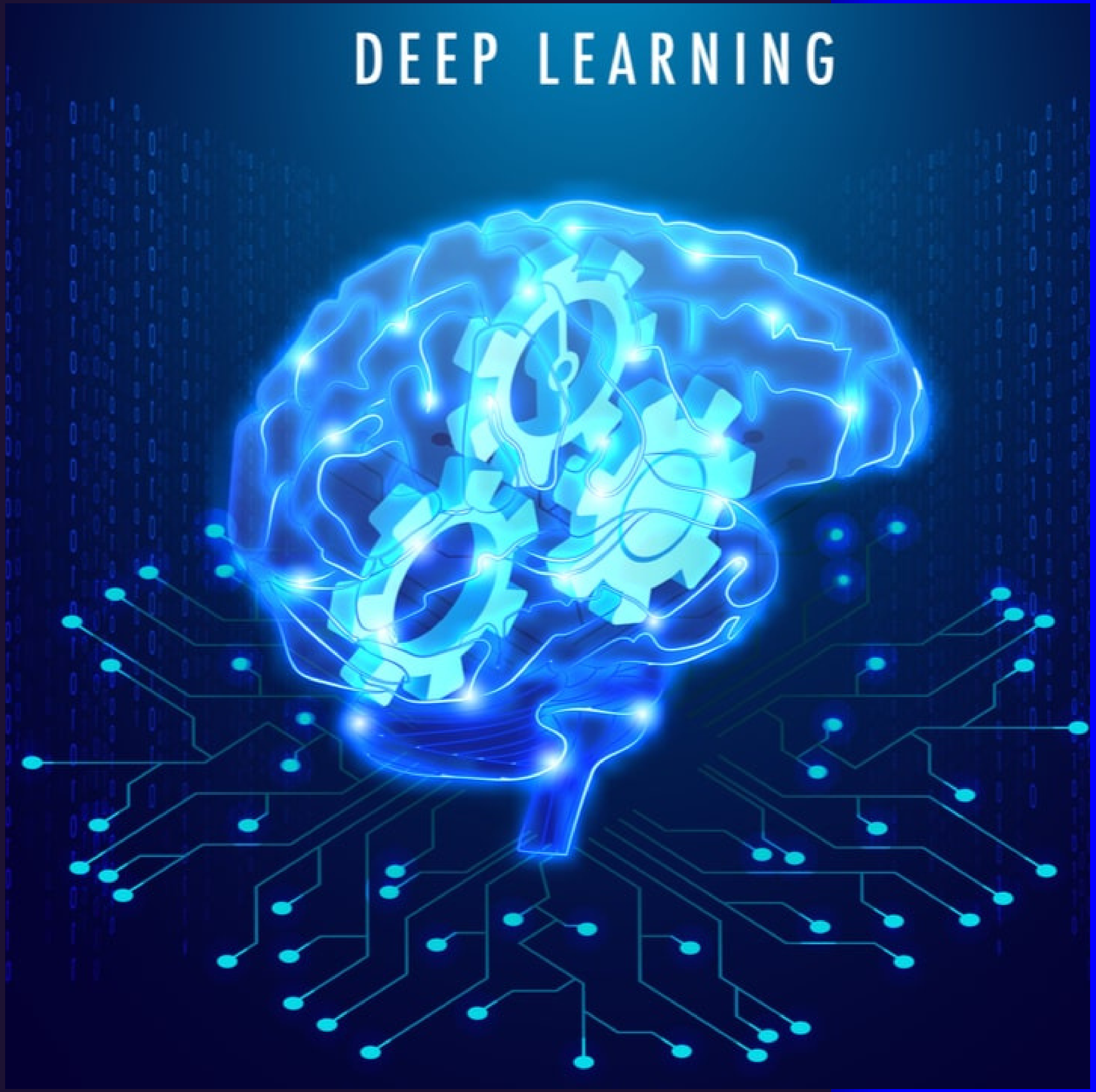
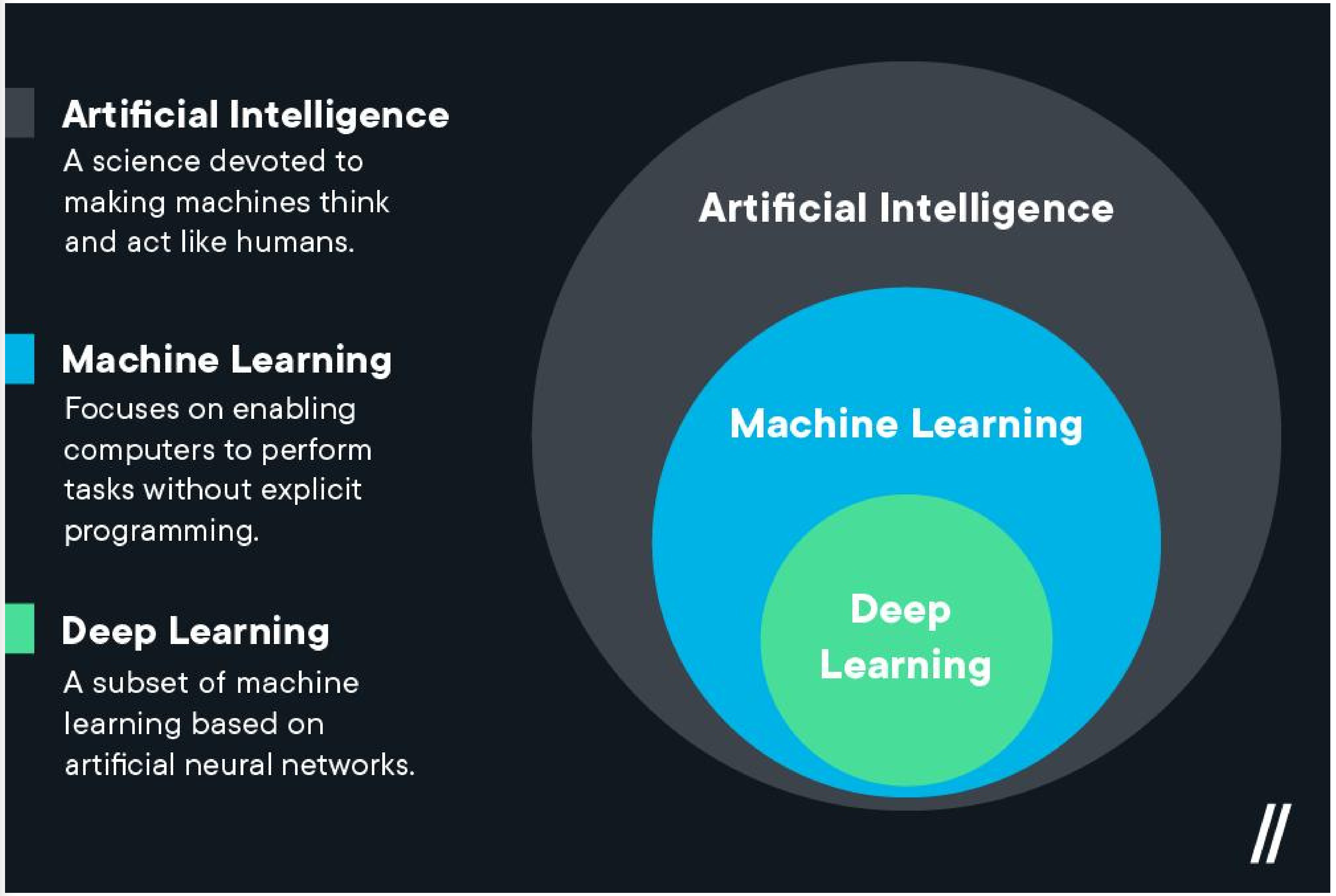
# DEEP LEARNING

NAME: APOORVE SHUKLA
ROLL-NO: 20075014
BRANCH: CSE 3RD



## DEEP LEARNING

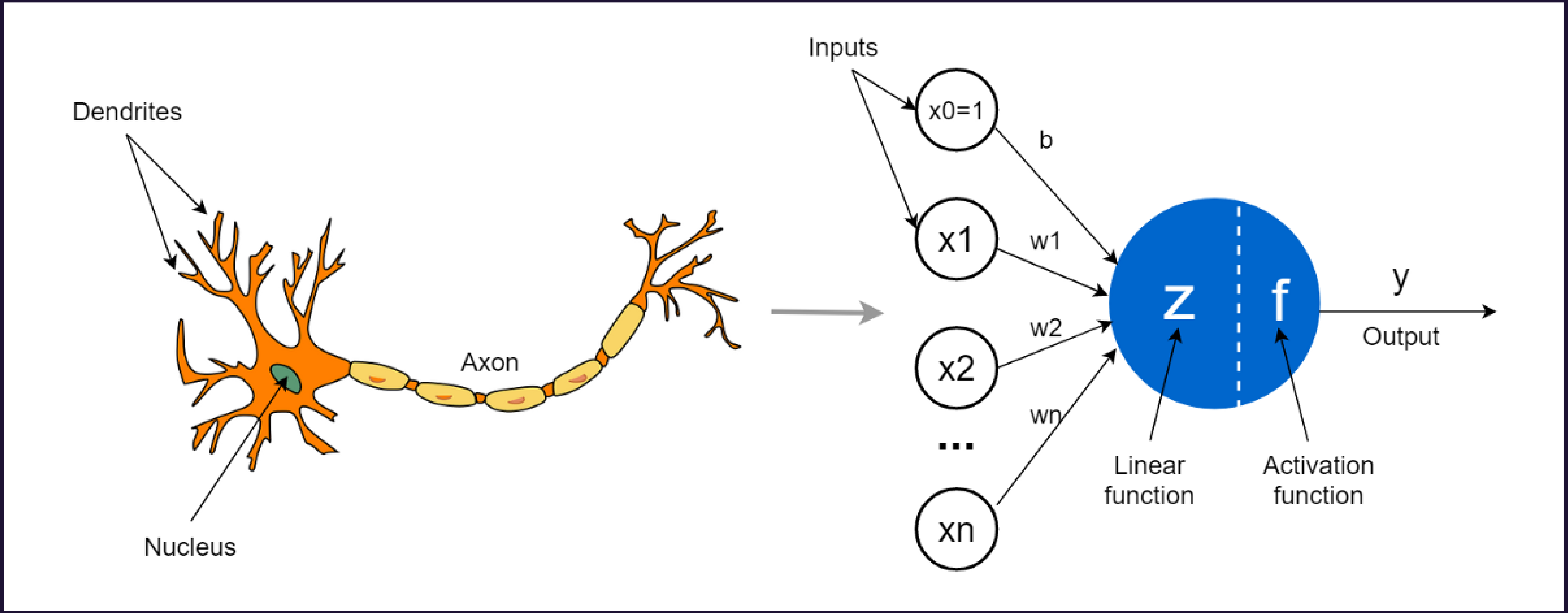
Deep learning networks learn by discovering intricate structures in the data they experience. By building computational models that are composed of multiple processing layers, the networks can create multiple levels of abstraction to represent the data.



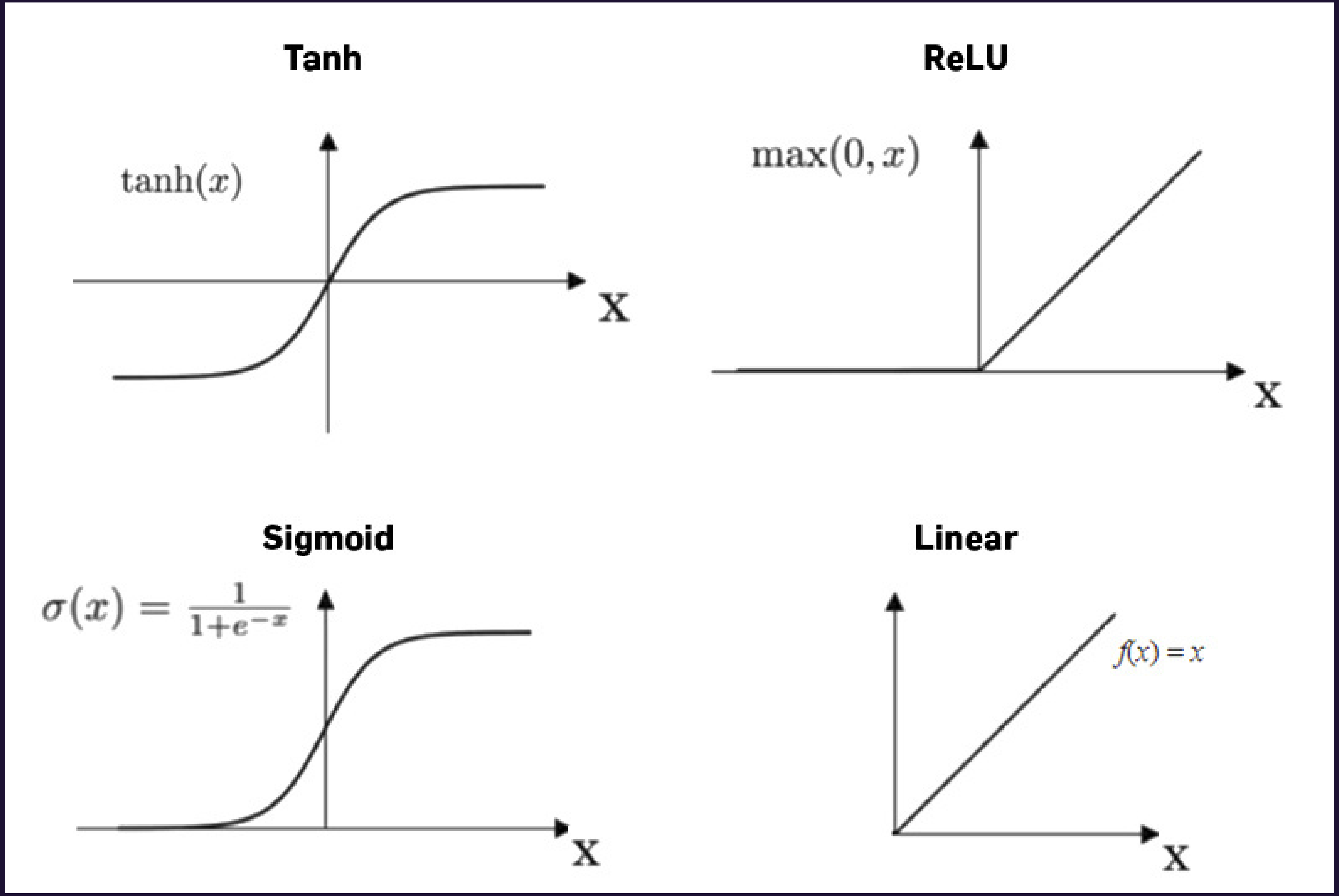
**Artificial Intelligence**
A science devoted to making machines think and act like humans.

**Machine Learning**
Focuses on enabling computers to perform tasks without explicit programming.

**Deep Learning**
A subset of machine learning based on artificial neural networks.



- M is the number of training vectors
- Nx is the size of the input vector
- Ny is the size of the output vector
- X(1) is the first input vector
- Y(1) is the first output vector
- X = [x(1) x(2).. x(M)]
- Y = (y(1) y(2).. y(M))
- xi is the ith input feature
- f is the activation function
- wi is the weight assigned to ith feature
- b is the bias
- Z=W(TRANSPOSE).X+B
- W=[W(1) W(2) W(3)..W(M)]



# Activation functions

The activation function is used to convert the input signal on the node of ANN to an output signal.

Y = SIGMOID(W(TRANSPOSE)X + B)

## Loss function:

Loss function is a method of evaluating “how well your algorithm models your dataset”. If your predictions are totally off, our loss function will output a higher number.

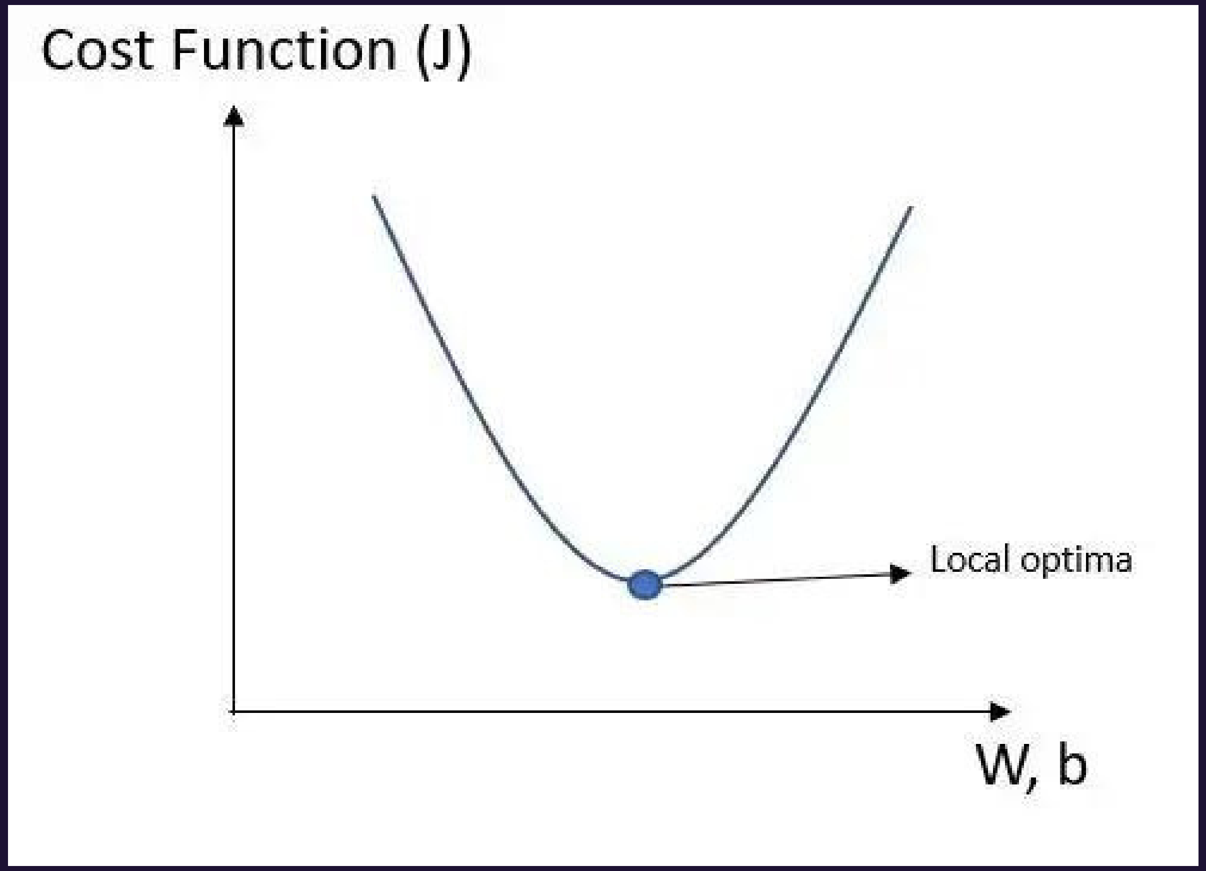
Types of Loss function:

- Regression Loss Function
- Binary Classification Loss Functions
- Multi-class Classification Loss Functions

One important Loss function is: Categorical cross-entropy

$$L(y,\hat{y}) = -\sum_{j=0}^{M}\sum_{i=0}^{N}(y_{ij} * log(\hat{y}_{ij}))$$

## Cost function:



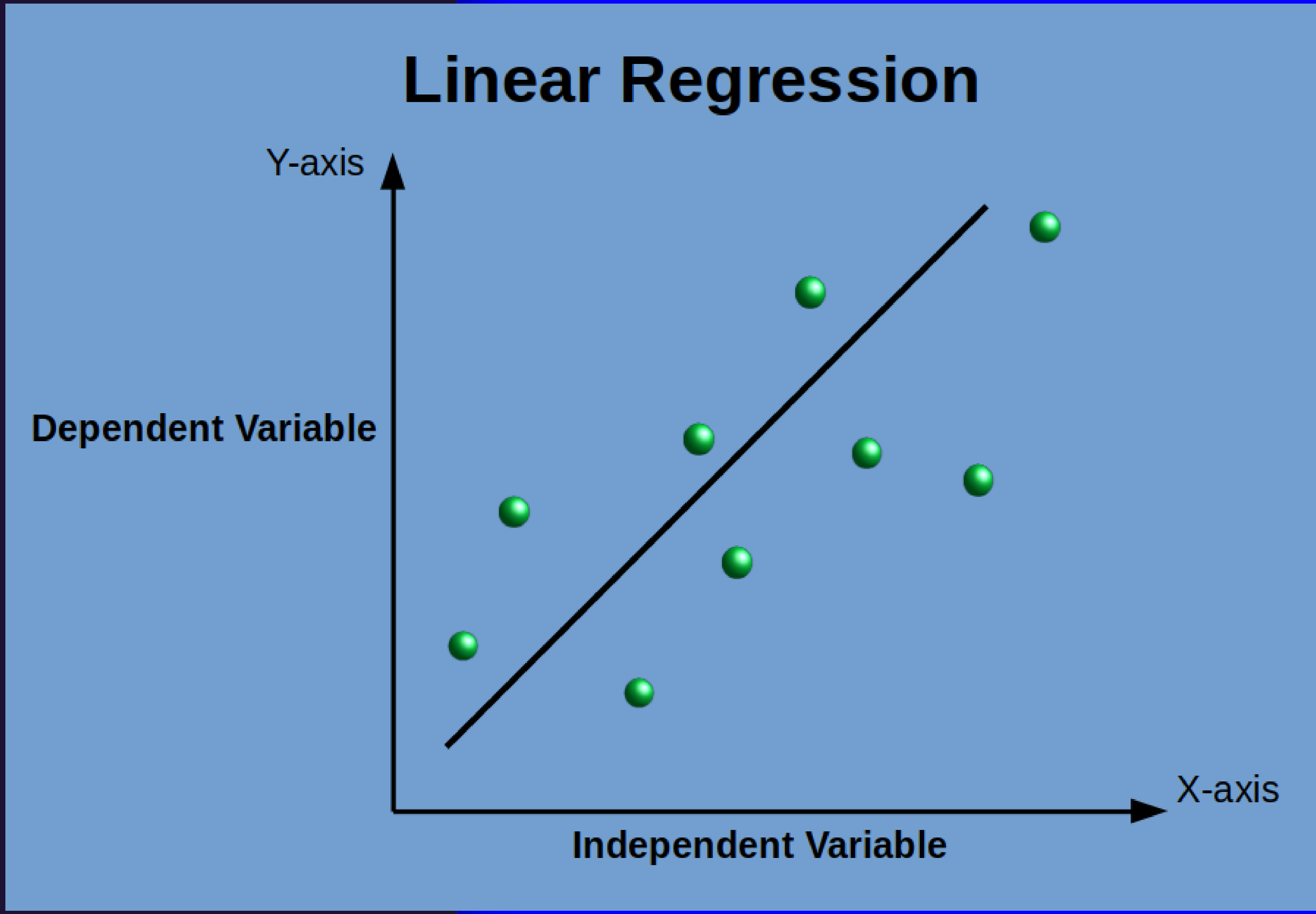
The loss error is computed for a single training example. If we have 'm' number of examples then the average of the loss function of the entire training set is called 'Cost function'.

if is our loss function, then we calculate the cost function by aggregating the loss over the training.

$$Cost(f, \mathcal{D}) = \frac{1}{n}\sum_{i=1}^{n} L\left(\hat{y}_i, y_i\right) \qquad (\hat{y}_i = f(\mathbf{x}_i))$$ (the mean loss cost):

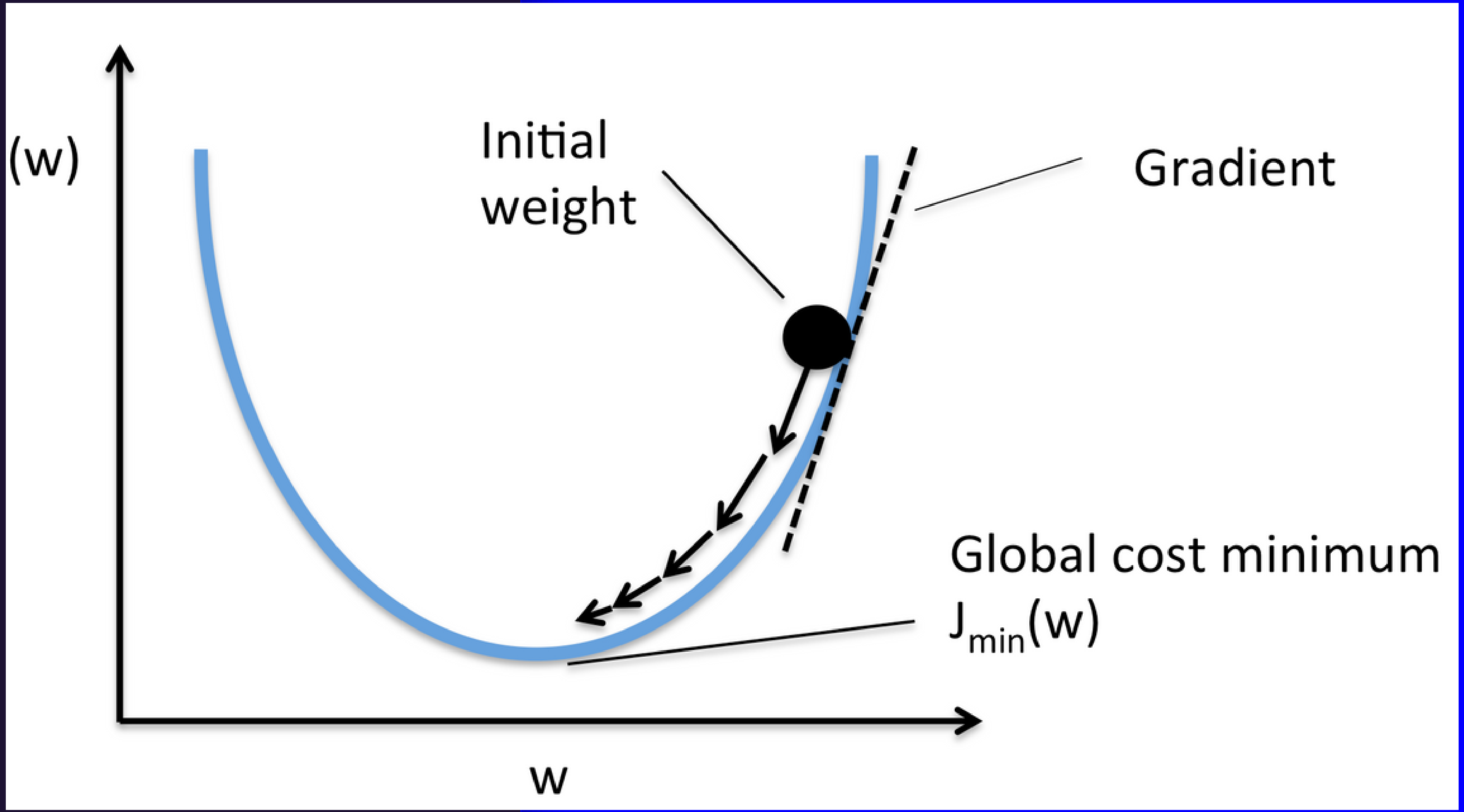
The cost function will be used to update the parameters of the model using some algorithms like Gradient Descent

- The algorithm is used for the classification algorithm of 2 classes.
- y = w(transpose)x + b
- If we need y to be in between 0 and 1 (probability): y = sigmoid(w(transpose)x + b)
- The loss function that used: L(y',y) = -(y*log(y') + (1-y)*log(1-y'))
- Then the Cost function is: J(w,b) = (1/m) * Sum(L(y'[i],y[i]))
- The algorithm used for parameters update is called **Gradient Descent**.



## Gradient Descent

- First, initialize w and b to 0,0 or initialize them to a random value in the convex function and then try to improve the values the reach minimum value.
- The gradient descent algorithm repeats: w = w - alpha * dw where alpha is the learning rate and dw is the derivative of w (Change to w) The derivative is also the slope of w
- The final equations we will implement:
  - w = w - alpha * d(J(w,b) / dw) (how much the function slopes in the w direction)
  - b = b - alpha * d(J(w,b) / db) (how much the function slopes in the d direction)



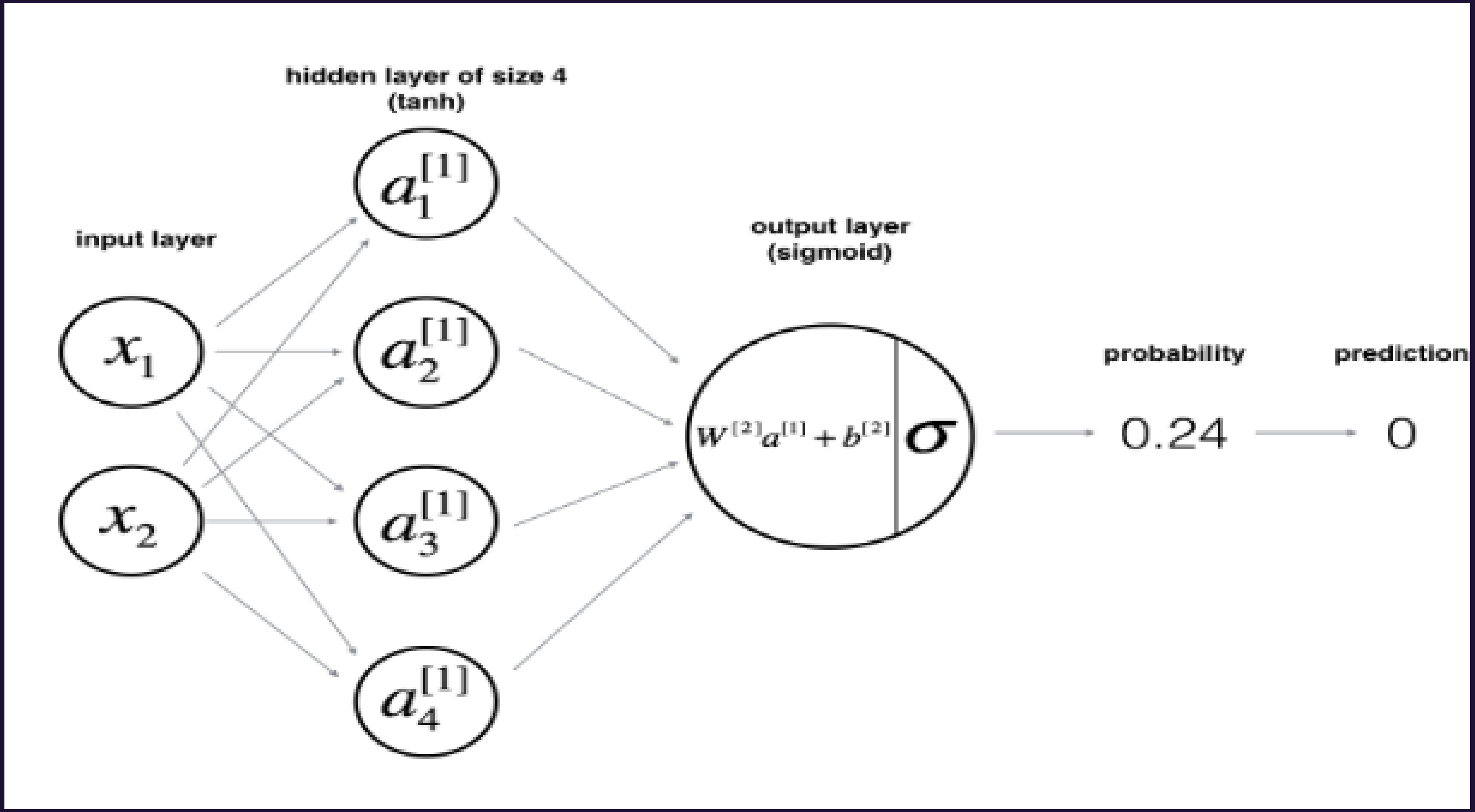
## Apply gradient descent in Linear regression

- This step is also known as backpropagation for a NN.
- As follows:
    - $$\frac{\partial J}{\partial w} = \frac{1}{m} X(A-Y)^T$$
      $$\frac{\partial J}{\partial b} = \frac{1}{m} \sum_{i=1}^{m} (a^{(i)} - y^{(i)})$$

w = w - alpha * dw

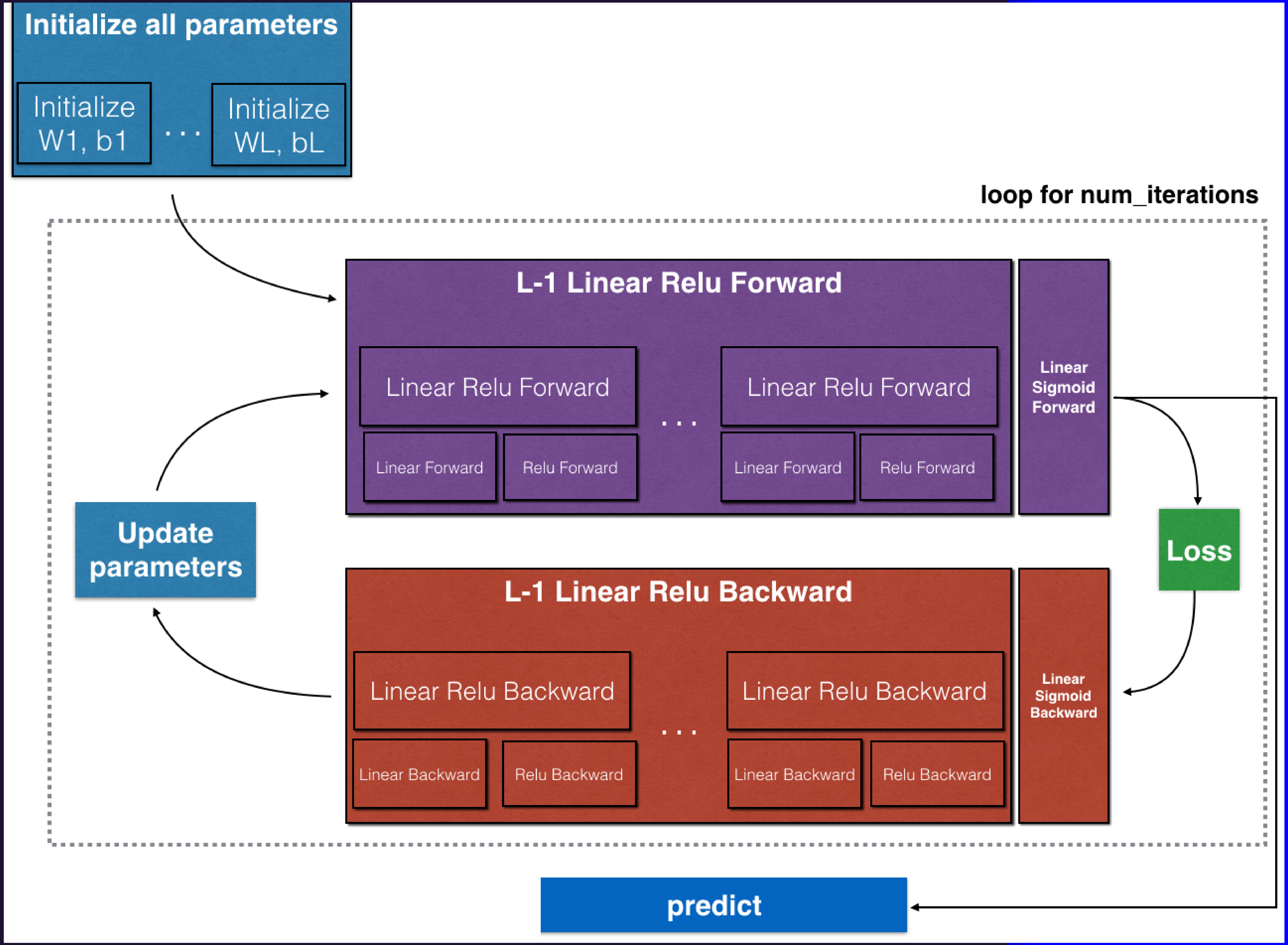
b = b - alpha * db

- Perform the parameter updation on every iteration (epochs) as above.



# Neural network model

- Interconnected nodes or neurons in a layered structure that resembles the human brain
- Simple neural network architecture
  - Input Layer
  - Hidden Layer
  - Output Layer
- Each neuron of every layer is connected to every neuron of the next layer.
- Weights and biases are associated with each neuron in every layer except the input layer.


Initialize all parameters
Initialize W1, b1
. . .
Initialize WL, bL
loop for num_iterations
L-1 Linear Relu Forward
Linear Relu Forward
Linear Forward
Relu Forward
. . .
Linear Relu Forward
Linear Forward
Relu Forward
Linear Sigmoid Forward
Loss
Update parameters
L-1 Linear Relu Backward
Linear Relu Backward
Linear Backward
Relu Backward
. . .
Linear Relu Backward
Linear Backward
Relu Backward
Linear Sigmoid Backward
predict

Some notations for maths involved in the propagations of neural networks-

- Superscript [$l$] denotes a quantity associated with the $lth$lth layer
- Superscript ($i$) denotes a quantity associated with the $ith$
- Lowerscript $i$ denotes the $ith$ith entry of a vector.

Dimensions of different vectorized parameters:

- Wl : weight matrix of shape (layer_dims[l], layer_dims[l-1])
- bl: bias vector of shape (layer_dims[l], 1)
- Al: Activation of lth layer of dimension(layer_dimension[l],m)
- m: Number of training examples

## Forward and backward propagation

$$Z^{[1]} = W^{[1]}X + b^{[1]}$$
$$A^{[1]} = g^{[1]}(Z^{[1]})$$
$$Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]}$$
$$A^{[2]} = g^{[2]}(Z^{[2]})$$
$$\vdots$$
$$A^{[L]} = g^{[L]}(Z^{[L]}) = \hat{Y}$$

## Forward and backward propagation

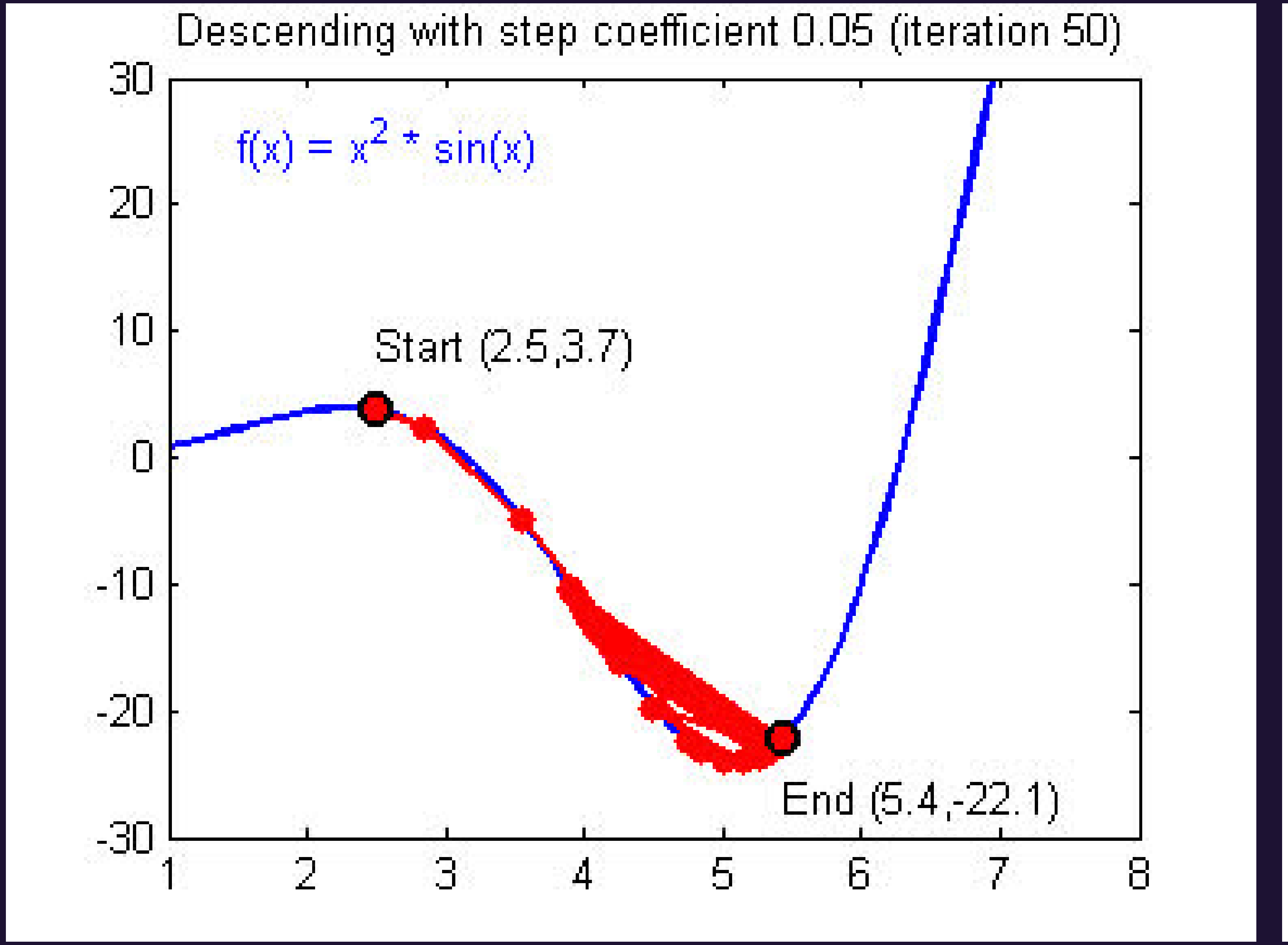
$$
\begin{aligned}
dZ^{[L]} &= A^{[L]} - Y \\
dW^{[L]} &= \frac{1}{m} dZ^{[L]} A^{[L]^T} \\
db^{[L]} &= \frac{1}{m} np.\mathrm{sum}(\mathrm{dZ}^{[L]}, axis = 1, keepdims = True) \\
dZ^{[L-1]} &= dW^{[L]^T} dZ^{[L]} g'^{[L]}(Z^{[L-1]}) \\
&\vdots \\
dZ^{[1]} &= dW^{[L]^T} dZ^{[2]} g'^{[1]}(Z^{[1]}) \\
dW^{[1]} &= \frac{1}{m} dZ^{[1]} A^{[1]^T} \\
db^{[1]} &= \frac{1}{m} np.\mathrm{sum}(\mathrm{dZ}^{[1]}, axis = 1, keepdims = True)
\end{aligned}
$$

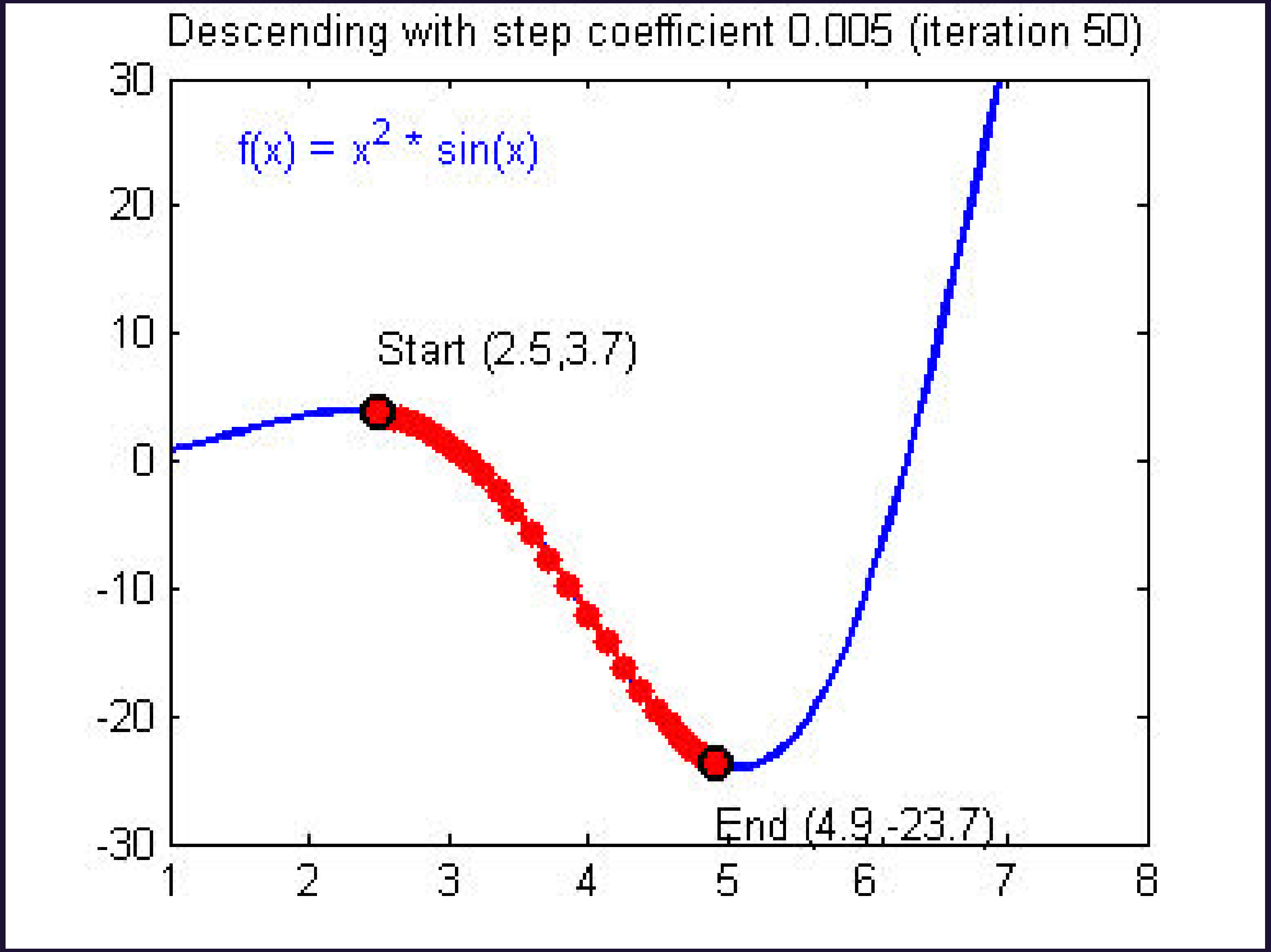
## Update parameters in each iterations:

If n is the learning rate or update hyperparameter then the update equation for parameter θ will be:

- θ(new)=θ(old)-n*dθ, where dθ =dJ/dθ

The learning rate is a hyper-parameter used to govern the pace at which an algorithm updates or learns the values of a parameter estimate.In other words, the learning rate regulates the weights of our neural network concerning the loss gradient





## Choice of learning rate

## Splitting dataset for Train/Dev/Tests sets

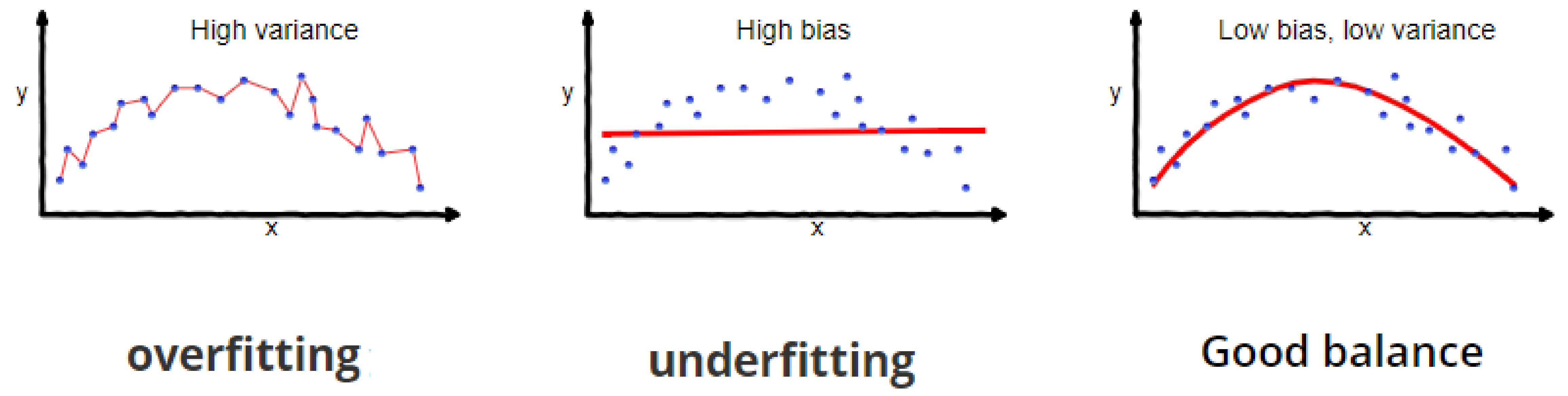
Practically data should be split into three parts:

- Training set. (Has to be the largest set)
- Hold-out cross-validation set / Development or "dev" set.
- Testing set.

Try to build a model upon the training set then try to optimize hyperparameters on the dev set as much as possible. Then after your model is ready you try and evaluate the testing set.

Make sure the dev and test sets are coming from the same distribution

# Bias and Variance



- If the model is underfitting (logistic regression of nonlinear data) it has a "high bias"
- If the model is overfitting then it has a "high variance"
- The model will be all right if you balance the Bias / Variance

# For high Bias or Variance

## Bias

- Try to make the NN bigger (size of hidden units, number of layers)
- Try a different model that is suitable for your data.
- Try to run it longer.
- Different (advanced) optimization algorithms.

## Variance

- More data.
- Try regularization.
- Try a different model that is suitable for your data.

- We should try the previous two points until you have a low bias and low variance.
- Training a bigger neural network never hurts.

## Regularization

- Regularization refers to a set of different techniques that lower the complexity of a neural network model during training and thus prevent overfitting.

- Adding regularization to NN will help it reduce variance (overfitting)
- This technique discourages learning a more complex or flexible model, avoiding the risk of Overfitting.
- It constrains, regularizes, or shrinks the coefficient estimates towards zero.

## L2 Regularization

- The Euclidean Norm (or L2 norm)

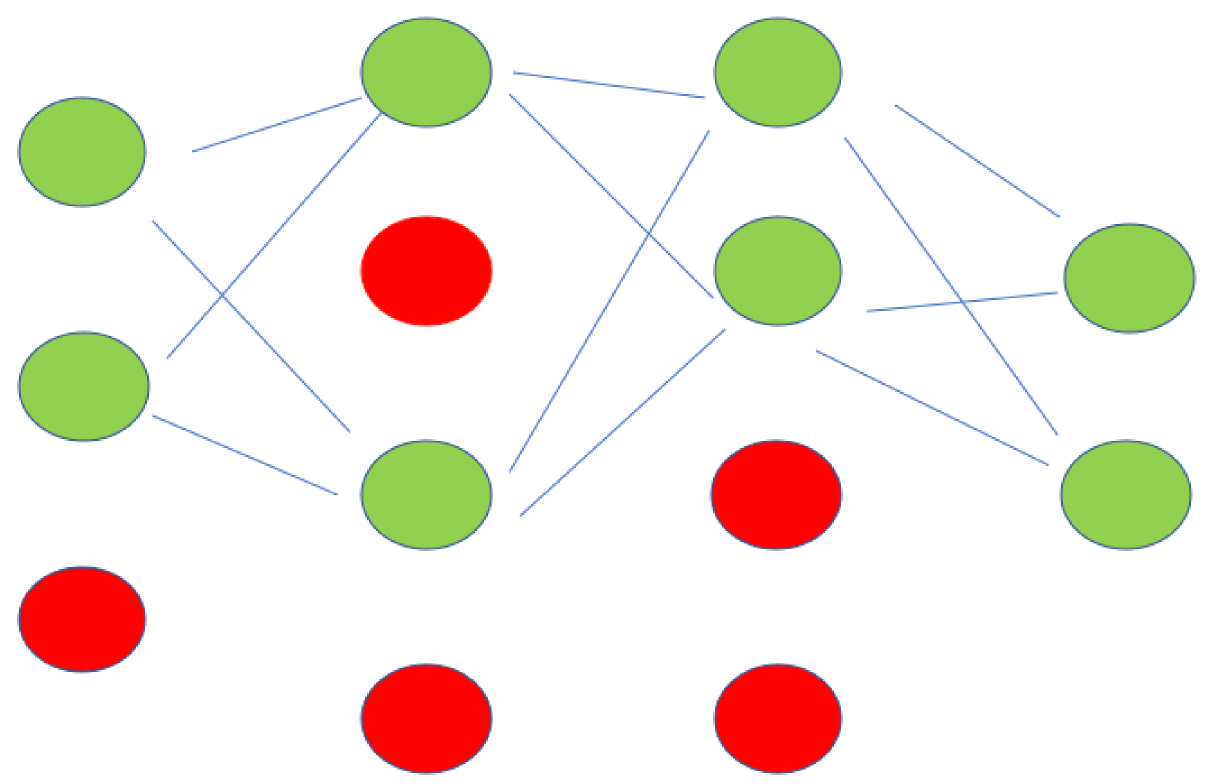
$$\Omega(W) = ||W||_2^2 = \sum_i \sum_j w_{ij}^2$$

- J(w,b) = (1/m) * Sum(L(y(i),y'(i))) + (lambda/2m) * Sum((||W[l]||^2)

- dw[l] = (from back propagation) + lambda/m * w[l]

## Dropout Regularization

- The dropout regularization eliminates some neurons/weights on each iteration based on a probability
- A common dropout technique is inverted dropout in which we initialise our parameters by scaling with some keep_prob matrix
- This way some of parameter's values will be zeroed and so that neuron will be dropped out



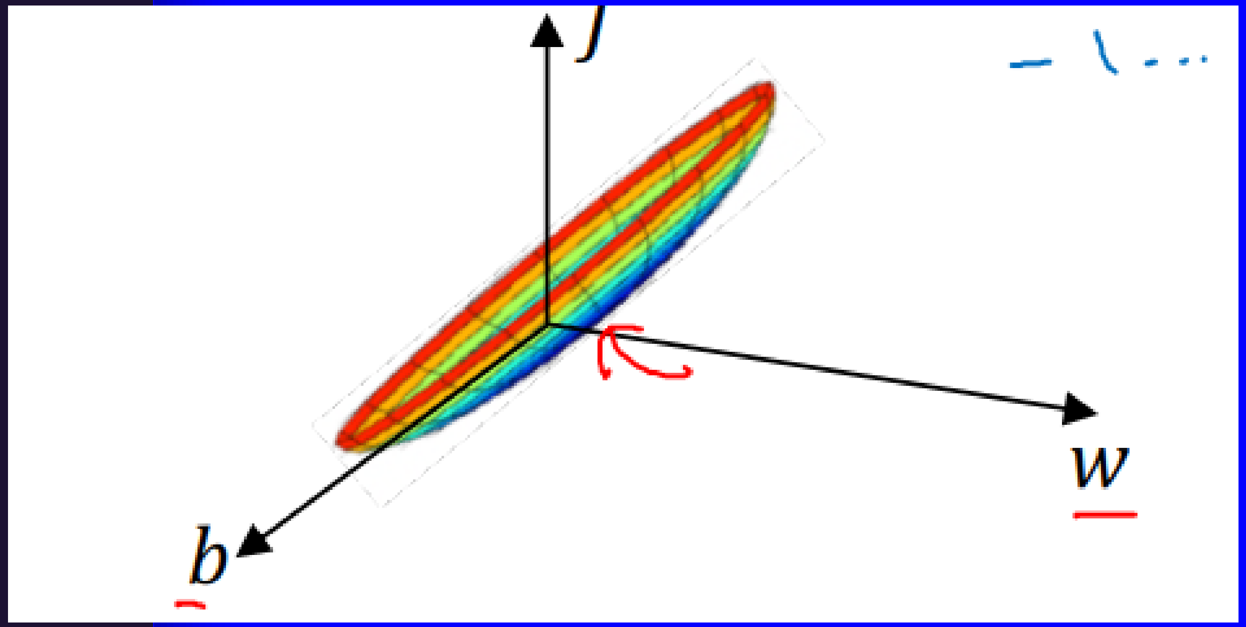
## Other regularization methods

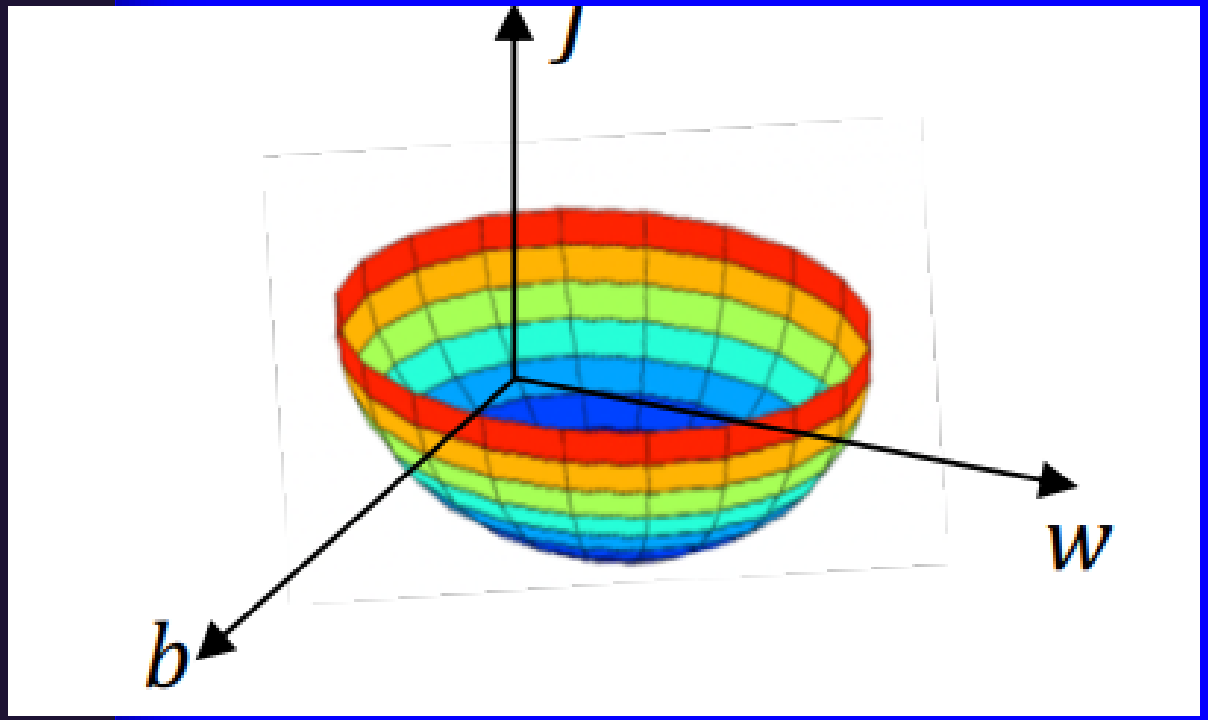
- Data augmentation: Increase the size of the dataset by modifying given data.

- Early stopping: In this technique, we plot the training set and the dev set cost together for each iteration. At some iteration, the dev set cost will stop decreasing and will start increasing, we will take these parameters as the best parameters

## Normalizing inputs

- If we normalize your inputs this will speed up the training process a lot.
- steps:
  - Get the mean of the training set: mean = (1/m) * sum(x(i))
  - Subtract the mean from each input: X = X - mean
  - This makes your inputs centered around 0.
  - Get the variance of the training set: variance = (1/m) * sum(x(i)^2)
  - Normalize the variance. X /= variance
- These steps should be applied to training, dev, and testing sets (but using mean and variance of the train set).





## Weight Initialization for Deep Networks

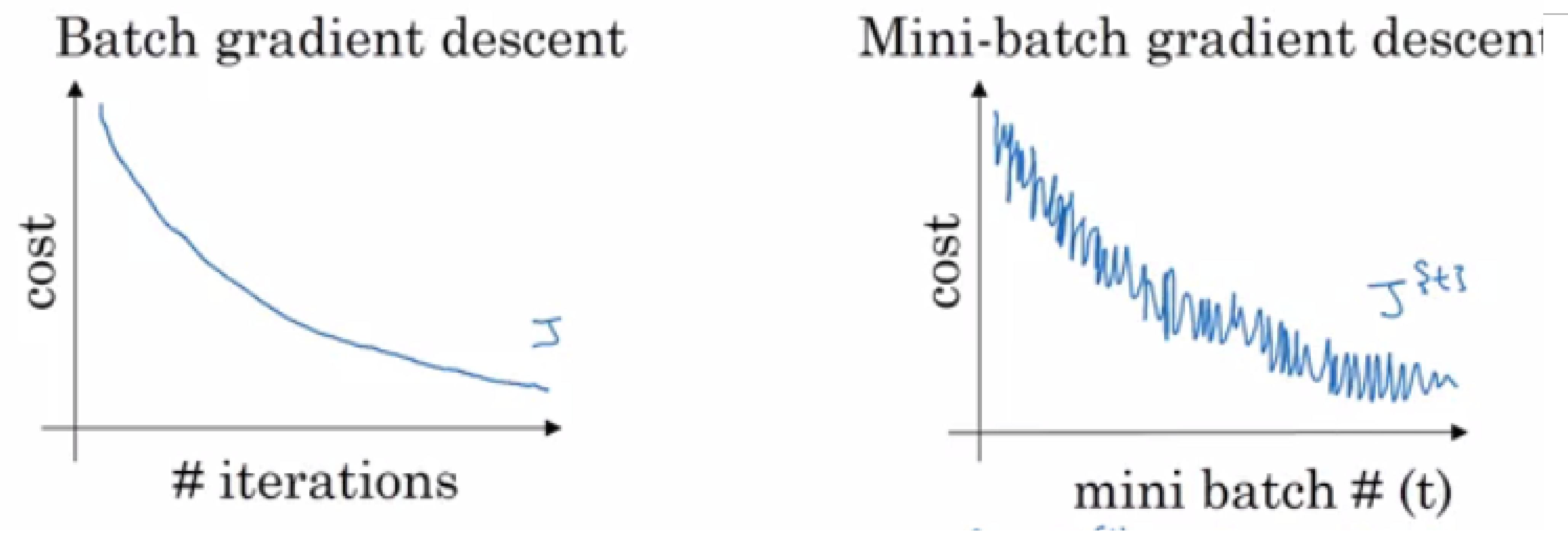
- The Vanishing / Exploding gradients occur when our derivatives become very small or very big and it depends on the initialization of parameters.
- A partial solution to the Vanishing / Exploding gradients in NN is better or more careful choice of the random initialization of weights
- np.random.rand(shape) * np.sqrt(2/n[l-1])

# Mini-batch gradient descent
**(Optimization algorithms)**

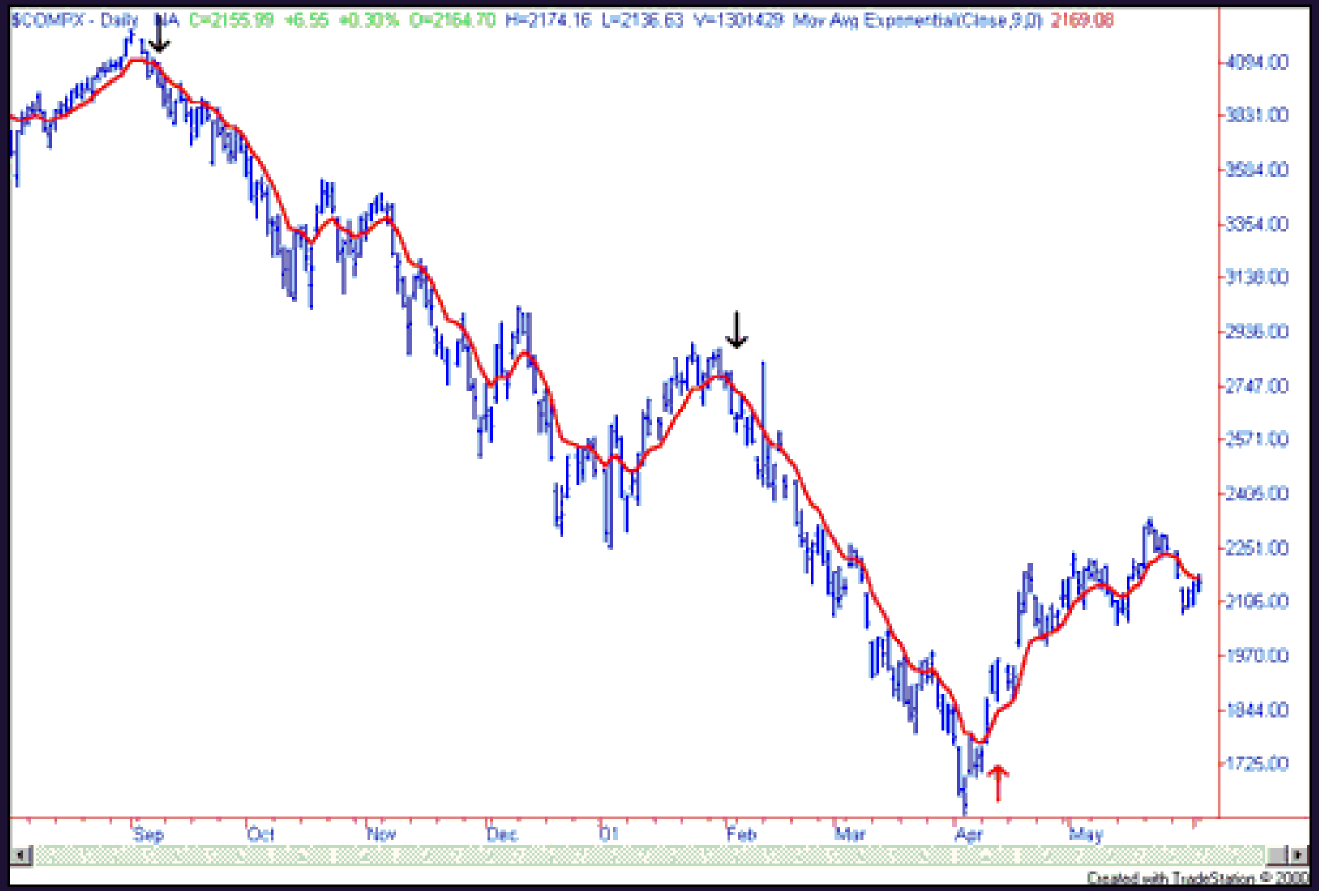
- mini batch size m (Batch gradient descent)
- mini batch size 0 ( Stochastic gradient descent)
- mini batch size between 1 and m ( Mini batch gradient descent)

- Training NN with a large data is slow. So to find an optimization algorithm that runs faster is a good idea.

- We split the dataset into batches of specific sizes with input and output labels both.
- Then we run the gradient descent on the mini datasets.



# Exponentially weighted averages



- The Exponentially Weighted Moving Average (EWMA) is a quantitative or statistical measure used to model or describe a time series. The EWMA is widely used in finance, the main applications being technical analysis and volatility modeling.

$$\mathrm{EWMA}_t = \alpha * r_t + (1 - \alpha) * \mathrm{EWMA}_{t-1}$$

- (Average about last 1/(1-alpha) entries)

- it smoothens out the averages of skewed data points (oscillations w.r.t. Gradient descent terminology). So this reduces oscillations in gradient descent and hence makes faster and smoother path towards minima.
- Bias corrected equation is:

  *v(t) = (beta * v(t-1) + (1-beta) * theta(t)) / (1 - beta^t)*

## Gradient descent with momentum **(Optimization algorithms)**

- The momentum algorithm almost always works faster than standard gradient descent.
- The simple idea is to calculate the exponentially weighted averages for your gradients and then update your weights with the new values.
  - vdW = beta * vdW + (1 - beta) * dW
  - vdb = beta * vdb + (1 - beta) * db
  - W = W - learning_rate * vdW
  - b = b - learning_rate * vdb
- beta is another hyperparameter. beta = 0.9 is very common and works very well in most cases.

## RMSprop
**(Optimization algorithms)**

- Stands for Root mean square prop.
- RMSprop will make the cost function move slower on the vertical direction and faster on the horizontal direction

  - sdW = (beta * sdW) + (1 - beta) * dW^2
  - sdb = (beta * sdb) + (1 - beta) * db^2
  - W = W - learning_rate * dW / sqrt(sdW)
  - b = B - learning_rate * db / sqrt(sdb)
- Then we run the gradient descent on the mini datasets.

- With RMSprop we can increase your learning rate.

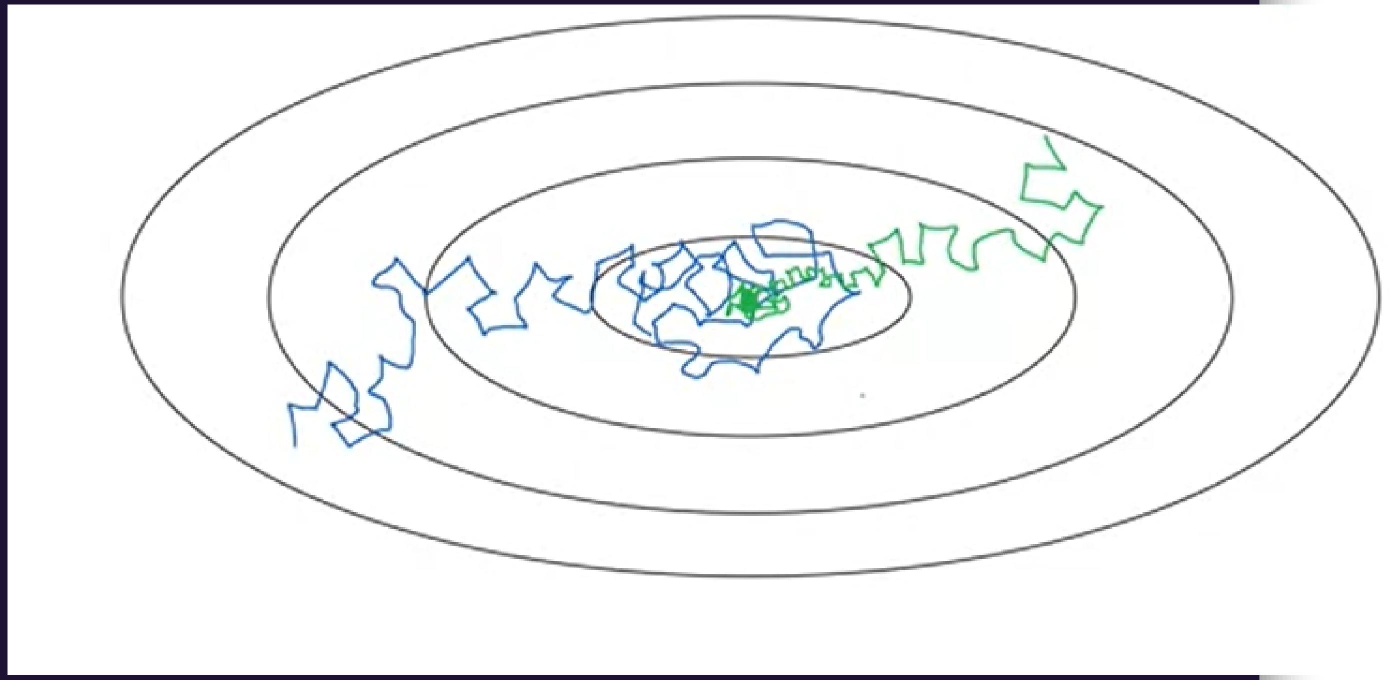
## Adam optimization algorithm
**(Optimization algorithms)**

- Stands for Adaptive Moment Estimation.
- Adam optimization simply puts RMSprop and momentum together!

  - W = W - learning_rate * vdW / (sqrt(sdW) + epsilon)
  - b = B - learning_rate * vdb / (sqrt(sdb) + epsilon)

- Hyperparameters for Adam:
- Learning rate: needed to be tuned.
- beta1: parameter of the momentum - 0.9 is recommended by default.
- beta2: parameter of the RMSprop - 0.999 is recommended by default.
- epsilon: 10^-8 is recommended by default.

# Learning rate decay
**(Optimization algorithms)**



- Slowly reduce learning rate.
- As mentioned before mini-batch gradient descent won't reach the optimum point (converge). But by making the learning rate decay with iterations it will be much closer to it because the steps (and possible oscillations) near the optimum are smaller.

  - learning_rate = (1 / (1 + decay_rate * epoch_num)) * learning_rate_0 b = B - learning_rate * vdb / (sqrt(sdb) + epsilon)
  - learning_rate = (0.95 ^ epoch_num) * learning_rate_0
  - learning_rate = (k / sqrt(epoch_num)) * learning_rate_0

## Normalizing activations in a network:

**Batch Normalization**

- Batch Normalization speeds up learning.
- In practice, normalizing Z[l] is done much more often and that is what Andrew Ng presents.
    - Z_norm[i] = (z[i] - mean) / np.sqrt(variance + epsilon)
    - Z_tilde[i] = gamma * Z_norm[i] + beta

- beta[l] - (n[l], m)
- gamma[l] - (n[l], m)

- gamma and beta are learnable parameters of the model.
- Making the NN learn the distribution of the outputs.

## Softmax Regression

**(Multiclass classification)**

- There are a generalization of logistic regression called Softmax regression that is used for multiclass classification/regression.

  - t = e^(Z[L])
    A[L] = e^(Z[L]) / sum(t)
  - J(w[1], b[1], ...) = - 1 / m * (sum(L(y[i], y_hat[i]))) # i = 0 to m
  - dZ[L] = Y_hat - Y

Orthogonalization: In this have some controls, but each control does a specific task and doesn't affect other controls.

Single number evaluation metric:
F1 score as an average of precision and recall $F1 = 2 / ((1/P) + (1/R))$
P: Precision
R: Recall

Satisfying and Optimizing metric: So we can solve that by choosing a single optimizing metric and deciding that other metrics are satisfying.

Avoidable bias: Training error - Human (Bayes) error
Variance = Dev error - Training error

## Transfer learning

- Apply the knowledge you took in a task A and apply it in another task B.
- To do transfer learning, delete the last layer of NN and it's weights and
    - fine-tuning: keep all the other weights as a fixed weights.
    - retrain all the weights.

## Multi-task learning

In multi-task learning, westart off simultaneously, trying to have one neural network do several things at the same time. And then each of these tasks helps hopefully all of the other tasks.

## End to end learning

- Some systems have multiple stages to implement. An end-to-end deep learning system implements all these stages with a single NN.