Deep Learning: Introduction

Adam Moss

School of Physics and Astronomy

adam.moss@nottingham.ac.uk

The University of Nottingham

UNITED KINGDOM · CHINA · MALAYSIA

Early Learning

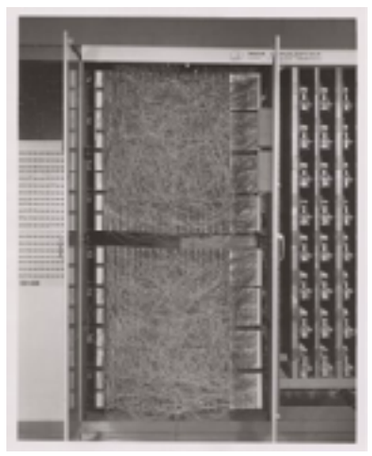
The University of Nottingham
UNITED KINGDOM · CHINA · MALAYSIA

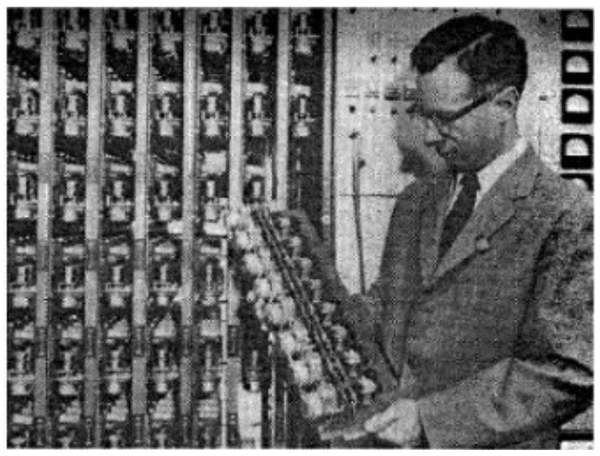
- Feature extraction

Hand crafted feature extraction

Trainable classifier

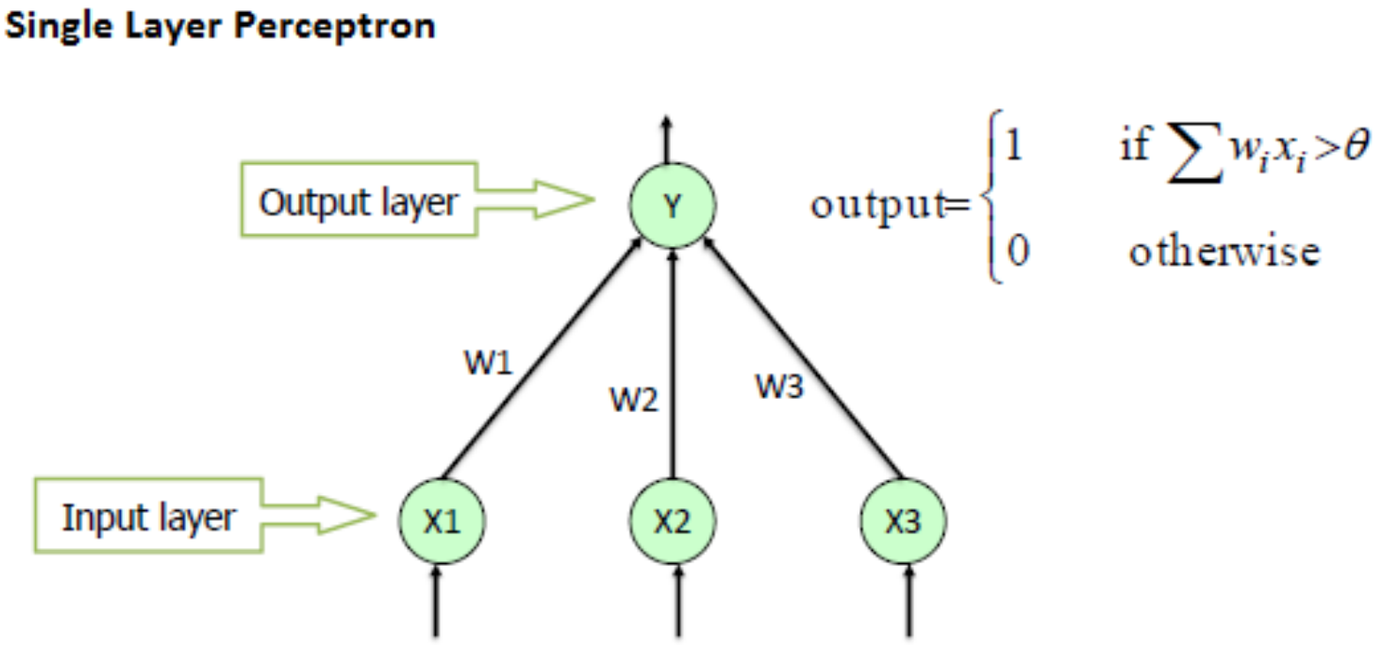
- Perceptron (binary classifier). Mark 1 perceptron machine (1957) used motors, potentiometers!





Perceptron

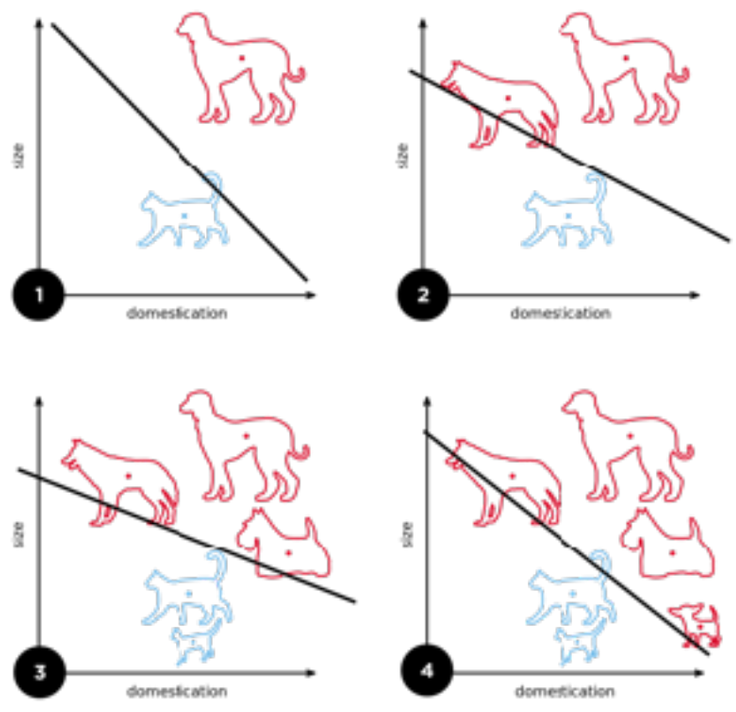
- Simplest perceptron: set of inputs x_i mapped to output $\hat{y}$



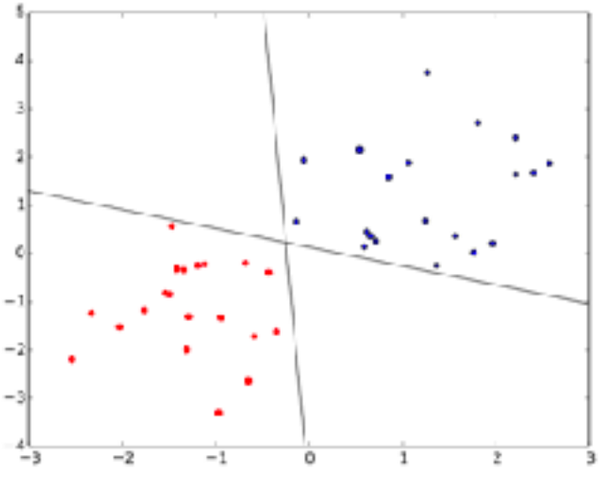
- Each input has a weight w_i
- Weights are trained using **supervised learning**
- Training sets of $D = \{x_{i,j}, y_j\}$ where j is the sample number and y_j the desired output for that sample
- Prediction is $\hat{y}$ and weights are updated to minimise loss $\sum_j (y_j - \hat{y}_j)^2$

Perceptron

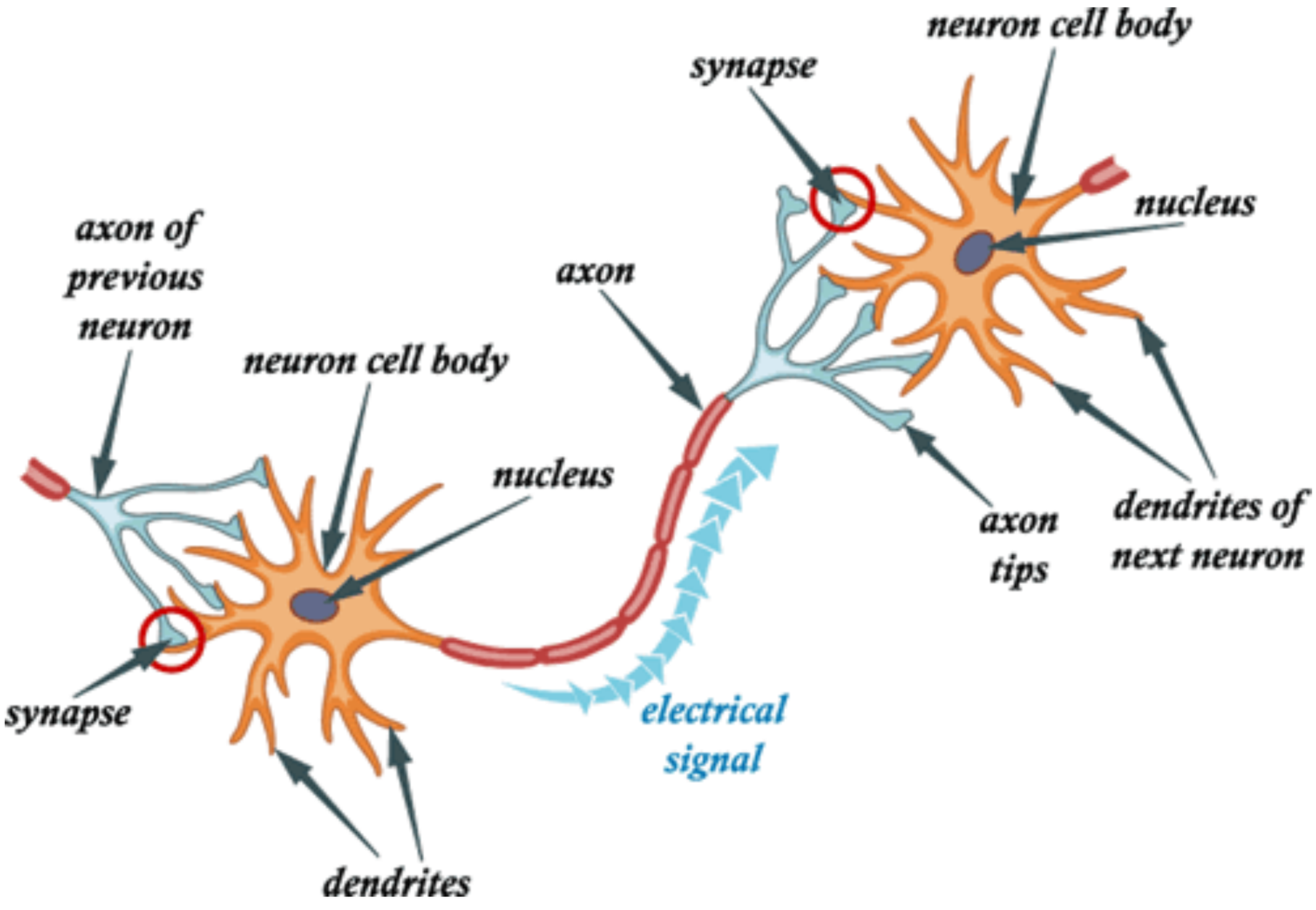
- The perceptron is only able to classify linearly separable training sets
- E.g. if two features are size and domestication



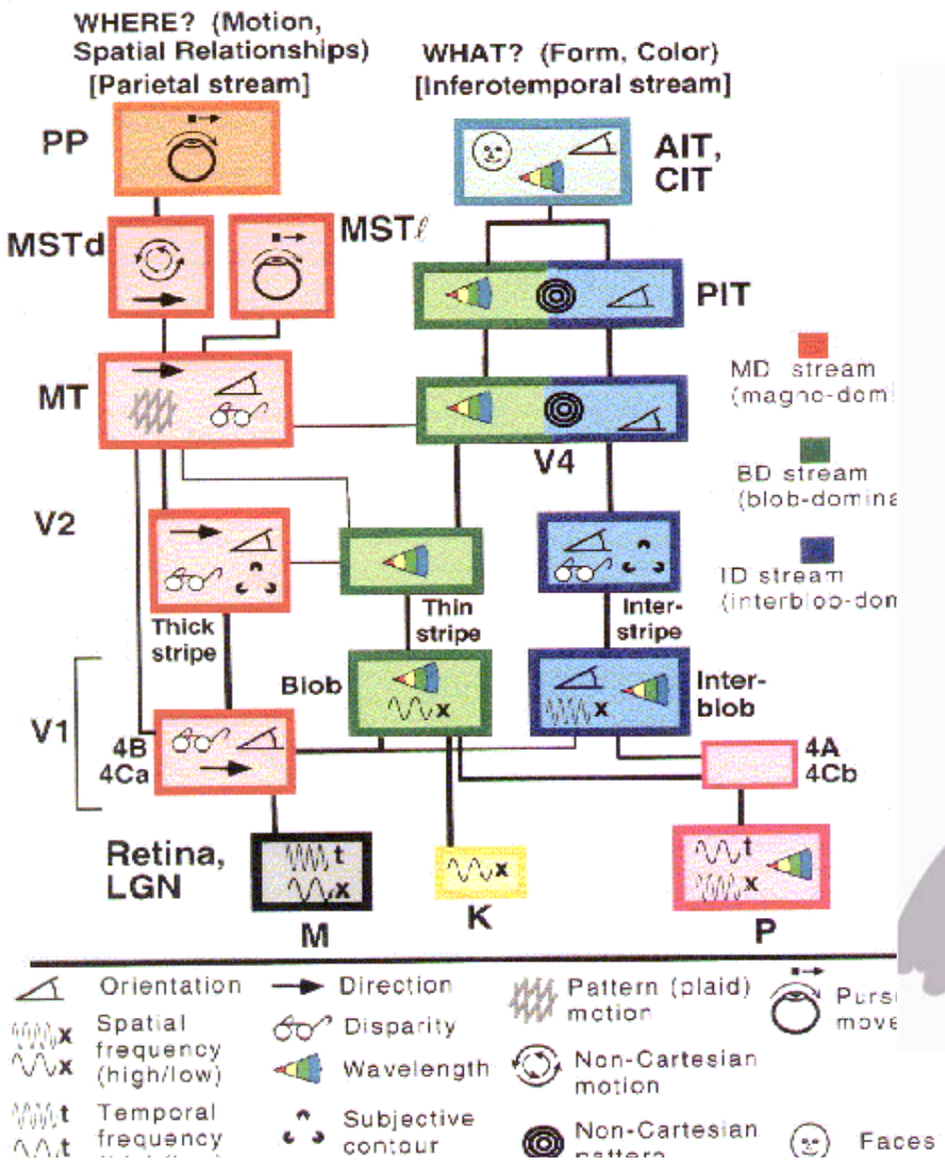
- May admit solutions of different quality (general problem for machine learning if training data is not representative)

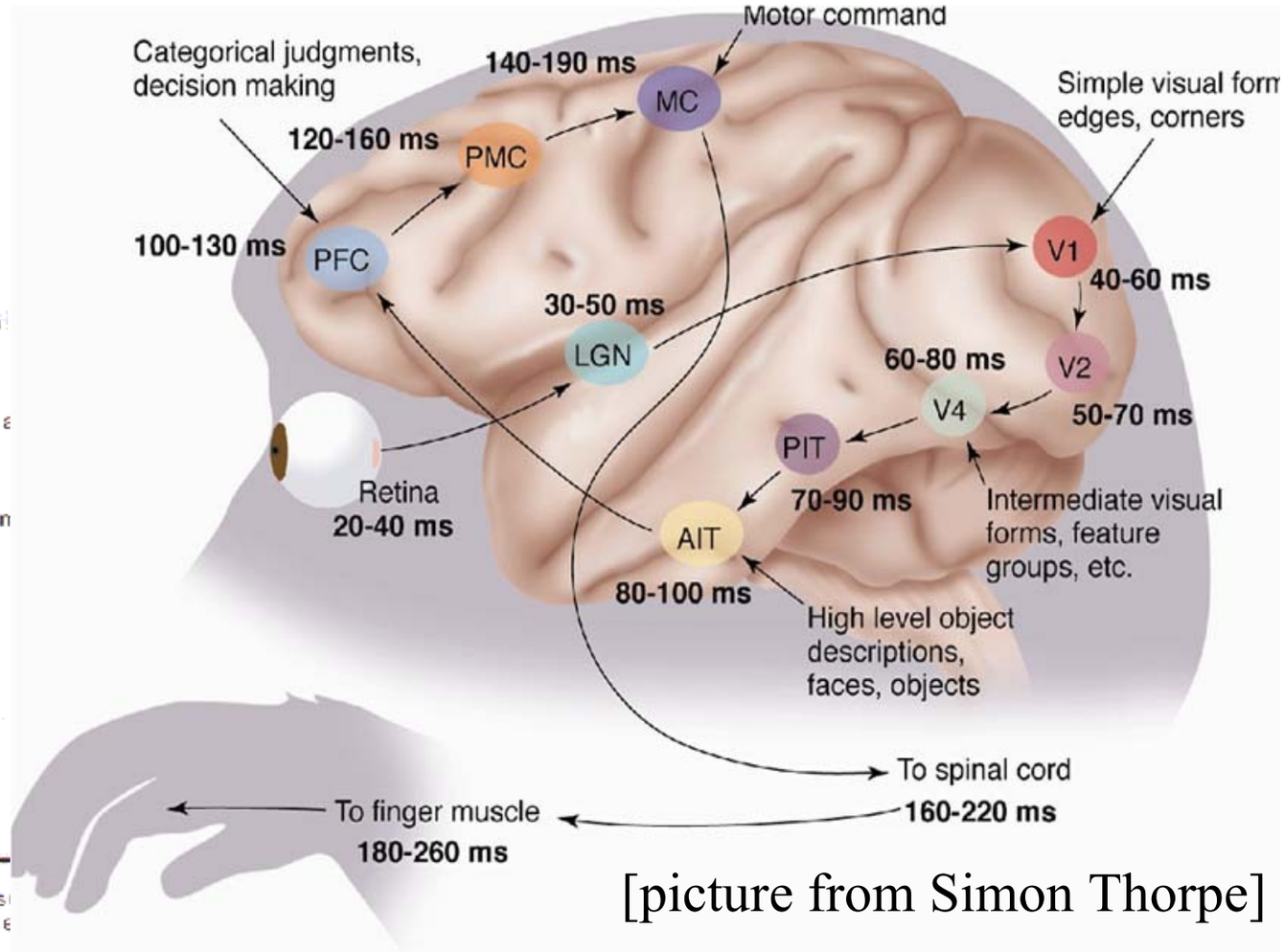


Biological Neuron



Visual Cortex





[picture from Simon Thorpe]

[Gallant & Van Essen]

LGN=lateral geniculate nucleus

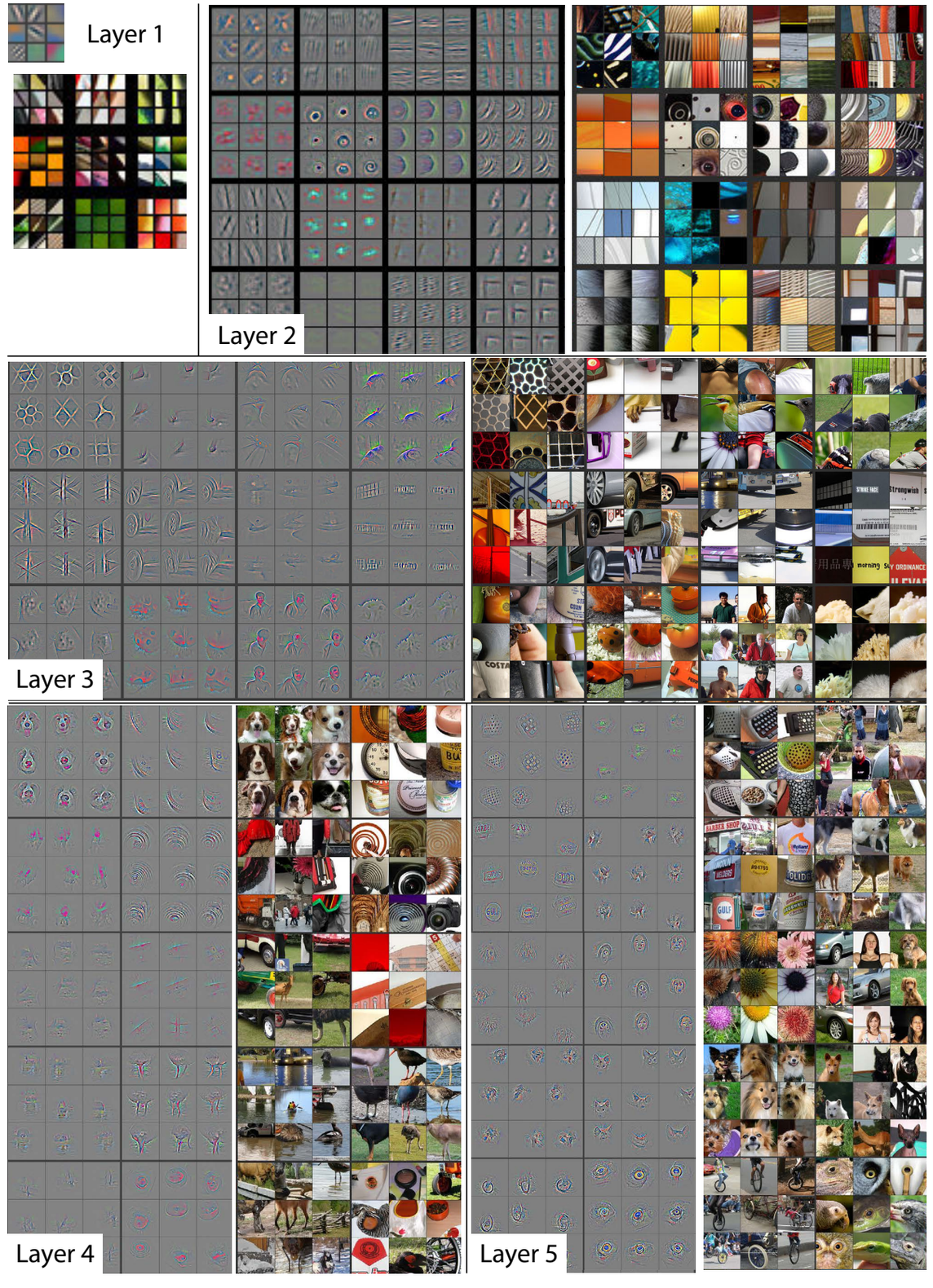
AIT=anterior inferior temporal cortex

PIT=posterior inferior temporal cortex

Shallow vs Deep

The University of Nottingham
UNITED KINGDOM · CHINA · MALAYSIA

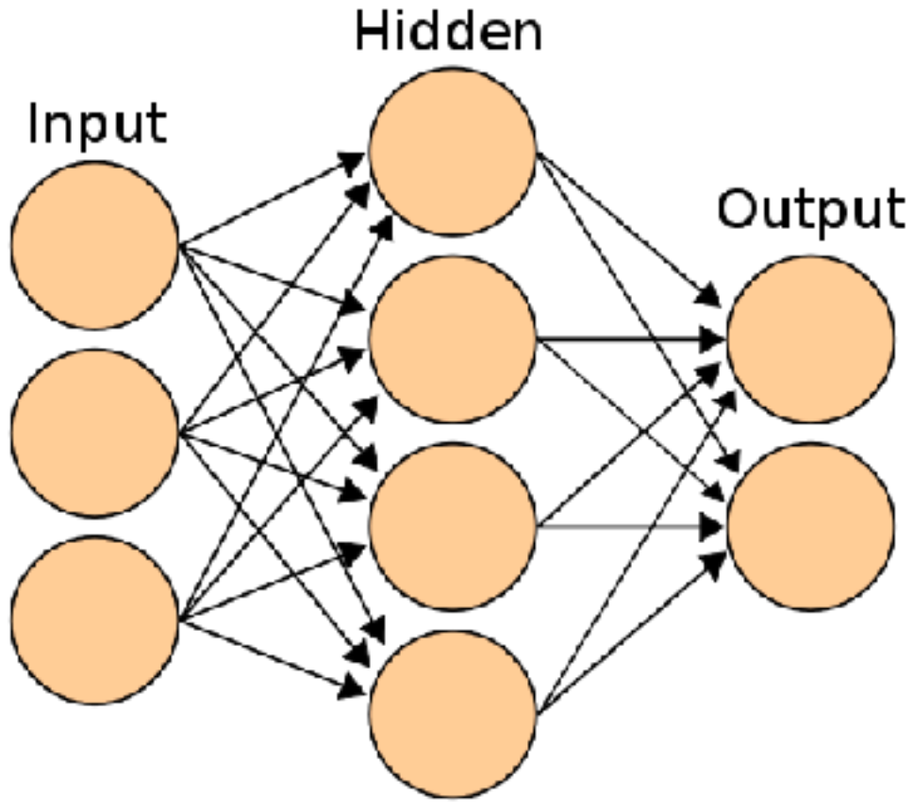
- A network is **deep** if it has more than one layer of non-linear feature abstraction
- Hierarchy of representations with increasing levels of abstraction (e.g. pixel -> edge -> eye -> face)
- Deep networks can store more memory than equivalent number of units in a single layer



MultiLayer Perceptron

The University of Nottingham
UNITED KINGDOM · CHINA · MALAYSIA

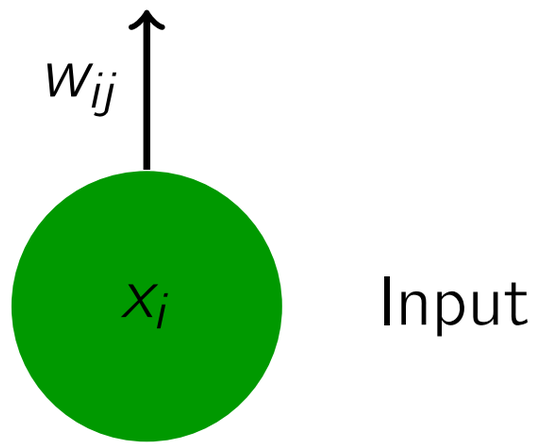
- Type of **feedforward** artificial neural network
- Can distinguish data which is not linearly separable
- Some neurons use non-linear **activation** functions (functions which map the weighted input to the output of a neuron)
- The brain is thought to work in a similar way when biological neurons are fired
- MLPs use supervised learning to update network weights using **back-propagation**



Back-propagation

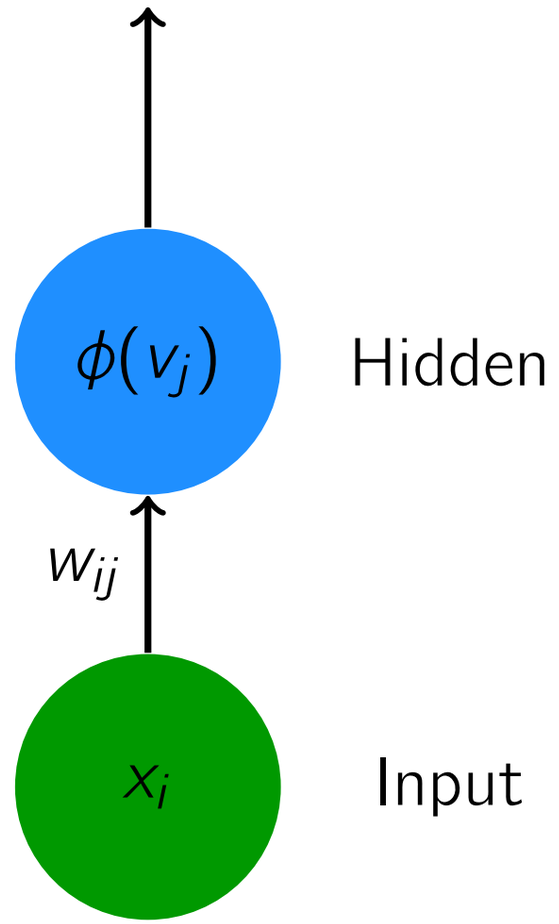
The University of Nottingham
UNITED KINGDOM · CHINA · MALAYSIA

- Weighted inputs $v_j = w_{ij} x_i$



Back-propagation

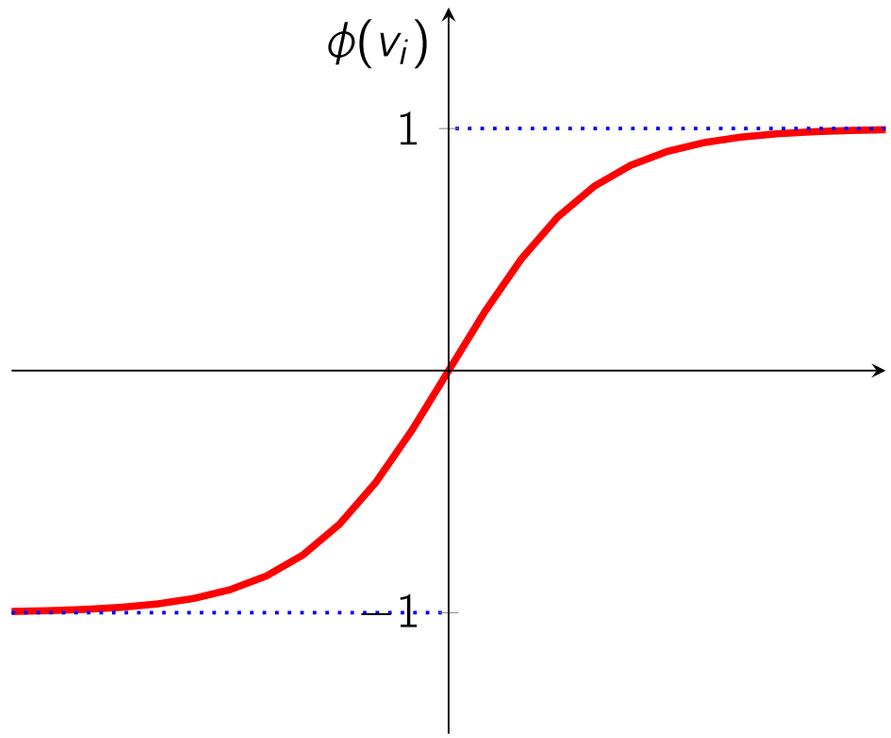
- Weighted inputs $v_j = w_{ij} x_i$
- Activation function $\phi(v_j)$



Back-propagation

- Weighted inputs $v_j = w_{ij} x_i$
- Activation function $\phi(v_j)$

$$\phi(v_i) = \tanh(v_i)$$



Tanh function

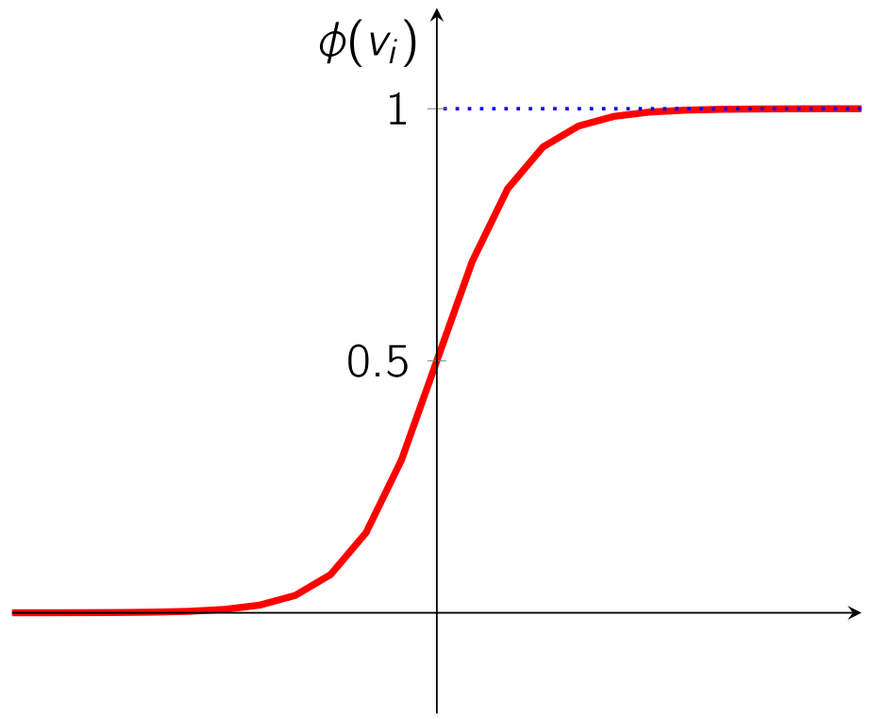
Back-propagation

The University of Nottingham
UNITED KINGDOM · CHINA · MALAYSIA

- Weighted inputs $v_j = w_{ij} x_i$
- Activation function $\phi(v_j)$

$$\phi(v_i) = \tanh(v_i)$$

$$\phi(v_i) = (1 + e^{-v_i})^{-1}$$



Sigmoid function

Back-propagation

- Weighted inputs $v_j = w_{ij} x_i$
- Activation function $\phi(v_j)$

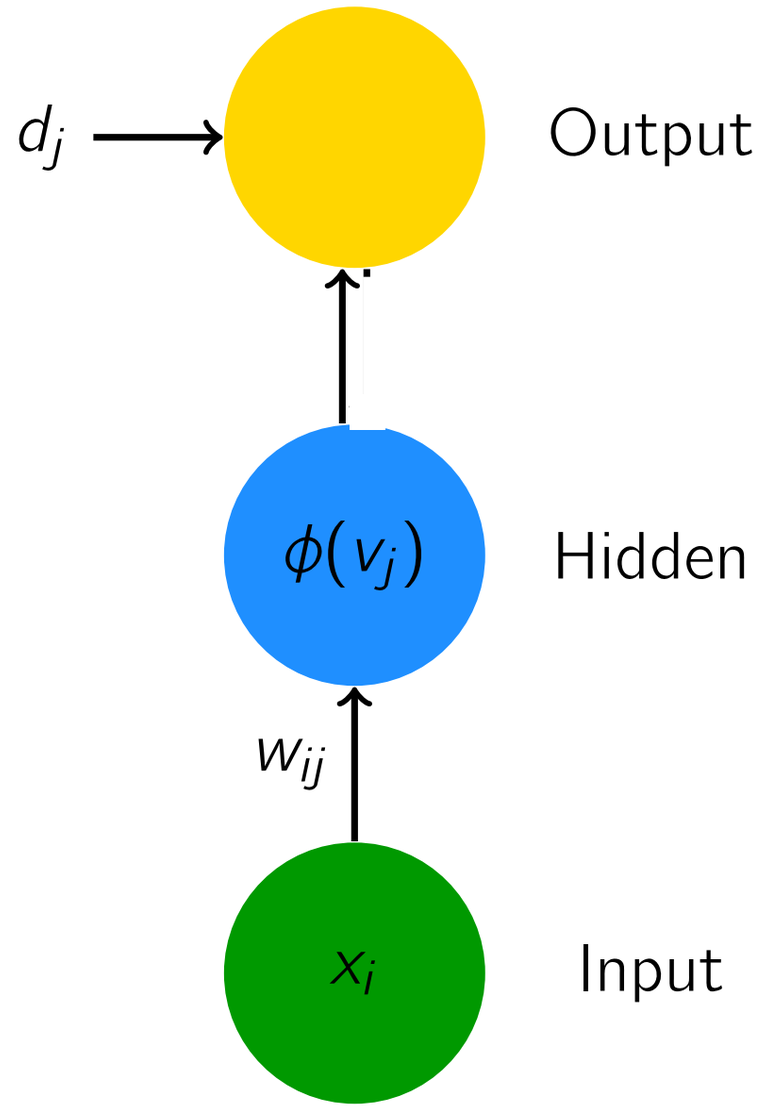
$$\phi(v_i) = \tanh(v_i)$$

$$\phi(v_i) = (1 + e^{-v_i})^{-1}$$

- Loss function

$$e_j = y_j - \phi(v_j)$$

$$\mathcal{E} = \sum_j e_j^2$$



Back-propagation

The University of Nottingham
UNITED KINGDOM · CHINA · MALAYSIA

- Weighted inputs $v_j = w_{ij} x_i$
- Activation function $\phi(v_j)$

$$\phi(v_i) = \tanh(v_i)$$

$$\phi(v_i) = (1 + e^{-v_i})^{-1}$$

- Loss function

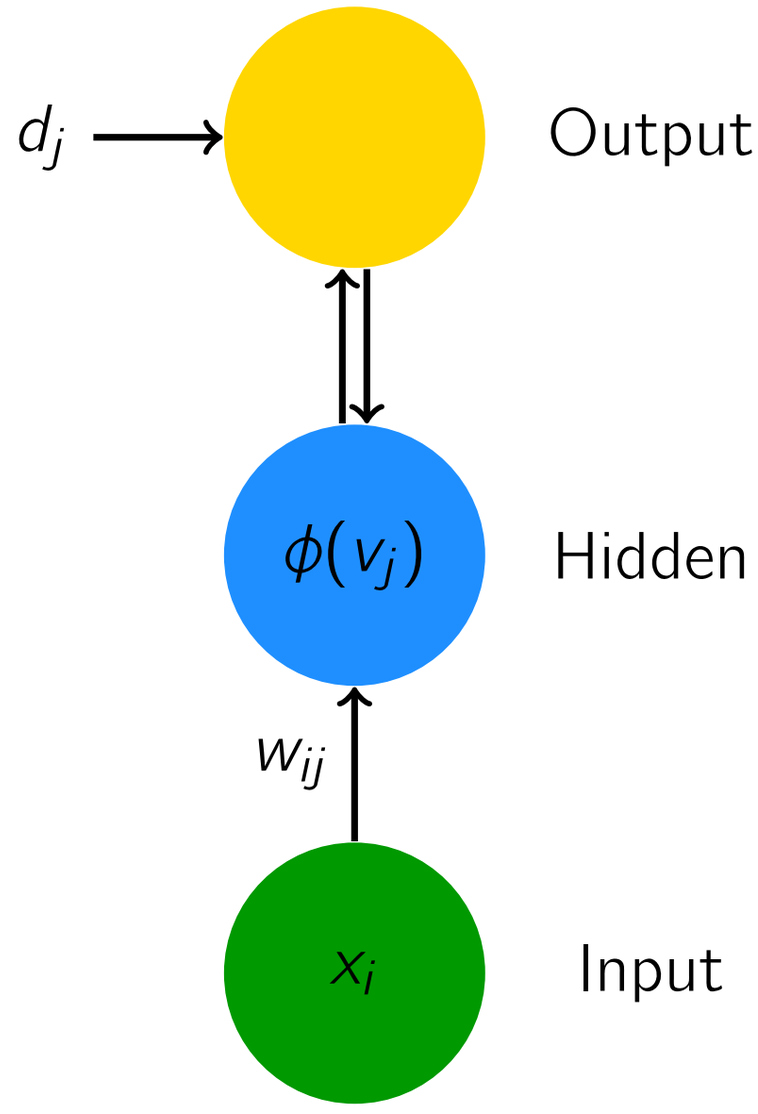
$$e_j = y_j - \phi(v_j)$$

$$\mathcal{E} = \sum_j e_j^2$$

- Back-propagation

$$\Delta w_{ij} = -\eta \frac{\partial \mathcal{E}}{\partial w_{ij}}$$

$$\Delta w_{ij} = -\eta \frac{\partial \mathcal{E}}{\partial \phi(v_j)} \frac{\partial \phi(v_j)}{\partial v_j} \frac{\partial v_j}{\partial w_{ij}}$$



Back-propagation

The University of Nottingham
UNITED KINGDOM · CHINA · MALAYSIA

- Can the cortex do back-propagation?
- Maybe or maybe not!
- Many neuroscientists think the brain **can't** back-propagate
  - ➡ Source of supervision signal?
  - ➡ Neurons send all-or-nothing spikes
  - ➡ Neurons must be able to send different signals forward and backward
- But see work of Geoffrey Hinton (e.g. Lillicrap et. al., Nature Communications **7,** 2016) who presents arguments how the brain can back-propagate
- How much should we be led by the function of the brain when developing deep learning algorithms?

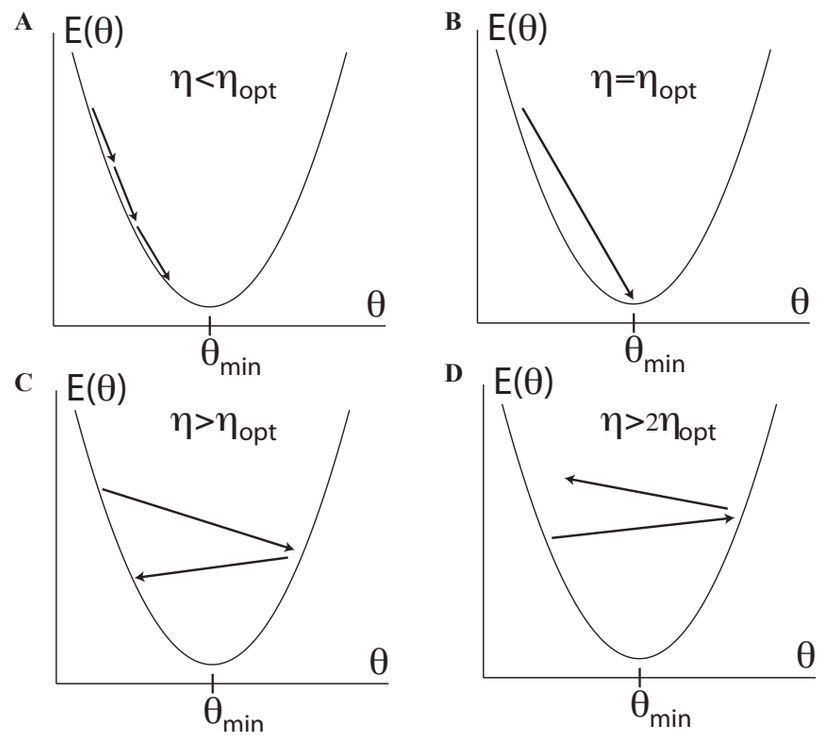
Gradient Descent (GD)

- Most supervised machine learning algorithms involve finding a set of parameters **Θ** that minimise a cost/loss function
- Loss functions can be complicated non-convex functions with many local minima
- Simplest algorithm that attempts to minimise the loss E(**Θ**) is gradient descent

$$\mathbf{v}_t = \eta_t \nabla_\theta E(\boldsymbol{\theta}_t),$$
$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \mathbf{v}_t$$

η_t = learning rate

- Requires careful choice of learning rate
- Can become stuck in local minima
- Sensitive to initial conditions
- Learning rate is same for all gradients
- Long time to escape saddle points



Stochastic Gradient Descent (SGD)

- Stochasticity is added by approximating the gradient on a subset of data called a **mini batch**
- Size of mini batch is smaller than dataset - typically 10's to 100's of data points
- Full iteration using all mini batches of a dataset is called an **epoch**
- Update rule is now just

$$\mathbf{v}_t = \eta_t \nabla_\theta E^{MB}(\boldsymbol{\theta}),$$
$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \mathbf{v}_t.$$

- Decreases chance of becoming stuck in local minima
- Also has been shown to help alleviate **over-fitting**

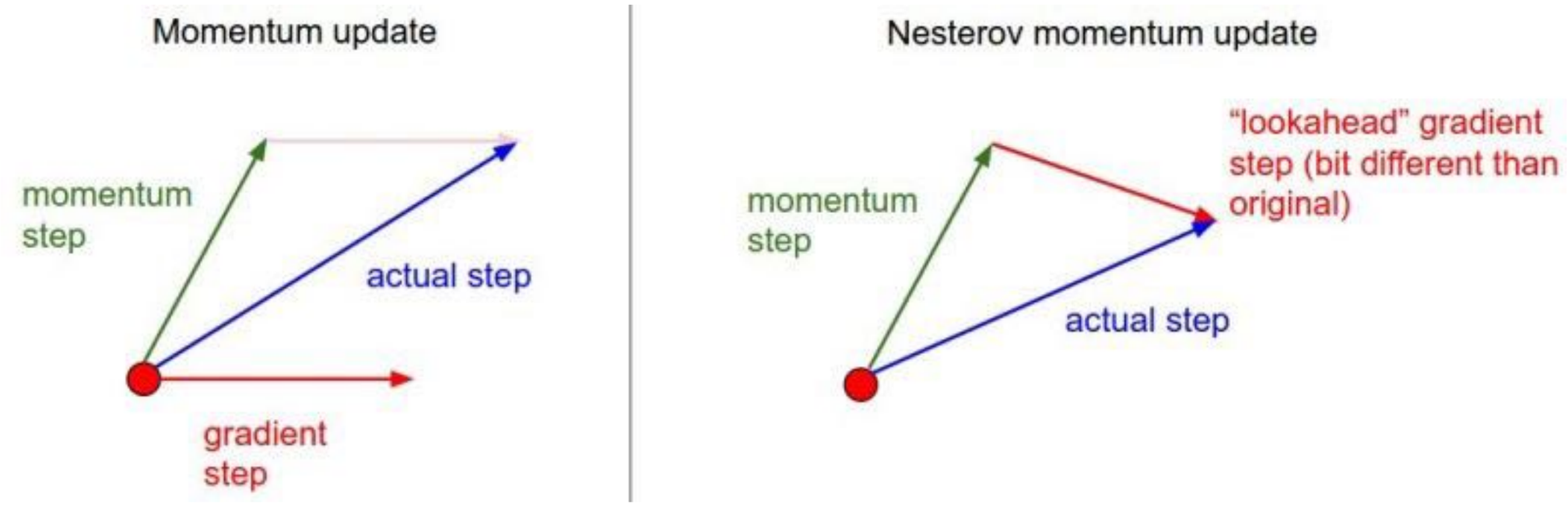
SGD with momentum

- SGD with momentum adds an inertia term that retains some memory of the direction being moved in

$$\mathbf{v}_t = \gamma \mathbf{v}_{t-1} + \eta_t \nabla_\theta E(\boldsymbol{\theta}_t)$$
$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \mathbf{v}_t,$$

$\gamma = \text{momentum parameter}$

- SGD helps parameter updates gain ‘speed’ in persistent smaller gradients while suppressing oscillatory high gradients



Second order moments

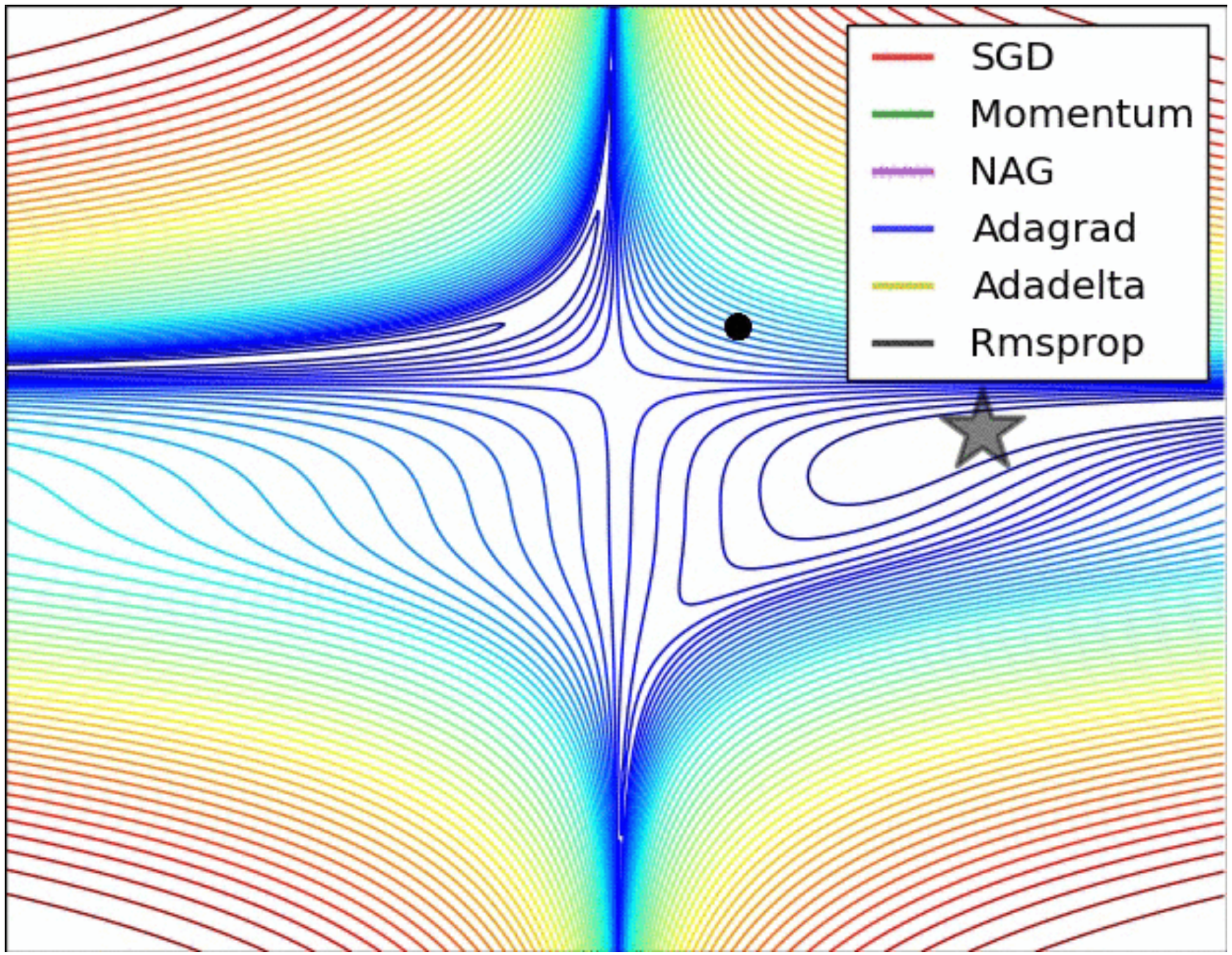
- All methods so far require a schedule for the learning rate as a function of time
- Optimal learning rate is actually inverse of the Hessian $\eta_{\text{opt}} = [\partial_\theta^2 E(\theta)]^{-1}$.
- Expensive to compute. Second order moment methods keep track of the **squared gradient**
- Take large steps in shallow directions and small steps in steep directions
- Algorithms include RMSprop (Teileman and Hinton 2012), AdaDelta (Zeiler, 2012) and ADAM (Kingma and Ba, 2014)
- E.g. RMSprop update rule

$$\mathbf{g}_t = \nabla_\theta E(\boldsymbol{\theta}) \qquad \mathbf{s}_t = \mathbb{E}[\mathbf{g}_t^2]$$

$$\mathbf{s}_t = \beta \mathbf{s}_{t-1} + (1-\beta)\mathbf{g}_t^2 \qquad \beta = \text{averaging time}$$

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta_t \frac{\mathbf{g}_t}{\sqrt{\mathbf{s}_t + \epsilon}}, \qquad \epsilon = \text{regularisation constant}$$

Second order methods



Deep learning in practice

- Created by Google
- Math library used for machine learning and neural networks
- Supports Python and C++
- Pros
  - ✓ Documentation
  - ✓ Backed by large community
  - ✓ In built monitoring for training processes (Tensorboard)
- Cons
  - x Static computational graphs
  - x Higher learning curve than other libraries (low level, debugging harder)
  - x Some performance issues

https://www.tensorflow.org/

Deep learning in practice

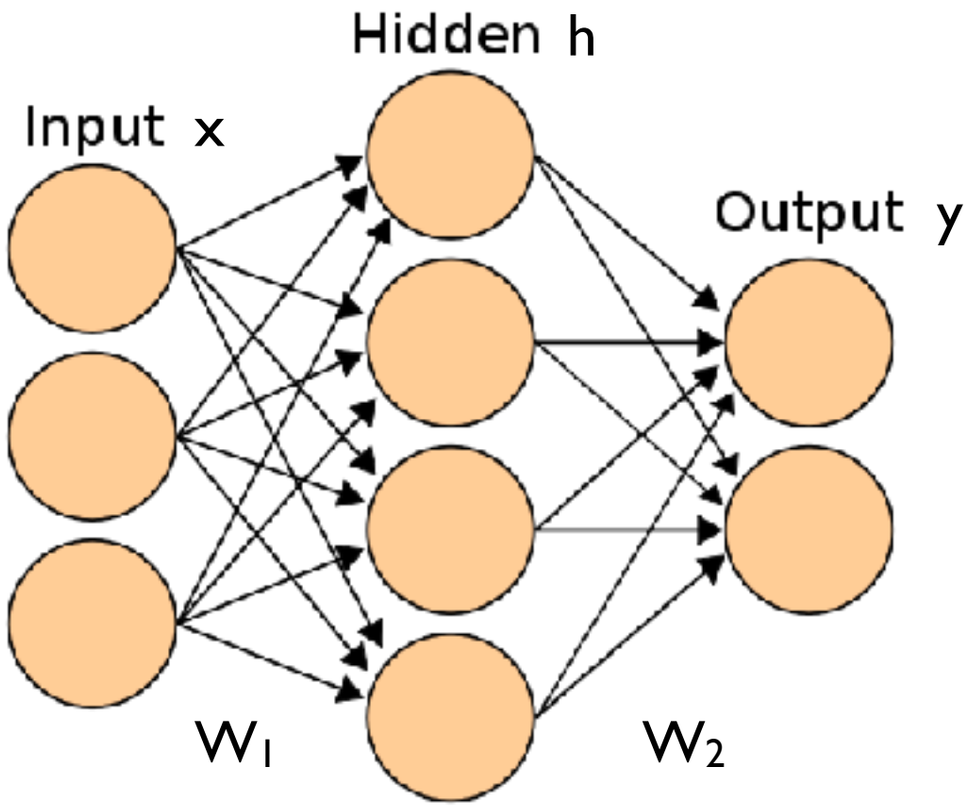
The University of Nottingham
UNITED KINGDOM · CHINA · MALAYSIA

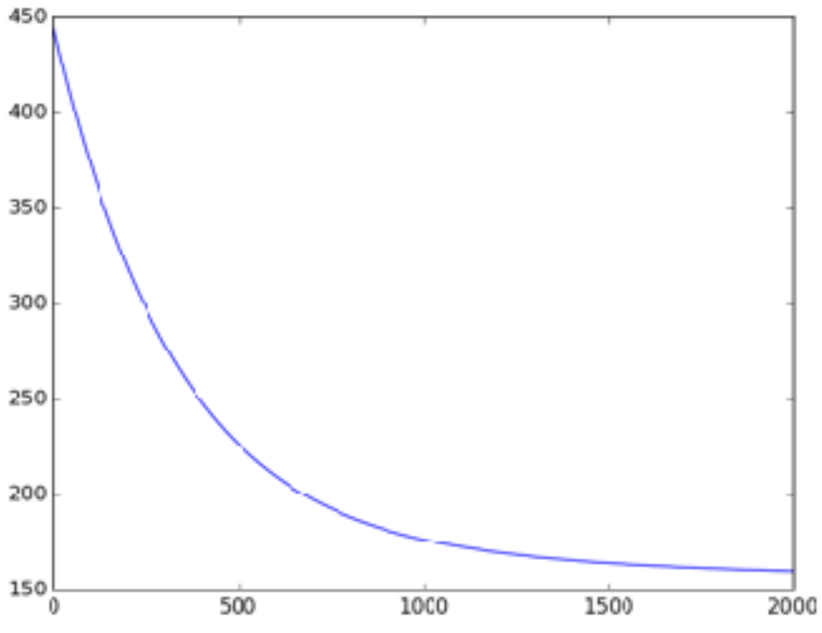
- Created by Facebook
- Tensor computation (like numpy) with GPU acceleration
- Supports Python
- Pros
  - ✓ Dynamic computational graphs (useful for e.g. RNNs)
  - ✓ Lower learning curve (more pythonic, easier to debug)
  - ✓ Easy to write own layer types
- Cons
  - x Lacks in built monitoring
  - x Not yet production ready (at v0.4 but less of an issue for research)
  - x Documentation not as detailed

https://pytorch.org/

PyTorch Example

The University of Nottingham
UNITED KINGDOM · CHINA · MALAYSIA





autograd_example.py

```python
import torch
import matplotlib.pyplot as plt

device = torch.device('cpu')

N, D_in, H, D_out = 64, 3, 4, 2

x = torch.randn(N, D_in, device=device)
y = torch.randn(N, D_out, device=device)

w1 = torch.randn(D_in, H, device=device, requires_grad=True)
w2 = torch.randn(H, D_out, device=device, requires_grad=True)

learning_rate = 1e-5
loss_data = []

for t in range(2000):
    h = x.mm(w1)
    phi = h.tanh()
    y_pred = phi.mm(w2)

    loss = (y_pred - y).pow(2).sum()
    loss_data.append(loss.item())

    loss.backward()

    with torch.no_grad():
        w1 -= learning_rate * w1.grad
        w2 -= learning_rate * w2.grad
        w1.grad.zero_()
        w2.grad.zero_()

plt.plot(loss_data)
plt.show()
```

Deep learning in practice

- Keras is built on top of TensorFlow/Theano
- Supports Python
- Pros
  - ✓ Easiest learning curve
  - ✓ Very intuitive interface for building neural networks
  - ✓ Easy to write own layer types
- Cons
  - x High level and not always as customisable
  - x Not as many functionalities, less control

https://keras.io/

Loss Function

The University of Nottingham
UNITED KINGDOM · CHINA · MALAYSIA

- The first thing to do to train a neural network is define a loss function
- For **continuous** outputs these include the mean squared error and mean absolute error

$$E(\mathbf{w}) = \frac{1}{N}\sum_i (y_i - \hat{y}_i(\mathbf{w}))^2 \qquad E(\mathbf{w}) = \frac{1}{N}\sum_i |y_i - \hat{y}_i(\mathbf{w})|$$

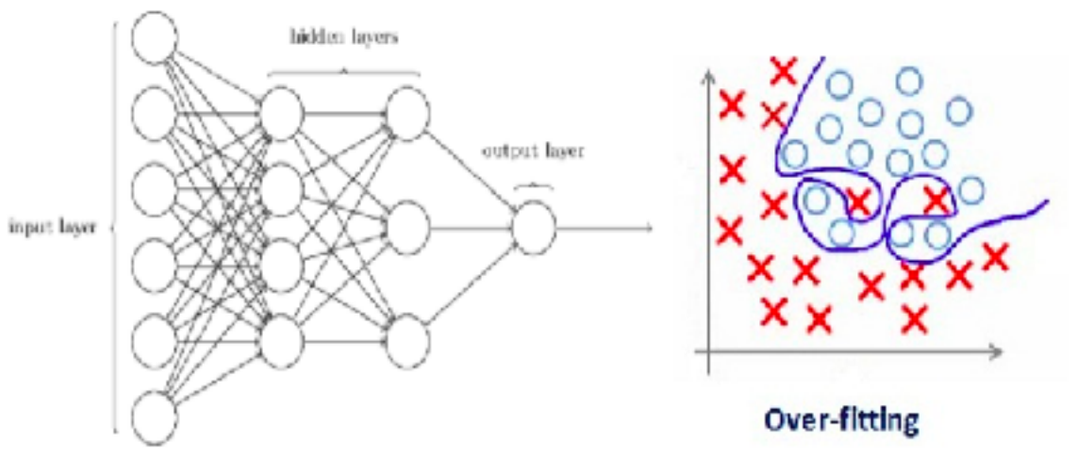
- Full loss function can include additional regularization terms
- For **categorical** outputs loss function is usually the categorical cross-entropy
- Last layer typically has a **soft-max** activation (turns M outputs into normalized probabilities)

$$E((\mathbf{w})) = -\sum_{i=1}^{n}\sum_{m=0}^{M-1} y_{im}\log\hat{y}_{im}(\mathbf{w}) + (1-y_{im})\log\left[1-\hat{y}_{im}(\mathbf{w})\right] \qquad y_{im} = \begin{cases} 1, & \text{if } y_i = m \\ 0, & \text{otherwise.} \end{cases}$$

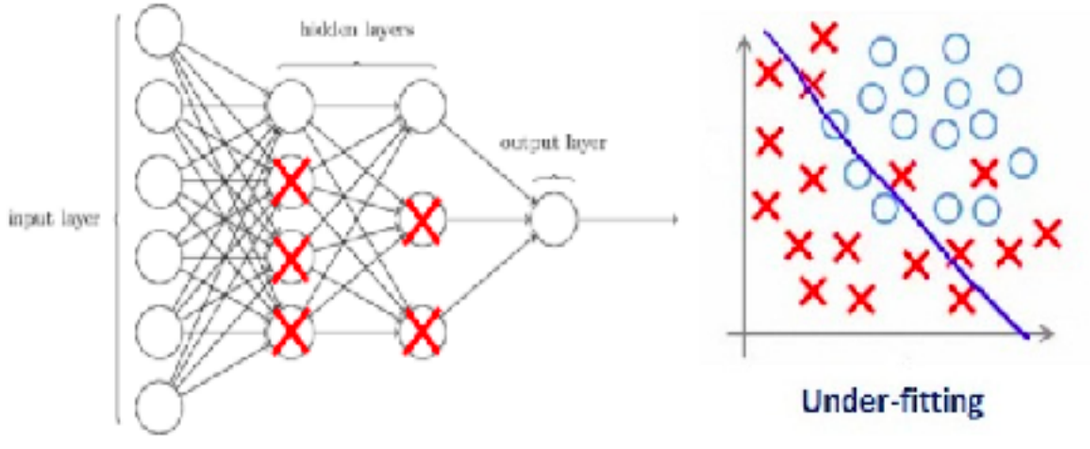
- Loss increased when predicted probability further from actual label

Regularization

- Regularization helps ensure neural networks do not **over-fit** and generalise well to unseen data
- L1 and L2 regularization apply penalties on a per layer basis during optimization
- **Dropout** turns off random neurons with probability p for each mini batch during training



Model has too many free parameters and data is over-fitted



Too many neurons either dropped out or regularisation so high that weight matrices close to zero - under-fitting!

Training

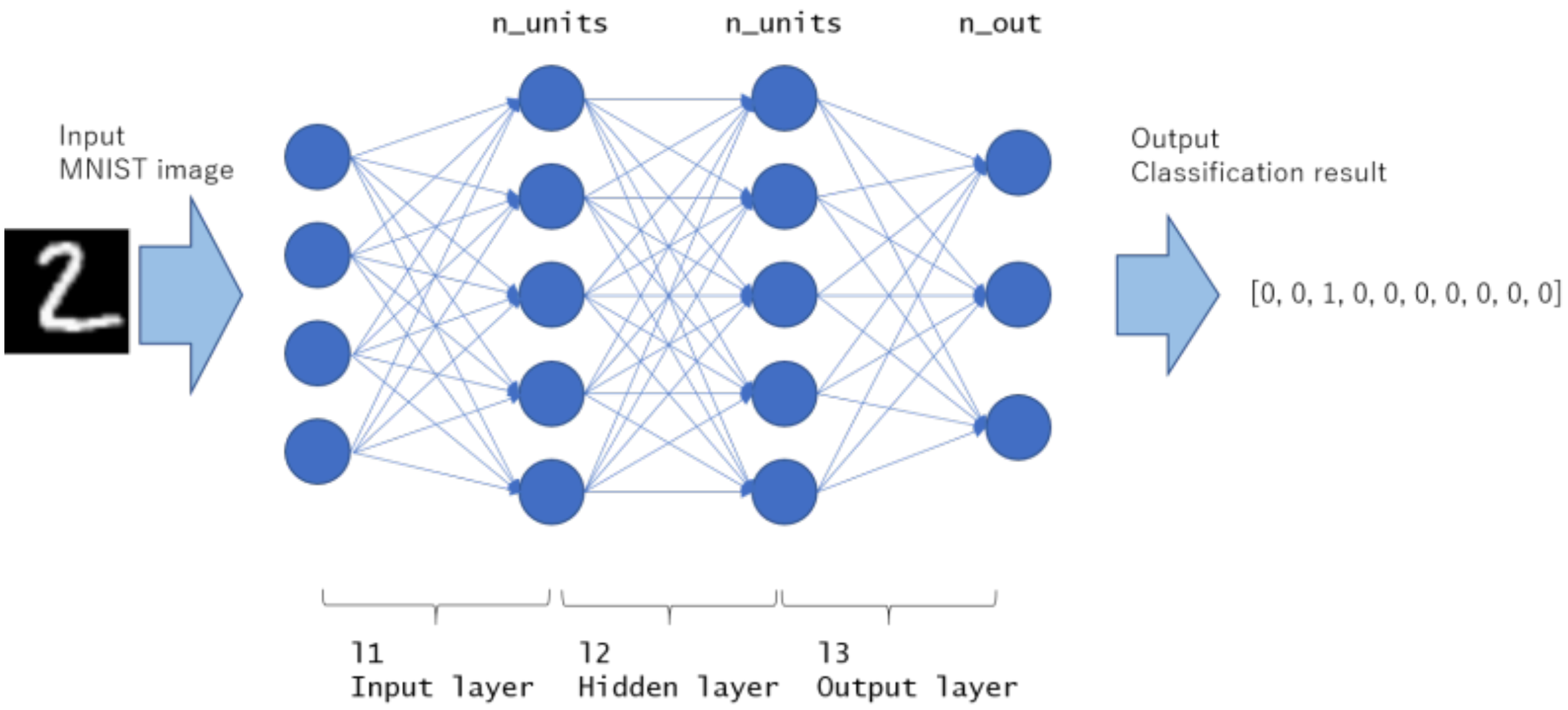
- Data should first be separated into training, test and holdout datasets
  - Training data is used to fit the parameters of the network
  - Test data to to evaluate the performance of the trained model on unseen data. It can be used to tune the various hyper parameters of the network (e.g. number of layers, hidden units etc)
  - Holdout data is used to assess the final performance of the tuned model
- During training test loss can be monitoring, and training should be stopped when this increases
- Early stopping can also be seen as a type of regularization and avoids over-fitting

Hello World

The University of Nottingham
UNITED KINGDOM · CHINA · MALAYSIA

- MNIST is `Hello Word' of deep learning
- Black and white images of integers from 0 to 9
- 28 x 28 pixel images
- 60,000 training images and 10,000 test images

MNIST MLP



MNIST MLP (Keras)

- Very simple API!
- 2 hidden layers each with 512 units
- Relu activation functions
- Dropout with p=0.2
- Final layer has softmax activation
- Gets to 98.40% test accuracy after 20 epochs
- State of the art has 99.8% accuracy

autograd_example.py | keras_mnist_mlp.py

```python
from __future__ import print_function

import keras
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, Dropout
from keras.optimizers import RMSprop

batch_size = 128
num_classes = 10
epochs = 20

# the data, split between train and test sets
(x_train, y_train), (x_test, y_test) = mnist.load_data()

x_train = x_train.reshape(60000, 784)
x_test = x_test.reshape(10000, 784)
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255
x_test /= 255
print(x_train.shape[0], 'train samples')
print(x_test.shape[0], 'test samples')

# convert class vectors to binary class matrices
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)

model = Sequential()
model.add(Dense(512, activation='relu', input_shape=(784,)))
model.add(Dropout(0.2))
model.add(Dense(512, activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(num_classes, activation='softmax'))

model.summary()

model.compile(loss='categorical_crossentropy',
              optimizer=RMSprop(),
              metrics=['accuracy'])

history = model.fit(x_train, y_train,
                    batch_size=batch_size,
                    epochs=epochs,
                    verbose=1,
                    validation_data=(x_test, y_test))
score = model.evaluate(x_test, y_test, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

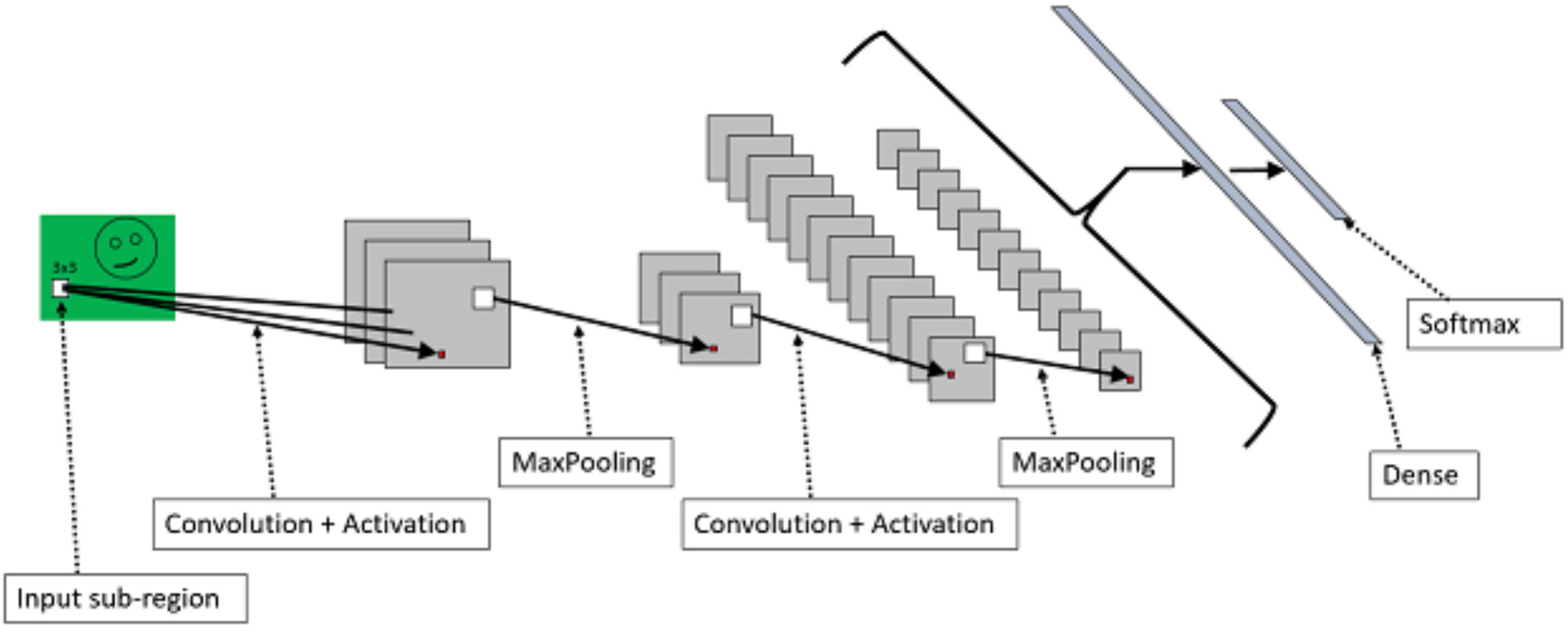
MNIST MLP (pyTorch)

The University of Nottingham
UNITED KINGDOM · CHINA · MALAYSIA

```python
from __future__ import print_function
import argparse
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torchvision import datasets, transforms


class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(28*28, 512)
        self.fc1_drop = nn.Dropout(0.2)
        self.fc2 = nn.Linear(512, 512)
        self.fc2_drop = nn.Dropout(0.2)
        self.fc3 = nn.Linear(512, 512)

    def forward(self, x):
        x = x.view(-1, 28*28)
        x = F.relu(self.fc1(x))
        x = self.fc1_drop(x)
        x = F.relu(self.fc2(x))
        x = self.fc2_drop(x)
        return F.log_softmax(self.fc3(x))


def train(args, model, device, train_loader, optimizer, epoch):
    model.train()
    for batch_idx, (data, target) in enumerate(train_loader):
        data, target = data.to(device), target.to(device)
        optimizer.zero_grad()
        output = model(data)
        loss = F.nll_loss(output, target)
        loss.backward()
        optimizer.step()
        if batch_idx % args.log_interval == 0:
            print('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}'.format(
                epoch, batch_idx * len(data), len(train_loader.dataset),
                100. * batch_idx / len(train_loader), loss.item()))


def test(args, model, device, test_loader):
    model.eval()
    test_loss = 0
    correct = 0
    with torch.no_grad():
        for data, target in test_loader:
            data, target = data.to(device), target.to(device)
            output = model(data)
            test_loss += F.nll_loss(output, target, reduction='sum').item()  # sum up batch loss
            pred = output.max(1, keepdim=True)[1]  # get the index of the max log-probability
            correct += pred.eq(target.view_as(pred)).sum().item()

    test_loss /= len(test_loader.dataset)
    print('\nTest set: Average loss: {:.4f}, Accuracy: {}/{} ({:.0f}%)\n'.format(
        test_loss, correct, len(test_loader.dataset),
        100. * correct / len(test_loader.dataset)))
```

```python
def main():
    # Training settings
    parser = argparse.ArgumentParser(description='PyTorch MNIST Example')
    parser.add_argument('--batch-size', type=int, default=128, metavar='N',
                        help='input batch size for training (default: 64)')
    parser.add_argument('--test-batch-size', type=int, default=1000, metavar='N',
                        help='input batch size for testing (default: 1000)')
    parser.add_argument('--epochs', type=int, default=20, metavar='N',
                        help='number of epochs to train (default: 10)')
    parser.add_argument('--lr', type=float, default=0.01, metavar='LR',
                        help='learning rate (default: 0.01)')
    parser.add_argument('--momentum', type=float, default=0.5, metavar='M',
                        help='SGD momentum (default: 0.5)')
    parser.add_argument('--no-cuda', action='store_true', default=False,
                        help='disables CUDA training')
    parser.add_argument('--seed', type=int, default=1, metavar='S',
                        help='random seed (default: 1)')
    parser.add_argument('--log-interval', type=int, default=10, metavar='N',
                        help='how many batches to wait before logging training status')
    args = parser.parse_args()
    use_cuda = not args.no_cuda and torch.cuda.is_available()

    torch.manual_seed(args.seed)

    device = torch.device("cuda" if use_cuda else "cpu")

    kwargs = {'num_workers': 1, 'pin_memory': True} if use_cuda else {}
    train_loader = torch.utils.data.DataLoader(
        datasets.MNIST('../data', train=True, download=True,
                       transform=transforms.Compose([
                           transforms.ToTensor(),
                           transforms.Normalize((0.1307,), (0.3081,))
                       ])),
        batch_size=args.batch_size, shuffle=True, **kwargs)
    test_loader = torch.utils.data.DataLoader(
        datasets.MNIST('../data', train=False, transform=transforms.Compose([
                           transforms.ToTensor(),
                           transforms.Normalize((0.1307,), (0.3081,))
                       ])),
        batch_size=args.test_batch_size, shuffle=True, **kwargs)


    model = Net().to(device)
    optimizer = optim.SGD(model.parameters(), lr=args.lr, momentum=args.momentum)

    for epoch in range(1, args.epochs + 1):
        train(args, model, device, train_loader, optimizer, epoch)
        test(args, model, device, test_loader)


if __name__ == '__main__':
    main()
```

MNIST CNN



MNIST CNN

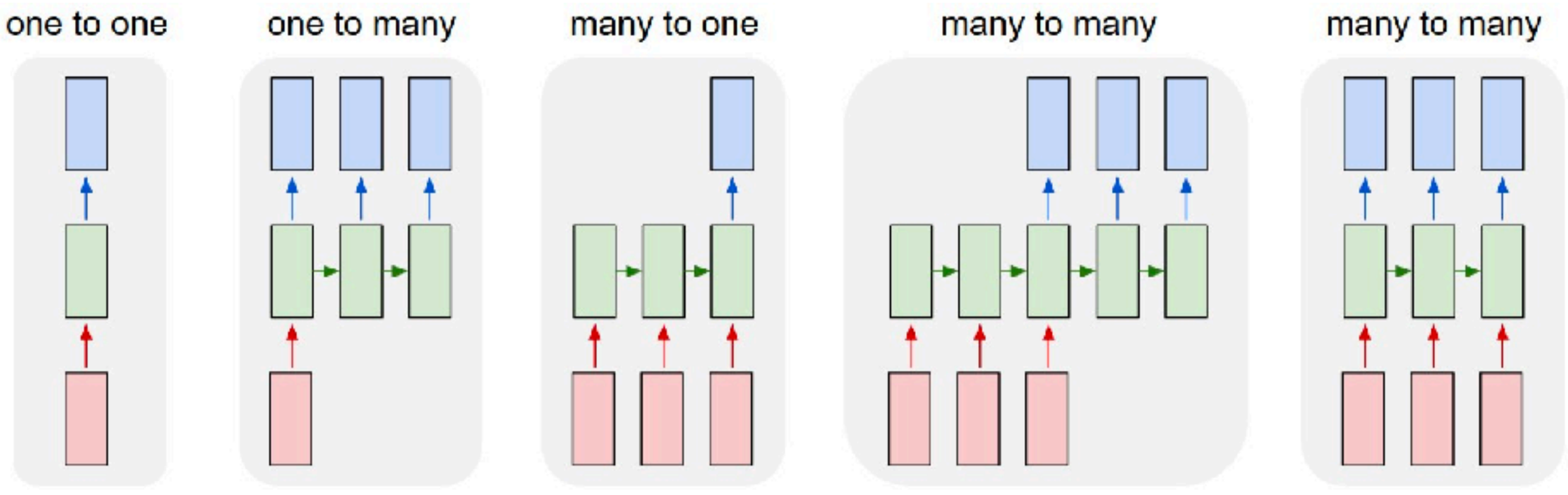
The University of Nottingham
UNITED KINGDOM · CHINA · MALAYSIA

- First uses 32 convolutional units with 3x3 kernel
- Next 64 convolutional units with 3x3 kernel
- Max pooling down-samples units
- Dropout
- Dense fully connected layer
- Additional dropout and softmax activation
- Gets to 99.25% test accuracy after 12 epochs
- Each epoch takes ~16 seconds on K520 GPU

```
batch_size = 128
num_classes = 10
epochs = 12

# input image dimensions
img_rows, img_cols = 28, 28

# the data, split between train and test sets
(x_train, y_train), (x_test, y_test) = mnist.load_data()

if K.image_data_format() == 'channels_first':
    x_train = x_train.reshape(x_train.shape[0], 1, img_rows, img_cols)
    x_test = x_test.reshape(x_test.shape[0], 1, img_rows, img_cols)
    input_shape = (1, img_rows, img_cols)
else:
    x_train = x_train.reshape(x_train.shape[0], img_rows, img_cols, 1)
    x_test = x_test.reshape(x_test.shape[0], img_rows, img_cols, 1)
    input_shape = (img_rows, img_cols, 1)

x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255
x_test /= 255
print('x_train shape:', x_train.shape)
print(x_train.shape[0], 'train samples')
print(x_test.shape[0], 'test samples')

# convert class vectors to binary class matrices
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)

model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3),
                 activation='relu',
                 input_shape=input_shape))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax'))

model.compile(loss=keras.losses.categorical_crossentropy,
              optimizer=keras.optimizers.Adadelta(),
              metrics=['accuracy'])

model.fit(x_train, y_train,
          batch_size=batch_size,
          epochs=epochs,
          verbose=1,
          validation_data=(x_test, y_test))
score = model.evaluate(x_test, y_test, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

Recurrent Network

- Recurrent neural networks (RNNs) are a class of neural network that can learn about sequential data (e.g. time series, natural language)



E.g. image classification

E.g. image caption

E.g. sentiment analysis

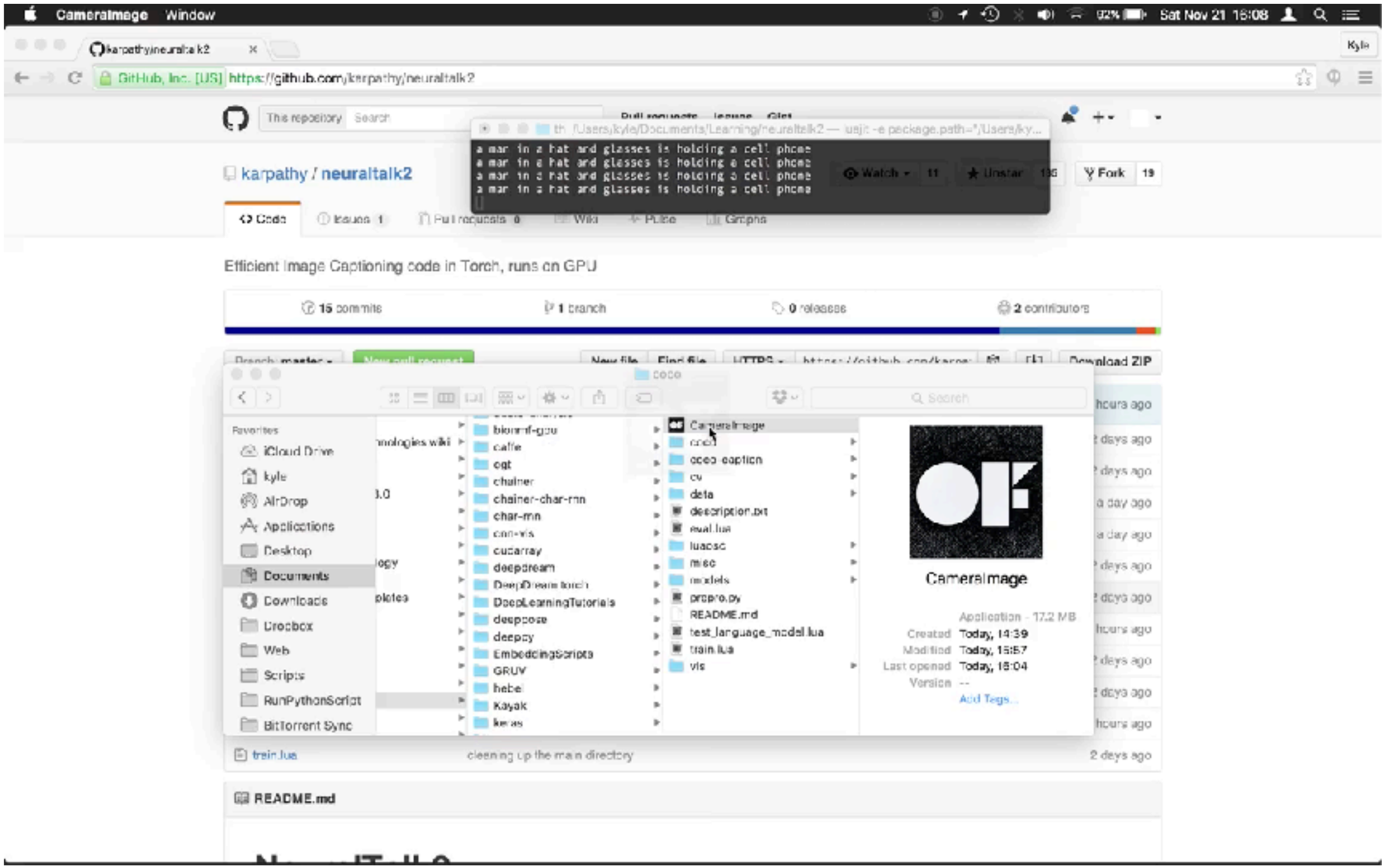
E.g. language translation

E.g. predict next word in sentence

http://karpathy.github.io/2015/05/21/rnn-effectiveness/

Recurrent Network

The University of Nottingham
UNITED KINGDOM • CHINA • MALAYSIA



Recurrent Network

Proof. Omitted. □

Lemma 0.1. *Let $\mathcal{C}$ be a set of the construction.*

Let $\mathcal{C}$ be a gerber covering. Let $\mathcal{F}$ be a quasi-coherent sheaves of $\mathcal{O}$-modules. We have to show that

$$\mathcal{O}_{\mathcal{O}_X} = \mathcal{O}_X(\mathcal{L})$$

.

Proof. This is an algebraic space with the composition of sheaves $\mathcal{F}$ on $X_{\acute{e}tale}$ we have

$$\mathcal{O}_X(\mathcal{F}) = \{morph_1 \times_{\mathcal{O}_X} (\mathcal{G}, \mathcal{F})\}$$

where $\mathcal{G}$ defines an isomorphism $\mathcal{F} \to \mathcal{F}$ of $\mathcal{O}$-modules. □

Lemma 0.2. *This is an integer $\mathcal{Z}$ is injective.*

Proof. See Spaces, Lemma ??. □

Lemma 0.3. *Let S be a scheme. Let X be a scheme and X is an affine open covering. Let $\mathcal{U} \subset \mathcal{X}$ be a canonical and locally of finite type. Let X be a scheme. Let X be a scheme which is equal to the formal complex.*

The following to the construction of the lemma follows.

Let X be a scheme. Let X be a scheme covering. Let

$$b : X \to Y' \to Y \to Y \to Y' \times_X Y \to X.$$

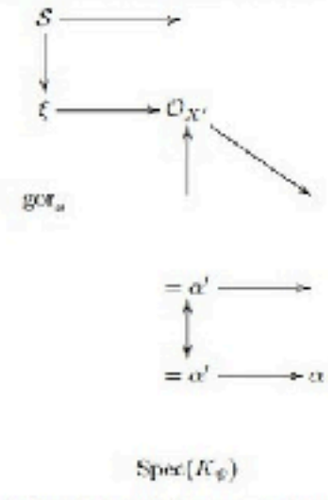
be a morphism of algebraic spaces over S and Y.

Proof. Let X be a nonzero scheme of X. Let X be an algebraic space. Let $\mathcal{F}$ be a quasi-coherent sheaf of $\mathcal{O}_X$-modules. The following are equivalent

(1) $\mathcal{F}$ is an algebraic space over S.
(2) If X is an affine open covering.

Consider a common structure on X and X the functor $\mathcal{O}_X(U)$ which is locally of finite type. □

This since $\mathcal{F} \in \mathcal{F}$ and $x \in \mathcal{G}$ the diagram



is a limit. Then $\mathcal{G}$ is a finite type and assume S is a flat and $\mathcal{F}$ and $\mathcal{G}$ is a finite type f_*. This is of finite type diagrams, and

- the composition of $\mathcal{G}$ is a regular sequence,
- $\mathcal{O}_{X'}$ is a sheaf of rings.

□

Proof. We have see that $X = \mathrm{Spec}(R)$ and $\mathcal{F}$ is a finite type representable by algebraic space. The property $\mathcal{F}$ is a finite morphism of algebraic stacks. Then the cohomology of X is an open neighbourhood of U. □

Proof. This is clear that $\mathcal{G}$ is a finite presentation, see Lemmas ??.
A *reduced above* we conclude that U is an open covering of $\mathcal{C}$. The functor $\mathcal{F}$ is a "Field

$$\mathcal{O}_{X,x} \longrightarrow \mathcal{F}_{\overline{x}} \quad \text{-1}(\mathcal{O}_{X_{\text{étale}}}) \longrightarrow \mathcal{O}_{X'_i}^{-1}\mathcal{O}_{X'_i}(\mathcal{O}_{X'_i}^{\vee})$$

is an isomorphism of covering of $\mathcal{O}_{X_i}$. If $\mathcal{F}$ is the unique element of $\mathcal{F}$ such that X is an isomorphism.
The property $\mathcal{F}$ is a disjoint union of Proposition ?? and we can filtered set of presentations of a scheme $\mathcal{O}_X$-algebra with $\mathcal{F}$ are opens of finite type over S.
If $\mathcal{F}$ is a scheme theoretic image points. □

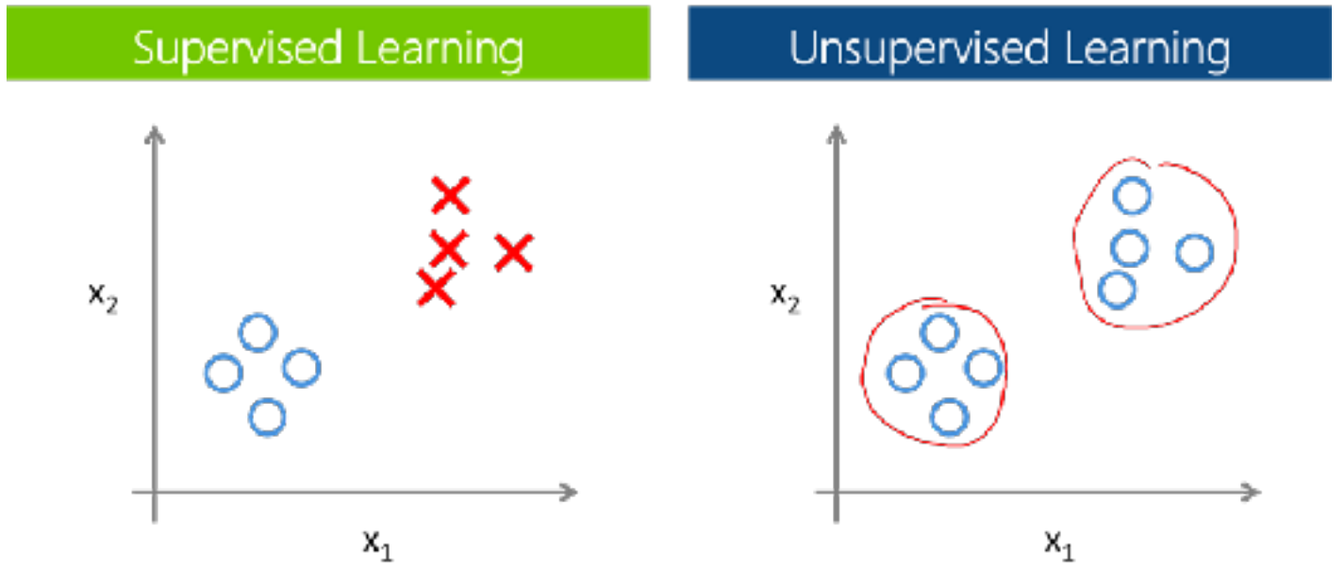
If $\mathcal{F}$ is a finite direct sum $\mathcal{O}_{X_\lambda}$ is a closed immersion, see Lemma ??. This is a sequence of $\mathcal{F}$ is a similar morphism.

http://karpathy.github.io/2015/05/21/rnn-effectiveness/

Unsupervised Learning

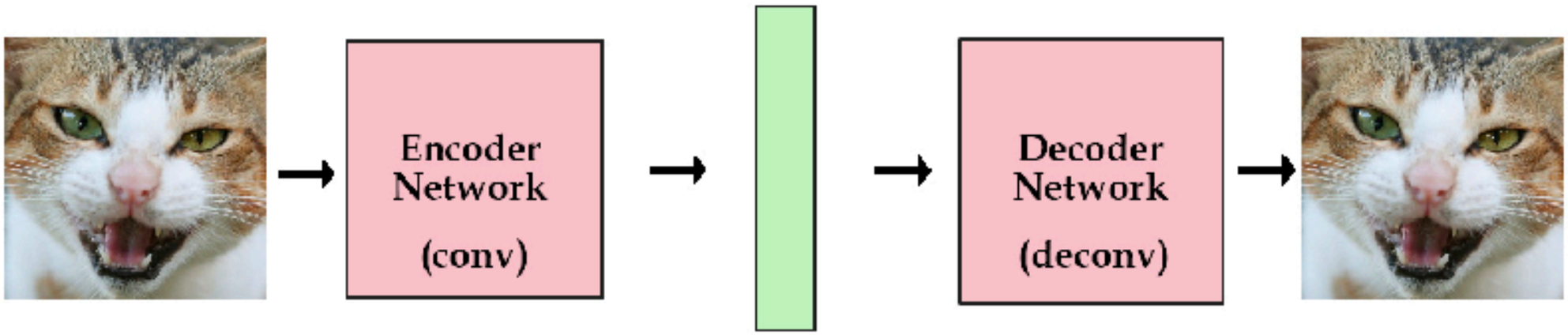
The University of Nottingham
UNITED KINGDOM · CHINA · MALAYSIA

- **Much** more unlabelled than labelled data
- Labelling can be subjective
- Unsupervised learning finds the most important features in the dataset



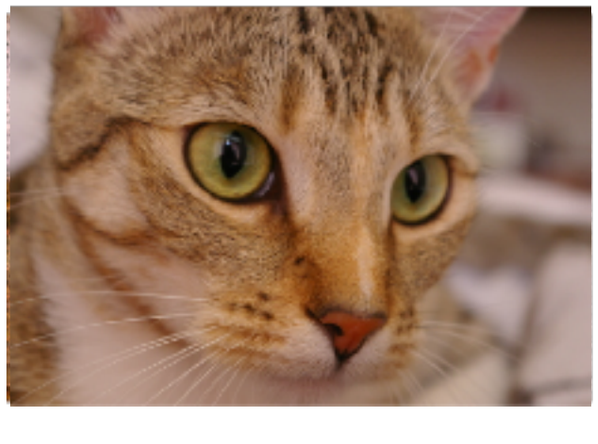
- Examples of unsupervised learning
  - Auto-encoders and variational auto-encoders (generative)
  - Adversarial networks (generative)
  - Boltzmann machines (generative)
  - Reinforcement learning (self-supervised)

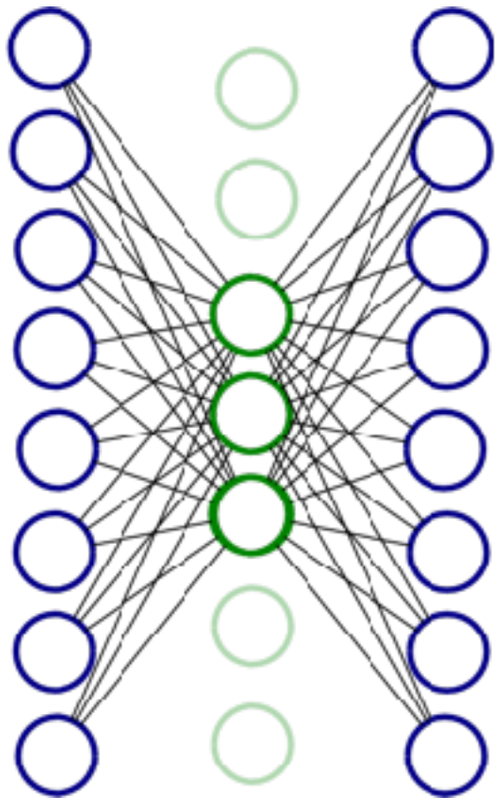
Auto-encoders



- Data compression algorithm
  - Data specific (will only be able to compress data similar to what they are trained on)
  - Lossy (decompressed output degraded compared to input)
  - No labelling required

Auto-encoders





- Latent space should have fewer dimensions
  - Under-complete representation
  - Bottleneck architecture
- Otherwise over-complete representation might just learn the identity function

MNIST

The University of Nottingham
UNITED KINGDOM · CHINA · MALAYSIA

- Used Keras Python library
- Discard labels
- Normalised values between 0 and 1 and reshape into 784 vector as before
- Encoding dimension 32, corresponds to compression factor of 24.5
- Can visualise reconstructed inputs from the test set

```
from keras.layers import Input, Dense
from keras.models import Model
from keras.datasets import mnist
import numpy as np

(x_train, _), (x_test, _) = mnist.load_data()

x_train = x_train.astype('float32') / 255.
x_test = x_test.astype('float32') / 255.
x_train = x_train.reshape((len(x_train), np.prod(x_train.shape[1:])))
x_test = x_test.reshape((len(x_test), np.prod(x_test.shape[1:])))
print x_train.shape
print x_test.shape

encoding_dim = 32

input_img = Input(shape=(784,))
encoded = Dense(encoding_dim, activation='relu')(input_img)
decoded = Dense(784, activation='sigmoid')(encoded)

# this model maps an input to its reconstruction
autoencoder = Model(input_img, decoded)

autoencoder.compile(optimizer='adadelta', loss='binary_crossentropy')

autoencoder.fit(x_train, x_train,
                epochs=50,
                batch_size=256,
                shuffle=True,
                validation_data=(x_test, x_test))

decoded_imgs = autoencoder.predict(x_test)
```

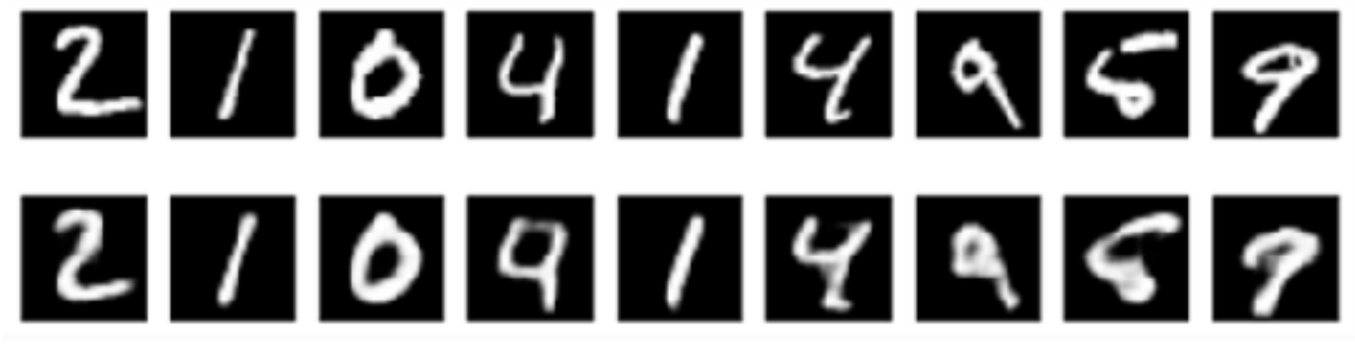
- Losing detail with this approach - how do we improve it?

CNN Auto-encoder

The University of Nottingham
UNITED KINGDOM · CHINA · MALAYSIA

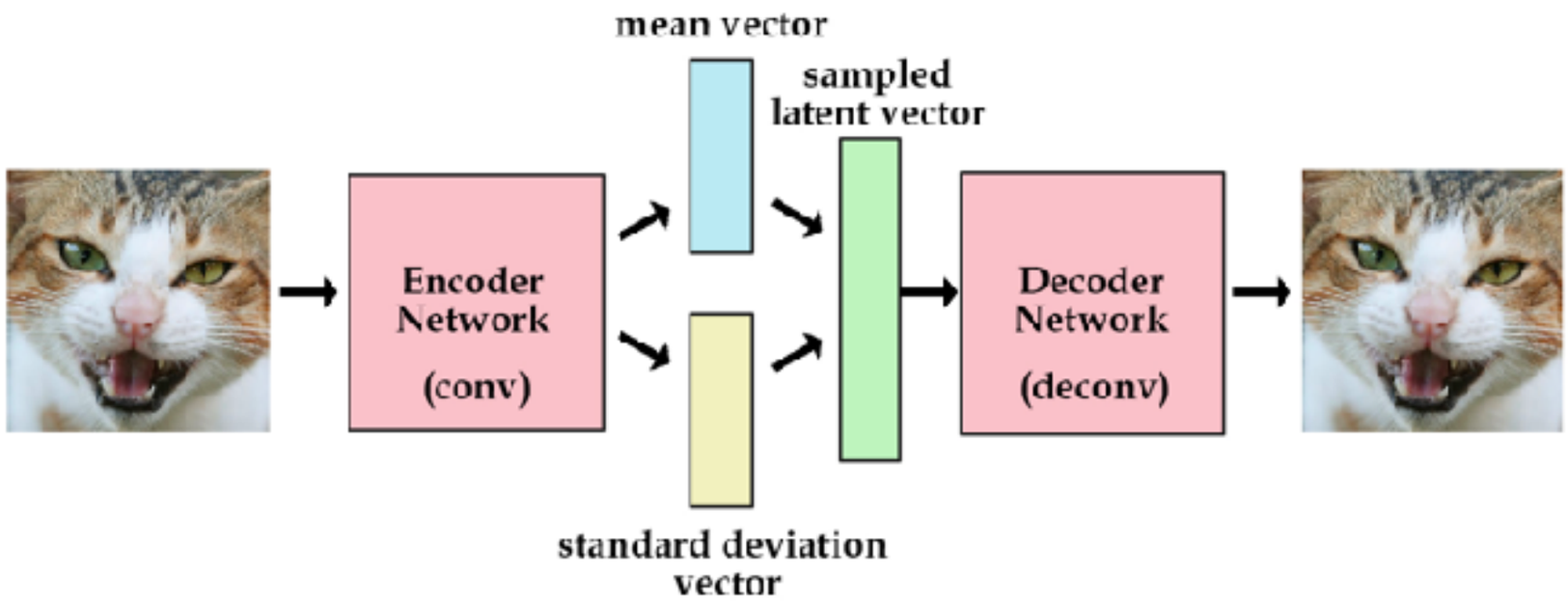
- Since inputs and outputs are images it makes sense to use convolutional network as encoder and decoder
- We also do not have to limit ourselves to a single latent layer
- Both encoder and decoder are stacked CNNS
- Loss function significantly improved (0.094 vs 0.11)

```
from keras.layers import Input, Dense
from keras.models import Model
from keras.layers import Conv2D, MaxPooling2D, UpSampling2D
from keras import backend as K
from keras.datasets import mnist
import numpy as np

(x_train, _), (x_test, _) = mnist.load_data()

x_train = x_train.astype('float32') / 255.
x_test = x_test.astype('float32') / 255.
x_train = np.reshape(x_train, (len(x_train), 28, 28, 1))
x_test = np.reshape(x_test, (len(x_test), 28, 28, 1))

input_img = Input(shape=(28, 28, 1))

x = Conv2D(16, (3, 3), activation='relu', padding='same')(input_img)
x = MaxPooling2D((2, 2), padding='same')(x)
x = Conv2D(8, (3, 3), activation='relu', padding='same')(x)
x = MaxPooling2D((2, 2), padding='same')(x)
x = Conv2D(8, (3, 3), activation='relu', padding='same')(x)
encoded = MaxPooling2D((2, 2), padding='same')(x)

# at this point the representation is (4, 4, 8) i.e. 128-dimensional

x = Conv2D(8, (3, 3), activation='relu', padding='same')(encoded)
x = UpSampling2D((2, 2))(x)
x = Conv2D(8, (3, 3), activation='relu', padding='same')(x)
x = UpSampling2D((2, 2))(x)
x = Conv2D(16, (3, 3), activation='relu')(x)
x = UpSampling2D((2, 2))(x)
decoded = Conv2D(1, (3, 3), activation='sigmoid', padding='same')(x)

autoencoder = Model(input_img, decoded)
autoencoder.compile(optimizer='adadelta', loss='binary_crossentropy')

autoencoder.fit(x_train, x_train,
                epochs=50,
                batch_size=128,
                shuffle=True,
                validation_data=(x_test, x_test))
```



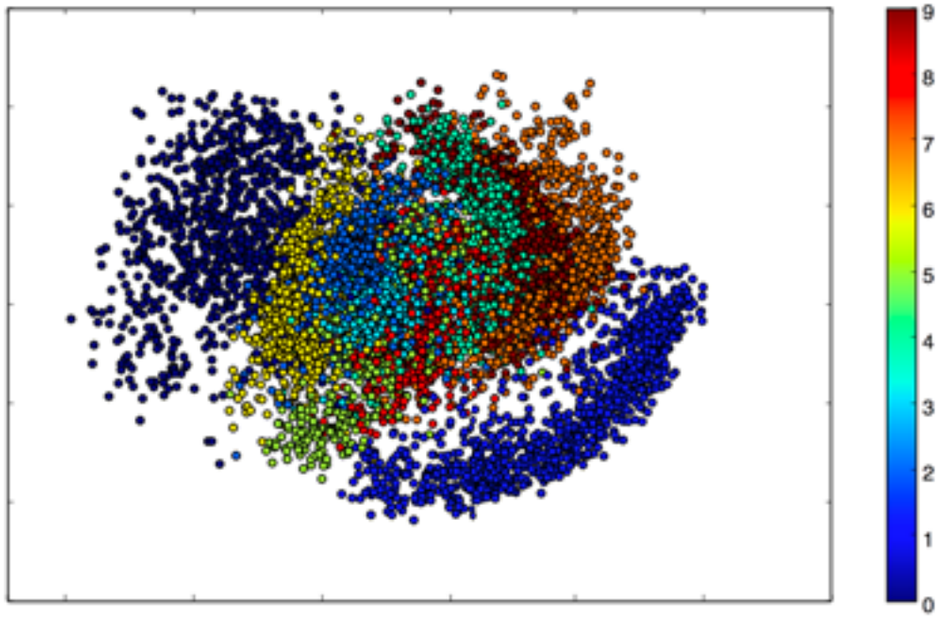
Variational AE

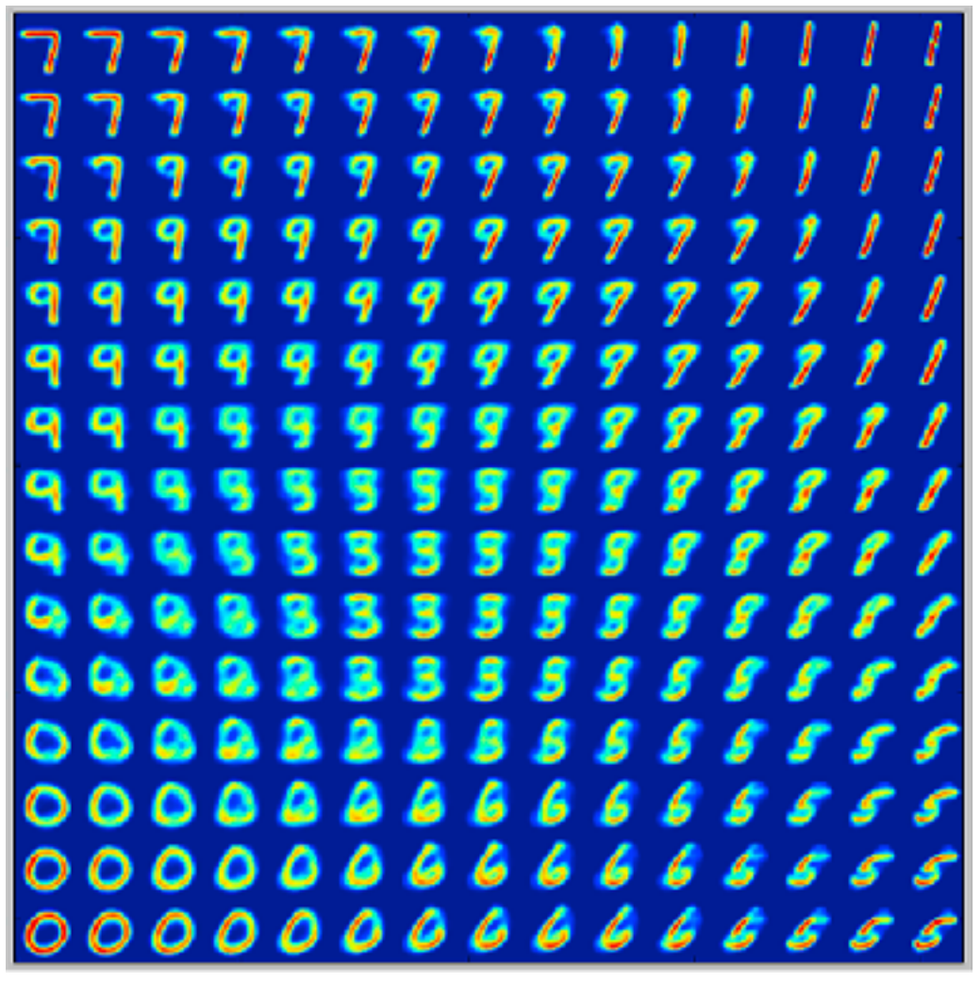
- An auto-encoder represents data in latent space
- Can’t generate anything yet, since we do not know how to create latent vectors
- Variational AE adds constraints on latent space, i.e. it learns a latent model such as a Gaussian probability density
- If you then sample points from this probability density, you can generate new samples



Variational AE

- Since latent space is 2D can visualise it (each coloured cluster is digit)





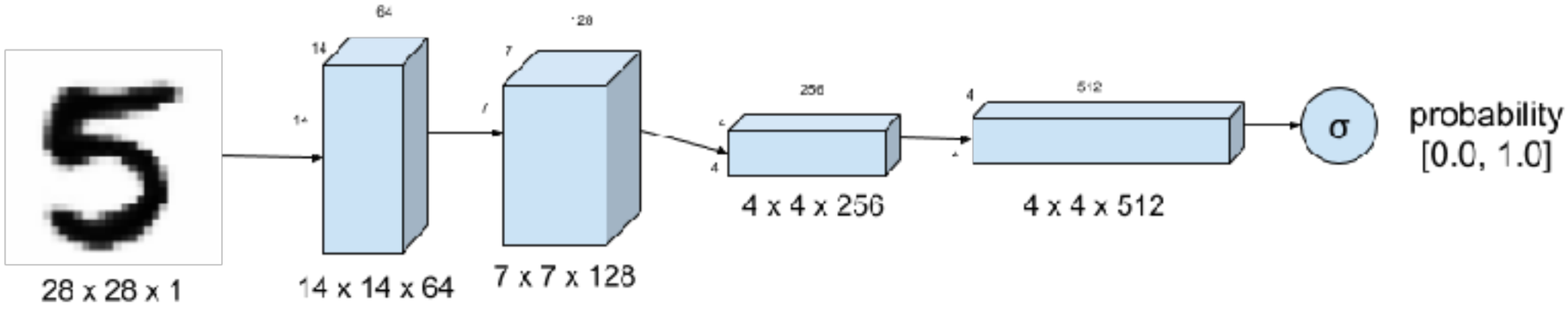
- Generated digits

Adversarial Networks

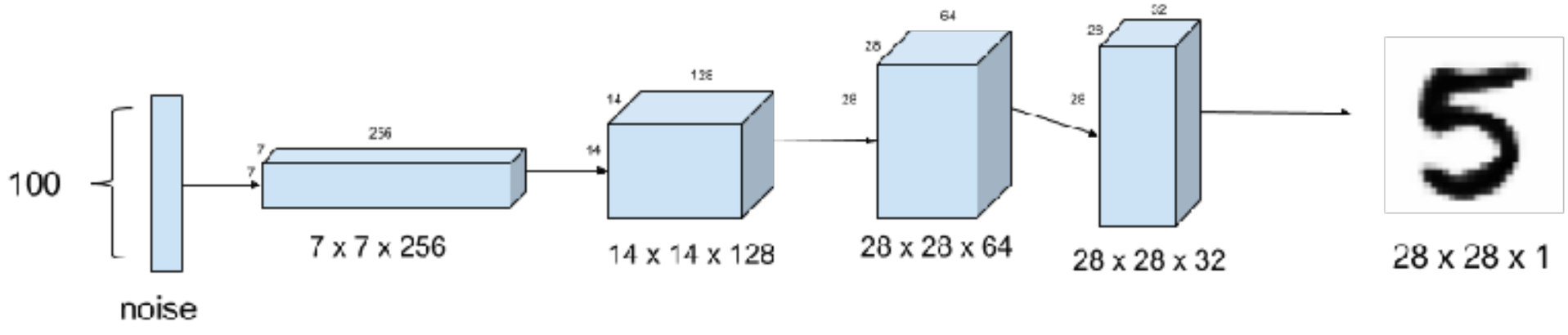
- VAEs are easy to train and have a tractable likelihood, but can produce images with lower fidelity
- GANs (Generative Adversarial Network) produce richer images but are much more difficult to train
- A GAN consists of 2 networks - a generator and discriminator, that compete and co-operate
- Real word example - counterfeiter (generator) and police (discriminator)
- Counterfeiter produces fakes, police give feedback why money is fake
- Counterfeiter improves fakes to trick police based on feedback
- At same time police are improving how to better tell apart real and fake money

GAN

- Discriminator tells if something is generative or real, e.g money, MNIST digits. Example is CNN

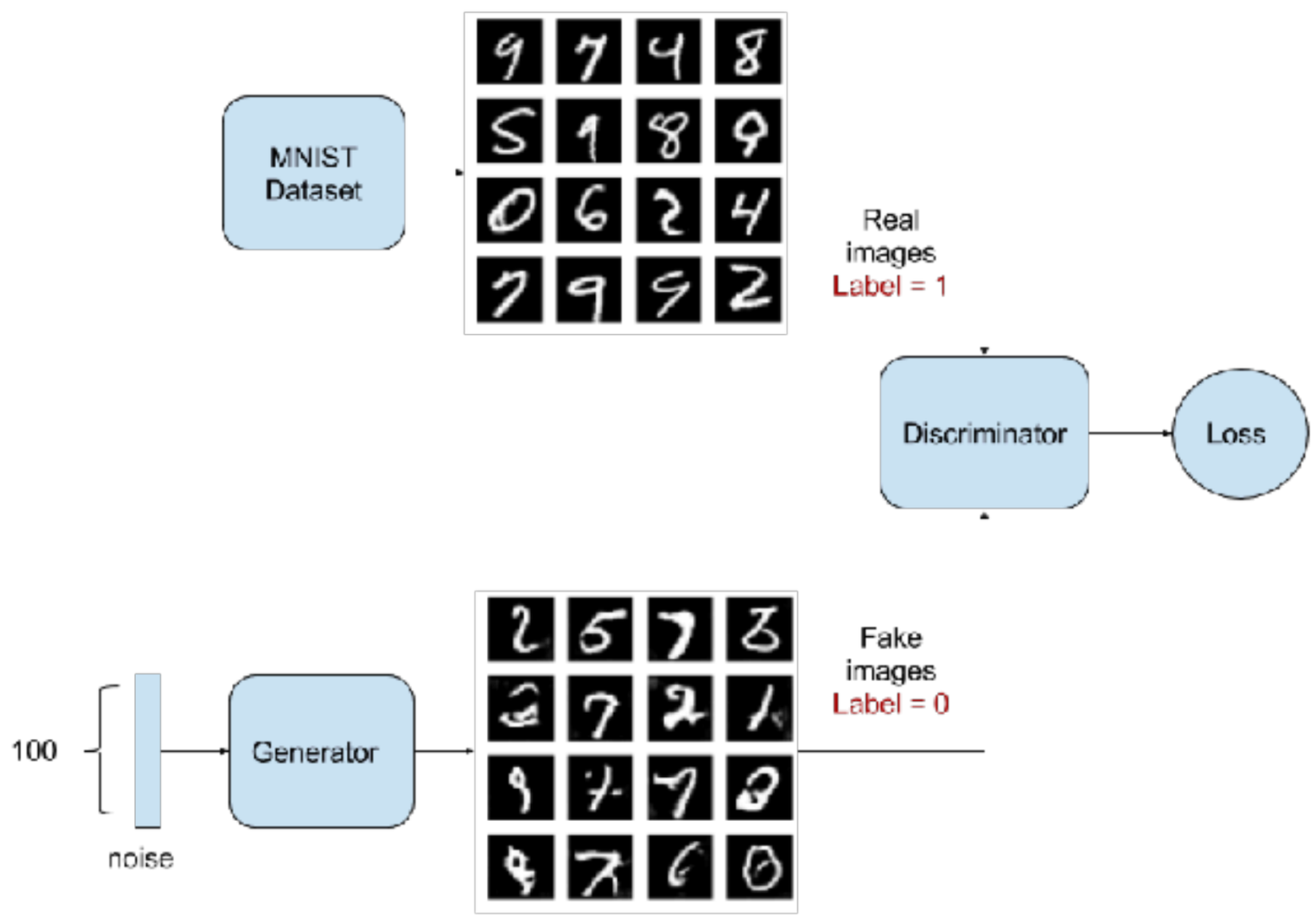


- Generator synthesises fakes. Input is generated from noise

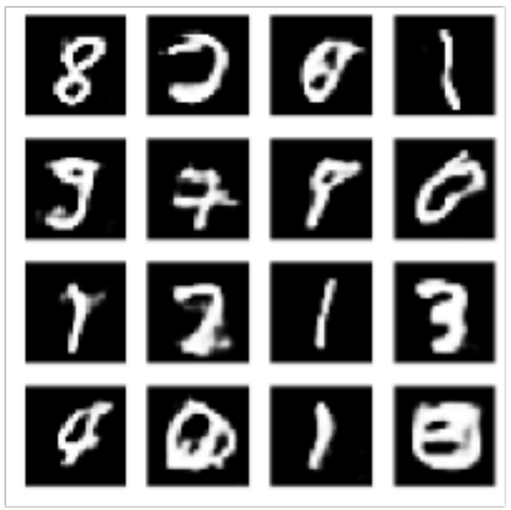


GAN

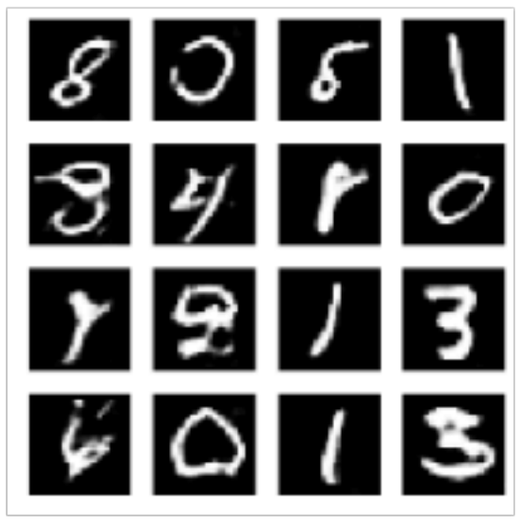
- Full discriminator model is generator + discriminator with both real and fake data correctly labelled



Samples

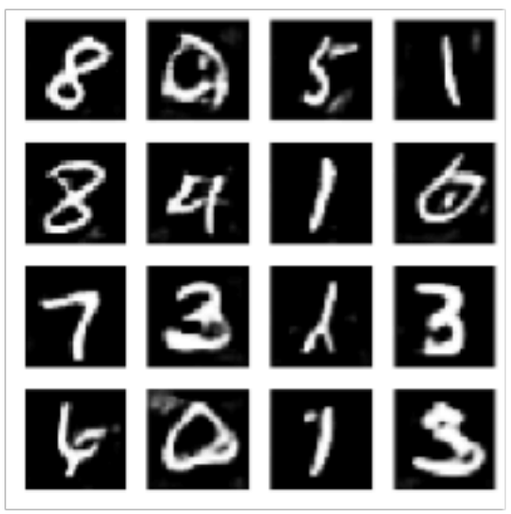


1000 steps

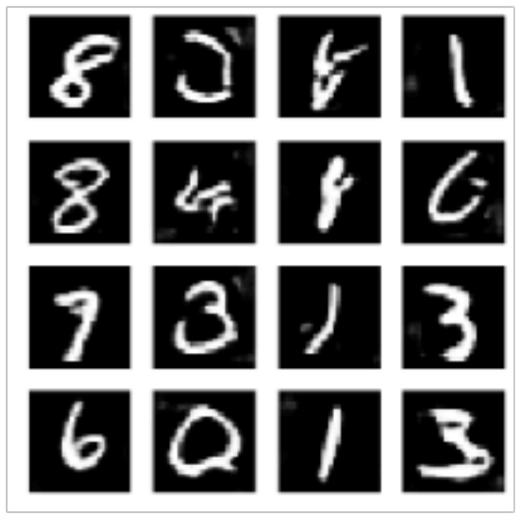


3000 steps

5000 steps



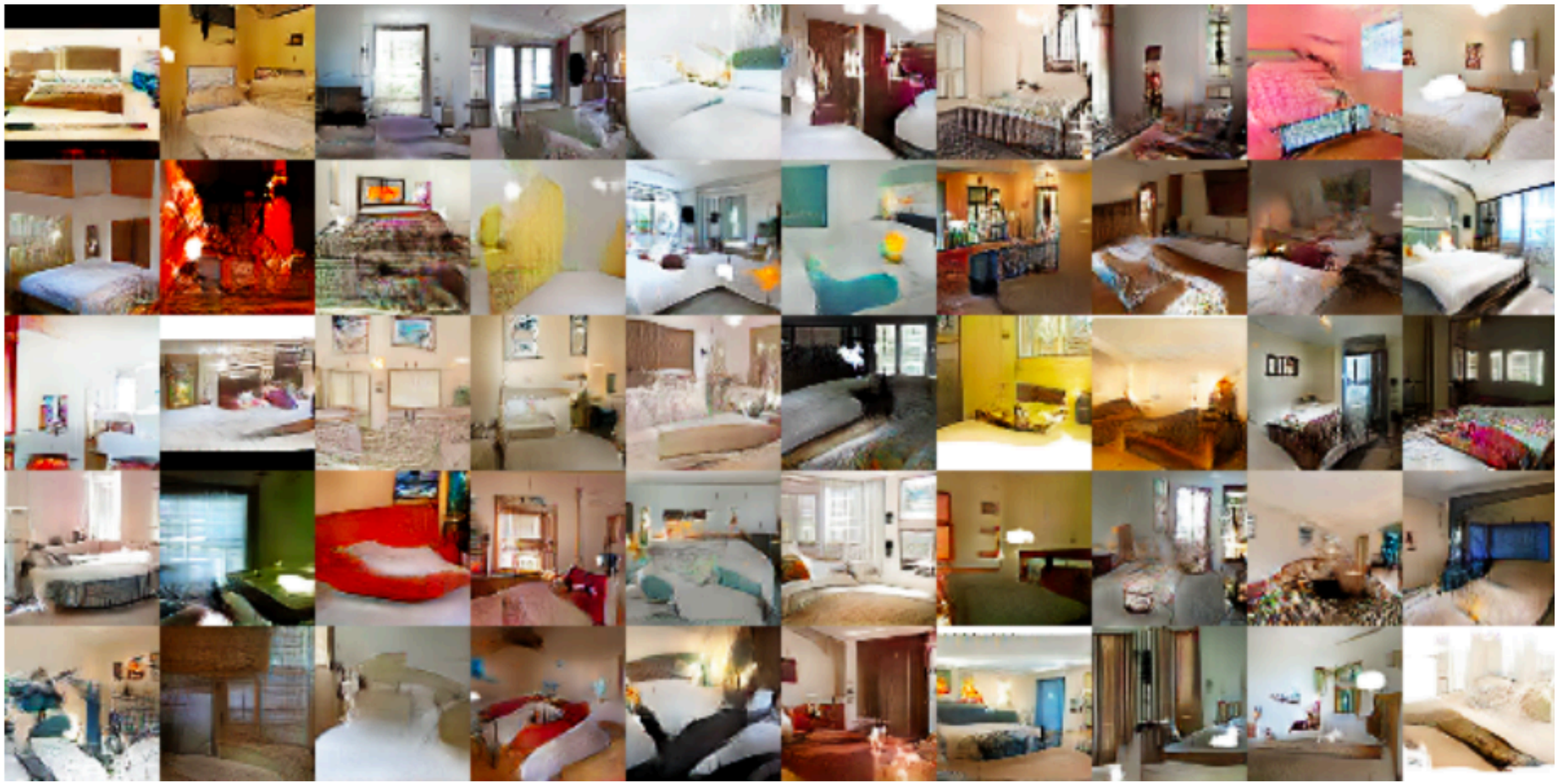
7000 steps



9000 steps

10000 steps

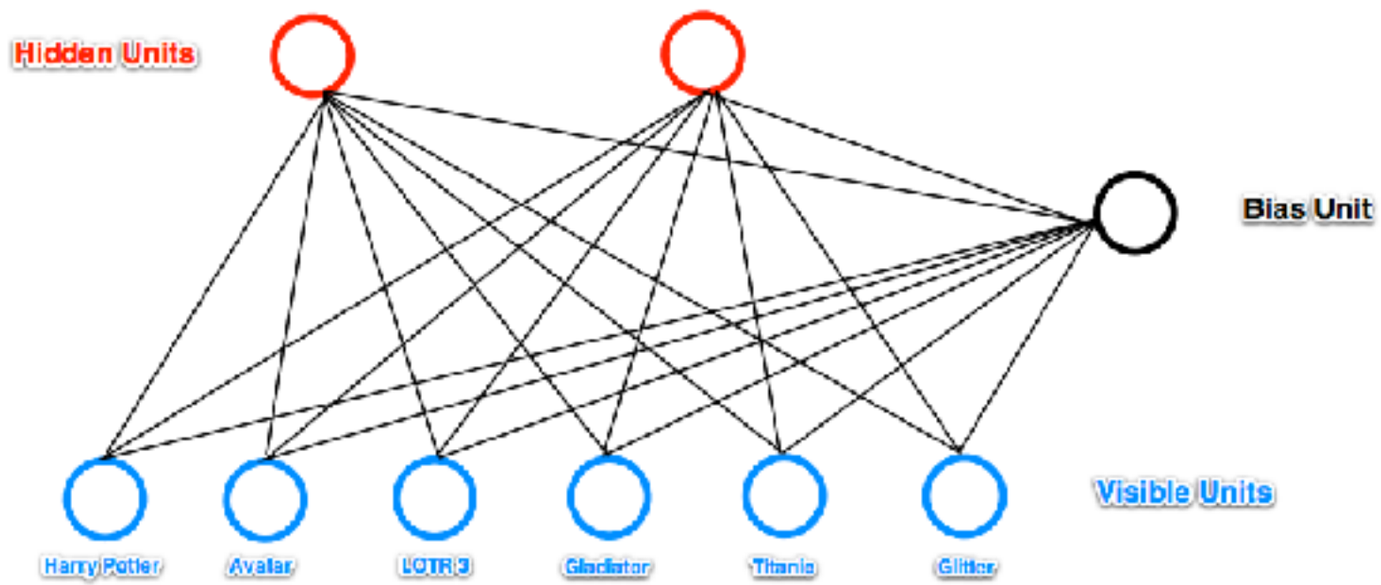
Other Examples



Boltzmann Machines

The University of Nottingham
UNITED KINGDOM · CHINA · MALAYSIA

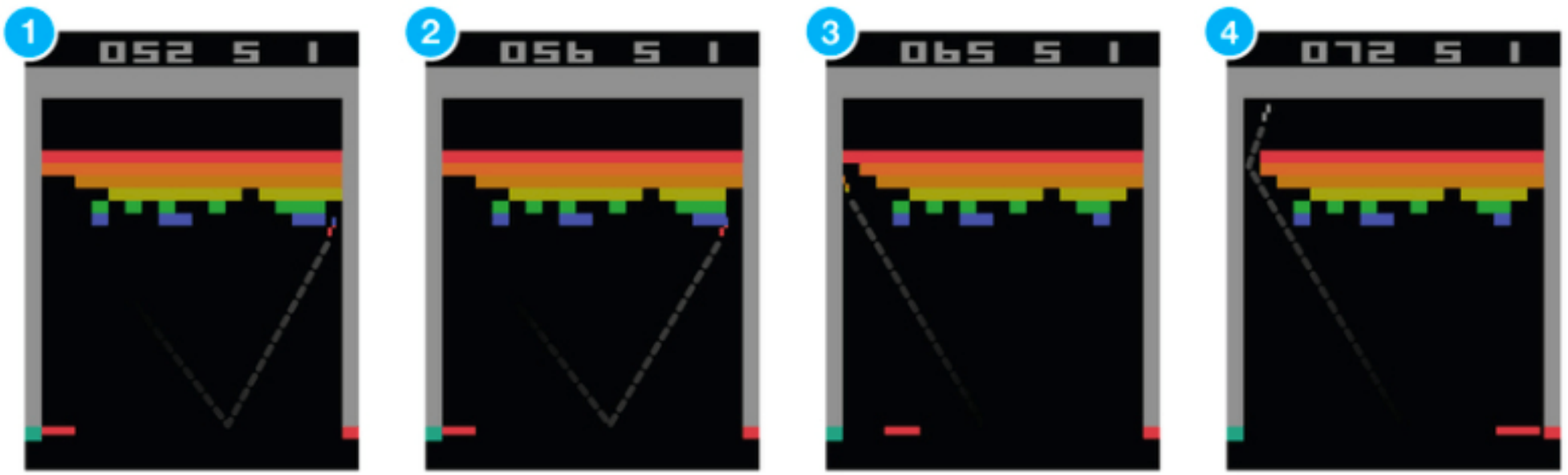
- Another type of generative network
- Important difference is that it is **stochastic**, i.e. activations in network have a probabilistic element
- Restricted Boltzmann Machine (RBM) consists of one visible layer and one hidden layer (contained latent factors)
- E.g. we have 6 movies and ask people which ones they want to watch. We will learn 2 latent variables



Reinforcement Learning

The University of Nottingham
UNITED KINGDOM · CHINA · MALAYSIA

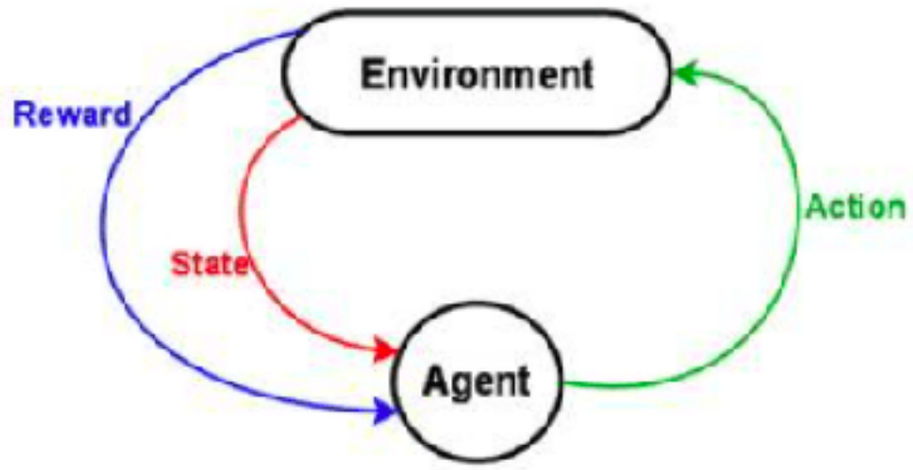
- Self-supervised learning, reward based feedback
- E.g. breakout game. Goal is to get ball though the blocks
- Can teach neural network how to play by giving the image of the screen as an input to the network
- Network will then decide which action to take (move paddle left or right) to maximise future possible reward

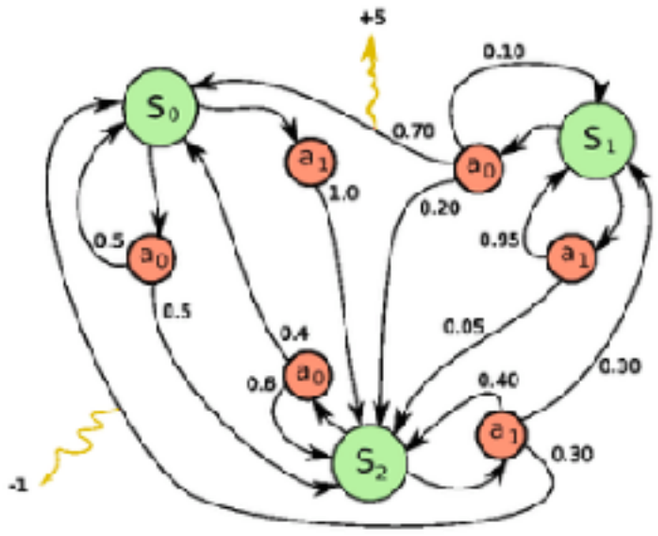


Reinforcement Learning

The University of Nottingham
UNITED KINGDOM · CHINA · MALAYSIA

- In general have an agent in an environment
- Environment is in certain state
- Agent can perform actions. These sometimes result in a reward
- Actions transform environment and lead to new state
- Sequence of states, actions, rewards $s_0, a_0, r_0, s_1, a_1, r_1, \ldots$
- A full sequence before terminal state is called an episode
- Rules for how to choose actions is called a policy





Q Learning

- Total discounted future reward in episode from time t onwards

$$R_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \dots$$

- Define Q function as maximum possible discounted future reward when we perform an action a in state s at time t

$$Q(s_t, a_t) = \max R_t$$

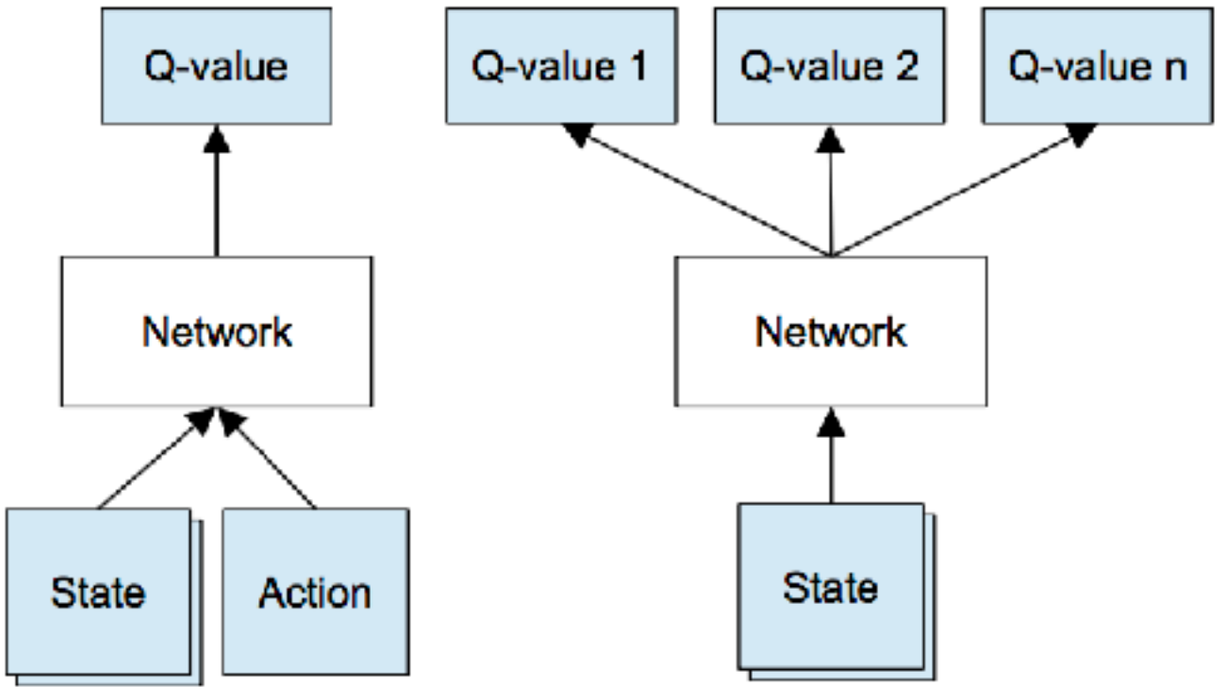
- Can be iteratively estimated using the **Bellman equation**

$$Q(s, a) = r + \gamma \max_{a'} Q(s', a')$$

- No neural network yet - could be done with a table of states vs actions

Deep Q Network

- Can train a neural network to estimate the Q function by playing out many episodes of a game



- Several subtleties - e.g. training using experience replay

Deep Q Network