# Neural Networks

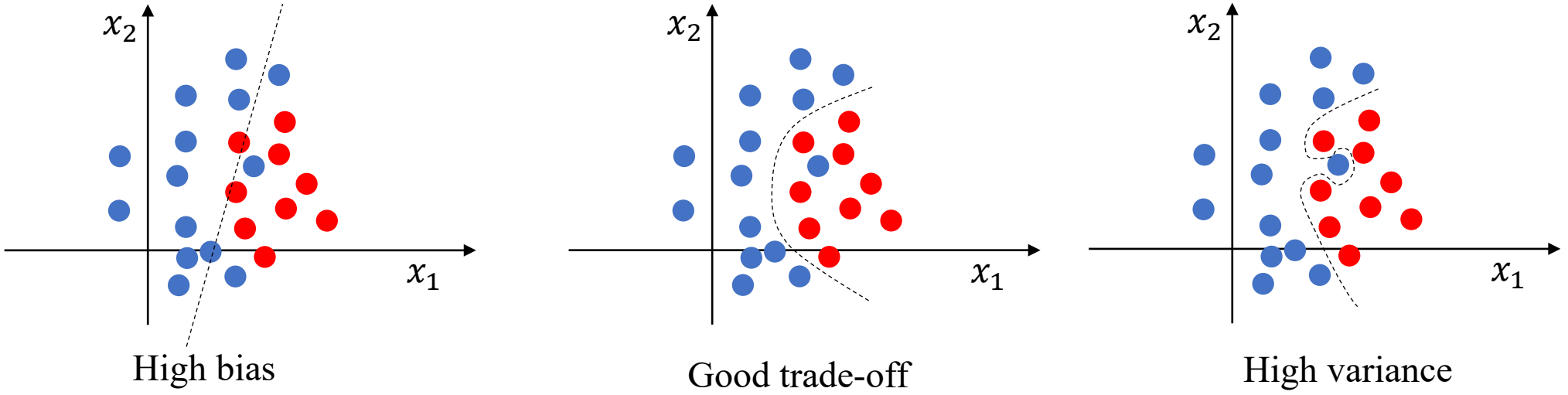
Kaiqi Zhao
The University of Auckland

*Slides are partially based on the materials from Stanford University and UC Berkeley*

# Regularization

# Overfitting

- Recall the fundamental trade-off




- Neural networks are easy to overfit because of its model complexity and non-linearity

# Regularization

- Regularization is a technique to mitigate the overfitting problem
- In its simplest form, if we have a minimization problem:

$$\min_w J(w) = \min_w \sum_{i=1}^{m} \mathcal{L}(\hat{y}_i, y_i; w)$$

Where $\mathcal{L}(\hat{y}_i, y_i; w)$ is the loss for each instance. For classification, it could be cross-entropy, for regression, it could be sum of square errors.

- Regularization is to add a penalty $R(w)$ to large values of $w$

$$\min_w J_R(w) = \min_w \sum_{i=1}^{m} \mathcal{L}(\hat{y}_i, y_i; w) + \lambda R(w)$$

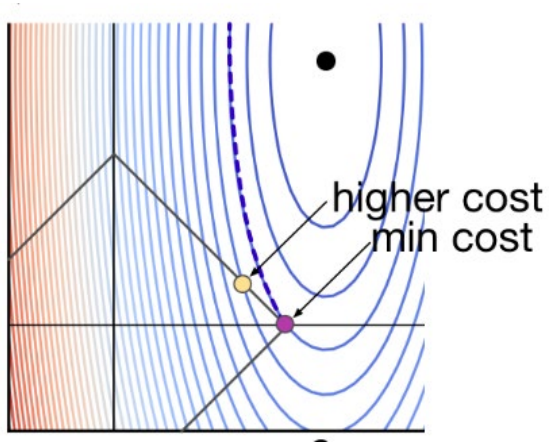
The hyperparameter $\lambda$ controls the strengths of the penalty

# Regularization

- Regularization is to add a penalty $R(w)$ to large values of $w$

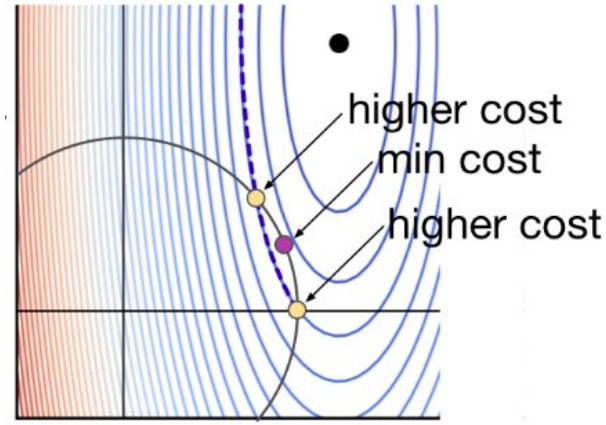
$$\min_w J_R(w) = \min_w \sum_{i=1}^{m} \mathcal{L}(\hat{y}_i, y_i; w) + \lambda R(w)$$

- L1 regularization:
- $R(w) = \sum_j |w_j|$




L1 is often sparser

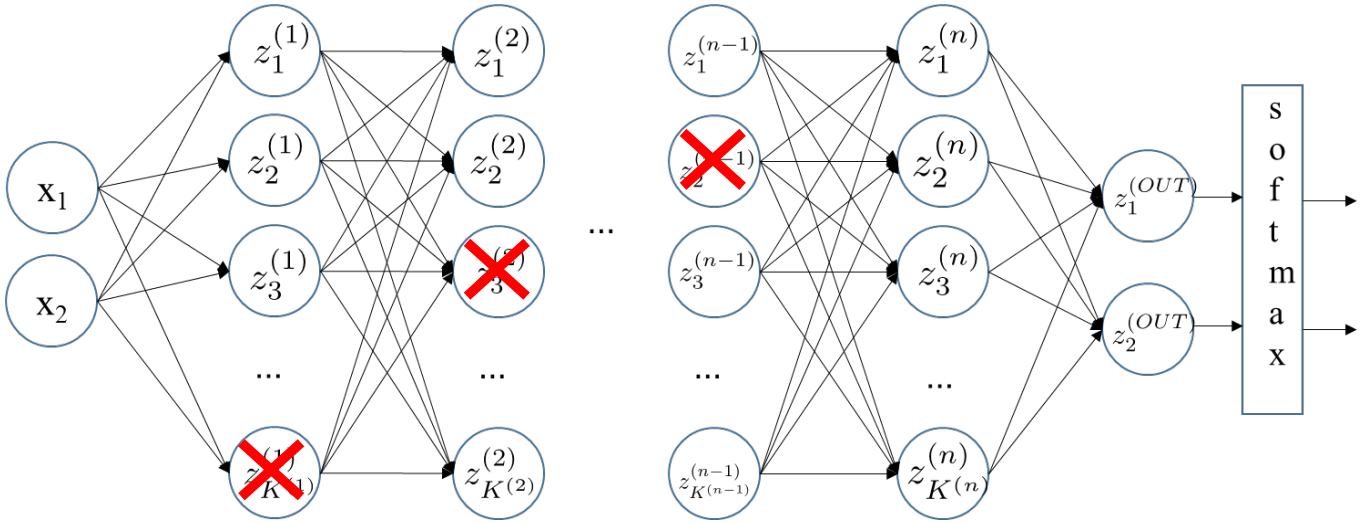
- L2 regularization:
- $R(w) = \|w\|^2 = w^T w = \sum_j w_j^2$




[Image from https://explained.ai/regularization/L1vsL2.html]

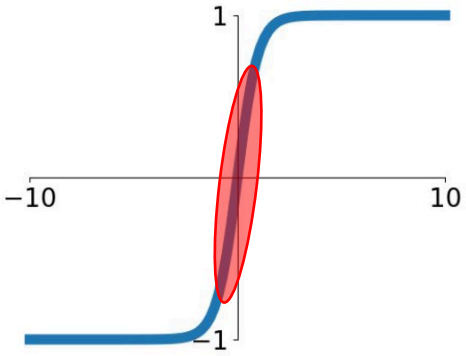
# Why Regularization Reduce Overfitting?

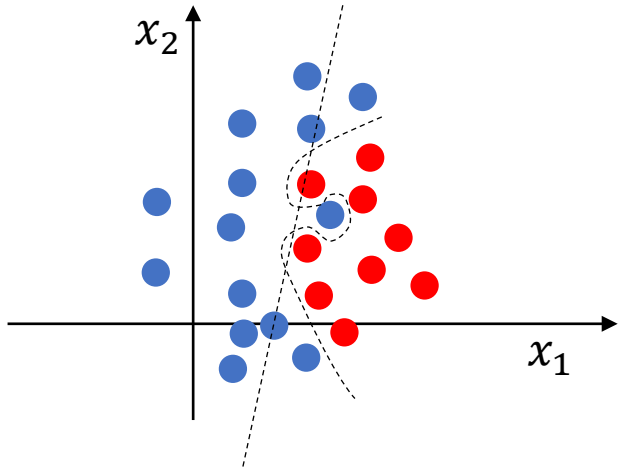
- The general idea of regularization is to keep the weights small
- Why does it help?
  - When the weights get close to zero, you can think of some of the neurons are deactivated. This simplifies your model.



# Why Regularization Reduce Overfitting?

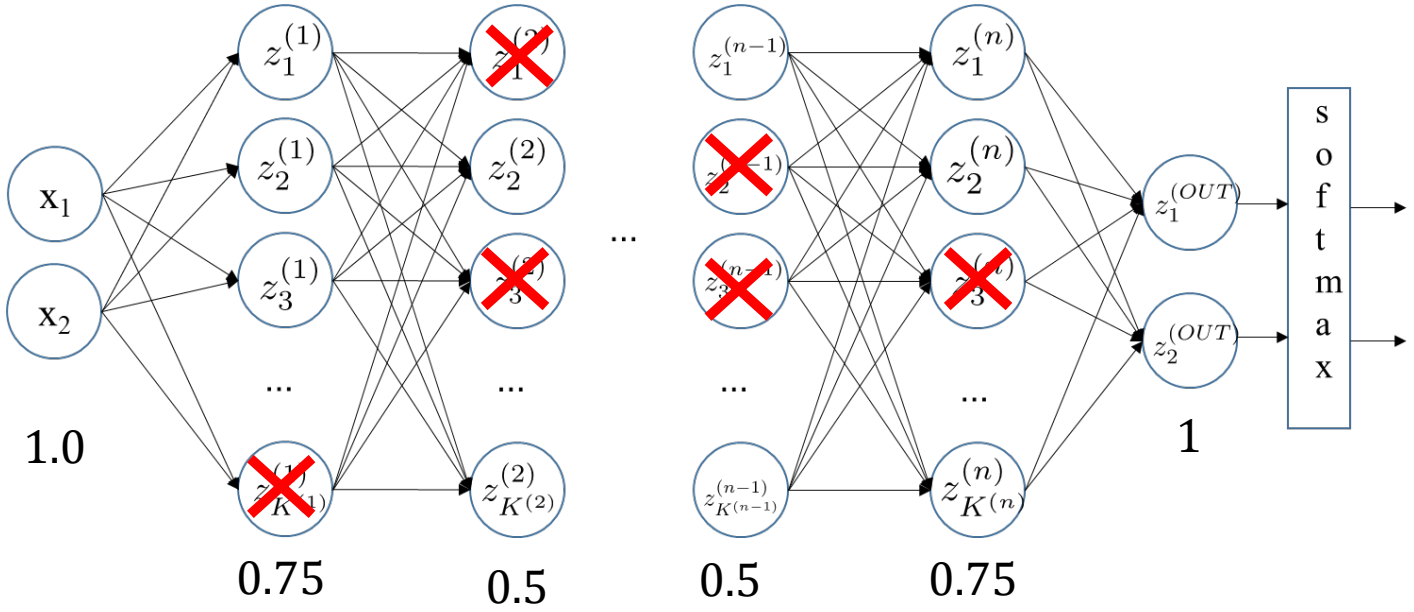
- The general idea of regularization is to keep the weights small
- Why does it help?
  - When the weights get close to zero, you can think of some of the neurons are deactivated. This simplifies your model.
  - For some activation function like tanh and sigmoid, small weights around zero makes it closer to a (simple) linear model.



# Other Regularization Techniques

- Dropout – not to rely on a few features, spread out the weights
  - In the **training phase**, randomly deactivate some of the nodes in each instance/batch/mini-batch
  - In the **test phase**, we use the whole network without dropout.



# Other Regularization Techniques

- Dropout
  - Randomly deactivate some of the nodes in each instance/batch/mini-batch
  - How to implement dropout?
    - Let's say you want to keep 80% of your neurons at a layer
    - For each neuron at the layer
      - Generate a random number in the range [0,1] and compare it with 0.8
      - If the random number is smaller than 0.8 keep the neuron, otherwise, multiply the output of the neuron with 0 (i.e., deactivation).
      - Inverted dropout - divide the output of each neuron at this layer by 0.8. Why??

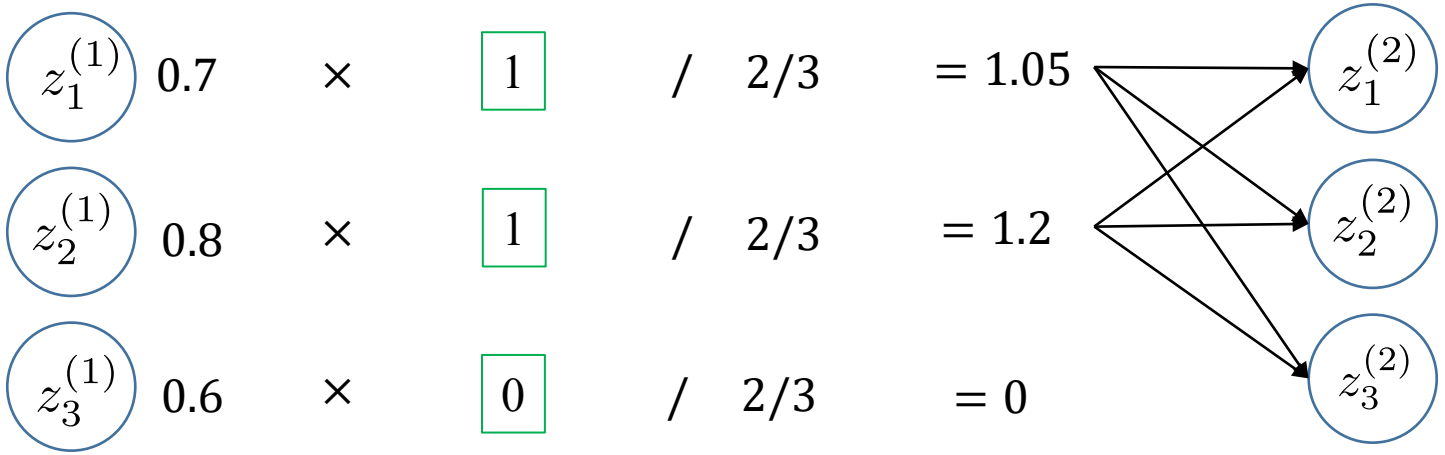
$$z_i^{(k)} = g(\sum_j W_{i,j}^{(k-1,k)} z_j^{(k-1)})$$

- Remind that in the test phase, we use the whole network
- The values for these neurons are 20% less than usual. Rescale it back to make sure both training and test phases have similar scale.

# Other Regularization Techniques

- Dropout - Example

Keep prob = $2/3$




Keep/drop randomly

# Other Regularization Techniques

- Data Augmentation
  - Data transformation - rotation, zoom in/out
  - Add small noise

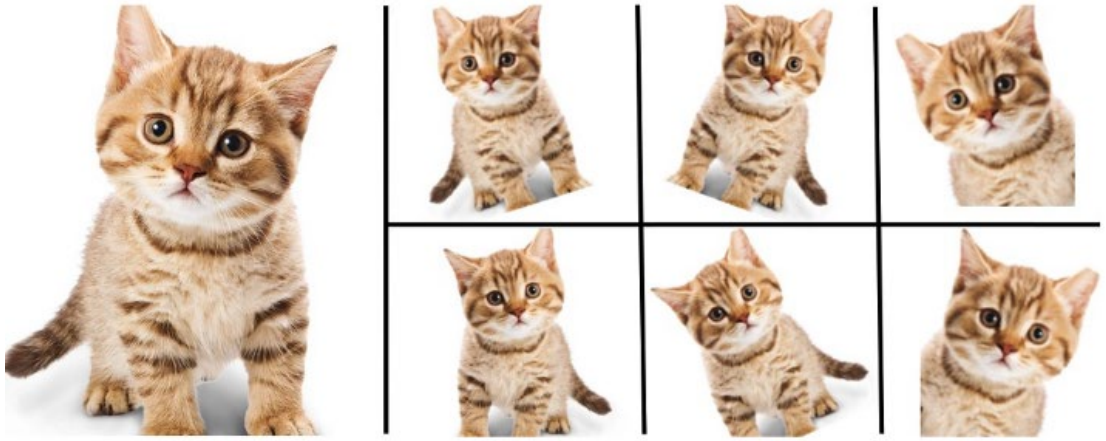
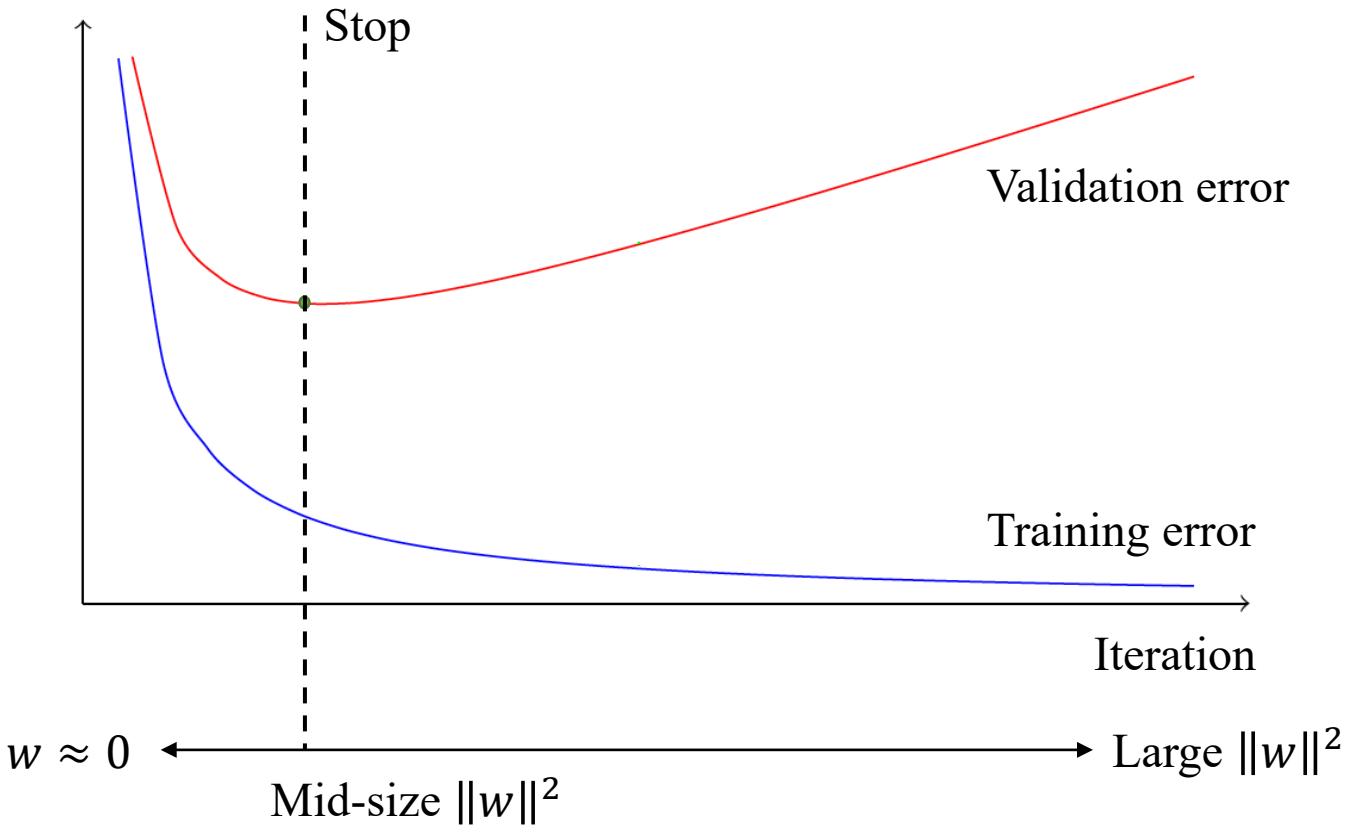


Image source: https://nanonets.com/blog/data-augmentation-how-to-use-deep-learning-when-you-have-limited-data-part-2/

# Other Regularization Techniques

- Early stopping – Stop when the validation error gets worse



# Optimization

# SGD

We often use (mini-batch) SGD for training a neural network. Let us consider a generate optimization problem:

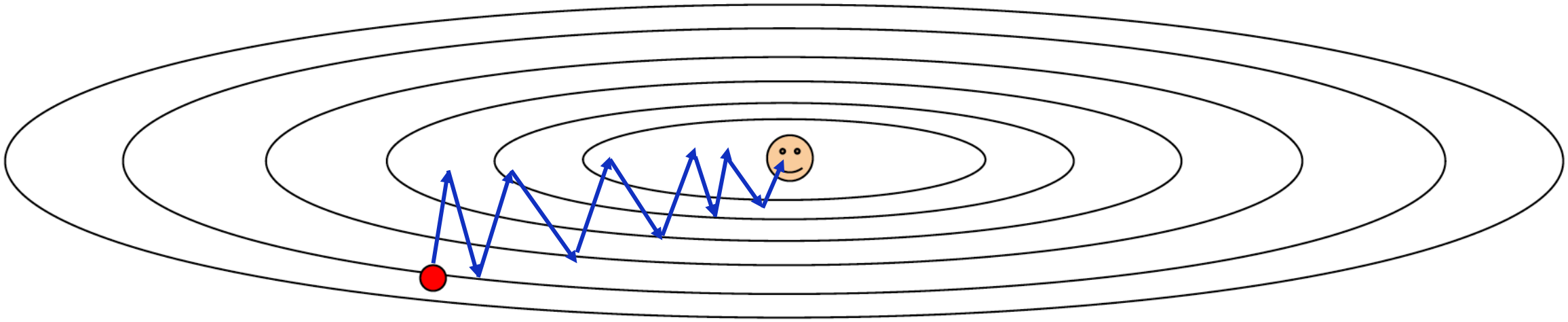
$$L(\boldsymbol{w}) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}_i(x_i, y_i, \boldsymbol{w}) + \lambda R(\boldsymbol{w})$$

$$\nabla_{\boldsymbol{w}} L(\boldsymbol{w}) = \frac{1}{m}\sum_{i=1}^{m} \nabla_{\boldsymbol{w}} \mathcal{L}_i(x_i, y_i, \boldsymbol{w}) + \lambda \nabla_{\boldsymbol{w}} R(\boldsymbol{w})$$

**Note**: To avoid confusion, we will use $\boldsymbol{w}$ (bold font) to denote the weight vector, and $\boldsymbol{w}[j]$ to denote the j-th weight in $\boldsymbol{w}$.

# Problems with SGD

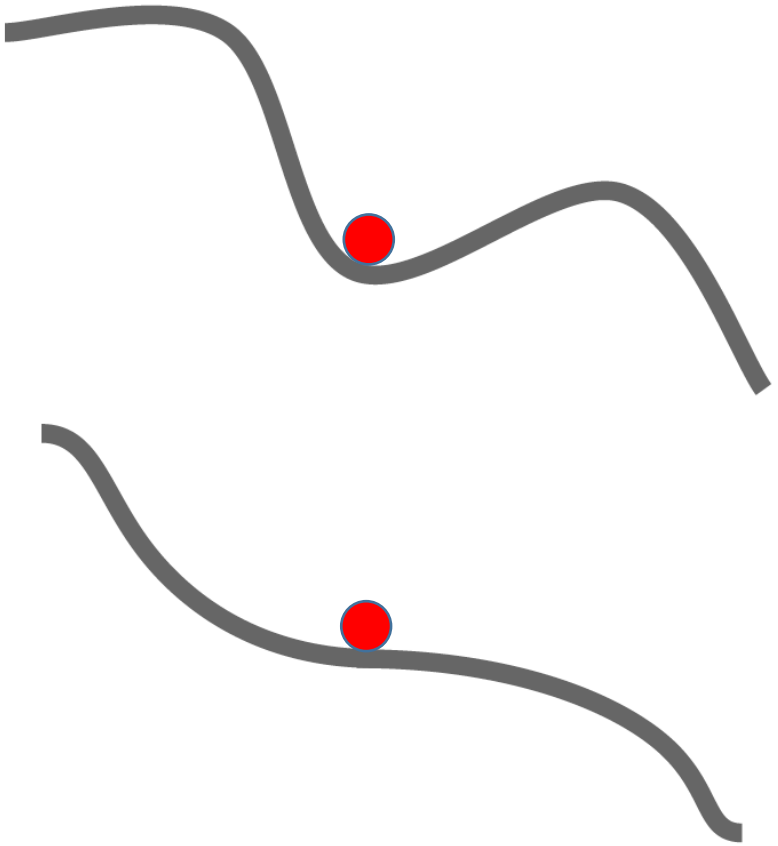
- What if loss changes quickly in one direction and slowly in another?
- What does gradient descent do?



SGD often takes noisy updates

# Problems with SGD

What if the loss function has a local minima or saddle point?



Zero gradient, gradient descent gets stuck

Saddle points much more common in high dimension



Dauphin et al, “Identifying and attacking the saddle point problem in high-dimensional non-convex optimization”, NIPS 2014

# SGD + Momentum

- **Motivation**: prevent big changes in gradients, try to keep some of the momentum from the past

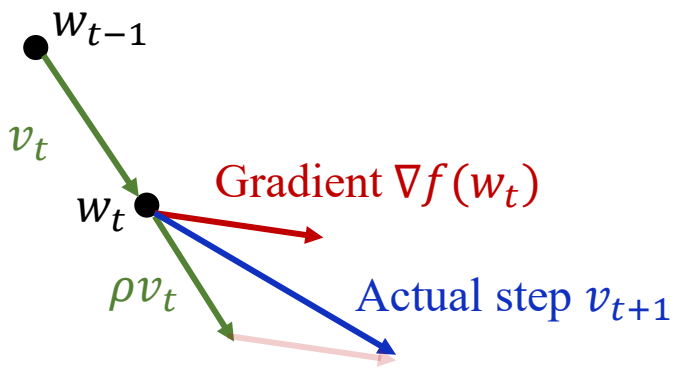
SGD

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \alpha \nabla f(\boldsymbol{w}_t)$$

SGD+Momentum

$$\boldsymbol{v}_{t+1} = \rho \boldsymbol{v}_t + (1-\rho)\nabla f(\boldsymbol{w}_t)$$

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \alpha \boldsymbol{v}_{t+1}$$




Where $t$ is denotes the $t$-th iteration

- Build up "velocity" $v$ as a running mean of gradients
- $\rho$ controls how much we keep the previous velocity
- Typically, $\rho = 0.9$ or $0.99$

# RMSProp

- **Motivation**: adapting the learning rate with running mean of squared gradients

$$S_{t+1,\boldsymbol{w}[j]} = \beta S_{t,\boldsymbol{w}[j]} + (1-\beta)\left(\nabla_{\boldsymbol{w}[j]} f(\boldsymbol{w}_t)\right)^2$$

$$\boldsymbol{w}_{t+1}[j] = \boldsymbol{w}_t[j] - \frac{\alpha}{\sqrt{S_{t+1,\boldsymbol{w}[j]} + \epsilon}} \nabla_{\boldsymbol{w}[j]} f(\boldsymbol{w}_t)$$

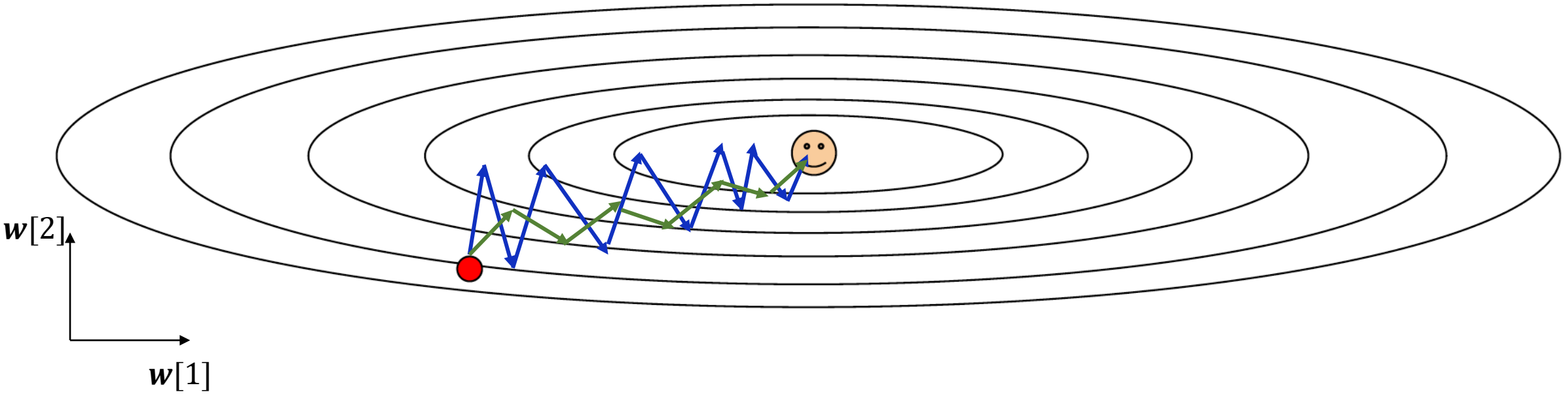
The gradient along a particular weight $\boldsymbol{w}[j]$

Constant to avoid zero denominator, typically $10^{-8}$

- Note that the learning weight is adjusted differently for different weights $\boldsymbol{w}[j]$
  - The larger gradient along the weight, the slower update is made
  - The smaller gradient along the weight, the faster update is made

# RMSProp

- Example:
  - Let $\boldsymbol{w}[2]$ be the direction that has larger gradient, and $\boldsymbol{w}[1]$ be the direction that has smaller gradient
  - The learning rate for $\boldsymbol{w}[1]$ will be enlarged, while the learning rate of $\boldsymbol{w}[2]$ will be decreased



# Adam

- **Motivation**: combine momentum and RMSProp

SGD+Momentum

$$\boldsymbol{v}_{t+1} = \rho \boldsymbol{v}_t + (1-\rho)\nabla f(\boldsymbol{w}_t)$$

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \alpha \boldsymbol{v}_{t+1}$$

RMSProp

$$S_{t+1,\boldsymbol{w}[j]} = \beta S_{t,\boldsymbol{w}[j]} + (1-\beta)\left(\nabla_{\boldsymbol{w}[j]} f(\boldsymbol{w}_t)\right)^2$$

$$\boldsymbol{w}_{t+1}[j] = \boldsymbol{w}_t[j] - \frac{\alpha}{\sqrt{S_{t+1,\boldsymbol{w}[j]} + \epsilon}} \nabla_{\boldsymbol{w}[j]} f(\boldsymbol{w}_t)$$

Adam

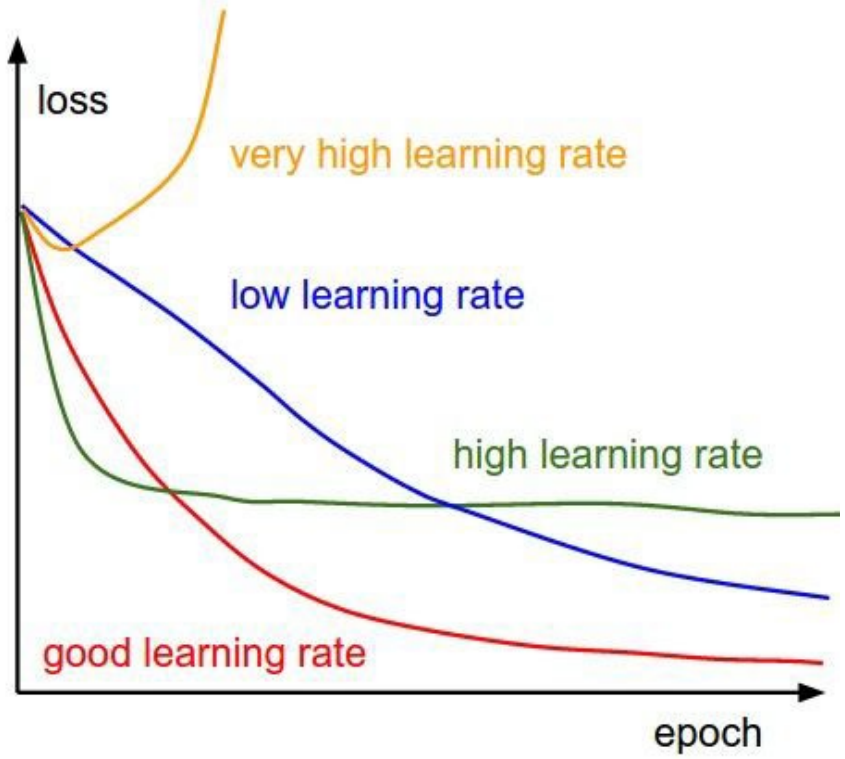
$$\boldsymbol{v}_{t+1} = \rho \boldsymbol{v}_t + (1-\rho)\nabla f(\boldsymbol{w}_t)$$

$$S_{t+1,\boldsymbol{w}[j]} = \beta S_{t,\boldsymbol{w}[j]} + (1-\beta)\left(\nabla_{\boldsymbol{w}[j]} f(\boldsymbol{w}_t)\right)^2$$

$$\boldsymbol{w}_{t+1}[j] = \boldsymbol{w}_t[j] - \frac{\alpha}{\sqrt{S_{t+1,\boldsymbol{w}[j]} + \epsilon}} \boldsymbol{v}_t[j]$$

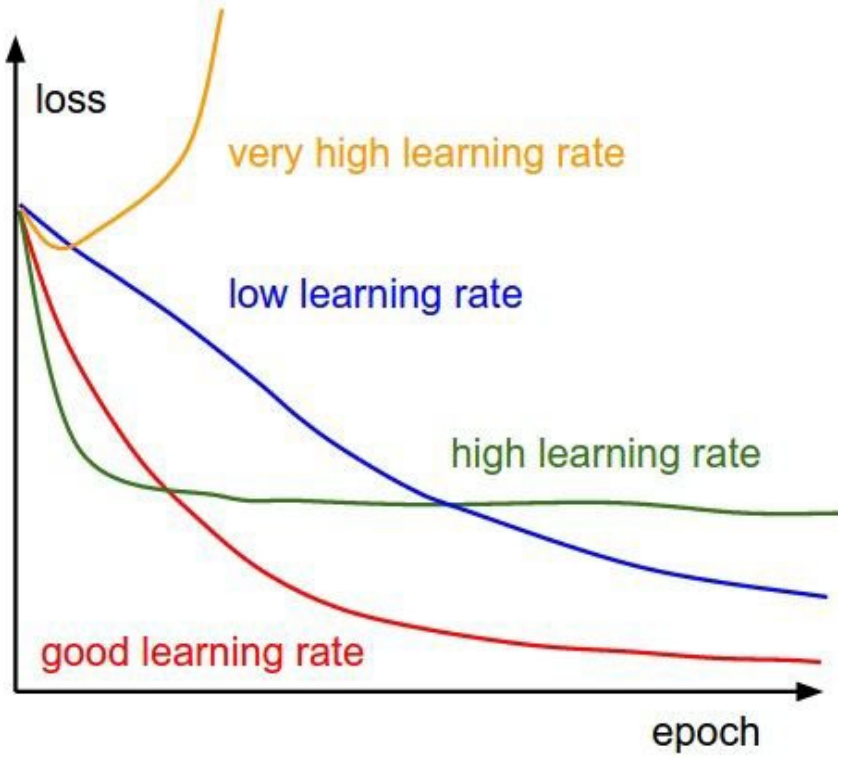
- People tend to use Adam in most cases
- You can set a higher learning rate $\alpha$, the algorithm can adjust it accordingly.
- Typically converge faster

# SGD, SGD+Momentum, RMSProp, Adam all have learning rate as a hyperparameter.




Q: Which one of these learning rates is best to use?

# SGD, SGD+Momentum, RMSProp, Adam all have learning rate as a hyperparameter.




$\Longrightarrow$ Learning rate decay over time!

**step decay:**
e.g. decay learning rate by half every few epochs.

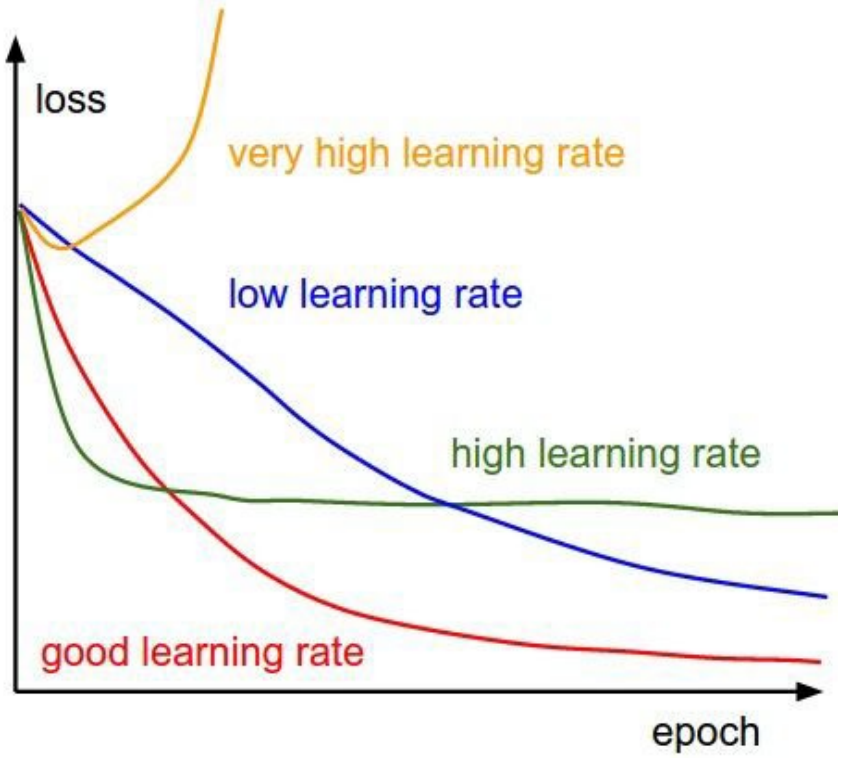
**exponential decay:**

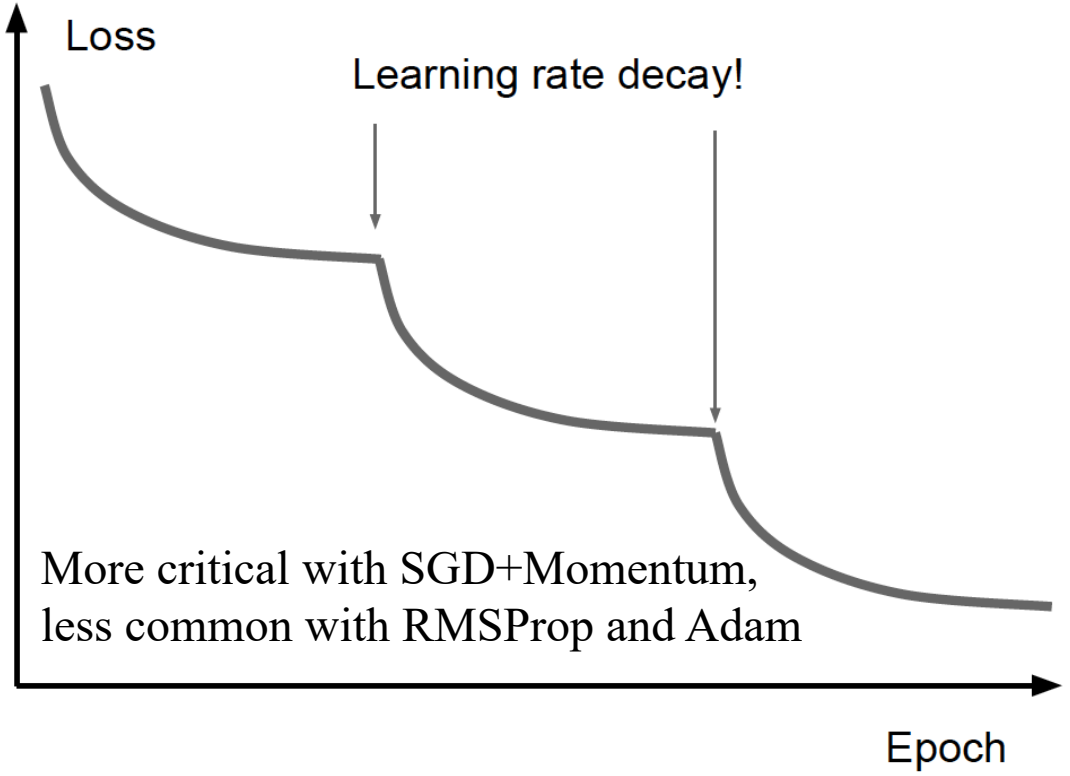
$$\alpha = \alpha_0 e^{-kt}$$

**1/t decay:**

$$\alpha = \alpha_0/(1 + kt)$$

# SGD, SGD+Momentum, RMSProp, Adam all have learning rate as a hyperparameter.



# Summary

- Neural Networks
  - Stacking non-linear layers of neurons to learn features from different levels
  - Logistic regression is a special case of NN with one single layer.
- Training neural networks
  - Applying the chain rule of derivatives → backpropagation
  - Optimization methods:
    - Gradient ascent/gradient descent
    - Batch/mini-batch/stochastic gradient ascent/descent
    - Momentum, RMSProp, Adam
- Activation functions
- Regularization – L1, L2, Dropout, Data Augmentation

# Literature

- Chapter 5 of Bishop's Pattern *Recognition and Machine Learning*
- Chapter 4 of Mitchell's *Machine Learning*