Deep Learning using Tensor Flow

Tensor Flow Feedforward NN

```
import numpy as np
# import data
from keras.datasets import mnist
import tensorflow as tf

mnist_data = mnist.load_data()
```

Load Mnist data

Mnist dataset is a tuple containing training and testing data.

Mnist = ([training_data], [testing_data])

[training_data] and [testing_data] are both tuple of length 2 which contains the hand written images (X) and labels for each image (y)

[training_data] = ([X_train], [y_train])
[testing_data] = ([X_test], [y_test])

Then, [X_train], [y_train], [X_test] and [y_test] are arrays with the following shape:

[X_train] = (60000, 28, 28)
, which means an array with 60000 images and each image is an matrix of 28x28

$$\left[\begin{bmatrix} 0 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & 0 \end{bmatrix} \quad \begin{bmatrix} 0 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & 0 \end{bmatrix} \quad (\ldots) \quad \begin{bmatrix} 0 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & 0 \end{bmatrix}\right] \longrightarrow \begin{bmatrix} 0 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & 0 \end{bmatrix}$$

60000 matrices that represents the hand written images

A matrix that represents one hand written image of 28x28 pixels

[y_train] = (60000,)
,which means an array with 10000 labels of the hand written images

$$\begin{bmatrix} 1 & 8 & 5 & 2 & 3 & 4 & (\ldots) & 7 & 0 \end{bmatrix}$$

The same idea is applied for:

[X_test] = (10000, 28, 28)
[y_test] = (10000,)

```
# load data
(X_train, y_train), (X_test, y_test) = mnist.load_data()
```

Split the Mnist data into training and testing sets

We still need to do some data manipulation:

Defining a seed to be able to replicate the results again

```
np.random.seed(0)
train_indices = np.random.choice(60000, 50000, replace=False)
valid_indices = [i for i in range(60000) if i not in train_indices]

X_valid, y_valid = X_train[valid_indices,:,:], y_train[valid_indices]
X_train, y_train = X_train[train_indices,:,:], y_train[train_indices]
print(X_train.shape, X_valid.shape, X_test.shape)
# (50000, 28, 28) (10000, 28, 28) (10000, 28, 28)

image_size = 28
num_labels = 10

def reformat(dataset, labels):
    dataset = dataset.reshape((-1, image_size * image_size)).astype(np.float32)
    # one hot encoding: Map 1 to [0.0, 1.0, 0.0 ...], 2 to [0.0, 0.0, 1.0 ...]
    labels = (np.arange(num_labels) == labels[:,None]).astype(np.float32)
    return dataset, labels

X_train, y_train = reformat(X_train, y_train)
X_valid, y_valid = reformat(X_valid, y_valid)
X_test, y_test = reformat(X_test, y_test)
print('Training set', X_train.shape, y_train.shape)
print('Validation set', X_valid.shape, y_valid.shape)
print('Test set', X_test.shape, y_test.shape)
# Training set (50000, 784) (50000, 10) # Validation set (10000, 10) (10000, 10) # Test set (10000, 784) (10000, 10)
```

We still need to do some data manipulation:

```
np.random.seed(0)
train_indices = np.random.choice(60000, 50000, replace=False)
valid_indices = [i for i in range(60000) if i not in train_indices]

X_valid, y_valid = X_train[valid_indices,:,:], y_train[valid_indices]
X_train, y_train = X_train[train_indices,:,:], y_train[train_indices]
print(X_train.shape, X_valid.shape, X_test.shape)
# (50000, 28, 28) (10000, 28, 28) (10000, 28, 28)

image_size = 28
num_labels = 10

def reformat(dataset, labels):
  dataset = dataset.reshape((-1, image_size * image_size)).astype(np.float32)
  # one hot encoding: Map 1 to [0.0, 1.0, 0.0 ...], 2 to [0.0, 0.0, 1.0 ...]
  labels = (np.arange(num_labels) == labels[:,None]).astype(np.float32)
  return dataset, labels

X_train, y_train = reformat(X_train, y_train)
X_valid, y_valid = reformat(X_valid, y_valid)
X_test, y_test = reformat(X_test, y_test)
print('Training set', X_train.shape, y_train.shape)
print('Validation set', X_valid.shape, y_valid.shape)
print('Test set', X_test.shape, y_test.shape)
# Training set (50000, 784) (50000, 10) # Validation set (10000, 10) (10000, 10) # Test set (10000, 784) (10000, 10)
```

Chosing 50000 numbers without repetition between 0 to 59999

We still need to do some data manipulation:

```
np.random.seed(0)
train_indices = np.random.choice(60000, 50000, replace=False)
valid_indices = [i for i in range(60000) if i not in train_indices]

X_valid, y_valid = X_train[valid_indices,:,:], y_train[valid_indices]
X_train, y_train = X_train[train_indices,:,:], y_train[train_indices]
print(X_train.shape, X_valid.shape, X_test.shape)
# (50000, 28, 28) (10000, 28, 28) (10000, 28, 28)

image_size = 28
num_labels = 10

def reformat(dataset, labels):
  dataset = dataset.reshape((-1, image_size * image_size)).astype(np.float32)
  # one hot encoding: Map 1 to [0.0, 1.0, 0.0 ...], 2 to [0.0, 0.0, 1.0 ...]
  labels = (np.arange(num_labels) == labels[:,None]).astype(np.float32)
  return dataset, labels

X_train, y_train = reformat(X_train, y_train)
X_valid, y_valid = reformat(X_valid, y_valid)
X_test, y_test = reformat(X_test, y_test)
print('Training set', X_train.shape, y_train.shape)
print('Validation set', X_valid.shape, y_valid.shape)
print('Test set', X_test.shape, y_test.shape)
# Training set (50000, 784) (50000, 10) # Validation set (10000, 10) (10000, 10) # Test set (10000, 784) (10000, 10)
```

Chosing 10000 numbers without repetition which train_indices didnt pick up

We still need to do some data manipulation:

```
np.random.seed(0)
train_indices = np.random.choice(60000, 50000, replace=False)
valid_indices = [i for i in range(60000) if i not in train_indices]

X_valid, y_valid = X_train[valid_indices,:,:], y_train[valid_indices]
X_train, y_train = X_train[train_indices,:,:], y_train[train_indices]
print(X_train.shape, X_valid.shape, X_test.shape)
# (50000, 28, 28) (10000, 28, 28) (10000, 28, 28)

image_size = 28
num_labels = 10

def reformat(dataset, labels):
    dataset = dataset.reshape((-1, image_size * image_size)).astype(np.float32)
    # one hot encoding: Map 1 to [0.0, 1.0, 0.0 ...], 2 to [0.0, 0.0, 1.0 ...]
    labels = (np.arange(num_labels) == labels[:,None]).astype(np.float32)
    return dataset, labels

X_train, y_train = reformat(X_train, y_train)
X_valid, y_valid = reformat(X_valid, y_valid)
X_test, y_test = reformat(X_test, y_test)
print('Training set', X_train.shape, y_train.shape)
print('Validation set', X_valid.shape, y_valid.shape)
print('Test set', X_test.shape, y_test.shape)
# Training set (50000, 784) (50000, 10) # Validation set (10000, 10) (10000, 10) # Test set (10000, 784) (10000, 10)
```

Creating X and y valid arrays by array indices

We still need to do some data manipulation:

```
np.random.seed(0)
train_indices = np.random.choice(60000, 50000, replace=False)
valid_indices = [i for i in range(60000) if i not in train_indices]

X_valid, y_valid = X_train[valid_indices,:,:], y_train[valid_indices]
X_train, y_train = X_train[train_indices,:,:], y_train[train_indices]
print(X_train.shape, X_valid.shape, X_test.shape)
# (50000, 28, 28) (10000, 28, 28) (10000, 28, 28)

image_size = 28
num_labels = 10

def reformat(dataset, labels):
  dataset = dataset.reshape((-1, image_size * image_size)).astype(np.float32)
  # one hot encoding: Map 1 to [0.0, 1.0, 0.0 ...], 2 to [0.0, 0.0, 1.0 ...]
  labels = (np.arange(num_labels) == labels[:,None]).astype(np.float32)
  return dataset, labels

X_train, y_train = reformat(X_train, y_train)
X_valid, y_valid = reformat(X_valid, y_valid)
X_test, y_test = reformat(X_test, y_test)
print('Training set', X_train.shape, y_train.shape)
print('Validation set', X_valid.shape, y_valid.shape)
print('Test set', X_test.shape, y_test.shape)
# Training set (50000, 784) (50000, 10) # Validation set (10000, 10) (10000, 10) # Test set (10000, 784) (10000, 10)
```

Creating X and y train arrays by array indices

We still need to do some data manipulation:

```
np.random.seed(0)
train_indices = np.random.choice(60000, 50000, replace=False)
valid_indices = [i for i in range(60000) if i not in train_indices]

X_valid, y_valid = X_train[valid_indices,:,:], y_train[valid_indices]
X_train, y_train = X_train[train_indices,:,:], y_train[train_indices]
print(X_train.shape, X_valid.shape, X_test.shape)
# (50000, 28, 28) (10000, 28, 28) (10000, 28, 28)

image_size = 28
num_labels = 10

def reformat(dataset, labels):
  dataset = dataset.reshape((-1, image_size * image_size)).astype(np.float32)
  # one hot encoding: Map 1 to [0.0, 1.0, 0.0 ...], 2 to [0.0, 0.0, 1.0 ...]
  labels = (np.arange(num_labels) == labels[:,None]).astype(np.float32)
  return dataset, labels

X_train, y_train = reformat(X_train, y_train)
X_valid, y_valid = reformat(X_valid, y_valid)
X_test, y_test = reformat(X_test, y_test)
print('Training set', X_train.shape, y_train.shape)
print('Validation set', X_valid.shape, y_valid.shape)
print('Test set', X_test.shape, y_test.shape)
# Training set (50000, 784) (50000, 10) # Validation set (10000, 10) (10000, 10) # Test set (10000, 784) (10000, 10)
```

Defining some variables which we already know

We still need to do some data manipulation:

```
np.random.seed(0)
train_indices = np.random.choice(60000, 50000, replace=False)
valid_indices = [i for i in range(60000) if i not in train_indices]

X_valid, y_valid = X_train[valid_indices,:,:], y_train[valid_indices]
X_train, y_train = X_train[train_indices,:,:], y_train[train_indices]
print(X_train.shape, X_valid.shape, X_test.shape)
# (50000, 28, 28) (10000, 28, 28) (10000, 28, 28)

image_size = 28
num_labels = 10

def reformat(dataset, labels):
  dataset = dataset.reshape((-1, image_size * image_size)).astype(np.float32)
  # one hot encoding: Map 1 to [0.0, 1.0, 0.0 ...], 2 to [0.0, 0.0, 1.0 ...]
  labels = (np.arange(num_labels) == labels[:,None]).astype(np.float32)
  return dataset, labels

X_train, y_train = reformat(X_train, y_train)
X_valid, y_valid = reformat(X_valid, y_valid)
X_test, y_test = reformat(X_test, y_test)
print('Training set', X_train.shape, y_train.shape)
print('Validation set', X_valid.shape, y_valid.shape)
print('Test set', X_test.shape, y_test.shape)
# Training set (50000, 784) (50000, 10) # Validation set (10000, 10) (10000, 10) # Test set (10000, 784) (10000, 10)
```

This part will transform the image's matrix (2-D array) into a 1-D vector and the image's labels into one hot encoding

We still need to do some data manipulation:

```
np.random.seed(0)
train_indices = np.random.choice(60000, 50000, replace=False)
valid_indices = [i for i in range(60000) if i not in train_indices]

X_valid, y_valid = X_train[valid_indices,:,:], y_train[valid_indices]
X_train, y_train = X_train[train_indices,:,:], y_train[train_indices]
print(X_train.shape, X_valid.shape, X_test.shape)
# (50000, 28, 28) (10000, 28, 28) (10000, 28, 28)

image_size = 28
num_labels = 10

def reformat(dataset, labels):
  dataset = dataset.reshape((-1, image_size * image_size)).astype(np.float32)
  # one hot encoding: Map 1 to [0.0, 1.0, 0.0 ...], 2 to [0.0, 0.0, 1.0 ...]
  labels = (np.arange(num_labels) == labels[:,None]).astype(np.float32)
  return dataset, labels

X_train, y_train = reformat(X_train, y_train)
X_valid, y_valid = reformat(X_valid, y_valid)
X_test, y_test = reformat(X_test, y_test)
print('Training set', X_train.shape, y_train.shape)
print('Validation set', X_valid.shape, y_valid.shape)
print('Test set', X_test.shape, y_test.shape)
# Training set (50000, 784) (50000, 10) # Validation set (10000, 10) (10000, 10) # Test set (10000, 784) (10000, 10)
```

The final dimesion is expected to be (number_of_samples, image_size*image_size)

We still need to do some data manipulation:

```
np.random.seed(0)
train_indices = np.random.choice(60000, 50000, replace=False)
valid_indices = [i for i in range(60000) if i not in train_indices]

X_valid, y_valid = X_train[valid_indices,:,:], y_train[valid_indices]
X_train, y_train = X_train[train_indices,:,:], y_train[train_indices]
print(X_train.shape, X_valid.shape, X_test.shape)
# (50000, 28, 28) (10000, 28, 28) (10000, 28, 28)

image_size = 28
num_labels = 10

def reformat(dataset, labels):
  dataset = dataset.reshape((-1, image_size * image_size)).astype(np.float32)
  # one hot encoding: Map 1 to [0.0, 1.0, 0.0 ...], 2 to [0.0, 0.0, 1.0 ...]
  labels = (np.arange(num_labels) == labels[:,None]).astype(np.float32)
  return dataset, labels

X_train, y_train = reformat(X_train, y_train)
X_valid, y_valid = reformat(X_valid, y_valid)
X_test, y_test = reformat(X_test, y_test)
print('Training set', X_train.shape, y_train.shape)
print('Validation set', X_valid.shape, y_valid.shape)
print('Test set', X_test.shape, y_test.shape)
# Training set (50000, 784) (50000, 10) # Validation set (10000, 10) (10000, 10) # Test set (10000, 784) (10000, 10)
```

The final dimesion is expected to be (number_of_samples, number_of_labels)

We still need to do some data manipulation:

```
np.random.seed(0)
train_indices = np.random.choice(60000, 50000, replace=False)
valid_indices = [i for i in range(60000) if i not in train_indices]

X_valid, y_valid = X_train[valid_indices,:,:], y_train[valid_indices]
X_train, y_train = X_train[train_indices,:,:], y_train[train_indices]
print(X_train.shape, X_valid.shape, X_test.shape)
# (50000, 28, 28) (10000, 28, 28) (10000, 28, 28)

image_size = 28
num_labels = 10

def reformat(dataset, labels):
  dataset = dataset.reshape((-1, image_size * image_size)).astype(np.float32)
  # one hot encoding: Map 1 to [0.0, 1.0, 0.0 ...], 2 to [0.0, 0.0, 1.0 ...]
  labels = (np.arange(num_labels) == labels[:,None]).astype(np.float32)
  return dataset, labels

X_train, y_train = reformat(X_train, y_train)
X_valid, y_valid = reformat(X_valid, y_valid)
X_test, y_test = reformat(X_test, y_test)
print('Training set', X_train.shape, y_train.shape)
print('Validation set', X_valid.shape, y_valid.shape)
print('Test set', X_test.shape, y_test.shape)
# Training set (50000, 784) (50000, 10) # Validation set (10000, 10) (10000, 10) # Test set (10000, 784) (10000, 10)
```

Reshaping our data

We still need to do some data manipulation:

```
np.random.seed(0)
train_indices = np.random.choice(60000, 50000, replace=False)
valid_indices = [i for i in range(60000) if i not in train_indices]

X_valid, y_valid = X_train[valid_indices,:,:], y_train[valid_indices]
X_train, y_train = X_train[train_indices,:,:], y_train[train_indices]
print(X_train.shape, X_valid.shape, X_test.shape)
# (50000, 28, 28) (10000, 28, 28) (10000, 28, 28)

image_size = 28
num_labels = 10

def reformat(dataset, labels):
  dataset = dataset.reshape((-1, image_size * image_size)).astype(np.float32)
  # one hot encoding: Map 1 to [0.0, 1.0, 0.0 ...], 2 to [0.0, 0.0, 1.0 ...]
  labels = (np.arange(num_labels) == labels[:,None]).astype(np.float32)
  return dataset, labels

X_train, y_train = reformat(X_train, y_train)
X_valid, y_valid = reformat(X_valid, y_valid)
X_test, y_test = reformat(X_test, y_test)
print('Training set', X_train.shape, y_train.shape)
print('Validation set', X_valid.shape, y_valid.shape)
print('Test set', X_test.shape, y_test.shape)
# Training set (50000, 784) (50000, 10) # Validation set (10000, 10) (10000, 10) # Test set (10000, 784) (10000, 10)
```

Checking dimension

Here should be 784

```
import tensorflow as tf

def accuracy(predictions, labels):
    return (100.0 * np.sum(np.argmax(predictions, 1) == np.argmax(labels, 1)) / predictions.shape[0])

batch_size = 256
num_hidden_units = 1024
lambda1 = 0.1
lambda2 = 0.1

graph = tf.Graph()
```

Calculates the model accuracy

```python
import tensorflow as tf

def accuracy(predictions, labels):
  return (100.0 * np.sum(np.argmax(predictions, 1) == np.argmax(labels, 1)) / predictions.shape[0])

batch_size = 256
num_hidden_units = 1024
lambda1 = 0.1
lambda2 = 0.1

graph = tf.Graph()
```

Counts how many of the predictions is the same from the expected labels

Axis = 1 means to compare along columns

```
import tensorflow as tf

def accuracy(predictions, labels):
  return (100.0 * np.sum(np.argmax(predictions, 1) == np.argmax(labels, 1)) / predictions.shape[0])

batch_size = 256
num_hidden_units = 1024
lambda1 = 0.1
lambda2 = 0.1

graph = tf.Graph()
```

Axis = 1 means to compare along columns

```python
import tensorflow as tf

def accuracy(predictions, labels):
  return (100.0 * np.sum(np.argmax(predictions, 1) == np.argmax(labels, 1)) / predictions.shape[0])

batch_size = 256
num_hidden_units = 1024
lambda1 = 0.1
lambda2 = 0.1

graph = tf.Graph()
```

An example of how this works

```python
import tensorflow as tf

def accuracy(predictions, labels):
  return (100.0 * np.sum(np.argmax(predictions, 1) == np.argmax(labels, 1)) / predictions.shape[0])

batch_size = 256
num_hidden_units = 1024
lambda1 = 0.1
lambda2 = 0.1

graph = tf.Graph()
```

An example of how this works

```python
# Example prediction arrays (probabilities for each class for each sample)

predictions = np.array([
   [0.2, 0.5, 0.3],  # Sample 1: Probability for Class A, B, and C
   [0.7, 0.1, 0.2],  # Sample 2: Probability for Class A, B, and C
   [0.4, 0.3, 0.3],  # Sample 3: Probability for Class A, B, and C
   [0.1, 0.2, 0.7],  # Sample 4: Probability for Class A, B, and C
])

# Find the index of the maximum probability for each sample
predicted_classes = np.argmax(predictions, axis=1)
print(predicted_classes)

#Returned output
[1 0 0 2]
```

It returns the index of the element from each array that contains the max value

```python
import tensorflow as tf

def accuracy(predictions, labels):
  return (100.0 * np.sum(np.argmax(predictions, 1) == np.argmax(labels, 1)) / predictions.shape[0])

batch_size = 256
num_hidden_units = 1024
lambda1 = 0.1
lambda2 = 0.1

graph = tf.Graph()
```

Returns the amount of same index matching between this 2 arrays

```python
import tensorflow as tf

def accuracy(predictions, labels):
  return (100.0 * np.sum(np.argmax(predictions, 1) == np.argmax(labels, 1)) / predictions.shape[0])

batch_size = 256
num_hidden_units = 1024
lambda1 = 0.1
lambda2 = 0.1

graph = tf.Graph()
```

Computes the % of accuracy when divided by the total amount of elements and then multiplied by 100

```python
import tensorflow as tf

def accuracy(predictions, labels):
  return (100.0 * np.sum(np.argmax(predictions, 1) == np.argmax(labels, 1)) / predictions.shape[0])

batch_size = 256
num_hidden_units = 1024
lambda1 = 0.1
lambda2 = 0.1

graph = tf.Graph()
```

Starting to define the neural network structure

```python
import tensorflow as tf

def accuracy(predictions, labels):
  return (100.0 * np.sum(np.argmax(predictions, 1) == np.argmax(labels, 1)) / predictions.shape[0])

batch_size = 256
num_hidden_units = 1024
lambda1 = 0.1
lambda2 = 0.1

graph = tf.Graph()
```

For each training loop, we will have 256 training sample

```
import tensorflow as tf

def accuracy(predictions, labels):
  return (100.0 * np.sum(np.argmax(predictions, 1) == np.argmax(labels, 1)) / predictions.shape[0])

batch_size = 256
num_hidden_units = 1024
lambda1 = 0.1
lambda2 = 0.1

graph = tf.Graph()
```

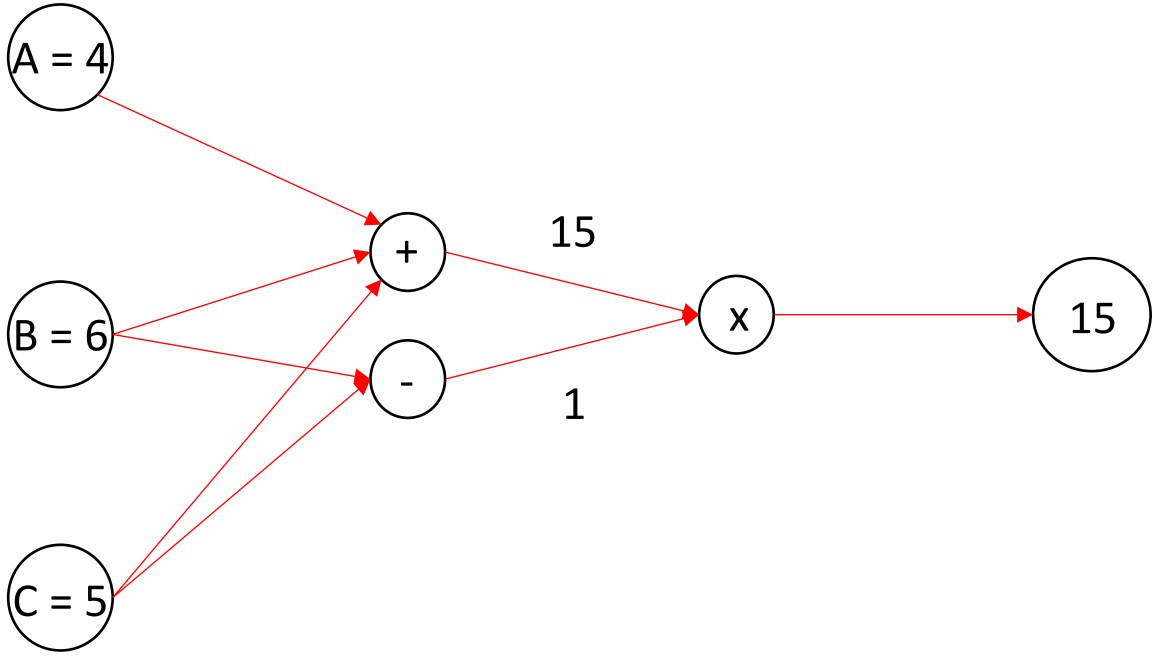
For each layer, the neural network will have 1024 hidden units

```python
import tensorflow as tf

def accuracy(predictions, labels):
  return (100.0 * np.sum(np.argmax(predictions, 1) == np.argmax(labels, 1)) / predictions.shape[0])

batch_size = 256
num_hidden_units = 1024
lambda1 = 0.1
lambda2 = 0.1

graph = tf.Graph()
```

This are the regularization terms by calculating the L2 norm (Euclidian distance) for the first and second hidden layers

About tf.Graph:

- tf.Graph in tensor flow is a method to create a computational graph.
- Computational graphs are directed graphs that represents mathematical expressions.
- Isn’t used matrix operation.

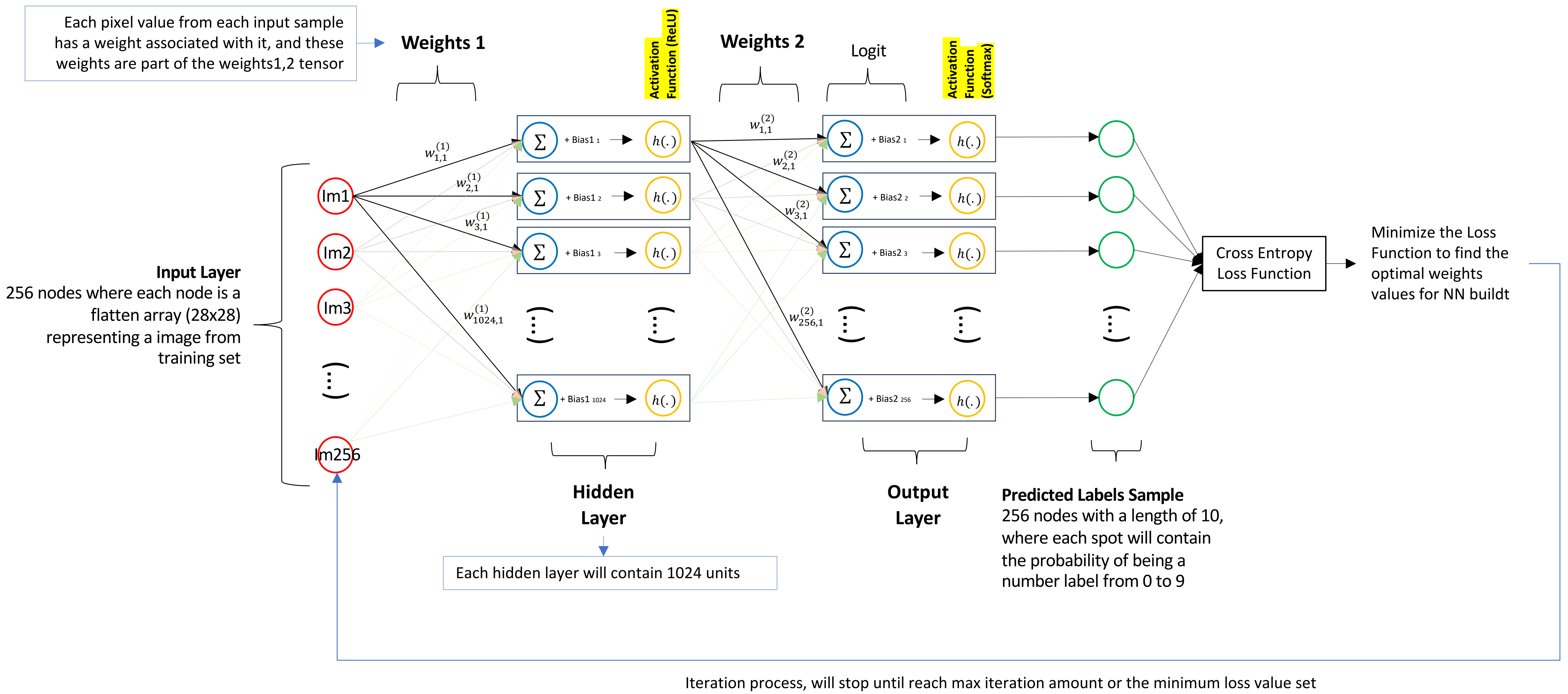


Design consideration when building a neural networks:

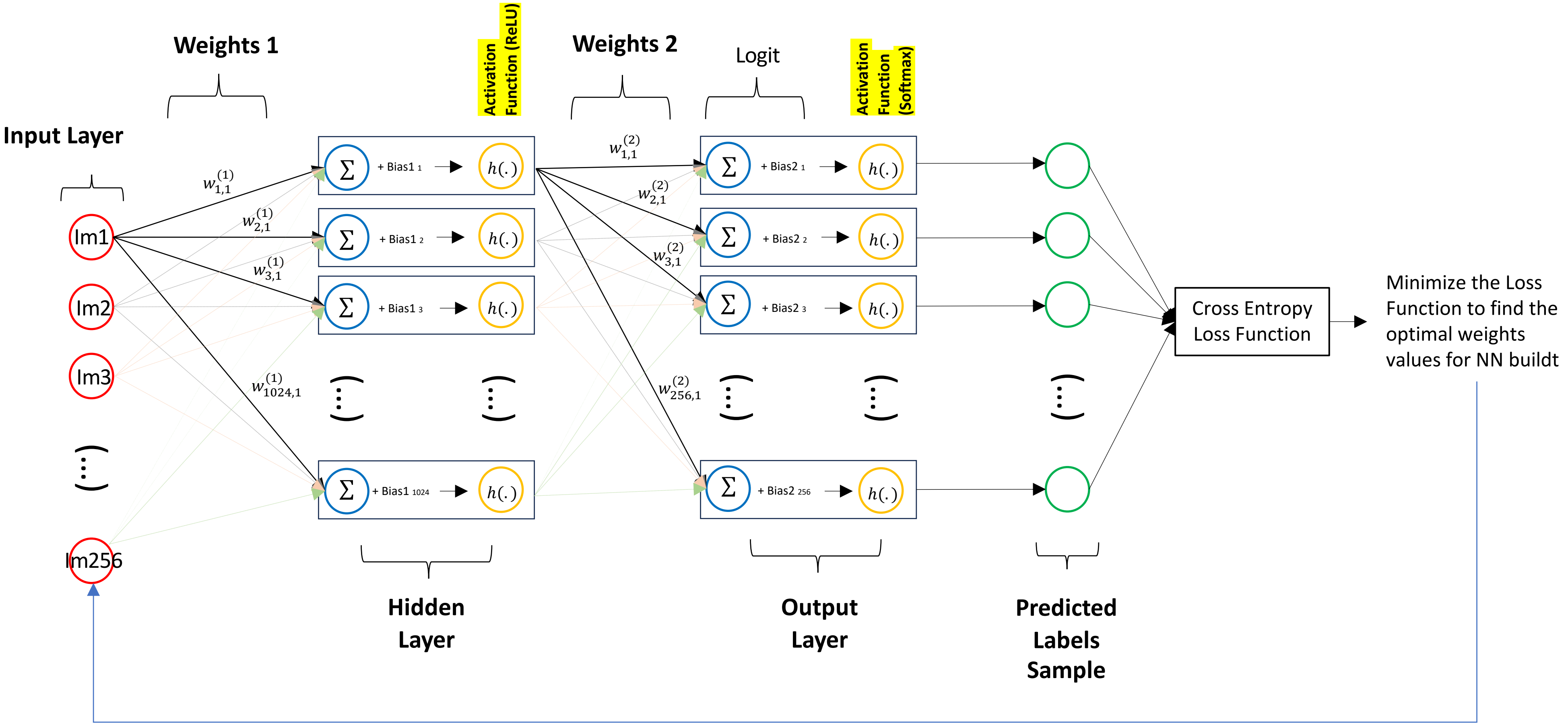
- How many layers? (Will define the depth of Neural Networks)
- How many layers are connected?
- How many units per layers?
- Activation function will be used in each hidden layer?
- Activation function will be used in output layer?
- Which cost function to use?
- Which optimizer to use?

```python
with graph.as_default():
    # Input data placeholders
    tf_train_dataset = tf.compat.v1.placeholder(tf.float32, shape=(batch_size, image_size * image_size))
    tf_train_labels = tf.compat.v1.placeholder(tf.float32, shape=(batch_size, num_labels))
    tf_valid_dataset = tf.constant(X_valid)
    tf_test_dataset = tf.constant(X_test)

    # Variables
    weights1 = tf.Variable(tf.random.truncated_normal([image_size * image_size, num_hidden_units]))
    biases1 = tf.Variable(tf.zeros([num_hidden_units]))
    weights2 = tf.Variable(tf.random.truncated_normal([num_hidden_units, num_labels]))
    biases2 = tf.Variable(tf.zeros([num_labels]))

    # Hidden layer computation with ReLU activation
    hidden_layer_output = tf.nn.relu(tf.matmul(tf_train_dataset, weights1) + biases1


    # Logits computation
    logits = tf.matmul(hidden_layer_output, weights2) + biases2

    # Softmax activation for logits
    softmax_logits = tf.nn.softmax(logits)

    # Loss computation
    loss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(labels=tf_train_labels, logits=softmax_logits) + \
                          lambda1 * tf.nn.l2_loss(weights1) + lambda2 * tf.nn.l2_loss(weights2))

    # Optimizer
    optimizer = tf.compat.v1.train.GradientDescentOptimizer(0.008).minimize(loss)

    # Predictions for the training, validation, and test data.
    train_prediction = softmax_logits
    valid_hidden_layer_output = tf.nn.relu(tf.matmul(tf_valid_dataset, weights1) + biases1)
    valid_prediction = tf.nn.softmax(tf.matmul(valid_hidden_layer_output, weights2) + biases2)
    test_hidden_layer_output = tf.nn.relu(tf.matmul(tf_test_dataset, weights1) + biases1)
    test_prediction = tf.nn.softmax(tf.matmul(test_hidden_layer_output, weights2) + biases2)
```

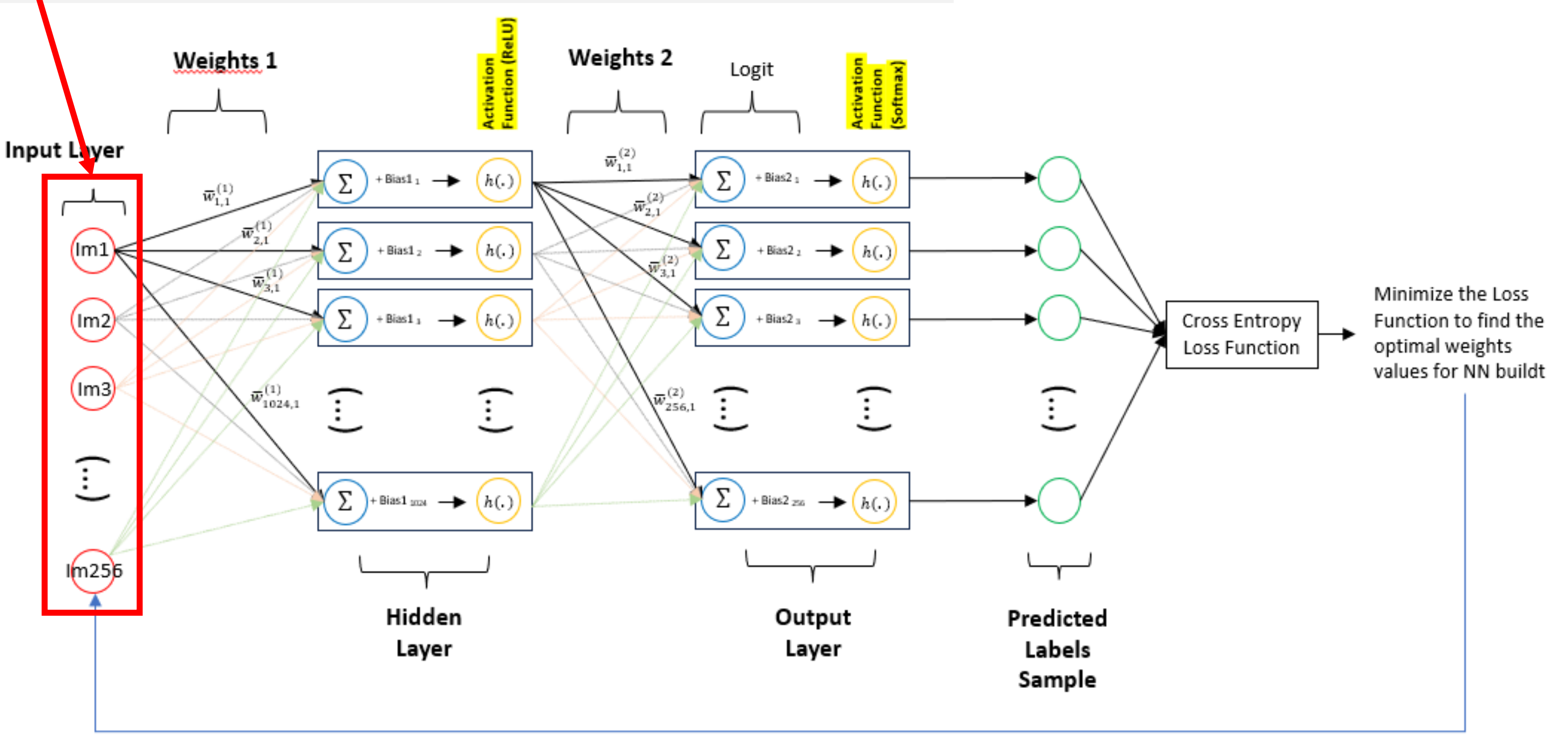
The code represents the following Neural Network structure:



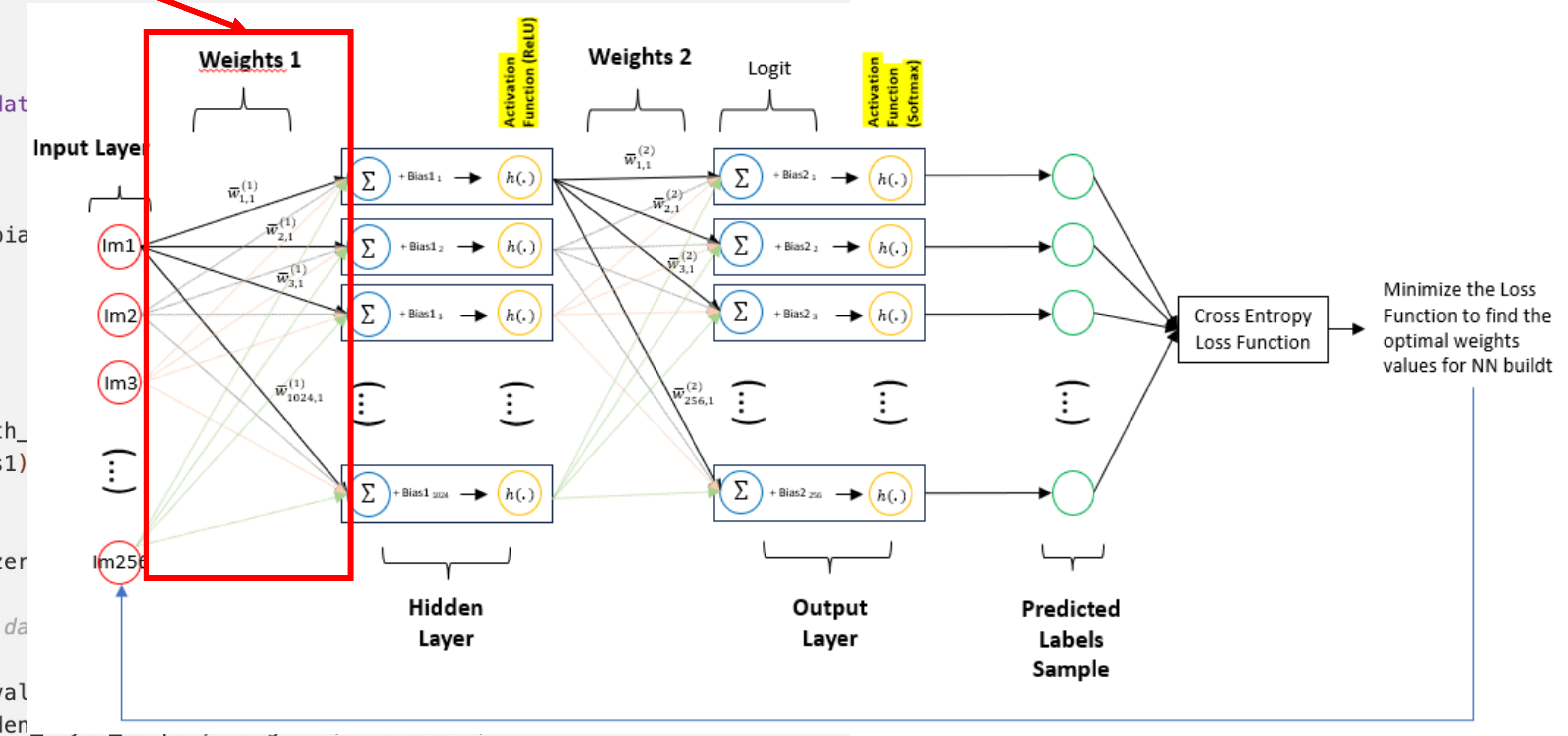
The code represents the following Neural Network structure:



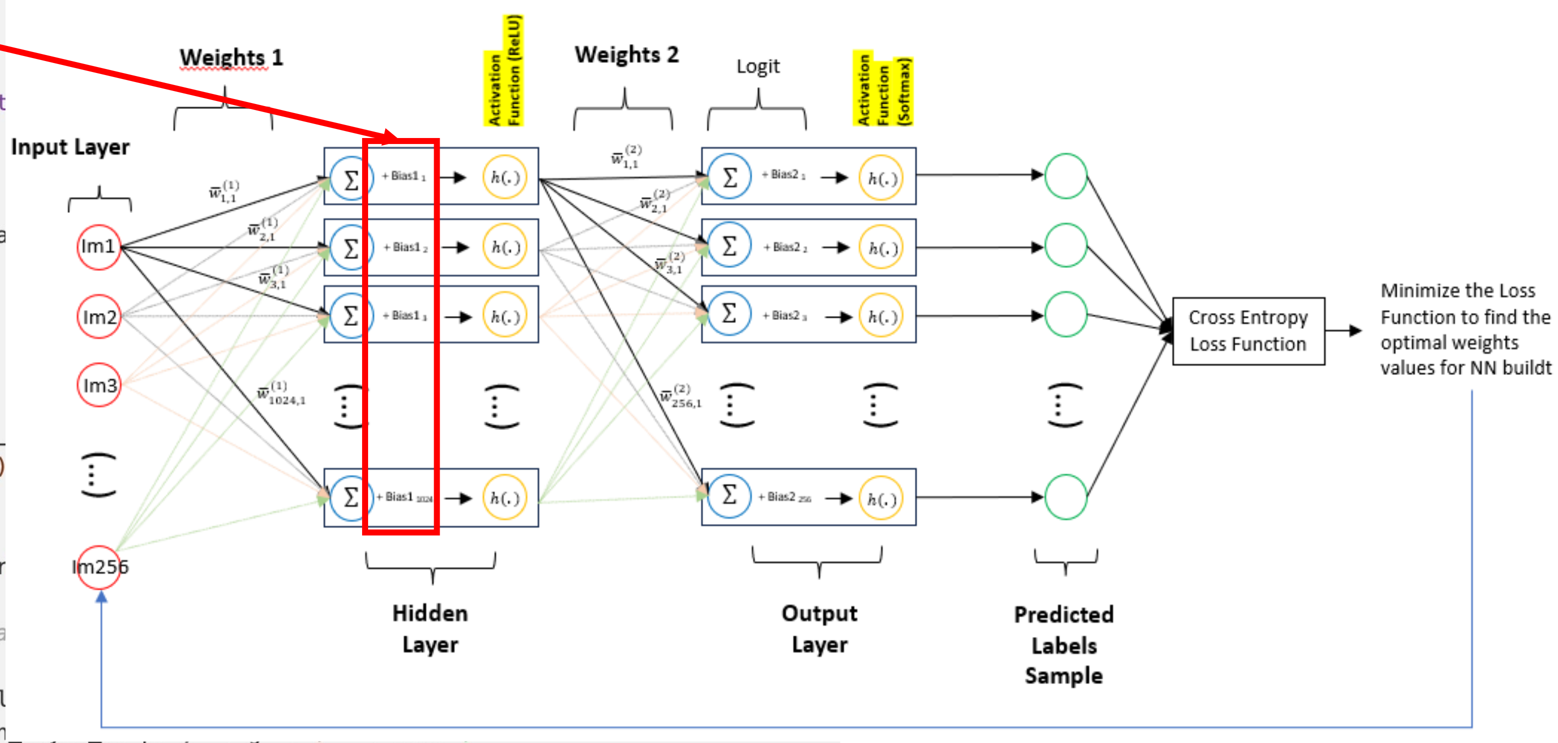
```python
with graph.as_default():
    # Input data placeholders
    tf_train_dataset = tf.compat.v1.placeholder(tf.float32, shape=(batch_size, image_size * image_size))
    tf_train_labels = tf.compat.v1.placeholder(tf.float32, shape=(batch_size, num_labels))
    tf_valid_dataset = tf.constant(X_valid)
    tf_test_dataset = tf.constant(X_test)

    # Variables
    weights1 = tf.Variable(tf.random.truncated_nor
    biases1 = tf.Variable(tf.zeros([num_hidden_uni
    weights2 = tf.Variable(tf.random.truncated_nor
    biases2 = tf.Variable(tf.zeros([num_labels]))

    # Hidden layer computation with ReLU activatio
    hidden_layer_output = tf.nn.relu(tf.matmul(tf_


    # Logits computation
    logits = tf.matmul(hidden_layer_output, weight

    # Softmax activation for logits
    softmax_logits = tf.nn.softmax(logits)

    # Loss computation
    loss = tf.reduce_mean(tf.nn.softmax_cross_entr
                          lambda1 * tf.nn.l2_loss(

    # Optimizer
    optimizer = tf.compat.v1.train.GradientDescent

    # Predictions for the training, validation, and test data.
    train_prediction = softmax_logits
    valid_hidden_layer_output = tf.nn.relu(tf.matmul(tf_valid_dataset, weights1) + biases1)
    valid_prediction = tf.nn.softmax(tf.matmul(valid_hidden_layer_output, weights2) + biases2)
    test_hidden_layer_output = tf.nn.relu(tf.matmul(tf_test_dataset, weights1) + biases1)
    test_prediction = tf.nn.softmax(tf.matmul(test_hidden_layer_output, weights2) + biases2)
```



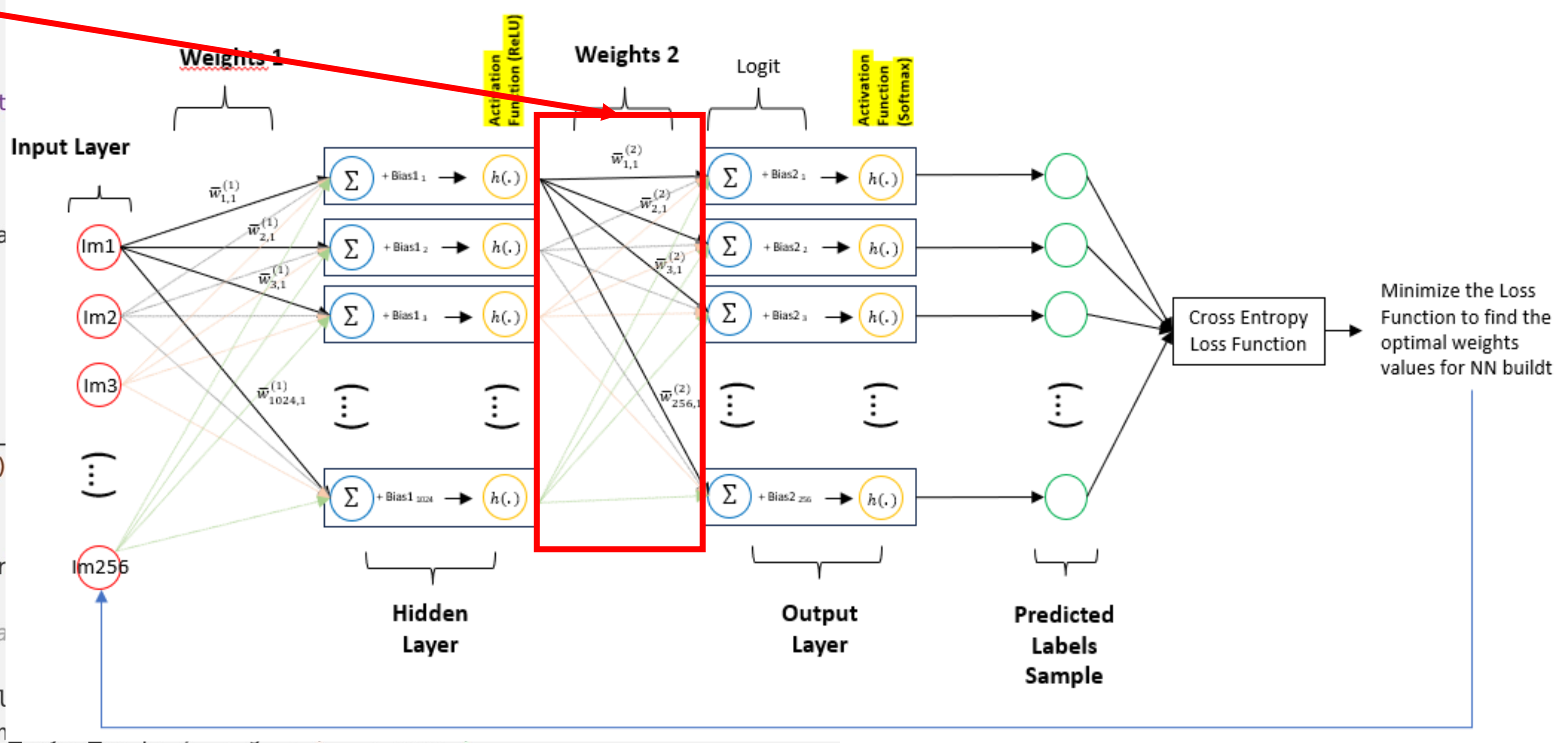
```python
with graph.as_default():
    # Input data placeholders
    tf_train_dataset = tf.compat.v1.placeholder(tf.float32, shape=(batch_size, image_size * image_size))
    tf_train_labels = tf.compat.v1.placeholder(tf.float32, shape=(batch_size, num_labels))
    tf_valid_dataset = tf.constant(X_valid)
    tf_test_dataset = tf.constant(X_test)

    # Variables
    weights1 = tf.Variable(tf.random.truncated_normal([image_size * image_size, num_hidden_units]))
    biases1 = tf.Variable(tf.zeros([num_hidden_units]))
    weights2 = tf.Variable(tf.random.truncated_normal([num_hidden_units, num_labels]))
    biases2 = tf.Variable(tf.zeros([num_labels]))

    # Hidden layer computation with ReLU activation
    hidden_layer_output = tf.nn.relu(tf.matmul(tf_train_dat

    # Logits computation
    logits = tf.matmul(hidden_layer_output, weights2) + bia

    # Softmax activation for logits
    softmax_logits = tf.nn.softmax(logits)

    # Loss computation
    loss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_
                          lambda1 * tf.nn.l2_loss(weights1)

    # Optimizer
    optimizer = tf.compat.v1.train.GradientDescentOptimizer

    # Predictions for the training, validation, and test da
    train_prediction = softmax_logits
    valid_hidden_layer_output = tf.nn.relu(tf.matmul(tf_val
    valid_prediction = tf.nn.softmax(tf.matmul(valid_hidden_
    test_hidden_layer_output = tf.nn.relu(tf.matmul(tf_test_dataset, weights1) + biases1)
    test_prediction = tf.nn.softmax(tf.matmul(test_hidden_layer_output, weights2) + biases2)
```



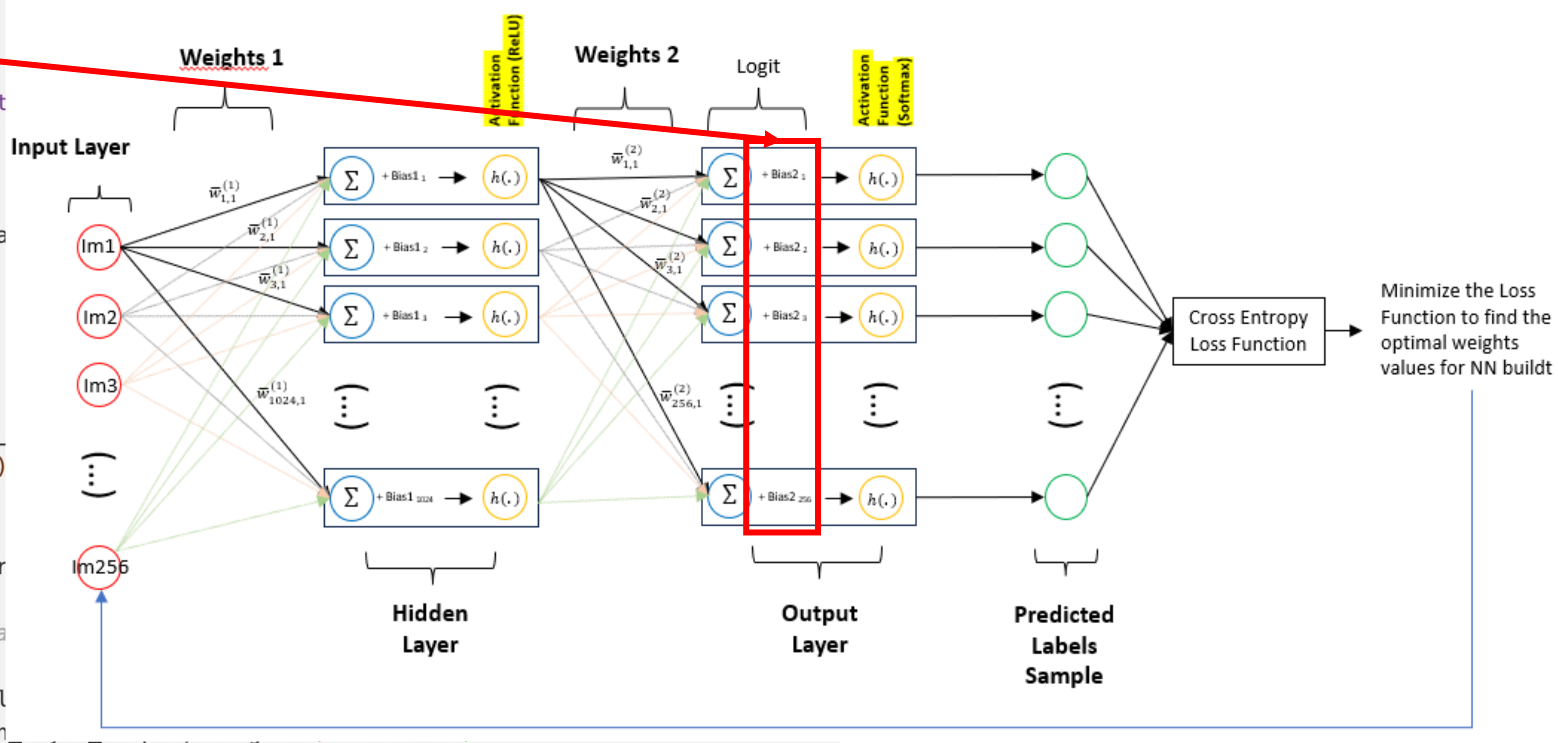
```
with graph.as_default():
    # Input data placeholders
    tf_train_dataset = tf.compat.v1.placeholder(tf.float32, shape=(batch_size, image_size * image_size))
    tf_train_labels = tf.compat.v1.placeholder(tf.float32, shape=(batch_size, num_labels))
    tf_valid_dataset = tf.constant(X_valid)
    tf_test_dataset = tf.constant(X_test)

    # Variables
    weights1 = tf.Variable(tf.random.truncated_normal([image_size * image_size, num_hidden_units]))
    biases1 = tf.Variable(tf.zeros([num_hidden_units]))
    weights2 = tf.Variable(tf.random.truncated_normal([num_hidden_units, num_labels]))
    biases2 = tf.Variable(tf.zeros([num_labels]))

    # Hidden layer computation with ReLU activation
    hidden_layer_output = tf.nn.relu(tf.matmul(tf_train_dat

    # Logits computation
    logits = tf.matmul(hidden_layer_output, weights2) + bia

    # Softmax activation for logits
    softmax_logits = tf.nn.softmax(logits)

    # Loss computation
    loss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_
                          lambda1 * tf.nn.l2_loss(weights1)

    # Optimizer
    optimizer = tf.compat.v1.train.GradientDescentOptimizer

    # Predictions for the training, validation, and test da
    train_prediction = softmax_logits
    valid_hidden_layer_output = tf.nn.relu(tf.matmul(tf_val
    valid_prediction = tf.nn.softmax(tf.matmul(valid_hidden_
    test_hidden_layer_output = tf.nn.relu(tf.matmul(tf_test_dataset, weights1) + biases1)
    test_prediction = tf.nn.softmax(tf.matmul(test_hidden_layer_output, weights2) + biases2)
```



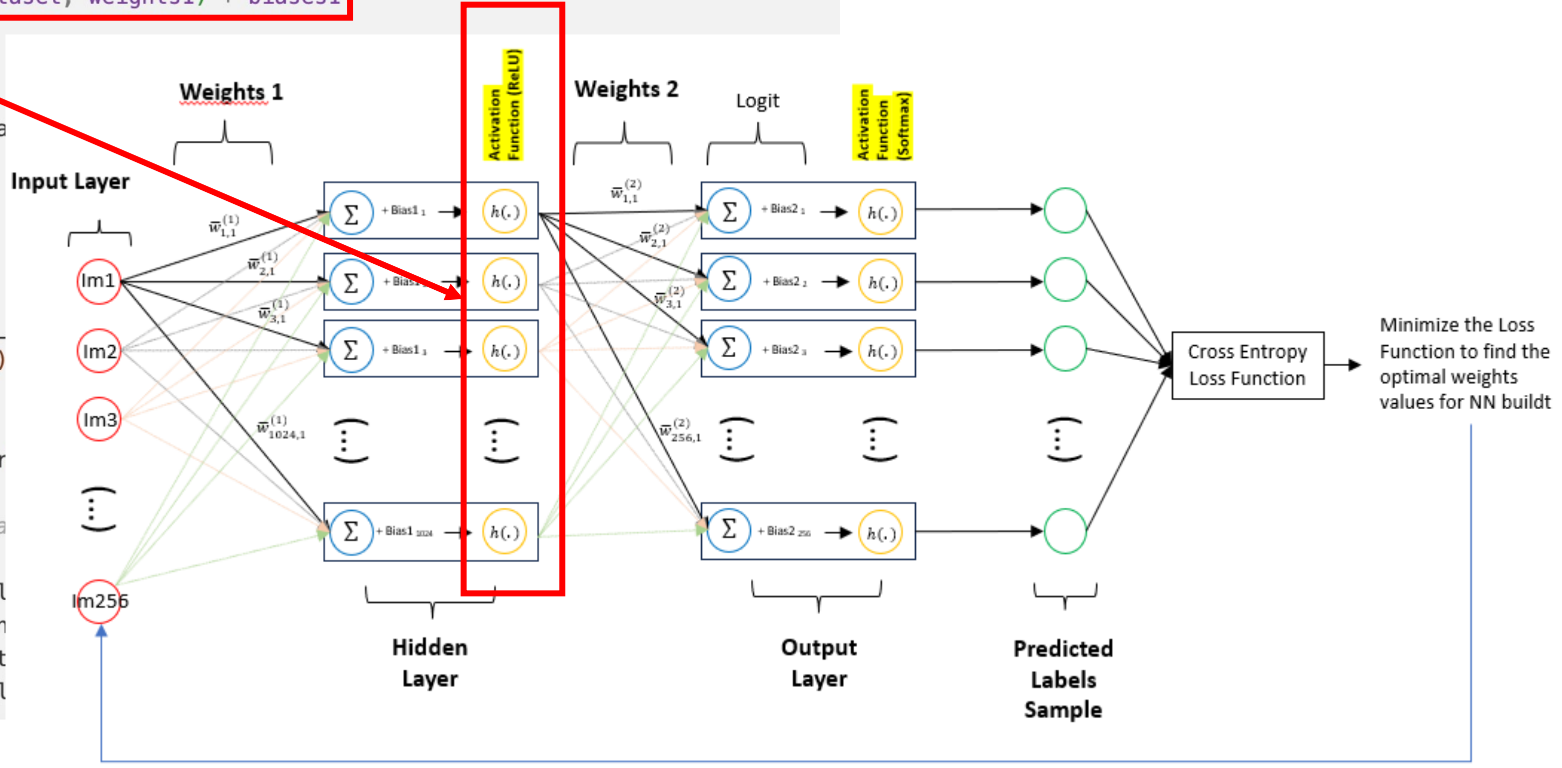
```python
with graph.as_default():
    # Input data placeholders
    tf_train_dataset = tf.compat.v1.placeholder(tf.float32, shape=(batch_size, image_size * image_size))
    tf_train_labels = tf.compat.v1.placeholder(tf.float32, shape=(batch_size, num_labels))
    tf_valid_dataset = tf.constant(X_valid)
    tf_test_dataset = tf.constant(X_test)

    # Variables
    weights1 = tf.Variable(tf.random.truncated_normal([image_size * image_size, num_hidden_units]))
    biases1 = tf.Variable(tf.zeros([num_hidden_units]))
    weights2 = tf.Variable(tf.random.truncated_normal([num_hidden_units, num_labels]))
    biases2 = tf.Variable(tf.zeros([num_labels]))

    # Hidden layer computation with ReLU activation
    hidden_layer_output = tf.nn.relu(tf.matmul(tf_train_dat

    # Logits computation
    logits = tf.matmul(hidden_layer_output, weights2) + bia

    # Softmax activation for logits
    softmax_logits = tf.nn.softmax(logits)

    # Loss computation
    loss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_
                          lambda1 * tf.nn.l2_loss(weights1)

    # Optimizer
    optimizer = tf.compat.v1.train.GradientDescentOptimizer

    # Predictions for the training, validation, and test da
    train_prediction = softmax_logits
    valid_hidden_layer_output = tf.nn.relu(tf.matmul(tf_val
    valid_prediction = tf.nn.softmax(tf.matmul(valid_hidden_
    test_hidden_layer_output = tf.nn.relu(tf.matmul(tf_test_dataset, weights1) + biases1)
    test_prediction = tf.nn.softmax(tf.matmul(test_hidden_layer_output, weights2) + biases2)
```



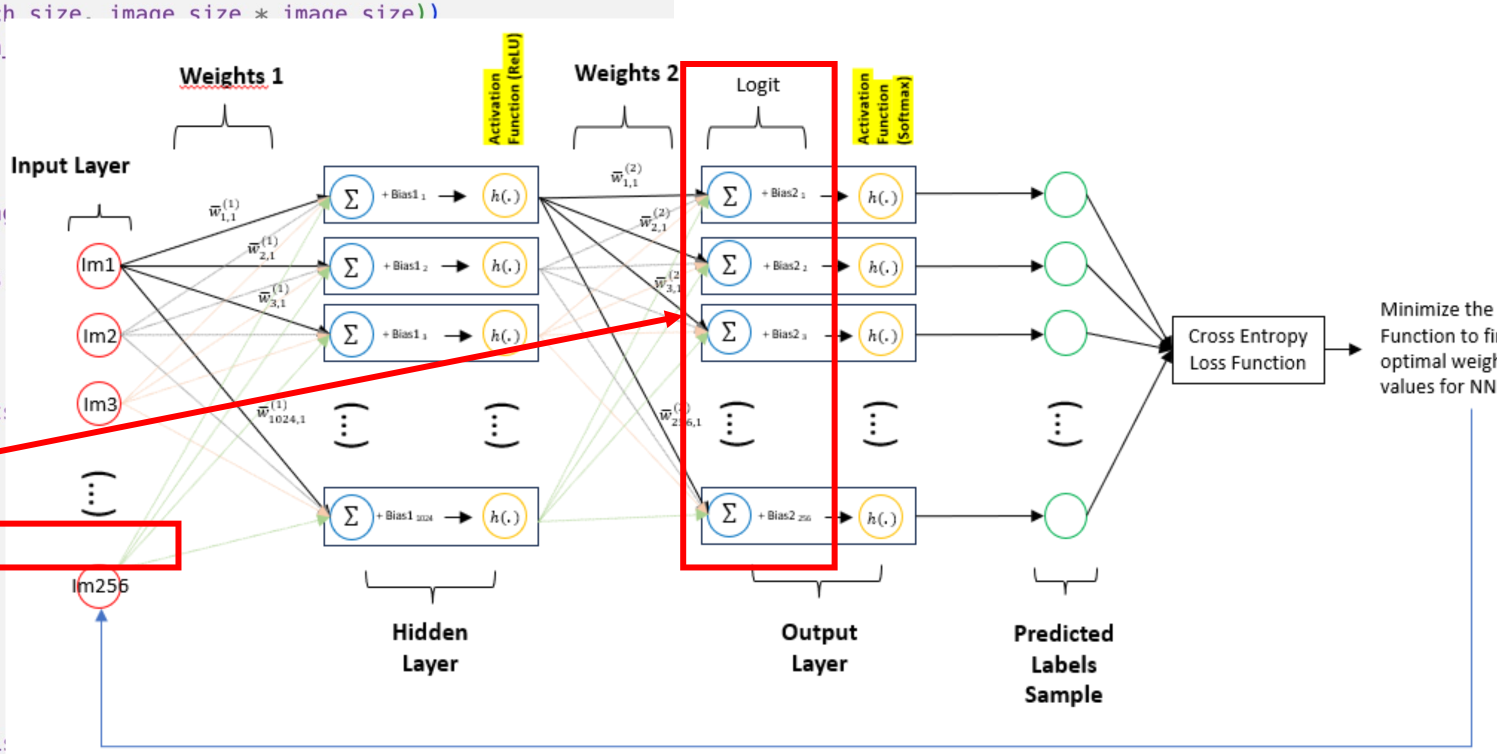
```
with graph.as_default():
    # Input data placeholders
    tf_train_dataset = tf.compat.v1.placeholder(tf.float32, shape=(batch_size, image_size * image_size))
    tf_train_labels = tf.compat.v1.placeholder(tf.float32, shape=(batch_size, num_labels))
    tf_valid_dataset = tf.constant(X_valid)
    tf_test_dataset = tf.constant(X_test)

    # Variables
    weights1 = tf.Variable(tf.random.truncated_normal([image_size * image_size, num_hidden_units]))
    biases1 = tf.Variable(tf.zeros([num_hidden_units]))
    weights2 = tf.Variable(tf.random.truncated_normal([num_hidden_units, num_labels]))
    biases2 = tf.Variable(tf.zeros([num_labels]))

    # Hidden layer computation with ReLU activation
    hidden_layer_output = tf.nn.relu(tf.matmul(tf_train_dat

    # Logits computation
    logits = tf.matmul(hidden_layer_output, weights2) + bia

    # Softmax activation for logits
    softmax_logits = tf.nn.softmax(logits)

    # Loss computation
    loss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_
                          lambda1 * tf.nn.l2_loss(weights1)

    # Optimizer
    optimizer = tf.compat.v1.train.GradientDescentOptimizer

    # Predictions for the training, validation, and test da
    train_prediction = softmax_logits
    valid_hidden_layer_output = tf.nn.relu(tf.matmul(tf_val
    valid_prediction = tf.nn.softmax(tf.matmul(valid_hidden_
    test_hidden_layer_output = tf.nn.relu(tf.matmul(tf_test_dataset, weights1) + biases1)
    test_prediction = tf.nn.softmax(tf.matmul(test_hidden_layer_output, weights2) + biases2)
```



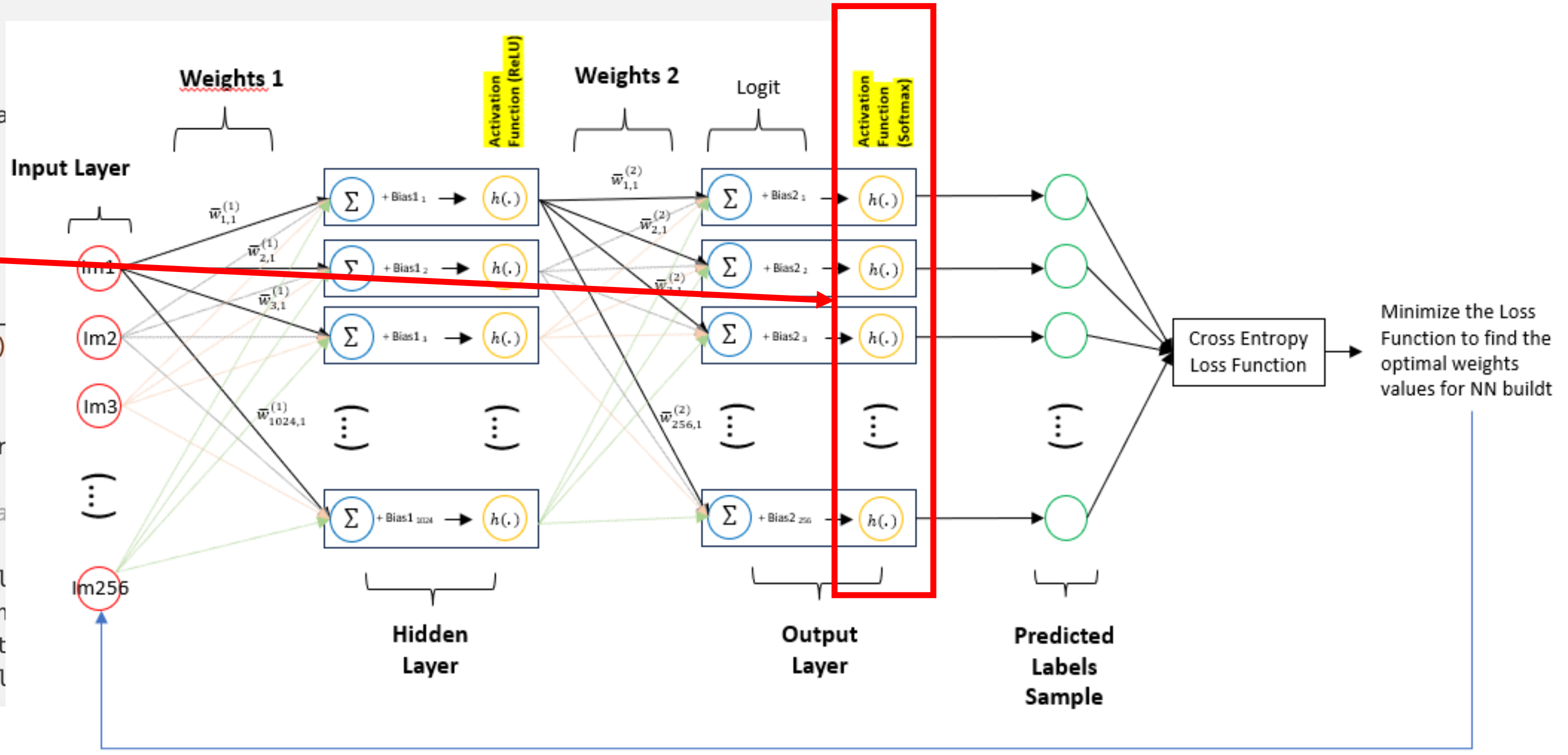
```
with graph.as_default():
    # Input data placeholders
    tf_train_dataset = tf.compat.v1.placeholder(tf.float32, shape=(batch_size, image_size * image_size))
    tf_train_labels = tf.compat.v1.placeholder(tf.float32, shape=(batch_size, num_labels))
    tf_valid_dataset = tf.constant(X_valid)
    tf_test_dataset = tf.constant(X_test)

    # Variables
    weights1 = tf.Variable(tf.random.truncated_normal([image_size * image_size, num_hidden_units]))
    biases1 = tf.Variable(tf.zeros([num_hidden_units]))
    weights2 = tf.Variable(tf.random.truncated_normal([num_hidden_units, num_labels]))
    biases2 = tf.Variable(tf.zeros([num_labels]))

    # Hidden layer computation with ReLU activation
    hidden_layer_output = tf.nn.relu(tf.matmul(tf_train_dataset, weights1) + biases1


    # Logits computation
    logits = tf.matmul(hidden_layer_output, weights2) + bia

    # Softmax activation for logits
    softmax_logits = tf.nn.softmax(logits)

    # Loss computation
    loss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_
                                  lambda1 * tf.nn.l2_loss(weights1)

    # Optimizer
    optimizer = tf.compat.v1.train.GradientDescentOptimizer

    # Predictions for the training, validation, and test da
    train_prediction = softmax_logits
    valid_hidden_layer_output = tf.nn.relu(tf.matmul(tf_val
    valid_prediction = tf.nn.softmax(tf.matmul(valid_hidden
    test_hidden_layer_output = tf.nn.relu(tf.matmul(tf_test
    test_prediction = tf.nn.softmax(tf.matmul(test_hidden_l
```



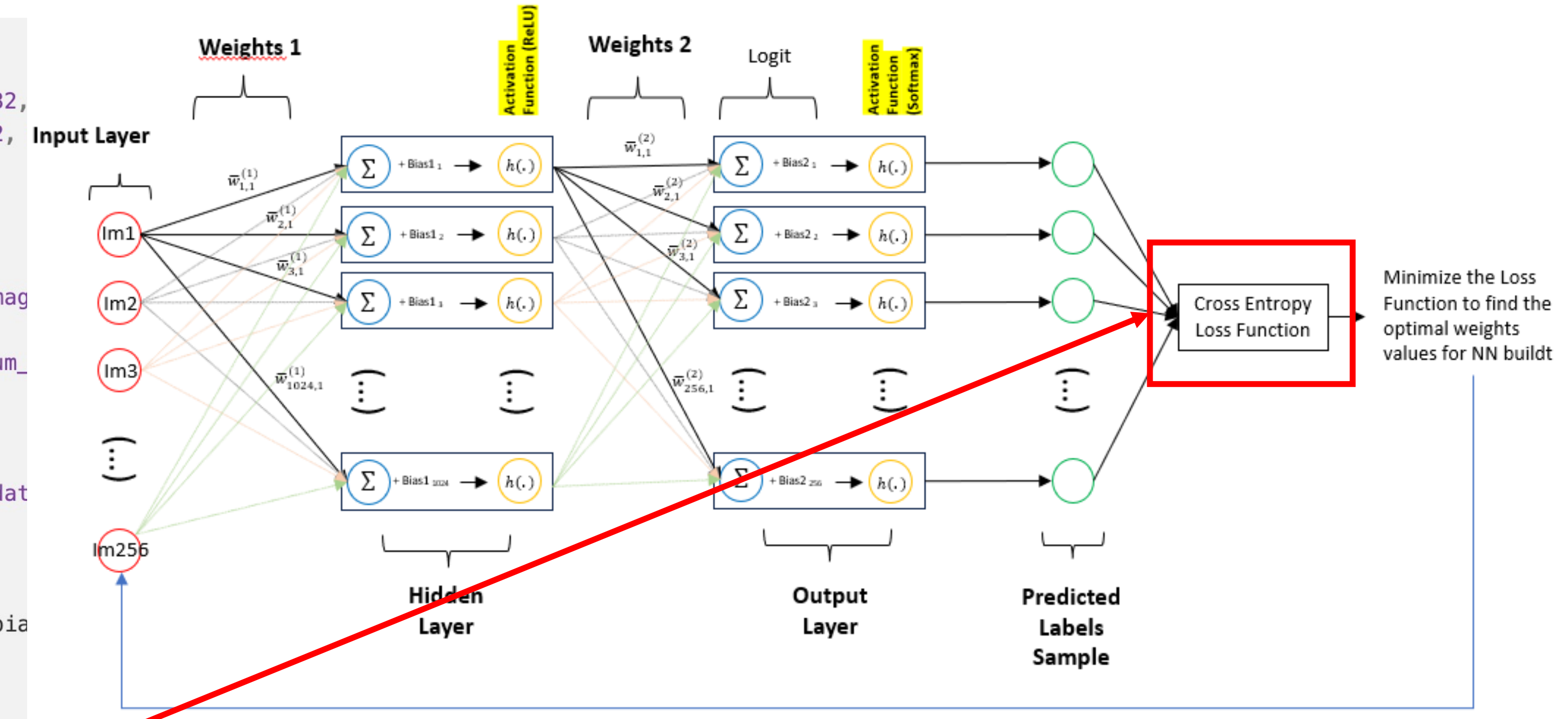
```
with graph.as_default():
    # Input data placeholders
    tf_train_dataset = tf.compat.v1.placeholder(tf.float32, shape=(batch_size, image_size * image_size))
    tf_train_labels = tf.compat.v1.placeholder(tf.float32, shape=(batch_
    tf_valid_dataset = tf.constant(X_valid)
    tf_test_dataset = tf.constant(X_test)

    # Variables
    weights1 = tf.Variable(tf.random.truncated_normal([image_size * ima
    biases1 = tf.Variable(tf.zeros([num_hidden_units]))
    weights2 = tf.Variable(tf.random.truncated_normal([num_hidden_units
    biases2 = tf.Variable(tf.zeros([num_labels]))

    # Hidden layer computation with ReLU activation
    hidden_layer_output = tf.nn.relu(tf.matmul(tf_train_dataset, weight


    # Logits computation
    logits = tf.matmul(hidden_layer_output, weights2) + biases2

    # Softmax activation for logits
    softmax_logits = tf.nn.softmax(logits)

    # Loss computation
    loss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(label
                          lambda1 * tf.nn.l2_loss(weights1) + lambda2 * tf.nn.l2_loss(weights2))

    # Optimizer
    optimizer = tf.compat.v1.train.GradientDescentOptimizer(0.008).minimize(loss)

    # Predictions for the training, validation, and test data.
    train_prediction = softmax_logits
    valid_hidden_layer_output = tf.nn.relu(tf.matmul(tf_valid_dataset, weights1) + biases1)
    valid_prediction = tf.nn.softmax(tf.matmul(valid_hidden_layer_output, weights2) + biases2)
    test_hidden_layer_output = tf.nn.relu(tf.matmul(tf_test_dataset, weights1) + biases1)
    test_prediction = tf.nn.softmax(tf.matmul(test_hidden_layer_output, weights2) + biases2)
```



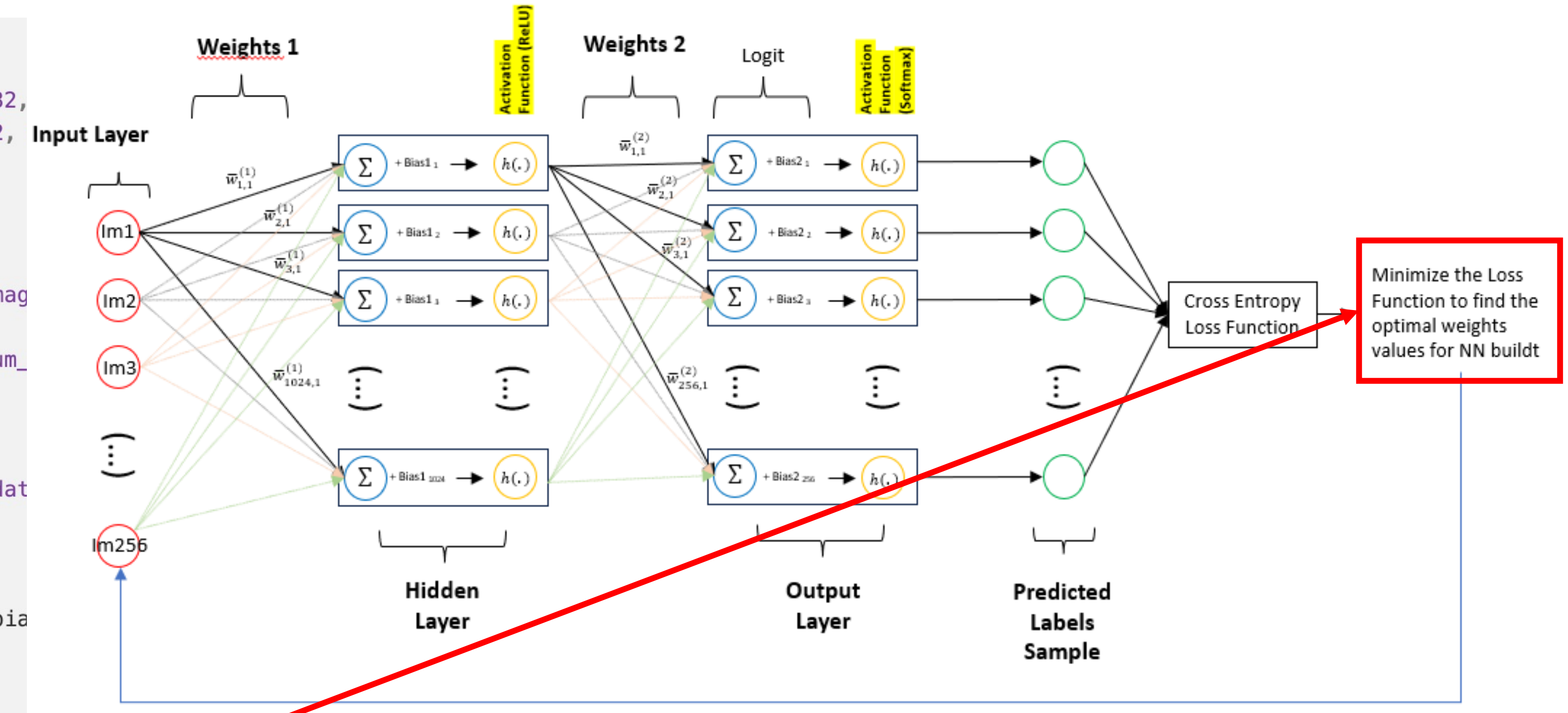
```python
with graph.as_default():
    # Input data placeholders
    tf_train_dataset = tf.compat.v1.placeholder(tf.float32, shape=(batch_size, image_size * image_size))
    tf_train_labels = tf.compat.v1.placeholder(tf.float32, shape=(batch_size, num_labels))
    tf_valid_dataset = tf.constant(X_valid)
    tf_test_dataset = tf.constant(X_test)

    # Variables
    weights1 = tf.Variable(tf.random.truncated_normal([image_size * image_size, num_hidden_units]))
    biases1 = tf.Variable(tf.zeros([num_hidden_units]))
    weights2 = tf.Variable(tf.random.truncated_normal([num_hidden_units, num_labels]))
    biases2 = tf.Variable(tf.zeros([num_labels]))

    # Hidden layer computation with ReLU activation
    hidden_layer_output = tf.nn.relu(tf.matmul(tf_train_dataset, weights1) + biases1)


    # Logits computation
    logits = tf.matmul(hidden_layer_output, weights2) + bia

    # Softmax activation for logits
    softmax_logits = tf.nn.softmax(logits)

    # Loss computation
    loss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_
                          lambda1 * tf.nn.l2_loss(weights1)

    # Optimizer
    optimizer = tf.compat.v1.train.GradientDescentOptimizer

    # Predictions for the training, validation, and test da
    train_prediction = softmax_logits
    valid_hidden_layer_output = tf.nn.relu(tf.matmul(tf_val
    valid_prediction = tf.nn.softmax(tf.matmul(valid_hidden
    test_hidden_layer_output = tf.nn.relu(tf.matmul(tf_test
    test_prediction = tf.nn.softmax(tf.matmul(test_hidden_l
```



```
with graph.as_default():
    # Input data placeholders
    tf_train_dataset = tf.compat.v1.placeholder(tf.float32,
    tf_train_labels = tf.compat.v1.placeholder(tf.float32,
    tf_valid_dataset = tf.constant(X_valid)
    tf_test_dataset = tf.constant(X_test)

    # Variables
    weights1 = tf.Variable(tf.random.truncated_normal([imag
    biases1 = tf.Variable(tf.zeros([num_hidden_units]))
    weights2 = tf.Variable(tf.random.truncated_normal([num_
    biases2 = tf.Variable(tf.zeros([num_labels]))

    # Hidden layer computation with ReLU activation
    hidden_layer_output = tf.nn.relu(tf.matmul(tf_train_dat


    # Logits computation
    logits = tf.matmul(hidden_layer_output, weights2) + bia

    # Softmax activation for logits
    softmax_logits = tf.nn.softmax(logits)

    # Loss computation
    loss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(labels=tf_train_labels, logits=softmax_logits) + \
                          lambda1 * tf.nn.l2_loss(weights1) + lambda2 * tf.nn.l2_loss(weights2))

    # Optimizer
    optimizer = tf.compat.v1.train.GradientDescentOptimizer(0.008).minimize(loss)

    # Predictions for the training, validation, and test data.
    train_prediction = softmax_logits
    valid_hidden_layer_output = tf.nn.relu(tf.matmul(tf_valid_dataset, weights1) + biases1)
    valid_prediction = tf.nn.softmax(tf.matmul(valid_hidden_layer_output, weights2) + biases2)
    test_hidden_layer_output = tf.nn.relu(tf.matmul(tf_test_dataset, weights1) + biases1)
    test_prediction = tf.nn.softmax(tf.matmul(test_hidden_layer_output, weights2) + biases2)
```



```python
with graph.as_default():
    # Input data placeholders
    tf_train_dataset = tf.compat.v1.placeholder(tf.float32,
    tf_train_labels = tf.compat.v1.placeholder(tf.float32,
    tf_valid_dataset = tf.constant(X_valid)
    tf_test_dataset = tf.constant(X_test)

    # Variables
    weights1 = tf.Variable(tf.random.truncated_normal([imag
    biases1 = tf.Variable(tf.zeros([num_hidden_units]))
    weights2 = tf.Variable(tf.random.truncated_normal([num_
    biases2 = tf.Variable(tf.zeros([num_labels]))

    # Hidden layer computation with ReLU activation
    hidden_layer_output = tf.nn.relu(tf.matmul(tf_train_dat


    # Logits computation
    logits = tf.matmul(hidden_layer_output, weights2) + bia

    # Softmax activation for logits
    softmax_logits = tf.nn.softmax(logits)

    # Loss computation
    loss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(labels=tf_train_labels, logits=softmax_logits) + \
                          lambda1 * tf.nn.l2_loss(weights1) + lambda2 * tf.nn.l2_loss(weights2))

    # Optimizer
    optimizer = tf.compat.v1.train.GradientDescentOptimizer(0.008).minimize(loss)

    # Predictions for the training, validation, and test data.
    train_prediction = softmax_logits
    valid_hidden_layer_output = tf.nn.relu(tf.matmul(tf_valid_dataset, weights1) + biases1)
    valid_prediction = tf.nn.softmax(tf.matmul(valid_hidden_layer_output, weights2) + biases2)
    test_hidden_layer_output = tf.nn.relu(tf.matmul(tf_test_dataset, weights1) + biases1)
    test_prediction = tf.nn.softmax(tf.matmul(test_hidden_layer_output, weights2) + biases2)
```



“n” refers to the sample indice

Since is a 2-layer neural network, we have: $y^n = h^{(2)}(\sum_{j=0}^{M} w_{kj}^{(2)} h^{(1)}(\sum_{i=0}^{D} w_{ji}^{(1)} x_i + bias1) + bias2)$

- Each sample will have y^n equation, for this case, we will have 256 for each iteration

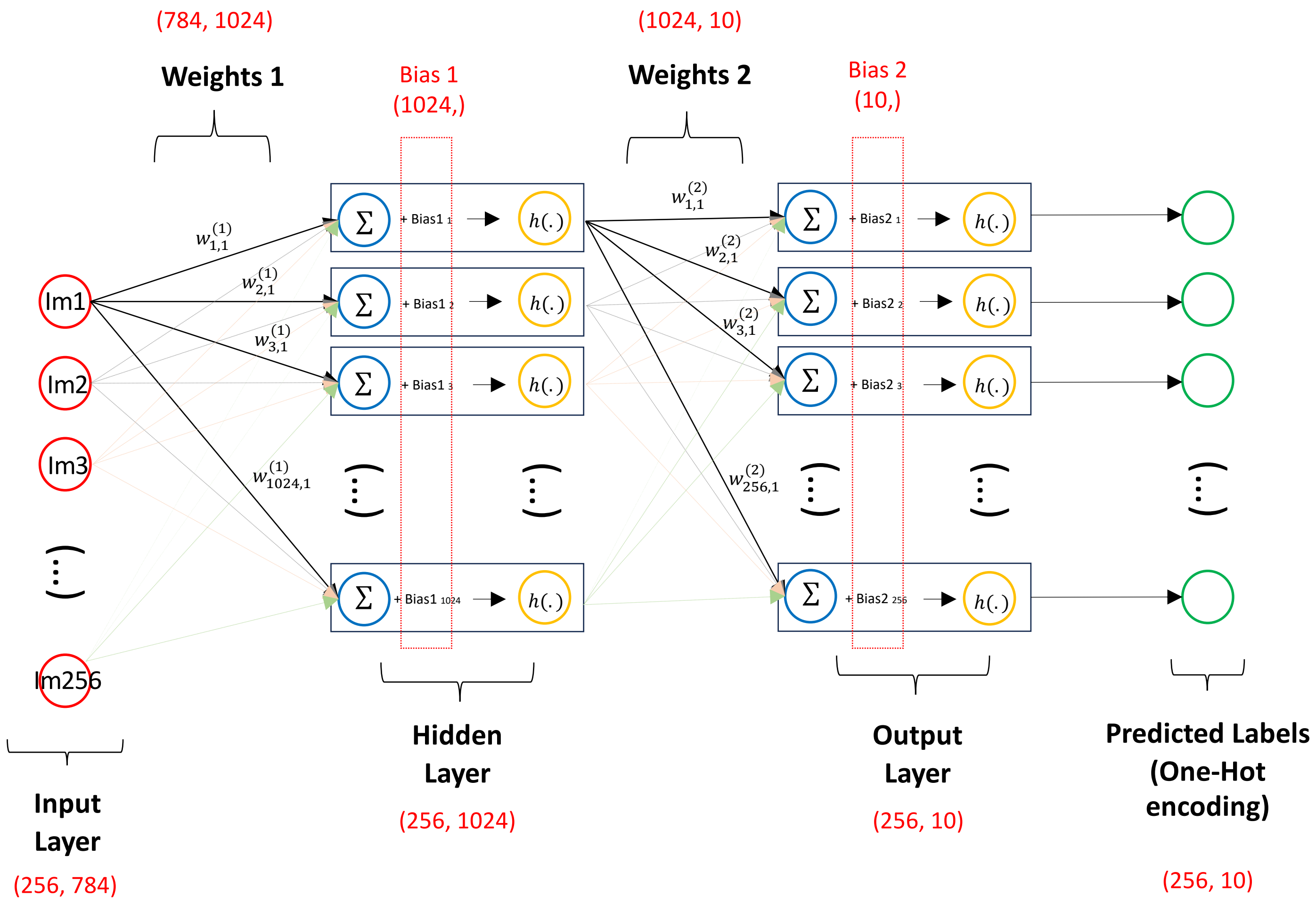
Cross Entropy Loss Function(E): $min_{\underline{w}} - \sum_{n=1}^{N} t^n \log(y^n) + (1 - t^n)\log(1 - y^n)$

- We will have a sum of 256 $t^n \log(y^n) + (1 - t^n)\log(1 - y^n)$ for this example

Gradient Descent algorithm: $\underline{w^{i+1}} = \underline{w^i} - \propto \nabla_{\underline{w}} E$

- Will be done 6001 iterations

Dimension Analysis:



Here is when the optimization part starts:

```
with tf.compat.v1.Session(graph=graph) as session:
    session.run(tf.compat.v1.global_variables_initializer())

    num_steps = 6001
    ll = []
    atr = []
    av = []

    for step in range(num_steps):
        offset = (step * batch_size) % (y_train.shape[0] - batch_size)
        batch_data = X_train[offset:(offset + batch_size), :]
        batch_labels = y_train[offset:(offset + batch_size), :]

        feed_dict = {tf_train_dataset : batch_data, tf_train_labels : batch_labels}
        _, l, predictions = session.run([optimizer, loss, train_prediction], feed_dict=feed_dict)

        if (step % 500 == 0):
            ll.append(l)
            a = accuracy(predictions, batch_labels)
            atr.append(a)
            print("Minibatch loss at step %d: %f" % (step, l))
            print("Minibatch accuracy: %.1f%%" % a)
            a = accuracy(valid_prediction.eval(), y_valid)
            av.append(a)
            print("Validation accuracy: %.1f%%" % a)
            print("Test accuracy: %.1f%%" % accuracy(test_prediction.eval(), y_test))
```

Since our Neural Network has only 256 input spot, we will divide our train dataset into mini batches

Here is when the optimization part starts:

```
with tf.compat.v1.Session(graph=graph) as session:
    session.run(tf.compat.v1.global_variables_initializer())

    num_steps = 6001
    ll = []
    atr = []
    av = []

    for step in range(num_steps):
        offset = (step * batch_size) % (y_train.shape[0] - batch_size)
        batch_data = X_train[offset:(offset + batch_size), :]
        batch_labels = y_train[offset:(offset + batch_size), :]

        feed_dict = {tf_train_dataset : batch_data, tf_train_labels : batch_labels}
        _, l, predictions = session.run([optimizer, loss, train_prediction], fe

        if (step % 500 == 0):
            ll.append(l)
            a = accuracy(predictions, batch_labels)
            atr.append(a)
            print("Minibatch loss at step %d: %f" % (step, l))
            print("Minibatch accuracy: %.1f%%" % a)
            a = accuracy(valid_prediction.eval(), y_valid)
            av.append(a)
            print("Validation accuracy: %.1f%%" % a)
            print("Test accuracy: %.1f%%" % accuracy(test_prediction.eval(), y_test))
```

This represents the total number of samples processed so far in the training loop

Here is when the optimization part starts:

```
with tf.compat.v1.Session(graph=graph) as session:
    session.run(tf.compat.v1.global_variables_initializer())

    num_steps = 6001
    ll = []
    atr = []
    av = []

    for step in range(num_steps):
        offset = (step * batch_size) % (y_train.shape[0] - batch_size)
        batch_data = X_train[offset:(offset + batch_size), :]
        batch_labels = y_train[offset:(offset + batch_size), :]

        feed_dict = {tf_train_dataset : batch_data, tf_train_labels : batch_labels}
        _, l, predictions = session.run([optimizer, loss, train_prediction], fe

        if (step % 500 == 0):
            ll.append(l)
            a = accuracy(predictions, batch_labels)
            atr.append(a)
            print("Minibatch loss at step %d: %f" % (step, l))
            print("Minibatch accuracy: %.1f%%" % a)
            a = accuracy(valid_prediction.eval(), y_valid)
            av.append(a)
            print("Validation accuracy: %.1f%%" % a)
            print("Test accuracy: %.1f%%" % accuracy(test_prediction.eval(), y_test))
```

Represents the maximum valid indice value that will not exceed the bounds of the dataset.

Here is when the optimization part starts:

```python
with tf.compat.v1.Session(graph=graph) as session:
    session.run(tf.compat.v1.global_variables_initializer())

    num_steps = 6001
    ll = []
    atr = []
    av = []

    for step in range(num_steps):
        offset = (step * batch_size) % (y_train.shape[0] - batch_size)
        batch_data = X_train[offset:(offset + batch_size), :]
        batch_labels = y_train[offset:(offset + batch_size), :]

        feed_dict = {tf_train_dataset : batch_data, tf_train_labels : batch_labels}
        _, l, predictions = session.run([optimizer, loss, train_prediction], feed_dict=fee

        if (step % 500 == 0):
            ll.append(l)
            a = accuracy(predictions, batch_labels)
            atr.append(a)
            print("Minibatch loss at step %d: %f" % (step, l))
            print("Minibatch accuracy: %.1f%%" % a)
            a = accuracy(valid_prediction.eval(), y_valid)
            av.append(a)
            print("Validation accuracy: %.1f%%" % a)
            print("Test accuracy: %.1f%%" % accuracy(test_prediction.eval(), y_test))
```

By doing the modulo operation at each step, we will find the indices for each mini batch.

This assures that we are getting different minibatches from training and also will wrap around once get to the end of the training dataset.

An example:

X_train:
[[1.76405235 0.40015721 0.97873798]
 [2.2408932   1.86755799 -0.97727788]
 [0.95008842 -0.15135721 -0.10321885]
 [0.4105985   0.14404357 1.45427351]
 [0.76103773 0.12167502 0.44386323]
 [0.33367433 1.49407907 -0.20515826]
 [0.3130677 -0.85409574 -2.55298982]
 [0.6536186   0.8644362 -0.74216502]
 [2.26975462 -1.45436567 0.04575852]
 [-0.18718385 1.53277921 1.46935877]]

```
batch_size = 3

y_train.shape[0] = 10
```

Step 0
Offset: 0
Batch Data:
[[1.76405235 0.40015721 0.97873798]
 [2.2408932   1.86755799 -0.97727788]
 [0.95008842 -0.15135721 -0.10321885]]

Offset = $(0 * 3)\%(10 - 3) = 0$

Step 1
Offset: 3
Batch Data:
[[0.4105985   0.14404357 1.45427351]
 [0.76103773 0.12167502 0.44386323]
 [0.33367433 1.49407907 -0.20515826]]

Offset = $(1 * 3)\%(10 - 3) = 3$

- Since 3 cant be divided by 7, then offset will be 3

Step 2
Offset: 6
Batch Data:
[[0.3130677 -0.85409574 -2.55298982]
 [0.6536186   0.8644362 -0.74216502]
 [2.26975462 -1.45436567 0.04575852]]

Offset = $(2 * 3)\%(10 - 3) = 6$

- Since 6 cant be divided by 7, then offset will be 6

Step 3
Offset: 2
Batch Data:
[[0.95008842 -0.15135721 -0.10321885]
 [0.4105985   0.14404357 1.45427351]
 [0.76103773 0.12167502 0.44386323]]

When the code wraps around once get to almost the end of the training dataset

Offset = $(3 * 3)\%(10 - 3) = 2$

- Since 9 can be divided by 7 once, then offset will be 2

Step 4
Offset: 5
Batch Data:
[[0.33367433 1.49407907 -0.20515826]
 [0.3130677 -0.85409574 -2.55298982]
 [0.6536186   0.8644362 -0.74216502]]

Offset = $(4 * 3)\%(10 - 3) = 5$

- Since 12 can be divided by 7 once, then offset will be 5

Here is when the optimization part starts:

```
with tf.compat.v1.Session(graph=graph) as session:
    session.run(tf.compat.v1.global_variables_initializer())

    num_steps = 6001
    ll = []
    atr = []
    av = []

    for step in range(num_steps):
        offset = (step * batch_size) % (y_train.shape[0] - batch_size)
        batch_data = X_train[offset:(offset + batch_size), :]
        batch_labels = y_train[offset:(offset + batch_size), :]

        feed_dict = {tf_train_dataset : batch_data, tf_train_labels : batch_labels}
        _, l, predictions = session.run([optimizer, loss, train_prediction], feed_dict=feec

        if (step % 500 == 0):
            ll.append(l)
            a = accuracy(predictions, batch_labels)
            atr.append(a)
            print("Minibatch loss at step %d: %f" % (step, l))
            print("Minibatch accuracy: %.1f%%" % a)
            a = accuracy(valid_prediction.eval(), y_valid)
            av.append(a)
            print("Validation accuracy: %.1f%%" % a)
            print("Test accuracy: %.1f%%" % accuracy(test_prediction.eval(), y_test))
```

Creating a dictionary to be fed in to the placeholder that we have defined as input layers

Here is when the optimization part starts:

```
with tf.compat.v1.Session(graph=graph) as session:
    session.run(tf.compat.v1.global_variables_initializer())

    num_steps = 6001
    ll = []
    atr = []
    av = []

    for step in range(num_steps):
        offset = (step * batch_size) % (y_train.shape[0] - batch_size)
        batch_data = X_train[offset:(offset + batch_size), :]
        batch_labels = y_train[offset:(offset + batch_size), :]

        feed_dict = {tf_train_dataset : batch_data, tf_train_labels : batch_labels}
        _, l, predictions = session.run([optimizer, loss, train_prediction], feed_dict=feed_dict)

        if (step % 500 == 0):
            ll.append(l)
            a = accuracy(predictions, batch_labels)
            atr.append(a)
            print("Minibatch loss at step %d: %f" % (step, l))
            print("Minibatch accuracy: %.1f%%" % a)
            a = accuracy(valid_prediction.eval(), y_valid)
            av.append(a)
            print("Validation accuracy: %.1f%%" % a)
            print("Test accuracy: %.1f%%" % accuracy(test_prediction.eval(), y_test))
```

Creating a dictionary to be fed in to the placeholder that we have defined as input layers

Logic behind session run() :

session run() receives 2 inputs, session.run(fetches, feed_dict)

- fetches: This is where you specify what you want to compute or evaluate. It can be a single TensorFlow tensor or operation, or a list of tensors/operations.
- feed_dict (optional): If you have placeholders in your computation graph that need to be provided with actual data, you use a feed_dict dictionary to feed data into those placeholders during the session execution. It's used to map placeholders to the actual data you want to use for that particular run.

Example

```
# Create a session
with tf.compat.v1.Session() as session:
   # Define tensors
   a = tf.constant(3)
   b = tf.constant(5)

   # Add tensors and evaluate
   sum_result = session.run(a + b)
   print("Sum result:", sum_result)  # Output: 8

   # Using a feed_dict
   x = tf.compat.v1.placeholder(tf.float32)
   y = x * 2
   feed_dict = {x: 5.0}
   product_result = session.run(y, feed_dict=feed_dict)
   print("Product result:", product_result)  # Output: 10.0
```

In case of Neural Networks:

session run() receives:

- fetches: [optimizer, loss, train_prediction]
- feed_dict (optional): `feed_dict = {tf_train_dataset : batch_data, tf_train_labels : batch_labels}`

"optimizer" do the following calculations:

- Optimizer will do the forward pass starting with random weights and bias values

$$y^n = h^{(2)}(\sum_{j=0}^{M} w_{kj}^{(2)} h^{(1)}(\sum_{i=0}^{D} w_{ji}^{(1)} x_i + bias1) + bias2)$$

- Then runs the gradient descent algorithm

Computing $\nabla_{\underline{w}} E \; from$ Cross Entropy Loss Function(E): $min_{\underline{w}} - \sum_{n=1}^{N} t^n \log(y^n) + (1 - t^n)\log(1 - y^n)$

Gradient Descent algorithm: $\underline{w^{i+1}} = \underline{w^i} - \propto \nabla_{\underline{w}} E$

In case of Neural Networks:

session run() receives:

- fetches: [optimizer, loss, train_prediction]
- feed_dict (optional): `feed_dict = {tf_train_dataset : batch_data, tf_train_labels : batch_labels}`

“loss” and “train” are given to fetches to calculate the loss over each iteration and get the predicted labels from the training process

Here is when the optimization part starts:

```
with tf.compat.v1.Session(graph=graph) as session:
    session.run(tf.compat.v1.global_variables_initializer())

    num_steps = 6001
    ll = []
    atr = []
    av = []

    for step in range(num_steps):
        offset = (step * batch_size) % (y_train.shape[0] - batch_size)
        batch_data = X_train[offset:(offset + batch_size), :]
        batch_labels = y_train[offset:(offset + batch_size), :]

        feed_dict = {tf_train_dataset : batch_data, tf_train_labels : batch_labels}
        _, l, predictions = session.run([optimizer, loss, train_prediction], feed_dict=feed_dict)

        if (step % 500 == 0):
            ll.append(l)
            a = accuracy(predictions, batch_labels)
            atr.append(a)
            print("Minibatch loss at step %d: %f" % (step, l))
            print("Minibatch accuracy: %.1f%%" % a)
            a = accuracy(valid_prediction.eval(), y_valid)
            av.append(a)
            print("Validation accuracy: %.1f%%" % a)
            print("Test accuracy: %.1f%%" % accuracy(test_prediction.eval(), y_test))
```

Finally we compute some graphs to analyze the loss decay and accuracy

Keras CNN