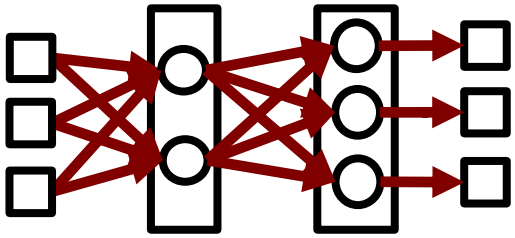
Photogrammetry & Robotics Lab

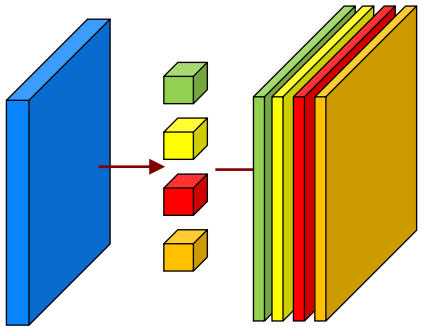
Machine Learning for Robotics and Computer Vision

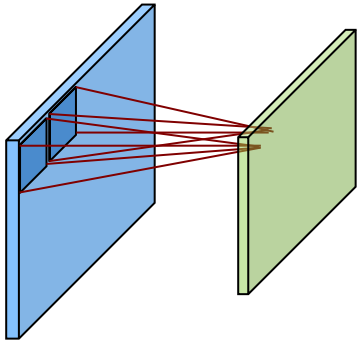
Learning CNNs

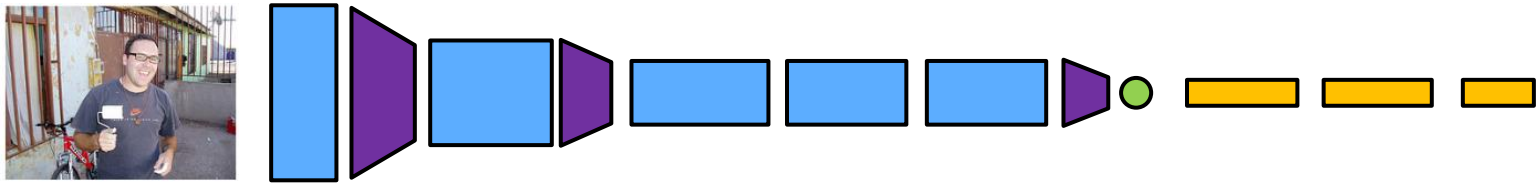
Jens Behley

Last Lecture



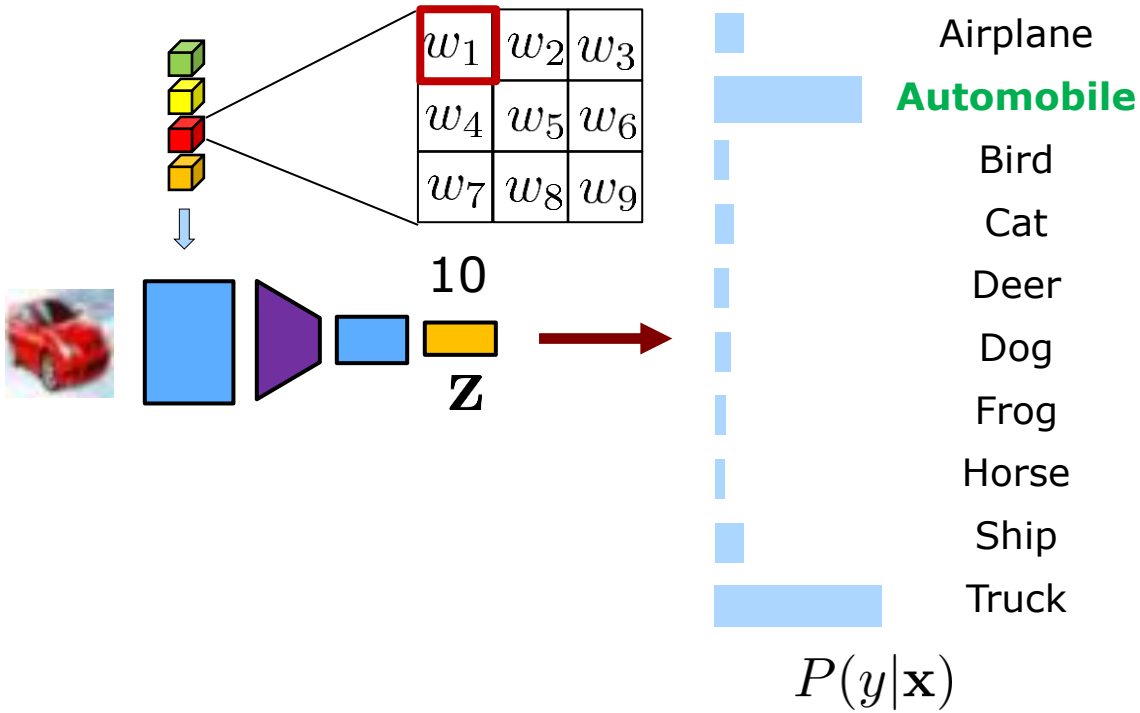






- Building Blocks of Convolutional Neural Networks (CNN)
- Structure of CNN that excelled at ImageNet 2012 competition

How to learn parameters?



- Given training data, we want to find parameters that provide good predictions
- Loss measures difference between predicted outcome and desired outcome

Loss Minimization

- Loss $\ell(y_i, f(\mathbf{x}_i; \theta)) \in \mathbb{R}$ determines difference between prediction $f(\mathbf{x}_i; \theta)$ and target value y_i

$$L(\theta) = \frac{1}{N} \sum_i \ell(y_i, f(\mathbf{x}_i; \theta))$$

- Typical loss functions
  - Regression: **L2 loss**
  - Classification: **Cross-entropy loss**

L2 Loss

- L2-loss is defined as

$$\ell(y_i, f(\mathbf{x}_i)) = (y_i - f(\mathbf{x}_i, \theta))^2$$

- Intuitively, predicted values $f(\mathbf{x})$ should be close to target values
- Equivalent to negative log-likelihood with normal distributed error (see **regression lecture**)

MSELOSS

L2 Loss in PyTorch

CLASS `torch.nn.MSELoss(size_average=None, reduce=None, reduction='mean')` [SOURCE]

Creates a criterion that measures the mean squared error (squared L2 norm) between each element in the input x and target y .

The unreduced (i.e. with `reduction` set to `'none'`) loss can be described as:

$$\ell(x, y) = L = \{l_1, \dots, l_N\}^\top, \quad l_n = (x_n - y_n)^2,$$

where N is the batch size. If `reduction` is not `'none'` (default `'mean'`), then:

Cross Entropy Loss

- As before, for classification:

$$P(y = j|\mathbf{x}) = \mathrm{softmax}_j(f_1(\mathbf{x}), \ldots, f_C(\mathbf{x}))$$
$$= \frac{\exp(f_j(\mathbf{x}))}{\sum_k \exp(f_k(\mathbf{x})))}$$

- The negative log-likelihood is then:

$$\ell(j, f(\mathbf{x})) = -\log \frac{\exp(f_j(\mathbf{x}))}{\sum_k \exp(f_k(\mathbf{x}))}$$
$$= -f_j(\mathbf{x}) + \log(\sum_k \exp(f_k(\mathbf{x}))$$

- But why is this called *cross entropy* then?

Relation to Cross Entropy

- Cross entropy defined as

$$H(P,Q) = -\sum_k P(x_k) \log Q(x_k)$$

- For a target label j, we define:

$$P(x) = (0, \ldots, 0, 1, 0, \ldots, 0)$$

j-th entry

- With $Q(x) = \mathrm{softmax}(f_1(\mathbf{x}), \ldots, f_C(\mathbf{x}))$, we get

$$\begin{aligned} H(P,Q) &= -1 \log Q(x_j) \\ &= -f_j(\mathbf{x}) + \log \sum_k \exp(f_k(\mathbf{x})) \end{aligned}$$

- $P(x)$ can have other distribution!

Cross Entropy in PyTorch

CROSSENTROPYLOSS

CLASS `torch.nn.CrossEntropyLoss(weight=None, size_average=None, ignore_index=-100, reduce=None, reduction='mean')` [SOURCE]

This criterion combines `LogSoftmax` and `NLLLoss` in one single class.

It is useful when training a classification problem with C classes. If provided, the optional argument `weight` should be a 1D *Tensor* assigning weight to each of the classes. This is particularly useful when you have an unbalanced training set.

The *input* is expected to contain raw, unnormalized scores for each class.

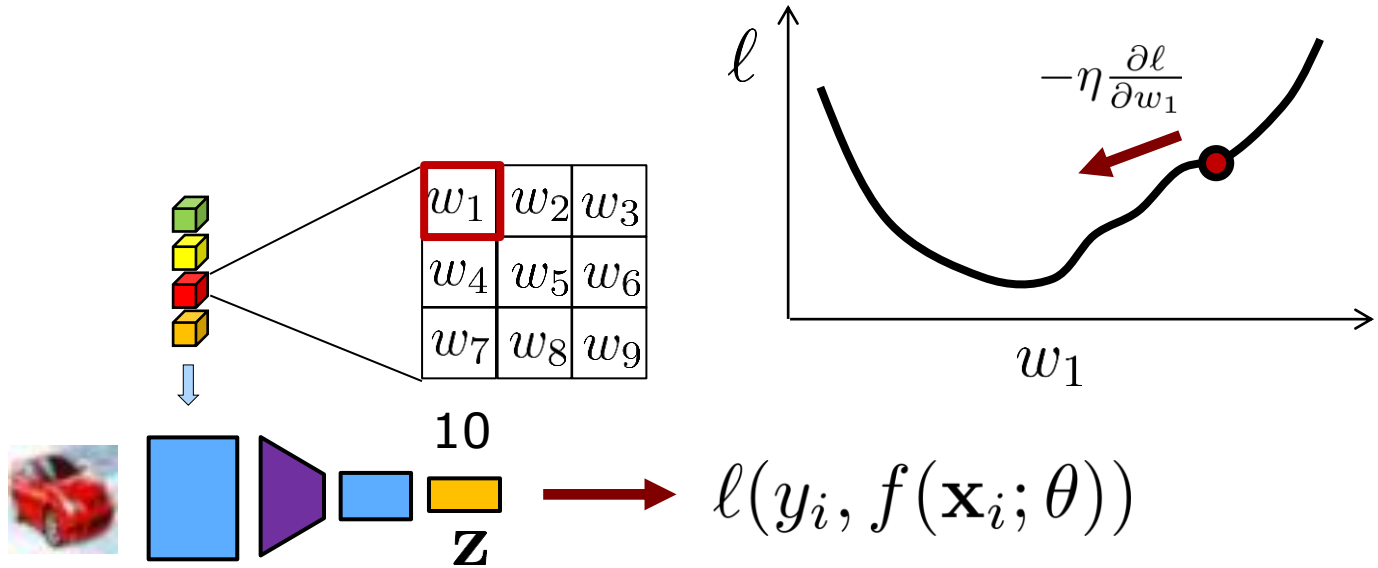
input has to be a Tensor of size either $(minibatch, C)$ or $(minibatch, C, d_1, d_2, ..., d_K)$ with $K \geq 1$ for the *K*-dimensional case (described later).

This criterion expects a class index in the range $[0, C-1]$ as the *target* for each value of a 1D tensor of size *minibatch*; if *ignore_index* is specified, this criterion also accepts this class index (this index may not necessarily be in the class range).

The loss can be described as:

$$\text{loss}(x, class) = -\log\left(\frac{\exp(x[class])}{\sum_j \exp(x[j])}\right) = -x[class] + \log\left(\sum_j \exp(x[j])\right)$$

Gradient-based Optimization



- As before, partial derivatives tell us in which direction to change parameters such that loss is minimized

Chain Rule

- Given that we have a composition of functions, we can apply chain rule:

$$L(\theta_1, \theta_2, \theta_3) = f_3(f_2(f_1(\mathbf{x}; \theta_1); \theta_2); \theta_3)$$

- To get the partial derivatives, we apply the chain rule:

$$\frac{\partial L}{\partial \theta_3} = \boxed{\frac{\partial L}{\partial f_3} \frac{\partial f_3}{\partial \theta_3}}$$

$$\frac{\partial L}{\partial \theta_2} = \frac{\partial L}{\partial f_3} \boxed{\frac{\partial f_3}{\partial f_2} \frac{\partial f_2}{\partial \theta_2}}$$

$$\frac{\partial L}{\partial \theta_1} = \frac{\partial L}{\partial f_3} \frac{\partial f_3}{\partial f_2} \boxed{\frac{\partial f_2}{\partial f_1} \frac{\partial f_1}{\partial \theta_1}}$$

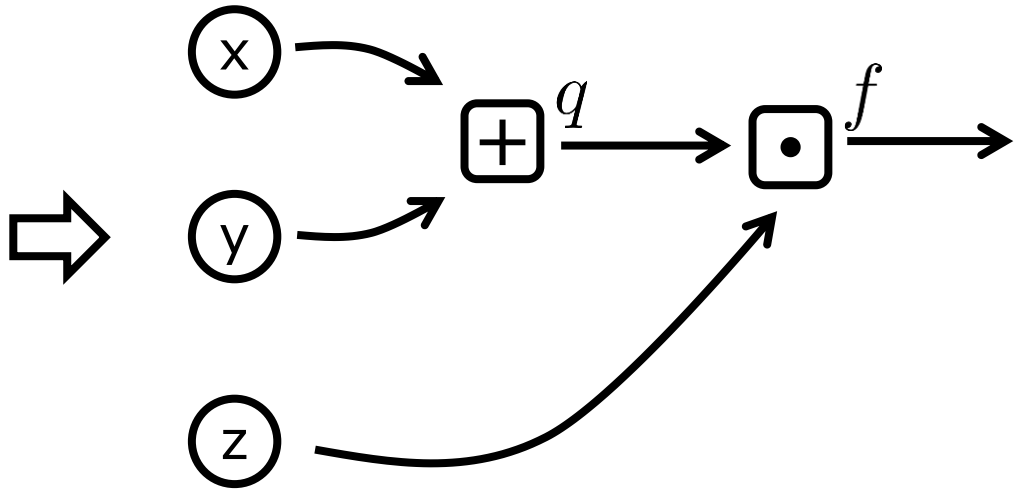
How to determine order of computation?

Compute Graph

$$f(x, y, z) = (x + y) \cdot z$$

$$q = x + y$$

$$f = q \cdot z$$

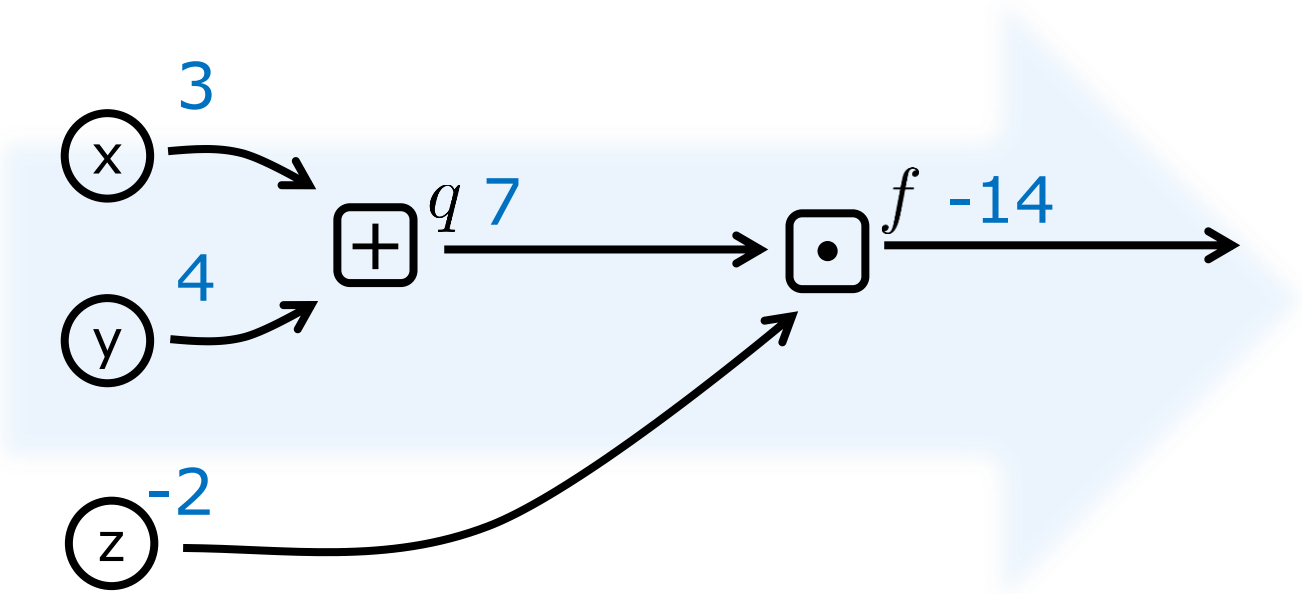


- Directed graph that encodes order of operations
- Edges determine information flow and dependencies of operations

Slides inspired by Justin Johnson's lecture on backprop

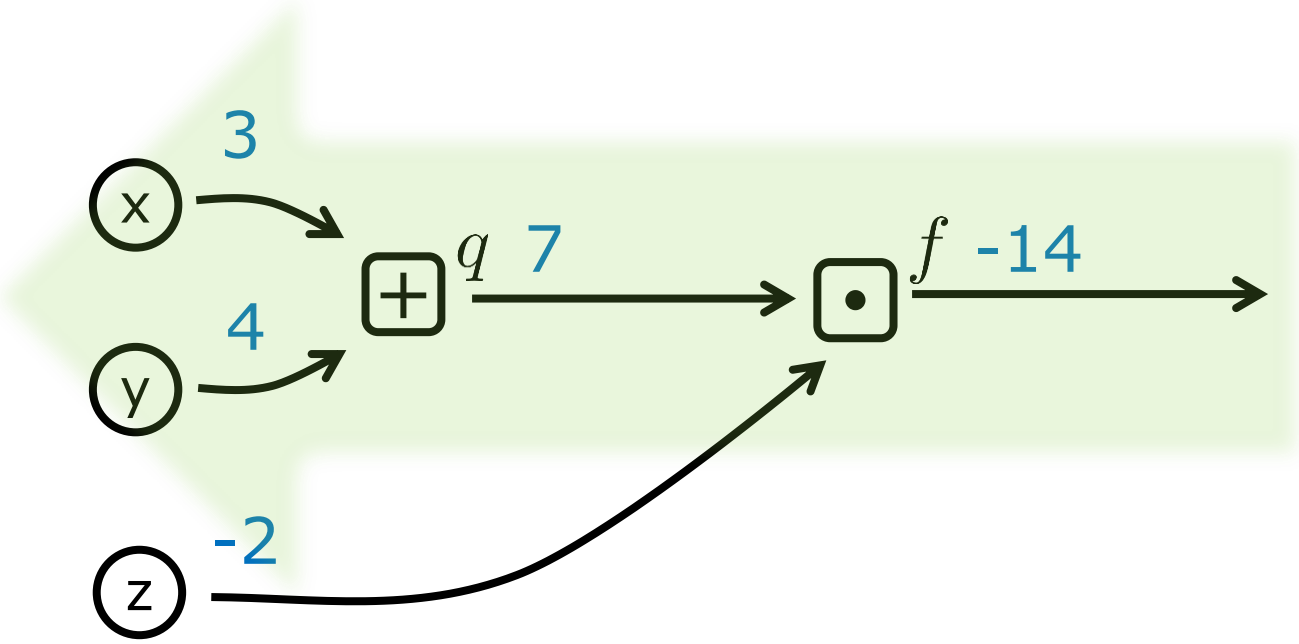
Backpropagation: Forward Pass

- For a given input, we first compute all activations in the forward pass



Backpropagation: Backward Pass

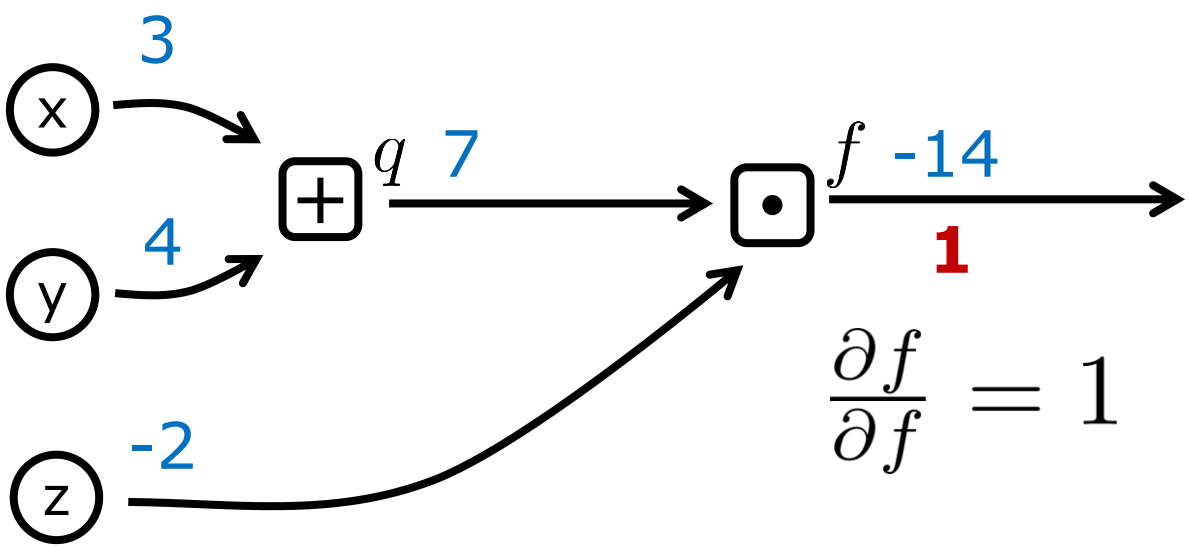
- In the backward pass, we use the chain rule and can reuse already calculated derivatives



Slides inspired by Justin Johnson's lecture on backprop

Backpropagation: Backward Pass

$$q = x + y$$
$$f = q \cdot z$$

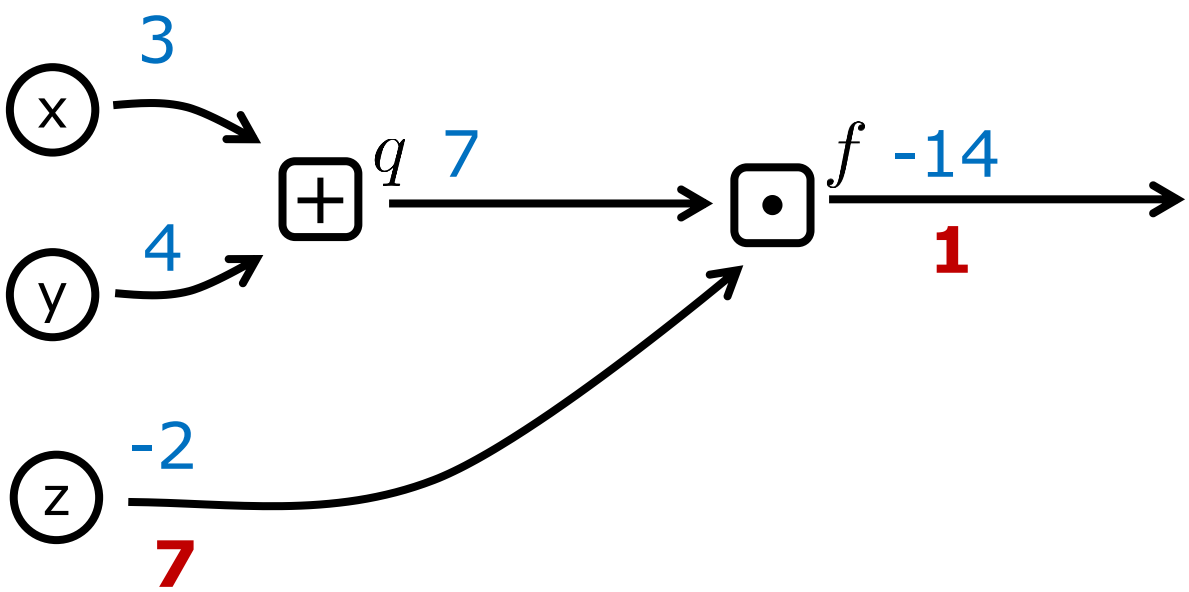


$$\frac{\partial f}{\partial f} = 1$$

Slides inspired by Justin Johnson's lecture on backprop

Backpropagation: Backward Pass

$$q = x + y$$
$$f = q \cdot z$$



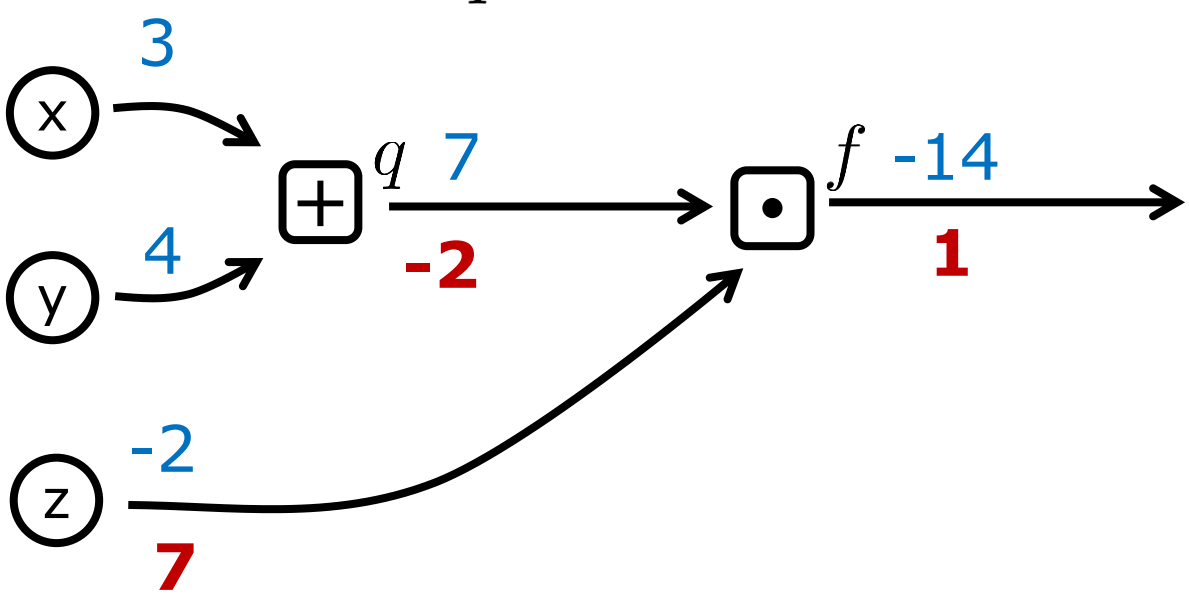
$$\frac{\partial f}{\partial z} = q$$

Slides inspired by Justin Johnson's lecture on backprop

Backpropagation: Backward Pass

$$q = x + y$$
$$f = q \cdot z$$

$$\frac{\partial f}{\partial q} = z$$

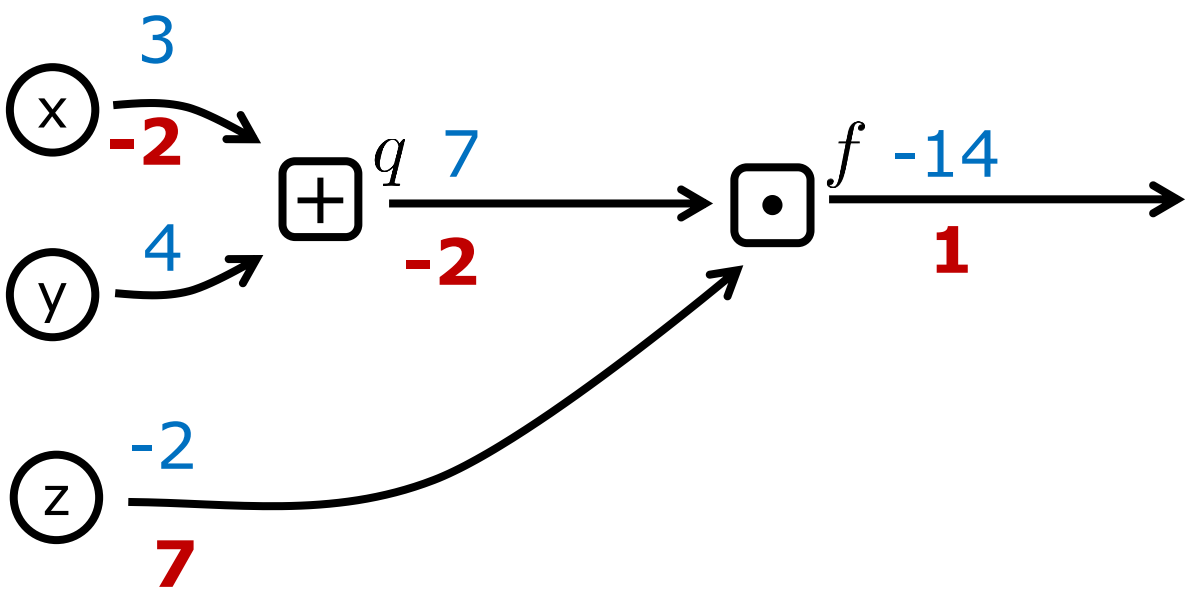


Slides inspired by Justin Johnson's lecture on backprop

Backpropagation: Backward Pass

$$q = x + y$$
$$f = q \cdot z$$

$$\frac{\partial f}{\partial x} = \frac{\partial q}{\partial x} \cdot \frac{\partial f}{\partial q} = 1 \cdot -2$$

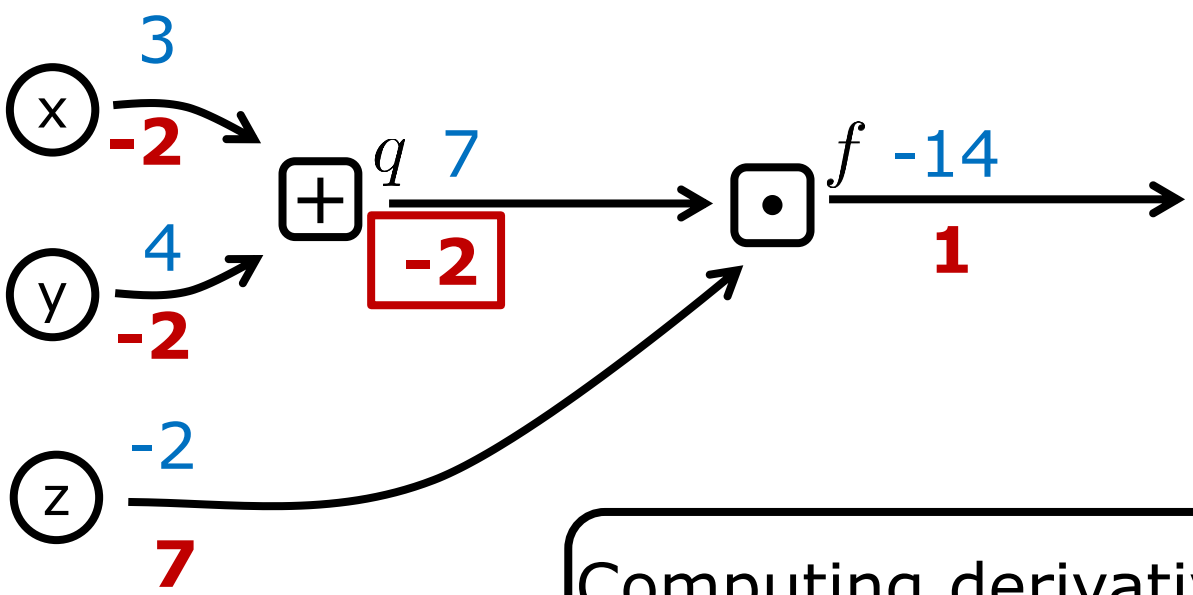


Slides inspired by Justin Johnson's lecture on backprop

Backpropagation: Backward Pass

$$\boxed{q = x + y}$$
$$f = q \cdot z$$

$$\frac{\partial f}{\partial y} = \frac{\partial q}{\partial y} \cdot \frac{\partial f}{\partial q} = 1 \cdot -2$$

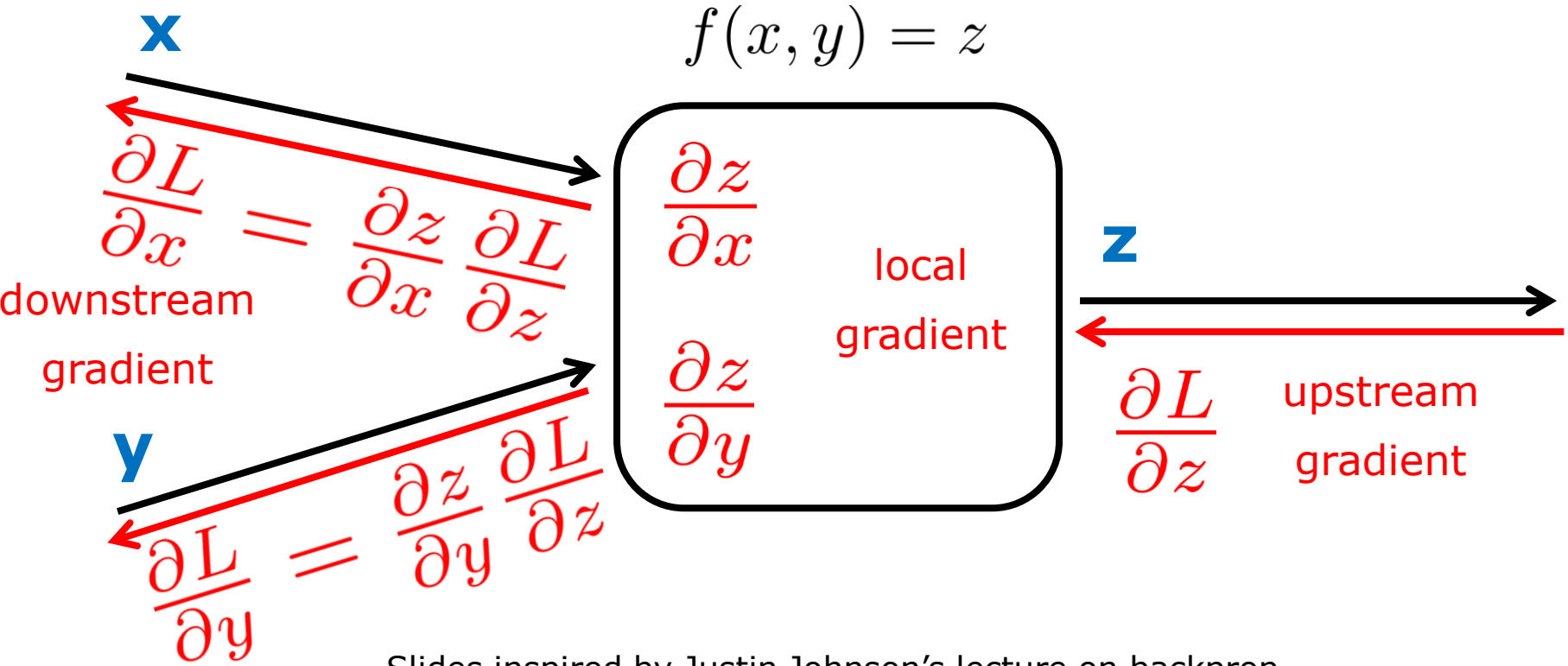


Computing derivatives only involves **local** information

Slides inspired by Justin Johnson's lecture on backprop

Local connectivity

- For each compute node only the incident and outgoing edges are relevant
- Gradient "messages" are passed along the edges



Slides inspired by Justin Johnson's lecture on backprop

Backpropagation with vectors

$$f : \mathbb{R} \mapsto \mathbb{R} \qquad f : \mathbb{R}^N \mapsto \mathbb{R} \qquad f : \mathbb{R}^N \mapsto \mathbb{R}^M$$

$$\frac{df}{dx} \in \mathbb{R} \qquad \frac{df}{d\mathbf{x}} \in \mathbb{R}^N \qquad \frac{df}{d\mathbf{x}} \in \mathbb{R}^{M \times N}$$

$$\begin{pmatrix} \frac{\partial f}{\partial x_1} \\ \vdots \\ \frac{\partial f}{\partial x_N} \end{pmatrix} \qquad \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_N} \\ \vdots & \ddots & \cdots \\ \frac{\partial f_M}{\partial x_1} & \cdots & \frac{\partial f_M}{\partial x_N} \end{pmatrix}$$

- With $f : \mathbb{R}^N \mapsto \mathbb{R}$, we have gradient vector as derivative
- With $f : \mathbb{R}^N \mapsto \mathbb{R}^M$, we have Jacobian matrix as derivative

Slides inspired by Justin Johnson's lecture on backprop

Backpropagation with vectors

$\mathbf{x} \in \mathbb{R}^{D_x}$

$f : \mathbb{R}^{D_x} \times \mathbb{R}^{D_y} \mapsto \mathbb{R}^{D_z}$

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

downstream gradient

$\mathbf{y} \in \mathbb{R}^{D_y}$

$$\frac{\partial L}{\partial y} = \frac{\partial z}{\partial y}\frac{\partial L}{\partial z}$$

$$\frac{\partial z}{\partial x} \in \mathbb{R}^{D_z \times D_x}$$

local Jacobians

$$\frac{\partial z}{\partial y} \in \mathbb{R}^{D_z \times D_y}$$

$\mathbf{z} \in \mathbb{R}^{D_z}$

$$\frac{\partial L}{\partial z} \in \mathbb{R}^{D_z}$$

upstream gradient

- **Important:** For matrix-vector multiplication in the downstream gradient, we use transpose of Jacobian

Slides inspired by Justin Johnson's lecture on backprop

Example: ReLU

$$f(\mathbf{x}) = \max(0, \mathbf{x})$$

$\mathbf{x}$ $\begin{pmatrix} 1 \\ -2 \\ 3 \end{pmatrix}$ $\longrightarrow$

$\longleftarrow$ $\frac{\partial L}{\partial \mathbf{x}}$ $\begin{pmatrix} -1 \\ 0 \\ 2 \end{pmatrix}$

$$\frac{\partial y}{\partial x} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

$\mathbf{y}$ $\begin{pmatrix} 1 \\ 0 \\ 3 \end{pmatrix}$ $\longrightarrow$

$\longleftarrow$ $\frac{\partial L}{\partial \mathbf{y}}$ $\begin{pmatrix} -1 \\ 4 \\ 2 \end{pmatrix}$

$$\frac{\partial L}{\partial x_i} = \begin{cases} \frac{\partial L}{\partial y_i} & \text{, if } x_i > 0 \\ 0 & \text{, otherwise.} \end{cases}$$

- ReLU "blocks" partial derivatives
- Computing explicitly the Jacobian is not need, but can be evaluated on the fly!

Slides inspired by Justin Johnson's lecture on backprop

Implementation in PyTorch

```
class Multiply(torch.autograd.Function):
  def forward(ctx, x, y):
     ctx.save_for_backward(x,y)
     return x * y
  def backward(ctx, grad_z):
     x, y = ctx.saved_tensors
     grad_x = y * grad_z
     grad_y = x * grad_z
     return grad_x, grad_y
```

- Operations have `forward` and `backward` method
- Forward caches activations: Roughly model size additionally needed while training!
- With gradient checkpointing only “heavy” forward passes are stored

Optimization Methods

- With given partial derivatives, we can now update parameters with gradient descent
- But computing gradients for all data:

$$L(\theta) = \frac{1}{N} \sum_i \ell(y_i, f(\mathbf{x}_i; \theta))$$

with large N very expensive

- Better: compute many small inaccurate updates to make faster progress

Stochastic Gradient Descent

```
while not converged:
```

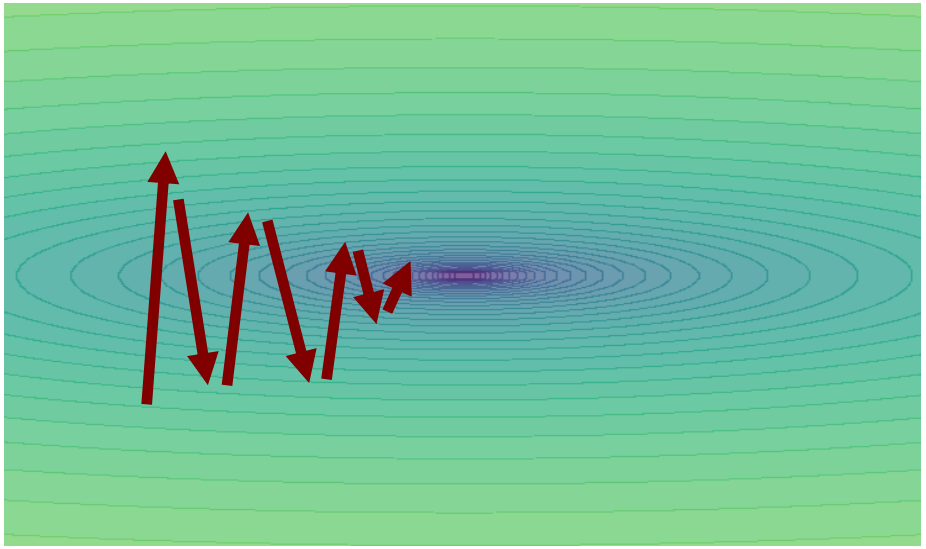
Compute $\frac{dL}{d\theta}$ for current batch

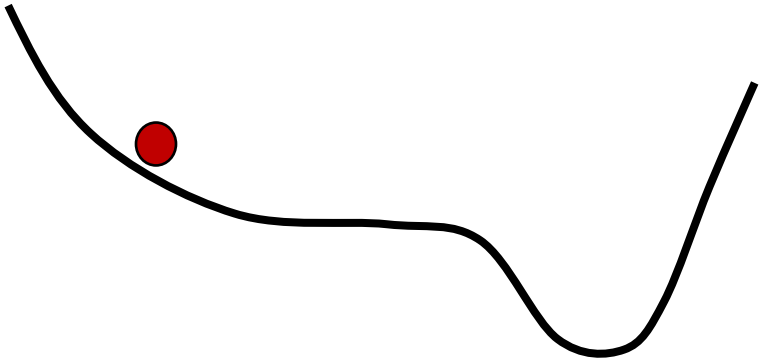
$$\theta_t = \theta_{t-1} - \eta \frac{dL}{d\theta}$$

- Compute updates with a small **batch** of examples (e.g. 32, 64, 128)
- Batch randomly sampled from all data
- η is the **learning rate** (hyper parameter)

[Robbins, 1951]

Problems with SGD





- In certain situations slow convergence rate
- Saddle points can stall learning progress

Momentum

$v_0 = 0, \gamma = 0.9$

while not converged:

Compute $\frac{dL}{d\theta}$ for current batch

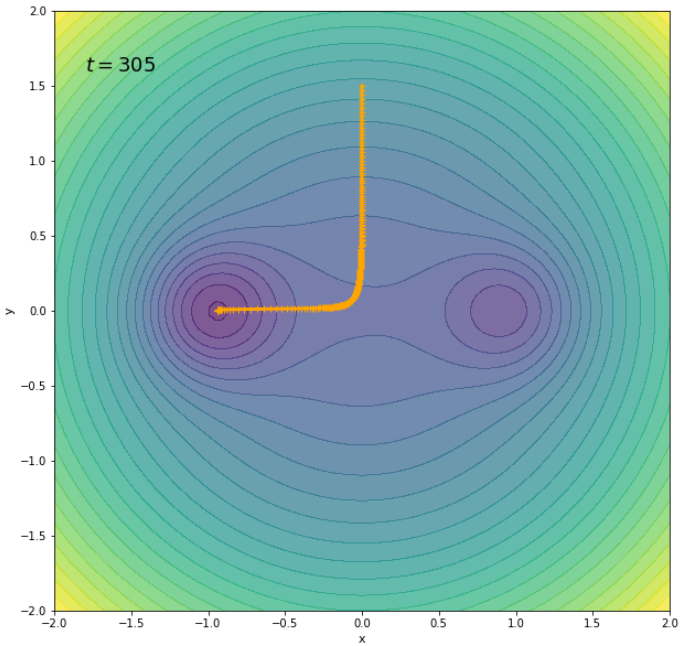
$v_t = \gamma v_{t-1} + \eta \frac{dL}{d\theta}$

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

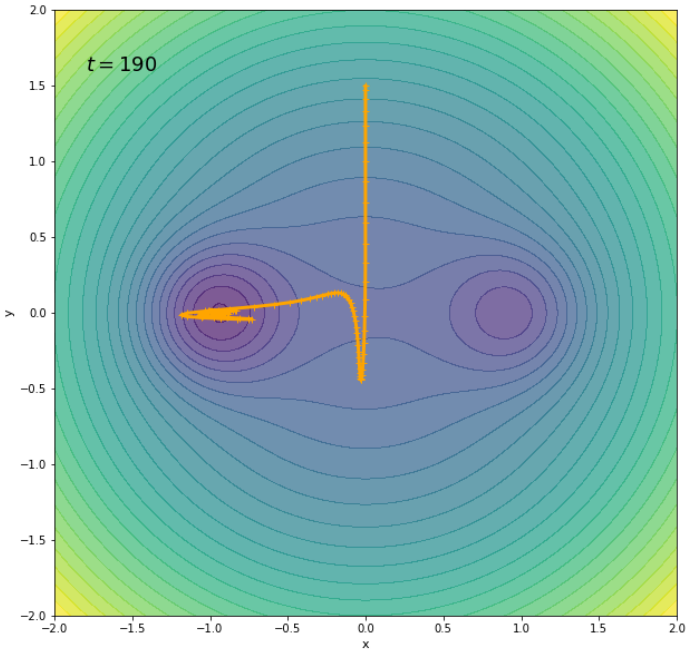
- Momentum allows to "roll" over saddle points and reduced oscillation
- Moving average over past updates and current update
- Overshooting possible

[Polyak, 1964]

SGD vs. SGD+Momentum



SGD



SGD+Momentum

- SGD+Momentum converges faster, but overshoots

Nestrov Accelerated Gradient

$v_0 = 0, \gamma = 0.9$

while not converged:

Compute $\frac{dL}{d(\theta - \gamma v_{t-1})}$

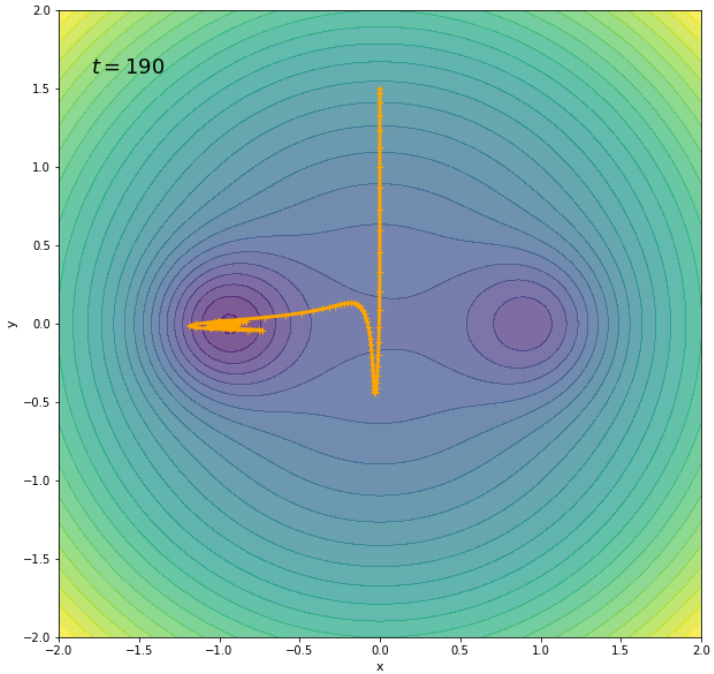
$$v_t = \gamma v_{t-1} + \eta \frac{dL}{d(\theta - \gamma v_{t-1})}$$

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

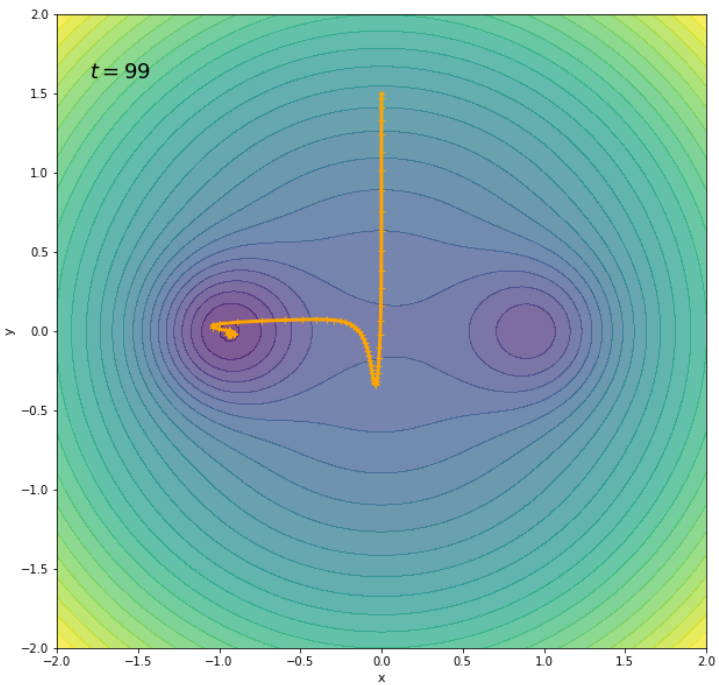
- Compute gradient $\frac{dL}{d(\theta - \gamma v_{t-1})}$ at look ahead position and account for future change

[Nestrov, 1983]

Momentum vs. NAG



SGD+Momentum



Nestrov

- Less overshooting and usually faster convergence than SGD+Momentum

Adagrad

$G_0 = 0$

while not converged:

Compute $\frac{dL}{d\theta}$ for current batch

$$G_t = G_{t-1} + \left(\frac{dL}{d\theta}\right)^2$$

$$\theta_t = \theta_{t-1} - \frac{\eta}{\sqrt{G_t + \epsilon}} \frac{dL}{d\theta}$$

- Adjust learning rate depending on squared gradients
- Larger/smaller updates for parameters with small/large gradient

[Duchi, 2011]

Adagrad

$G_0 = 0$

while not converged:

Compute $\frac{dL}{d\theta}$ for current batch

$$\boxed{G_t = G_{t-1} + \left(\frac{dL}{d\theta}\right)^2}$$

$$\theta_t = \theta_{t-1} - \frac{\eta}{\sqrt{G_t + \epsilon}} \frac{dL}{d\theta}$$

- Adagrad only increases G_t and therefore the influence of an gradient update decreases!
- Quickly very slow progress due to small learning rate

RMSprop

$G_0 = 0$

while not converged:

Compute $\frac{dL}{d\theta}$ for current batch

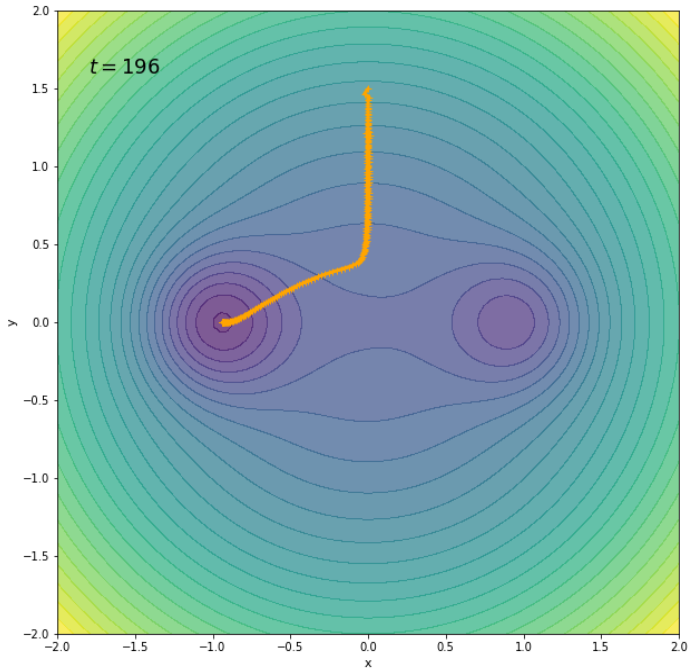
$$G_t = 0.9 G_{t-1} + 0.1 \left(\frac{dL}{d\theta}\right)^2$$

$$\theta_t = \theta_{t-1} - \frac{\eta}{\sqrt{G_t + \epsilon}} \frac{dL}{d\theta}$$

- Moving average over squared gradients ensures that progress does not stop

[Tieleman, 2012]

RMSProp



- No overshooting and fast convergence.

ADAM

$$m_0 = 0, v_0 = 0, \beta_1 = 0.9, \beta_2 = 0.999$$

while not converged:

Compute $\frac{dL}{d\theta}$ for current batch

Momentum $$m_t = \beta_1 m_{t-1} + (1-\beta_1)\left(\frac{dL}{d\theta}\right)$$

RMSprop $$v_t = \beta_2 v_{t-1} + (1-\beta_2)\left(\frac{dL}{d\theta}\right)^2$$

$$\hat{m}_t = (1-\beta_1^t)^{-1} m_t$$

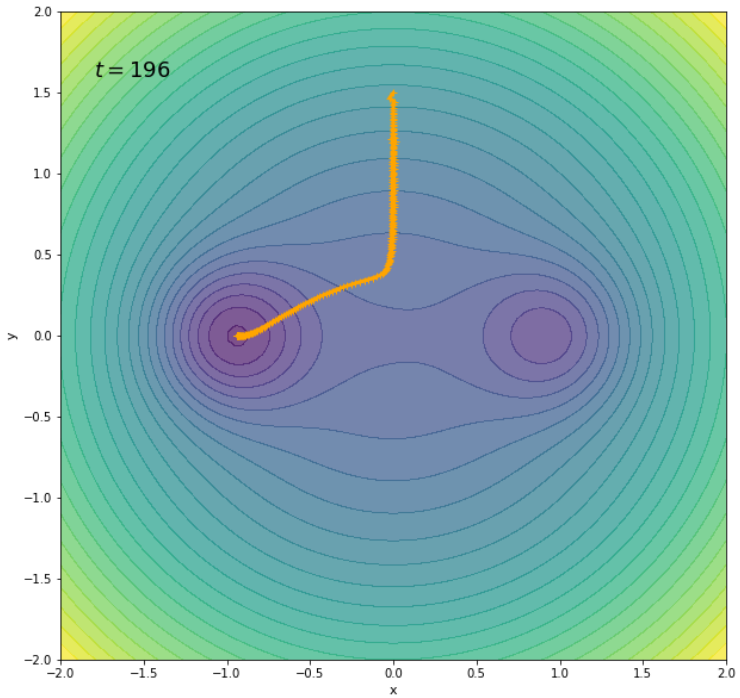
$$\hat{v}_t = (1-\beta_2^t)^{-1} v_t$$

$$\theta_t = \theta_{t-1} - \frac{\eta}{\sqrt{\hat{v}_t}+\epsilon}\hat{m}_t$$

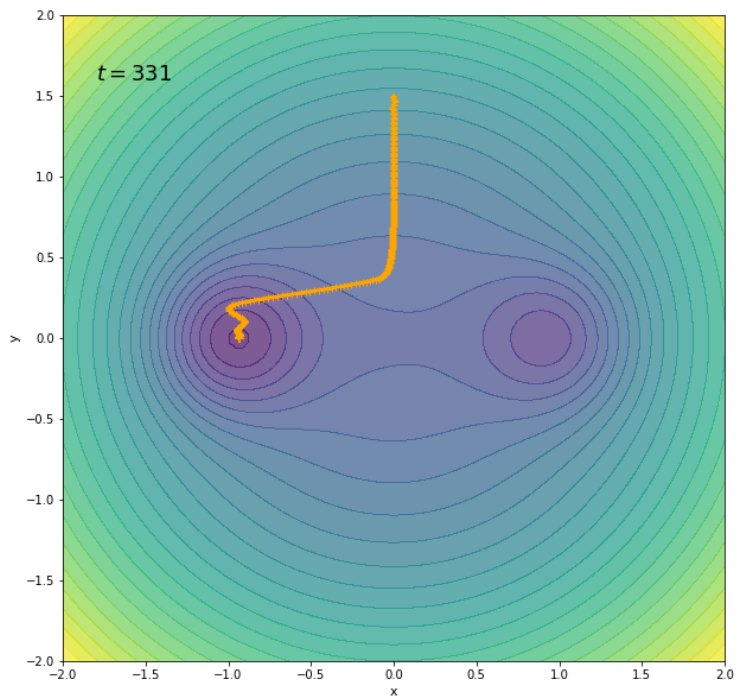
- ADAM combines advantages of momentum and parameter-wise adaptive learning rates

[Kingma, 2015]

RMSProp vs. ADAM

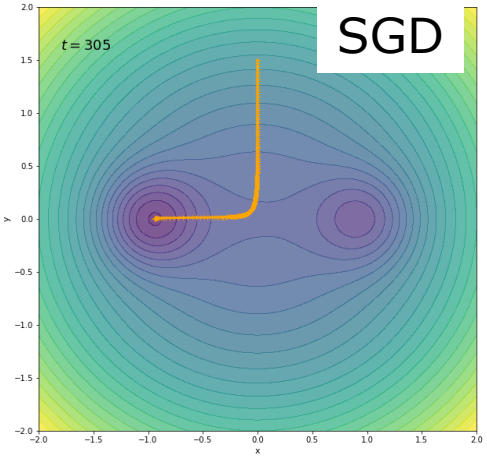


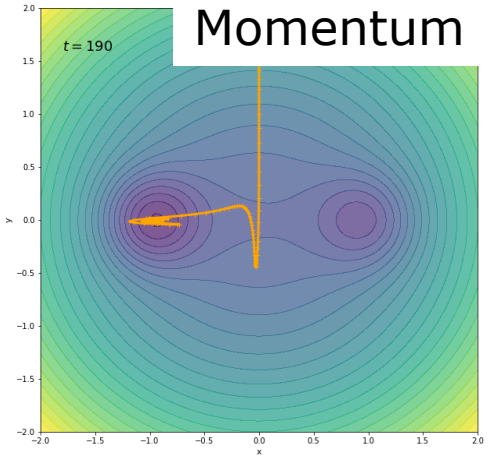
RMSProp

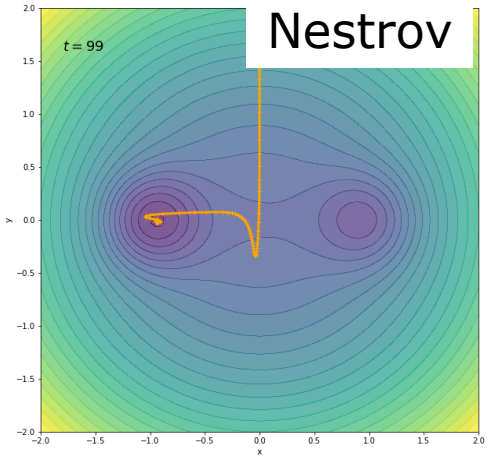


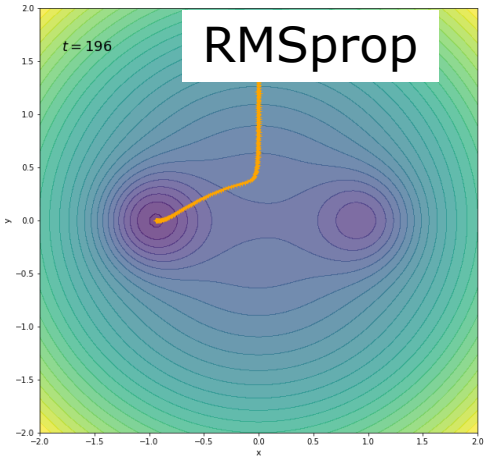
ADAM

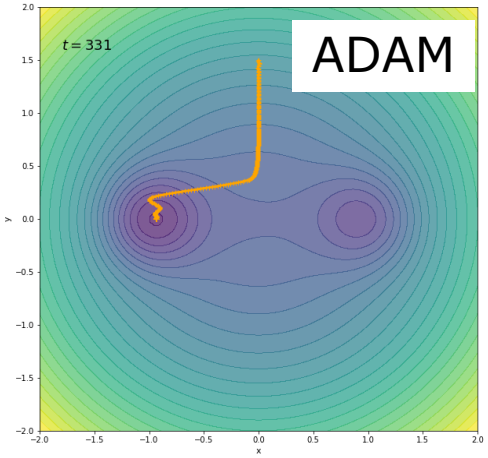
Which optimizer should we use?



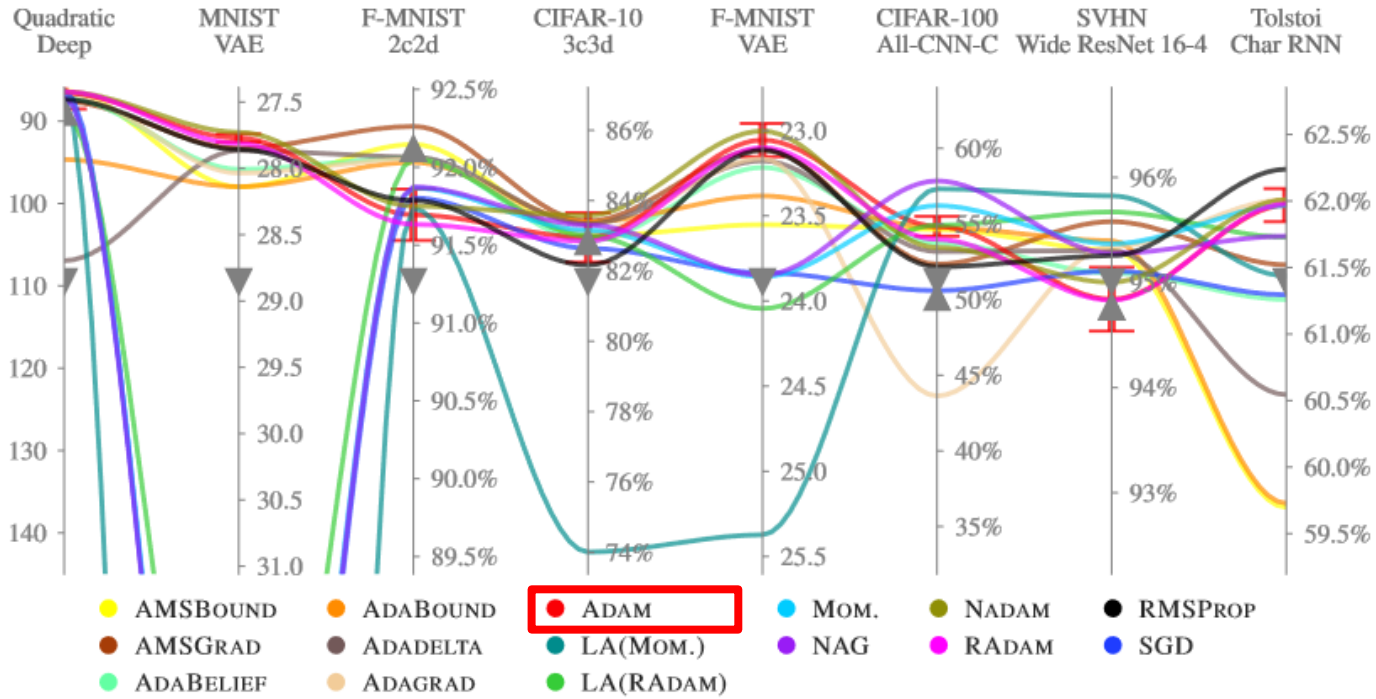








Experimental Study

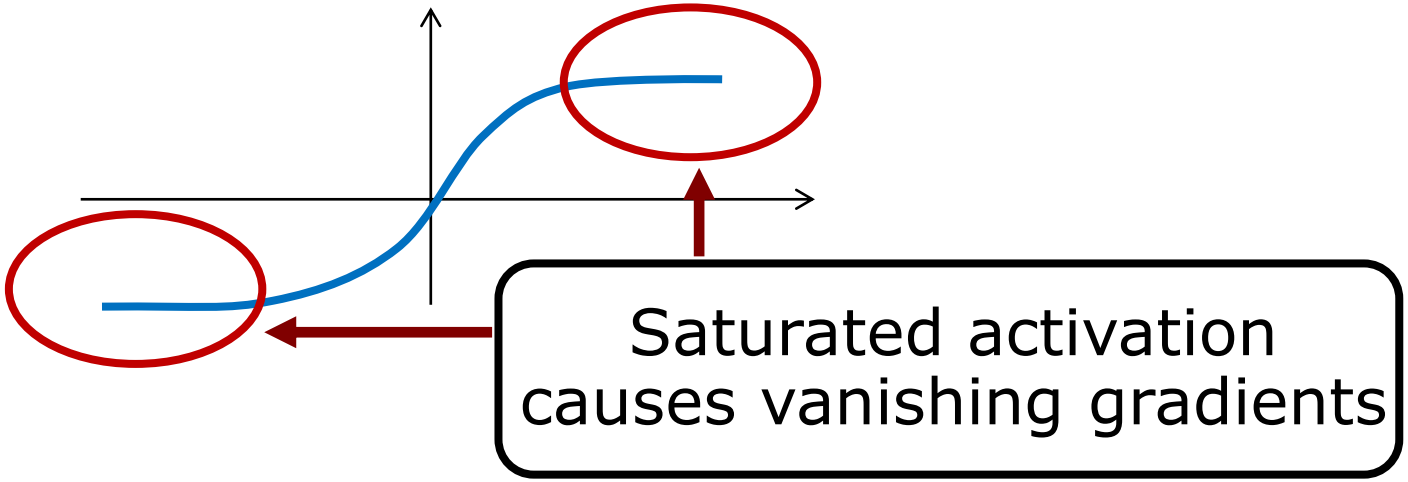


- Large set of experiments indicate good untuned performance of ADAM
- But generally no clear winner! (ADAM, RMSProp,…)
- Suggested receipt: run set of optimizers with default settings and pick best.

[Schmidt et al., 2021] Figure from [Schmidt et al., 2021]

Vanishing/Exploding Gradients

- Training deep networks might run into the problem of vanishing or exploding gradients
- **Example:** tanh activation

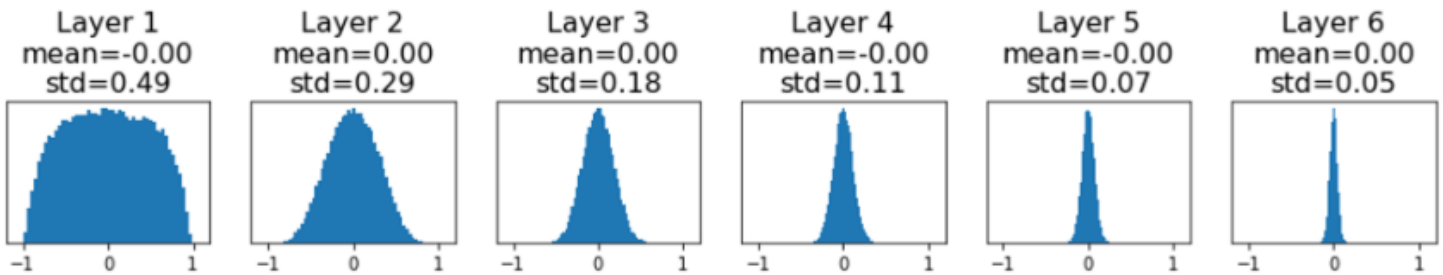


- Solution:
  1. Careful initialization of layers
  2. Normalization of layer outputs

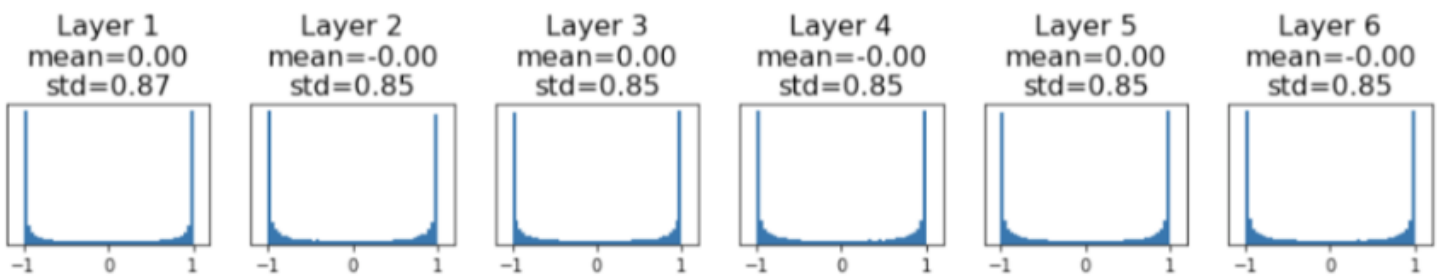
Initialization

- AlexNet used $w \sim \mathcal{N}(0, 0.01)$ for initialization
- Variance has large influence on learning!
  - Example: Fully-connected network with tanh

$w \sim \mathcal{N}(0, 0.01)$



$w \sim \mathcal{N}(0, 0.05)$



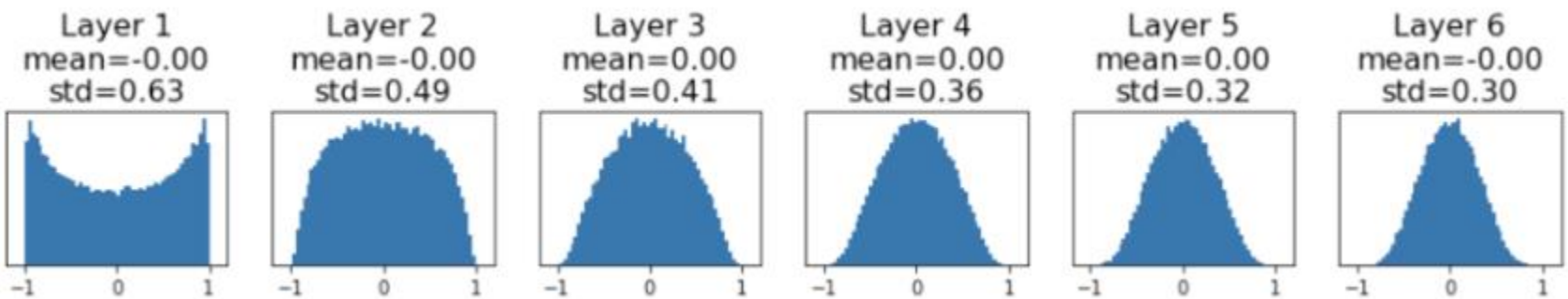
  - **Problem:** Already at initialization activations are saturated or zero

Figures from Justin Johnson's lecture

Xavier Initialization

- Initial weights are initialized with $w \sim \mathcal{N}(0, \frac{1}{\sqrt{D_{\text{in}}}})$
- D_{in} is the input dimensions/channels
- Convolutions: $D_{\text{in}} = K \cdot K \cdot C_{\text{in}}$
- Example: $D_{\text{in}} = 4096$

$$w \sim \mathcal{N}(0, \frac{1}{\sqrt{D_{\text{in}}}} = 0.015)$$

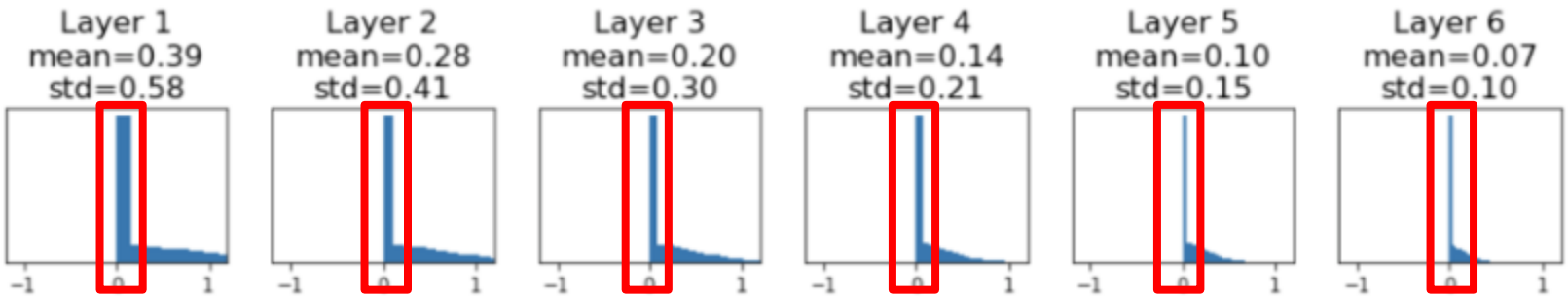


[Glorot, 2010]

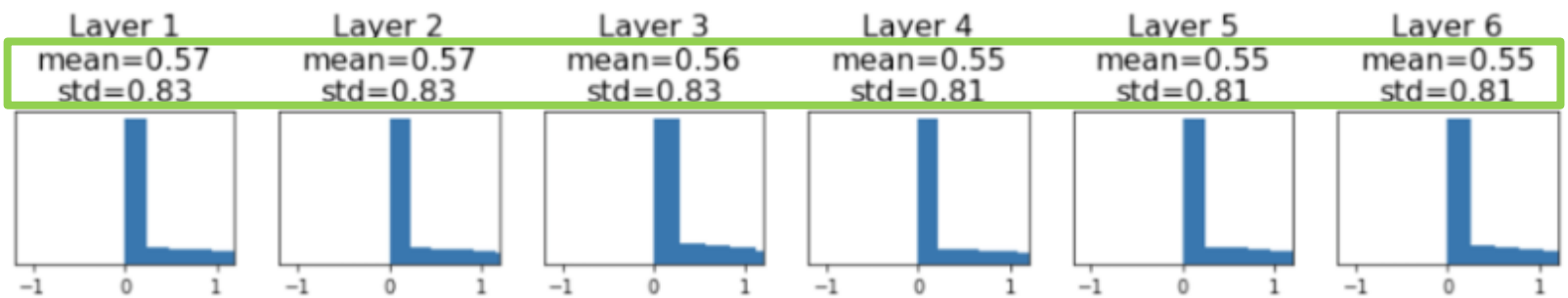
Figures from Justin Johnson's lecture

He Initialization

$w \sim \mathcal{N}(0, \frac{1}{\sqrt{D_{\text{in}}}} = 0.015)$ with ReLU



$w \sim \mathcal{N}(0, \sqrt{2}\frac{1}{\sqrt{D_{\text{in}}}} = \sqrt{2} \cdot 0.015)$

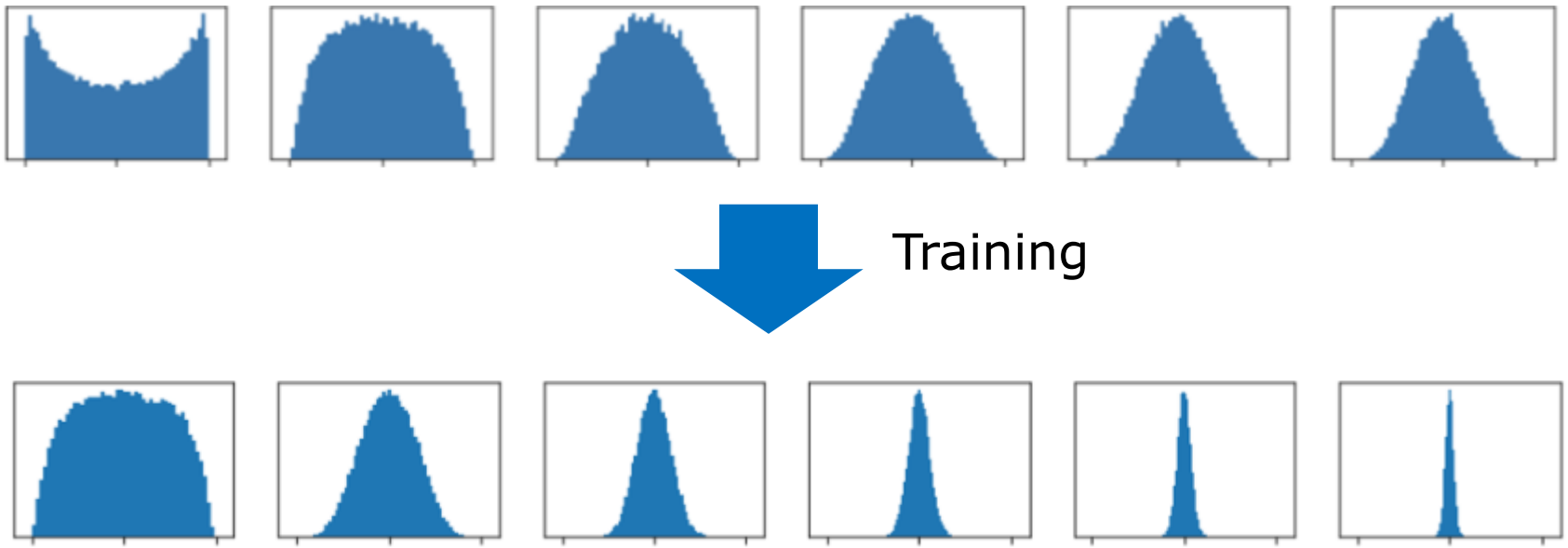


- Still with ReLU not optimal, since almost all activations near zero!
- For ReLUs: we need a factor of $\sqrt{2}$ (called **gain**)

[He, 2015]

Figures from Justin Johnson's lecture

Normalization

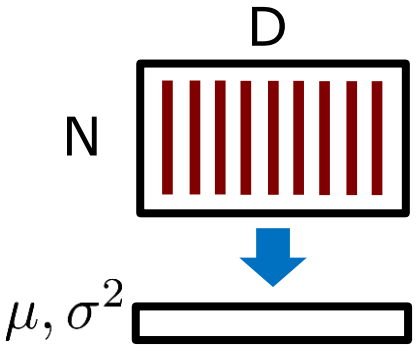


- Still can be problematic while training as activations can get too large or too small
- **Idea:** normalize layer outputs

Figures from Justin Johnson's lecture

Batch Normalization

- Compute mean and variance for each feature channel of batch:



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

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

- Learnable parameters γ and β change range:

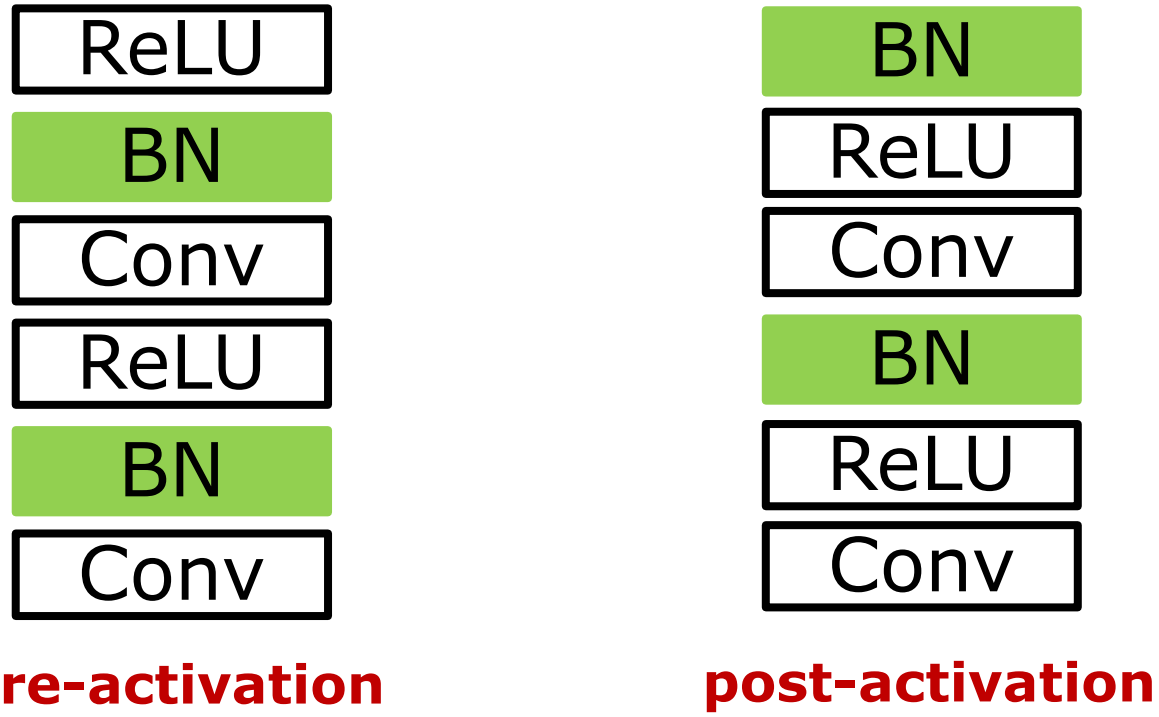
$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}\gamma + \beta$$

- Running average that can be used at test time

$$\bar{\mu}_j = 0.9\bar{\mu}_j + 0.1\mu_j \qquad \bar{\sigma}_j^2 = 0.9\bar{\sigma}_j^2 + 0.1\sigma_j$$

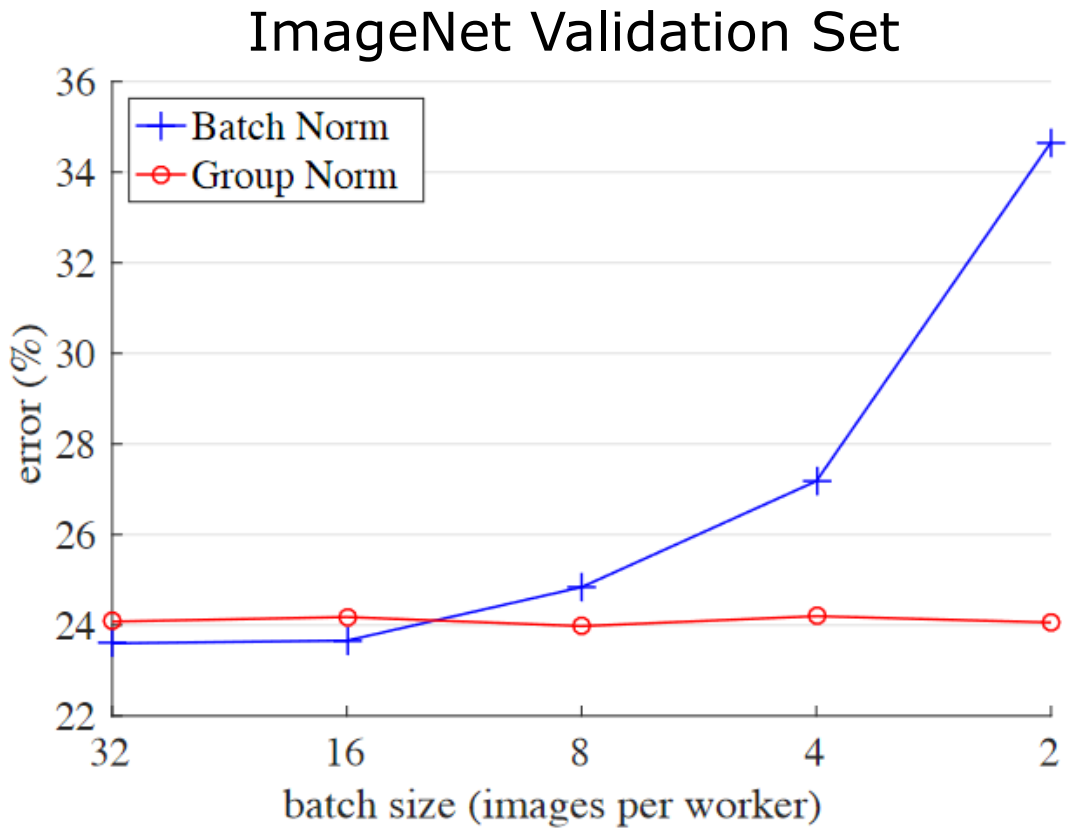
[Ioffe, 2015]

Pre- vs. Post-activation Norm



- Common: pre-activation normalization
- But also possible to use post-activation
- Empirically no big difference

Batch Norm with small batches

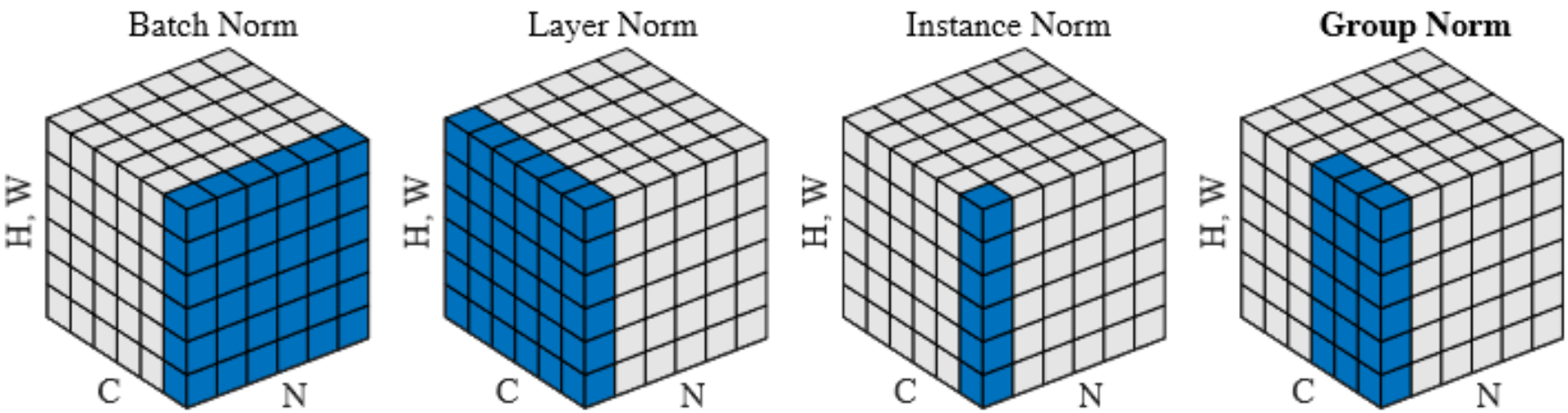


- With small batch size, BN statistics inaccurate and can have negative effect

[Wu et al., 2018]

Figure from [Wu et al., 2018]

Other Normalizations

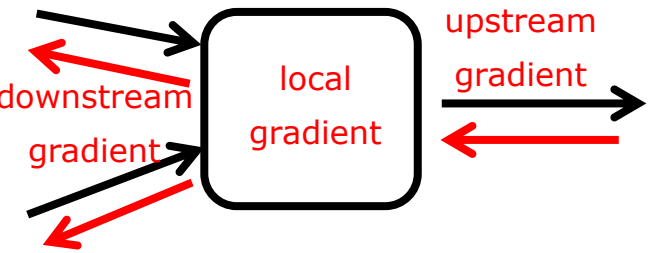


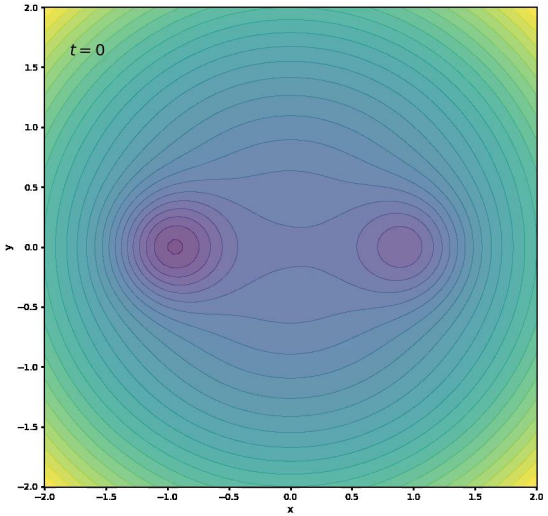
- Different ways to compute moments
- **Group Norm** provides good results for vision tasks with small batch sizes

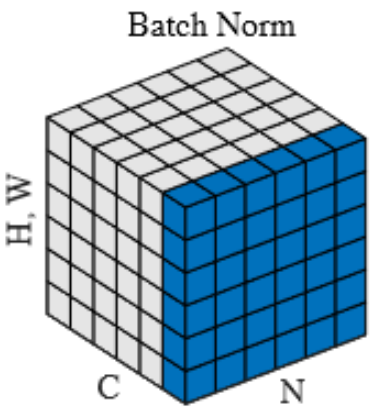
[Wu et al., 2018]

Figure from [Wu et al., 2018]

Summary







- Backpropagation for gradient computation
- Beyond gradient descent
- Good start for learning: Initialization
- Keep learning: Layer normalization

See you next week!

References

- Duchi et al. Adaptive Subgradient methods for online learning and stochastic Optimization. JMLR, 2011.
- Glorot et al. “Understanding the difficulty of training deep feedforward neural networks”, AISTAT, 2010.
- He et al. “Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification”, ICCV, 2015.
- Ioffe et al. “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift”, PMLR, vol. 37, 2015.
- Kingma et al. “Adam: A Method for Stochastic Optimization”, ICLR, 2015.
- Nesterov. “A method for solving the convex programming problem with convergence rate O(1/k2).” Soviet Mathematics Doklady, 1983.
- Polyak, “Some methods of speeding up the convergence of iteration methods.“ USSR Computational Mathematics and Mathematical Physics, 4(5), 1964.
- Robbins et al., “A Stochastic Approximation Method.” The Annals of Mathematical Statistics, 22(3), 1951
- Schmidt et al. “Descending through a Crowded Valley - Benchmarking Deep Learning Optimizers”, arxiv, 2021.
- Tieleman et al. “Lecture 6.5 – RMSProp, COURSERA: Neural Networks for Machine Learning”, 2012.
- Wu et al. “Group Normalization”, ECCV, 2018.