

Lecture 6:

Activations, Normalizations, Optimizers

Administrative: EdStem

Please make sure to check and read all pinned EdStem posts.

Administrative: Assignment 2

Due 4/27 11:59pm

- Multi-layer Neural Networks,
- Image Features,
- Optimizers

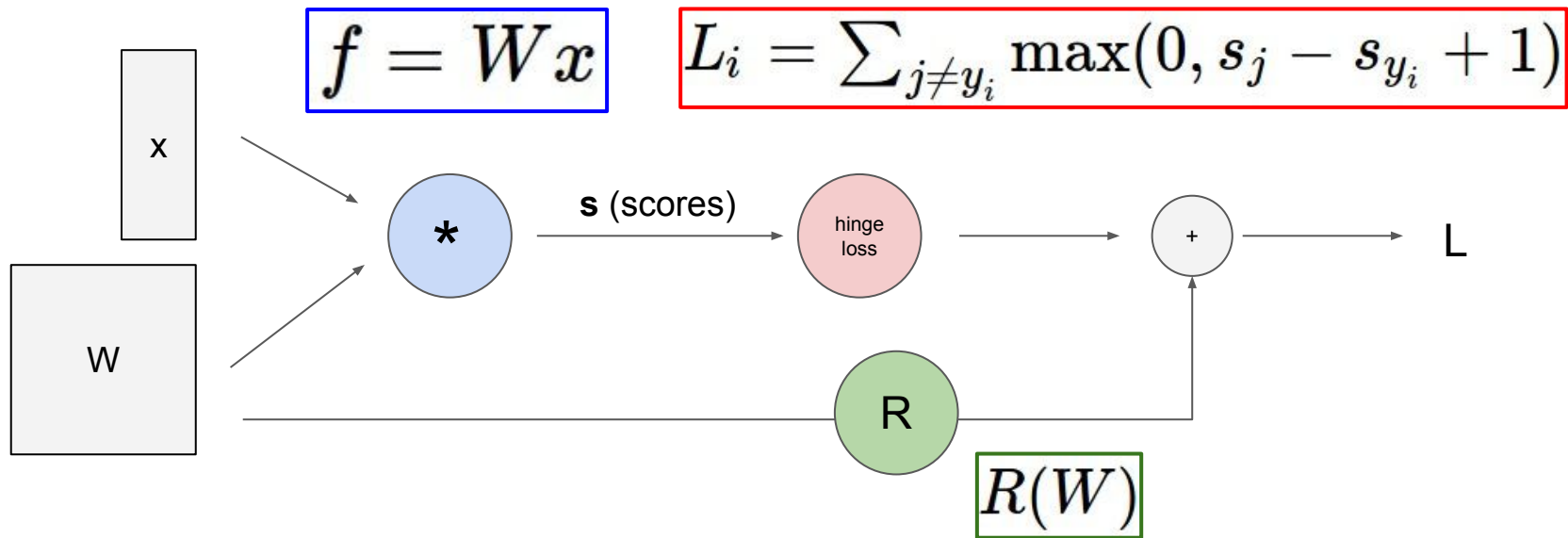
Administrative: Fridays

This Friday

Convolutions & Vectorization

Where we are now...

Computational graphs



Where we are now...

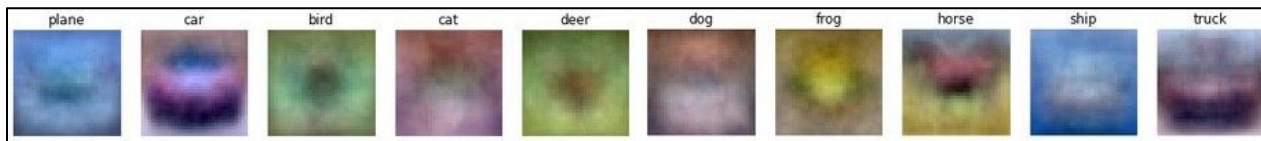
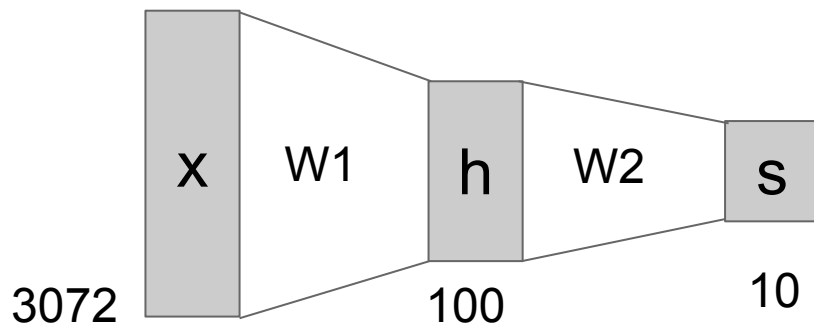
Neural Networks

Linear score function:

$$f = Wx$$

2-layer Neural Network

$$f = W_2 \max(0, W_1 x)$$



Where we are now...

Convolutional Neural Networks

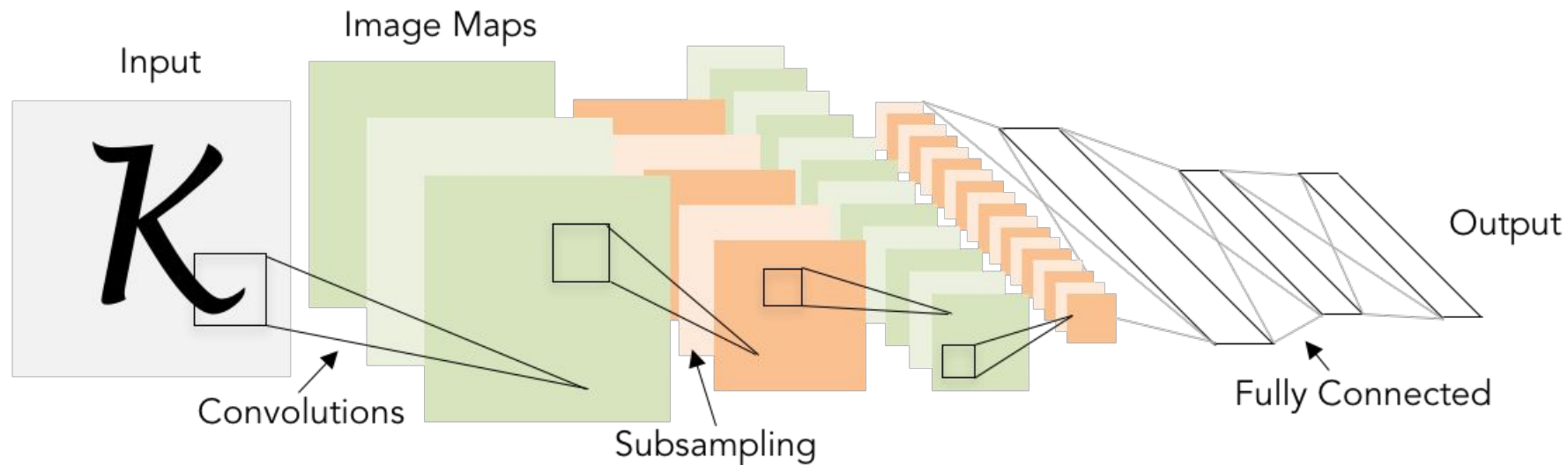
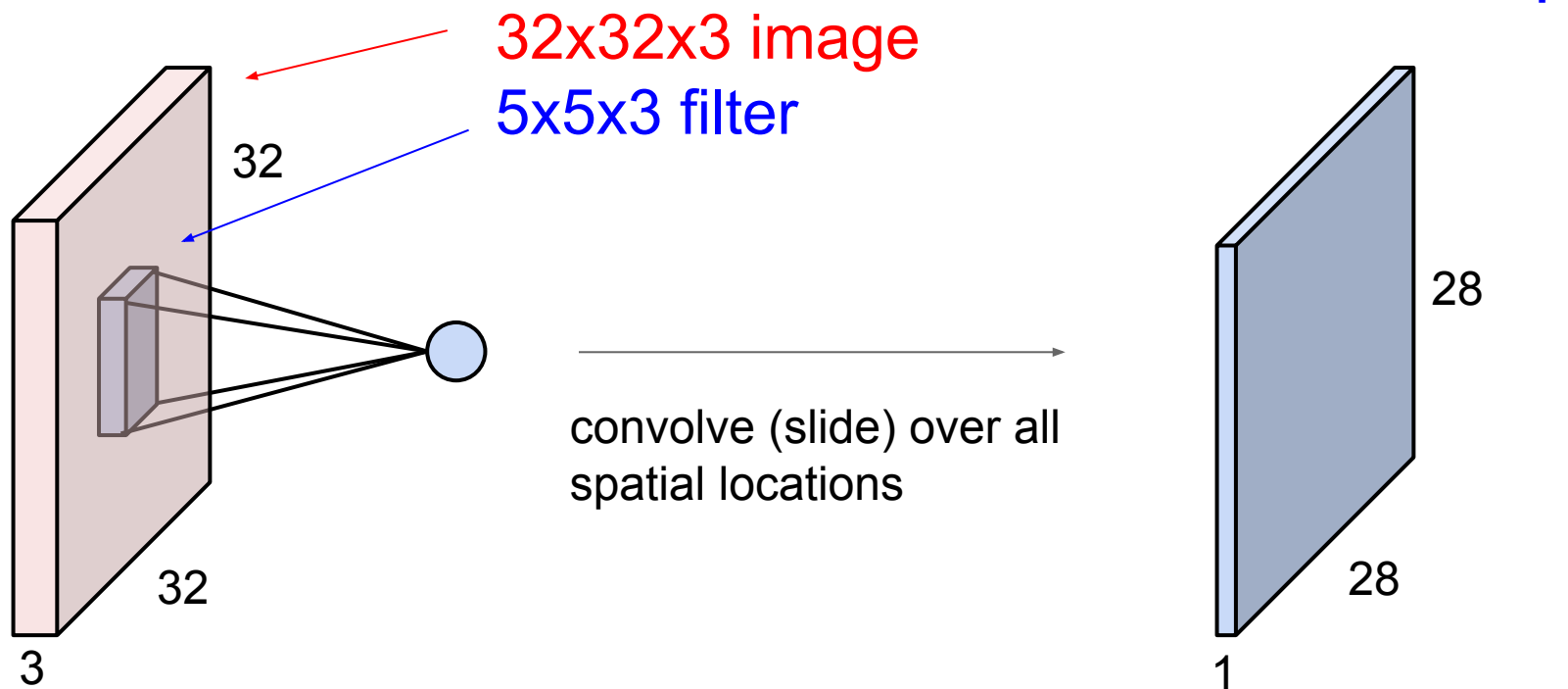


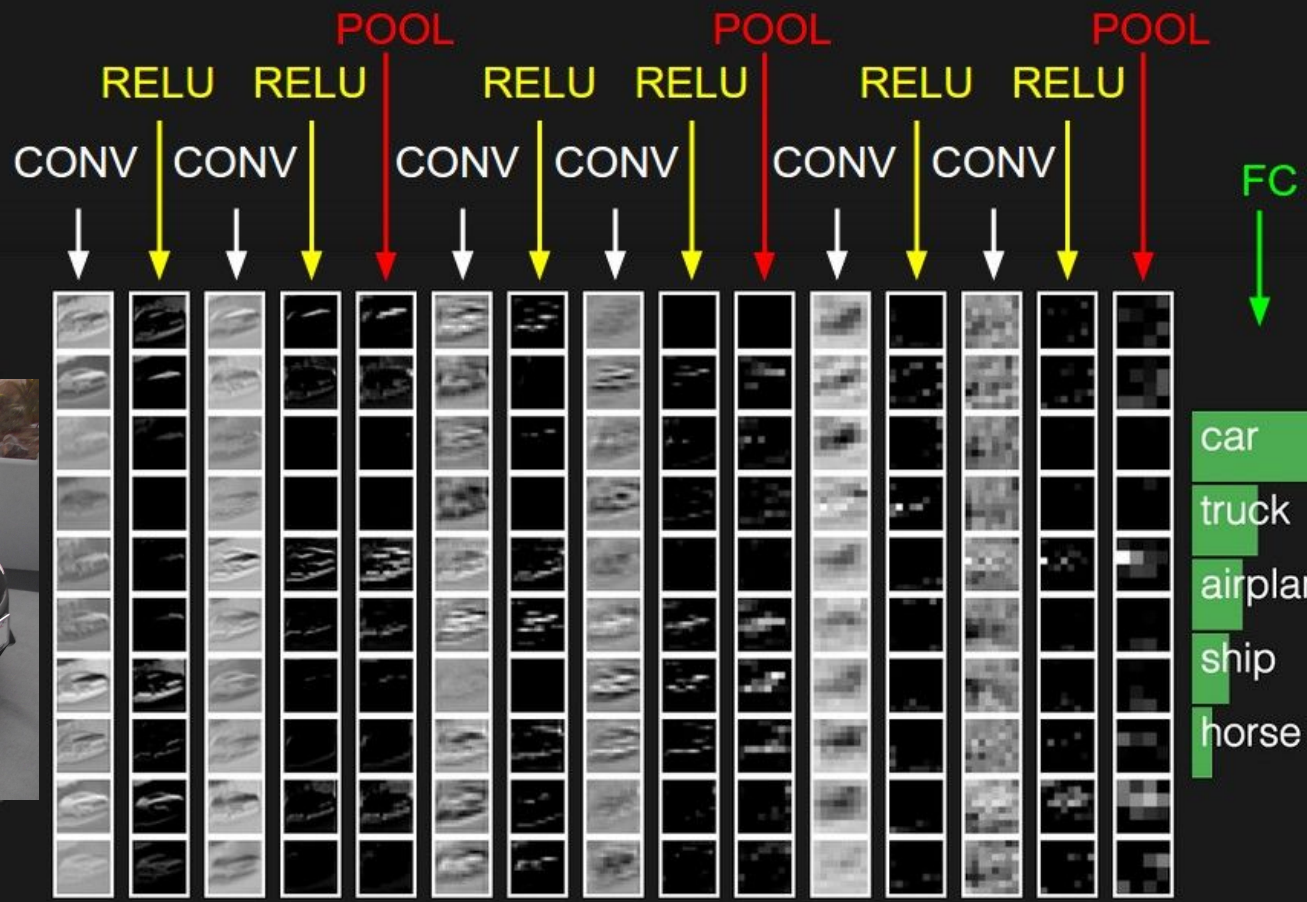
Illustration of LeCun et al. 1998 from CS231n 2017 Lecture 1

Where we are now...

Convolutional Layer

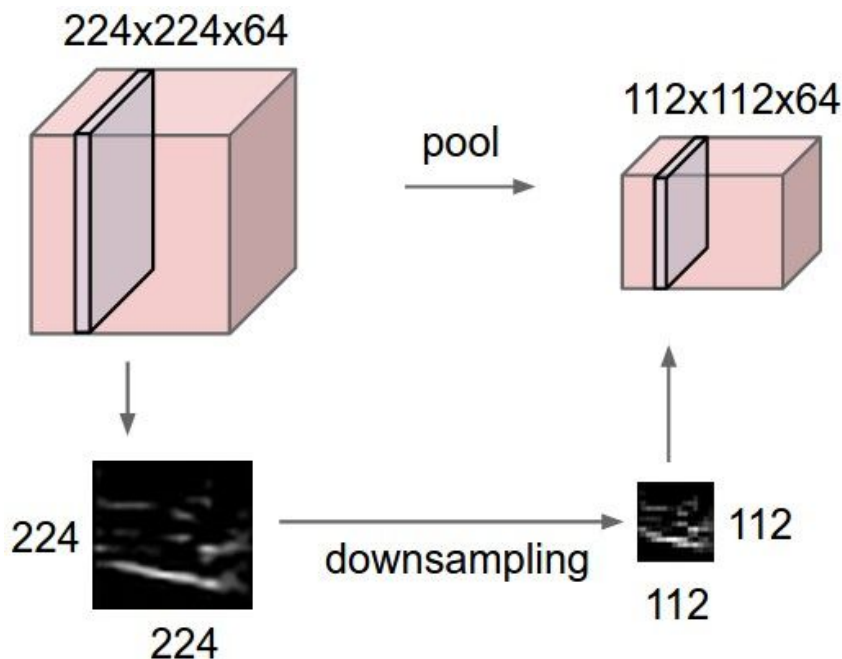


FOUR layers in total: CONV/ReLU/POOL/FC



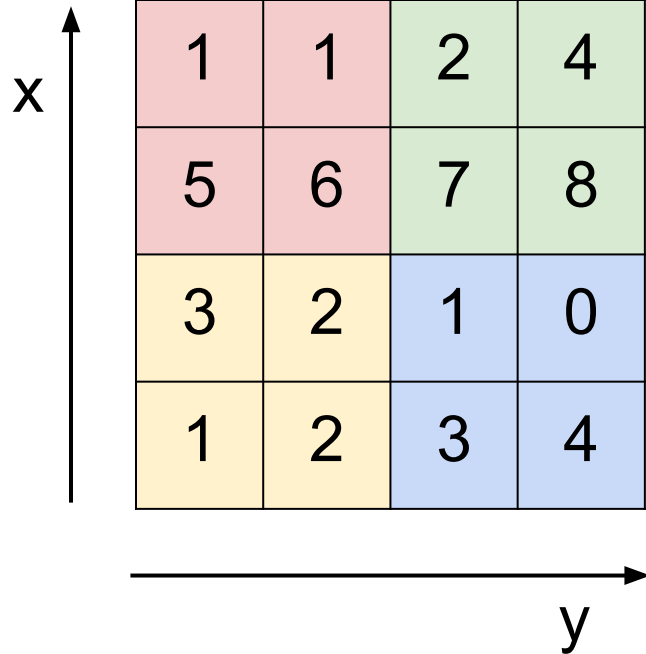
Pooling layer

- makes the representations smaller and more manageable
- operates over each activation map independently:

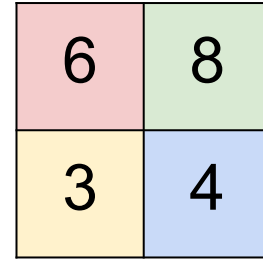


MAX POOLING

Single depth slice

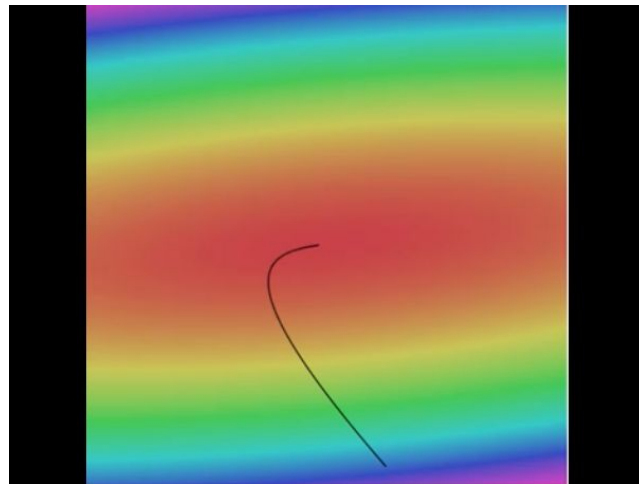
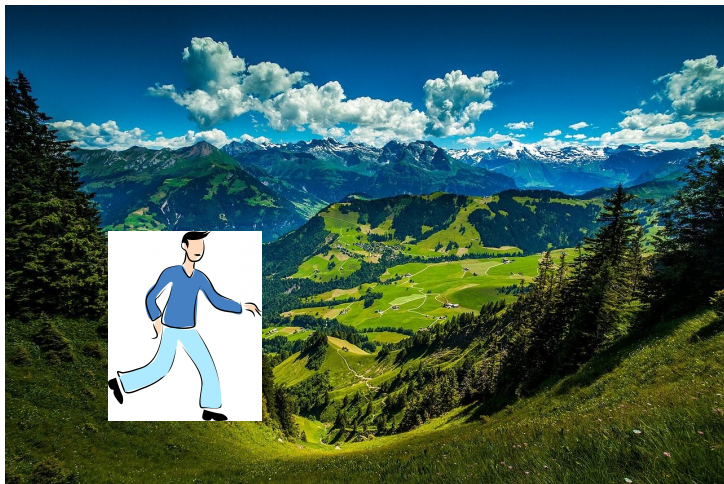


max pool with 2x2 filters
and stride 2



Where we are now...

Learning network parameters through optimization



```
# Vanilla Gradient Descent

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

[Landscape image](#) is [CC0 1.0](#) public domain
[Walking man image](#) is [CC0 1.0](#) public domain

Where we are now...

Mini-batch SGD

Loop:

1. **Sample** a batch of data
2. **Forward** prop it through the graph (network), get loss
3. **Backprop** to calculate the gradients
4. **Update** the parameters using the gradient

There are a few more steps before we start training

1. **One time setup**

activation functions, preprocessing, weight initialization, regularization, gradient checking

2. **Training dynamics**

babysitting the learning process, parameter updates, hyperparameter optimization

3. **Evaluation**

model ensembles, test-time augmentation, transfer learning

Today's agenda

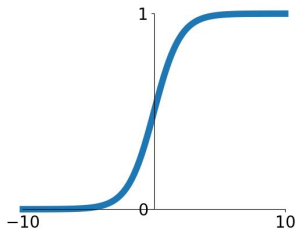
- Activation Functions
- Normalization Layers
- Optimizers

Activation Functions

Activation Functions

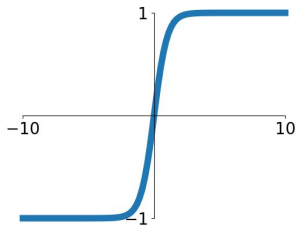
Sigmoid

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



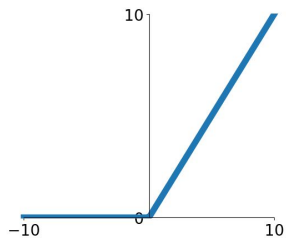
tanh

$$\tanh(x)$$



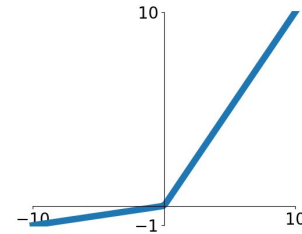
ReLU

$$\max(0, x)$$



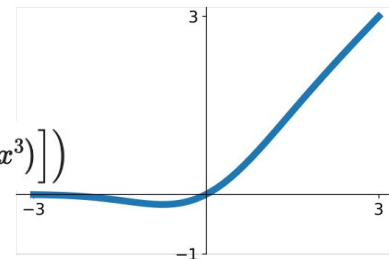
Leaky ReLU

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



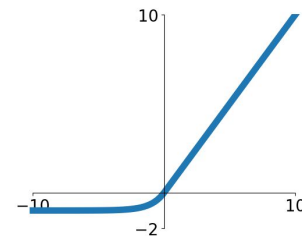
GeLU

$$0.5x \left(1 + \tanh \left[\sqrt{2/\pi} (x + 0.044715x^3) \right] \right)$$



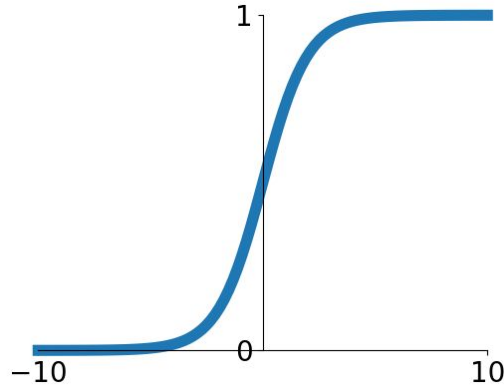
ELU

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



Sigmoid

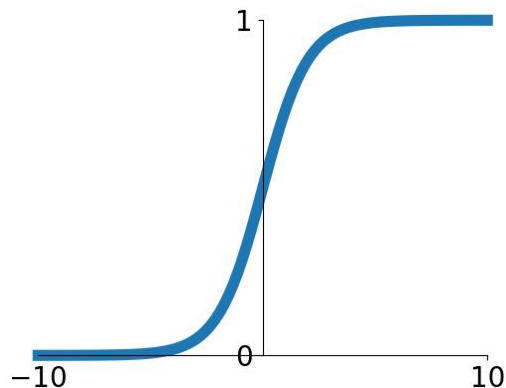
$$\sigma(x) = 1 / (1 + e^{-x})$$



Sigmoid

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

Sigmoid



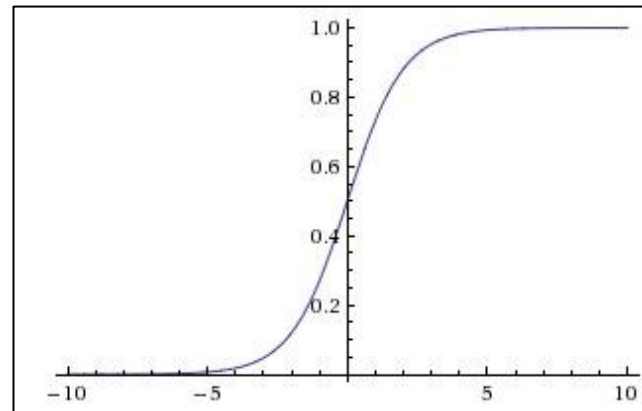
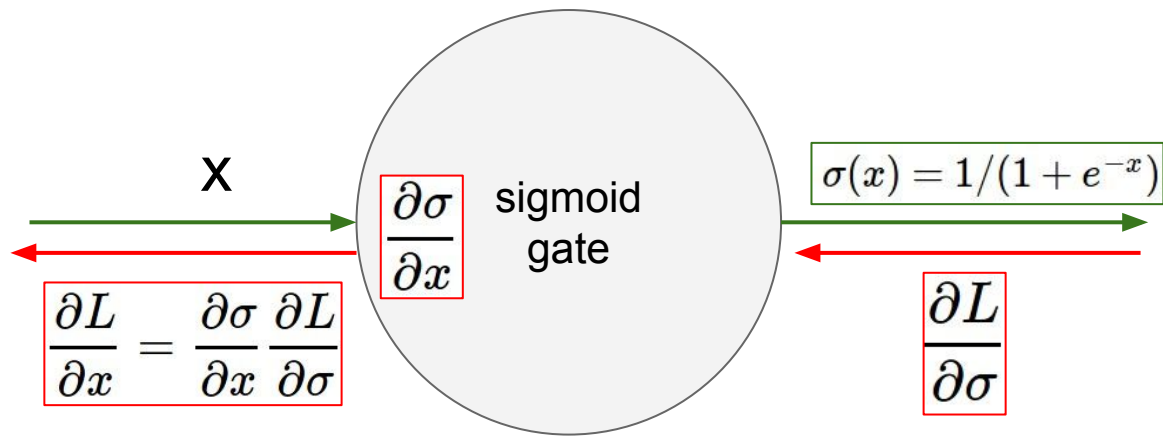
Sigmoid

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

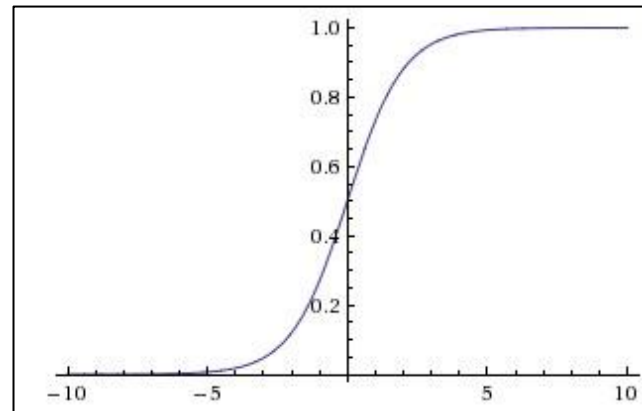
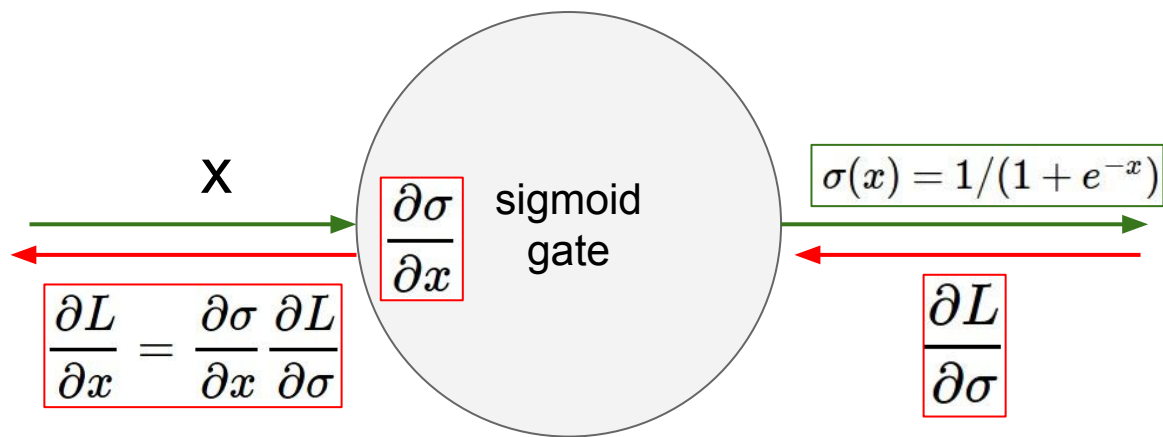
- Squashes numbers to range [0,1]
- Historically popular since they have nice interpretation as a saturating “firing rate” of a neuron

3 problems:

1. Saturated neurons “kill” the gradients

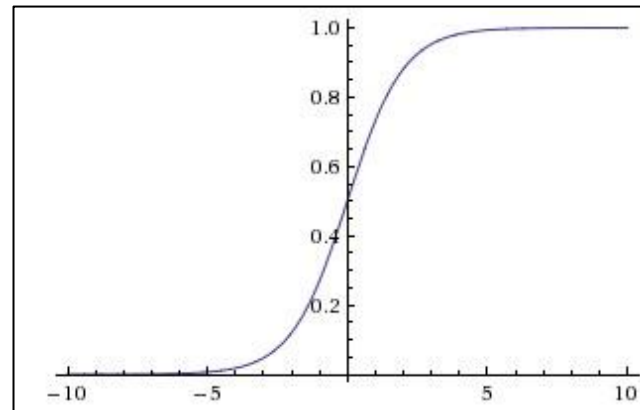
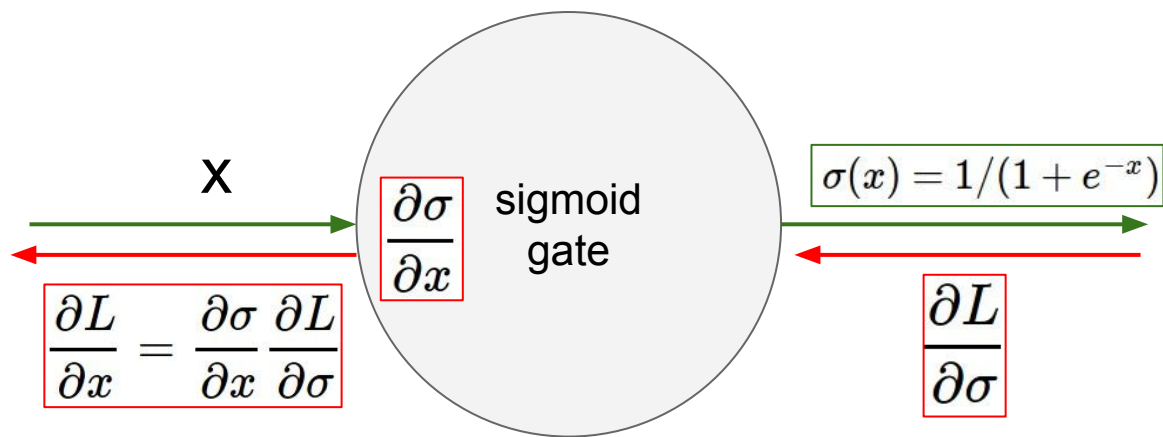


$$\frac{\partial \sigma(x)}{\partial x} = \sigma(x) (1 - \sigma(x))$$



What happens when $x = -10$?

$$\frac{\partial \sigma(x)}{\partial x} = \sigma(x) (1 - \sigma(x))$$

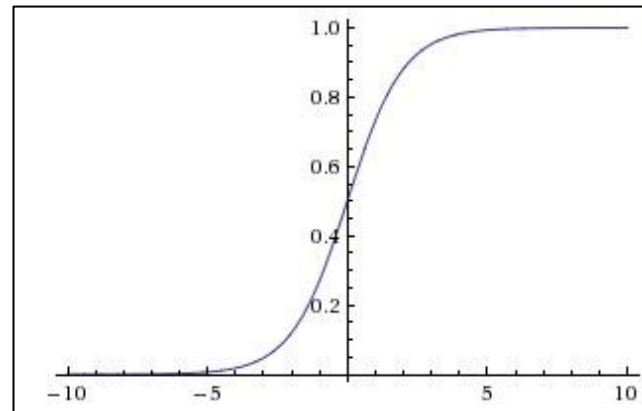
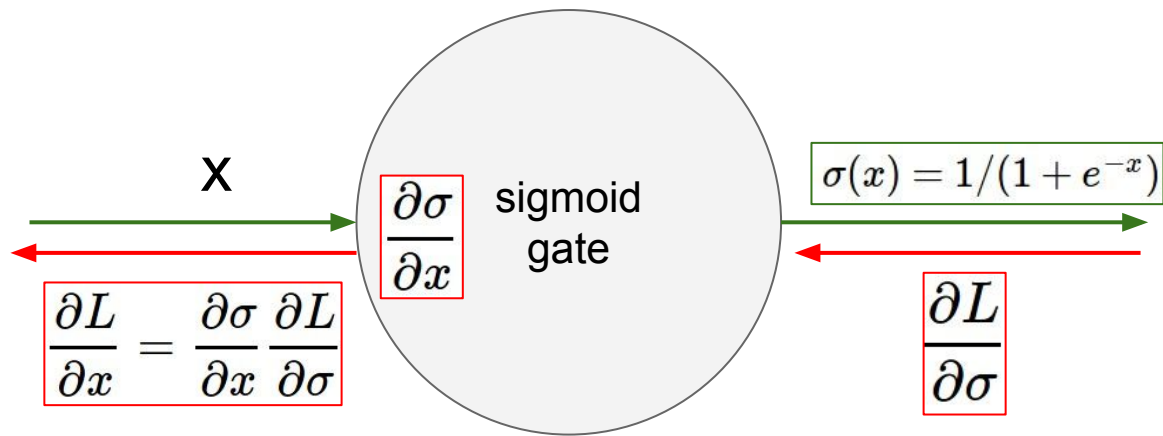


What happens when $x = -10$?

$$\sigma(x) \approx 0$$

$$\frac{\partial \sigma(x)}{\partial x} = \sigma(x) (1 - \sigma(x)) = 0(1 - 0) = 0$$

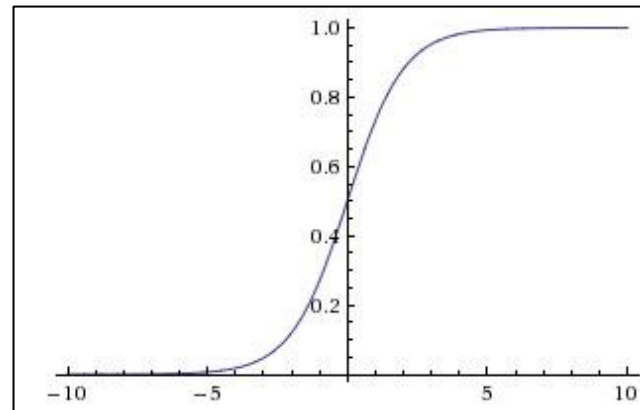
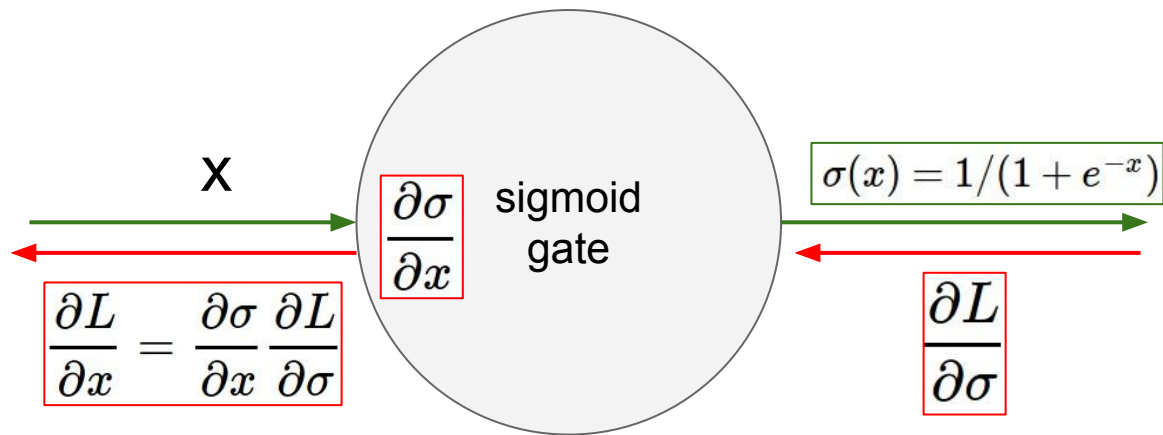
$$\frac{\partial \sigma(x)}{\partial x} = \sigma(x) (1 - \sigma(x))$$



What happens when $x = -10$?

What happens when $x = 0$?

$$\frac{\partial \sigma(x)}{\partial x} = \sigma(x) (1 - \sigma(x))$$

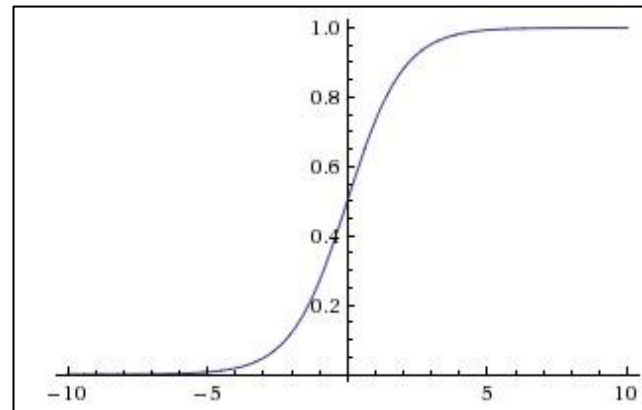
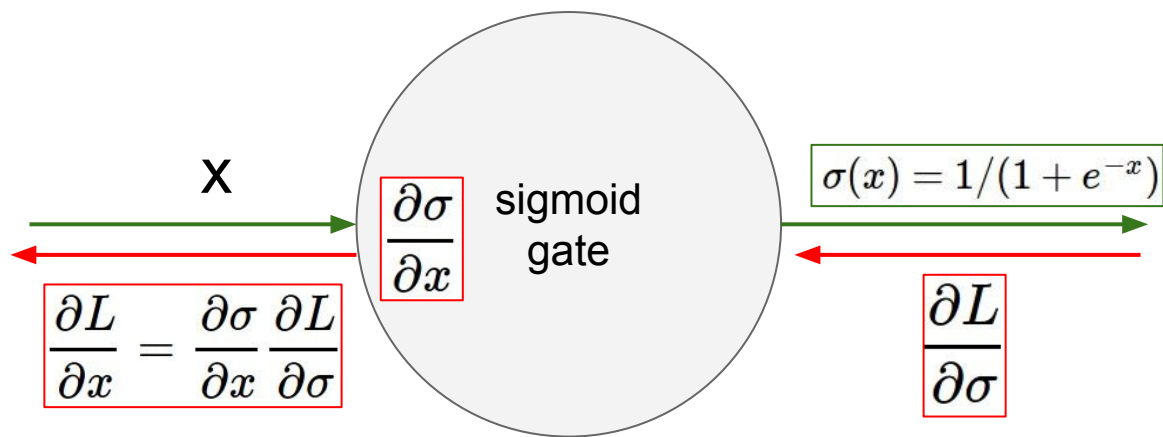


What happens when $x = -10$?

What happens when $x = 0$?

What happens when $x = 10$?

$$\frac{\partial \sigma(x)}{\partial x} = \sigma(x) (1 - \sigma(x))$$



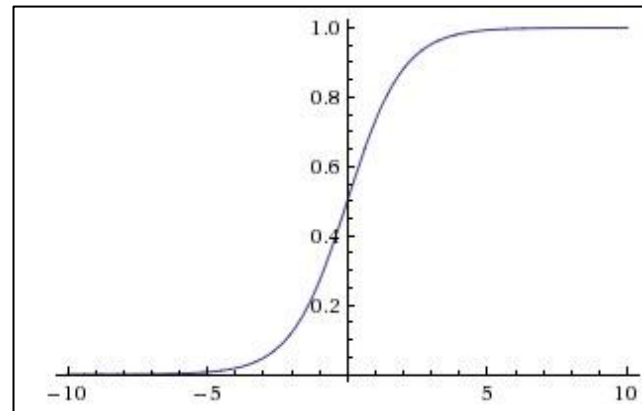
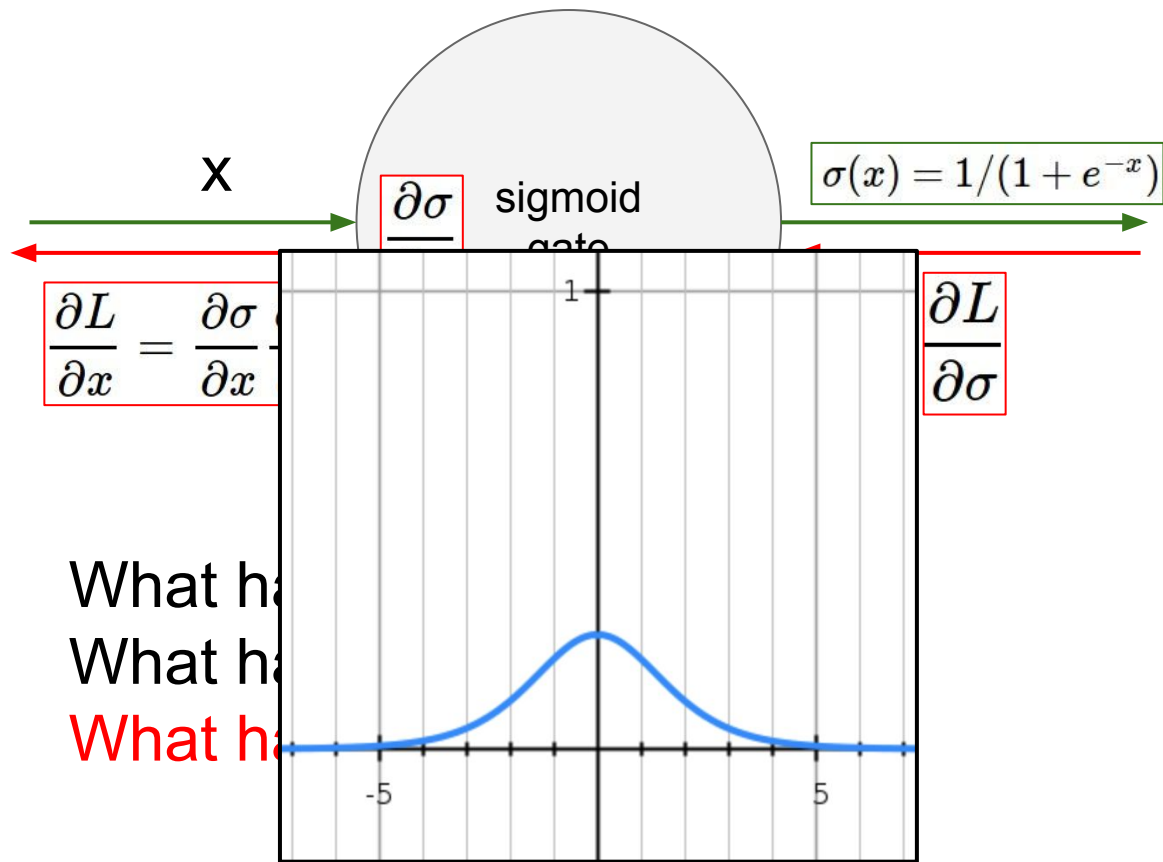
What happens when $x = -10$?

What happens when $x = 0$?

What happens when $x = 10$?

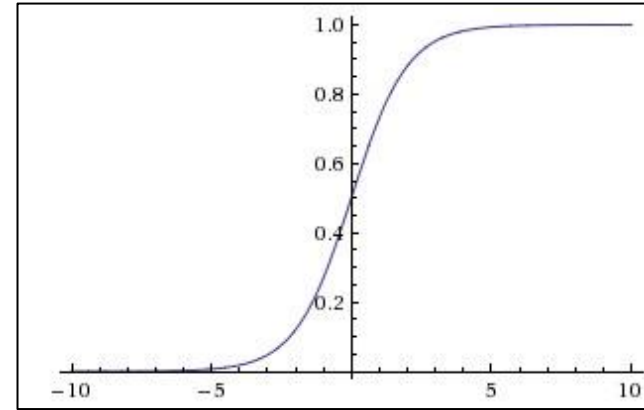
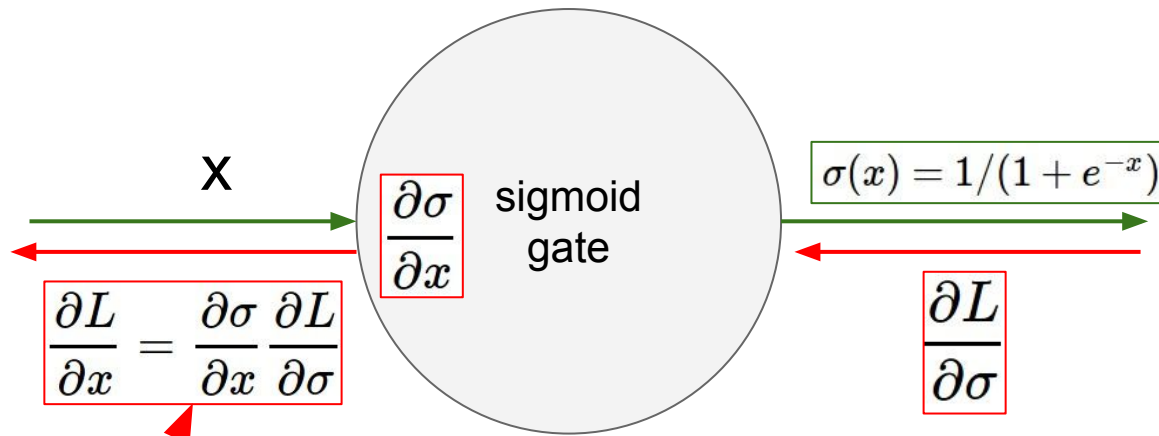
$$\frac{\partial \sigma(x)}{\partial x} = \sigma(x) (1 - \sigma(x))$$

$$\sigma(x) \approx 1 \quad \frac{\partial \sigma(x)}{\partial x} = \sigma(x) (1 - \sigma(x)) = 1(1 - 1) = 0$$



What has
 What has
 What has

$$\frac{\partial \sigma(x)}{\partial x} = \sigma(x) (1 - \sigma(x))$$

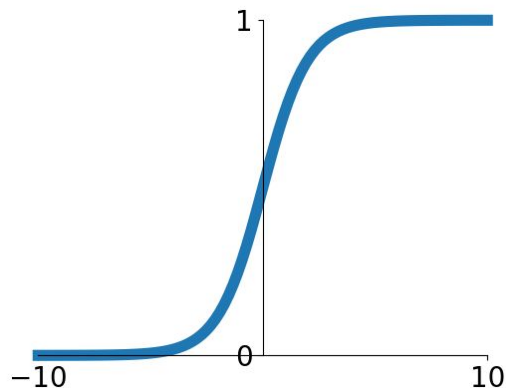


Why is this a problem?

If all the gradients flowing back will be zero and weights will never change

$$\frac{\partial \sigma(x)}{\partial x} = \sigma(x) (1 - \sigma(x))$$

Sigmoid



Sigmoid

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

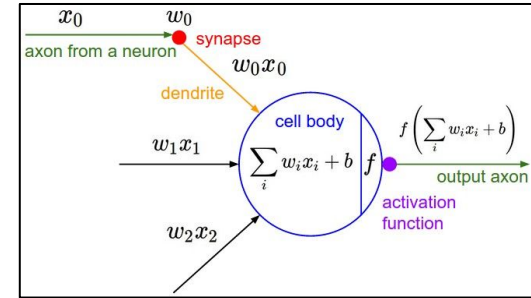
- Squashes numbers to range [0,1]
- Historically popular since they have nice interpretation as a saturating “firing rate” of a neuron

3 problems:

1. Saturated neurons “kill” the gradients
2. Sigmoid outputs are not zero-centered

Consider what happens when sigmoid activations are used

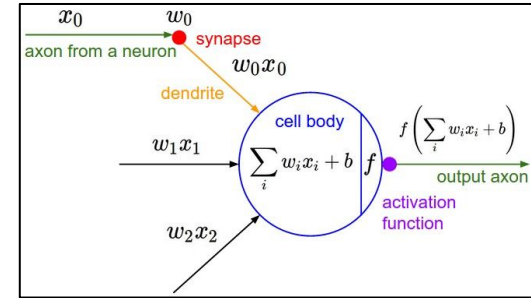
$$f \left(\sum_i w_i x_i + b \right)$$



What can we say about the gradients on \mathbf{w} ?

Consider what happens when sigmoid activations are used

$$f \left(\sum_i w_i x_i + b \right)$$



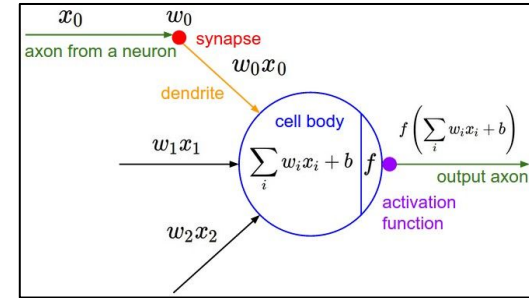
What can we say about the gradients on \mathbf{w} ?

$$\frac{\partial \sigma(x)}{\partial x} = \sigma(x) (1 - \sigma(x))$$

$$\frac{\partial L}{\partial x} = \frac{\partial \sigma}{\partial x} \frac{\partial L}{\partial \sigma}$$

Consider what happens when sigmoid activations are used

$$f \left(\sum_i w_i x_i + b \right)$$



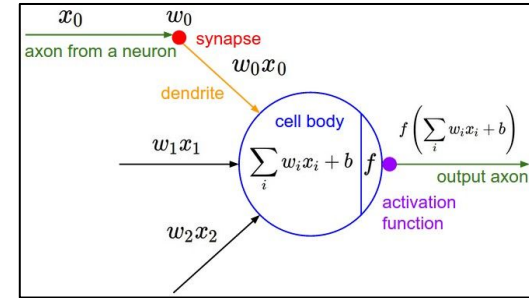
What can we say about the gradients on \mathbf{w} ?

$$\frac{\partial \sigma(x)}{\partial x} = \sigma(x) (1 - \sigma(x)) \quad \frac{\partial L}{\partial x} = \frac{\partial \sigma}{\partial x} \frac{\partial L}{\partial \sigma}$$

$$\frac{\partial L}{\partial w} = \sigma \left(\sum_i w_i x_i + b \right) (1 - \sigma \left(\sum_i w_i x_i + b \right)) x \times \textit{upstream_gradient}$$

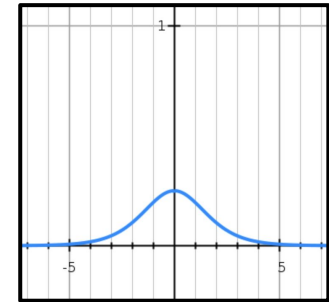
Consider what happens when sigmoid activations are used

$$f\left(\sum_i w_i x_i + b\right)$$



What can we say about the gradients on \mathbf{w} ?

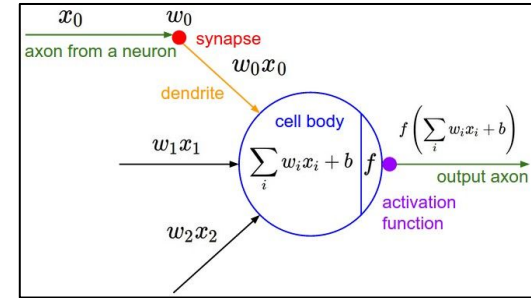
We know that local gradient of sigmoid is always positive



$$\frac{\partial L}{\partial w} = \sigma\left(\sum_i w_i x_i + b\right) \left(1 - \sigma\left(\sum_i w_i x_i + b\right)\right) x \times upstream_gradient$$

Consider what happens when sigmoid activations are used

$$f\left(\sum_i w_i x_i + b\right)$$



What can we say about the gradients on \mathbf{w} ?

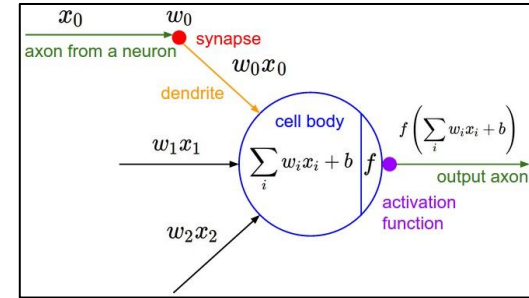
We know that local gradient of sigmoid is always positive

Input x is always positive (because it is output from previous layer's sigmoid)

$$\frac{\partial L}{\partial w} = \sigma\left(\sum_i w_i x_i + b\right) (1 - \sigma\left(\sum_i w_i x_i + b\right)) x \times \text{upstream_gradient}$$

All weight updates are correlated

$$f \left(\sum_i w_i x_i + b \right)$$



What can we say about the gradients on \mathbf{w} ?

We know that local gradient of sigmoid is always positive

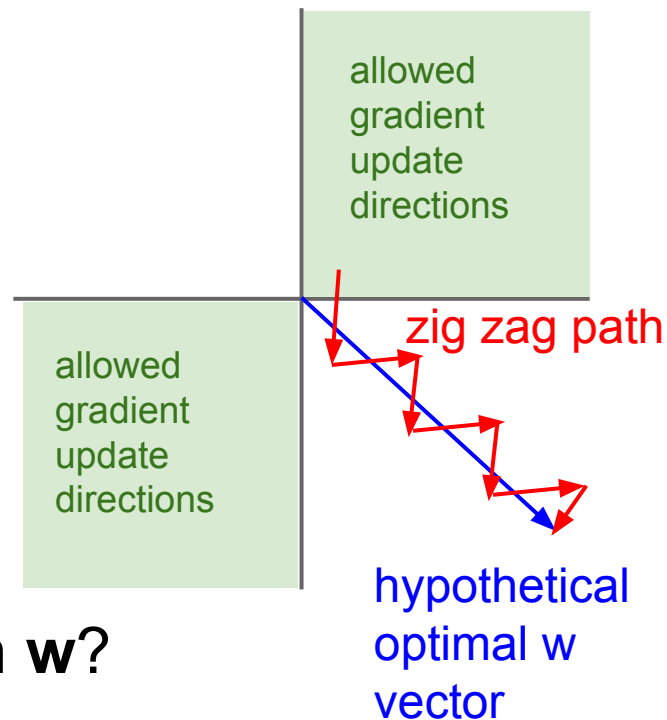
Input x is always positive (because it is output from previous layer's sigmoid)

So!! Sign of gradient **for all w_i** is the same as the sign of upstream scalar gradient!

$$\frac{\partial L}{\partial w} = \sigma \left(\sum_i w_i x_i + b \right) (1 - \sigma \left(\sum_i w_i x_i + b \right)) x \times \text{upstream_gradient}$$

Because all weights are updated in the same direction, it is harder to optimize.

$$f \left(\sum_i w_i x_i + b \right)$$

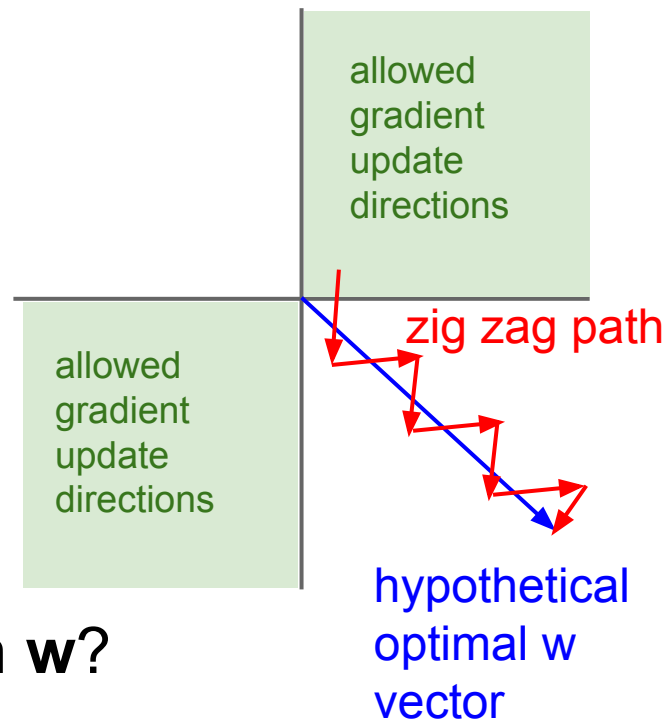


What can we say about the gradients on \mathbf{w} ?

Always all positive or all negative :(

Because all weights are updated in the same direction, it is harder to optimize.

$$f \left(\sum_i w_i x_i + b \right)$$

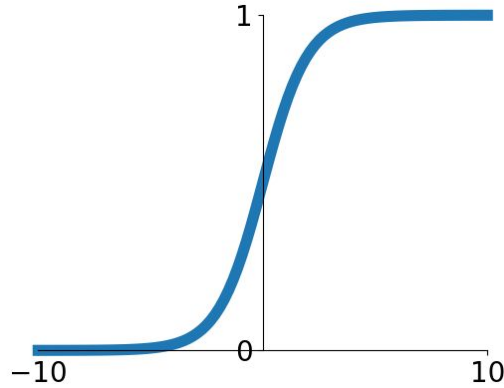


What can we say about the gradients on \mathbf{w} ?

Always all positive or all negative :(

(For a single element! Minibatches help)

Sigmoid



Sigmoid

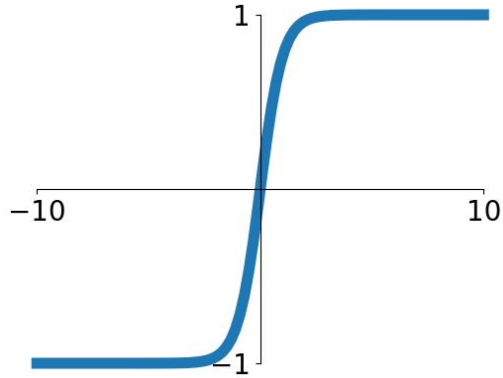
$$\sigma(x) = 1/(1 + e^{-x})$$

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

3 problems:

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

Tanh

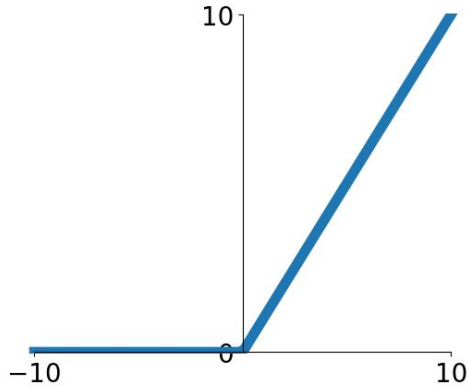


tanh(x)

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

[LeCun et al., 1991]

ReLU

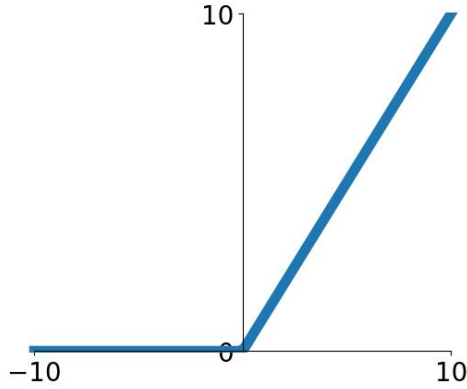


- Computes $f(x) = \max(0, x)$
- Does not saturate (in +region)
- Very computationally efficient
- Converges much faster than sigmoid/tanh in practice (e.g. 6x)

ReLU
(Rectified Linear Unit)

[Krizhevsky et al., 2012]

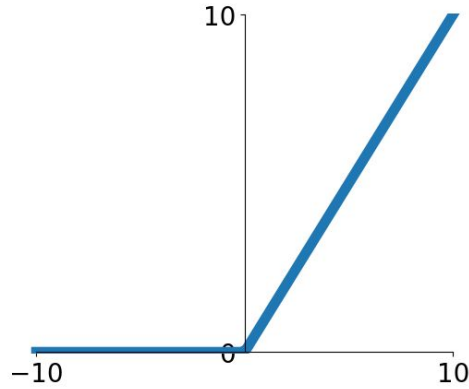
ReLU



ReLU
(Rectified Linear Unit)

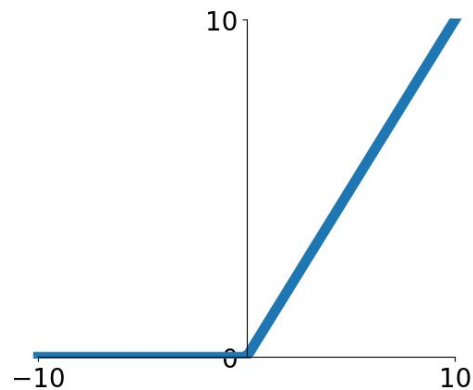
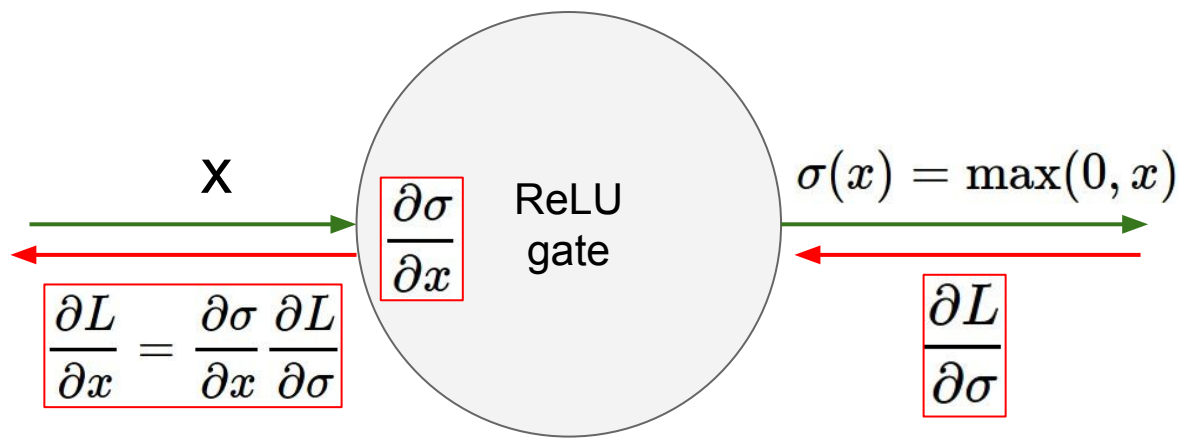
- Computes $f(x) = \max(0, x)$
- Does not saturate (in +region)
- Very computationally efficient
- Converges much faster than sigmoid/tanh in practice (e.g. 6x)
- Not zero-centered output

ReLU



ReLU
(Rectified Linear Unit)

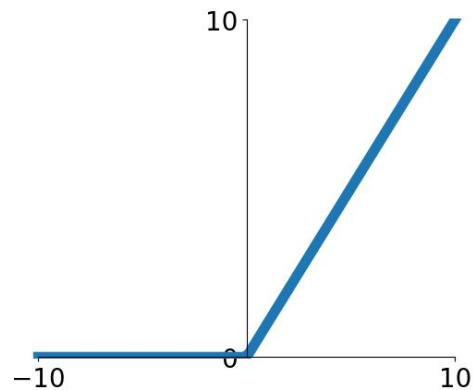
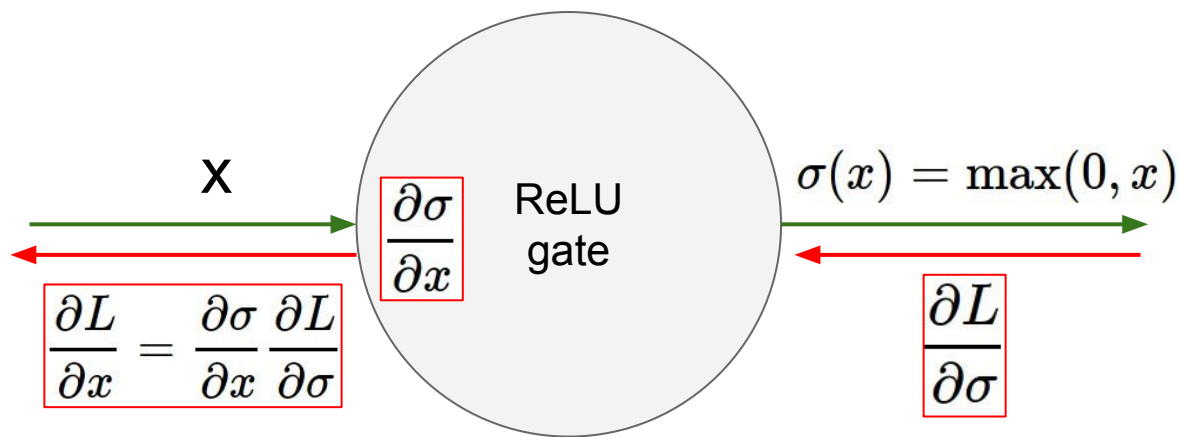
- Computes $f(x) = \max(0, x)$
 - Does not saturate (in +region)
 - Very computationally efficient
 - Converges much faster than sigmoid/tanh in practice (e.g. 6x)
 - Not zero-centered output
 - An annoyance:
- hint: what is the gradient when $x < 0$?



What happens when $x = -10$?

What happens when $x = 0$?

What happens when $x = 10$?



What happens when $x = -10$?

What happens when $x = 0$?

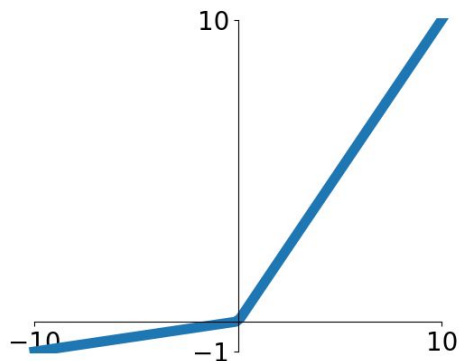
What happens when $x = 10$?

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

Leaky ReLU

[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 see disappearing gradients.**

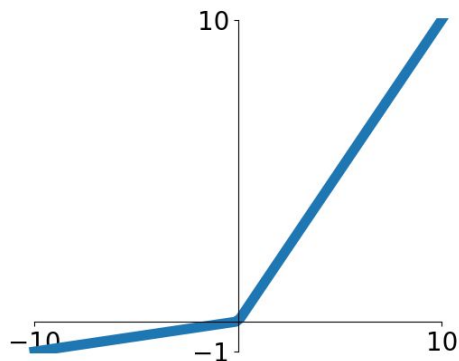
Leaky ReLU

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

Leaky ReLU

[Mass et al., 2013]

[He et al., 2015]



Leaky ReLU

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

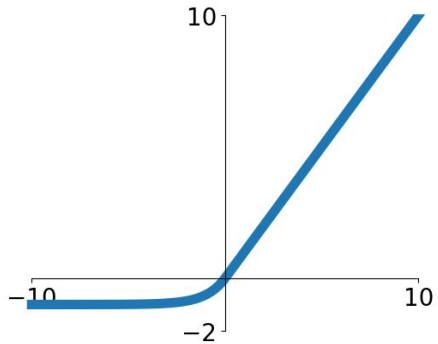
- Does not saturate
- Computationally efficient
- Converges much faster than sigmoid/tanh in practice! (e.g. 6x)
- **will not see disappearing gradients**

Parametric Rectifier (PReLU)

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

backprop into α
(parameter)

Exponential Linear Units (ELU)

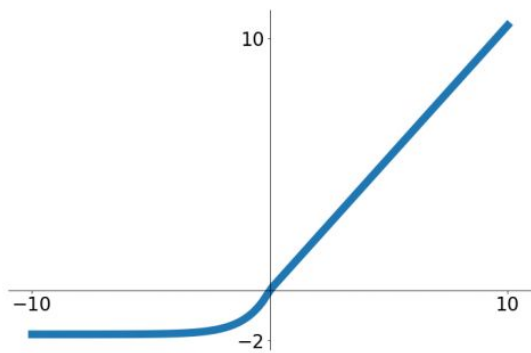


- All benefits of ReLU
- Closer to zero mean outputs
- Computation requires $\exp()$
- Negative saturation can kill gradients for large negative

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

(Alpha default = 1)

Scaled Exponential Linear Units (SELU)



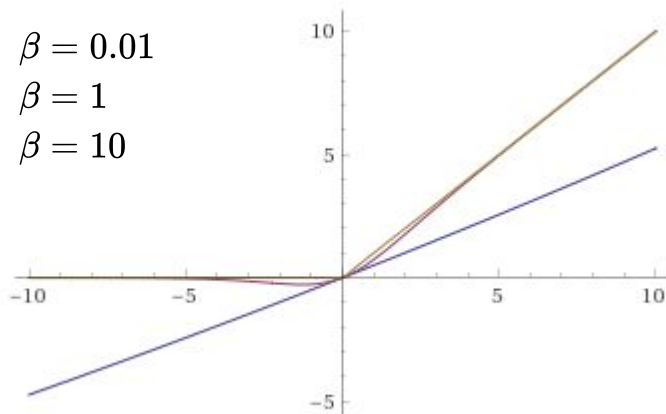
- Scaled version of ELU that works better for deep networks
- “Self-normalizing” property;
- Can train deep SELU networks without BatchNorm
 - (will discuss more later)

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

$\alpha = 1.6733, \lambda = 1.0507$

Swish

[Ramachandran et al. 2018]



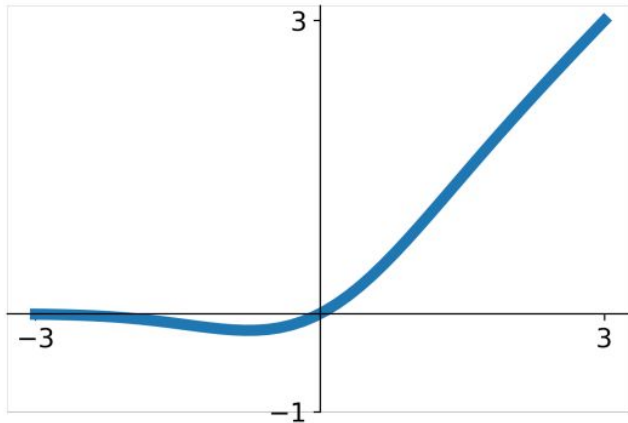
- They trained a neural network to generate and test out different non-linearities.
- Swish outperformed all other options for CIFAR-10 accuracy

Swish

$$f(x) = x\sigma(\beta x)$$

GeLU

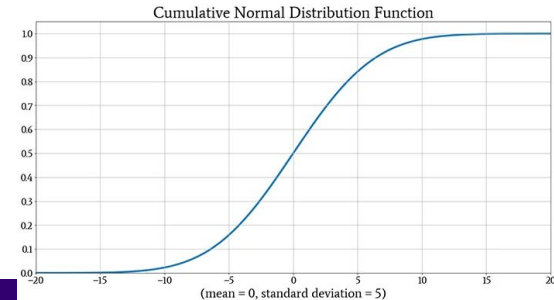
Gaussian Error Linear Units



$X \sim N(0, 1)$

$$\text{gelu}(x) = xP(X \leq x) = \frac{x}{2} (1 + \text{erf}(x/\sqrt{2}))$$

- Idea: combine the best parts of tanh/sigmoid and relu.
- GELU transitions more smoothly
- It weights each input according to the Gaussian (Normal) CDF



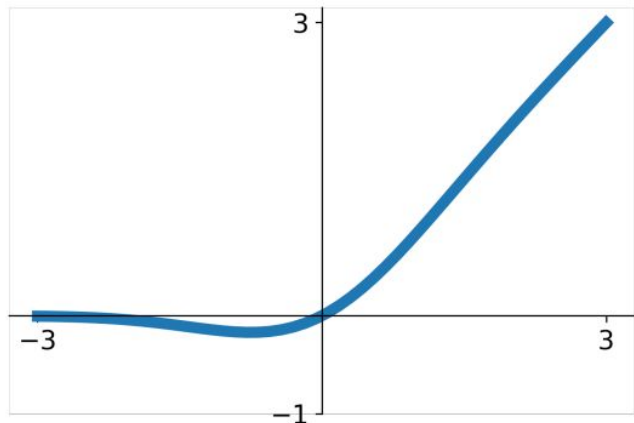
GeLU

[Hendrycks and Gimpel, Gaussian Error Linear Units (GELUs), 2016]

Mathematically, $\text{GELU}(x) = x \times \Phi(x)$

where $\Phi(x)$ is the cumulative distribution function (CDF) of a standard normal distribution $N(0, 1)$

$$\Phi(x) = \frac{1}{2} \left[1 + \text{erf} \left(\frac{x}{\sqrt{2}} \right) \right]$$

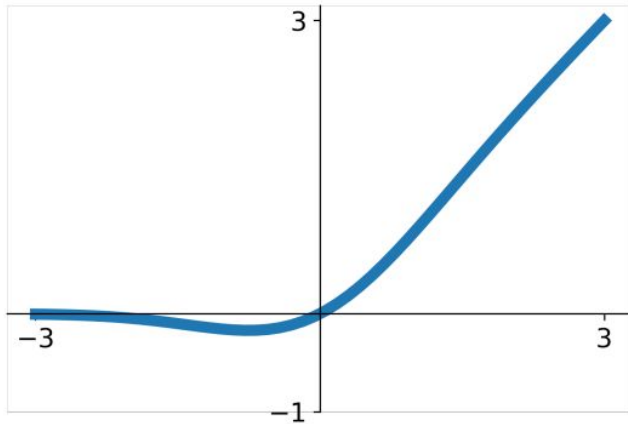


$$\text{GELU}(x) \approx 0.5 x \left[1 + \tanh \left(\sqrt{\frac{2}{\pi}} (x + 0.044715 x^3) \right) \right]$$

← It is approximated as

Activation Functions

GeLU



$X \sim N(0, 1)$

$$\text{gelu}(x) = xP(X \leq x) = \frac{x}{2} (1 + \text{erf}(x/\sqrt{2}))$$

- More continuous version of ReLU. It scales the input x by a probability factor dependent on x
- Avoid zero gradients
- Empirically stable training
- Very common in Transformers (BERT, GPT, ViT)

TLDR: In practice:

- Use **GeLU** when using transformers
- Play around with other variants of **ReLU**.
 - Be careful with your learning rates
- Don't use **sigmoid** or **tanh**

Batch Normalization

Batch Normalization

[Ioffe and Szegedy, 2015]

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

consider a batch of activations at some layer. To make each dimension zero-mean unit-variance, apply:

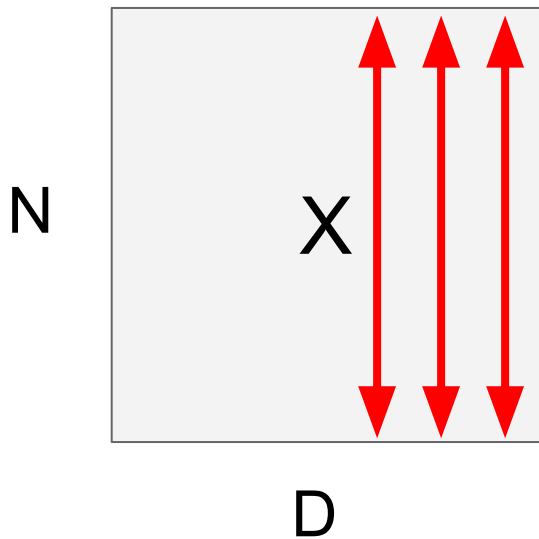
$$\hat{x}^{(k)} = \frac{x^{(k)} - \mathbb{E}[x^{(k)}]}{\sqrt{\text{Var}[x^{(k)}]}}$$

this is a vanilla
differentiable function...

Batch Normalization

[Ioffe and Szegedy, 2015]

Input: $x : N \times D$



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

Per-channel mean,
shape is D

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

Per-channel var,
shape is D

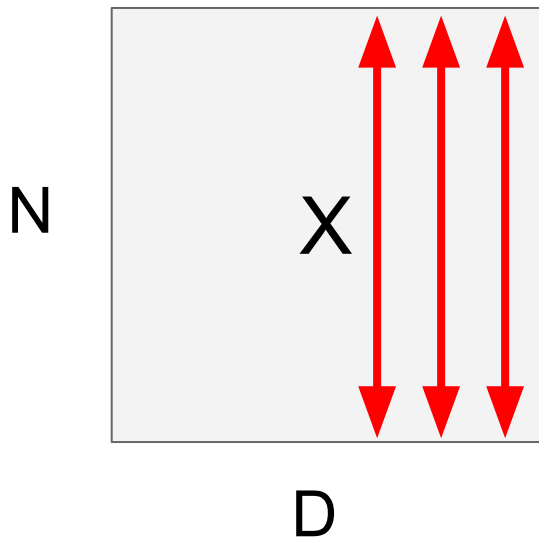
$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}$$

Normalized x,
Shape is N x D

Batch Normalization

[Ioffe and Szegedy, 2015]

Input: $x : N \times D$



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

Per-channel mean,
shape is D

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

Per-channel var,
shape is D

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

Normalized x,
Shape is N x D

Problem: What if zero-mean, unit variance is too hard of a constraint?

Batch Normalization

[Ioffe and Szegedy, 2015]

Input: $x : N \times D$

Learnable scale and shift parameters:

$$\gamma, \beta : D$$

Learning $\gamma = \sigma$,
 $\beta = \mu$ will recover the
identity function!

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

Per-channel mean,
shape is D

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

Per-channel var,
shape is D

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

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:**

$$\gamma, \beta : D$$

Learning $\gamma = \sigma$,
 $\beta = \mu$ will recover the
identity function!

$$\mu_j = \frac{1}{N} \sum_{i=1}^N x_{i,j} \quad \text{Per-channel mean, shape is D}$$
$$\sigma_j^2 = \frac{1}{N} \sum_{i=1}^N (x_{i,j} - \mu_j)^2 \quad \text{Per-channel var, shape is D}$$

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}} \quad \text{Normalized x, Shape is N x D}$$

$$y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j \quad \text{Output, Shape is N x D}$$

Batch Normalization: Test-Time

Input: $x : N \times D$

$$\mu_j = \text{(Running) average of values seen during training}$$

Per-channel mean,
shape is D

Learnable scale and shift parameters:

$$\gamma, \beta : D$$

$$\sigma_j^2 = \text{(Running) average of values seen during training}$$

Per-channel var,
shape is D

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

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

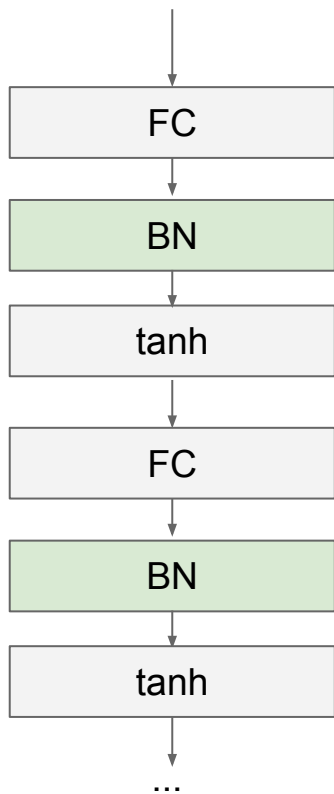
Normalized x,
Shape is N x D

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

Output,
Shape is N x D

Batch Normalization

[Ioffe and Szegedy, 2015]

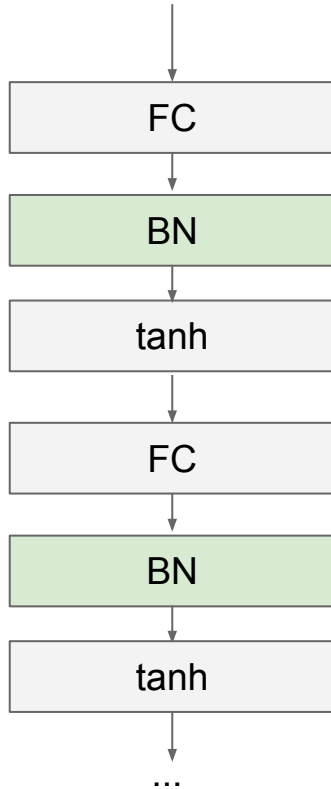


Usually inserted after Fully Connected or Convolutional layers, and before nonlinearity.

$$\hat{x}^{(k)} = \frac{x^{(k)} - \mathbb{E}[x^{(k)}]}{\sqrt{\text{Var}[x^{(k)}]}}$$

Batch Normalization

[Ioffe and Szegedy, 2015]



- Makes deep networks **much** easier to train!
- Improves gradient flow
- Allows higher learning rates, faster convergence
- Networks become more robust to initialization
- Acts as regularization during training
- Zero overhead at test-time: can be fused with conv!
- **Behaves differently during training and testing: this is a very common source of debugging!**

Batch Normalization for convolutions

Batch Normalization for
fully-connected networks

$$\mathbf{x} : \mathbf{N} \times \mathbf{D}$$

Normalize 

$$\boldsymbol{\mu}, \boldsymbol{\sigma} : 1 \times \mathbf{D}$$

$$\boldsymbol{\gamma}, \boldsymbol{\beta} : 1 \times \mathbf{D}$$

$$\mathbf{y} = \boldsymbol{\gamma} (\mathbf{x} - \boldsymbol{\mu}) / \boldsymbol{\sigma} + \boldsymbol{\beta}$$

Batch Normalization for
convolutional networks
(Spatial Batchnorm, BatchNorm2D)

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

Normalize   

$$\boldsymbol{\mu}, \boldsymbol{\sigma} : 1 \times \mathbf{C} \times 1 \times 1$$

$$\boldsymbol{\gamma}, \boldsymbol{\beta} : 1 \times \mathbf{C} \times 1 \times 1$$

$$\mathbf{y} = \boldsymbol{\gamma} (\mathbf{x} - \boldsymbol{\mu}) / \boldsymbol{\sigma} + \boldsymbol{\beta}$$

Layer Normalization

Layer Normalization for MLPs

Batch Normalization for fully-connected networks

$$\mathbf{x} : \mathbf{N} \times \mathbf{D}$$

Normalize



$$\boldsymbol{\mu}, \boldsymbol{\sigma} : \mathbf{1} \times \mathbf{D}$$

$$\boldsymbol{\gamma}, \boldsymbol{\beta} : \mathbf{1} \times \mathbf{D}$$

$$\mathbf{y} = \boldsymbol{\gamma} (\mathbf{x} - \boldsymbol{\mu}) / \boldsymbol{\sigma} + \boldsymbol{\beta}$$

Layer Normalization for fully-connected networks
Same behavior at train and test!
Often used in transformers

$$\mathbf{x} : \mathbf{N} \times \mathbf{D}$$

Normalize



$$\boldsymbol{\mu}, \boldsymbol{\sigma} : \mathbf{N} \times \mathbf{1}$$

$$\boldsymbol{\gamma}, \boldsymbol{\beta} : \mathbf{1} \times \mathbf{D}$$

$$\mathbf{y} = \boldsymbol{\gamma} (\mathbf{x} - \boldsymbol{\mu}) / \boldsymbol{\sigma} + \boldsymbol{\beta}$$

Ba, Kiros, and Hinton, "Layer Normalization", arXiv 2016

Instance Normalization

Instance Normalization for Convolutions

Batch Normalization for convolutional networks

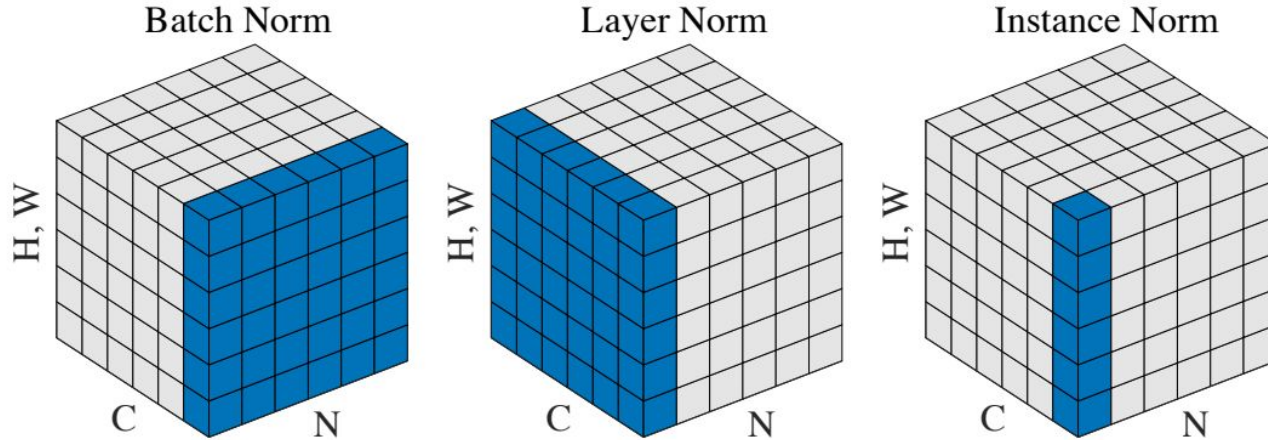
$$\begin{array}{l} \mathbf{x} : \mathbf{N} \times \mathbf{C} \times \mathbf{H} \times \mathbf{W} \\ \text{Normalize} \quad \downarrow \quad \downarrow \quad \downarrow \\ \boldsymbol{\mu}, \boldsymbol{\sigma} : \mathbf{1} \times \mathbf{C} \times \mathbf{1} \times \mathbf{1} \\ \boldsymbol{\gamma}, \boldsymbol{\beta} : \mathbf{1} \times \mathbf{C} \times \mathbf{1} \times \mathbf{1} \\ \mathbf{y} = \boldsymbol{\gamma} (\mathbf{x} - \boldsymbol{\mu}) / \boldsymbol{\sigma} + \boldsymbol{\beta} \end{array}$$

Instance Normalization for convolutional networks
Same behavior at train / test!

$$\begin{array}{l} \mathbf{x} : \mathbf{N} \times \mathbf{C} \times \mathbf{H} \times \mathbf{W} \\ \text{Normalize} \quad \quad \quad \downarrow \quad \downarrow \\ \boldsymbol{\mu}, \boldsymbol{\sigma} : \mathbf{N} \times \mathbf{C} \times \mathbf{1} \times \mathbf{1} \\ \boldsymbol{\gamma}, \boldsymbol{\beta} : \mathbf{1} \times \mathbf{C} \times \mathbf{1} \times \mathbf{1} \\ \mathbf{y} = \boldsymbol{\gamma} (\mathbf{x} - \boldsymbol{\mu}) / \boldsymbol{\sigma} + \boldsymbol{\beta} \end{array}$$

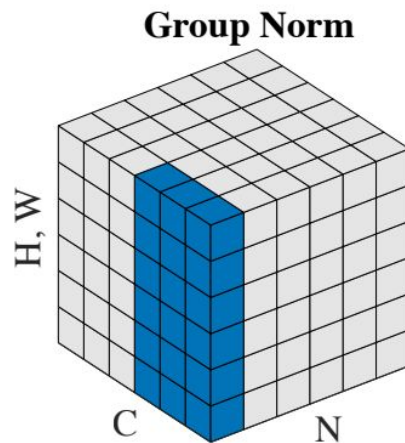
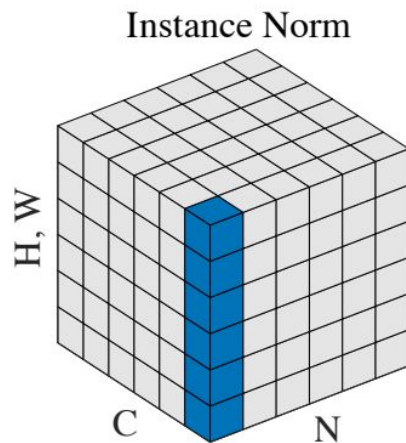
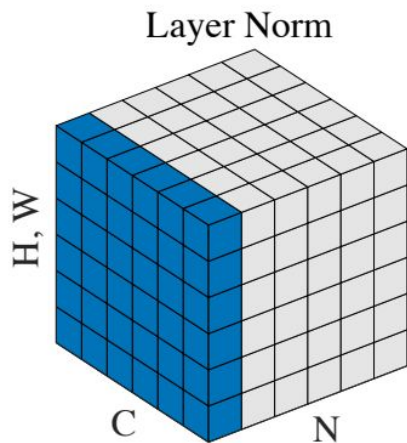
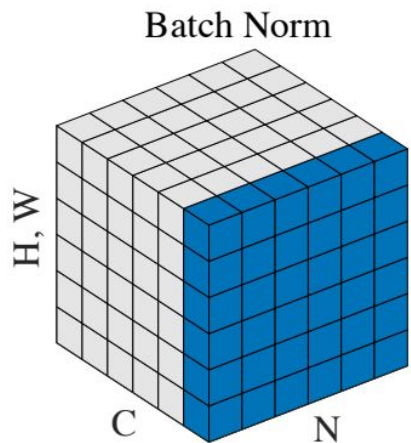
Ulyanov et al, Improved Texture Networks: Maximizing Quality and Diversity in Feed-forward Stylization and Texture Synthesis, CVPR 2017

Comparison of Normalization Layers



Wu and He, "Group Normalization", ECCV 2018

Group Normalization



Wu and He, "Group Normalization", ECCV 2018

(Fancier) Optimizers

Can we do better than SGD?

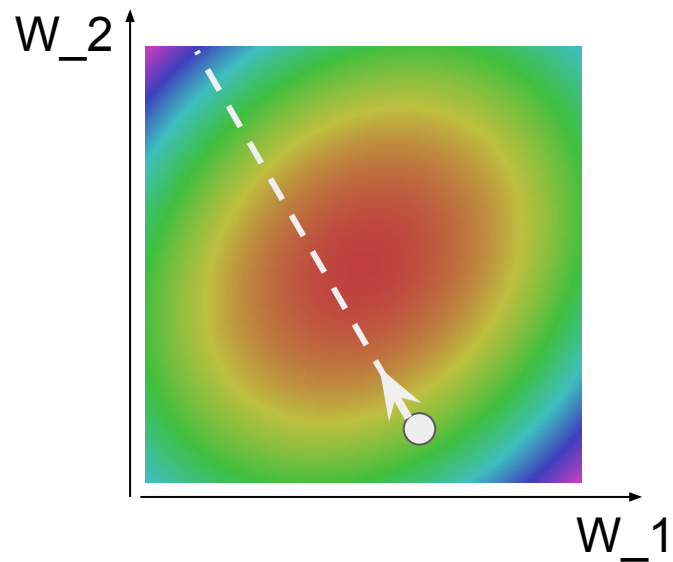
Vanilla SGD

```
# Vanilla Gradient Descent
```

```
while True:
```

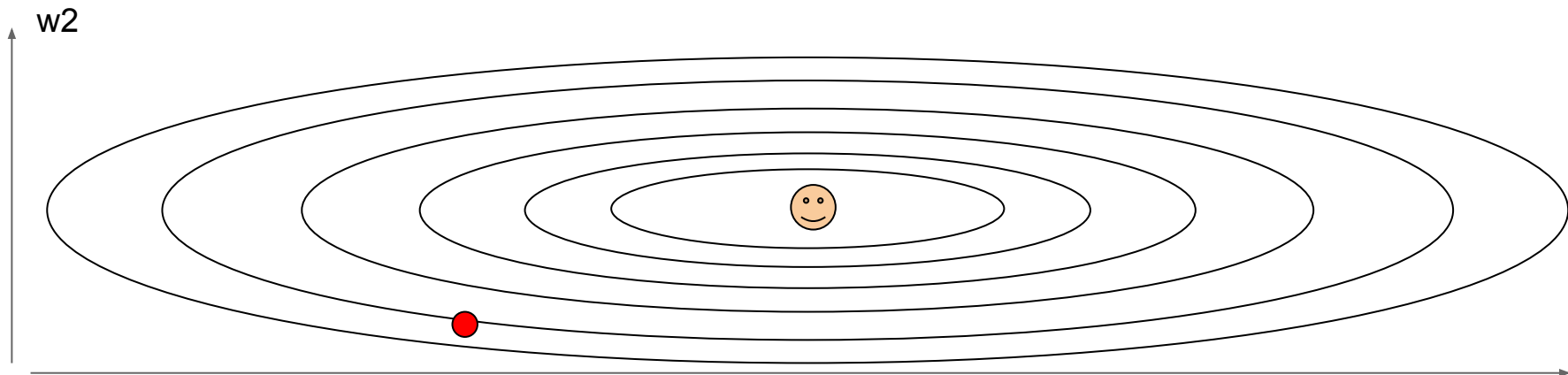
```
    weights_grad = evaluate_gradient(loss_fun, data, weights)
```

```
    weights += - step_size * weights_grad # perform parameter update
```



Optimization: Problem #1 with SGD

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

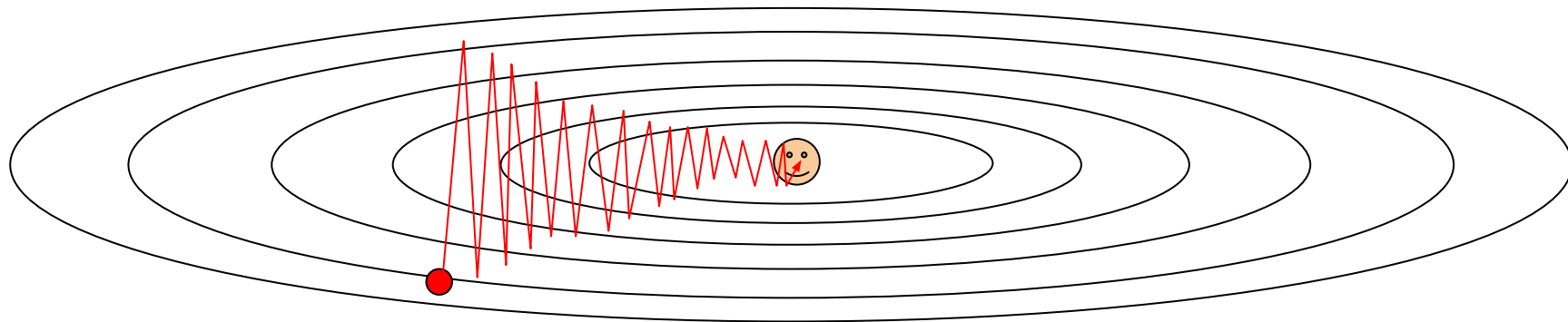


Aside: You can detect this situation by calculating the **condition number**, w_1 which is the ratio of largest to smallest singular value of the Hessian matrix
Large ratio implies high condition number implies the loss is uneven

Optimization: Problem #1 with SGD

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

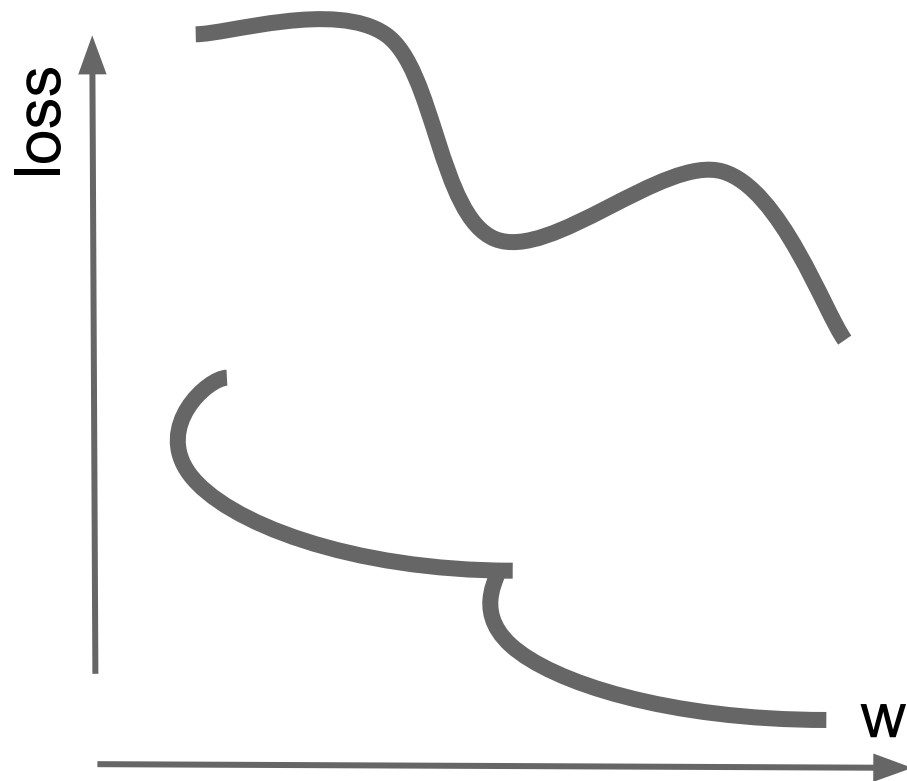
Very slow progress along shallow dimension, jitter along steep direction



Loss function has high **condition number**: ratio of largest to smallest singular value of the Hessian matrix is large

Optimization: Problem #2 with SGD

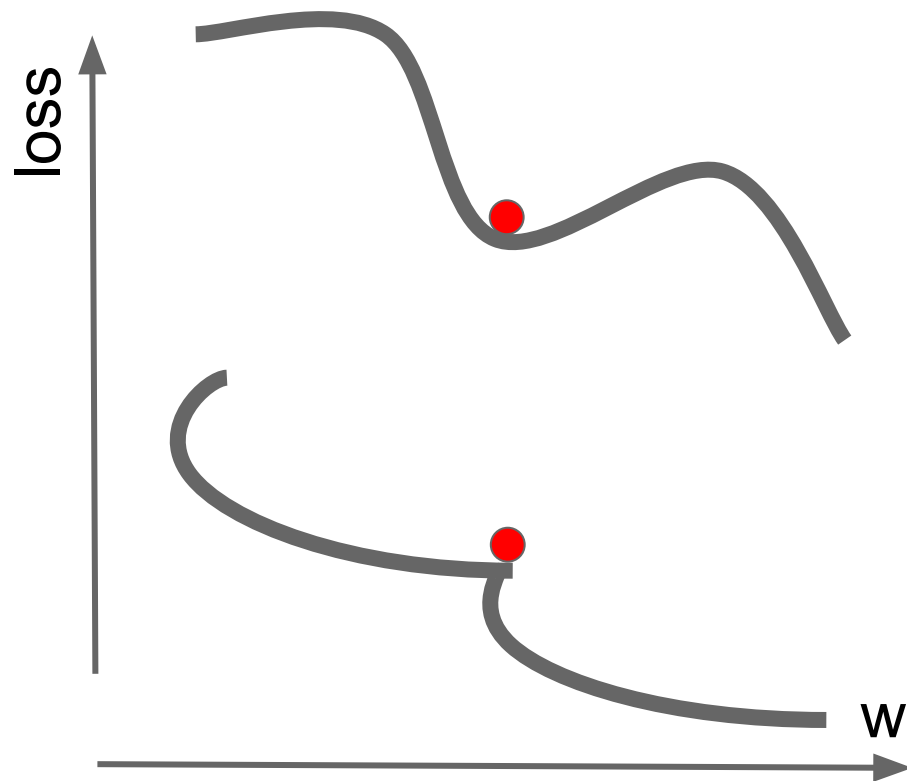
What if the loss function has a **local minima** or **saddle point**?



Optimization: Problem #2 with SGD

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

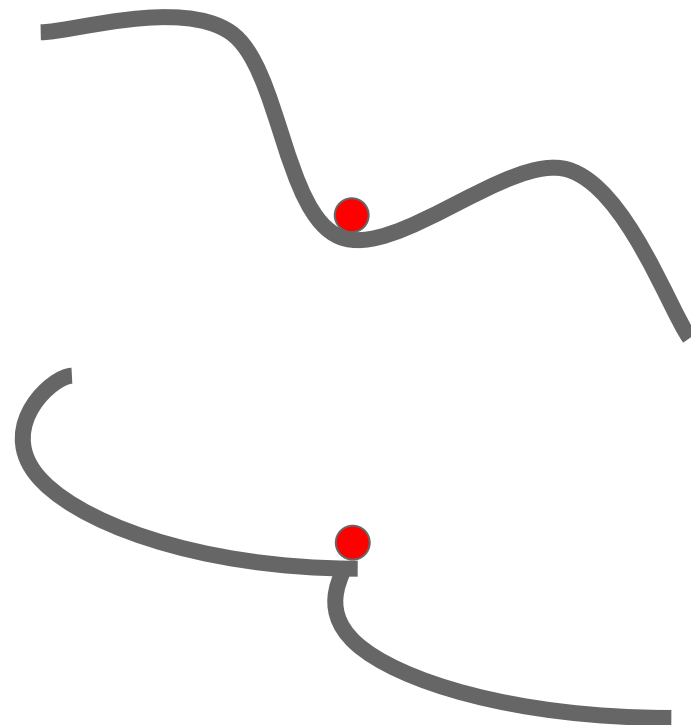
Zero gradient,
gradient descent
gets stuck



Optimization: Problem #2 with SGD

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

Saddle points much more common in high dimension



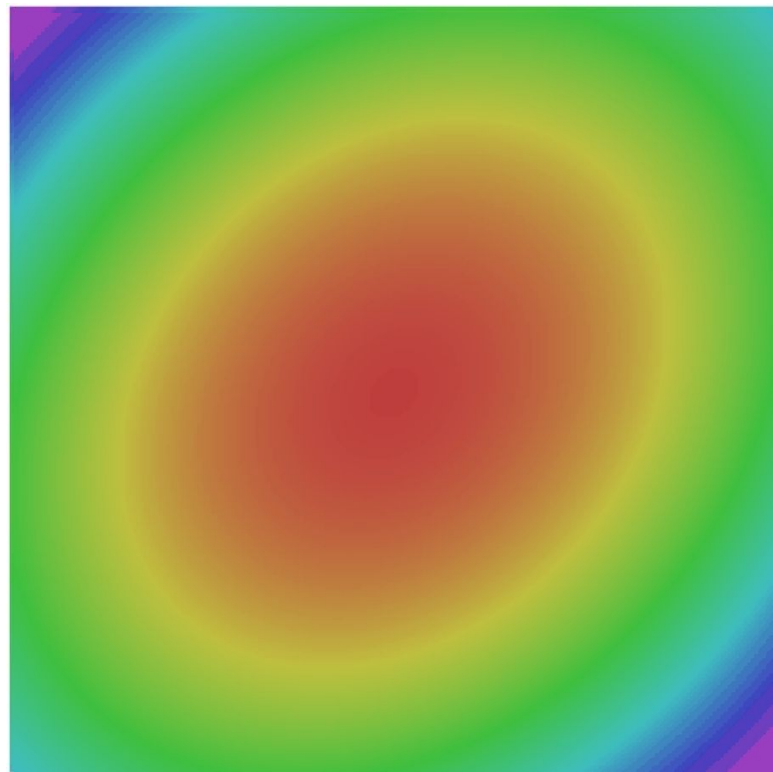
Dauphin et al, "Identifying and attacking the saddle point problem in high-dimensional non-convex optimization", NIPS 2014

Optimization: Problem #3 with SGD

Our gradients come from minibatches so they can be noisy!

$$L(W) = \frac{1}{N} \sum_{i=1}^N L_i(x_i, y_i, W)$$

$$\nabla_W L(W) = \frac{1}{N} \sum_{i=1}^N \nabla_W L_i(x_i, y_i, W)$$



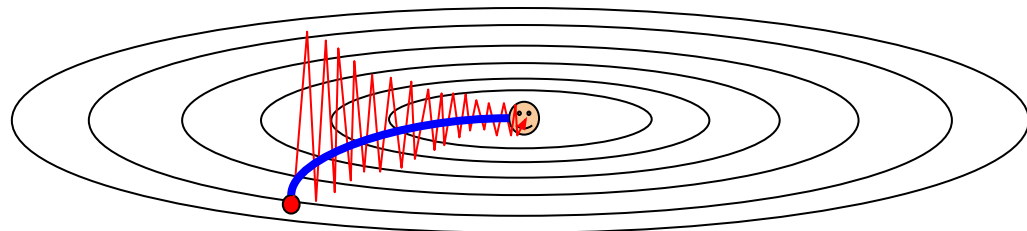
SGD + Momentum

Local Minima

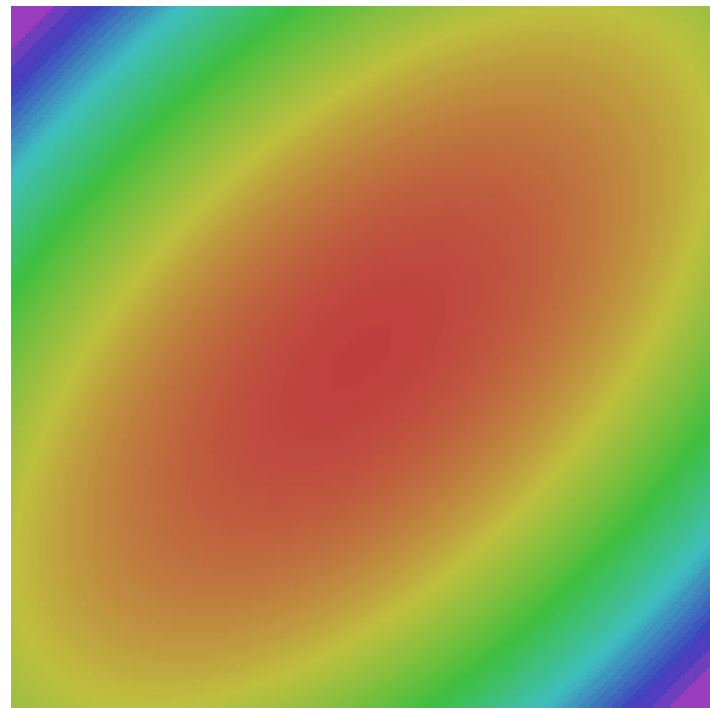
Saddle points



Poor Conditioning



Gradient Noise



SGD

SGD+Momentum

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

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

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

- Build up “velocity” 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:

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” 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

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

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

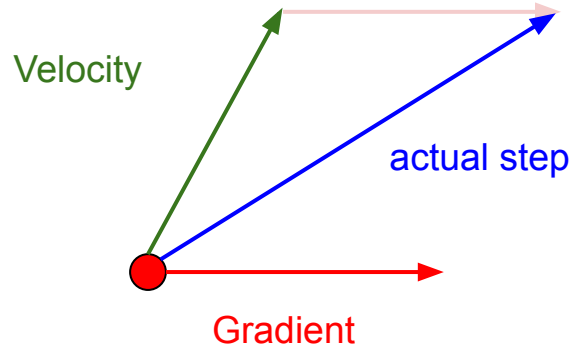
```
vx = 0
while True:
    dx = compute_gradient(x)
    vx = rho * vx + dx
    x -= learning_rate * vx
```

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

AdaGrad

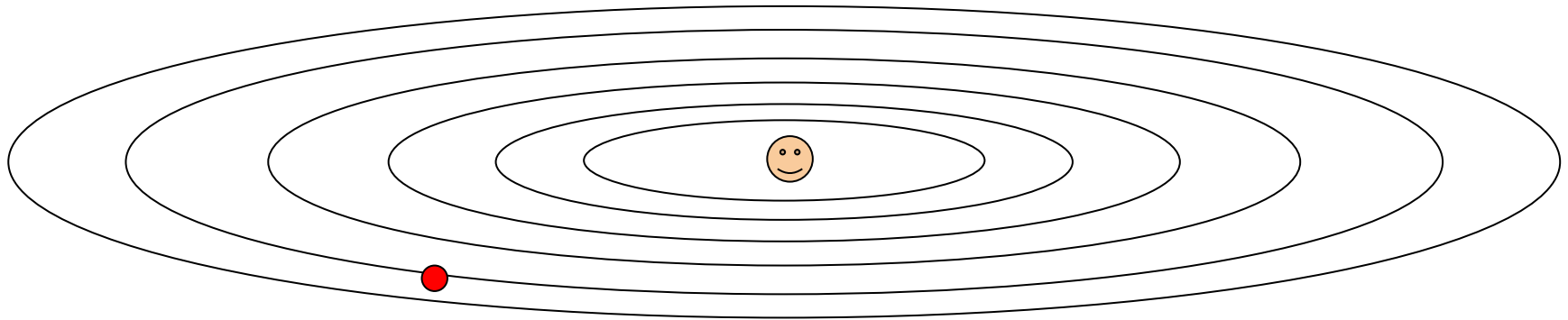
```
grad_squared = 0
while True:
    dx = compute_gradient(x)
    grad_squared += dx * dx
    x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

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”

AdaGrad

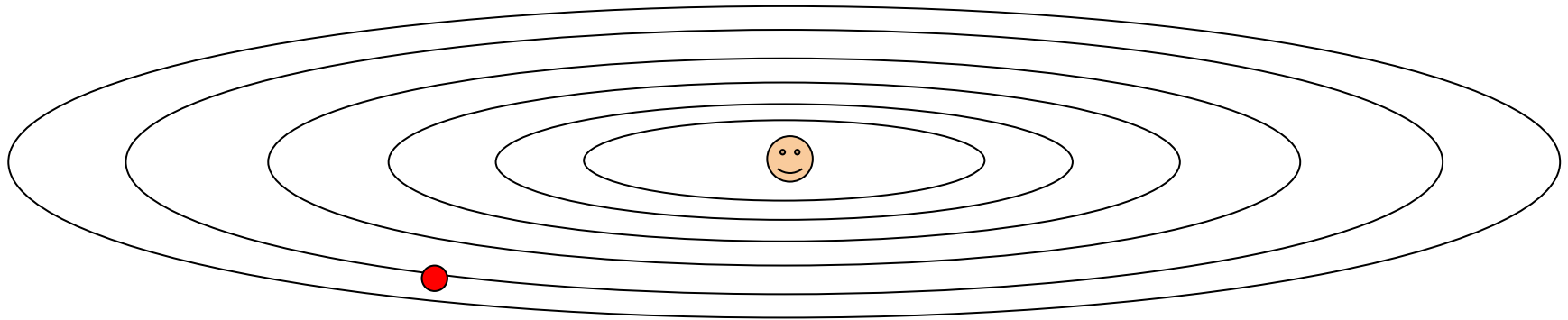
```
grad_squared = 0
while True:
    dx = compute_gradient(x)
    grad_squared += dx * dx
    x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```



Q: What happens with AdaGrad?

AdaGrad

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

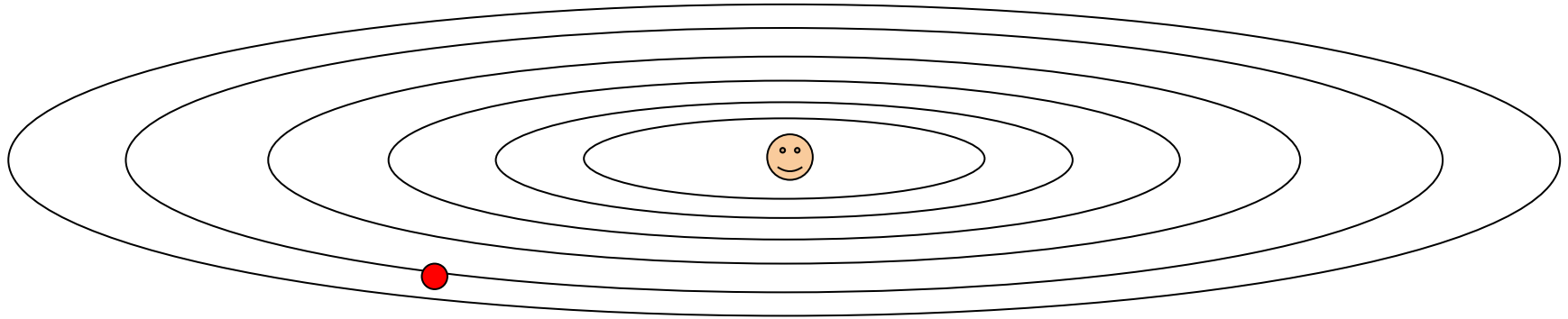


Q: What happens with AdaGrad?

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

AdaGrad

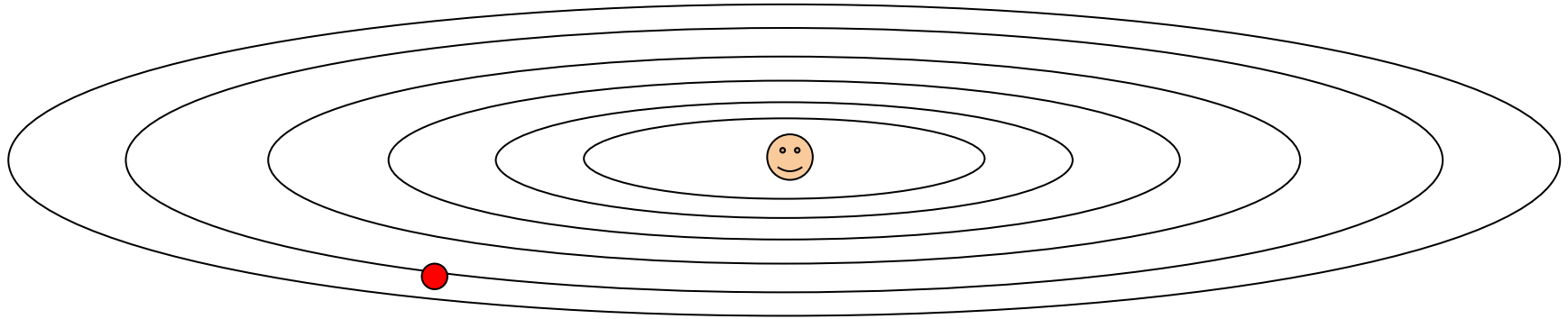
```
grad_squared = 0
while True:
    dx = compute_gradient(x)
    grad_squared += dx * dx
    x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```



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

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

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)
```

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))
```

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

Sort of like RMSProp with momentum

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

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

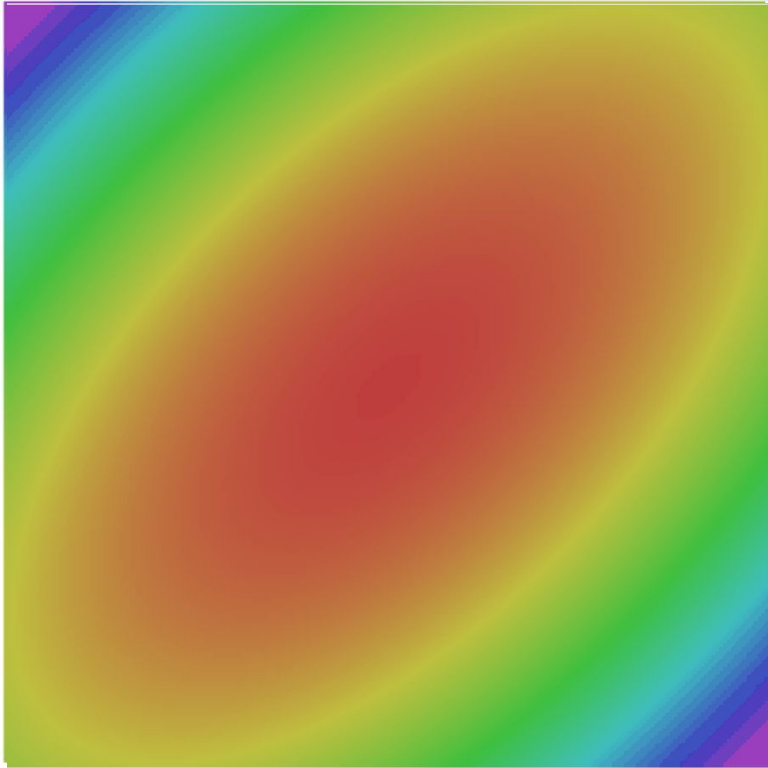
Bias correction

AdaGrad / RMSProp

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

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

Adam



- SGD
- SGD+Momentum
- RMSProp
- Adam

L2 Regularization vs Weight Decay

Optimization Algorithm

$$L(w) = L_{data}(w) + L_{reg}(w)$$

$$g_t = \nabla L(w_t)$$

$$s_t = \text{optimizer}(g_t)$$

$$w_{t+1} = w_t - \alpha s_t$$

L2 Regularization

$$L(w) = L_{data}(w) + \lambda |w|^2$$

$$g_t = \nabla L(w_t) = \nabla L_{data}(w_t) + 2\lambda w_t$$

$$s_t = \text{optimizer}(g_t)$$

$$w_{t+1} = w_t - \alpha s_t$$

L2 Regularization vs Weight Decay

Optimization Algorithm

$$L(w) = L_{data}(w) + L_{reg}(w)$$

$$g_t = \nabla L(w_t)$$

$$s_t = \text{optimizer}(g_t)$$

$$w_{t+1} = w_t - \alpha s_t$$

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

L2 Regularization

$$L(w) = L_{data}(w) + \lambda |w|^2$$

$$g_t = \nabla L(w_t) = \nabla L_{data}(w_t) + 2\lambda w_t$$

$$s_t = \text{optimizer}(g_t)$$

$$w_{t+1} = w_t - \alpha s_t$$

Weight Decay

$$L(w) = L_{data}(w)$$

$$g_t = \nabla L_{data}(w_t)$$

$$s_t = \text{optimizer}(g_t) + 2\lambda w_t$$

$$w_{t+1} = w_t - \alpha s_t$$

L2 Regularization vs Weight Decay

Optimization Algorithm

$$L(w) = L_{data}(w) + L_{reg}(w)$$

$$g_t = \nabla L(w_t)$$

$$s_t = \text{optimizer}(g_t)$$

$$w_{t+1} = w_t - \alpha s_t$$

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

$$L(w) = L_{data}(w) + \lambda |w|^2$$

$$g_t = \nabla L(w_t) = \nabla L_{data}(w_t) + 2\lambda w_t$$

$$s_t = \text{optimizer}(g_t)$$

$$w_{t+1} = w_t - \alpha s_t$$

Weight Decay

$$L(w) = L_{data}(w)$$

$$g_t = \nabla L_{data}(w_t)$$

$$s_t = \text{optimizer}(g_t) + 2\lambda w_t$$

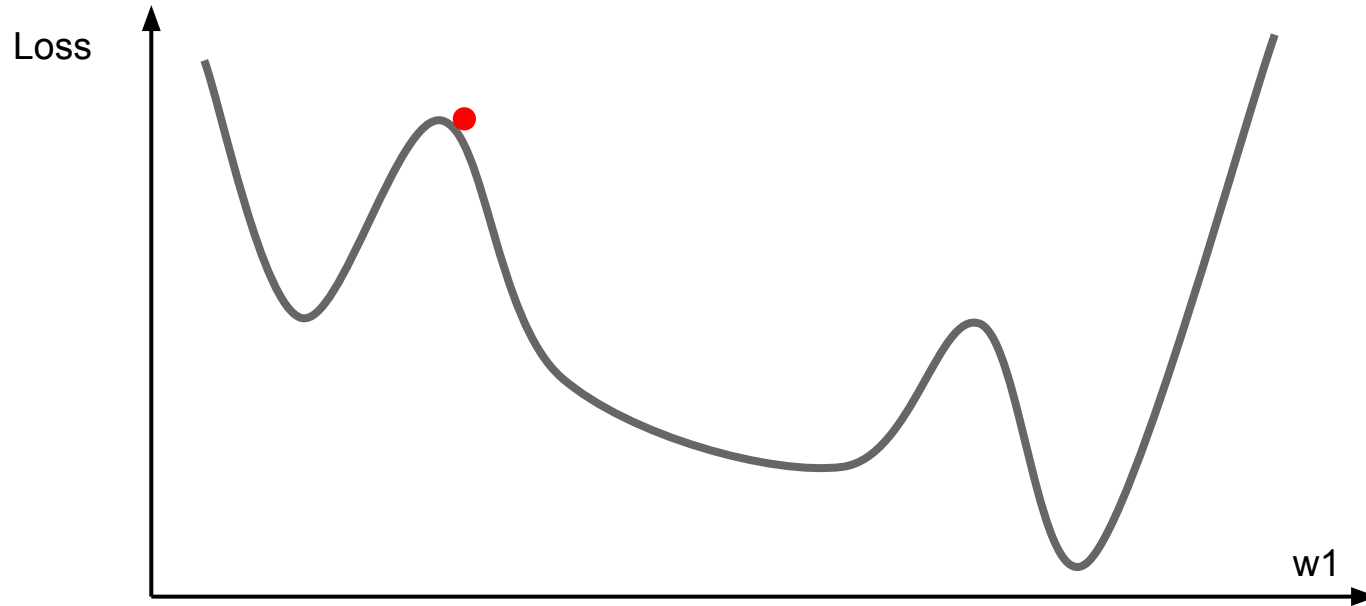
$$w_{t+1} = w_t - \alpha s_t$$

AdamW: Decoupled Weight Decay

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

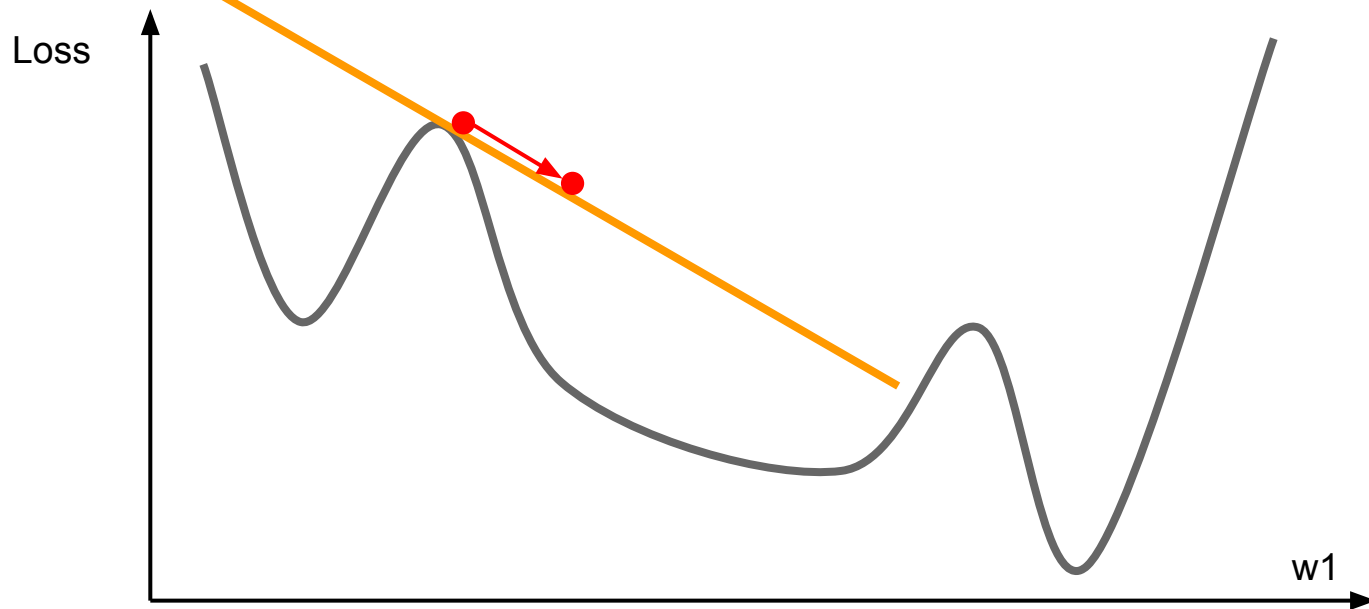
- 1: **given** $\alpha = 0.001, \beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 10^{-8}, \lambda \in \mathbb{R}$
- 2: **initialize** time step $t \leftarrow 0$, parameter vector $\boldsymbol{\theta}_{t=0} \in \mathbb{R}^n$, first moment vector $\mathbf{m}_{t=0} \leftarrow \mathbf{0}$, second moment vector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

First-Order Optimization



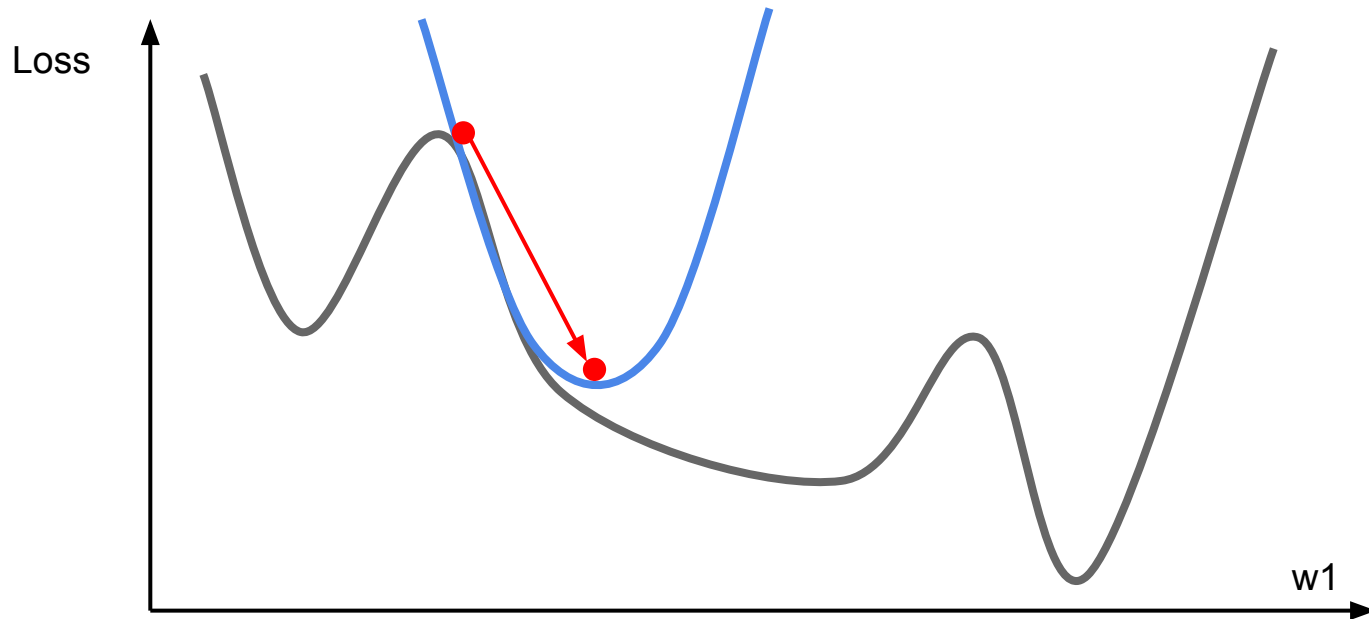
First-Order Optimization

- (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:

$$L(\theta) \approx L(\theta_0) + (\theta - \theta_0)^T \nabla_{\theta} L(\theta_0) + \frac{1}{2} (\theta - \theta_0)^T H (\theta - \theta_0)$$

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

$$\theta^* = \theta_0 - H^{-1} \nabla_{\theta} L(\theta_0)$$

Q: Why is this bad for deep learning?

In practice:

- **AdamW** should probably be your “default” optimizer for new problems
- **Adam** is a good second 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!

Summary

TLDRs

We looked in detail at:

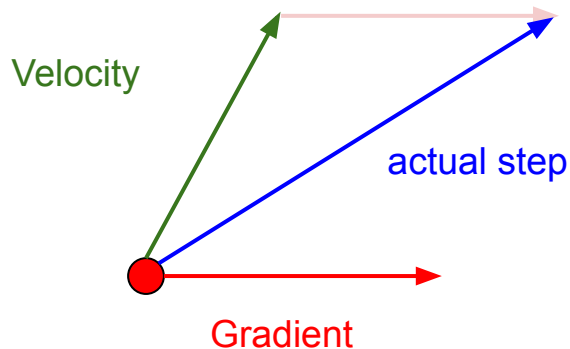
- Activation Functions (use ReLU or GeLU)
- Layer Normalization (used in transformers!)
- Batch Normalization (use this if not using layer norm!)
- Optimizers (use AdamW)

Next time:

Tokenization & Introduction to language

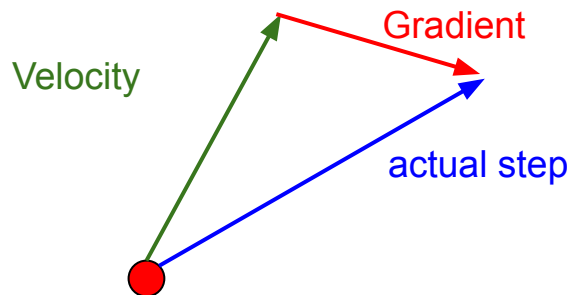
Nesterov Momentum

Momentum update:



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

Nesterov Momentum



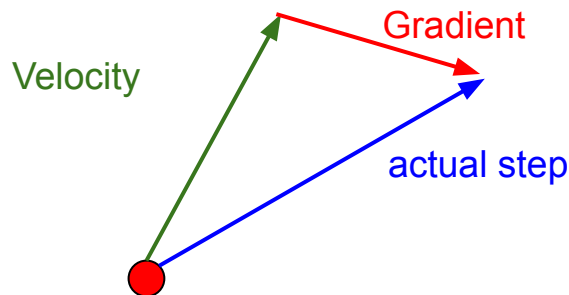
“Look ahead” to the point where updating using velocity would take us; compute gradient there and mix it with velocity to get actual update direction

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

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

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



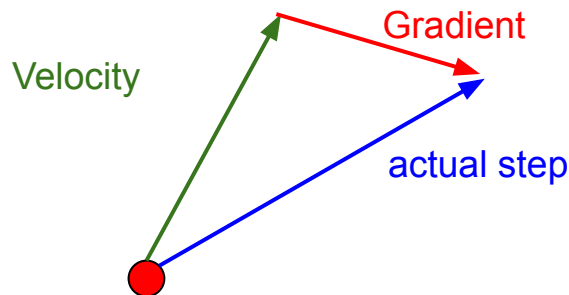
“Look ahead” to the point where updating using velocity would take us; compute gradient there and mix it with velocity to get actual update direction

Nesterov 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 $x_t, \nabla f(x_t)$



“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

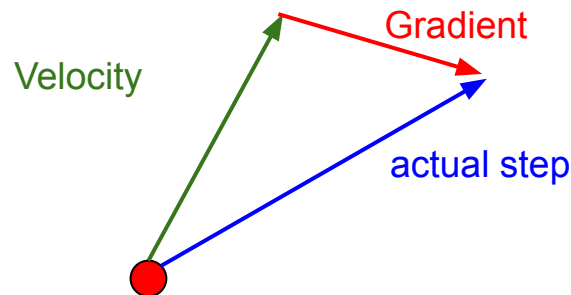
Nesterov Momentum

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

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

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

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



“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 Momentum

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

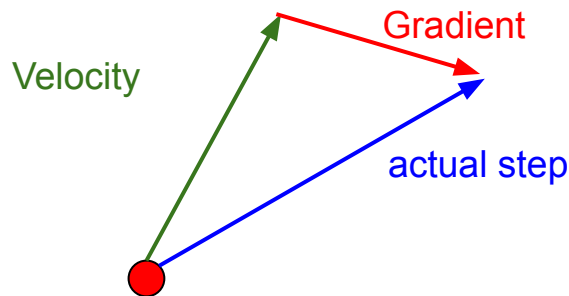
$$x_{t+1} = x_t + v_{t+1}$$

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

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

$$\begin{aligned}\tilde{x}_{t+1} &= \tilde{x}_t - \rho v_t + (1 + \rho)v_{t+1} \\ &= \tilde{x}_t + v_{t+1} + \rho(v_{t+1} - v_t)\end{aligned}$$

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



“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

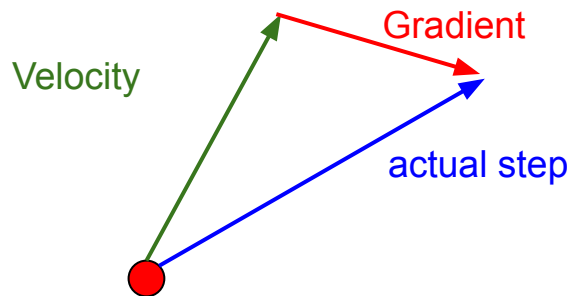
Nesterov Momentum

That's it!

Step 1: Calculate the velocity at $t+1$

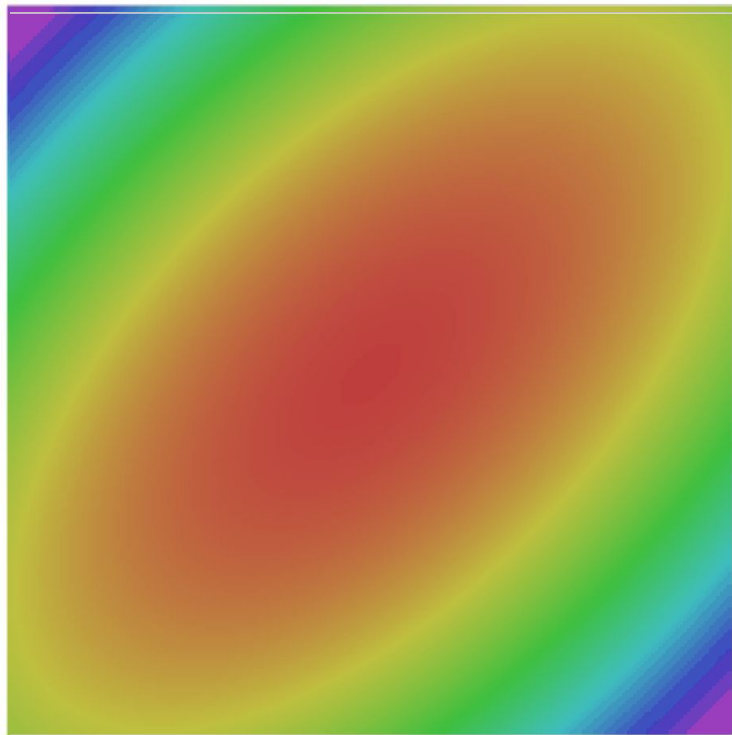
Step 2: Update the parameters using the velocities at $t+1$ and t

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



“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 Momentum



- SGD
- SGD+Momentum
- Nesterov

Optional: Second-Order Optimization

second-order Taylor expansion:

$$L(\theta) \approx L(\theta_0) + (\theta - \theta_0)^T \nabla_{\theta} L(\theta_0) + \frac{1}{2} (\theta - \theta_0)^T H (\theta - \theta_0)$$

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

$$\theta^* = \theta_0 - H^{-1} \nabla_{\theta} L(\theta_0)$$

Hessian has $O(N^2)$ elements
Inverting takes $O(N^3)$
 $N = (\text{Tens or Hundreds of}) \text{ Billions}$

Q: Why is this bad for deep learning?

Optional: Second-Order Optimization

$$\theta^* = \theta_0 - H^{-1} \nabla_{\theta} L(\theta_0)$$

- 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.

Optional: 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