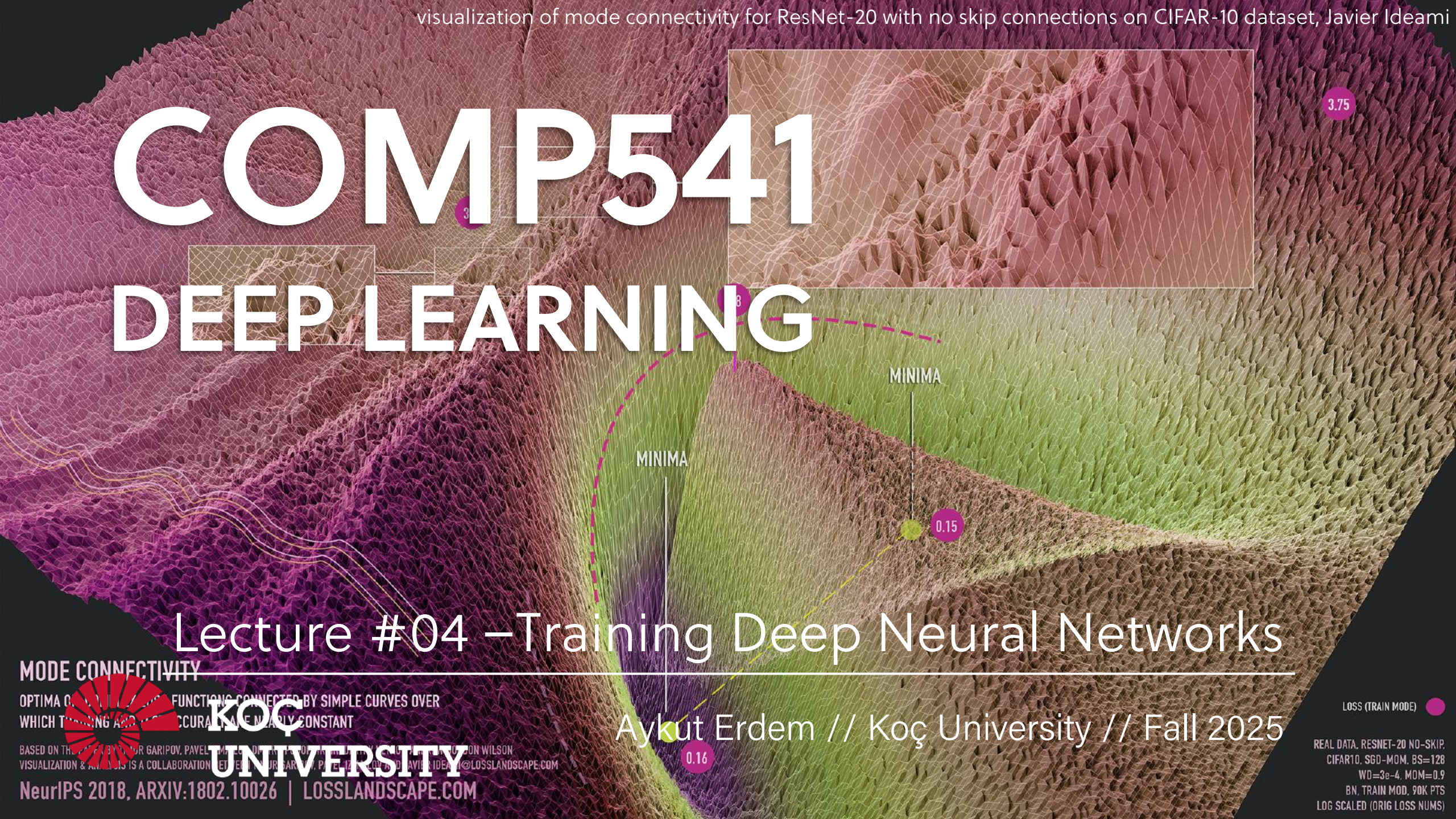
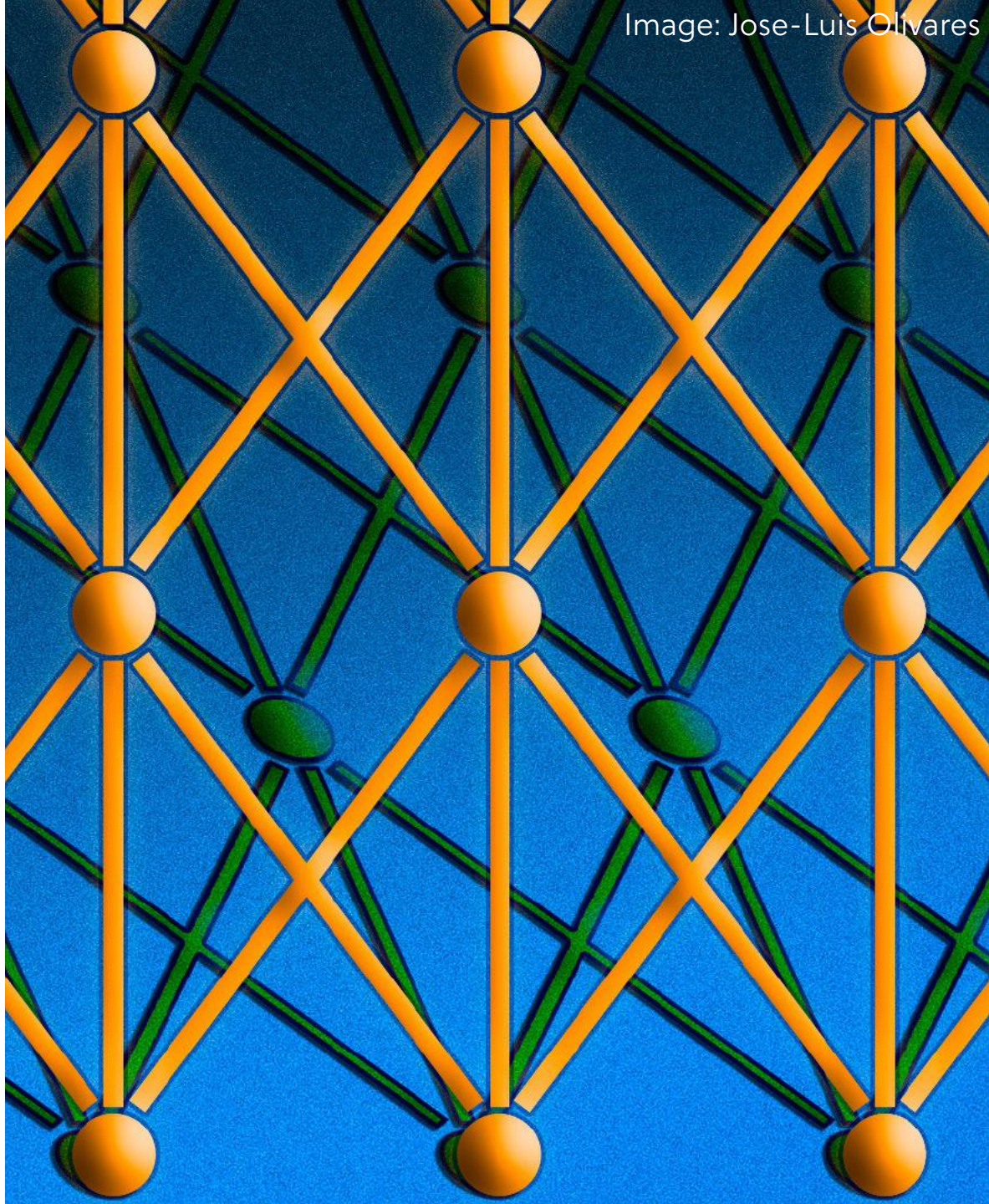


Lecture #04 – Training Deep Neural Networks

Aykut Erdem // Koç University // Fall 2025

Previously on COMP541

- multi-layer perceptrons
- activation functions
- chain rule
- backpropagation algorithm
- computational graph
- distributed word representations



Lecture overview

- data preprocessing and normalization
- weight initializations
- ways to improve generalization
- optimization

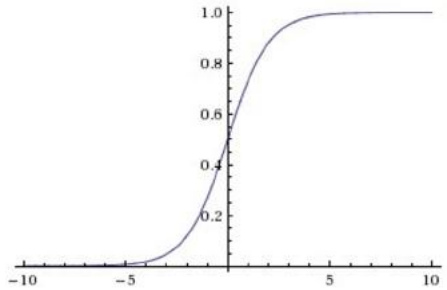
Disclaimer: Much of the material and slides for this lecture were borrowed from
—Fei-Fei Li, Andrej Karpathy and Justin Johnson's CS231n class
—Roger Grosse's CSC321 class
—Shubhendu Trivedi and Risi Kondor's CMSC 35246 class
—Efstratios Gavves and Max Welling's UvA deep learning class
—Hinton's Neural Networks for Machine Learning class
—Justin Johnson's EECS 498/598 class

Activation Functions

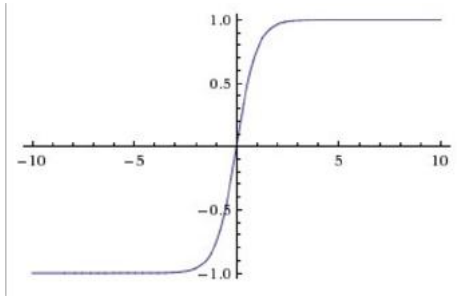
Activation Functions

Sigmoid

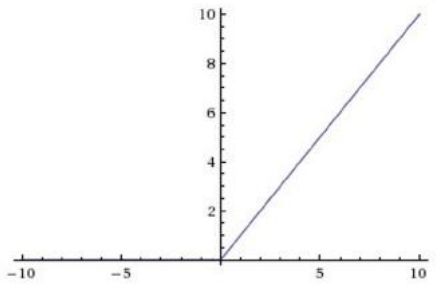
$$\sigma(x) = 1/(1 + e^{-x})$$



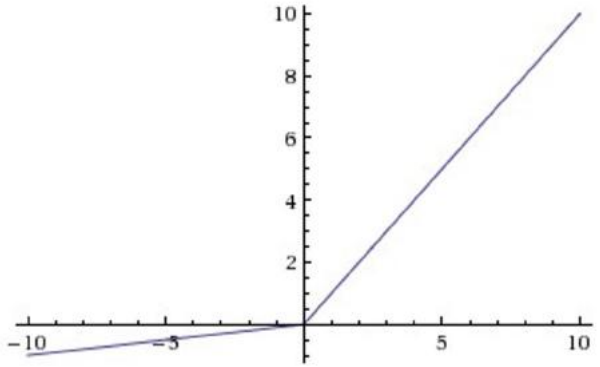
tanh tanh(x)



ReLU max(0,x)



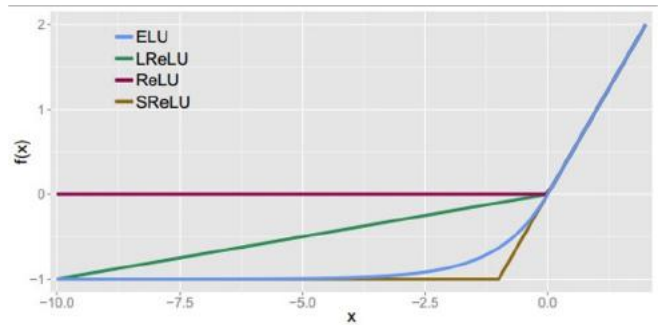
Leaky ReLU
max(0.1x, x)



Maxout $\max(w_1^T x + b_1, w_2^T x + b_2)$

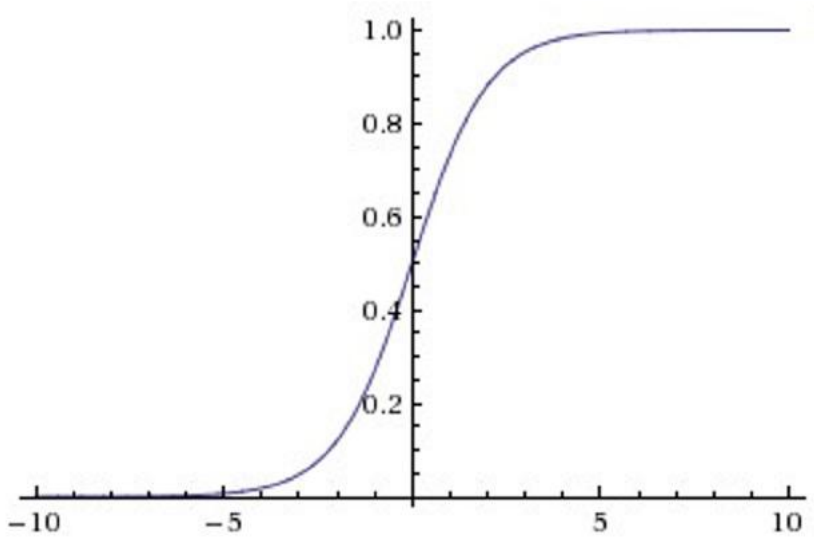
ELU

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha\,(\exp(x) - 1) & \text{if } x \leq 0 \end{cases}$$



Activation Functions

$$\sigma(x) = 1/(1 + e^{-x})$$



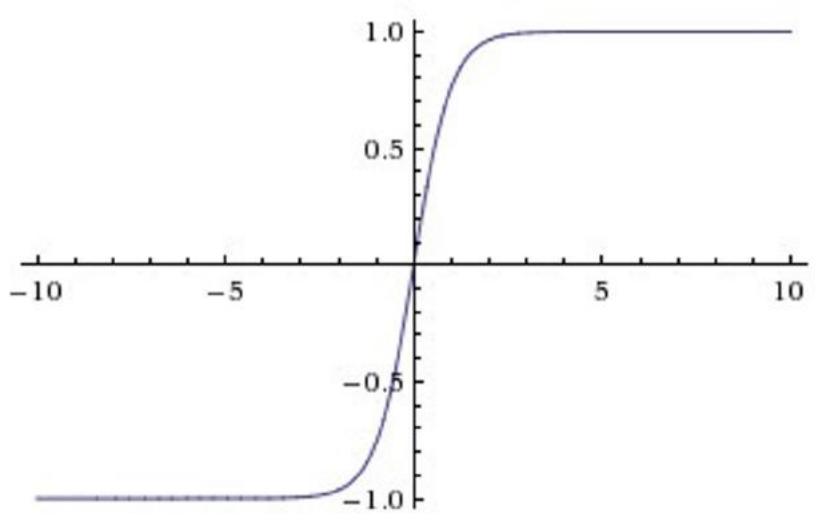
Sigmoid

- Squashes numbers to range [0,1]
- Historically popular since they have nice interpretation as a saturating "firing rate" of a neuron

<u>3 problems:</u>

1. Saturated neurons "kill" the gradients
2. Sigmoid outputs are not zero-centered
3. exp() is a bit compute expensive

Activation Functions



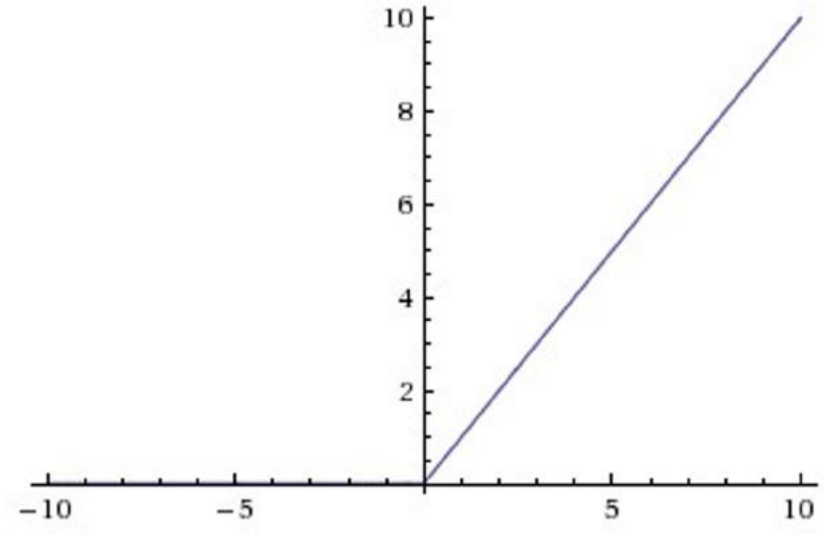
- Squashes numbers to range [-1,1]
- zero centered (nice)
- still kills gradients when saturated :(

tanh(x)

$$\tanh(x) = \frac{\exp(x) - \exp(-x)}{\exp(x) + \exp(-x)} = \frac{\exp(2x) - 1}{\exp(2x) + 1}$$

[LeCun et al., 1991]

Activation Functions



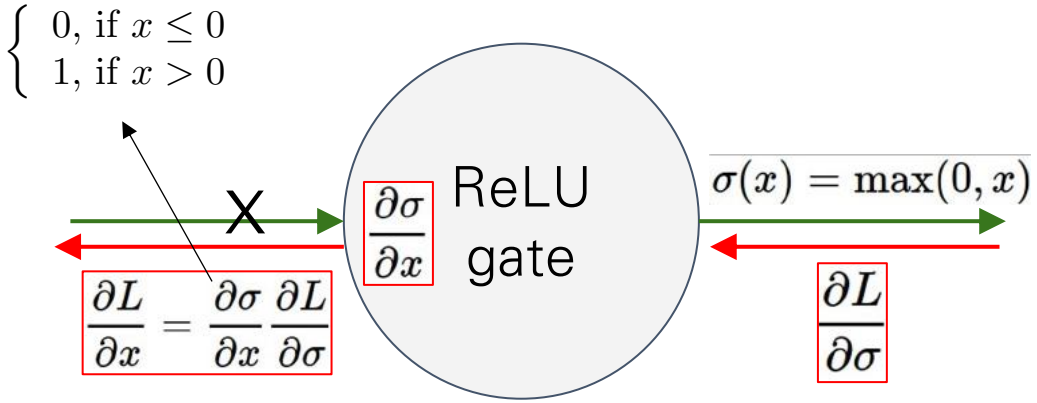
ReLU
(Rectified Linear Unit)

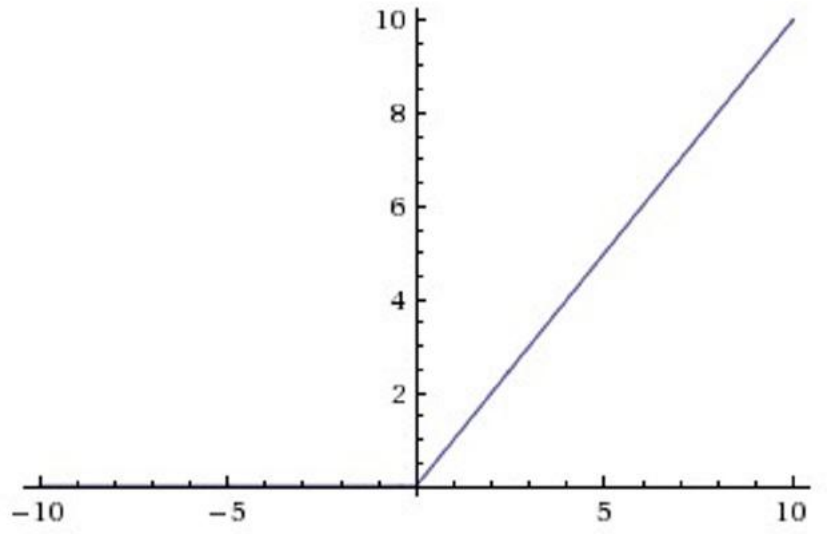
- Computes f(x) = max(0,x)

- Does not saturate (in +region)
- Very computationally efficient
- Converges much faster than sigmoid/tanh in practice (e.g. 6x)

[Krizhevsky et al., 2012]

Activation Functions





ReLU

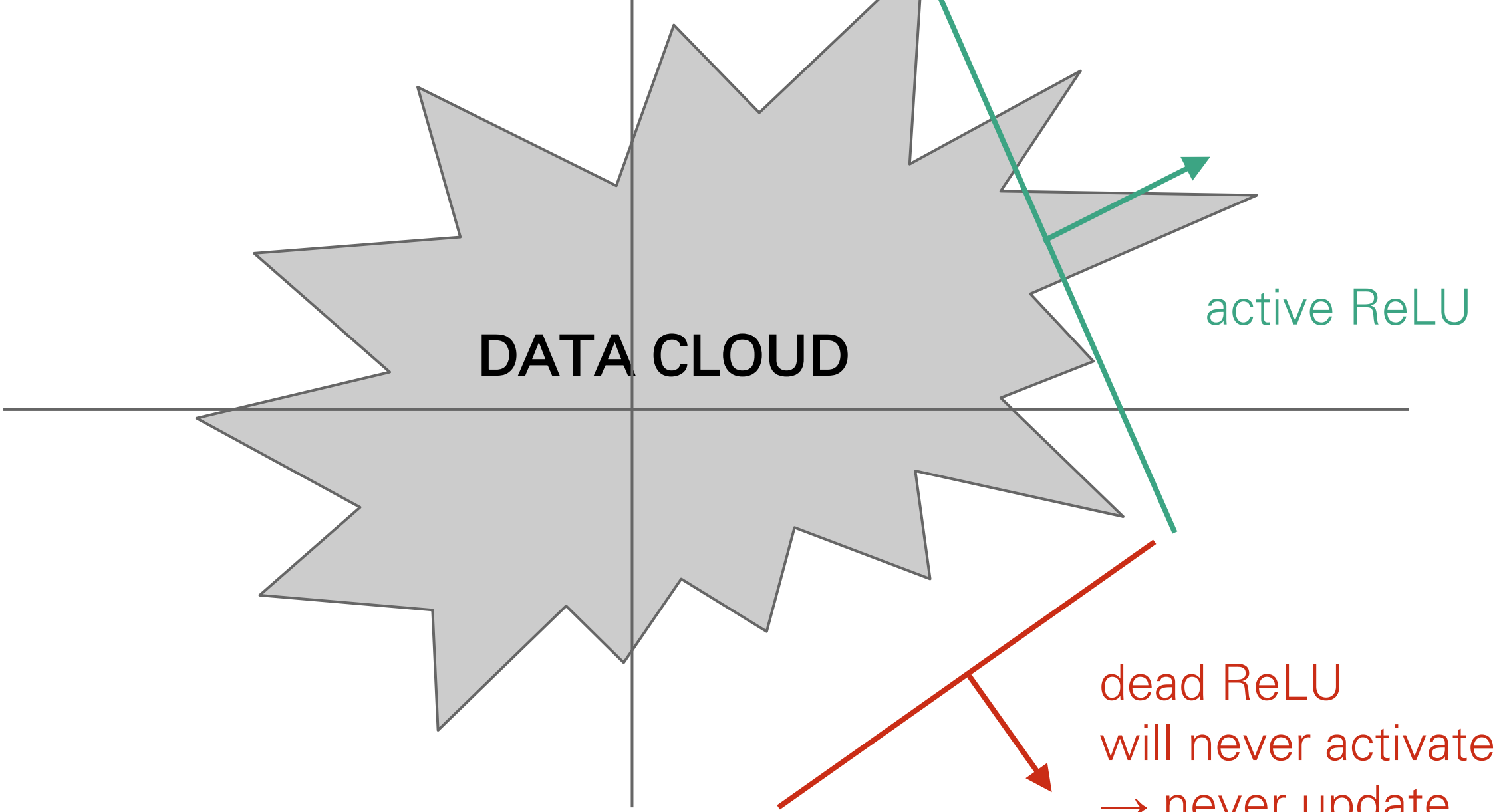
- Computes f(x) = max(0,x)

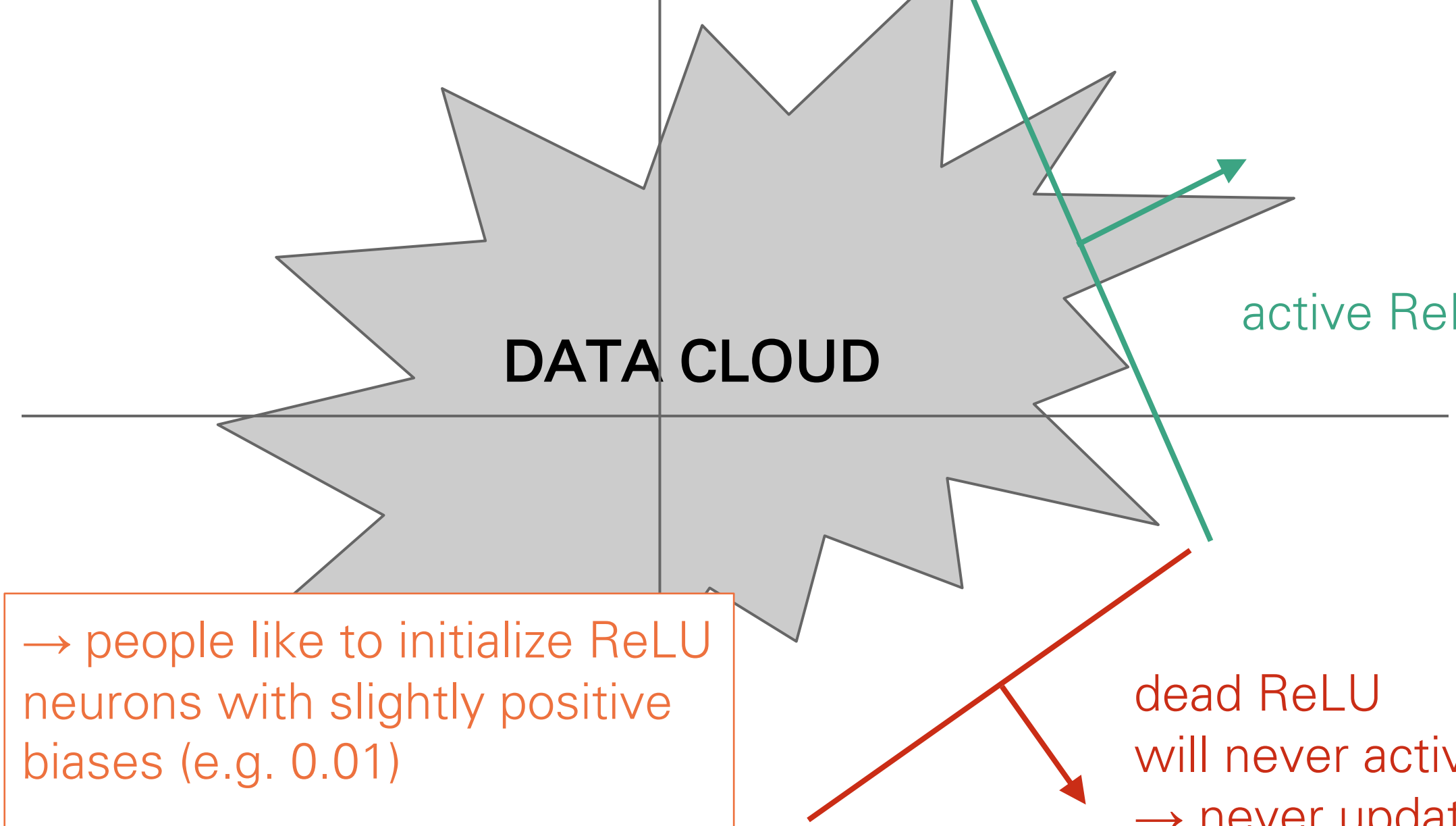
- Does not saturate (in +region)
- Very computationally efficient
- Converges much faster than sigmoid/tanh in practice (e.g. 6x)

- Not zero-centered output
- An annoyance:

Hint: what is the gradient when x < 0?

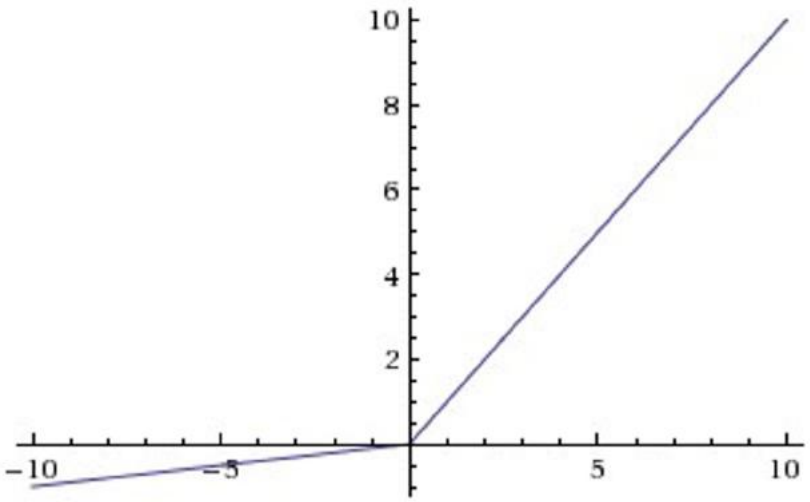
[Krizhevsky et al., 2012]


DATA CLOUD
active ReLU
dead ReLU
will never activate
→ never update



→ people like to initialize ReLU neurons with slightly positive biases (e.g. 0.01)

Activation Functions



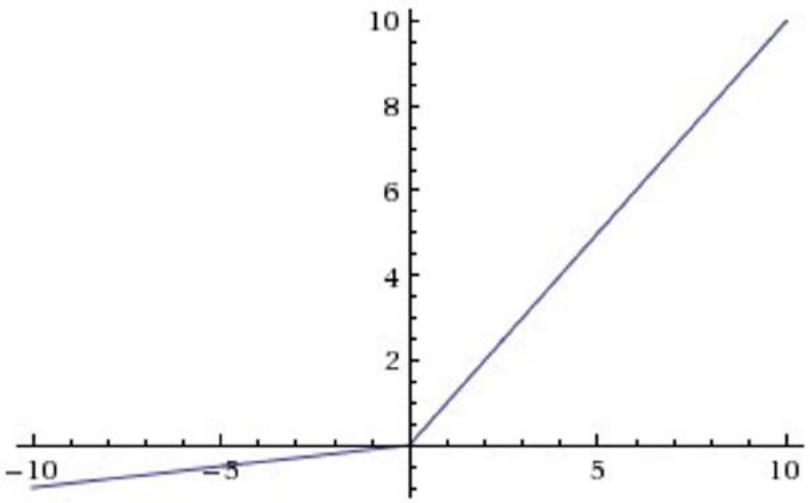
- Does not saturate
- Computationally efficient
- Converges much faster than sigmoid/tanh in practice! (e.g. 6x)
- will not “die”.

Leaky ReLU

$$f(x) = \max(0.01x, x)$$

[Mass et al., 2013]
[He et al., 2015]

Activation Functions



- Does not saturate
- Computationally efficient
- Converges much faster than sigmoid/tanh in practice! (e.g. 6x)
- will not "die".

Leaky ReLU

$$f(x) = \max(0.01x, x)$$

Parametric Rectifier (PReLU)

$$f(x) = \max(\alpha x, x)$$

backprop into \alpha (parameter)

[Mass et al., 2013]
[He et al., 2015]

Maxout "Neuron"

- Does not have the basic form of dot product -> nonlinearity
- Generalizes ReLU and Leaky ReLU
- Linear Regime! Does not saturate! Does not die!

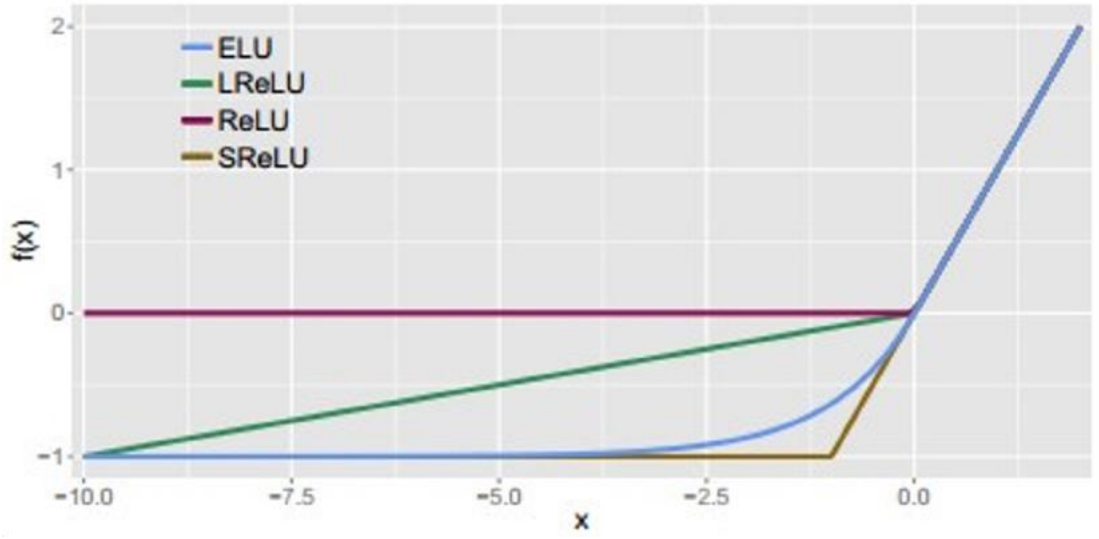
$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

Problem: doubles the number of parameters/neuron :(

[Goodfellow et al., 2013]

Activation Functions

Exponential Linear Units (ELU)



- All benefits of ReLU
- Does not die
- Closer to zero mean outputs

- Computation requires exp()

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha(\exp(x) - 1) & \text{if } x \leq 0 \end{cases}$$

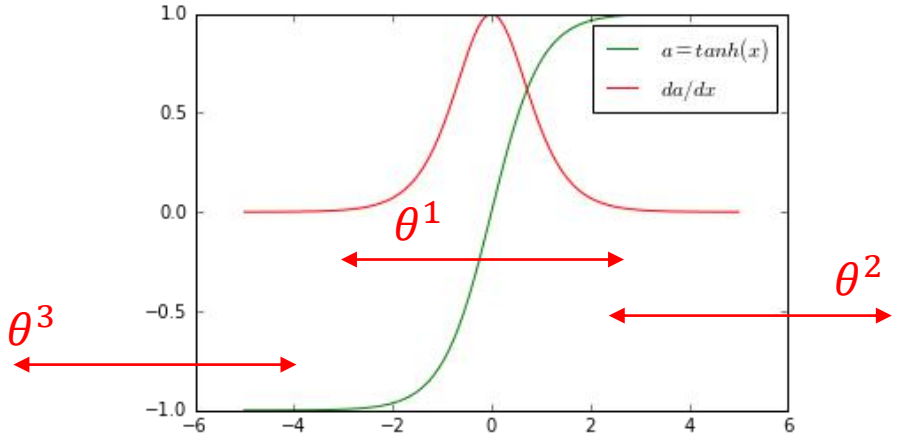
[Clevert et al., 2015]

Data Preprocessing and Normalization

Data preprocessing

- Scale input variables to have similar diagonal covariances $c_i = \sum_j (x_i^{(j)})^2$
  - Similar covariances → more balanced rate of learning for different weights
  - Rescaling to 1 is a good choice, unless some dimensions are less important

$x = [x^1, x^2, x^3]^T, \theta = [\theta^1, \theta^2, \theta^3]^T, a = \tanh(\theta^T x)$



x^1, x^2, x^3 → much different covariances

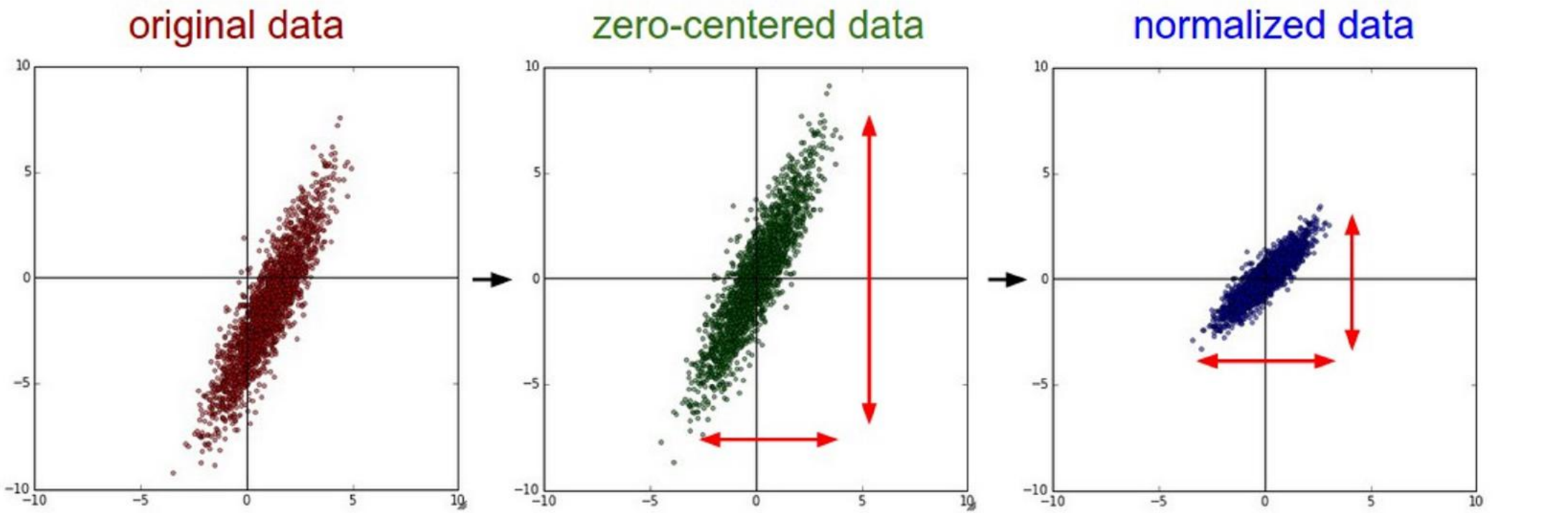
Generated gradients $\left.\frac{\partial \mathcal{L}}{\partial \theta}\right|_{x^1, x^2, x^3}$: much different

Gradient update harder: $\theta^{t+1} = \theta^t - \eta_t \begin{bmatrix} \partial \mathcal{L}/\partial \theta^1 \\ \partial \mathcal{L}/\partial \theta^2 \\ \partial \mathcal{L}/\partial \theta^3 \end{bmatrix}$

Data preprocessing

- Input variables should be as decorrelated as possible
  - Input variables are “more independent”
  - Network is forced to find non-trivial correlations between inputs
  - Decorrelated inputs → Better optimization
  - Obviously not the case when inputs are by definition correlated (sequences)

Data preprocessing



(Assume X [NxD] is data matrix, each example in a row)

TLDR: In practice for Images: center only

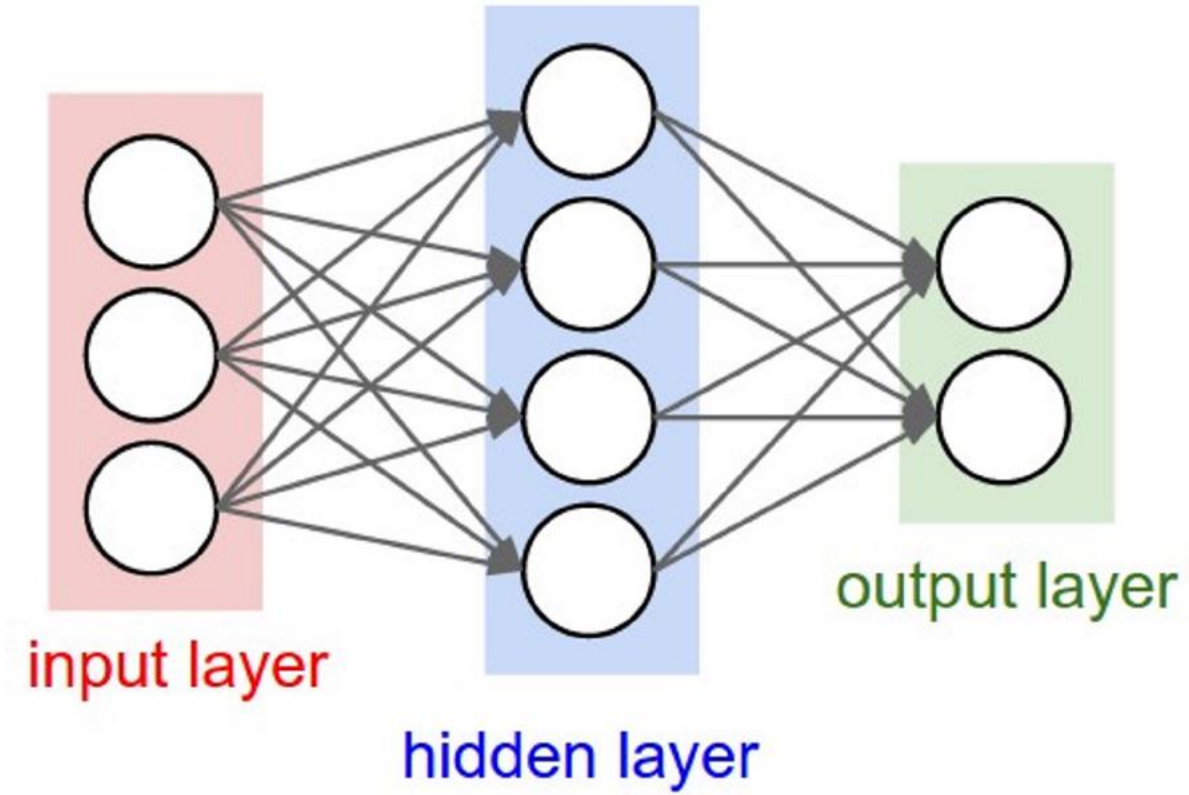
e.g. consider CIFAR-10 example with [32,32,3] images

- Subtract the mean image (e.g. AlexNet)
  (mean image = [32,32,3] array)
- Subtract per-channel mean (e.g. VGGNet)
  (mean along each channel = 3 numbers)

Not common to normalize variance, to do PCA or whitening

Weight Initialization

Q: what happens when W=0 init is used?



First idea: Small random numbers

(Gaussian with zero mean and 1e-2 standard deviation)

```
W = 0.01* np.random.randn(D,H)
```

First idea: Small random numbers

(Gaussian with zero mean and 1e-2 standard deviation)

```
W = 0.01* np.random.randn(D,H)
```

Works ~okay for small networks, but can lead to non-homogeneous distributions of activations across the layers of a network.

Lets look at some activation statistics

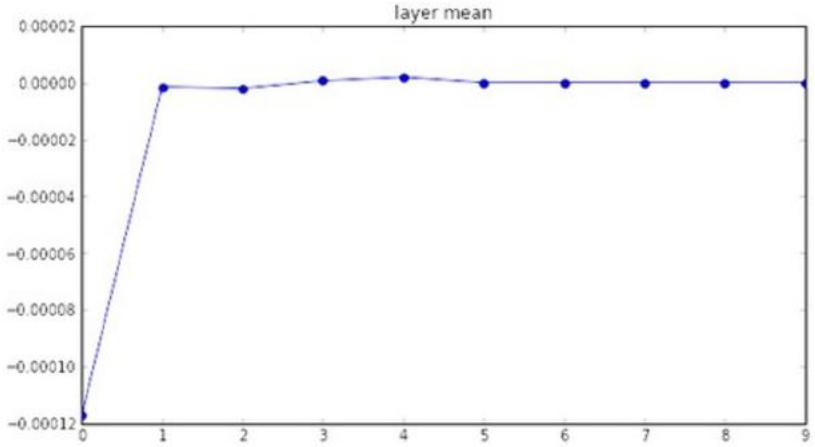
E.g. 10-layer net with 500 neurons on each layer, using tanh non-linearities, and initializing as described in last slide.

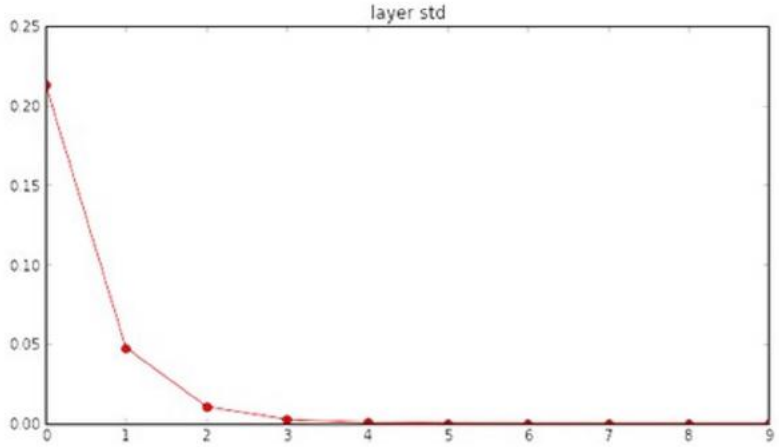
```
# assume some unit gaussian 10-D input data
D = np.random.randn(1000, 500)
hidden_layer_sizes = [500]*10
nonlinearities = ['tanh']*len(hidden_layer_sizes)
```

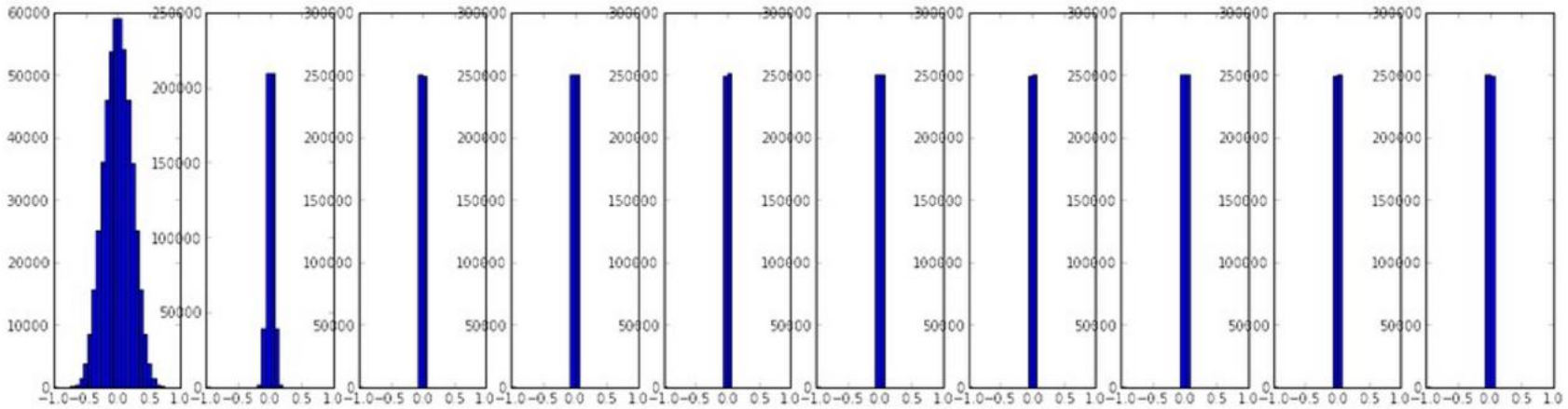
```
act = {'relu':lambda x:np.maximum(0,x), 'tanh':lambda x:np.tanh(x)}
Hs = {}
for i in xrange(len(hidden_layer_sizes)):
    X = D if i == 0 else Hs[i-1] # input at this layer
    fan_in = X.shape[1]
    fan_out = hidden_layer_sizes[i]
    W = np.random.randn(fan_in, fan_out) * 0.01 # layer initialization

    H = np.dot(X, W) # matrix multiply
    H = act[nonlinearities[i]](H) # nonlinearity
    Hs[i] = H # cache result on this layer
```

```
# look at distributions at each layer
print 'input layer had mean %f and std %f' % (np.mean(D), np.std(D))
layer_means = [np.mean(H) for i,H in Hs.iteritems()]
layer_stds = [np.std(H) for i,H in Hs.iteritems()]
for i,H in Hs.iteritems():
    print 'hidden layer %d had mean %f and std %f' % (i+1, layer_means[i], layer_stds[i])

# plot the means and standard deviations
plt.figure()
plt.subplot(121)
plt.plot(Hs.keys(), layer_means, 'ob-')
plt.title('layer mean')
plt.subplot(122)
plt.plot(Hs.keys(), layer_stds, 'or-')
plt.title('layer std')

# plot the raw distributions
plt.figure()
for i,H in Hs.iteritems():
    plt.subplot(1,len(Hs),i+1)
    plt.hist(H.ravel(), 30, range=(-1,1))
```

```
input layer had mean 0.000927 and std 0.998388
hidden layer 1 had mean -0.000117 and std 0.213081
hidden layer 2 had mean -0.000001 and std 0.047551
hidden layer 3 had mean -0.000002 and std 0.010630
hidden layer 4 had mean 0.000001 and std 0.002378
hidden layer 5 had mean 0.000002 and std 0.000532
hidden layer 6 had mean -0.000000 and std 0.000119
hidden layer 7 had mean 0.000000 and std 0.000026
hidden layer 8 had mean -0.000000 and std 0.000006
hidden layer 9 had mean 0.000000 and std 0.000001
hidden layer 10 had mean -0.000000 and std 0.000000
```







All activations become zero!

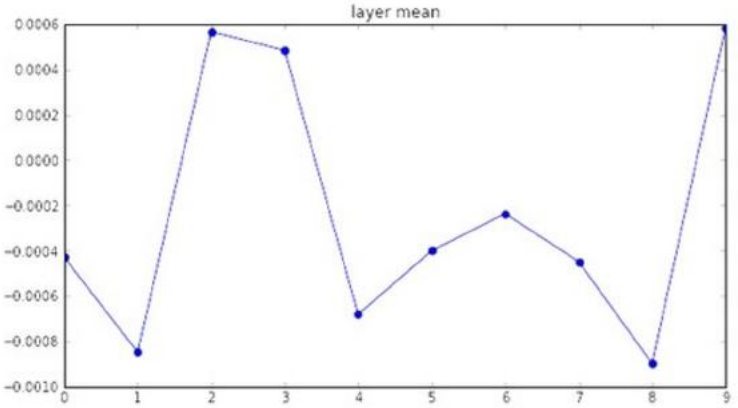
Q: think about the backward pass. What do the gradients look like?

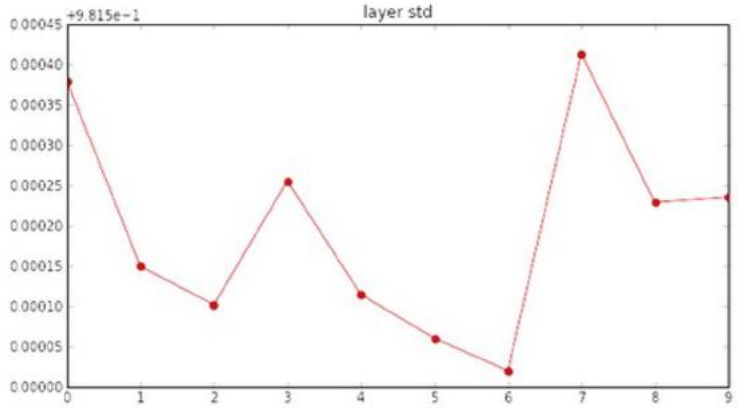
Hint: think about backward pass for a W*X gate.

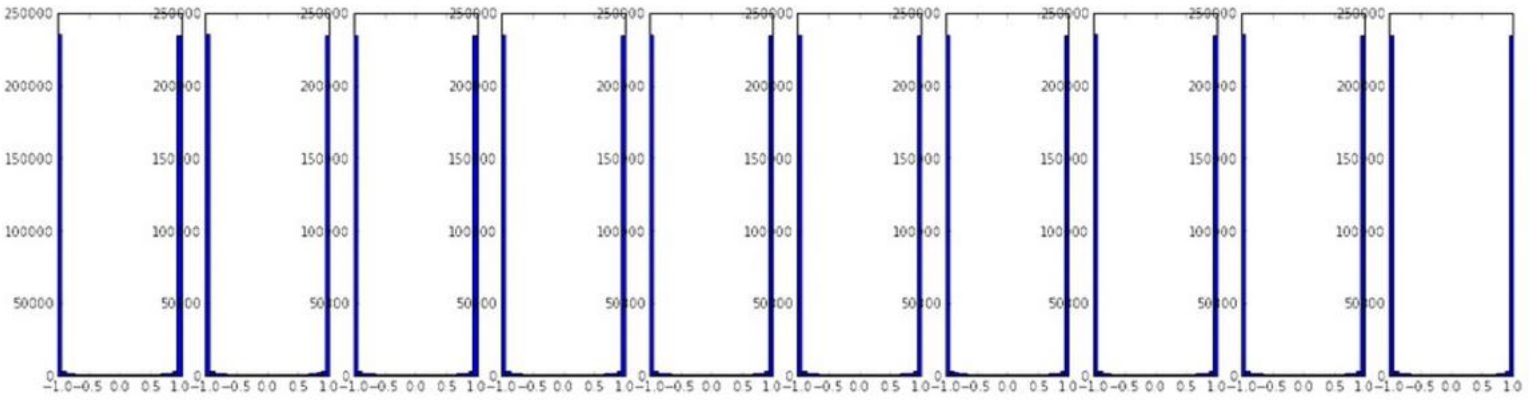
```
W = np.random.randn(fan_in, fan_out) * 1.0 # layer initialization
```

```
input layer had mean 0.001800 and std 1.001311
hidden layer 1 had mean -0.000430 and std 0.981879
hidden layer 2 had mean -0.000849 and std 0.981649
hidden layer 3 had mean 0.000566 and std 0.981601
hidden layer 4 had mean 0.000483 and std 0.981755
hidden layer 5 had mean -0.000682 and std 0.981614
hidden layer 6 had mean -0.000401 and std 0.981560
hidden layer 7 had mean -0.000237 and std 0.981520
hidden layer 8 had mean -0.000448 and std 0.981913
hidden layer 9 had mean -0.000899 and std 0.981728
hidden layer 10 had mean 0.000584 and std 0.981736
```

*1.0 instead of *0.01







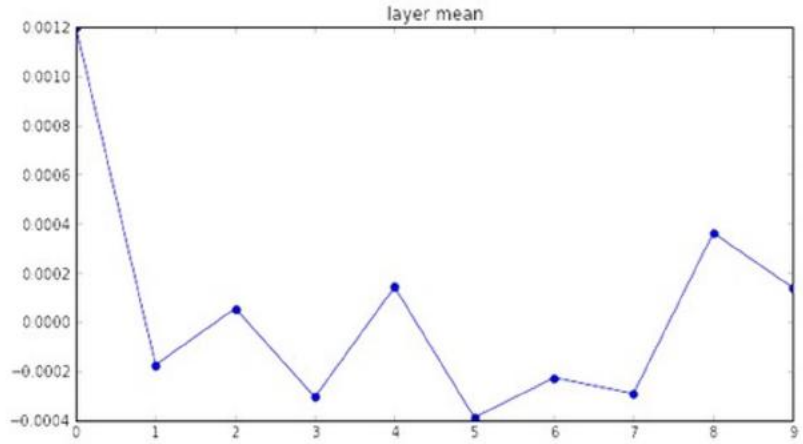
Almost all neurons completely saturated, either -1 and 1. Gradients will be all zero.

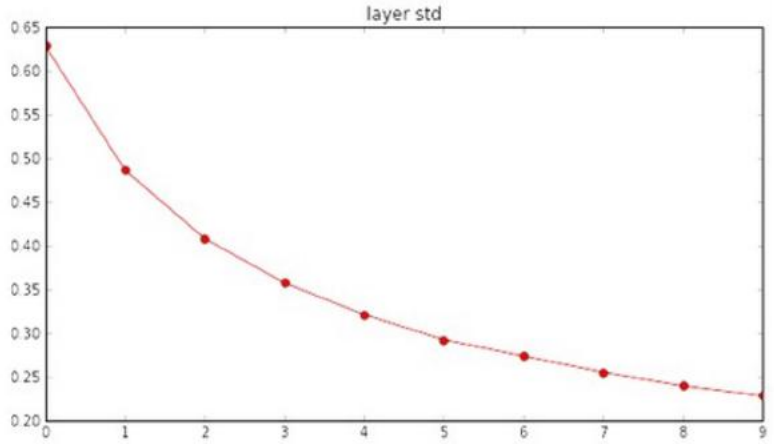
```
input layer had mean 0.001800 and std 1.001311
hidden layer 1 had mean 0.001198 and std 0.627953
hidden layer 2 had mean -0.000175 and std 0.486051
hidden layer 3 had mean 0.000055 and std 0.407723
hidden layer 4 had mean -0.000306 and std 0.357108
hidden layer 5 had mean 0.000142 and std 0.320917
hidden layer 6 had mean -0.000389 and std 0.292116
hidden layer 7 had mean -0.000228 and std 0.273387
hidden layer 8 had mean -0.000291 and std 0.254935
hidden layer 9 had mean 0.000361 and std 0.239266
hidden layer 10 had mean 0.000139 and std 0.228008
```

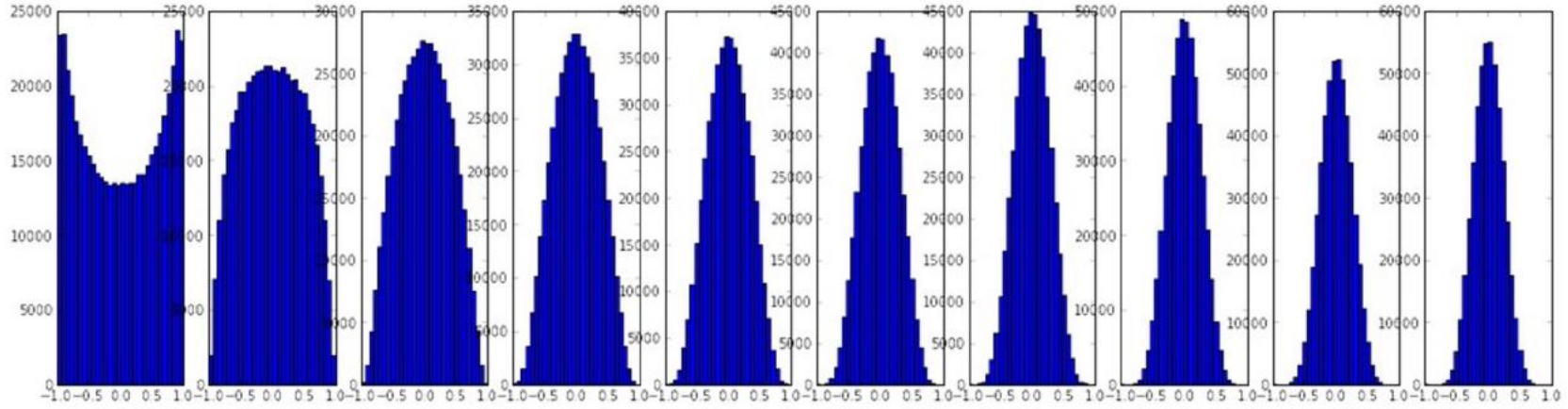
```
W = np.random.randn(fan_in, fan_out) / np.sqrt(fan_in) # layer initialization
```

Keep the variance the same across every layer!

"Xavier initialization" [Glorot et al., 2010]







Reasonable initialization. (Mathematical derivation assumes linear activations)

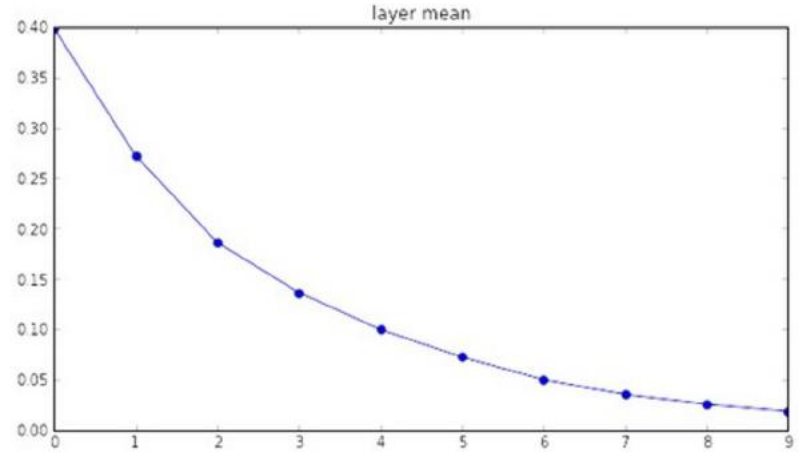
- If a hidden unit has a big fan-in, small changes on many of its incoming weights can cause the learning to overshoot.
  - We generally want smaller incoming weights when the fan-in is big, so initialize the weights to be proportional to sqrt(fan-in).
- We can also scale the learning rate the same way. More on this later!

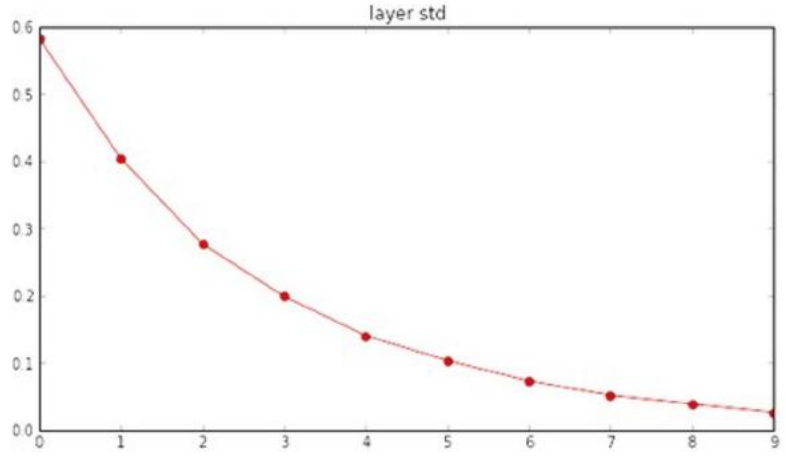
(from Hinton's notes)

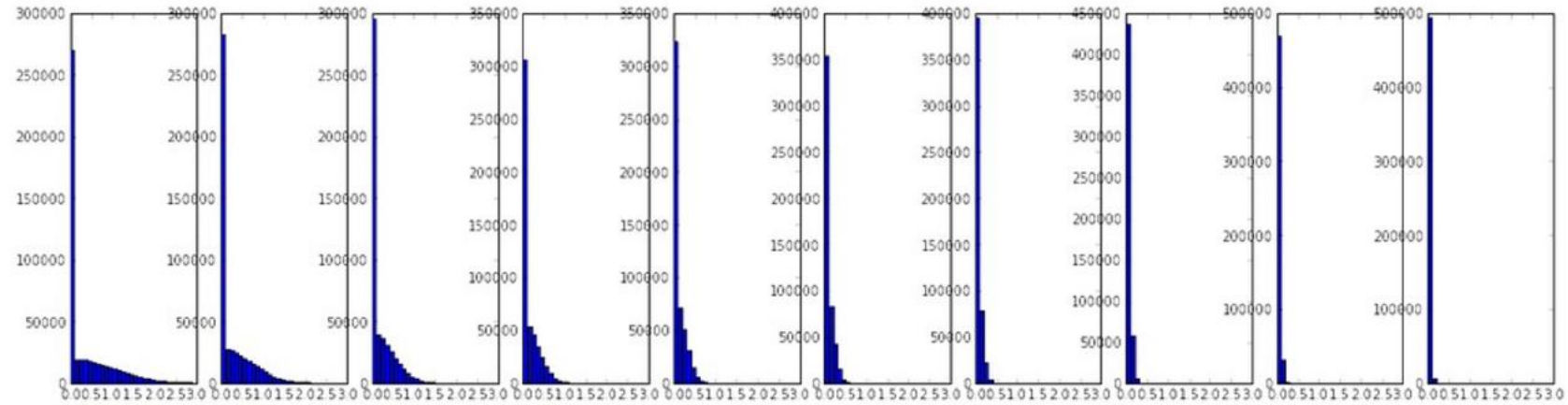
```
input layer had mean 0.000501 and std 0.999444
hidden layer 1 had mean 0.398623 and std 0.582273
hidden layer 2 had mean 0.272352 and std 0.403795
hidden layer 3 had mean 0.186076 and std 0.276912
hidden layer 4 had mean 0.136442 and std 0.198685
hidden layer 5 had mean 0.099568 and std 0.140299
hidden layer 6 had mean 0.072234 and std 0.103280
hidden layer 7 had mean 0.049775 and std 0.072748
hidden layer 8 had mean 0.035138 and std 0.051572
hidden layer 9 had mean 0.025404 and std 0.038583
hidden layer 10 had mean 0.018408 and std 0.026076
```

```
W = np.random.randn(fan_in, fan_out) / np.sqrt(fan_in) # layer initialization
```

but when using the ReLU nonlinearity it breaks.

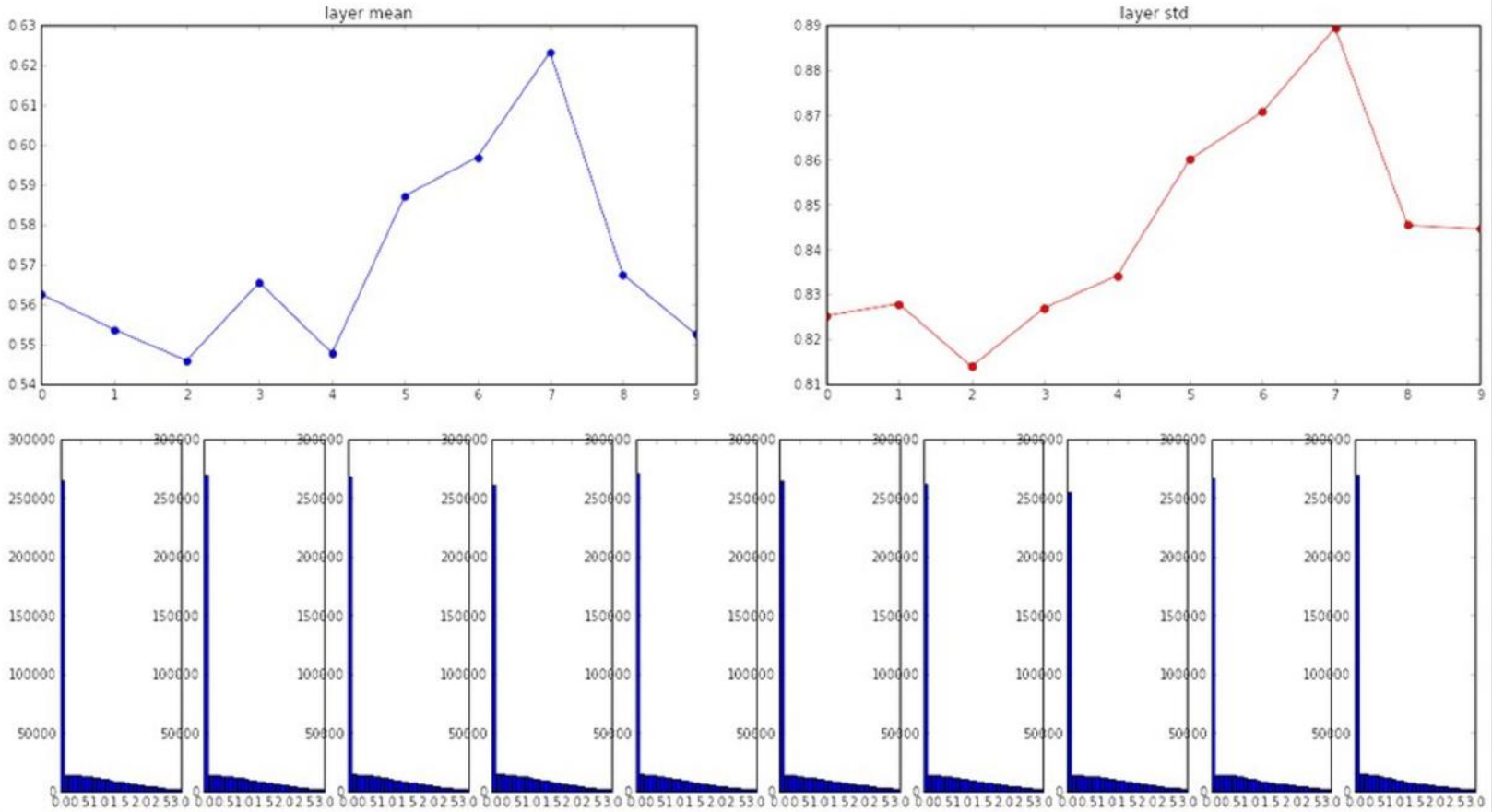




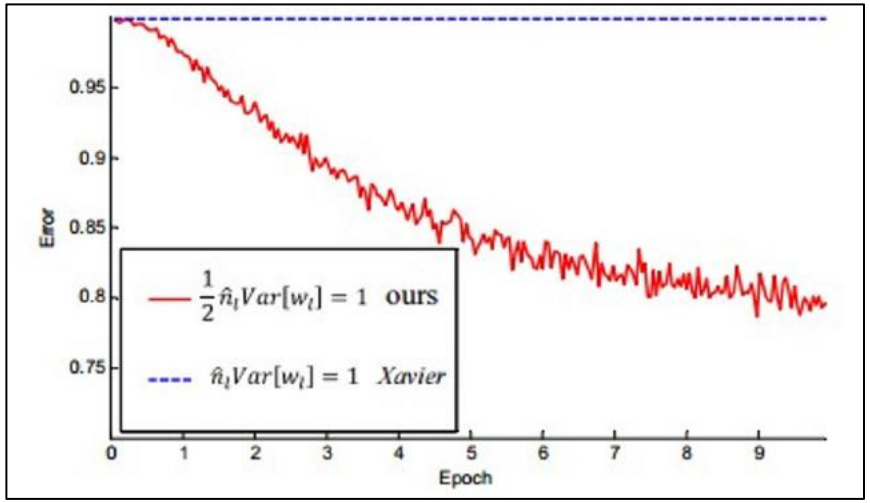


```
input layer had mean 0.000501 and std 0.999444
hidden layer 1 had mean 0.562488 and std 0.825232
hidden layer 2 had mean 0.553614 and std 0.827835
hidden layer 3 had mean 0.545867 and std 0.813855
hidden layer 4 had mean 0.565396 and std 0.826902
hidden layer 5 had mean 0.547678 and std 0.834092
hidden layer 6 had mean 0.587103 and std 0.860035
hidden layer 7 had mean 0.596867 and std 0.870610
hidden layer 8 had mean 0.623214 and std 0.889348
hidden layer 9 had mean 0.567498 and std 0.845357
hidden layer 10 had mean 0.552531 and std 0.844523
```

```
W = np.random.randn(fan_in, fan_out) / np.sqrt(fan_in/2) # layer initialization
```



He et al., 2015 (note additional /2)



Proper initialization is an active area of research…

- **Understanding the difficulty of training deep feedforward neural networks.** Glorot and Bengio, 2010
- **Exact solutions to the nonlinear dynamics of learning in deep linear neural networks.** Saxe et al, 2013
- **Random walk initialization for training very deep feedforward networks.** Sussillo and Abbott, 2014
- **Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification.** He et al., 2015
- **Data-dependent Initializations of Convolutional Neural Networks.** Krähenbühl et al., 2015
- **All you need is a good init.** Mishkin and Matas, 2015
- **How to start training: The effect of initialization and architecture.** Hanin and Rolnick, 2018
- **How to Initialize your Network? Robust Initialization for WeightNorm & ResNets.** Arpit et al., 2019

…

Batch Normalization

"you want unit Gaussian activations? just make them so."

consider a batch of activations at some layer.
To make each dimension unit gaussian, apply:

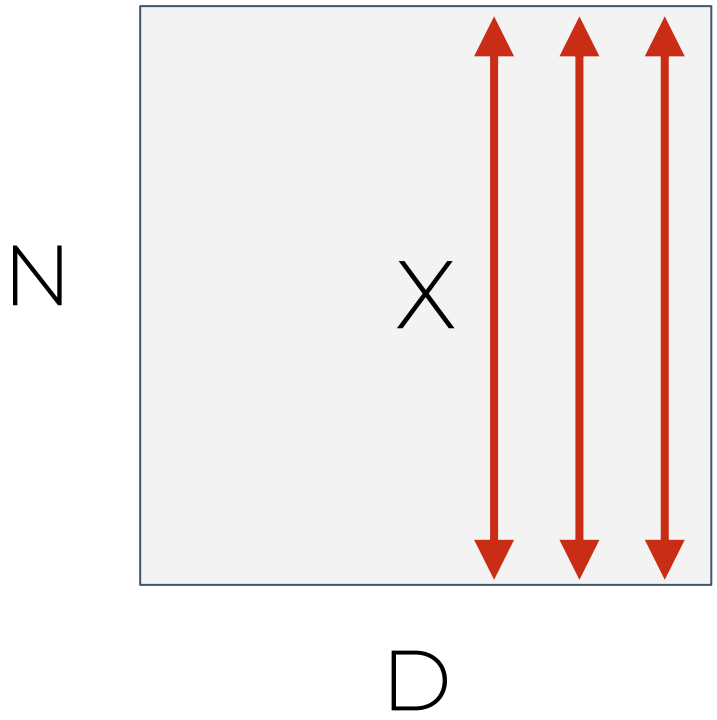
$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

this is a vanilla differentiable function...

[Ioffe and Szegedy, 2015]

Batch Normalization

"you want unit gaussian activations? just make them so."



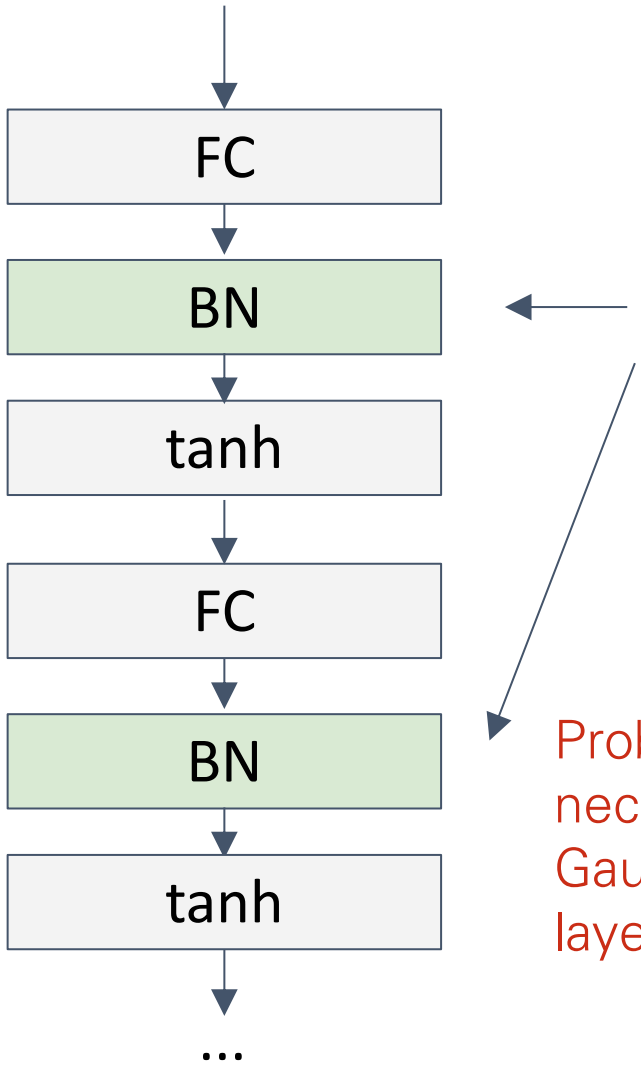
1. compute the empirical mean and variance independently for each dimension.

2. Normalize

$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

[Ioffe and Szegedy, 2015]

Batch Normalization



Usually inserted after Fully Connected / (or Convolutional, as we'll see soon) layers, and before nonlinearity.

Problem: do we necessarily want a unit Gaussian input to a tanh layer?

$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

[Ioffe and Szegedy, 2015]

Batch Normalization

Normalize:

$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

And then allow the network to squash the range if it wants to:

$$y^{(k)} = \gamma^{(k)}\widehat{x}^{(k)} + \beta^{(k)}$$

Note, the network can learn:

$$\gamma^{(k)} = \sqrt{\mathrm{Var}[x^{(k)}]}$$

$$\beta^{(k)} = \mathrm{E}[x^{(k)}]$$

to recover the identity mapping.

[Ioffe and Szegedy, 2015]

Batch Normalization

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

- Improves gradient flow through the network
- Allows higher learning rates
- Reduces the strong dependence on initialization
- Acts as a form of regularization in a funny way, and slightly reduces the need for dropout, maybe

[Ioffe and Szegedy, 2015]

Batch Normalization

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Note: at test time BatchNorm layer functions differently:

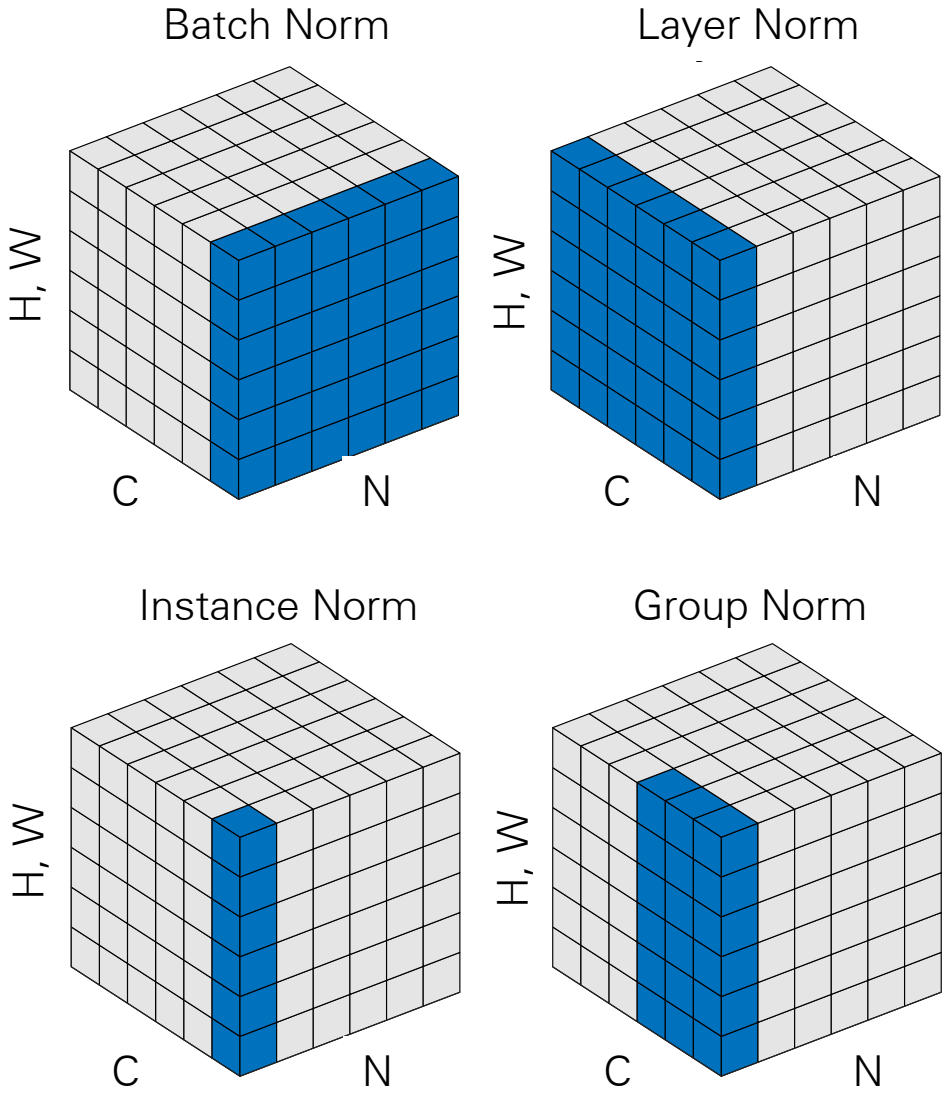
The mean/std are not computed based on the batch. Instead, a single fixed empirical mean of activations during training is used.

(e.g. can be estimated during training with running averages)

[Ioffe and Szegedy, 2015]

Other normalization schemes

- **Layer Normalization**
  Ba et al., Layer Normalization, arXiv preprint, 2016
- **Weight Normalization**
  Salimans, Weight Normalization: A Simple Reparameterization to Accelerate Training of Deep Neural Networks, NIPS, 2016
- **Instance Normalization**
  Ulyanov et al., Instance normalization: The missing ingredient for fast stylization. arXiv preprint, 2016
- **Batch Renormalization**
  Ioffe, Batch Renormalization: Towards Reducing Minibatch Dependence in Batch-Normalized Models, NIPS 2017
- **Group Renormalization**
  Wu and He, Group Normalization, ECCV 2018



Improving Generalization

Preventing Overfitting

- **Approach 1:** Get more data!
  - Almost always the best bet if you have enough compute power to train on more data.
- **Approach 2:** Use a model that has the right capacity:
  - enough to fit the true regularities.
  - not enough to also fit spurious regularities (if they are weaker).
- **Approach 3:** Average many different models.
  - Use models with different forms.
  - Or train the model on different subsets of the training data (this is called "bagging").
- **Approach 4: (Bayesian)** Use a single neural network architecture, but average the predictions made by many different weight vectors.

Some ways to limit the capacity of a neural net

- The capacity can be controlled in many ways:
  - **Architecture:** Limit the number of hidden layers and the number of units per layer.
  - **Early stopping:** Start with small weights and stop the learning before it overfits.
  - **Weight-decay:** Penalize large weights using penalties or constraints on their squared values (L2 penalty) or absolute values (L1 penalty).
  - **Noise:** Add noise to the weights or the activities.

- Typically, a combination of several of these methods is used.

Regularization

- Neural networks typically have thousands, if not millions of parameters
  - Usually, the dataset size smaller than the number of parameters
- Overfitting is a grave danger
- Proper weight regularization is crucial to avoid overfitting

$$\theta^* \leftarrow \arg\min_{\theta} \sum_{(x,y)\subseteq(X,Y)} \ell(y, a_L(x; \theta_{1,\ldots,L})) + \lambda\Omega(\theta)$$

- Possible regularization methods
  - l_2-regularization
  - l_1-regularization
  - Dropout

l_2-regularization

- Most important (or most popular) regularization

$$\theta^* \leftarrow \arg\min_{\theta} \sum_{(x,y)\subseteq(X,Y)} \ell(y, a_L(x; \theta_{1,\ldots,L})) + \frac{\lambda}{2} \sum_{l} \|\theta_l\|^2$$

- The l_2-regularization can pass inside the gradient descend update rule

$$\theta^{(t+1)} = \theta^{(t)} - \eta_t(\nabla_\theta \mathcal{L} + \lambda\theta_l) \Rightarrow$$

$$\theta^{(t+1)} = (1 - \lambda\eta_t)\theta^{(t)} - \eta_t \nabla_\theta \mathcal{L}$$

"Weight decay", because weights get smaller

- λ is usually about 10^{-1}, 10^{-2}

l_1-regularization

- l_1-regularization is one of the most important techniques

$$\theta^* \leftarrow \arg\min_{\theta} \sum_{(x,y)\subseteq(X,Y)} \ell(y, a_L(x; \theta_{1,\ldots,L})) + \frac{\lambda}{2} \sum_{l} \|\theta_l\|$$

- Also l_1-regularization passes inside the gradient descend update rule

$$\theta^{(t+1)} = \theta^{(t)} - \lambda\eta_t \frac{\theta^{(t)}}{|\theta^{(t)}|} - \eta_t \nabla_\theta \mathcal{L}$$

Sign function

- l_1-regularization $\rightarrow$ sparse weights
- $\lambda \uparrow \rightarrow$ more weights become 0

Data augmentation [Krizhevsky2012]

Original

Flip

Random crop

Contrast

Tint

Paper presentations start next week

Peri-LN: Revisiting Normalization Layer in the Transformer
Jeonghoon Kim, Byeongchan Lee, Cheonbok Park, Yeontaek Oh, Beomjun Kim, Taehwan Yoo, Seongjin Shin, Dongyoon Han, Jinwoo Shin, Kang Min Yoo. ICML'25

- Paper presentations will start next week!
- You'll start writing **paper critiques**, too.

Peri-LN: Revisiting Normalization Layer in the Transformer Architecture

Jeonghoon Kim[1 2] Byeongchan Lee[2] Cheonbok Park[1 2] Yeontaek Oh[1] Beomjun Kim[2] Taehwan Yoo[1] Seongjin Shin[1] Dongyoon Han[3] Jinwoo Shin[1 2] Kang Min Yoo[2 1]

Abstract

Selecting a layer normalization (LN) strategy that stabilizes training and speeds convergence in Transformers remains difficult, even for today's large language models (LLM). We present a comprehensive analytical foundation for understanding how different LN strategies influence training dynamics in large-scale Transformers. Until recently, Pre-LN and Post-LN have long dominated practices despite their limitations in large-scale training. However, several open-source models have recently begun silently adopting a third strategy without much explanation. This strategy places normalization layer *peripherally* around sublayers, a design we term *Peri-LN*. While Peri-LN has demonstrated promising performance, its precise mechanisms and benefits remain almost unexplored. Our in-depth analysis delineates the distinct behaviors of LN strategies, showing how each placement shapes activation variance and gradient propagation. To validate our theoretical insight, we conduct extensive experiments on Transformers up to 3.2B parameters, showing that Peri-LN consistently achieves more balanced variance growth, steadier gradient flow, and convergence stability. Our results suggest that Peri-LN warrants broader consideration for large-scale Transformer architectures, providing renewed insights into the optimal placement of LN.

1. Introduction

Building on a rapidly expanding lineage of Transformer-based large language models, open-source models have shown remarkable impact (Hoffmann et al., 2022; Guo et al., 2025; Yoo et al., 2024). As the demand for larger and more powerful models grows, various training stabilization techniques have been introduced (Yang et al., 2022; Zhai et al., 2023; Loshchilov et al., 2024). Among these, the choice of where and how to apply layer normalization (LN: LayerNorm or RMSNorm, Ba et al., 2016; Zhang & Sennrich, 2019) critically influences model convergence (Xiong et al., 2020; Kedia et al., 2024; Wortsman et al., 2024). However, their immense computational requirements have restricted deeper exploration of the underlying Transformer structure. Are we truly employing the optimal LN placement? In practice, fully revealing the results of massive resource investments can be challenging (Rivière et al., 2024). Despite its importance, there is still no consensus on a single best LN placement strategy.

Two prominent LN placements have been widely explored. Post-LN (Vaswani et al., 2017) normalizes the hidden state after adding the sub-layer output to the residual stream (that is, $\mathrm{Norm}(z + \mathrm{Module}(z))$ where z is input hidden state. Norms is LN). This helps constrain the variance of hidden states but may inadvertently weaken gradient signals, particularly in deeper models (Kedia et al., 2024). Pre-LN (Dubey et al., 2024), by contrast, normalizes before passing the hidden state to the sub-layer (that is, $z + \mathrm{Module}(\mathrm{Norm}(z))$). While this can enhance gradient propagation, it also admits so-called "massive activations," where hidden states grow exponentially across layers (Sun et al., 2024).

Previous studies on deep convolutional neural networks (CNNs) have analyzed the impact of batch normalization on variance changes during the initialization stage of ResNet architectures, demonstrating its relationship to model performance (De & Smith, 2020). They noted that, in models without normalization, hidden activation growth at initialization can be exponential, leading to poor performance and stability. In contrast, in pre-normalized CNNs, the variance of hidden activations was shown to increase linearly as model depth grows. In the same vein, Kedia et al. (2024) reported that, for Transformer architectures as well, the variance in the forward propagation of Transformer-based language models *at initialization* increases linearly with depth. However, in the context of Transformer architectures, we observed that this variance growth at initialization does

[1]Equal contribution. [1]NAVER Cloud [2]Korea Advanced Institute of Science and Technology (KAIST) [3]NAVER AI Lab. Correspondence to: Jeonghoon Kim <jeonghoon.samuel@gmail.com>, Jinwoo Shin <jinwoos@kaist.ac.kr>, Kang Min Yoo <kangmin.yoo@navercorp.com>.

Proceedings of the 42nd International Conference on Machine Learning, Vancouver, Canada. PMLR 267, 2025.

Paper Reviews

Think deeply about the papers we read and try to learn from them as much as possible (and then even more). If you do not understand something, we should discuss it and dissect it together. Whatever you think others understand, they understand less (the instructor included), but together we will get it.

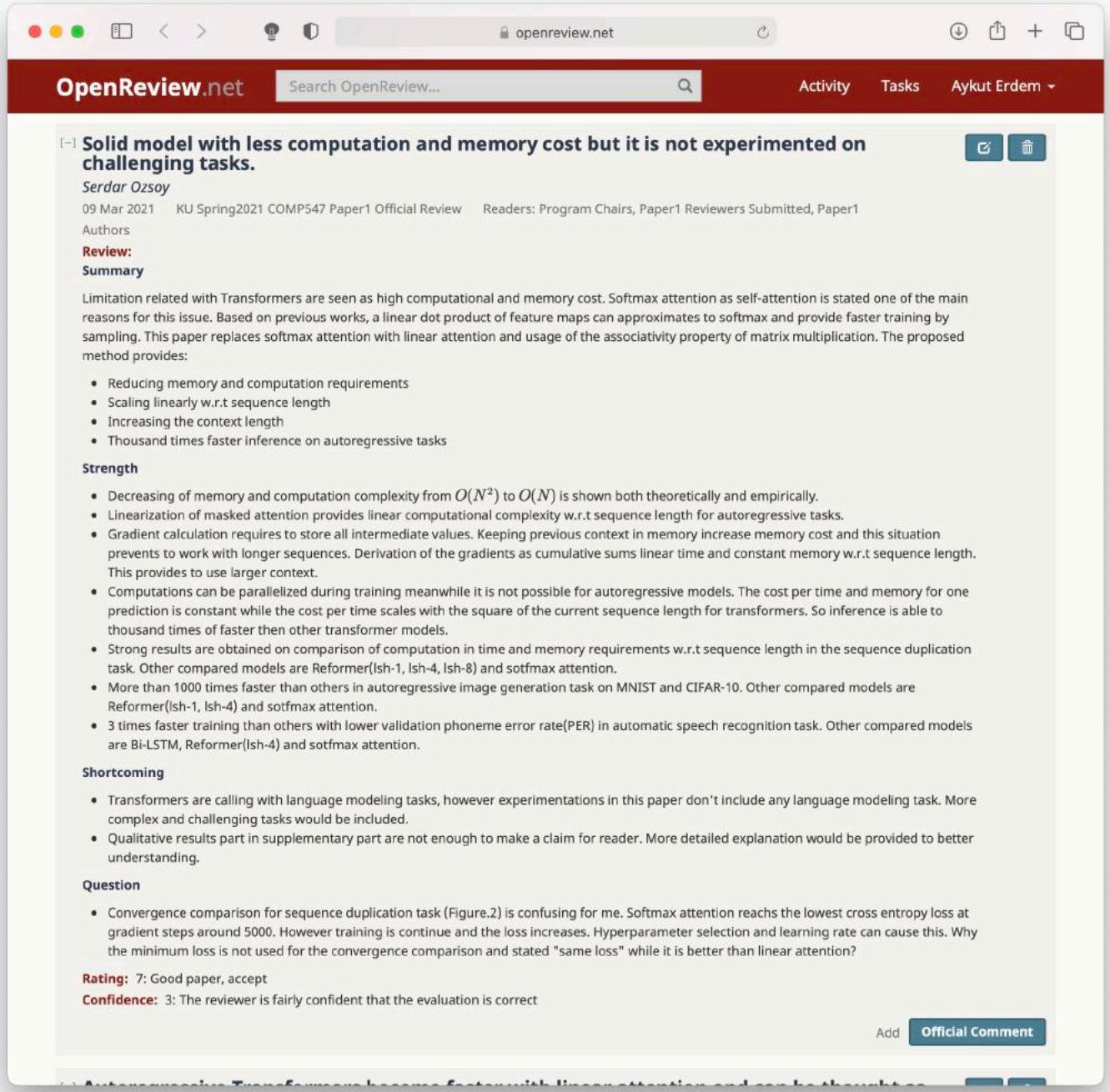
- Identify the key questions the paper studies, and the answers it provides to these questions.
- Consider the challenges of the problem or scenario studied, and how the paper's approach addresses them.
- Deconstruct the formal and technical parts to understand their fine details. Note to yourself aspects that are not clear to you

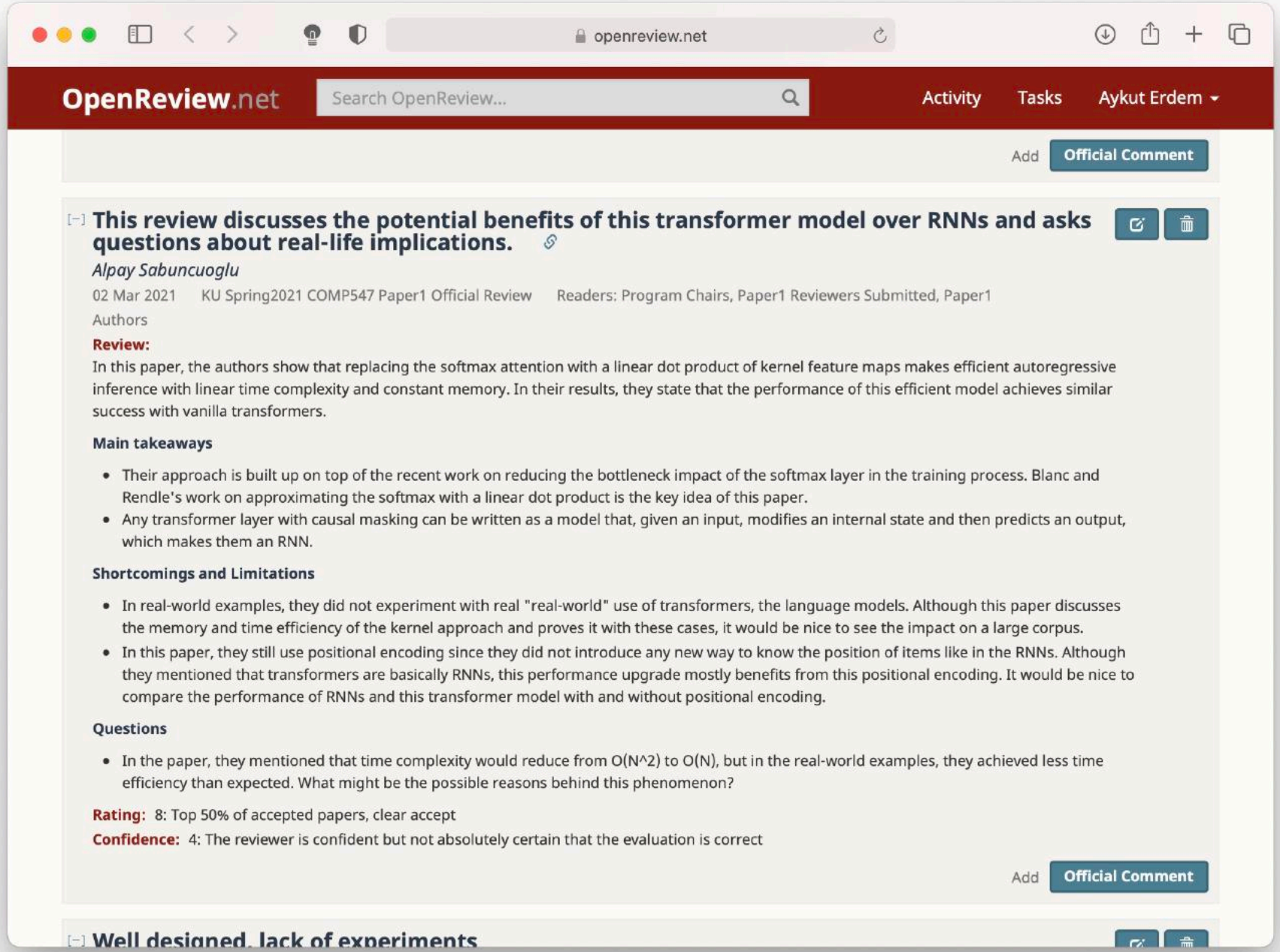
Borrowed from Yoav Artzi

Paper Reviewing Guidelines

- When reviewing the paper, start with 1–2 sentences summarizing what the paper is about.
- Continue with the strength of the paper. Outline its contribution, and your main takeaways. What did you learn?
- Highlight shortcomings and limitations. Please focus on weaknesses that fundamental to the method. Unlike conference or journal reviewing, this part is intended for your understanding and discussion.
- Try to suggest ways to address the paper's limitations. Any idea is welcome and will contribute to the discussion.
- Suggest questions for discussion in class. As part of the discussion in class, you are asked to raise these questions during the class.

Borrowed from Yoav Artzi


openreview.net
OpenReview.net
Search OpenReview...
Activity
Tasks
Aykut Erdem
Solid model with less computation and memory cost but it is not experimented on challenging tasks.
Serdar Ozsoy
09 Mar 2021 KU Spring2021 COMP547 Paper1 Official Review Readers: Program Chairs, Paper1 Reviewers Submitted, Paper1 Authors
Review:
Summary
Limitation related with Transformers are seen as high computational and memory cost. Softmax attention as self-attention is stated one of the main reasons for this issue. Based on previous works, a linear dot product of feature maps can approximates to softmax and provide faster training by sampling. This paper replaces softmax attention with linear attention and usage of the associativity property of matrix multiplication. The proposed method provides:
Reducing memory and computation requirements
Scaling linearly w.r.t sequence length
Increasing the context length
Thousand times faster inference on autoregressive tasks
Strength
Decreasing of memory and computation complexity from $O(N^2)$ to $O(N)$ is shown both theoretically and empirically.
Linearization of masked attention provides linear computational complexity w.r.t sequence length for autoregressive tasks.
Gradient calculation requires to store all intermediate values. Keeping previous context in memory increase memory cost and this situation prevents to work with longer sequences. Derivation of the gradients as cumulative sums linear time and constant memory w.r.t sequence length. This provides to use larger context.
Computations can be parallelized during training meanwhile it is not possible for autoregressive models. The cost per time and memory for one prediction is constant while the cost per time scales with the square of the current sequence length for transformers. So inference is able to thousand times of faster then other transformer models.
Strong results are obtained on comparison of computation in time and memory requirements w.r.t sequence length in the sequence duplication task. Other compared models are Reformer(lsh-1, lsh-4, lsh-8) and sotfmax attention.
More than 1000 times faster than others in autoregressive image generation task on MNIST and CIFAR-10. Other compared models are Reformer(lsh-1, lsh-4) and sotfmax attention.
3 times faster training than others with lower validation phoneme error rate(PER) in automatic speech recognition task. Other compared models are Bi-LSTM, Reformer(lsh-4) and sotfmax attention.
Shortcoming
Transformers are calling with language modeling tasks, however experimentations in this paper don't include any language modeling task. More complex and challenging tasks would be included.
Qualitative results part in supplementary part are not enough to make a claim for reader. More detailed explanation would be provided to better understanding.
Question
Convergence comparison for sequence duplication task (Figure.2) is confusing for me. Softmax attention reachs the lowest cross entropy loss at gradient steps around 5000. However training is continue and the loss increases. Hyperparameter selection and learning rate can cause this. Why the minimum loss is not used for the convergence comparison and stated "same loss" while it is better than linear attention?
Rating: 7: Good paper, accept
Confidence: 3: The reviewer is fairly confident that the evaluation is correct
Add
Official Comment


openreview.net
OpenReview.net
Search OpenReview...
Activity
Tasks
Aykut Erdem
Add
Official Comment
This review discusses the potential benefits of this transformer model over RNNs and asks questions about real-life implications.
Alpay Sabuncuoglu
02 Mar 2021
KU Spring2021 COMP547 Paper1 Official Review
Readers: Program Chairs, Paper1 Reviewers Submitted, Paper1 Authors
Review:
In this paper, the authors show that replacing the softmax attention with a linear dot product of kernel feature maps makes efficient autoregressive inference with linear time complexity and constant memory. In their results, they state that the performance of this efficient model achieves similar success with vanilla transformers.
Main takeaways
Their approach is built up on top of the recent work on reducing the bottleneck impact of the softmax layer in the training process. Blanc and Rendle's work on approximating the softmax with a linear dot product is the key idea of this paper.
Any transformer layer with causal masking can be written as a model that, given an input, modifies an internal state and then predicts an output, which makes them an RNN.
Shortcomings and Limitations
In real-world examples, they did not experiment with real "real-world" use of transformers, the language models. Although this paper discusses the memory and time efficiency of the kernel approach and proves it with these cases, it would be nice to see the impact on a large corpus.
In this paper, they still use positional encoding since they did not introduce any new way to know the position of items like in the RNNs. Although they mentioned that transformers are basically RNNs, this performance upgrade mostly benefits from this positional encoding. It would be nice to compare the performance of RNNs and this transformer model with and without positional encoding.
Questions
In the paper, they mentioned that time complexity would reduce from O(N^2) to O(N), but in the real-world examples, they achieved less time efficiency than expected. What might be the possible reasons behind this phenomenon?
Rating: 8: Top 50% of accepted papers, clear accept
Confidence: 4: The reviewer is confident but not absolutely certain that the evaluation is correct
Add
Official Comment
Well designed, lack of experiments

openreview.net

OpenReview.net | Search OpenReview... | Activity | Tasks | Aykut Erdem

Add **Official Comment**

Improves the time and space complexity of Transformer to O(N) complexity and Derives an Autoregressive version of Transformer

Samet Demir

09 Mar 2021 KU Spring2021 COMP547 Paper1 Official Review Readers: Program Chairs, Paper1 Reviewers Submitted, Paper1 Authors

Review:

Summary:

There are two contributions of this paper. First, the authors propose a kernel-based formulation of self-attention that reduces time and space complexity of Transformer to $\mathcal{O}(N)$ complexity with the help of the associative property of matrix products. They call it Linear Transformer. Second, they derive an autoregressive version of Transformer using Causal Masking. This helps them show a relation between Recurrent Neural Networks and Transformer.

They experimentally show the performance of Linear Transformer in comparison to Standard Transformer ($O(N^2)$ complexity) [Vaswani et al., 2017] and Reformer ($O(N \log N)$ complexity) [Kitaev et al., 2020]. They use Image Generation (MNIST and CIFAR) and Automatic Speech Recognition (WSJ dataset [Paul & Baker, 1992]) tasks in addition to simple Synthetic Tasks. In these experiments, authors show that Linear Transformer is computationally more efficient while its performance in term of accomplishing the given tasks is similar to the other Transformer variants. Also, in Automatic Speech Recognition, Linear Transformer slightly outperforms the LSTM while being slightly faster to train and evaluate.

Pros:

1. Complexity improvement for Transformer is a significant contribution
2. The shown relation between Recurrent Neural Networks and Transformer is interesting and also makes Transformer a option for autoregressive tasks.
3. Well-written, easy to read paper
4. Mentions related works comprehensively
5. Simple but elegant ideas

Cons:

1. Concurrent work by Shen et al. (2020) explored the use of linearized attention
2. Experiments should be diversified. I think MNIST & CIFAR are not good enough for this purpose. Transformer-variants are applied to many tasks starting from NLP tasks. I was expecting to see at least one text based NLP task such as translation. They use Automatic Speech Recognition task but it is mainly for the evaluation of autoregressive abilities of Transformer.

Rating: 8: Top 50% of accepted papers, clear accept

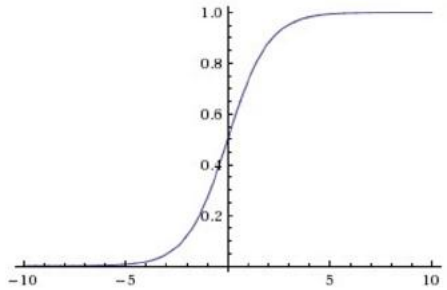
Confidence: 5: The reviewer is absolutely certain that the evaluation is correct and very familiar with the relevant literature

Add **Official Comment**

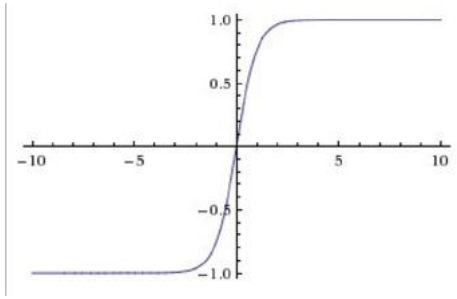
Recap: Activation Functions

Sigmoid

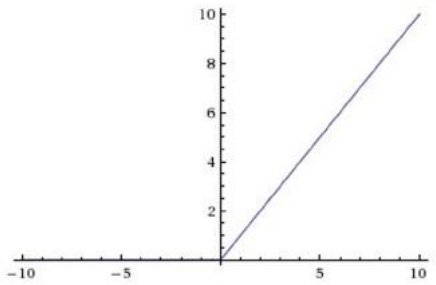
$$\sigma(x) = 1/(1 + e^{-x})$$



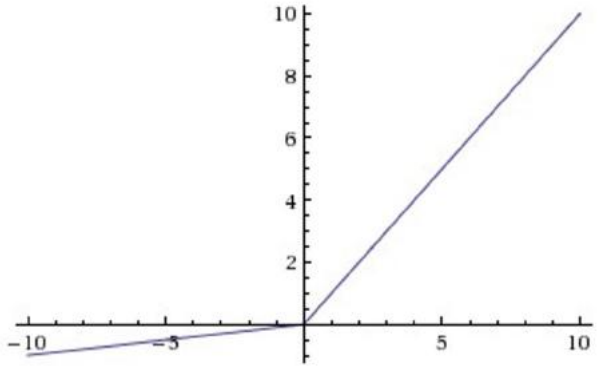
tanh tanh(x)



ReLU max(0,x)



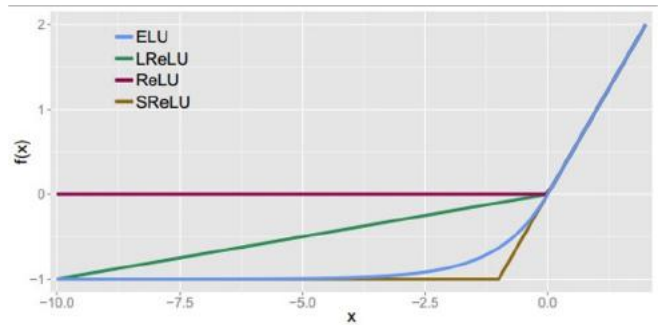
Leaky ReLU
max(0.1x, x)



Maxout $\max(w_1^T x + b_1, w_2^T x + b_2)$

ELU

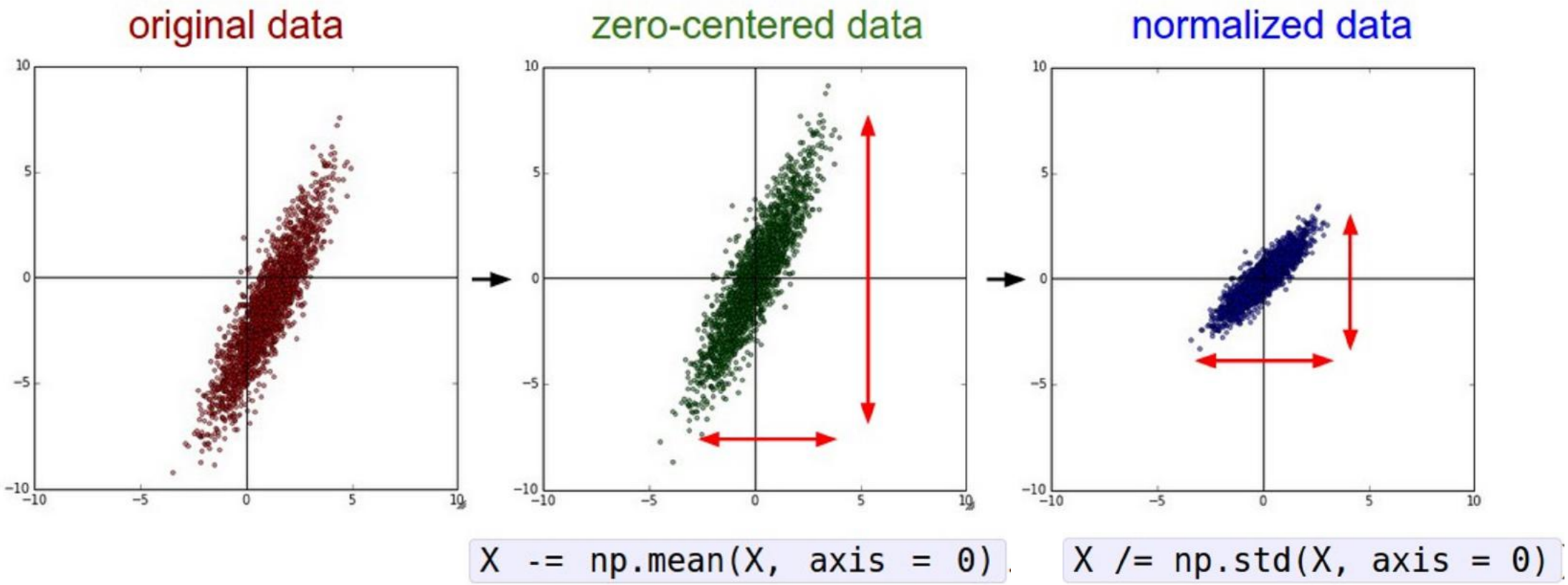
$$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha\,(\exp(x) - 1) & \text{if } x \leq 0 \end{cases}$$



Recap: Data preprocessing

- Input variables should be as decorrelated as possible
  - Input variables are “more independent”
  - Network is forced to find non-trivial correlations between inputs
  - Decorrelated inputs → Better optimization
  - Obviously not the case when inputs are by definition correlated (sequences)

Recap: Data preprocessing



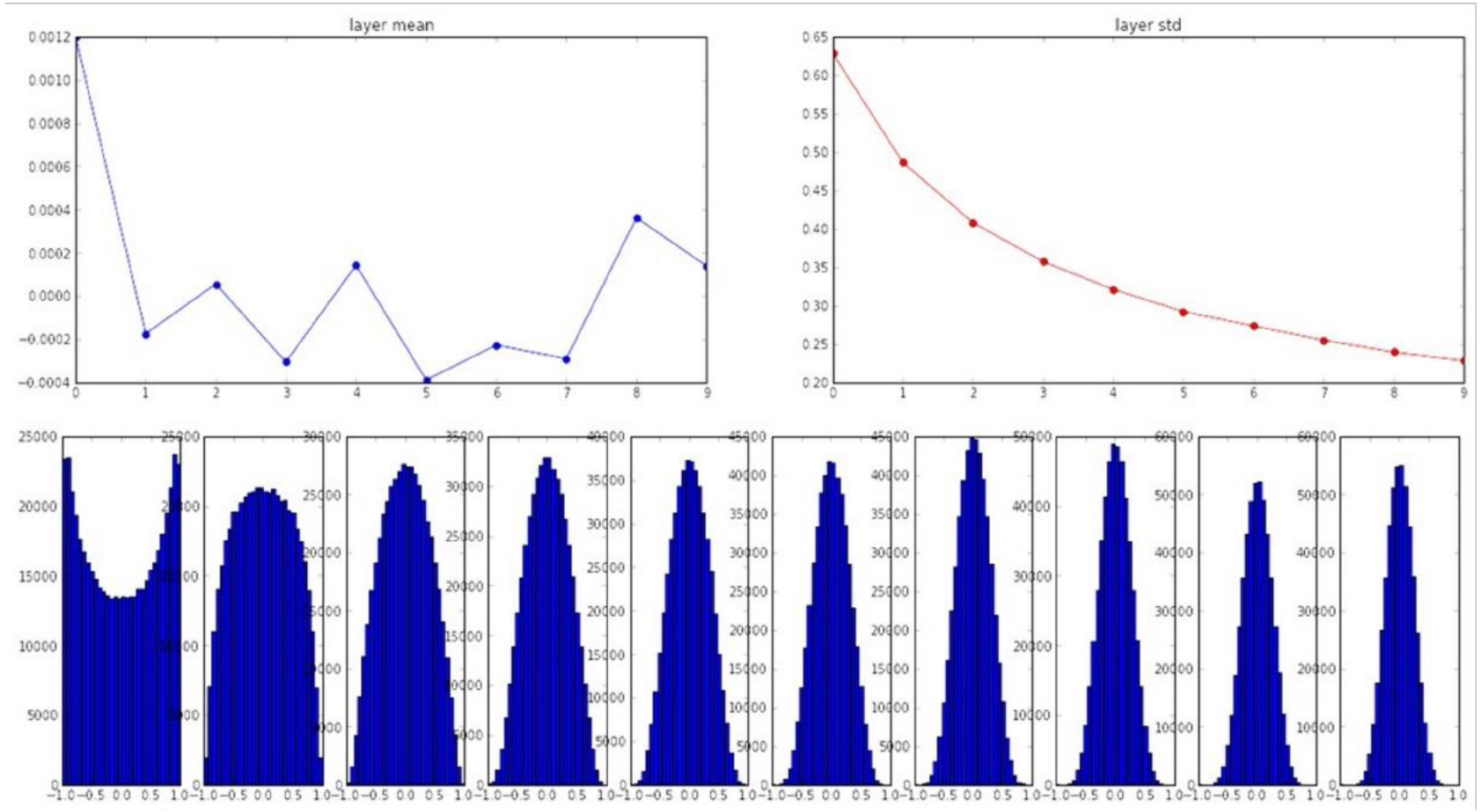
(Assume X [NxD] is data matrix, each example in a row)

Recap: Initialization

```
W = np.random.randn(fan_in, fan_out) / np.sqrt(fan_in) # layer initialization
```

Keep the variance the same across every layer!

"Xavier initialization" [Glorot et al., 2010]



Reasonable initialization. (Mathematical derivation assumes linear activations)

- If a hidden unit has a big fan-in, small changes on many of its incoming weights can cause the learning to overshoot.
  - We generally want smaller incoming weights when the fan-in is big, so initialize the weights to be proportional to sqrt(fan-in).
- We can also scale the learning rate the same way. More on this later!

(from Hinton's notes)

Recap: Batch Normalization

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

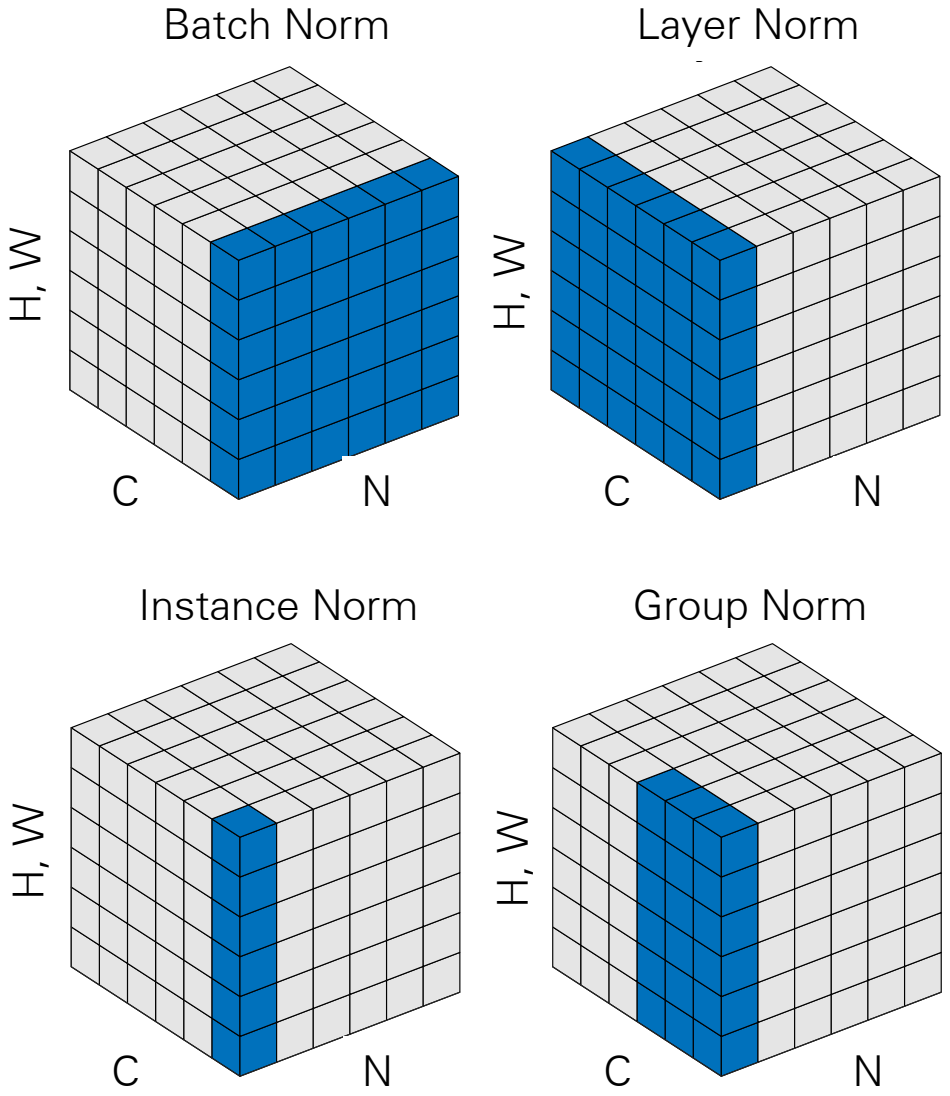
$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

- Improves gradient flow through the network
- Allows higher learning rates
- Reduces the strong dependence on initialization
- Acts as a form of regularization in a funny way, and slightly reduces the need for dropout, maybe

[Ioffe and Szegedy, 2015]

Recap: Other normalization schemes

- **Layer Normalization**
  Ba et al., Layer Normalization, arXiv preprint, 2016
- **Weight Normalization**
  Salimans, Weight Normalization: A Simple Reparameterization to Accelerate Training of Deep Neural Networks, NIPS, 2016
- **Instance Normalization**
  Ulyanov et al., Instance normalization: The missing ingredient for fast stylization. arXiv preprint, 2016
- **Batch Renormalization**
  Ioffe, Batch Renormalization: Towards Reducing Minibatch Dependence in Batch-Normalized Models, NIPS 2017
- **Group Renormalization**
  Wu and He, Group Normalization, ECCV 2018



Recap: Preventing Overfitting

- **Approach 1:** Get more data!
  - Almost always the best bet if you have enough compute power to train on more data.
- **Approach 2:** Use a model that has the right capacity:
  - enough to fit the true regularities.
  - not enough to also fit spurious regularities (if they are weaker).
- **Approach 3:** Average many different models.
  - Use models with different forms.
  - Or train the model on different subsets of the training data (this is called "bagging").
- **Approach 4: (Bayesian)** Use a single neural network architecture, but average the predictions made by many different weight vectors.

Recap: Regularization

- Neural networks typically have thousands, if not millions of parameters
  - Usually, the dataset size smaller than the number of parameters
- Overfitting is a grave danger
- Proper weight regularization is crucial to avoid overfitting

$$\theta^* \leftarrow \arg\min_{\theta} \sum_{(x,y)\subseteq(X,Y)} \ell(y, a_L(x; \theta_{1,\ldots,L})) + \lambda\Omega(\theta)$$

- Possible regularization methods
  - l_2-regularization
  - l_1-regularization
  - Dropout

Recap: Data augmentation [Krizhevsky'12]

Original

Flip

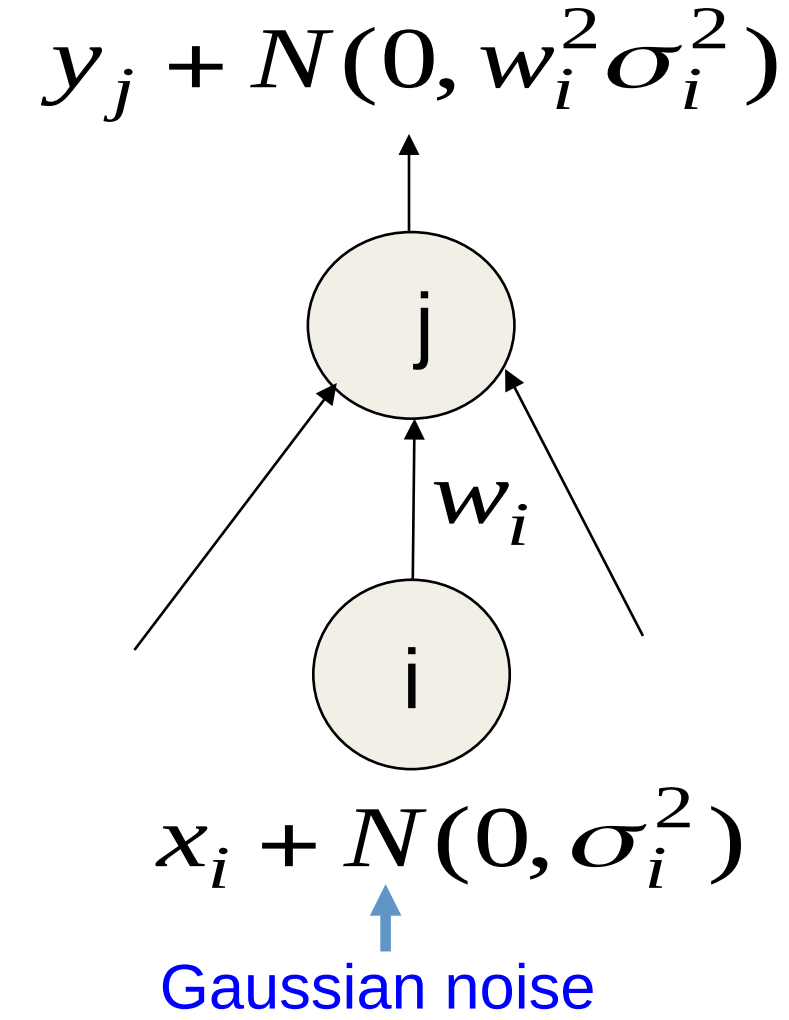
Random crop

Contrast

Tint

Noise as a regularizer

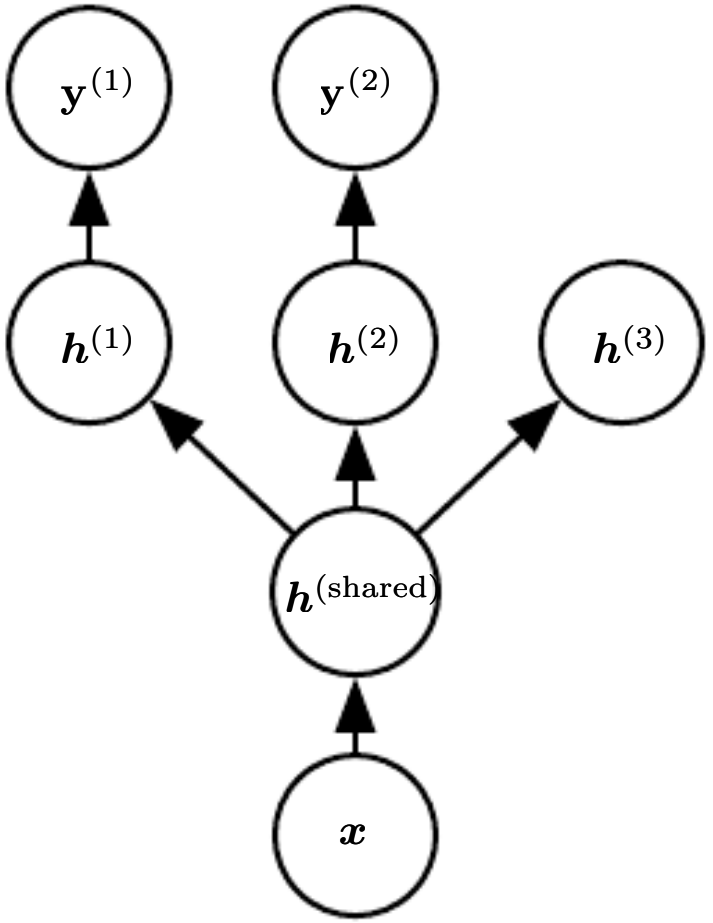
- Suppose we add Gaussian noise to the inputs.
  - The variance of the noise is amplified by the squared weight before going into the next layer.
- In a simple net with a linear output unit directly connected to the inputs, the amplified noise gets added to the output.
- This makes an additive contribution to the squared error.
  - So minimizing the squared error tends to minimize the squared weights when the inputs are noisy.



Not exactly equivalent to using an L2 weight penalty.

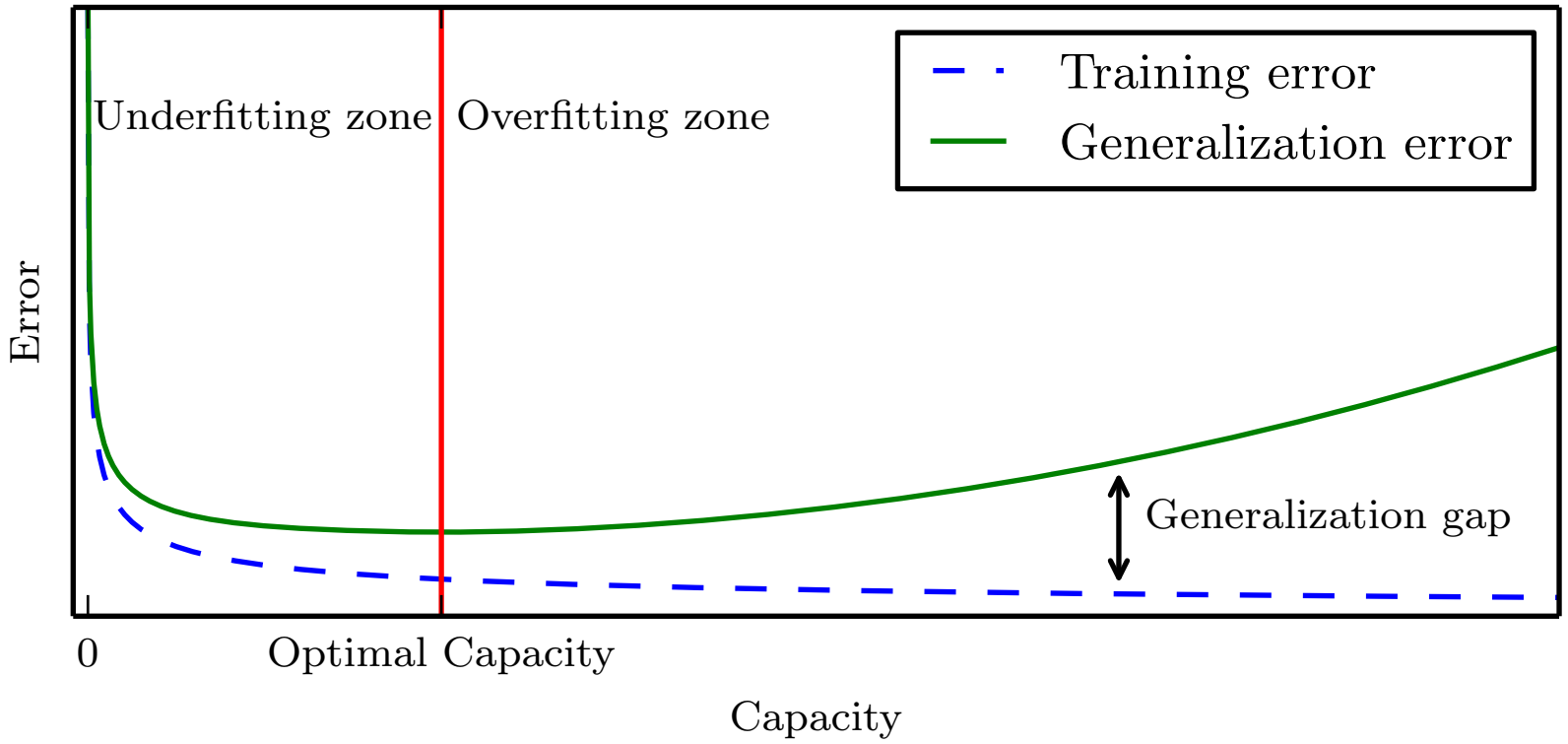
Multi-task Learning

- Improving generalization by pooling the examples arising out of several tasks.
- Different supervised tasks share the same input x, as well as some intermediate-level representation h(shared)
  - Task-specific parameters
  - Generic parameters (shared across all the tasks)



Early stopping

- Start with small weights and stop the learning before it overfits.
- Think early stopping as a very efficient hyperparameter selection.
  - The number of training steps is just another hyperparameter.



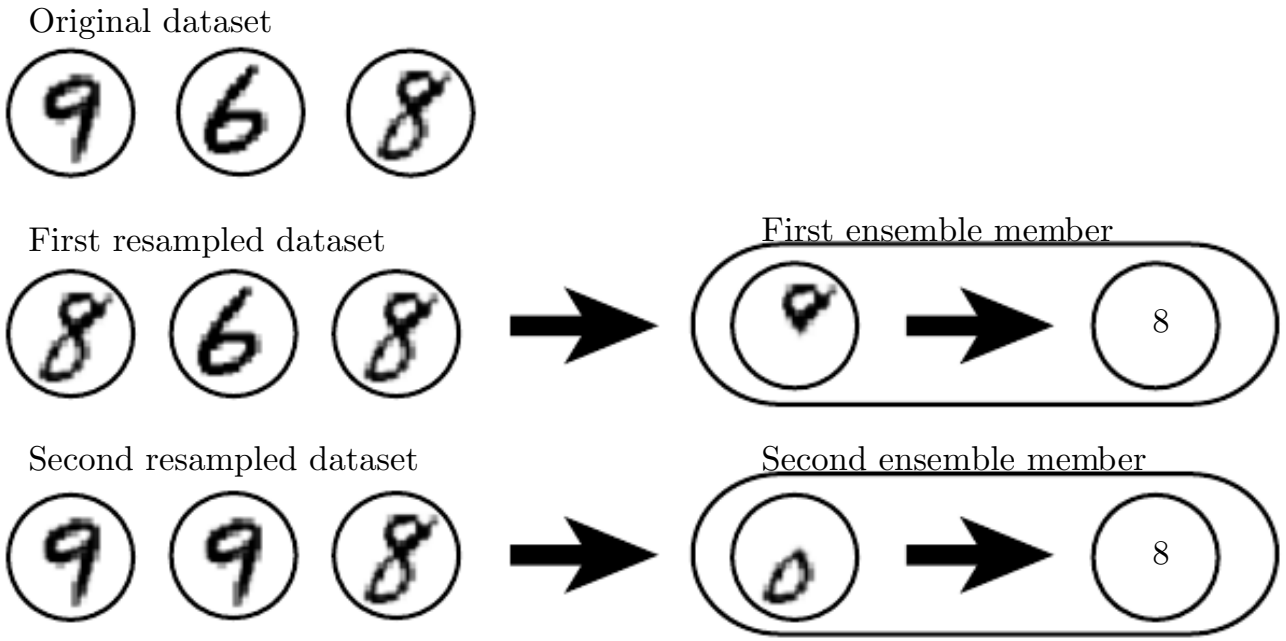
Model Ensembles: The bias-variance trade-off

- When the amount of training data is limited, we get overfitting.
  - Averaging the predictions of many different models is a good way to reduce overfitting.
  - It helps most when the models make very different predictions.
- For regression, the squared error can be decomposed into a "bias" term and a "variance" term.
  - The bias term is big if the model has too little capacity to fit the data.
  - The variance term is big if the model has so much capacity that it is good at fitting the sampling error in each particular training set.
- By averaging away the variance we can use individual models with high capacity. These models have high variance but low bias.

Model Ensembles

- Train several different models separately, then have all of the models vote on the output for test examples.
- Different models will usually not make all the same errors on the test set.

- Usually ~2% gain!



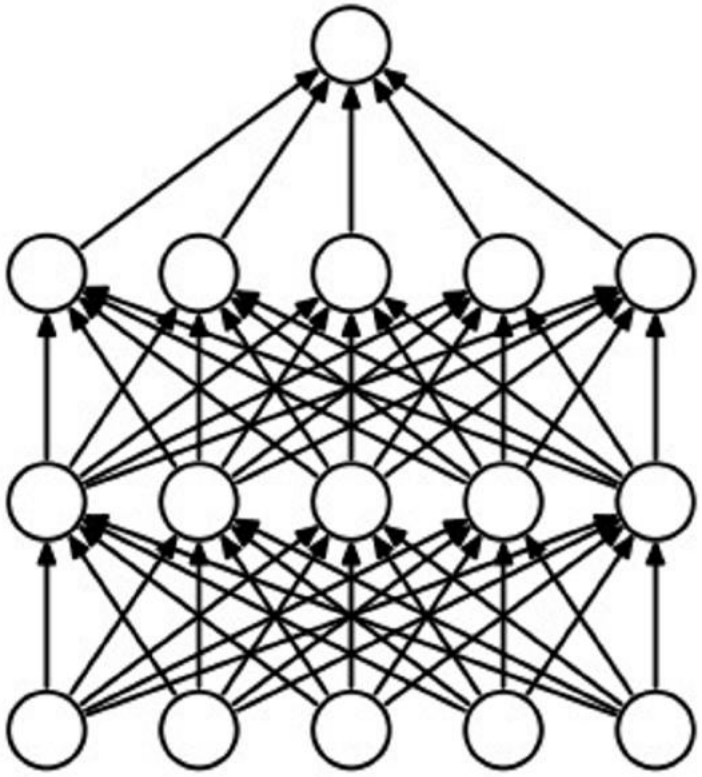
Model Ensembles

- We can also get a small boost from averaging multiple model checkpoints of a single model.
- keep track of (and use at test time) a running average parameter vector:

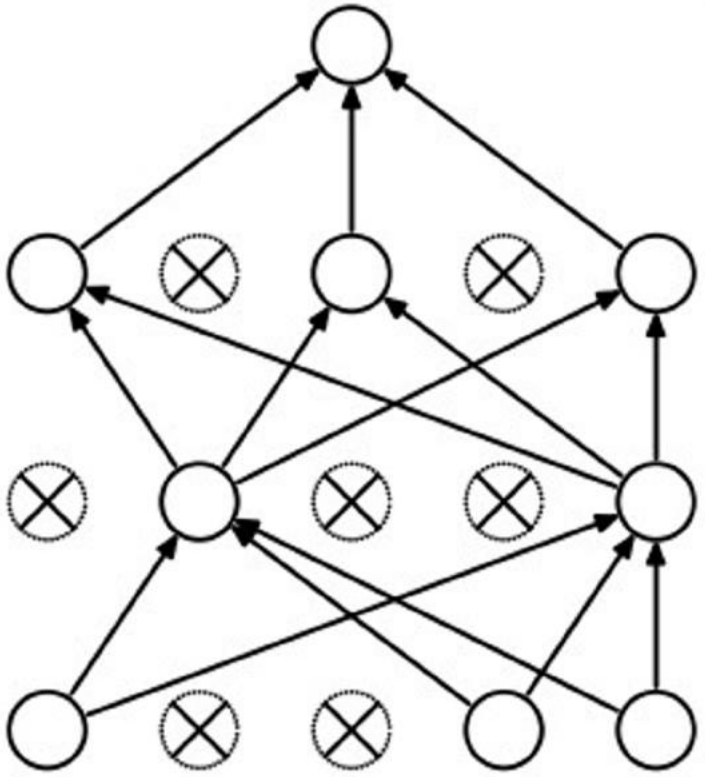
```
while True:
  data_batch = dataset.sample_data_batch()
  loss = network.forward(data_batch)
  dx = network.backward()
  x += - learning_rate * dx
  x_test = 0.995*x_test + 0.005*x # use for test set
```

Dropout

- "randomly set some neurons to zero in the forward pass"



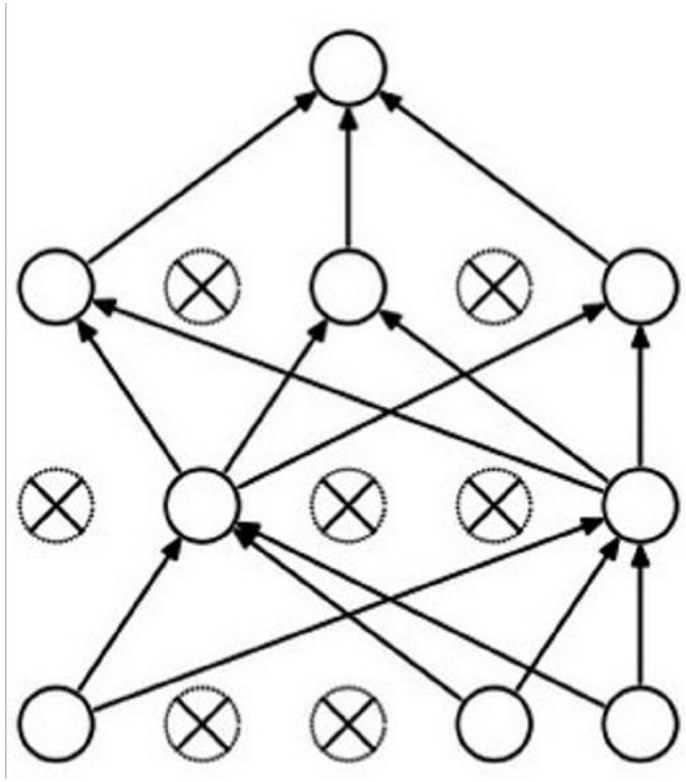
(a) Standard Neural Net



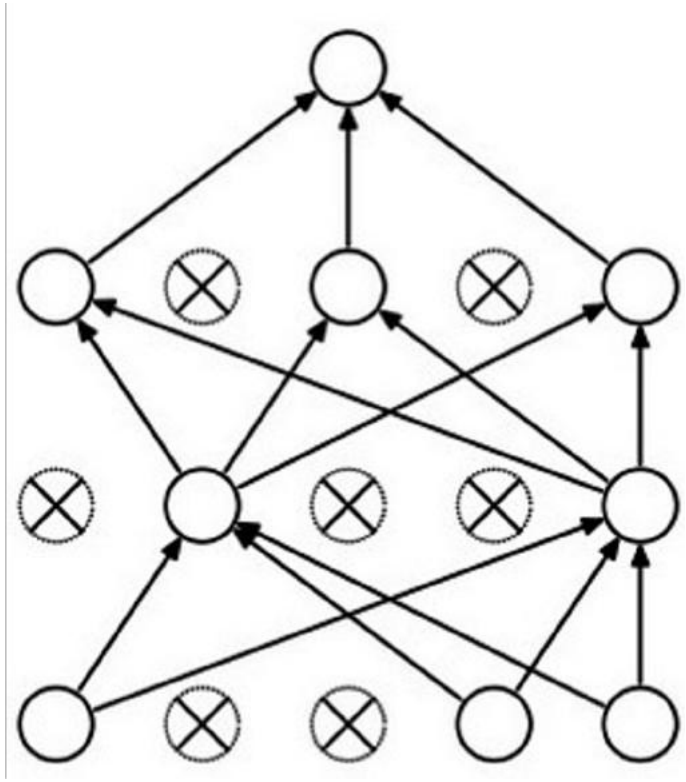
(b) After applying dropout.

[Srivastava et al., 2014]

Waaaait a second...
How could this possibly be a good idea?



Waaaait a second...
How could this possibly be a good idea?



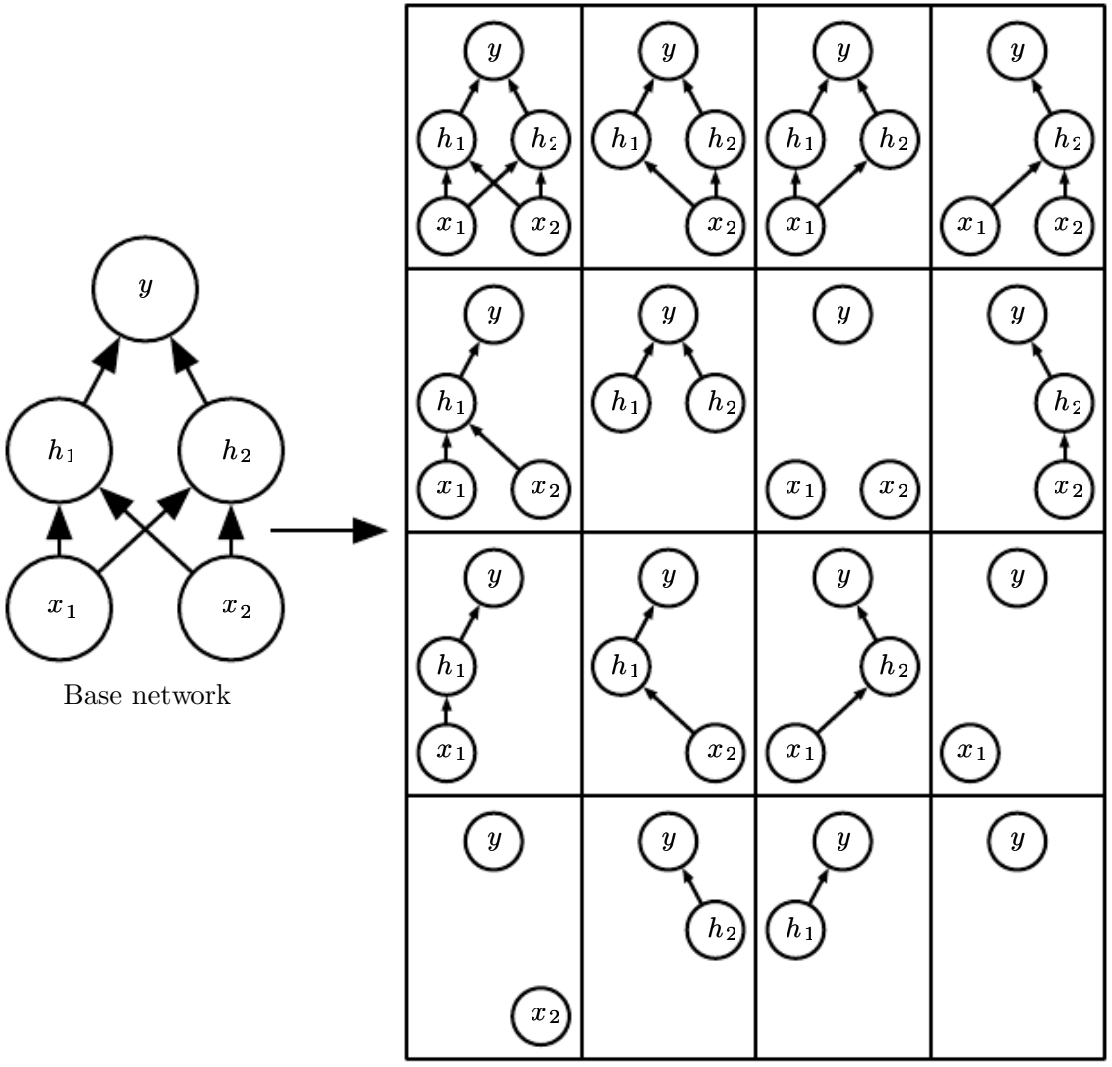
Forces the network to have a redundant representation.

Waaaait a second…
How could this possibly be a good idea?

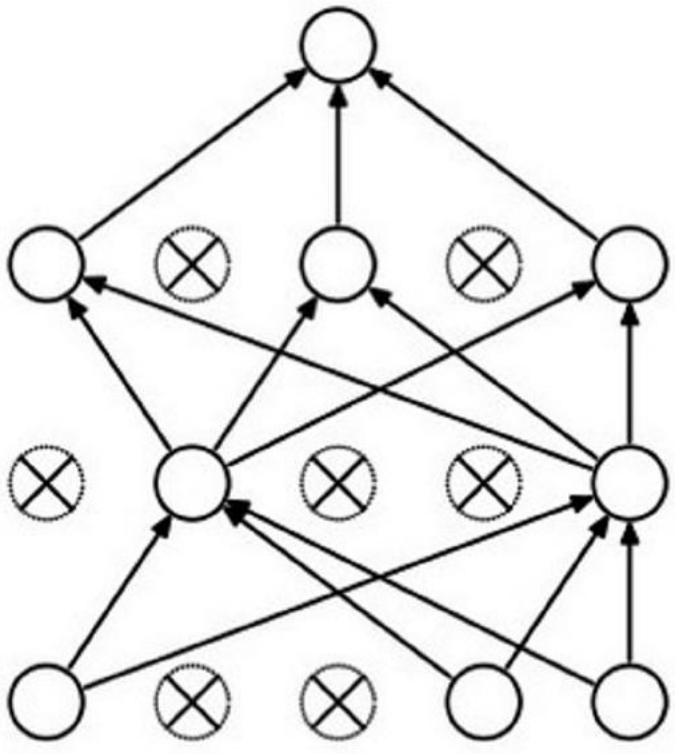
Another interpretation:

- Dropout is training a large ensemble of models (that share parameters).

- Each binary mask is one model, gets trained on only ~one datapoint.



At test time….



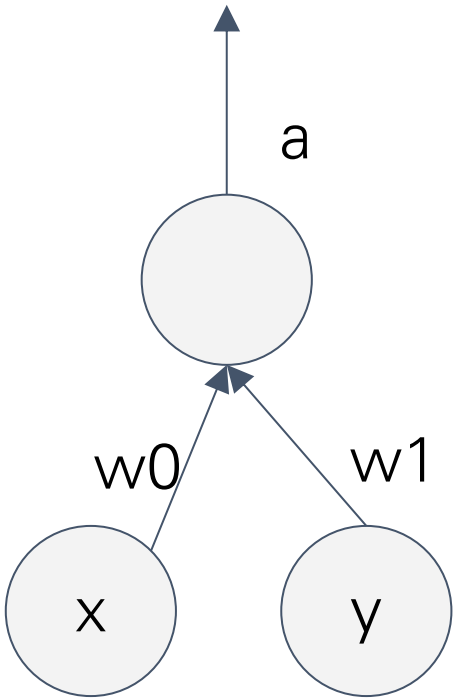
Ideally:
want to integrate out all the noise

Monte Carlo approximation:
do many forward passes with different dropout masks, average all predictions

At test time....

Can in fact do this with a single forward pass! (approximately)

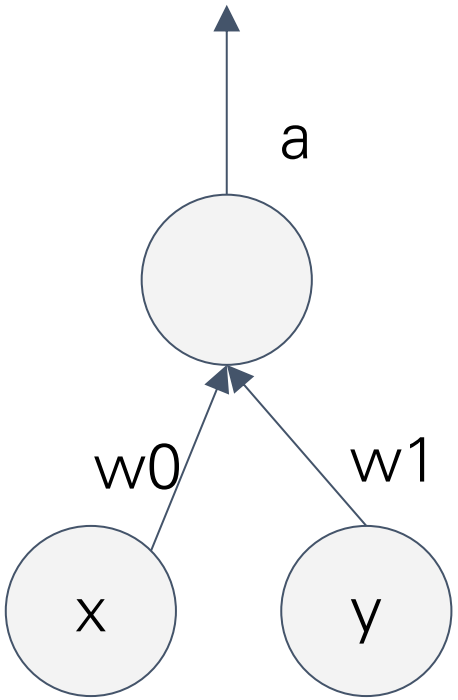
Leave all input neurons turned on (no dropout).



At test time....

Can in fact do this with a single forward pass! (approximately)

Leave all input neurons turned on (no dropout).

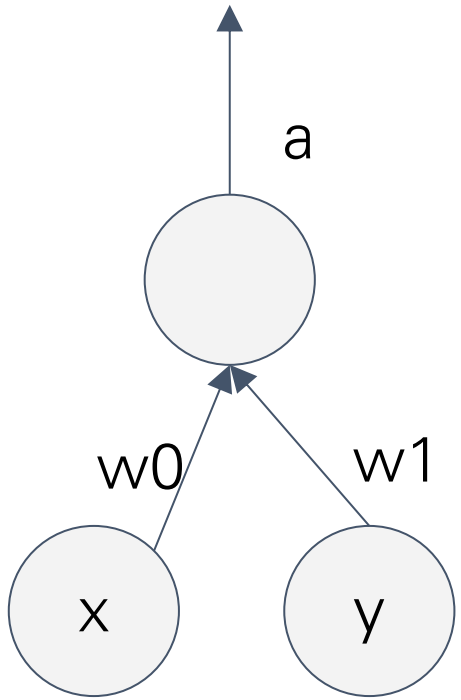


(this can be shown to be an approximation to evaluating the whole ensemble)

At test time....

Can in fact do this with a single forward pass! (approximately)

Leave all input neurons turned on (no dropout).



during test: **a = w0*x + w1*y**

during train:

E[a] = ¼ * (w0*0 + w1*0
w0*0 + w1*y
w0*x + w1*0
w0*x + w1*y)

= ¼ * (2 w0*x + 2 w1*y)

= **½ * (w0*x + w1*y)**

With p=0.5, using all inputs in the forward pass would inflate the activations by 2x from what the network was "used to" during training!
=> Have to compensate by scaling the activations back down by ½

We can do something approximate analytically

```
def predict(X):
  # ensembled forward pass
  H1 = np.maximum(0, np.dot(W1, X) + b1) * p # NOTE: scale the activations
  H2 = np.maximum(0, np.dot(W2, H1) + b2) * p # NOTE: scale the activations
  out = np.dot(W3, H2) + b3
```

At test time all neurons are active always
=> We must scale the activations so that for each neuron:
output at test time = expected output at training time

Dropout Summary

```
""" Vanilla Dropout: Not recommended implementation (see notes below) """

p = 0.5 # probability of keeping a unit active. higher = less dropout

def train_step(X):
  """ X contains the data """

  # forward pass for example 3-layer neural network
  H1 = np.maximum(0, np.dot(W1, X) + b1)
  U1 = np.random.rand(*H1.shape) < p # first dropout mask
  H1 *= U1 # drop!
  H2 = np.maximum(0, np.dot(W2, H1) + b2)
  U2 = np.random.rand(*H2.shape) < p # second dropout mask
  H2 *= U2 # drop!
  out = np.dot(W3, H2) + b3

  # backward pass: compute gradients... (not shown)
  # perform parameter update... (not shown)

def predict(X):
  # ensembled forward pass
  H1 = np.maximum(0, np.dot(W1, X) + b1) * p # NOTE: scale the activations
  H2 = np.maximum(0, np.dot(W2, H1) + b2) * p # NOTE: scale the activations
  out = np.dot(W3, H2) + b3
```

drop in forward pass

scale at test time

More common: "Inverted dropout"

```python
p = 0.5 # probability of keeping a unit active. higher = less dropout

def train_step(X):
  # forward pass for example 3-layer neural network
  H1 = np.maximum(0, np.dot(W1, X) + b1)
  U1 = (np.random.rand(*H1.shape) < p) / p # first dropout mask. Notice /p!
  H1 *= U1 # drop!
  H2 = np.maximum(0, np.dot(W2, H1) + b2)
  U2 = (np.random.rand(*H2.shape) < p) / p # second dropout mask. Notice /p!
  H2 *= U2 # drop!
  out = np.dot(W3, H2) + b3

  # backward pass: compute gradients... (not shown)
  # perform parameter update... (not shown)

def predict(X):
  # ensembled forward pass
  H1 = np.maximum(0, np.dot(W1, X) + b1) # no scaling necessary
  H2 = np.maximum(0, np.dot(W2, H1) + b2)
  out = np.dot(W3, H2) + b3
```

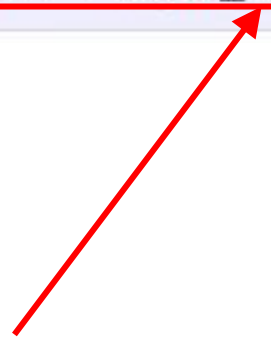
Optimization

Training a neural network, main loop:

```
# Vanilla Gradient Descent

while True:
  weights_grad = evaluate_gradient(loss_fun, data, weights)
  weights += - step_size * weights_grad # perform parameter update
```

Training a neural network, main loop:

```
# Vanilla Gradient Descent

while True:
  weights_grad = evaluate_gradient(loss_fun, data, weights)
  weights += - step_size * weights_grad # perform parameter update
```



simple gradient descent update
now: complicate.

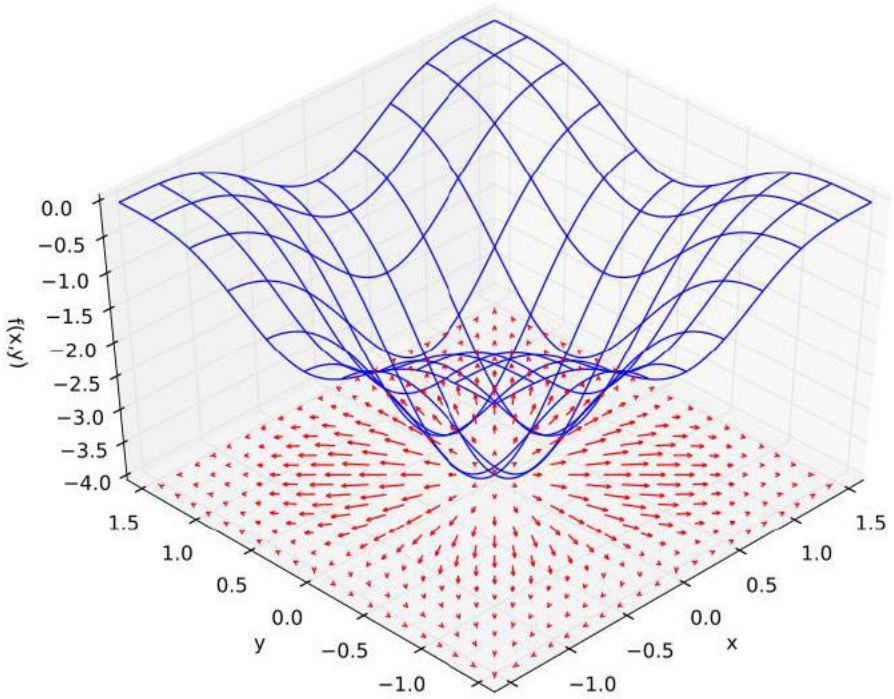
Gradients

- When we write $\nabla_W L(W)$, we mean the vector of partial derivatives wrt all coordinates of W:

$$\nabla_W L(W) = \left[\frac{\partial L}{\partial W_1}, \frac{\partial L}{\partial W_2}, \cdots, \frac{\partial L}{\partial W_m} \right]^T$$

  where $\frac{\partial L}{\partial W_i}$ measures how fast the loss changes vs. change in W_i

- **In figure:** loss surface is blue, gradient vectors are red:



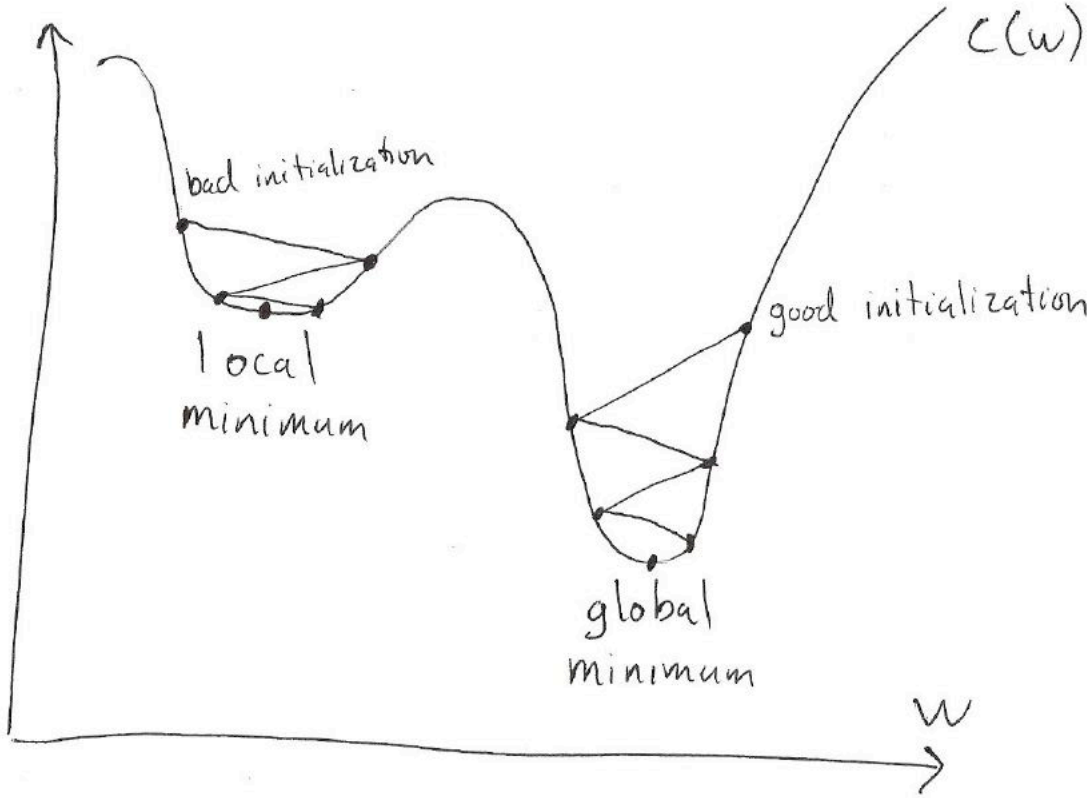
- When $\nabla_W L(W) = 0$, it means all the partials are zero, i.e. the loss is not changing in any direction.

- Note: arrows point out from a minimum, in toward a maximum

Slide adapted from John Canny

Optimization

- Visualizing gradient descent in one dimension:



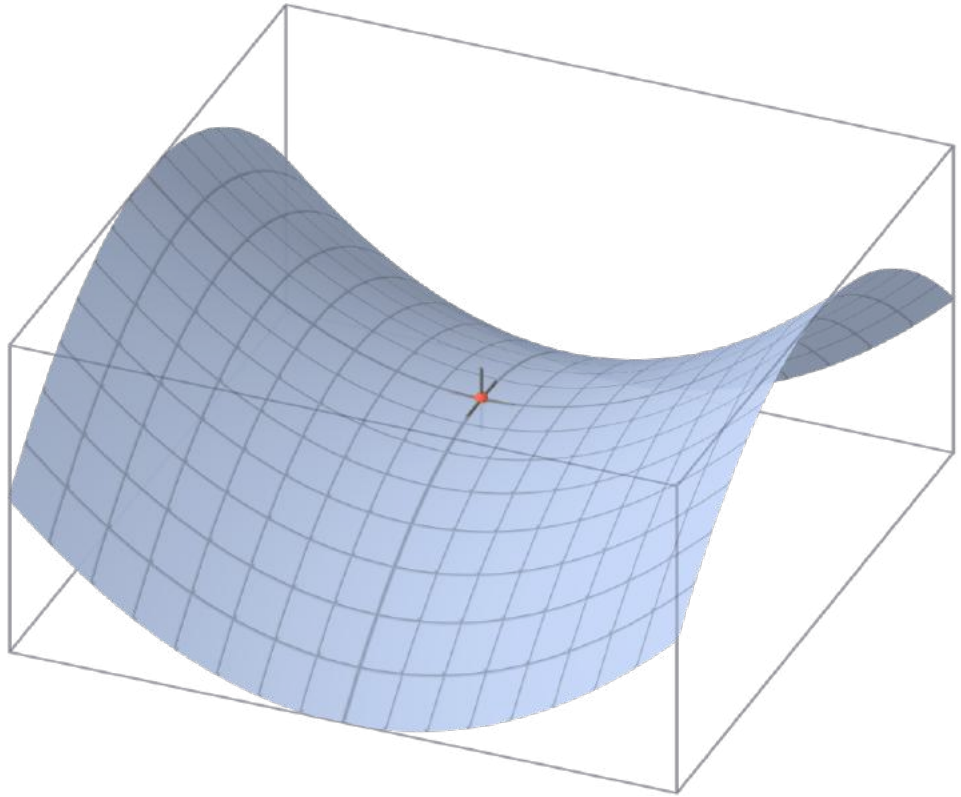
- The regions where gradient descent converges to a particular local minimum are called basins of attraction.

Local Minima

- Since the optimization problem is non-convex, it probably has local minima.
- This kept people from using neural nets for a long time, because they wanted guarantees they were getting the optimal solution.
- But are local minima really a problem?
  - Common view among practitioners: yes, there are local minima, but they're probably still pretty good.
    - Maybe your network wastes some hidden units, but then you can just make it larger.
  - It's very hard to demonstrate the existence of local minima in practice.
  - In any case, other optimization-related issues are much more important.

Saddle Points

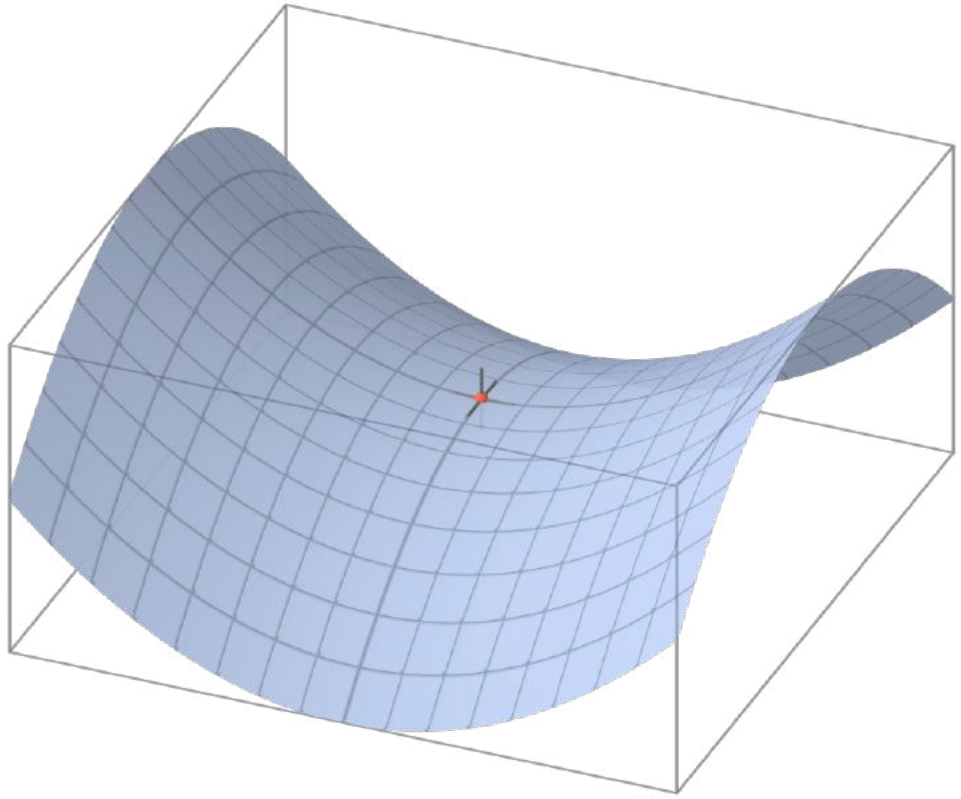
- At a saddle point, $\frac{\partial L}{\partial W} = 0$ even though we are not at a minimum. Some directions curve upwards, and others curve downwards.
- When would saddle points be a problem?
  - If we're exactly on the saddle point, then we're stuck.
  - If we're slightly to the side, then we can get unstuck.



Saddle Points

- At a saddle point, $\frac{\partial L}{\partial W} = 0$ even though we are not at a minimum. Some directions curve upwards, and others curve downwards.

- When would saddle points be a problem?
  - If we're exactly on the saddle point, then we're stuck.
  - If we're slightly to the side, then we can get unstuck.

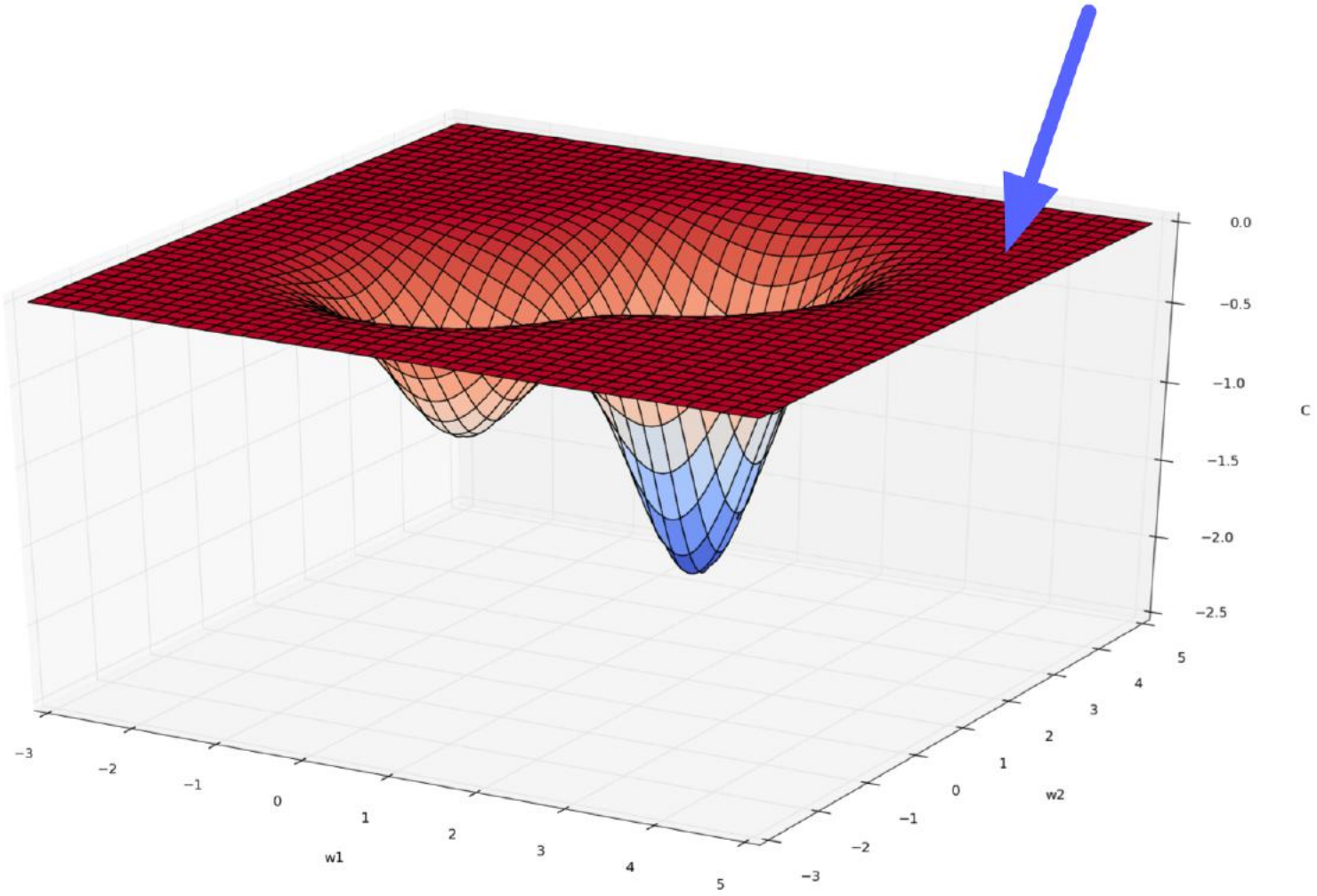


Saddle points much more common in high dimensions!

Y. Dauphin et al. Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. In NIPS 2014

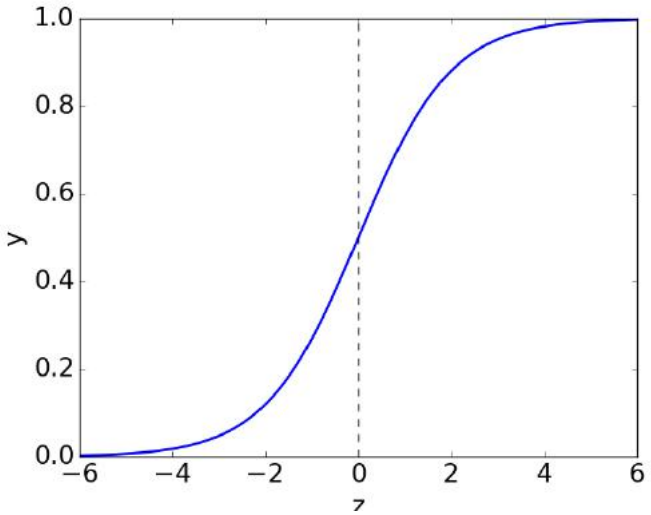
Plateaux

- A flat region is called a **plateau**. (Plural: plateaux)

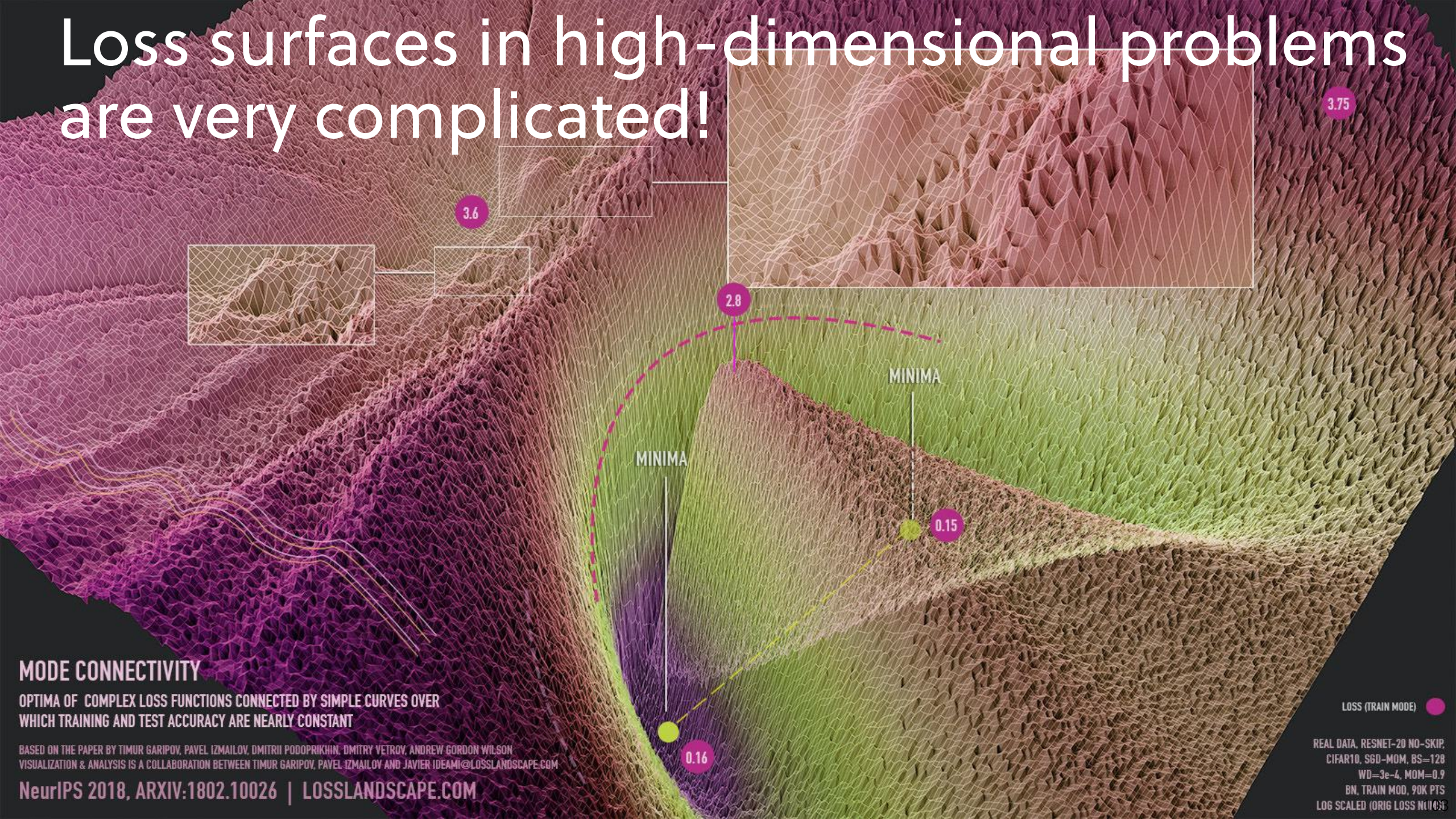


Plateaux

- An important example of a plateau is a **saturated unit**. This is when it is in the flat region of its activation function.
- If $\varphi'(z_i)$ is always close to zero, then the weights will get stuck.
- If there is a ReLU unit whose input z_i is always negative, the weight derivatives will be exactly 0. We call this a **dead unit**.



Loss surfaces in high-dimensional problems are very complicated!



Batch Gradient Descent

Algorithm 1 Batch Gradient Descent at Iteration k

Require: Learning rate ϵ_k

Require: Initial Parameter θ

1: **while** stopping criteria not met **do**

2: Compute gradient estimate over N examples:

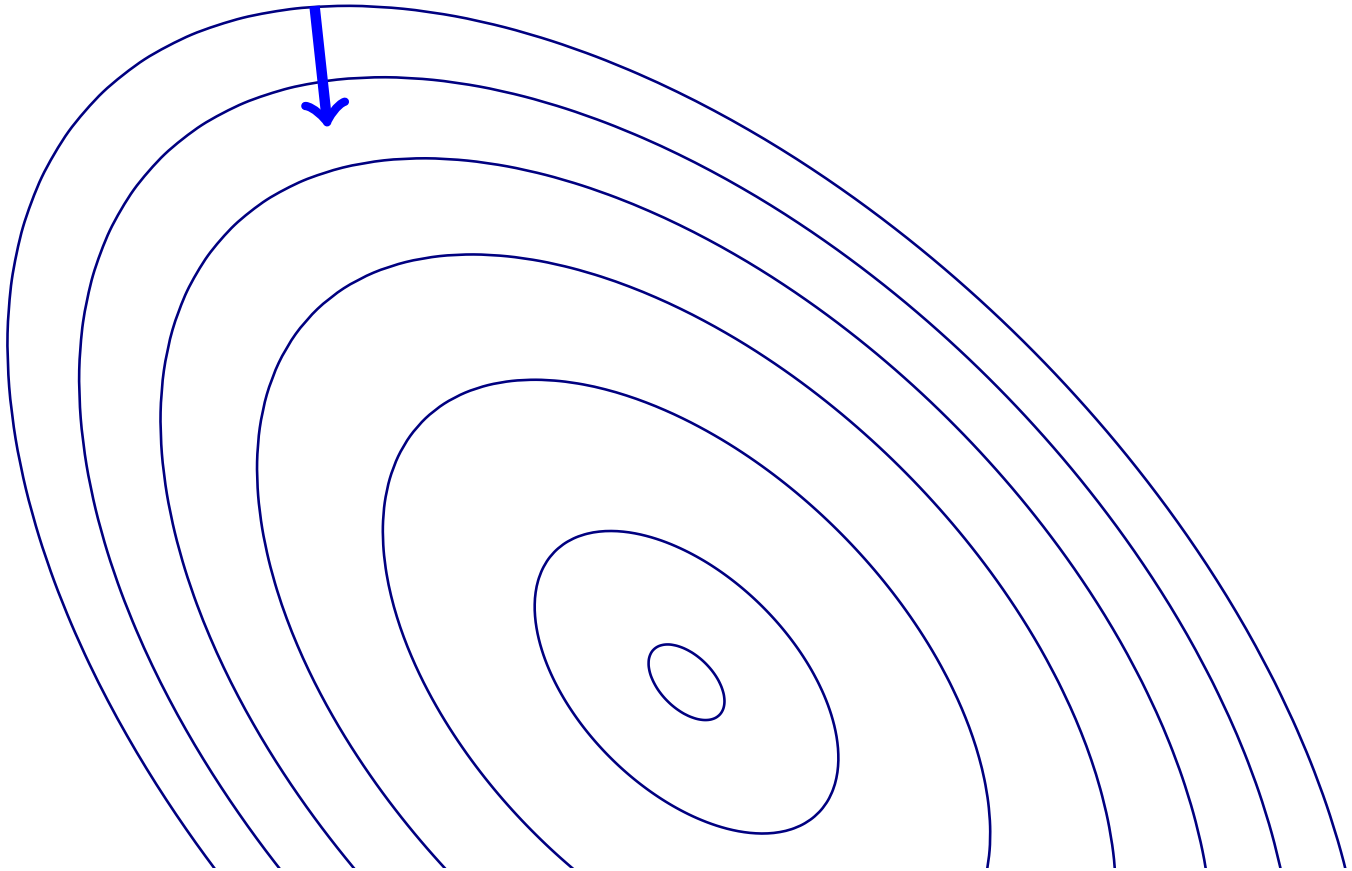
3: $\hat{\mathbf{g}} \leftarrow +\frac{1}{N}\nabla_\theta \sum_i L(f(\mathbf{x}^{(i)};\theta), \mathbf{y}^{(i)})$

4: Apply Update: $\theta \leftarrow \theta - \epsilon\hat{\mathbf{g}}$

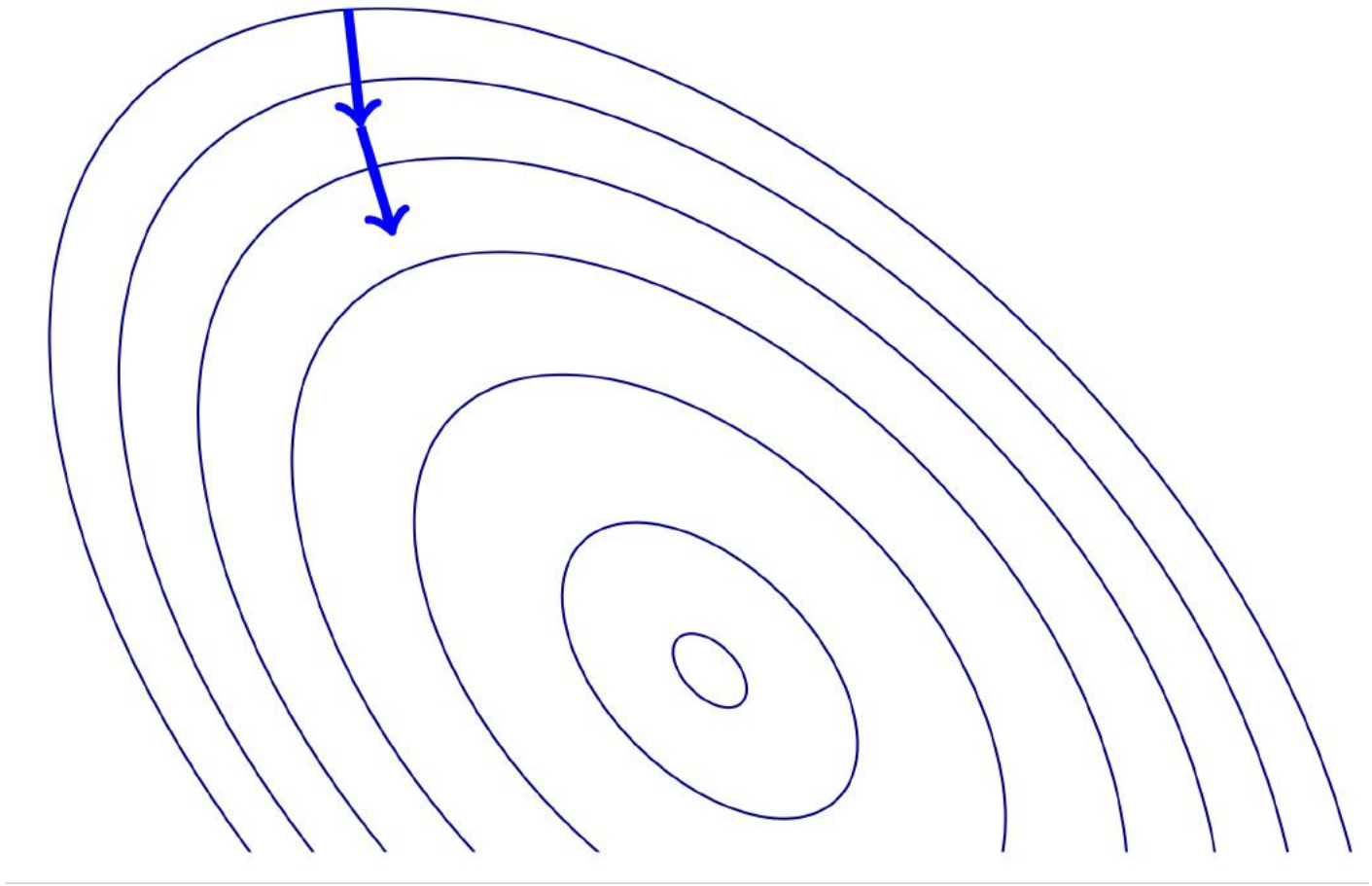
5: **end while**

- Positive: Gradient estimates are stable
- Negative: Need to compute gradients over the entire training for one update

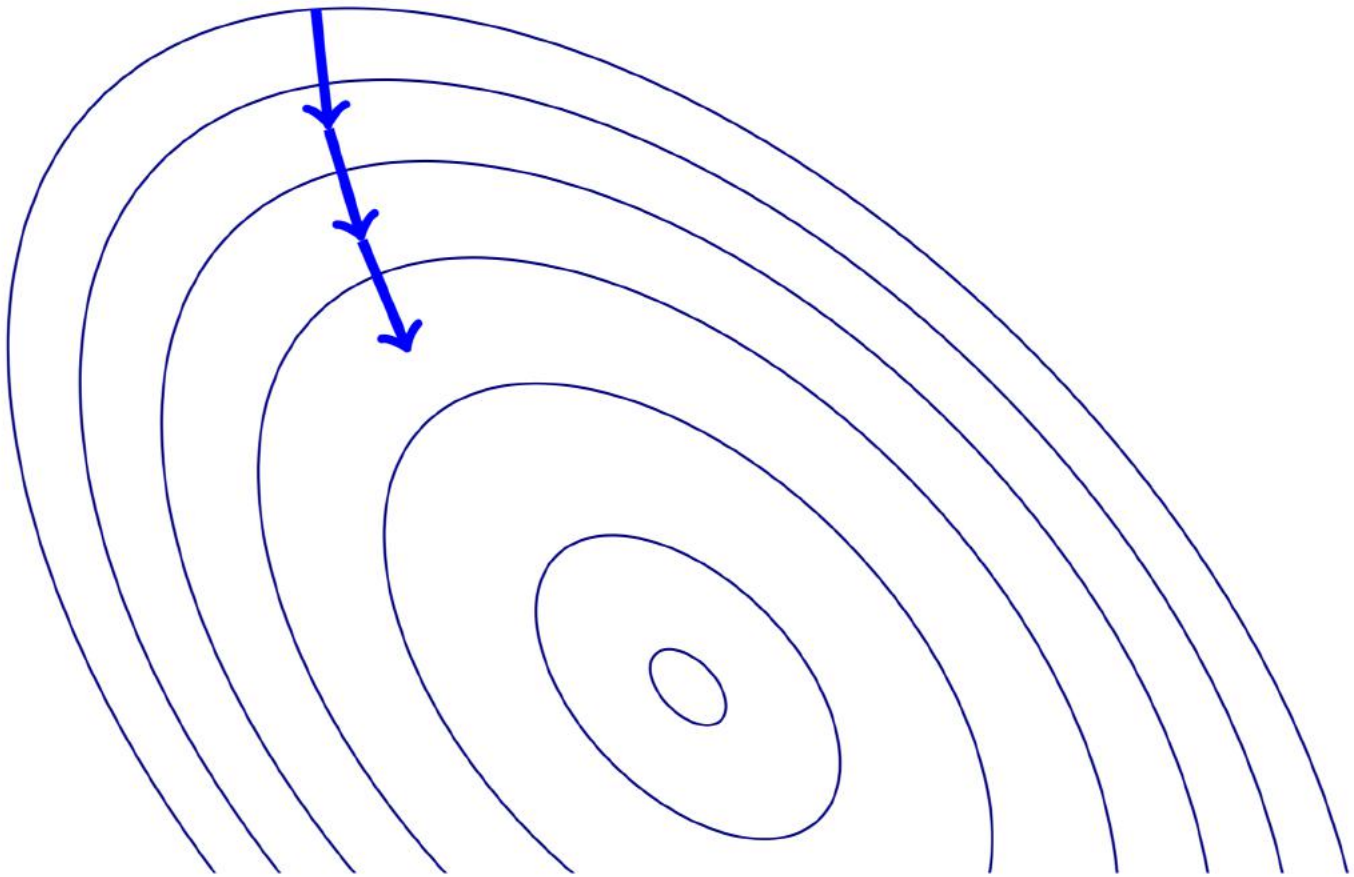
Gradient Descent



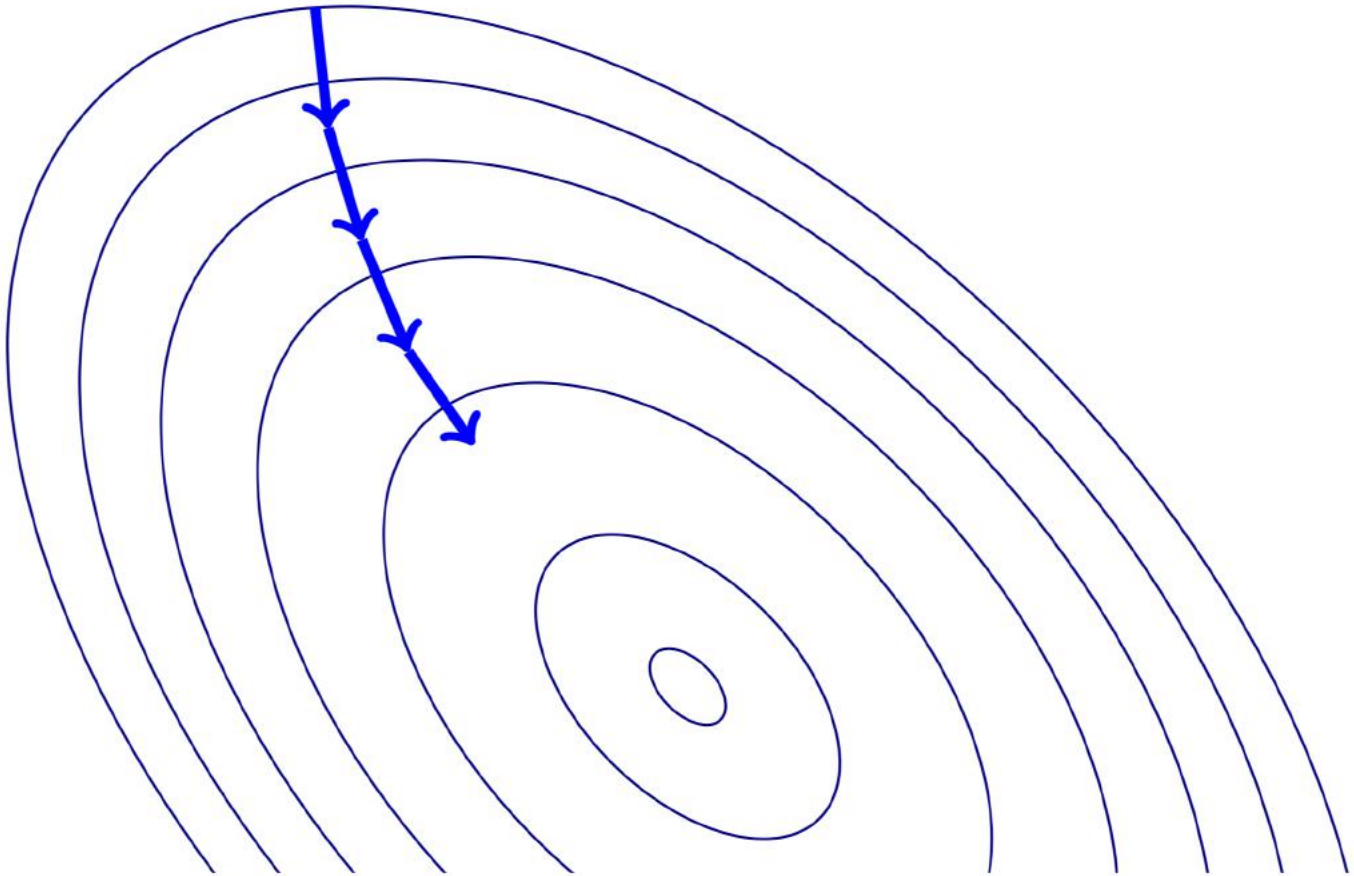
Gradient Descent



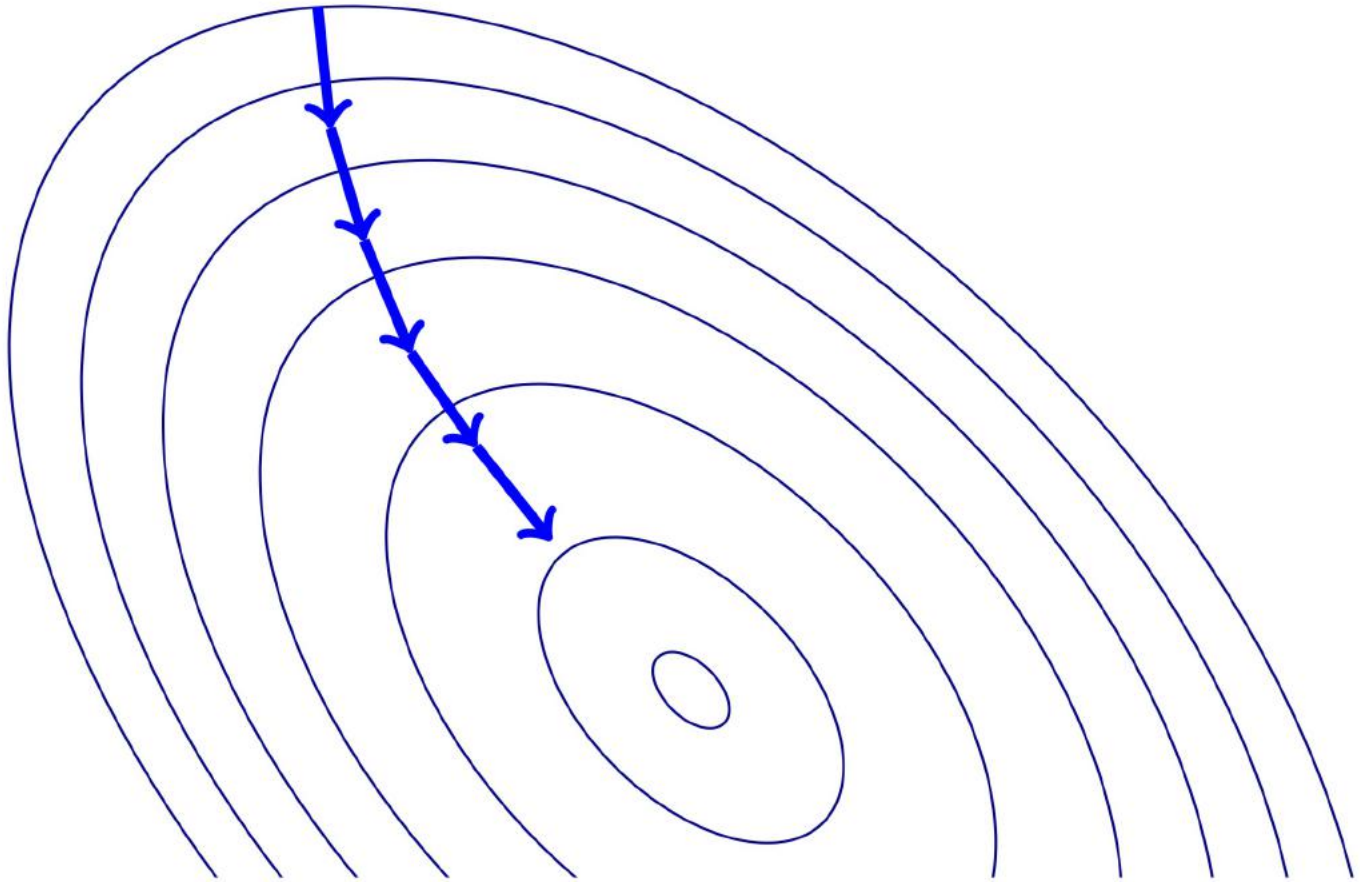
Gradient Descent



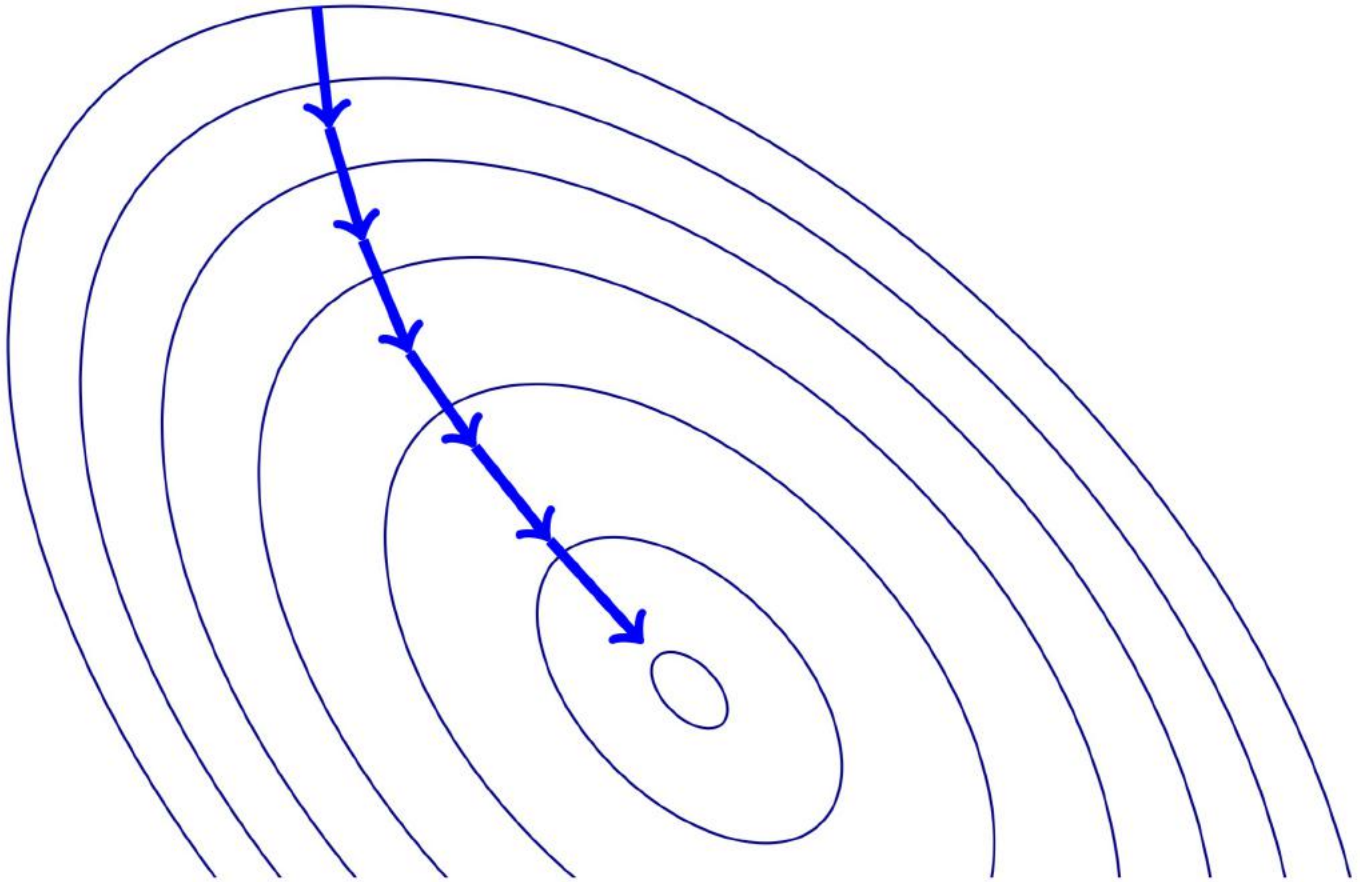
Gradient Descent



Gradient Descent



Gradient Descent



Stochastic Batch Gradient Descent

Algorithm 2 Stochastic Gradient Descent at Iteration k

Require: Learning rate ϵ_k

Require: Initial Parameter θ

1: **while** stopping criteria not met **do**

2: Sample example $(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})$ from training set

3: Compute gradient estimate:

4: $\hat{\mathbf{g}} \leftarrow +\nabla_\theta L(f(\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)})$

5: Apply Update: $\theta \leftarrow \theta - \epsilon\hat{\mathbf{g}}$

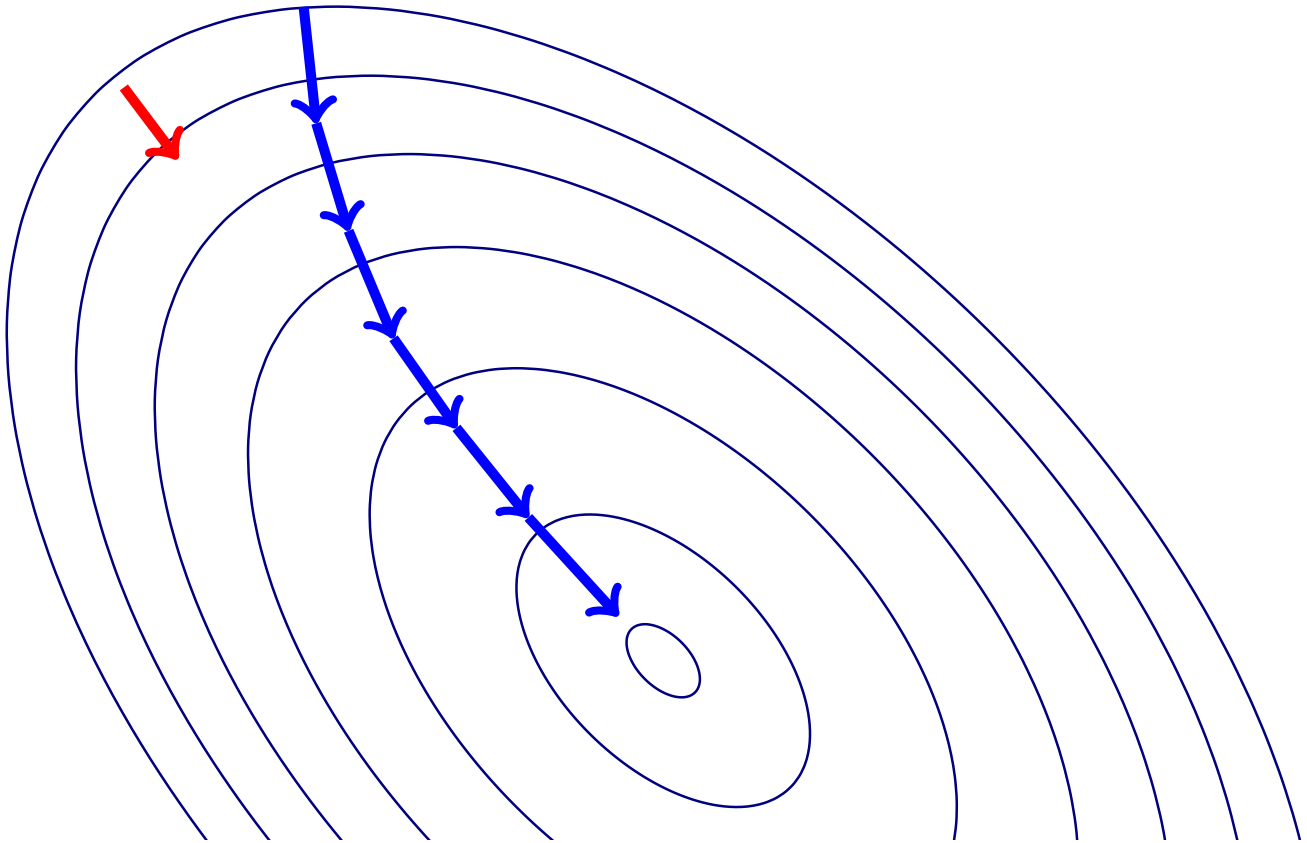
6: **end while**

Minibatching

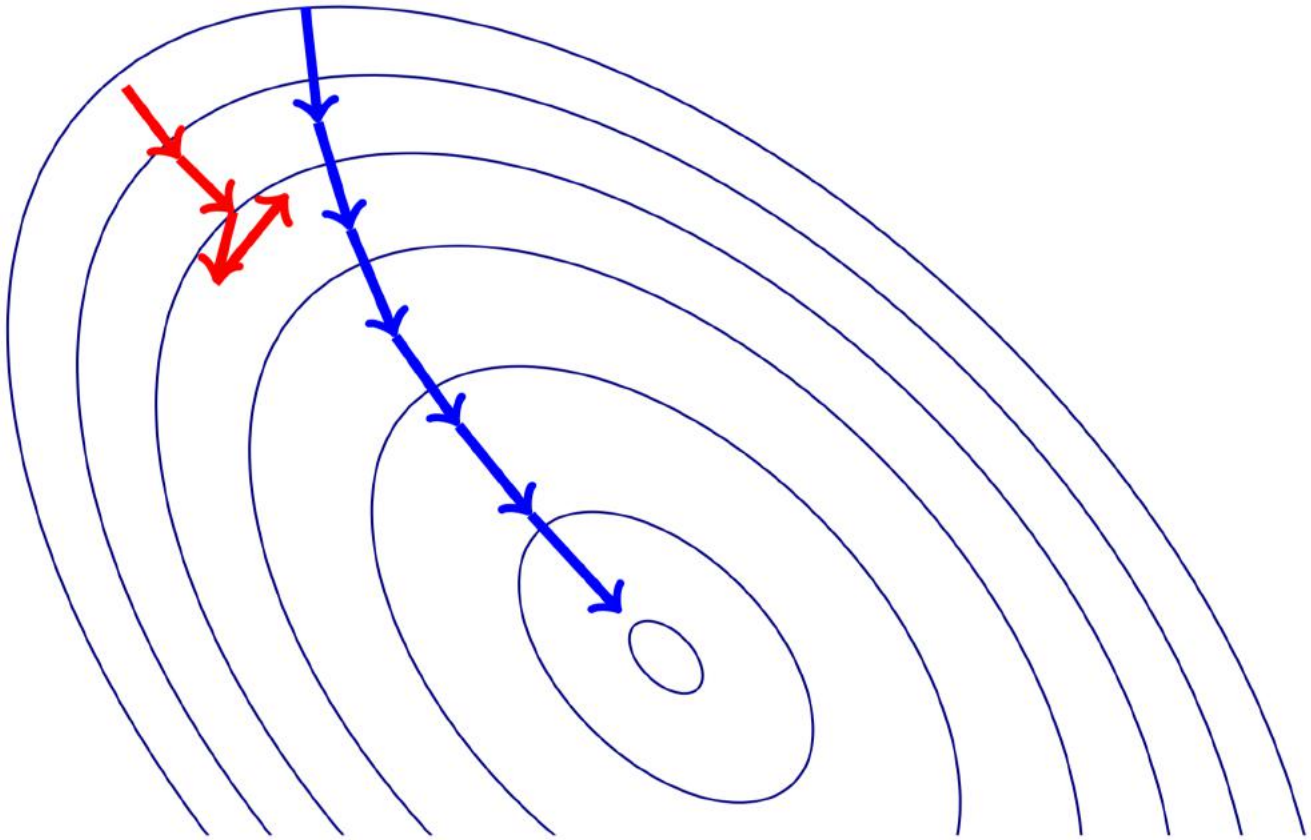
- Potential Problem: Gradient estimates can be very noisy
- Obvious Solution: Use larger mini-batches
- **Advantage:** Computation time per update does not depend on number of training examples N

- This allows convergence on extremely large datasets
- See: Large Scale Learning with Stochastic Gradient Descent by Leon Bottou

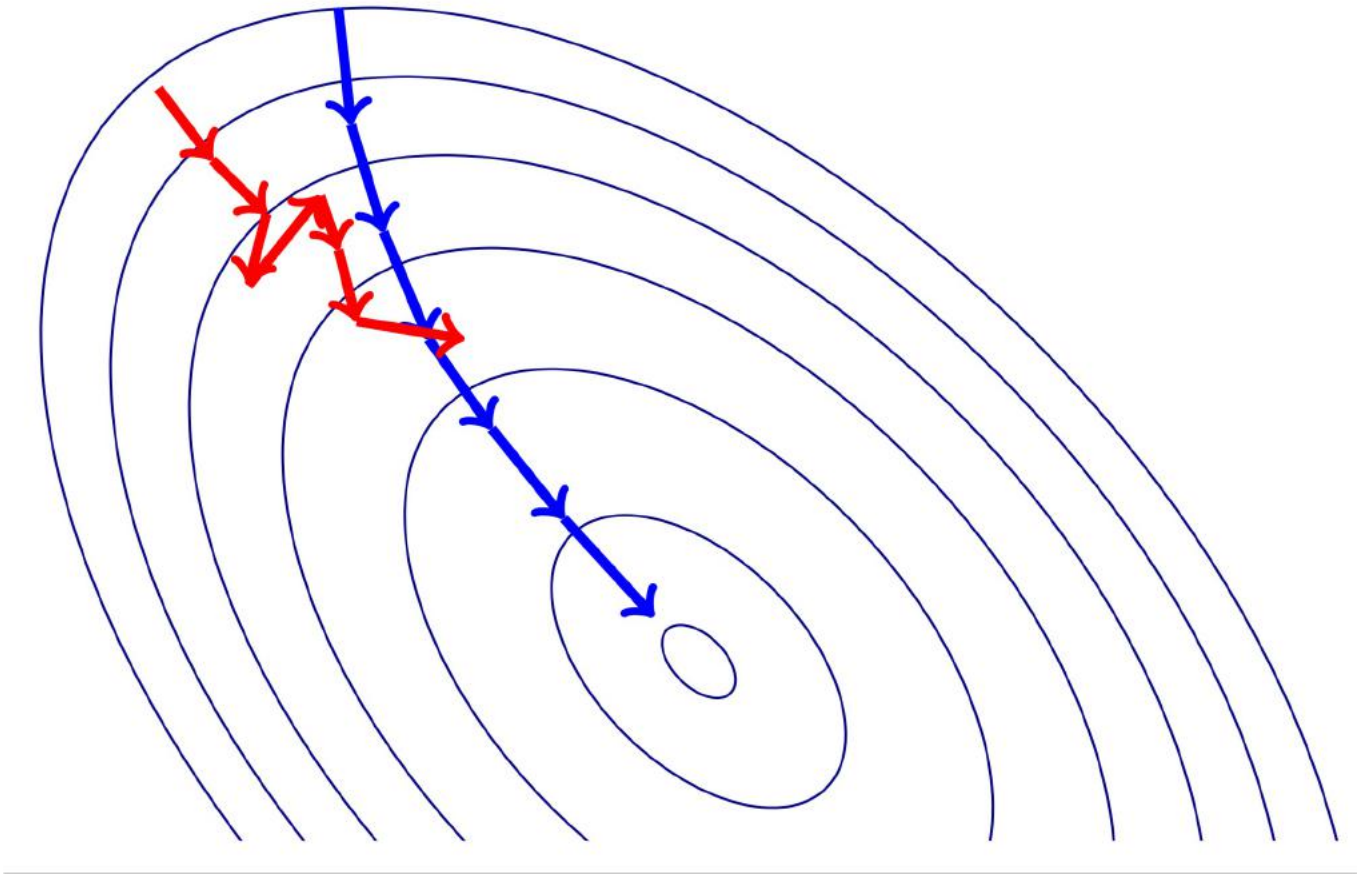
Stochastic Gradient Descent



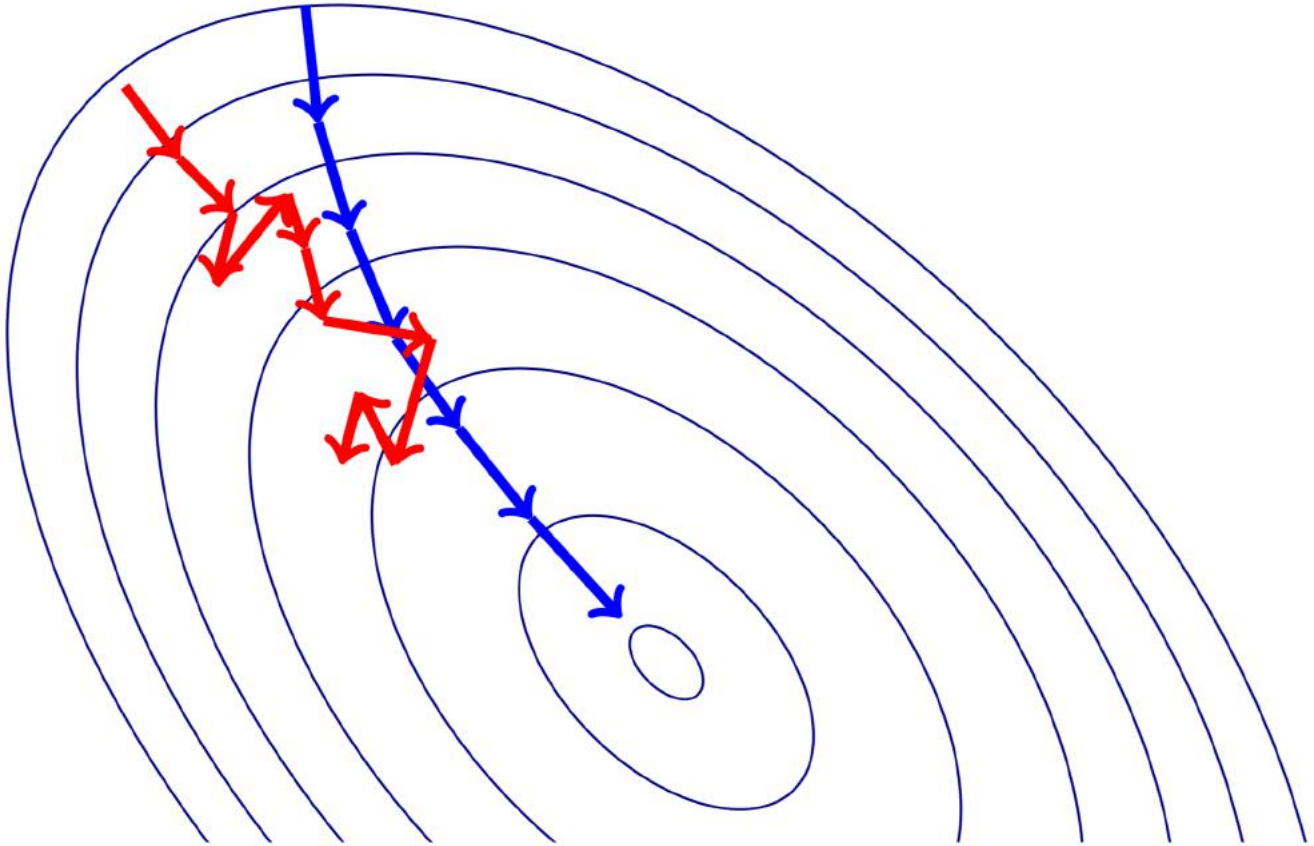
Stochastic Gradient Descent



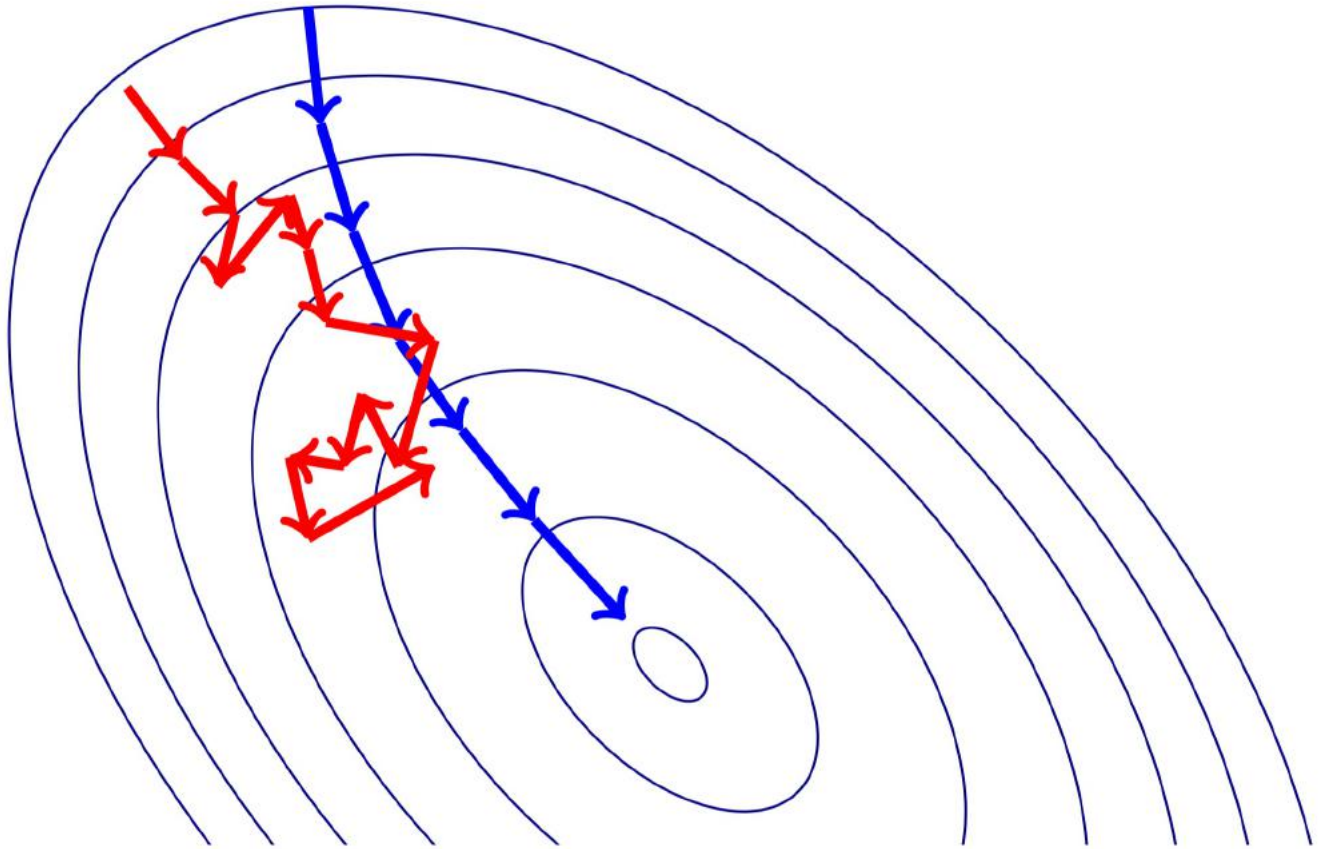
Stochastic Gradient Descent



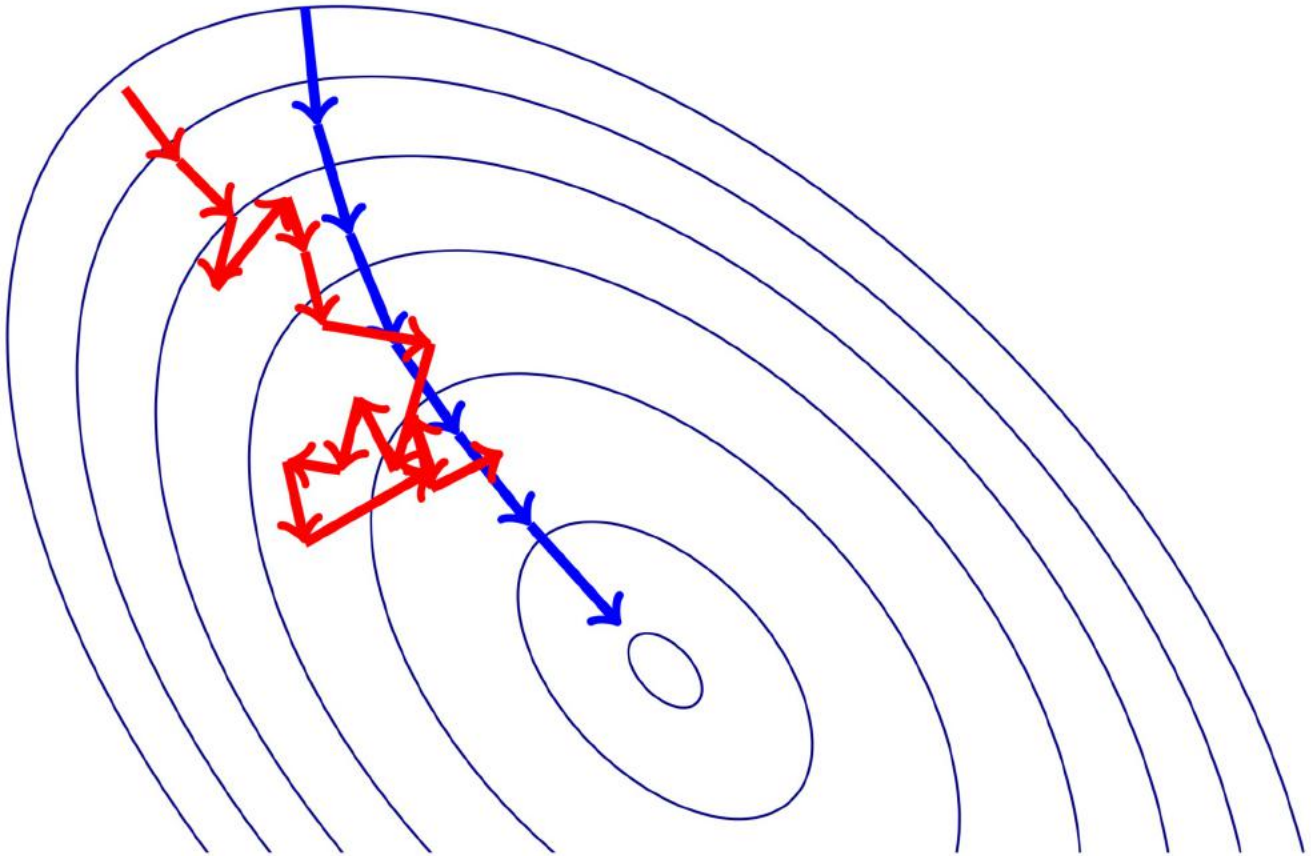
Stochastic Gradient Descent



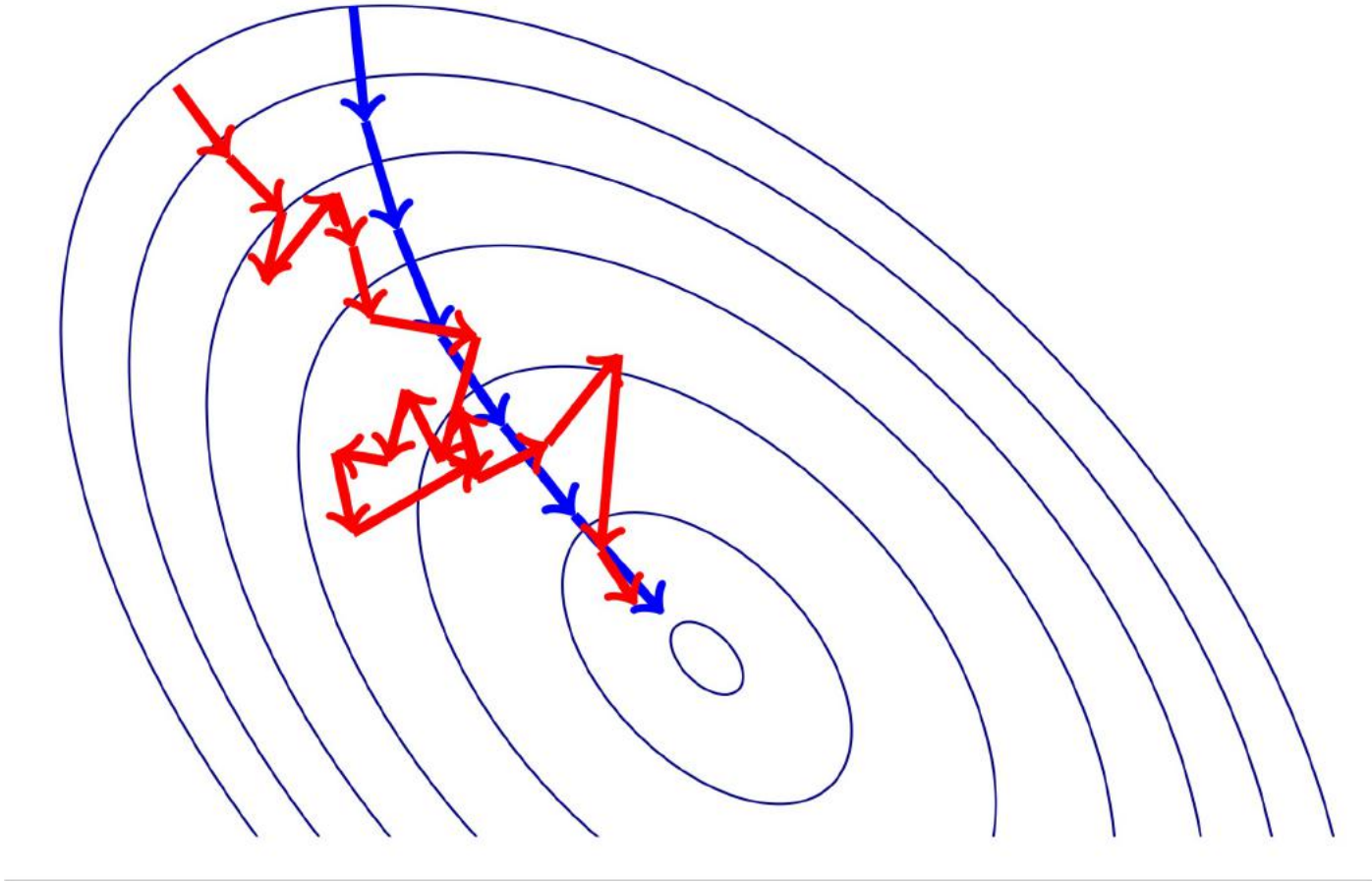
Stochastic Gradient Descent



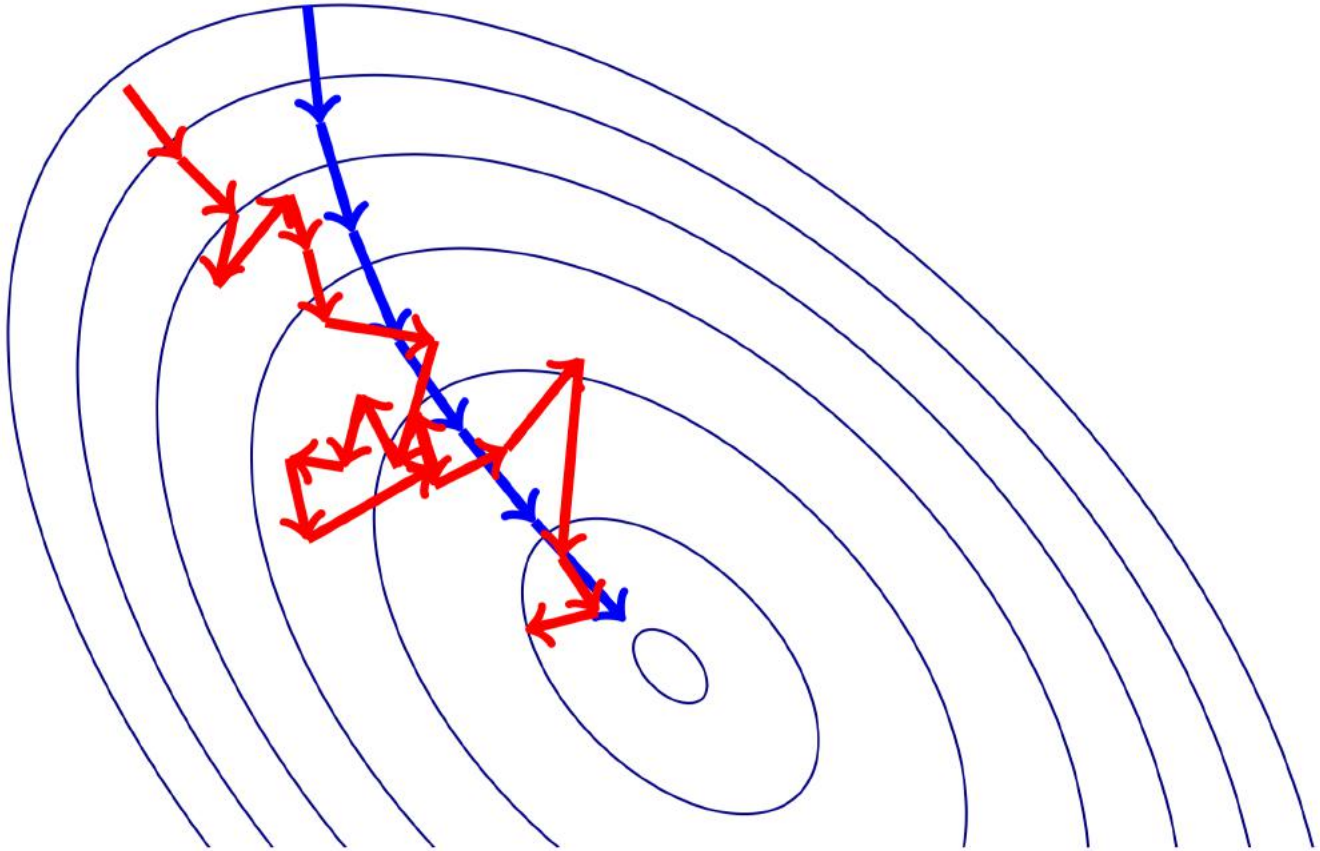
Stochastic Gradient Descent



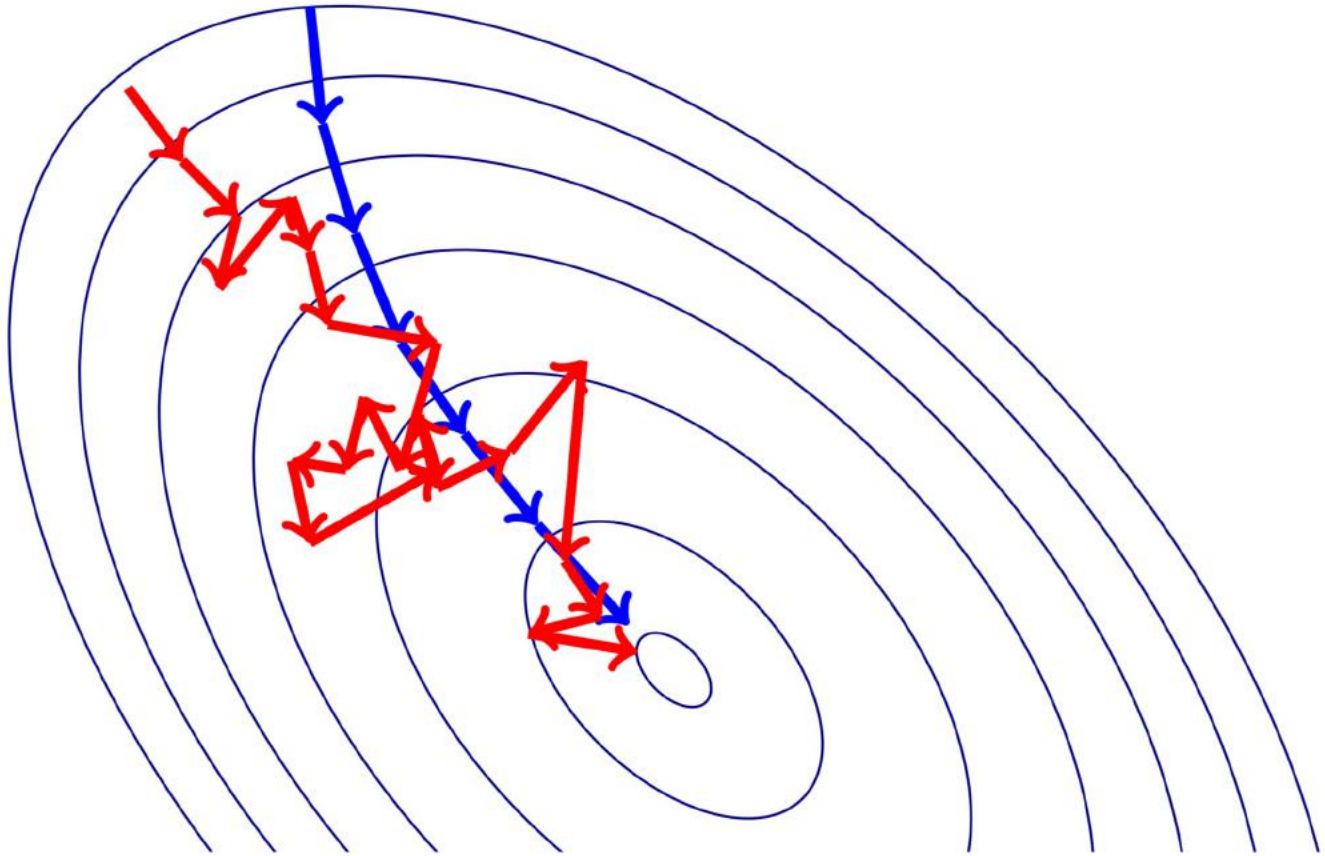
Stochastic Gradient Descent



Stochastic Gradient Descent



Stochastic Gradient Descent



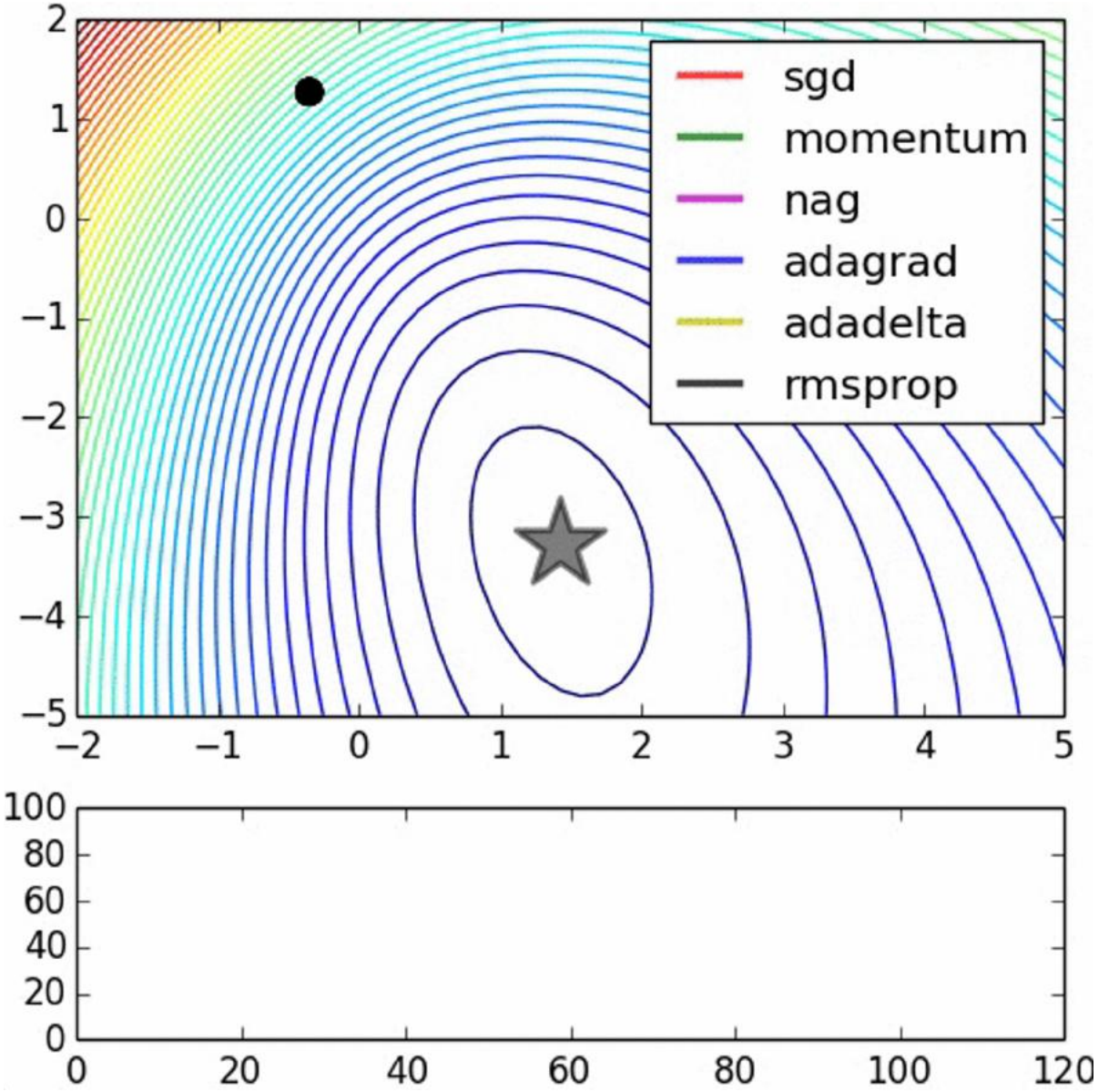
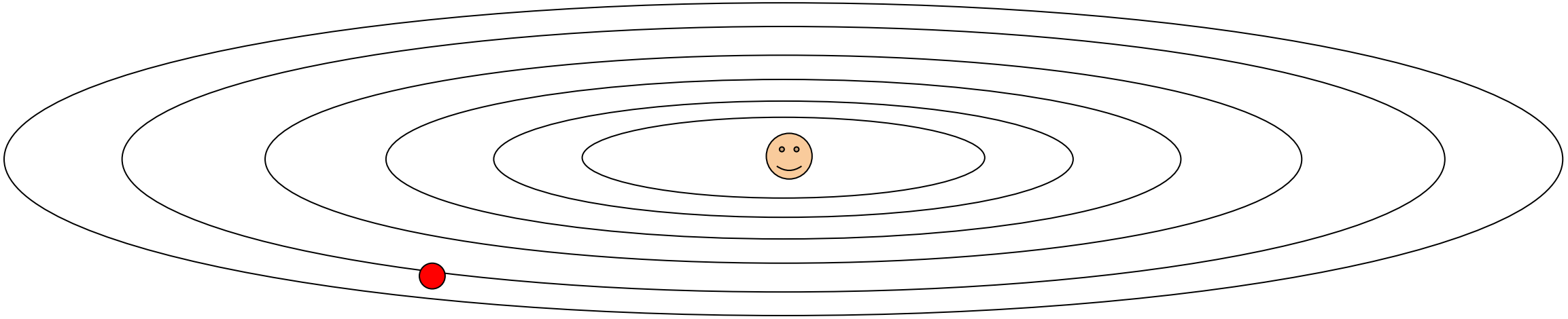


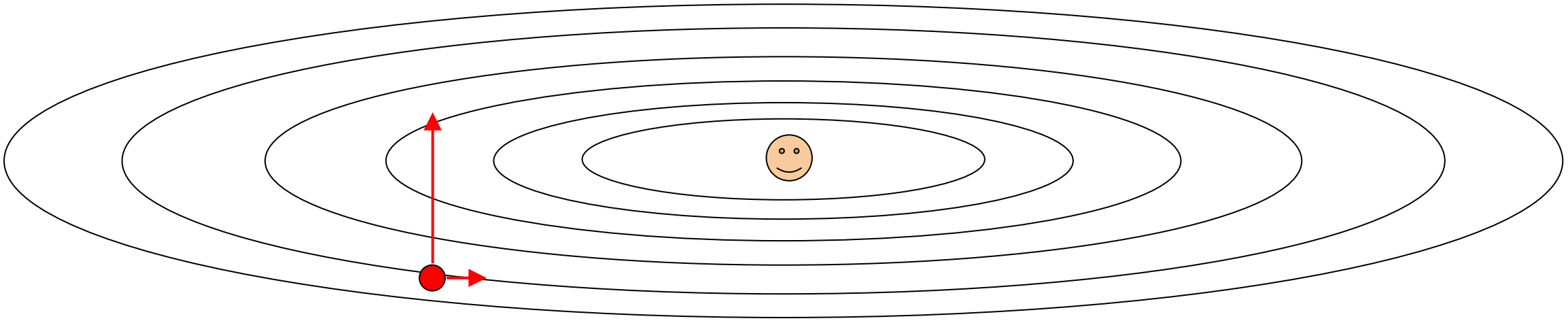
Image credits: Alec Radford

Suppose loss function is steep vertically but shallow horizontally:



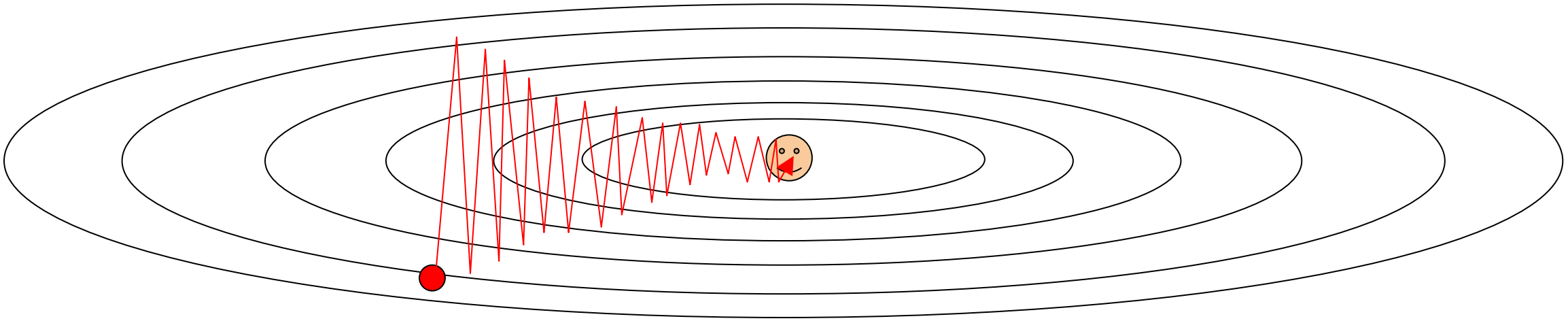
Q: What is the trajectory along which we converge towards the minimum with SGD?

Suppose loss function is steep vertically but shallow horizontally:



Q: What is the trajectory along which we converge towards the minimum with SGD?

Suppose loss function is steep vertically but shallow horizontally:



Q: What is the trajectory along which we converge towards the minimum with SGD?

very slow progress along flat direction, jitter along steep one

Momentum update

SGD

$$x_{t+1} = x_t - \alpha \nabla f(x_t)$$

```
while True:
  dx = compute_gradient(x)
  x += learning_rate * dx
```

SGD+Momentum

$$v_{t+1} = \rho v_t + \nabla f(x_t)$$

$$x_{t+1} = x_t - \alpha v_{t+1}$$

```
vx = 0
while True:
  dx = compute_gradient(x)
  vx = rho * vx + dx
  x += learning_rate * vx
```

Momentum update

SGD

$$x_{t+1} = x_t - \alpha \nabla f(x_t)$$

```
while True:
  dx = compute_gradient(x)
  x += learning_rate * dx
```

SGD+Momentum

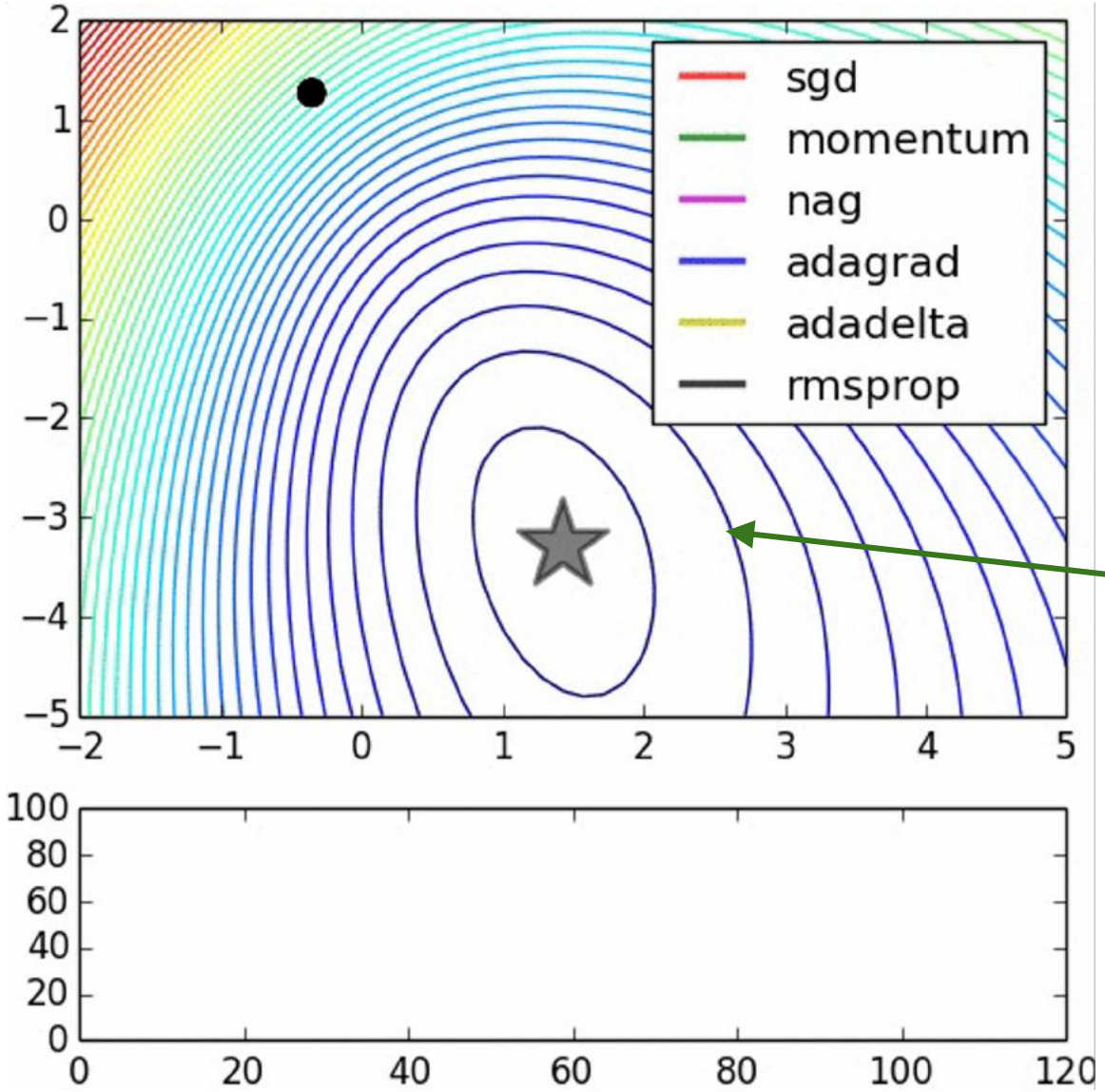
$$v_{t+1} = \rho v_t + \nabla f(x_t)$$

$$x_{t+1} = x_t - \alpha v_{t+1}$$

```
vx = 0
while True:
  dx = compute_gradient(x)
  vx = rho * vx + dx
  x += learning_rate * vx
```

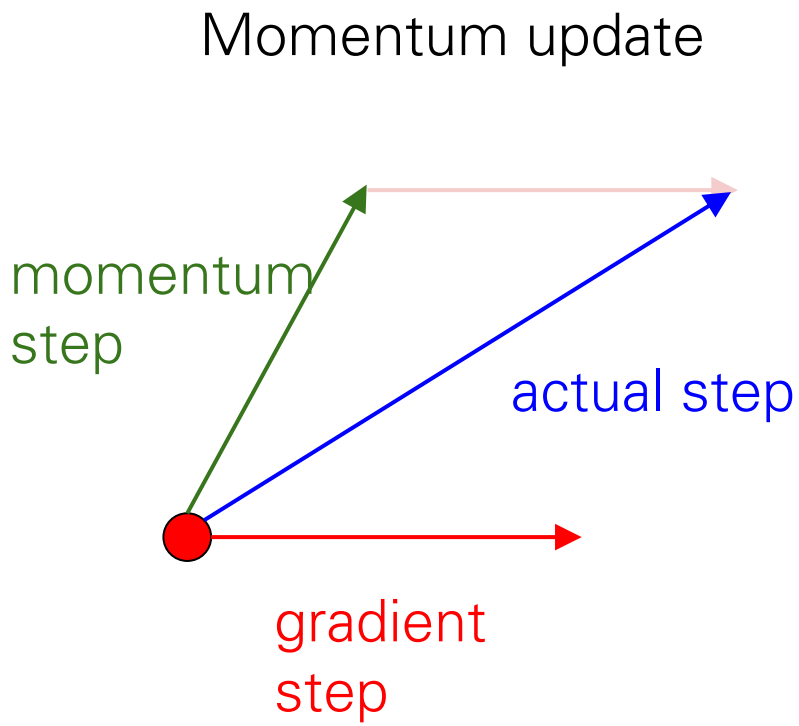
- Build up "velocity" as a running mean of gradients
- Rho gives "friction"; typically rho=0.9 or 0.99

SGD vs Momentum



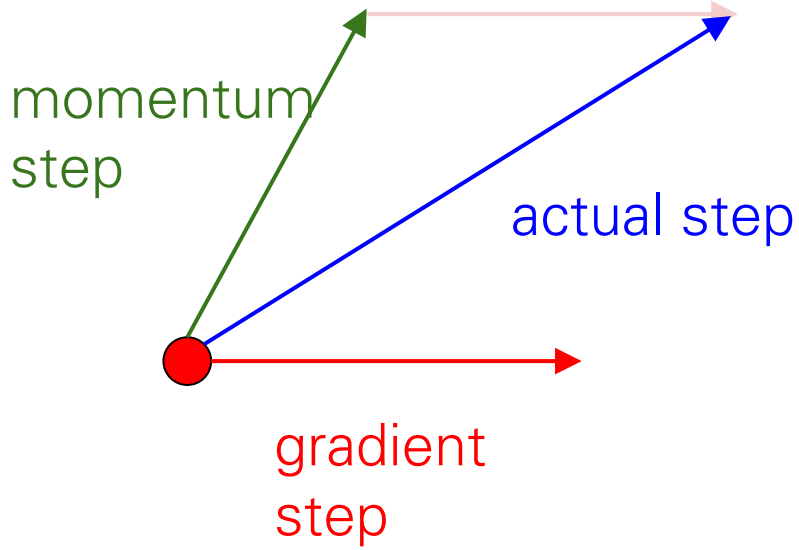
notice momentum overshooting the target, but overall getting to the minimum much faster.

SGD + Momentum

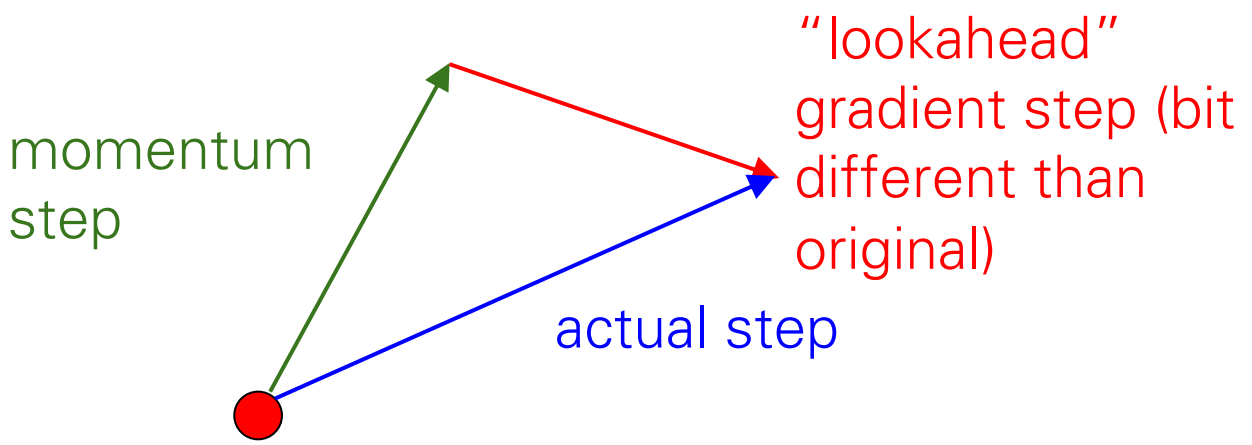


Nesterov Momentum

Momentum update



Nesterov momentum update



Nesterov: the only difference...

$$v_t = \mu v_{t-1} - \epsilon \nabla f(\theta_{t-1} + \mu v_{t-1})$$

$$\theta_t = \theta_{t-1} + v_t$$

Nesterov Momentum

$$v_{t+1} = \rho v_t - \alpha \nabla f(x_t + \rho v_t)$$

$$x_{t+1} = x_t + v_{t+1}$$

Nesterov Momentum

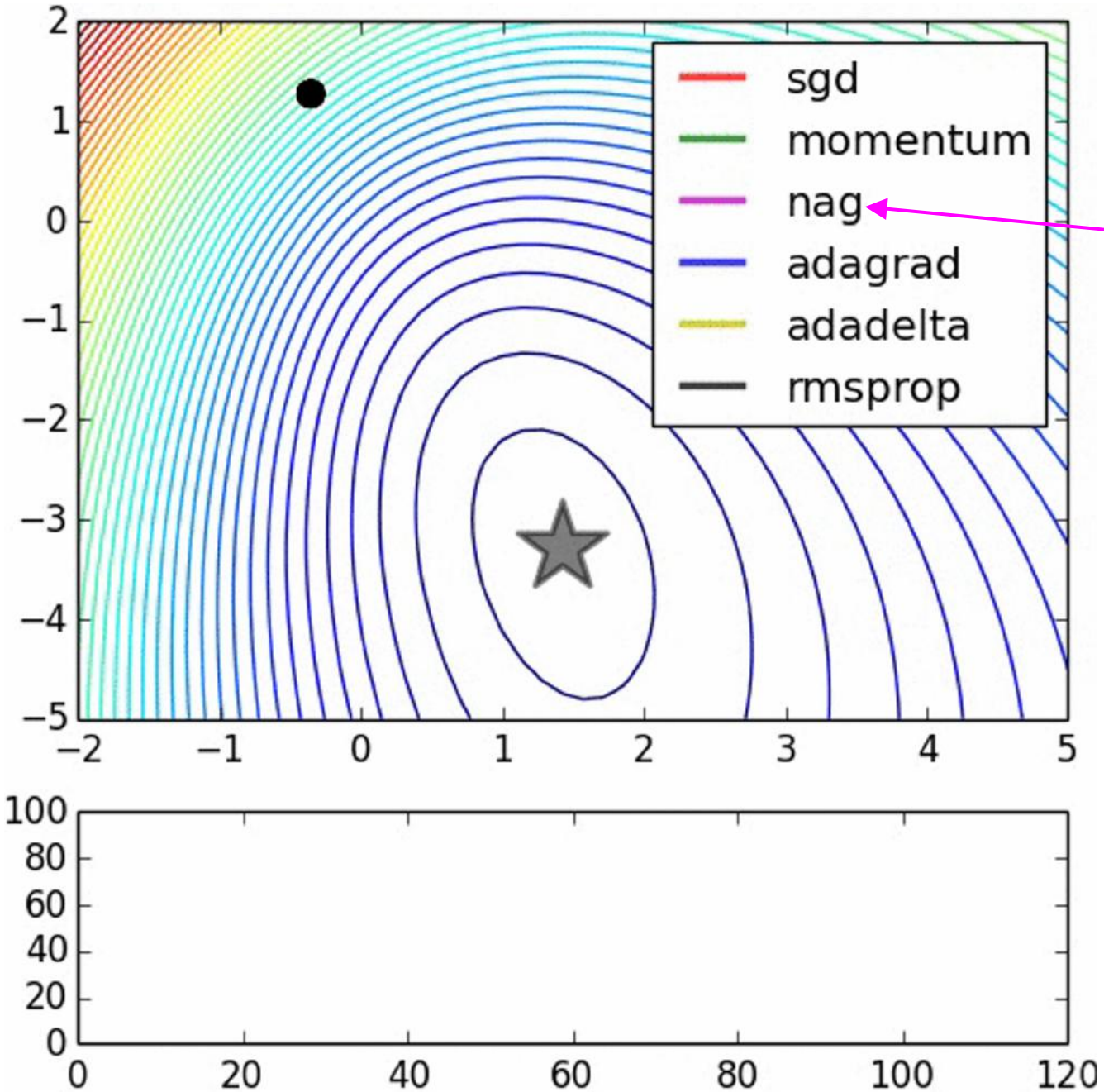
$$v_{t+1} = \rho v_t - \alpha \nabla f(x_t + \rho v_t)$$
$$x_{t+1} = x_t + v_{t+1}$$

Annoying, usually we want update in terms of $x_t, \nabla f(x_t)$

Change of variables $\tilde{x}_t = x_t + \rho v_t$ and rearrange:

$$v_{t+1} = \rho v_t - \alpha \nabla f(\tilde{x}_t)$$
$$\tilde{x}_{t+1} = \tilde{x}_t - \rho v_t + (1+\rho) v_{t+1}$$
$$= \tilde{x}_t + v_{t+1} + \rho(v_{t+1} - v_t)$$

```
dx = compute_gradient(x)
old_v = v
v = rho * v - learning_rate * dx
x += -rho * old_v + (1 + rho) * v
```



nag = Nesterov Accelerated Gradient

AdaGrad update

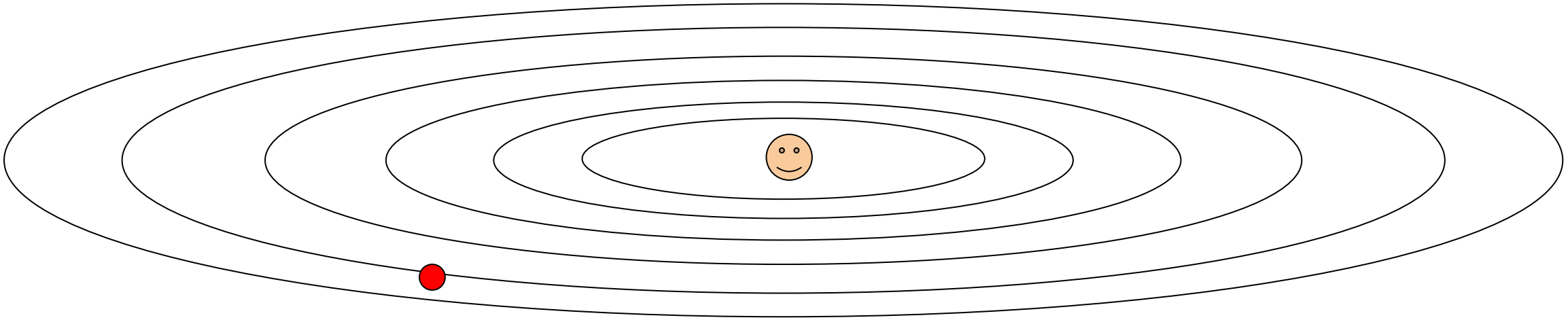
```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared += dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

Added element-wise scaling of the gradient based on the historical sum of squares in each dimension

[Duchi et al., 2011]

AdaGrad update

```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared += dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

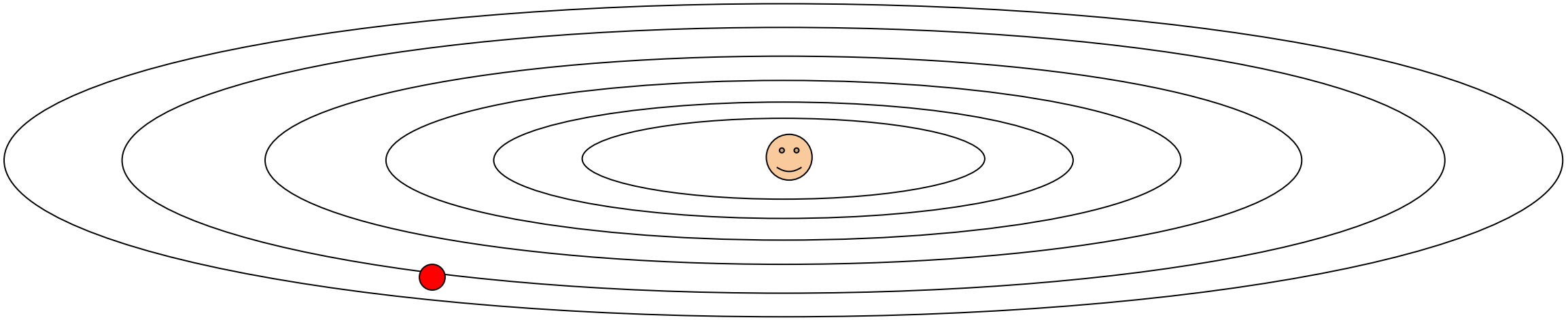


Q: What happens with AdaGrad?

Weights that receive high gradients will have their effective learning rate reduced, while weights that receive small updates will have their effective learning rate increased!

AdaGrad update

```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared += dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```



Q2: What happens to the step size over long time?

The adaptive learning scheme is monotonic, which is usually too aggressive and stops the learning process too early.

RMSProp

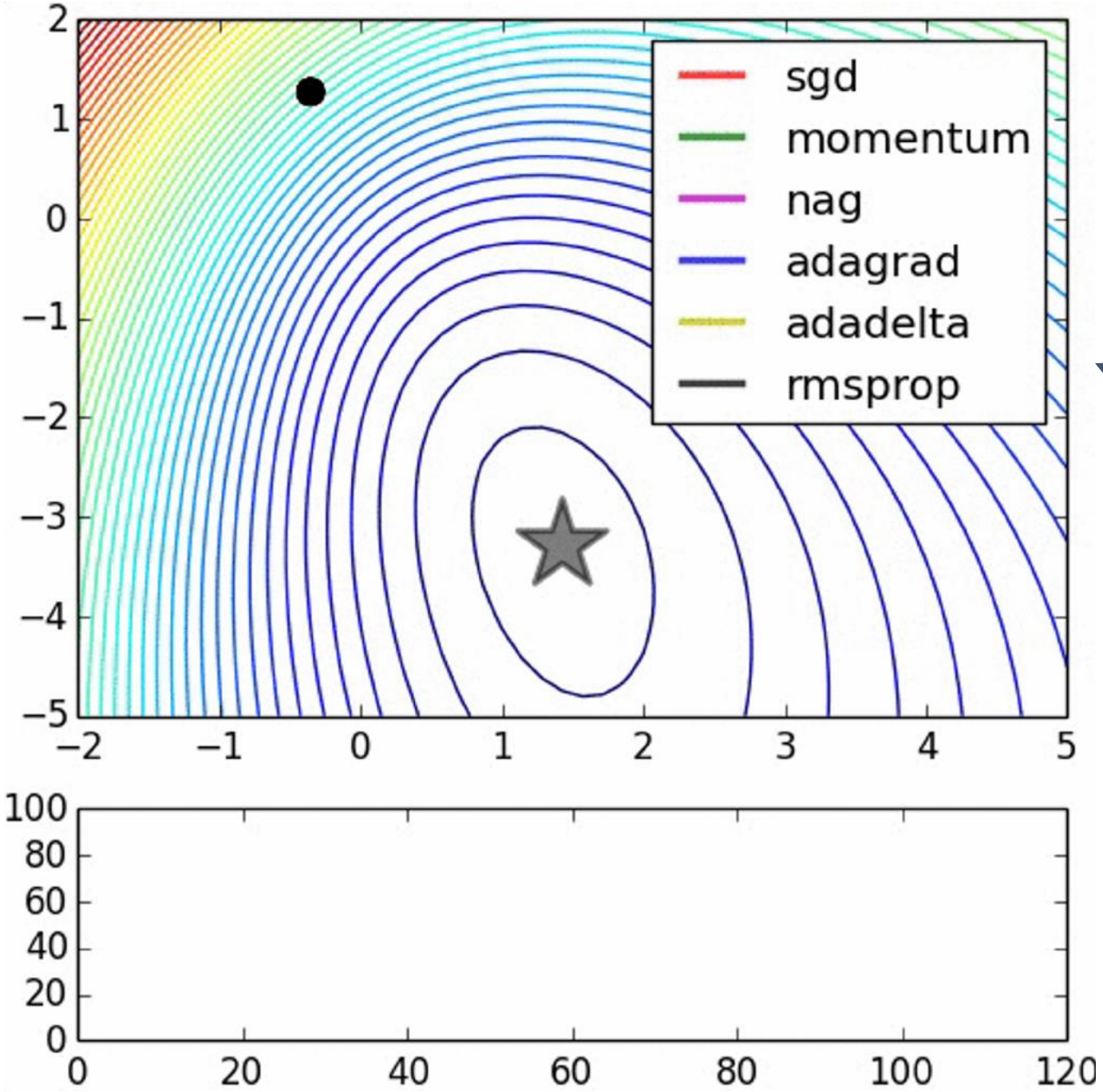
AdaGrad

```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared += dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

RMSProp

```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared = decay_rate * grad_squared + (1 - decay_rate) * dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

[Tieleman and Hinton, 2012]



adagrad
rmsprop

Adaptive Moment Estimation (Adam)
(incomplete, but close)

```
first_moment = 0
second_moment = 0
while True:
  dx = compute_gradient(x)
  first_moment = beta1 * first_moment  + (1 - beta1) * dx
  second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
  x -= learning_rate * first_moment / (np.sqrt(second_moment) + 1e-7))
```

momentum

AdaGrad / RMSProp

Looks a bit like RMSProp with momentum

[Kingma and Ba, 2014]

Adam (full form)

```
first_moment = 0
second_moment = 0
for t in range(num_iterations):
  dx = compute_gradient(x)
  first_moment = beta1 * first_moment  + (1 - beta1) * dx
  second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
  first_unbias = first_moment / (1 - beta1 ** t)
  second_unbias = second_moment / (1 - beta2 ** t)
  x -= learning_rate * first_unbias / (np.sqrt(second_unbias) + 1e-7))
```

momentum

Bias correction

AdaGrad / RMSProp

The bias correction compensates for the fact that m,v are initialized at zero and need some time to "warm up".

[Kingma and Ba, 2014]

Adam (full form)

```
first_moment = 0
second_moment = 0
for t in range(num_iterations):
  dx = compute_gradient(x)
  first_moment = beta1 * first_moment  + (1 - beta1) * dx
  second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
  first_unbias = first_moment / (1 - beta1 ** t)
  second_unbias = second_moment / (1 - beta2 ** t)
  x -= learning_rate * first_unbias / (np.sqrt(second_unbias) + 1e-7))
```

momentum

Bias correction

AdaGrad / RMSProp

The bias correction compensates for the fact that m,v are initialized at zero and need some time to "warm up".

Adam with beta1 = 0.9, beta2 = 0.999, and learning_rate = 1e-3 or 5e-4 is a great starting point for many models!

[Kingma and Ba, 2014]

Optimization Algorithm Comparison

Algorithm	Tracks first moments (Momentum)	Tracks second moments (Adaptive learning rates)	Leaky second moments	Bias correction for moment estimates
SGD	✗	✗	✗	✗
SGD+Momentum	✓	✗	✗	✗
Nesterov	✓	✗	✗	✗
AdaGrad	✗	✓	✗	✗
RMSProp	✗	✓	✓	✗
Adam	✓	✓	✓	✓

L2 Regularization vs Weight Decay

Optimization Algorithm

$$L(w) = L_{data}(w) + L_{reg}(w)$$
$$g_t = \nabla L(w_t)$$
$$s_t = optimizer(g_t)$$
$$w_{t+1} = w_t - \alpha s_t$$

L2 Regularization and Weight Decay are equivalent for SGD, SGD+Momentum so people often use the terms interchangeably!

But they are not the same for adaptive methods (AdaGrad, RMSProp, Adam, etc)

L2 Regularization

$$L(w) = L_{data}(w) + \boldsymbol{\lambda |w|^2}$$
$$g_t = \nabla L(w_t) = \nabla L_{data}(w_t) + \boldsymbol{2\lambda w_t}$$
$$s_t = optimizer(g_t)$$
$$w_{t+1} = w_t - \alpha s_t$$

Weight Decay

$$L(w) = L_{data}(w)$$
$$g_t = \nabla L_{data}(w_t)$$
$$s_t = optimizer(g_t) + \boldsymbol{2\lambda w_t}$$
$$w_{t+1} = w_t - \alpha s_t$$

[Loshchilov and Hutter, 2019]

AdamW: Decoupled Weight Decay

Algorithm 2 Adam with L_2 regularization and Adam with decoupled weight decay (AdamW)

given $\alpha = 0.001, \beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 10^{-8}, \lambda \in \mathbb{R}$
initialize time step $t \leftarrow 0$, parameter vector $\boldsymbol{\theta}_{t=0} \in \mathbb{R}^n$, first moment vector $\boldsymbol{m}_{t=0} \leftarrow \boldsymbol{0}$, second moment vector $\boldsymbol{v}_{t=0} \leftarrow \boldsymbol{0}$, schedule multiplier $\eta_{t=0} \in \mathbb{R}$
repeat
  $t \leftarrow t + 1$
  $\nabla f_t(\boldsymbol{\theta}_{t-1}) \leftarrow \text{SelectBatch}(\boldsymbol{\theta}_{t-1})$ ▷ select batch and return the corresponding gradient
  $\boldsymbol{g}_t \leftarrow \nabla f_t(\boldsymbol{\theta}_{t-1}) + \lambda \boldsymbol{\theta}_{t-1}$
  $\boldsymbol{m}_t \leftarrow \beta_1 \boldsymbol{m}_{t-1} + (1 - \beta_1) \boldsymbol{g}_t$ ▷ here and below all operations are element-wise
  $\boldsymbol{v}_t \leftarrow \beta_2 \boldsymbol{v}_{t-1} + (1 - \beta_2) \boldsymbol{g}_t^2$
  $\hat{\boldsymbol{m}}_t \leftarrow \boldsymbol{m}_t / (1 - \beta_1^t)$ ▷ β_1 is taken to the power of t
  $\hat{\boldsymbol{v}}_t \leftarrow \boldsymbol{v}_t / (1 - \beta_2^t)$ ▷ β_2 is taken to the power of t
  $\eta_t \leftarrow \text{SetScheduleMultiplier}(t)$ ▷ can be fixed, decay, or also be used for warm restarts
  $\boldsymbol{\theta}_t \leftarrow \boldsymbol{\theta}_{t-1} - \eta_t \left(\alpha \hat{\boldsymbol{m}}_t / (\sqrt{\hat{\boldsymbol{v}}_t} + \epsilon) + \lambda \boldsymbol{\theta}_{t-1} \right)$
until *stopping criterion is met*
return optimized parameters $\boldsymbol{\theta}_t$

[Loshchilov and Hutter, 2019]

AdamW: Decoupled Weight Decay

Algorithm 2 Adam with L_2 regularization and Adam with decoupled weight decay (AdamW)

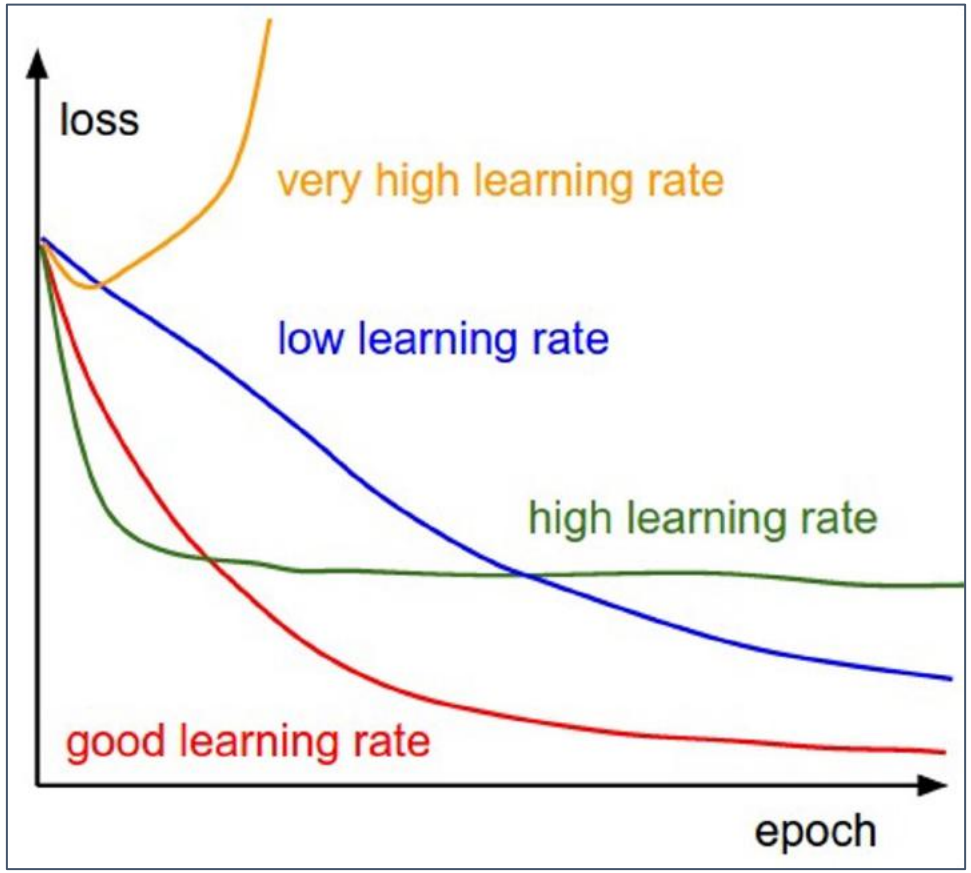
1: **given** $\alpha = 0.001, \beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 10^{-8}, \lambda \in \mathbb{R}$
2: **initialize** time step $t \leftarrow 0$, parameter vector $\boldsymbol{\theta}_{t=0} \in \mathbb{R}^n$, first moment vector $\boldsymbol{m}_{t=0} \leftarrow \boldsymbol{0}$, second moment vector $\boldsymbol{v}_{t=0} \leftarrow \boldsymbol{0}$, schedule multiplier $\eta_{t=0} \in \mathbb{R}$

AdamW should probably be your "default" optimizer for new problems

8: $\boldsymbol{v}_t \leftarrow \beta_2 \boldsymbol{v}_{t-1} + (1 - \beta_2)\boldsymbol{g}_t^2$
9: $\hat{\boldsymbol{m}}_t \leftarrow \boldsymbol{m}_t / (1 - \beta_1^t)$ ▷ β_1 is taken to the power of t
10: $\hat{\boldsymbol{v}}_t \leftarrow \boldsymbol{v}_t / (1 - \beta_2^t)$ ▷ β_2 is taken to the power of t
11: $\eta_t \leftarrow \text{SetScheduleMultiplier}(t)$ ▷ can be fixed, decay, or also be used for warm restarts
12: $\boldsymbol{\theta}_t \leftarrow \boldsymbol{\theta}_{t-1} - \eta_t \left(\alpha \hat{\boldsymbol{m}}_t / (\sqrt{\hat{\boldsymbol{v}}_t} + \epsilon) + \lambda \boldsymbol{\theta}_{t-1} \right)$
13: **until** *stopping criterion is met*
14: **return** optimized parameters $\boldsymbol{\theta}_t$

[Loshchilov and Hutter, 2019]

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



=> Learning rate decay over time!

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

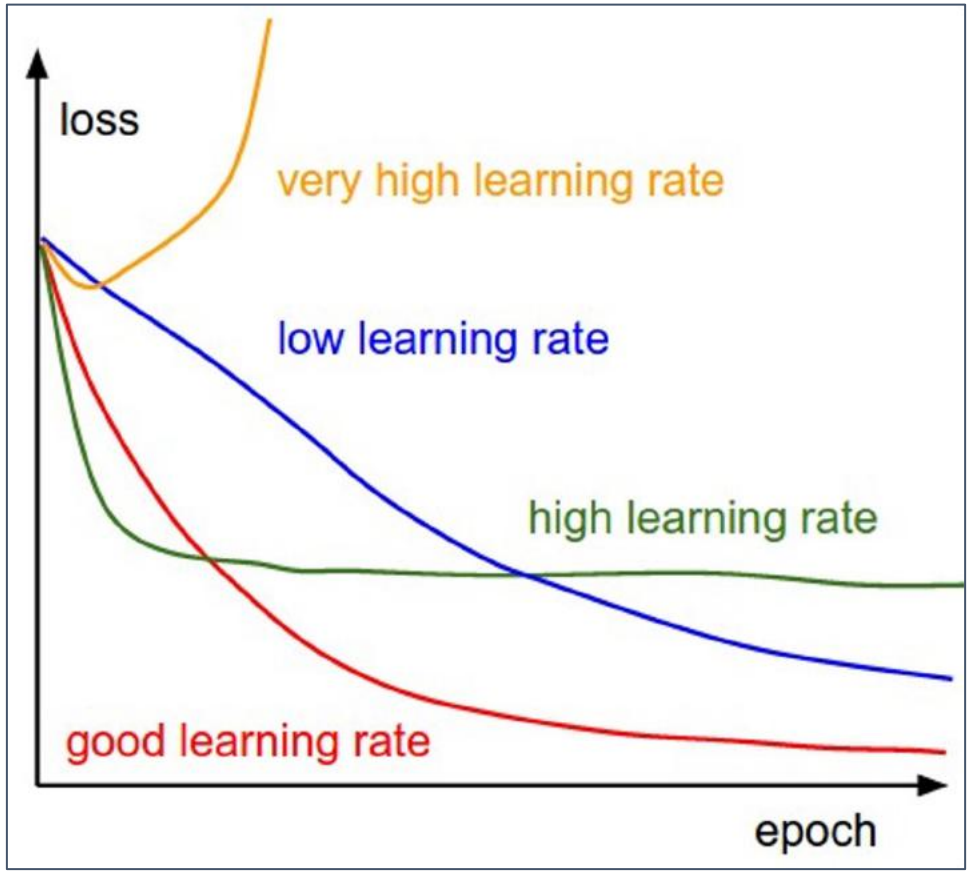
exponential decay:

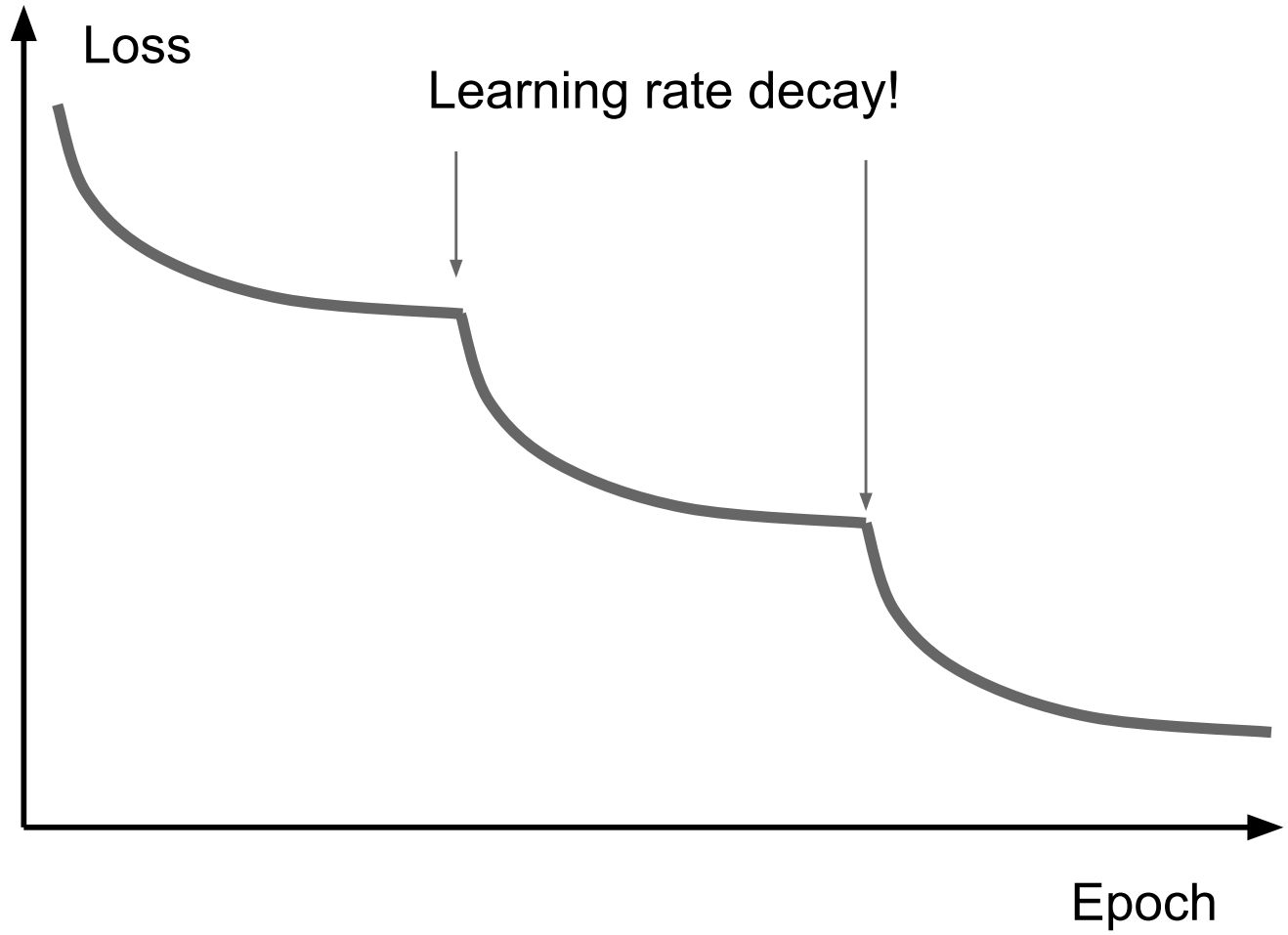
$$\alpha = \alpha_0 e^{-kt}$$

1/t decay:

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

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





Take Home Messages

Optimization Tricks

- SGD with momentum, batch-normalization, and dropout usually works very well
- Pick learning rate by running on a subset of the data
  - Start with large learning rate & divide by 2 until loss does not diverge
  - Decay learning rate by a factor of ~100 or more by the end of training
- Use ReLU nonlinearity
- Initialize parameters so that each feature across layers has similar variance. Avoid units in saturation.

From Marc'Aurelio Ranzato, CVPR 2014 tutorial

Ways To Improve Generalization

- Weight sharing (greatly reduce the number of parameters)
- Dropout
- Weight decay (L2, L1)
- Sparsity in the hidden units

From Marc'Aurelio Ranzato, CVPR 2014 tutorial

Next lecture:
Convolutional
Neural Networks