

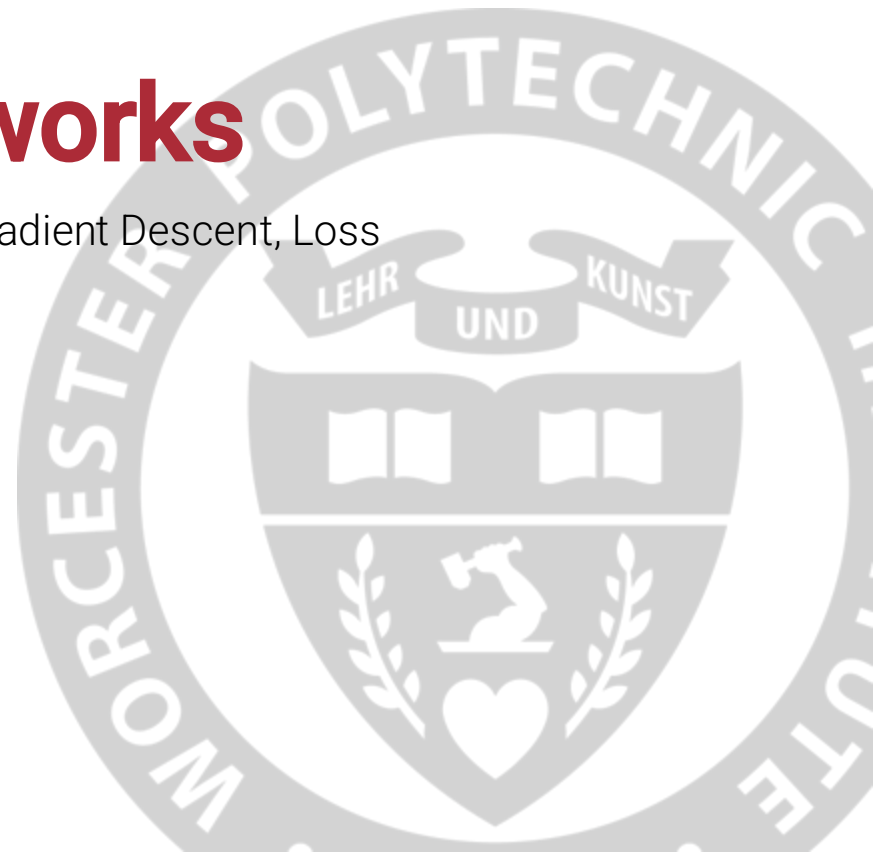
LECTURE 03

Training Neural Networks

Basics of Training, Gradient Descent, Stochastic Gradient Descent, Loss Surfaces, Convergence Issues

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

Fabricio Murai

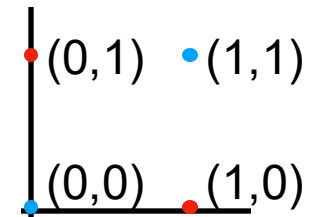


Lec 02 Recap: Feed-forward Neural Nets, Universal Approximation Theorem

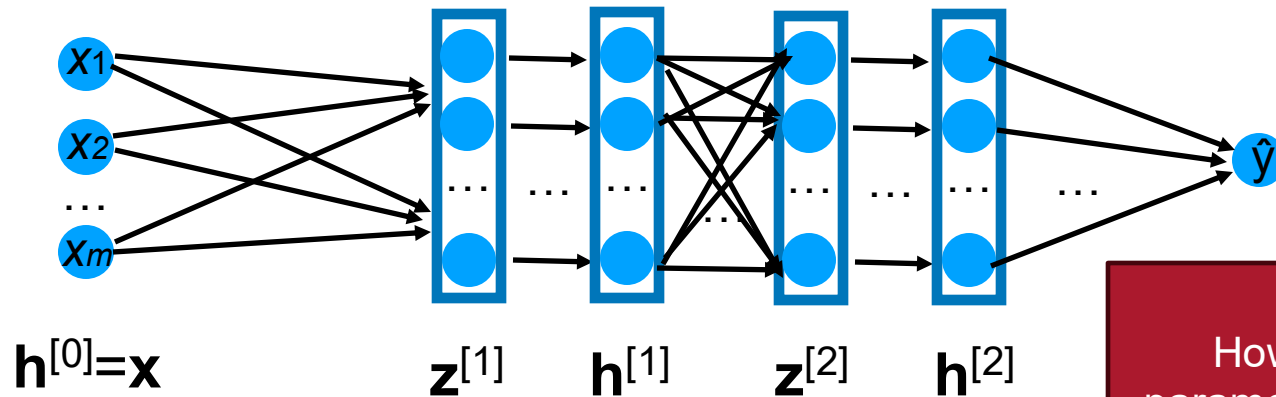
- Perceptron can only solve linearly separable problems

- Remember the XOR problem

- We need hidden layers + non-linear activation functions



- Multi-layer Perceptron (MLP)



How many parameters in this network?

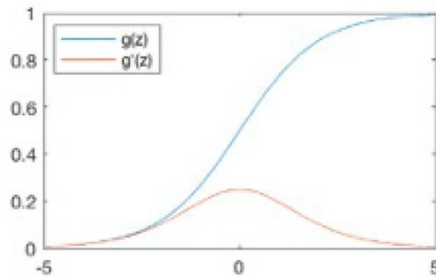
pre-activation values: $z^{[1]} = W^{[1]} x + b^{[1]}$; $z^{[\ell]} = W^{[\ell]} h^{[\ell-1]} + b^{[\ell]}$

activation values: $h^{[1]} = g(z^{[1]})$; $h^{[\ell]} = g(z^{[\ell]})$

Lec 02 Recap: Feed-forward Neural Nets, Universal Approximation Theorem

- Common Activation Functions

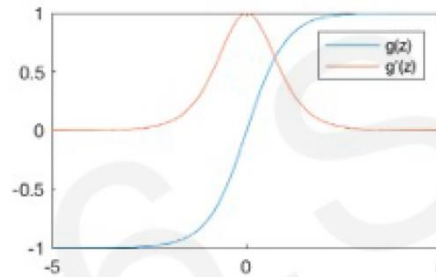
Sigmoid Function



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

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

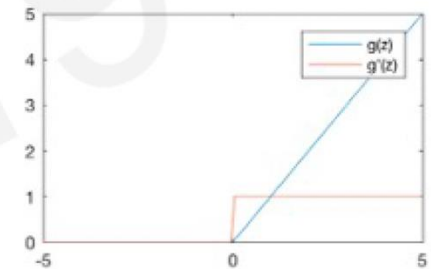
Hyperbolic Tangent



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

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

Rectified Linear Unit (ReLU)

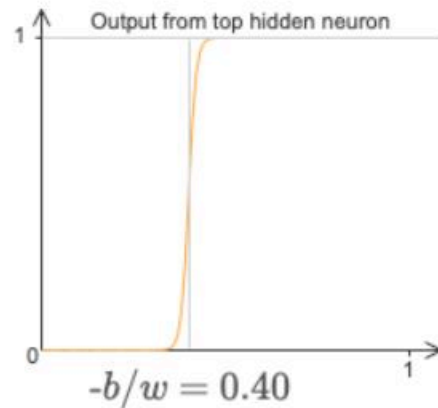
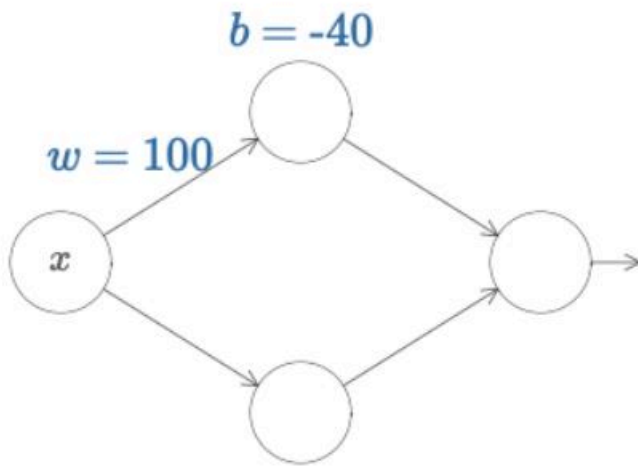


$$g(z) = \max(0, z)$$

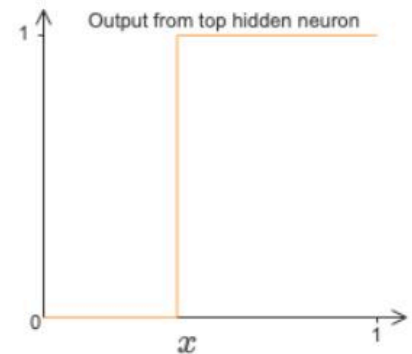
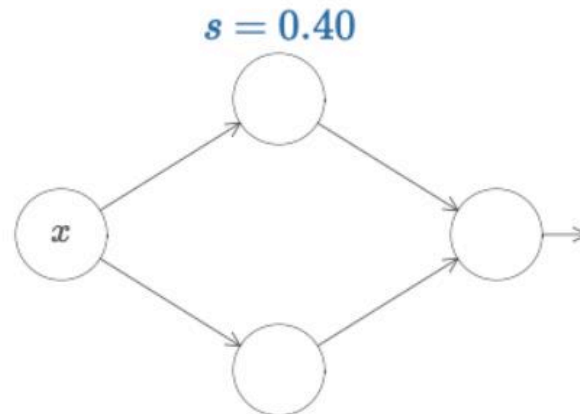
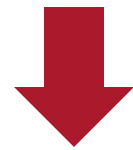
$$g'(z) = \begin{cases} 1, & z > 0 \\ 0, & \text{otherwise} \end{cases}$$

Why not to use a
step function?

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



Approximating by
step function +
defining $s = -b/w$



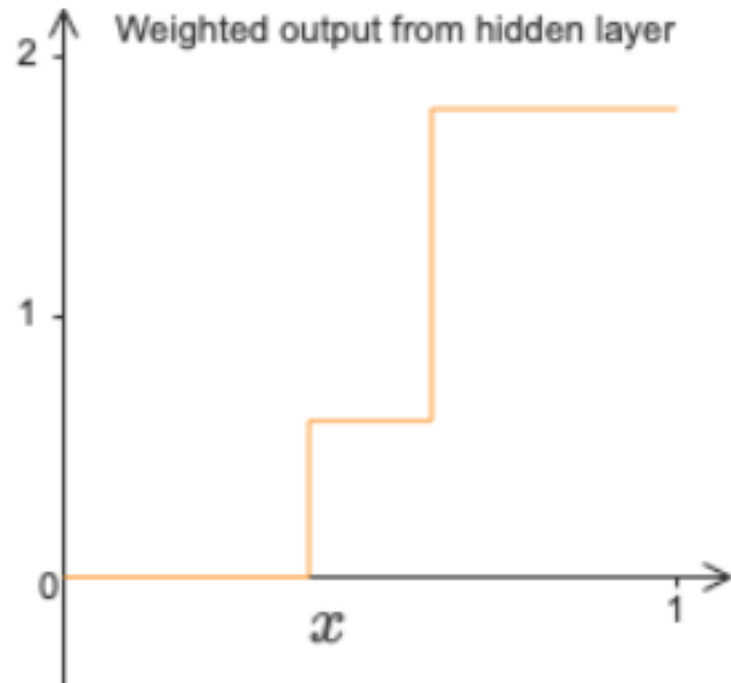
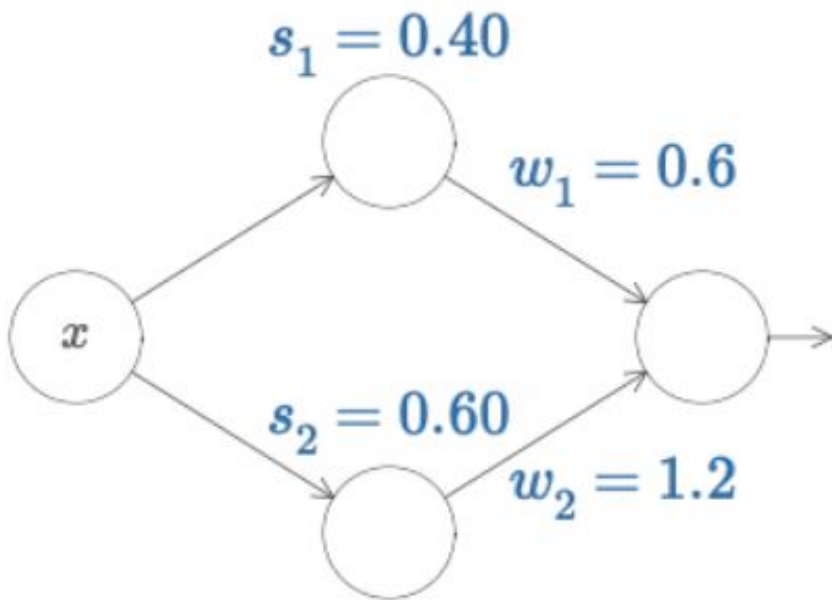
Neural Nets are Universal Function Approximators

Proof idea

- Using pairs of sigmoid functions, we can construct pulse functions that represent vertical “bars”.
- Using enough vertical bars, we can approximate any f (akin to the trapezoidal rule of calculus).
- Next slides are based on Michael Nielsen’s visual proof sketch [here](#).

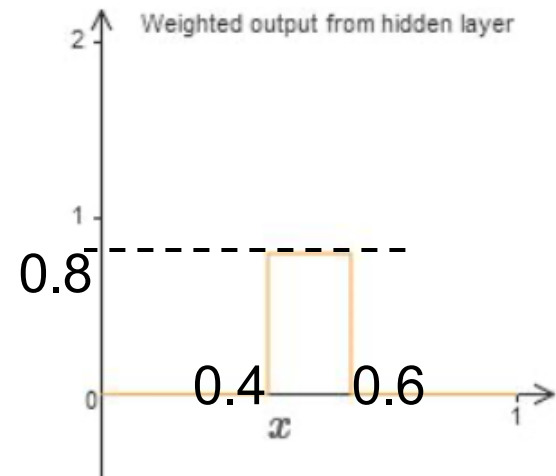
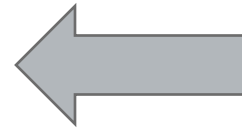
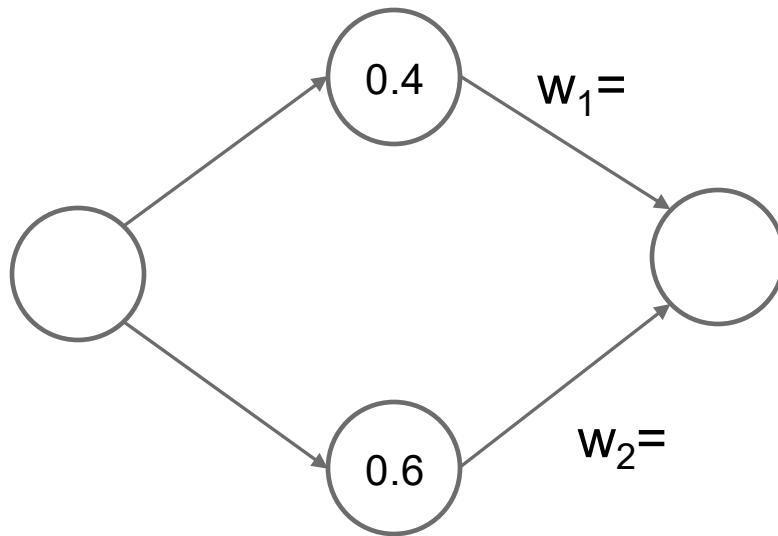
What functions can we represent with 1 hidden layer and a sigmoid?

- Try to draw the output function for this MLP

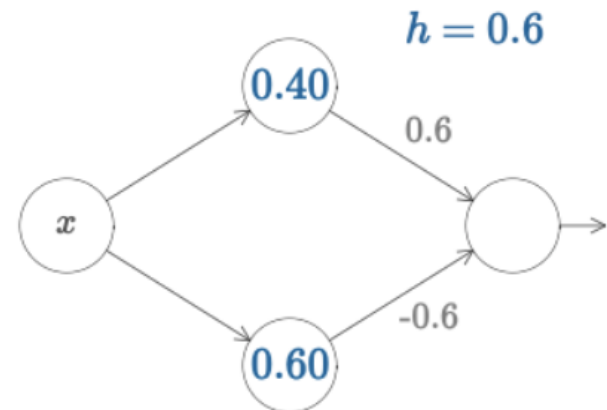
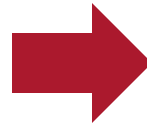


What functions can we represent with 1 hidden layer and a sigmoid?

- How to create a pulse?

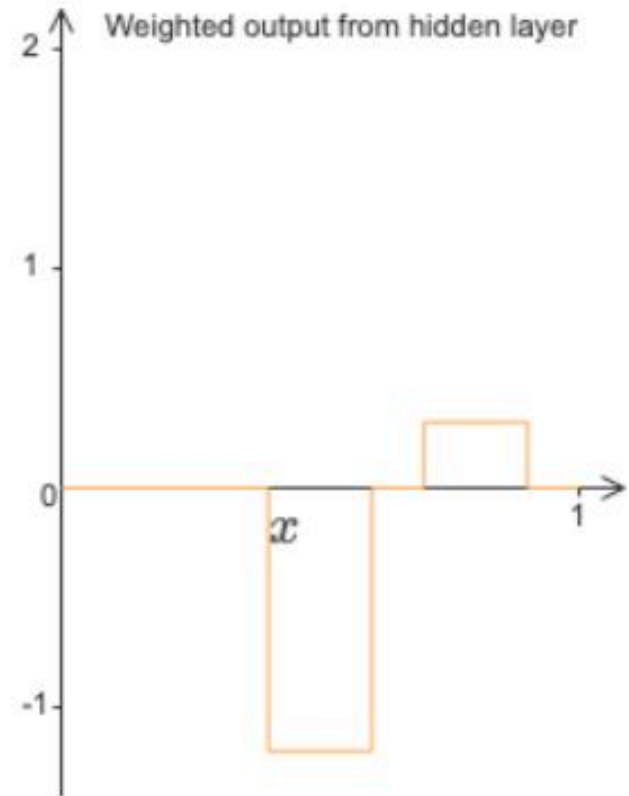
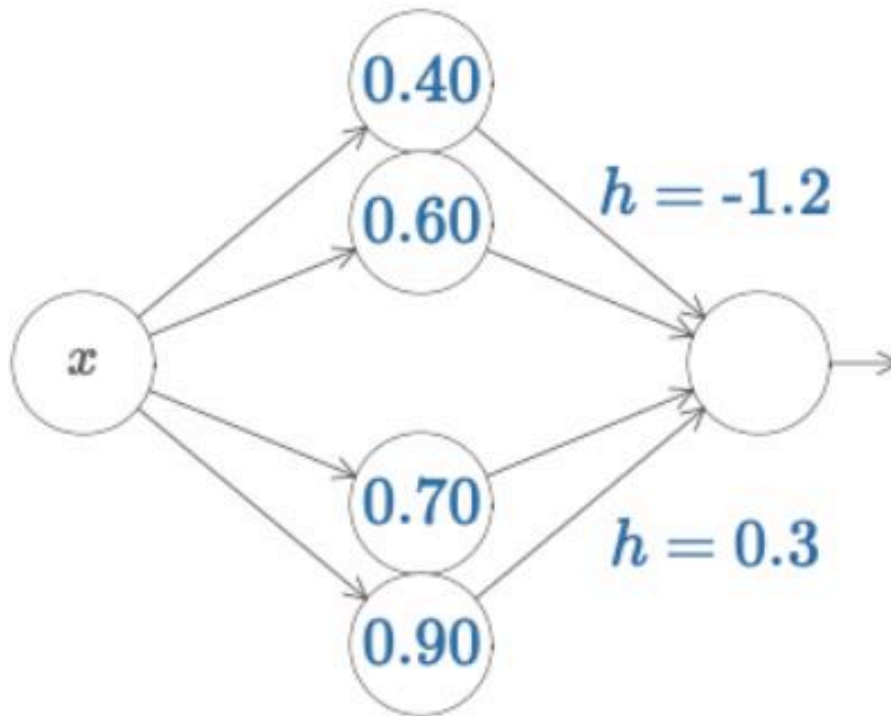


Using a single parameter h to represent w_1 & w_2



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

- We can add pairs of nodes to the hidden layer to create pulses



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

- We can combine pulses to describe any closed, bounded, continuous function

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:
$$|f(x) - \hat{f}(x)| < \epsilon \quad \forall x \quad L^1 \text{ distance}$$

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*.

Universal function approximation theorem

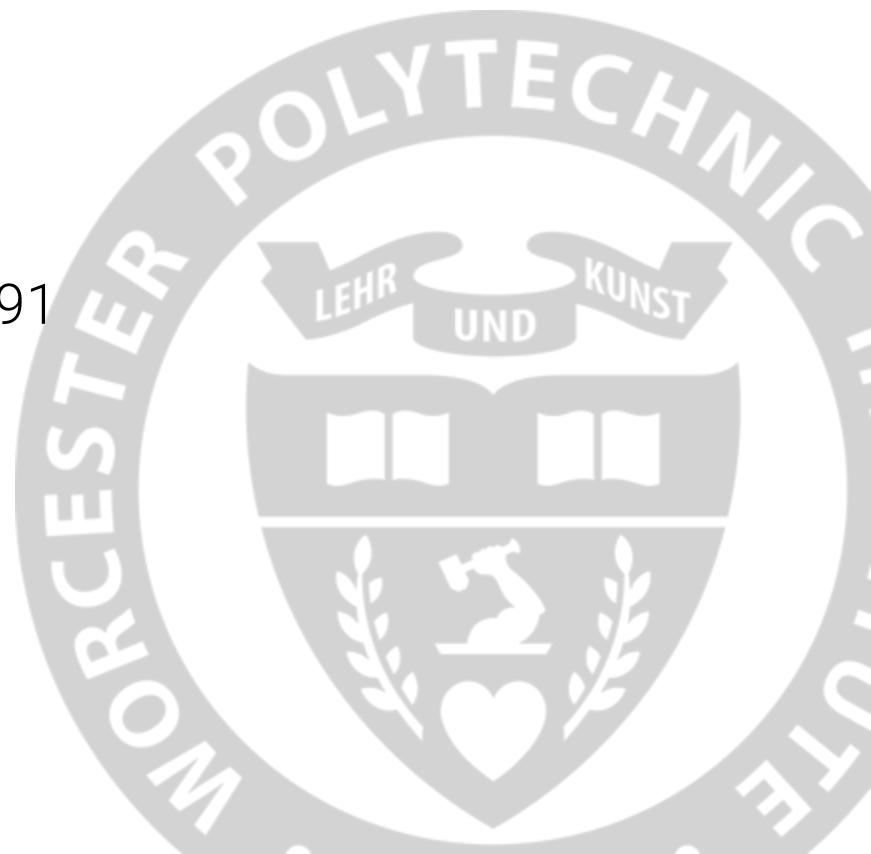
- **Previous versions** of theorem consider bounded depth (1 hidden layer) and arbitrary width (#neurons in hidden)
- **Dual versions** of the theorem: consider networks of bounded width and arbitrary depth.
- (Lu et al. 2017) Networks of width $n+4$ with ReLu activation functions can approximate any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

Story so far

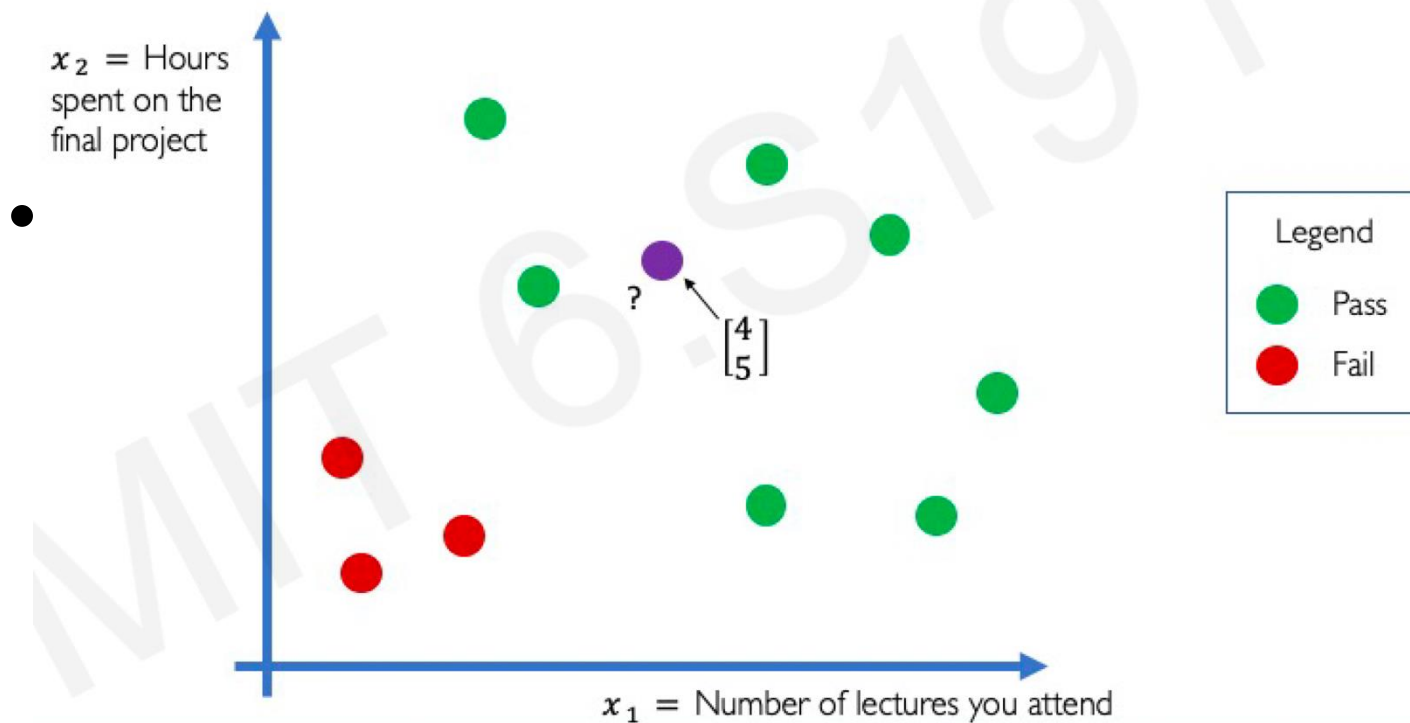
- Neural networks are universal approximators
 - Can model any odd thing
 - Provided they have the right architecture
- We must train them to approximate any function
 - Specify the architecture
 - Learn their weights and biases
- Just like other ML models, neural nets are trained to minimize total “loss” on a training set
 - We train neural nets based on the gradient of loss with respect to each parameter

Training Basics

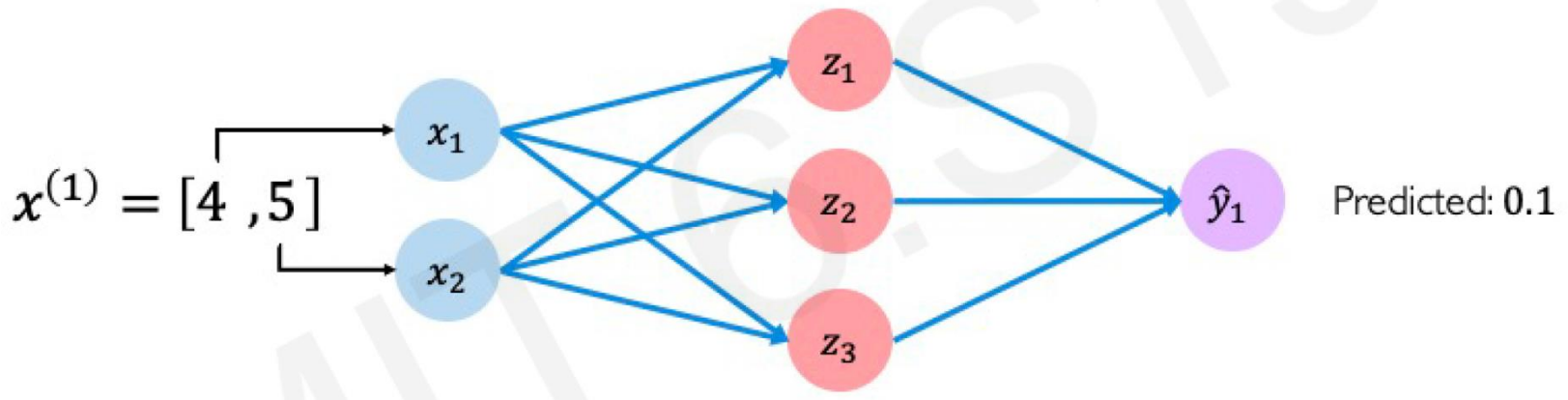
Credits: Based on slides from MIT 6.S191



Example Problem

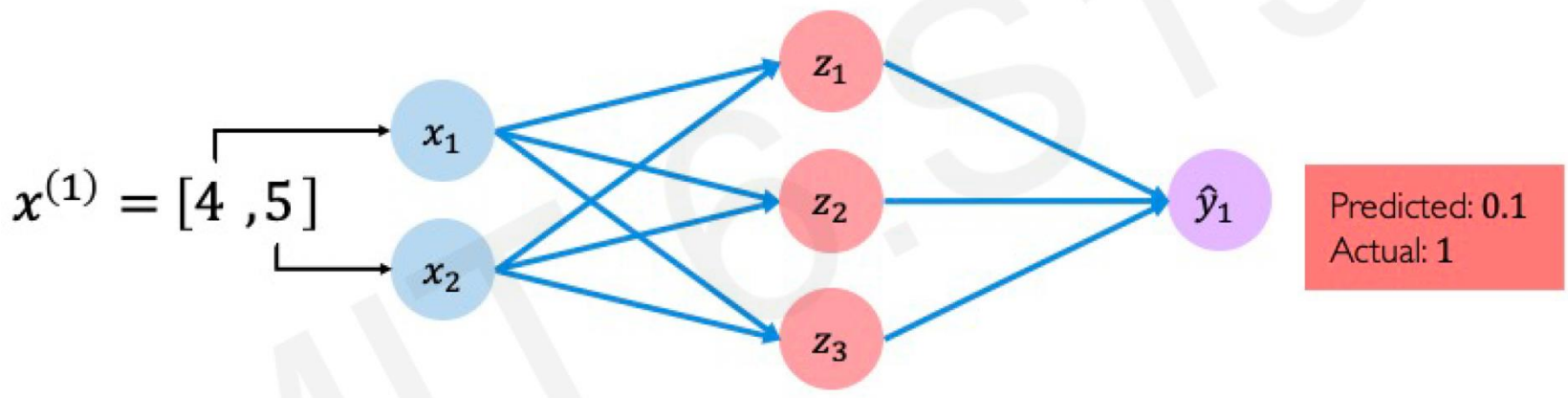


Example problem: Will I pass this class?

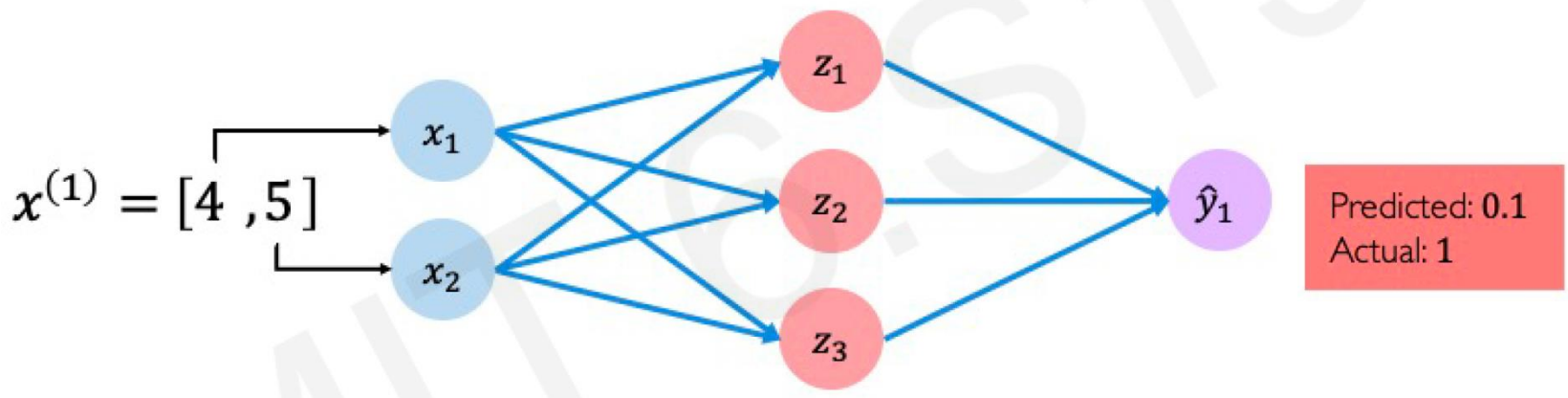


Ingredient #1
(Model)

Example problem: Will I pass this class?



Loss Function

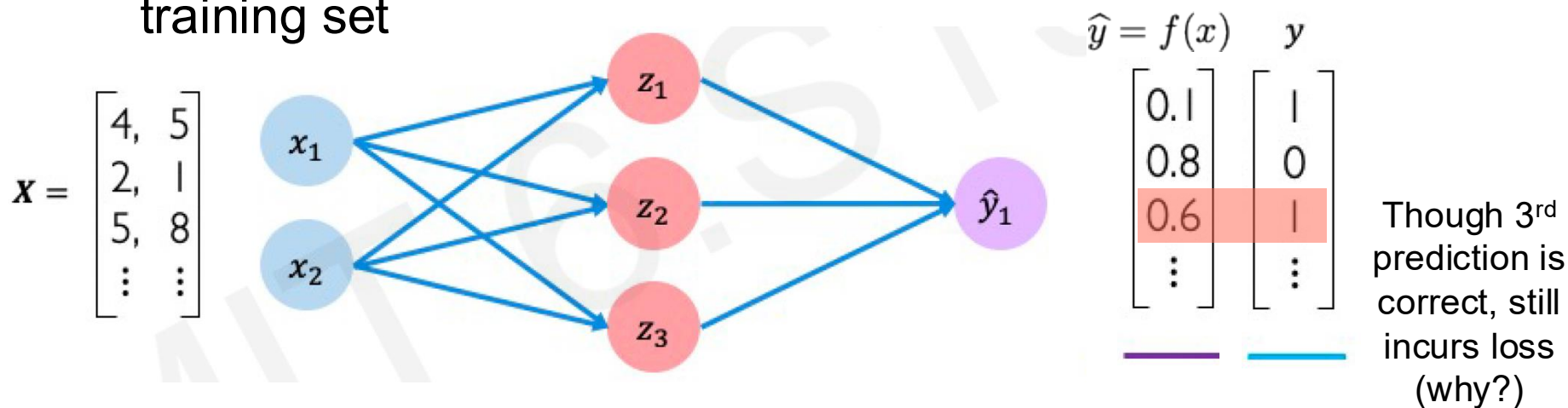


Ingredient #2

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

Cost Function

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



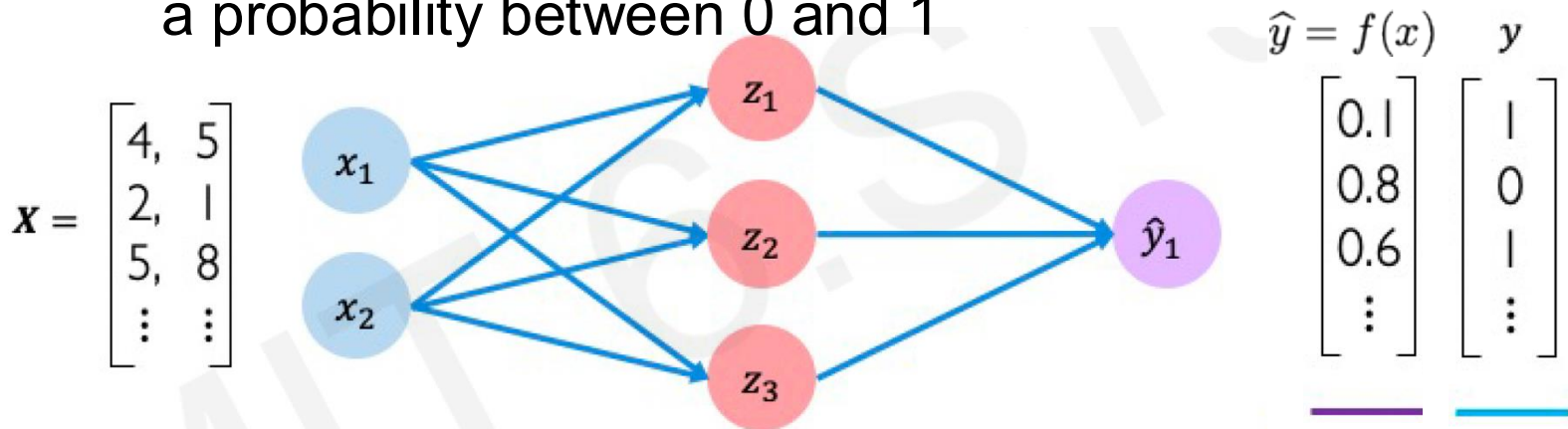
Also known as:

- Objective function
- Empirical loss
- Empirical risk

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

Binary Cross Entropy Loss

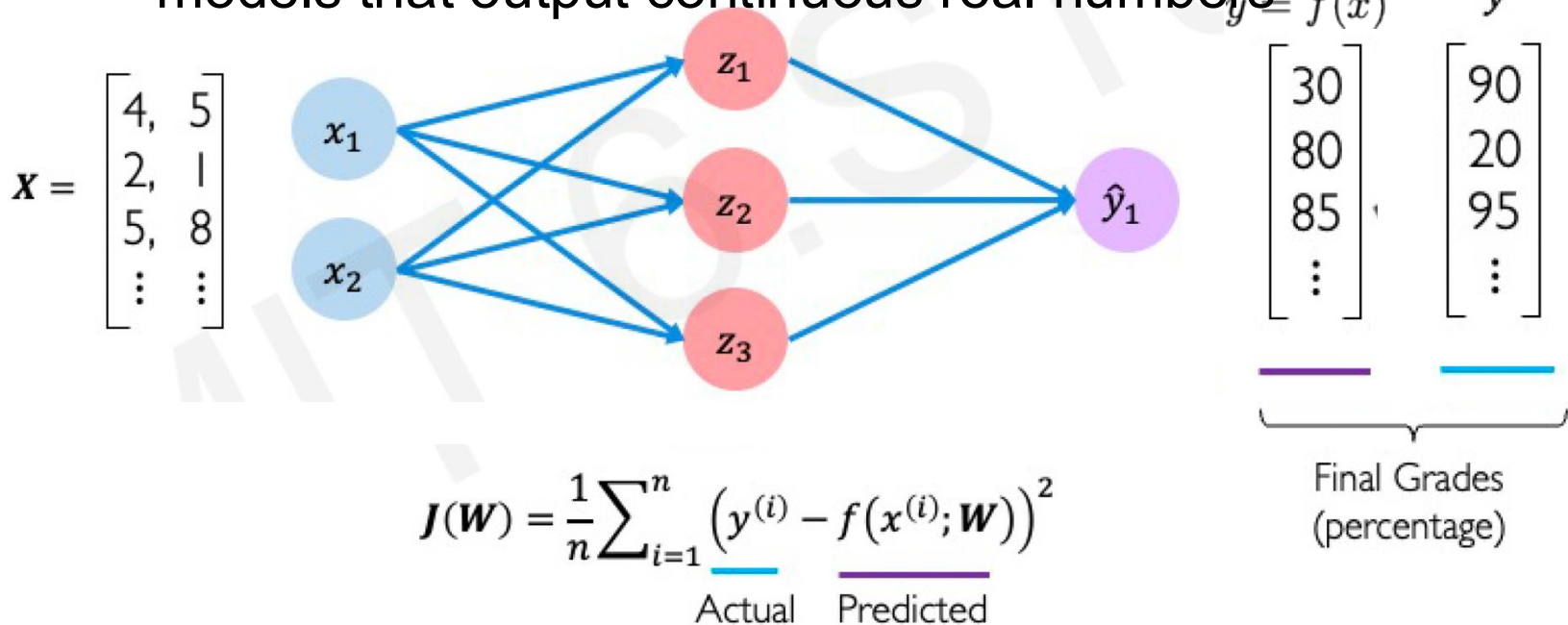
- **Cross entropy loss** can be used with models that output a probability between 0 and 1



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

Mean Squared Error Loss

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



Loss Optimization

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

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

$$\mathbf{W}^* = \operatorname{argmin}_{\mathbf{W}} J(\mathbf{W})$$

Remember:

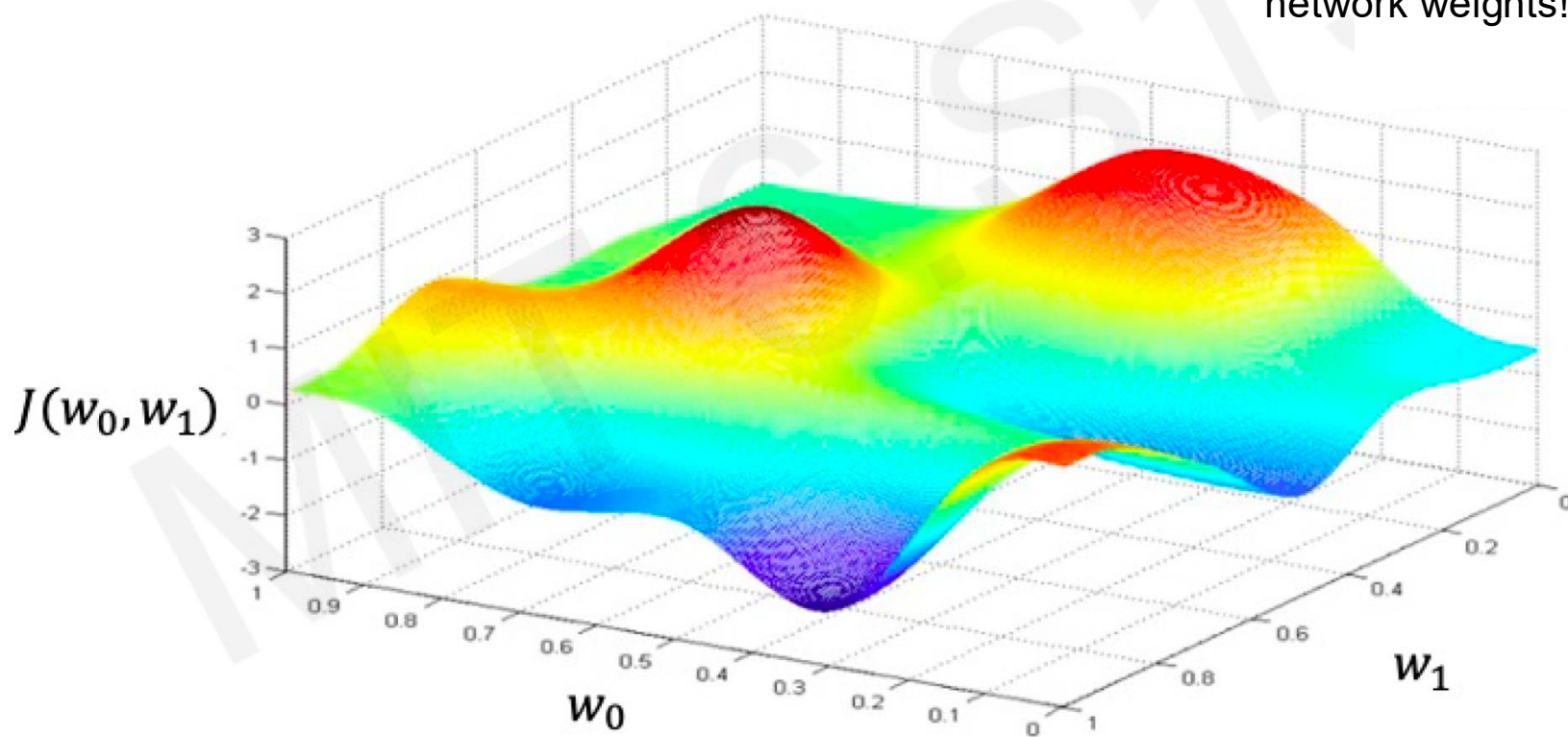
$$\mathbf{W} = \{ \mathbf{W}^{[1]}, \mathbf{W}^{[1]}, \dots, \mathbf{W}^{[L]} \}$$

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

Loss Optimization

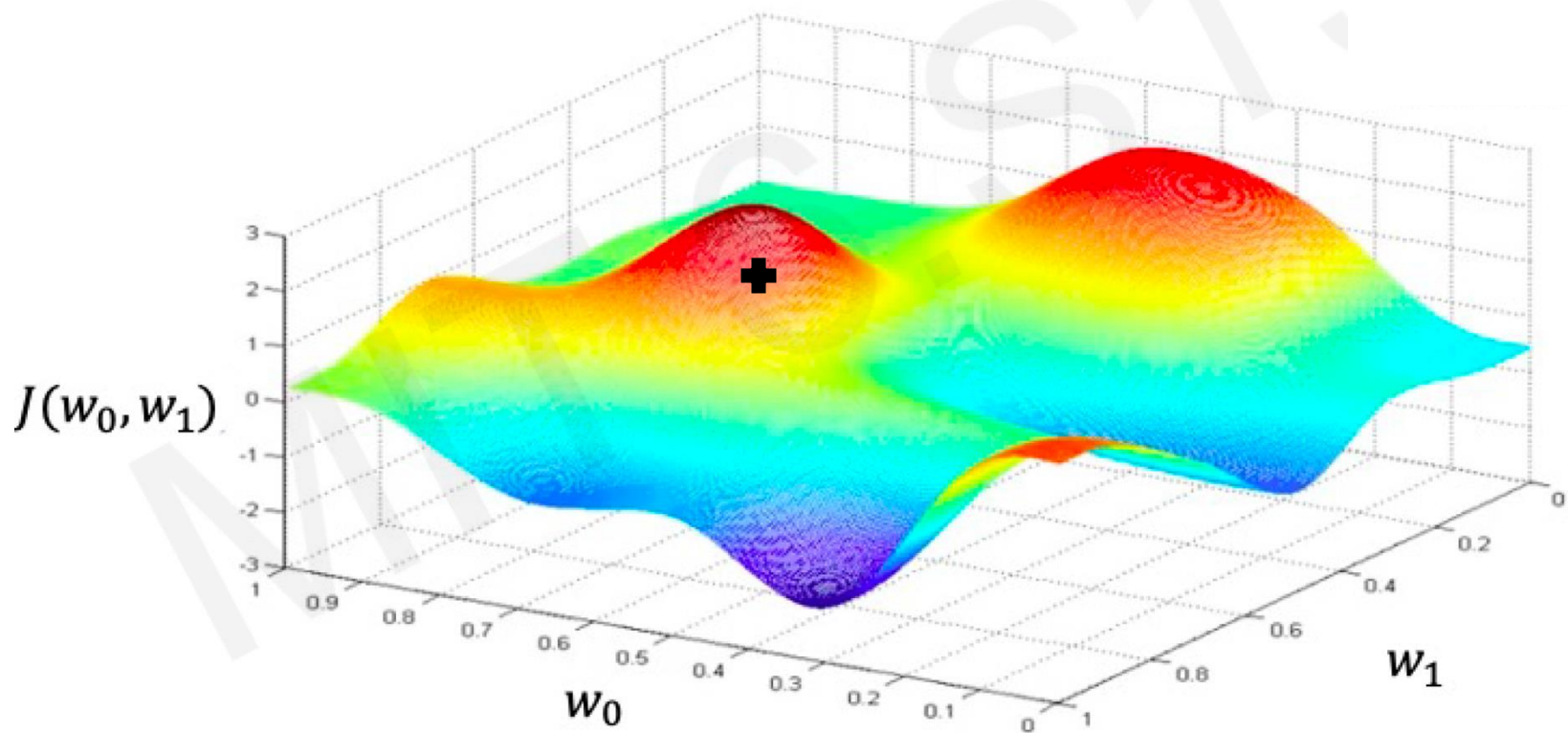
$$W^* = \underset{W}{\operatorname{argmin}} J(W)$$

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



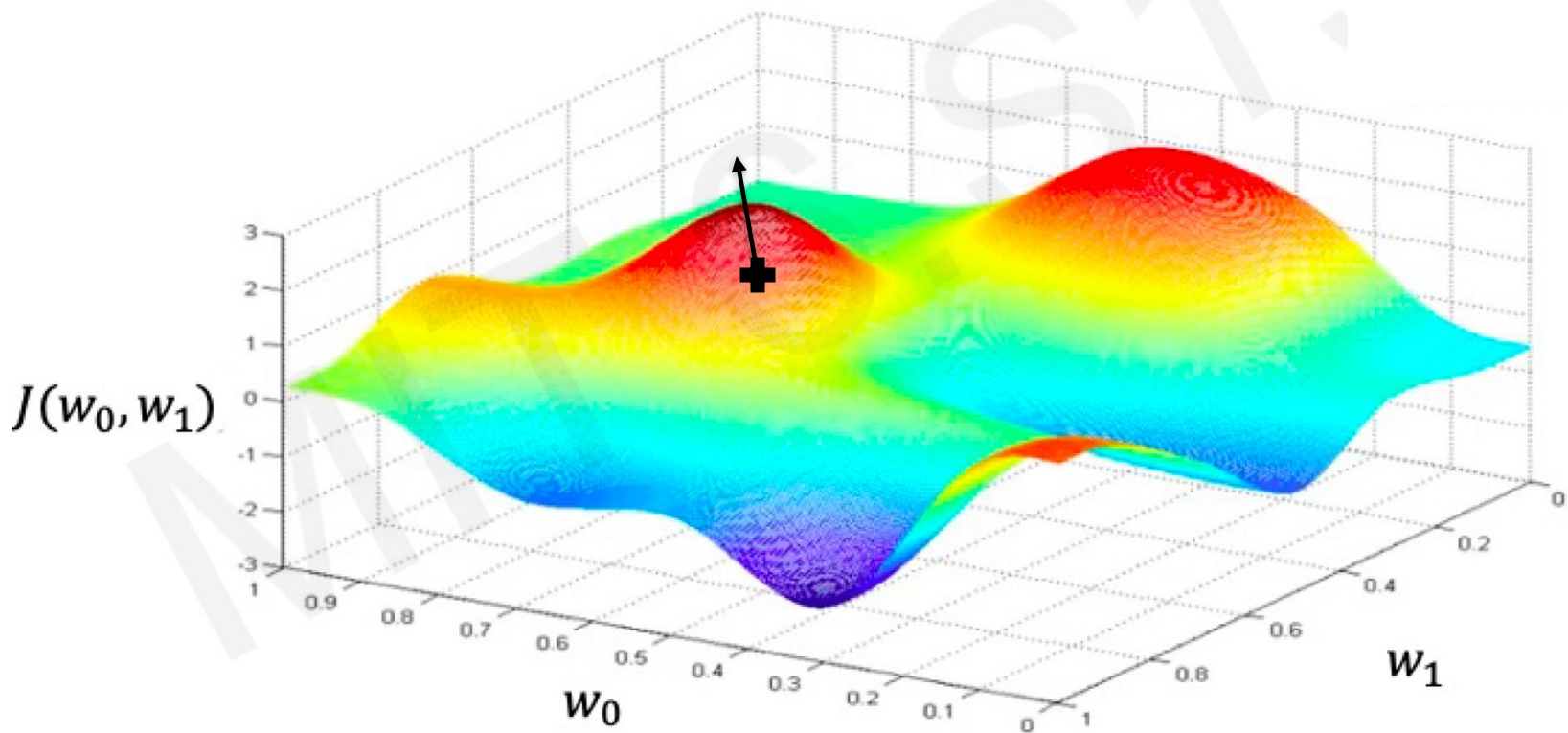
Loss Optimization

- Randomly pick an initial (w_0, w_1)



Loss Optimization

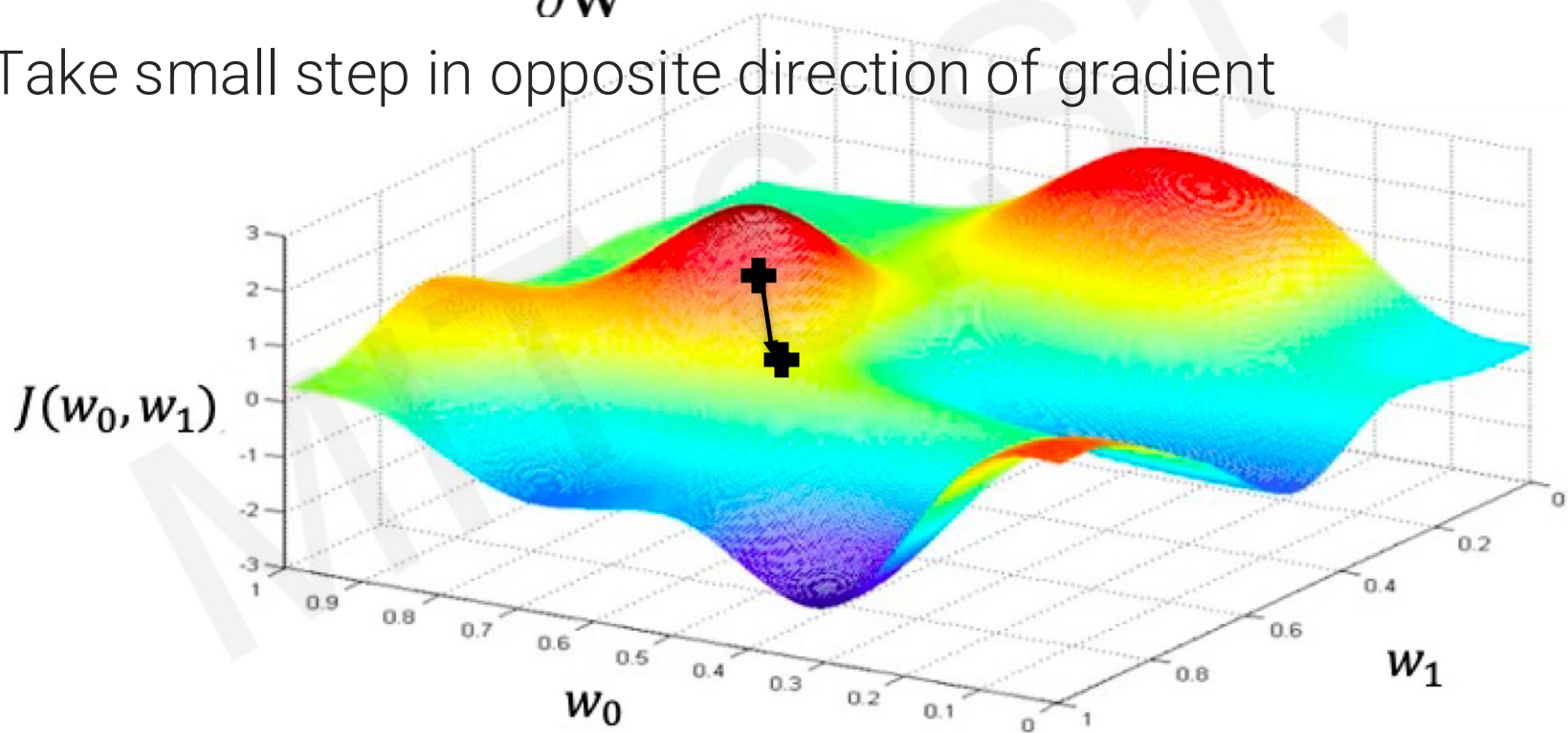
- Randomly pick an initial (w_0, w_1)
- Compute gradient* $\frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$



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

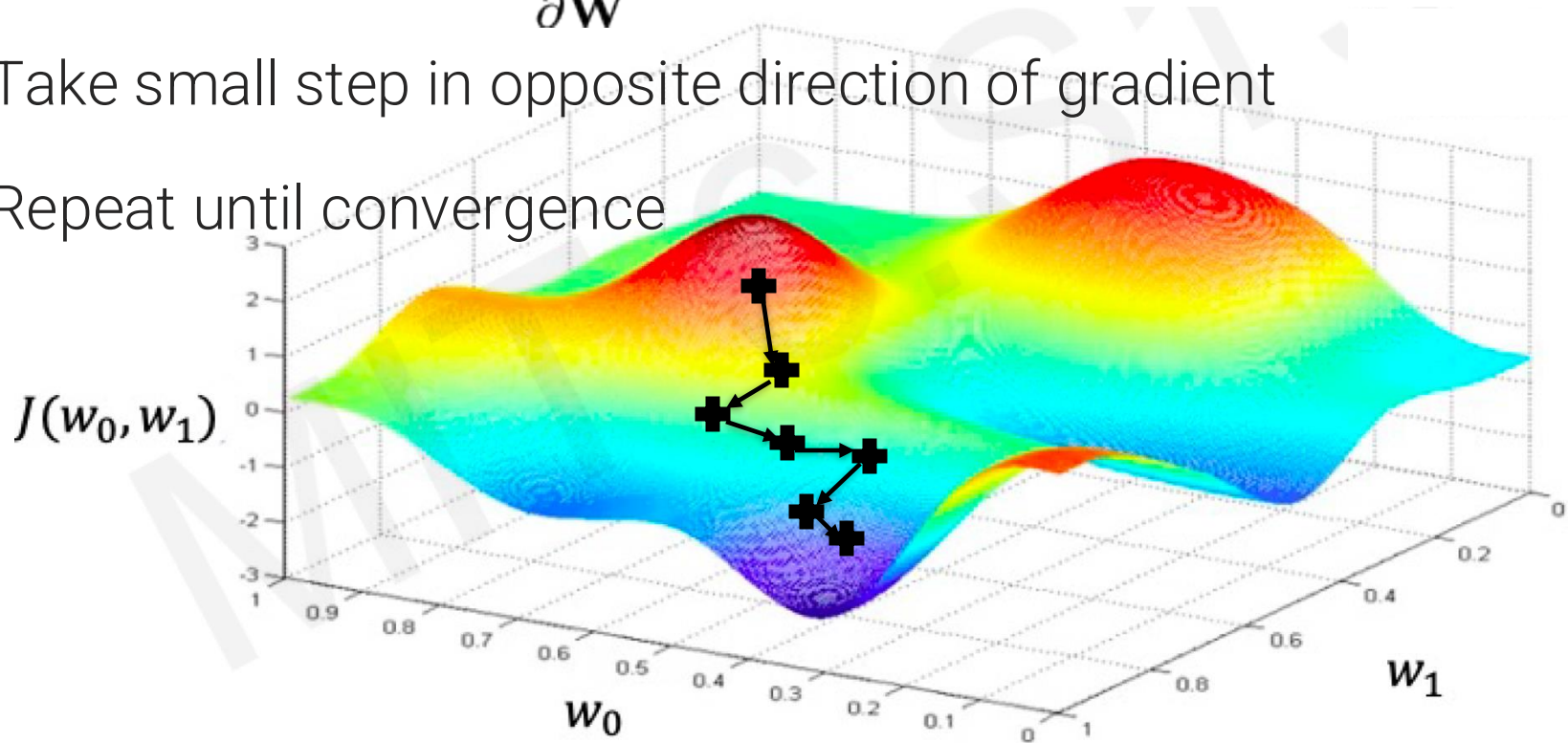
Loss Optimization

- 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



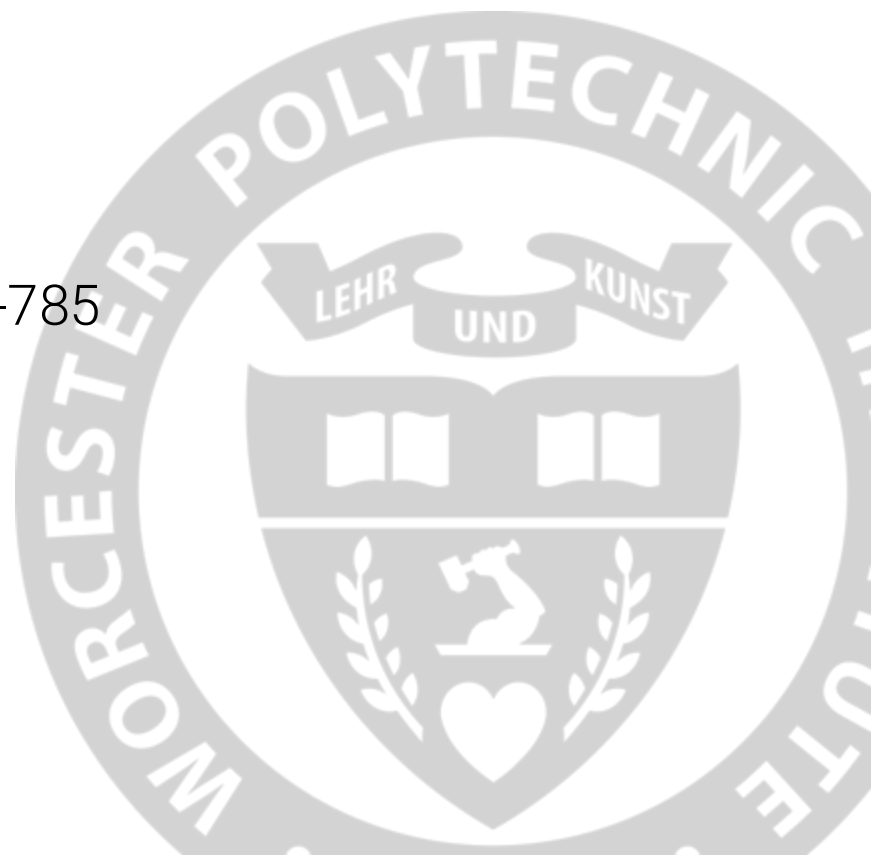
Gradient Descent

Algorithm

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

Convergence issues

Credits: some slides based on CMU 11-785

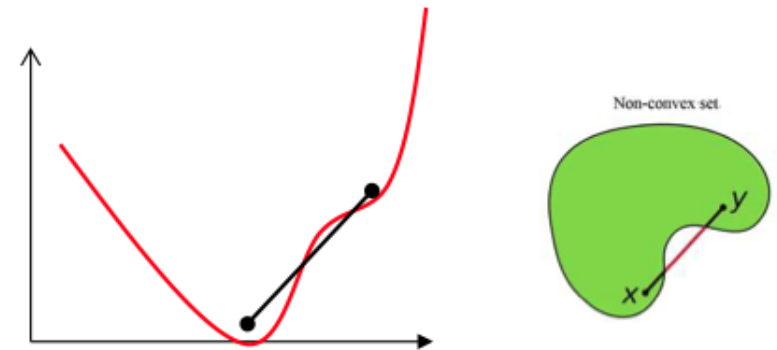
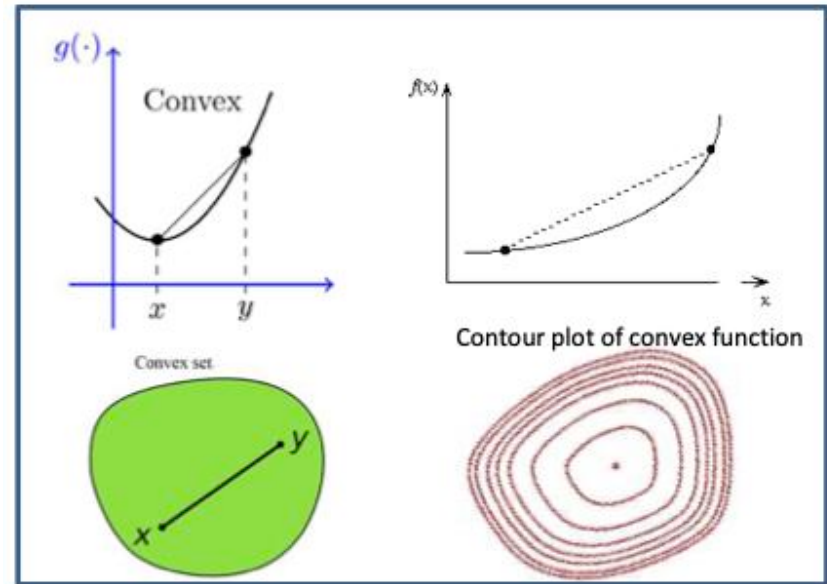


Loss Functions can be difficult to optimize

- Optimization via Gradient Descent:

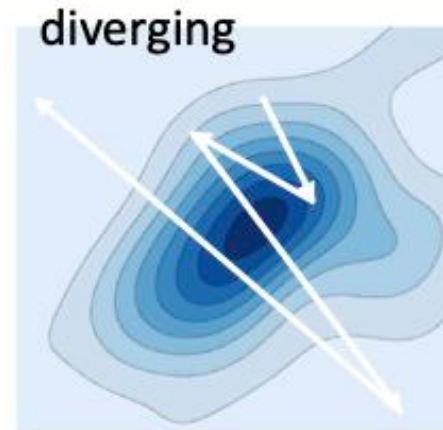
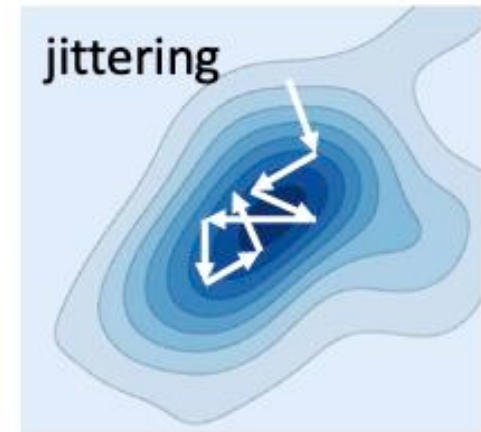
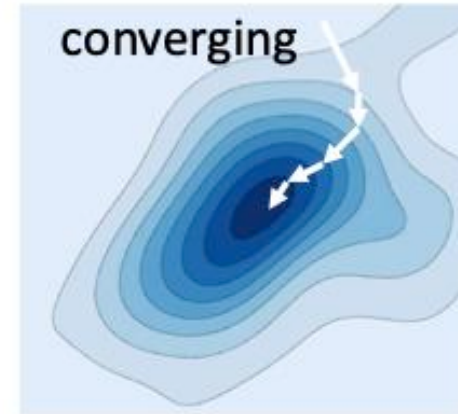
$$\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$$

- Parameter space \mathbf{W} is VERY high dimensional
- Loss Surface $J(\mathbf{W})$ can be highly non-convex
 - What is a convex loss function?
A surface is “convex” if it is continuously curving upward
 - We can connect any two points on or above the surface without intersecting it
 - Many mathematical definitions that are equivalent



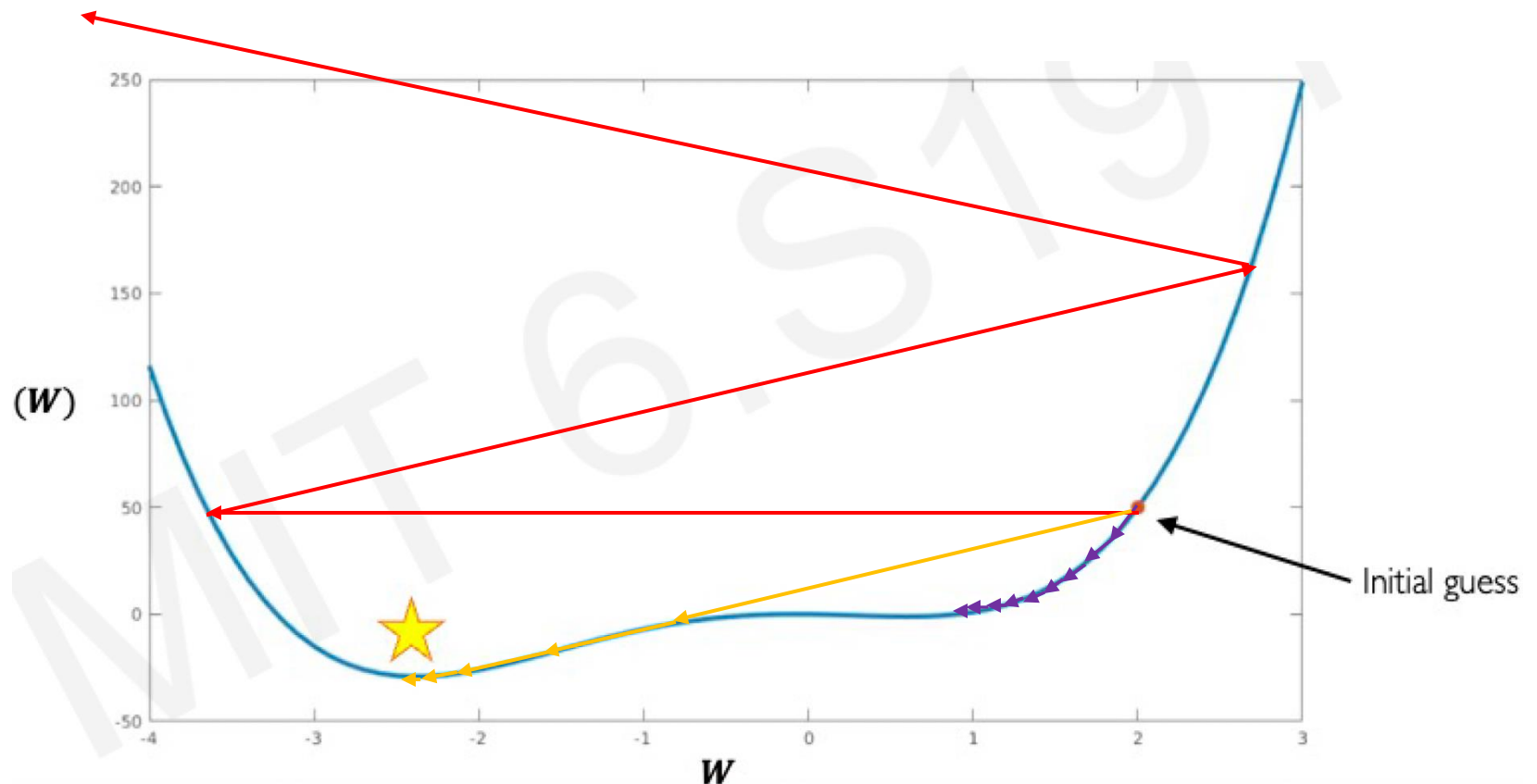
Convergence of Gradient Descent

- An iterative algorithm is said to converge to a solution if the value updates arrive at a fixed point
 - Where the gradient is 0 and further updates do not change the estimate
- The algorithm may not actually converge
 - It may jitter around the local minimum
 - It may even diverge
- Conditions for convergence?



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



Convergence and convergence rate

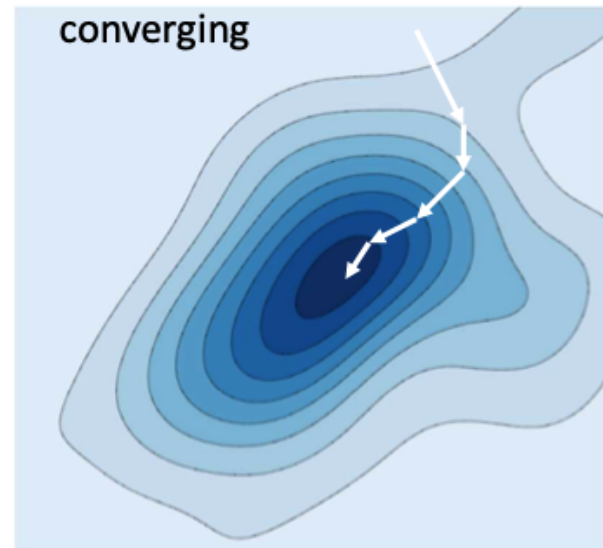
- Convergence rate: How fast the iterations arrive at the solution
- Generally quantified as

$$R = \frac{|f(x^{(k+1)}) - f(x^*)|}{|f(x^{(k)}) - f(x^*)|}$$

- $x^{(k+1)}$ is the k -th iteration
- x^* is the optimal value of x

- If R is a constant (or upper bounded), the convergence is *linear*
 - In reality, its arriving at the solution exponentially fast

$$|f(x^{(k)}) - f(x^*)| \leq R^k |f(x^{(0)}) - f(x^*)|$$



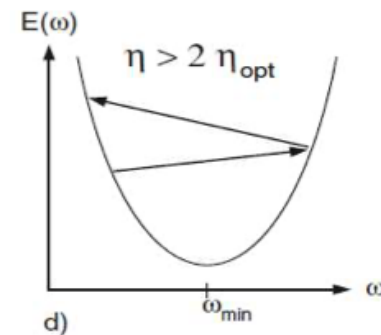
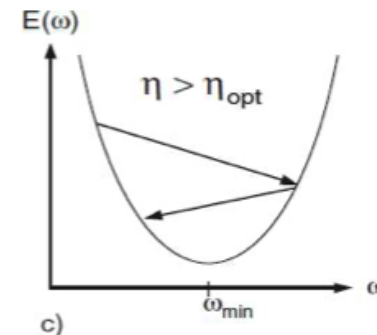
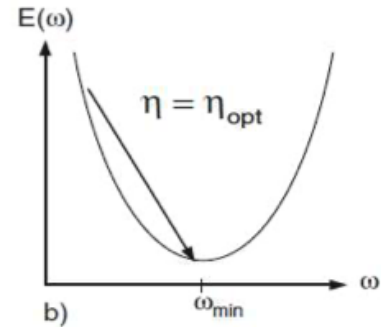
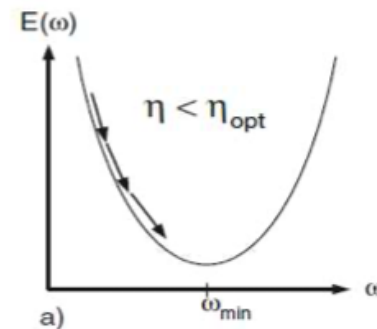
Convergence for quadratic surfaces

- Consider the problem

$$\min E = \frac{1}{2}aw^2 + bw + c$$

$$w_{k+1} = w_k - \eta \frac{dE(w_k)}{dw}$$

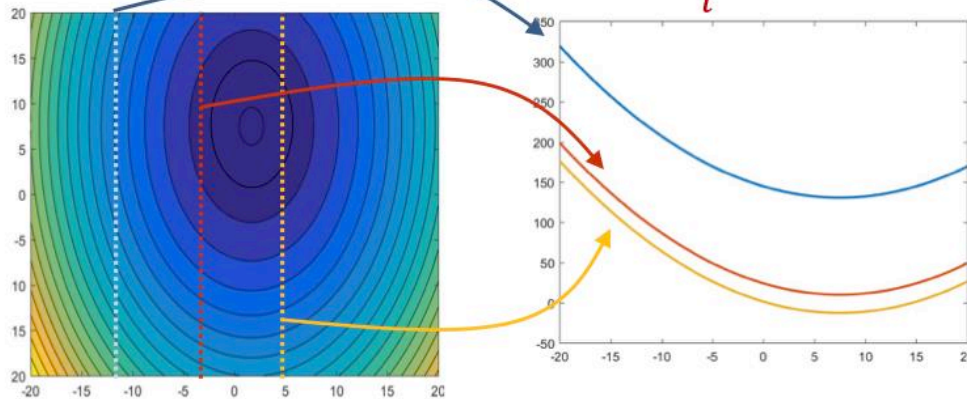
- It is easy to show that the optimal step size is $\eta_{\text{opt}} = a^{-1}$
 - For $\eta < \eta_{\text{opt}}$ the algorithm will converge monotonically
 - For $2\eta_{\text{opt}} > \eta > \eta_{\text{opt}}$ we have oscillating convergence
 - For $\eta > 2\eta_{\text{opt}}$ we get divergence



For functions of multivariate inputs

- $E = g(w)$, w is a vector $w = [w_1, w_2, \dots, w_N]$
- Example of Multivariate Quadratic with Diagonal A

$$E = \frac{1}{2} \mathbf{w}^T \mathbf{A} \mathbf{w} + \mathbf{w}^T \mathbf{b} + c = \frac{1}{2} \sum_i (a_{ii} w_i^2 + b_i w_i) + c$$

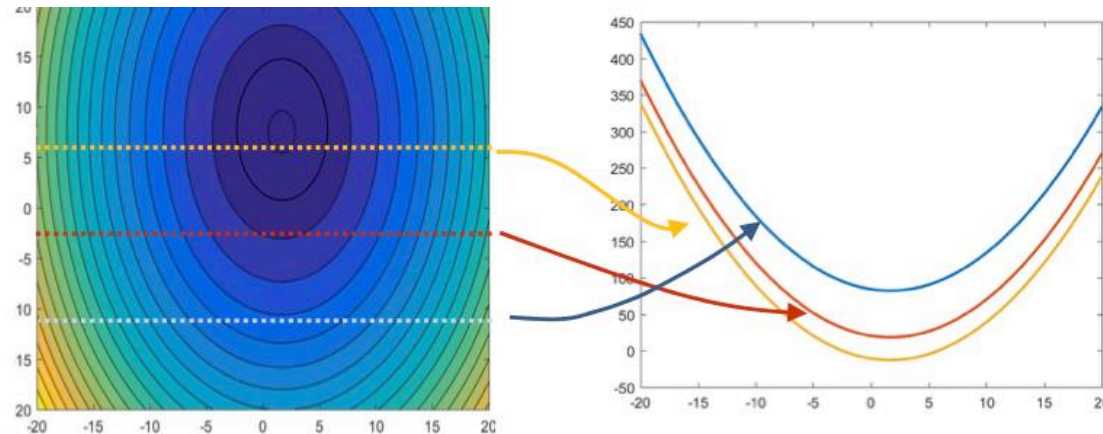


$$w_{k+1} = w_k - \eta \frac{dE(w_k)}{dw}$$

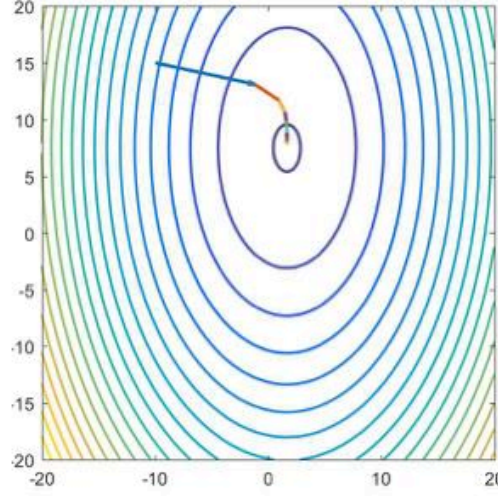
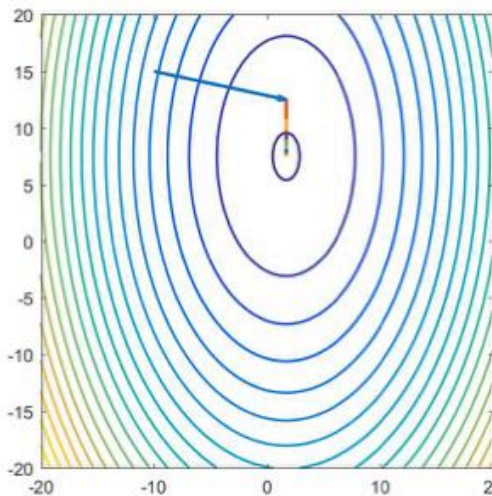
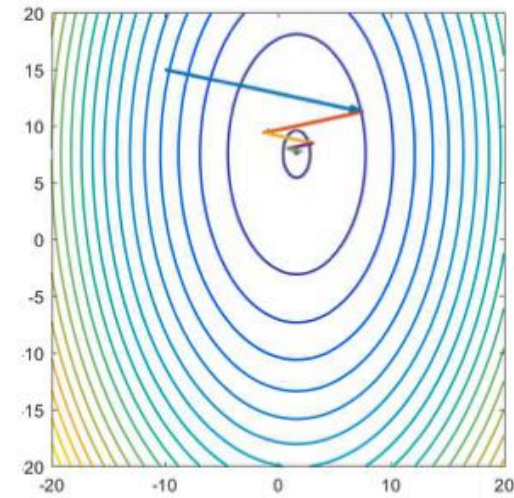
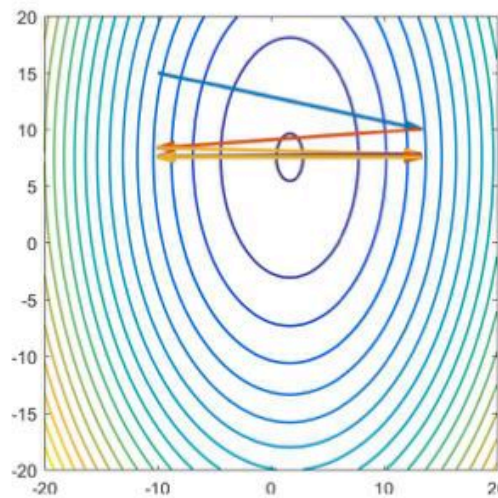
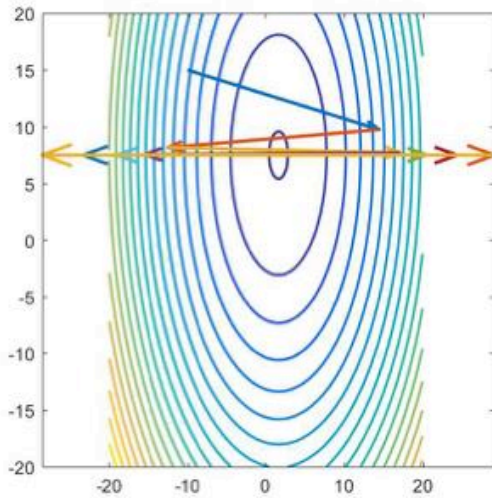
Optimal step size: $\eta_{1,\text{opt}} = a_{11}^{-1}$

Optimal step size: $\eta_{2,\text{opt}} = a_{22}^{-1}$

Issue: vector update rule
uses the same scalar η for
all parameters



Dependence on learning rate



- $\eta_{1,opt} = 1; \eta_{2,opt} = 0.33$
- $\eta = 2.1\eta_{2,opt}$
- $\eta = 2\eta_{2,opt}$
- $\eta = 1.5\eta_{2,opt}$
- $\eta = \eta_{2,opt}$
- $\eta = 0.75\eta_{2,opt}$

55

Choosing the smallest among the optimal rates should work, but can be very slow and get stuck in false local minimum

Training Neural Networks is Difficult

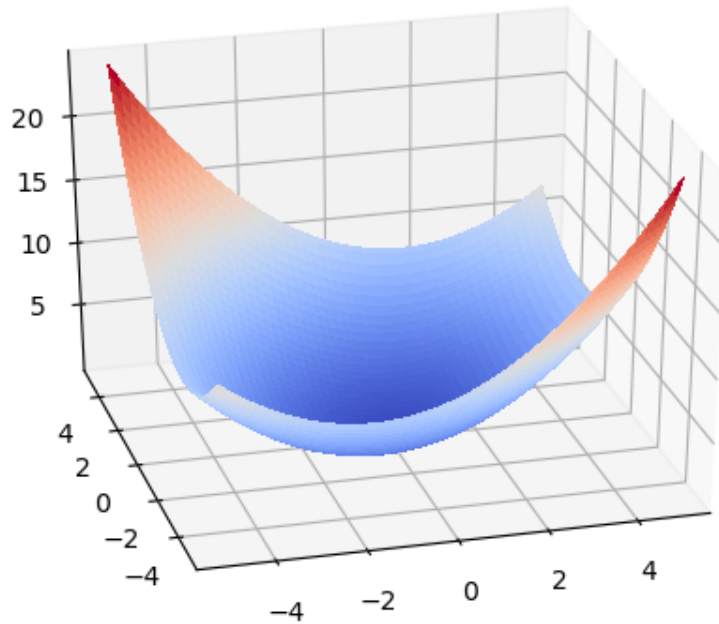


Fig. 1: Loss landscape of a linear regression with 2 parameters ([source¹](#))

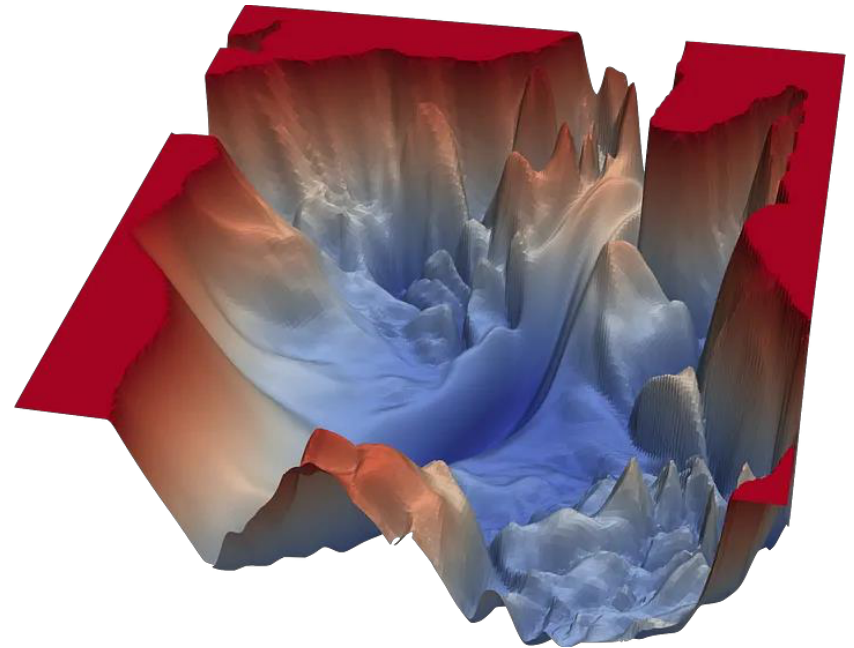
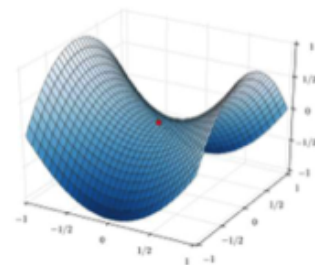
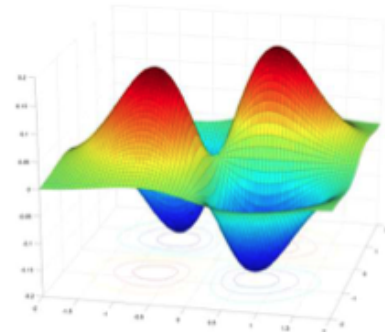
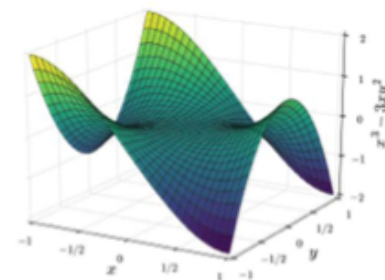


Fig. 2: Loss landscape of a convolutional neural network with 56 layers (VGG-56, [source¹](#))

The Loss Surface

- **Popular hypothesis:**
 - In large networks, saddle points are far more common than local minima
 - Frequency of occurrence exponential in network size
 - Most local minima are equivalent
 - And close to global minimum
 - This is not true for small networks
- **Saddle point:** A point where
 - The slope is zero
 - The surface increases in some directions, but decreases in others
 - Some of the Eigenvalues of the Hessian are positive; others are negative
 - Gradient descent algorithms often get “stuck” in saddle points



The Controversial Loss Surface

- **Baldi and Hornik (89)**, *“Neural Networks and Principal Component Analysis: Learning from Examples Without Local Minima”* : An MLP with a *single* hidden layer has only saddle points and no local Minima
- **Dauphin et. al (2015)**, *“Identifying and attacking the saddle point problem in high-dimensional non-convex optimization”* : An exponential number of saddle points in large networks
- **Chomoranksa et. al (2015)**, *“The loss surface of multilayer networks”* : For large networks, most local minima lie in a band and are equivalent
 - Based on analysis of spin glass models
- **Swirszcz et. al. (2016)**, *“Local minima in training of deep networks”*, In networks of finite size, trained on finite data, you *can* have horrible local minima
- Watch this space...



How to deal with this?

- Idea 1:
 - Hyperparameter tuning: try lots of different learning rates and see what works “just right”
- Idea 2:
 - Learning rate scheduler: fixed algorithms that change learning rate over epochs (typically, start at higher value and gradually lower it)
- Idea 3:
 - Sophisticated optimizers: techniques that adapt learning rate to current loss landscape

ALWAYS do
hyper tuning






Can help, but
interacts with
other ideas

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

Idea 3: Sophisticated Optimizers

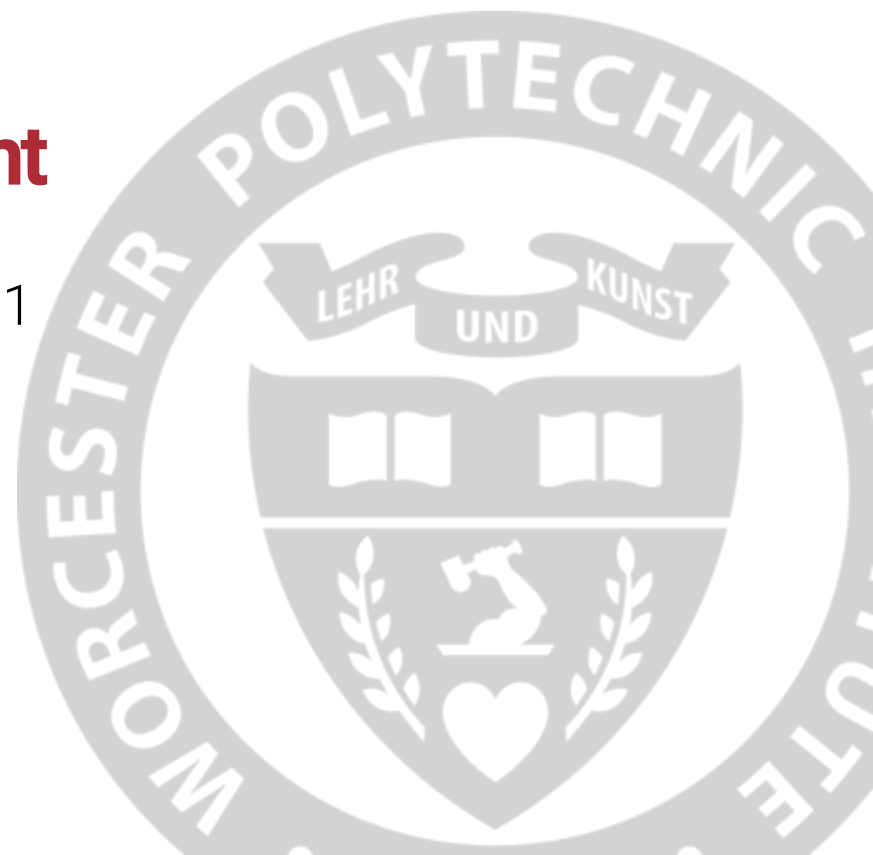
- Based on mini-batches (subsets of training data)
- Learning rates are no longer fixed
- Can be made larger or smaller depending on:
 - Whether previous gradients are going “in the same direction”
 - Whether gradients in certain directions are changing much or little

Gradient-based Optimizers

Algorithm	Pytorch implementation	Reference
<ul style="list-style-type: none">Stochastic Gradient Descent (SGD)	 <code>torch.optim.SGD</code>	Kiefer & Wolfowitz. "Stochastic Estimation of the Maximum of a Regression Function." 1952.
<ul style="list-style-type: none">Adagrad	 <code>torch.optim.Adagrad</code>	Methods for Online Learning and Stochastic Optimization." 2011.
<ul style="list-style-type: none">RMSProp	 <code>torch.optim.RMSprop</code>	Hinton. Lecture 6e . Coursera.
<ul style="list-style-type: none">Adam	 <code>torch.optim.Adam</code>	Duchi et al. "Adaptive Subgradient
<ul style="list-style-type: none">AdamW	 <code>torch.optim.AdamW</code>	Kingma et al. "Adam: A Method for Stochastic Optimization." 2014.

Stochastic Gradient Descent

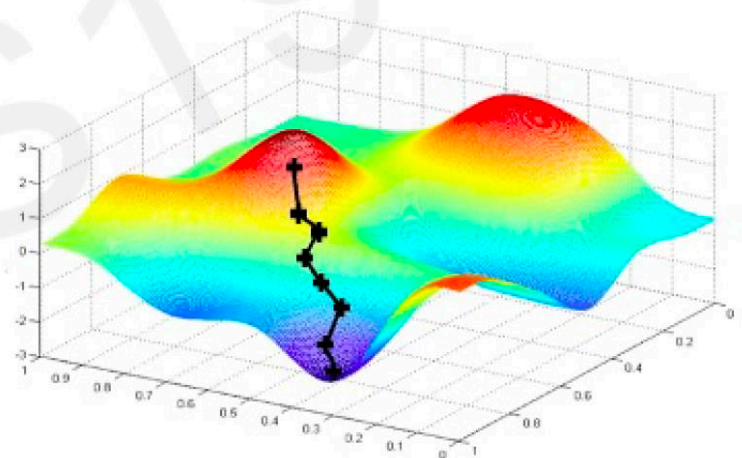
Credits: first slides based on MIT 6.S191



Gradient Descent

Algorithm

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

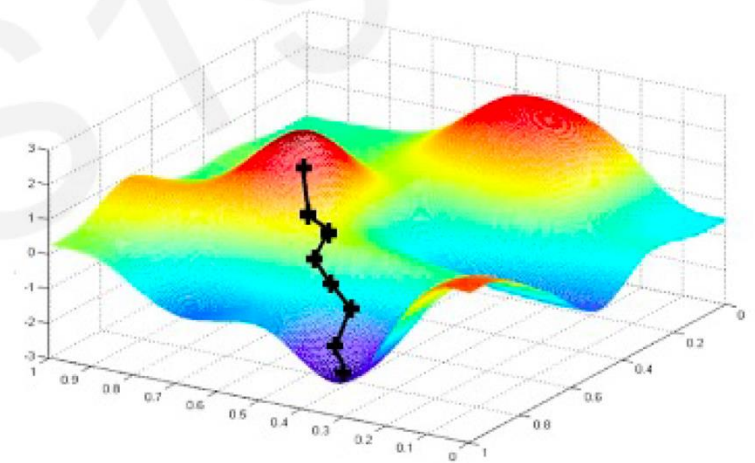


Computes loss gradients over **entire training data**; can be very **computationally intensive**!

Stochastic Gradient Descent

Algorithm

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

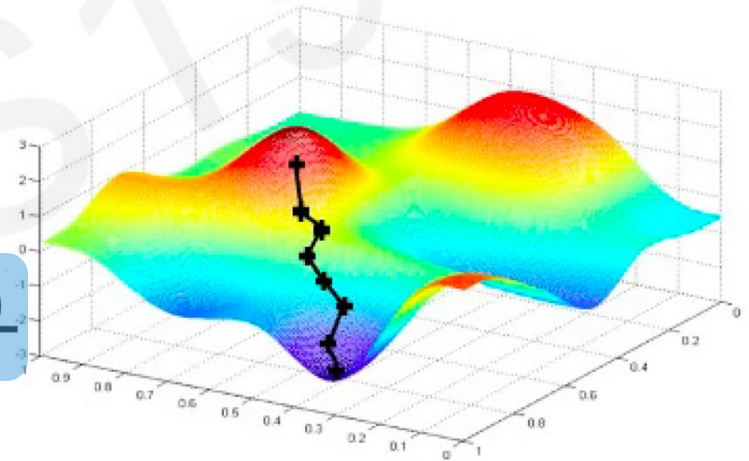


Easy to compute but
very noisy (stochastic)!

(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, $\frac{\partial J(\mathbf{W})}{\partial \mathbf{W}} = \frac{1}{B} \sum_{k=1}^B \frac{\partial J_k(\mathbf{W})}{\partial \mathbf{W}}$
5. Update weights, $\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$
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).

“Sampling” description of SGD

- This is the idea behind **stochastic gradient descent** (SGD):
 - Randomly sample a small ($\ll n$) **mini-batch** (or sometimes just **batch**) of training examples.
 - Estimate the gradient on just the mini-batch.
 - Update weights based on *mini-batch* gradient estimate.
- Repeat.

Is expected value of
stochastic gradient equal
to gradient?

In practice, we don't
“sample”: shuffle and
iterate over mini batches

“Shuffling” description of SGD

- In practice, SGD is usually conducted over multiple epochs.
 - An **epoch** is a single pass through the entire training set.

- Procedure:

1. Let $\tilde{n} \ll n$ equal the size of the mini-batch.

2. For $e = 0$ to numEpochs:

- I. Randomize the order of the examples in the training set.

- II. For $i = 0$ to $\tilde{n} \ll n$ (one epoch):

- A. Select a mini-batch \mathcal{J} containing the next \tilde{n} examples.

- B. Compute the gradient on this mini-batch: $\frac{1}{\tilde{n}} \sum_{i \in \mathcal{J}} \nabla_{\mathbf{w}} f(\mathbf{y}^{(i)}, \hat{\mathbf{y}}^{(i)}; \mathbf{W})$

- C. Update the weights based on the current mini-batch gradient.

What is the advantage of shuffling in every epoch?

SGD: variable learning rate

- One common learning rate “schedule” is to multiply ϵ by $c \in (0, 1)$ after every epoch.
 - This is called **exponential decay**.
- Another possibility (which avoids the issue) is to set the number of epochs T to a finite number.
 - SGD may not fully converge, but the machine might still perform well.
- There are many other strategies.

Pro tip: watch the loss function

Stochastic gradient descent

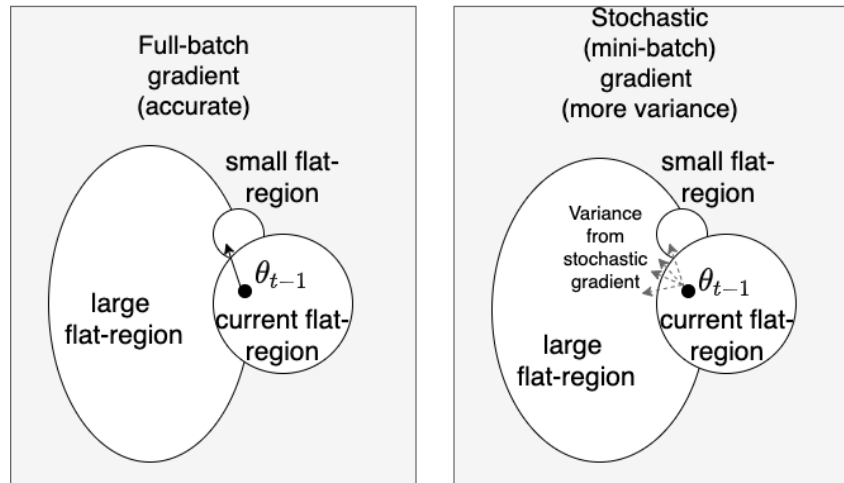
- **Despite** the “noise” (statistical inaccuracy) in the mini-batch gradient estimates, we often converge to good parameterizations.
- Reaching a “low” loss value can be much faster than regular gradient descent. Why?
 - Because we adjust the weights *many times* per epoch.

Why SGD matters?

- We often think of SGD as workaround for training with large data (cannot load it all in memory at once)
- In practice, it also leads to weights that **generalize better** at test time than full-batch gradient descent. Why?
 - We do not fully understand yet!
 - A lot of the ⁵⁸explanations are speculative, but with mounting empirical evidence

Hypothesis: SGD prefers large flat minima regions

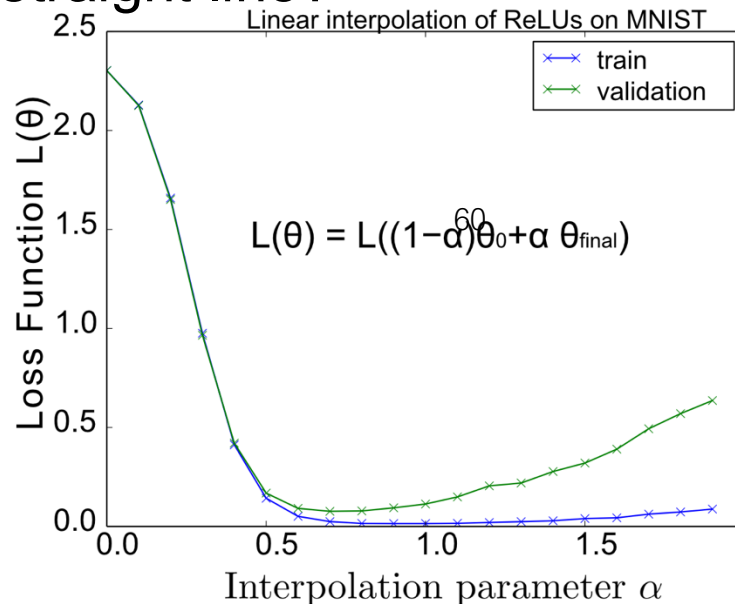
- **Full-batch gradient** (entire dataset) at step $t-1$: low-noise gradient, points to the lowest minima in the vicinity of θ_{t-1} , often a small region
- **SGD** at step $t-1$: noisy gradient, points to various lower-loss regions in the vicinity of θ_{t-1} .



*Variance from SGD tends to move towards **large flat regions**, preventing it from getting "stuck"*

Evidence of moving between flat regions

- ([Goodfellow et al., ICLR 2015](#)) considers a MLP for handwritten digit classification.
- Let θ_0 be the initial (random) params and θ_f be the final parameters after SGD with very high accuracy.
- What the loss looks like if we interpolate θ_0 and θ_f using a straight line?



*If flat regions were small,
training loss should go
up sometimes*

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.

Visual Examples of Training issues

1. Loss diverged
2. Loss too jittery
3. Stopped before convergence
4. Overfitting
5. Converges to different values every time

DONE ON THE WHITEBOARD