STAT 453: Introduction to Deep Learning and Generative Models

Ben Lengerich

Lecture 11: Normalization / Initialization

October 8, 2025

A quick note about projects

How to decide on good model architectures before we've studied them in depth?

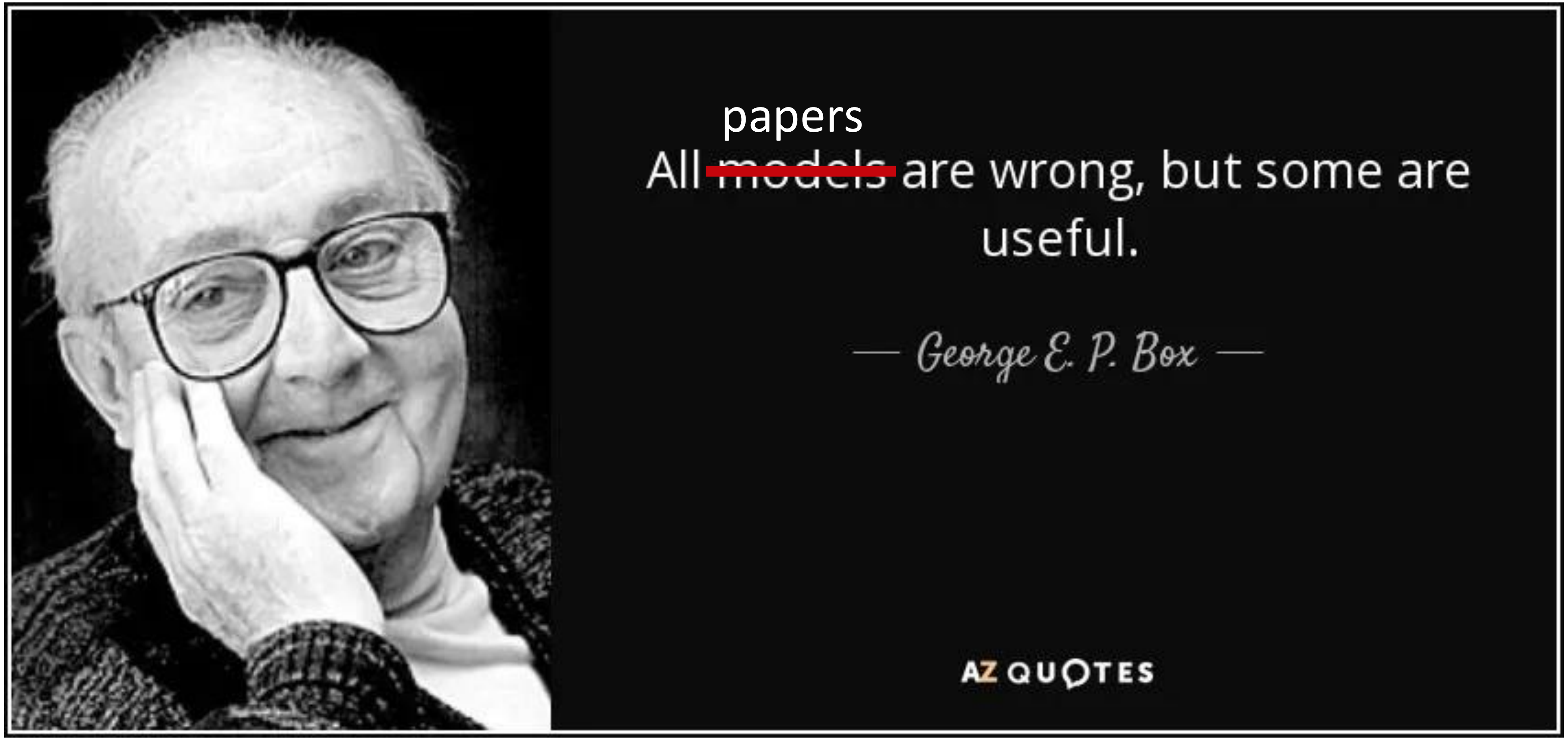
A note on research papers

How we imagine research papers:

How research papers **actually** are:

Holes big enough to drive a car through!

A note on research papers → let's be optimists.



Last Time: Regularization

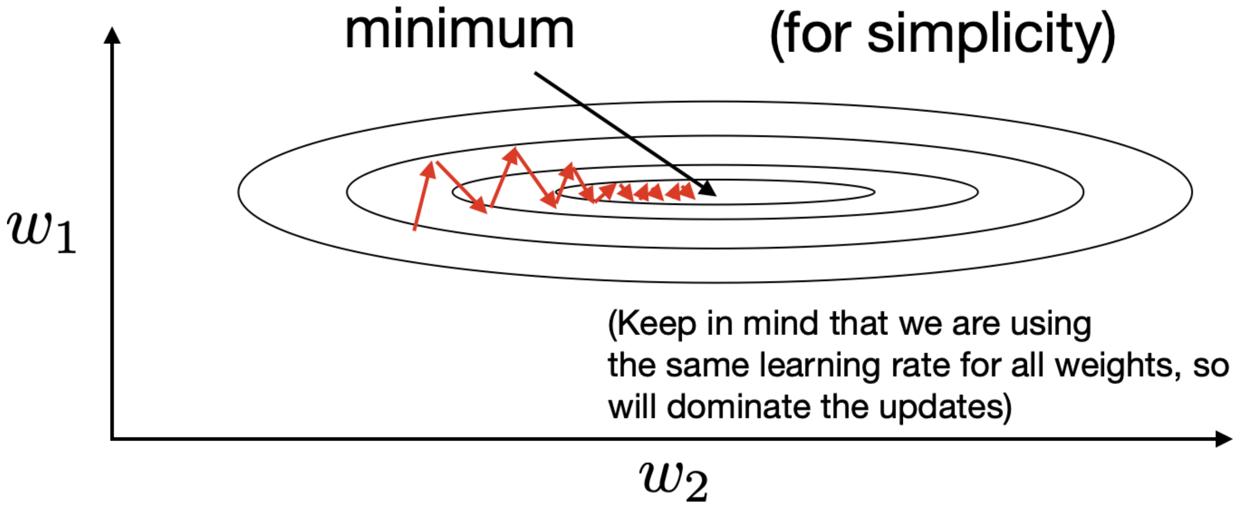
1. Improving generalization performance
2. Avoiding overfitting with (1) more data and (2) data augmentation
3. Reducing network capacity & early stopping
4. Adding norm penalties to the loss: L1 & L2 regularization
5. Dropout

Today: Feature Normalization & Weight Initialization

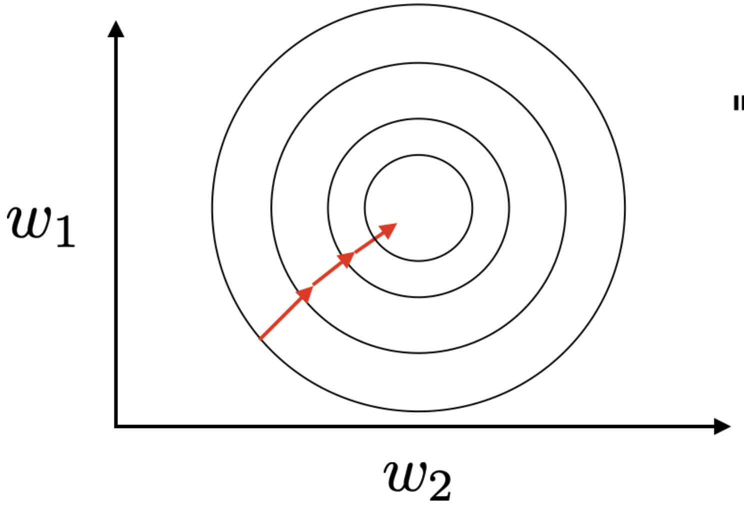
1. **Input normalization**
2. Batch normalization
3. BatchNorm in PyTorch
4. Why does BatchNorm work?
5. Weight initialization -- why do we care?
6. Xavier & He Initialization
7. Weight initialization schemes in PyTorch

Normalization and 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

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

(scaled feature will have zero mean,
unit variance)

In deep models…

Normalizing the **inputs** only affects the first hidden layer…what about the rest?

Today: Feature Normalization & Weight Initialization

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")

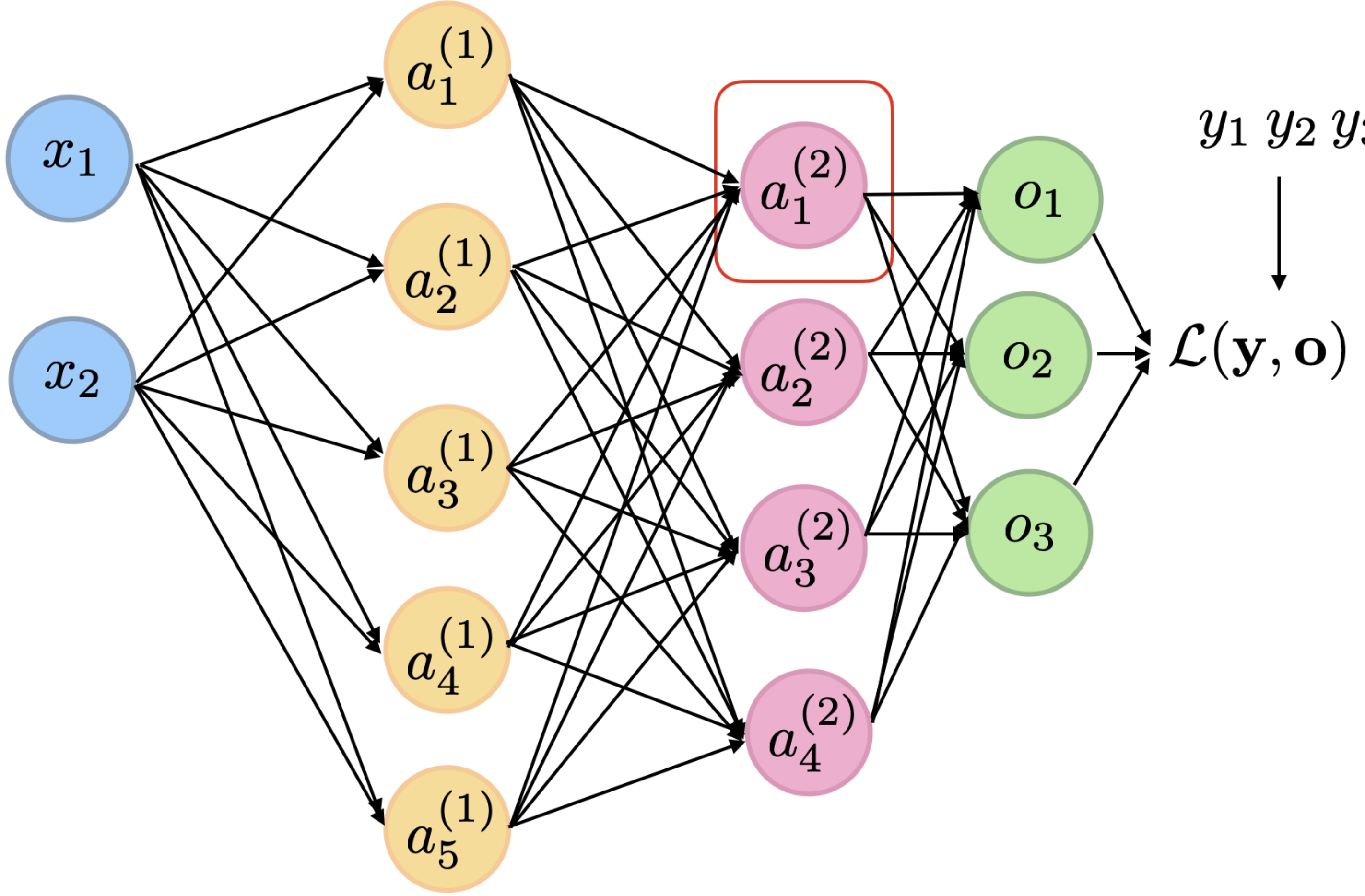
Ioffe, S., & Szegedy, C. (2015). Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. In *International Conference on Machine Learning* (pp. 448-456).

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

- 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)

Batch Normalization ("BatchNorm")

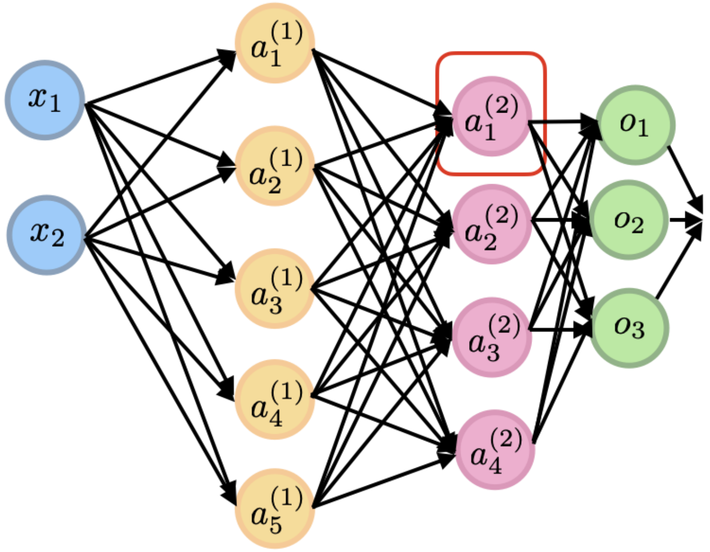
Suppose, we have net input $z_1^{(2)}$
associated with an activation in the 2nd hidden layer



Batch Normalization ("BatchNorm")

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

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

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

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

<u>In practice:</u>

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

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

BatchNorm Step 2: Pre-Activation Scaling

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

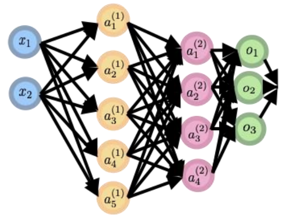
$$a'^{[i]}_j = \gamma_j \cdot z'^{[i]}_j + \beta_j$$

Controls the mean

Controls the spread or scale

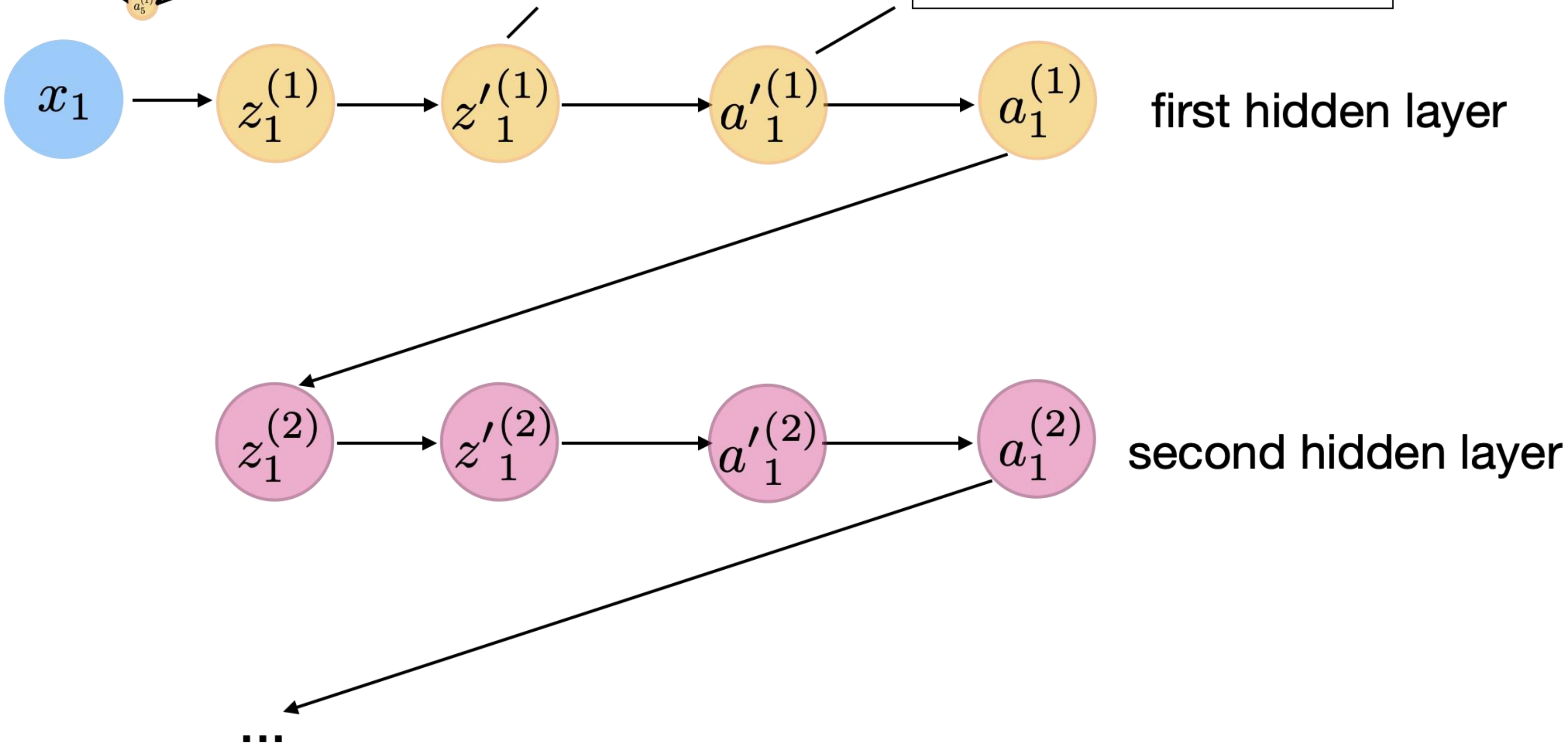
Technically, a BatchNorm layer could learn to perform "standardization" with zero mean and unit variance

BatchNorm Steps 1+2 Together



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

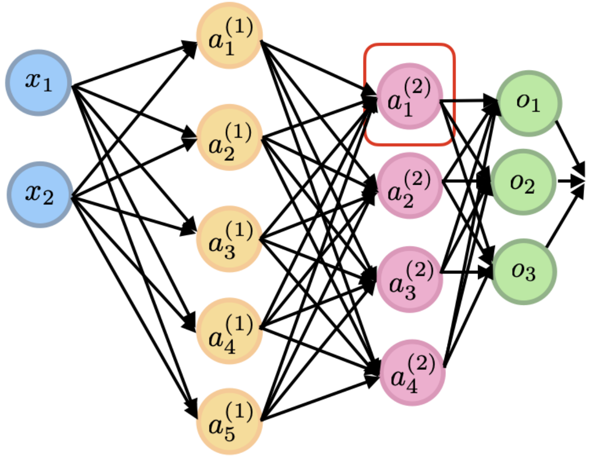
$$a'^{[i]}_j = \gamma_j \cdot z'^{[i]}_j + \beta_j$$



BatchNorm Steps 1+2 Together

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

This parameter makes the bias units redundant

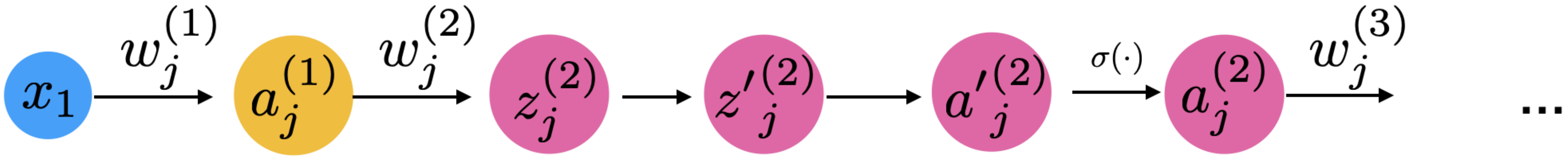


Also, note that the batchnorm parameters are vectors with the same number of elements as the bias vector

Can we learn BatchNorm params by backprop?

BatchNorm and Backprop

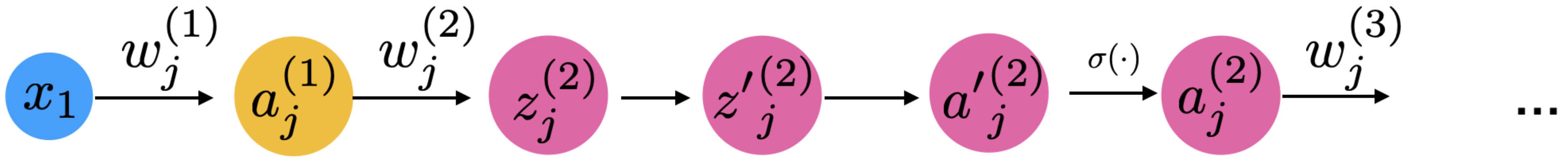
$$z'^{(2)}_j = \frac{z^{(2)}_j - \mu_j}{\sigma_j} \qquad a'^{(2)}_j = \gamma_j \cdot z'^{(2)}_j + \beta_j$$



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

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

BatchNorm and Backprop



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

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

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

BatchNorm and Backprop

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

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

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

Today: Feature Normalization & Weight Initialization

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 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-learningss21/blob/main/L11/code/batchnorm.ipynb

BatchNorm in PyTorch

```
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 at Test-Time

- 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

Today: Feature Normalization & Weight Initialization

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

Why does BatchNorm work?

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

Hmm…do we know anything about covariate *shift*?

Why does BatchNorm work?

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.

Smooth Optimization → Larger Learning Rates

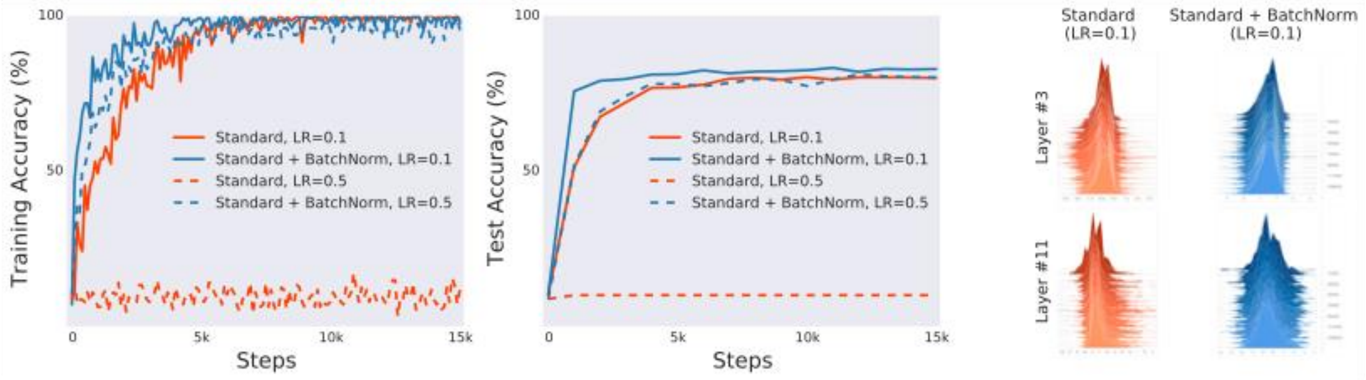


Figure 1: Comparison of (a) training (optimization) and (b) test (generalization) performance of a standard VGG network trained on CIFAR-10 with and without BatchNorm (details in Appendix A). There is a consistent gain in training speed in models with BatchNorm layers. (c) Even though the gap between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

BatchNorm benefit seems unrelated to covariate shift

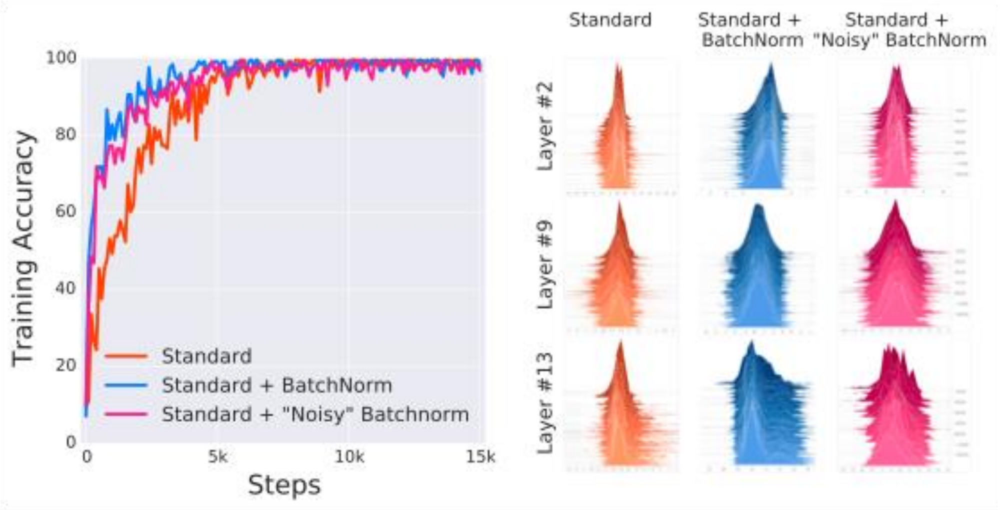


Figure 2: Connections between distributional stability and BatchNorm performance: We compare VGG networks trained without BatchNorm (Standard), with BatchNorm (Standard + BatchNorm) and with explicit "covariate shift" added to BatchNorm layers (Standard + "Noisy" BatchNorm). In the later case, we induce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

https://arxiv.org/abs/1805.11604

A note on research papers

How we imagine research papers:

How research papers **actually** are:

Holes big enough to drive a car through!

Many interpretations

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.*

Many interpretations

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.*

Should BatchNorm happen after activation?

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	

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

Practical Consideration

- BatchNorm becomes more stable with larger mini-batch sizes

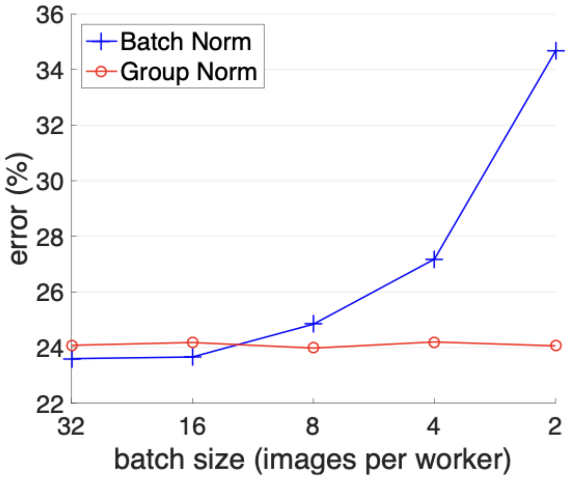
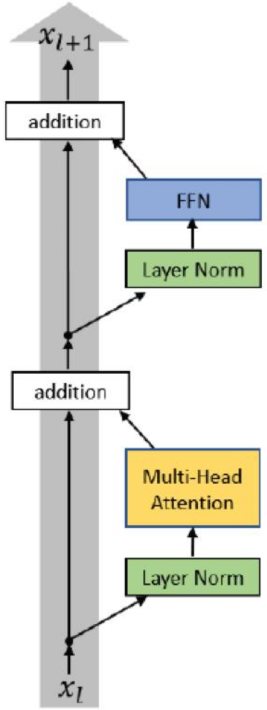


Figure 1. ImageNet classification error *vs*. batch sizes. The model is ResNet-50 trained in the ImageNet training set using 8 workers (GPUs) and evaluated in the validation set. BN's error increases rapidly when reducing the batch size. GN's computation is independent of batch sizes, and its error rate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

Related: LayerNorm

- Layer normalization (LN)
- BN calculates mean/std based on a mini batch, whereas LN calculates mean/std based on feature/embedding vectors
- In the stats language, BN zero mean unit variance, whereas LN projects feature vector to **unit sphere**
- LN in Transformers



Pre-LN Transformer

$$
\begin{aligned}
x_{l,i}^{pre,1} &= \text{LayerNorm}(x_{l,i}^{pre}) \\
x_{l,i}^{pre,2} &= \text{MultiHeadAtt}(x_{l,i}^{pre,1}, [x_{l,1}^{pre,1}, \cdots, x_{l,n}^{pre,1}]) \\
x_{l,i}^{pre,3} &= x_{l,i}^{pre} + x_{l,i}^{pre,2} \\
x_{l,i}^{pre,4} &= \text{LayerNorm}(x_{l,i}^{pre,3}) \\
x_{l,i}^{pre,5} &= \text{ReLU}(x_{l,i}^{pre,4} W^{1,l} + b^{1,l}) W^{2,l} + b^{2,l} \\
x_{l+1,i}^{pre} &= x_{l,i}^{pre,5} + x_{l,i}^{pre,3}
\end{aligned}
$$

Final LayerNorm: $x_{Final,i}^{pre} \leftarrow \text{LayerNorm}(x_{L+1,i}^{pre})$

Normalize everything?

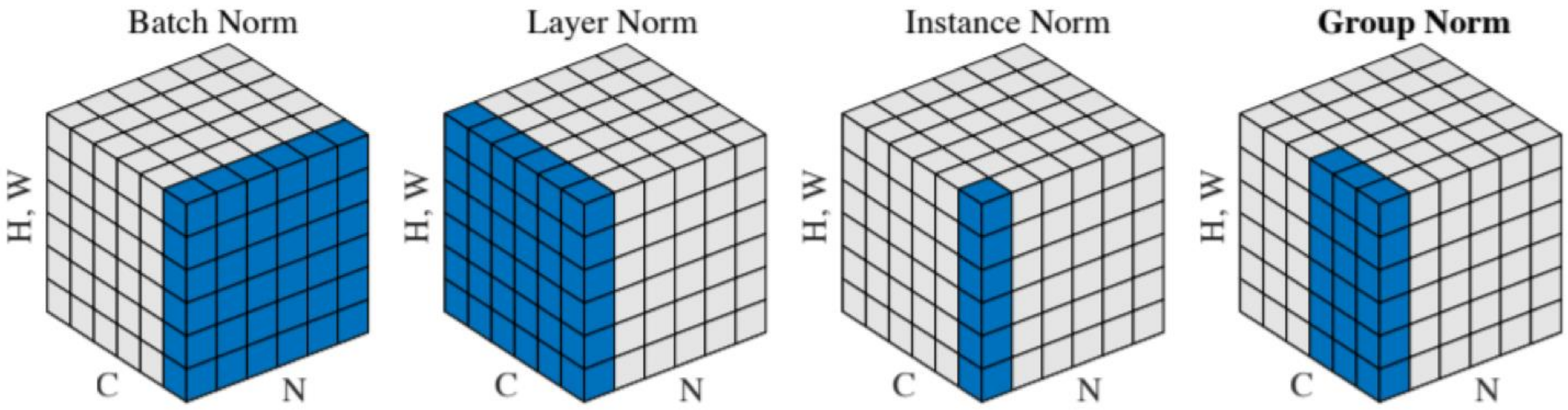


Figure 2. **Normalization methods**. Each subplot shows a feature map tensor, with N as the batch axis, C as the channel axis, and (H, W) as the spatial axes. The pixels in blue are normalized by the same mean and variance, computed by aggregating the values of these pixels.

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

Today: Feature Normalization & Weight Initialization

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

- Recall: Can’t initialize all weights to 0 (symmetry problem)
- But we want weights to be relatively small.
  - 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)

Today: Feature Normalization & Weight Initialization

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

Xavier Initialization

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.)

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.

Xavier Initialization

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.)

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

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)

Xavier Initialization

<u>Rationale behind this scaling:</u>
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

$$\operatorname{Var}\left(z_j^{(l)}\right) = \operatorname{Var}\left(\sum_{j=1}^{m_{l-1}} W_{jk}^{(l)} a_k^{(l-1)}\right)$$

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

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

He Initialization

- 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, the **activations are not centered at zero**
- He initialization takes this into account
- The result is that we add a scaling factor of $\sqrt{2}$

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

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.

Initialization

- When neural network models change, proper initialization schemes may change; we will see this in transformers
- Research frontier: training very deep neural networks requires control of the weight matrix spectrum
  - *Random matrix theory suggests a sophisticated initialization for training 10,000 layer network,* https://arxiv.org/abs/1806.05393

Today: Feature Normalization & Weight Initialization

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 in PyTorch

PyTorch (now) uses the Kaiming He scheme by default

```
    def __init__(self, in_features: int, out_features: int, bias: bool = True) -> None:
        super(Linear, self).__init__()
        self.in_features = in_features
        self.out_features = out_features
        self.weight = Parameter(torch.Tensor(out_features, in_features))
        if bias:
            self.bias = Parameter(torch.Tensor(out_features))
        else:
            self.register_parameter('bias', None)
        self.reset_parameters()

    def reset_parameters(self) -> None:
        init.kaiming_uniform_(self.weight, a=math.sqrt(5))
        if self.bias is not None:
            fan_in, _ = init._calculate_fan_in_and_fan_out(self.weight)
            bound = 1 / math.sqrt(fan_in)
            init.uniform_(self.bias, -bound, bound)
```

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

Questions?

WISCONSIN
UNIVERSITY OF WISCONSIN–MADISON