



# 10. Multilayer Artificial Neural Network

COMP3314  
Machine Learning

# Outline

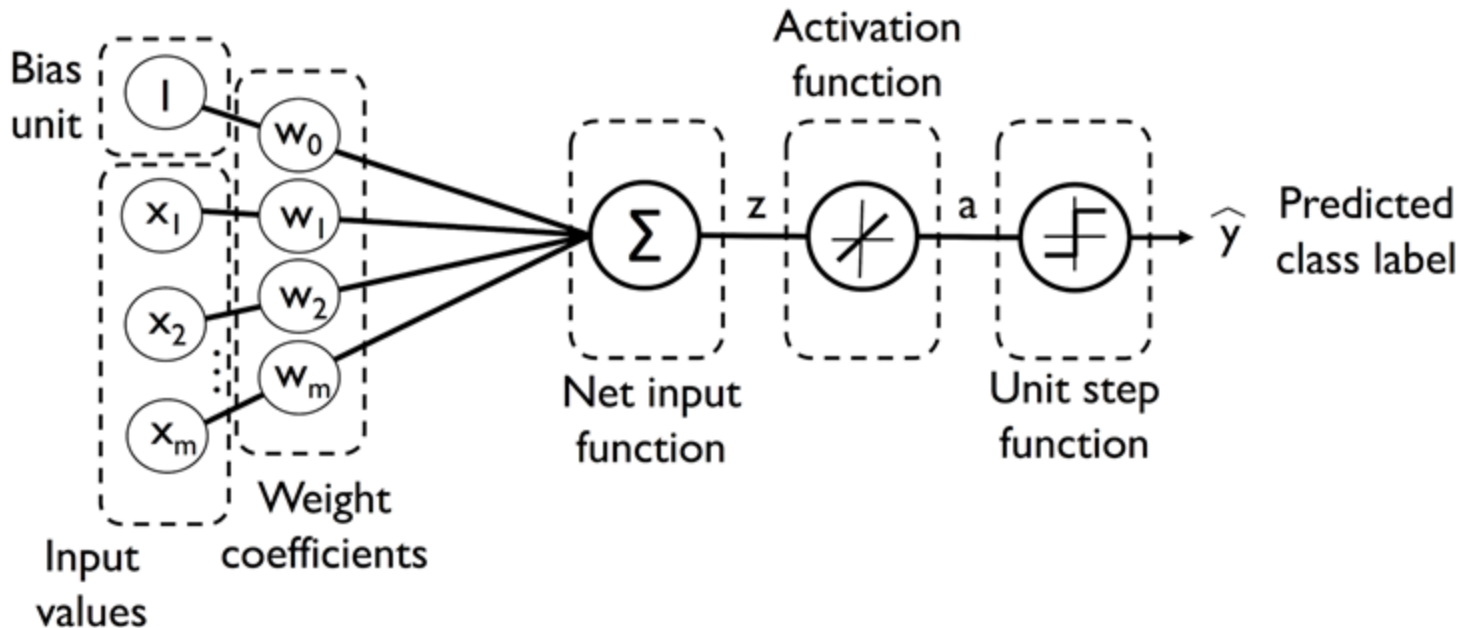
- Deep learning can be understood as a set of algorithms that were developed to train neural networks (NNs) with many layers most efficiently
- In this chapter, you will learn the basic concepts of NNs
- The topics that we will cover in this chapter are as follows
  - Getting a conceptual understanding of multilayer NNs
  - Implementing the fundamental backpropagation algorithm
  - Training a basic multilayer NN for image classification

# AI Winter

- In the decades that followed the first implementation of the McCulloch-Pitt neuron model and Rosenblatt's perceptron in the 1950s, researchers and machine learning practitioners slowly began to lose interest in neural networks
  - No one had a good solution for training a neural network with multiple layers
- Eventually, interest in neural networks (NN) was rekindled in 1986 with a [publication](#) that (re)discovery and popularized the backpropagation algorithm
- However, NNs have never been as popular as they are today, thanks to the many major breakthroughs that have been made in the previous decade
  - This resulted in what we now call deep learning
- Readers who are interested in the history of Artificial Intelligence (AI), machine learning, and NN are encouraged to read [this](#)

# Single-layer Neural Network Recap

- Before we dig deeper into a particular multilayer neural network architecture, let's briefly reiterate some of the concepts of single-layer neural networks that we introduced in Chapter 2



# Single-layer Neural Network Recap

- We used the gradient descent optimization algorithm to learn the weight coefficients of the model
- In every epoch (pass over the training set), we updated the weight vector  $\mathbf{w}$  using the following update rule

$$\mathbf{w} := \mathbf{w} + \Delta \mathbf{w}, \quad \text{where } \Delta \mathbf{w} = -\eta \nabla J(\mathbf{w})$$

- In gradient descent optimization, we updated all weights simultaneously after each epoch, and we defined the partial derivative for each weight  $w_j$  as

$$\frac{\partial}{\partial w_j} J(\mathbf{w}) = -\sum_i (y^{(i)} - a^{(i)}) x_j^{(i)}$$

# Single-layer Neural Network Recap

- We defined the activation function as

$$\phi(z) = z = a$$

- Here, the net input  $z$  is a linear combination of the weights that are connecting the input to the output layer

$$z = \sum_j w_j x_j = \mathbf{w}^T \mathbf{x}$$

- While we used the activation to compute the gradient update, we implemented a threshold function to squash the continuous valued output into binary class labels for prediction

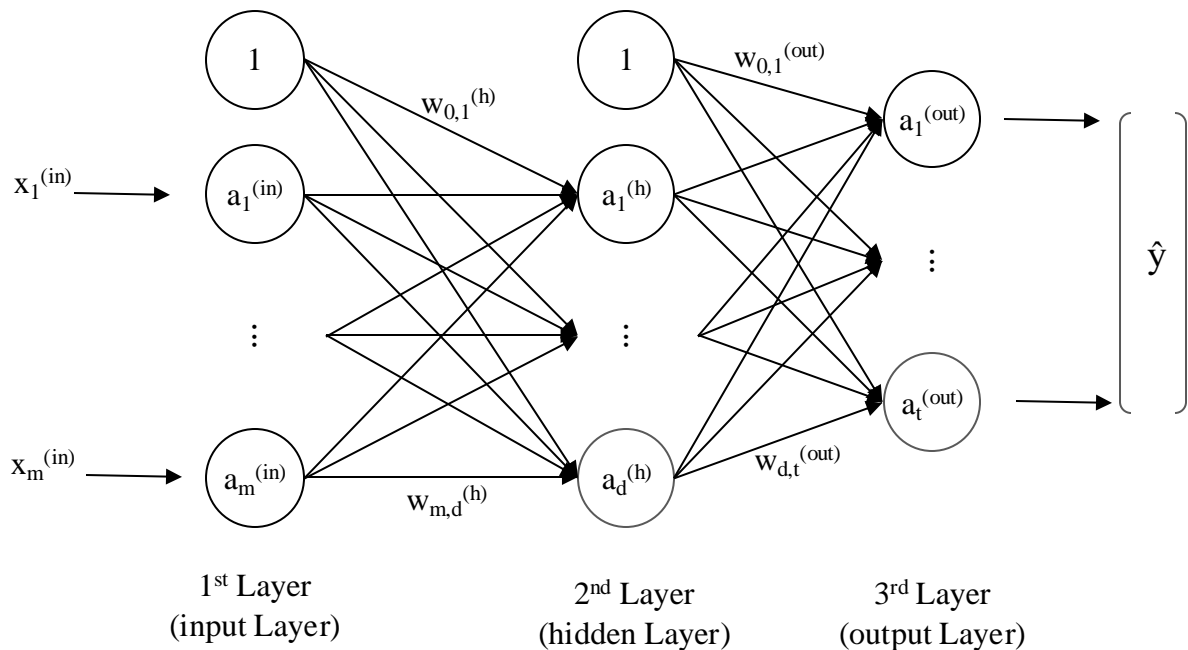
$$\hat{y} = \begin{cases} 1 & \text{if } g(z) \geq 0 \\ -1 & \text{otherwise} \end{cases}$$

# Single-layer Neural Network Recap

- We learned about a certain trick to accelerate the model learning, the so-called stochastic gradient descent optimization
  - Stochastic gradient descent approximates the cost from
    - a single training sample (online learning) or
    - a small subset of training samples (mini-batch learning)
- Apart from faster learning—due to the more frequent weight updates compared to gradient descent—its noisy nature is also regarded as beneficial when training multilayer neural networks with non-linear activation functions, which do not have a convex cost function
  - The added noise can help to escape local cost minima

# Multilayer Neural Network Architecture

- In this section, you will learn how to connect multiple single neurons to a multilayer feedforward NN
  - This special type of fully connected network is also called **Multilayer Perceptron (MLP)**
- The MLP depicted has
  - one input layer,
  - one hidden layer, and
  - one output layer
- If such a network has more than one hidden layer, we call it a **deep artificial NN**



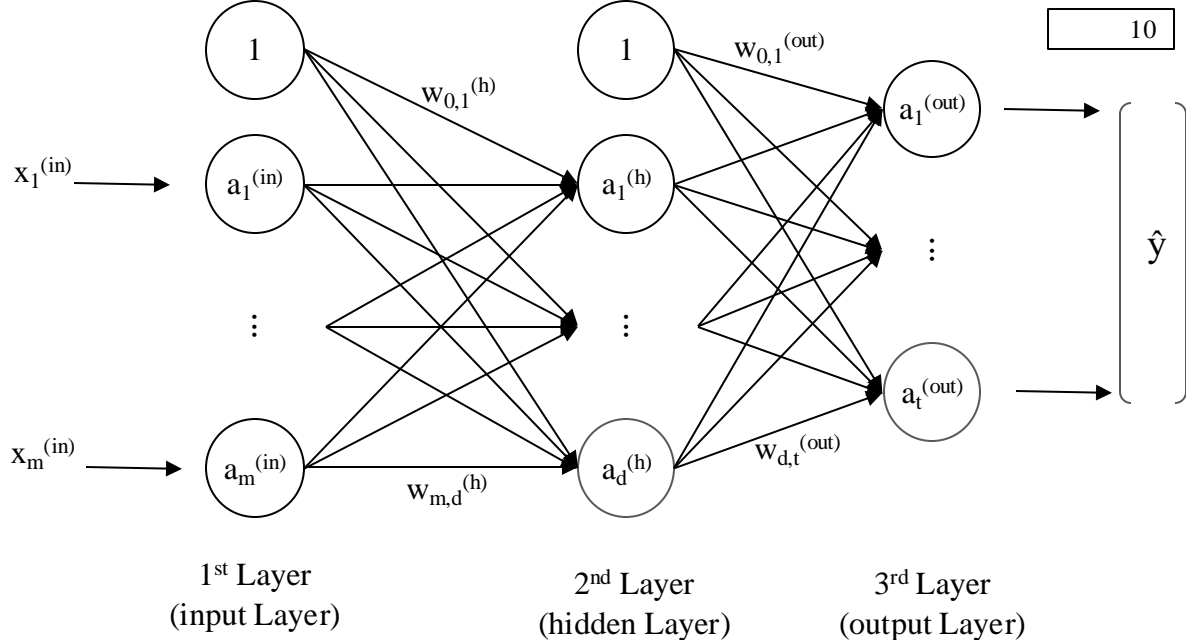


# Deep Learning

- An MLP can have an arbitrary number of hidden layers to create deeper network architectures
  - The number of layers and units in a NN are additional hyperparameters that we want to optimize for a given problem
    - Use the cross-validation technique that we discussed to do this
- Unfortunately, the error gradients that we will calculate later via backpropagation will become increasingly small as more layers are added to a network
  - This vanishing gradient problem makes the model learning more challenging
- Therefore, special algorithms have been developed to help train such deep neural network structures
  - This is known as deep learning

# Notation

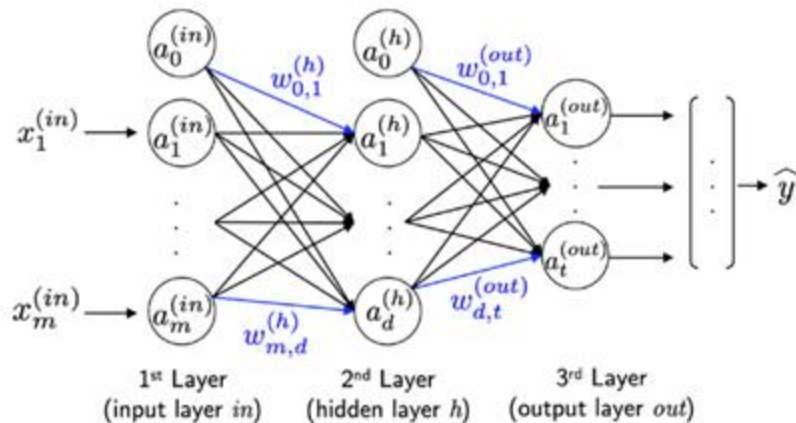
- Let's denote the  $i$ -th activation unit in the  $l$ -th layer as  $a_i^{(l)}$
- For our simple MLP we use the
  - $in$  for the input layer,
  - $h$  for the hidden layer
  - $out$  for the output layer
- For instance
  - $a_i^{(in)}$  is the  $i$ -th value in the input layer
  - $a_i^{(h)}$  is the  $i$ -th unit in the hidden layer
  - $a_i^{(out)}$  is the  $i$ -th unit in the output layer



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

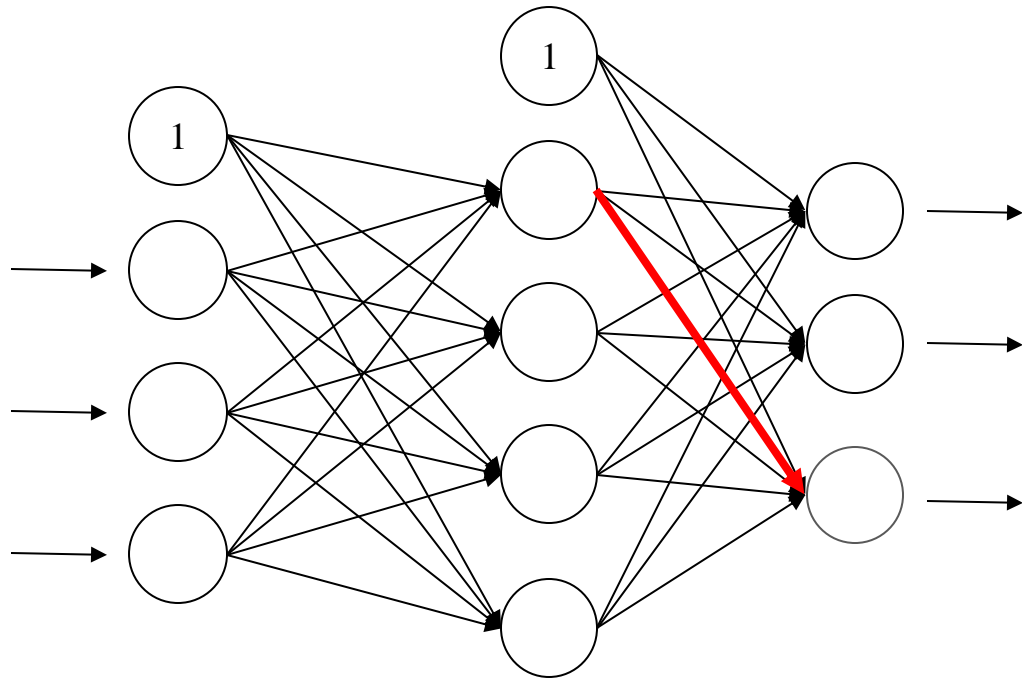
# Notation

- Each unit in layer  $l$  is connected to all units in layer  $l + 1$  via a weight coefficient
  - E.g., the connection between the  $k$ -th unit in layer  $l$  to the  $j$ -th unit in layer  $l + 1$  will be written as  $w_{k,j}^{(l+1)}$
- We denote the weight matrix that connects the input to the hidden layer as  $\mathbf{W}^{(h)}$ , and we write the matrix that connects the hidden layer to the output layer as  $\mathbf{W}^{(out)}$



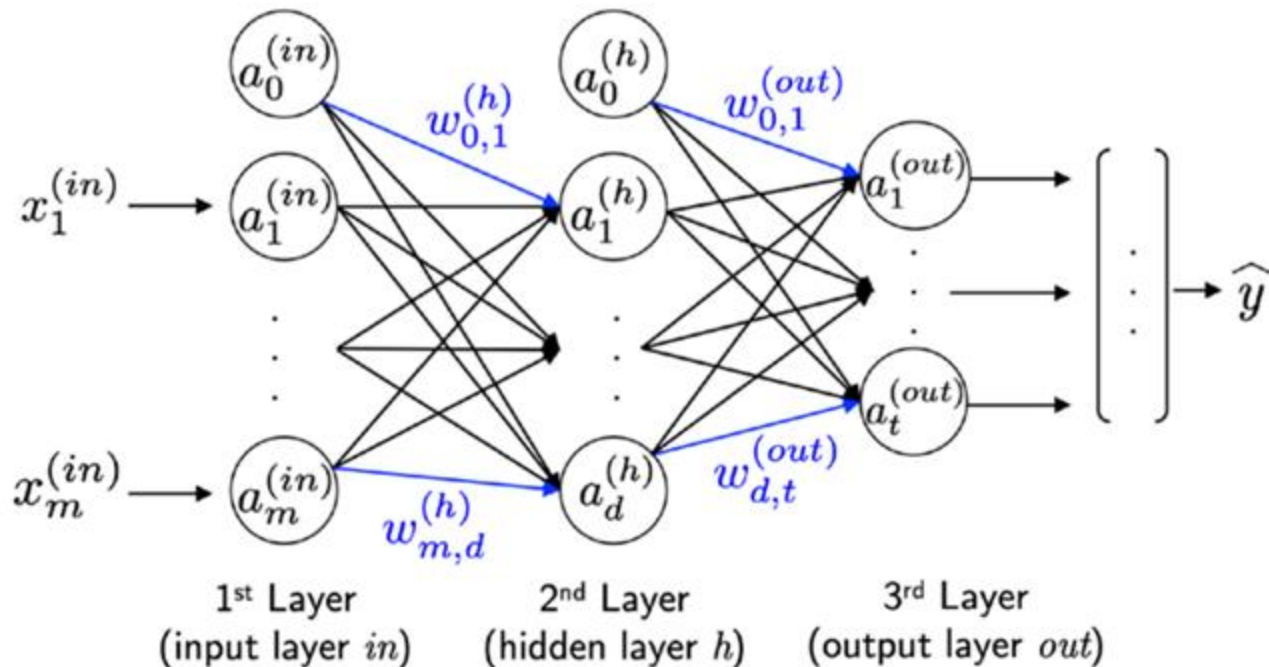
# Quiz

1.  $m = ?$  (exclude bias)
2.  $d = ?$  (exclude bias)
3.  $t = ?$
4. Number of layers  $L = ?$
5. What is the notation for the weight of the thick arc ?
6. What is the notation for the unit that the thick arc points to ?
7. How many weights (incl. for the bias) are there in this neural net ?



# Quiz

- Let  $m = 39$ ,  $d = 43$  and  $t = 10$ . How many weights are there in this network?



# Learning Procedure

- The MLP learning procedure can be summarized in three simple steps
  - Starting at the input layer, we **forward propagate** the patterns of the training data through the network to generate an output
  - Based on the network's output, we **calculate the error** that we want to minimize using a cost function
  - We **backpropagate the error**, find its derivative with respect to each weight in the network, and update the model
- Repeat these three steps for multiple epochs to learn the weights of the MLP
- Use forward propagation to calculate the network output and apply a threshold function to obtain the predicted class labels in the one-hot representation

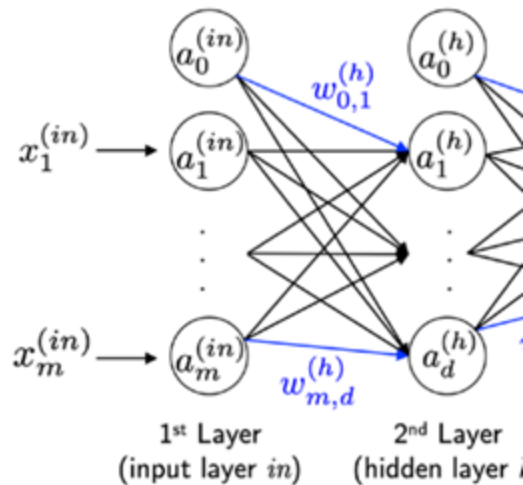
# Forward Propagation

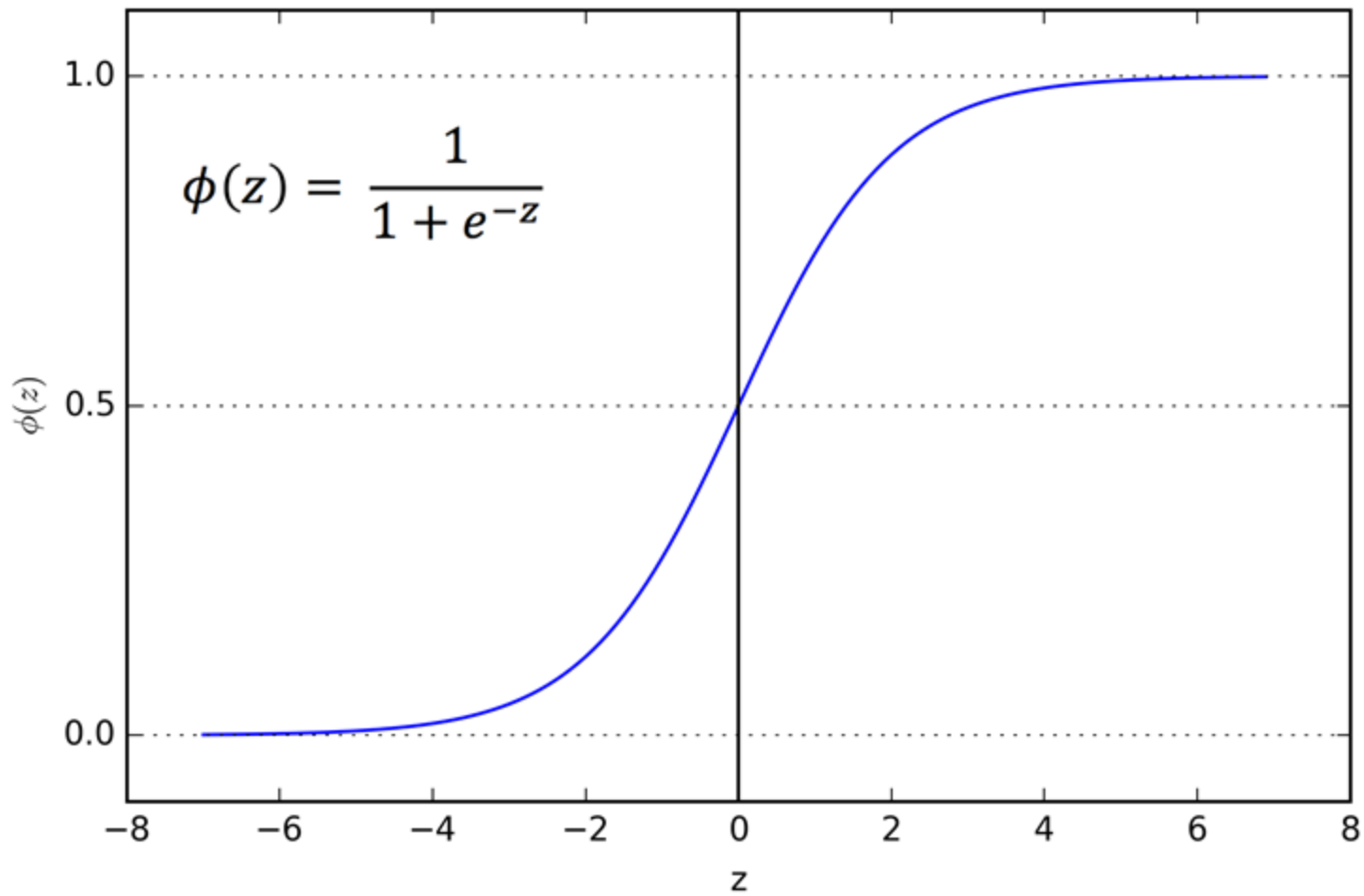
- Let's walk through the individual steps of forward propagation to generate an output from the patterns in the training data
- Since each unit in the hidden layer is connected to all units in the input layers, we first calculate the first activation unit of the hidden layer  $a_1^{(h)}$  as follows

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

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

- $z_1^{(h)}$  is the net input
- $\phi$  is the activation function
  - It has to be differentiable to learn the weights that connect the neurons using a gradient-based approach
  - To solve complex problems such as image classification, we need non-linear activation function, e.g., sigmoid







# MLP - A Feedforward NN

- MLP is a typical example of a **feedforward NN**
  - The term feedforward refers to the fact that each layer serves as the input to the next layer without loops, in contrast to a recurrent NN ([RNN](#))

# MLP - A Confusing Name

- The term multilayer perceptron (MLP) is little bit confusing
  - Note that the neurons in an MLP are typically sigmoid units, not perceptrons
- Intuitively, we can think of the neurons in the MLP as logistic regression units that return values in the continuous range between 0 and 1
- Sometimes the term MLP is used loosely to any feedforward NN
  - We use MLP to refer to networks composed of multiple layers of logistic regression units
- Multilayer perceptrons are sometimes colloquially referred to as "vanilla" neural networks
  - Especially when they have a single hidden layer

# Forward Propagation

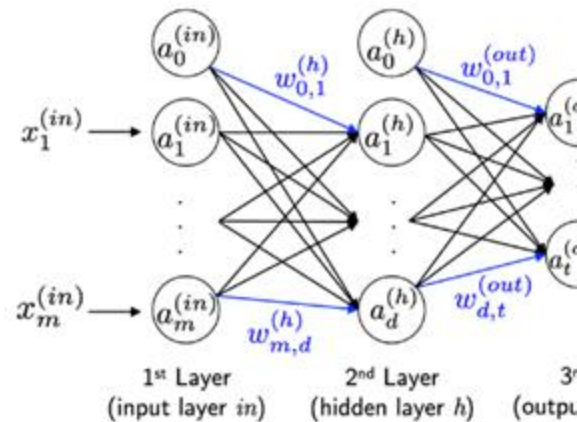
$$z_1^{(h)} = w_{0,1}^{(h)} + a_1^{(in)}w_{1,1}^{(h)} + \dots + a_m^{(in)}w_{m,1}^{(h)}$$

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

- We will now write the activation in a more compact form
  - This will allow us to vectorize operations via NumPy rather than writing multiple nested and computationally expensive loops

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

$$\mathbf{a}^{(h)} = \phi(z^{(h)})$$

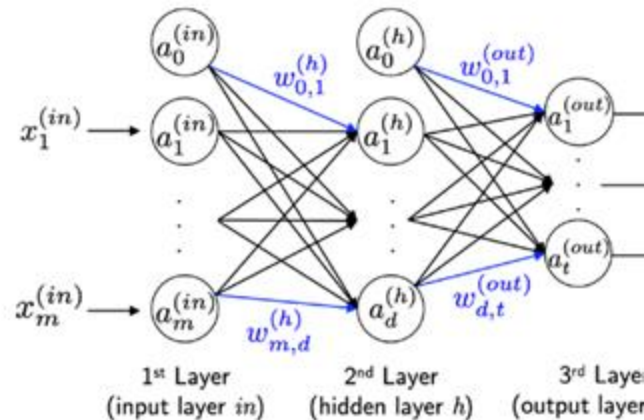


- $\mathbf{a}^{(in)}$  is our  $1 \times m$  dimensional feature vector of a sample  $\mathbf{x}^{(in)}$  and a bias unit
- $\mathbf{W}^{(h)}$  is an  $m \times d$  dimensional weight matrix where  $d$  is the number of units in the hidden layer
- After matrix-vector multiplication, we obtain the  $1 \times d$  dimensional net input vector  $z^{(h)}$  to calculate the activation  $\mathbf{a}^{(h)}$  ( where  $\mathbf{a}^{(h)} \in \mathbb{R}^{1 \times d}$  )

# Forward Propagation (all samples)

- We can generalize the computation on the previous slide to all  $n$  samples in the training set

$$\mathbf{Z}^{(h)} = \mathbf{A}^{(in)} \mathbf{W}^{(h)}$$



- $\mathbf{A}^{(in)}$  is a  $n \times m$  matrix, and the matrix-matrix multiplication will result in an  $n \times d$  dimensional net input matrix  $\mathbf{Z}^{(h)}$
- Finally, we apply the activation function  $\phi$  to each value in the net input matrix to get the  $n \times d$  activation matrix  $\mathbf{A}^{(h)}$  for the next layer

$$\mathbf{A}^{(h)} = \phi(\mathbf{Z}^{(h)})$$

# Forward Propagation (all samples)

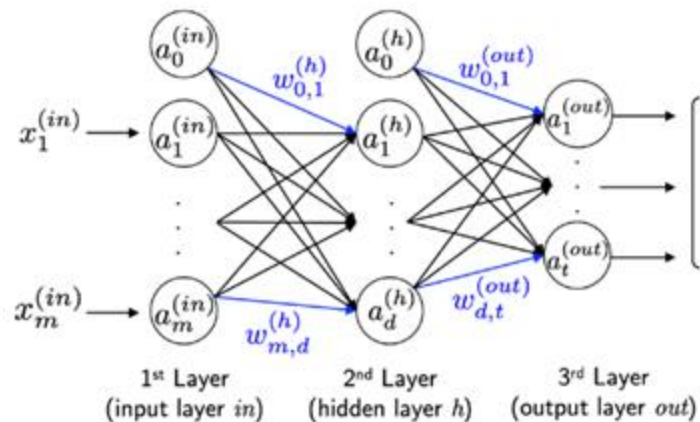
- We can also write the activation of the output layer in vectorized form

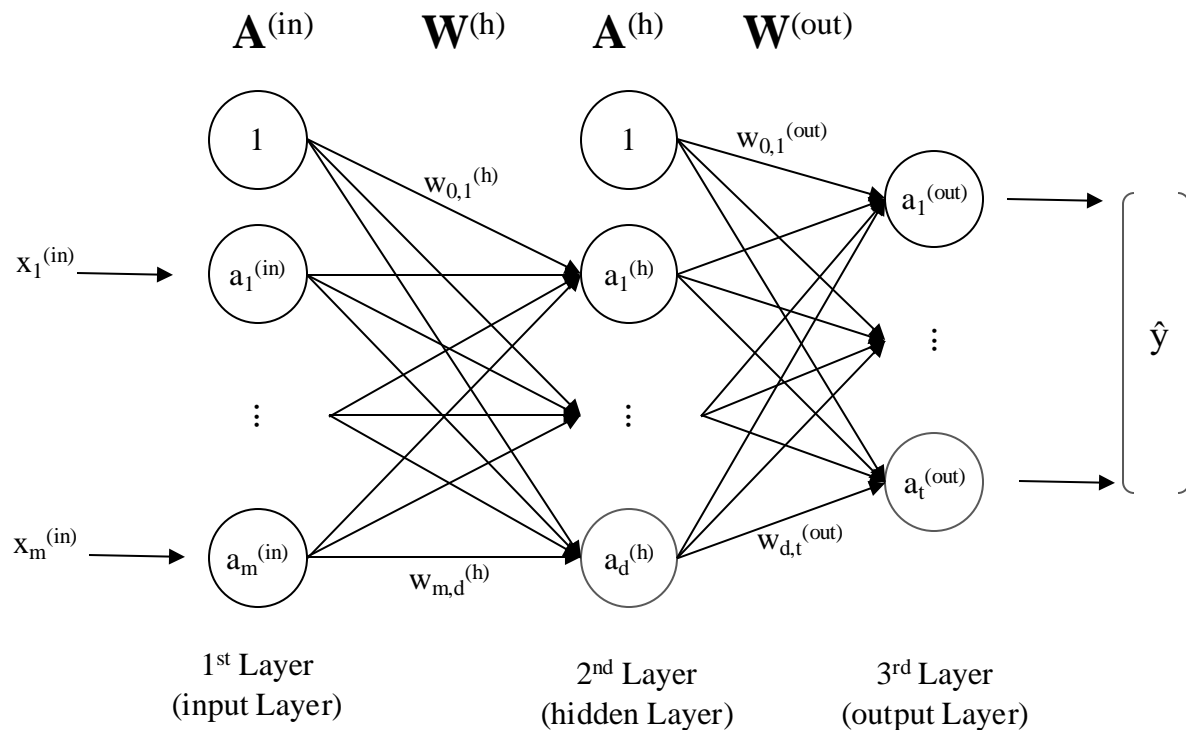
$$\mathbf{Z}^{(\text{out})} = \mathbf{A}^{(h)} \mathbf{W}^{(\text{out})}$$

- Here, we multiply the  $d \times t$  matrix  $\mathbf{W}^{(\text{out})}$  ( $t$  is the number of output units) with the  $n \times d$  dimensional matrix  $\mathbf{A}^{(h)}$  to obtain the  $n \times t$  dimensional matrix  $\mathbf{Z}^{(\text{out})}$  (the columns in this matrix represent the outputs for each sample)
- Lastly, we apply the sigmoid activation function to obtain the continuous valued output of our network

$$\mathbf{A}^{(\text{out})} = \phi(\mathbf{Z}^{(\text{out})})$$

- $\mathbf{A}^{(\text{out})}$  is a  $n \times t$  matrix





Note: equations  
exclude bias

$$\begin{array}{ccccccc}
 \mathbf{A}^{(in)} & \mathbf{W}^{(h)} & = & \mathbf{Z}^{(h)} & \longrightarrow & \phi(\mathbf{Z}^{(h)}) = \mathbf{A}^{(h)} & \longrightarrow & \mathbf{A}^{(h)} \mathbf{W}^{(out)} = \mathbf{Z}^{(out)} & \longrightarrow & \phi(\mathbf{Z}^{(out)}) = \mathbf{A}^{(out)} \\
 n \times m & m \times d & & p \times d & & n \times d & & n \times d & & n \times t & & n \times t \\
 & & & p \times d & & p \times d & & d \times t & & t \times t & & t \times t
 \end{array}$$

# Example: Classifying handwritten digits

- Let's take a short break from the theory and see a neural network in action
- We will implement and train our first multilayer neural network to classify handwritten digits from the popular Mixed National Institute of Standards and Technology (MNIST) dataset
  - [MNIST](#) was constructed by [Yann LeCun](#) and others
    - Reference: [Gradient-Based Learning Applied to Document Recognition](#), Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, 1998
  - The training set consists of handwritten digits from 250 different people
    - 50 percent high school students
    - 50 percent employees from the census bureau
  - The test set contains handwritten digits from people not in the training set

# MNIST Dataset

- Obtaining MNIST
  - The dataset is publicly available [here](#) and consists of the following four parts
    - Training set images: [train-images-idx3-ubyte.gz](#)  
(9.9 MB, 47 MB unzipped, 60,000 samples)
    - Training set labels: [train-labels-idx1-ubyte.gz](#)  
(29 KB, 60 KB unzipped, 60,000 labels)
    - Test set images: [t10k-images-idx3-ubyte.gz](#)  
(1.6 MB, 7.8 MB, 10,000 samples)
    - Test set labels: [t10k-labels-idx1-ubyte.gz](#)  
(5 KB, 10 KB unzipped, 10,000 labels)
- Each image in the dataset consist of 28 x 28 pixels
  - Each pixel is represented by a grayscale intensity value



# Code - MLP.ipynb

- Available [here](#) on CoLab

# Downloading & Unzipping MNIST

```
1 !wget http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz
2 !wget http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz
3 !wget http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz
4 !wget http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz
```

```
1 !gzip -d train-images-idx3-ubyte.gz
2 !gzip -d train-labels-idx1-ubyte.gz
3 !gzip -d t10k-images-idx3-ubyte.gz
4 !gzip -d t10k-labels-idx1-ubyte.gz
```

```
1 import os
2 import struct
3 import numpy as np
4 def load_mnist(path, kind='train'):
5     labels_path = os.path.join(path, '%s-labels-idx1-ubyte' % kind)
6     images_path = os.path.join(path, '%s-images-idx3-ubyte' % kind)
7     with open(labels_path, 'rb') as lbpath:
8         magic, n = struct.unpack('>II', lbpath.read(8))
9         labels = np.fromfile(lbpath, dtype=np.uint8)
10    with open(images_path, 'rb') as imgpath:
11        magic, num, rows, cols = struct.unpack(">IIII", imgpath.read(16))
12        images = np.fromfile(imgpath, dtype=np.uint8).reshape(len(labels), 784)
13        images = ((images / 255.) - .5) * 2
14    return images, labels
```

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

Rows: 60000, columns: 784

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

Rows: 10000, columns: 784

# load\_mnist()

- The load\_mnist function returns two arrays
  - An  $n \times m$  dimensional NumPy array (images), where  $n$  is the number of samples and  $m$  is the number of features (here, pixels)
- The images consist of  $28 \times 28$  pixels, and each pixel is represented by a gray scale intensity value
- Here, we unroll the  $28 \times 28$  pixels into one-dimensional row vectors, which represent the rows in our images array (784 per row or image)
- The second array (labels) returned by the load\_mnist function contains the corresponding target variable, the class labels (integers 0-9) of the handwritten digits
- Note that we normalized the pixels values in MNIST to the range -1 to 1 (originally 0 to 255) via the following code line

```
images = ((images / 255.) - .5) * 2
```

# Plotting

- Let's visualize examples of the digits 0-9
- To do this we reshape the 784-pixel vectors from our feature matrix into a  $28 \times 28$  image

```
1 import matplotlib.pyplot as plt
2 fig, ax = plt.subplots(nrows=2, ncols=5, sharex=True, sharey=True,)
3 ax = ax.flatten()
4 for i in range(10):
5     img = X_train[y_train == i][0].reshape(28, 28)
6     ax[i].imshow(img, cmap='Greys')
7 ax[0].set_xticks([])
8 ax[0].set_yticks([])
9 plt.tight_layout()
10 plt.show()
```



# Plotting

- In addition, let's also plot multiple examples of the same digit to see how different the handwriting really is
- We should now see the first 25 variants of the digit 7

```
1 fig, ax = plt.subplots(nrows=5, ncols=5, sharex=True, sharey=True,)
2 ax = ax.flatten()
3 for i in range(25):
4     img = X_train[y_train == 7][i].reshape(28, 28)
5     ax[i].imshow(img, cmap='Greys')
6 ax[0].set_xticks([])
7 ax[0].set_yticks([])
8 plt.tight_layout()
9 plt.show()
```



# np.savez()

- To avoid the overhead of reading in and processing the data again we save the scaled images in a format that we can load quickly
- A method to save multidimensional arrays to disk is NumPy's [savez](#) function
- The following code snippet will save both the training and test datasets to the archive file 'mnist\_scaled.npz'

```
1 import numpy as np
2 np.savez_compressed('mnist_scaled.npz', X_train=X_train, y_train=y_train, X_test=X_test, y_test=y_test)
3 !ls -l
```

```
total 86584
-rw-r--r-- 1 root root 22088791 Nov 18 05:13 mnist_scaled.npz
drwxr-xr-x 1 root root    4096 Nov 13 17:33 sample_data
-rw-r--r-- 1 root root  7840016 Jul 21  2000 t10k-images-idx3-ubyte
-rw-r--r-- 1 root root 16488877 Jul 21  2000 t10k-images-idx3-ubyte.gz
-rw-r--r-- 1 root root   10008 Jul 21  2000 t10k-labels-idx1-ubyte
-rw-r--r-- 1 root root    4542 Jul 21  2000 t10k-labels-idx1-ubyte.gz
-rw-r--r-- 1 root root 47040016 Jul 21  2000 train-images-idx3-ubyte
-rw-r--r-- 1 root root  9912422 Jul 21  2000 train-images-idx3-ubyte.gz
-rw-r--r-- 1 root root   60008 Jul 21  2000 train-labels-idx1-ubyte
-rw-r--r-- 1 root root   28881 Jul 21  2000 train-labels-idx1-ubyte.gz
```

# np.load()

- After we created the .npz files, we can load the preprocessed MNIST image arrays using NumPy's load function

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

- The mnist variable now references to an object that can access the four data arrays as we provided them keyword arguments to the np.savez function
- Using a list comprehension, we can retrieve all four data arrays as follows

```
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)
```



# Loading MNIST using scikit-learn

- Using scikit-learn's new [fetch\\_openml](#) function, it is now also possible to load the MNIST dataset more conveniently
- For example, you can use the following code to create a 50,000-example training dataset and a 10,000-example test dataset by fetching the dataset
  - From [www.openml.org/d/554](http://www.openml.org/d/554)

```
1 from sklearn.datasets import fetch_openml
2 from sklearn.model_selection import train_test_split
3 X, y = fetch_openml('mnist_784', version=1, return_X_y=True)
4 y = y.astype(int)
5 X = ((X / 255.) - .5) * 2
6 X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=10000, random_state=123, stratify=y)
```

```
1 X_train.shape
```

```
(60000, 784)
```

# Implementing a Multilayer Perceptron

- Let's now implement an MLP with one input, one hidden, and one output layer
- Note
  - The code will contain parts that we have not talked about yet
    - Such as the [backpropagation algorithm](#)
  - We will use separate vectors for bias
    - Why?
      - More efficient and easier to read
      - Also used by popular libraries such as [TensorFlow](#)
    - Why didn't I add the bias to the equations on the slides?
      - Equations would appear more complex if we had to work with bias
      - Appending the 1s to the input vector (as shown previously) and using a weight variable as bias is exactly the same as operating with separate bias vectors
        - Just a different convention

```

import numpy as np
import sys
class NeuralNetMPLP(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)
        self.n_hidden = n_hidden
        self.l2 = l2
        self.epochs = epochs
        self.eta = eta
        self.shuffle = shuffle
        self.minibatch_size = minibatch_size

    def _onehot(self, y, n_classes):
        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):
        return 1. / (1. + np.exp(-np.clip(z, -250, 250)))

    def _forward(self, X):
        z_h = np.dot(X, self.w_h) + self.b_h
        a_h = self._sigmoid(z_h)
        z_out = np.dot(a_h, self.w_out) + self.b_out
        a_out = self._sigmoid(z_out)
        return z_h, a_h, z_out, a_out

```

$$\begin{array}{ccccc}
 \mathbf{A}^{(\text{in})} & \mathbf{W}^{(\text{h})} & = & \mathbf{Z}^{(\text{h})} & \longrightarrow & \phi(\mathbf{Z}^{(\text{h})}) = \mathbf{A}^{(\text{h})} & \longrightarrow & \\
 \begin{array}{c} n \times m \\ m \times p \end{array} & \begin{array}{c} m \times d \\ p \times d \end{array} & & \begin{array}{c} n \times d \\ p \times d \end{array} & & \begin{array}{c} n \times d \\ p \times d \end{array} & & 
 \end{array}$$

$$\begin{array}{ccccc}
 \longrightarrow & \mathbf{A}^{(\text{h})} & \mathbf{W}^{(\text{out})} & = & \mathbf{Z}^{(\text{out})} & \longrightarrow & \phi(\mathbf{Z}^{(\text{out})}) = \mathbf{A}^{(\text{out})} & \longrightarrow & \\
 \begin{array}{c} n \times d \\ p \times d \end{array} & \begin{array}{c} d \times t \\ d \times t \end{array} & \begin{array}{c} n \times t \\ p \times t \end{array} & & \begin{array}{c} n \times t \\ p \times t \end{array} & & \begin{array}{c} n \times t \\ p \times t \end{array} & & 
 \end{array}$$

• Rectangular Snip

```
def _compute_cost(self, y_enc, output):  
    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  
  
def predict(self, X):  
    z_h, a_h, z_out, a_out = self._forward(X)  
    y_pred = np.argmax(z_out, axis=1)  
    return y_pred
```

```

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]
    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))
    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))
    self.eval_ = {'cost': [], 'train_acc': [], 'valid_acc': []}
    y_train_enc = self._onehot(y_train, n_output)
    for i in range(self.epochs):
        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):
            |...
        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%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

```

```
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]
    z_h, a_h, z_out, a_out = self._forward(X_train[batch_idx])
    delta_out = a_out - y_train_enc[batch_idx]
    sigmoid_derivative_h = a_h * (1. - a_h)
    delta_h = (np.dot(delta_out, self.w_out.T) *
               sigmoid_derivative_h)
    grad_w_h = np.dot(X_train[batch_idx].T, delta_h)
    grad_b_h = np.sum(delta_h, axis=0)
    grad_w_out = np.dot(a_h.T, delta_out)
    grad_b_out = np.sum(delta_out, axis=0)
    delta_w_h = (grad_w_h + self.l2*self.w_h)
    delta_b_h = grad_b_h
    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
    self.w_out -= self.eta * delta_w_out
    self.b_out -= self.eta * delta_b_out
```

# Implementing a Multilayer Perceptron

- If you read through the NeuralNetMLP code, you've probably already guessed what these parameters are for
  - `l2`: This is the  $\lambda$  parameter for L2 regularization to decrease the degree of overfitting
  - `epochs`: This is the number of passes over the training set
  - `eta`: This is the learning rate  $\eta$
  - `shuffle`: This is for shuffling the training set prior to every epoch to prevent that the algorithm gets stuck in circles
  - `seed`: This is a random seed for shuffling and weight initialization
  - `minibatch_size`: This is the number of training samples in each mini-batch when splitting of the training data in each epoch for stochastic gradient descent
    - The gradient is computed for each mini-batch separately instead of the entire training data for faster learning

# Training on MNIST

- In general, training (deep) NNs is relatively computationally expensive compared with the other models we discussed so far
- Thus, we may want to stop it early and start over with different hyperparameter
  - If we find that it increasingly tends to overfit the training data (noticeable by an increasing gap between training and validation set performance), we may want to stop the training early
- Let's now train the MLP using 55,000 samples from the MNIST training dataset and use the remaining 5,000 samples for 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:])
```

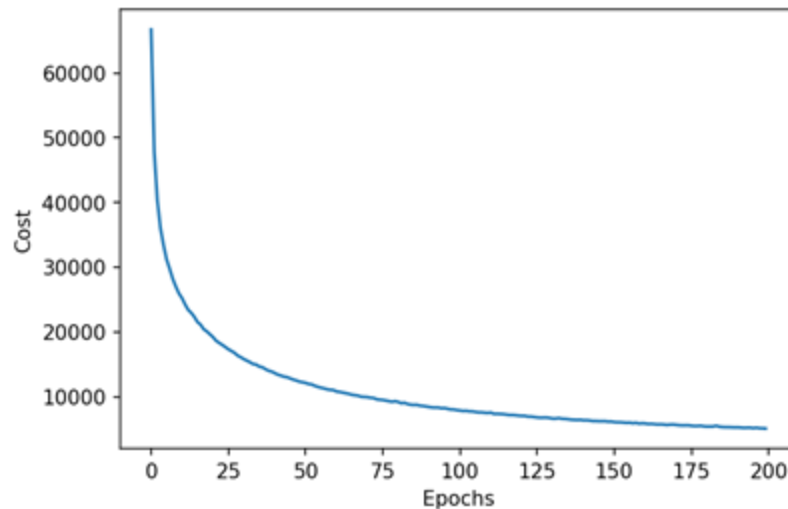
200/200 | Cost: 5065.78 | Train/Valid Acc.: 99.28%/97.98%



# Cost Plot

- In our NeuralNetMLP implementation, we defined an eval\_ attribute that collects the cost, training, and validation accuracy for each epoch so that we can visualize the results using Matplotlib
- The following code plots the cost over the 200 epochs

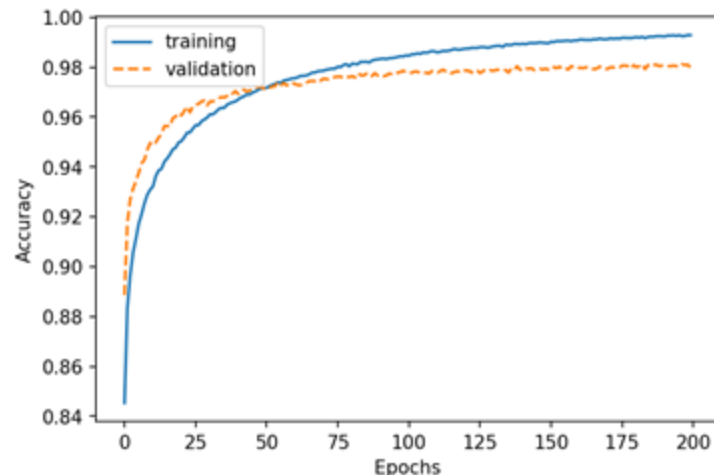
```
import matplotlib.pyplot as plt
import matplotlib as mpl
mpl.rcParams['figure.dpi'] = 150
plt.plot(range(nn.epochs), nn.eval_['cost'])
plt.ylabel('Cost')
plt.xlabel('Epochs')
plt.show()
```



# Training and Validation Accuracy

```
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.show()
```

- The plot reveals that the gap between training and validation accuracy increases the more epochs we train the network
- At approximately the 50th epoch, the training and validation accuracy values are equal, and then, the network starts overfitting the training data



# Generalization Performance

- Finally, let's evaluate the generalization performance of the model by calculating the prediction accuracy on the test set

```
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%

- Despite the slight overfitting on the training data, our relatively simple one-hidden layer neural network achieved a relatively good performance on the test dataset
- To further fine-tune the model, we could change the number of hidden units, values of the regularization parameters, and the learning rate or use various other tricks that have been developed over the years

# Misclassifications

- Let's take a look at some of the images that our MLP struggles with

```
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.show()
```

1) t: 5 p: 6



2) t: 4 p: 9



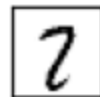
3) t: 4 p: 2



4) t: 6 p: 0



5) t: 2 p: 7



6) t: 5 p: 3



7) t: 3 p: 7



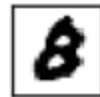
8) t: 6 p: 0



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



# Training Details - Compute Cost

- Let's dig a little bit deeper into some of the concepts, such as the logistic cost function and the backpropagation algorithm that we implemented to learn the weights
- The logistic cost function that we implemented as the `_compute_cost` method is the same cost function that we described in the logistic regression section earlier

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

Here,  $a^{[i]}$  is the sigmoid activation of the  $i$ -th sample in the dataset, which we compute in the forward propagation step

$$a^{[i]} = \phi(z^{[i]})$$

```
def _compute_cost(self, y_enc, output):  
    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
```

# Training Details - Compute Cost

- Let's add a regularization term, which allows us to reduce the degree of overfitting
  - As you recall from earlier chapters, the L2 regularization term is defined as follows (we don't regularize the bias units)

$$L2 = \lambda \|\mathbf{w}\|_2^2 = \lambda \sum_{j=1}^m w_j^2$$

- By adding the L2 regularization term to our logistic cost function, we obtain the following equation

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

```
def _compute_cost(self, y_enc, output):  
    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
```

# Training Details - Compute Cost

- Since we implemented an MLP for multiclass classification that returns an output vector of  $t$  elements that we need to compare to the  $t \times 1$  dimensional target vector in the one-hot encoding representation, for example, the activation of the third layer and the target class for a particular sample may look like this

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

- We need to generalize the logistic cost function to all  $t$  activation units in our network
- The cost function (without the regularization term) becomes the following

$$J(W) = -\sum_{i=1}^n \sum_{j=1}^t y_j^{[i]} \log(a_j^{[i]}) + (1 - y_j^{[i]}) \log(1 - a_j^{[i]})$$



# Training Details - Compute Cost

- The generalized regularization term just calculates the sum of all weights of an  $l$  layer (without the bias term) that we added to the first column

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

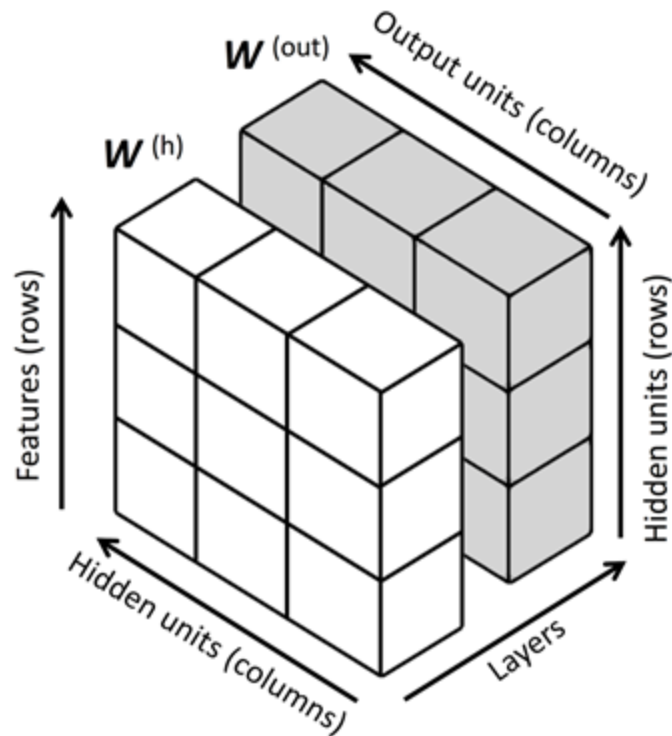
- Here,  $u_l$  refers to the number of units in a given layer  $l$
- Remember that our goal is to minimize the cost function  $J(\mathbf{W})$ 
  - Thus we need to calculate the partial derivative of the parameters  $\mathbf{W}$  with respect to each weight for every layer in the network

$$\frac{\partial}{\partial w_{j,i}^{(l)}} J(\mathbf{W})$$

# Training Details - Compute Cost

$$\frac{\partial}{\partial w_{j,i}^{(l)}} J(W)$$

- Note that  $W$  consists of multiple matrices
- In a multilayer perceptron with one hidden unit
  - $W^{(h)}$  connects the input to the hidden layer, and
  - $W^{(out)}$  connects the hidden layer to the output layer
- Note that  $W^{(h)}$  and  $W^{(out)}$  typically don't have the same number of rows and columns, unless we initialize an MLP with the same number of hidden units, output units, and input features



# Backpropagation - Big Picture

- Backpropagation was popularized more than 30 years ago
  - It is the algorithms used to train NNs, i.e., determine the weights
- Big picture
  - Compute the partial derivatives of a cost function
  - Use the partial derivatives to update weights
    - This is challenging because we are dealing with many weights
  - The error is not convex or smooth with respect to the parameters
    - Unlike a single-layer neural network
    - There are many bumps in this high-dimensional cost surface that we have to overcome in order to find the global minimum of the cost function

# Backpropagation - Big Picture

- Recall the concept of the chain rule
  - The chain rule computes the derivative of a function, such as  $f(g(x))$ , as follows

$$\frac{d}{dx}[f(g(x))] = \frac{df}{dg} \cdot \frac{dg}{dx}$$

- We can use the chain rule for an arbitrarily long function composition

$$\frac{dF}{dx} = \frac{d}{dx} F(x) = \frac{d}{dx} f\left(g\left(h\left(u\left(v(x)\right)\right)\right)\right) = \frac{df}{dg} \cdot \frac{dg}{dh} \cdot \frac{dh}{du} \cdot \frac{du}{dv} \cdot \frac{dv}{dx}$$

- In computer algebra, [automatic differentiation](#) (AD) has been developed
  - AD comes with two modes: forward and reverse
  - Backpropagation is simply [a special case](#) of reverse AD

# Backpropagation - Big Picture

- The key point is that applying the chain rule in the forward mode is expensive because we would have to multiply large matrices for each layer that we eventually multiply by a vector to obtain the output
- The trick of reverse mode is that we start from right to left
  - We multiply a matrix by a vector, which yields another vector that is multiplied by the next matrix and so on
- Matrix-vector multiplication is computationally much cheaper than matrix-matrix multiplication
  - This leads to the efficiency of backpropagation and made it one of the most popular algorithms used in neural network training

# Backpropagation: Math

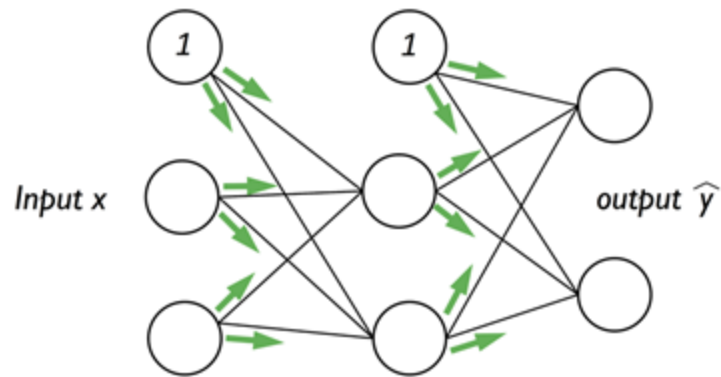
- We will go through the math of backpropagation to understand how you can learn the weights in a neural network efficiently
- Previously, we saw how to calculate the cost as the difference between the activation of the last layer and the target class label
- Now, we will see how the backpropagation algorithm works to update the weights in our MLP model
- Recall from earlier, we first need to apply forward propagation in order to obtain the activation of the output layer, which we formulated as shown in the upper right
  - We just forward-propagate the input features through the connection in the network

$$\mathbf{Z}^{(h)} = \mathbf{A}^{(in)} \mathbf{W}^{(h)} \text{ (net input of the hidden layer)}$$

$$\mathbf{A}^{(h)} = \phi(\mathbf{Z}^{(h)}) \text{ (activation of the hidden layer)}$$

$$\mathbf{Z}^{(out)} = \mathbf{A}^{(h)} \mathbf{W}^{(out)} \text{ (net input of the output layer)}$$

$$\mathbf{A}^{(out)} = \phi(\mathbf{Z}^{(out)}) \text{ (activation of the output layer)}$$



Forward Propagation

# Backpropagation: Math

- We propagate the error from right to left
- We start by calculating the *error term* of the output layer

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

- $\mathbf{y}$  is the vector of the true class labels
  - In the code: `delta_out = a_out - y_train_enc[batch_idx]`
- Next, we calculate the *error term* of the hidden layer

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

- The last term is the derivative of the activation function

$$\frac{\partial \phi(z)}{\partial z} = \left( a^{(h)} \odot (1 - a^{(h)}) \right)$$

- In the code `sigmoid_derivative_h = a_h * (1. - a_h)`

$\odot$  : element-wise multiplication

# Derivative of Sigmoid

$$\phi'(z) = \frac{\partial}{\partial z} \left( \frac{1}{1 + e^{-z}} \right)$$

$$= \frac{e^{-z}}{(1 + e^{-z})^2}$$

$$= \frac{1 + e^{-z}}{(1 + e^{-z})^2} - \left( \frac{1}{1 + e^{-z}} \right)^2$$

$$= \frac{1}{(1 + e^{-z})} - \left( \frac{1}{1 + e^{-z}} \right)^2$$

$$= \phi(z) - (\phi(z))^2$$

$$= \phi(z)(1 - \phi(z))$$

$$= a(1 - a)$$



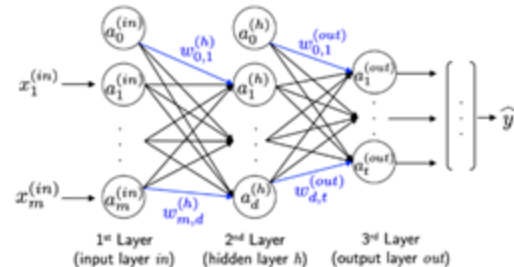
# Backpropagation: Math

- The error term of the hidden layer is therefore

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

- $\mathbf{W}^{(out)}$  is  $d \times t$  matrix
  - $t$  is the number of output class labels
  - $d$  is the number of hidden units
- The matrix multiplication between the  $n \times t$  matrix  $\delta^{(out)}$  and the  $t \times d$  matrix  $(\mathbf{W}^{(out)})^T$ , results in a  $n \times d$  matrix that we multiply element-wise with the derivative of the sigmoid (which is also a  $n \times d$  matrix)

```
sigmoid_derivative_h = a_h * (1. - a_h)
sigma_h = (np.dot(sigma_out, self.w_out.T) * sigmoid_derivative_h)
```



# Backpropagation: Math

- After obtaining the  $\delta$  terms, we can now write the derivation of the cost function as follows
  - Detailed derivations are omitted, please watch the video on slide 62

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

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

# Backpropagation: Math

- Next, we need to accumulate the partial derivative of every node in each layer and the error of the node in the next layer
- However, remember that we need to compute  $\Delta_{i,j}^{(l)}$  for every sample in the training set
- Thus, it is easier to implement it as a vectorized version, like in our code

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

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

# Backpropagation: Math

- After we have accumulated the partial derivatives, we can add the regularization term

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

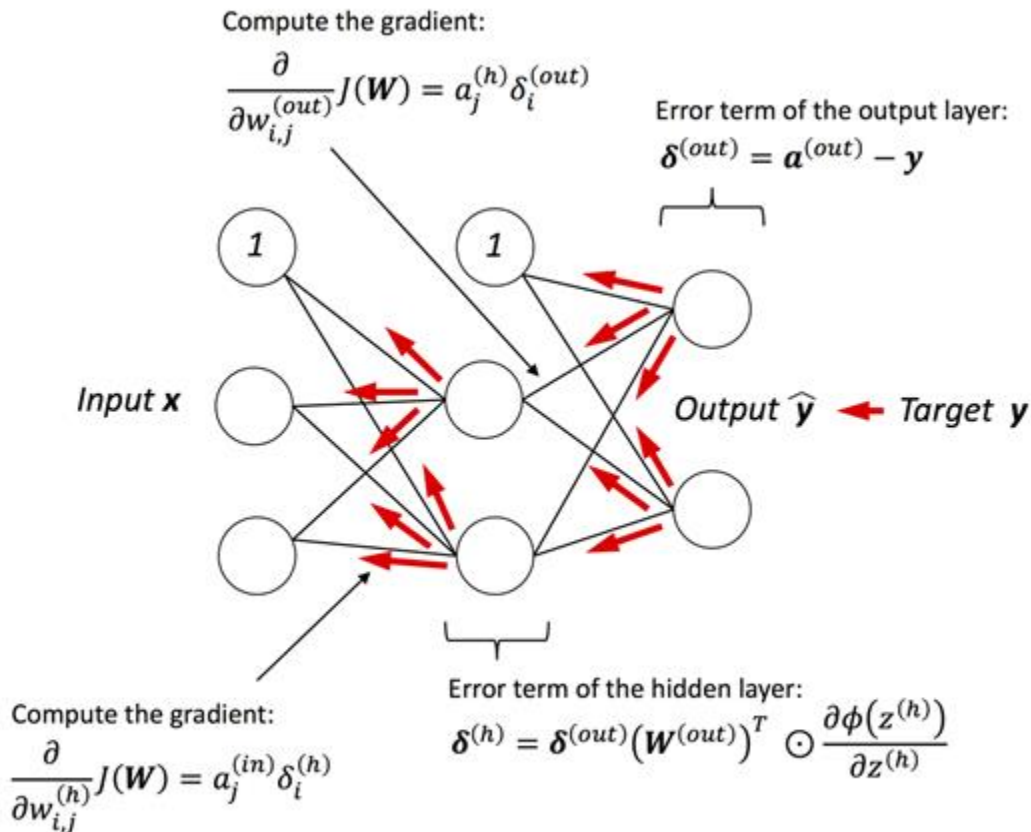
- The two previous mathematical equations correspond to the code variables `delta_w_h`, `delta_b_h`, `delta_w_out`, and `delta_b_out` in our code
- Lastly, after we have computed the gradients, we can now update the weights by taking an opposite step towards the gradient for each layer  $l$

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

- This is implemented as follows

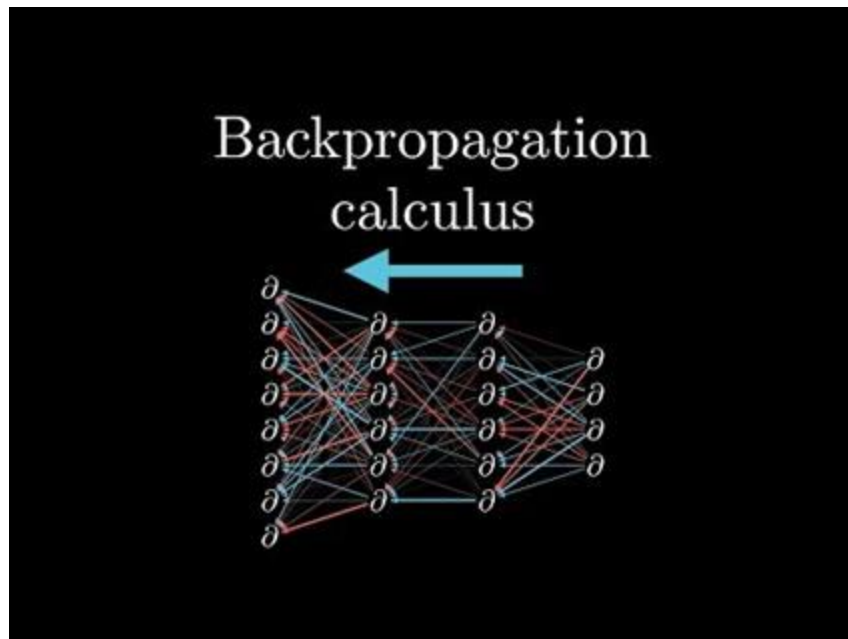
```
self.w_h -= self.eta * delta_w_h
self.b_h -= self.eta * delta_b_h
self.w_out -= self.eta * delta_w_out
self.b_out -= self.eta * delta_b_out
```

# Backpropagation: Math (Summary)



# Backpropagation: Math

- To obtain a better mathematical understanding of this, please watch the following video

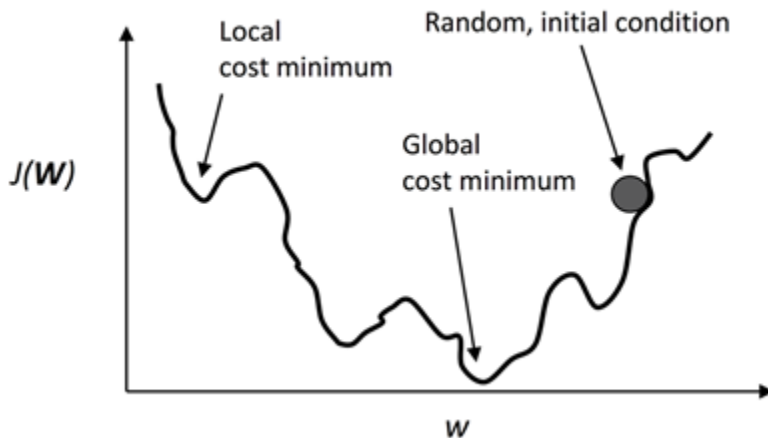


# Convergence

- Why not use regular gradient descent instead of mini-batch learning?
- Recall our discussion on stochastic gradient descent
  - In online learning, we compute the gradient based on a single training example ( $k = 1$ ) at a time to perform the weight update
  - Although this is a stochastic approach, it often leads to very accurate solutions with a much faster convergence than regular gradient descent
- Mini-batch learning is a special form of stochastic gradient descent where we compute the gradient based on a subset  $k$  of the  $n$  training samples with  $1 < k < n$
- Mini-batch learning has the advantage over online learning that we can make use of our vectorized implementations to improve computational efficiency
- Intuitively, you can think of mini-batch learning as predicting the voter turnout of a presidential election from a poll by asking only a representative subset of the population rather than asking the entire population

# Convergence

- Multilayer neural networks are much harder to train than simpler algorithms such as Adaline, logistic regression, or support vector machines
- In multilayer neural networks, we typically have hundreds, thousands, or even billions of weights that we need to optimize
- Unfortunately, the output function has a rough surface and the optimization algorithm can easily become trapped in local minima





# NN Implementations

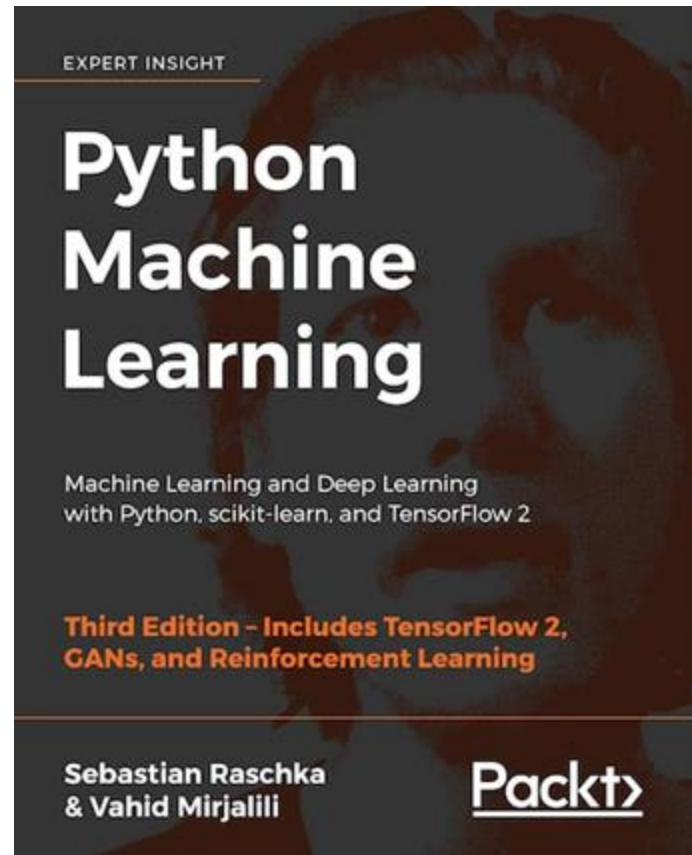
- We use open source machine learning libraries to implement NNs in practice
- In the next chapter we will be using TensorFlow (TF) library
  - Released in November 2015
  - Able to utilize graphics processing units (GPUs) for optimal performance
    - Much more efficient than any of our previous NumPy implementations
  - TensorFlow can be considered a low-level deep learning library
  - Simplifying APIs such as Keras have been developed on top of it (and part of TF) that make the construction of common deep learning models even more convenient

# Conclusion

- Although the implementation in this chapter seems a bit tedious at first, it was a good exercise for understanding the basics behind backpropagation and neural network training, and a basic understanding of algorithms is crucial for applying machine learning techniques appropriately and successfully

# References

- Most materials in this chapter are based on
  - [Book](#)
  - [Code](#)



# References

- Chapter 6, Deep Feedforward Networks, Deep Learning, I. Goodfellow, Y. Bengio, and A. Courville, MIT Press, 2016. (Manuscripts freely accessible [here](#))

# References

- Pattern Recognition and Machine Learning, C. M. Bishop and others, Volume 1. Springer New York, 2006.