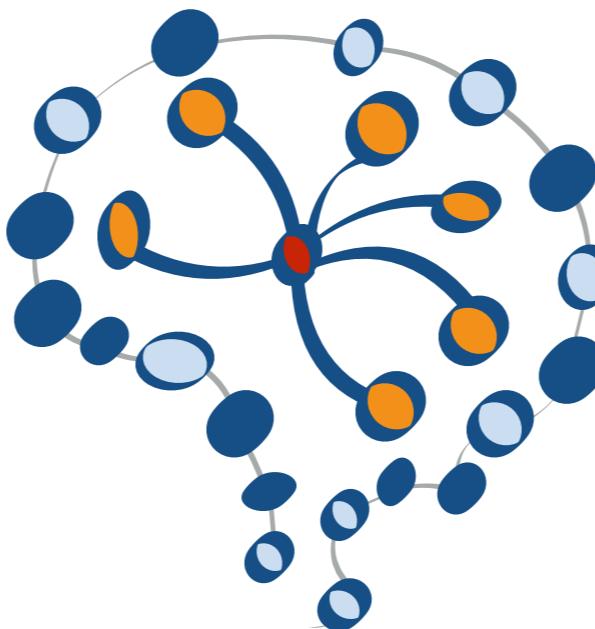


STAT 453: Introduction to Deep Learning and Generative Models

Sebastian Raschka

<http://stat.wisc.edu/~sraschka/teaching>



Lecture 11

Feature Normalization and Weight Initialization

"Tricks" for Improving *Deep* Neural Network Training

Today:

1. Feature/Input Normalization (BatchNorm)
2. Weight Initialization (Xavier Glorot, Kaiming He)

Next Lecture:

3. Optimization Algorithms (RMSProp, Adagrad, ADAM)



Lecture Overview

1. Input normalization
2. Batch normalization
3. BatchNorm in PyTorch
4. Why does BatchNorm work?
5. Weight initialization -- why do we care?
6. Xavier & He Initialization
7. Weight initialization schemes in PyTorch

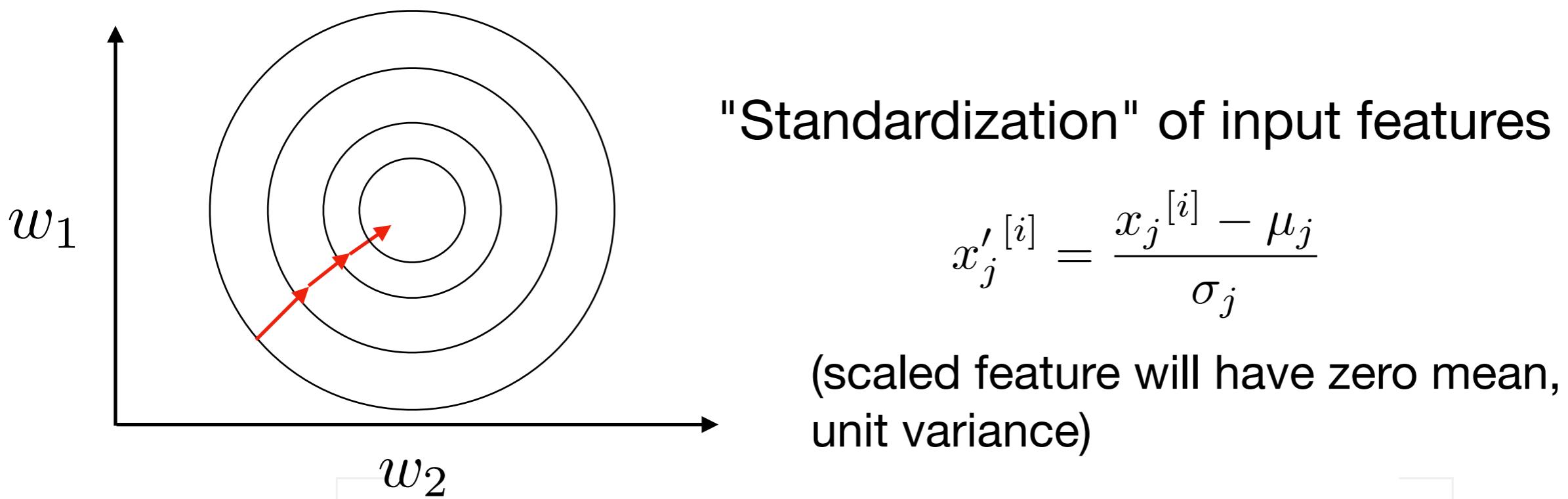
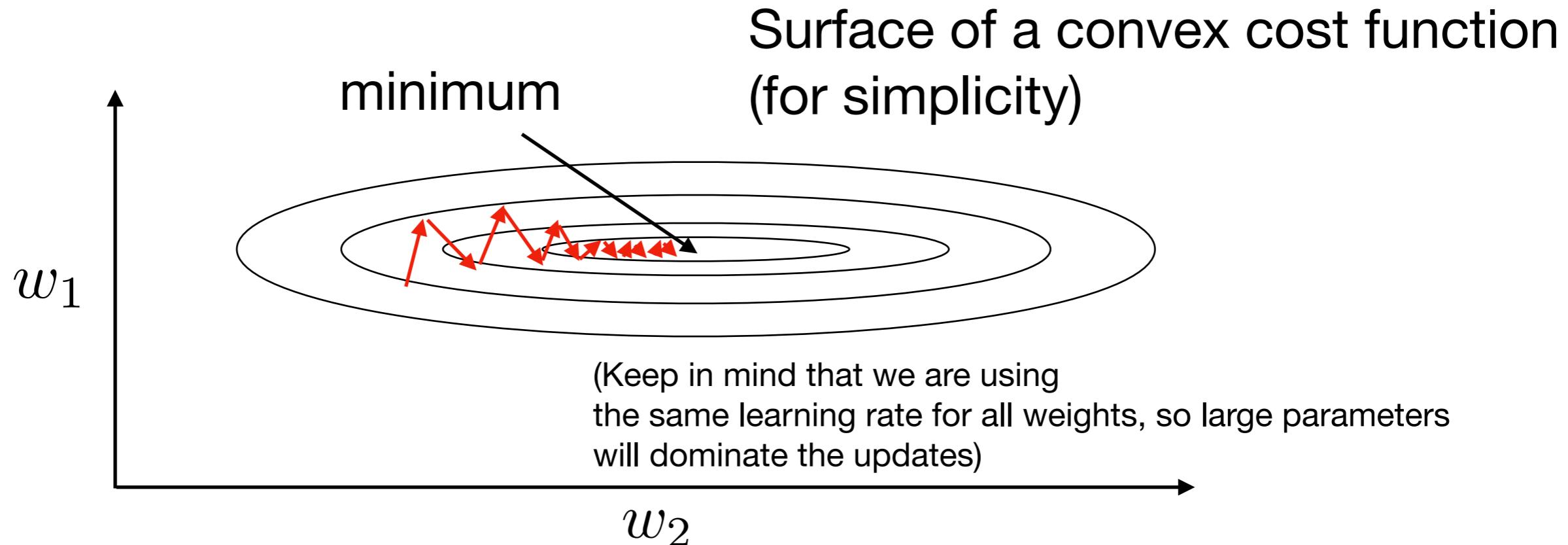


Normalizing inputs to improve gradient descent

1. **Input normalization**
2. Batch normalization
3. BatchNorm in PyTorch
4. Why does BatchNorm work?
5. Weight initialization -- why do we care?
6. Xavier & He Initialization
7. Weight initialization schemes in PyTorch



Recap: Why We Normalize Inputs for Gradient Descent



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

Extending input normalization to the hidden layers

1. Input normalization
2. **Batch normalization**
3. BatchNorm in PyTorch
4. Why does BatchNorm work?
5. Weight initialization -- why do we care?
6. Xavier & He Initialization
7. Weight initialization schemes in PyTorch



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).

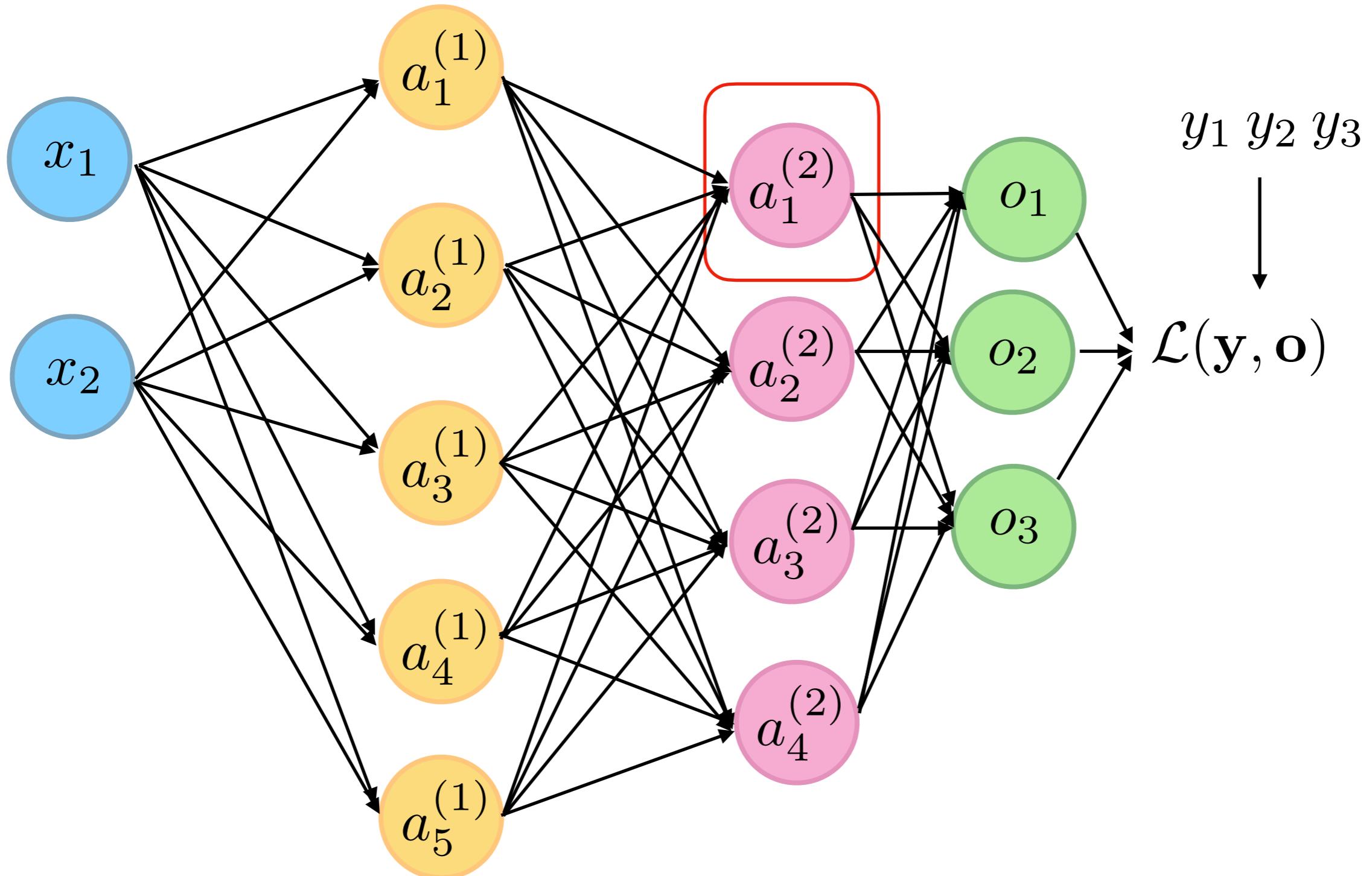
<http://proceedings.mlr.press/v37/ioffe15.html>



Batch Normalization ("BatchNorm")

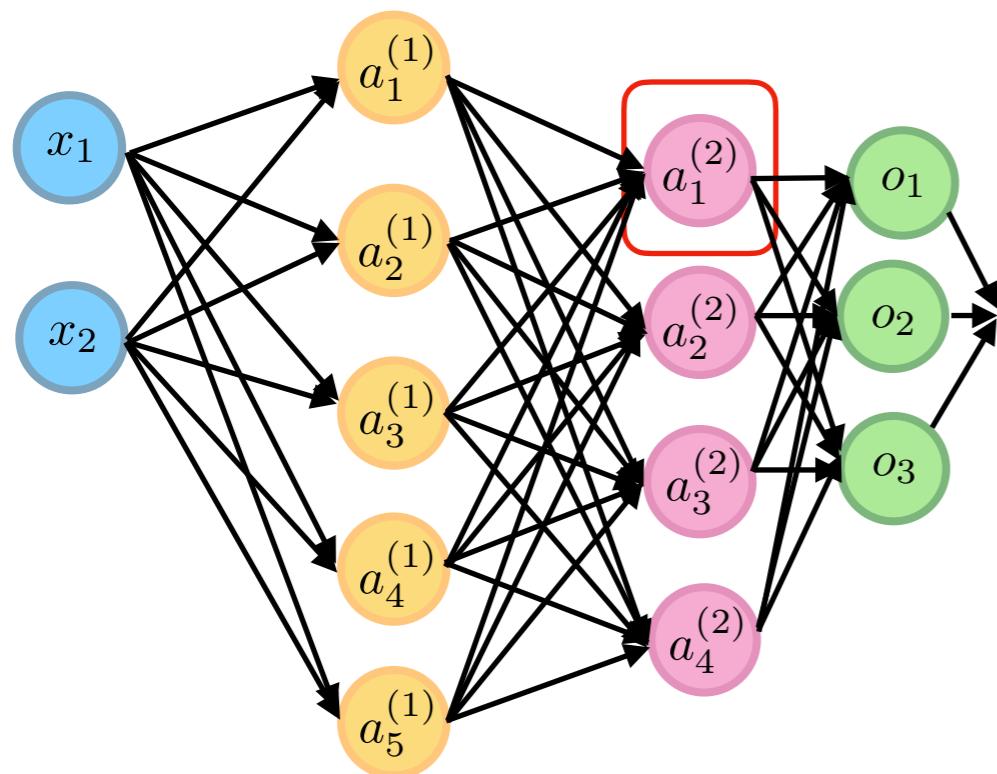
- Normalizes hidden layer inputs
- Helps with exploding/vanishing gradient problems
- Can increase training stability and convergence rate
- Can be understood as additional (normalization) layers (with additional parameters)

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 $i \in \{1, \dots, n\}$



In the next slides, let's omit the layer index, as it may be distracting...

BatchNorm Step 1: Normalize Net Inputs

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

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

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



BatchNorm Step 1: Normalize Net Inputs

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

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

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

In practice:

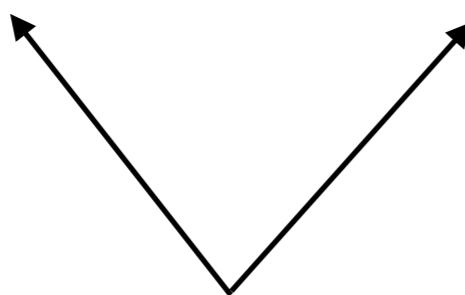
$$z'_j^{[i]} = \frac{z_j^{[i]} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}$$

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

BatchNorm Step 2: Pre-Activation Scaling

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

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



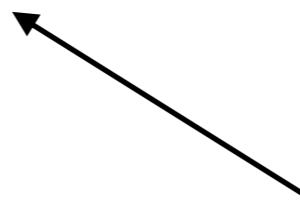
These are learnable parameters



BatchNorm Step 2: Pre-Activation Scaling

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

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



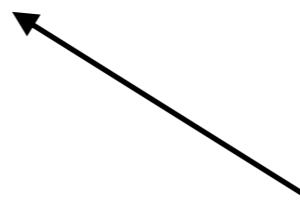
Controls the mean

Controls the spread or scale

BatchNorm Step 2: Pre-Activation Scaling

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

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



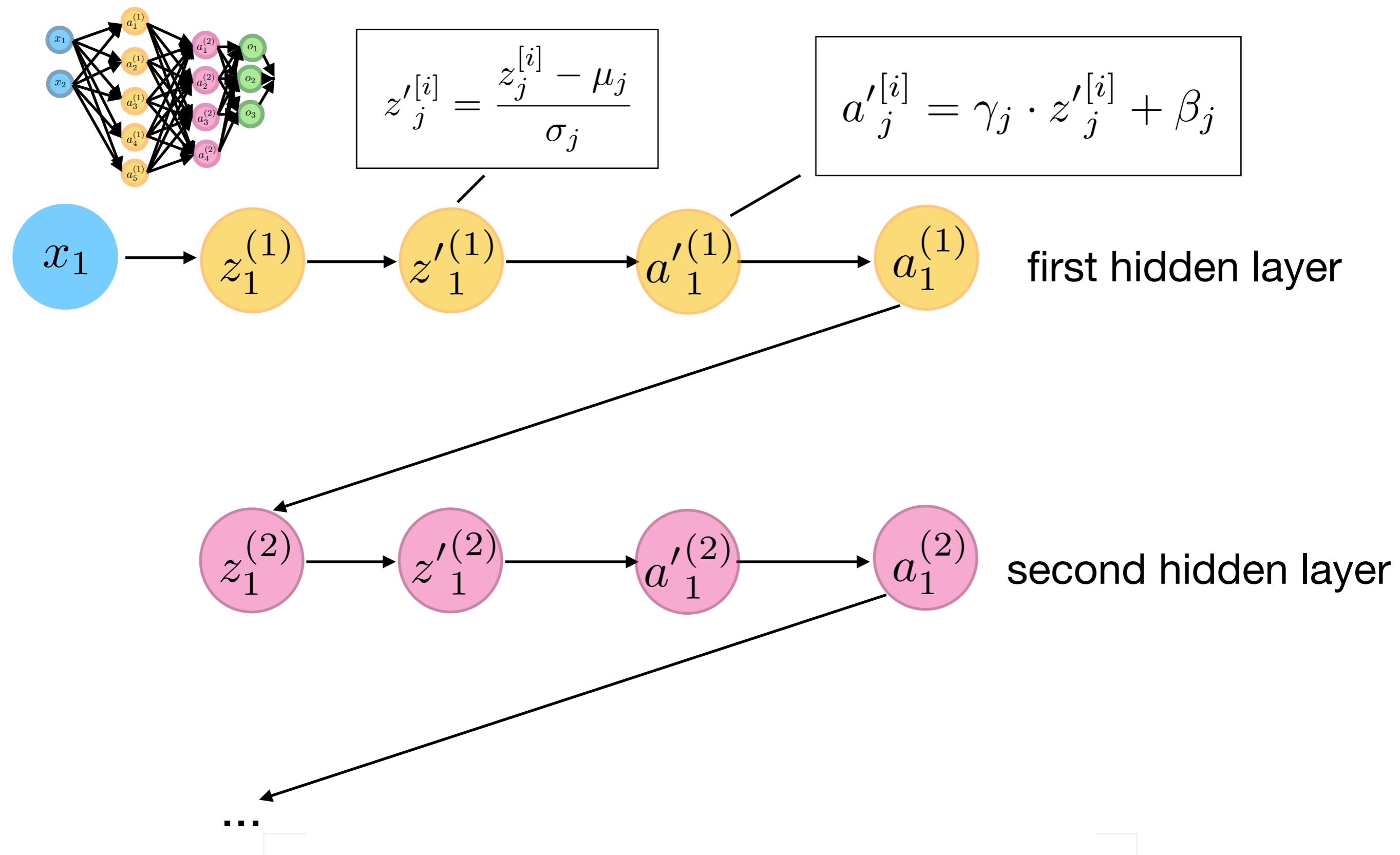
Controls the mean

Controls the spread or scale

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

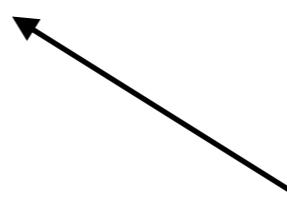


BatchNorm Step 1 & 2 Summarized



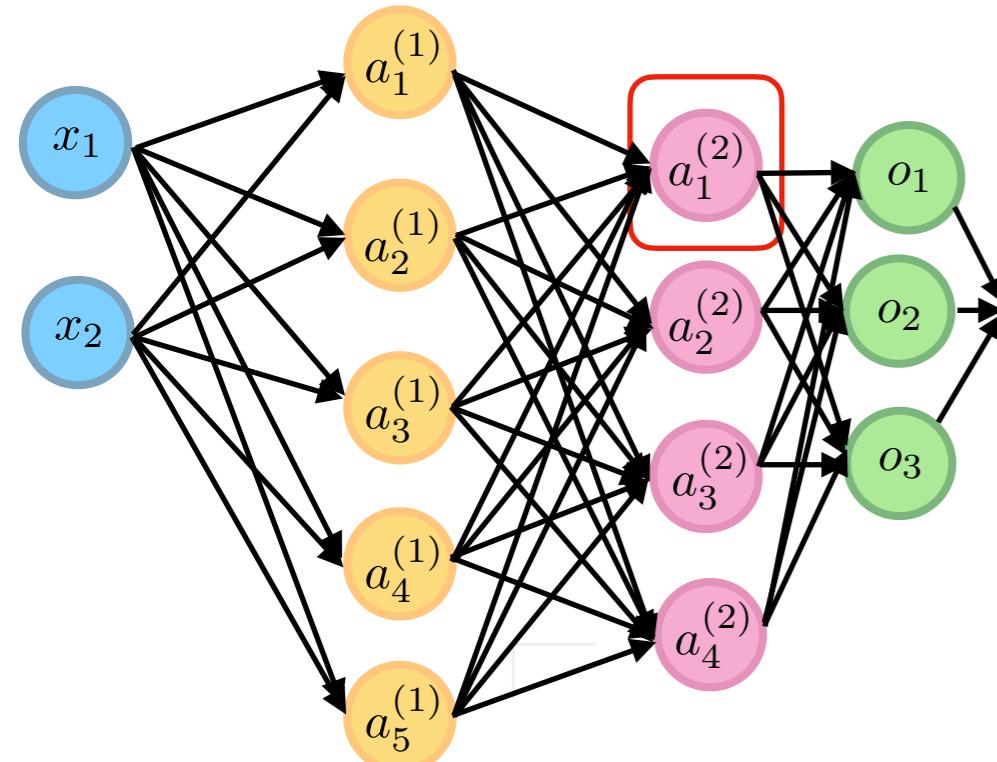
BatchNorm -- Additional Things to Consider

$$a_j'[i] = \gamma_j \cdot z_j'[i] + \beta_j$$



This parameter makes the bias units redundant

Also, note that the batchnorm parameters are vectors with the same number of elements as the bias vector



How to use BatchNorm in Practice and During Inference

1. Input normalization
2. Batch normalization
- 3. BatchNorm in PyTorch**
4. Why does BatchNorm work?
5. Weight initialization -- why do we care?
6. Xavier & He Initialization
7. Weight initialization schemes in PyTorch



```
class MultilayerPerceptron(torch.nn.Module):

    def __init__(self, num_features, num_classes, drop_proba,
                 num_hidden_1, num_hidden_2):
        super().__init__()

        self.my_network = torch.nn.Sequential(
            # 1st hidden layer
            torch.nn.Flatten(),
            torch.nn.Linear(num_features, num_hidden_1, bias=False),
            torch.nn.BatchNorm1d(num_hidden_1),
            torch.nn.ReLU(),
            # 2nd hidden layer
            torch.nn.Linear(num_hidden_1, num_hidden_2, bias=False),
            torch.nn.BatchNorm1d(num_hidden_2),
            torch.nn.ReLU(),
            # output layer
            torch.nn.Linear(num_hidden_2, num_classes)
        )

    def forward(self, x):
        logits = self.my_network(x)
        return logits
```

<https://github.com/rasbt/stat453-deep-learning-ss21/blob/main/L11/code/batchnorm.ipynb>

before activation, no bias

```
self.my_network = torch.nn.Sequential(  
    # 1st hidden layer  
    torch.nn.Flatten(),  
    torch.nn.Linear(num_features, num_hidden_1, bias=False),  
    torch.nn.BatchNorm1d(num_hidden_1),  
    torch.nn.ReLU(),  
    # 2nd hidden layer  
    torch.nn.Linear(num_hidden_1, num_hidden_2, bias=False),  
    torch.nn.BatchNorm1d(num_hidden_2),  
    torch.nn.ReLU(),  
    # output layer  
    torch.nn.Linear(num_hidden_2, num_classes)  
)
```

before activation, with bias

```
self.my_network = torch.nn.Sequential(  
    # 1st hidden layer  
    torch.nn.Flatten(),  
    torch.nn.Linear(num_features, num_hidden_1),  
    torch.nn.BatchNorm1d(num_hidden_1),  
    torch.nn.ReLU(),  
    # 2nd hidden layer  
    torch.nn.Linear(num_hidden_1, num_hidden_2),  
    torch.nn.BatchNorm1d(num_hidden_2),  
    torch.nn.ReLU(),  
    # output layer  
    torch.nn.Linear(num_hidden_2, num_classes)  
)
```

after activation, with bias

```
self.my_network = torch.nn.Sequential(  
    # 1st hidden layer  
    torch.nn.Flatten(),  
    torch.nn.Linear(num_features, num_hidden_1, bias=True),  
    torch.nn.ReLU(),  
    torch.nn.BatchNorm1d(num_hidden_1),  
    # 2nd hidden layer  
    torch.nn.Linear(num_hidden_1, num_hidden_2, bias=True),  
    torch.nn.ReLU(),  
    torch.nn.BatchNorm1d(num_hidden_2),  
    # output layer  
    torch.nn.Linear(num_hidden_2, num_classes)  
)
```

before activation + dropout

```
self.my_network = torch.nn.Sequential(  
    # 1st hidden layer  
    torch.nn.Flatten(),  
    torch.nn.Linear(num_features, num_hidden_1, bias=False),  
    torch.nn.BatchNorm1d(num_hidden_1),  
    torch.nn.ReLU(),  
    torch.nn.Dropout(drop_proba),  
    # 2nd hidden layer  
    torch.nn.Linear(num_hidden_1, num_hidden_2, bias=False),  
    torch.nn.BatchNorm1d(num_hidden_2),  
    torch.nn.ReLU(),  
    torch.nn.Dropout(drop_proba),  
    # output layer  
    torch.nn.Linear(num_hidden_2, num_classes)  
)
```

after activation + dropout

```
self.my_network = torch.nn.Sequential(  
    # 1st hidden layer  
    torch.nn.Flatten(),  
    torch.nn.Linear(num_features, num_hidden_1, bias=True),  
    torch.nn.ReLU(),  
    torch.nn.BatchNorm1d(num_hidden_1),  
    torch.nn.Dropout(drop_proba),  
    # 2nd hidden layer  
    torch.nn.Linear(num_hidden_1, num_hidden_2, bias=True),  
    torch.nn.ReLU(),  
    torch.nn.BatchNorm1d(num_hidden_2),  
    torch.nn.Dropout(drop_proba),  
    # output layer  
    torch.nn.Linear(num_hidden_2, num_classes)  
)
```

```

def train_model(model, num_epochs, train_loader,
               valid_loader, test_loader, optimizer, device):

    start_time = time.time()
    minibatch_loss_list, train_acc_list, valid_acc_list = [], [], []
    for epoch in range(num_epochs):

        model.train()
        for batch_idx, (features, targets) in enumerate(train_loader):

            features = features.to(device)
            targets = targets.to(device)

            # ## FORWARD AND BACK PROP
            logits = model(features)
            loss = torch.nn.functional.cross_entropy(logits, targets)
            optimizer.zero_grad()

            loss.backward()

            # ## UPDATE MODEL PARAMETERS
            optimizer.step()

            # ## LOGGING
            minibatch_loss_list.append(loss.item())
            if not batch_idx % 50:
                print(f'Epoch: {epoch+1:03d}/{num_epochs:03d} '
                      f'| Batch {batch_idx:04d}/{len(train_loader):04d} '
                      f'| Loss: {loss:.4f}')

        model.eval()
        with torch.no_grad(): # save memory during inference
            train_acc = compute_accuracy(model, train_loader, device=device)

```

**don't forget `model.train()`
 and `model.eval()`
 in training and test loops**

BatchNorm During Prediction ("Inference")

- Use exponentially weighted average (moving average) of mean and variance

```
running_mean = momentum * running_mean  
              + (1 - momentum) * sample_mean
```

(where momentum is typically ~0.1; and same for variance)

- Alternatively, can also use global training set mean and variance

BatchNorm: Some theories and practical advice

1. Input normalization
2. Batch normalization
3. BatchNorm in PyTorch
- 4. Why does BatchNorm work?**
5. Weight initialization -- why do we care?
6. Xavier & He Initialization
7. Weight initialization schemes in PyTorch



BatchNorm and Internal Covariate Shift

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

<http://proceedings.mlr.press/v37/ioffe15.html>

Internal Covariate Shift is jargon for saying that the layer input distribution changes ("feature shift" in hidden layers)

But there is no guarantee or strong evidence that BatchNorm helps with covariate shift

Maybe BatchNorm just provides additional parameters that will help layers to learn a little bit more independently



How Does Batch Normalization Help Optimization?

Shibani Santurkar*
MIT
shibani@mit.edu

Dimitris Tsipras*
MIT
tsipras@mit.edu

Andrew Ilyas*
MIT
ailyas@mit.edu

Aleksander Mądry
MIT
madry@mit.edu

Abstract

Batch Normalization (BatchNorm) is a widely adopted technique that enables faster and more stable training of deep neural networks (DNNs). Despite its pervasiveness, the exact reasons for BatchNorm’s effectiveness are still poorly understood. The popular belief is that this effectiveness stems from controlling the change of the layers’ input distributions during training to reduce the so-called “internal covariate shift”. In this work, we demonstrate that such distributional stability of layer inputs has little to do with the success of BatchNorm. Instead, we uncover a more fundamental impact of BatchNorm on the training process: it makes the optimization landscape significantly smoother. This smoothness induces a more predictive and stable behavior of the gradients, allowing for faster training.

BatchNorm Enables Faster Convergence By Allowing Larger Learning Rates

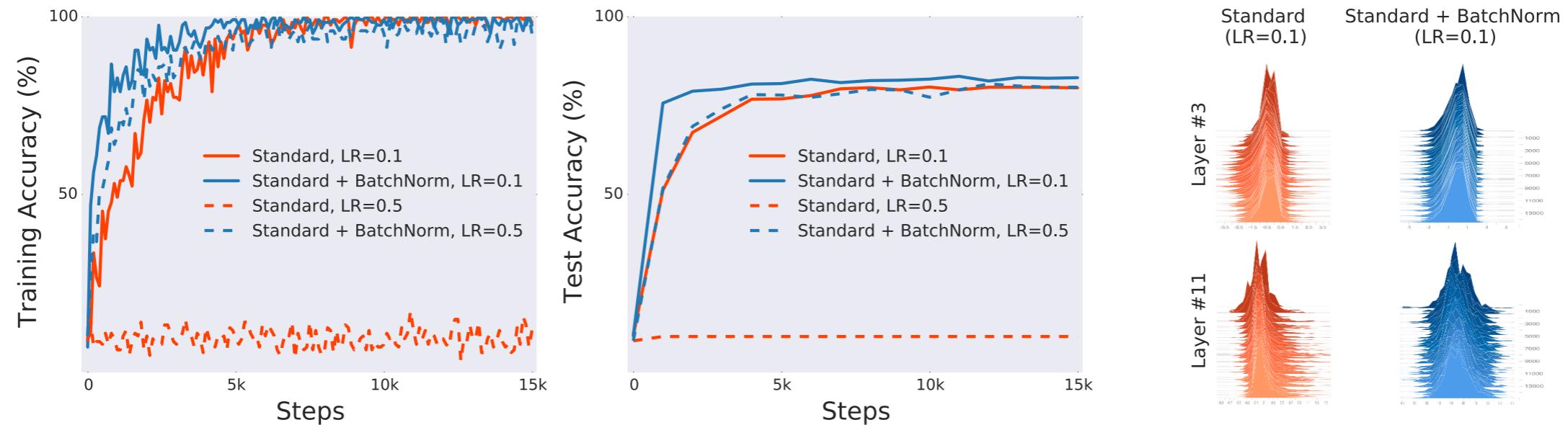


Figure 1: Comparison of (a) training (optimization) and (b) test (generalization) performance of a standard VGG network trained on CIFAR-10 with and without BatchNorm (details in Appendix A). There is a consistent gain in training speed in models with BatchNorm layers. (c) Even though the gap between the performance of the BatchNorm and non-BatchNorm networks is clear, the difference in the evolution of layer input distributions seems to be much less pronounced. (Here, we sampled activations of a given layer and visualized their distribution over training steps.)

Santurkar, S., Tsipras, D., Ilyas, A., & Madry, A. (2018). How does batch normalization help optimization?. In *Advances in Neural Information Processing Systems* (pp. 2488-2498).

<https://arxiv.org/abs/1805.11604>

Good Performance of BatchNorm Seems Unrelated to Covariate Shift Prevention

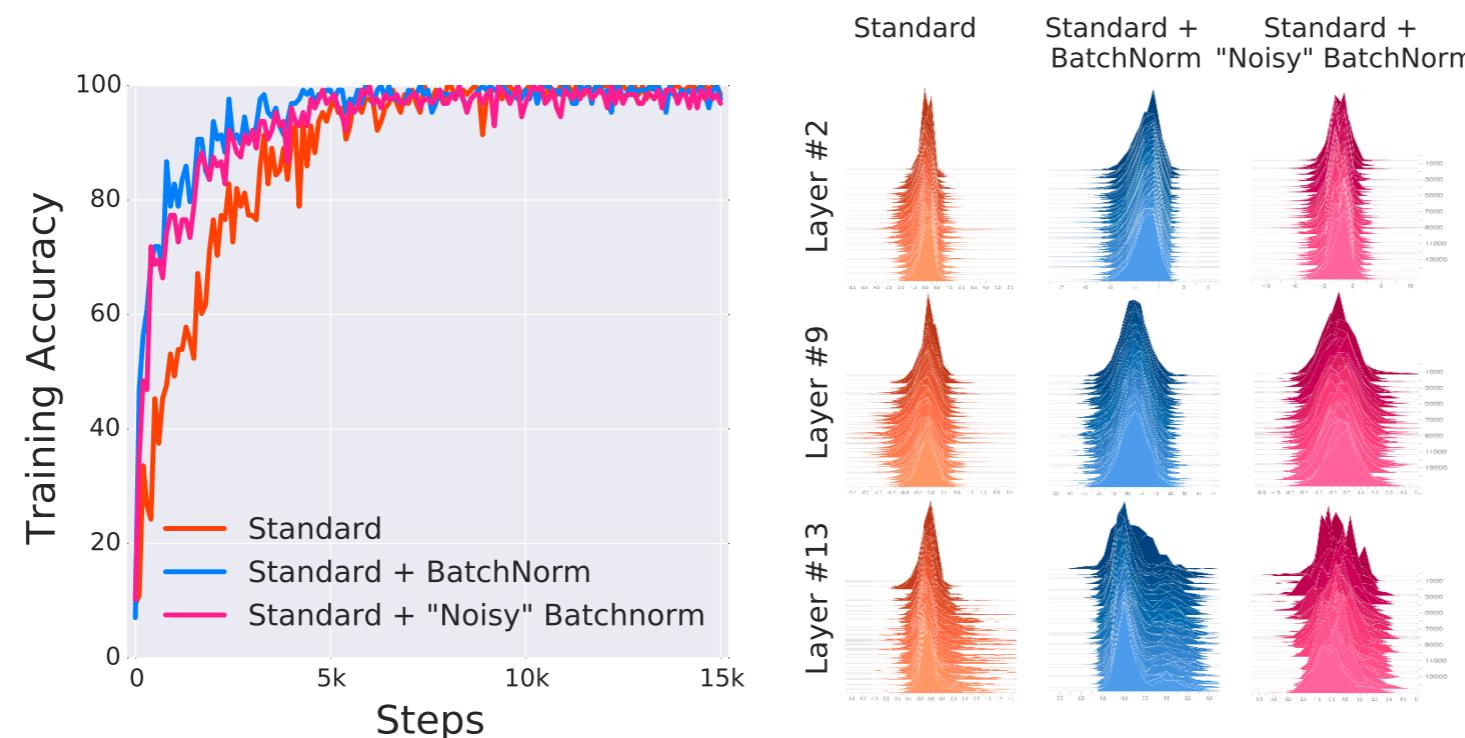


Figure 2: Connections between distributional stability and BatchNorm performance: We compare VGG networks trained without BatchNorm (Standard), with BatchNorm (Standard + BatchNorm) and with explicit “covariate shift” added to BatchNorm layers (Standard + “Noisy” BatchNorm). In the later case, we induce distributional instability by adding *time-varying, non-zero* mean and *non-unit* variance noise independently to each batch normalized activation. The “noisy” BatchNorm model nearly matches the performance of standard BatchNorm model, despite complete distributional instability. We sampled activations of a given layer and visualized their distributions (also cf. Figure 7).

Santurkar, S., Tsipras, D., Ilyas, A., & Madry, A. (2018). How does batch normalization help optimization?. In *Advances in Neural Information Processing Systems* (pp. 2488-2498).

How Does BatchNorm Work? Why Does BatchNorm Help?

2015:

Reduces covariate shift.

Ioffe, S., & Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*.

2018:

Networks with BatchNorm train well with or without ICS.

Hypothesis is that BatchNorm makes the optimization landscape smoother.

Santurkar, S., Tsipras, D., Ilyas, A., & Madry, A. (2018). How does batch normalization help optimization? In *Advances in Neural Information Processing Systems* (pp. 2483-2493).

2018:

"Batch normalization implicitly discourages single direction reliance" (here, "single direction reliance" means that an input influences only a single unit or linear combination of single units)

Morcos, A. S., Barrett, D. G., Rabinowitz, N. C., & Botvinick, M. (2018). On the importance of single directions for generalization. *arXiv preprint arXiv:1803.06959*.

How Does BatchNorm Work? Why Does BatchNorm Help?

2018:

BatchNorm acts as an implicit regularizer and improves generalization accuracy

Luo, P., Wang, X., Shao, W., & Peng, Z. (2018). Towards understanding regularization in batch normalization. *arXiv preprint arXiv:1809.00846*.

2019:

BatchNorm causes exploding gradients, requiring careful tuning when training deep neural nets without skip connections (more about skip connections soon)

Yang, G., Pennington, J., Rao, V., Sohl-Dickstein, J., & Schoenholz, S. S. (2019). A mean field theory of batch normalization. *arXiv preprint arXiv:1902.08129*.



BatchNorm Variants

Pre-Activation

"Original" version
as discussed in
previous slides

compute net inputs



BatchNorm



apply activation function



compute next-layer net
inputs

Post-Activation

May make more sense,
but less common

compute net inputs



apply activation function



BatchNorm



compute next-layer net
inputs

Some Benchmarks

<https://github.com/ducha-aiki/caffenet-benchmark/blob/master/batchnorm.md#bn----before-or-after-relu>

BN -- before or after ReLU?

| Name | Accuracy | LogLoss | Comments |
|---------------------------|--------------|-------------|-------------|
| Before | 0.474 | 2.35 | As in paper |
| Before + scale&bias layer | 0.478 | 2.33 | As in paper |
| After | 0.499 | 2.21 | |
| After + scale&bias layer | 0.493 | 2.24 | |

Practical Consideration

BatchNorm become more stable with larger minibatch sizes

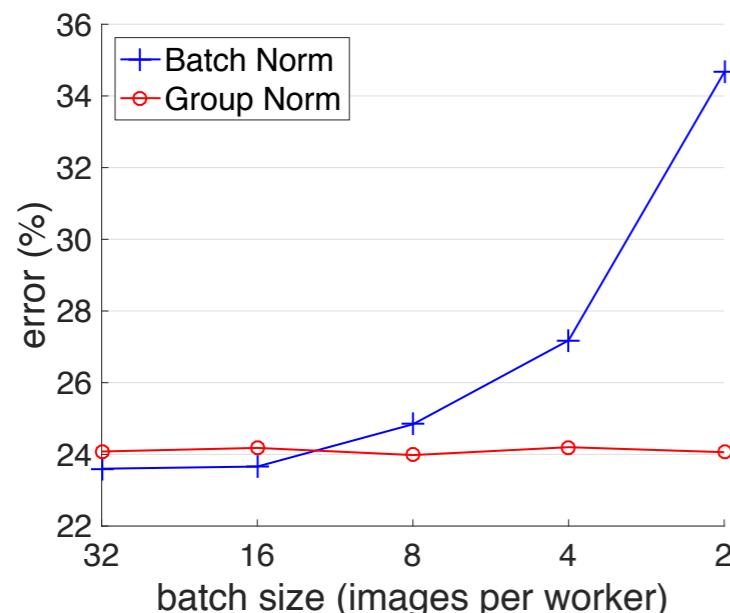


Figure 1. ImageNet classification error *vs.* batch sizes. The model is ResNet-50 trained in the ImageNet training set using 8 workers (GPUs) and evaluated in the validation set. BN's error increases rapidly when reducing the batch size. GN's computation is independent of batch sizes, and its error rate is stable despite the batch size changes. GN has substantially lower error (by 10%) than BN with a batch size of 2.

Wu, Y., & He, K. (2018). Group normalization. In *Proceedings of the European Conference on Computer Vision (ECCV)* (pp. 3-19).

Further Reading

Conditional batch norm

De Vries, H., Strub, F., Mary, J., Larochelle, H., Pietquin, O., & Courville, A. (2017). Modulating early visual processing by language. *arXiv preprint arXiv:1707.00683*.

<https://github.com/pytorch/pytorch/issues/8985>

<https://arxiv.org/abs/1707.00683>

Shallue, C. J., Lee, J., Antognini, J., Sohl-Dickstein, J., Frostig, R., & Dahl, G. E. (2018). **Measuring the effects of data parallelism on neural network training.** *arXiv preprint arXiv:1811.03600*.

-> "We find no evidence that larger batch sizes degrade out-of-sample performance"

<https://arxiv.org/abs/1811.03600>

Cai, Z., Ravichandran, A., Maji, S., Fowlkes, C., Tu, Z., & Soatto, S. (2021). **Exponential Moving Average Normalization for Self-supervised and Semi-supervised Learning.** *arXiv preprint arXiv:2101.08482*.

-> "We present a plug-in replacement for batch normalization (BN) called exponential moving average normalization (EMAN), which improves the performance of existing student-teacher based self- and semi-supervised learning techniques. ..."

<https://arxiv.org/abs/2101.08482>

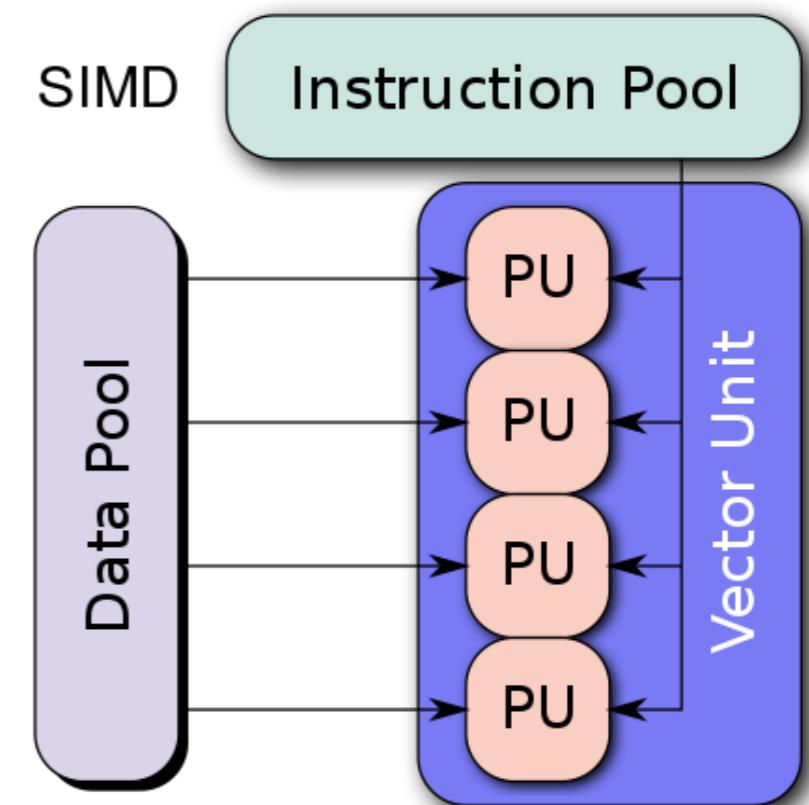
Brock, A., De, S., Smith, S. L., & Simonyan, K. (2021). **High-Performance Large-Scale Image Recognition Without Normalization.** *arXiv preprint arXiv:2102.06171*.

"Although recent work has succeeded in training deep ResNets without normalization layers, these models do not match the test accuracies of the best batch-normalized networks, and are often unstable for large learning rates or strong data augmentations. In this work, we develop an adaptive gradient clipping technique which overcomes these instabilities, and design a significantly improved class of Normalizer-Free ResNets."

<https://arxiv.org/abs/2102.06171>

Why Minibatch Sizes as Powers of 2?

- Related to SIMD - Single Instruction Multiple Data - paradigm used by CPUs/GPUs
- Comes from mapping the computations (e.g., dot products) to physical processing cores on the GPU, where the number of processing cores is usually a power of 2
- E.g., if we have 32 columns in a matrix, we can map 2 dot products to each processing core if we have 16 processing cores (GPUs usually have many, many more processing cores)



Source: <https://upload.wikimedia.org/wikipedia/commons/thumb/c/ce/SIMD2.svg/440px-SIMD2.svg.png>

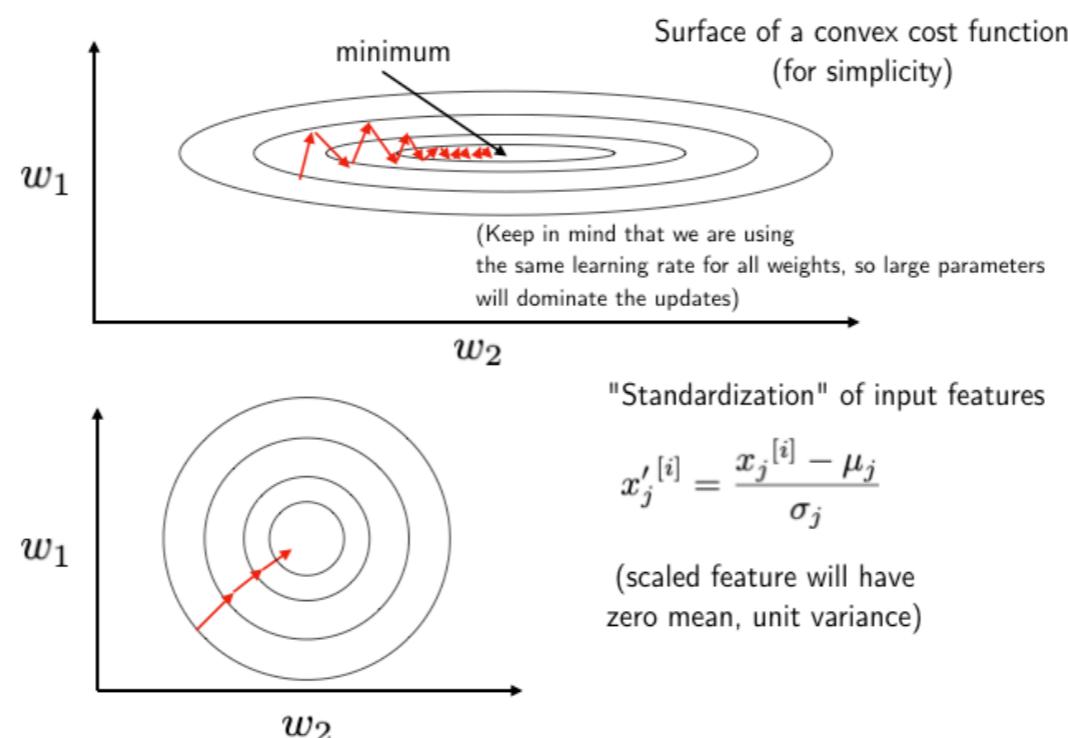


Besides input normalization, weight initialization matters, too

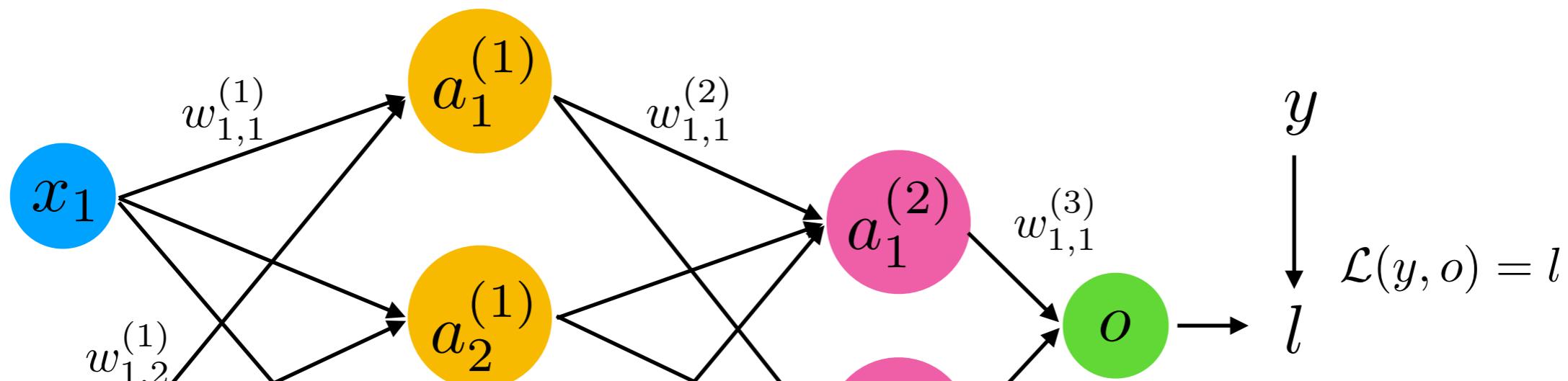
1. Input normalization
2. Batch normalization
3. BatchNorm in PyTorch
4. Why does BatchNorm work?
- 5. Weight initialization -- why do we care?**
6. Xavier & He Initialization
7. Weight initialization schemes in PyTorch

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



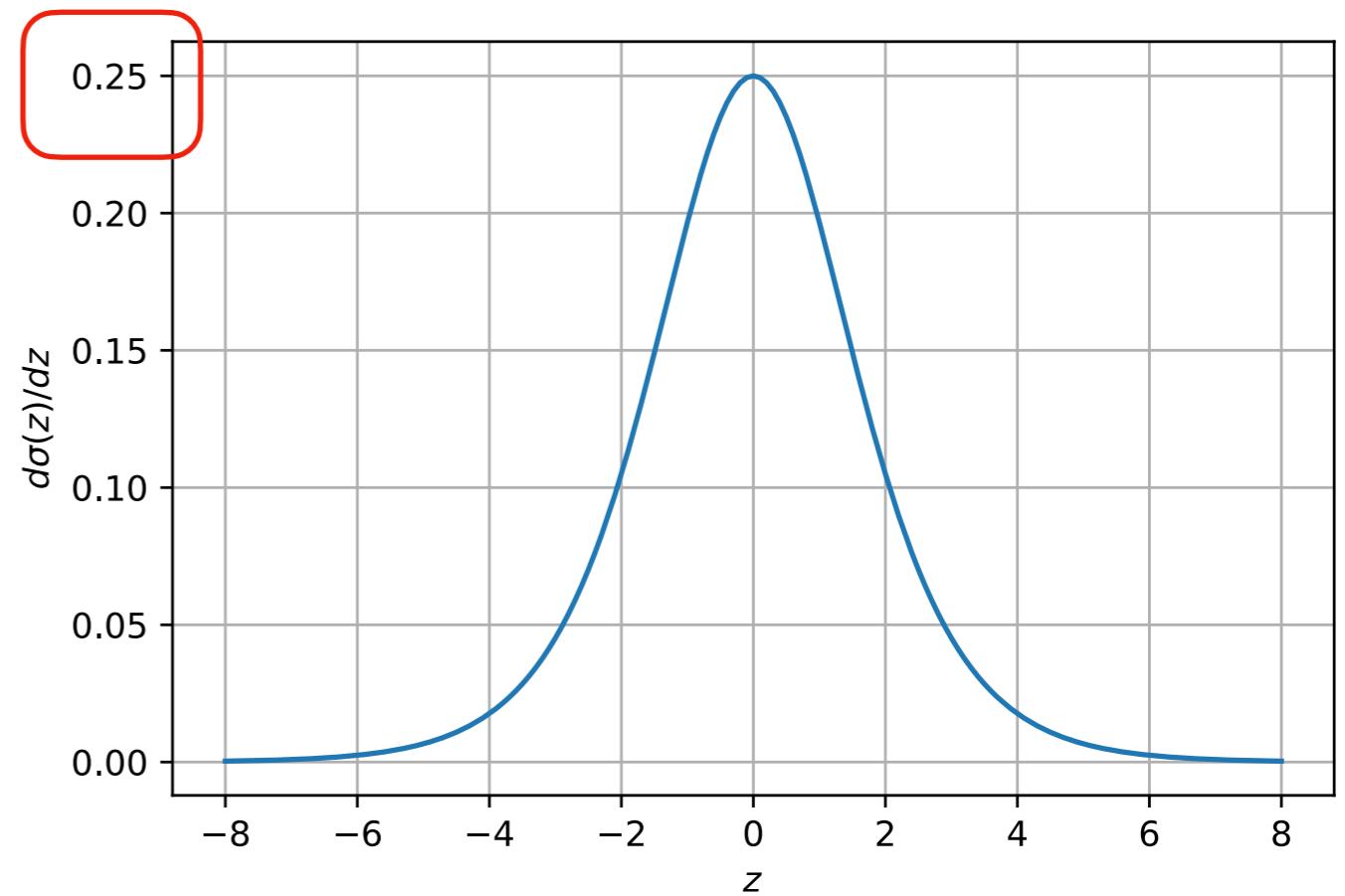
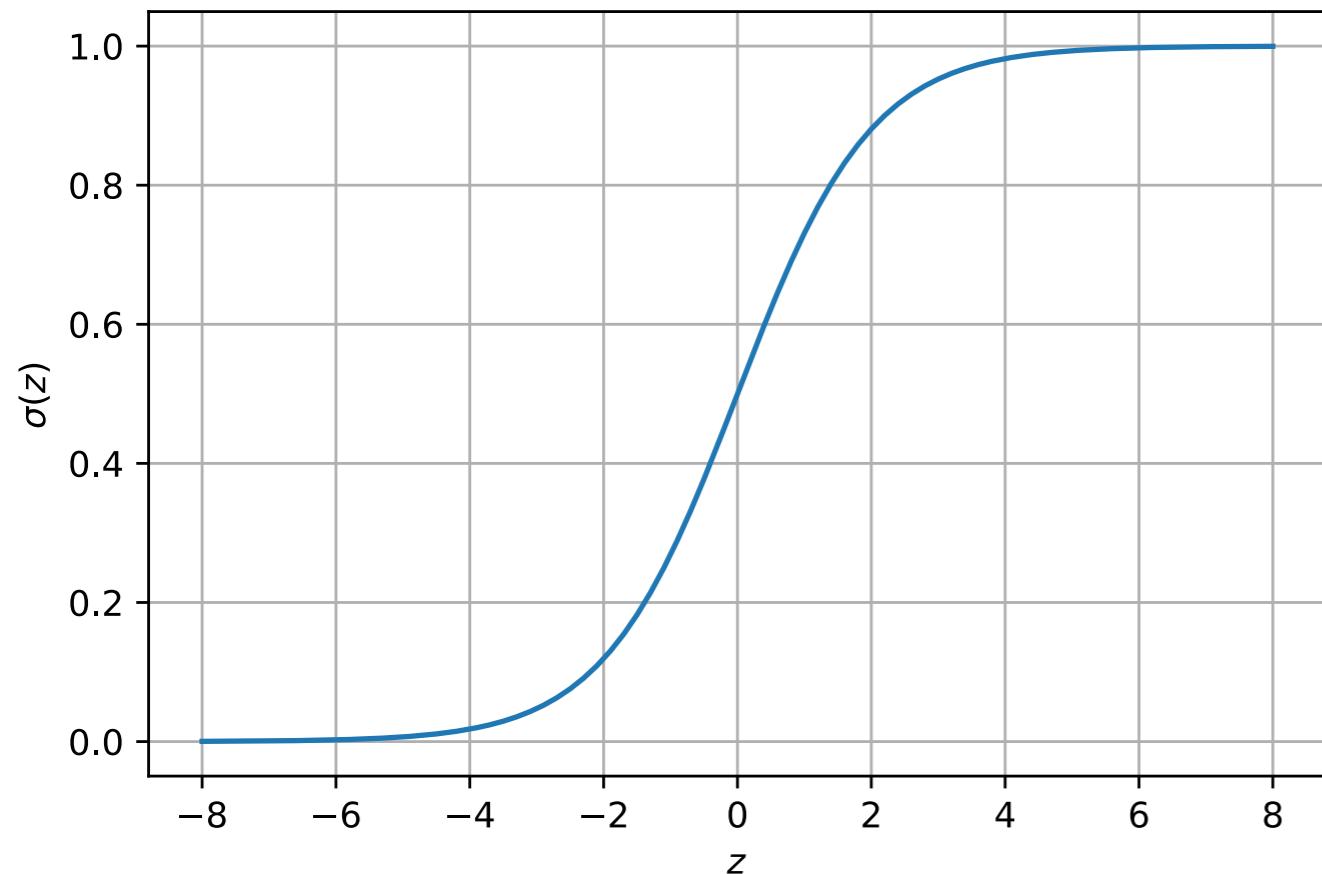
Now, imagine, we have many layers and sigmoid activations ...

$$\begin{aligned}\frac{\partial l}{\partial w_{1,1}^{(1)}} &= \frac{\partial l}{\partial o} \cdot \frac{\partial o}{\partial a_1^{(2)}} \cdot \boxed{\frac{\partial a_1^{(2)}}{\partial a_1^{(1)}} \cdot \frac{\partial a_1^{(1)}}{\partial w_{1,1}^{(1)}}} \\ &\quad + \frac{\partial l}{\partial o} \cdot \frac{\partial o}{\partial a_2^{(2)}} \cdot \frac{\partial a_2^{(2)}}{\partial a_1^{(1)}} \cdot \frac{\partial a_1^{(1)}}{\partial w_{1,1}^{(1)}}\end{aligned}$$

Sidenote: Vanishing/Exploding Gradient Problems

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

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



Sidenote: Vanishing/Exploding Gradient Problems

Assume, we have the largest gradient:

$$\frac{d}{dz}\sigma(0.0) = \sigma(0.0)(1 - \sigma(0.0)) = 0.25$$

Even then, for, e.g., 10 layers, we degrade the other gradients substantially!

$$0.25^{10} \approx 10^{-6}$$



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)



A quick look at two common weight initialization schemes

1. Input normalization
2. Batch normalization
3. BatchNorm in PyTorch
4. Why does BatchNorm work?
5. Weight initialization -- why do we care?
- 6. Xavier & He Initialization**
7. Weight initialization schemes in PyTorch

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.

- 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.

Method:

Scale the weights proportional to the number of inputs to the layer

In particular, scale as follows:

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

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 -- 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.

Sidenote: If you didn't initialize the bias units to all zeros, also include those in the scaling.

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

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 -- Xavier Initialization

Glorot, Xavier, 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 (variance of the sum of independent variables is the sum of the variances); square root for standard deviation

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

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 -- Xavier Initialization

Glorot, Xavier, 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 (variance of the sum of independent variables is the sum of the variances); square root for standard deviation

$$\begin{aligned}\text{Var} \left(z_j^{(l)} \right) &= \text{Var} \left(\sum_{j=1}^{m_l-1} W_{jk}^{(l)} a_k^{(l-1)} \right) \\ &= \sum_{j=1}^{m^{(l-1)}} \text{Var} \left[W_{jk}^{(l)} a_k^{(l-1)} \right] = \sum_{i=1}^{m^{(l-1)}} \text{Var} \left[W_{jk}^{(l)} \right] \text{Var} \left[a_k^{(l-1)} \right] \\ &= \sum_{j=1}^{m^{(l-1)}} \text{Var} \left[W^{(l)} \right] \text{Var} \left[a^{(l-1)} \right] = m^{(l-1)} \text{Var} \left[W^{(l)} \right] \text{Var} \left[a^{(l-1)} \right]\end{aligned}$$

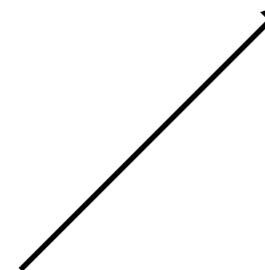
Weight Initialization -- Xavier Initialization

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

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



Weight Initialization -- Xavier Initialization

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


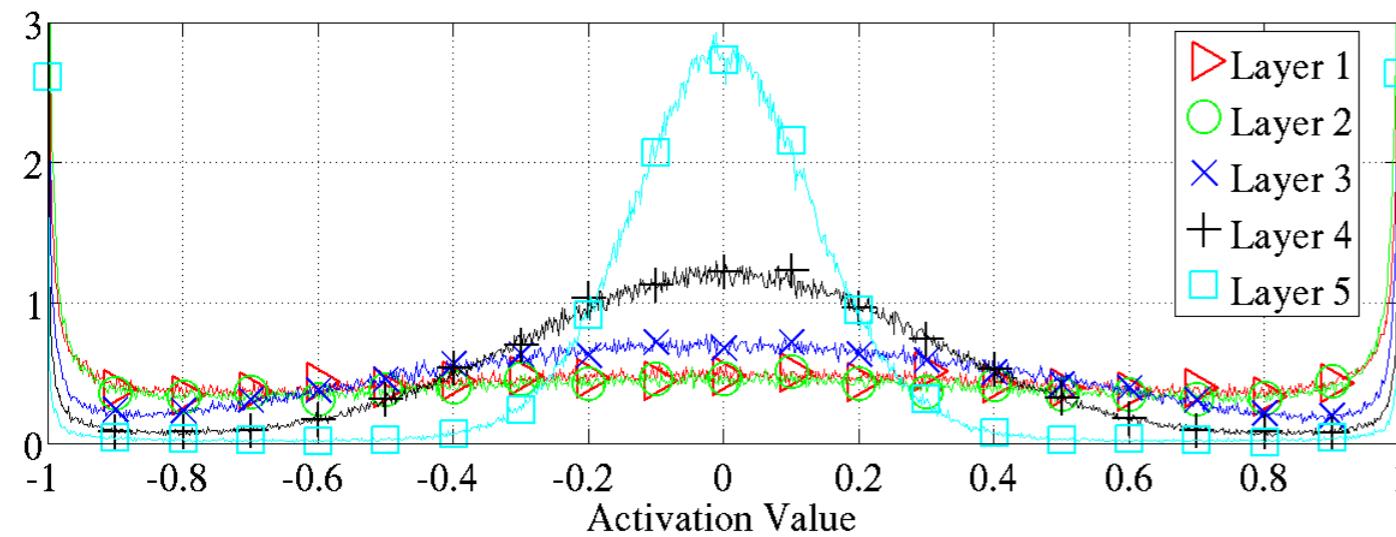
However, in practice,
some people also
use "fan in" + "fan
out" in the
denominator,
and it works fine

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

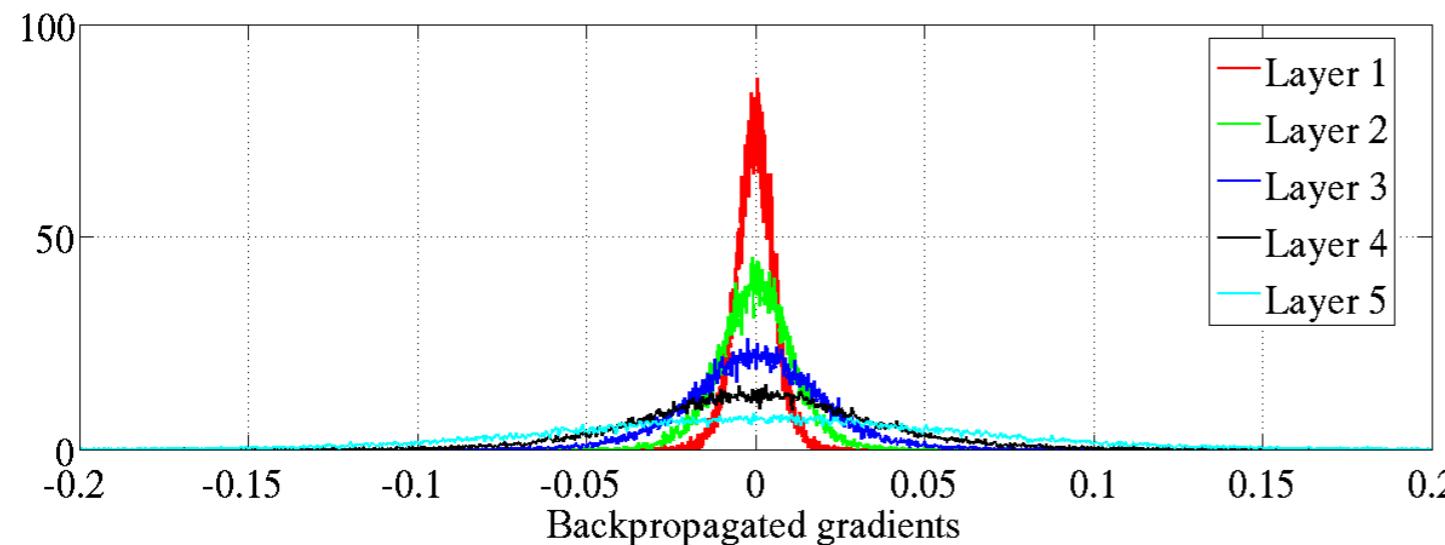


Weight Initialization -- Xavier Initialization

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



Vanishing gradient problem!



Weight Initialization -- Xavier Initialization

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

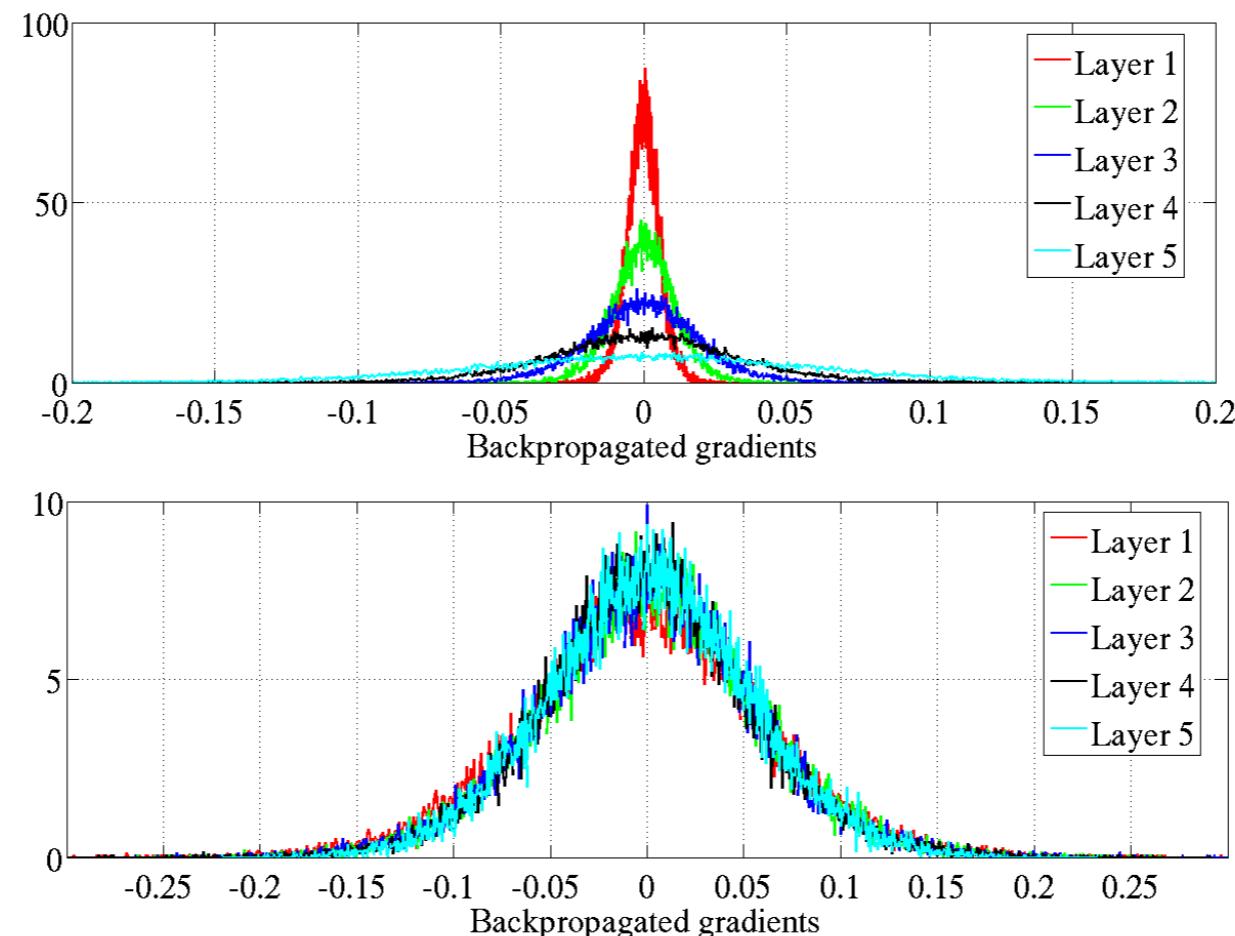


Figure 7: Back-propagated gradients normalized histograms with hyperbolic tangent activation, with standard (top) vs normalized (bottom) initialization. Top: 0-peak decreases for higher layers.

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 $2^{0.5}$

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

How does PyTorch handle weight initialization and how do we override it?

1. Input normalization
2. Batch normalization
3. BatchNorm in PyTorch
4. Why does BatchNorm work?
5. Weight initialization -- why do we care?
6. Xavier & He Initialization
7. **Weight initialization schemes in PyTorch**

PyTorch Default Weights

PyTorch (now) uses the Kaiming He scheme by default

```
75      def __init__(self, in_features: int, out_features: int, bias: bool = True) -> None:
76          super(Linear, self).__init__()
77          self.in_features = in_features
78          self.out_features = out_features
79          self.weight = Parameter(torch.Tensor(out_features, in_features))
80          if bias:
81              self.bias = Parameter(torch.Tensor(out_features))
82          else:
83              self.register_parameter('bias', None)
84          self.reset_parameters()
85
86      def reset_parameters(self) -> None:
87          init.kaiming_uniform_(self.weight, a=math.sqrt(5))
88          if self.bias is not None:
89              fan_in, _ = init._calculate_fan_in_and_fan_out(self.weight)
90              bound = 1 / math.sqrt(fan_in)
91              init.uniform_(self.bias, -bound, bound)
92
```

<https://github.com/pytorch/pytorch/blob/master/torch/nn/modules/linear.py#L86>

```
class MultilayerPerceptron(torch.nn.Module):

    def __init__(self, num_features, num_classes, drop_proba,
                 num_hidden_1, num_hidden_2):
        super().__init__()

        self.my_network = torch.nn.Sequential(
            # 1st hidden layer
            torch.nn.Flatten(),
            torch.nn.Linear(num_features, num_hidden_1),
            torch.nn.ReLU(),
            # 2nd hidden layer
            torch.nn.Linear(num_hidden_1, num_hidden_2),
            torch.nn.ReLU(),
            # output layer
            torch.nn.Linear(num_hidden_2, num_classes)
        )

        for m in self.modules():
            if isinstance(m, torch.nn.Linear):
                torch.nn.init.kaiming_uniform_(m.weight, mode='fan_in', nonlinearity='relu')
                if m.bias is not None:
                    m.bias.detach().zero_()

    def forward(self, x):
        logits = self.my_network(x)
        return logits
```

```
class MultilayerPerceptron(torch.nn.Module):

    def __init__(self, num_features, num_classes, drop_proba,
                 num_hidden_1, num_hidden_2):
        super().__init__()

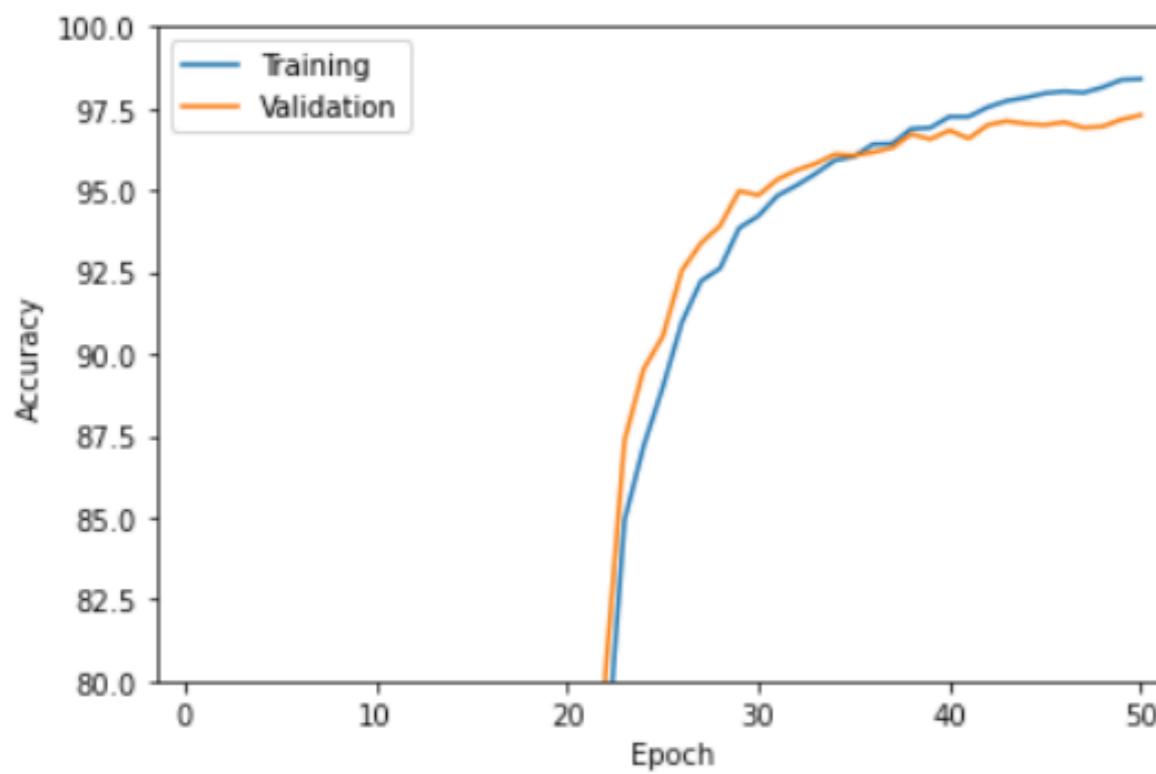
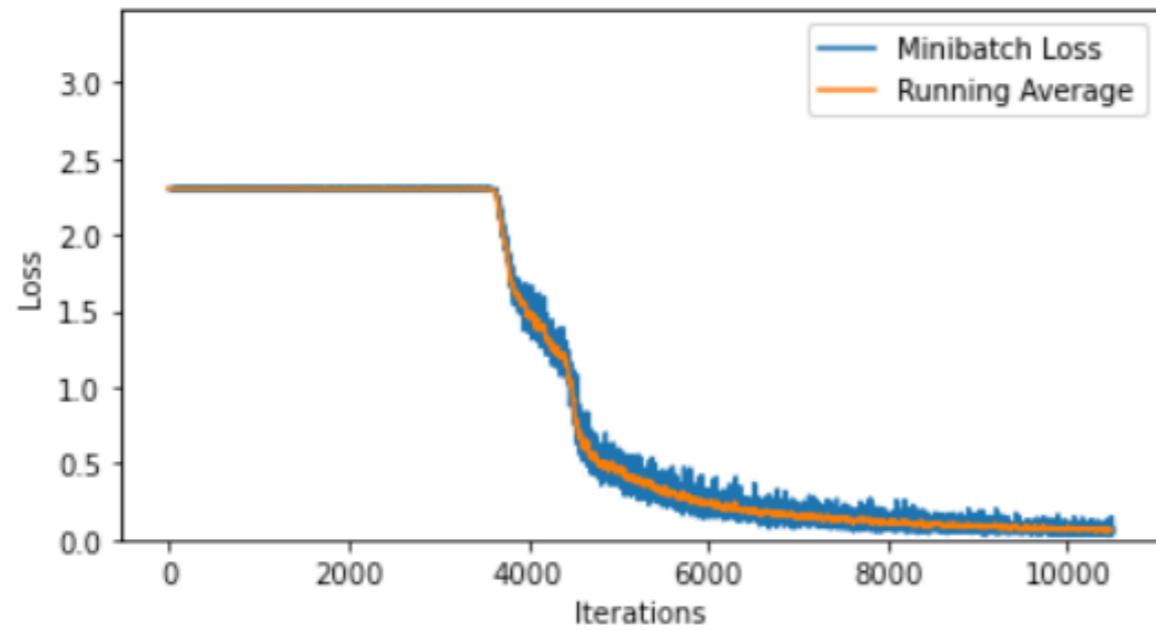
        self.my_network = torch.nn.Sequential(
            # 1st hidden layer
            torch.nn.Flatten(),
            torch.nn.Linear(num_features, num_hidden_1),
            torch.nn.ReLU(),
            # 2nd hidden layer
            torch.nn.Linear(num_hidden_1, num_hidden_2),
            torch.nn.ReLU(),
            # output layer
            torch.nn.Linear(num_hidden_2, num_classes)
        )

        for m in self.modules():
            if isinstance(m, torch.nn.Linear):
                m.weight.detach().normal_(0, 0.001)
                if m.bias is not None:
                    m.bias.detach().zero_()

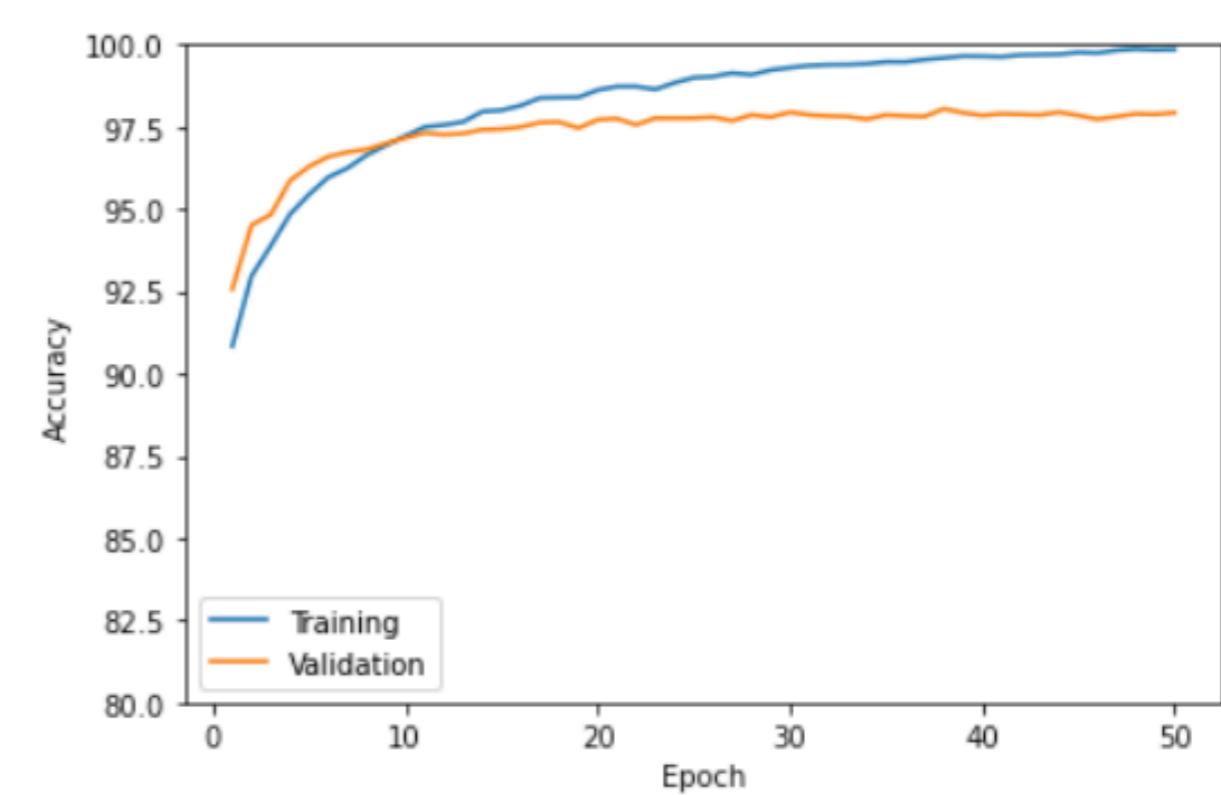
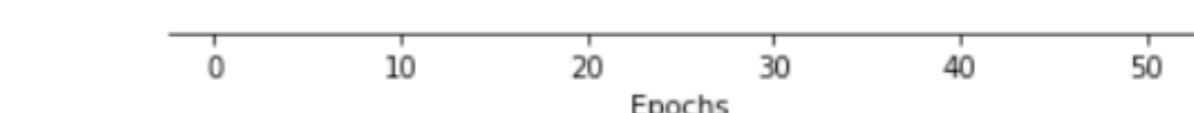
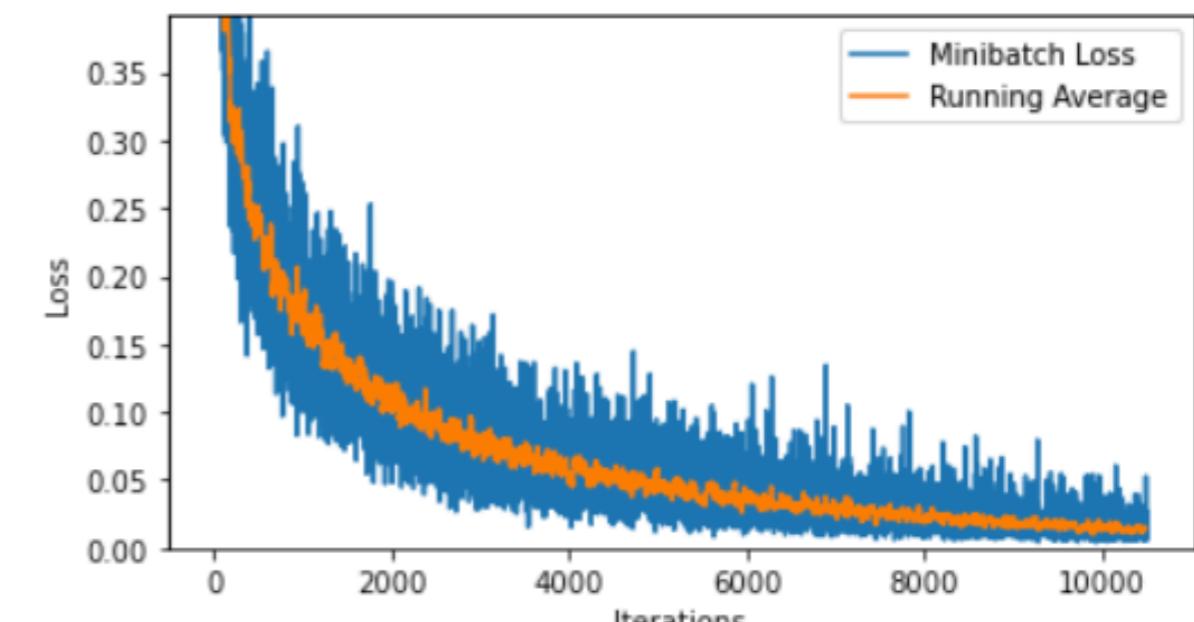
    def forward(self, x):
        logits = self.my_network(x)
        return logits
```

https://github.com/rasbt/stat453-deep-learning-ss21/blob/main/L11/code/weight_normal.ipynb

Normal (Gaussian) initialization



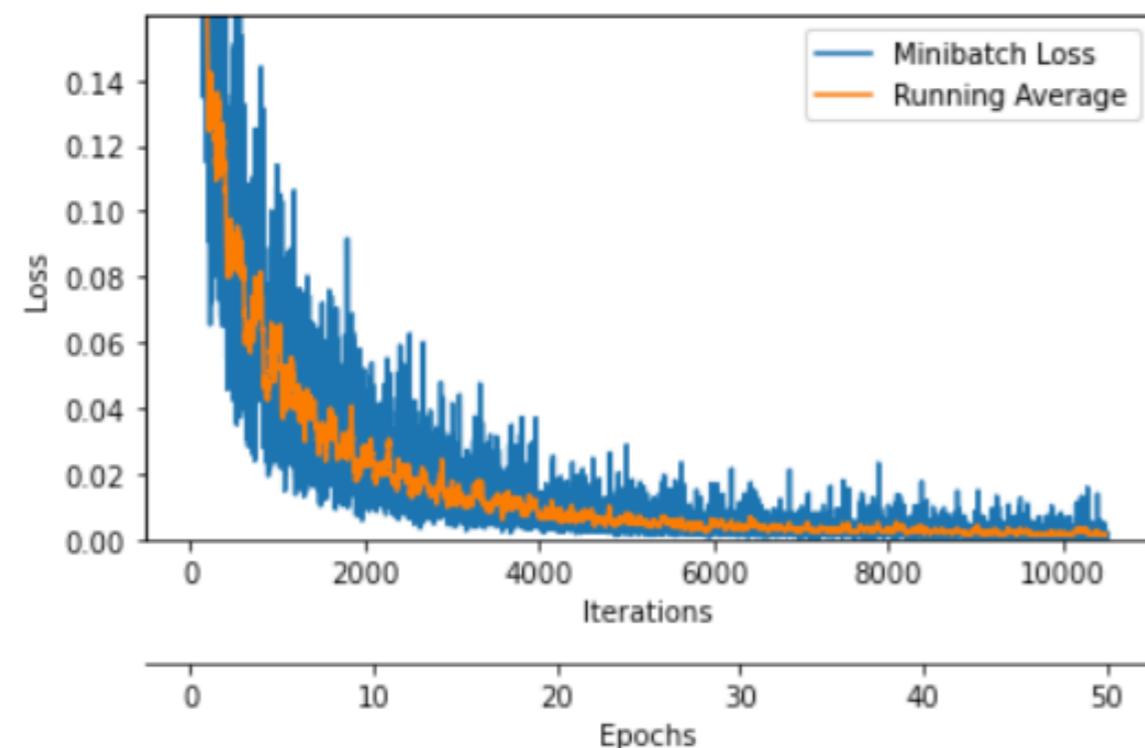
Kaiming He initialization



Note that if BatchNorm is used,
initial feature weight choice is less
important anyway



Normal (Gaussian) initialization+ BatchNorm



Kaiming He initialization

