CSE517A Machine Learning **Spring 2020**

Lecture 14: Neural Networks

Instructor: Marion Neumann *Scribe: Jingyu Xin*

Reading: LFD eCH 7.1- 7.3 (Neural Networks)

1 Introduction

Challenges in machine learning and advances in technology are the motivations of "*new*" approaches.[1]

Feature Engineering

In most machine learning algorithms we consider the features to be given. In practice, these features need to be created and *feature engineering* and *feature selection* are important to achieve good performance for machine learning applications. In image classification, we may for example compute *texture*, *shape* and *color features*.

The fact that feature engineering is not an easy problem to tackle leads to the idea of *representation learning*. Approaches to representation learning automatically discover the important features from data.

In (deep) neural networks, the features are automatically learned from the raw data, e.g. the (raw) input image. Hence, deep learning can be seen as representation learning where we learn are multiple levels of features that are then composed together hierarchically to produce the output. Each level represents abstract features that are discovered from the features represented in the previous level.

Smoothness Assumptions

Non-parametric machine learning methods heavily rely on local smoothness assumptions. That means that the targets should be similar for similar input data points. However, those smoothness assumptions need to be explicitly expressed (prior on function/parameters or appropriate kernel(s)). The issue is that this assumption is very difficult to exploit for high-dimensional data (*curse of dimensionality*). One way to combat this curse it to have enough training data that covers our input space well. This is difficult to impossible. Another solution is to exploit that the input data lies on a lower-dimensional manifold. Finding this manifold will help with the curse of dimensionality.

Yet another solution is to assume that the data results from a composition of pieces. Very much like natural languages exploit compositionality to give representations and meanings to complex ideas. The composition maybe be *parallel* or *sequential*. Parallel composition (which will be reflected by the widths of the neural network layers) gives us the idea of distributed representations (*different views*). Sequential composition (which will be reflected by the *depth* of the network – number of layers) deals with multiple levels of feature learning.

Why now?

All these challenges existed for a long time and we have had the models (e.g. multi-layer neural networks) to tackle those challenges for almost as long.

Question: So, what is different now? Why can we do deep learning? Why now?

Answer: We have **Big data** and we have the **infrastructure** (software/hardware) to tackle the optimization needed for learning!

Example Neural Networks → cf. lecture slides

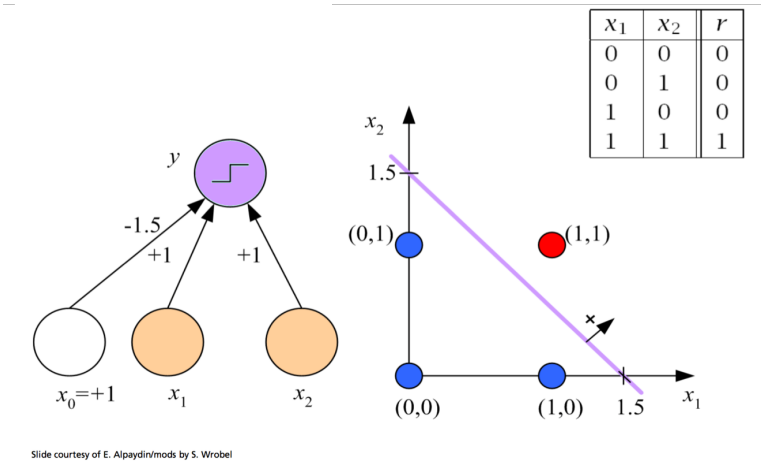
[1] http://rinuboney.github.io/2015/10/18/theoretical-motivations-deep-learning.html

2 Feed-Forward Neural Networks

Feed-forward neural networks (FF-NNs) in its simplest form can be derived as *multi-layer perceptrons*.

Recall: *Perceptron*: $h(\mathbf{x}) = sign(\mathbf{w}^\top \mathbf{x})$
The perceptron is a linear model $\Rightarrow$ does not work for non-linearly separable data.

Insight: Perceptron is a *single layer NN* as shown in Figure 1.



x_1	x_2	r
0	0	0
0	1	0
1	0	0
1	1	1

Figure 1: Example structure of a single layer neural network modeling a *perceptron*. There is the *input layer* and **one** layer with **one** unit with the *sign function* as its *activation function*.

Insight: How to make a linear model non-linear?
$\Rightarrow$ feature transformation: $\mathbf{w}^\top \phi(\mathbf{x})$

Idea: Learn $\phi(\mathbf{x})$

$$\phi(\mathbf{x}) = \begin{bmatrix} g_1(\mathbf{x}) \\ \vdots \\ g_m(\mathbf{x}) \end{bmatrix} = \begin{bmatrix} z_1 \\ \vdots \\ z_m \end{bmatrix}$$

where each $g_j(\mathbf{x})$ is a linear classifier $g_j(\mathbf{x}) = sign(\mathbf{w}_j^\top \mathbf{x} + \mathbf{b}_j)$, that applies a linear transformation to its inputs followed by a (non-linear) *activation function*. This learns low level problems that are "simpler". Their output then becomes the input to the main linear classifier $h(\mathbf{z})$.

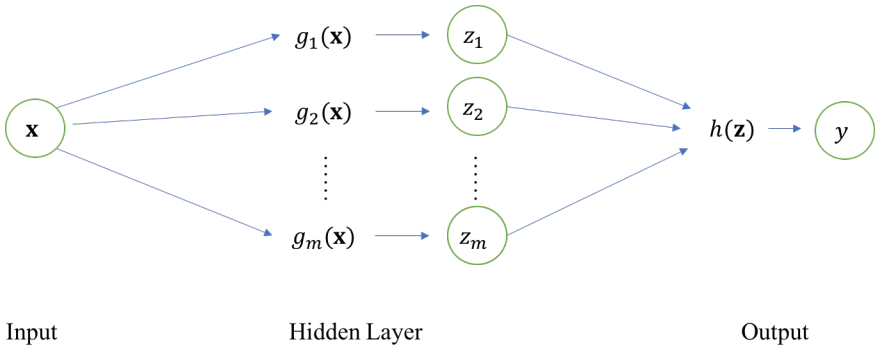


Figure 2: Structure of a simple neural network with **two** layers of activations functions (one *hidden* and one *output* layer).

2.1 Example: XOR

The structure of the network can be derived from a decomposition of the original probelm: x_1 `XOR` $x_2 = (x_1$ `AND` $\sim x_2)$ `OR` $(x_2$ `AND` $\sim x_1)$, where $\sim$ denotes logical negation (`not`). This structure is shown in Fig 3.

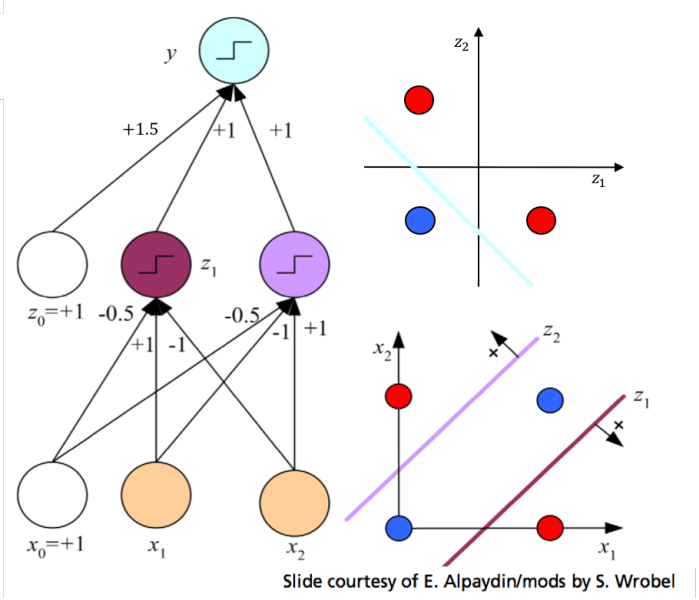


Figure 3: Example of XOR function

Goal: Given points $(0,0),(1,0),(0,1)$, and $(1,1)$ and the above network structure. Both $(0,0)$ and $(1,1)$ should end up in the same point/region. This can be achieved by choosing *appropriate*[2] network parameters as for instance the following ones:

$$W^{(1)} = \begin{bmatrix} +1 & -1 \\ -1 & +1 \end{bmatrix} \in \mathbb{R}^{m \times d} \quad \mathbf{b}^{(1)} = \begin{bmatrix} -0.5 \\ -0.5 \end{bmatrix} \in \mathbb{R}^{m}$$
$$W^{(2)} = \begin{bmatrix} +1 & +1 \end{bmatrix} \in \mathbb{R}^{1 \times m} \quad \mathbf{b}^{(2)} = \begin{bmatrix} 1.5 \end{bmatrix} \in \mathbb{R}^{1}$$

Now, we can apply the neural network transformations given by those paramters to our input data.
UNIT 1:

$$(0,0) : W^{(1)}_{1:}\mathbf{x}_1 + b^{(1)}_1 = [+1 \; -1] \begin{bmatrix} 0 \\ 0 \end{bmatrix} - 0.5 = -0.5 \qquad \Rightarrow z_{11} = sign(-0.5) = -1$$
$$(1,0) : W^{(1)}_{1:}\mathbf{x}_2 + b^{(1)}_1 = +0.5 \qquad \Rightarrow z_{12} = +1$$
$$(0,1) : W^{(1)}_{1:}\mathbf{x}_3 + b^{(1)}_1 = -1.5 \qquad \Rightarrow z_{13} = -1$$
$$(1,1) : W^{(1)}_{1:}\mathbf{x}_4 + b^{(1)}_1 = -0.5 \qquad \Rightarrow z_{14} = -1$$

[2]Note that there are many different ways to pick parameters that solve this problem.

UNIT 2:

$$
\begin{aligned}
(0,0): W^{(1)}_{2:}\mathbf{x}_1 + b^{(1)}_2 = [-1+1]\begin{bmatrix}0\\0\end{bmatrix} - 0.5 = -0.5 \quad &\Rightarrow z_{21} = sign(-0.5) = -1\\
(1,0): W^{(1)}_{2:}\mathbf{x}_2 + b^{(1)}_2 = -1.5 \quad &\Rightarrow z_{22} = -1\\
(0,1): W^{(1)}_{2:}\mathbf{x}_3 + b^{(1)}_2 = 0.5 \quad &\Rightarrow z_{23} = +1\\
(1,0): W^{(1)}_{2:}\mathbf{x}_4 + b^{(1)}_2 = -0.5 \quad &\Rightarrow z_{24} = -1
\end{aligned}
$$

UNIT 3:

$$
\begin{aligned}
(-1,-1): W^{(2)}_{1:}\underbrace{\mathbf{z}_1}_{=\mathbf{z}_4} + b^{(2)}_1 = -0.5 \quad &\Rightarrow \underbrace{y_1}_{=y_4} = -1\\
(+1,-1): W^{(2)}_{1:}\mathbf{z}_2 + b^{(2)}_1 = 1.5 \quad &\Rightarrow\ y_2 = +1\\
(-1,+1): W^{(2)}_{1:}\mathbf{z}_3 + b^{(2)}_1 = 1.5 \quad &\Rightarrow\ y_3 = +1
\end{aligned}
$$

3 Parameter Learning

Question: How to come up with the parameters (we assume the architecture is given)?

$\Rightarrow$ **learn** them by **minimizing the loss** for $g_j(\mathbf{x})\ \forall j$ and $h(\mathbf{z})$ via gradient descent

- we need differentiable activation functions! $\Rightarrow$ e.g use *sigmoid* instead of *sign*
- input of $h(\mathbf{z})$ depends on outputs of $g_j(\mathbf{x}) \Rightarrow$ we need an *iterative strategy* to update the parameters of each layer

LEARNING STRATEGY:

(0) initialize weights randomly

(1) feed training points though network (*Forward propagation*)

(2) compute gradient descent update on weights by propagating back the error (*Back propagation*)

Notation:

- $W^{(l)}$ weight parameter matrix for layer l. The jth row $W^{(l)}_{j:}$ are the weights that correspond to unit j.
- $\mathbf{b}^{(l)}$ bias terms for layer l.
- $\mathbf{a}^{(l)} = W^{(l)}\mathbf{z}^{(l-1)} + \mathbf{b}^{(l)}$ linear transformation of the input to layer l.
- $\mathbf{z}^{(l)}$ are the outputs of layer l after applying the activation function $g_l : \mathbb{R}^{m_l} \to \mathbb{R}^{m_l}$ to every entry in $\mathbf{a}^{(l)}$. We will write this in short as:
$$g_l(\mathbf{a}^{(l)}) = \mathbf{z}^{(l)}.$$
- number of layers $l = \{1, \dots, L\}$
- m_l is the number of units in layer l. The size of the input layer is $m_0 = d$. Note that not every layer needs to have the same numer of units.

3.1 Forward Propagation

Pass input data through network layer by layer:

(1) **apply linear transformation**: $W^{(l)}\mathbf{z}^{(l-1)}$ with $\mathbf{z}^{(0)} = \mathbf{x} \in \mathbb{R}^d$ and weights $W^{(l)} \in \mathbb{R}^{m_l \times m_{l-1}}$. m_l is the output dimension and m_{l-1} is the input dimension to layer l.

(2) **apply activation function**: $g_l(W^{(l)}\mathbf{z}^{(l-1)} + \mathbf{b}^{(l)}) = \mathbf{z}^{(l)}$

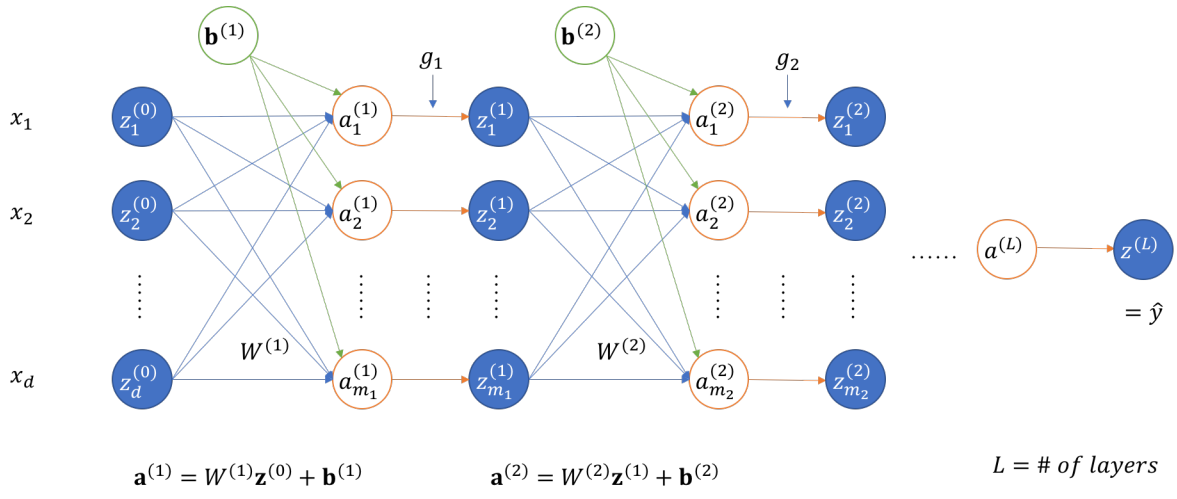


Figure 4: Forward propagation of a multiple-layer neural network

3.2 Compute Loss in Output Layer

Let's look at one training example $\mathbf{x}$ for now and let's use the squared loss $\ell(\mathbf{x}, y) = \frac{1}{2}(z^{(L)} - y)^2$
Then, let's update $W^{(l)}$'s according to gradeint descent (GD) update rules.

For the last layer L:

$$\ell(\mathbf{x}, y) = \frac{1}{2}(g_L(\underbrace{W^{(L)}\mathbf{z}^{(L-1)} + b^{(L)}}_{a^{(L)}}) - y)^2 \tag{1}$$

$$\frac{\partial \ell}{\partial w_{ij}^{(L)}} = \frac{\partial \ell}{\partial g_L}\frac{\partial g_L}{\partial a_i^{(L)}}\frac{\partial a_i^{(L)}}{\partial w_{ij}^{(L)}} \tag{2}$$

$$= \underbrace{(z^{(L)} - y) * g_L'(a_i^{(L)})}_{=\delta_i^{(L)}} * z_j^{(L-1)} \tag{3}$$

where $i = 1$, $j = 1, \ldots, m_{L-1}$ and $\delta_i^{(L)}$ is the "error term" in the output layer for output dimension i.

Weight update:

$$W^{(L)} \leftarrow W^{(L)} - \alpha \nabla \ell(W^{(L)})$$

α is the GD *learning rate* and $\nabla \ell(W^{(L)})$ can be computed using Eq. (3) and $\mathbf{z}^{(L-1)}$, $\mathbf{z}^{(L)}$, $\mathbf{a}^{(L)}$ as given from the forward propagation step. In terms of δ this update can be written as

$$W^{(L)} \leftarrow W^{(L)} - \alpha\, \delta^{(L)} \mathbf{z}^{(L-1)^\top}.$$

3.3 Back Propagation

Recursively compute weight updates for $L-1, L-2, \ldots, 1$

For 2nd last layer $L-1$:

in Eq. (1) substitute $\mathbf{z}^{(L-1)} = g_{L-1}(\underbrace{W^{(L-1)}\mathbf{z}^{(L-2)} + \mathbf{b}^{(L-1)}}_{=\mathbf{a}^{(L-1)}})$ and take derivative

$$\frac{\partial \ell}{\partial w_{jk}^{(L-1)}} = \underbrace{\underbrace{\frac{\partial \ell}{\partial g_L}\frac{\partial g_L}{\partial a^{(L)}}}_{\delta^{(L)}} \underbrace{\frac{\partial a^{(L)}}{\partial g_{L-1}}\frac{g_{L-1}}{\partial a_j^{(L-1)}}}_{W_{:j}^{(L)} g'_{L-1}(a_j^{(L-1)})}}_{=\delta_j^{(L-1)}} \underbrace{\frac{\partial a_j^{(L-1)}}{\partial w_{jk}^{(L-1)}}}_{z_k^{(L-2)}} \tag{4}$$

where $j = 1, \ldots, m_{L-1}$ and $k = 1, \ldots, m_{L-2}$ and $\delta_i^{(L-1)}$ is the "error term" in that layer for the layer output dimension j.

Weight update:

$$W^{(L-1)} \leftarrow W^{(L-1)} - \alpha\, \boldsymbol{\delta}^{(L-1)} \mathbf{z}^{(L-2)\top}$$

Insight: represent error term recursively

$$\boldsymbol{\delta}^{(l-1)} = \underbrace{g'_{l-1}(\mathbf{a}^{(l-1)})}_{\nabla g_{l-1}(\mathbf{a}^{(l-1)})} \circ W^{(l)\top} \boldsymbol{\delta}^{(l)} \tag{5}$$

where $\circ$ is element-wise multiplication.

$\Rightarrow$ view this step as *propagating* the error term δ *back* through the neural network with **same structure**, **same weights**, **no biases**, **reverse arrows** and **modified activation functions** $\tilde{g}_l(s_i) = s_i\, g'(a_i^{(l)})$, where the $a_i^{(l)}$'s are given by the forward propagation. The modified network is illustrated in Fig. 5.

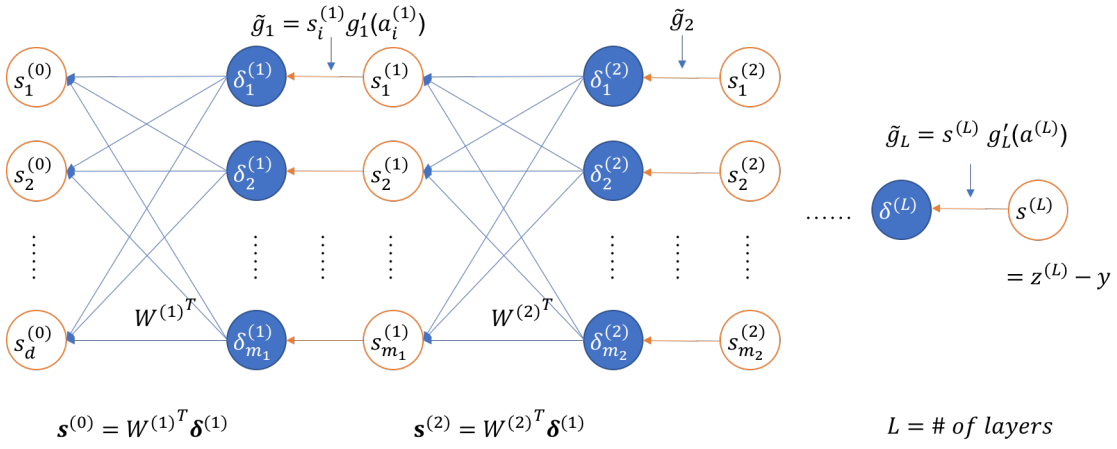


Figure 5: Forward propagation of a multiple-layer neural network

3.4 Training Algorithm

Now, we can summarize these steps into one algorithm for training feed-forward NNs, cf. Algorithm 1.

Algorithm 1 Train FF-NN

INPUT: $D = \{(\mathbf{x}_i, y_i)\}_{i=1,\dots,n}$, NN $= \begin{cases} L \\ g_l & \forall\, l = 1, \dots, L \\ m_l & \forall\, l = 1, \dots, L \end{cases}$

INITIALIZATION: $\mathbf{b}^{(l)}, W^{(l)} \leftarrow$ some initial distinct value (e.g. random small weight) $\forall\, l = 1, \dots, L$

repeat
  for each training example $(\mathbf{x}, y) \in D$ **do**
    $\mathbf{z}^{(0)} = \mathbf{x}$

    FORWARD PROPAGATION:
    for $l = 1, \dots, L$ **do**
      $\mathbf{a}^{(l)} = W^{(l)}\mathbf{z}^{(l-1)} + b^{(l)}$
      $\mathbf{z}^{(l)} = g_l(\mathbf{a}^{(l)})$
    end for

    $\boldsymbol{\delta}^{(L)} = (z^{(L)} - y) \cdot g'_L(\mathbf{a}^{(L)})$ // compute loss in output layer

    BACK PROPAGATION:
    $\boldsymbol{\delta}^{(L-1)} = g'_{L-1}(\mathbf{a}^{(L-1)}) \circ {W^{(L)}}^\top \boldsymbol{\delta}^{(L)}$ // use *old* weights
    $W^{(L)} \leftarrow W^{(L)} - \alpha\, \boldsymbol{\delta}^{(L)} {\mathbf{z}^{(L-1)}}^\top$
    $\mathbf{b}^{(L)} \leftarrow \mathbf{b}^{(L)} - \alpha\, \boldsymbol{\delta}^{(L)}$
    for $l = L-1, \dots, 1$ **do**
      $\Delta W^{(l)} = \boldsymbol{\delta}^{(l)} {\mathbf{z}^{(l-1)}}^\top$
      $\Delta \mathbf{b}^{(l)} = \boldsymbol{\delta}^{(l)}$
      $\boldsymbol{\delta}^{(l-1)} = g'_{l-1}(\mathbf{a}^{l-1}) \circ {W^{(l)}}^\top \boldsymbol{\delta}^{(l)}$ // use *old* weights
      $W^{(l)} \leftarrow W^{(l)} - \alpha\, \Delta W^{(l)}$
      $\mathbf{b}^{(l)} \leftarrow \mathbf{b}^{(l)} - \alpha\, \Delta \mathbf{b}^{(l)}$
    end for

  end for
until *error* $= \frac{1}{n} \sum_{i=1}^{n} (z_i^{(L)-y_i}) < \varepsilon$

4 NNs are Universal Non-linear Models

- hidden units consist of a **non-linear** activation function on top of a **linear transformation**
- NNs are **universal approximators**
- theoretically we do not need more than one hidden layer (*shallow network*) ⇒ large number of units
- practically we use more than one hidden layer (*deep network*) ⇒ share information

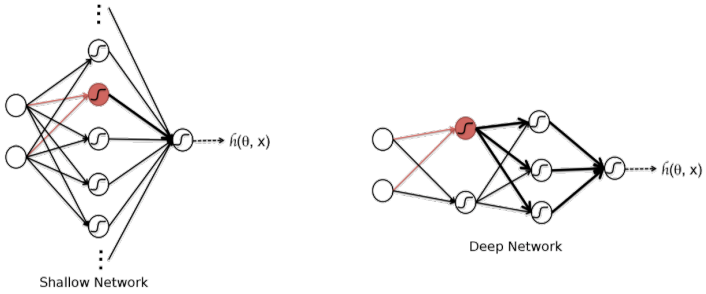


Figure 6: In theory a shallow network with one hidden layer (*left*) can model any possible decision boundary/prediction function. However, the number of units grows *super-exponentially* with the number of input dimensions. Thus, we use deeper networks (*right*) in practice.

5 Model Choices

5.1 Output Units

The activation function used in the output unit(s) is straightforward (*no magic*) as it depends on the output space:

- **regression** – linear unit: $h(a) = a = \mathbf{w}^\top \mathbf{z} + b$ (activation function is the **identity function**)
- **classification** – sigmoid unit: $h(a) = \frac{1}{1+e^{-a}}$
- **multi-classification** – softmax: $h(\mathbf{a}) = \frac{e^{\mathbf{a}_i}}{\sum_j e^{\mathbf{a}_j}}$ with $\mathbf{a} \in \mathbb{R}^k$

5.2 Activation Functions

Some well-known functions used as activation functions in the hidden layers are:

- **sign function**: *Perceptron*, not in NNs (not differentiable)
- **sigmoid function**: $sigm(a) = \frac{1}{1+e^{-a}}$, $sigm'(a) = sigm(a)(1 - sigm(a))$
- **tanh**: $tanh(a) = \frac{e^a - e^{-a}}{e^a + e^{-a}}$, $tanh'(a) = 1 - tanh^2(a)$
- **rectified linear unit**: $relu(a) = \max(a, 0)$, $relu'(a) = \begin{cases} 0 & a < 0 \\ 1 & a > 0 \\ NAN & a = 0 \end{cases}$ (use either left or right derivative)

Which hidden units to use?
Both *sigmoid* and **tanh** are not ideal as they saturate (gradient does not change for very high or very low inputs). **Relu** is the most commonly used as it overcomes this problem and is easy to optimize. A generalization is the **parametric relu**: $\max(0, a) + \alpha \min(0, a)$.

→ cf. lecture slides

Some other activation functions:

- maxout (use groups of a_i): $z \leftarrow \max_i(a_i)$
- RBF unit: $z_i = \exp(\frac{1}{\sigma_i^2}||\mathbf{w} - \mathbf{z}||^2)$
- soft plus: $g(a) = \log(1 + e^{-a})$ (smooth version of relu)

5.3 Architecture Design

How to choose **depths** (numbers of layers) and **width** of each layer?

Empirically, greater depth does seem to result in better generalization for a wide variety of tasks. → cf. lecture slides

But the true answer is: *nobody knows* (yet)!

6 Summary

+ Flexible and general function approximation framework
+ Can build extremely powerful models by adding more layers
– Hard to analyze theoretically (e.g., training is prone to local optima)
– Huge amount of training data, computing power may be required to get good performance
– The space of implementation choices is huge (architecture design, parameters)