TensorFlow

Lecture 1 - TensorFlow Basics & Gradient Descent

pip install tensorflow

rank n tensor $\Leftrightarrow$ n dimensional array

TensorFlow datatype $\neq$ numpy datatype

```
a = tf.constant(2, tf.float32)
```

print(a.numpy()) casts as a normal float.

Two friendly matrix multiplications! $A^{n \times m}$, $B^{m \times l}$

	Hadamard product	Standard $C^{n \times l}$
maths	$c_{ij} = a_{ij}\, b_{ij}$	$c_{ij} = \sum_{k=1}^{m} a_{ik}\, b_{kj}$
python	tf.multiply(a, b)	tf.matmul(a, b)

Obviously, this is

$A^{n \times r} \times B^{n \times m} \rightarrow C^{n \times m}$

so different matrix dimensions required.

E.g.

```
a = tf.constant([[1,2],[3,4]])
b = tf.constant([[1],[1]])
c = tf.matmul(a, b)
```

$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 3 \\ 7 \end{pmatrix}$

Datatypes

int32, float32

Should always specify datatype when you create a tensor.

tf.cast(a, tf.int32) to cast from one to another.

can't add datatypes that don't match $\Rightarrow$ you must cast

True/False is boolean $\xrightarrow{\text{cast}}$ 1, 0 in tf.int32

Tensor Shape

can't add matrices of different sizes...

But there is 'broadcasting'

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} + 1 = \begin{pmatrix} 2 & 3 \\ 4 & 5 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} + (10, 20) = \begin{pmatrix} 11 & 22 \\ 13 & 24 \end{pmatrix}$$

Elementwise Tensor operations

eg. tf.FUNCTION(a).numpy()

comparisons too tf.greater([1,2,3], [2])

⇒ [false, false, True]

some operator shortcuts

a * b = tf.multiply(a, b)

[Hamard product]

Variables

tf.Variable(___) can be reassigned

x = tf.constant([0.3], tf.float32)

x = tf.constant([-1], tf.float32)

also reassigns...

but need to declare variable type...

W.assign_add / W.assign_sub

analogous to += or -=

Aggregation Functions

function	action
tf.reduce_sum(a)	sums elements
tf.reduce_mean(a)	mean of elements
tf.reduce_max(a)	max of elements
tf.argmax(a)	counts index where max appears

a could be of matrix etc...

Example

$a = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$, float 3

tf.reduce_sum(tf.cast(a>1, tf.float32)).numpy() = 3

if entry bigger than 1, set as true

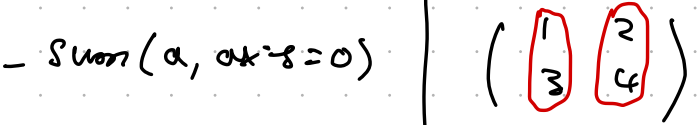
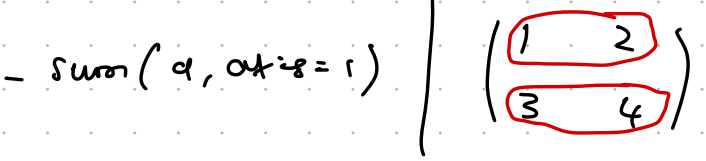
true → cast 1

false → cast 0

sum overall results.

Moving on axis

$a = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$

command	Axis	output
tf.reduce_sum(a, axis=0)		[4 6]
tf.reduce_sum(a, axis=1)		[3 7]

Automatic Differentiation

- NOT numerical or symbolic
  → give code to make f(x). It applies chain rule all the way down.

E.g.

```
x = tf.Variable(3.0, tf.float32)
with tf.GradientTape() as tape:
   y=tf.pow(x,2.0)
dydx=tape.gradient(y, x)
print(dydx.numpy())
6.0
```

$y = x^2$

$\frac{dy}{dx} = 2x$

$\left.\frac{dy}{dx}\right|_{x=3} = 6$

```
x=tf.Variable(4.0,tf.float32)
y=tf.Variable(2.0,tf.float32)
with tf.GradientTape() as tape:
   f=tf.pow(x,2.0)*3.0+y
[dfdx, dfdy]=tape.gradient(f, [x,y])
print(dfdx.numpy(), dfdy.numpy())
```

```
24.0 1.0
```

$f(x,y) = 3x^2 + y$

$\frac{\partial f}{\partial x} = 6x$

$\frac{\partial f}{\partial y} = 1$

Diff not constant here...

```
x=tf.constant(4.0,tf.float32)
with tf.GradientTape() as tape:
   tape.watch(x)
   f=tf.pow(x,3.0)
dfdx=tape.gradient(f, x)
print(dfdx.numpy())
```

```
48.0
```

$x = 4.0$

$f(x) = x^3$

$\frac{df}{dx} = 3x^2$

$\left.\frac{df}{dx}\right|_{x=4} = 3 \times 16 = 30 + 18 = 48$

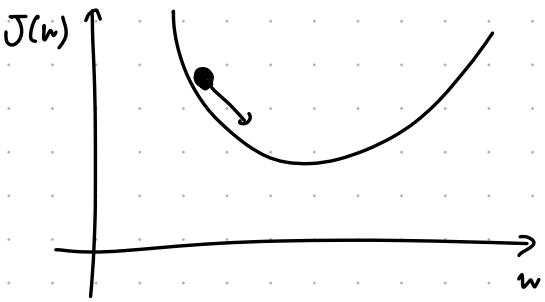
can use auto diff on

$f: \mathbb{R}^{2\times 2} \times \mathbb{R}^2 \times \mathbb{R}^2 \longrightarrow \mathbb{R}$

$f(W, x, y) = \| \tanh(Wx) - y \|^2$

Introduction to Gradient Descent

$$w_{t+1} = w_t - \eta \frac{dJ}{dw}$$



Example

Find minimum of $y = x^2 - 4x + 4$

```
import tensorflow as tf

eta = 0.1 # learning rate
x = tf.Variable(10.0, tf.float32) # arbitrary initial value

for i in range(50):
   with tf.GradientTape() as tape:
      y=#TODO put in formula for y in terms of x here
   dydx=tape.gradient(# TODO finish this line
   x.assign(#TODO finish x_(t+1)=x_t-eta*dydx
   print("iteration:",i, "x:", x.numpy(), "y:", y.numpy())
```

There are also built in optimizers!

We can speed this up a bit by more declarative calculations outside of the loop & using @tf.function

def do_update(x): & inside, use x.assign(—>

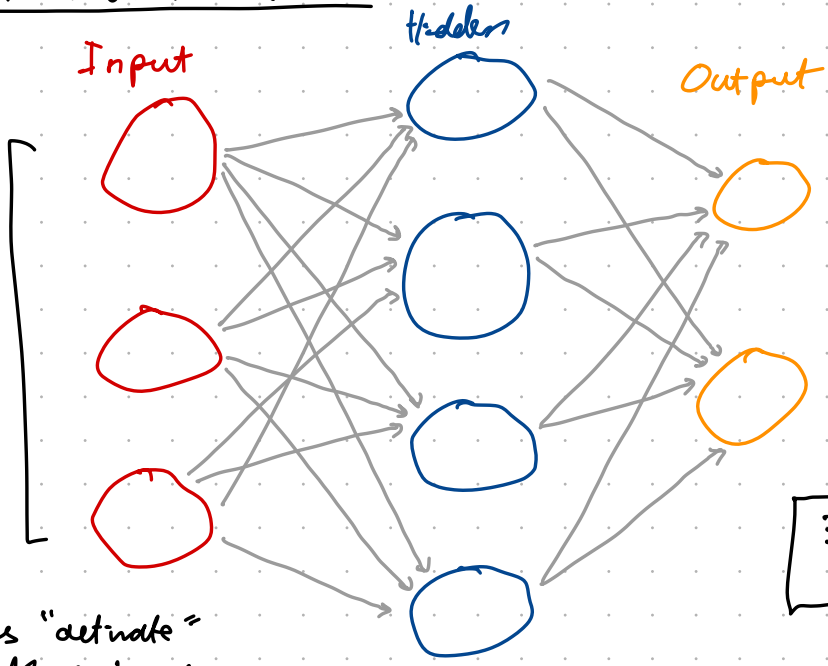
Lecture 2 - Neural Networks

Note: deep learning is deep neural networks. Deep learning is a rebrand because neural networks started working ↑. Why?

1. More training data
2. More CPU/GPU power
3. Clever neural net architecture
   - CNNs
   - Dropout
   - LSTM nodes
   - Transformers
4. Better activation functions & weight initializations.
   - ReLu
   - Xavier initialization
5. Better learning algorithms (e.g. Adam)

breakthroughs that enable modern learning today

What is a neural network?



takes in 3 input values [vector of length 3]

output vector - here of length 2

3-4-2 feedforward neural network

(*)

1. Hidden nodes "activate" with variable intensity
2. Output nodes "activate" with "
3. Arrows are weights which amplify / reduce signals

processes it (*)

A feedforward neural network: $y = f(W, x)$

- W - weights (a list of tensors)
- Give enough appropriately set weights & neural network can represent ANY function. [Universal function approximator]

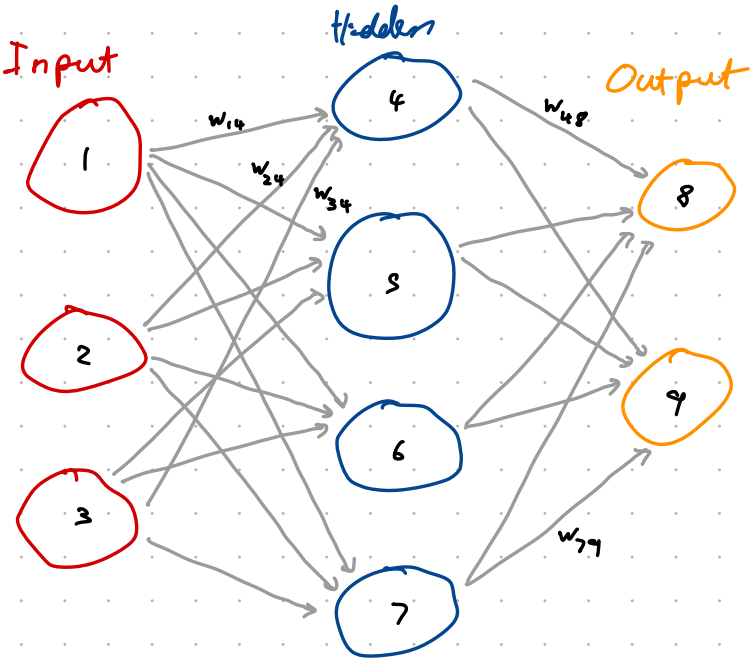
Example	Input	Output
Vision system	grid of pixel intensities in an image	classification of image (cat / house)
Language translation	vector of words in English sentence	vector of words in French sentence

Feedforward Algorithm

- Number the nodes
- Label the weights
  - w_{ij} connects node i to node j.
- Input vector to this network must be dimension 3
  $x = (x_1, x_2, x_3)$
- The "electrical" signals passing along the wires to node 4 are therefore total to

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

$$\boxed{s_k = \sum_{i=1}^{3} x_i w_{ik}}$$



- Activation of node 4: Node 4 will fire when s_4 is bigger than some threshold b_4

$$\left(a_4 = \begin{cases} 1 & \text{if } s_4 > b_4 \\ 0 & \text{if } s_4 \le b_4 \end{cases} \right) \iff \left(a_4 = g(x_1 w_{14} + x_2 w_{24} + x_3 w_{34} - b_4) \text{ where } g(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x \le 0 \end{cases} \right)$$

a_4: activation of node 4

"activation function" / "nonlinearity"

In total:

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

or with matrices:

$$\underset{1\times 4}{(a_4 \; a_5 \; a_6 \; a_7)} = g\left(\underset{1\times 3}{(x_1 \; x_2 \; x_3)} \underset{3\times 4}{\begin{pmatrix} w_{14} & w_{15} & w_{16} & w_{17} \\ w_{24} & w_{25} & w_{26} & w_{27} \\ w_{34} & w_{35} & w_{36} & w_{37} \end{pmatrix}} - \underset{1\times 4}{(b_1 \; b_2 \; b_3 \; b_4)} \right)$$

And for the output layer:

$$\underset{1\times 2}{(a_8 \; a_9)} = g\left(\underset{1\times 4}{(a_1 \; a_2 \; a_3 \; a_4)} \underset{4\times 2}{\begin{pmatrix} w_{48} & w_{49} \\ w_{58} & w_{59} \\ w_{68} & w_{69} \\ w_{78} & w_{79} \end{pmatrix}} - \underset{1\times 2}{(b_8 \; b_9)} \right)$$

By "training" the neural network, we aim to find values of the w_{ij} and b_i so the neural network behaves as we want.

More concisely:

Hidden layer: $h_1 = g(x w_1 + b_1)$

Output layer: $y = g(h_1 w_2 + b_2)$

$h_1 = (a_1 \; a_2 \; a_3 \; a_4)$

$w_1 = \begin{pmatrix} w_{14} & w_{15} & w_{16} & w_{17} \\ w_{24} & w_{25} & w_{26} & w_{27} \\ w_{34} & w_{35} & w_{36} & w_{37} \end{pmatrix}$

$x = (x_1 \; x_2 \; x_3)$

$b_1 = -(b_4 \; b_5 \; b_6 \; b_7)$ ← note minus

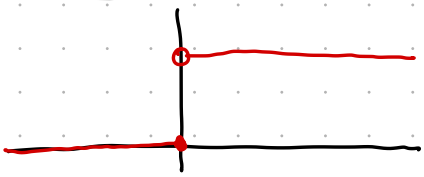
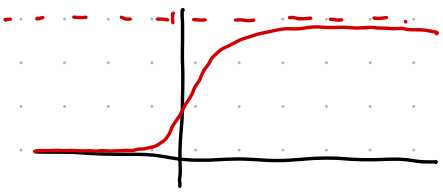
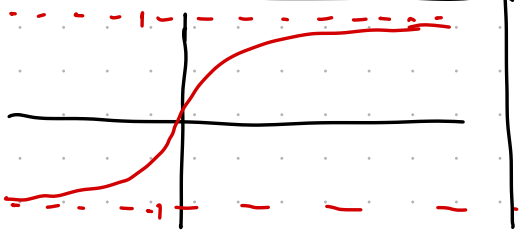
(analogous for w_2, b_2)

$y = (a_8 \; a_9)$

Note: for gradient descent to work, the activation function $g(x)$ needs to be differentiable so $g(x) = 1_{x>0}$ doesn't work!

$\frac{e^x - e^{-x}}{e^x + e^{-x}} \to 1$

Change it to

$g(x)$	Picture	name
$g(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases}$		step function
$g(x) = \frac{1}{1+e^{-x}}$		logistic sigmoid function
$g(x) = \tanh(x)$		Note: Anything non-linear, smooth, increasing works...

$g(x) = \frac{1}{1+e^{-x}}$

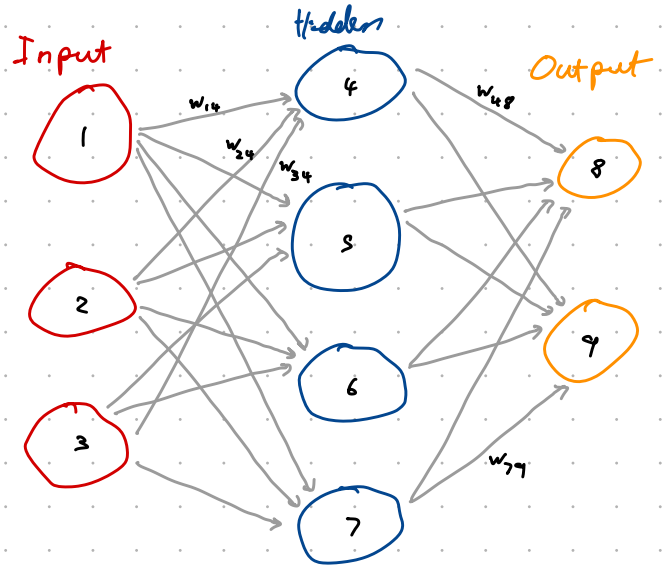
	In maths	In TensorFlow code
for input row vector x	$h_1 = g(x W_1 + b_1)$	`h1=tf.sigmoid(tf.matmul(x,W1) + b1)`
	$y = g(h_1 W_2 + b_2)$	`y=tf.sigmoid(tf.matmul(h1,W2) + b2)`

Can easily add 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)
```

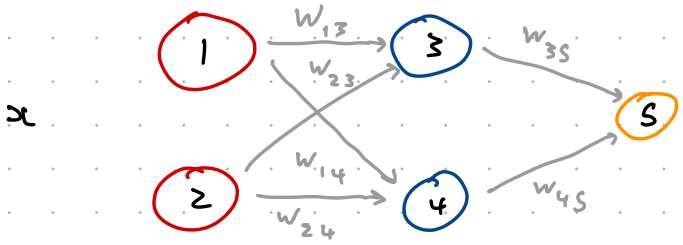
Can only matrix multiply if both arguments have equal rank. $\Rightarrow$ rank(x) = 2

Shape cheat:



Variable		Shape
x		[1, 3]
W_1	"weight matrices" "kernels"	[4, 3]
W_2		[4, 2]
b_1		[1, 4]
b_2		[1, 2]
h_1		[1, 4]
y		[1, 2]

Exercise: Build a 2-2-1 feedforward network



$$h_1 = (a_3 \quad a_4)$$

$$\boxed{h_1 = g(x W_1 + b_1)}$$

$$b_1 = -(b_3 \quad b_4)$$

$$\boxed{y = g(h_1 W_2 + b_2)}$$

$$b_2 = (-b_5)$$

In full:

$$\underset{1\times 2}{(a_3 \quad a_4)} = g\left(\underset{1\times 2}{(x_1 \quad x_2)}\underset{2\times 2}{\begin{pmatrix} w_{13} & w_{14} \\ w_{23} & w_{24} \end{pmatrix}} - \underset{1\times 2}{(b_3 \quad b_4)}\right)$$

$$\underset{1\times 1}{a_5} = g\left(\underset{1\times 2}{(a_3 \quad a_4)}\underset{2\times 1}{\begin{pmatrix} w_{35} \\ w_{45} \end{pmatrix}} - \underset{1\times 1}{b_5}\right)$$

a_5: y; $(a_3 \quad a_4)$: h_1; $\begin{pmatrix} w_{35} \\ w_{45} \end{pmatrix}$: W_2; b_5: b_2

```
import tensorflow as tf

x=tf.constant([[1,1]], tf.float32) # our input vector
tf.random.set_seed(1) # this is just for teaching
# Build our random weight and bias matrices, of appropriate shapes
W1 = tf.Variable(tf.random.truncated_normal([#,#], stddev=0.1))
b1 = tf.Variable(tf.random.truncated_normal([#,#], stddev=0.1))
W2 = tf.Variable(tf.random.truncated_normal([#,#], stddev=0.1))
b2 = tf.Variable(tf.random.truncated_normal([#,#], stddev=0.1))

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

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

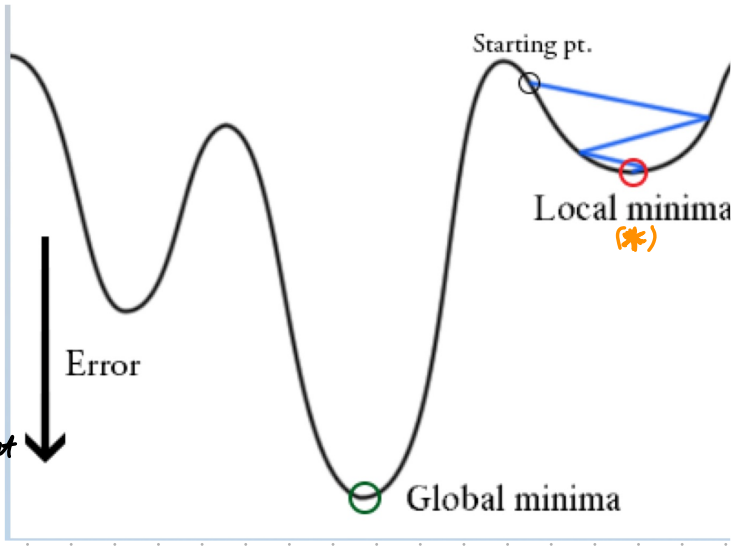
This script is in lecture2_notebook_ffnn.ipynb under "Exercise 1". Complete it now.

Easy now! Just put in the matrices...

Randomising Initial weights

Must randomise weights in a neural network **before** training.

- Symmetry breaking for gradient descent
- Randomised start point for gradient descent.
- Need multiple starts from different random weights to avoid (*)
- magnitude VERY important...



Learning the XOR function

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

4 input vectors

XOR Truth Table

Need to choose the weight & bias tensors to make the network behave as desired.

Now need to modify our feedforward network to push through all 4 input vectors.

- Change `x=tf.constant([[1,1]], tf.float32)`

  To `x=tf.constant([[0,0],[0,1],[1,0],[1,1]], tf.float32)`

Now, instead of passing in $x = (x_1, x_2)$ we pass in a vector of vectors. 4 different 1×2 vectors, so our output (thanks to broadcasting with b_1 & b_2) is 1×4.

we output $y = \begin{pmatrix} 0.531\ldots & \cdots \\ 0.531\ldots & \cdots \\ 0.531\ldots & \cdots \\ 0.531\ldots & \cdots \end{pmatrix}$ but we want $t = \begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \end{pmatrix}$ ← targets

The squared error between output & target is $L = \|y - t\|^2$ ← error function / cost function

LOSS FUNCTION

REDUCE THIS!

We train the network by doing gradient descent on this.

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

← TensorFlow will calculate this for us using Autodiff [Back propagation]

where $w = (W_1, W_2, b_1, b_2)$ ← all weights & all biases.

y_labels = tf.constant([[0],[1],[1],[0]])

Defining our 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
```

$L = \|y - t\|^2$

$\delta_i = y_i - t_i$

$L = \sum_i \delta_i^2$

Now, training our network

```
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())
```

(*)

this lets us implement gradient descent.

variables to minimize

input function

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

Dunno what's going on in these three lines (*)

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

Is short for $w \leftarrow w - \eta \frac{dL}{dw}$ $\quad L = \|y - t\|^2$

```
for i in range(20000):
  with tf.GradientTape() as tape:
    L=calc_loss()                    #define our L = ||y - t||^2
  [dLdW1,dLdW2,dLdb1,dLdb2]=tape.gradient(L, [W1,W2,b1,b2])   calculate partial derivatives
  W1.assign(W1-dLdW1*eta)
  W2.assign(W2-dLdW2*eta)
        b1.assign(b1-dLdb1*eta)
        b2.assign(b2-dLdb2*eta)
```

$\frac{dL}{dw} = \left(\frac{dL}{dw_1}, \frac{dL}{dw_2}, \frac{dL}{db_1}, \frac{dL}{db_2}\right)$

shorthand for backpropagation auto diff calculation!

These lines implement gradient descent:

$w_1 \leftarrow w_1 - \eta \frac{dL}{dw_1}$

$w_2 \leftarrow w_2 - \eta \frac{dL}{dw_2}$

$b_1 \leftarrow b_1 - \eta \frac{dL}{db_1}$

$b_2 \leftarrow b_2 - \eta \frac{dL}{db_2}$

These give new & improved data for $(w_1, w_2, b_1, b_2) = w$.

A lot going on & getting condensed into just three lines!

A "pattern" is a pairing of input & output vectors. Our neural network has (wasn't) 4 patterns.

$$x = \begin{pmatrix} (0,0) \\ (1,0) \\ (0,1) \\ (1,1) \end{pmatrix} \longrightarrow y = \begin{pmatrix} 0.01094\ldots \\ 0.9824\ldots \\ 0.9896\ldots \\ 0.0098\ldots \end{pmatrix}$$

Running Neural Network on New Data

Now, lets evaluate our network on a new pattern (make a prediction based on our training data).

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

Now, restored the random seed function, will make a table to see if we always converge to a valid solution:

Attempt #	Loss at iteration 0	Loss at iteration 10,000	Loss at iteration 20,000	x value ((0,0),(1,0),(0,1),(1,1))	x value (0.5, 0.5)
1	1.0...	0.0007	0.0003	(0.009, 0.99, 0.99, 0.008)	0.9918
2	1.0...	0.9999...	0.999	(0.06, 0.93, 0.92, 0.05)	0.062
3	1.0...	0.9999...	0.00124	0.015, 0.985, 0.98, 0.014	0.0148
4	1.0...	0.0007	0.0003	0.007, 0.99, 0.99, 0.009	0.9919

doesn't seem to ALWAYS converge to zero C

limitations of XOR

To train deeper networks than XOR, we need
- better weight initialisation
- better activation functions
- more training data
- more CPU time

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

M.Fairbank, University of L

Introducing keras layers

Function	Before	After

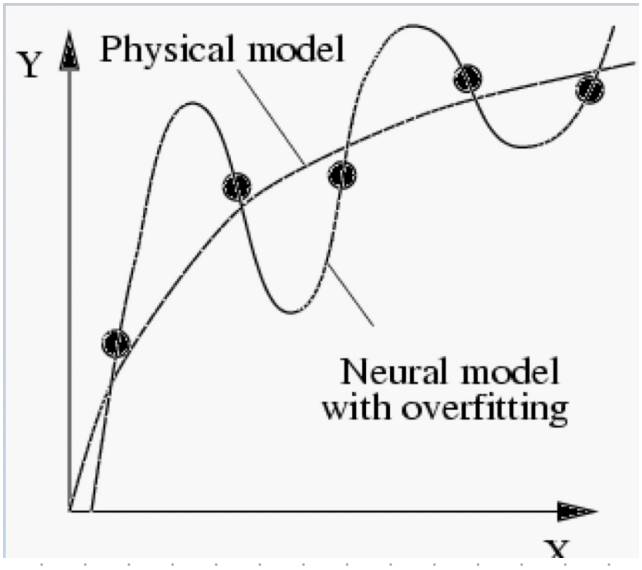
create two network layers	`W1 = tf.Variable(tf.random.truncated_normal([2,2], stddev=0.1))`
	`b1 = tf.Variable(tf.random.truncated_normal([1,2], stddev=0.1))`
	`W2 = tf.Variable(tf.random.truncated_normal([2,1], stddev=0.1))`
	`b2 = tf.Variable(tf.random.truncated_normal([1,1], stddev=0.1))`
	`def run_network(x):`
	`  h1=tf.sigmoid(tf.matmul(x,W1) + b1)`
	`  y=tf.sigmoid(tf.matmul(h1,W2) + b2)`
	`  return y`
	(creates new for us & randomises them for us!)
		`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`
		(layer 1 & layer 2 act as callable functions)

Even easier!	`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`
		`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)`

Fit loop	`optimizer = tf.keras.optimizers.Adam()`
	`for i in range(10000):`
	`  optimizer.minimize(calc_training_loss, trainable_variables)`
	`  if (i%100)==0:`
	`  [train_loss, train_acc, train_cross_entropy]=calc_loss_and_accuracy(inputs_train, labels_train)`
	`  [test_loss, test_acc , test_cross_entropy]=calc_loss_and_accuracy(inputs_test, labels_test)`
	`  print("iteration ",i," loss:", train_loss.numpy()," accuracy:", train_acc.numpy(), "testLoss:",test_loss.numpy(), "test_accuracy:", test_acc.numpy())`
		`optimizer = tf.keras.optimizers.Adam()`
		`model.compile(optimizer=optimizer, loss='mse') # This uses the "mean squared error" as the loss function`
		`history=model.fit(x, y_labels, epochs=10000, batch_size=x.shape[0])`

This is how most people train tensorflow + keras neural networks

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



"Fit" because trying to fit our neural network's behaviour to match the data.

BUT, really its running the gradient descent training loop in full. This is what we've done by hand above.

Versteeg Neural Networks video

- Neurons can either fire or not, a 0 or 1, our $y(x)$ function
- The input of one neuron is the output of many others. Each can be given a weight. Then can excite or inhibit.

- You train to adjust weights.
- The bigger the network:
  - more layers
  - more nodes per layer

  The better the accuracy will be.
- Activation could be step / cat etc.

2-2-1 XOR neural network training

```
import tensorflow as tf

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])

print("Untrained outputs", model(x))
print("trainable vars",len(model.trainable_variables))
print("trainable weights",len(model.trainable_weights))
optimizer = tf.keras.optimizers.SGD(0.5)
model.compile(optimizer=optimizer,
              loss='mse') # This uses the "mean squared error" as the loss function
print("Running training loop...")
history=model.fit(x, y_labels, epochs=10000, batch_size=x.shape[0],
                  verbose=1 # change this to 1 to show output
                 )
print("Finished training")
print("Trained outputs",model(x).numpy())
```

```
Epoch 9994/10000
1/1 [==============================] - 0s 28ms/step - loss: 6.1407e-04
Epoch 9995/10000
1/1 [==============================] - 0s 24ms/step - loss: 6.1400e-04
Epoch 9996/10000
1/1 [==============================] - 0s 18ms/step - loss: 6.1392e-04
Epoch 9997/10000
1/1 [==============================] - 0s 15ms/step - loss: 6.1385e-04
Epoch 9998/10000
1/1 [==============================] - 0s 16ms/step - loss: 6.1378e-04
Epoch 9999/10000
1/1 [==============================] - 0s 18ms/step - loss: 6.1370e-04
Epoch 10000/10000
1/1 [==============================] - 0s 21ms/step - loss: 6.1363e-04
Finished training
Trained outputs [[0.02736123]
 [0.97644806]
 [0.9764228 ]
 [0.02439287]]
```

Lecture 3 — 30/8/23

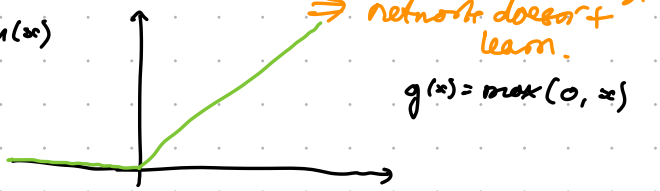
Tricks for Improving Network Training

① Better activation functions

improving

- $g(x) = \frac{1}{1+e^{-x}}$ (logistic sigmoid) is bad. Vanishing gradient possible, then weights don't change because $w \leftarrow w - \eta \frac{\partial L}{\partial w}$ ← if this is small weights don't change ⇒ network doesn't learn.
- $g(x) = \tanh(x)$
- $g(x) = \begin{cases} x & x > 0 \\ 0 & x \leq 0 \end{cases}$ is best. tf.nn.relu(x)

ReLu (Rectified linear function)



② Better weight initialisation

- Need random initialised weights to help gradient descent & break symmetry. Need to avoid vanishing / exploding learning gradients ($\frac{\partial L}{\partial w_i}$) so magnitude of initial weights is important.
- Glorot / Xavier initialisation. Set the standard deviation for a weight matrix of dimension $n \times m$

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

This formula varies depending on:
- (a) Activation function used [how much input before neuron fired]
- (b) What distribution used (? surely always normal?)
- (c) What exact research you follow!

Easy to implement when using higher level functions to create your network layers...

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

Preparing Your Training Data

VERY IMPORTANT TO NORMALISE YOUR INPUTS!!! ← Beginners often forget this step.

Input Pre-Processing

- Neural networks work best when input vectors normalised to have variance 1. For each IR column of your dataset input tensor, subtract off column means & divide each column by the column s.d.

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

$$\text{inputs} = \begin{pmatrix} 0 & 0 \\ 0 & 10 \\ 10 & 0 \\ 10 & 1 \end{pmatrix} \quad \text{column means: } 5 \quad 5.5$$

$$\text{Subtracted inputs} = \begin{pmatrix} -5 & -5.5 \\ -5 & 4.5 \\ 5 & -5.5 \\ 5 & -4.5 \end{pmatrix} \quad \sigma = (\sigma_1, \sigma_2)$$

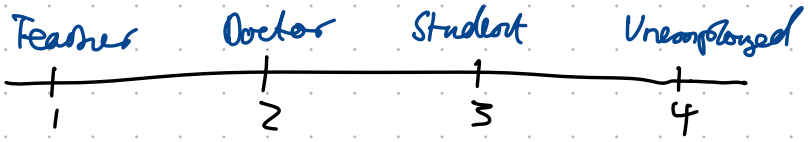
$$\sigma = \sqrt{\frac{\sum (x_i - \bar{x})^2}{n-1}}$$

- Can also do "whitening", seems very cool:
  If the input data is multivariate gaussian, then the whitened data will be gaussian w/ zero mean & identity covariance matrix
  ⇒ remove underlying correlations, eigenbasis vectors something...

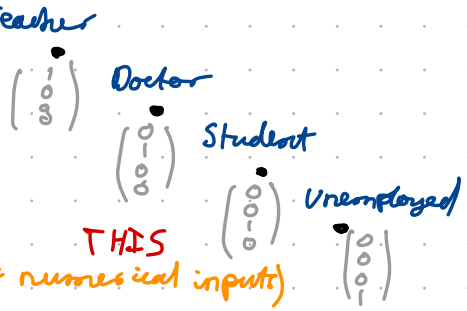
$$\text{inputs normalised} = \begin{pmatrix} -5/\sigma_1 & -5.5/\sigma_2 \\ -5/\sigma_1 & 4.5/\sigma_2 \\ 5/\sigma_1 & -5.5/\sigma_2 \\ 5/\sigma_1 & -4.5/\sigma_2 \end{pmatrix}$$

Categorical Inputs & One-hot encoding

When there is no logical order to data numerically, don't give as a single numerical input into the neural network (NN). Instead use 4 different inputs (one-hot encoding):



NOT THIS
(single numerical input)



THIS
(4 numerical inputs)

Example
- Age of a child: 1 input
- Colour of eyes: multiple inputs (Brown, Blue, Green etc)
- Position of tennis ball: 2 inputs (x & y)
- Alphabet vision system: $\geq 26 \times 2$ characters + punctuation.

Use tf.one_hot function to do this.
Seems like you can reverse it with argmax ???

Training Classification Networks

For say a vision system (classification), we want the NN to output probabilities. Say we have chair, table, door, bed. We want a output vector $y = (-0.2, 0.5, 0.1, 0.1) \longrightarrow$?

$$p = (0.175, 0.352, 0.237, 0.287)$$

$$p_i = \frac{e^{y_i}}{\sum_k e^{y_k}}$$

(softmax function)

makes positive

normalises so they sum to 1.

Cross-entropy loss function

Our NN now outputs a p vector with $p_i = \frac{e^{y_i}}{\sum e^{y_k}}$. We want these trained to become TRUE probabilities. To do this, train the NN w/ a "cross entropy loss function". (physics???) For a single pattern, this is defined to be

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

CROSS ENTROPY LOSS FUNCTION

Where t (vector) is the one-hot encoded true output classification & i is the category index.

Example

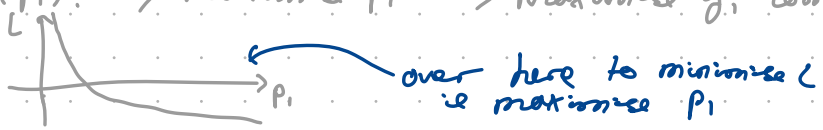
Consider a "table", $t = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix}$, actual NN output $y = \begin{pmatrix} -0.2 \\ 0.5 \\ 0.1 \\ 0.1 \end{pmatrix}$

Category index of "table" is 1 so

	chair	table	door	bed
t_i	t_0	t_1	t_2	t_3
value	0	1	0	0

So here, $L = -\log(p_1)$

THIS MEANS: When training to minimise L, it has to minimise $-\log(p_1) \Rightarrow$ maximise $p_1 \Rightarrow$ maximise y_1 compared to y_0, y_2, y_3.



Where does the formula come from?

Can show (proof??) that:

minimising $L = -\sum_i t_i \log(p_i)$ over your whole data set $\iff$ Finding the weights which maximise your NN's estimation of the total probability that your given training dataset members each have their given (fixed) category label.

Example

① Train NN w/ 60 cat, 40 dog images. Then NN should learn to output $P(\text{cat}) \approx 1$ for each cat image, $P(\text{dog}) \approx 1$ for every dog image. Then here w/ new dog image, get $P(\text{dog}) \approx 1$ for the new data.

② 100 identical photos, 40 labelled "1", 60 labelled "2"; then NN should converge to $P(\text{category } 1) \approx 40\%$

[see further reading... function is clever!]

In TensorFlow

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)
```

MNIST Problem

Breakdown of code

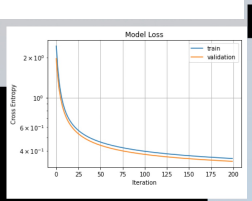
① Load data in Two, (train images, train labels) & same for test.
② Plot some pictures
③ Reshape the image from matrix to vector. Rescale from greyscale to float
④ Create a 1 layer deep 10 node network.
⑤ SGD to optimize, cross entropy loss function.
⑤ Plot loss vs iteration #
⑥ Test against new data.

Floating between 90 - 95% accurate.

Using Matplotlib

To plot the graph just use:

```
import matplotlib.pyplot as plt
plt.plot(history.history['loss'],label="train")
plt.plot(history.history['val_loss'],label="validation")
plt.title('Model Loss')
plt.yscale('log')
plt.ylabel('Cross Entropy')
plt.xlabel('Iteration')
plt.grid()
plt.legend()
plt.show()
```



Using Matplotlib

To log results for plotting, note that the keras "fit" method returns a "history" variable:

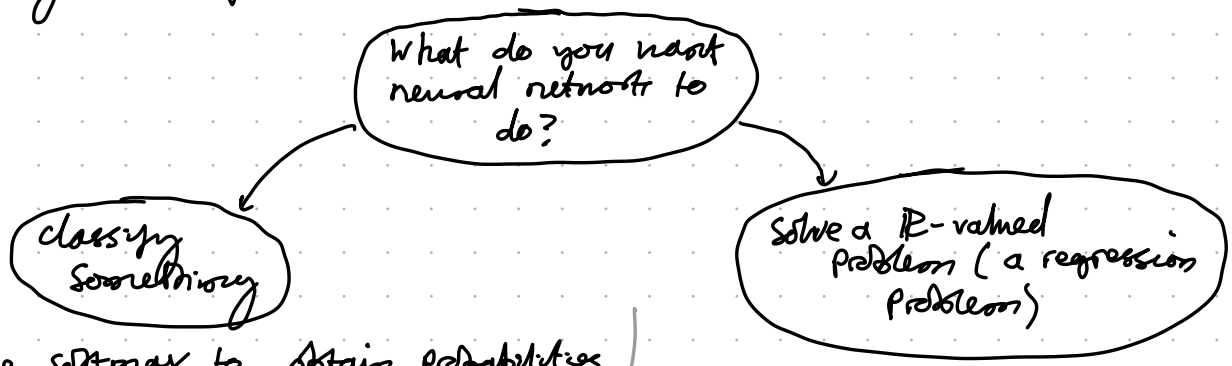
```
history = keras_model.fit(..)
```

This contains a dictionary:

history.history={"loss":[…], "val_loss":[…], "sparse_categorical_accuracy":[…], "val_sparse_categorical_accuracy,[…]}

Each of the values of this dictionary is a numeric array, which we can plot

Summary: Classification vs Regression Problems



- Use softmax to obtain probabilities

$$p_i = \frac{e^{y_i}}{\sum_k e^{y_k}}$$

- Use cross entropy loss function to train

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

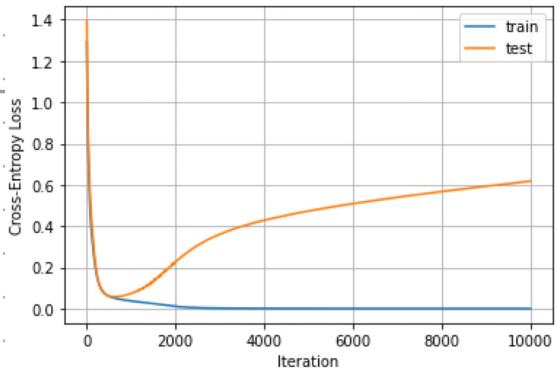
- Use max to pick NN's favoured category

- Train w/ a sum of squares error

$$L = \|y - t\|^2$$

- Don't use soft max on the final layer.

Classic overfitting curve



Example: Iris Dataset

4 inputs, 3 outputs. See notebook for code...

Fighting Overfitting

Whole point of Machine learning is to learn a model that's useful on unseen data. To stop it, we can

① Early stopping: stop training when orange curve ↑.

② More training data: impossible to learn it all

③ Use a simpler model: fewer hidden layers/nodes, constrain the weights to be smaller (Regularisation, L^2). Think Occam's Razor.

④ Use Dropout (major breakthrough!)

⑤ Combine multiple neural networks (ensemble learning)

L^2 Regularisation

$$\text{Total Loss} = \text{cross entropy} + k_{L^2} \sum_i w_i^2$$

choose the constant k_{L^2} (0.01)

We aim to constrain the weights to be smaller. How?

Add a $\sum_i w_i^2$ term to the loss function! [for all neural weights w_i]

Gradient descent minimises "Loss" so this'll force most weights to ↓ in 1.1.

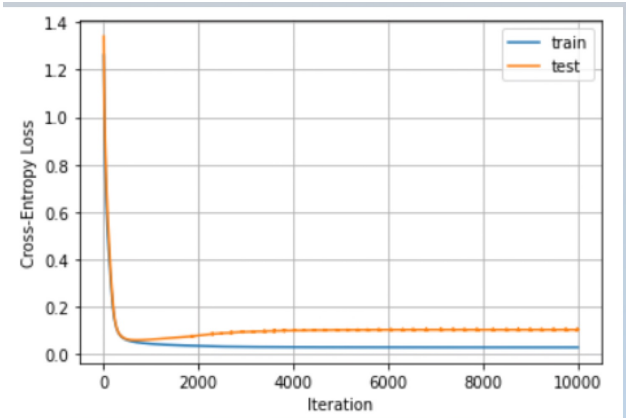
If all weights smaller ⇒ "simpler" model $\xrightarrow[\text{razor}]{\text{occam's}}$ the simpler model that explains the data is more likely to be correct.

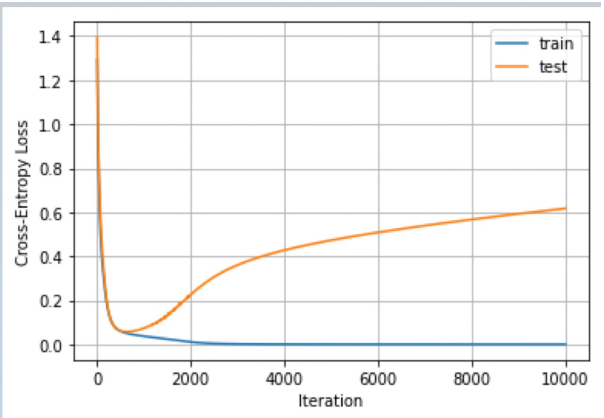
TensorFlow code:

modifies loss function to be composite.

```
hids=[4,20,20,3]
k_l2=0.001
layer1=tf.keras.layers.Dense(hids[1], activation='tanh',kernel_regularizer=keras.regularizers.l2(k_l2))
layer2=tf.keras.layers.Dense(hids[2], activation='tanh',kernel_regularizer=keras.regularizers.l2(k_l2))
layer3=tf.keras.layers.Dense(hids[3], activation='softmax',kernel_regularizer=keras.regularizers.l2(k_l2))
keras_model = tf.keras.Sequential([layer1,layer2,layer3])
```

Results





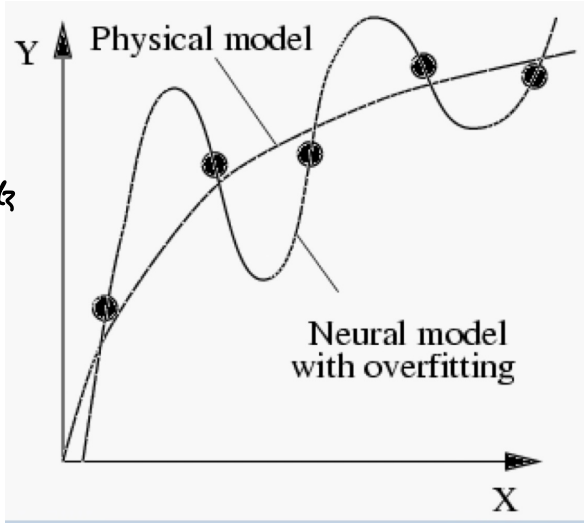
with L^2 regularisation

without L^2 regularisation

[i.e. ↑ k_{L^2}, overfitting ↓ more]

Problems with L^2 regularisation

- When you apply L^2 regularisation, we stiffen the curve on right, prevents it wiggling too much (stops overfitting). It also makes the curve less flexible.
- L^2 regularisation:
  1. Increases "bias" (stiffness)
  2. Reduces "variance" (flexibility)

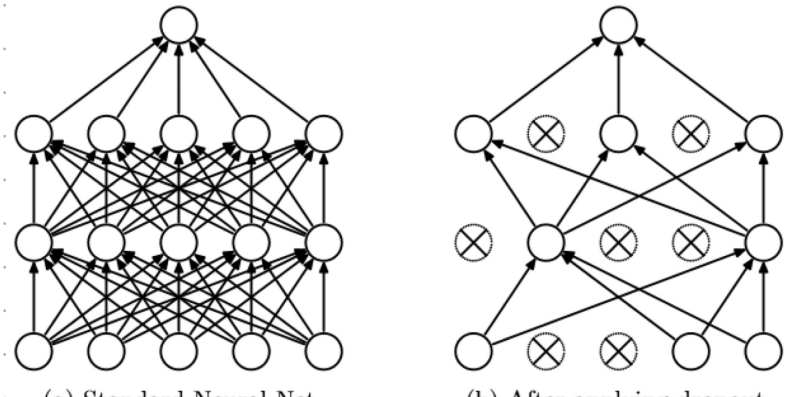


Fighting overfitting: Dropout

Modify hidden layers so nodes randomly switch off 50% of the time.

Radical approach, paper in 2014.

⇒ makes it much harder for neural network to simply memorise all the data.



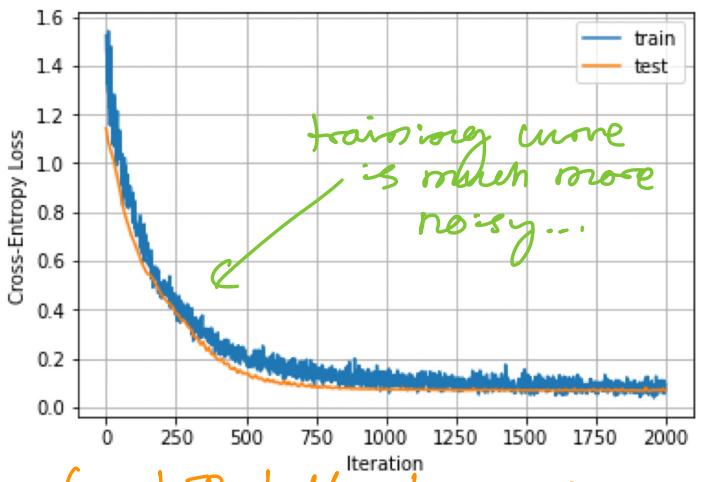
(a) Standard Neural Net (b) After applying dropout.

In TensorFlow code:

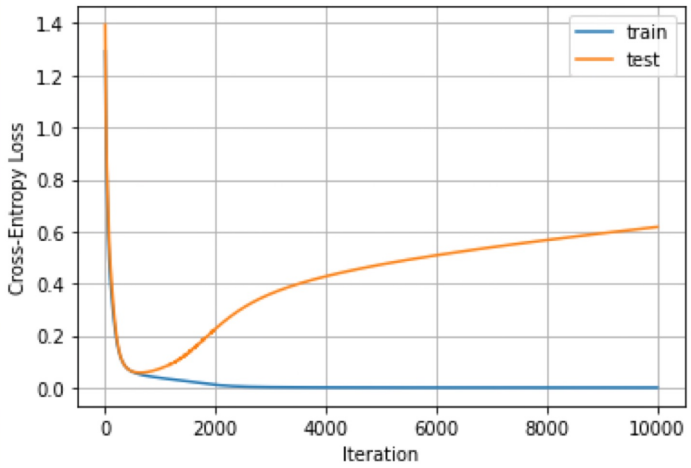
```
hids=[4,20,20,3]
layer1=tf.keras.layers.Dense(hids[1], activation=tf.tanh)
layer1do=tf.keras.layers.Dropout(rate=0.5)
layer2=tf.keras.layers.Dense(hids[2], activation=tf.tanh)
layer2do=tf.keras.layers.Dropout(rate=0.5)
layer3=tf.keras.layers.Dense(hids[3], activation=tf.keras.activations.softmax)
model = tf.keras.Sequential([layer1,layer1do,layer2,layer2do,layer3])
```

Results:

WITH DROPOUT



WITHOUT DROPOUT



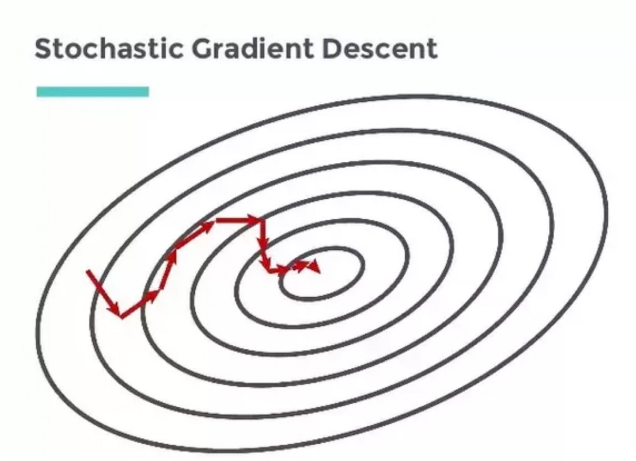
(on both hidden layers of this (4, 20, 20, 3) network)

Takeaways:

- Dropout > L^2 regularization (sometimes)
- Battle overfitting w/ combo of L^2 &/or dropout
- Dropout is a big breakthrough in deep learning

Using Mini Batches

On massive datasets, you need mini Batches. Mini Batches changes "Gradient Descent" to "stochastic Gradient Descent".



Why better?

- Can shake gradient descent out of local minima.
- Improves generalization.
- SGD can be the best learning algorithm (despite being slow)

We don't count # training iterations, we count # training epochs

$$\text{epoch \#} = \frac{\text{iterations} \times \text{mini batch size}}{\text{training set size}}$$

The fit loop will automatically shuffle mini-batches of size 20 for us...

```
history = keras_model.fit(
  inputs_train,
  labels_train,
  batch_size=20,
  epochs=2000,
  validation_data=(inputs_test, labels_test),
)
```

we can save network weights too

This to load a model

```
# Save the final model:
keras_model.save("IrisModel")
```

```
model2 = keras.models.load_model('IrisModel')
```