Deep Learning in NLP

Dr. Usman Zia

Loss Functions

Training Neural Networks

- ***Classification tasks***

Training examples

Pairs of N inputs x_i and ground-truth class labels y_i

Output Layer

Softmax Activations
[maps to a probability distribution]

$$P(y = j \mid \mathbf{x}) = \frac{e^{\mathbf{x}^\top \mathbf{w}_j}}{\sum_{k=1}^{K} e^{\mathbf{x}^\top \mathbf{w}_k}}$$

Loss function

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 $\hat{y}_i$

Loss Functions

Training Neural Networks

- ***Regression tasks***

Training examples

Pairs of N inputs x_i and ground-truth output values y_i

Output Layer

Linear (Identity) or Sigmoid Activation

Loss function

Mean Squared Error

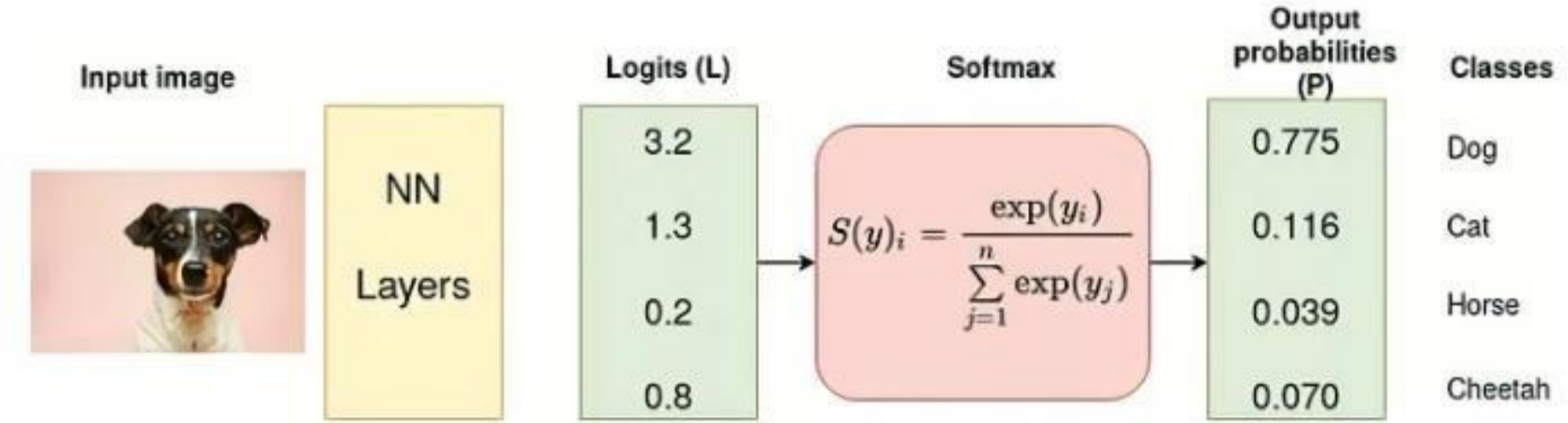
$$\mathcal{L}(\theta) = \frac{1}{n}\sum_{i=1}^{n}\left(y^{(i)} - \hat{y}^{(i)}\right)^2$$

Mean Absolute Error

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

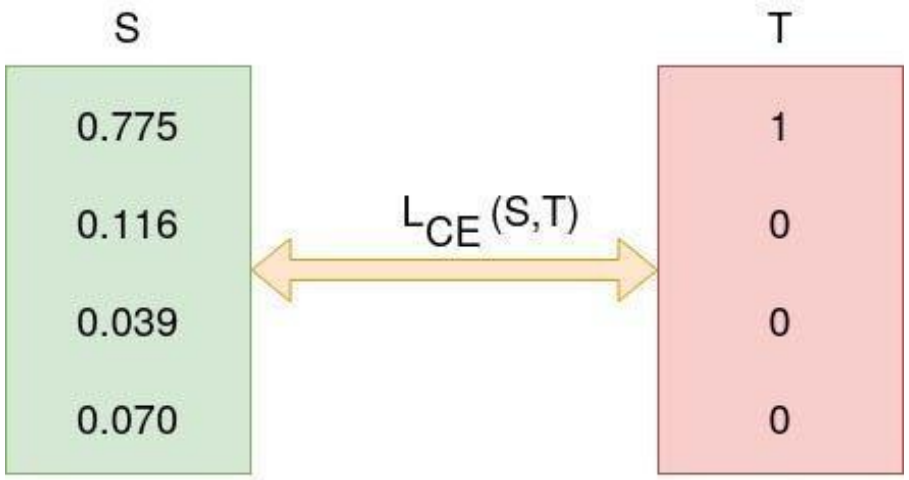
Cross-entropy Loss

- Consider a 4-class classification task where an image is classified as either a dog, cat, horse or cheetah



Cross-entropy Loss

- The purpose of the Cross-Entropy is to take the output probabilities (P) and measure the distance from the truth values



- Desired output is [1,0,0,0] for the class dog but the model outputs [0.775, 0.116, 0.039, 0.070]
- The objective is to make the model output be as close as possible to the desired output (truth values).
- During model training, the model weights are iteratively adjusted accordingly with the aim of minimizing the Cross-Entropy loss.
- As the model keeps training and the loss is getting minimized, we say that the model is *learning*.

What is Entropy?

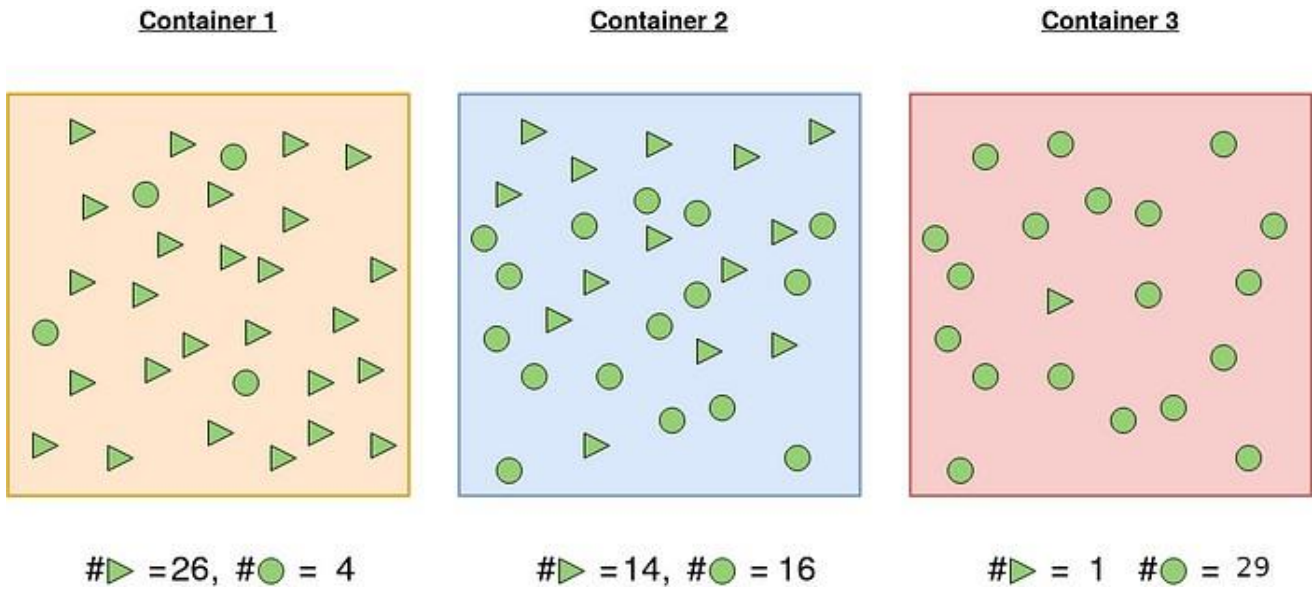
- Entropy of a random variable X is the level of uncertainty inherent in the variables possible outcome.
- For p(x) — probability distribution and a random variable X, entropy is defined as follows

$$H(X) = \begin{cases} -\int_x p(x) \log p(x), & \text{if } X \text{ is continous} \\ -\sum_x p(x) \log p(x), & \text{if } X \text{ is discrete} \end{cases}$$

- *log p(x)* measures the "Information Content"……. How surprising the outcome is?
- Reason for negative sign: log(p(x))<0 for all p(x) in (0,1) . p(x) is a probability distribution and therefore the values must range between 0 and 1.

What is Entropy?

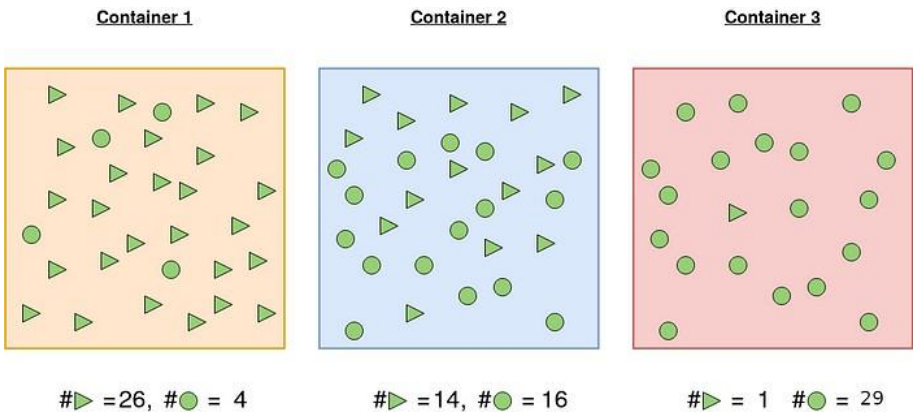
- Consider the following 3 "containers" with shapes: triangles and circles



- **Container 1:** The probability of picking a triangle is 26/30 and the probability of picking a circle is 4/30. For this reason, the probability of picking one shape and/or not picking another is more certain.
- **Container 2:** Probability of picking the a triangular shape is 14/30 and 16/30 otherwise. There is almost 50–50 chance of picking any particular shape. Less certainty of picking a given shape than in 1.
- **Container 3:** A shape picked from container 3 is highly likely to be a circle. Probability of picking a circle is 29/30 and the probability of picking a triangle is 1/30. It is highly certain than the shape picked will be circle.

What is Entropy?

- Let us calculate the entropy so that we ascertain our assertions about the certainty of picking a given shape.



Entropy for container 1:

$$
\begin{aligned}
H(X) &= -\sum_{x} p(x)\log(p(x)) \\
&= -[p(x_1)\log_2(p(x_1)) + p(x_2)\log_2(p(x_2))] \\
&= -\left[\frac{26}{30}\log_2\left(\frac{26}{30}\right) + \frac{4}{30}\log_2\left(\frac{4}{30}\right)\right] \\
&= 0.5665
\end{aligned}
$$

Entropy for container 2:

$$
\begin{aligned}
H(X) &= -\sum_{x} p(x)\log(p(x)) \\
&= -[p(x_1)\log_2(p(x_1)) + p(x_2)\log_2(p(x_2))] \\
&= -\left[\frac{14}{30}\log_2\left(\frac{14}{30}\right) + \frac{16}{30}\log_2\left(\frac{16}{30}\right)\right] \\
&= 0.9968
\end{aligned}
$$

Entropy for container 3:

$$
\begin{aligned}
H(X) &= -\sum_{x} p(x)\log(p(x)) \\
&= -[p(x_1)\log_2(p(x_1)) + p(x_2)\log_2(p(x_2))] \\
&= -\left[\frac{1}{30}\log_2\left(\frac{1}{30}\right) + \frac{29}{30}\log_2\left(\frac{29}{30}\right)\right] \\
&= 0.2108
\end{aligned}
$$

Cross-Entropy Loss Function

- Also called **logarithmic loss**, **log loss** or **logistic loss**.
- Each predicted class probability is compared to the actual class desired output 0 or 1 and a score/loss is calculated that penalizes the probability based on how far it is from the actual expected value.
- The penalty is logarithmic in nature yielding a large score for large differences close to 1 and small score for small differences tending to 0.

- Cross-entropy is defined as

$$L_{\text{CE}} = -\sum_{i=1}^{n} t_i \log(p_i), \quad \text{for n classes,}$$

where t_i is the truth label and p_i is the Softmax probability for the i^{th} class.

Binary Cross-Entropy Loss

- For binary classification (a classification task with two classes — 0 and 1), we have binary cross-entropy defined as

$$
\begin{aligned}
L &= -\sum_{i=1}^{2} t_i \log(p_i) \\
&= -[t_1 \log(p_1) + t_2 \log(p_2)] \\
&= -[t \log(p) + (1-t) \log(1-p)]
\end{aligned}
$$

where t_i is the truth value taking a value 0 or 1 and p_i is the Softmax probability for the i^{th} class. Since we have two classes 1 and 0 we can have $t_1 = 1$ and $t_2 = 0$ and since $p's$ are probabilities then $p_1 + p_2 = 1 \implies p_1 = 1 - p_2$. For the convenience of notation, we can then let $t_1 = t, t_2 = 1 - t, p_1 = p$ and $p_2 = 1 - p$.

Binary Cross-Entropy Loss

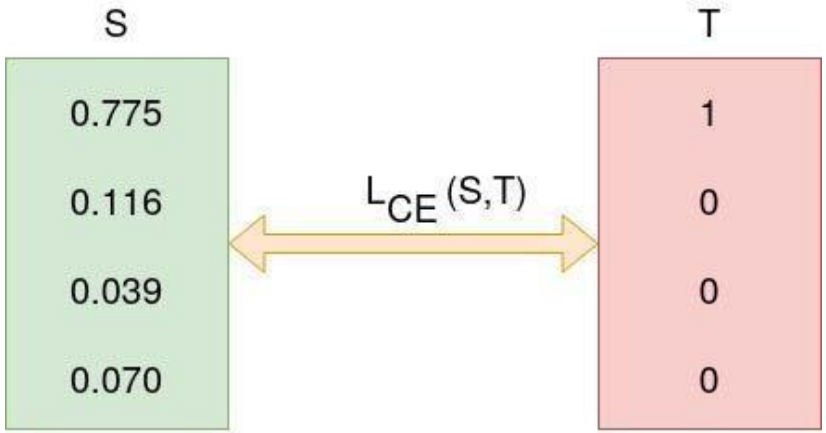
- Binary cross-entropy is often calculated as the average cross-entropy across all data examples, that is,

$$L = -\frac{1}{N}\left[\sum_{j=1}^{N}[t_j \log(p_j) + (1 - t_j)\log(1 - p_j)]\right]$$

for N data points where t_i is the truth value taking a value 0 or 1 and p_i is the Softmax probability for the i^{th} data point.

Cross-Entropy Loss

- Consider the classification problem with the following Softmax probabilities (S) and the labels (T). The objective is to calculate for cross-entropy loss given these information.

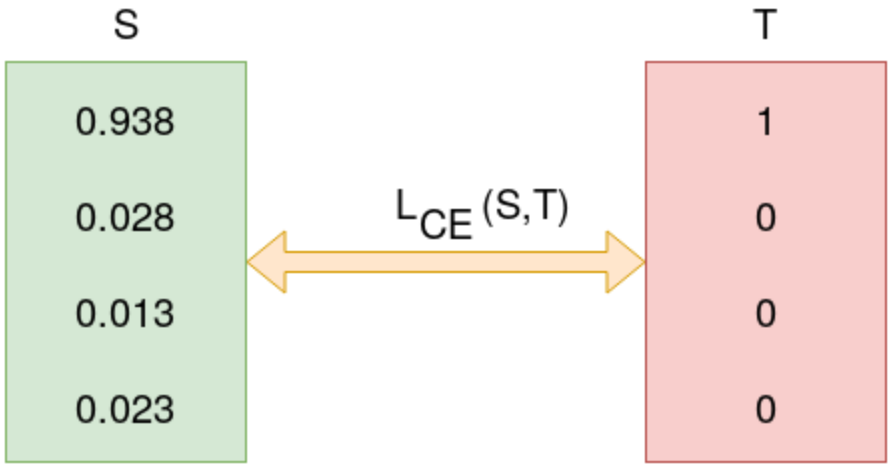


- The categorical cross-entropy is computed as follows

$$
\begin{aligned}
L_{CE} &= -\sum_{i=1} T_i \log(S_i) \\
&= -\left[1 \log_2(0.775) + 0 \log_2(0.126) + 0 \log_2(0.039) + 0 \log_2(0.070)\right] \\
&= -\log_2(0.775) \\
&= 0.3677
\end{aligned}
$$

Cross-Entropy Loss

- Assume that after some iterations of model training the model outputs the following vector o



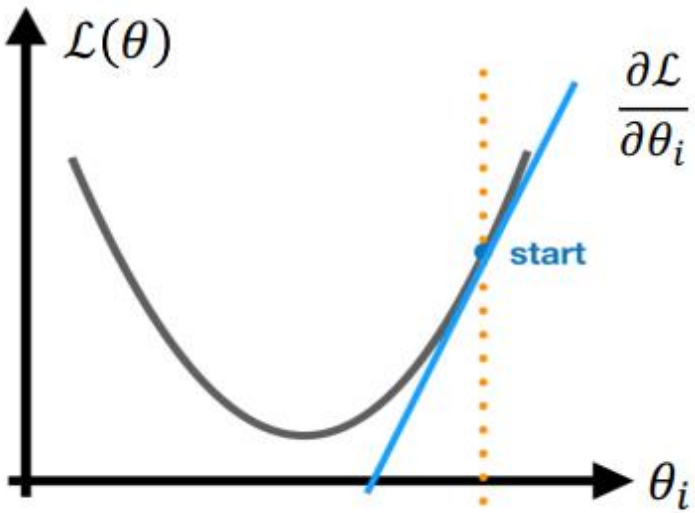
- The categorical cross-entropy now becomes

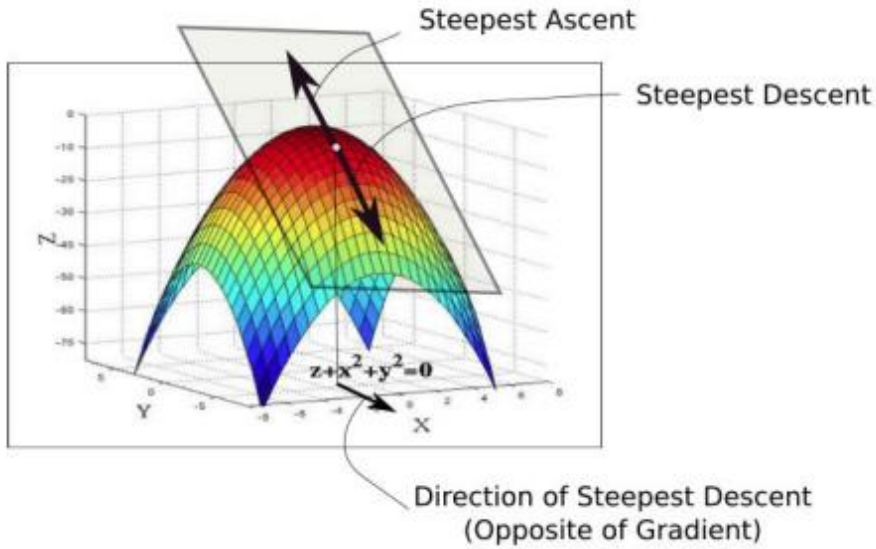
$$L_{CE} = -1\log_2(0.936) + 0 + 0 + 0$$
$$= 0.095$$

Training NNs

Training Neural Networks

- Optimizing the loss function $\mathcal{L}(\theta)$
  - Almost all DL models these days are trained with a variant of the ***gradient descent*** (GD) algorithm
  - GD applies iterative refinement of the network parameters θ
  - GD uses the opposite direction of the gradient of the loss with respect to the NN parameters (i.e.,$\nabla\mathcal{L}(\theta) = [\partial\mathcal{L}/\partial\theta_i]$) for updating θ
    - The gradient of the loss function $\nabla\mathcal{L}(\theta)$ gives the direction of fastest increase of the loss function $\mathcal{L}(\theta)$ when the parameters θ are changed

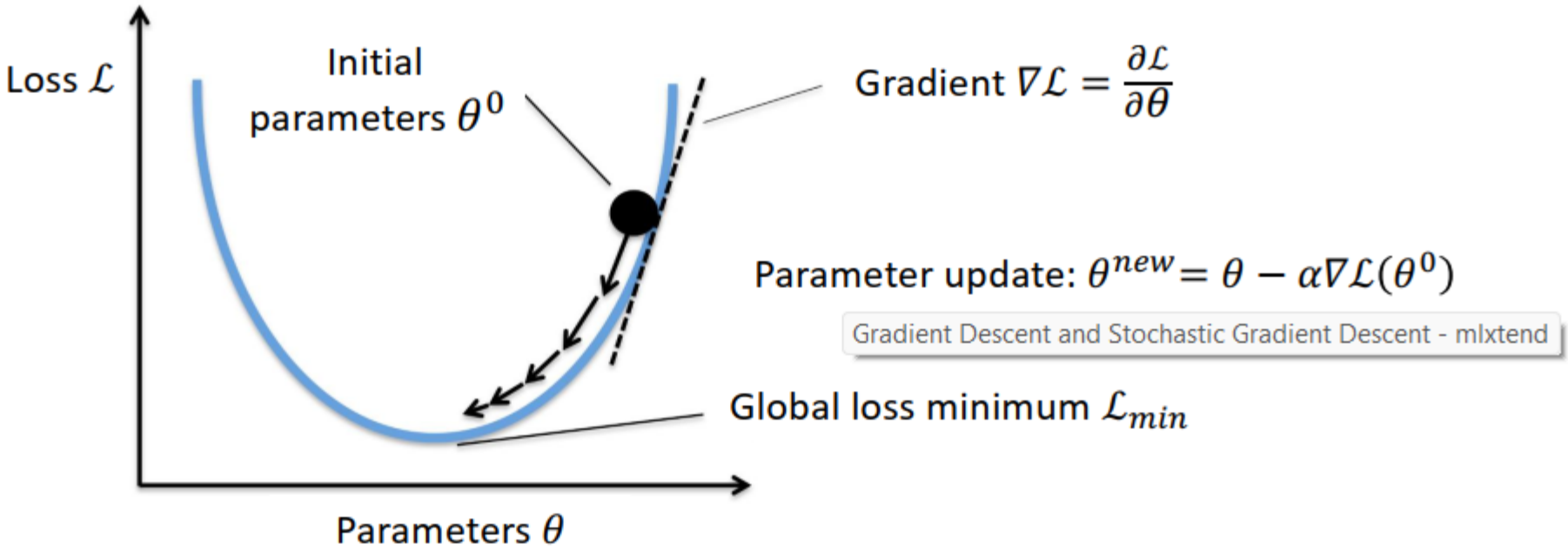




Gradient Descent Algorithm

Training Neural Networks

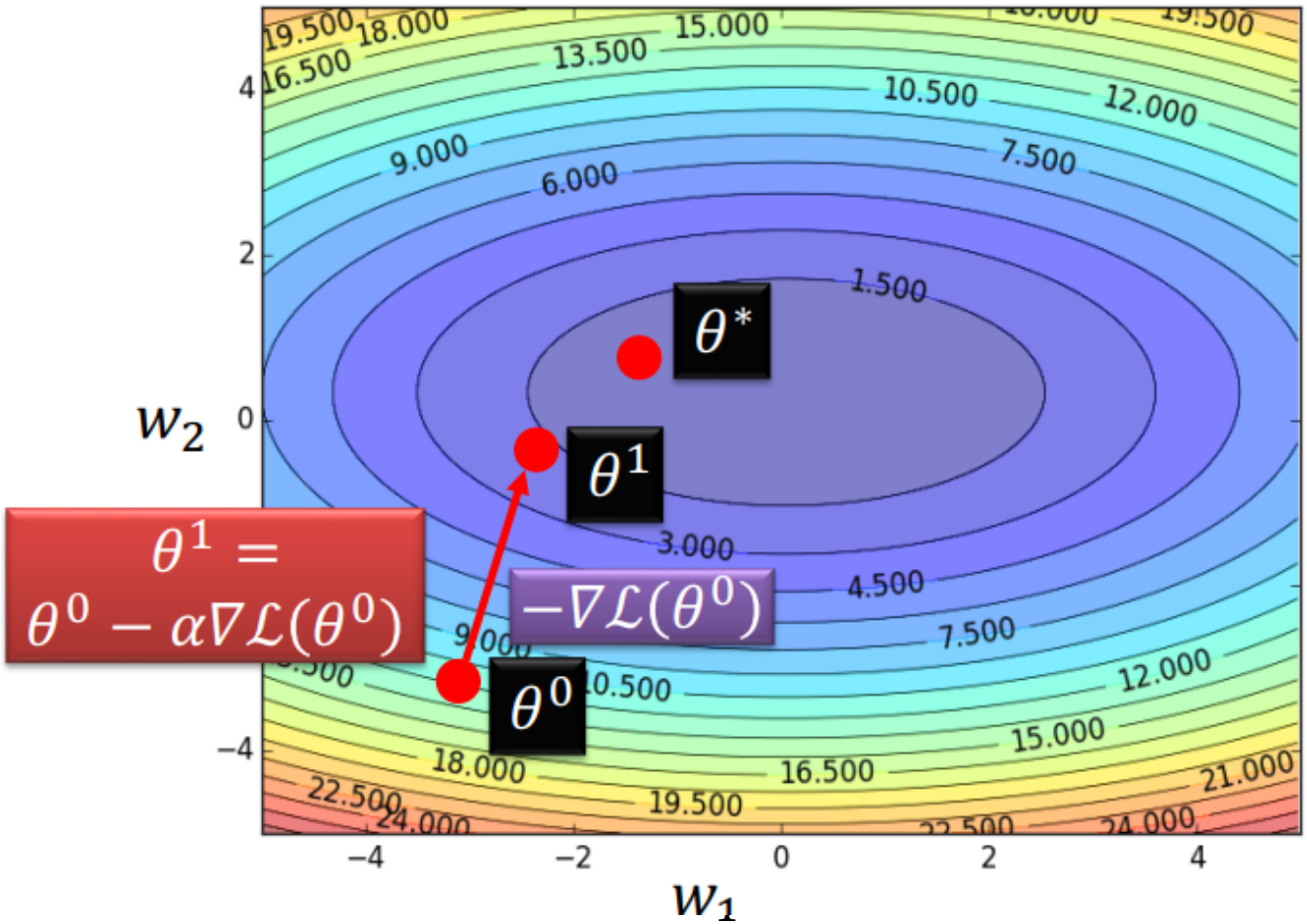
- Steps in the ***gradient descent algorithm***:
  1. Randomly initialize the model parameters, θ^0
  2. Compute the gradient of the loss function at the initial parameters θ^0: $\nabla\mathcal{L}(\theta^0)$
  3. Update the parameters as: $\theta^{new} = \theta^0 - \alpha\nabla\mathcal{L}(\theta^0)$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



Gradient Descent Algorithm

Training Neural Networks

- 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)



1. Randomly pick a starting point θ^0

2. Compute the gradient at θ^0, $\nabla\mathcal{L}(\theta^0)$

3. Times the learning rate α, and update θ, $\theta^{new} = \theta^0 - \alpha\nabla\mathcal{L}(\theta^0)$

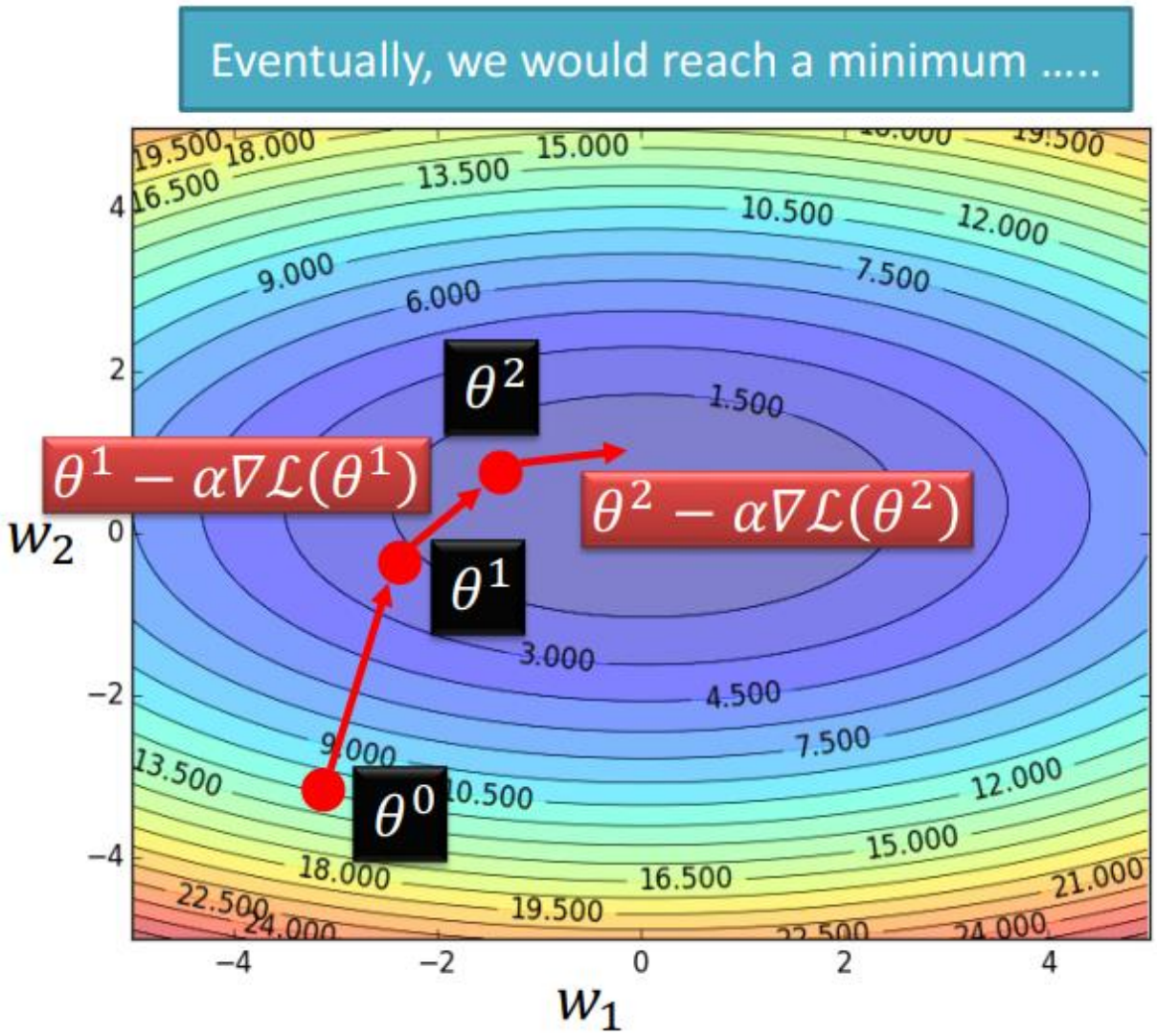
4. Go to step 2, repeat

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

Gradient Descent Algorithm

Training Neural Networks

- Example (contd.)



2. Compute the gradient at θ^{old}, $\nabla\mathcal{L}(\theta^{old})$

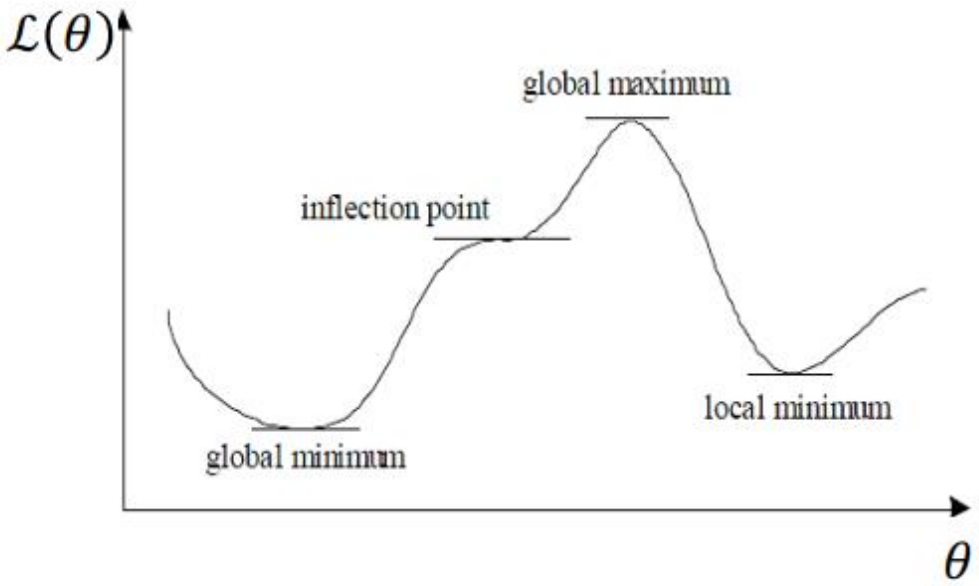
3. Times the learning rate α, and update θ, $\theta^{new} = \theta^{old} - \alpha\nabla\mathcal{L}(\theta^{old})$

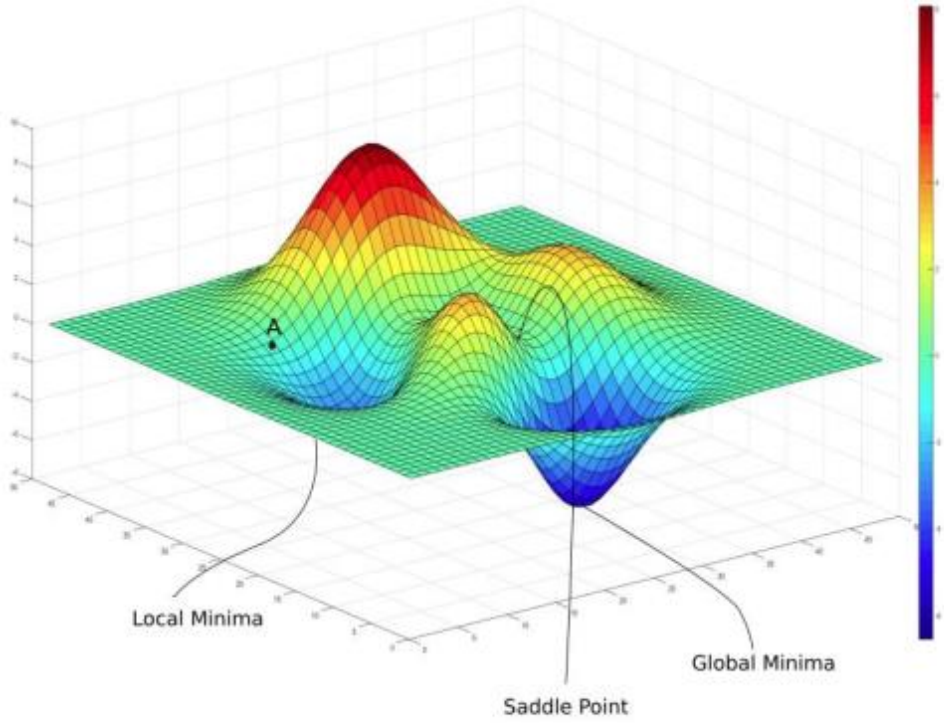
4. Go to step 2, repeat

Gradient Descent Algorithm

Training Neural Networks

- 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

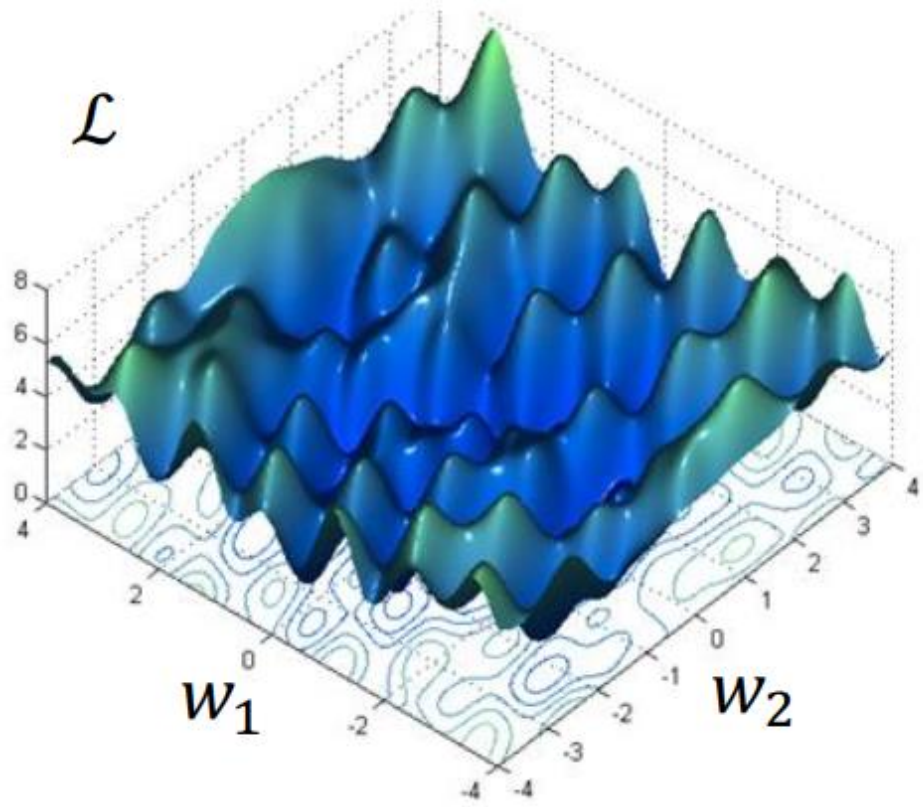




Gradient Descent Algorithm

Training Neural Networks

- For most tasks, the loss surface $\mathcal{L}(\theta)$ is highly complex (and non-convex)
- Random initialization in NNs results in different initial parameters θ^0 every time the NN is trained
  - Gradient descent may reach different minima at every run
  - Therefore, NN will produce different predicted outputs
- In addition, currently we don't have algorithms that guarantee reaching a global minimum for an arbitrary loss function



Backpropagation

Training Neural Networks

- Modern NNs employ the ***backpropagation*** method for calculating the gradients of the loss function $\nabla\mathcal{L}(\theta) = \partial\mathcal{L}/\partial\theta_i$
  - Backpropagation is short for "backward propagation"
- For training NNs, 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 $\nabla\mathcal{L}(\theta)$
  - The chain rule is used for calculating the partial derivatives of the loss function with respect to the parameters θ in the different layers in the network
- Each update of the model parameters θ during training takes one forward and one backward pass (e.g., of a batch of inputs)
- Automatic calculation of the gradients (automatic differentiation) is available in all current deep learning libraries
  - It significantly simplifies the implementation of deep learning algorithms, since it obviates deriving the partial derivatives of the loss function by hand

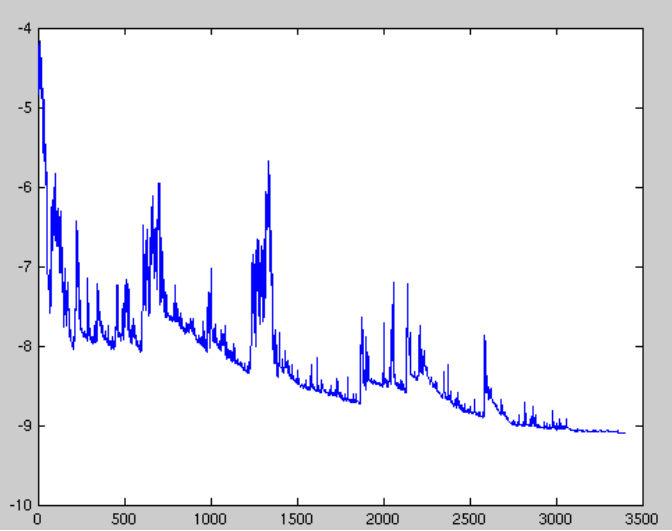
Mini-batch Gradient Descent

- 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
  - Therefore, GD (a.k.a. vanilla 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 images, update the parameters θ, and repeat until all images are used
    - At the next epoch, shuffle the training data, and repeat the above process
  - Mini-batch GD results in much faster training
  - Typical mini-batch size: 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

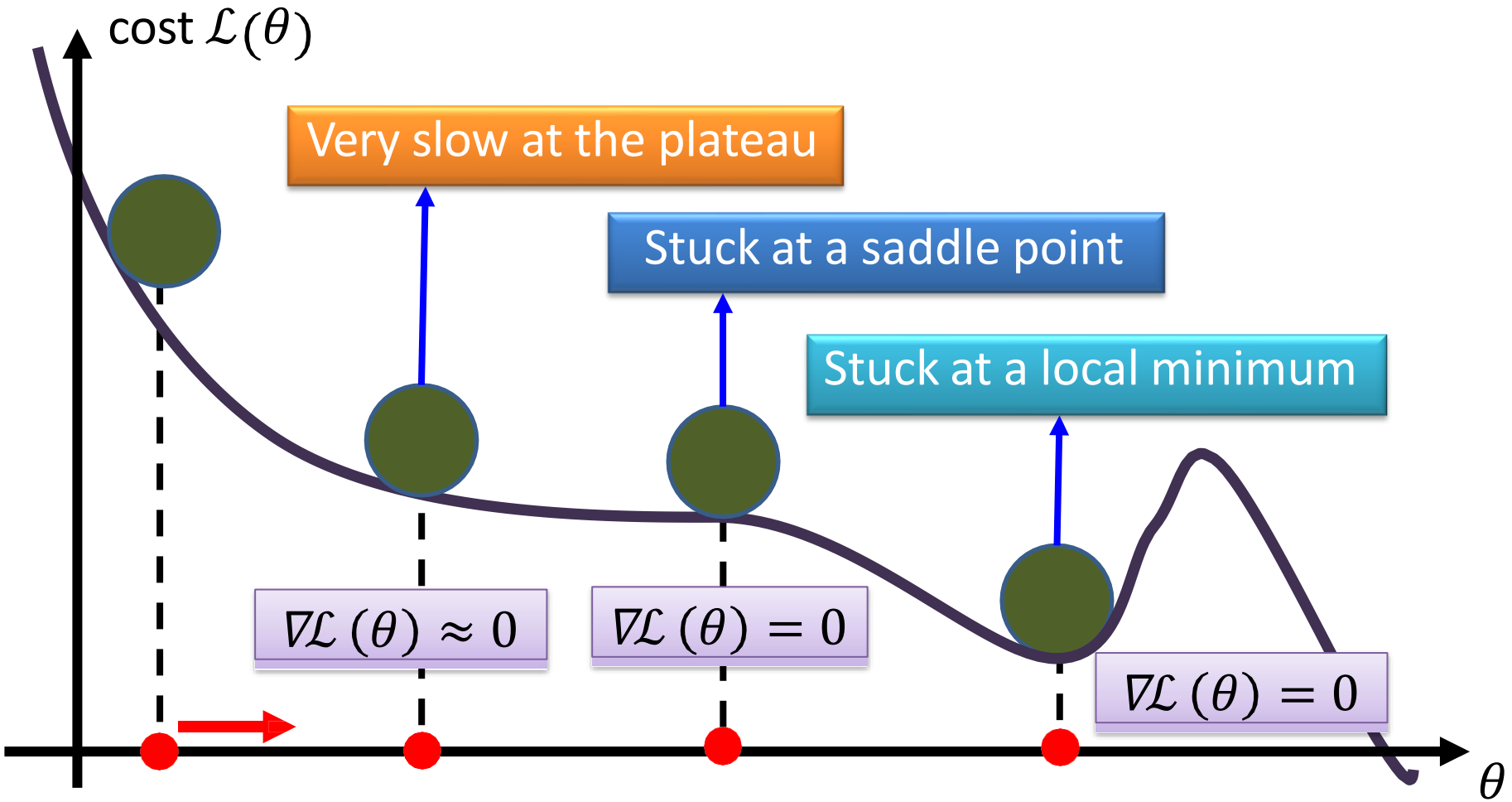
Training Neural Networks

- ***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 fluctuations 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)



Problems with Gradient Descent

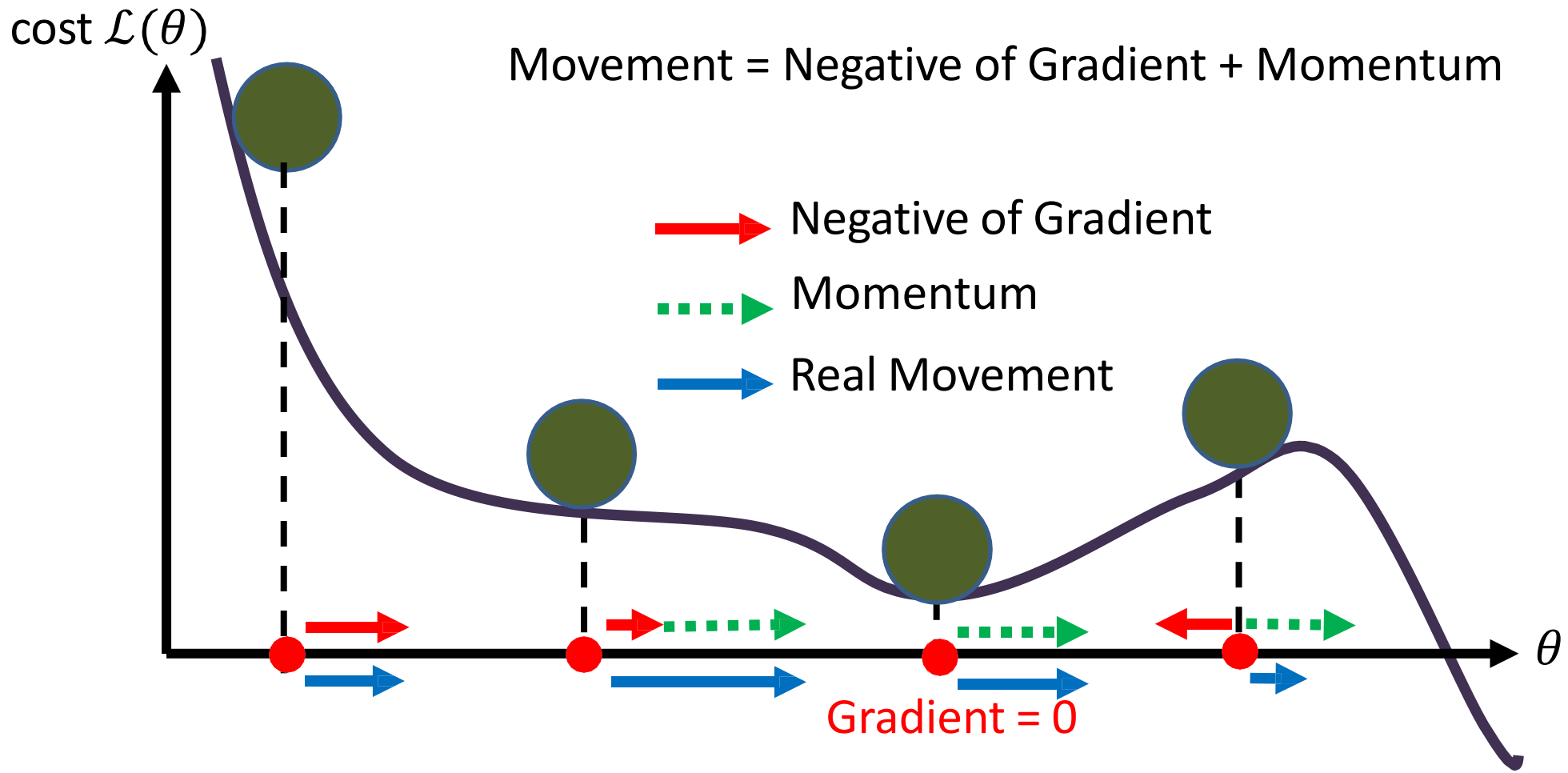
- Besides the local minima problem, the GD algorithm can be very slow at plateaus, and it can get stuck at saddle points



Gradient Descent with Momentum

Training Neural Networks

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



Gradient Descent with Momentum

Training Neural Networks

- 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.,

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

  - This term 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
- This method updates the parameters θ in the direction of the weighted average of the past gradients

Adam

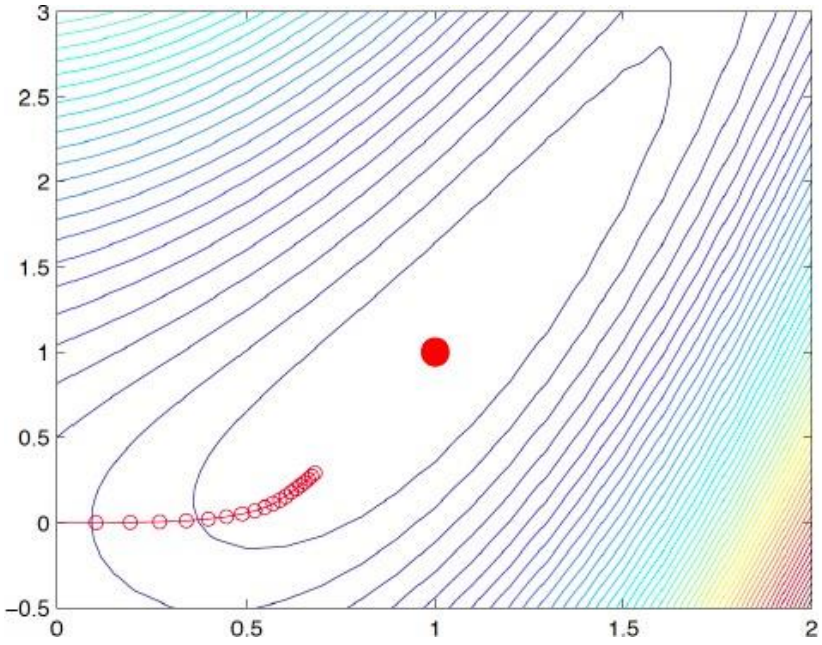
- ***Adaptive Moment Estimation (Adam)***
  - Adam combines insights from the momentum optimizers that accumulate the values of past gradients, and it also introduces new terms based on the second moment of the gradient
    - Similar to GD with momentum, Adam computes a **weighted average of past gradients** (first moment of the gradient), i.e., $V^t = \beta_1 V^{t-1} + (1-\beta_1)\nabla\mathcal{L}(\theta^{t-1})$
    - Adam also computes a **weighted average of past squared gradients** (second moment of the gradient), , i.e., $U^t = \beta_2 U^{t-1} + (1-\beta_2)\left(\nabla\mathcal{L}(\theta^{t-1})\right)^2$
  - The parameter update is: $\theta^t = \theta^{t-1} - \alpha \frac{\hat{V}^t}{\sqrt{\hat{U}^t}+\epsilon}$
    - Where: $\hat{V}^t = \frac{V^t}{1-\beta_1}$ and $\hat{U}^t = \frac{U^t}{1-\beta_2}$
    - The proposed default values are $\beta_1 = 0.9$, $\beta_2 = 0.999$, and $\epsilon = 10^{-8}$
- Other commonly used optimization methods include:
  - Adagrad, Adadelta, RMSprop, Nadam, etc.
  - Most commonly used optimizers nowadays are Adam and SGD with momentum

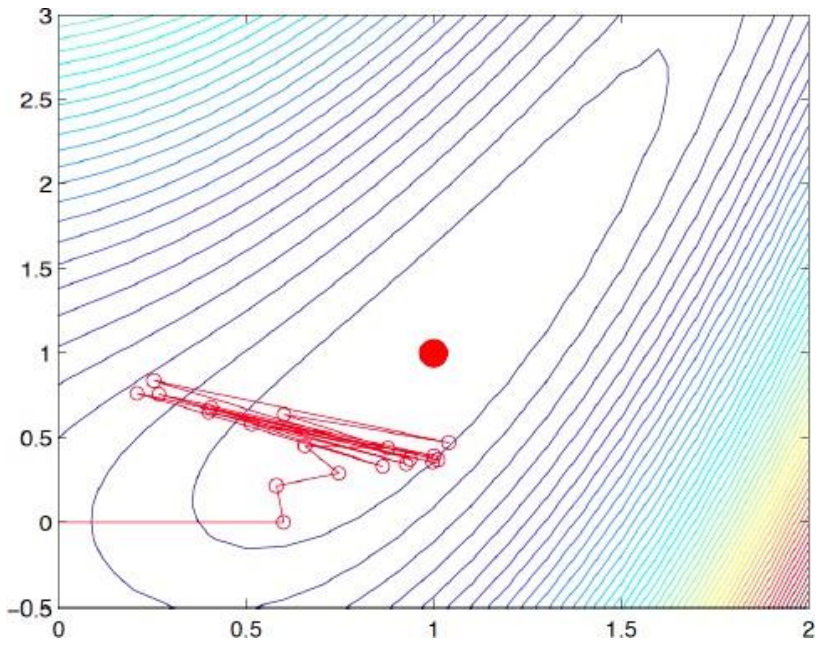
Learning Rate

Training Neural Networks

- *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



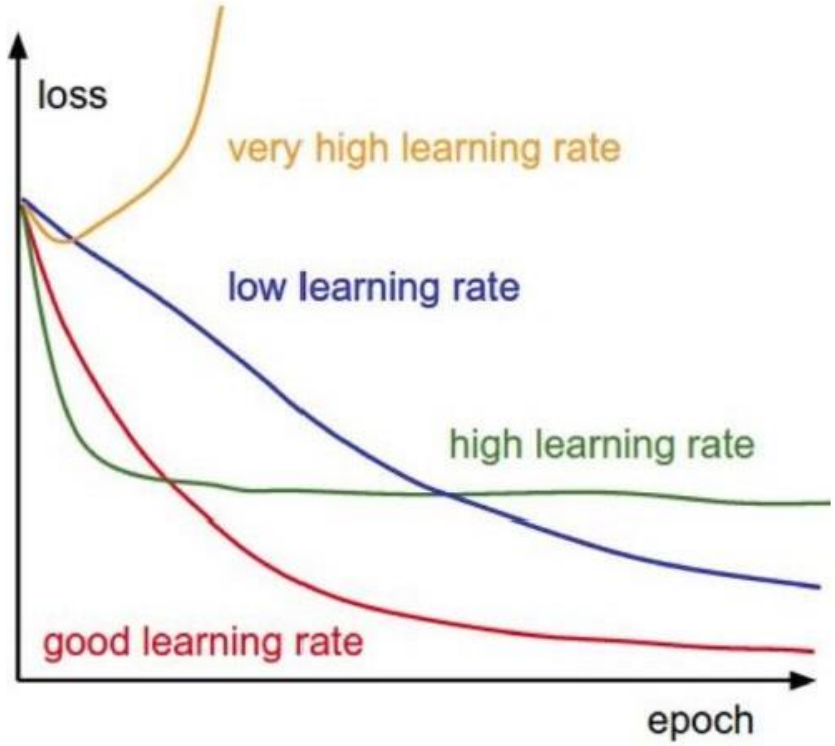


LR too large

Learning Rate

Training Neural Networks

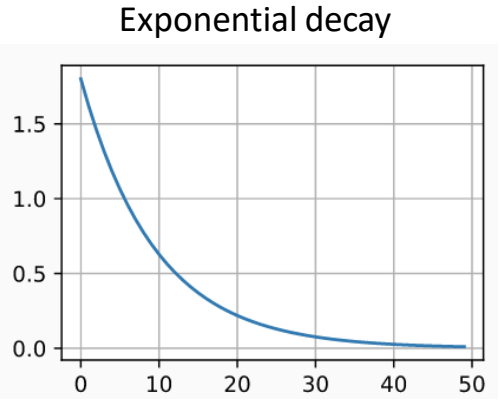
- Training loss for different learning rates
  - High learning rate: the loss increases or plateaus too quickly
  - Low learning rate: the loss decreases too slowly (takes many epochs to reach a solution)

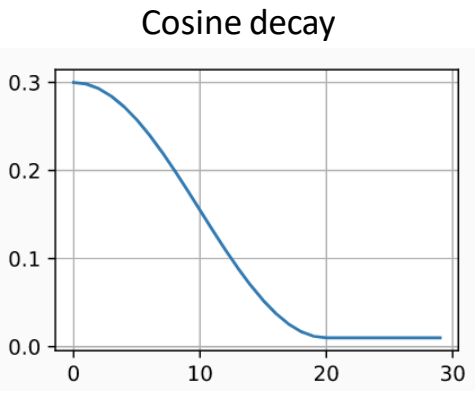


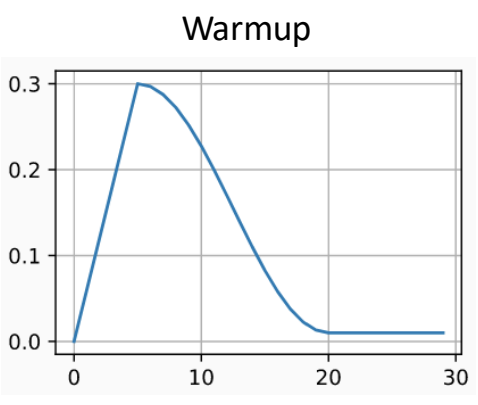
Learning Rate Scheduling

Training Neural Networks

- ***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



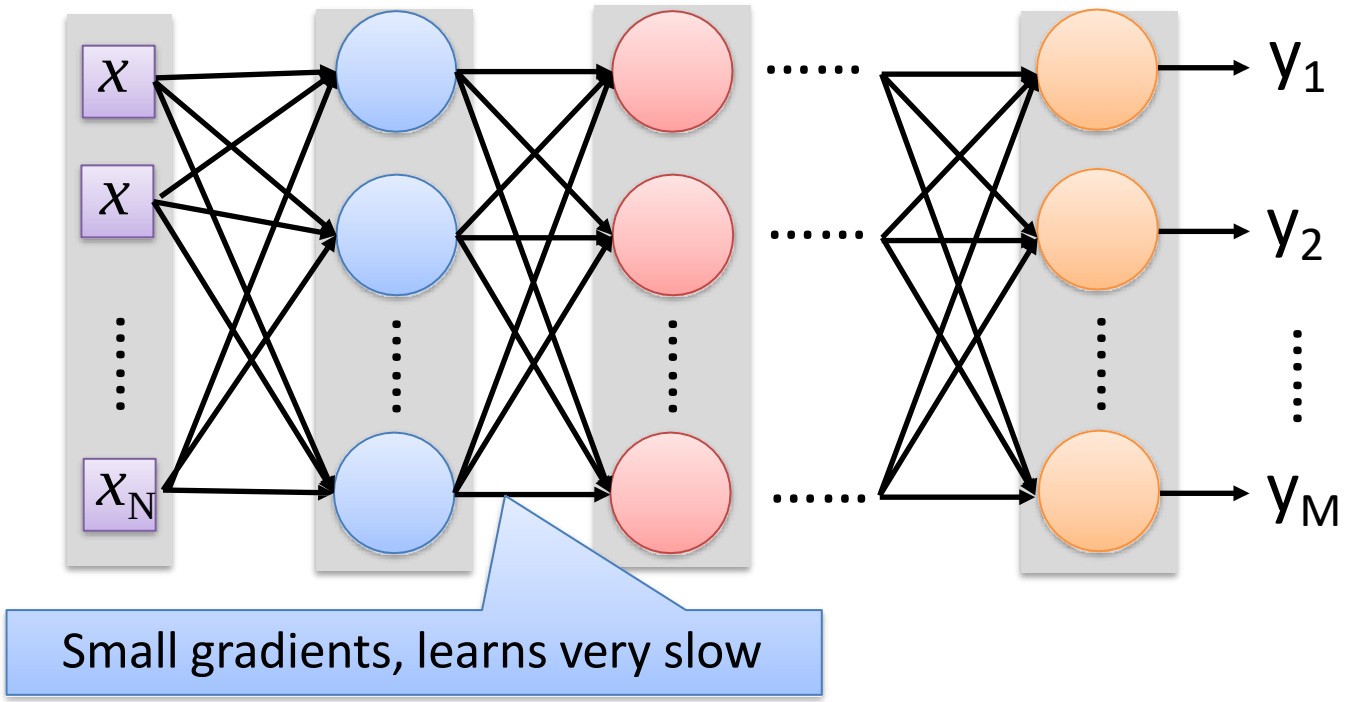




Vanishing Gradient Problem

Training Neural Networks

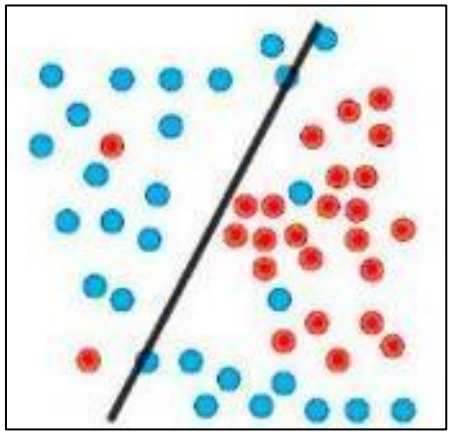
- In some cases, during training, the gradients can become either very small (vanishing gradients) or very large (exploding gradients)
  - They result in very small or very large update of the parameters
  - Solutions: change learning rate, ReLU activations, regularization …..

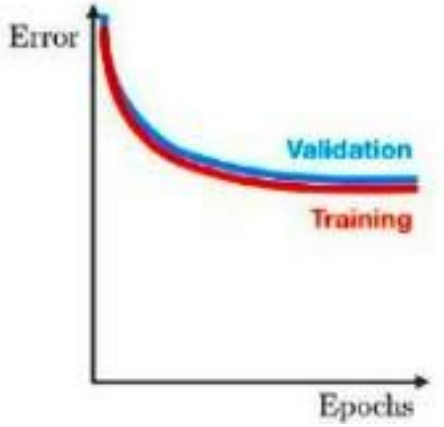


Generalization

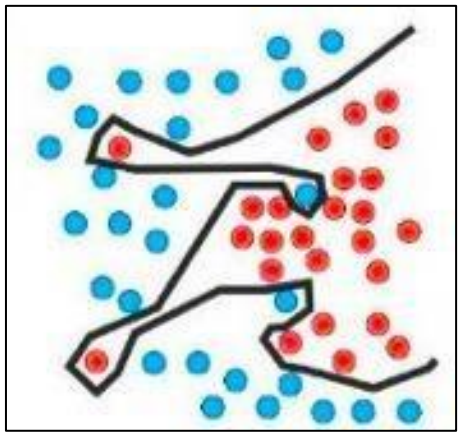
Generalization

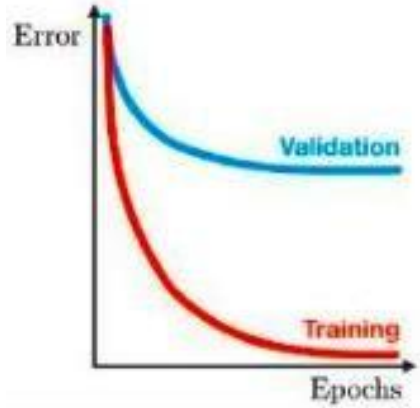
- ***Underfitting***
  - The model is too "simple" to represent all the relevant class characteristics
  - E.g., model with too few parameters
  - Produces high error on the training set and high error on the validation set





- ***Overfitting***
  - The model is too "complex" and fits irrelevant characteristics (noise) in the data
  - E.g., model with too many parameters
  - Produces low error on the training error and high error on the validation set

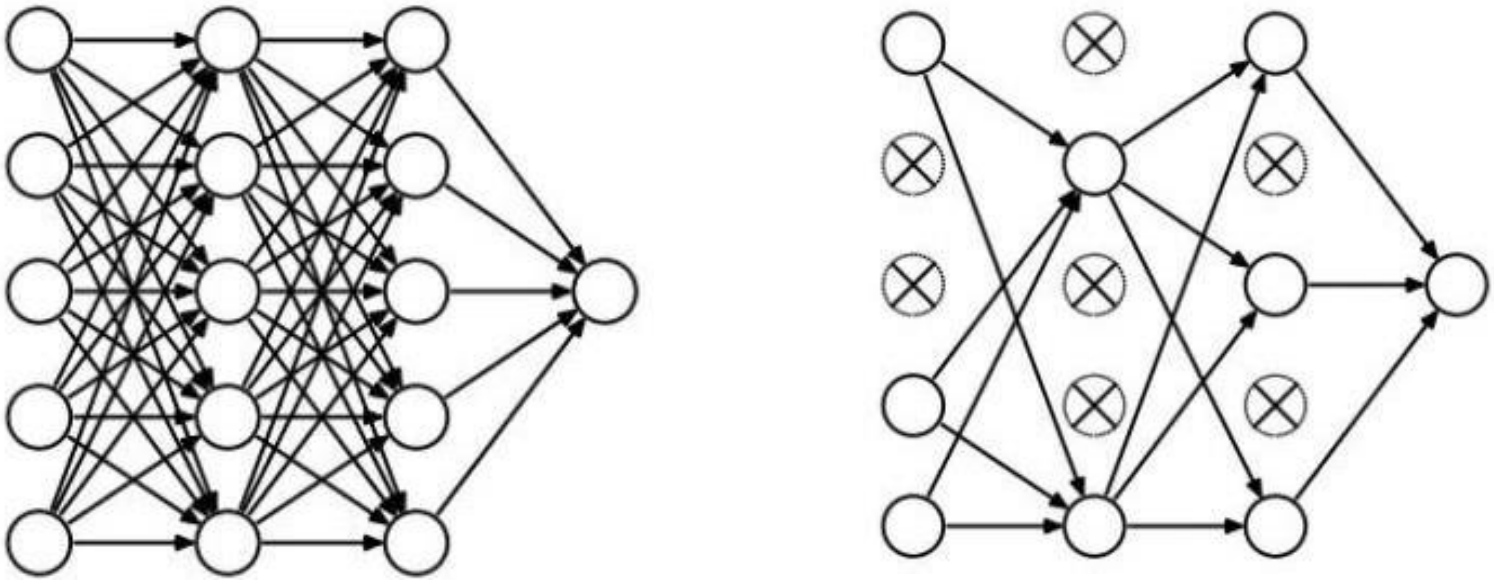




Regularization: Dropout

Regularization

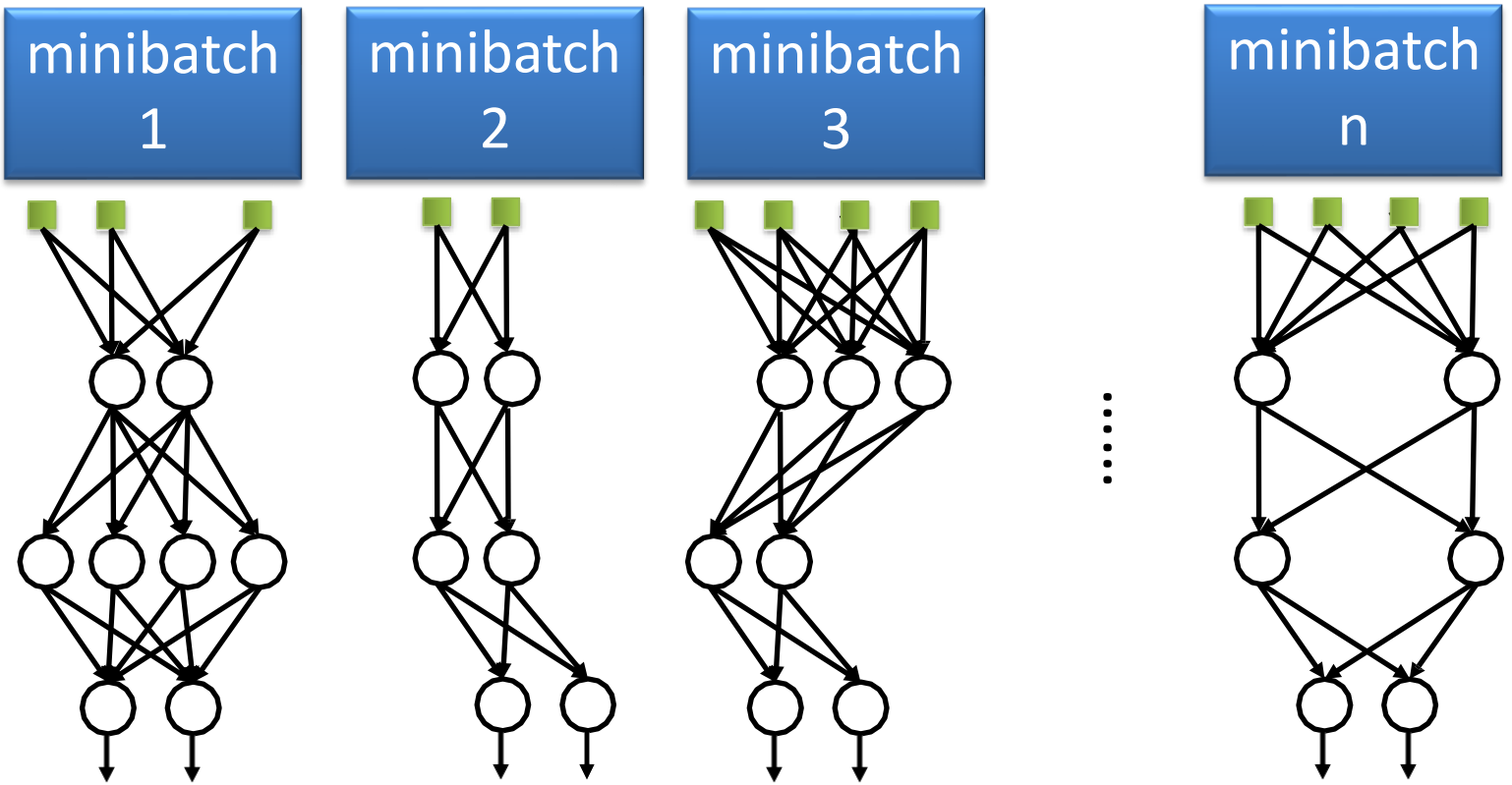
- ***Dropout***
  - Randomly drop units (along with their connections) during training
  - Each unit is retained with a fixed dropout rate p, independent of other units
  - The hyper-parameter p needs to be chosen (tuned)
    - Often, between 20% and 50% of the units are dropped



Regularization: Dropout

Regularization

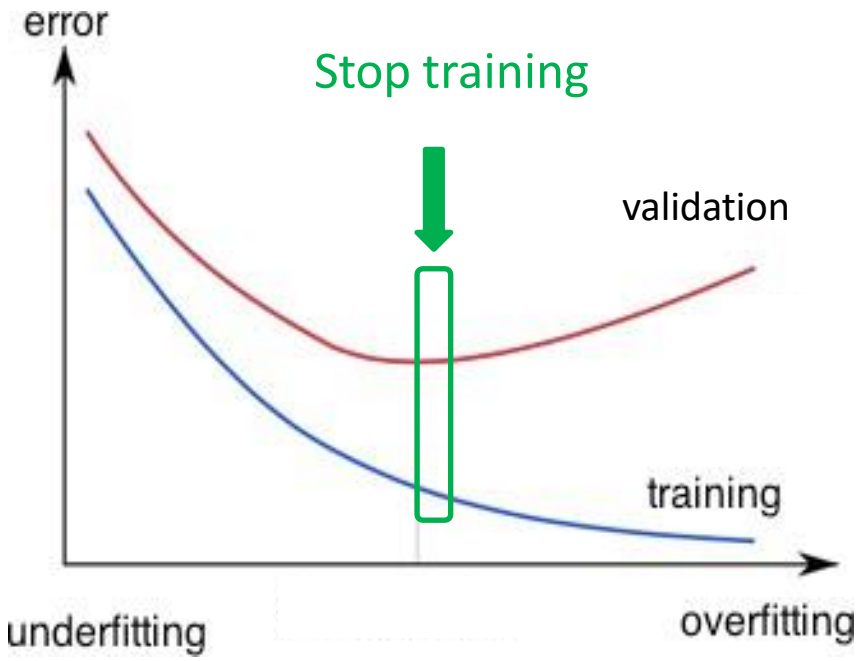
- Dropout is a kind of ensemble learning
  - Using one mini-batch to train one network with a slightly different architecture



Regularization: Early Stopping

Regularization

- ***Early-stopping***
  - During model training, use a validation set
    - E.g., validation/train ratio of about 20% to 80%
  - Stop when the validation accuracy (or loss) has not improved after *n* epochs
    - The parameter *n* is called patience



Batch Normalization

Regularization

- ***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 alleviate 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
- BatchNorm layers are inserted immediately after convolutional layers or fully-connected layers, and before activation layers
  - They are very common with convolutional NNs

Hyper-parameter Tuning

Hyper-parameter Tuning

- Training NNs can involve setting many ***hyper-parameters***
- The most common hyper-parameters include:
  - Number of layers, and number of neurons per layer
  - Initial learning rate
  - Learning rate decay schedule (e.g., decay constant)
  - Optimizer type
- Other hyper-parameters may include:
  - Regularization parameters (P_2 penalty, dropout rate)
  - Batch size
  - Activation functions
  - Loss function
- Hyper-parameter tuning can be time-consuming for larger NNs

Deep vs Shallow Networks

Deep vs Shallow Networks

- Deeper networks perform better than shallow networks
  - But only up to some limit: after a certain number of layers, the performance of deeper networks plateaus