Building a General Neural Network

1. Theory

Neural networks operate as a sophisticated class of artificial, functionally connected "nodes" that are structurally inspired by the human brain. With an input layer, output layer, and numerous hidden layers, these nodes are connected through a set of weights which function to determine its "activation" levels; if the "activation" levels are above a certain threshold, the node can pass information along to its connections.

Through deep learning algorithms, these neural networks can use training examples to fine-tune their accuracy for a variety of tasks – such as classification problems, natural language processing, recommendation engines, speech recognition, etc.

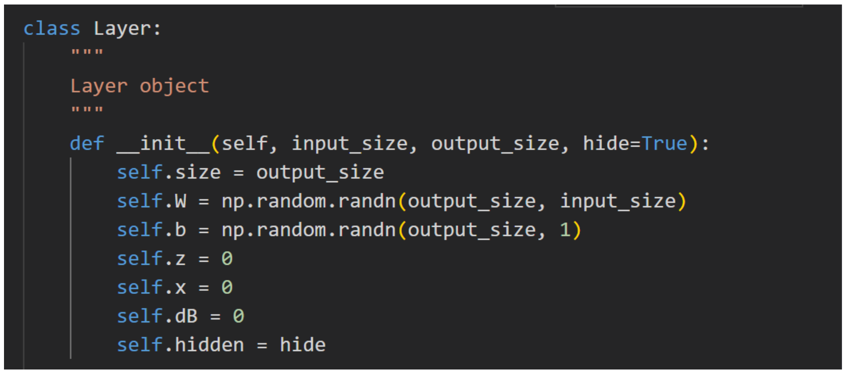
2. Goal

For our project, we aimed to create a general neural network from scratch, forgoing the use of TensorFlow or PyTorch and instead simply using Python and NumPy. By creating a general network, we aim to make the system adaptable, allowing us to control the number of inputs, the learning rate, the shape of the hidden layers, and the number of hidden layers – variables we will explore further within our experimentation section, understanding how each can affect the accuracy of the system.

When creating and experimenting with our network, we will use the MNIST dataset of handwritten numbers. This will act as a classification problem, as the network will take in the 784 pixel values of each digitally handwritten number, and it will output some classification of the number. Specifically, it will output the probability of the input being each, single-digit number, 0 - 9.

3. Implementation

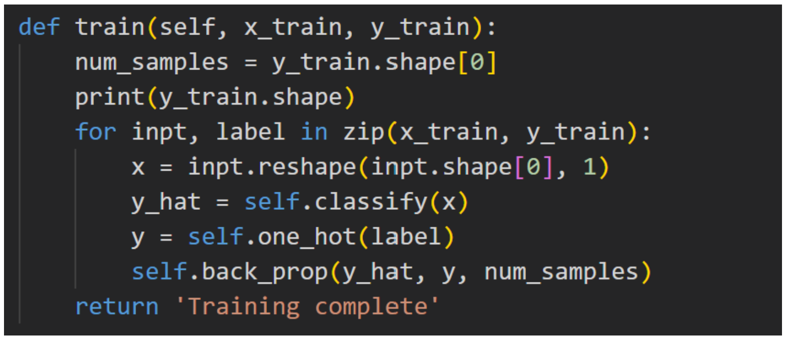
To implement the network, we used object-oriented programming, defining classes of layers and networks. In the Layer class, we are able to initialize the size, weights, biases, weighted inputs (as z), activation of the previous layer (as x), and the status of the layer as hidden or not hidden.



```python
class Layer:
    """
    Layer object
    """
    def __init__(self, input_size, output_size, hide=True):
        self.size = output_size
        self.W = np.random.randn(output_size, input_size)
        self.b = np.random.randn(output_size, 1)
        self.z = 0
        self.x = 0
        self.dB = 0
        self.hidden = hide
```

We are also able to define ReLU and softmax as well as self-made functions such as feed_forward, update_param, and back_prop, all of which execute the mathematical computations necessary for the learning. For the softmax function, we subtract the maximum value of the weighted input z from each element to help with numerical stability. In the Network class, we are able to call these functions more specifically as we guide the network through each training example.

In our network class, in addition to initializing the parameters, we have a “train” function, which does the heavy-lifting of iterating all of the training examples through the network. First, the train function calls upon a classify function – which is defined within the network itself. Network.classify, upon intaking, x (the activation of the previous layer), is able to iterate through all layers of a network, continuously calling layer.feed_forward in order to calculate the activation of the current layer. This moves the training example through the network until it finally generates an output with softmax, labeled as y_hat.



```python
def train(self, x_train, y_train):
    num_samples = y_train.shape[0]
    print(y_train.shape)
    for inpt, label in zip(x_train, y_train):
        x = inpt.reshape(inpt.shape[0], 1)
        y_hat = self.classify(x)
        y = self.one_hot(label)
        self.back_prop(y_hat, y, num_samples)
    return 'Training complete'
```

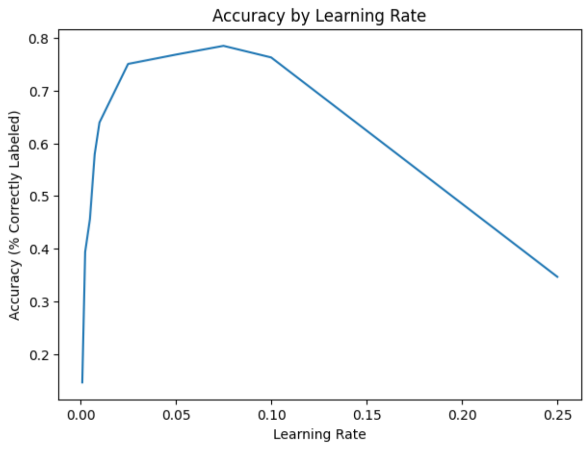
Following the classification of a training example, network.train, using a defined one_hot function, generates the actual, expected output value of the network, which is, alongside _hat, passed into network.back_prop. Network.back_prop calls the layer class back_prop function, which calculates the respective dW and db and updates these parameters accordingly.

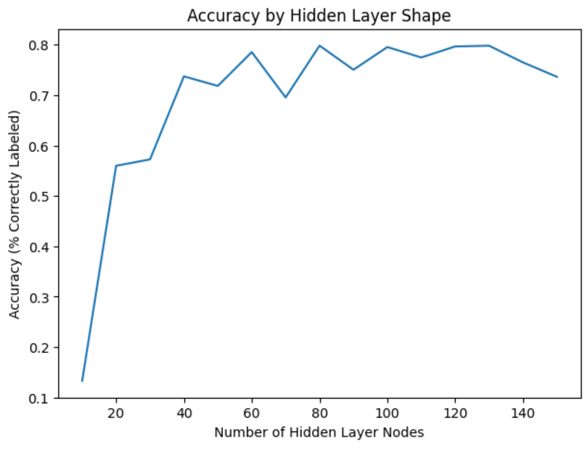
Upon iterating through all training examples, we are able to use the network’s “test” function, which cycles through the testing data, classifies each image, and then, comparing it to the correct label, enumerates the correctly classified examples.

4. Experimentation

After completing the network, we aimed to experiment with the hyperparameters, evaluating how each would affect the accuracy of the classifications. First, we adjusted the learning rate – which controls how much the network adjusts for each training example. We found that, with a network of one hidden layer with 70 nodes, accuracy peaked when the learning rate was between .05 and .1.

We also experimented with the amount of nodes in the hidden layer. We found that, with a learning rate of .075, maximum accuracy was attained when the hidden layer had anywhere from about 80 - 130 nodes. Furthermore, we found that because we have a large number of input features, increasing the depth of the network beyond one hidden layer reduces the numerical stability of the network, and we experienced overflow errors. Thus, our experimentation focused on optimizing the shape of the single allowable hidden layer and the learning rate.





5. References

Parr, Terence, and Jeremy Howard. "The matrix calculus you need for deep learning." *arXiv preprint arXiv:1802.01528* (2018).

Nielsen, Michael A. *Neural Networks and Deep Learning*, 1 Jan. 1970, Dominion Press, neuralnetworksanddeeplearning.com/chap1.html

3Blue1Brown. *Neural Networks Series*, 2018, YouTube, https://www.youtube.com/playlist?list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi.

Lague, Sebastian. “How to Create a Neural Network (and Train It to Identify Doodles).” *YouTube*, 12 Aug. 2022, youtu.be/hfMk-kjRv4c.