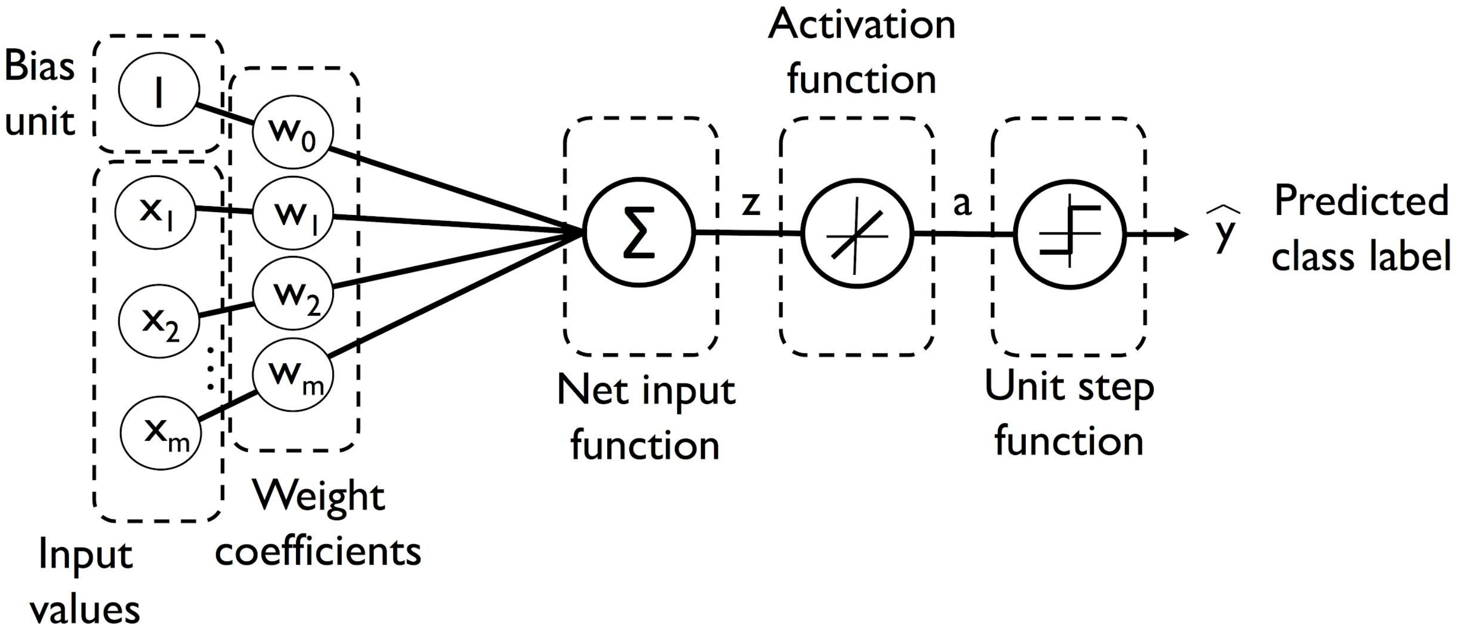
Multilayer Artificial Neural Network

COMP3314

Machine Learning

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 before



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 **w** using the following update rule

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

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

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

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

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

- 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

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

Single-layer Neural Network Recap

- We learned about a certain trick to accelerate 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)

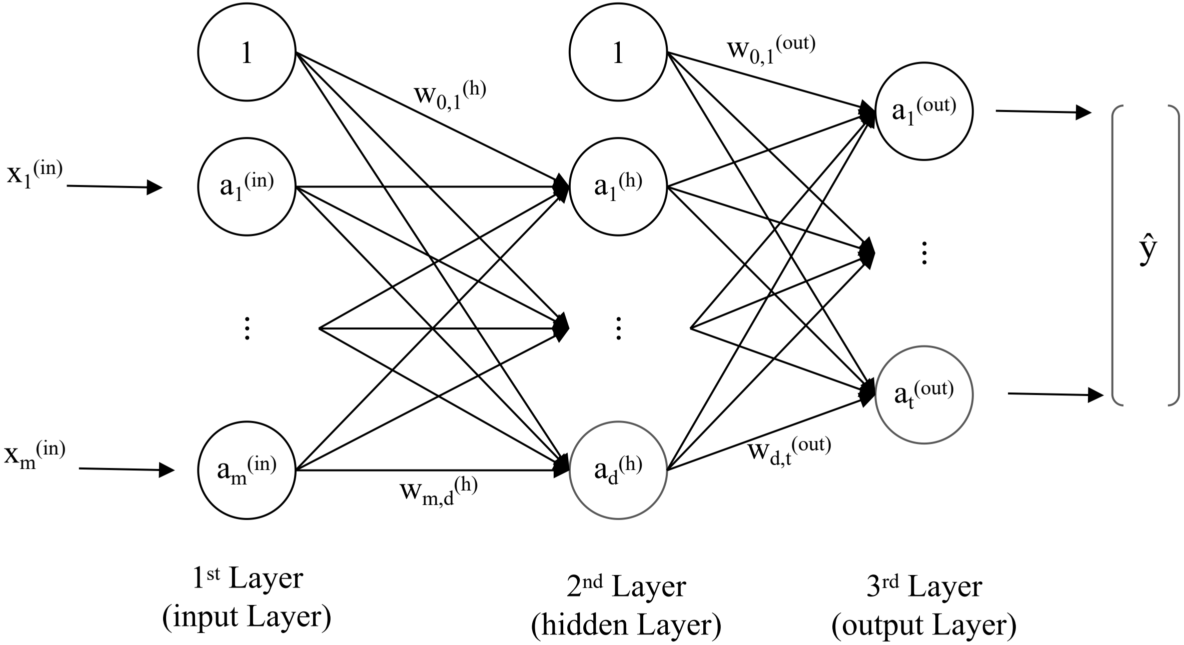
- Its noisy nature is also beneficial when training multilayer neural networks with non-linear activation functions, which do not have a convex cost function
  - The added noise can help to escape local cost minima

Multilayer Neural Network Architecture

- Here you will learn how to connect multiple neurons to a multilayer feedforward NN
  - This type of fully connected network is also called **Multilayer Perceptron** (MLP)

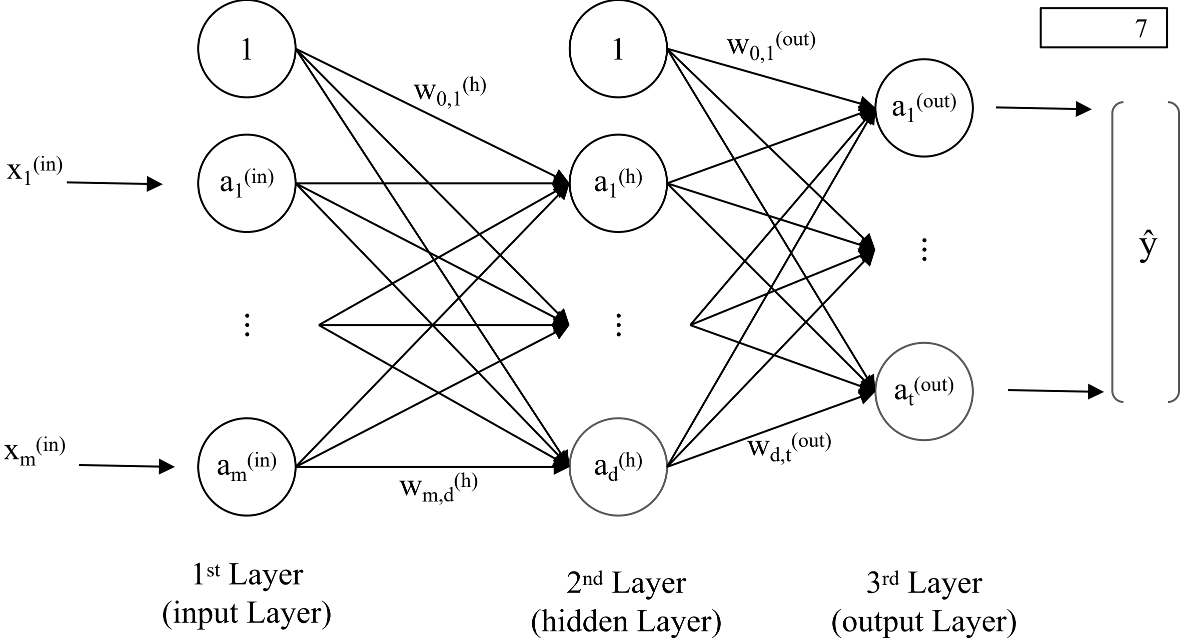
- The MLP depicted has
  - one input layer,
  - one hidden layer, and
  - one output layer

- If such a network has more than one hidden layer, we call it a **deep artificial neural network**



Notation

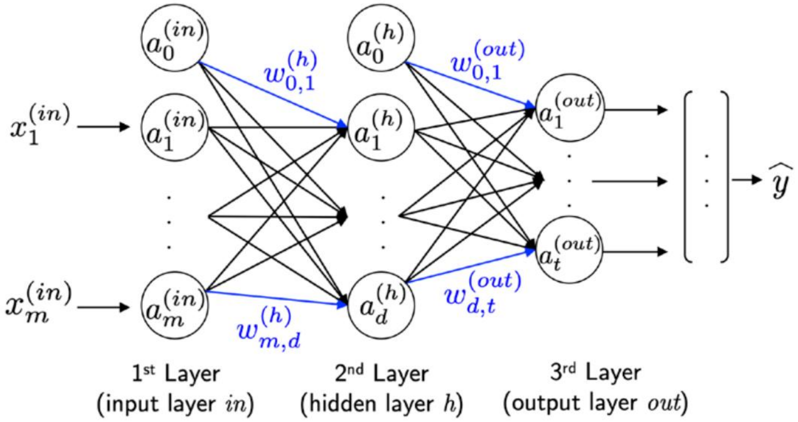
- Let's denote the *i*-th activation unit in the *l*-th layer as $a_i^{(l)}$
- For our simple MLP we use the
  - *in* for the input layer,
  - *h* for the hidden layer
  - *out* for the output layer
- For instance
  - $a_i^{(in)}$ is the *i*-th value in the input layer
  - $a_i^{(h)}$ is the *i*-th unit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



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

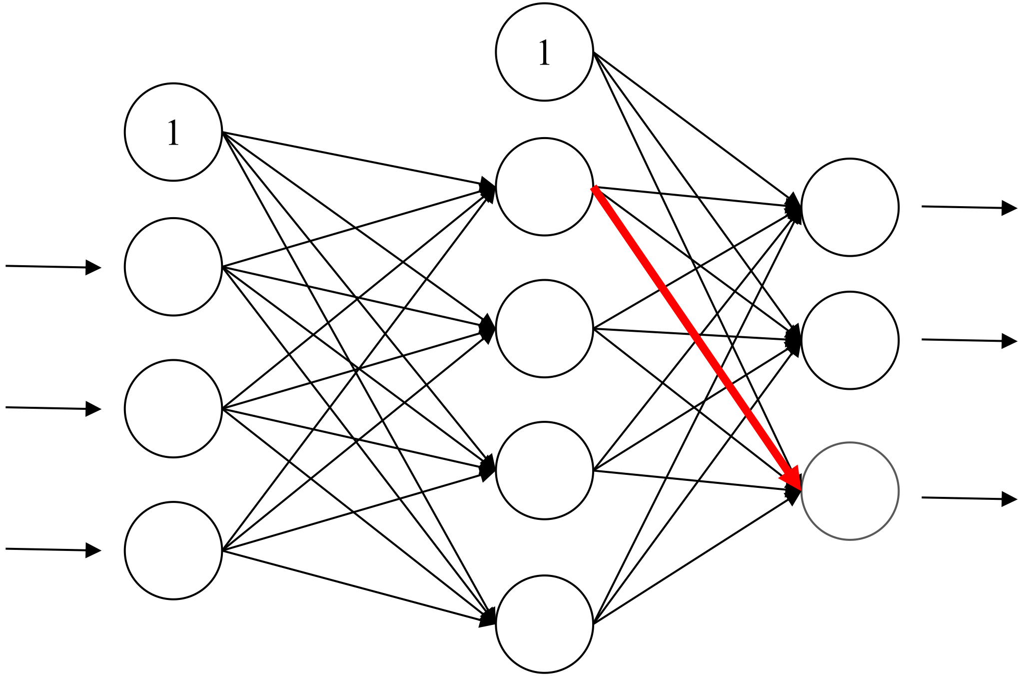
Notation

- 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 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. Home many weights (incl. for the bias) are there in this neural net ?

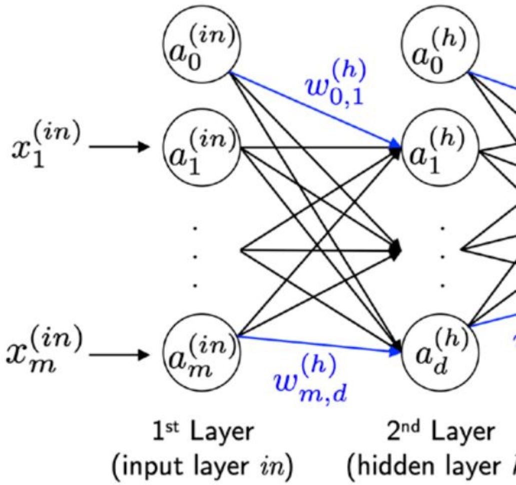


Learning Procedure

- 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

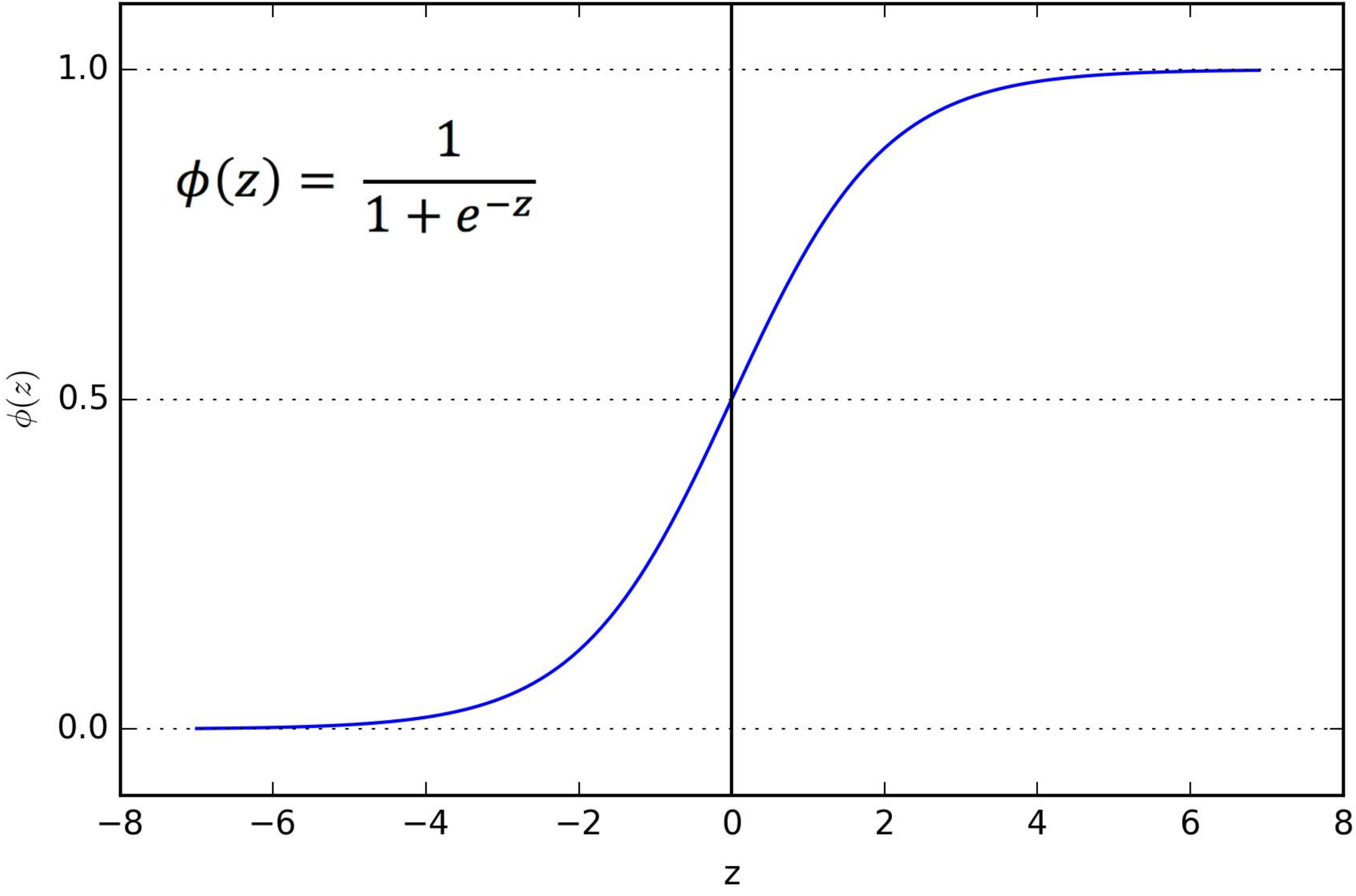
- Let's walk through the individual steps of forward propagation to generate an output from the patterns in the training data
- Since each unit in the hidden layer is connected to all units in the input layers, we first calculate the first activation unit of the hidden layer $a_1^{(h)}$ as follows



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

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

  - $z_1^{(h)}$ is the net input
  - ϕ 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


$\phi(z) = \frac{1}{1+e^{-z}}$
1.0
0.5
0.0
$\phi(z)$
−8
−6
−4
−2
0
2
4
6
8
z

Multilayer Perceptron (MLP)

- MLP is a typical example of a feedforward artificial neural network
  - The term feedforward refers to the fact that each layer serves as the input to the next layer without loops, in contrast to recurrent neural networks
- Note that the artificial 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

Forward Propagation

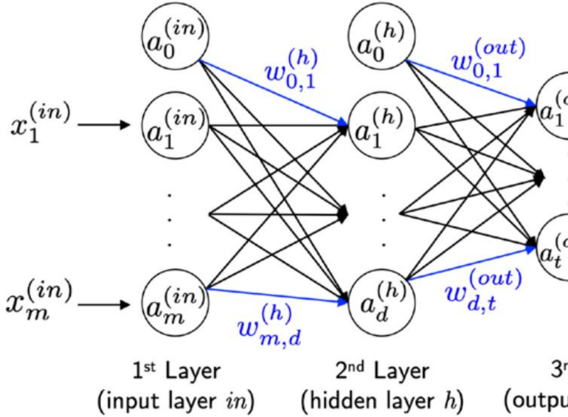
$$z_1^{(h)} = w_{0,1}^{(h)} + a_1^{(in)} w_{1,1}^{(h)} + \dots + a_m^{(in)} w_{m,1}^{(h)}$$

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

- 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

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

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

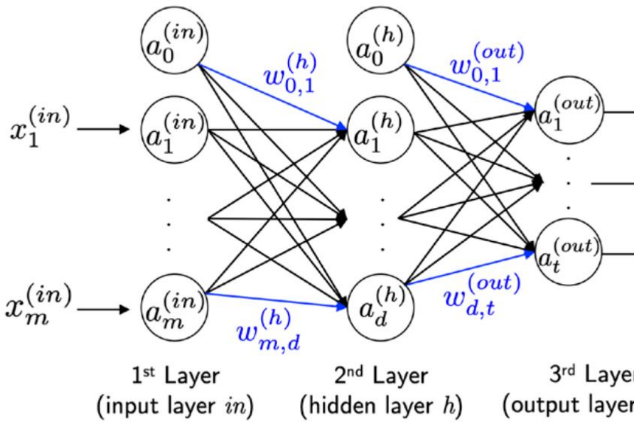


- $\boldsymbol{a}^{(in)}$ is our $1 \times (m+1)$ dimensional feature vector of a sample $\boldsymbol{x}^{(in)}$ and a bias unit
- $\mathbf{W}^{(h)}$ is an $(m+1) \times d$ dimensional weight matrix where d is the number of units in the hidden layer
- After matrix-vector multiplication, we obtain the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

Forward Propagation (all samples)

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

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



- $\mathbf{A}^{(\mathrm{in})}$ is a $n \times (m+1)$ matrix, and the matrix-matrix multiplication will result in an $n \times d$ dimensional net input matrix $\mathbf{Z}^{(\mathrm{h})}$
- Finally, we apply the activation function ϕ to each value in the net input matrix to get the $n \times (d+1)$ activation matrix $\mathbf{A}^{(\mathrm{h})}$ for the next layer

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

Forward Propagation (all samples)

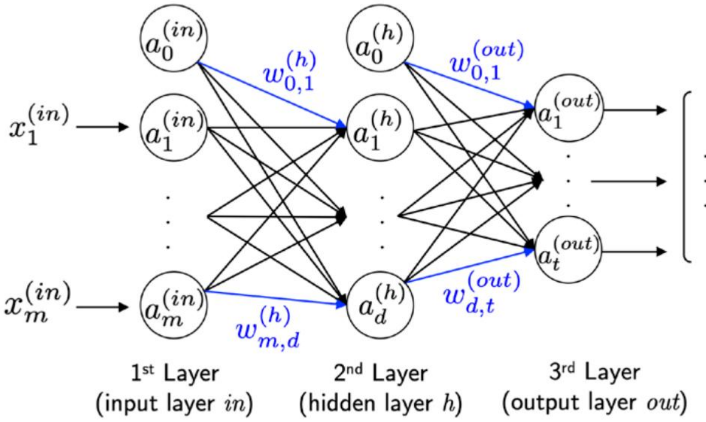
- We can also write the activation of the output layer in vectorized form

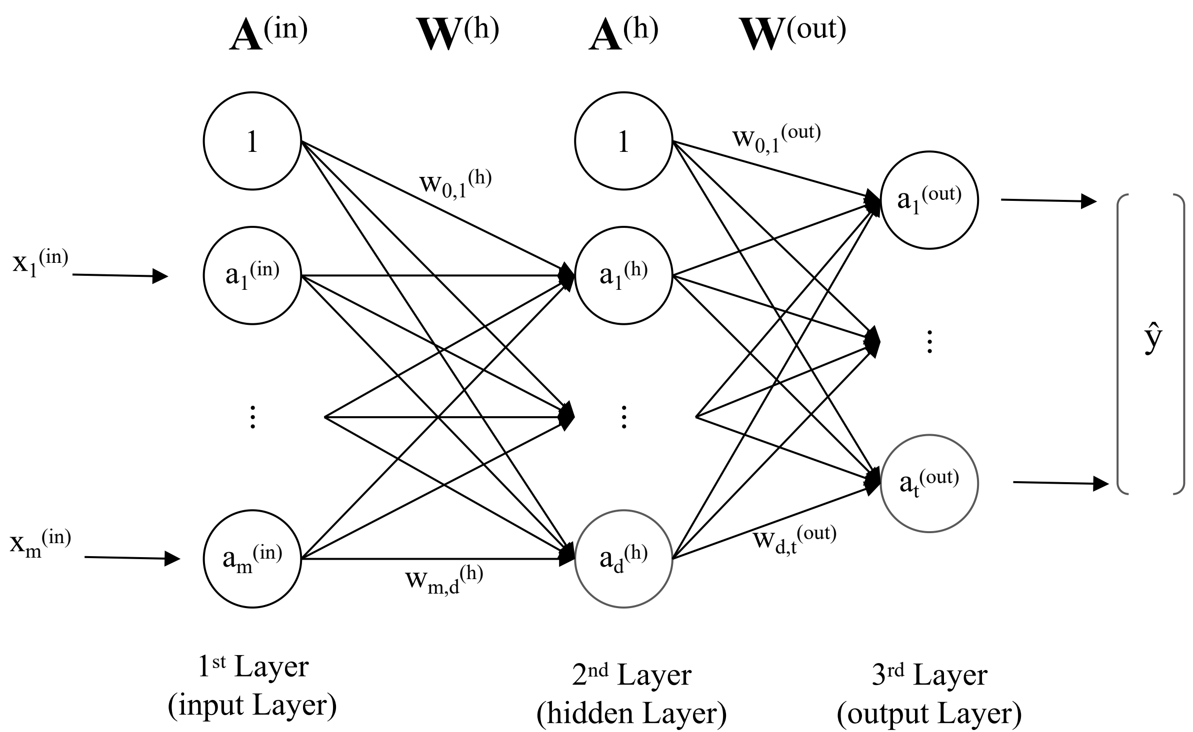
$$\mathbf{Z}^{(\text{out})} = \mathbf{A}^{(\text{h})}\mathbf{W}^{(\text{out})}$$

- Here, we multiply the $(d+1) \times t$ matrix $\mathbf{W}^{(\text{out})}$ (t is the number of output units) with the $n \times (d+1)$ dimensional matrix $\mathbf{A}^{(\text{h})}$ 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

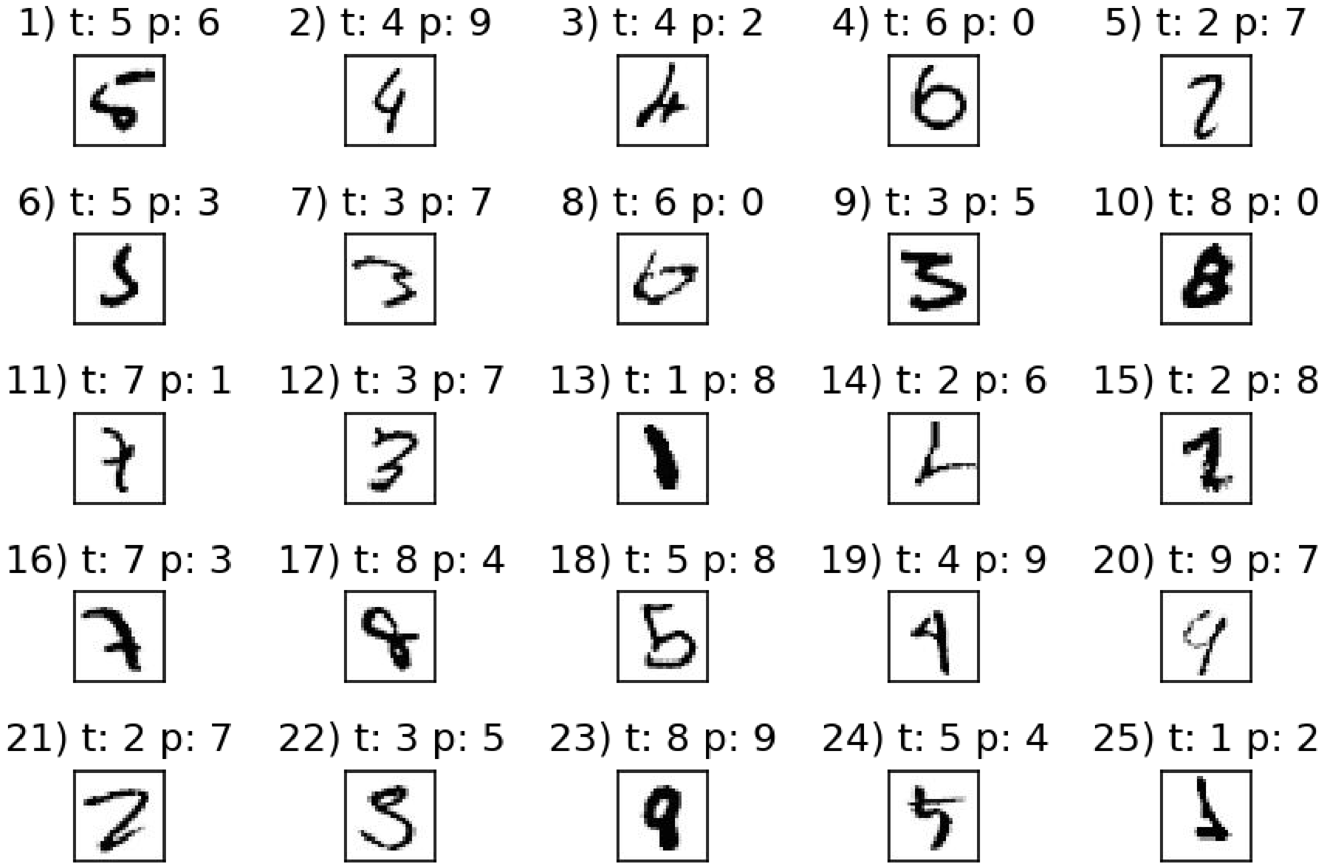
$$\underset{n \times m}{\mathbf{A}^{(in)}}\,\underset{m \times d}{\mathbf{W}^{(h)}} = \underset{n \times d}{\mathbf{Z}^{(h)}} \longrightarrow \underset{n \times d}{\phi(\mathbf{Z}^{(h)})} = \underset{n \times d}{\mathbf{A}^{(h)}} \longrightarrow \underset{n \times d}{\mathbf{A}^{(h)}}\,\underset{d \times t}{\mathbf{W}^{(out)}} = \underset{n \times t}{\mathbf{Z}^{(out)}} \longrightarrow \underset{n \times t}{\phi(\mathbf{Z}^{(out)})} = \underset{n \times t}{\mathbf{A}^{(out)}}$$

Example: Classifying handwritten digits

- We can train a 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, and 50 percent employees from the Census Bureau
  - The test set contains handwritten digits from people not in the training set


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

MNIST dataset

- Obtaining the MNIST dataset
  - The MNIST 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)

Training Details - Compute Cost

- Let's dig a little bit deeper into some of the concepts, such as the cost function and the gradient descent algorithm that we need to learn the weights
- The cost function is the same cost function that we described in the logistic regression section

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

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

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

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 (remember that we don't regularize the bias units)

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

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

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

Training Details - Compute Cost

- Since our MLP for multiclass classification returns an output vector of t elements, we need to compare it to the t x 1 dimensional target vector in the one-hot encoding representation, for example, the activation of the third layer and the target class (here, class 2) 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

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

Training Details - Compute Cost

- The generalized regularization term just calculates the sum of all weights of all *L* layers (without the bias term)

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

- 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(**W**)
  - Thus we need to calculate the partial derivative of J(**W**) with respect to every weight of every layer in the network

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

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

- Note: In the following we assume that you are familiar with calculus
  - You can find a refresher on function derivatives, partial derivatives, gradients, and the Jacobian here

Backpropagation - Big Picture

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

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

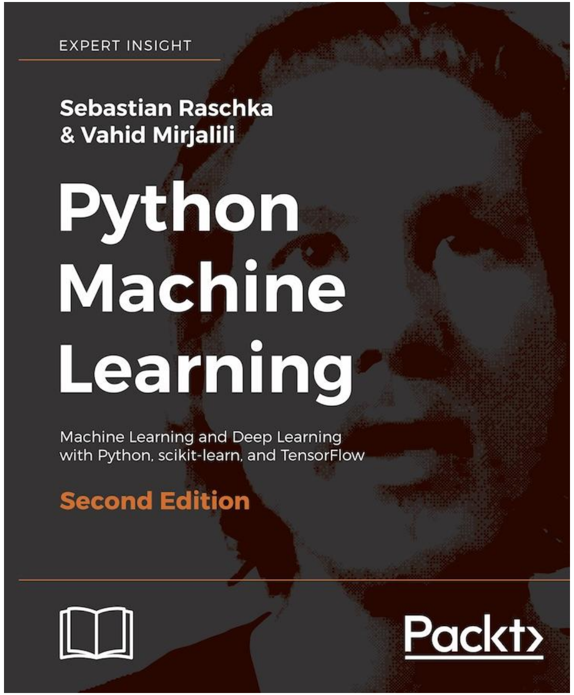
  - We can use the chain rule for an arbitrarily long function composition

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

  - 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

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.