CS 4644-DL / 7643-A: LECTURE 10
DANFEI XU

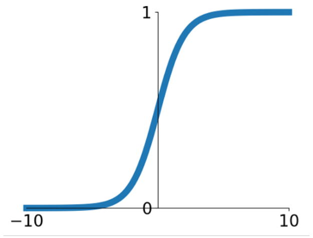
Topics:

- Training Neural Networks (Part 2)

Activation Functions

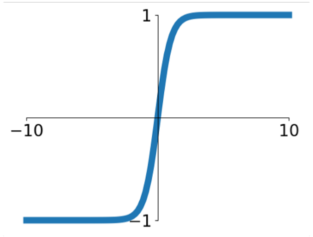
Sigmoid

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



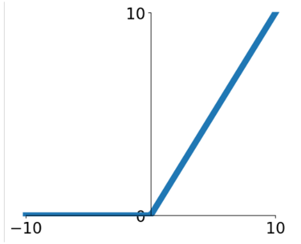
tanh

$$\tanh(x)$$



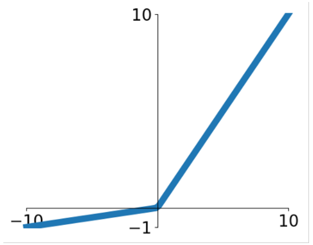
ReLU

$$\max(0, x)$$



Leaky ReLU

$$\max(0.1x, x)$$

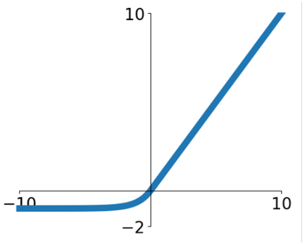


Maxout

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

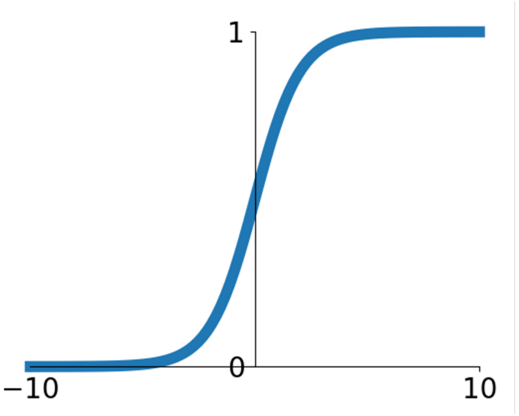
ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



Activation Functions

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



Sigmoid

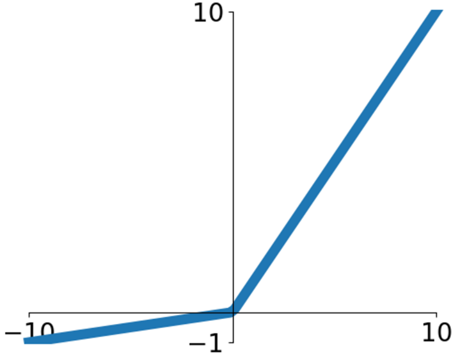
- Squashes numbers to range [0,1]
- Historically popular since they have nice interpretation as a saturating “firing rate” of a neuron

2 problems:

1. **Saturated neurons “kill” the gradients**
2. exp() is a bit compute expensive

Activation Functions

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



- 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)

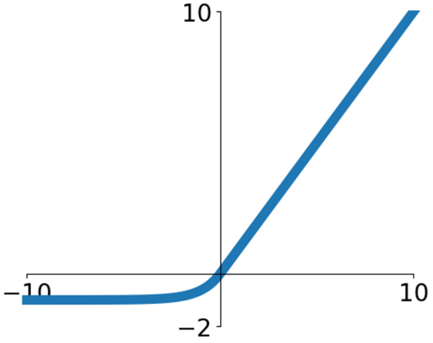
$$f(x) = \max(\alpha x, x)$$

backprop into \alpha
(parameter)

Activation Functions

[Clevert et al., 2015]

Exponential Linear Units (ELU)



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

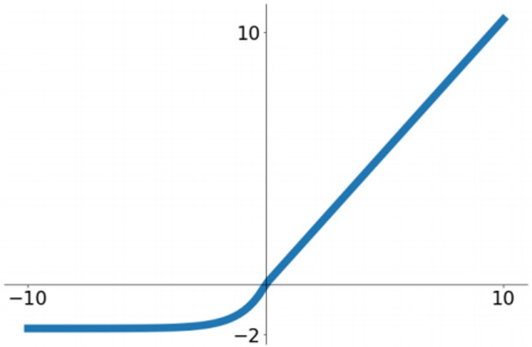
(Alpha default = 1)

- All benefits of ReLU
- Negative saturation encodes presence of features (all goes to -\alpha), not magnitude
- Same in backprop
- Compared with Leaky ReLU: more robust to noise

Activation Functions

[Klambauer et al. ICLR 2017]

Scaled Exponential Linear Units (SELU)



$$f(x) = \begin{cases} \lambda x & \text{if } x > 0 \\ \lambda\alpha(e^x - 1) & \text{otherwise} \end{cases}$$

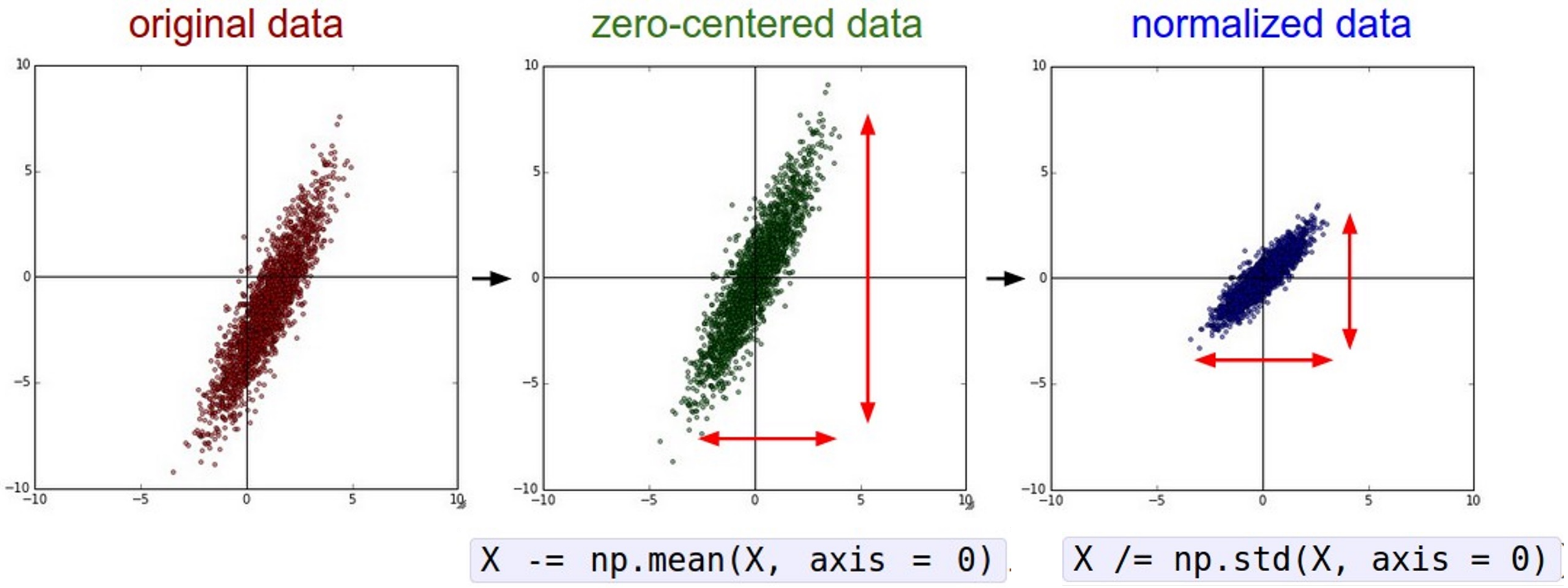
α = 1.6732632423543772848170429916717
λ = 1.0507009873554804934193349852946

- Scaled versionof ELU that works better for deep networks
- “Self-normalizing” property;
- Can train deep SELU networks without BatchNorm
  - (will discuss more later)

Derivation takes 91 pages of math in appendix…

(Klambauer et al, Self-Normalizing Neural Networks, ICLR 2017)

Data Preprocessing



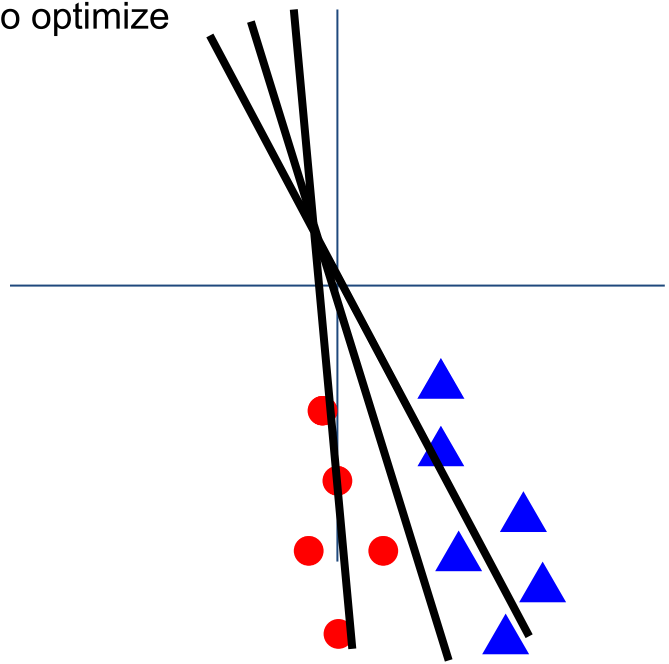
```
X -= np.mean(X, axis = 0)
```

```
X /= np.std(X, axis = 0)
```

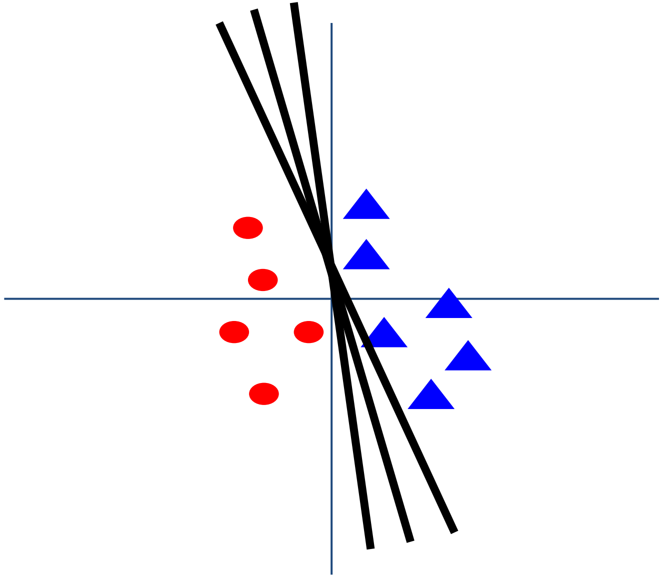
(Assume X [NxD] is data matrix,
each example in a row)

Data Preprocessing

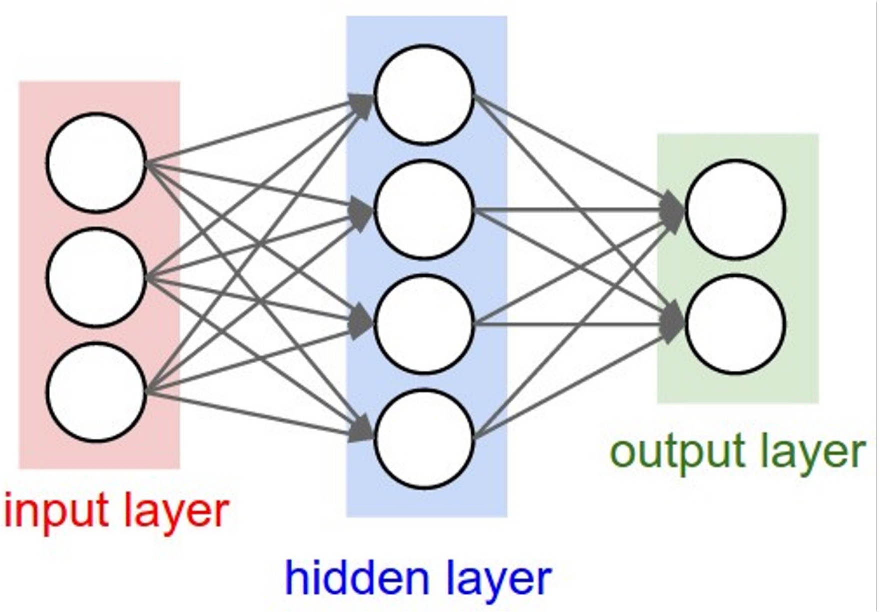
Before normalization: classification loss very sensitive to changes in weight matrix; hard to optimize



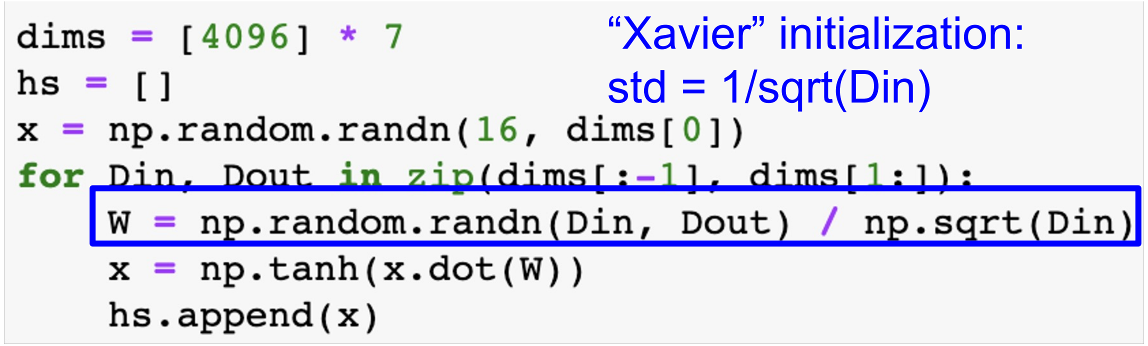
After normalization: less sensitive to small changes in weights; easier to optimize



Weight initialization: goal is to **maintain both diversity and variance** of layer output throughout the network, at least at the beginning of the training



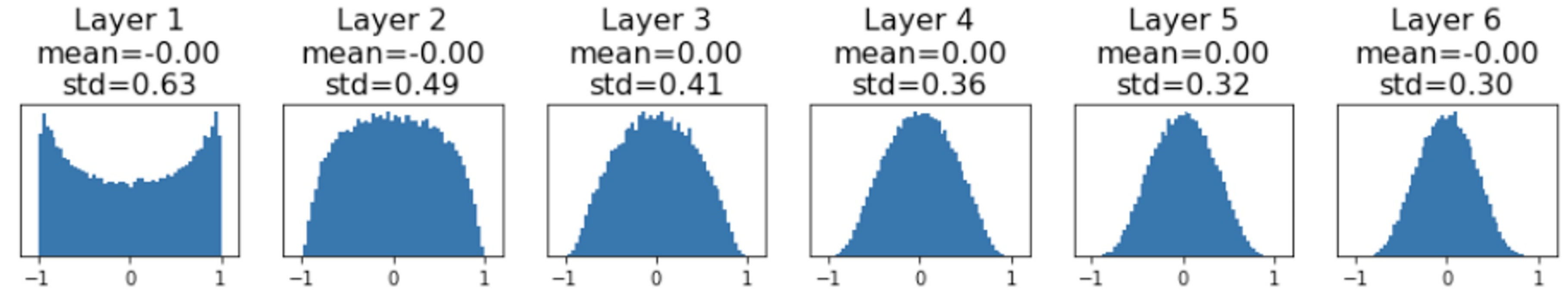
Weight Initialization: “Xavier” Initialization



```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = np.random.randn(Din, Dout) / np.sqrt(Din)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

“Xavier” initialization:
std = 1/sqrt(Din)

“Just right”: Activations are nicely scaled for all layers!



Glorot and Bengio, “Understanding the difficulty of training deep feedforward neural networks”, AISTAT 2010

Visualize distribution of activations

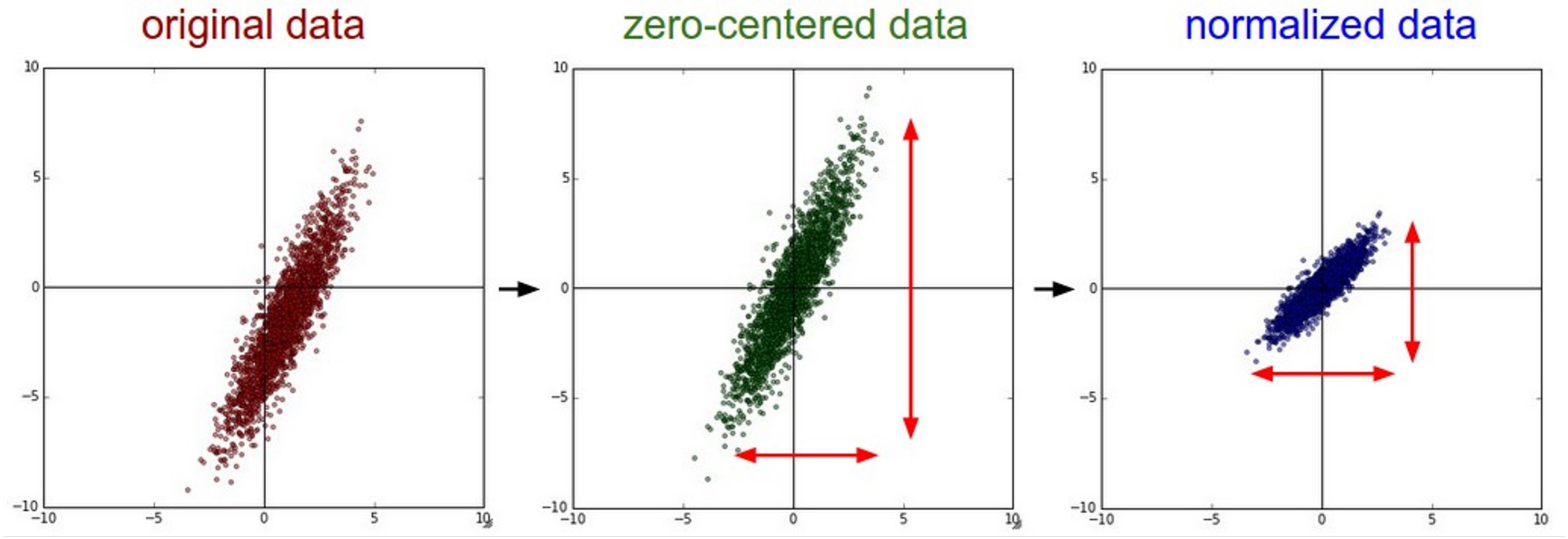
This Time:

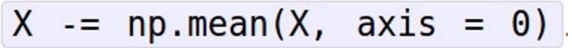
Training Deep Neural Networks

- Details of the non-linear activation functions
- Data normalization
- Weight Initialization
- Batch Normalization
- Advanced Optimization
- Regularization
- Data Augmentation
- Transfer learning
- Hyperparameter Tuning
- Model Ensemble

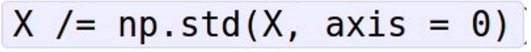
Batch Normalization

Recall: Input Normalization



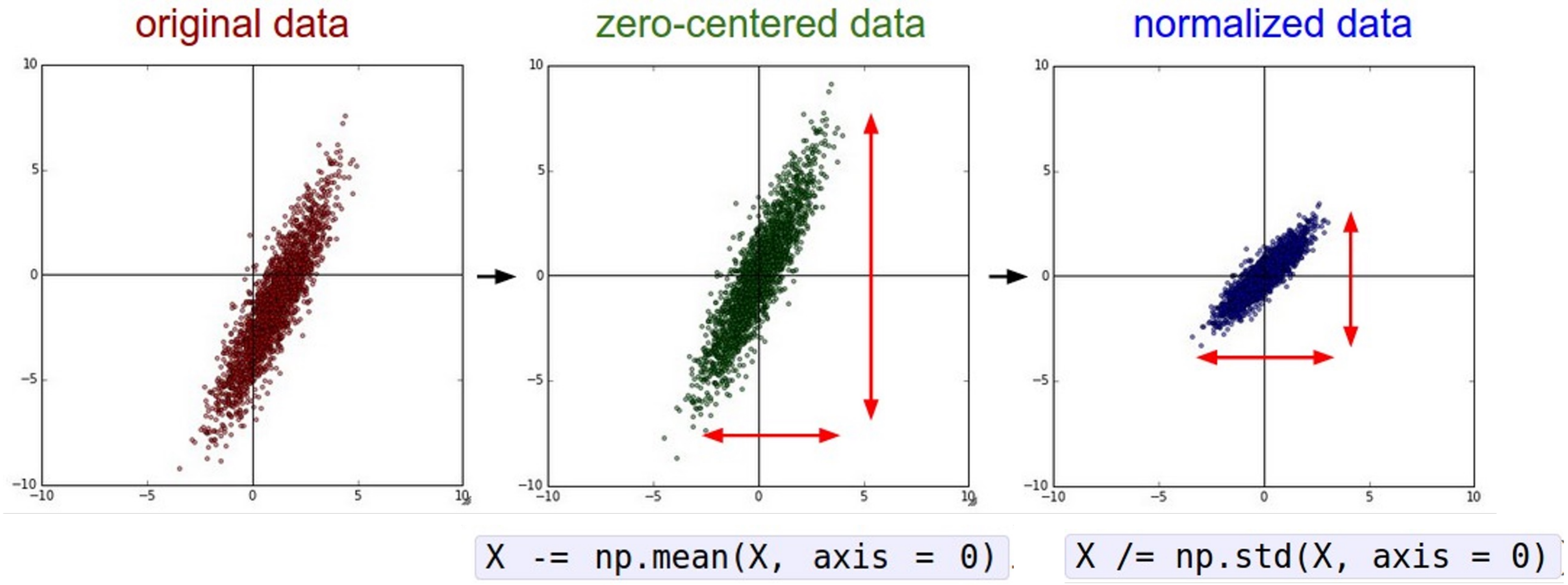


```
X -= np.mean(X, axis = 0)
```



```
X /= np.std(X, axis = 0)
```

Recall: Input Normalization



```
X -= np.mean(X, axis = 0)
```

```
X /= np.std(X, axis = 0)
```

Problem: Only for input to the first layer. Input for later layers are longer normalized!

But can’t do dataset normalization for intermediate layers! Activation distribution changes as the training progresses.

Batch Normalization

[Ioffe and Szegedy, 2015]

“you want zero-mean unit-variance activations? just make them so.”

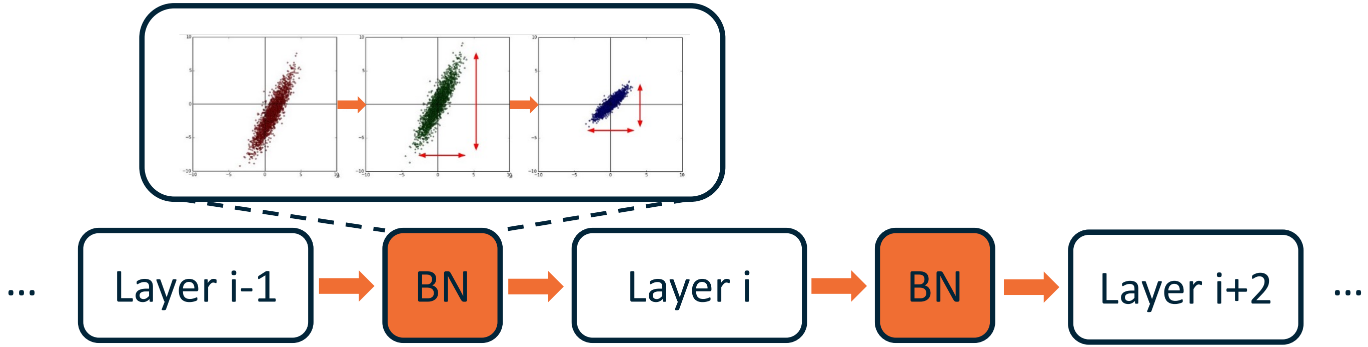
consider a **batch of activations** $\boldsymbol{x}$ at some layer. To make each dimension zero-mean unit-variance, apply:

$$\hat{x} = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x]}}$$

this is a vanilla differentiable function...

Batch Normalization

“you want zero-mean unit-variance activations? just make them so.”



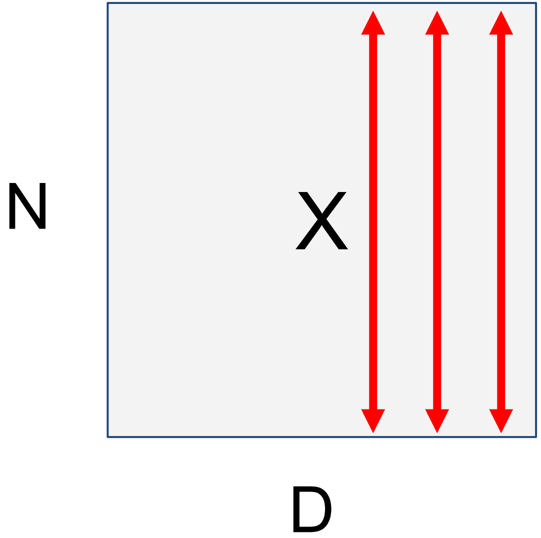
… Layer i-1 → BN → Layer i → BN → Layer i+2 …

$$\hat{x} = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x]}}$$

Batch Normalization

[Ioffe and Szegedy, 2015]

Input: $x : N \times D$



$$\mu_j = \frac{1}{N} \sum_{i=1}^{N} x_{i,j}$$

Per-batch mean, shape is D

$$\sigma_j^2 = \frac{1}{N} \sum_{i=1}^{N} (x_{i,j} - \mu_j)^2$$

Per-batch var, shape is D

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}}$$

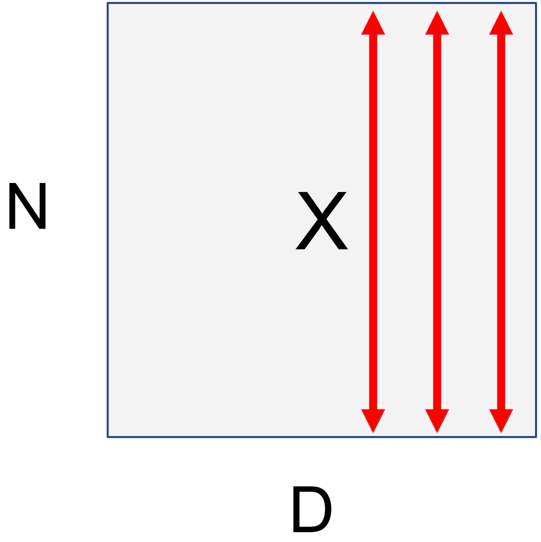
Normalized x, Shape is N x D

(Prevent div by 0 err)

Batch Normalization

[Ioffe and Szegedy, 2015]

Input: $x : N \times D$



$$\mu_j = \frac{1}{N}\sum_{i=1}^{N} x_{i,j}$$

Per-batch mean, shape is D

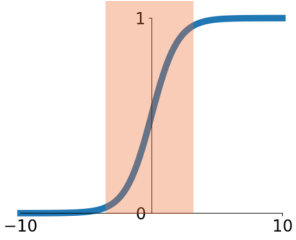
$$\sigma_j^2 = \frac{1}{N}\sum_{i=1}^{N} (x_{i,j} - \mu_j)^2$$

Per-batch var, shape is D

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}}$$

Normalized x, Shape is N x D

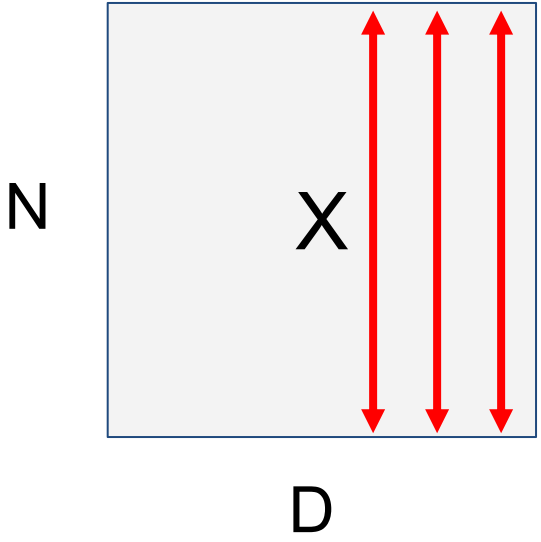
Problem: What if zero-mean, unit variance is too hard of a constraint? E.g., inserting a BN before sigmoid will constrain it to (mostly) linear regime



Batch Normalization

[Ioffe and Szegedy, 2015]

Input: $x : N \times D$



$$\mu_j = \frac{1}{N}\sum_{i=1}^{N} x_{i,j}$$

Per-batch mean, shape is D

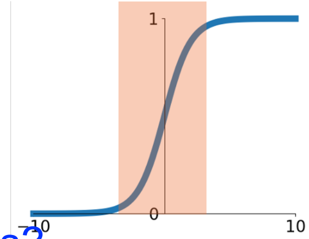
$$\sigma_j^2 = \frac{1}{N}\sum_{i=1}^{N}(x_{i,j} - \mu_j)^2$$

Per-batch var, shape is D

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}}$$

Normalized x, Shape is N x D

Problem: What if zero-mean, unit variance is too hard of a constraint? E.g., inserting a BN before sigmoid will constrain it to (mostly) linear regime



Can we learn the normalization parameters?

Batch Normalization

[Ioffe and Szegedy, 2015]

Input: $x : N \times D$

Learnable scale and shift parameters:

γ, β: $\mathbb{R}^D$

We want to give the model a chance to **adjust batchnorm** if the default is not optimal.

Learning $\gamma = \sigma$ and $\beta = \mu$ will recover the original input batch!

$$\mu_j = \frac{1}{N}\sum_{i=1}^{N} x_{i,j}$$

Per-batch mean, shape is D

$$\sigma_j^2 = \frac{1}{N}\sum_{i=1}^{N}(x_{i,j} - \mu_j)^2$$

Per-batch var, shape is D

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}}$$

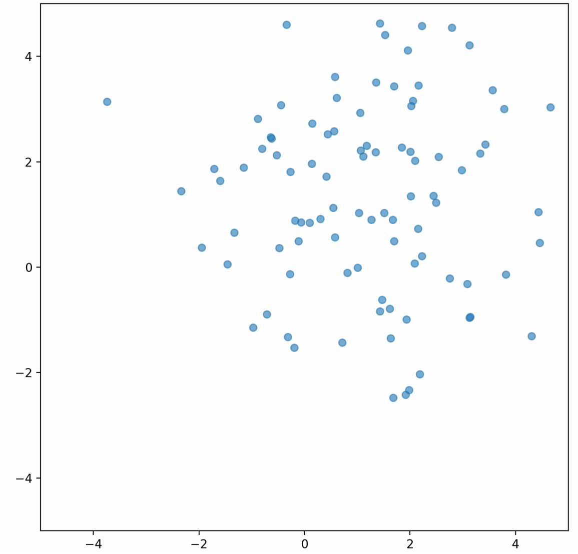
Normalized x, Shape is N x D

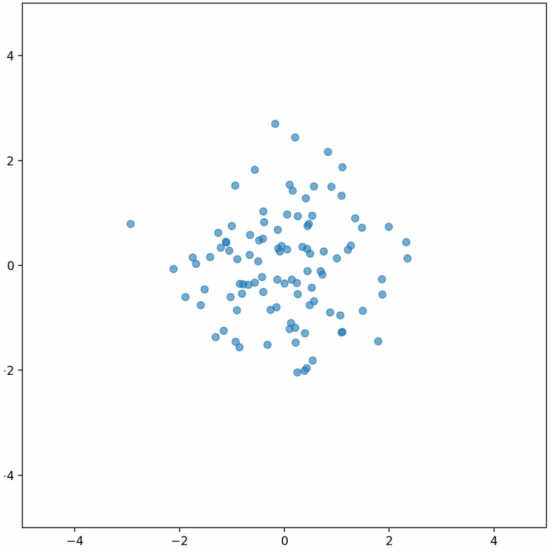
$$y_{i,j} = \underline{\gamma_j}\hat{x}_{i,j} + \underline{\beta_j}$$

Output, Shape is N x D

Initialize $\gamma = 1, \beta = 0$

What does it look like?





$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}}$$

$$y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j$$

$\gamma = [2, 1.5], \beta = [1, -1]$

Batch Normalization: Test-Time

Estimates depend on minibatch; can't do this at test-time!

Input: $x : N \times D$

Learnable scale and shift parameters:

γ, β: $\mathbb{R}^D$

$$\mu_j = \frac{1}{N}\sum_{i=1}^{N} x_{i,j}$$

Per-batch mean, shape is D

$$\sigma_j^2 = \frac{1}{N}\sum_{i=1}^{N}(x_{i,j} - \mu_j)^2$$

Per-batch var, shape is D

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}}$$

Normalized x, Shape is N x D

$$y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j$$

Output, Shape is N x D

Batch Normalization: Test-Time

Estimates depend on minibatch; can't do this at test-time!

Input: $x : N \times D$

Learnable scale and shift parameters:

γ, β: $\mathbb{R}^D$

Activations become fixed after training. Can calculate training set-wide statistics for inference-time normalization.

At training time, do moving average to save compute.

$$\mu_j = \frac{1}{N}\sum_{i=1}^{N} x_{i,j}$$

Per-batch mean, shape is D

$$\sigma_j^2 = \frac{1}{N}\sum_{i=1}^{N}(x_{i,j} - \mu_j)^2$$

Per-batch var, shape is D

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}}$$

Normalized x, Shape is N x D

$$y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j$$

Output, Shape is N x D

Batch Normalization: Test-Time

Input: $x : N \times D$

Learnable scale and shift parameters:

γ, β: $\mathbb{R}^D$

$\mu_j =$ (Moving) average of values seen during training

Per-batch mean, shape is D

$\sigma_j^2 =$ (Moving) average of values seen during training

Per-batch var, shape is D

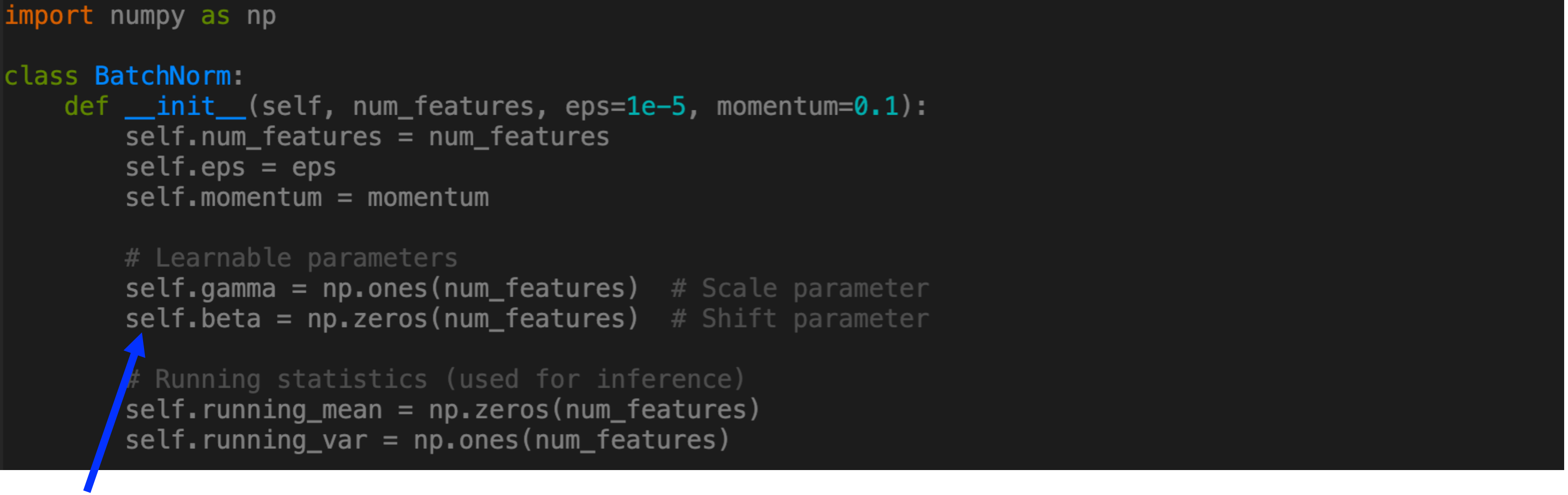
$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}}$$

Normalized x, Shape is N x D

$$y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j$$

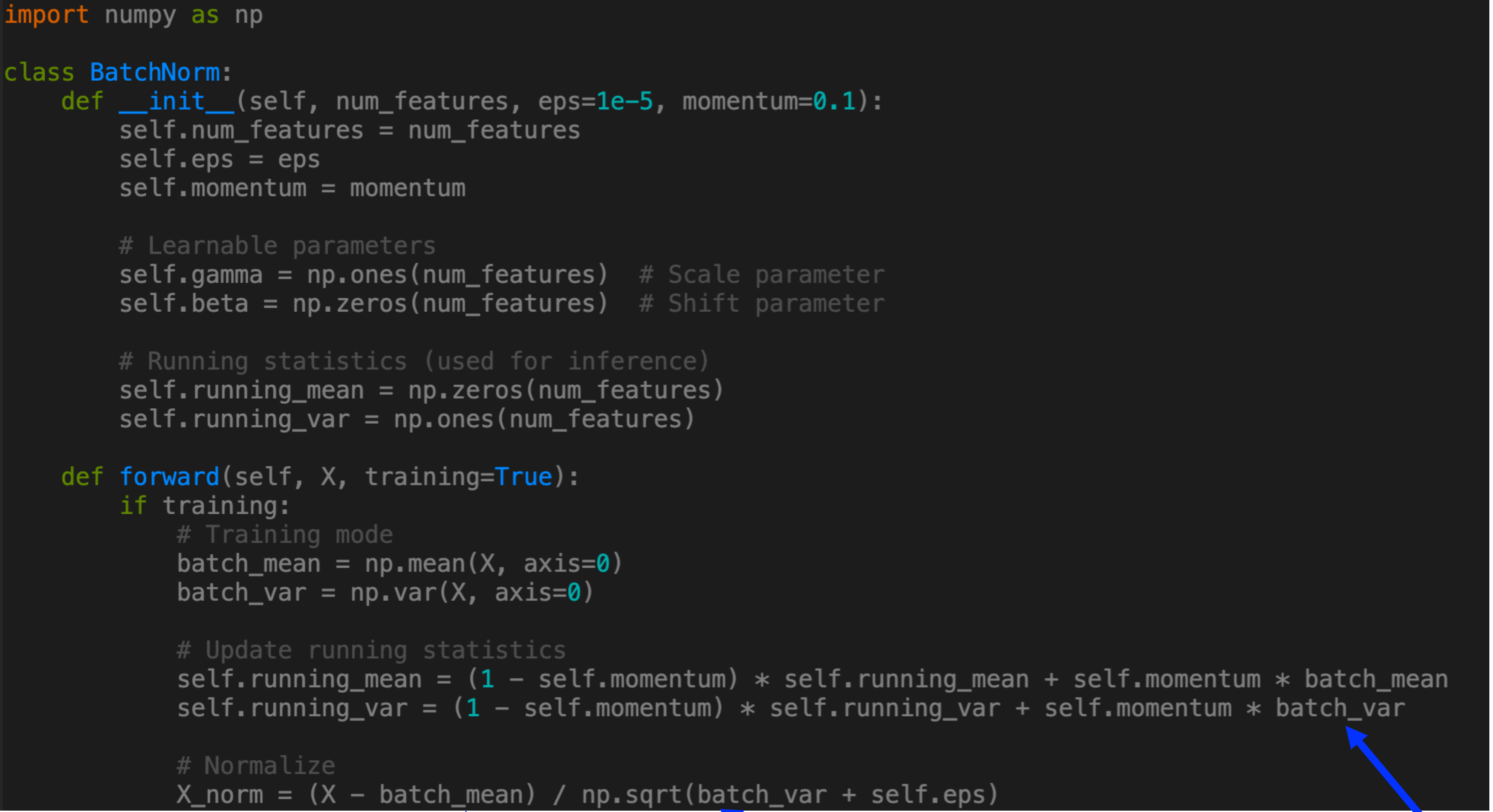
Output, Shape is N x D

During testing batchnorm becomes a linear operator! Can be fused with the previous fully-connected or conv layer



```python
import numpy as np

class BatchNorm:
    def __init__(self, num_features, eps=1e-5, momentum=0.1):
        self.num_features = num_features
        self.eps = eps
        self.momentum = momentum

        # Learnable parameters
        self.gamma = np.ones(num_features)  # Scale parameter
        self.beta = np.zeros(num_features)  # Shift parameter

        # Running statistics (used for inference)
        self.running_mean = np.zeros(num_features)
        self.running_var = np.ones(num_features)
```

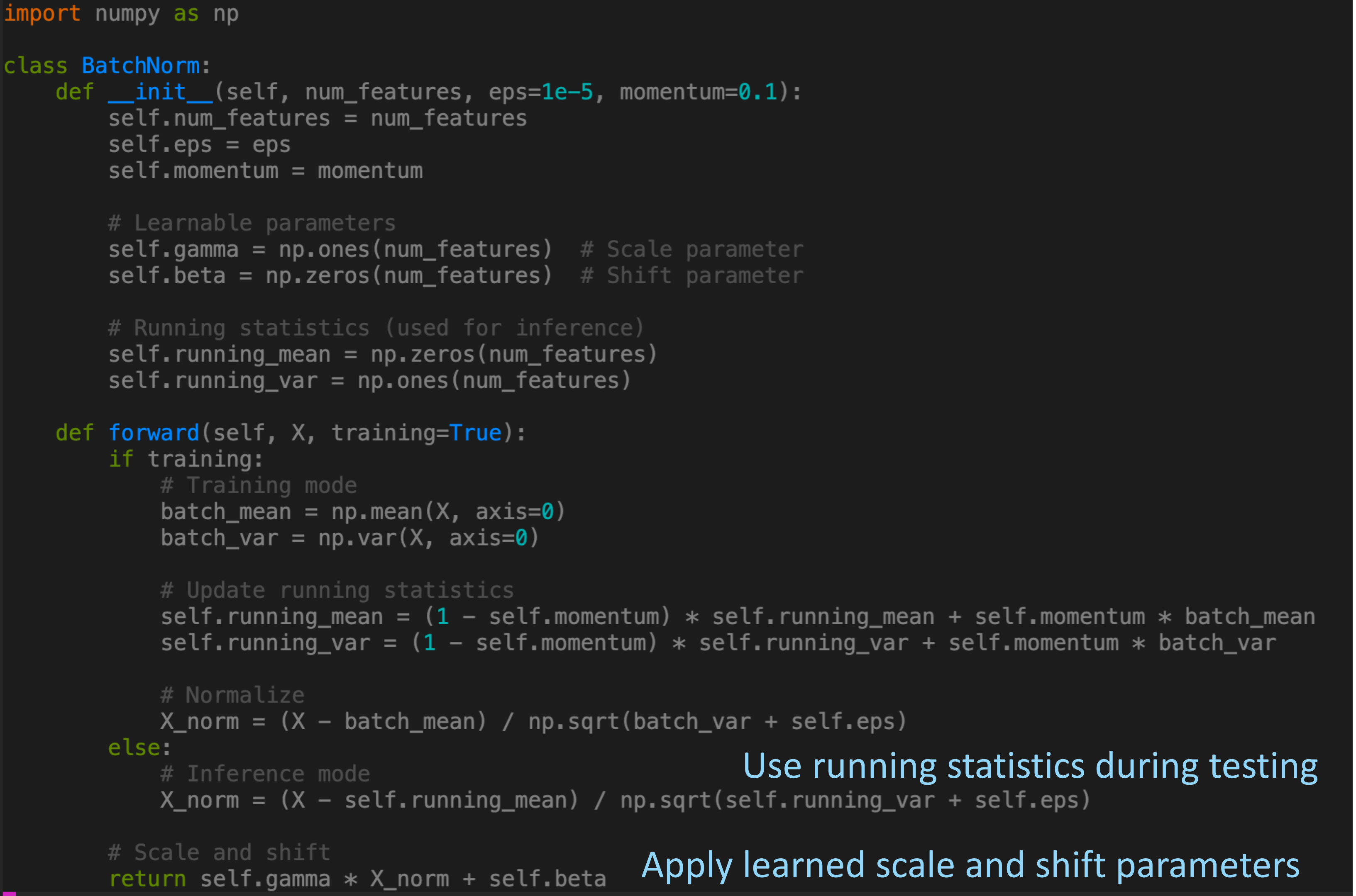
You can think of gamma and beta as the layer parameters



```
import numpy as np

class BatchNorm:
    def __init__(self, num_features, eps=1e-5, momentum=0.1):
        self.num_features = num_features
        self.eps = eps
        self.momentum = momentum

        # Learnable parameters
        self.gamma = np.ones(num_features)  # Scale parameter
        self.beta = np.zeros(num_features)  # Shift parameter

        # Running statistics (used for inference)
        self.running_mean = np.zeros(num_features)
        self.running_var = np.ones(num_features)

    def forward(self, X, training=True):
        if training:
            # Training mode
            batch_mean = np.mean(X, axis=0)
            batch_var = np.var(X, axis=0)

            # Update running statistics
            self.running_mean = (1 - self.momentum) * self.running_mean + self.momentum * batch_mean
            self.running_var = (1 - self.momentum) * self.running_var + self.momentum * batch_var

            # Normalize
            X_norm = (X - batch_mean) / np.sqrt(batch_var + self.eps)
```

Use batch statistics during training

Keep running dataset statistics



```python
import numpy as np

class BatchNorm:
    def __init__(self, num_features, eps=1e-5, momentum=0.1):
        self.num_features = num_features
        self.eps = eps
        self.momentum = momentum

        # Learnable parameters
        self.gamma = np.ones(num_features)  # Scale parameter
        self.beta = np.zeros(num_features)  # Shift parameter

        # Running statistics (used for inference)
        self.running_mean = np.zeros(num_features)
        self.running_var = np.ones(num_features)

    def forward(self, X, training=True):
        if training:
            # Training mode
            batch_mean = np.mean(X, axis=0)
            batch_var = np.var(X, axis=0)

            # Update running statistics
            self.running_mean = (1 - self.momentum) * self.running_mean + self.momentum * batch_mean
            self.running_var = (1 - self.momentum) * self.running_var + self.momentum * batch_var

            # Normalize
            X_norm = (X - batch_mean) / np.sqrt(batch_var + self.eps)
        else:
            # Inference mode
            X_norm = (X - self.running_mean) / np.sqrt(self.running_var + self.eps)

        # Scale and shift
        return self.gamma * X_norm + self.beta
```

Use running statistics during testing

Apply learned scale and shift parameters

Batch Normalization

[Ioffe and Szegedy, 2015]

Q: Should you put batchnorm before or after ReLU?
A: Topic of debate. Original paper says BN->ReLU. Now most commonly ReLU->BN. If BN-> ReLU and zero mean, ReLU kills half of the activations, but in practice makes insignificant differences.

Q: Should you normalize the **input** (e.g., images) with batchnorm?
A: No, you already have the fixed & correct dataset statistics, no need to do batchnorm.

Q: How many parameters does a batchnorm layer have?
A: Input dimension * 4: beta, gamma, moving average mu, moving average sigma. Only beta and gamma are trainable parameters.

Batch Normalization

[Ioffe and Szegedy, 2015]

- Makes deep networks **much** easier to train!
  - If you are interested in the theory, read https://arxiv.org/abs/1805.11604
  - TL;DR: makes optimization landscape smoother
- Allows higher learning rates, faster convergence
- More useful in deeper networks
- Networks become more robust to initialization
- More robust to range of input
- Zero overhead at test-time: can be fused with conv!
- Behaves differently during training and testing: this is a very common source of bugs!
- Needs large batch size to calculate accurate stats

Batch Normalization for ConvNets

Batch Normalization for **fully-connected** networks

$\mathbf{x}$: N × D

Normalize ↓

μ, σ: 1 × D

γ, β: 1 × D

$$y = \gamma(x-\mu)/\sigma+\beta$$

Batch Normalization for **convolutional** networks (Spatial Batchnorm, BatchNorm2D)

$\mathbf{x}$: N×C×H×W

Normalize ↓ ↓ ↓

μ, σ: 1×C×1×1

γ, β: 1×C×1×1

$$y = \gamma(x-\mu)/\sigma+\beta$$

Keep the spatial equivariance property of conv: all locations should be normalized in similar ways

Layer Normalization

Batch Normalization for fully-connected networks

$$x: N \times D$$

Normalize ↓

$$\mu, \sigma: 1 \times D$$

$$\gamma, \beta: 1 \times D$$

$$y = \gamma(x-\mu)/\sigma+\beta$$

Ba, Kiros, and Hinton, “Layer Normalization”, arXiv 2016

Layer Normalization for fully-connected networks

Same behavior at train and test!

$$x: N \times D$$

Normalize ↓

$$\mu, \sigma: N \times 1$$

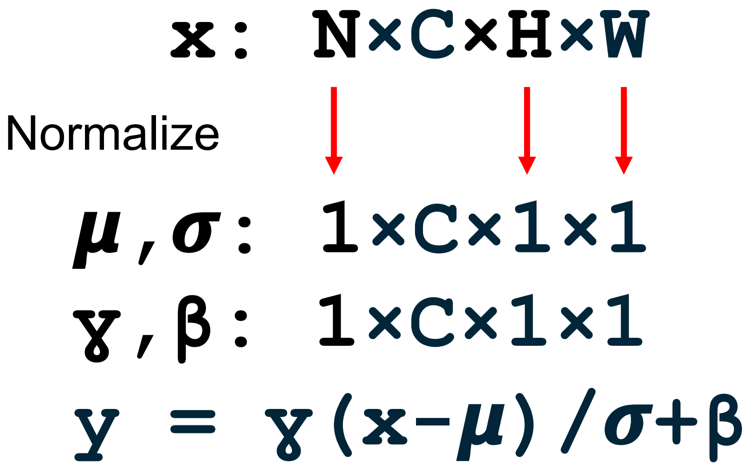
$$\gamma, \beta: 1 \times D$$

$$y = \gamma(x-\mu)/\sigma+\beta$$

More flexible (can use N = 1!), works well with sequence models (RNN, Transformers)

Instance Normalization

Batch Normalization for convolutional networks



$\mathbf{x}: N \times C \times H \times W$

Normalize

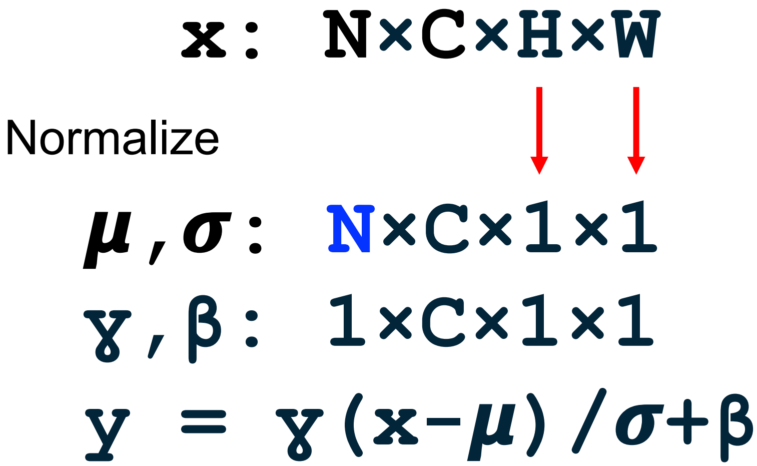
$\mu, \sigma: 1 \times C \times 1 \times 1$

$\gamma, \beta: 1 \times C \times 1 \times 1$

$y = \gamma(x - \mu)/\sigma + \beta$

Instance Normalization for convolutional networks

Same behavior at train / test!



$\mathbf{x}: N \times C \times H \times W$

Normalize

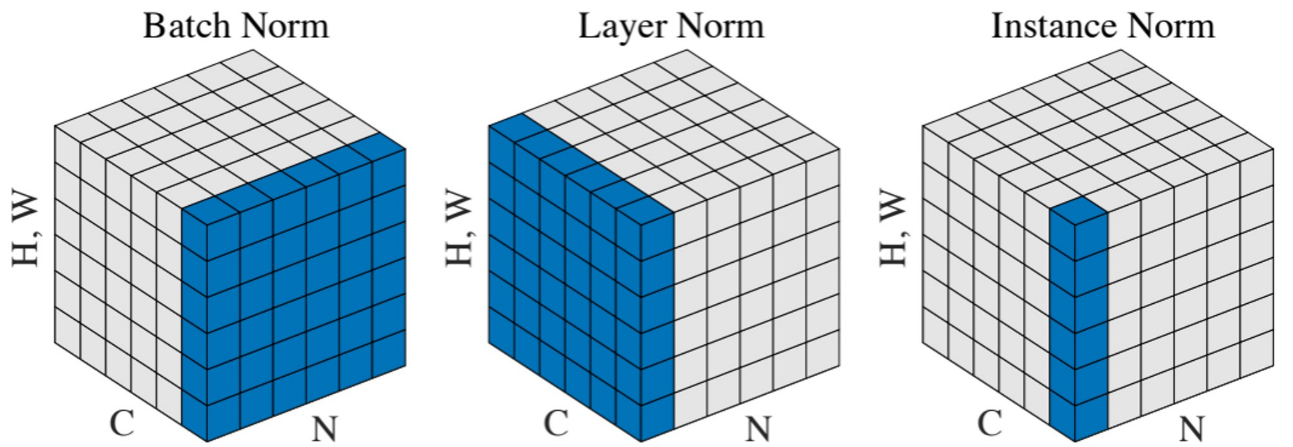
$\mu, \sigma: N \times C \times 1 \times 1$

$\gamma, \beta: 1 \times C \times 1 \times 1$

$y = \gamma(x - \mu)/\sigma + \beta$

Ulyanov et al, Improved Texture Networks: Maximizing Quality and Diversity in Feed-forward Stylization and Texture Synthesis, CVPR 2017

Comparison of Normalization Layers



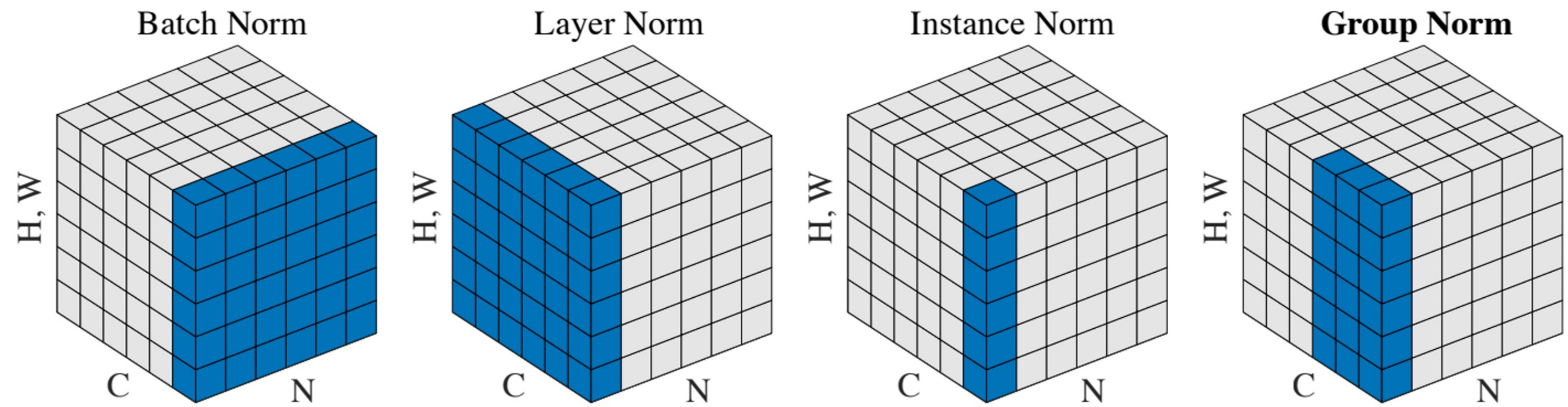
N x C x H x W -> 1 x C x 1 x 1

N x C x H x W -> N x 1 x 1 x 1

N x C x H x W -> N x C x 1 x 1

Wu and He, “Group Normalization”, ECCV 2018

Group Normalization



N x C x H x W -> 1 x C x 1 x 1

N x C x H x W -> N x 1 x 1 x 1

N x C x H x W -> N x C x 1 x 1

N x C x H x W -> N x C/G x 1 x 1

Wu and He, "Group Normalization", ECCV 2018

(Fancier) Optimizers

Optimization: (Stochastic) Gradient Descent

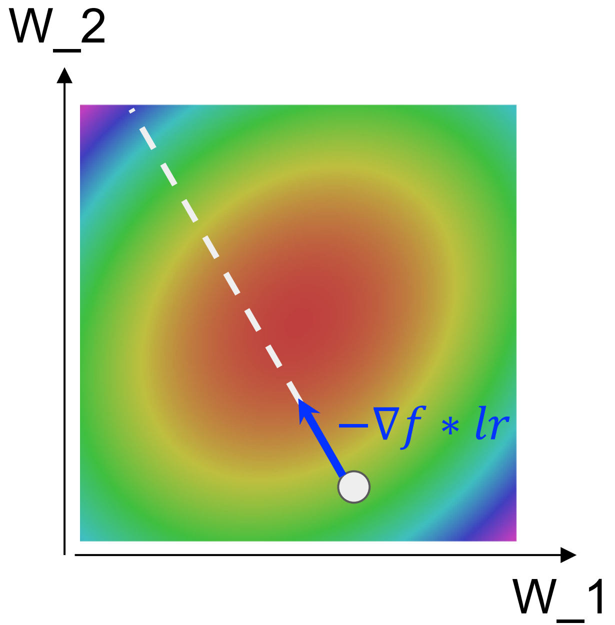
(Batch) Gradient Descent

```
While True:
     loss = model.compute_loss(dataset)
     loss.backward()
     model.weights -= model.weights.grad * lr
```

Dataset may be really large (millions of images)!

Minibatch (Stochastic) Gradient Descent

```
While True:
     minibatch = sample(dataset, batch_size)
     loss = model.compute_loss(minibatch)
     loss.backward()
     model.weights -= model.weights.grad * lr
```



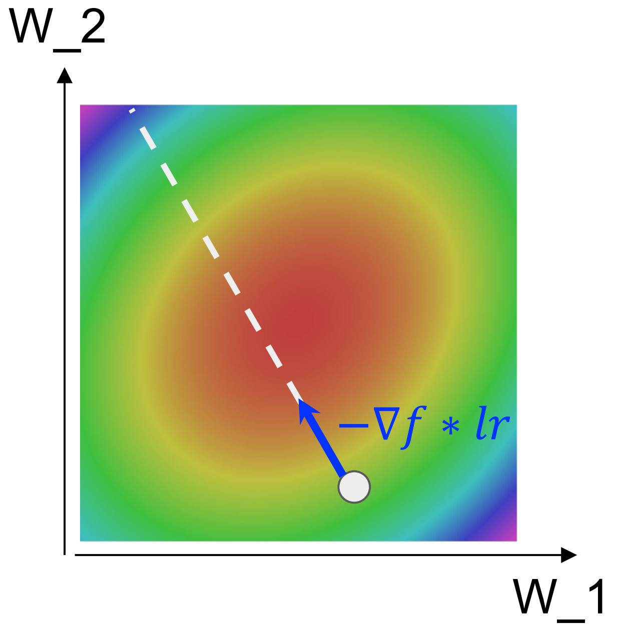
Optimization: (Stochastic) Gradient Descent

Minibatch (Stochastic) Gradient Descent

```
While True:
     minibatch = sample(dataset, batch_size)
     loss = model.compute_loss(minibatch)
     loss.backward()
     model.weights -= model.weights.grad * lr
```

Reasons to prefer SGD over GD for Deep Learning:

- More computationally-tractable
- GD doesn’t guarantee optimality for non-convex functions anyways



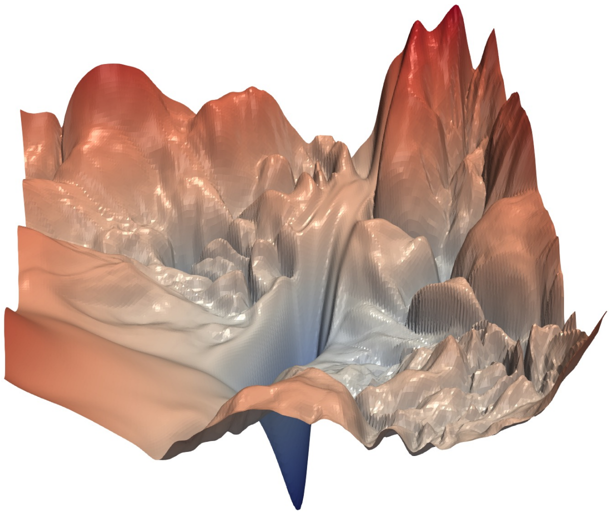
Optimization: (Stochastic) Gradient Descent

Minibatch (Stochastic) Gradient Descent

```
While True:
    minibatch = sample(dataset, batch_size)
    loss = model.compute_loss(minibatch)
    loss.backward()
    model.weights -= model.weights.grad * lr
```

Reasons to prefer SGD over GD for Deep Learning:

- More computationally-tractable
- GD doesn’t guarantee optimality for non-convex functions anyways
- SGD usually has faster convergence in wall-clock time, even if you can run GD

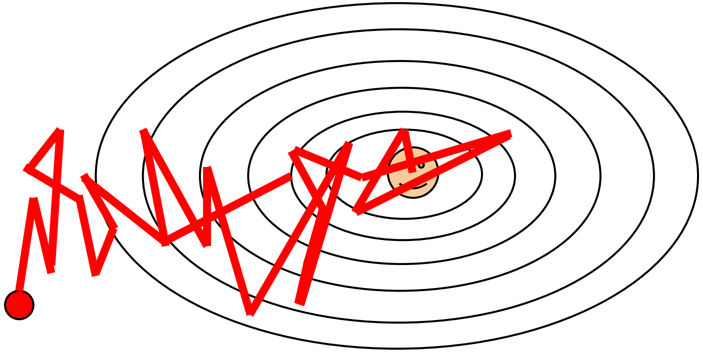


Loss landscape for DNN

https://www.cs.umd.edu/~tomg/projects/landscapes/

Optimization: Problem #1 with SGD

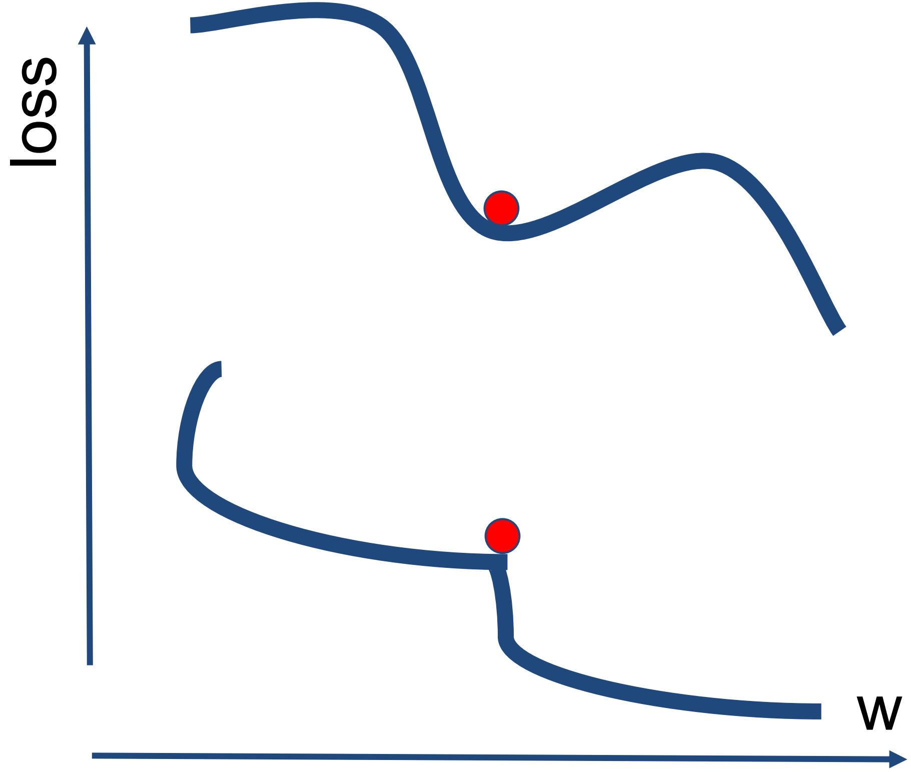
- Stochastic minibatch gives a **noisy estimate of the true gradient direction**. Very problematic when the batch size is small (e.g., due to compute resource limit).
- Large batch size helps, but doesn't solve the problem entirely in non-convex settings
- Poorly-selected learning rate makes the oscillation worse (overshoot)



http://web.cs.ucla.edu/~chohsieh/teaching/CS260_Winter2019/lecture4.pdf

Optimization: Problem #2 with SGD

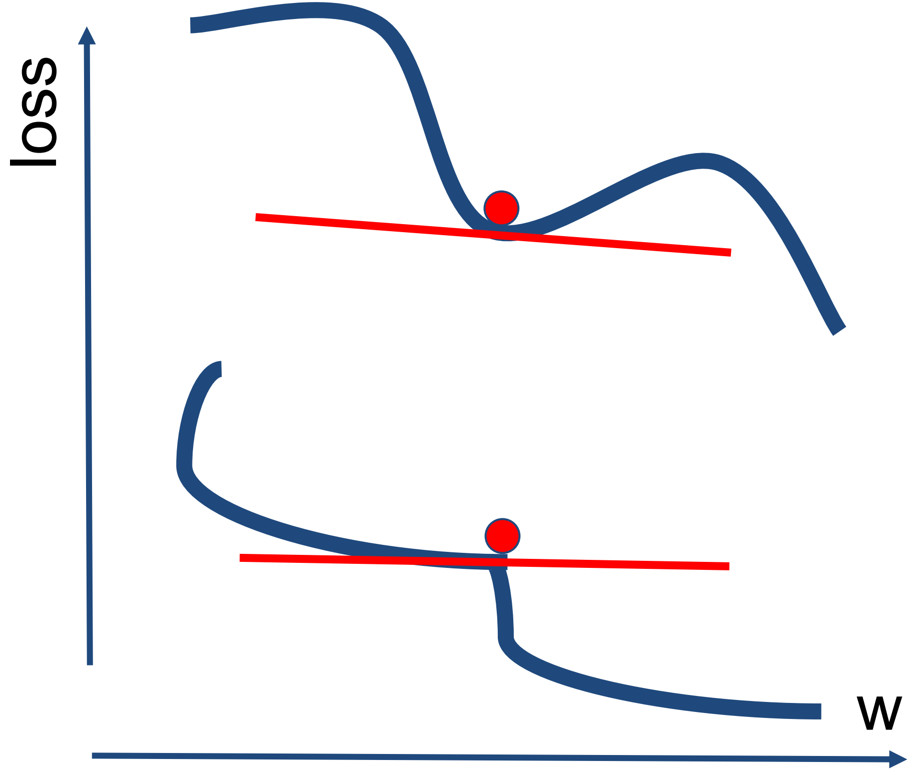
What if the loss function has a **local minima** or **saddle point**?



Optimization: Problem #2 with SGD

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

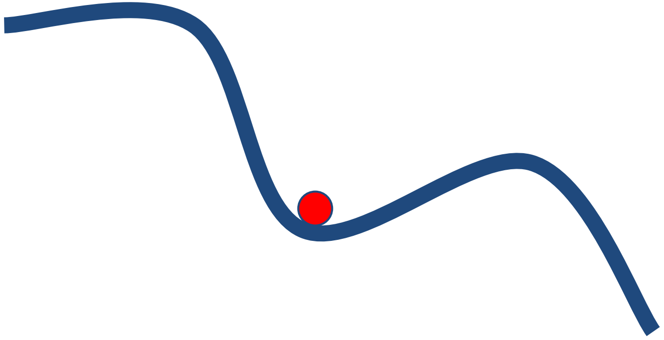
Zero gradient, gradient descent gets stuck

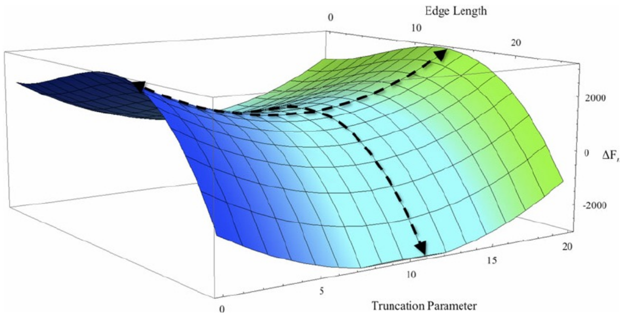


Optimization: Problem #2 with SGD

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

Saddle points are much more common in high dimension





https://blog.paperspace.com/intro-to-optimization-in-deep-learning-gradient-descent/

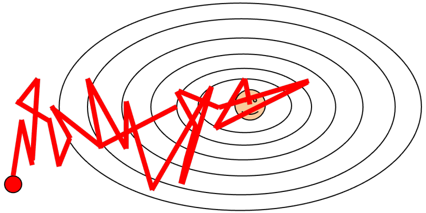
Dauphin et al, “Identifying and attacking the saddle point problem in high-dimensional non-convex optimization”, NIPS 2014

SGD + Momentum

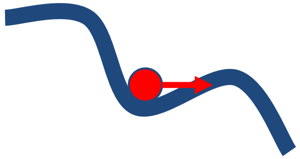
Intuitions:

- Think of a ball (set of parameters) moving in space (loss landscape), with momentum keeping it going in a direction.
- Individual gradient step may be noisy, the general trend accumulated over a few steps will point to the right direction.
- Momentum can “push” the ball over saddle points or local minima.

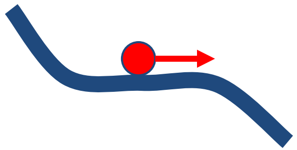
Noisy gradients



Local Minima



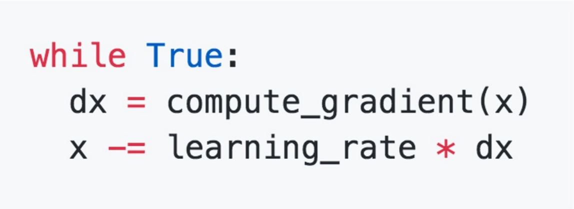
Saddle points



SGD: the simple two line update code

SGD

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



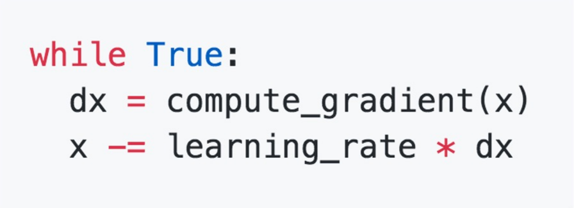
```
while True:
  dx = compute_gradient(x)
  x -= learning_rate * dx
```

SGD + Momentum:

continue moving in the general direction as the previous iterations

SGD

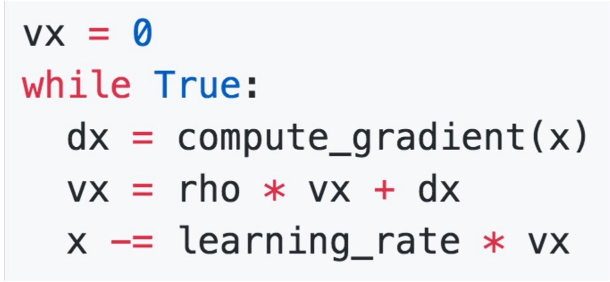
$$x_{t+1} = x_t - \alpha \nabla f(x_t)$$



```
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/momentum” as a running mean of gradients
- Rho gives “friction”; typically rho=0.9 or 0.99

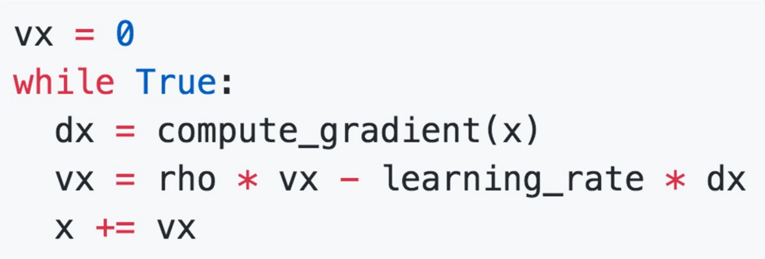
Sutskever et al, “On the importance of initialization and momentum in deep learning”, ICML 2013

SGD + Momentum:

alternative equivalent formulation

SGD+Momentum

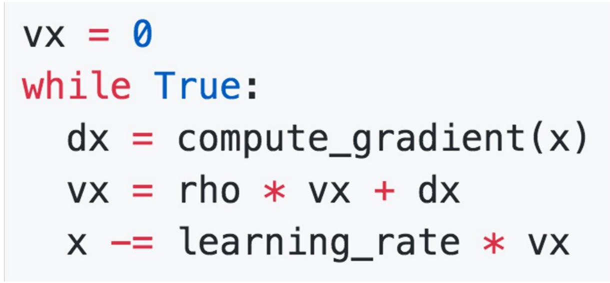
$$v_{t+1} = \rho v_t - \alpha \nabla f(x_t)$$
$$x_{t+1} = x_t + v_{t+1}$$



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

SGD+Momentum

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



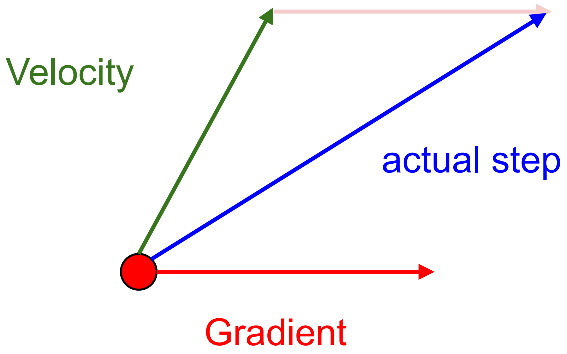
```
vx = 0
while True:
  dx = compute_gradient(x)
  vx = rho * vx + dx
  x -= learning_rate * vx
```

You may see SGD+Momentum formulated different ways, but they are equivalent - give same sequence of x

Sutskever et al, “On the importance of initialization and momentum in deep learning”, ICML 2013

SGD+Momentum

Momentum update:

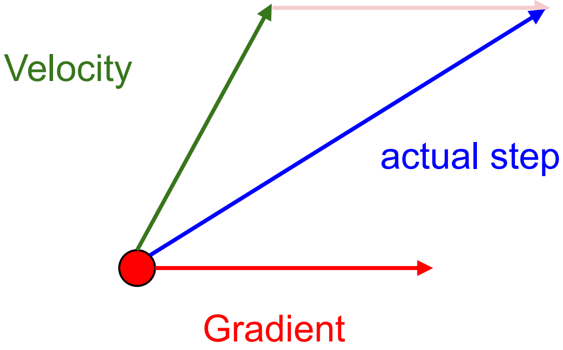


Combine gradient at current point with velocity to get step used to update weights

Nesterov, “A method of solving a convex programming problem with convergence rate O(1/k^2)”, 1983
Nesterov, “Introductory lectures on convex optimization: a basic course”, 2004
Sutskever et al, “On the importance of initialization and momentum in deep learning”, ICML 2013

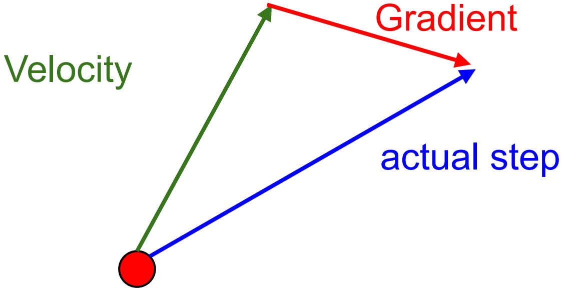
Nesterov Momentum

Momentum update:



Combine gradient at current point with velocity to get step used to update weights

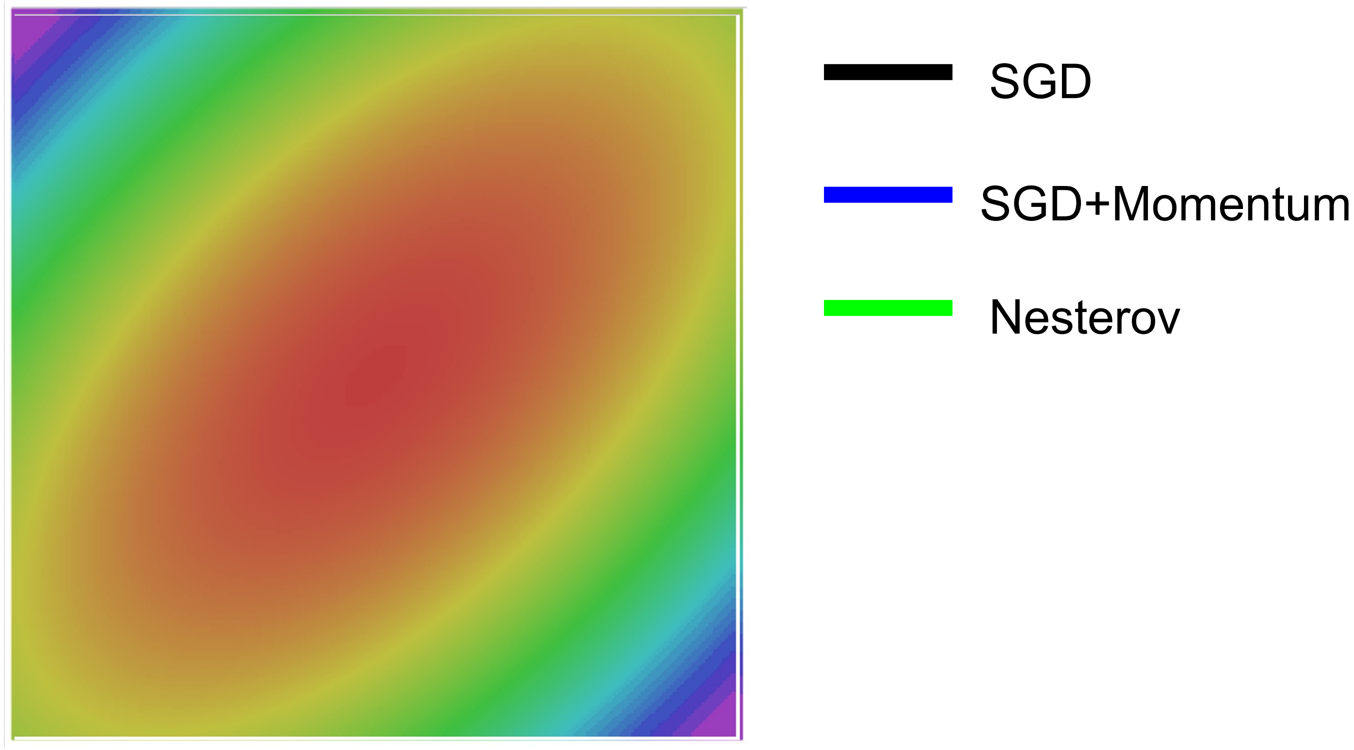
Nesterov Momentum



“Look ahead” to the point where updating using velocity would take us; compute gradient there and mix it with velocity to get actual update direction

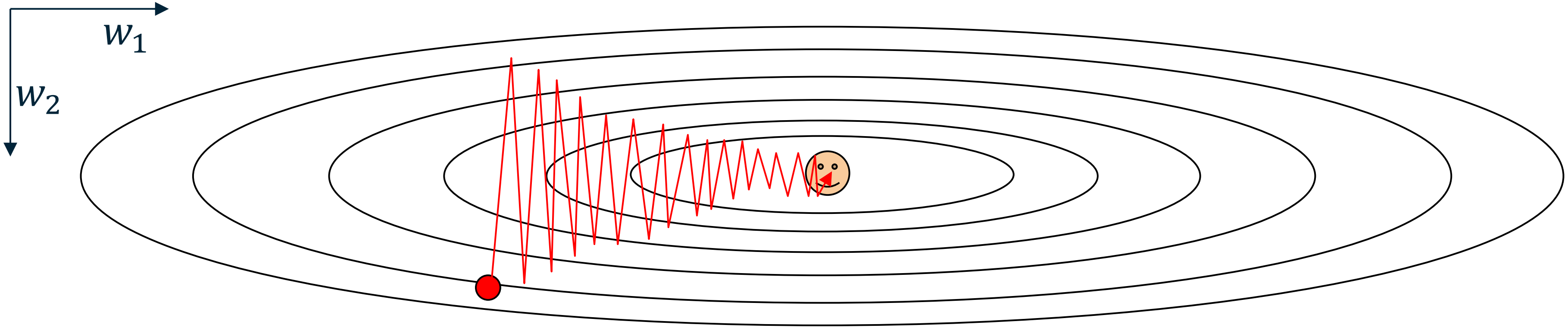
Nesterov, “A method of solving a convex programming problem with convergence rate O(1/k^2)”, 1983
Nesterov, “Introductory lectures on convex optimization: a basic course”, 2004
Sutskever et al, “On the importance of initialization and momentum in deep learning”, ICML 2013

Nesterov Momentum



Optimization: Problem #3 with SGD

What if loss **changes quickly** in one direction and slowly in another?
What does gradient descent do?
Very slow progress along shallow dimension, jitter along steep direction



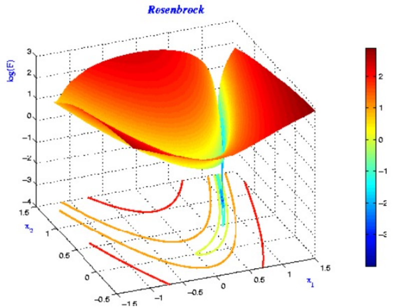
Assume each contour line has the same loss
(iso-loss contour)

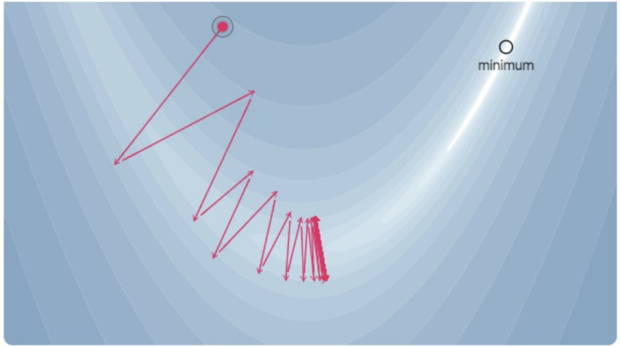
Optimization: Problem #3 with SGD

What if loss changes quickly in one direction and slowly in another?

Very slow progress along shallow dimension, jitter along steep direction

Long, narrow ravines:





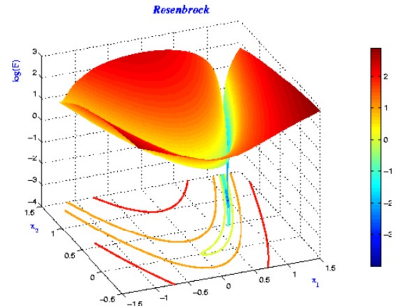
https://www.cs.toronto.edu/~rgrosse/courses/csc421_2019/slides/lec07.pdf

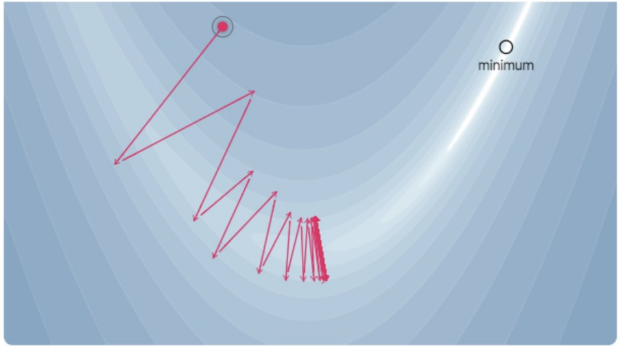
Optimization: Problem #3 with SGD

What if loss changes quickly in one direction and slowly in another?

Very slow progress along shallow dimension, jitter along steep direction

Long, narrow ravines:





https://www.cs.toronto.edu/~rgrosse/courses/csc421_2019/slides/lec07.pdf

Loss function has high **condition number**: ratio of largest to smallest eigen value ($\lambda_{max}/\lambda_{min}$) of the Hessian matrix of a loss function is large

Small condition number in loss Hessian -> circular contour

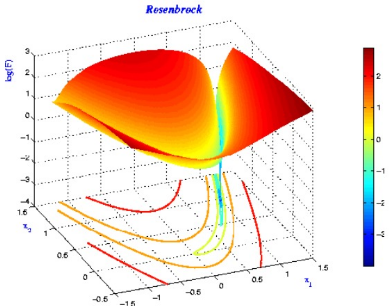
Large condition number in loss Hessian -> skewed contour

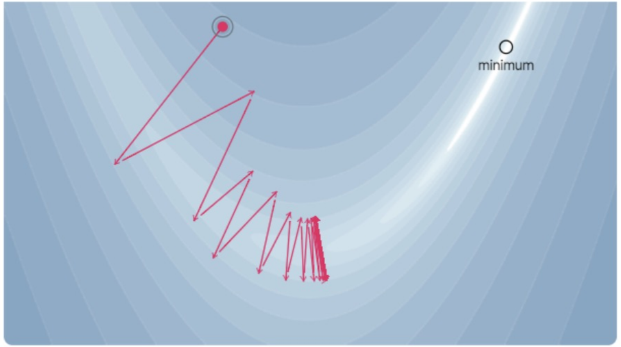
Optimization: Problem #3 with SGD

What if loss changes quickly in one direction and slowly in another?

Very slow progress along shallow dimension, jitter along steep direction

Long, narrow ravines:



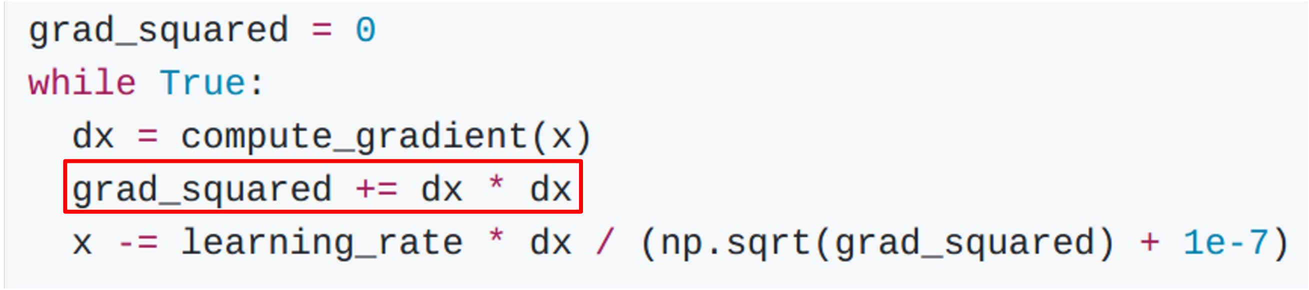


https://www.cs.toronto.edu/~rgrosse/courses/csc421_2019/slides/lec07.pdf

Ideally, we want different learning rate for **each weight dimension** to account for the skewness of the loss landscape, i.e., low LR for fast-changing direction and small LR for slow-changing direction.

Manually picking an optimal LR for each weight dimension seems hard …

AdaGrad



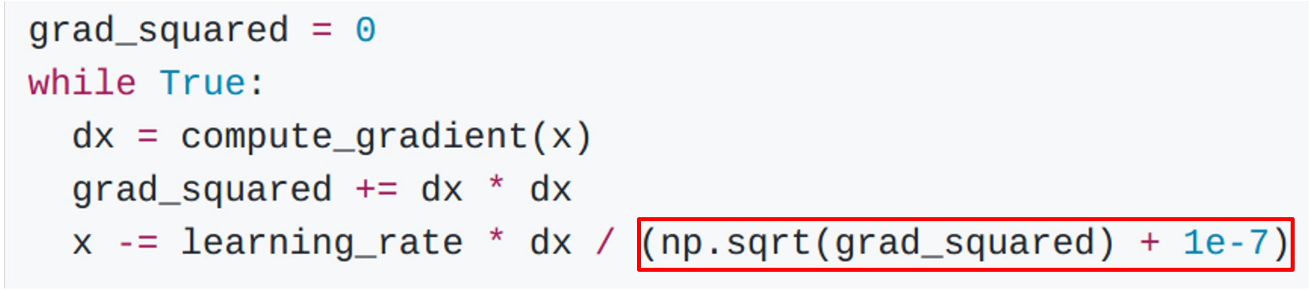
```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared += dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

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

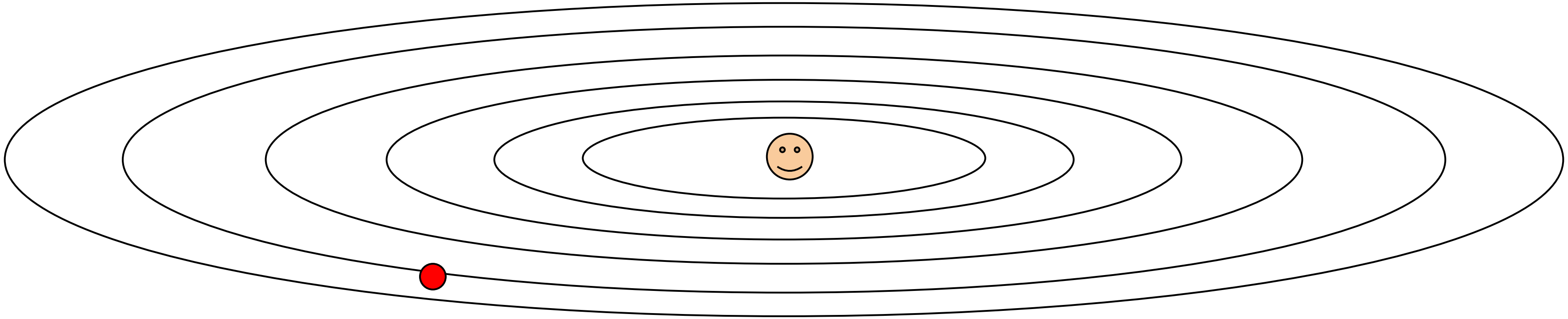
“Per-parameter learning rates”
or “adaptive learning rates”

Duchi et al, “Adaptive subgradient methods for online learning and stochastic optimization”, JMLR 2011

AdaGrad



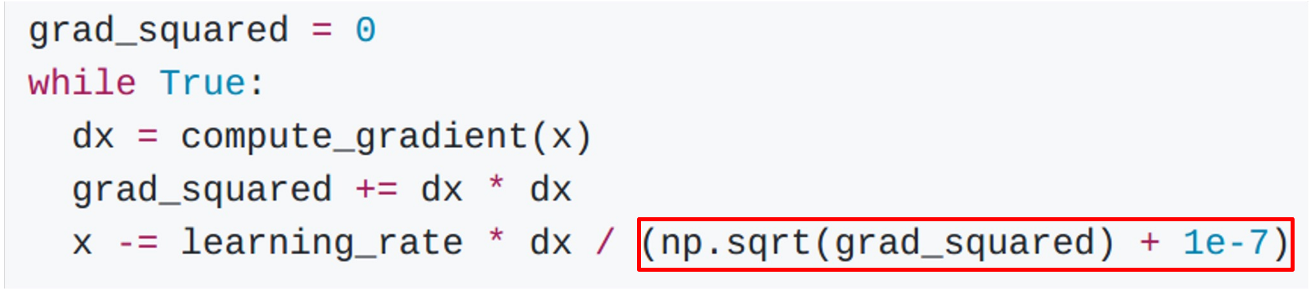
```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared += dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```



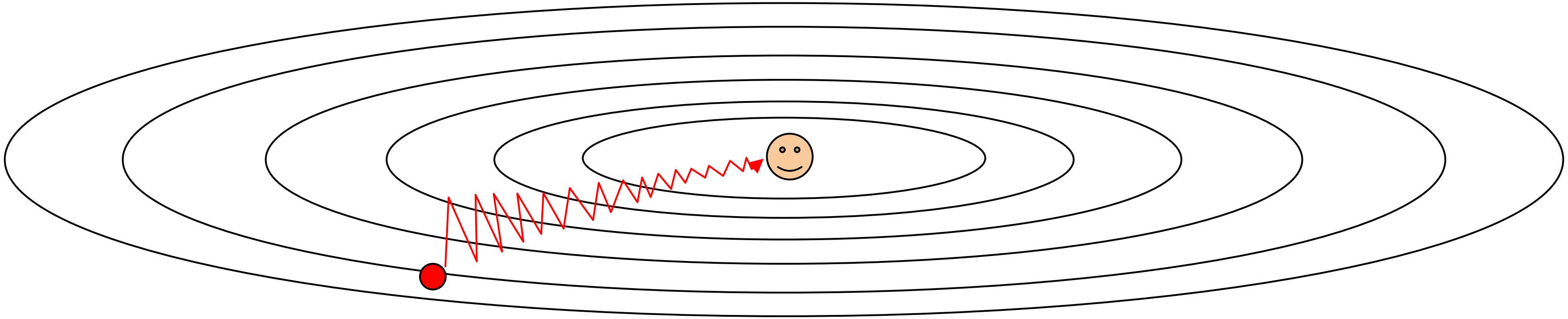
Q: What happens with AdaGrad?

Duchi et al, “Adaptive subgradient methods for online learning and stochastic optimization”, JMLR 2011

AdaGrad



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

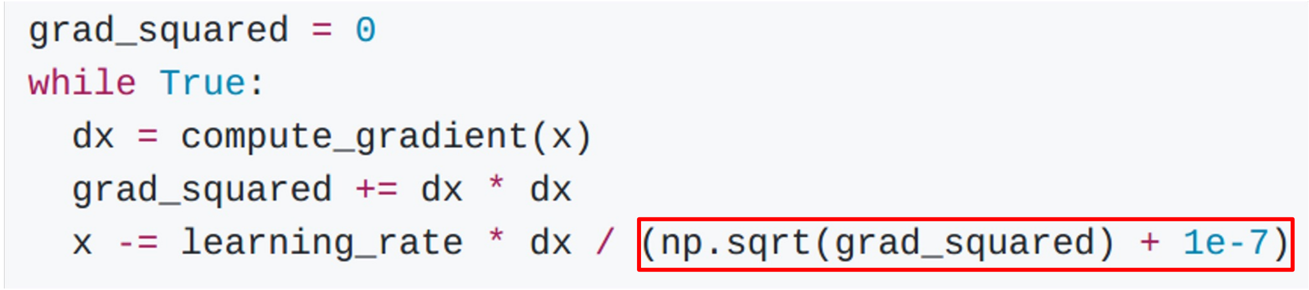


Q: What happens with AdaGrad?

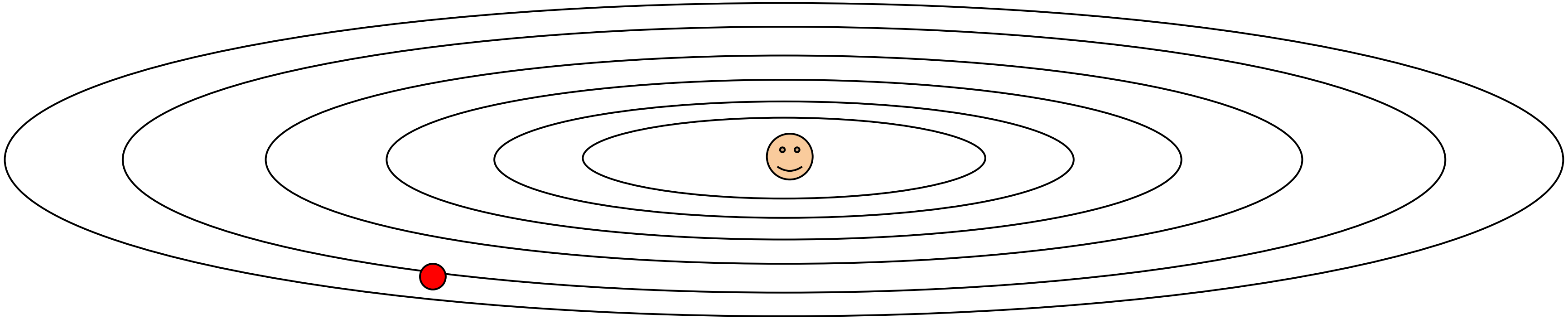
Progress along “steep” directions is damped; progress along “flat” directions is accelerated ☺

Duchi et al, “Adaptive subgradient methods for online learning and stochastic optimization”, JMLR 2011

AdaGrad



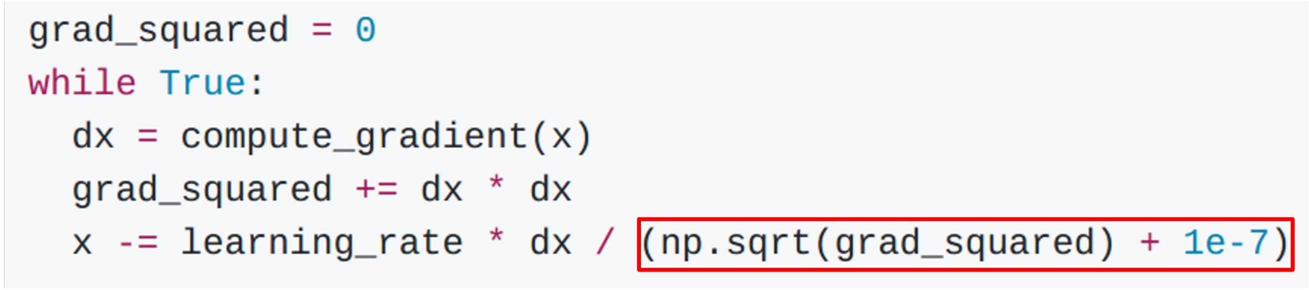
```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared += dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```



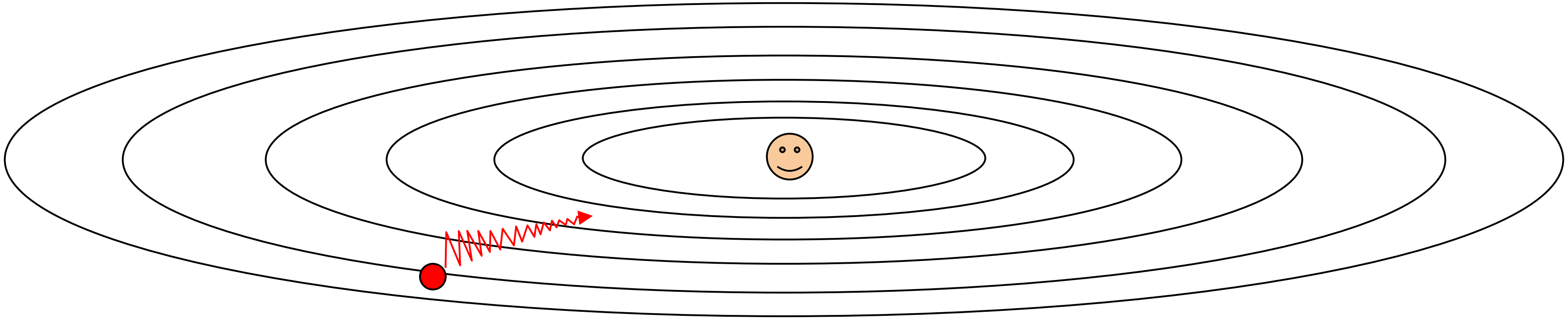
Q2: What happens to the step size over long time?

Duchi et al, “Adaptive subgradient methods for online learning and stochastic optimization”, JMLR 2011

AdaGrad



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



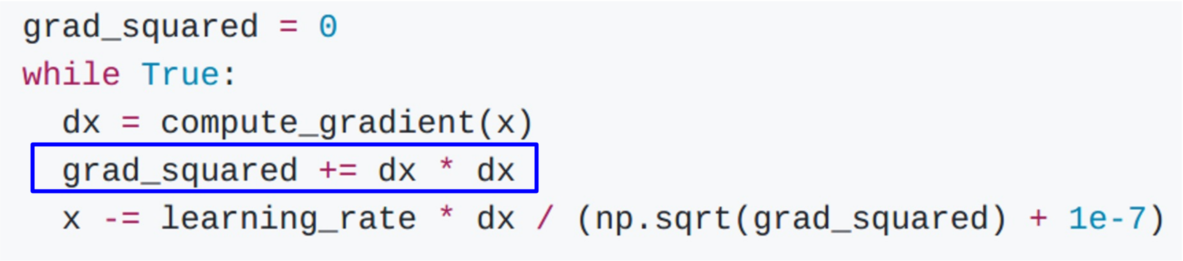
Q2: What happens to the step size over long time?

Decays to zero ☹

Duchi et al, “Adaptive subgradient methods for online learning and stochastic optimization”, JMLR 2011

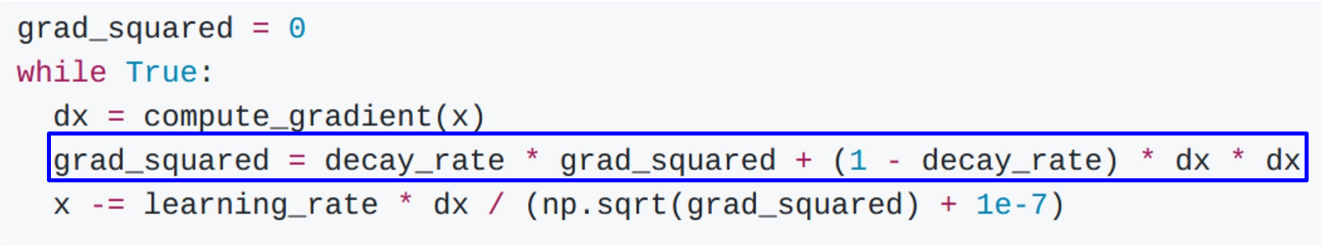
RMSProp: “Leaky AdaGrad”

AdaGrad



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

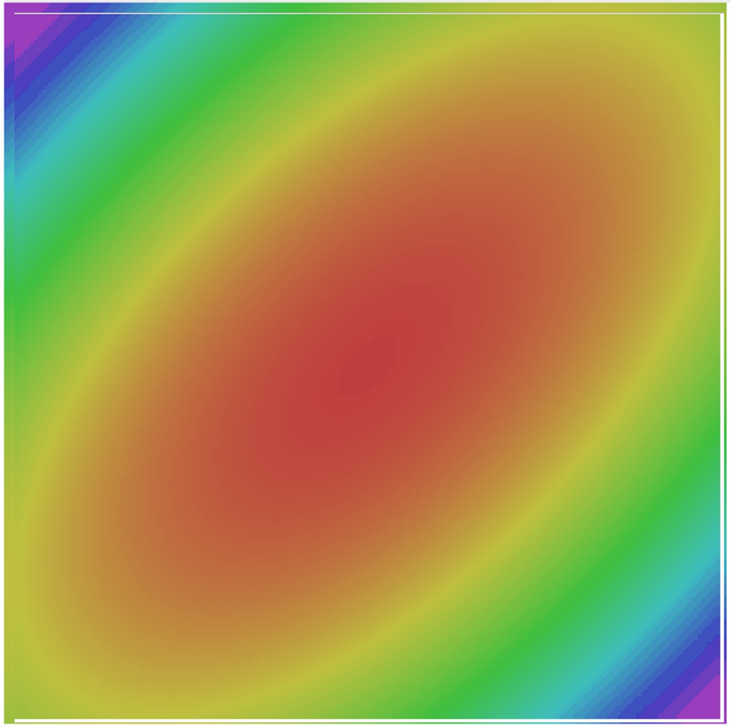
RMSProp



```
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

RMSProp



⬛ SGD

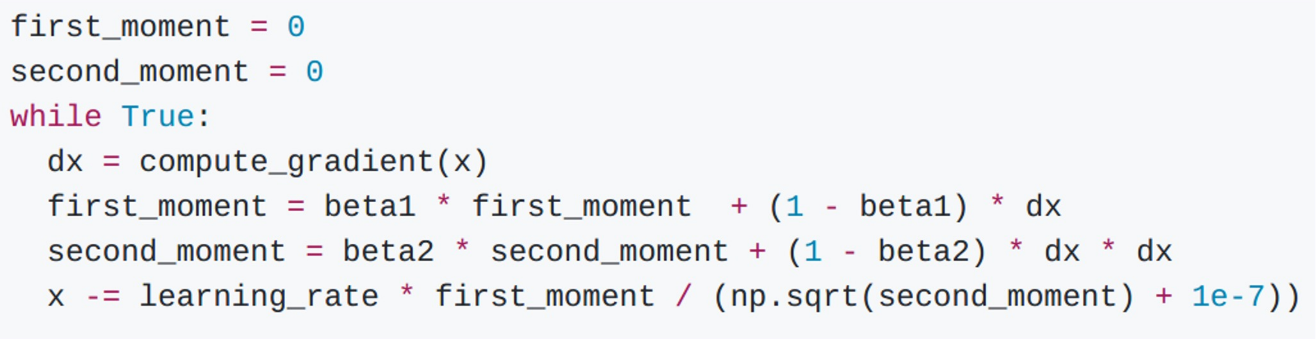
🟦 SGD+Momentum

🟥 RMSProp

🟩 AdaGrad

(stuck due to decaying lr)

Adam (almost)

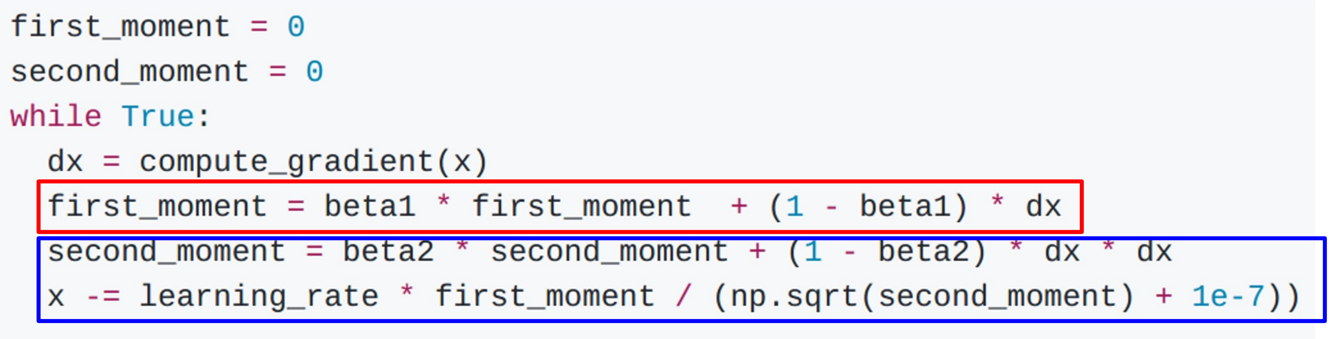


```
first_moment = 0
second_moment = 0
while True:
  dx = compute_gradient(x)
  first_moment = beta1 * first_moment  + (1 - beta1) * dx
  second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
  x -= learning_rate * first_moment / (np.sqrt(second_moment) + 1e-7))
```

Typical hyperparams: beta1=0.9, beta2=0.999

Kingma and Ba, “Adam: A method for stochastic optimization”, ICLR 2015

Adam (almost)



```
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

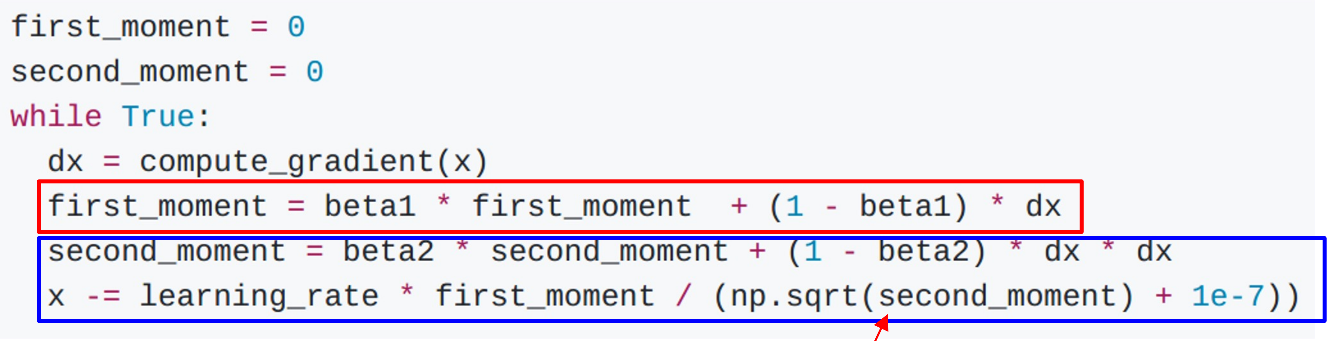
Typical hyperparams: beta1=0.9, beta2=0.999

Sort of like RMSProp with momentum

Q: What happens at first timestep?

Kingma and Ba, “Adam: A method for stochastic optimization”, ICLR 2015

Adam (almost)



```
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

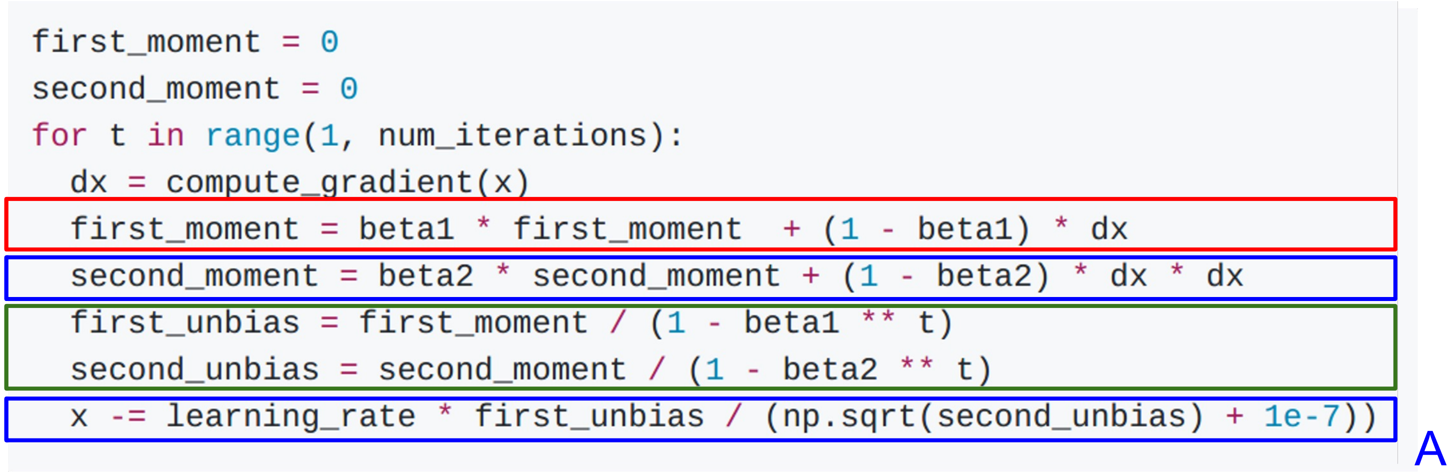
Typical hyperparams: beta1=0.9, beta2=0.999

Small -> divide by small number -> bad initial step

Q: What happens at first timestep?

Kingma and Ba, “Adam: A method for stochastic optimization”, ICLR 2015

Adam (full form)



```
first_moment = 0
second_moment = 0
for t in range(1, 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

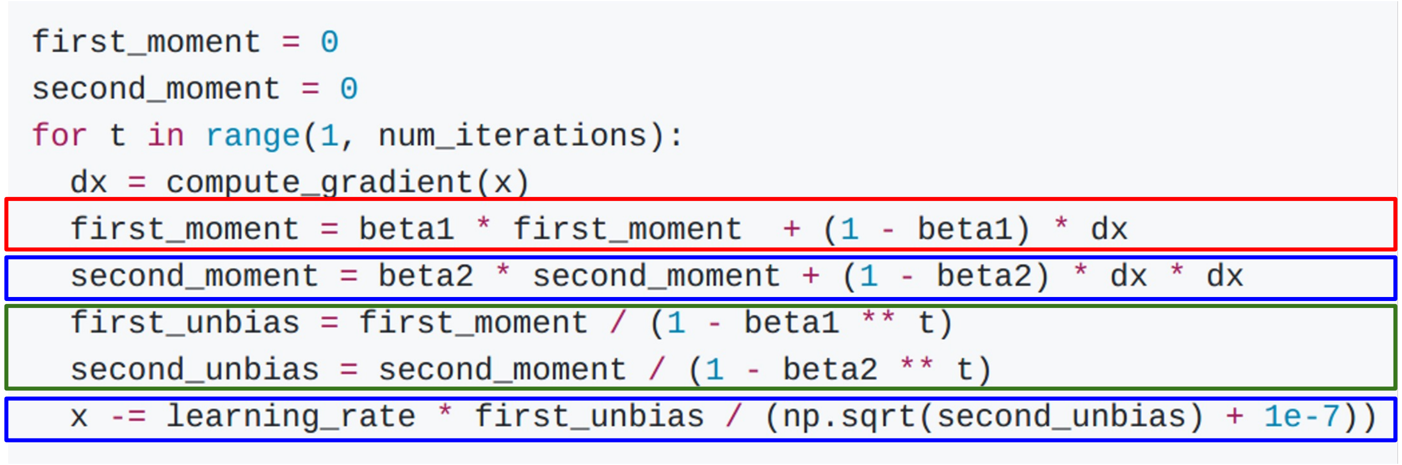
AdaGrad / RMSProp

Typical hyperparams: beta1=0.9, beta2=0.999

Bias correction for the fact that first and second moment estimates start at zero

Kingma and Ba, “Adam: A method for stochastic optimization”, ICLR 2015

Adam (full form)



```
first_moment = 0
second_moment = 0
for t in range(1, 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

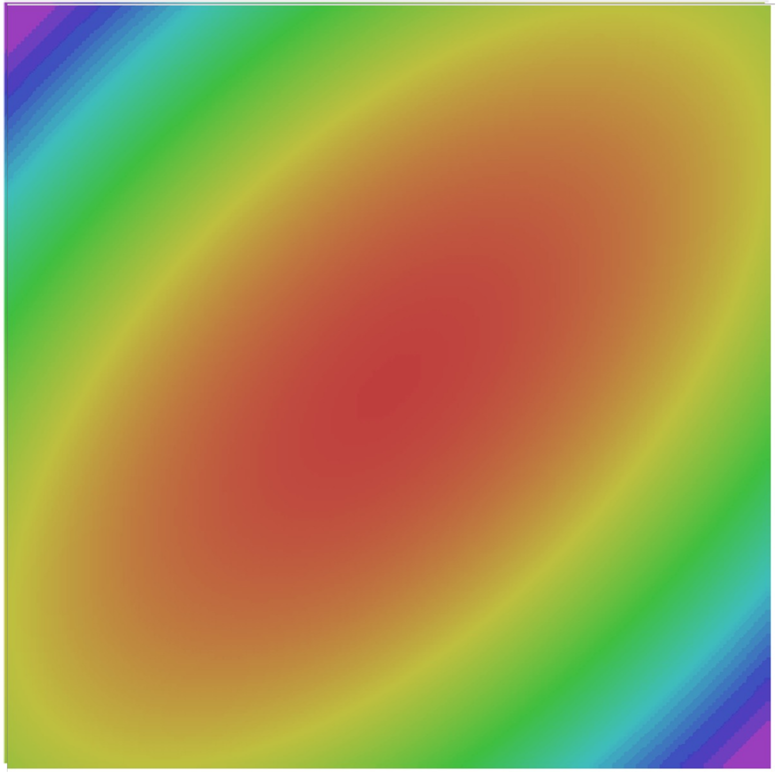
Typical hyperparams: beta1=0.9, beta2=0.999

Bias correction for the fact that first and second moment estimates start at zero

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, "Adam: A method for stochastic optimization", ICLR 2015

Adam



⬛ SGD

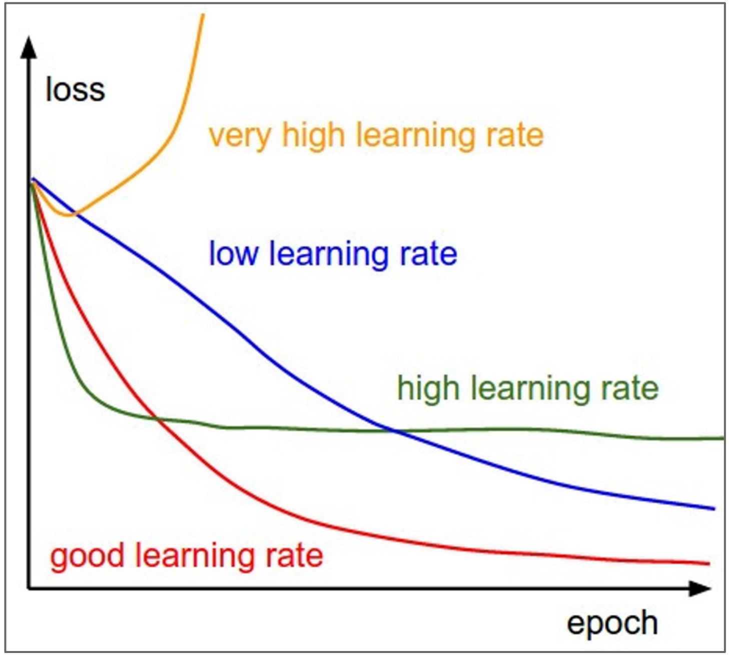
🟦 SGD+Momentum

🟥 RMSProp

🟪 Adam

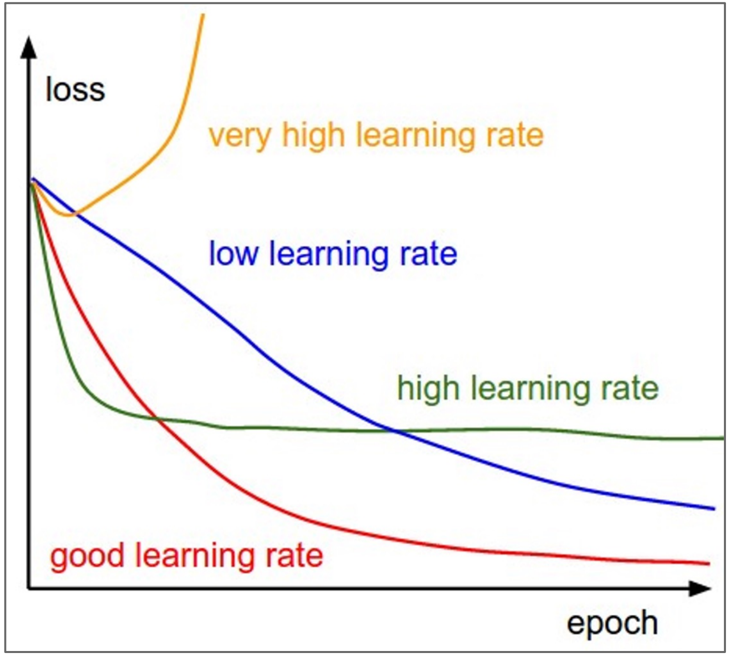
Learning rate schedules

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



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

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

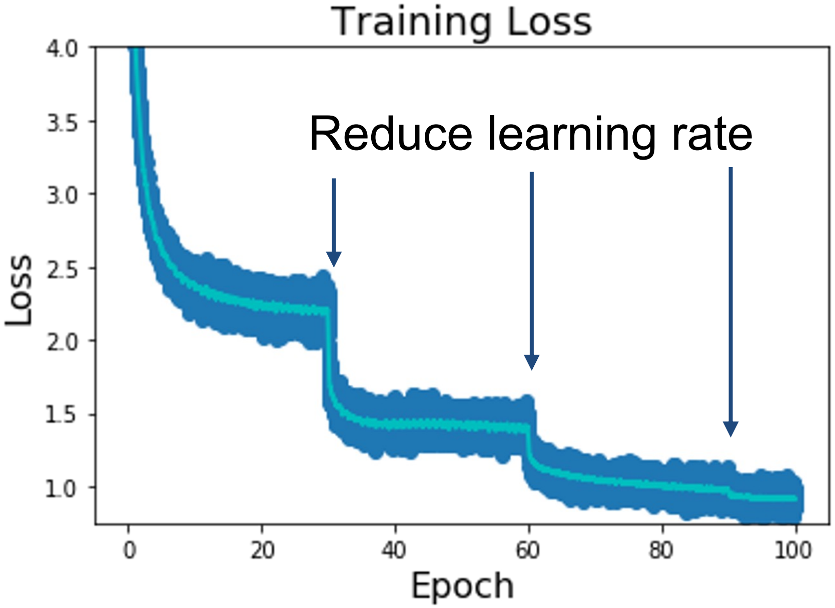


Q: Which one of these learning rates is best to use?

A: In reality, all of these are good learning rates.

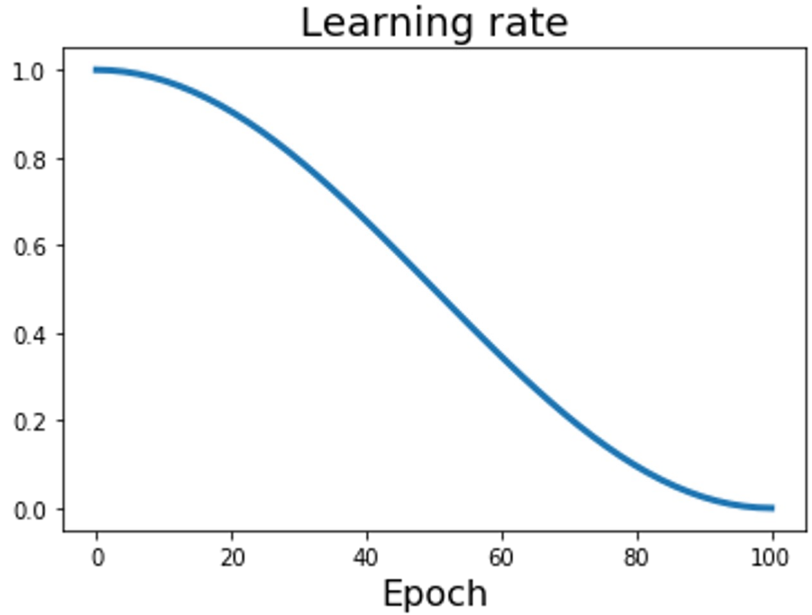
Need finer adjustment closer to convergence, so we want to reduce learning rate over time to keep making progress.

Learning rate decays over time



Step: Reduce learning rate at a few fixed points. E.g. for ResNets, multiply LR by 0.1 after epochs 30, 60, and 90.

Learning Rate Decay



Step: Reduce learning rate at a few fixed points. E.g. for ResNets, multiply LR by 0.1 after epochs 30, 60, and 90.

Cosine: $\alpha_t = \frac{1}{2}\alpha_0\left(1 + \cos(t\pi/T)\right)$

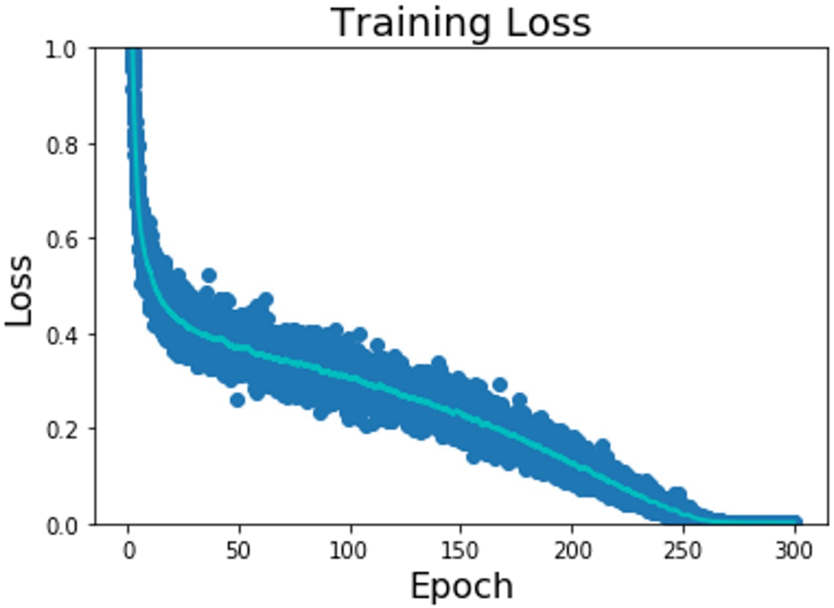
α_0 : Initial learning rate

α_t : Learning rate at epoch t

T : Total number of epochs

Loshchilov and Hutter, "SGDR: Stochastic Gradient Descent with Warm Restarts", ICLR 2017
Radford et al, "Improving Language Understanding by Generative Pre-Training", 2018
Feichtenhofer et al, "SlowFast Networks for Video Recognition", arXiv 2018
Child at al, "Generating Long Sequences with Sparse Transformers", arXiv 2019

Learning Rate Decay



Step: Reduce learning rate at a few fixed points. E.g. for ResNets, multiply LR by 0.1 after epochs 30, 60, and 90.

Cosine: $\alpha_t = \frac{1}{2}\alpha_0 \left(1 + \cos(t\pi/T)\right)$

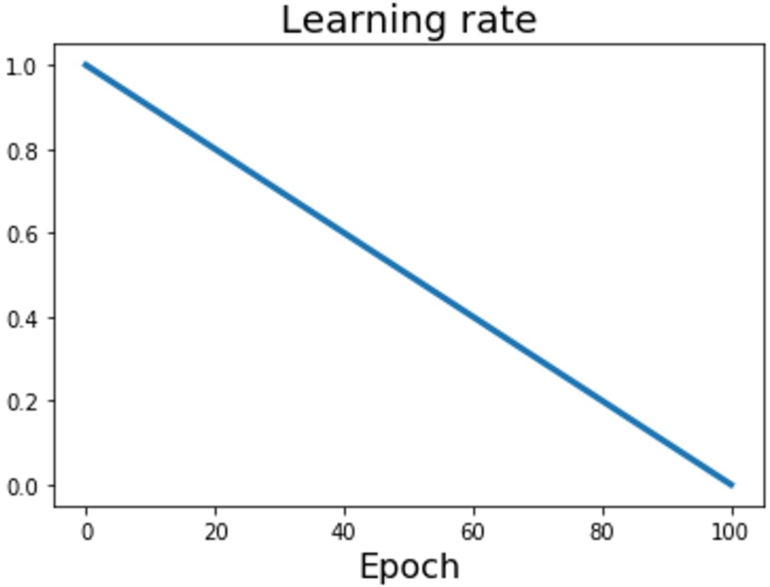
α_0 : Initial learning rate

α_t : Learning rate at epoch t

T : Total number of epochs

Loshchilov and Hutter, “SGDR: Stochastic Gradient Descent with Warm Restarts”, ICLR 2017
Radford et al, “Improving Language Understanding by Generative Pre-Training”, 2018
Feichtenhofer et al, “SlowFast Networks for Video Recognition”, arXiv 2018
Child at al, “Generating Long Sequences with Sparse Transformers”, arXiv 2019

Learning Rate Decay



Step: Reduce learning rate at a few fixed points. E.g. for ResNets, multiply LR by 0.1 after epochs 30, 60, and 90.

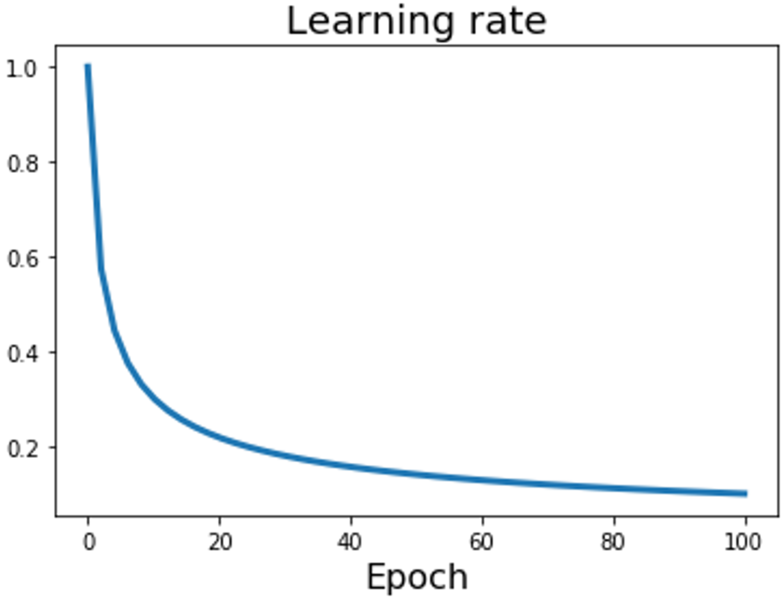
Cosine: $\alpha_t = \frac{1}{2}\alpha_0 \left(1 + \cos(t\pi/T)\right)$

Linear: $\alpha_t = \alpha_0(1 - t/T)$

α_0 : Initial learning rate
α_t : Learning rate at epoch t
T : Total number of epochs

Devlin et al, “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding”, 2018

Learning Rate Decay



Step: Reduce learning rate at a few fixed points. E.g. for ResNets, multiply LR by 0.1 after epochs 30, 60, and 90.

Cosine: $\alpha_t = \frac{1}{2}\alpha_0\left(1 + \cos(t\pi/T)\right)$

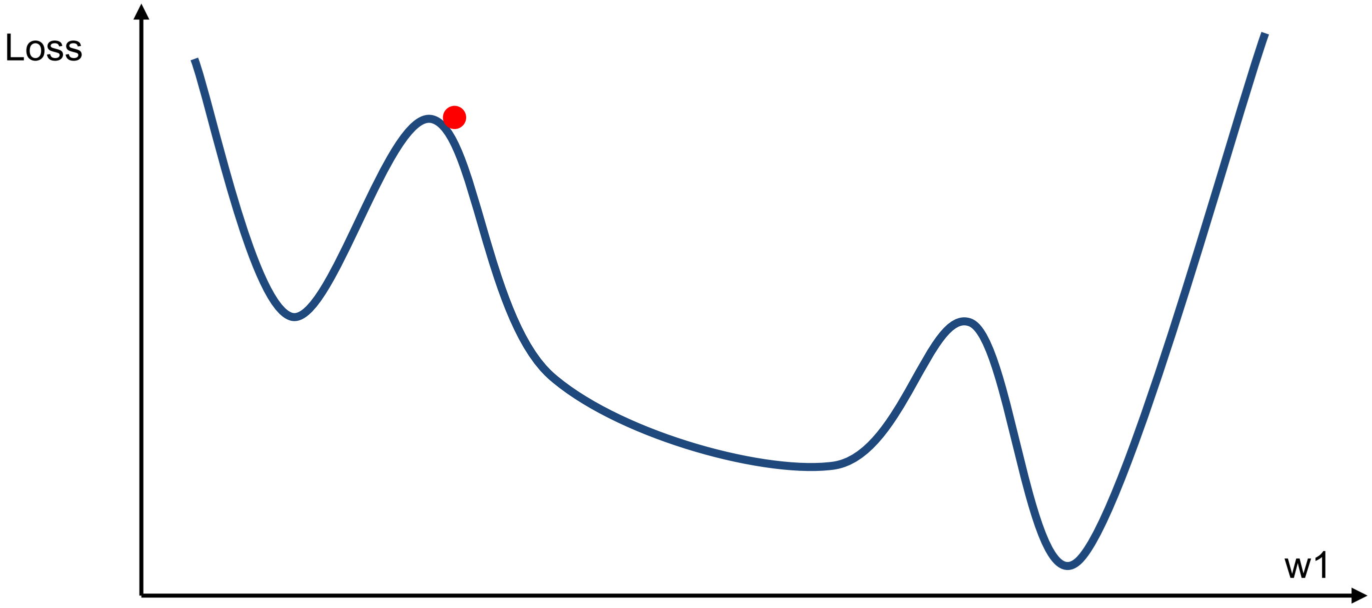
Linear: $\alpha_t = \alpha_0(1 - t/T)$

Inverse sqrt: $\alpha_t = \alpha_0/\sqrt{t}$

α_0 : Initial learning rate
α_t : Learning rate at epoch t
T : Total number of epochs

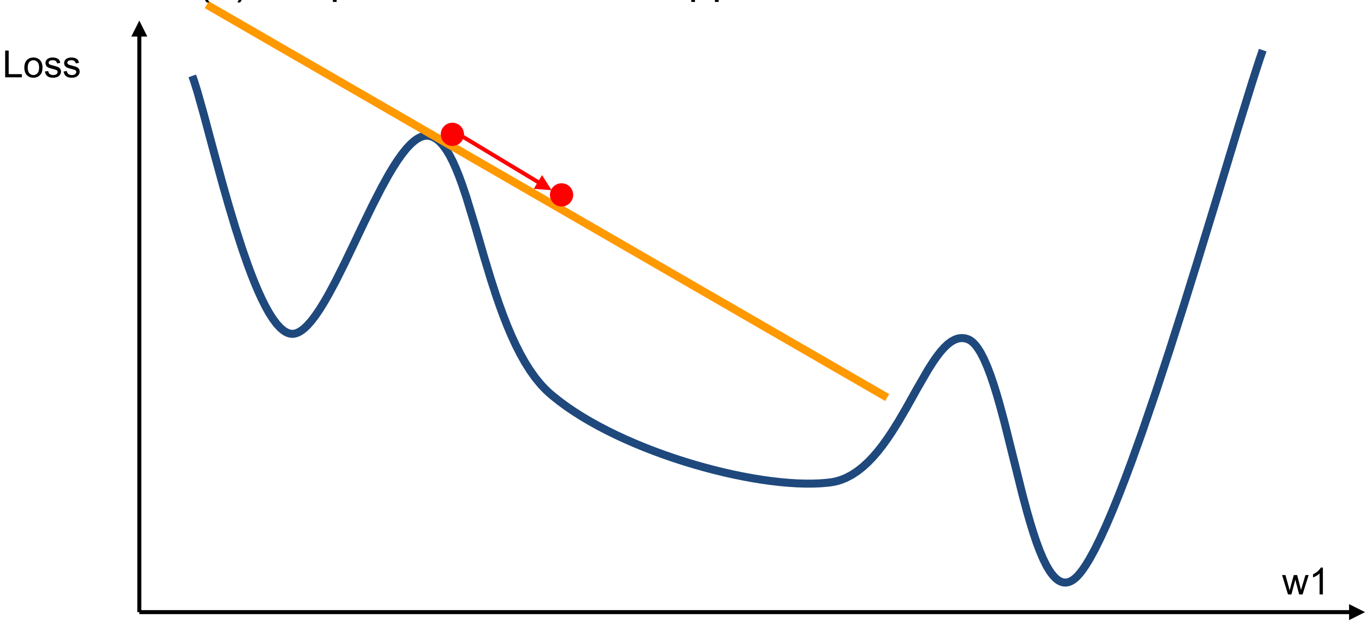
Vaswani et al, “Attention is all you need”, NIPS 2017

First-Order Optimization



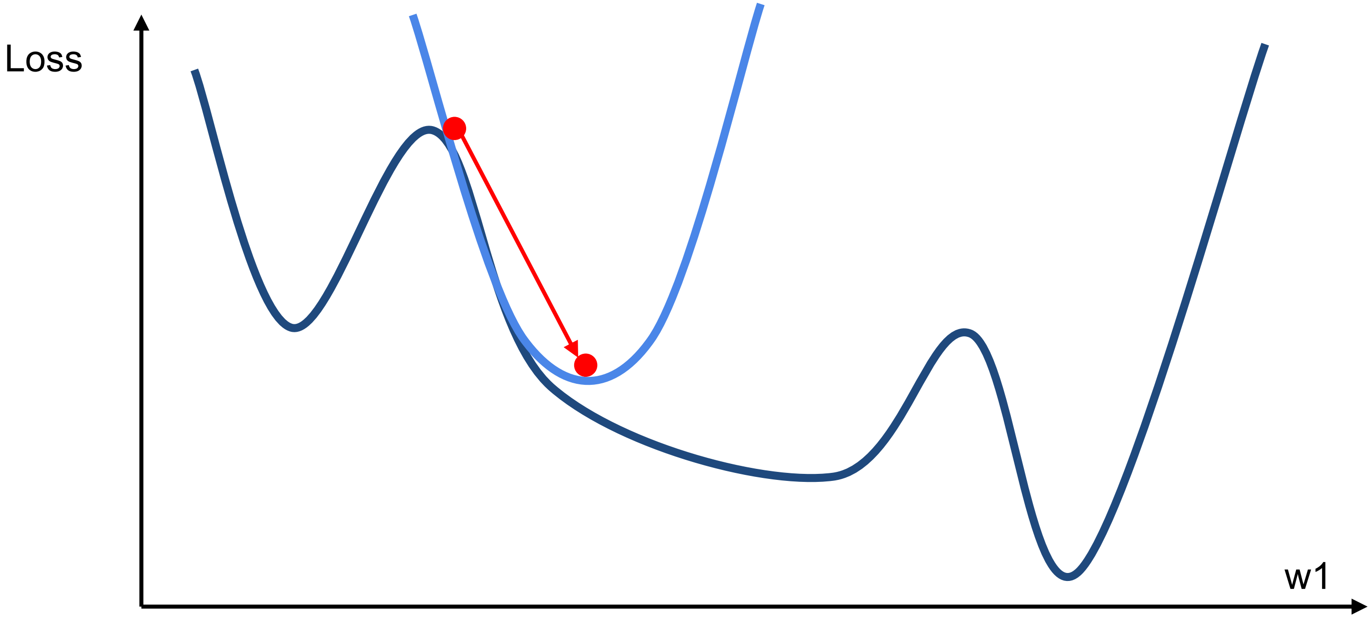
First-Order Optimization

(1) Use gradient form linear approximation
(2) Step to minimize the approximation



Second-Order Optimization

(1) Use gradient **and Hessian** to form **quadratic** approximation
(2) Step to the **minima** of the approximation



Second-Order Optimization

second-order Taylor expansion:

$$J(\boldsymbol{\theta}) \approx J(\boldsymbol{\theta}_0) + (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0) + \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \boldsymbol{H}(\boldsymbol{\theta} - \boldsymbol{\theta}_0)$$

Solving for the critical point we obtain the Newton parameter update:

$$\boldsymbol{\theta}^* = \boldsymbol{\theta}_0 - \boldsymbol{H}^{-1} \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0)$$

Q: Why is this bad for deep learning?

Second-Order Optimization

second-order Taylor expansion:

$$J(\boldsymbol{\theta}) \approx J(\boldsymbol{\theta}_0) + (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0) + \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \boldsymbol{H}(\boldsymbol{\theta} - \boldsymbol{\theta}_0)$$

Solving for the critical point we obtain the Newton parameter update:

$$\boldsymbol{\theta}^* = \boldsymbol{\theta}_0 - \boldsymbol{H}^{-1} \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0)$$

Hessian has O(N^2) elements

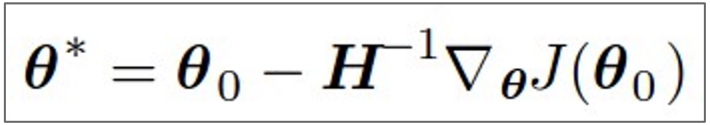
Inverting takes O(N^3)

N = Millions

$$\mathbf{H}_f = \begin{bmatrix} \dfrac{\partial^2 f}{\partial x_1^2} & \dfrac{\partial^2 f}{\partial x_1\,\partial x_2} & \cdots & \dfrac{\partial^2 f}{\partial x_1\,\partial x_n} \\ \dfrac{\partial^2 f}{\partial x_2\,\partial x_1} & \dfrac{\partial^2 f}{\partial x_2^2} & \cdots & \dfrac{\partial^2 f}{\partial x_2\,\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \dfrac{\partial^2 f}{\partial x_n\,\partial x_1} & \dfrac{\partial^2 f}{\partial x_n\,\partial x_2} & \cdots & \dfrac{\partial^2 f}{\partial x_n^2} \end{bmatrix},$$

Q: Why is this bad for deep learning?

Second-Order Optimization



$$\boldsymbol{\theta}^* = \boldsymbol{\theta}_0 - \boldsymbol{H}^{-1}\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0)$$

- Quasi-Newton methods (**BGFS** most popular):
  instead of inverting the Hessian (O(n^3)), approximate inverse Hessian with rank 1 updates over time (O(n^2) each).

- **L-BFGS** (Limited memory BFGS):
  Does not form/store the full inverse Hessian.

L-BFGS

- **Usually works very well in full batch, deterministic mode** i.e. if you have a single, deterministic f(x) then L-BFGS will probably work very nicely

- **Does not transfer very well to mini-batch setting**. Gives bad results. Adapting second-order methods to large-scale, stochastic setting is an active area of research.

Le et al, “On optimization methods for deep learning, ICML 2011”
Ba et al, “Distributed second-order optimization using Kronecker-factored approximations”, ICLR 2017

In practice:

- **Adam** is a good default choice in many cases; it often works ok even with constant learning rate
- **SGD+Momentum** can outperform Adam but may require more tuning of LR and schedule
  - Try cosine schedule, very few hyperparameters!

- If you can afford to do full batch updates (very rare for deep learning applications) then try out **L-BFGS** (and don’t forget to disable all sources of noise)

Next Time:

Training Deep Neural Networks

- Details of the non-linear activation functions
- Data normalization
- Weight Initialization
- Batch Normalization
- Advanced Optimization
- Regularization
- Data Augmentation
- Transfer learning
- Hyperparameter Tuning
- Model Ensemble