SPARK – A NEURAL NETWORK DESCRIPTION LANGUAGE

Spark is the name of the language created as a result of this project. The language allows users to create arbitrary sized neural networks easily and to reuse previously trained components with new networks. This section describes the various features of the language and attempts to demonstrate its power.

Create a layer:

A layer is the basic unit of computation in Spark. Unlike in conventional neural networks in which the neuron is the most basic unit of computation, Spark works directly at the level of layers in order to make use of efficient matrix operations and decrease the time required to train a network.

In order to create a layer and assign it to a variable we supply various parameters to the *layer* command as shown in Figure 1.

```
inputLayer = layer with id = inp, length = 4;
hiddenLayer = layer with id = hid, length = 10, activationFunction = Tan
Sigmoid, weightInitializer = RandSmall;
outputLayer = layer with id = out, length = 3, activationFunction = Soft
Max, weightInitializer = RandSmall;
```

Figure 1: Creating a layer

As can be seen, executing the above commands creates three layer namely *inputLayer, hiddenLayer and outputLayer*. Each layer has different parameters supplied to it.

The various parameter that can be supplied to a layer are given below:

- *Id* – It is a user given string used to identify the layer when printing a network / layer (as we will see later). The id should describe the function of the layer.
- *Length* – The most important parameter for a layer, it tells the interpreter the length of the layer to create. This parameter determines the number of neurons in the layer i.e. length of layer = number of neurons in layer.
- *ActivationFunction* – The activation function of each neuron in the layer. It is an optional parameter which can take one of several predefined values such as:
    1. *Linear (default)* – Pass the inputs without modification.
    2. *TanSigmoid* – A variation of the sigmoid function.

3. *Sigmoid* – Already discussed in previous sections.
4. *SoftMax* – A probabilistic activation function mainly used for classification networks in the output layer.
5. *Thresholder* – The simple perceptron activation function.
6. *BipolarThresholder* – A modification of the *Thresholder* for bipolar inputs and outputs.

- *InputFunction* – The input function to be used by each neuron in the layer. It is an optional parameter which can take one of the following values:
  1. *Sum (default)* – Finds the sum of all inputs.
  2. *Product* – Multiplies all inputs.
- *WeightFunction* – The weight function to be used by each neuron in the layer. It is an optional parameter which can take currently take only one value i.e. *DotProduct* which multiplies each input to a weight to the output neurons.
- *WeightInitializer* – The function to use to initialize weights between two layer (when creating a matrix). It is also an optional parameter which can take one of the following parameters.
  1. *InitZero (default)* – Initialize the weights to zero.
  2. *RandSmall* – Initialize the weights to small random values in the range (0 – 0.2)

Layer can be used as building block to create neural networks, hence we can use the same neural network more than once in different networks. We can see the properties of out neural network by printing it using the *print* command (Figure 2).

```
print inputLayer;
```

Figure 2: Print a layer

The output of the command can be seen in Figure 3.

```
Code succesfully evaluated.
Layer Id: inp
Length: 4
Activation Function: Linear
Weight Function: DotProduct
Input Function: Sum
Weight Initializer: InitZero
```

Figure 3: Output of print statement

Create a network:

A neural network can be created by joining individual layers in a feed forward manner. A single network cannot two networks with the same id, although you can create recurrent connections. Unfortunately due to the scope of this project, recurrent networks cannot be run or trained.

Creating a simple feed forward network is illustrated in Figure 4.

```
fNet = inputLayer -> hiddenLayer -> outputLayer;
```

Figure 4: Create a feed forward network

You can view the net by printing it with the *print* command as shown in Figure 5.

```
print fNet;
```

Figure 5: Print a feed forward network

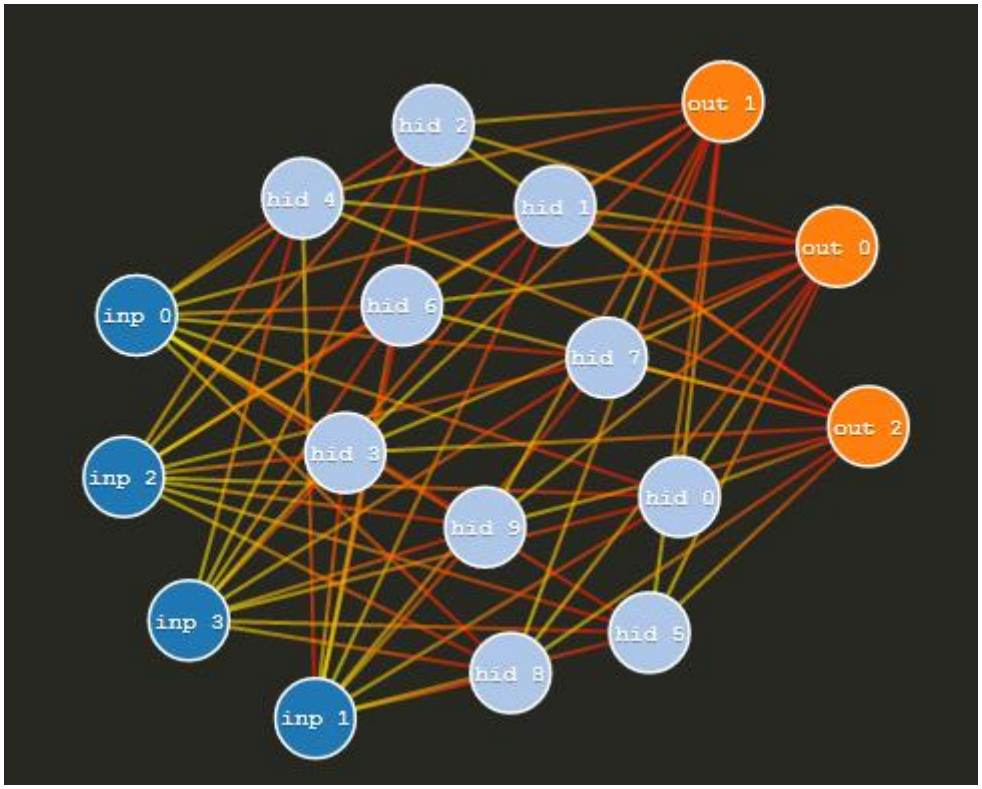

Figure 6: Feed forward network printed

Similarly we can create recurrent networks too as shown in Figure 7

```
recNet = inputLayer -> hiddenLayer -> hiddenLayer -> outputLayer;
```

Figure 7: Create a recurrent network.

Here we see that the *hiddenLayer* is connected to itself in a self-recurrent fashion. The network is shown in Figure 8.

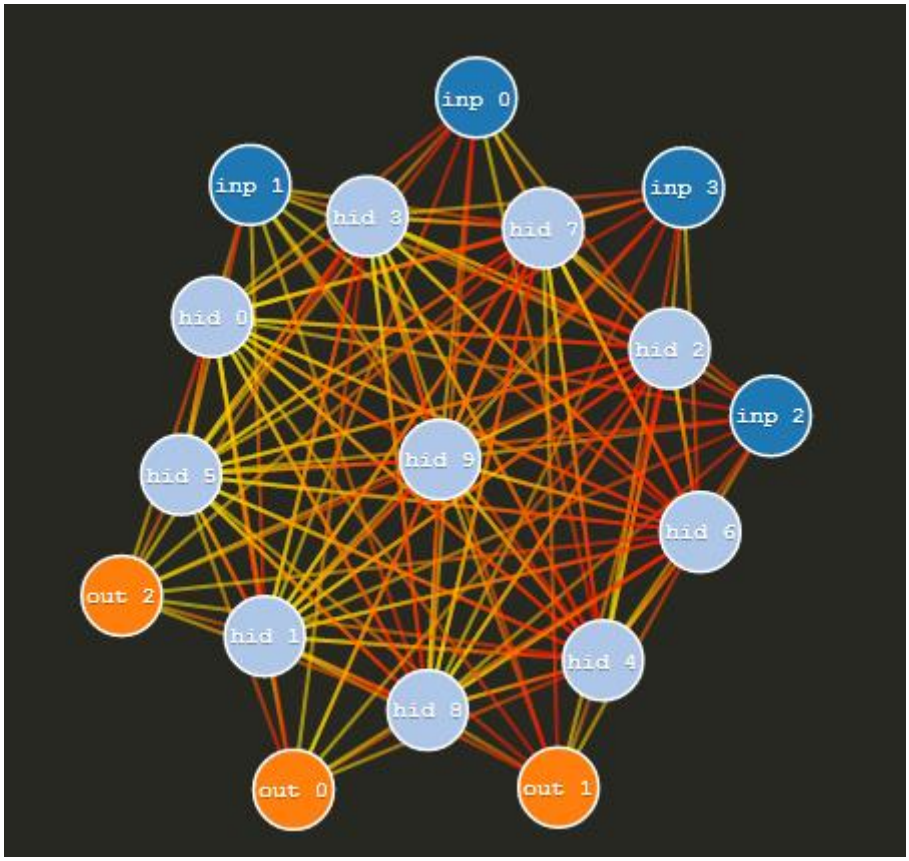

Figure 8: Recurrent network

The network seems too clustered to be able to make sense of, so let's create a simple single layer recurrent network (Figure 9).

```
rec3 = inputLayer -> inputLayer;
print rec3;
```

Figure 9: Single layer recurrent network

The network created is shown in Figure 10.

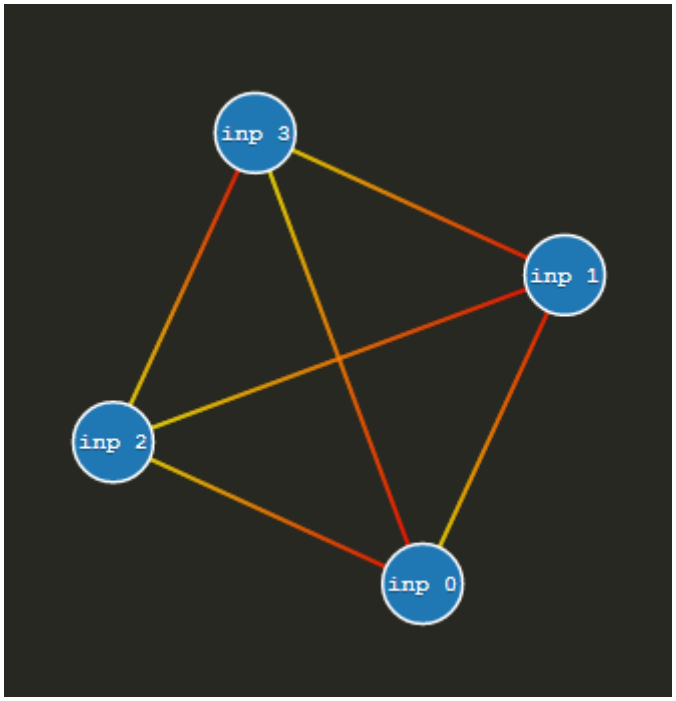

Figure 10: Single Layer recurrent network printed

As can be seen the main operator used to create a network is the → operator, tells the interpreter to connect the two layers. The → operator works by making the layer on the left an input layer, while the layer on the right is made the output layer.

Since the operator is a binary operator, by chaining the operator we may create arbitrary sized network with any number of hidden layers. The → operator is overloaded in order to enable it to take both *Networks* and *Layers* as arguments. The mode of operation is described in Table 1.

Table 1: → operator modes of operation

Left Argument	Right Argument	Result
Layer	Layer	Network with left argument as input layer and right argument as output layer.
Network	Layer	Right argument is appended to the left argument as output layer and is returned.
Layer	Network	Left argument is appended as input layer to right argument and returned.

Network	Network	A new network with the right argument appended to the left argument in a feed forward manner.

Import Data from file:

In order to train or run the network we need some sort of inputs and / or targets. Most real life neural networks deal with thousands of input vectors. These are not typed manually by the user and normally data-minded from external sources like the internet. Therefore, the neural network is fed data which is stored in files on the system.

Spark allows you to read such data files and load it into memory in order to feed to the neural network as shown in Figure 11.

```
irisInputs = load csv @"C:\Users\Likhit\Downloads\Datasets\Iris\inputs.csv
" as row major with headers;
irisTargets = load csv @"C:\Users\Likhit\Downloads\Datasets\Iris\targets.c
sv" as row major;
```

Figure 11: Loading input and target matrices

As you can see the load command takes four arguments:

- *File type* – Specifies what parser to use to read the file. There are three file types supported:
    1. *CSV (default)* – Comma separated values
    2. *TSV* – Tab separated values
    3. *SSV* – Space separated values
- *File path* – Specifies the path of the file to load on the user's system. The path is enclosed it double quotes prefixed by an @ symbol.
- *Matrix type* – Specifies the type of matrix stored in the file i.e.
    1. *Row major* – A row major matrix has each input vector stored as one row of the file.
    2. *Column major (default)* – A column major matrix has each input vector stored as one column of the file.

- *Headers* – Specifies if there are headers in the file or not. If so, the headers are discarded from the matrix and stored separately. The headers option can only be supplied when the matrix type is *column major*.

An example of *row major* matrix is given in Table 1, and for a *column major* if given in Table 2.

Table 2: Row major matrix with header

Sepal Length(cm)	Sepal Width(cm)	Petal Length(cm)	Petal Width(cm)
5.1	3.5	1.4	0.2
4.9	3	1.4	0.2
4.7	3.2	1.3	0.2
4.6	3.1	1.5	0.2
5	3.6	1.4	0.2
5.4	3.9	1.7	0.4
4.6	3.4	1.4	0.3
5	3.4	1.5	0.2
4.4	2.9	1.4	0.2

Table 3: Column major matrix

14.23	13.2	13.16	14.37	13.24	14.2
1.71	1.78	2.36	1.95	2.59	1.76
2.43	2.14	2.67	2.5	2.87	2.45
15.6	11.2	18.6	16.8	21	15.2

127	100	101	113	118	112
2.8	2.65	2.8	3.85	2.8	3.27
3.06	2.76	3.24	3.49	2.69	3.39
0.28	0.26	0.3	0.24	0.39	0.34
2.29	1.28	2.81	2.18	1.82	1.97
5.64	4.38	5.68	7.8	4.32	6.75
1.04	1.05	1.03	0.86	1.04	1.05
3.92	3.4	3.17	3.45	2.93	2.85
1065	1050	1185	1480	735	1450

We can also print the table in the interpreter shell for easy viewing and verifying as shown in Figure 12.

```
Code succesfully evaluated.
```

#	Sepal Length(cm)	Sepal Width(cm)	Petal Length(cm)	Petal Width(cm)
0	5.1	3.5	1.4	0.2
1	4.9	3	1.4	0.2
2	4.7	3.2	1.3	0.2
3	4.6	3.1	1.5	0.2
4	5	3.6	1.4	0.2
5	5.4	3.9	1.7	0.4
6	4.6	3.4	1.4	0.3
7	5	3.4	1.5	0.2
8	4.4	2.9	1.4	0.2
9	4.9	3.1	1.5	0.1
10	5.4	3.7	1.5	0.2

Figure 12: Input data printed to interpreter shell

Create a trainer:

A trainer can be created to train a network with predefined parameter and algorithm. Figure 13 shows how we can create a new trainer.

```
irisTrainer = trainer of type BackPropogationTrainer with learnRate = 0.
01, maxEpochs = 300, minError = 0.02, show = 10;
```

Figure 13: Create a trainer

A trainer takes a number of parameters. The exact parameters varies from algorithm to algorithm, but some of them are common to all:

- *Type* – Specifies the algorithm to use to train the network. Can be:
    1. *BackPropagationTrainer*
    2. *ConstructiveTrainer*
- *Learn rate* – Specifies the learn rate to be used during weight update.
- *Max Epochs* – Specifies the maximum number of iterations to train the network.
- *Min Error* – Specifies the minimum value of tolerable error.
- *Show* – Specifies after how many iterations should we display the "current" state of the network (during training).

Train a network:

After we created the network, loaded the inputs and targets and created a trainer we can finally train the network to learn the presented patterns. Let us train the feed forward network created previously (*fNet)* using our *irisInputs* and *irisTargets* along with the *irisTrainer*. The command for training the network in such a way can be seen in Figure 14.

```
train fNet with irisTrainer on inputs irisInputs and targets irisTargets
;
```

Figure 14: Train a network

Training the network will print a line graph to the interpreter shell which shows in real time the current state of training (as specified by the show parameter). This is shown in Figure 15.

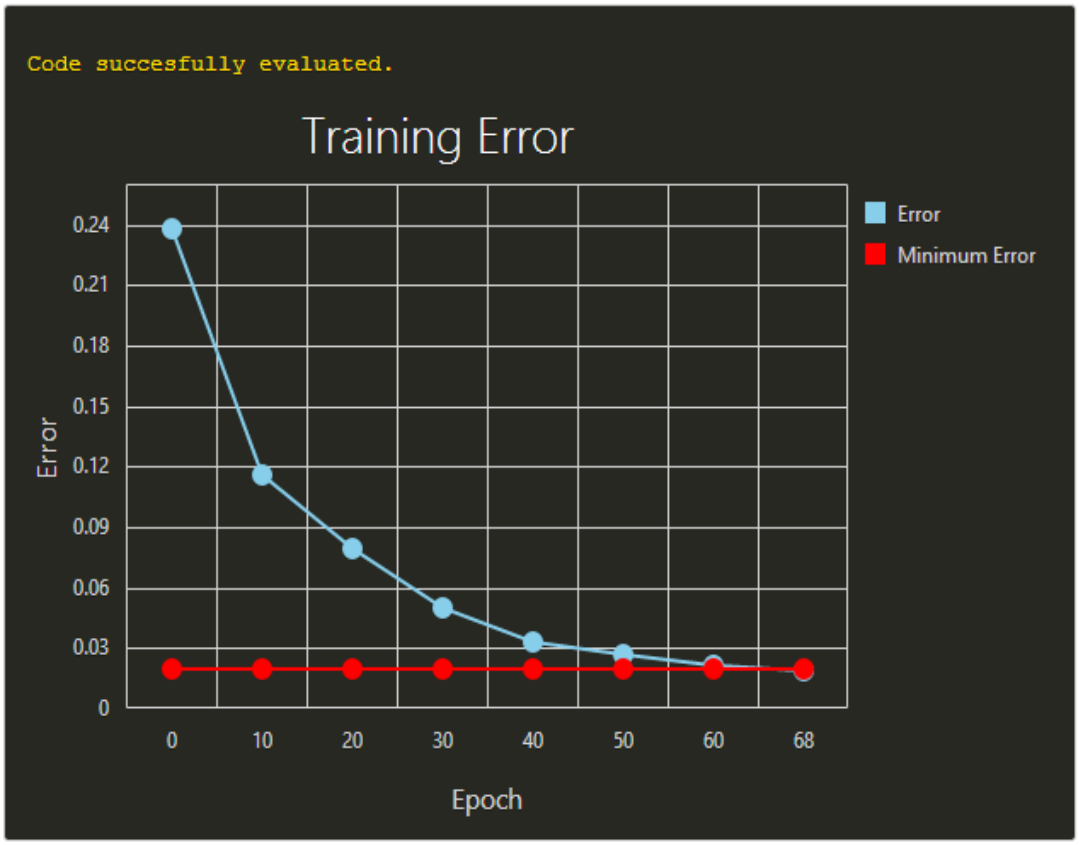

Figure 15: Training output

After the training finishes, the network's weights would have been updated to have learnt a part or whole of the training data.

Running the network:

The network can be run on any inputs to see the outputs of the network. This can be done using the run command as shown in Figure 16.

```
outputs = run fNet on irisInputs;
```

Figure 16: Running the trained network on inputs

This stores the outputs of the network (as a result of using *irisInputs* as inputs) to the variable outputs. We can print these outputs using *print* command (see Figure 17).

Code succesfully evaluated.

#	Column 0	Column 1	Column 2
0	0.9595811949221923	0.07509598565532663	-0.10736723692207581
1	0.9517228207201076	0.08331670630954327	-0.10735856540163882
2	0.9574580358340068	0.07730458960209047	-0.10736457726516484
3	0.9491697953719944	0.08601318772829641	-0.107355016158917
4	0.960493269914143	0.0741496755259194	-0.10736808864557269
5	0.956673524319519	0.07812346139494875	-0.1073641081334566
6	0.9566225964654631	0.07817586228962982	-0.1073632777412344
7	0.9557885384178124	0.0790482328235175	-0.10736313078664599
8	0.9480202430280185	0.08723130512992103	-0.10735315627045369
9	0.9514837008420228	0.08356906325017077	-0.10735858423131554
10	0.9602258177236072	0.07442728594529846	-0.10736813019858085

Figure 17: Outputs of the network

Converting the format of a matrix:

Often we have our target or input matrix encoded in one form when we want to use it in a different form. In neural networks this is often the case with representation of classes. Two forms are used to represent the network as shown in Table 4.

Table 4: Representation of classes

Indices	Vector		
1	1	0	0
2	0	1	0
3	0	0	1

The indices technique is more human readable and intuitive, while the vector technique is more apt for the machine. It is therefore necessary to be able to convert between the two so that both the human and the machine can read it easily. This interchange can be done using the following two commands (Figure 18, 19):

```
outputInds = convert outputs to class indices;
targetInds = convert irisTargets to class indices;
```

Figure 18: Convert vector representation to indices representation

```
outputMatrix = convert outputInds to class vector;
```

Figure 19: Convert indices representation to vector representation

The outputs of 18 can be found in Figure 20. The variable *outputMatrix* will be equivalent to the variable *outputs*.

```
Code succesfully evaluated.
```

#	
0	0
1	0
2	0
3	0
4	0
5	0
6	0
7	0
8	0
9	0
10	0

Figure 20: Converted indices representation

Calculating error:

After training, often we want to find out the error or number of mismatches. A mismatch is a pattern or input vector which doesn't give the right output i.e. output and target do not match. Getting the error helps us determine the usefulness of the training algorithm and the network. The mismatches can be calculated using the two commands illustrated in Figures 4.21 and 4.22.

```
err = count mismatches between outputInds and targetInds;
```

Figure 21: Count the number of mismatches between targets and outputs

```
misMatches = get mismatches between outputInds and targetInds;
```

Figure 22: Get the mismatched vectors

We can print the above two variables to get a look at their values (Figure 23).

```
Code succesfully evaluated.

Number of mismatches: 3
```

#	outputInds	targetInds
70	2	1
72	2	1
83	2	1

Figure 23: Values of the above defined variables

As we can see, we get the index of the output vector which was mismatched. Hence we can trace it back to the input vector which was mismatched quite easily.

Saving the outputs:

Once we are done and have gotten out desired outputs, we can save back those outputs to a file on our disc using the save command. The save command is similar to the load command but in reverse.

The only difference being that the save command always stores in column major form, and hence has no headers.

The save command is illustrated in Figure 24.

```
save outputs as csv to @"C:\Users\Likhit\Downloads\Datasets\Iris\outputs
.csv";
```

Figure 24: Save command

If the command was executed properly, it gives a *File successfully saved* notification.

Spark Internals:

The spark language is implemented using the context free grammar specified below. The actual algorithms and networks are all implemented in *c#.*

program → <*statement*> ;

| <*statement*> ; <*program*>

statement → <*assignStmt*>

| <*commandStmt*>

assignStmt → <*identifier*> = <*assignRHS*>

assignRHS → <*createLayer*>

| <*createNetwork*

| <*createTrainer*>

| <*runNetwork*>

| <*loadFile*>

| <*convertData*>

| <*findError*>

createLayer → *layer with* <*kvPair*>;

createNetwork → <*basicNetwork*>

| <*advancedNetwork*>

basicNetwork → <*identifier*> "->" <*identifier*>

advancedNetwork → <*createNetwork*> "->" <*identifier*>

runNetwork → *run* <*identifier*> *on* <*identifier*>

loadFile → *load* <*path*> *as* <*matrixLoadType*>

| load <supportedFileTypes> <path> as <matrixLoadType>

matrixLoadType → row major

| row major with headers

| column major

createTrainer → trainer of type <identifier> with <kvPair>

kvPair → <identifier> = (<number> | <identifier>)

| identifier = (number | identifier) , <kvPair>

convertData → <convertToVector>

| <convertToIndices>

convertToVector → convert <identifier> to class vector

| convert <identifier> to class vector of size = <number>

convertToIndices → convert <identifier> to class indices

findError → count mismatches between <identifier> and <identifier>

| get mismatches between <identifier> and <identifier>

commandStmt → <printStatement>

| <trainStatement>

| <saveStatement>

printStatement → print <identifier>

trainStatement → train <identifier> with <identifier> on inputs <identifier> and targets <identifier>

saveStatement → save <identifier> to <path>

| save <identifier> as <supportedFileTypes> to <path>

supportedFileTypes → csv | tsv | ssv