Foundations of Machine Learning (CS 725)

FALL 2024

Lecture 14:
- Backpropagation

Instructor: Preethi Jyothi

Training Feedforward Neural Networks

Optimization Problem

- To train a neural network, define a loss function $L(y,\tilde{y})$:
  a function of the true output y and the predicted output $\tilde{y}$
- $L(y,\tilde{y})$ assigns a non-negative numerical score to the neural network's output, $\tilde{y}$
- The parameters of the network are set to minimise L over the training examples (i.e. a sum of losses over different training samples)
- L is typically minimised using a *gradient-based method*

Loss Function

Overall loss function, $J(\theta)$, measures the total loss over the entire training set:

$$J(\theta) = \sum_{i=1}^{N} L(\mathrm{NN}(\mathbf{x}_i; \theta), y_i)$$

Cross-entropy loss is one of the most popular classification-based loss functions. Assuming $\mathrm{NN}(\mathbf{x}_i; \theta)$ returns a probability, binary cross-entropy can be defined as:

$$J(\theta) = -\sum_{i=1}^{N} y_i \log\left(\mathrm{NN}(\mathbf{x}_i; \theta)\right) + (1 - y_i) \log\left(1 - \mathrm{NN}(\mathbf{x}_i; \theta)\right)$$

Stochastic Gradient Descent (SGD)

SGD Algorithm

Inputs: $\mathrm{NN}(x; \theta)$, Training examples, $x_1 \ldots x_n$; outputs, $y_1 \ldots y_n$ and Loss function L

Randomly initialize θ

do until **stopping criterion**

    Pick a training example $\{x_i, y_i\}$

    Compute the loss $L(\mathrm{NN}(x_i; \theta), y_i)$

    Compute gradient of L, $\nabla_\theta L$ with respect to θ

    $\theta \leftarrow \theta - \eta \nabla_\theta L$

Weight Update Rule

done

Learning Rate

Return: θ

Mini-batch Gradient Descent (GD)

Mini-batch GD Algorithm

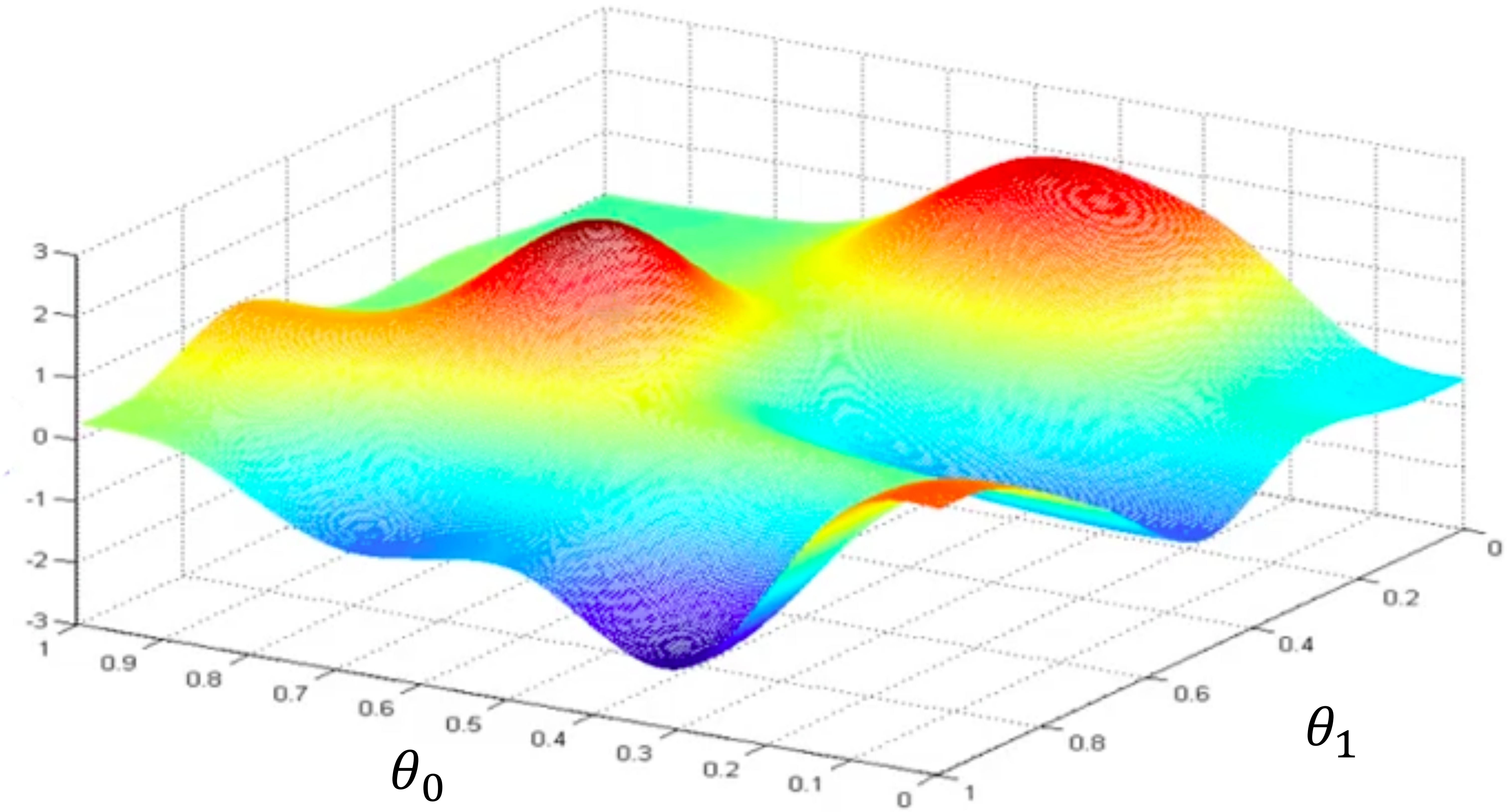
Inputs: $\mathrm{NN}(x;\theta)$, Training examples, $x_1 \dots x_n$; outputs, $y_1 \dots y_n$ and Loss function L

Randomly initialize θ
do until **stopping criterion**
Randomly sample a batch of training examples $\{x_i, y_i\}_{i=1}^{b}$
(where the batch size, b, is a hyperparameter)
Compute gradient of L over the batch, $\nabla_\theta L$ with respect to θ
$\theta \longleftarrow \theta - \eta \nabla_\theta L$
done

Return: θ

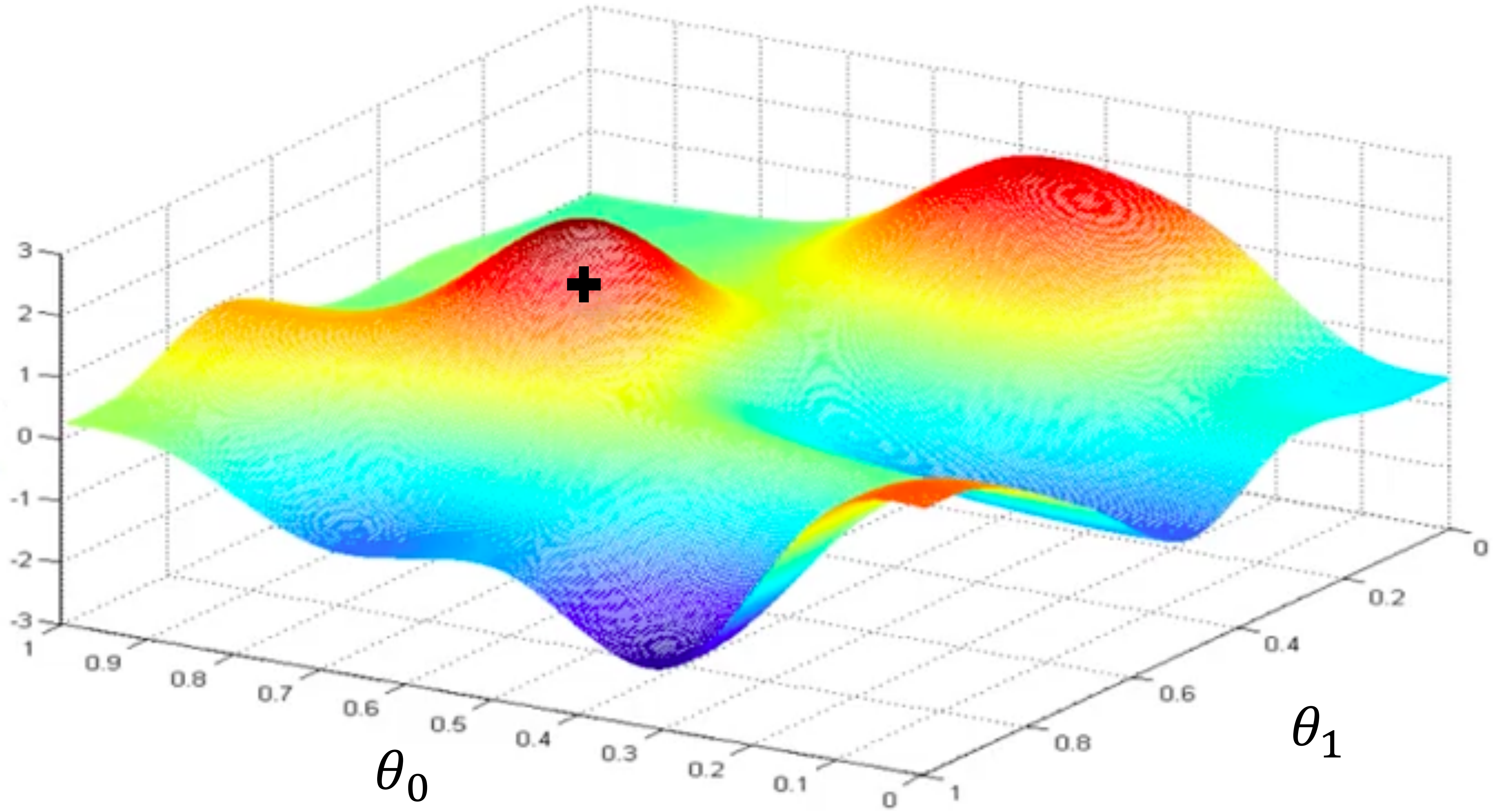
Loss Optimization

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



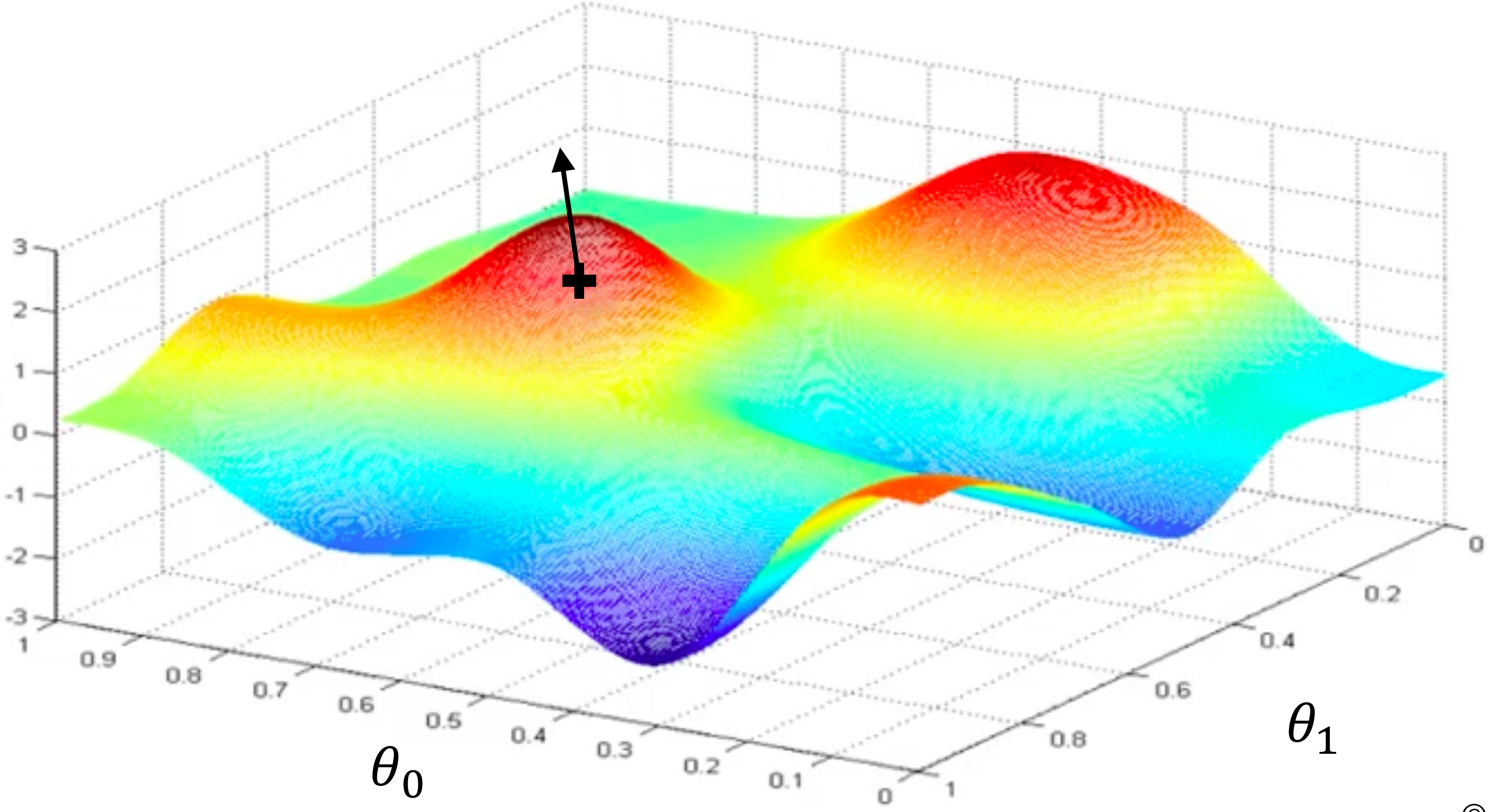
Loss Optimization

Randomly pick an initial (θ_0, θ_1)



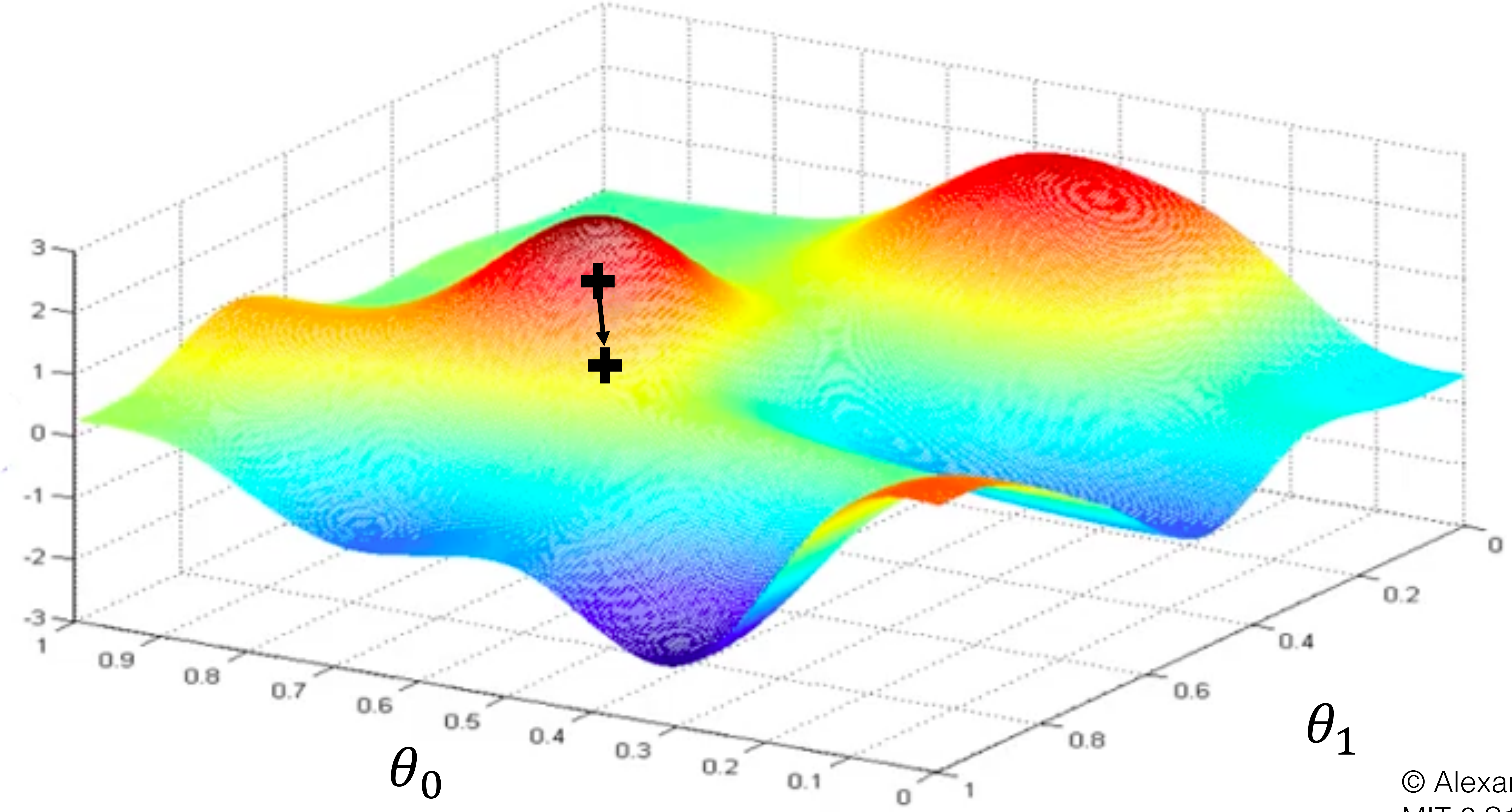
Loss Optimization

Compute gradient, $\frac{\partial J(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}}$



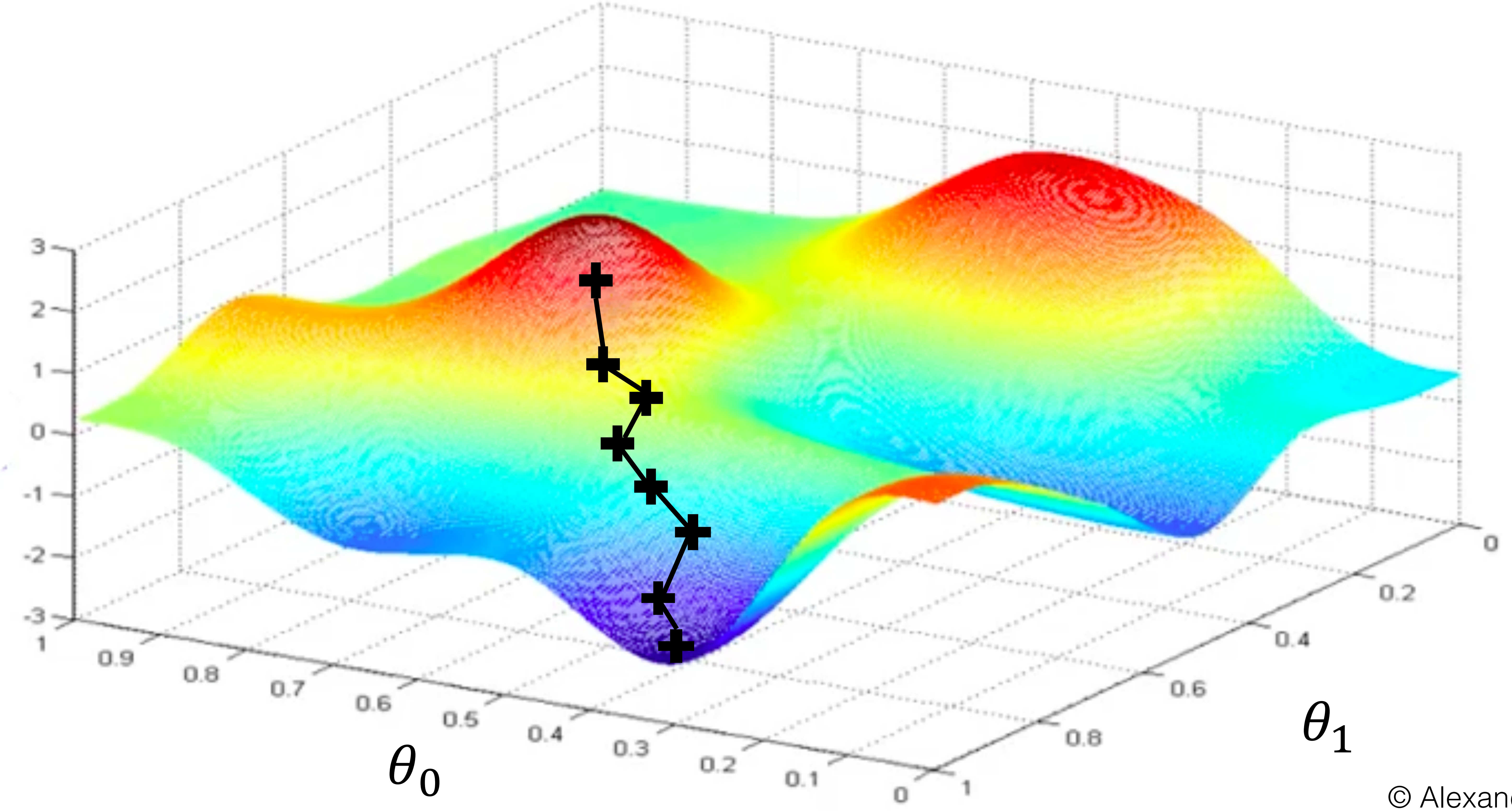
Loss Optimization

Take small step in opposite direction of gradient



Loss Optimization

Repeat until convergence



Training a Neural Network

Define the **Loss function** to be minimised as a node L

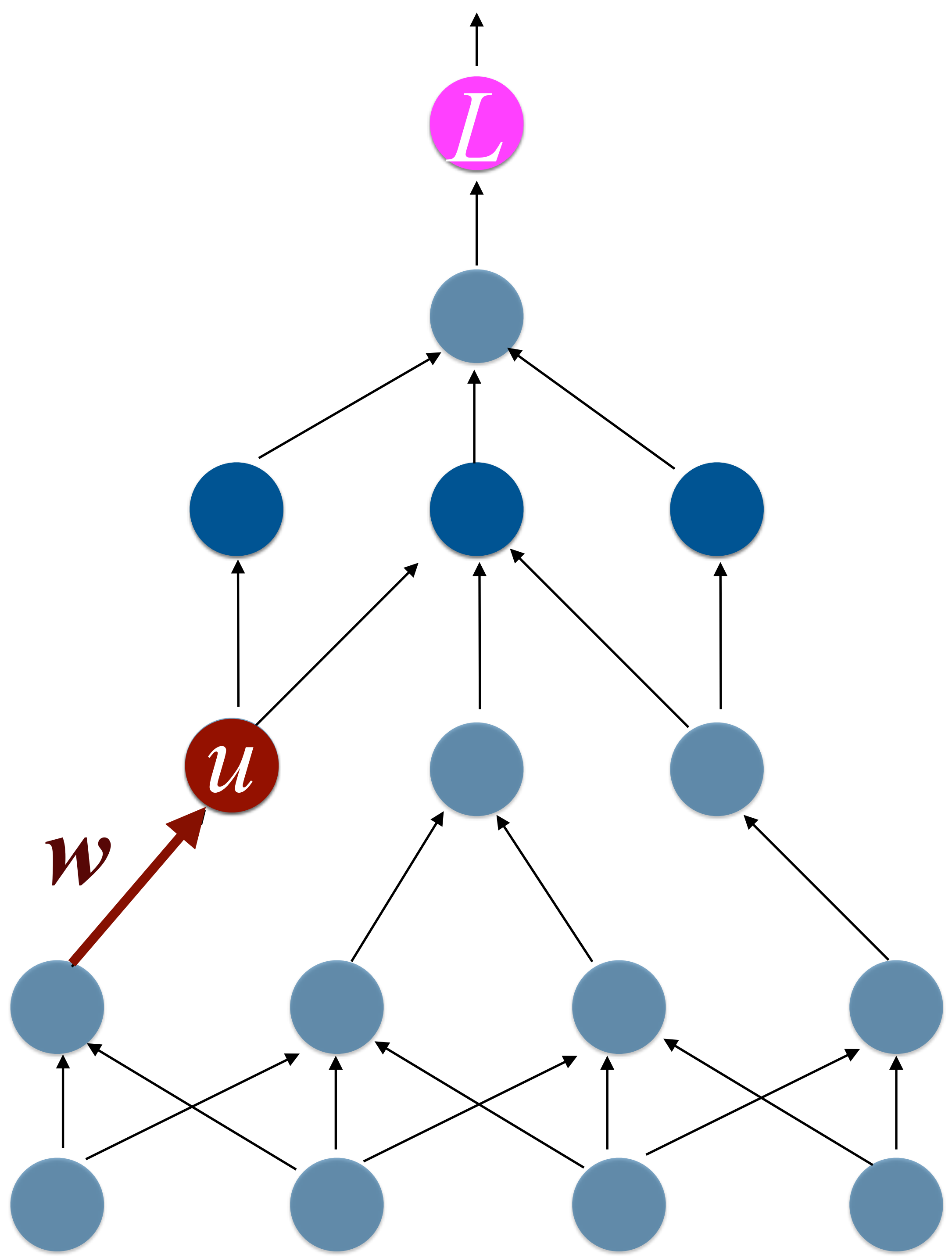
Goal: Learn weights for the neural network which minimise L

Gradient Descent: Find $\partial L/\partial w$ for every weight w, and update it as
$w \leftarrow w - \eta\ \partial L/\partial w$

How do we efficiently compute $\partial L/\partial w$ for all w?

Will compute $\partial L/\partial u$ for every node u in the network!

$\partial L/\partial w = \partial L/\partial u \cdot \partial u/\partial w$ where u is the node which uses w



Training a Neural Network

New goal: compute $\partial L/\partial u$ for every node u in the network

Algorithm: **Backpropagation**

Key fact: Chain rule of differentiation

If L can be written as a function of variables $v_1,\ldots, v_n$, which in turn depend (partially) on another variable u, then

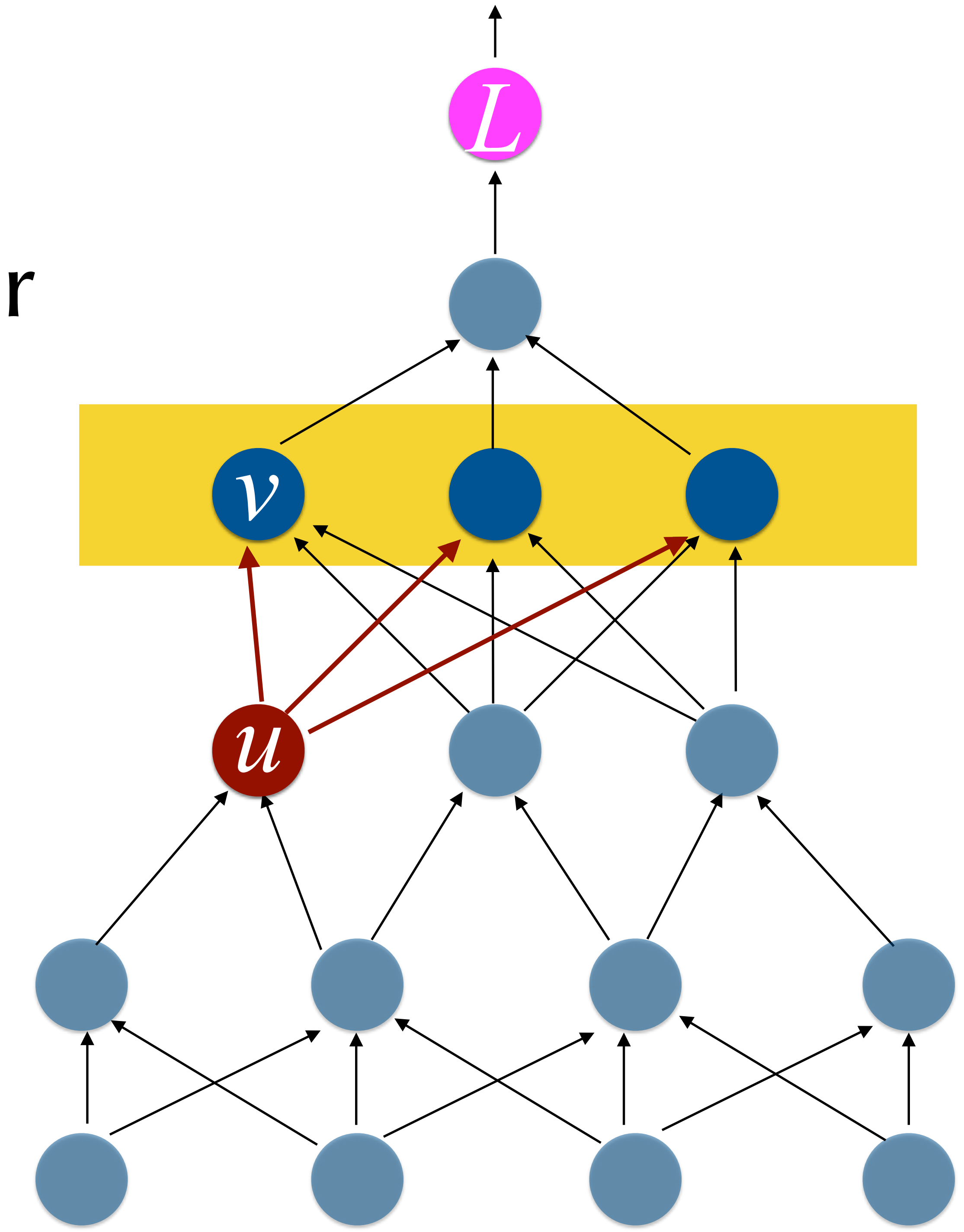
$$\partial L/\partial u = \Sigma_i \, \partial L/\partial v_i \cdot \partial v_i/\partial u$$

Backpropagation

If L can be written as a function of variables $v_1,\ldots, v_n$, which in turn depend (partially) on another variable u, then

$$\partial L/\partial u = \Sigma_i \, \partial L/\partial v_i \cdot \partial v_i/\partial u$$

Consider $v_1,\ldots, v_n$ as the layer above u, $\Gamma(u)$



Then, the chain rule gives

$$\partial L/\partial u = \Sigma_{v \in \Gamma(u)} \; \partial L/\partial v \cdot \partial v/\partial u$$

Backpropagation

$$\partial L/\partial u = \Sigma_{v \in \Gamma(u)}\ \partial L/\partial v \cdot \partial v/\partial u$$

Backpropagation

Base case: $\partial L/\partial L = 1$

For each u (top to bottom):

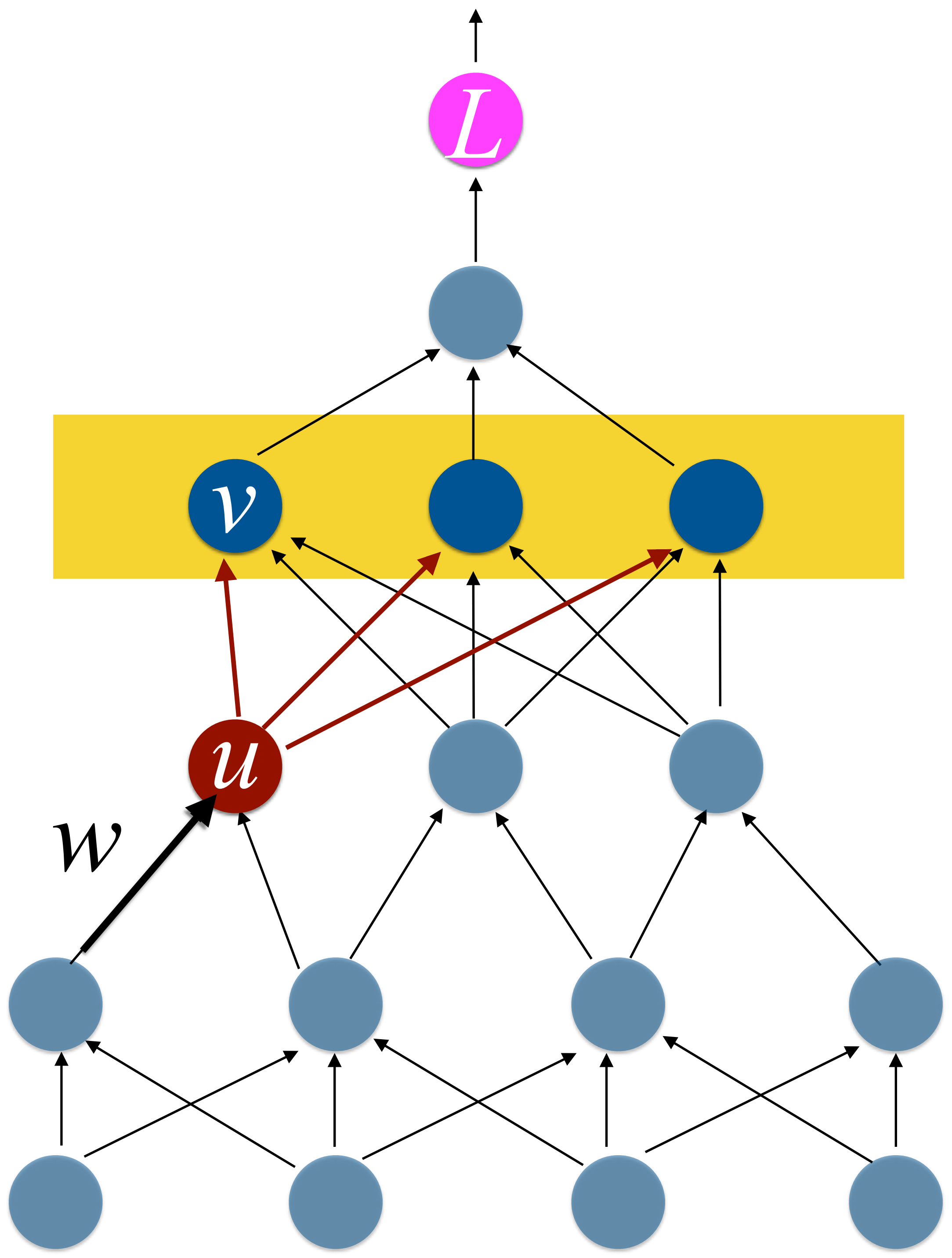
For each $v \in \Gamma(u)$:

Inductively, have computed $\partial L/\partial v$

Directly compute $\partial v/\partial u$

Compute $\partial L/\partial u$

Compute $\partial L/\partial w$ where $\partial L/\partial w = \partial L/\partial u \cdot \partial u/\partial w$



Where values computed in the forward pass are needed

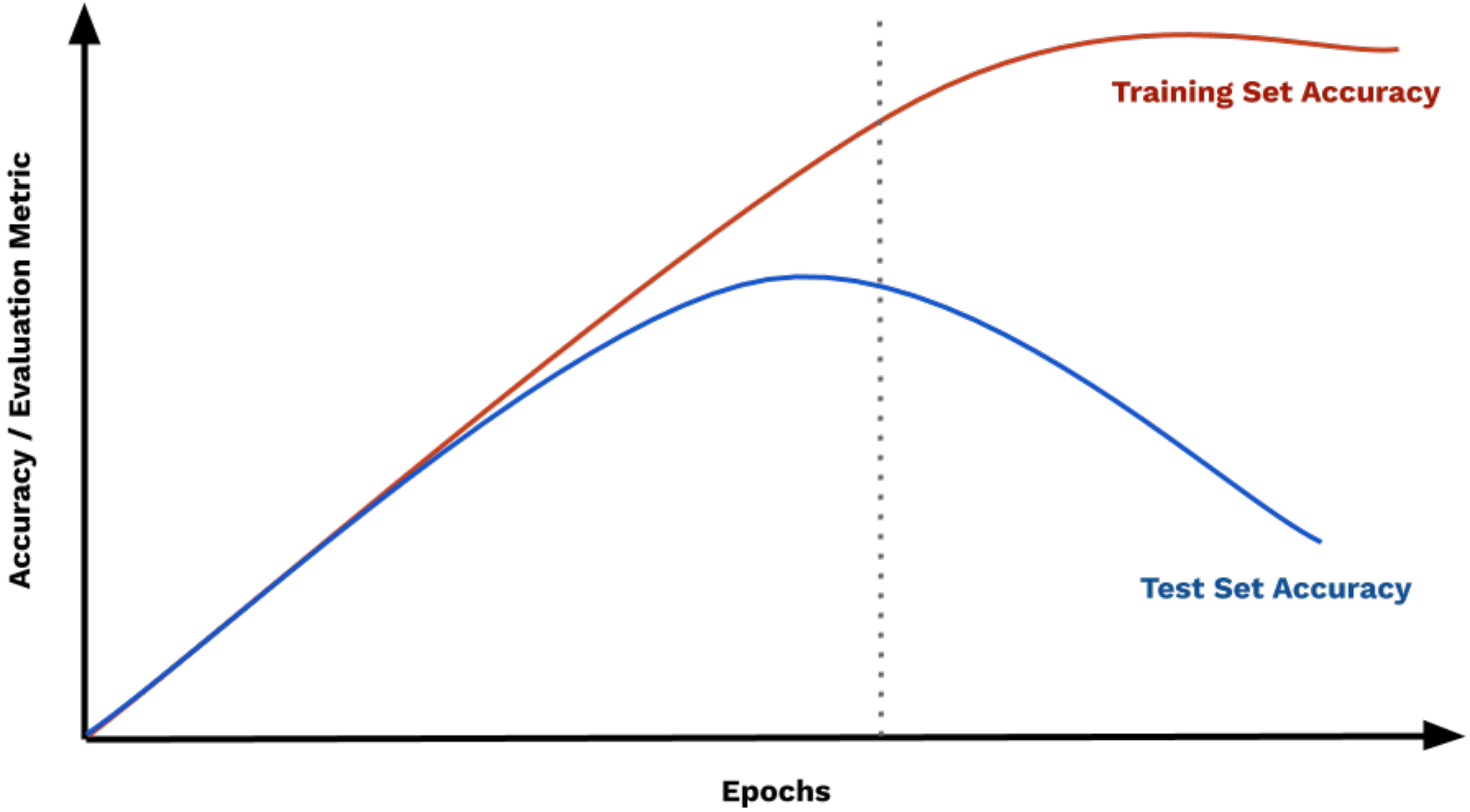
Forward Pass

First, in a forward pass, compute values of all nodes given an input (The values of each node will be needed during backprop)

Useful exercise to compute quantities in the backward pass: https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/

Regularization

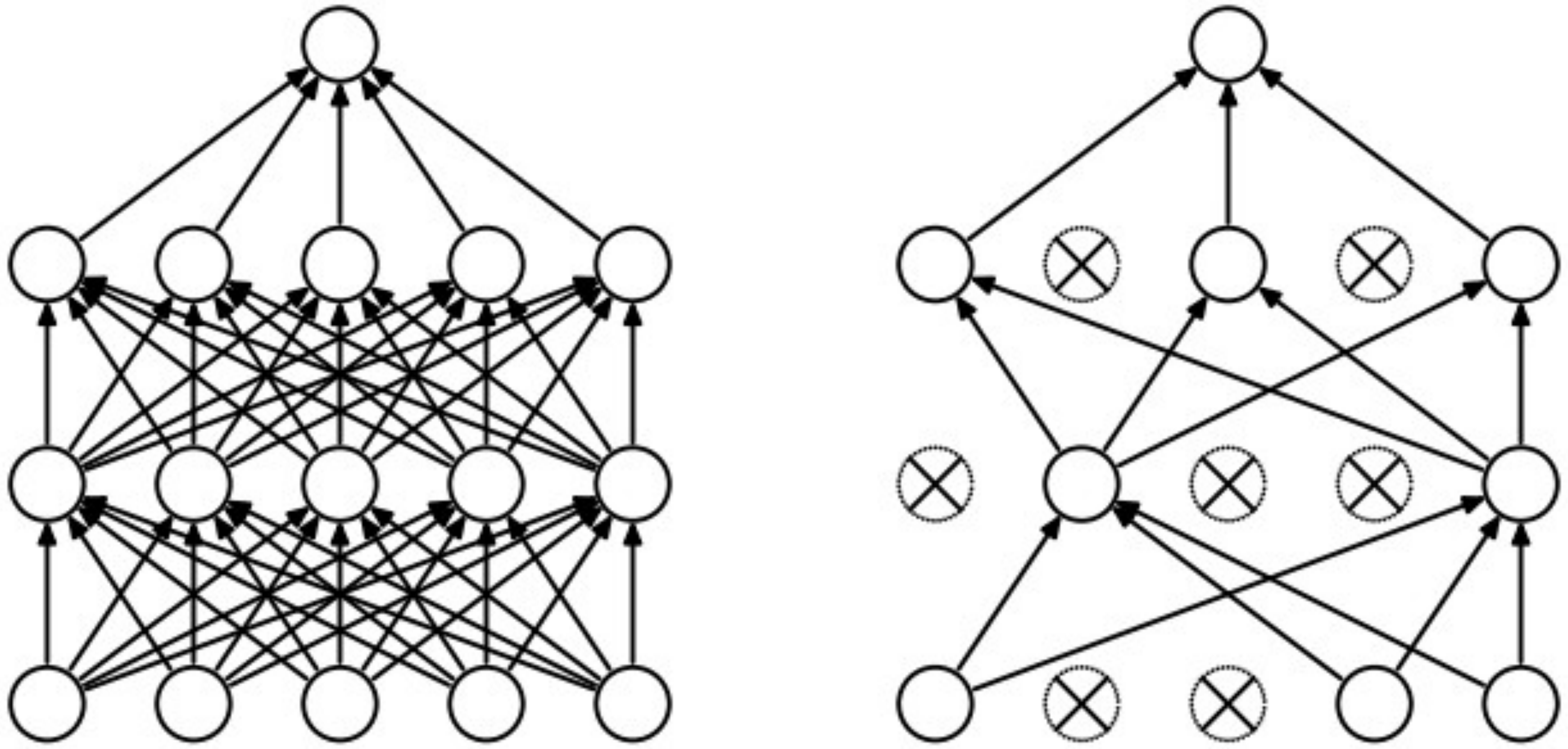
Early stopping: Stop training when performance on a validation set has stopped improving



Regularization

L2 regularization: Introduce a loss term that penalizes the squared magnitude of all parameters. That is, for every weight w in the network, add the term λw^2 to the objective.

Dropout: During training, keep a neuron active with a (keep) probability of p or set it to 0 otherwise. During inference/testing, all units will be present and their outputs will be scaled by a factor of p.



(a) Standard Neural Net

(b) After applying dropout.

Dropout

Dropout trains an ensemble consisting of various subnetworks (constructed by removing non-output nodes from a base network at random)

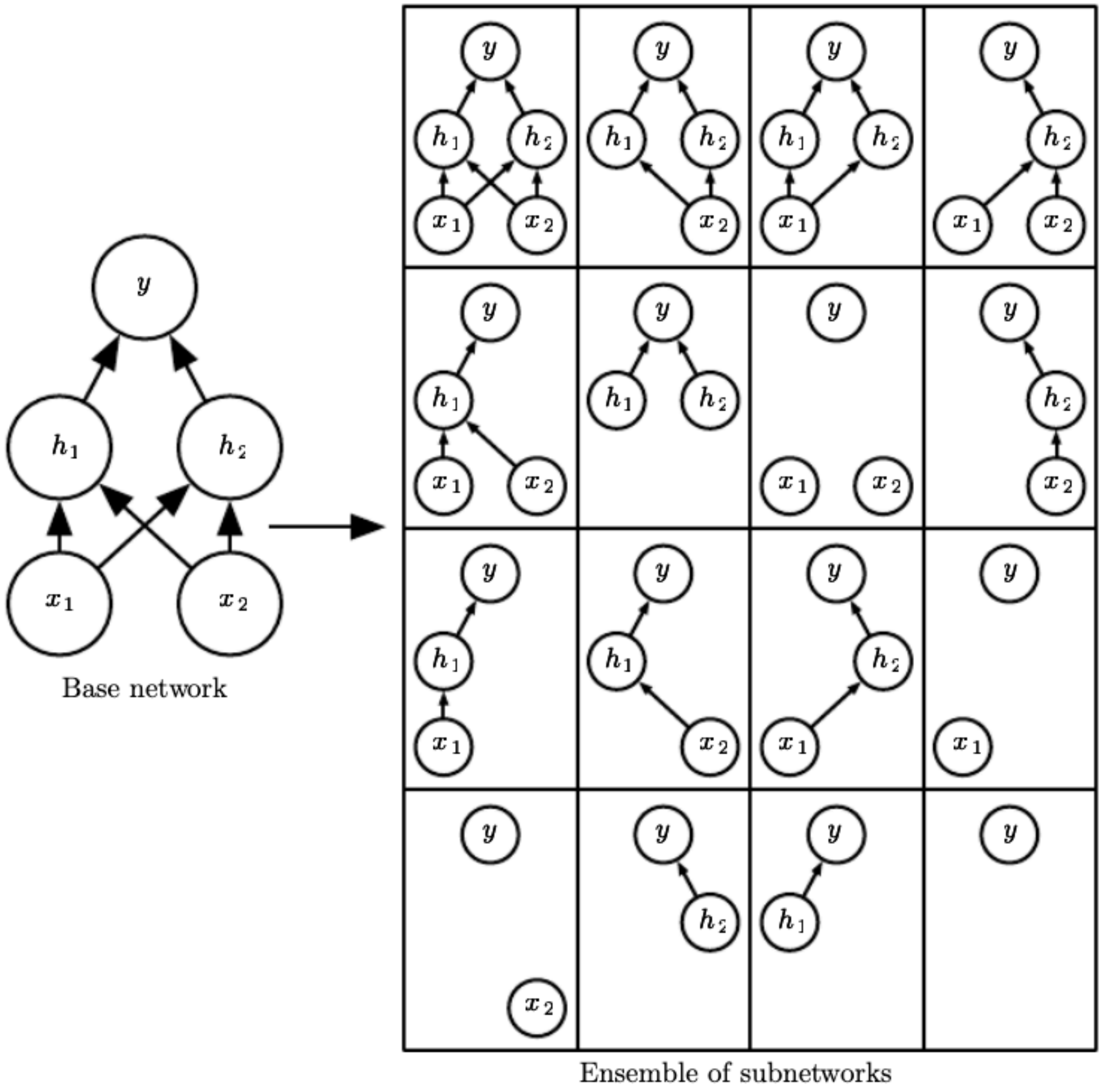


Image from: https://www.deeplearningbook.org/contents/regularization.html

Learning Rate Schedule

- Observe training losses to understand the effect of different learning rates
- Helpful to decay the learning rate over time. E.g. step decay, exponential decay, etc.

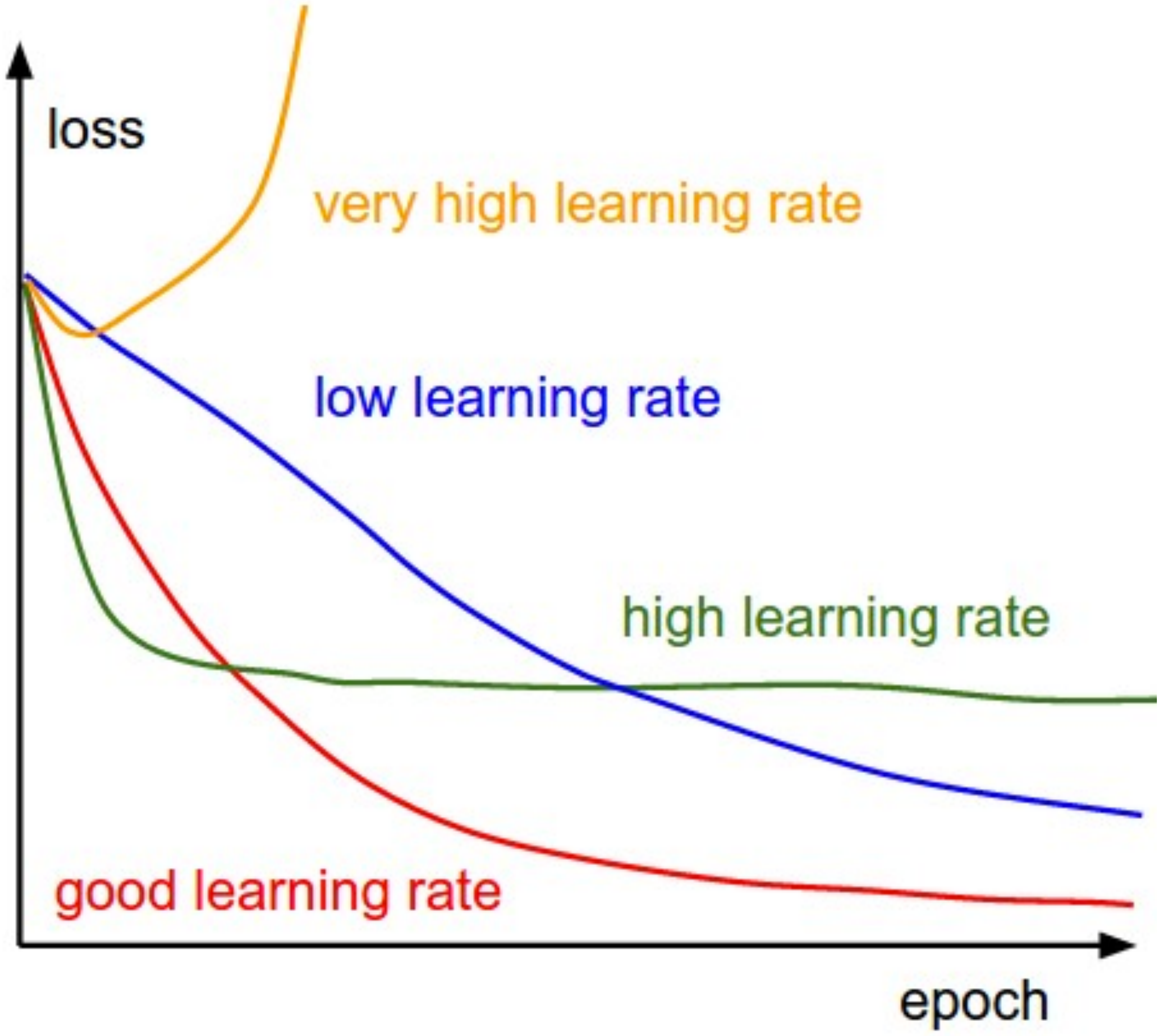
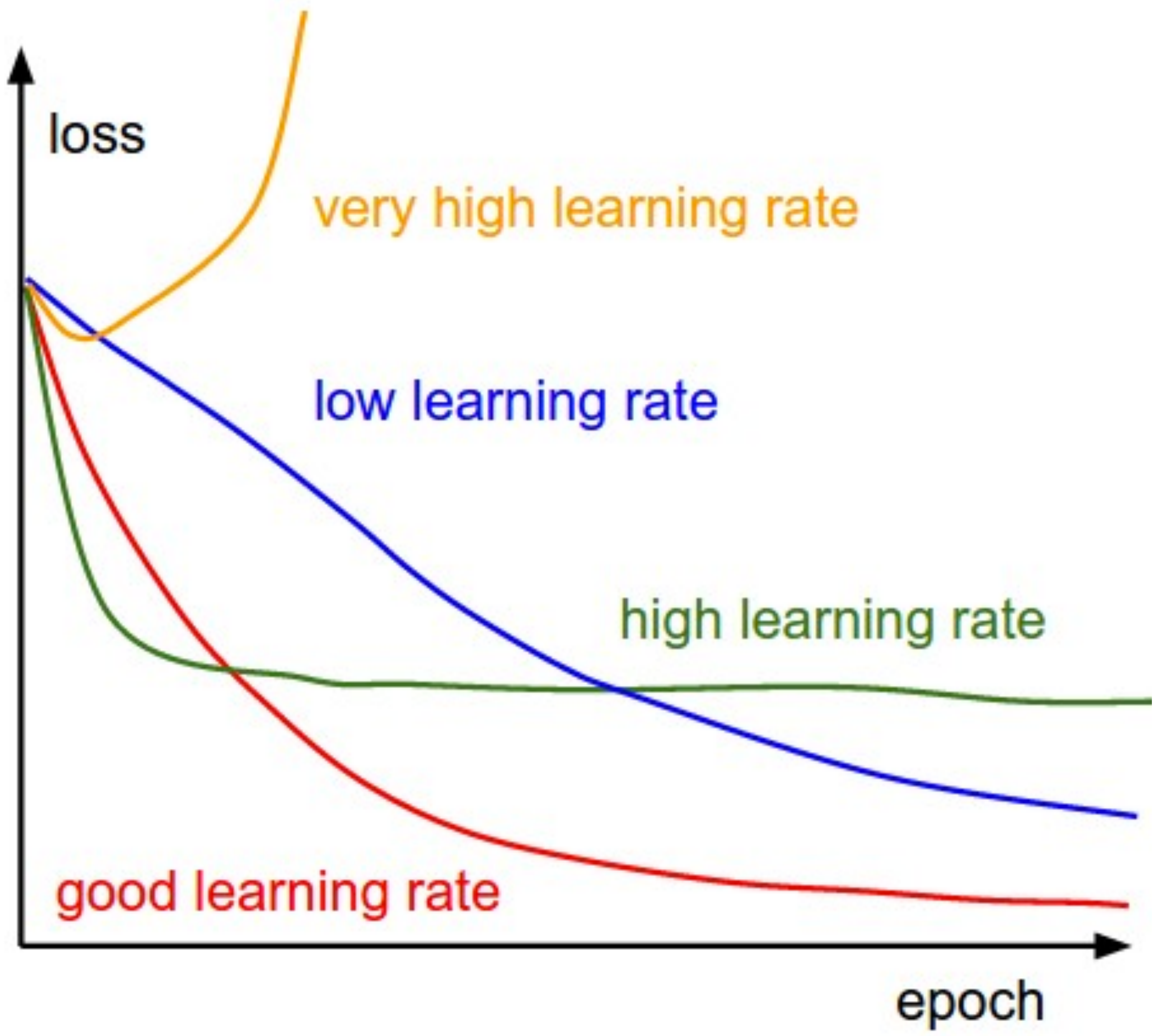


Image from: http://cs231n.github.io/neural-networks-3/

Learning Rate Schedule

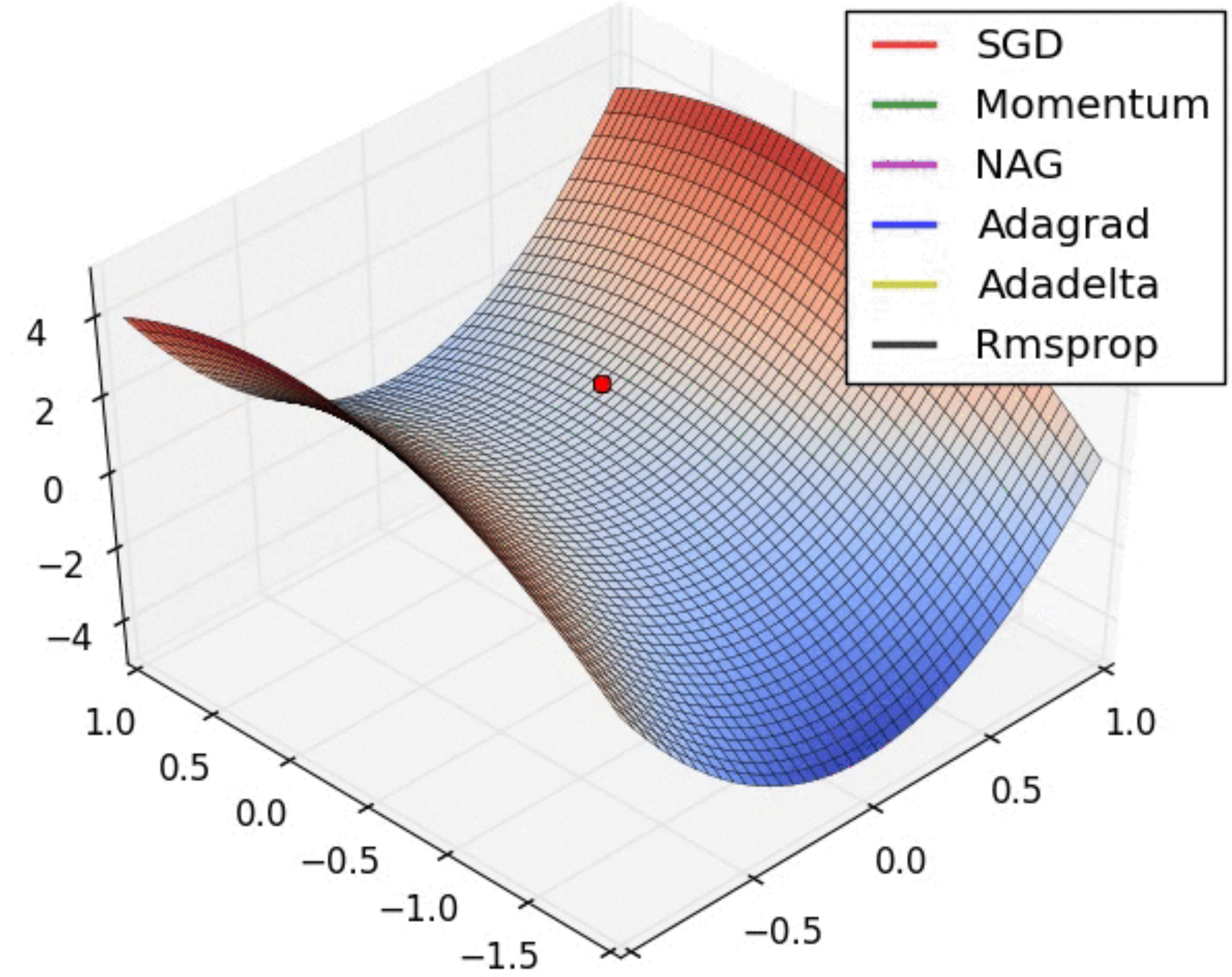
- Observe training losses to understand the effect of different learning rates
- Helpful to decay the learning rate over time. E.g. step decay, exponential decay, etc.
- Adaptive learning rate methods like Adagrad, Adam are popular optimizers.



Animation from: http://cs231n.github.io/neural-networks-3/

Good reference for optimizers: https://ruder.io/optimizing-gradient-descent/

Illustration



Images credit: Alec Radford.