Introduction to TensorFlow and Deep Learning

Lecture 3: Managing Data

IADS Summer School, 1st August 2022

Dr Michael Fairbank

University of Essex, UK

Email: m.fairbank@essex.ac.uk

Recap

- Summary so far:
  - TensorFlow basics
  - Gradient Descent and automatic gradient finding
  - Feedforward neural networks
- This Lecture (1.30pm-3.00pm):
  - Keras Fit Loop
  - MNIST vision task
  - Loading data
  - Visualising graphs of learning progress
  - Fighting Overfitting: Regularisation and dropout
  - Saving the learned neural networks
  - Shuffling data into minibatches
- Next lecture (3.30pm-5.00pm):
  - CNNs (Convolutional Neural Networks)
  - MNIST revisited (will score $\approx 97.5\%$)
  - Introduction to Recurrent Neural Networks

Higher-level Keras functions: Fit loop

Instead of this

```
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())
```

We can do this:

```
model.compile(optimizer=tf.keras.optimizers.Adam(),
	loss='sparse_categorical_crossentropy',
	metrics=['accuracy’])
history=model.fit(inputs_train, labels_train, epochs=1000, batch_size=inputs_train.shape[0], validation_data=(inputs_test,
labels_test))
```

Higher-level Keras functions: Fit loop

- The Keras “Fit” loop is how most people train Tensorflow+Keras neural networks.
  - (See https://www.tensorflow.org/guide/keras/train_and_evaluate/ for more information)
- It is what we will use in the rest of today’s course.

Higher-level Keras functions: Fit loop

- It's called "fit" because it's trying to fit our neural network's behaviour to match the data.

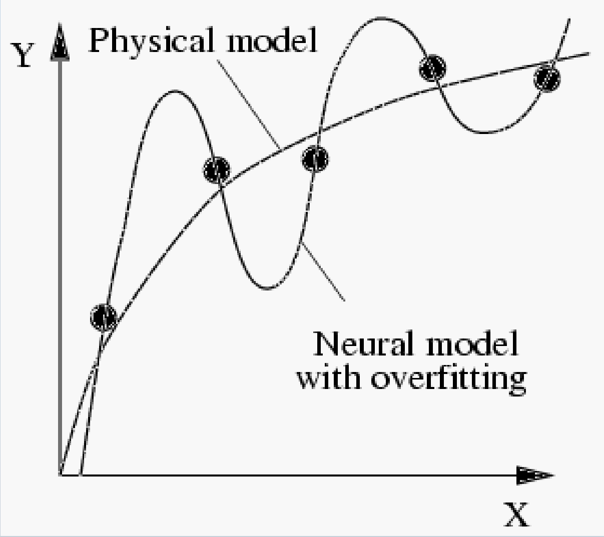


Image source: https://www.gch.ulaval.ca/nnfit/english/man/surappr.gif

- But really it is running the gradient-descent training loop in full:
  - just like we did by hand in the previous 2 lectures!

MNIST digits problem

MNIST problem

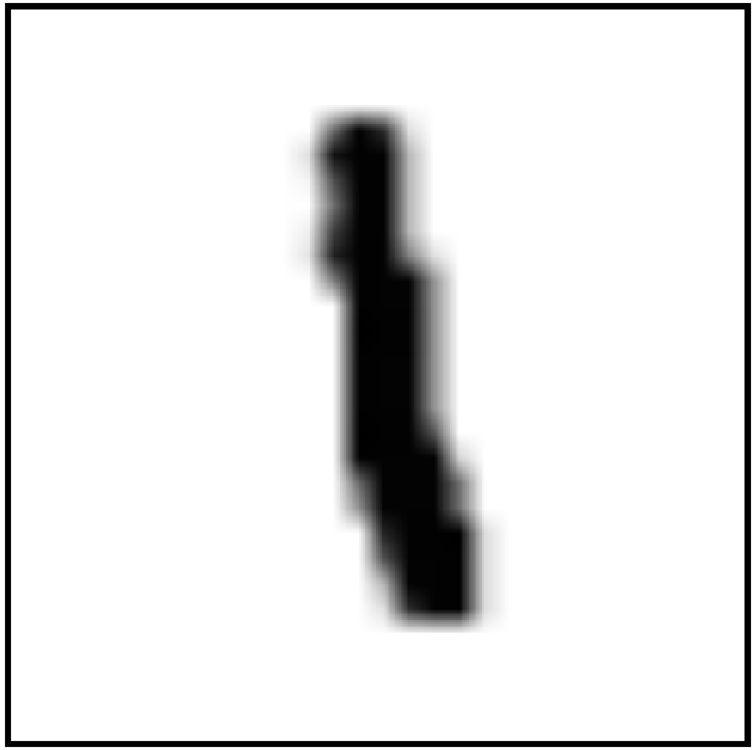
MNIST is a simple computer vision dataset. It consists of 70000 images of handwritten digits like these:

It also includes labels for each image, telling us which digit it is. For example, the labels for the above images are 5, 0, 4, and 1

MNIST problem

Each image is 28*28 pixels or greyscale intensity = array of 784 numbers

Images have already been centered, suitably scaled, and have normalized greyscales.



$$\simeq \begin{bmatrix}
0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&.6&.8&0&0&0&0&0&0\\
0&0&0&0&0&0&.7&1&0&0&0&0&0&0\\
0&0&0&0&0&0&.7&1&0&0&0&0&0&0\\
0&0&0&0&0&0&.5&1&.4&0&0&0&0&0\\
0&0&0&0&0&0&0&1&.4&0&0&0&0&0\\
0&0&0&0&0&0&0&1&.4&0&0&0&0&0\\
0&0&0&0&0&0&0&1&.7&0&0&0&0&0\\
0&0&0&0&0&0&0&1&1&0&0&0&0&0\\
0&0&0&0&0&0&0&.9&1&.1&0&0&0&0\\
0&0&0&0&0&0&0&.3&1&.1&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0
\end{bmatrix}$$

MNIST problem

We can download the images into TensorFlow efficiently:

```
mnist = tf.keras.datasets.mnist
(train_images, train_labels),(test_images, test_labels) = mnist.load_data()
```

(train_images, train_labels) are just 60000 of the 70000 images. The “Training Set”

(test_images, test_labels) are 10000 images for the “test set”:

The labels are integers from 0…9

The images are grayscale as 8-bit integers (i.e. 0 to 255)

MNIST problem

- MNIST images are N*28*28. We will reshape that here to be N*784 – flattens each input image into a single 784-length vector.

```
test_images=test_images.reshape(10000,784) # 10000 test patterns
train_images=train_images.reshape(60000,784) # 60000 train patterns
print("test_images shape",test_images.shape,"train_images shape",train_images.shape)
```

- Also rescale greyscale from 8 bit to floating point (by dividing by 255)

```
test_images=test_images/255.0
train_images=train_images/255.0
```

MNIST problem

Exercise: Run all of the code-blocks in lecture3-notebook-mnist.ipynb

Study the code. Make sure you understand all sections.

Note, the first and last Jupyter code blocks are just cosmetic aspects – loading the data and visualising them.

The middle code blocks build and train the neural network.

Q. How many hidden layers does this NN have?

Q. What recognition rate does it achieve for these hand-written characters?

Q. Which recognition rate is best to use here – test set or training set?

MNIST problem

```
# Create the model
layer1=keras.layers.Dense(10, activation="softmax")
keras_model=keras.models.Sequential(layer1)
keras_model.build(input_shape=[None,784])
```

View the model, using keras_model.summary()

```
Model: "sequential_5"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_5 (Dense)             (None, 10)                7850

=================================================================
Total params: 7,850
Trainable params: 7,850
Non-trainable params: 0
_________________________________________________________________
```

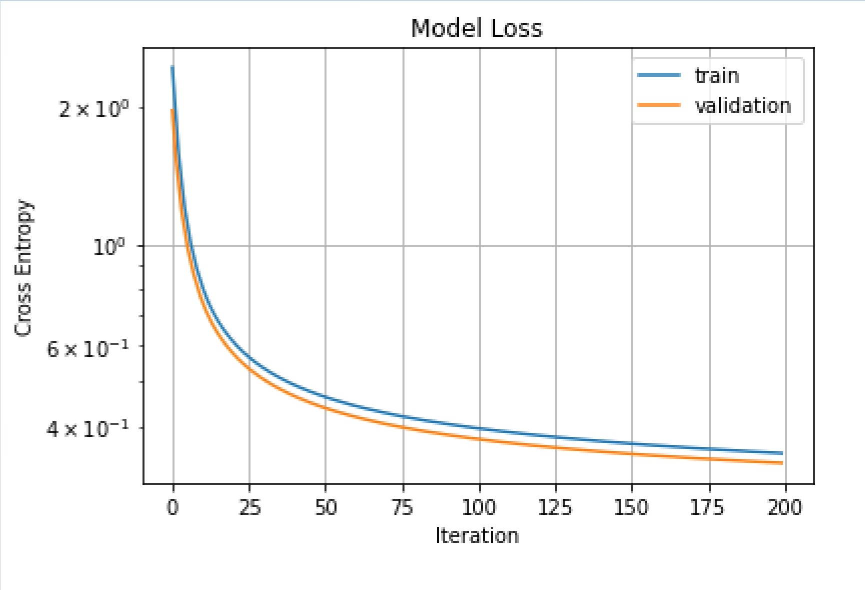
MNIST problem

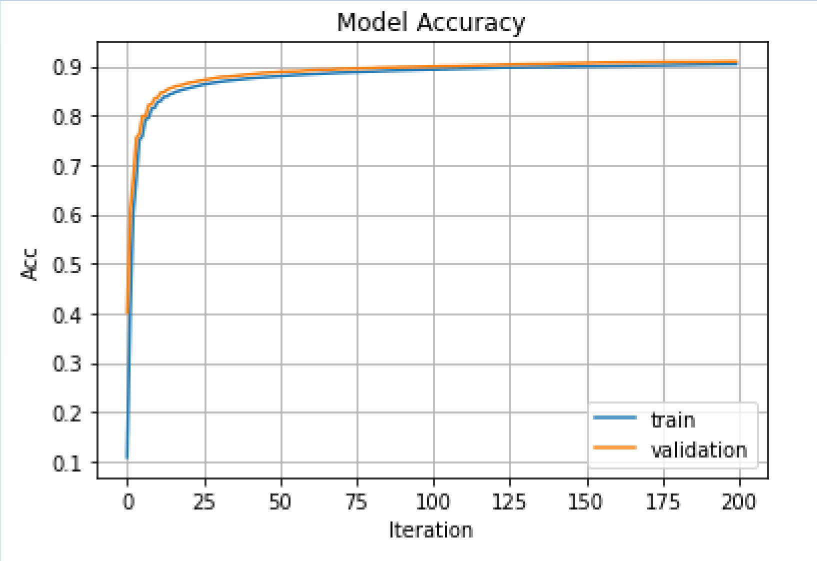
```
optimizer = tf.keras.optimizers.SGD(0.5)
keras_model.compile(
    optimizer=optimizer,  # Optimizer
    # Loss function to minimize
    loss=tf.keras.losses.SparseCategoricalCrossentropy(),
    # List of metrics to monitor
    metrics=[keras.metrics.SparseCategoricalAccuracy()]
)
```

MNIST problem

```
# Train loop
history = keras_model.fit(
    train_images,
    train_labels0,
    batch_size=len(train_images),
    epochs=200,
    validation_data=(test_images, test_labels0),
)
```

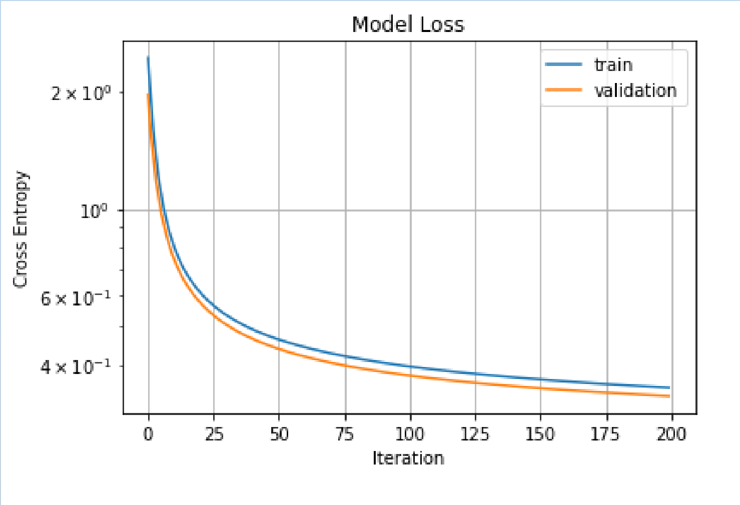
Graphing Training Performance





Using Matplotlib

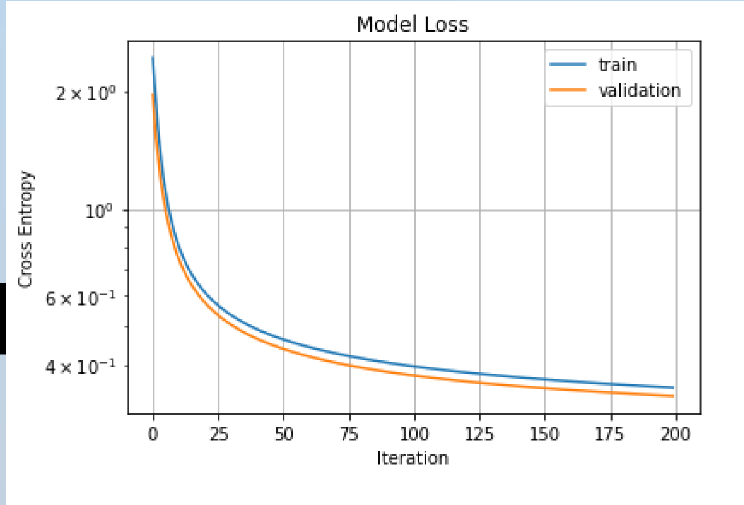
- We want to plot the training loss to see if our neural network performance is improving over time
- To do this we can use matplotlib
- Matplotlib is a python library which enables easy graph plotting
  - (An alternative to use would be "tensorboard": see https://www.tensorflow.org/tensorboard for details)



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:

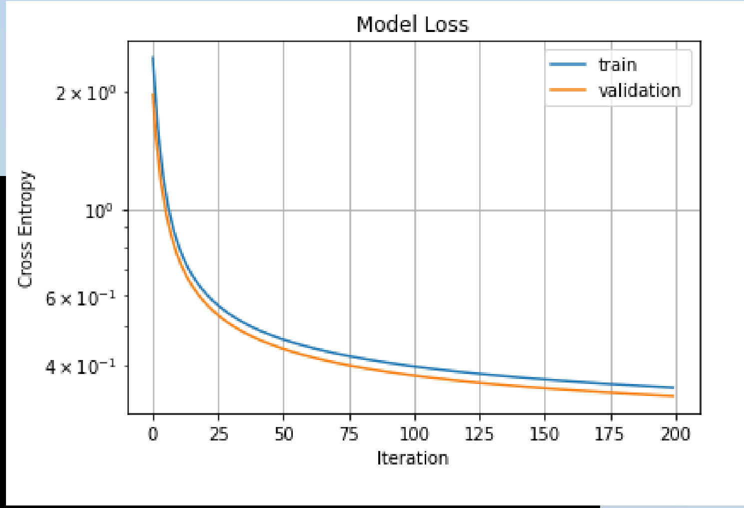
history.history={“loss”:[…], “val_loss”:[…], “sparse_categorical_accuracy”:[…], “val_sparse_categorical_accuracy,[…]}

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

Using Matplotlib

To plot the graph just use:

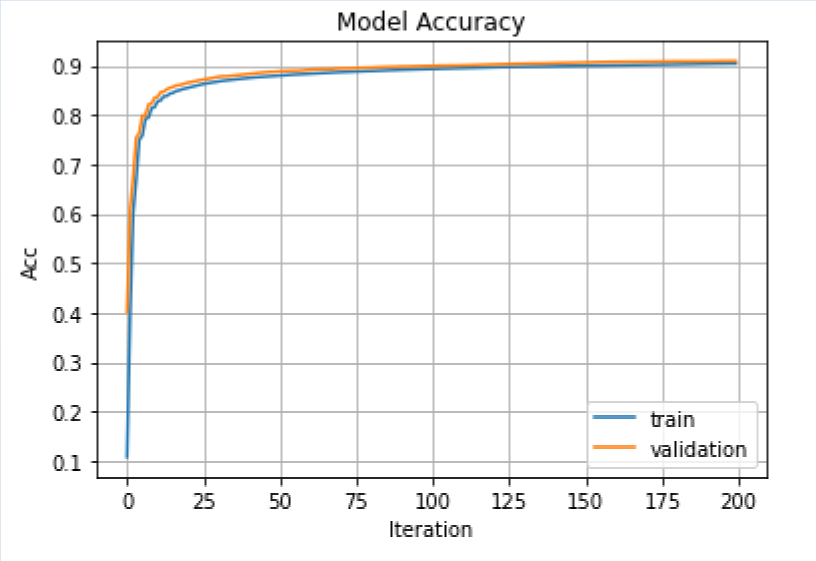
```
import matplotlib.pyplot as plt
plt.plot(history.history['loss'],label="train")
plt.plot(history.history['val_loss'],label="validation")
plt.title('Model Loss')
plt.yscale('log')
plt.ylabel('Cross Entropy')
plt.xlabel('Iteration')
plt.grid()
plt.legend()
plt.show()
```



This code is in the notebook.

Using Matplotlib

Challenge: Add a new code block which plots a second graph – one of accuracy versus iteration (for the two datasets sets, train and validation)



MNIST problem

Challenges:

- Does adding a hidden layer produce better performance on the MNIST dataset?

Iris Dataset

A classification problem to identify 3 different types of Iris flower from their measurements:

0: *Iris Setosa*

1: *Iris Versicolor*

2: *Iris Viriginica*

Data is given for each sampled flower:

(Sepal length, sepal width, petal length, petal width), (species)

Q: How many inputs and outputs would our NN need?

Images by Frank Mayfield.

Iris Dataset: Loading the data

Data is in csv format. There are 150 Rows (120 “train” and 30 “test”)

There are 5 cols in the csv file (4 measurements, followed by the label $\in \{0,1,2\}$). No header row.

First load the data from csv into python, e.g. using “pandas” package:

```
import pandas as pd
inputs_train=pd.read_csv('datasets/iris_train.csv',usecols = [0,1,2,3],skiprows = None,header=None).values
labels_train = pd.read_csv('datasets/iris_train.csv',usecols = [4],skiprows = None,header=None).values.reshape(-1)
```

Iris Dataset: Loading the data

Load the test set csv:

```
inputs_test=pd.read_csv('datasets/iris_test.csv',usecols = [0,1,2,3],skiprows = None,header=None).values
labels_test = pd.read_csv('datasets/iris_test.csv',usecols = [4],skiprows = None ,header=None).values.reshape(-1)
```

This is in the first Jupyter code-block of lecture3-notebook-iris.ipynb

Iris Dataset

Build a 4-20-20-3 neural network:

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

This is in the second Jupyter code-block of the lecture3 notebook

Ignore the training argument – it is only necessary for “dropout” which is explained later.

Iris Dataset

Set up an optimiser:

```
optimizer = tf.keras.optimizers.Adam()
```

Set up training loss function, and compile the model

```
optimizer = tf.keras.optimizers.Adam()
keras_model.compile(
    optimizer=optimizer,  # Optimizer
    # Loss function to minimize
    loss=tf.keras.losses.SparseCategoricalCrossentropy(),
    # List of metrics to monitor
    metrics=[keras.metrics.SparseCategoricalAccuracy()]
)
```

This tells it to also record the "accuracy" metric into the "history". This is useful for plotting later.

Iris Dataset

Set up main training loop:

```
# Train loop
history = keras_model.fit(
    inputs_train,
    labels_train,
    batch_size=len(inputs_train),
    epochs=2000,
    validation_data=(inputs_test, labels_test),
)
```

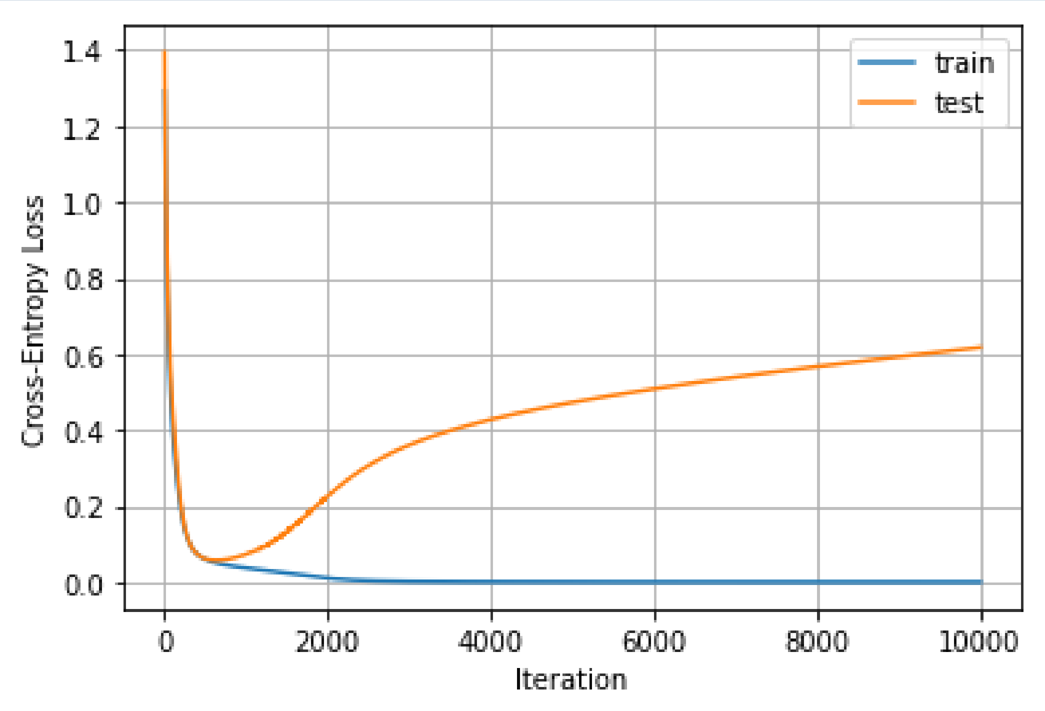
Iris Dataset

- Exercise: see the first 6 code-blocks of the lecture3 Jupyter notebook, run them, and solve the Iris problem.
- Run and study this program code
  - ask questions if necessary

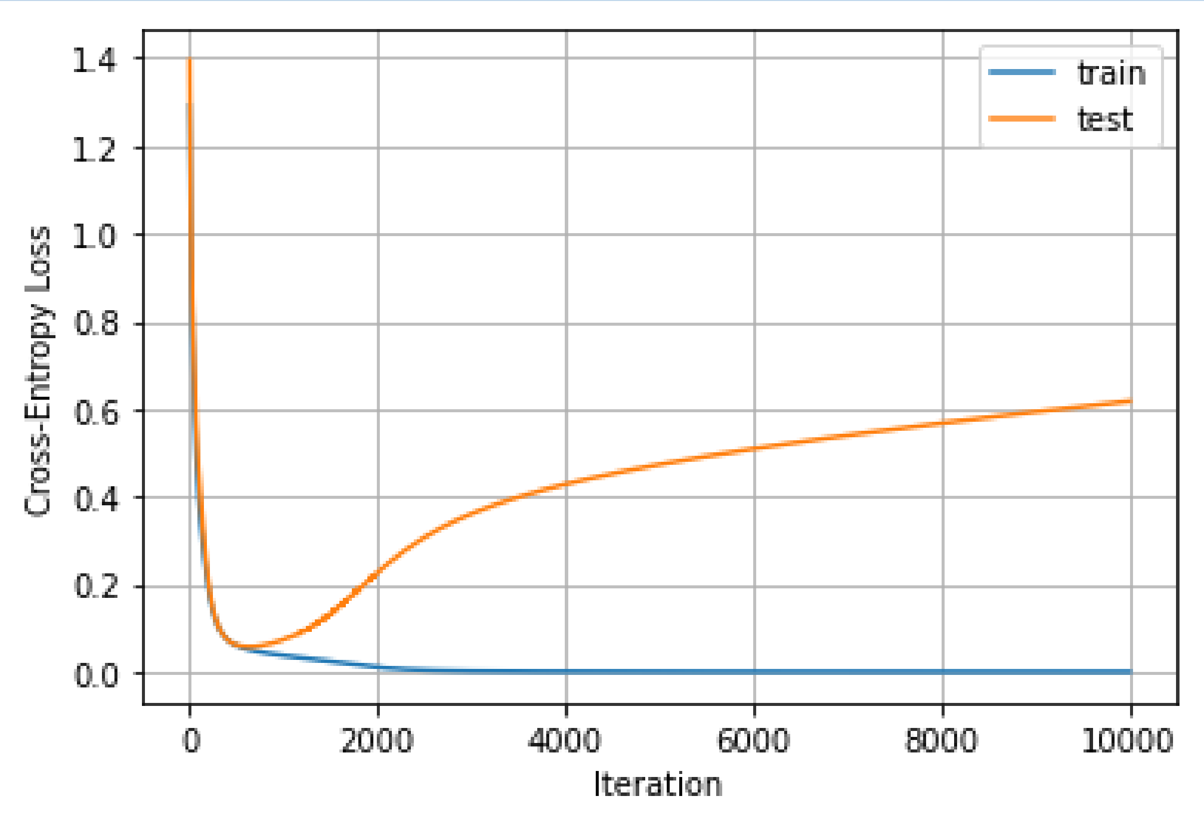
Overfitting

We run the training algorithm, and find that the NN started "overfitting" at around 400 iterations.

Therefore that would have been the best NN to use.

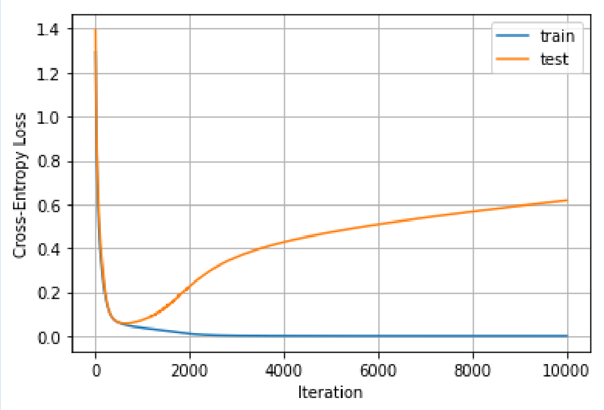


Fighting Overfitting



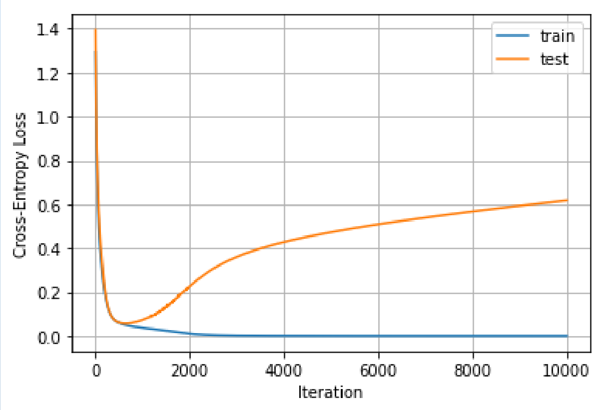
This graph is a classic example of overfitting!

Fighting Overfitting



- The whole point of Machine Learning is to learn a model that is useful on *unseen data*.
  - We don’t want to simply memorise the training data

Fighting Overfitting



- Some methods to try to prevent overfitting
  1. Early stopping
     - Stop training when we see the orange curve start increasing
  2. Get more training data
     - …so it becomes impossible to simply memorise it all
  3. Use a simpler model
     - e.g. fewer hidden layers/nodes
     - Or constrain the weights to be smaller
       - Called "Regularisation".
       - Includes L2 Regularisation
  4. Use Dropout
  5. Combine multiple neural networks (Ensemble learning)

Work by "Occam's Razor"

Fighting Overfitting: L2 Regularisation

We aim to constrain the weights to be smaller

- Add to the loss function a term $\sum_i (w_i)^2$ for *all* neural weights w_i
- Gradient descent minimises "Loss", so this will force most weights to decrease in magnitude
- If all the weights are smaller, in some sense we have a "simpler" model.
  - By Occams' razor, the simpler model that explains the data is more likely to be correct.
  - Further reading: Rasmussen, Carl Edward, and Zoubin Ghahramani. "Occam's razor." *Advances in neural information processing systems*. 2001.

Fighting Overfitting: L2 Regularisation

Total Loss = cross entropy + $k_{L2}\sum_i (w_i)^2$ where k_{L2} is a constant you must choose.

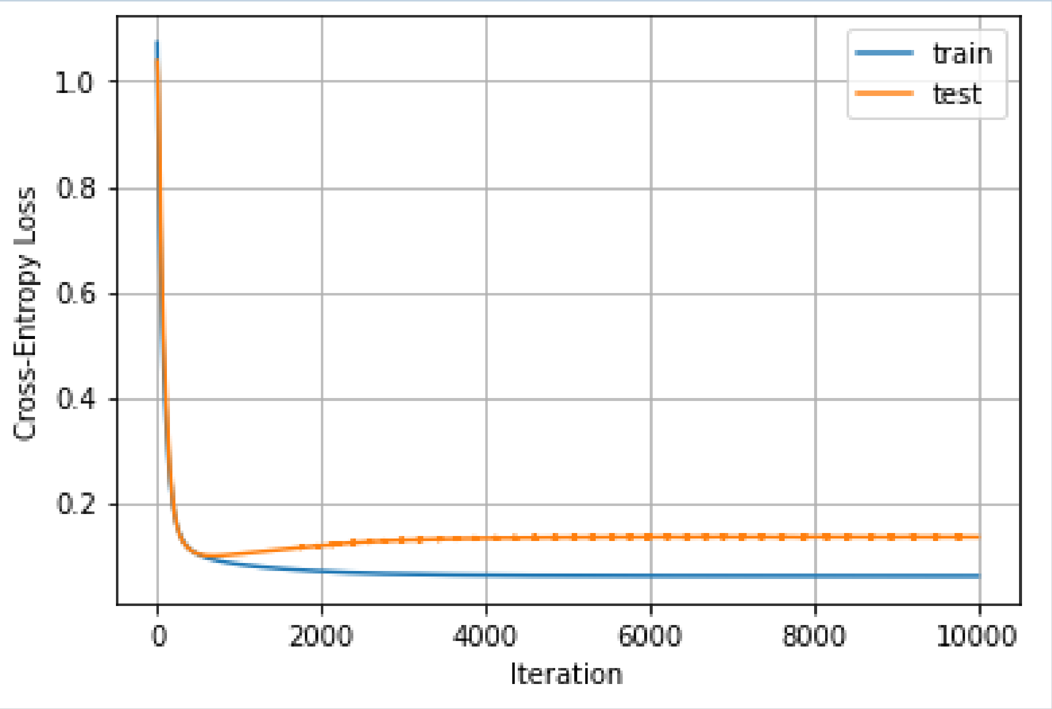
Tensorflow code:

```
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])
```

- Declaring a "kernel_regularizer" with l2 regulariser will modify the loss function used by the fit loop to be a composite:
  - cross entropy + $k_{L2}\sum_i (w_i)^2$
- Exercise: retrain your NN with $k_{L2} = 0.001$.
  - The code is there for you in the notebook; you just need to uncomment some sections

Fighting Overfitting: L2 Regularisation

Results:



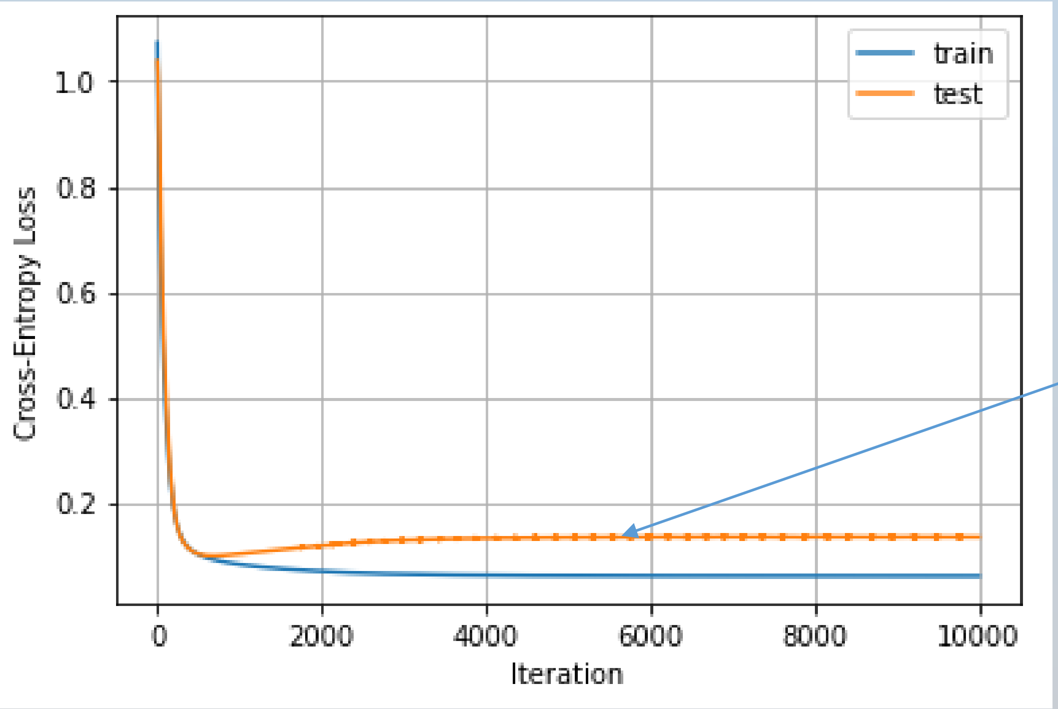
Without L2 regularisation ($k_{L2} = 0$)
Min test cross entropy ≈ 0.06
Strong overfitting

With L2 regularisation ($k_{L2} = 0.001$)
Much less overfitting. If we increase k_{L2} we should reduce overfitting further

Without L2 regularisation ($k_{L2} = 0$)
Min test cross entropy ≈ 0.06
Strong overfitting

Fighting Overfitting: L2 Regularisation

Results:



Note: this loss curve includes the l2 loss; so it's difficult to see the pure cross-entropy loss here.

With L2 regularisation ($k_{L2} = 0.001$)
Much less overfitting. If we increase k_{L2} we should reduce overfitting further

Fighting Overfitting

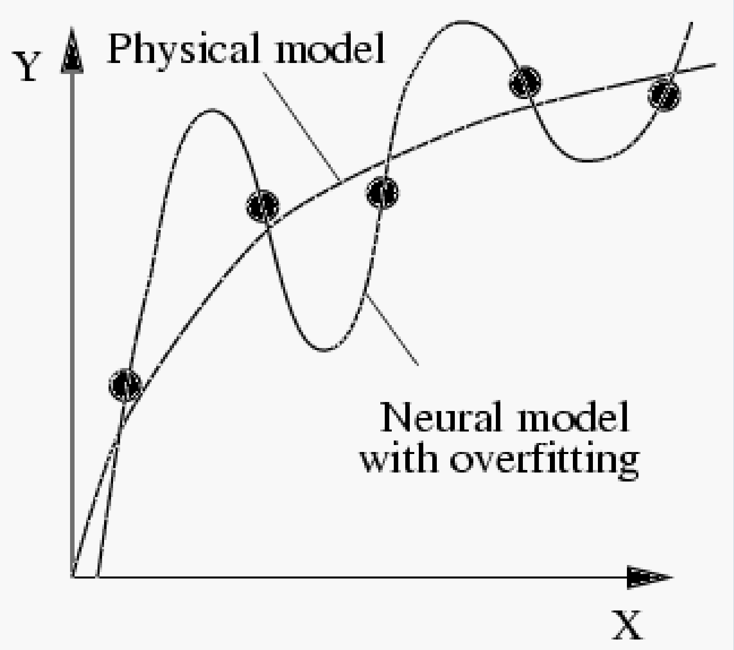


Image source: https://www.gch.ulaval.ca/nnfit/english/man/surappr.gif

- When we apply L2 regularization, we tend to stiffen the curve above, preventing it from wiggling too much (preventing overfitting), but also making it less flexible.
- L2 regularization increases “bias” (stiffness), and reduces “variance” (flexibility)

Fighting Overfitting: Dropout

We modify the hidden layers so that nodes randomly completely switch off 50% of the time.

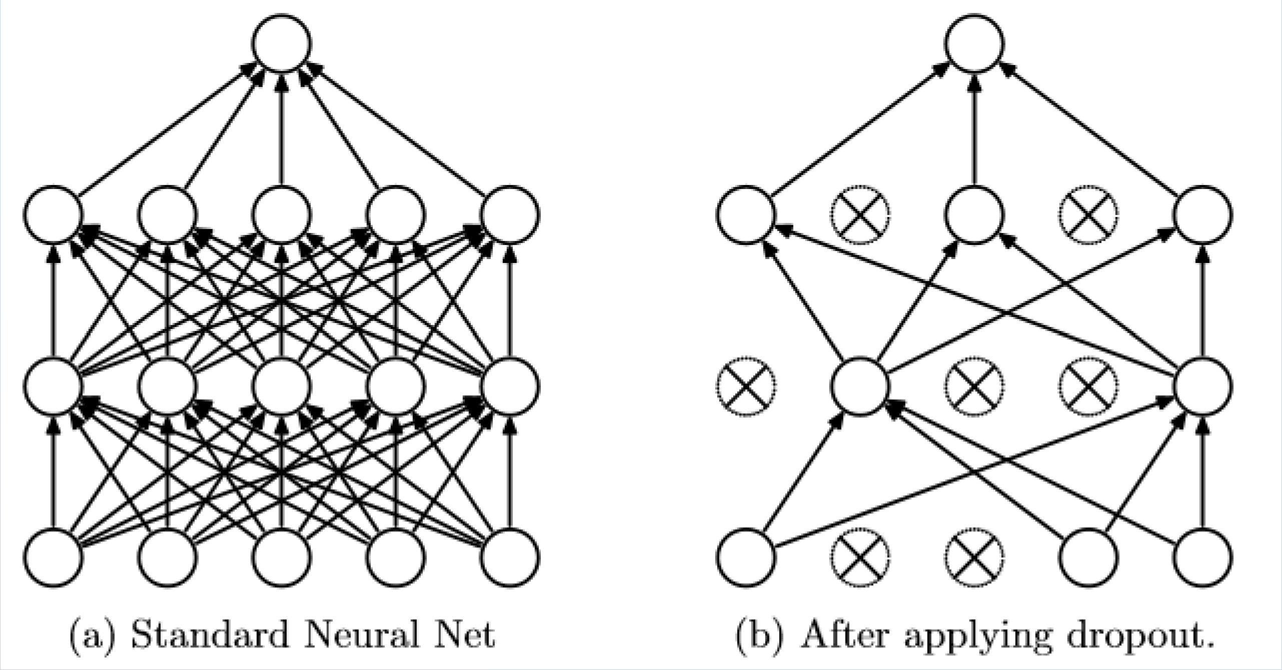


Image by Srivastava et al

Fighting Overfitting: Dropout

This is quite a radical approach developed by

Srivastava, N., Hinton, G. E., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, *15*(1), 1929-1958.

http://www.jmlr.org/papers/volume15/srivastava14a.old/source/srivastava14a.pdf

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

- Other more involved explanation is given by the paper authors involving a comparison to “ensemble learning”

Fighting Overfitting: Dropout

Tensorflow code:

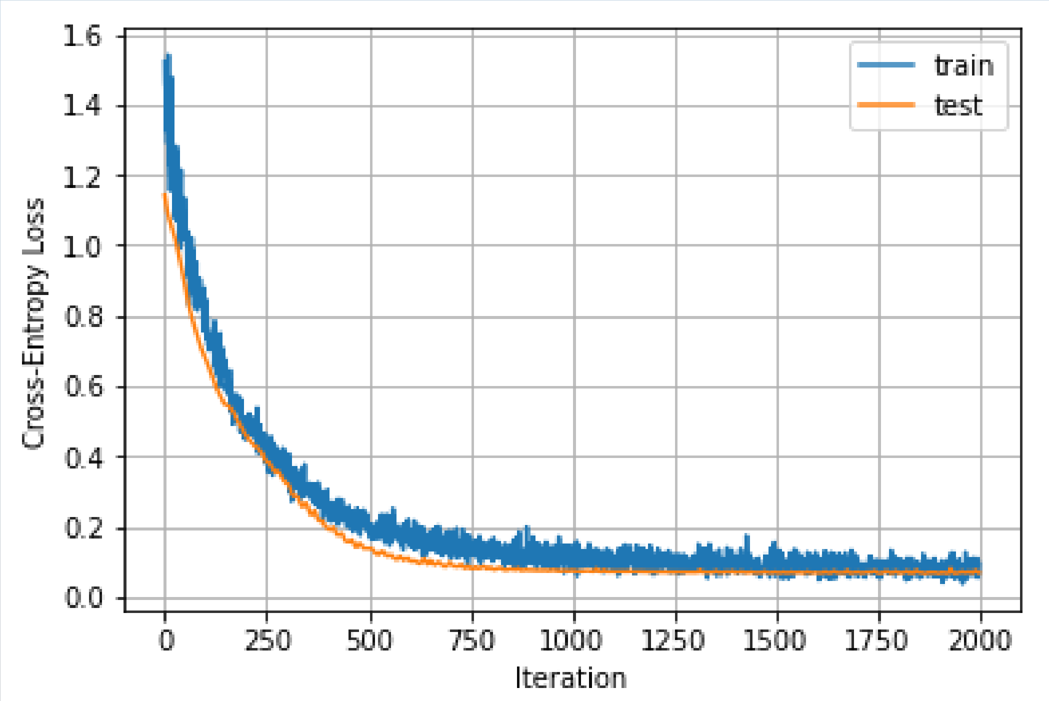
```
hids=[4,20,20,3]
layer1=tf.keras.layers.Dense(hids[1], activation=tf.tanh)
layer1do=tf.keras.layers.Dropout(rate=0.5)
layer2=tf.keras.layers.Dense(hids[2], activation=tf.tanh)
layer2do=tf.keras.layers.Dropout(rate=0.5)
layer3=tf.keras.layers.Dense(hids[3], activation=tf.keras.activations.softmax)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

Fighting Overfitting: Dropout

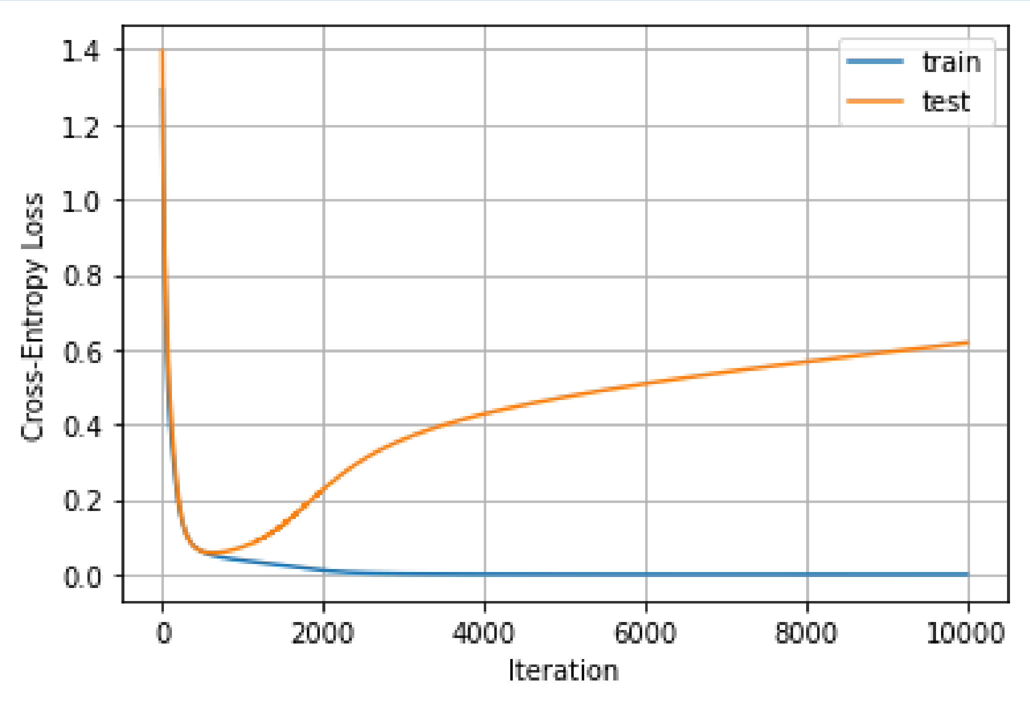
- Exercise: retrain your NN with dropout $rate = 0.5$ (and no L2)
  - You just need to change your network to have the extra two “dropout” layers added to it as shown on previous slide, and remove the L2 regularisation code

Fighting Overfitting: Dropout

Results:



With dropout on both hidden layers ($rate = 0.5$)
Training curve is much more noisy – randomness due to dropout. Much less strong overfitting



Without dropout
Min test cross entropy ≈ 0.06
Strong overfitting

Fighting Overfitting: Dropout

- Dropout can be better than L2 regularisation
- Normally we battle against overfitting by using a combination of L2 and/or dropout
- Dropout is one of the big breakthroughs in deep learning

Using MiniBatches

- On massive datasets, you need minibatches
- Minibatches change "Gradient Descent" into "Stochastic Gradient Descent" (SGD)

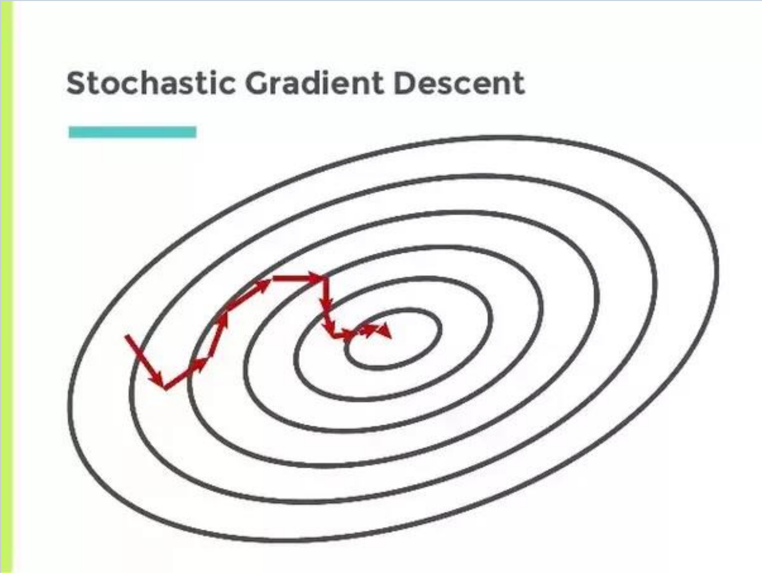


Image source: https://qph.ec.quoracdn.net/main-qimg-d76baf475801ee1a0b850b11f932e719

Using MiniBatches

- Can shake gradient descent out of local minima
- And improve generalisation
- Hence SGD can be the best learning algorithm (despite being slow)

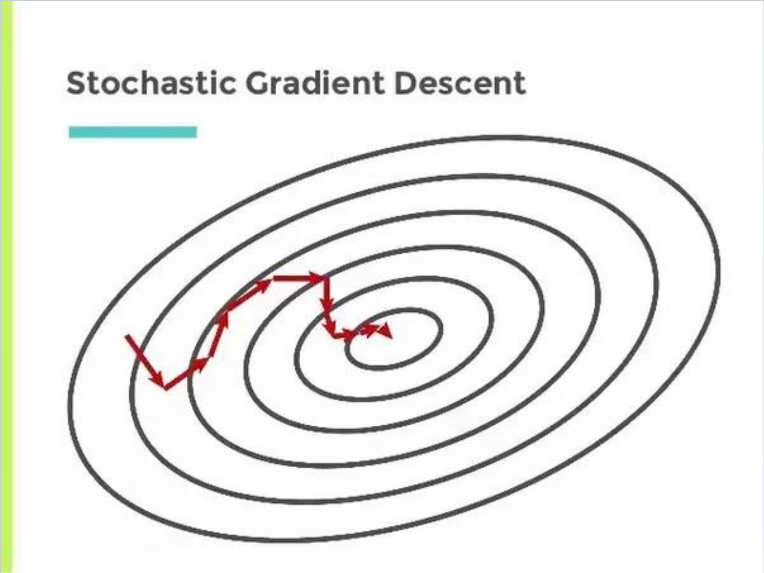


Image source: https://qph.ec.quoracdn.net/main-qimg-d76baf475801ee1a0b850b11f932e719

Using MiniBatches

- With minibatches, instead of counting the number of training "iterations", we often talk about number of training "epochs"
- $Epoch\ number = \dfrac{Iterations \times minibatch\ size}{training\ set\ size}$

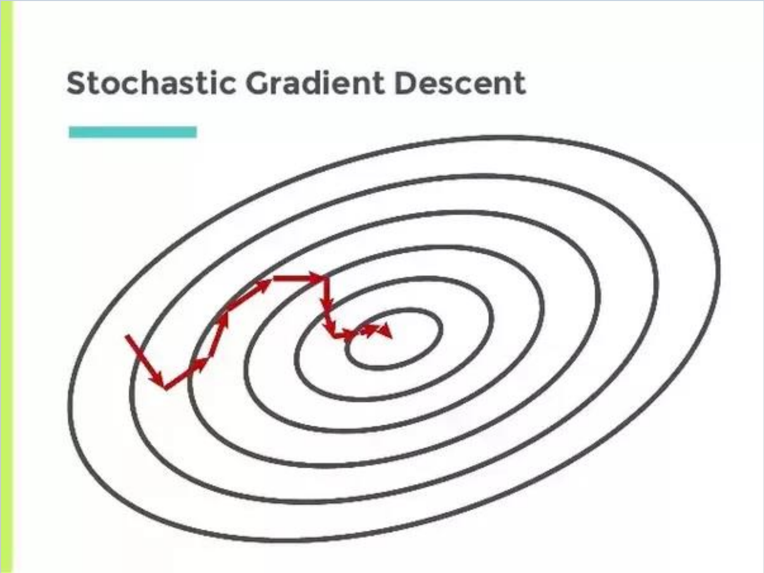


Image source: https://qph.ec.quoracdn.net/main-qimg-d76baf475801ee1a0b850b11f932e719

Using MiniBatches

- Example python code:

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

This is the mini-batch size (20).
The fit loop will automatically shuffle mini-batches of size 20 for us..

Saving your network weights

- The two yellow lines are all that's required.
- This will save your model at the end of training iterations

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

- See https://www.tensorflow.org/guide/keras/save_and_serialize for more information

Saving your network weights

To load the saved model, simply use:

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

This loads the complete model:

- no need to define it beforehand with tf.keras.Sequential
- it includes the saved weights too

Further reading

Further reading:

- Read up on adding a tf.keras.layers.BatchNormalization layer