LECTURE 02

Feed-forward Neural Nets, Universal Approximation & Gradient Descent

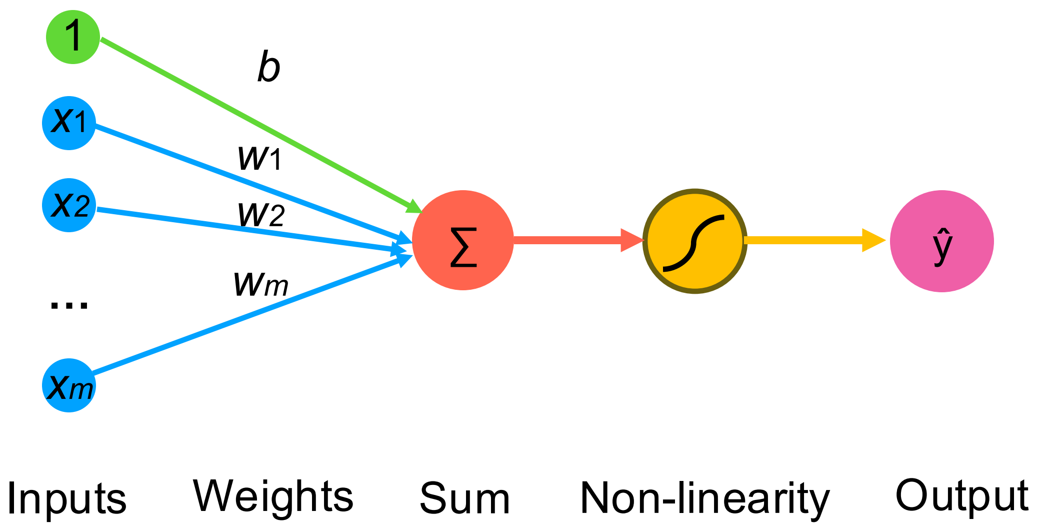
Perceptron, MLP, FFNNs, Universal Approximation Theorem, Basics of Training Neural Nets, Gradient Descent

CS/DS 541: Deep Learning, Fall 2025 @ WPI

Fabricio Murai

The Perceptron

- It also has bias term b that shifts base level (i.e., result when all inputs are zero) even if not explicitly shown



Linear combination of inputs

Output

$$\hat{y} = g\left(b + \sum_{i=1}^{m} x_i\, w_i\right)$$

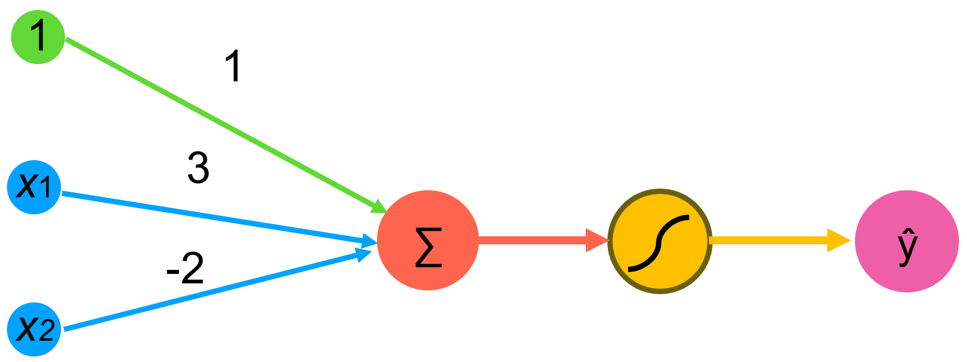
Non-linearity

bias

Vector notation (one instance): $\hat{y} = g\left(b + \mathbf{w}^\top \mathbf{x}\right)$

In this course, we assume vectors are “column”, unless stated otherwise

The Perceptron: Forward Propagation

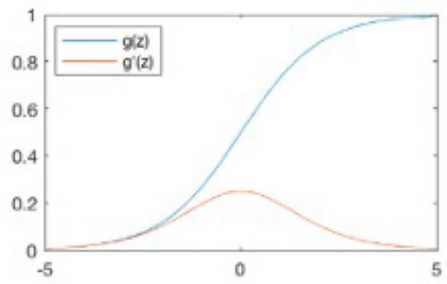


We have: b = 1 and $\mathbf{w} = \begin{bmatrix} 3 \\ -2 \end{bmatrix}$

$$\begin{aligned} \hat{y} &= g\left(b + \mathbf{x}^\top \mathbf{w} \right) \\ &= g\left(1 + \begin{bmatrix} 3 & -2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \right) \\ &= g\left(\underbrace{1 + 3x_1 - 2x_2}_{\text{setting this equal to const C defines a line in 2D!}} \right) \end{aligned}$$

Common Activation Functions

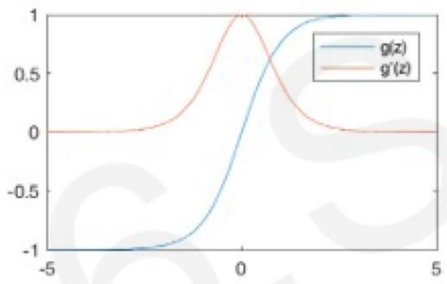
Sigmoid Function



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

$$g'(z) = g(z)(1 - g(z))$$

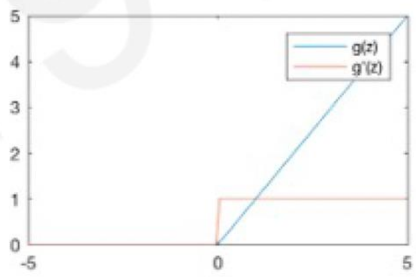
Hyperbolic Tangent



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

$$g'(z) = 1 - g(z)^2$$

Rectified Linear Unit (ReLU)

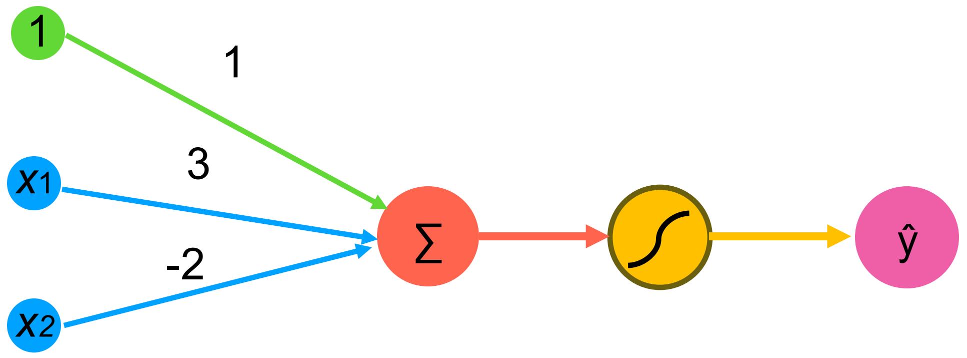


$$g(z) = \max(0, z)$$

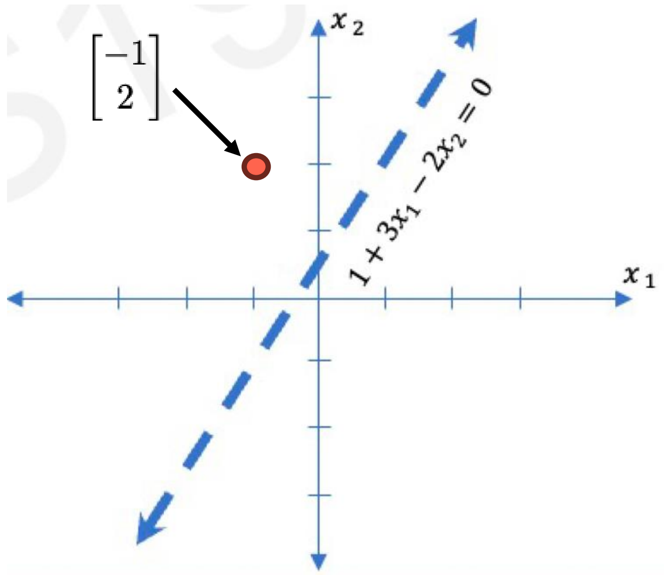
$$g'(z) = \begin{cases} 1, & z > 0 \\ 0, & \text{otherwise} \end{cases}$$

NOTE: All activation functions are non-linear

The Perceptron: Forward Propagation



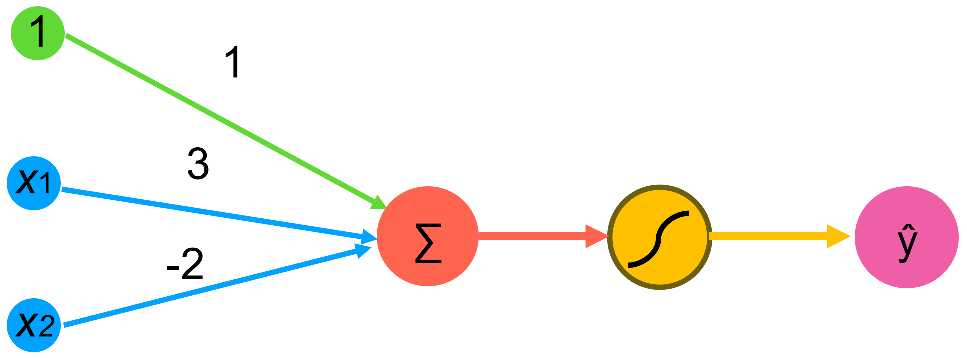
$$\hat{y} = g\,(\,1 + 3x_1 - 2x_2)$$



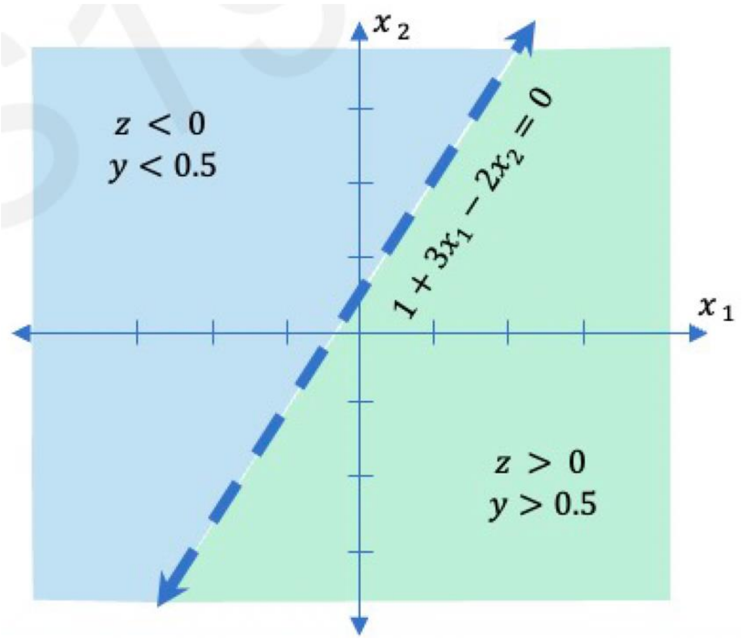
Assume we have input: $\mathbf{x} = \begin{bmatrix} -1 \\ 2 \end{bmatrix}$

$$\begin{aligned} \hat{y} &= g\left(1 + (3 * -1) - (2 * 2)\right) \\ &= g(-6) \approx 0.002 \end{aligned}$$

The Perceptron: Forward Propagation

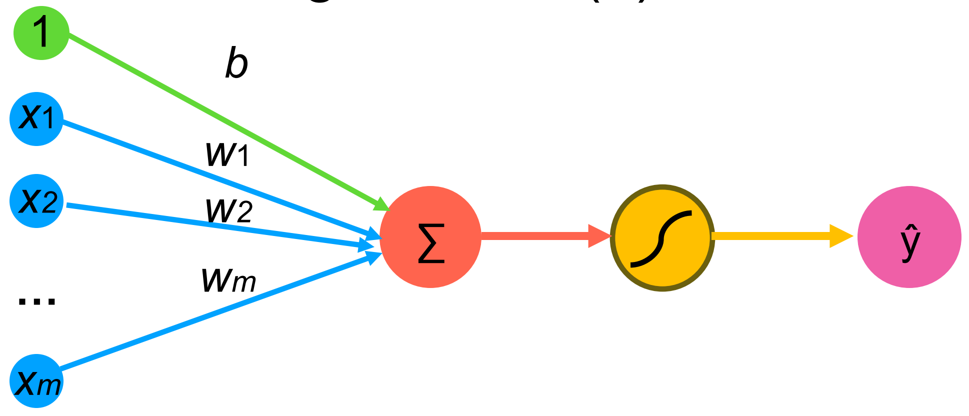


$$\hat{y} = g\,(\underbrace{1 + 3x_1 - 2x_2}_{z})$$



What is the difference between Linear Regression & Perceptron?

- Linear Regression: $f(\mathbf{x}) = w_1x_1 + w_2x_2 + \dots + w_mx_m + b$



$$\hat{y} = g\left(b + \sum_{i=1}^{m} x_i\, w_i\right)$$

Inputs Weights Sum Non-linearity Output

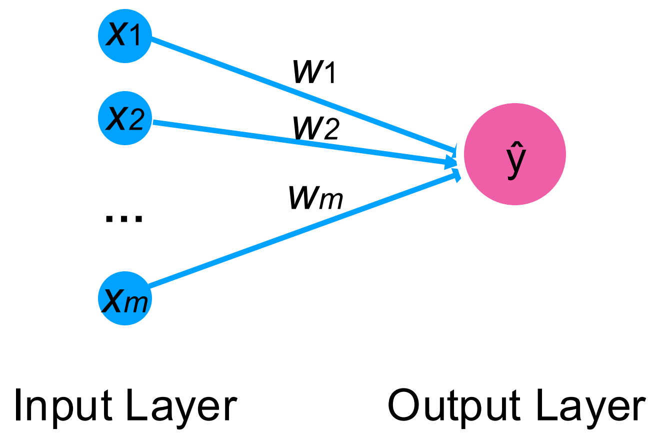
- Replace g by identity function (i.e., $1(z) = z$):

$$\hat{y} = g\left(\sum_{j=1}^{m} w_j x_j + b\right) = \sum_{j=1}^{m} w_j x_j + b$$

How many layers in the perceptron?

It is common to lump together, as a single output layer:

- multiplication by w and sum with b
- transformation g

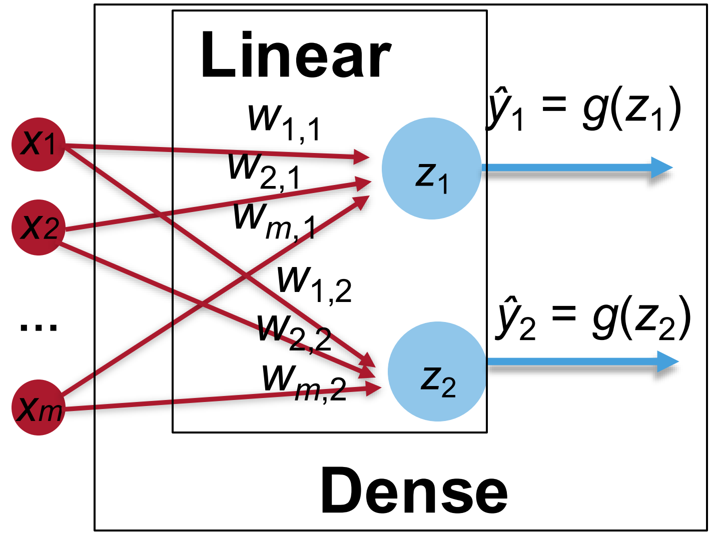


The perceptron has two layers

It is very easy to train

Multi-output Perceptron

- We can easily extend perceptron to output a vector
- Because all inputs are densely connected to all outputs, these layers are called **Dense** layers
  - If non-linearity *g* not included, are called **Linear** layers



$$z_i = b_i + \sum_{j=1}^{m} x_j \, w_{j,i}$$

In matrix notation:

(Our course's convention; column vector notation)

$$\mathbf{z} = \mathbf{W}\mathbf{x} + \mathbf{b}$$

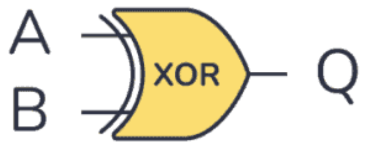
(Alternative convention; row vector notation)

$$\mathbf{z} = \mathbf{x}\mathbf{W} + \mathbf{b}$$

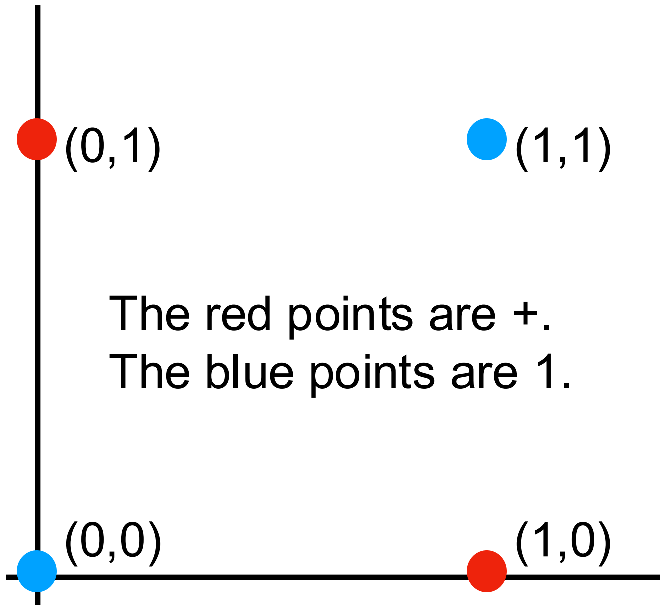
WPI

Limitations of Perceptron

- For binary classification, perceptron corresponds to a single line decision boundary.
- Perceptron can only solve linearly separable problems (Minsky & Papert 1969)
- CANNOT solve the XOR problem:



A	B	Q
0	0	0
0	1	1
1	0	1
1	1	0



The red points are +.
The blue points are 1.

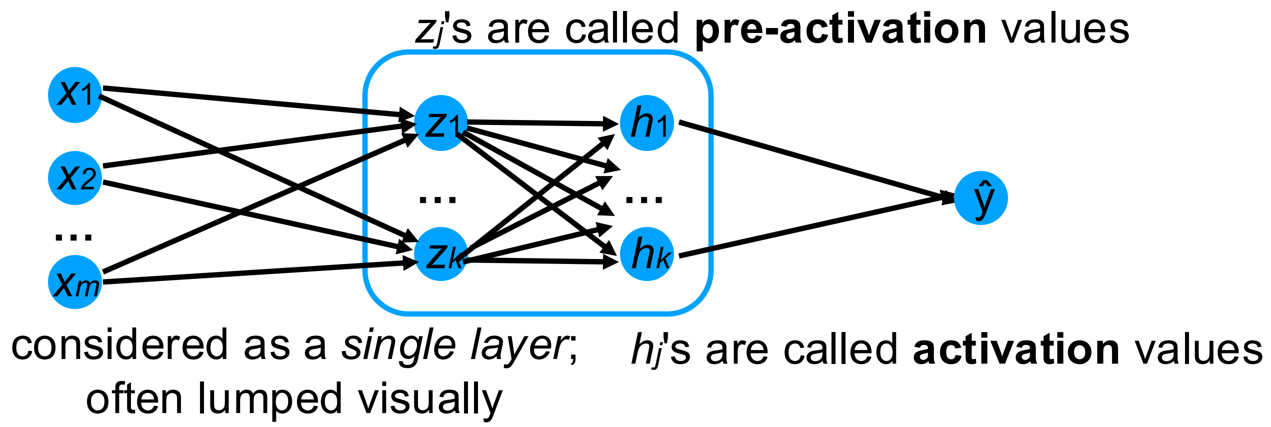
This discovery was a pivotal and sobering moment in the early history of AI. This limitation, highlighted by Minsky and Papert, was so profound that it significantly dampened enthusiasm and funding for neural network research, contributing to a period known as the 1st "AI winter." It wasn't until the development of multi-layer perceptrons and the backpropagation algorithm years later that researchers could overcome this fundamental hurdle.

Multi-Layer Perceptron (MLP)

- We can add ≥ 1 intermediate layers to get **more complex/interesting combinations** of the inputs.
- MLP: each intermediate (aka hidden) layer is a linear combination of the previous "neurons":

$$z_1 = \sum_{j=1}^{m} w_{1j} x_j$$

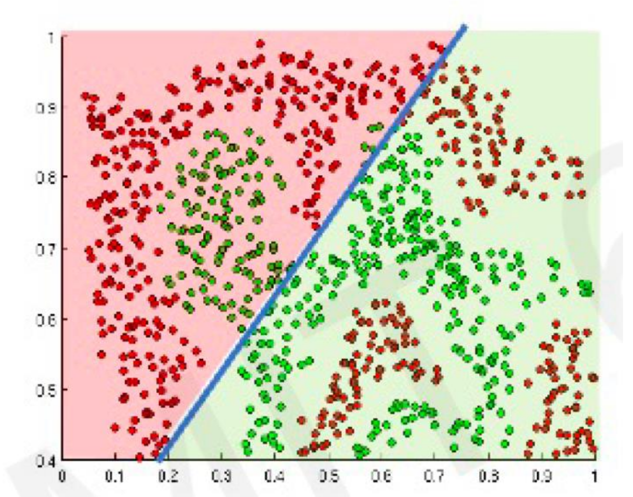
followed by a non-linear activation **h** = *g*(**z**).
- **Example**: MLP with 1 intermediate layer



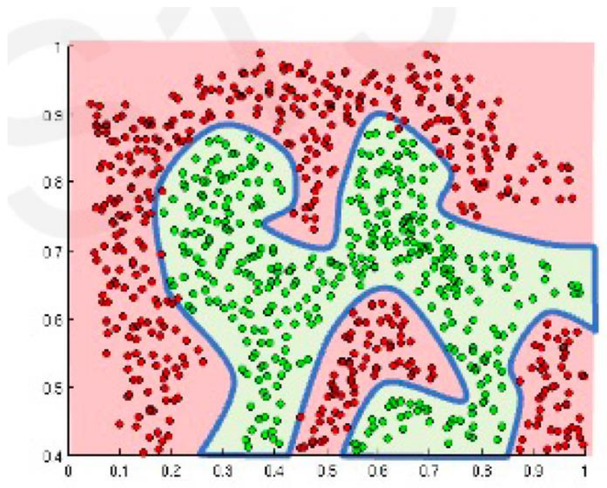
z_j's are called **pre-activation** values
x_1
x_2
...
x_m
z_1
...
z_k
h_1
...
h_k
ŷ
considered as a *single layer*; often lumped visually
h_j's are called **activation** values

Importance of Activation Functions

Combination of intermediate layers + activation functions allows us to **introduce non-linearities** into the network



Linear activation functions produce linear decisions no matter the network size



Non-linearities allow us to approximate arbitrarily complex functions

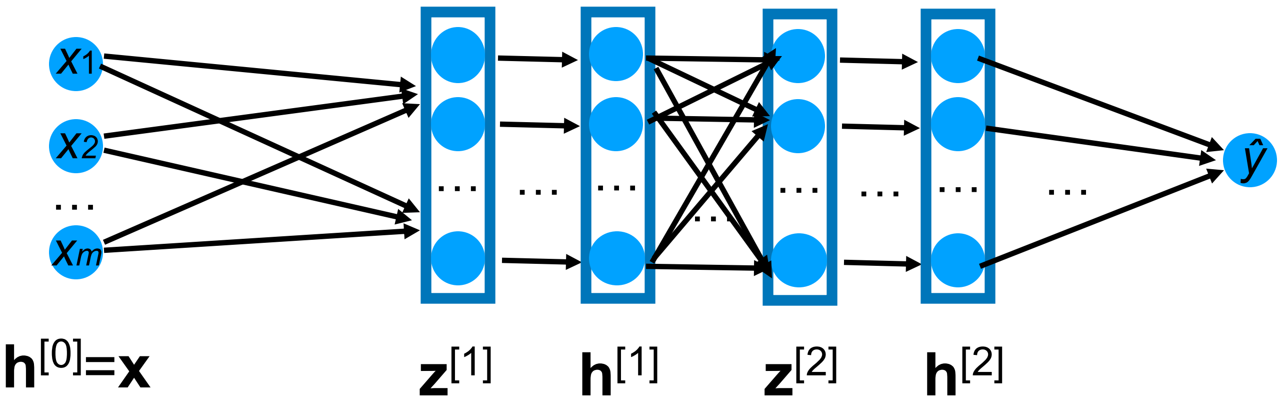
Multi-Layer Perceptron (MLP)

Example: MLP with 2 intermediate layers

- In MLP: at layer *l*, pre-activation values $\mathbf{z}^{[l]}$ are linear combinations of activation values from previous layer $\mathbf{h}^{[l-1]}$:

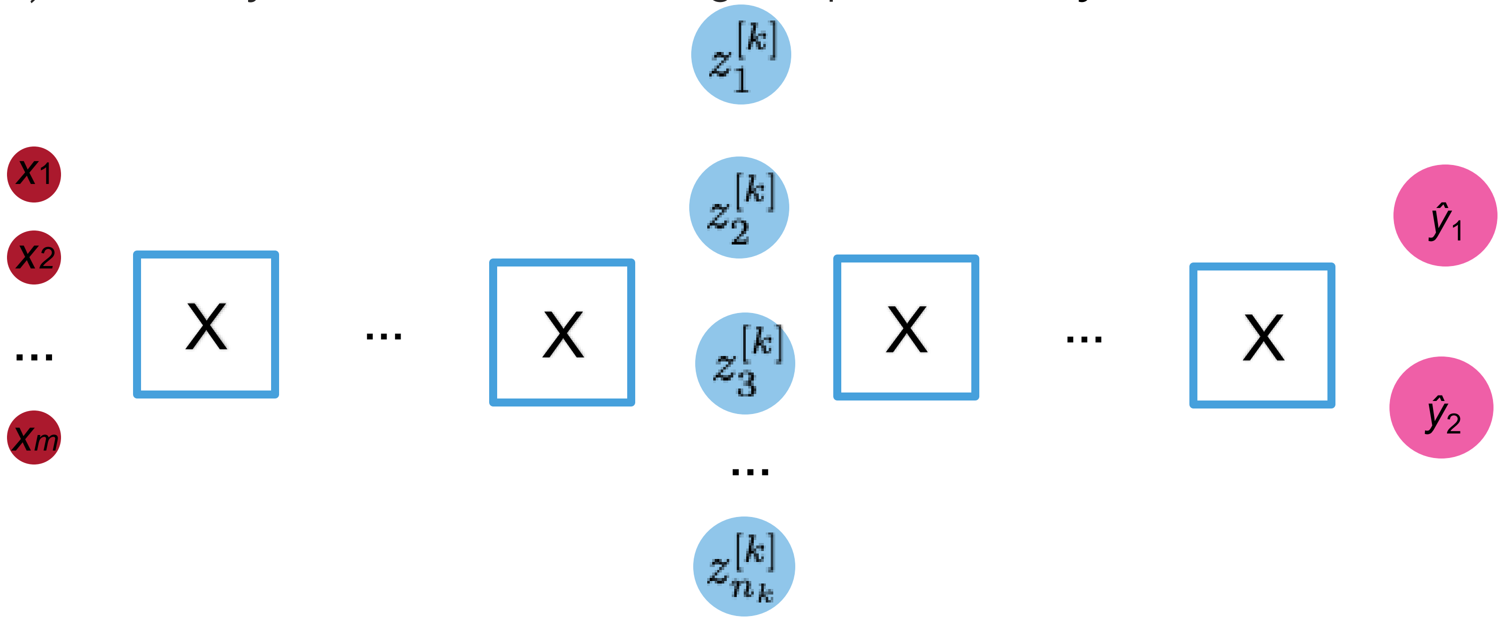
$$z_k^{[l]} = \sum_{j=1}^{m} w_{kj}^{[l]} h_j^{[l-1]}$$

- Next, they are transformed by activation $\mathbf{h}^{[l]} = g(\mathbf{z}^{[l]})$.



Forward Propagation

- Forward propagation is the process of computing intermediate and final outputs starting from inputs
- Starting from *x*, compute pre-activation values $z^{[k]}$ and activation values $g(z^{[k]})$ for all layers k, until obtaining output values $\hat{y}$



$$z_i^{[k]} = b_i^{[k]} + \sum_{j=1}^{n_{k-1}} w_{i,j}^{[k]} \, g(z_j^{[k-1]})$$

Feed-forward Neural Networks

- A common network design is a **feed-forward neural network**.
  - Consists of multiple layers of neurons, each of which feeds to the next layer (except the last).
  - MLP is an example of FFNN which uses linear layers.
- Other examples:
  - Convolutional Neural Network (CNN), which use convolutional layers
  - Residual Networks: use skip connections
- A Recurrent Neural Network (RNN) is not considered FFNN (more details in a few weeks).

What is "deep"?

- Much of DL (and ML in general) can be formulated as computing a function *f* — which defines the behavior of a **machine** — that transforms input **x** into desired output *ŷ*

- In "classical" ML (e.g., SVMs, boosting, decision trees), *f* is often a "shallow" function of **x**, e.g.:

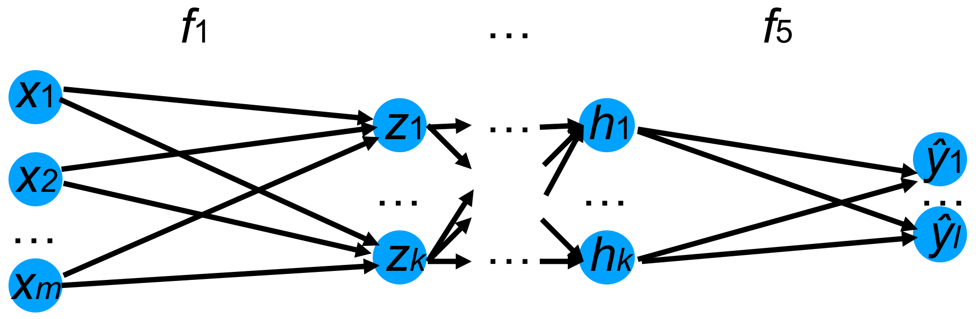
$$\hat{y} = f(\mathbf{x}) = \mathbf{x}^\top \mathbf{w}$$

- In contrast, with DL, *f* is the **composition** (possibly 1000s of "layers" deep!) of many functions, e.g.:

$$f(\mathbf{x}) = f_5(f_4(f_3(f_2(f_1(\mathbf{x})))))$$

What is “deep”?

- Architecturally, this corresponds to an artificial neural network with many layers:



$$f(\mathbf{x}) = f_5(f_4(f_3(f_2(f_1(\mathbf{x})))))$$

Tensorflow Playground

- What can we do with MLPs with 0-6 hidden layers? https://playground.tensorflow.org

What functions can we represent with 1 hidden layer and a sigmoid?

- Done on whiteboard

Neural Nets are Universal Function Approximators

Universal function approximation theorem

- Many papers in 1980s-1990s established several universal approximation theorems for **arbitrary width** and bounded depth.
- (Cybenko 1989) For any closed, bounded, continuous function *f* and any ϵ, we can train a feed-forward 3-layer NN $\hat{f}$ with **sigmoidal activation functions** and **sufficiently many neurons** in the hidden layer such that:
$$|f(x) - \hat{f}(x)| < \epsilon \quad \forall x$$

  Theorem also generalizes to *f* with multidimensional inputs and outputs.
- More general version (Leshno et al. 1993, Pinkus 1999) showed that universal approximation property holds if and only if activation is non-polynomial.

Cybenko, G. (1989). "Approximation by superpositions of a sigmoidal function." *Mathematics of Control, Signals, and Systems*.
Leshno, M; Lin, V Y.; Pinkus, A; Schocken, S (1993). "Multilayer feedforward networks with a nonpolynomial activation function can approximate any function". Neural Networks.
Pinkus, Allan (1999). "Approximation theory of the MLP model in neural networks". *Acta Numerica*.

Proof idea

- Using pairs of sigmoid functions, we can construct delta functions that represent vertical “bars”.
- Using enough vertical bars, we can approximate any *f* (akin to the trapezoidal rule of calculus).
- See visual proof sketch here.

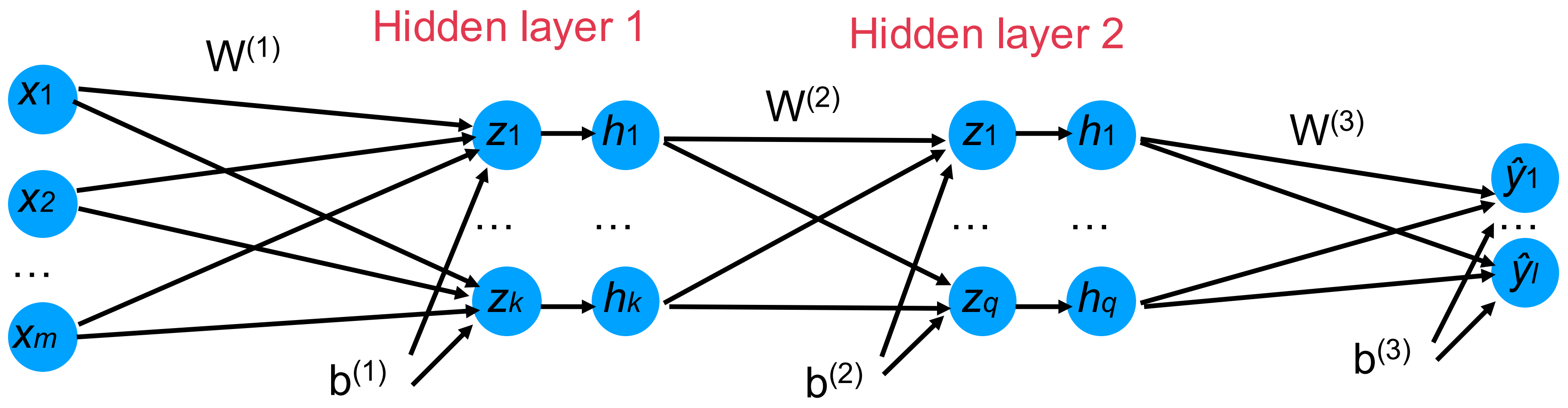
Universal function approximation theorem

- Dual versions of the theorem: consider networks of bounded width and arbitrary depth.
- (Lu et al. 2017) Networks of width *n*+4 with ReLu activation functions can approximate any Lebesgue-integral function on *n*-dimensional input space with respect to L^1 distance if network depth is allowed to grow.

Recap: Feed-Forward Neural Networks

- Feed-forward neural network consists of layers that apply:
  - A *linear** transformation of inputs h i.e., hW + b
  - Followed by an activation function e.g., sigmoid, tanh, ReLU
- Example with 2 hidden layers:

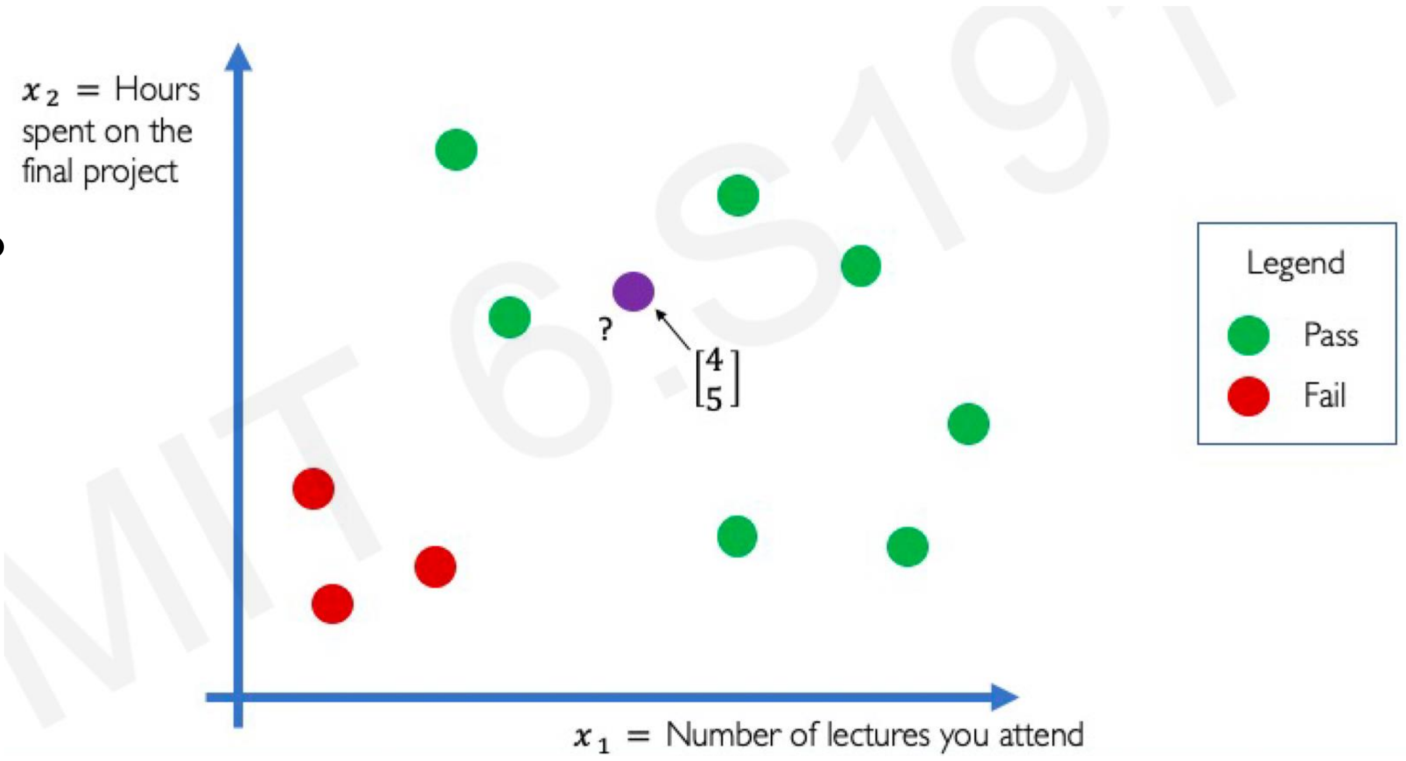
Activation Functions. Which one…
Has only 0 or 1 as derivatives?
Crosses x=0 at y=0.5?
Is bounded between -1 and 1?



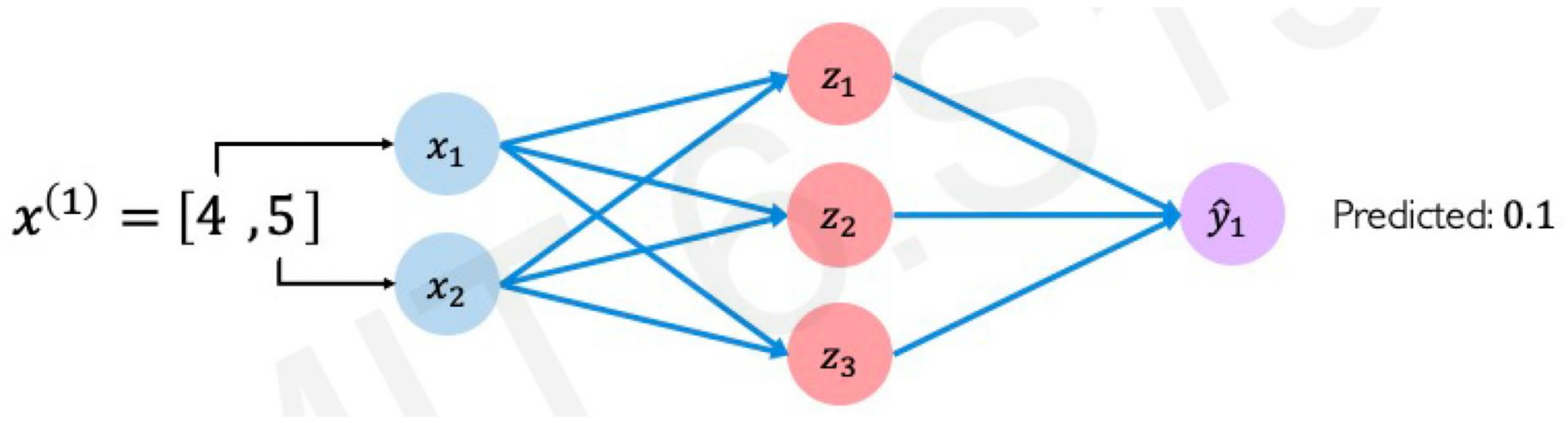
* Technically, if there is a bias term, it is an *affine* transformation

Example Problem

-

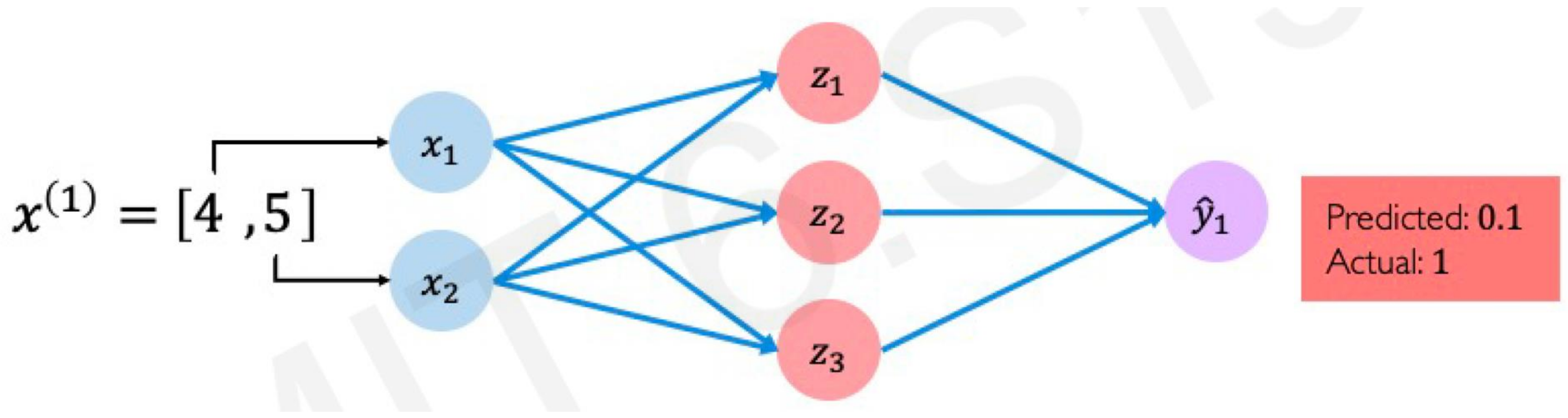


Example problem: Will I pass this class?

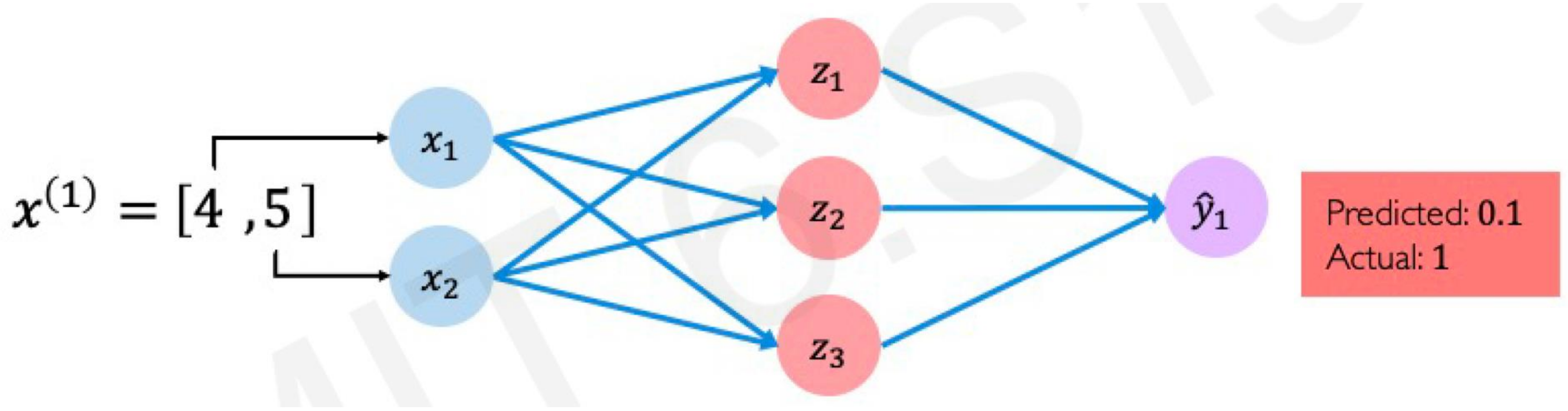


Ingredient #1
(Model)

Example problem: Will I pass this class?



Loss Function

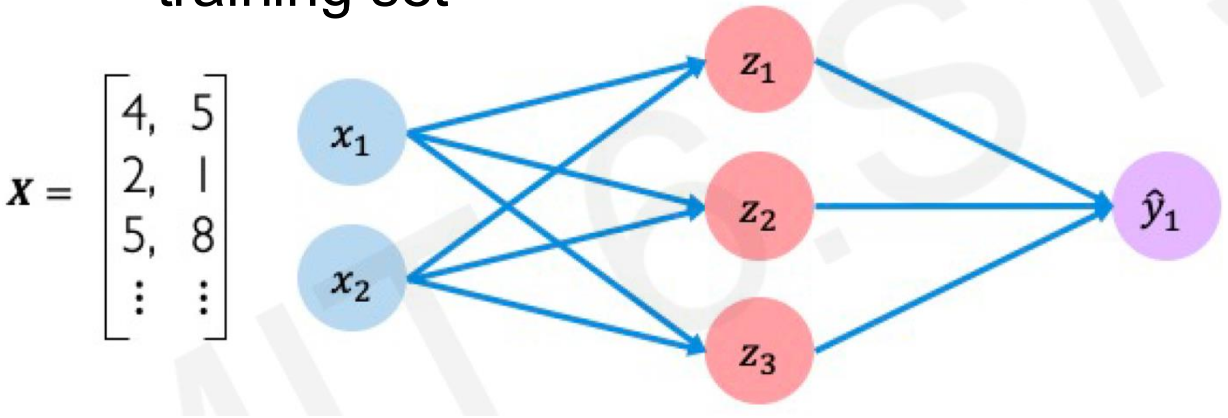


Ingredient #2

$$\mathcal{L}\big(\underbrace{f\left(x^{(i)};\boldsymbol{W}\right)}_{\text{Predicted}},\underbrace{y^{(i)}}_{\text{Actual}}\big)$$

Cost Function

- During training, we need to compute loss over the entire training set



$X = \begin{bmatrix} 4, & 5 \\ 2, & 1 \\ 5, & 8 \\ \vdots & \vdots \end{bmatrix}$
x_1
x_2
z_1
z_2
z_3
$\hat{y}_1$

$$\begin{array}{cc} \hat{y} = f(x) & y \\ \begin{bmatrix} 0.1 \\ 0.8 \\ 0.6 \\ \vdots \end{bmatrix} & \begin{bmatrix} 1 \\ 0 \\ 1 \\ \vdots \end{bmatrix} \end{array}$$

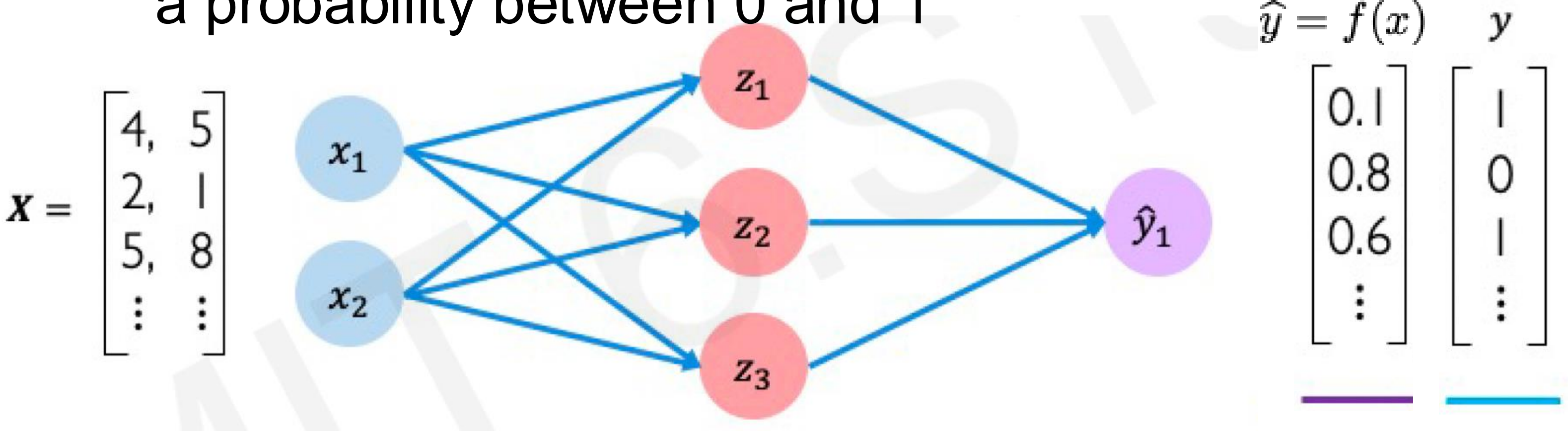
Though 3rd prediction is correct, still incurs loss (why?)

$$J(W) = \frac{1}{n}\sum_{i=1}^{n} \mathcal{L}\Big(\underbrace{f\big(x^{(i)}; W\big)}_{\text{Predicted}}, \underbrace{y^{(i)}}_{\text{Actual}}\Big)$$

Also known as:
- Objective function
- Empirical loss
- Empirical risk

Binary Cross Entropy Loss

- **Cross entropy loss** can be used with models that output a probability between 0 and 1

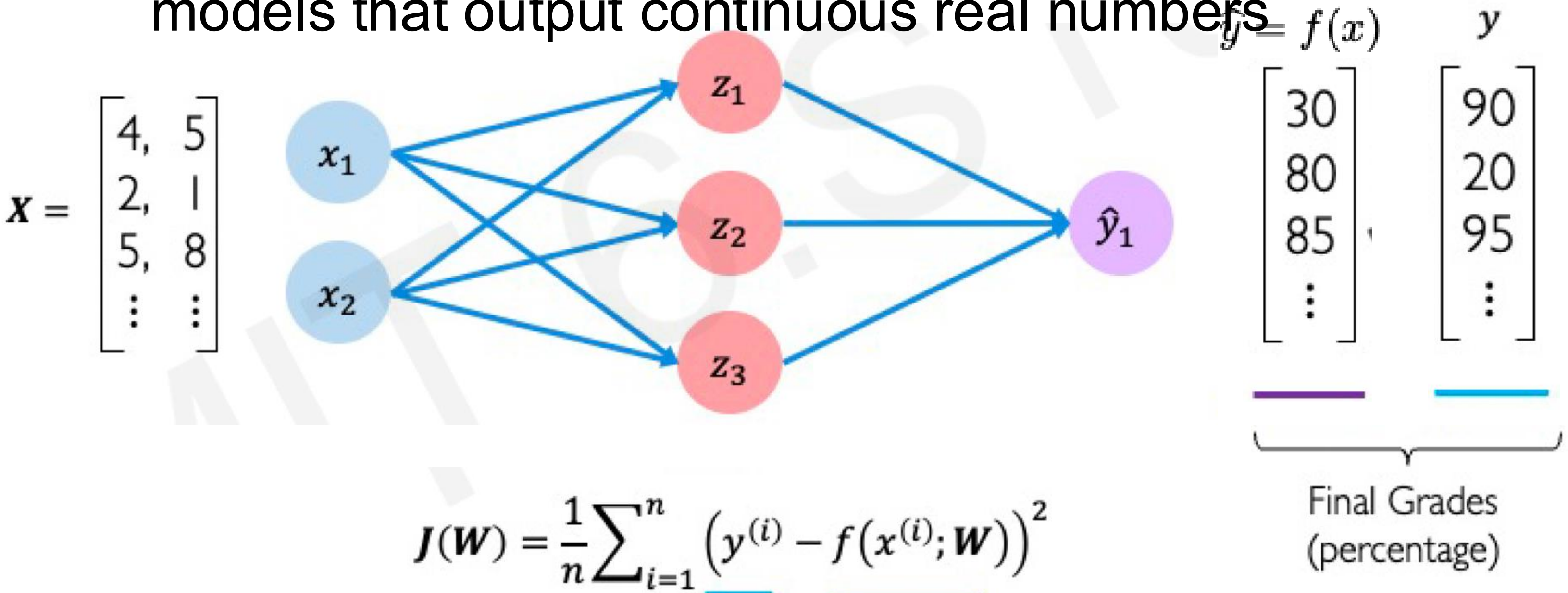


$X = \begin{bmatrix} 4, & 5 \\ 2, & 1 \\ 5, & 8 \\ \vdots & \vdots \end{bmatrix}$
x_1
x_2
z_1
z_2
z_3
$\hat{y}_1$
$\hat{y} = f(x)$
y
$\begin{bmatrix} 0.1 \\ 0.8 \\ 0.6 \\ \vdots \end{bmatrix}$
$\begin{bmatrix} 1 \\ 0 \\ 1 \\ \vdots \end{bmatrix}$

$$J(\boldsymbol{W}) = -\frac{1}{n}\sum_{i=1}^{n} \underbrace{y^{(i)}}_{\text{Actual}} \log\left(\underbrace{f\left(x^{(i)};\boldsymbol{W}\right)}_{\text{Predicted}}\right) + (1 - \underbrace{y^{(i)}}_{\text{Actual}}) \log\left(1 - \underbrace{f\left(x^{(i)};\boldsymbol{W}\right)}_{\text{Predicted}}\right)$$

Mean Squared Error Loss

- **Mean squared error (MSE) loss** can be used with models that output continuous real numbers



$X = \begin{bmatrix} 4, & 5 \\ 2, & 1 \\ 5, & 8 \\ \vdots & \vdots \end{bmatrix}$
x_1
x_2
z_1
z_2
z_3
$\hat{y}_1$
$\hat{y} = f(x)$
y
$\begin{bmatrix} 30 \\ 80 \\ 85 \\ \vdots \end{bmatrix}$
$\begin{bmatrix} 90 \\ 20 \\ 95 \\ \vdots \end{bmatrix}$
Final Grades (percentage)

$$J(\boldsymbol{W}) = \frac{1}{n}\sum_{i=1}^{n}\Big(\underbrace{y^{(i)}}_{\text{Actual}} - \underbrace{f\big(x^{(i)};\boldsymbol{W}\big)}_{\text{Predicted}}\Big)^2$$

Loss Optimization

Ingredient #3

- Assume for now we want to find network weights that achieve the lowest loss*

$$\boldsymbol{W}^* = \underset{\boldsymbol{W}}{\operatorname{argmin}} \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}\left(f\left(x^{(i)}; \boldsymbol{W}\right), y^{(i)}\right)$$

$$\boldsymbol{W}^* = \underset{\boldsymbol{W}}{\operatorname{argmin}} \, J(\boldsymbol{W})$$

$\uparrow$

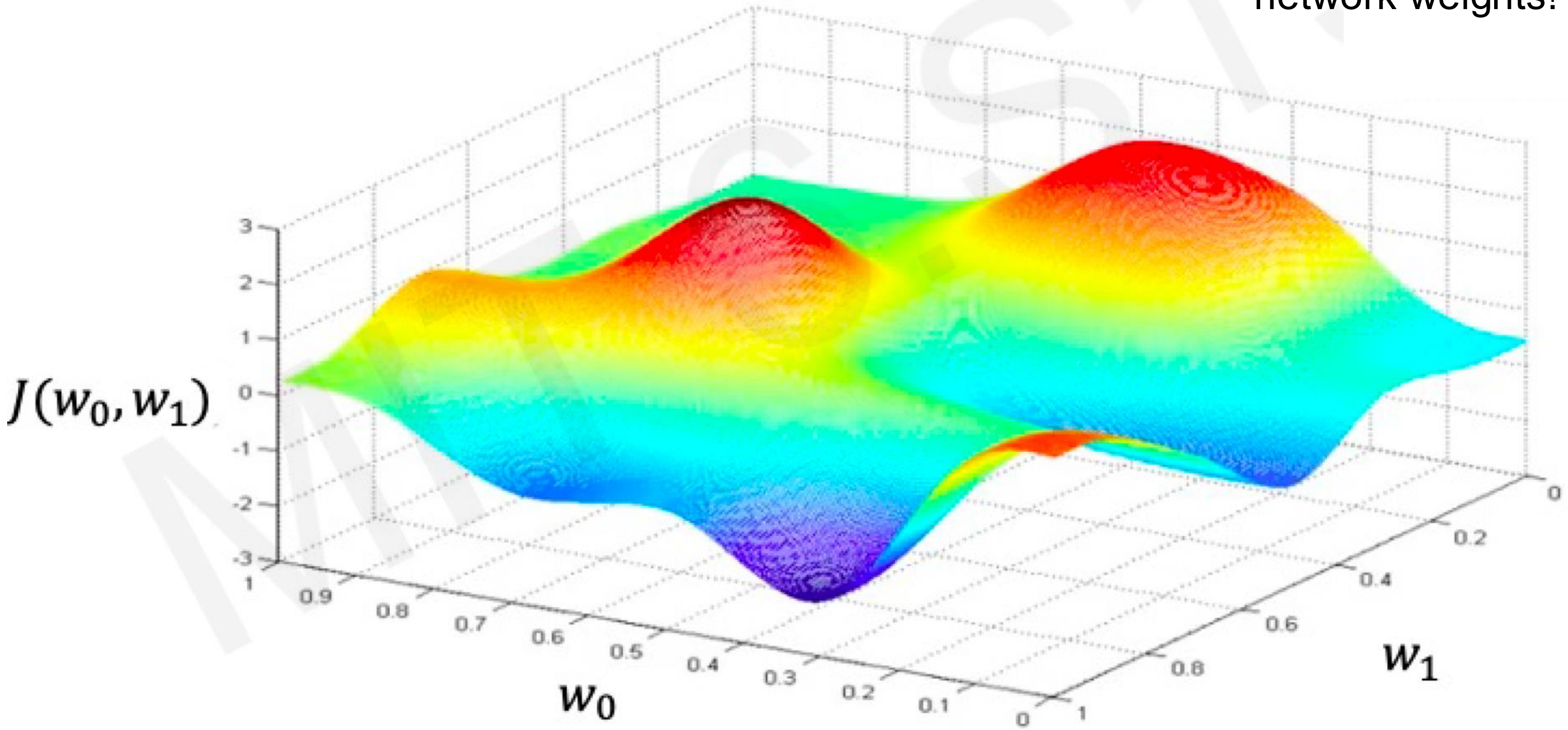
Remember:
$\mathbf{W} = \{ \mathbf{W}^{[1]}, \mathbf{W}^{[1]}, \ldots, \mathbf{W}^{[L]} \}$

*We'll see in the next slides why this is not exactly true…

Loss Optimization

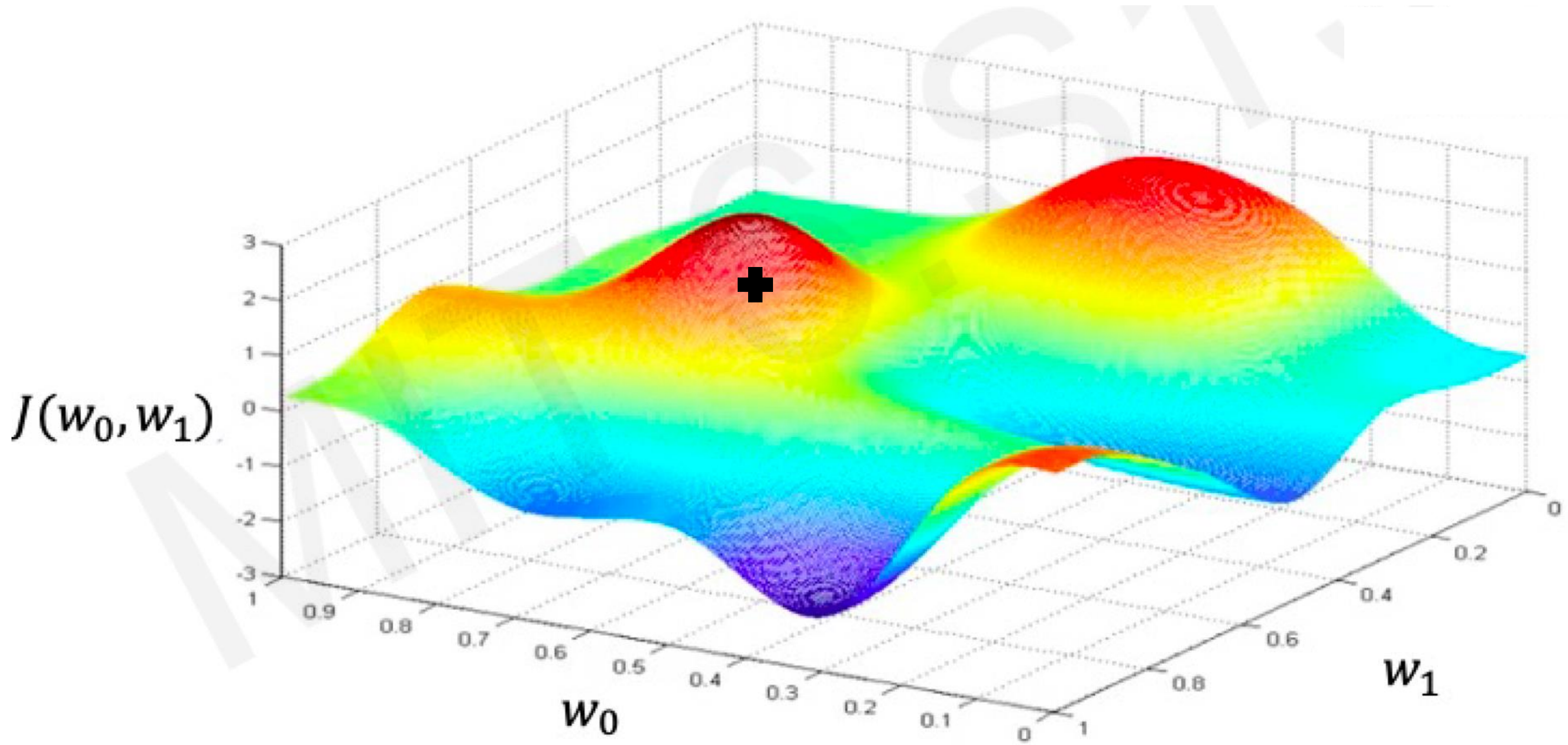
$$W^* = \underset{W}{\operatorname{argmin}}\, J(W)$$

Remember: Our loss is a function of the network weights!



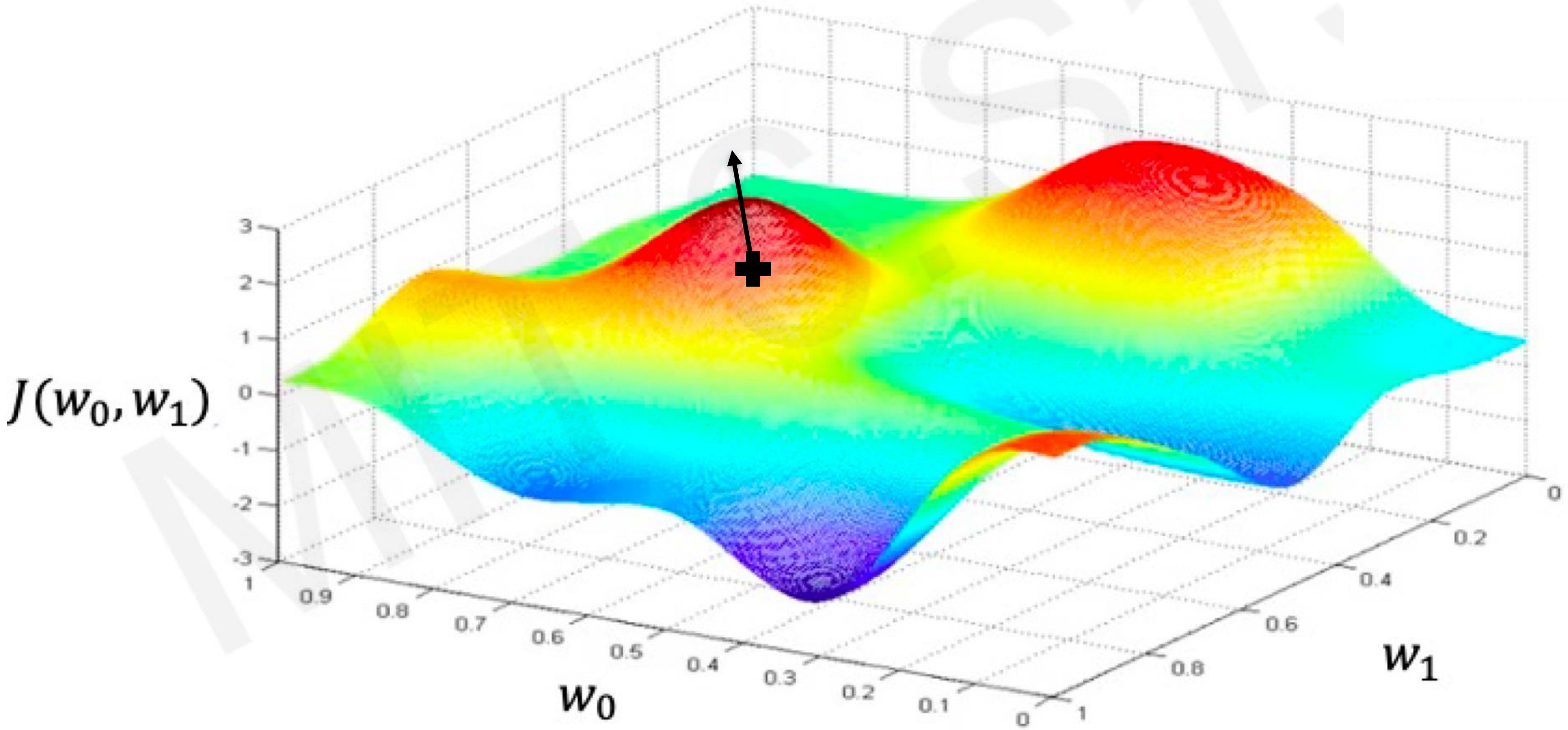
Loss Optimization

- Randomly pick an initial (w_0, w_1)



Loss Optimization

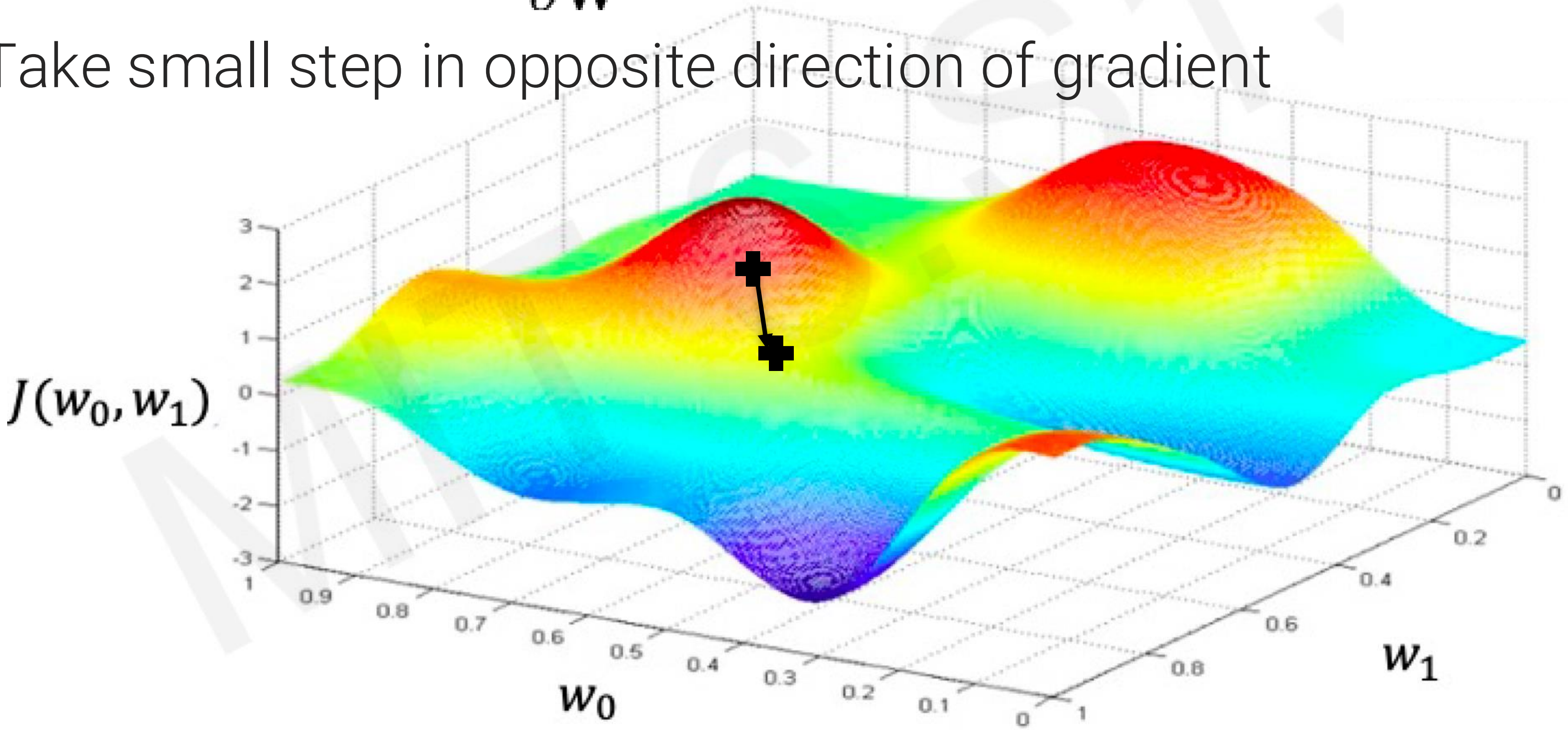
- Randomly pick an initial (w_0, w_1)
- Compute gradient* $\frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$



* In this course, I will use same notation/terminology as in MIT 6.S191 Intro to DL course. Many refs define W in a transpose manner and distinguish between Jacobian and gradient.

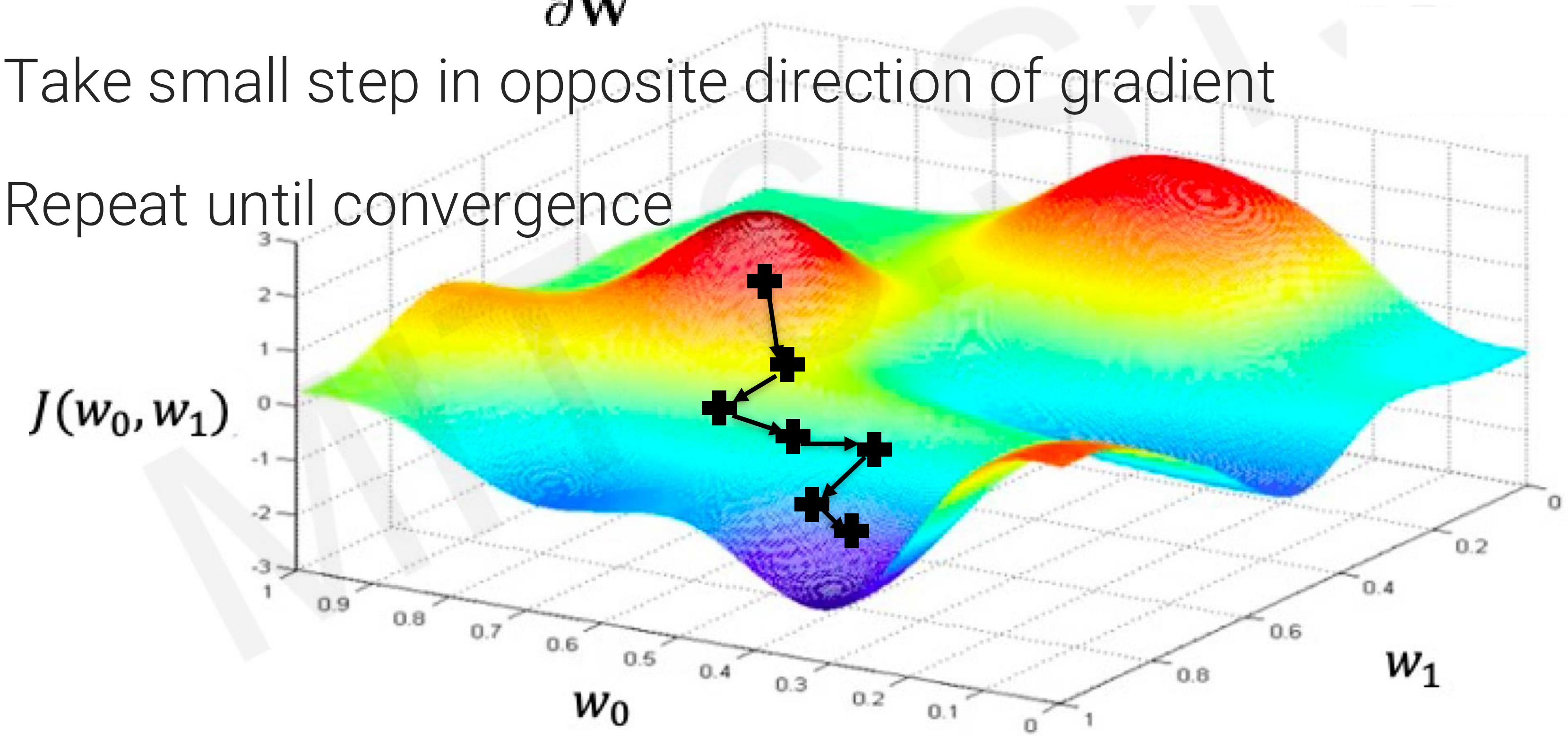
Loss Optimization

- Randomly pick an initial (w_0, w_1)
- Compute gradient $\frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$
- Take small step in opposite direction of gradient

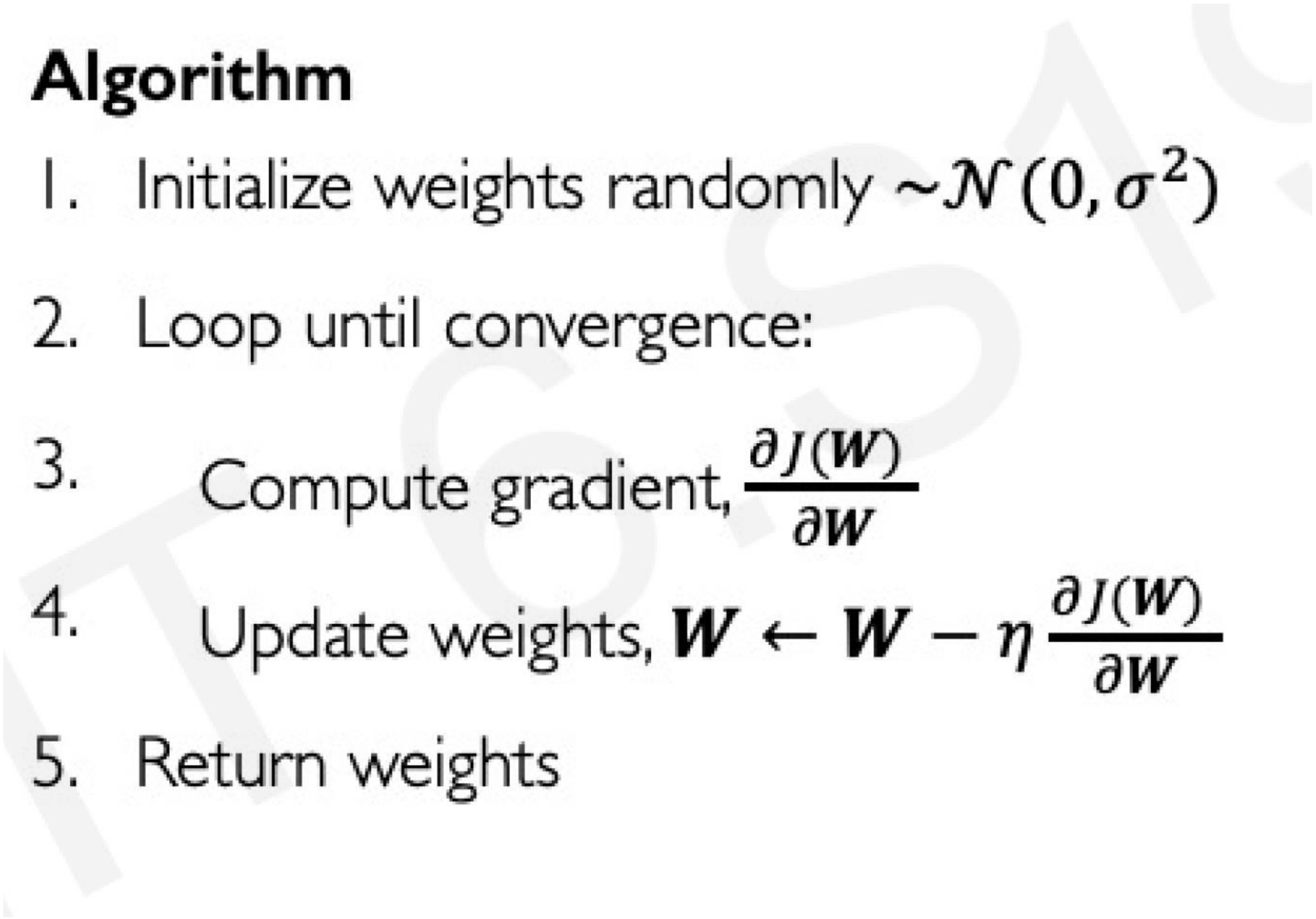


Loss Optimization

- Randomly pick an initial (w_0, w_1)
- Compute gradient $\frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$
- Take small step in opposite direction of gradient
- Repeat until convergence



Gradient Descent



Algorithm

```
1. Initialize weights randomly $\sim\mathcal{N}(0,\sigma^2)$
2. Loop until convergence:
3.     Compute gradient, $\frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
4.     Update weights, $\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
5. Return weights
```

Training Neural Networks is Difficult

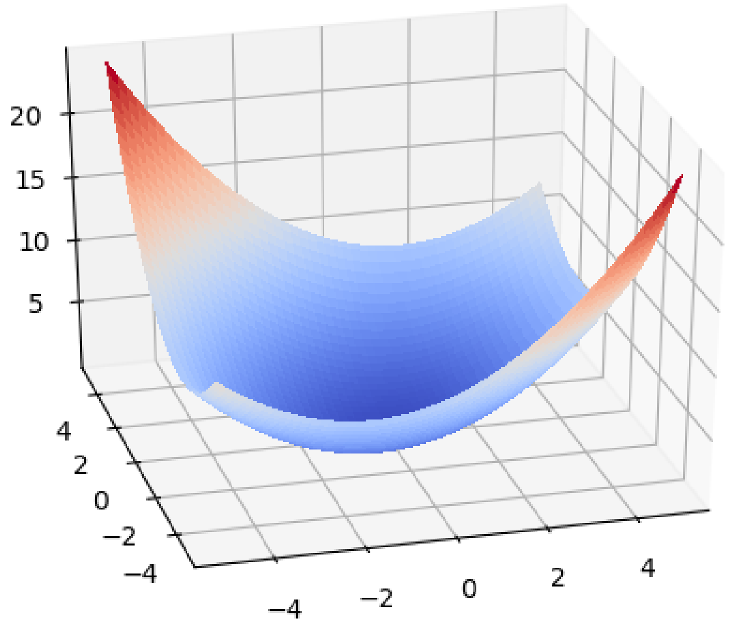


__Fig. 1__: Loss landscape of a linear regression with 2 parameters (source[1])

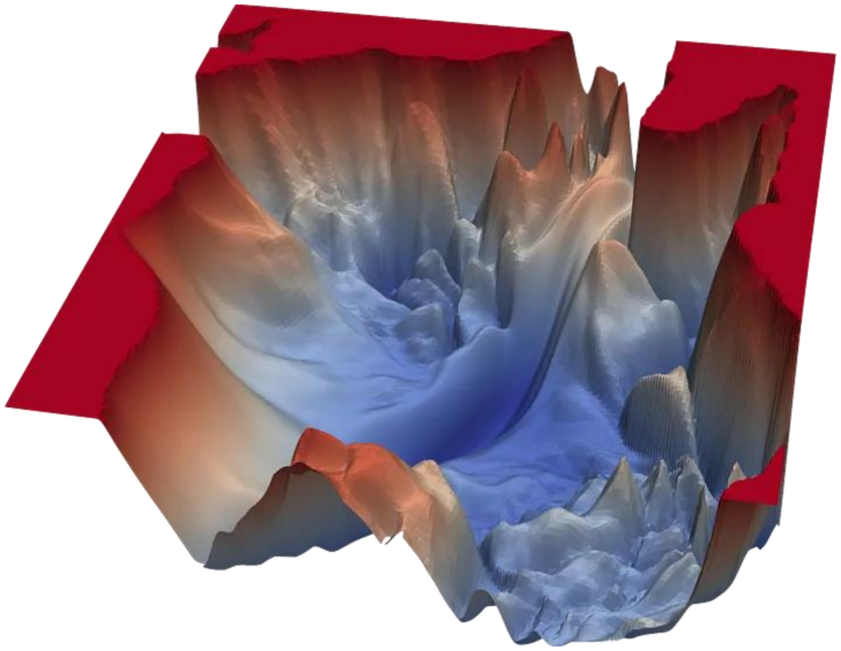


__Fig. 2__: Loss landscape of a convolutional neural network with 56 layers (VGG-56, source[1])

Loss Functions can be difficult to optimize

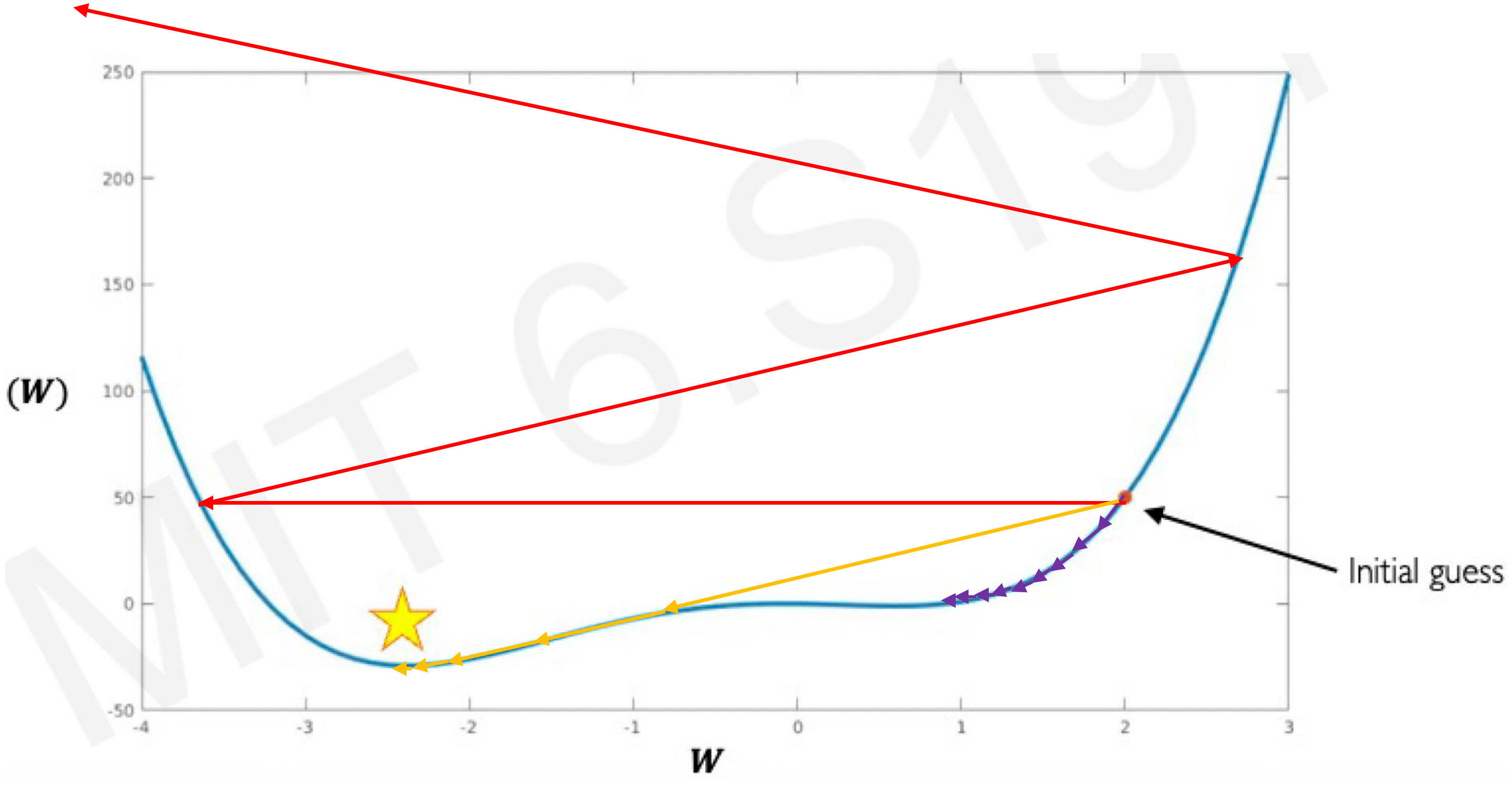
Remember:

Optimization through Gradient Descent:

$$\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$$

Setting the Learning Rate

- Small learning rate converges slowly and gets stuck in false local minima
- Large learning rates overshoot, become unstable and diverge
- Stable learning rates converge smoothly and avoid local minima



How to deal with this?

- Idea 1:
  - Hyperparameter tunning: try lots of different learning rates and see what works "just right"

ALWAYS do hyper tuning

- Idea 2:
  - Learning rate scheduler: fixed algorithms that change learning rate over epochs (typically, start at higher value and gradually lower it)

Can help, but interacts with other ideas

- Idea 3:
  - Sophisticated optimizers: techniques that adapt learning rate to current loss landscape

In practice, we do NOT use (pure) Gradient Descent

Idea 3: Sophisticated Optimizers

- Based on mini-batches (subsets of training data)
- Learning rates are no longer fixed
- Can be made larger or smaller depending on:
  - Whether previous gradients are going “in the same direction”
  - Whether gradients in certain directions are changing much or little

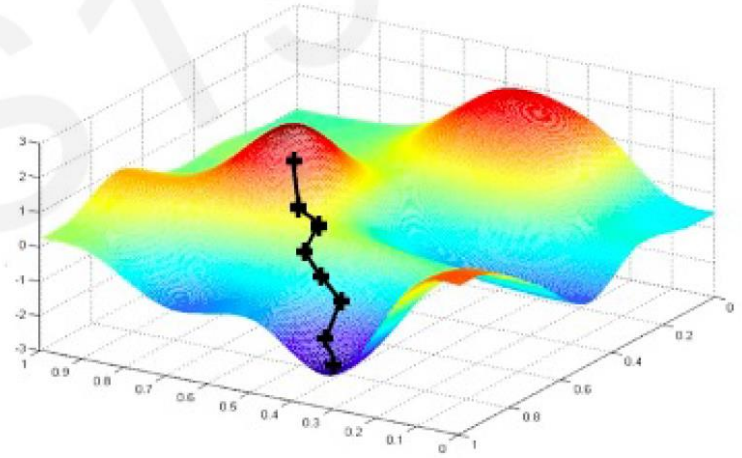
Gradient-based Optimizers

Algorithm	Pytorch implementation	Reference
• Stochastic Gradient Descent (SGD)	`torch.optim.SGD`	Kiefer & Wolfowitz. “Stochastic Estimation of the Maximum of a Regression Function.” 1952.
• Adagrad	`torch.optim.Adagrad`	Methods for Online Learning and Stochastic Optimization.” 2011.
• RMSProp	`torch.optim.RMSprop`	Hinton. Lecture 6e. Coursera.
• Adam	`torch.optim.Adam`	Duchi et al. “Adaptive Subgradient Kingma et al. “Adam: A Method for Stochastic Optimization.” 2014.
• AdamW	`torch.optim.AdamW`	

Gradient Descent

Algorithm

```
1. Initialize weights randomly $\sim\mathcal{N}(0,\sigma^2)$
2. Loop until convergence:
3.     Compute gradient, $\frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
4.     Update weights, $\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
5. Return weights
```

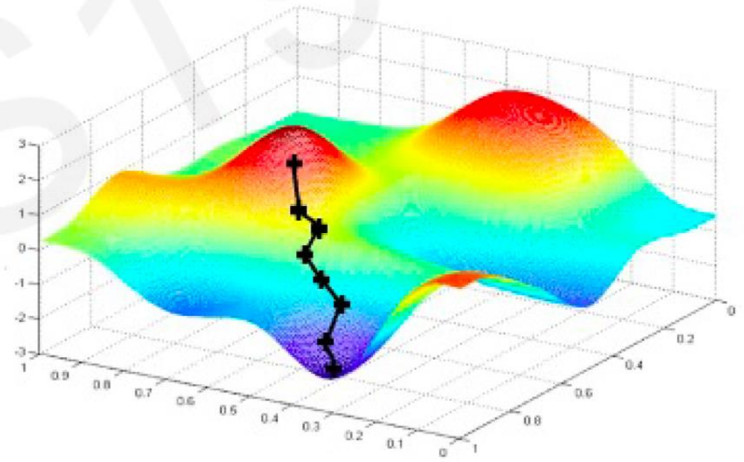


Computes loss gradients over **entire training data**; can be very **computationally intensive**!

Stochastic Gradient Descent

Algorithm

1. Initialize weights randomly $\sim\mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Pick single data point $\boldsymbol{i}$
4. Compute gradient, $\frac{\partial J_i(\boldsymbol{W})}{\partial \boldsymbol{W}}$
5. Update weights, $\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
6. Return weights

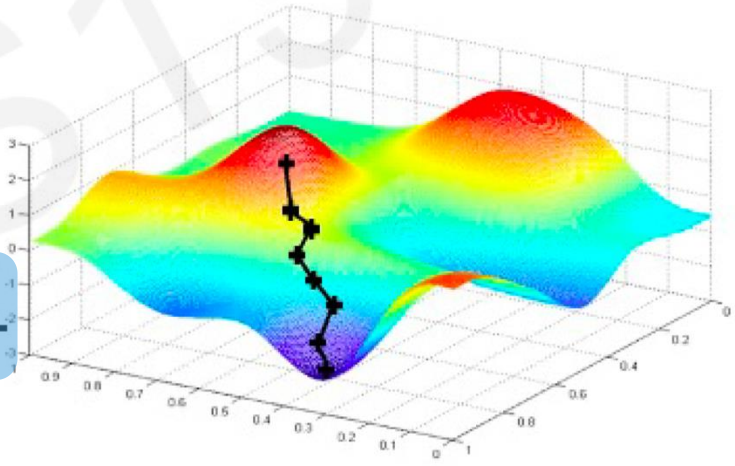


Easy to compute but
very noisy (stochastic)!

(Mini-batch) Stochastic Gradient Descent

Algorithm

```
1. Initialize weights randomly $\sim\mathcal{N}(0,\sigma^2)$
2. Loop until convergence:
3.     Pick batch of $B$ data points
4.     Compute gradient, $\frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}} = \frac{1}{B}\sum_{k=1}^{B}\frac{\partial J_k(\boldsymbol{W})}{\partial \boldsymbol{W}}$
5.     Update weights, $\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta\frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
6. Return weights
```



Fast to compute and a much better estimate of true gradient!

Mini-batches while training

- In comparison to pure SGD (i.e., B=1):
  - Mini-batch SGD **estimates gradient more accurately**
    - Smoother convergence
    - Allows for larger learning rates
  - Mini-batch SGD leads to faster training
    - Parallel computation achieves significant speedups on GPUs (Faster to compute gradient for mini-batch of size B than for B observations sequentially)
    - Model is likely to improve after each iteration (= processing each batch)

How to pick mini-batch size B?

1. Start from "small" initial value B
2. Train model for a couple of iterations
3. If program doesn't crash for running out of memory then DOUBLE the mini-batch size B
4. Repeat until program crashes
5. Go back to previous value of B (that worked)

"Small" is a value that doesn't crash your program; depends on model size and memory

Important: A good rule of thumb is to increase learning rate η proportionally to mini-batch size (so if you tuned η before tuning B, you will need to change η again for best results).

Stochastic gradient descent

- ~~Despite~~ Thanks (!) to the “noise” (statistical inaccuracy) in the mini-batch gradient estimates, we often converge to good parameterizations.

- Reaching a certain loss value can be much faster than regular gradient descent because we adjust the weights *many times* per epoch.

Putting it all together

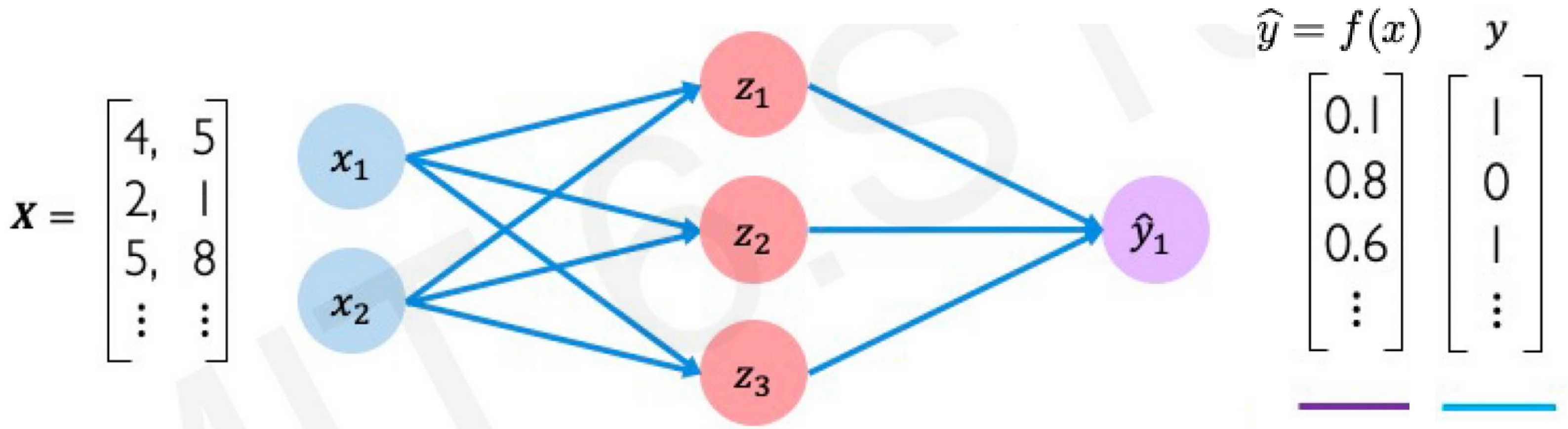
```python
import torch
import torch.nn as nn

X, y = ... # Load data

# Define the model using torch.Sequential
model = nn.Sequential(...)

# Initialize the loss function and optimizer
loss_fn = nn.MSELoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)

# Run SGD in an infinite loop
while True:
    # Shuffle the data at the start of each epoch
    permutation = torch.randperm(X.size(0))
    X = X[permutation]
    y = y[permutation]

    for i in range(0, len(X), 10):  # iterate over mini-batches
        batch_X = X[i:i+10]
        batch_y = y[i:i+10]

        # Forward pass
        predictions = model(batch_X)
        loss = loss_fn(predictions, batch_y)

        # Backward pass
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

Can be replaced with any pytorch optimizer!

Pytorch implementation

Binary Cross Entropy Loss

- **Cross entropy loss** can be used with models that output a probability between 0 and 1

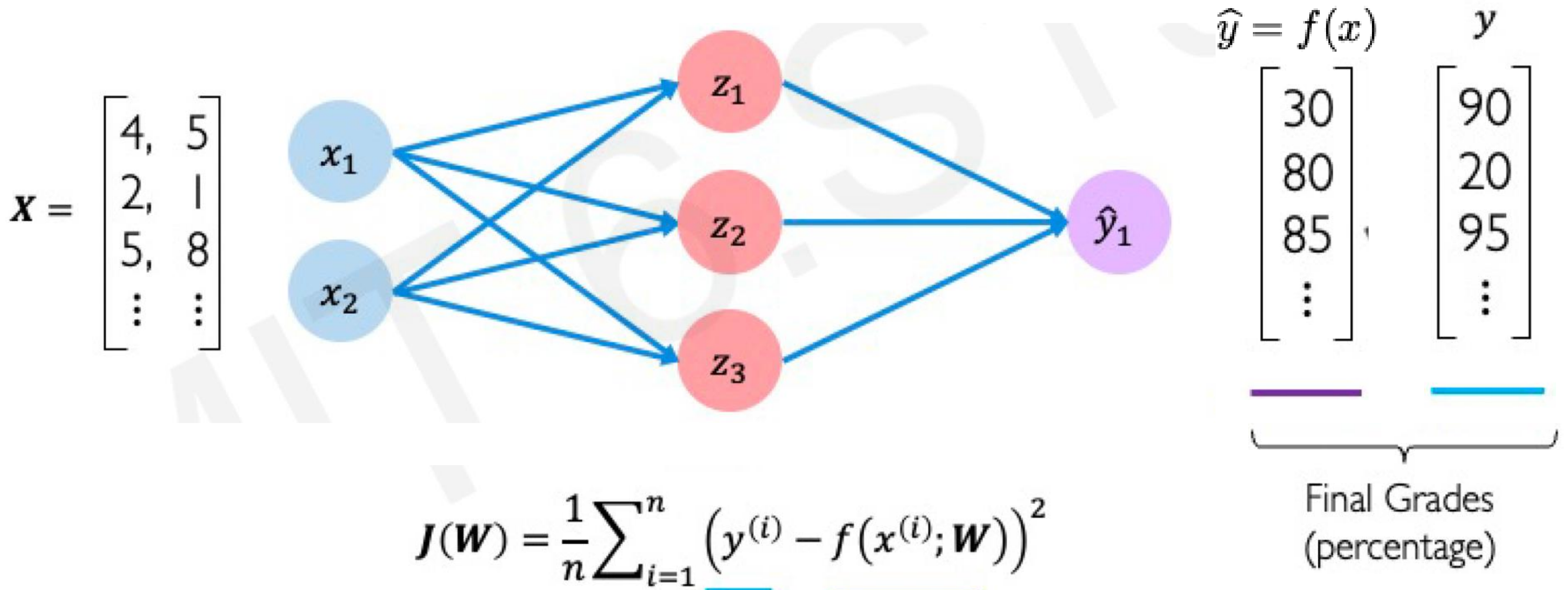

$X = \begin{bmatrix} 4, & 5 \\ 2, & 1 \\ 5, & 8 \\ \vdots & \vdots \end{bmatrix}$
x_1
x_2
z_1
z_2
z_3
$\hat{y}_1$
$\hat{y} = f(x)$
y
$\begin{bmatrix} 0.1 \\ 0.8 \\ 0.6 \\ \vdots \end{bmatrix}$
$\begin{bmatrix} 1 \\ 0 \\ 1 \\ \vdots \end{bmatrix}$

$$J(\boldsymbol{W}) = -\frac{1}{n}\sum_{i=1}^{n} \underbrace{y^{(i)}}_{\text{Actual}} \log\Big(\underbrace{f\big(x^{(i)};\boldsymbol{W}\big)}_{\text{Predicted}}\Big) + (1-\underbrace{y^{(i)}}_{\text{Actual}})\log\Big(1-\underbrace{f\big(x^{(i)};\boldsymbol{W}\big)}_{\text{Predicted}}\Big)$$

```
loss = nn.CrossEntropyLoss(reduce='mean')
output = loss(predicted, y)
```

Mean Squared Error Loss

- **Mean squared error (MSE) loss** can be used with models that output continuous real numbers

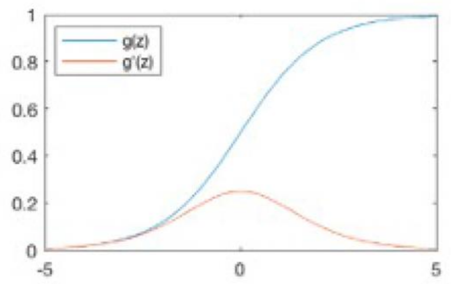

$X = \begin{bmatrix} 4, & 5 \\ 2, & 1 \\ 5, & 8 \\ \vdots & \vdots \end{bmatrix}$
x_1
x_2
z_1
z_2
z_3
$\hat{y}_1$
$\hat{y} = f(x)$
y
$\begin{bmatrix} 30 \\ 80 \\ 85 \\ \vdots \end{bmatrix}$
$\begin{bmatrix} 90 \\ 20 \\ 95 \\ \vdots \end{bmatrix}$
Final Grades (percentage)

$$J(\boldsymbol{W}) = \frac{1}{n}\sum_{i=1}^{n}\Big(\underbrace{y^{(i)}}_{\text{Actual}} - \underbrace{f\big(x^{(i)};\boldsymbol{W}\big)}_{\text{Predicted}}\Big)^2$$

```
loss = nn.MSEloss(reduce='mean')
output = loss(predicted, y)

# loss = torch.mean(torch.square(y-predicted))
```

Common Activation Functions

Sigmoid Function

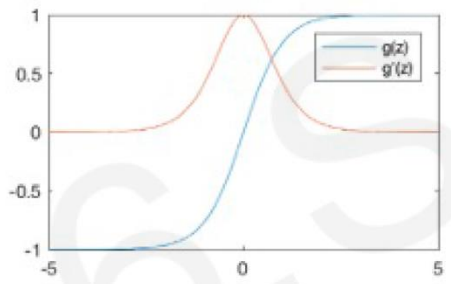


$$g(z) = \frac{1}{1 + e^{-z}}$$

$$g'(z) = g(z)(1 - g(z))$$

```
torch.sigmoid(z)
```

Hyperbolic Tangent

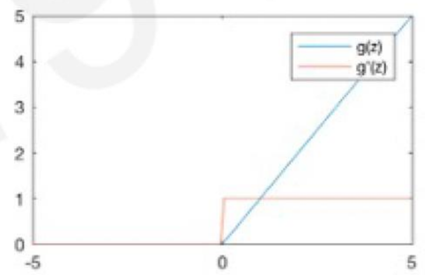


$$g(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$

$$g'(z) = 1 - g(z)^2$$

```
torch.tanh(z)
```

Rectified Linear Unit (ReLU)



$$g(z) = \max(0, z)$$

$$g'(z) = \begin{cases} 1, & z > 0 \\ 0, & \text{otherwise} \end{cases}$$

```
torch.relu(z)
```

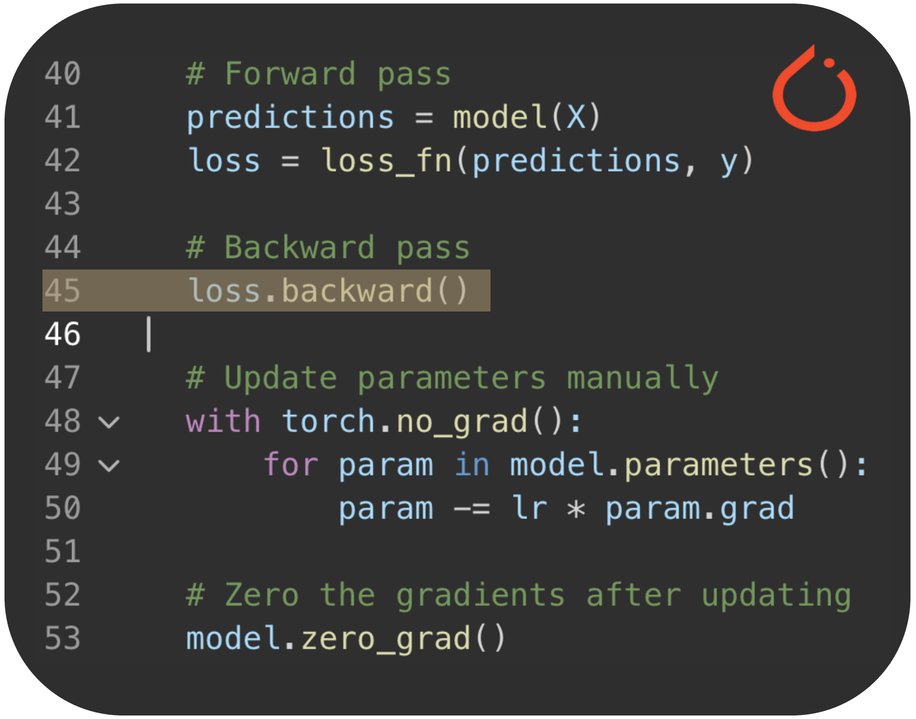
NOTE: All activation functions are non-linear

Pytorch code blocks

Gradient Descent

Algorithm

1. Initialize weights randomly $\sim\mathcal{N}(0,\sigma^2)$
2. Loop until convergence:
3. Compute gradient, $\frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
4. Update weights, $\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
5. Return weights



```python
    # Forward pass
    predictions = model(X)
    loss = loss_fn(predictions, y)

    # Backward pass
    loss.backward()

    # Update parameters manually
    with torch.no_grad():
        for param in model.parameters():
            param -= lr * param.grad

    # Zero the gradients after updating
    model.zero_grad()
```

Gradient-based Optimizers

Algorithm

- Stochastic Gradient Descent (SGD)
- Adagrad
- RMSProp
- Adam
- AdamW

Pytorch implementation

```
torch.optim.SGD
```

```
torch.optim.Adagrad
```

```
torch.optim.RMSprop
```

```
torch.optim.Adam
```

```
torch.optim.AdamW
```

Reference

Kiefer & Wolfowitz. “Stochastic Estimation of the Maximum of a Regression Function.” 1952.

Methods for Online Learning and Stochastic Optimization.” 2011.

Hinton. Lecture 6e. Coursera.

Duchi et al. “Adaptive Subgradient
Kingma et al. “Adam: A Method for Stochastic Optimization.” 2014.