Implementing a Multilayer Artificial Neural Network from Scratch

Hsi-Pin Ma 馬席彬

http://lms.nthu.edu.tw/course/40724

Department of Electrical Engineering

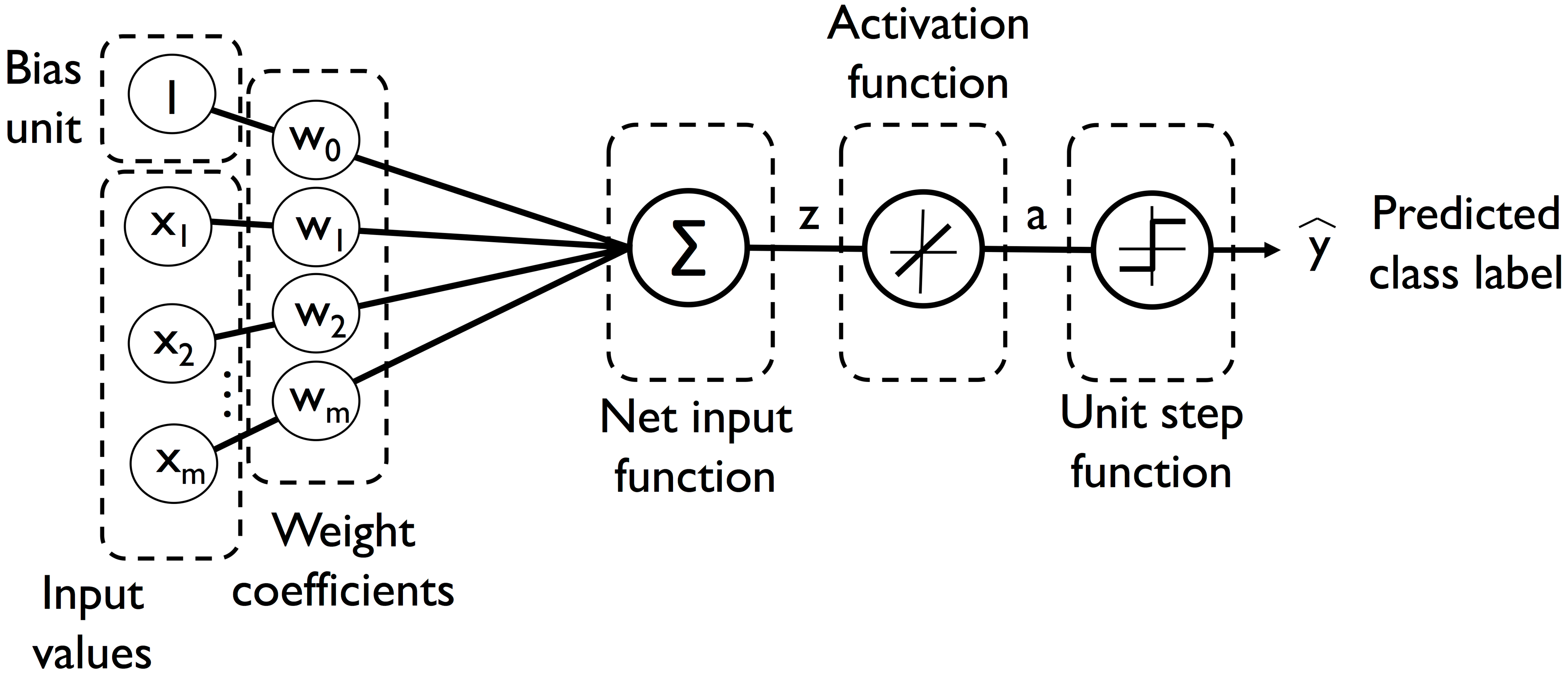
National Tsing Hua University

Outline

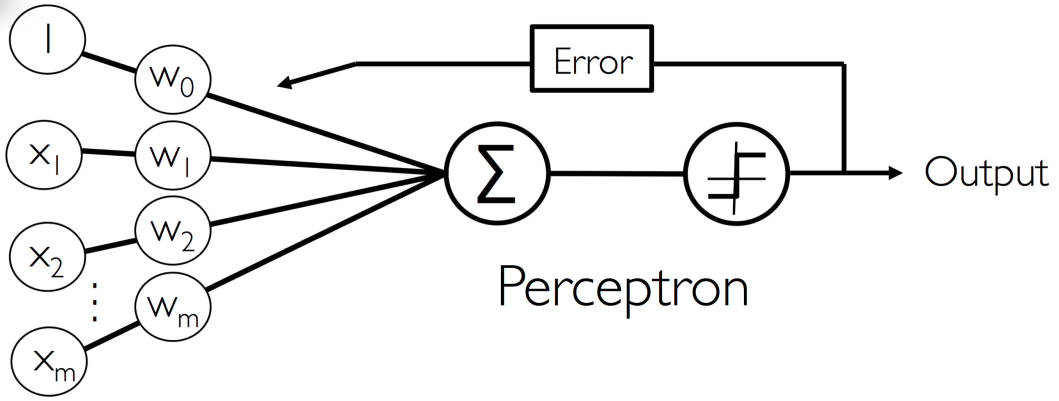
- Modeling complex functions with artificial neural networks
- Classifying handwritten digits
- Training an artificial neural network
- Convergence in neural networks

Modeling Complex Functions with Artificial Neural Networks

Single Layer Neural Network Recap



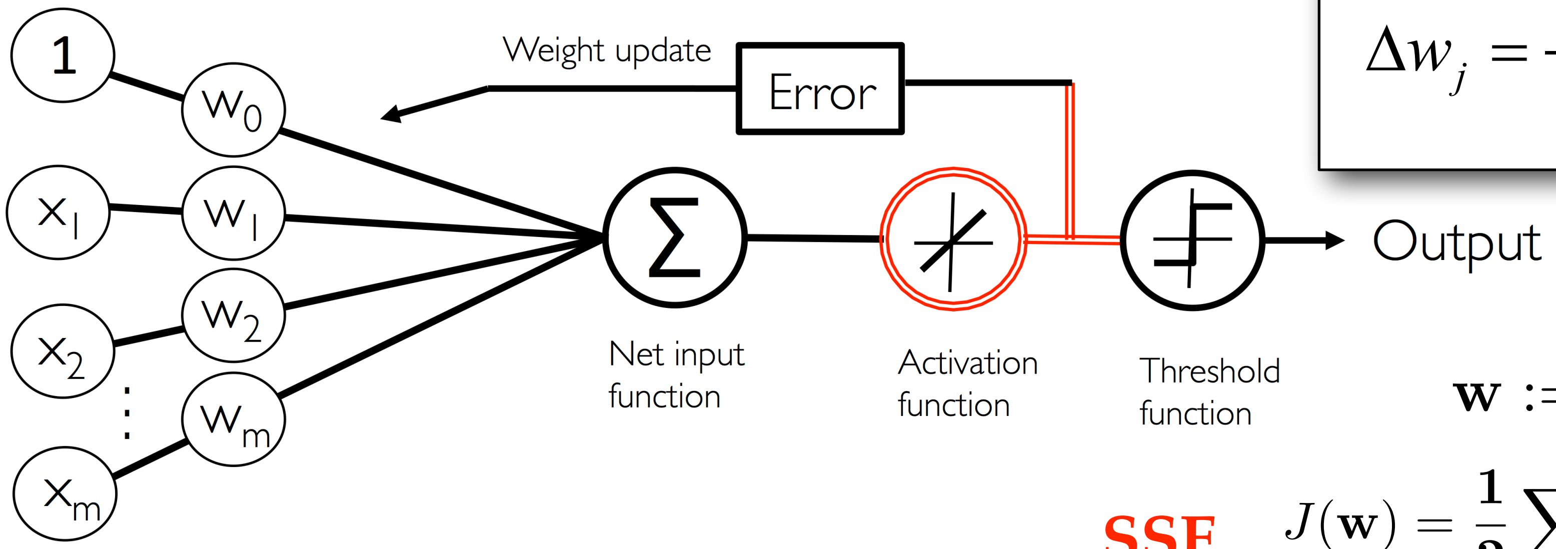
Adaline



- Perceptron
  - For each training sample $\mathbf{x}^{(i)}$
    - compute the prediction value $\hat{y}^{(i)}$ with current vector $\mathbf{w}^{(i)}$
    - Update the weights

$$\boxed{\Delta\mathbf{w}^{(i)} = \eta(\mathbf{y}^{(i)} - \hat{\mathbf{y}}^{(i)})\mathbf{x}^{(i)}}$$

- Adaline
  - Weight update done after entire training set has been seen



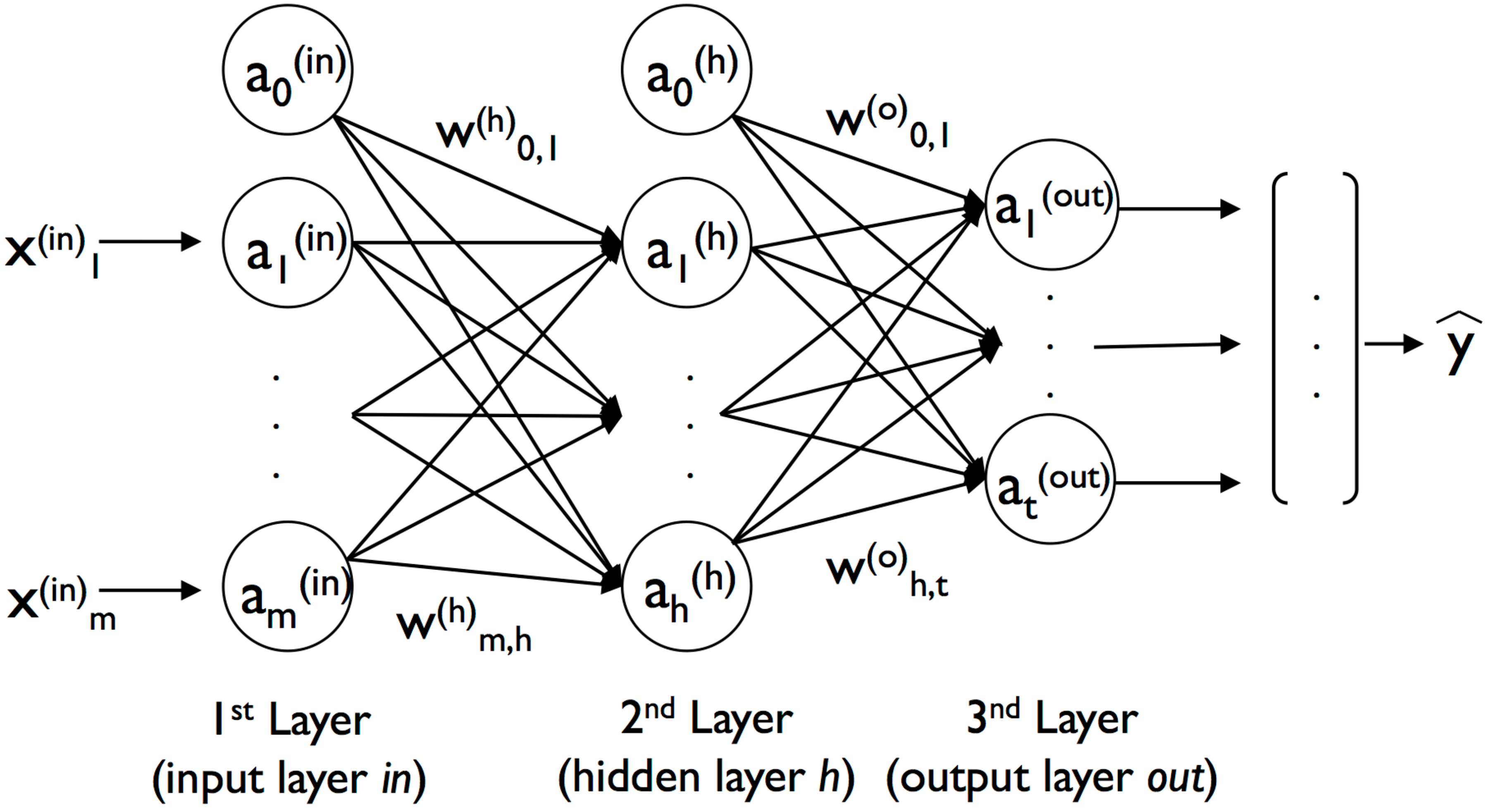
$$\boxed{\Delta w_j = -\eta\frac{\partial J}{\partial w_j} = \eta\sum_i\left(y^{(i)} - \phi\left(z^{(i)}\right)\right)x_j^{(i)}}$$

$$\mathbf{w} := \mathbf{w} + \Delta\mathbf{w} \qquad \Delta\mathbf{w} = -\eta\nabla\mathbf{J}(\mathbf{w})$$

SSE $J(\mathbf{w}) = \frac{1}{2}\sum_{i}(\mathbf{y}^{(i)} - \phi(\mathbf{z}^{(i)}))^2 = \frac{1}{2}\sum_{i}(\mathbf{y}^{(i)} - \mathbf{w}^{(i)}\mathbf{x}^{(i)})^2$

Multilayer Feedforward Neural Network

- Multilayer Perceptron (MLP) with 3 layers



- Deep artificial neural network
  - A network has more than one hidden layer

Notation

- *i*th activation unit in the *l*th layer as $a_i^{(l)}$
- The activation units $a_0^{(in)}$ and $a_0^{(h)}$ are the ***bias units***, respectively, which we set equal to 1
- The activation of the units in the input layer

$$a^{(in)} = \begin{bmatrix} a_0^{(in)} \\ a_1^{(in)} \\ \vdots \\ a_m^{(in)} \end{bmatrix} = \begin{bmatrix} 1 \\ x_1^{(in)} \\ \vdots \\ x_m^{(in)} \end{bmatrix}$$

- The connection between ***k***th unit in layer ***l*** to the ***j***th unit in layer ***l***+1 written as $w_{\kappa,j}^{(l)}$

Notation Summary

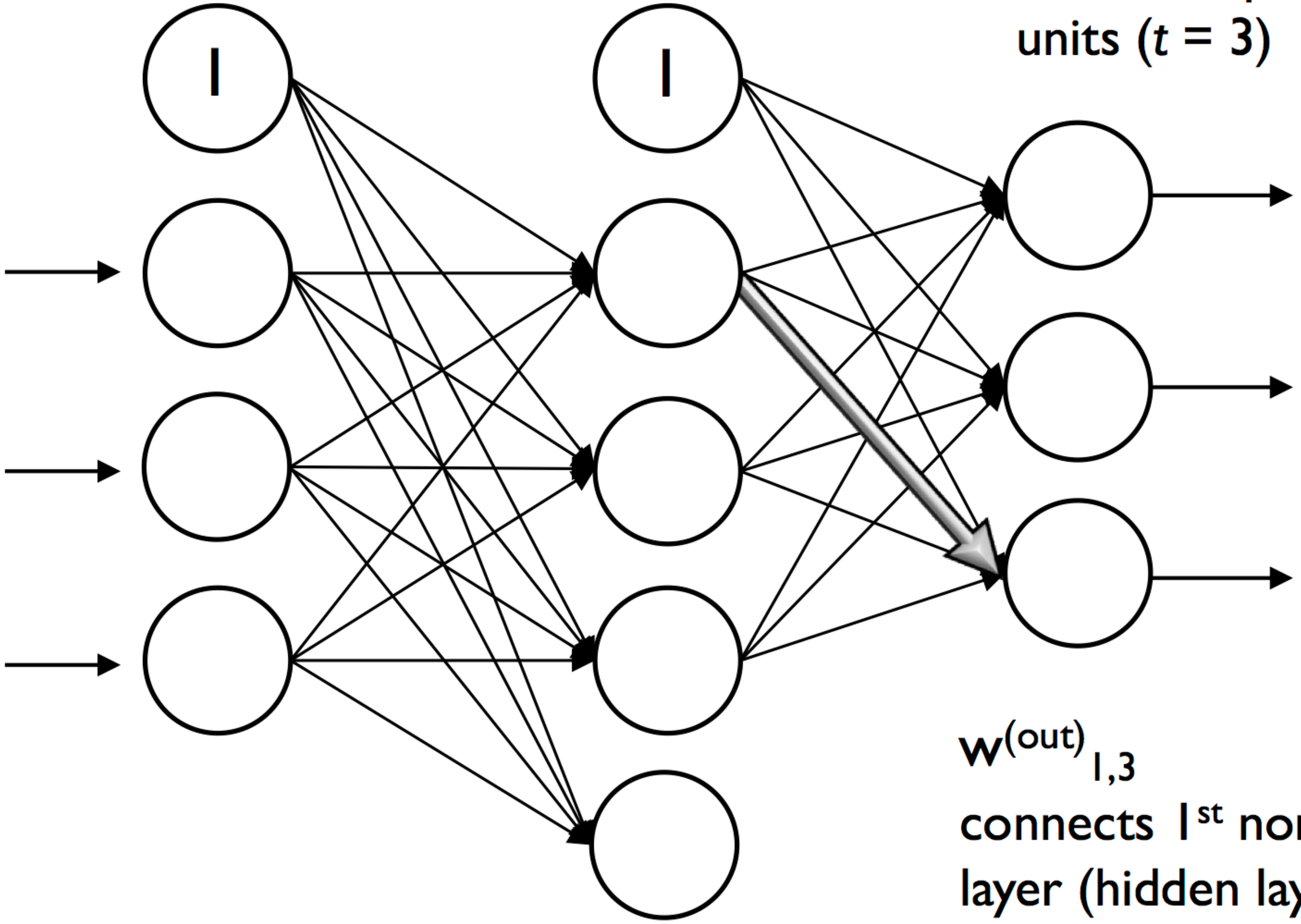
$W^{(h)} \in \mathbb{R}^{m \times d}$

A 3-4-3 MLP

Input layer with 3 input units plus bias unit (m = 3+1)

Hidden layer with 4 hidden units plus bias unit (d = 4+1)

Output layer with 3 output units (t = 3)



$w^{(out)}_{1,3}$

connects 1st non-bias neuron in the 2nd layer (hidden layer h) to the 3rd unit in the 3rd layer (output layer out)

Number of layers: L = 3

MLP Learning Procedure

- Three steps
  - Starting at the input layer, forward propagate the training data $\boldsymbol{x}^{(i)}$ to generate an output
  - Calculate the error we want to minimize with a cost function
  - Backpropagate the error, find its derivative with respective to each weight, and update the model
- After repeating the steps, use ***forward propagation*** to calculate the network output and apply a threshold function to obtain the predict the class label in 1-hot representation

Forward Propagation

- Calculate the net input for unit 1 in the hidden layer

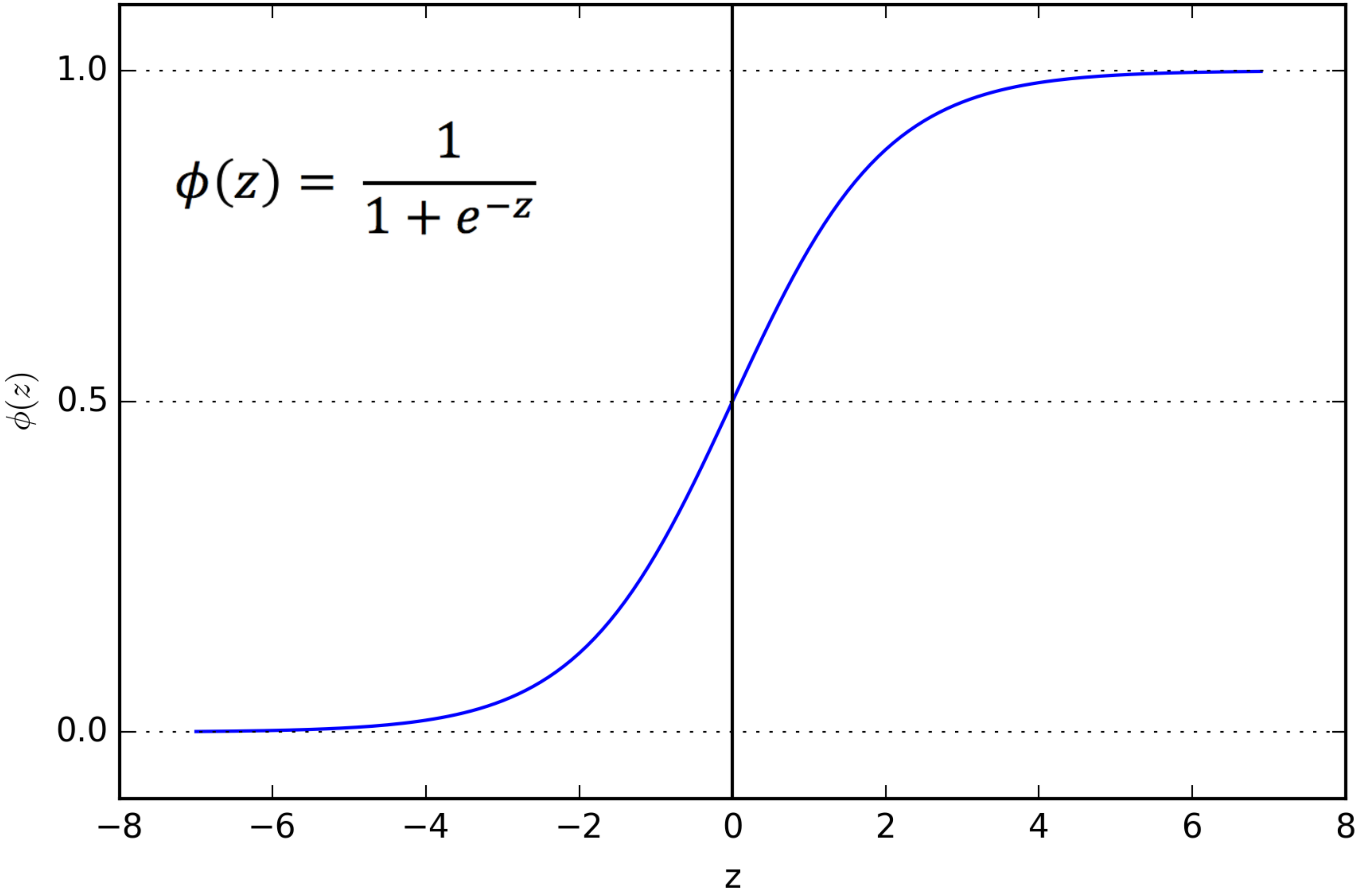
$$z_1^{(h)} = a_0^{(in)} w_{0,1}^{(h)} + a_1^{(in)} w_{1.1}^{(h)} + \cdots + a_m^{(in)} w_{m,1}^{(h)}$$

- Calculate the activation for unit 1 in the hidden layer

$$a_1^{(h)} = \phi\left(z_1^{(h)}\right)$$

  - To be able to solve complex problems such as image classification, we need non-linear activation functions
  - sigmoid (logistic) activation function $\phi(z) = \frac{1}{1+e^{-z}}$

Sigmoid Function



Forward Propagation

- MLP is a typical feedforward ANN
  - Feedforward: each layer serves as the input to the next layer without loops, in contrast to recurrent neural networks
- Neurons are typically sigmoid units, not perceptrons
  - Intuitively consider neurons in MLP as logistic regression units

Vectorized Notation (1/2)

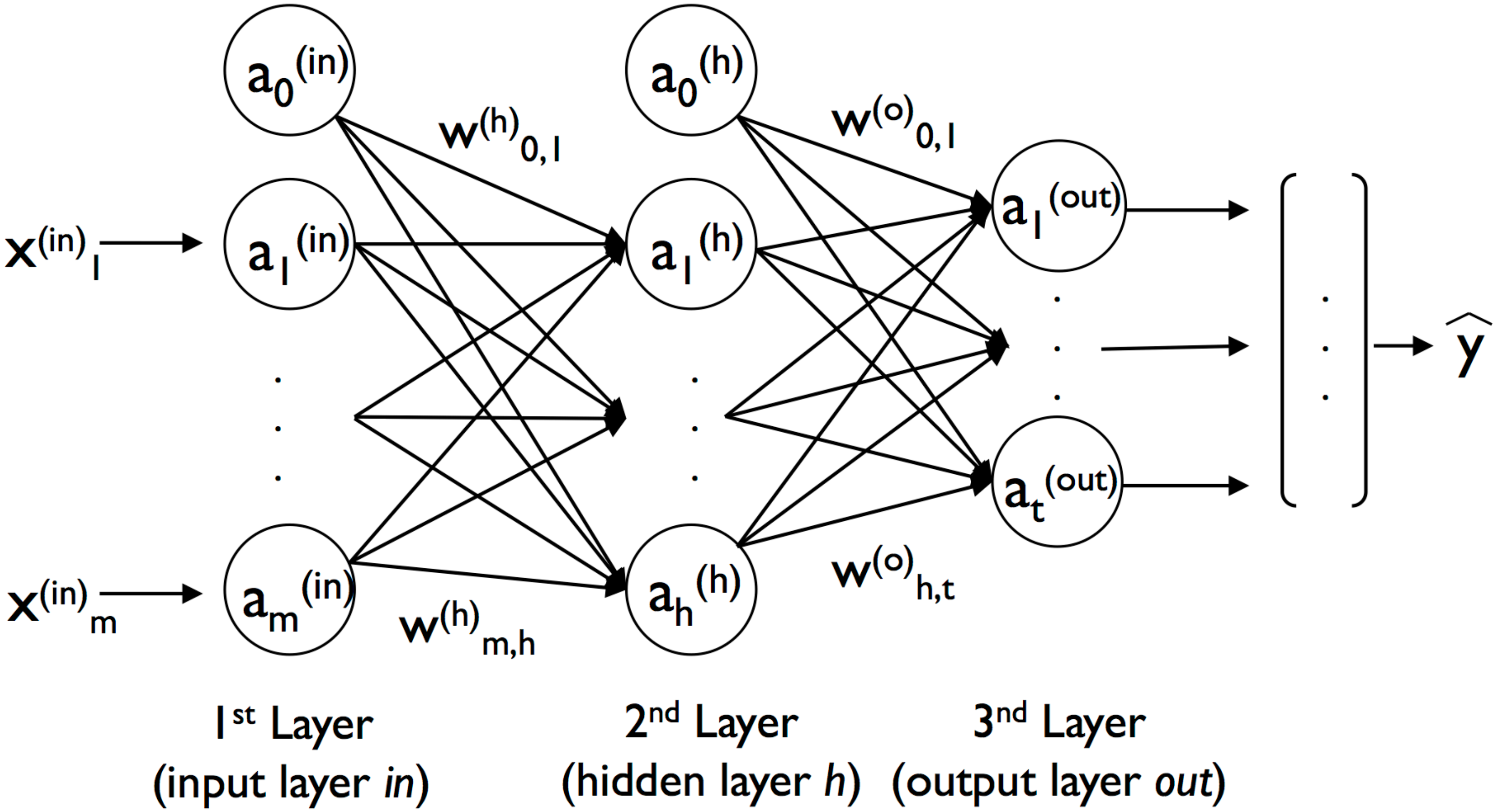
- Write activation in a matrix form
- Readability + more efficient code
- Net inputs for the hidden layer

$$\boldsymbol{z}^{(h)} = \boldsymbol{a}^{(in)} \boldsymbol{W}^{(h)}$$

- Dimensions (ignore bias unit for simplicity)
  - [h x 1] = [h x m] [m x 1]
- Activations for the hidden layer

$$\boldsymbol{a}^{(h)} = \phi\left(\boldsymbol{z}^{(h)}\right)$$

Vectorized Notation (2/2)



$$\boldsymbol{Z}^{(h)} = \boldsymbol{A}^{(in)}\boldsymbol{W}^{(h)} \qquad \boldsymbol{Z}^{(out)} = \boldsymbol{A}^{(h)}\boldsymbol{W}^{(out)}$$

$$A^{(h)} = \phi\left(Z^{(h)}\right) \qquad \boldsymbol{A}^{(out)} = \phi\left(\boldsymbol{Z}^{(out)}\right), \qquad \boldsymbol{A}^{(out)} \in \mathbb{R}^{n\times t}$$

Training an Artificial Neural Network

Cost Function

- The logistic cost function is the same we used for logistic regression

$$J(\boldsymbol{w}) = -\sum_{i=1}^{n} y^{[i]} log\left(a^{[i]}\right) + \left(1 - y^{[i]}\right) log\left(1 - a^{[i]}\right)$$

- $a^{[i]}$ is the sigmoid activation of the *i*th sample in the dataset $a^{[i]} = \phi\left(z^{[i]}\right)$
- Regularization: use L2 $L2 = \lambda \|\boldsymbol{w}\|_2^2 = \lambda \sum_{j=1}^{m} w_j^2$
- Final cost function

$$J(\boldsymbol{w}) = -\left[\sum_{i=1}^{n} y^{[i]} log\left(a^{[i]}\right) + \left(1 - y^{[i]}\right) log\left(1 - a^{[i]}\right)\right] + \frac{\lambda}{2}\|\boldsymbol{w}\|_2^2$$

Cost Function for All Units in Output Layer

- The activation of the 3rd layer and the target class (class 2) for a particular sample may look like

$$a^{(out)} = \begin{bmatrix} 0.1 \\ 0.9 \\ \vdots \\ 0.3 \end{bmatrix}, \quad y = \begin{bmatrix} 0 \\ 1 \\ \vdots \\ 0 \end{bmatrix}$$

- Need to generalize the logistic cost function to all ***t*** activation units (without regularization)

$$J(\boldsymbol{W}) = -\sum_{i=1}^{n}\sum_{j=1}^{t} y_j^{[i]} log\left(a^{[i]}{}_j\right) + \left(1 - y_j^{[i]}\right) log\left(1 - a^{[i]}{}_j\right)$$

Cost Function for the Entire Network

- Sum all the weights in the entire network in the regularization term

L2 penalty term

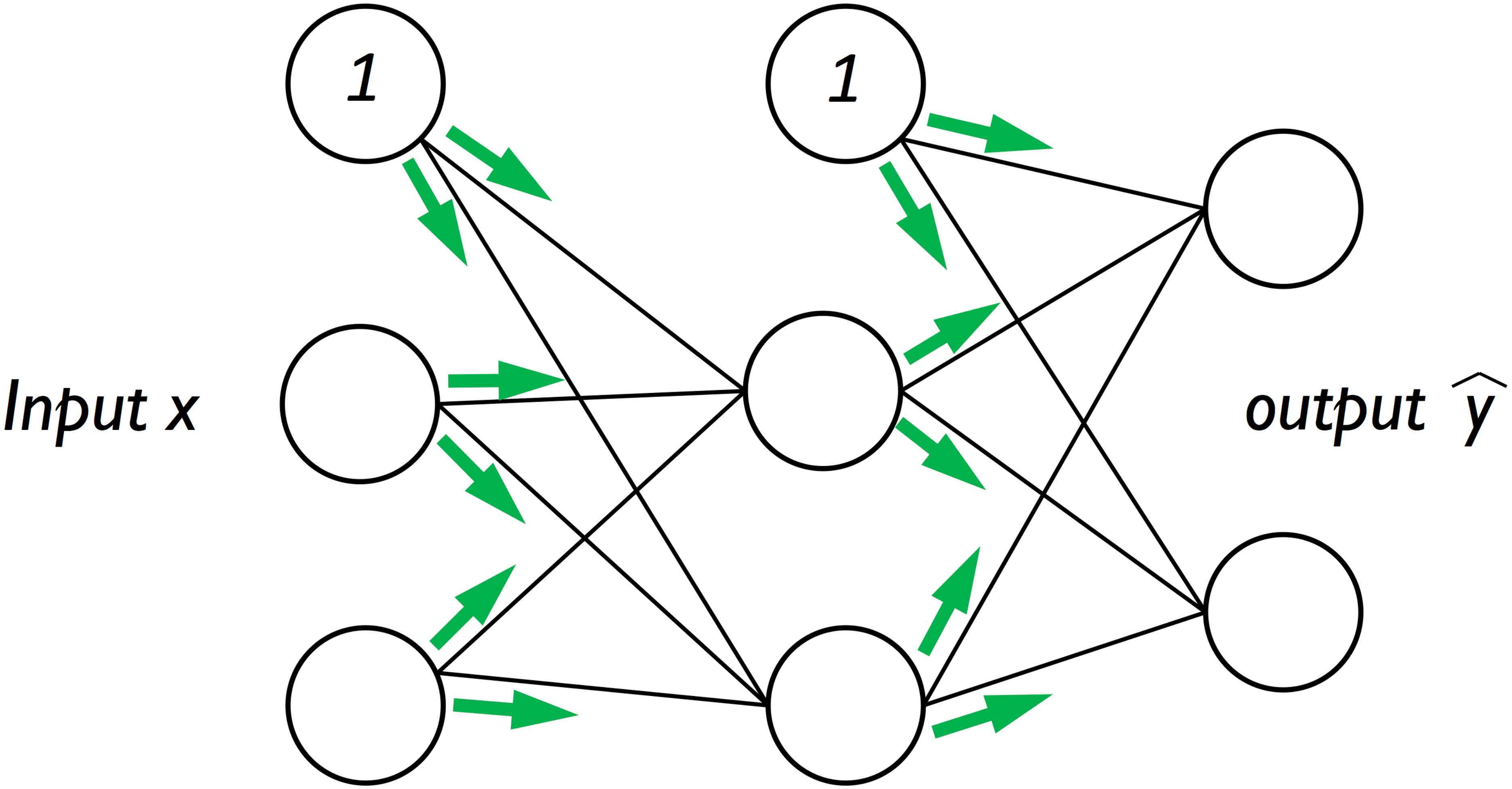
$$J(\boldsymbol{W}) = -\left[\sum_{i=1}^{n}\sum_{j=1}^{t} y_j^{[i]} log\left(a^{[i]}{}_j\right) + \left(1 - y_j^{[i]}\right) log\left(1 - a^{[i]}{}_j\right)\right] + \boxed{\frac{\lambda}{2}\sum_{l=1}^{L-1}\sum_{i=1}^{u_l}\sum_{j=1}^{u_{l+1}}\left(w_{j,i}^{(l)}\right)^2}$$

  - u_l refers to the number of units in a given layer l

- To minimize the cost function $\frac{\partial}{\partial w_{j,i}^{(l)}} J(\boldsymbol{W})$
  - Use back propagation algorithm

Backpropagation

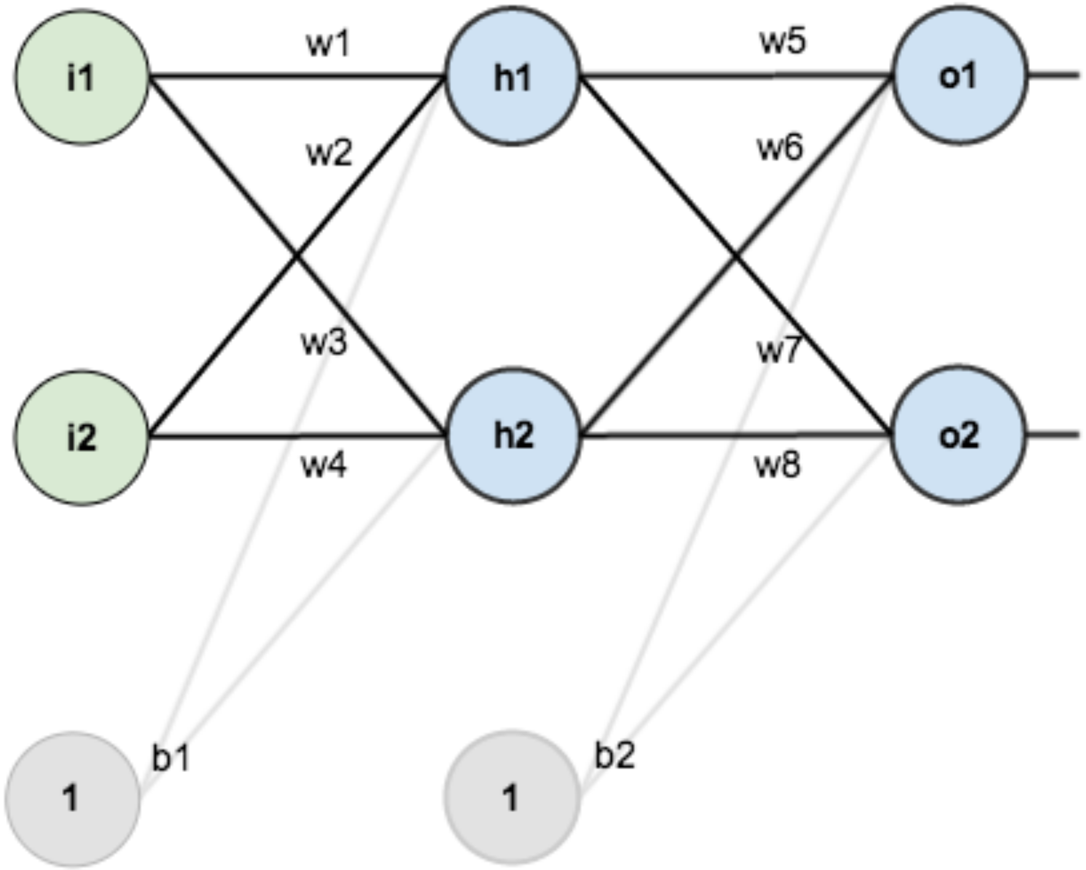
- Consider the forward propagation of the inputs



$$\boldsymbol{Z}^{(h)} = \boldsymbol{A}^{(in)}\boldsymbol{W}^{(h)} \qquad \boldsymbol{Z}^{(out)} = \boldsymbol{A}^{(h)}\boldsymbol{W}^{(out)}$$

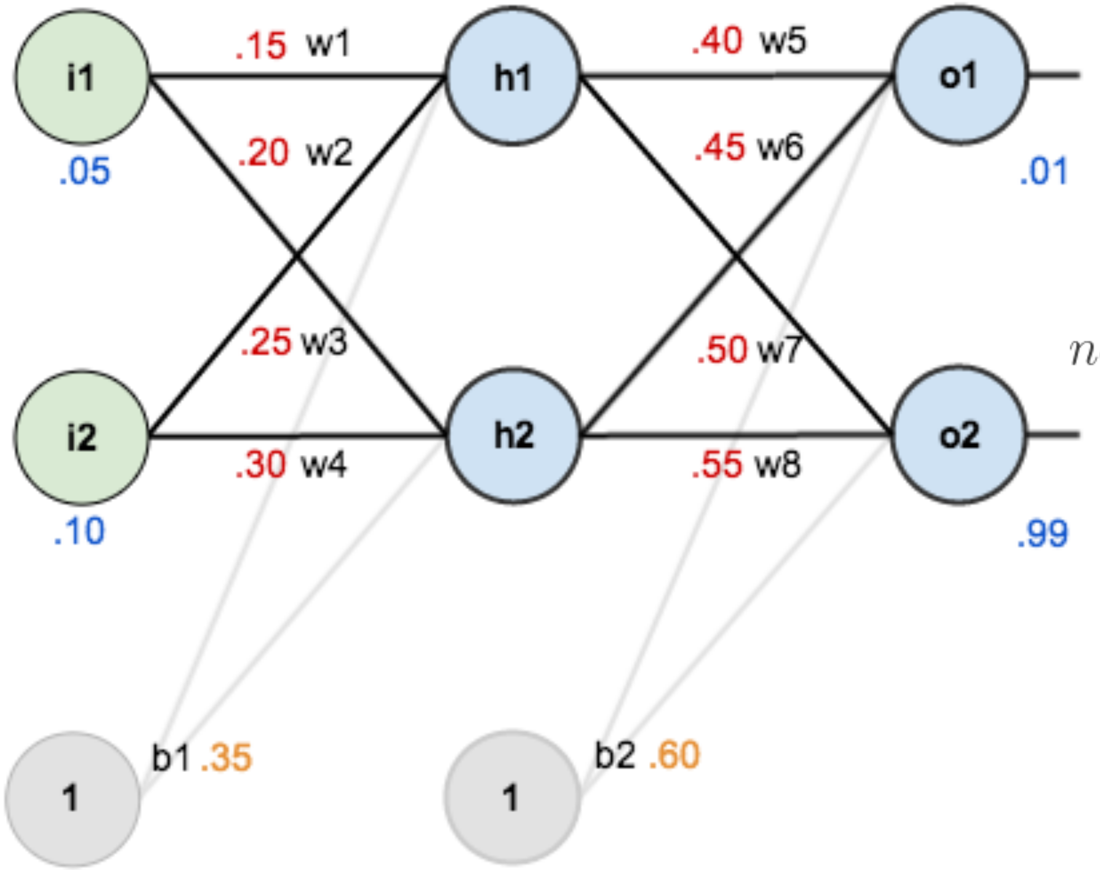
$$A^{(h)} = \phi\left(Z^{(h)}\right) \qquad \boldsymbol{A}^{(out)} = \phi\left(\boldsymbol{Z}^{(out)}\right), \qquad \boldsymbol{A}^{(out)} \in \mathbb{R}^{n\times t}$$

Example



[https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/]

Feedforward



$$net_{h1} = 0.15 * 0.05 + 0.2 * 0.1 + 0.35 * 1 = 0.3775$$

$$out_{h1} = \frac{1}{1+e^{-net_{h1}}} = \frac{1}{1+e^{-0.3775}} = 0.593269992$$

$$out_{h2} = 0.596884378$$

$$net_{o1} = 0.4 * 0.593269992 + 0.45 * 0.596884378 + 0.6 * 1 = 1.105905967$$

$$out_{o1} = \frac{1}{1+e^{-net_{o1}}} = \frac{1}{1+e^{-1.105905967}} = 0.75136507$$

$$out_{o2} = 0.772928465$$

$$E_{total} = \sum \tfrac{1}{2}(target - output)^2$$

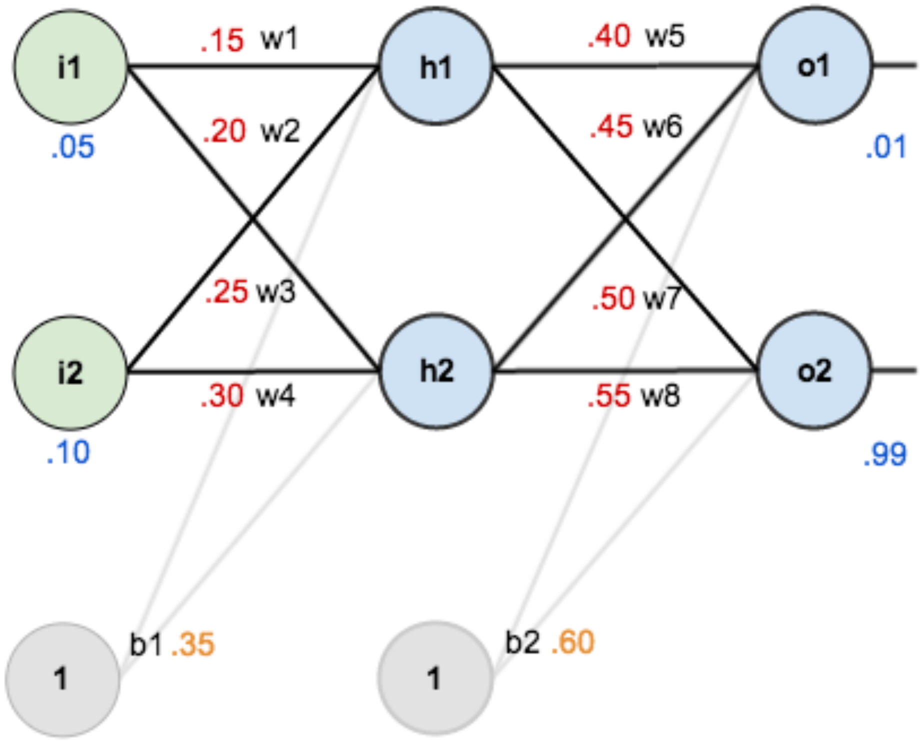
$$E_{o1} = \tfrac{1}{2}(target_{o1} - out_{o1})^2 = \tfrac{1}{2}(0.01 - 0.75136507)^2 = 0.274811083$$

$$E_{o2} = 0.023560026$$

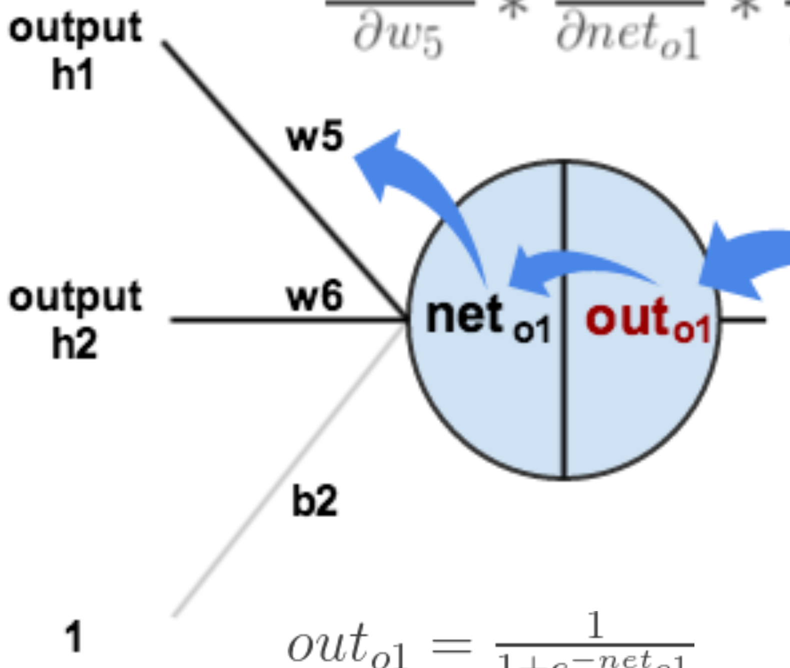
$$E_{total} = E_{o1} + E_{o2} = 0.274811083 + 0.023560026 = 0.298371109$$

Backpropagation (1/2)

- To update w_5 $\frac{\partial E_{total}}{\partial w_5} = \frac{\partial E_{total}}{\partial out_{o1}} * \frac{\partial out_{o1}}{\partial net_{o1}} * \frac{\partial net_{o1}}{\partial w_5}$



$$\frac{\partial net_{o1}}{\partial w_5} * \frac{\partial out_{o1}}{\partial net_{o1}} * \frac{\partial E_{total}}{\partial out_{o1}} = \frac{\partial E_{total}}{\partial w_5}$$



E $_{o1}$ **= ½(target** $_{o1}$ **- out** $_{o1}$ **)²**

E $_{total}$ **= E** $_{o1}$ **+ E** $_{o2}$

$$out_{o1} = \frac{1}{1+e^{-net_{o1}}}$$

$$E_{total} = \frac{1}{2}(target_{o1} - out_{o1})^2 + \frac{1}{2}(target_{o2} - out_{o2})^2$$

$$\frac{\partial out_{o1}}{\partial net_{o1}} = out_{o1}(1 - out_{o1})$$

$$\frac{\partial E_{total}}{\partial out_{o1}} = -(target_{o1} - out_{o1})$$

$$net_{o1} = w_5 * out_{h1} + w_6 * out_{h2} + b_2 * 1$$

$$\frac{\partial net_{o1}}{\partial w_5} = 1 * out_{h1} * w_5^{(1-1)} + 0 + 0 = out_{h1}$$

node delta

$$\frac{\partial E_{total}}{\partial w_5} = \boxed{-(target_{o1} - out_{o1}) * out_{o1}(1 - out_{o1})} * out_{h1}$$

$$w_5^{+} = w_5 - \eta * \frac{\partial E_{total}}{\partial w_5} = 0.4 - 0.5 * 0.082167041 = 0.35891648$$

Backpropagation (2/2)

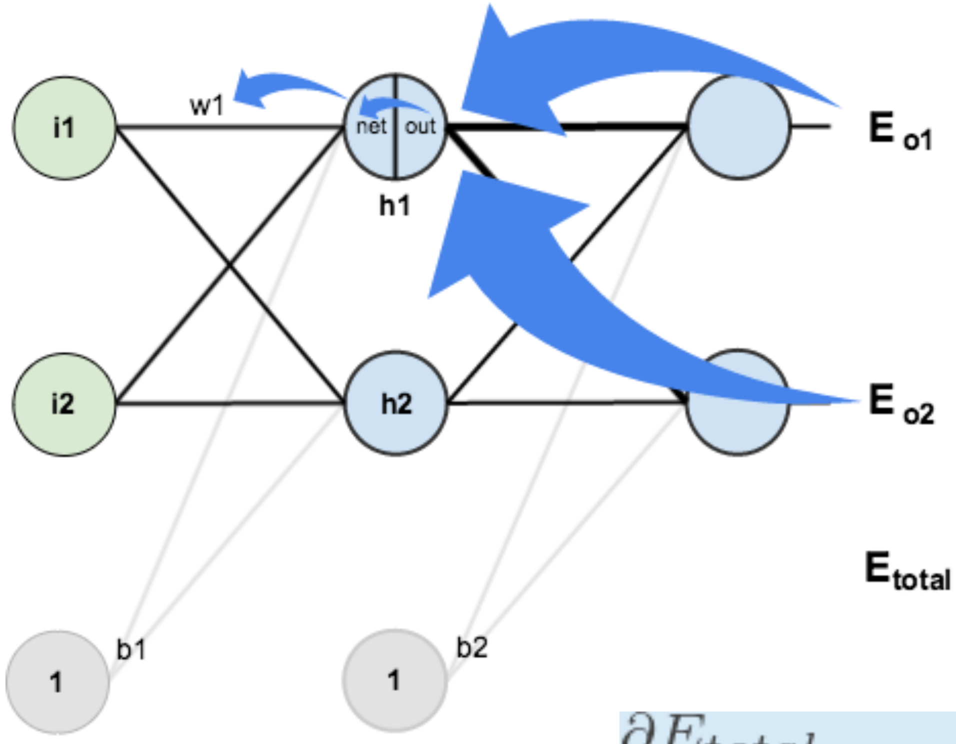
- To update w_1

$$\frac{\partial E_{total}}{\partial w_1} = \frac{\partial E_{total}}{\partial out_{h1}} * \frac{\partial out_{h1}}{\partial net_{h1}} * \frac{\partial net_{h1}}{\partial w_1}$$

$$\frac{\partial E_{total}}{\partial w_1} = \frac{\partial E_{total}}{\partial out_{h1}} * \frac{\partial out_{h1}}{\partial net_{h1}} * \frac{\partial net_{h1}}{\partial w_1}$$

$$\downarrow$$

$$\frac{\partial E_{total}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial out_{h1}} + \frac{\partial E_{o2}}{\partial out_{h1}}$$



$E_{total} = E_{o1} + E_{o2}$

$$\frac{\partial E_{total}}{\partial w_1} = \left(\sum_o \frac{\partial E_{total}}{\partial out_o} * \frac{\partial out_o}{\partial net_o} * \frac{\partial net_o}{\partial out_{h1}}\right) * \frac{\partial out_{h1}}{\partial net_{h1}} * \frac{\partial net_{h1}}{\partial w_1}$$

$$\frac{\partial E_{total}}{\partial w_1} = \boxed{\left(\sum_o \delta_o * w_{ho}\right) * out_{h1}(1 - out_{h1})} * i_1$$

Textbook Version (1/2)

- Error vector of the output layer

$$\boldsymbol{\delta}^{(out)} = \boldsymbol{a}^{(out)} - \boldsymbol{y}$$

- Error term of the hidden layer

$$\boldsymbol{\delta}^{(h)} = \boldsymbol{\delta}^{(out)} \left(\boldsymbol{W}^{(out)}\right)^T \odot \frac{\partial \phi\left(z^{(h)}\right)}{\partial z^{(h)}}$$

$$\delta^{(h)} = \delta^{(out)} \left(\boldsymbol{W}^{(out)}\right)^T \odot \left(a^{(h)} \odot \left(1 - a^{(h)}\right)\right)$$

- Derivation of the cost function

$$\frac{\partial}{\partial w_{i,j}^{(out)}} J\left(\boldsymbol{W}\right) = a_j^{(h)} \delta_i^{(out)}$$

$$\Delta^{(h)} = \Delta^{(h)} + \left(\boldsymbol{A}^{(in)}\right)^T \delta^{(h)}$$

$$\frac{\partial}{\partial w_{i,j}^{(h)}} J\left(\boldsymbol{W}\right) = a_j^{(in)} \delta_i^{(h)}$$

$$\Delta^{(out)} = \Delta^{(out)} + \left(\boldsymbol{A}^{(h)}\right)^T \delta^{(out)}$$

Textbook Version (2/2)

- Add the regularization term

$$\Delta^{(l)} := \Delta^{(l)} + \lambda^{(l)} \left(except\ for\ the\ bias\ term \right)$$

- Final weight update

$$\boldsymbol{W}^{(l)} := \boldsymbol{W}^{(l)} - \eta \Delta^{(l)}$$

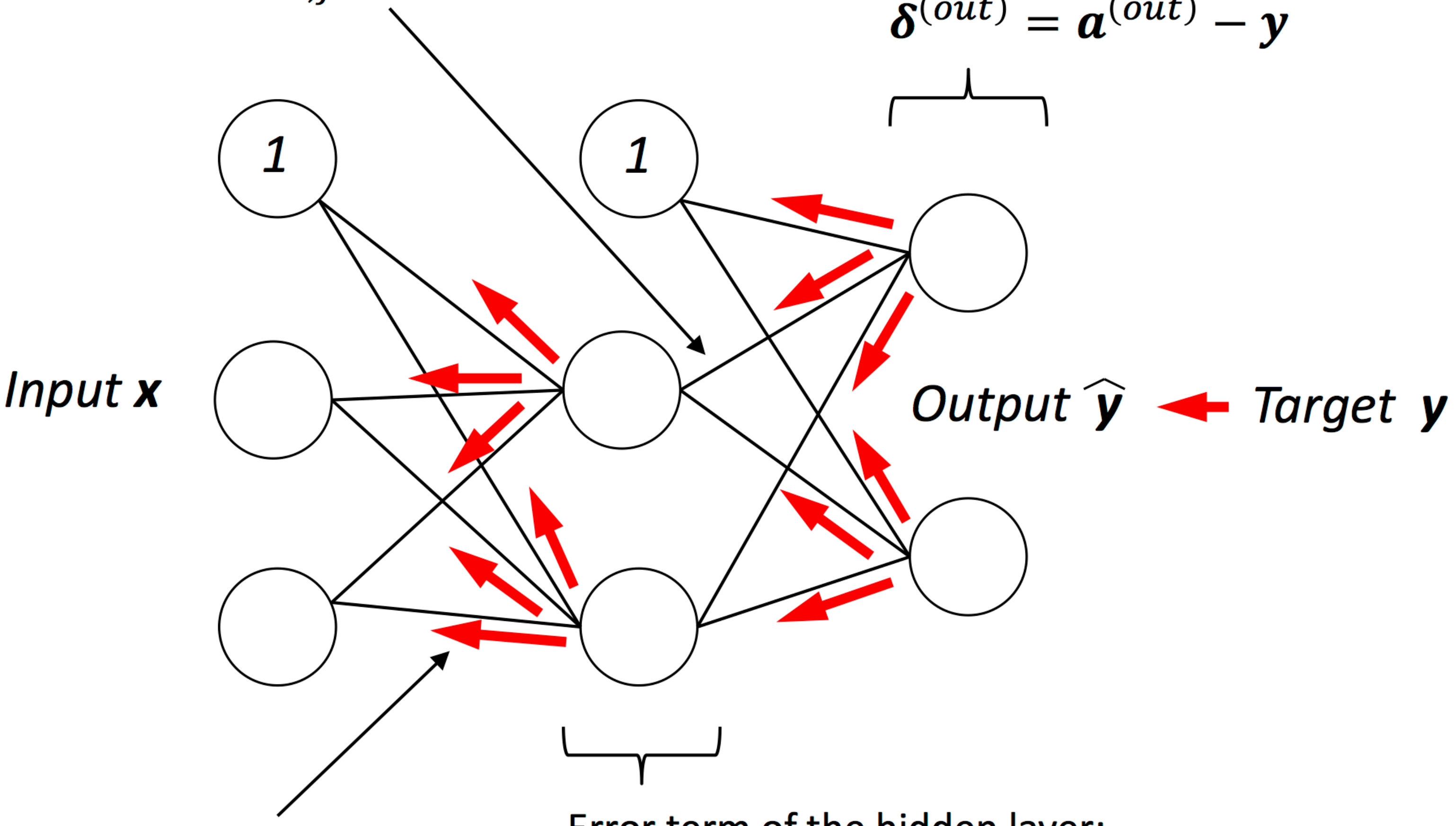
Summary

Compute the gradient:

$$\frac{\partial}{\partial w_{i,j}^{(out)}} J(\boldsymbol{W}) = a_j^{(h)} \delta_i^{(out)}$$

Error term of the output layer:

$$\boldsymbol{\delta}^{(out)} = \boldsymbol{a}^{(out)} - \boldsymbol{y}$$



Compute the gradient:

$$\frac{\partial}{\partial w_{i,j}^{(h)}} J(\boldsymbol{W}) = a_j^{(in)} \delta_i^{(h)}$$

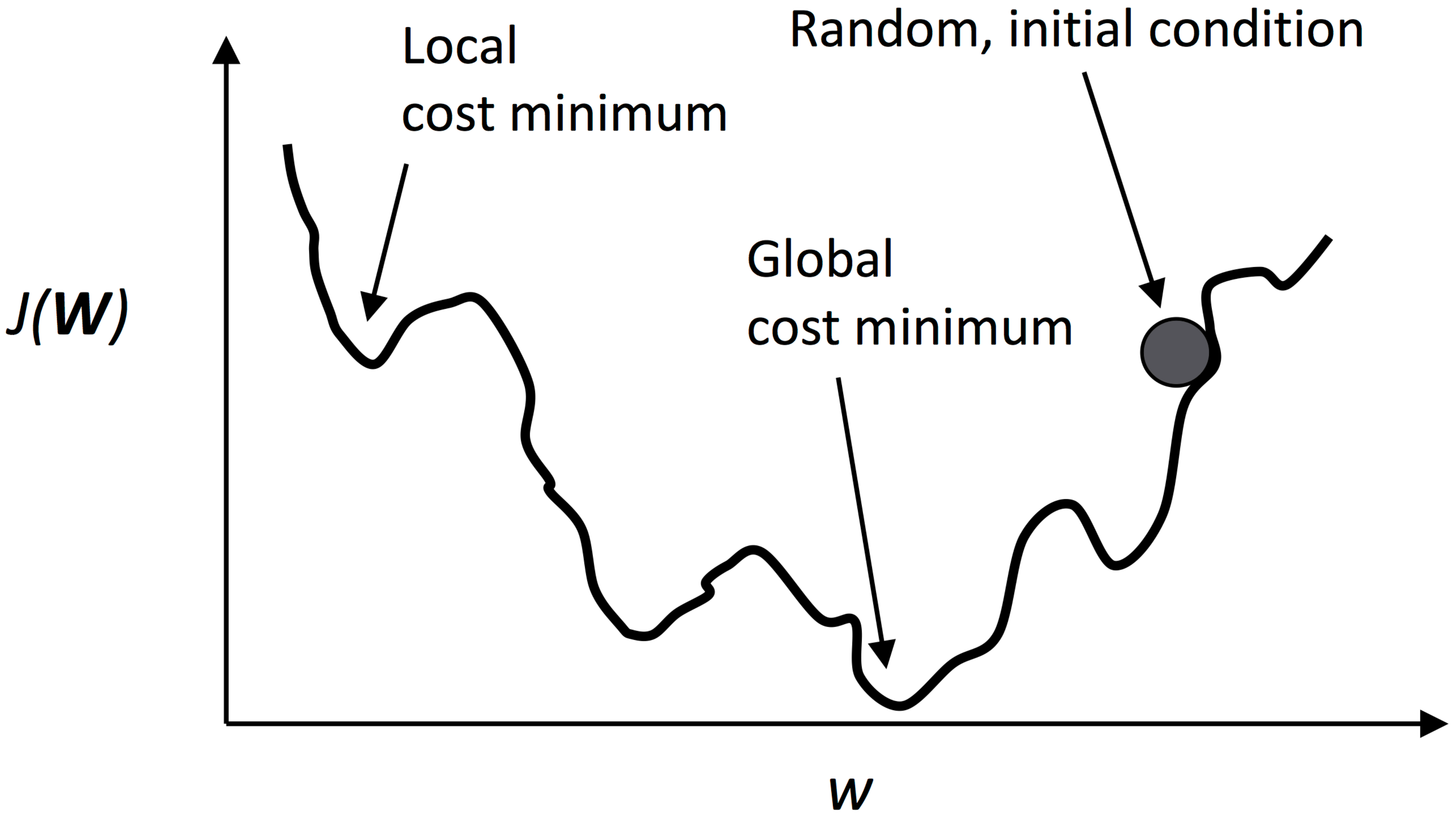
Error term of the hidden layer:

$$\boldsymbol{\delta}^{(h)} = \boldsymbol{\delta}^{(out)} \left(\boldsymbol{W}^{(out)}\right)^T \odot \frac{\partial \phi\left(z^{(h)}\right)}{\partial z^{(h)}}$$

Convergence in Neural Networks

Min-batch Learning

- Speed and convergence



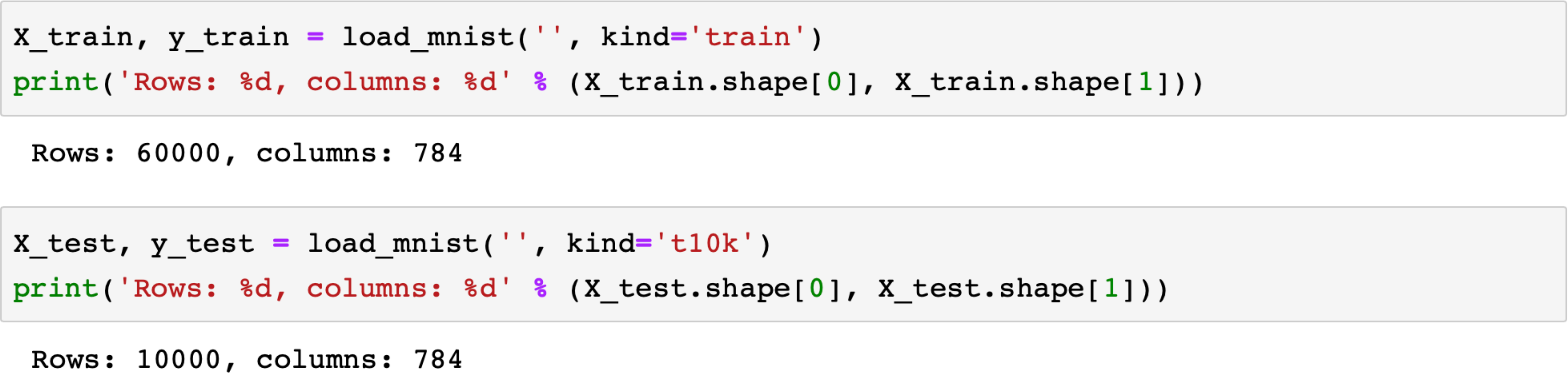
Classifying Handwritten Digits

MNIST Dataset

- Mixed National Institute of Standards and Technology (MNIST) dataset
  - Constructed by Yan LeCun and others as a popular benchmark for ML
  - http://yann.lecun.com/exdb/mnist/
    - Training set images
    - Training set labels
    - Test set images
    - Test set labels
  - was constructed from two datasets of the US NIST
  - handwritten digits from 250 different people, 50% high school students and 50% employees from the Census Bureau

Load Dataset

- **load_mnist** returns two arrays
  - images: n x m array, n: # of samples, m: # of features (pixels)
    - Each pixel is represented by a grey intensity value (0-255), and is normalized to [-1,1]
  - labels: target variable of the class labels (0-9)



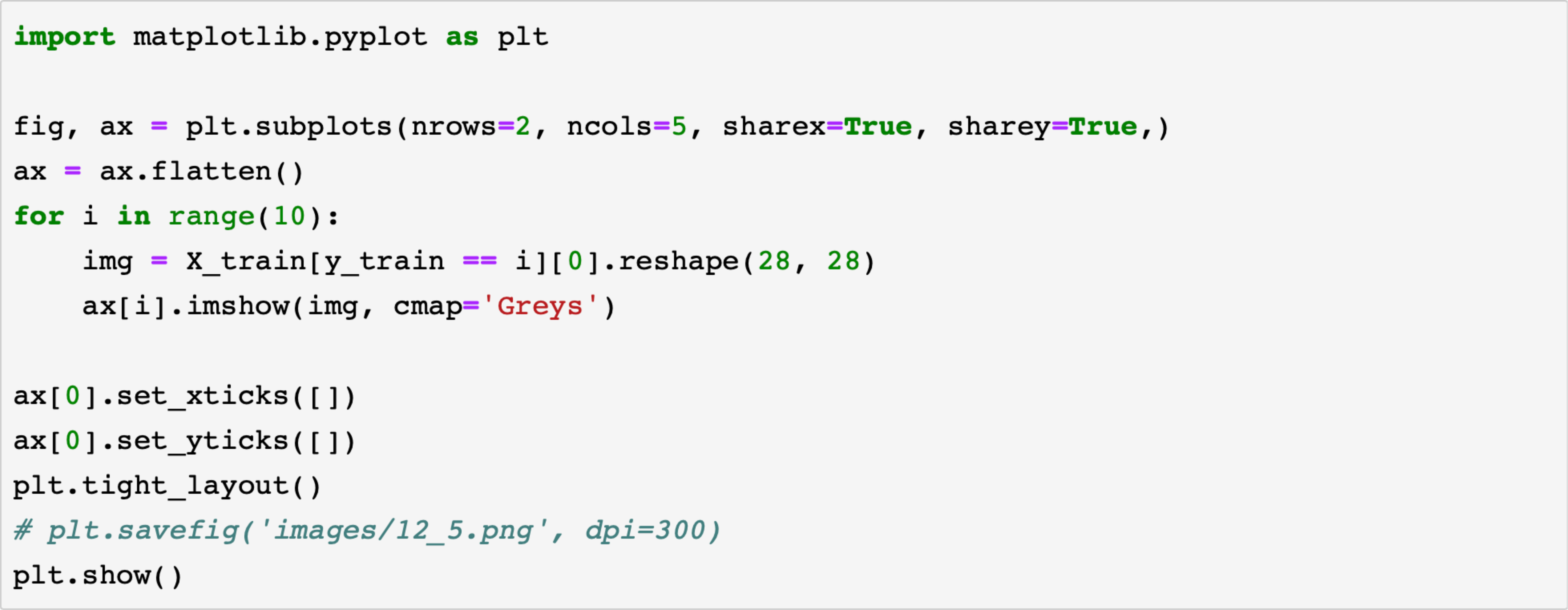
```
X_train, y_train = load_mnist('', kind='train')
print('Rows: %d, columns: %d' % (X_train.shape[0], X_train.shape[1]))
```

```
Rows: 60000, columns: 784
```

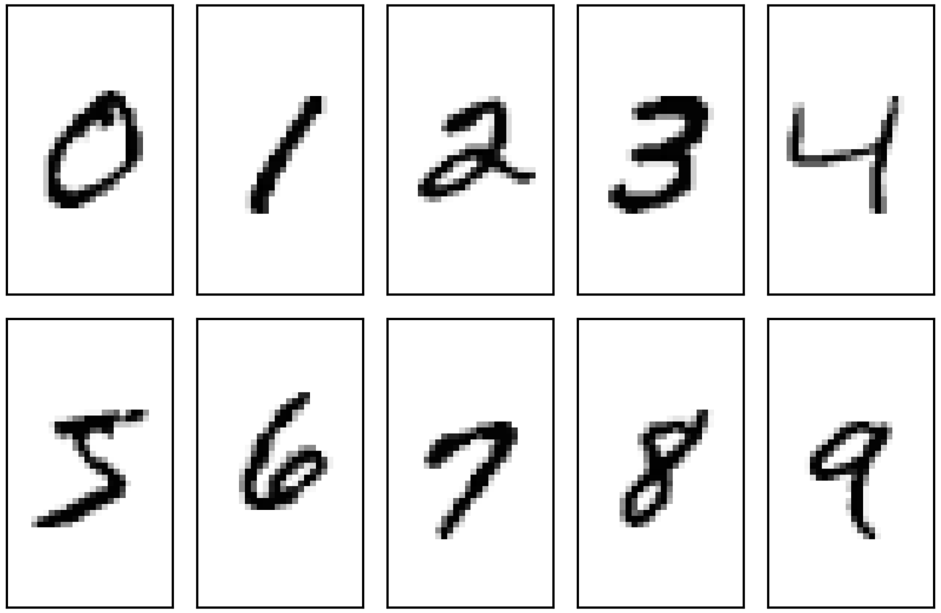
```
X_test, y_test = load_mnist('', kind='t10k')
print('Rows: %d, columns: %d' % (X_test.shape[0], X_test.shape[1]))
```

```
Rows: 10000, columns: 784
```

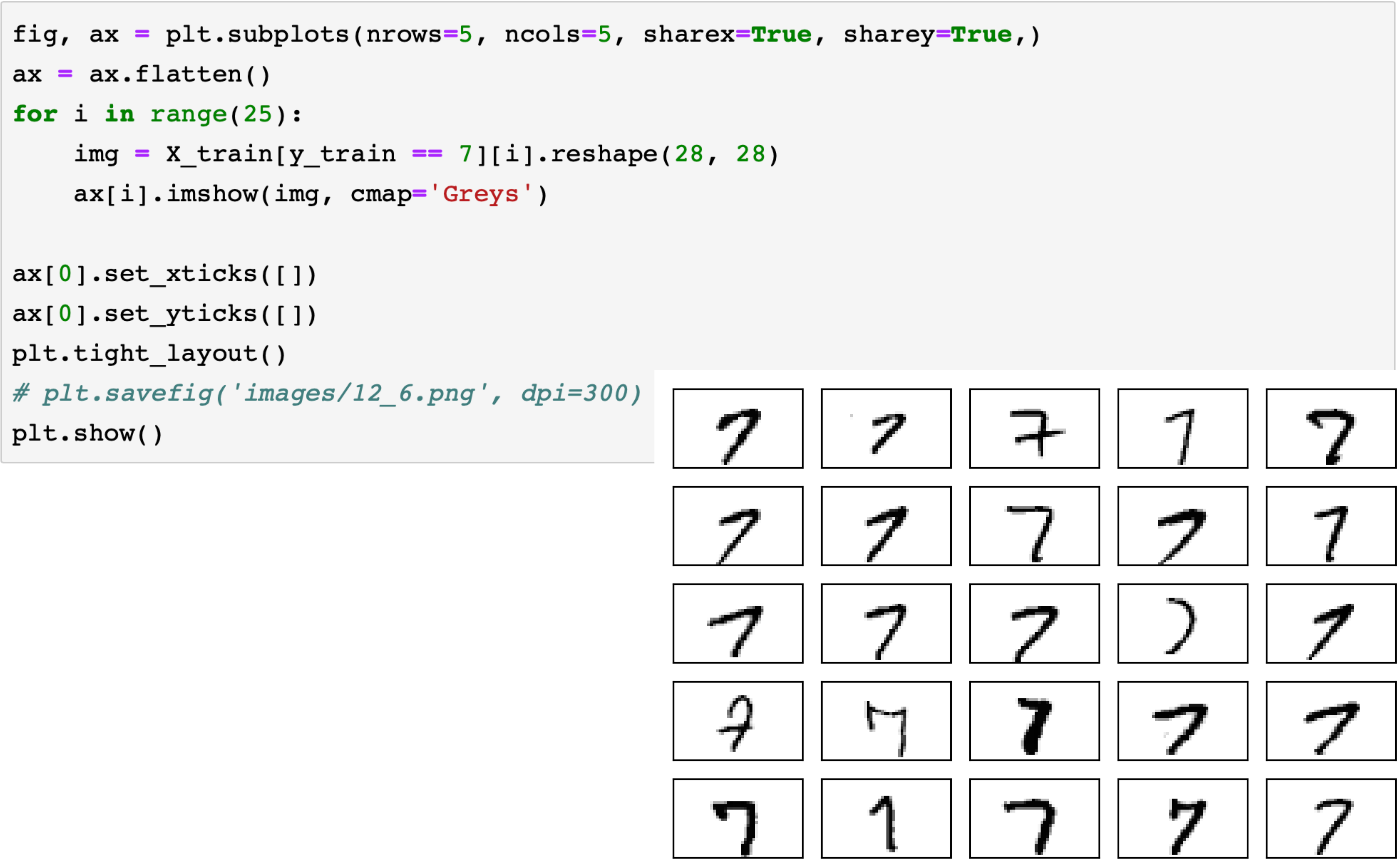
Visualize the First Digit of Each Class



```
import matplotlib.pyplot as plt

fig, ax = plt.subplots(nrows=2, ncols=5, sharex=True, sharey=True,)
ax = ax.flatten()
for i in range(10):
    img = X_train[y_train == i][0].reshape(28, 28)
    ax[i].imshow(img, cmap='Greys')

ax[0].set_xticks([])
ax[0].set_yticks([])
plt.tight_layout()
# plt.savefig('images/12_5.png', dpi=300)
plt.show()
```

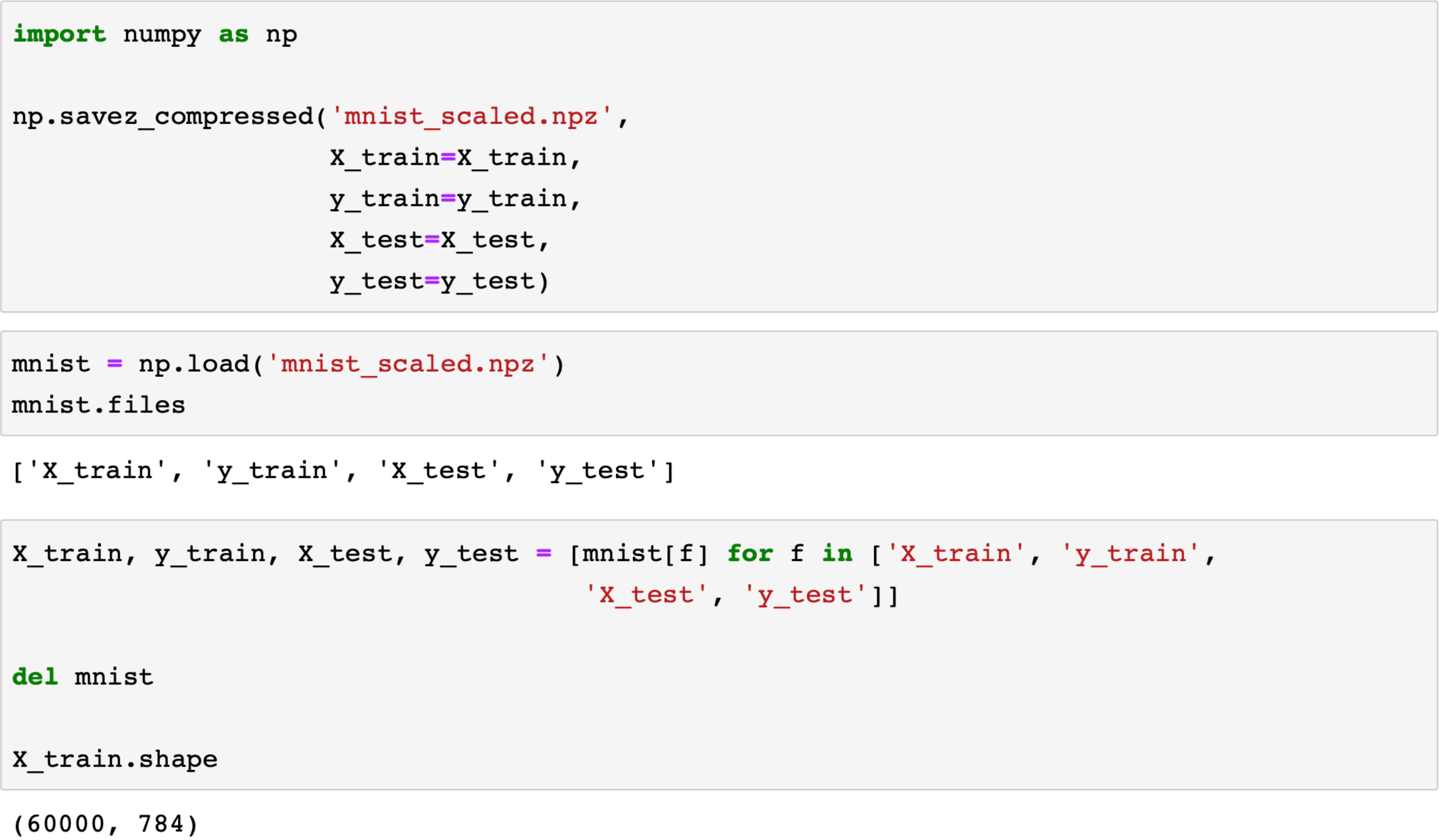


Visualize 25 Different Versions of 7



```
fig, ax = plt.subplots(nrows=5, ncols=5, sharex=True, sharey=True,)
ax = ax.flatten()
for i in range(25):
    img = X_train[y_train == 7][i].reshape(28, 28)
    ax[i].imshow(img, cmap='Greys')

ax[0].set_xticks([])
ax[0].set_yticks([])
plt.tight_layout()
# plt.savefig('images/12_6.png', dpi=300)
plt.show()
```

Save Scaled Images to Avoid Overhead Again



```
import numpy as np

np.savez_compressed('mnist_scaled.npz',
                    X_train=X_train,
                    y_train=y_train,
                    X_test=X_test,
                    y_test=y_test)
```

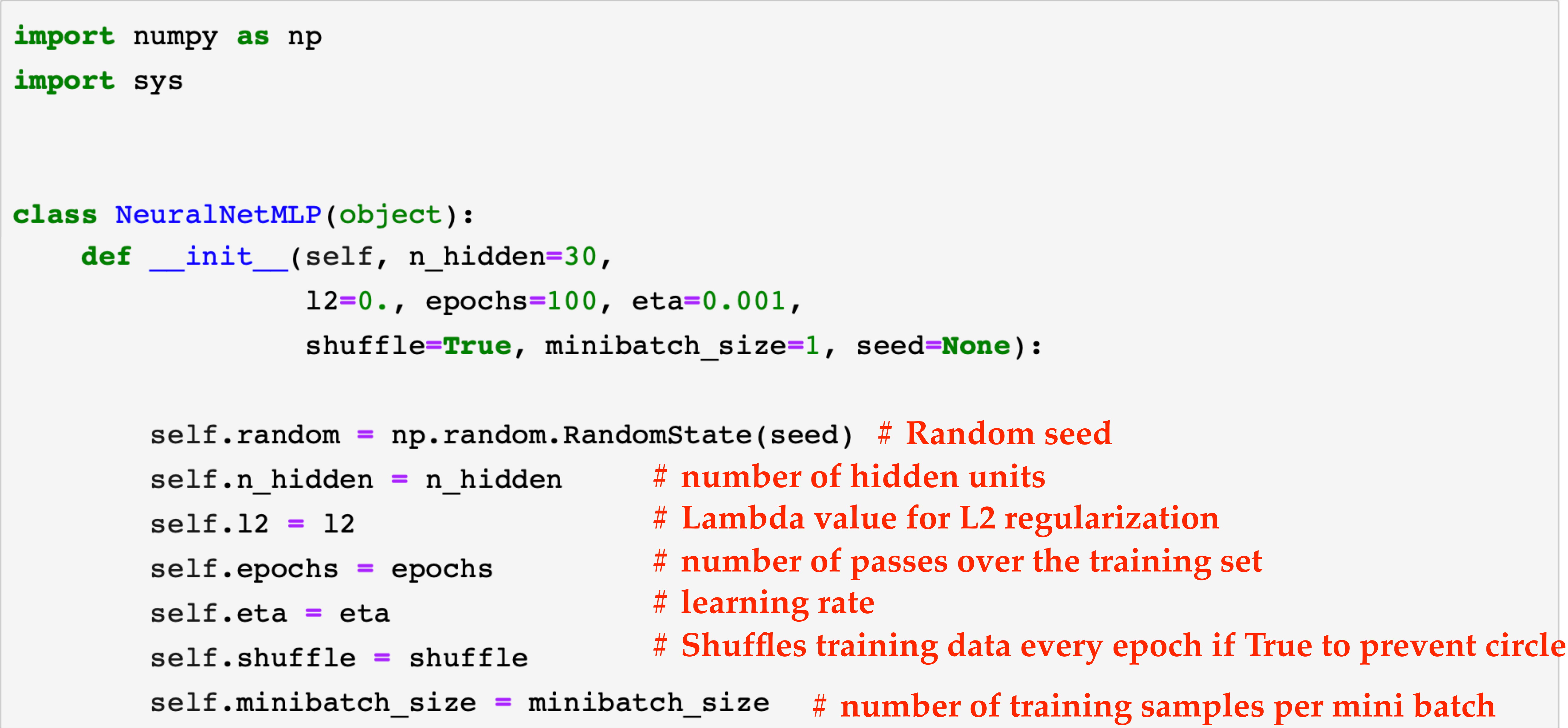
```
mnist = np.load('mnist_scaled.npz')
mnist.files
```

```
['X_train', 'y_train', 'X_test', 'y_test']
```

```
X_train, y_train, X_test, y_test = [mnist[f] for f in ['X_train', 'y_train',
                                    'X_test', 'y_test']]

del mnist

X_train.shape
```

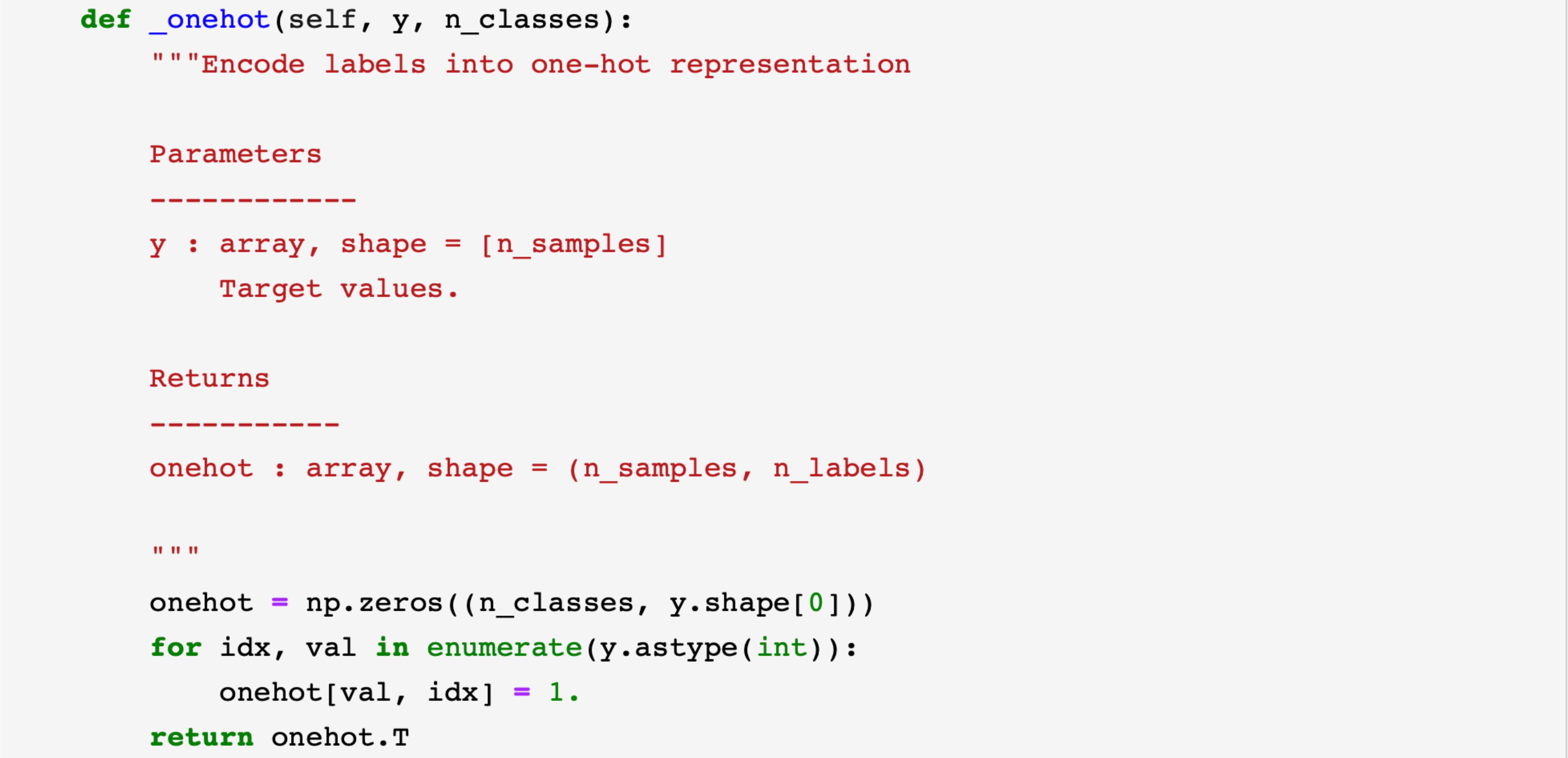
```
(60000, 784)
```

Implementing a Multilayer Perceptron

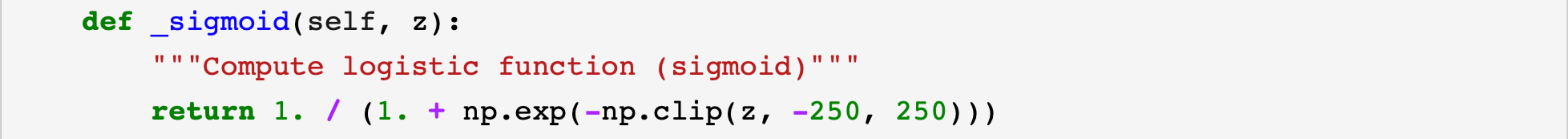


```
import numpy as np
import sys


class NeuralNetMLP(object):
    def __init__(self, n_hidden=30,
                 l2=0., epochs=100, eta=0.001,
                 shuffle=True, minibatch_size=1, seed=None):

        self.random = np.random.RandomState(seed)  # Random seed
        self.n_hidden = n_hidden            # number of hidden units
        self.l2 = l2                        # Lambda value for L2 regularization
        self.epochs = epochs                # number of passes over the training set
        self.eta = eta                      # learning rate
        self.shuffle = shuffle              # Shuffles training data every epoch if True to prevent circles
        self.minibatch_size = minibatch_size   # number of training samples per mini batch
```

Label Onehot Encoding and Sigmoid

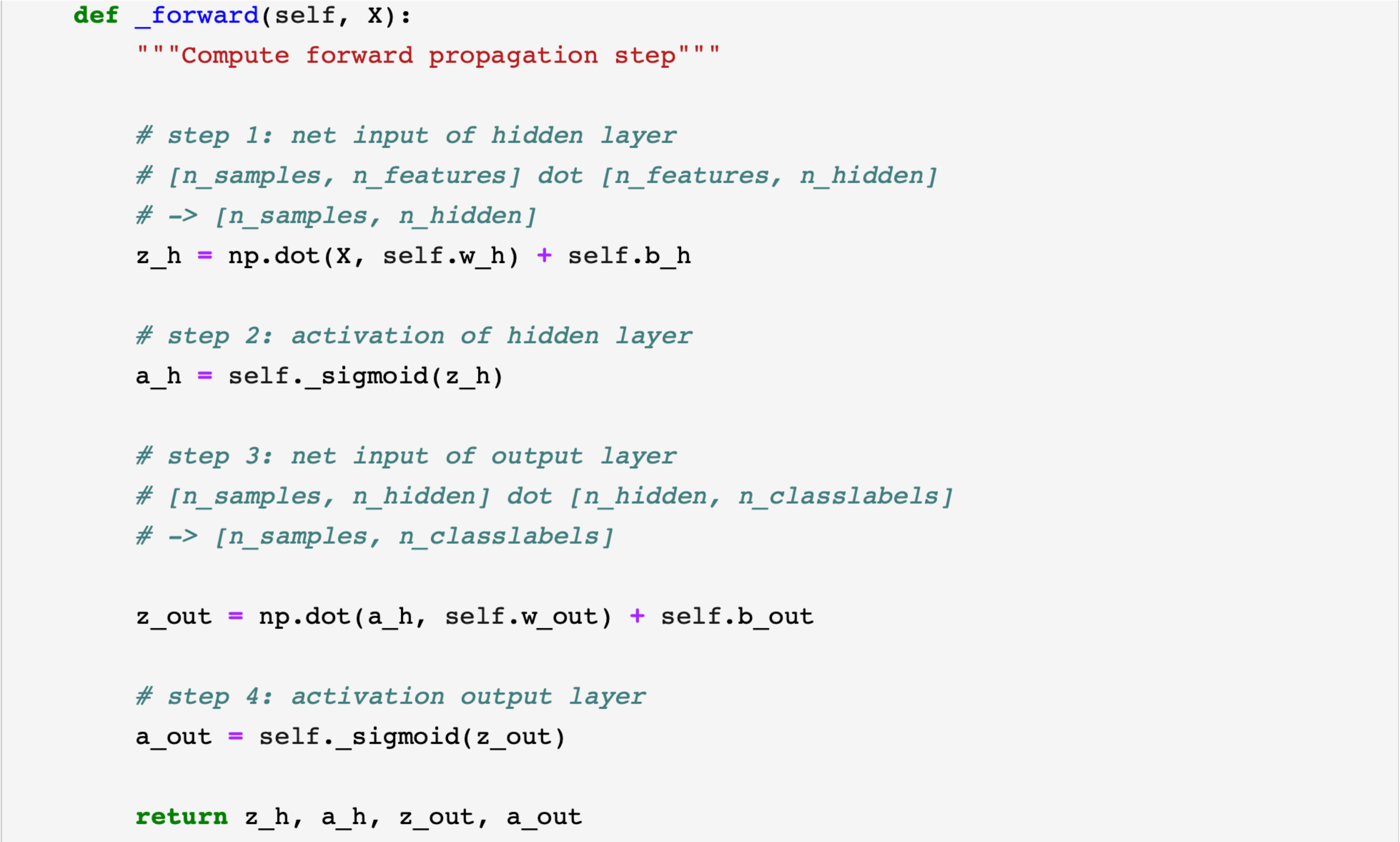


```
def _onehot(self, y, n_classes):
    """Encode labels into one-hot representation

    Parameters
    ------------
    y : array, shape = [n_samples]
        Target values.

    Returns
    -----------
    onehot : array, shape = (n_samples, n_labels)

    """
    onehot = np.zeros((n_classes, y.shape[0]))
    for idx, val in enumerate(y.astype(int)):
        onehot[val, idx] = 1.
    return onehot.T
```



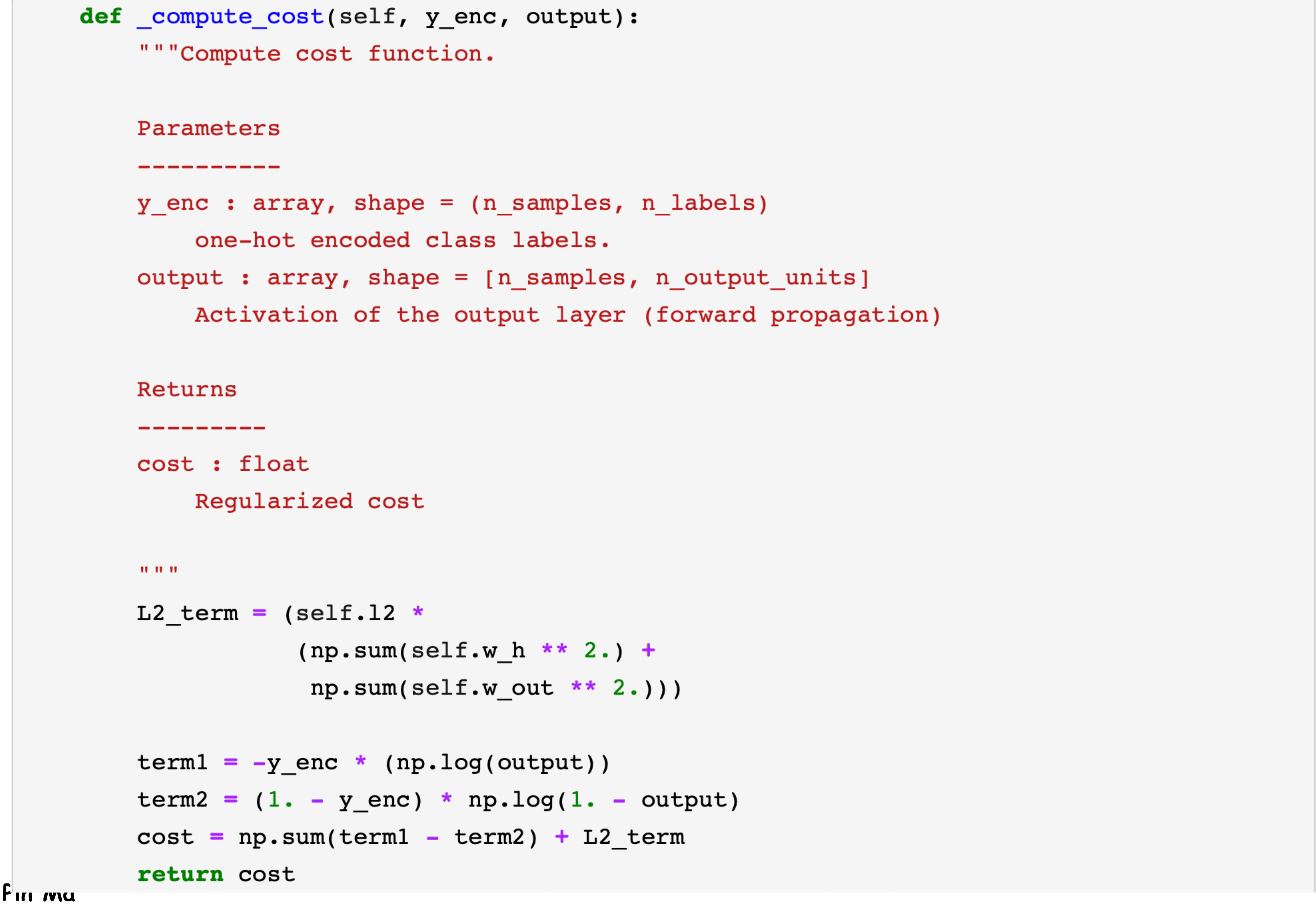
```
def _sigmoid(self, z):
    """Compute logistic function (sigmoid)"""
    return 1. / (1. + np.exp(-np.clip(z, -250, 250)))
```

Forward Calculation



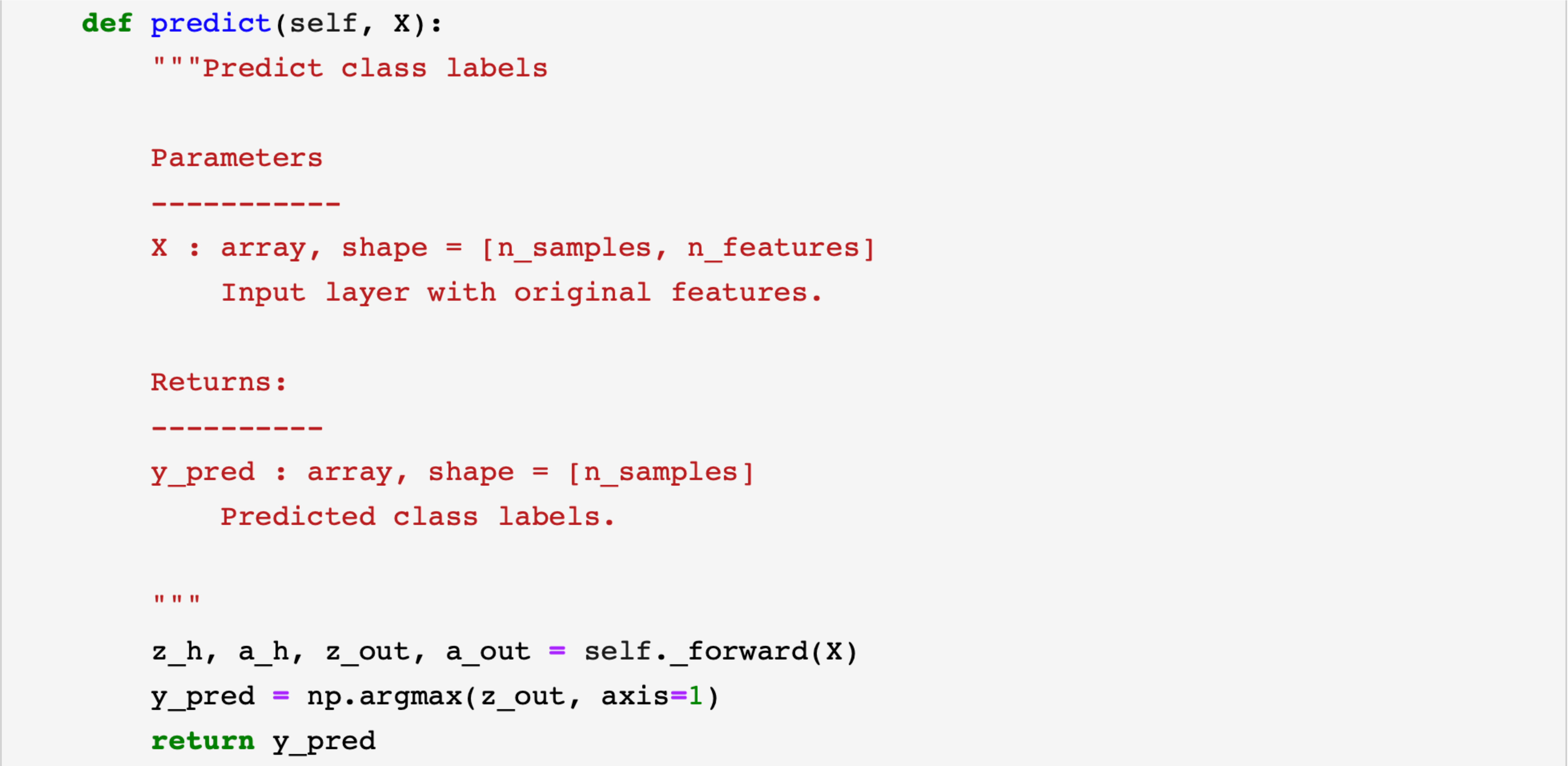
```python
    def _forward(self, X):
        """Compute forward propagation step"""

        # step 1: net input of hidden layer
        # [n_samples, n_features] dot [n_features, n_hidden]
        # -> [n_samples, n_hidden]
        z_h = np.dot(X, self.w_h) + self.b_h

        # step 2: activation of hidden layer
        a_h = self._sigmoid(z_h)

        # step 3: net input of output layer
        # [n_samples, n_hidden] dot [n_hidden, n_classlabels]
        # -> [n_samples, n_classlabels]

        z_out = np.dot(a_h, self.w_out) + self.b_out

        # step 4: activation output layer
        a_out = self._sigmoid(z_out)

        return z_h, a_h, z_out, a_out
```

Cost Calculation



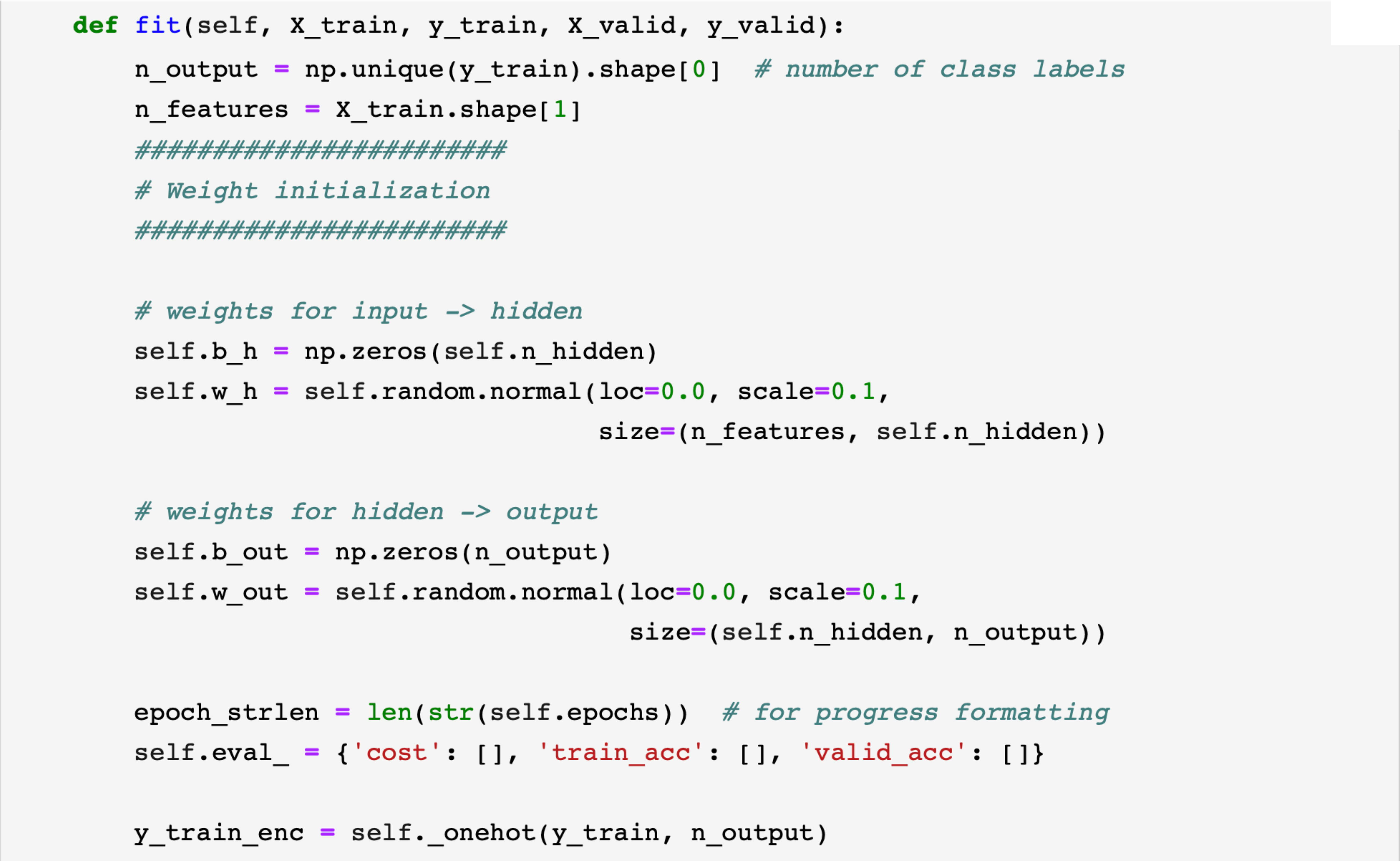
```python
    def _compute_cost(self, y_enc, output):
        """Compute cost function.

        Parameters
        ----------
        y_enc : array, shape = (n_samples, n_labels)
            one-hot encoded class labels.
        output : array, shape = [n_samples, n_output_units]
            Activation of the output layer (forward propagation)

        Returns
        ---------
        cost : float
            Regularized cost

        """
        L2_term = (self.l2 *
                   (np.sum(self.w_h ** 2.) +
                    np.sum(self.w_out ** 2.)))

        term1 = -y_enc * (np.log(output))
        term2 = (1. - y_enc) * np.log(1. - output)
        cost = np.sum(term1 - term2) + L2_term
        return cost
```

Prediction



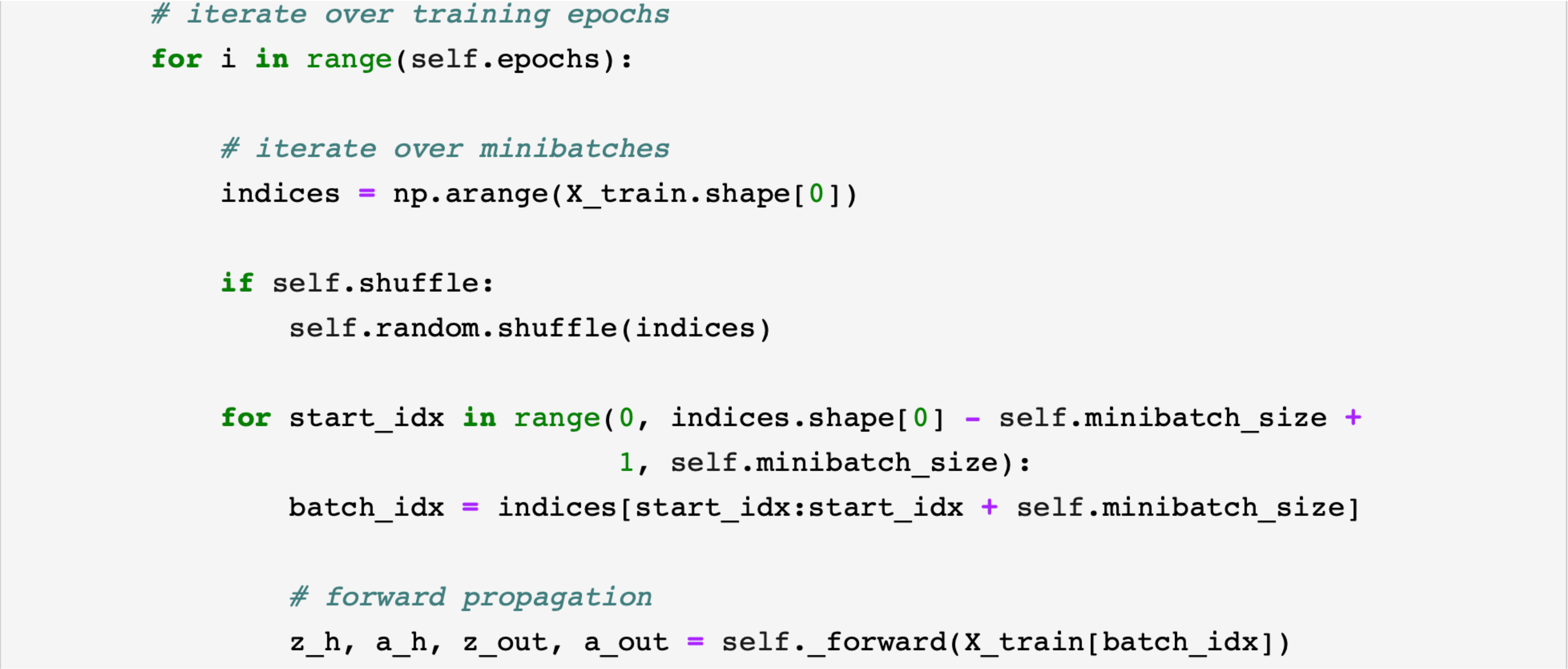
```
    def predict(self, X):
        """Predict class labels

        Parameters
        -----------
        X : array, shape = [n_samples, n_features]
            Input layer with original features.

        Returns:
        ----------
        y_pred : array, shape = [n_samples]
            Predicted class labels.

        """
        z_h, a_h, z_out, a_out = self._forward(X)
        y_pred = np.argmax(z_out, axis=1)
        return y_pred
```

Fit (1/3)



```
    def fit(self, X_train, y_train, X_valid, y_valid):
        n_output = np.unique(y_train).shape[0]  # number of class labels
        n_features = X_train.shape[1]
        ########################
        # Weight initialization
        ########################

        # weights for input -> hidden
        self.b_h = np.zeros(self.n_hidden)
        self.w_h = self.random.normal(loc=0.0, scale=0.1,
                                      size=(n_features, self.n_hidden))

        # weights for hidden -> output
        self.b_out = np.zeros(n_output)
        self.w_out = self.random.normal(loc=0.0, scale=0.1,
                                        size=(self.n_hidden, n_output))

        epoch_strlen = len(str(self.epochs))  # for progress formatting
        self.eval_ = {'cost': [], 'train_acc': [], 'valid_acc': []}

        y_train_enc = self._onehot(y_train, n_output)
```

Fit (2/3)

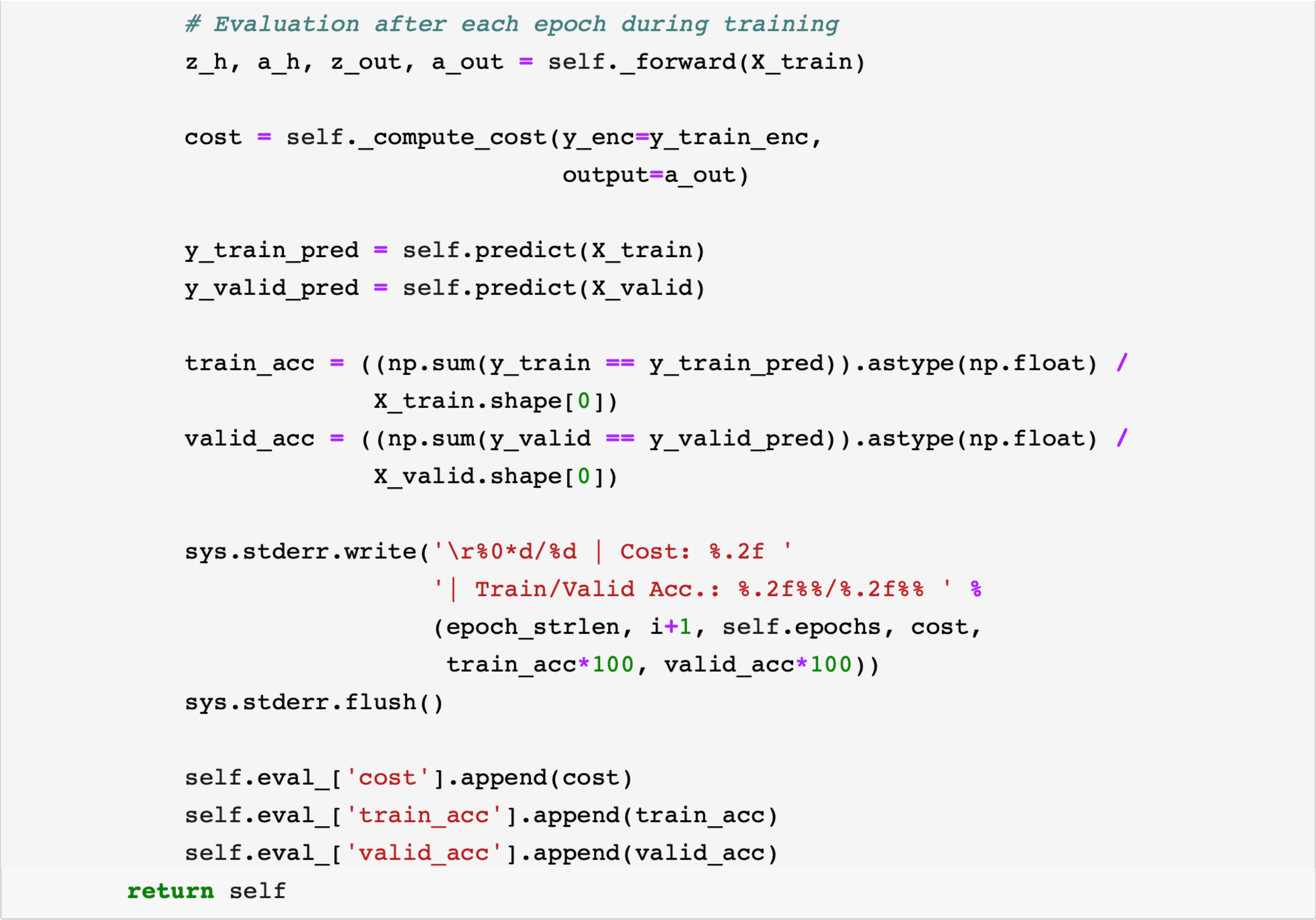


```
        # iterate over training epochs
        for i in range(self.epochs):

            # iterate over minibatches
            indices = np.arange(X_train.shape[0])

            if self.shuffle:
                self.random.shuffle(indices)

            for start_idx in range(0, indices.shape[0] - self.minibatch_size +
                                   1, self.minibatch_size):
                batch_idx = indices[start_idx:start_idx + self.minibatch_size]

                # forward propagation
                z_h, a_h, z_out, a_out = self._forward(X_train[batch_idx])
```

Fit (3/3)

```
#################
# Backpropagation
#################

# [n_samples, n_classlabels]
sigma_out = a_out - y_train_enc[batch_idx]

# [n_samples, n_hidden]
sigmoid_derivative_h = a_h * (1. - a_h)

# [n_samples, n_classlabels] dot [n_classlabels, n_hidden]
# -> [n_samples, n_hidden]
sigma_h = (np.dot(sigma_out, self.w_out.T) *
           sigmoid_derivative_h)

# [n_features, n_samples] dot [n_samples, n_hidden]
# -> [n_features, n_hidden]
grad_w_h = np.dot(X_train[batch_idx].T, sigma_h)
grad_b_h = np.sum(sigma_h, axis=0)
# [n_hidden, n_samples] dot [n_samples, n_classlabels]
# -> [n_hidden, n_classlabels]
grad_w_out = np.dot(a_h.T, sigma_out)
grad_b_out = np.sum(sigma_out, axis=0)

# Regularization and weight updates
delta_w_h = (grad_w_h + self.l2*self.w_h)
delta_b_h = grad_b_h # bias is not regularized
self.w_h -= self.eta * delta_w_h
self.b_h -= self.eta * delta_b_h

delta_w_out = (grad_w_out + self.l2*self.w_out)
delta_b_out = grad_b_out  # bias is not regularized
self.w_out -= self.eta * delta_w_out
self.b_out -= self.eta * delta_b_out
```

Evaluation



```
            # Evaluation after each epoch during training
            z_h, a_h, z_out, a_out = self._forward(X_train)

            cost = self._compute_cost(y_enc=y_train_enc,
                                      output=a_out)

            y_train_pred = self.predict(X_train)
            y_valid_pred = self.predict(X_valid)

            train_acc = ((np.sum(y_train == y_train_pred)).astype(np.float) /
                         X_train.shape[0])
            valid_acc = ((np.sum(y_valid == y_valid_pred)).astype(np.float) /
                         X_valid.shape[0])

            sys.stderr.write('\r%0*d/%d | Cost: %.2f '
                             '| Train/Valid Acc.: %.2f%%/%.2f%% ' %
                             (epoch_strlen, i+1, self.epochs, cost,
                              train_acc*100, valid_acc*100))
            sys.stderr.flush()

            self.eval_['cost'].append(cost)
            self.eval_['train_acc'].append(train_acc)
            self.eval_['valid_acc'].append(valid_acc)
        return self
```

Training and Validation

```
n_epochs = 200
```

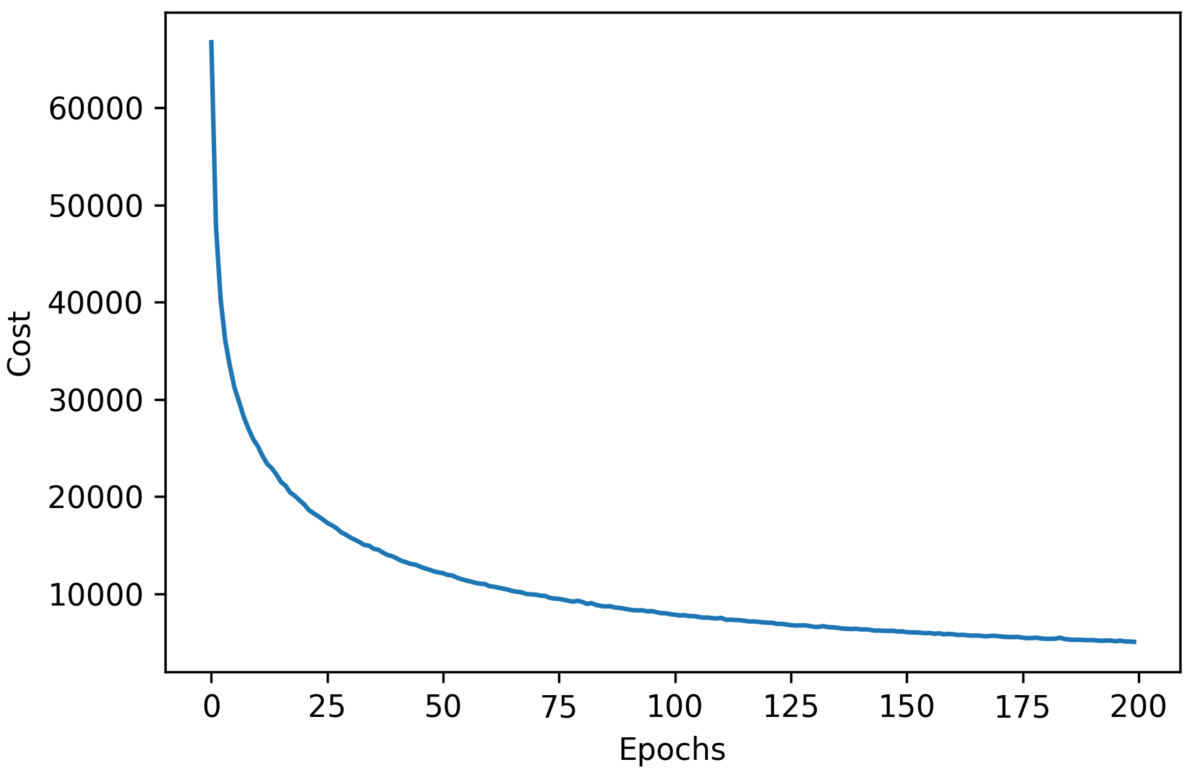
```
nn = NeuralNetMLP(n_hidden=100,
                  l2=0.01,
                  epochs=n_epochs,
                  eta=0.0005,
                  minibatch_size=100,
                  shuffle=True,
                  seed=1)

nn.fit(X_train=X_train[:55000],
       y_train=y_train[:55000],
       X_valid=X_train[55000:],
       y_valid=y_train[55000:])
```

```
200/200 | Cost: 5065.78 | Train/Valid Acc.: 99.28%/97.98%
```

Training Epochs Evaluation

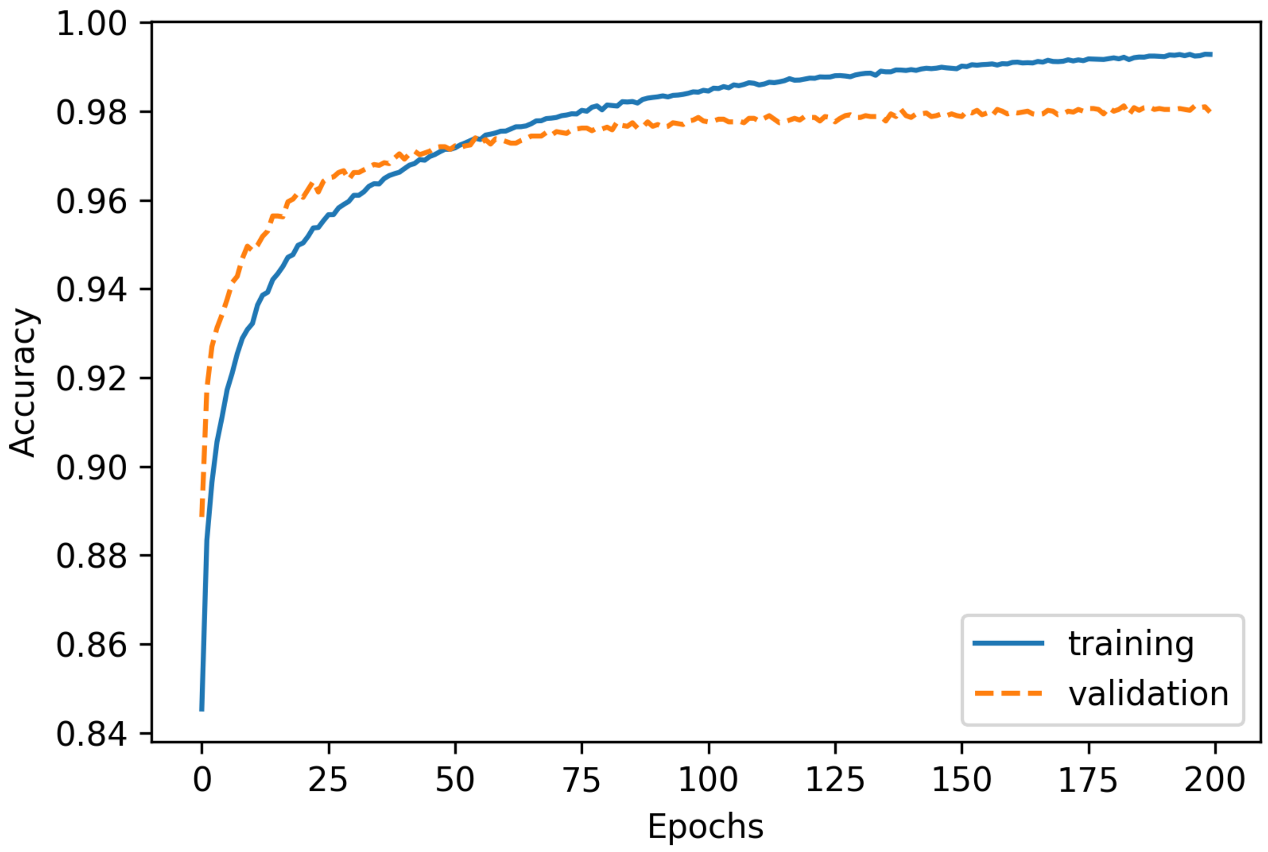
- eval_ attribute

```
import matplotlib.pyplot as plt


plt.plot(range(nn.epochs), nn.eval_['cost'])
plt.ylabel('Cost')
plt.xlabel('Epochs')
#plt.savefig('images/12_07.png', dpi=300)
plt.show()
```

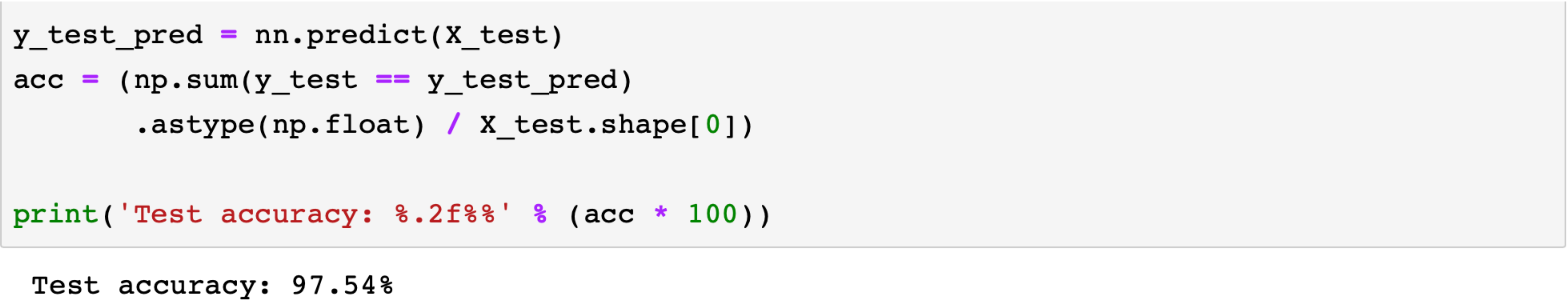


Model Training

```
plt.plot(range(nn.epochs), nn.eval_['train_acc'],
         label='training')
plt.plot(range(nn.epochs), nn.eval_['valid_acc'],
         label='validation', linestyle='--')
plt.ylabel('Accuracy')
plt.xlabel('Epochs')
plt.legend()
#plt.savefig('images/12_08.png', dpi=300)
plt.show()
```



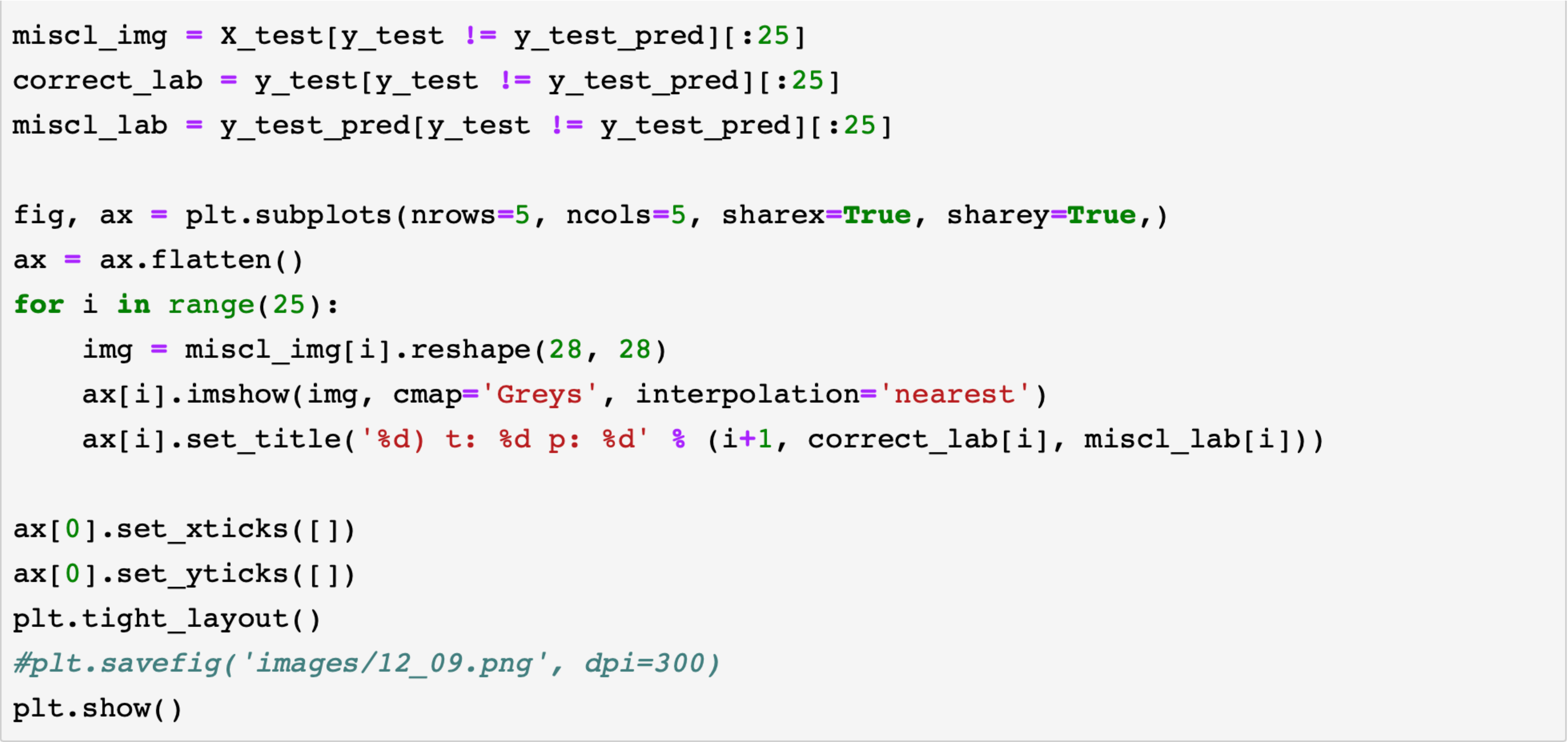
Generalization Performance



```
y_test_pred = nn.predict(X_test)
acc = (np.sum(y_test == y_test_pred)
       .astype(np.float) / X_test.shape[0])

print('Test accuracy: %.2f%%' % (acc * 100))
```

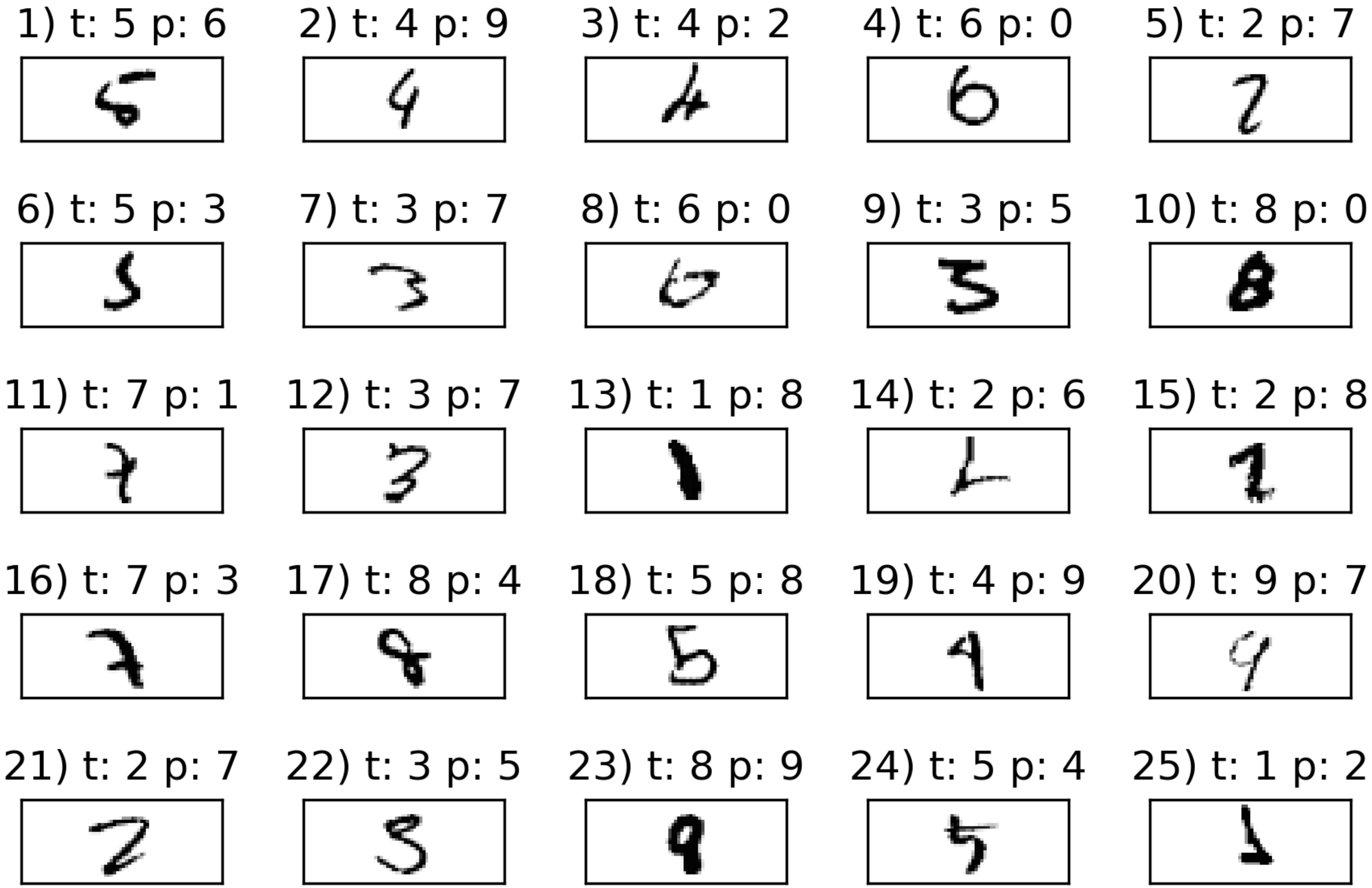
```
Test accuracy: 97.54%
```

Check the Misclassified Samples (1/2)



```
miscl_img = X_test[y_test != y_test_pred][:25]
correct_lab = y_test[y_test != y_test_pred][:25]
miscl_lab = y_test_pred[y_test != y_test_pred][:25]

fig, ax = plt.subplots(nrows=5, ncols=5, sharex=True, sharey=True,)
ax = ax.flatten()
for i in range(25):
    img = miscl_img[i].reshape(28, 28)
    ax[i].imshow(img, cmap='Greys', interpolation='nearest')
    ax[i].set_title('%d) t: %d p: %d' % (i+1, correct_lab[i], miscl_lab[i]))

ax[0].set_xticks([])
ax[0].set_yticks([])
plt.tight_layout()
#plt.savefig('images/12_09.png', dpi=300)
plt.show()
```

Check the Misclassified Samples (2/2)



1) t: 5 p: 6
2) t: 4 p: 9
3) t: 4 p: 2
4) t: 6 p: 0
5) t: 2 p: 7
6) t: 5 p: 3
7) t: 3 p: 7
8) t: 6 p: 0
9) t: 3 p: 5
10) t: 8 p: 0
11) t: 7 p: 1
12) t: 3 p: 7
13) t: 1 p: 8
14) t: 2 p: 6
15) t: 2 p: 8
16) t: 7 p: 3
17) t: 8 p: 4
18) t: 5 p: 8
19) t: 4 p: 9
20) t: 9 p: 7
21) t: 2 p: 7
22) t: 3 p: 5
23) t: 8 p: 9
24) t: 5 p: 4
25) t: 1 p: 2