MATH 4322 - Lecture 21

Neural Networks

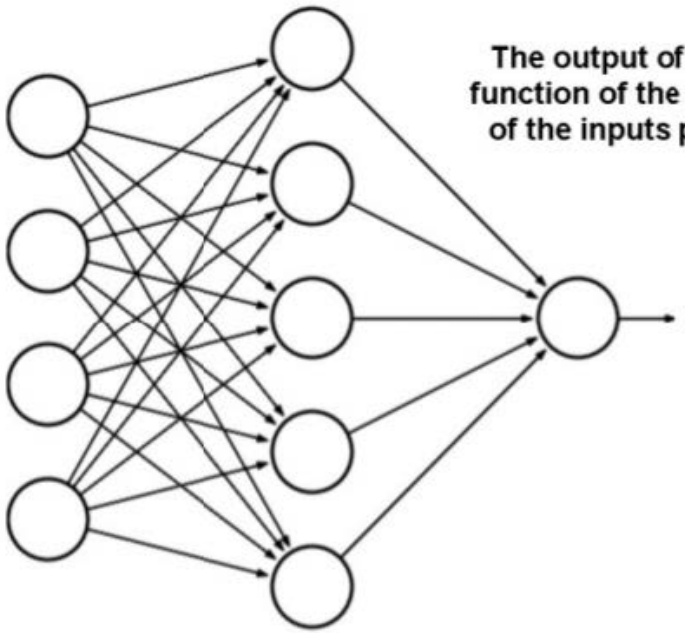
Dr. Cathy Poliak, cpoliak@uh.edu

University of Houston

How do neural netWORKs WORK?

Similar to the biological neuron structure, ANNs define the neuron as a:

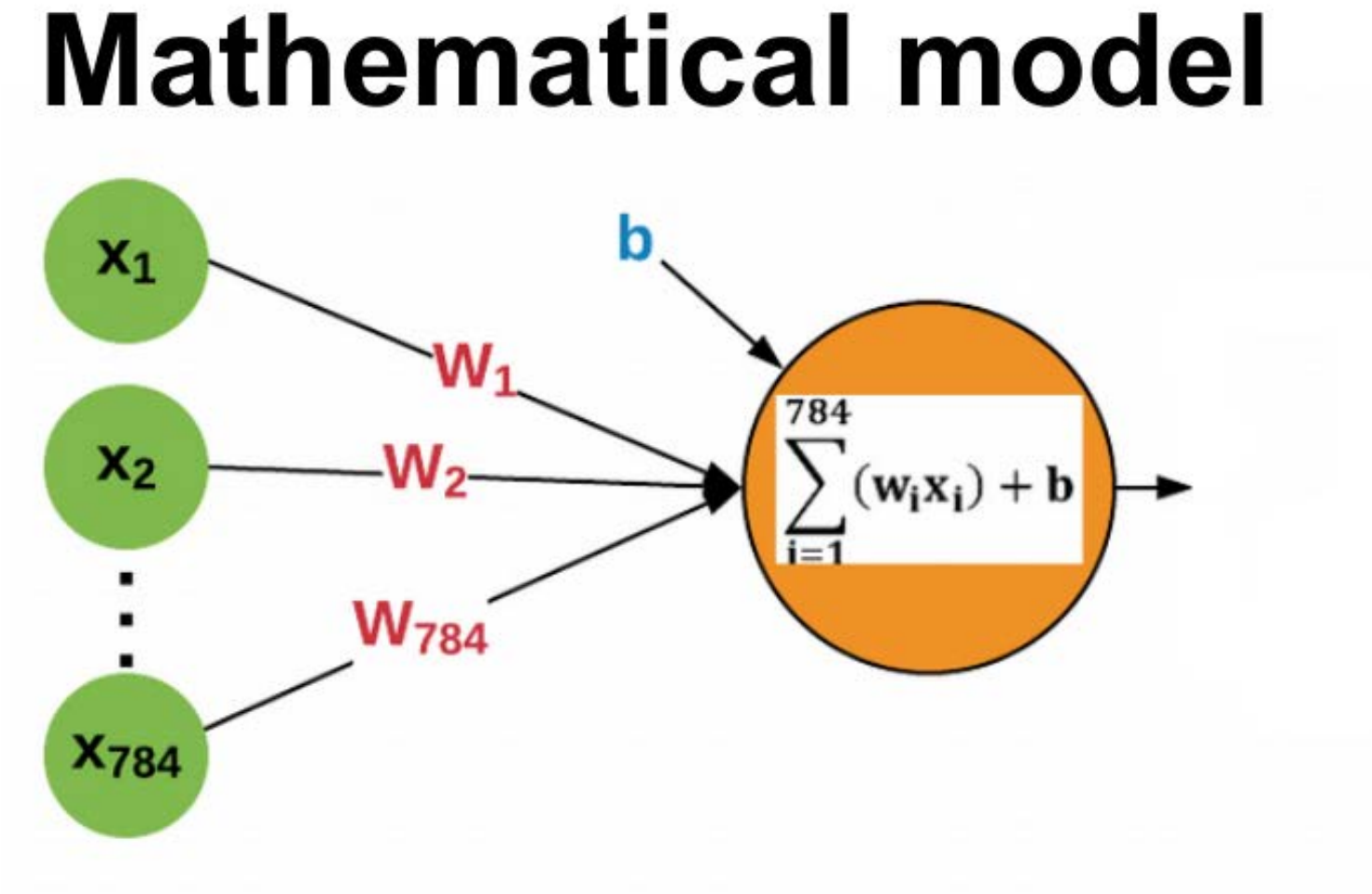
"Central processing unit that performs a mathematical operation to generate output from a set of inputs."



Essentially, ANN is a set of mathematical function approximations.

Mathematical Model of a (Linear) Neuron

For a linear neuron, mathematical model would look like:



Neuron Model: Weights and Bias

Let y - neuron output, then

$$y = b + \sum_{i=1}^{b} w_i x_i$$

Recall multiple linear regression:

$$y = \beta_0 + \beta_1 x_1 + \cdots + \beta_p x_p$$

Only now the parameters we need to learn are called

- **weights** (instead of coefficients), and
- **bias** (instead of intercept)

and denoted as

- w_i (instead of β_i), $i = 1, \ldots, p$,
- b (instead of β_0)

Our task is to estimate weights w_i and bias b, where

- weights determine how strongly one neuron affects the other,
- bias off-sets some of the effects.

Lab: Example in R

- Data collected at a restaurant through customer interviews. The data set is named RestuarntTips
- The customers were asked to give a score to the following aspects: Service, Ambience, and Food.
- Also were asked whether they would leave the tip on the basis of these scores, CustomerWillTip (Tip = 1 and No-tip = 0)

1. What type of problem is this?

   a) Classification problem

   b) Regression problem

R Code

Type and run the following in R

```
library(neuralnet)
library(NeuralNetTools)
library(nnet)
#Import dataset RestaurantTips
attach(RestaurantTips)
RestaurantTips$CustomerWillTip = as.factor(CustomerWillTip)
names(RestaurantTips)
#Train the model based on output from imput
model = nnet(CustomerWillTip ~ Service + Ambience + Food,
                  data = RestaurantTips,
                  size = 5,
                  rang = 0.1,
                  decay = 5e-2,
                  maxit = 5000)
print(model)
```

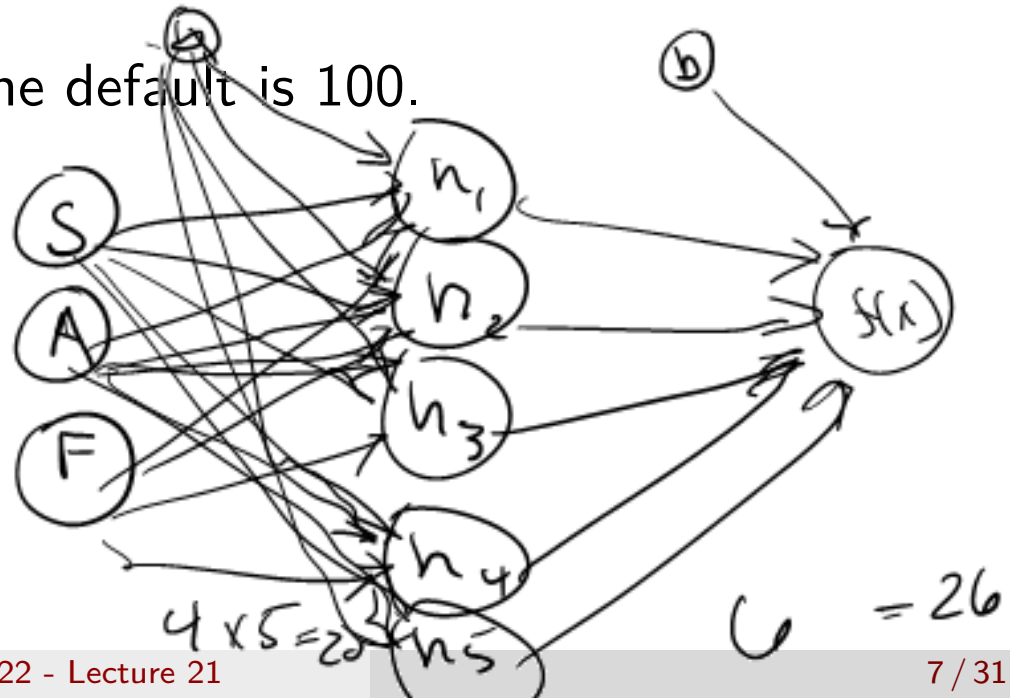
This output processes the forward and backpropagation until convergence.

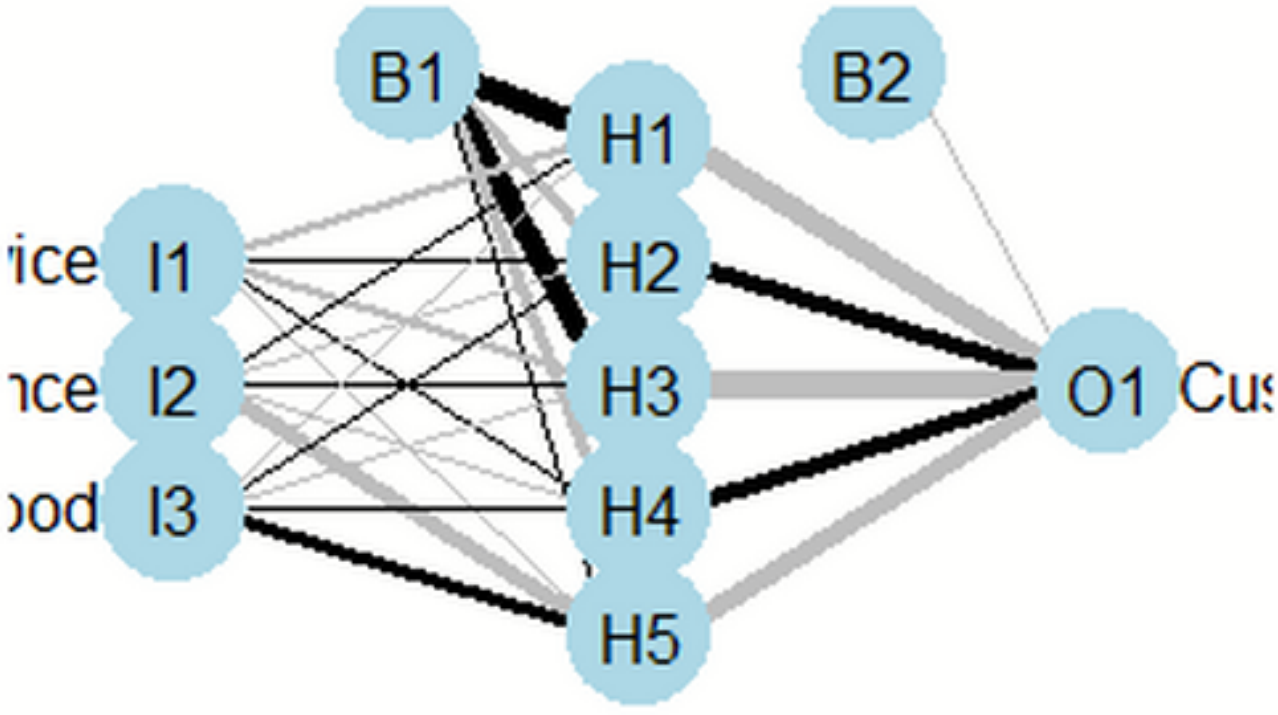
The Parameters in the `nnet()` Function

The parameters used in the `nnet()` function can be tuned to improve performance.

- Size : Number of units in the hidden layer
- Decay : Weight decay specifies regularization in the neural network. As a rule of thumb, the more training examples you have, the weaker this term should be. The more parameters you have the higher this term should be.
- Maxit : Maximum iterations, the default is 100.

```
> print(model)
a 3-5-1 network with 26 weights
inputs: Service Ambience Food
output(s): CustomerWillTip
options were - entropy fitting  decay=0.05
```




B1
B2
H1
ice
I1
H2
nce
I2
H3
O1
Cus
ood
I3
H4
H5

Lab Questions

2. How many weights are there?

a) 5

b) 3

c) 15

d) 26

3. Type and run the following below. What is the training error rate?

a) 16.67%

b) 50%

c) 86.67%

d) 4%

```
pred_nnet<-predict(model,RestaurantTips,type = "class")
(mtab<-table(RestaurantTips$CustomerWillTip,pred_nnet))
```

```
pred_nnet
  0 1
0 12 3
1  2 13
```

5/30

Plots of the Model

4. Type and run the following `plotnet(model)`. How many nodes are in the hidden layer?

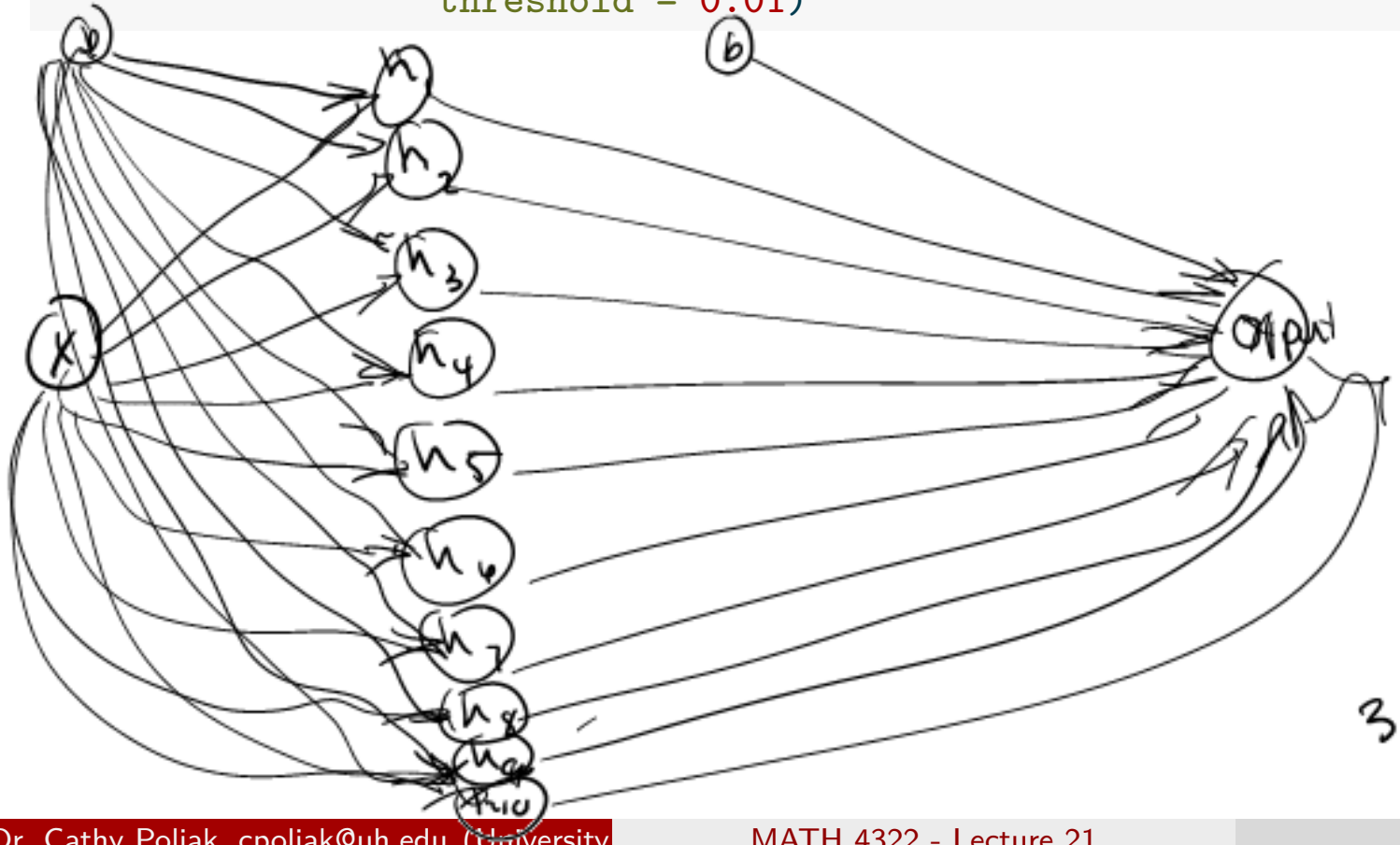
   a) 3
   c) 1
   b) 5
   d) 26

5. Type are run the following `garson(model)`, this is using the `NeuralNetTools`. Which input parameter has the greatest influence to give a tip?

   a) Ambience
   b) Food
   c) Service
   d) All of them are equal.

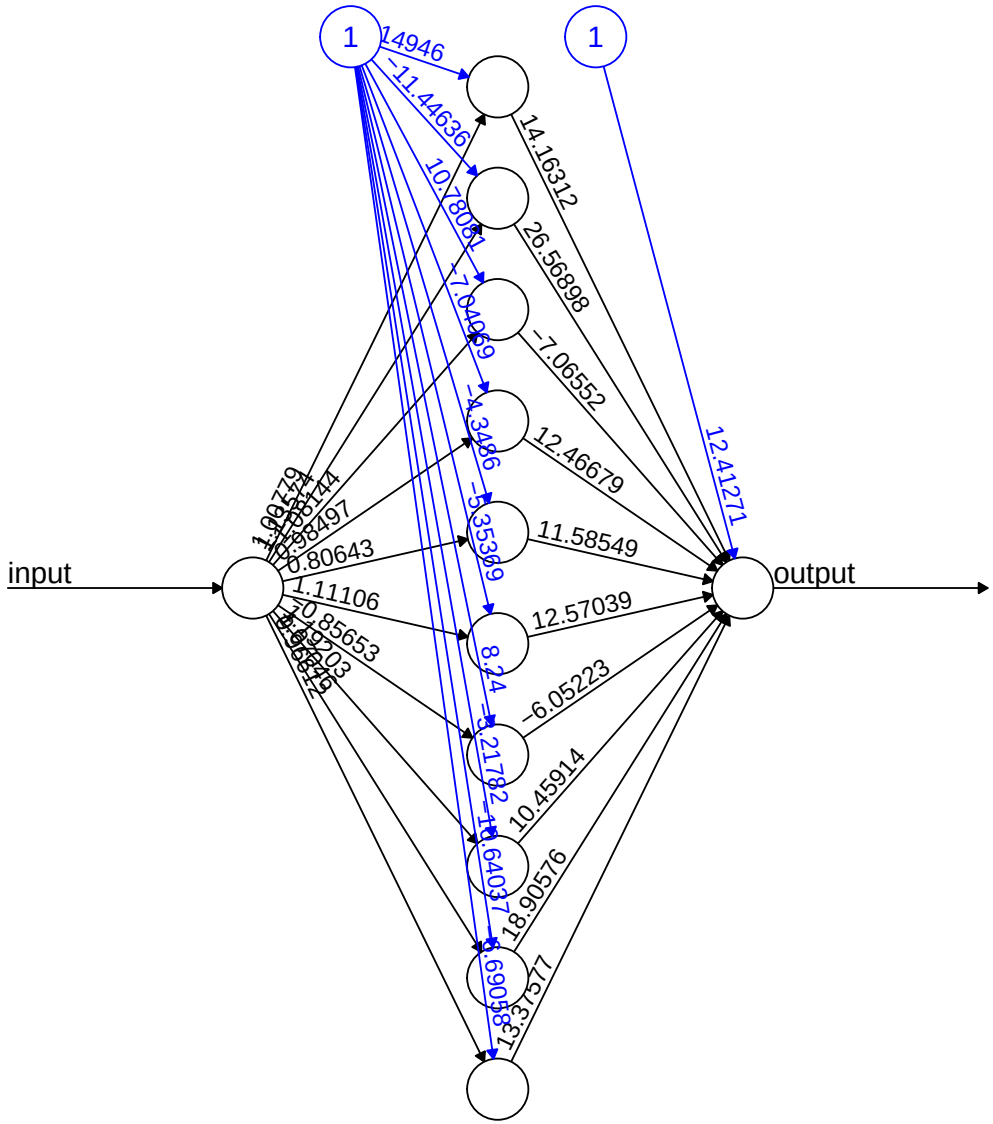
Example 2 Using neuralnet Library

```
#Example 2 Using neuralnet library
input = c(0,1,2,3,4,5,6,7,8,9,10)
output = input^2
mydata = data.frame(cbind(input,output))
names(mydata) = c("input","output")
set.seed(1)
model = neuralnet(output ~ input,  data = mydata, hidden = 10,
                  threshold = 0.01)
```



31 total weights

Plot of the model



Error: 0.010432 Steps: 15595

MSE

```
yhat = as.data.frame(model$net.result)
final_output = cbind(input, output,yhat)
colnames(final_output) = c("Input","Expected output", "Neural Net Output")
sum((output - as.data.frame(model$net.result))^2/2)
```

```
[1] 0.01043176
```

Step-by-step ANN training

Let us analyze in detail, step by step, all the operations to be done for network training:

1. Initialize the weights w^0 and biases b^0 with random values (one time).
2. Repeat the steps 3 to 5 for each training pattern (presented in random order), until the error is minimized or a stopping criteria is reached.
3. Conduct forward propagation for ANN with weights w^0 and biases b^0:

$$\text{input layer} \rightarrow \text{hidden layer(s)} \rightarrow \text{output layer}$$

4. Conduct backpropagation:

$$\text{compute error} \rightarrow \text{take derivative} \rightarrow \text{adjust weights/biases}$$

5. Set w^0 and b^0 equal to adjusted weights/biases, back to step 3.

The complete pass back and forth is called a training cycle or epoch. The updated weights and biases are used in the next cycle. We keep recursively training until the error is very minimal.

Training ANN: Minimizing the Error

We desire to adjust the weight and bias parameters such that to **minimize** the error $\hat{y} - y =$ predicted output $-$ true output.

Example **Single Neuron Model:**

- p nodes $x_1, \ldots, x_p$ in the input layer,
- one output node y, calculated as $\hat{y} = b + \sum_{j=1}^{p} w_j x_j$

Presume we are supplied with:

- n data samples $\mathbf{x}_i = (x_{1,i}, \ldots, x_{p,i}), i = 1, \ldots, n$
- n corresponding true responses (or labels) $y_i, i = 1, \ldots, n$

Training ANN: Minimizing the Error

Feed the whole data $(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)$ to our ANN at once. Then the total error formula is

$$\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 = \sum_{i=1}^{n}[y_i - (b + \sum_{j=1}^{p} w_j x_{j,i})]^2 \equiv f_{err}(b, w_1, \ldots, w_p)$$

and we need to minimize it with respect to $b, w_1, \ldots, w_p$

$$\min_{b,w_1,\ldots,w_p} \sum_{i=1}^{n}[y_i - (b + \sum_{j=1}^{p} w_j x_{j,i})]^2 = \min_{b,w_1,\ldots,w_p} f_{err}(b, w_1, \ldots, w_p) \quad (1)$$

Similar to the **least squares regression**:

- $\hat{y}_i = \beta_0 + \sum_{j=1}^{p} \beta_j x_{j,i}$
- $\min_{\beta_0,\ldots,\beta_p} \sum_{i=1}^{n}[y_i - (\beta_0 + \sum_{j=1}^{p} \beta_j x_{j,i})]^2 = \min_{\beta_0,\ldots,\beta_p} f_{err}(\beta_0, \beta_1, \ldots, \beta_p)$

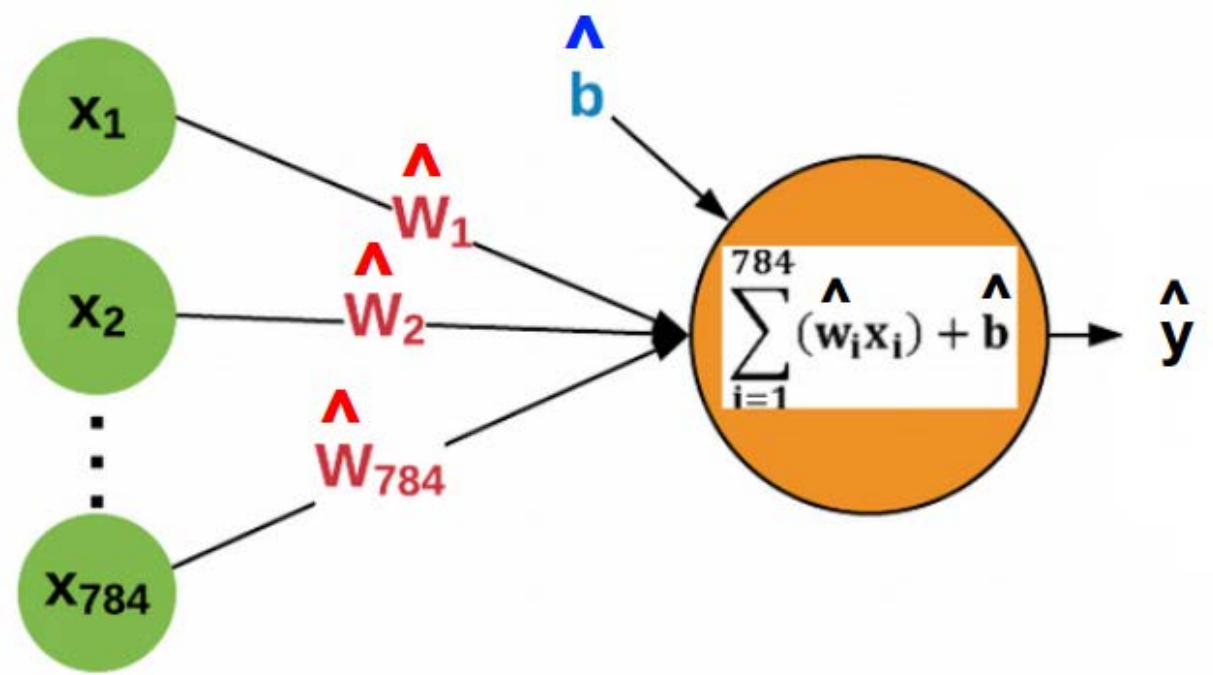
Which we could solve **analytically** via $\Delta_{\boldsymbol{\beta}=(\beta_0,\ldots,\beta_p)} f(\boldsymbol{\beta}) \equiv 0$.

Training ANN: Minimizing the Error

Likewise for the Linear Neuron Model, in order to obtain optimal values $b, w_1, \ldots, w_p$ that minimize (1), we **could** analytically solve:

$$\Delta_{\boldsymbol{w}=(b,w_1,\ldots,w_p)} f(\boldsymbol{w}) \equiv 0$$

$$\implies \hat{\boldsymbol{w}} = (\hat{b}, \hat{w}_1, \ldots, \hat{w}_p) = \underset{b,w_1,\ldots,w_p}{\mathit{argmin}} \sum_{i=1}^{n} [y_i - (b + \sum_{j=1}^{p} w_j x_{j,i})]^2$$



Training ANN: Gradient Descent

While such simple case as Linear Neuron Model could be solved analytically (read "one could calculate an explicit formula for weight and bias estimates"), there are multiple aspects to note:

1. We'd like to avoid solving large systems of equations analytically.
2. We want a method that real neurons are likely using. Hint: they're probably not analytically solving any equations.
3. We want a method that can be generalized to **multi**-layer, **non**-linear neural networks, for most of which the closed-form analytical solutions (read "explicit formulas") simply won't be available.

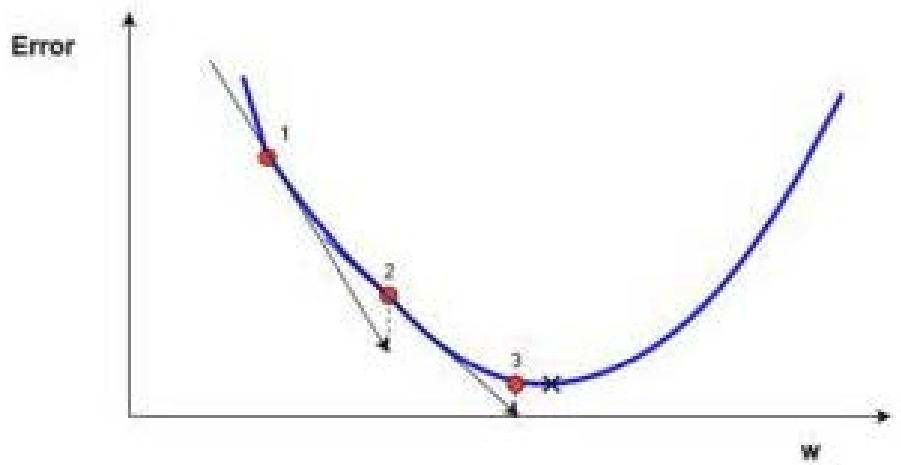
That's where such numerical optimization technique as **gradient descent** comes in extremely handy.

Gradient Descent

Gradient descent is an optimization approach, which entails using:

- Partial derivatives (gradients) of a function, and
- a step size parameter,

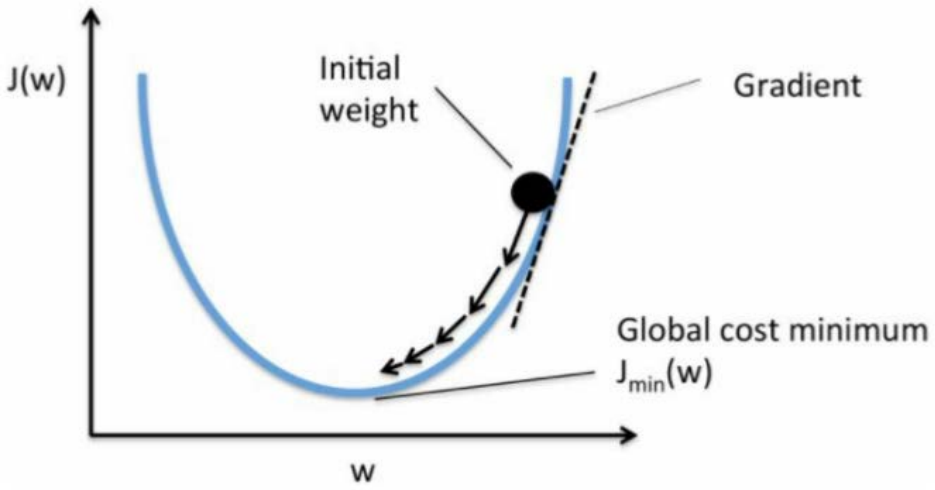
in order to calculate the minimum (or maximum) of **that** function.



We move by the direction of steepest descent.

Gradient Descent for ANN

For neural networks, the gradient descent approach is used when iterating the **updates** of weights and biases.



Global cost function in our case is the squared prediction error,

$$\frac{1}{2}(\hat{y}-y)^2, \quad \text{or} \quad \frac{1}{2}\sum_{i=1}^{n}(\hat{y}_i-y_i)^2$$

which we are trying to minimize.

Gradient Descent: online- and batch-learning

Questions about weight updates:

- Which function to optimize, $\frac{1}{2}(\hat{y} - y)^2$? Or is it $\frac{1}{2}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2$?
- How often do we need to update the parameter estimates?

Gradient Descent: online- and batch-learning

Questions about weight updates:

- Which function to optimize, $\frac{1}{2}(\hat{y} - y)^2$? Or is it $\frac{1}{2}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2$?
- How often do we need to update the parameter estimates?

Given data $(\mathbf{x}^1, y^1), \ldots, (\mathbf{x}^n, y^n)$, where $\mathbf{x}^i = (x_1^i, \ldots, x_p^i)$, gradient descent can be performed for

- **online**-learning - ANN is fed one training sample $(\mathbf{x}^i, y^i)$ at a time) $\Longrightarrow$ weights are updated each time by minimizing $\frac{1}{2}(\hat{y}^i - y^i)^2$.
- **full-batch** - ANN is fed full training set all at once $\Longrightarrow$ weights are updated once all $\hat{y}^i$ got calculated, minimize $\frac{1}{2}\sum_{i=1}^{n}(\hat{y}^i - y^i)^2$.
- **mini-batches** - ANN is iteratively fed subsets of training data $\Longrightarrow$ weights are updated once all $\hat{y}^i \in$ subset got calculated, minimize $\frac{1}{2}\sum_{i \in subset}(\hat{y}^i - y^i)^2$

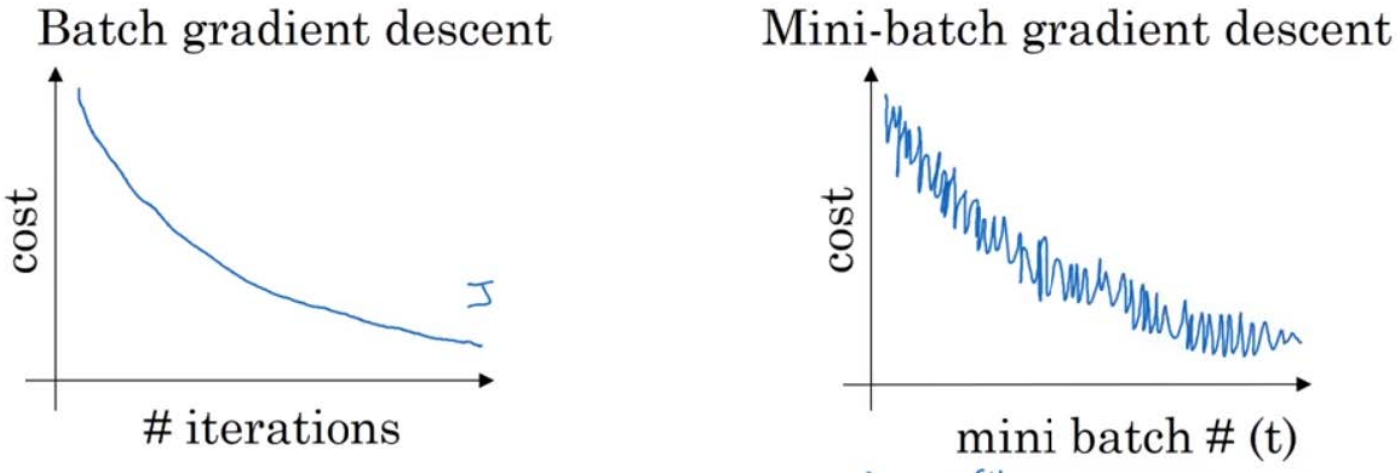
Gradient Descent: online- and batch-learning

For either of three approaches (online, full- or mini-batch),

1. The weights & biases are randomly initialized at first.
2. They get updated each time after:
   a) Data is fed to ANN (be it single sample, full- or mini-batch),
   b) Forward propagation is performed to get $\hat{y}_i$'s,
   c) Error function is calculated (be it single $\frac{1}{2}(\hat{y} - y)$ or $\frac{1}{2}\sum_i(\hat{y}_i - y_i)^2$)
   d) Weights are updated via gradient descent.
3. The step 2 is iterated until all training observations were used, with updated weights being used for next iteration.
4. Steps 2 & 3 formulate an **epoch** - a single pass through the whole training set. ANNs are trained over many epochs, with each epoch potentially containing multiple iterations (bar the full-batch approach).

Gradient Descent: full- vs and mini-batch example

Example. Below is an exemplary progression for a full-batch gradient descent (left) as opposed to mini-batch (right). Cost here is the error function, $\frac{1}{2}\sum_i(\hat{y}_i - y_i)$.



Changes of error function are

- fewer and smoother for full-batch,
- more frequent and bumpier for mini-batch.

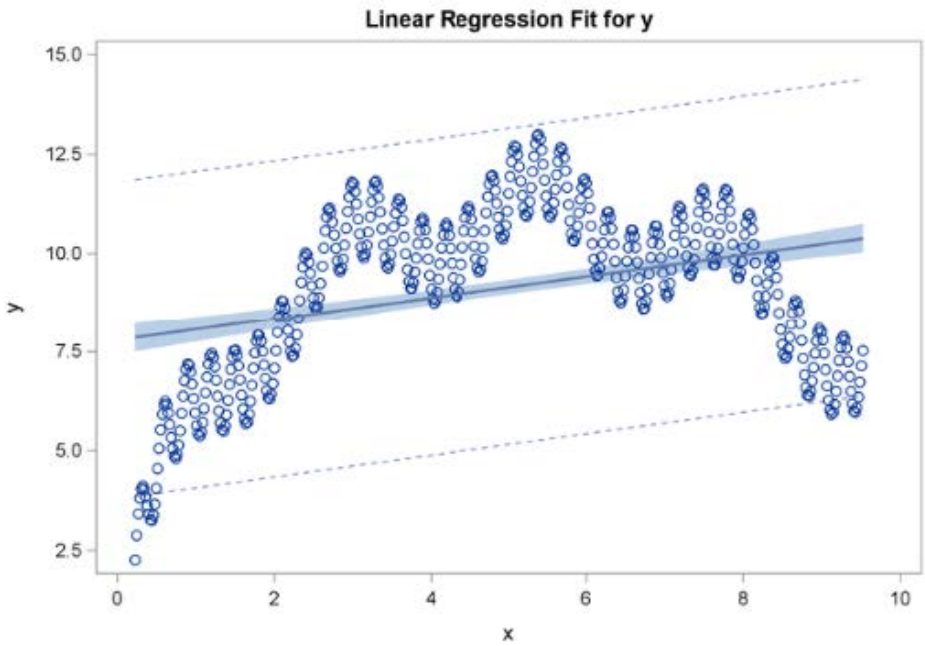
Linear Neuron Model: Limitations

Question. If we stick with a Linear Neuron Model, where each neuron simply outputs a weighted linear combination of its outputs, what type of function of original inputs $x_1, \ldots, x_n$ would we inevitably get?

Linear Neuron Model: Limitations

Question. If we stick with a Linear Neuron Model, where each neuron simply outputs a weighted linear combination of its outputs, what type of function of original inputs $x_1, \ldots, x_n$ would we inevitably get? **Answer.**

Linear function.

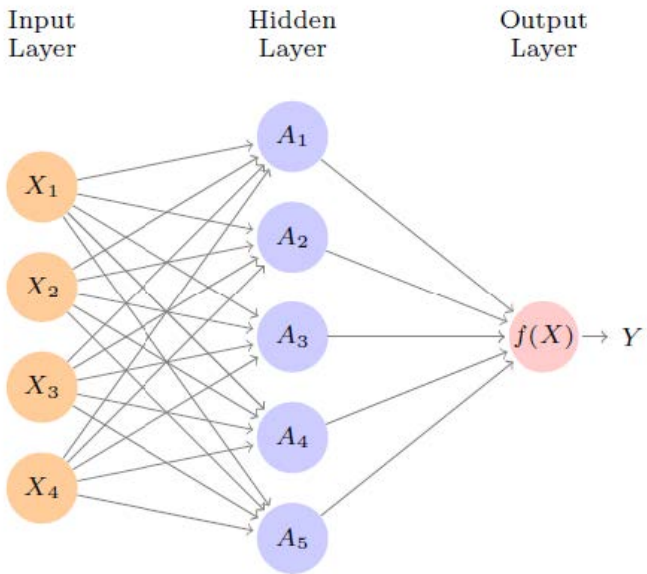


Linear functions can only do that much when dealing with non-linearity.

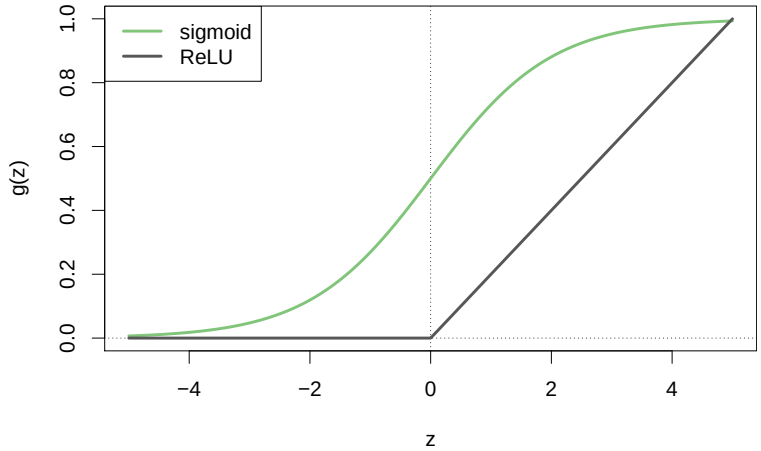
Recall Single Layer Neural Network

$$f(X) = \beta_0 + \sum_{k=1}^{K} \beta_k h_k(X)$$

$$= \beta_0 + \sum_{k=1}^{K} \beta_k \left(w_{k0} + \sum_{j=1}^{p} w_{kj} X_j \right)$$



Activation Functions



- $A_k = h_k(X) = g\left(W_{k0} + \sum_{j=1}^{p} w_{kj} X_j\right)$ are called the **activations** in the hidden layer.
- $g(Z)$ is called the **activation function**.
- Activation functions in hidden layers are typically nonliner.
- The model from the previous slide derives five new factures by computing five different linear combinations of X, and then puts each through an activation function $g(.)$ to transform it.

$$f(x) = \beta_0 + \sum \beta_i \, g\left(w_{k0} + \sum w_{ij} x_j\right)$$

Sigmoid Activation Function

$$g(z) = \frac{e^z}{1+e^z} = \frac{1}{1+e^{-z}}$$

- Same function used in logistic regression.
- Converts a linear function into probabilities between zero and one.

RELU Activation Function

$$g(z) = (z)_+ = \begin{cases} 0 & \text{if } z < 0 \\ z & \text{otherwise} \end{cases}$$

- REctified Linear Unit activation function.
- Preferred choice in modern neural networks.
- This is known as the "dying ReLU" problem, and it can be solved in a number of ways, such as using variations on this formula, including the leaky ReLU, exponential ReLU or parametric ReLU function.

Sofmax Activation function

$$f_m(X) = Pr(Y = m|X) = \frac{e^{Z_m}}{\sum_{i=1}^{M} e^{Z_i}}$$

- Used for multi-class classification problem.
- This estimates a probability for each of the classes, then assigns the observation to the class with the higest probability.

Example of Activation Function

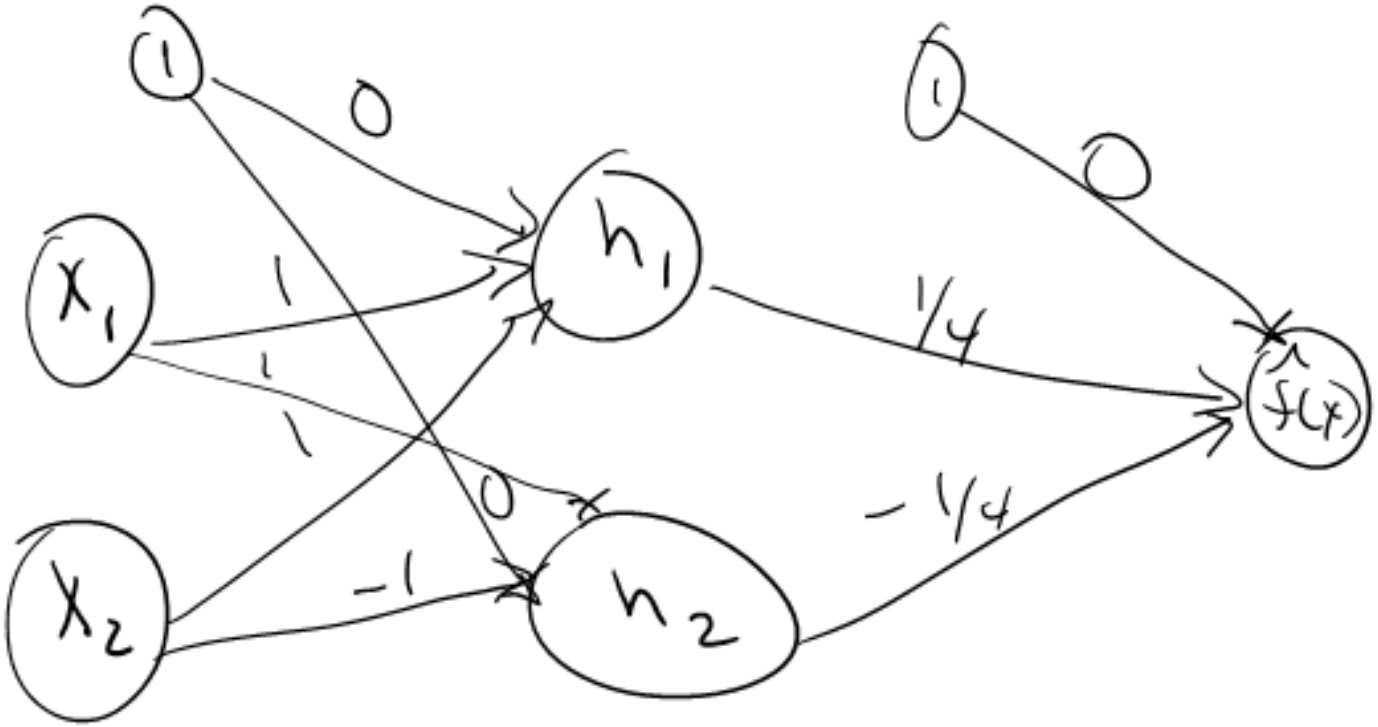
Let $p = 2$ input variables, and $K = 2$ hidden units $h_1(X)$ and $h_2(X)$, with $g(z) = z^2$. We specify the other parameters as:

$$\beta_0 = 0, \beta_1 = \frac{1}{4}, \beta_2 = -\frac{1}{4}$$
$$w_{10} = 0, w_{11} = 1, w_{12} = 1$$
$$w_{20} = 0, w_{21} = 1, w_{22} = -1$$

Determine $f(X) = \beta_0 + \beta_1 g(w_{10} + w_{11}X_1 + w_{12}X_2) + \beta_2 g(w_{20} + w_{21}X_1 + w_{22}X_2)$

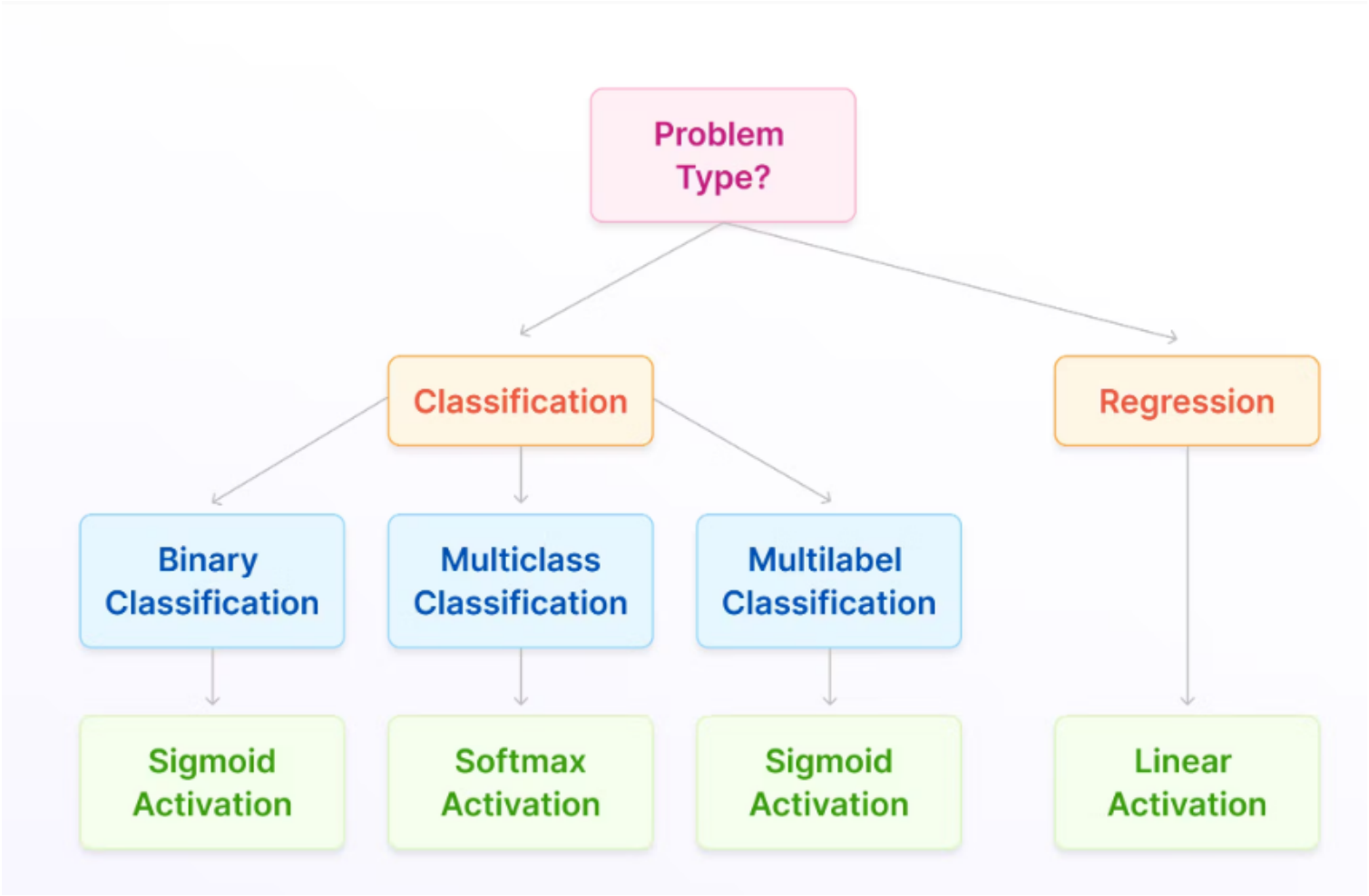
$$= 0 + \frac{1}{4}[0 + 1X_1 + 1X_2]^2 - \frac{1}{4}[0 + 1X_1 - 1X_2]^2$$
$$= \frac{1}{4}[X_1^2 + 2X_1X_2 + X_2^2] - \frac{1}{4}[X_1^2 - 2X_1X_2 + X_2^2] = \frac{1}{2}X_1X_2 + \frac{1}{2}X_1X_2$$
$$= X_1X_2$$

Plot of the Neural Network



$$g(z) = z^2$$

Which Activation Function to Use?



https://encord.com/blog/activation-functions-neural-networks/#h3