Australian National University

COMP4650/6490 Document Analysis

Deep Neural Networks — Part I

ANU School of Computing

Administrative matters

- Assignment 2
  - Minor update: Fixed 2 small issues in `clustering.py`
- Assignment 1
  - Expected return date: Tuesday 29 August

Outline

- Neural networks & why NN
- Feedforward neural network
  - From logistic regression to Feedforward NN
  - Non-linear activation functions
- Back-propagation
  - Gradient descent
  - Back-propagation essentials
  - Computation graph
- Optimisers
  - Stochastic gradient descent
  - SGD with momentum & Adam

Outline

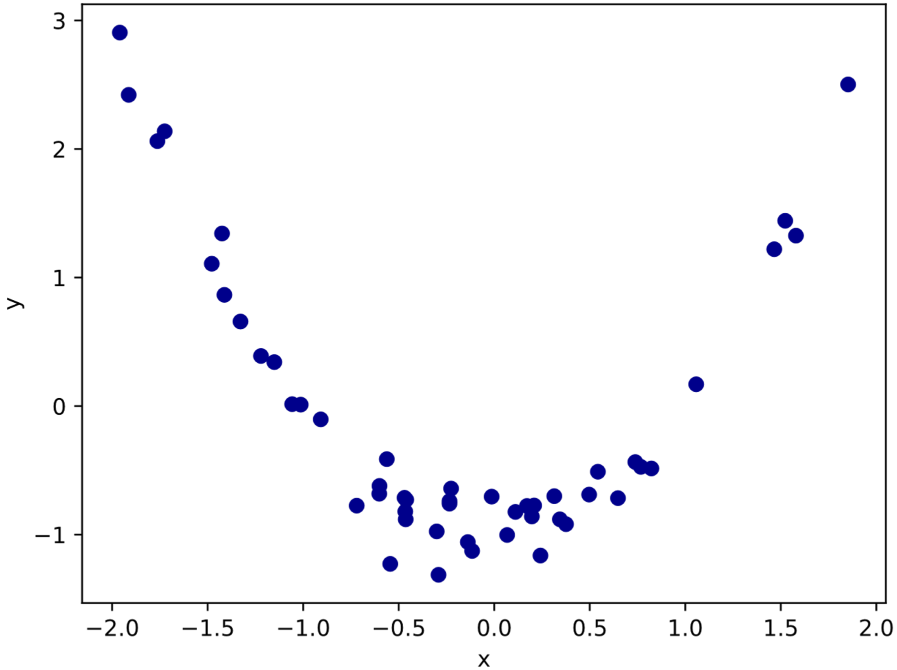
- Neural networks & why NN
- Feedforward neural network
  - From logistic regression to Feedforward NN
  - Non-linear activation functions
- Back-propagation
  - Gradient descent
  - Back-propagation essentials
  - Computation graph
- Optimisers
  - Stochastic gradient descent
  - SGD with momentum & Adam

Neural networks

- Computing systems initially inspired by *simplified* models of the brain
- We keep some of the ideas and terminology, but it is *NOT* a current model of the brain
- A better way to think of modern neural networks
  - Differentiable vector cascades
  - Complex functions: A neural network is a stack of linear and non-linear functions

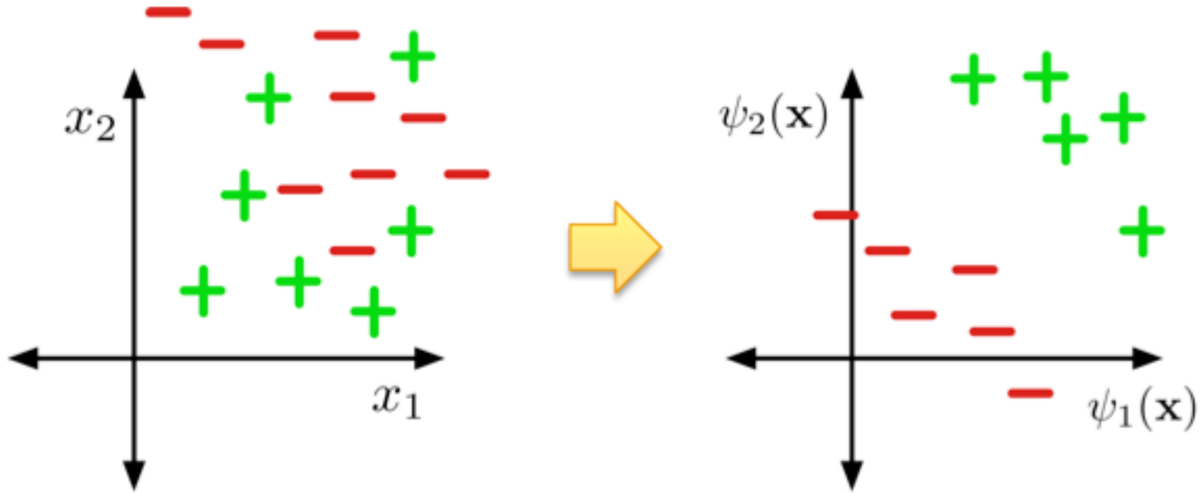
Why neural networks

- Non-linear relationships
  - What do we do if the target does not have a linear relationship with the input?
  - We need to fit a *non-linear* function to our data.



Why neural networks

- Non-linear relationships
  - What do we do if the target does not have a linear relationship with the input?
  - We need to fit a *non-linear* function to our data.
- Feature learning
  - Neural nets can be viewed as a way of learning features

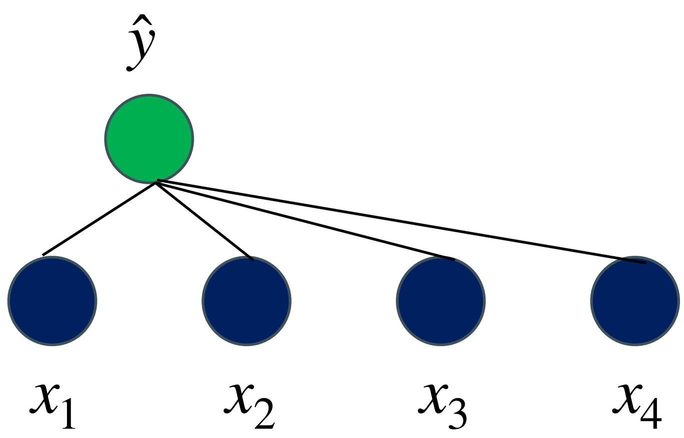


Outline

- Neural networks & why NN
- Feedforward neural network
  - From logistic regression to Feedforward NN
  - Non-linear activation functions
- Back-propagation
  - Gradient descent
  - Back-propagation essentials
  - Computation graph
- Optimisers
  - Stochastic gradient descent
  - SGD with momentum & Adam

From logistic regression to feedforward NN

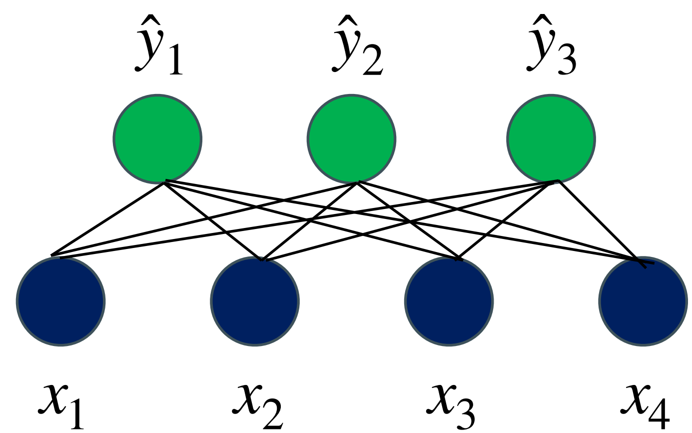
- Binary logistic regression
  - A 1-layer NN (single output)
- Multinomial logistic regression
  - A 1-layer NN (multiple outputs)
- Multi-layer perceptron (MLP)
  - Stack multiple (non-)linear models
  - Fully connected feedforward NN
  - Composition of functions



$$\hat{y} = \sigma(\mathbf{w}^\top \mathbf{x} + b)$$

From logistic regression to feedforward NN

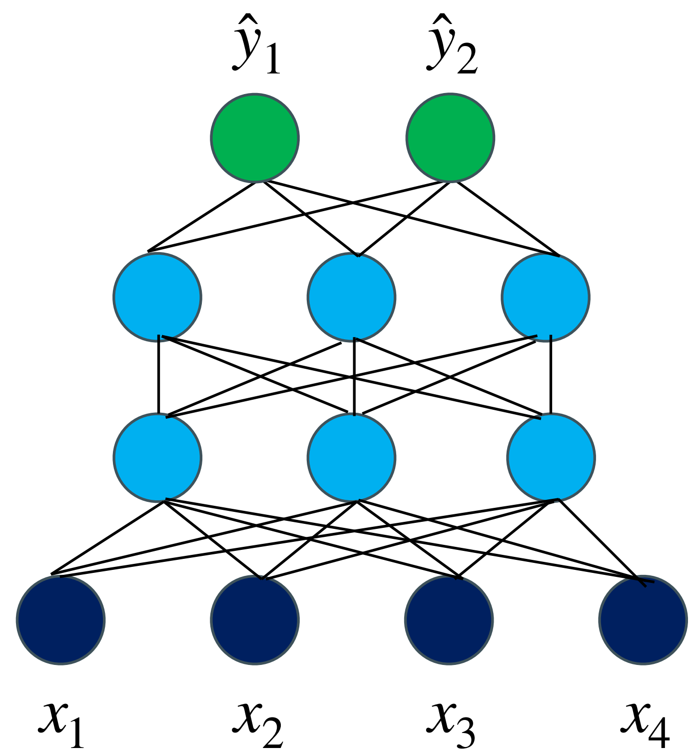
- Binary logistic regression
  - A 1-layer NN (single output)
- Multinomial logistic regression
  - A 1-layer NN (multiple outputs)
- Multi-layer perceptron (MLP)
  - Stack multiple (non-)linear models
  - Fully connected feedforward NN
  - Composition of functions



$$\hat{\mathbf{y}} = \text{softmax}(W\mathbf{x} + \mathbf{b})$$

From logistic regression to feedforward NN

- Binary logistic regression
  - A 1-layer NN (single output)
- Multinomial logistic regression
  - A 1-layer NN (multiple outputs)
- Multi-layer perceptron (MLP)
  - Stack multiple (non-)linear models
  - Fully connected feedforward NN
  - Composition of functions



$\mathbf{x}$ → $\phi^{[1]}(W^{[1]}\mathbf{x} + \mathbf{b}^{[1]})$ → $\phi^{[2]}(W^{[2]}\mathbf{h}^{[1]} + \mathbf{b}^{[2]})$ → $\phi^{[3]}(W^{[3]}\mathbf{h}^{[2]} + \mathbf{b}^{[3]})$

Input | Hidden $\mathbf{h}^{[1]}$ | Hidden $\mathbf{h}^{[2]}$ | Output $\hat{\mathbf{y}}$

Notation

Computation in layer l (superscript $[l]$ notation denotes layer l)

$$\mathbf{h}^{[l]} = \phi^{[l]}(W^{[l]}\mathbf{h}^{[l-1]} + \mathbf{b}^{[l]})$$

- $\mathbf{h}^{[l]}$: output of layer l
- $\phi^{[l]}$: activation function of layer l (usually a non-linear function)
- $W^{[l]}$: weight matrix of layer l
- $\mathbf{h}^{[l-1]}$: input of layer l, i.e. output of layer l-1, and $\mathbf{h}^{[0]} = \mathbf{x}$
- $\mathbf{b}^{[l]}$: biases of layer l, note that sometimes we write without the bias since it can be absorbed into $W^{[l]}$ by adding an extra dimension of value 1 to the input vector $\mathbf{h}^{[l-1]}$

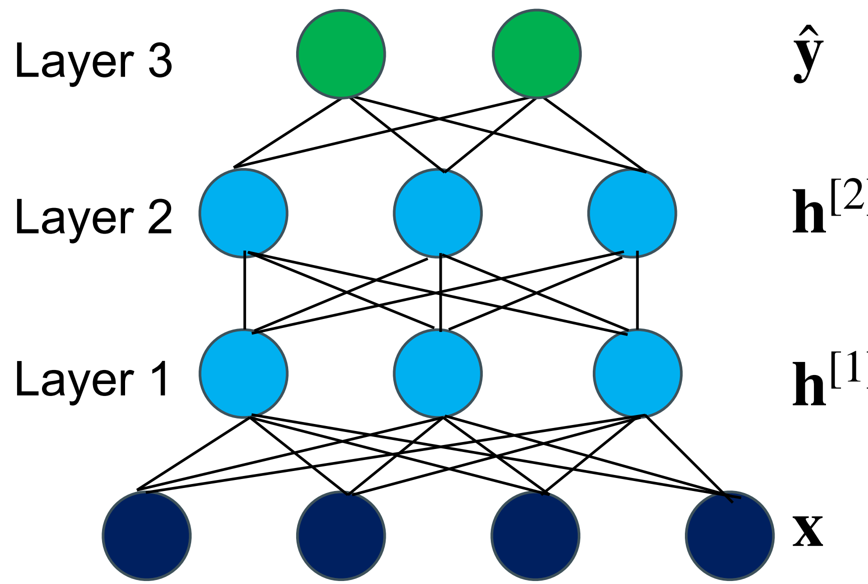
Notation

Computation in layer l (superscript $[l]$ notation denotes layer l)

$$\mathbf{h}^{[l]} = \phi^{[l]}(W^{[l]}\mathbf{h}^{[l-1]} + \mathbf{b}^{[l]})$$

Example:

- A 3-layer fully connected Feedforward NN (i.e. MLP) and its forward computation
- We usually don't count the input layer



$$\mathbf{x} \longrightarrow \phi^{[1]}(W^{[1]}\mathbf{x} + \mathbf{b}^{[1]}) \longrightarrow \phi^{[2]}(W^{[2]}\mathbf{h}^{[1]} + \mathbf{b}^{[2]}) \longrightarrow \phi^{[3]}(W^{[3]}\mathbf{h}^{[2]} + \mathbf{b}^{[3]})$$

Input	Hidden $\mathbf{h}^{[1]}$	Hidden $\mathbf{h}^{[2]}$	Output $\hat{\mathbf{y}}$

Non-linear activation functions

Why use a non-linearity?

$$
\begin{aligned}
NN(\mathbf{x}) &= W^{[2]}(W^{[1]}\mathbf{x} + \mathbf{b}^{[1]}) + \mathbf{b}^{[2]} \\
&= W^{[2]}W^{[1]}\mathbf{x} + (W^{[2]}\mathbf{b}^{[1]} + \mathbf{b}^{[2]})
\end{aligned}
$$

Non-linear activation functions

Why use a non-linearity?

$$\begin{aligned} NN(\mathbf{x}) &= W^{[2]}(W^{[1]}\mathbf{x} + \mathbf{b}^{[1]}) + \mathbf{b}^{[2]} \\ &= W^{[2]}W^{[1]}\mathbf{x} + (W^{[2]}\mathbf{b}^{[1]} + \mathbf{b}^{[2]}) \\ &= W'\mathbf{x} + \mathbf{b}' \end{aligned}$$

where $W' = W^{[2]}W^{[1]}$ and $\mathbf{b}' = W^{[2]}\mathbf{b}^{[1]} + \mathbf{b}^{[2]}$.

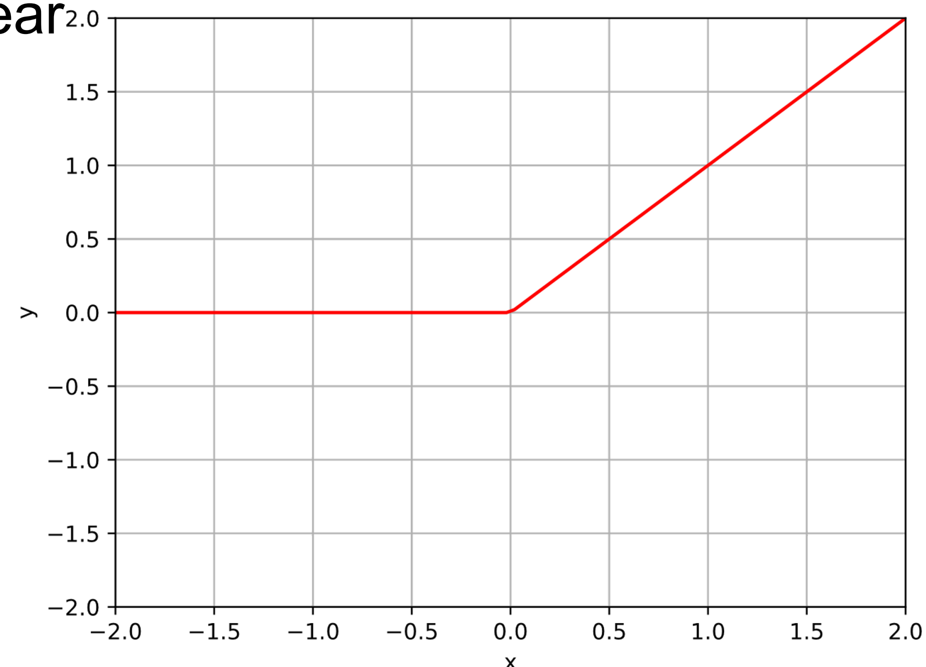
This is still a linear model!

Non-linear activation functions

Why use a non-linearity?

$$NN(\mathbf{x}) = W^{[2]}\phi(W^{[1]}\mathbf{x} + \mathbf{b}^{[1]}) + \mathbf{b}^{[2]}$$

- ϕ is *some* function that is non-linear and (mostly) differentiable
- ϕ is called the *activation function*
- In principle we can use any non-linear (mostly) differentiable function as the activation function
- Commonly used in practice is ReLU$(x) = \max(x, 0)$ which is fast to compute and works well



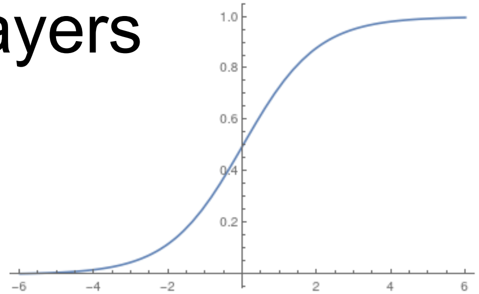
Non-linear activation functions

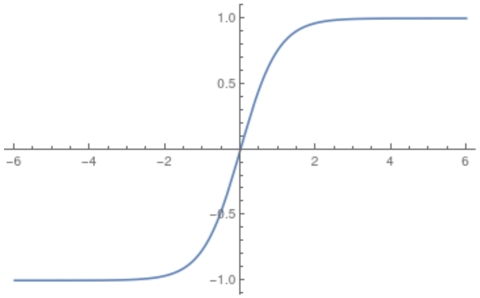
Typical activation functions for *hidden* layers

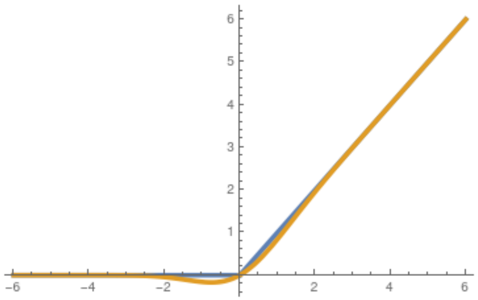
- Sigmoid
  - $\sigma(z) = (1 + \exp(-z))^{-1}$
  - It was popular for a long period of time
- Tanh:

$$\tanh(z) = \frac{\exp(z) - \exp(-z)}{\exp(z) + \exp(-z)}$$
$$= 2\sigma(2z) - 1$$

- Rectified linear unit (ReLU) and its smooth approximations
  - $\mathrm{ReLU}(z) = \max(z, 0)$
  - e.g. GELU, see https://en.wikipedia.org/wiki/Rectifier_(neural_networks)







Non-linear activation functions

Typical activation functions for the *output* layer

- For classification
  - Use *softmax* (multi-class) or *sigmoid* (multi-label)
- For regression
  - Use a linear last layer
    i.e. $\phi^{[L]}$ is the identity function in an L-layer NN

Feedforward NN

Terminology

- Layers
  - For a Feedforward NN, we typically think of each composition of an activation function with a linear function as a *layer*
  - Layers except the input and output layers are *hidden layers*
  - In general, *almost anything*, regardless of complexity, can be called a layer, much like almost anything can be called a function.
  - The term is used to conceptually distinguish between different components

Feedforward NN

Terminology

- Hidden dimensions
  - The size of the *input* layer is defined by the number of input features
  - The size of the *output* layer is defined by the number of output targets
  - The size of the *hidden* layers can be anything, this is a hyper-parameter for you to choose
  - The dimension of the output of a *hidden* layer is called the *number of hidden units* in that layer

Terminology

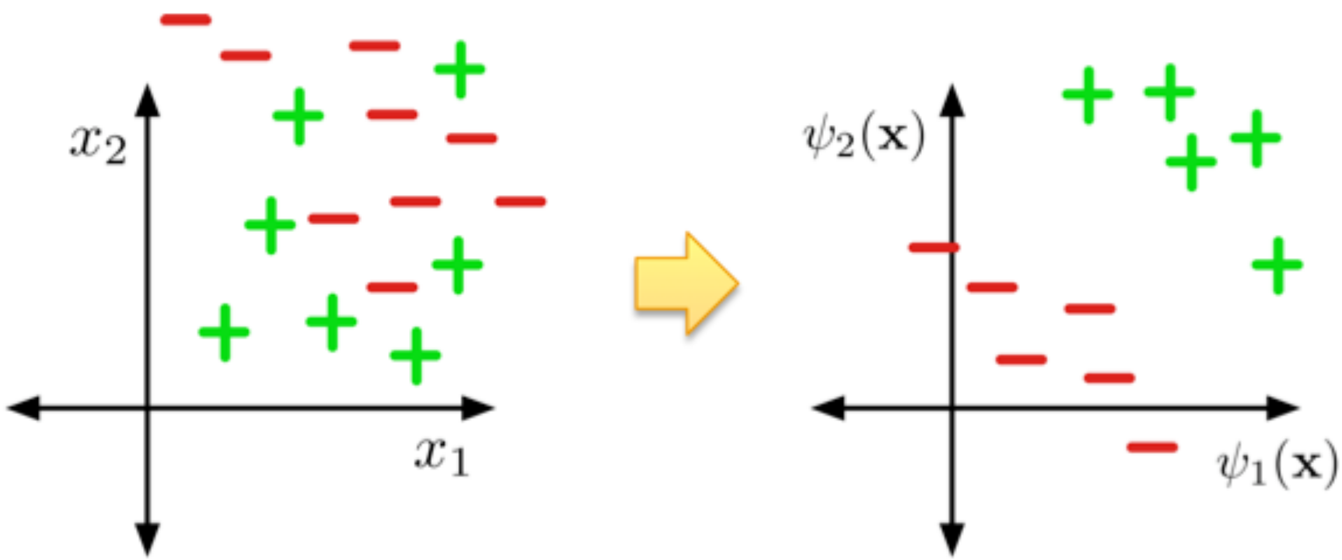
- Feature / representation learning
  - The intermediate models learns to output a *useful representation* of the input, e.g. $\mathbf{h}^{[1]}$ and $\mathbf{h}^{[2]}$ below.
  - The final model still learns to predict the desired output
  - Neural nets can be viewed as a way of learning features

$$\mathbf{x} \longrightarrow \phi^{[1]}(W^{[1]}\mathbf{x} + \mathbf{b}^{[1]}) \longrightarrow \phi^{[2]}(W^{[2]}\mathbf{h}^{[1]} + \mathbf{b}^{[2]}) \longrightarrow \phi^{[3]}(W^{[3]}\mathbf{h}^{[2]} + \mathbf{b}^{[3]})$$

Input | Hidden $\mathbf{h}^{[1]}$ | Hidden $\mathbf{h}^{[2]}$ | Output $\hat{\mathbf{y}}$

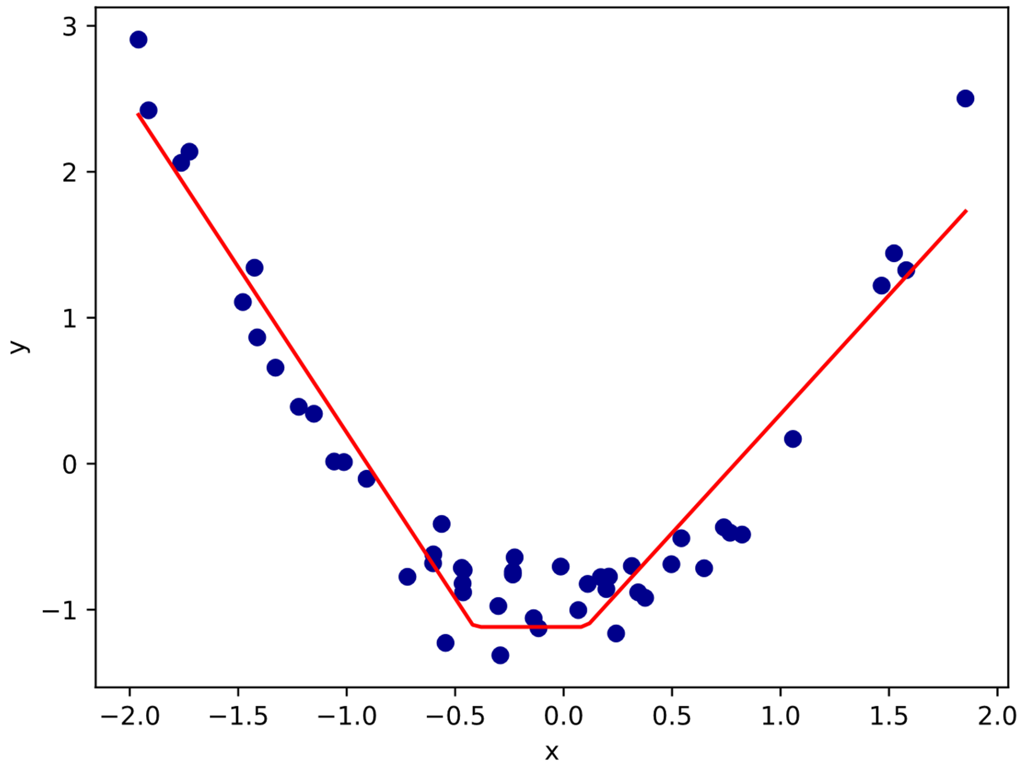
Terminology

- Feature / representation learning
  - The intermediate models learns to output a *useful representation* of the input, e.g. $\mathbf{h}^{[1]}$ and $\mathbf{h}^{[2]}$ below.
  - The final model still learns to predict the desired output
  - Neural nets can be viewed as a way of learning features



Example

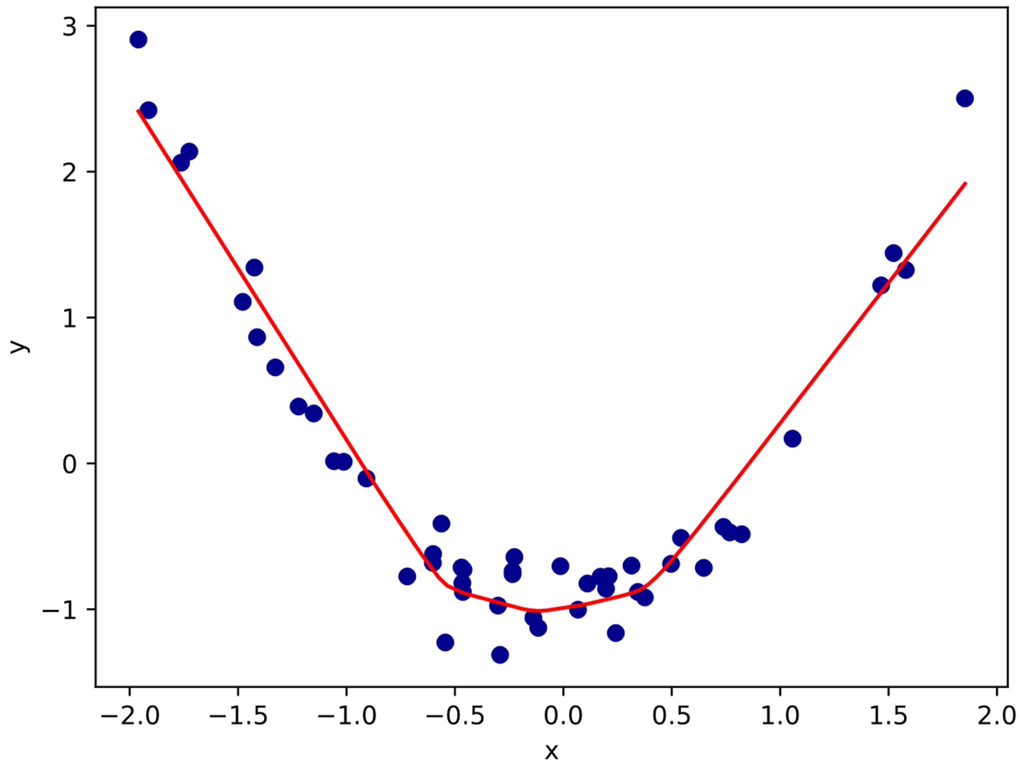
Fitting a Feedforward NN with 1 hidden layer of 2 hidden units with ReLU activations



- Each hidden unit learns a linear function then applies ReLU.
- Output is then a linear function of the hidden outputs.
- Resulting graph is equivalent to scaling, shifting and flipping two ReLU graphs and then adding them.

Example

Fitting a Feedforward NN with 1 hidden layer of 50 hidden units with ReLU activations



- Increasing the number of hidden units means we add together more (scaled) versions of the activation function.
- As we add more units we can approximate more complicated functions.
- As we add more units, our model becomes more capable of fitting noise in the training dataset — overfitting.

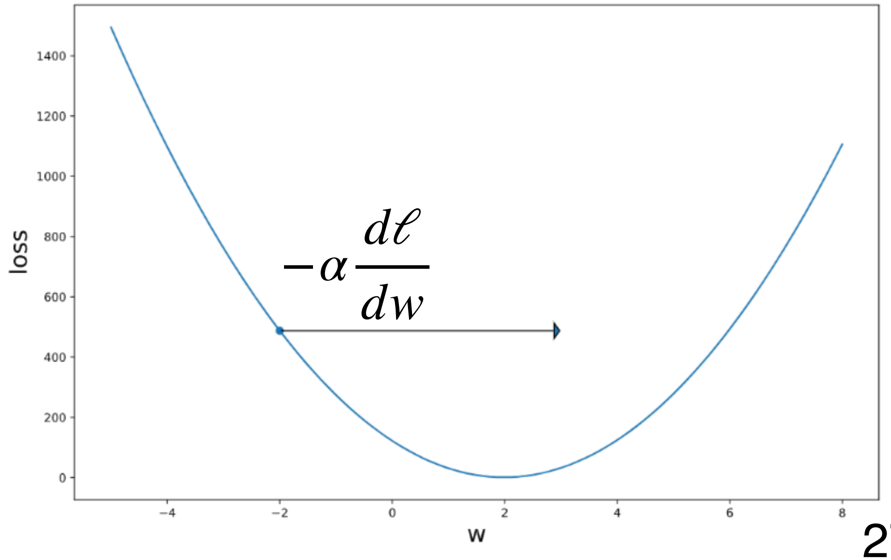
Outline

- Neural networks & why NN
- Feedforward neural network
  - From logistic regression to Feedforward NN
  - Non-linear activation functions
- Back-propagation
  - Gradient descent
  - Back-propagation essentials
  - Computation graph
- Optimisers
  - Stochastic gradient descent
  - SGD with momentum & Adam

Gradient descent

How can we learn the parameters $W^{[l]}$ and $\mathbf{b}^{[l]}$?

- Use gradient descent, and we need
  - Loss function $\ell(\mathbf{y}, \hat{\mathbf{y}})$
  - Partial derivatives of the loss function ℓ w.r.t. the parameters $W^{[l]}$ and $\mathbf{b}^{[l]}$
- To efficiently compute $\dfrac{\partial \ell}{\partial W^{[l]}}$ and $\dfrac{\partial \ell}{\partial \mathbf{b}^{[l]}}$
  - Use back-propagation



Back-propagation essentials

To efficiently compute the partial derivatives of the loss with respect to the parameters

- Forward pass
  - Compute the loss and the output of each layer
- Backward pass
  - Compute the partial derivatives layer by layer, starting from the last layer, using the *chain rule* and the results computed in the forward pass
  - A (bottom-up) *dynamic programming* (or memoisation) algorithm that avoids redundant calculations of intermediate terms

Back-propagation

Chain rule

- Given $z = f(x(t))$, the derivative

$$\frac{dz}{dt} = \frac{dz}{dx}\frac{dx}{dt}$$

- Given $z = g(x(t), y(t))$, the derivative

$$\frac{dz}{dt} = \frac{\partial z}{\partial x}\frac{dx}{dt} + \frac{\partial z}{\partial y}\frac{dy}{dt}$$

Example: Forward pass

Given the model with loss function for data point $(x,\ y)$:

$$\ell(y,\ \hat{y}) = \frac{1}{2}\big(\phi(wx+b) - y\big)^2$$

We can introduce intermediate variables $z, \hat{y}$ to give:

$$z = wx + b$$

$$\hat{y} = \phi(z)$$

$$\ell = \frac{1}{2}(\hat{y} - y)^2$$

Compute and store $z, \hat{y}, \ell$ in the forward pass.

Example: Backward pass

Compute the derivative of the loss ℓ w.r.t. $\hat{y}$

$$\frac{d\ell}{d\hat{y}} = \frac{d\ \frac{1}{2}(\hat{y} - y)^2}{d\hat{y}} = \hat{y} - y$$

Note that both $z, \hat{y}, y$ are known:

- $z, \hat{y}$ are computed and stored during the forward pass
- y is the ground truth

Forward pass:

$z = wx + b$

$\hat{y} = \phi(z)$

$\ell = \frac{1}{2}(\hat{y} - y)^2$

Example: Backward pass

Compute the derivative of the loss ℓ w.r.t. $\hat{y}$

$$\frac{d\ell}{d\hat{y}} = \frac{d\ \frac{1}{2}(\hat{y} - y)^2}{d\hat{y}} = \hat{y} - y$$

Note that both $z, \hat{y}, y$ are known:

- $z, \hat{y}$ are computed and stored during the forward pass
- y is the ground truth

The chain rule states $\frac{d\ell}{dz} = \frac{d\ell}{d\hat{y}}\frac{d\hat{y}}{dz} = \frac{d\ell}{d\hat{y}}\phi'(z)$, and

$$\frac{d\ell}{dw} = \frac{d\ell}{dz}\frac{dz}{dw} = \frac{d\ell}{dz}x, \text{ and } \frac{d\ell}{db} = \frac{d\ell}{dz}\frac{dz}{db} = \frac{d\ell}{dz} \times 1$$

Forward pass:

$z = wx + b$

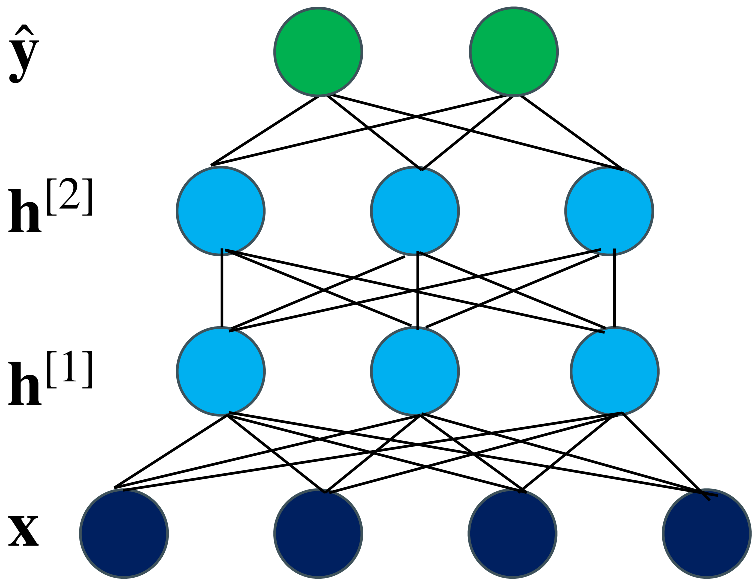
$\hat{y} = \phi(z)$

$\ell = \frac{1}{2}(\hat{y} - y)^2$

Back-propagation

Another example: Forward pass

Given data point $(\mathbf{x}, \mathbf{y})$ and loss function $\ell(\mathbf{y}, \hat{\mathbf{y}})$



$$\mathbf{a}^{[1]} = \mathbf{W}^{[1]}\mathbf{x} + \mathbf{b}^{[1]}$$
$$\mathbf{h}^{[1]} = \phi^{[1]}(\mathbf{a}^{[1]})$$
$$\mathbf{a}^{[2]} = \mathbf{W}^{[2]}\mathbf{h}^{[1]} + \mathbf{b}^{[2]}$$
$$\mathbf{h}^{[2]} = \phi^{[2]}(\mathbf{a}^{[2]})$$
$$\mathbf{a}^{[3]} = \mathbf{W}^{[3]}\mathbf{h}^{[2]} + \mathbf{b}^{[3]}$$
$$\hat{\mathbf{y}} = \phi^{[3]}(\mathbf{a}^{[3]})$$
$$\ell(\mathbf{y}, \hat{\mathbf{y}})$$

Another example: Backward pass

Given data point $(\mathbf{x},\ \mathbf{y})$ and loss function $\ell(\mathbf{y},\ \hat{\mathbf{y}})$

Forward pass:

$$\mathbf{a}^{[1]} = \mathbf{W}^{[1]}\mathbf{x} + \mathbf{b}^{[1]}$$
$$\mathbf{h}^{[1]} = \phi^{[1]}(\mathbf{a}^{[1]})$$
$$\mathbf{a}^{[2]} = \mathbf{W}^{[2]}\mathbf{h}^{[1]} + \mathbf{b}^{[2]}$$
$$\mathbf{h}^{[2]} = \phi^{[2]}(\mathbf{a}^{[2]})$$
$$\mathbf{a}^{[3]} = \mathbf{W}^{[3]}\mathbf{h}^{[2]} + \mathbf{b}^{[3]}$$
$$\hat{\mathbf{y}} = \phi^{[3]}(\mathbf{a}^{[3]})$$
$$\ell(\mathbf{y},\ \hat{\mathbf{y}})$$

Compute $\frac{\partial \ell}{\partial \mathbf{W}^{[1]}}$ using vector calculus[1] (more in this week's lab):

$$\frac{\partial \ell}{\partial \hat{\mathbf{y}}},\quad \frac{\partial \ell}{\partial \mathbf{a}^{[3]}} = \frac{\partial \ell}{\partial \hat{\mathbf{y}}}\frac{\partial \hat{\mathbf{y}}}{\partial \mathbf{a}^{[3]}}$$

$$\frac{\partial \ell}{\partial \mathbf{h}^{[2]}} = \frac{\partial \ell}{\partial \mathbf{a}^{[3]}}\frac{\partial \mathbf{a}^{[3]}}{\partial \mathbf{h}^{[2]}},\quad \frac{\partial \ell}{\partial \mathbf{a}^{[2]}} = \frac{\partial \ell}{\partial \mathbf{h}^{[2]}}\frac{\partial \mathbf{h}^{[2]}}{\partial \mathbf{a}^{[2]}}$$

$$\frac{\partial \ell}{\partial \mathbf{h}^{[1]}} = \frac{\partial \ell}{\partial \mathbf{a}^{[2]}}\frac{\partial \mathbf{a}^{[2]}}{\partial \mathbf{h}^{[1]}},\quad \frac{\partial \ell}{\partial \mathbf{a}^{[1]}} = \frac{\partial \ell}{\partial \mathbf{h}^{[1]}}\frac{\partial \mathbf{h}^{[1]}}{\partial \mathbf{a}^{[1]}}$$

$$\frac{\partial \ell}{\partial \mathbf{W}^{[1]}} = \frac{\partial \ell}{\partial \mathbf{a}^{[1]}}\frac{\partial \mathbf{a}^{[1]}}{\partial \mathbf{W}^{[1]}}$$

[1] Using numerator layout, see https://en.wikipedia.org/wiki/Matrix_calculus

Back-propagation

Computation graph

- Computing partial derivatives by hand is tedious and error-prone
- Luckily this can be *automated* by representing a network as a *computation graph* and providing the forward/backward procedures of operators involved
- We can diagram out the computations using a computation graph
  - Nodes represent all the inputs and computed quantities
  - Edges represent which nodes are computed directly as a function of which other nodes

Back-propagation

Computation graph

Example

$$\ell(y, \hat{y}) = \frac{1}{2}\left(\phi(wx + b) - y\right)^2$$

$$z = wx + b$$
$$\hat{y} = \phi(z)$$
$$\ell = \frac{1}{2}(\hat{y} - y)^2$$

Back-propagation

Computation graph

$$
\begin{aligned}
z &= wx + b \\
\hat{y} &= \phi(z) \\
\ell &= \frac{1}{2}(\hat{y} - y)^2
\end{aligned}
$$

Example

$$\ell(y, \hat{y}) = \frac{1}{2}\left(\phi(wx + b) - y\right)^2$$

Forward pass

$$
\begin{tikzcd}
x \arrow[dr] & & y \arrow[dr] & \\
w \arrow[r] & z \arrow[r] & \hat{y} \arrow[r] & \ell \\
b \arrow[ur] & & &
\end{tikzcd}
$$

Back-propagation

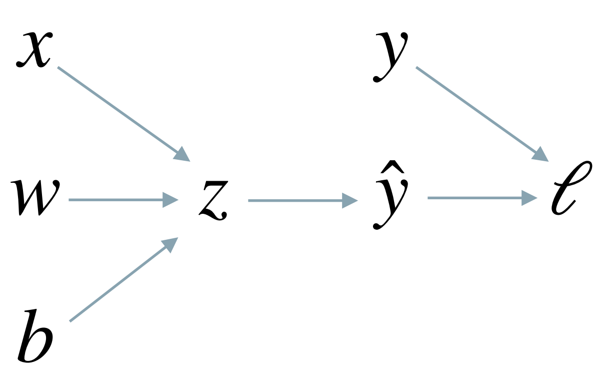
Computation graph

$$z = wx + b$$
$$\hat{y} = \phi(z)$$
$$\ell = \frac{1}{2}(\hat{y} - y)^2$$

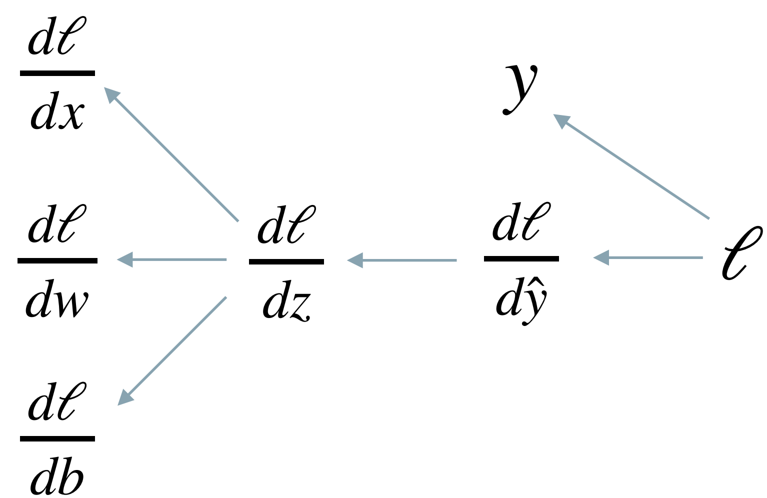
Example

$$\ell(y, \hat{y}) = \frac{1}{2}\left(\phi(wx + b) - y\right)^2$$

Forward pass



Backward pass



Note: updating x if learning embeddings

Back-propagation

Full back-propagation algorithm

Let $v_1, \ldots, v_N$ be a topological ordering of the computation graph (i.e. parents come before children), and v_N denotes the variable we're trying to compute derivatives (e.g. loss)

forward pass

For $i = 1, \ldots, N$

Compute v_i as a function of $\mathrm{Pa}(v_i)$

backward pass

$\overline{v_N} = 1$

For $i = N - 1, \ldots, 1$

$$\overline{v_i} = \sum_{j \in \mathrm{Ch}(v_i)} \overline{v_j} \frac{\partial v_j}{\partial v_i}$$

Automatic differentiation with PyTorch

Calculus is hard! Use Autograd tools such as PyTorch (more in this week's lab), TensorFlow, JAX

```
import torch
x = torch.tensor([3., 2., 1.], requires_grad=False)
y = torch.tensor([0.7], requires_grad=False)

w = torch.tensor([1., 2., 3.], requires_grad=True)
b = torch.tensor([1.], requires_grad=True)
y_hat = torch.dot(w, x) + b

L = 1/2 * (y_hat - y)**2
L.backward()

print(w.grad)
print(b.grad)
```

```
tensor([30.9000, 20.6000, 10.3000])
tensor([10.3000])
```

Back-propagation

PyTorch example

```
import torch
import torch.nn as nn
from sklearn.datasets import load_iris

# Get classification dataset
X_raw, y_raw = load_iris(return_X_y=True)
X = torch.tensor(X_raw, dtype=torch.float32)
y = torch.tensor(y_raw, dtype=torch.long)

loss_fn = nn.CrossEntropyLoss()  # Setup loss function

# Define layers in our MLP
linear1 = nn.Linear(in_features=X.shape[1], out_features=10)
linear2 = nn.Linear(in_features=10, out_features=int(max(y_raw))+1)

# Define optmiser and give it layer parameters
all_params = list(linear1.parameters()) + list(linear2.parameters())
optimiser = torch.optim.SGD(all_params, lr=0.001)

# Run the MLP forwards
h = nn.functional.relu(linear1(X))
out_logits = linear2(h)

loss = loss_fn(out_logits, y)  # Calculate loss

optimiser.zero_grad()  # Zero out the stored gradients: important don't forget this
loss.backward()        # Compute gradients (Backpropagation)
optimiser.step()       # Do SGD step
```

Outline

- Neural networks & why NN
- Feedforward neural network
  - From logistic regression to Feedforward NN
  - Non-linear activation functions
- Back-propagation
  - Gradient descent
  - Back-propagation essentials
  - Computation graph
- Optimisers
  - Stochastic gradient descent
  - SGD with momentum & Adam

Stochastic gradient descent (SGD)

- Standard gradient descent
  - Gradients are computed on the loss of the entire training dataset
  - Slow if the dataset is very large
- Mini-batch SGD
  - At each update we randomly sample a batch of B data points and compute gradients only of the loss on these data points
  - The batch size B is a hyper-parameter
  - Usually B is one of 32, 64, …, 512
  - SGD is much faster to compute per step
  - SGD acts as a regulariser: models trained with SGD usually generalise better

Optimisers

Popular optimisers

- Several improvements can be made to the basic stochastic gradient descent optimisation algorithm
- Two popular optimisers are *SGD with momentum* and *Adam*

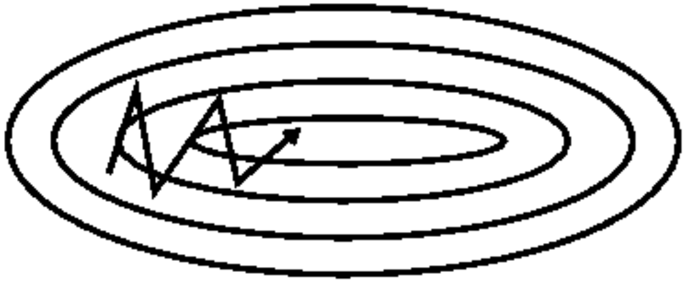
SGD with momentum

- Momentum is an exponentially decaying moving average of gradients from previous batches
- Momentum helps traverse gullies quicker, and can drastically reduce the number of steps needed for training.
- Update rule: $\boldsymbol{\nu}_t \leftarrow \alpha\boldsymbol{\nu}_{t-1} - \eta\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t)$

  where $\boldsymbol{\theta}_{t+1} \leftarrow \boldsymbol{\theta}_t + \boldsymbol{\nu}_t$

  - $\boldsymbol{\nu}$ is the momentum, initialised to $\mathbf{0}$.
  - $\alpha \in [0, 1]$ is the momentum coefficient, a hyper-parameter determining how quickly the previous gradients terms in $\boldsymbol{\nu}$ decay. Typically values: 0.5, 0.9, 0.99.
  - η is the learning rate, $\boldsymbol{\theta}$ the parameters, and $J(\boldsymbol{\theta})$ the cost function.

SGD without momentum



SGD with momentum

Optimisers

Adam (Adaptive moment estimation)

- Adam keeps a running average of gradients and variances of gradients and uses them to normalise changes
- Adam is less sensitive to the learning rate that SGD with momentum
- In practice, Adam often converges in fewer iterations than SGD with momentum (Though this is not always the case).

Learning rate schedules

- Change the learning rate throughout training
- Exponentially decaying schedule multiplies learning rate by some $\lambda \in (0, 1)$ after every step
- Allows for larger learning rate to be used initially, while still converging.

Summary

- Neural networks & why NN
- Feedforward neural network
  - From logistic regression to Feedforward NN
  - Non-linear activation functions
- Back-propagation
  - Gradient descent
  - Back-propagation essentials
  - Computation graph
- Optimisers
  - Stochastic gradient descent
  - SGD with momentum & Adam

Reference

- Chapter 7, Speech and Language Processing