Deeplearning with Tensorflow

Input layer → Hidden layer → Output layer → Output

The number of hidden layers represents how "deep" a model is

Tensorflow

Tensorflow is one of the most popular deep learning frameworks. The name is derived from tensors, which are basically multidimensional (ie generalised) vectors/matrices

→ import tensorflow as tf.

Model Initialisation

When building the computation graph of the model, tf.placeholder acts as a "placeholder" for the input data and ~~la~~ labels. Without it, we would not be able to train the model on real data.

→ tf.placeholder(type, shape= , name=)

name → allows us to give a name to placholder

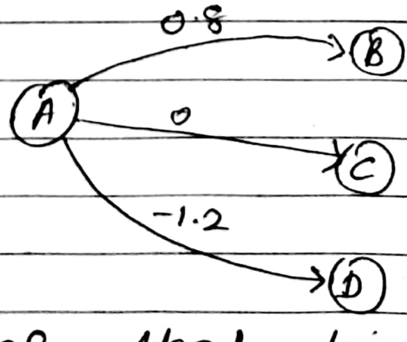
type → data type

shape → tuple of integers representing the size of each of the placeholder tensor's dimensions

The first integer represents number of data points (ie rows). It is referred to as batch size
The second integer represents number of features (ie. number of columns) (input_size)
Each datapoint also has a label. The labels we use will have a two dimension shape, ~~[illegible]~~ with output_size as second dim.
output_size refers to the number of possible classes a label can have

Note - use of None in place of dimension size. When we use None in the shape tuple, we are allowing that dimension ~~to~~ to take any size.

Logits



The forces that drive the neural network are the weights attached to each connection. The weight on connection from neuron A to neuron B tells how strongly A affects B as well as whether the effect is positive or negative, ie direct vs inverse relationship

optimisation.

The weights that determine what a neural network outputs are called trainable variables. meaning that we need to train our neural network to find the optimal weights

For any neural network, training involves setting up a loss function, The loss function tells us how bad the neural network's output is

Loss as error:-

In regression problems, common loss functions are the L1 norm:-

$$\sum_i |actual_i - predicted_i|$$

and the L2 Norm:-

$$\sum_i (actual_i - predicted_i)^2$$

These provide an error metric for how far the predictions are from labels, so the goal is to minimise L1 and L2 norm

Cross entropy-

Rather than defining error as being right or wrong, we can instead define it interms of probability. We want a loss function that is small when the probability is close to the label and large when it is far. The loss function achieves this is known as cross entropy or ~~as~~ log loss

We can optimise the cross-entropy based on models logits. We do this through gradient descent, where the model updates its weights based on a gradient of loss function until it reaches minimum loss. Gradient descent is implemented as an object in tf as tf.train.GradientDescentOptimizer

The size of the gradient depends on something called learning rate. Usually learning rate is kept between 0.001 to 0.1.

We usually use Adam which is implemented in tf as tf.train.AdamOptimizer. It has default values already set for params (eg learning-rate)

```
→ #training example
   t = tf.constant([1,2,3])
   sess = tf.session()
   arr = sess.run(t)
   print(arr)
   t2 = tf.constant(4)
   tup = sess.run([t,t2])
   print(tup)
```

```
→ #example
inputs = tf.placeholder(tf.float32, shape=(None,2))
feed_dict = {
     inputs: [[0.1,-0.3],[0.2,0.1]]
}
```

```
Sess = tf.Session()
arr = sess.run(inputs, feed_dict = feed_dict)
print(arr)
```

→ When training from scratch, none of our variables have values yet. We need to initialise the variables.

```
inputs = tf.placeholder(tf.float32, shape=(None, 32))
feed_dict = { .... }
logits = tf.layers.dense(inputs, 1, name='logits')
init_op = tf.global_variables_initializer()

Sess = tf.session()
sess.run(init_op)
arr = sess.run(logits, feed_dict = feed_dict)
print(arr)
```

Hidden layes & Non linearity

We add non linearities to our model through activation functions. These are non linear functions that are applied within the neurons of a hidden layer.

The three most common activation functions in deep learning are tanh, ReLU and sigmoid. Each has its own uses in deep learning. However ReLU function works well in most general-purpose situations.

→ ~~#~~multilayer ~~class~~ multiclass classification

```
def model_layers(inputs, output_size):
    hidden1 = tf.layers.dense(inputs, 5,
                              activation = tf.nn.relu,
                              name = 'hidden1')
    logits = tf.layers.dense(hidden1, output_size,
                             name = 'logits')
    return logits
```

Softmax

Softmax function takes in a vector of numbers (logits for each class), and converts the numbers to a probability distribution

```
→ t = tf.constant([... ])
  softmax_t = tf.nn.softmax(t)
  sess = tf.session()
  print (sess.run(t), sess.run(softmax_t))
```

Predictions

```
→ probs = tf.constant([... ])
  preds = tf.argmax(probs, axis = -1)
  sess. tf. session()
  print(sess.run(probs))
  print(sess.run(preds))
```

Deep learning with keras

The most commonly used keras neural network is Dense layer. We can either initialise an empty Sequential object and add layers onto the model using the add function or we can directly initialise the sequential object with a list of layers.

```
→ model = Sequential()
  layer1 = Dense(5, input_dim = 4)
  model.add(layer1)
  layer2 = Dense(3, activation='relu')
  model.add(layer2)

→ layer1 = Dense(5, input_dim = 4)
  layer2 = Dense(3, activation = 'relu')
  model = Sequential([layer1, layer2])
```

- The Dense object takes in a single required argument, which is the number of neurons in the fully connected layer.

Model output

In Keras, the cross entropy loss functions only calculate cross-entropy, without applying the sigmoid/softmax function to the MLP output. Therefore we can have the model directly output class probabilities instead of logits

```
→ model= Sequential()  # for binary clasification
layer1 = Dense(5, activation='relu', input_dim=4)
model.add(layer1)
layer2 = Dense(1, activation='sigmoid')
model.add(Layer2)
```

```
→ # for multiclass (3)
model = Sequential()
layer1 = Dense(5, input_dim = 4)
model.add(layer1)
layer2 = Dense(3, activation='softmax')
model.add(layer2)
```

Model configuration.

compile function takes in a single required argument which is the optimiser to use. eg. 'adam', 'sgd', 'ada grad'.

The loss keyword argument specifies the loss function to use. for binary classification, we set it to 'binary-crossentropy'. for multiclass we set it to 'categorical-crossentropy'.

The metrics keyword argument takes in a list of strings, representing metrics we want to track during training & evaluation

```
→ model= Sequential()
layer1 = Dense(5, activation='relu', input_dim=4)
```

```
model.add(layer1)
layer2 = Dense(1, activation='sigmoid')
model.add(layer2)
model.compile('adam', loss='binary_crossentropy',
                      metrics=['accuracy'])
```

Model Execution

→ Training

The first two arguments of the fit function are input data and labels. We can use numpy arrays as input. The training batch size can be specified using batch_size (default is 32) We can also specify epochs ie number of full run-throughs of the dataset during training using the epochs keyword (default is 1)

```
train_output = model.fit(data, labels,
                         batch_size=20,
                         epochs=5)
```

The output is a History object which records the training metrics.

→ Evaluation

```
print(model.evaluate(data, labels))
```

→ Prediction

```
print(model.predict(new_data))
# will return the probability for each class
```