Lecture 11

Feature Normalization and Weight Initialization

STAT 479: Deep Learning, Spring 2019

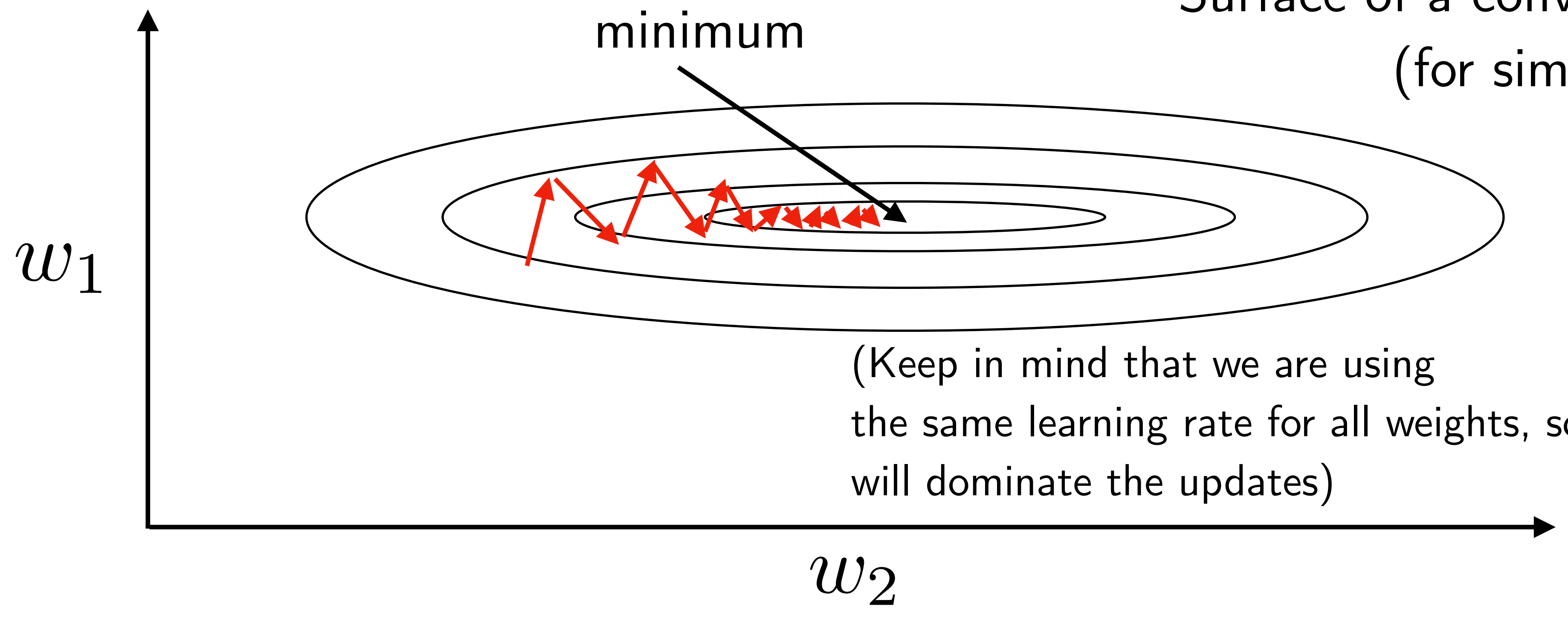
Sebastian Raschka

http://stat.wisc.edu/~sraschka/teaching/stat479-ss2019/

Overview: Additional Tricks for Neural Network Training

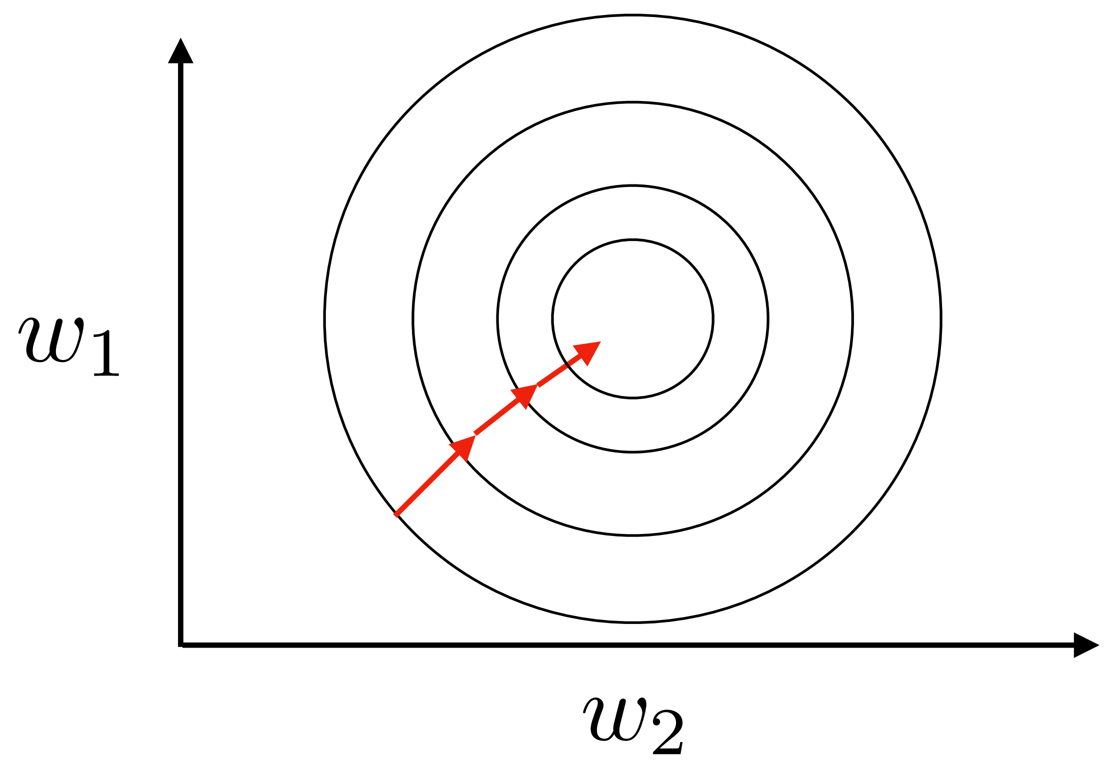
- Input Normalization (BatchNorm, InstanceNorm, GroupNorm, LayerNorm)
- Weight Initialization (Xavier, Kaiming He)
- Optimization Algorithms (RMSProp, Adagrad, ADAM)
  -- after Spring Break

Recap: 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 only affects the first hidden layer ... what about the other hidden layers?

Batch Normalization

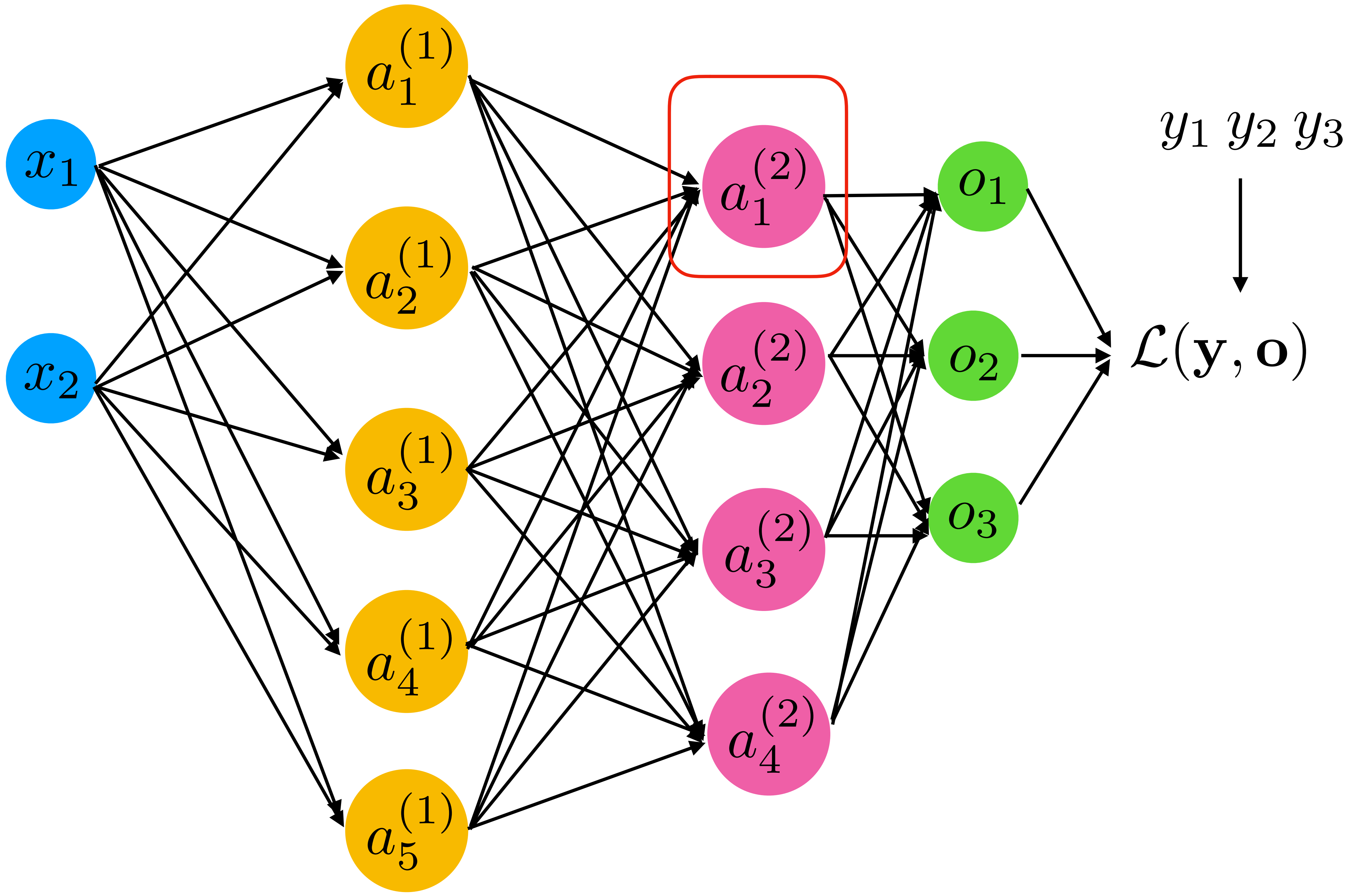
Ioffe, S., & Szegedy, C. (2015, June). Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. In *International Conference on Machine Learning* (pp. 448-456).

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

Batch Normalization

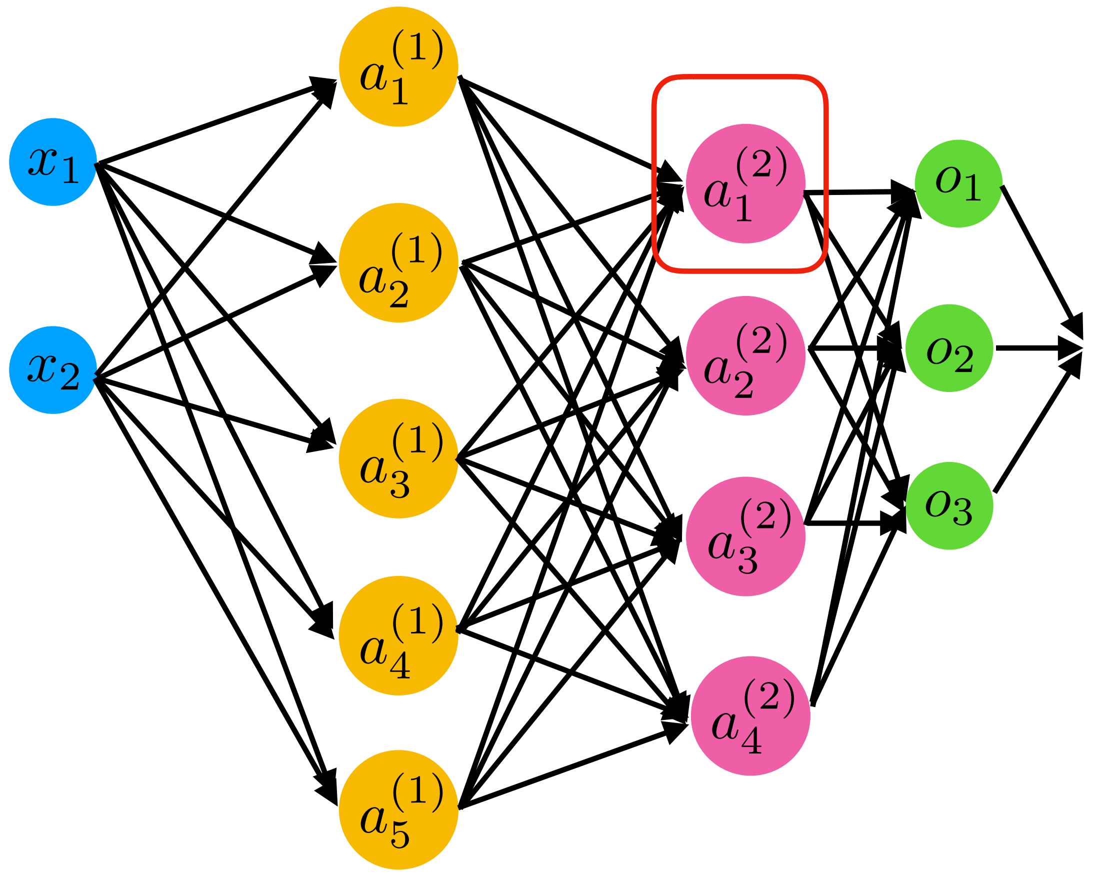
- Normalization of inputs for hidden layers
- Helps with exploding/vanishing gradient problems
- Can increase training stability and convergence rate
- Can be understood as additional normalization layers (with additional parameters)

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



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

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

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

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

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

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

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

These are learnable parameters

BatchNorm Step 2: Pre-Activation Scaling

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

$$a'^{[i]}_j = \underbrace{\gamma_j}_{\text{Controls the spread or scale}} \cdot z'^{[i]}_j + \underbrace{\beta_j}_{\text{Controls the mean}}$$

BatchNorm Step 2: Pre-Activation Scaling

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

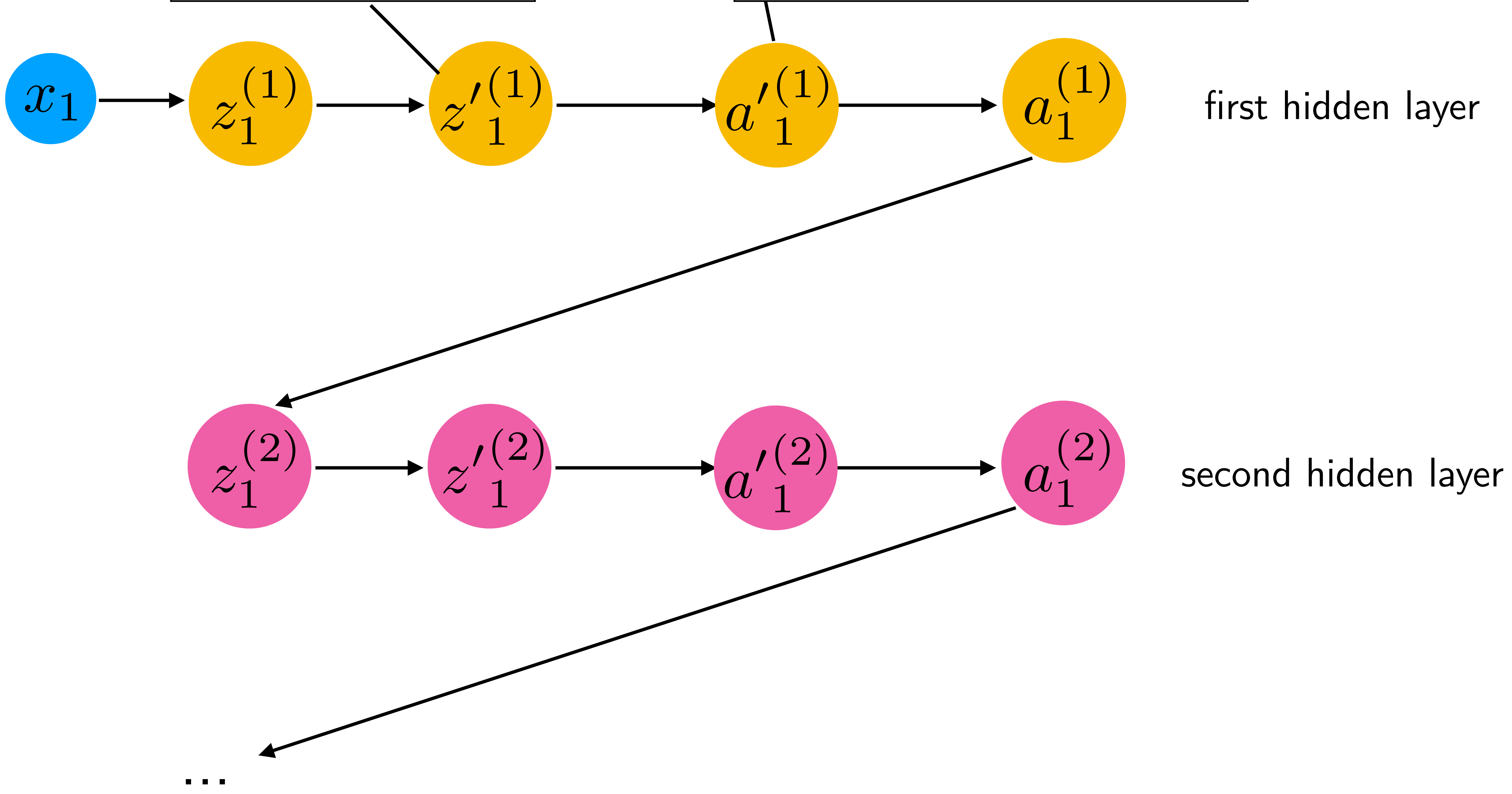
$$a'^{[i]}_j = \underbrace{\gamma_j}_{\text{Controls the spread or scale}} \cdot z'^{[i]}_j + \underbrace{\beta_j}_{\text{Controls the mean}}$$

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

BatchNorm Step 1 & 2 Summarized

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

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



BatchNorm -- Additional Things to Consider

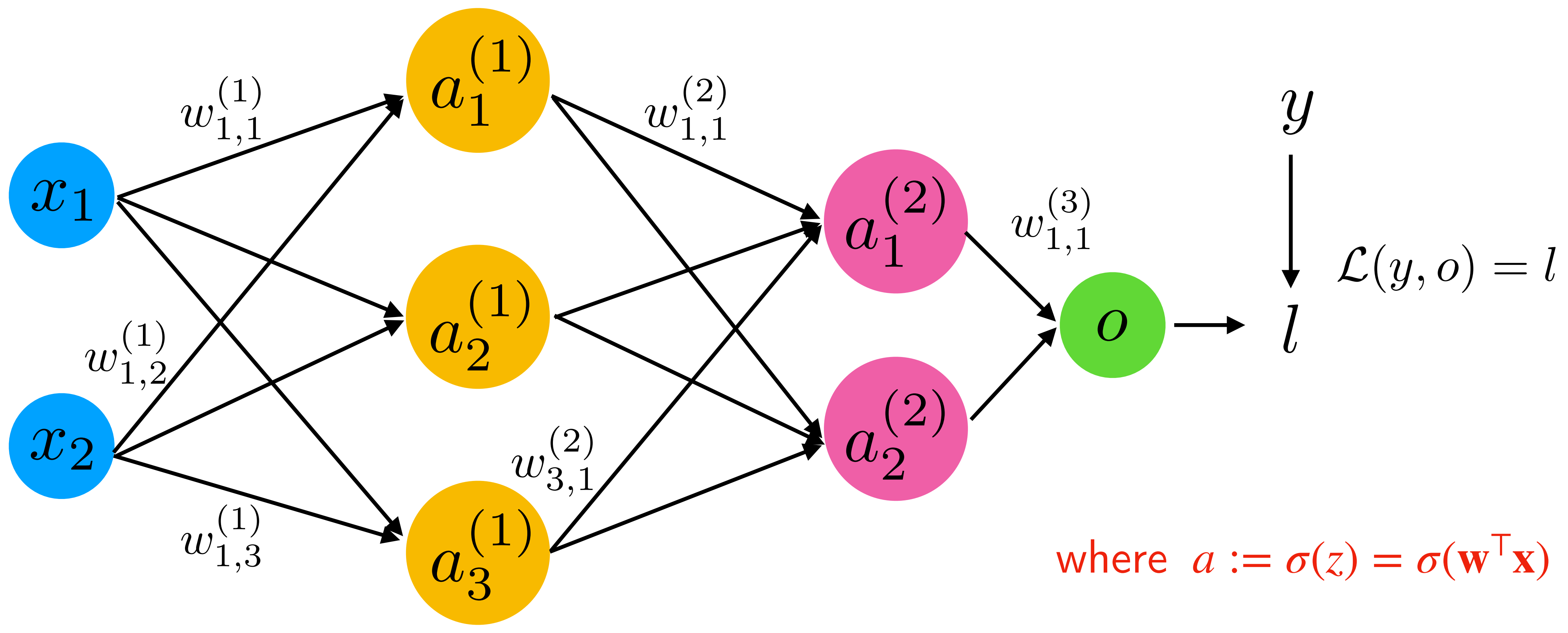
$${a'}_j^{[i]} = \gamma_j \cdot {z'}_j^{[i]} + \beta_j$$

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

Backpropagation for BatchNorm Parameters

Reminder: Multilayer Perceptron (from lecture 9)



where $a := \sigma(z) = \sigma(\mathbf{w}^\top \mathbf{x})$

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

(Assume network for binary classification)

Let's consider a simpler case ...

$$\begin{tikzcd}
 & & & & y \arrow[d, "\mathcal{L}(y, o) = l"] \\
x_1 \arrow[r, "w_j^{(1)}"] & a_j^{(1)} \arrow[r, "w_j^{(2)}"] & a_j^{(2)} \arrow[r, "w_j^{(3)}"] & o \arrow[r] & l
\end{tikzcd}$$

$$\frac{\partial l}{\partial w_j^{(3)}} = \frac{\partial l}{\partial o} \cdot \frac{\partial o}{\partial w_j^{(3)}}$$

$$\frac{\partial l}{\partial w_j^{(2)}} = \frac{\partial l}{\partial o} \cdot \frac{\partial o}{\partial a_j^{(2)}} \cdot \frac{\partial a_j^{(2)}}{\partial w_j^{(2)}}$$

$$\frac{\partial l}{\partial w_j^{(1)}} = \frac{\partial l}{\partial o} \cdot \frac{\partial o}{\partial a_j^{(2)}} \cdot \frac{\partial a_j^{(2)}}{\partial a_j^{(1)}} \cdot \frac{\partial a_j^{(1)}}{\partial w_j^{(1)}}$$

Same as on previous slide, but more verbose ...

$$\begin{tikzcd}
 & & & & y \arrow[d, "\mathcal{L}(y,o)=l"] \\
x_1 \arrow[r, "w_j^{(1)}"] & a_j^{(1)} \arrow[r, "w_j^{(2)}"] & a_j^{(2)} \arrow[r, "w_j^{(3)}"] & o \arrow[r] & l
\end{tikzcd}$$

$$\frac{\partial l}{\partial w_j^{(3)}} = \frac{\partial l}{\partial o} \cdot \frac{\partial o}{\partial w_j^{(3)}} \longrightarrow \frac{\partial l}{\partial w_j^{(3)}} = \frac{\partial l}{\partial o} \cdot \frac{\partial o}{\partial z_j^{(3)}} \cdot \frac{\partial z_j^{(3)}}{\partial w_j^{(3)}}$$

$$\frac{\partial l}{\partial w_j^{(2)}} = \frac{\partial l}{\partial o} \cdot \frac{\partial o}{\partial a_j^{(2)}} \cdot \frac{\partial a_j^{(2)}}{\partial w_j^{(2)}} \longrightarrow \cdots$$

$$\frac{\partial l}{\partial w_j^{(1)}} = \frac{\partial l}{\partial o} \cdot \frac{\partial o}{\partial a_j^{(2)}} \cdot \frac{\partial a_j^{(2)}}{\partial a_j^{(1)}} \cdot \frac{\partial a_j^{(1)}}{\partial w_j^{(1)}} \longrightarrow \cdots$$

(previously, we didn't write the net input explicitly in the comp. graph)

$$\begin{tikzcd}
 & & & & & & y \arrow[d, "\mathcal{L}(y, o) = l"] \\
x_1 \arrow[r, "w_j^{(1)}"] & a_j^{(1)} \arrow[r, "w_j^{(2)}"] & z_j^{(2)} \arrow[r, "\sigma(\cdot)"] & a_j^{(2)} \arrow[r, "w_j^{(3)}"] & \ldots & \arrow[r] & l
\end{tikzcd}$$

Adding a BatchNorm layer ...

$$\begin{tikzcd}
x_1 \arrow[r, "w_j^{(1)}"] & a_j^{(1)} \arrow[r, "w_j^{(2)}"] & z_j^{(2)} \arrow[r] & {z'}_j^{(2)} \arrow[r] & {a'}_j^{(2)} \arrow[r, "\sigma(\cdot)"] & a_j^{(2)} \arrow[r, "w_j^{(3)}"] & \ldots
\end{tikzcd}$$

$${z'}_j^{(2)} = \frac{z_j^{(2)[i]} - \mu_j}{\sigma_j} \qquad {a'}_j^{(2)} = \gamma_j \cdot {z'}_j^{(2)} + \beta_j$$

Backprop for BatchNorm Parameters

$$z'^{(2)}_j = \frac{z^{(2)[i]}_j - \mu_j}{\sigma_j} \qquad a'^{(2)}_j = \gamma_j \cdot z'^{(2)}_j + \beta_j$$

$$\begin{tikzcd}
x_1 \arrow[r, "w^{(1)}_j"] & a^{(1)}_j \arrow[r, "w^{(2)}_j"] & z^{(2)}_j \arrow[r] & z'^{(2)}_j \arrow[r] & a'^{(2)}_j \arrow[r, "\sigma(\cdot)"] & a^{(2)}_j \arrow[r, "w^{(3)}_j"] & \cdots
\end{tikzcd}$$

$$\frac{\partial l}{\partial \beta_j} = \sum_{i=1}^{n} \frac{\partial l}{\partial a'^{(2)[i]}_j} \cdot \frac{\partial a'^{(2)[i]}_j}{\partial \beta_j} = \sum_{i=1}^{n} \frac{\partial l}{\partial a'^{(2)[i]}_j}$$

$$\frac{\partial l}{\partial \gamma_j} = \sum_{i=1}^{n} \frac{\partial l}{\partial a'^{(2)[i]}_j} \cdot \frac{\partial a'^{(2)[i]}_j}{\partial \gamma_j} = \sum_{i=1}^{n} \frac{\partial l}{\partial a'^{(2)[i]}_j} \cdot z'^{(2)[i]}_j$$

Backprop Beyond the BatchNorm Layer

$$\begin{tikzcd}
x_1 \arrow[r, "w_j^{(1)}"] & a_j^{(1)} \arrow[r, "w_j^{(2)}"] & z_j^{(2)} \arrow[r] & {z'}_j^{(2)} \arrow[r] & {a'}_j^{(2)} \arrow[r, "\sigma(\cdot)"] & a_j^{(2)} \arrow[r, "w_j^{(3)}"] & \cdots
\end{tikzcd}$$

Since the minibatch mean and variance act as parameters, we can/have to apply the multivariable chain rule

$$\frac{\partial l}{\partial z_j^{(2)[i]}} = \frac{\partial l}{\partial {z'}_j^{(2)[i]}} \cdot \frac{\partial {z'}_j^{(2)[i]}}{\partial z_j^{(2)[i]}} + \frac{\partial l}{\partial \mu_j} \cdot \frac{\partial \mu_j}{\partial z_j^{(2)[i]}} + \frac{\partial l}{\partial \sigma_j^2} \cdot \frac{\partial \sigma_j^2}{\partial z_j^{(2)[i]}}$$

$$= \frac{\partial l}{\partial {z'}_j^{(2)[i]}} \cdot \frac{1}{\sigma_j} + \frac{\partial l}{\partial \mu_j} \cdot \frac{1}{n} + \frac{\partial l}{\partial \sigma_j^2} \cdot \frac{1(z_j^{(2)} - \mu_j)}{n}$$

Backprop for BatchNorm Parameters

$$\frac{\partial l}{\partial z_j^{(2)[i]}} = \frac{\partial l}{\partial {z'}_j^{(2)[i]}} \cdot \frac{\partial {z'}_j^{(2)[i]}}{\partial z_j^{(2)[i]}} + \frac{\partial l}{\partial \mu_j} \cdot \frac{\partial \mu_j}{\partial z_j^{(2)[i]}} + \frac{\partial l}{\partial \sigma_j^2} \cdot \frac{\partial \sigma_j^2}{\partial z_j^{(2)[i]}}$$

$$= \boxed{\frac{\partial l}{\partial {z'}_j^{(2)[i]}}} \cdot \frac{1}{\sigma_j} + \boxed{\frac{\partial l}{\partial \mu_j}} \cdot \frac{1}{n} + \boxed{\frac{\partial l}{\partial \sigma_j^2}} \cdot \frac{1(z_j^{(2)} - \mu_j)}{n}$$

If you like engineering math, you can solve the remaining terms as an ungraded HW exercise ;)

BatchNorm in PyTorch

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

    def __init__(self, num_features, num_classes):
        super(MultilayerPerceptron, self).__init__()

        ### 1st hidden layer
        self.linear_1 = torch.nn.Linear(num_features, num_hidden_1)
        self.linear_1_bn = torch.nn.BatchNorm1d(num_hidden_1)

        ### 2nd hidden layer
        self.linear_2 = torch.nn.Linear(num_hidden_1, num_hidden_2)
        self.linear_2_bn = torch.nn.BatchNorm1d(num_hidden_2)

        ### Output layer
        self.linear_out = torch.nn.Linear(num_hidden_2, num_classes)

    def forward(self, x):
        out = self.linear_1(x)
        # note that batchnorm is in the classic
        # sense placed before the activation
        out = self.linear_1_bn(out)
        out = F.relu(out)

        out = self.linear_2(out)
        out = self.linear_2_bn(out)
        out = F.relu(out)

        logits = self.linear_out(out)
        probas = F.softmax(logits, dim=1)
        return logits, probas
```

BatchNorm in PyTorch

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

    def __init__(self, num_features, num_classes):
        super(MultilayerPerceptron, self).__init__()

        ### 1st hidden layer
        self.linear_1 = torch.nn.Linear(num_features, num_hidden_1)
        self.linear_1_bn = torch.nn.BatchNorm1d(num_hidden_1)

        ### 2nd hidden layer
        self.linear_2 = torch.nn.Linear(num_hidden_1, num_hidden_2)
        self.linear_2_bn = torch.nn.BatchNorm1d(num_hidden_2)

        ### Output layer
        self.linear_out = torch.nn.Linear(num_hidden_2, num_classes)

    def forward(self, x):
        out = self.linear_1(x)
        # note that batchnorm is in the classic
        # sense placed before the activation
        out = self.linear_1_bn(out)
        out = F.relu(out)

        out = self.linear_2(out)
        out = self.linear_2_bn(out)
        out = F.relu(out)

        logits = self.linear_out(out)
        probas = F.softmax(logits, dim=1)
        return logits, probas
```

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 and Internal Covariate Shift

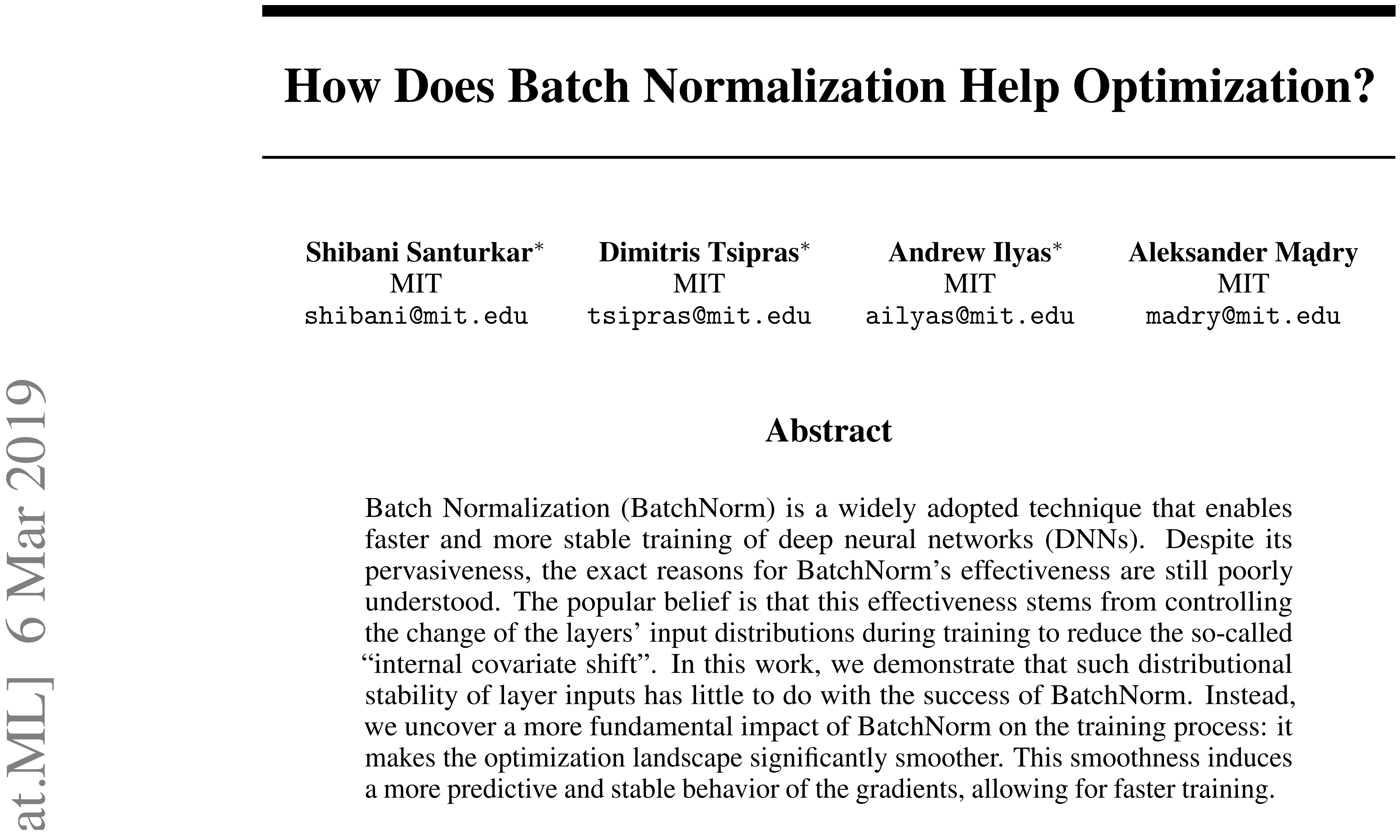
Ioffe, S., & Szegedy, C. (2015, June). Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. In *International Conference on Machine Learning* (pp. 448-456).

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 actually no guarantee or evidence that BatchNorm helps with covariate shift

In my opinion, 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

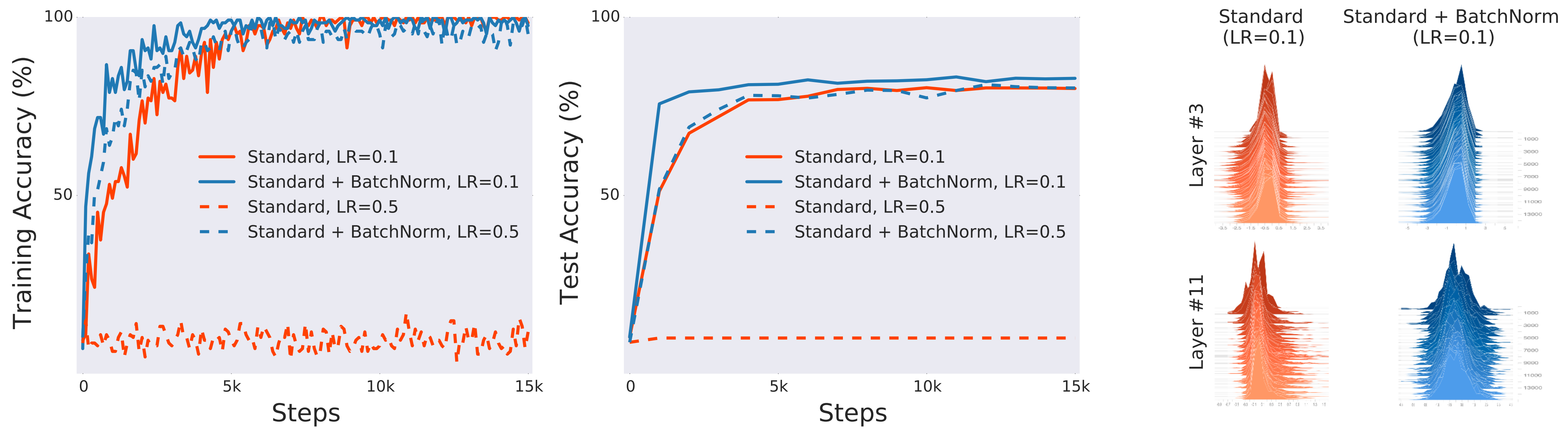


Figure 1: Comparison of (a) training (optimization) and (b) test (generalization) performance of a standard VGG network trained on CIFAR-10 with and without BatchNorm (details in Appendix A). There is a consistent gain in training speed in models with BatchNorm layers. (c) Even though the gap between 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).

Good Performance of BatchNorm Seems Unrelated to Covariate Shift Prevention

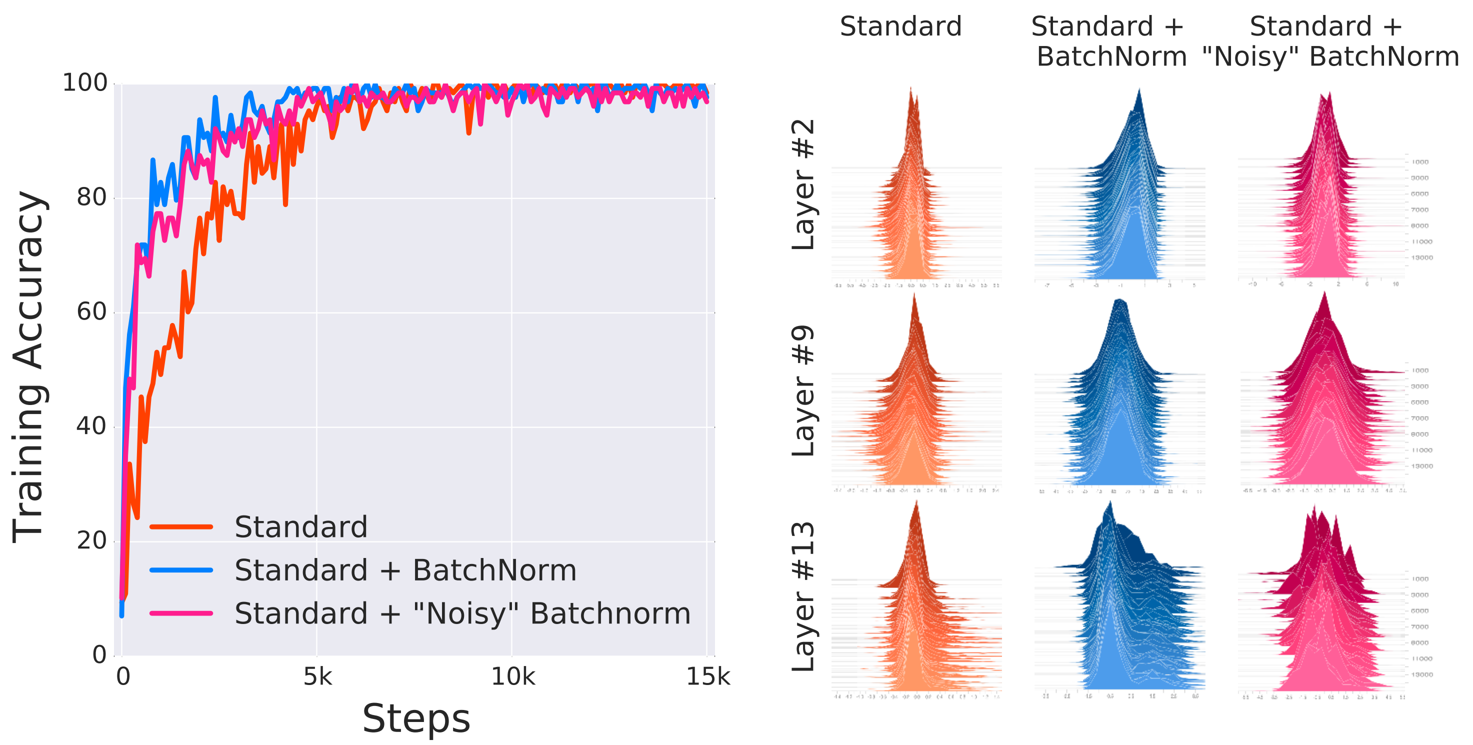


Figure 2: Connections between distributional stability and BatchNorm performance: We compare VGG networks trained without BatchNorm (Standard), with BatchNorm (Standard + BatchNorm) and with explicit "covariate shift" added to BatchNorm layers (Standard + "Noisy" BatchNorm). In the later case, we induce 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).

BatchNorm Variants

Pre-Activation

"Original" version
as discussed in
previous slides

$$\begin{tikzcd}
\text{compute net inputs} \arrow[d] \\
\text{BatchNorm} \arrow[d] \\
\text{apply activation function} \arrow[d] \\
\text{compute next-layer net inputs}
\end{tikzcd}$$

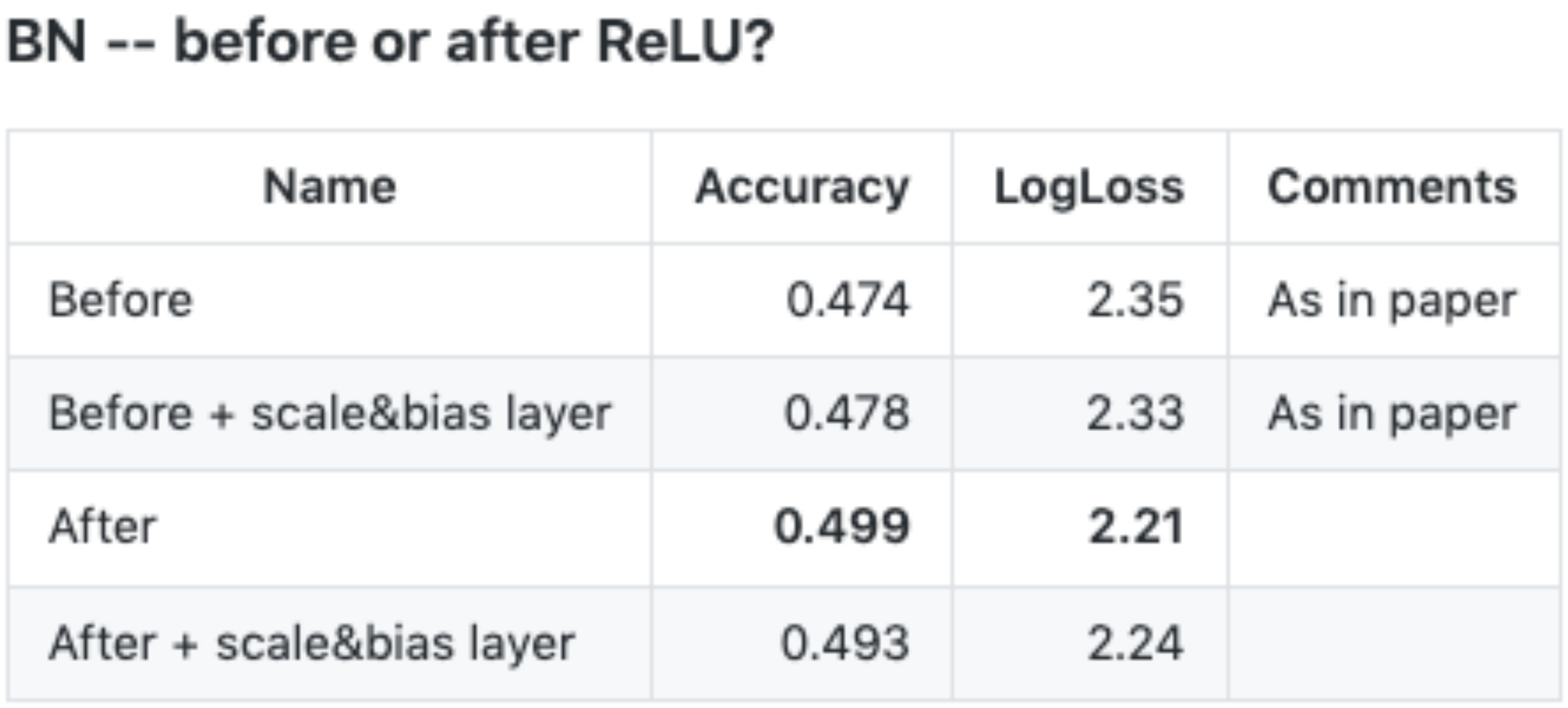
Post-Activation

May make more sense,
but less common

$$\begin{tikzcd}
\text{compute net inputs} \arrow[d] \\
\text{apply activation function} \arrow[d] \\
\text{BatchNorm} \arrow[d] \\
\text{compute next-layer net inputs}
\end{tikzcd}$$

Some Benchmarks

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

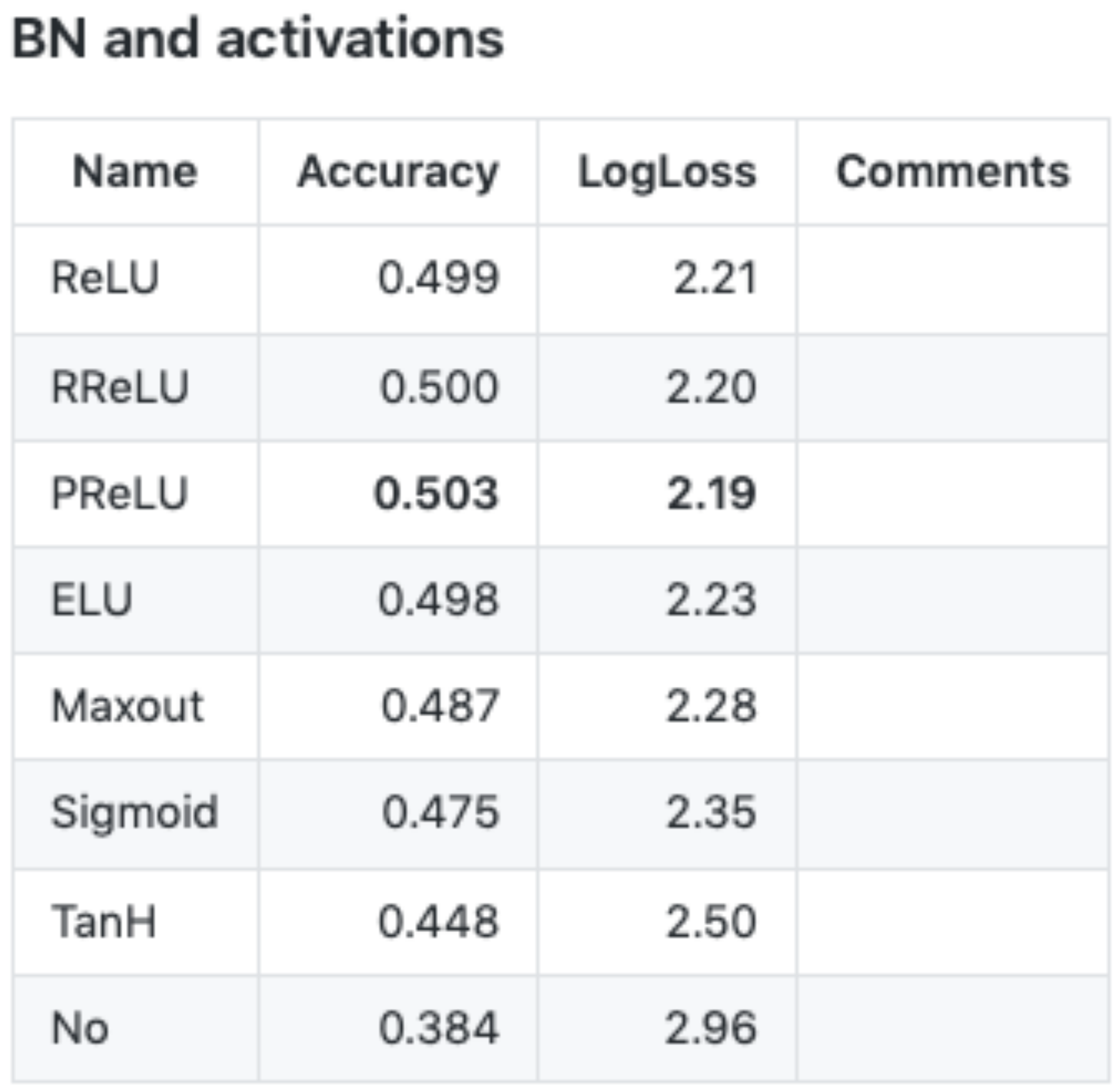


BN -- before or after ReLU?

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	

Some Benchmarks

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

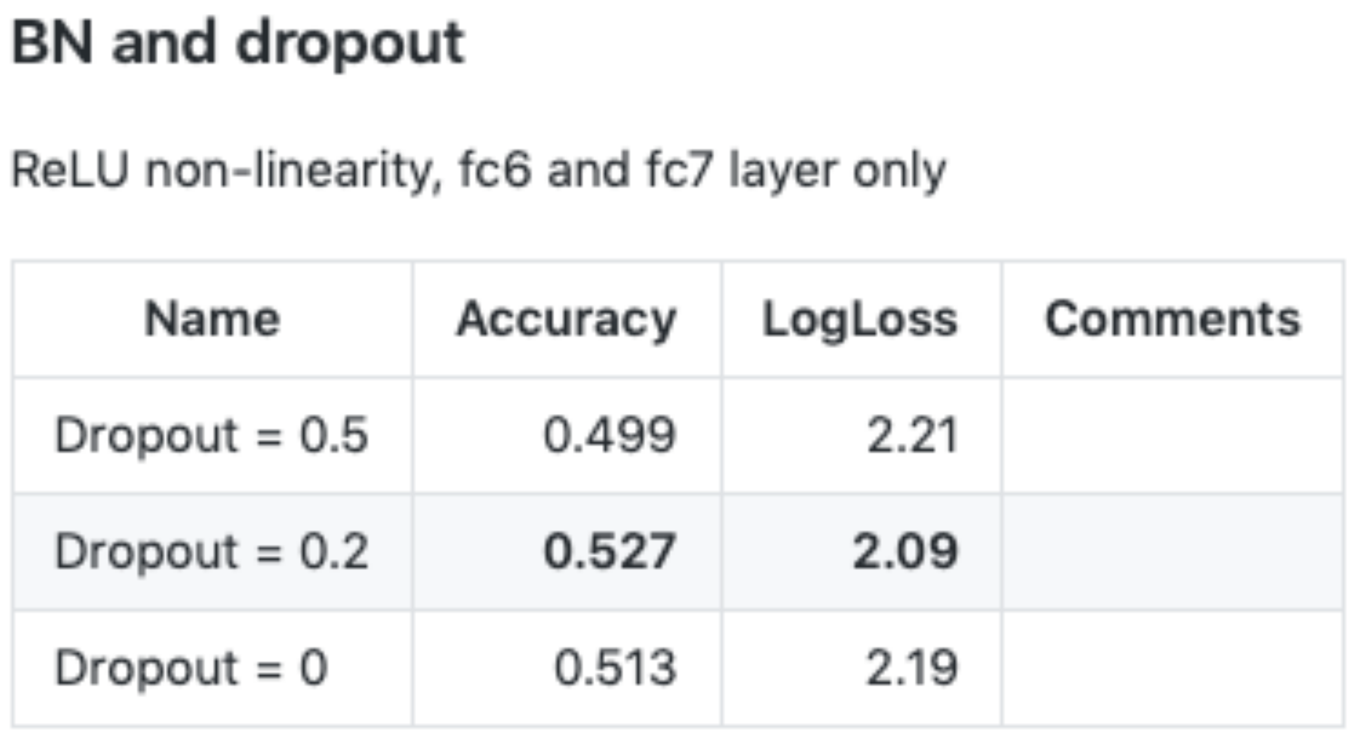


BN and activations

Name	Accuracy	LogLoss	Comments
ReLU	0.499	2.21	
RReLU	0.500	2.20	
PReLU	**0.503**	**2.19**	
ELU	0.498	2.23	
Maxout	0.487	2.28	
Sigmoid	0.475	2.35	
TanH	0.448	2.50	
No	0.384	2.96	

Some Benchmarks

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



BN and dropout

ReLU non-linearity, fc6 and fc7 layer only

Name	Accuracy	LogLoss	Comments
Dropout = 0.5	0.499	2.21	
Dropout = 0.2	**0.527**	**2.09**	
Dropout = 0	0.513	2.19	

Practical Consideration

BatchNorm become more stable with larger minibatch sizes

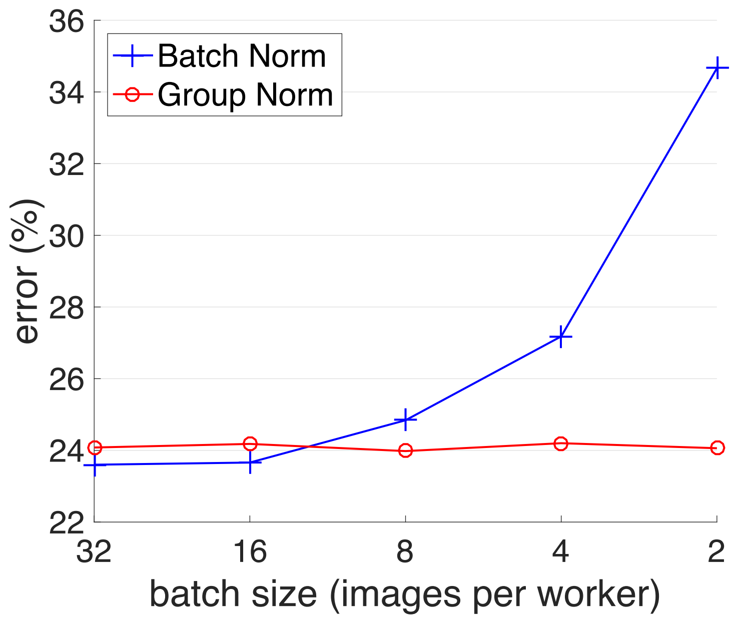


Figure 1. ImageNet classification error *vs*. batch sizes. The model is ResNet-50 trained in the ImageNet training set using 8 workers (GPUs) and evaluated in the validation set. BN's error increases rapidly when reducing the batch size. GN's computation is independent of batch sizes, and its error rate 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).

Other Normalization Methods for Hidden Activations

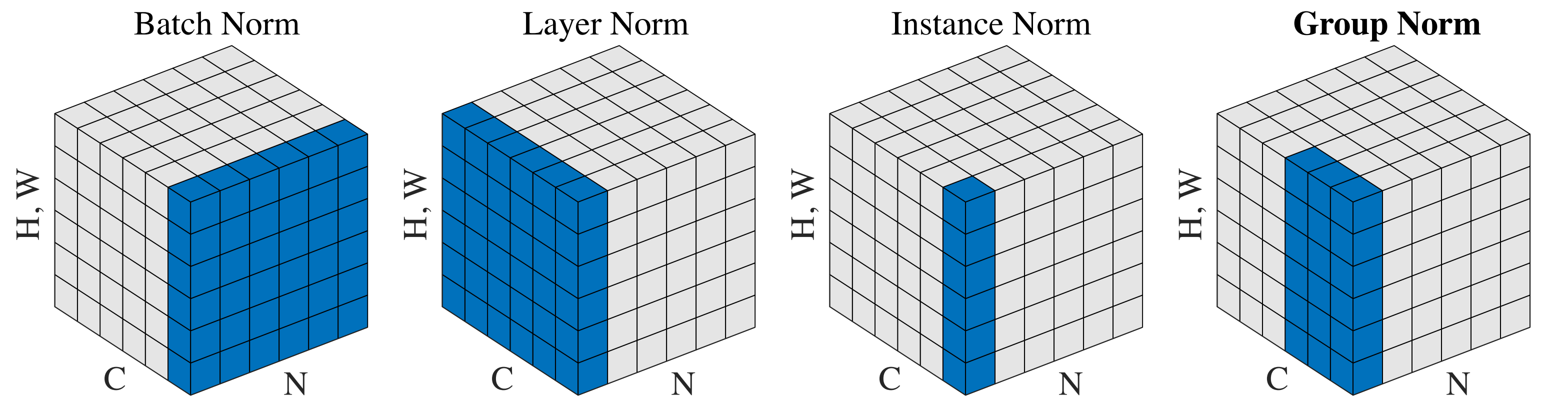


Figure 2. Normalization methods. Each subplot shows a feature map tensor. The pixels in blue are normalized by the same mean and variance, computed by aggregating the values of these pixels. Group Norm is illustrated using a group number of 2.

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

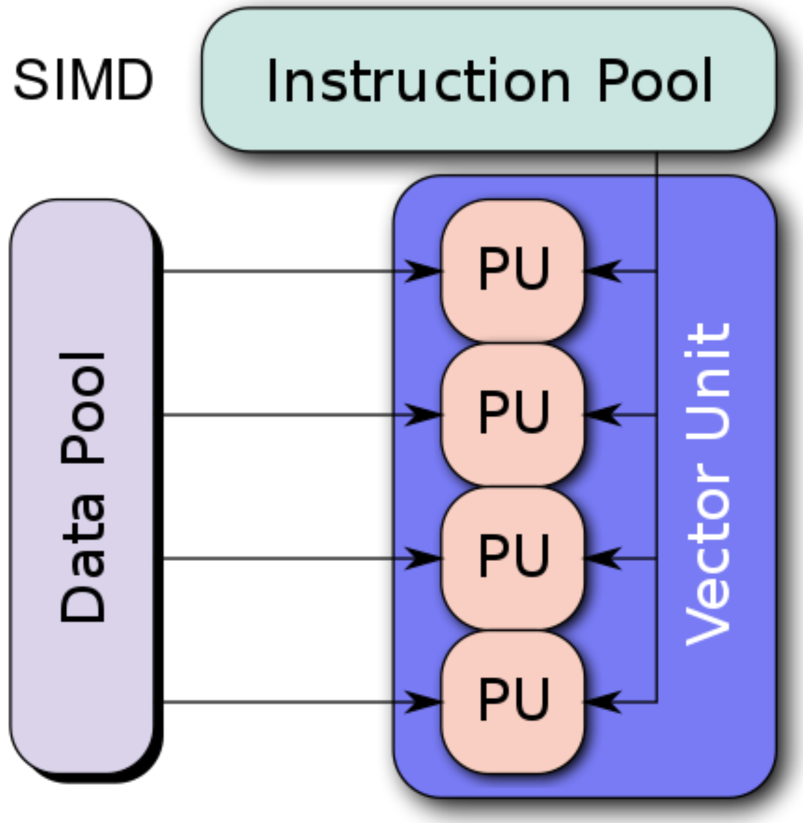
(will revisit after Spring Break after introducing Convolutional Neural Networks)

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 48 columns in a matrix, we can map 3 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

(It might be one of the archaic DL conventions/traditions, and I don't think this matters much anymore for modern frameworks)

Reading Assignments (Optional)

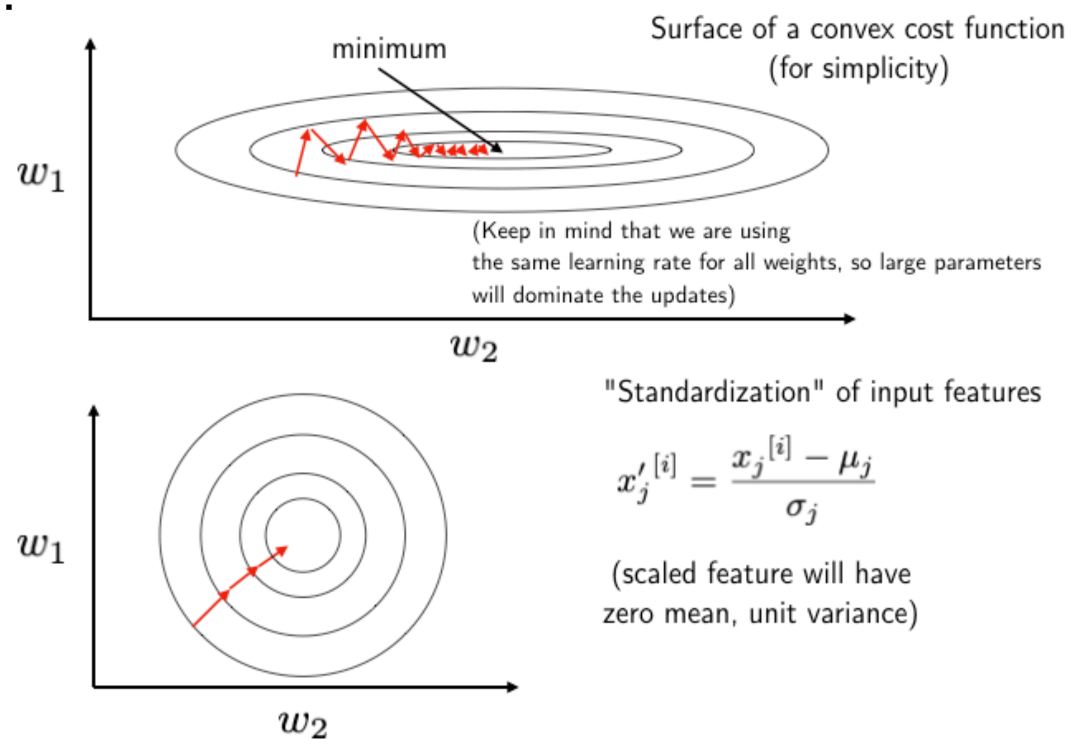
Ioffe, S., & Szegedy, C. (2015, June). Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. In *International Conference on Machine Learning* (pp. 448-456).

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

Weight Initialization

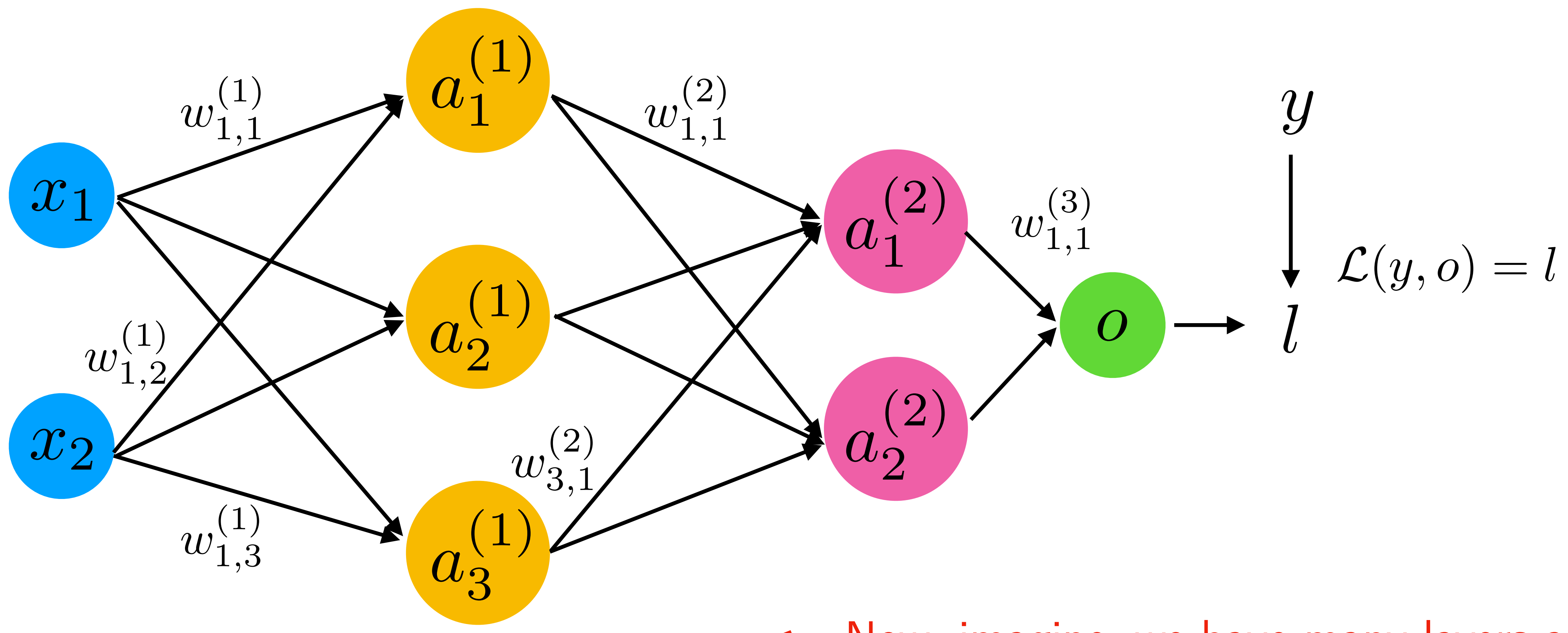
- We previously discussed that we want to initialize weight to small, random numbers to break symmetry
- Also, we want the weights to be relatively, why?

Tip (from an earlier slide):



minimum
Surface of a convex cost function (for simplicity)
w_1
w_2
(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
$x'^{[i]}_j = \frac{x_j^{[i]} - \mu_j}{\sigma_j}$
(scaled feature will have zero mean, unit variance)
w_1
w_2

Sidenote: Vanishing/Exploding Gradient problems



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

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

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

Sidenote: Vanishing/Exploding Gradient problems

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

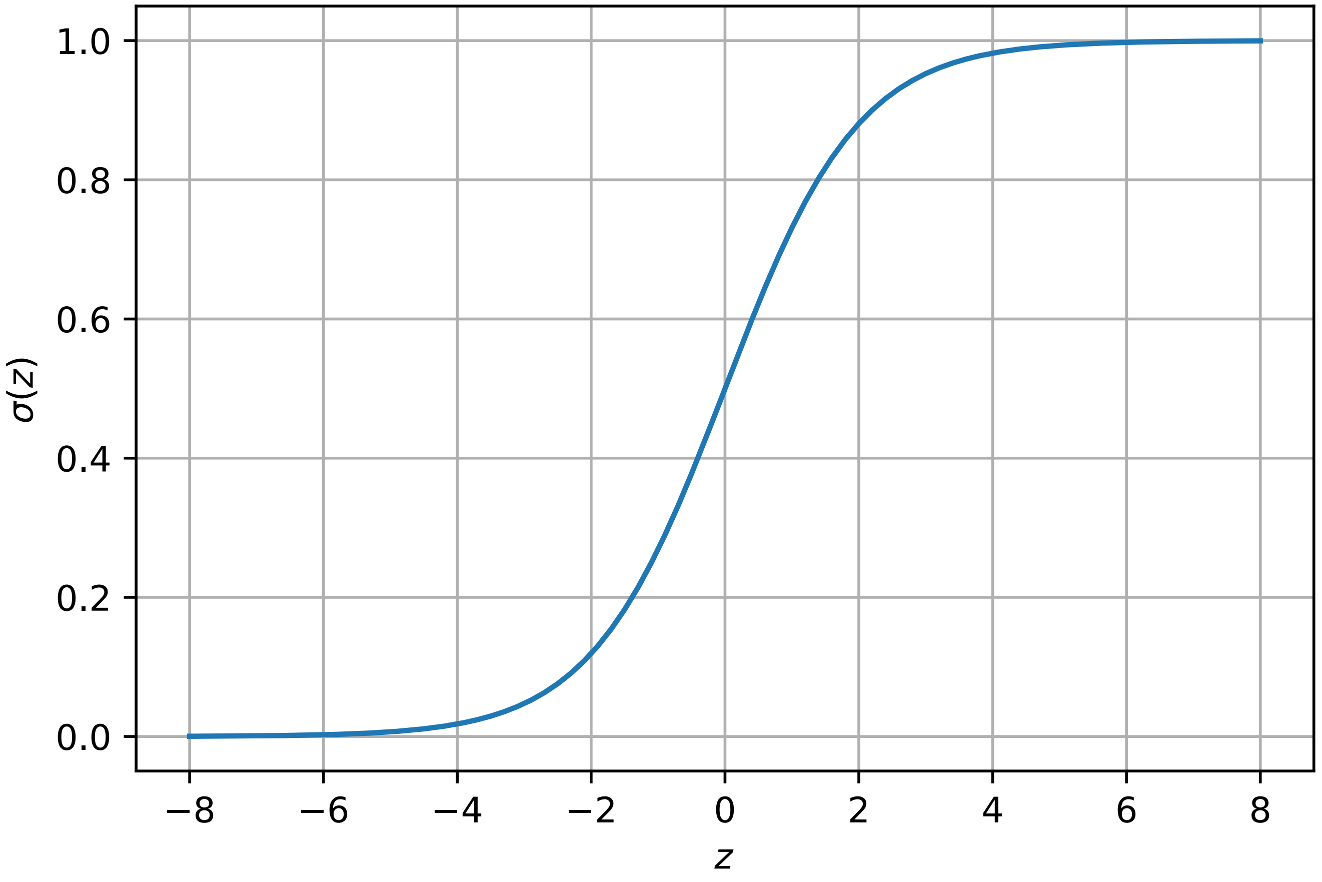
$$\frac{\partial l}{\partial w_{1,1}^{(1)}} = \frac{\partial l}{\partial o} \cdot \frac{\partial o}{\partial a_1^{(2)}} \cdot \boxed{\frac{\partial a_1^{(2)}}{\partial a_1^{(1)}}} \cdot \frac{\partial a_1^{(1)}}{\partial w_{1,1}^{(1)}} + \frac{\partial l}{\partial o} \cdot \frac{\partial o}{\partial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

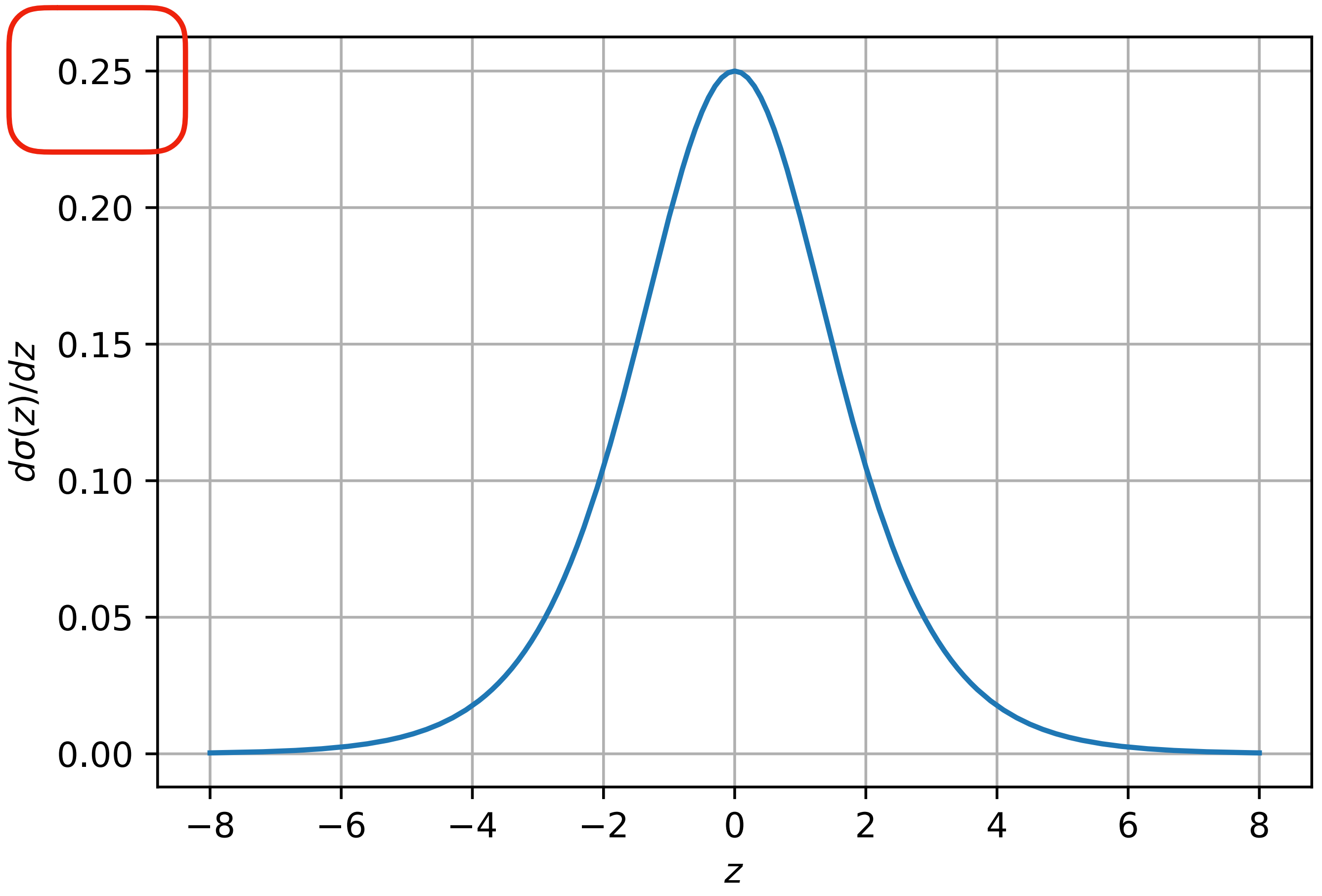
$$\sigma'(z^{[i]}) = \sigma(z^{[i]}) \cdot (1 - \sigma(z^{[i]}))$$

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!

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

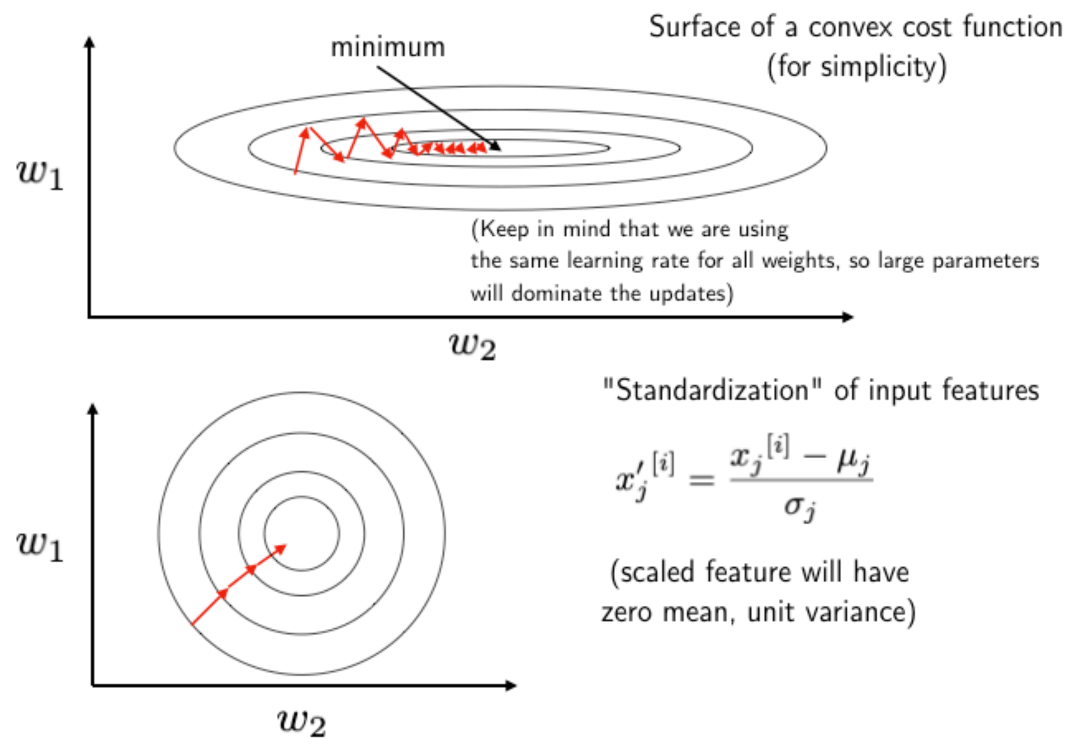
Sidenote: Vanishing/Exploding Gradient problems

How, do you think, does ReLU behave?

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)
- When would you choose which?

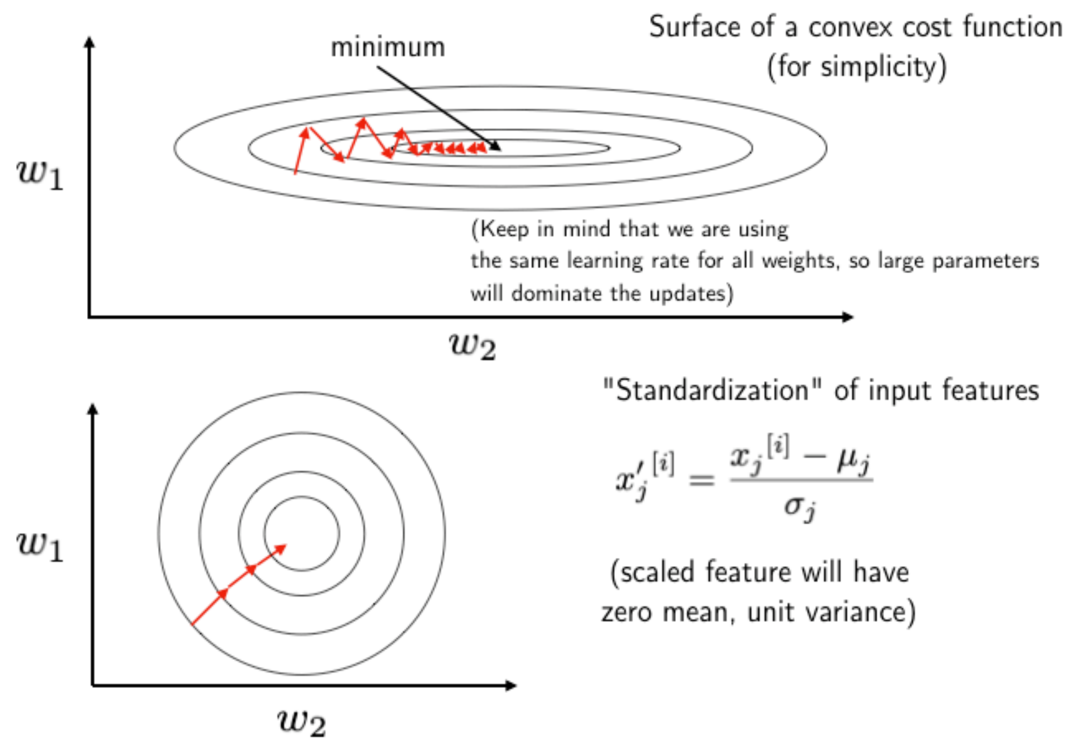
Tip (from an earlier slide):


minimum
Surface of a convex cost function
(for simplicity)
w_1
w_2
(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
$x'^{[i]}_j = \frac{x_j^{[i]} - \mu_j}{\sigma_j}$
(scaled feature will have zero mean, unit variance)
w_1
w_2

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)
- When would you choose which?

Tip (from an earlier slide):


minimum
Surface of a convex cost function
(for simplicity)
w_1
w_2
(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
$x'^{[i]}_j = \frac{x_j^{[i]} - \mu_j}{\sigma_j}$
(scaled feature will have zero mean, unit variance)
w_1
w_2

Sidenote: You can initialize the bias units to all zeros

Custom Weight Initialization in PyTorch

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

    def __init__(self, num_features, num_hidden, num_classes):
        super(MLP, self).__init__()

        self.num_classes = num_classes

        ### 1st hidden layer
        self.linear_1 = torch.nn.Linear(num_features, num_hidden)
        self.linear_1.weight.detach().normal_(0.0, 0.1)
        self.linear_1.bias.detach().zero_()

        ### Output layer
        self.linear_out = torch.nn.Linear(num_hidden, num_classes)
        self.linear_out.weight.detach().normal_(0.0, 0.1)
        self.linear_out.bias.detach().zero_()

    def forward(self, x):
        out = self.linear_1(x)
        out = torch.sigmoid(out)
        logits = self.linear_out(out)
        probas = torch.sigmoid(logits)
        return logits, probas
```

Custom Weight Initialization in PyTorch

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

    def __init__(self, num_features, num_hidden, num_classes):
        super(MLP, self).__init__()

        self.num_classes = num_classes

        ### 1st hidden layer
        self.linear_1 = torch.nn.Linear(num_features, num_hidden)
        self.linear_1.weight.detach().normal_(0.0, 0.1)
        self.linear_1.bias.detach().zero_()

        ### Output layer
        self.linear_out = torch.nn.Linear(num_hidden, num_classes)
        self.linear_out.weight.detach().normal_(0.0, 0.1)
        self.linear_out.bias.detach().zero_()

    def forward(self, x):
        out = self.linear_1(x)
        out = torch.sigmoid(out)
        logits = self.linear_out(out)
        probas = torch.sigmoid(logits)
        return logits, probas
```

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.

- 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

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.

<u>Method:</u>

<u>Step 1</u>: Initialize weights from Gaussian or uniform distribution with (previous slide)

<u>Step 2</u>: 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

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.

<u>Method:</u>

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

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.

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

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

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

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

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

$$\mathrm{Var}\left(a^{(l)}\right) \approx \mathrm{Var}\left(z^{(l)}\right) = \mathrm{Var}\left(z_j^{(l)}\right) = \mathrm{Var}\left(\sum_{i=1}^{m_{l-1}} W_{jk}^{(l)} a_k^{(l-1)}\right)$$

$$= \sum_{i=1}^{m^{(l-1)}} \mathrm{Var}\left[W_{jk}^{(l)} a_k^{(l-1)}\right] = \sum_{i=1}^{m^{(l-1)}} \mathrm{Var}\left[W_{jk}^{(l)}\right] \mathrm{Var}\left[a_k^{(l-1)}\right]$$

$$= \sum_{i=1}^{m^{(l-1)}} \mathrm{Var}\left[W^{(l)}\right] \mathrm{Var}\left[a^{(l-1)}\right] = m^{(l-1)} \mathrm{Var}\left[W^{(l)}\right] \mathrm{Var}\left[a^{(l-1)}\right]$$

Xavier Initialization in PyTorch

Semi-Automatic:

```
...
self.linear = torch.nn.Linear(...)
torch.nn.init.xavier_uniform_(conv1.weight)
...
```

More conveniently for all weights in e.g., fully-connected layers:

```
...
def weights_init(m):
    if isinstance(m, nn.Linear):
        torch.nn.init.xavier_uniform_(m.weight)
        torch.nn.init.xavier_uniform_(m.bias)

model.apply(weights_init)
...
```

Weight Initialization -- Xavier Initialization

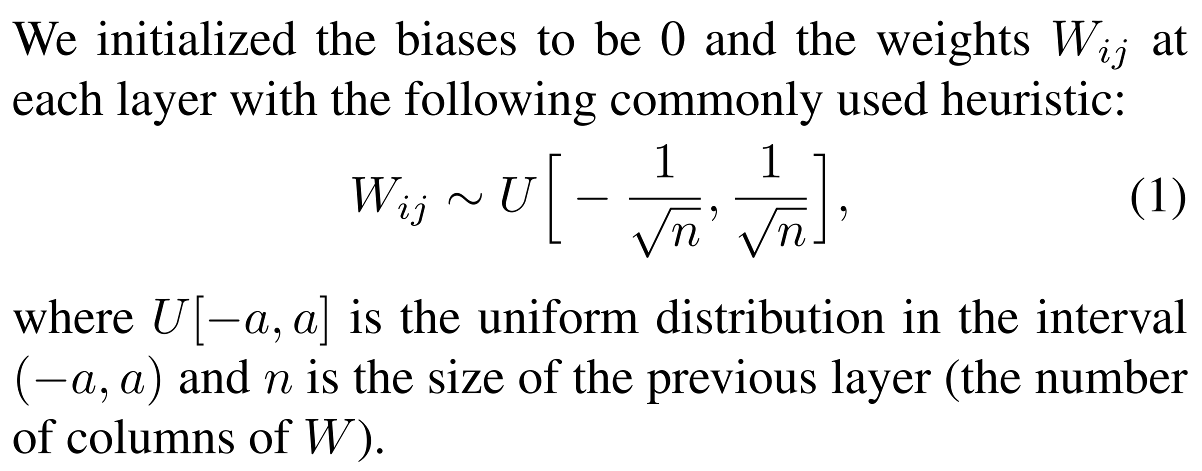
$$\mathbf{W}^{(l)} := \mathbf{W}^{(l)} \cdot \sqrt{\frac{1}{m^{[l-1]}}}$$

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

Weight Initialization -- Xavier Initialization

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

From the original paper:



We initialized the biases to be 0 and the weights W_{ij} at each layer with the following commonly used heuristic:

$$W_{ij} \sim U\left[-\frac{1}{\sqrt{n}}, \frac{1}{\sqrt{n}}\right], \tag{1}$$

where $U[-a, a]$ is the uniform distribution in the interval $(-a, a)$ and n is the size of the previous layer (the number of columns of W).

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

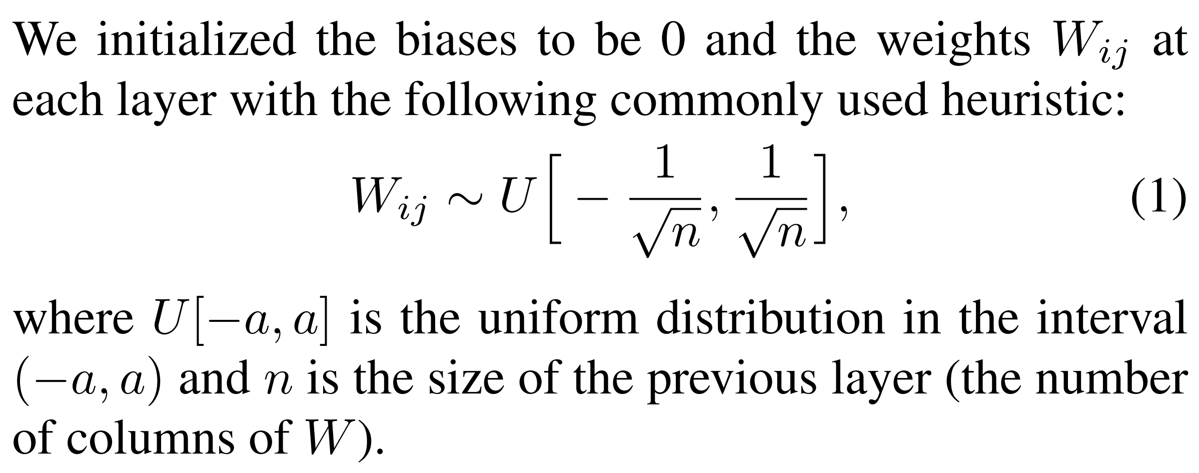
Glorot, Xavier, and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks." *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. 2010.

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

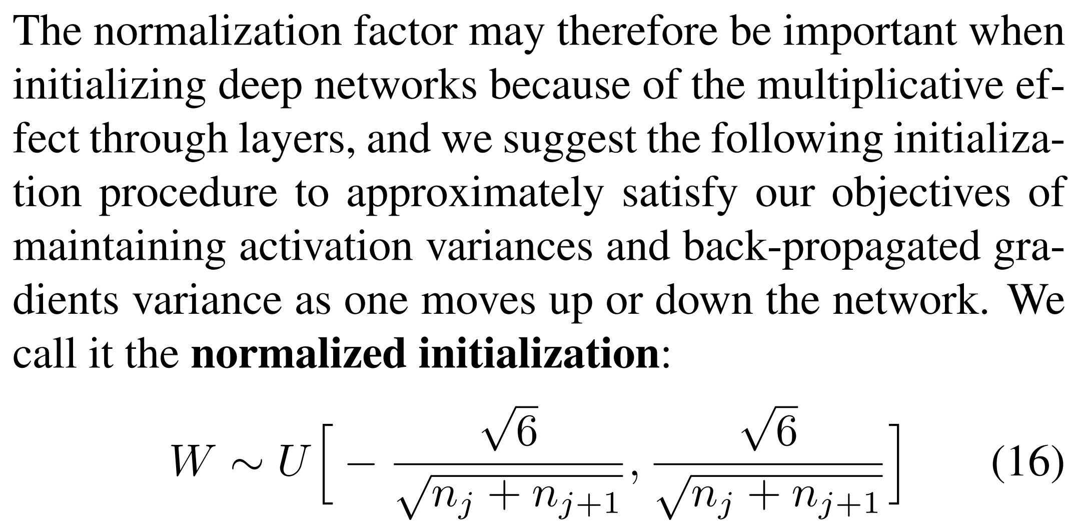
From the original paper:


We initialized the biases to be 0 and the weights W_{ij} at each layer with the following commonly used heuristic:

$$W_{ij} \sim U\Big[-\frac{1}{\sqrt{n}}, \frac{1}{\sqrt{n}}\Big], \tag{1}$$

where $U[-a, a]$ is the uniform distribution in the interval $(-a, a)$ and n is the size of the previous layer (the number of columns of W).

Also from the original paper:

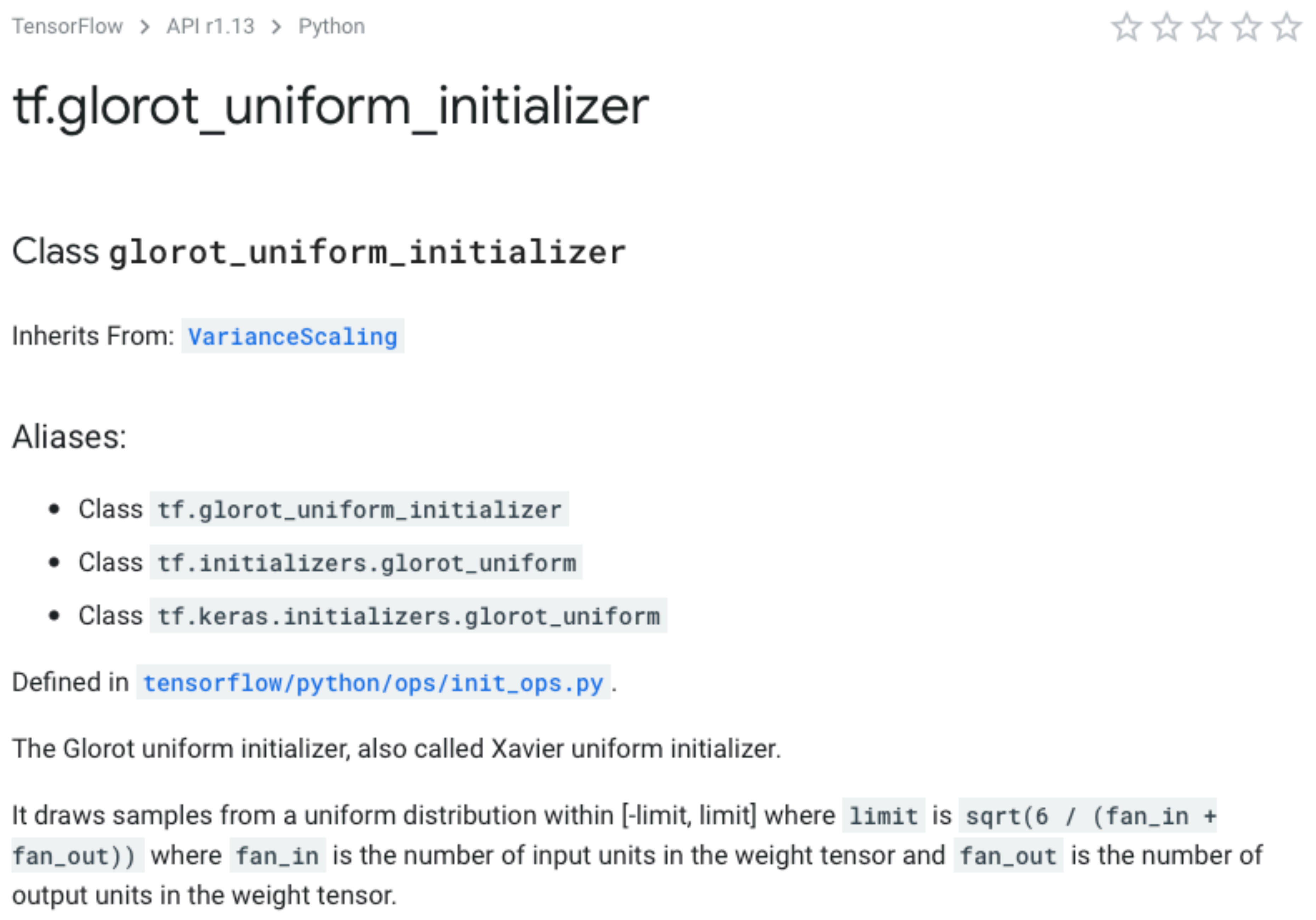

The normalization factor may therefore be important when initializing deep networks because of the multiplicative effect through layers, and we suggest the following initialization procedure to approximately satisfy our objectives of maintaining activation variances and back-propagated gradients variance as one moves up or down the network. We call it the **normalized initialization**:

$$W \sim U\Big[-\frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}, \frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}\Big] \tag{16}$$

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.

Weight Initialization -- Xavier Initialization

There are many variants used in different frameworks:



TensorFlow > API r1.13 > Python

tf.glorot_uniform_initializer

Class `glorot_uniform_initializer`

Inherits From: `VarianceScaling`

Aliases:

- Class `tf.glorot_uniform_initializer`
- Class `tf.initializers.glorot_uniform`
- Class `tf.keras.initializers.glorot_uniform`

Defined in `tensorflow/python/ops/init_ops.py`.

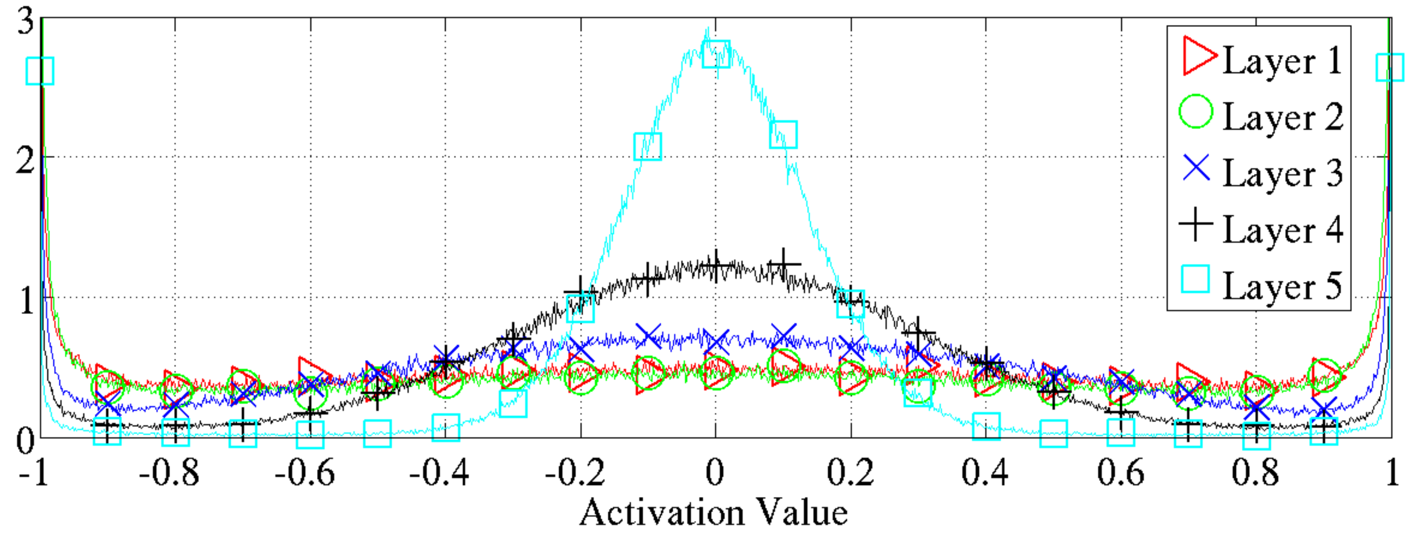
The Glorot uniform initializer, also called Xavier uniform initializer.

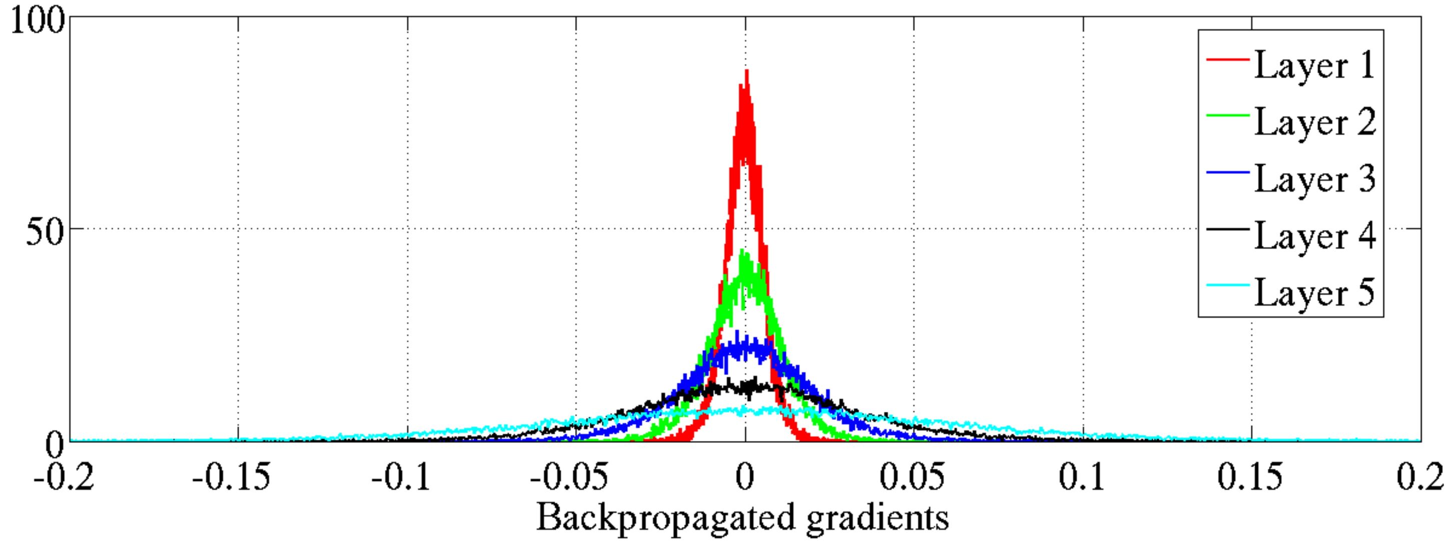
It draws samples from a uniform distribution within [-limit, limit] where `limit` is `sqrt(6 / (fan_in + fan_out))` where `fan_in` is the number of input units in the weight tensor and `fan_out` is the number of output units in the weight tensor.

Source: https://www.tensorflow.org/api_docs/python/tf/glorot_uniform_initializer

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.





Exploding 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.

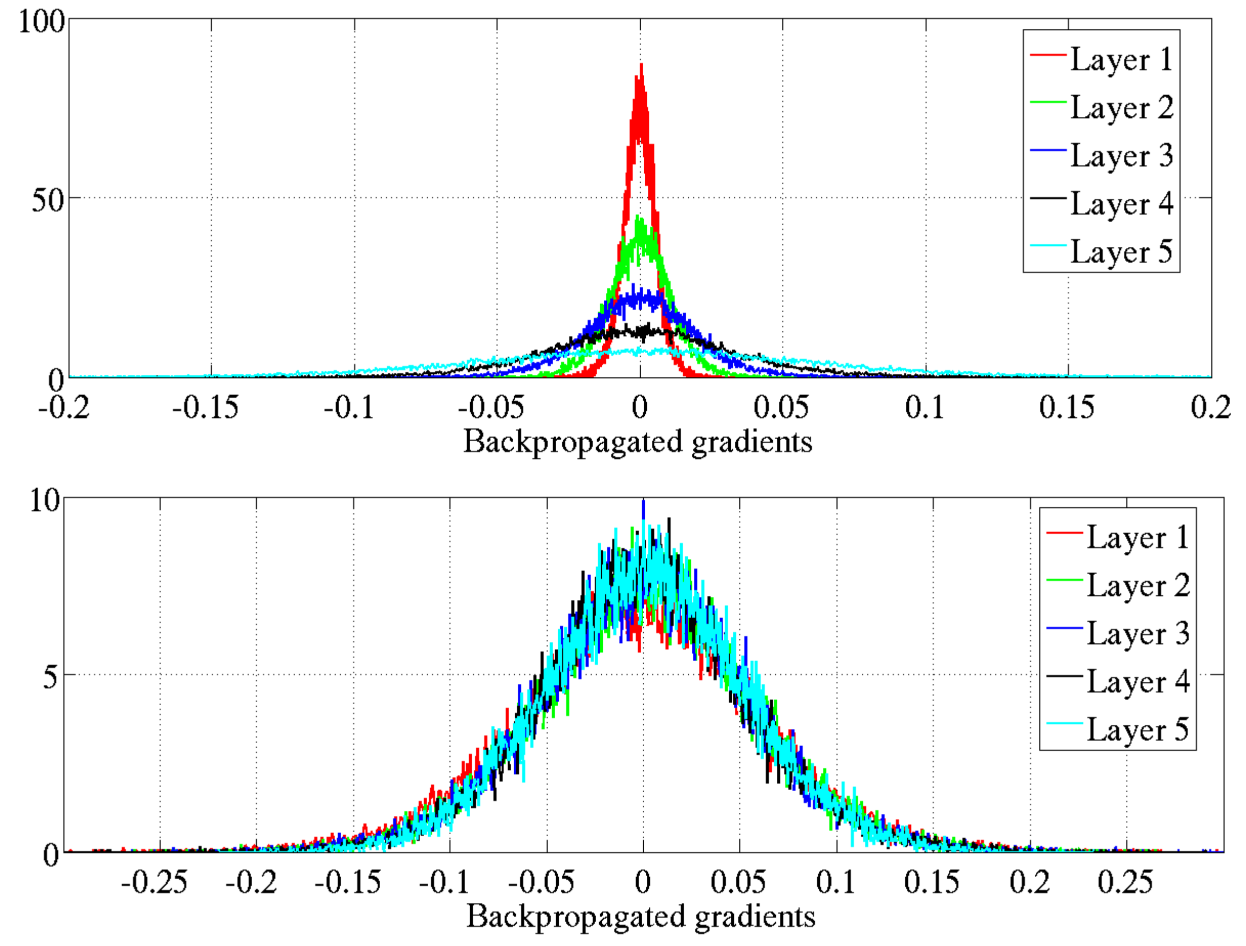


Figure 7: *Back-propagated gradients normalized histograms with hyperbolic tangent activation, with standard (top) vs normalized (bottom) initialization. Top: 0-peak decreases for higher layers.*

Weight Initialization -- He Initialization

He, Kaiming, 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 $2^{0.5}$

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

Weight Initialization -- He Initialization

He, Kaiming, 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 $2^{0.5}$

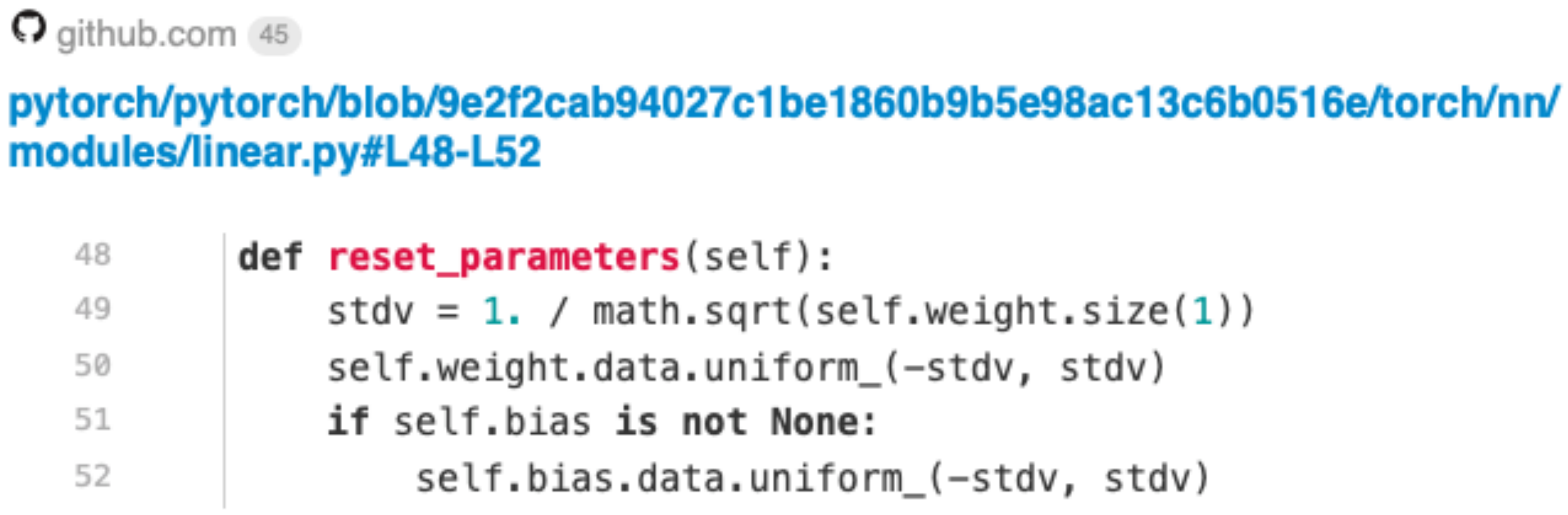
$$\mathbf{W}^{(l)} := \mathbf{W}^{(l)} \cdot \sqrt{\frac{2}{m^{[l-1]}}}$$

For Leaky ReLU with negative slope alpha:

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

PyTorch Default Weights

PyTorch uses the following scheme by default, which is somewhat similar to Xavier initialization, and works ok in practice most of the time


github.com 45

pytorch/pytorch/blob/9e2f2cab94027c1be1860b9b5e98ac13c6b0516e/torch/nn/modules/linear.py#L48-L52

```
def reset_parameters(self):
    stdv = 1. / math.sqrt(self.weight.size(1))
    self.weight.data.uniform_(-stdv, stdv)
    if self.bias is not None:
        self.bias.data.uniform_(-stdv, stdv)
```

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