الجامعة السورية الخاصة
SYRIAN PRIVATE UNIVERSITY

Topic 2

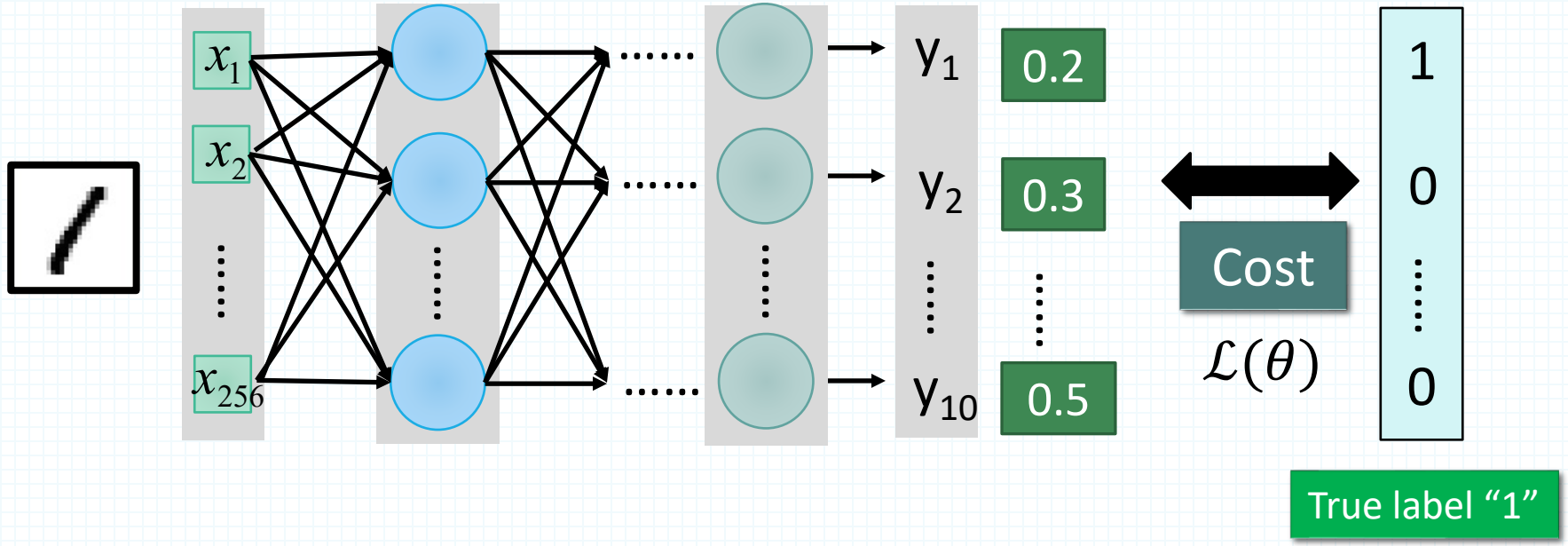
Deep Learning

Optimization in Deep Neural Networks

Dr. Riad Sonbol

Training NNs

- The network ***parameters*** θ include $\theta = \{W^1, b^1, W^2, b^2, \cdots W^L, b^L\}$
- Define a ***loss function***/objective function/cost function $\mathcal{L}(\theta)$ that calculates the **difference (error) between the model prediction and the true label**



Loss Functions

- ***Classification tasks***

Cross-entropy

$$\mathcal{L}(\theta) = -\frac{1}{N}\sum_{i=1}^{N}\sum_{k=1}^{K}\left[y_k^{(i)}\log\hat{y}_k^{(i)} + \left(1-y_k^{(i)}\right)\log\left(1-\hat{y}_k^{(i)}\right)\right]$$

Ground-truth class labels y_i and model predicted class labels $\hat{y}_i$

- ***Regression tasks***

Mean Squared Error

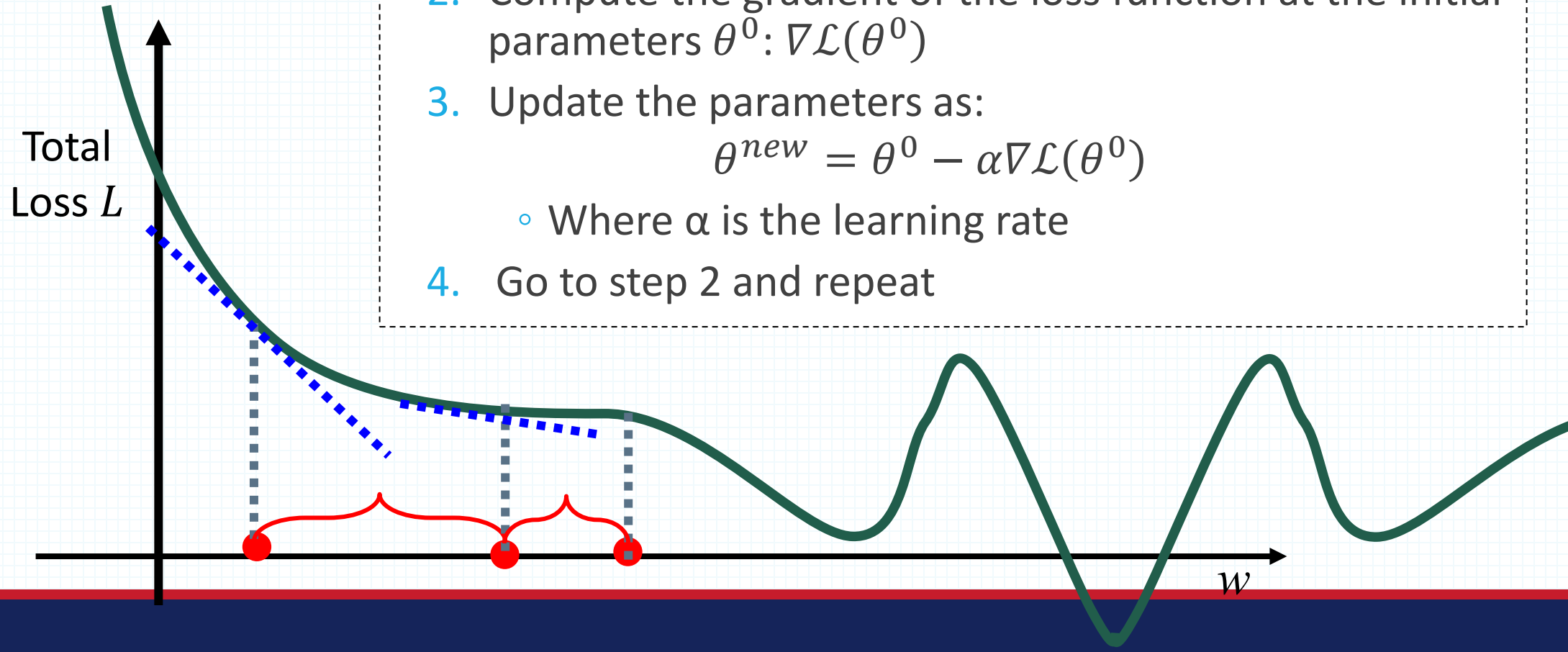
$$\mathcal{L}(\theta) = \frac{1}{n}\sum_{i=1}^{n}\left(y^{(i)} - \hat{y}^{(i)}\right)^2$$

Mean Absolute Error

$$\mathcal{L}(\theta) = \frac{1}{n}\sum_{i=1}^{n}\left|y^{(i)} - \hat{y}^{(i)}\right|$$

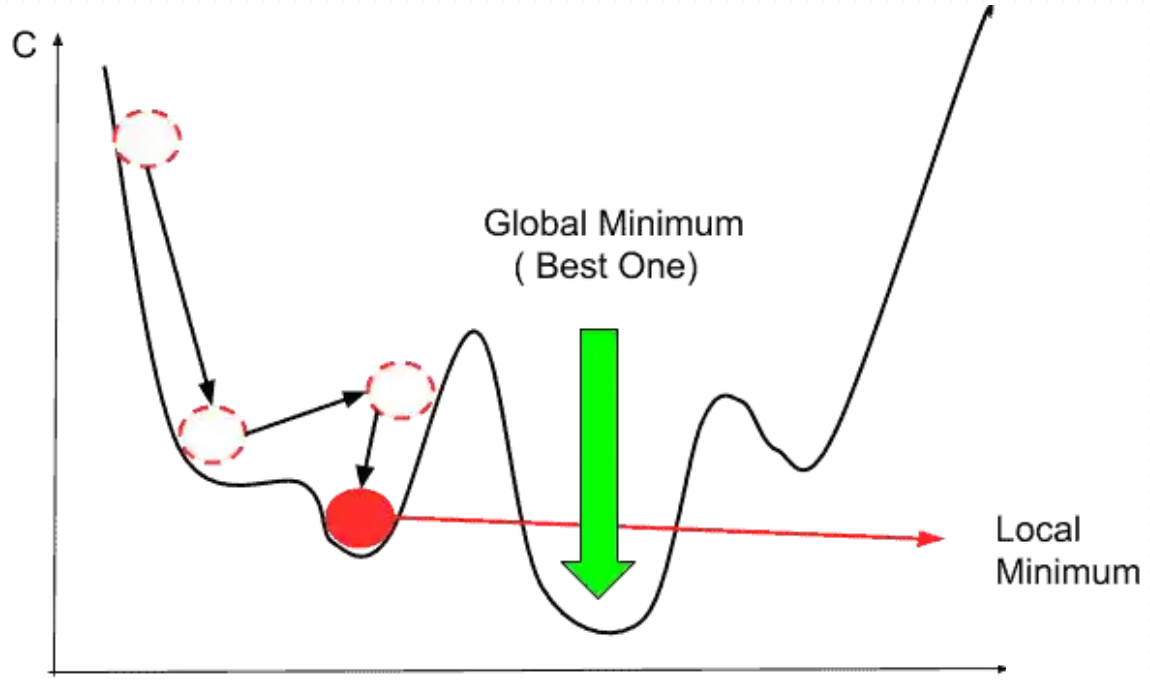
Gradient Descent Algorithm

- Steps in the ***gradient descent algorithm***:
  1. Randomly initialize the model parameters, θ^0
  2. Compute the gradient of the loss function at the initial parameters θ^0: $\nabla\mathcal{L}(\theta^0)$
  3. Update the parameters as:
     $$\theta^{new} = \theta^0 - \alpha\nabla\mathcal{L}(\theta^0)$$
     - Where α is the learning rate
  4. Go to step 2 and repeat



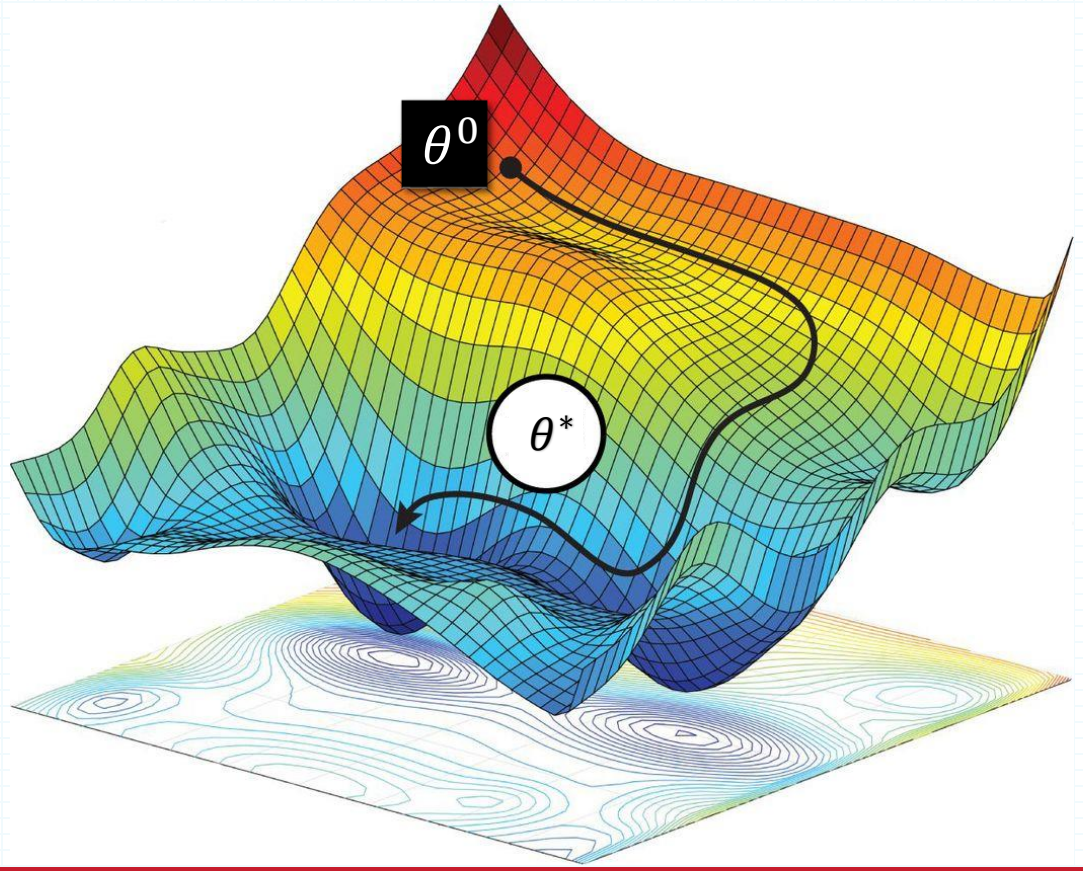
Gradient Descent Algorithm

- Gradient descent algorithm stops when a local minimum of the loss surface is reached
  - GD does not guarantee reaching a global minimum
  - However, empirical evidence suggests that GD works well for NNs



Gradient Descent Algorithm

- Example: a NN with only 2 parameters w_1 and w_2, i.e., $\theta = \{w_1, w_2\}$
  - The different colors represent the values of the loss (minimum loss θ^* is ≈ 1.3)



$$\theta^{new} = \theta^0 - \alpha \nabla \mathcal{L}(\theta^0)$$

$$\nabla \mathcal{L}(\theta^0) = \begin{bmatrix} \partial \mathcal{L}(\theta^0)/\partial w_1 \\ \partial \mathcal{L}(\theta^0)/\partial w_2 \end{bmatrix}$$

Backpropagation

- Forward propagation (forward pass) refers to passing the inputs x through the hidden layers to obtain the model outputs (predictions) y
- The loss $\mathcal{L}(y, \hat{y})$ function is then calculated
- Backpropagation traverses the network in reverse order, from the outputs y backward toward the inputs x to calculate the gradients of the loss $\nabla\mathcal{L}(\theta)$
- Each update of the model parameters θ during training takes one forward and one backward pass (e.g., of a batch of inputs)

Optimization in Deep Neural Networks:

More Advanced Concepts

Problem 1

- It is **wasteful** to compute the loss over the entire training dataset to perform a single parameter update for large datasets
  - E.g., ImageNet has 14M images

$$w_i \leftarrow w_i + \Delta w_i$$

Where,

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i}$$

and

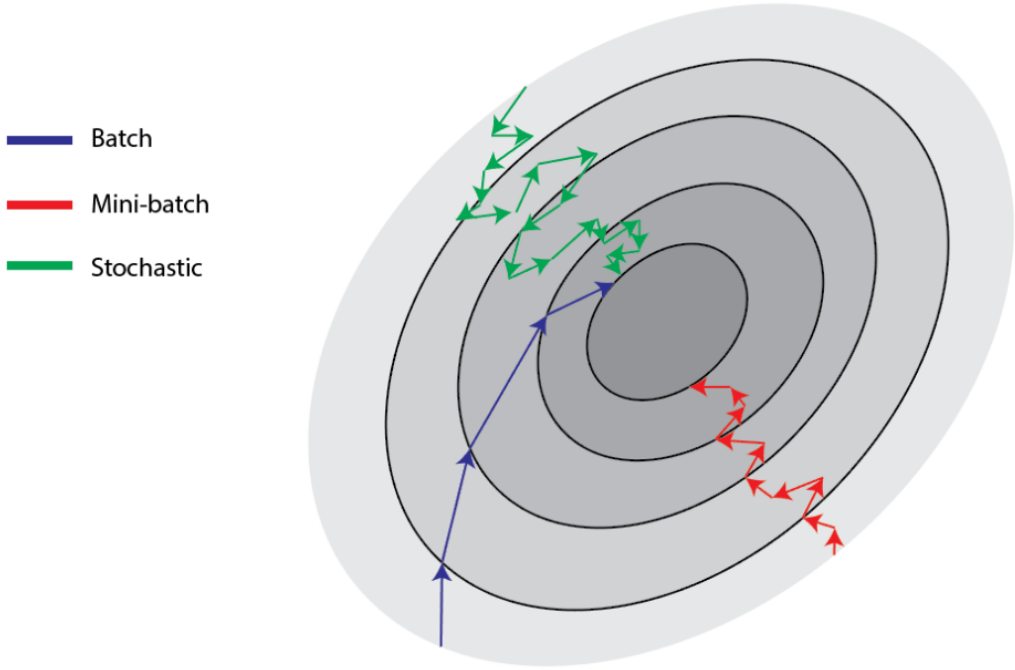
$$\frac{\partial E}{\partial w_i} = \frac{\partial}{\partial w_i} \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$$

Mini-batch Gradient Descent

- Therefore, GD (a.k.a. vanilla or standard GD) is almost always replaced with mini-batch GD
- ***Mini-batch gradient descent***
  - Approach:
    - Compute the loss $\mathcal{L}(\theta)$ on a mini-batch of data, update the parameters θ, and repeat until all data are used
    - At the next epoch, shuffle the training data, and repeat the above process
  - Mini-batch GD results in much faster training
    - Example: 32 to 256 images
  - It works because the gradient from a mini-batch is a good **approximation** of the gradient from the entire training set

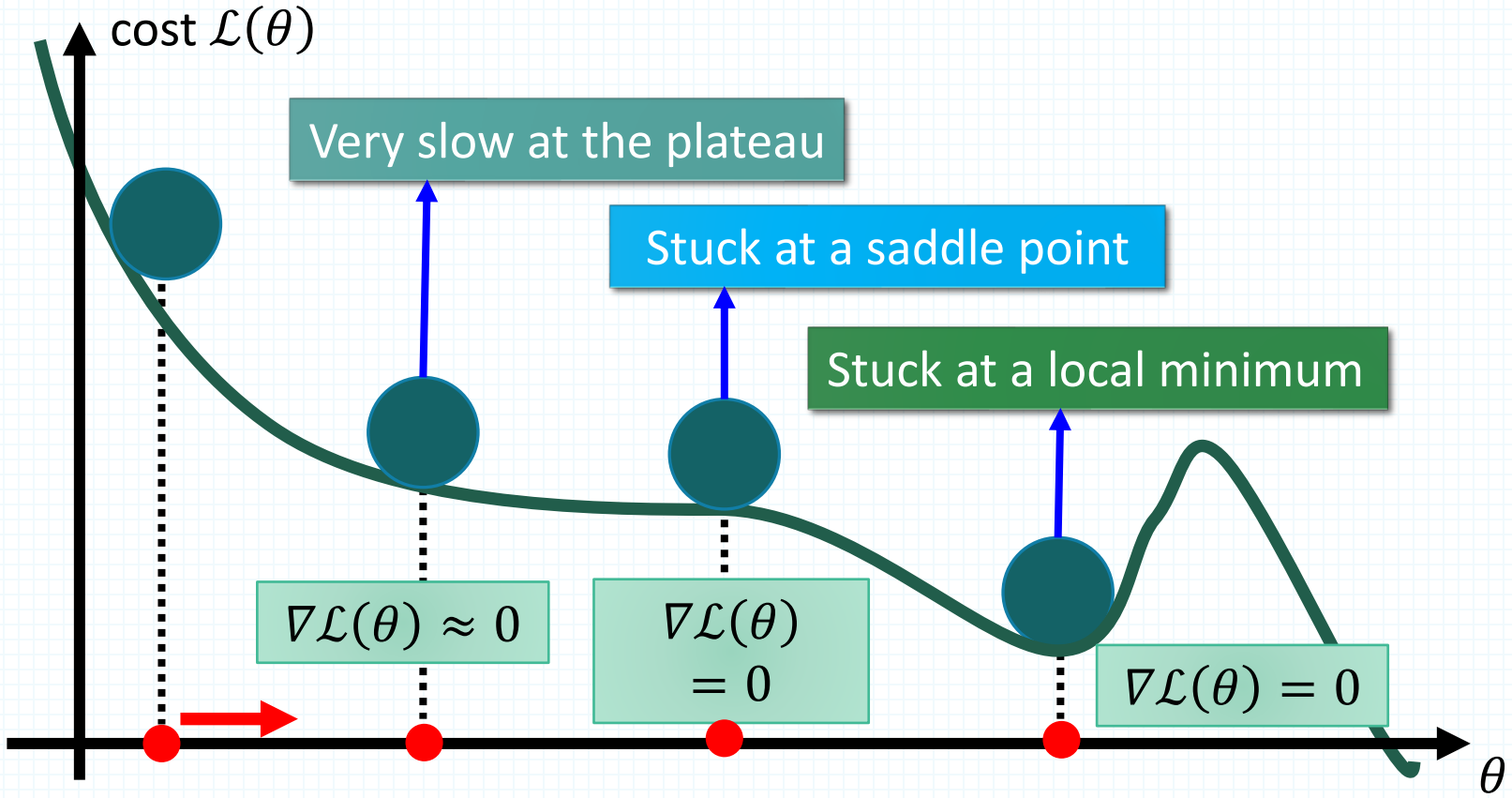
Stochastic Gradient Descent

- ***Stochastic gradient descent***
  - SGD uses mini-batches that consist of a single input example
    - E.g., one image mini-batch
  - Although this method is very fast, it may cause significant instabilities in the loss function
    - Therefore, it is less commonly used, and mini-batch GD is preferred
  - In most DL libraries, SGD typically means a mini-batch GD (with an option to add momentum).



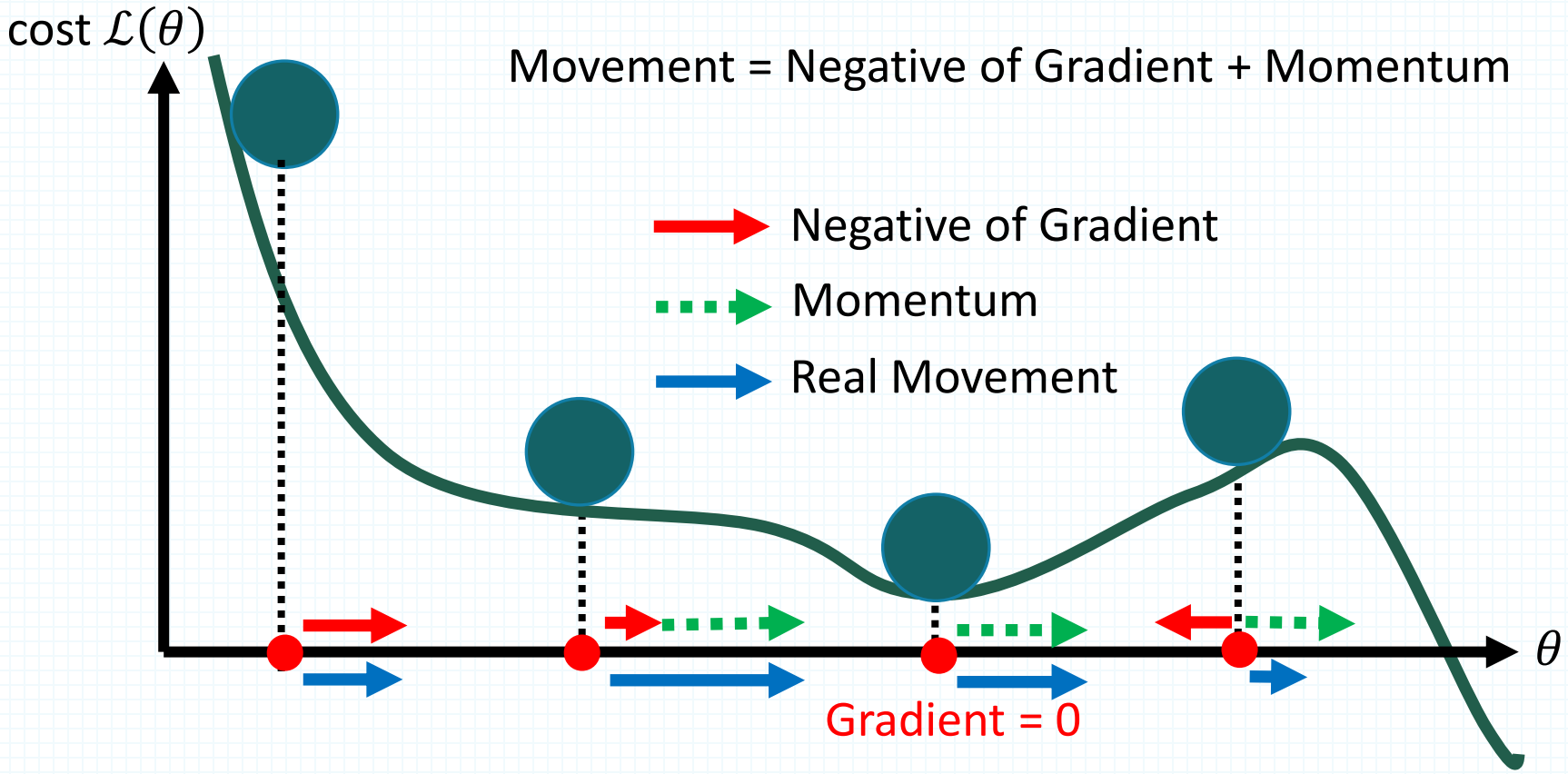
Problem 2

- Besides the local minima problem, the GD algorithm can be very slow at plateaus, and it can get stuck at saddle points



Gradient Descent with Momentum

- ***Gradient descent with momentum*** uses the momentum of the gradient for parameter optimization



Gradient Descent with Momentum

- Parameters update in ***GD with momentum*** at iteration t: $\theta^t = \theta^{t-1} - V^t$
  - Where: $V^t = \beta V^{t-1} + \alpha \nabla \mathcal{L}(\theta^{t-1})$
  - I.e., $\theta^t = \theta^{t-1} - \alpha \nabla \mathcal{L}(\theta^{t-1}) - \beta V^{t-1}$
- Compare to vanilla GD: $\theta^t = \theta^{t-1} - \alpha \nabla \mathcal{L}(\theta^{t-1})$
  - Where θ^{t-1} are the parameters from the previous iteration $t-1$
- The term V^t is called momentum
  - This term accumulates the gradients from the past several steps, i.e.,

$$V^t = \beta V^{t-1} + \alpha \nabla \mathcal{L}(\theta^{t-1})$$
$$= \beta\left(\beta V^{t-2} + \alpha \nabla \mathcal{L}(\theta^{t-2})\right) + \alpha \nabla \mathcal{L}(\theta^{t-1})$$
$$= \beta^2 V^{t-2} + \beta\alpha \nabla \mathcal{L}(\theta^{t-2}) + \alpha \nabla \mathcal{L}(\theta^{t-1})$$
$$= \beta^3 V^{t-3} + \beta^2 \alpha \nabla \mathcal{L}(\theta^{t-3}) + \beta\alpha \nabla \mathcal{L}(\theta^{t-2}) + \alpha \nabla \mathcal{L}(\theta^{t-1})$$

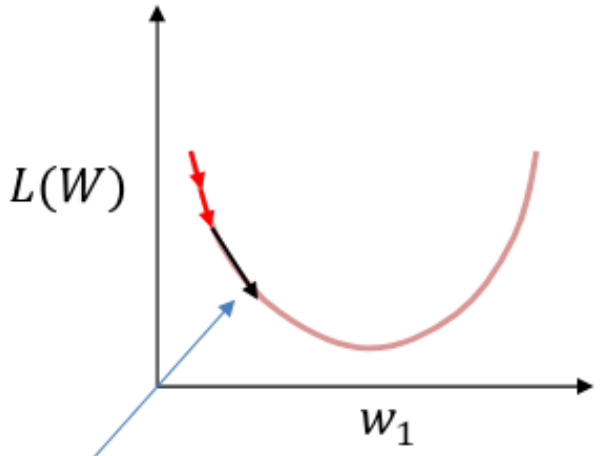
Gradient Descent with Momentum

- Vanilla GD: $\theta^t = \theta^{t-1} - \alpha\nabla\mathcal{L}(\theta^{t-1})$
- ***GD with momentum:*** $\theta^t = \theta^{t-1} - V^t = \theta^{t-1} - (\alpha\nabla\mathcal{L}(\theta^{t-1}) + \beta V^{t-1})$

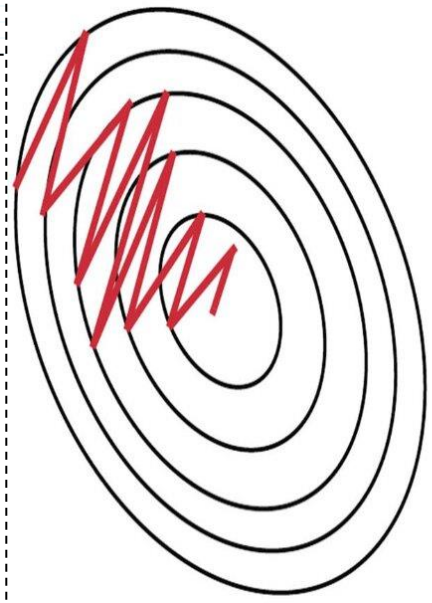
  $V^t = \beta^3 V^{t-3} + \beta^2\alpha\nabla\mathcal{L}(\theta^{t-3}) + \beta\alpha\nabla\mathcal{L}(\theta^{t-2}) + \alpha\nabla\mathcal{L}(\theta^{t-1})$
- This method updates the parameters θ in the direction of the weighted average of the past gradients
- This term ***momentum*** is analogous to a momentum of a heavy ball rolling down the hill
- The parameter β is referred to as a coefficient of momentum
  - A typical value of the parameter β is 0.9

Another reason to use Momentum

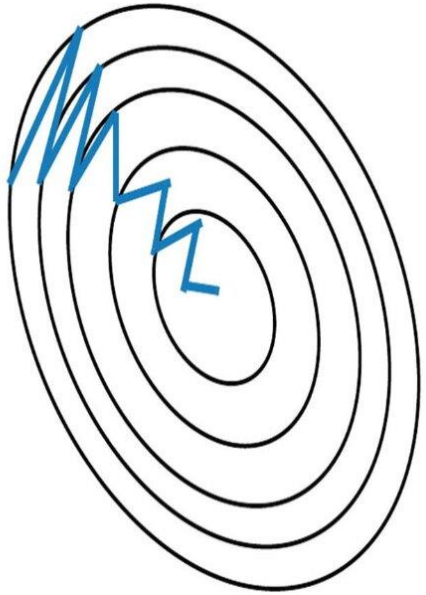
Reducing the oscillations in the optimization path



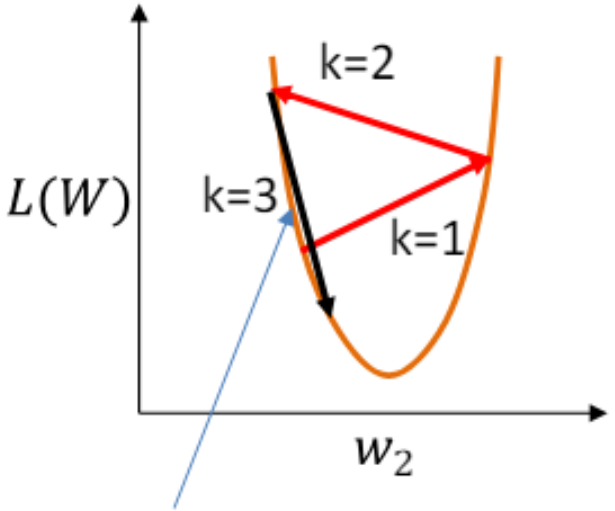
Increase stepsize because previous updates consistently moved weight right



Stochastic Gradient Descent **without** Momentum



Stochastic Gradient Descent **with** Momentum

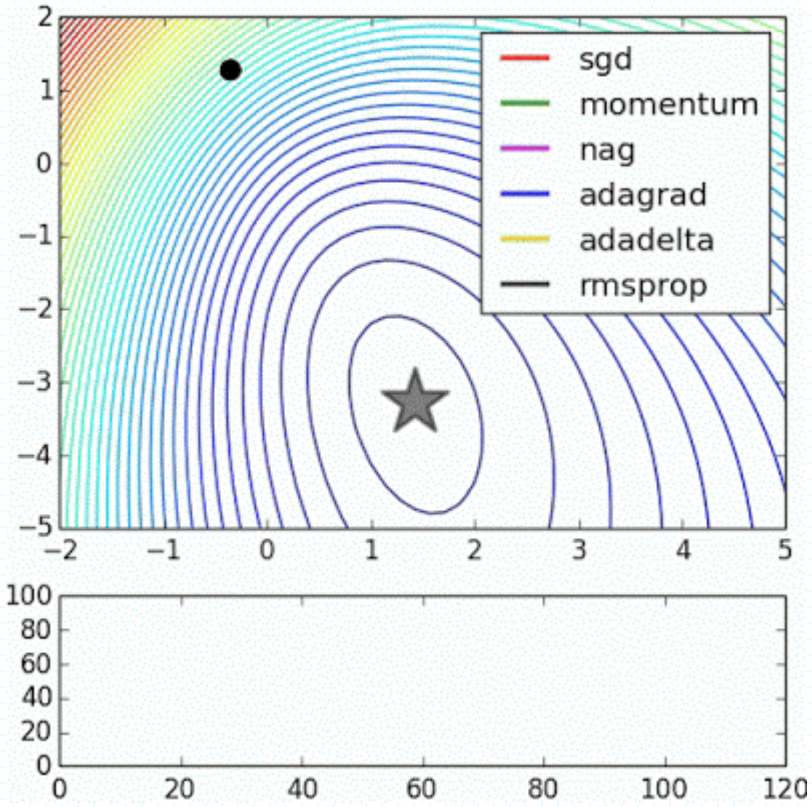


Decrease stepsize because previous updates kept changing direction

- ✓ Shrink step size in directions where the weight oscillates
- ✓ Expand step size in directions where the weight moves consistently in one direction

Adaptive Moment

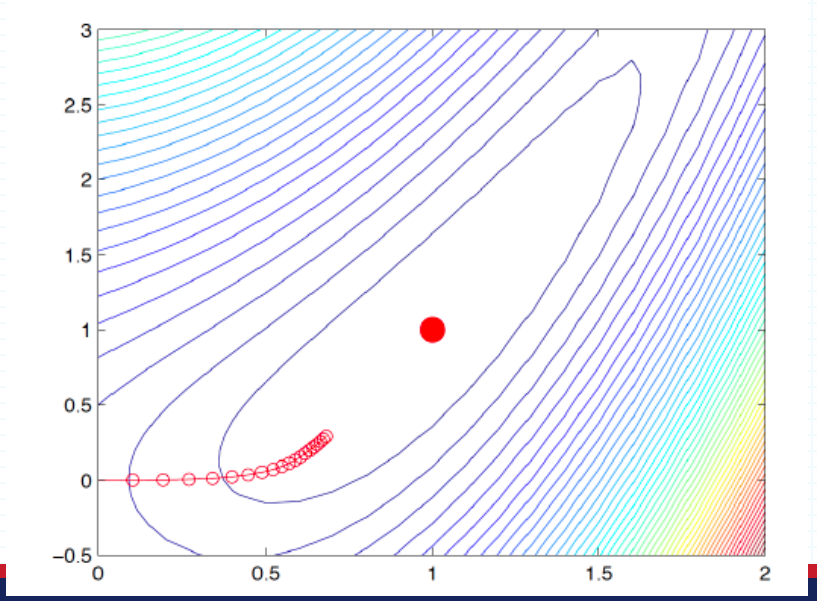
- Storing an exponentially decaying average of past squared gradients s like Adadelta and RMSprop.
- Other commonly used optimization methods include:
  - Adam, Adagrad, Adadelta, RMSprop, Nadam, etc.
  - Most commonly used optimizers nowadays are Adam and SGD with momentum

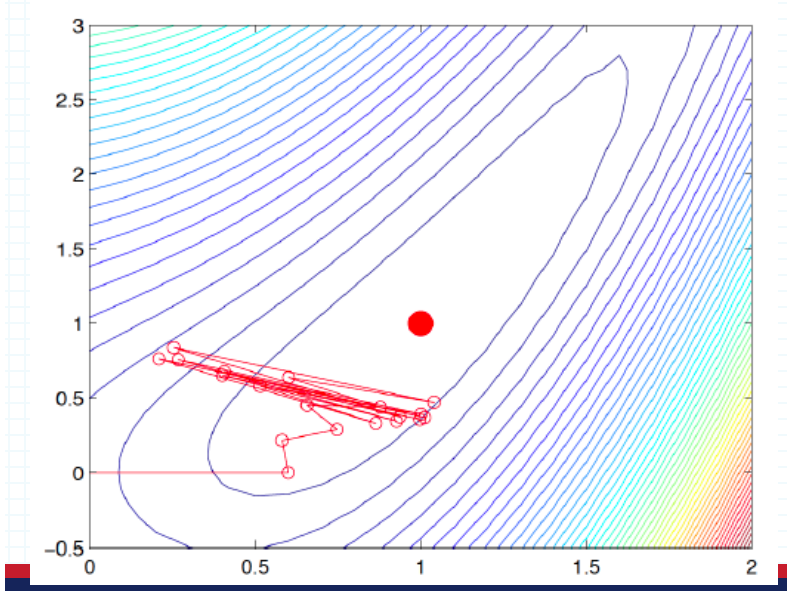


Problem 3:

- ***Learning rate***
  - The gradient tells us the direction in which the loss has the steepest rate of increase, but it does not tell us how far along the opposite direction we should step
  - Choosing the learning rate (also called the step size) is one of the most important hyper-parameter settings for NN training

LR too small

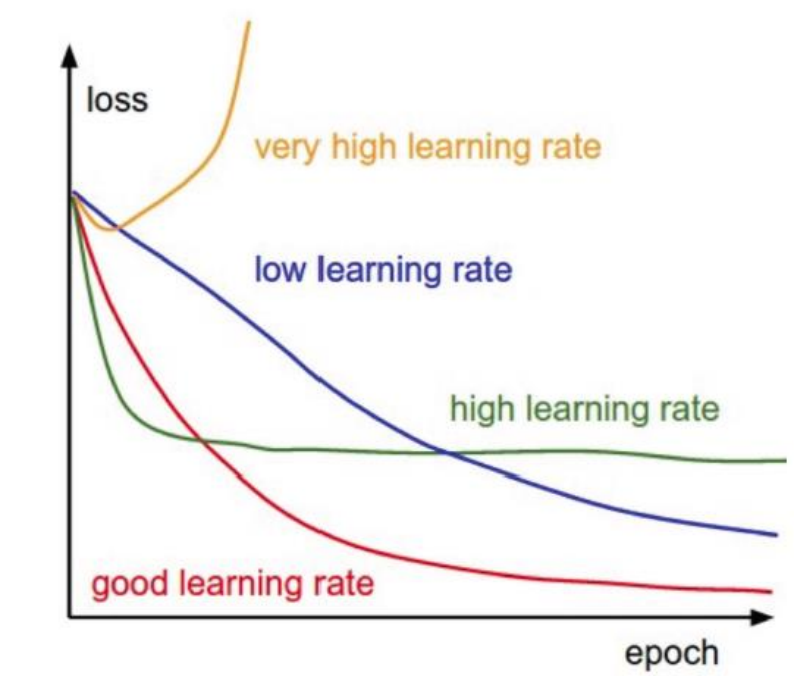




LR too large

Learning Rate

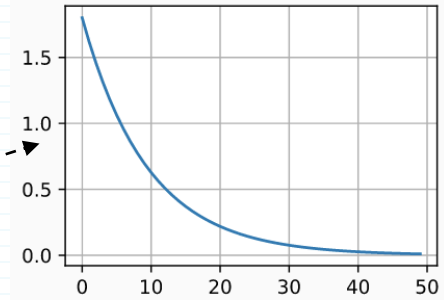
- Training loss for different learning rates
  - High learning rate: the loss increases or plateaus too quickly
  - Low learning rate: the loss decreases too slowly (takes many epochs to reach a solution)



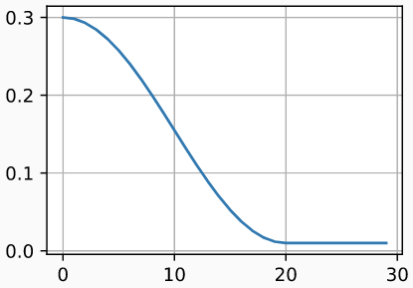
Learning Rate Scheduling

- ***Learning rate scheduling*** is applied to change the values of the learning rate during the training
  - ***Annealing*** is reducing the learning rate over time (a.k.a. learning rate decay)
    - Approach 1: reduce the learning rate by some factor every few epochs
      - Typical values: reduce the learning rate by a half every 5 epochs, or divide by 10 every 20 epochs
    - Approach 2: exponential or cosine decay gradually reduce the learning rate over time
    - Approach 3: reduce the learning rate by a constant (e.g., by half) whenever the validation loss stops improving
  - ***Warmup*** is gradually increasing the learning rate initially, and afterward let it cool down until the end of the training

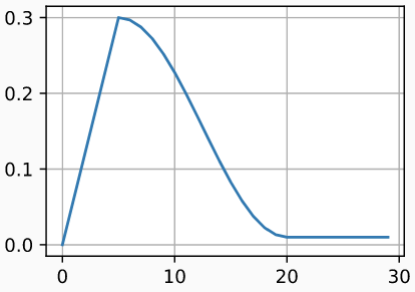
Exponential decay



Cosine decay

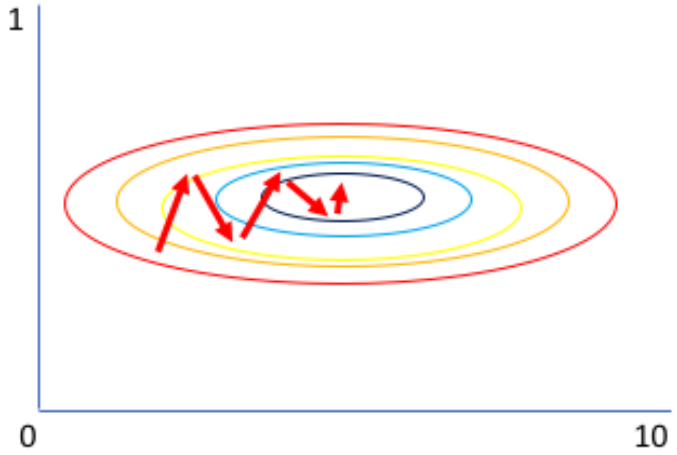


Warmup

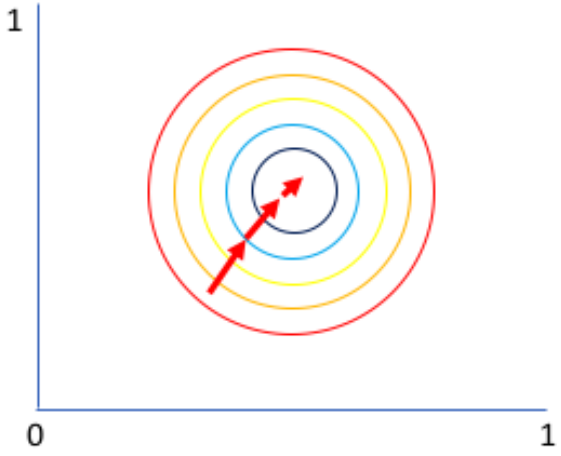


Problem 4: Unnormalized Data

- It can lead toward an awkward loss function topology which places more emphasis on certain parameter gradients.
- Example: The first input value, x1, varies from 0 to 1 while the second input value, x2, varies from 0 to 0.01. Since your network is tasked with learning how to combine these inputs through a series of linear combinations and nonlinear activations, the parameters associated with each input will also exist on different scales.



Gradient of larger parameter dominates the update



Both parameters can be updated in equal proportions

Batch Normalization

- ***Batch normalization layers*** act similar to the data preprocessing steps mentioned earlier
  - They calculate the mean μ and variance σ of a batch of input data, and normalize the data *x* to a zero mean and unit variance
  - I.e., $\hat{x} = \frac{x-\mu}{\sigma}$
- BatchNorm layers reduce the problems of proper initialization of the parameters and hyper-parameters
  - Result in faster convergence training, allow larger learning rates
  - Reduce the internal covariate shift