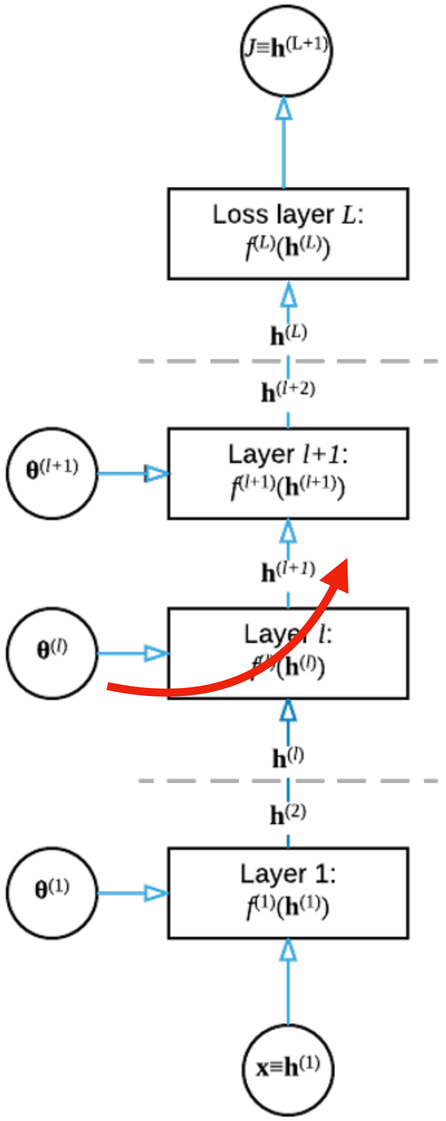


$$J(\boldsymbol{\theta}) = f^{(L)}(\mathbf{h}^{(L)})$$

$$\mathbf{h}^{(L)} = f^{(L-l)}(\mathbf{h}^{(L-1)})$$

$$J(\boldsymbol{\theta}) = f^{(L)}(f^{(L-1)}(\ldots f^{(1)}(\mathbf{x})))$$

The back propagation algorithm



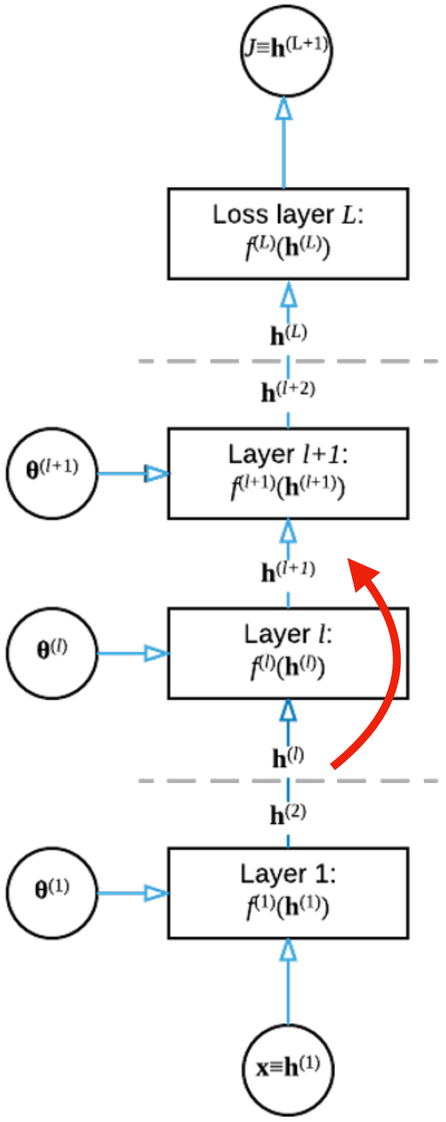
$$\frac{\partial J}{\partial \boldsymbol{\theta}^{(l)}} = \frac{\partial J}{\partial \mathbf{h}^{(l+1)}} \frac{\partial \mathbf{h}^{(l+1)}}{\partial \boldsymbol{\theta}^{(l)}}$$

$$\frac{\partial J}{\partial \boldsymbol{\theta}^{(l)}} = \frac{\partial J}{\partial \mathbf{h}^{(l+1)}} \frac{\partial f^{(l)}\left(\mathbf{h}^{(l)}\right)}{\partial \boldsymbol{\theta}^{(l)}}$$

We need to figure out how to compute this.

We know the function $f^{(l)}(h^{(l)})$ so we know how to implement/compute this!

The back propagation algorithm



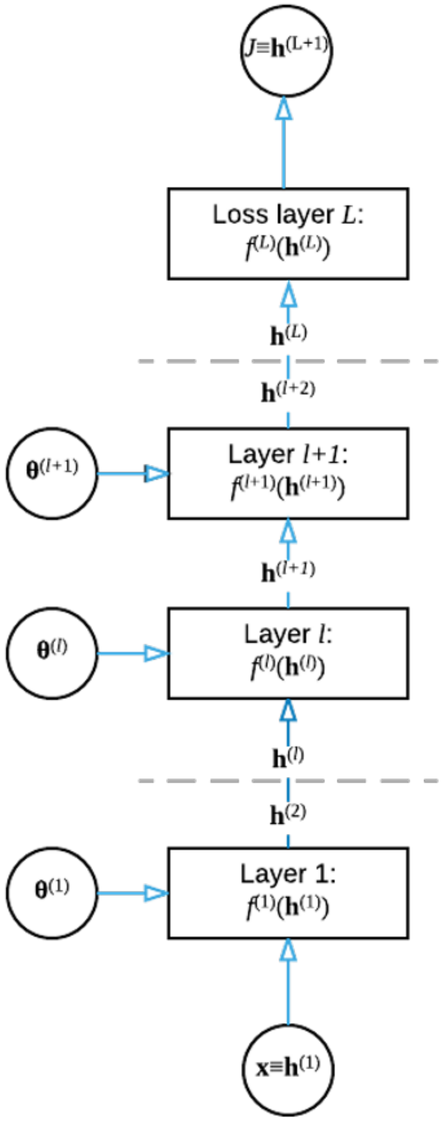
$$\frac{\partial J}{\partial \mathbf{h}^{(l)}} = \delta^{(l)} = \frac{\partial J}{\partial \mathbf{h}^{(l+1)}} \frac{\partial \mathbf{h}^{(l+1)}}{\partial \mathbf{h}^{(l)}}$$

$$\delta^{(l)} = \delta^{(l+1)} \frac{\partial \mathbf{h}^{(l+1)}}{\partial \mathbf{h}^{(l)}}$$

$$\delta^{(l)} = \delta^{(l+1)} \frac{\partial f^{(l)}\left(\mathbf{h}^{(l)}\right)}{\partial \mathbf{h}^{(l)}}$$

We know the function $f^{(l)}(h^{(l)})$ so we know how to implement/compute this!

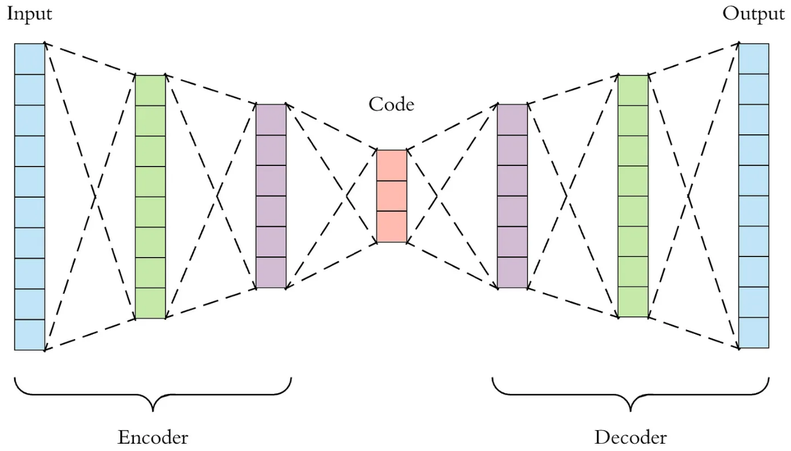
The back propagation algorithm

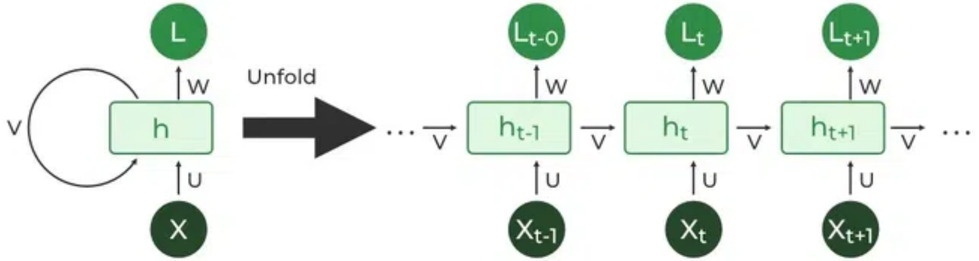


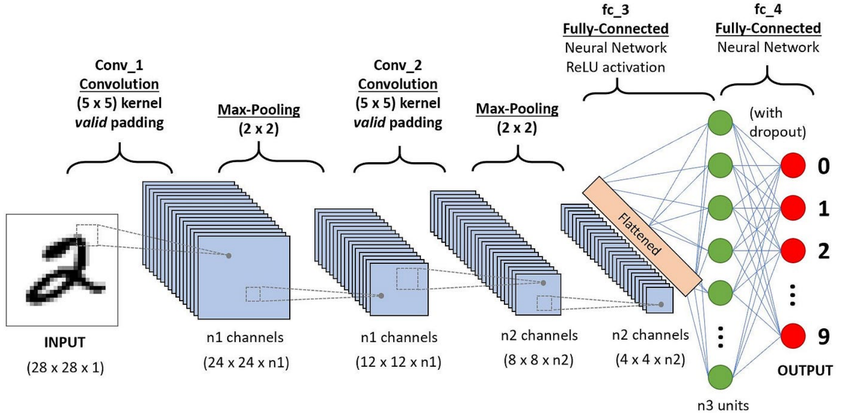
- **Forward computation:**
  - For every layer, given the input and parameters, compute the output.
- **Backward computation:**
  - For every layer l, given the input $\mathbf{h}^{(l)}$, parameters $\theta^{(l)}$ and the "error term" $\delta^{(l)}$ passed from the layer above compute:
    - (1) the derivative of the loss w.r.t. the layer parameters $\frac{\partial J}{\partial \boldsymbol{\theta}^{(l)}}$ to be used for the optimisation.
    - (2) the error term $\delta^{(l)}$ to be passed to the layer below for further computation.

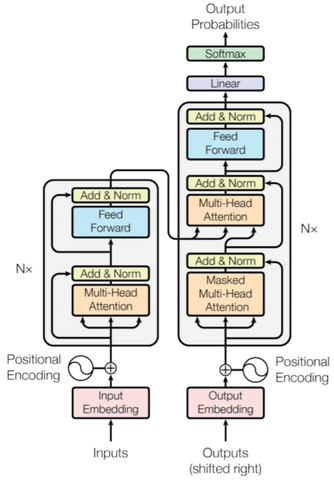
Overview of other neural network models

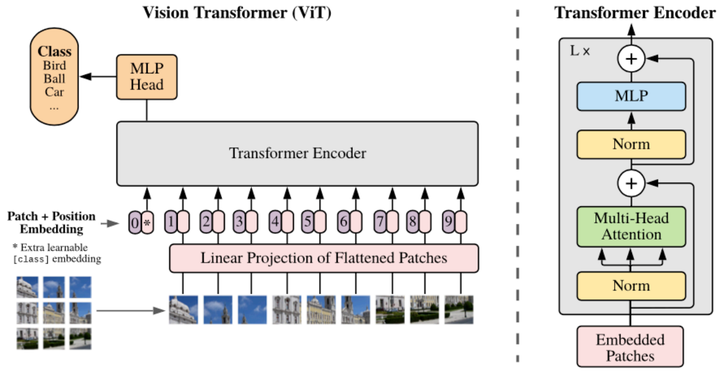
Brief and incomplete (also not in the exam)











Questions?