机器学习与数据挖掘

Machine Learning & Data Mining

权小军 教授

中山大学数据科学与计算机学院

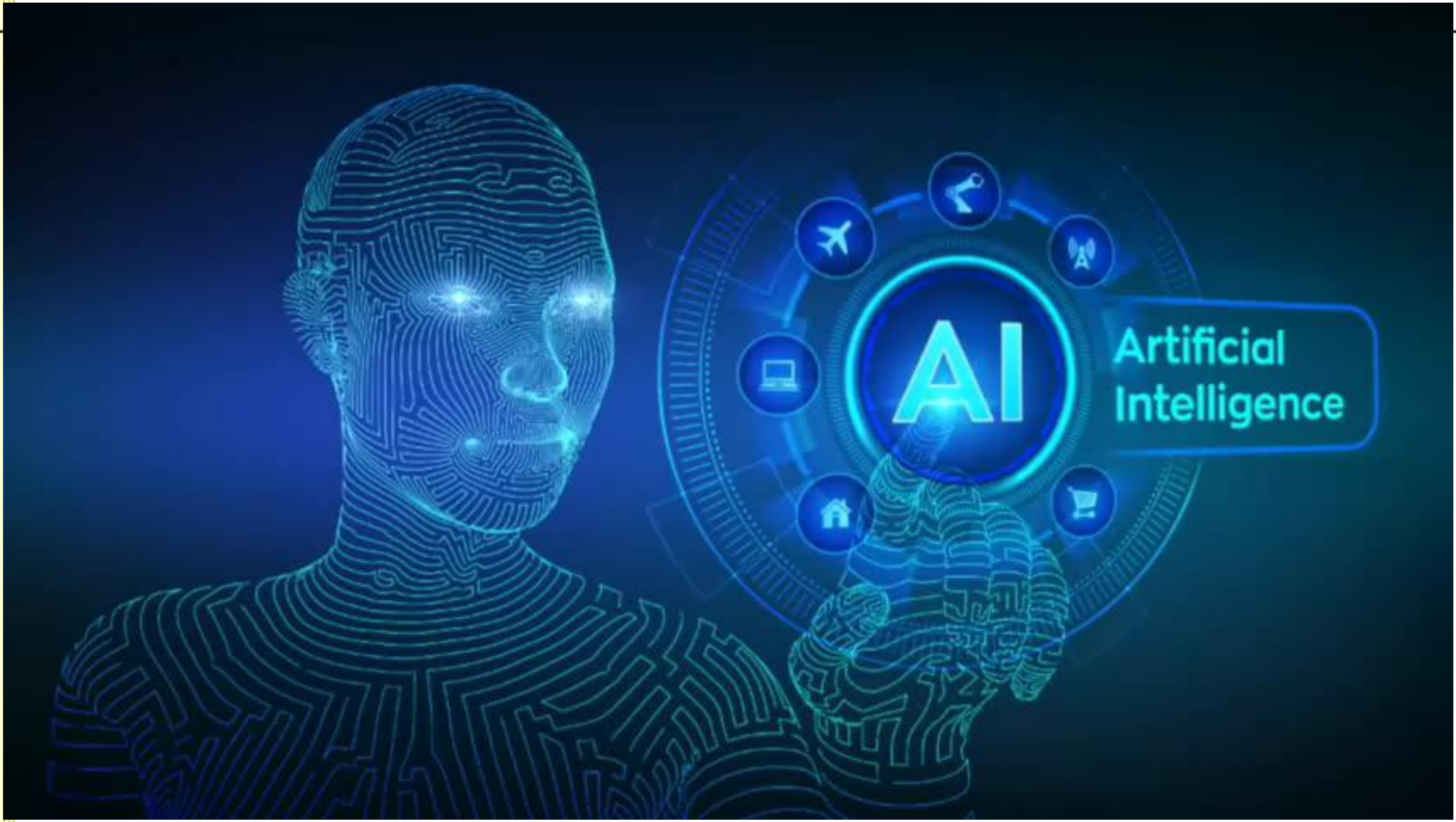
quanxj3@mail.sysu.edu.cn

Preface

深度学习“炼丹术”

深度学习的模型训练就是“炼丹”。把精选原始数据，按照神经网络的规定法则通过计算框架提炼，从而得到一个远小于数据数倍的模型。好的模型能抓取数据中的模式，以及更加一般化规则用来预测新的数据。

Preface



Source: https://www.bilibili.com/video/BV1ht4y117Me?from=search&seid=3013146020457294931

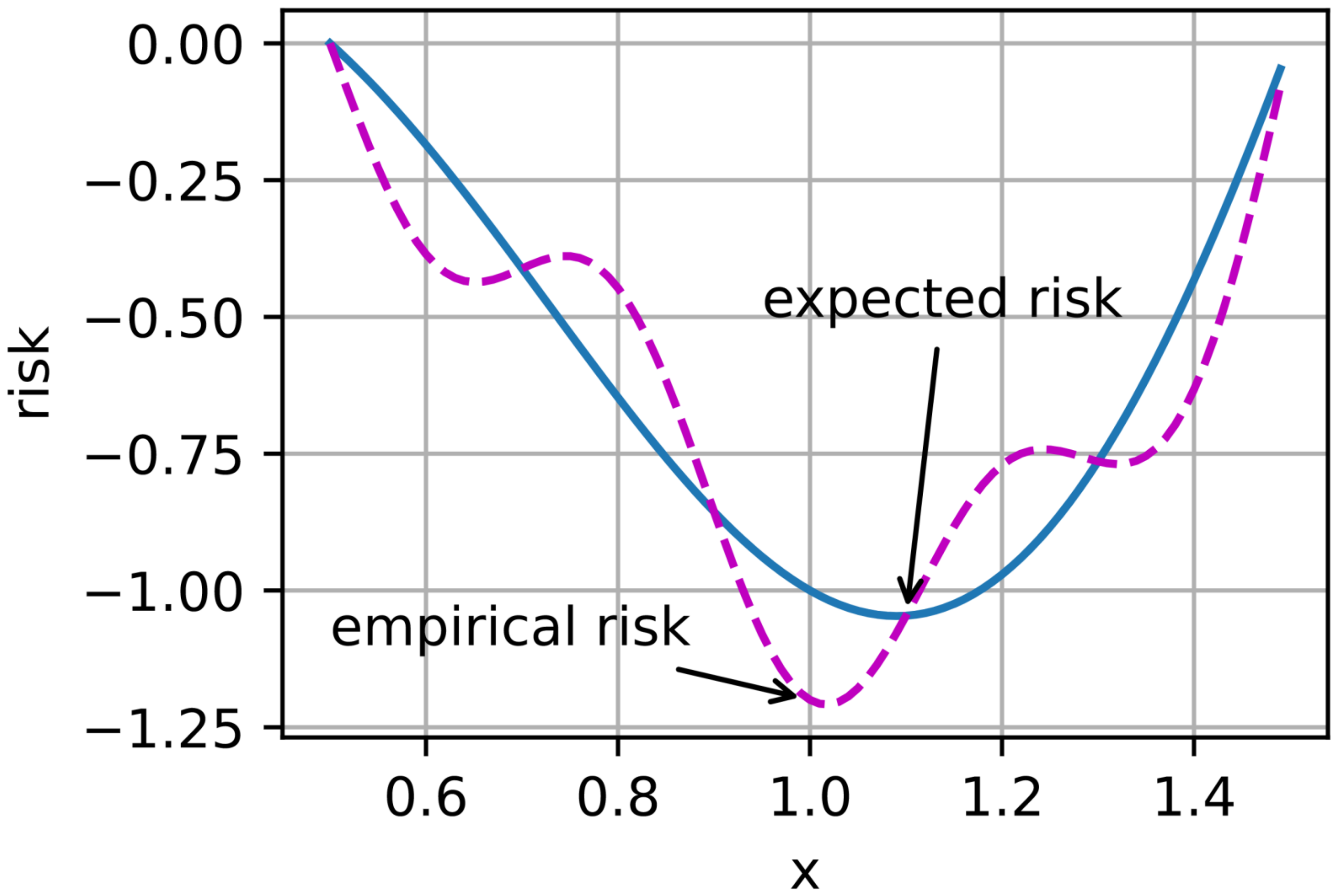
Preface

- Although optimization provides a way to minimize the loss function for deep learning, in essence, the goals of optimization and deep learning are fundamentally different.

- Optimization is primarily concerned with minimizing an objective whereas the latter is concerned with finding a suitable model, given a finite amount of data.

- The goal of optimization is to reduce the training error. However, the goal of statistical inference (and thus of deep learning) is to reduce the generalization error.

Preface


0.00
−0.25
−0.50
−0.75
−1.00
−1.25
risk
expected risk
empirical risk
0.6
0.8
1.0
1.2
1.4
x

Preface

Optimization Challenges in Deep Learning

1. Local Minima
2. Vanishing Gradients
3. Mini-batch
4. Overfitting
5. Learning Rate

Lecture 11: Deep Learning II - Optimization

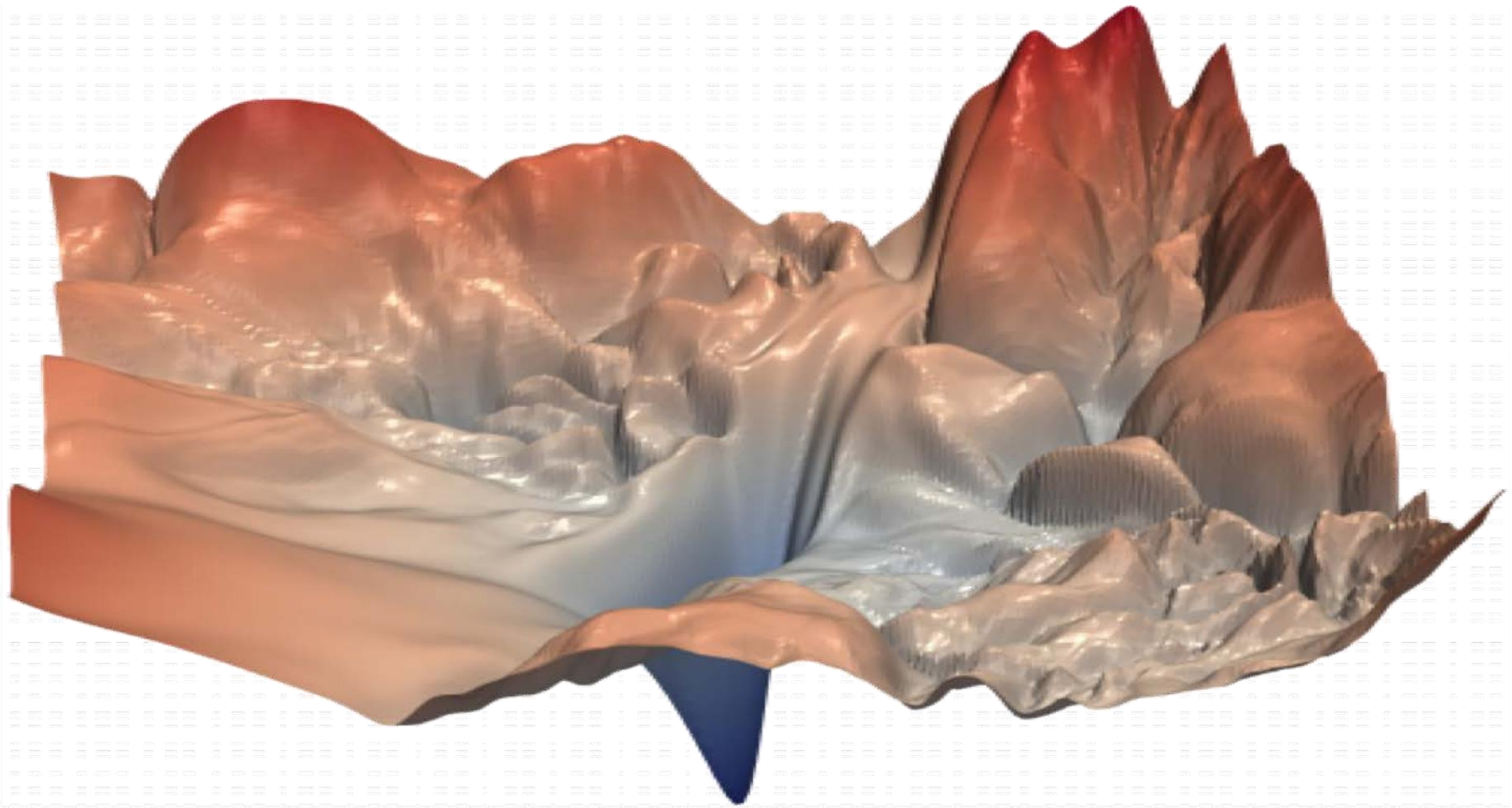
Optimization

Training neural networks is difficult!

- In deep learning, most objective functions are complicated and do not have analytical solutions.
- Instead, we must use numerical optimization algorithms.

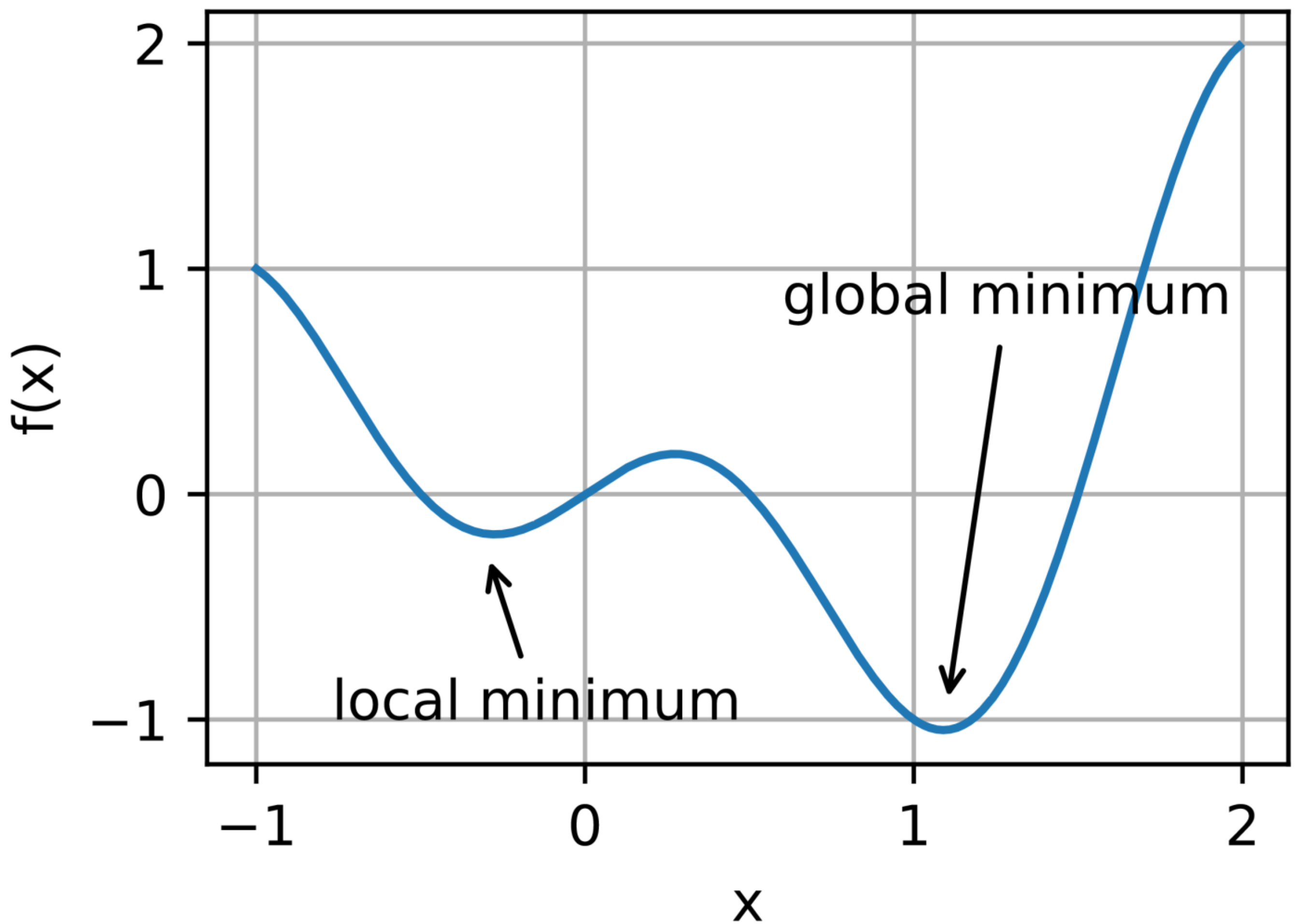
Optimization

Training neural networks is difficult!



11.1 Local Minima

Local Minima



Local Minima

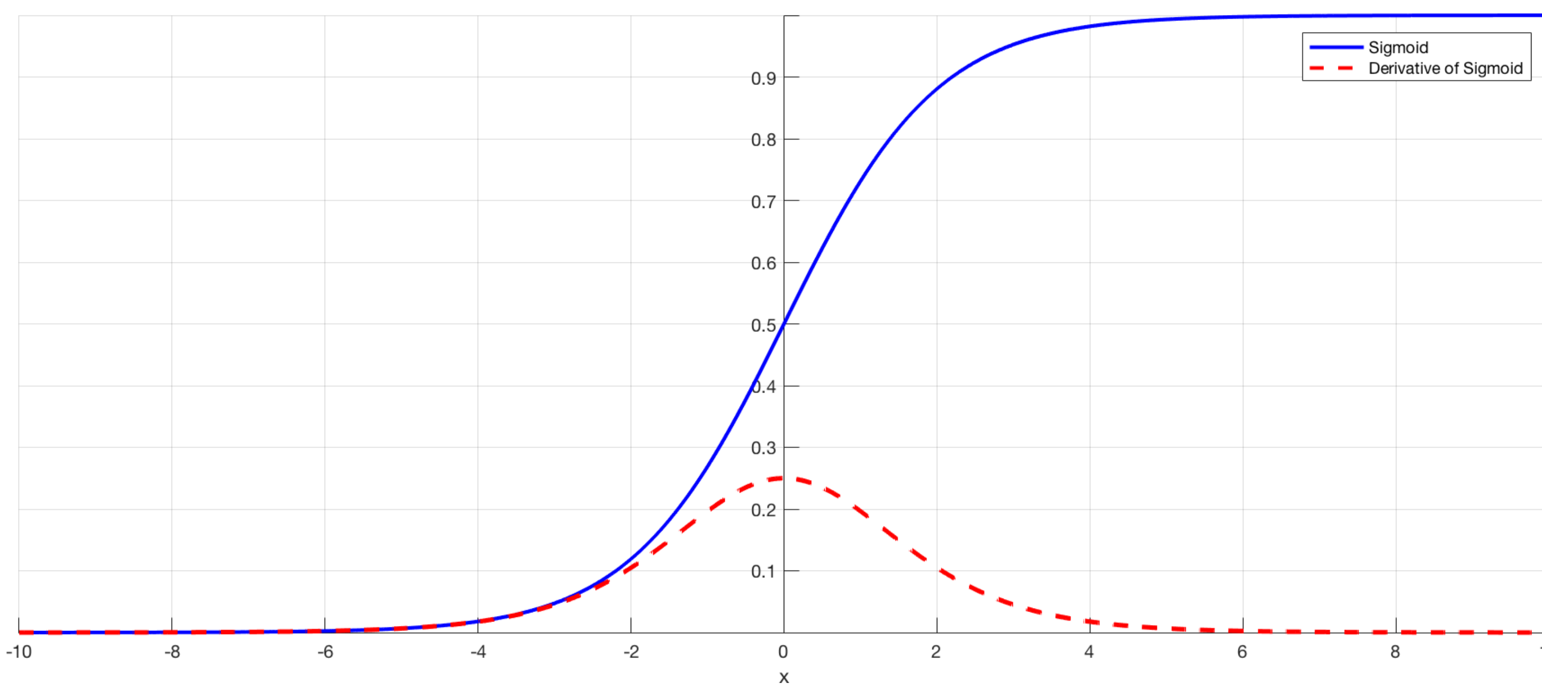
- One of the beneficial properties of stochastic gradient descent is that the natural variation of gradients over minibatches is able to dislodge the parameters from local minima.

Robert Kleinberg, Yuanzhi Li, Yang Yuan. An Alternative View: When Does SGD Escape Local Minima? ICML 2018.

11.2 Vanishing Gradients

Vanishing Gradients

- Certain activation functions, like the sigmoid function, squishes a large input space into a small input space between 0 and 1. Therefore, a large change in the input of the sigmoid function will cause a small change in the output. Hence, the derivative becomes small.



Vanishing Gradients

Why it's significant?

- For shallow network with only a few layers that use these activations, this isn't a big problem. However, when more layers are used, it can cause the gradient to be too small for training to work effectively.

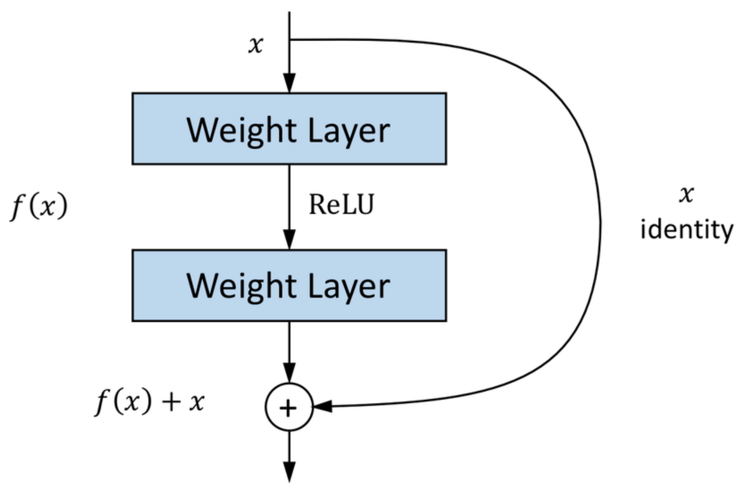
- However, when *n* hidden layers use an activation like the sigmoid function, *n* small derivatives are multiplied together. Thus, the gradient decreases exponentially as we propagate down to the initial layers.

Vanishing Gradients

Solutions

I. The simplest solution is to use other activation functions, such as ReLU, which doesn't cause a small derivative.

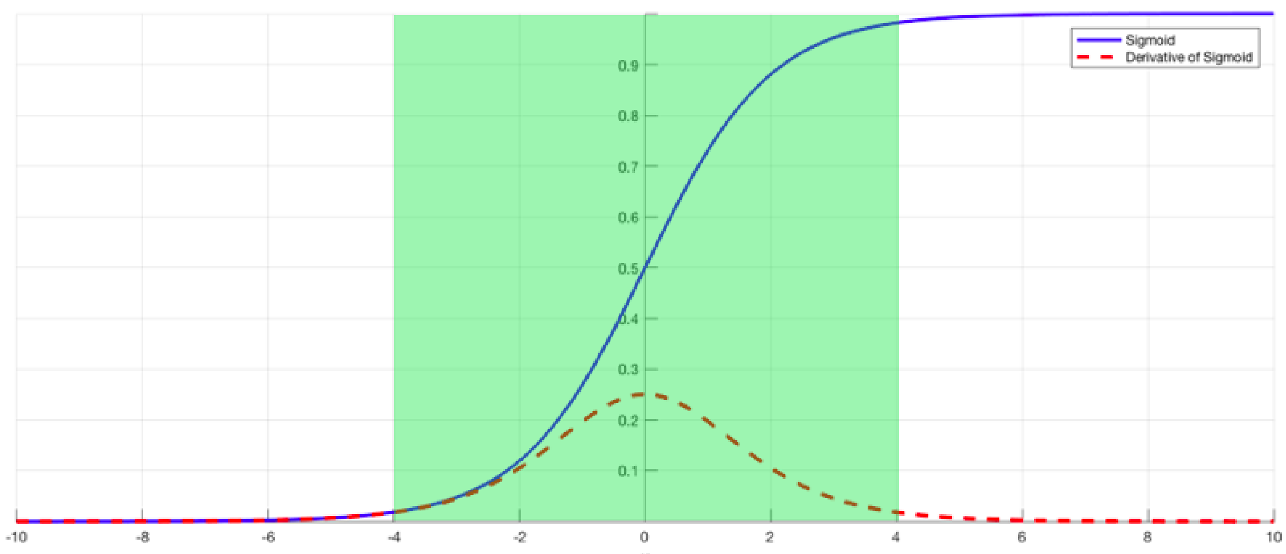
II. Residual networks are another solution, as they provide residual connections straight to earlier layers.



Vanishing Gradients

Solutions

III. Batch normalization layers can also resolve the issue. Vanishing Gradients arises when a large input space is mapped to a small one, causing the derivatives to disappear. Batch normalization reduces this problem by simply normalizing the input.

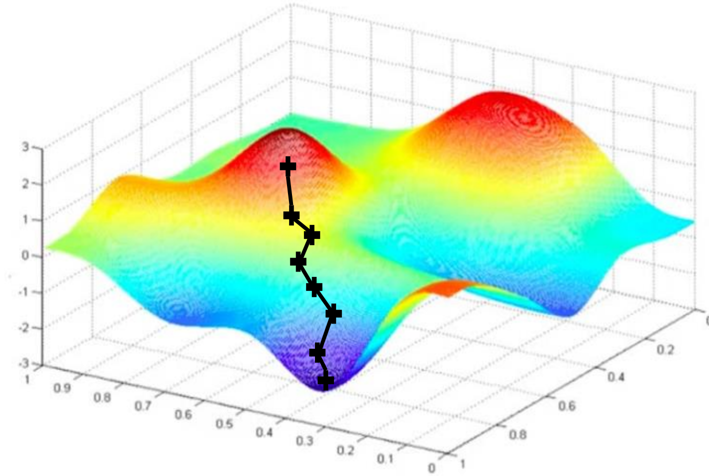


11.3 Mini-batches

Gradient Descent

Algorithm

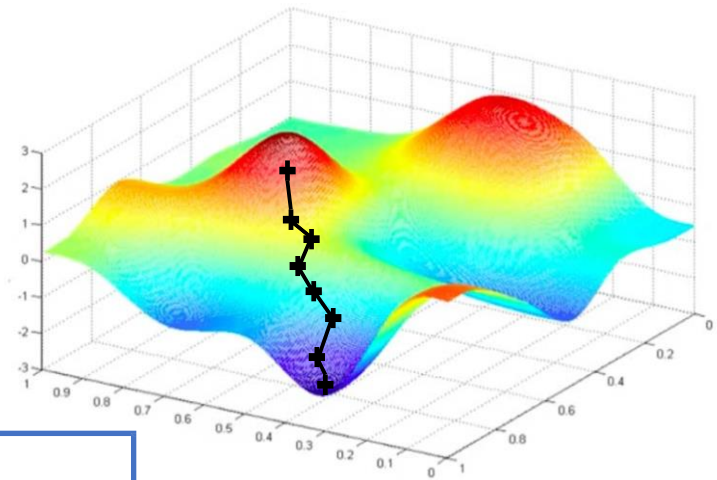
1. Initialize weights randomly $\sim N(0, \sigma^2)$
2. Loop until convergence:
3. Compute gradient $\frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
4. Update weights: $\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
5. Return weights



Gradient Descent

Algorithm

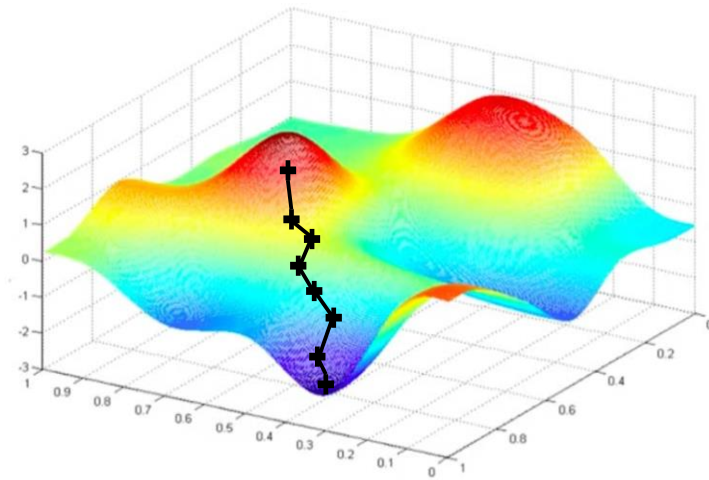
1. Initialize weights randomly $\sim N(0,\ \sigma^2)$
2. Loop until convergence:
3. Compute gradient $\frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
4. Update weights: $\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
5. Return weights



Can be very computationally intensive!

Stochastic Gradient Descent

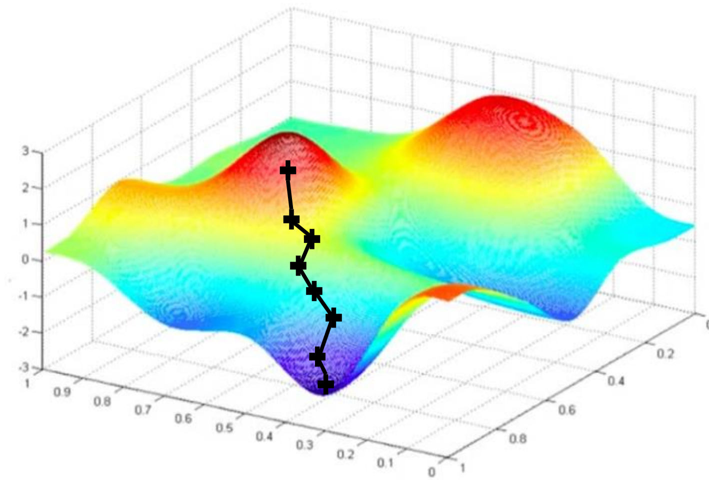
Algorithm



1. Initialize weights randomly $\sim N(0, \sigma^2)$
2. Loop until convergence:
3. Pick single data point 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

Stochastic Gradient Descent

Algorithm

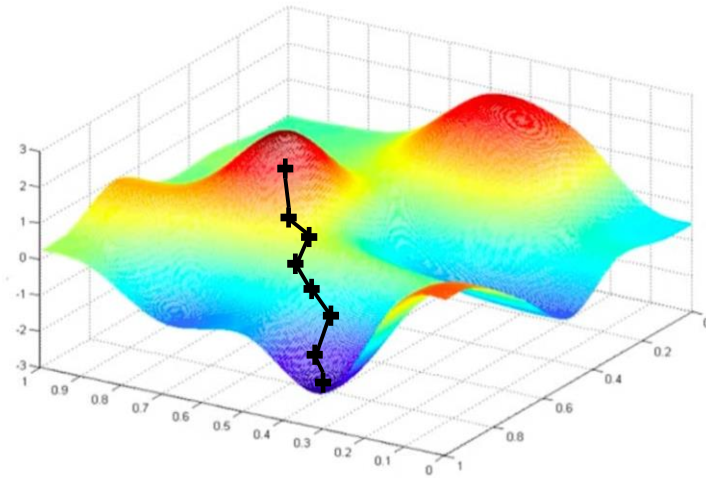


1. Initialize weights randomly $\sim N(0,\ \sigma^2)$
2. Loop until convergence:
3. Pick single data point i
4. Compute gradient $\dfrac{\partial J_i(\boldsymbol{W})}{\partial \boldsymbol{W}}$
5. Update weights: $\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \dfrac{\partial J(\boldsymbol{W})}{\partial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

Easy to compute but
very noisy (stochastic)!

Stochastic Gradient Descent

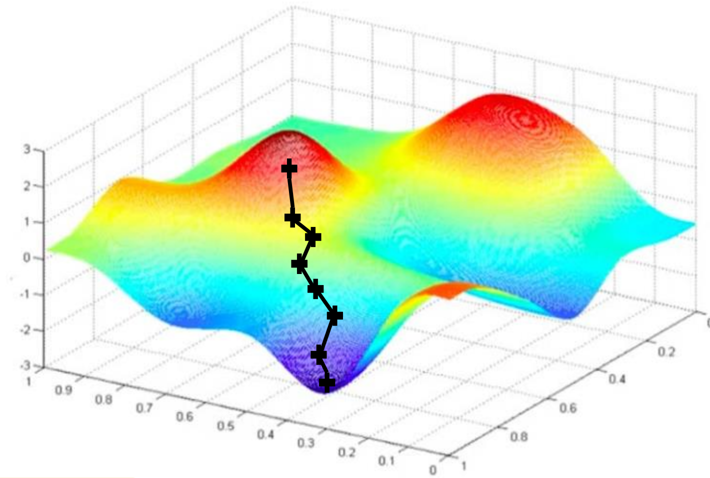
Algorithm



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

Stochastic Gradient Descent

Algorithm



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

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

Mini-batches

More accurate estimation of gradient

Smoother convergence

Allows for larger learning rates

Mini-batches

Mini-batches lead to fast training!

Can parallelize computation + achieve significant speed increases on GPU's

Mini-batches

Epoch vs **Batch Size** vs **Iterations**

- **Epoch**： One **Epoch** is when an ENTIRE dataset is passed forward and backward through the neural network only ONCE.

- **Batch Size**：Total number of training examples present in a single batch.

- **Iterations**： Iterations is the number of batches needed to complete one epoch.

Mini-batches

Epoch vs Batch Size vs Iterations

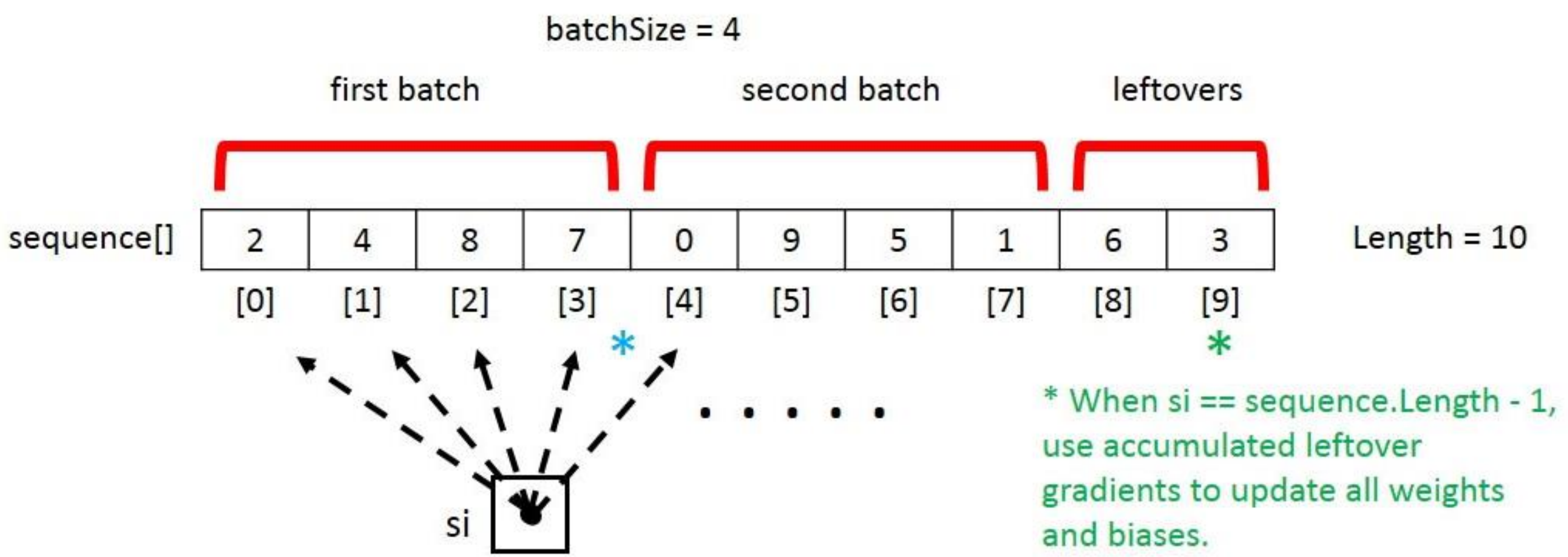
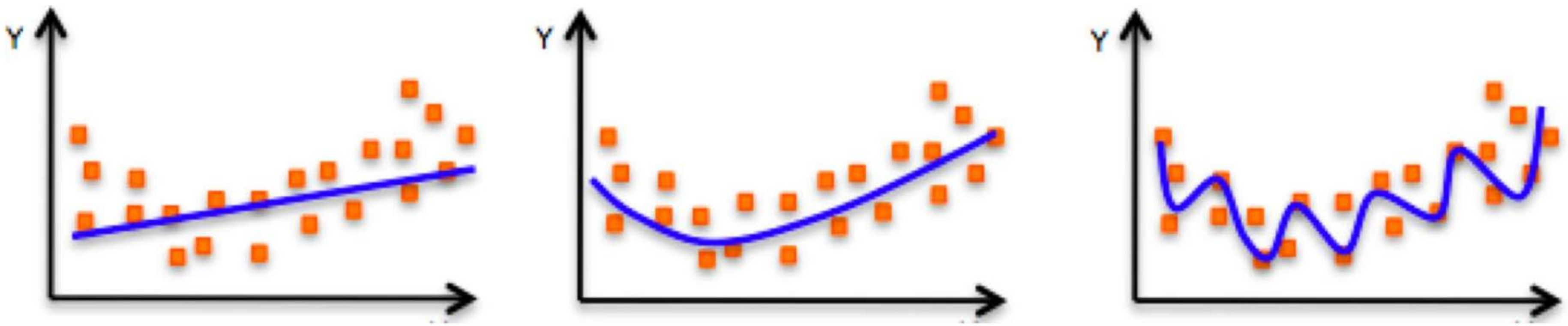


Figure source: https://visualstudiomagazine.com/articles/2015/07/01/variation-on-back-propagation.aspx

11.4 Overfitting

The Problem of Overfitting



Underfitting
Model does not have capacity to fully learn the data

Ideal fit

Overfitting
Too complex, extra parameters, does not generalize well

Regularization

What is it?

Technique that constrains our optimization problem to discourage complex models

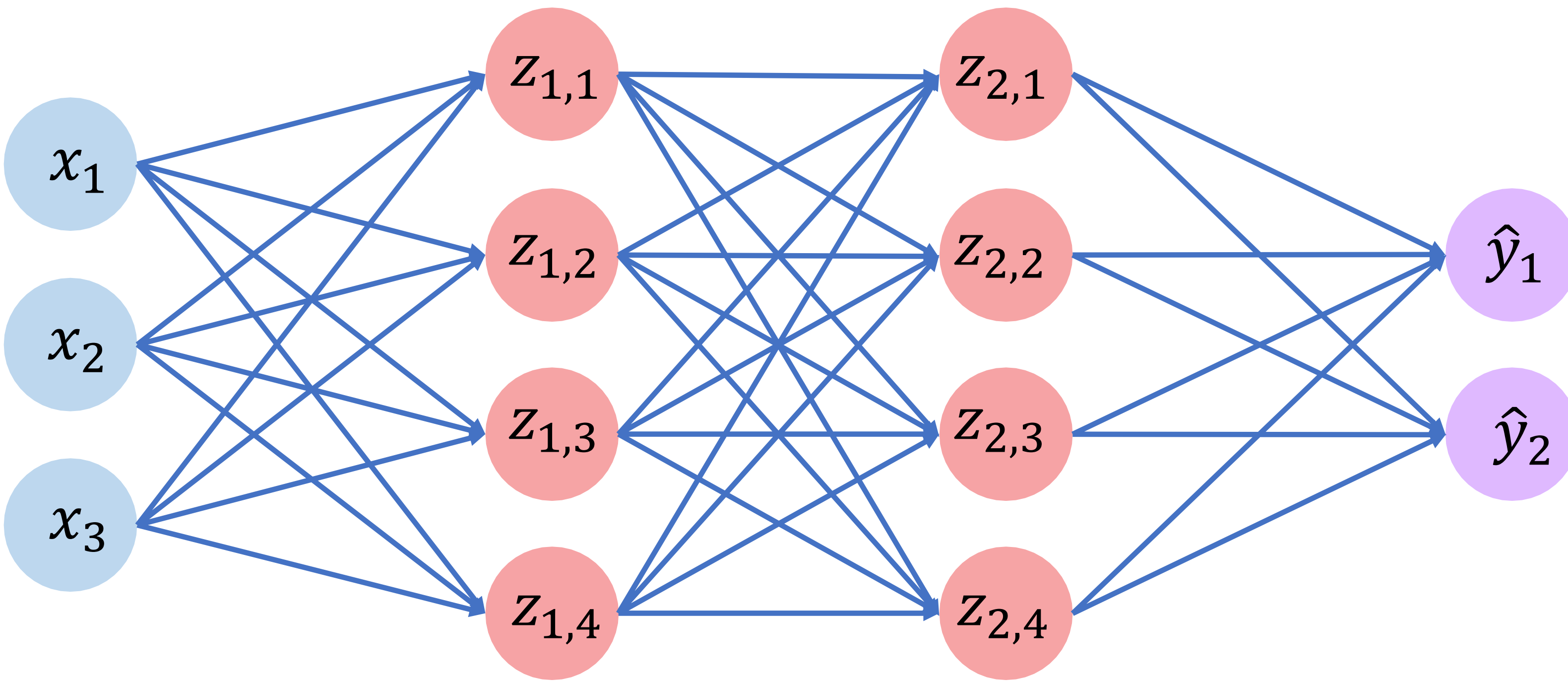
Regularization

Why do we need it?

Improve generalization of our model on unseen data

Regularization 1: Dropout

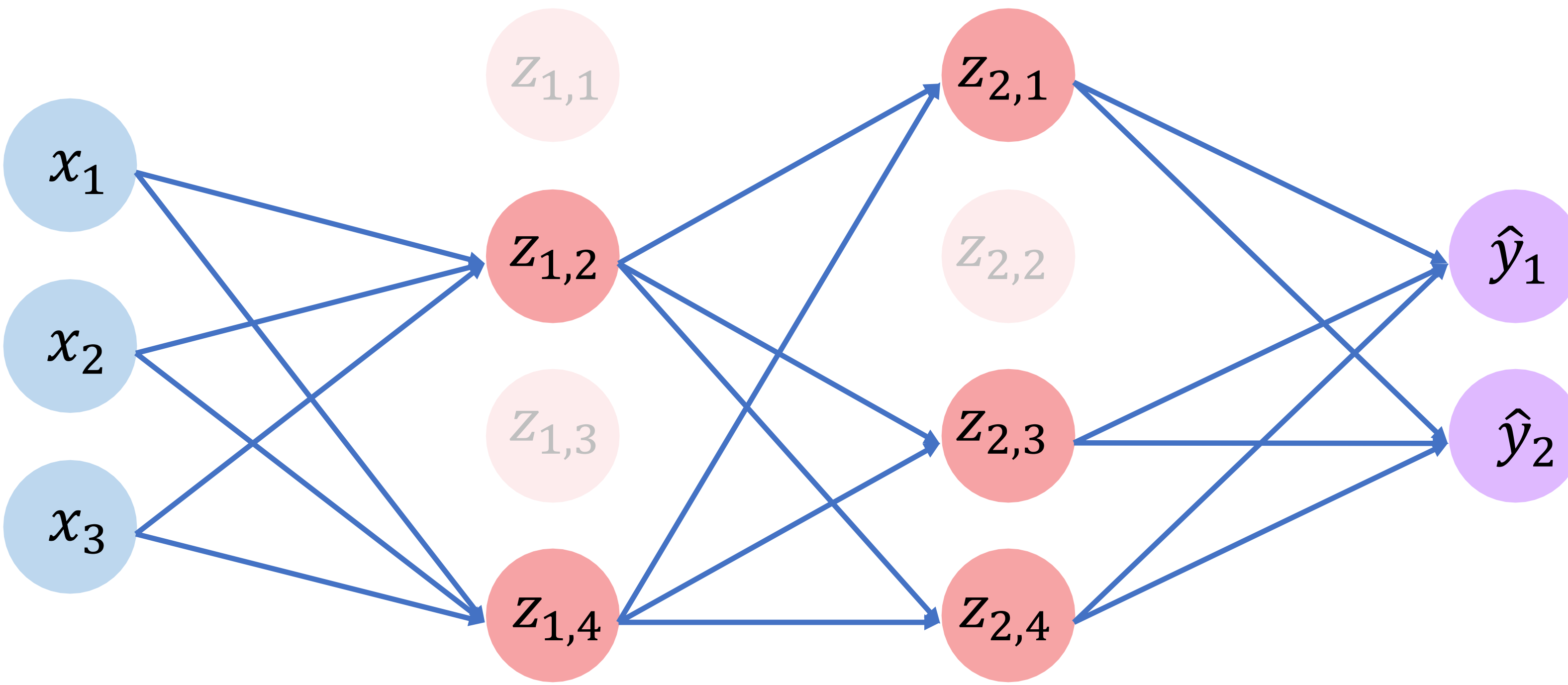
- During training, randomly set some activations to 0



Regularization 1: Dropout

- During training, randomly set some activations to 0
  - Typically ‘drop’ 50% of activations in layer
  - Forces network to not rely on any 1 node

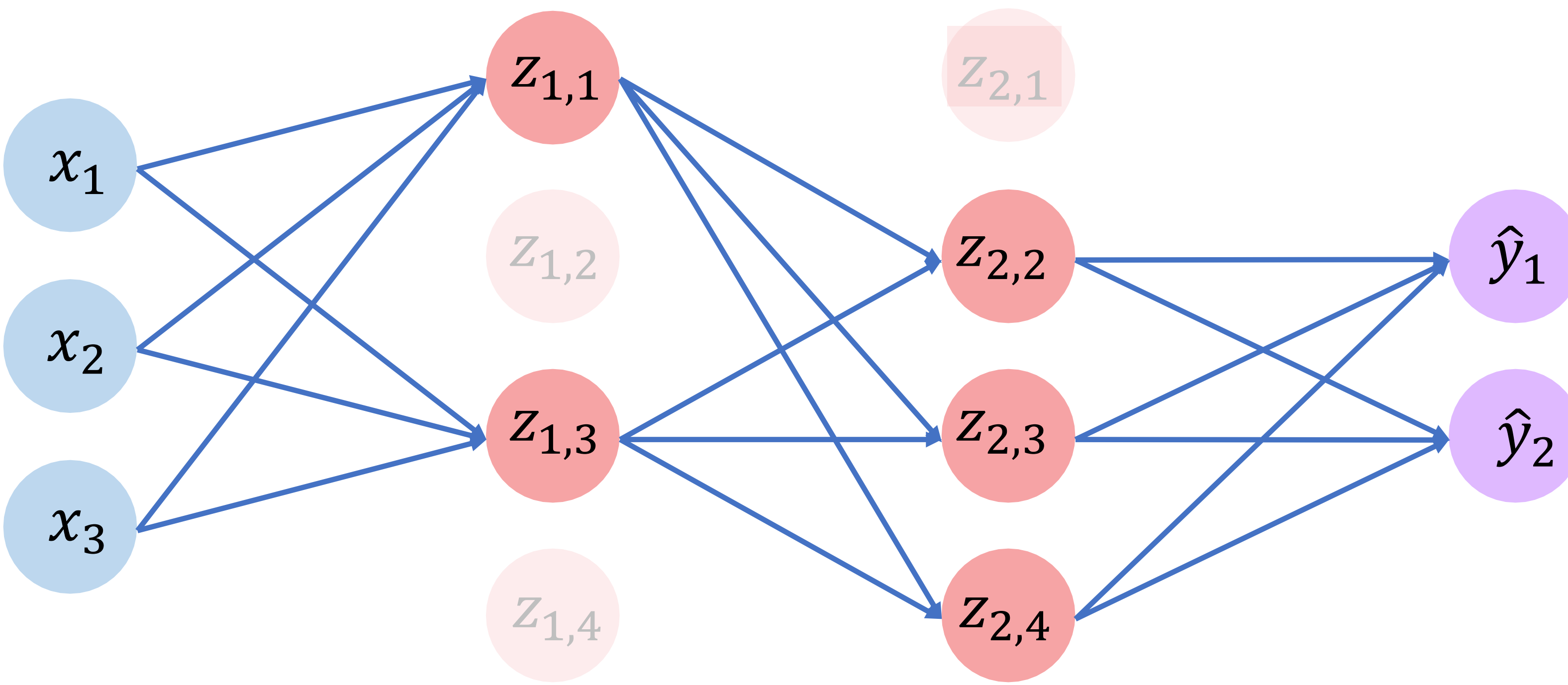
```
tf.keras.layers.Dropout(p=0.5)
```



Regularization 1: Dropout

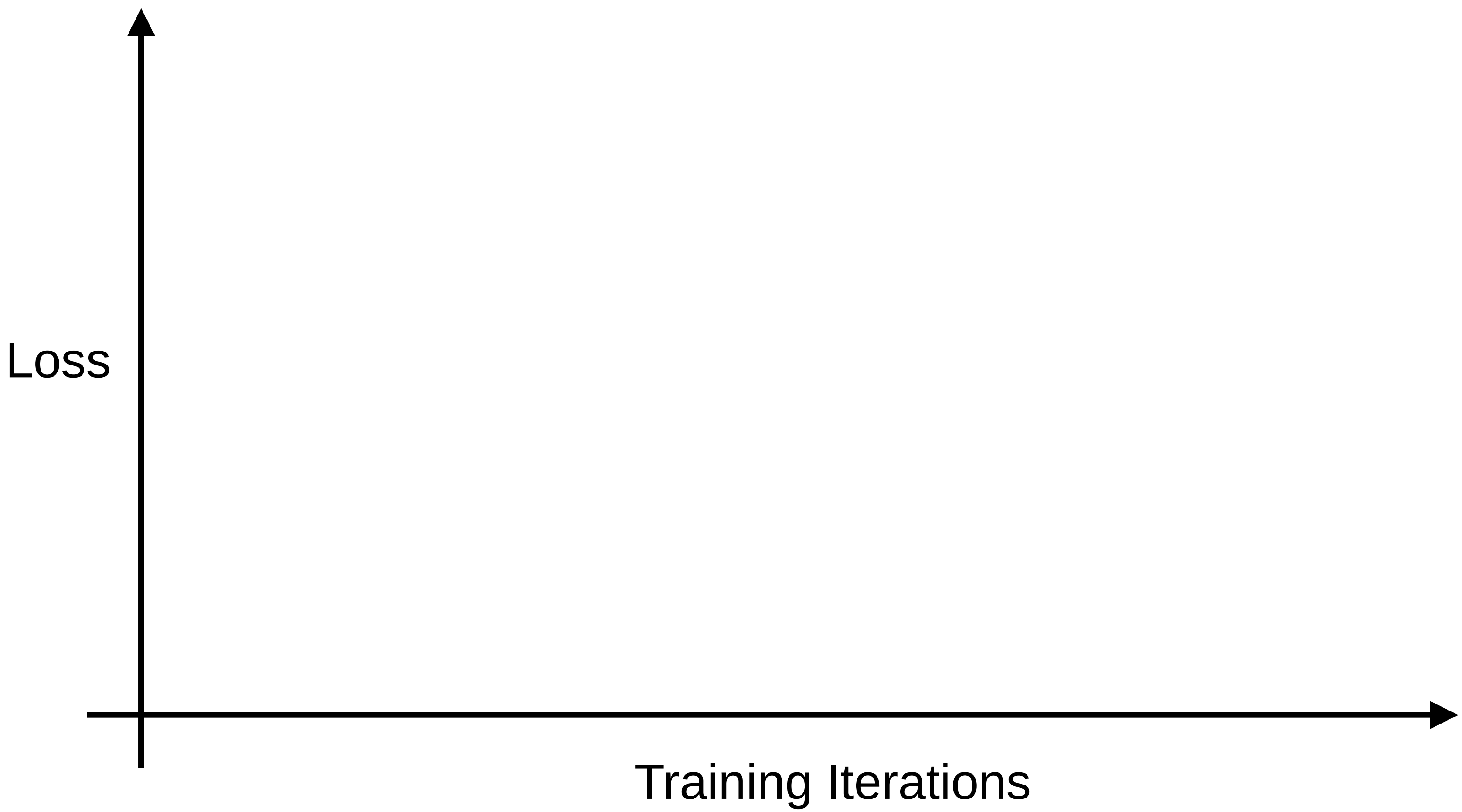
- During training, randomly set some activations to 0
  - Typically 'drop' 50% of activations in layer
  - Forces network to not rely on any 1 node

```
tf.keras.layers.Dropout(p=0.5)
```



Regularization 2: Early Stopping

- Stop training before we have a chance to overfit

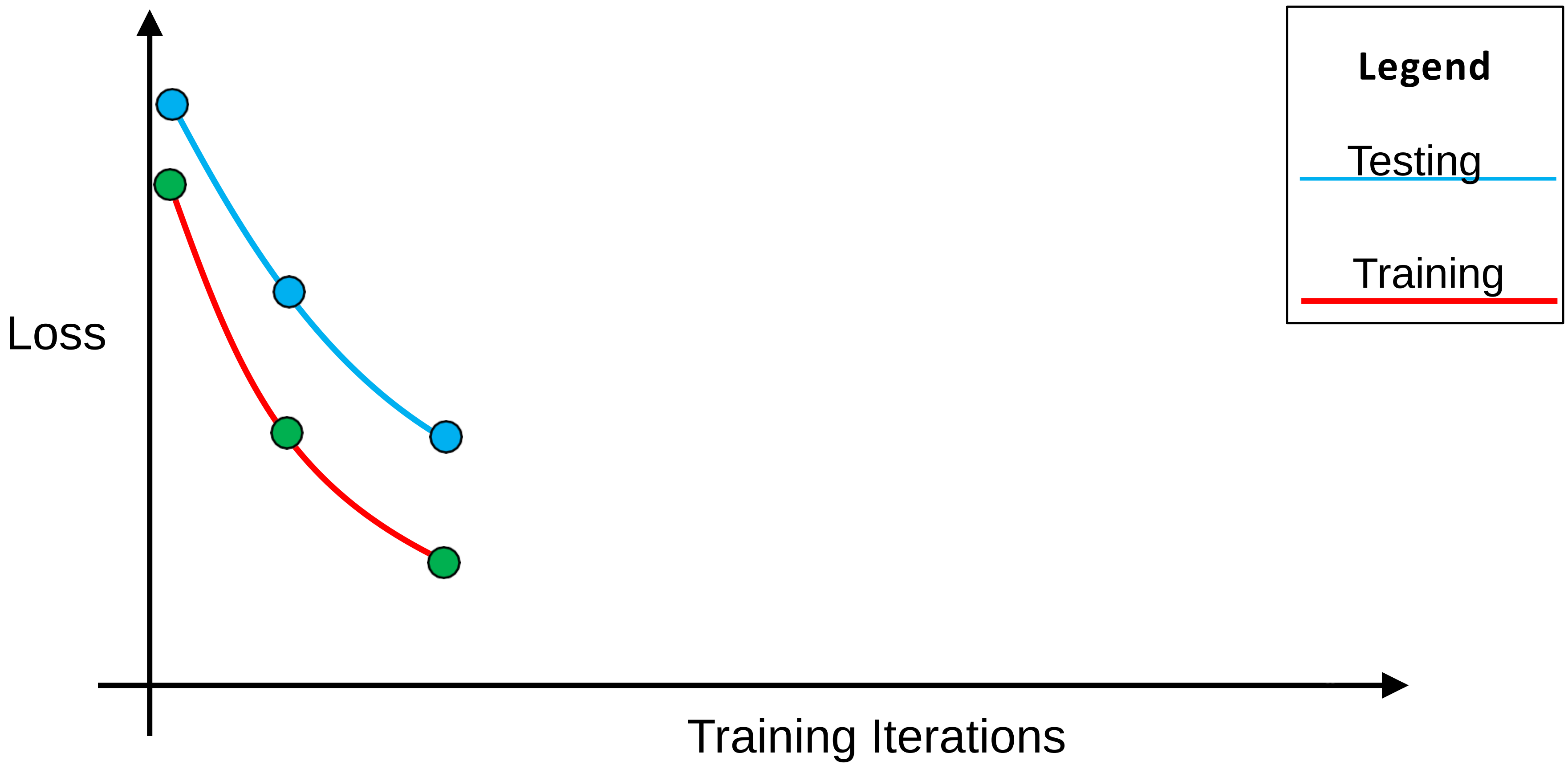


Regularization 2: Early Stopping

- Stop training before we have a chance to overfit

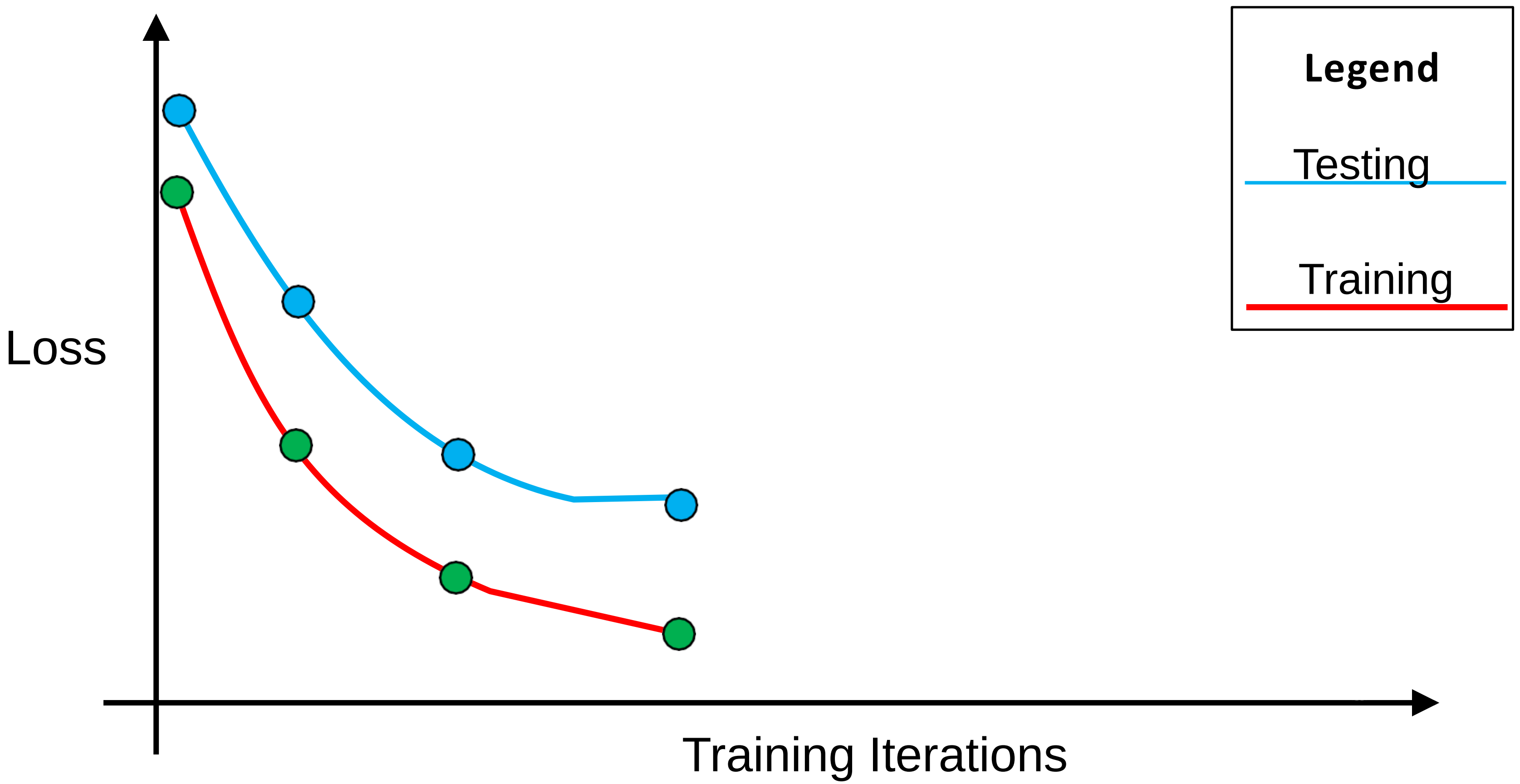
Regularization 2: Early Stopping

- Stop training before we have a chance to overfit



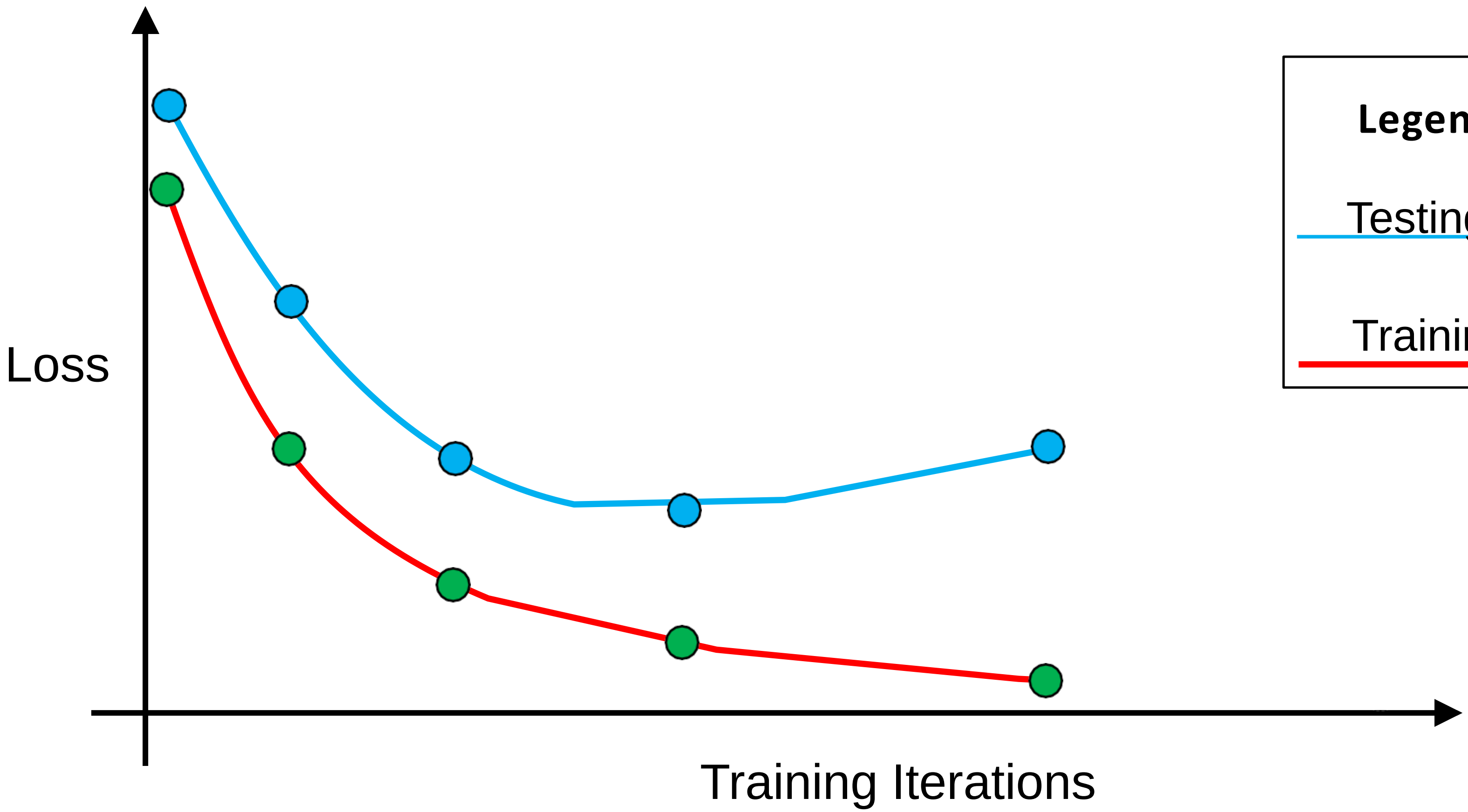
Regularization 2: Early Stopping

- Stop training before we have a chance to overfit



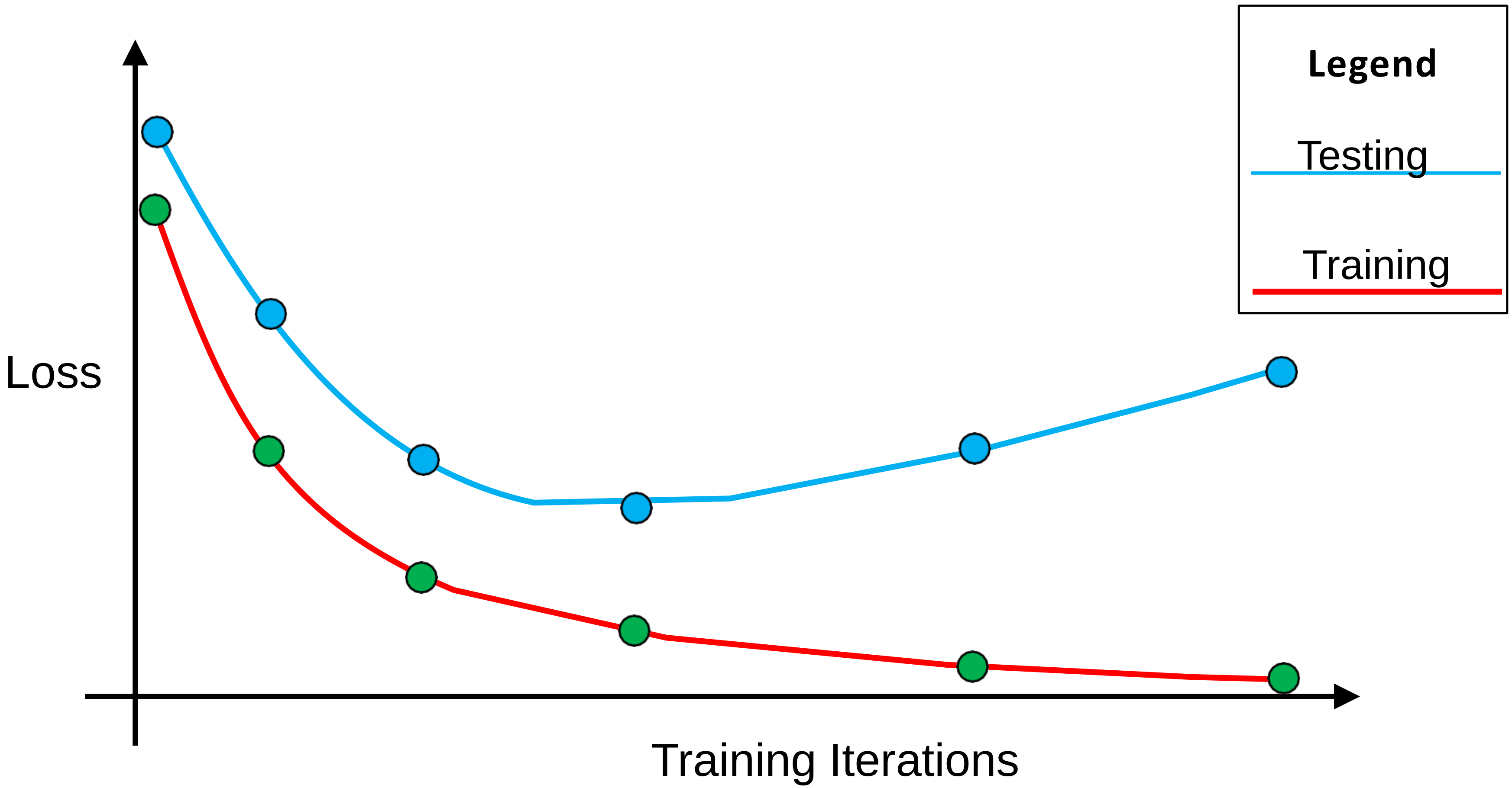
Regularization 2: Early Stopping

- Stop training before we have a chance to overfit



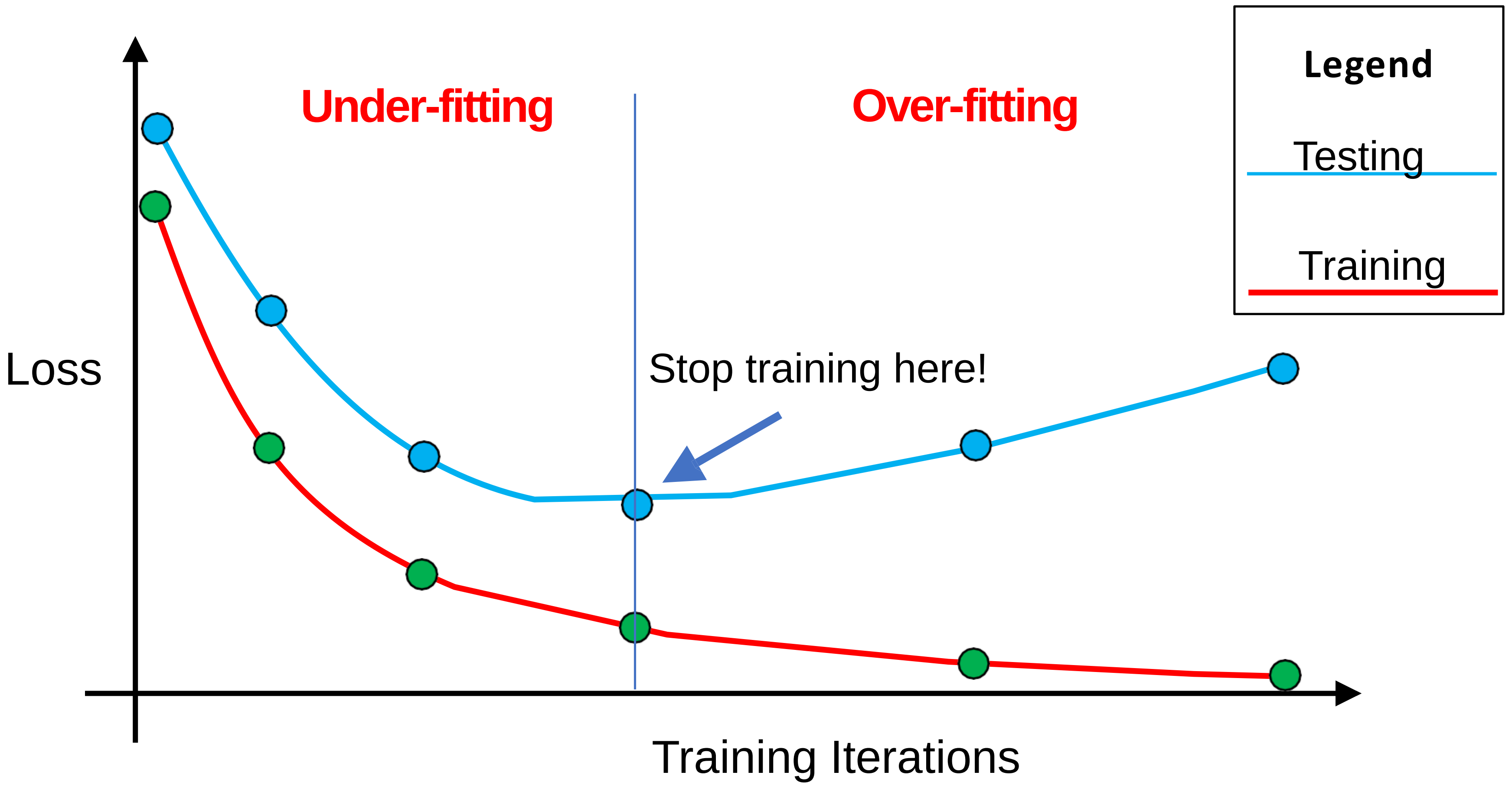
Regularization 2: Early Stopping

- Stop training before we have a chance to overfit



Regularization 2: Early Stopping

- Stop training before we have a chance to overfit



11.5 Learning Rate

Optimization

Remember: Optimization through gradient descent

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

Optimization

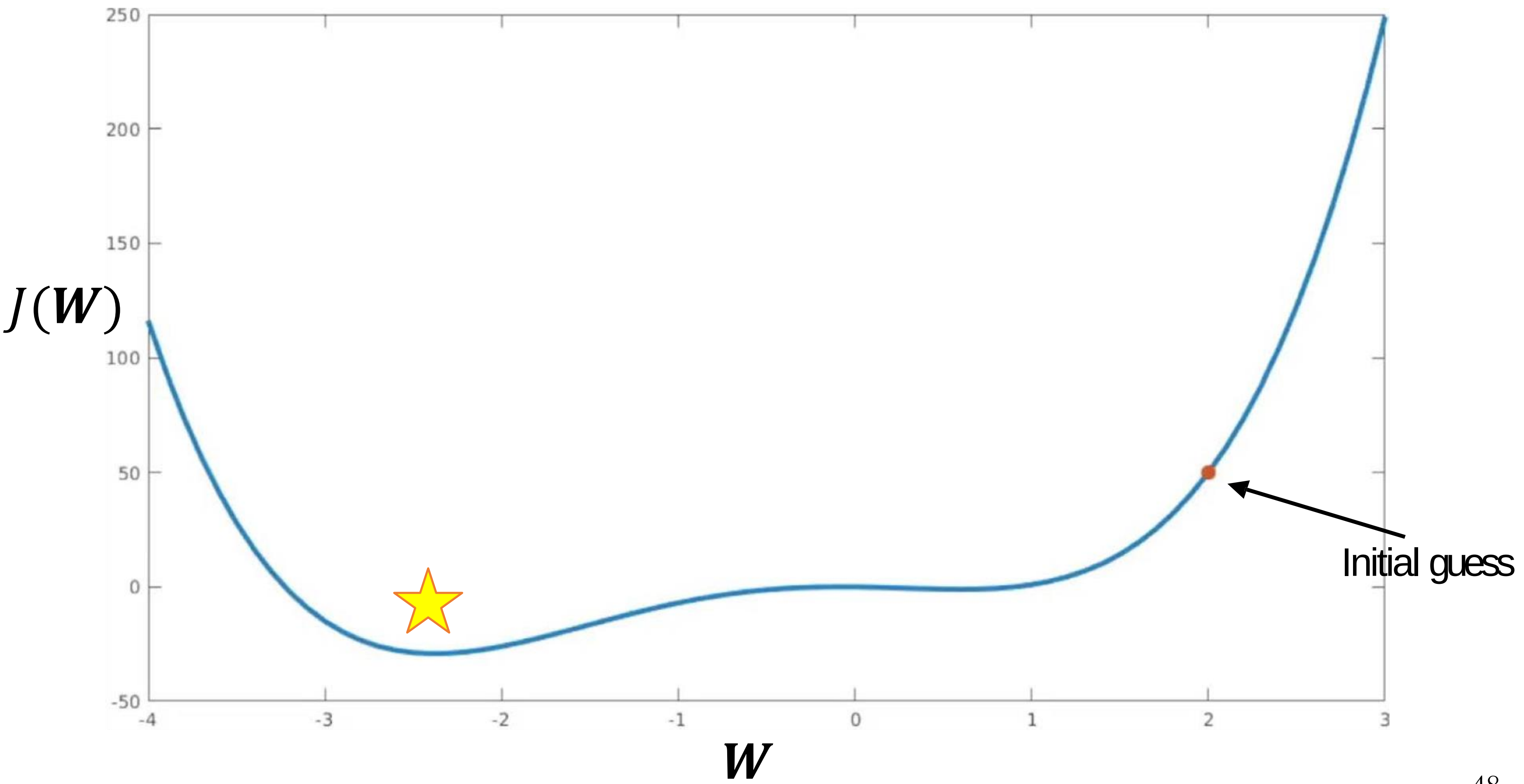
Remember: Optimization through gradient descent

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

How can we set the learning rate?

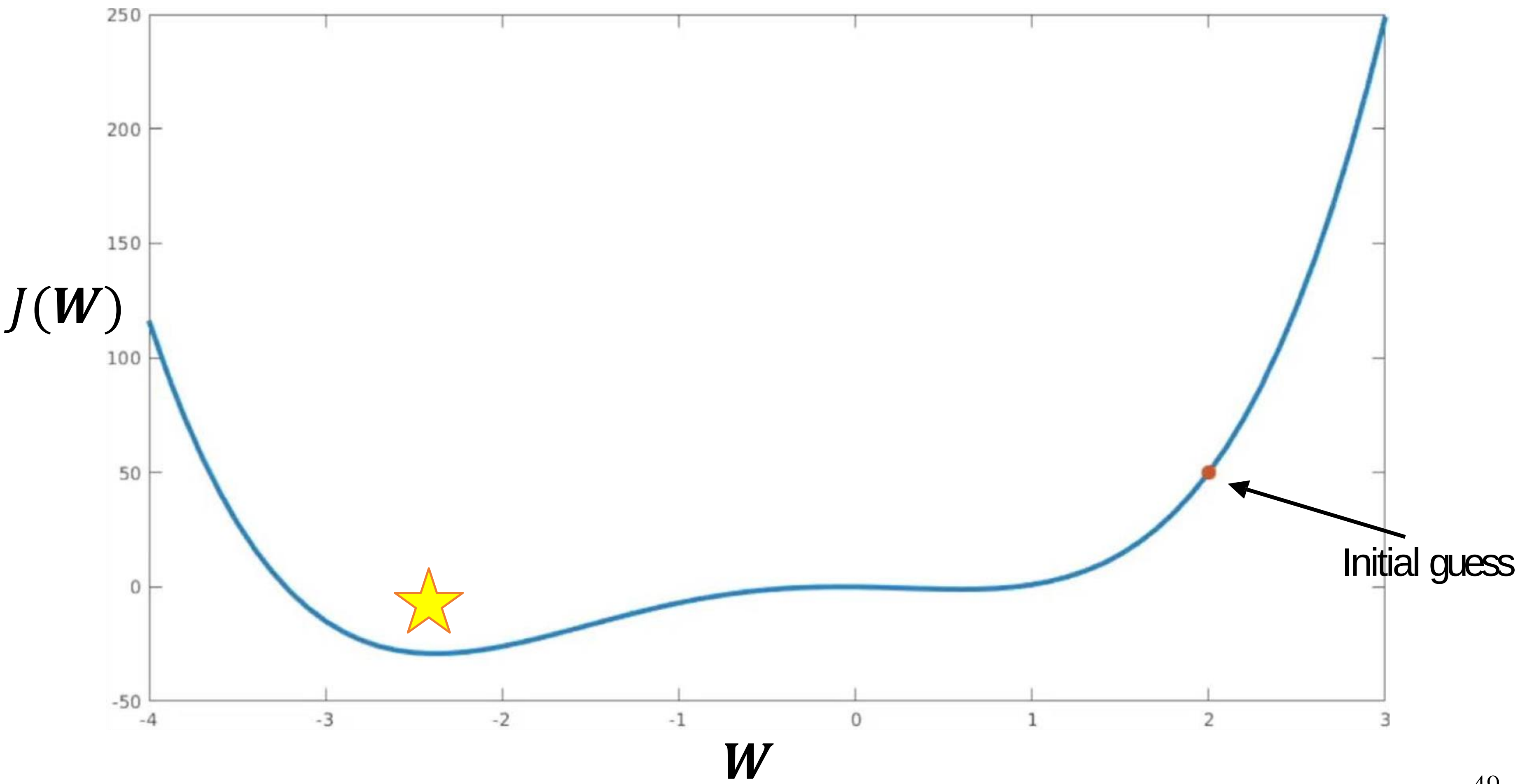
Learning Rate

Small learning rate converges slowly and gets stuck in false local minima



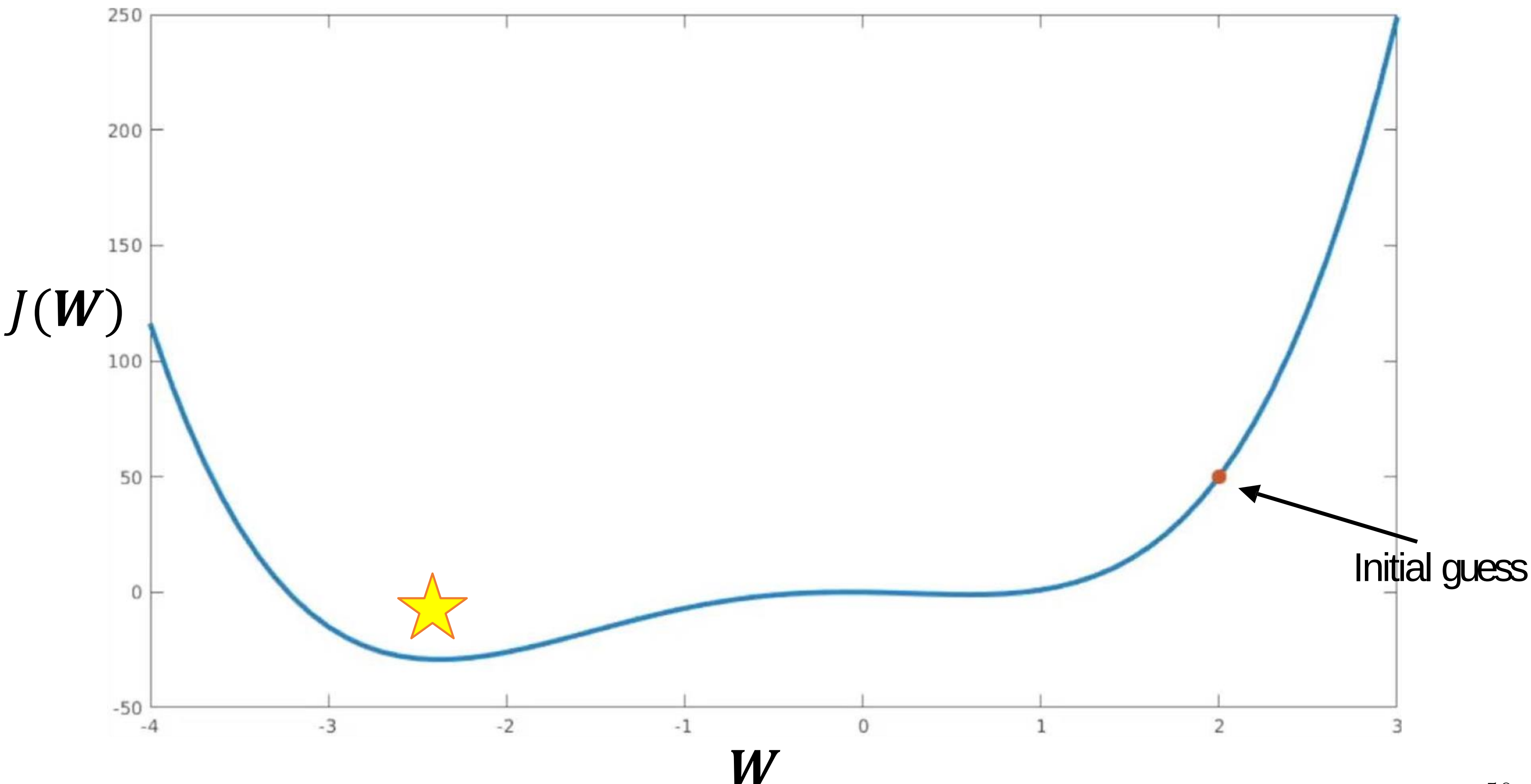
Learning Rate

Large learning rates overshoot, become unstable and diverge



Learning Rate

Stable learning rates converge smoothly and avoid local minima



How to deal with this?

Idea 1:

Try lots of different learning rates and see what works “just right”

How to deal with this?

Idea 2:

Do something smarter!

Design an adaptive learning rate that “adapts” to the landscape

Optimization

Adaptive Learning Rates

- Learning rates are no longer fixed
- Can be made larger or smaller depending on:
  - how large gradient is
  - how fast learning is happening
  - size of particular weights
  - etc...

Optimization

Gradient Descent Algorithms

Algorithm	TF Implementation
• SGD	tf.keras.optimizers.SGD
• Adam	tf.keras.optimizers.Adam
• Adadelta	tf.keras.optimizers.Adadelta
• Adagrad	tf.keras.optimizers.Adagrad
• RMSProp	tf.keras.optimizers.RMSProp

11.6 Gradient Descent Optimization Algorithms

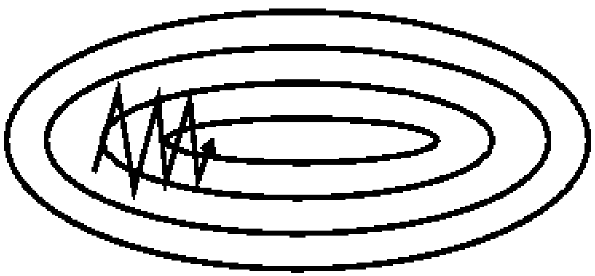
Gradient Descent Optimization Algorithms

1. Momentum
2. Nesterov accelerated gradient
3. Adagrad
4. Adadelta
5. RMSprop
6. Adam
7. Adam extensions

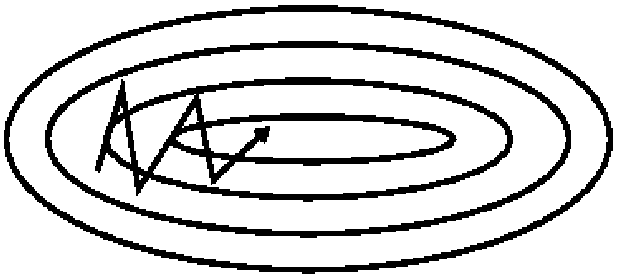
Momentum

- SGD has trouble navigating **ravines**.
- Momentum [Qian, 1999] helps SGD **accelerate**.
- Adds a fraction γ of the update vector of the past step v_{t-1} to current update vector v_t . Momentum term γ is usually set to 0.9.

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta)$$
$$\theta = \theta - v_t$$



(a) SGD without momentum



(b) SGD with momentum

Figure: Source: Genevieve B. Orr

Momentum

- **Reduces updates** for dimensions whose gradients **change directions**.
- **Increases updates** for dimensions whose gradients **point in the same directions**.

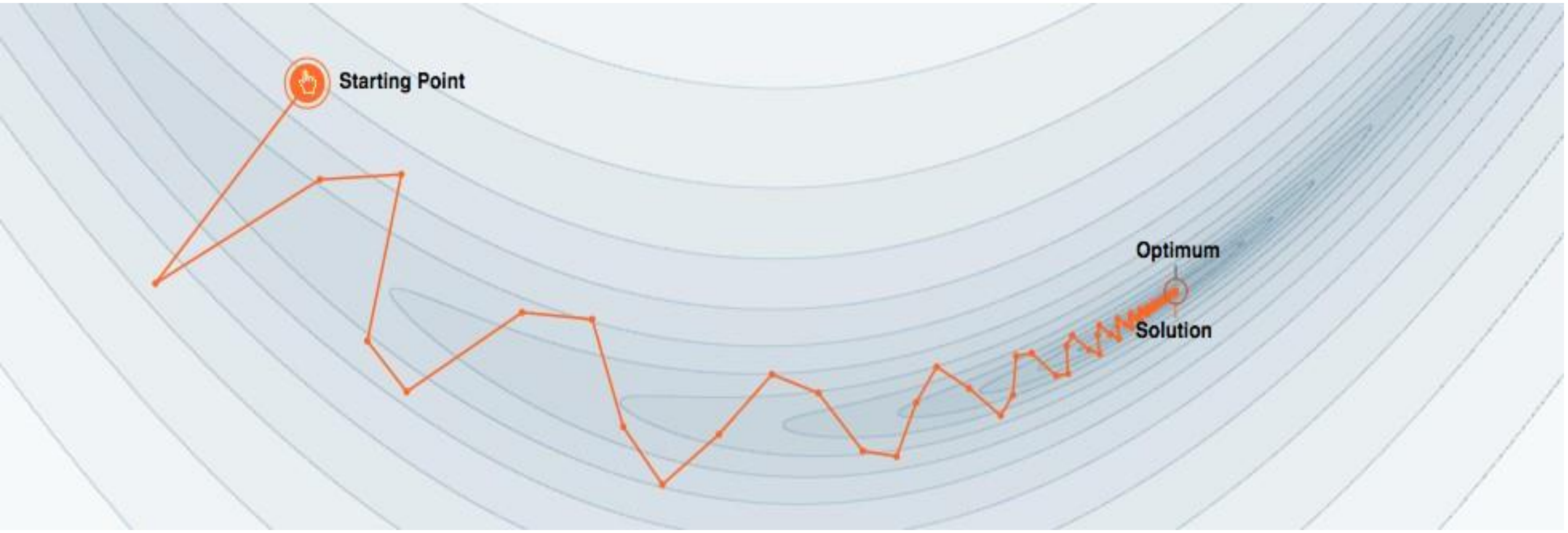


Figure: Optimization with momentum (Source: distill.pub)

Nesterov accelerated gradient

- Momentum **blindly accelerates** down slopes: First computes gradient, then makes a big jump.
- Nesterov accelerated gradient (NAG) [Nesterov, 1983] first makes a big jump in the direction of the previous accumulated gradient $\vartheta - \gamma v_{t-1}$. Then measures where it ends up and makes a correction, resulting in the complete update vector.

$$v_t = \gamma\, v_{t-1} + \eta \nabla_\theta J(\theta - \gamma v_{t-1})$$
$$\theta = \theta - v_t$$

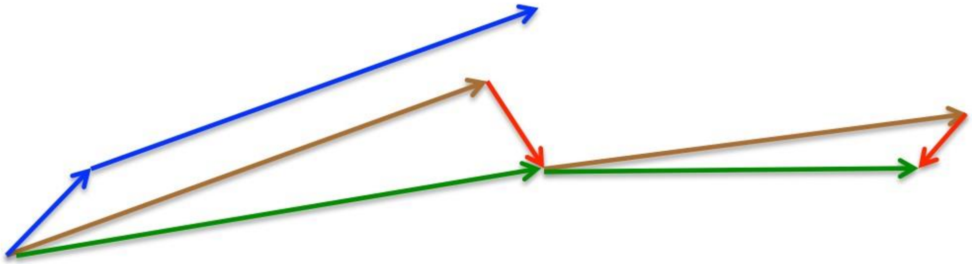


Figure: Nesterov update (Source: G. Hinton's lecture 6c)

Adagrad

- Previous methods: **Same learning rate** η for all parameters θ**.**
- Adagrad [Duchi et al., 2011] **adapts** the learning rate to the parameters (**large** updates for **infrequent** parameters, **small** updates for **frequent** parameters).
- SGD update: $\theta_{t+1} = \theta_t - \eta \cdot g_t$
  - $g_t = \nabla_{\theta_t} J(\theta_t)$
- Adagrad divides the learning rate by the **square root of the sum of squares of historic gradients**.

Adagrad

- Adagrad update:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t$$

- $G_t \in \mathbb{R}^{d \times d}$: diagonal matrix where each diagonal element i, i is the sum of the squares of the gradients w.r.t. θ_i up to time step t
- ϵ: smoothing term to avoid division by zero
- $\odot$: element-wise multiplication

Adagrad

- **Pros**
  - Well-suited for dealing with **sparse data**.
  - Significantly **improves robustness** of SGD.
  - Lesser need to manually tune learning rate.

- **Cons**
  - **Accumulates squared gradients** in denominator. Causes the learning rate to **shrink** and become **infinitesimally small**.

Adadelta

- Adadelta [Zeiler, 2012] restricts the window of accumulated past gradients to a **fixed size**. SGD update:

$$\Delta\theta_t = -\eta \cdot g_t$$
$$\theta_{t+1} = \theta_t + \Delta\theta_t$$

- Defines **running average** of squared gradients $E[g^2]_t$ at time t:

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma)g_t^2$$

  - γ: fraction similarly to momentum term, around 0.9
- Adagrad update:

$$\Delta\theta_t = -\frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t$$

- Preliminary Adadelta update:

$$\Delta\theta_t = -\frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$$

Adadelta

$$\Delta\theta_t = -\frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$$

- Denominator is just root mean squared (RMS) error of gradient:

$$\Delta\theta_t = -\frac{\eta}{RMS[g]_t} g_t$$

- Define **running average of squared parameter updates** and RMS:

$$E[\Delta\theta^2]_t = \gamma E[\Delta\theta^2]_{t-1} + (1-\gamma)\Delta\theta_t^2$$
$$RMS[\Delta\theta]_t = \sqrt{E[\Delta\theta^2]_t + \epsilon}$$

- Approximate with $RMS[\Delta\theta]_{t-1}$, replace η for **final Adadelta update**:

$$\Delta\theta_t = -\frac{RMS[\Delta\theta]_{t-1}}{RMS[g]_t} g_t$$
$$\theta_{t+1} = \theta_t + \Delta\theta_t$$

Adam

- Adaptive Moment Estimation (Adam) [Kingma and Ba, 2015] also stores **running average of past squared gradients** v_t like Adadelta and RMSprop.
- Like Momentum, stores **running average of past gradients** m_t.

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$
$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

  - m_t: first moment (mean) of gradients
  - v_t: second moment (uncentered variance) of gradients
  - β_1, β_2: decay rates

Adam

- m_t and v_t are initialized as 0-vectors. For this reason, they are biased towards 0.
- Compute bias-corrected first and second moment estimates:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$
$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

- Adam update rule:

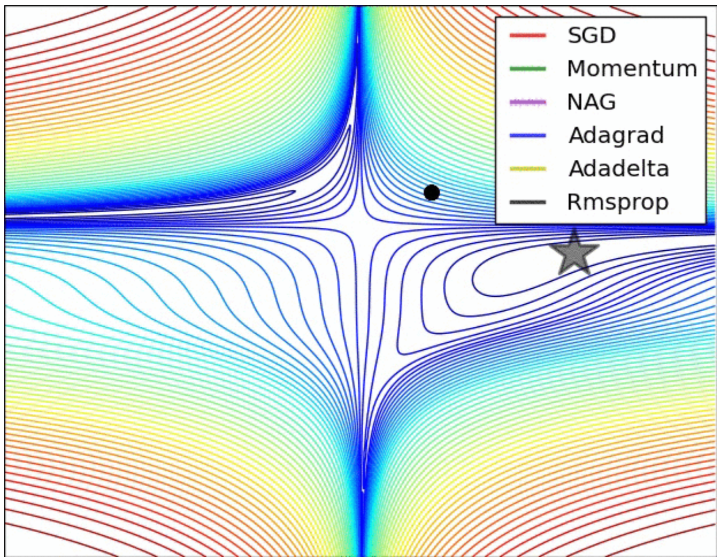
$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t$$

Update equations

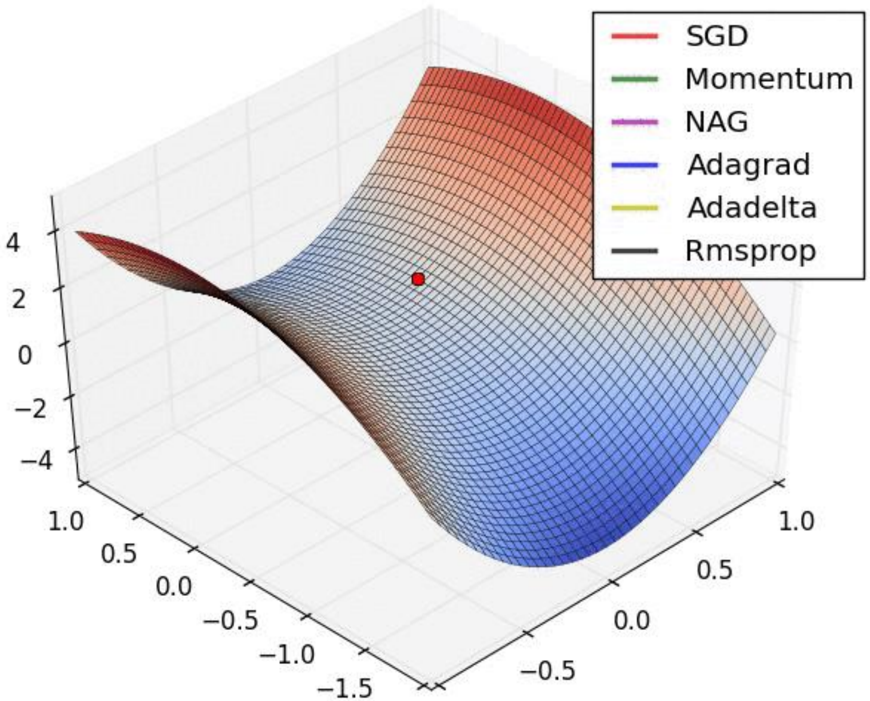
Method	Update equation
SGD	$g_t = \nabla_{\theta_t} J(\theta_t)$ $\Delta\theta_t = -\eta \cdot g_t$ $\theta_t = \theta_t + \Delta\theta_t$
Momentum	$\Delta\theta_t = -\gamma\, v_{t-1} - \eta g_t$
NAG	$\Delta\theta_t = -\gamma\, v_{t-1} - \eta \nabla_\theta J(\theta - \gamma v_{t-1})$
Adagrad	$\Delta\theta_t = -\frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t$
Adadelta	$\Delta\theta_t = -\frac{RMS[\Delta\theta]_{t-1}}{RMS[g]_t} g_t$
RMSprop	$\Delta\theta_t = -\frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$
Adam	$\Delta\theta_t = -\frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t$

Table: Update equations for the gradient descent optimization algorithms.

Visualization of algorithms



(a) SGD optimization on loss surface contours



(b) SGD optimization on saddle point

Figure: Source and full animations: Alec Radford

Which optimizer to choose?

- Adaptive learning rate methods (Adagrad, Adadelta, RMSprop, Adam) are **particularly useful for sparse features**.
- Adagrad, Adadelta, RMSprop, and Adam work well in similar circumstances.
- [Kingma and Ba, 2015] show that bias-correction helps Adam **slightly outperform RMSprop**.

References

- **MIT 6.S191 Introduction to Deep Learning**
  - http://introtodeeplearning.com/
- **Dive into Deep Learning**
  - https://d2l.ai/chapter_optimization/optimization-intro.html
- **Optimization for Deep Learning**

  -Sebastian Ruder

Thank you!

权小军 中山大学数据科学与计算机学院