# Course assessment

- To be graded for this course, you need to complete the course Project (no exams)
- Project tasks will be released: late-April
- Submission deadline: mid-June
- Project tasks will be introduced in detail during a Tutorial session
- To be graded, you need to submit solution files for each task and a brief written report
- Will consist of 3 or 4 separate tasks, broadly covering the topics presented in the course

# Deep Learning in Scientific Computing

# Introduction to Deep Learning Part 1

Spring Semester 2023

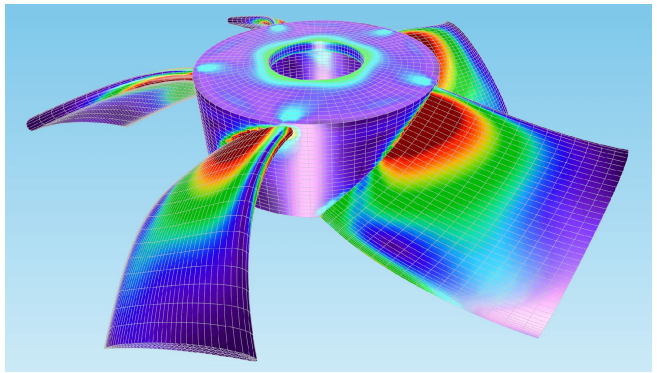
Siddhartha Mishra
Ben Moseley

ETH zürich

# Recap – key scientific tasks

## Simulation

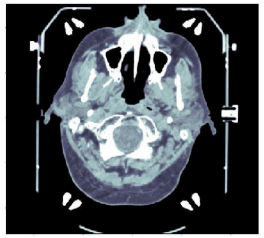
$b = F(a)$



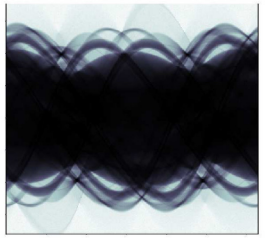
Mesh for finite element method
Source: COMSOL

## Inverse problems

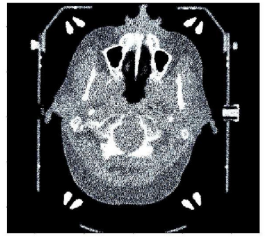
$b = F(a)$



Ground truth computed tomography image



Resulting tomographic data (sinogram)

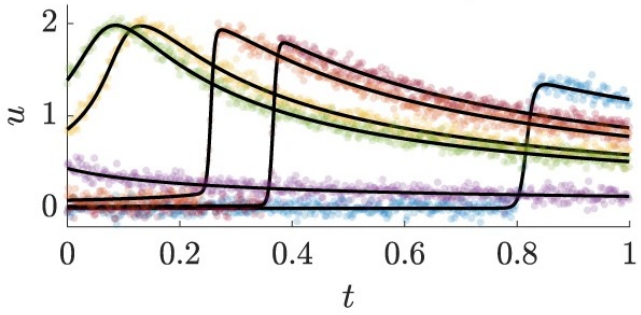


Result of inverse algorithm (filtered back-projection)

Adler et al, Solving ill-posed inverse problems using iterative deep neural networks, Inverse Problems (2017)

## Equation discovery

$b = F(a)$

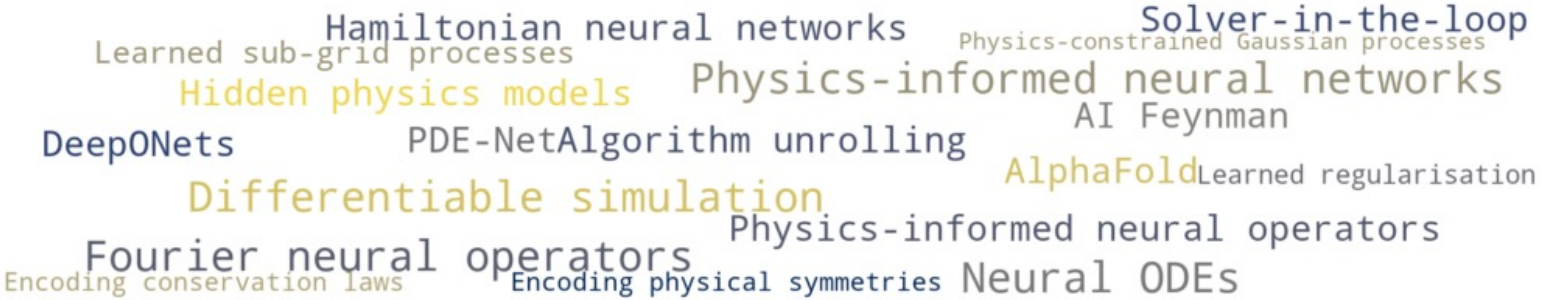


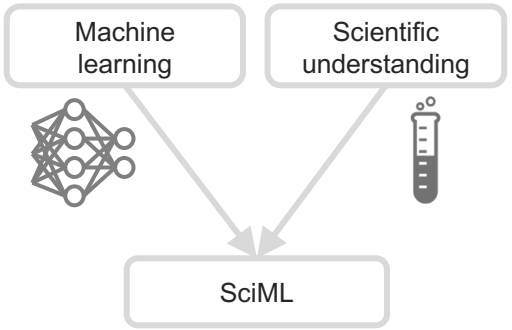

Ground truth: $u_t + uu_x - 0.0032u_{xx} = 0$

Discovered: $u_t + 1.002uu_x - 0.0032u_{xx} = 0$

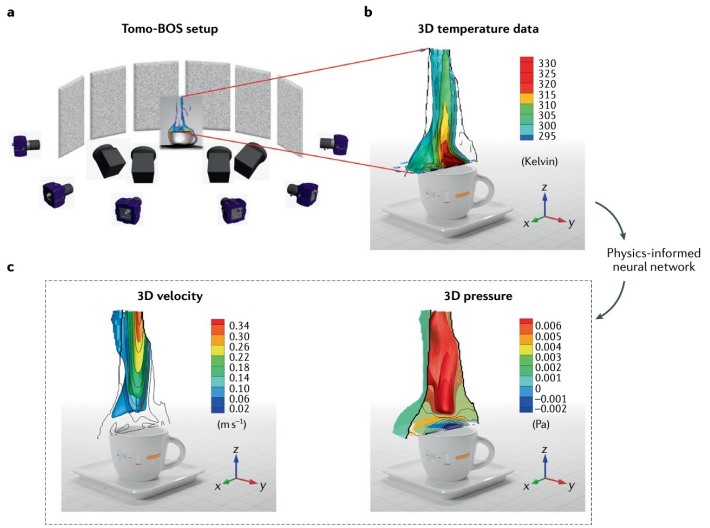
Chen et al, Physics-informed learning of governing equations from scarce data, Nature communications (2021)

# Recap – scientific machine learning

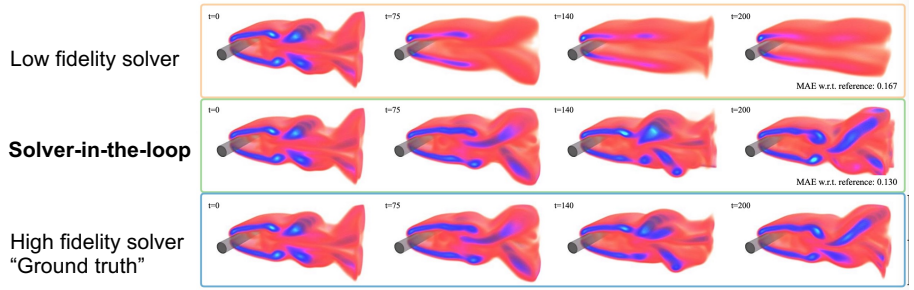



more **powerful, robust, interpretable** models



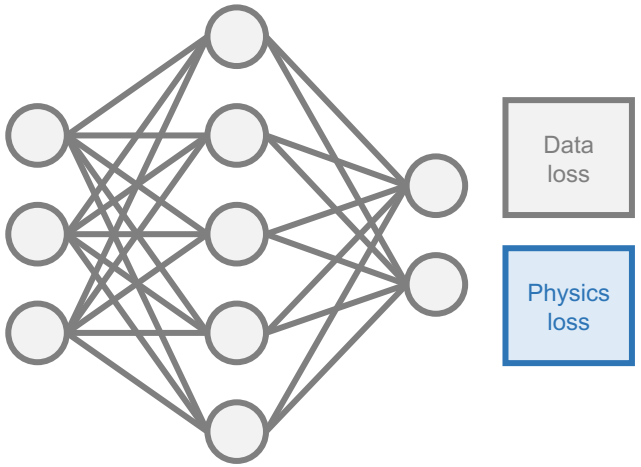

Cai et al, Flow over an espresso cup: inferring 3-D velocity and pressure fields from tomographic background oriented Schlieren via **physics-informed neural networks**, JFM (2021)




Um et al, **Solver-in-the-loop**: Learning from differentiable physics to interact with iterative PDE-solvers, NeurIPS (2020)

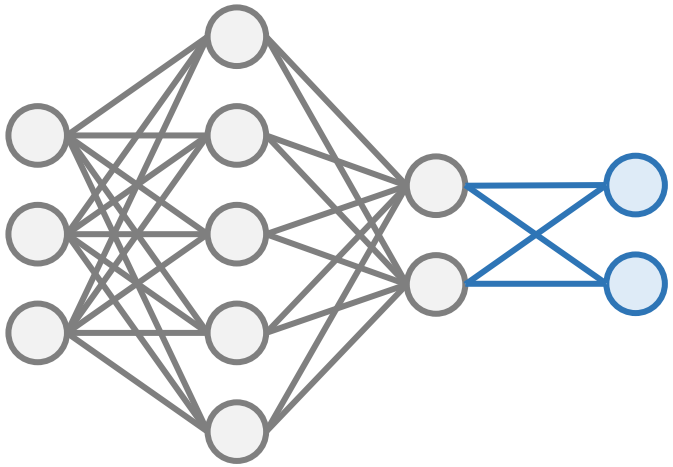
# Recap – scientific machine learning

## Loss function



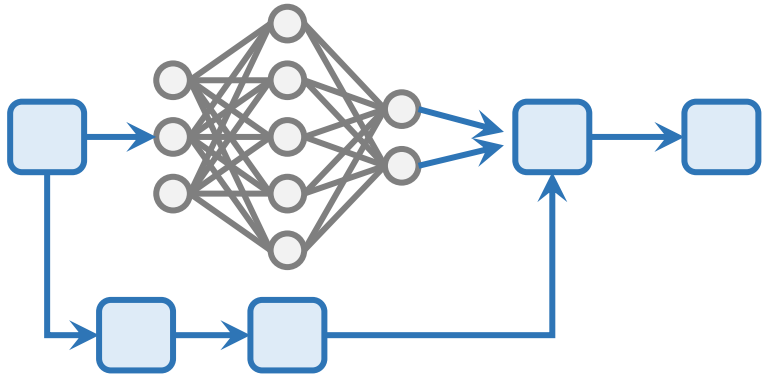

Example:
Physics-informed neural networks
(add governing equations to loss function)

## Architecture



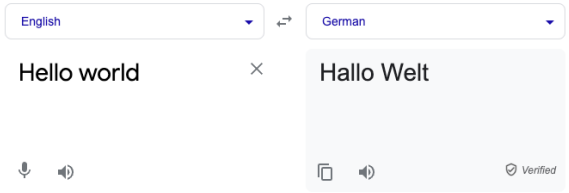
Example:
Encoding symmetries / conservation laws
(e.g. energy conservation, rotational invariance)

## Hybrid approaches

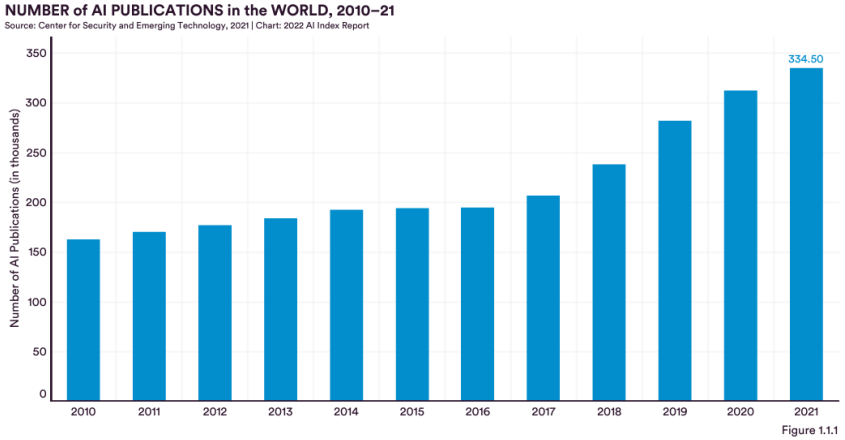


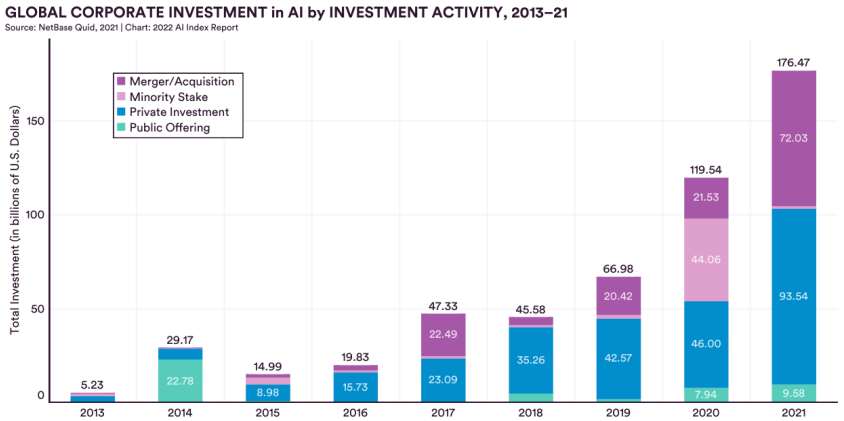
Example:
Neural differential equations
(incorporating neural networks into traditional PDE solvers)

# The rise of deep learning

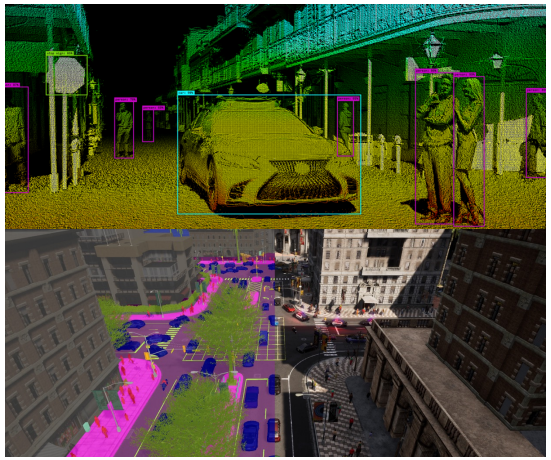



Source: Google Translate




Source: AI Index Report, Stanford University



Source: Machine Learning for Autonomous Driving Workshop, NeurIPS (2022)

Prompt:
“a photograph of an astronaut riding a horse”

Source: Stable Diffusion
Rombach et al, High-Resolution Image Synthesis with Latent Diffusion Models, CVPR (2022)

```python
import datetime

def parse_expenses(expenses_string):
    """Parse the list of expenses and return the list of triples (date, value, currency).
    Ignore lines starting with #.
    Parse the date using datetime.
    Example expenses_string:
        2016-01-02 -34.01 USD
        2016-01-03 2.59 DKK
        2016-01-03 -2.72 EUR
    """
    expenses = []
    for line in expenses_string.splitlines():
        if line.startswith("#"):
            continue
        date, value, currency = line.split(" ")
        expenses.append((datetime.datetime.strptime(date, "%Y-%m-%d"),
                         float(value),
                         currency))
    return expenses
```

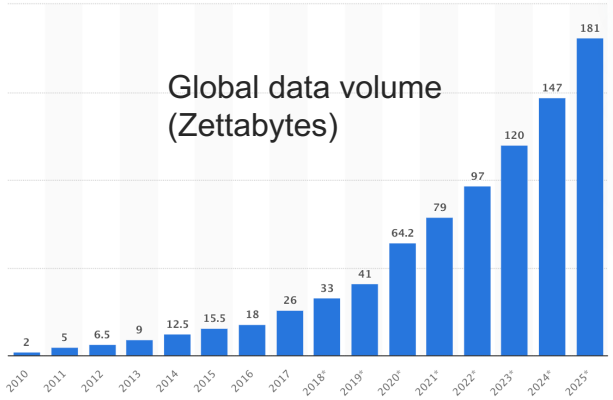
Copilot

Source: GitHub Copilot

# Why now?

Neural networks date back to the 1950's – so why is deep learning so popular today?

## Rapidly increasing amounts of data

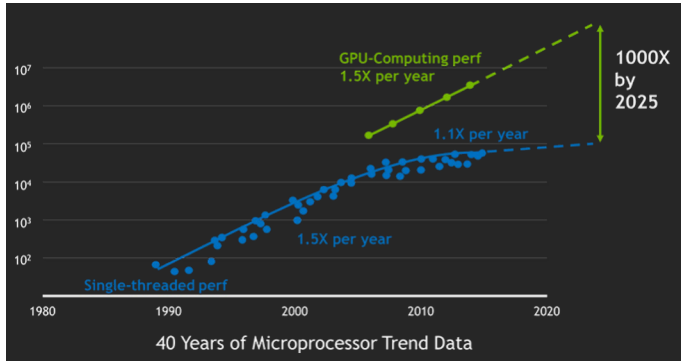



Source: Statista

IMAGENET

WIKIPEDIA
The Free Encyclopedia

## Hardware improvements




Source: NVIDIA

- Graphical processing units (GPUs)
- Highly optimised for deep learning (massively parallel)

## Software improvements

PyTorch

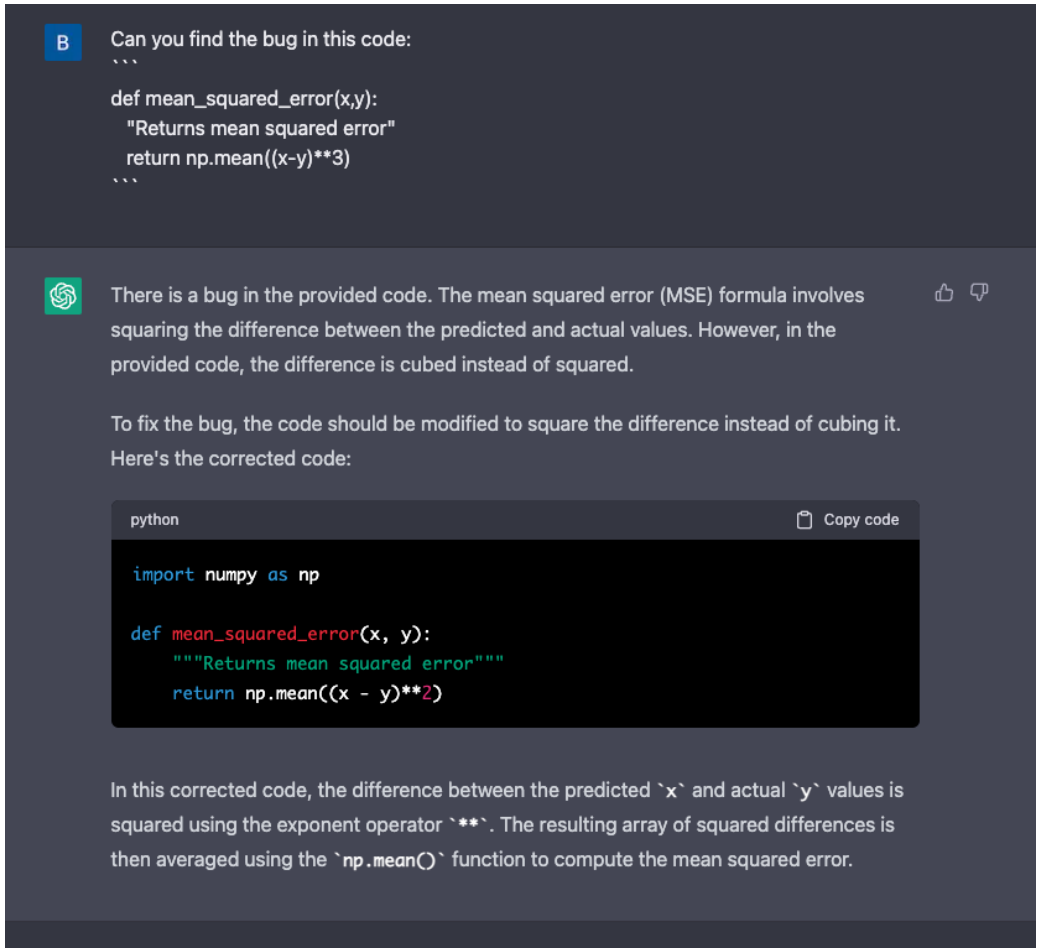
TensorFlow

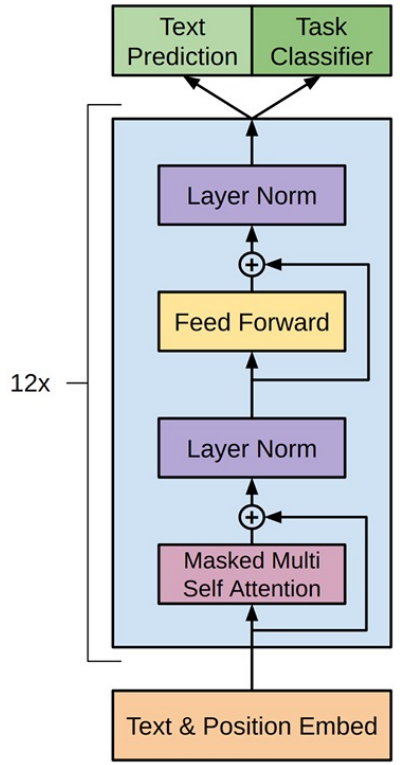
Keras

- Mature deep learning frameworks
- Better training algorithms

# State-of-the-art

- Inside ChatGPT – by end of these two lectures, 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)

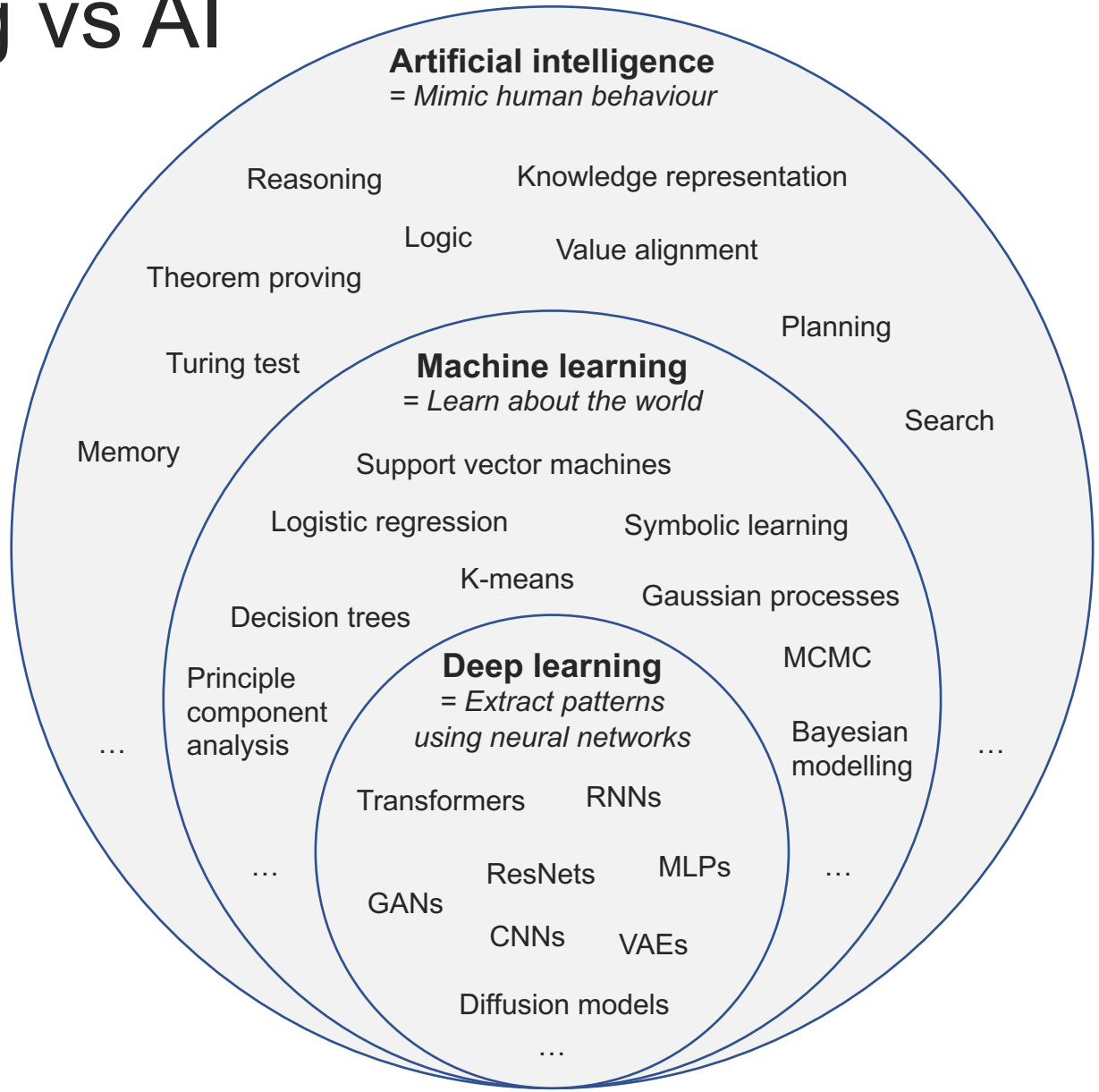
# Course timeline

| 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 |

# Lecture overview

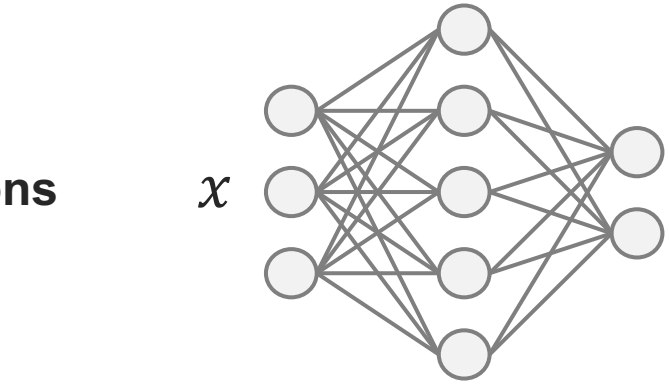
- What is deep learning?
  - Multilayer perceptrons
  - Universal approximation
- Popular deep learning tasks
  - Supervised learning
  - Unsupervised learning
- Training a deep neural network
  - Gradient descent
  - Backpropagation
  - Autodifferentiation
- Live coding – implementing a DNN from scratch

# Deep learning vs AI



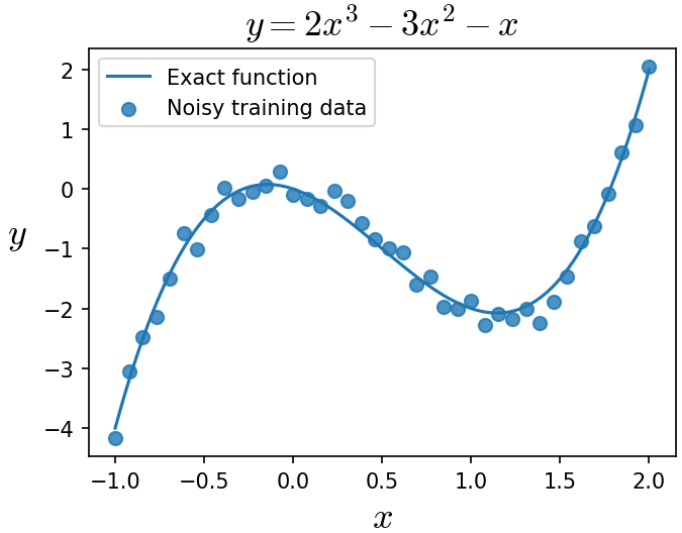
# What is a neural network?

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




$$\hat{y} = NN(x;\theta)$$

Example dataset:




Goal: given training data, find a function (with flexible parameters $\theta$) which approximates the true function,

$$\hat{y} = NN(x;\theta) \approx y(x)$$

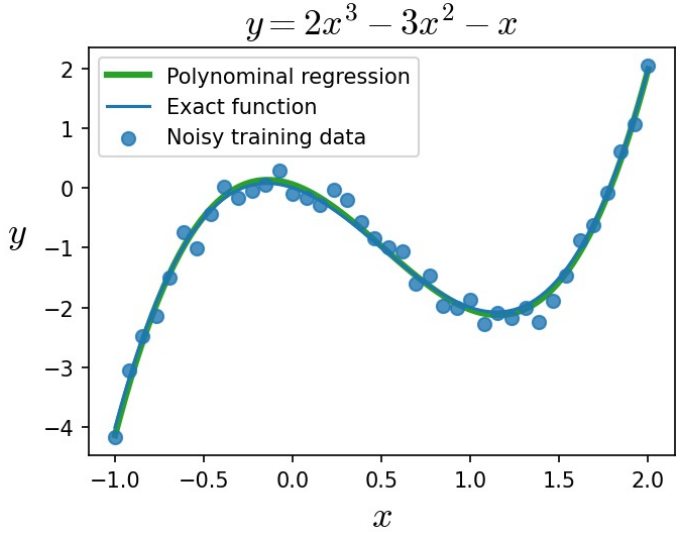
# Function fitting

Simple polynomial regression

$$\hat{y}(x;\theta) = \theta_4 x^3 + \theta_3 x^2 + \theta_2 x + \theta_1$$

To fit, use least-squares:

$$\theta^* = \min_\theta \sum_i^N (\hat{y}(x_i;\theta) - y(x_i))^2 \qquad (1)$$




Re-write using linear algebra:

$$\begin{pmatrix} \hat{y}_1 \\ \hat{y}_2 \\ \dots \end{pmatrix} = \begin{pmatrix} 1 & x_1 & x_1^2 & x_1^3 \\ 1 & x_2 & x_2^2 & x_2^3 \\ \dots & \dots & \dots & \dots \end{pmatrix} \begin{pmatrix} \theta_1 \\ \theta_2 \\ \theta_3 \\ \theta_4 \end{pmatrix} \text{ or } \hat{Y} = \Phi^T \theta$$

$$\theta^* = \min_\theta \|\Phi^T \theta - Y\|^2$$

In this case, it can be shown (1) has an analytical solution:

$$\theta^* = (\Phi^T \Phi)^{-1} \Phi^T Y$$

# Function fitting

## Simple polynomial regression

$$\hat{y}(x;\theta) = \theta_4 x^3 + \theta_3 x^2 + \theta_2 x + \theta_1$$

To fit, use least-squares:

$$\theta^* = \min_\theta \sum_i^N (\hat{y}(x_i;\theta) - y(x_i))^2 \qquad (1)$$

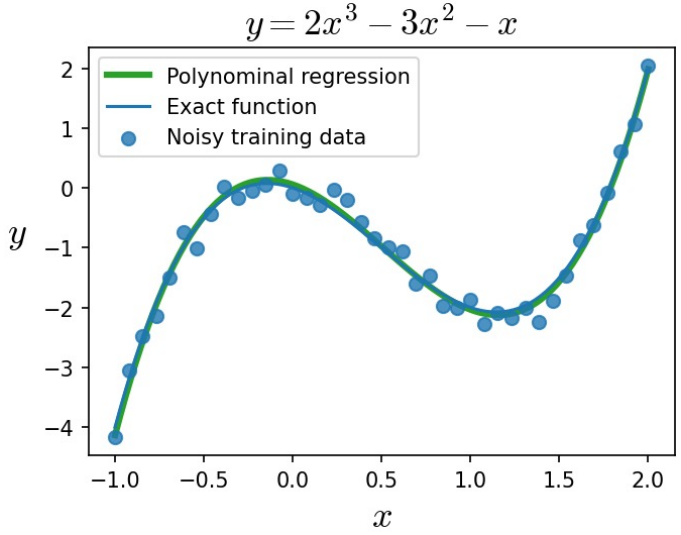
Re-write using linear algebra:

$$\begin{pmatrix} \hat{y}_1 \\ \hat{y}_2 \\ \dots \end{pmatrix} = \begin{pmatrix} 1 & x_1 & x_1^2 & x_1^3 \\ 1 & x_2 & x_2^2 & x_2^3 \\ \dots & \dots & \dots & \dots \end{pmatrix} \begin{pmatrix} \theta_1 \\ \theta_2 \\ \theta_3 \\ \theta_4 \end{pmatrix} \text{ or } \hat{Y} = \Phi^T \theta$$

$$\theta^* = \min_\theta \|\Phi^T \theta - Y\|^2$$

In this case, it can be shown (1) has an analytical solution:

$$\theta^* = (\Phi^T \Phi)^{-1} \Phi^T Y$$




## Neural network regression

$$\hat{y}(x;\theta) = NN(x;\theta)$$

To fit, use least-squares:

$$\theta^* = \min_\theta \sum_i^N (NN(x_i;\theta) - y(x_i))^2 \qquad (2)$$

In general, no analytical solution to (2) exists, so we must use **optimization**

For example, gradient descent:

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

or equally

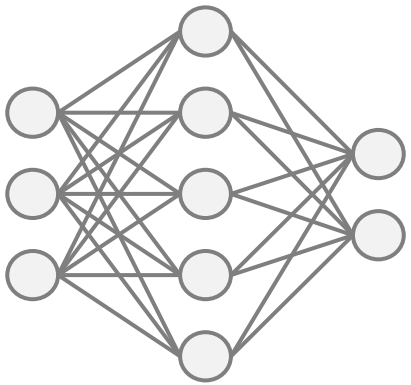
$$\theta_j \leftarrow \theta_j - \gamma \frac{\partial L(\theta)}{\partial \theta_j}$$

Where $\gamma$ is the learning rate and $L(\theta)$ is the **loss function**

# Neural network architecture

So, what exactly is $\hat{y} = NN(x; \theta)$?

This depends on the network **architecture** you choose (CNN, ResNet, Transformer, … etc)



2-layer MLP

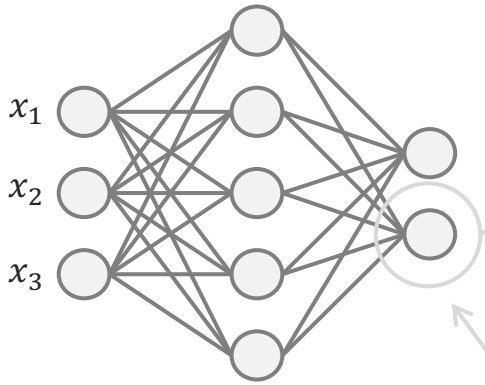
The most basic architecture is the **multilayer perceptron** (MLP) (aka **fully connected network**)

For example, a 2-layer MLP is defined as:

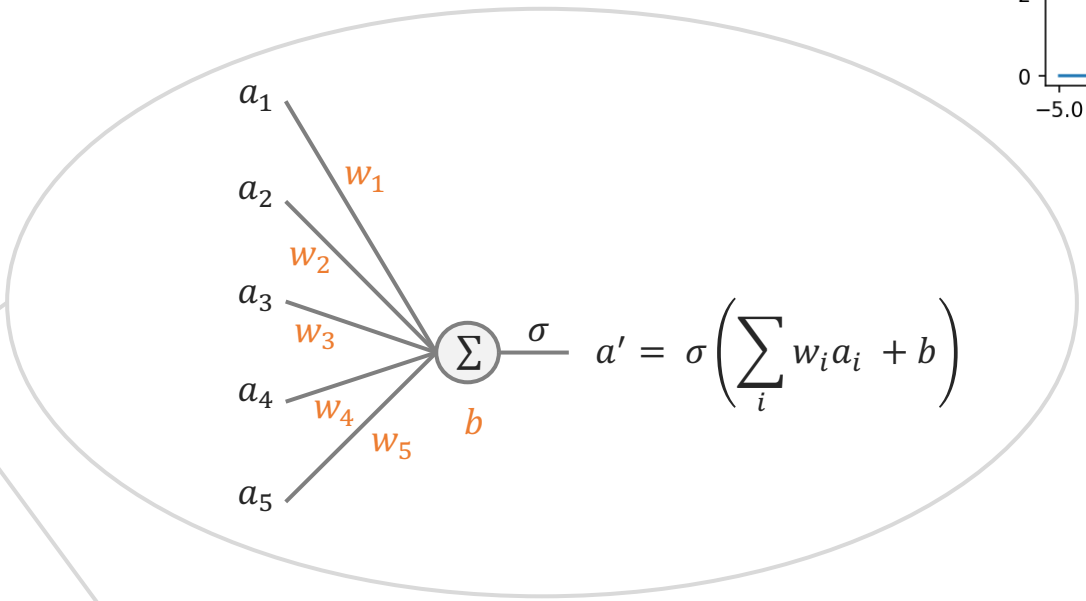
$$NN(x; \theta) = W_2 \sigma(W_1 x + b_1) + b_2$$

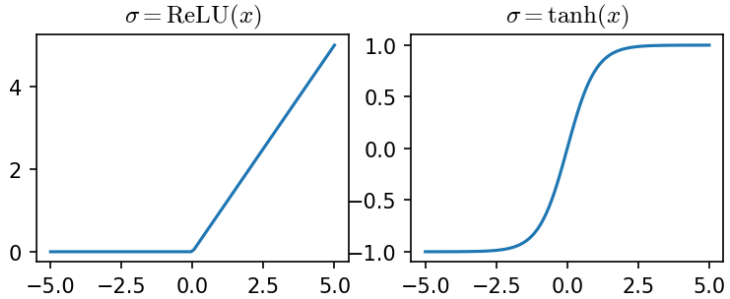
Where $x$ is an input vector, $W_1$ and $W_2$ are learnable weight matrices, $b_1$ and $b_2$ are learnable bias vectors, and $\sigma$ is an activation function, for example, $\sigma = \tanh(\cdot)$

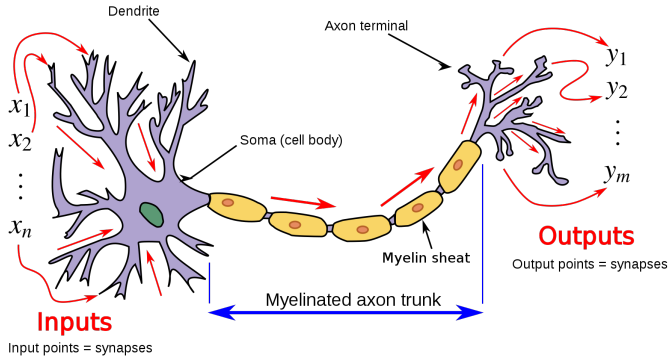
# Biological inspiration




2-layer MLP




Biological neuron
(Source: Wikipedia)

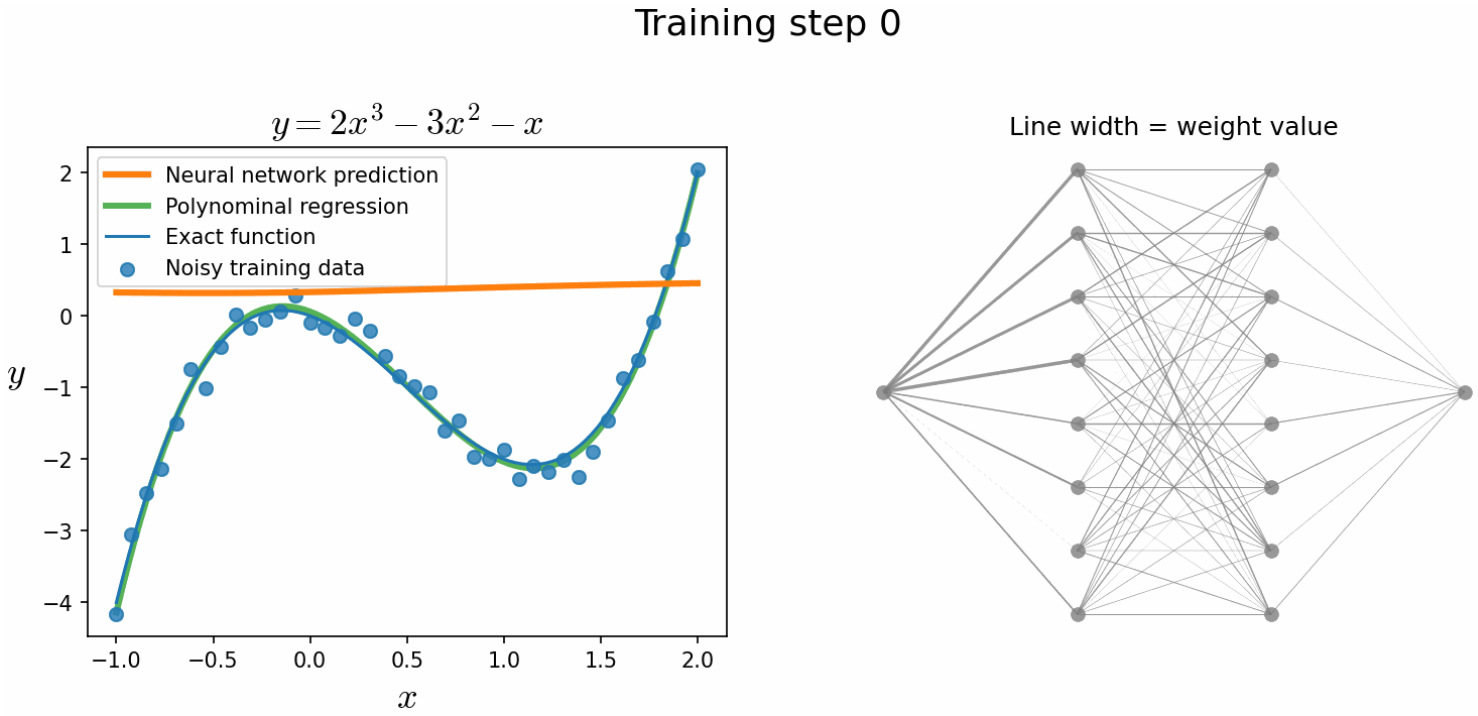
For last layer:

$$\begin{pmatrix} a_1' \\ a_2' \end{pmatrix} = \sigma\left( \begin{pmatrix} w_{11} & w_{12} & w_{13} & w_{14} & w_{15} \\ w_{21} & w_{22} & w_{23} & w_{24} & w_{25} \end{pmatrix} \begin{pmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} \right)$$

Entire network:

$$NN(\boldsymbol{x};\theta) = \sigma(W_2\sigma(W_1\boldsymbol{x} + \boldsymbol{b_1}) + \boldsymbol{b_2}) = \boldsymbol{f} \circ \boldsymbol{g}\ (\boldsymbol{x};\theta)$$

# Polynomial regression example




$$NN(x;\theta) = W_3(\sigma(W_2\sigma(W_1x + b_1) + b_2) + b_3$$

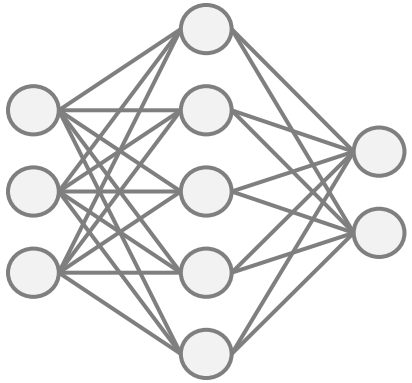
Trained using gradient descent

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

# Universal approximation

So why not just use linear regression?

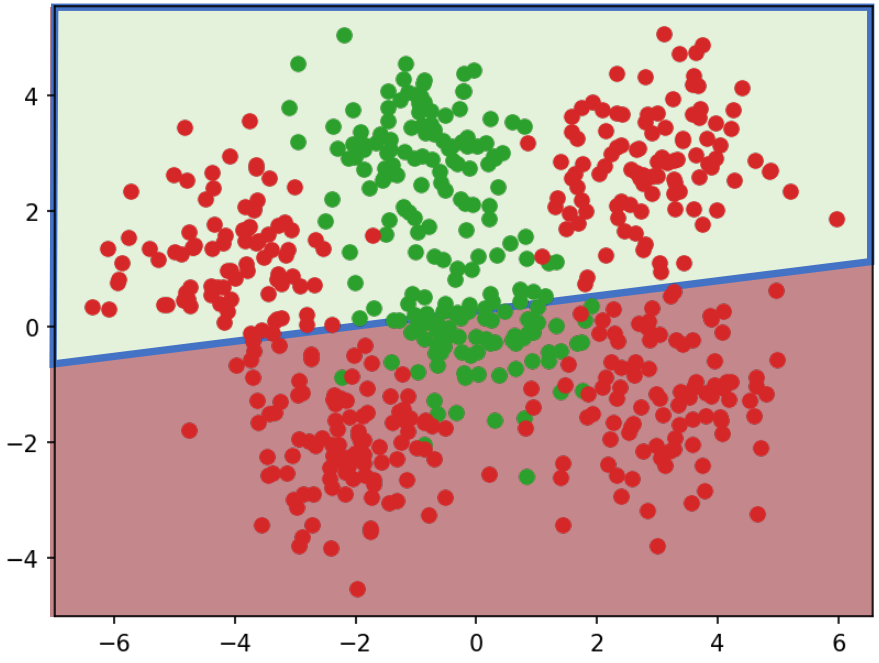
- Neural networks are simply **flexible functions** fit to data
- With enough parameters, neural networks can approximate any* arbitrarily complex function
  = **universal approximation**

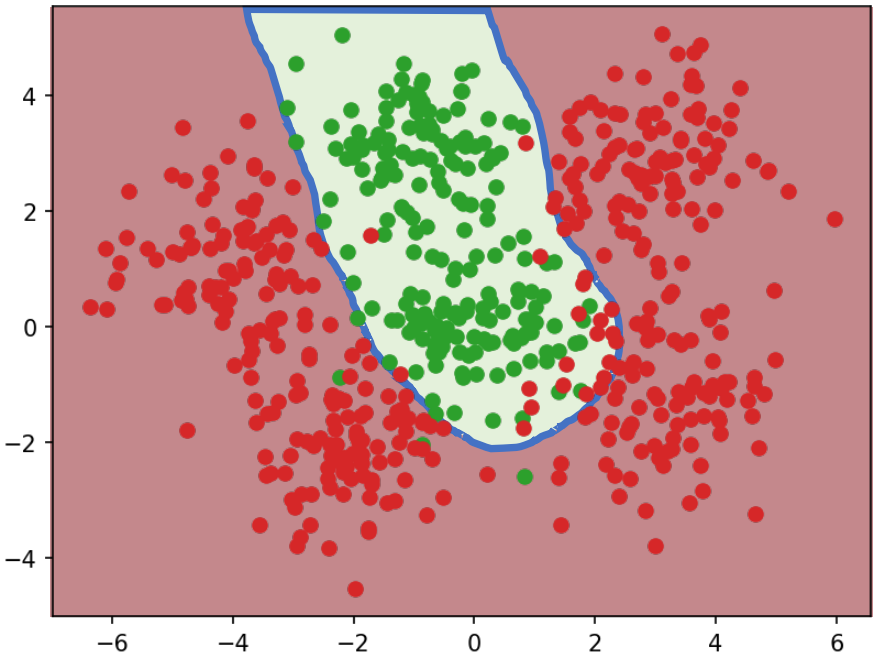


$\hat{y} = P(\text{dog} \mid x) = 1$

$x$ = array of RGB values

# Importance of activation functions

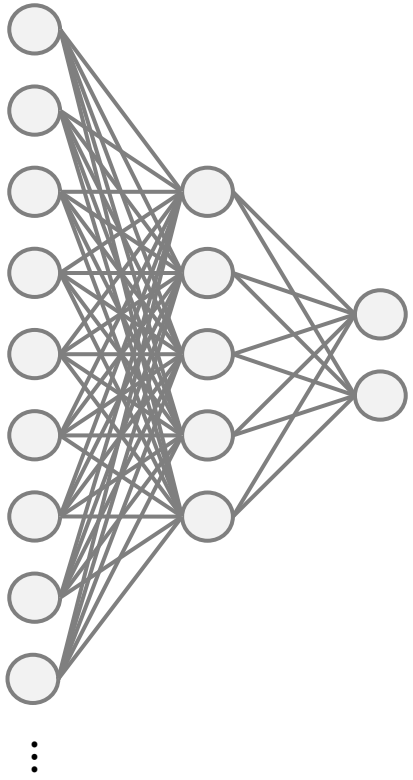



Non-linearities allow us to approximate arbitrary **non-linear** functions

# MLPs use lots of parameters

=> Flatten to 1D =>



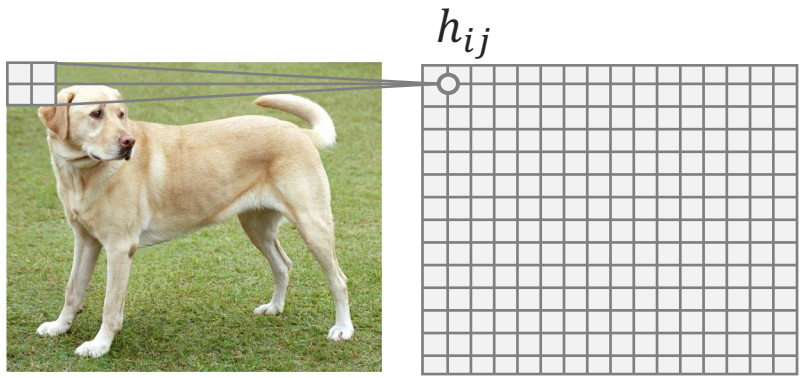
$$NN(x;\theta) = W_3(\sigma(W_2\sigma(W_1x + b_1) + b_2) + b_3$$

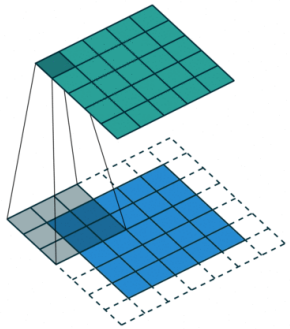
Assume the image has shape 128 x 128, and we have 100 hidden units in the first layer, then $W_1$ has shape (100 x (128 x 128)) = (100 x 16,384)

= 1.6M parameters!

=> A simple MLP image classifier can have millions of parameters

# Convolutional neural network (CNN)



$$NN(x;\theta) = W_3 \star (\sigma(W_2 \star \sigma(\underbrace{W_1 \star x + b_1}_{h}) + b_2) + b_3$$

$$h_{ij} = \sum_{i'}^{l} \sum_{j'}^{m} W_{i'j'} x_{i+i',j+j'} + b$$

Convolutional neural networks honor the **spatial correlations** in their inputs

Each neuron;
- Has a **limited** field of view
- **Shares** the same weights as the other neurons in the layer
- Mathematically, CNNs use cross-correlation

CNNs have translation equivariance (an inductive bias)

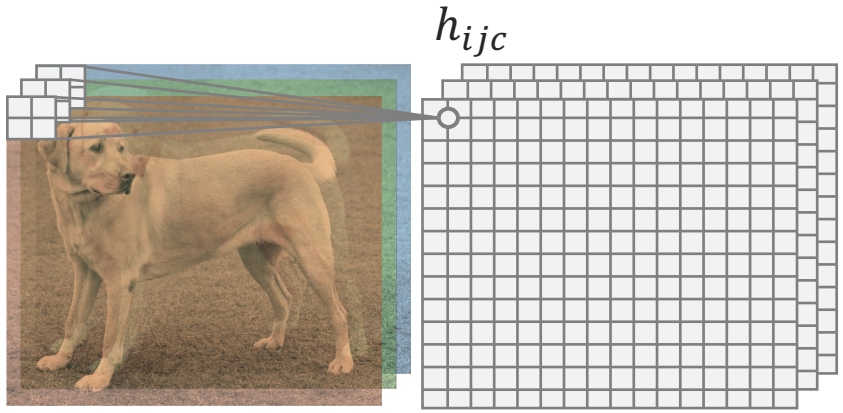
Let the size of the convolutional filter be 3 x 3

Then $W_1$ has shape (3 x 3)

= 9 parameters! (much, much smaller than a MLP)

Image source:
github/vdumoulin/conv_arithmetic

# Convolutional neural network (CNN)




Then the convolutional layer is defined by:

$$h_{ijc} = \sum_{i'}^{l} \sum_{j'}^{m} \sum_{c'}^{C_{\text{in}}} W_{i'j'c'c} x_{i+i',j+j',c'} + b_c$$

In practice, CNNs are usually extended so they can have multiple **channels** in the inputs and outputs of each layer

e.g. (R,G,B) image as input, where each channel is a color

Also:
- 1D and 3D CNNs follow analogously
- And we can add dilations and strides too

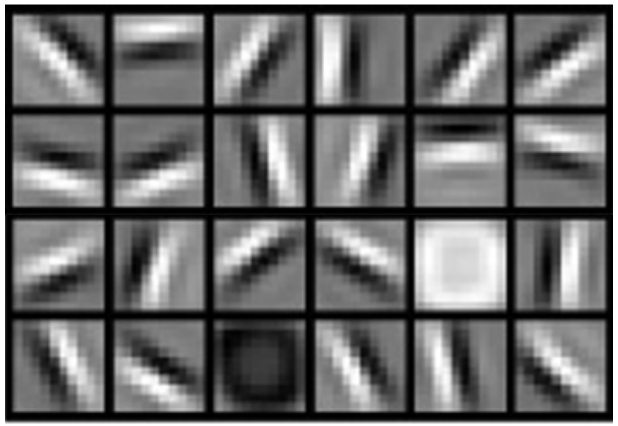
Let the size of the convolutional filter be 3 x 3

Then $W$ has shape (3 x 3 x $C_{\text{in}}$ x $C_{\text{out}}$)

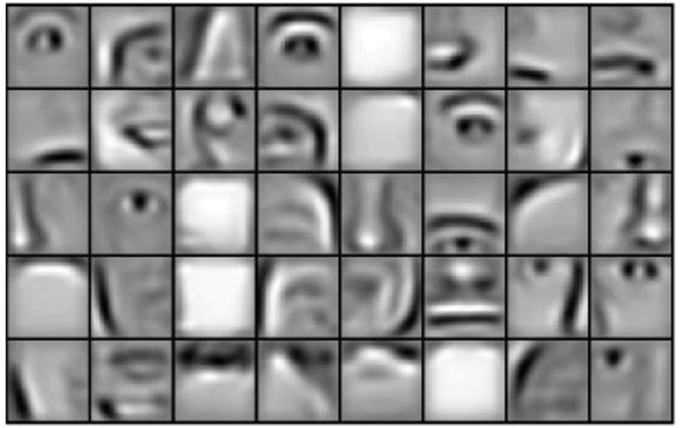
= 81 parameters for 3 input and 3 output channels

# Deep CNNs

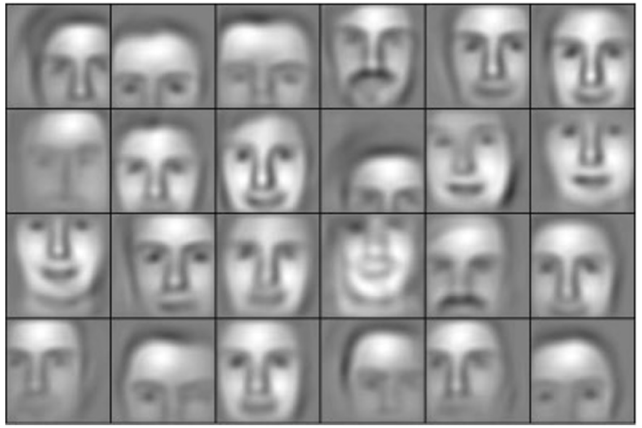
First layer



Second layer



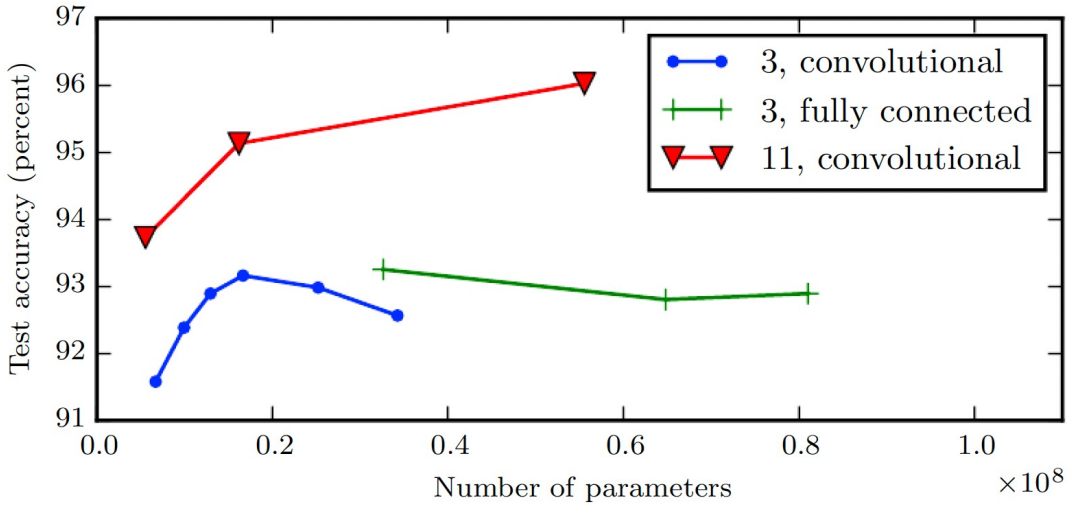
Third layer



Deep CNNs learn **hierarchical** features

Lee et al, Unsupervised Learning of Hierarchical Representations with Convolutional Deep Belief Networks, Communications of the ACM (2011)

# Depth is key




Goodfellow et al, Multi-digit number recognition from street view imagery using deep convolutional neural networks, ICLR (2014)

Empirically, deep neural networks perform better than shallow neural networks

=> encode a very general belief that the true function is **composed** of simpler functions

# Popular deep learning tasks

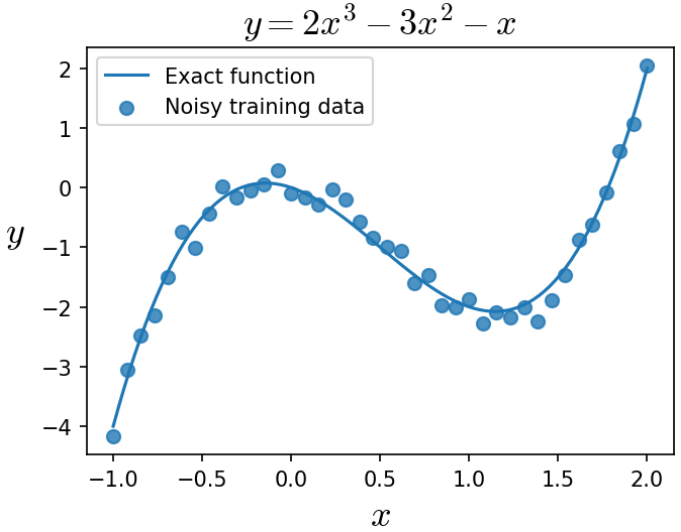
# Popular deep learning tasks

- Supervised learning
  - Regression
  - Classification

- Unsupervised learning
  - Feature learning
  - Autoregression
  - Generative models

- …but in all cases, the neural network is still a function fit to data! 

# Supervised learning - regression




$$\hat{y} = NN(x; \theta)$$

**Supervised learning - regression**:

Given a set of example inputs and outputs (labels) $\{(x_1, y_1), \dots, (x_N, y_N)\}$ from some true function $y(x)$ where $x \in \mathbb{R}^n, y \in \mathbb{R}^m$

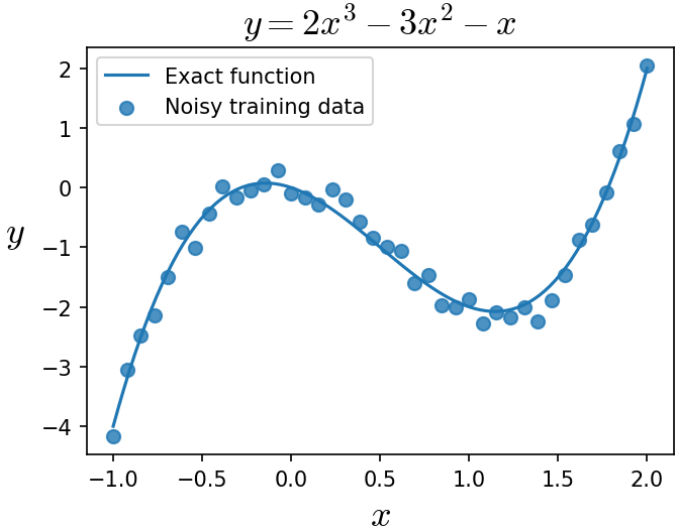
Find

$$\hat{y} = NN(x; \theta) \approx y(x)$$

Loss function (mean squared error)

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

# Supervised learning - regression




$$\hat{y} = NN(x; \theta)$$

Loss function (mean squared error)

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

**Supervised learning - regression**:

Given a set of example inputs and outputs (labels) $\{(x_1, y_1), \dots, (x_N, y_N)\}$ from some true function $y(x)$ where $x \in \mathbb{R}^n, y \in \mathbb{R}^m$

Find

$$\hat{y} = NN(x; \theta) \approx y(x)$$

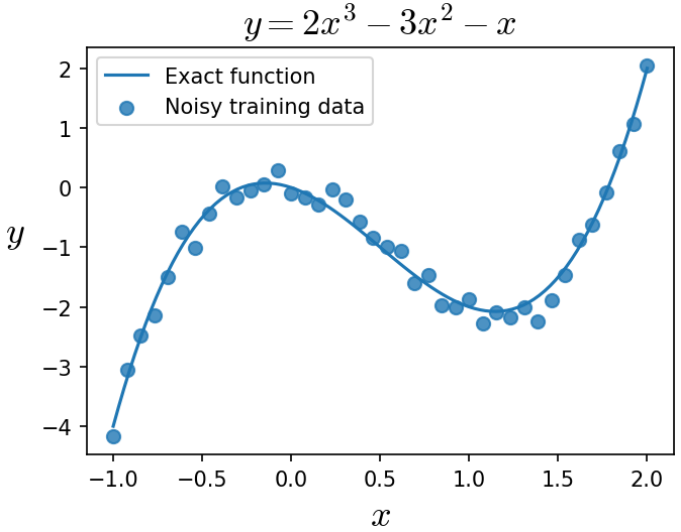
**Probabilistic perspective:**

Given a set of example inputs and outputs (labels) $\{(x_1, y_1), \dots, (x_N, y_N)\}$ drawn from the probability distribution $p(y|x)$

Find

$$\hat{p}(y|x, \theta) \approx p(y|x)$$

# Supervised learning - regression




$$\hat{y} = NN(x;\theta)$$

Loss function (mean squared error)

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

**Probabilistic perspective:**

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

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

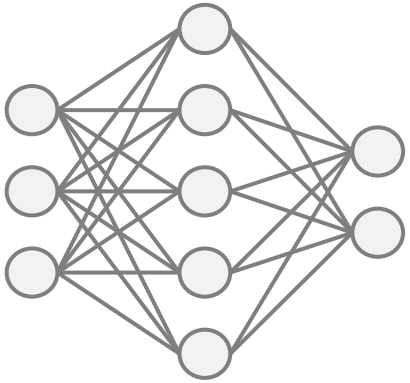
Then, assume each training datapoint is independently and identically distributed (iid), then the **data likelihood** can be written as:

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

Then use **maximum likelihood estimation** (MLE) to estimate $\theta^*$:

$$\theta^* = \max_{\theta}\hat{p}(D|\theta)$$
$$= \max_{\theta}\prod_{i}^{N} e^{-\frac{1}{2}\left(\frac{y(x_i)-NN(x_i;\theta)}{1}\right)^2}$$
$$= \min_{\theta}\sum_{i}^{N}\left(NN(x_i;\theta) - y(x_i)\right)^2$$

# Supervised learning - classification



$P(\text{dog} \mid x) = 0.99$

$P(\text{cat} \mid x) = 0.01$

**Supervised learning - classification**:

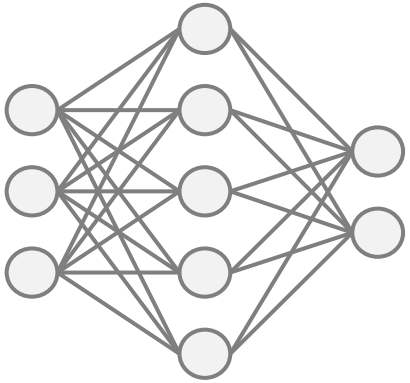
Given a set of example inputs and outputs (labels) $\{(x_1, y_1), \ldots, (x_N, y_N)\}$ drawn from the discrete probability distribution $P(y|x)$

where $y \in Y$, for example, $Y = \{\text{dog}, \text{cat}\}$

Find

$$\hat{P}(y|x, \theta) \approx P(y|x)$$

# Supervised learning - classification



$P(\text{dog} \mid x) = 0.99$

$P(\text{cat} \mid x) = 0.01$

Let each class be encoded as a one-hot vector of length $C$, e.g

$$y = (0,0,1,0)$$

Then assume

$$\hat{P}(y|x,\theta) = \prod_j^C NN(x;\theta)_j^{y_j}, \qquad \sum_j^C NN(x;\theta)_j = 1$$

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

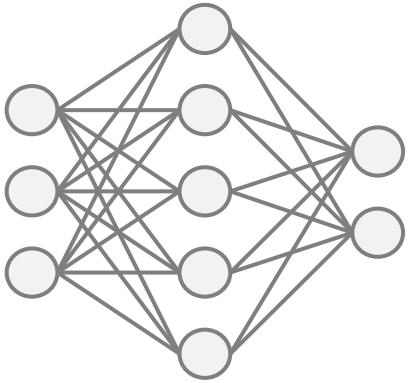
$$\hat{P}(D|\theta) = \hat{P}(x_1, y_1, \dots, x_n, y_n|\theta) = \prod_i^N \hat{P}(y_i|x_i,\theta)$$

Then use **maximum likelihood estimation** (MLE) to estimate $\theta^*$:

$$\begin{aligned}\theta^* &= \max_\theta \hat{P}(D|\theta) \\ &= \max_\theta \prod_i^N \prod_j^C NN(x_i;\theta)_j^{y_{ij}} \\ &= \min_\theta -\sum_i^N \sum_j^C y_{ij} \log NN(x_i;\theta)_j\end{aligned}$$

Also known as the **cross-entropy loss**

# Supervised learning - classification



$P(\text{dog} \mid x) = 0.99$

$P(\text{cat} \mid x) = 0.01$

Let each class be encoded as a one-hot vector of length $C$, e.g

$$y = (0,0,1,0)$$

Typically, we use a softmax output layer to assert $\sum_j^C NN(x;\theta)_j = 1$;

$$\sigma(\boldsymbol{z})_i = \frac{e^{z_i}}{\sum_j^C e^{z_j}}$$

Then assume

$$\hat{P}(y|x,\theta) = \prod_j^C NN(x;\theta)_j^{y_j}, \qquad \sum_j^C NN(x;\theta)_j = 1$$

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

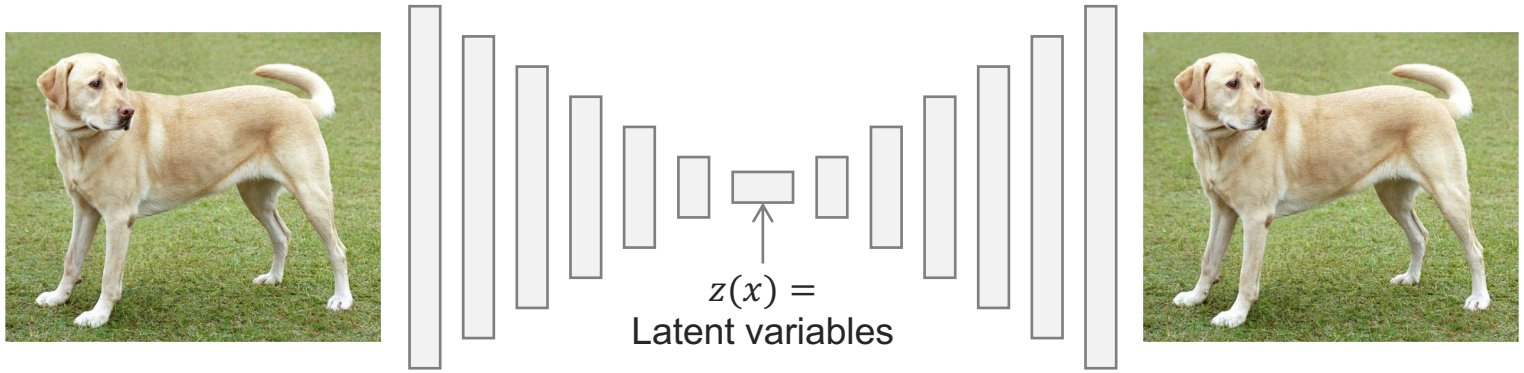
$$\hat{P}(D|\theta) = \hat{P}(x_1, y_1, \dots, x_n, y_n|\theta) = \prod_i^N \hat{P}(y_i|x_i,\theta)$$

Then use **maximum likelihood estimation** (MLE) to estimate $\theta^*$:

$$\begin{aligned}\theta^* &= \max_\theta \hat{P}(D|\theta) \\ &= \max_\theta \prod_i^N \prod_j^C NN(x_i;\theta)_j^{y_{ij}} \\ &= \min_\theta -\sum_i^N \sum_j^C y_{ij} \log NN(x_i;\theta)_j\end{aligned}$$

Also known as the **cross-entropy loss**

# Unsupervised learning - feature learning




Loss function

Many different possibilities, a simple choice is

$$L(\theta) = \sum_i^N (NN(x_i;\theta) - x_i))^2$$

**Unsupervised learning – feature learning**

Given a set of examples $\{x_1, \dots, x_N\}$, find some features $z(x)$

Which are salient descriptors of $x$, where $x \in \mathbb{R}^n, z \in \mathbb{R}^d$

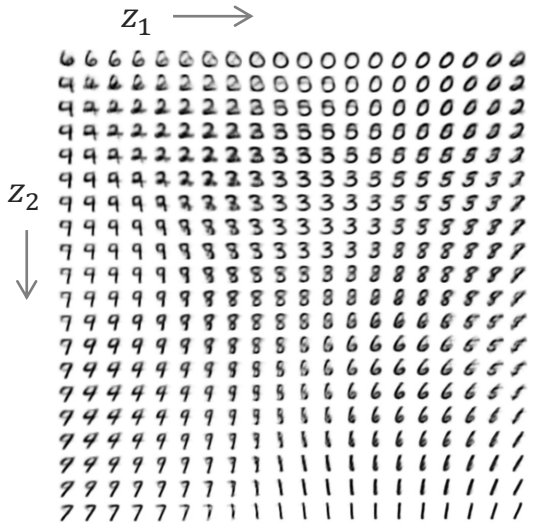
Typically, $d \ll n$ (= compression)

$z$ can be used for downstream tasks, e.g. clustering / classification

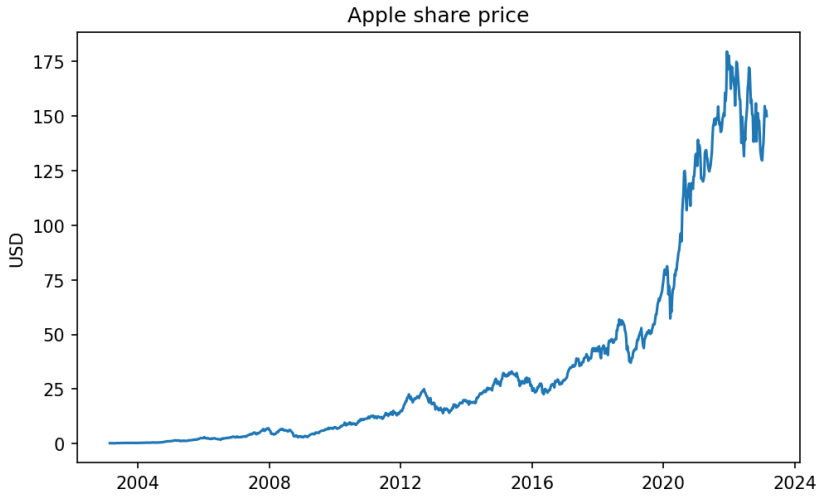
For example:

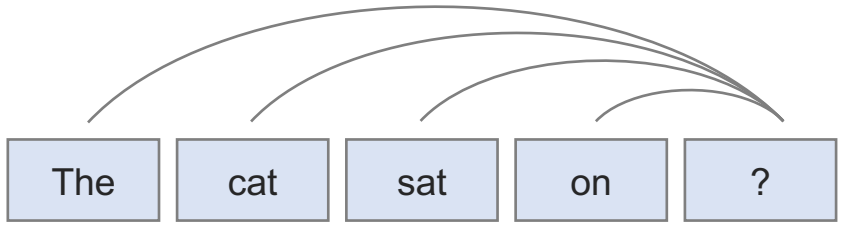
Variational autoencoders (VAEs)

Kingma et al, 2014



# Unsupervised learning - autoregression

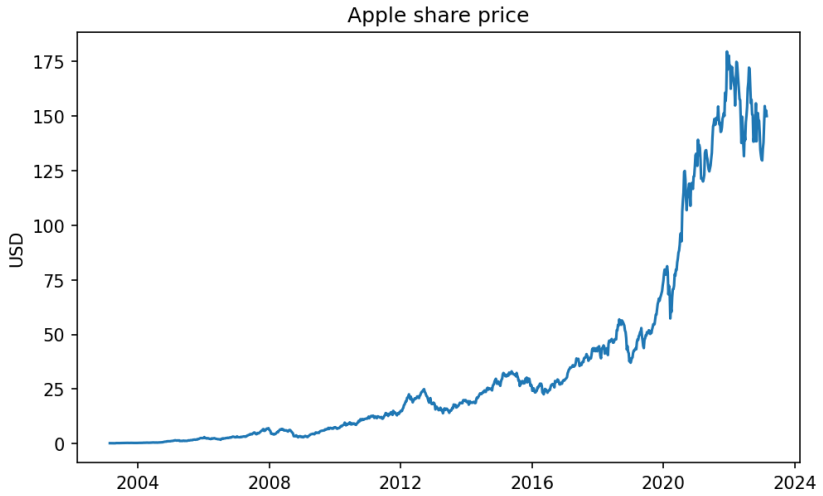


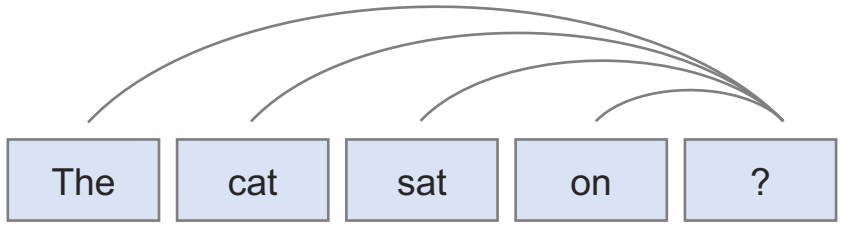




**Unsupervised learning – autoregression**

Given many examples sequences, train a model to predict future values from past values

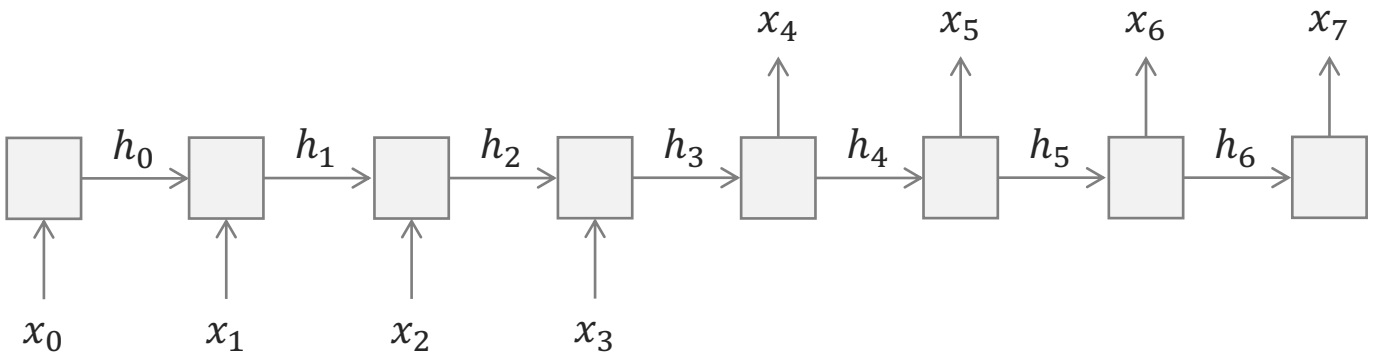
# Unsupervised learning - autoregression

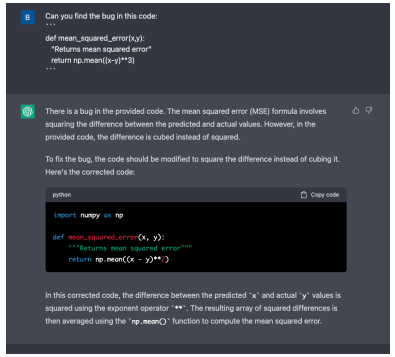



For example:

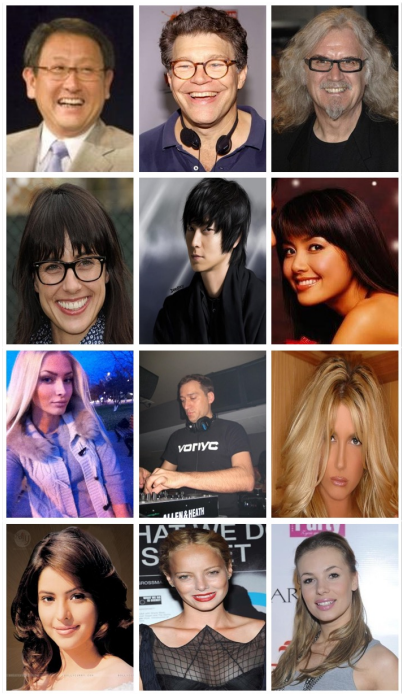
ChatGPT



# Unsupervised learning - generative modelling

Training dataset



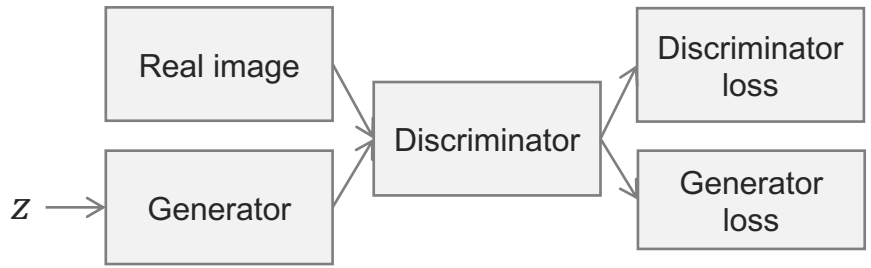
Source: CelebA

Generative model


For example:

Generative adversarial networks (GANs)

Goodfellow et al, 2014




**Unsupervised learning – generative modelling**

Given many examples $\{x_1, \ldots, x_N\}$ sampled from some distribution $p(x)$, learn to sample from $p(x)$

# 5 min break

# Lecture overview

- What is deep learning?
  - Multilayer perceptrons
  - Universal approximation
- Popular deep learning tasks
  - Supervised learning
  - Unsupervised learning
- Training a deep neural network
  - Gradient descent
  - Backpropagation
  - Autodifferentiation
- Live coding – implementing a DNN from scratch

# How do we train neural networks?

## Gradient descent

$$\hat{y}(x;\theta) = NN(x;\theta)$$

To fit, use least-squares:

$$\theta^* = \min_\theta \sum_i^N (NN(x_i;\theta) - y(x_i))^2 \qquad (2)$$

In general, no analytical solution to (2) exists, so we must use **optimization**

For example, gradient descent:

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

or equally

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

Where $\gamma$ is the learning rate and $L(\theta)$ is the **loss function**

# How do we train neural networks?

## Gradient descent

$$\hat{y}(x;\theta) = NN(x;\theta)$$

To fit, use least-squares:

$$\theta^* = \min_\theta \sum_i^N (NN(x_i;\theta) - y(x_i))^2 \qquad (2)$$

In general, no analytical solution to (2) exists, so we must use **optimization**

For example, gradient descent:

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

or equally

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

Where $\gamma$ is the learning rate and $L(\theta)$ is the **loss function**

Note that

$$\frac{\partial L(\theta)}{\partial \theta_j} = \sum_i^N 2(NN(x_i;\theta) - y(x_i)) \frac{\partial NN(x_i;\theta)}{\partial \theta_j}$$

Let's consider a fully connected network

$$NN(\boldsymbol{x};\theta) = W_3(\sigma(\underbrace{W_2\sigma(\underbrace{W_1\boldsymbol{x} + \boldsymbol{b_1}}_{\boldsymbol{h}}) + \boldsymbol{b_2}}_{\boldsymbol{g}}) + \boldsymbol{b_3} = \boldsymbol{f} \circ \boldsymbol{g} \circ \boldsymbol{h}(\boldsymbol{x};\theta)$$

How do we calculate $\frac{\partial NN(\boldsymbol{x_i};\theta)}{\partial W_1}$ ?

Note $\boldsymbol{f}, \boldsymbol{g}$, and $\boldsymbol{h}$ are vector functions =>

Use the **multivariate chain rule** (= matrix multiplication of **Jacobians**)

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

$$J = \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{g}} = \begin{pmatrix} \frac{\partial f_1}{\partial g_1} & \cdots & \frac{\partial f_1}{\partial g_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial g_1} & \cdots & \frac{\partial f_m}{\partial g_n} \end{pmatrix}$$

# Evaluating the chain rule

$$NN(x;\theta) = W_3(\sigma(W_2\sigma(\underbrace{W_1\boldsymbol{x} + \boldsymbol{b_1}}_{\boldsymbol{h}}) + \boldsymbol{b_2}) + \boldsymbol{b_3} = \boldsymbol{f} \circ \overbrace{\boldsymbol{g} \circ \boldsymbol{h}}(\boldsymbol{x};\theta)$$

Then

$$\boldsymbol{f} = W_3\sigma(\boldsymbol{g}) + \boldsymbol{b_3} \Rightarrow \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{g}} = W_3 \operatorname{diag}(\sigma'(\boldsymbol{g}))$$

$$\boldsymbol{g} = W_2\sigma(\boldsymbol{h}) + \boldsymbol{b_2}) \Rightarrow \frac{\partial \boldsymbol{g}}{\partial \boldsymbol{h}} = W_2 \operatorname{diag}(\sigma'(\boldsymbol{h}))$$

$$\boldsymbol{h} = W_1\boldsymbol{x} + \boldsymbol{b_1} \Rightarrow \frac{\partial \boldsymbol{h}}{\partial W_1} = \frac{\partial(W_1\boldsymbol{x} + \boldsymbol{b_1})}{\partial W_1}$$

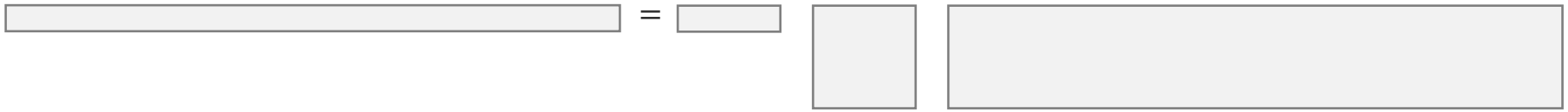
$$\frac{\partial NN}{\partial W_1} = \frac{\partial f}{\partial \boldsymbol{g}}\frac{\partial \boldsymbol{g}}{\partial \boldsymbol{h}}\frac{\partial \boldsymbol{h}}{\partial W_1} = W_3 \operatorname{diag}(\sigma'(\boldsymbol{g}))W_2 \operatorname{diag}(\sigma'(\boldsymbol{h}))\frac{\partial(W_1\boldsymbol{x} + \boldsymbol{b_1})}{\partial W_1}$$

# Dimensionality

$$NN(x;\theta) = W_3(\sigma(W_2\sigma(\underbrace{W_1\boldsymbol{x} + \boldsymbol{b_1}}_{\boldsymbol{h}}) + \boldsymbol{b_2}) + \boldsymbol{b_3} = \underbrace{\boldsymbol{f} \circ \boldsymbol{g}}_{\boldsymbol{g}} \circ \boldsymbol{h}(\boldsymbol{x};\theta)$$

Let's think about **dimensionality**. Let $\boldsymbol{f}$ have 1 element (scalar output), $\boldsymbol{g}$ and $\boldsymbol{h}$ both have 100 elements (100 hidden units) and $\boldsymbol{x}$ be a flattened 128 x 128 image, then $W_1$ has shape (100 x (128 x 128)) = 1.6M elements and

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

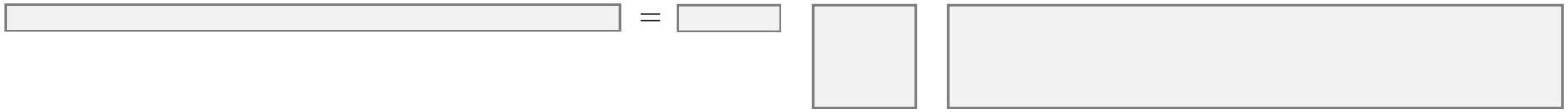


(1 x 1.6M) = (1 x 100) (100 x 100) (100 x 1.6M)

⚠ The last Jacobian is extremely large! = 160M elements (or 0.7 GB!)

# Forward- versus reverse-mode differentiation

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



(1 x 1.6M) = (1 x 100) (100 x 100) (100 x 1.6M)

**Consider evaluating the chain rule (RHS):**

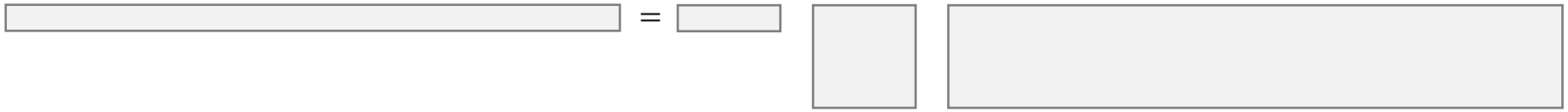
1) From right to left **(forward-mode)**

(100 x 100) (100 x 1.6M) → (100 x 1.6M)
(1 x 100) (100 x 1.6M) → (1 x 1.6M)

= lots of computation (large matrix-matrix multiply)

# Forward- versus reverse-mode differentiation

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



(1 x 1.6M) = (1 x 100) (100 x 100) (100 x 1.6M)

**Consider evaluating the chain rule (RHS):**

1) From right to left **(forward-mode)**

(100 x 100) (100 x 1.6M) → (100 x 1.6M)
(1 x 100) (100 x 1.6M) → (1 x 1.6M)

= lots of computation (large matrix-matrix multiply)

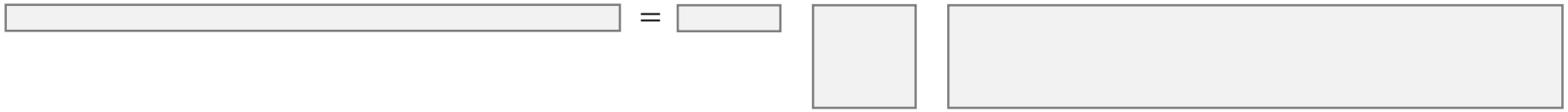
2) From left to right **(reverse-mode)**

(1 x 100) (100 x 100) → (1 x 100)
(1 x 100) (100 x 1.6M) → (1 x 1.6M)

= much less computation (vector-matrix multiplies)

# Forward- versus reverse-mode differentiation

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



(1 x 1.6M) = (1 x 100) (100 x 100) (100 x 1.6M)

**Consider evaluating the chain rule (RHS):**

1) From right to left **(forward-mode)**

(100 x 100) (100 x 1.6M) → (100 x 1.6M)
(1 x 100) (100 x 1.6M) → (1 x 1.6M)

= lots of computation (large matrix-matrix multiply)

2) From left to right **(reverse-mode)**

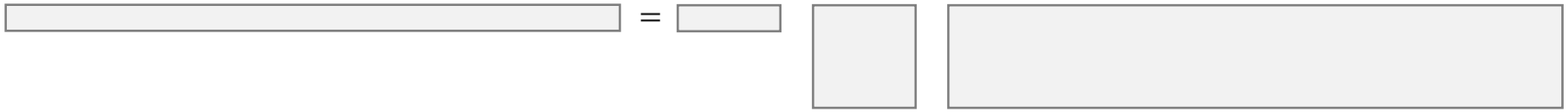
(1 x 100) (100 x 100) → (1 x 100)
(1 x 100) (100 x 1.6M) → (1 x 1.6M)

= much less computation (vector-matrix multiplies)

*=> Order matters! Typically, reverse mode differentiation is more efficient for* ***scalar*** *loss functions with large numbers of parameters ("wide" Jacobians), whilst forward mode is more efficient for evaluating "tall" Jacobians*

# Forward- versus reverse-mode differentiation

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



(1 x 1.6M) = (1 x 100) (100 x 100) (100 x 1.6M)

But note, the last matrix multiply ($\frac{\partial f}{\partial \boldsymbol{h}} \frac{\partial \boldsymbol{h}}{\partial W_1}$) is still
(1 x 100) (100 x 1.6M)!

2) From left to right **(reverse-mode)**

(1 x 100) (100 x 100) → (1 x 100)
(1 x 100) (100 x 1.6M) → (1 x 1.6M)

= much less computation (vector-matrix multiplies)

# Forward- versus reverse-mode differentiation

But note:

$$\frac{\partial \boldsymbol{h}}{\partial W_1} = \frac{\partial (W_1 \boldsymbol{x} + \boldsymbol{b_1})}{\partial W_1} = \begin{pmatrix} x_1 & x_2 & \cdots & x_n & 0 & 0 & \cdots & 0 \\ \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots \\ 0 & 0 & \cdots & 0 & x_1 & x_2 & \cdots & x_n \end{pmatrix}$$

Then

$$\frac{\partial NN}{\partial W_1} = \frac{df}{\partial \boldsymbol{h}} \frac{\partial \boldsymbol{h}}{\partial W_1} = \left( \frac{df}{\partial h_1}(x_1, \cdots x_N), \cdots, \frac{df}{\partial h_{100}}(x_1, \cdots, x_N) \right)$$
$$= \frac{\partial f}{\partial \boldsymbol{h}}^{\boldsymbol{T}} \otimes \boldsymbol{x}$$

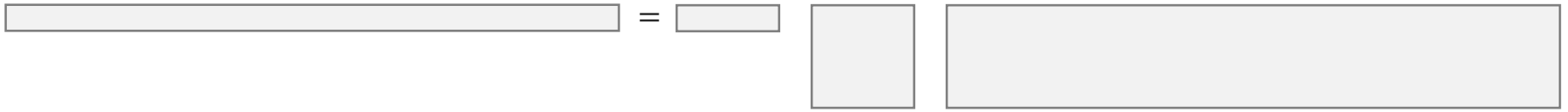
This is just the (flattened) **outer product** of two vectors (100 x 1) $\otimes$ (16,384 x 1)

$\Rightarrow$ We don't have to fully populate the last Jacobian ($\frac{\partial \boldsymbol{h}}{\partial W_1}$) when computing its vector-Jacobian product

$\Rightarrow$ Computing vector-Jacobian products is usually simpler and **more** efficient than directly computing Jacobians

# Backpropagation

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



(1 x 1.6M) = (1 x 100) (100 x 100) (100 x 1.6M)

2) From left to right **(reverse-mode)**

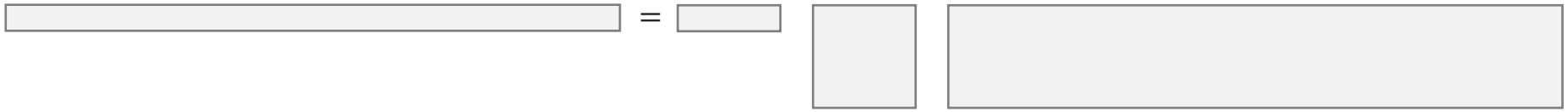
(1 x 100) (100 x 100) → (1 x 100)
~~(1 x 100) (100 x 1.6M) → (1 x 1.6M)~~
(100 x 1) ⊗ (16,384 x 1) → (1 x 1.6M)

= Efficient training code

Allows us to train neural networks with **billions** of parameters

# Backpropagation

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



(1 x 1.6M) = (1 x 100) (100 x 100) (100 x 1.6M)

Note:
Training neural networks = Lots of matrix – matrix multiplications

These types of parallelisable operations are perfect for GPUs

10x to 100x speed-ups compared to CPUs

2) From left to right **(reverse-mode)**

(1 x 100) (100 x 100) → (1 x 100)
~~(1 x 100) (100 x 1.6M) → (1 x 1.6M)~~
(100 x 1) ⊗ (16,384 x 1) → (1 x 1.6M)

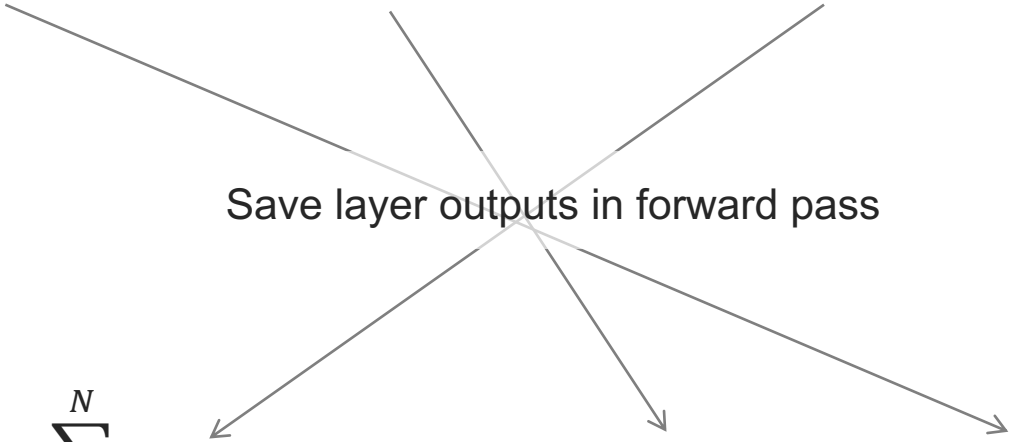
= Efficient training code

Allows us to train neural networks with **billions** of parameters

# Backpropagation

Forward pass:

$$\boldsymbol{x_i} \to \boldsymbol{h_i} = W_1\boldsymbol{x_i} + \boldsymbol{b_1} \to \boldsymbol{g_i} = W_2\sigma(\boldsymbol{h_i}) + \boldsymbol{b_2} \to f_i = W_3\sigma(\boldsymbol{g_i}) + \boldsymbol{b_3}$$



Save layer outputs in forward pass

Backward pass:

$$\frac{\partial L}{\partial W_1} = \sum_i^N 2(f_i - y(\boldsymbol{x_i}))\ W_3\ \mathrm{diag}(\sigma'(\boldsymbol{g_i}))\ W_2\ \mathrm{diag}(\sigma'(\boldsymbol{h_i})) \otimes \boldsymbol{x_i}$$

Evaluate from left to right (reverse-mode)

**Batch** matrix multiplies over training examples $\boldsymbol{x_i}$

Similar equations for other weight matrices and bias vectors

# Backpropagation

Forward pass:

$$\boldsymbol{x_i} \to \boldsymbol{h_i} = W_1\boldsymbol{x_i} + \boldsymbol{b_1} \to \boldsymbol{g_i} = W_2\sigma(\boldsymbol{h_i}) + \boldsymbol{b_2} \to f_i = W_3\sigma(\boldsymbol{g_i}) + \boldsymbol{b_3}$$

Backward pass:

In practice:

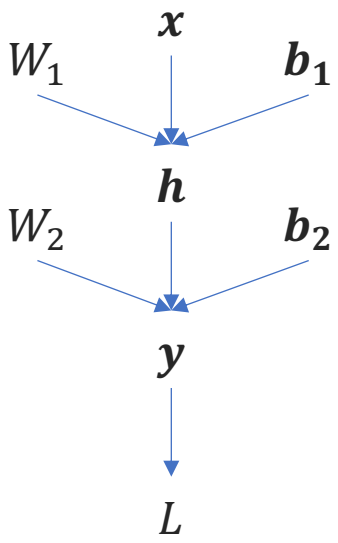
PyTorch

TensorFlow

**Autodifferentiation** tracks all your forward computations and their gradients and applies the chain rule automatically for you, so you don't have to worry!

# Autodifferentiation

$$\boldsymbol{y} = W_2\sigma(W_1\boldsymbol{x} + \boldsymbol{b_1}) + \boldsymbol{b_2}$$




Autodifferentiation = gradients of **arbitrary** programs

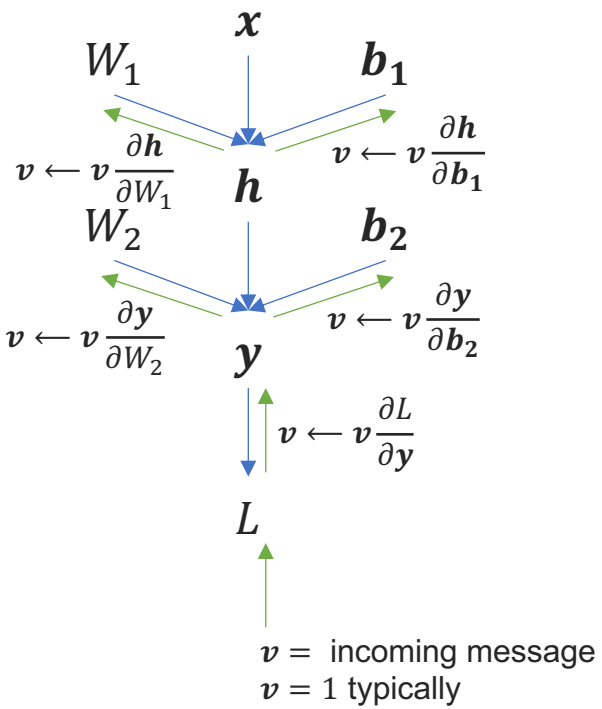
Fundamentally, autodifferentiation builds a directed **graph** of **primitive operations**

Each primitive operation defines:

1) **Forward operation**
2) **vector-Jacobian product** (for reverse mode autodiff)
3) **Jacobian-vector product** (for forward-mode autodiff)

# Autodifferentiation

$$\boldsymbol{y} = W_2\sigma(W_1\boldsymbol{x} + \boldsymbol{b_1}) + \boldsymbol{b_2}$$



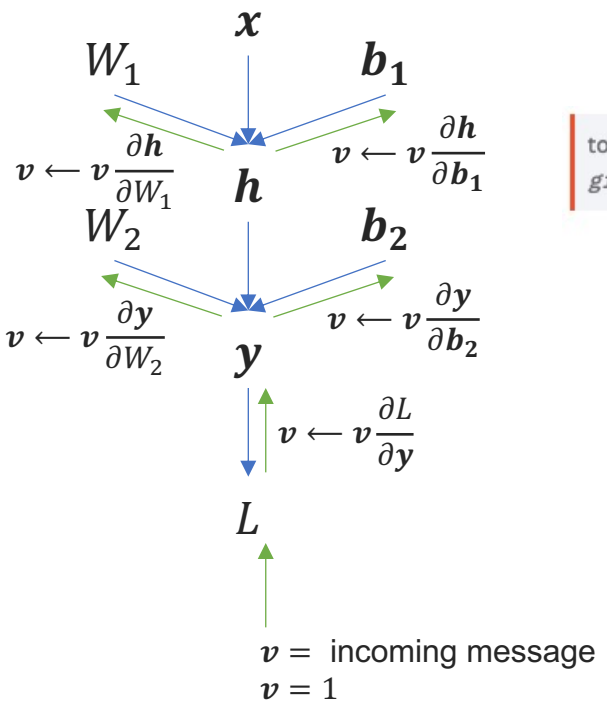

Then, autodifferentiation can be thought of as **message-passing** between the graph's nodes

Each message computes the vjp or jvp

Combined, evaluates the **chain rule**

# Autodifferentiation

$$\boldsymbol{y} = W_2\sigma(W_1\boldsymbol{x} + \boldsymbol{b_1}) + \boldsymbol{b_2}$$




loss.backward()

PyTorch

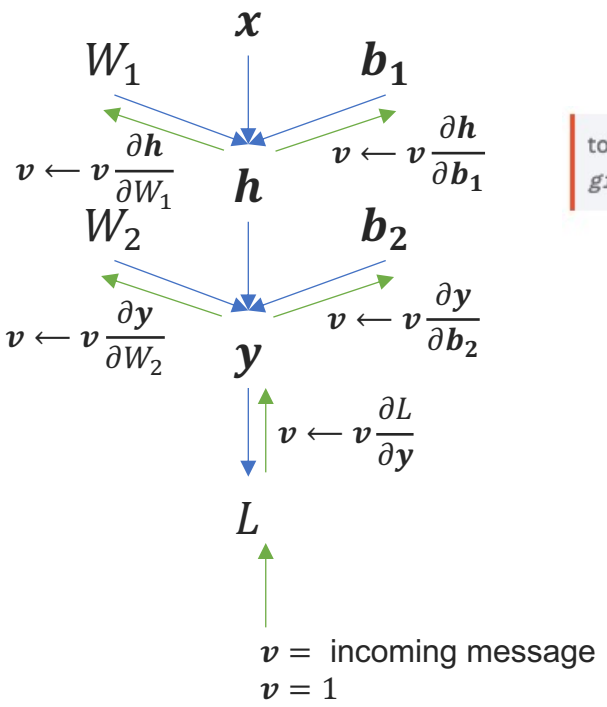
torch.autograd.backward(*tensors*, *grad_tensors=None*, *retain_graph=None*, *create_graph=False*, *grad_variables=None*, *inputs=None*) [SOURCE]

Computes the sum of gradients of given tensors with respect to graph leaves.

The graph is differentiated using the chain rule. If any of `tensors` are non-scalar (i.e. their data has more than one element) and require gradient, then the Jacobian-vector product would be computed, in this case the function additionally requires specifying `grad_tensors`. It should be a sequence of matching length, that contains the "vector" in the Jacobian-vector product, usually the gradient of the differentiated function w.r.t. corresponding tensors (`None` is an acceptable value for all tensors that don't need gradient tensors).

# Autodifferentiation

$$\boldsymbol{y} = W_2\sigma(W_1\boldsymbol{x} + \boldsymbol{b_1}) + \boldsymbol{b_2}$$




loss.backward()

PyTorch

torch.autograd.backward(*tensors*, *grad_tensors=None*, *retain_graph=None*, *create_graph=False*, *grad_variables=None*, *inputs=None*) [SOURCE]

Computes the sum of gradients of given tensors with respect to graph leaves.

The graph is differentiated using the chain rule. If any of `tensors` are non-scalar (i.e. their data has more than one element) and require gradient, then the Jacobian-vector product would be computed, in this case the function additionally requires specifying `grad_tensors`. It should be a sequence of matching length, that contains the "vector" in the Jacobian-vector product, usually the gradient of the differentiated function w.r.t. corresponding tensors (`None` is an acceptable value for all tensors that don't need gradient tensors).

Note autodiff is **not**
- Symbolic differentiation
- Finite differences

It is a way of efficiently computing exact gradients

# Live coding a simple MLP in Python