# Machine Learning I (DATS 6202)
# Shallow Neural Network

Yuxiao Huang

Data Science, Columbian College of Arts & Sciences
George Washington University

Spring 2020

## Reference

- This set of slices was largely built on the following 8 wonderful books and a wide range of fabulous papers:

| | |
|---|---|
| HML | Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow (2nd Edition) |
| PML | Python Machine Learning (3rd Edition) |
| ESL | The Elements of Statistical Learning (2nd Edition) |
| PRML | Pattern Recognition and Machine Learning |
| LFD | Learning From Data |
| NND | Neural Network Design (2nd Edition) |
| NNDL | Neural Network and Deep Learning |
| RL | Reinforcement Learning: An Introduction |

- For most materials covered in the slides, we will specify their corresponding books and papers for further reference.

## Code Example & Case Study

- See code example of topics discussed in this section in github repository: /p1_c2_s7_shallow_neural_network/code_example
- See case study of Kaggle Competition related to this section in github repository: /p1_c2_s7_shallow_neural_network/case_study

## Overview

## Learning Objectives

- It is **expected** to understand
  - the idea of single-layer perceptron and perceptron learning rule
  - the idea of multi-layer perceptron and backpropagation
  - the pros and cons of batch, stochastic and mini-batch gradient descent
  - how to use sklearn single-layer perceptron and multi-layer perceptron
- It is **recommended** to understand
  - the math behind perception learning rule and backpropagation
  - how to implement single-layer perceptron (using perception learning rule) and multi-layer perceptron (using backpropagation and mini-batch gradient descent)

## Biological Neuron

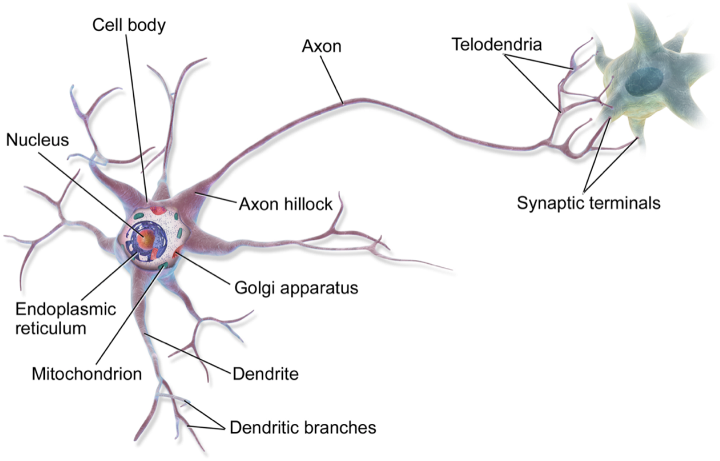

Figure 1: Biological neuron. Picture courtesy of *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow (2nd Edition)*.

## Single-Layer Perceptron (SLP)

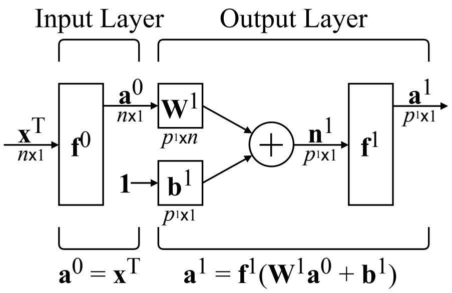

Figure 2: A single-layer perceptron.

Here:

- $\mathbf{x}^\mathsf{T}$ is the input of the input layer, with $n$ features and 1 sample
- $\mathbf{a}^1$ is the output of the output layer, with $p^1$ perceptrons

## Some Popular Activation Functions

- HardLimit:

$$HardLimit(n) = \begin{cases} 1, & \text{if} \quad n \geq 0, \\ 0, & \text{otherwise.} \end{cases} \tag{1}$$

- Rectified Linear Unit (ReLU):

$$ReLU(n) = \begin{cases} n, & \text{if} \quad n \geq 0, \\ 0, & \text{otherwise.} \end{cases} \tag{2}$$

- Linear:

$$Linear(n) = n. \tag{3}$$

- Sigmoid ($\sigma$):

$$\sigma(n) = \frac{1}{1 + e^{-n}}. \tag{4}$$

- Hyperbolic Tangent ($tanh$):

$$tanh(n) = 2\sigma(2n) – 1. \tag{5}$$

## The Logic AND Data

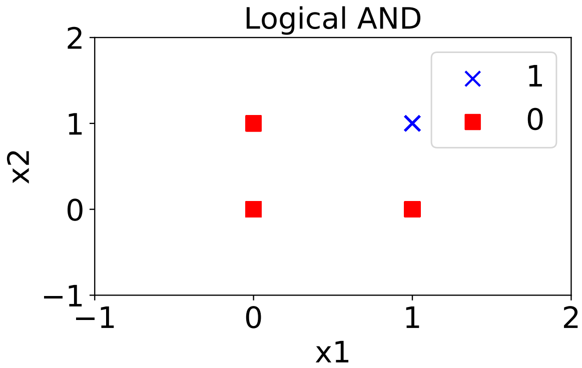

Figure 3: The logic AND data.

## A Corresponding SLP

- **Q:** Can you design a slp that perfectly separates the logic AND data?

## A Corresponding SLP

- **Q:** Can you design a slp that perfectly separates the logic AND data?

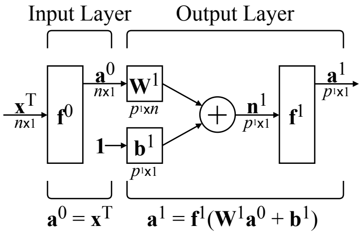

Figure 4: A single-layer perceptron.

- **A:** A slp in fig. 4 with the following parameter settings:
  - $\mathbf{w}^1 = \begin{bmatrix} 1 & 1 \end{bmatrix}$
  - $b^1 = -1.5$
  - $f^1 = HardLimit$

## The Logic OR Data

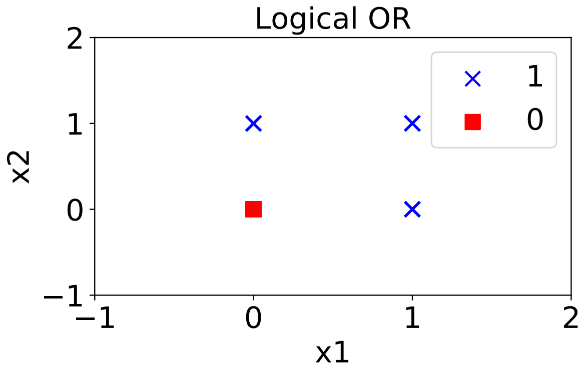

Figure 5: The logic OR data.

## A Corresponding SLP

- **Q:** Can you design a slp that perfectly separates the logic OR data?

## A Corresponding SLP

- **Q:** Can you design a slp that perfectly separates the logic OR data?

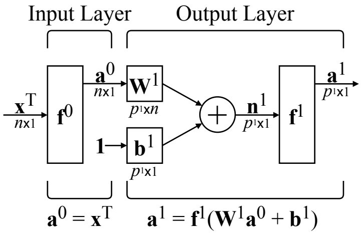

Figure 6: A single-layer perceptron.

- **A:** A slp in fig. 6 with the following parameter settings:
  - $\mathbf{w}^1 = \begin{bmatrix} 1 & 1 \end{bmatrix}$
  - $b^1 = -0.5$
  - $f^1 = HardLimit$

## The Motivation

- As shown in the previous two problems (logic AND and logic OR), designing the parameters of a slp manually may not be easy.
- Would not it be great if we can learn the parameters of slp automatically, as what we did in linear and logistic regression?
- It turns out that we can use the *Perceptron Learning Rule* to do so.

## Perceptron Learning Rule: The Idea + Math

- For each sample, the parameters of a slp are updated using the perceptron learning rule:

$$\mathbf{W}^1 = \mathbf{W}^1 + \eta(\mathbf{y} - \widehat{\mathbf{y}})\mathbf{x}, \tag{6}$$

$$\mathbf{b}^1 = \mathbf{b}^1 + \eta(\mathbf{y} - \widehat{\mathbf{y}}). \tag{7}$$

Here:

- $\mathbf{W}^1$ is the connecting weight between the output layer and input layer
- $\mathbf{b}^1$ is the bias of the output layer
- $\mathbf{y}$ is the target of the current sample
- $\widehat{\mathbf{y}}$ is the output of the output layer
- $\mathbf{x}$ is the current sample

## Further Reading

- See NND Chapter 4 for a very nice explanation of why the perceptron learning rule works, from a linear algebra perspective.
- See NND Chapter 4 for the proof of convergence for the perceptron learning rule (i.e., the proof that slp can always perfectly separate linearly-separable data).

## The Logic XOR Data

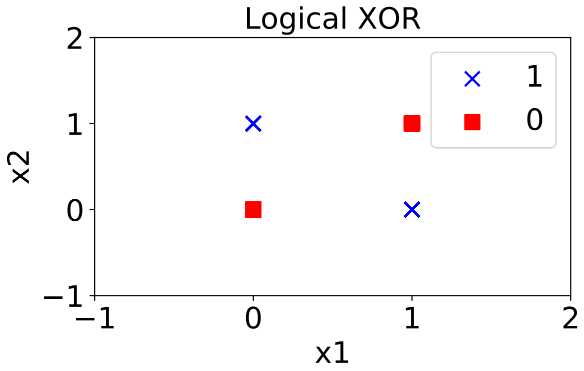

Figure 7: The logic XOR data.

## A Corresponding SLP

- **Q:** Can you design a slp that perfectly separates the logic XOR data?

.

## A Corresponding SLP

- **Q:** Can you design a slp that perfectly separates the logic XOR data?
- **A:** Unfortunately, no, since the data is not linearly-separable.

## Biological Neural Network

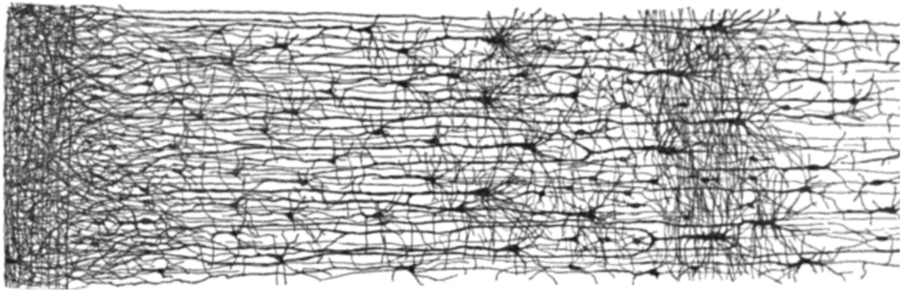

Figure 8: Biological neural network. Picture courtesy of *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow (2nd Edition)*.

## Multi-Layer Perceptron (MLP)

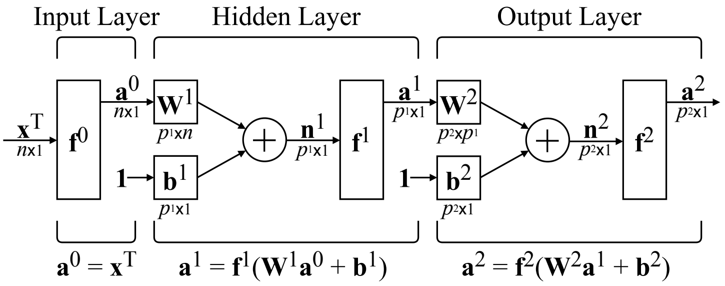

Figure 9: A fully connected three-layer mlp.

Here:

- $\mathbf{x}^\intercal$ is the input of the input layer, with $n$ features and 1 sample
- $\mathbf{a}^1$ is the output of the hidden layer, with $p^1$ perceptrons
- $\mathbf{a}^2$ is the output of the output layer, with $p^2$ perceptrons

## A MLP for the Logic XOR Data

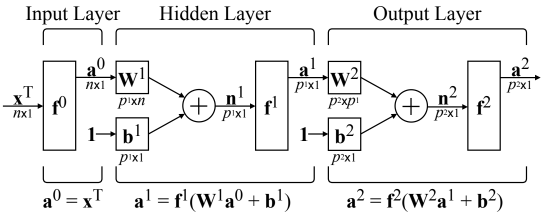

Figure 10: A fully connected three-layer mlp for the logic XOR data.

- A mlp in fig. 10 with the following parameter settings:
  - $\mathbf{w}^1 = \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix}$, $b^1 = \begin{bmatrix} -0.5 \\ -0.5 \end{bmatrix}$
  - $\mathbf{w}^2 = \begin{bmatrix} 1 & 1 \end{bmatrix}$, $b^2 = -0.5$
  - $f^1 = f^2 = HardLimit$

## MLP with Multiple Hidden Layers

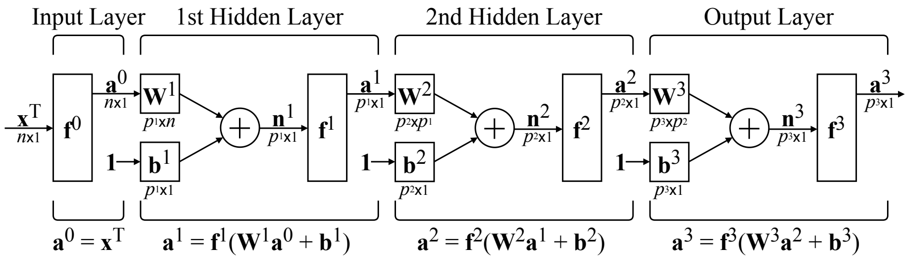

Figure 11: A fully connected four-layer mlp.

Here:

- $\mathbf{x}^\mathsf{T}$ is the input of the input layer, with $n$ features and 1 sample
- $\mathbf{a}^1$ is the output of the first hidden layer, with $p^1$ perceptrons
- $\mathbf{a}^2$ is the output of the second hidden layer, with $p^2$ perceptrons
- $\mathbf{a}^3$ is the output of the output layer, with $p^3$ perceptrons

## The Motivation

- Since manually designing the parameters of a slp is not easy, doing so for a mlp is much more difficult, if not impossible.
- Unfortunately, we cannot simply use perceptron learning rule to automatically train a mlp (we will see the reason later).
- Instead, we can use the *Backpropagation* to do so.

## The Idea

- Backpropagation is an iterative process.
- There are three steps (in order) in each iteration:
  1. forward pass: sends information from input layer up to higher layers
  2. backward pass: sends information from output layer down to lower layers
  3. gradient descent: updates the parameters using information obtained in the forward and backward pass

## Forward Pass: The Idea

- Forward pass works as follows:
  1. passes the input to the input layer
  2. from the first hidden layer up to the output layer, calculates the output of the current layer (e.g., layer $k$) from the output of the previous layer (e.g., layer $k - 1$)
- In forward pass, the net input and activation of each layer are preserved and reused in backward pass and gradient descent.

## Backward Pass: The Idea

- Backward pass works as follows:
  1. calculates the objective function (e.g., Mean Squared Error) based on the target and the output of the output layer
  2. calculates the *Sensitivity* (more on this later) of the output layer
  3. from the last hidden layer down to the first hidden layer, calculates the sensitivity of the current layer (e.g., layer $k$) from the sensitivity of the next layer (e.g., layer $k+1$)
- In backward pass, the sensitivity of each layer is preserved and reused in gradient descent

## Gradient Descent: The Idea

- Gradient descent works as follows:
  1. uses the output (obtained in forward pass) and sensitivity (obtained in backward pass) of each layer to update the parameters of the layer

## The Math

- Let us take a look at the math underlying the three steps in backpropagation (forward pass, backward pass and gradient descent)
- The order in which the steps work:
  1. forward pass
  2. backward pass
  3. gradient descent
- The order in which we will discuss the steps (which better explains the dependence between the steps):
  1. gradient descent
  2. forward pass
  3. backward pass

## Batch / Stochastic / Mini-Batch Gradient Descent

- Based on the amount of data that we use in each iteration of updating the parameters, we can divide gradient descent into three categories:
  - *batch gradient descent*: where we use all the training data
  - *stochastic gradient descent*: where we use a sample in the training data
  - *mini-batch gradient descent*: where we use a subset (with more than one sample) of the training data
- We can think of batch gradient descent / stochastic gradient descent as two extremes (using as many / as few data as possible).
- In that sense, mini-batch lies between the two extremes.

## Batch Gradient Descent

- pros:
  - utilizes parallel computing to the most extent (making it fast)
  - does not require shuffling the data (making it even faster)
- cons:
  - requires loading all the training data into the memory (not suitable for large datasets)
  - requires processing all the training data to complete one update of the parameters (not suitable for large datasets)
- Batch gradient descent is suitable for small datasets.

## Stochastic Gradient Descent

- pros:
  - allows loading only some training data into the memory (suitable for large datasets)
  - allows processing only one sample in the training data to complete one update of the parameters (suitable for large datasets)
- cons:
  - utilizes parallel computing to the least extent (making it slow)
  - requires shuffling the data in the beginning of each epoch (making it even slower)
- Stochastic gradient descent is suitable for large datasets.

## Mini-Batch Gradient Descent

- pros:
  - allows loading only some training data into the memory (suitable for large datasets)
  - allows processing only some training data to complete one update of the parameters (suitable for large datasets)
- cons:
  - utilizes parallel computing to some extent (making it faster than stochastic gradient descent but slower than batch gradient descent)
  - requires shuffling the data in the beginning of each epoch (making it even slower than batch gradient descent)
- Mini-batch gradient descent is more suitable for large datasets (compared to stochastic gradient descent).
- **Good Practice**: It is recommended to use small batches (from 2 to 32) in mini-batch gradient descent [1].

## The Objective Function

- While mini-batch gradient descent usually works better for large datasets, here we use stochastic gradient descent for mathematical convenience.
- Since stochastic gradient descent processes one sample at a time, the objective function (Mean Squared Error), $J$, can be written as

$$J = (\mathbf{y} - \widehat{\mathbf{y}})^2. \tag{8}$$

Here $\widehat{\mathbf{y}}$ is the output of the output layer:

$$\widehat{\mathbf{y}} = \mathbf{a}^l = \mathbf{f}^l(\mathbf{n}^l) = \mathbf{f}^l(\mathbf{W}^l \mathbf{a}^{l-1} + \mathbf{b}^l), \tag{9}$$

where $l$ is the index of the last layer and $\mathbf{a}^l$ the output of the output layer.

## Gradient Descent: The Math

- Using gradient descent, the weight with respect to layer $k$, $\mathbf{W}^k$, can be updated as

$$\mathbf{W}^k = \mathbf{W}^k - \eta \frac{\partial J}{\partial \mathbf{W}^k}. \tag{10}$$

- We cannot directly calculate the gradient in eq. (10) since, as shown in eqs. (8) and (9), while $J$ is an explicit function of weights on the output layer, $\mathbf{W}^l$, it is not an explicit function of weights on the hidden layers, $\mathbf{W}^k$ (where $k < l$).
- This is why we cannot use the perceptron learning rule for mlp, as we did for slp (where there is no hidden layer).
- This is also why we use the chain rule to rewrite eq. (10):

$$\mathbf{W}^k = \mathbf{W}^k - \eta \frac{\partial J}{\partial \mathbf{n}^k} \frac{\partial \mathbf{n}^k}{\partial \mathbf{W}^k}, \tag{11}$$

where $\mathbf{n}^k$ is the net input of layer $k$ (and an explicit function of $\mathbf{W}^k$):

$$\mathbf{n}^k = \mathbf{W}^k \mathbf{a}^{k-1} + \mathbf{b}^k. \tag{12}$$

## Gradient Descent: The Math

- We define the first derivation in eq. (11), $\frac{\partial J}{\partial \mathbf{n}^k}$, as the *Sensitivity* of $J$ to the changes of $\mathbf{n}^k$, denoted by $\mathbf{s}^k$:

$$\mathbf{s}^k = \frac{\partial J}{\partial \mathbf{n}^k}. \tag{13}$$

- Based on eq. (12), we rewrite the second derivation in eq. (11), $\frac{\partial \mathbf{n}^k}{\partial \mathbf{W}^k}$, as

$$\frac{\partial \mathbf{n}^k}{\partial \mathbf{W}^k} = \frac{\partial (\mathbf{W}^k \mathbf{a}^{k-1} + \mathbf{b}^k)}{\partial \mathbf{W}^k} = \left(\mathbf{a}^{k-1}\right)^\intercal. \tag{14}$$

## Gradient Descent: The Math

- By substituting eqs. (13) and (14) into eq. (11)

$$\mathbf{W}^k = \mathbf{W}^k - \eta \frac{\partial J}{\partial \mathbf{n}^k} \frac{\partial \mathbf{n}^k}{\partial \mathbf{W}^k},$$

  we can update the weight as

$$\mathbf{W}^k = \mathbf{W}^k - \eta \mathbf{s}^k \left(\mathbf{a}^{k-1}\right)^\intercal. \tag{15}$$

- Similarly, we can update the bias as

$$\mathbf{b}^k = \mathbf{b}^k - \eta \mathbf{s}^k. \tag{16}$$

- We use forward pass to calculate $\mathbf{a}^{k-1}$ in eq. (15).
- We use backward pass to calculate $\mathbf{s}^k$ in eqs. (15) and (16).

## Gradient Descent: The Idea + Math

- Gradient descent works as follows:
  1. uses eq. (15), the output (obtained in forward pass) and sensitivity (obtained in backward pass) of each layer to update the parameters of the layer:

$$\begin{aligned}\mathbf{W}^k &= \mathbf{W}^k - \eta \mathbf{s}^k \left(\mathbf{a}^{k-1}\right)^\intercal,\\ \mathbf{b}^k &= \mathbf{b}^k - \eta \mathbf{s}^k.\end{aligned}$$

## Forward Pass: The Math

- The key in forward pass is a recursive relationship between the output of two consecutive layers.
- Since the output of layer $k$, $\mathbf{a}^k$, is

$$\mathbf{a}^k = \mathbf{f}^k(\mathbf{n}^k), \tag{17}$$

  where $\mathbf{f}^k$ is the activation function of layer $k$ and $\mathbf{n}^k$ the net input of layer $k$, given in eq. (12)

$$\mathbf{n}^k = \mathbf{W}^k \mathbf{a}^{k-1} + \mathbf{b}^k,$$

  we can rewrite eq. (17) as

$$\mathbf{a}^k = \mathbf{f}^k(\mathbf{n}^k) = \mathbf{f}^k(\mathbf{W}^k \mathbf{a}^{k-1} + \mathbf{b}^k). \tag{18}$$

- Eq. (18) shows that the output of layer $k$, $\mathbf{a}^k$, relies on the output of layer $k-1$, $\mathbf{a}^{k-1}$.
- This is why we use *forward* pass to calculate the output from lower layers (e.g., layer $k-1$) up to higher layers (e.g., layer $k$).

## Forward Pass: The Idea + Math

- Forward pass works as follows:
  1. passes the input to the input layer:

$$\mathbf{a}^0 = \mathbf{x}^\intercal.$$

  2. from the first hidden layer (layer 1) up to the output layer (layer $l$), uses eq. (18) to calculate the output of the current layer (e.g., layer $k$) from the output of the previous layer (e.g., layer $k-1$):

$$\mathbf{a}^k = \mathbf{f}^k(\mathbf{n}^k) = \mathbf{f}^k(\mathbf{W}^k\mathbf{a}^{k-1} + \mathbf{b}^k), \quad \text{where} \quad 1 \leq k \leq l.$$

## Backward Pass: The Math

- Similar to forward pass, the key in backward pass is also a recursive relationship.
- Unlike forward pass where the relationship is between *output* of two consecutive layers, the relationship in backward pass is between *sensitivity* of two consecutive layers.
- Concretely, by applying the chain rule to eq. (13)

$$\mathbf{s}^k = \frac{\partial J}{\partial \mathbf{n}^k},$$

  we have

$$\mathbf{s}^k = \frac{\partial J}{\partial \mathbf{n}^k} = \left(\frac{\partial \mathbf{n}^{k+1}}{\partial \mathbf{n}^k}\right)^\intercal \frac{\partial J}{\partial \mathbf{n}^{k+1}} = \left(\frac{\partial \mathbf{n}^{k+1}}{\partial \mathbf{n}^k}\right)^\intercal \mathbf{s}^{k+1}. \tag{19}$$

- Eq. (19) shows that the sensitivity of layer $k$, $\mathbf{s}^k$, relies on the sensitivity of layer $k+1$, $\mathbf{s}^{k+1}$.
- This is why we use *backward* pass to calculate the sensitivity from higher layers (e.g., layer $k+1$) down to lower layers (e.g., layer $k$).

## The Sensitivity

- Based on eq. (13)

$$\mathbf{s}^k = \frac{\partial J}{\partial \mathbf{n}^k},$$

the sensitivity of the output layer (with index $l$), $\mathbf{s}^l$, is

$$\mathbf{s}^l = \frac{\partial J}{\partial \mathbf{n}^l}. \tag{20}$$

- By substituting eqs. (8) and (9)

$$J = (\mathbf{y} - \widehat{\mathbf{y}})^2 \quad \text{and} \quad \widehat{\mathbf{y}} = \mathbf{a}^l = \mathbf{f}^l(\mathbf{n}^l)$$

into eq. (20), we have

$$\mathbf{s}^l = \frac{\partial J}{\partial \mathbf{n}^l} = \frac{\partial (\mathbf{y} - \widehat{\mathbf{y}})^2}{\partial \mathbf{n}^l} = -2(\mathbf{y} - \widehat{\mathbf{y}}) \frac{\partial \mathbf{f}^l(\mathbf{n}^l)}{\partial \mathbf{n}^l} = -2(\mathbf{y} - \widehat{\mathbf{y}}) \dot{\mathbf{f}}^l(\mathbf{n}^l). \tag{21}$$

## The Jacobian Matrix

- The last piece in backward pass is the derivation in eq. (19)

$$\mathbf{s}^k = \left(\frac{\partial \mathbf{n}^{k+1}}{\partial \mathbf{n}^k}\right)^\intercal \mathbf{s}^{k+1}.$$

The transpose of the derivation is a Jacobian matrix:

$$\frac{\partial \mathbf{n}^{k+1}}{\partial \mathbf{n}^k} = \begin{bmatrix} \frac{\partial n_1^{k+1}}{\partial n_1^k} & \frac{\partial n_1^{k+1}}{\partial n_2^k} & \cdots & \frac{\partial n_1^{k+1}}{\partial n_{p^k}^k} \\ \frac{\partial n_2^{k+1}}{\partial n_1^k} & \frac{\partial n_2^{k+1}}{\partial n_2^k} & \cdots & \frac{\partial n_2^{k+1}}{\partial n_{p^k}^k} \\ \vdots & \vdots & & \vdots \\ \frac{\partial n_{p^{k+1}}^{k+1}}{\partial n_1^k} & \frac{\partial n_{p^{k+1}}^{k+1}}{\partial n_2^k} & \cdots & \frac{\partial n_{p^{k+1}}^{k+1}}{\partial n_{p^k}^k} \end{bmatrix}. \quad (22)$$

Here:

- entry $(i, j)$ in the matrix is $\frac{\partial n_i^{k+1}}{\partial n_j^k}$
- $n_i^{k+1}$ is the net input of the $i^{th}$ perceptron on layer $k+1$
- $n_j^k$ is the net input of the $j^{th}$ perceptron on layer $k$
- $p^{k+1}$ and $p^k$ are the number of perceptrons on layer $k+1$ and $k$

## The Jacobian Matrix

- In order to derive the equation of entry $(i, j)$ in the Jacobian matrix, $\frac{\partial n_i^{k+1}}{\partial n_j^k}$, we first derive the equation of the net input of the $i^{th}$ perceptron on layer $k+1$, $n_i^{k+1}$.
- Since the $i^{th}$ perceptron on layer $k+1$ is fully connected with all the $p^k$ perceptrons on layer $k$, we can write $n_i^{k+1}$ as

$$n_i^{k+1} = \sum_{q=1}^{p^k} w_{iq}^{k+1} a_q^k + b_i^{k+1}. \tag{23}$$

Here:

  - $w_{iq}^{k+1}$ is the connecting weight between the $i^{th}$ perceptron on layer $k+1$ and the $q^{th}$ perceptron on layer $k$
  - $a_q^k$ is the output of the $q^{th}$ perceptron on layer $k$
  - $b_i^{k+1}$ is the bias of the $i^{th}$ perceptron on layer $k+1$

## The Jacobian Matrix

- By substituting eq. (23) into $\frac{\partial n_i^{k+1}}{\partial n_j^k}$, we have

$$\frac{\partial n_i^{k+1}}{\partial n_j^k} = \frac{\partial \left( \sum_{q=1}^{p^k} w_{iq}^{k+1} a_q^k + b_i^{k+1} \right)}{\partial n_j^k}. \tag{24}$$

- Since $b_i^{k+1}$ (the bias of the $i^{th}$ perceptron on layer $k+1$) is a scaler, we can rewrite eq. (24) as

$$\frac{\partial n_i^{k+1}}{\partial n_j^k} = \frac{\partial \sum_{q=1}^{p^k} w_{iq}^{k+1} a_q^k}{\partial n_j^k}. \tag{25}$$

## The Jacobian Matrix

- The output of the $q^{th}$ perceptron on layer $k$, $a_q^k$, is

$$a_q^k = f_q^k(n_q^k), \tag{26}$$

  where $f_q^k$ and $n_q^k$ are the activation function and net input of the $q^{th}$ perceptron on layer $k$.
- Based on eq. (26), $a_q^k$ is only related to $n_j^k$ when $q = j$, we can rewrite eq. (25) as

$$\frac{\partial n_i^{k+1}}{\partial n_j^k} = \frac{\partial w_{ij}^{k+1} a_j^k}{\partial n_j^k} = w_{ij}^{k+1} \frac{\partial f_j^k(n_j^k)}{\partial n_j^k} = w_{ij}^{k+1} \dot{f}_j^k(n_j^k). \tag{27}$$

## The Jacobian Matrix

- By substituting eq. (27) into the Jacobian matrix in eq. (22)

$$\frac{\partial \mathbf{n}^{k+1}}{\partial \mathbf{n}^k} = \begin{bmatrix} \frac{\partial n_1^{k+1}}{\partial n_1^k} & \frac{\partial n_1^{k+1}}{\partial n_2^k} & \cdots & \frac{\partial n_1^{k+1}}{\partial n_{p^k}^k} \\ \frac{\partial n_2^{k+1}}{\partial n_1^k} & \frac{\partial n_2^{k+1}}{\partial n_2^k} & \cdots & \frac{\partial n_2^{k+1}}{\partial n_{p^k}^k} \\ \vdots & \vdots & & \vdots \\ \frac{\partial n_{p^{k+1}}^{k+1}}{\partial n_1^k} & \frac{\partial n_{p^{k+1}}^{k+1}}{\partial n_2^k} & \cdots & \frac{\partial n_{p^{k+1}}^{k+1}}{\partial n_{p^k}^k} \end{bmatrix},$$

we have

$$\frac{\partial \mathbf{n}^{k+1}}{\partial \mathbf{n}^k} = \mathbf{W}^{k+1}\dot{\mathbf{F}}^k(\mathbf{n}^k), \tag{28}$$

where $\dot{\mathbf{F}}^k(\mathbf{n}^k)$ is a diagonal matrix:

$$\dot{\mathbf{F}}^k(\mathbf{n}^k) = \begin{bmatrix} \dot{f}_1^k(n_1^k) & 0 & \cdots & 0 \\ 0 & \dot{f}_2^k(n_2^k) & \cdots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \cdots & \dot{f}_{p^k}^k(n_{p^k}^k) \end{bmatrix}. \tag{29}$$

## The Jacobian Matrix

- By substituting eq. (28) into eq. (19)

$$\mathbf{s}^k = \left(\frac{\partial \mathbf{n}^{k+1}}{\partial \mathbf{n}^k}\right)^{\mathsf{T}} \mathbf{s}^{k+1},$$

we have

$$\mathbf{s}^k = \dot{\mathbf{F}}^k(\mathbf{n}^k) \left(\mathbf{W}^{k+1}\right)^{\mathsf{T}} \mathbf{s}^{k+1}, \tag{30}$$

where $\dot{\mathbf{F}}^k(\mathbf{n}^k)$ is given in eq. (29)

$$\dot{\mathbf{F}}^k(\mathbf{n}^k) = \begin{bmatrix} \dot{f}_1^k(n_1^k) & 0 & \cdots & 0 \\ 0 & \dot{f}_2^k(n_2^k) & \cdots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \cdots & \dot{f}_{p^k}^k(n_{p^k}^k) \end{bmatrix}.$$

## Backward Pass: The Idea + Math

- Backward pass works as follows:
  1. calculates the objective function (Mean Squared Error) based on the target and the output of the output layer using eq. (8):
  $$J = (\mathbf{y} - \widehat{\mathbf{y}})^2.$$
  2. calculates the sensitivity of the output layer (i.e., the last layer with index $l$) using eq. (21):
  $$\mathbf{s}^l = \frac{\partial J}{\partial \mathbf{n}^l} = -2(\mathbf{y} - \widehat{\mathbf{y}})\dot{\mathbf{f}}^l(\mathbf{n}^l).$$
  3. from the last hidden layer down to the first hidden layer, uses eq. (30) to calculate the sensitivity of the current layer (e.g., layer $k$) from the sensitivity of the next layer (e.g., layer $k+1$):
  $$\mathbf{s}^k = \dot{\mathbf{F}}^k(\mathbf{n}^k)\left(\mathbf{W}^{k+1}\right)^\intercal \mathbf{s}^{k+1}, \quad \text{where} \quad l-1 \geq k \geq 1.$$

## Summary

1. Forward pass
   1. passes the input to the input layer:
   $$\mathbf{a}^0 = \mathbf{x}^\intercal.$$
   2. calculates the output from the first hidden layer up to the output layer:
   $$\mathbf{a}^k = \mathbf{f}^k(\mathbf{n}^k) = \mathbf{f}^k(\mathbf{W}^k \mathbf{a}^{k-1} + \mathbf{b}^k), \quad \text{where} \quad 1 \leq k \leq l.$$
2. Backward pass
   1. calculates the sensitivity of the output layer:
   $$\mathbf{s}^l = \frac{\partial J}{\partial \mathbf{n}^l} = -2(\mathbf{y} - \widehat{\mathbf{y}})\dot{\mathbf{f}}^l(\mathbf{n}^l).$$
   2. calculates the sensitivity from the last hidden layer down to the first:
   $$\mathbf{s}^k = \dot{\mathbf{F}}^k(\mathbf{n}^k)\left(\mathbf{W}^{k+1}\right)^\intercal \mathbf{s}^{k+1}, \quad \text{where} \quad l-1 \geq k \geq 1.$$
3. Gradient descent
   1. updates the parameters of each layer:
   $$\mathbf{W}^k = \mathbf{W}^k - \eta \mathbf{s}^k \left(\mathbf{a}^{k-1}\right)^\intercal,$$
   $$\mathbf{b}^k = \mathbf{b}^k - \eta \mathbf{s}^k.$$

## Example

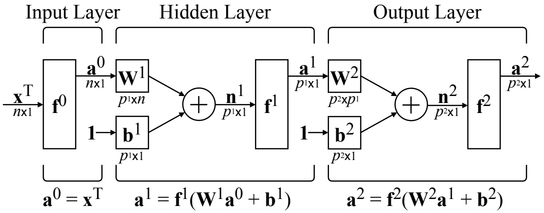

Figure 12: A fully connected three-layer mlp.

- Here is the first iteration of backpropagation on mlp in fig. 12, where:
  - $\mathbf{x}^\intercal = \begin{bmatrix}1\\1\end{bmatrix}$, $y = 0$, $n = 2$
  - $\mathbf{W}^1 = \begin{bmatrix}1 & 1\\1 & 1\end{bmatrix}$, $\mathbf{b}^1 = \begin{bmatrix}0\\0\end{bmatrix}$, $p^1 = 2$
  - $\mathbf{w}^2 = \begin{bmatrix}1 & 1\end{bmatrix}$, $b^2 = 0$, $p^2 = 1$
  - $\mathbf{f}^1 = \mathbf{ReLU}$, $f^2 = Linear$

## Forward Pass

- Passes the input to the input layer:

$$\mathbf{a}^0 = \mathbf{x}^\intercal = \begin{bmatrix}1\\1\end{bmatrix}. \tag{31}$$

- Calculates the output of the hidden layer:

$$\mathbf{n}^1 = \mathbf{W}^1\mathbf{a}^0 + \mathbf{b}^1 = \begin{bmatrix}1 & 1\\1 & 1\end{bmatrix}\begin{bmatrix}1\\1\end{bmatrix} + \begin{bmatrix}0\\0\end{bmatrix} = \begin{bmatrix}2\\2\end{bmatrix}, \tag{32}$$

$$\mathbf{a}^1 = \mathbf{f}^1(\mathbf{n}^1) = \mathbf{ReLU}\left(\begin{bmatrix}2\\2\end{bmatrix}\right) = \begin{bmatrix}2\\2\end{bmatrix}. \tag{33}$$

- Calculates the output of the output layer:

$$n^2 = \mathbf{w}^2\mathbf{a}^1 + b^2 = \begin{bmatrix}1 & 1\end{bmatrix}\begin{bmatrix}2\\2\end{bmatrix} + 0 = 4, \tag{34}$$

$$\widehat{y} = a^2 = f^2(n^2) = Linear(4) = 4. \tag{35}$$

## Backward Pass

- Calculate the sensitivity of the output layer:

$$\dot{f}^2(n^2) = \dot{Linear}(4) = 1, \tag{36}$$

$$s^2 = -2(y - \widehat{y})\dot{f}^2(n^2) = -2 \times (0 - 4) \times 1 = 8. \tag{37}$$

- Calculate the sensitivity of the hidden layer:

$$\dot{\mathbf{F}}^1(\mathbf{n}^1) = \begin{bmatrix} \dot{f}_1^1(n_1^1) & 0 \\ 0 & \dot{f}_2^1(n_2^1) \end{bmatrix} = \begin{bmatrix} \dot{ReLU}(2) & 0 \\ 0 & \dot{ReLU}(2) \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \tag{38}$$

$$\mathbf{s}^1 = \dot{\mathbf{F}}^1(\mathbf{n}^1) \left(\mathbf{w}^2\right)^\intercal s^2 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} \times 8 = \begin{bmatrix} 8 \\ 8 \end{bmatrix}. \tag{39}$$

## Gradient Descent

- Update the parameters of the hidden layer (with $\eta = 0.1$):

$$\mathbf{W}^1 = \mathbf{W}^1 - \eta \mathbf{s}^1 \left(\mathbf{a}^0\right)^\intercal = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} - 0.1 \times \begin{bmatrix} 8 \\ 8 \end{bmatrix} \begin{bmatrix} 1 & 1 \end{bmatrix} = \begin{bmatrix} 0.2 & 0.2 \\ 0.2 & 0.2 \end{bmatrix}, \tag{40}$$

$$\mathbf{b}^1 = \mathbf{b}^1 - \eta \mathbf{s}^1 = \begin{bmatrix} 0 \\ 0 \end{bmatrix} - 0.1 \times \begin{bmatrix} 8 \\ 8 \end{bmatrix} = \begin{bmatrix} -0.8 \\ -0.8 \end{bmatrix}. \tag{41}$$

- Update the parameters of the output layer (with $\eta = 0.1$):

$$\mathbf{w}^2 = \mathbf{w}^2 - \eta s^2 \left(\mathbf{a}^1\right)^\intercal = \begin{bmatrix} 1 & 1 \end{bmatrix} - 0.1 \times 8 \times \begin{bmatrix} 2 & 2 \end{bmatrix} = \begin{bmatrix} -0.6 & -0.6 \end{bmatrix}, \tag{42}$$

$$b^2 = b^2 - \eta s^2 = 0 - 0.1 \times 8 = -0.8. \tag{43}$$

## References

D. Masters and C. Luschi.
Revisiting Small Batch Training for Deep Neural Networks.
*arXiv preprint arXiv:1804.07612*, 2018.