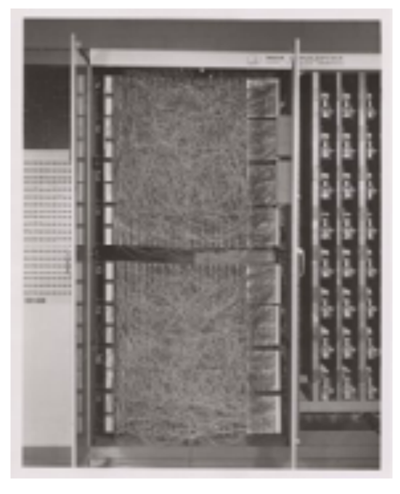
Deep Learning: Introduction

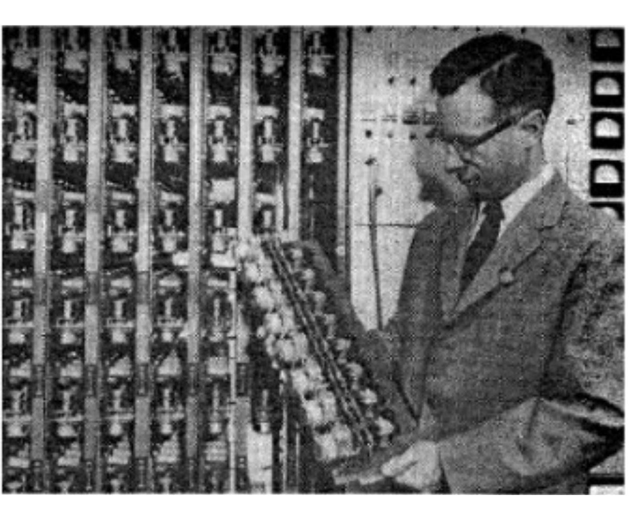
Adam Moss
School of Physics and Astronomy
adam.moss@nottingham.ac.uk

Early Learning

- Fe

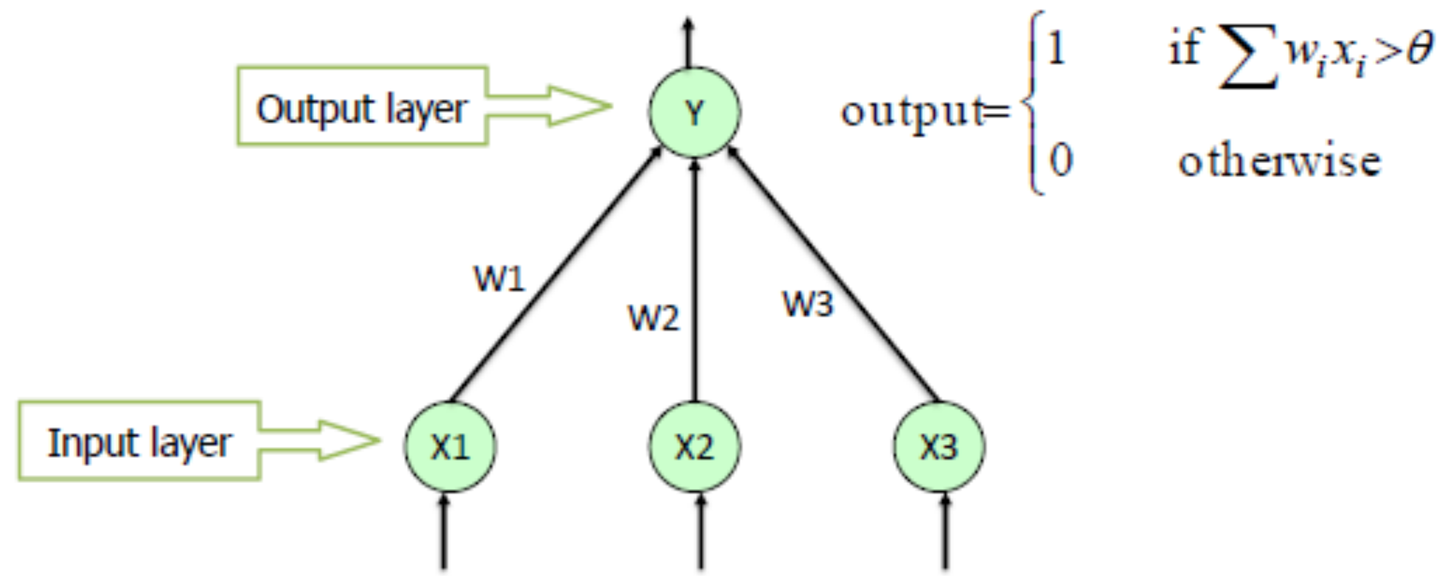
- Perceptron (binary classifier). Mark 1 perceptron machine (1957) used motors, potentiometers!





Perceptron

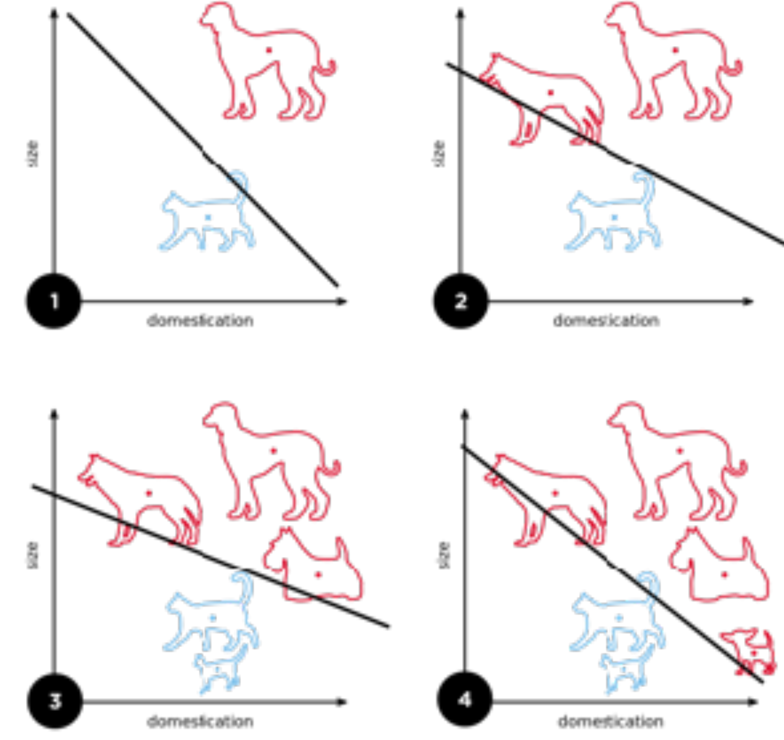
- Simplest perceptron: set of inputs x_i mapped to output $\hat{y}$



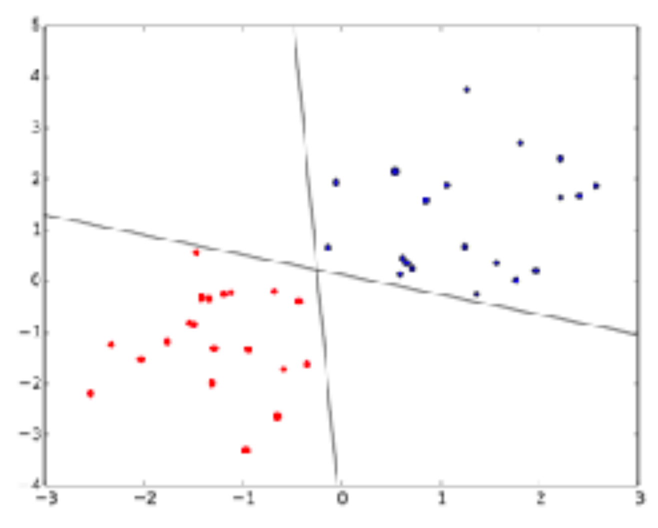
- Each input has a weight w_i
- Weights are trained using **supervised learning**
- Training sets of $D = \{x_{i,j}, y_j\}$ where j is the sample number and y_j the desired output for that sample
- Prediction is $\hat{y}$ and weights are updated to minimise loss $\sum_j (y_j - \hat{y}_j)^2$

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

neuron cell body
synapse
nucleus
axon of previous neuron
axon
neuron cell body
nucleus
axon tips
dendrites of next neuron
synapse
electrical signal
dendrites

[picture from Simon Thorpe]

[Gallant & Van Essen]

LGN=lateral geniculate nucleus

AIT=anterior inferior temporal cortex

PIT=posterior inferior temporal cortex

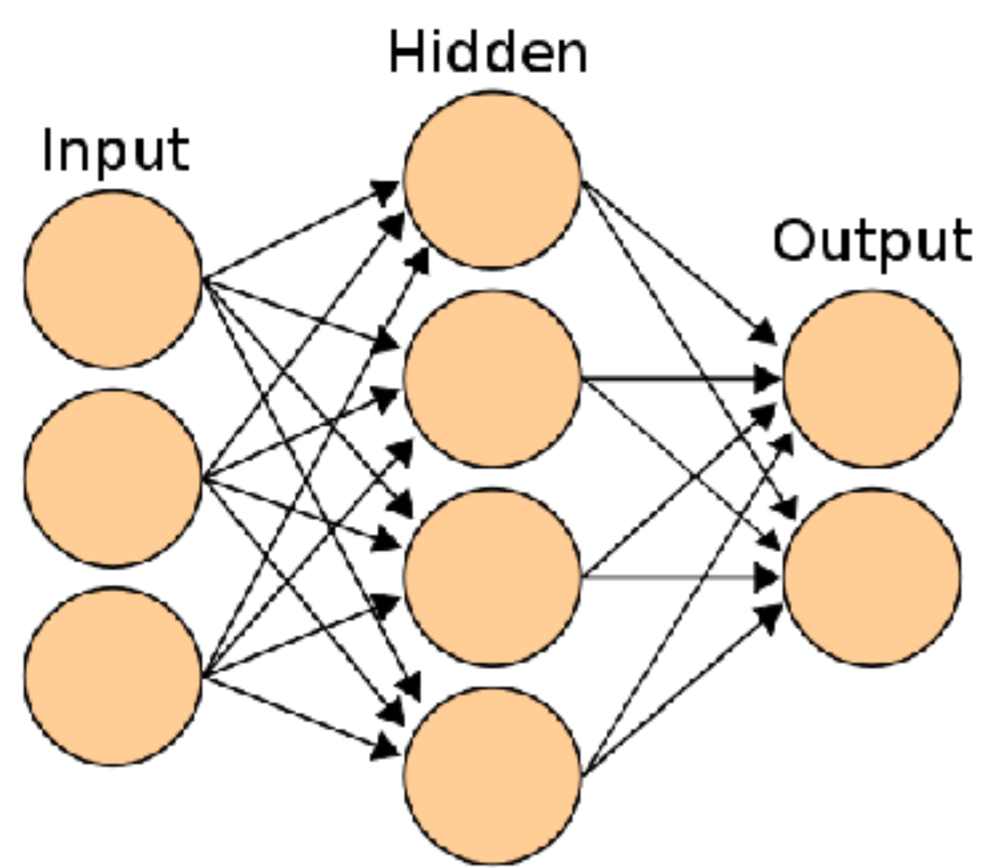
Shallow vs Deep

- 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

Layer 1
Layer 2
Layer 3
Layer 4
Layer 5

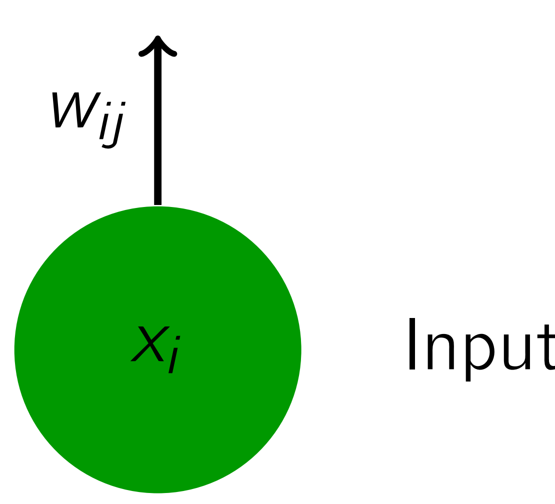
MultiLayer Perceptron

- 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**



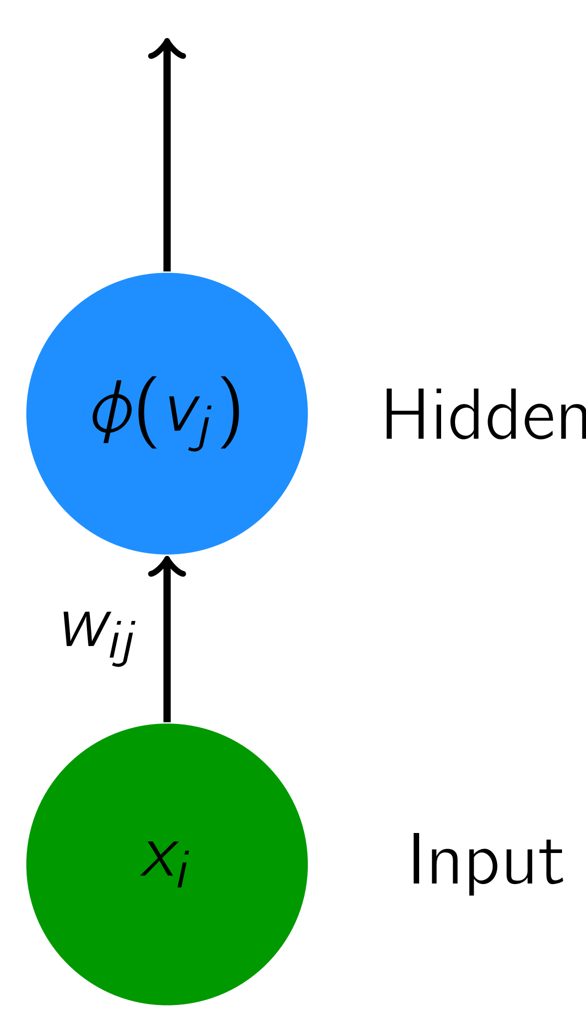
Back-propagation

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



Back-propagation

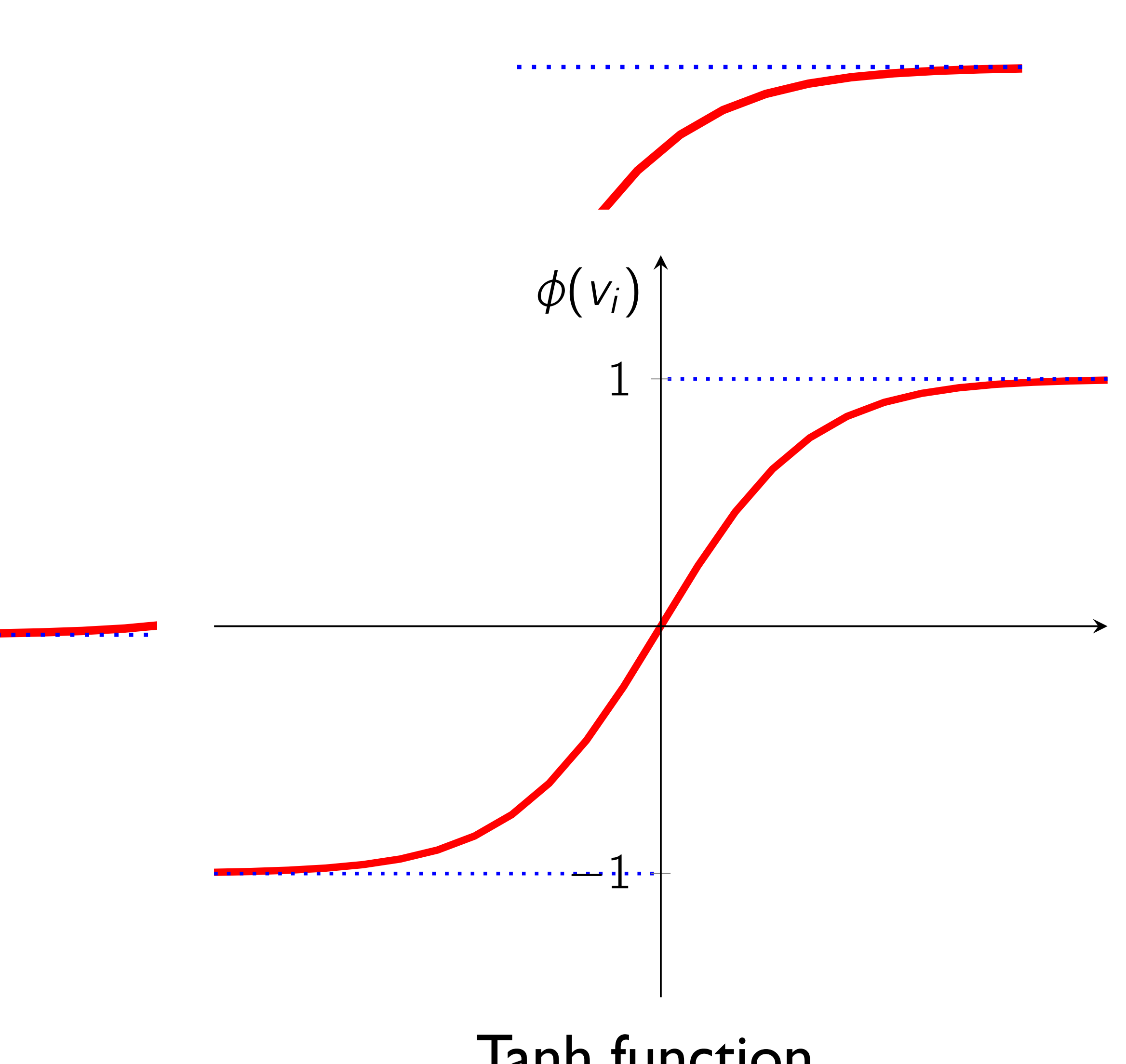
- Weighted inputs $v_j = w_{ij} x_i$
- Activation function $\phi(v_j)$



Back-propagation

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

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



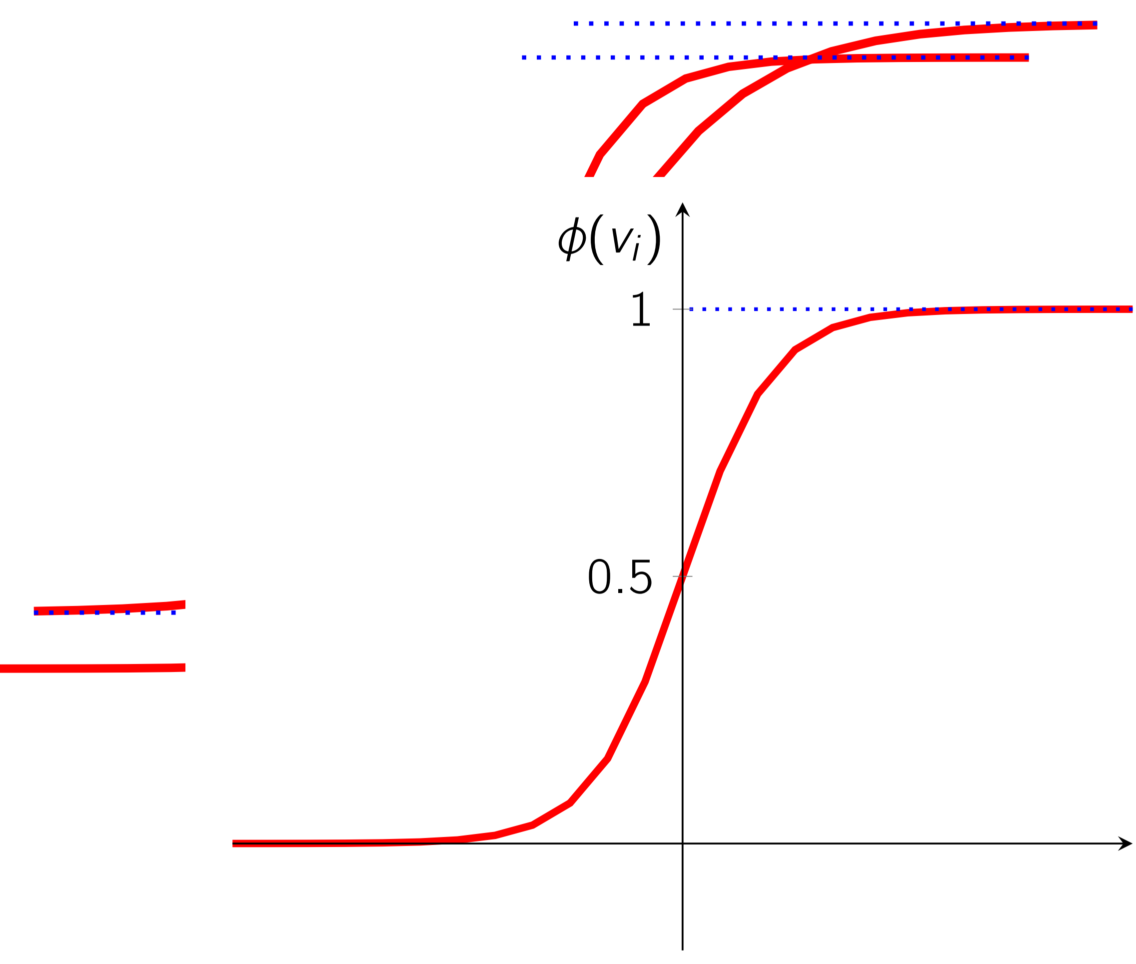
Tanh function

Back-propagation

- 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 $\phi(v_j)$

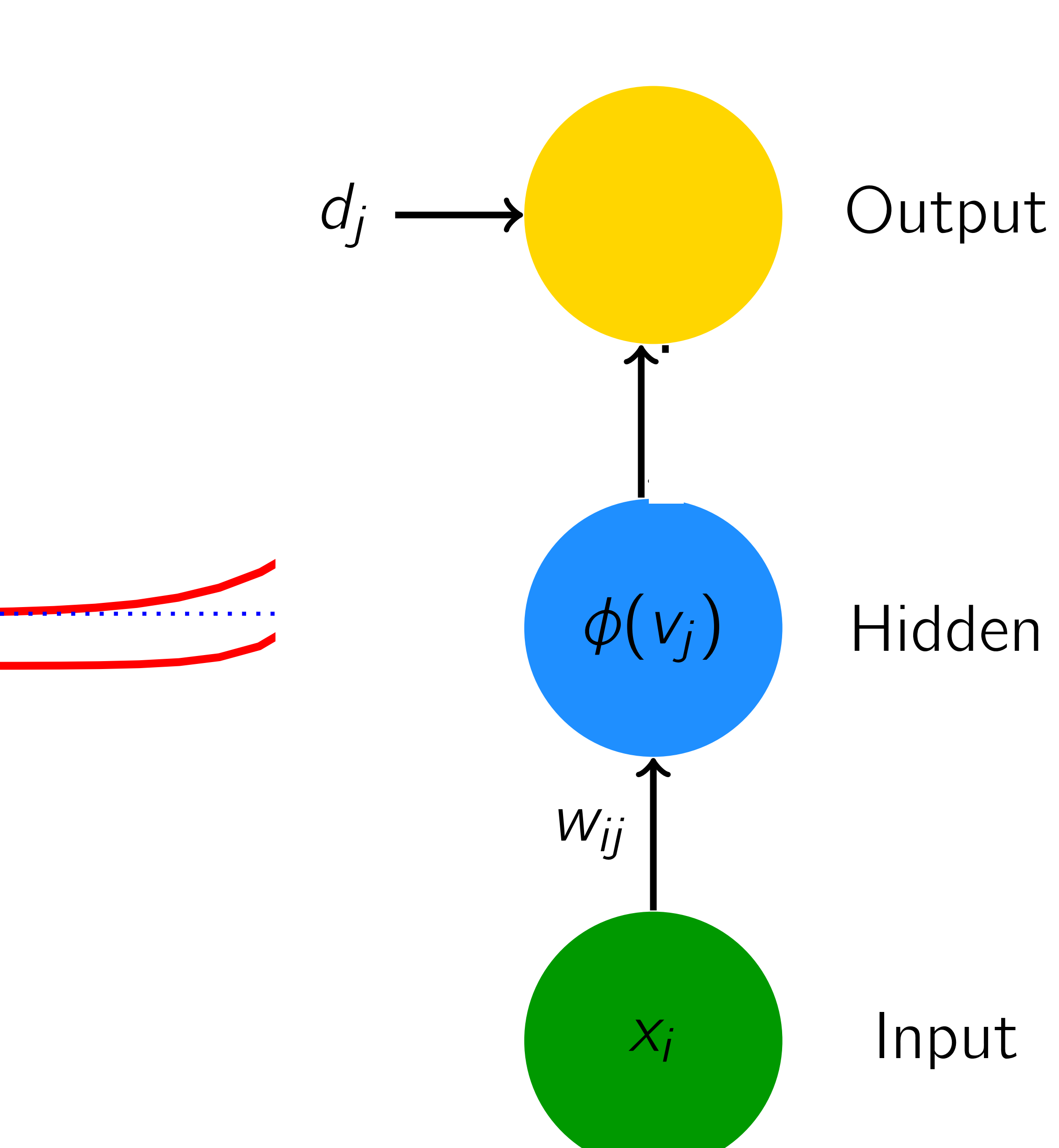
$$\phi(v_i) = \tanh(v_i)$$

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

- Loss function

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

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



Back-propagation

- 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

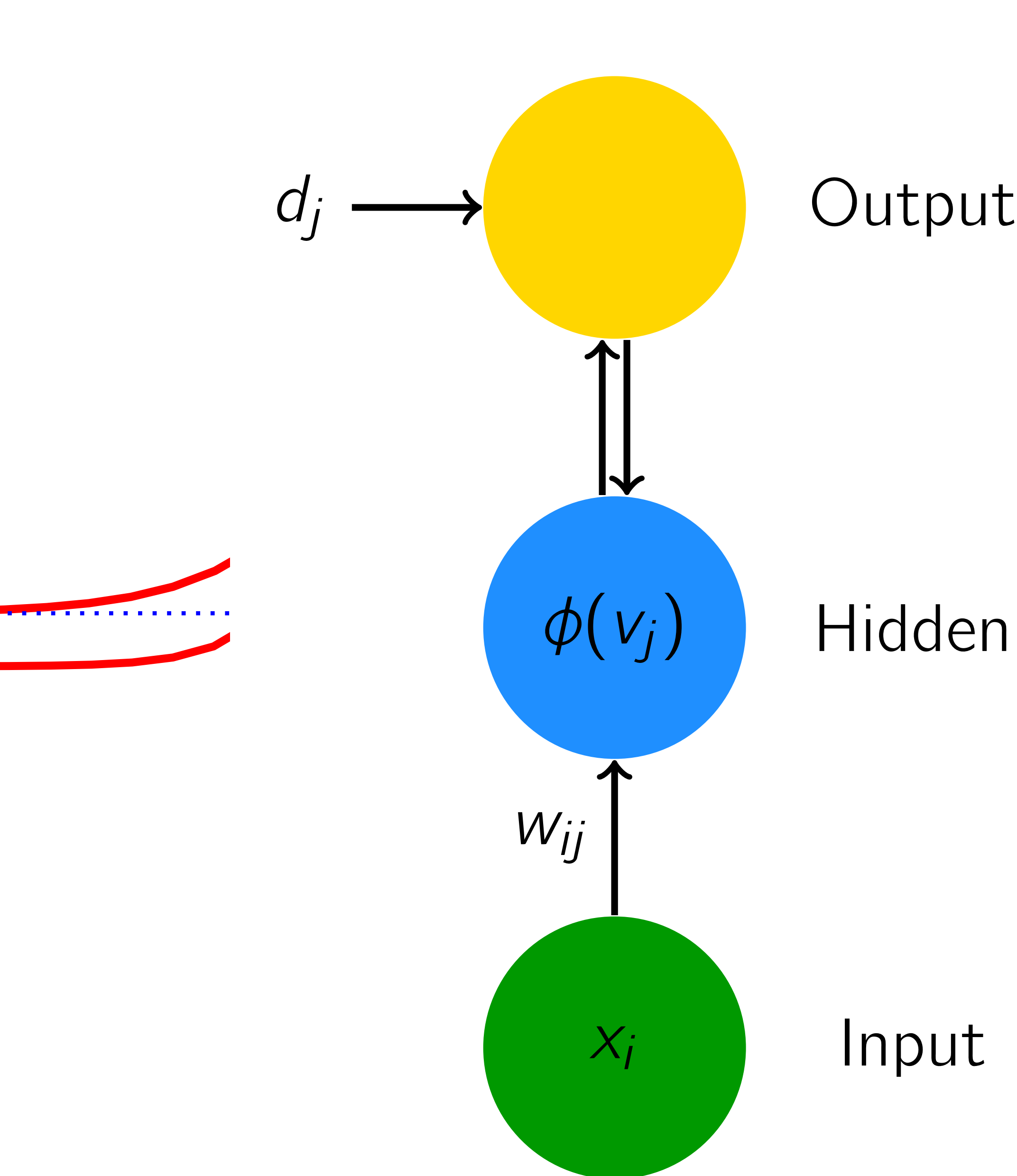
$$e_j = y_j - \phi(v_j)$$

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

- 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

- 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)

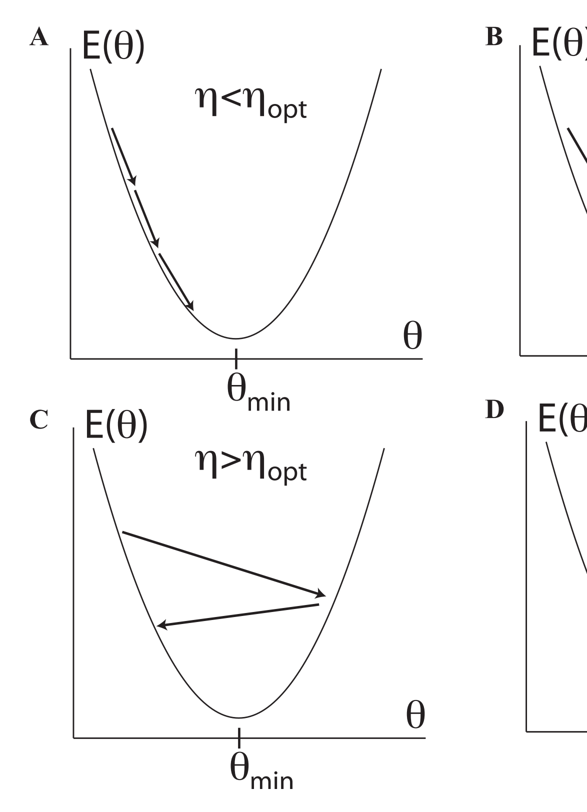
- Most supervised machine learning algorithms involve finding a set of parameters **Θ** that minimise a cost/loss function
- Loss functions can be complicated non-convex functions with many local minima
- Simplest algorithm that attempts to minimise the loss E(**Θ**) is gradient descent

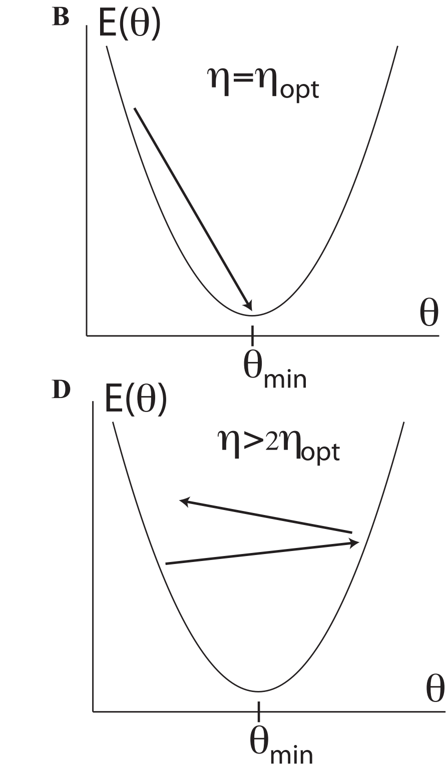
$$\mathbf{v}_t = \eta_t \nabla_\theta E(\boldsymbol{\theta}_t),$$

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \mathbf{v}_t$$

$$\eta_t = \text{learning rate}$$

- 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**

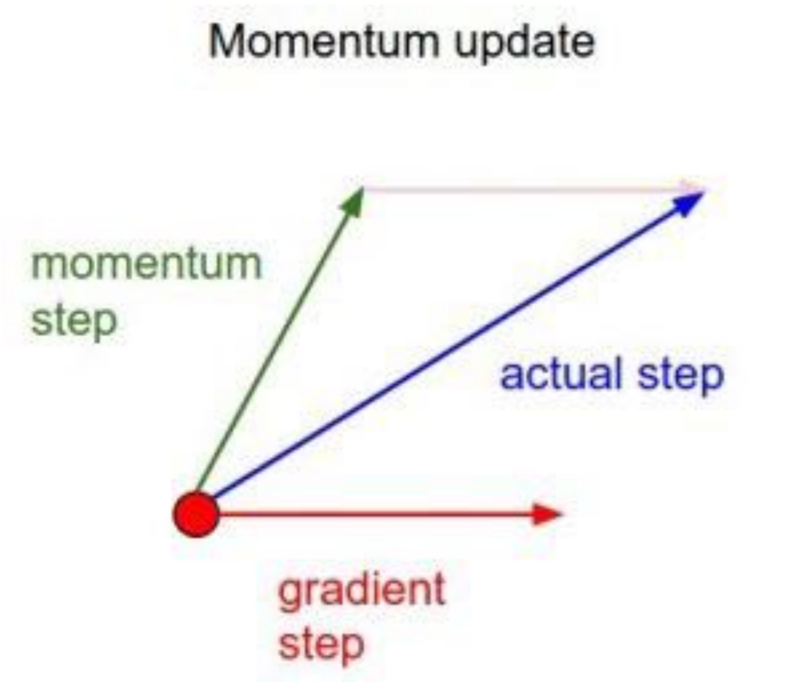
SGD with momentum

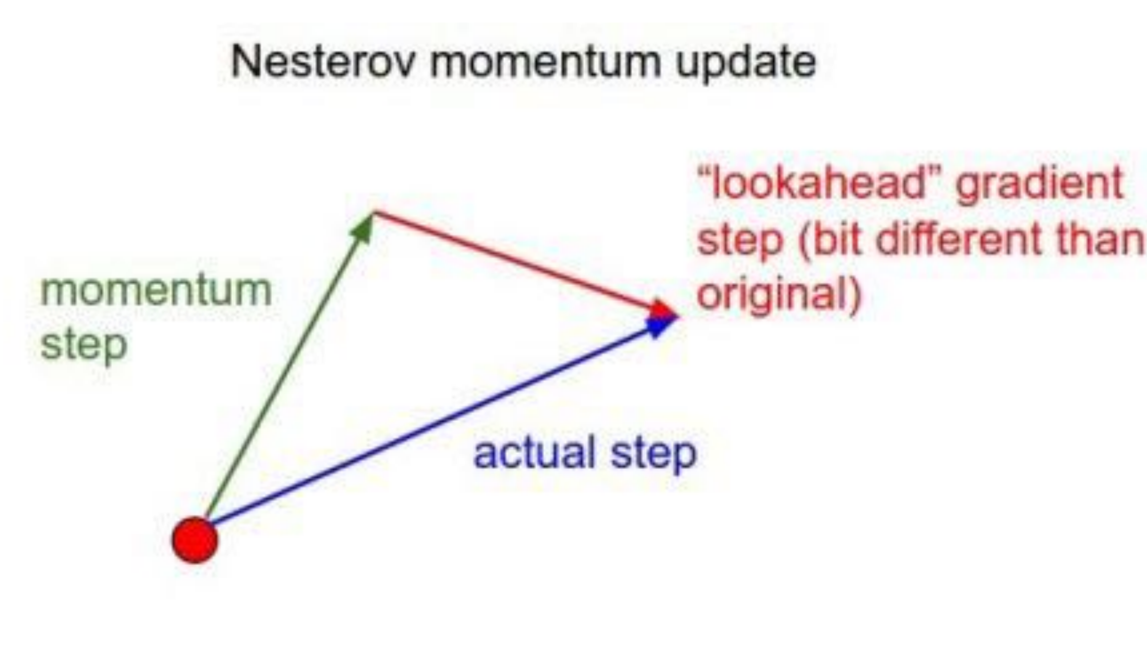
- SGD with momentum adds an inertia term that retains some memory of the direction being moved in

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

γ = 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 $\eta_{\text{opt}} = [\partial^2_\theta E(\theta)]^{-1}$.
- 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

$$\begin{aligned}\mathbf{g}_t &= \nabla_\theta E(\boldsymbol{\theta}) \\ \mathbf{s}_t &= \beta \mathbf{s}_{t-1} + (1-\beta)\mathbf{g}_t^2 \\ \boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t - \eta_t \frac{\mathbf{g}_t}{\sqrt{\mathbf{s}_t + \epsilon}},\end{aligned}$$

$$\begin{aligned}\mathbf{s}_t &= \mathbb{E}[\mathbf{g}_t^2] \\ \beta &= \text{averaging time} \\ \epsilon &= \text{regularisation constant}\end{aligned}$$

Second order methods

SGD
Momentum
NAG
Adagrad
Adadelta
Rmsprop

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/

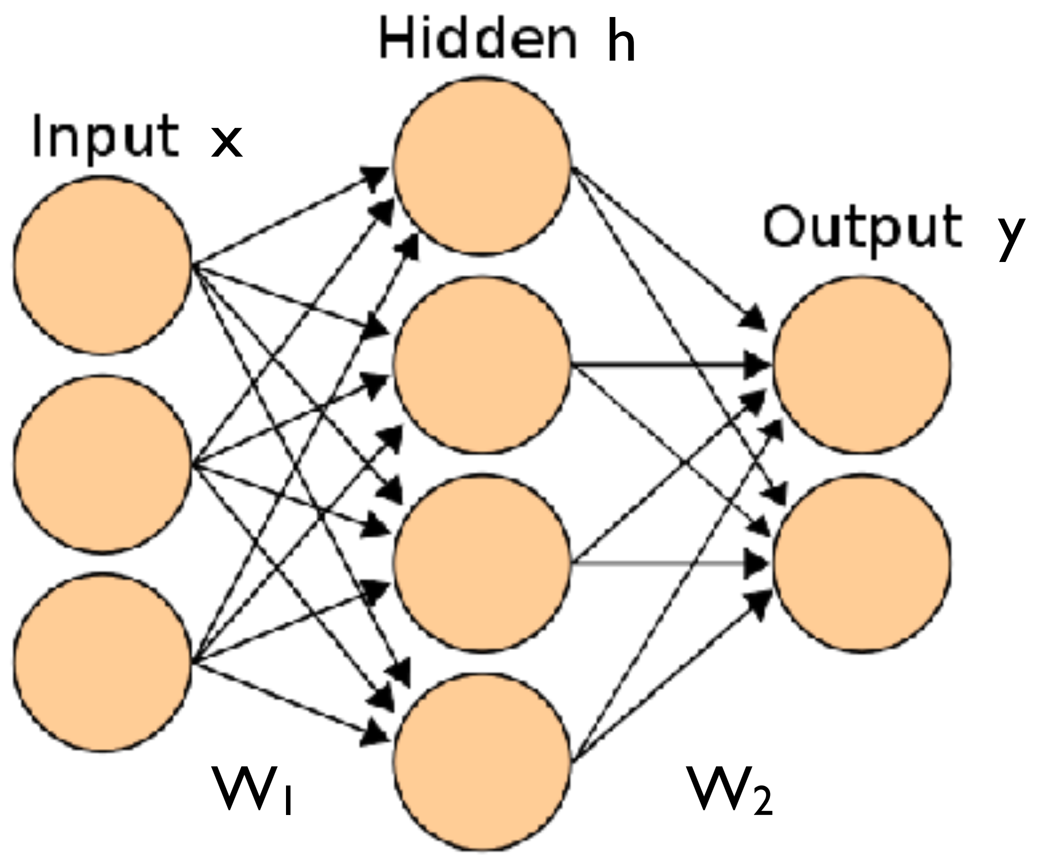
Deep learning in practice

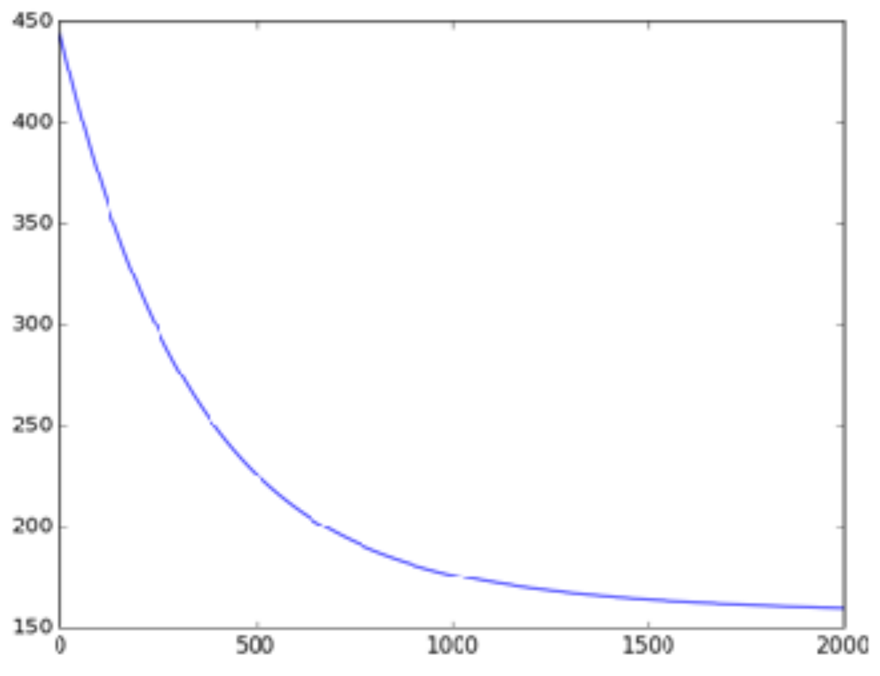
PyTorch

- 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





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 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

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

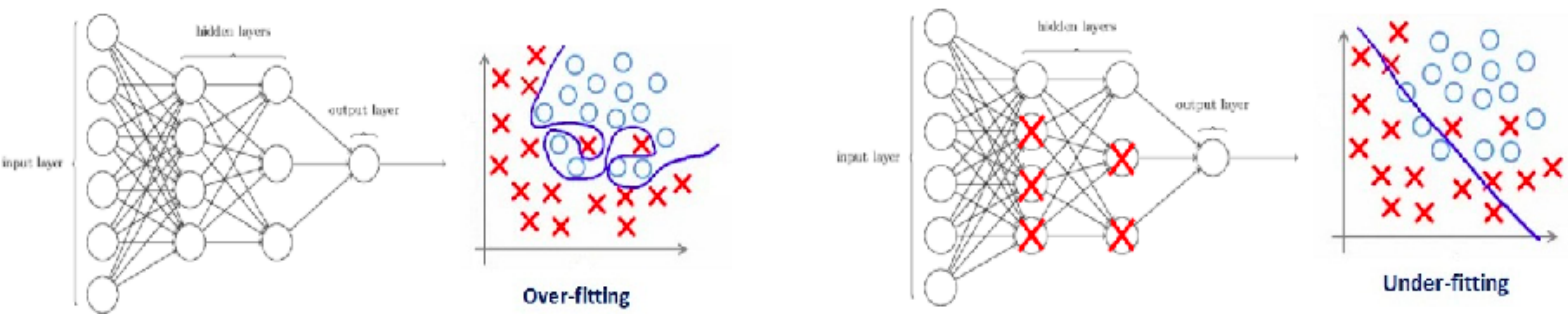
- Full loss function can include additional regularization terms
- For **categorical** outputs loss function is usually the categorical cross-entropy
- Last layer typically has a **soft-max** activation (turns M outputs into normalized probabilities)

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

- 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

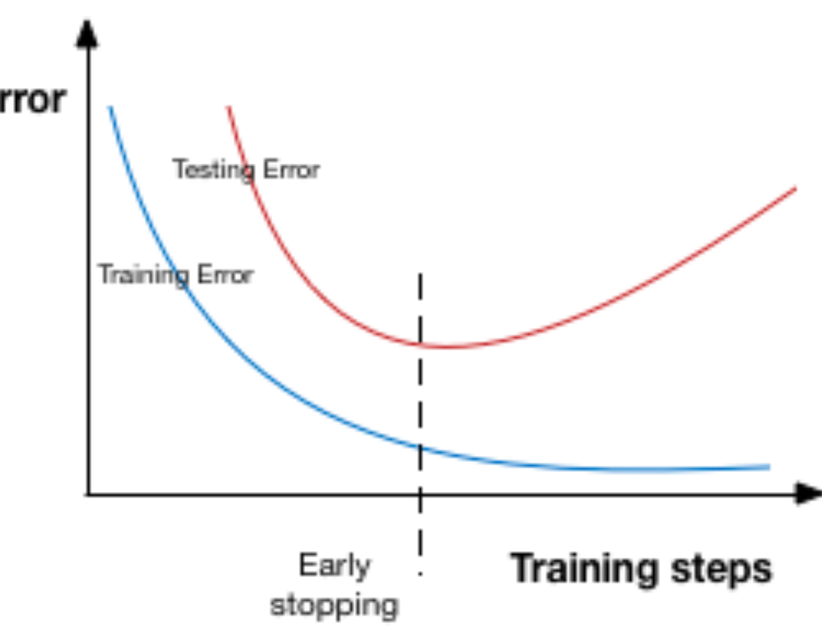


Model has too many free parameters and data is over-fitted

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

- 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

n_units n_units n_out

Input
MNIST image

Output
Classification result

[0, 0, 1, 0, 0, 0, 0, 0, 0, 0]

l1
Input layer

l2
Hidden layer

l3
Output layer

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)

autograd_example.py × keras_mnist_mlp.py × pytorch_mnist_mlp.py ×

```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()
```