1. Forward prop ⇒ using existing w, b matrices for each layer

2. Backward prop ⇒ update w, b matrices for each layer based on current w, b values and the stipulated loss function ???

3. Repeat

Functions:

```
forward_prop_multiclass(X, W, B):
# use activation functions (ReLu)
# Return activation list (a)
```

```
backward_prop_multiclass(X, y, W, B, a):
# use chain rule to execute backprop
and update w, b for each layer accordingly
# Update based on loss function
# Return W, B
```

```
train_nn_multiclass(X, y, layer_list,
epochs):
# Generate W, B or prepopulate
# Iterate forward prop, backprop, update
for i in range(epochs)
# Return W, B
```

Data Structures:

X: Numpy array, matrix, m x n

y: Numpy array, matrix, m x 1

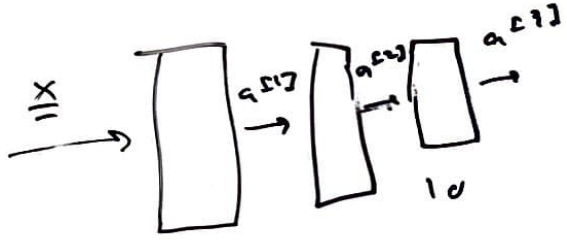
W: Python list of Numpy arrays (matrices), # corresponds to # of layers

B: Python list of Numpy arrays (matrices), # corresponds to # of layer

layer_list: Python list of ints, defining the # of units/layer

a: Python list of Numpy arrays

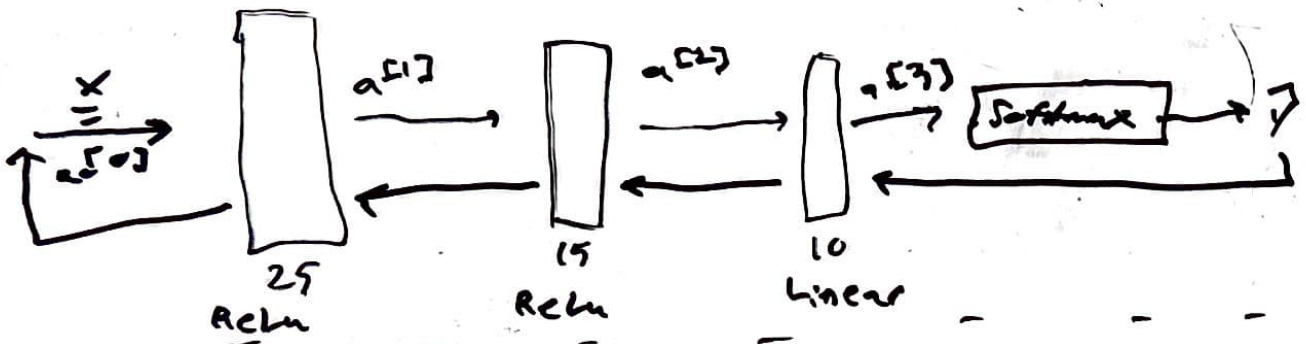
NN Structure



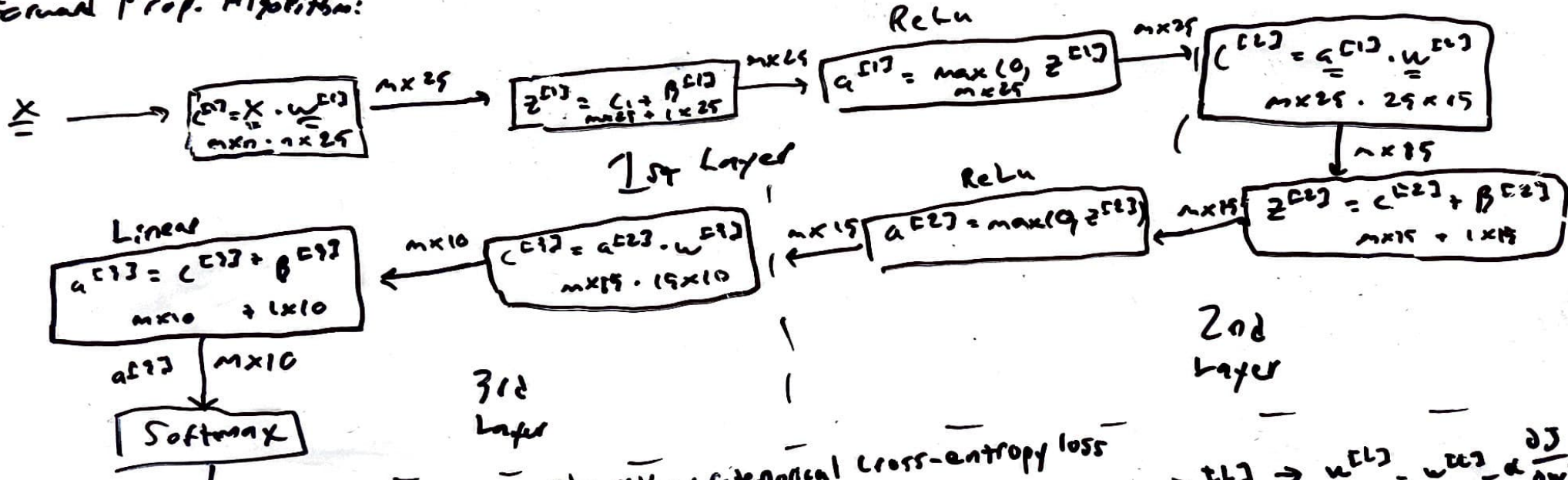
X: m x n: m training examples, n features per example

W: n x k: n weights per unit, k units

Training Algorithm:



Forward Prop. Algorithm:



Backprop. Algorithm: Categorical cross-entropy loss

$$\frac{\partial J}{\partial W^{[L]}} = \delta^{[L]} \cdot a^{[L-1]} = (\delta^{[L+1]} \cdot W^{[L+1]}) \circ Z^{[L]} \Rightarrow W^{[L]} = W^{[L]} - \alpha \frac{\partial J}{\partial W^{[L]}}$$

$$\frac{\partial J}{\partial \beta^{[L]}} = \frac{1}{m} \left(\sum_{j=1}^{m} \delta^{[L](j)} \right) \Rightarrow \beta^{[L]} = \beta^{[L]} - \alpha \frac{\partial J}{\partial \beta^{[L]}}$$