CSE517A Machine Learning **Spring 2025**

Lecture 9: Feed-Forward Neural Networks

Instructor: Marion Neumann

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 Multi-layer Perceptron (MLP)

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

2.1 Recall: The Perceptron

The preceptron is a simple linear classifier: $h(\mathbf{x}) = sign(\mathbf{w}^\top \mathbf{x})$. It's main drawback is that is does not work for non-linearly separable data.

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

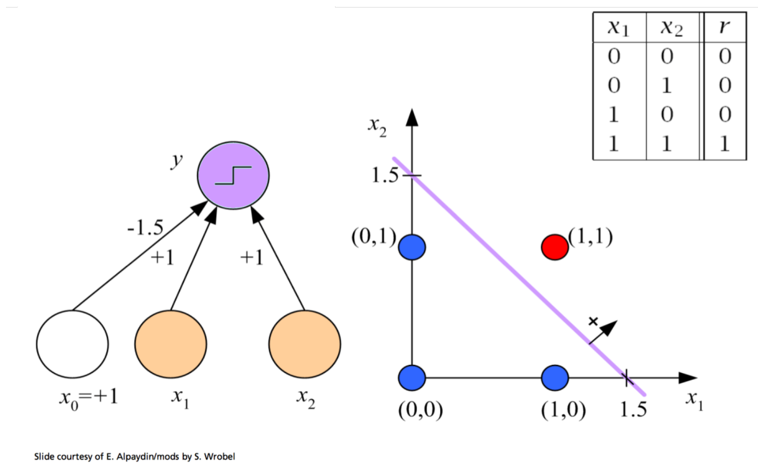


Figure 1: Example structure of a single layer neural network modeling a *perceptron*. There is the *input layer* and **one** layer with **one** unit with the *sign function* as its *activation function*.

2.2 Recall: Feature-Space Transformation

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 $h(\mathbf{z})$.

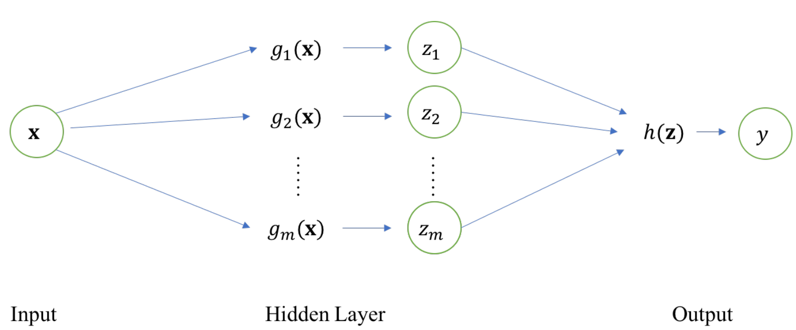


Figure 2: Structure of a simple NN with **two** layers (one *hidden* and one *output*).

2.3 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 ~ x_2) OR (x_2 AND ~ x_1), where ~ denotes logical negation (not). This structure is shown in Fig 3.

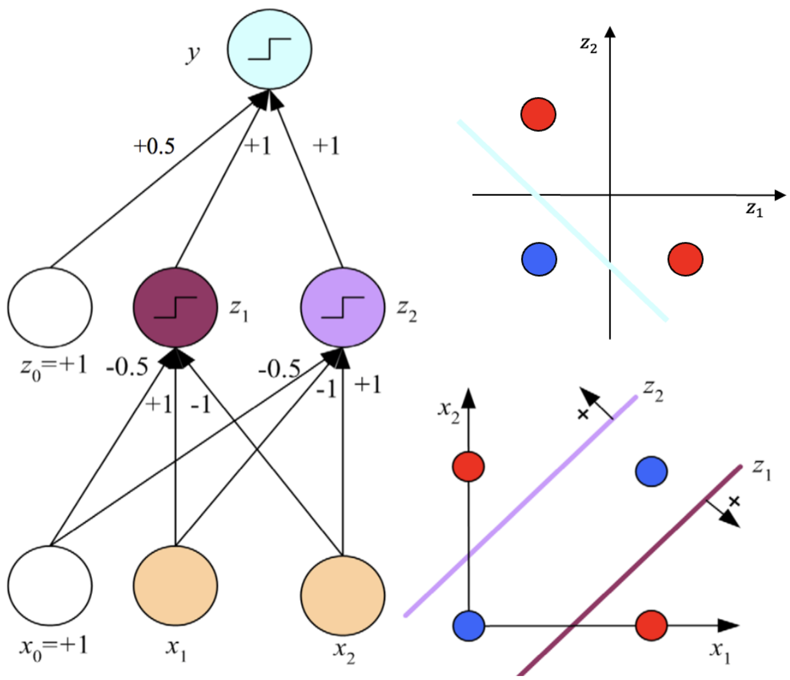


Figure 3: Example of XOR function

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:

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

Now, we can apply the neural network transformations given by those parameters 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:

$$(0,0): W_{2:}^{(1)}\mathbf{x}_1 + b_2^{(1)} = [-1 + 1]\begin{bmatrix}0\\0\end{bmatrix} - 0.5 = -0.5 \qquad \Rightarrow z_{21} = sign(-0.5) = -1$$

$$(1,0): W_{2:}^{(1)}\mathbf{x}_2 + b_2^{(1)} = -1.5 \qquad \Rightarrow z_{22} = -1$$

$$(0,1): W_{2:}^{(1)}\mathbf{x}_3 + b_2^{(1)} = 0.5 \qquad \Rightarrow z_{23} = +1$$

$$(1,0): W_{2:}^{(1)}\mathbf{x}_4 + b_2^{(1)} = -0.5 \qquad \Rightarrow z_{24} = -1$$

UNIT 3:

$$(-1,-1): W_{1:}^{(2)}\underbrace{\mathbf{z}_1}_{=\mathbf{z}_4} + b_1^{(2)} = -1.5 \qquad \Rightarrow \underbrace{y_1}_{=y_4} = -1$$

$$(+1,-1): W_{1:}^{(2)}\mathbf{z}_2 + b_1^{(2)} = 0.5 \qquad \Rightarrow y_2 = +1$$

$$(-1,+1): W_{1:}^{(2)}\mathbf{z}_3 + b_1^{(2)} = 0.5 \qquad \Rightarrow y_3 = +1$$

3 Parameter Learning: From MLP to Feed-Forward NNs

Question: How to come up with the parameters assuming that the architecture (number of layers and number of units per layer) is given?

$\Rightarrow$ **learn** them by **minimizing the loss** for $g_j(\mathbf{x})\ \forall j$ and $h(\mathbf{z})$ via gradient descent. This is what turns the *multi-layer perceptron* into a *feed-forward neural network*.
The main insights to make this work are:

- 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 point though network (*Forward propagation*)

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

(ITERATE) over (1) and (2) for each training data point

THEN (REPEAT ITERATION) until weights don't change anymore

Notation and Definitions:

- $W^{(l)}$ weight parameter matrix for layer l. The jth row $W_{j:}^{(l)}$ 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:
$$g_l(\mathbf{a}^{(l)}) = \mathbf{z}^{(l)}.$$
- g_l is applied to a vector, so we treat it as a **vector valued function**. $g_l(\mathbf{a}^{(l)})$ means that g_l is applied to every dimension of $\mathbf{a}^{(l)}$.

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

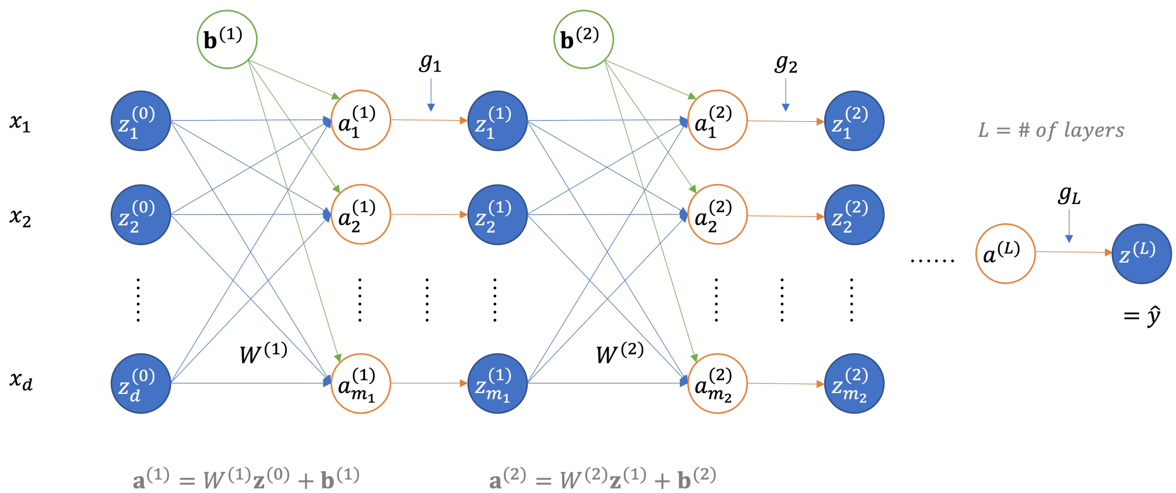


Figure 4: Forward propagation of a multi-layer neural network for a regression task.

3.2 Compute Loss in Output Layer (assume $y \in \mathbb{R}$ - Regression)

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 gradient descent (GD) update rules.

For the last layer L:

$$\ell(\mathbf{x}, y) = \frac{1}{2}(g_L(\underbrace{W^{(L)}\mathbf{z}^{(L-1)} + b^{(L)}}_{a_1^{(L)}}) - y)^2 \tag{1}$$

$$\frac{\partial \ell}{\partial w_{1j}^{(L)}} = \frac{\partial \ell}{\partial g_L}\frac{\partial g_L}{\partial a_1^{(L)}}\frac{\partial a_1^{(L)}}{\partial w_{ij}^{(L)}} \tag{2}$$

$$= \underbrace{(z^{(L)} - y) * g_L'(a_1^{(L)})}_{=\delta_1^{(L)}} * z_j^{(L-1)} \tag{3}$$

where $j = 1, \dots, m_{L-1}$ and $\delta_1^{(L)}$ is the "error term" in the output layer. Note that for regression $W^{(L)} \in \mathbb{R}^{1 \times m_{L-1}}$ is a row vector and $\delta_1^{(L)} = \delta^{(L)}$ is a scalar. If your network had **more than one output dimension** (as for multi-class classification or unsupervised learning) $W^{(L)}$ would be a matrix and $\boldsymbol{\delta}^{(L)}$ would be a vector for which you get it's ith dimension by substituting i for 1 in Eq. (3).

Weight update:

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

α is the GD *learning rate* and $\Delta W^{(L)}$ can be computed using Eq. (3) and $\mathbf{z}^{(L-1)}$, $z^{(L)}$, $a^{(L)}$ 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 $L-1, L-2, \dots, 1$

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

$$\frac{\partial \ell}{\partial w_{jk}^{(L-1)}} = \frac{\partial \ell}{\partial g_L} \frac{\partial g_L}{\partial \mathbf{a}^{(L)}} \frac{\partial \mathbf{a}^{(L)}}{\partial g_{L-1}} \frac{g_{L-1}}{\partial \mathbf{a}^{(L-1)}} \frac{\partial \mathbf{a}^{(L-1)}}{\partial w_{jk}^{(L-1)}} \tag{4}$$

where $j = 1, \dots, m_{L-1}$ and $k = 1, \dots, m_{L-2}$.
Note: now we have to take a couple of less-straightforward derivatives (g_{L-1} is a *vector valued function* and we take its derivative wrt. a vector $\mathbf{a}^{(L-1)}$ returning a matrix of partial derivatives; then we take the derivative of vector $\mathbf{a}^{(L-1)}$ wrt. a scalar $w_{jk}^{(L-1)}$ returning a (column) vector of partial derivatives).

$$\begin{aligned}
\frac{\partial \ell}{\partial w_{jk}^{(L-1)}} &= \frac{\partial \ell}{\partial g_L} \quad \frac{\partial g_L}{\partial a^{(L)}} \qquad \frac{\partial a^{(L)}}{\partial g_{L-1}} \qquad \frac{g_{L-1}}{\partial \mathbf{a}^{(L-1)}} \qquad \frac{\partial \mathbf{a}^{(L-1)}}{\partial w_{jk}^{(L-1)}} \\
&= \underbrace{(z^{(L)} - y) * g_L'(a^{(L)})}_{=\delta^{(L)}} \quad W^{(L)} \begin{bmatrix} g'(a_1^{(L-1)}) & & & & \\ & \ddots & & & 0 \\ & & g'(a_j^{(L-1)}) & & \\ & 0 & & \ddots & \\ & & & & g'(a_{m_{L-1}}^{(L-1)}) \end{bmatrix} \begin{bmatrix} 0 \\ \vdots \\ z_k^{(L-2)} \\ 0 \\ \vdots \end{bmatrix} \leftarrow jth\ row \\
&= \delta^{(L)} \quad \begin{bmatrix} w_{11}^{(L)} & \cdots & w_{1j}^{(L)} & \cdots & w_{1m_{L-1}}^{(L)} \end{bmatrix} \begin{bmatrix} 0 \\ \vdots \\ g'(a_j^{(L-1)}) z_k^{(L-2)} \\ 0 \\ \vdots \end{bmatrix} \\
&= \underbrace{\delta^{(L)}\ w_{1j}^{(L)}\ g'(a_j^{(L-1)})}_{\delta_j^{(L-1)}} \ z_k^{(L-2)}
\end{aligned}$$

Weight update:

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

Exercise 3.1. What about b?

(a) State the weight update rule for $b^{(L)}$ and $\mathbf{b}^{(L-1)}$.

(b) Derive the weight updates for both following the chain rule all the way through!

Now, let's do the same derivation for an arbitrary $(l-1)$. Note that now, $\boldsymbol{\delta}^{(l)}$ is a vector (will need to be transposed) and $W^{(l)}$ is a matrix.

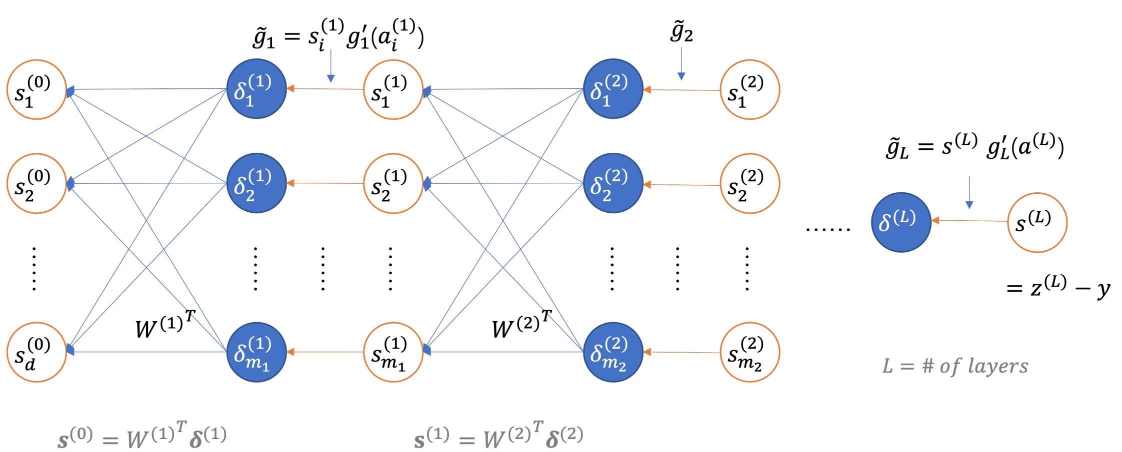
$$
\begin{aligned}
\frac{\partial \ell}{\partial w_{jk}^{(l-1)}} &= \frac{\partial \ell}{\partial g_l} \quad \frac{\partial g_l}{\partial \mathbf{a}^{(l)}} \quad \frac{\partial \mathbf{a}^{(l)}}{\partial g_{l-1}} \quad \frac{g_{l-1}}{\partial \mathbf{a}^{(l-1)}} \quad \frac{\partial \mathbf{a}^{(l-1)}}{\partial w_{jk}^{(l-1)}} \\
&= \boldsymbol{\delta}^{(l)^{\top}} \quad W^{(l)} \begin{bmatrix} g'(a_1^{(l-1)}) & & & & 0 \\ & \ddots & & & \\ & & g'(a_j^{(l-1)}) & & \\ & & & \ddots & \\ 0 & & & & g'(a_{m_{l-1}}^{(l-1)}) \end{bmatrix} \begin{bmatrix} 0 \\ \vdots \\ z_k^{(l-2)} \\ 0 \\ \vdots \end{bmatrix} \leftarrow jth\ row \\
&= \boldsymbol{\delta}^{(l)^{\top}} \begin{bmatrix} w_{11}^{(l)} & \cdots & w_{1j}^{(l)} & \cdots & w_{1m_{l-1}}^{(l)} \\ \vdots & & \vdots & & \vdots \\ w_{m_l 1}^{(l)} & \cdots & w_{lj}^{(l)} & \cdots & w_{lm_{l-1}}^{(l)} \end{bmatrix} \begin{bmatrix} 0 \\ \vdots \\ g'(a_j^{(l-1)}) z_k^{(l-2)} \\ 0 \\ \vdots \end{bmatrix} \\
&= \underbrace{g'(a_j^{(l-1)}) \ \boldsymbol{\delta}^{(l)^{\top}} \ W_{:j}^{(l)}}_{=\delta_j^{(l-1)}} \ z_k^{(l-2)}
\end{aligned}
$$

Insight: represent error term recursively

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

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 can be illustrated as follows:



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

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

    BACK PROPAGATION:
    $\boldsymbol{\delta}^{(L-1)} = g'_{L-1}(\mathbf{a}^{(L-1)}) \circ W^{(L)^\top}\delta^{(L)}$ // use *old* weights
    $W^{(L)} \leftarrow W^{(L)} - \alpha\,\delta^{(L)}\mathbf{z}^{(L-1)^\top}$
    $\mathbf{b}^{(L)} \leftarrow \mathbf{b}^{(L)} - \alpha\,\delta^{(L)}$
    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* $= \frac{1}{n}\sum_{i=1}^{n}(z_i^{(L)} - y_i) < \varepsilon$

Exercise 3.2. Lets look at some properties of this training algorithm.

(a) What can go wrong if you simply initialize all the weights to exactly zero?

(b) Let V be the total number of units in the NN. Express V in terms $m_0,\dots,m_L$.

(c) Let Q be the total number of weights (including b) in the NN. Express Q in terms $m_0,\dots,m_L$.

(d) In terms of V and Q, how many computations are made in forward propagation (additions, multiplications and evaluations of $g(\cdot)$).

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

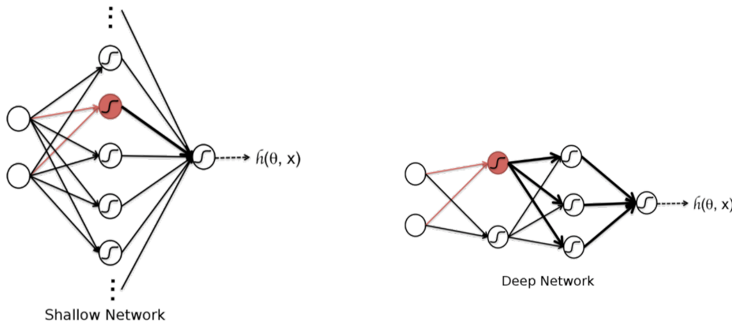


Figure 5: In theory a shallow network with one hidden layer (*left*) can model any possible decision boundary/prediction function. However, the number of units grows *super-exponentially* with the number of input dimensions. Thus, we use deeper networks (*right*) in practice.

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**)
- **binary classification** – sigmoid unit: $h(a) = \frac{1}{1+e^{-a}}$
- k-class **multi-class classification** – softmax: $h_c(\mathbf{a}) = \frac{e^{\mathbf{a}_c}}{\sum_j e^{\mathbf{a}_j}}$ with $\mathbf{a} \in \mathbb{R}^k$

5.2 Loss Functions

The loss function used for computing the error in the output unit is fairly straightforward as it is dictated by the (supervised) learning task:

- **regression** – we use the *squared loss* $\ell(\mathbf{x}, y) = \frac{1}{2}(z^{(L)} - y)^2$
- **binary classification** with classes $y \in \{0, 1\}$ – *cross-entropy loss*: $\ell(\mathbf{x}, y) = -\left[(1-y)\ln z_0^{(L)} + y \ln z_1^{(L)}\right]$
- k-class **multi-class classification** – *cross-entropy loss*: $\ell(\mathbf{x}, y) = -\sum_{c=1}^{k} \delta_{y=c} \ln z_c^{(L)}$

Exercise 5.1. Weight (and Bias) updates for Classification

(a) Derive the weight update rule for $W^{(L)}$, $W^{(L-1)}$, $b^{(L)}$, and $b^{(L-1)}$ for a **multi-class classification** problem.

(b) The weight update rule for an arbitrary layer ℓ is given by $W^{(\ell)} = W^{(\ell)} - \alpha \boldsymbol{\delta}^{(\ell)} \mathbf{z}^{(\ell)^\top}$. What is $\boldsymbol{\delta}^{(\ell)}$ $\forall \ell$ (for multi-class classification)?

5.3 Activation Functions

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

- **sign function**: *Perceptron*, not used 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**: $relu(a) = \max(a, 0)$, $relu'(a) = \begin{cases} 0 & a < 0 \\ 1 & a > 0 \\ NAN & a = 0 \quad \text{(use either left or right derivative)} \end{cases}$
- **parametric relu**: $prelu(a) = \max(0, a) + \lambda \min(0, a)$.
- **leaky relu**: $lrelu(a) = \max(0.1a, a)$
- **exponential linear unit**: $elu(a) = \begin{cases} \lambda(e^a - 1) & a < 0 \\ a & a \geq 0 \end{cases}$
- **maxout** (use groups of a_k as input): $z \leftarrow \max_i(a_i)$
- 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)

Which hidden units/activation functions 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.
→ cf. lecture slides

Exercise 5.2. Activation Functions

(a) Draw all of the activation functions. Carefully label your axes and mark the units.

(b) Compute and draw the the gradient functions (of all activation functions). Carefully label your axes and mark the units.

(c) Discuss the pros and cons of each activation function.

5.4 Architecture Design

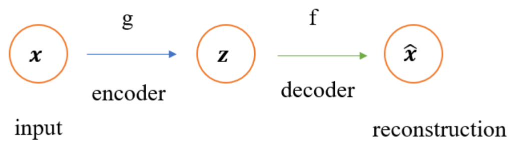
How to choose **depths** (numbers of layers) and **width** of each layer?

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 Autoencoder

Autoencoders are unsupervised neural networks that are used for dimensionality reduction/manifold learning, generative ML, and *representation learning*. The goal of representation learning is to avoid manual feature engineering and perform data-driven feature extraction or feature space transformation. In fact, the learned data embeddings may be used as input to a traditional supervised ML method such as a support vector machine.
Since autoencoders can be trained in an unsupervised manner, they are also extremely useful for *pre-training* (cf. Strategies for Deep Learning lecture).



The basic architecture of an autoencoder with 2 layers – one *enocder* and one *decoder* –, where $\mathbf{x}, \hat{\mathbf{x}} \in \mathbb{R}^d$ and $\mathbf{z} \in \mathbb{R}^m$. The encoder encodes $\mathbf{x}$ into $\mathbb{R}^m$ and the decoder maps $\mathbf{z}$ back into $\mathbb{R}^d$. Typically, we use an *undercomplete* autoencoder, where $m < d$.

Note that this is just a simple 2 layer FF-NN that can be trained using GD or back-propagation. For the loss function $\ell(\mathbf{x}, \underbrace{f(g(\mathbf{x}))}_{\hat{\mathbf{x}}})$ we choose the *mean square error* aka *squared reconstruction error*.

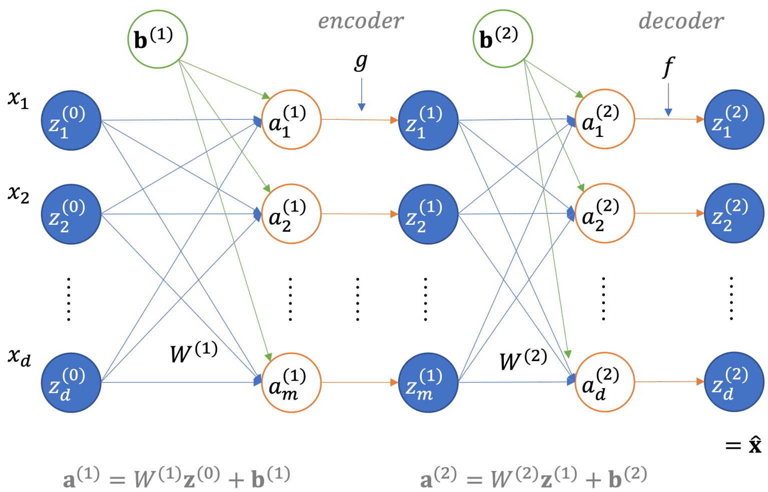


Figure 6: Detailed illustration of encoder and decoder architecture.

Relation to PCA

We get PCS, if use the mean square error as loss function and a linear encoder function with identity activation $g(\mathbf{x}) = W\mathbf{x} + b$.

$\Rightarrow$ use non-linear g and f to get a non-linear generalization of PCA

Note: if we allow g and f to be arbitrarily complex, we will simply learn a code to represent the training data (overfitting). The same problem arises if we have

- an overcomplete autoencoder: $m > d$
- a deep autoencoder: $L > 2$

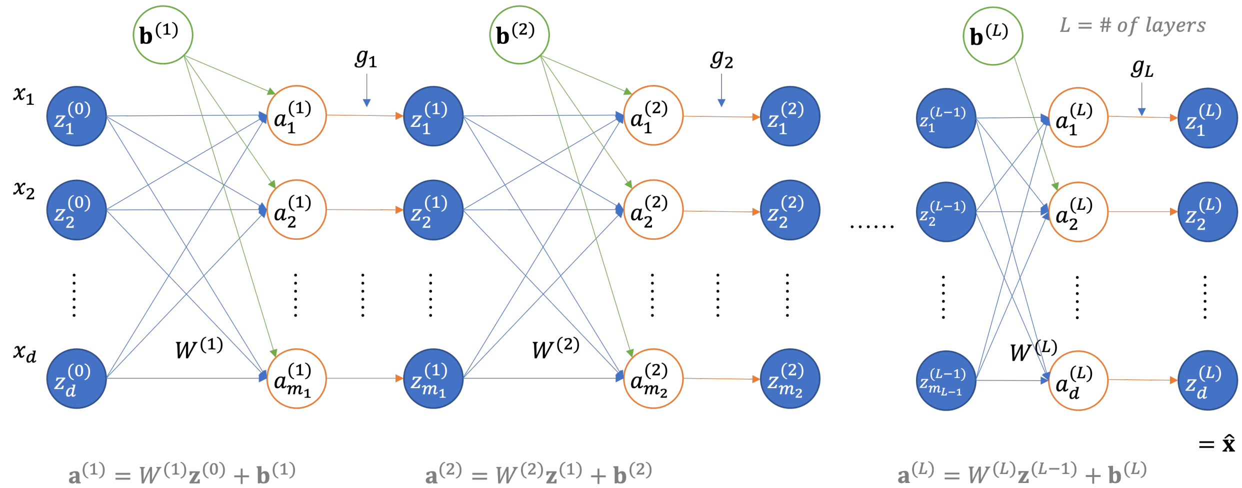


Figure 7: Deep autoencoder.

To train such autoencoders we need to perform regularization:

(1) **sparse autoencoder** $\ell(\mathbf{x}, f(g(\mathbf{x}))) + \lambda r(\mathbf{z})$

(2) **denoising autoencoder** $\ell(\mathbf{x}, f(g(\tilde{\mathbf{x}})))$ where $\tilde{\mathbf{x}}$ is noise-corrupted version of $\mathbf{x}$

(3) **contractive autoencoder** $\rightarrow$ penalize derivatives: $r(\mathbf{z}, \mathbf{x}) = \sum_{j=1}^{m} \| \underbrace{\nabla_{\mathbf{x}} z_j}_{\frac{\partial g(\mathbf{x})}{\partial \mathbf{x}}} \|^2$. Function should not change much when $\mathbf{x}$ changes slightly.

Autoencoders can be used to **generate data** (such as images or text). Prominent example models are variational autoencoders (VAEs) or generative adverserial networks (GANs).

7 Discussion

+ 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)

8 Summary – Vanilla NNs (Feed-Forward NNs)

- Every layer is a **linear transformation** $W^{(l)}\mathbf{z}^{(l-1)} + \mathbf{b}^{(l)}$ plugged into a **non-linear** *activation function*. Typically the activation function operates element-wise on the input vector.

- Each activation function output (corresponding to one dimension in $\mathbf{z}^{(l)}$) is one **unit**. We typically apply the same activation function for every unit of the same layer. Often the same activation function is used across layers.

- The number of rows m_l in $W^{(l)}$ defines the number of units in layer l.

- The number of columns m_{l-1} in $W^{(l)}$ is defined by the number of input units to layer l (which is the number of units in the previous layer $l-1$).

- To train a FF-NN we use a version of **gradient descent** that updates the model parameters iteratively layer by layer starting with the last layer. For each parameter update we need a forward pass (forward propagation) and a backwards pass (back propagation).

- **Forward propagation** computes all $\mathbf{a}^{(l)}$'s and all $\mathbf{z}^{(l)}$'s for a given training input $\mathbf{x}_i$.
- **Back propagation** computes all the error contributions $\boldsymbol{\delta}^{(l)}$'s and the parameter updates $\Delta W^{(l)}$'s and $\Delta\mathbf{b}^{(l)}$'s.
- To use a FF-NN for **predictions** we perform a forward pass on the test input $\mathbf{x}_*$ all the way to the end to get $h(\mathbf{x}_*) = \hat{y}_*$.

Exercise 8.1. Practice Retrieving!
For this summary exercise, it is intended that your answers are based on **your own** (current) understanding of the concepts (and not on the definitions you read and copy from these notes or from elsewhere). Don't hesitate to **say it out loud** to your seat neighbor, your pet or stuffed animal, or to yourself before **writing it down**. After writing it down, check your answers with your (lecture)notes and the provided reading. Correct any mistakes you made. Research studies show that this practice of retrieval and phrasing out loud will help you retain the knowledge!

(a) Using *your own words*, repeat and explain each of the above summary points in 2-3 sentences and make sure you completely understand it. Write it down!

(b) Using *your own words*, justify each of the above discussion points in 2-3 sentences by retrieving the knowledge from the top of your head. Write it down!

(c) Now using your these lecture notes, our lecture slides, and your notes from class, check your justifications, correct them if necessary, and add anything that is missing!

And always remember: It's not bad to get it wrong. *Getting it wrong is part of learning!* Use your notes or other resources to get the correct answer or come to our office hours to get help!