Optimization

Deep Learning

Brad Quinton, Scott Chin

Learning Objectives

Introduce a number of commonly used techniques to improve the training optimization process

- Learning Rate Schedules
- Parameter Initialization
- Data Preprocessing/Feature Normalization
- Batch Norm

Learning Rate Schedules

Last time: Optimizers (i.e. Gradient Descent)

- Talked about various flavours of Gradient Descent
- They all have a learning rate hyperparameter

Learning Rate – Common Patterns

- Too Low
- Too High
- Crazy High
- Just Right

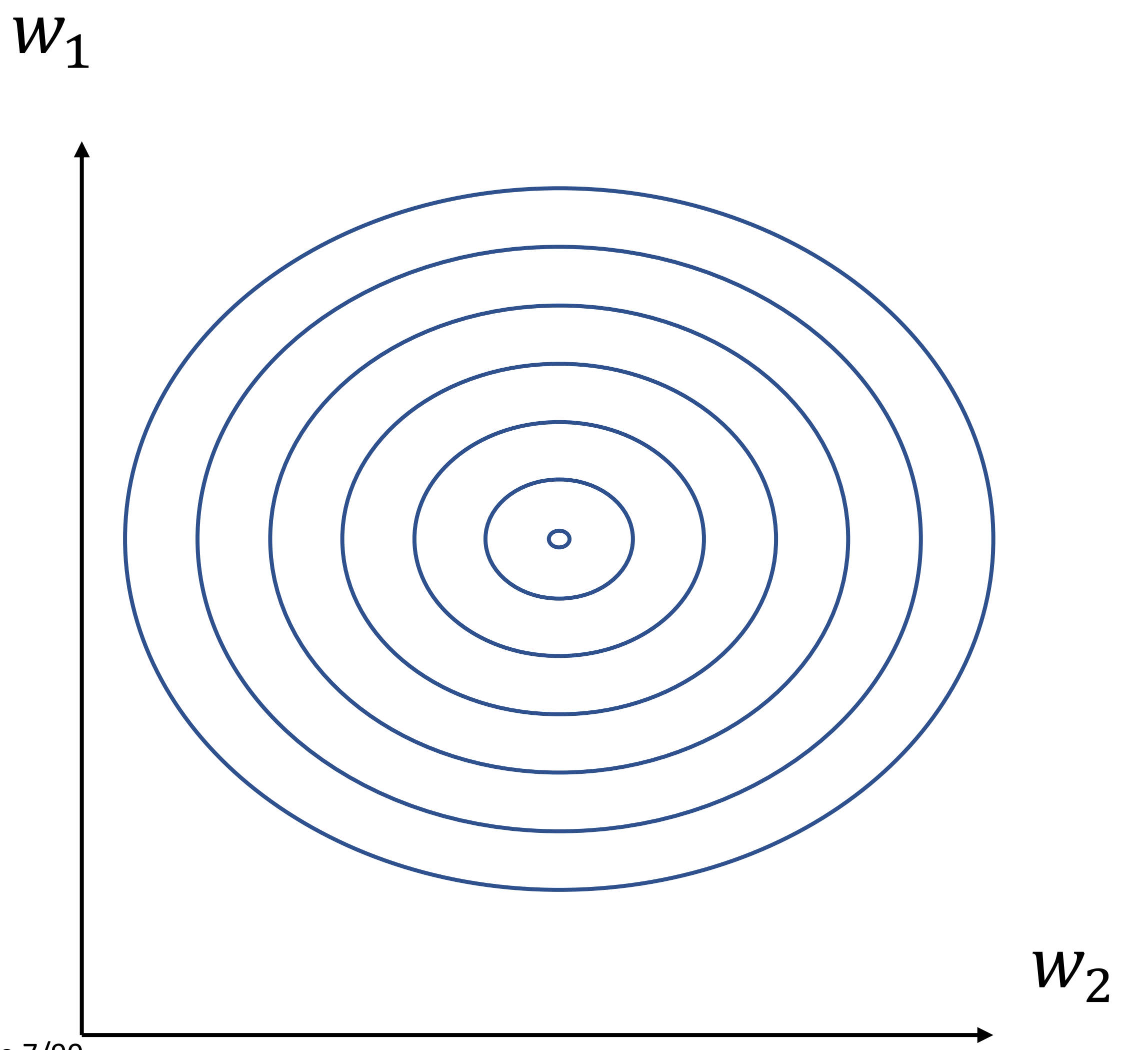
How To Choose Learning Rate

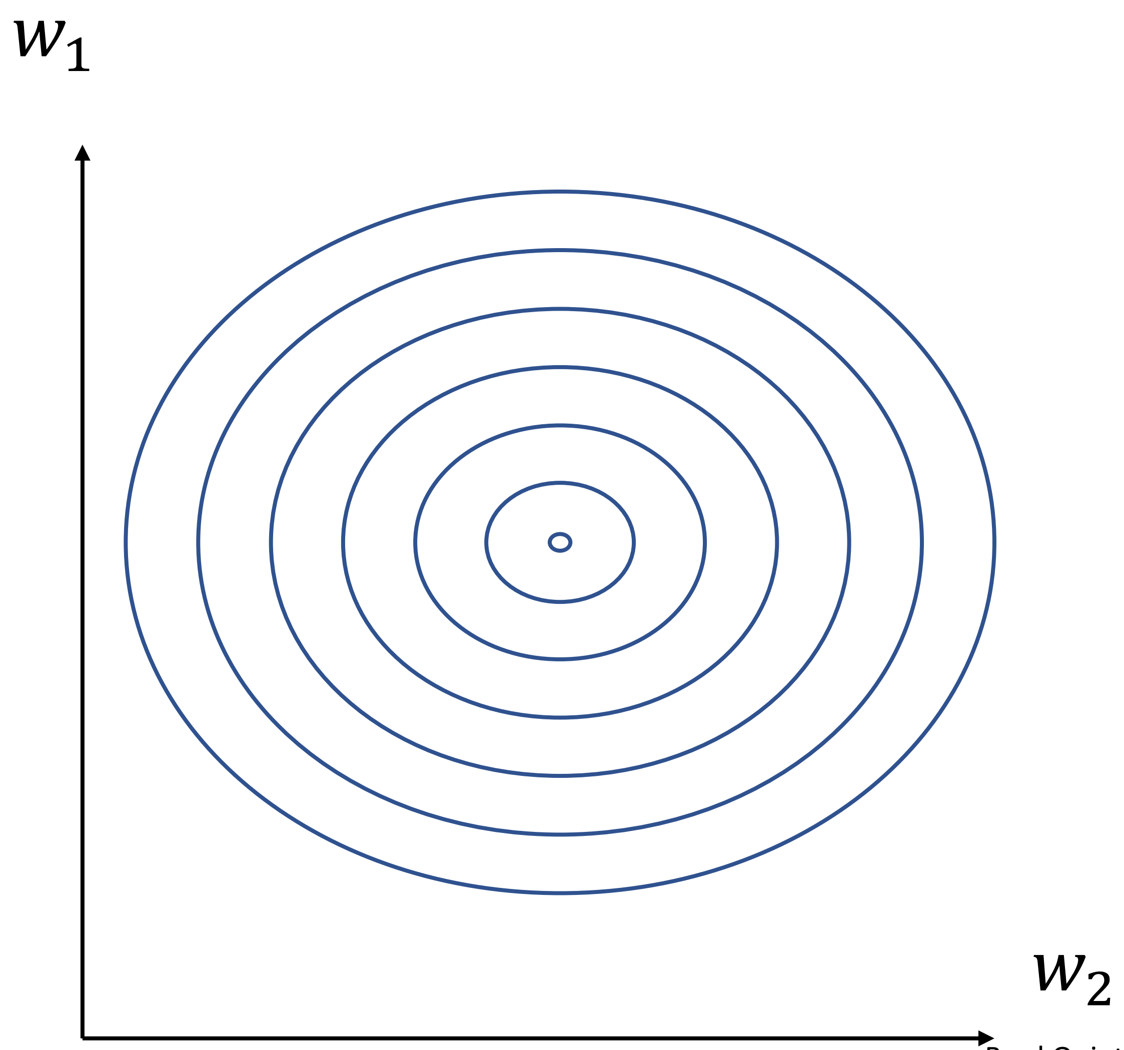
- So far, we pick a fixed learning rate that is used throughout training

Instead, vary learning rate over time

- Specifically, start somewhat high and reduce over time.
- Sometimes called annealing the learning rate
- Sometimes called decaying the learning rates

Why Decay Learning Rate





How to Decay?

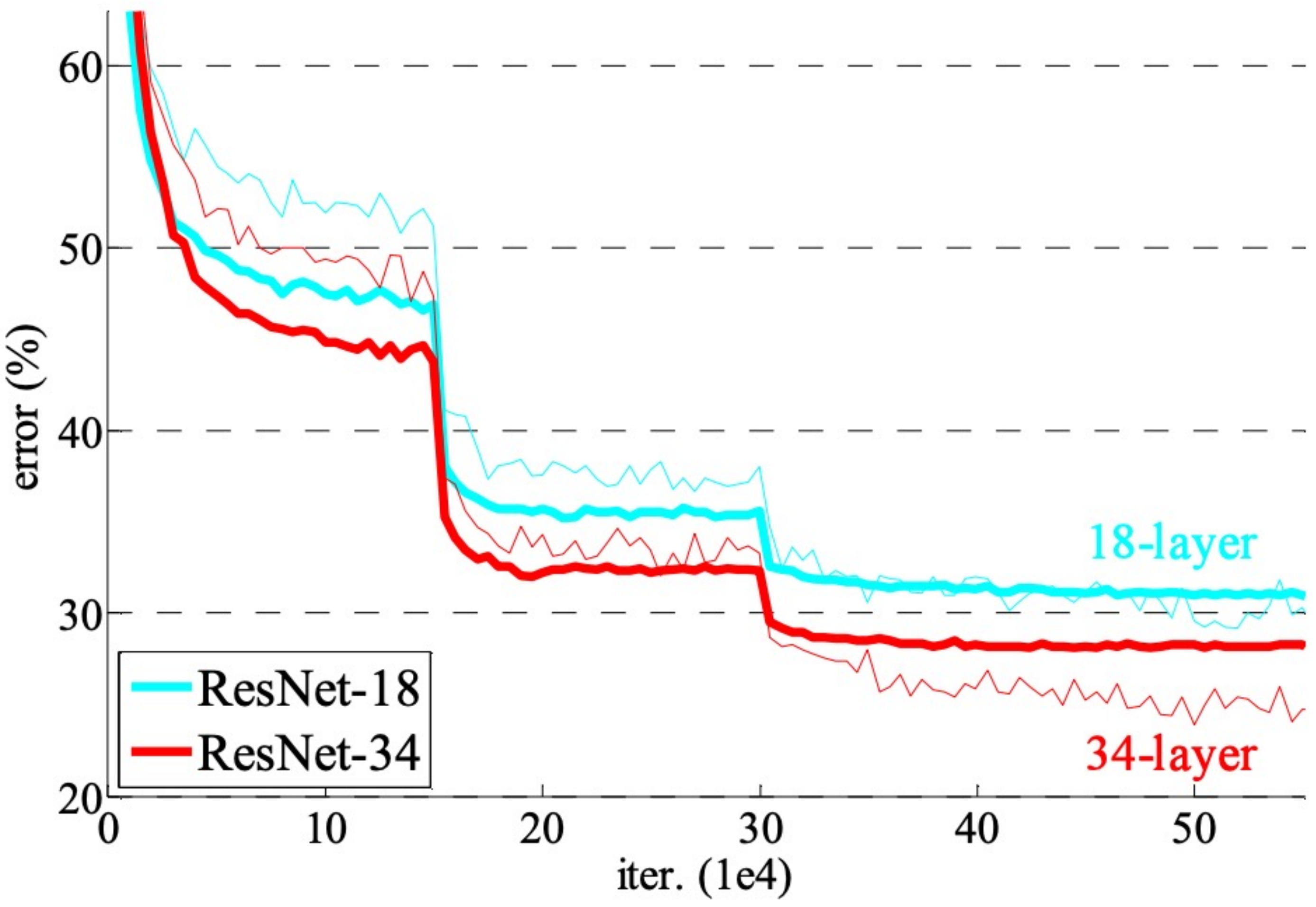
- The method in which you decay/anneal the learning rate is sometimes referred to as the Decay Schedule, or Annealing Schedule
- Generally, want to reduce learning rate once progress plateaus

Competing Tradeoffs

- Decay too slow → wasting time bounding around
- Decay too fast → slow down your training unneccesarily

Let’s look at some common decay schedules

Step Decay



- Reduce learning rate at fixed points
- E.g in ResNet Paper, starts at 0.1, divide by 10 when plateau. 600,000 iterations
- New Hyperparameters
  - Which intervals to decay
  - How to decay at each interval

"Deep Residual Learning for Image Recognition", He, Zhang, Ren, Sun, 2015, https://arxiv.org/pdf/1512.03385.pdf

Decay Based On Some Function

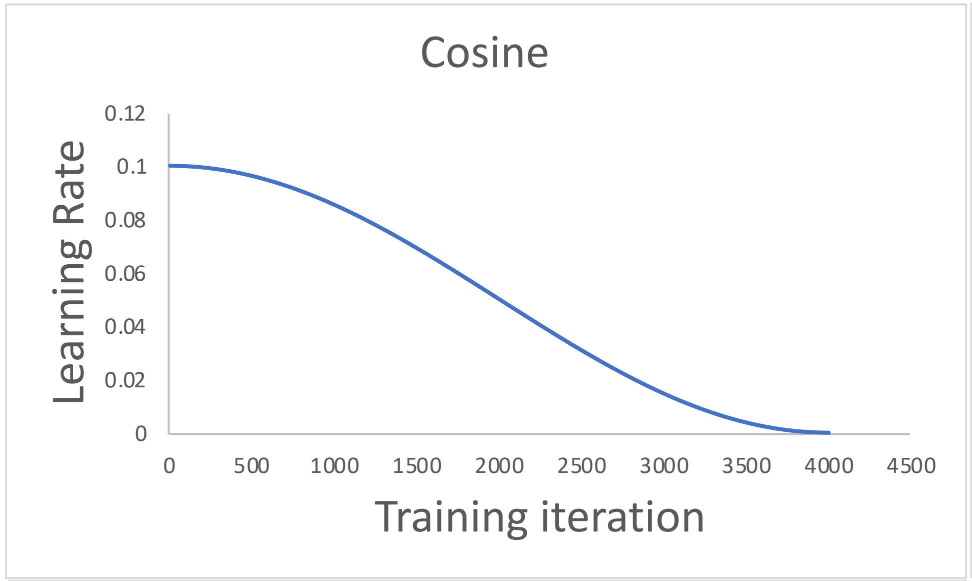
- Step decay requires you to choose when to decay, and by how much each time.
- Instead, we can decay based on some function.
  - Typically no new hyperparameters
  - i.e. only need initial learning rate and number of training epochs

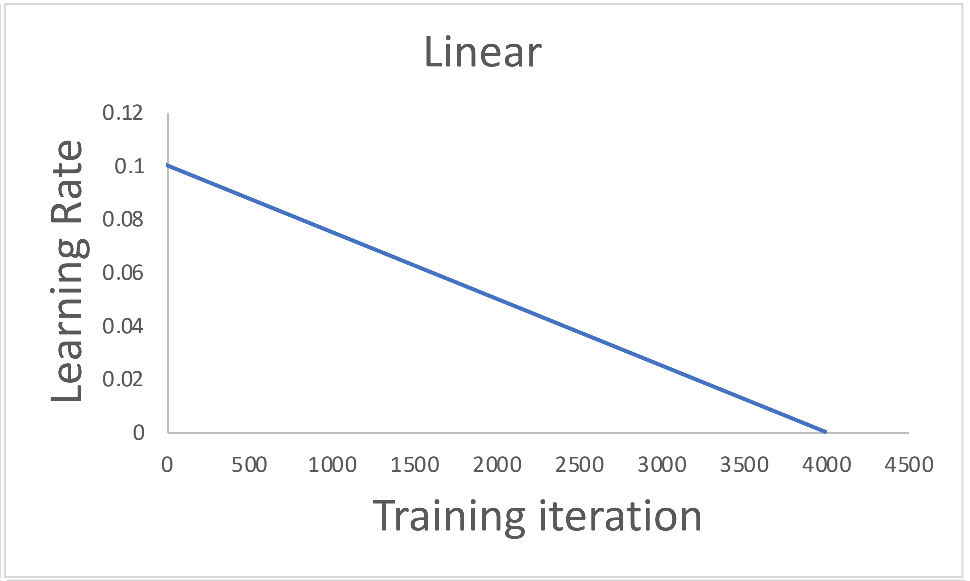
Decay Based On Some Function

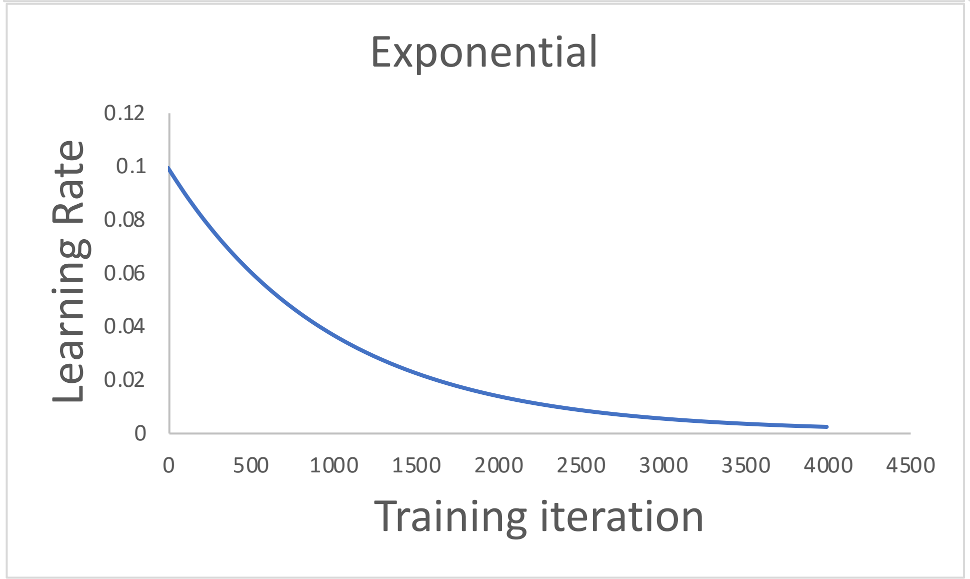
- Exponential Decay $\alpha_t = \alpha_0 0.95^t \qquad \alpha_t = \alpha_0 e^{-kt}$
- Linear Decay $\alpha_t = \alpha_0 \left(1 - \frac{t}{T}\right)$
- Cosine Decay $\alpha_t = \alpha_0 0.5 \left(1 + \cos\left(\frac{\pi t}{T}\right)\right)$
- Inverse sqrt Decay $\alpha_t = \alpha_0 \frac{1}{\sqrt{t}}$
- 1/t Decay $\alpha_t = \alpha_0 \frac{1}{1 + k * t}$

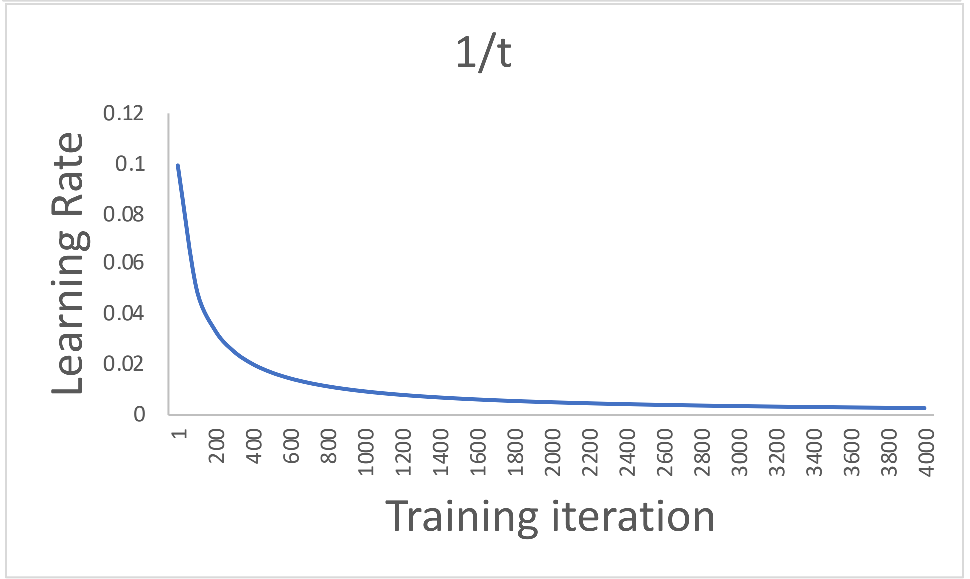
- t – current training iteration
- T – Total training iterations
- α_0– Initial learning rate
- α_t - learning rate at iteration t
- k – a hyperaparameter

Decay Based On Some Function





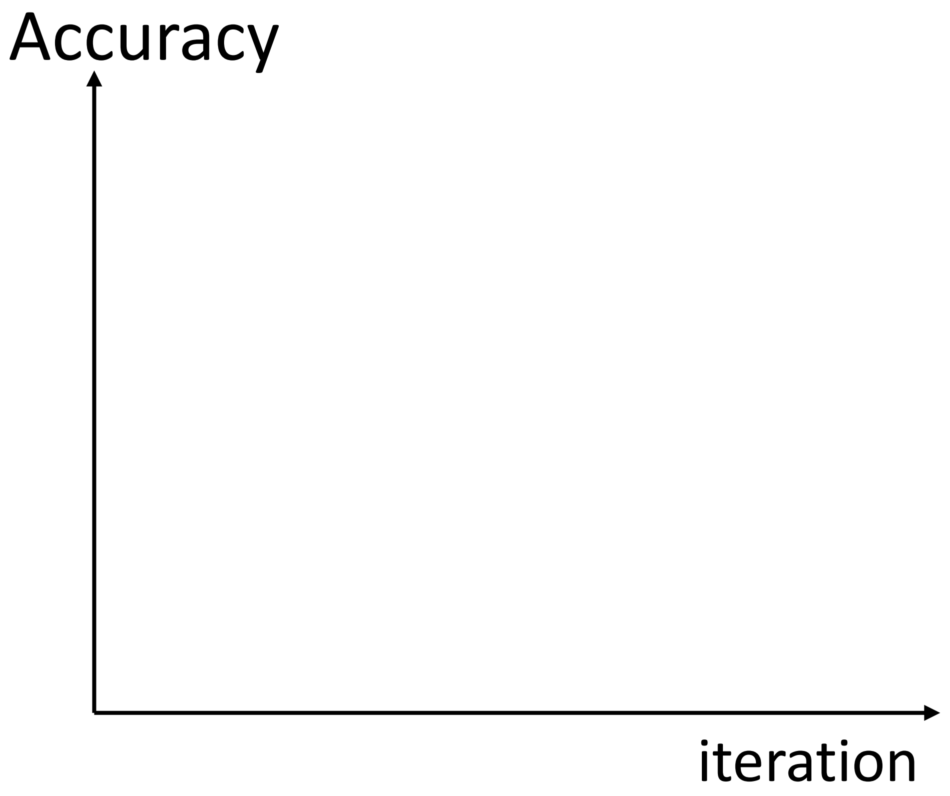




What to Choose?

- Try a constant learning rate first (no decay)
- I find step decay easier to interpret – manually decay after progress plateaus
- Some people prefer function decay schedules due to no new hyperparameters

How Long To Train?



Parameter Initialization

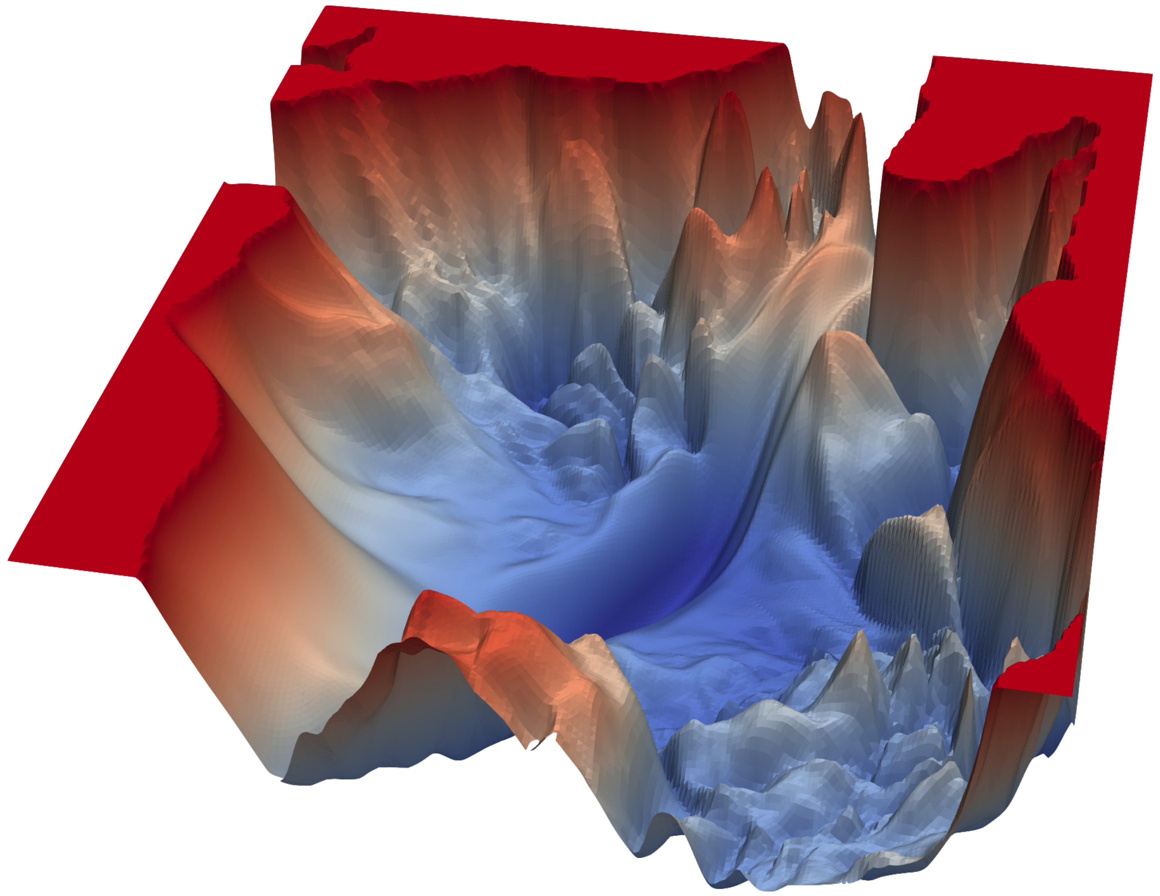
Optimization: Need to start somewhere

```
w = initialize()
for i in range(num_iterations):
  dJ_dw = compute_gradients(train_data, cost_func, w)
  w = w - learning_rate*dJ_dw
```

- Does it matter how we initialize our parameters?
- Can’t our optimizer just eventually find its way?

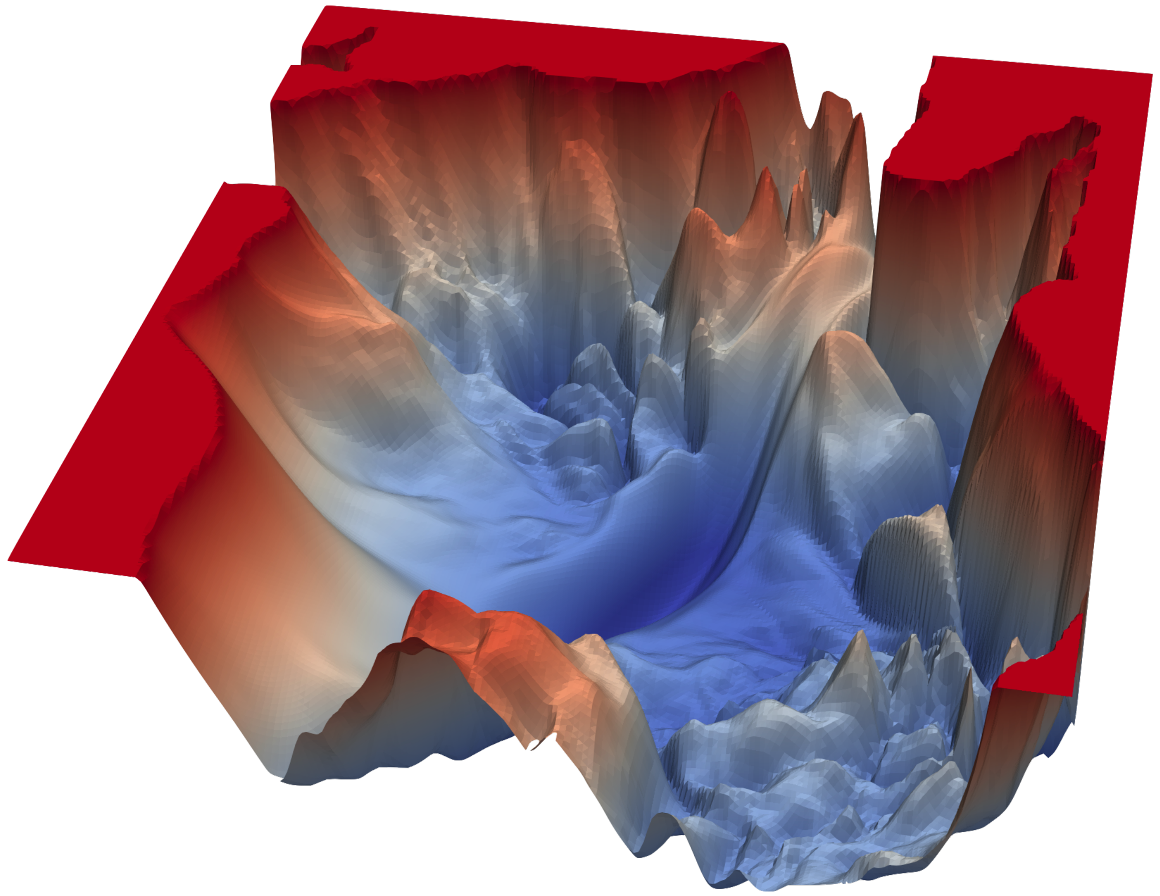
How can we initialize the parameters?

- How about we try to start close to a global minima?

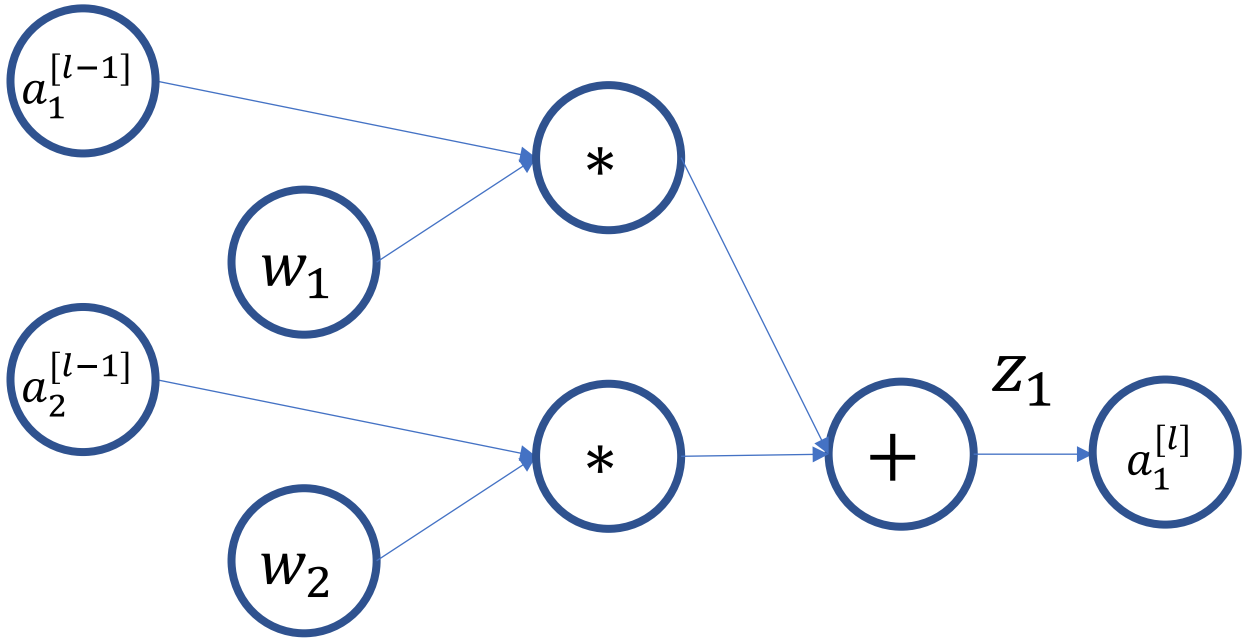


How can we initialize the parameters?

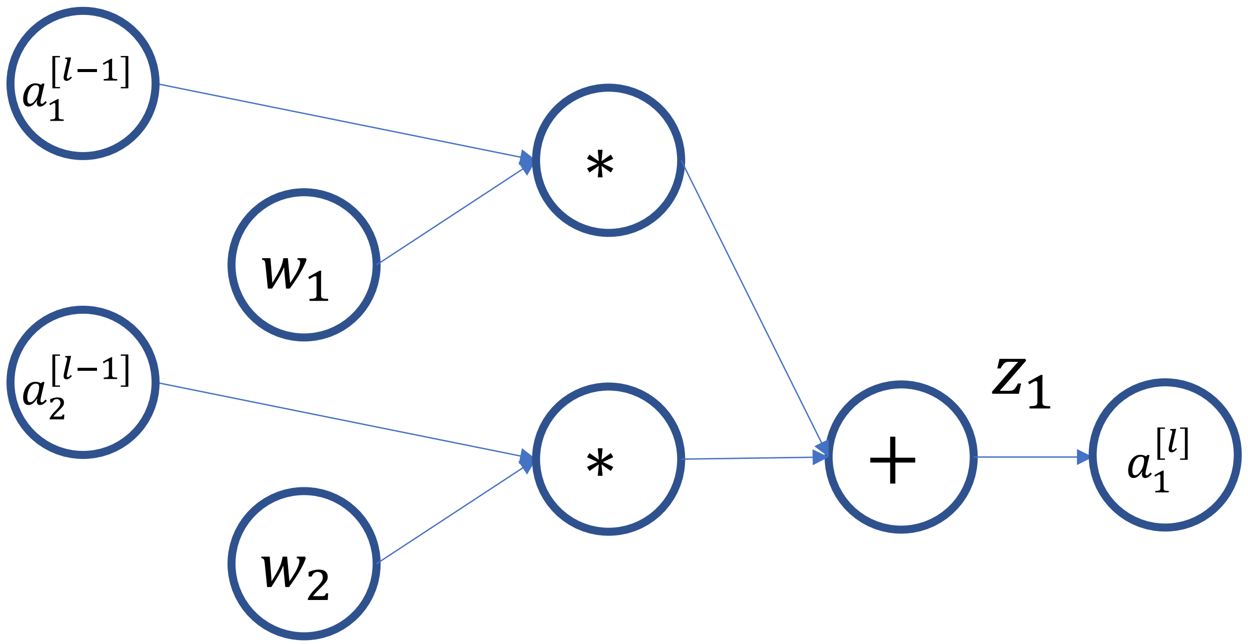
- How about we try to start close to a global minima? Not clear how to.
- Want gradients to be well-behaved. I.e don't want to start off in a place where gradients are all 0
- Re: objective landscape. Don't want to start someplace that is very flat. I.e. not clear which way to move



Initializing with 0’s



Initializing with a Constant



Initializing with a Gaussian Random Dist

```
W = 0.01*np.random.randn(fan_in, fan_out)
```

- Breaks symmetry (i.e. not all initialized to same value)
- Why Gaussian with mean 0?

Initializing with a Gaussian Random Dist

```
W = 0.01*np.random.randn(fan_in, fan_out)
```

- Breaks symmetry (i.e. not all initialized to same value)
- Why Gaussian with mean 0? With zero-centered inputs, we can also expect that the final weights might be zero-centered.
- Multiplying a Guassian random variable by x will give the random variable a standard deviations equal to x

Initializing with a Gaussian Random Dist

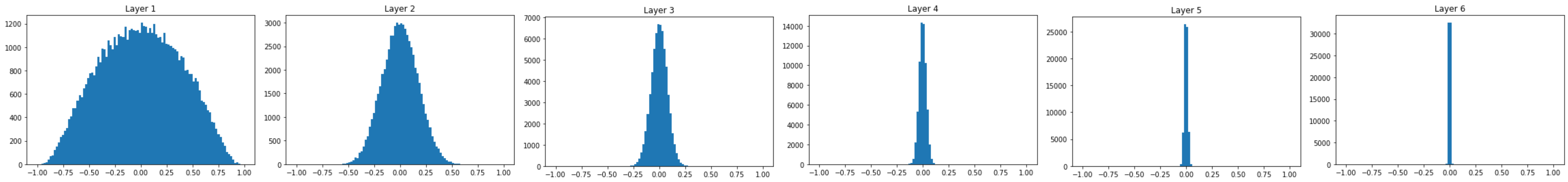
```
W = 0.01*np.random.randn(fan_in, fan_out)
```

- Breaks symmetry (i.e. not all initialized to same value)
- Why Gaussian with mean 0? With zero-centered inputs, we can also expect that the final weights might be zero-centered.
- Multiplying a Guassian random variable by x will give the random variable a standard deviations equal to x
- Ok for shallow networks, but for deeper networks….

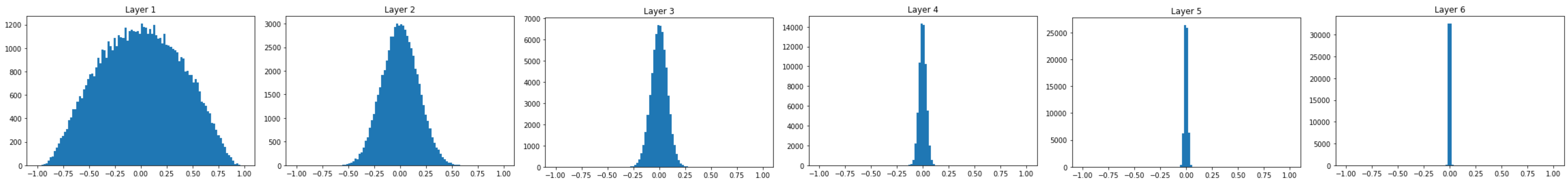
Initializing with a Gaussian Random Dist

- Consider a 6 layer fully connected network with 2048 units per layer whos parameters are initialized with random Gaussian distribution

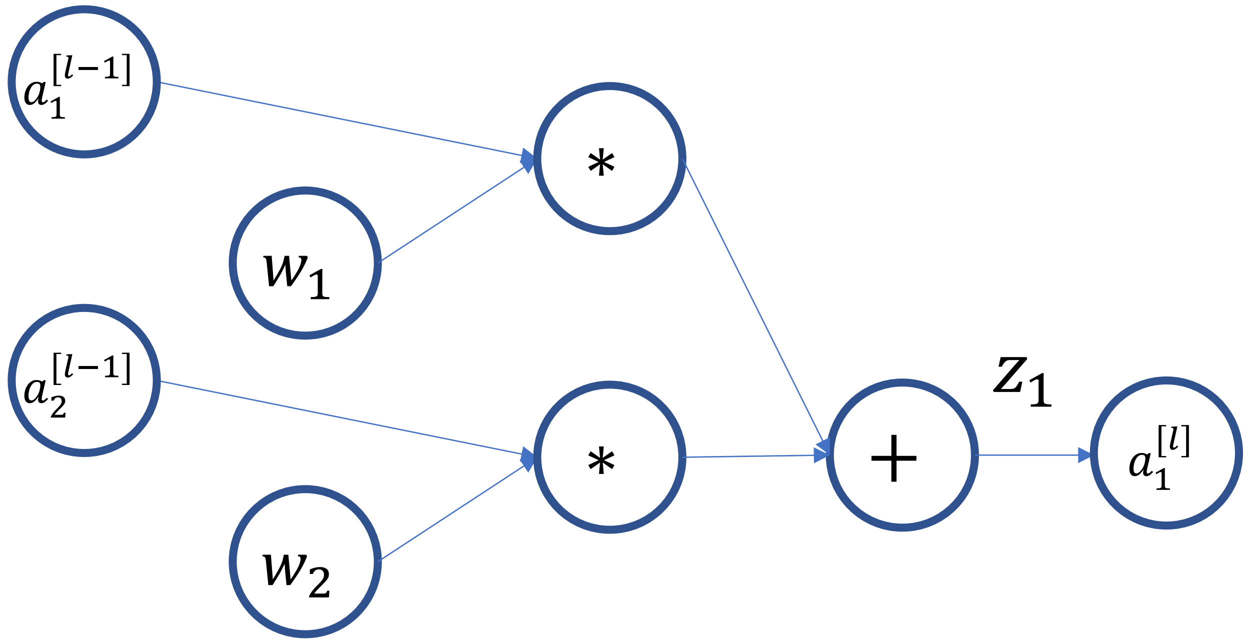
```
W = 0.01*np.random.randn(fan_in, fan_out)
```



- Activations get closer and closer to 0



- As you go deeper, activations of the units become 0 or close to 0
- This means the gradients also approach 0

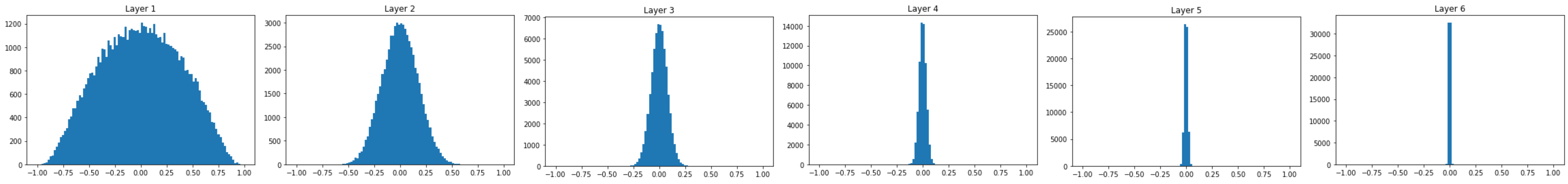


Initializing with a Gaussian Random Dist

Was 0.01 before

```
W = 0.1*np.random.randn(fan_in, fan_out)
```

- Ok let’s try a larger stdev perhaps?

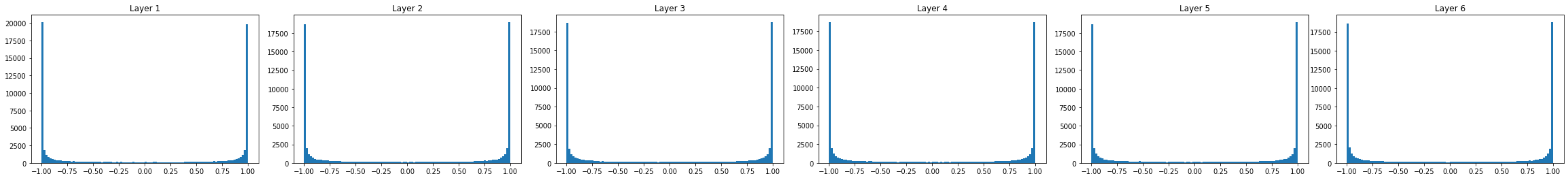


Initializing with a Gaussian Random Dist

Was 0.01 before

```
W = 0.1*np.random.randn(fan_in, fan_out)
```

- Ok let’s try a larger stdev perhaps?
- For tanh activation function, most activations are now in saturation

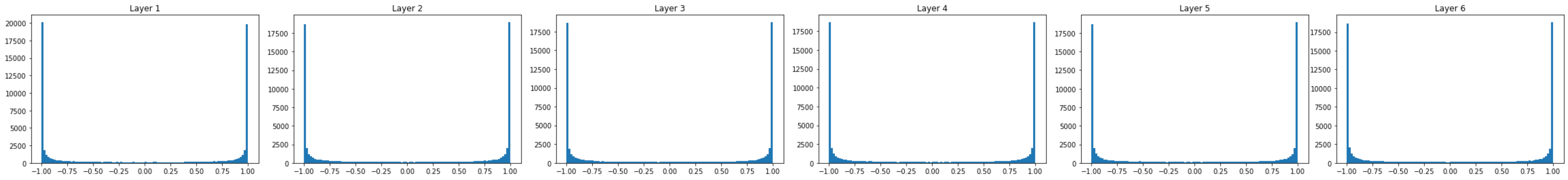


Initializing with a Gaussian Random Dist

Was 0.01 before

```
W = 0.1*np.random.randn(fan_in, fan_out)
```

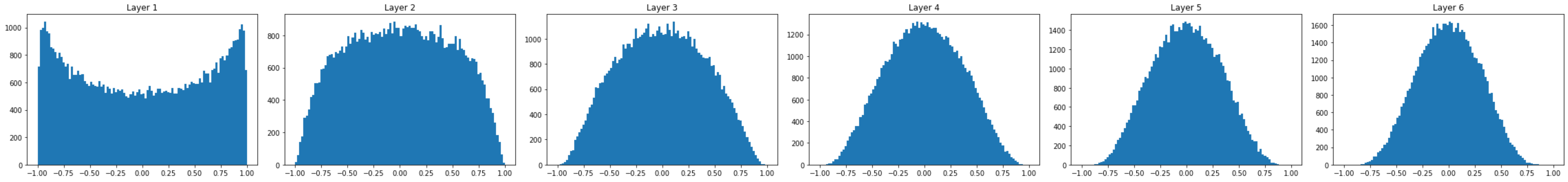
- Ok let’s try a larger stdev perhaps?
- For tanh activation function, most activations are now in saturation
- So there is an ideal variance somewhere
- What is that number?



Xavier Initialization (for TanH)

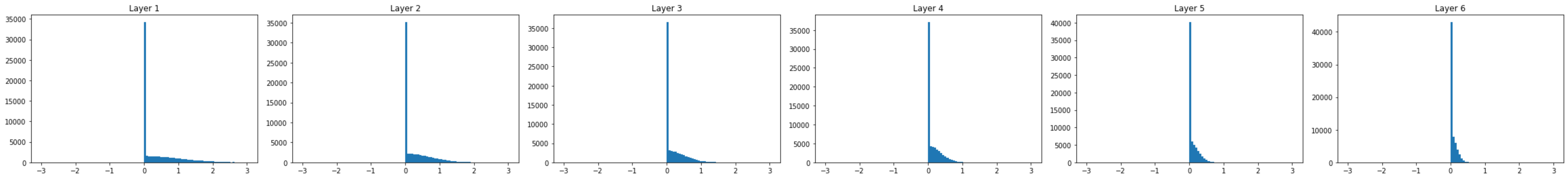
```
W = (1/np.sqrt(fan_in))*np.random.randn(fan_in, fan_out)`
```

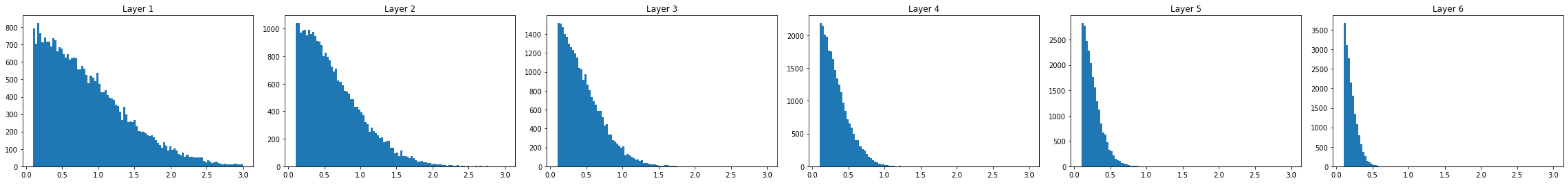
- Set variance equal to the number of inputs to the layer (fan_in)



Xavier for ReLU

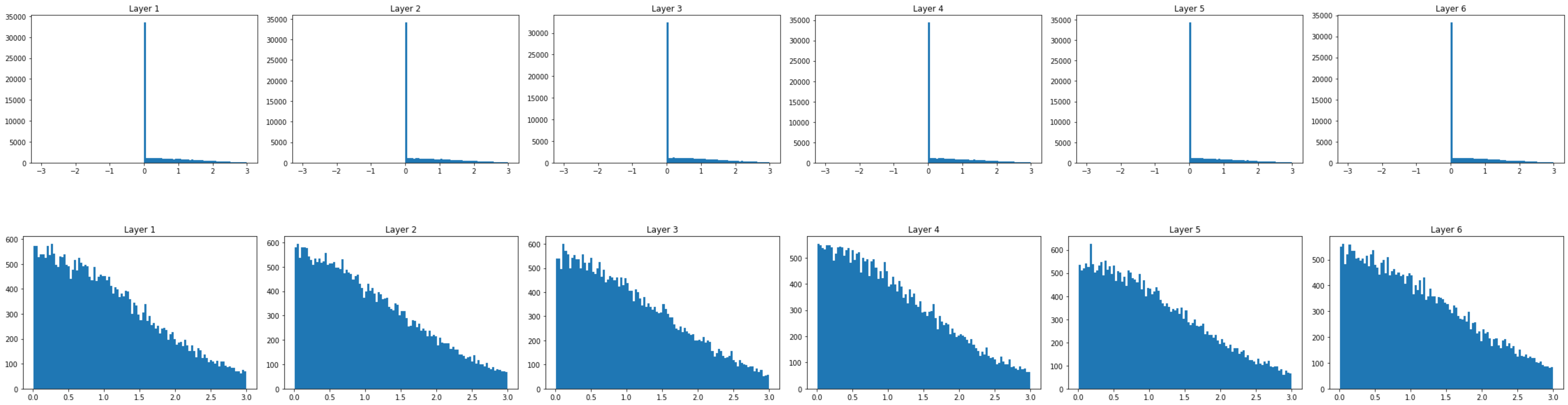
```
W = (1/np.sqrt(fan_in))*np.random.randn(fan_in, fan_out)
```





Kaiming/he_normal

```
W = (2/np.sqrt(fan_in))*np.random.randn(fan_in, fan_out)
```



Brad Quinton, Scott Chin

Initializing Bias

- Can simply initialize with 0s
- Symmetry breaking is performed in initializing the weight parameters
- Some work mentions initializing bias with small positive number (e.g. 0.01) when using ReLU activations to try and help the units start off in the active region. But verdict is not in on this yet.

Gaussian vs Uniform Distribution

- You will see versions of the aforementioned initialization schemes that draw from a uniform distribution instead of Gaussian.
- In those cases, need a different variance
- Not clear that using uniform is necessarily better

Ok what should I use?!

For now, good place to start is to use

- Kaiming/He_normal initialization for weight parameters (assuming you are using ReLU activations
- 0s for biases

Initializers in Keras

```
Dense(64,
      kernel_initializer=‘he_normal’,
      bias_initializer='zeros'))
```

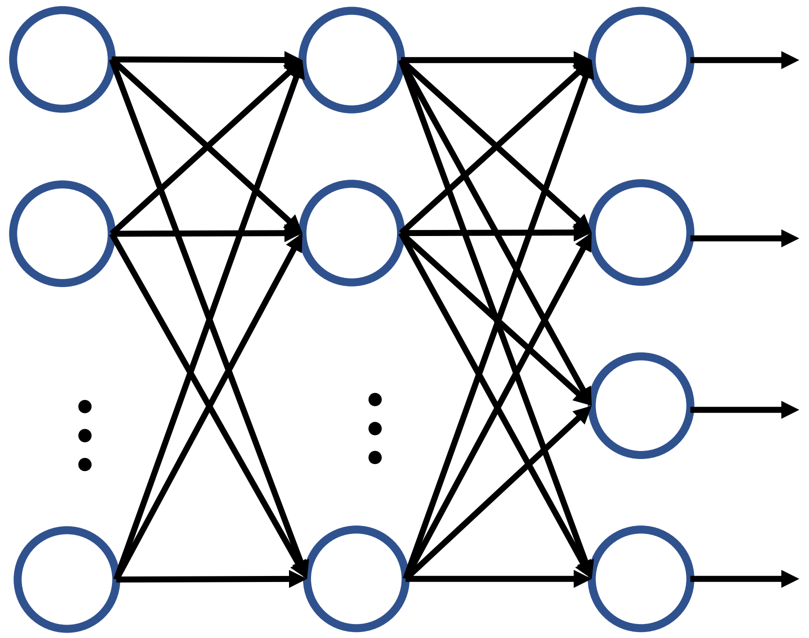
Further Readings

- “Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification”, Kaiming He et al, 2015 https://arxiv.org/abs/1502.01852
- “Understanding the difficulty of training deep feedforward neural networks”, Glorot and Bengio,2010, http://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf
- Keras Initializer source code (older version but easier to read) https://github.com/keras-team/keras/blob/998efc04eefa0c14057c1fa87cab71df5b24bf7e/keras/initializations.py

Data Preprocessing

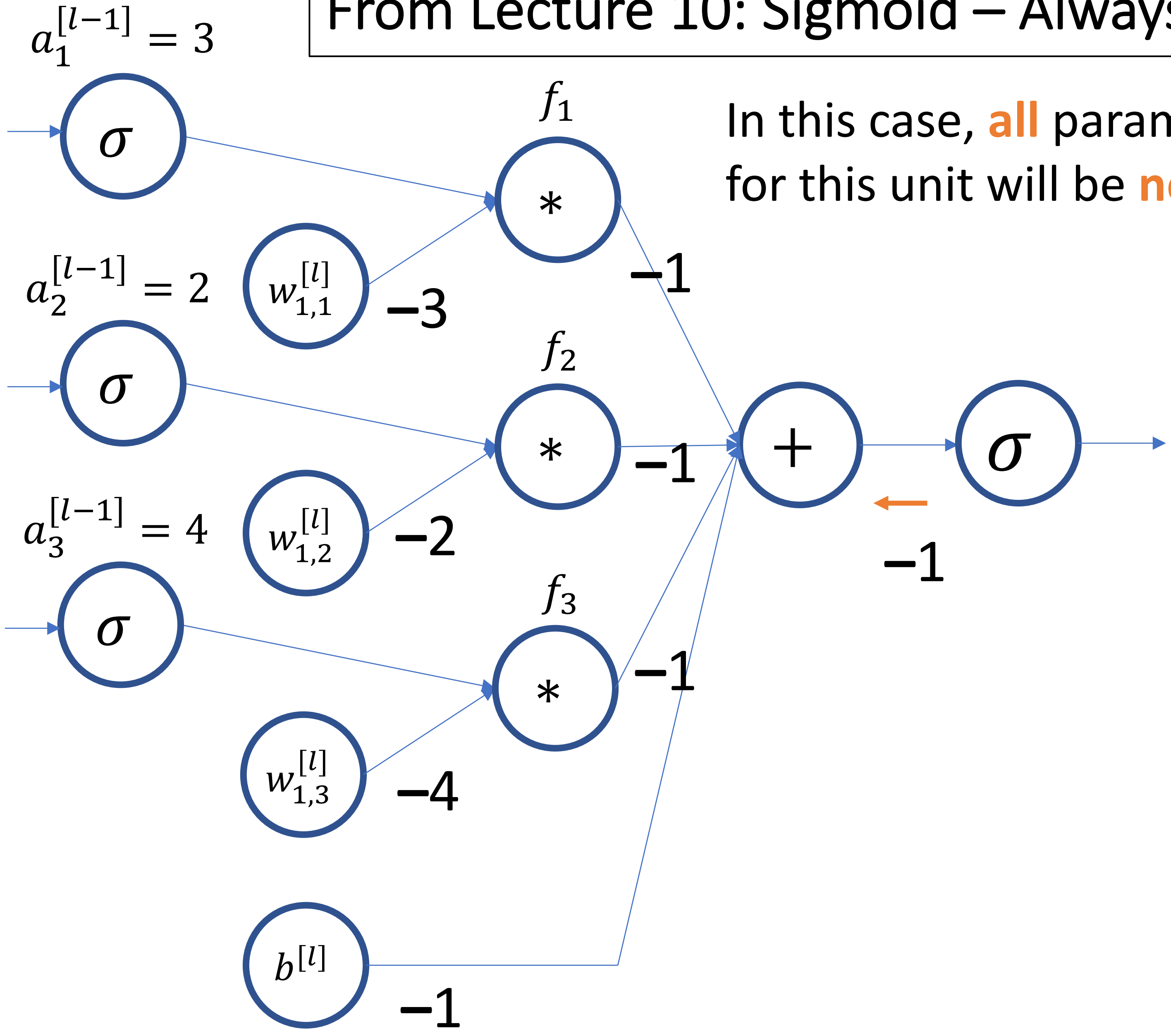
Recall From Lecture 10: Sigmoid – Always positive

- Another Problem with Sigmoid is that it is always positive
- Assuming all units in the layer use the same activation function, then all units in a layer will output a positive number

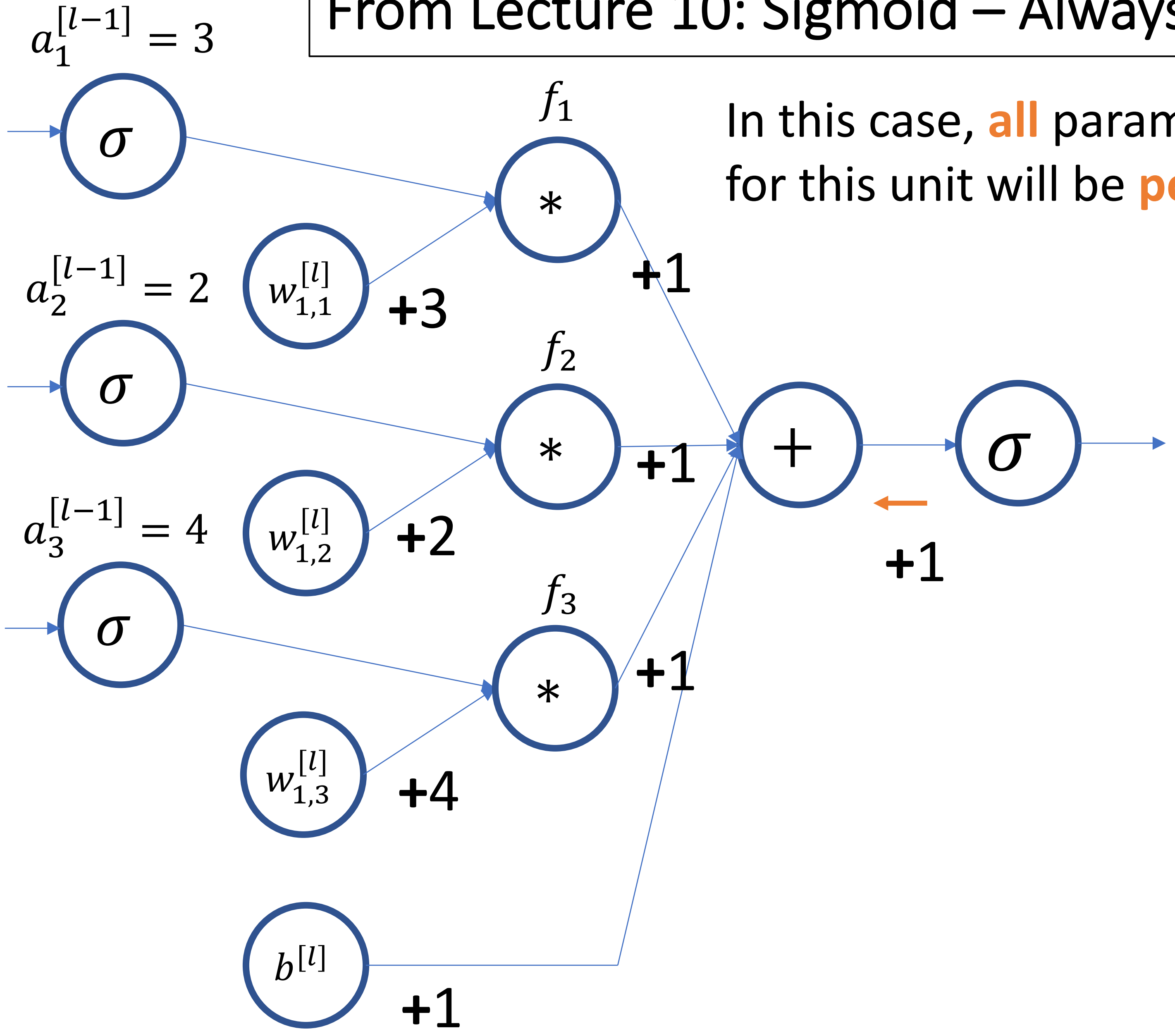


From Lecture 10: Sigmoid – Always positive

In this case, **all** parameter updates for this unit will be **negative**!

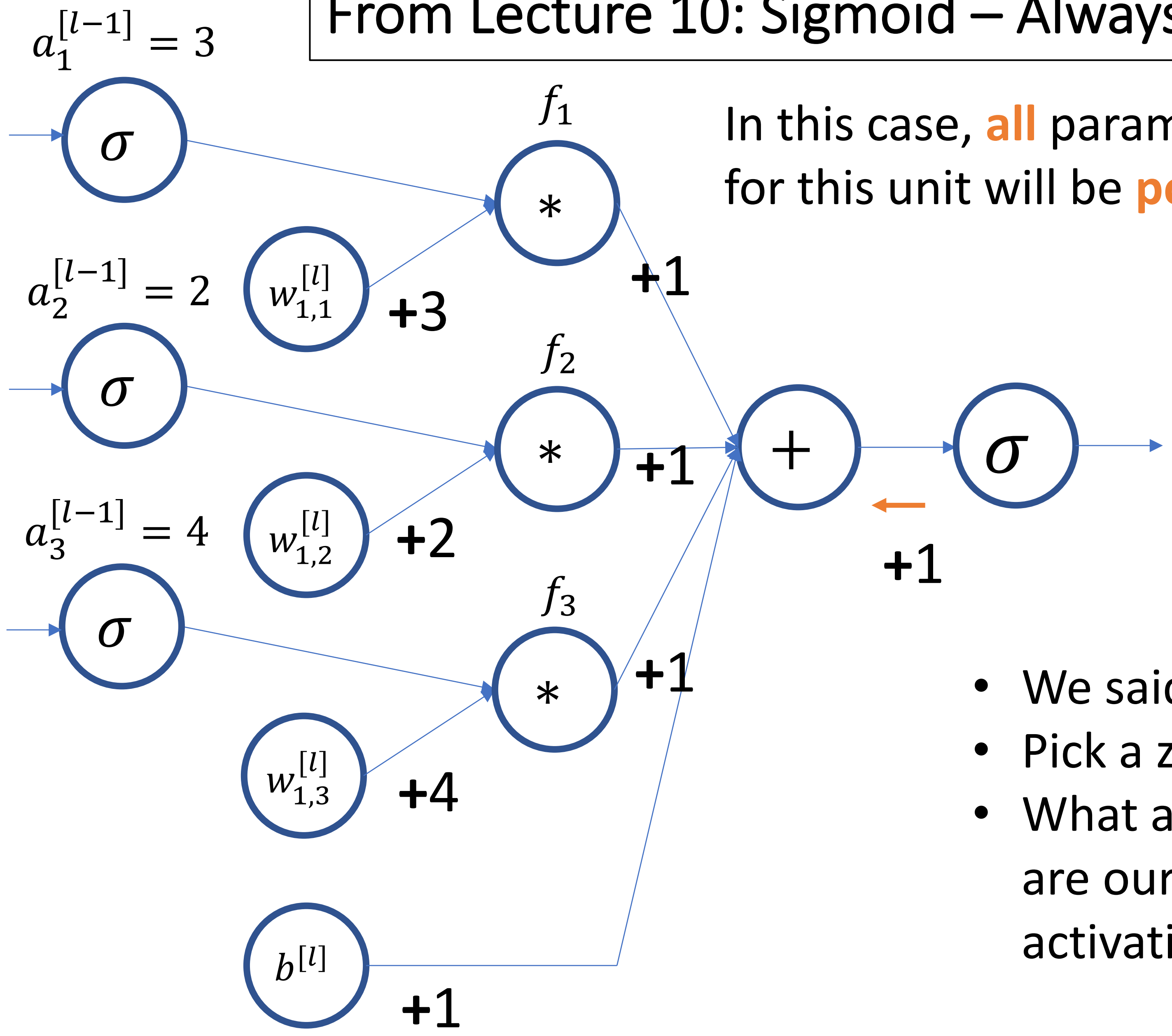


From Lecture 10: Sigmoid – Always positive



In this case, **all** parameter updates for this unit will be **positive**!

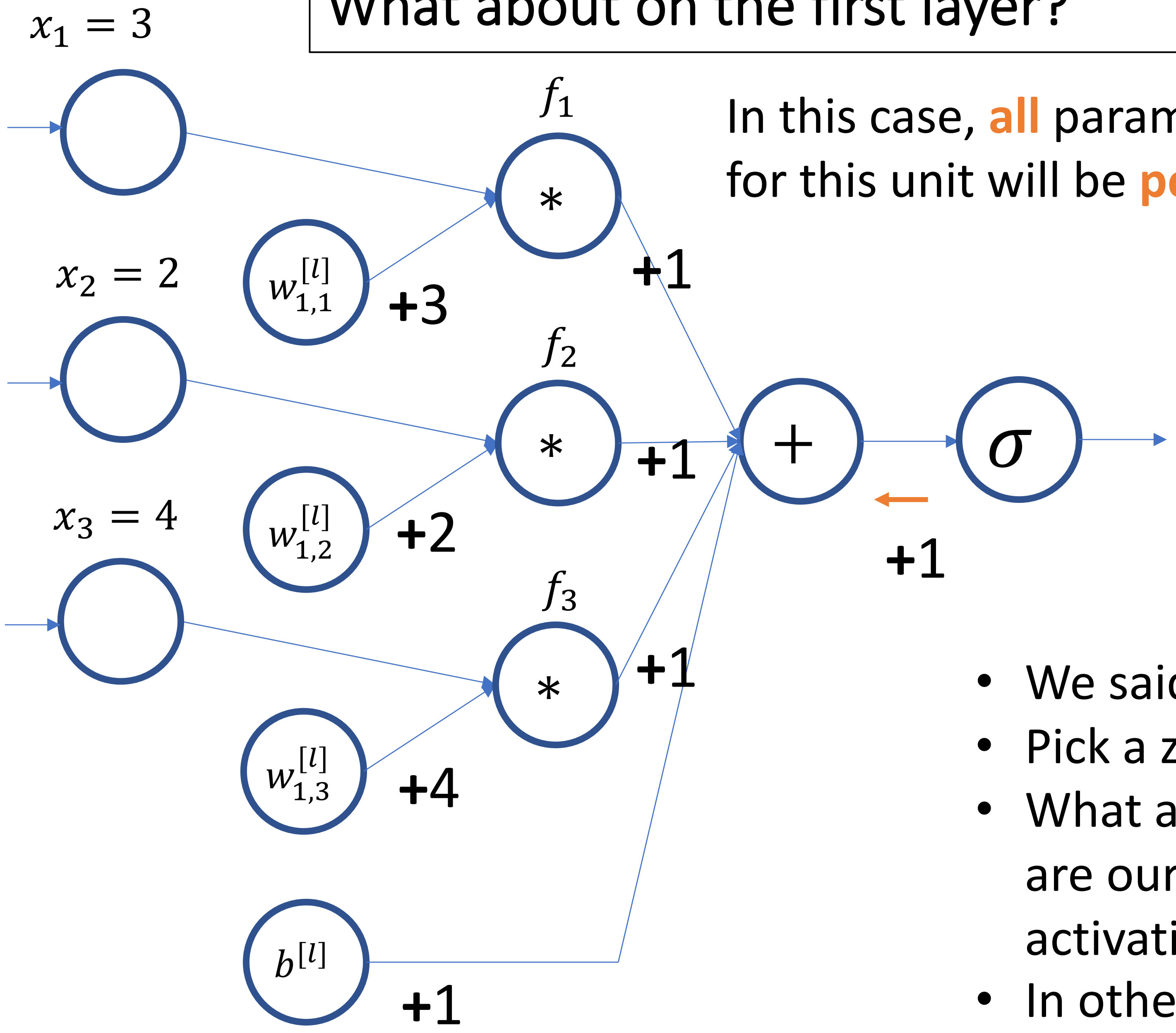
From Lecture 10: Sigmoid – Always positive



In this case, **all** parameter updates for this unit will be **positive**!

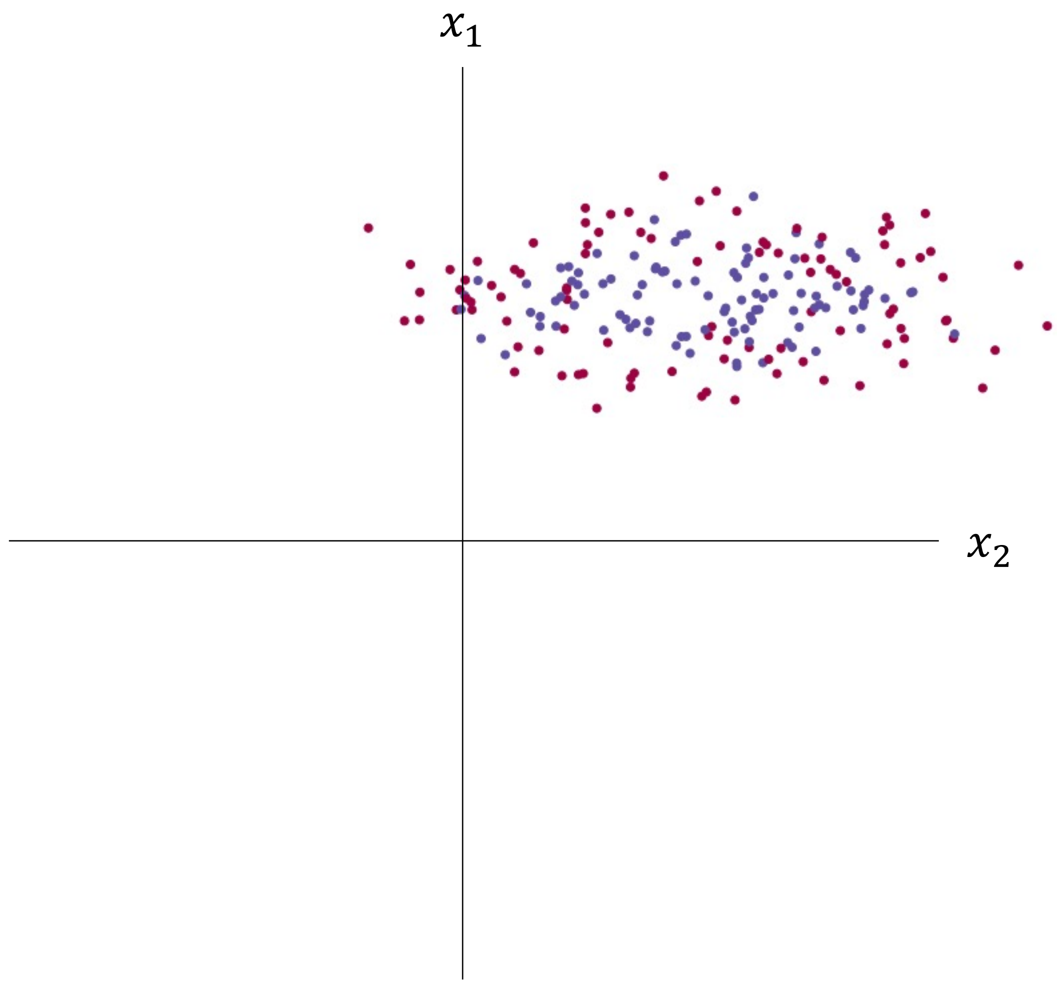
- We said this can lead to inefficient training
- Pick a zero-centered activation function
- What about at the first layer when the inputs are our training data, and there are no activation functions involved?

What about on the first layer?

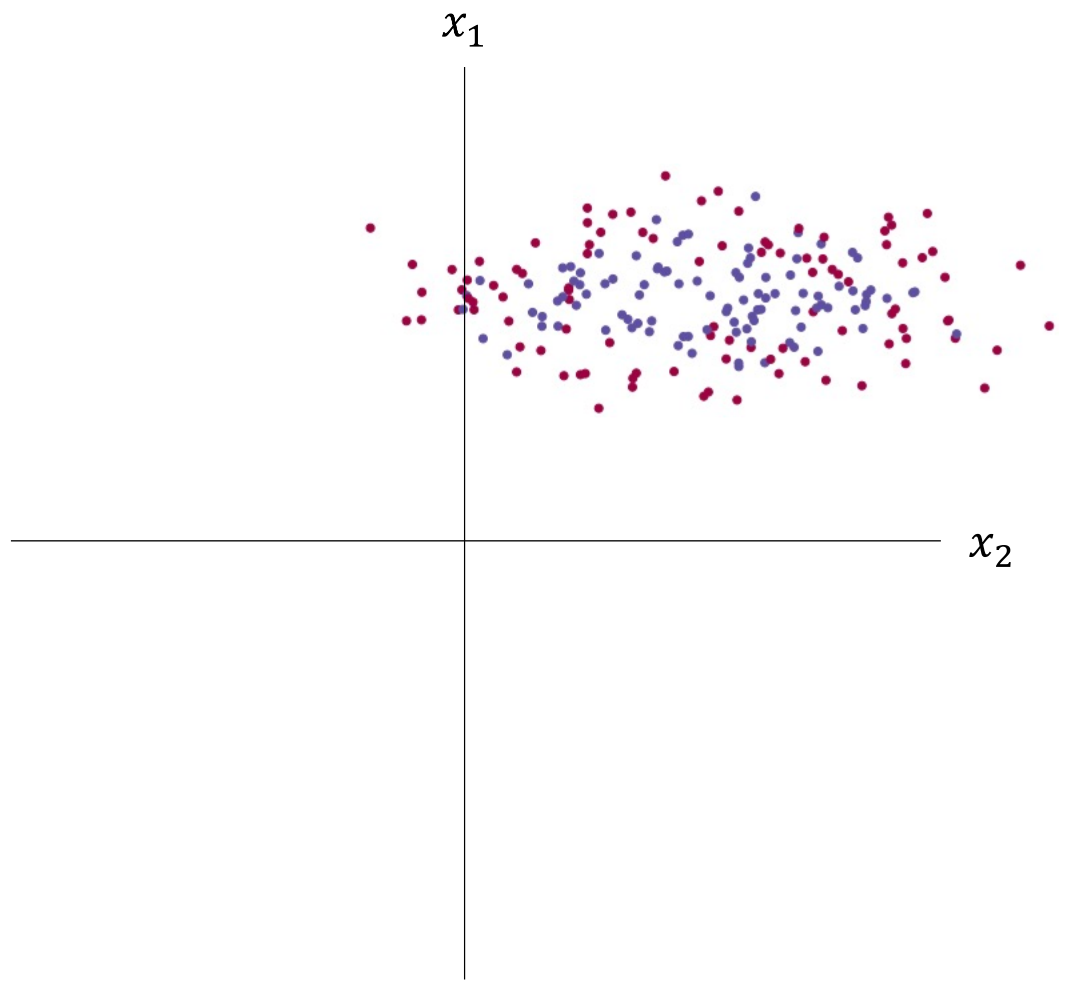


In this case, **all** parameter updates for this unit will be **positive**!

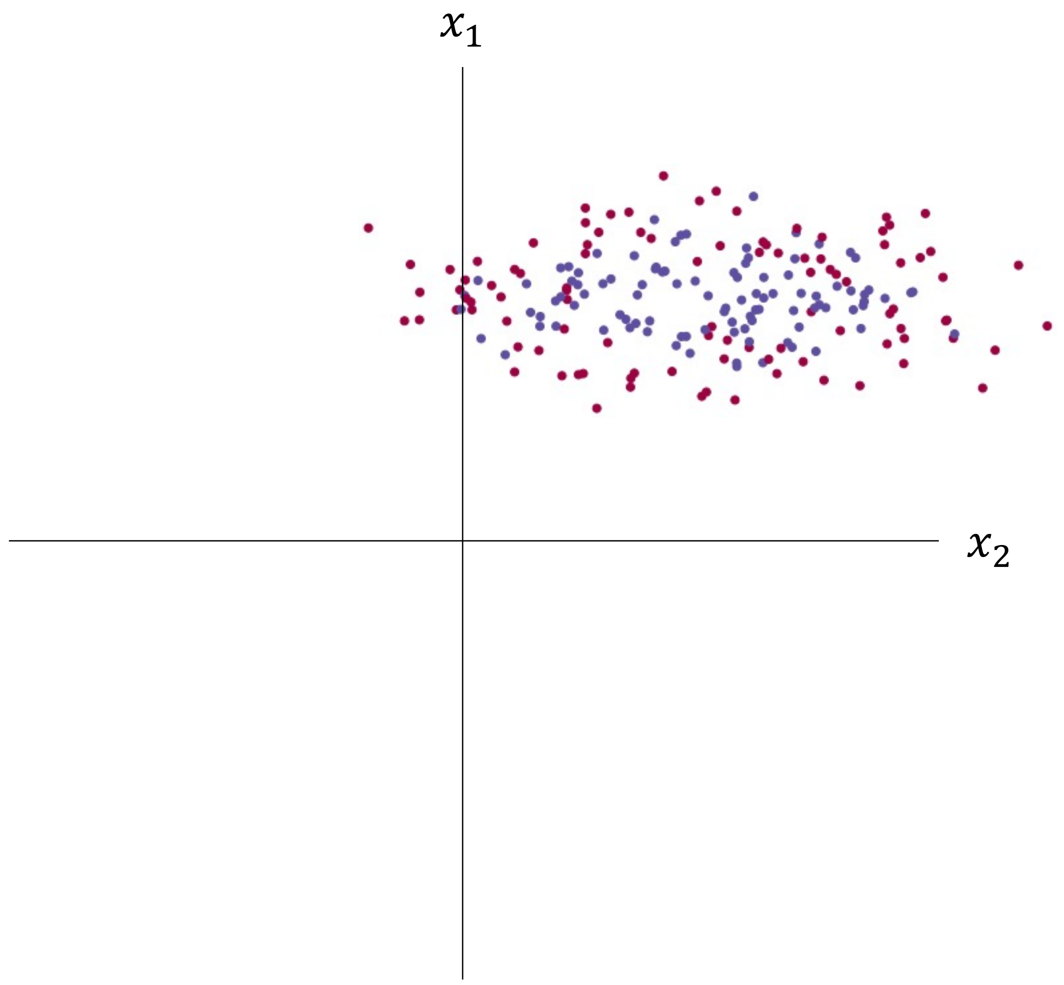
- We said this can lead to inefficient training
- Pick a zero-centered activation function
- What about at the first layer when the inputs are our training data, and there are no activation functions involved?
- In other words, what if our data is non-zero centered


x_1
x_2

Mean Subtraction



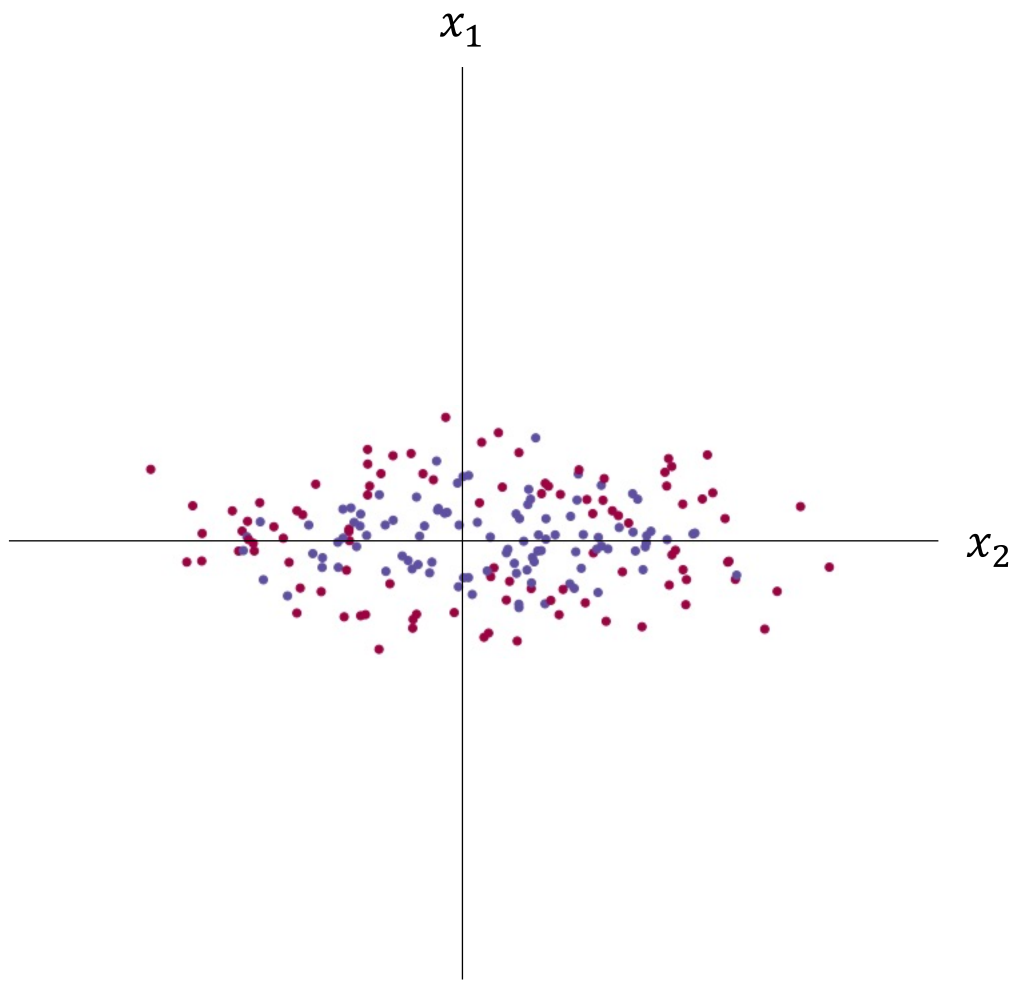
Mean Subtraction



- Compute mean of each feature across all training samples

$$\mu_i = \frac{1}{m}\sum_{j=1}^{m} x_i^{(j)}$$

Mean Subtraction



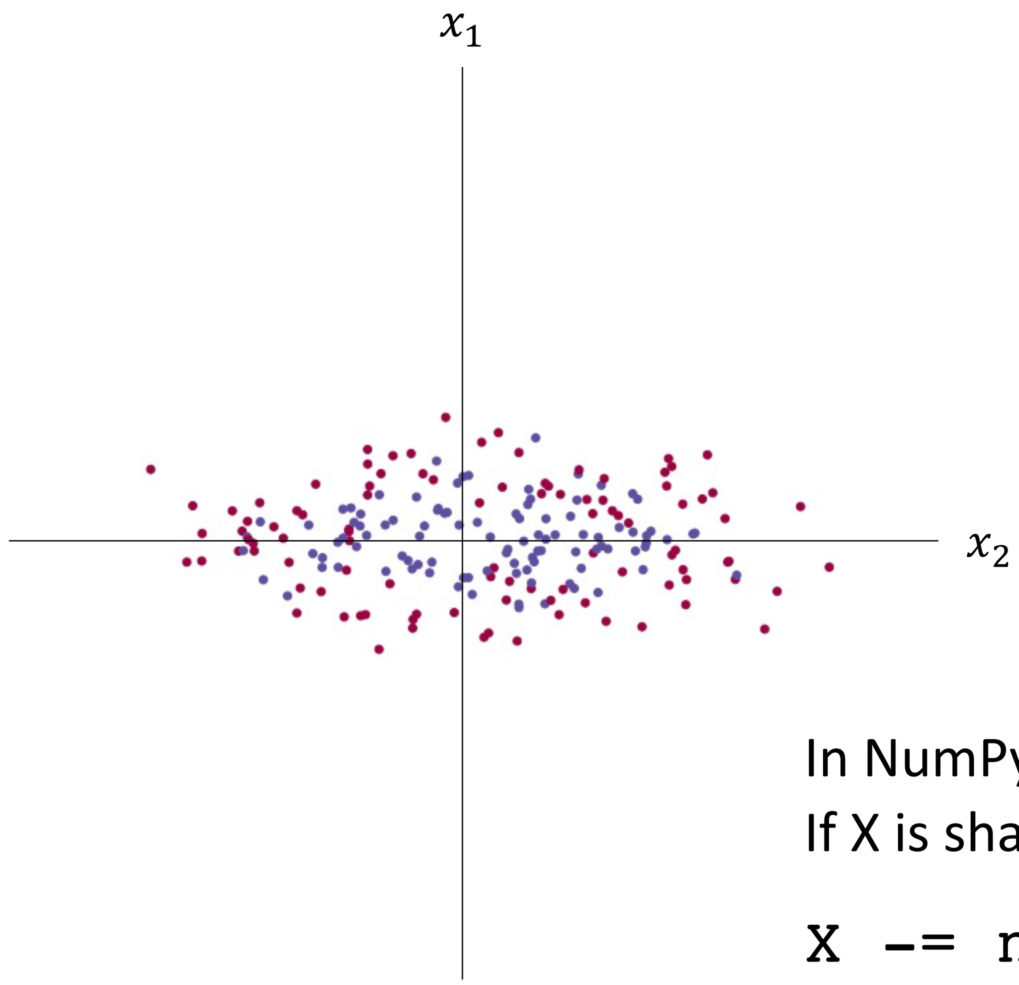
- Compute mean of each feature across all training samples

$$\mu_i = \frac{1}{m}\sum_{j=1}^{m} x_i^{(j)}$$

- Subtract mean from each sample's features

$$x' = x - \mu$$

Mean Subtraction



- Compute mean of each feature across all training samples

$$\mu_i = \frac{1}{m}\sum_{j=1}^{m} x_i^{(j)}$$

- Subtract mean from each sample's features

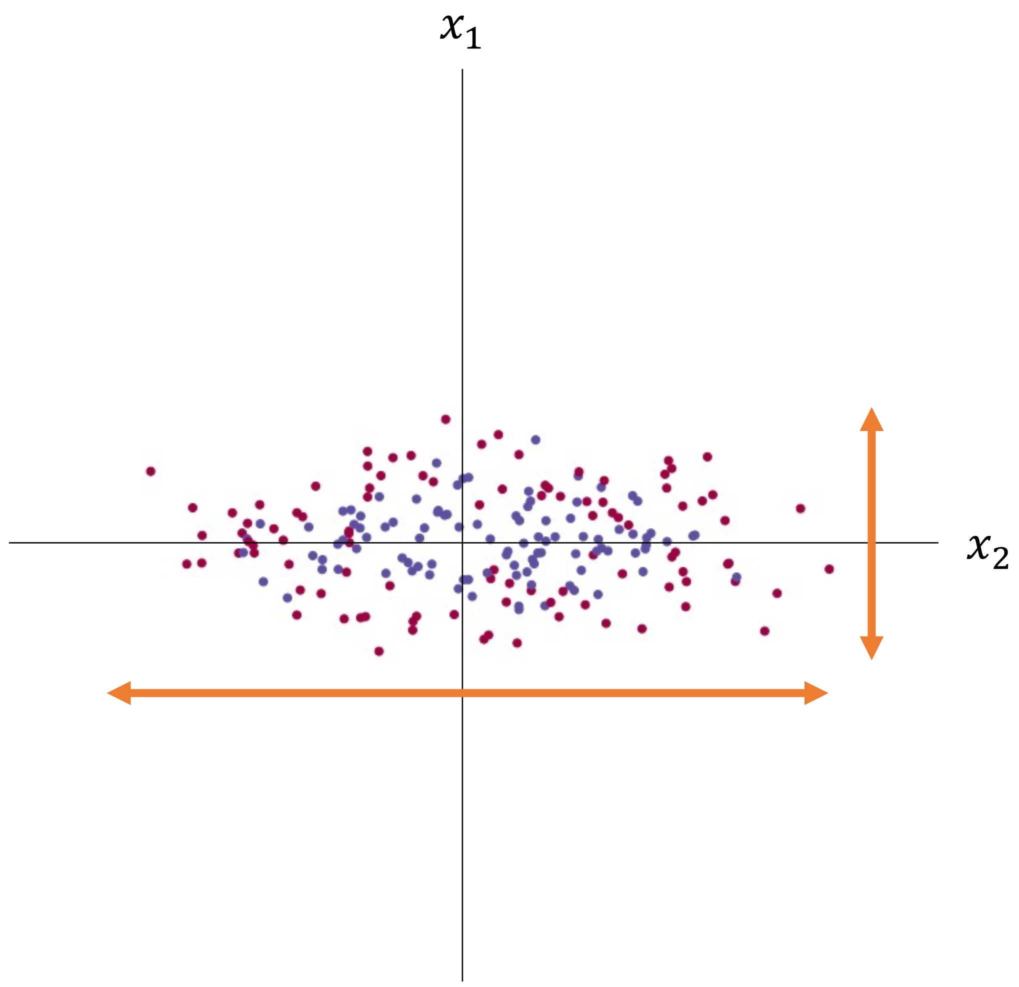
$$x' = x - \mu$$

In NumPy:

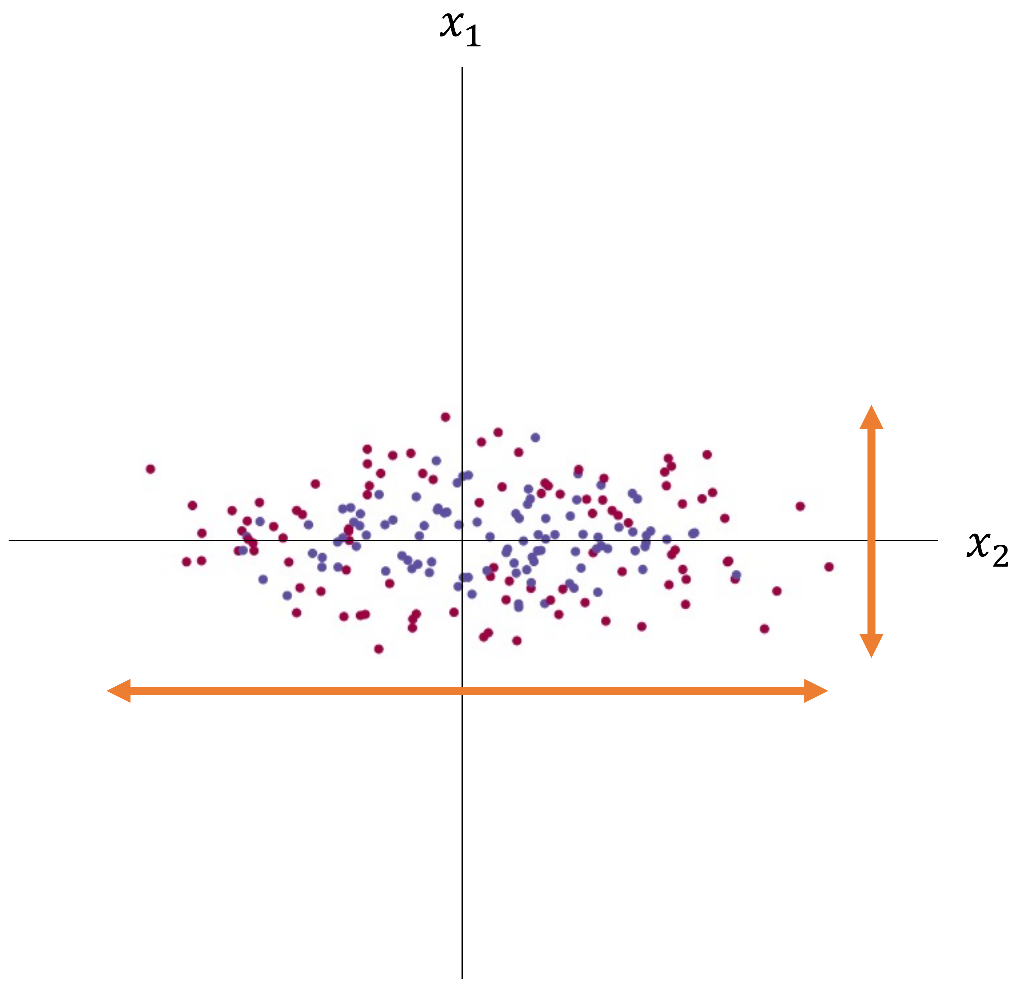
If X is shape (n,m) where n is #features, m is #samples

```
X -= np.mean(X, axis=1)
```

Normalization/Scaling



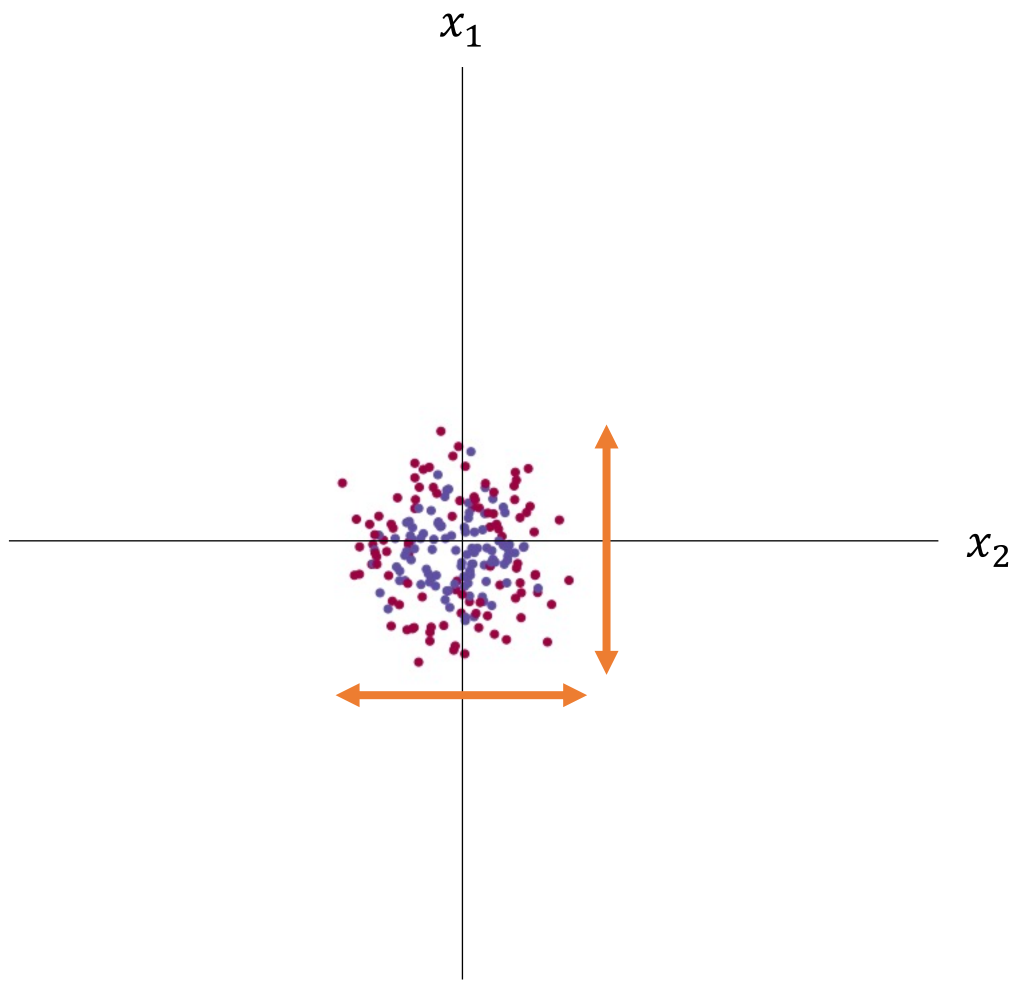
Normalization/Scaling



- Compute variance of each feature across all training samples

$$\sigma_i{}^2 = \frac{1}{m}\sum_{j=1}^{m}\left(x_i^{(j)} - \mu_i\right)^2$$

Normalization/Scaling



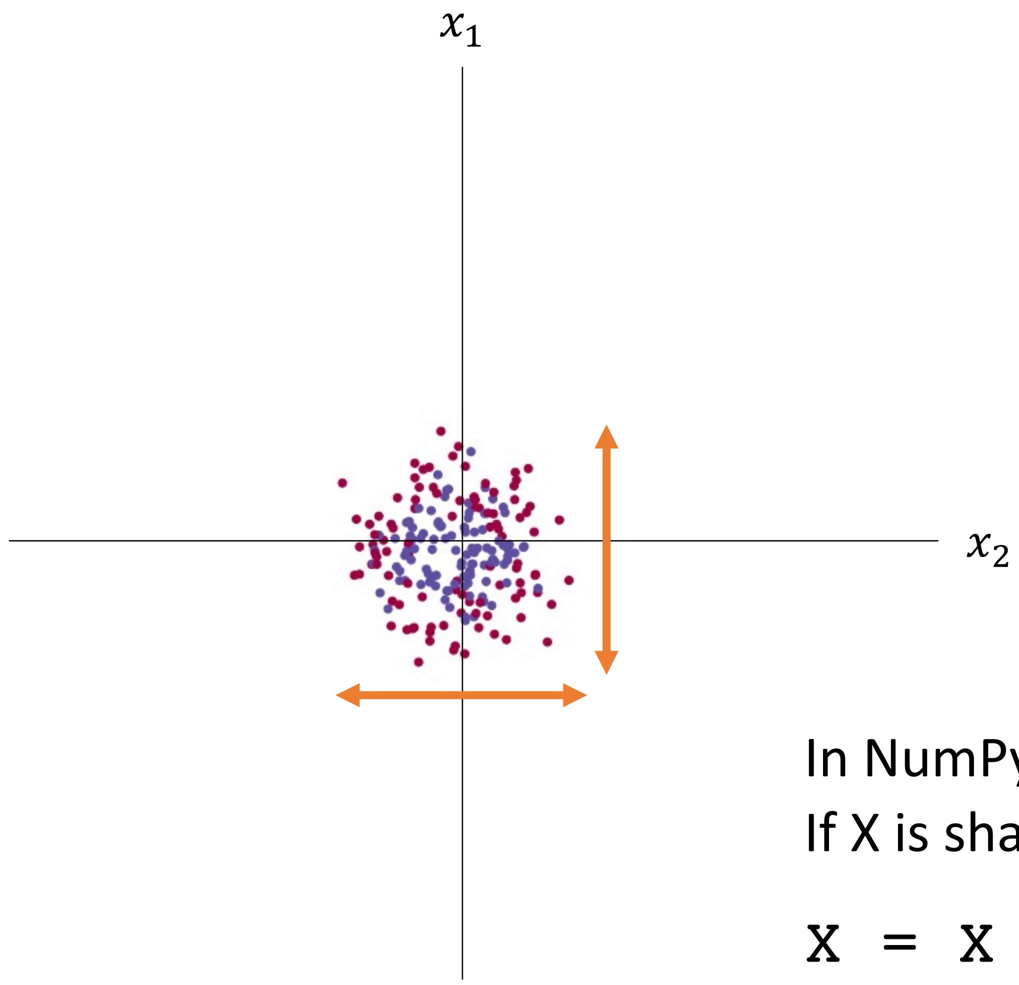
- Compute variance of each feature across all training samples

$$\sigma_i{}^2 = \frac{1}{m}\sum_{j=1}^{m}\left(x_i^{(j)} - \mu_i\right)^2$$

- Divide each feature by its standard deviation

$$x' = x/\sigma$$

Normalization/Scaling



- Compute variance of each feature across all training samples

$$\sigma_i{}^2 = \frac{1}{m}\sum_{j=1}^{m}\left(x_i^{(j)} - \mu_i\right)^2$$

- Divide each feature by its standard deviation

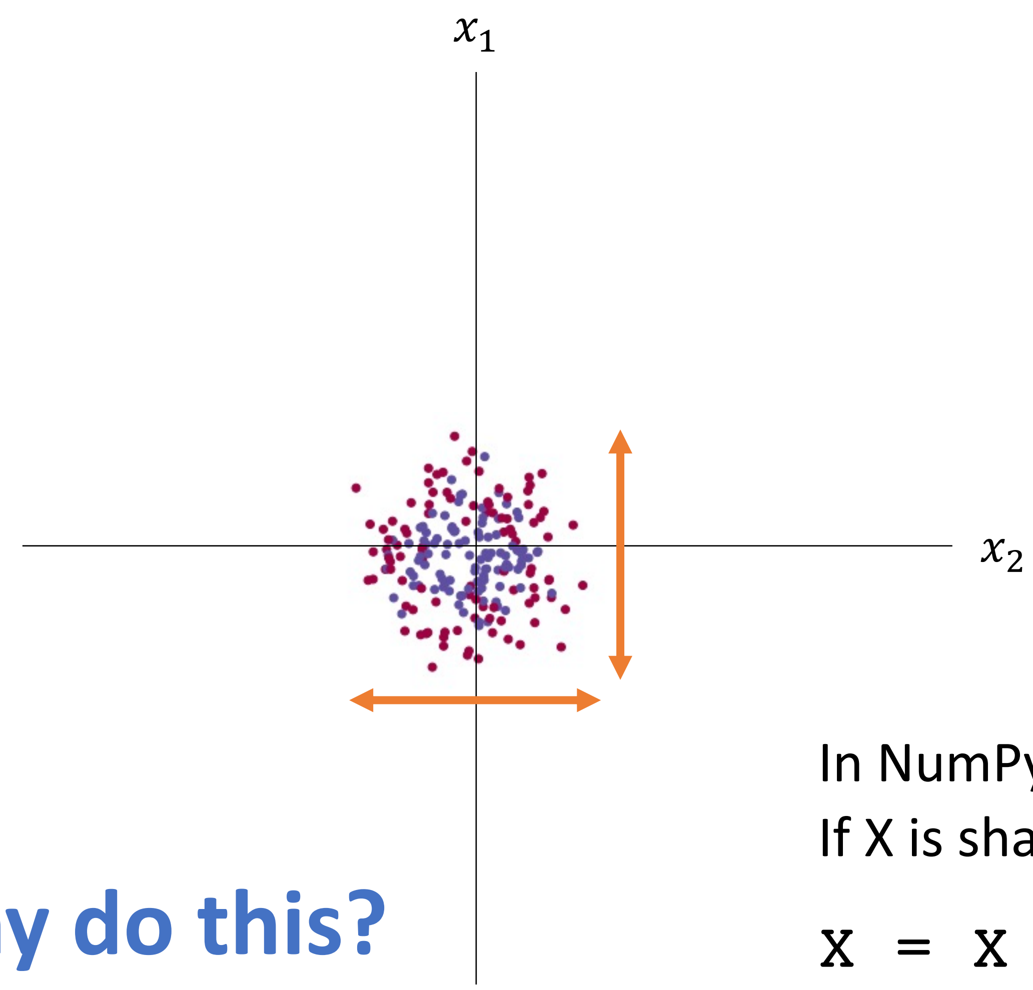
$$x' = x/\sigma$$

In NumPy:

If X is shape (n,m) where n is #features, m is #samples

```
X = X / np.std(X, axis=1)
```

Normalization/Scaling



- Compute variance of each feature across all training samples

$$\sigma_i^{\,2} = \frac{1}{m}\sum_{j=1}^{m}\left(x_i^{(j)} - \mu_i\right)^2$$

- Divide each feature by its standard deviation

$$x' = x/\sigma$$

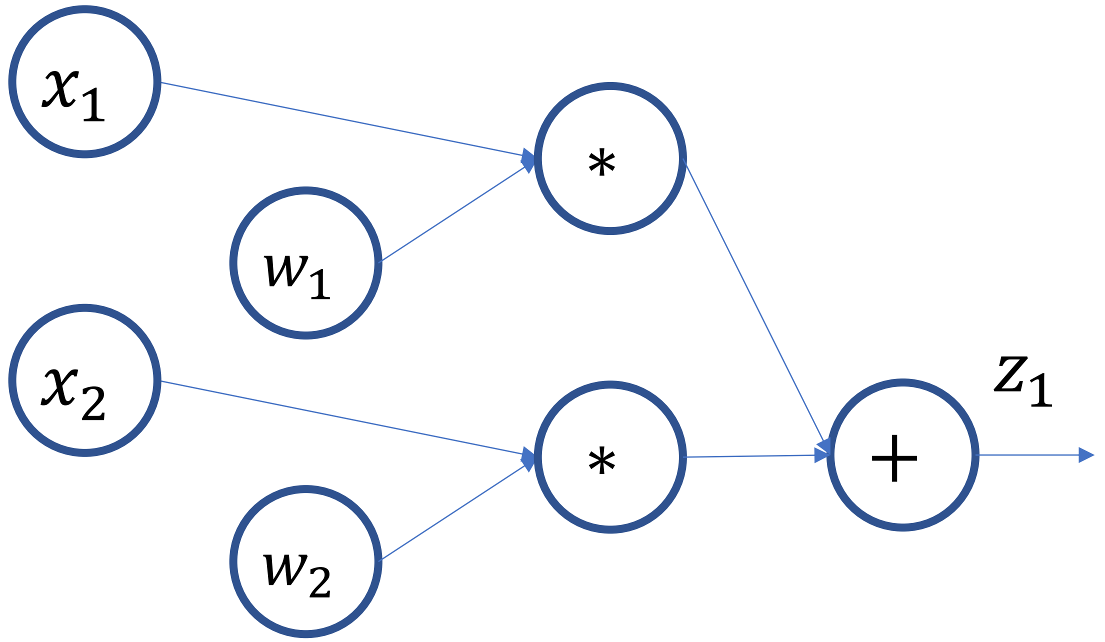
In NumPy:

If X is shape (n,m) where n is #features, m is #samples

```
X = X / np.std(X, axis=1)
```

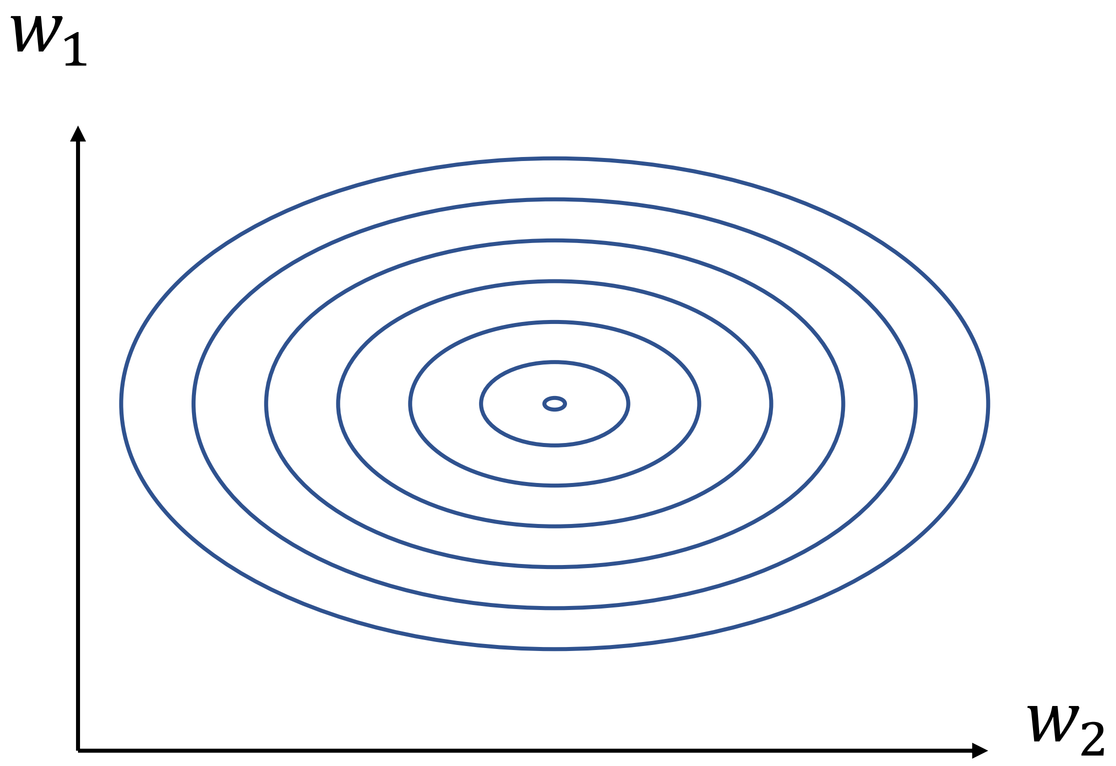
Why do this?

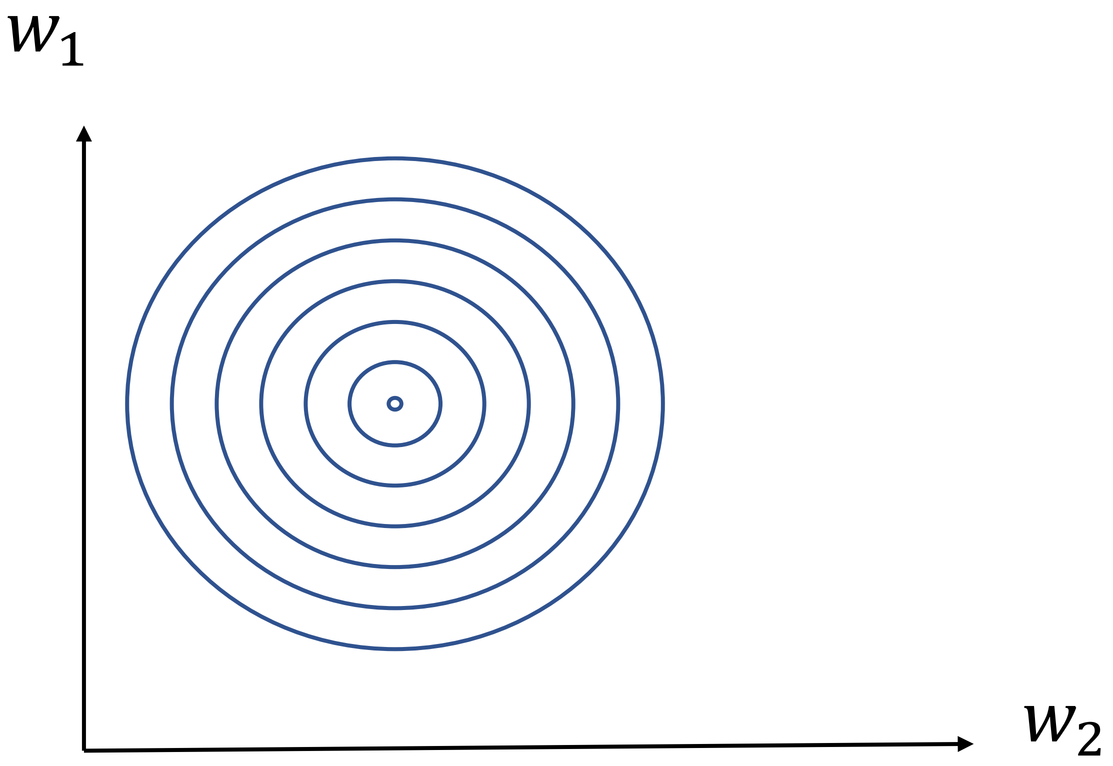
Why Normalize?



Why Normalize? – Relative Feature Scales

Corresponding weights will tend to become similar scale as well



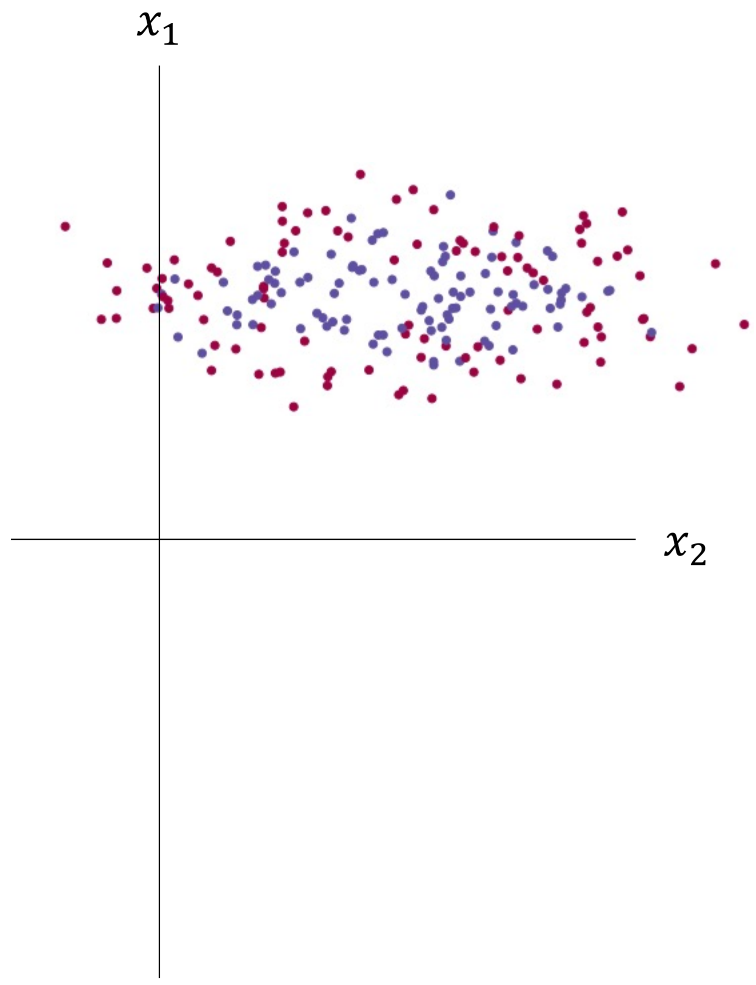


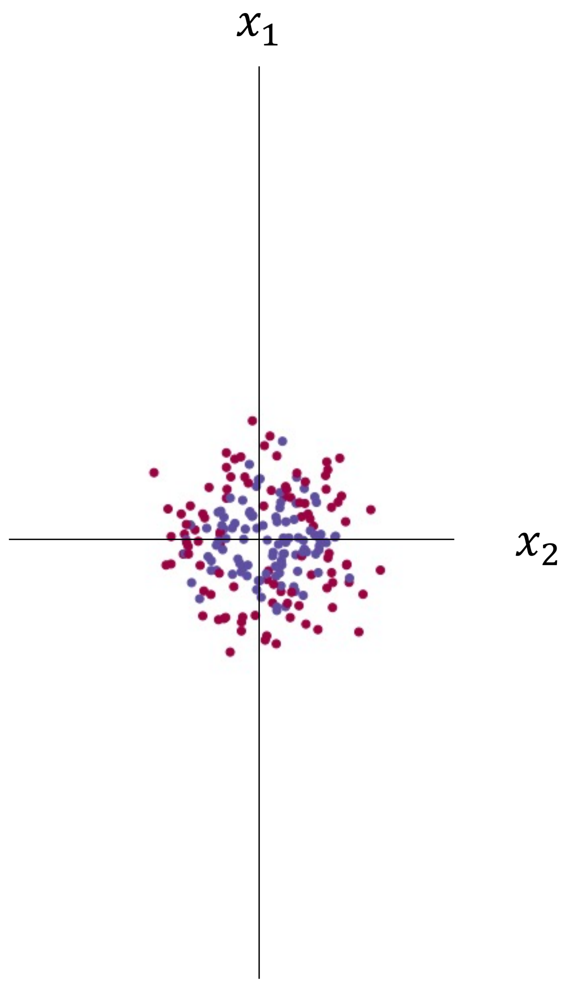
Under this view, can see we don't need exactly same scale, but the closer the better

Why Normalize? – Absolute Feature Scales

- Even if all features are on similar scale, we don’t want these scales to be large.
- Still leads to large gradients. Means that a small change will lead to big changes in final Cost
- This means Cost will be very sensitive to small changes to weights
- Makes problem harder to optimize

Standardization (Z-Score Normalization)



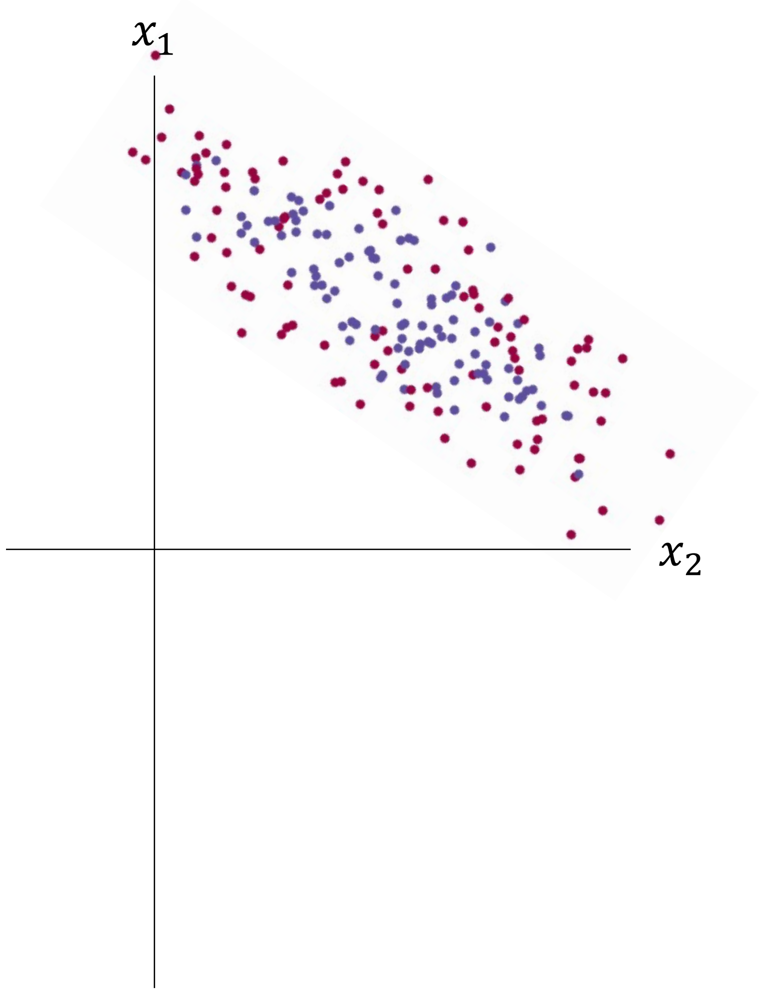


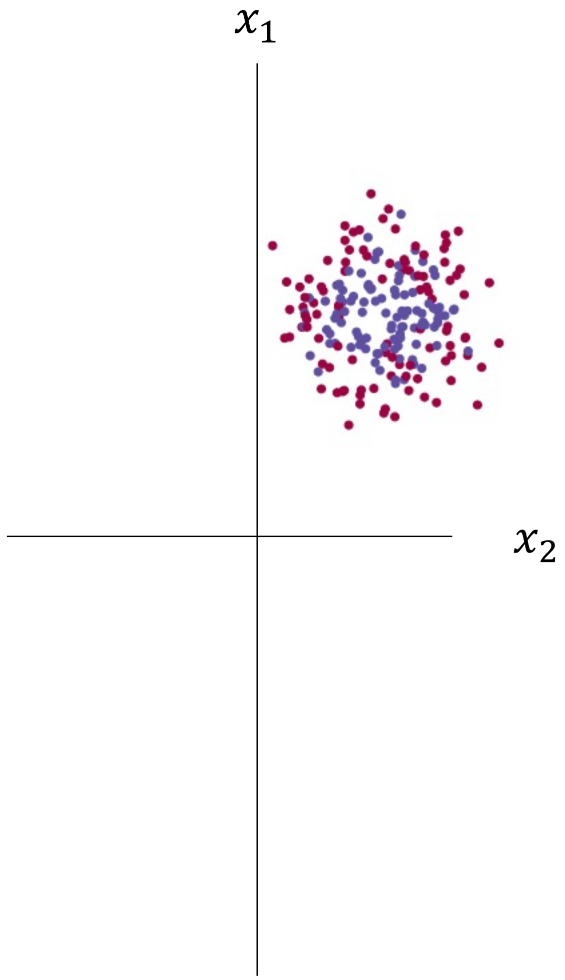
$$\mu_i = \frac{1}{m}\sum_{j=1}^{m} x_i^{(j)}$$

$$\sigma_i{}^2 = \frac{1}{m}\sum_{j=1}^{m}\left(x_i^{(j)} - \mu_i\right)^2$$

$$x_i{}' = \frac{x_i - \mu_i}{\sigma_i}$$

Whitening / Decorrelating





For Image Data

- Pixel data is usually stored as values from 0-255
- Each pixel is a feature, therefore, each feature is on the same scale relative to each other
- May still want to normalize

For Image Data – Examples

- AlexNet
  - Subtracted mean image
  - i.e. computed mean on each feature (pixel) across dataset
- VGGNet
  - Calculated a mean for each color channel (e.g. μ_r, μ_g, μ_b)
  - Each channel's feature (pixel) subtracted its corresponding channel mean
- ResNet
  - Subtracted channel mean like VGGNet
  - Also divided by channel standard deviation

At Inference/Prediction Time

- Any transformation you performed on an input (e.g. input image) for training must be performed for inputs supplied at prediction time

- For example, if you used Z-Score Normalization, you need to apply the same function with the same mean and std during prediction time, that you used during training

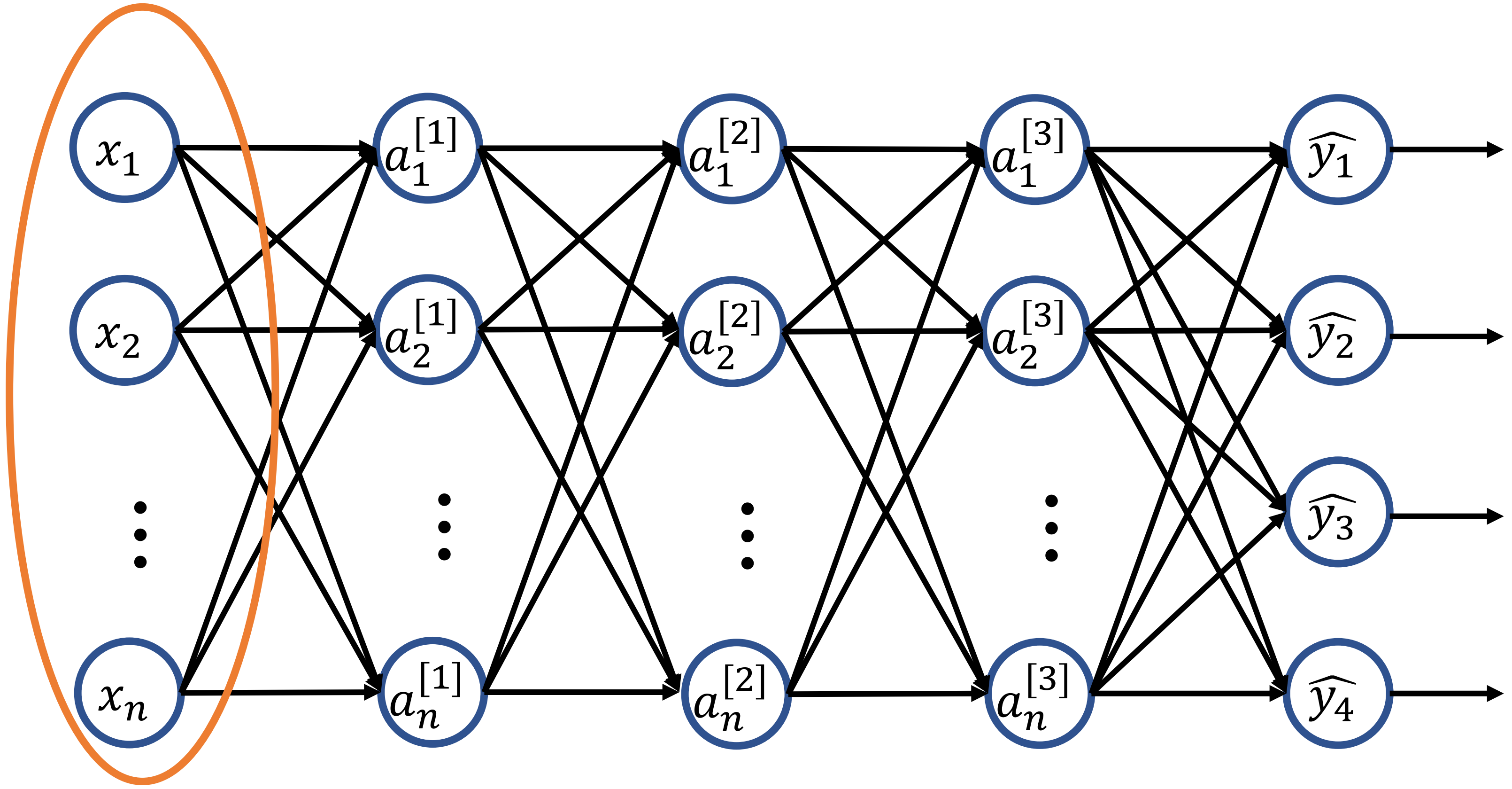
$$x_i{}' = \frac{x_i - \mu_i}{\sigma_i}$$

Batch Normalization

” Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift”, Ioffe and Szegedy, 2015, https://arxiv.org/pdf/1502.03167.pdf

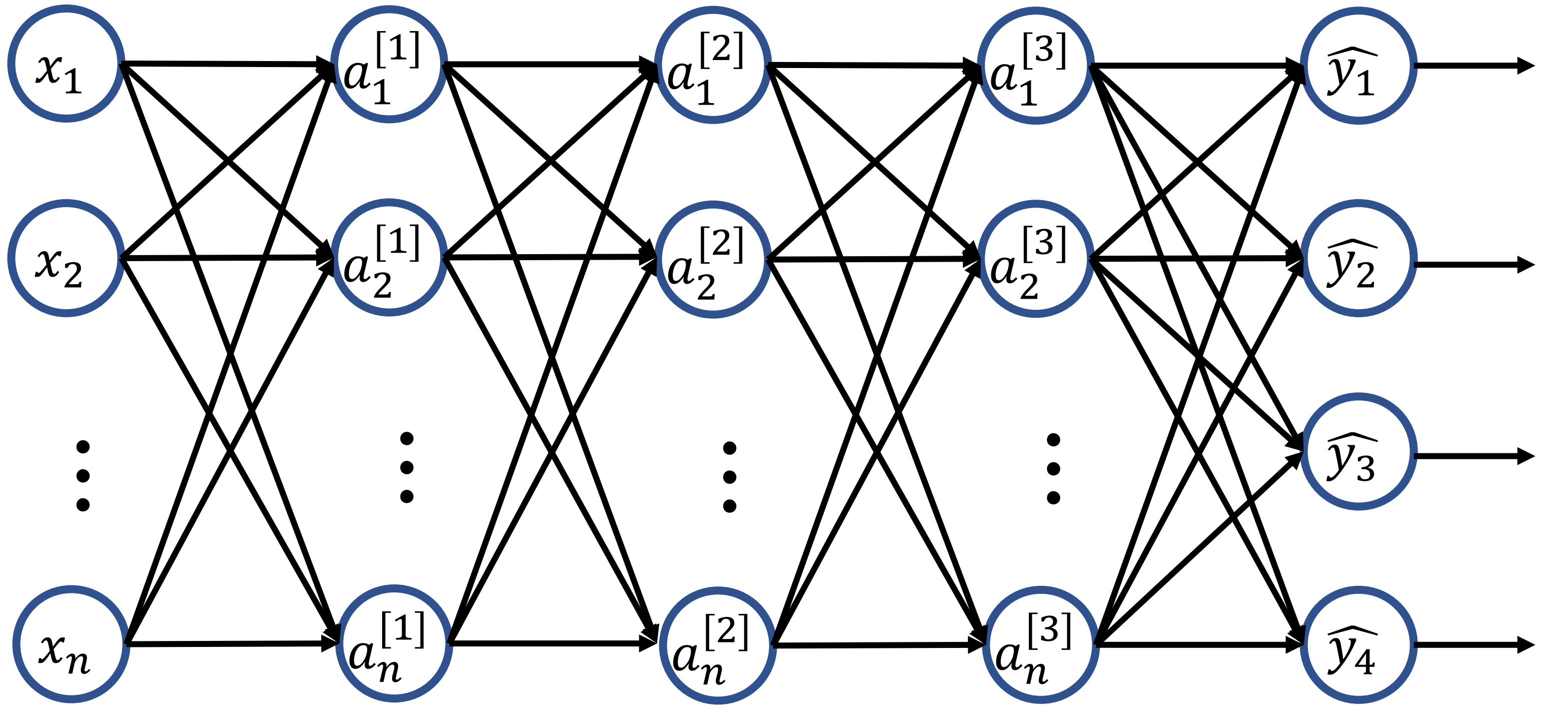
Normalizing the Network Inputs

- We discussed how preprocessing the input data can help optimization
- This will help training the parameters in the first layer



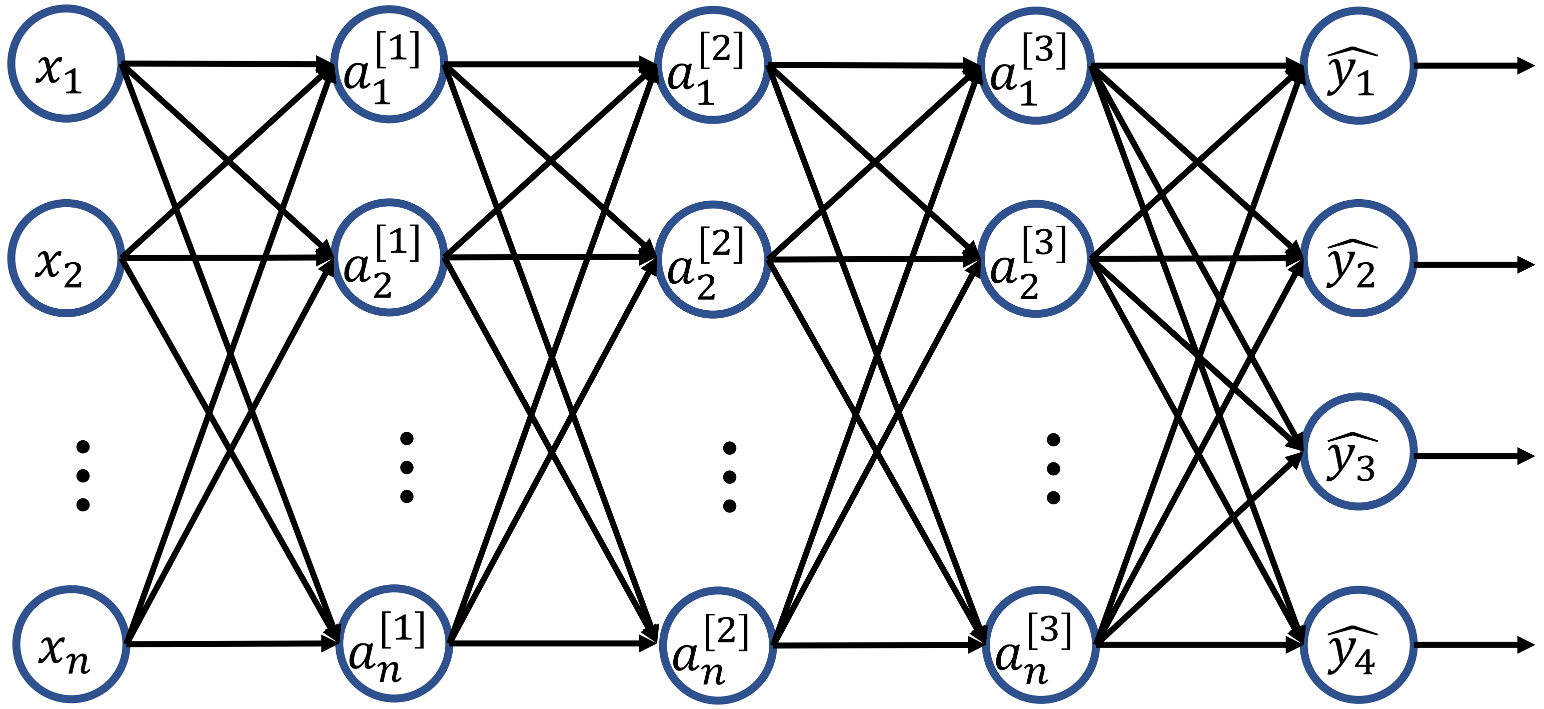
What about normalizing inputs of the hidden layers?

- Inputs to the hidden layers are the outputs of previous layers



What about normalizing inputs of the hidden layers?

- Inputs to the hidden layers are the outputs of previous layers
- Batch Normalization



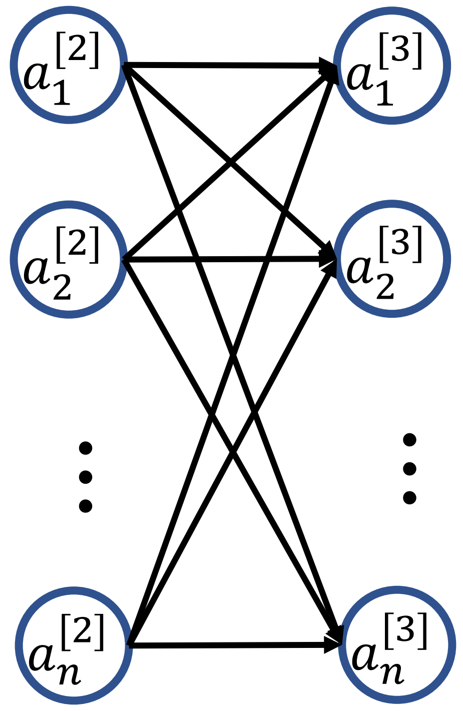
Intuition of Batch Normalization

- The inputs to each layer is the output of the previous layer
- But the function of each layer is changing as training progresses
- This makes the optimization problem hard for intermediate layers as the inputs for which it is optimizing against is changing!
- Furthermore, for deep networks, a small change in an earlier layer's outputs, can have significant impact on the inputs to a deeper layer
- Batch Normalization helps stabilize the optimization problem somewhat by giving each layer a target mean and variance

At least this was the original hypothesis of why it works ..

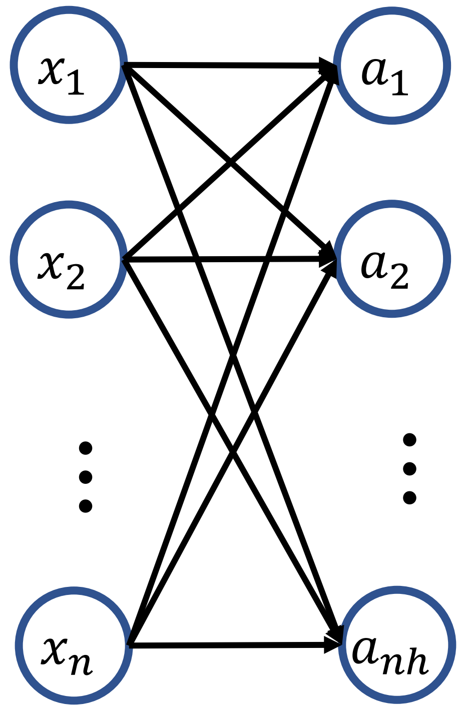
Batch Normalization

- Let’s look at an arbitrary intermediate layer



Batch Normalization

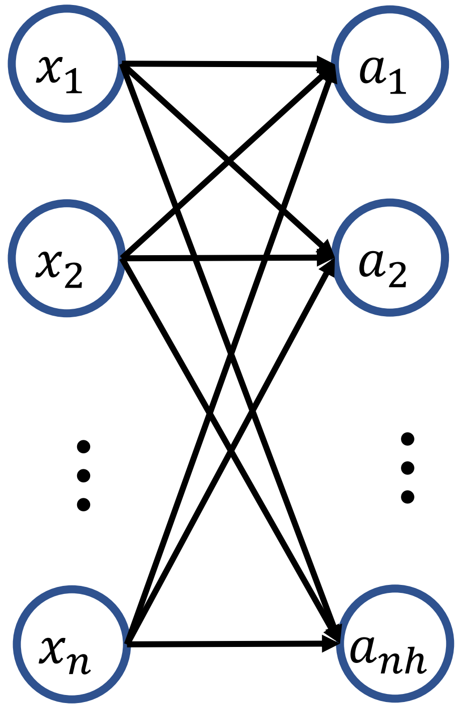
- Let's look at an arbitrary intermediate layer
- Let's simplify the terminology
  - Call the inputs x
  - Remove the $[l]$



Batch Normalization - How it works

- For a given mini-batch with m samples,
  x is a matrix of shape (n, m)
- For each input x_i, compute its
  - mean μ_i across the mini-batch → n means
  - variance ${\sigma_i}^2$ across the mini-batch → n variances

$$\mu_i = \frac{1}{m}\sum_{j=1}^{m} x_i^{(j)} \qquad {\sigma_i}^2 = \frac{1}{m}\sum_{j=1}^{m}\left(x_i^{(j)} - \mu_i\right)^2$$



- For each sample and each feature, normalize → (n, m)

$${x_i}' = \frac{x_i - \mu_i}{\sigma_i}$$

Zero Mean Unit Variance Too Strict

- Forcing each layer’s outputs to have a zero-mean and unit variance distribution is too strict.
- This will make the optimization problem even harder
- Recall the whole point is to make the optimization problem easier!
- Instead of a specific mean of 0 and variance of 1 for all layer outputs, let the model learn some target mean and variance for each layer

Learned Mean and Variance

- Two new trainable parameters, γ_i, β_i for each layer output that act to shift and scale the normalized layer outputs
- Normalize like before

$$\mu_i = \frac{1}{m}\sum_{j=1}^{m} x_i^{(j)} \qquad {\sigma_i}^2 = \frac{1}{m}\sum_{j=1}^{m}\left(x_i^{(j)} - \mu_i\right)^2 \qquad {x_i}' = \frac{x_i - \mu_i}{\sigma_i}$$

- Now shift and scale normalized values → (n, m)

$$\widetilde{x_i} = \gamma_i x_i' + \beta_i$$

Intuition of Shift and Scale Hyperparameters

$$\widetilde{x_i} = \gamma_i x_i' + \beta_i \qquad {x_i}' = \frac{x_i - \mu_i}{\sigma_i}$$

Examples

- if $\gamma_i = \sigma_i$ and $\beta_i = \mu_i$, then $\widetilde{x_i} = x_i$ and hence $\widetilde{x_i}$ has the same mean and variance as the original input distribution
- if $\gamma_i = 1$ and $\beta_i = 0$, then $\widetilde{x_i} = {x_i}'$ and hence $\widetilde{x_i}$ has 0 mean and unit variance
- For other values of γ_i and β_i, x_i will have some other mean and variance

Batch Normalization

- We saw that zero-centering and normalizing the input data can help first layer’s training
- Batch Normalization extends this concept to deeper layers
  - Key difference is that we don’t want to restrict what functions can be learned by the model too much. So instead of forcing a strict zero mean and unit variance, we let the model learn a suitable mean and variance for each layer that uses Batch Normalization

Where to do BatchNorm

- All the computations in Batch Normalization are differentiable equations.
- Can therefore introduce this operation into our neural network compute graph
- Backprop will work
- We can create a new Batch Normalization layer

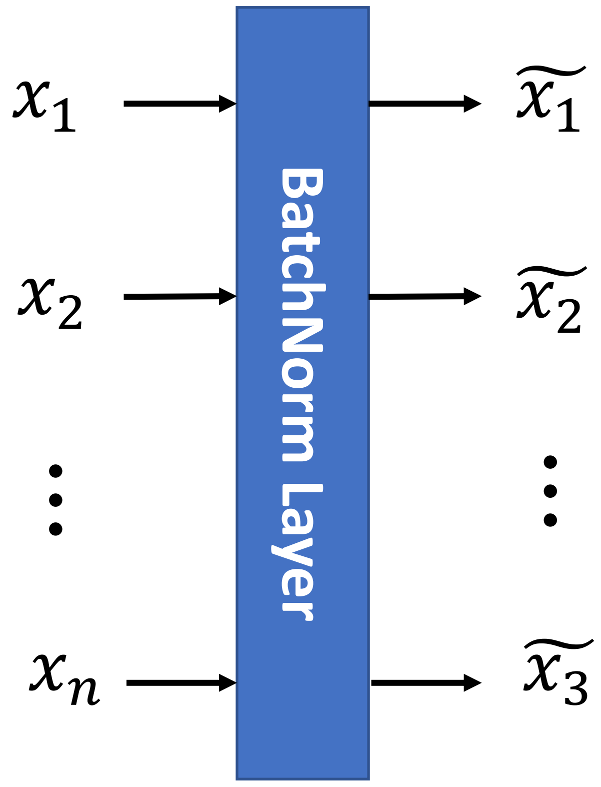
BatchNorm as a aLayer

$$\mu_i = \frac{1}{m}\sum_{j=1}^{m} x_i^{(j)}$$

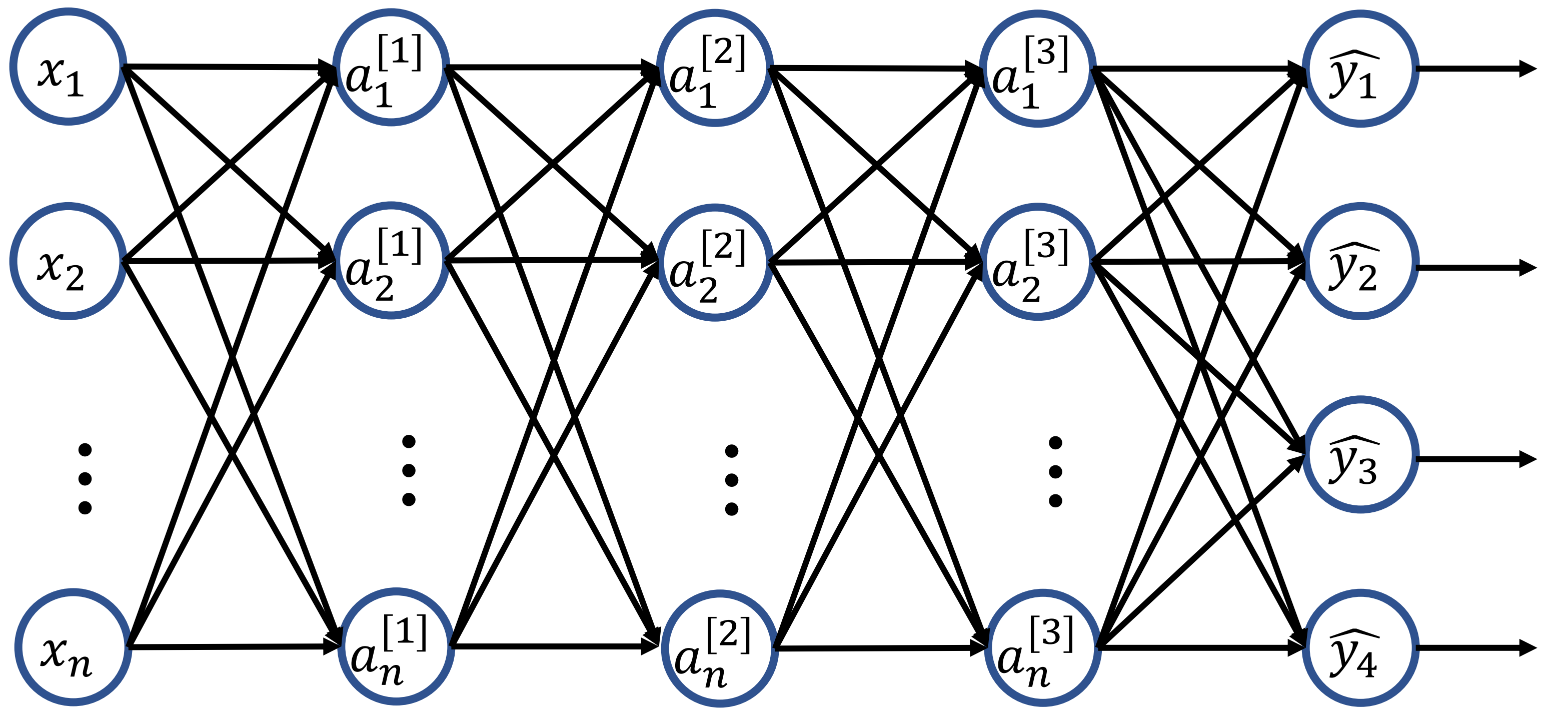
$$\sigma_i{}^2 = \frac{1}{m}\sum_{j=1}^{m}\left(x_i^{(j)} - \mu_i\right)^2$$

$$x_i{}' = \frac{x_i - \mu_i}{\sigma_i}$$

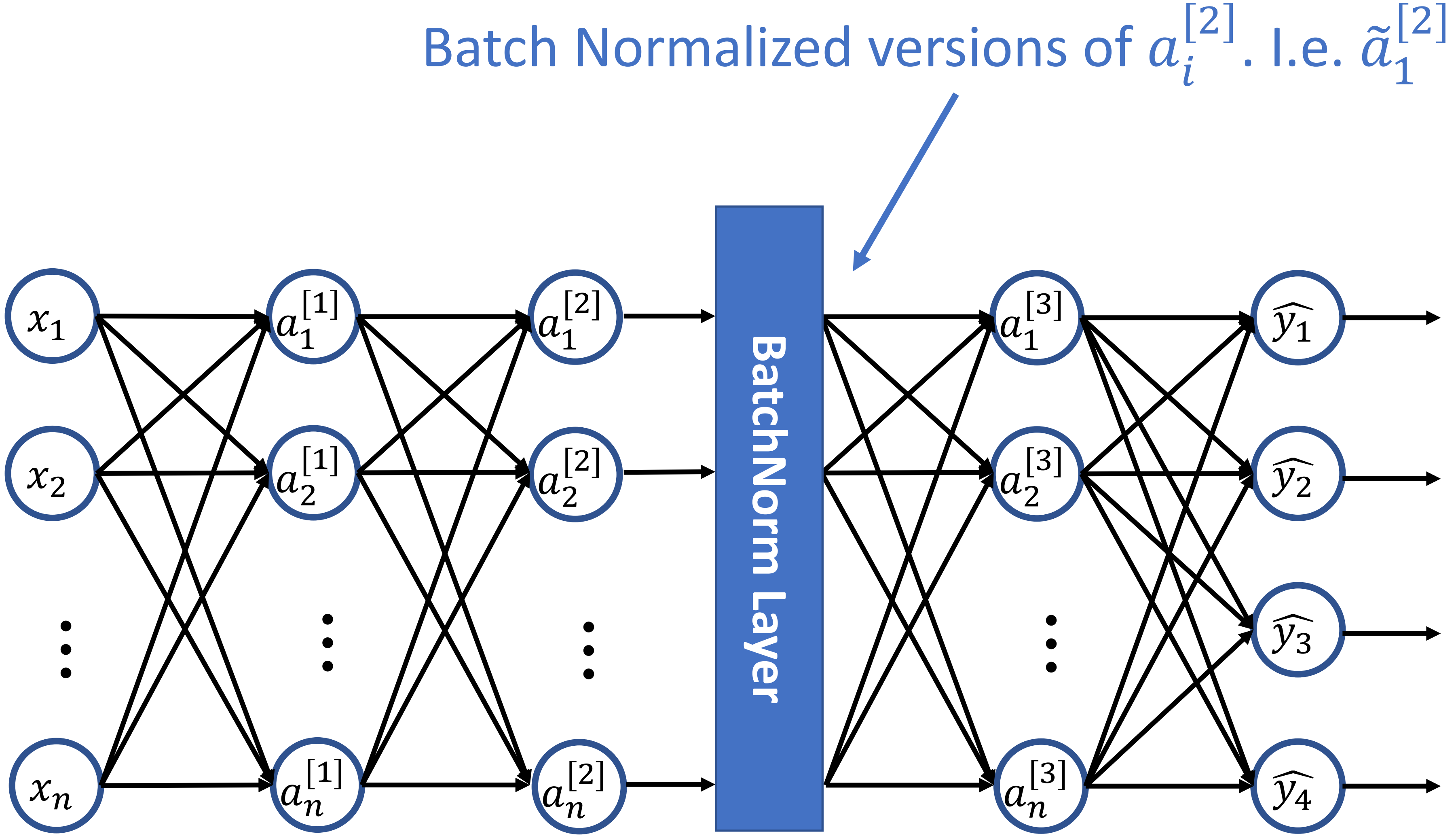
$$\widetilde{x_i} = \gamma_i x_i' + \beta_i$$



How BatchNorm Layer Fits Into Network

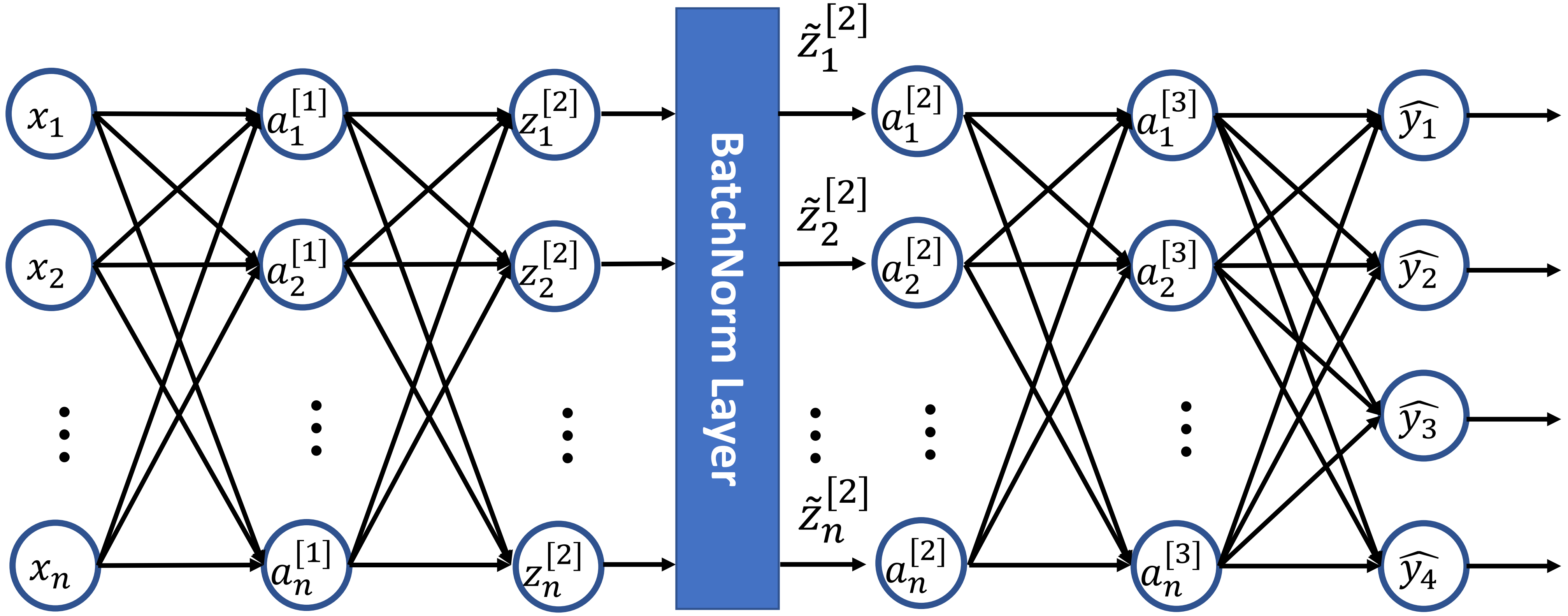


How BatchNorm Layer Fits Into Network



How BatchNorm Layer Fits Into Network

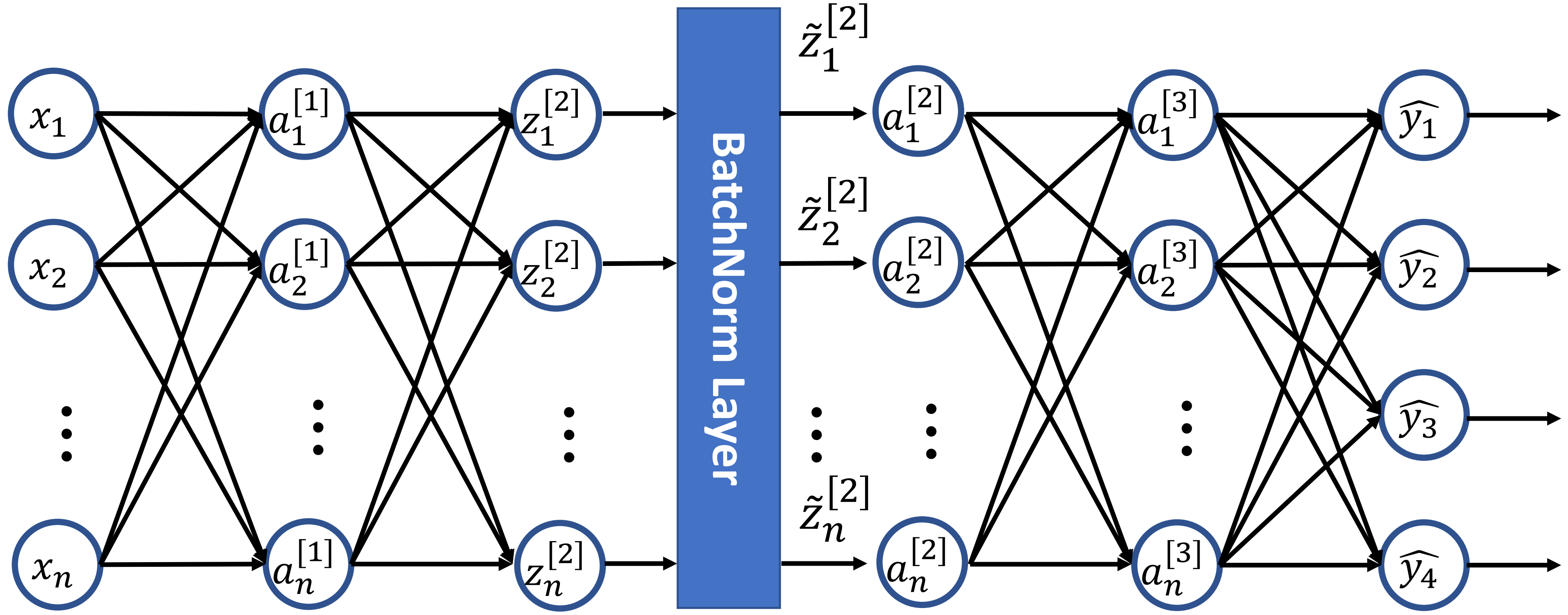
Can also perform Batch Normalization before the nonlinear activation



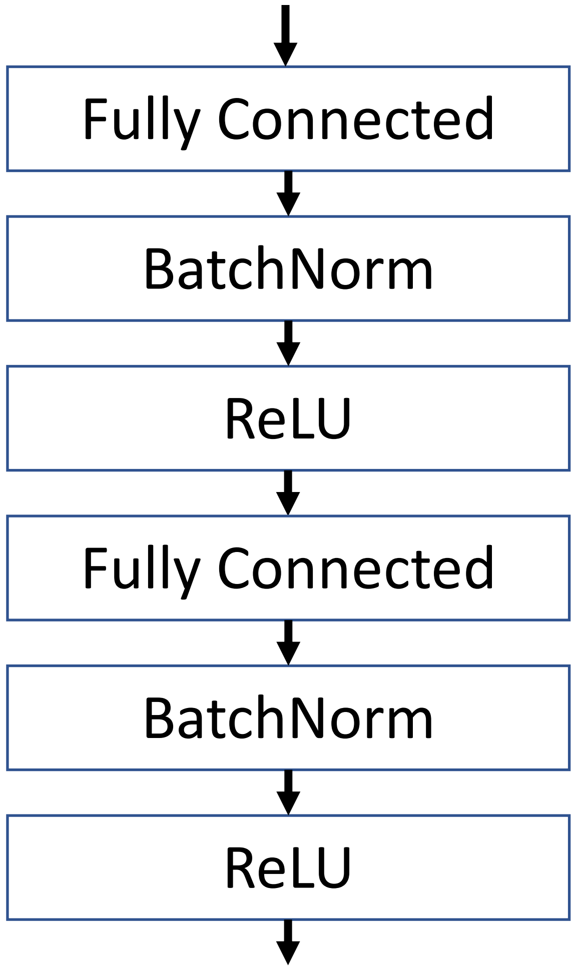
How BatchNorm Layer Fits Into Network

Can also perform Batch Normalization before the nonlinear activation

This is what most people do and it works well

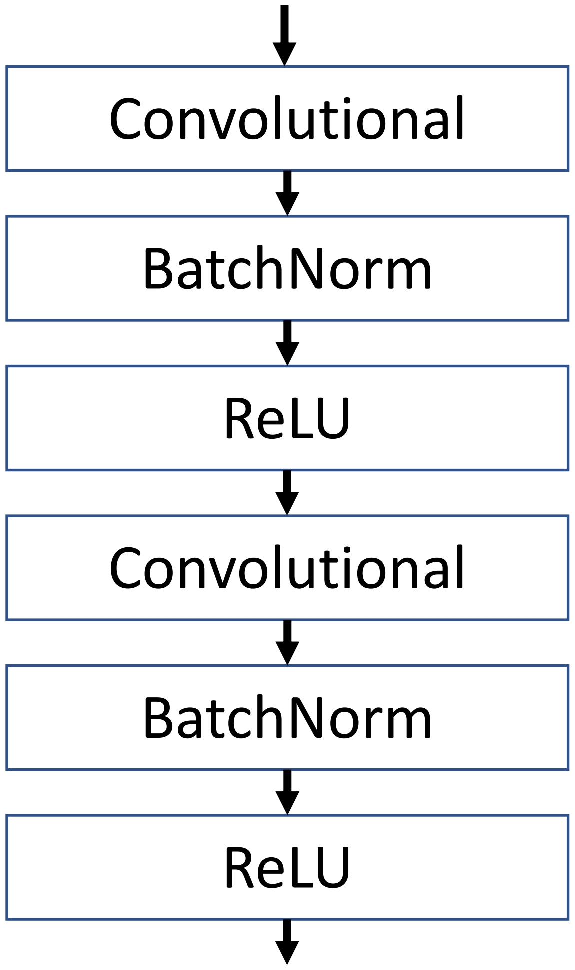


BatchNorm and Fully-Connected Layers



Term	Symbol	Shape
BatchNorm Inputs	x	(n, m)
Mean	μ	$(n,)$
Stddev/Variance	σ	$(n,)$
Scale param	γ	$(n,)$
Shift param	β	$(n,)$
Normalized Inputs	x'	(n, m)
Batch Normalized Inputs	$\tilde{x}$	(n, m)

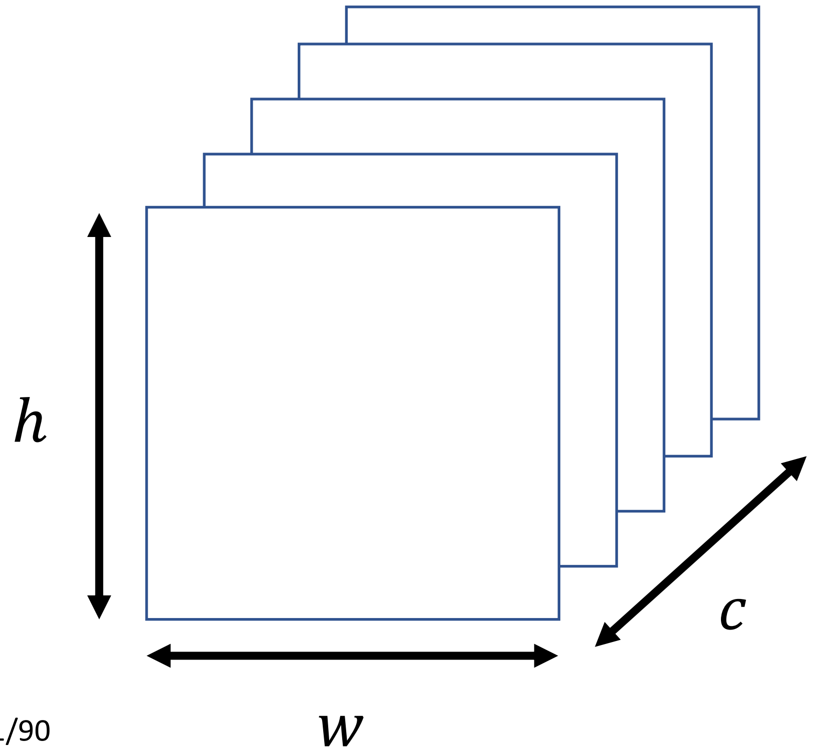
BatchNorm and Convolutional Layers



Term	Symbol	Shape
BatchNorm Inputs	x	(h, w, c, m)
Mean	μ	$(c,)$
Stddev/Variance	σ	$(c,)$
Scale param	γ	$(c,)$
Shift param	β	$(c,)$
Normalized Inputs	x'	(h, w, c, m)
Batch Normalized Inputs	$\tilde{x}$	(h, w, c, m)

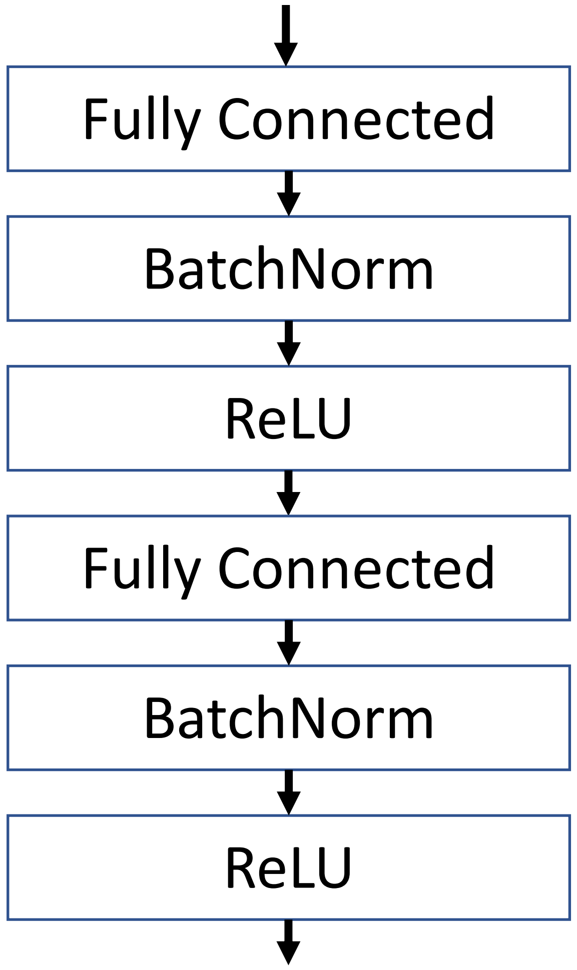
BatchNorm and Convolutional Layers

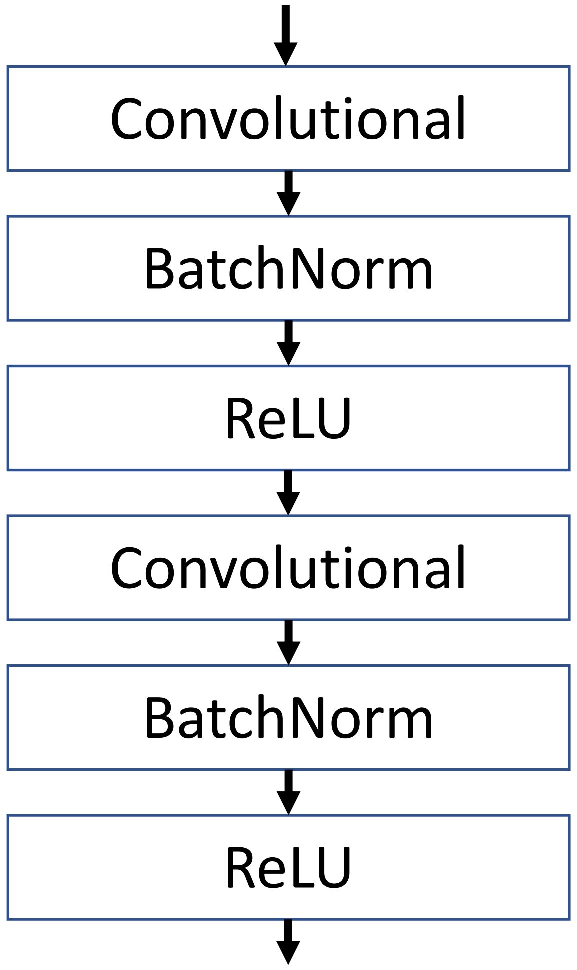
- Calculate μ and σ for each channel (e.g. activation map)
- Perform batch normalization over each element in a single activation map using its own μ and σ



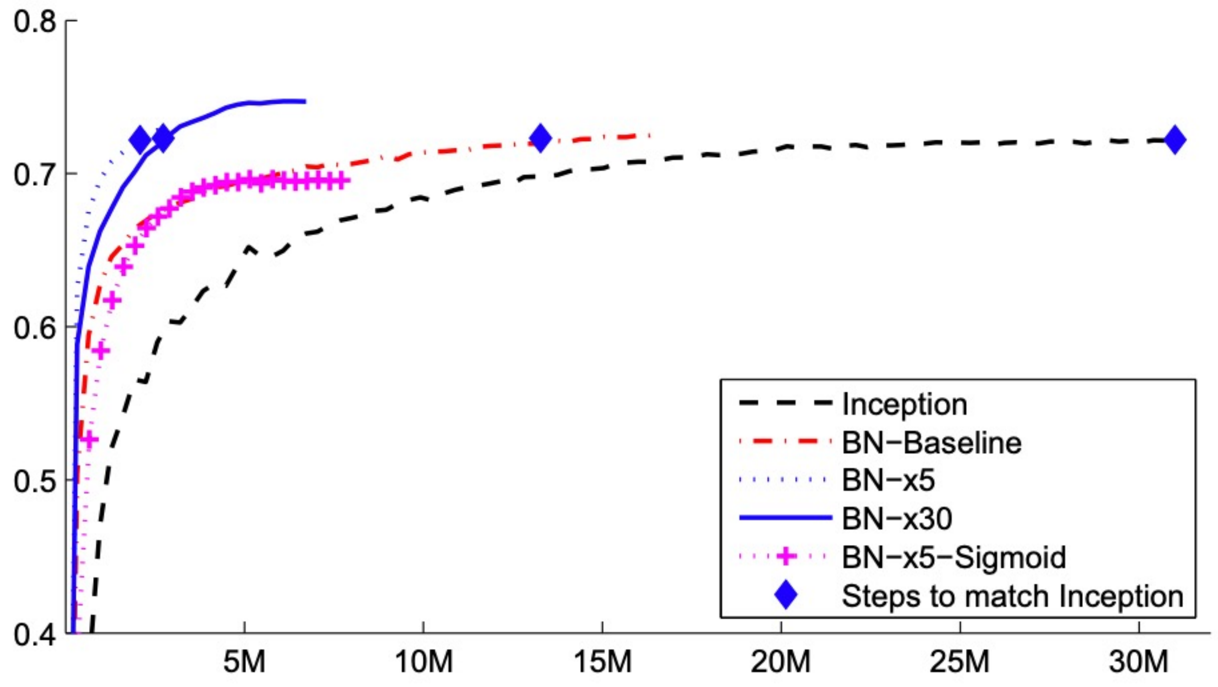
Term	Symbol	Shape
BatchNorm Inputs	x	(h, w, c, m)
Mean	μ	$(c,)$
Stddev/Variance	σ	$(c,)$
Scale param	γ	$(c,)$
Shift param	β	$(c,)$
Normalized Inputs	x'	(h, w, c, m)
Batch Normalized Inputs	$\tilde{x}$	(h, w, c, m)

Summary – Where to Put Batch Norm Layers





BatchNorm Can Speed up Training



Model	Steps to 72.2%	Max accuracy
Inception	$31.0 \cdot 10^6$	72.2%
BN-Baseline	$13.3 \cdot 10^6$	72.7%
BN-x5	$2.1 \cdot 10^6$	73.0%
BN-x30	$2.7 \cdot 10^6$	74.8%
BN-x5-Sigmoid		69.8%

- Simply adding BatchNorm layers can help
- Can use larger learning rate
- See original paper for more details

” Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift”, Ioffe and Szegedy, 2015, https://arxiv.org/pdf/1502.03167.pdf

BatchNorm at Prediction Time

- BatchNorm is a function of all samples in the mini-batch!

$$\mu_i = \frac{1}{m}\sum_{j=1}^{m} x_i^{(j)} \qquad \sigma_i{}^2 = \frac{1}{m}\sum_{j=1}^{m}\left(x_i^{(j)} - \mu_i\right)^2$$

- What do you do when the model is trained and deployed and you want to use it for prediction?
- Consider you are trying to use the model on one input at a time
- You can't meaningfully compute a mean and variance of one sample.

BatchNorm at Prediction Time

- For each μ and σ computed during training keep a moving average (e.g. Exponentially Weighted Average)
- Uses these averaged values of μ and σ during prediction time

Slight Regularization Effect

- Mean and variance on mini-batch is only an approximation to the actual mean and variance compared to the entire training set activations
- This introduces noise
- Unintended regularization (reduce overfitting) effect. More about regularization next!

BatchNorm Summary

- Makes training a lot faster/easier for deep networks
- Optimization problem is now less sensitive to learning rate, and weight initialization
- Extra processing at inference time
- People still don’t understand exactly why it works…

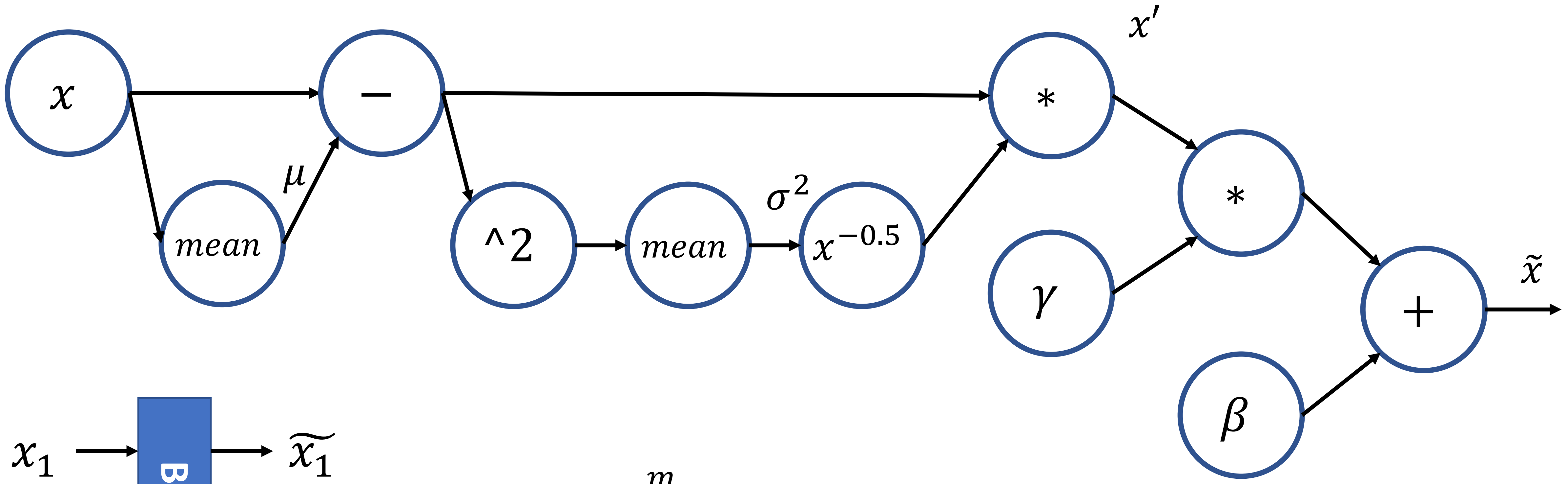
Why Does BatchNorm Work? Revisited

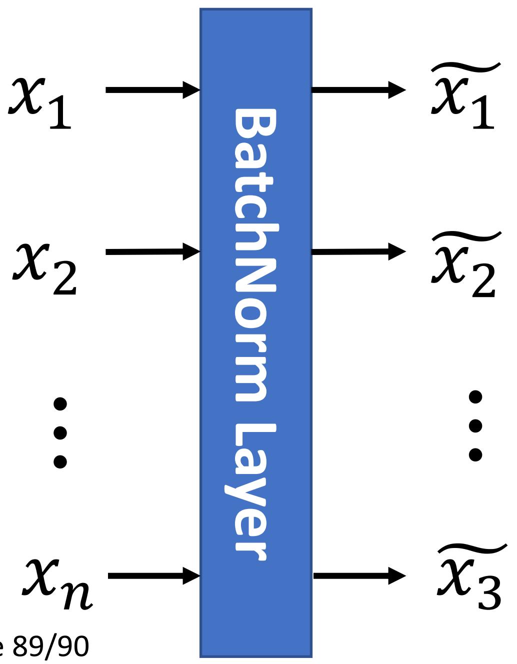
- Helps stabilize a layer’s output’s distribution to make it easier for next layer to optimize upon
- “Reduces Internal Covariate Shift”
- Some recent research seems to show that this is not actually the case

- More recent hypotheses:
  - Smooths the objective landscape
  - Length-Direction Decoupling

“Exponential convergence rates for Batch Normalization: The power of length-direction decoupling in non-convex optimization”, Kohler et al, 2018, https://arxiv.org/abs/1805.10694
“How Does Batch Normalization Help Optimization?”, Santurkar et al, 2018, https://arxiv.org/abs/1805.11604
“A Mean Field Theory of Batch Normalization”, Yang et al, 2019, https://arxiv.org/abs/1902.08129

Backpropagation Through BackNorm Layer





$$\mu_i = \frac{1}{m}\sum_{j=1}^{m} x_i^{(j)}$$

$$\sigma_i{}^2 = \frac{1}{m}\sum_{j=1}^{m}\left(x_i^{(j)} - \mu_i\right)^2 \qquad x_i{}' = \frac{x_i - \mu_i}{\sigma_i} \qquad \widetilde{x_i} = \gamma_i x_i' + \beta_i$$

Learning Objectives

Introduce a number of commonly used techniques to improve the training optimization process

- Learning Rate Schedules
- Parameter Initialization
- Data Preprocessing/Feature Normalization
- Batch Norm