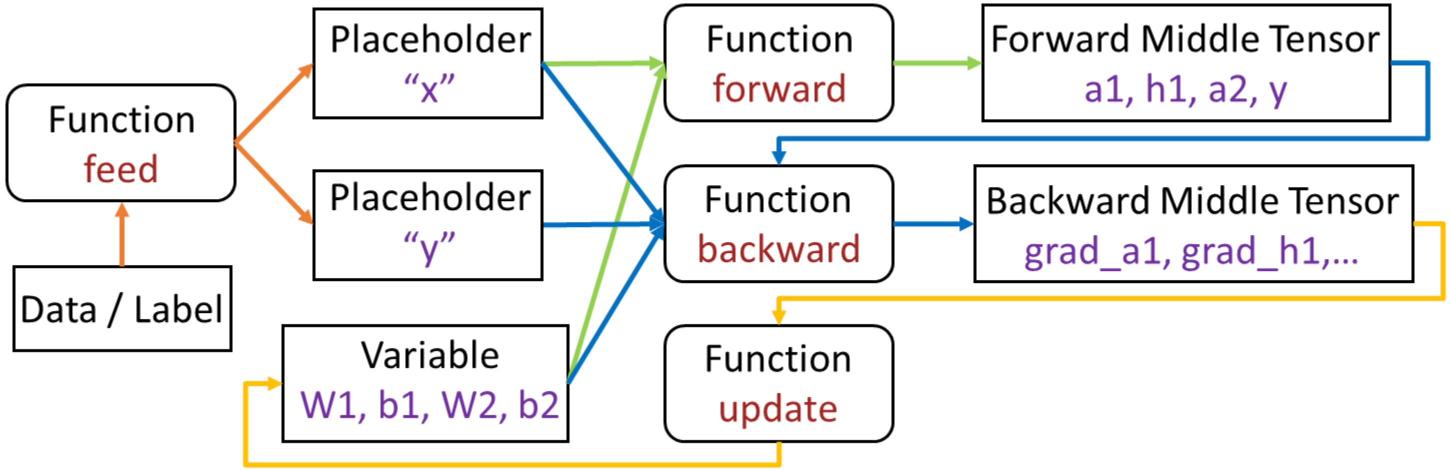
Machine Learning HW1

Neural Network Implementation

Framework




```
import NeuralNetwork
# Create Object (linear/sigmoid/softmax)
nn = NeuralNetwork.NN(input, hidden, output, “linear”)
# Training Iteration
for i in range(10001):
    nn.feed({x:x_data, y:y_data}) # Feed Placeholder
    y_out = nn.forward() # Forward Propagation
    nn.backward() # Backward Propagation
    nn.update(1e-3) # Update Weights
    loss = nn.computeLoss() # Compute Loss
```

Empty Class

```
import numpy as np

class NN():
    def __init__(self, input_size, hidden_size, output_size, activation):
        pass
    # Feed Placeholder
    def feed(self, feed_dict):
        pass
    # Forward Propagation
    def forward(self):
        pass
    # Backward Propagation
    def backward(self):
        pass
    # Update Weights
    def update(self, learning_rate):
        pass
    # Loss Functions
    def computeLoss(self):
        pass
```

Initialize & Feed Placeholder

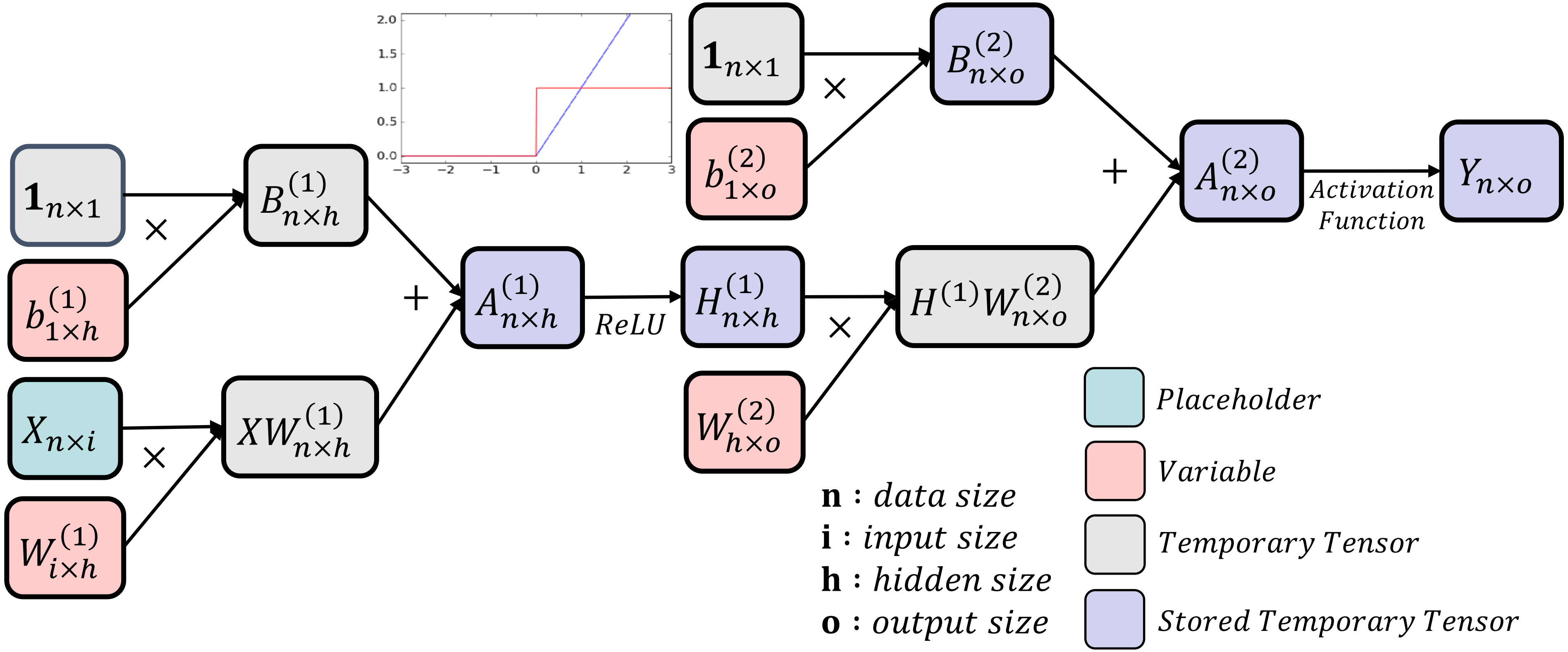
```
def __init__(self, input_size, hidden_size, output_size, activation):
        # Trainable Variables
        self.W1 = np.random.randn(input_size, hidden_size) / np.sqrt(input_size)
        self.b1 = np.zeros([1, hidden_size])
        self.W2 = np.random.randn(hidden_size, output_size) / np.sqrt(input_size)
        self.b2 = np.zeros([1, output_size])

        # Activation Function & Placeholders
        self.activation = activation # "linear"/"softmax"/"sigmoid"
        self.placeholder = {"x":None, "y":None}
```

Xavier Initialize

```
# Feed Placeholder
def feed(self, feed_dict):
    for key in feed_dict:
        self.placeholder[key] = feed_dict[key].copy()
```

Forward Computation




```python
n = self.placeholder["x"].shape[0]
self.a1 = self.placeholder["x"].dot(self.W1) + np.ones((n,1)).dot(self.b1)
self.h1 = np.maximum(self.a1,0) # ReLU Activation
self.a2 = self.h1.dot(self.W2) + np.ones((n,1)).dot(self.b2)
```

Output Activation Function

Sigmoid

$$sigmoid(a_i) = \frac{1}{1 + e^{-a_i}}$$

label	1	2	3	4	5
Prob.	0.5	0.1	0.8	0.2	0.3

Each data belong to **multiple** class

Softmax

$$softmax(a_i) = \frac{e^{a_i}}{\sum_i e^{a_i}}$$

label	1	2	3	4	5
Prob.	0.1	0.2	0.4	0.2	0.1

Each data belong to **only one** class

```
# Linear Activation
if self.activation == "linear":
    self.y = self.a2.copy()
# Softmax Activation
elif self.activation == "softmax":
    self.y_logit = np.exp(self.a2 - np.max(self.a2, 1, keepdims=True))
    self.y = self.y_logit / np.sum(self.y_logit, 1, keepdims=True)
# Sigmoid Activation
elif self.activation == "sigmoid":
    self.y = 1.0 / (1.0 + np.exp(-self.a2))
```

Loss Function

Mean–Square Loss

$$E = \frac{1}{2}\sum_{i}^{n}(y_i - t_i)^2$$

Cross–Entropy Loss (Binary)

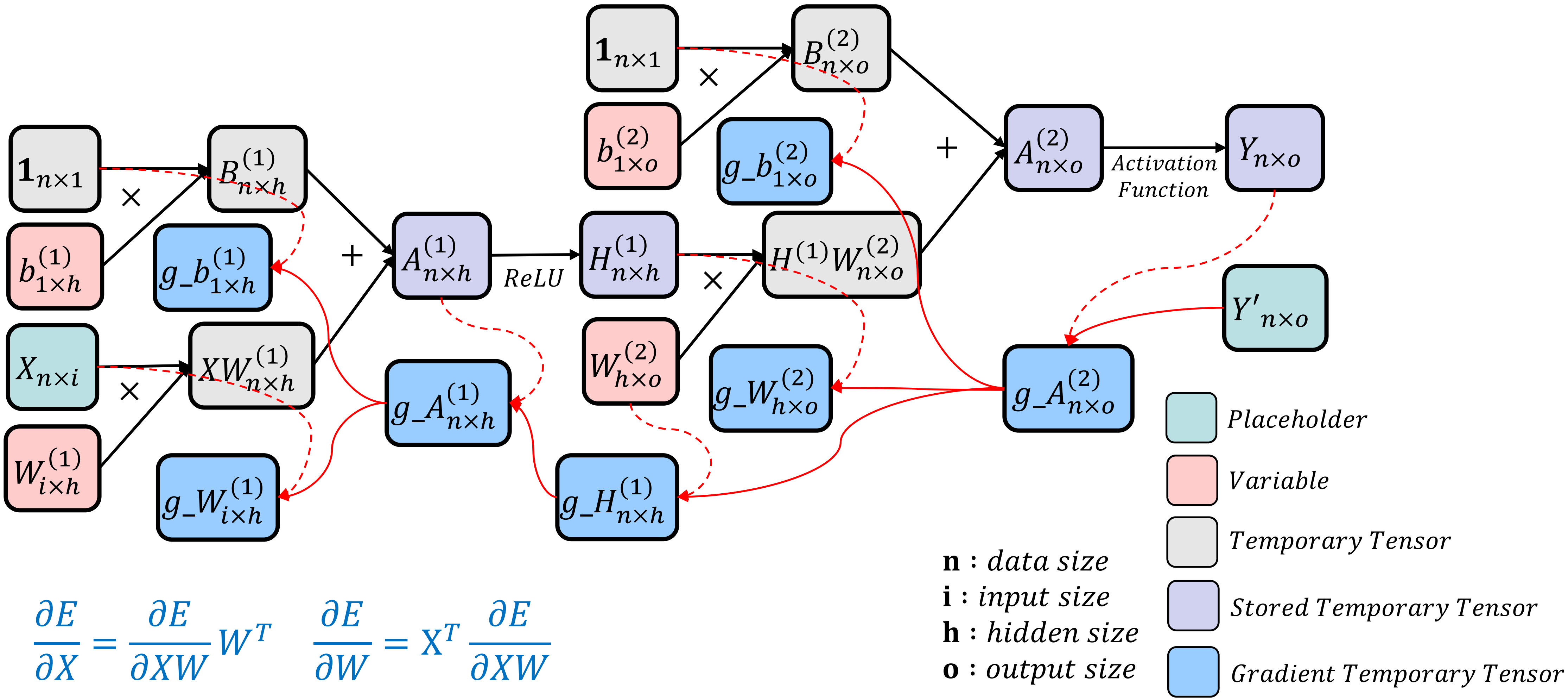
$$E = -\sum_{i}^{n} t_i \log y_i + (1 - t_i)\log(1 - t_i)$$

Cross–Entropy Loss (Multi–Classes)

$$E = -\sum_{i}^{n} t_i \log y_i$$

```python
# Loss Functions
def computeLoss(self):
    loss = 0.0
    # Mean Square Error
    if self.activation == "linear":
        loss = 0.5 * np.square(self.y - self.placeholder["y"]).mean()
    # Softmax Cross Entropy
    elif self.activation == "softmax":
        loss = -self.placeholder["y"] * np.log(self.y + 1e-6)
        loss = np.sum(loss, 1).mean()
    # Sigmoid Cross Entropy
    elif self.activation == "sigmoid":
        loss = -self.placeholder["y"] * np.log(self.y + 1e-6) - \
        (1-self.placeholder["y"]) * np.log(1-self.y + 1e-6)
        loss = np.mean(loss)
    return loss
```

Backward Computation



$$\frac{\partial E}{\partial X} = \frac{\partial E}{\partial XW} W^T \qquad \frac{\partial E}{\partial W} = X^T \frac{\partial E}{\partial XW}$$

Backward Propagation

Mean–Square Loss

$$E = \frac{1}{2}\sum_{i}^{n}(y_i - t_i)^2$$

Linear + Mean–Square

$$\frac{\partial E}{\partial y_i} = y_i - t_i$$

$$\frac{\partial y_i}{\partial a_i} = 1$$

$$\frac{\partial E}{\partial a_i} = \frac{\partial E}{\partial y_i}\frac{\partial y_i}{\partial a_i} = \boldsymbol{y_i - t_i}$$

Cross–Entropy Loss (Binary)

$$E = -\sum_{i}^{n} t_i \log y_i + (1 - t_i)\log(1 - t_i)$$

Sigmoid + Cross–Entropy

$$\frac{\partial E}{\partial y_i} = \frac{-t_i}{y_i} + \frac{1 - t_i}{1 - y_i} = \frac{y_i - t_i}{y_i(1 - y_i)}$$

$$\frac{\partial y_i}{\partial a_i} = y_i(1 - y_i)$$

$$\frac{\partial E}{\partial a_i} = \frac{\partial E}{\partial y_i}\frac{\partial y_i}{\partial a_i} = \boldsymbol{y_i - t_i}$$

Cross–Entropy Loss (Multi–Classes)

$$E = -\sum_{i}^{n} t_i \log y_i$$

Softmax + Cross–Entropy

$$\frac{\partial E}{\partial y_i} = -\frac{t_i}{y_i}$$

$$\frac{\partial y_i}{\partial a_j} = \begin{cases} y_i(1 - y_i), & i = j \\ -y_i y_j, & i \neq j \end{cases}$$

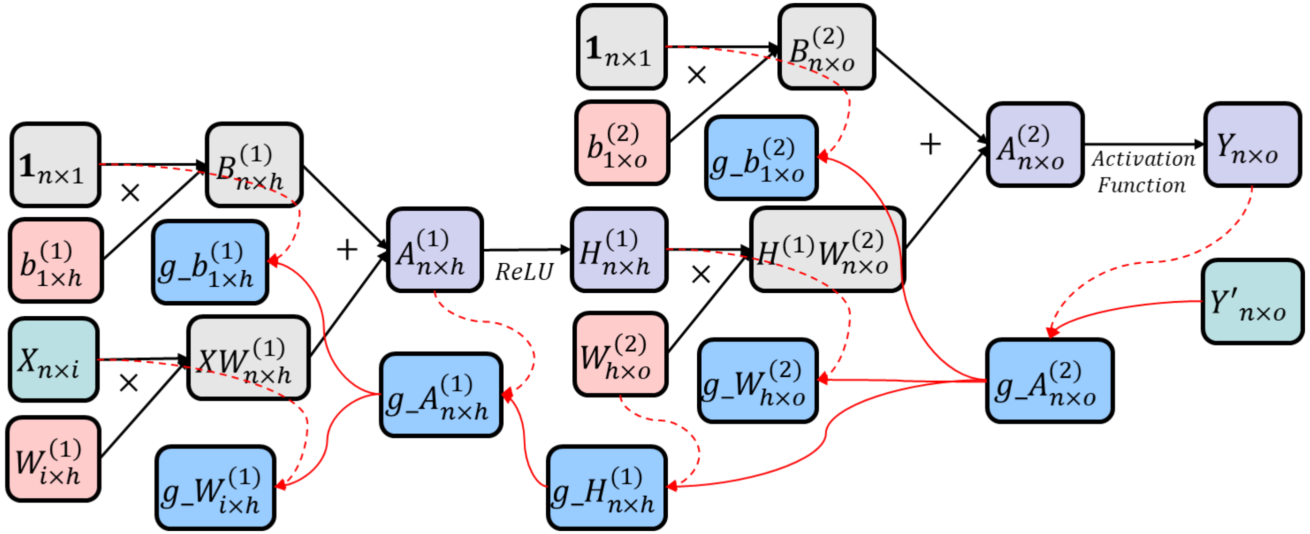
$$\frac{\partial E}{\partial a_i} = \frac{\partial E}{\partial y_i}\frac{\partial y_i}{\partial a_i} + \sum_{j \neq i}\frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial a_i}$$

$$= -t_i(1 - y_i) + \sum_{j \neq i} t_j y_i$$

$$= -t_i + y_i \sum_{j} t_j = \boldsymbol{y_i - t_i}$$

https://www.ics.uci.edu/~pjsadows/notes.pdf

Backward Propagation




```python
# Backward Propagation
def backward(self):
    n = self.placeholder["y"].shape[0]
    self.grad_a2 = (self.y - self.placeholder["y"]) / n
    self.grad_b2 = np.ones((n, 1)).T.dot(self.grad_a2)
    self.grad_W2 = self.h1.T.dot(self.grad_a2)
    self.grad_h1 = ...
    self.grad_a1 = ...
    self.grad_b1 = ...
    self.grad_W1 = ...
```

Update Weights

$\eta: Learning\ Rate$

$$W_{t+1}^{(2)} = W_t^{(2)} - \eta \frac{\partial E}{\partial W^{(2)}} \qquad W_{t+1}^{(1)} = W_t^{(1)} - \eta \frac{\partial E}{\partial W^{(1)}}$$

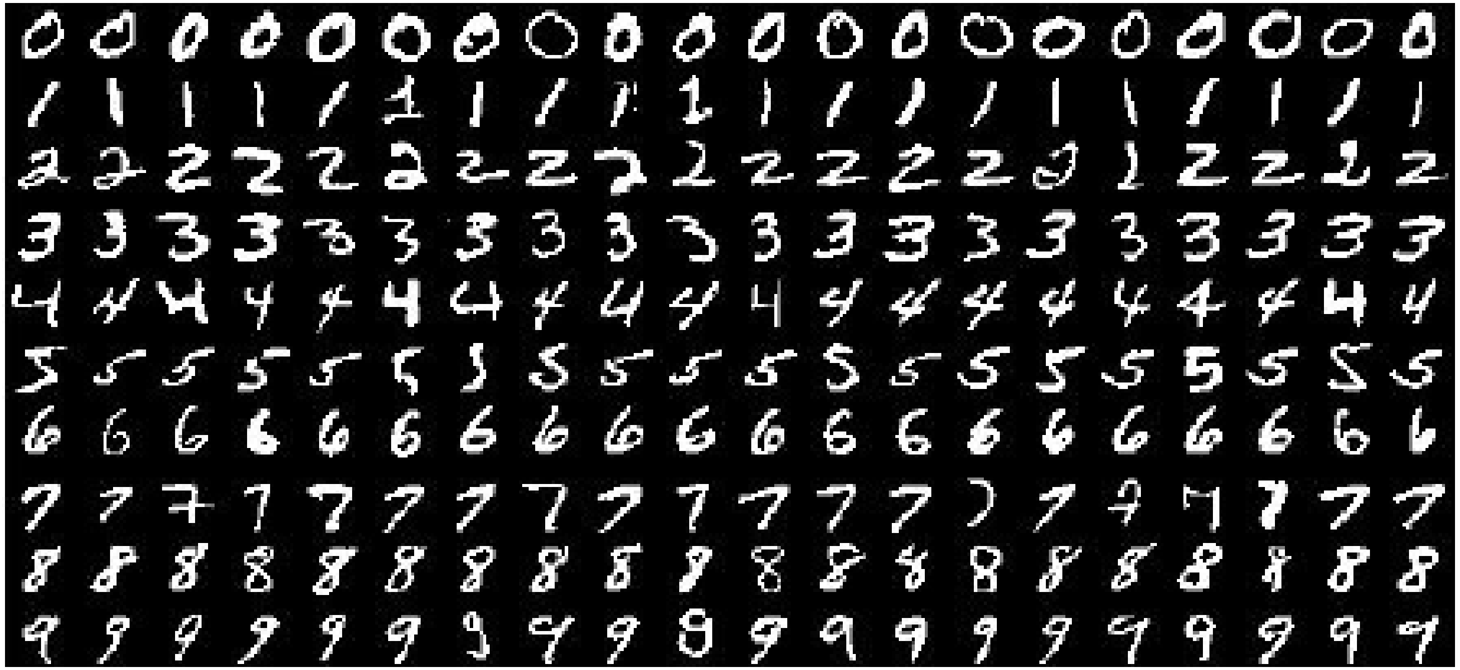
$$b_{t+1}^{(2)} = b_t^{(2)} - \eta \frac{\partial E}{\partial b^{(2)}} \qquad b_{t+1}^{(1)} = b_t^{(1)} - \eta \frac{\partial E}{\partial b^{(1)}}$$

```
# Update Weights
def update(self, learning_rate=1e-3):
    self.W1 = self.W1 - learning_rate * self.grad_W1
    self.b1 = self.b1 - learning_rate * self.grad_b1
    self.W2 = self.W2 - learning_rate * self.grad_W2
    self.b2 = self.b2 - learning_rate * self.grad_b2
```

MNIST Datasets

- Image Size: 28x28
- Label: 0~9
- Train/Test: 60,000/10,000

http://yann.lecun.com/exdb/mnist/

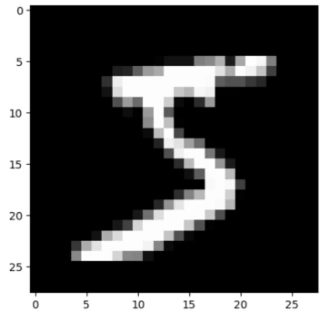


Read MNIST Dataset

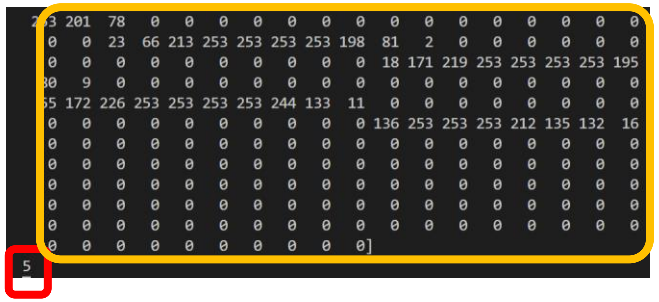
```python
# Dataset
MNISTtools.downloadMNIST(path='MNIST_data', unzip=True)
x_train, y_train = MNISTtools.loadMNIST(dataset="training", path="MNIST_data")
x_test, y_test = MNISTtools.loadMNIST(dataset="testing", path="MNIST_data")

# Show Data and Label
print(x_train[0])
print(y_train[0])
plt.imshow(x_train[0].reshape((28,28)), cmap='gray')
plt.show()

# Data Processing
x_train = x_train.astype(np.float32) / 255.
x_test = x_test.astype(np.float32) / 255.
y_train = OneHot(y_train)
y_test = OneHot(y_test)
```

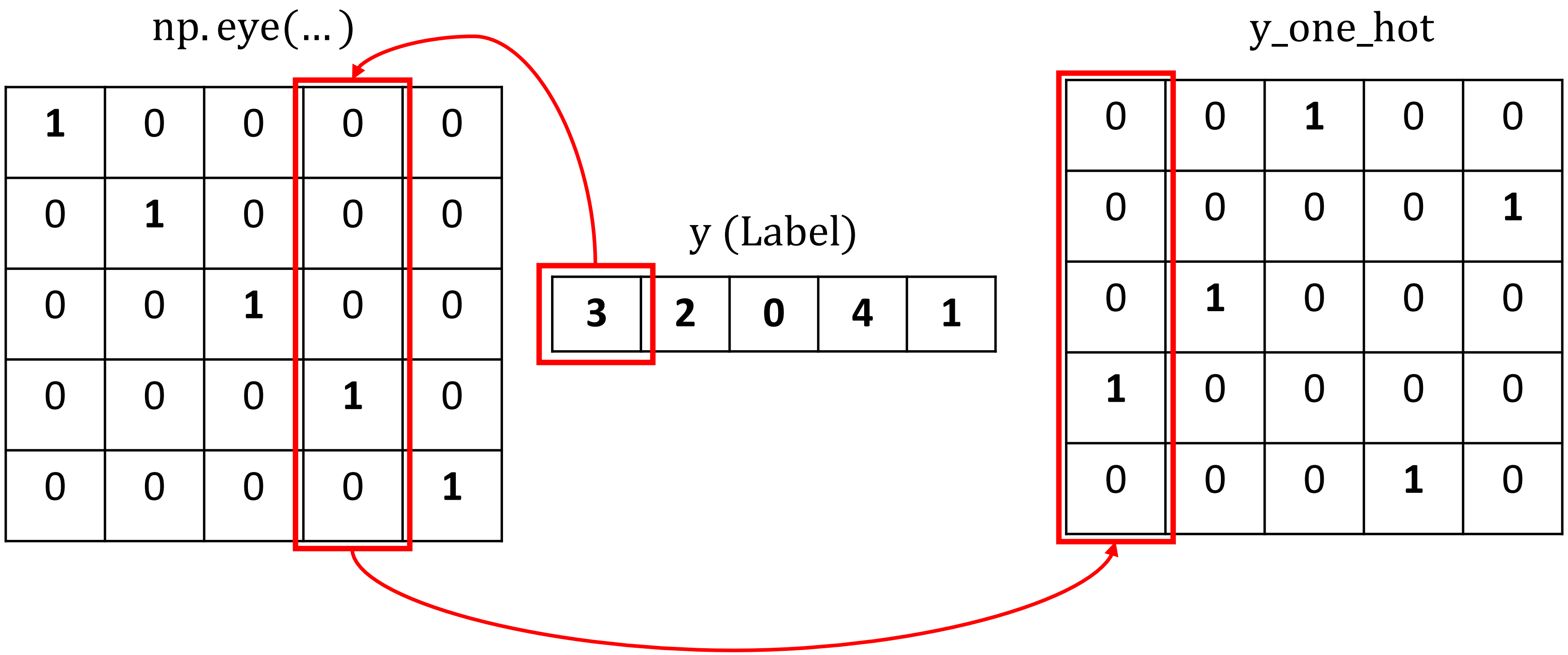


Need to be normalized



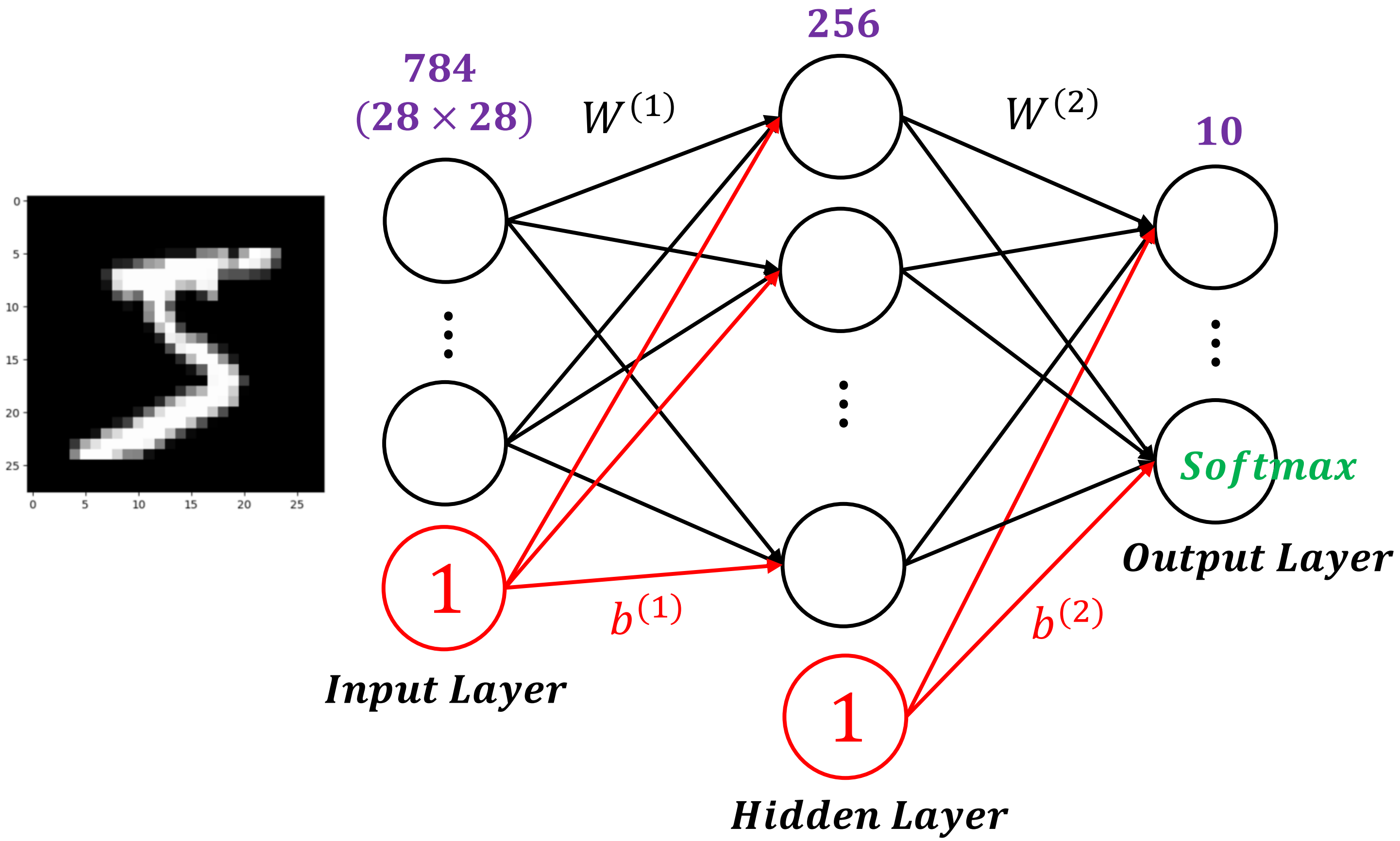
Need to transform to one-hot key

One-Hot Key




```python
# Digit Label to One-Hot Key
def OneHot(y):
    y_one_hot = np.eye(10, dtype=np.float32)[y]
    return y_one_hot
```

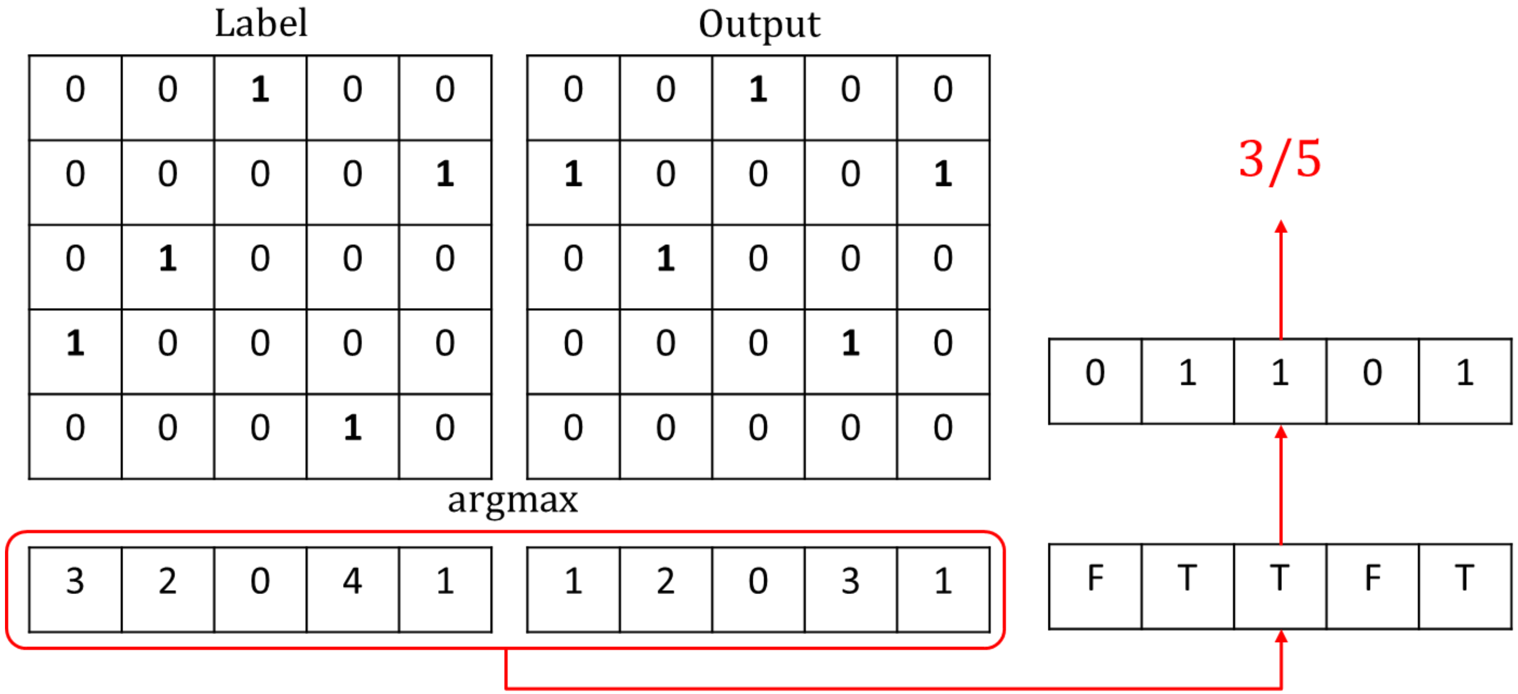
Create NN Model



label	value
0	**0**
1	**0**
2	**0**
3	**0**
4	**0**
5	**1**
6	**0**
7	**0**
8	**0**
9	**0**

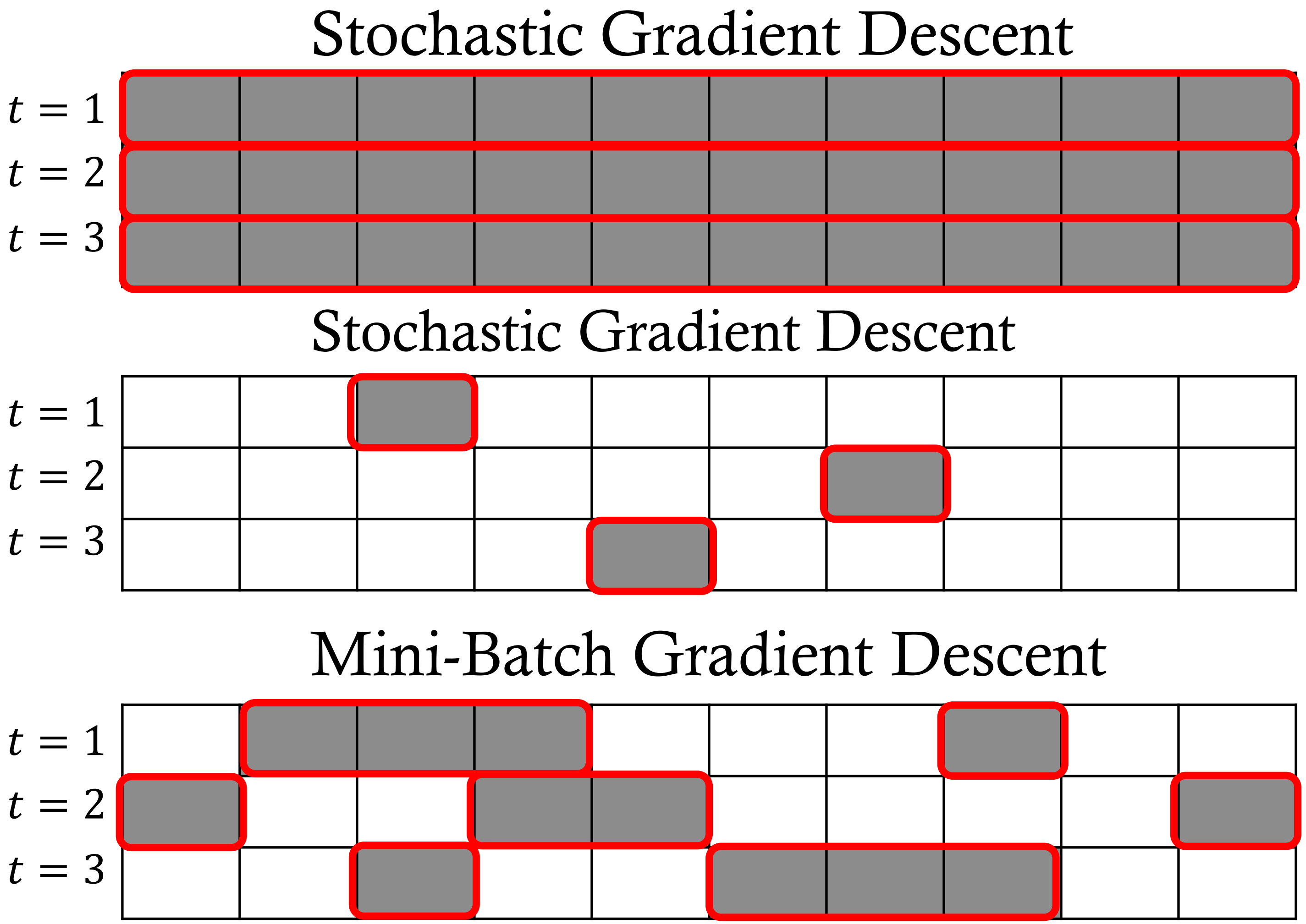
```
# Create NN Model
nn = NeuralNetwork.NN(784,256,10,"softmax")
```

Compute Accuracy



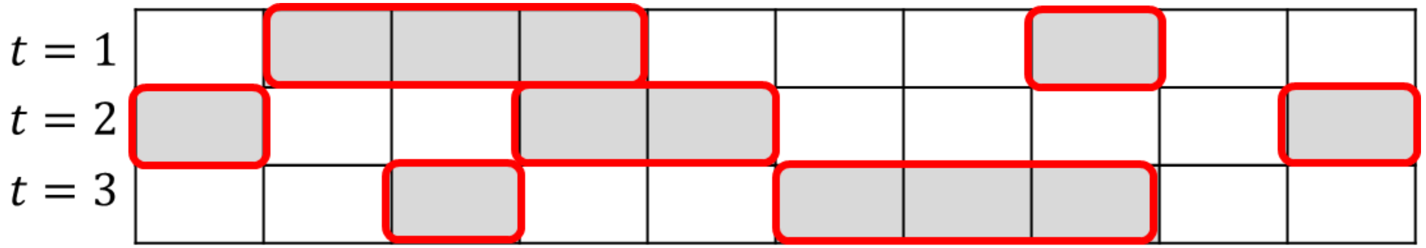

```python
# Compute Accuracy
def Accuracy(y,y_):
    y_digit = np.argmax(y,1)
    y_digit_ = np.argmax(y_,1)
    temp = np.equal(y_digit, y_digit_).astype(np.float32)
    return np.sum(temp) / float(y_digit.shape[0])
```

Sample Data Batch



Sample Data Batch

- Mini-Batch Gradient Descent




```
# Sample Data Batch
batch_id = np.random.choice(data_size, batch_size)
x_batch = x_train[batch_id]
y_batch = y_train[batch_id]
```

Train the Model

```
# Sample Data Batch
batch_id = np.random.choice(x_train.shape[0], batch_size)
x_batch = x_train[batch_id]
y_batch = y_train[batch_id]

# Forward & Backward & Update
nn.feed({"x":x_batch, "y":y_batch})
nn.forward()
nn.backward()
nn.update(1e-2)

# Loss
loss = nn.computeLoss()
loss_rec.append(loss)

# Evaluation
batch_id = np.random.choice(x_test.shape[0], batch_size)
x_test_batch = x_test[batch_id]
y_test_batch = y_test[batch_id]
nn.feed({"x":x_test_batch})
y_test_out = nn.forward()
acc = Accuracy(y_test_out, y_test_batch)

if i%100 == 0:
    print("\r[Iteration {:5d}] Loss={:.4f} | Acc={:.3f}".format(i,loss,acc))
```

Digits Classification

- 1. Implement two neural networks with (a) wide hidden layer and (b) deep hidden layer to classify the digits in MNIST dataset. You have to show the accuracy and loss curve of the training and testing data for each model.

The details of the wide model:
(# of parameters: 203530)

Wide Model	Neurons	Activation
Input Layer	784	-
Hidden Layer	256	ReLU
Output Layer	10	Softmax

The details of the deep model:
(# of parameters: 203170)

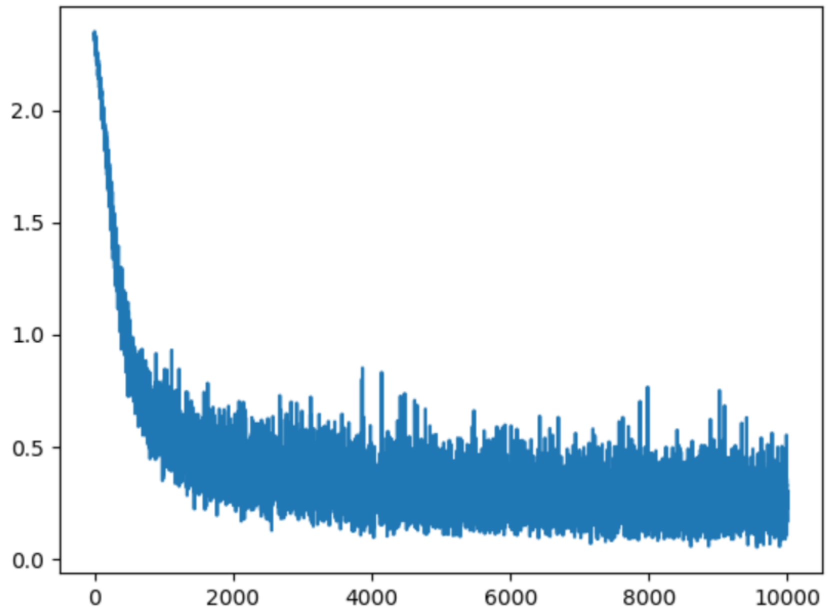
Deep Model	Neurons	Activation
Input Layer	784	-
Hidden Layer 1	204	ReLU
Hidden Layer 2	202	ReLU
Output Layer	10	Softmax

Digits Classification

Training

```
[Iteration     0] Loss=2.314301 | Acc=0.093750
[Iteration   100] Loss=1.998981 | Acc=0.593750
[Iteration   200] Loss=1.744965 | Acc=0.812500
[Iteration   300] Loss=1.401926 | Acc=0.796875
[Iteration   400] Loss=1.164139 | Acc=0.781250
[Iteration   500] Loss=0.871977 | Acc=0.734375
[Iteration   600] Loss=0.683609 | Acc=0.906250
[Iteration   700] Loss=0.559228 | Acc=0.828125
[Iteration   800] Loss=0.508052 | Acc=0.859375
[Iteration   900] Loss=0.568591 | Acc=0.875000
[Iteration  1000] Loss=0.480719 | Acc=0.859375
[Iteration  1100] Loss=0.462497 | Acc=0.906250
```

Loss Curve



Feature Learning

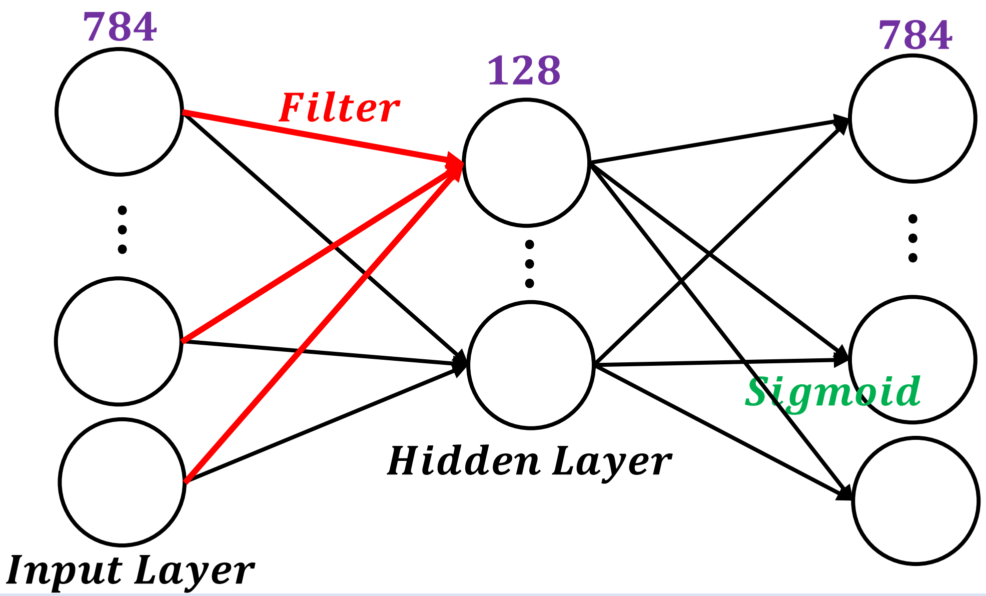
2. Implement an autoencoder (AE) to learn the representation of the MNIST datasets.

(a) Visualize the reconstruction results and the first 16 filters.

(b) (Bonus) Apply **denoising** and **dropout** mechanism, and visualize the reconstruction results and the filters.

The details of the autoencoder:

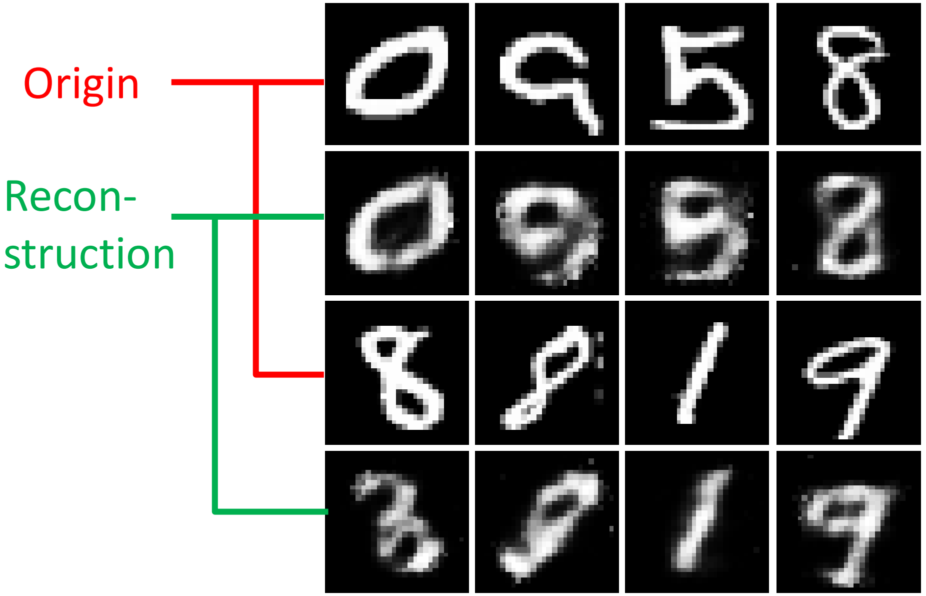
	Neurons	Activation
Input Layer	784	-
Hidden Layer	128	ReLU
Output Layer	784	Sigmoid



Feature Learning

- Visualization example

Reconstruction Results



Filters (dAE + dropout)