**Neural Networks**

* Black box framework: give input and it produces an output
* Once the model has been trained, can feed it data and it will produce an output training involves:
  + Data: usually historical data
  + Model: the easiest is linear model (find variables, coefficients, etc), but more complex require non-linear models
  + Objective function: estimates how accurate the model's output is on average, looking to minimise the objective function
  + Optimisation algorithm: vary the parameters of the model to optimise the objective function, for each set of parameters calculate the objective function and calculate the predictive power
  + Repeat the steps above until we find the most optimised model for the data
* Training the model
  + Do not provide step by step instructions to the machine, just state our goal and let the machine work out the problem on its own
  + After thousands of trials and errors, the machine will train itself to reach the goal every time
  + Self-driving cars do not follow rules like "don’t drive on the curb", they watch thousands of hours of footage and mimic that, the goal is to drive safely and efficiently
* Types of machine learning
  + Supervised: provide the machine with inputs and their corresponding desired outputs, based on this it learns how to produce outputs very close to the ones we are looking for
    - Classification supervised learning: provides outputs which are categories
    - Regression supervised learning: provides outputs that are numerical
  + Unsupervised: feed inputs but no target outputs, do not tell the goal but ask to find dependence or underlying logic in the data provided (so a human does not have to match inputs to outputs manually with millions of data points)
  + Reinforcement: train a model to act based on the rewards it receives
* Linear model
  + Although simple, linear model is the basis for more complicated models
  + The machine learning, the coefficient of each variable is called a weight and the intercept is called the bias
  + Y = xw + b
  + The goal of the machine learning algorithm would be to find such values w and b so the output is as close to the observed values as possible
  + With multiple inputs, the formula is the same but x and w are treated as vectors which are multiplied by each other to get a scalar
  + For several outputs, there needs to be several models
  + Machine generated alternative text:
    Y11Y12 • YIM 
    M OOT?OTS 
    Y=XW+B 
    11 X12 
    I N?OTS 
    K x M WEIGHTS 
    M BIASES 
* The objective function
  + Measure used to evaluate how well the model's outputs match the desired output
  + Loss functions: the lower the loss function, the higher the level of accuracy of the model, want to minimise the error of prediction, so minimise the loss, used in supervised learning
  + Reward functions: opposite, the higher the reward function, the higher the level of accuracy of the model used in reinforcement learning
* Regression L2-norm supervised learning
  + Target: the desired value at which we are aiming, generally want y to be as close as possible to the target
  + Calculated by the least squares method (OLS)
  + Norm: comes from the fact it is the vector norm, or euclidean distance of the outputs and the targets
  + The lower the sum is, the lower the error of prediction
* Classification Cross-entropy loss
  + Classifies each category into 0s and 1s and measures the probability that the data is one of those categories
  + After going through the formula, it calculates the accuracy of the model and outputs a L(y,t)
  + Machine generated alternative text:
    DOG 
    y = [0.4, 0.4, 0.2] 
    HORSE 
    Y = [ 0.1, 0.2, 0.7 ] 
    L(y, t) 
    L(y, t) 
    = -Ox In 0.4 -Ix In 0.4 -Ox In 0.2 
    = 0.92 
    = -Ox In 0.1 -ox In 0.2 -IxinO.7 
    = 0.36 
  + The lower the loss function, the more accurate the model
* Optimisation algorithm
  + The actual optimisation process happens when the optimisation algorithm varies the models parameters until the loss function has been minimised (varying w and b)
  + One parameter gradient descent
    - Trying to reach the minimum with as few iterations as possible
    - Eta: learning rate at which the machine learning algorithm forgets old beliefs for new ones
    - Want the learning rate to be high enough so we can reach the closes minimum in a rational amount of time, but low enough so we don’t oscillate around the minimum
    - We find the minimum by trial and error, each trial is better than the previous one, need to think about the learning rate, once converged stop updating
  + N parameter gradient descent
    - Machine generated alternative text:
      xo+l 
      bv+l 
    - First is for one parameter, second is for n parameters
    - Try to vary the weights and biases to minimise the loss function (optimise it with regards to w and b)

**Simple regression: minimal example** (refer to Minimal\_example lecture notes for comments)

* Import the relevant libraries:

|  |
| --- |
| import numpy as np  import matplotlib.pyplot as plt  from mpl\_toolkits.mplot3d import Axes3D |

* Input data, in this case data will be randomly generated
  + Check lecture notes how to generate random data inputs
* Create targets
* Plot the training data
* Initialise the weights and biases
* Set a learning rate
* Train the model
  + Machine generated alternative text:
    Game plan for each iteration: 
    • Calculate outputs 
    • Compare outputs to targets through the loss 
    • Print the loss 
    • Adjust weights and biases 

**TensorFlow 1.0**

* Tensor: a 3D matrix (tensor rank 3)
* Google developed TensorFlow first internally and then released it to the public, now it is the leading library for neural networks
* Uses not only the CPU but also the GPU to make calculations and is crucial for speed, recently introduced TPU (tensor processing unit)
* Kmeans clustering sklearn and random forests is better at the moment
* TensorFlow in Python
  + In TF, each object we create using the tf library would do nothing unless explicitly told to, it rather describes the logic of the algorithm
  + Prefers formats that store tensors (csv files not suitable), prefers .npz files (numpy's file type to save n-d arrays)
  + Data --> pre-process --> save as .npz and build algorithm using the npz
  + To do that: np.savez("file\_name", label=arrays, inputs=generated\_inputs, targets=generated\_targets)
  + Define and input and output sizes

|  |
| --- |
| input\_size = 2 |
| output\_size = 1 |

* + Define inputs

|  |
| --- |
| Inputs = tf.placeholder(tf.float32,[None, input\_size]) |

* + - This is where we feed the data, the data would go into the placeholder, also need to feed targets

|  |
| --- |
| Targets = tf.placeholder(tf.float,[None,output\_size]) |

* + - Use the same names as were specified in the save file (inputs, targets)
    - None means do not need to specify a dimension of observations, just the size of the columns
    - Keep in mind, no data has been fed yet, we just told the algorithm how it will be done
  + Define weights and biases, and outputs
    - Declared using tf.variable(), variables preserve their value through iterations, while placeholders don’t
    - Weights and biases are preserved through each iteration BUT we update them and keep them in the same variable through each iteration (feed the data in the placeholder and vary and variables)

|  |
| --- |
| Weights = tf.variable(tf.random\_uniform([input\_size, output\_size], minval = -0.1, maxval = 0.1)) |
| Biases = tf.variable(tf.random\_uniform([output\_size], minval = -0.1, maxval = 0.1)) |

* + - Random uniform is being used because it is the method used on the data of the minimal example more details on the methods later

|  |  |
| --- | --- |
| Outputs = tf.matmul(inputs, weights) + biases | (y = xw + b) |

* + - matmul is the same as np's dot product but optimised for tensors
    - All of this is just setting the logic and not actually using any data yet
  + Choose the objective function

|  |
| --- |
| mean\_loss = tf.losses.mean\_squared\_error(labels = targets, predictions = outputs) |

* + - Losses is the module within tf that contains most of the common loss functions
    - Mean\_squared\_error: method equal to the average L2 norm loss
    - This method has two arguments: labels and predictions
  + Choose the optimisation algorithm

|  |
| --- |
| optimize = tf.train.GradientDescentOptimizer(learning\_rate = 0.05).minimize(mean\_loss) |

* + - Tf.train is a module that contains most of the common optimisation algorithms
    - GradientDescentOptimizer is the type of optimiser, there are others which can be used as well which will be mentioned later
    - Learning rate is the argument of the optimiser
    - Minimise the loss function that was defined earlier as mean\_loss
  + Create a session
    - tf.InteractiveSession()is a tf class that is used whenever we want to execute anything
    - Create a variable sess so that don’t have to write the function everytime to execute something

|  |
| --- |
| sess = tf.InteractiveSession() |

* + - Training happens in sessions, when we use the function we actually tell it to execute
    - At this point, the entire framework is ready and it's time for the algorithm to learn
  + Initialisation
    - The process of assignment of initial values for variables

|  |
| --- |
| initializer = tf.global\_variables\_initializer() |

* + - Method that initialises all tensor objects "marked" as variables (weights and biases)
    - To run this initializer:
    - sess.run(initializer)
  + Load training data

|  |
| --- |
| training\_data = np.load("FileName.npz") |

* + - Method that loads data from an npz, npy file into an object (has to be on the same directory)
  + Leaning for loop

|  |
| --- |
| for e in range(100):  \_, curr\_loss = sess.run ([optimize, mean\_loss],  feed\_dict = {inputs: training\_data["inputs"], targets: training\_data["targets"]})  print(curr\_loss) |

* + - e stand for epoch (an iteration over the full dataset)
    - Curr\_loss to calculate the loss function at the end of each epoch
    - Sess.run needs to arguments for curr loss, the first is a list where we must state what we want to run (optimise and mean loss), the second argument is feed\_dict which tells the algorithm how the data is going to be fed (placeholder1: data, placeholder2:data)
    - The meaning of the whole curr\_loss is: run the optimiser and mean\_loss operations by filling the placeholder objects with data specified in the feed\_dict parameter)
    - The \_, before is a special symbol to disregard a return value of a function, it is included here because we do not want a return value for the optimize argument
    - This is a generic function that can be used with any data and different machine learning algorithms because the arguments are defined before the loop, just change the appropriate objects rather than the loop
  + Plot the data

|  |
| --- |
| out = sess.run([outputs],  feed\_dict={inputs: training\_data['inputs']})  plt.plot(np.squeeze(out), np.squeeze(training\_data['targets']))  plt.xlabel('outputs')  plt.ylabel('targets')  plt.show() |

* + - Pass outputs as that is the only parameter we need
    - np.squeeze fits the arrays in order to fit them to what the plot function expects
    - The closer the line is to a 45 degree angle, the better the model

**Tensorflow 2.0**

* Logic is the same but some functionality has increased and code has changed in 2.0
* Load training data

|  |
| --- |
| training\_data = np.load('FileName.npz') |

* Define the input and output size

|  |
| --- |
| input\_size = 2 |
| output\_size = 1 |

* Build the model

|  |
| --- |
| model = tf.keras.Sequential ([  tf.keras.layers.Dense(output\_size)]) |

* + TF2 is based on Keras
  + keras.sequential() specifies how the model will be laid down (stacks layers)
  + keras.layers.Dense() takes the inputs provided to the model and calculates the dot product of the inputs and the weights and adds the bias
  + Can add bias and weight initializers with a comma after output\_size in the argument

|  |
| --- |
| kernel\_initializer=tf.random\_uniform\_initializer(minval=-0.1, maxval=0.1) |
| bias\_initializer=tf.random\_uniform\_initializer(minval=-0.1, maxval=0.1) |

* + Can also choose a custom optimizer with a custom learning rate

|  |
| --- |
| custom\_optimizer = tf.keras.optimizers.SGD(learning\_rate=0.02) |

* Objective function and optimisation algorithm

|  |
| --- |
| model.compile(optimizer, loss) |
| model.compile(optimizer = "sgd", loss = "mean\_squared\_error") **OR** |
| model.compile(optimizer = custom\_optimizer, loss = "mean\_squared\_error") |

* + Configures the model for training, the exact names of the optimizer and loss can be found on the tensorflow website

|  |
| --- |
| model.fit(training\_data["inputs"], training\_data["targets"], epochs=100, verbose=2) |

* + Fits the model to the data, epochs is the number is iterations
  + Verbose = 0 stands for silent, o output about the training is displayed
  + Verbose = 1 will print progress bar
  + Verbose = 2 stands for "one line per epoch", the one we want
* Extract the weights and biases to check the model

|  |  |
| --- | --- |
| model.layers[0].get\_weights() | For both weights and bias |
| weights = model.layers[0].get\_weights()[0] | For just weights |
| biases = model.layers[0].get\_weights()[1] | For just biases |

* Make predictions

|  |
| --- |
| model.predict\_on\_batch(data) |
| model.predict\_on\_batch(training\_data["inputs"]).round(1) |

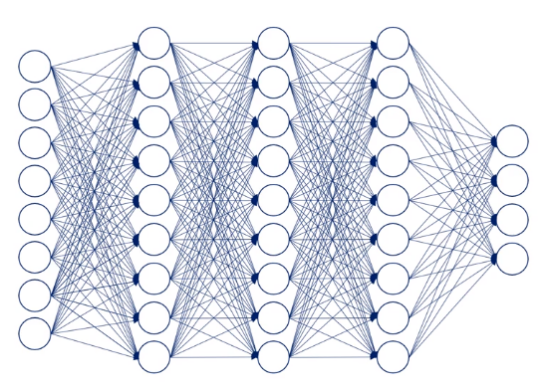
* + Calculates the outputs given inputs
  + These are the values that were compared to the targets to evaluate the loss function
  + Can manually compare by also printing the targets:

|  |
| --- |
| training\_data["targets"].round(1) |

* Plot the data

|  |
| --- |
| plt.plot(np.squeeze(model.predict\_on\_batch(training\_data['inputs'])),  np.squeeze(training\_data['targets']))  plt.xlabel('outputs')  plt.ylabel('targets')  plt.show() |

**Deep nets overview**

* Mixing linear combinations and non-linearities allows us to model arbitrary functions
* Inputs that are linearly combined and go through a non-linear transformation resulting in outputs
  + One common transformation is a sigmoid for example
  + The combination of the linear and non-linear transformation makes up a layer which is a basis of neural networks
  + When there is more than one layer, it is a deep neural network
* Common representation of deep neural networks
  + 
  + The first later is inputs, the rationale is that after getting outputs (the second layer) these can now be used as an input layer until we decide to stop, the last layer is the output layer that we compare the targets to
  + All the layers between are called hidden layers: because we know the inputs and outputs but don’t know what happens in between
  + Each circle in the hidden layers is called a hidden unit/hidden node
  + The number of hidden units in a layer is referred to as the width of the layer, usually layers are stacked with the dame width
  + Depth: the number of hidden layers within a network
  + When we create an algorithm we can choose its width and depth (these are called hyperparameters)
  + Machine generated alternative text:
    Hyperparameters 
    pre-set by us 
    Width 
    Depth 
    Learning rate ( ) 
    vs 
    Parameters 
    found by optimizing 
    Weights ( w ) 
    Biases (b ) 
* Application of deep neural networks
  + Each circle in the input layer is a separate input (the data we feed to train the model): in tensorflow this is the placeholder (tf.placeholder)
  + Combine these inputs linearly and then add non-linearity to produce hidden layer 1
  + Machine generated alternative text:
    Average temperature 
    Max temperature 
    Min temperature 
    Humidity 
    Precipitation 
    Atmospheric pressure 
    Cloud cover 
    Input layer 
    tf.placeholder 
    An illustration of deep nets 
    Hidden layer 1 
    Input 
    Non-linearity 
    Linear 
    combination 
  + Each arrow represents the mathematical transformation of a certain value (non-linearity + weights)
  + Non-linearity doesn’t change the shape of the expression, only its linearity
  + Using the same logic for the second hidden layer (can be as many as 100 hidden layers)
  + Then reach the output layer which depends on how many outputs we would like to have
* Why do we need non-linearity at each step of the hidden layer?
  + One reason is to be able to stack layers: the process of placing one layer after the other in a meaningful way and layers cannot be stacked with only linear relationships
  + Machine generated alternative text:
    Input layer 
    Imagine a network with no non- 
    linearities, just linear combinations 
    Ix8 
    Hidden layer 
    Ix9 
    Output layer 
    Ix4 
    8x9 
    9x4 
    8x4 
    WI 
    8x9 
    9x4 
  + This would mean that the hidden layer is useless, can just train the simple linear model and get the same result
  + Without depth, it is machine learning not deep learning
  + In order to have deep nets and find complex relationships through arbitrary functions, we need non-linearities
* Activation functions (non-linearities)
  + Transform inputs into outputs of a different kind
  + Machine generated alternative text:
    Common activation functions 
    Formula 
    Name 
    sigmoid 
    (logistic function) 
    Tan H 
    tanh(w) ¯ 
    (hyperbolic tangent) 
    ReLu 
    relu(a) 
    (rectified linear unit) 
    softmax 
    Derivative 
    — — a(a)) 
    Where is 1 if izj, O otherwise 
    Graph 
    different 
    every time 
    Range 
    (0,1) 
    (-1, 
    (0,1) 
  + Derivative is an essential part of the gradient descent, in tensor flow this is calculated automatically
  + Range: how the outputs will be contained
  + All common activation functions are monotonic, continuous, and differentiable
  + Softmax function (has no graph because it's different every time)
    - Considers the information from all elements
    - Each element in the output depends on the entire set of elements of the output (different from other functions that just output a transformation for each individual element)
    - Softmax transformation transforms a bunch of arbitrary large or small numbers into a valid probability distribution
    - Because it is related to probabilities that are simple to interpret, this function is often used as the activation of the output layer in classification problems so that no matter what happens before, the output of the algorithm is a probability distribution
* Backpropagation
  + Updates of weights directly related to the partial derivatives of the loss and indirectly related to the errors (deltas, the differences between the targets and the outputs)
  + Deltas for the hidden layers are tricky to define
  + Procedure for calculating them is called backpropagation of errors
  + Forward propagation: process of pushing inputs through the net, at the end of the epoch the outputs are compared to the targets to form the errors
  + Then backpropagate and go backwards and change the parameters accordingly so error is minimised at each epoch
  + Update the parameters as if we had "hidden targets"
  + Trace the contribution of each unit (hidden or not) to the error of the output
  + The weights for each layer are also different
  + Take the errors and backpropogate them through the net (using the U weights, the weights at hidden layer 1), then once we know the contribution towards the error, can update the W weights
  + Machine generated alternative text:
    t1 
    x2 
    The algorithm adjusts: 
    h3 
    the weights that have a bigger contribution to the errors by more; 
    the weights that have a smaller contribution to the errors by less 
  + Pictorially this is straightforward, but mathematically it is rough
  + This is why backpropagation is one of the biggest challenges for the speed of an algorithm

**Overfitting and underfitting**

* Overfitting: the regression has focused on a particular data set so much that is has missed the point
* Underfitting: the model has not captures the underlying logic of the data
* e.g. a linear model for a non-linear regression would be underfitted, it works but does not capture the underlying logic (low accuracy)
* e.g. an overfitted model would be one that touches every point and is so good at modelling the training data that it misses the point when new data is fed to it (random noise is captured)
* First rule of programming: the computer is never wrong, it is us who made the mistake
* Bias-variance tradeoff: the balance between underfitting and overfitting
* Solution: split data set into training, validation, and test
  + Training: helps us train the model to its final form
  + Validation: helps identify and prevent overfitting
  + The model we have updated the weights and biases for the training set only
  + Then take the model and apply it to the validation data set, this time it is run without updating the weights, so only propagate forward not backward (calculate the loss function which should be the same as the training set)
  + We know that the training\_loss should decrease to the minimum value, when performing this on the validation set, if the validation\_loss starts increasing back, that is a red flag
  + Getting better at predicting the training set but moving away from the logic data
  + Machine generated alternative text:
    loss: e.192 
    loss: 
    loss: e. 115. 
    loss: e.e93 
    loss: e.e87 
    loss: e.eu 
    loss: e.e42. 
    loss: e.e74 
    Training 
    loss: e.e29. 
    loss: e.eye 
    e.en 
    training_loss 
    Validation 
    e.egs 
    • e.egs 
    validation loss 
    e.e92 
    Epoch 
    Epoch 
    Epoch 
    Epoch 
    E ROC h 
    Ep•OCh 
    Epoch 
    Epoc h 
    Epoch 
    Epoc h 
    Epoch 
    Epoch 
    EPOC h 
    Epoch 
    EPOC h 
    Epoc h 
    EPOC h 
    Epoc h 
    ENC h 
    Epoc h 
    EPOC h 
    Epoch 
    EPOC h 
    Epoch 
    EPOC h 
    Epoch 
    EPOC h 
    Epoch 
    1. Training 
    2. Training 
    3. Trainlng 
    Training 
    5. Training 
    S. Training 
    7. Training 
    9. Training 
    Training 
    11. 
    12. 
    Trai n 
    Training 
    13. 
    T r •al n Eng 
    Trai nine 
    17. 
    Train i ng 
    IS. Training 
    Training 
    19. 
    2e. Training 
    Train I ng 
    21. 
    22 _ 
    Train 
    Train Eng 
    23. 
    Train 
    Train f ng 
    25. 
    26. 
    Train 
    27. 
    28. Training 
    Tra i ng 
    29. 
    Training 
    loss: 
    loss. 
    loss: 
    loss: 
    e. 341. 
    e.ess. 
    0.868. 
    loss: 
    loss: 
    loss: 
    loss: 
    loss: 
    loss: 
    loss 
    loss 
    loss: 
    loss: 
    loss: e.e18. 
    loss: e.ell. 
    loss: 
    loss: 
    loss: e.ees. 
    loss: 
    loss: 
    e. ae7. 
    e. ae8. 
    e. oea. 
    e.aes. 
    Validation 
    Validation 
    Val i d at 20 n 
    Validation 
    Validation 
    Validation 
    Validation 
    Val i dation 
    Val i dation 
    Val i dation 
    Val t dation 
    Val i dation 
    Val [datlon 
    Val i daticm 
    Val Edation 
    Val i daticm 
    Val i dation 
    Val i dation 
    Val i dation 
    Val i daticm 
    Val [dation 
    Val i dation 
    Val [dation 
    Val i dation 
    Val E "at ion 
    Val i daticm 
    Val 
    Val i dation 
    loss: 
    loss: 
    loss. 
    loss: 
    loss: 
    loss: 
    loss: 
    loss: 
    loss: 
    loss: 
    loss: 
    loss: 
    loss: 
    loss. 
    loss: 
    loss, 
    loss: 
    loss: 
    loss: 
    loss: 
    loss: 
    loss 
    e.las 
    e.106 
    e.e74 
    e.e7s 
    e.e81 
    e.egs 
    8.085 
    e. 391 
    e. 099 
    e. lee 
    e.e92 
    0.094 
    e.e88 
    e. e87 
    e. 089 
    e. 291 
    e.&92 
  + At this point need to stop training the model because beyond that it will start to morph into an overfitted model
  + After we trained the model and validate it, need to test the predictive power
  + Test dataset: measures the final predictive power of the model, running on a test dataset is equivalent to applying it in real life
  + The accuracy that we get by forward propagating the test dataset, is the accuracy we expect the model to have if we deploy it in real life
  + Splitting the data set: usually 80 % training, 10% validation, 10% test is common but not set (or 70/20/10)
  + Usually validate at every epoch, if training loss goes hand in hand with the validation loss, we continue training the model, if validation loss is increasing stop because overfitting
  + What if have a small dataset?
    - N fold cross-validation: resembles the general strategy but combines the training and validation sets in a clever way but still requires the test subset
    - Machine generated alternative text:
      IO-FOLD CROSS-VALIDATION 
      Dataset 
      Val idation 
      Training 
      Validation 
      9,000 
      1,000 
      Chunk I 
      Epoch 1 
      Epoch 2 
      Epoch 3 
      Epoch 4 
      Epoch 5 
      Epoch 6 
      Epoch 7 
      Epoch 8 
      Epoch 9 
      Epoch 10 
      Epoch 1 1 
      For each epoch, we don't overlap training and validation 
      365%'DataScience 
    - Although they don’t overlap, we still trained on the validation dataset which is not a good idea, possible that there is going to be overfitting
    - Should not be used as the norm because very likely to produce an overfitted model
  + Early stopping is the technique to prevent overfitting
    - Train through a preset number of epochs but this is for beginners
    - More sophisticated is to stop when the updates become too small: when it comes less than 0.001. Save computing power by using few iterations and don’t iterate uselessly
    - Each epoch that changes nothing is useless and should be dropped
    - Validation set strategy (best technique):
    - Machine generated alternative text:
      Early stopping 
      Solves the problem 
      Certain that loss is minimized 
      Doesn't iterate uselessly 
      Prevents overfitting 
      Preset epochs 
      x 
      Minimal example 
      Updates too small 
      Validation set 
      Best practice 
    - Use both methods: stop when the validation loss starts increasing or when the training loss becomes very small

**Initialisation**

* When you use clumsy or inappropriate methods, even the fastest computer in the world won't be able to help you
* Initialisation: the process in which we set the initial values of weights
* Types of simple initialisations:
  + Simple approach would be to intialise weights randomly within a certain range: np.random.uniform
  + Could choose a normal initialiser: pick the numbers from a zero mean distribution, variance is arbitrary but should be small (usually sd = 0.1, mean = 0)
  + These methods are problematic but was the norm until 2010
  + Xavier (Glorot) initialisation
    - Made by Xavier Glorot in the paper: Understanding the difficulty of training deep feedforward neural networks
    - Machine generated alternative text:
      Xavierinitialization 
      Uniform Xavier initialization: draw each weight, w, from a random uniform distribution 
      in [-x,x] for x = 
      inputs *outputs 
      Normal Xavier initialization: draw each weight, w, from a normal distribution with a mean 
      of 0, and a standard deviation o = 
      inputs +outputs 
      Main idea: Method is not so important. The number of inputs and outputs is 
    - In tensor flow the Xavier initialiser is default (glorot\_uniform\_initializer)

**Optimisers**

* Algorithms we will use to vary our model's parameters
* Gradient descent (the clumsiest optimiser)
  + Iterates over the whole training set before updating the weights and each update is very small (because the learning rate is small so need more iterations/steps)
* Stochastic gradient descent
  + Works the same way but instead of updating the weights once per epoch, it updates the weights many times inside a single epoch
  + Similar to the process of batching: splitting the dataset into n batches (mini-batches)
  + Update the weights after every batch instead of every epoch
  + Becomes much faster than the gradient descent
  + Comes at a cost: approximates things a bit (lose accuracy) but tradeoff is worth it
  + Splitting the data into batches allows the CPU to train on different batches in parallel and gives a computational speed boost
  + Technically, SGD is updating weights after every input (batch size = 1)
* Gradient descent pitfalls
  + Could be showing the local minima rather than the global
  + Machine generated alternative text:
    Loss 
    Global minimum 
    Epochs 
  + How to reach the global minimum rather than getting stuck in the local one
    - SGD just needs to be extended rather than change (adding momentum)
    - Considering momentum: include the speed with which it has been descending so far
    - Add the previous step to the formula
    - Machine generated alternative text:
      Momentum 
      Loss 
      w e w(t) — n 
      (t) — a 71 
      Current update 
      0.9 is conventional 
      Global minimum 
      Epochs 
      Update a 
      moment ago 
* Learning rate
  + Much be small enough so we gently descend instead of oscillating around the minimum but big enough so the optimisation is done in a reasonable amount of time
  + These are not scientific, need to adopt a learning rate schedule
    - Start from a high initial learning rate (faster training)
    - At some point lower the rate gradually to avoid oscilation
    - Around the end of the training we pick a very small rate to get a precise answer
    - Machine generated alternative text:
      Learning rate schedules 
      1. We start from a high initial 
      learning rate 
      2. At some point we lower the rate to 
      avoid oscillation 
      3. Around the end we pick a very small 
      rate to get a precise answer 
      A learning rate schedule causes the loss to converge much faster 
      First 5 epochs 
      Next 5 epochs 
      n = 0.01 
      Until the end 
      = 0.001 
    - Not best practice but good for beginners
    - Smarter approach is the exponential schedule: smoothly decays the learning rate
    - Machine generated alternative text:
      1. We start from a high initial 
      learning rate 
      2. At some point we lower the rate to 
      avoid oscillation 
      3. Around the end we pick a very small 
      rate to get a precise answer 
      Learning rate schedules 
      71 = noe 
      - 0.1 
      = 0.0967 
      = 0.0905 
      = 0.0819 
      = 0.0717 
      = 0.0607 
      - 0.0497 
      = 0.0393 
      = 0.0301 
      current epoch 
      -n/20 
      some constant 
    - The constant doesn’t matter that much, what matters is the presence of a learning schedule, can try different values and see how it affects
    - Good to experiment with different hyperparameters before sticking to one
    - Machine generated alternative text:
      Lear ing rate. A picture 
      Very high learning rate 
      Loss 
      Good learning rate 
      High learning rate 
      Low learning rate 
      Number of epochs 
      A learning rate following a schedule would minimize the loss faster 
      than a low learning rate, and more accurately than a high learning rate 
      365' 
  + AdaGrad: adaptive gradient algorithm
    - Dynamically varies the learning rate at each update and for each weight individually
  + RMSprop: root mean square propogation
    - Not monotonous, it can adapt upwards and downwards
  + Machine generated alternative text:
    AdaGrad 
    /adaptive gradient algorithm/ 
    aL 
    Am (t) = — 
    Gi(t) + 
    with beginning point Gi(0) = O 
    RMSprop 
    /root mean square propagation/ 
    Awi(t) = 
    Gr(t) + e 
    Gi(t) — ßGi(t 
    with beginning point Gi(0) = O 
    ß — yet another hyperparameter 
    usually, around —0.9 
  + Adam: adaptive moment estimation
    - Combining momentum and AdaGrad and RMSprop
    - Most advanced optimiser applied in practice (relatively new, proposed in 2015)
    - Machine generated alternative text:
      Adam 
      /adaptive moment estimation/ 
      AWi(t) - 
      + e Dwi 
      AWi(t) = 
      Mi(t) 
      aMi(t — 
      MI(O) = O 
      Mi(t) 
      As with all science. data science is a long chain of academic research building on top of each other 
      365%'DataScil 

**Pre-processing**

* Preprocessing: any manipulation of the dataset before running it through the model
* Saving the file into .npz file is a type of preprocessing
* Must make the data compatible with the tools you use
* May need to adjust inputs of different magnitudes
* Generalisation is common, problems that seem different can be used on the same model after standardisation
* Relative metrics are important with time-series data --> log transformations
* Standardisation (feature scaling): Process of transforming the data into a standard scale
* PCA (principle components analysis): dimension reduction technique used to combine several variables into a bigger latent variable
* Whitening is performed after PCA and removes the underlying correlations between data points
* Dealing with categorical data: can't just number 1-n because it implies a hierarchy
  + Binary encoding
    - Turn the numbers (1,2,3, etc) into binary code:
    - 1: 01
    - 2: 10
    - 3: 11
    - Then divide the two numbers into two variables
    - But there is still implied correlation between then
  + One-hot encoding
    - Very common and used often
    - Create as many columns as there are variables
    - Each column asks "is this variable variable 1?"
    - This way, all columns have only 1 yes in each column indicating which one it is and 0s for everything else
    - Makes the variables uncorrelated and unequivocal
    - BUT requires a lot of new variables and impossible for large variables with lots of categories in which case binary would be better

**Deeper example - MNIST dataset**

* 70,000 handwritten digits and need to build an algorithm that classifies them correctly (0-9)
  + Easy to understand as it is visual
  + Extremely common and is the "hello, world" of machine learning
  + Easy to build up to convolutional neural networks later on
  + Very big and preprocessed (clean, no missing values, smudged photos)
  + Developed by Yann LeCun, one of the founding father of CNNs and image recognition and is VP and head of AI at Facebook
* How to approach the problem
  + Each photo is 28x28 pixels, so can think about the problem as a 28x28 matrix with values from 0 to 1
  + The approach for deep neural networks is to "flatten" each 28x28 image into a vector of 784x1
  + Each image would have 784 inputs, each pixel is an input for our neural network
  + Each pixel corresponds to the intensity of colour (0 completely white to 1 completely black)
  + For this example, there are 784 inputs, 2 hidden layers (enough for good accuracy), and 10 outputs (numbers 0-9)
  + One-hot encoding for the outputs and the targets
  + Since we want to see probabilities of digits being accurately labelled, we will use softmax activation function for the output layer
* Method approach:
  + Outline the model and choose the activation functions we want to employ
  + Describe placeholders, variables, and related operations
  + Choose the appropriate advanced optimisers
  + Split data set into batches for faster learning
  + Initialise the variables
  + Make the model learn
  + Test accuracy of the model
* The original course was written in tensorflow 1.0 (but from now one the notes will be for tf 2.0), see 12.9. TensorFlow\_MNIST\_with\_comments for the solution in 1.17
* All notes for the MNIST exercise are in Jupyter under "Deep Neural Networks with MNIST)

**Business case - audiobooks**

* Data from an audiobook app, where each customer has made a purchase at least once
  + Want to create a ML algorithm that will predict whether the customer will buy from the audiobook store again
* Focus on the customers that are more likely to come again to improve sales
* The ML algorithm will show which are the most important metrics for conversion
* Action plan:
  + Preprocess data: as it is rea life data
    - Balance the dataset
    - Divide dataset into training, validation, test
    - Save the data in a npz format
  + Create the ML algorithm
* Balancing dataset is important
  + If the dataset is not balanced and you have 90% cat photos and 10% dogs, the algorithm would optimise the time and power and always output cats since it will give 90% accuracy
  + Balanced if half of the dataset was dogs and half cats
  + To balance, counting the numbers of target 1s and matching the same number of target 0s so it's not unbalanced
* Rest of the notes on Jupyter

**Conclusion**

* CNNs: convolutional neural networks
  + Deals with the problem without flattening the images
  + Instead applies e.g. 5x5 kernels to each position of the image
  + Mainly used in image recognition because it matters where in the photo we find the detail
* RNNs: recurring neural networks
  + Designed for sequential data: trading, music, speech recognition