upGrad

Session 4: Neural Network Implementation Using Keras

Keras is a high-level library designed to work on top of Theano or TensorFlow. The main advantage of using Keras is that it is an easy-to-use, minimalistic API that can be used to build and deploy deep learning models quickly.

Model Building Using Keras

Six main steps are involved in building a model using Keras:

1. Load the data

We load the available data using the following function.

```
load_data()
```

Note: The shape of the matrices changes in the Keras implementation with respect to the NumPy implementation. Hence, we take the transpose of the matrices that we get after preprocessing on the load_data() matrices.

```
train_set_x = train_set_x.T
train_set_y = train_set_y.T
test_set_x = test_set_x.T
test_set_y = test_set_y.T
```

2. Define the model.

Typically, models in Keras are defined as a sequence of layers. So, we first need to create a model with a variable name, say, 'nn_model'.

```
nn_model = Sequential()
```

This model has no layers at this stage. We can add as many layers as we want to it. Let's add the first hidden layer. As we add a layer, we also need to specify the number of neurons in the layer and the activation function that they will use. You may have noticed that we used 'Dense', which is used to specify that the layers are fully connected, i.e., every neuron in one layer will be connected to every neuron in the next layer.

```
nn_model.add(Dense(35, input_dim=784, activation='relu'))
```

Here, as an example, '35' denotes the number of neurons in the hidden layer and '784' denotes the input size.

3. Compile the model.

After we have defined the model, we need to compile it. During this step, Keras uses the backend libraries to efficiently represent the model for training and prediction. We need to specify the loss function, the metrics as well as the optimiser in this step. In a classification problem, the loss function is the cross-entropy loss, which is 'categorical_loss' in Keras. The metrics and the optimiser we will use are 'accuracy' and 'adam' (like gradient descent, Adam optimiser helps to find the optimal parameters).

```
nn_model.compile(loss='categorical_crossentropy', optimizer='adam',
metrics=['accuracy'])
```

4. Fit the model.

The next step is to fit the model on the data set. You need to pass training set of x and y('train_set_x' and 'train_set_y') that you created earlier. You also need to pass the number of epochs that you want for the training to happen. Essentially, one epoch is one pass through the entire data set in mini-batches. So, you also need to specify the mini-batch size (the number of data points to be sent through the neural network in one go).

```
nn_model.fit(train_set_x, train_set_y, epochs=10, batch_size=32)
```

Note: Unless we call the 'model.fit()' function, the training does not begin.

5. Evaluate the model.

In this step, we can see the accuracy scores that we finally achieved using the following command.

```
scores_train = nn_model.evaluate(train_set_x, train_set_y)
print("\n%s: %.2f%%" % (nn_model.metrics_names[1],
scores_train[1]*100))
```

To get the score on the test data, we can write the following code:

```
scores_test = nn_model.evaluate(test_set_x, test_set_y)
print("\n%s: %.2f%%" % (nn_model.metrics_names[1],
scores_test[1]*100))
```

Note that we only changed the data set from train to test.

6. Make predictions.

Predictions can be performed using the '.predict()' command in the following manner.

```
predictions = nn_model.predict(test_set_x)
```

These are the steps involved in building a model in Keras. You must have observed that we need not write any code for feedforward or backpropagation as we did when we implemented the neural network using TensorFlow. Using Keras eliminates all those efforts. Keras is used almost everywhere because it is quite flexible.

Epoch, Batch, Overfitting and Underfitting

The main elements of the architecture other than the parameters of the architecture (input, weights, biases and output) are called hyperparameters.

Let's start with the term 'epoch'.

```
model.fit(X_train, y_train, batch_size=64, epochs=5, validation_data=(X_val,
y_val))
```

The number of epochs mentioned in the code snippet defines the number of times the learning algorithm will work through the entire data set. One epoch indicates that each training example has had an opportunity to update the internal model parameters, i.e., the weights and biases.

The term 'batch size' refers to the number of training examples utilised in one iteration. The model decides the number of examples to work with in each iteration before updating the internal model parameters.

```
model.fit(X_train, y_train, batch_size=32, epochs=5, validation_data=(X_val,
y_val))
```

Regularisation

Following are two points to keep in mind regarding training a model on a data set:

1. The model should be able to determine generalised trends in a proper manner for smarter predictions.
2. It should be able to apply these observations and trends to future data (the data that the model has never seen) and accurately make predictions.

To measure how the model is performing on these two bases, we assess whether the model may be overfitting or underfitting. If the model overfits, it will perform well on the training data set but not on the testing data set. If the model underfits, it will find it difficult to identify even the major patterns present in the data. If the model is not able to understand the underlying trend of the given data set, then it is said to be underfitting. Both underfitting and overfitting are issues that need to be addressed.

Neural networks that are usually large, complex models with tens of thousands of parameters have a tendency to **overfit** the training data. As with many other ML models, **regularisation** is a common technique used in neural networks to address this problem. Let's now take a look at a popular regularisation technique used for neural networks called **dropouts.**

Dropouts

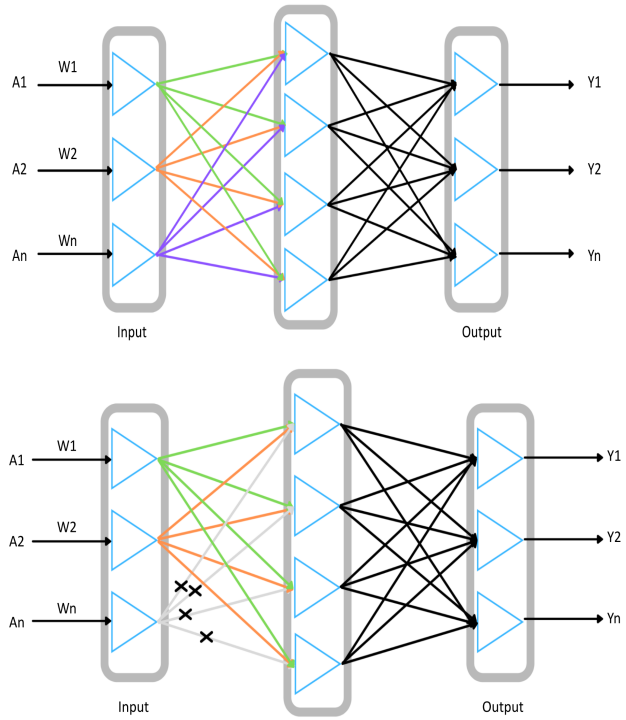
The main purpose of using dropouts is to reduce overfitting. Sometimes, a model trains on the training data set, and its weights and biases converge to very specific values that are ideal for only the training data set. Adding a dropout layer to the neural network helps to break that specific combination of weights and biases. This enables the neural network to search for a broader and more general pattern and makes predictions more robust.

The dropout operation is performed by multiplying the weight matrix W^l with an α **mask vector,** as shown below.

$$W^l . \alpha$$

You can see the differences between the ANN without dropout and the ANN with dropout in the image given below. Adding a dropout layer essentially removes the

links from the third neuron in the first layer to all the neurons in the next layer. The cross on the interconnections indicates that the interconnection has been removed.



Neural Network Without Dropout Layer Neural Network With Dropout Layer

Here are some important points to note regarding dropouts:

- Dropouts can be applied only to some layers of the network (in fact, this is a common practice; you choose some layer arbitrarily to apply dropouts to).
- The mask α is generated independently for each layer during feedforward, and the same mask is used in backpropagation.
- The mask changes with each mini-batch/iteration and is randomly generated in each iteration (sampled from a Bernoulli distribution with some p(1) = q).

Why the dropout strategy works well can be understood through the notion of a **manifold.** Manifold captures the observation that in high-dimensional spaces, data points often actually lie in a **lower-dimensional manifold.** This is observed experimentally and can be understood intuitively as well.

There are other ways to create the mask: One is to create a matrix (instead of a vector) that has 'q' percentage of the elements set to 1 and the rest set to 0. You can

$$\begin{bmatrix} w_{11}^1 & w_{12}^1 & w_{13}^1 \\ w_{21}^1 & w_{22}^1 & w_{23}^1 \\ w_{31}^1 & w_{32}^1 & w_{33}^1 \\ w_{41}^1 & w_{42}^1 & w_{43}^1 \end{bmatrix}$$

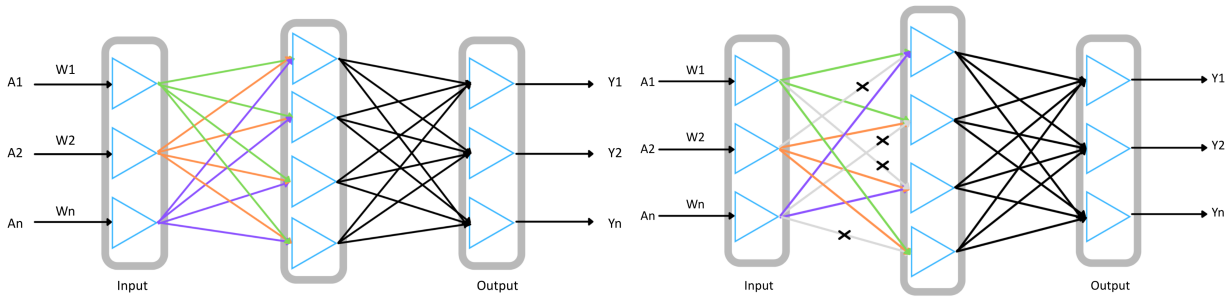
then multiply this matrix by the weight matrix element-wise to get the final weight matrix. Hence, for the weight matrix , the mask matrix for ‘q’ = 0.66 can be as given below.

$$\begin{bmatrix} 1 & 0 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 1 & 0 \end{bmatrix}$$

Multiplying the aforementioned matrices element-wise, we get the following matrix.

$$\begin{bmatrix} w_{11}^1 & 0 & w_{13}^1 \\ w_{21}^1 & w_{22}^1 & 0 \\ 0 & w_{32}^1 & w_{33}^1 \\ w_{41}^1 & w_{42}^1 & 0 \end{bmatrix}$$

Visually, this matrix can be shown as given below.



Neural Network Without Dropout Layer Neural Network With Dropout Layer Above

In order to implement dropouts, you just need to write one simple line of code to add a dropout layer in Keras. You can write the line of code mentioned below.

```
# dropping out 20% neurons in a layer in Keras
model.add(Dropout(0.2))
```

Listed below are important points to note regarding the implementation of dropouts:

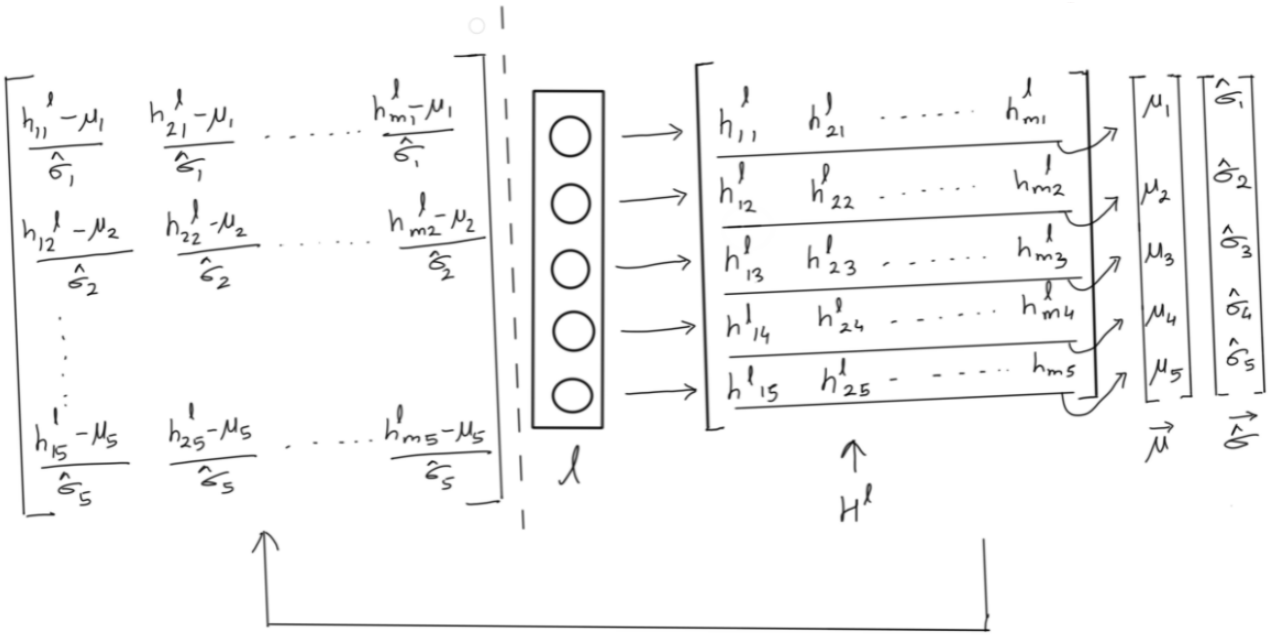
1) Here, '0.2' is the **probability of zeros** and not ones.
2) This is one of the hyperparameters to be experimented with when building a neural network.
3) You do not apply dropout to the output layer.
4) The mask used here is a matrix.
5) Dropout is applied only during training, not while testing.

Batch Normalisation

It is generally a good idea to have your data on a common scale while training a neural network. Sometimes, when training a neural network, large activations might be produced. Different sizes of activations can result in unstable training behaviour. This is where 'normalisation' can be helpful. Normalisation is seen as an aid to the optimisation process. Commonly seen benefits are that fewer epochs are required to complete the network's training process, and sometimes, it avoids the neural network to get stuck during the training process.

Batch normalisation is performed on the output of the layers of each batch, H^l. It essentially involves normalising the matrix H^l across all data points in the batch. Each vector in H^l is normalised by the mean vector μ and the standard deviation vector $\hat{\sigma}$ computed across a batch.

The image given below shows the batch normalisation process for layer l. Each column in the matrix H^l represents the output vector of layer l, h^l, for each of the 'm' data points in the batch. We compute the μ and $\hat{\sigma}$ vectors, which represent the 'mean output from layer l across all points in the batch'. We then normalise each column of the matrix H^l using μ and $\hat{\sigma}$.



Hence, if a particular layer l has five neurons, we will have H^l of the shape (5, m), wherein 'm' is the batch size, and the μ and $\hat{\sigma}$ vectors are of shape (5, 1). The first element of μ (μ1) is the mean of the outputs of the first neuron for all the 'm' data points, the second element μ2 is the mean of the outputs of the second neuron for all the 'm' data points and so on. Similarly, we get the vector $\hat{\sigma}$ as the standard deviation of the outputs of the five neurons across the 'm' points. The normalisation step then becomes:

$$H^l = \frac{H^l - \mu}{\hat{\sigma}}$$

The final H^l after batch normalisation is shown on the left side of the image given above.

Batch normalisation might not be useful in the neural networks that you have learnt about so far. Batch normalisation is usually used in large architectures that train over a variety of data and solve complex problems. You will understand the use of the batch normalisation layer when you are introduced to the concept of Convolutional Neural Networks in another module.