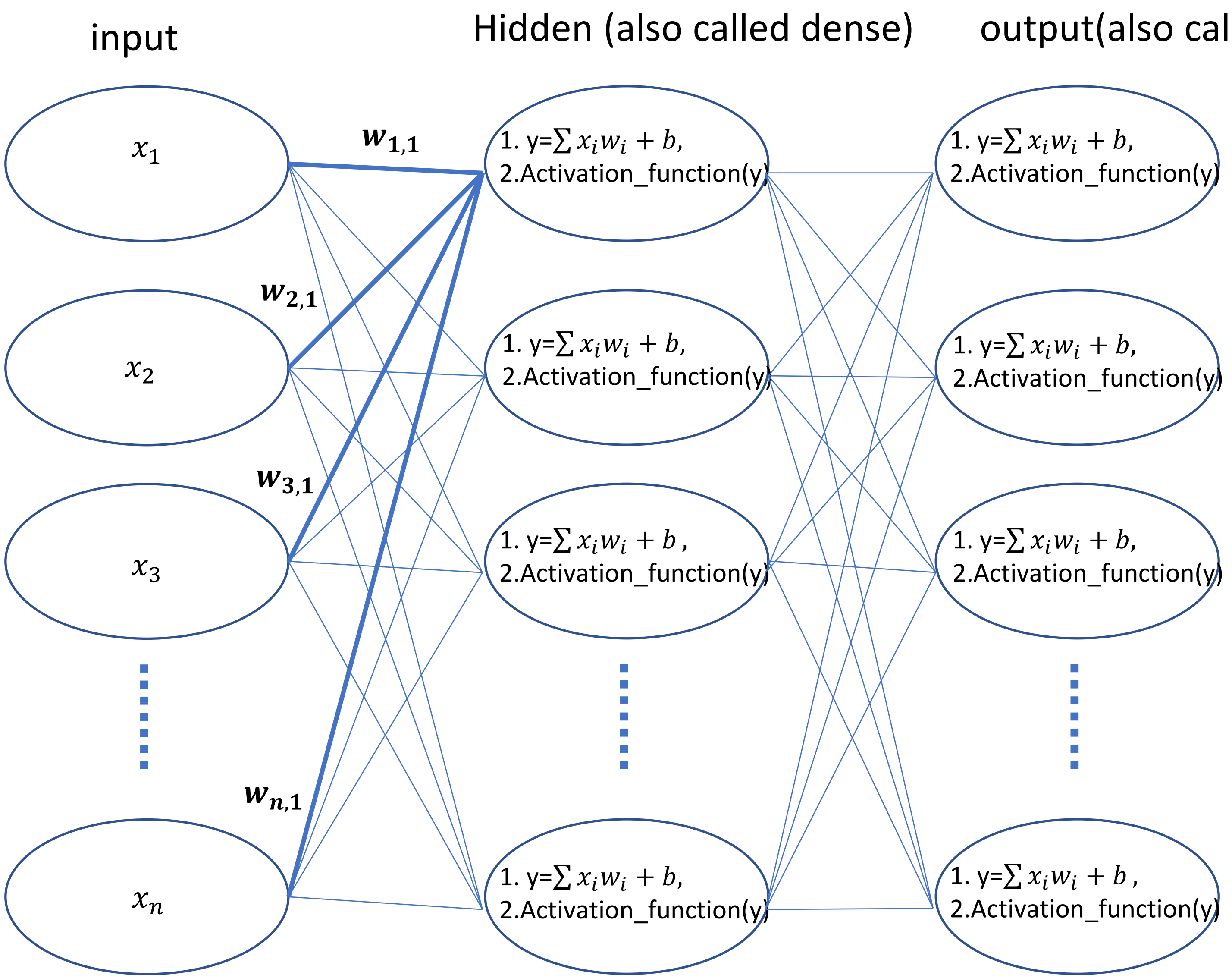
ANN with TensorFlow: Theory and Practice

Junghwan Lee, Ph.D. (jhrrlee@gmail.com)

2^{nd} lecture

- Structure of Neural Networks and Neurons
- Gradient descent
- Code: Custom_model




```
model = keras.Sequential()
Model.add(layers.Dense(n)
Model.add(layers.Dense(n, activation='relu’))
Model.add(layers.Dense(n, activation='relu’))
.
.
.
Model.add(layers.Dense(n, activation='relu’))
```

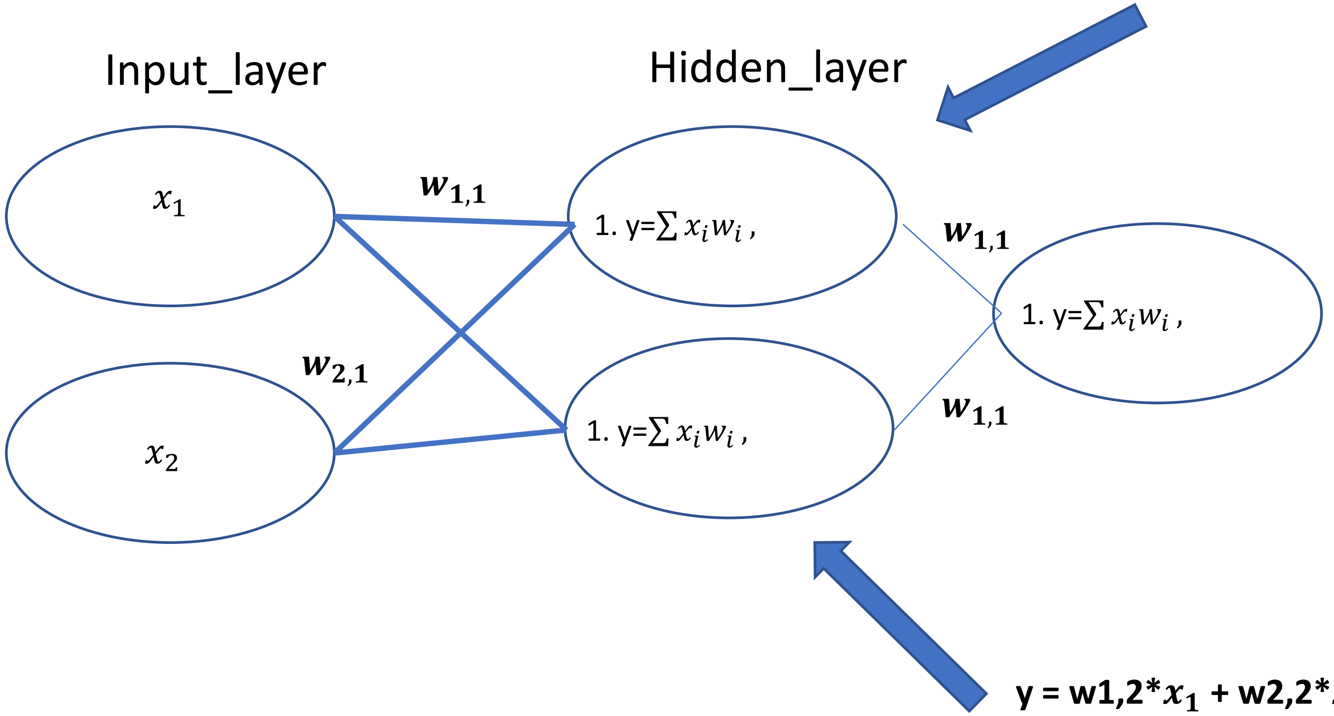
Activation function

- RelU
- Sigmoid
- Hyperbolic tangent
- Leaky ReLU
- Softmax

We will make a neural network has two inputs, one output, and
One hidden layer
We do not use an activation function in this example.

```
input_layer = keras.Input(shape=(2,))
hidden_layer = keras.layers.Dense(2)(input_layer)
output_layer = keras.layers.Dense(1)(hidden_layer)
```

y = w1,1*x_1 + w2,1*x_2 + bias



y = w1,1*x_1 + w2,1*x_2 + bias

Every weight needs to have a layer index.
I expect readers understand that weights in each layer are different.

y = w1,2*x_1 + w2,2*x_2 + bias

Update weights

- Training is the updating of weights.
- To update weights, Gradient descent is used.
- **Gradient descent** is the same way to solve a regression problem in the last lecture.
- I will explain **gradient descent** with an example and code.
- Learning methods
  - Hebbian Learning (Hebbian Rule)
  - Perceptron Rule
  - **Gradient Descent** (Delta Rule, **Least Mean Square**)
  - Back propagation

data

x	y	z
1	1	7
2	2	12
2	2	12
3	3	17
4	4	22
5	5	27
6	21	77
7	1	19
1	2	10
1	3	13
2	4	18
3	5	23
4	6	28
5	7	33
6	2	20
7	1	19
1	2	10
2	3	15
3	4	20

We will make a neural network that will find the z value with the given x, y.

This is an example.
The equation ("z=x*2+y*3+2") makes the data.
We can easily verify if our neural network is learning well.

Although this is a simple example, we can easily extend it to solve many regression problems with various inputs.
I did not use scaling of the data set for beginner's understanding.
However, in practice, we need to use scaling of the data set.

Code (https://github.com/jhrrlee/mlcode.git) for the custom model is written for low-level handling of losses tracked by the model from a reference to the TensorFlow doc (https://www.tensorflow.org/guide/keras/writing_a_training_loop_from_scratch).

Handling low-level is better for studying and seeing operations during learning.

We consider gradient descent with Least Mean Squre using Mean Squre Error $E(\hat{Y}_i - Y_i)^2 = \frac{1}{n}\sum(\hat{Y}_i - Y_i)^2$

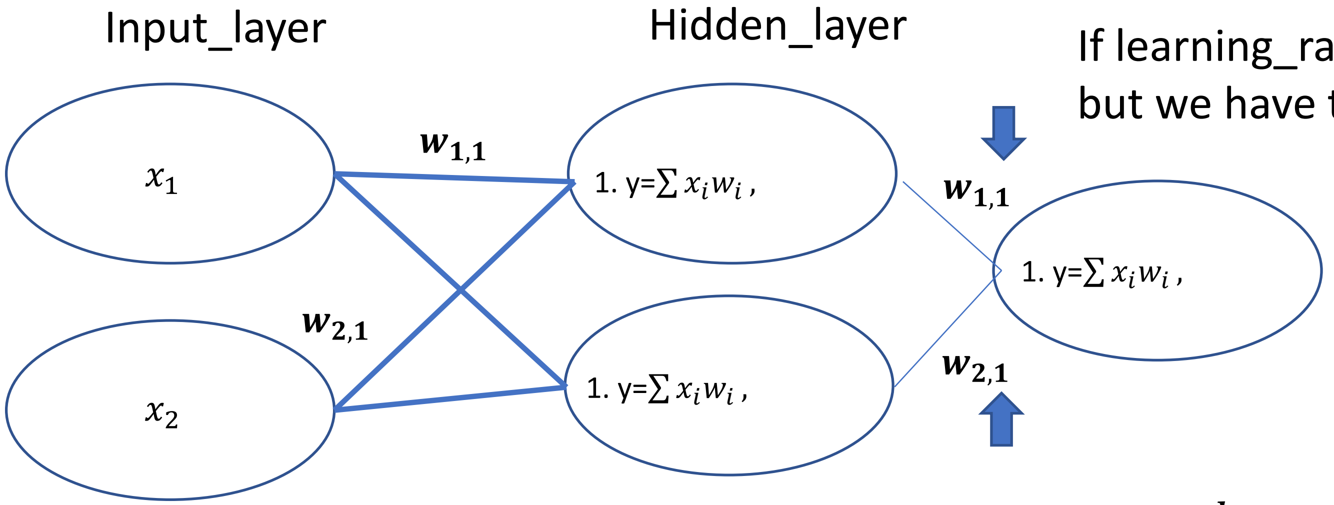
We can use any loss function other than MSE; it is just for convenience to use mathematical theory.
However, mathematically, it is impossible to calculate the minimum error since the result is experimental result ($\hat{y}_i$).
This is an important point that we may find some better ways in experiments.
We can try

We learned this in the last lecture.
Learning sequence:

1. Get loss value
2. Partial derivative to update weights (weights between the input layer and hidden layer can be calculated by chain rule

$\boldsymbol{w_{1,1}} = \boldsymbol{w_{1,1}} + learning\_rate * \frac{\partial f}{\partial \boldsymbol{w_{1,1}}}$, f = Loss_function,

If learning_rate has large value then learning _speed will be fast, but we have to consider diffusion.



$\hat{y}_i$ = $\boldsymbol{w_{1,1}} * \boldsymbol{x_1} + \boldsymbol{w_{2,1}} * \boldsymbol{x_2}$ + **bias**

Loss_function = $\frac{1}{2m}\sum(\hat{y}_i - z_i)^2$: z_i in data set

$\boldsymbol{w_{2,1}} = \boldsymbol{w_{2,1}} + learning\_rate * \frac{\partial f}{\partial \boldsymbol{w_{2,1}}}$

$\frac{1}{2}$ in $\frac{1}{2m}$ is because of convenience of calculation of derivative?
BTW, it will not be important because it depends on $learninrate$ again in $learning\_rate * \frac{\partial f(x_1,x_2)}{\partial x_1}$

Reference:
Logistic regression (https://en.wikipedia.org/wiki/Logistic_regression)
Least mean square (https://en.wikipedia.org/wiki/Least_mean_squares_filter)

Summary

- E$(\hat{Y}_i - Y_i)^2$ =$\frac{1}{n}\sum(\hat{Y}_i - Y_i)^2$
- *Chain rule:* $\frac{\partial \mathrm{y}}{\partial \boldsymbol{z}} = \frac{\partial \mathrm{y}}{\partial \boldsymbol{x}}\frac{\partial x}{\partial \boldsymbol{z}}$, $y = x^2$, $x = z^2$
- $\hat{y}_i$ **=** $\boldsymbol{w}_{\mathbf{1},\mathbf{1}}$*****$\boldsymbol{x}_{\mathbf{1}}$ **+** $\boldsymbol{w}_{\mathbf{2},\mathbf{1}}$*****$\boldsymbol{x}_{\mathbf{2}}$ **+ bias**
- $\frac{\partial \mathrm{Loss\_function}}{\partial \boldsymbol{w}_{\mathbf{1},\mathbf{1}}} = \frac{\partial}{\partial \boldsymbol{w}_{\mathbf{1},\mathbf{1}}}\left(\frac{1}{2m}\sum(\hat{y}_i - z_i)^2\right)$
- $\frac{\partial \mathrm{Loss\_function}}{\partial \boldsymbol{w}_{\boldsymbol{i}}} = \frac{\partial}{\partial \boldsymbol{w}_{\mathbf{1},\mathbf{1}}}\left(\frac{1}{2m}\sum\left(\left(\boldsymbol{w}_{\boldsymbol{i}}*\boldsymbol{x}_{\mathbf{1},\boldsymbol{i}} + \boldsymbol{w}_{\boldsymbol{i}}*\boldsymbol{x}_{\mathbf{2},\boldsymbol{i}} + \mathbf{bias}\right) - z_i\right)^2\right)$, z_i, $\boldsymbol{x}_{\mathbf{1},\boldsymbol{i}}$, and$\boldsymbol{x}_{\mathbf{2},\boldsymbol{i}}$ are just constant from the given data. $\boldsymbol{w}_{\boldsymbol{i}}$ is each weight in iterations. $\boldsymbol{w}_{\mathbf{1},\mathbf{1},\boldsymbol{i}}$ will be more exact expression for $\boldsymbol{w}_{\mathbf{1},\mathbf{1}}$. Also, layer index is required for weights strictly.
- $\boldsymbol{w}_{\mathbf{1},\mathbf{1}}$=$\boldsymbol{w}_{\mathbf{1},\mathbf{1}} + learning\_rate * \frac{\partial f}{\partial \boldsymbol{w}_{\mathbf{1},\mathbf{1}}}$, this is just for a weight. all weights are update like this

- Next, we will learn from code (https://github.com/jhrrlee/mlcode.git)