# Lecture 3:
# Training Neural Networks

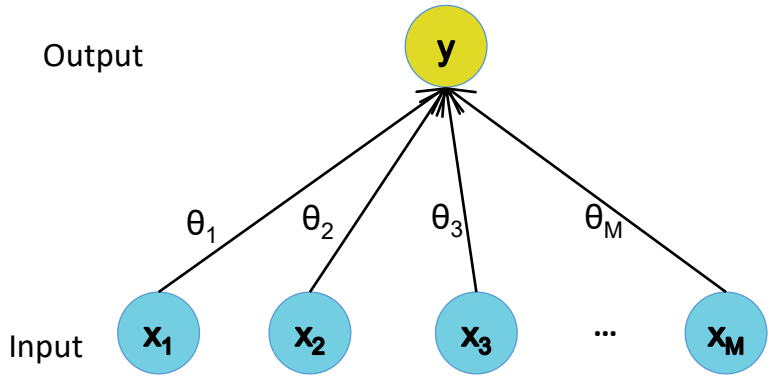
Olexandr Isayev

Department of Chemistry, CMU

olexandr@cmu.edu

# Backpropagation

**Case 1: Logistic Regression**




## Forward

$$J = y^* \log y + (1 - y^*) \log(1 - y)$$

$$y = \frac{1}{1 + \exp(-a)}$$

$$a = \sum_{j=0}^{D} \theta_j x_j$$

## Backward

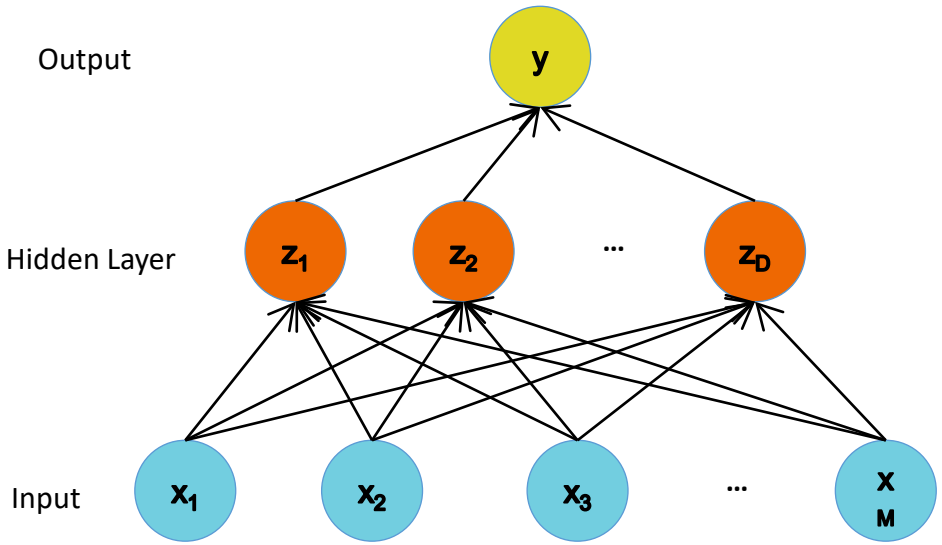
$$\frac{dJ}{dy} = \frac{y^*}{y} + \frac{(1 - y^*)}{y - 1}$$

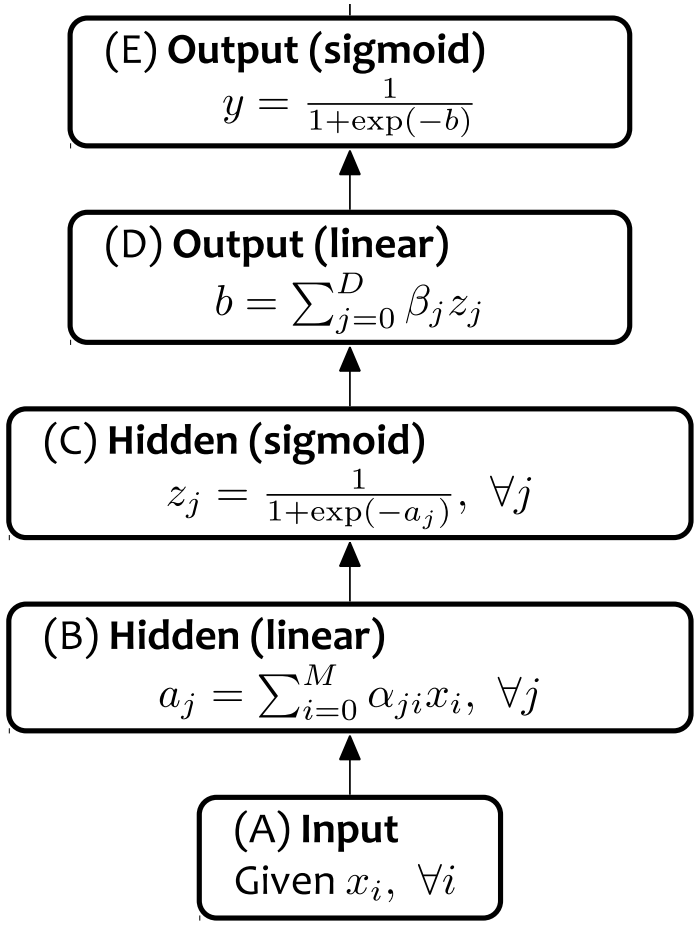
$$\frac{dJ}{da} = \frac{dJ}{dy}\frac{dy}{da}, \frac{dy}{da} = \frac{\exp(-a)}{(\exp(-a) + 1)^2}$$

$$\frac{dJ}{d\theta_j} = \frac{dJ}{da}\frac{da}{d\theta_j}, \frac{da}{d\theta_j} = x_j$$

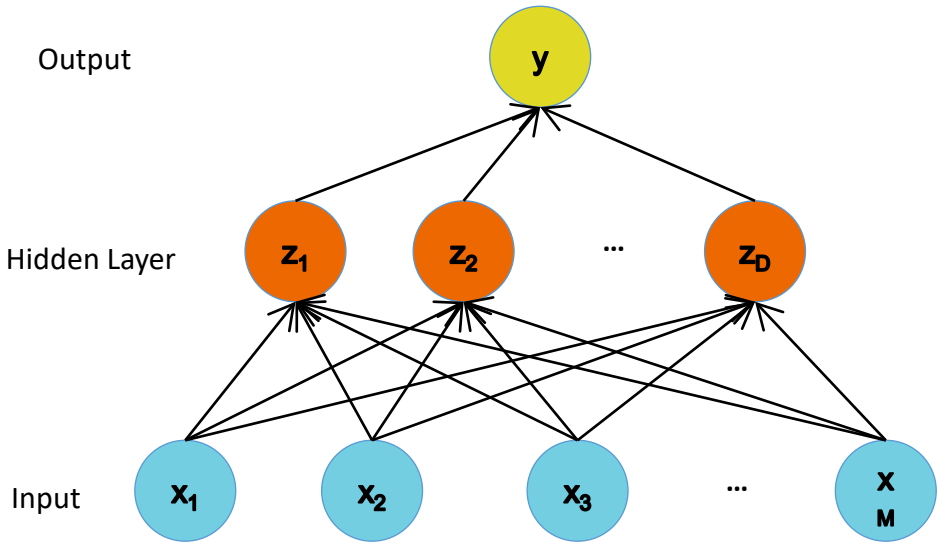
$$\frac{dJ}{dx_j} = \frac{dJ}{da}\frac{da}{dx_j}, \frac{da}{dx_j} = \theta_j$$

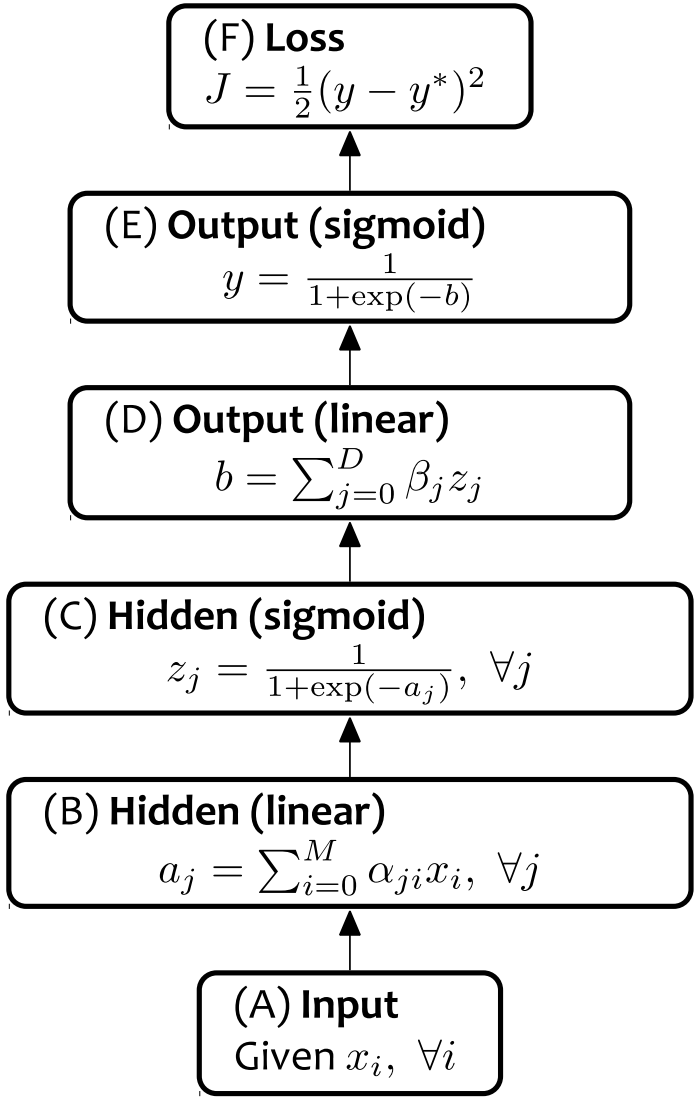
# Backpropagation



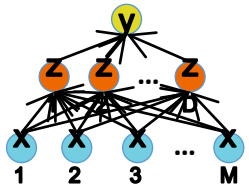
# Backpropagation



# Backpropagation

**Case 2: Neural Network**




Forward

$$J = y^* \log y + (1 - y^*) \log(1 - y)$$

$$y = \frac{1}{1 + \exp(-b)}$$

$$b = \sum_{j=0}^{D} \beta_j z_j$$

$$z_j = \frac{1}{1 + \exp(-a_j)}$$

$$a_j = \sum_{i=0}^{M} \alpha_{ji} x_i$$

Backward

$$\frac{dJ}{dy} = \frac{y^*}{y} + \frac{(1 - y^*)}{y - 1}$$

$$\frac{dJ}{db} = \frac{dJ}{dy}\frac{dy}{db}, \frac{dy}{db} = \frac{\exp(-b)}{(\exp(-b) + 1)^2}$$

$$\frac{dJ}{d\beta_j} = \frac{dJ}{db}\frac{db}{d\beta_j}, \frac{db}{d\beta_j} = z_j$$

$$\frac{dJ}{dz_j} = \frac{dJ}{db}\frac{db}{dz_j}, \frac{db}{dz_j} = \beta_j$$

$$\frac{dJ}{da_j} = \frac{dJ}{dz_j}\frac{dz_j}{da_j}, \frac{dz_j}{da_j} = \frac{\exp(-a_j)}{(\exp(-a_j) + 1)^2}$$

$$\frac{dJ}{d\alpha_{ji}} = \frac{dJ}{da_j}\frac{da_j}{d\alpha_{ji}}, \frac{da_j}{d\alpha_{ji}} = x_i$$

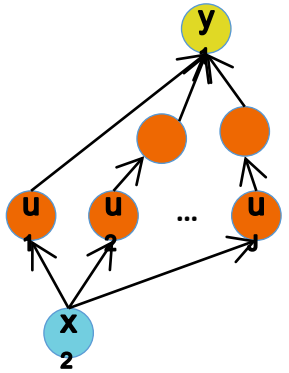
$$\frac{dJ}{dx_i} = \frac{dJ}{da_j}\frac{da_j}{dx_i}, \frac{da_j}{dx_i} = \sum_{j=0}^{D} \alpha_{ji}$$

# Chain Rule

**Given:** $\boldsymbol{y} = g(\boldsymbol{u})$ and $\boldsymbol{u} = h(\boldsymbol{x})$.

**Chain Rule:**

$$\frac{dy_i}{dx_k} = \sum_{j=1}^{J} \frac{dy_i}{du_j} \frac{du_j}{dx_k}, \quad \forall i, k$$




**Backpropagation:**

1. **Instantiate the computation as a directed acyclic graph**, where each intermediate quantity is a node
2. At each node, store (a) the quantity computed in the forward pass and (b) the **partial derivative** of the goal with respect to that node's intermediate quantity.
3. **Initialize** all partial derivatives to 0.
4. Visit each node in **reverse topological order**. At each node, add its contribution to the partial derivatives of its parents

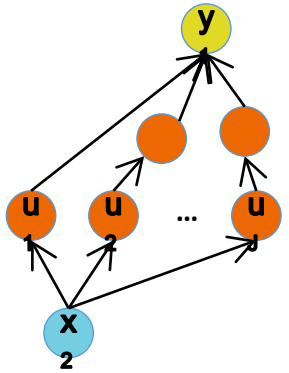
This algorithm is also called **automatic differentiation in the reverse-mode**

# Chain Rule

**Given:** $\boldsymbol{y} = g(\boldsymbol{u})$ and $\boldsymbol{u} = h(\boldsymbol{x})$.

**Chain Rule:**

$$\frac{dy_i}{dx_k} = \sum_{j=1}^{J} \frac{dy_i}{du_j} \frac{du_j}{dx_k}, \quad \forall i, k$$



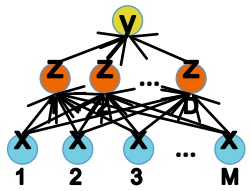

**Backpropagation:**

1. **Instantiate the computation as a directed acyclic graph**, where each node represents a Tensor.
2. At each node, store (a) the quantity computed in the forward pass and (b) the **partial derivatives** of the goal with respect to that node's Tensor.
3. **Initialize** all partial derivatives to 0.
4. Visit each node in **reverse topological order**. At each node, add its contribution to the partial derivatives of its parents

This algorithm is also called **automatic differentiation in the reverse-mode**

# Backpropagation

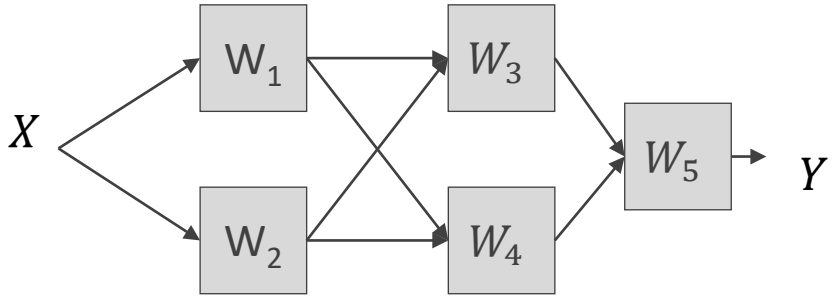
**Case 2: Neural Network**




| | Forward | Backward |
|---|---|---|
| Module 5 | $J = y^* \log y + (1 - y^*) \log(1 - y)$ | $\frac{dJ}{dy} = \frac{y^*}{y} + \frac{(1 - y^*)}{y - 1}$ |
| Module 4 | $y = \frac{1}{1 + \exp(-b)}$ | $\frac{dJ}{db} = \frac{dJ}{dy}\frac{dy}{db}, \frac{dy}{db} = \frac{\exp(-b)}{(\exp(-b) + 1)^2}$ |
| Module 3 | $b = \sum_{j=0}^{D} \beta_j z_j$ | $\frac{dJ}{d\beta_j} = \frac{dJ}{db}\frac{db}{d\beta_j}, \frac{db}{d\beta_j} = z_j$<br>$\frac{dJ}{dz_j} = \frac{dJ}{db}\frac{db}{dz_j}, \frac{db}{dz_j} = \beta_j$ |
| Module 2 | $z_j = \frac{1}{1 + \exp(-a_j)}$ | $\frac{dJ}{da_j} = \frac{dJ}{dz_j}\frac{dz_j}{da_j}, \frac{dz_j}{da_j} = \frac{\exp(-a_j)}{(\exp(-a_j) + 1)^2}$ |
| Module 1 | $a_j = \sum_{i=0}^{M} \alpha_{ji} x_i$ | $\frac{dJ}{d\alpha_{ji}} = \frac{dJ}{da_j}\frac{da_j}{d\alpha_{ji}}, \frac{da_j}{d\alpha_{ji}} = x_i$<br>$\frac{dJ}{dx_i} = \frac{dJ}{da_j}\frac{da_j}{dx_i}, \frac{da_j}{dx_i} = \sum_{j=0}^{D} \alpha_{ji}$ |

# Autograd: Auto-differentiation

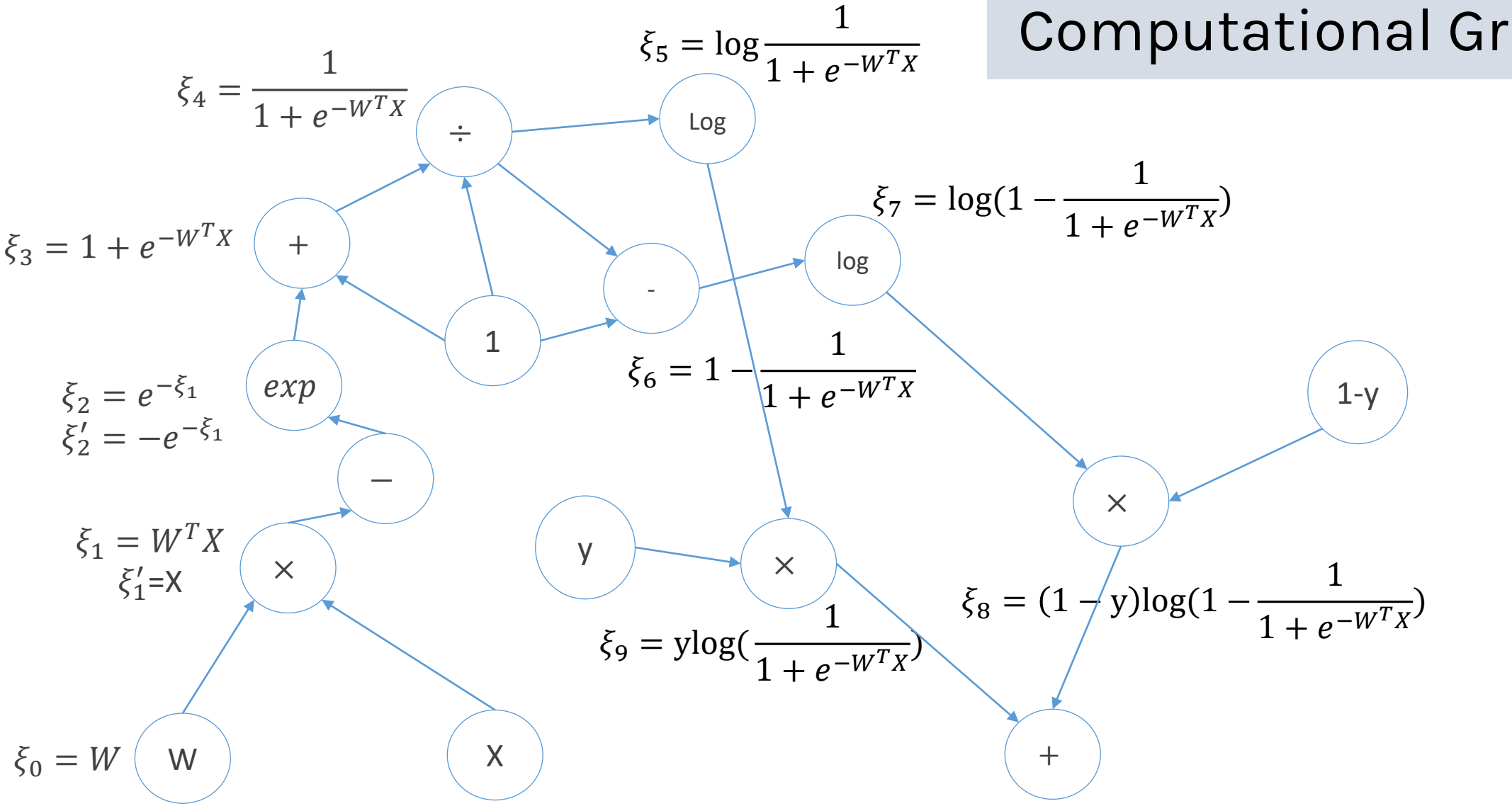
- 1. We specify the network structure




2. We create the computational graph …

What is computational graph?

# Computational Graph



# Forward mode: Evaluate the derivative at:

| Variables | derivatives | Value of the variable | Value of the partial derivative | $\frac{d\mathcal{L}}{d\xi_n}$ |
|---|---|---|---|---|
| $\xi_1 = -W^T X$ | $\frac{\partial \xi_1}{\partial W} = -X$ | $-9$ | -3 | -3 |
| $\xi_2 = e^{\xi_1} = e^{-W^T X}$ | $\frac{\partial \xi_2}{\partial \xi_1} = e^{\xi_1}$ | $e^{-9}$ | $e^{-9}$ | $-3e^{-9}$ |
| $\xi_3 = 1 + \xi_2 = 1 + e^{-W^T X}$ | $\frac{\partial \xi_3}{\partial \xi_2} = 1$ | $1+e^{-9}$ | 1 | $-3e^{-9}$ |
| $\xi_4 = \frac{1}{\xi_3} = \frac{1}{1 + e^{-W^T X}} = p$ | $\frac{\partial \xi_4}{\partial \xi_3} = -\frac{1}{\xi_3^2}$ | $\frac{1}{1 + e^{-9}}$ | $\left(\frac{1}{1 + e^{-9}}\right)^2$ | $-3e^{-9}\left(\frac{1}{1+e^{-9}}\right)^2$ |
| $\xi_5 = \log \xi_4 = \log p = \log \frac{1}{1 + e^{-W^T X}}$ | $\frac{\partial \xi_5}{\partial \xi_4} = \frac{1}{\xi_4}$ | $\log \frac{1}{1 + e^{-9}}$ | $1 + e^{-9}$ | $-3e^{-9}\left(\frac{1}{1+e^{-9}}\right)$ |
| $\mathcal{L}_i^A = -y\xi_5$ | $\frac{\partial \mathcal{L}}{\partial \xi_5} = -y$ | $-\log \frac{1}{1 + e^{-9}}$ | $-1$ | $3e^{-9}\left(\frac{1}{1+e^{-9}}\right)$ |
| $\frac{\partial \mathcal{L}_i^A}{\partial W} = \frac{\partial \mathcal{L}_i}{\partial \xi_5}\frac{\partial \xi_5}{\partial \xi_4}\frac{\partial \xi_4}{\partial \xi_3}\frac{\partial \xi_3}{\partial \xi_2}\frac{\partial \xi_2}{\partial \xi_1}\frac{\partial \xi_1}{\partial W}$ | | | $-3$ | 0.00037018372 |

# Backward mode: Evaluate the derivative at:

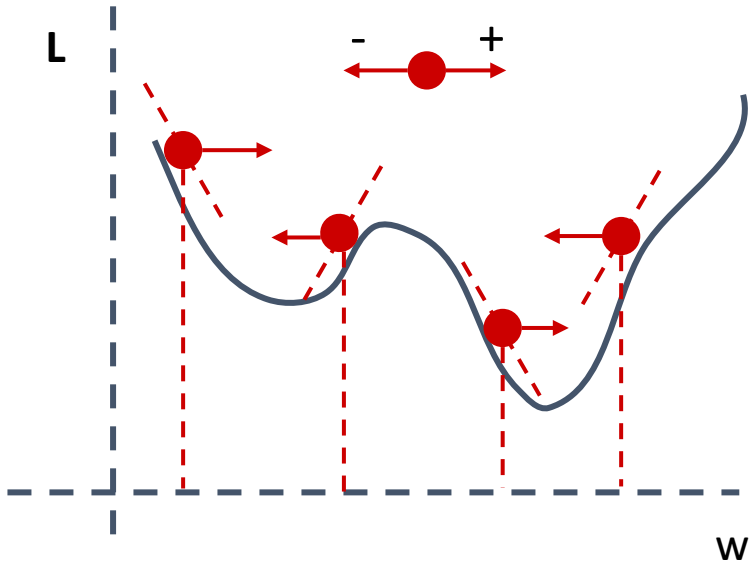
| Variables | derivatives | Value of the variable | Value of the partial derivative |
|---|---|---|---|
| $\xi_1 = -W^T X$ | $\frac{\partial \xi_1}{\partial W} = -X$ | $-9$ | -3 |
| $\xi_2 = e^{\xi_1} = e^{-W^T X}$ | $\frac{\partial \xi_2}{\partial \xi_1} = e^{\xi_1}$ | $e^{-9}$ | $e^{-9}$ |
| $\xi_3 = 1 + \xi_2 = 1 + e^{-W^T X}$ | $\frac{\partial \xi_3}{\partial \xi_2} = 1$ | $1+e^{-9}$ | 1 |
| $\xi_4 = \frac{1}{\xi_3} = \frac{1}{1 + e^{-W^T X}} = p$ | $\frac{\partial \xi_4}{\partial \xi_3} = -\frac{1}{\xi_3^2}$ | $\frac{1}{1 + e^{-9}}$ | $\left(\frac{1}{1 + e^{-9}}\right)^2$ |
| $\xi_5 = \log \xi_4 = \log p = \log \frac{1}{1 + e^{-W^T X}}$ | $\frac{\partial \xi_5}{\partial \xi_4} = \frac{1}{\xi_4}$ | $\log \frac{1}{1 + e^{-9}}$ | $1 + e^{-9}$ |
| $\mathcal{L}_i^A = -y\xi_5$ | $\frac{\partial \mathcal{L}}{\partial \xi_5} = -y$ | $-\log \frac{1}{1 + e^{-9}}$ | $-1$ |
| $\frac{\partial \mathcal{L}_i^A}{\partial W} = \frac{\partial \mathcal{L}_i}{\partial \xi_5}\frac{\partial \xi_5}{\partial \xi_4}\frac{\partial \xi_4}{\partial \xi_3}\frac{\partial \xi_3}{\partial \xi_2}\frac{\partial \xi_2}{\partial \xi_1}\frac{\partial \xi_1}{\partial W}$ | | | Type equation here. |

Store all these values

# Gradient Descent

- Algorithm for optimization of first order to finding a minimum of a function.
- It is an iterative method.
- *L* is decreasing in the direction of the negative derivative.
- The learning rate is controlled by the magnitude of $\lambda$.

$$w^{(i+1)} = w^{(i)} - \lambda \frac{d\mathcal{L}}{dw}$$



# HW1: Implement a Neural Network

In this homework, you will implement a single layer neural network **from scratch**, then validate it using a PyTorch implementation.

All HW1 files can be found inHW1.zip under HW1assignment.

HW1 out: today

HW1 due: **2 weeks**

Submission1**:** A report(pdf) containing the loss and accuracy plots for the training process and test process. Compare and comment on the results from your implementation vs PyTorch implementation.

Submission2: Submit your code as a single **.ipynb** file or zipped **.py** files folder.

Please be aware of the two separate submissions on Gradescope.

# Suggestions

Feel free to play around with your implementation (though not mandatory), such as stacking two layers or more, changing hidden layer dimensions, etc. (Bonus points)

This is a binary classification task, but your implementation should generalize to multi-label classification (classes > 2), too.

# Optimization & Deep Learning Tricks

- Challenges in Optimization
- Momentum
- Adaptive Learning Rate
- Parameter Initialization
- Batch Normalization
- Dropout
- Shortcuts & Highways

Tips & Tricks!

# Warning!

At some point, you will think:

Why are you telling us all this madness?

Because pretty much all of it is commonly used.

This is collection of tips and tricks. No general theory…

# Avoiding Local Optima (min and max)

One of hardest problem for designing neural network architectures and optimization methods

Ensure that model converges to at least to a set of parameter values that give results close to this optimum on unseen test data.

There is no real solution to this problem.

It requires experimentation and analysis that is more craft than science.

Still, this lecture presents a number of methods that generally help avoiding getting stuck in local optima.

# Learning vs. Optimization

Goal of learning: minimize generalization error

In practice, empirical risk minimization:

$$J(\theta) = \mathbf{E}_{(x,y)\sim p_{data}} \left[ L(f(x;\theta), y) \right]$$

$$\hat{J}(\theta) = \frac{1}{m} \sum_{i=1}^{m} L(f(x^{(i)};\theta), y^{(i)})$$

Quantity optimized
different from the quantity
we care about

# GD vs. Stochastic GD Algorithms

GD algorithms

- Optimize empirical risk using exact gradients

Stochastic GD algorithms

- Estimates gradient from a small random sample

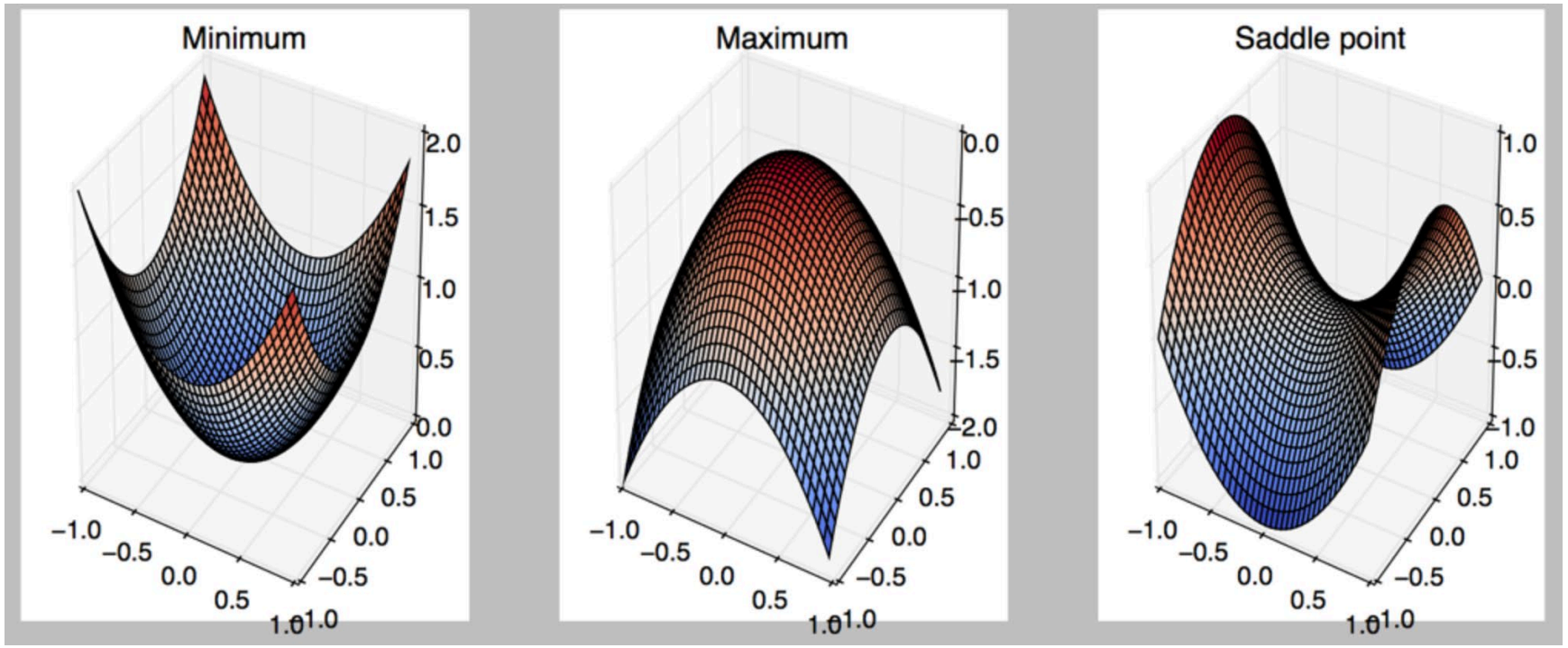
$$\nabla J(\theta) = \mathbf{E}_{(x,y)\sim p_{data}} \left[ \nabla L(f(x;\theta), y) \right]$$

*Large mini-batch*: gradient computation expensive

*Small mini-batch*: greater variance in estimate, longer steps for convergence

# Critical Points

- Points with zero gradient
- 2nd-derivate (Hessian) determines curvature




Goodfellow et al. (2016)

# Stochastic Gradient Descent

Take small steps in direction of negative gradient

Sample *m* examples from training set and compute:

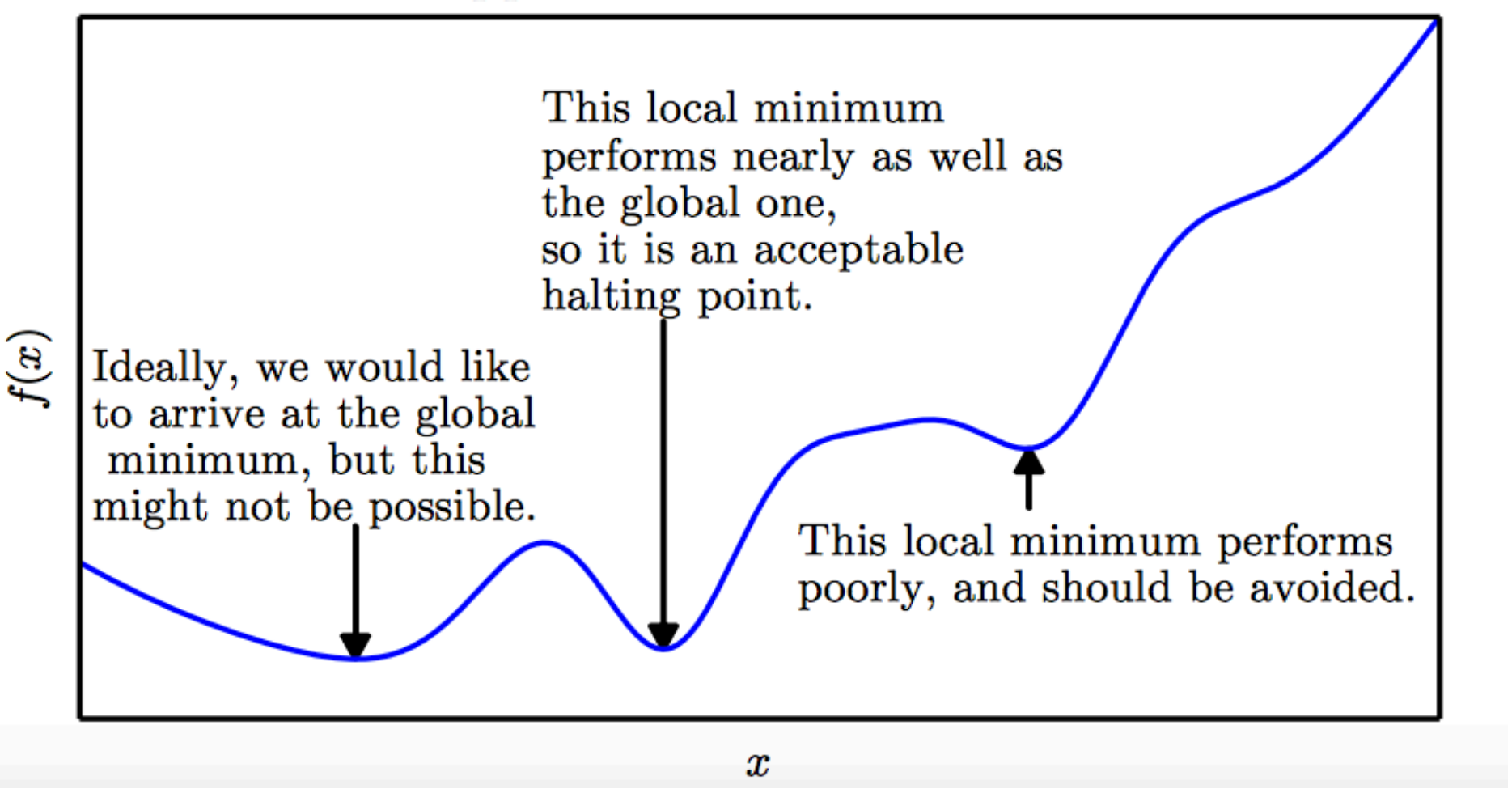
$$g = \frac{1}{m} \sum_i \nabla L(f(x^{(i)}; \theta), y^{(i)})$$

Update parameters:

$$\theta = \theta - \varepsilon_k g$$

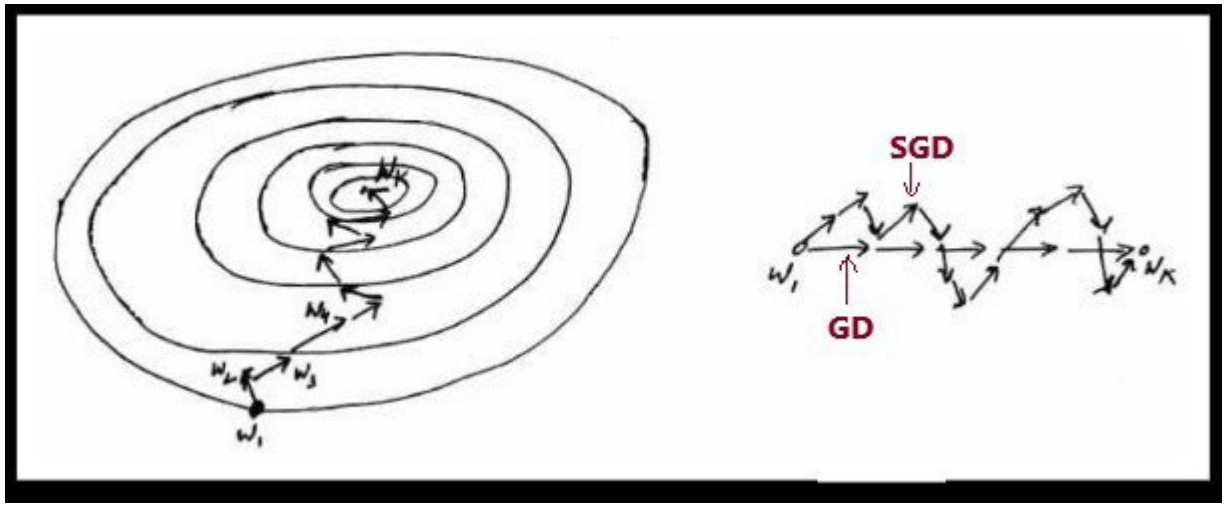
In practice: shuffle training set once and pass through multiple times

# Challenges: Local Minima




Goodfellow et al. (2016)

# SGD

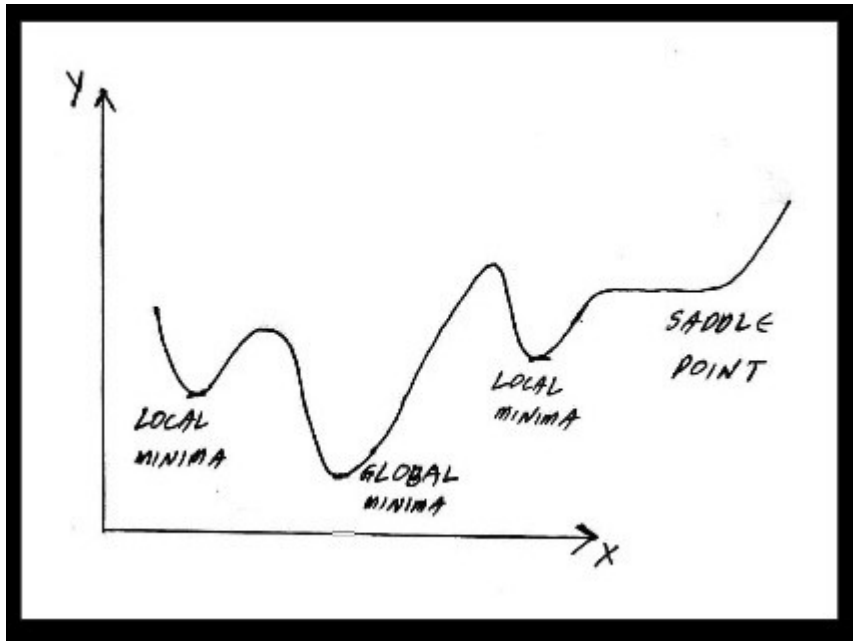



**Advantage:**

1. Memory requirement is less compared to the GD algorithm as derivative is computed taking only a handful of points.

**Disadvantages:**

1. The update of MB-SGD is much noisy compared to the update of the GD algorithm.
2. Take a longer time to converge than the GD algorithm.
3. May still stuck at local minima.



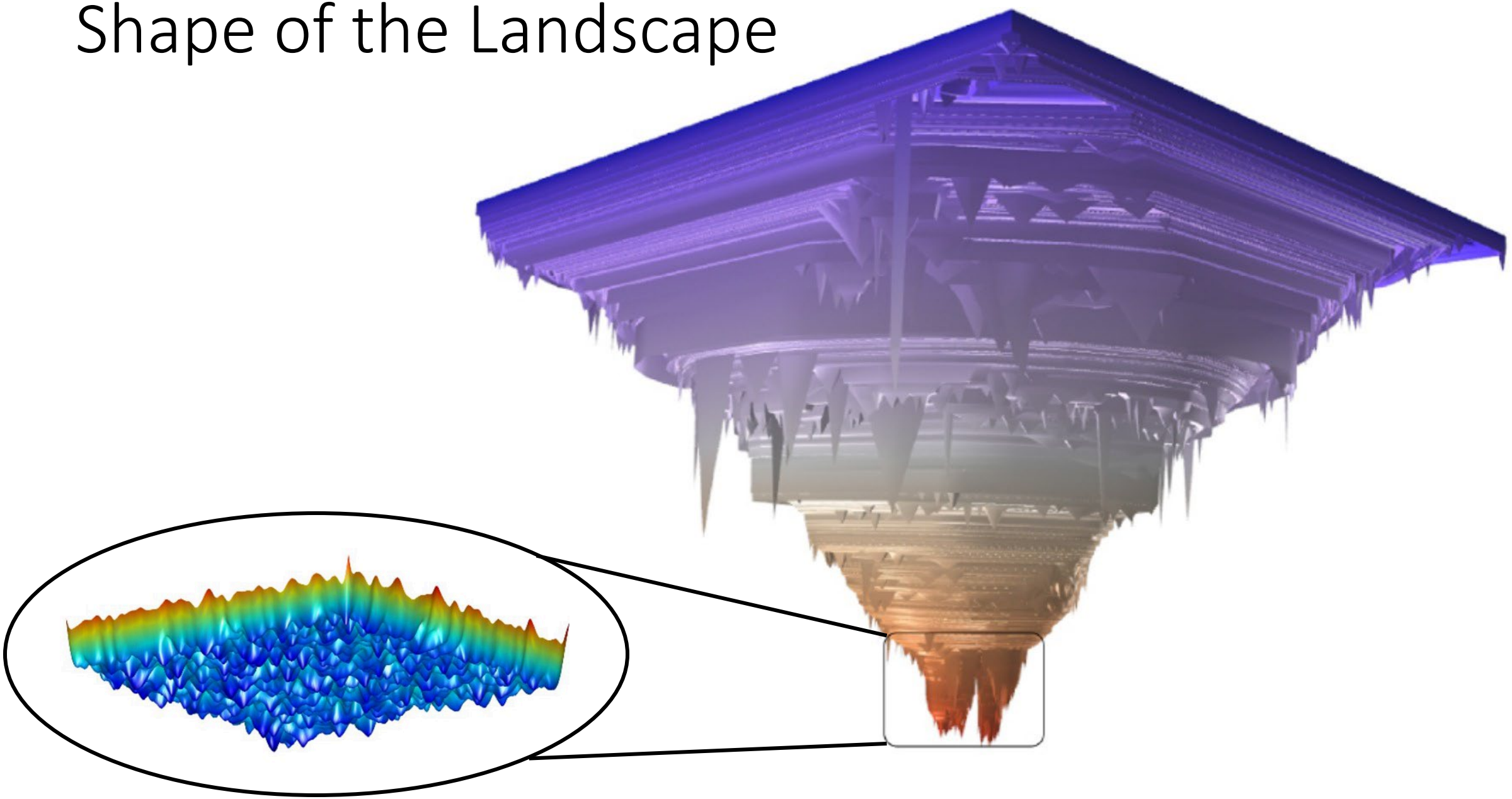
# Local Minima

Old view: local minima is major problem in neural network training

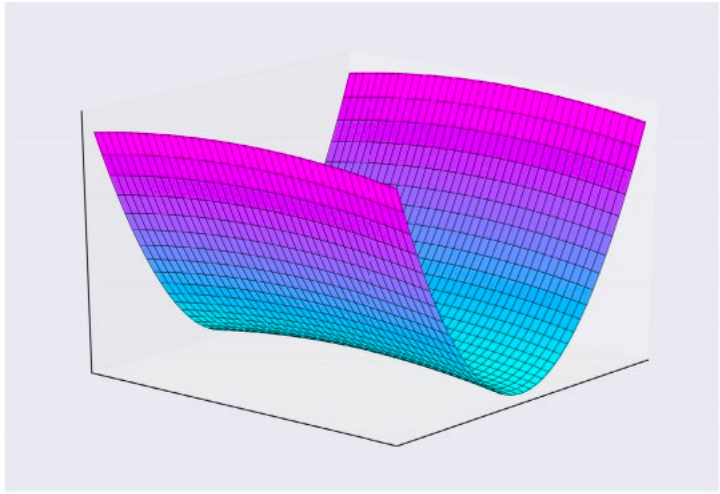
Recent view:

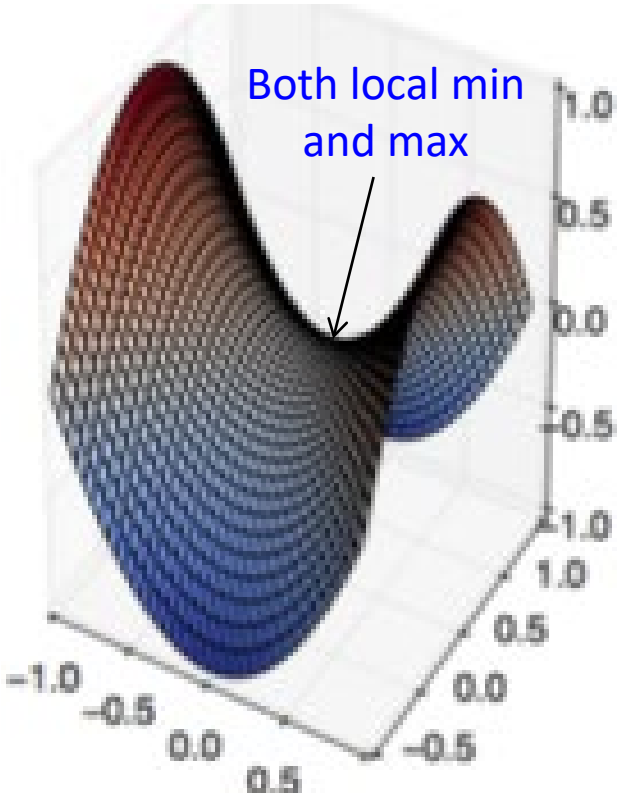
- For sufficiently large neural networks, most local minima incur low cost
- Not important to find true global minimum

# Shape of the Landscape



# Saddle Points






- Recent studies indicate that in high dim, saddle points are more likely than local min
- Gradient can be very small near saddle points

Goodfellow et al. (2016)

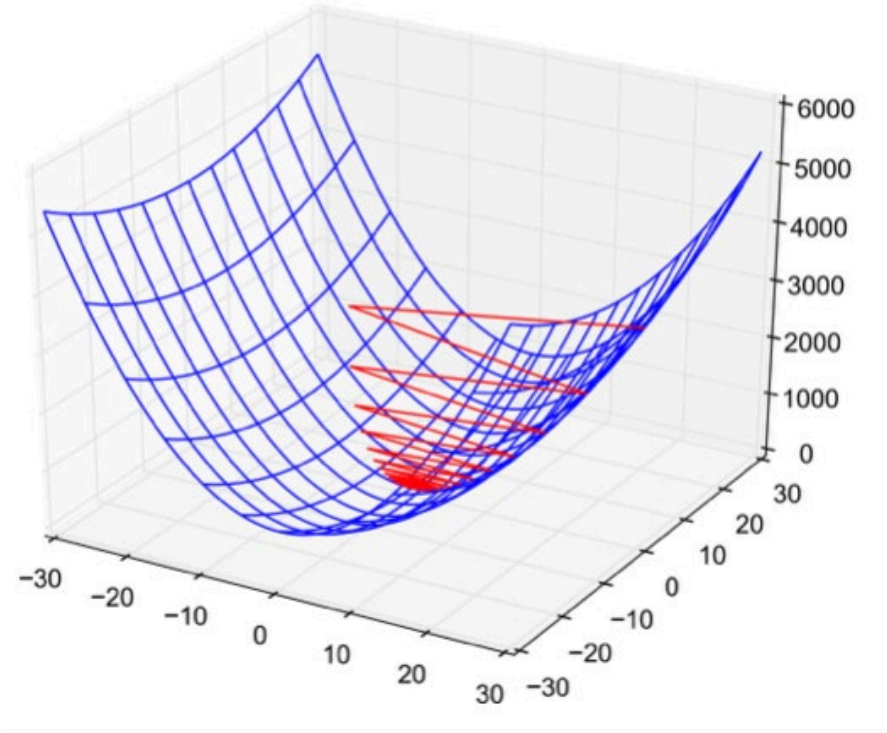
# Poor Conditioning (Initialization)

Poorly conditioned Hessian matrix

- High curvature: small steps leads to huge increase

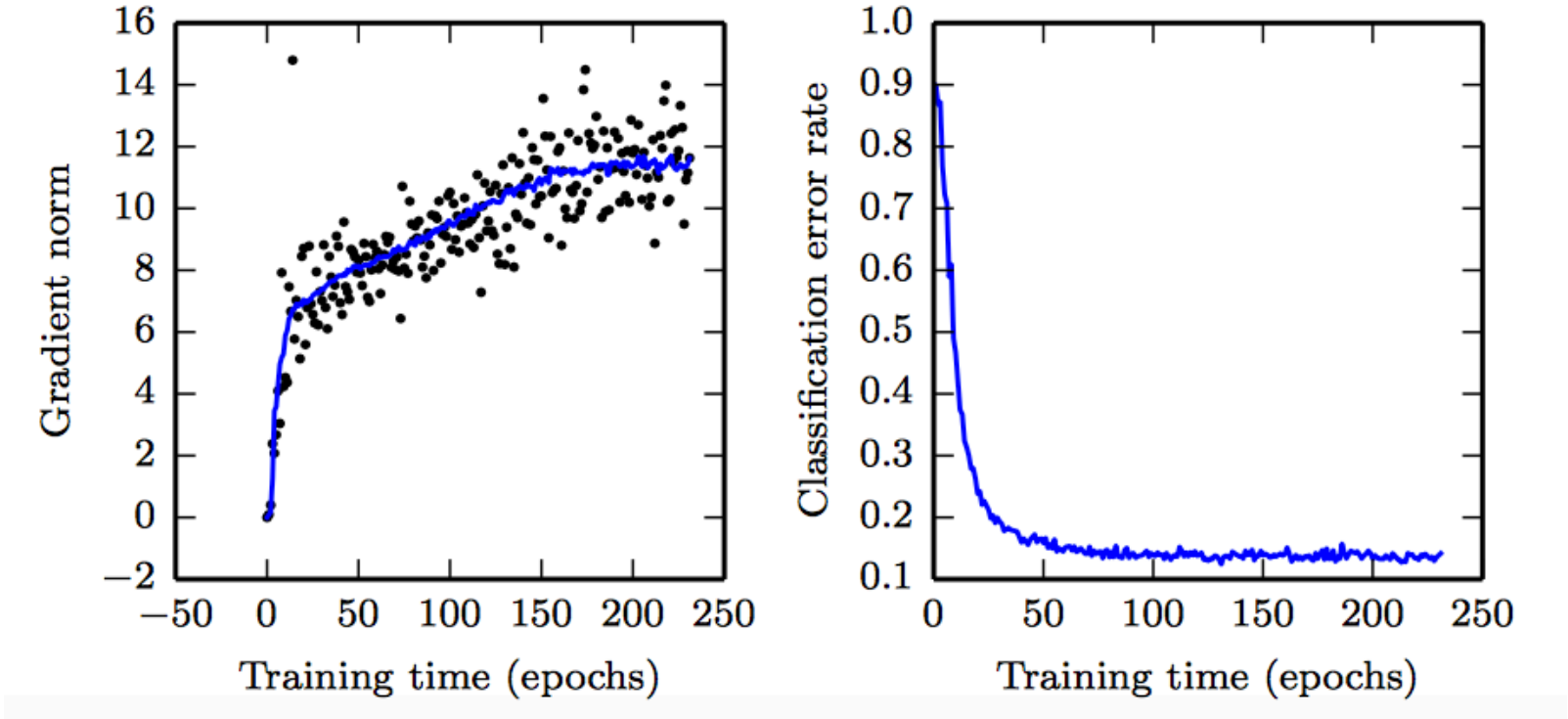
Learning is slow despite strong gradients

Oscillations slow down progress



# No Critical Points

- Gradient norm increases, but validation error decreases

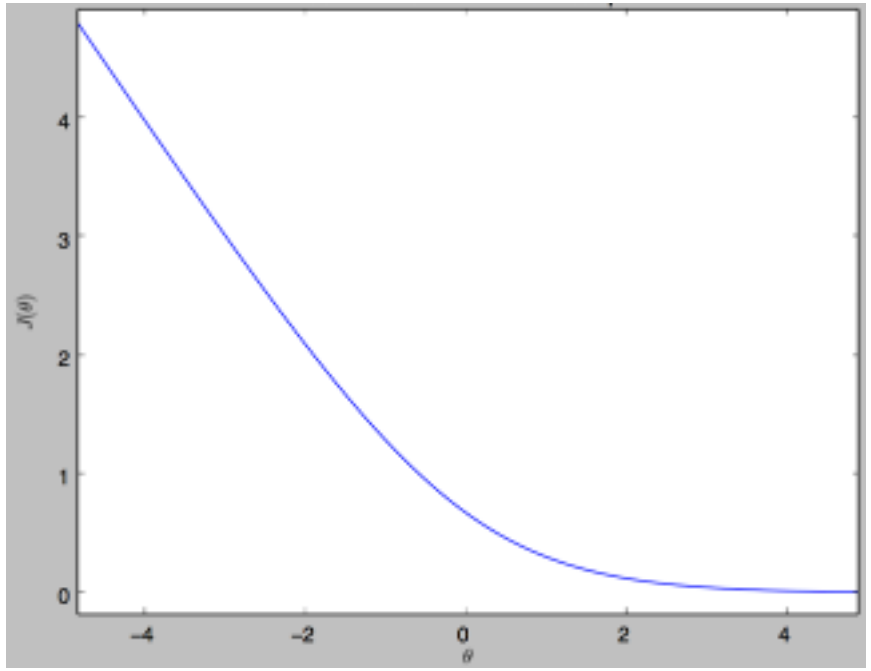



*Convolution Nets for Object Detection*

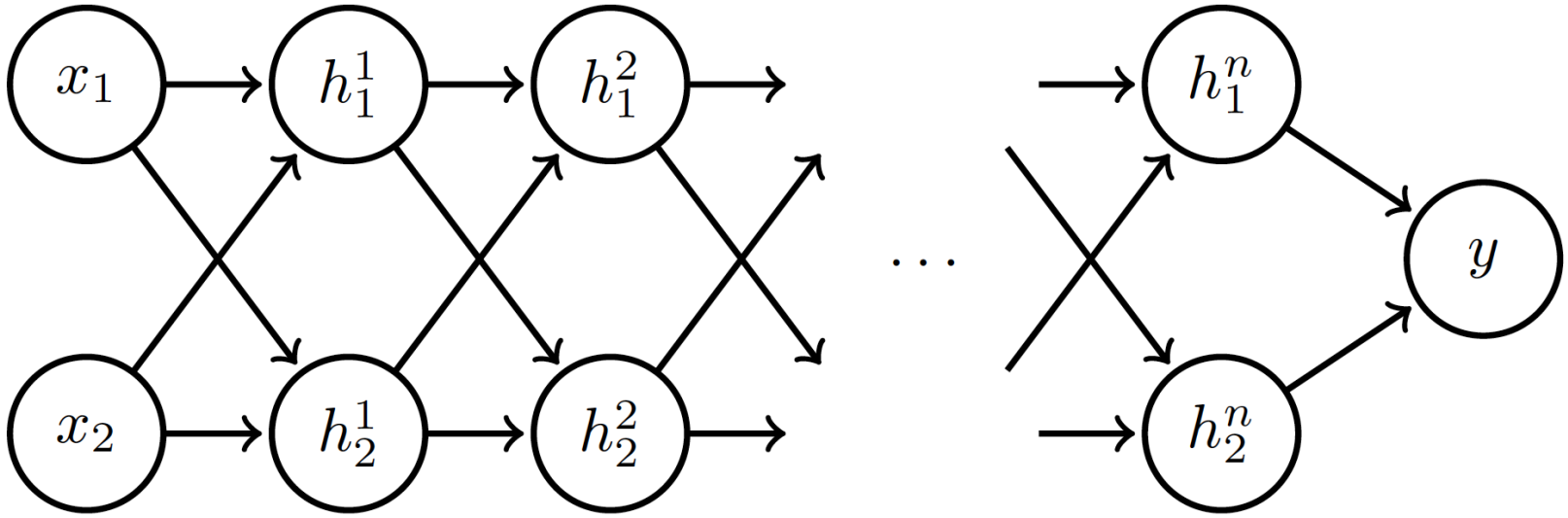
Goodfellow et al. (2016)

# No Critical Points

- Some cost functions do not have critical points. In particular classification.



# Exploding and Vanishing Gradients




Linear activation

$$h_i = Wx$$

$$h_i = Wh_{i-1}, \qquad i = 2, \ldots, n$$

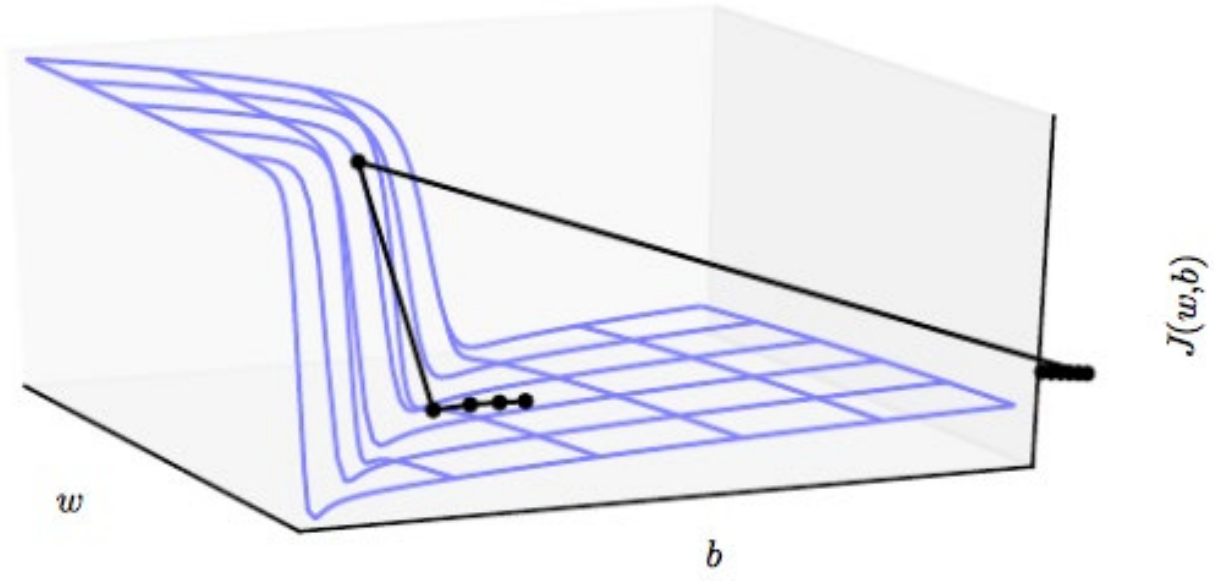
# Exploding and Vanishing Gradients

Exploding gradients lead to cliffs

Can be mitigated using gradient clipping

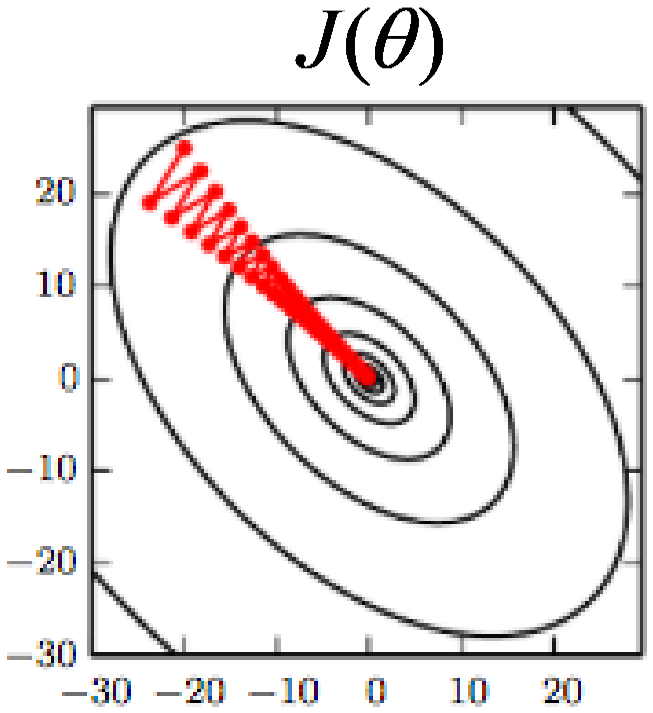
$$\text{if } \|g\| > u$$

$$g \leftarrow \frac{gu}{\|g\|}$$




Goodfellow et al. (2016)

# Stochastic Gradient Descent

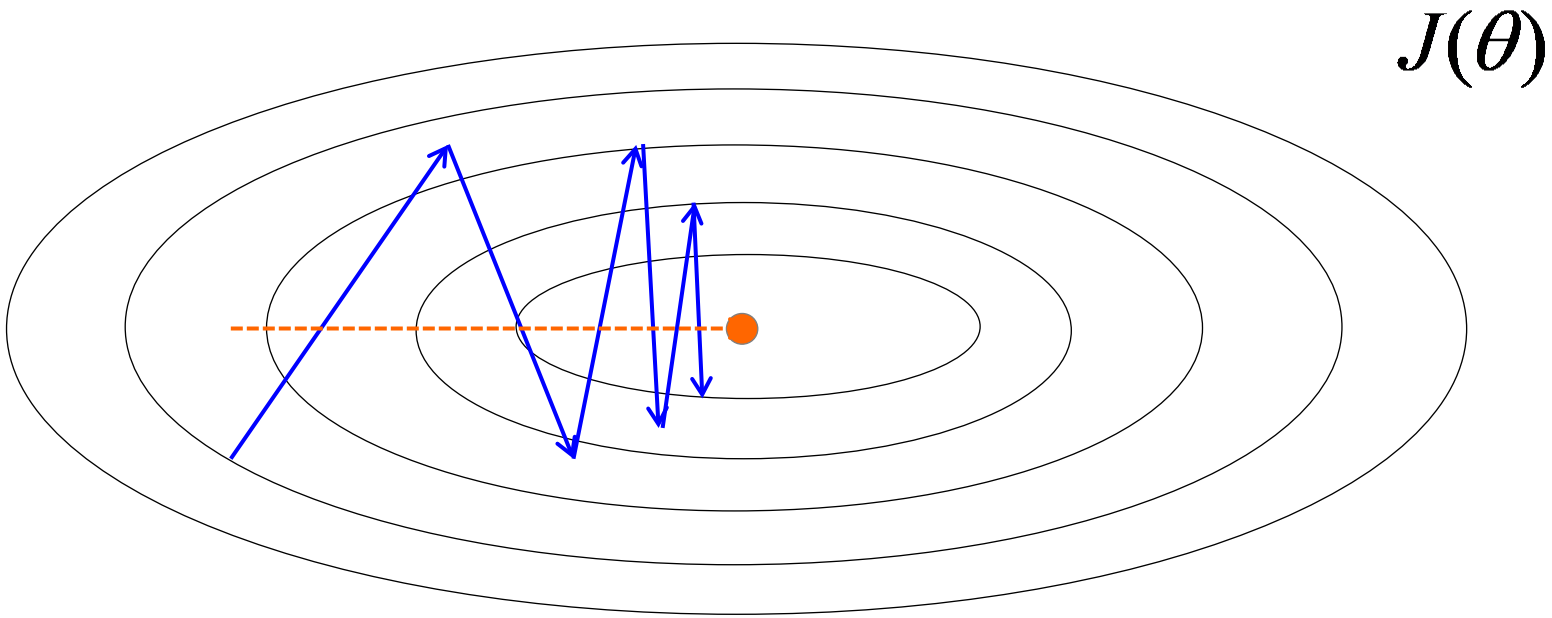



Oscillations because updates do not exploit curvature information

Goodfellow et al. (2016)

# Momentum

SGD is slow when there is high curvature




Average gradient presents faster path to opt:

- vertical components cancel out

# Momentum

- Uses past gradients for update
- Maintains a new quantity: 'velocity'
- *Exponentially decaying average* of gradients:

$$g = \frac{1}{m}\sum_i \nabla_\theta L(f(x^{(i)};\theta), y^{(i)})$$

Current gradient update

$$v = \alpha v + (-\varepsilon g)$$

$\alpha \in [0,1)$ controls how quickly effect of past gradients decay

# Momentum

- Compute gradient estimate:

$$g = \frac{1}{m} \sum_i \nabla_\theta L(f(x^{(i)}; \theta), y^{(i)})$$

- Update velocity:

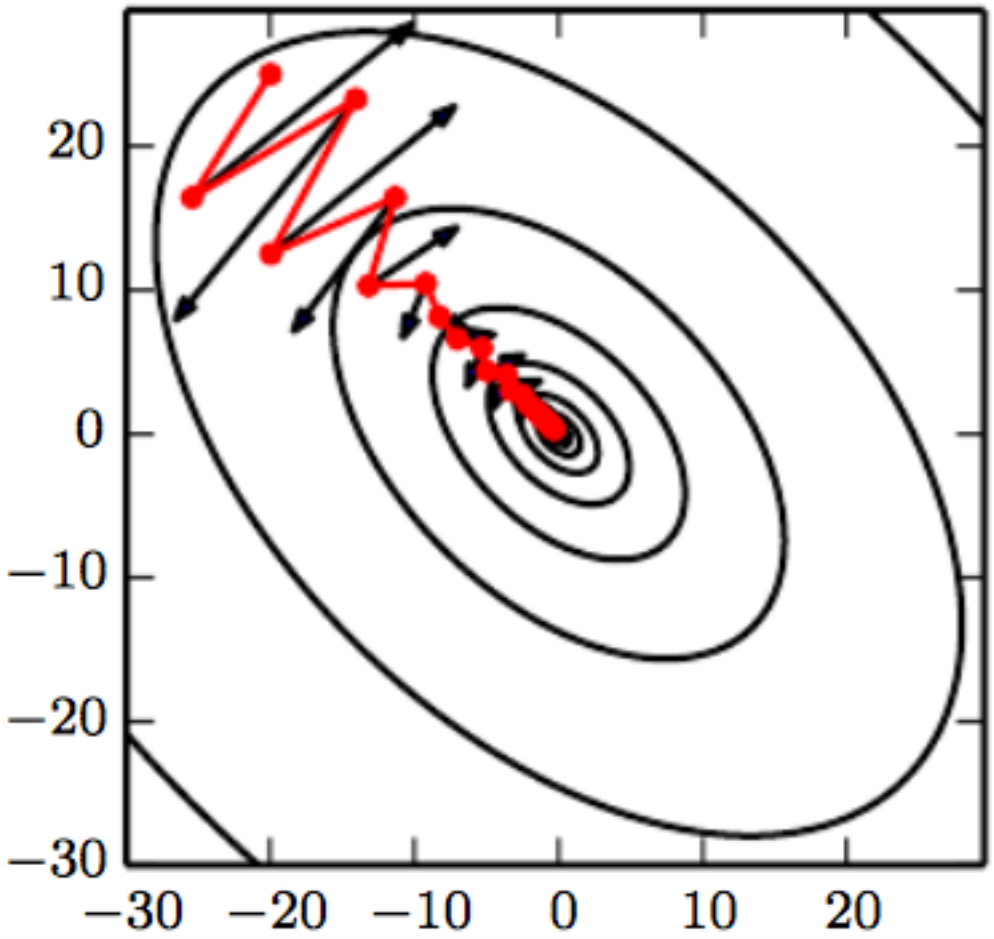
$$v = \alpha v - \varepsilon g$$

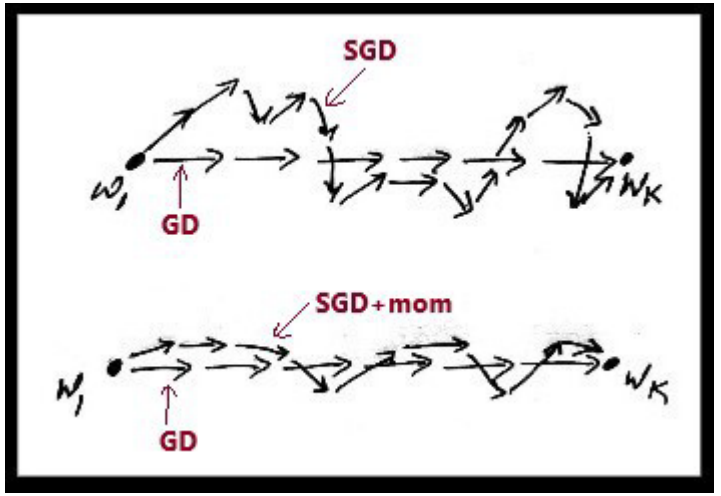
- Update parameters:

$$\theta = \theta + v$$

# Momentum

*Damped oscillations:* gradients in opposite directions get cancelled out




Goodfellow et al. (2016)

# Nesterov Momentum

- Apply an interim update: $\tilde{\theta} = \theta + v$

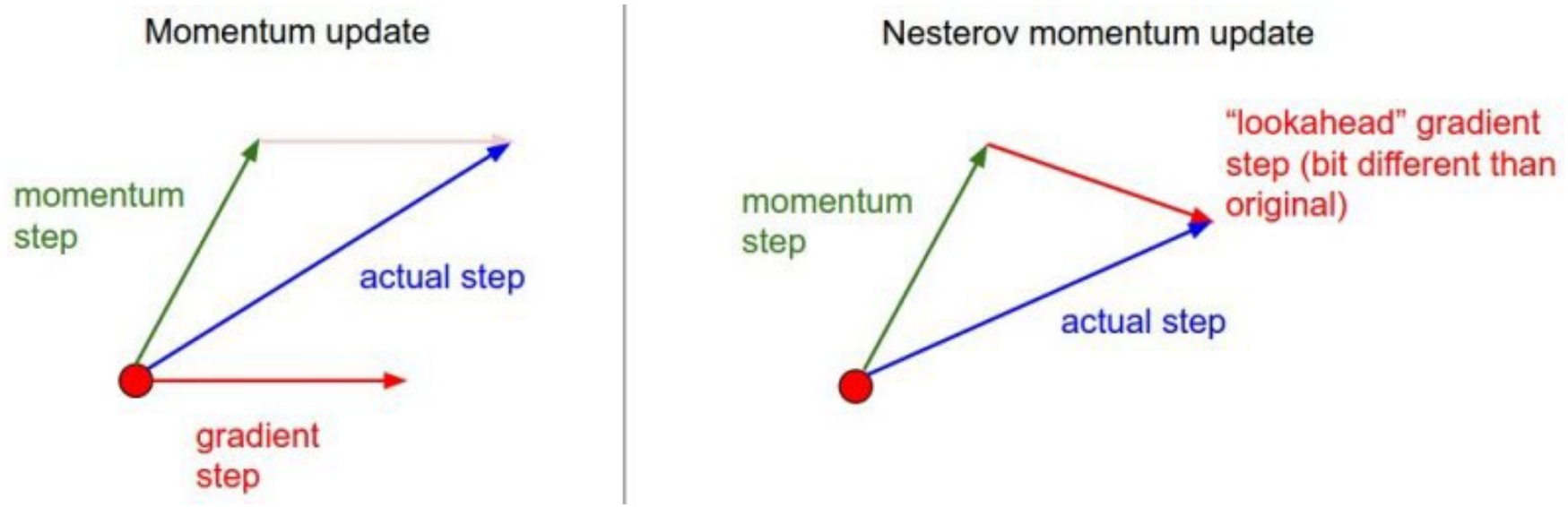
- Perform a correction based on gradient at the interim point:

$$g = \frac{1}{m} \sum_i \nabla_\theta L(f(x^{(i)}; \tilde{\theta}), y^{(i)})$$

$$v = \alpha v - \varepsilon g$$

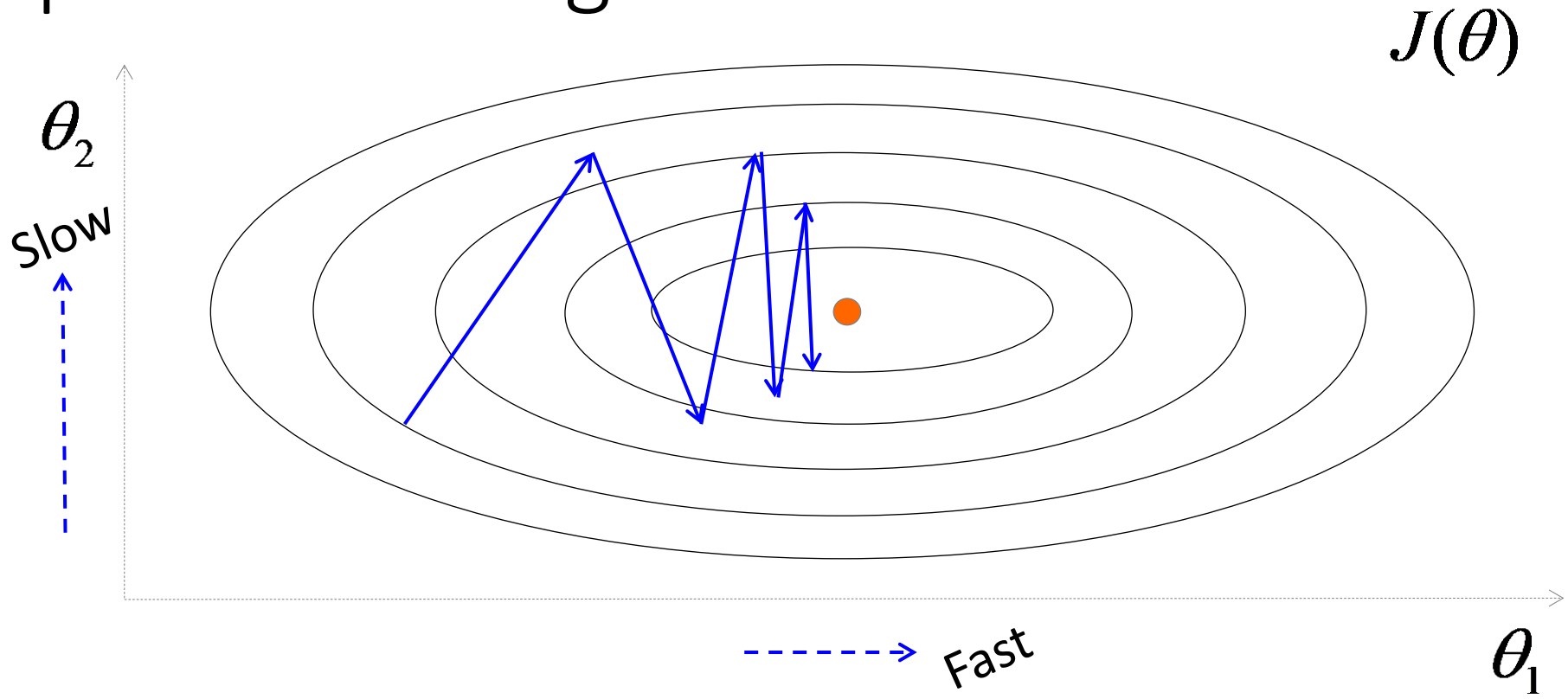
$$\theta = \theta + v$$

Momentum based on look-ahead slope




Nesterov momentum. Instead of evaluating gradient at the current position (red circle), we know that our momentum is about to carry us to the tip of the green arrow. With Nesterov momentum we therefore instead evaluate the gradient at this "looked-ahead" position.

# Adaptive Learning Rates




- Oscillations along vertical direction
  - Learning must be slower along parameter 2
- Use a different learning rate for each parameter?

# AdaGrad

- Accumulate squared gradients:

$$r_i = r_i + g_i^2$$

- Update each parameter:

$$\theta_i = \theta_i - \frac{\varepsilon}{\delta + \sqrt{r_i}} g_i$$

Inversely proportional to cumulative squared gradient

- Greater progress along gently sloped directions

**Advantage:**

1.No need to update the learning rate manually as it changes adaptively with iterations.

**Disadvantage:**

1.As the number of iteration becomes very large learning rate decreases to a very small number which leads to slow convergence.

# AdaDelta

- Adadelta is an extension of Adagrad that attempts to solve its radically diminishing learning rates.
- The idea behind Adadelta is that instead of summing up all the past squared gradients from 1 to “t” time steps, what if we could restrict the window size.
- For example, computing the squared gradient of the past 10 gradients and average out. This can be achieved using **Exponentially Weighted Averages** over Gradient**.**

# RMSProp

- For non-convex problems, AdaGrad can prematurely decrease learning rate
- Use exponentially weighted average for gradient accumulation

$$r_i = \rho r_i + (1-\rho) g_i^2$$

$$\theta_i = \theta_i - \frac{\varepsilon}{\delta + \sqrt{r_i}} g_i$$

# Adam

- RMSProp + Momentum
- Estimate first moment: $v_i = \rho_1 v_i + (1-\rho_1) g_i$

Also applies bias correction to $v$ and $r$

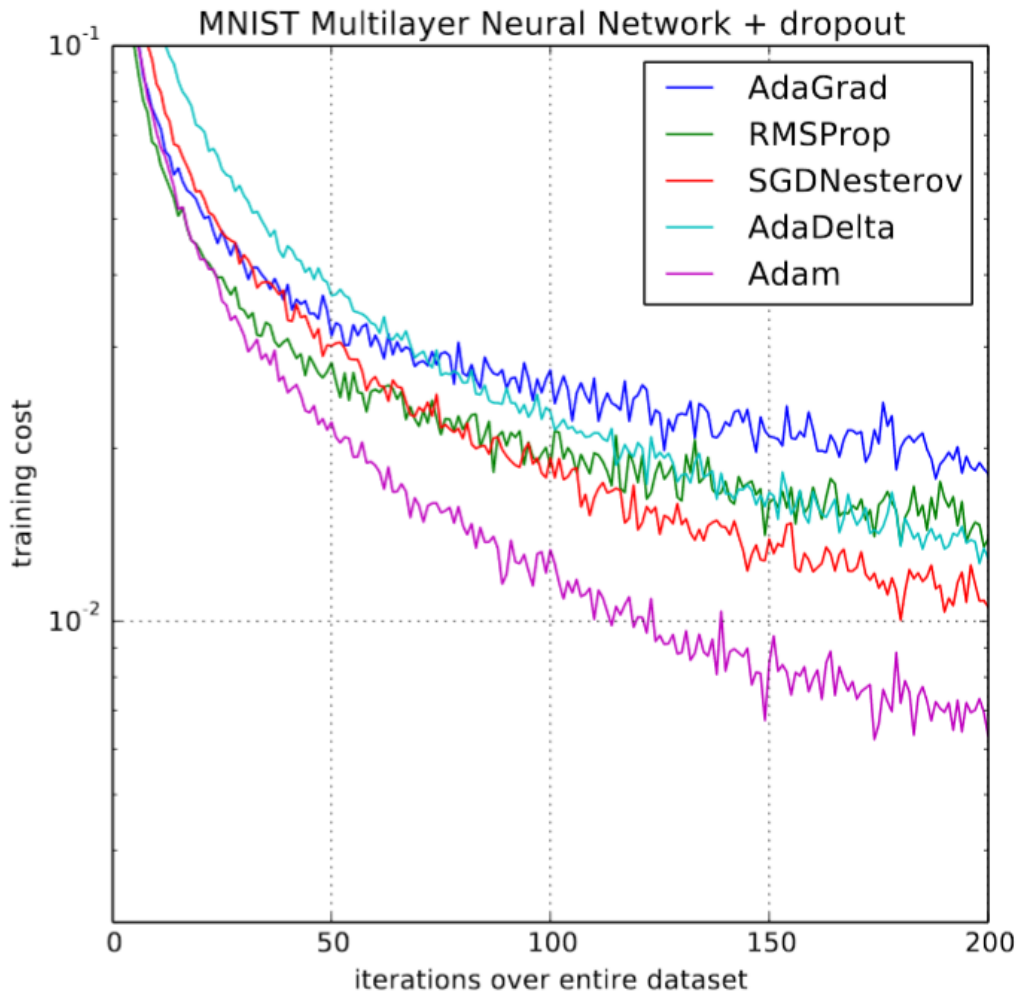
- Estimate second moment: $r_i = \rho_2 r_i + (1-\rho_2) g_i^2$

- Update parameters:

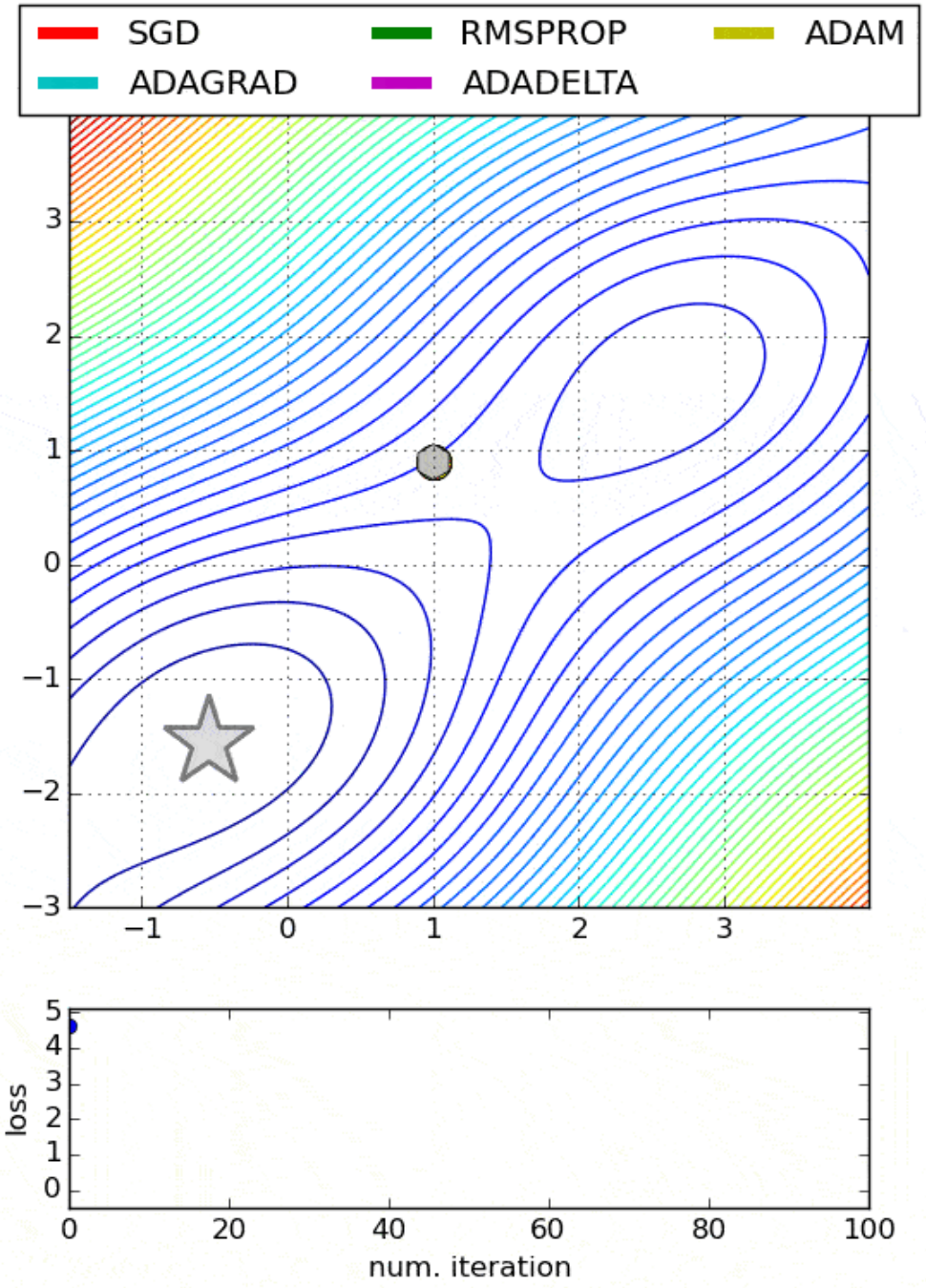
$$\theta_i = \theta_i - \frac{\varepsilon}{\delta + \sqrt{r_i}} v_i$$

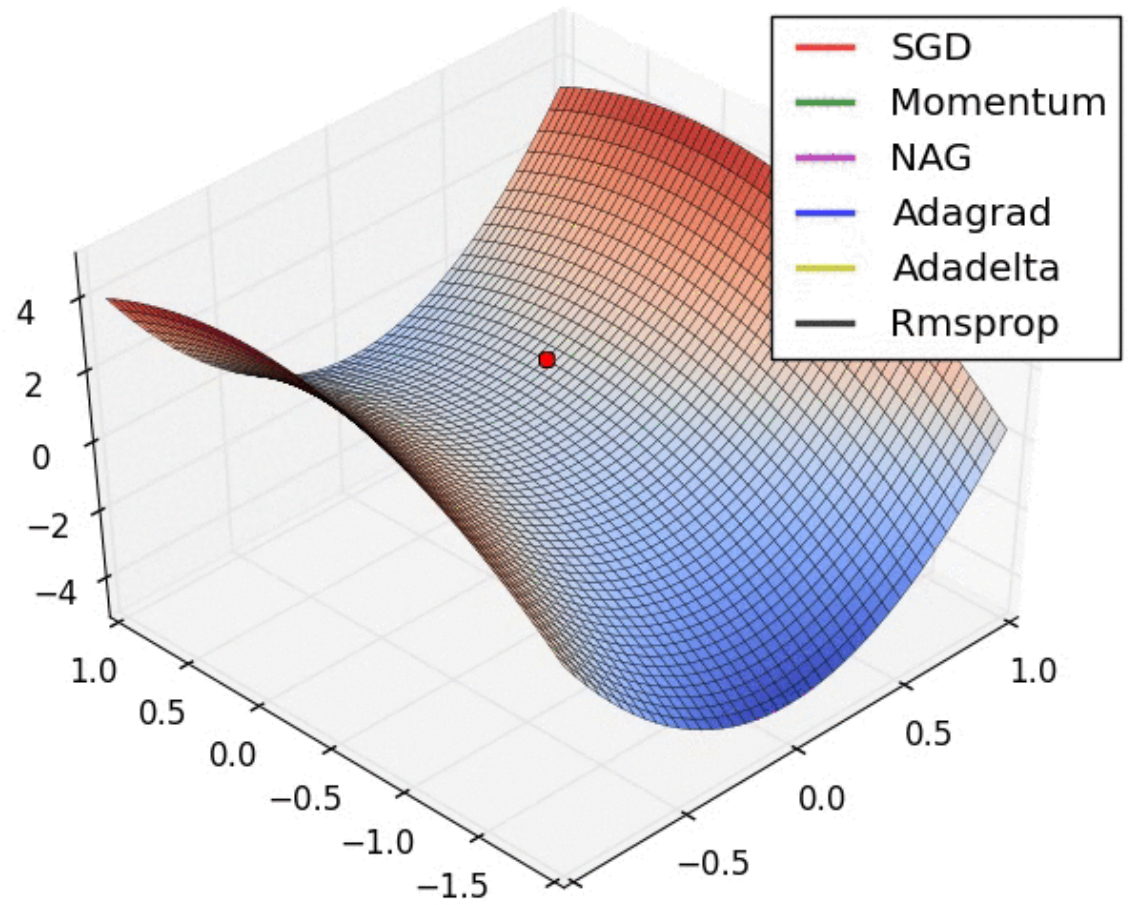
Works well in practice, is fairly robust to hyper-parameters

# Adam is a Good Default Choice



# Comparison between various optimizers




SGD
Momentum
NAG
Adagrad
Adadelta
Rmsprop
4
2
0
−2
−4
1.0
0.5
0.0
−0.5
−1.0
−1.5
1.0
0.5
0.0
−0.5

# Which optimizer should we use?

**Adam** *works well in practice and outperforms other Adaptive techniques.*

There are newer flavors of Adam: **NAdam**, **AdamW**…

If your input data is sparse then methods such as **SGD,NAG and momentum** are inferior and perform poorly. **For sparse data sets one should use one of the *adaptive learning-rate* methods**. An additional benefit is that we won't need to adjust the learning rate but likely achieve the best results with the default value.

If one wants fast convergence and train a deep Neural Network Model or a highly complex Neural Network then **Adam or any other Adaptive learning rate techniques** should be used because they outperforms every other optimization algorithms.

# Large Batch Optimizers

- As batch size grows, the number of iterations per epoch decreases.

Layerwise Adaptive Rate Scaling (LARS) explain their trick to solve this problem:

*To analyze the training stability with large LRs we measured the ratio between the norm of the layer weights and norm of gradients update. We observed that if this ratio is too high, the training may become unstable. On the other hand, if the ratio is too small, then weights don't change fast enough.*

They call this ratio the "trust ratio". When it's higher, the gradients change faster and vice versa.

https://arxiv.org/pdf/1708.03888.pdf

# LAMB

- LAMB stands for “Layer-wise Adaptive Moments optimizer for Batch training.” It makes a few small changes to LARS

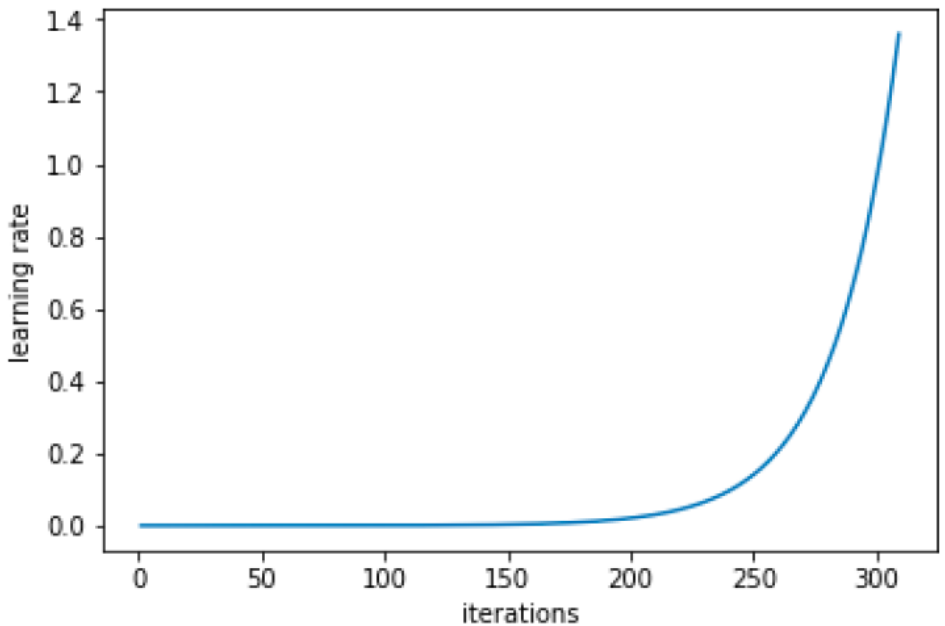
1. If the numerator ($r_1$) or denominator ($r_2$) of the trust ratio is 0, then use 1 instead.
2. Fixing weight decay: in LARS, the denominator of the trust ratio is $|\nabla L| + \beta |w|$, whereas in LAMB it’s $|\nabla L + \beta w|$. This preserves more information.
3. Instead of using the SGD update rule, they use the Adam update rule.
4. Clip the trust ratio at 10.

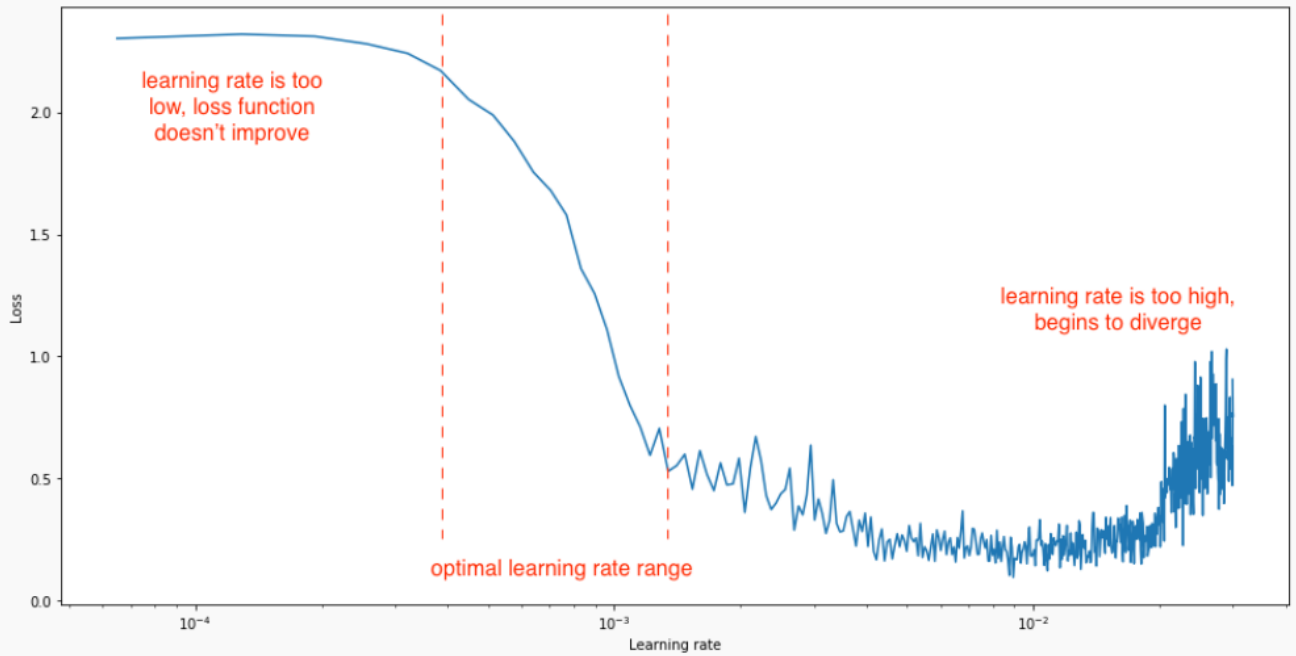
$$
\begin{aligned}
r_1 &= \|w_{t-1}^l\|_2 \\
r_2 &= \|\frac{\hat{m}_t^l}{\sqrt{\hat{v}_t^l + \epsilon}} + \lambda w_{t-1}^l\|_2 \\
r &= r_1 / r_2 \\
\eta^l &= r \times \eta \\
w_t^l &= w_{t-1}^l - \eta^l \times (\frac{\hat{m}_t^l}{\sqrt{\hat{v}_t^l + \epsilon}} + \lambda w_{t-1}^l)
\end{aligned}
$$

https://arxiv.org/pdf/1904.00962.pdf

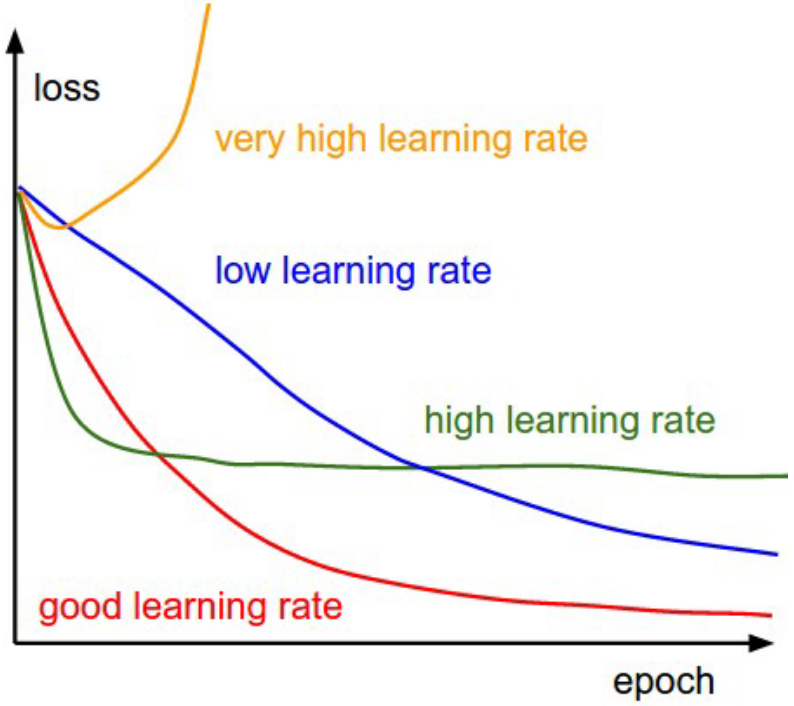
# Estimating the learning rate

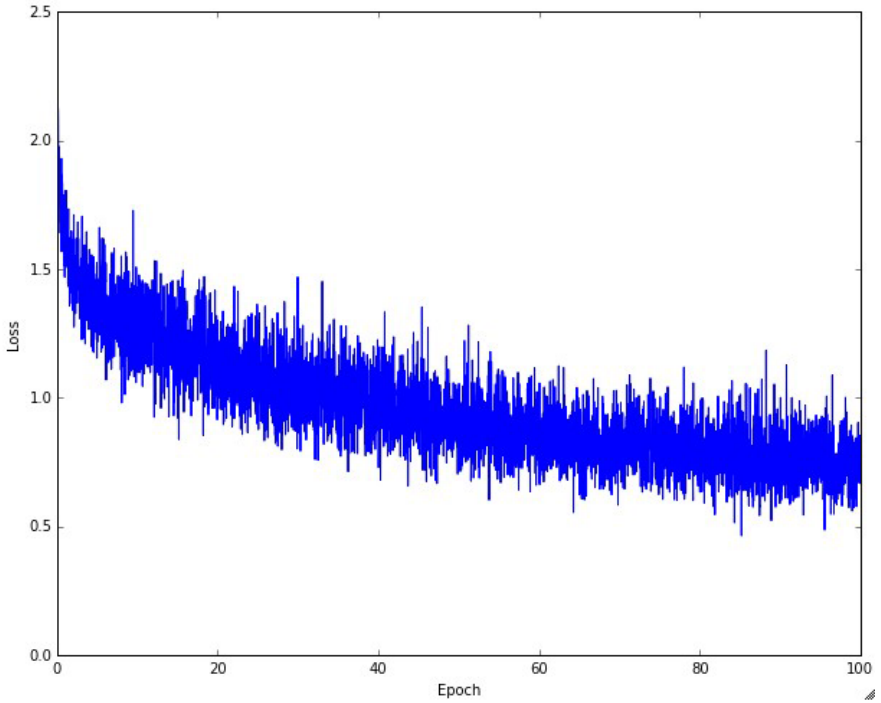
- How can we get a good LR estimate?
- Start with a small LR and increase it on every batch exponentially.
- Simultaneously, compute loss function on validation set.
- This also works for finding bounds for cyclic LRs & curriculum learning.


learning rate
iterations
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
0
50
100
150
200
250
300


learning rate is too low, loss function doesn't improve
optimal learning rate range
learning rate is too high, begins to diverge
Loss
Learning rate
0.0
0.5
1.0
1.5
2.0
$10^{-4}$
$10^{-3}$
$10^{-2}$

# Optimal learning rate: Monitor loss function

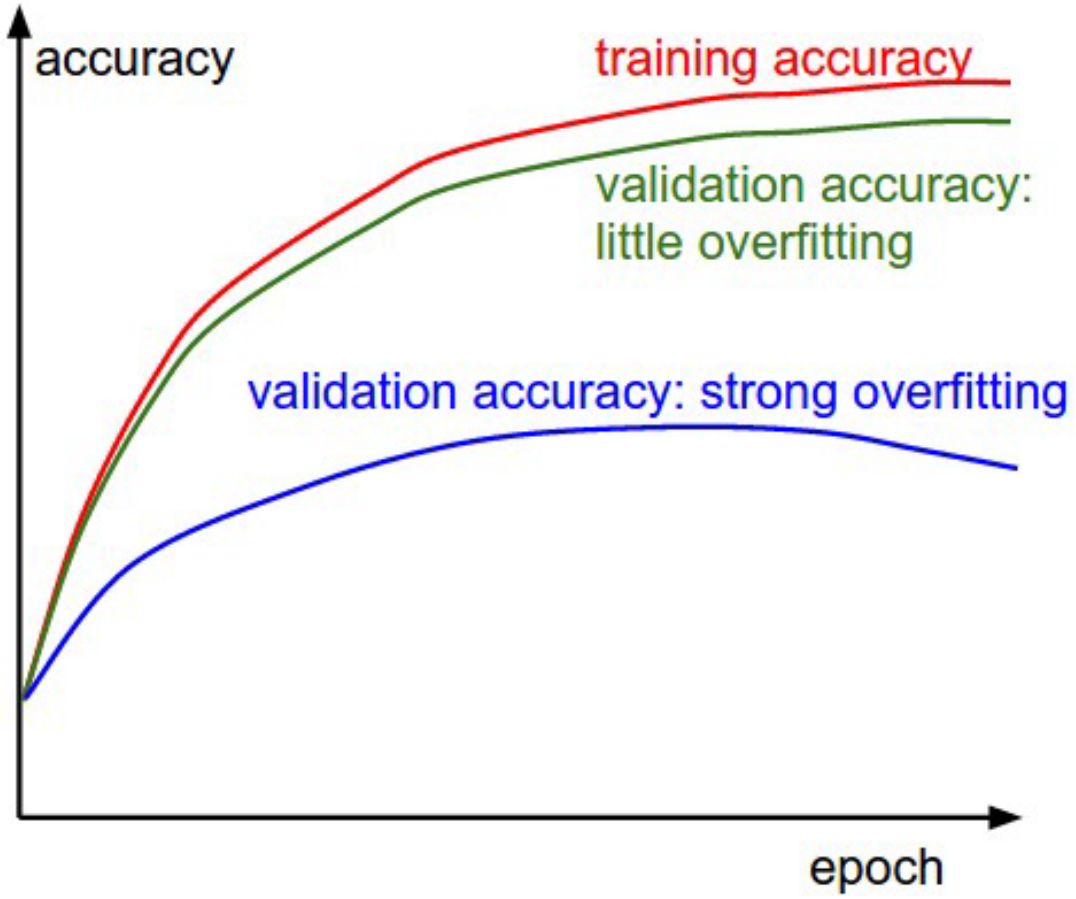



**Left:** A cartoon depicting the effects of different learning rates. With low learning rates the improvements will be linear. With high learning rates they will start to look more exponential. Higher learning rates will decay the loss faster, but they get stuck at worse values of loss (green line). This is because there is too much "energy" in the optimization and the parameters are bouncing around chaotically, unable to settle in a nice spot in the optimization landscape. **Right:** An example of a typical loss function over time.

From: https://cs231n.github.io/

# Train/Val accuracy




The gap between the training and validation accuracy indicates the amount of overfitting. Two possible cases are shown in the diagram on the left. The blue validation error curve shows very small validation accuracy compared to the training accuracy, indicating strong overfitting (note, it's possible for the validation accuracy to even start to go down after some point). When you see this in practice you probably want to increase regularization (stronger L2 weight penalty, more dropout, etc.) or collect more data. The other possible case is when the validation accuracy tracks the training accuracy fairly well. This case indicates that your model capacity is not high enough: make the model larger by increasing the number of parameters.

From: https://cs231n.github.io/

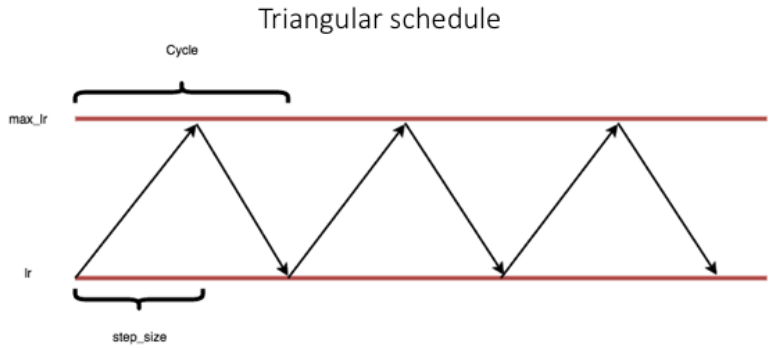
# Cyclical Learning Rates for Neural Networks

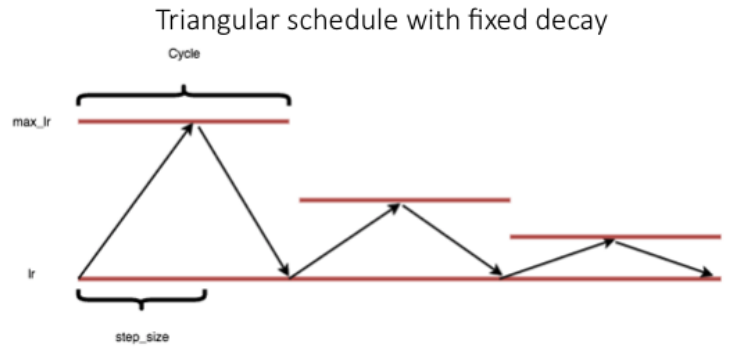
Use cyclical learning rates to escape local extreme points.

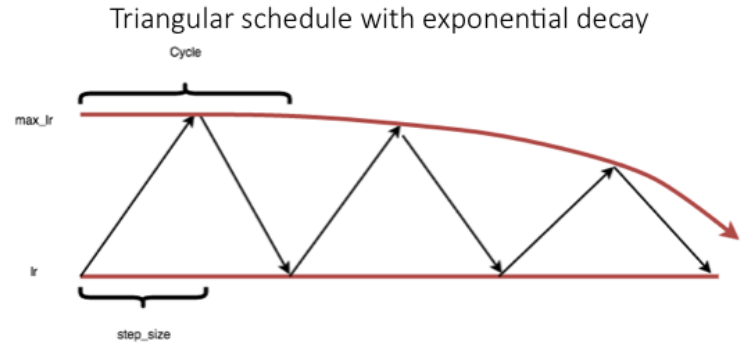
Saddle points are abundant in high dimensions, and convergence becomes very slow. Furthermore, they can help escape sharp local minima (overfitting).

Cyclic learning rates raise the learning rate periodically: short term negative effect and yet achieve a longer term beneficial effect.

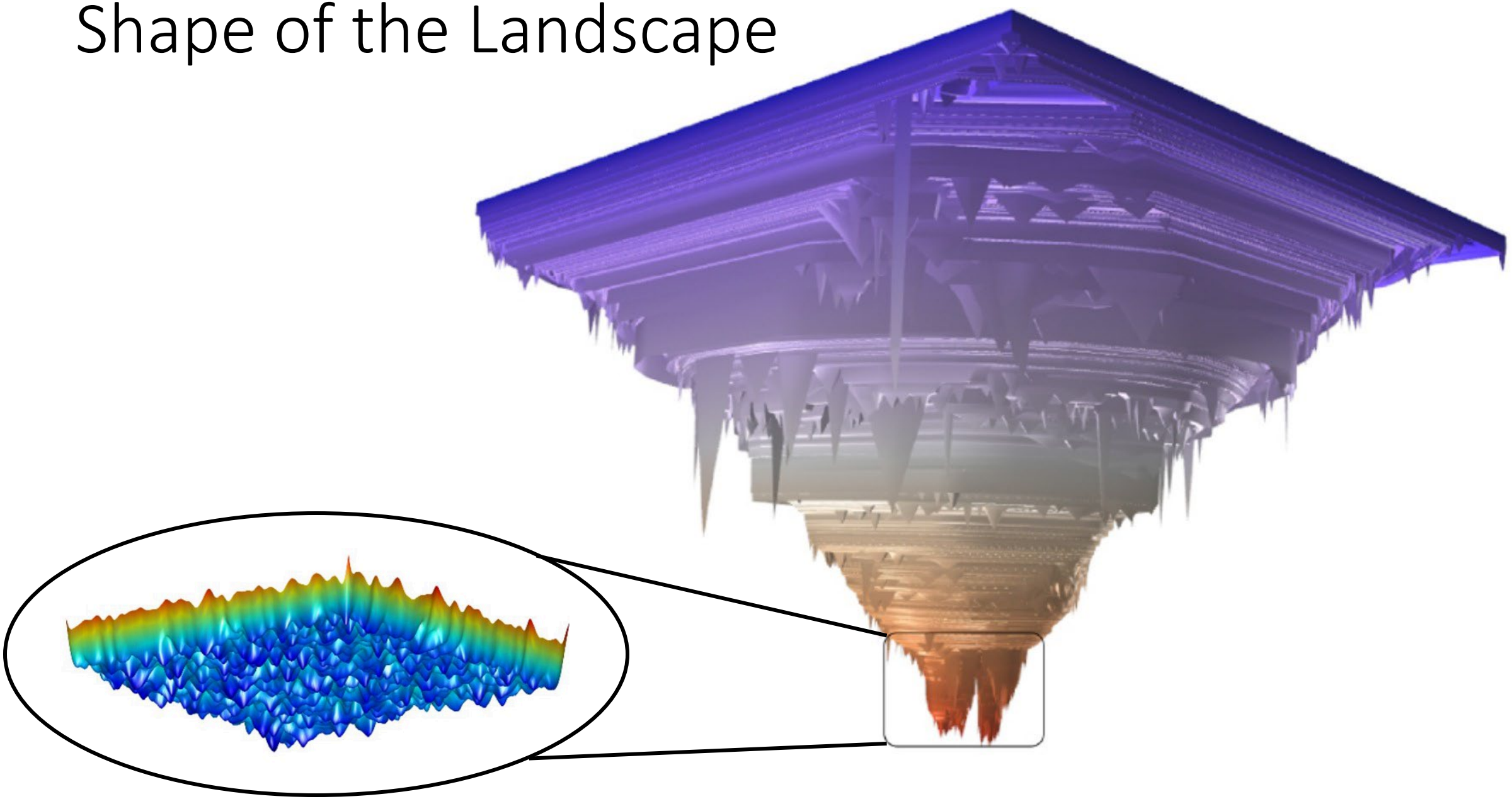
Decreasing learning rates may still help reduce error towards the end.



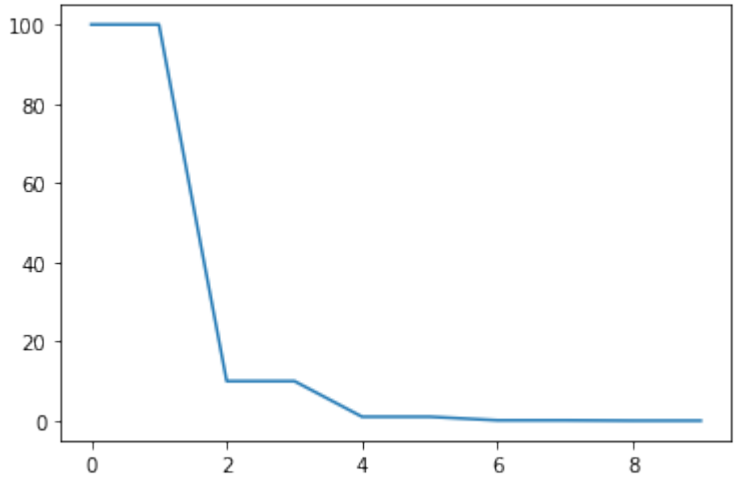



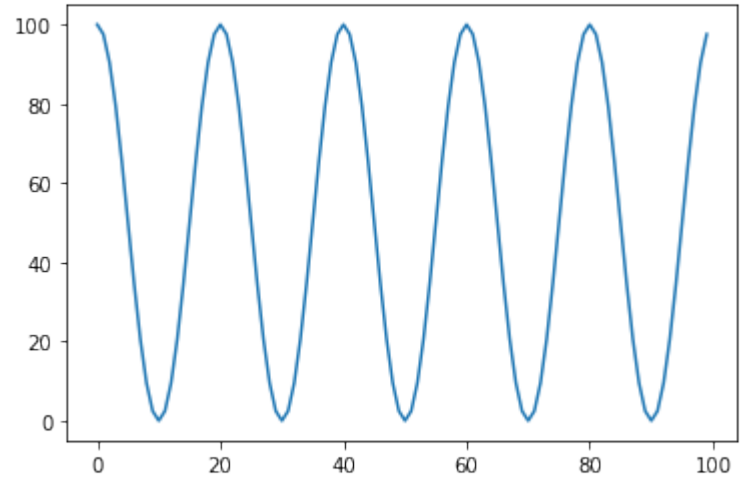
# Shape of the Landscape



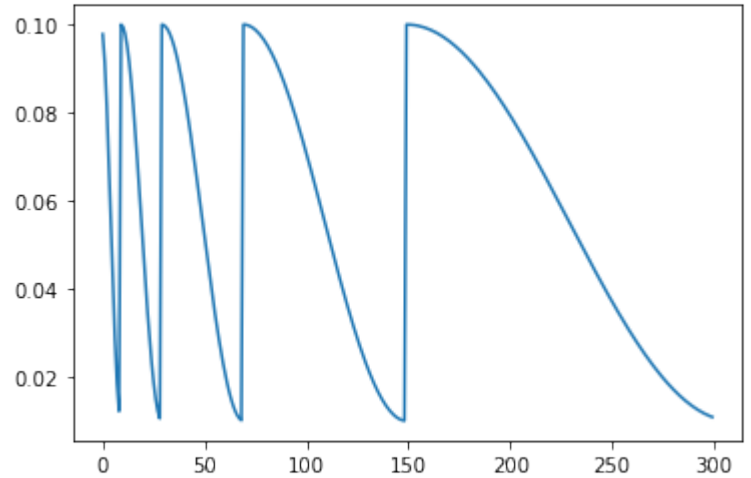
# No Need to Hand code LR Schedule



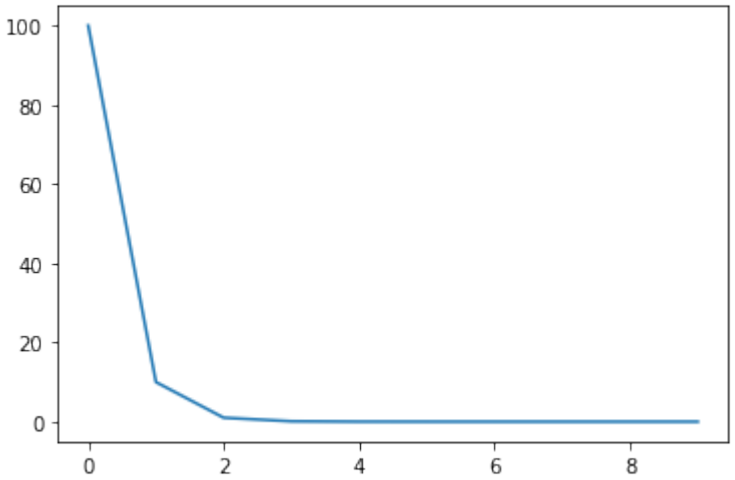

torch.optim.lr_scheduler.StepLR



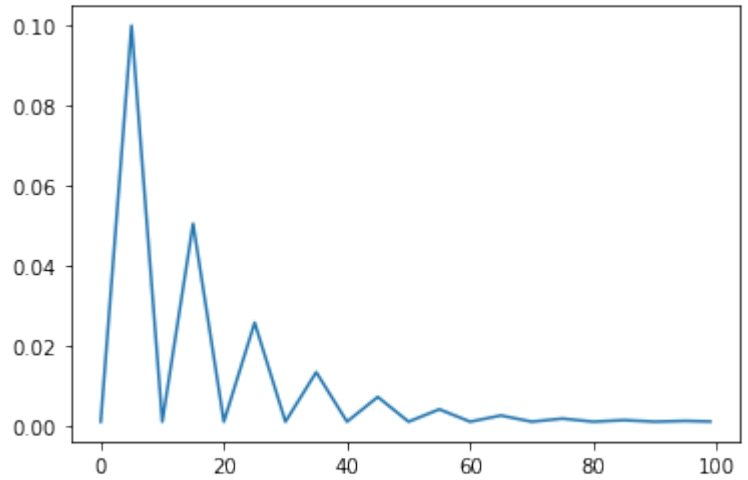

torch.optim.lr_scheduler.CosineAnnealingLR

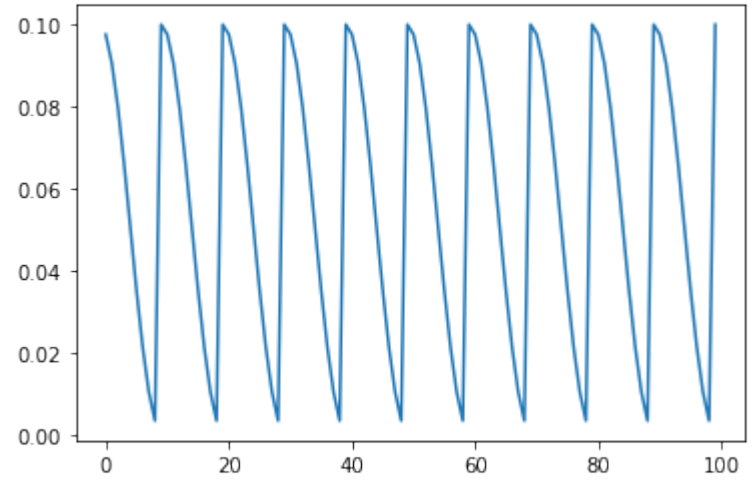



CosineAnnealingWarmRestarts




torch.optim.lr_scheduler.ExponentialLR




torch.optim.lr_scheduler.CyclicLR(…,mode="triangular2")

# Parameter Initialization

- Goal: break symmetry between units
  - so that each unit computes a different function
- Initialize all weights (not biases) randomly
  - Gaussian or uniform distribution
- Scale of initialization?
  - *Large* -> grad explosion, *Small* -> grad vanishing

# Xavier Initialization

Heuristic for all outputs to have unit variance

- For a fully-connected layer with $m$ inputs:

$$W_{ij} \sim N\left(0, \frac{1}{m}\right)$$

- For ReLU units, it is recommended:

$$W_{ij} \sim N\left(0, \frac{2}{m}\right)$$

https://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf

# He initialization

**He initialization:** we just simply multiply random initialization with

$$\sqrt{\frac{2}{size^{[l-1]}}}$$

$$W^{[l]} = np.random.randn(size\_l, size\_l\text{-}1) * np.sqrt(2/size\_l\text{-}1)$$

https://arxiv.org/abs/1502.01852

# Normalized Initialization

Fully-connected layer with $m$ inputs, $n$ outputs:

$$W_{ij} \sim U\left(-\sqrt{\frac{6}{m+n}}, \sqrt{\frac{6}{m+n}}\right)$$

Heuristic trades off between initialize all layers have same activation and gradient variance

Sparse variant when $m$ is large

- Initialize $k$ nonzero weights in each unit

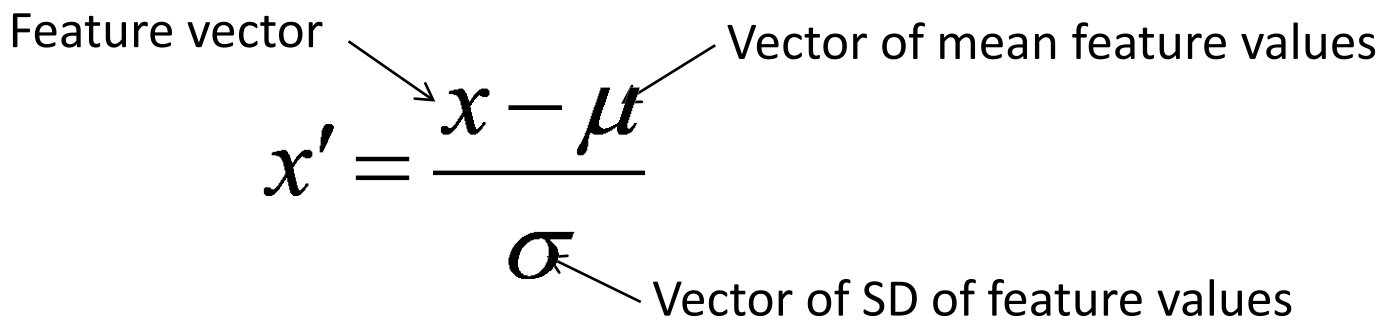
# How to do initializaiton in PyTorch?

Use a function from `torch.nn.init`

- `uniform_, normal_, constant_, ones_ and zeros_`
- `xavier_uniform_, xavier_normal_,`
- `kaiming_uniform_, kaiming_normal_`
- `orthogonal_`

# Feature Normalization

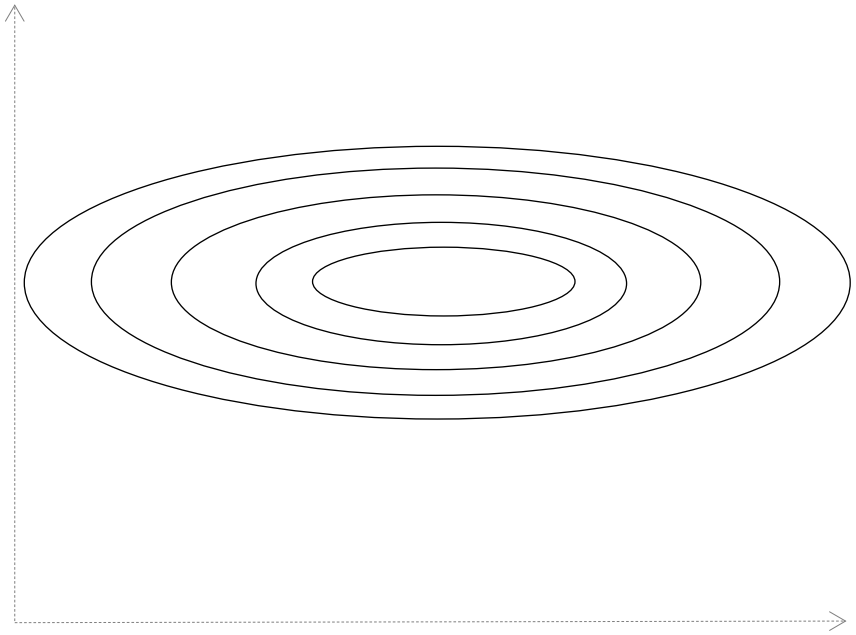
Good practice to normalize features before applying learning algorithm:



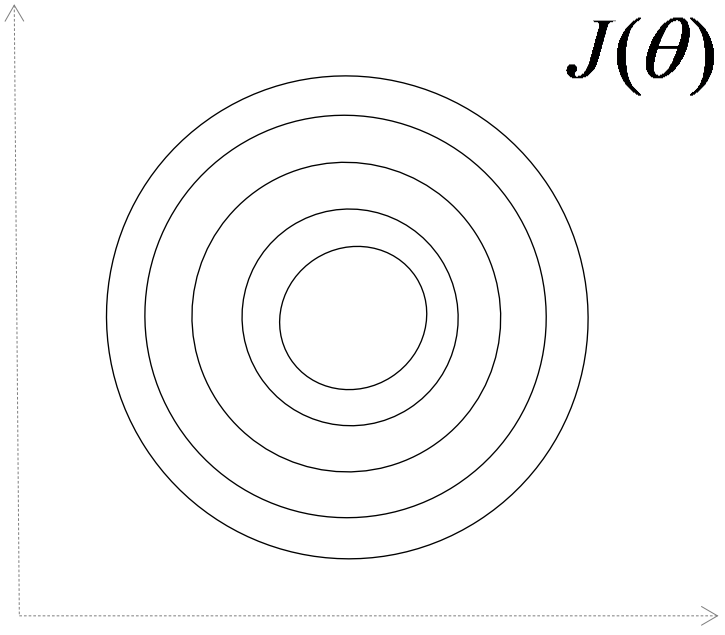

Features in same scale: mean 0 and variance 1

- Speeds up learning

# Feature Normalization



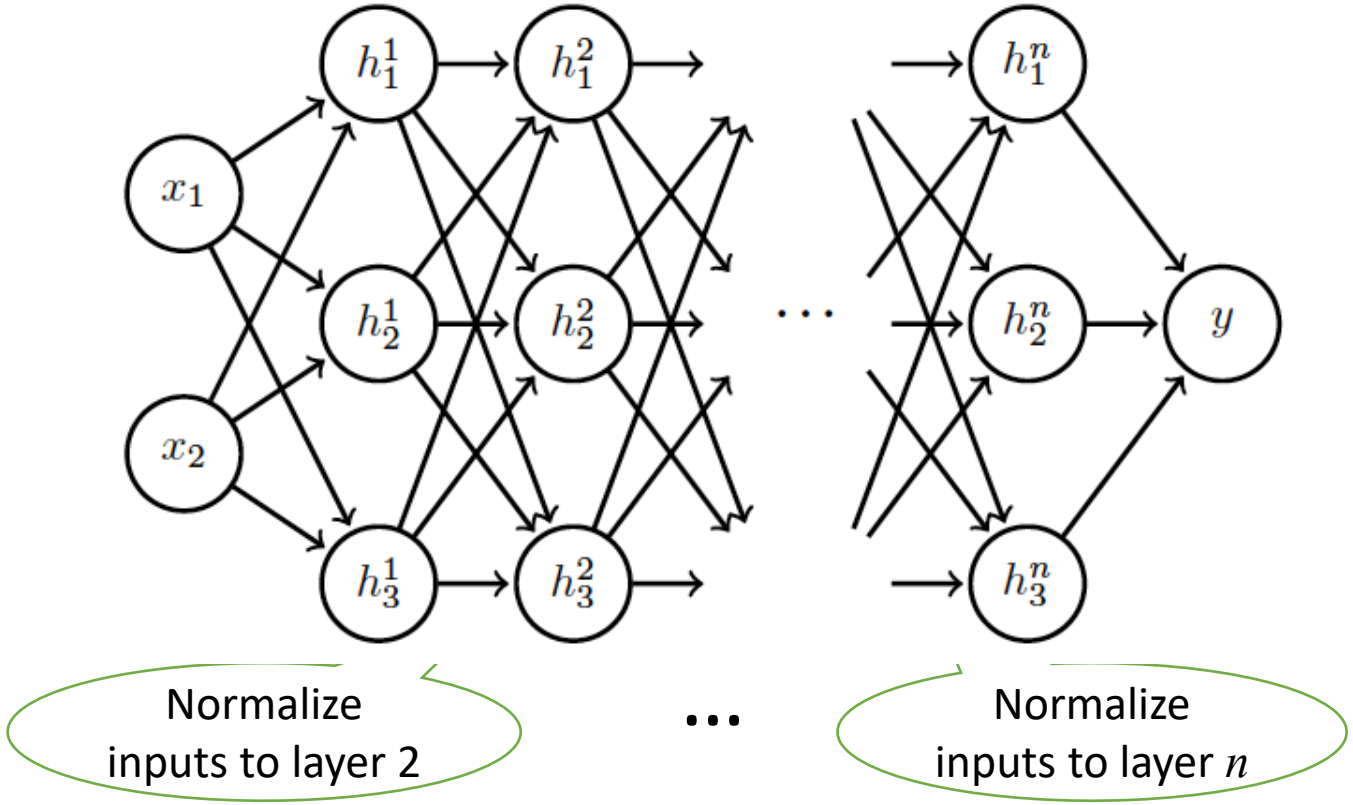
Before normalization




After normalization

# Internal Covariance Shift

- Each hidden layer changes distribution of inputs to next layer: *slows down learning*



# Batch Normalization

- Training time:
  - Mini-batch of activations for layer to normalize

$$H = \begin{bmatrix} H_{11} & \cdots & H_{1K} \\ \vdots & \ddots & \vdots \\ H_{N1} & \cdots & H_{NK} \end{bmatrix}$$

$K$ hidden layer activations

$N$ data points in mini-batch

# Batch Normalization

- Training time:
  - Mini-batch of activations for layer to normalize

where

$$H' = \frac{H - \mu}{\sigma}$$

$$\mu = \frac{1}{m}\sum_i H_{i,:} \qquad \sigma = \sqrt{\frac{1}{m}\sum_i (H - \mu)_i^2 + \delta}$$

Vector of mean activations across mini-batch

Vector of SD of each unit across mini-batch

# Batch Normalization

Training time:

- Normalization can reduce expressive power
- Instead use:

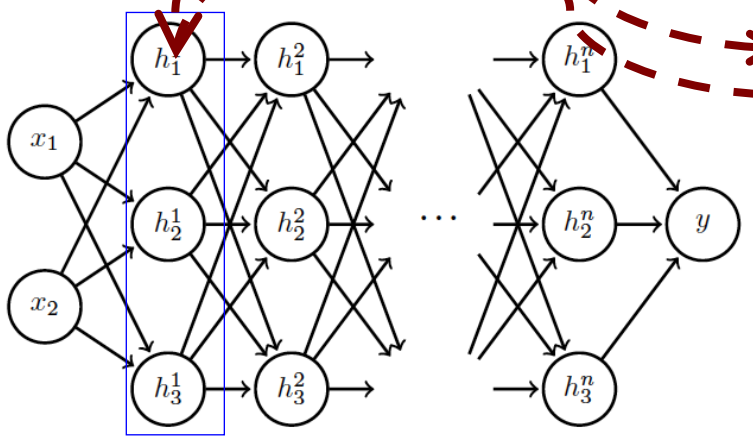
$$\gamma H' + \beta$$

Learnable parameters

Allows network to control range of normalization

# Batch Normalization

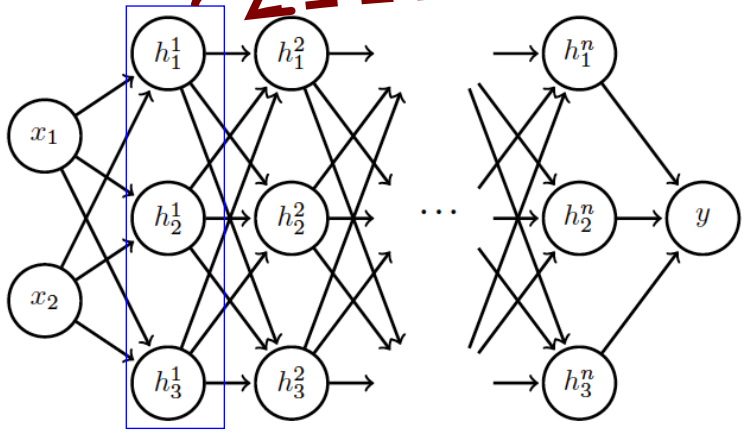
Batch 1




$$\mu^1 = \frac{1}{m}\sum_i H_{i,:}$$

$$\sigma^1 = \sqrt{\frac{1}{m}\sum_i (H-\mu)_i^2 + \delta}$$

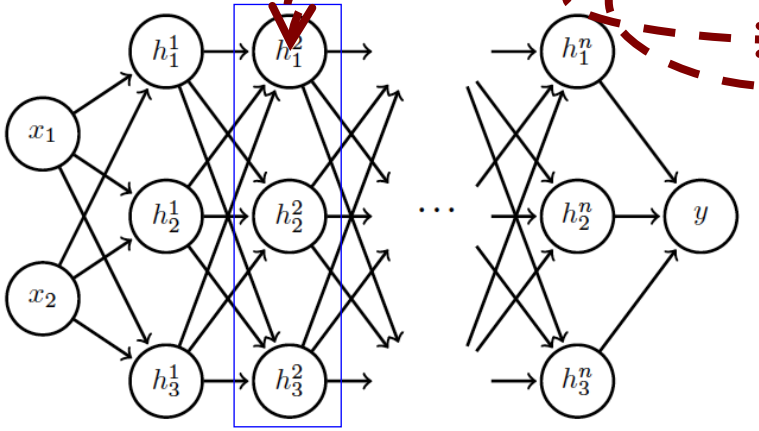
Batch *N*




Add normalization operations for layer 1

# Batch Normalization

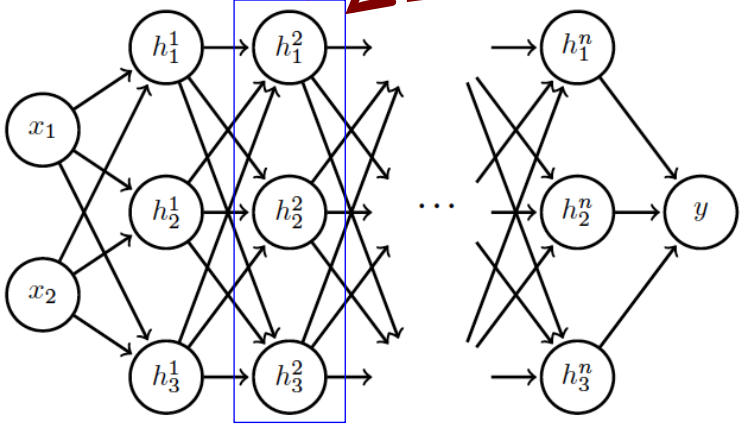
Batch 1



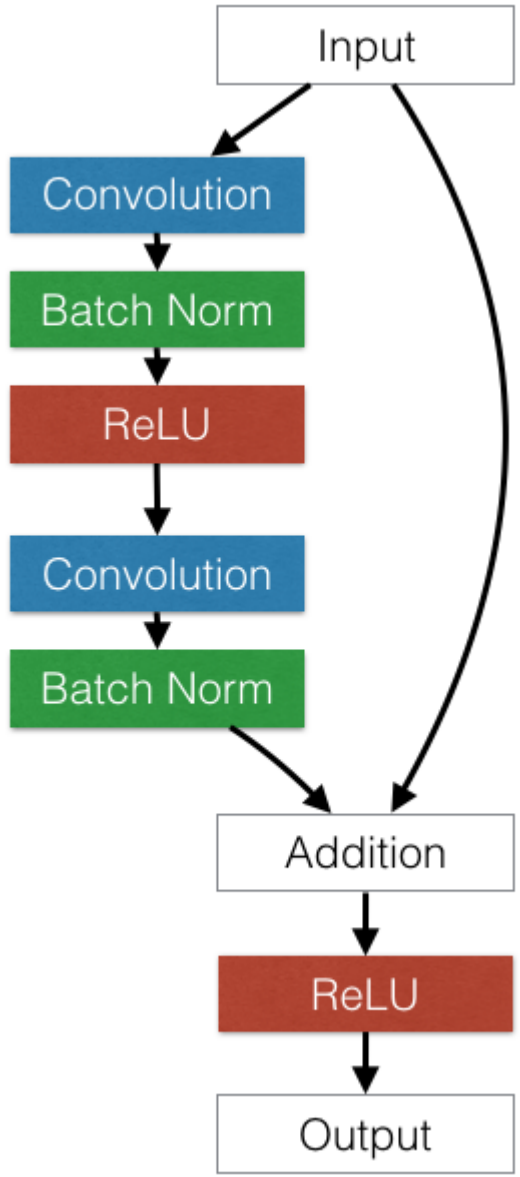

$$\mu^2 = \frac{1}{m}\sum_i H_{i,:}$$

$$\sigma^2 = \sqrt{\frac{1}{m}\sum_i (H-\mu)_i^2 + \delta}$$

Batch *N*




Add normalization operations for layer 2 and so on …


Input
Convolution
Batch Norm
ReLU
Convolution
Batch Norm
Addition
ReLU
Output