Week 4: Perceptron and Neural Network

Yunwen Lei

School of Computer Science, University of Birmingham

1 Perceptron

We first consider binary classification problems. In this case, we have a dataset of n input/output pairs

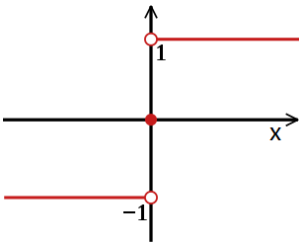
$$S = \{(\mathbf{x}^{(1)}, y^{(1)}), \ldots, (\mathbf{x}^{(n)}, y^{(n)})\}, \quad y_i \in \{+1, -1\}$$

- Training examples with $y = +1$ are called positive examples
- Training examples with $y = -1$ are called negative examples

Aim: find a classification rule $\mathbf{x} \mapsto \{-1, +1\}$

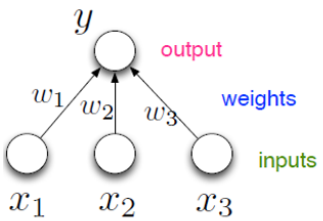
- we always find first a function $f : \mathcal{X} \mapsto \mathbb{R}$ and do the prediction by $\text{sgn}(f(x))$. That is, we predicate it to be $+1$ if the output $f(x)$ is positive, and -1 otherwise.

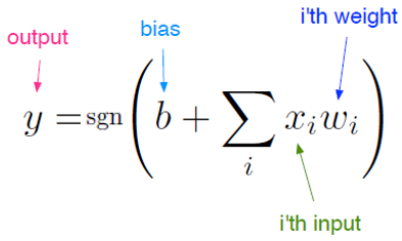
$$\text{sgn}(x) = \begin{cases} 1, & \text{if } x > 0 \\ 0, & \text{if } x = 0 \\ -1, & \text{if } x < 0. \end{cases}$$



1.1 Classification by Perceptron

Definition 1. A perceptron outputs the sign of a linear function: $\text{sgn}(\mathbf{w}^\top \mathbf{x} + b)$.





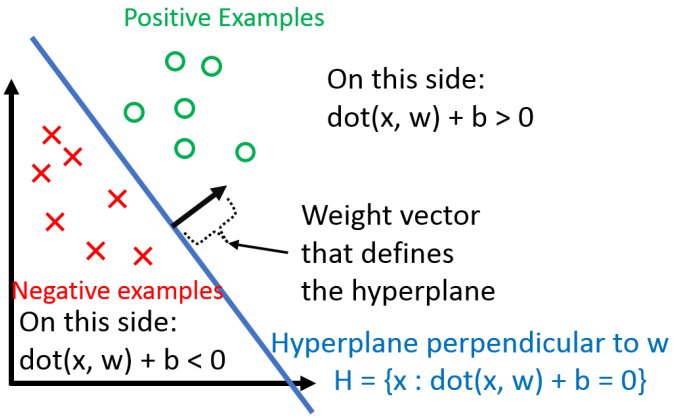
From the definition, we know that a perceptron is the composition of a sign function and a linear function. For the linear function, we have bias and weight parameters w. The use of sign function is necessary since in classification the output is either -1 or 1. We see a difference between linear regression and classification. For the linear regression, we want the linear function to approximate the output. For the classification, we want the sign of the linear function to approximation the output.

- Training: compute $(\mathbf{w}, b)$ such that $\text{sgn}(\mathbf{w}^\top \mathbf{x}^{(i)} + b) \approx y^{(i)}$ for all i

- Prediction: given a testing example $\mathbf{x}$, predict the output as

$$\text{sgn}(\mathbf{w}^\top \mathbf{x} + b)$$

Since a perceptron involves a linear function, the classifier built by a perceptron corresponds to finding a linear hyperplane to separate positive examples from negative examples. A hyperplane $\{H : \mathbf{w}^\top \mathbf{x} + b = 0\}$ consists of points perpendicular to $\mathbf{w}$. From this expression, a hyperplane corresponds to a linear model. We predict a example to be positive and negative depending on whether $\mathbf{w}^\top x + b$ is positive or negative. Geometrically speaking, positive and negative examples (predicted by a hyperplane) are on different sides of this hyperplane.



1.2 Perceptron Algorithm

An efficient algorithm to build a perceptron is the following Perceptron algorithm.

1: Initialize $\mathbf{w} = 0$ ▷ we assume no bias for simplicity
2: **while** All training examples are not correctly classified **do**
3: **for** $(\mathbf{x}, y) \in S$ **do** ▷ Loop over each (feature, label) pair in the dataset
4: **if** $y \cdot \mathbf{w}^\top \mathbf{x} \leq 0$ **then** ▷ If the pair $(\mathbf{x}, y)$ is misclassified
5: $\mathbf{w} \leftarrow \mathbf{w} + y\mathbf{x}$ ▷ Update the weight vector $\mathbf{w}$

Perceptron algorithm is an iterative algorithm. We sequentially update the model up on the observation of each example. For simplicity we do not consider the bias here, which can be addressed by adding a feature of 1.

- If $y \cdot \mathbf{w}^\top \mathbf{x} > 0$, this means that the true output y and the predicted output $\mathbf{w}^\top \mathbf{x}$ has the same sign. In this case, the perceptron already predicts correctly on this example. Therefore, there is no need to update the model for this example $(\mathbf{x}, y)$.

- It updates the model if encountering a misclassification on $(\mathbf{x}, y)$

$$y\mathbf{w}^\top \mathbf{x} \leq 0 \iff y \neq \text{sgn}(\mathbf{w}^\top \mathbf{x})$$

Update $\mathbf{w}$ so that after update it is more likely to predict correctly on $(\mathbf{x}, y)$

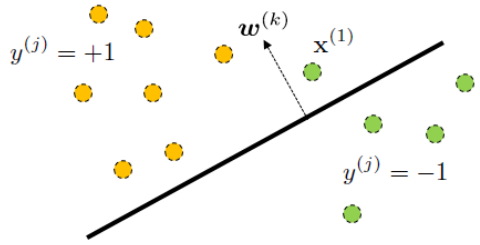
$$y\big(\underbrace{\mathbf{w} + y\mathbf{x}}_{\text{after update}}\big)^\top \mathbf{x} = y\mathbf{w}^\top \mathbf{x} + \underbrace{y^2 \mathbf{x}^\top \mathbf{x}}_{>0} > y \underbrace{\mathbf{w}^\top}_{\text{before update}} \mathbf{x}.$$

We know that a model $\mathbf{w}$ makes a correct prediction if $y\mathbf{w}^\top \mathbf{x} > 0$. We call the term $y\mathbf{w}^\top \mathbf{x}$ the margin of the model $\mathbf{w}$ on the example $(\mathbf{x}, y)$. The above computation shows that within one update the new model $\mathbf{w} + y\mathbf{x}$ would have a larger margin as compared to the previous model $\mathbf{w}$ on the example $(\mathbf{x}, y)$. This explains why the new model is more likely to predict correctly on $(\mathbf{x}, y)$.

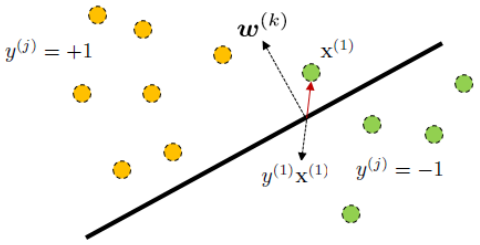
The performance of Perceptron algorithm is guaranteed by the following theorem.

Theorem 1. *If the dataset is linearly separable (meaning there is a hyperplane which can separate positive and negative examples with no errors), then Perceptron algorithm would always stop and find such a hyperplane.*

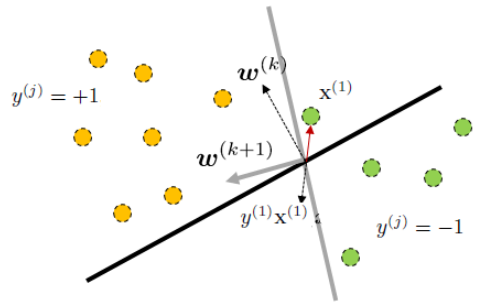
Example 1. Here we visualize the process of implementing the perceptron algorithm.



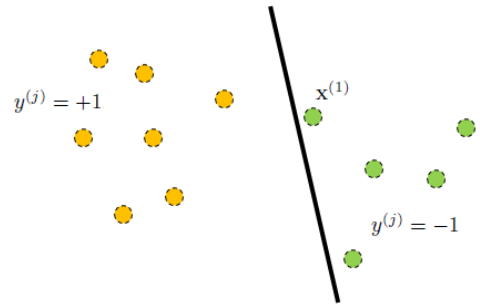
- Initially there is a $\mathbf{w}^{(k)}$
- There is a misclassified training example $\mathbf{x}^{(1)}$



- Perceptron algorithm finds $y^{(1)}\mathbf{x}^{(1)}$
- $y^{(1)}\mathbf{x}^{(1)}$ is in the opposite direction of $\mathbf{x}^{(1)}$. The perceptron makes a wrong prediction on this example and we need to update $\mathbf{w}^{(k)}$.



- $\mathbf{w}^{(k+1)}$ is a linear combination of $\mathbf{w}^{(k)}$ and $y^{(1)}\mathbf{x}^{(1)}$ (note $y^{(1)} = -1$).
- $\mathbf{w}^{(k+1)}$ gives a new separating hyperplane which is perpendicular to it.



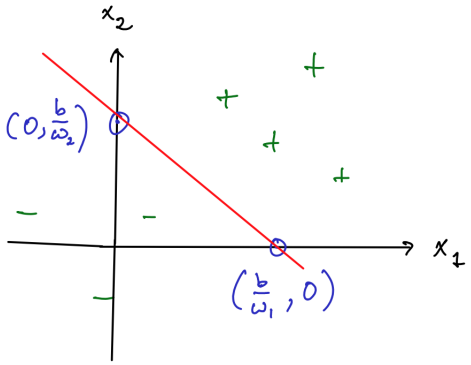
- Now you are happy because al the examples are classified correctly!

2 Single-Layer Neural Network

A limitation of Perceptron is that it only constructs a linear classifier. Indeed, it classifies an example according to whether the example is on the one-side of the hyperplane or the other.

Example 2. In two dimensions, we have

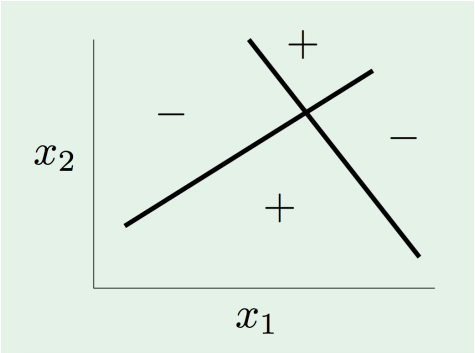
$$f(x_1, x_2) = \text{sgn}(\omega_1 x_1 + \omega_2 x_2 - b)$$



$$\omega_1 x_1 + \omega_2 x_2 - b > 0 \iff x_2 \begin{cases} > \frac{b-\omega_1 x_1}{\omega_2}, & \text{if } w_2 > 0 \\ \leq \frac{b-\omega_1 x_1}{\omega_2}, & \text{otherwise.} \end{cases}$$

Here are some further drawbacks of perceptrons.

- While the Perceptron algorithm can stop if the data are linear separable, it does not stop if the data are linearly non-separable (there is no hyperplane that can perfectly separate positive examples from negative examples)
- Weights **w** are adjusted for misclassified data only (correctly classified data are not considered at all)
- Multiple perceptron can form complex decision boundaries (piecewise-linear), but it is hard to train. For example, with the following two perceptrons, we partition the space into four regions which can separate the positive examples from negative examples.



These drawbacks motivate the following question

How can we resolve the problem of training?

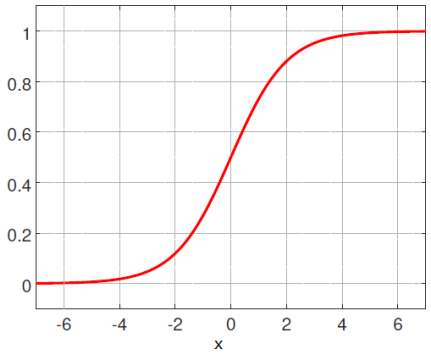
2.1 Single-Layer Neural Networks

In this subsection, we consider a method to improve perceptrons. The idea is to replace the sgn function with a differentiable non-linear function.

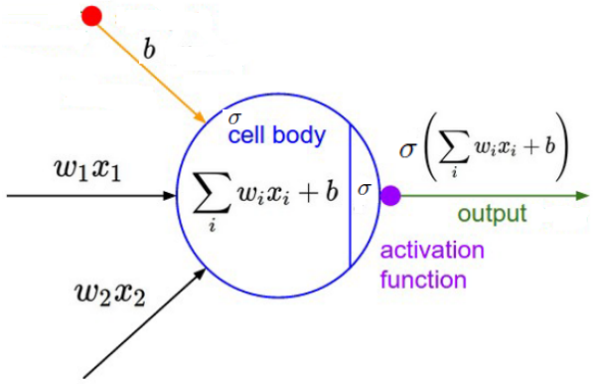
- A popular choice is the sigmoid function

$$\sigma(x) = \frac{1}{1 + \exp(-x)}$$

- Mapping: $(-\infty, +\infty) \mapsto (0, 1)$
- $\sigma'(x) = \sigma(x)(1 - \sigma(x))$ (exercise)



This leads to a single-layer neural network with the following prediction rule



$$f(\mathbf{x}) = \sigma\Big(\sum_i w_i x_i + b\Big)$$

The only difference of single-layer neural network from perceptron is that it uses a highly nonlinear sigmoid function to replace the sign function in the perceptron. Since the sigmoid function can output any real number, the single-layer neural network outputs a real-valued number. This means that the single-layer neural network can be applied to regression problems. Furthermore, sign function is not continuous while sigmoid function is differentiable.

Example 3 (Training Single-Layer Neural Network)**.** We can apply gradient descent to train single-layer neural networks.

- Dataset: n input/output pairs $S = \{(\mathbf{x}^{(1)}, y^{(1)}), \ldots, (\mathbf{x}^{(n)}, y^{(n)})\}$
- Mean-Square Error
$$C(\mathbf{w}) = \frac{1}{2n}\sum_{i=1}^{n}\Big(\underbrace{\sigma(\mathbf{w}^\top \mathbf{x}^{(i)} + b)}_{\text{predicted output}} - \underbrace{y^{(i)}}_{\text{output}}\Big)^2.$$
- Unlike the linear regression problems, we do not have closed form solution for the minimizer of $C(\mathbf{w})$
- Use Gradient Descent to train a $\mathbf{w} \in \mathbb{R}^d$ with a small $C(\mathbf{w})$
$$\mathbf{w}^{\text{new}} = \mathbf{w} - \eta\nabla C(\mathbf{w}) \quad \Longleftrightarrow w_i^{\text{new}} = w_i - \eta\frac{\partial C(\mathbf{w})}{\partial w_i}.$$
- We need to find a way to compute gradient and more specifically $\frac{\partial(\sigma(\mathbf{w}^\top \mathbf{x}+b)-y)^2}{\partial w_i}$
- It suffices to know how to compute the derivative of a univariate function $\frac{d(\sigma(wx+b)-y)^2}{dw}$ since (if we want to compute the partial derivative w.r.t. w_i we can fix other variables and get a univariate function of w_i)
$$(\sigma(\mathbf{w}^\top \mathbf{x} + b) - y)^2 = \Big(\sigma\big(w_i x_i + \underbrace{\sum_{j:j\neq i} w_j x_j + b}_{:=\bar{b} \text{ is a const w.r.t } w_i}\big) - y\Big)^2.$$
This univariate function is a composition of a sigmoid function and a linear function. We can use chain rule to compute the derivative.

3 Chain Rule and Gradient Descent for Single-Layer Neural Network

3.1 Univariate Chain Rule

Definition 2. The composition of two functions f and g is defined by
$$(f \circ g)(x) = f(g(x)).$$

For example, if $f(x) = x^3$ and $g(x) = \sin x$, then

$$f \circ g(x) = f(g(x)) = f(\sin x) = \sin^3(x)$$
$$g \circ f(x) = g(f(x)) = g(x^3) = \sin(x^3).$$

Chain rule shows how to compute the derivative of a composite function. We first consider the one-dimensional case.

Theorem 2 (Chain Rule: One Dimensional Case)**.** *For one dimensional differentiable functions f and g, we have*

$$\frac{d}{dx} f(g(x)) = f'(g(x)) \cdot g'(x)$$

In words, the derivative of $f(g(x))$ is the derivative of the outside function f evaluated at the inside function g, times the derivative of the inside function. Below we give an example to show how to apply chain rule.

Example 4 (Univariate Chain Rule)**.** Let us consider the following function C (σ is the sigmoid function)

$$C = \frac{1}{2}\big(\sigma(wx + b) - y\big)^2, \quad w, b \in \mathbb{R}.$$

To apply the chain rule, we first need to represent C by composite functions:

We can write C by composite functions

$$C = \frac{1}{2}p^2$$
$$p = \sigma(q) - y$$
$$q = wx + b$$

$$\begin{aligned}\frac{\partial C}{\partial w} &= \frac{\partial C}{\partial p}\frac{\partial p}{\partial w} = p\frac{\partial p}{\partial q}\frac{\partial q}{\partial w} \\ &= p\sigma'(q)x = \big(\sigma(q) - y\big)\sigma'(q)x \\ &= \big(\sigma(wx + b) - y\big)\sigma'(wx + b)x\end{aligned}$$

Similarly we can show

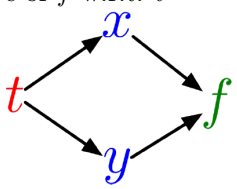
$$\frac{\partial C}{\partial b} = \big(\sigma(wx + b) - y\big)\sigma'(wx + b)$$

Here $\sigma'(x) = \sigma(x)(1 - \sigma(x))$ for sigmoid activation function

3.2 Multivariate Chain Rule

Now we consider multivariate chain rule. Suppose we have a function $f(x, y)$ and functions $x(t)$ and $y(t)$. Then we can use multivariate chain rule to compute the derivative of f w.r.t. t

$$\frac{d}{dt} f(x(t), y(t)) = \frac{\partial f}{\partial x}\frac{dx}{dt} + \frac{\partial f}{\partial y}\frac{dy}{dt}$$



According to the definition, df/dt means the change rate of f w.r.t. t. By the above figure, there are two ways that t can affect f. t can affect x first, and then x will affect f. t can also affect y first and then y will affect f. The multivariate chain rule tells us that we need to consider both routs together. In each route, we take a multiplication of partial derivative along the route. We then take a summation over all the routes to get the final derivative.

Example 5. Now we give a specific example to show this process.

$$f(x, y) = y + e^{xy}$$
$$x(t) = \cos t$$
$$y(t) = t^2$$

$$\begin{aligned}\frac{df}{dt} &= \frac{\partial f}{\partial x}\frac{dx}{dt} + \frac{\partial f}{\partial y}\frac{dy}{dt} \\ &= (ye^{xy}) \cdot (-\sin t) + (1 + xe^{xy}) \cdot 2t.\end{aligned}$$

The following theorem is the chain rule in the general case.

Theorem 3 (Chain Rule: Multivariate Case)**.** *For $f(u_1, \ldots, u_m)$ with $u_i = g_i(x_1, \ldots, x_n), i = 1, \ldots, m$, then*

$$\frac{\partial f}{\partial x_i} = \sum_{j=1}^{m} \frac{\partial f}{\partial u_j} \cdot \frac{\partial u_j}{\partial x_i}.$$

Now we apply the chain rule to compute the gradient for single-layer neural networks.

- Let $C_i(\mathbf{w}) = \frac{1}{2}(\sigma(\mathbf{w}^\top \mathbf{x}^{(i)} + b) - y^{(i)})^2$. Then $C(\mathbf{w}) = \frac{1}{n}\sum_{i=1}^{n} C_i(\mathbf{w})$
- According to Example 4, we know

$$\frac{\partial C_i}{\partial w_j} = \big(\sigma(\mathbf{w}^\top \mathbf{x}^{(i)} + b) - y^{(i)}\big)\sigma'(\mathbf{w}^\top \mathbf{x}^{(i)} + b)x_j^{(i)} \tag{3.1}$$

$$\frac{\partial C_i}{\partial b} = \big(\sigma(\mathbf{w}^\top \mathbf{x}^{(i)} + b) - y^{(i)}\big)\sigma'(\mathbf{w}^\top \mathbf{x}^{(i)} + b). \tag{3.2}$$

Vectorization of (3.1): $$\frac{\partial C_i}{\partial \mathbf{w}} = \big(\sigma(\mathbf{w}^\top \mathbf{x}^{(i)} + b) - y^{(i)}\big)\sigma'(\mathbf{w}^\top \mathbf{x}^{(i)} + b)\mathbf{x}^{(i)}. \tag{3.3}$$

We can plug the above gradient computation into the framework of gradient descent and get the following algorithm for training single-layer neural networks.

1: Initialize $\mathbf{w}^{(1)} = 0, b^{(1)} = 0$
2: **for** $t = 1, 2, \ldots, T$ **do** ▷ T is the number of iterations
3: Use (3.3), (3.2) to compute gradients

$$\nabla_{\mathbf{w}} = \frac{1}{n}\sum_{i=1}^{n} \frac{\partial C_i(\mathbf{w}^{(t)})}{\partial \mathbf{w}^{(t)}}, \qquad \nabla_b = \frac{1}{n}\sum_{i=1}^{n} \frac{\partial C_i(\mathbf{w}^{(t)})}{\partial b^{(t)}}$$

4: Update the model

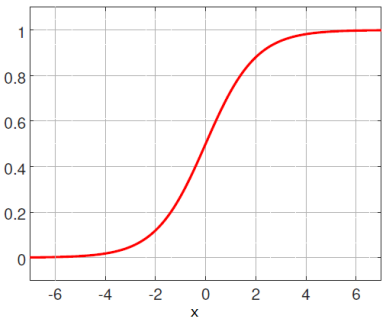
$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta_t \nabla_{\mathbf{w}}, \qquad b^{(t+1)} = b^{(t)} - \eta_t \nabla_b.$$

Gradient Descent for Single-Layer NN

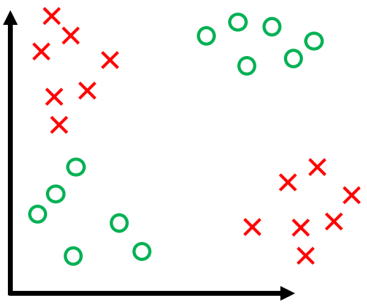
Remark 1 (Single-layer Neural Network still yields linear classifiers)**.** Single-layer neural networks can be used for regression since it outputs a real number. We can also use it to do classification. We can still use the sign of $\mathbf{w}^\top \mathbf{x} + b$ for prediction. Since $\sigma(0) = 1/2$ we can predict it to be positive if $f(\mathbf{x}) = \sigma(\mathbf{w}^\top \mathbf{x} + b) \geq 1/2$. This is illustrated as follows

$$\hat{y} = 1 \text{ iff } f(\mathbf{x}) \geq 1/2$$
$$f(\mathbf{x}) \geq 1/2 \iff \mathbf{w}^\top \mathbf{x} + b \geq 0$$
$$\hat{y} = 1 \text{ iff } \mathbf{w}^\top \mathbf{x} + b \geq 0$$

This is a linear classifier!



Suppose we have a dataset which is nonlinearly separable. How to build a classifier?
The idea is to use multiple neurons!

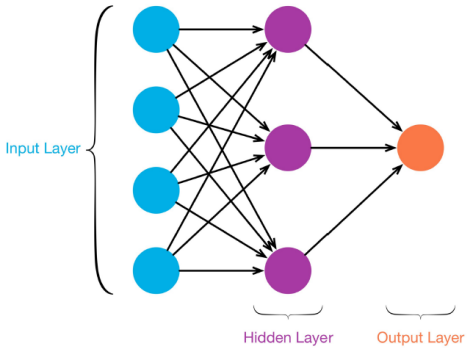


4 Feed-Forward Neural Networks

Feed-forward neural network is a very powerful model to address nonlinear data. In this section, we will give the structure of this network and show how to use it to do prediction. In the next week, we will show how to train a neural network from the training examples.

The basic idea of feed-forward neural networks is to use several neurons. Between neurons there are some edges with direction.

- We can connect lots of units/neurons together into a directed acyclic graph, meaning that if you follow the direction indicated by the edge, you will never return to the neuron you have visited.
- Typically, units are grouped together into layers. We have an input layer, some hidden layers and an output layer. Each neuron in the input layer receives a feature of the examples. These features are then passed to the hidden layers via edges. The output layer predicts the output.
- Each unit (simplest case) in a layer is connected to each unit in the next layer. This is a fully connected neural network. We call it Feed-Forward neural network or multi-layer perceptron (MLP).
- The number of units in the input layer is equal to the number d of features. Output layer can have one neuron or more neuron, depending on the learning task.
- Right gives an example of MLP with 3 layers: an input layer with 4 neurons, a hidden layer with 3 neurons and an output layer with 1 neuron.



By introducing more neurons structured into several layers, we get a highly nonlinear model with improved expression power!

4.1 Weights and Bias

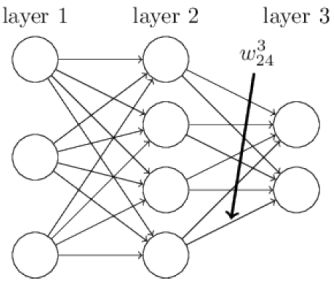
We use edge to connect different neurons. Each edge is associated with a number called weight. Each neuron is associated with a number called bias. Both weights and bias are called trainable parameters. We need to train these parameters from the dataset, and once these parameters are fixed we get a model.

Since there are many weights, we need to distinguish these weights by introducing 2 sub-indices and 1 super-index

- ω_{24}^{3}: from 4-th neuron in 2nd layer to 2-nd neuron in 3-rd layer

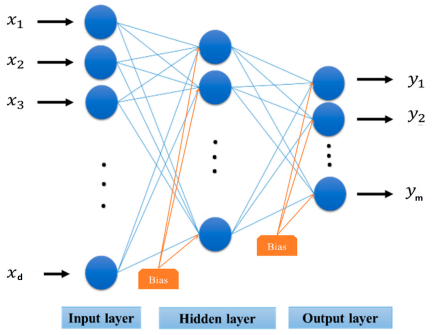
Each node is associated with a number called bias. We introduce a subindex and a super-index to distinguish different bias.

- b_3^2: the 3-rd node in the 2-nd layer



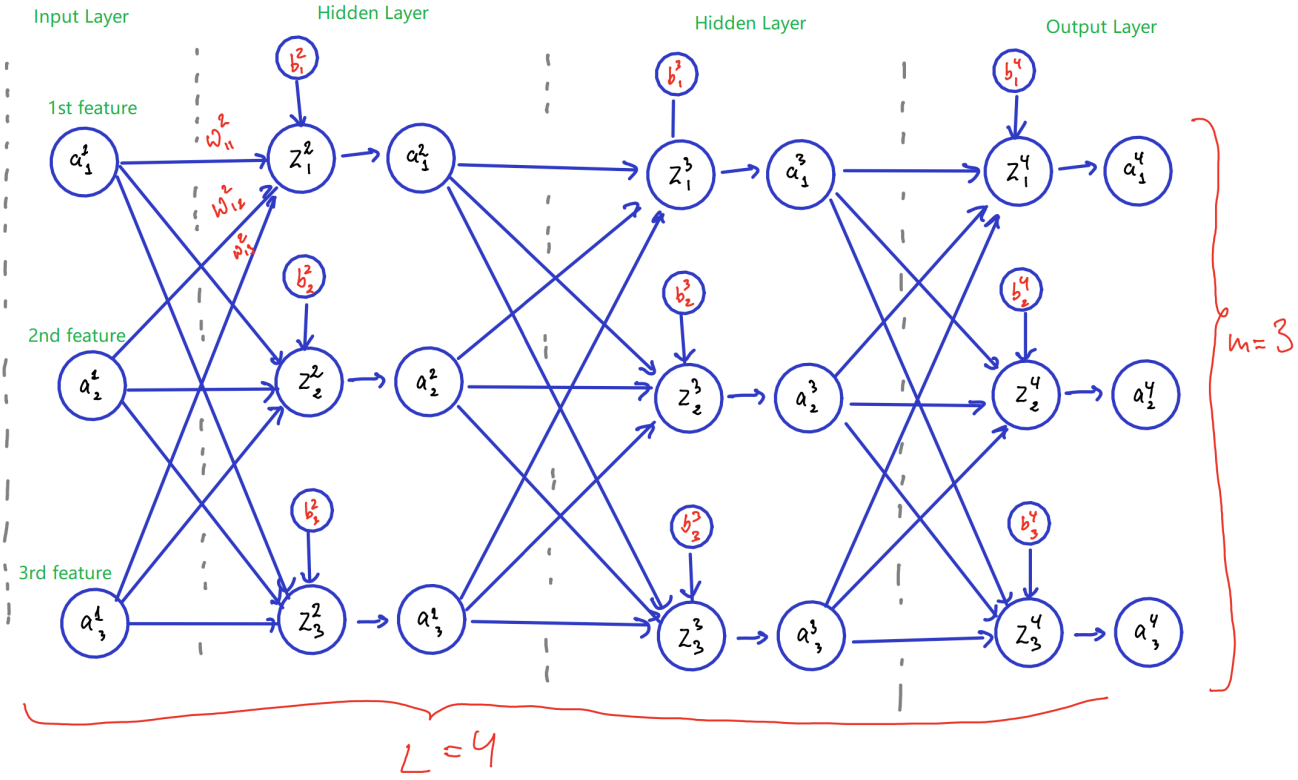
w^l_{jk} is the weight from the k^{th} neuron in the $(l-1)^{\text{th}}$ layer to the j^{th} neuron in the l^{th} layer

weights



bias

Each neuron can be considered as a processor. It receives some input information from previous layers (or input feature), does some processing and then outputs it to neuron in the next layer. We need to introduce some notations to explain this:



- L: number of layers (1-st layer is "input layer", L-th layer is "output layer")
- m: "width" of network (can vary between layers)
- ω^ℓ_{jk}: "weight" of connection between k-th unit in layer $\ell-1$, to j-th unit in layer ℓ
- b^ℓ_j: "bias" of j-th unit in layer ℓ

Let us consider the j-th unit in the ℓ-th layer. We associate two numbers: z^ℓ_j and a^ℓ_j.

- This neuron receives all the output from the previous layer $a^{\ell-1}_k, k = 1, 2, \ldots, m$ and applies a linear function

$$z^\ell_j = \sum_k \omega^\ell_{jk} a^{\ell-1}_k + b^\ell_j. \tag{4.1}$$

The weights of this linear function are ω^ℓ_{jk}, which connects the k-th neuron in the layer $\ell - 1$ to the j-th neuron in the layer ℓ.

- Until now, we get a linear function. To achieve nonlinearity, we apply the activation function to z_j^ℓ and get "activation"

$$a_j^\ell = \sigma(z_j^\ell). \tag{4.2}$$

In summary, each neuron in MLP receives activations from all the neurons in the previous layer, and outputs activations which are passed to all the neurons in the next layer.

4.2 Vectorization

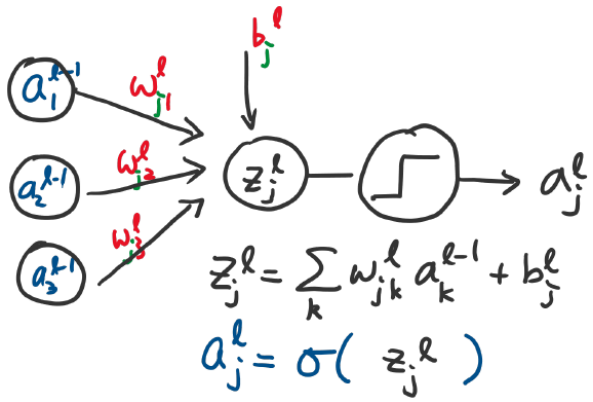
While the equations (4.1), (4.2) precisely describe the information flow of MLP, they are not convenient to implement in python. In python, we wish a vectorization of implementation, i.e., represent the computation in terms of matrices. To this aim, we collect weights into several matrices, and bias into several vectors (trainable parameters)

$$\mathbf{W}^\ell = \begin{pmatrix} \omega_{11}^\ell & \omega_{12}^\ell & \cdots & \omega_{1m}^\ell \\ \omega_{21}^\ell & \omega_{22}^\ell & \cdots & \omega_{2m}^\ell \\ \vdots & \vdots & \ddots & \vdots \\ \omega_{m1}^\ell & \omega_{m2}^\ell & \cdots & \omega_{mm}^\ell \end{pmatrix}, \quad \mathbf{b}^\ell = \begin{pmatrix} b_1^\ell \\ b_2^\ell \\ \vdots \\ b_m^\ell \end{pmatrix}, \qquad \ell = 2, \ldots, L.$$

Each matrix $\mathbf{W}^\ell$ has m^2 parameters, and each $\mathbf{b}^\ell$ has m parameters. Therefore, the number of trainable parameters in each layer is m^2+m. This leads to a total number of $(m^2+m)(L-1)$ trainable parameters (for simplicity, we assume the number of neurons is the same for all layers. Different layers can have different number of neurons and in this case the total number of trainable parameters can be computed similarly). The i-th row of the matrix contains the weights related to the i-th neuron in the layer ℓ, the j-th column contains the weights related to the j-th neuron in the layer $\ell-1$.

Recall the main equation

$$a_j^\ell = \sigma\Big(\underbrace{\sum_k \omega_{jk}^\ell a_k^{\ell-1} + b_j^\ell}_{z_j^\ell}\Big) \quad \text{or} \quad a_j^\ell = \sigma(z_j^\ell)$$



We can relate $(a_1^{\ell-1}, \ldots, a_m^{\ell-1})$ to $(z_1^\ell, \ldots, z_m^\ell)$ in terms of matrix

$$\begin{pmatrix} z_1^\ell = \sum_k \omega_{1k}^\ell a_k^{\ell-1} + b_1^\ell \\ z_2^\ell = \sum_k \omega_{2k}^\ell a_k^{\ell-1} + b_2^\ell \\ \vdots \\ z_m^\ell = \sum_k \omega_{mk}^\ell a_k^{\ell-1} + b_m^\ell \end{pmatrix} = \begin{pmatrix} \omega_{11}^\ell & \omega_{12}^\ell & \cdots & \omega_{1m}^\ell \\ \omega_{21}^\ell & \omega_{22}^\ell & \cdots & \omega_{2m}^\ell \\ \vdots & \vdots & \vdots & \vdots \\ \omega_{m1}^\ell & \omega_{m2}^\ell & \cdots & \omega_{mm}^\ell \end{pmatrix} \begin{pmatrix} a_1^{\ell-1} \\ a_2^{\ell-1} \\ \vdots \\ a_m^{\ell-1} \end{pmatrix} + \begin{pmatrix} b_1^\ell \\ b_2^\ell \\ \vdots \\ b_m^\ell \end{pmatrix}$$

This amounts to saying

$$\mathbf{z}^\ell = \mathbf{W}^\ell \mathbf{a}^{\ell-1} + \mathbf{b}^\ell$$
$$\mathbf{a}^\ell = \sigma(\mathbf{z}^\ell)$$
$$\ell = 2, \ldots, L.$$

$$\mathbf{W}^\ell := \begin{pmatrix} \omega_{11}^\ell & \omega_{12}^\ell & \cdots & \omega_{1m}^\ell \\ \omega_{21}^\ell & \omega_{22}^\ell & \cdots & \omega_{2m}^\ell \\ \vdots & \vdots & \vdots & \vdots \\ \omega_{m1}^\ell & \omega_{m2}^\ell & \cdots & \omega_{mm}^\ell \end{pmatrix}$$

$$\mathbf{z}^\ell := \begin{pmatrix} z_1^\ell \\ z_2^\ell \\ \vdots \\ z_m^\ell \end{pmatrix}, \quad \mathbf{a}^\ell := \begin{pmatrix} a_1^\ell \\ a_2^\ell \\ \vdots \\ a_m^\ell \end{pmatrix}, \quad \mathbf{b}^\ell := \begin{pmatrix} b_1^\ell \\ b_2^\ell \\ \vdots \\ b_m^\ell \end{pmatrix}$$

In summary, we have

$$\mathbf{a}^\ell = \sigma(\mathbf{z}^\ell) = \sigma\big(\mathbf{W}^\ell \mathbf{a}^{\ell-1} + \mathbf{b}^\ell\big)$$

4.3 Forward Propagation to Compute Prediction

We are now ready to introduce the forward propagation algorithm. This algorithm receives input from the features and outputs the predictions. We go from the input layer with the original features $\mathbf{a}^1$, then apply linear function together with an activation function to derive a new vector $\mathbf{a}^2$. This vector is the input of the third layer and can be considered as a new feature. We continue this process until arrive at the output layer.

Algorithm 1 (Forward Propagation)**. Input:** $\mathbf{x} \in \mathbb{R}^d, \mathbf{W}^\ell, \mathbf{b}^\ell, \ell = 2, \ldots, L$

1:
Output: $\mathbf{a}^L$ ▷ We assign the original feature to the first layer

2: Set $\mathbf{a}^1 = \mathbf{x}$

3: **for** $\ell = 2, \ldots, L$ **do** ▷ from bottom to top

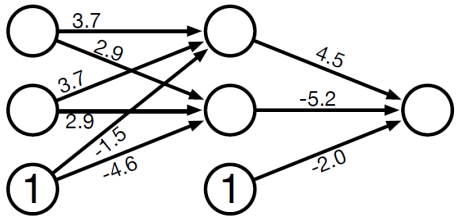
4: Compute the activations in ℓ-th layer via

$$\mathbf{a}^\ell = \sigma\big(\mathbf{W}^\ell \mathbf{a}^{\ell-1} + \mathbf{b}^\ell\big)$$

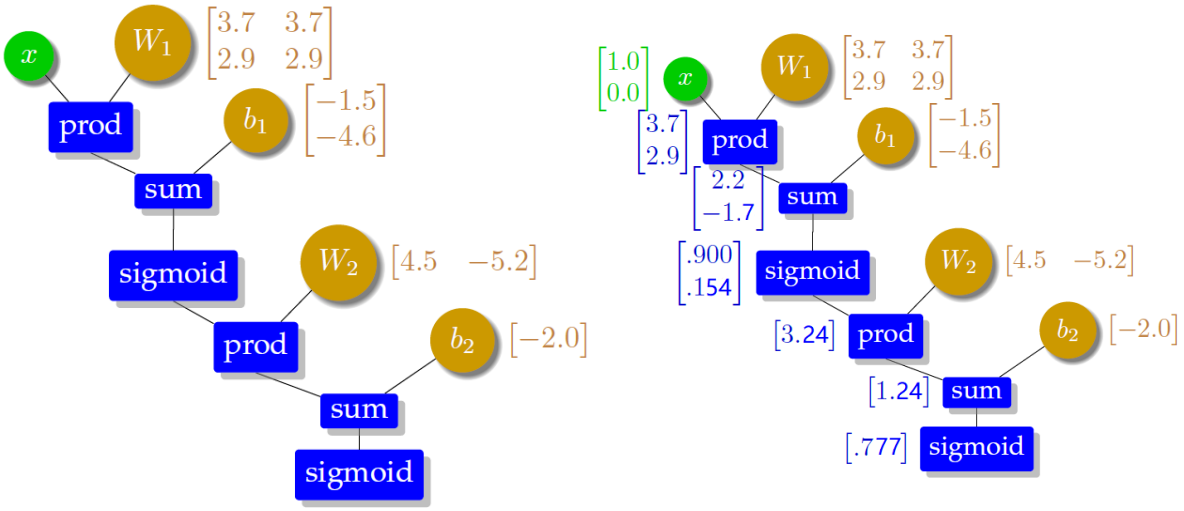
Since $\mathbf{a}^\ell$ are inputs of the layer $\ell + 1$, $\mathbf{a}^\ell$ can be considered as new feature learned from data! This feature is a nonlinear combination of the original features since we have nonlinear activation function.

In this sense, training by neural networks can be interpreted as feature learning!

Example 6 (Forward Propagation)**.** We now consider a neural network with 2 neurons in the input layer, 2 neurons in the hidden layer and 1 neuron in the output layer. The weights and bias are indicated by the following figure. The input feature is $\mathbf{x} = (1, 0)^\top$.

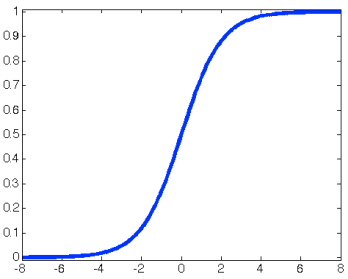


The following figure shows the implementation process of forward propagation. We first collect weights in two matrices $\mathbf{W}^1, \mathbf{W}^2$, and bias in two vectors $\mathbf{b}^1, \mathbf{b}^2$. Then we do the computation sequentially from the bottom to the top.



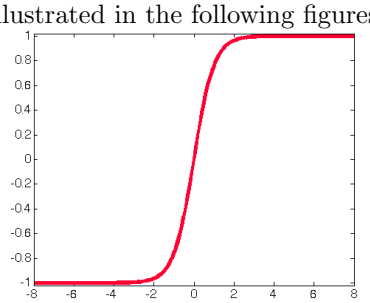
4.4 Continuous Activation Functions

There are several choices of activation functions. We have introduced activation functions in our previous section. In this section we introduce another two activation functions: the ReLU activation and the Tanh activation. These activation functions are essential in constructing nonlinear models by neural networks. Their behavior is illustrated in the following figures.



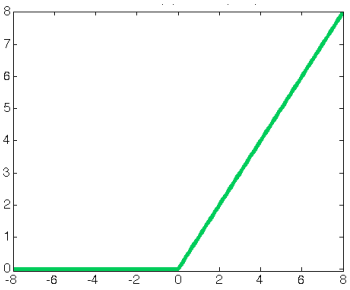
$$\sigma(x) = \frac{1}{1+\exp(-x)}$$

$$(-\infty, \infty) \mapsto (0, 1)$$



$$\mathrm{Tanh}(x) = \frac{\exp(x) - \exp(-x)}{\exp(x) + \exp(-x)}$$

$$(-\infty, \infty) \mapsto (-1, 1)$$



$$\mathrm{ReLU}(x) = \max(0, x)$$

$$(-\infty, \infty) \mapsto (0, \infty)$$

- ReLU is simple to compute since it involves only the maximum operator. This is a continuous function but not differentiable. It can be checked from the definition that ReLU is not differentiable at the zero point. ReLU is widely used in modern neural networks due to its simplicity.
- Sigmoid and Tanh are differentiable. However, these two use exponential function and require more computation (exponential function is more difficult to compute than a maximum operator). We can see the behavior of sigmoid and Tanh are very similar. The difference is the sigmoid function outputs value in $[0, 1]$, while Tanh outputs values in $[-1, 1]$.

4.5 Training MLPs

Now we discuss how to train a MLP from the dataset. As we discussed before, the problem becomes the tuning of weights and bias. We use a loss function to measure the performance of the MLP on an example $(\mathbf{x}, y)$

$$C_{(\mathbf{x},y)}(\mathbf{W}^2, \ldots, \mathbf{W}^L, \mathbf{b}^2, \ldots, \mathbf{b}^L) = \Big(y - \underbrace{\sigma(\mathbf{W}^L(\sigma \cdots \sigma(\underbrace{\mathbf{W}^2\mathbf{x} + \mathbf{b}^2}_{\mathbf{z}^2})) + \mathbf{b}^L)}_{\mathbf{z}^L}\Big)^2,$$

which is the squared difference between the true output y and the predicted output z^L. The objective function we want to minimize becomes

$$C(\mathbf{W}, \mathbf{b}) = \frac{1}{n} \sum_{(\mathbf{x},y)\in S} C_{(\mathbf{x},y)}(\mathbf{W}^2, \ldots, \mathbf{W}^L, \mathbf{b}^2, \ldots, \mathbf{b}^L),$$

where S is the training dataset. We apply gradient descent to solve this problem. This leads to

$$\begin{aligned}
\mathbf{W}^2 &\leftarrow \mathbf{W}^2 - \frac{\eta}{n} \sum_{(\mathbf{x},y)\in S} \frac{\partial C_{(\mathbf{x},y)}(\mathbf{W}^2, \ldots, \mathbf{W}^L, \mathbf{b}^2, \ldots, \mathbf{b}^L)}{\partial \mathbf{W}^2}, \\
\mathbf{W}^3 &\leftarrow \mathbf{W}^3 - \frac{\eta}{n} \sum_{(\mathbf{x},y)\in S} \frac{\partial C_{(\mathbf{x},y)}(\mathbf{W}^2, \ldots, \mathbf{W}^L, \mathbf{b}^2, \ldots, \mathbf{b}^L)}{\partial \mathbf{W}^3}, \\
&\vdots \\
\mathbf{W}^L &\leftarrow \mathbf{W}^L - \frac{\eta}{n} \sum_{(\mathbf{x},y)\in S} \frac{\partial C_{(\mathbf{x},y)}(\mathbf{W}^2, \ldots, \mathbf{W}^L, \mathbf{b}^2, \ldots, \mathbf{b}^L)}{\partial \mathbf{W}^L}.
\end{aligned}$$

That is, we need to apply the weights $\mathbf{W}^2, \ldots, \mathbf{W}^L$ simultaneously. In a similar way, we also need to apply gradient descent to update the bias. The implementation of gradient descent requires to compute the gradients of C for all $\mathbf{W}^\ell, \mathbf{b}^\ell$. This is a bit complicated. Unlike linear regression, we do not have a closed-form solution. Furthermore, the structure of MLPs involve several neurons organized in different layers, which are a bit complicated. There are a lot of activation functions. Brute force is impossible if there are lots of parameters.

To train MLPs, we need an efficient way to compute the gradients. This is achieved by the very famous backpropagation algorithm. The mathematical foundation is chain rule. As we will see, backpropagation uses a clever idea to avoid repeated computation! This is the topic of the next week.