Introduction to Machine Learning (CS419M)

Lecture 12:

- Regularization and Optimization of Neural Networks
- Introducing CNNs

Mar 11, 2020

Project Topics

Object detection and classification

Deep flow based generative models for text prediction

Hourly/Daily/Weekly Electricity Consumption Prediction

Exploration and Comparison of Reinforcement Learning Algorithms on OpenAI Gym Environment

Football Match Outcome prediction

Investigating Enron Scandal

Super-Resolution with Cascading Residual Network

M5 Forecasting - Accuracy: Estimate the sales of Walmart goods

ANDEC - Audio Number Detection

Credit Card Fraud Detection

Twitter Emotion Analysis Classifier

Retrieval of Image using Captions

Handwritten character recognition

Cancer prognosis and post diagnosis life expectancy

Human Activity Recognition using wearable sensor data

Predict Students Performance in Exams

Speech-to-Text using CNN

Movie Box Office Prediction System

Applications of Machine Learning in Power Engineering

Panic of Corona and Financial Volatility

Classifying Leukemia based on patient’s genes

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

Simple 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

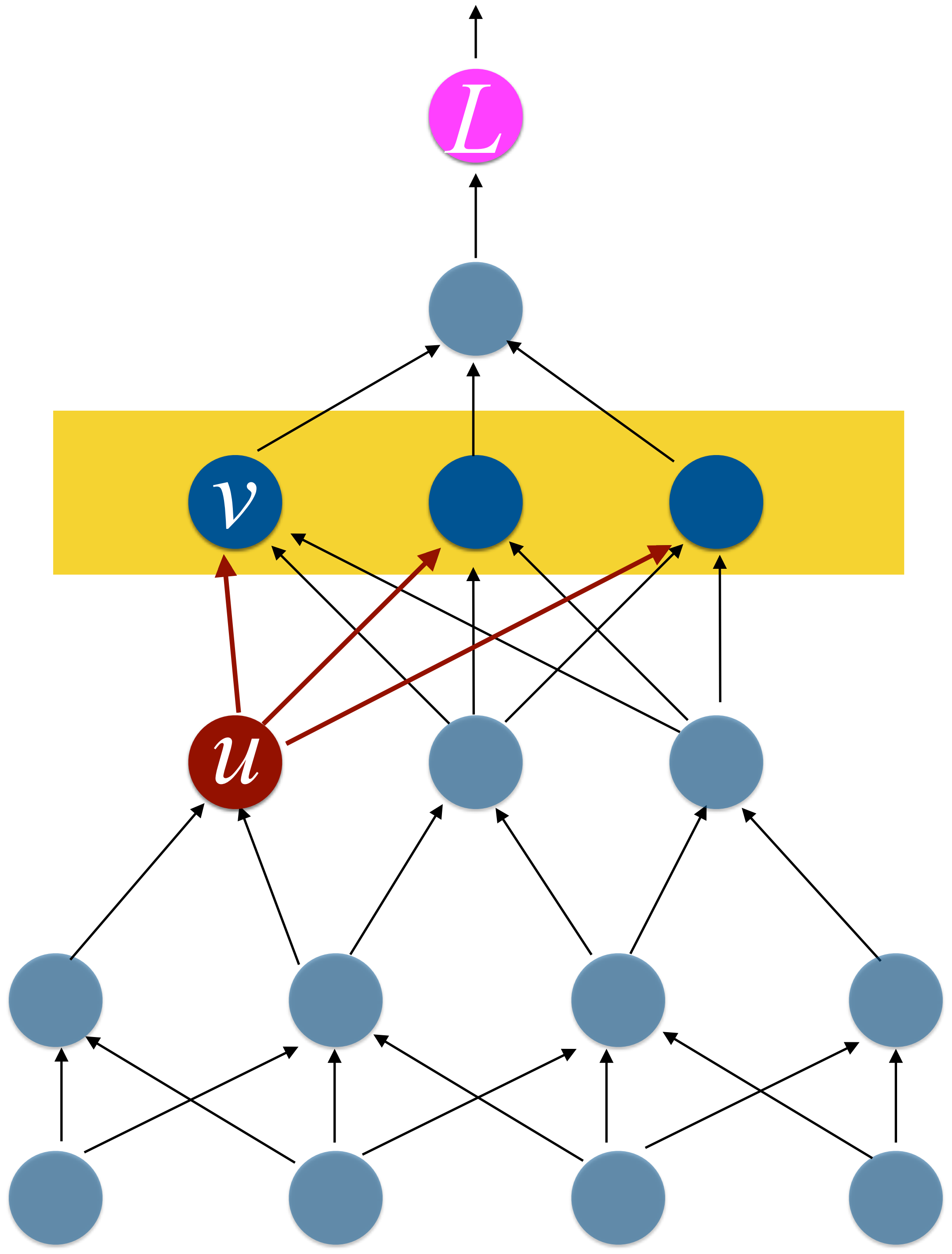
$$\partial L/\partial u = \Sigma_i\, \partial L/\partial v_i \cdot \partial v_i/\partial u$$

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

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

<u>Backpropagation</u>

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

For each u (top to bottom):

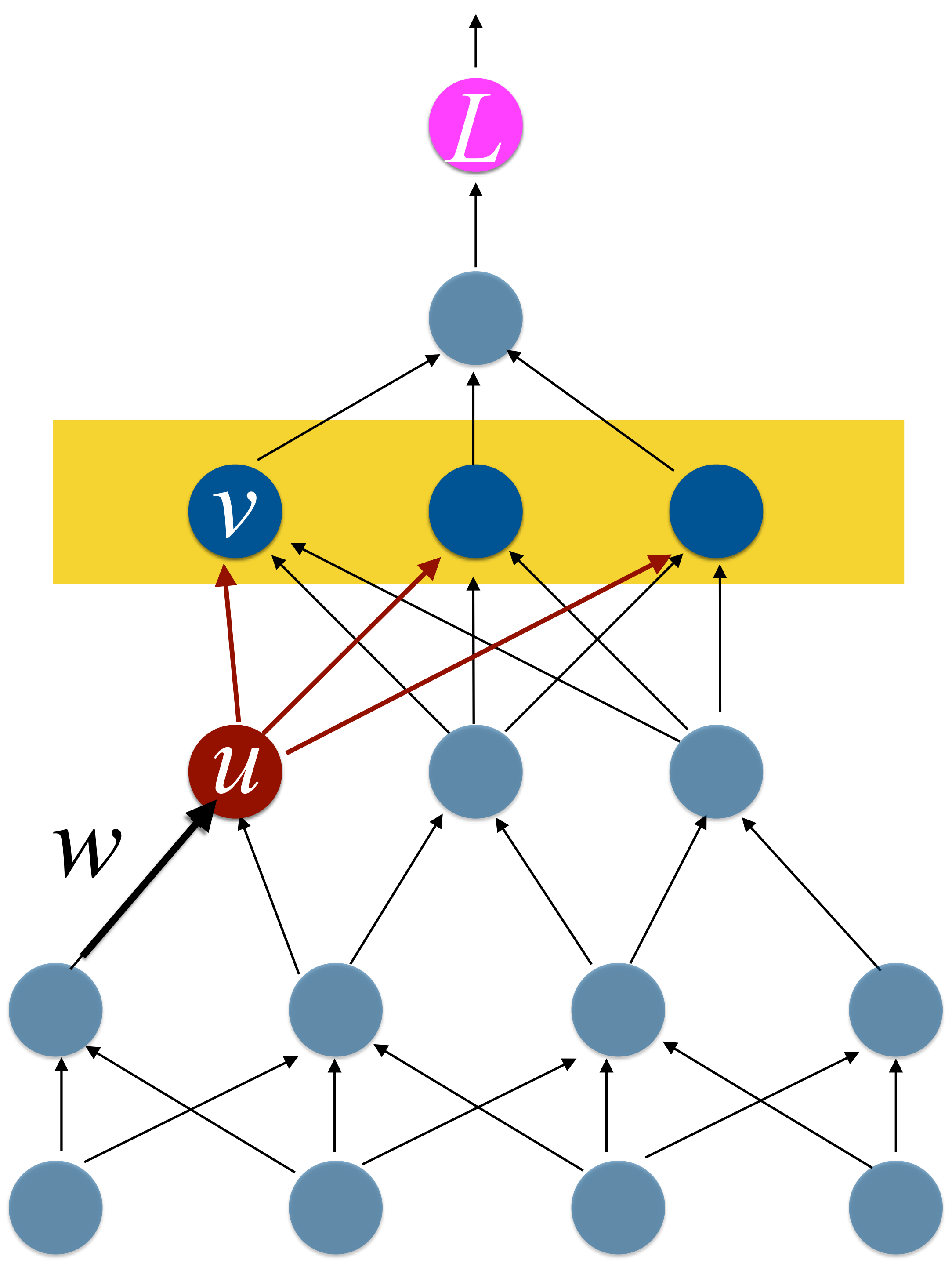
For each $v \in \Gamma(u)$:

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 may be needed

<u>Forward Pass</u>

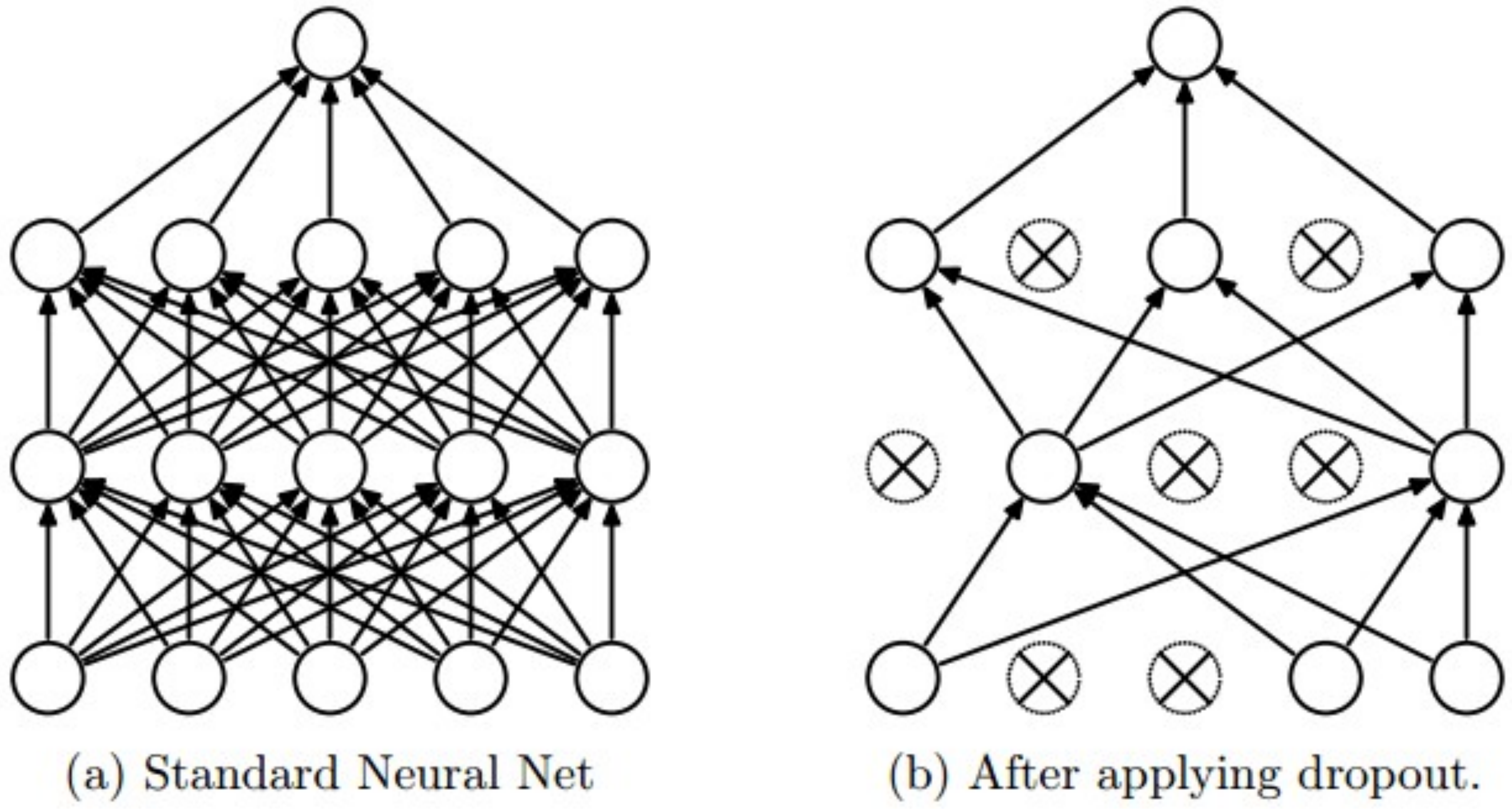
First, in a forward pass, compute values of all nodes given an input
(The values of each node will be needed during backprop)

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

Regularization

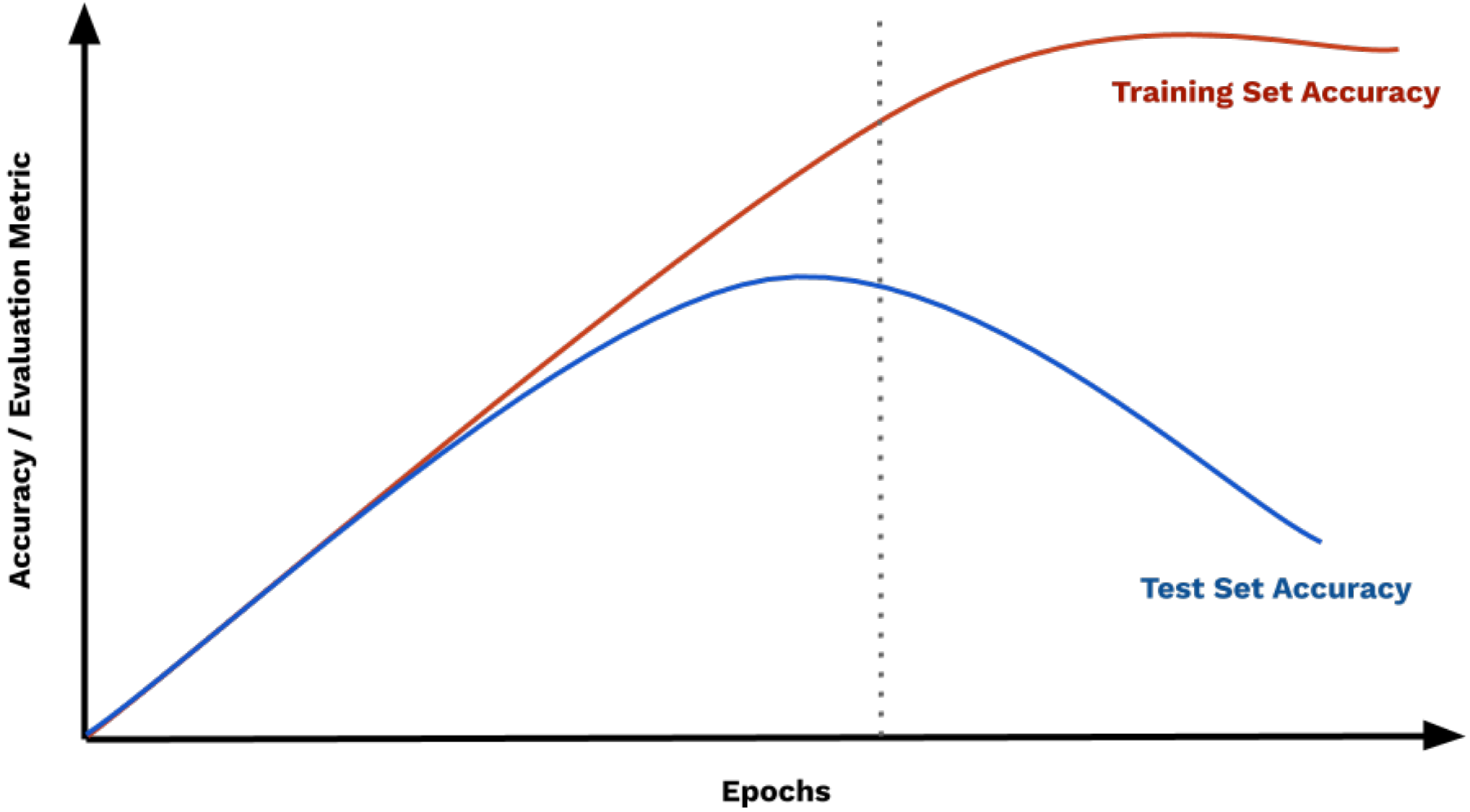
L2 regularization: Introduce a loss term that penalizes the squared magnitude of all parameters. That is, for every weight w in the network, add the term λw^2 to the objective.

Dropout: During training, keep a neuron active with a (keep) probability of p or set it to 0 otherwise.



Regularization

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



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.

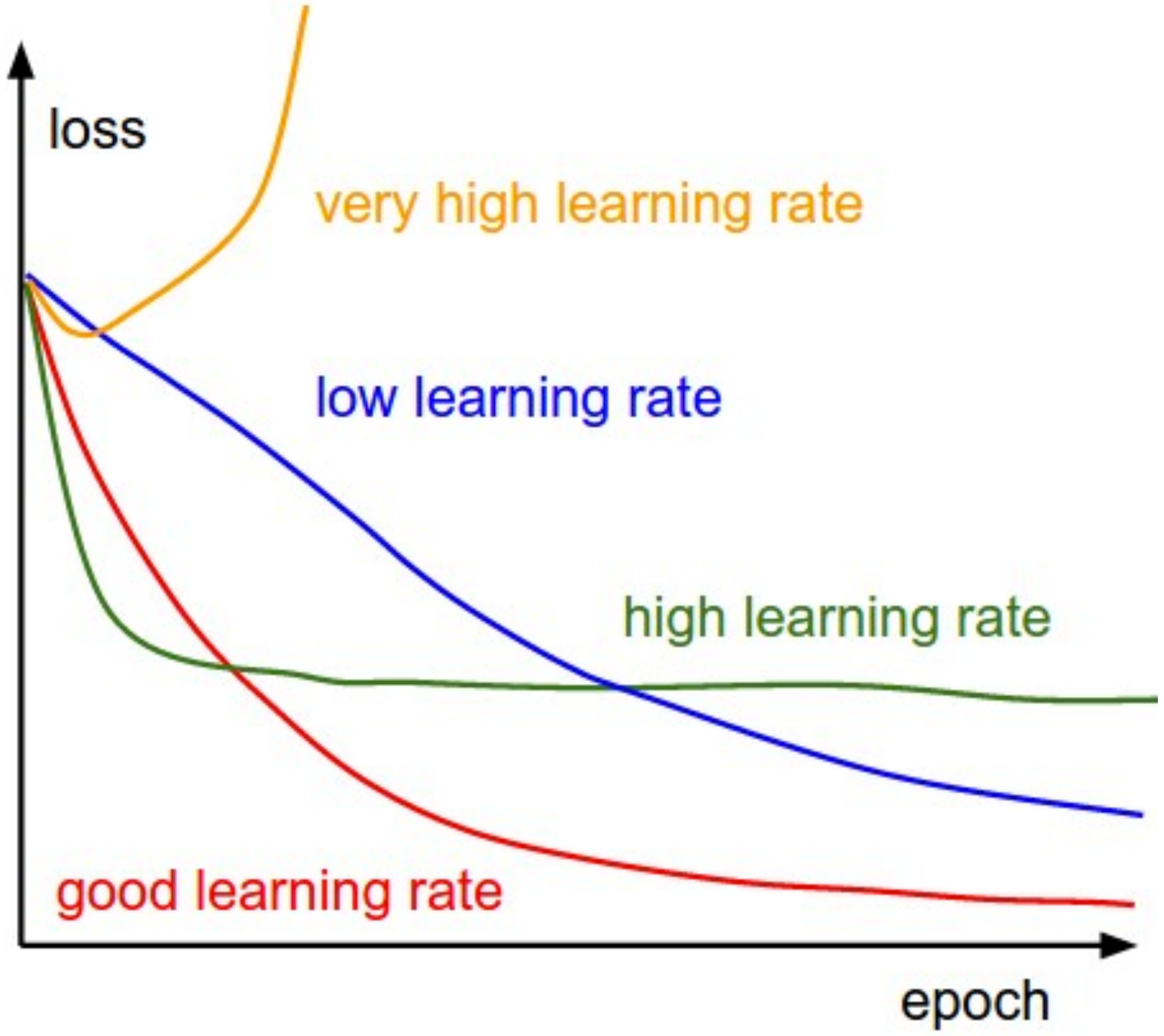
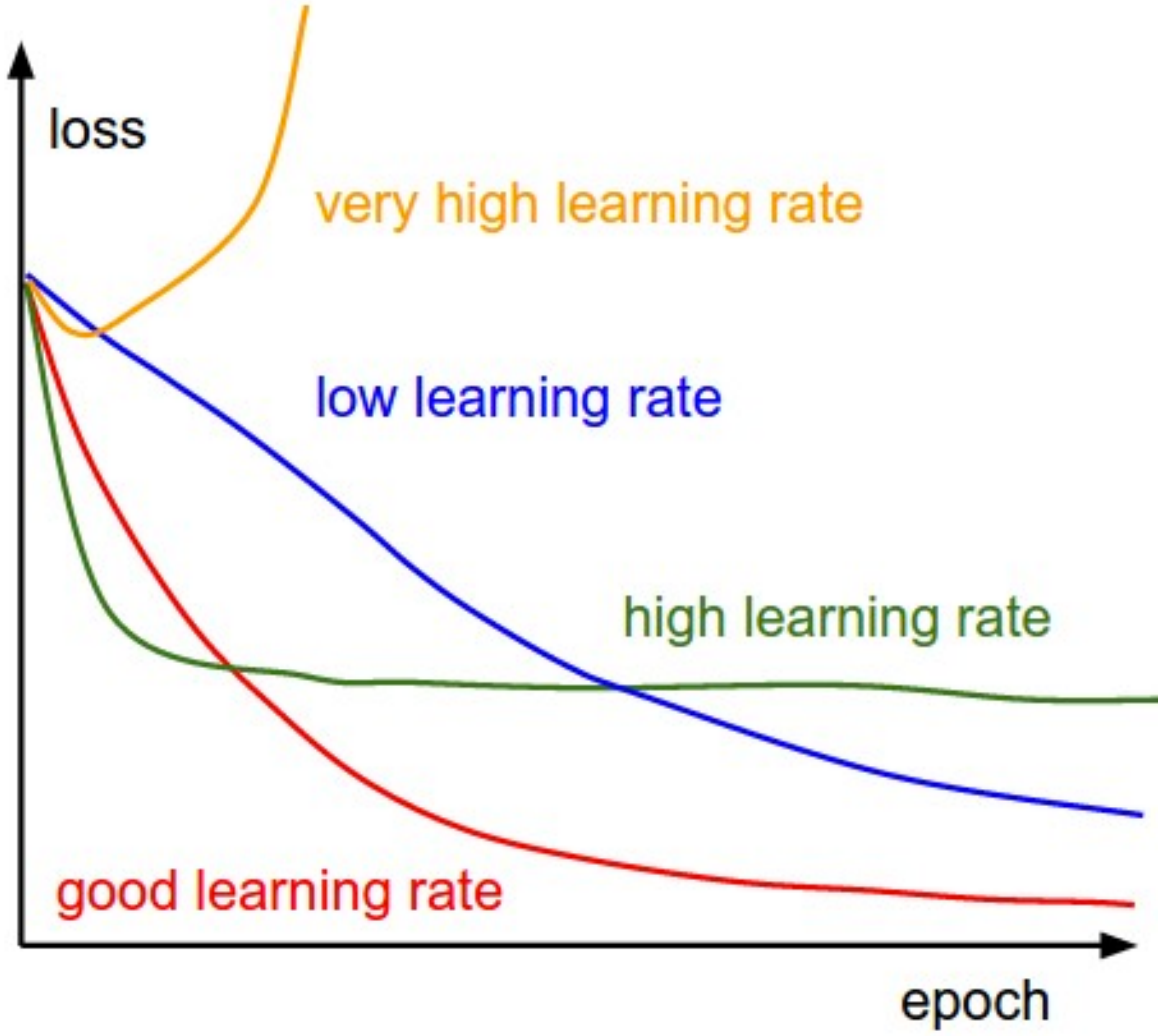


Image from: http://cs231n.github.io/neural-networks-3/

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/

Optimization Algorithms (I)

- "**SGD with Momentum**" weight update rule:

$$\mathbf{v}_t = \beta \mathbf{v}_{t-1} + (1-\beta)\nabla_{\mathbf{w}_t} L(\mathbf{w}_t) \qquad \text{for } 0 \leq \beta < 1$$

$$\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t - \eta \mathbf{v}_t$$

- Smooths parameter updates with exponentially decaying weights

Optimization Algorithms (II)

- "**RMSProp (Root Mean Squared Propagation)**" weight update rule:

$$\mathbf{s}_t = \gamma \mathbf{s}_{t-1} + (1-\gamma)\mathbf{g}_t \odot \mathbf{g}_t$$

$$\mathbf{g}_t = \nabla_{\mathbf{w}_t} L(\mathbf{w}_t)$$

$$\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t - \frac{\eta}{\sqrt{\mathbf{s}_t} + \epsilon} \odot \mathbf{g}_t$$

Element-wise multiplication

- Need an adaptive learning rate that adapts to each dimension.

Optimization Algorithms (III)

- "**Adam**" weight update rule: Makes use of both momentum and adaptive learning rate

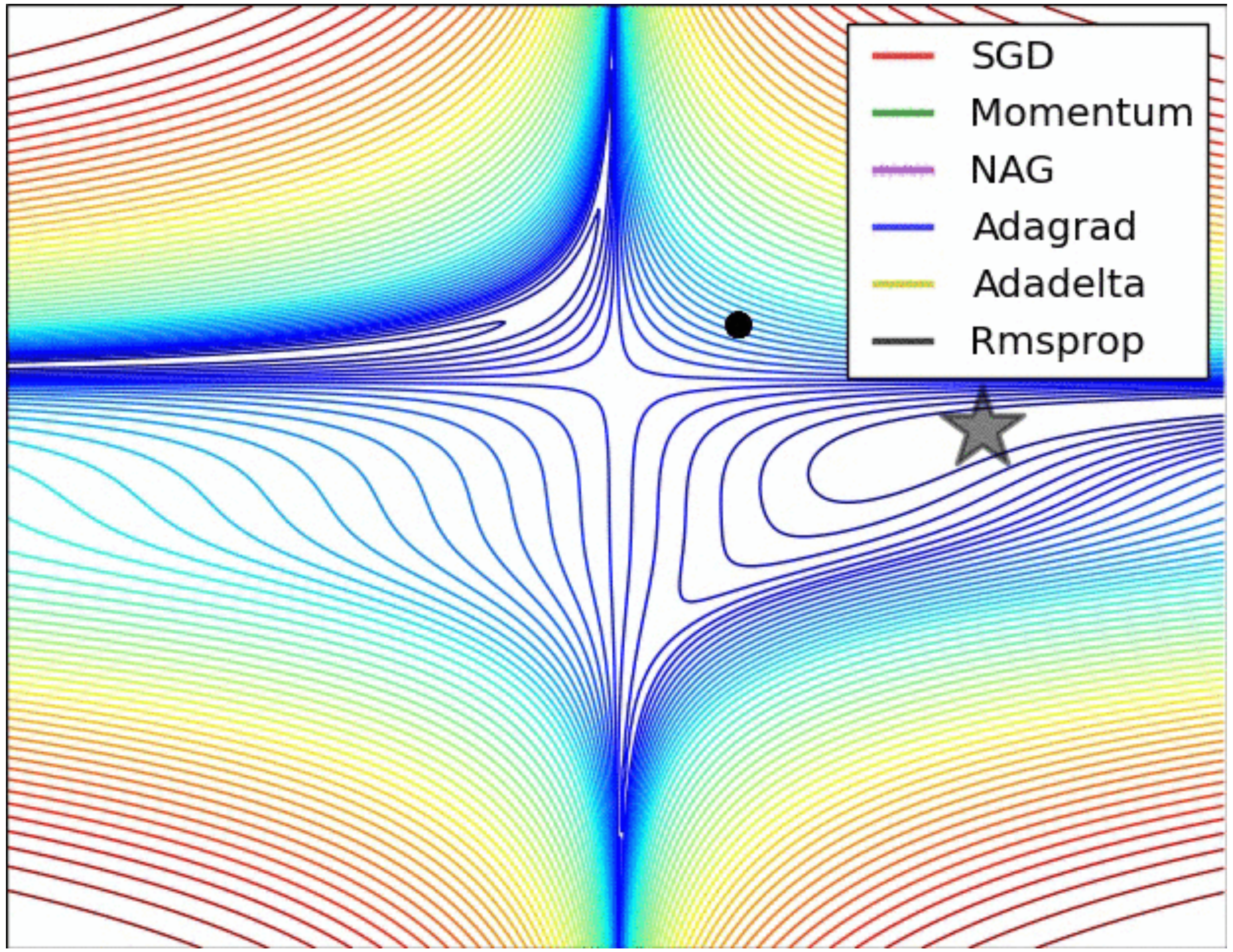
$$\mathbf{v}_t = \beta \mathbf{v}_{t-1} + (1-\beta)\mathbf{g}_t$$

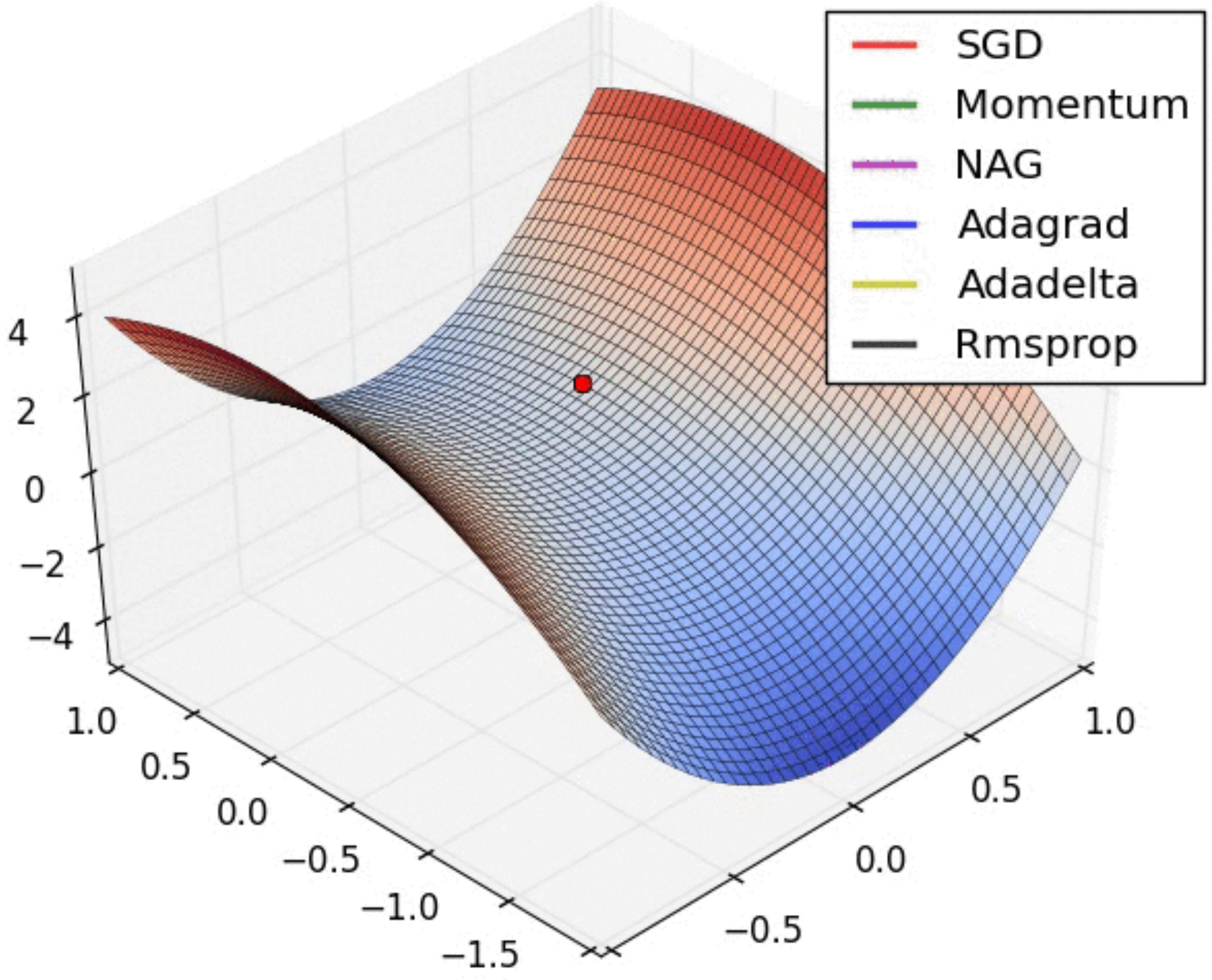
$$\mathbf{s}_t = \gamma \mathbf{s}_{t-1} + (1-\gamma)\mathbf{g}_t \odot \mathbf{g}_t$$

$$\hat{\mathbf{s}}_t \leftarrow \frac{\mathbf{s}_t}{1-\gamma^t} \qquad \hat{\mathbf{v}}_t \leftarrow \frac{\mathbf{v}_t}{1-\beta^t}$$

$$\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t - \frac{\eta \hat{\mathbf{v}}_t}{\sqrt{\hat{\mathbf{s}}_t} + \epsilon}$$

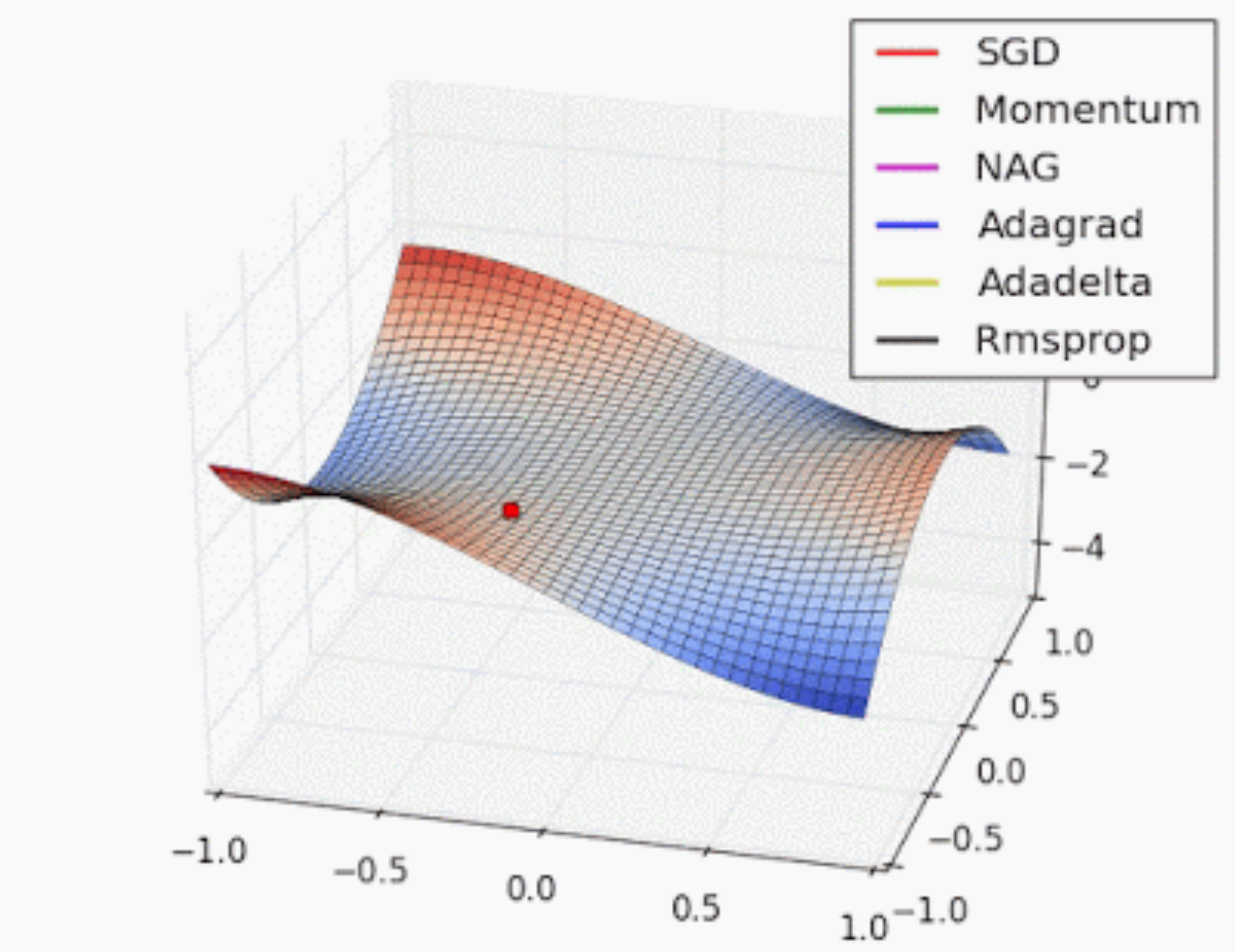
Illustration





Images credit: Alec Radford.

Illustration



Images credit: Alec Radford.