Deep Learning in Scientific Computing

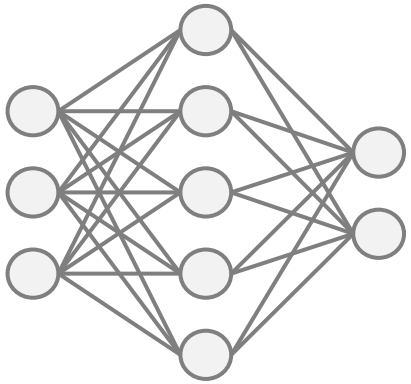
Introduction to Deep Learning Part 2

Spring Semester 2023

Siddhartha Mishra
Ben Moseley

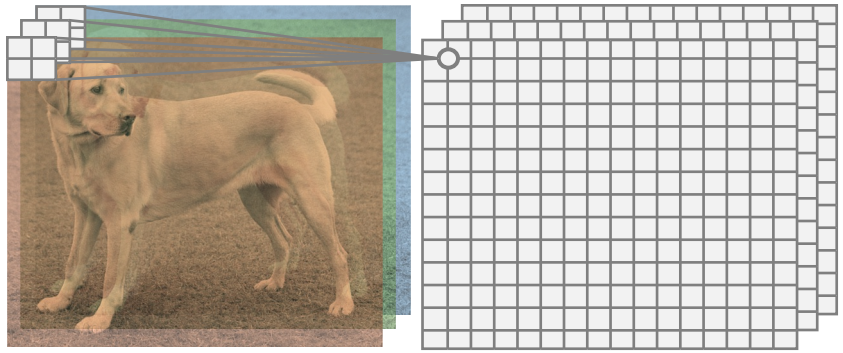
ETH zürich

Recap – MLPs and CNNs



Multilayer perceptron (MLP)

$$NN(x;\theta) = W_2\sigma(W_1 x + b_1) + b_2$$



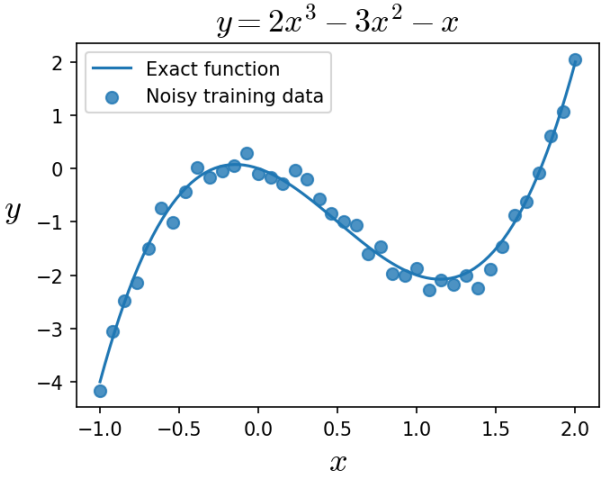
Convolutional neural network (CNN)

$$NN(x;\theta) = W_2 \star \sigma(W_1 \star x + b_1) + b_2$$

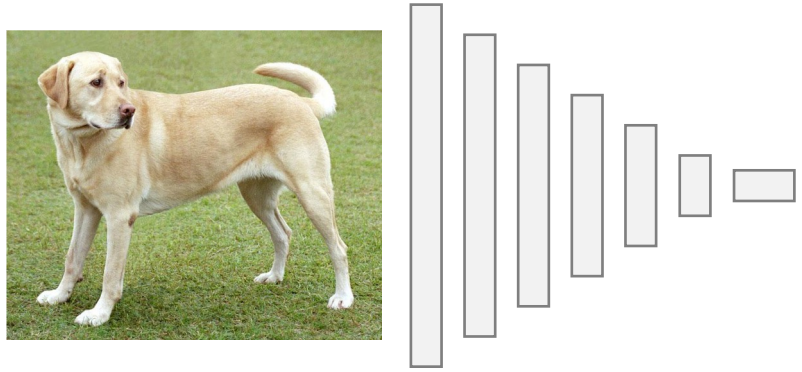
Neural networks are simply **flexible functions** fit to data

Recap – Popular deep learning tasks

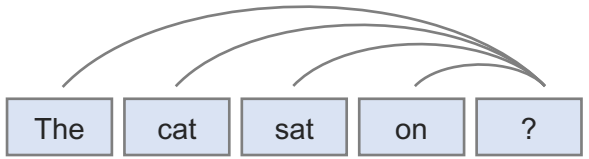
Supervised learning - **regression**



Unsupervised learning – **feature learning**



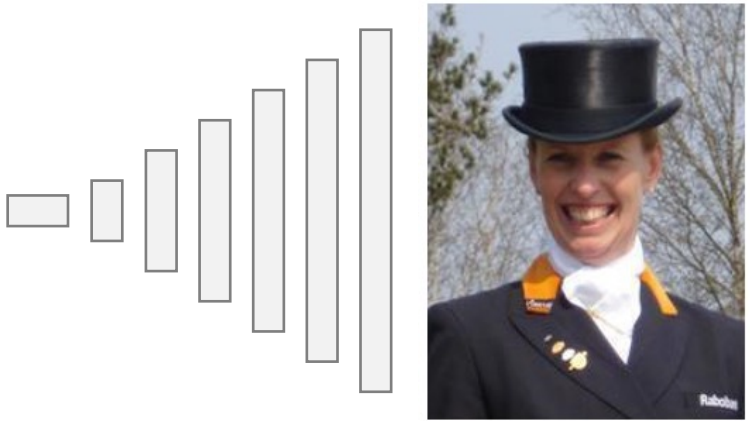
Unsupervised learning – **autoregression**



Supervised learning - **classification**

$P(\text{dog} \mid x) = 1$

Unsupervised learning – **generative modelling**



Recap – Automatic differentiation

Neural networks can be trained using gradient descent:

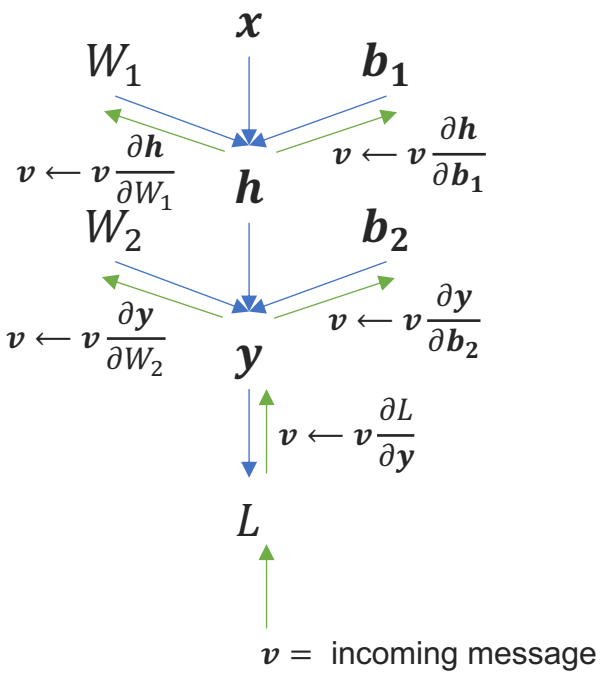
$$\theta_j \leftarrow \theta_j - \gamma \frac{\partial \sum_i^N (NN(x_i;\theta) - y_i)^2}{\partial \theta_j}$$

Multivariate **chain** rule:

$$NN(\boldsymbol{x};\theta) = \boldsymbol{f} \circ \boldsymbol{g} \circ \boldsymbol{h}(\boldsymbol{x};\theta)$$

$$\frac{\partial NN}{\partial W_1} = \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{g}} \frac{\partial \boldsymbol{g}}{\partial \boldsymbol{h}} \frac{\partial \boldsymbol{h}}{\partial W_1}$$

Chain rule can be efficiently evaluated using vector-Jacobian products (**reverse-mode differentiation**)



Autodifferentiation = exact gradients of **arbitrary** programs

Constructs a graph and uses message-passing of vjps / jvps

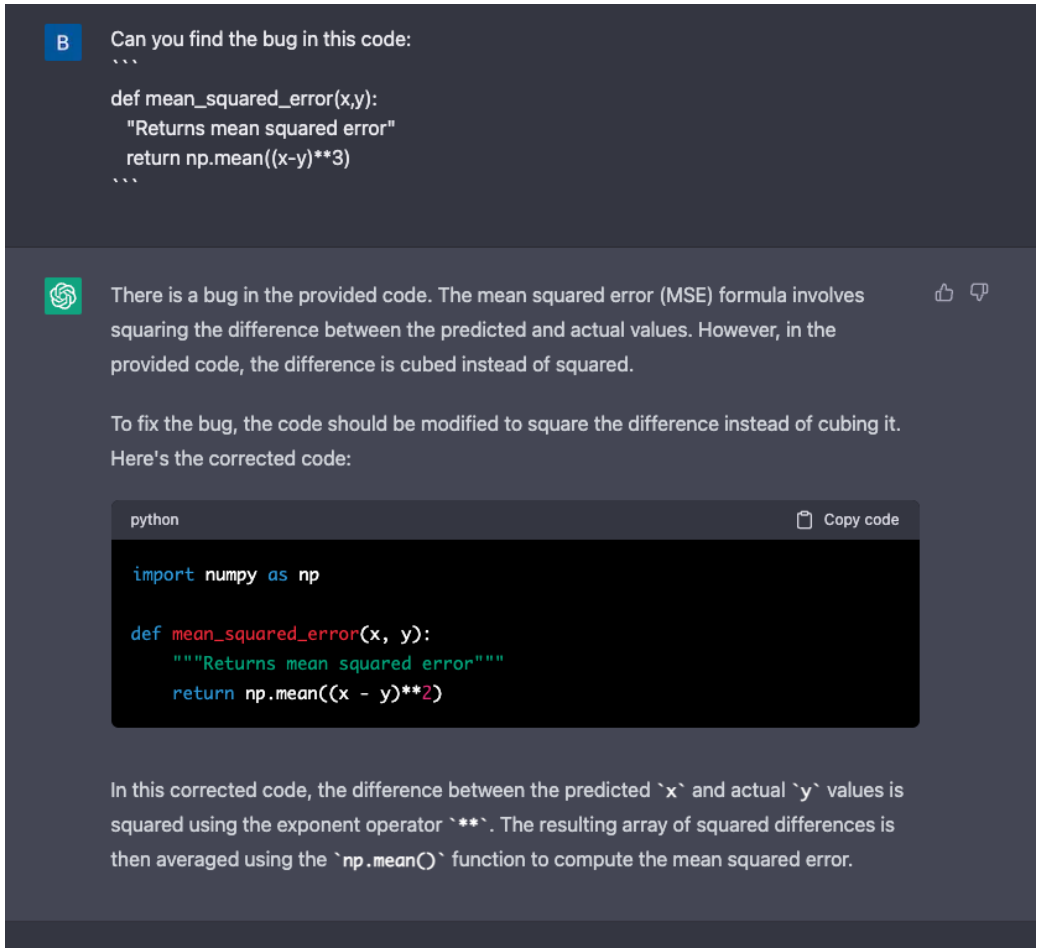
Combined, evaluates the chain rule

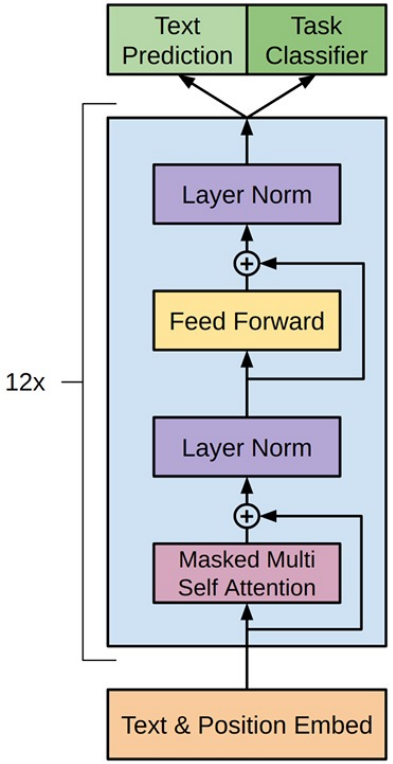
Course timeline

Tutorials		Lectures	
Tue 3:15-14:00, HG E5		Fri 12:15-14:00, HG D1.1	
21.02.		24.02.	~~Course introduction~~
28.02.	~~Intro to PyTorch~~	03.03.	~~Introduction to deep learning I~~
07.03.	~~Simple DNNs in PyTorch~~	10.03.	**Introduction to deep learning II**
14.03.	Advanced DNNs in PyTorch	17.03.	Physics-informed neural networks – introduction and theory
21.03.	PINN exercises	24.03.	Physics-informed neural networks – applications
28.03.	Implementing PINNs I	31.03.	Physics-informed neural networks – extensions
04.04.	Implementing PINNs II	07.04.	
11.04.		14.04.	
18.04.	Introduction to projects	21.04.	Neural operators – introduction and theory
25.04.	Implementing neural operators I	28.04.	Neural operators – applications
02.05.	Implementing neural operators II	05.05.	Neural operators – extensions
09.05.	Operator learning exercises	12.05.	Graph and sequence models
16.05.	Project discussions	19.05.	Differentiable physics – introduction
23.05.	Implementing autodifferentiation	26.05.	Differentiable physics and neural differential equations
30.05.	Intro to JAX	02.06.	Future trends and overview of CAMLAB

State-of-the-art

- Inside ChatGPT – by end of this lecture, you will understand how this works!





Radford et al, Improving Language Understanding by Generative Pre-Training, ArXiv (2018)
Brown et al, Language Models are Few-Shot Learners, NeurIPS (2020)

Lecture overview

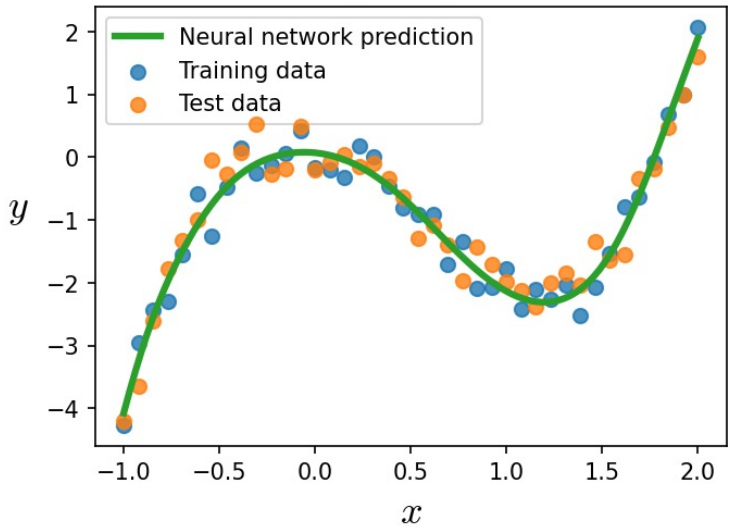
- Challenges of function fitting
  - Overfitting / underfitting
  - Bias / variance
- Regularising deep neural networks
  - Architecture
  - Training data
  - Loss function
- Optimising deep neural networks
  - Stochastic gradient descent
  - Adam / higher-order
- State-of-the-art models
  - Transformers, ChatGPT

Train loss vs Train error vs Test error

Training loss:

$$L(\theta) = \frac{1}{N}\sum_{i}^{N}(NN(x_i;\theta) - y_i)^2$$

$$D_{\text{train}} = \{(x_1, y_1), \dots, (x_N, y_N)\}$$



Train error:

$$L(\theta) = \frac{1}{N}\sum_{i}^{N}|NN(x_i;\theta) - y_i|$$

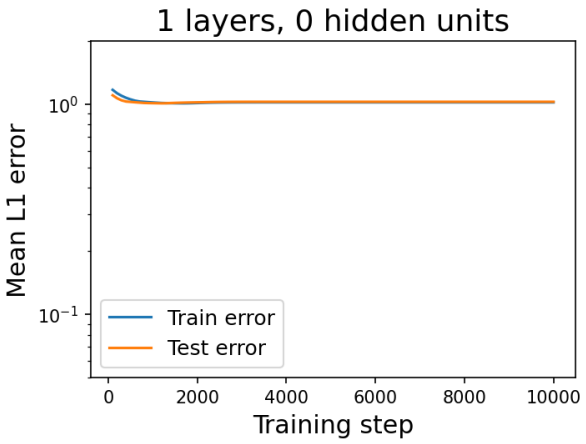
$$D_{\text{train}} = \{(x_1, y_1), \dots, (x_N, y_N)\}$$

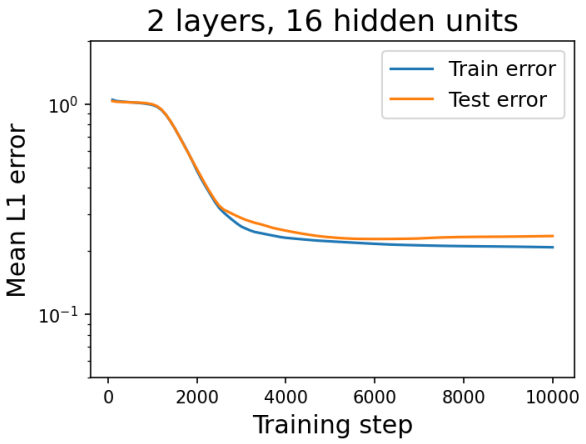
Test error:

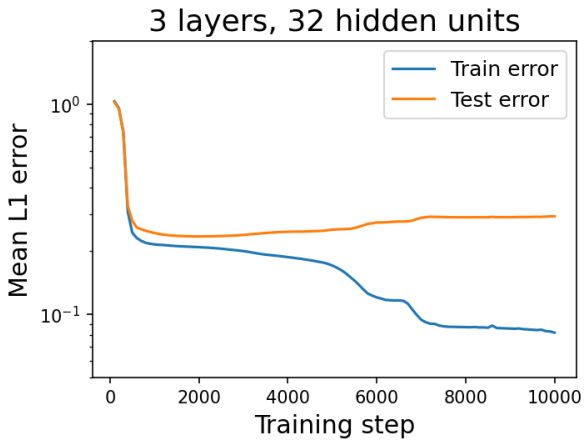
$$L(\theta) = \frac{1}{M}\sum_{i}^{M}|NN(x_i;\theta) - y_i|$$

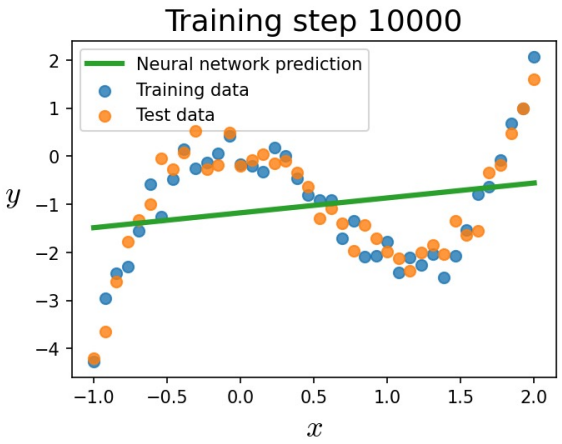
$$D_{\text{test}} = \{(x_1, y_1), \dots, (x_M, y_M)\}$$

Overfitting vs underfitting

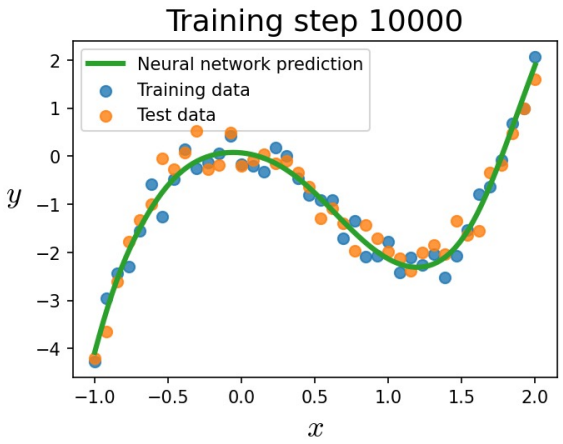




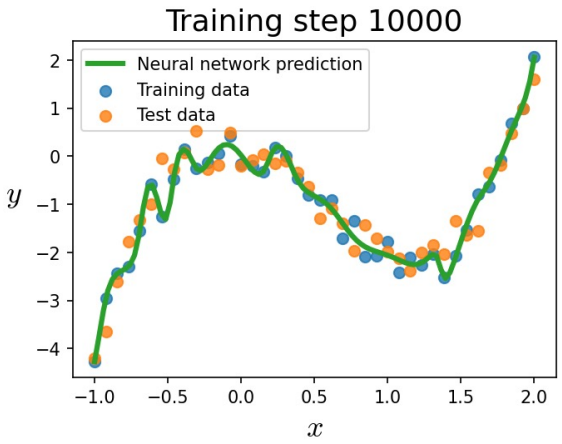




Underfit



Ideal model



Overfit

Understanding error sources

Training loss:

$$L(\theta) = \frac{1}{N}\sum_{i}^{N}(NN(x_i;\theta) - y_i)^2$$
$$D = \{(x_1, y_1), \dots, (x_N, y_N)\}$$

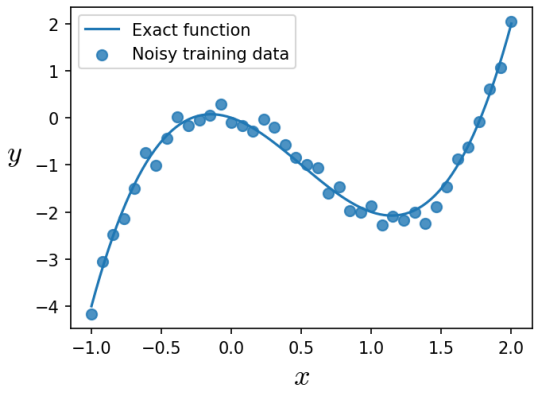
This is also known as the **empirical loss**, and can be more generally written as:

$$L(\theta) = \frac{1}{N}\sum_{i}^{N} l(NN(x_i;\theta), y_i), \qquad x, y \sim p(x, y)$$

But what we really want to minimise is the **expected loss**:

$$\mathcal{L}(\theta) = \iint l(NN(x;\theta), y)\, p(x, y) dx dy$$
$$= E_{(x,y)\sim p}[l(x, y; \theta)]$$

But this is impossible in practice!



Understanding error sources

Training loss:

$$L(\theta) = \frac{1}{N}\sum_{i}^{N}(NN(x_i;\theta) - y_i)^2$$
$$D = \{(x_1, y_1), \dots, (x_N, y_N)\}$$

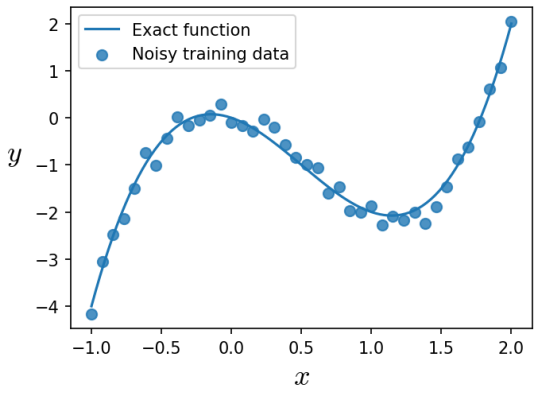
This is also known as the **empirical loss**, and can be more generally written as:

$$L(\theta) = \frac{1}{N}\sum_{i}^{N} l(NN(x_i;\theta), y_i), \qquad x, y \sim p(x, y)$$

But what we really want to minimise is the **expected loss**:

$$\mathcal{L}(\theta) = \iint l(NN(x;\theta), y)\, p(x, y) dx dy$$
$$= E_{(x,y)\sim p}[l(x, y; \theta)]$$

But this is impossible in practice!



This means there are two sources of error:

1) **Approximation error** (limited expressivity of neural network)

$$\mathcal{E}_{\text{app}} = \mathcal{L}(\underline{NN^*}) - \mathcal{L}(\underline{y^*})$$

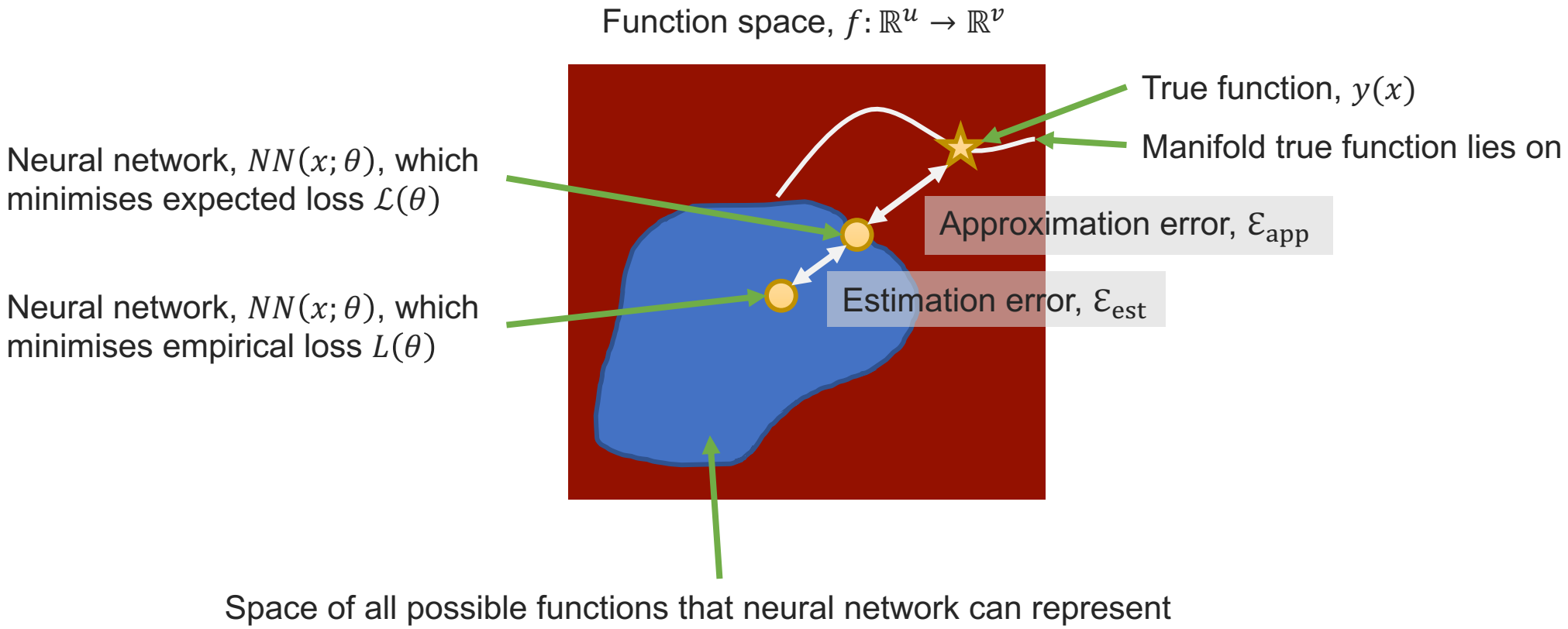
NN which minimises expected loss — True function which minimises expected loss

2) **Estimation error** (finite amount of training data) (aka generalisation error)

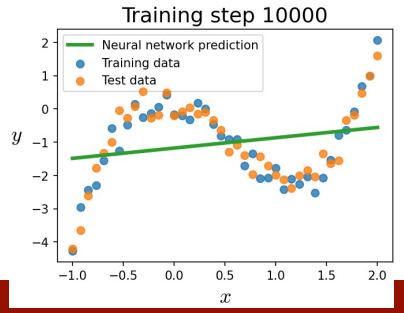
$$\mathcal{E}_{\text{est}} = \mathcal{L}(\underline{NN}) - \mathcal{L}(\underline{NN^*})$$

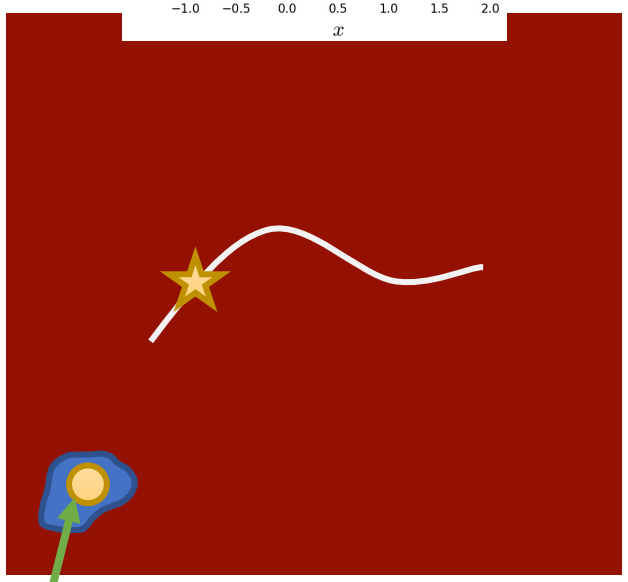
NN which minimises empirical loss — NN which minimises expected loss

Function space

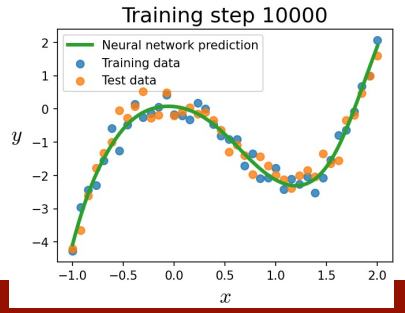


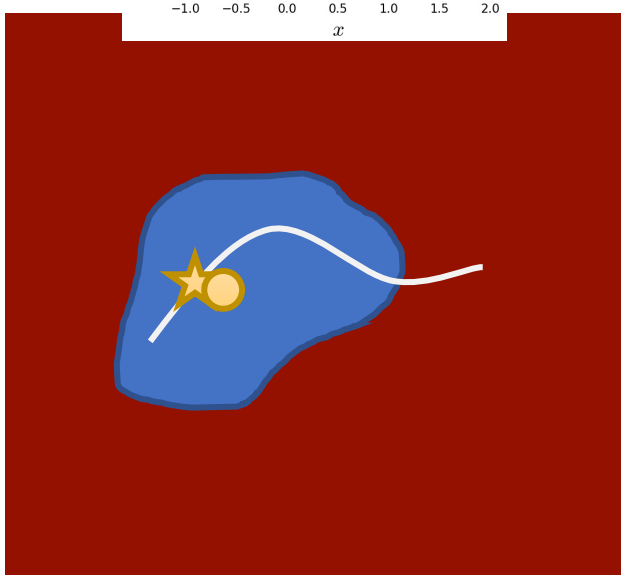
Function space



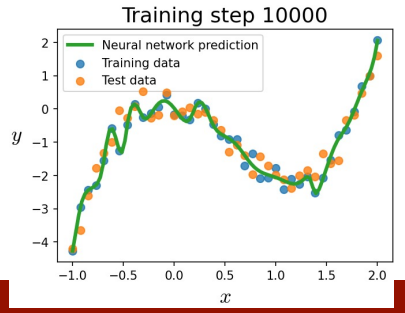


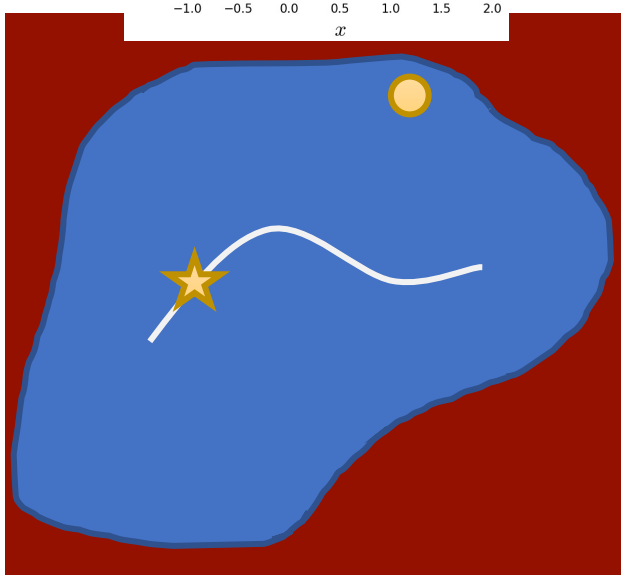
Underfit





Ideal model





Overfit

Neural network, $NN(x;\theta)$, which minimises empirical loss $L(\theta)$

Decomposing expected loss

Let $h(x, D) = NN(x; \theta)$ be the function learned by the neural network using a **particular** training dataset $D = \{(x_1, y_1), \dots, (x_N, y_N)\}$

Then we would like to understand what the expected loss is averaged over **all possible training datasets**, i.e.

$$E_{(x,y)\sim p,\, D\sim p^N}[l(x, y, D)]$$

Assume a simple least-squares loss function, e.g.

$$E_{x,y,D}[(h(x, D) - y)^2]$$

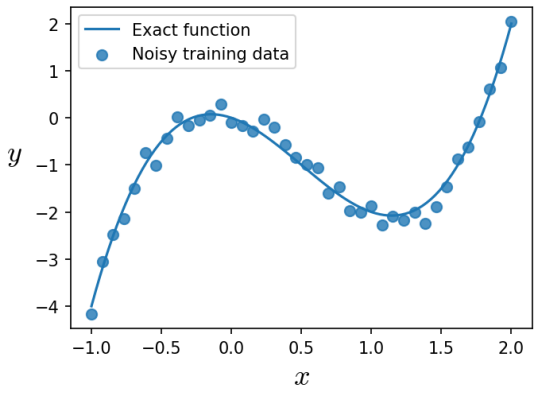
Then it can be shown*:

$$E_{x,y,D}[(h(x, D) - y)^2] = \underbrace{E_x\left[\left(\bar{h}(x) - \bar{y}(x)\right)^2\right]}_{\textbf{Bias}^2} + \underbrace{E_{x,D}\left[\left(h(x, D) - \bar{h}(x)\right)^2\right]}_{\textbf{Variance}} + \underbrace{E_{x,y}[(\bar{y}(x) - y)^2]}_{\textbf{Irreducible noise}}$$

Where

$\bar{y}(x) = E_{y|x}[y] = \int y\, p(y|x) dy$ = Average observation

$\bar{h}(x) = E_D[h(x, D)] = \int h(x, D)\, p(D) dD$ = Average model



*for full derivation, see e.g. https://www.cs.cornell.edu/courses/cs4780/2018fa/lectures/lecturenote12.html

Decomposing expected loss

From previous slide:

$$E_{x,y,D}[(h(x,D)-y)^2] = \underbrace{E_x\left[\left(\bar{h}(x)-\bar{y}(x)\right)^2\right]}_{\textbf{Bias}^2} + \underbrace{E_{x,D}\left[\left(h(x,D)-\bar{h}(x)\right)^2\right]}_{\textbf{Variance}} + \underbrace{E_{x,y}[(\bar{y}(x)-y)^2]}_{\textbf{Irreducible noise}}$$

Where

$\bar{y}(x) = E_{y|x}[y] = \int y\, p(y|x)dy$ = Average observation

$\bar{h}(x) = E_D[h(x,D)] = \int h(x,D)\, p(D)dD$ = Average model

Bias = “average” model error

Variance = sensitivity of model prediction to training data

Irreducible noise = inherent noise in the training observations; you can never beat this ($\bar{y}(x)$ is the best possible predictor of y)

Decomposing expected loss

From previous slide:

$$E_{x,y,D}[(h(x,D)-y)^2] = \underbrace{E_x\left[\left(\bar{h}(x)-\bar{y}(x)\right)^2\right]}_{\textbf{Bias}^2} + \underbrace{E_{x,D}\left[\left(h(x,D)-\bar{h}(x)\right)^2\right]}_{\textbf{Variance}} + \underbrace{E_{x,y}[(\bar{y}(x)-y)^2]}_{\textbf{Irreducible noise}}$$

Where

$\bar{y}(x) = E_{y|x}[y] = \int y\, p(y|x)dy$ = Average observation

$\bar{h}(x) = E_D[h(x,D)] = \int h(x,D)\, p(D)dD$ = Average model

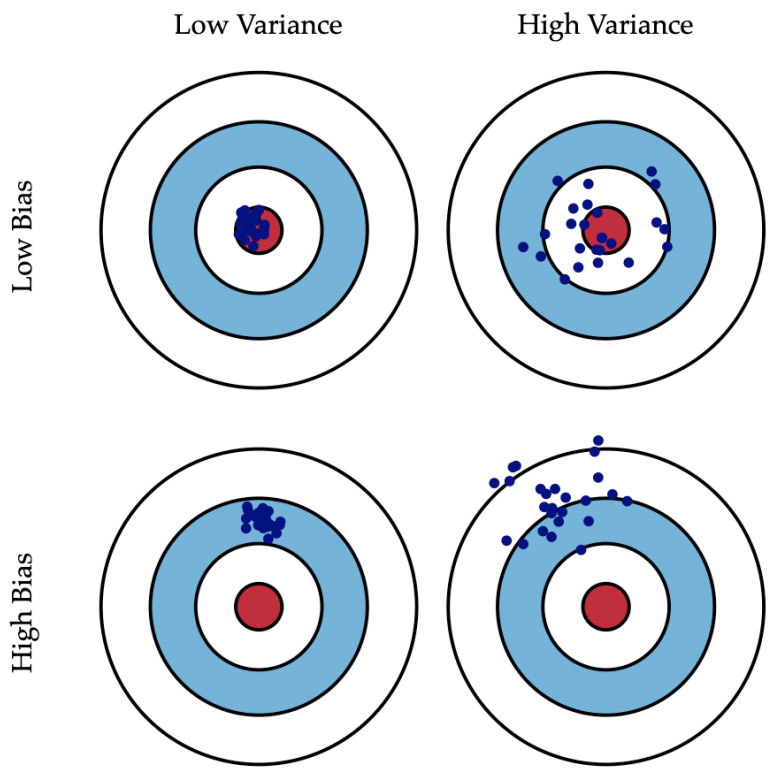
Bias = “average” model error

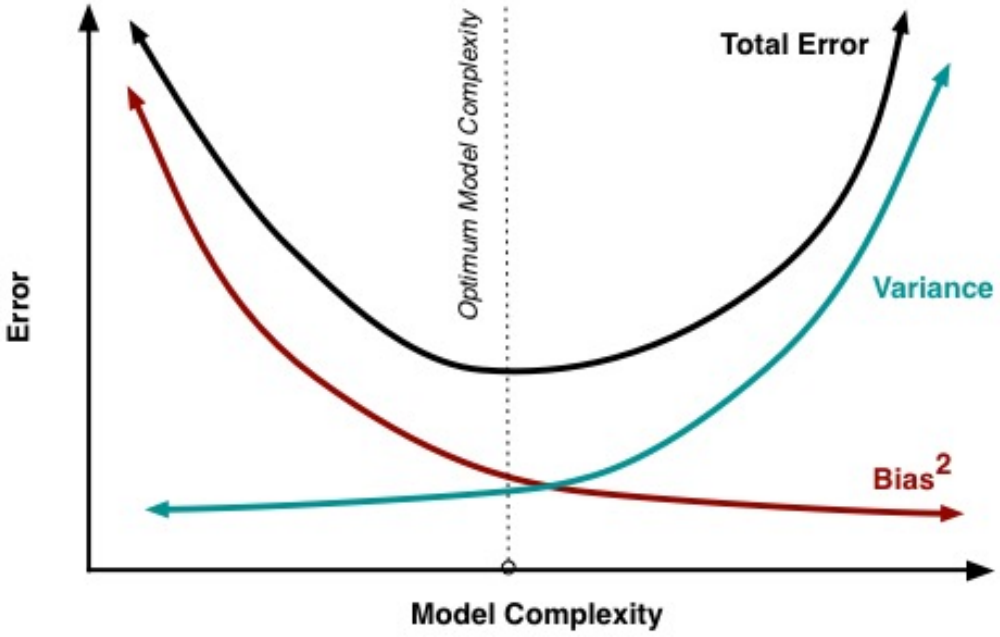
Variance = sensitivity of model prediction to training data

Irreducible noise = inherent noise in the training observations; you can never beat this ($\bar{y}(x)$ is the best possible predictor of y)

Typically, there is a bias-variance **tradeoff**!

Bias-variance tradeoff



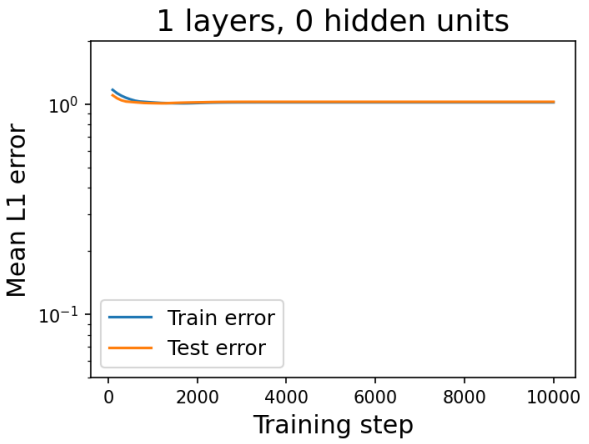


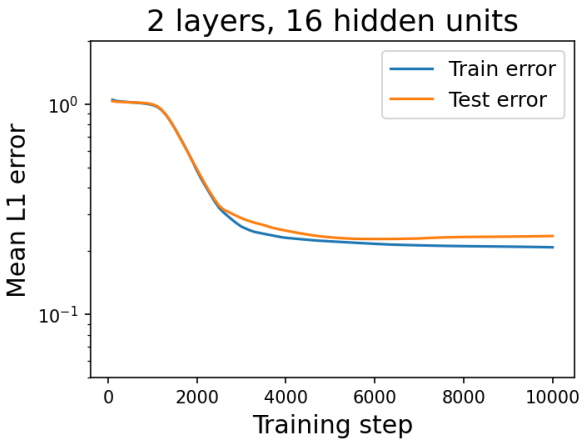
Source: http://scott.fortmann-roe.com/docs/BiasVariance.html

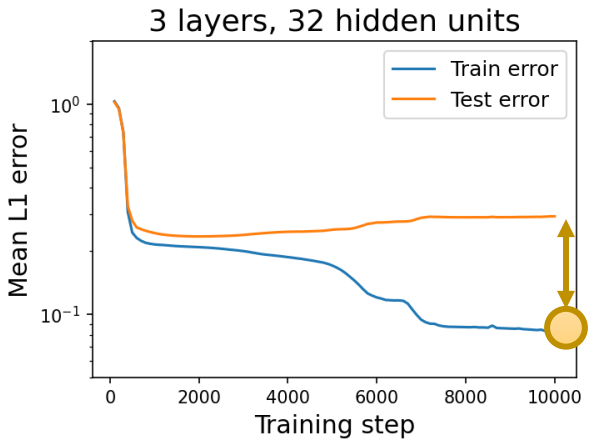
Bias = “average” model error

Variance = sensitivity of model prediction to training data

Overfitting vs underfitting

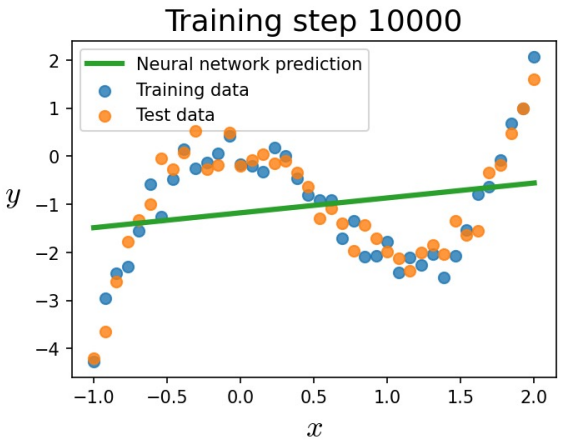


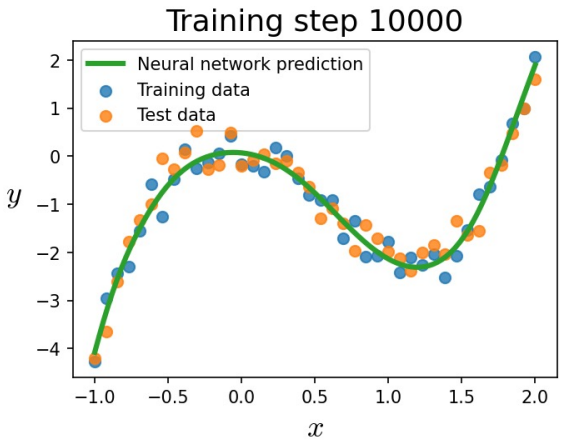


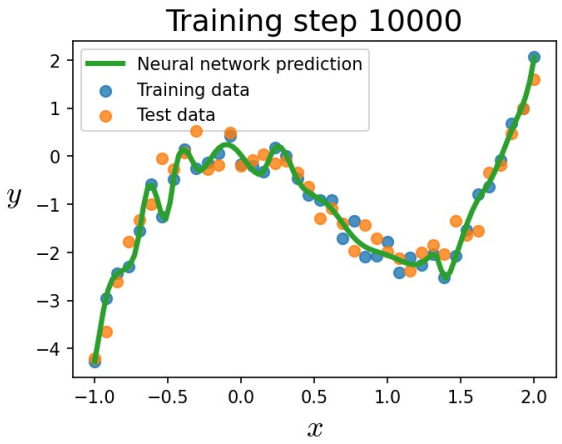


≈ Variance

≈ Bias







High bias
Low variance

Low bias
Low variance

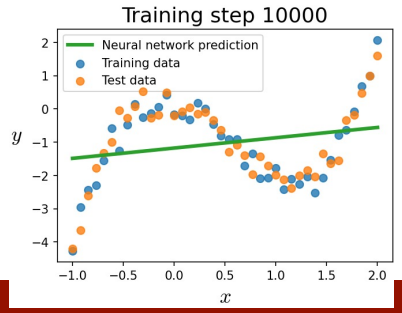
Low bias
High variance

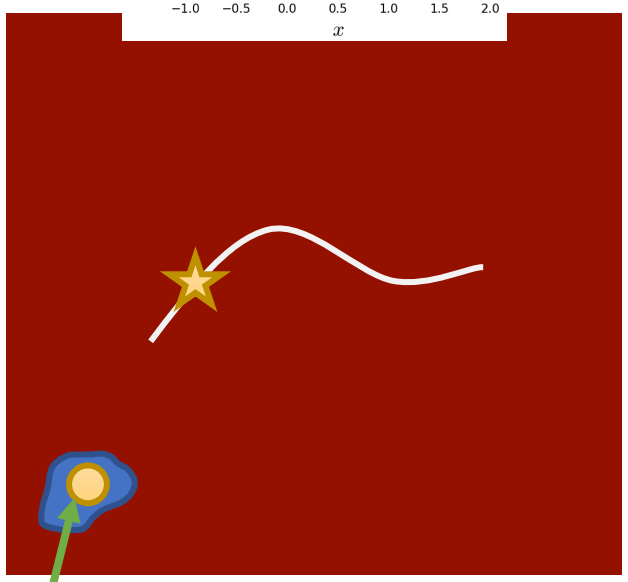
Improving model performance

If you have high bias (or underfitting)
- Increase model **complexity** (e.g. number of free parameters)
- Or modify model architecture (shift location of hypothesis space)

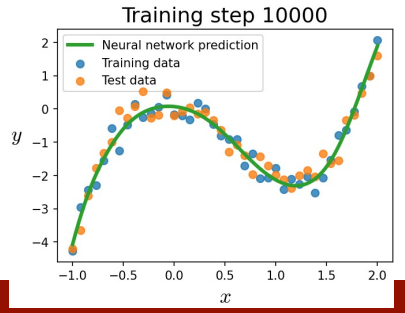
If you have high variance (or overfitting)
- Increase the amount of training **data**
- Or **regularise** the neural network in some other way

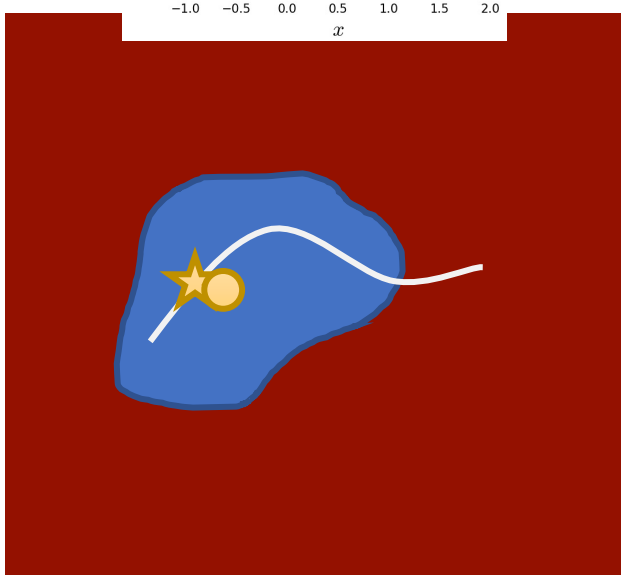
Function space



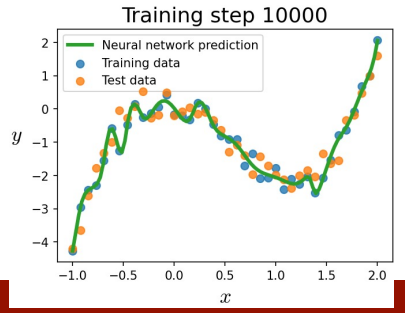


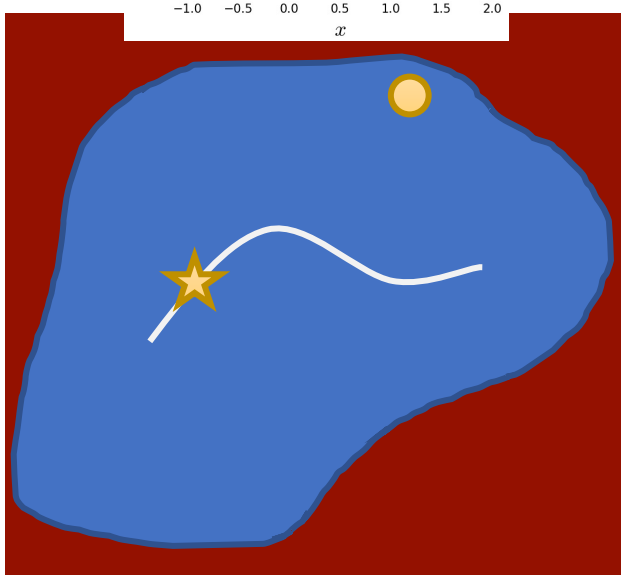
Underfit





Ideal model





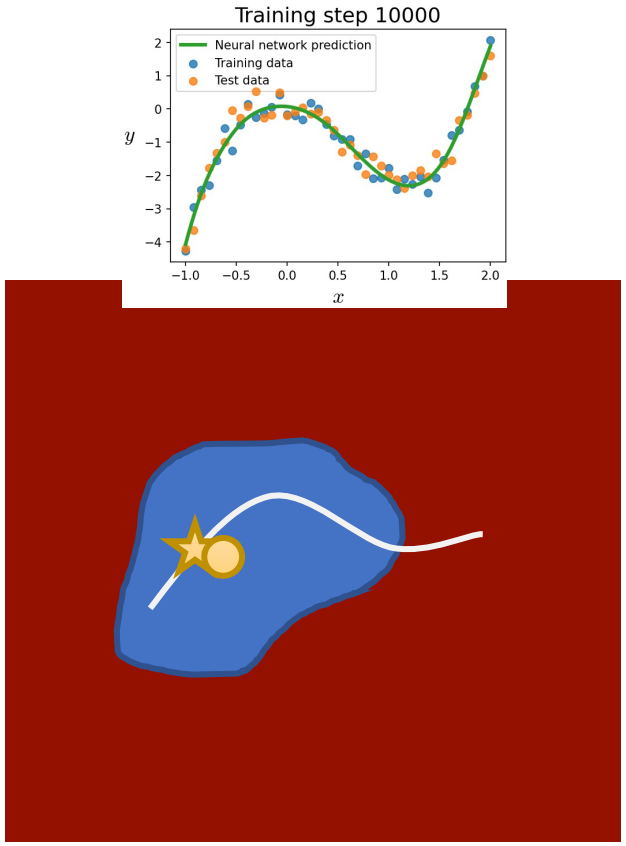
Overfit

Neural network, $NN(x;\theta)$, which minimises empirical loss $L(\theta)$

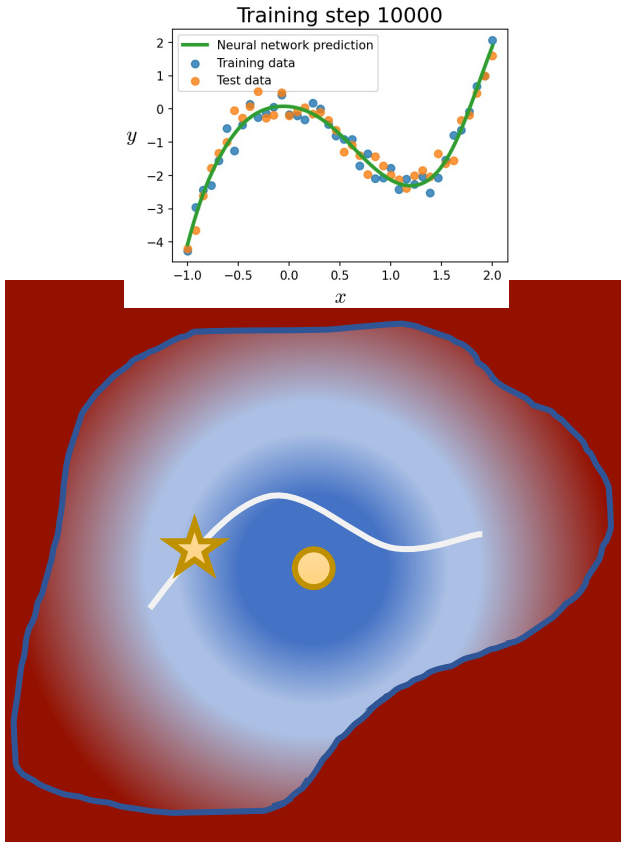
Regularisation

- As well as changing the boundaries of the function space (hard constraint)

- We can **prefer** certain regions of the function space over others (soft constraint)



Ideal model



Regularised

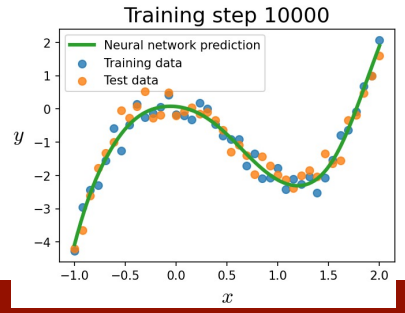
Regularisation

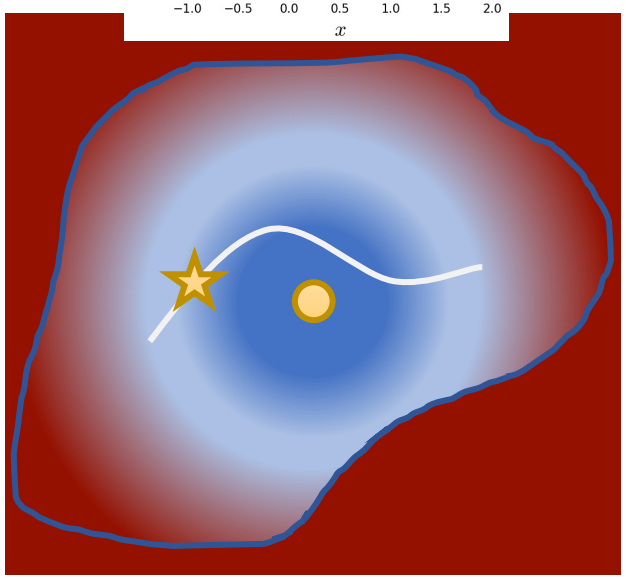
Regularisation = **restrict** the space of possible functions a neural network can learn / **prefer** certain regions of the function space / impart a **prior** on the learning algorithm

Ways to regularise neural networks:

- Reduce model complexity
- Modify model architecture

- Increase amount of (or augment) training data
- Weight regularization / pruning
- Additional loss terms
- Dropout
- Early stopping





Regularised

Weight regularisation

Simple idea: prefer weights that are 0:

$$L(\theta) = \frac{1}{N}\sum_{i}^{N}(NN(x_i;\theta) - y_i)^2 + \lambda\|\theta\|^2$$

Thus, the network learns to **prune** unnecessary connections

- Also known as ridge regression
- Other norms can be used (e.g. L1, L0 etc)

Weight regularisation

Simple idea: prefer weights that are 0:

$$L(\theta) = \frac{1}{N}\sum_{i}^{N}(NN(x_i;\theta) - y_i)^2 + \lambda\|\theta\|^2$$

Thus, the network learns to **prune** unnecessary connections

Probabilistic perspective:

Assume $\hat{p}(y|x,\theta)$ is a **normal** distribution:

$$\hat{p}(y|x,\theta) = \mathcal{N}(y;\mu = NN(x;\theta), \sigma = 1)$$

Then, assume each training datapoint is independently and identically distributed (iid), then the **data likelihood** can be written as:

$$\hat{p}(D|\theta) = p(x_1, y_1, \dots, x_N, y_N|\theta) = \prod_{i}^{N}\hat{p}(y_i|x_i,\theta)$$

Next, assume a **prior** on the parameters of the network:

$$\hat{p}\left(\theta_j\right) = \mathcal{N}\left(\theta_j;\mu = 0, \sigma = \sigma\right) \Rightarrow \hat{p}(\theta) = \prod_{j}\hat{p}(\theta_j)$$

Specifically, that they are iid normally distributed.

Then note that the parameter **posterior** distribution is given by Bayes rule:

$$\hat{p}(\theta|D) = \frac{\hat{p}(D|\theta)\hat{p}(\theta)}{\hat{p}(D)}$$

And use **maximum a posteriori estimation** (MAP) to estimate θ^*:

$$\theta^* = \max_{\theta}\hat{p}(\theta|D)$$

$$= \max_{\theta}\prod_{i}^{N} e^{-\frac{1}{2}\left(\frac{y_i - NN(x_i;\theta)}{1}\right)^2}\prod_{j} e^{-\frac{1}{2}\left(\frac{\theta_j - 0}{\sigma}\right)^2}$$

$$= \min_{\theta}\sum_{i}^{N}(NN(x_i;\theta) - y_i)^2 + \lambda\|\theta\|^2$$

Additional loss terms

Weight regularisation:

$$L(\theta) = \frac{1}{N}\sum_{i}^{N}(NN(x_i;\theta) - y_i)^2 + \lambda\|\theta\|^2$$

Alternatively, we can use any **additional** arbitrary loss term as a regulariser:

$$L(\theta) = \frac{1}{N}\sum_{i}^{N}(NN(x_i;\theta) - y_i)^2 + R(D,\theta)$$

Examples include:

Structural similarity index measure (SSIM) loss (perceptual image quality measure)

$$\mathrm{SSIM}(y,\hat{y}) = \frac{(2\mu_y\mu_{\hat{y}} + c_1)(2\sigma_{y\hat{y}} + c_2)}{(\mu_y^2 + \mu_{\hat{y}}^2 + c_1)(\sigma_y^2 + \sigma_{\hat{y}}^2 + c_2)}$$

Smoothness loss (e.g. in frequency domain)

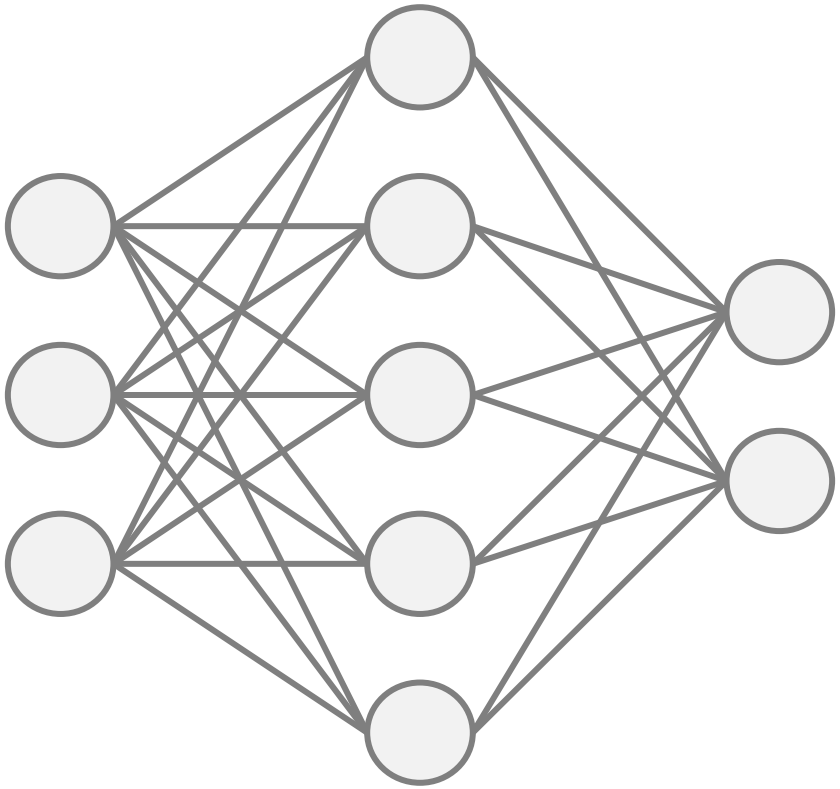
$$L(\hat{y}) = w(\omega)\mathrm{FFT}(\hat{y})$$

Feature space loss (e.g. using a pretrained CNN)

$$F(y,\hat{y}) = \|\mathrm{CNN}(y)_{\mathrm{hidden}} - \mathrm{CNN}(\hat{y})_{\mathrm{hidden}}\|^2$$

Dropout

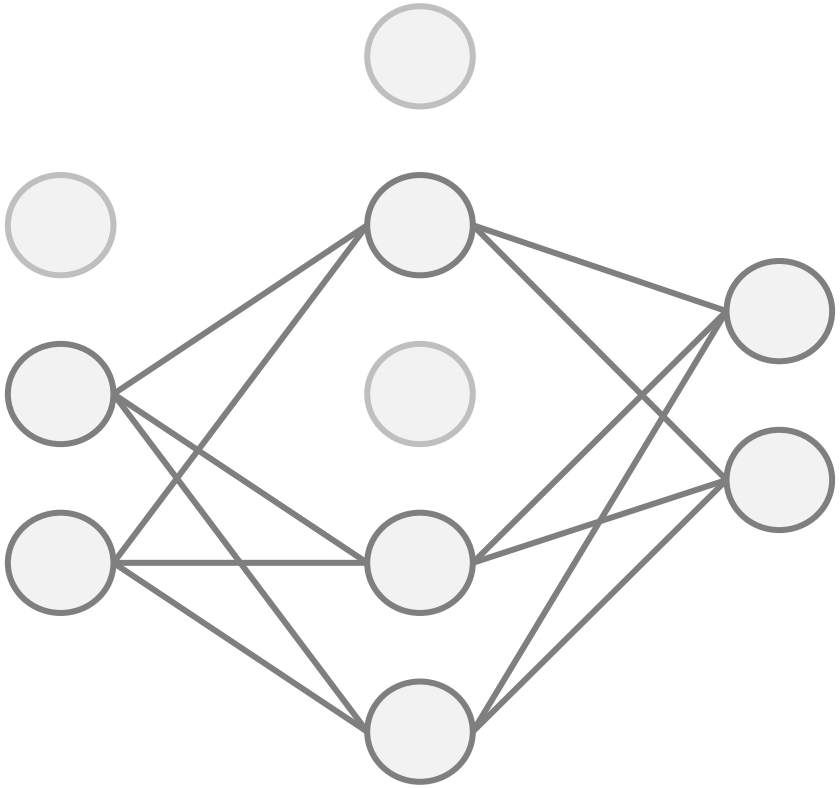
- During training, randomly set some activations to zero



Srivastava et al, Dropout: A Simple Way to Prevent Neural Networks from Overfitting, JMLR (2014)

Dropout

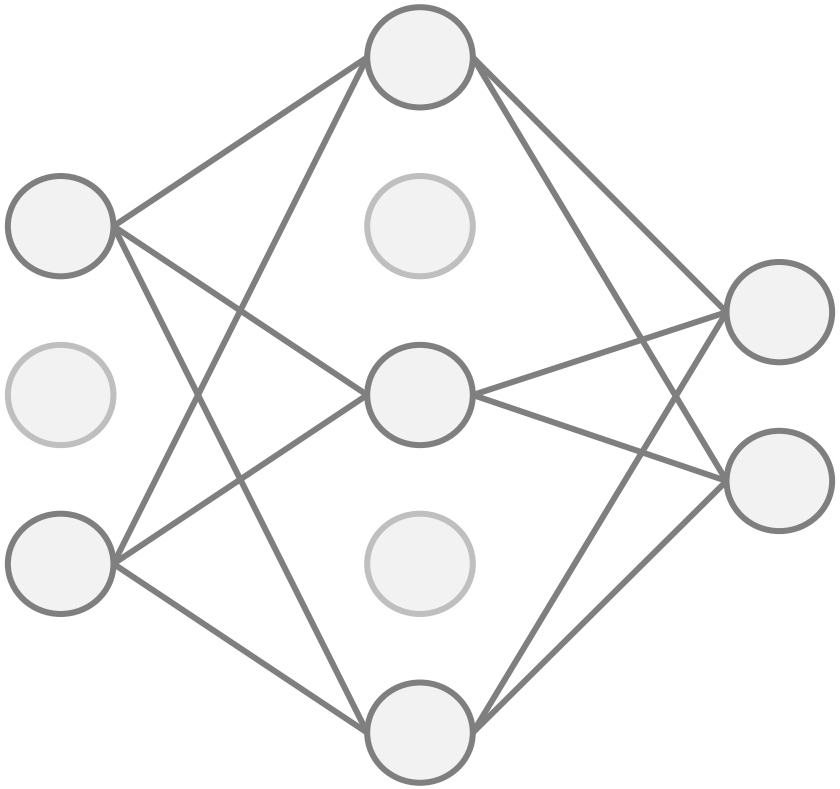
- During training, randomly set some activations to zero
- Typically drop neurons with probability = 50%
- Neural network cannot rely on any single node
- Individual neurons cannot rely on individual inputs



Srivastava et al, Dropout: A Simple Way to Prevent Neural Networks from Overfitting, JMLR (2014)

Dropout

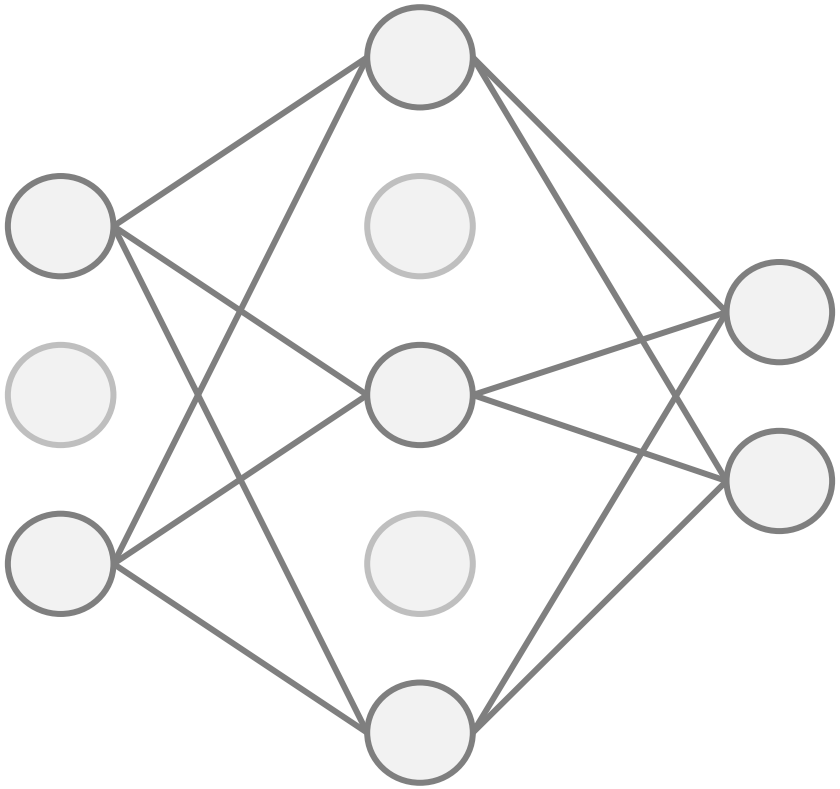
- During training, randomly set some activations to zero
- Typically drop neurons with probability = 50%
- Neural network cannot rely on any single node
- Individual neurons cannot rely on individual inputs



Srivastava et al, Dropout: A Simple Way to Prevent Neural Networks from Overfitting, JMLR (2014)

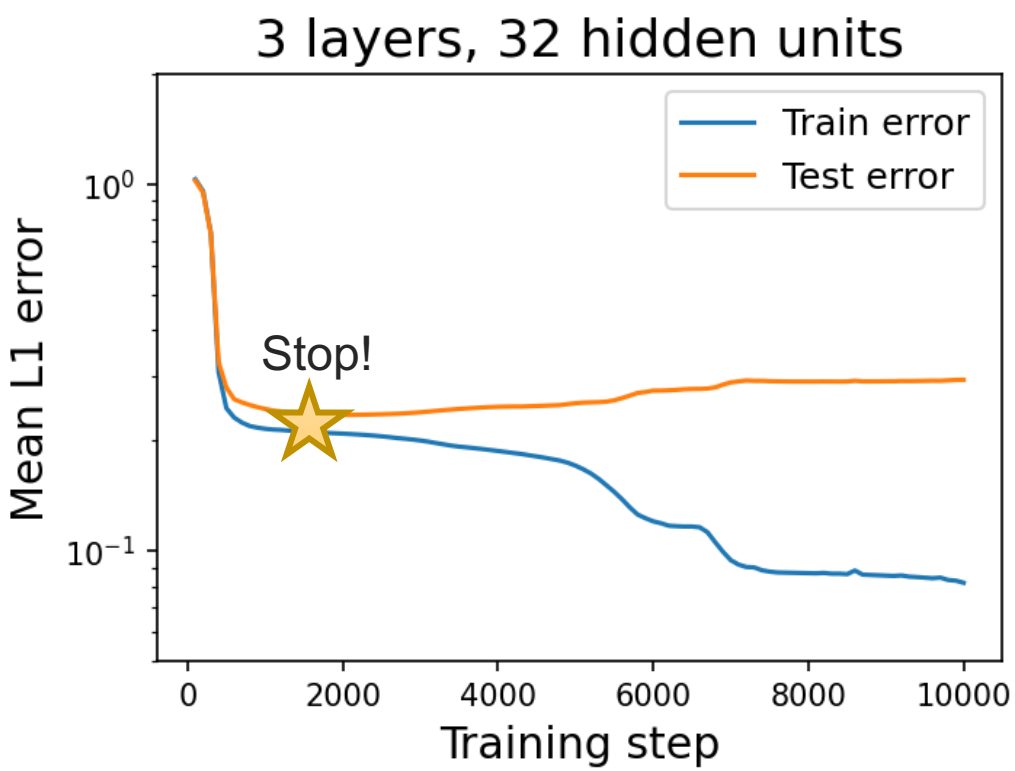
Dropout

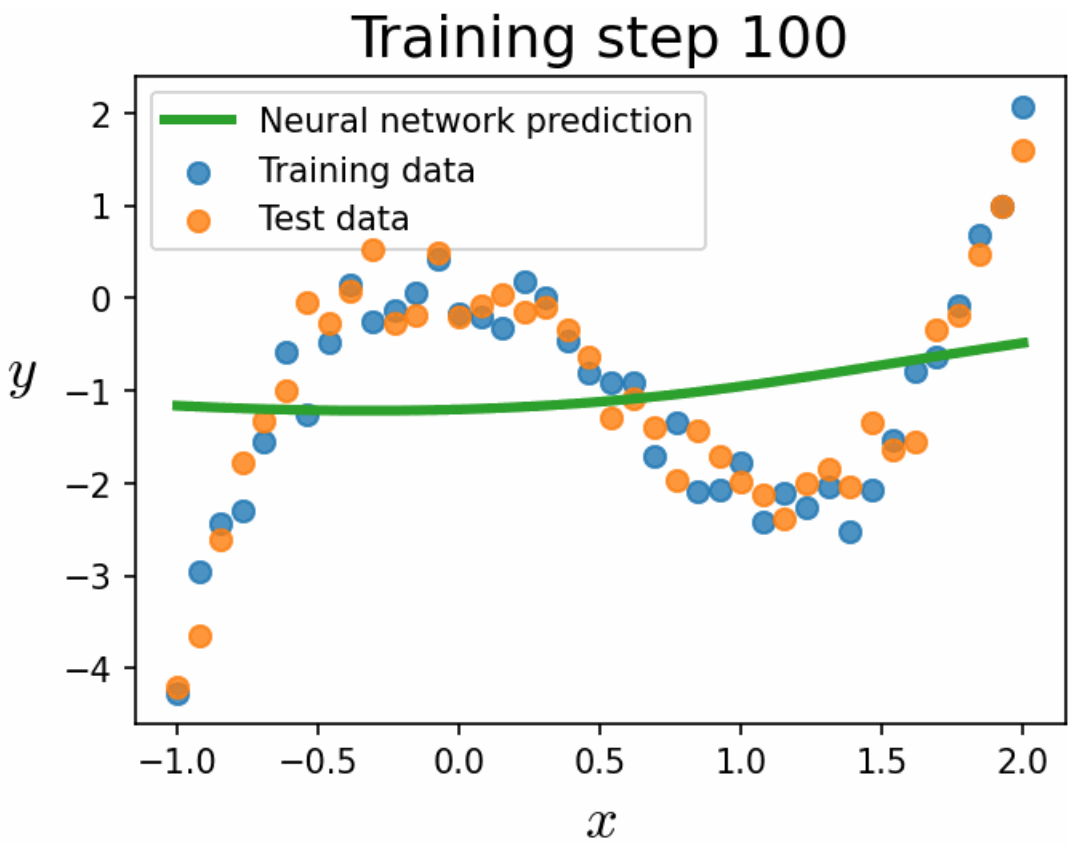
- During training, randomly set some activations to zero
- Typically drop neurons with probability = 50%
- Neural network cannot rely on any single node
- Individual neurons cannot rely on individual inputs
- Effectively trains an ensemble of models
- Samples from this ensemble can be used for uncertainty estimation of network's outputs



Srivastava et al, Dropout: A Simple Way to Prevent Neural Networks from Overfitting, JMLR (2014)

Early stopping





Related to the **spectral bias** of neural networks

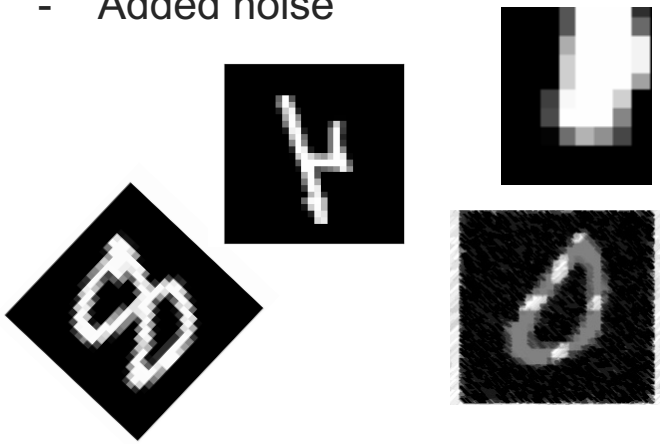
Increase / augment training data

Task: digit classification

Training data: only 9 images!
How to avoid overfitting?

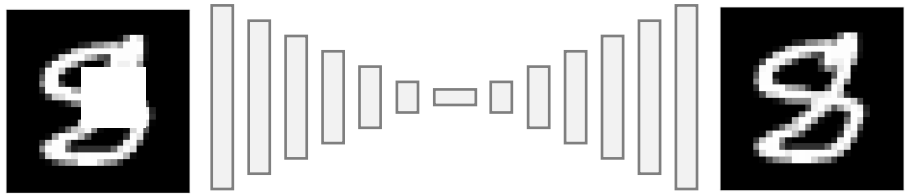
Strategy 1: **augment** training data, eg by using;

- Rotations
- Crops & zooms
- Flips
- Added noise



Strategy 2: **pre-train** the network on an unsupervised task = **self-supervised learning**

E.g.
- Reconstruction
- Predicting missing data
- Patch location

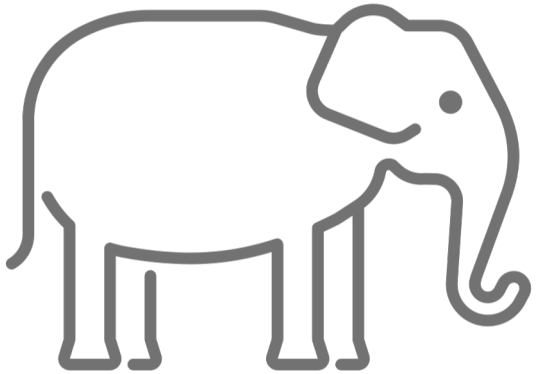


5 min break

Lecture overview

- Challenges of function fitting
  - Overfitting / underfitting
  - Bias / variance
- Regularising deep neural networks
  - Architecture
  - Training data
  - Loss function
- Optimising deep neural networks
  - Stochastic gradient descent
  - Adam / higher-order
- State-of-the-art models
  - Transformers, ChatGPT

Another source of error…



Up until now, we have assumed that the learned neural network parameters **globally** minimise the empirical loss, i.e.

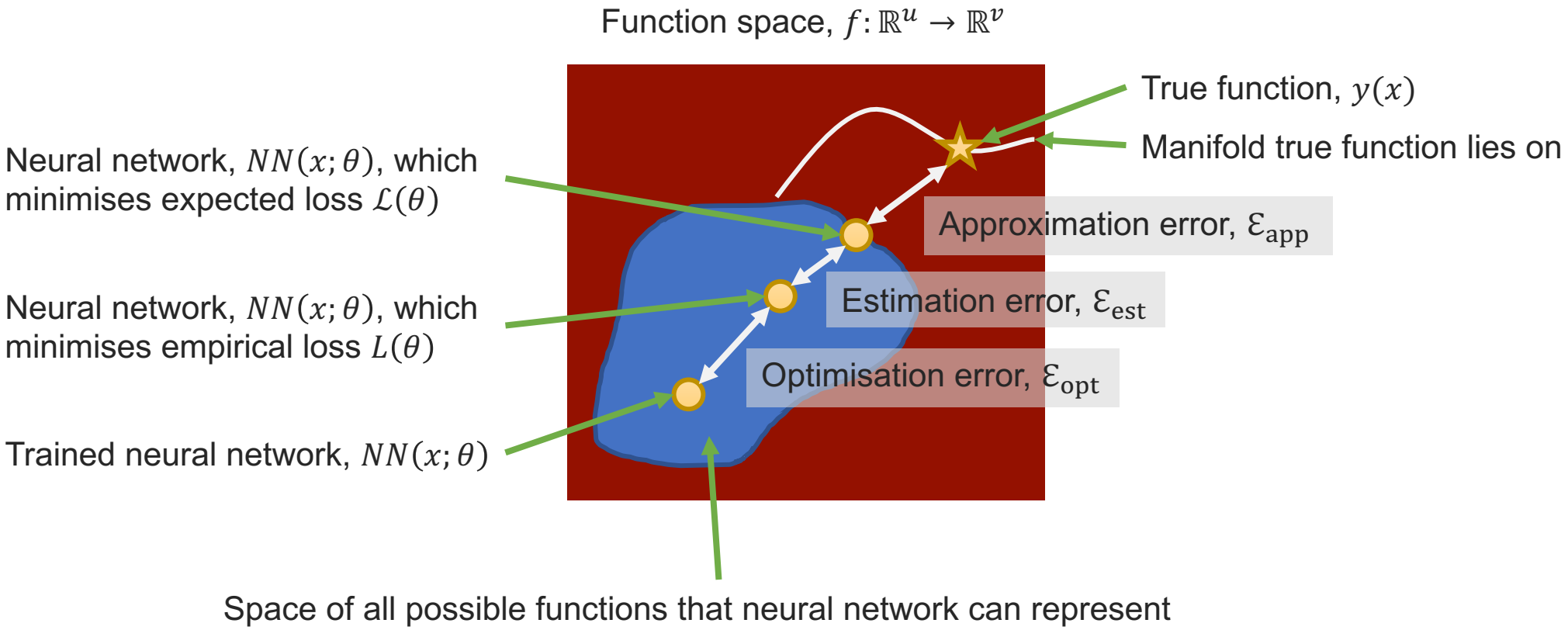
$$\theta^* = \min_{\theta} L(\theta)$$

⚠ But, in practice, finding the global minima is **extremely** hard

So, we must also consider the effect of **optimisation error** on the performance of neural networks

⚠ => Even though a large enough neural network can represent any arbitrary function (universal approximation theorem), there is **no guarantee** finding this network is tractable!

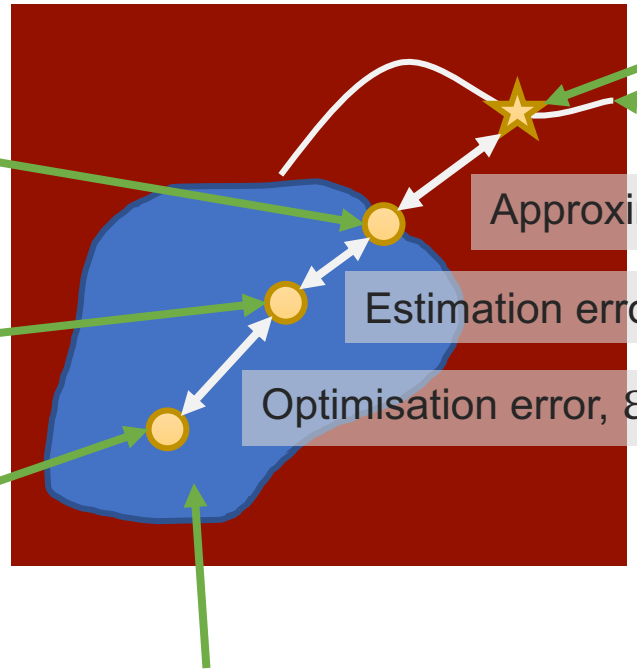
All error sources



$$\mathcal{E} = \mathcal{E}_{\text{app}} + \mathcal{E}_{\text{est}} + \mathcal{E}_{\text{opt}}$$

All error sources

Function space, $f: \mathbb{R}^u \to \mathbb{R}^v$



True function, $y(x)$

Manifold true function lies on

Neural network, $NN(x;\theta)$, which minimises expected loss $\mathcal{L}(\theta)$

Approximation error, $\mathcal{E}_{\text{app}}$ => Choose larger hypothesis space

Neural network, $NN(x;\theta)$, which minimises empirical loss $L(\theta)$

Estimation error, $\mathcal{E}_{\text{est}}$ => More training data / use regularisation

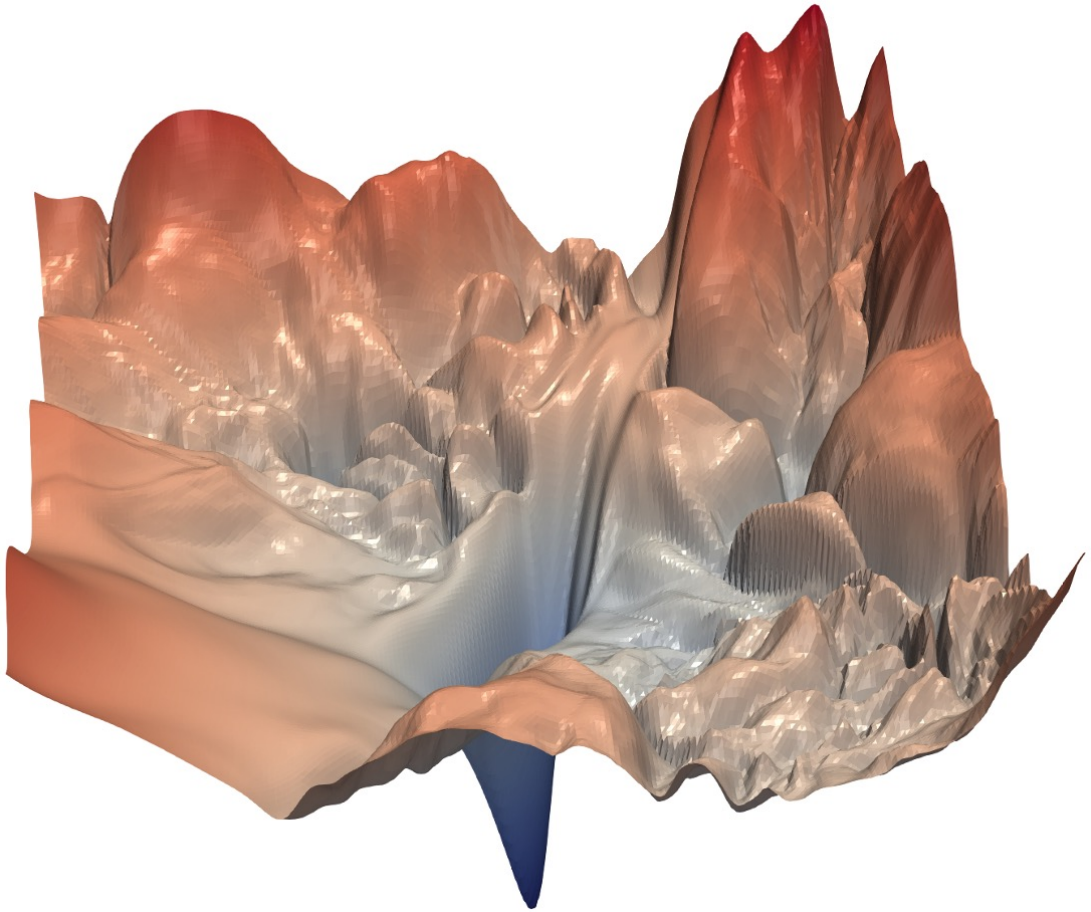
Optimisation error, $\mathcal{E}_{\text{opt}}$ => Improve optimiser

Trained neural network, $NN(x;\theta)$

Space of all possible functions that neural network can represent

$$\mathcal{E} = \mathcal{E}_{\text{app}} + \mathcal{E}_{\text{est}} + \mathcal{E}_{\text{opt}}$$

Loss landscape



Li et al, Visualizing the Loss Landscape of Neural Nets, NeurIPS (2018)

2D visualization of the loss function

$$l(\alpha, \beta) = L(\theta^* + \alpha u + \beta v)$$

Where u and v are 2 randomly sampled directional vectors in the parameter space

Loss surface for ResNet-56 without skip connections trained on CIFAR-10 (natural images)

Loss function is very high dimensional and contains many **local minima**

Vanishing gradients

Problem 1): Loss function is **flat everywhere**

I.e.

$$\frac{\partial L(\theta)}{\partial \theta} = 0$$

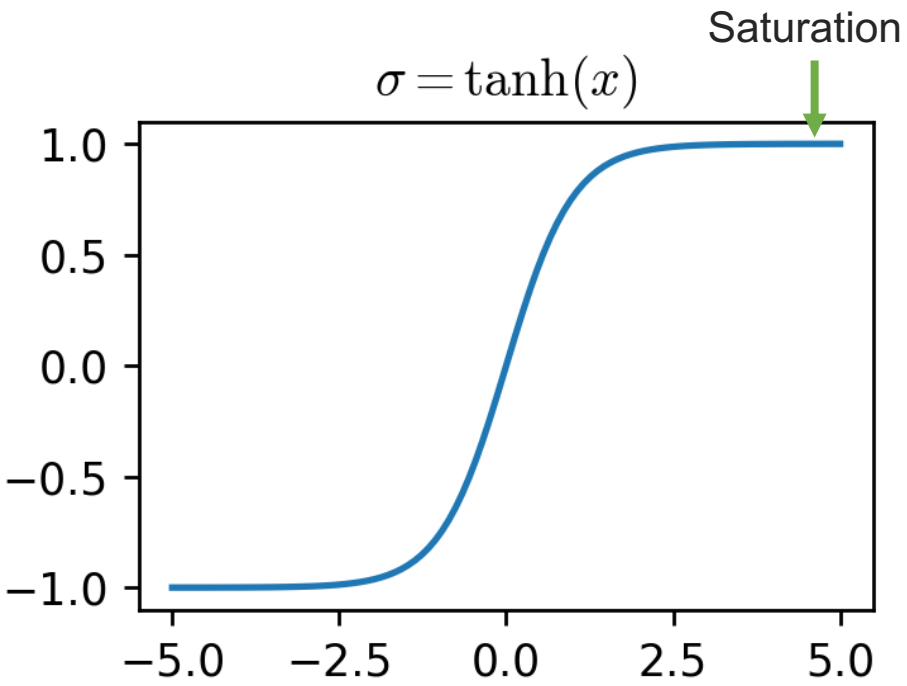
Why is this possible?

Consider the tanh activation function:

$$\sigma'(x) \to 0 \text{ as } |x| \to \infty$$

If $|x|$ is large, derivative is zero, for any value of x!

This means we must **normalise** the inputs and outputs of the network (i.e., training data) between (approximately) [-1,1]



Vanishing gradients

But, even with properly normalised inputs, the loss function can still be flat!

Consider the output of a **single neuron** with an identity activation function in a MLP:

$$y_j = \sum_i^{N_{\text{in}}} w_{ji} x_i$$

Then imagine that the inputs (x_i) and weights (w_{ji}) are continuous random variables, such that y_j is also a random variable. Then

$$\text{Var}(y_j) = \text{Var}\left(\sum_i^{N_{\text{in}}} w_{ji} x_i\right)$$

Then assume 1) the mean of w_{ji} and x_i is zero, 2) w_{ji} and x_i are all independent, 3) variance is the same across all input elements i, and then by using basic properties of variance, it can be shown

$$\text{Var}(y) = N_{\text{in}} \text{Var}(w) \text{Var}(x)$$

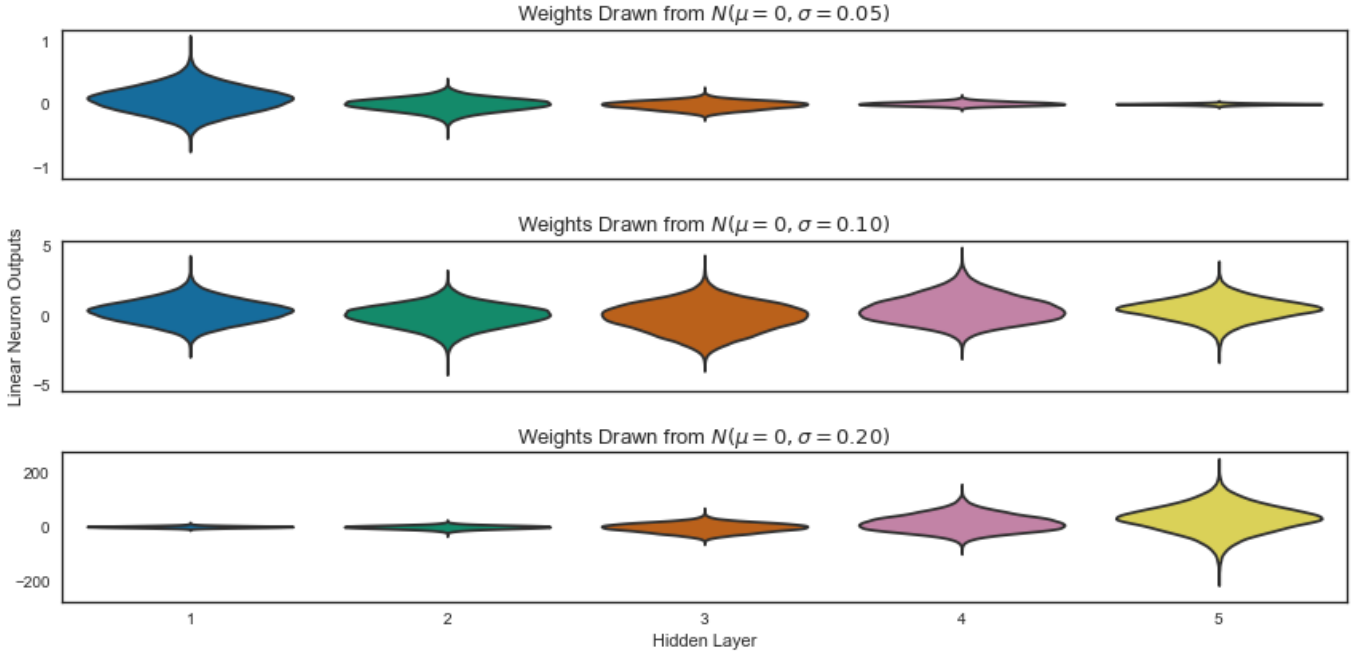
Thus, the **variance** of each neuron's **output** depends on the **number of input neurons** and the variance of their weights

Vanishing gradients

Consider training a MLP with 5 hidden layers, each with 100 hidden units and no activation function

1) Ensure the input to the MLP is normalized between [-1,1]
2) Initialise weights from normal distribution
3) Plot histogram of **output values of each hidden layer**:

$$\mathrm{Var}(y) = N_{\mathrm{in}}\mathrm{Var}(w)\mathrm{Var}(x)$$



$\mathrm{Var}(w) < \frac{1}{N_{\mathrm{in}}} \Rightarrow \mathrm{Var}(y)$ decreases exponentially with depth

$\mathrm{Var}(w) = \frac{1}{N_{\mathrm{in}}} \Rightarrow \mathrm{Var}(y)$ is approximately constant

$\mathrm{Var}(w) > \frac{1}{N_{\mathrm{in}}} \Rightarrow \mathrm{Var}(y)$ grows exponentially with depth

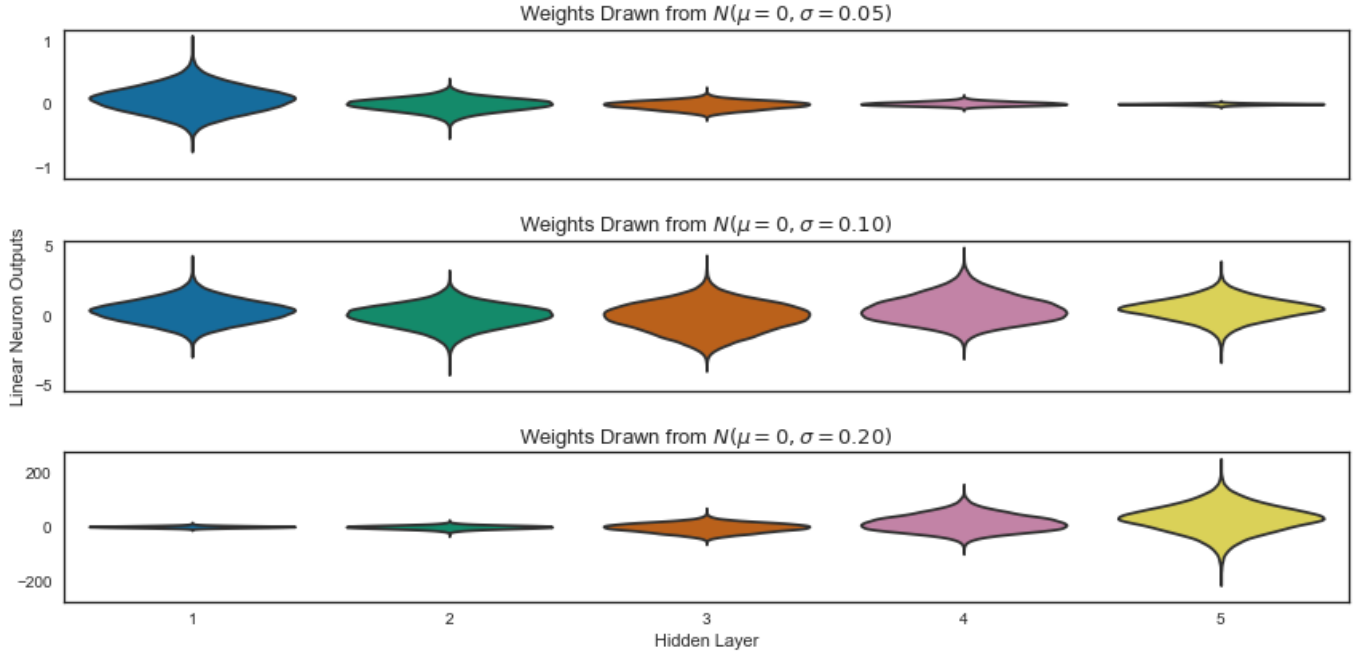
Source: https://intoli.com/blog/neural-network-initialization/

Vanishing gradients

Consider training a MLP with 5 hidden layers, each with 100 hidden units and no activation function

1) Ensure the input to the MLP is normalized between [-1,1]
2) Initialise weights from normal distribution
3) Plot histogram of **output values of each hidden layer**:

$$\mathrm{Var}(y) = N_{\mathrm{in}}\mathrm{Var}(w)\mathrm{Var}(x)$$



If all output values $\Rightarrow 0$, gradient of loss also tends to zero

If all output values $\Rightarrow \infty$, the tanh activation function saturates $\Rightarrow$ gradient of loss also tends to zero

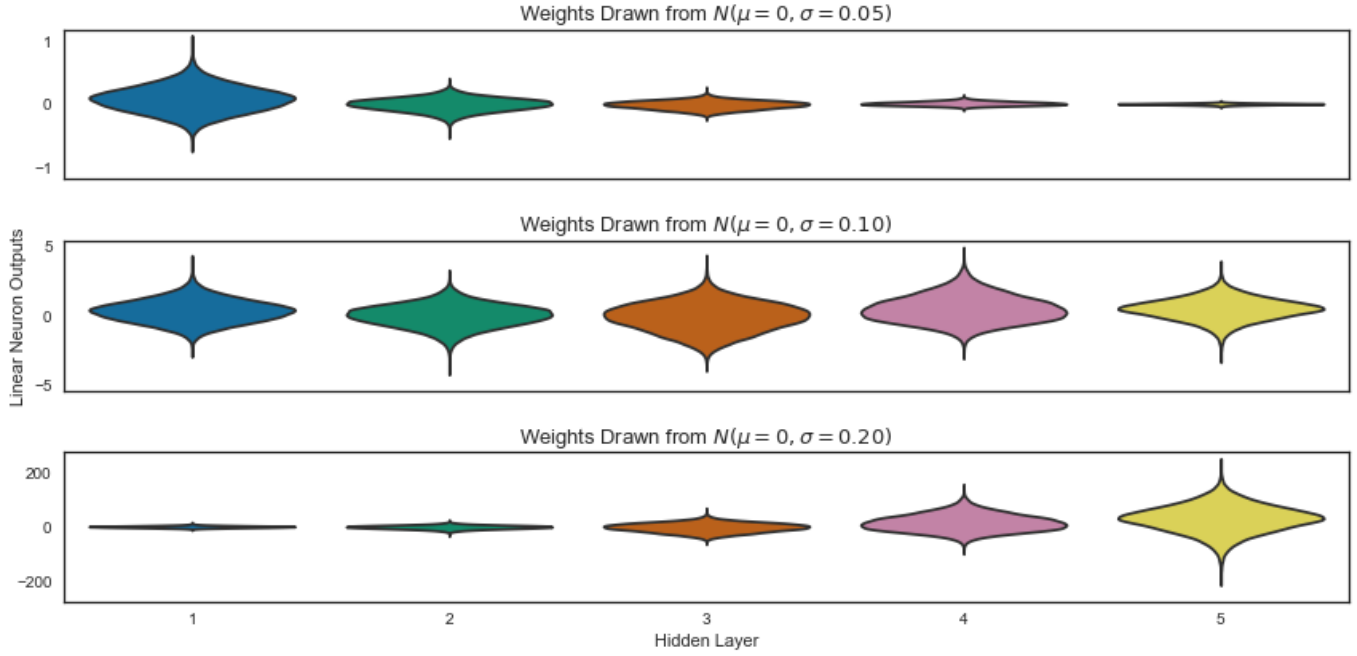
Source: https://intoli.com/blog/neural-network-initialization/

Vanishing gradients

Consider training a MLP with 5 hidden layers, each with 100 hidden units and no activation function

1) Ensure the input to the MLP is normalized between [-1,1]
2) Initialise weights from normal distribution
3) Plot histogram of **output values of each hidden layer**:

$$\mathrm{Var}(y) = N_{\mathrm{in}}\mathrm{Var}(w)\mathrm{Var}(x)$$



$\mathrm{Var}(w) = \frac{1}{N_{\mathrm{in}}} \Rightarrow$ LeCun initialisation

LeCun et al, Neural Networks: Tricks of the Trade, Springer (2012)

Source: https://intoli.com/blog/neural-network-initialization/

Vanishing gradients

LeCun initialisation:

$$\mathrm{Var}(w) = \frac{1}{N_{\mathrm{in}}}$$

Next consider the evolution of the gradient of the loss function during backpropagation of a single layer:

$$y_j = \sum_i^{N_{\mathrm{in}}} w_{ji} x_i$$

$$\frac{\partial L}{\partial x_i} = \frac{\partial L}{\partial \boldsymbol{y}} \frac{\partial \boldsymbol{y}}{\partial x_i} = \frac{\partial L}{\partial \boldsymbol{y}} \left(w_{0i}, \dots, w_{ji}\right)^{\mathrm{T}}$$

Then again assuming all elements are independent random variables with zero mean a similar result can be shown:

$$\mathrm{Var}\left(\frac{\partial L}{\partial x}\right) = N_{\mathrm{out}} \mathrm{Var}(w) \mathrm{Var}\left(\frac{\partial L}{\partial y}\right)$$

This suggests for stable gradient flow

$$\mathrm{Var}(w) = \frac{1}{N_{\mathrm{out}}}$$

A good compromise between stable forward and gradient flow is the **harmonic mean**, i.e.

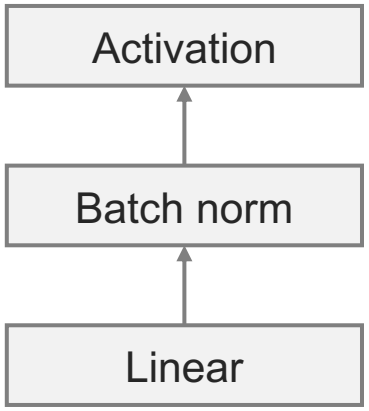
$$\mathrm{Var}(w) = \frac{2}{N_{\mathrm{in}} + N_{\mathrm{out}}}$$

= Xavier (Glorot) initialization

Glorot et al, Understanding the difficulty of training deep feedforward neural networks, AISTATS (2010)

Batch normalisation

Idea: insert extra layers that **dynamically** normalize the outputs of each hidden layer, by using statistics from the **batch** of training data



Ioffe et al, Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift, PMLR (2015)

$$\boldsymbol{y}(\boldsymbol{x};\boldsymbol{\gamma},\boldsymbol{\beta}) = \boldsymbol{\gamma}\frac{\boldsymbol{x}-\boldsymbol{\mu_B}}{\sqrt{\boldsymbol{\sigma_B^2}+\epsilon}} + \boldsymbol{\beta}$$

Where

$$\boldsymbol{\mu_B} = \frac{1}{B}\sum_i^B \boldsymbol{x}_i\,, \qquad \boldsymbol{\sigma_B^2} = \frac{1}{B}\sum_i^B (\boldsymbol{x}_i - \boldsymbol{\mu_B})^2$$

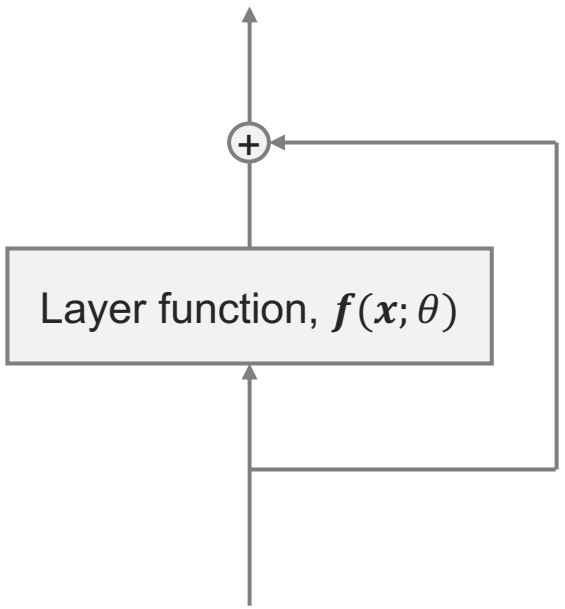
And $\boldsymbol{\gamma}$ and $\boldsymbol{\beta}$ are learnable vectors

- Allows network to achieve good results over wide range of initialization strategies
- Is also considered a regularisation strategy
- Must be careful to fix batch norm parameters during test-time inference (keep running estimates of $\boldsymbol{\mu_B}$, $\boldsymbol{\sigma_B^2}$)

Residual networks (ResNets)

Idea: each layer learns a **residual correction** to its input

Aka: **skip connections**



He et al, Deep Residual Learning for Image Recognition, CVPR (2015)

$$\boldsymbol{y}(\boldsymbol{x};\theta) = \boldsymbol{x} + \boldsymbol{f}(\boldsymbol{x};\theta)$$

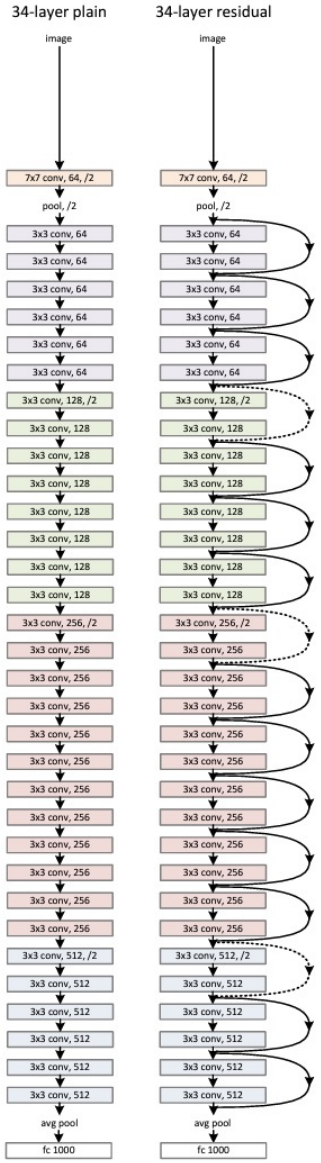
Consider backpropagating through this residual layer:

$$\frac{\partial L}{\partial \boldsymbol{x}} = \frac{\partial L}{\partial \boldsymbol{y}}\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}} = \frac{\partial L}{\partial \boldsymbol{y}}\left(\mathbf{1} + \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}}\right)$$

Thus if $\boldsymbol{f} = \mathbf{0}$, gradient flow is still preserved

- ResNets learn the “ideal” depth of a network; allows easy training of arbitrarily large numbers of layers (100+)
- Great for applications where input is like output (e.g. super resolution / segmentation)
- For later: ResNets have strong similarities with ODE solvers

Example residual networks



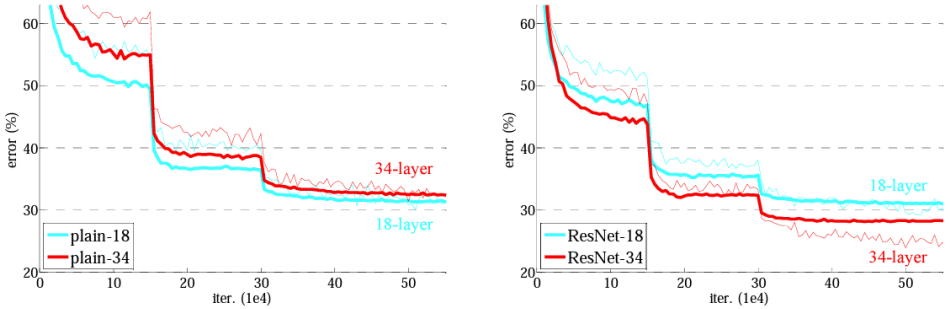
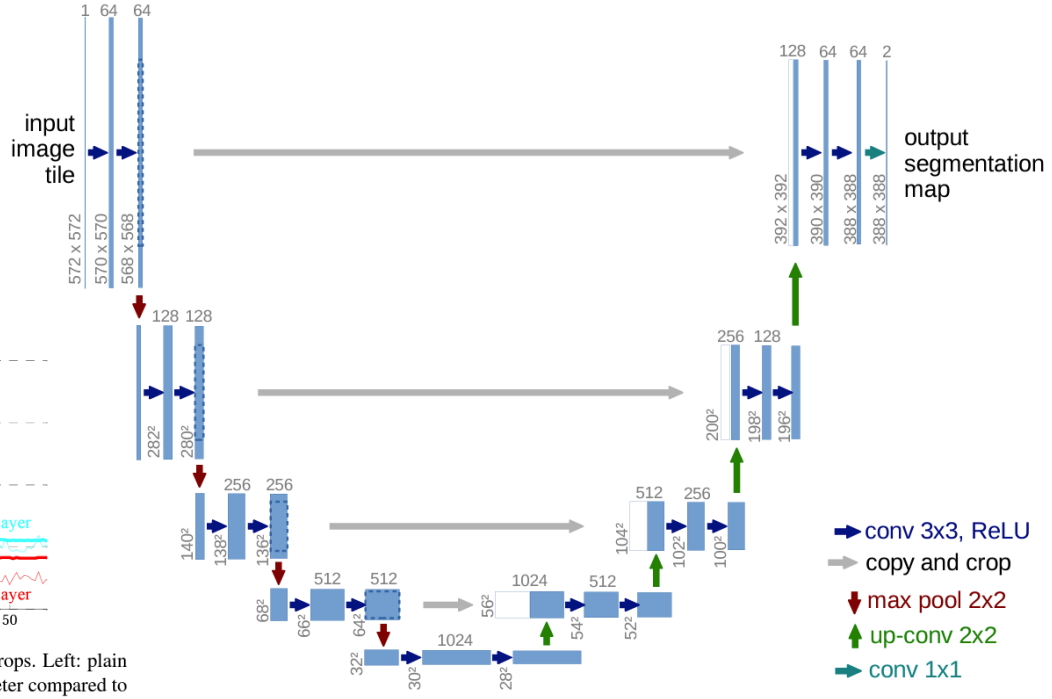


Figure 4. Training on **ImageNet**. Thin curves denote training error, and bold curves denote validation error of the center crops. Left: plain networks of 18 and 34 layers. Right: ResNets of 18 and 34 layers. In this plot, the residual networks have no extra parameter compared to their plain counterparts.

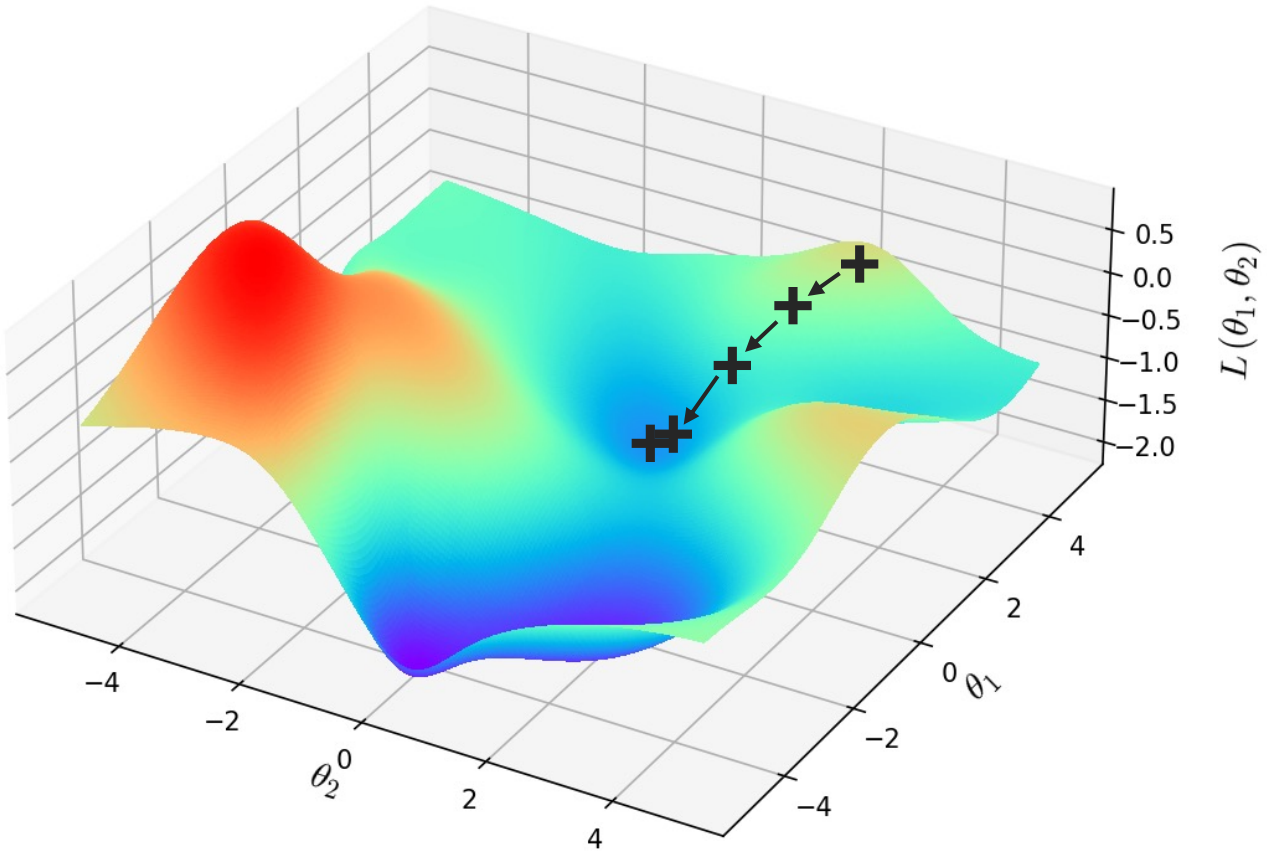
He et al, Deep Residual Learning for Image Recognition, CVPR (2015)



Ronneberger et al, U-Net: Convolutional Networks for Biomedical Image Segmentation, MICCAI (2015)

Gradient descent

- Problem 2): Loss function has lots of **local** minima

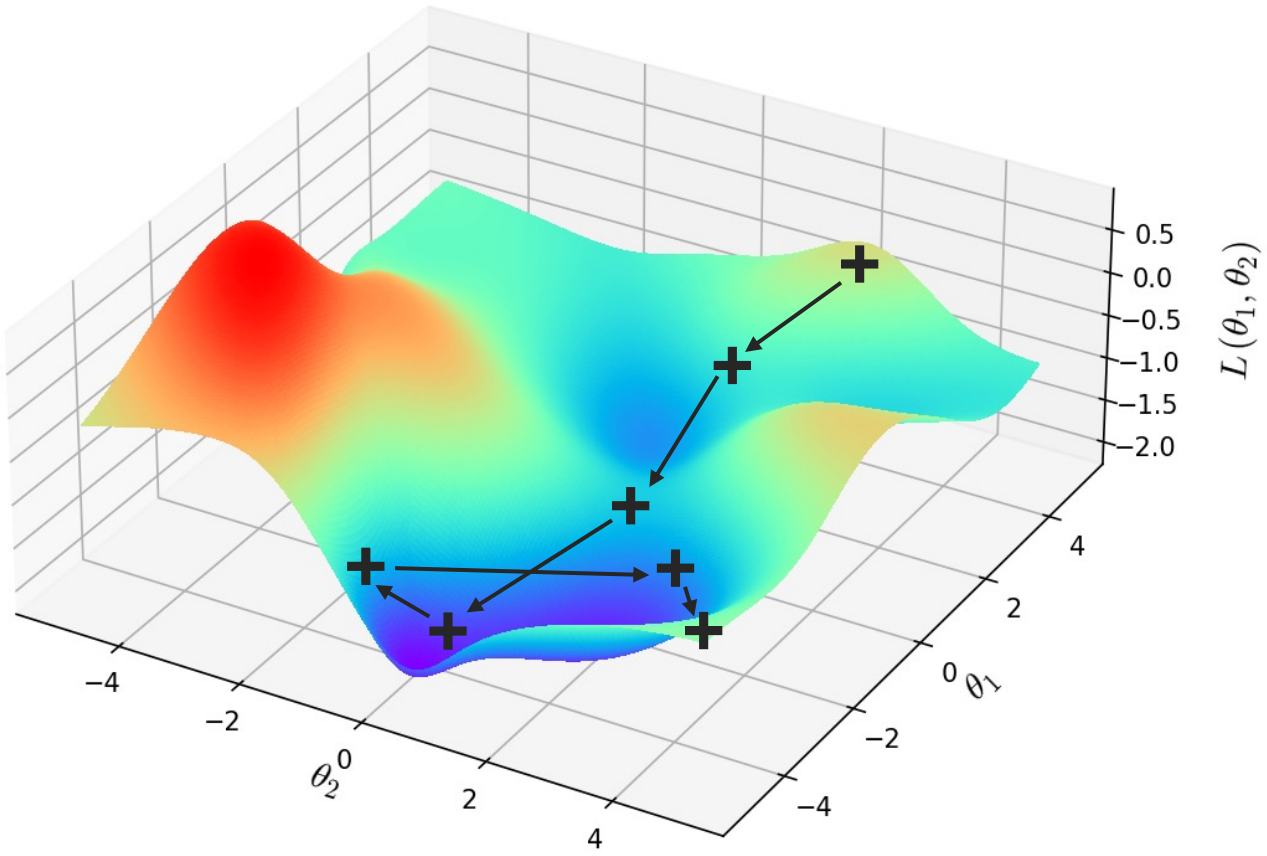


$$L(\theta) = \frac{1}{N}\sum_{i}^{N}(NN(x_i;\theta) - y_i)^2$$

1. Initialise weights randomly
2. Loop:
   1. Compute gradient, $\frac{\partial L(\theta)}{\partial \theta_j}$
   2. Update weights,

$$\theta_j \leftarrow \theta_j - \gamma \frac{\partial L(\theta)}{\partial \theta_j}$$

Gradient descent with large learning rate



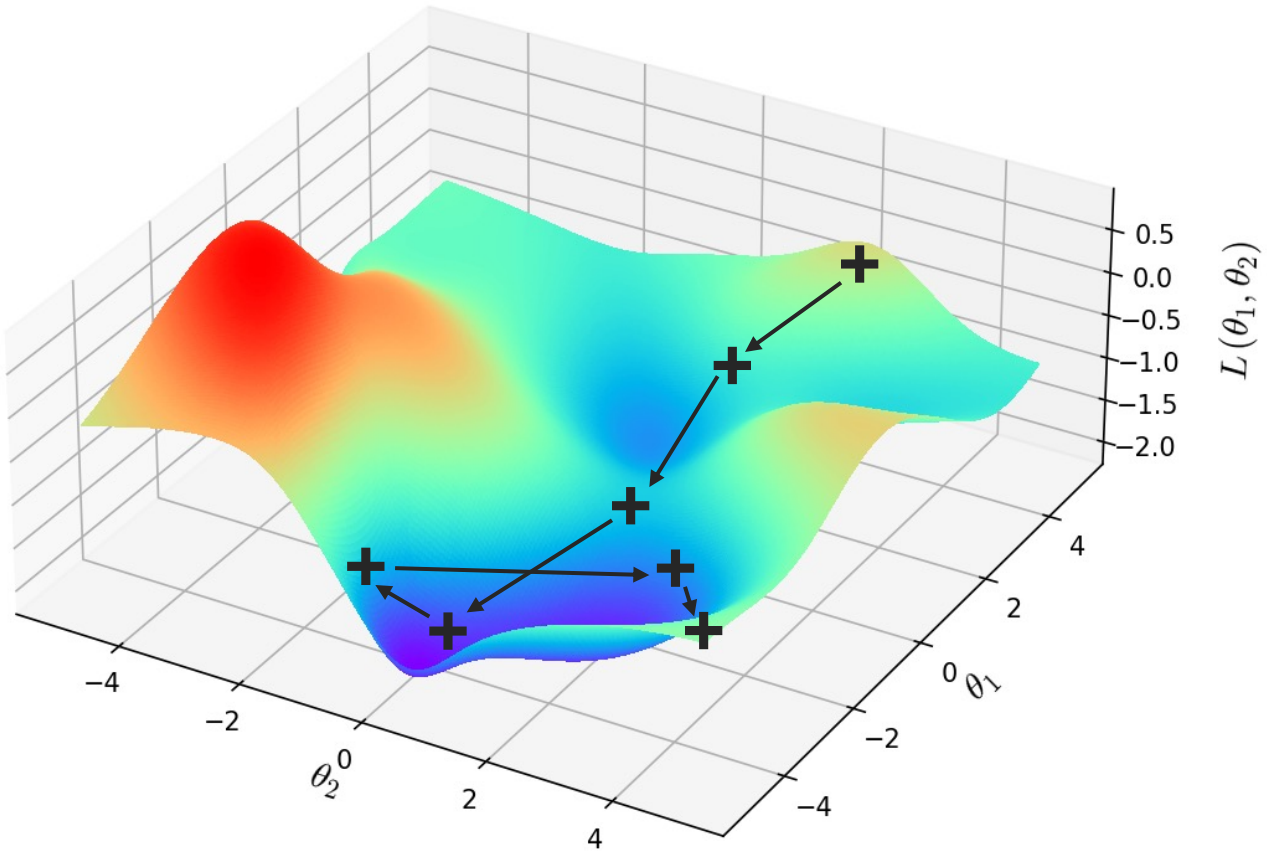
$$L(\theta) = \frac{1}{N}\sum_{i}^{N}(NN(x_i;\theta) - y_i)^2$$

1. Initialise weights randomly
2. Loop:
    1. Compute gradient, $\frac{\partial L(\theta)}{\partial \theta_j}$
    2. Update weights,

$$\theta_j \leftarrow \theta_j - \gamma \frac{\partial L(\theta)}{\partial \theta_j}$$

Gradient descent with large learning rate

- Can escape local minima (and global minima too!)
- Typically reduce learning rate with training step (e.g. exponentially decaying)

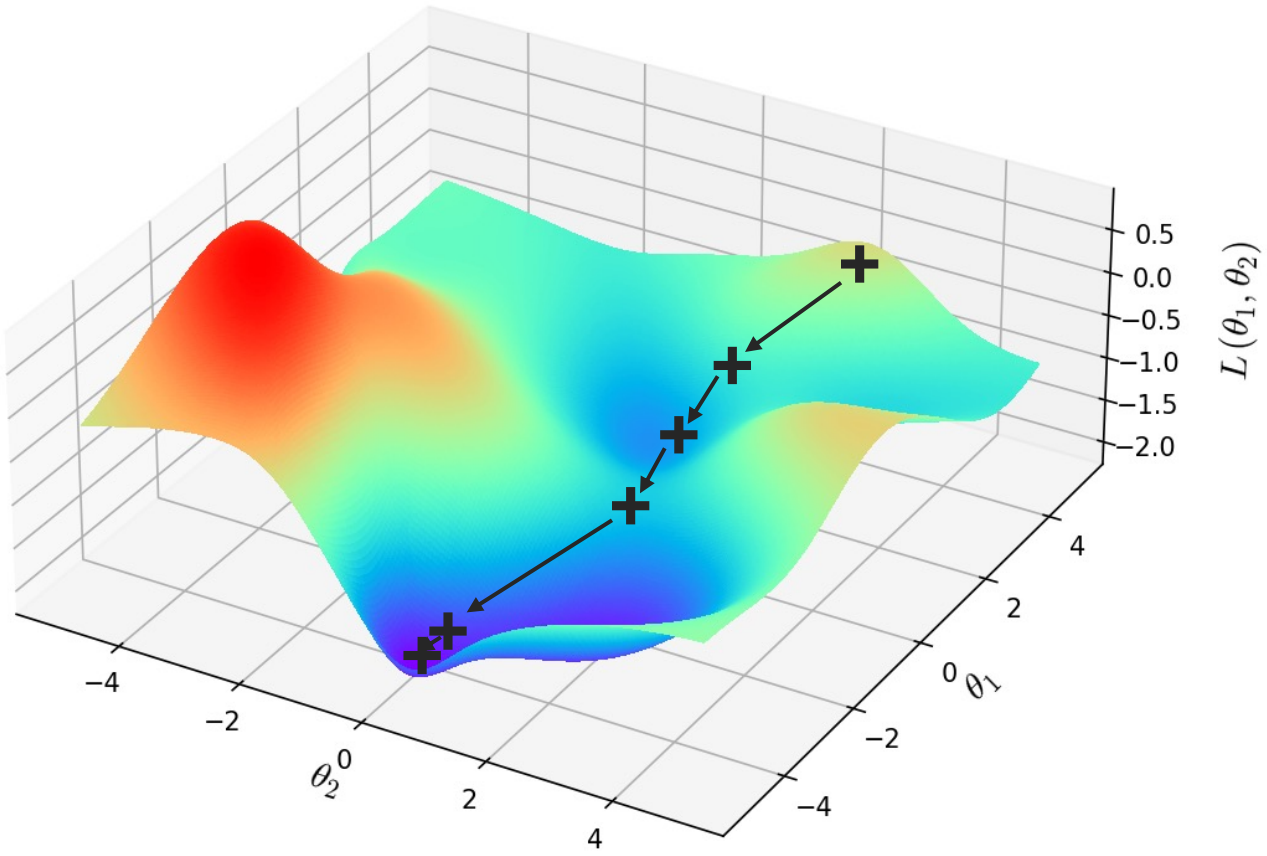


$$L(\theta) = \frac{1}{N}\sum_{i}^{N}(NN(x_i;\theta) - y_i)^2$$

1. Initialise weights randomly
2. Loop:
   1. Compute gradient, $\frac{\partial L(\theta)}{\partial \theta_j}$
   2. Update weights,

$$\theta_j \leftarrow \theta_j - \gamma \frac{\partial L(\theta)}{\partial \theta_j}$$

Gradient descent with adaptive learning rate

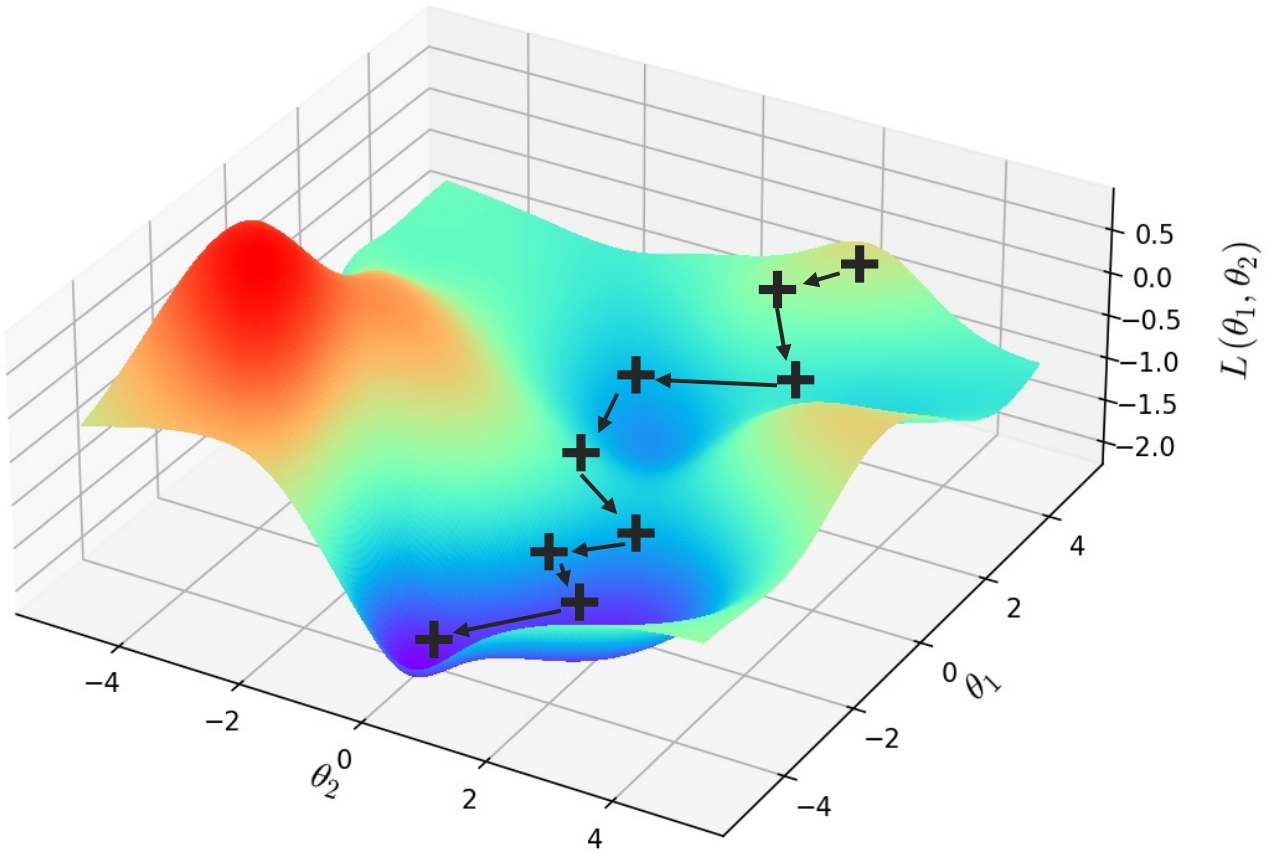


"Intelligent" learning rate

Can adapt learning rate based on

- How large gradient is
- Local curvature
- How fast learning is happening
- Values of weights
- .. many other ideas

(Mini-batch) stochastic gradient descent



$$L(\theta) = \frac{1}{N}\sum_{i}^{N}(NN(x_i;\theta) - y_i)^2$$

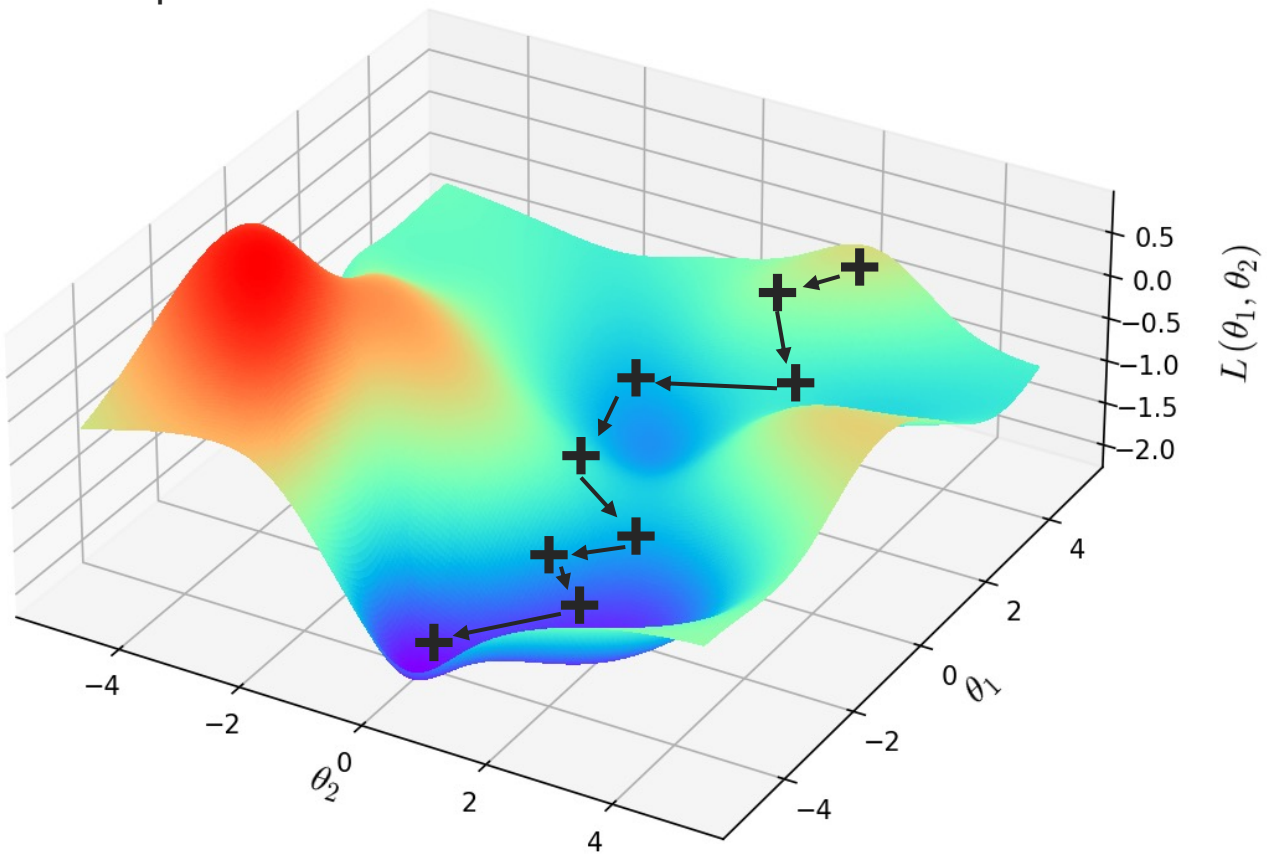
1. Initialise weights randomly
2. Loop:
   1. **Randomly** sample mini-batch (subset) of B training points
   2. Compute **mini-batch** gradient, $\frac{\partial L_b(\theta)}{\partial \theta_j}$
   3. Update weights,

$$\theta_j \leftarrow \theta_j - \gamma \frac{\partial L_b(\theta)}{\partial \theta_j}$$

(Mini-batch) stochastic gradient descent

- Produces noisy (approximate) gradient estimates
- Can escape local minima ✓
- Much **more** efficient gradient computation (batch size of 100s vs millions) ✓
- Widely used in practice

$$L(\theta) = \frac{1}{N}\sum_{i}^{N}(NN(x_i;\theta) - y_i)^2$$



1. Initialise weights randomly
2. Loop:
   1. **Randomly** sample mini-batch (subset) of B training points
   2. Compute **mini-batch** gradient, $\frac{\partial L_b(\theta)}{\partial \theta_j}$
   3. Update weights,

$$\theta_j \leftarrow \theta_j - \gamma \frac{\partial L_b(\theta)}{\partial \theta_j}$$

Adam optimiser

Adaptive moment estimation

Kingma et al, Adam: A Method for Stochastic Optimization, ICLR, (2015)

$$L(\theta) = \frac{1}{N}\sum_{i}^{N}(NN(x_i;\theta) - y_i)^2$$

1. Initialise θ randomly, and $m_0 \leftarrow 0$ (first moment) $v_0 \leftarrow 0$ (second moment)
2. For $t = 1$ to … :
   1. Compute gradient, $\frac{\partial L}{\partial \theta_{t-1}}$
   2. Update moments, $m_t \leftarrow \beta_1 m_{t-1} + (1-\beta_1)\frac{\partial L}{\partial \theta_{t-1}}, \quad v_t \leftarrow \beta_2 v_{t-1} + (1-\beta_2)\left(\frac{\partial L}{\partial \theta_{t-1}}\right)^2$
   3. Apply bias correction, $\widehat{m}_t \leftarrow \frac{m_t}{1-\beta_1^t}, \quad \hat{v}_t \leftarrow \frac{v_t}{1-\beta_2^t}$
   4. Update weights,

$$\theta_t \leftarrow \theta_{t-1} - \gamma \frac{\widehat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

Typically, $\beta_1 = 0.9, \;\; \beta_2 = 0.999$

Adam optimiser

Idea 1: keep some **momentum** when moving in parameter space

$$m_t \leftarrow \beta_1 m_{t-1} + (1-\beta_1)\frac{\partial L}{\partial \theta}_{t-1}$$

β_1 = “forgetting factor”
Essentially computes (exponentially-weighted) moving average of gradient

Idea 2: **adapt** the step size (per parameter) based on the **magnitude** of the gradient, again using some momentum

$$v_t \leftarrow \beta_2 v_{t-1} + (1-\beta_2)\left(\frac{\partial L}{\partial \theta}_{t-1}\right)^2$$

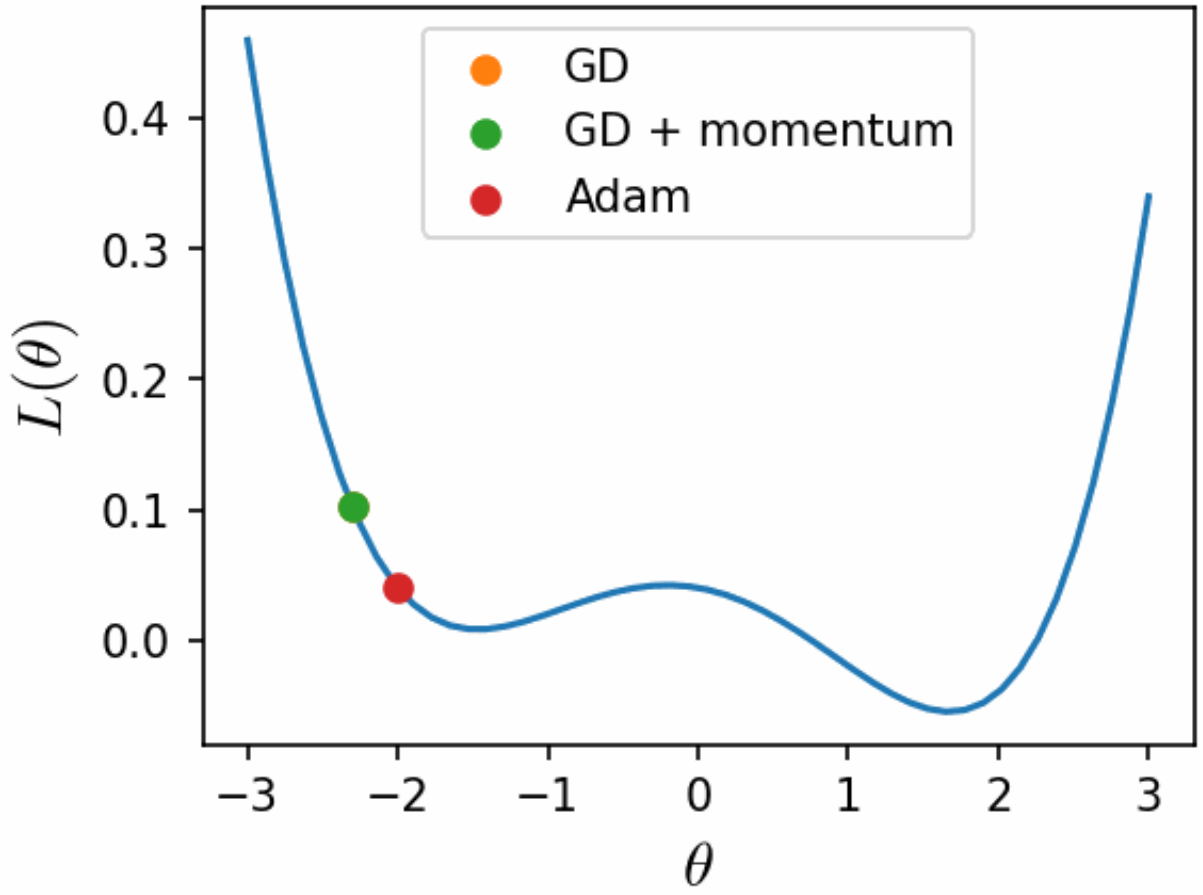
“Normalises” step size

$$\theta_t \leftarrow \theta_{t-1} - \gamma \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

Note - the bias correction gives a more accurate estimate of the moving average (removes effect of initial value)

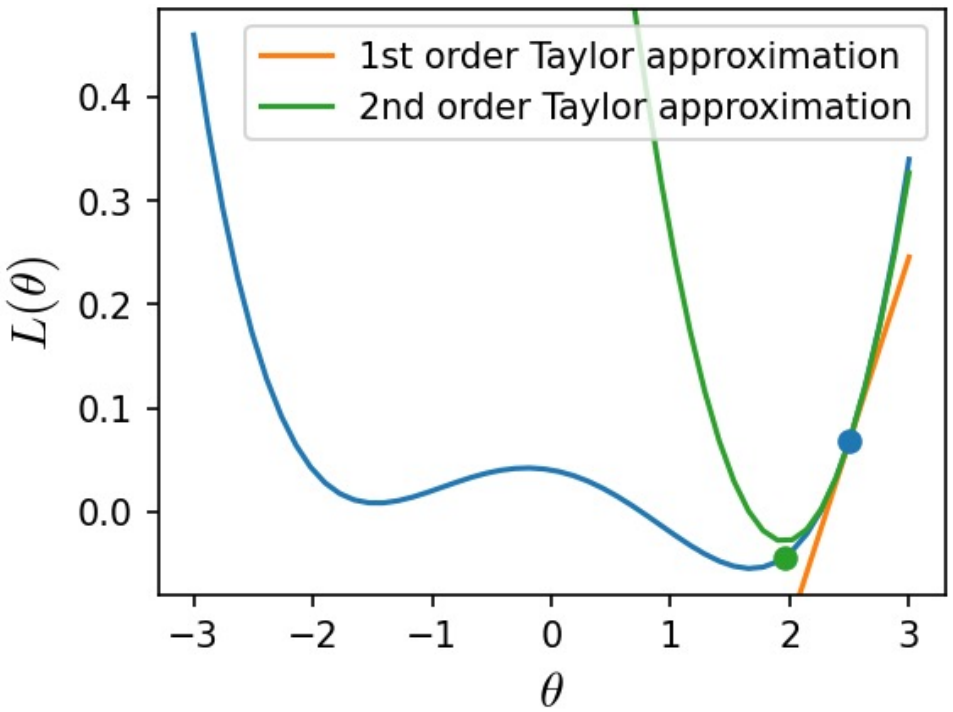
Adam optimiser

Widely used in practice!



Higher-order optimisation

Make use of **higher-order** derivatives (e.g. curvature) of loss function to make bigger steps



Taylor approximation:

$$L(\theta) = L(a) + L'(a)(\theta - a) + \frac{1}{2}L''(a)(\theta - a)^2 + \ \ldots$$

Newton's method:
Instead of following gradient, step towards the local **minima** of the 2nd order approximation

$$\theta \leftarrow \theta - \gamma \frac{L'(\theta)}{L''(\theta)}$$

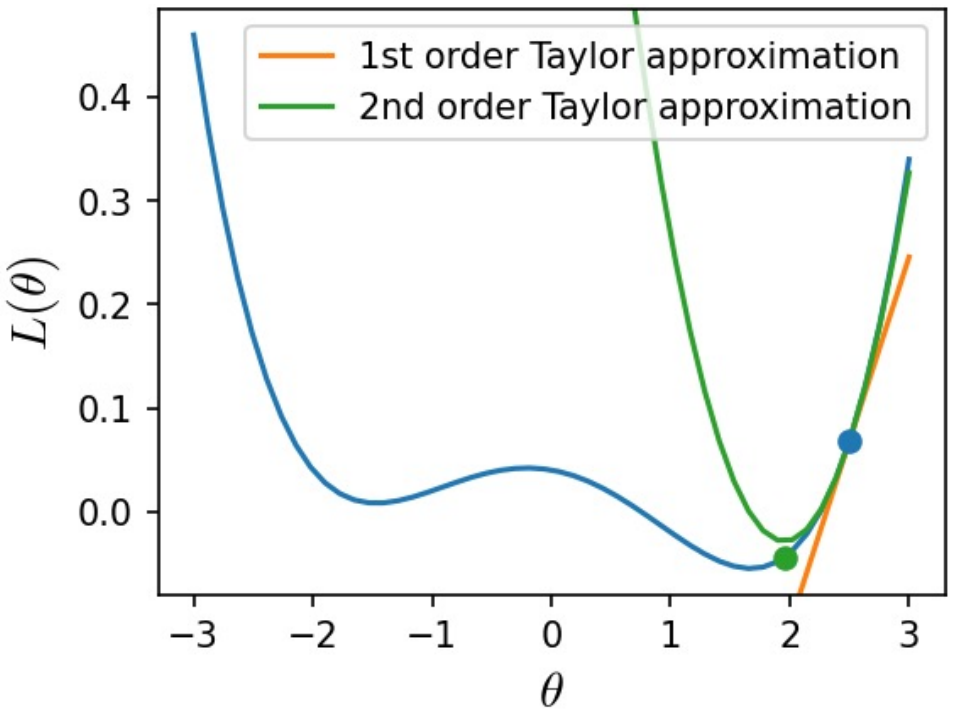
For a multivariate function:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \gamma \left(H(L(\boldsymbol{\theta})\right)^{-1} \nabla L(\boldsymbol{\theta})$$

Where $H(L(\boldsymbol{\theta}))$ is the **Hessian** matrix

Higher-order optimisation

Make use of **higher-order** derivatives (e.g. curvature) of loss function to make bigger steps



$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \gamma\left(H(L(\boldsymbol{\theta}))\right)^{-1}\nabla L(\boldsymbol{\theta})$$

Note

- It is usually prohibitively expensive to compute the Hessian $(p \times p)$ and its inverse
- So most modern methods (e.g. L-BFGS) use approximations
- These are called **quasi-Newton** methods

ChatGPT

Training dataset: a sequence of tokens (e.g. all of Wikipedia)

$$D = \{x_1, \dots, x_n\}$$

Input to network: sub-sequence of D (e.g. single sentence)

$$x_{i-k}, \dots, x_{i-1}$$

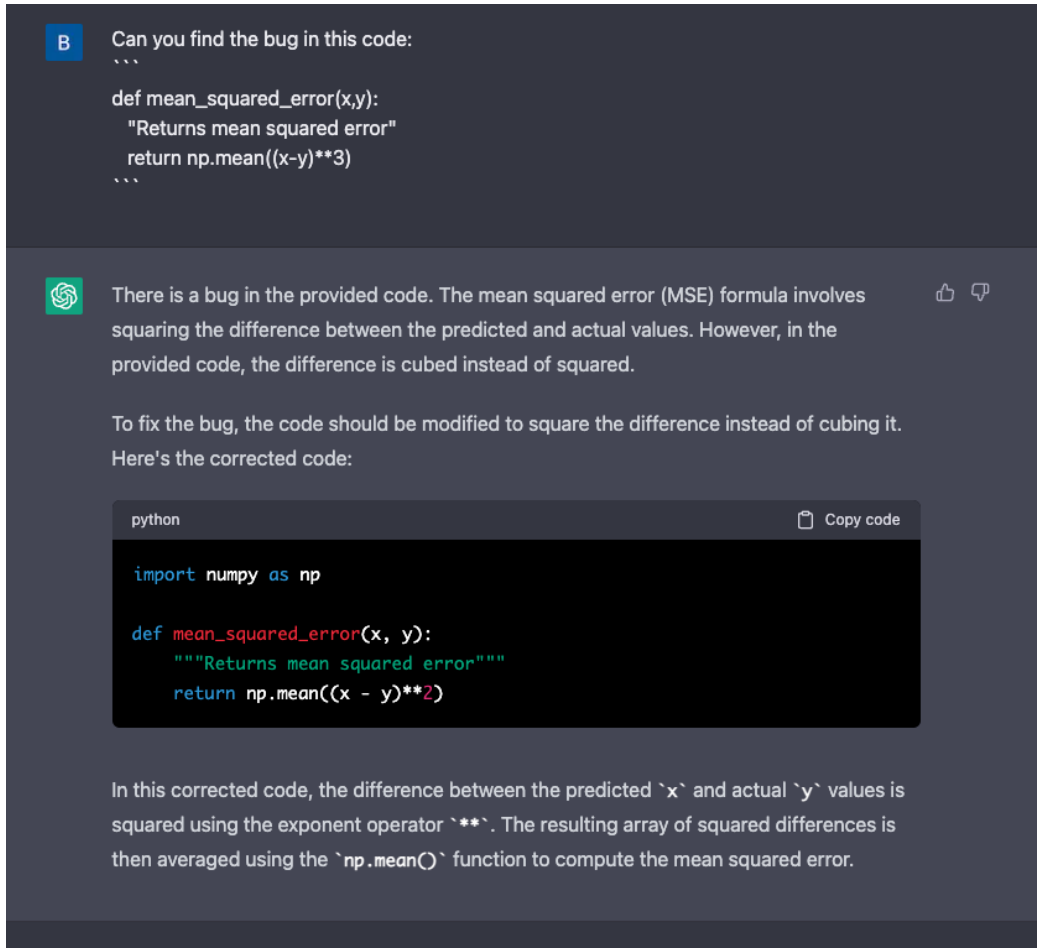
Output of network: probability of next token (use softmax output layer)

$$P(x_i | x_{i-k}, \dots, x_{i-1}, \theta) = NN(x_{i-k}, \dots, x_{i-1}, \theta)_{x_i}$$

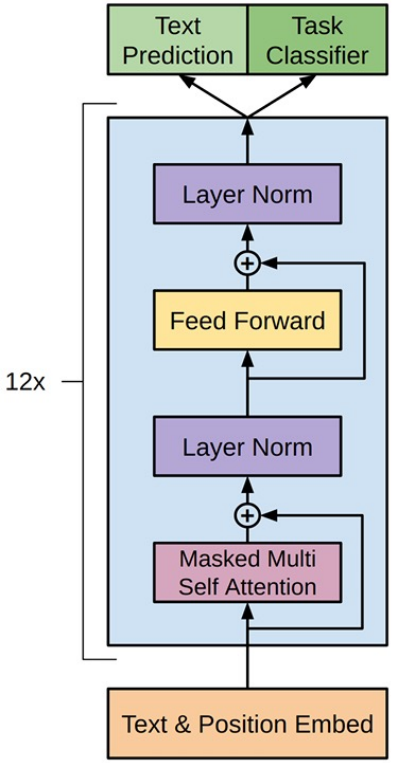
Loss function: cross entropy loss (classification loss)

$$L(\theta) = \sum_i^N \log NN(x_{i-k}, \dots, x_{i-1}, \theta)_{x_i}$$

Note: to generate multiple tokens as output: feed predictions back to model as inputs
Note: can additionally do supervised learning on this pre-trained model

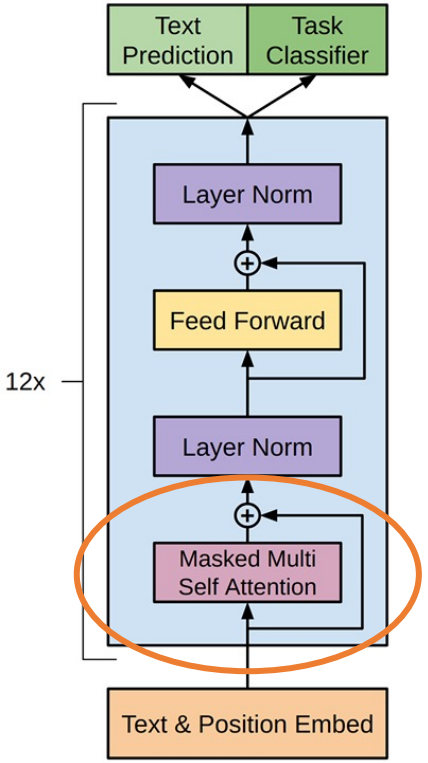


GPT – architecture (transformer)



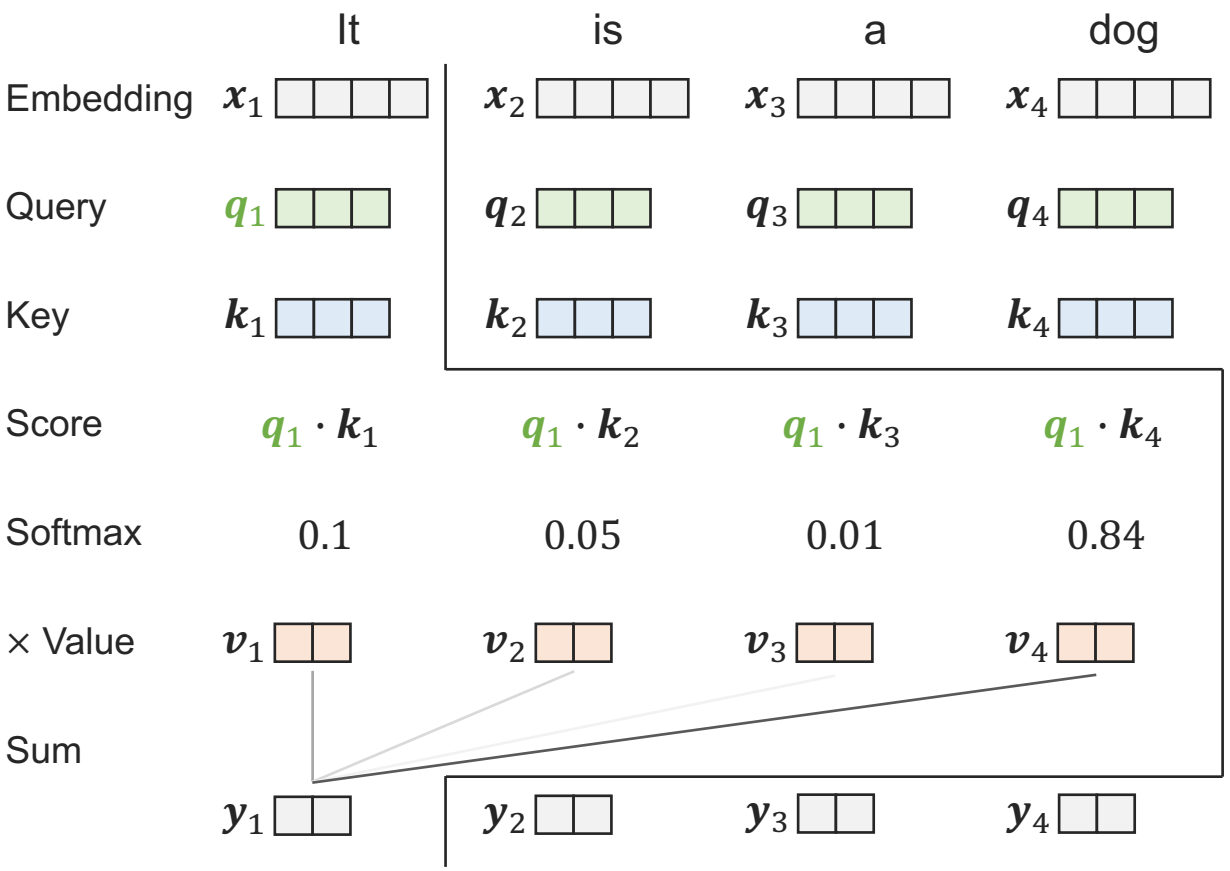
Radford et al, Improving Language Understanding by Generative Pre-Training, ArXiv (2018)

GPT – architecture (transformer)



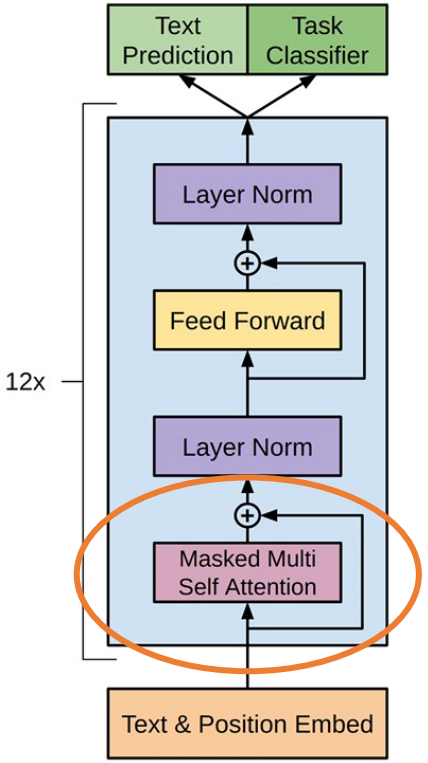
Radford et al, Improving Language Understanding by Generative Pre-Training, ArXiv (2018)

Key idea: **attention** layers
Attention is a **sequence-to-sequence** operation:



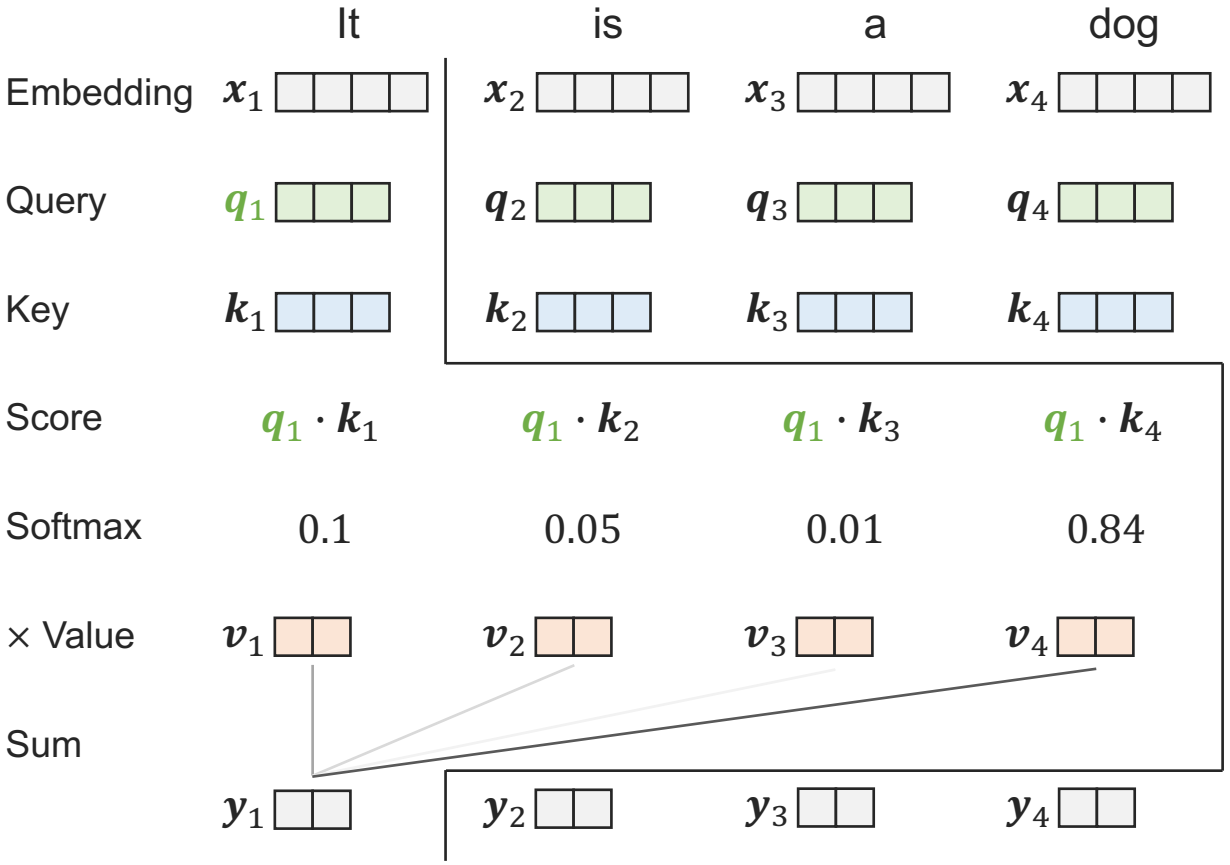
Vaswani et al, Attention Is All You Need, NeurIPS (2017)

GPT – architecture (transformer)



Radford et al, Improving Language Understanding by Generative Pre-Training, ArXiv (2018)

Key idea: **attention** layers
Attention is a **sequence-to-sequence** operation:



$$\boldsymbol{q}_i = W_q \boldsymbol{x}_i$$
$$\boldsymbol{k}_i = W_k \boldsymbol{x}_i$$
$$\boldsymbol{v}_i = W_v \boldsymbol{x}_i$$

$$Y = \mathrm{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

Where $\sqrt{d_k}$ is added for gradient stability

Interested in implementing? See here e.g.: https://github.com/karpathy/minGPT

Vaswani et al, Attention Is All You Need, NeurIPS (2017)

Lecture summary

- Three sources of error:
  - approximation error (hypothesis space)
  - estimation error (training data)
  - optimisation error (optimiser)

- Reduce approximation error -> modify network architecture
- Reduce estimation error -> use regularisation
- Reduce optimisation error -> choose better initialisation / optimiser

- Many strategies exist for all of these