Optimization

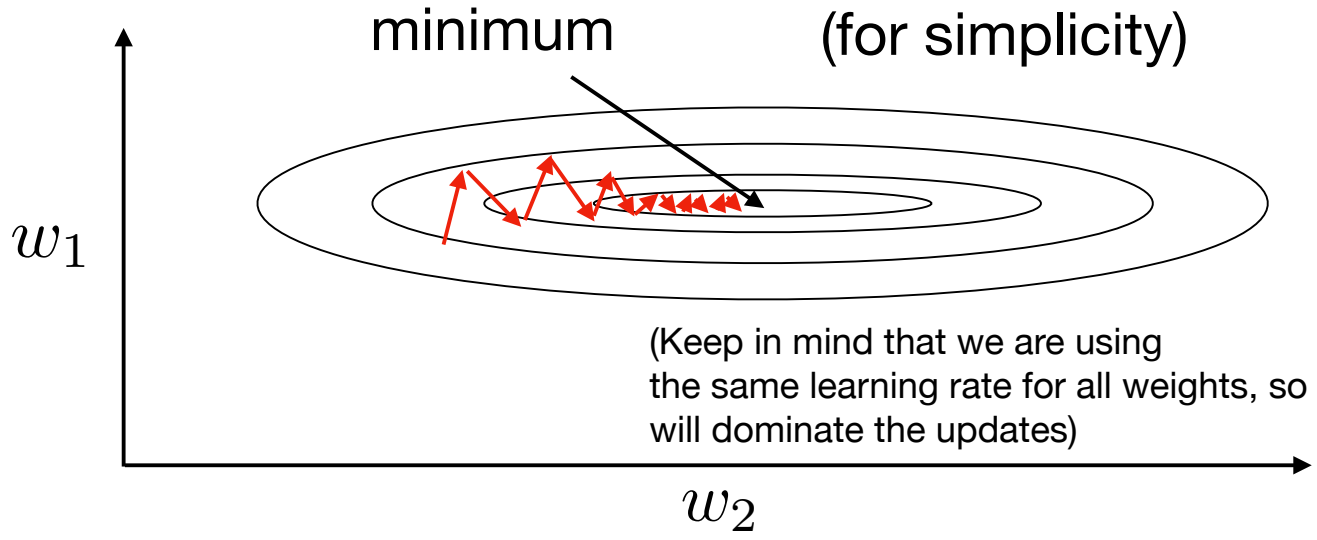
Deep Learning & Applications

Topics

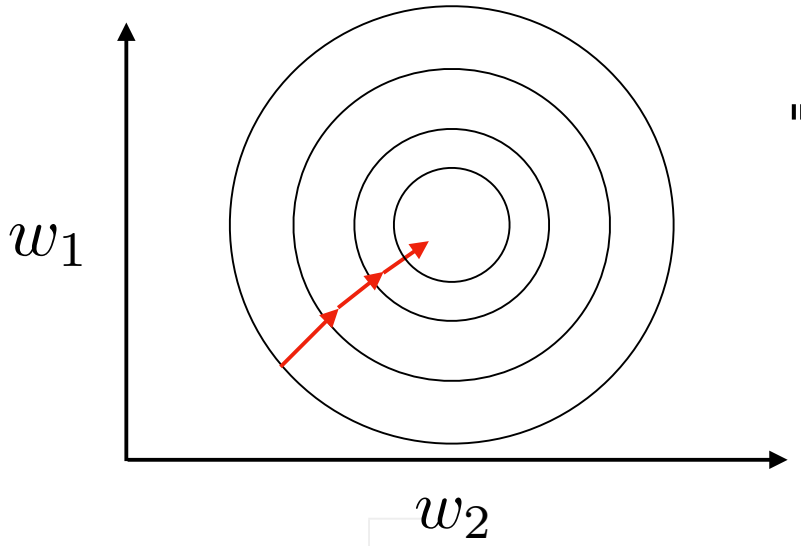
- Batch Normalization
- Weight Initialization
- Xavier and He Initialization
- Momentum
- Nesterov Accelerated Gradient (NAG)
- RMSProp
- Adam

Why We Normalize Inputs for Gradient Descent

Surface of a convex cost function
(for simplicity)



(Keep in mind that we are using
the same learning rate for all weights, so large parameters
will dominate the updates)



"Standardization" of input features

$$x_j'^{[i]} = \frac{x_j^{[i]} - \mu_j}{\sigma_j}$$

(scaled feature will have zero mean,
unit variance)

However, normalizing
the inputs to the network
only affects the first hidden layer ...
What about the other hidden layers?

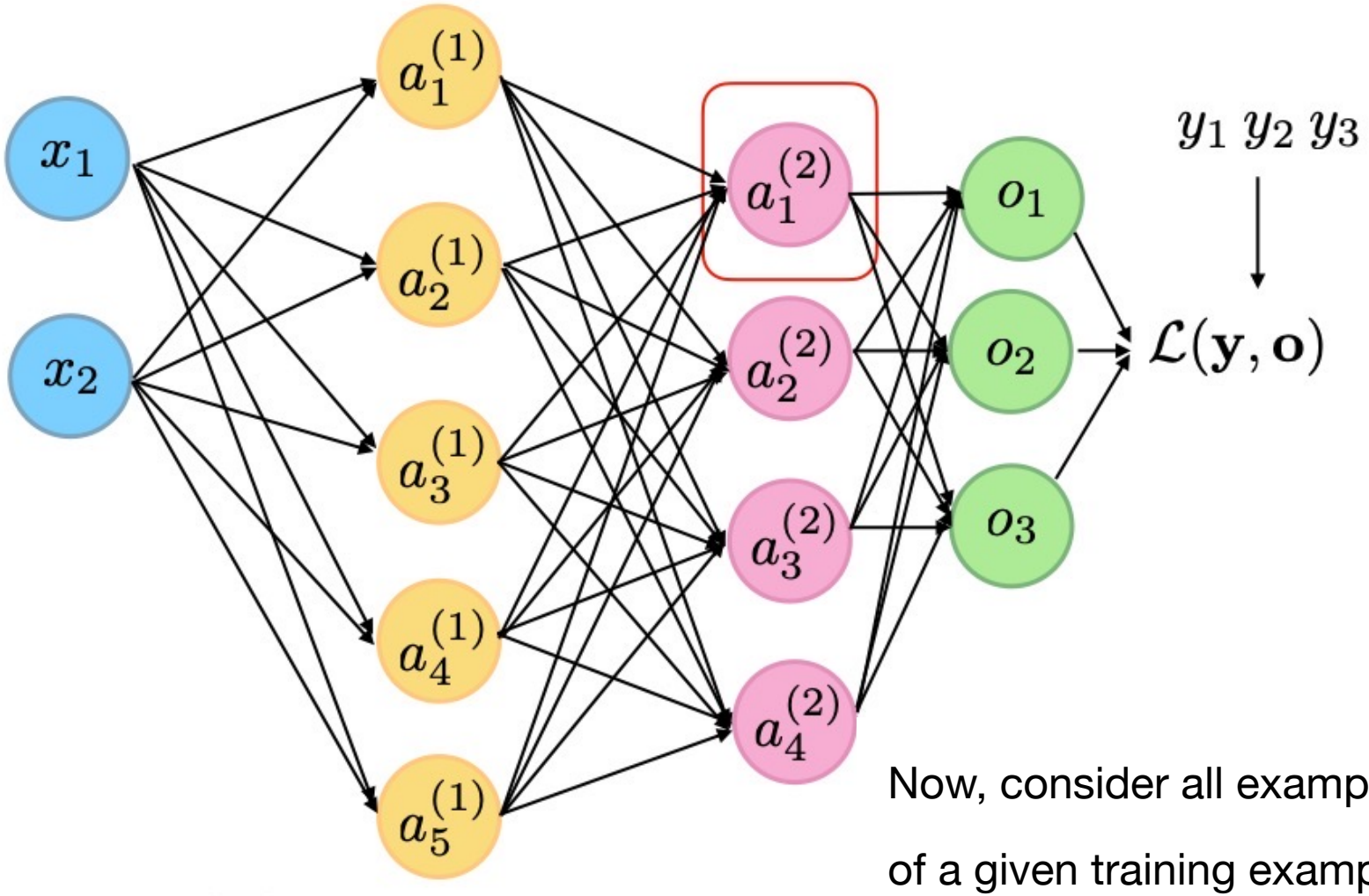
Batch Normalization ("BatchNorm")

Ioffe, S., & Szegedy, C. (2015). Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. In *International Conference on Machine Learning* (pp. 448-456).

Batch Normalization ("BatchNorm")

- Normalizes hidden layer inputs
- Helps with exploding/vanishing gradient problems
- Can increase training stability and convergence rate

Suppose, we have net input $z_1^{(2)}$
associated with an activation in the 2nd hidden layer



Now, consider all examples in a minibatch such that the net input of a given training example at layer 2 is written as $z_1^{(2)[i]}$

where $i \in \{1, ..., n\}$

BatchNorm Step 1: Normalize Net Inputs

$$\mu_j = \frac{1}{n}\sum_i z_j^{[i]}$$

$$\sigma_j^2 = \frac{1}{n}\sum_i (z_j^{[i]} - \mu_j)^2$$

$$z'^{[i]}_j = \frac{z_j^{[i]} - \mu_j}{\sigma_j}$$

In practice:

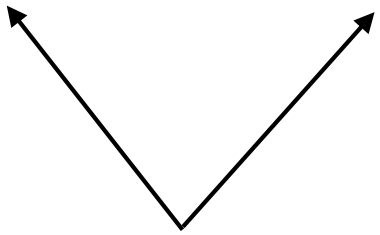
$$z'^{[i]}_j = \frac{z_j^{[i]} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}$$

For numerical stability, where epsilon is a small number like 1E-5

BatchNorm Step 2: Pre-Activation Scaling

$$z'^{[i]}_j = \frac{z^{[i]}_j - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}$$

$$a'^{[i]}_j = \gamma_j \cdot z'^{[i]}_j + \beta_j$$



These are learnable parameters

BatchNorm Step 2: Pre-Activation Scaling

$$z'^{[i]}_j = \frac{z^{[i]}_j - \mu_j}{\sigma_j}$$

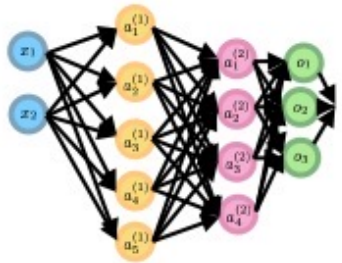
$$a'^{[i]}_j = \gamma_j \cdot z'^{[i]}_j + \beta_j$$

Controls the mean

Controls the spread or scale

Technically, a BatchNorm layer could learn to perform "standardization" with zero mean and unit variance

BatchNorm Step 1 & 2 Summarized



$$z'^{[i]}_j = \frac{z^{[i]}_j - \mu_j}{\sigma_j}$$

$$a'^{[i]}_j = \gamma_j \cdot z'^{[i]}_j + \beta_j$$

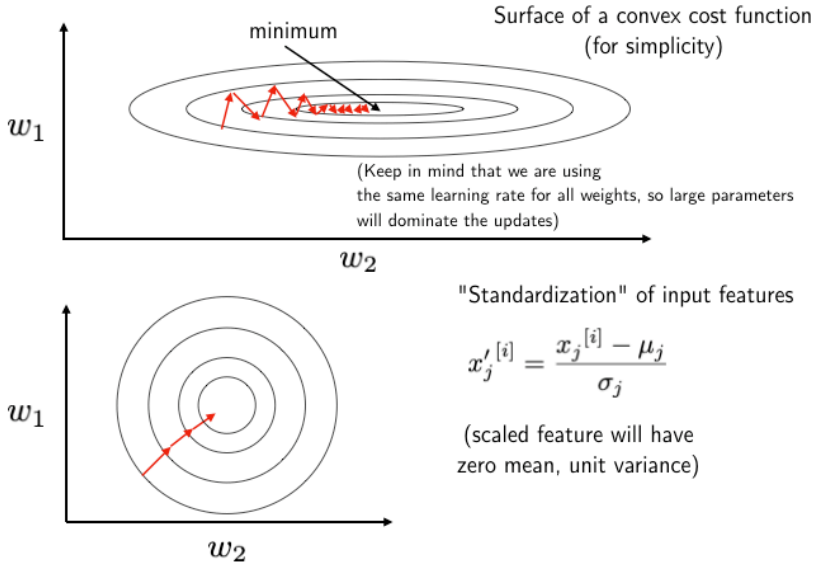
$x_1 \rightarrow z_1^{(1)} \rightarrow z'^{(1)}_1 \rightarrow a'^{(1)}_1 \rightarrow a_1^{(1)}$ first hidden layer

$z_1^{(2)} \rightarrow z'^{(2)}_1 \rightarrow a'^{(2)}_1 \rightarrow a_1^{(2)}$ second hidden layer

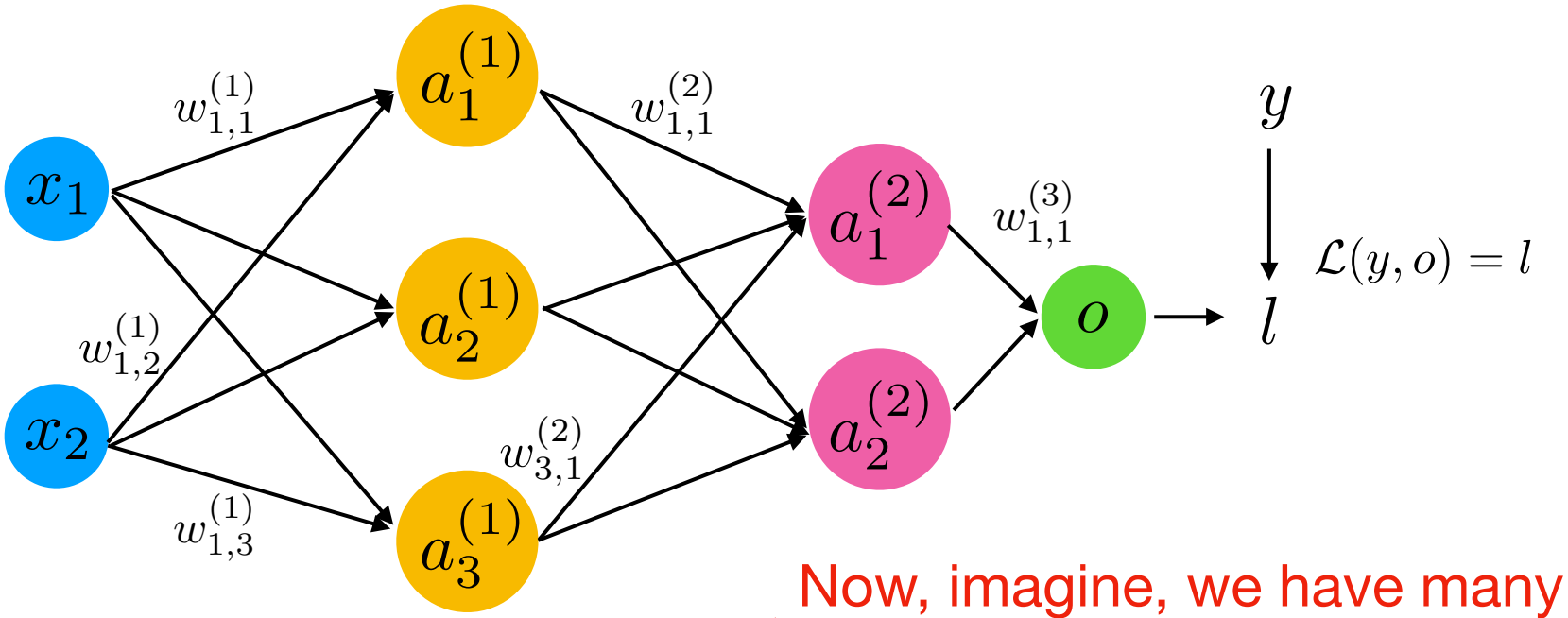
...

Weight Initialization

- We previously discussed that we want to initialize weight to small, random numbers to break symmetry
- Also, we want the weights to be relatively small, why?



Sidenote: Vanishing/Exploding Gradient Problems



Now, imagine, we have many layers and sigmoid activations ...

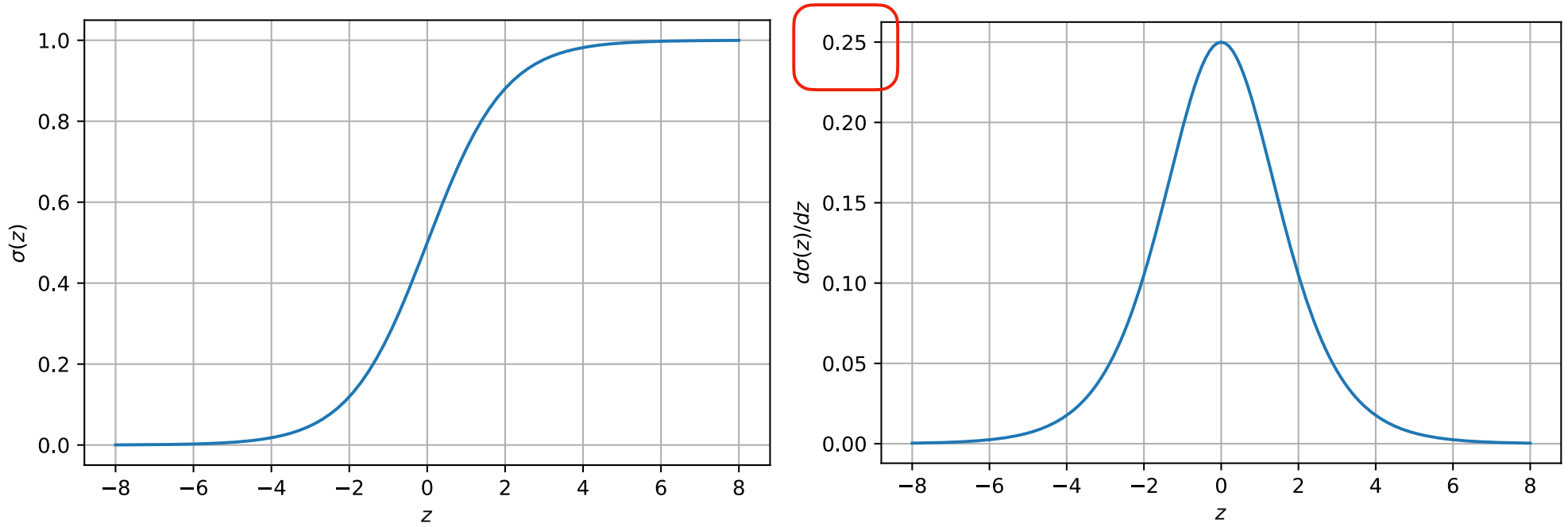
$$\frac{\partial l}{\partial w_{1,1}^{(1)}} = \frac{\partial l}{\partial o} \cdot \frac{\partial o}{\partial a_1^{(2)}} \cdot \frac{\partial a_1^{(2)}}{\partial a_1^{(1)}} \cdot \frac{\partial a_1^{(1)}}{\partial w_{1,1}^{(1)}}$$

$$+ \frac{\partial l}{\partial o} \cdot \frac{\partial o}{\partial a_2^{(2)}} \cdot \frac{\partial a_2^{(2)}}{\partial a_1^{(1)}} \cdot \frac{\partial a_1^{(1)}}{\partial w_{1,1}^{(1)}}$$

Sidenote: Vanishing/Exploding Gradient Problems

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

$$\frac{d}{dz}\sigma(z) = \frac{e^{-z}}{(1 + e^{-z})^2} = \sigma(z)(1 - \sigma(z))$$



we have the largest gradient $= 0.25$

Weight Initialization

- Traditionally, we can initialize weights by sampling from a random uniform distribution in range [0, 1], or better, [-0.5, 0.5]
- Or, we could sample from a Gaussian distribution with mean 0 and small variance (e.g., 0.1 or 0.01)

- TanH is a bit more robust regarding vanishing gradients (compared to logistic sigmoid)
- It still has the problem of saturation (near zero gradients if inputs are very large, positive or negative values)
- Xavier initialization is a small improvement for initializing weights for tanH

Weight Initialization -- Xavier Initialization

Xavier Glorot and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks." *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. 2010.

Method:

Step 1: Initialize weights from Gaussian or uniform distribution

Step 2: Scale the weights proportional to the number of inputs to the layer

(For the first hidden layer, that is the number of features in the dataset;
for the second hidden layer, that is the number of units in the 1st hidden layer
etc.)

Weight Initialization -- Xavier Initialization

Xavier Glorot and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks." *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. 2010.

Rationale behind this scaling:
Variance of the sample (between data points, not variance of the mean) linearly increases as the sample size increases

$$\mathbf{W}^{(l)} := \mathbf{W}^{(l)} \cdot \sqrt{\frac{1}{m^{(l-1)}}}$$

where m is the number of input units to the next layer

e.g.,

$$W_{i,j}{}^{(l)} \sim N(\mu = 0, \sigma^2 = 0.01)$$

(or uniform distr. in a fixed interval, as in the original paper)

Weight Initialization -- He Initialization

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification." In *Proceedings of the IEEE international conference on computer vision*, pp. 1026-1034. 2015.

- Assuming activations with mean 0, which is reasonable, Xavier Initialization assumes a derivative of 1 for the activation function (which is reasonable for tanH)
- For ReLU, this is different, as the activations are not centered at zero anymore
- He initialization takes this into account (to see that worked out in math, see the paper)
- The result is that we add a scaling factor of $2^{0.5}$

$$\mathbf{W}^{(l)} := \mathbf{W}^{(l)} \cdot \sqrt{\frac{2}{m^{(l-1)}}}$$

Again, some DL jargon: This is sometimes called "fan in" (= number of inputs to a layer)

Glorot

Assume a network layer with m inputs, n outputs

$$W \sim \mathcal{U}\left(-\sqrt{\frac{6}{m+n}}, \sqrt{\frac{6}{m+n}}\right) \quad Glorot\ Uniform$$

$$W \sim \mathcal{N}\left(w; 0, \sqrt{\frac{2}{m+n}}\right) \quad Glorot\ Normal$$

He

Assume a network layer with m inputs

$$W \sim \mathcal{U}\left(-\sqrt{\frac{6}{m}}, \sqrt{\frac{6}{m}}\right) \quad He\ Uniform$$

$$W \sim \mathcal{N}\left(w; 0, \sqrt{\frac{2}{m}}\right) \quad He\ Normal$$

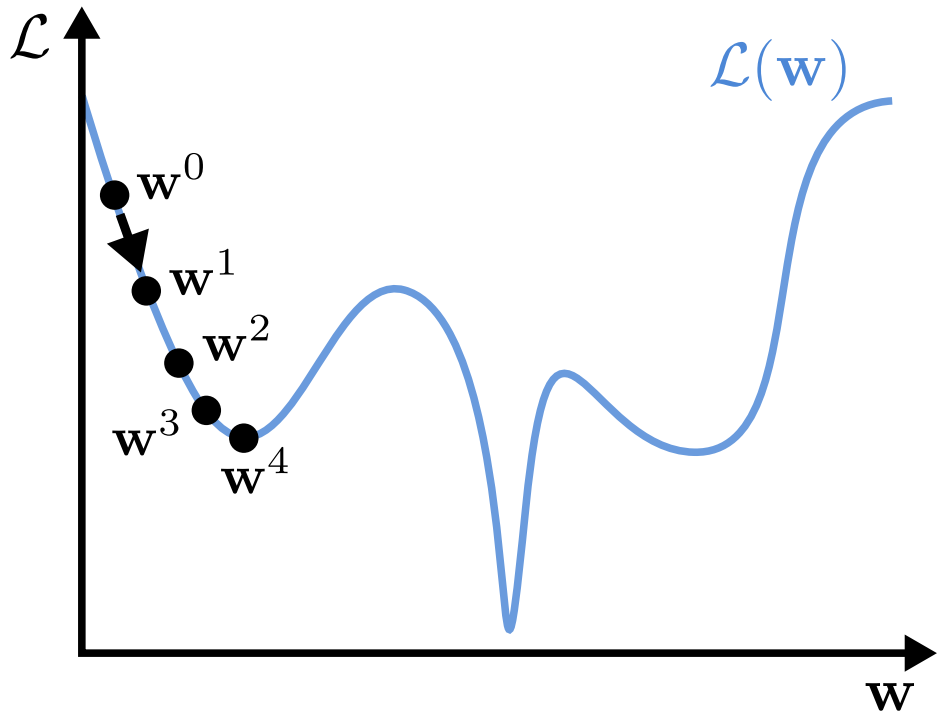
Optimizers

Optimization Challenges

Gradient Descent:

$$\mathbf{w}^0 = \mathbf{w}^{\text{init}}$$

$$\mathbf{w}^{t+1} = \mathbf{w}^t - \eta \nabla_{\mathbf{w}} \mathcal{L}(\mathbf{w}^t)$$



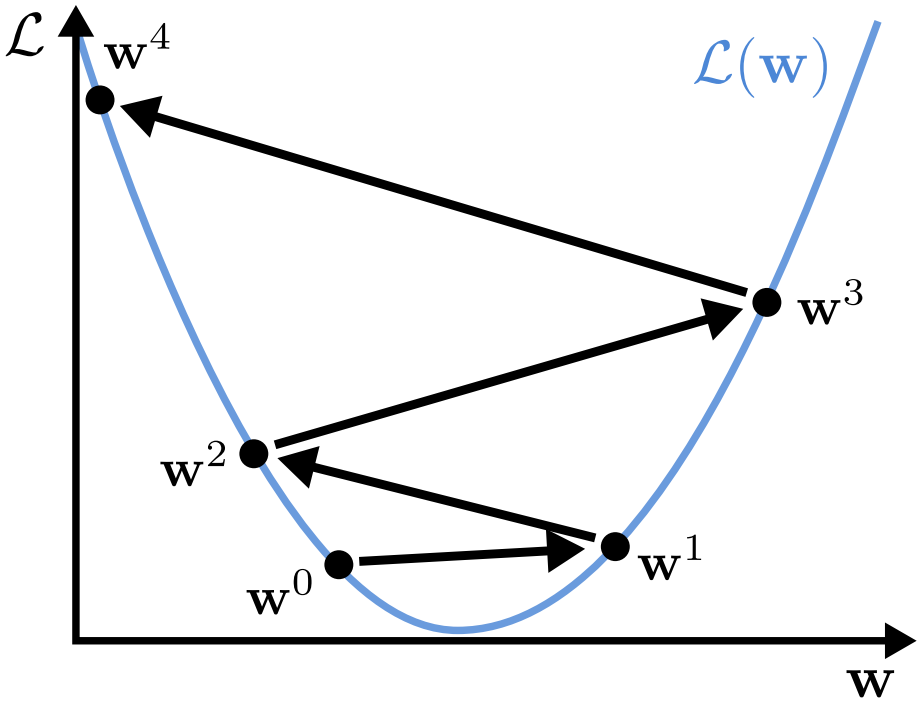
- Neural network loss $\mathcal{L}(\mathbf{w})$ is **not convex** wrt. the network parameters $\mathbf{w}$
- There exist **multiple local minima**, but we will find only one through optimization

Optimization Challenges

Gradient Descent:

$$\mathbf{w}^0 = \mathbf{w}^{\text{init}}$$

$$\mathbf{w}^{t+1} = \mathbf{w}^t - \eta \nabla_{\mathbf{w}} \mathcal{L}(\mathbf{w}^t)$$



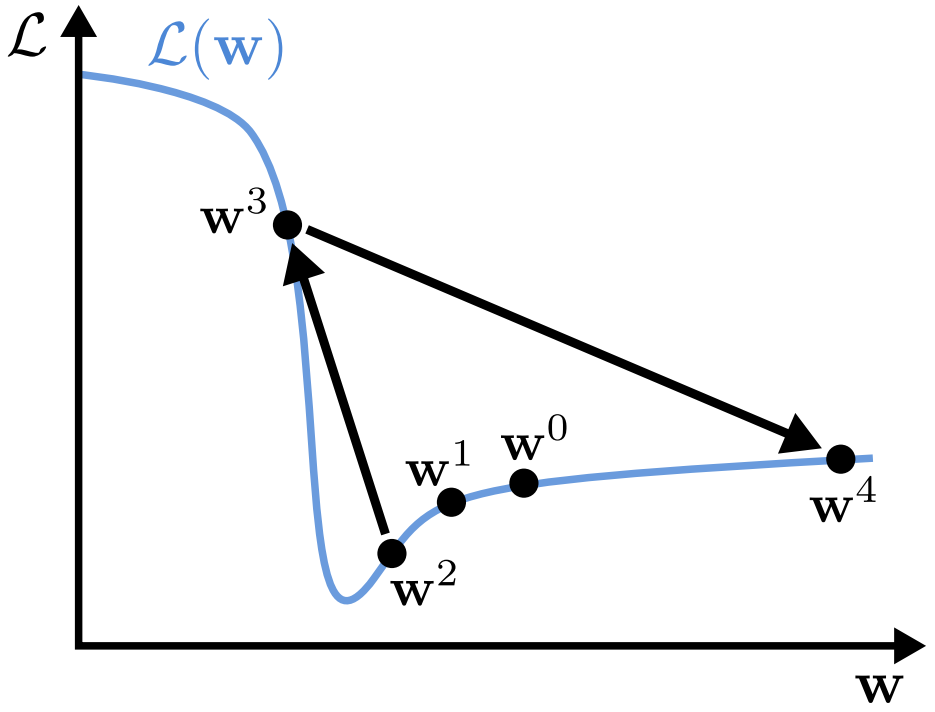
- Choosing the learning rate too low leads to very **slow progress**
- Choosing the learning rate too high might lead to **divergence**

Optimization Challenges

Gradient Descent:

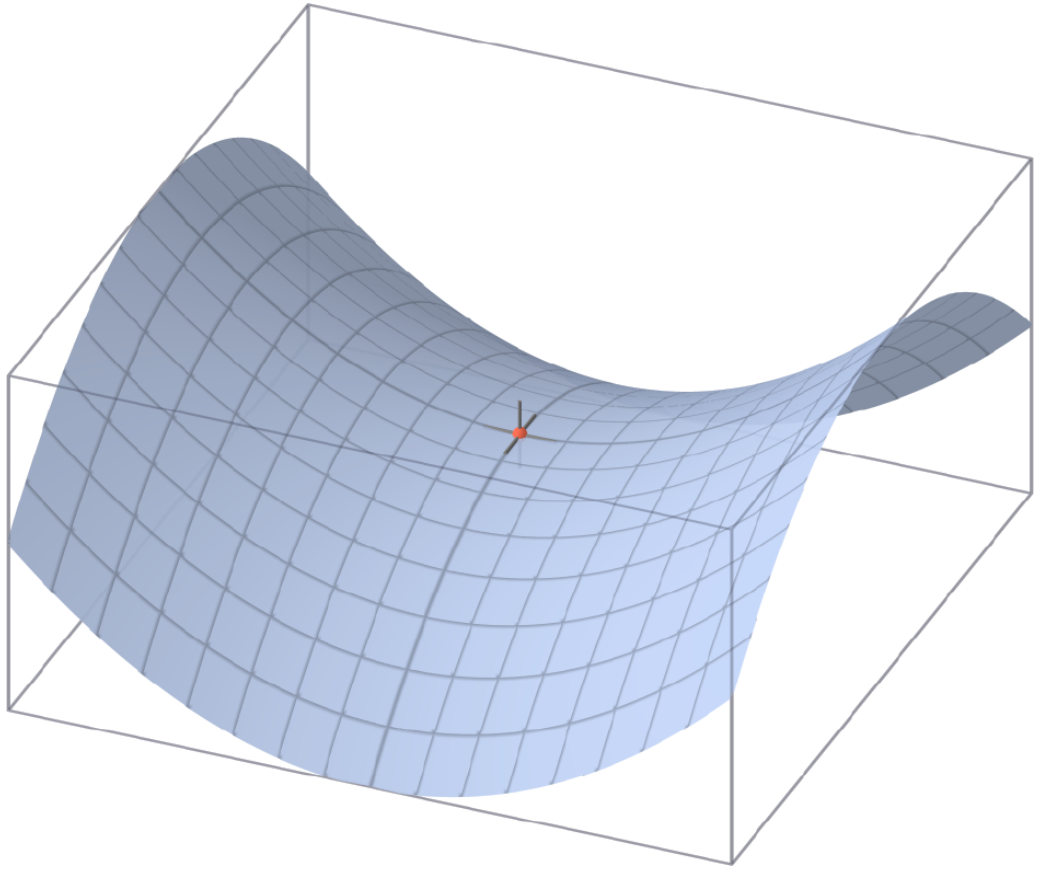
$$\mathbf{w}^0 = \mathbf{w}^{\text{init}}$$

$$\mathbf{w}^{t+1} = \mathbf{w}^t - \eta \nabla_{\mathbf{w}} \mathcal{L}(\mathbf{w}^t)$$



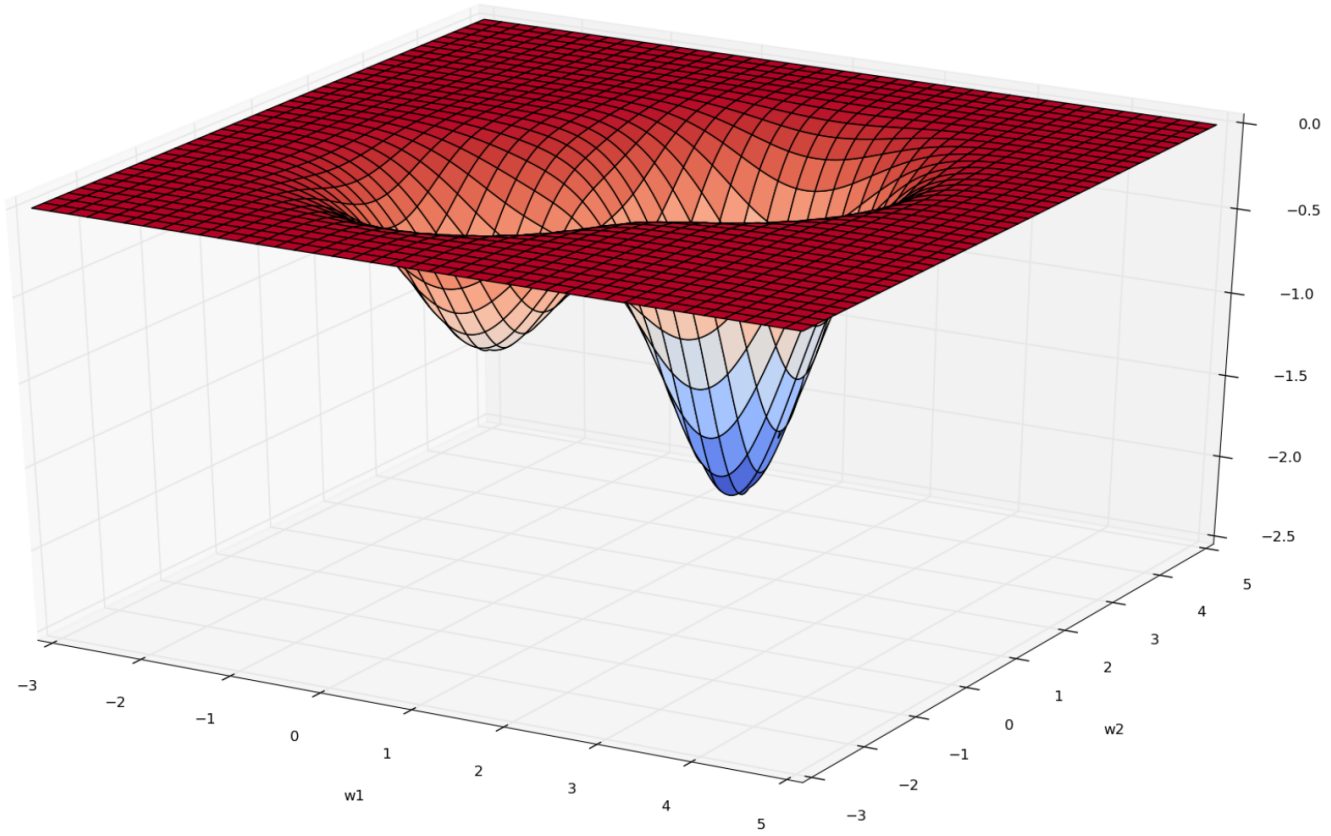
- Steep **cliffs** can pose great challenges to optimization
- Very high derivatives **catapult** the parameters $\mathbf{w}$ very far off
- **Gradient clipping** is a common heuristics to counteract such effects

Optimization Challenges



- At a **saddle point**, we have $\nabla_{\mathbf{w}} \mathcal{L}(\mathbf{w}) = \mathbf{0}$, but we are not at a minimum
- Many saddle points in DL, but only problematic if we exactly “hit” the saddle point

Optimization Challenges



- A flat region is called a **plateau** where $\nabla_{\mathbf{w}} \mathcal{L}(\mathbf{w}) \approx \mathbf{0} \Rightarrow$ **Slow progress**
- Example: Saturated sigmoid activation function, dead ReLUs, etc.

Gradient Descent

Algorithm:

1. Initialize weights $\mathbf{w}^0$ and pick learning rate η
2. For all data points $i \in \{1, \ldots, N\}$ do:
   2.1 Forward propagate $\mathbf{x}_i$ through network to calculate prediction $\hat{\mathbf{y}}_i$
   2.2 Backpropagate to obtain gradient $\nabla_{\mathbf{w}} \mathcal{L}_i(\mathbf{w}^t) \equiv \nabla_{\mathbf{w}} \mathcal{L}(\hat{\mathbf{y}}_i, \mathbf{y}_i, \mathbf{w}^t)$
3. Update gradients: $\mathbf{w}^{t+1} = \mathbf{w}^t - \eta \frac{1}{N} \sum_i \nabla_{\mathbf{w}} \mathcal{L}_i(\mathbf{w}^t)$
4. If validation error decreases, go to step 2, otherwise stop

Remark:

- Typically, millions of parameter $\Rightarrow \dim(\mathbf{w}) =$ 1 million or more
- Typically, millions of training points $\Rightarrow N =$ 1 million or more
- Becomes extremely expensive to compute and doesn't fit into memory

Stochastic Gradient Descent

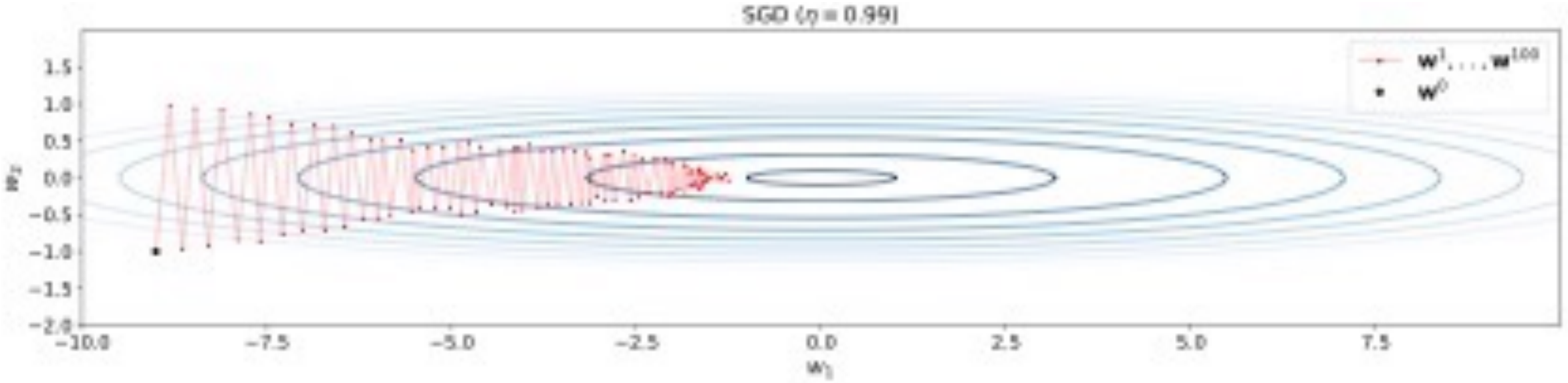
Algorithm:

1. Initialize weights $\mathbf{w}^0$, pick learning rate η and minibatch size $|\mathcal{X}_{\text{batch}}|$
2. Draw random minibatch $\{(\mathbf{x}_1, \mathbf{y}_1), \ldots, (\mathbf{x}_B, \mathbf{y}_B)\} \subseteq \mathcal{X}$ (with $B \ll N$)
3. For all minibatch elements $b \in \{1, \ldots, B\}$ do:
   3.1 Forward propagate $\mathbf{x}_b$ through network to calculate prediction $\hat{\mathbf{y}}_b$
   3.2 Backpropagate to obtain batch element gradient $\nabla_{\mathbf{w}} \mathcal{L}_b(\mathbf{w}^t) \equiv \nabla_{\mathbf{w}} \mathcal{L}(\hat{\mathbf{y}}_b, \mathbf{y}_b, \mathbf{w}^t)$
4. Update gradients: $\mathbf{w}^{t+1} = \mathbf{w}^t - \eta \frac{1}{B} \sum_b \nabla_{\mathbf{w}} \mathcal{L}_b(\mathbf{w}^t)$
5. If validation error decreases, go to step 2, otherwise stop

Remark:

- Allows for training with limited (GPU) memory, by splitting data into chunks
- Introduces stochasticity, batch gradients approximate the true gradient

Training with "Momentum"



Motivation for SGD with Momentum:

- SGD oscillates along w_2 axis $\Rightarrow$ we should dampen, e.g., by averaging over time
- SGD makes slow progress along w_1 axis $\Rightarrow$ we like to accelerate in this direction

Training with "Momentum"

Often referred to as "velocity" v

"velocity" from the previous iteration

$$\Delta w_{i,j}(t) := \alpha \cdot \Delta w_{i,j}(t-1) + \eta \cdot \frac{\partial \mathcal{L}}{\partial w_{i,j}}(t)$$

Usually, we choose a momentum rate between 0.9 and 0.999; you can think of it as a "friction" or "dampening" parameter

Regular partial derivative/ gradient multiplied by learning rate at current time step t

Weight update using the velocity vector:

$$w_{i,j}(t+1) := w_{i,j}(t) - \Delta w_{i,j}(t)$$

Qian, N. (1999). On the momentum term in gradient descent learning algorithms. Neural Networks : The Official Journal of the International Neural Network Society, 12(1), 145–151. http://doi.org/10.1016/S0893-6080(98)00116-6

Nesterov Momentum

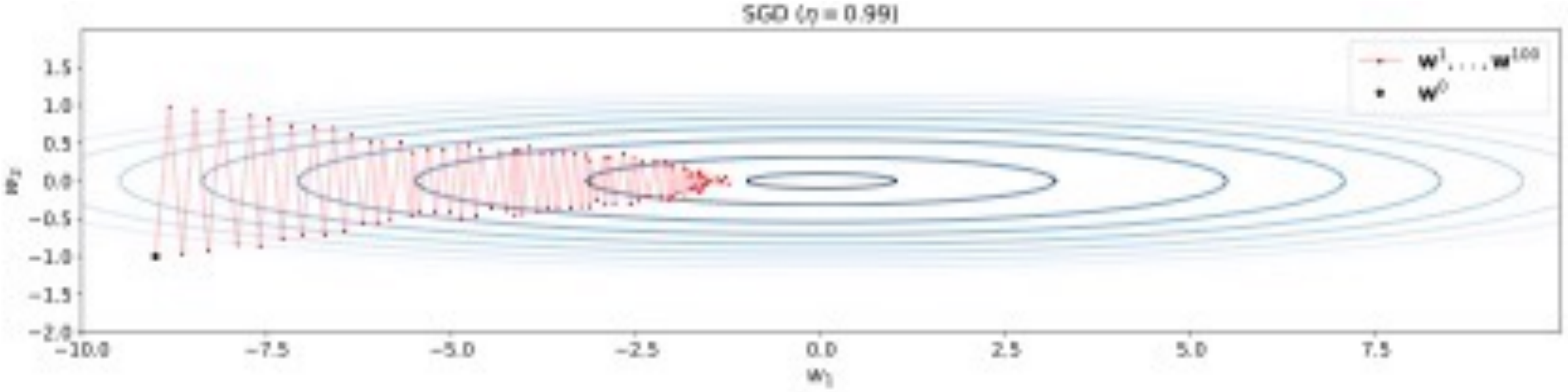
Better: **Look ahead** and calculate the gradient wrt. predicted parameters $\hat{w}_{i,j}(t+1)$

$$\hat{w}_{i,j}(t+1) := w_{i,j}(t) - \alpha \cdot \Delta w_{i,j}(t-1)$$

$$\Delta w_{i,j}(t) := \alpha \cdot \Delta w_{i,j}(t-1) + \eta \cdot \frac{\partial \mathcal{L}}{\partial \hat{w}_{i,j}}(t+1)$$

$$w_{i,j}(t+1) := w_{i,j}(t) - \Delta w_{i,j}(t)$$

Adaptive Learning Rates



Key take-aways:

- decrease learning if the gradient changes its direction
- increase learning if the gradient stays consistent

Adaptive Learning Rates

Key take-aways:

- decrease learning if the gradient changes its direction
- increase learning if the gradient stays consistent

Step 1: Define a local gain (*g*) for each weight (initialized with *g=1*)

$$\Delta w_{i,j} := \eta \cdot g_{i,j} \cdot \frac{\partial \mathcal{L}}{\partial w_{i,j}}$$

Adaptive Learning Rates

Step 1: Define a local gain (*g*) for each weight (initialized with *g=1*)

$$\Delta w_{i,j} := \eta \cdot g_{i,j} \cdot \frac{\partial \mathcal{L}}{\partial w_{i,j}}$$

Step 2:

If gradient is consistent

$$g_{i,j}(t) := g_{i,j}(t-1) + \beta$$

else

$$g_{i,j}(t) := g_{i,j}(t-1) \cdot (1-\beta)$$

Note that
multiplying by a factor has a larger impact if gains are large, compared to adding a term

(dampening effect if updates oscillate in the wrong direction)

Adaptive Learning Rate via RMSProp

- Unpublished algorithm by Geoff Hinton (but very popular) based on Rprop [1]
- Very similar to another concept called AdaDelta
- Concept: divide learning rate by an exponentially decreasing moving average of the squared gradients
- This takes into account that gradients can vary widely in magnitude
- Here, RMS stands for "Root Mean Squared"
- Also, damps oscillations like momentum (but in practice, works a bit better)

[1] Igel, Christian, and Michael Hüsken. "Improving the Rprop learning algorithm." *Proceedings of the Second International ICSC Symposium on Neural Computation (NC 2000)*. Vol. 2000. ICSC Academic Press, 2000.

Adaptive Learning Rate via RMSProp

$$MeanSquare(w_{i,j}, t) := \beta \cdot MeanSquare(w_{i,j}, t-1) + (1-\beta)\left(\frac{\partial \mathcal{L}}{w_{i,j}(t)}\right)^2$$

moving average of the squared gradient for each weight

$$w_{i,j}(t) := w_{i,j}(t) - \eta \cdot \frac{\partial \mathcal{L}}{\partial w_{i,j}(t)} / \left(\sqrt{MeanSquare\,(w_{i,j}, t)} + \epsilon\right)$$

where beta is typically between 0.9 and 0.999

small epsilon term to
avoid division by zero

Adaptive Learning Rate via ADAM

- ADAM (Adaptive Moment Estimation) is probably the most widely used optimization algorithm in DL as of today
- It is a combination of the momentum method and RMSProp

Momentum-like term:

$$\Delta w_{i,j}(t) := \alpha \cdot \Delta w_{i,j}(t-1) + \eta \cdot \frac{\partial \mathcal{L}}{\partial w_{i,j}}(t)$$

m_{t-1}

original momentum term

n_t

$$m_t := \alpha \cdot m_{t-1} + (1-\alpha) \cdot \frac{\partial \mathcal{L}}{\partial w_{i,j}}(t)$$

Kingma, D. P., & Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Adaptive Learning Rate via ADAM

Momentum-like term:

$$m_t := \alpha \cdot m_{t-1} + (1 - \alpha) \cdot \frac{\partial \mathcal{L}}{\partial w_{i,j}}(t)$$

RMSProp term:

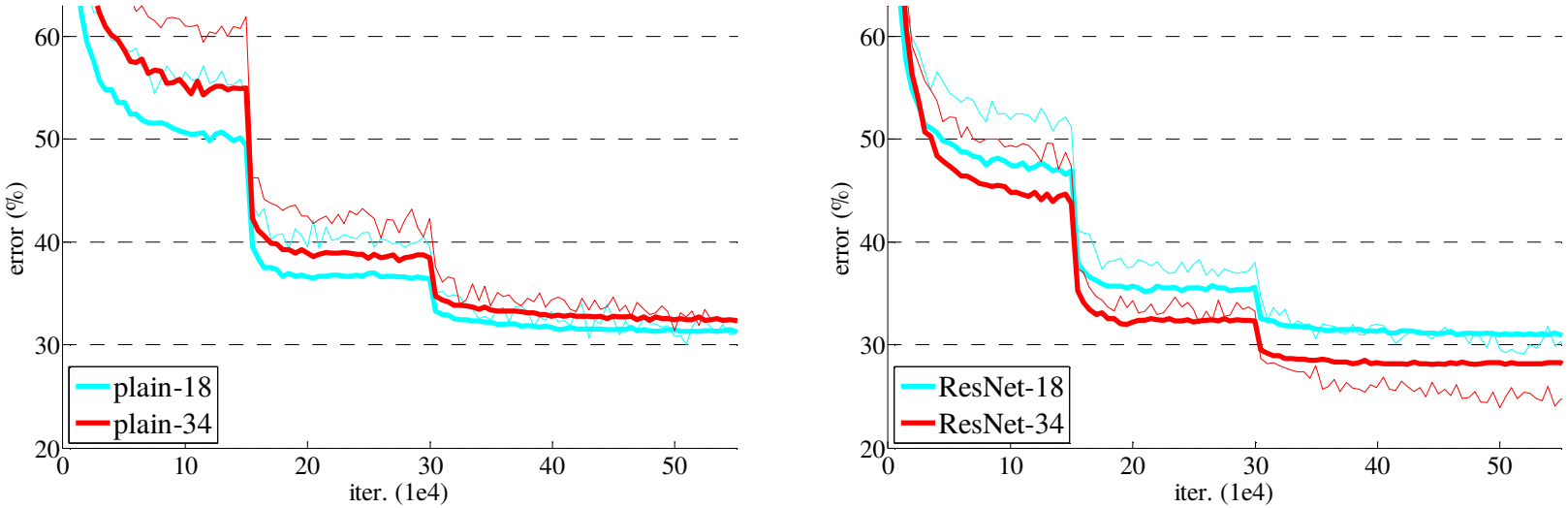
$$r := \beta \cdot MeanSquare(w_{i,j}, t-1) + (1 - \beta) \left(\frac{\partial \mathcal{L}}{\partial w_{i,j}(t)} \right)^2$$

ADAM update:

$$w_{i,j} := w_{i,j} - \eta \frac{m_t}{\sqrt{r} + \epsilon}$$

Kingma, D. P., & Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Learning Rate Schedules



Learning rate schedules:

- Fixed learning rate (not a good idea: too slow in the beginning and fast in the end)
- Inverse proportional decay: $\eta_t = \eta/t$ (Robbins and Monro)
- Exponential decay: $\eta_t = \eta\alpha^t$
- Step decay: $\eta \leftarrow \alpha\eta$ (every K iterations/epochs, common in practice: $\alpha = 0.5$)

He, Zhang, Ren andSun: Deep Residual Learning for Image Recognition. CVPR, 2016.

Hyperparameter Search

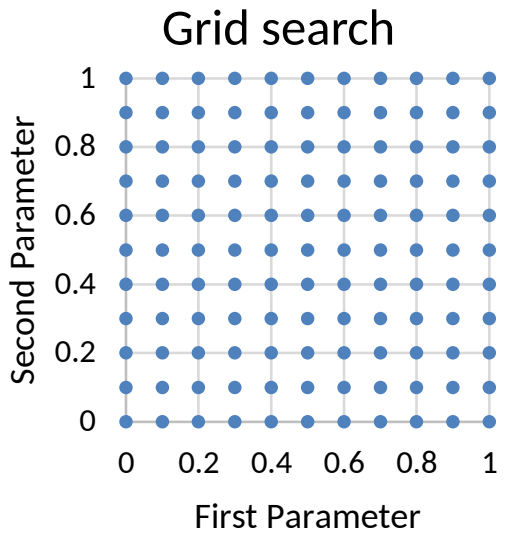
Hyperparameters:

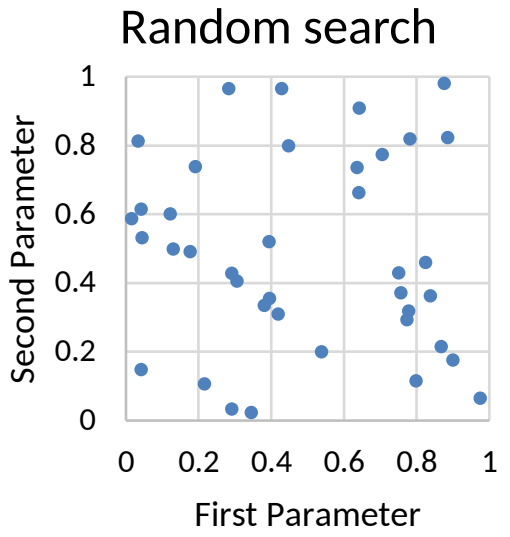
- Network architecture
- Number of iterations
- Batch size
- Learning rate schedule
- Regularization

Hyperparameter Search

Methods:

- Manual search
  - Most common
  - Build intuitions
- Grid search
  - Define ranges
  - Systematically evaluate
  - Requires large resources
- Random search
  - Like grid search but hyperparameters selected based on random draws





How to Start

1. Start with single training sample and use a small network

- First verify that the output is correct
- Then overfit, accuracy should be 100%, fast training/debug cycles
- Choose a good learning rate $(0.1, 0.01, 0.001, ..)$

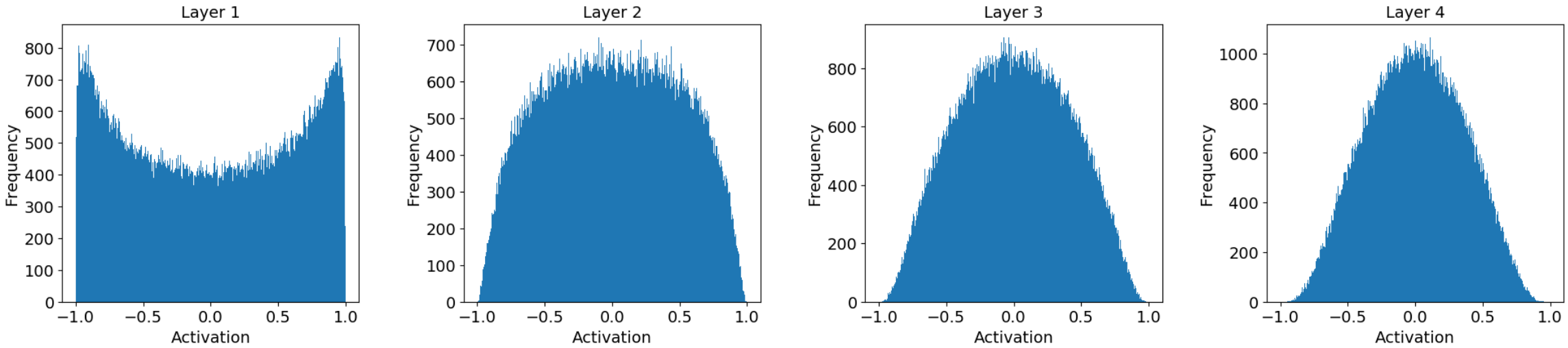
2. Increase to 10 training samples

- Again, verify that the output is correct
- Measure time for one iteration $(< 1s) \Rightarrow$ identify bottlenecks (e.g., data loading)
- Overfit to 10 samples, accuracy should be near 100%

3. Increase to 100, 1000, 10000 samples and increase network size

- Plot train and validation error $\Rightarrow$ now you should start to see generalization
- Important: Make only one change at a time to identify causes

Initialization



Xavier/He Initialization:

- Initialize such that activation distribution constant Gaussian across all layers

Batch Normalization

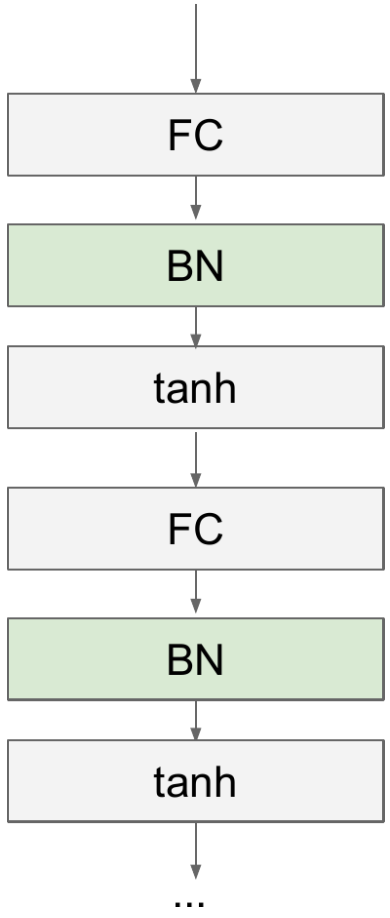
$$\mu_c = \frac{1}{B}\sum_{b=1}^{B} x_{b,c}$$

$$\sigma_c^2 = \frac{1}{B}\sum_{b=1}^{B}(x_{b,c} - \mu_c)^2$$

$$\hat{x}_{b,c} = \frac{x_{b,c} - \mu_c}{\sqrt{\sigma_c^2 + \epsilon}}$$

$$y_{b,c} = \gamma_c \, \hat{x}_{b,c} + \beta_c$$

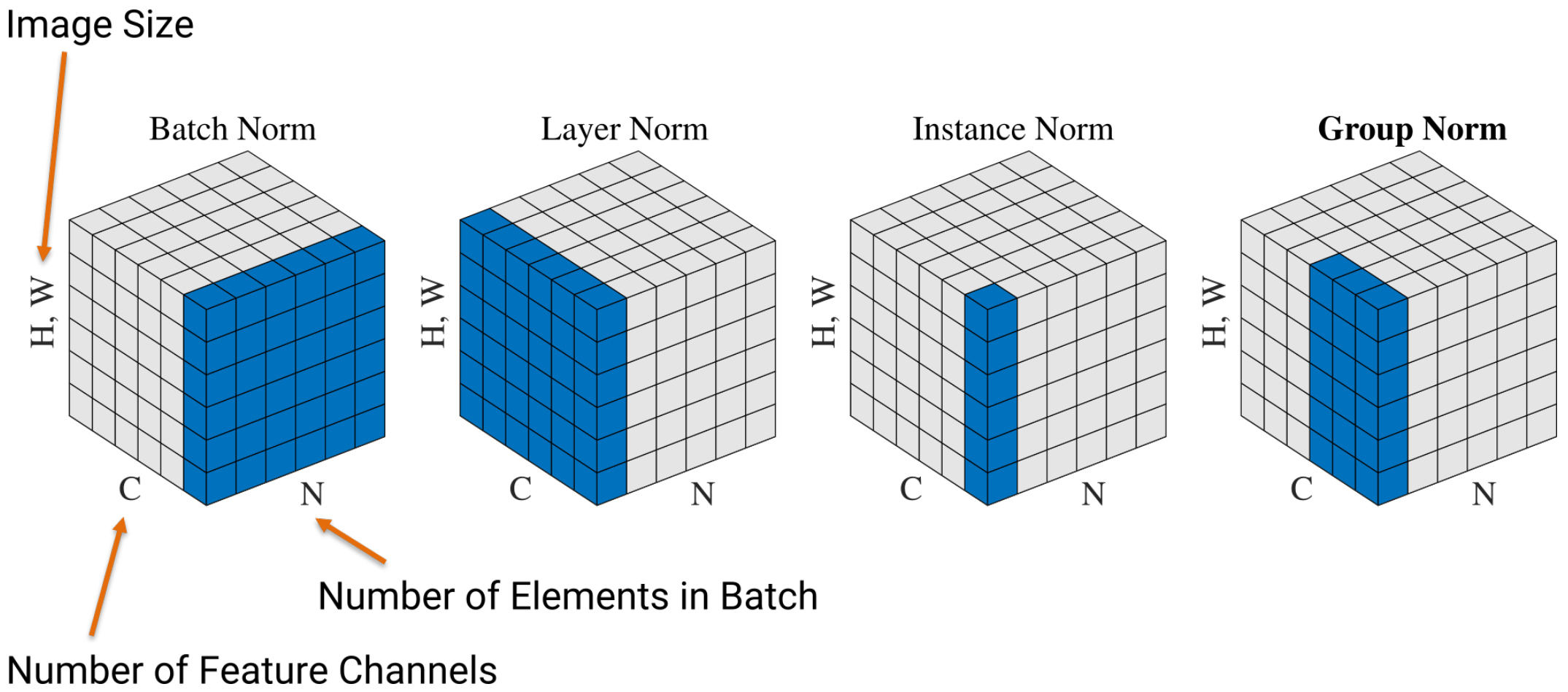
- Normalize each neuron (=channel) c by mean and variance over batch
- At test time: average during training
- Add learnable scale/shift parameter



- Insert before *or* after activation

Ioffe and Szegedy: Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. ICML, 2015.

Batch Normalization



Wu, He: Group Normalization. ECCV, 2018.

Training Schedules

Pretraining: (common practice)

- Pretrain backbone (all layers except last few layers) on another task for which a large dataset with labels is available
- Finetune last layers/full architecture on target task/dataset

Self Supervision:

- Pretrain backbone on a task for which supervision is generated from the data itself (e.g., denoising, inpainting, contrastive learning)

Curriculum Learning:

- Start learning on easy datasets and successively increase difficulty