# Machine Learning with Neural Networks

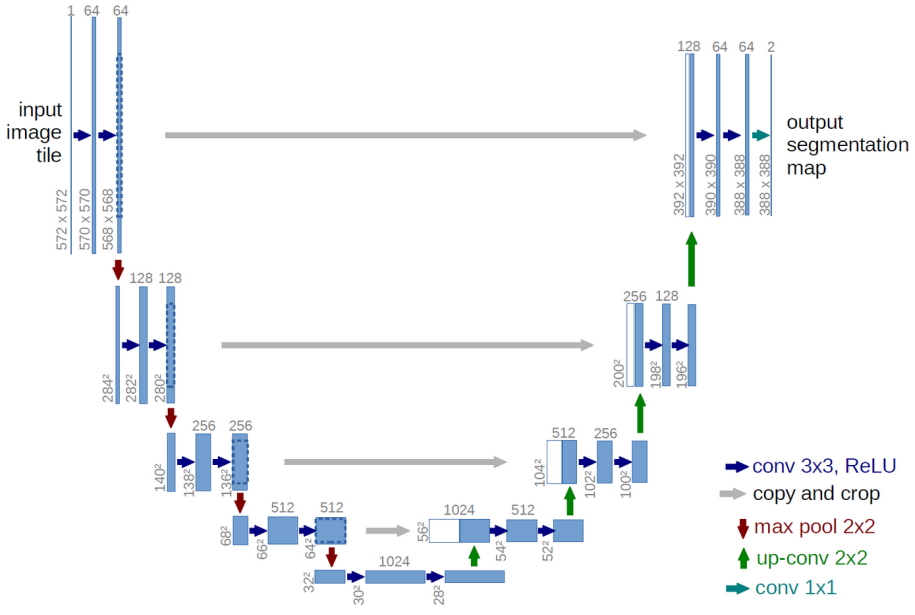
Stefan Siegert (s.siegert@exeter.ac.uk)

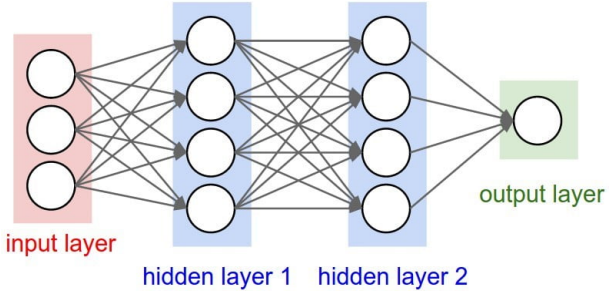
Machine Learning Workshop 30 June 2023

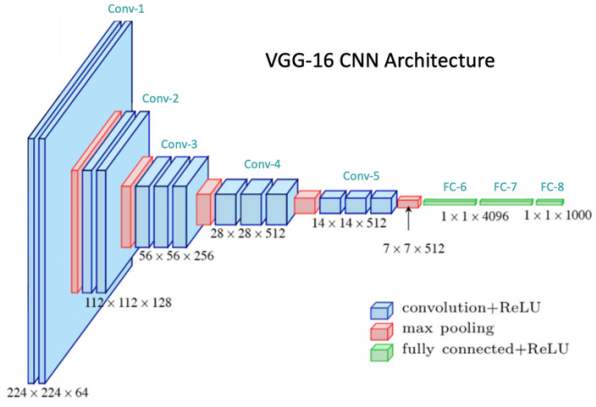
# Roadmap for this session

- Main goal: **Understand and implement Deep Neural Networks**
- Basics (some maths, terminology, principles)
- Software (Tensorflow, Keras)
- Convolutional neural network for classification (weather types)
- U-Net for spatio-temporal prediction (rain prediction)

# By the end of the workshop you will better understand these images:



# Learning from data

- Inputs $\boldsymbol{x}$ and outputs $\boldsymbol{y}$ (usually tensors)
- Training data $\{\boldsymbol{x}_i, \boldsymbol{y}_i\}_{i=1}^n$
- Model: Functional relationship $\hat{\boldsymbol{y}} = f(\boldsymbol{x}; \boldsymbol{w})$ where $f$ depends on trainable parameters $\boldsymbol{w}$ ("weights")
- Loss function $\ell(\hat{\boldsymbol{y}}, \boldsymbol{y})$ quantifies the mismatch between the function output and the training output
- Learning: Minimise the overall loss function $L(\boldsymbol{w}) = \sum_{i=1}^n \ell(\hat{\boldsymbol{y}}_i, \boldsymbol{y}_i)$ with respect to the trainable parameters $\boldsymbol{w}$:

$$\boldsymbol{w}^* = \underset{\boldsymbol{w}}{\mathit{argmin}} L(\boldsymbol{w})$$

- Much of machine learning research focusses on finding appropriate choices of $f(\boldsymbol{x}; \boldsymbol{w})$ and $\ell(\boldsymbol{y}, \hat{\boldsymbol{y}})$.

# Types of machine learning models

- Regression: Output $\boldsymbol{y}$ is continuous $\boldsymbol{y} \in \mathbb{R}^{N_y}$
- Classification: Each output is one of a finite set of labels $\boldsymbol{y} \in \{1, 2, \ldots, K\}$
- Density modelling: Model the probability density of the inputs $\boldsymbol{x}$, without any reference to a target output $\boldsymbol{y}$.
- Regression and classification are "supervised learning", whereas density modelling is "unsupervised"

# Statistical modelling vs machine learning

| **Statistics** | **Machine Learning** |
|---|---|
| pen & paper | computer-focused |
| optimality, convergence | computational speed |
| equations | algorithms |
| interpretability | predictive performance |
| parsimony | complexity |
| (probabilistic | deterministic) |

# Deep Neural Networks

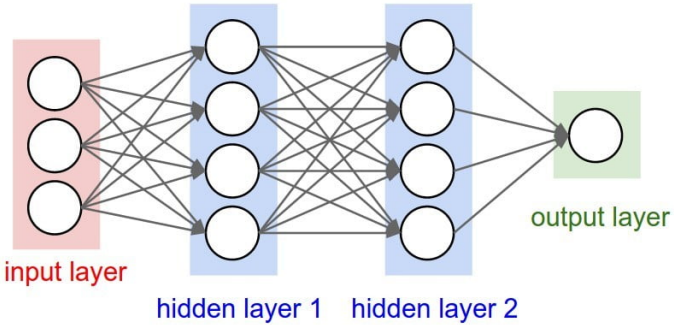
- Construct a complicated function $f(\boldsymbol{x})$ by composing relatively simple standard functions

$$f(\boldsymbol{x}) = f^{(p)}(f^{(p-1)}(...(f^{(2)}(f^{(1)}(\boldsymbol{x})))...))$$

- where typically each "layer" $f^{(i)}$ involves
  - a linear transformation $W\boldsymbol{x} + \boldsymbol{b}$ applied to the inputs $\boldsymbol{x}$ (where $W$ and $\boldsymbol{b}$ contain trainable weights); and
  - a nonlinear "activation" function applied element-wise.
  - For example $f^{(i)}(\boldsymbol{x}) = tanh(W\boldsymbol{x} + \boldsymbol{b})$
- the functions $f^{(1)}, \ldots, f^{(p-1)}$ are "hidden layers"
- More complicated transformations are in use these days (especially in LLMs), but linear + activation is the most common.

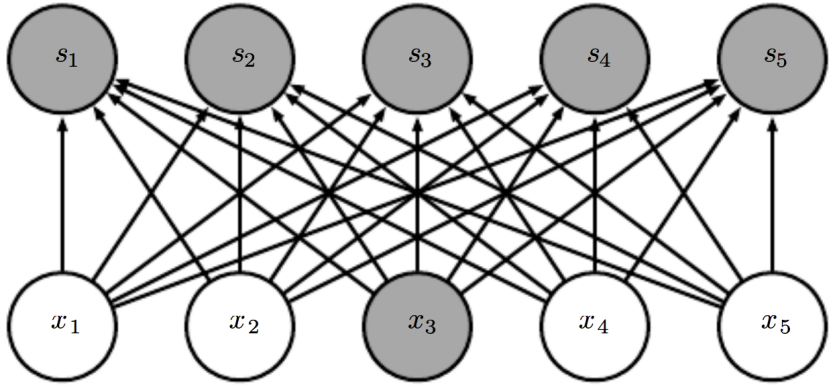
# Fully connected layers

A "layer" is a standard type of operation, that was found empirically to be a generic and efficient building block in network architectures.




Fully connected ("Dense") layers:

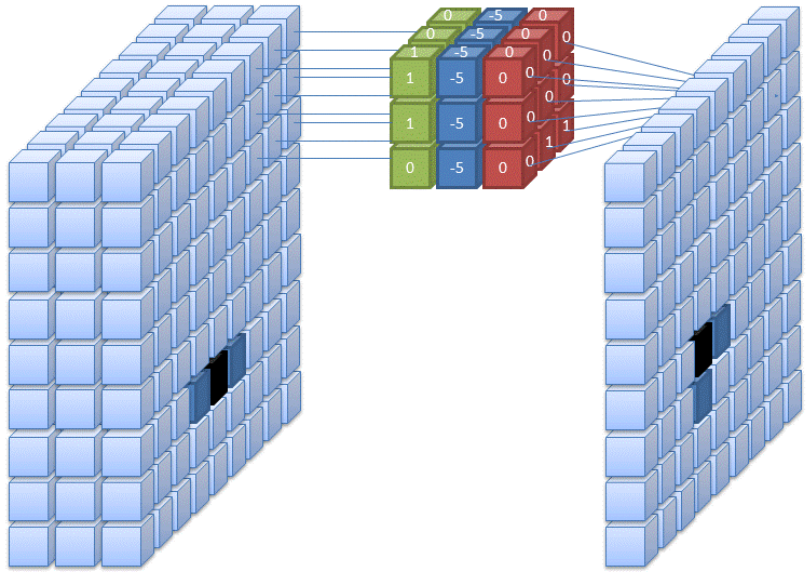
- Layer input of size $D$ and layer output of size $D'$, and each output can depend on each of the inputs.




- The transformation $W\boldsymbol{x} + \boldsymbol{b}$ has $D \times D' + D'$ trainable parameters.

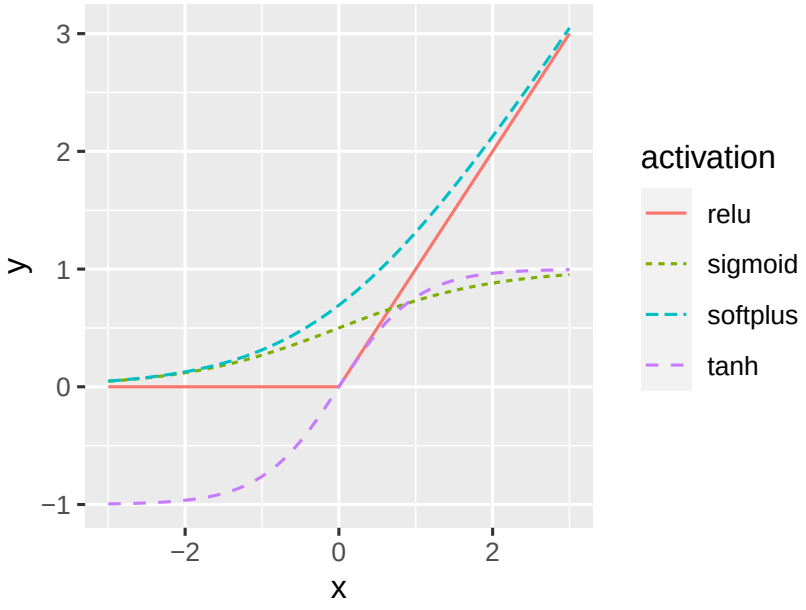
# Convolutional layer

- Input layer of shape $N \times M \times C$, for example 2-d image with $N \times M = 640 \times 480$ pixels and $C = 3$ color channels.
- Output has shape $N \times M \times D$.



- Kernel of width $K$ calculates local linear transformations of the inputs.
- $K^2 \times C \times D + D$ trainable parameters.
- $K$ and $D$ are metaparameters, with common choices $K = \{3, 5\}$ and $D = \{64, 128, 256\}$.

# Activation functions




- $Lin(x) = x$
- $ReLU(x) = max(0, x)$
- Sigmoid: $\sigma(x) = [1 + \exp(-x)]^{-1}$
- $softplus(x) = \log(1+\exp(x))$
- $tanh(x) = \frac{\exp(x)-\exp(-x)}{\exp(x)+\exp(-x)}$
- $softmax(\boldsymbol{x})_i = \exp(x_i)/\sum_j \exp(x_j)$

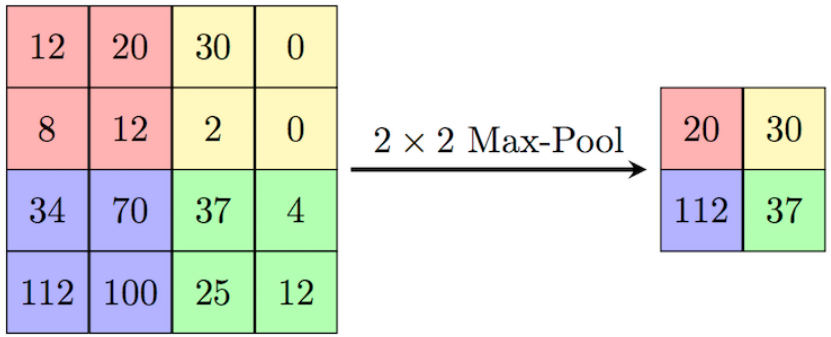
Common properties:

- Monotonic
- Nonlinear
- Applied element-wise to tensor-valued inputs
- Saturate in at least one direction

**ReLU is generally recommended as the default choice.**

# Pooling layers

- Input layer of shape $N \times M \times C$ to
- Pooling window size $D$
- Output layer output of shape $N/D \times M/D \times C$.




- No trainable parameters.
- **Max-pooling is most common, but average-pooling is also used.**

# Neural networks are universal function approximators

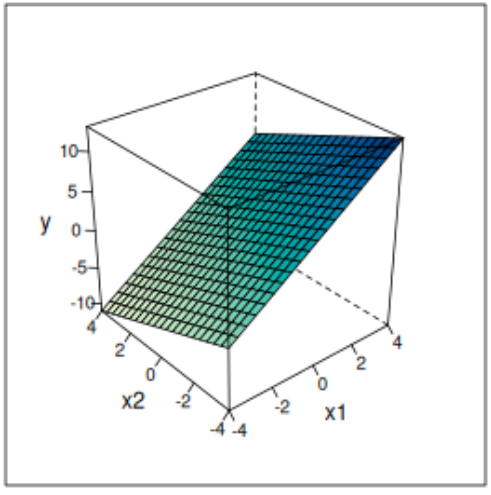
$$z_1 = \sigma(w_{10} + w_{11}x_1 + w_{12}x_2)$$

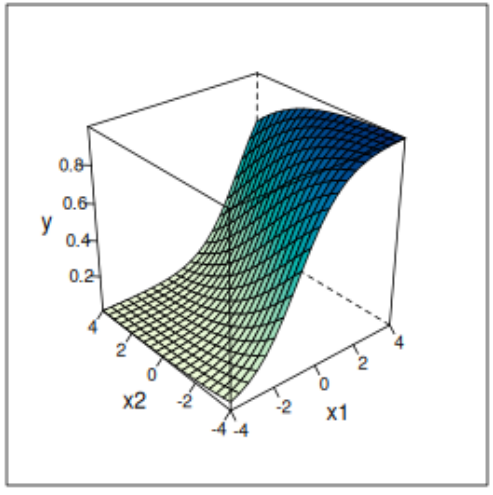
$$z_2 = \sigma(w_{20} + w_{21}x_1 + w_{22}x_2)$$

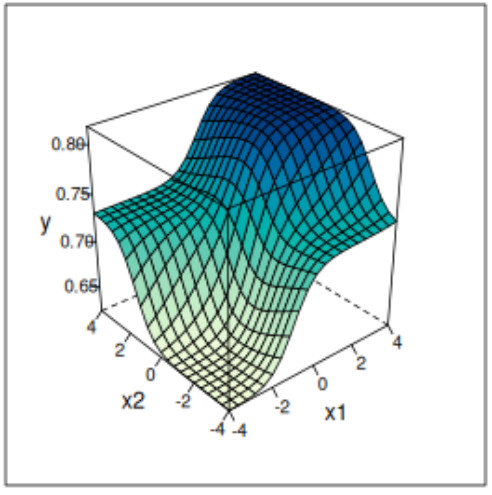
$$y = w_0 + w_1x_1 + w_2x_2$$

$$y = \sigma(w_0 + w_1x_1 + w_2x_2)$$

$$y = \sigma(w_{30} + w_{31}z_1 + w_{32}z_2)$$




Sufficiently deep neural networks can approximate any continuous function arbitrarily closely.

# Optimisation: Loss functions

Some of the most commonly used loss functions for regression are

- Mean squared error (MSE): $\ell(\hat{\boldsymbol{y}}, \boldsymbol{y}) = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2$
- Mean absolute error (MAE): $\ell(\hat{\boldsymbol{y}}, \boldsymbol{y}) = \frac{1}{n}\sum_{i=1}^{n}|\hat{y}_i - y_i|$

For classification, we usually use

- Binary cross entropy: $\ell(\hat{y}, y) = -y\log(\hat{y}) - (1-y)\log(1-\hat{y})$ where $\hat{y} \in (0,1)$ and $y \in \{0,1\}$
- Categorical cross entropy: $\ell(\hat{\boldsymbol{y}}, \boldsymbol{y}) = -\sum_{i=1}^{C} y_i \log \hat{y}_i$ where $\hat{\boldsymbol{y}} \in (0,1)^C$ and $\boldsymbol{y} \in \{0,1\}^C$

# Optimisation: Gradient Descent

For learning, our goal is to find $\mathbf{w}$ such that for a given input $\boldsymbol{x}$, the neural network output $\hat{\boldsymbol{y}}$ and the corresponding truth $\boldsymbol{y}$ are as close as possible.
The gradient of the loss function $L(\mathbf{w})$:

$$\frac{\partial L}{\partial \mathbf{w}} = \begin{pmatrix} \frac{\partial L}{\partial w_1} \\ \frac{\partial L}{\partial w_2} \\ \vdots \end{pmatrix}$$

points in the direction of the steepest increase of $L(\boldsymbol{w})$.
Gradient Descent is a stepwise algorithm to iteratively find weight vectors $\mathbf{w}$ with lower and lower loss function values:

$$\mathbf{w}_{i+1} = \mathbf{w_i} - \gamma \frac{\partial L}{\partial \mathbf{w}}\bigg|_{\boldsymbol{w}_i}$$

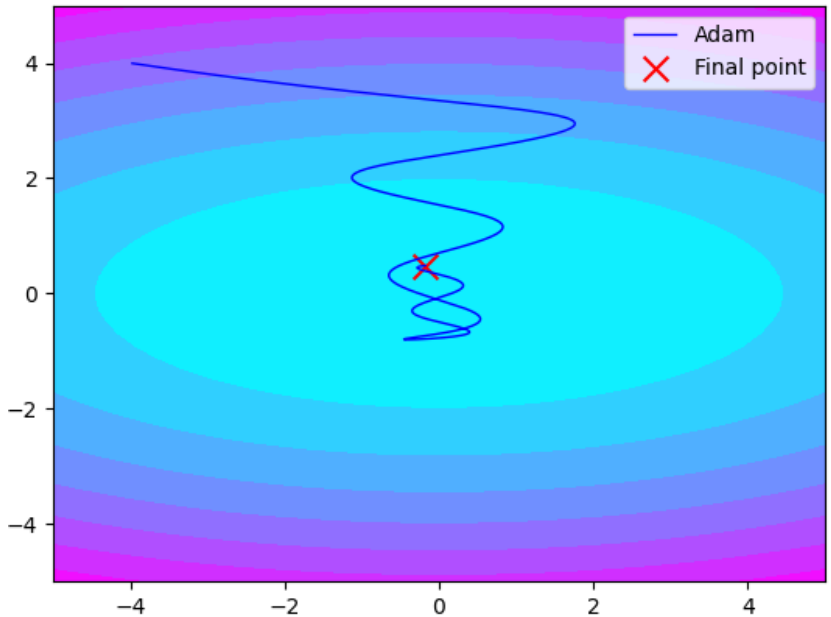
The step length $\gamma$ is a metaparameter called the learning rate.

# Stochastic gradient descent (SGD)

- In gradient descent, the loss function is calculated on all $n$ training data, e.g. $L(\mathbf{w}) = \sum_{i=1}^{n} \ell(y_i, \hat{y}_i)$.
- With large training data sets and complex models this is time consuming and prone to getting stuck in local minima.
- SGD addresses both issues by using a small random subset (batch) of the entire training data.
- On each iteration, a different batch is used to calculate the loss and its gradient.
- The time after which all the training data has been used exactly once is called an epoch.
- The small batches make gradient calculations more efficient, and the ever-changing shape of the loss function improves convergence.

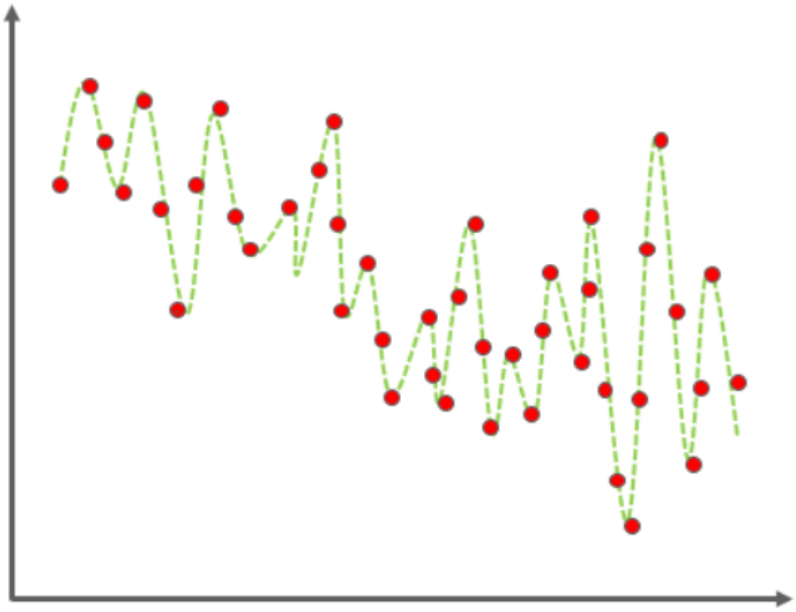
# Adam (a popular version of SGD)

- Adam uses exponentially smoothed gradient calculations (weighted averages of past gradients); and
- a different learning rate for each weight ($\gamma \to (\gamma_1, \gamma_2, \dots)'$).



# Overfitting

With their large number of trainable parameters, deep neural networks are prone to overfitting.



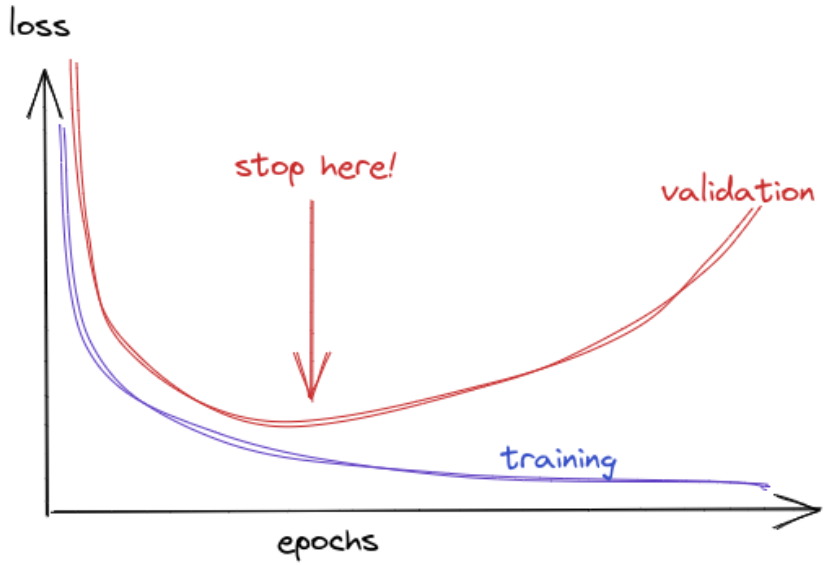
Overfitted models fit the training data extremely well, but perform poorly when applied to data that was not seen during training.

# Avoiding overfitting

## Cross validation

- split the training data into a training and a validation data set
- training data is used for fitting
- the held-out validation data is used to monitor the generalisation error

## Early stopping



# Avoiding overfitting

## Gradually reducing the learning rate

- gradually reduce the step size $\gamma$ used in gradient descent
- smaller and smaller parameter updates reduce risk of overfitting
- no hard rules on how fast learning rate should reduce

## Dropout layer

- a dropout layer between two densely connected hidden layers
- randomly set a fraction of weights to zero
- introduces error but helps robustness

## Weight regularisation

- overfitted models are characterised by large weights
- adding a term to the loss function that penalised large weights
- find compromise between good model fit and small weights

# Tensorflow


- `www.tensorflow.org`
- open-source machine learning platform
- provides tools and libraries for constructing, training, and applying neural network models
- tutorials, interactive examples, community support

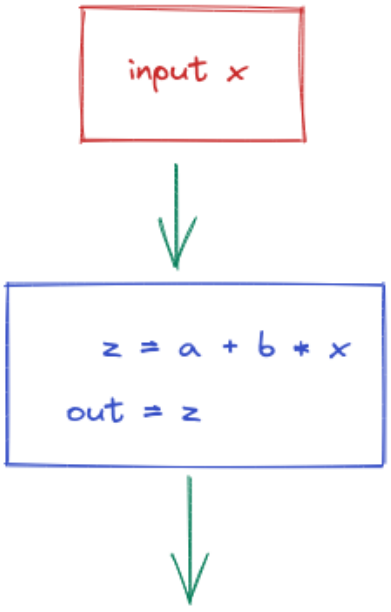
```
import tensorflow as tf
```

# Keras


- `www.keras.io`
- high-level machine learning API built on top of tensorflow
- easy construction, training, debugging of standard types of neural networks
- less flexible but more beginner-friendly than tensorflow
- **We are using keras in this workshop.**

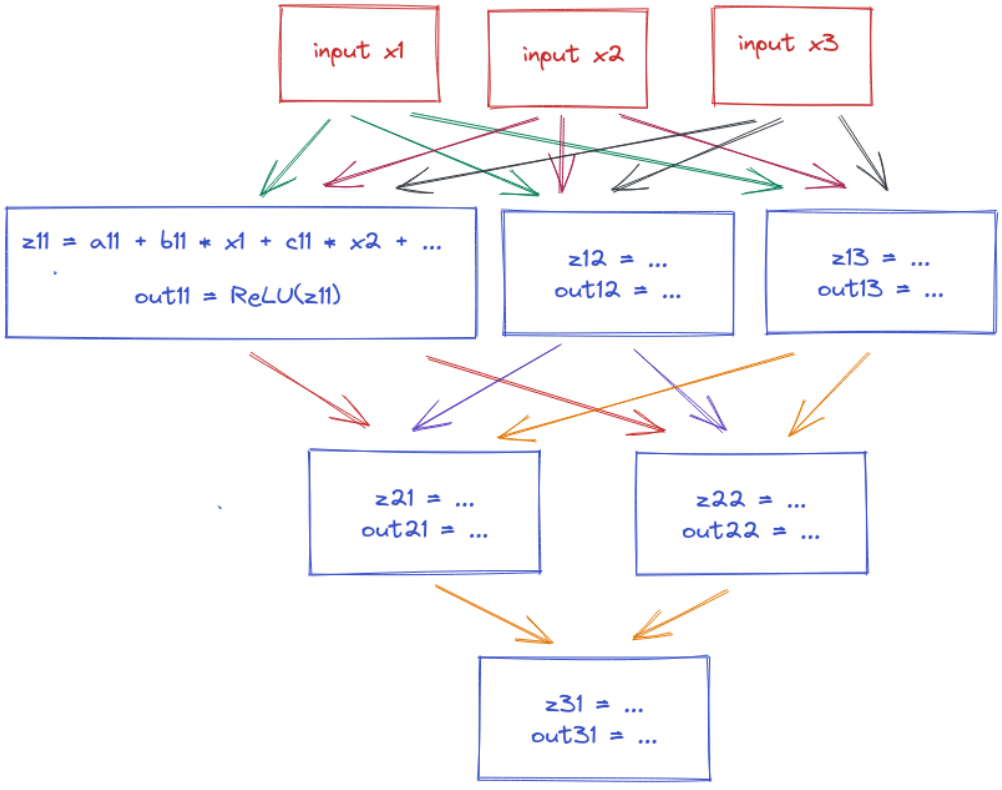
```
import keras
```

# Simple Linear Regression in Keras




```
import keras

# define model architecture by composing layers
inputs  = keras.layers.Input(shape=(1))
outputs = keras.layers.Dense(
            1, activation='linear')(inputs)
model   = keras.Model(inputs, outputs)

# compile the model (specify optimisation
# algorithm and loss function)
model.compile(optimizer='adam', loss='mse')

# fit the model to training data
# (inputs x, outputs y)
model.fit(x, y, batch_size=n, epochs=1000)

# calculate fitted values
y_pred = model(x)
```

# "Deep" Multiple Regression



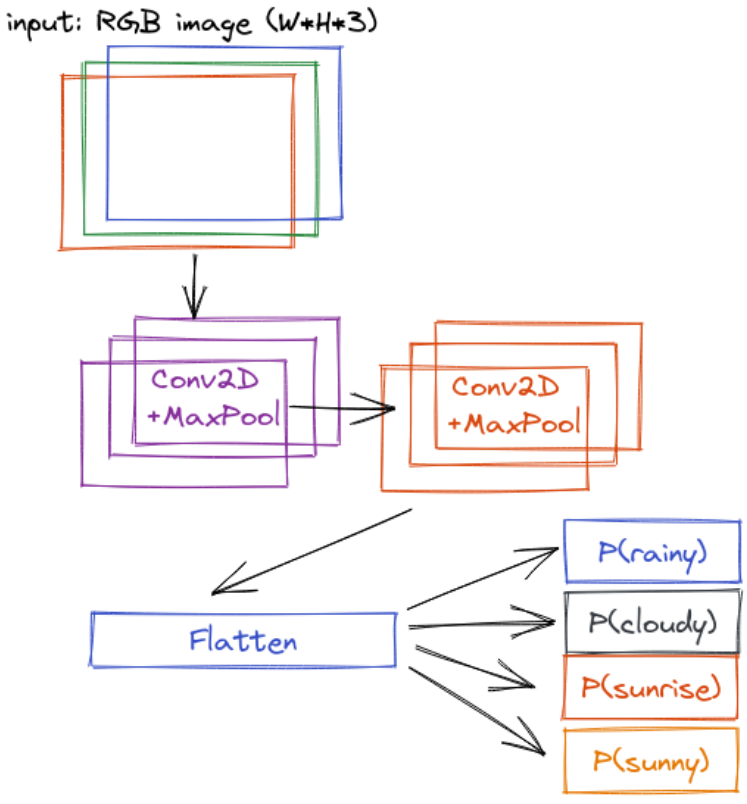

- 3 inputs $(x_1, x_2, x_3)$ mapped to 1 output $y$
- 2 hidden layers with 3 and 2 nodes
- each node represents a transformation, characterised by weights and activation function
- nonlinear activation function: ReLU(x) = max(x, 0)
- typical choices in practice: 3-10 hidden layers, each with 30-200 nodes

```
inputs  = keras.layers.Input(shape=(3))
layer1  = keras.layers.Dense(3, activation='relu')(inputs)
layer2  = keras.layers.Dense(2, activation='relu')(layer1)
outputs = keras.layers.Dense(1, activation='linear')(layer2)
model   = keras.Model(inputs, outputs)
```

# Image classification

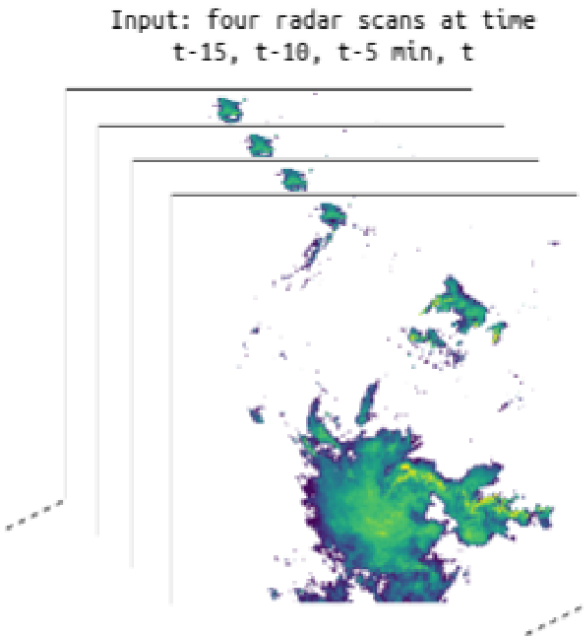

- training dataset of 1200 images, each labelled as cloudy/rain/sunrise/shine
- use keras with convolution and max-pooling layers to fit an image classifier
- input: RGB image of shape 150 x 150 x 3
- output: vector of 4 probabilities

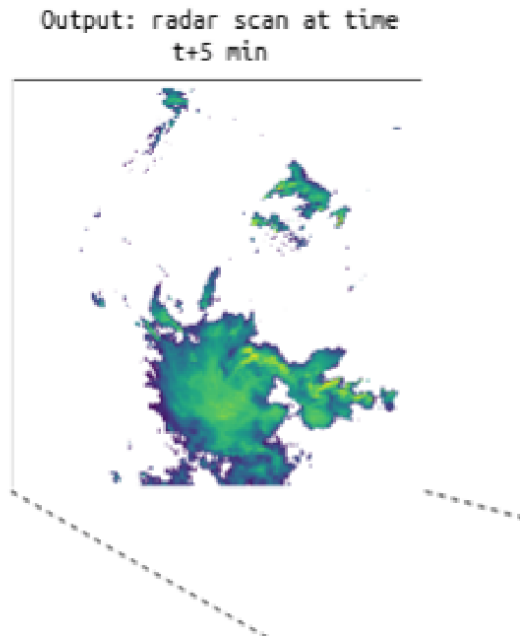
# Image classification




```
model = keras.Sequential([
  keras.layers.Input(shape=(150,150,3)),
  keras.layers.Conv2D(32, (3,3),
    activation='relu', padding='same'),
  keras.layers.MaxPool2D(pool_size=(2,2)),
  keras.layers.Conv2D(16, (3,3),
    activation='relu', padding='same'),
  keras.layers.MaxPool2D(pool_size=(2,2)),
  keras.layers.Flatten(),
  keras.layers.Dense(
    200, activation='relu'),
  keras.layers.Dense(
    4, activation='softmax')
])
```

# Next frame image prediction (rain radar nowcasting)




- $\sim$ 1000 examples of 5 consecutive radar images
- from Met Office Nimrod data set, between 2014 and 2020
- 5km spatial resolution, 15 minutes temporal resolution
- we train a model that takes the first 4 images as input, and predicts the 5th image
- using convolution and max-pooling in a "U-Net" architecture

# U-Net for next image prediction (rain radar nowcasting)