Introduction to TensorFlow and Deep Learning

Lecture 2: Neural Networks

IADS Summer School, 1st August 2022

Dr Michael Fairbank

University of Essex, UK

Email: m.fairbank@essex.ac.uk

Lecture Outline

- Summary so far:
  - TensorFlow basics
  - AutoDiff
  - Gradient Descent to solve a quadratic minimisation problem
- This Lecture (11.00am-12.30pm):
  - Feedforward neural networks
  - Training a network
    - (Using XOR benchmark problem)
  - Tricks for improving network training
    - Weight initialisation
    - Input data preparation
    - Choice of Activation Function
    - One-hot encoding and Cross Entropy loss function

Deep learning vs. Neural Networks

- Deep Learning = deep neural networks
  - Deep learning can be considered a re-branding of neural networks
  - Rebranded since neural networks started working much better
- Reasons for neural networks improving:
  - More training data
  - More processing power
  - Clever neural-network architectures
    - CNNs, Dropout, LSTM nodes, Transformers
  - Better activation functions and weight initialisations
    - ReLu. Xavier initialisation
  - Better learning algorithms (e.g. Adam)

What is a neural network?

A 3-4-2 feedforward neural network:

Takes in 3 input values

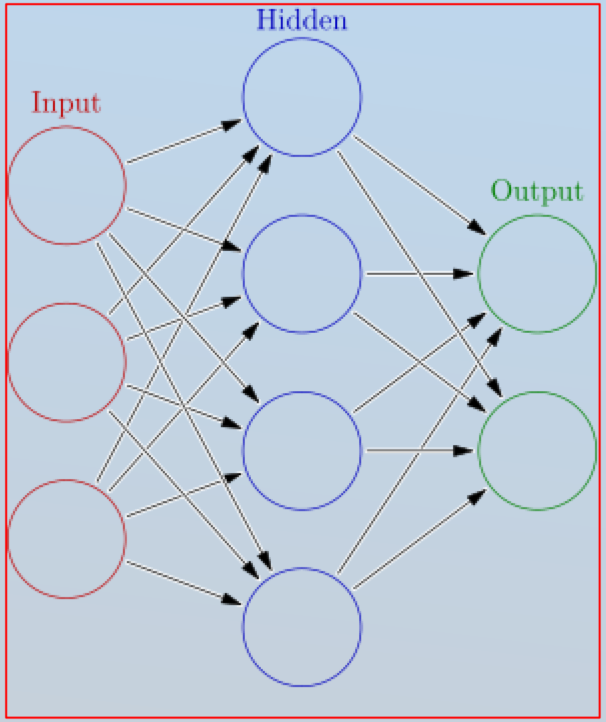
- a vector of length 3

Processes it

- hidden nodes “activate” with variable intensity
- output nodes “activate” with variable intensity
- Arrows are “weights” which amplify or reduce signals

Produces an output vector

- a vector of length 2 here



What is a neural network?

A feedforward neural network:

$$y = f(W, x)$$

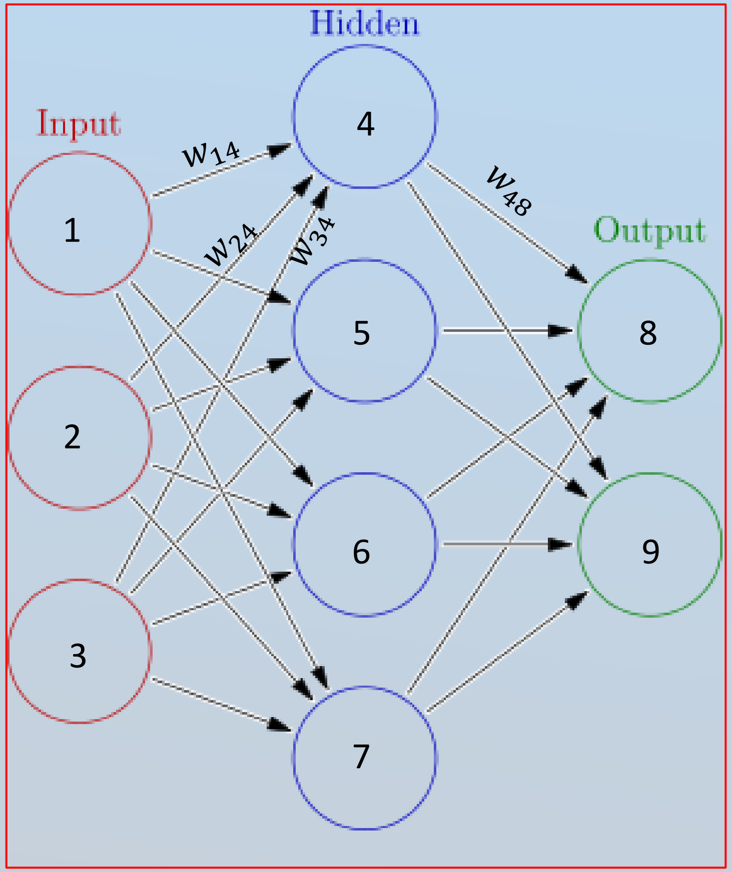
- W=all “weights”, e.g. a list of tensors
- Give it enough weights, each appropriately set, and the neural network can represent any function.
  - “Universal function approximator”
  - E.g. could be a vision system
    - input=grid of pixel intensities of image
    - Output = classification of what the image is (car, house, etc)
  - Could be a language translation system
    - Input = vector of words in an English sentence
    - Output = vector of words in an French sentence

Feedforward algorithm

Number the nodes.

Label the weights:

w_{ij} connects node i to node j



Feedforward algorithm

The input vector to this network must be of dimension 3

- Input vector $\vec{x} = (x_1, x_2, x_3)$

The "electrical" signals passing along the wires to node 4 therefore total to:

$$s_4 = x_1 w_{14} + x_2 w_{24} + x_3 w_{34}$$

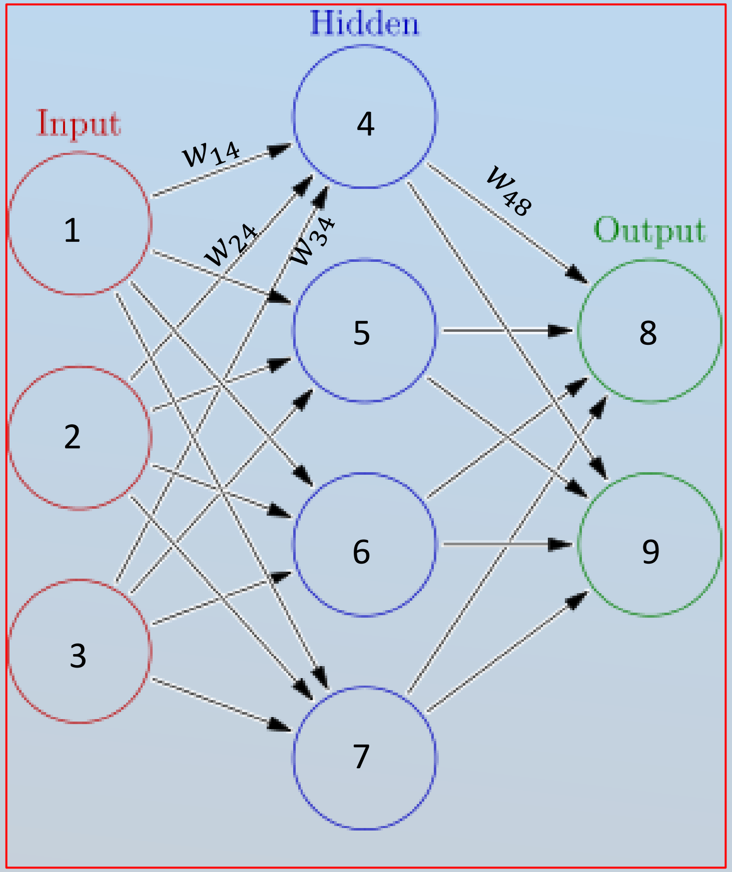
Traditionally, node 4 will "fire" if s_4 is bigger than some threshold, b_4, then

"Activation of node 4"

$$a_4 = \begin{cases} 1 & \text{if } s_4 > b_4 \\ 0 & \text{if } s_4 \le b_4 \end{cases}$$

Equivalently, $a_4 = g(x_1 w_{14} + x_2 w_{24} + x_3 w_{34} - b_4)$

where $g(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x \le 0 \end{cases}$ is called the "activation function" or "nonlinearity"

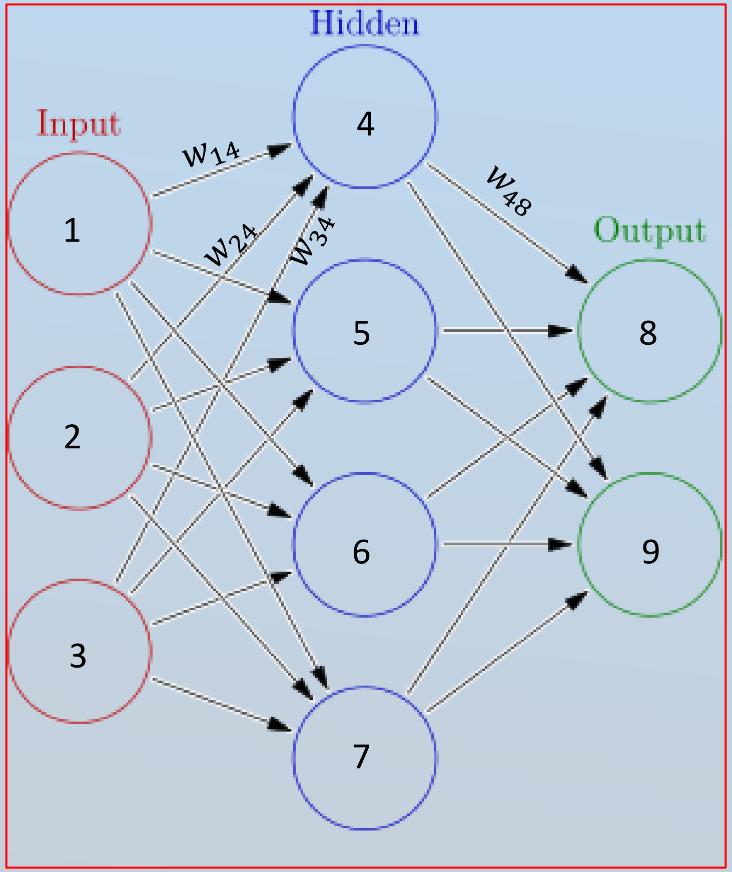


Feedforward algorithm

$$a_4 = g(x_1 w_{14} + x_2 w_{24} + x_3 w_{34} - b_4)$$
$$a_5 = g(x_1 w_{15} + x_2 w_{25} + x_3 w_{35} - b_5)$$
$$a_6 = g(x_1 w_{16} + x_2 w_{26} + x_3 w_{36} - b_6)$$
$$a_7 = g(x_1 w_{17} + x_2 w_{27} + x_3 w_{37} - b_7)$$

More concisely:

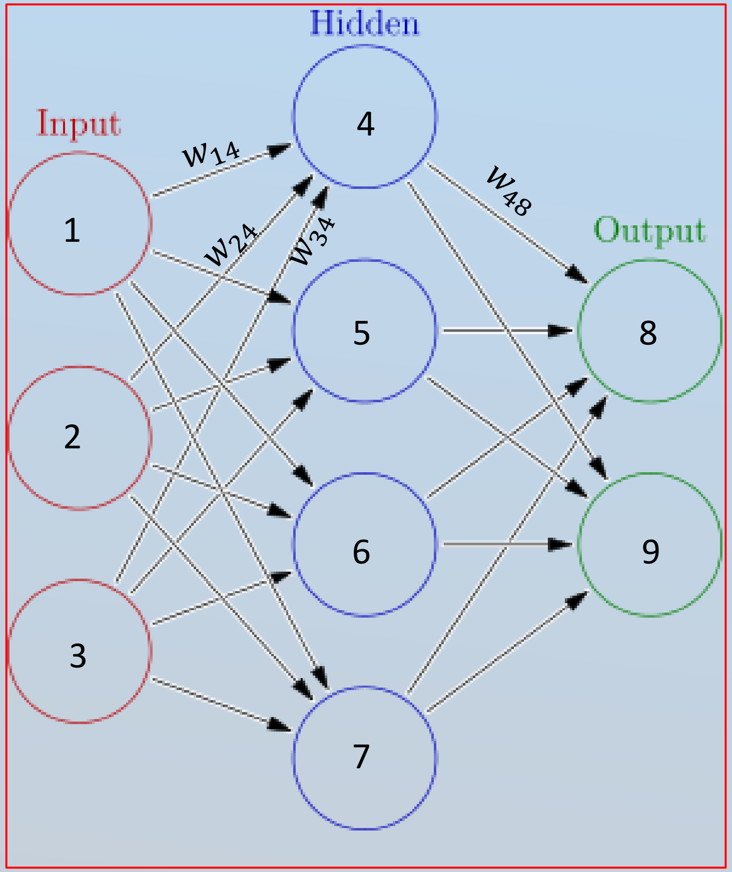
$$\begin{pmatrix} a_4 & a_5 & a_6 & a_7 \end{pmatrix} = g\left(\begin{pmatrix} x_1 & x_2 & x_3 \end{pmatrix} \begin{pmatrix} w_{14} & w_{15} & w_{16} & w_{17} \\ w_{24} & w_{25} & w_{26} & w_{27} \\ w_{34} & w_{35} & w_{36} & w_{37} \end{pmatrix} - \begin{pmatrix} b_4 & b_5 & b_6 & b_7 \end{pmatrix} \right)$$



Feedforward algorithm

And for the output layer :

$$(a_8 \quad a_9) = g\left((a_4 \quad a_5 \quad a_6 \quad a_7)\begin{pmatrix} w_{48} & w_{49} \\ w_{58} & w_{59} \\ w_{68} & w_{69} \\ w_{78} & w_{79} \end{pmatrix} - (b_8 \quad b_9) \right)$$

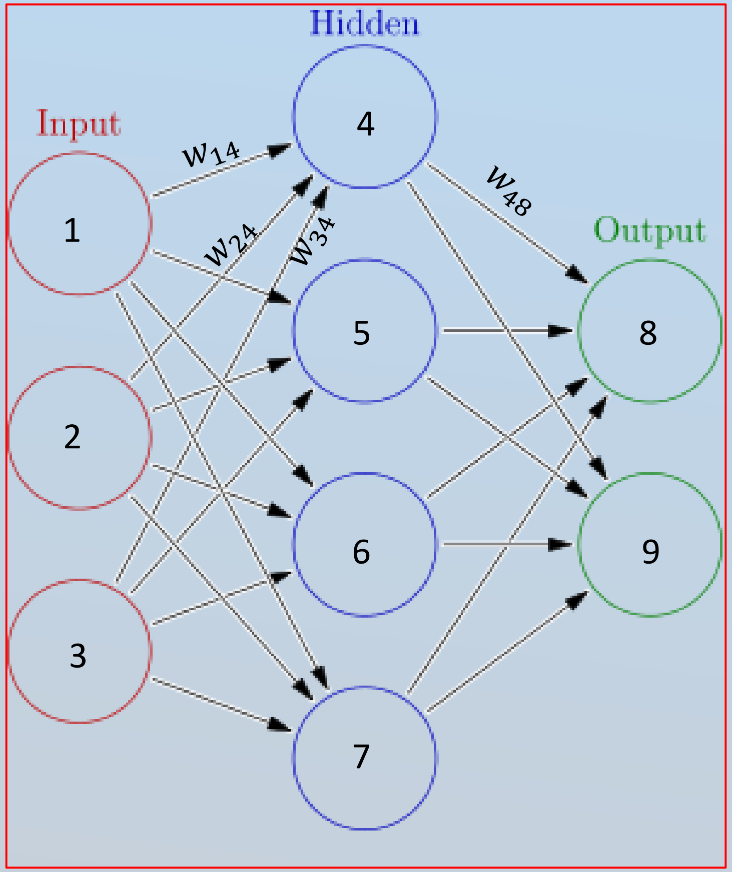


Feedforward algorithm

Together:

$$(a_4 \quad a_5 \quad a_6 \quad a_7) = g\left((x_1 \quad x_2 \quad x_3)\begin{pmatrix} w_{14} & w_{15} & w_{16} & w_{17} \\ w_{24} & w_{25} & w_{26} & w_{27} \\ w_{34} & w_{35} & w_{36} & w_{37} \end{pmatrix} - (b_4 \quad b_5 \quad b_6 \quad b_7)\right)$$

$$(a_8 \quad a_9) = g\left((a_4 \quad a_5 \quad a_6 \quad a_7)\begin{pmatrix} w_{48} & w_{49} \\ w_{58} & w_{59} \\ w_{68} & w_{69} \\ w_{78} & w_{79} \end{pmatrix} - (b_8 \quad b_9)\right)$$



By “Training” the neural network, we aim to find values of all the w_{ij} and b_i values so that the neural network behaves as we want it to.

Feedforward algorithm

More concisely:

Hidden layer: $\overrightarrow{h1} = g\left(\vec{x}W1 + \overrightarrow{b1}\right)$, where

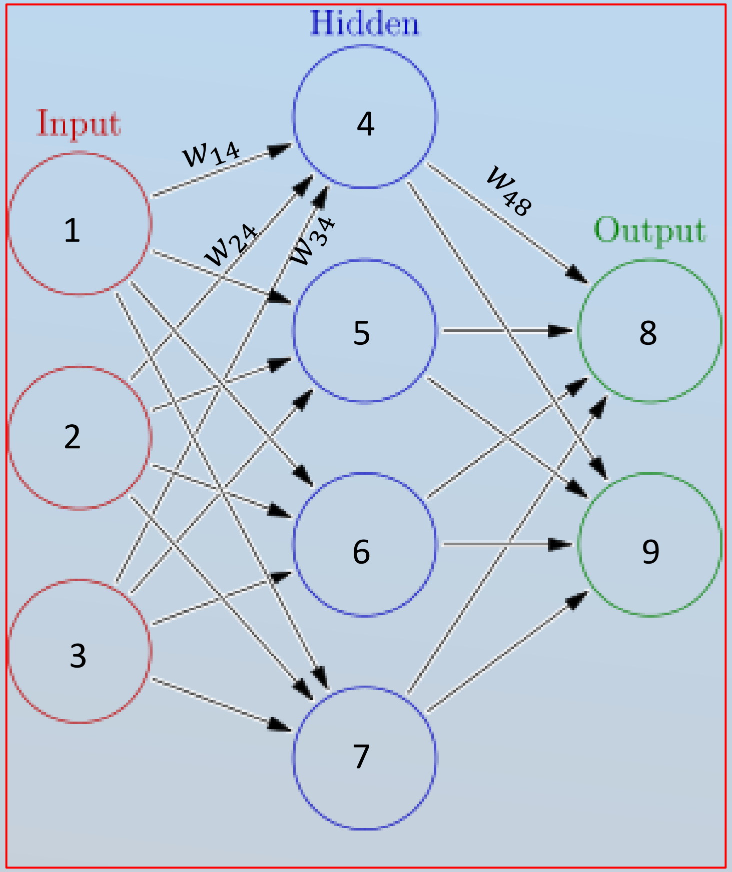
$$\overrightarrow{h1} = (a_4 \quad a_5 \quad a_6 \quad a_7)$$

$$W1 = \begin{pmatrix} w_{14} & w_{15} & w_{16} & w_{17} \\ w_{24} & w_{25} & w_{26} & w_{27} \\ w_{34} & w_{35} & w_{36} & w_{37} \end{pmatrix}$$

$$\vec{x} = (x_1 \quad x_2 \quad x_3)$$

Note: moved minus sign

$$\overrightarrow{b1} = -(b_4 \quad b_5 \quad b_6 \quad b_7)$$



Output layer: $\vec{y} = g(\overrightarrow{h1}W2 + \overrightarrow{b2})$, where

$$\vec{y} = (a_8 \quad a_9)$$

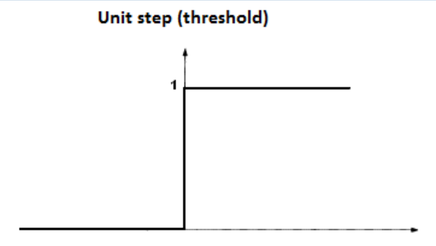
$$W2 = \begin{pmatrix} w_{48} & w_{49} \\ w_{58} & w_{59} \\ w_{68} & w_{69} \\ w_{78} & w_{79} \end{pmatrix}$$

$$\overrightarrow{b2} = -(b_8 \quad b_9)$$

Feedforward algorithm

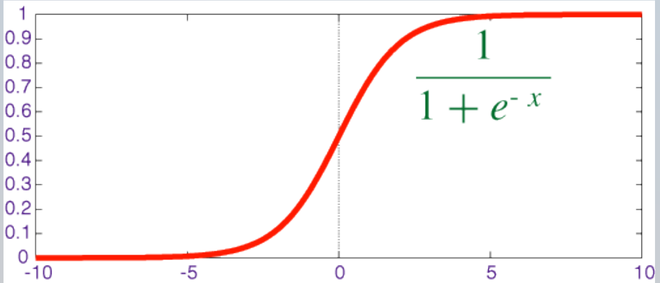
Reminder: The activation function is

$$g(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases}$$

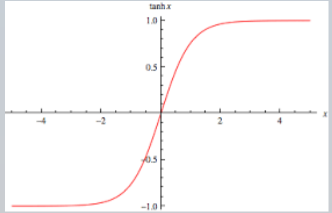


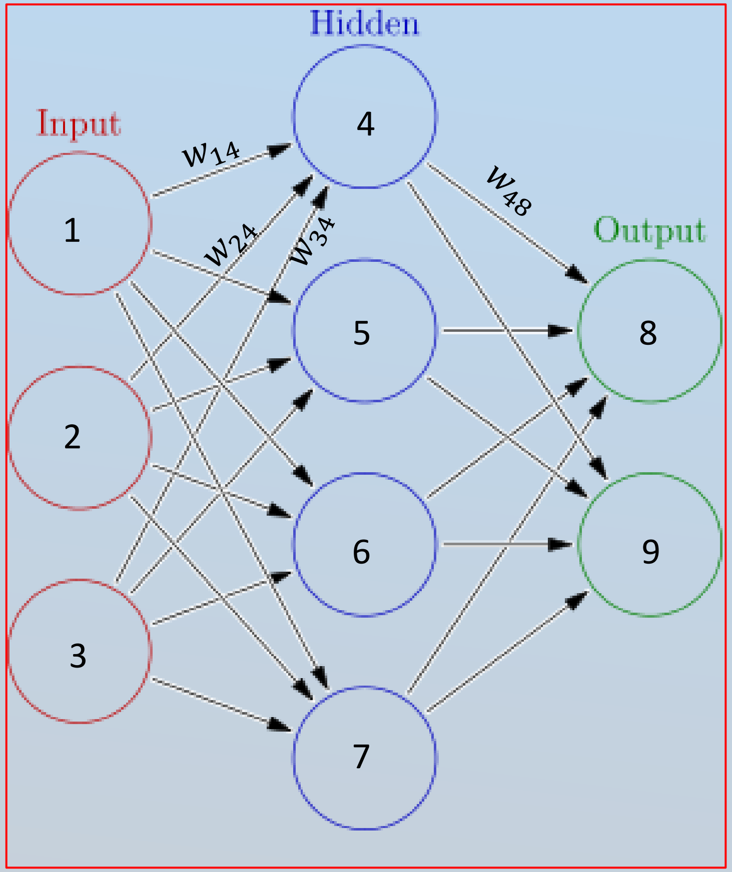
This is not suited for gradient descent (not differentiable)

Hence we change it to the logistic sigmoid function:



or any other non-linear, increasing, smooth function (e.g. tanh(x)).





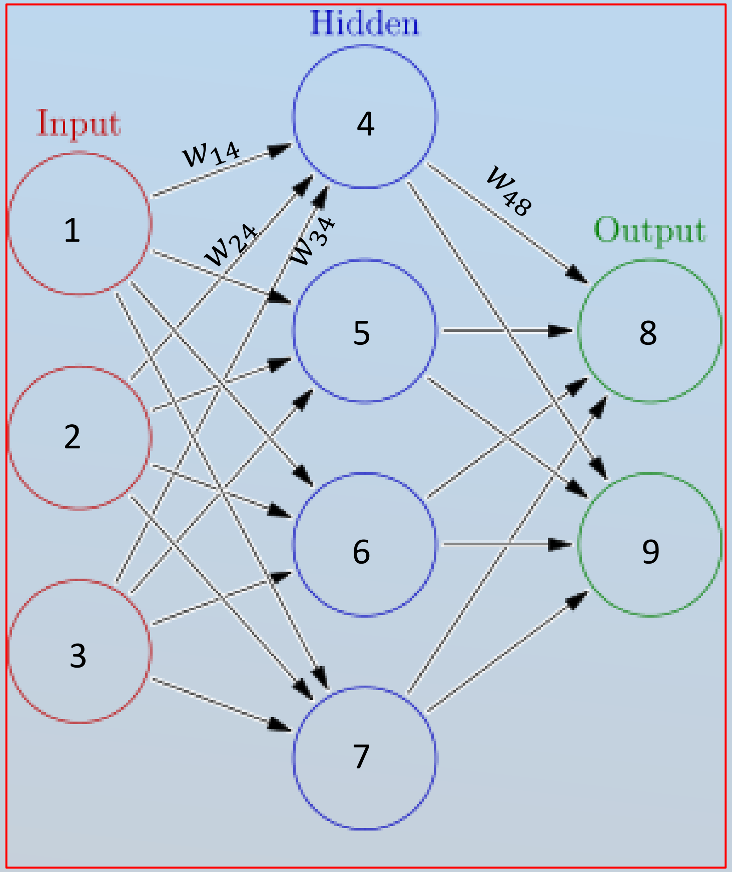
Feedforward algorithm

Summary, in TensorFlow code

For input row vector x:

```
h1=tf.sigmoid(tf.matmul(x,W1) + b1)
y=tf.sigmoid(tf.matmul(h1,W2) + b2)
```

A neural network coded in just 2 lines!



If we wanted a second hidden layer:

```
h1=tf.sigmoid(tf.matmul(x,W1) + b1)
h2=tf.sigmoid(tf.matmul(h1,W2) + b2)
y=tf.sigmoid(tf.matmul(h2,W3) + b3)
```

Note: for tf.matmul(…) to work, both arguments must be equal rank.

Therefore x must be rank 2

E.g. x has shape [1,m]   (i.e. a rank 2 row vector)

m=number of input nodes to network

Feedforward algorithm

Summary, in TensorFlow code

For input row vector x:

```
h1=tf.sigmoid(tf.matmul(x,W1) + b1)
y=tf.sigmoid(tf.matmul(h1,W2) + b2)
```

For the network shown, x has shape [1,3]

What shape does W1 have?

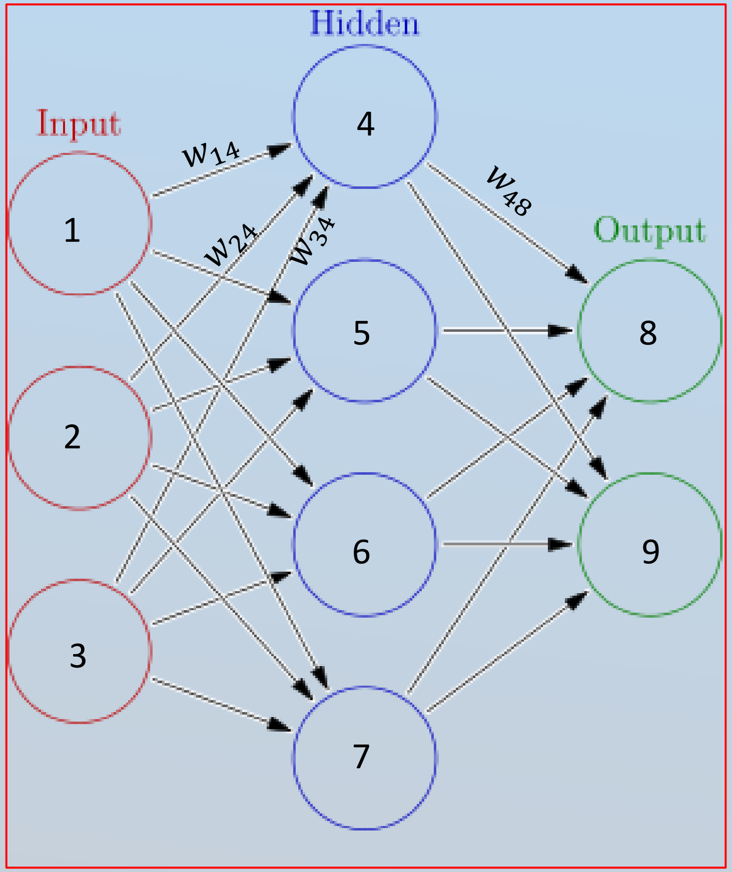
What shape does W2 have?

What shape does b1 have?

What shape does b2 have?

What shape does h1 have?

What shape does y have?



x:[1,3]

W1:[3,4]

b1:[1,4]

h1:[1,4]

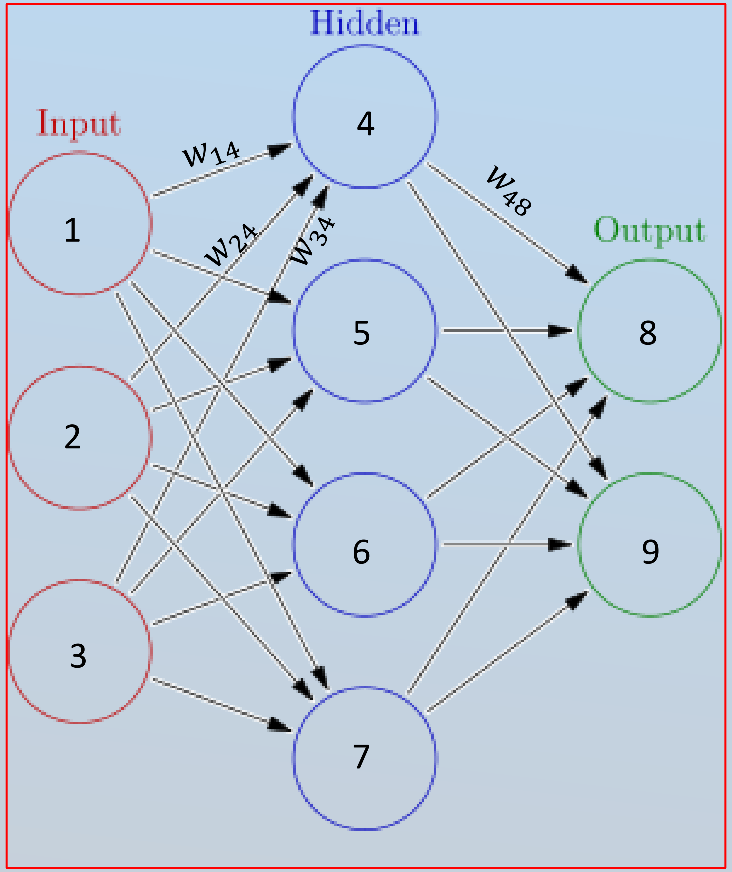
W2:[4,2]

b2:[1,2]

y:[1,2]

Feedforward algorithm

The matrices W1, W2 are called the "weight matrices" or "kernels".



Feedforward algorithm

Exercise 1: (Template code is on next slide and in lecture2-notebook-ffnn)

1. Create a 2-2-1 feedforward network. (One hidden layer, of just 2 nodes).
2. Use sigmoid activation functions at each layer. tf.sigmoid(....)
3. Randomise weights and biases.

> E.g. to set W1 to a random 4 by 5 matrix, use
> W1=tf.Variable(tf.random.truncated_normal([4,5], stddev=0.1))

However, to ensure we all get the same results, we will set the random seed.

> E.g. W1=tf.Variable(tf.random.truncated_normal([4,5], stddev=0.1, seed=1))

4. Then evaluate your network with input vector $\vec{x} = [[1,1]]$. Check this gives an output of $\vec{y} = [[0.50787264]]$

(assuming all of our machines and python + TensorFlow versions have consistent random number generator algorithms)

Feedforward algorithm

Template script for exercise. Replace green # signs.

```
import tensorflow as tf

x=tf.constant([[1,1]], tf.float32) # our input vector

# Build our random weight and bias matrices, of appropriate shapes
W1 = tf.Variable(tf.random.truncated_normal([#,#], stddev=0.1, seed=1))
b1 = tf.Variable(tf.random.truncated_normal([#,#], stddev=0.1, seed=2))
W2 = tf.Variable(tf.random.truncated_normal([#,#], stddev=0.1, seed=3))
b2 = tf.Variable(tf.random.truncated_normal([#,#], stddev=0.1, seed=4))

# define our feed-forward neural network here:
def run_network(x):
        h1=#TODO
        y=#TODO
        return y

print(run_network(x).numpy())
```

This script is in lecture2-notebook-ffnn.ipynb under "Exercise 1". Complete it now.

Randomising initial weights

- We need to randomise weights in a neural network before training
  - Symmetry breaking for gradient descent
  - Randomised start point for gradient descent

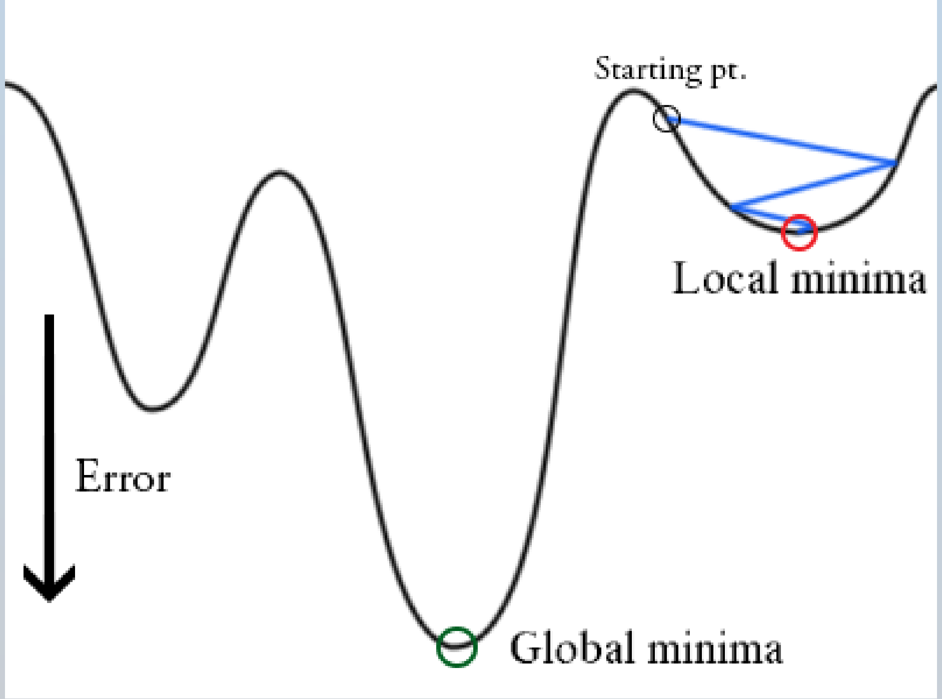


Image source: https://static.thinkingandcomputing.com/2014/03/bprop.png

Randomising initial weights

- We normally wouldn’t set the random seed like we did
  - might need multiple starts from different random weights, to try to avoid local minima
- The magnitude you choose for the initial random weights is very important
  - This is one of the big breakthroughs in deep learning. (More later…)

Training the network

- To train the network we need to choose the weights and bias tensors so as to make the network behave as desired.

- Let’s make it learn the XOR function.

XOR truth table

Input		Output
A	B	
0	0	0
0	1	1
1	0	1
1	1	0

4 input vectors

4 target output vectors

Training the network

- We can modify our feedforward network to push through all 4 input vectors at once:
- Change `x=tf.constant([[1,1]], tf.float32)`

  To `x=tf.constant([[0,0],[0,1],[1,0],[1,1]], tf.float32)`
- The equations still work fine:

  h1=tf.sigmoid(tf.matmul(x,W1) + b1)
  y=tf.sigmoid(tf.matmul(h1,W2) + b2)
- Exercise 2: Try this modification to your program.
  - Use the jupyter notebook template under "Exercise 2"
- Broadcasting is used for the "+b1" and "+b2"
- Our output tensor will now have shape [4,1]

XOR truth table

Input		Output
A	B	
0	0	0
0	1	1
1	0	1
1	1	0

Training the network

- Our network outputs $\vec{y} = \begin{pmatrix} 0.509233 \\ 0.509222 \\ 0.507883 \\ 0.507872 \end{pmatrix}$. However our targets were $\vec{t} = \begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \end{pmatrix}$

  Assuming our RNG seeds behave the same.
  Note: Non-deterministic behaviour in Jupyter; need to restart jupyter kernel to reset tensorflow's random state.

- The squared error between these is
  - $L = \|\vec{y} - \vec{t}\|^2$
- We train the network by doing gradient descent on this

$$w \leftarrow w - \eta \frac{\partial L}{\partial w}$$

Where *w* here represents *all* weights and biases

w=(W1,W2,b1,b2)

Training the network

$$L = \left\| \vec{y} - \vec{t} \right\|^2$$

$$w \leftarrow w - \eta \frac{\partial L}{\partial w}$$

Notes:

1. L is our "loss function", or "error function" or "cost function"
2. Tensorflow will calculate these partial derivatives $\frac{\partial L}{\partial w}$ for us using Autodiff (backpropagation)
3. Tensorflow will implement gradient descent $w \leftarrow w - \eta \frac{\partial L}{\partial w}$ using its canned optimizer
4. First we need to define L in terms of tensorflow code

Training the network

$$L = \left\|\vec{y} - \vec{t}\right\|^2$$

Define L in terms of TensorFlow code:

$\|A\|^2$ means sum and square every component of A

$$L = \left\|\vec{y} - \vec{t}\right\|^2$$

```
deltas=tf.subtract(y, y_labels)
squared_deltas=tf.square(deltas)
loss=tf.reduce_sum(squared_deltas)
```

Q: Why is this function called “reduce”…?

Exercise 3: Train your network

1. Define your constant 4*1 tensor *y_labels*
   - *Make sure it is shape [4,1] not shape [1,4]*
   - *Think about the nesting of your square brackets to achieve this.*
2. Define your loss function

```
def calc_loss():
        y=run_network(x)
        deltas=tf.subtract(y, y_labels)
        squared_deltas=tf.square(deltas)
        loss=tf.reduce_sum(squared_deltas)
        return loss
```

XOR truth table

Input		Output
A	B	
0	0	0
0	1	1
1	0	1
1	1	0

Exercise 3: Train your network

3. Train your network in a loop:

```
optimizer = tf.keras.optimizers.SGD(0.5)
for i in range(20000):
	optimizer.minimize(calc_loss, [W1,b1,W2,b2])
	if (i%1000)==0:
		print("iteration ",i," loss", calc_loss().numpy())
print(run_network(x).numpy())
```

Make your changes in the jupyter notebook under “Exercise 3”

XOR truth table

Input		Output
A	B	
0	0	0
0	1	1
1	0	1
1	1	0

Gotchas from this exercise:

Check targets and y are the same shape, i.e. both 4*1 and not 1*4.

(If you get this wrong then “deltas” will acquire size 4*4 (by broadcasting), and reduce_sum will give a strange result… and during training “loss” will never go below 4)

Exercise 3: Train your network

Review: Train your network

- Note that these 3 lines

```
optimizer = tf.keras.optimizers.SGD(eta)
for i in range(20000):
        optimizer.minimize(calc_loss, [W1,b1,W2,b2])
```

- were shorthand for:

```
for i in range(20000):
        with tf.GradientTape() as tape:
                L=calc_loss()
        [dLdW1,dLdW2,dLdb1,dLdb2]=tape.gradient(L, [W1,W2,b1,b2])
        W1.assign(W1-dLdW1*eta)
        W2.assign(W2-dLdW2*eta)
        b1.assign(b1-dLdb1*eta)
        b2.assign(b2-dLdb2*eta)
```

- …and the "tape.gradient" line is itself a massive shorthand
    - for the backpropagation derivative calculation…

XOR + Universal Function approximation

We’ve solved XOR. This is a tiny network. Modern deep learning networks are tens or hundreds of layers deep.

XOR relates to the universal function approximation capabilities of neural networks.

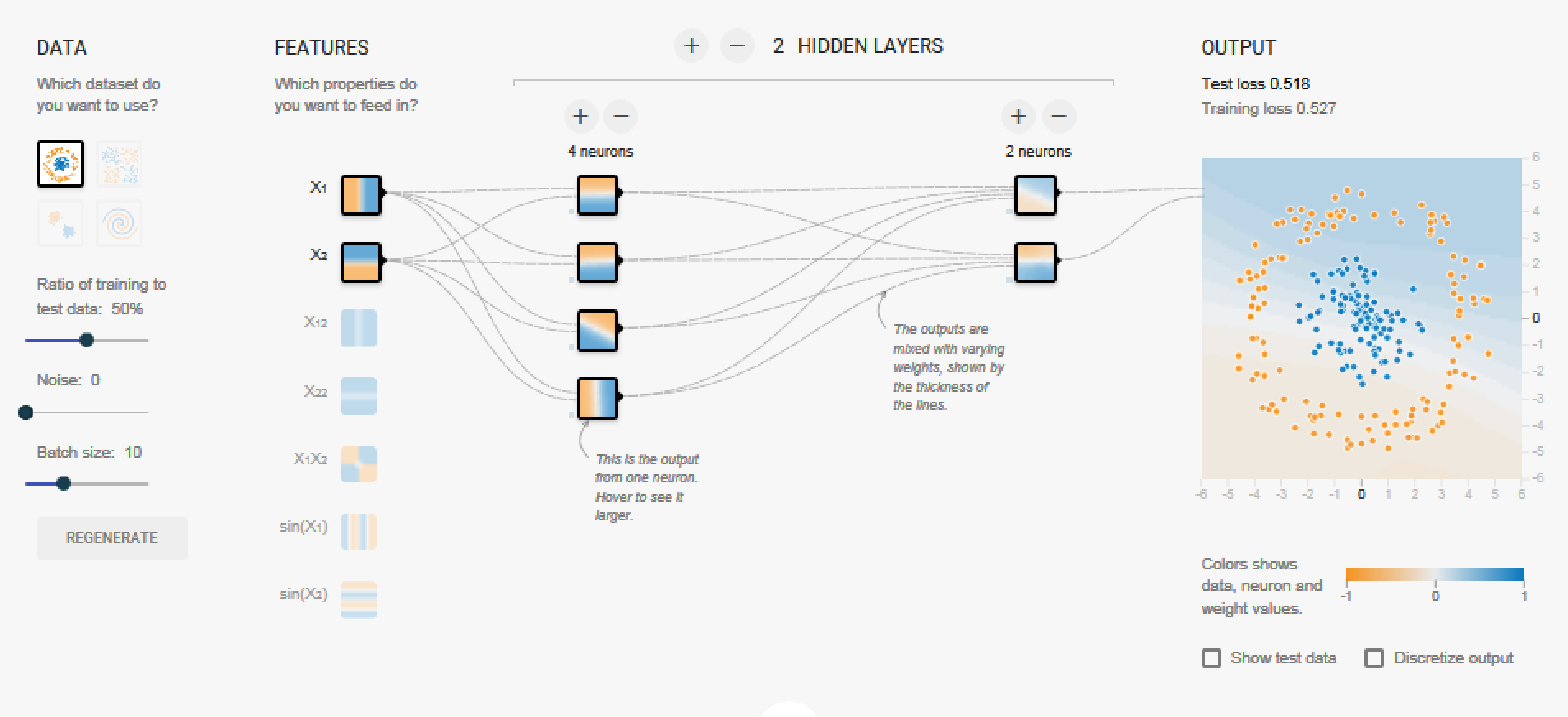
XOR truth table

Input		Output
A	B	
0	0	0
0	1	1
1	0	1
1	1	0

TensorFlow playground website

Interesting website for building intuition of what a neural network is:

https://playground.tensorflow.org/



Running the neural network on new data

- A “pattern” is a pairing of input and output vectors
  - Our network has learned 4 “patterns”.
- What if we wanted to input a new pattern? E.g. input (0.5,0.5)
  - How would we rewrite our TensorFlow code?

XOR truth table

Input		Output
A	B	
0	0	0
0	1	1
1	0	1
1	1	0

Running the neural network on new data

Exercise: try this.

Then we can evaluate our network on a new pattern, e.g.

```
print(run_network ( [[0.5,0.5]]).numpy())
```

XOR truth table

Input		Output
A	B	
0	0	0
0	1	1
1	0	1
1	1	0

Feedforward neural nets: Higher level functions

To create two network layers, instead of:

```
W1 = tf.Variable(tf.random.truncated_normal([2,2], stddev=0.11))
b1 = tf.Variable(tf.random.truncated_normal([1,2], stddev=0.1))
W2 = tf.Variable(tf.random.truncated_normal([2,1], stddev=0.1))
b2 = tf.Variable(tf.random.truncated_normal([1,1], stddev=0.1))
def run_network(x):
		h1=tf.sigmoid(tf.matmul(x,W1) + b1)
		y=tf.sigmoid(tf.matmul(h1,W2) + b2)
		return y
```

- “layer1” and “layer2” act as callable functions
- You can access the weights and biases for a layer as
  - “layer1.trainable_variables”, which evaluates as [W1,b1]
  - Note: You must run the network with an input tensor x before accessing trainable_variables though

We can just do:

```
layer1=tf.keras.layers.Dense(2, activation=tf.nn.sigmoid)
layer2=tf.keras.layers.Dense(1, activation=tf.nn.sigmoid)
def run_network(x):
		h1=layer1(x)
		y=layer2(h1)
		return y
```

This will create all 4 Variable weight and bias tensors for us, and randomise them appropriately

Feedforward neural nets: Higher level functions

Furthermore, instead of:

```
layer1=tf.keras.layers.Dense(2, activation=tf.nn.sigmoid)
layer2=tf.keras.layers.Dense(1, activation=tf.nn.sigmoid)
def run_network(x):
        h1=layer1(x)
        y=layer2(h1)
        return y
```

We can just do:

```
layer1=tf.keras.layers.Dense(2, activation=tf.nn.sigmoid)
layer2=tf.keras.layers.Dense(1, activation=tf.nn.sigmoid)
model = tf.keras.Sequential([layer1,layer2])
def run_network(x):
        return model(x)
```

model.trainable_variables will give us [W1,b1,W2,b2]

Exercise 4 (Optional): rewrite your code to use one of these 2 methods.

Limitations of XOR

To get deeper networks trained, we need

better weight initialisation,

better activation functions,

more training data

more CPU time.

Challenge: Enhance this code

Try to put the key line which computes gradients, i.e.,

```
optimizer = tf.keras.optimizers.SGD(0.5)
for i in range(20000):
	optimizer.minimize(calc_loss, [W1,b1,W2,b2])
	if (i%1000)==0:
		print("iteration ",i," loss", calc_loss().numpy())
print(run_network(x).numpy())
```

Into a separate python function which has the @tf.function annotation

Q: Why do this?

Tricks for improving network training

Input pre-processing

Neural networks work best when input vectors are normalised to have variance 1.

So for each real-valued column of your dataset input tensor, subtract off column means…

```
import numpy as np
inputs=[[0.0,0],[0,10],[10,0],[10,1]]
column_means=np.mean(inputs, axis=0)
inputs-=column_means
```

and then divide each column by column standard deviation.

```
column_stds=np.std(inputs, axis=0)
inputs/=column_stds
```

Don’t forget to do the same to any test data you later feed the neural network

(carefully record the above column_means and column_stds for this purpose)

This slide is really important:

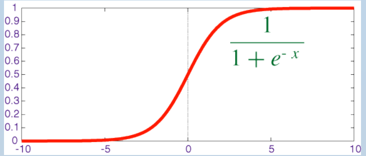
- Normalise your inputs.
- Beginners forget this step

For more sophisticated pre-processing, see whitening

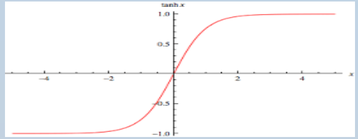
Better Activation functions

In increasing order of usefulness for deep learning:

1. Logistic Sigmoid function. $g(x)=\frac{1}{1+e^{-x}}$. tf.sigmoid(X)

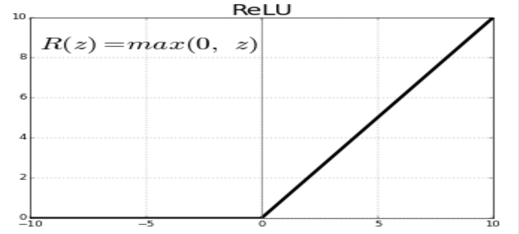


2. Tanh function. $g(x)=\tanh(x)$. tf.tanh(X)



3. ReLu (Rectified Linear)function.

$$g(x)=\begin{cases} x & \text{if } x>0 \\ 0 & \text{if } x\leq 0 \end{cases}.$$ tf.nn.relu(X)



ReLu is now the most commonly used activation function for training deep neural networks. It has nice properties in that learning gradients are less likely to decay / explode.

See https://en.wikipedia.org/wiki/Rectifier_(neural_networks)

https://en.wikipedia.org/wiki/Vanishing_gradient_problem

https://www.tensorflow.org/api_docs/python/tf/nn/relu

Better Weight Initialisation

We need random initialised weights to help gradient descent, and break symmetry. The magnitudes of those initial random weights is important, to try to prevent vanishing / exploding learning gradients.

Glorot/Xavier initialisation: Set the standard-deviation of for a weight matrix of dimension $n \times m$

$$\sigma = \sqrt{\frac{2}{n+m}}$$

The above formula varies depending on the choice of:

- Activation function used
- Whether you used normal distribution or Gaussian randomisation
- The exact research you follow

Source:

Glorot, Xavier, and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks." Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics. 2010.

http://www.jmlr.org/proceedings/papers/v9/glorot10a.html

Better Weight Initialisation

Glorot/Xavier initialisation: $\sigma = \sqrt{\frac{2}{n+m}}$

Glorot/Xavier initialisation is easy when using the higher level functions to create your network layers:

```
layer1=tf.keras.layers.Dense(2, activation=tf.nn.sigmoid, kernel_initializer='glorot_uniform')
```

In fact Glorot/Xavier initialisation is default here.

See https://www.tensorflow.org/api_docs/python/tf/keras/initializers/ for further information

Categorical inputs + one-hot encoding

If we have an input to a NN that represents a category, e.g. “profession:”

1=Teacher
2=Doctor
3=Student
4=Unemployed

Since there is no logical ordering of these 4 items numerically, we should not give them as a *single numerical input* to the NN.

Instead use *four different inputs* to the NN, with “one-hot encoding”:

(1,0,0,0)=Teacher
(0,1,0,0)=Doctor
(0,0,1,0)=Student
(0,0,0,1)=Unemployed

Categorical inputs + one-hot encoding

Q. How many inputs should we use for “age of child (1-10)”?

Q. How many inputs should we use for “colour of eyes”?

Q. How many inputs should we use for “position of tennis ball”?

Q. How many outputs should we have for a letter-of-alphabet reading (A-Z) vision system?

Similarly, target outputs corresponding to different categories should be one-hot encoded

You can encode into one-hot encoded via tf.one_hot function

https://www.tensorflow.org/api_docs/python/tf/one_hot

Categorical inputs + one-hot encoding

Q: what is

tf.argmax(tf.constant([[0,0,0,0,1], [0,0,0,1,0]]),axis=1)

If our output tensor is *y* (one-hot encoded) then tf.argmax(y, axis=1) will reverse the one-hot encoding.

If the targets are y_targets (integer class number; so NOT one-hot encoded), then we can calculate the number of correct rows as follows:

```
correct_prediction = tf.equal(tf.argmax(y, axis=1), y_targets)
corrent_count = tf.reduce_sum(tf.cast(correct_prediction, tf.float32))
```

Training Classification Networks

Training Classification Networks

For classification systems, e.g. a vision system, it is usual to try to make the neural network output “probabilities” for each classification possibility.

Suppose we have possible 4 furniture categories: “Chair/Table/Wardrobe/Door”.

Hence we need an NN with 4 outputs.

Suppose the neural output vector is $\vec{y} = (-0.2, 0.5, 0.1, 0.1)$

Which furniture item do you think this means?

How confident are you?

How can we convert this NN output into “probabilities”?

Probabilities must all be positive and must sum to 1

Training Classification Networks

Probabilities must all be positive and must sum to 1

$$\vec{y} = (-0.2, 0.5, 0.1, 0.1)$$

$$\Big\downarrow e^{y_i}$$

$$(0.819,\ 1.649,\ 1.105,\ 1.105)$$ All positive

$$\Big\downarrow \frac{e^{y_i}}{\sum_k e^{y_k}} \text{ (Normalisation)}$$

$$\vec{p} = (0.175,\ 0.352,\ 0.237,\ 0.237)$$ Now all sum to 1.

Summary: $p_i = \frac{e^{y_i}}{\sum_k e^{y_k}}$ (Softmax function)

Q: Does P(Table)=0.352?

Cross-entropy Loss function

Now the neural network outputs produce $\vec{p}$ with $p_i = \frac{e^{y_i}}{\sum_k e^{y_k}}$ (Softmax function)

We want these to eventually be trained to become true probabilities.

To achieve this, we train the NN with "Cross-Entropy Loss Function"

The cross-entropy loss for a single pattern is defined to be

$$L = -\sum_i t_i \log(p_i)$$

where $\vec{t}$ is the one-hot encoded true output classification, and i is the category index.

Cross-entropy Loss function

The cross-entropy loss for a single pattern is defined to be

$$L = -\sum_i t_i \log(p_i)$$

where $\vec{t}$ is the one-hot encoded true output classification, and i is the category index.

E.g. Consider a "table", $\vec{t}$=(0,1,0,0), with actual neural-network output $\vec{y} = (-0.2, 0.5, 0.1, 0.1)$

The category index of "table" is 1.

Hence $t_1 = 1 \quad t_0 = t_2 = t_3 = 0$.

Hence $L = -\log(p_1)$

In this example, when training attempts to minimise L, it will have to minimise $L = -\log(p_1)$, i.e. maximise p_1, i.e. maximise y_1 compared to all of y_0, y_2, y_3

Cross-entropy Loss function

The cross-entropy loss for a single pattern is defined to be

$$L = -\sum_i t_i \log(p_i)$$

where $\vec{t}$ is the one-hot encoded true output classification, and i is the category index.

Where does this formula come from? It can be shown that minimising the above *L* over your whole dataset is the same as *finding the weights which maximise your neural network's estimation of the total probability that your given training dataset members each have their given (fixed) individual category label.*

Example: If you trained it with 60 images of dogs and 40 images of cats, and your neural network could memorise all 100 images exactly, then it should learn to output $P(dog) \approx 1$ for every dog image and $P(cat) \approx 1$ for every single cat image.

- If the neural network is then given a new image of a dog we *hope* that the image is sufficiently similar to the other dog images such that the neural network gives $P(dog) \approx 1$ for that new dog image. But this is not guaranteed of course.

Example 2: if you had 100 identical photos in your training dataset and 40% of them happened to be labelled "1" and the other 60% were labelled "2", then the neural network would have to converge to giving a probability of 40% to category "1".

- In this bizarre example, there is nothing the neural network can do to distinguish category 1 from category 2 (since the training input photos were all identical).

Cross-entropy Loss function

SoftMax in TensorFlow:

```
y=tf.keras.activations.softmax(y, axis=1)
```

Cross entropy in TensorFlow:

```
cce=tf.keras.losses.SparseCategoricalCrossentropy()
cross_entropy = tf.reduce_mean(cce(y_true=train_labels, y_pred=y))
```

- This expects y_pred to have had softmax applied to it on entry.
- Also it expects train_labels to be an integer for the category number (so the label is NOT one-hot encoded)

The variable cross_entropy is returned by our “calc_loss()” function, to optimise through gradient descent

```
optimizer.minimize(calc_loss, trainable_variables)
```

Further Reading (Cross Entropy Loss)

Further reading:

[Where did the Binary Cross-Entropy Loss Function come from? | by Rafay Khan | Towards Data Science]

Also on cross entropy: [The Most Important Function in Machine Learning] (YouTube, P.Solver, 30mins)

Summary: Classification vs. Regression Problems

When you want a neural network to classify something-

- Use SoftMax to obtain probabilities
- Use cross entropy loss function to train
- Use max to pick the NN's favoured category

When you want a neural network to solve a real valued problem (A regression problem):

- Train with sum-of-squared error
- Don't use SoftMax on final layer

Optional Lecture Recap: How does a neural network work?

We have already studied “what a neural network is” now.

But a slick recap, especially on how data is propagated forwards through a network, is given by YouTuber Veritasium: Future Computers Will Be Radically Different – YouTube

Watch from 3m42s to 13m40s

0m00-3m42: Analogue vs digital computers

3m42-8m00: Single-layer neural networks (“Perceptron”)

8m00-10m00: Multi-layer neural networks

10m00-13m40s: Deep neural networks, Image Net