DEEP LEARNING FOR VISUAL RECOGNITION

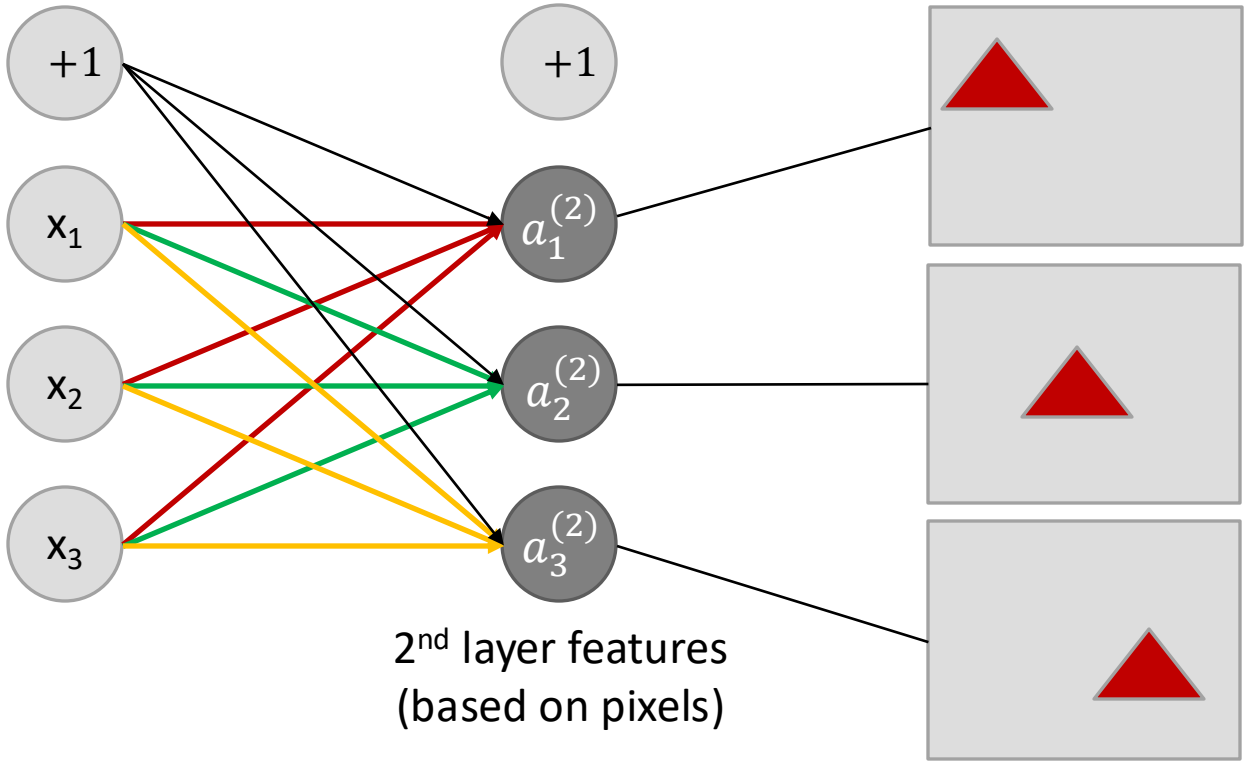
Lecture 5 – Training Convolutional Neural Networks (part 1)

Henrik Pedersen, PhD
Part-time lecturer
Department of Computer Science
Aarhus University
hpe@cs.au.dk

Today’s agenda

- You will learn about training ConvNets.
- Topics
  - Activation functions
  - Data preprocessing
  - Weight initialization
  - Batch normalization

Where we are now

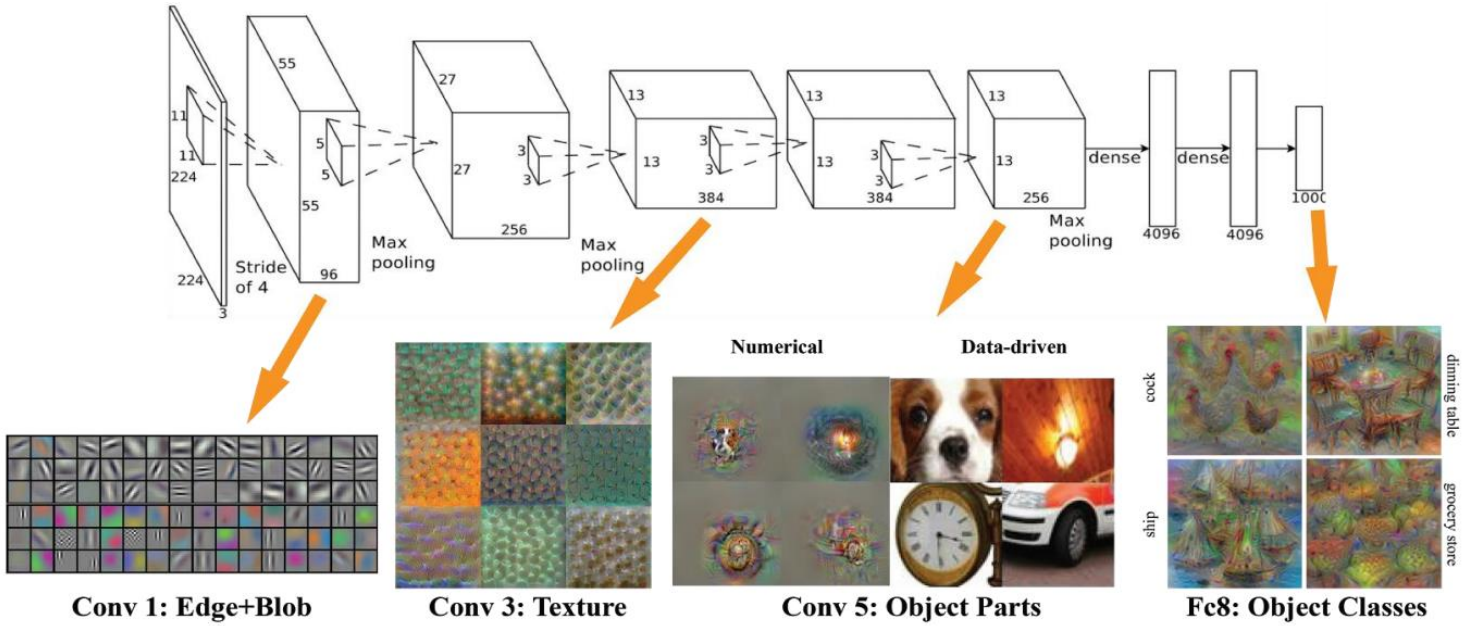


Fully connected networks:
Not well suited for images

Disadvantage: Layers are "fully connected", meaning that each neuron receives input from all neurons of the previous layer. This is an inefficient way to detect local features (like object parts in an image).

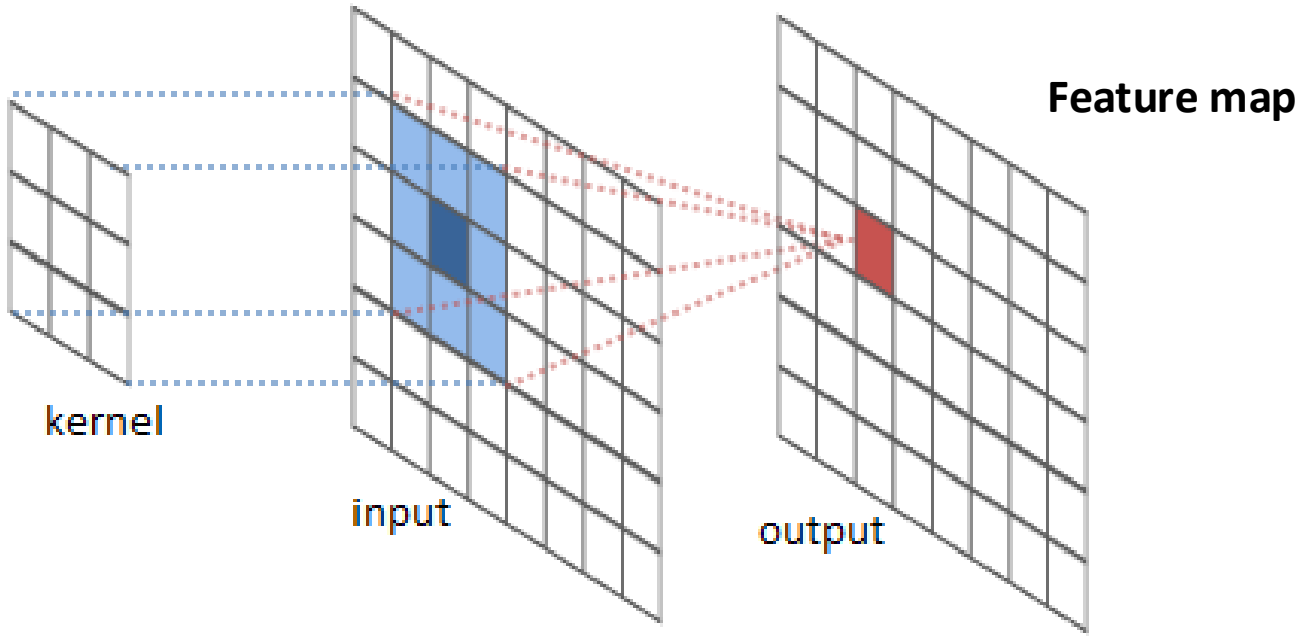
Where we are now

- Convolutional Neural Networks (ConvNets or CNNs).
- Better suited for images due to **sparse interactions** and **parameter sharing**.



Where we are now

- The **output neuron** is connected only to those pixels in **a certain region**.
- The output is called a **feature map** (or activation map).



Where we are now

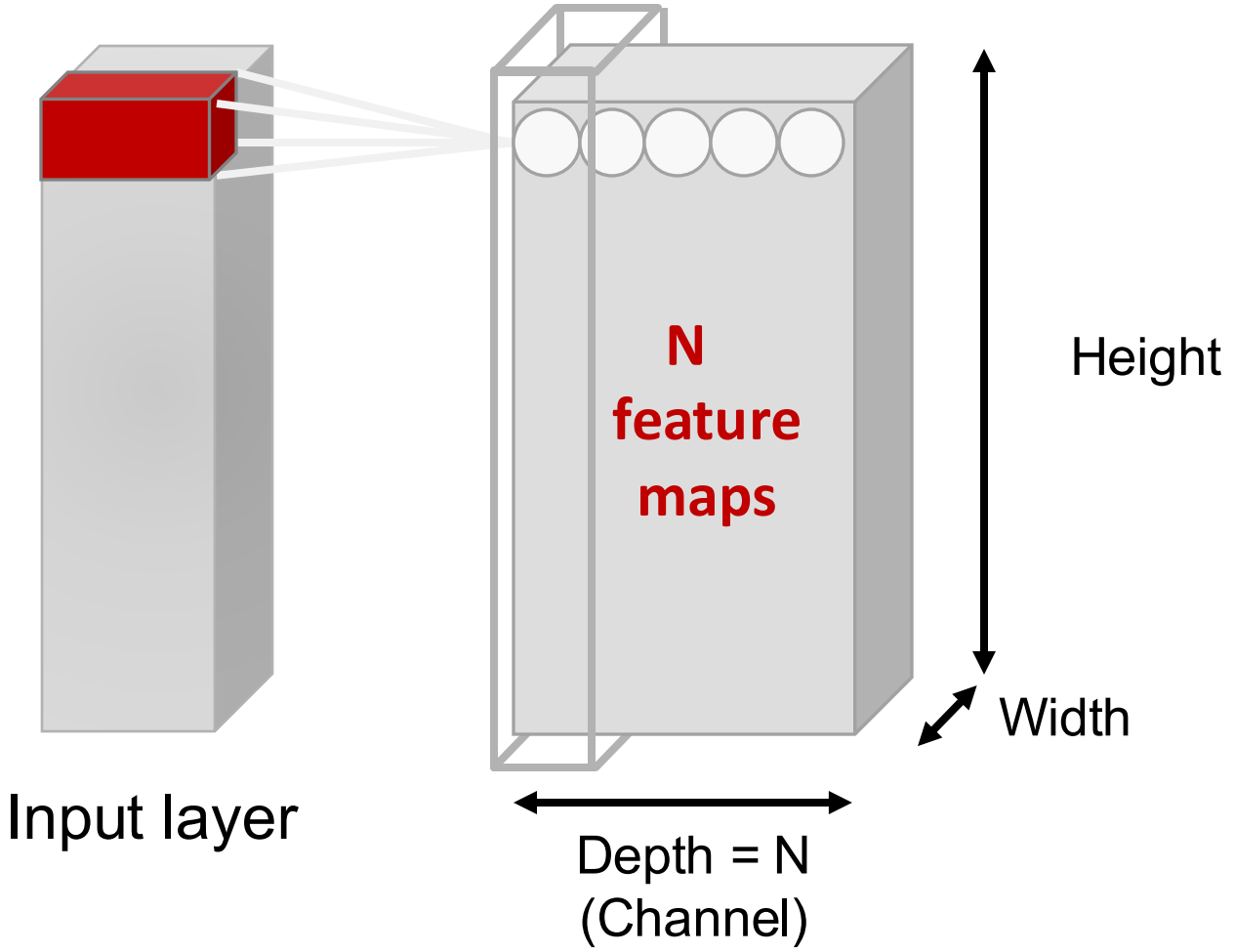
One output channel per filter, referred to as a **feature map**.

The filter connectivity is

- Local in **space** (height and width)
- Full in **depth** (all 3 RGB channels)

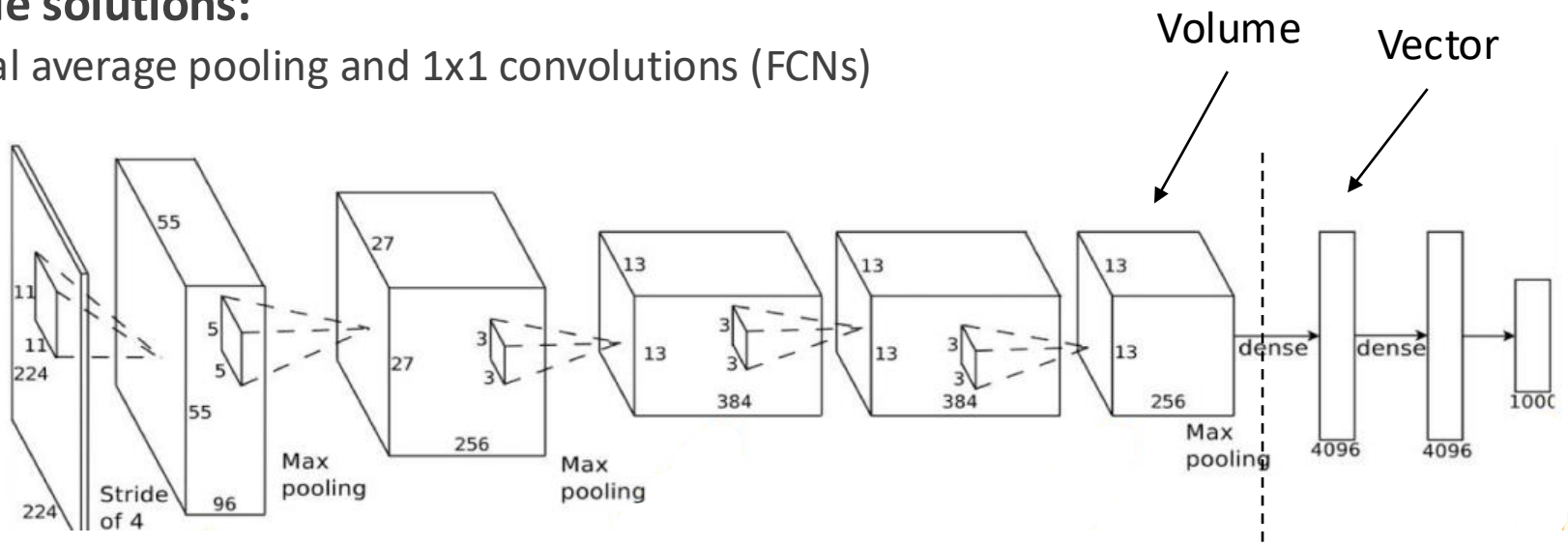
Advantages over fully connected neural networks:

- Sparse interactions
- Parameter sharing
- Translation equivariance



Where we are now

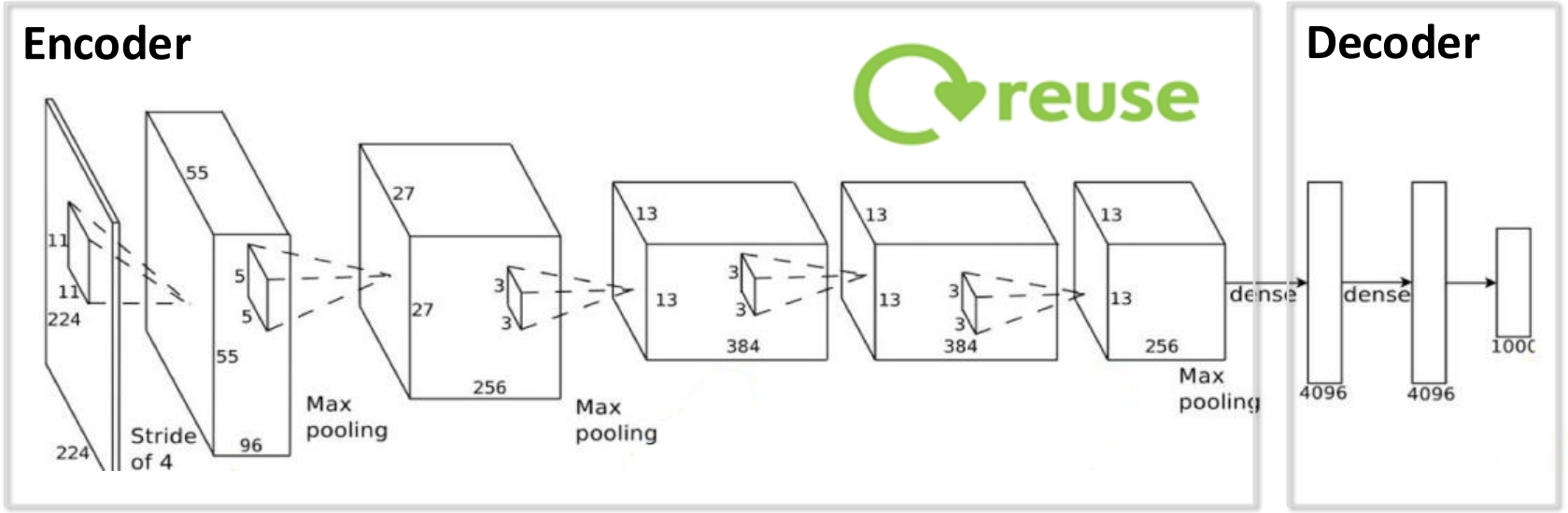
- We often have one or more fully connected (FC) layers **at the end of CNNs**.
- **Problem:**
  - Convolutional base/encoder/backbone outputs a **volume**, while FC layers expect **vectors** as input.
  - This becomes a problem when the size of the input image changes.
- **Possible solutions:**
  - Global average pooling and 1x1 convolutions (FCNs)



Convolutional base/backbone/encoder

Fully connected NN (classifier)

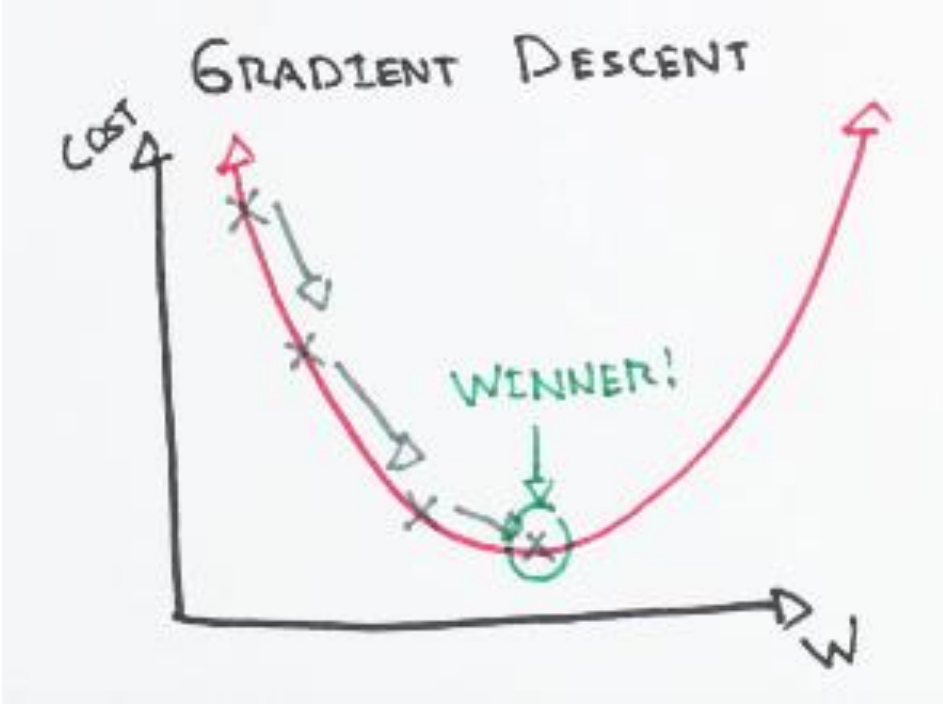
Where we are now



- We often reuse a convolutional base (encoder), pretrained on ImageNet, and add our own decoder on top.
- This is called **transfer learning**.
- Makes training a lot faster.
- In classification problems, the decoder is usually a fully connected neural network.

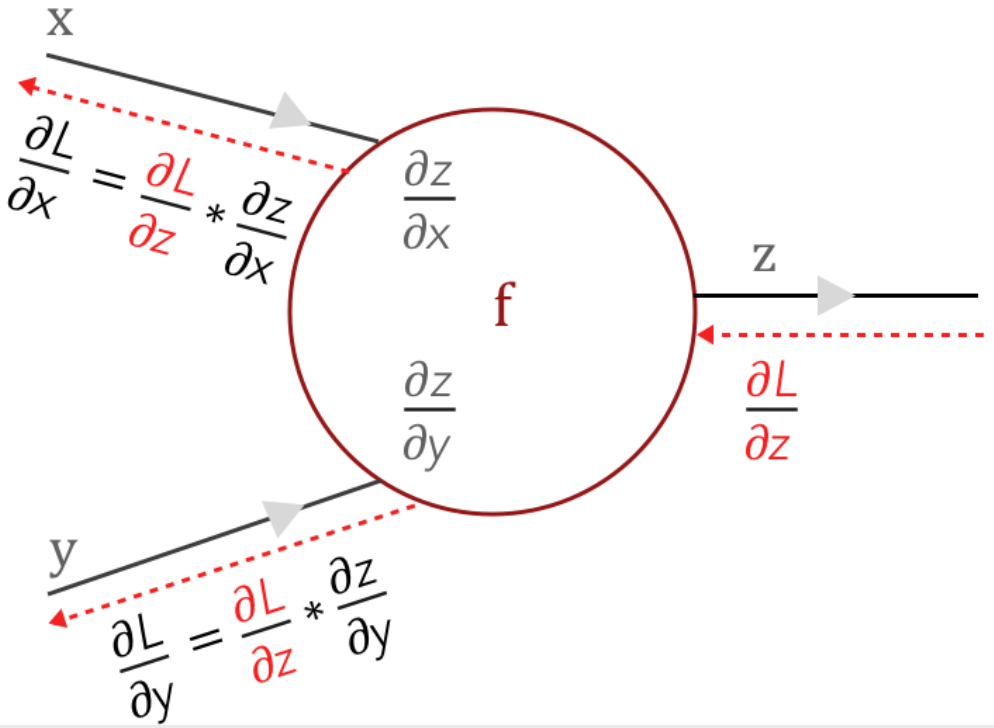
Where we are now

- Network parameters are learned through optimization.
- We define a **loss function** and use **gradient descent** to minimize it.



Computational graphs

- Backprop using **computational graphs**.
- This is what all modern deep learning frameworks use.
- Enables **automatic differentiation**.
- Performs both forward and backward pass.
- Calculates partial derivatives based on the chain rule.



$\frac{\partial z}{\partial x}$ & $\frac{\partial z}{\partial y}$ *are local gradients*

$\frac{\partial L}{\partial z}$ *is the loss from the previous layer which has to be backpropagated to other layers*

Computational graphs – example

- Consider the equation $f(x, y, z) = (x + y)z$

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q} * \frac{\partial q}{\partial x} = -4 * 1 = -4$$

$$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial q} * \frac{\partial q}{\partial y} = -4 * 1 = -4$$

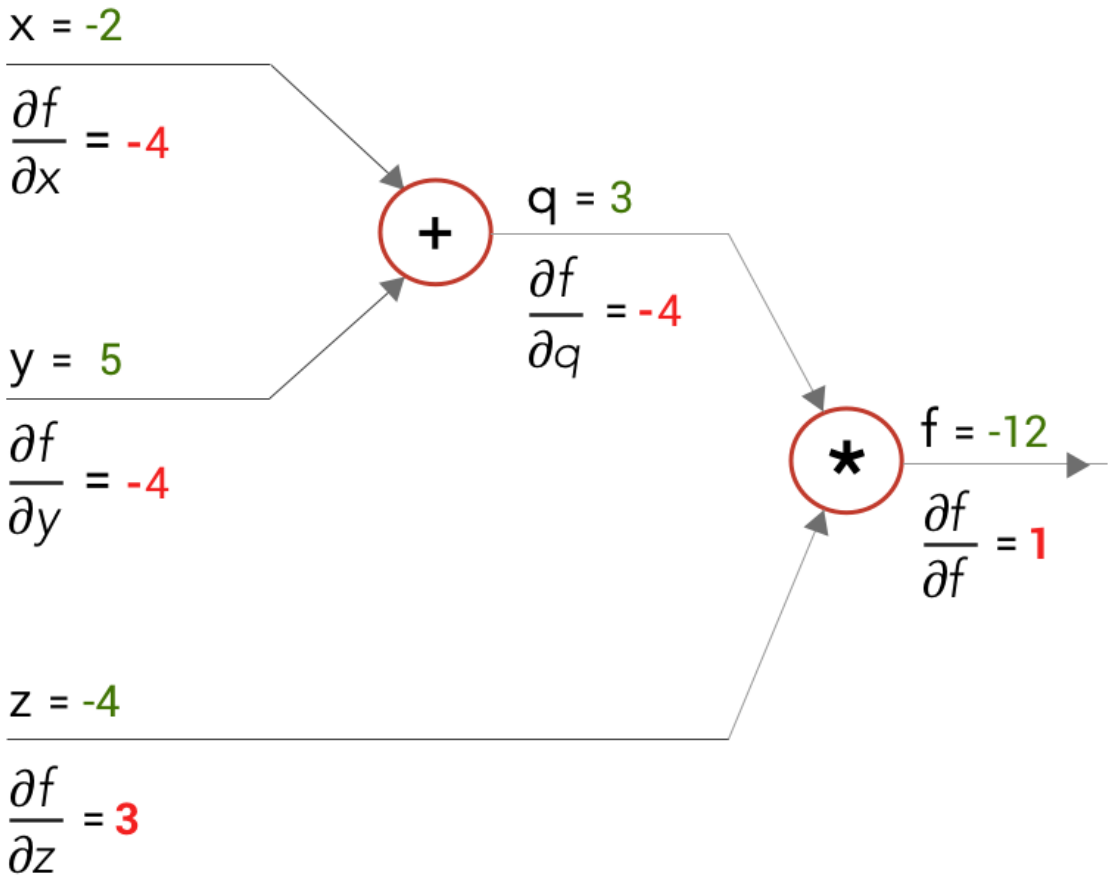
$f = q * z$

$\frac{\partial f}{\partial q} = z \mid z = -4$

$\frac{\partial f}{\partial z} = q \mid q = 3$

$q = x + y$

$\frac{\partial q}{\partial x} = 1 \quad \frac{\partial q}{\partial y} = 1$



Training pipeline

- Set up network architecture = computational graph
- Train using mini-batch Stochastic Gradient Descent (SGD)
- Loop
  - **Sample** a batch of images
  - **Forward** prop it through the network (computational graph), and get loss
  - **Backprop** to calculate the gradient
  - **Update** parameters using the gradient

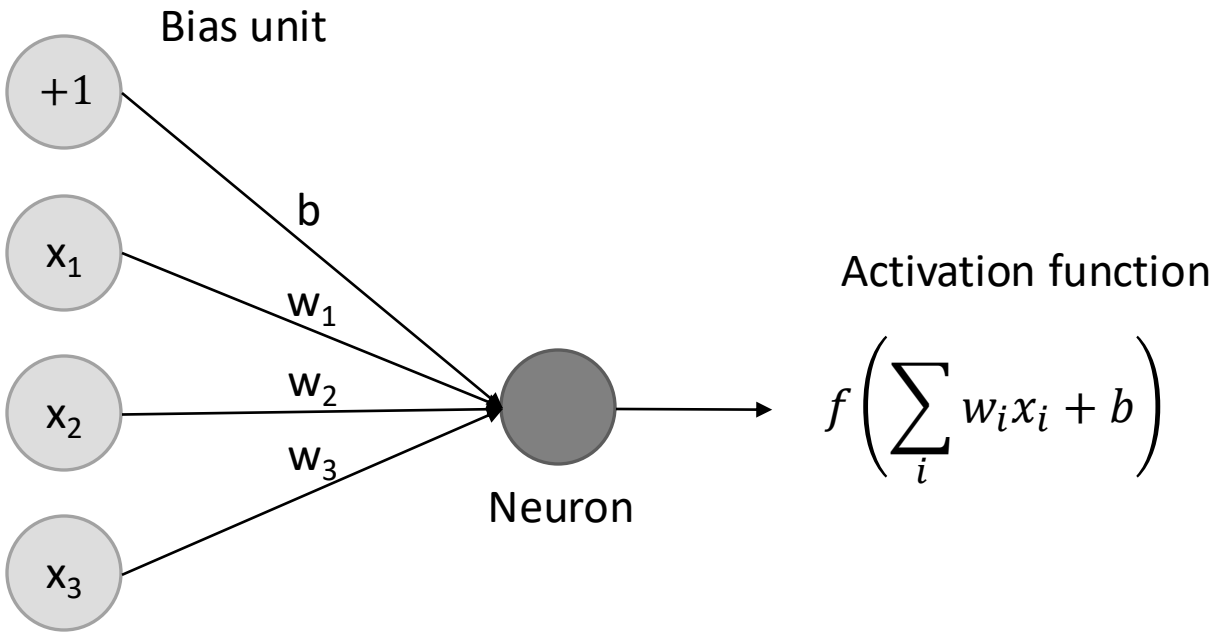
Next: Training ConvNets

- Activation functions
- Data preprocessing
- Weight initialization
- Batch normalization
- Stochastic Gradient Descent (SGD)
- SGD extensions: Momentum, AdaGrad, RMSProp, and Adam
- Learning rate decay and cycling
- Regularization: Early stopping, weight decay, dropout, data augmentation
- Hyperparameter search
- Transfer learning

Activation functions

Activations functions

Activation functions are necessary to make our model non-linear, and we need non-linearity because …

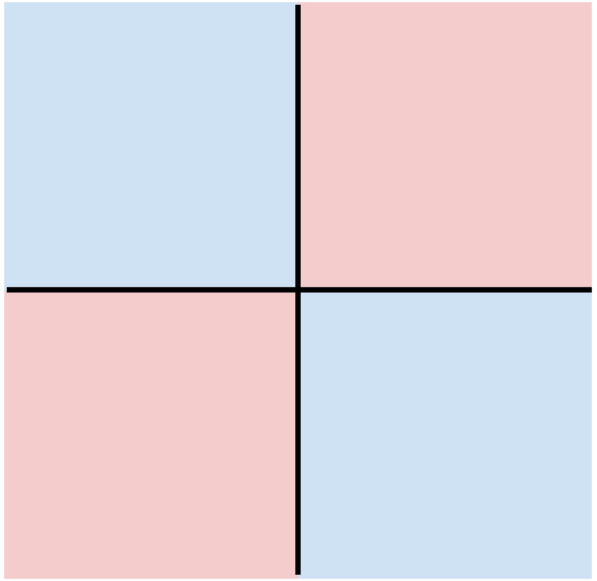


Hard cases for linear classifier

… linear models have a linear decision boundary, which is typically insufficient.

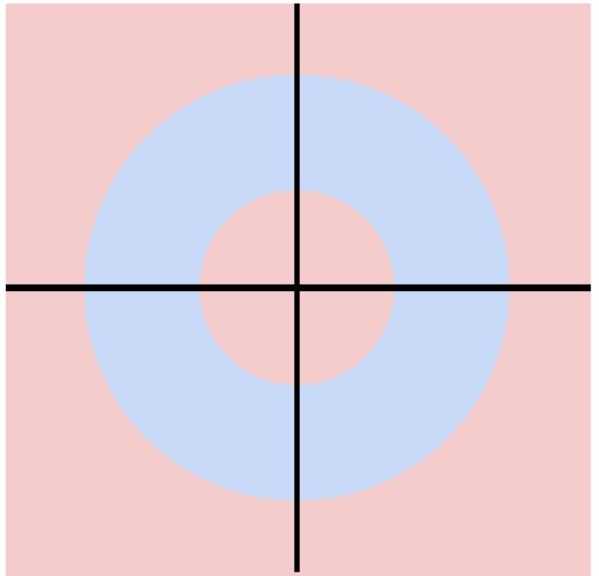
Class 1:
First and third quadrants

Class 2:
Second and fourth quadrants



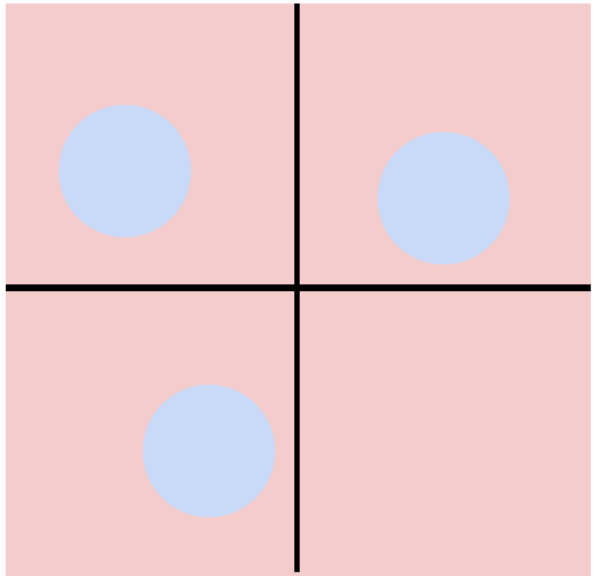
Class 1:
1 <= L2 norm <= 2

Class 2:
Everything else



Class 1:
Three modes

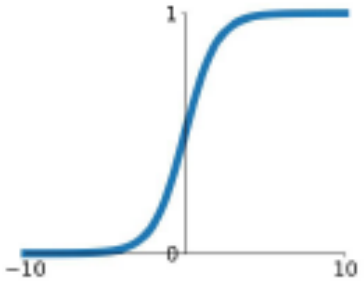
Class 2:
Everything else



Examples of activation functions

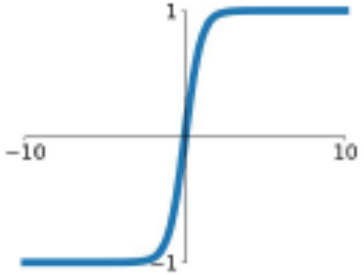
Sigmoid

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



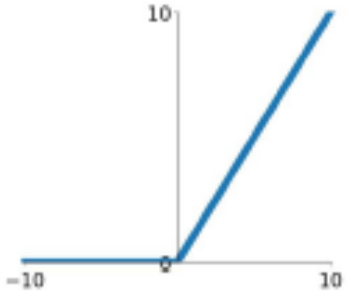
tanh

$$\tanh(x)$$



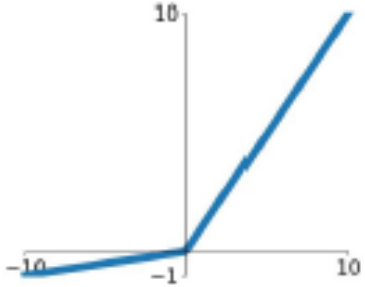
ReLU

$$\max(0, x)$$



Leaky ReLU

$$\max(0.1x, x)$$

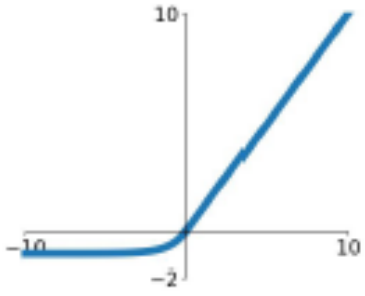


Maxout

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

ELU

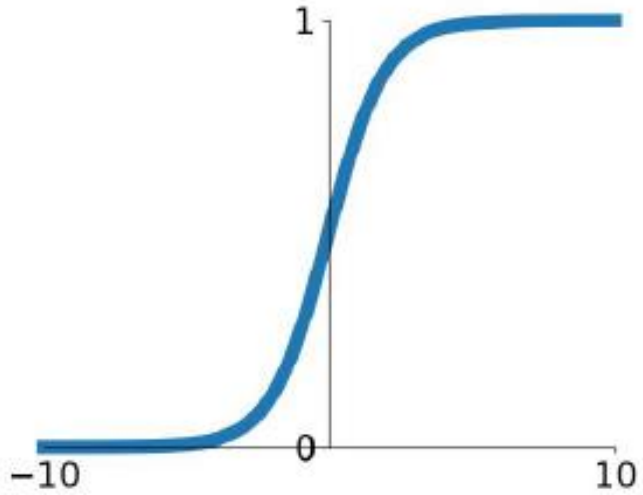
$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



Sigmoid

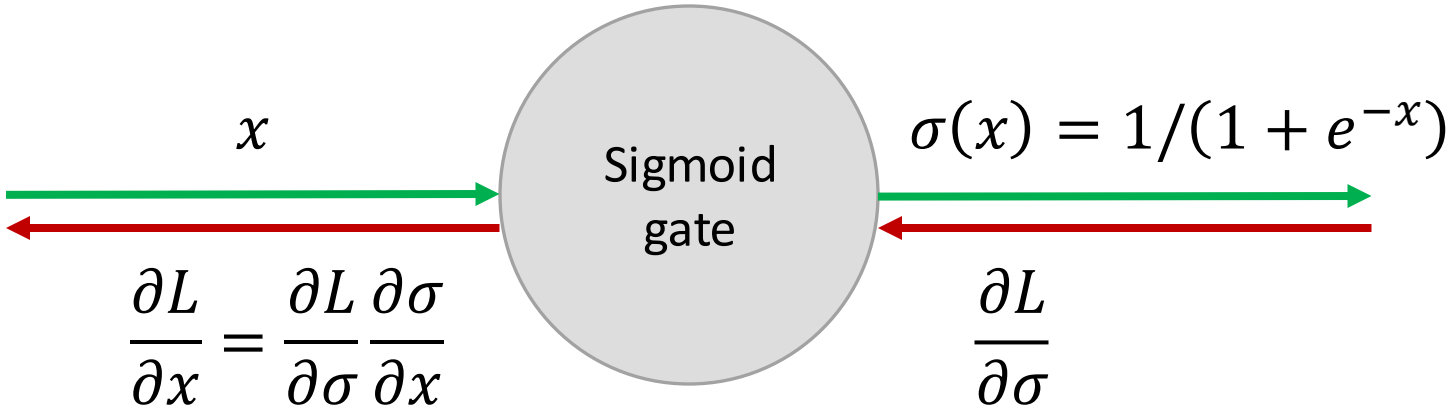
- Squashes numbers to range [0, 1].
- Useful for modelling probabilities (e.g., logistic regression).
- Historically popular, because it has a nice interpretation as a saturating “firing rate” of a neuron.
- **3 problems**
  1. Saturated neurons “kill the gradients” (see Lecture 3).
  2. Sigmoid outputs are not zero-centered
  3. exp() is a bit computational expensive

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



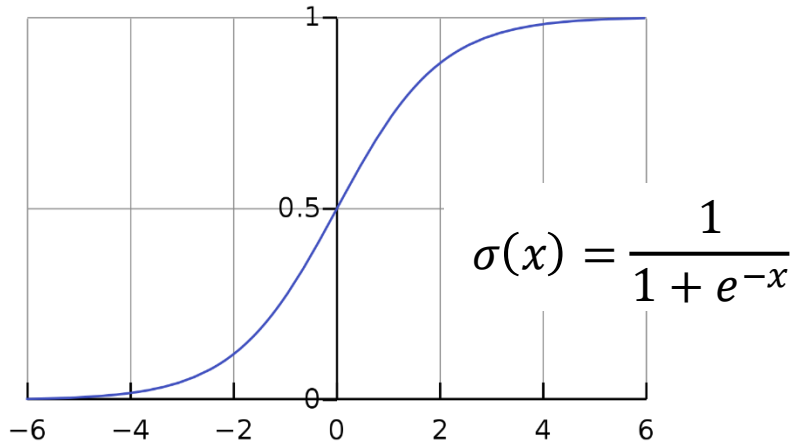
Sigmoid

Saturated neurons “kill the gradients”



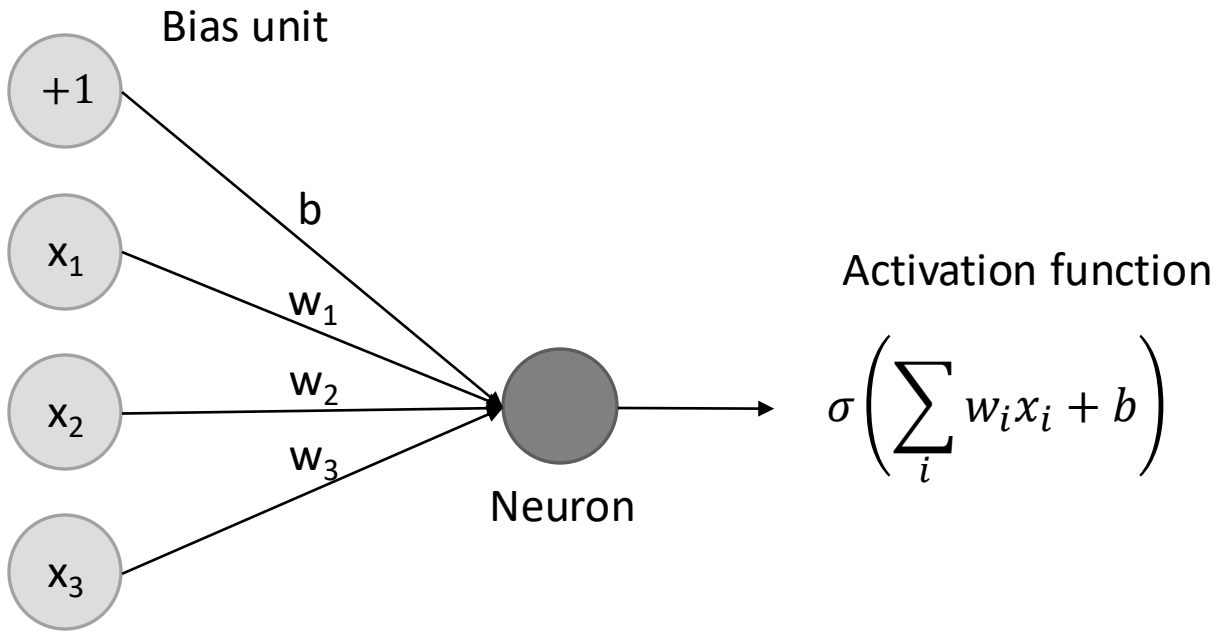
Saturation occurs when:

$$\frac{\partial \sigma}{\partial x} = \sigma(x)\big(1 - \sigma(x)\big) \approx 0 \quad \text{if } |x| \text{ is large} \rightarrow \frac{\partial L}{\partial x} \approx 0$$



Sigmoid outputs are not zero-centered

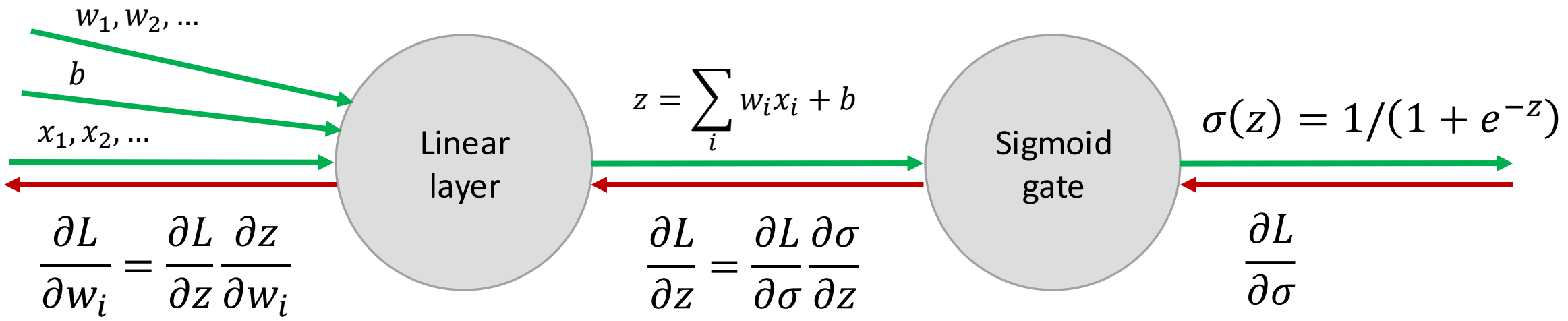
- Consider what happens when the input **x** to a neuron is always positive (as if produced by a σ).
- **Q:** What can we say about the gradient of the loss w.r.t. **w**?



Sigmoid outputs are not zero-centered

Computational graph of

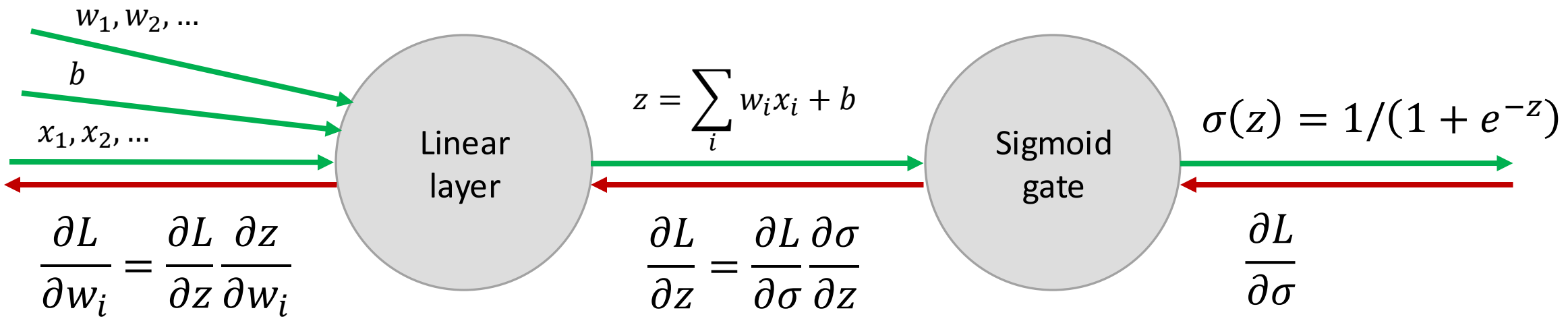
$$\sigma\left(\sum_i w_i x_i + b\right)$$



Sigmoid outputs are not zero-centered

Applying the chain rule again, we get:

$$\frac{\partial L}{\partial w_i} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial w_i} = \frac{\partial L}{\partial \sigma}\frac{\partial \sigma}{\partial z}x_i$$



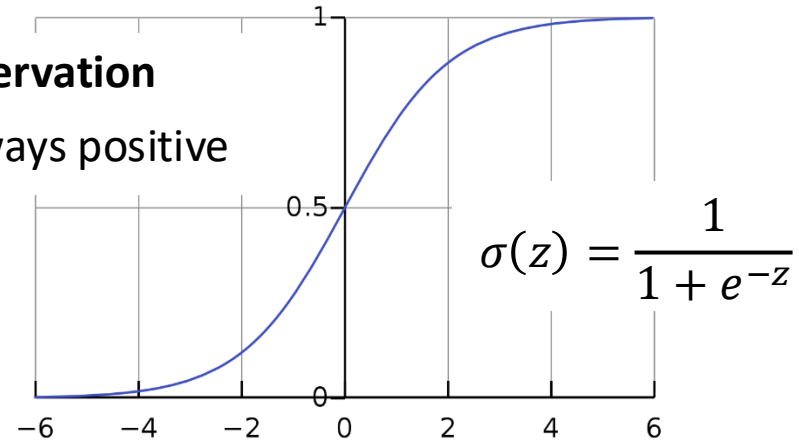
Sigmoid outputs are not zero-centered

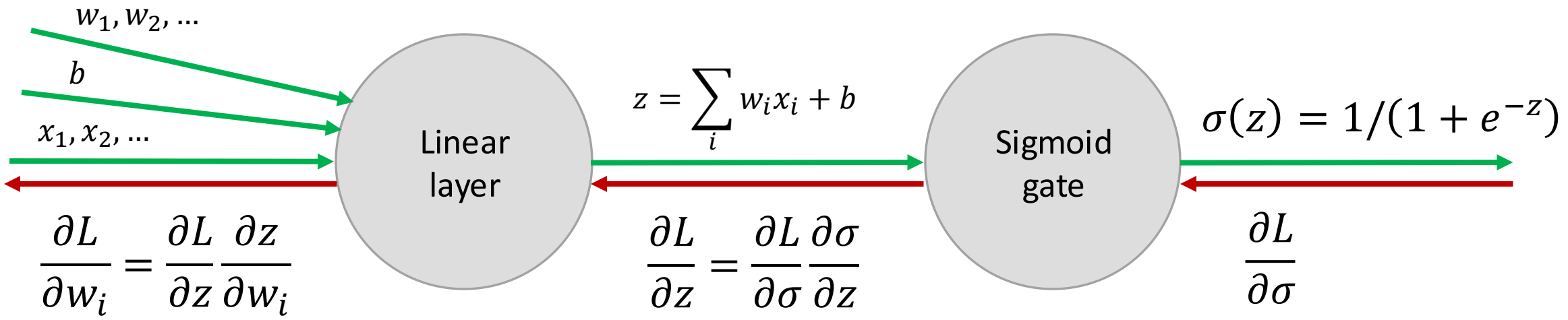
Applying the chain rule again, we get:

$$\frac{\partial L}{\partial w_i} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial w_i} = \frac{\partial L}{\partial \sigma}\frac{\partial \sigma}{\partial z}x_i$$

Observation

$\frac{\partial \sigma}{\partial z}$ is always positive





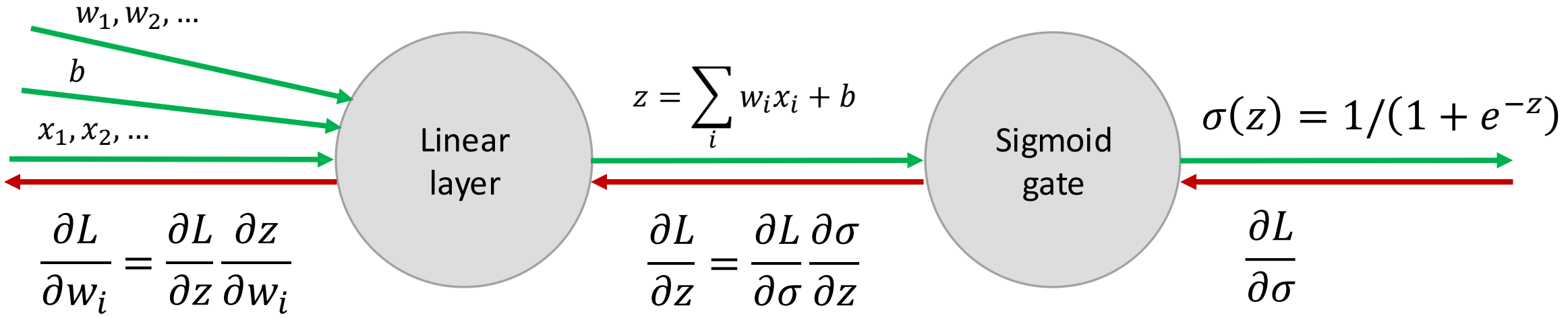
Sigmoid outputs are not zero-centered

Applying the chain rule again, we get:

$$\frac{\partial L}{\partial w_i} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial w_i} = \frac{\partial L}{\partial \sigma}\frac{\partial \sigma}{\partial z}x_i$$

Positive (according to our assumption)

Always positive

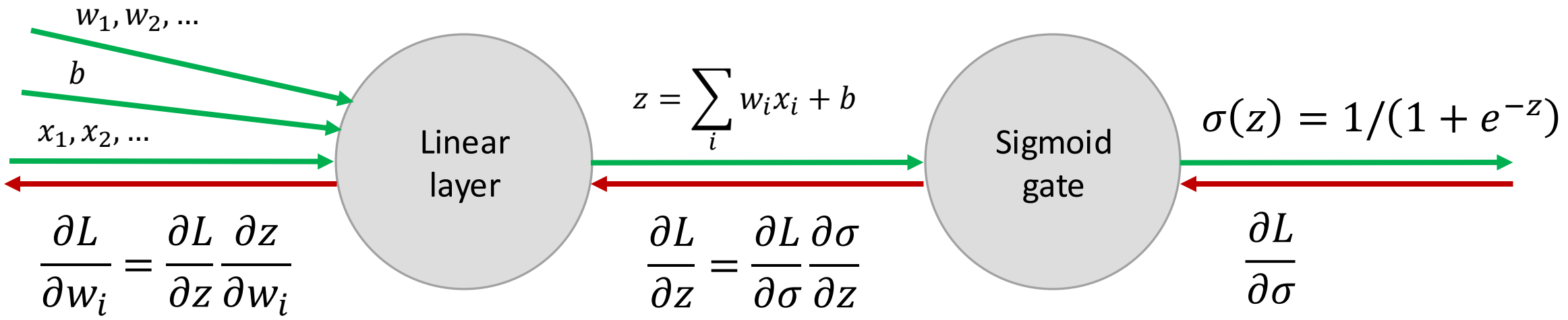


Sigmoid outputs are not zero-centered

Applying the chain rule again, we get:

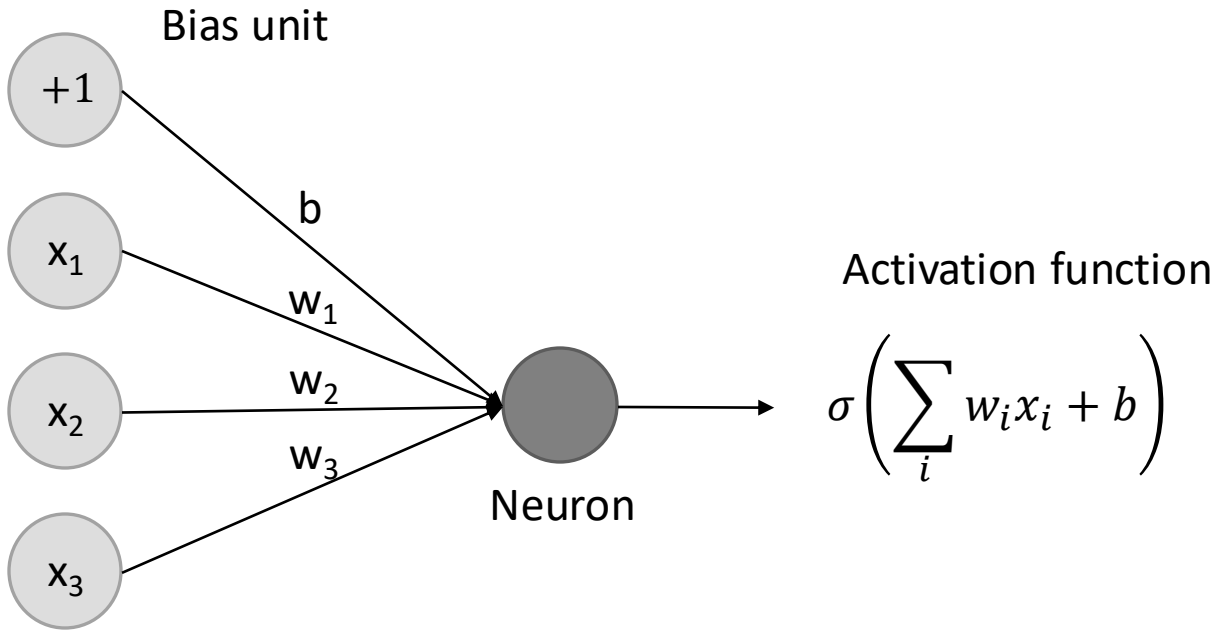
$$\frac{\partial L}{\partial w_i} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial w_i} = \frac{\partial L}{\partial \sigma}\frac{\partial \sigma}{\partial z}x_i$$

The sign of $\frac{\partial L}{\partial w_i}$ depends on the sign of the upstream gradient ($\frac{\partial L}{\partial \sigma}$).



Sigmoid outputs are not zero-centered

- Consider what happens when the input **x** to a neuron is always positive (as if produced by a σ).
- **Q:** What can we say about the gradient of the loss w.r.t. **w**?
- **A:** The partial derivatives are all positive or all negative*



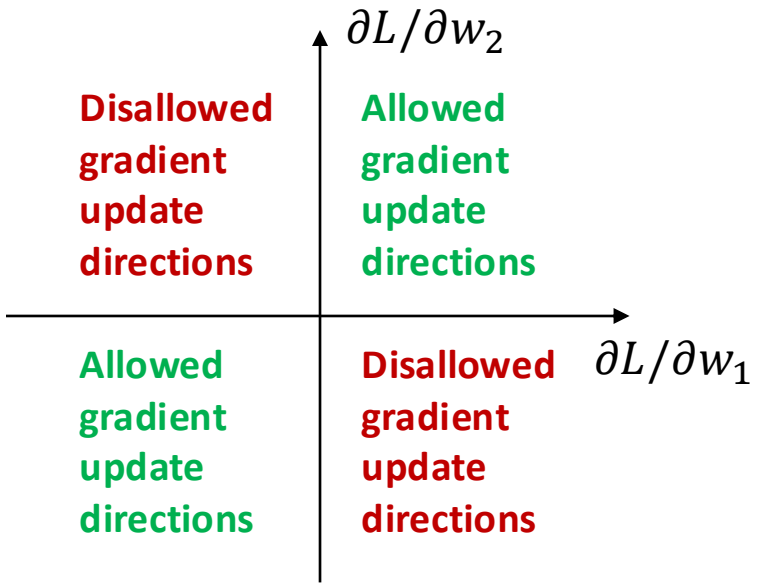
* At least for a single element. Mini-batches help…

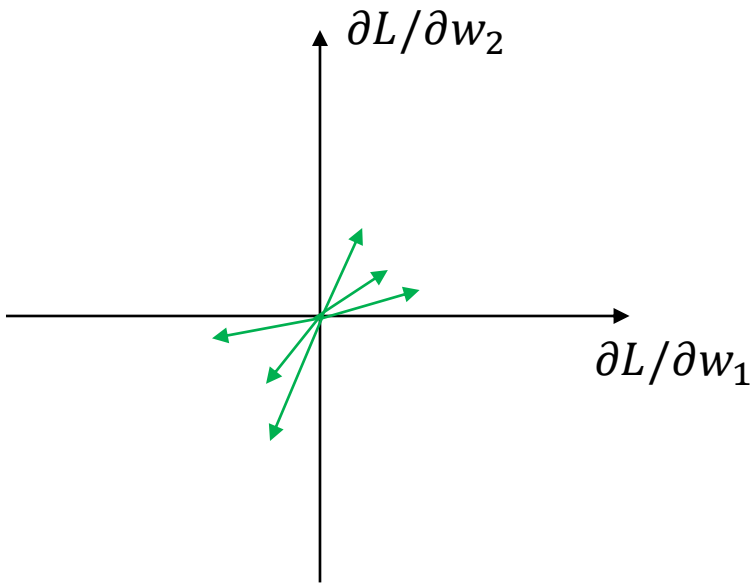
Sigmoid outputs are not zero-centered

- Consider what happens when the input **x** to a neuron is always positive (as if produced by a σ).
- **Q:** What can we say about the gradient of the loss w.r.t. **w**?
- **A:** The partial derivatives are all positive or all negative*
- **Implications:**
  - If the data coming into a neuron is always positive (e.g., x > 0 elementwise), then the gradient w.r.t. the weights **w** will during backpropagation become either all positive, or all negative (depending on the sign of the upstream gradient).
  - This could introduce undesirable **zig-zagging** dynamics in the gradient updates for the weights.
  - However, notice that once these gradients are added up across a mini-batch of data, the final update for the weights can have variable signs, somewhat mitigating this issue.

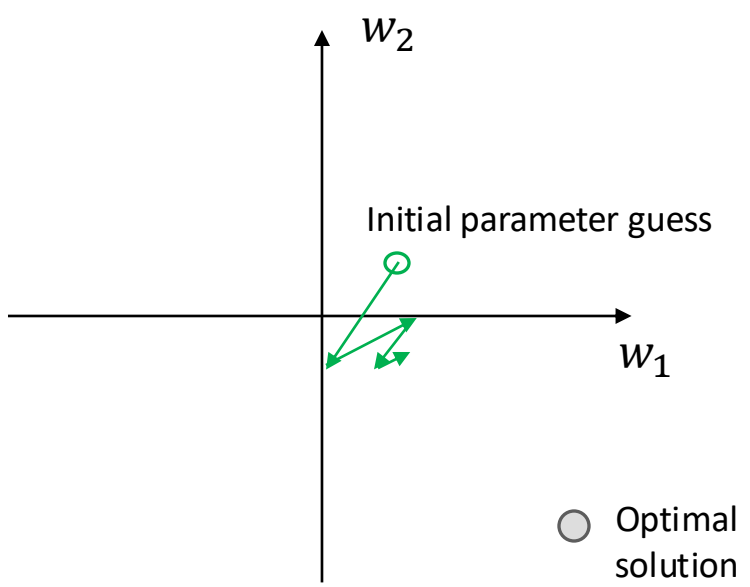
Sigmoid outputs are not zero-centered

- Consider what happens when the input **x** to a neuron is always positive (as if produced by a σ).
- **Q:** What can we say about the gradient of the loss w.r.t. **w**?
- **A:** The partial derivatives are all positive or all negative*





Examples of allowed gradients

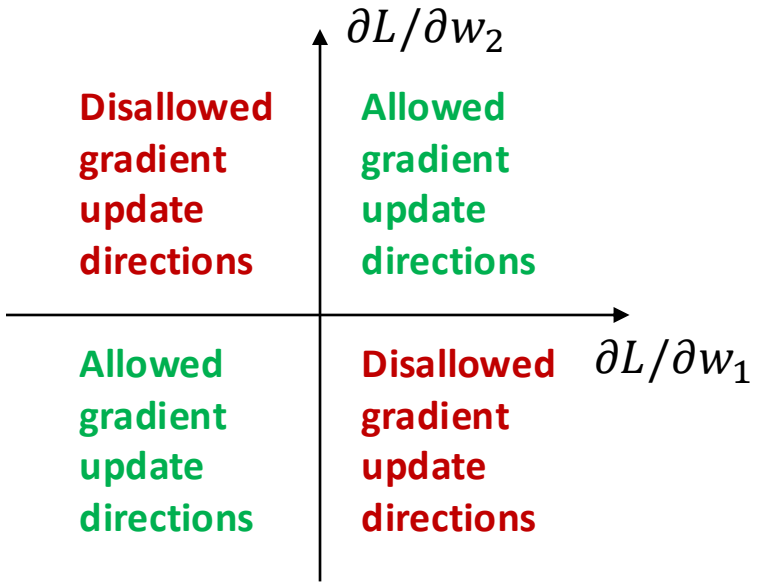


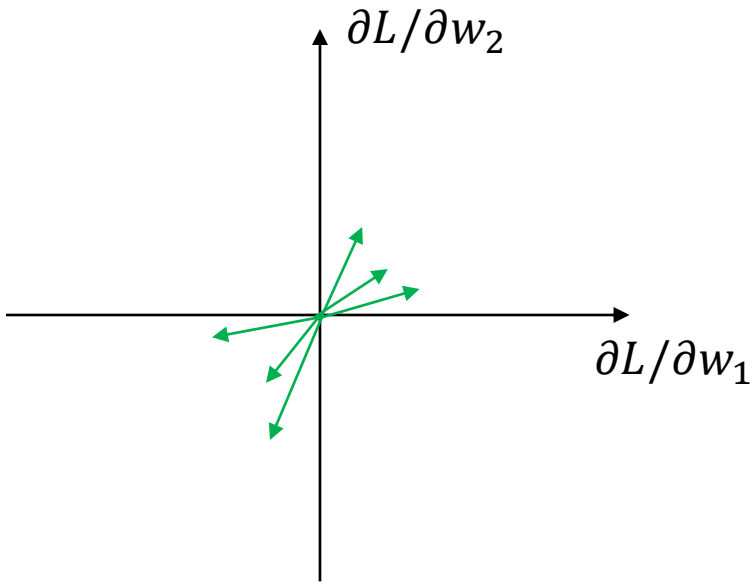
Zig-zagging = instability

Sigmoid outputs are not zero-centered

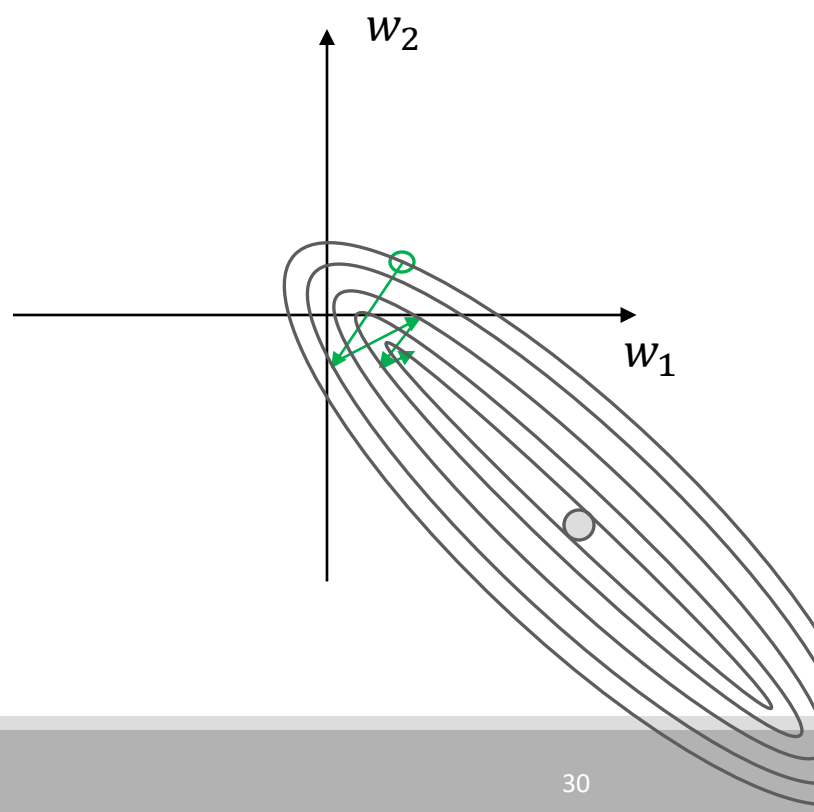
- Consider what happens when the input **x** to a neuron is always positive (as if produced by a σ).
- **Q:** What can we say about the gradient of the loss w.r.t. **w**?
- **A:** The partial derivatives are all positive or all negative*

Observation: The gradient is always perpendicular to the **iso-contour** of the loss landscape. By using sigmoid activations, we have effectively created a "ravine" in the loss landscape. Gradient descent sucks at navigating ravines 👎



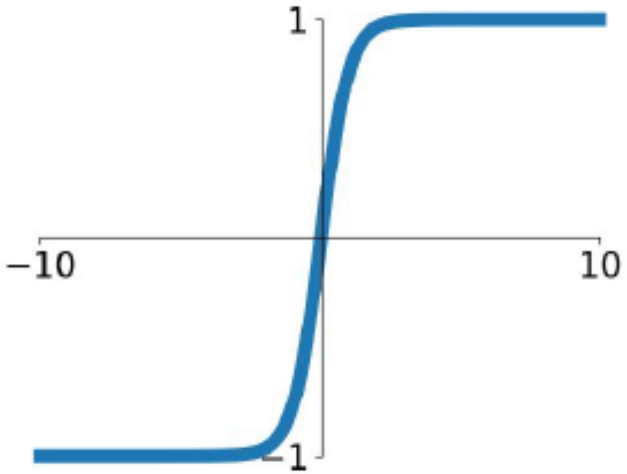


Examples of allowed gradients



Tanh

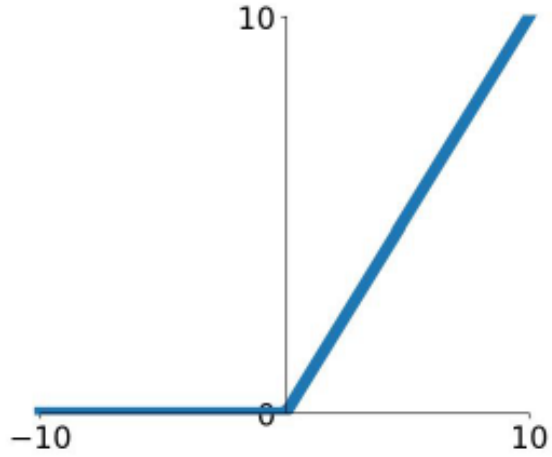
- Hyperbolic tangent
- Squashes numbers to range [-1, 1].
- Outputs are zero-centered (nice)
- Still kills gradient when saturated



tanh(x)

ReLU

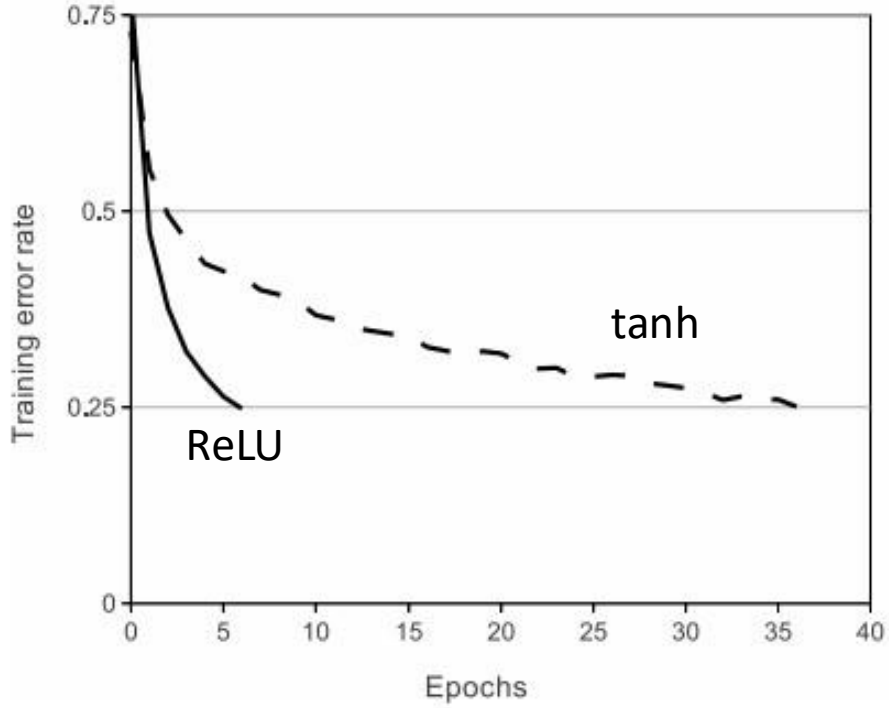
- Computes $f(x) = \max(0, x)$
- Does not saturate (in +region)
- Very computationally efficient
- Converges much faster than sigmoid/tanh in practice (e.g. 6x) [Krizhevsky et al., 2012]



ReLU
(Rectified Linear Unit)

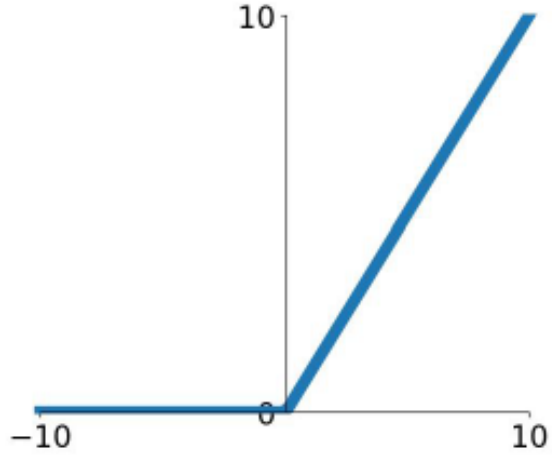
ReLU

- Computes $f(x) = \max(0, x)$
- Does not saturate (in +region)
- Very computationally efficient
- Converges much faster than sigmoid/tanh in practice (e.g. 6x) [Krizhevsky et al., 2012]



ReLU

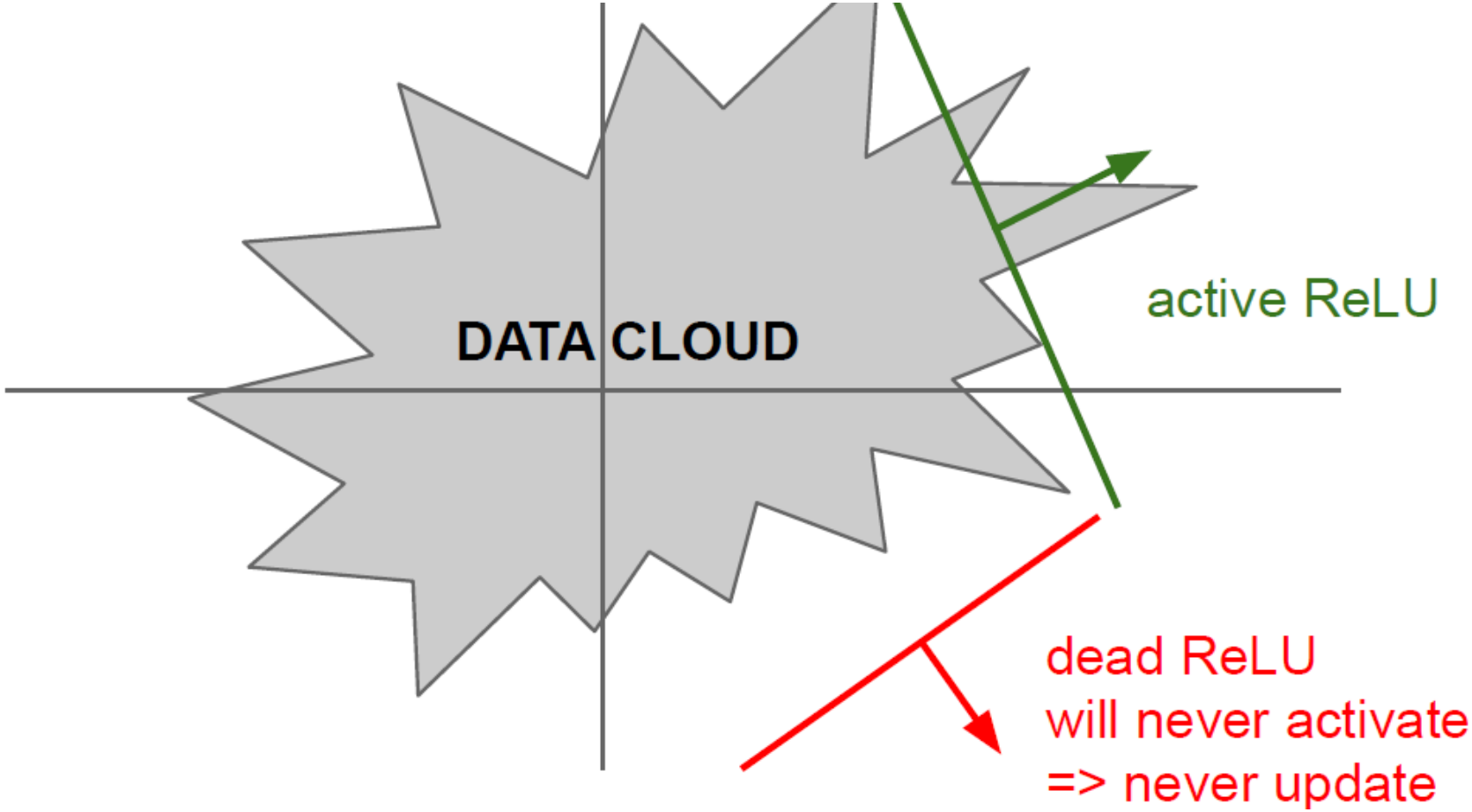
- Computes $f(x) = \max(0, x)$
- Does not saturate (in +region)
- Very computationally efficient
- Converges much faster than sigmoid/tanh in practice (e.g. 6x) [Krizhevsky et al., 2012]
- Not zero-centered output
- No gradient flow when x < 0 causing the neuron to "die":
  - A large gradient flowing through a ReLU could cause the weights to update in such a way that the neuron will never activate on any data point again.
  - With a proper setting of the learning rate this is less frequently an issue.



ReLU
(Rectified Linear Unit)

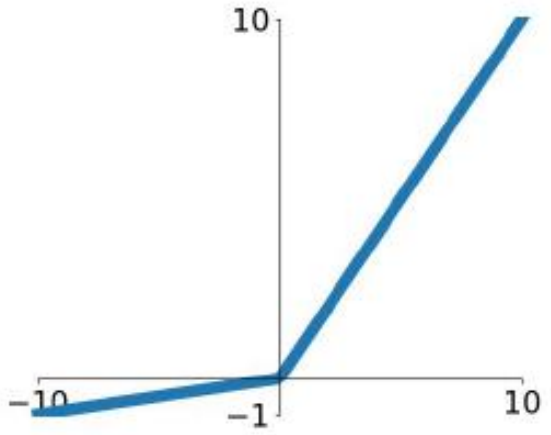
Dead ReLU

With a learning rate that is set too high, we risk pushing all the data so far "behind" the active part of a ReLU that it will never activate again.



Solution: Leaky ReLU

- Computes $f(x) = \max(\alpha \cdot x, x), \alpha < 1$
- Does not saturate
- Very computationally efficient
- Converges much faster than sigmoid/tanh in practice
- **Will not "die"**



- References:

https://arxiv.org/abs/1502.01852

https://ai.stanford.edu/~amaas/papers/relu_hybrid_icml2013_final.pdf

Leaky ReLU

$$f(x) = \max(0.01x, x)$$

Activation functions in practise

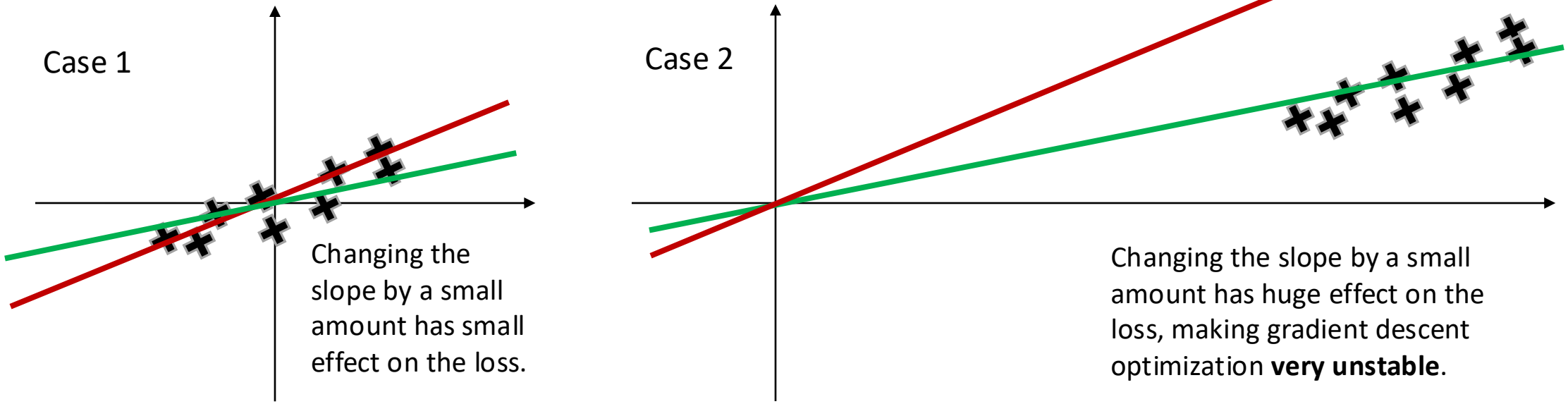
- Use **ReLU**, but be careful with your learning rates.
- Try out **Leaky ReLU** (or more exotic alternatives not covered here, like Maxout or ELU)
- Try out **tanh** but don't expect much.
- Don't use **sigmoid**, except maybe at the output layer, if you want values that reflect probabilities.

Data preprocessing

ALSO CALLED NORMALIZATION OR ZERO-CENTERING

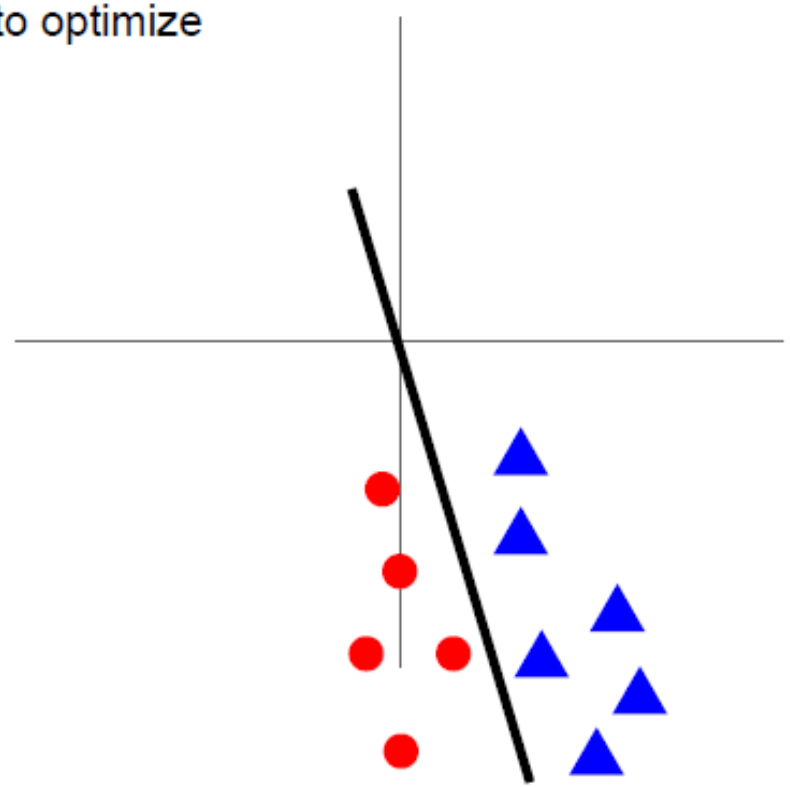
Why we prefer zero-centered data

- Your task is to fit a straight line to the datapoints shown below.
- **Q:** Which of the two cases is going to be the more challenging one? Why?
- **A:** Case 2 is harder, because the datapoints lie "far" away from the origin. Small changes in the slope (weights) cause large changes in the "prediction error" (i.e., the loss). The resulting fluctuations in the loss makes gradient descent optimization very unstable.

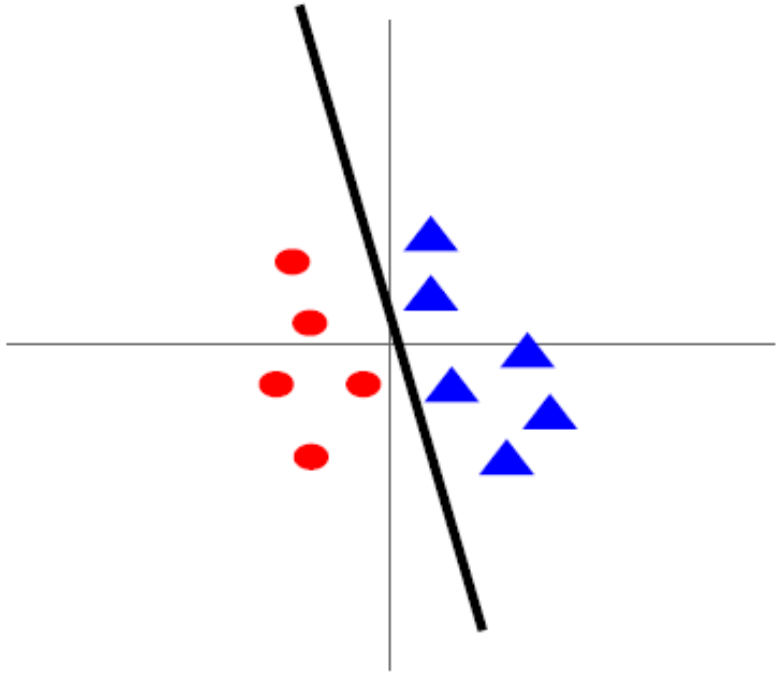


Classification example

Before normalization: classification loss very sensitive to changes in weight matrix; hard to optimize

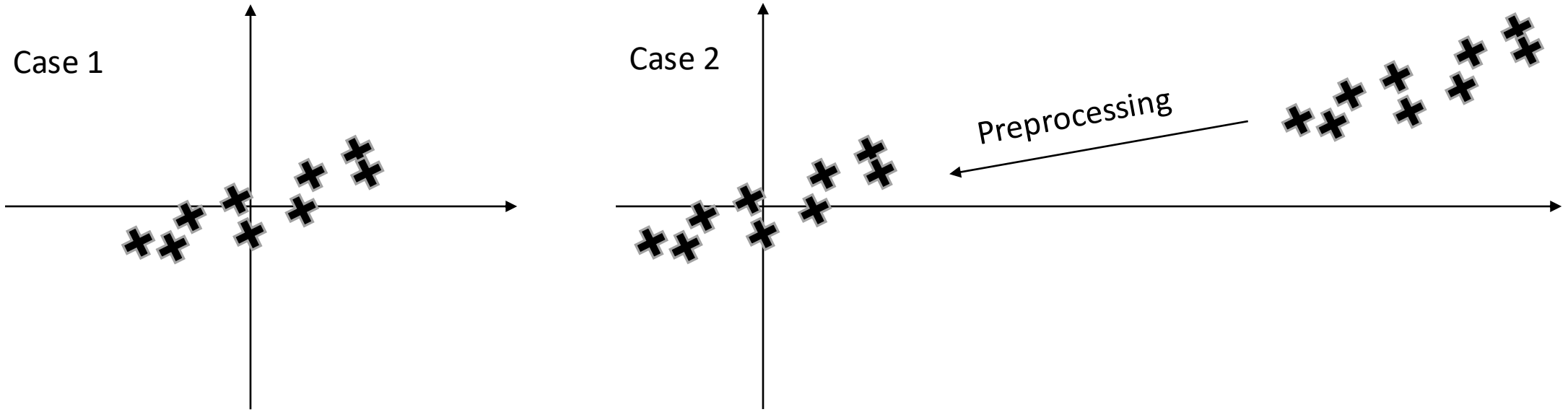


After normalization: less sensitive to small changes in weights; easier to optimize



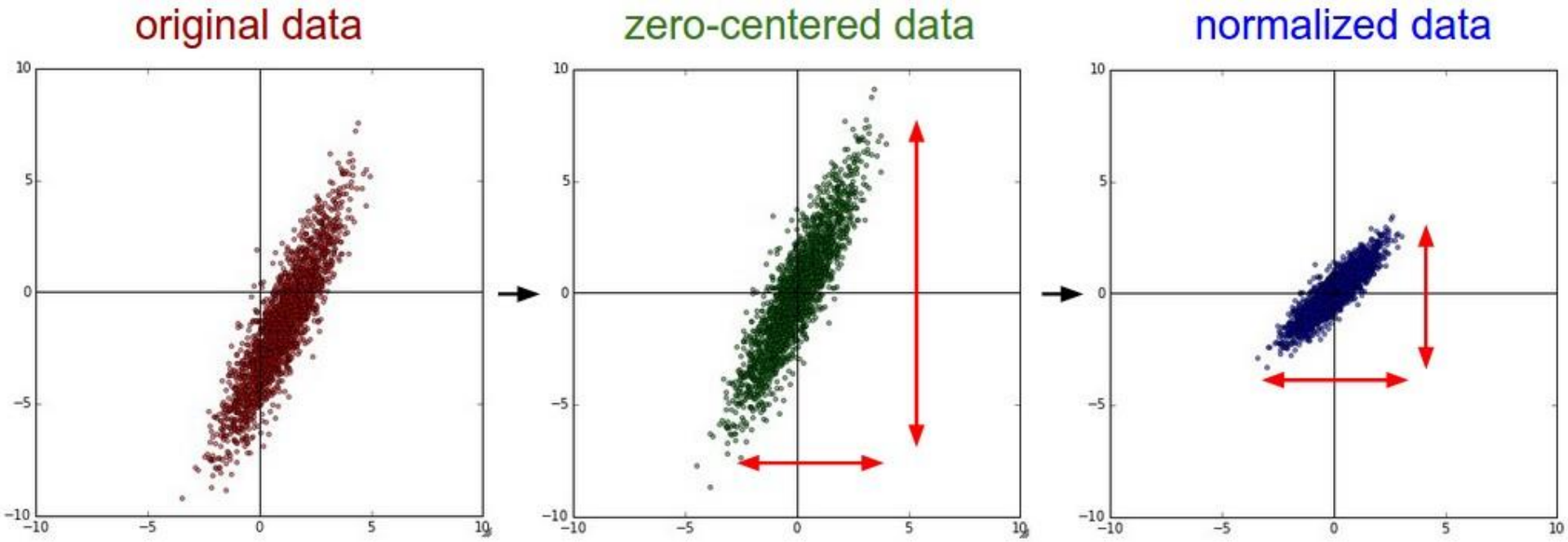
Why we prefer zero-centered data

- **Solution:** Preprocessing (or normalization, or zero-centering)
- Preprocessing may include
  - Centering data around zero
  - Making different data dimensions have the same variance or scale (i.e., equal importance).
  - Removing redundancy or correlations in the data (below it seems that **y** increases linearly with **x**).



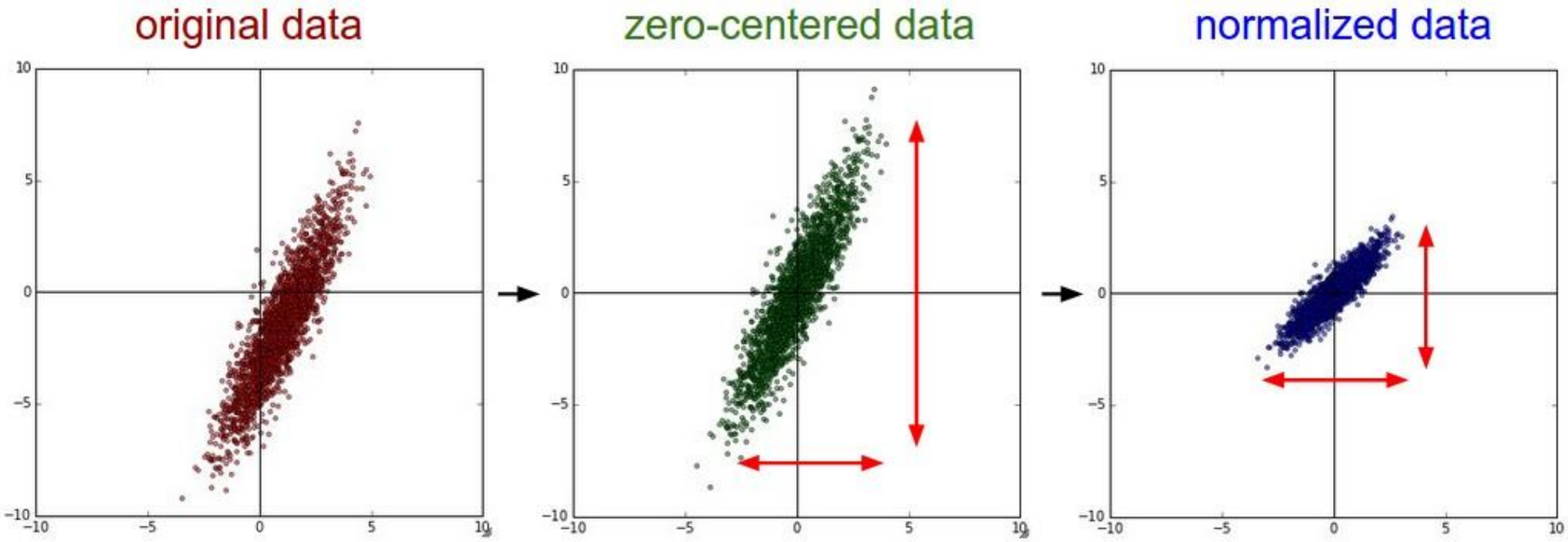
Mean subtraction and normalization

- **Mean subtraction** is the most common form of preprocessing. It involves subtracting the mean across every individual feature in the data and has the geometric interpretation of centering the cloud of data around the origin along every dimension.
- Assuming that data is stored in matrix X of size DxN (N number of datapoints, D is their dimensionality), then mean subtraction in numpy is `X -= np.mean(X, axis = 0)`.



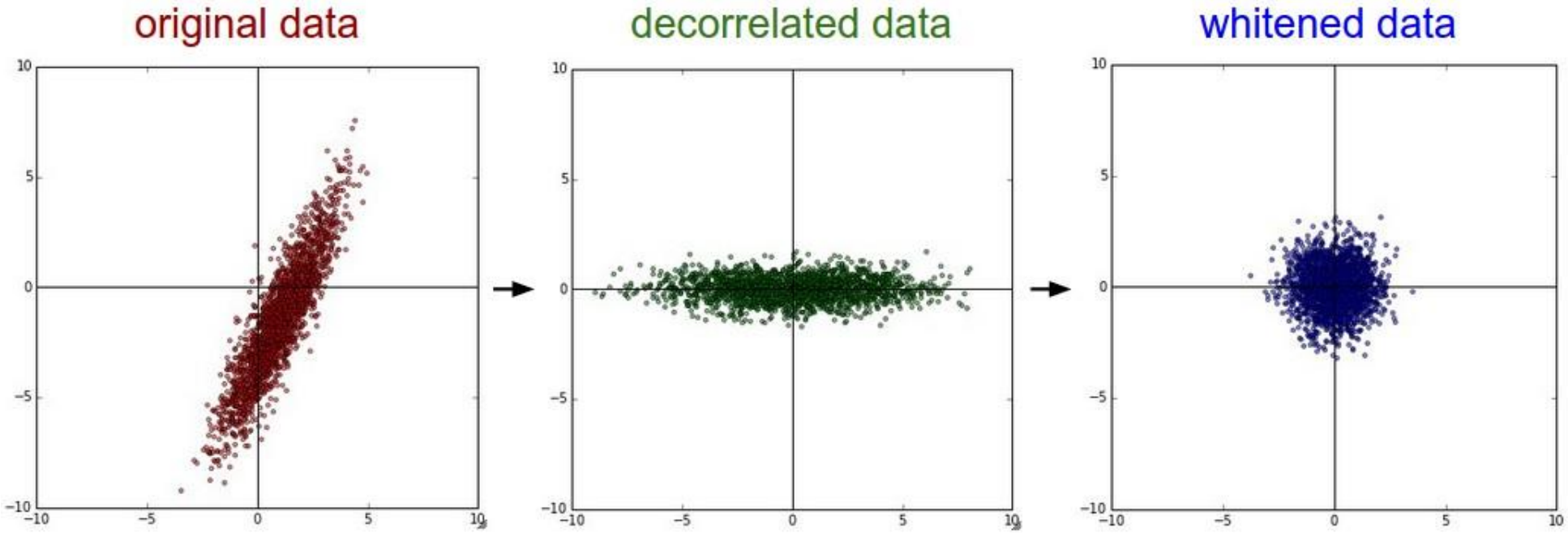
Mean subtraction and normalization

- **Normalization** refers to normalizing the data dimensions so that they are of approximately the same scale. There are two common ways of achieving this normalization.
- One is to divide each dimension by its standard deviation, once it has been zero-centered: `X /= np.std(X, axis = 0)`. Another form of this preprocessing normalizes each dimension so that the min and max along the dimensions are -1 and 1 respectively.



PCA and whitening

- Less commonly used in today's neural networks.
- **PCA – Principal Component Analysis:** Basically, it finds major trends in the data, which in turn allows spotting correlations and removing them (decorrelation).
- Whitening also normalizes the data in all dimensions.



http://ufldl.stanford.edu/tutorial/unsupervised/PCAWhitening/

Comments

- I mention PCA and whitening in these slides for completeness, but these transformations are not used with ConvNets.
- However, it is very important to zero-center the data, and it is common to see normalization of the data as well.
- **Common pitfalls:**
  - When deploying your ConvNet, remember to apply the same preprocessing as when you trained it.
  - This means that any preprocessing statistics (e.g., the data mean) must only be computed on the training data and then applied to the validation / test data.

In practise

- Example: CIFAR10
  - Subtract the mean image (e.g. AlexNet): mean image = [32,32,3] array
  - Subtract per-channel mean (e.g. VGGNet): mean along each channel = 3 numbers
  - Subtract per-channel mean and divide by per-channel standard deviation (e.g. ResNet): mean along each channel + std along each channel = 6 numbers
- Not common to do PCA or whitening.
- **Warning:** When using a pretrained network, it is important that you apply the same preprocessing as was used for training that particular network. In Keras, each pretrained model comes with its own pre-processor for this exact reason. See code examples here.

Weight initialization

Pitfall: all zero initialization

- Let's start with what we shouldn't do.
- With proper data normalization it is reasonable to **assume that approximately half of the input values are going to be positive and half of them are going to be negative**.
- A reasonable-sounding idea then might be to **set all the initial weights to zero**, which we expect to be the "best guess" in expectation.
- Turns out this doesn't work, because if every neuron in the network computes the same output, then they will also all compute the same gradients during backpropagation and **undergo the exact same parameter updates** (i.e., all neurons in a given layer will be identical – they will recognize the same pattern).
- In other words, there is **no source of asymmetry between neurons** if their weights are initialized to be the same.
- Note: It's safe to initialize the biases to zero.

Small random numbers

- To break symmetry, initialize the weights of the neurons to small random numbers.
- The idea is that the neurons are all random and unique in the beginning, so they will **compute distinct updates** and integrate themselves as diverse parts of the full network.
- Implementation for one weight matrix:

```
W = 0.01 * np.random.randn(D,N)
```

Small random numbers

- To break symmetry, initialize the weights of the neurons to small random numbers.
- The idea is that the neurons are all random and unique in the beginning, so they will **compute distinct updates** and integrate themselves as diverse parts of the full network.
- Implementation for one weight matrix:

```
W = 0.01 * np.random.randn(D,N)
```

Reminder:

If a random variable X is normally distributed with mean μ and standard deviation σ, we write $X \sim N(\mu, \sigma)$.

`np.random.randn` generates random samples from a standard normal distribution ($\mu = 0, \sigma = 1$).

Say that $Z \sim N(\mu = 0, \sigma = 1)$, and define $\mathrm{Y} = \alpha Z + \beta$. Then $\mathrm{Y} \sim N(\mu = \beta, \sigma = \alpha)$

Small random numbers

- To break symmetry, initialize the weights of the neurons to small random numbers.
- The idea is that the neurons are all random and unique in the beginning, so they will **compute distinct updates** and integrate themselves as diverse parts of the full network.
- Implementation for one weight matrix:

$N(\mu = 0, \sigma = 1)$

$N(\mu = 0, \sigma = 0.01)$

```
W = 0.01 * np.random.randn(D,N)
```

Reminder:

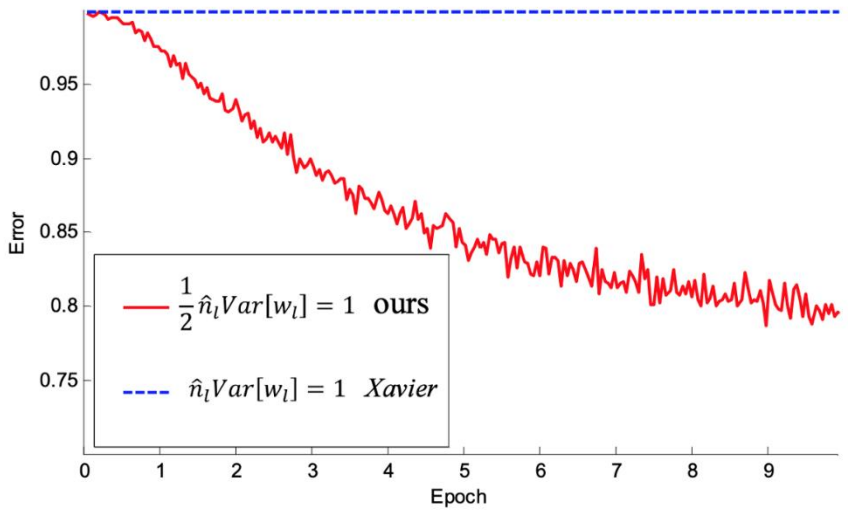
If a random variable X is normally distributed with mean μ and standard deviation σ, we write $X \sim N(\mu, \sigma)$.

`np.random.randn` generates random samples from a standard normal distribution ($\mu = 0, \sigma = 1$).

Say that $Z \sim N(\mu = 0, \sigma = 1)$, and define $\mathrm{Y} = \alpha Z + \beta$. Then $\mathrm{Y} \sim N(\mu = \beta, \sigma = \alpha)$

Why is weight initialization important?

- Initializing the network with the right weights can be the difference between the network converging in a reasonable amount of time and the network loss function not going anywhere even after hundreds of thousands of iterations.
- With improper weight initialization, the **layer activation outputs either explode or vanish** during the course of a forward pass through a deep neural network.
- If either occurs, **loss gradients will either be too large or too small to flow backwards** beneficially, and the network will take longer to converge, if it is even able to do so at all.
- What causes vanishing/exploding gradients?



He et al. (2015): Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification

The figure above illustrates the importance of selecting the correct weight initialization strategy. With Xavier initialization the loss doesn't decrease at all, but with Kaiming initialization, it does.

https://towardsdatascience.com/weight-initialization-in-neural-networks-a-journey-from-the-basics-to-kaiming-954fb9b47c79

Explanation

- In deep neural nets with several layers, one forward pass simply entails **performing consecutive matrix multiplications** at each layer, between that layer's inputs and weight matrix.
- At every iteration of the optimization loop (forward, cost, backward, update), the backpropagated gradients are either amplified or minimized as you move from the output layer towards the input layer.
- This result makes sense if you consider the following example:

$$\mathrm{y} = W^{(L)}W^{(L-1)} \ldots W^{(2)}W^{(1)}x$$

Case 1:

Define $W^{(L)} = W^{(L-1)} = \cdots = W^{(2)} = W^{(1)} = \begin{bmatrix} 1.5 & 0 \\ 0 & 1.5 \end{bmatrix}$

Then $\mathrm{y} = W^{(L)}1.5^{(L-1)}x$, the values of y increase exponentially.

Leads to exploding gradient problem.

https://www.deeplearning.ai/ai-notes/initialization/

Explanation

- In deep neural nets with several layers, one forward pass simply entails **performing consecutive matrix multiplications** at each layer, between that layer's inputs and weight matrix.
- At every iteration of the optimization loop (forward, cost, backward, update), the backpropagated gradients are either amplified or minimized as you move from the output layer towards the input layer.
- This result makes sense if you consider the following example:

$$\mathrm{y} = W^{(L)}W^{(L-1)} \ldots W^{(2)}W^{(1)}x$$

Case 2:

Define $W^{(L)} = W^{(L-1)} = \cdots = W^{(2)} = W^{(1)} = \begin{bmatrix} 0.5 & 0 \\ 0 & 0.5 \end{bmatrix}$

Then $\mathrm{y} = W^{(L)}0.5^{(L-1)}x$, the values of y decrease exponentially.

Leads to vanishing gradient problem.

https://www.deeplearning.ai/ai-notes/initialization/

Example

```
In [14]: x = torch.randn(512)
```

```
In [15]: for i in range(100):
             a = torch.randn(512,512)
             x = a @ x
         x.mean(), x.std()
Out[15]: (tensor(nan), tensor(nan))
```

Let's pretend that we have a simple 100-layer network with no activations, and that each layer has a matrix **a** that contains the layer's weights.

Initializing the weights with a standard normal distribution ($\mu = 0, \sigma = 1$), the layer outputs got so big that even the computer wasn't able to recognize their standard deviation and mean as numbers.

```
In [17]: x = torch.randn(512)

         for i in range(100):
             a = torch.randn(512,512) * 0.01
             x = a @ x
         x.mean(), x.std()
Out[17]: (tensor(0.), tensor(0.))
```

Initializing the weights with a normal distribution with $\mu = 0$ and $\sigma = 0.01$, during the course of the forward pass, the activation outputs completely vanished.

https://towardsdatascience.com/weight-initialization-in-neural-networks-a-journey-from-the-basics-to-kaiming-954fb9b47c79

Example

```
In [14]: x = torch.randn(512)
```

```
In [15]: for i in range(100):
             a = torch.randn(512,512)
             x = a @ x
         x.mean(), x.std()
Out[15]: (tensor(nan), tensor(nan))
```

Exploding gradients

Let's pretend that we have a simple 100-layer network with no activations, and that each layer has a matrix **a** that contains the layer's weights.

Initializing the weights with a standard normal distribution ($\mu = 0, \sigma = 1$), the layer outputs got so big that even the computer wasn't able to recognize their standard deviation and mean as numbers.

```
In [17]: x = torch.randn(512)

         for i in range(100):
             a = torch.randn(512,512) * 0.01
             x = a @ x
         x.mean(), x.std()
Out[17]: (tensor(0.), tensor(0.))
```

Vanishing gradients

Initializing the weights with a normal distribution with $\mu = 0$ and $\sigma = 0.01$, during the course of the forward pass, the activation outputs completely vanished.

https://towardsdatascience.com/weight-initialization-in-neural-networks-a-journey-from-the-basics-to-kaiming-954fb9b47c79

How can we find the sweet spot?

- To prevent the gradients of the network's activations from vanishing or exploding, we will stick to **the following rules of thumb**:

    1. The mean of the activations in any given layer should be zero.

    2. The variance of the activations should stay the same across every layer.

- Under these two assumptions, the backpropagated gradient signal should not be multiplied by values too small or too large in any layer, thereby avoiding exploding/vanishing gradients.

How can we find the sweet spot?

- To prevent the gradients of the network's activations from vanishing or exploding, we will stick to **the following rules of thumb**:

    1. The mean of the activations in any given layer should be zero.
    2. The variance of the activations should stay the same across every layer.

- Under these two assumptions, the backpropagated gradient signal should not be multiplied by values too small or too large in any layer, thereby avoiding exploding/vanishing gradients.
- Recall equations for forward pass:

$$a^{(1)} = x$$
$$z^{(j+1)} = W^{(j)} a^{(j)} + b^{(j)}$$
$$a^{(j+1)} = f\left(z^{(j+1)}\right)$$

$\longrightarrow$

1. $\mathrm{E}\left[a^{(j+1)}\right] = 0$
2. $\mathrm{Var}\left(a^{(j+1)}\right) = \mathrm{Var}\left(a^{(j)}\right)$

- where f can be any activation function (sigmoid, tanh, ReLU, …).

How can we find the sweet spot?

Reminder:

The mean or expected value of a random variable X is written $E[X]$ or μ.

Similarly, the variance is written $Var(X)$ or σ^2, where σ is the standard deviation.

Note that $\sigma = \sqrt{Var(X)}$

- Recall equations for forward pass:

$$a^{(1)} = x$$
$$z^{(j+1)} = W^{(j)} a^{(j)} + b^{(j)}$$
$$a^{(j+1)} = f\left(z^{(j+1)}\right)$$

$\longrightarrow$

1. $\mathrm{E}\left[a^{(j+1)}\right] = 0$
2. $\mathrm{Var}\left(a^{(j+1)}\right) = \mathrm{Var}\left(a^{(j)}\right)$

- where f can be any activation function (sigmoid, tanh, ReLU, …).

Xavier initialization

- Assume the activation function is the hyperbolic tangent (tanh):

$$a^{(1)} = x$$
$$z^{(j+1)} = W^{(j)} a^{(j)} + b^{(j)}$$
$$a^{(j+1)} = \tanh\left(z^{(j+1)}\right)$$

- Basic idea of derivation is to show that under certain assumptions:

$$\mathrm{Var}\left(a^{(j+1)}\right) = s_j \mathrm{Var}\left(W^{(j)}\right) \mathrm{Var}\left(a^{(j)}\right)$$

s_j is the number of neurons in layer *j*, i.e., the number of columns of $W^{(j)}$.

- If we want the variance to stay the same across layers, $\mathrm{Var}\left(a^{(j+1)}\right) = \mathrm{Var}\left(a^{(j)}\right)$, we must have

$$\mathrm{Var}\left(W^{(j)}\right) = 1/s_j$$

- See full derivation here: https://www.deeplearning.ai/ai-notes/initialization/

Glorot et al. (2010): Understanding the difficulty of training deep feedforward neural networks

Xavier initialization

- Thus, in order to avoid the vanishing or exploding of the forward propagated signal, we must set $s_j \mathrm{Var}\left(W^{(j)}\right) = 1$. We can do so by initializing $W^{(j)}$ such that $\mathrm{Var}\left(W^{(j)}\right) = 1/s_j$.

```
In [25]: def tanh(x): return torch.tanh(x)

In [26]: x = torch.randn(512)

         for i in range(100):
             a = torch.randn(512,512) * math.sqrt(1./512)
             x = tanh(a @ x)
         x.mean(), x.std()

Out[26]: (tensor(-0.0034), tensor(0.0613))
```

Example: $s_j = 512, \sigma = \sqrt{1/s_j} = \sqrt{1/512}$

The standard deviation of activation outputs of the 100th layer is down to about 0.06. This is definitely on the small side, but at least activations haven't totally vanished!

- Notice the following three cases:

$$s_j \mathrm{Var}\left(W^{(j)}\right) = \begin{cases} < 1 & \rightarrow \text{Vanishing signal} \\ = 1 & \rightarrow \mathrm{Var}\left(a^{(L)}\right) = \mathrm{Var}(x) \\ > 1 & \rightarrow \text{Exploding signal} \end{cases}$$

Glorot et al. (2010): Understanding the difficulty of training deep feedforward neural networks

Visualizing activation statistics

```
import numpy as np

# Forward pass for a neural net with 6 layers
dims = [4096]*7
hs = []
x = np.random.randn(16,dims[0])
for Din, Dout in zip(dims[:-1],dims[1:]):
  W = 0.01 * np.random.randn(Din,Dout)
  x = np.tanh(x.dot(W))
  hs.append(x)
```

Consider simple neural network:

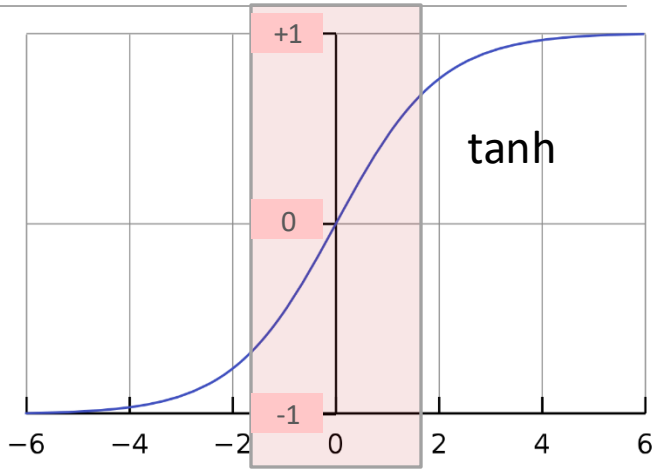
- 6 layers
- 4096 dimensional input
- 16 neurons in each layer
- tanh activation

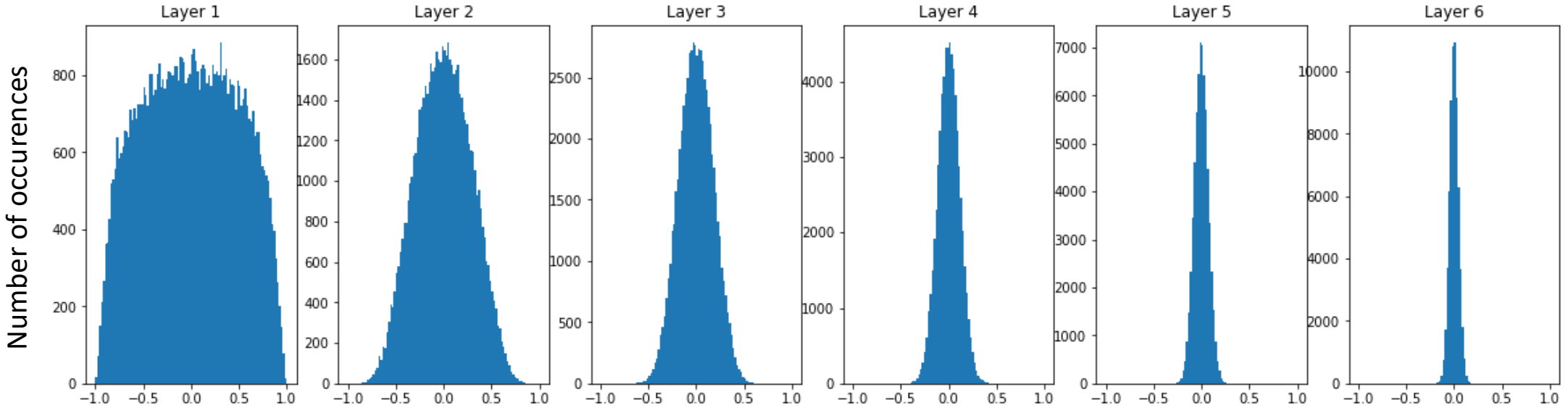
Initialize weights with small random values:

- Normal distribution
- Zero-mean
- Standard deviation 0.01

Visualizing activation statistics

- All activations tend to zero for deeper network layers (see histograms below).
- **Q:** What happens to the network's learning capacity?
- **A:** It decreases, because activation function (tanh or sigmoid) approaches **linear range**, so that we effectively end up with a linear model.





Values along the x-axis are the output activations: $a^{(j+1)} = \tanh\left(z^{(j+1)}\right)$

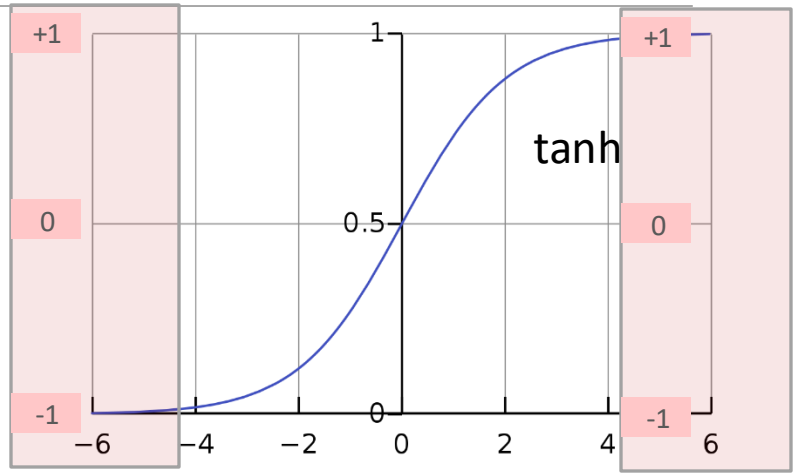
Visualizing activation statistics

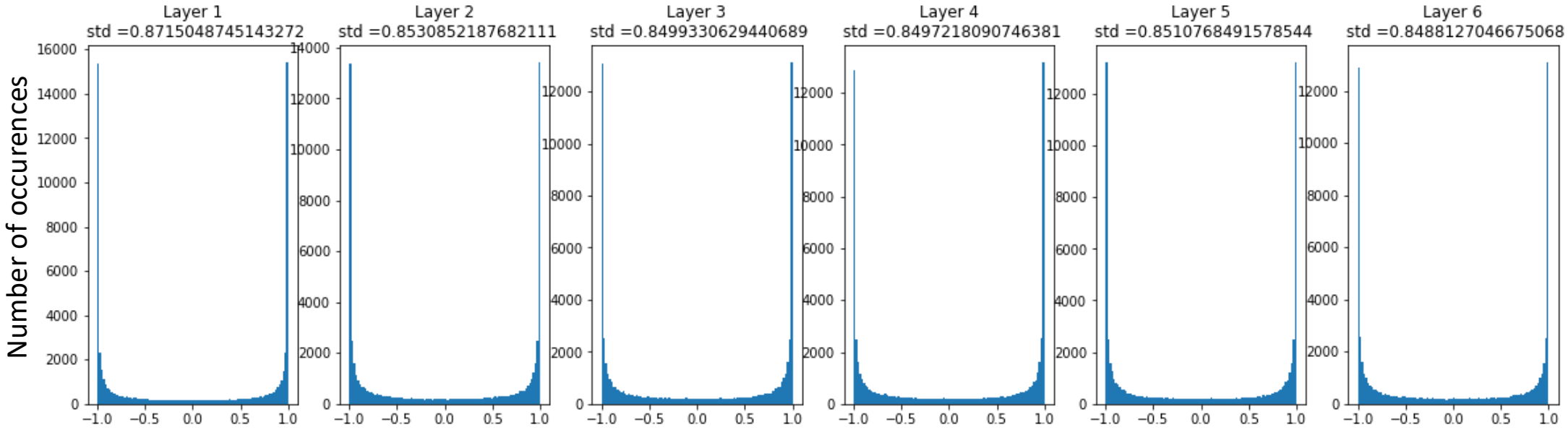
```
import numpy as np

# Forward pass for a neural net with 6 layers
dims = [4096]*7
hs = []
x = np.random.randn(16,dims[0])
for Din, Dout in zip(dims[:-1],dims[1:]):
  W = 0.05 * np.random.randn(Din,Dout)
  x = np.tanh(x.dot(W))
  hs.append(x)
```

What happens if we increase the standard deviation from 0.01 to 0.05?

Visualizing activation statistics

- Many activations saturate (see histograms below).
- **Q:** What do the gradients look like?
- **A:** Zero gradient due to saturation, so no learning…





Values along the x-axis are the output activations: $a^{(j+1)} = \tanh\left(z^{(j+1)}\right)$

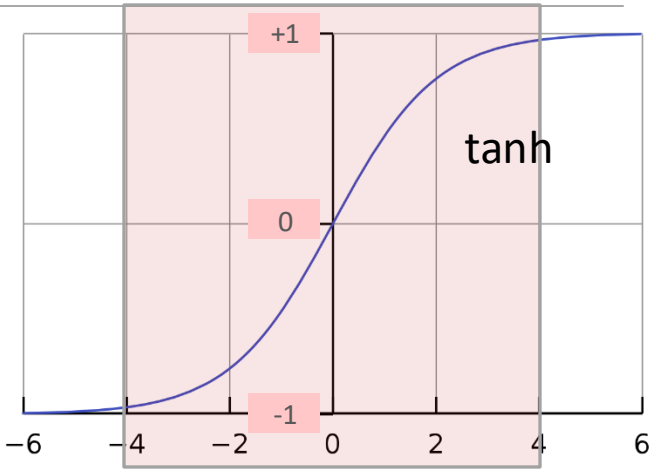
Visualizing activation statistics

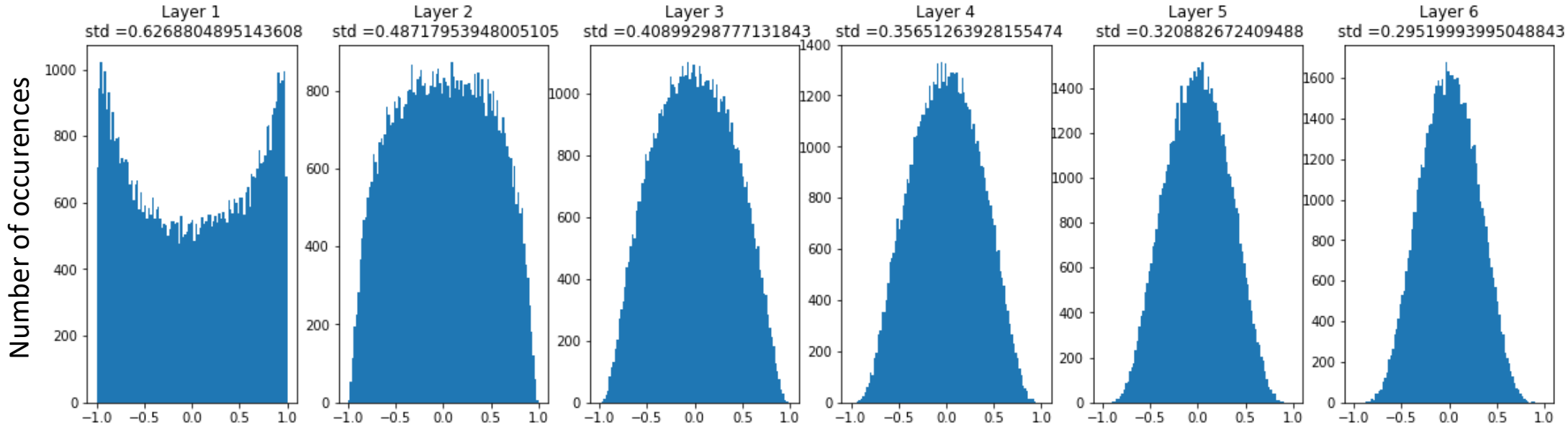
```
import numpy as np

# Forward pass for a neural net with 6 layers
dims = [4096]*7
hs = []
x = np.random.randn(16,dims[0])
for Din, Dout in zip(dims[:-1],dims[1:]):
  W = np.random.randn(Din,Dout) / np.sqrt(Din)
  x = np.tanh(x.dot(W))
  hs.append(x)
```

Xavier initialization:
Normalize weights by dividing by square root of number of input connections (D_{in})

Xavier initialization

- "Just right": Activations are nicely scaled for all layers so that we avoid saturation, while being able to move into the non-linear range of the tanh.
- For conv layers, D_{in} is kernel_size^2 * input_channels





Values along the x-axis are the output activations: $a^{(j+1)} = \tanh\left(z^{(j+1)}\right)$

What about ReLU?

- Conceptually, it makes sense that when using activation functions that are symmetric about zero and have outputs inside [-1,1], such as tanh, we'd want the activation outputs of each layer to have a mean of 0 and a standard deviation around 1, on average.
- This is precisely what Xavier initialization enables.
- But what if we're using ReLU activation functions? Would it still make sense to want to scale random initial weight values in the same way?
- The answer is **no**:

```
In [36]: x = torch.randn(512)

         for i in range(100):
             a = xavier(512, 512)
             x = relu(a @ x)
         x.mean(), x.std()

Out[36]: (tensor(5.3571e-16), tensor(7.7803e-16))
```

https://towardsdatascience.com/weight-initialization-in-neural-networks-a-journey-from-the-basics-to-kaiming-954fb9b47c79

What about ReLU?

- It turns out that when using a ReLU activation, a single layer will on average have a standard deviation that's very close to the square root of the number of input connections (D_{in}), *divided by the square root of two*, or √512/√2 in our example.

```
In [30]: def relu(x): return x.clamp_min(0.)

In [31]: mean, var = 0.,0.
         for i in range(10000):
             x = torch.randn(512)
             a = torch.randn(512,512)
             y = relu(a @ x)
             mean += y.mean().item()
             var += y.pow(2).mean().item()
         mean/10000, math.sqrt(var/10000)

Out[31]: (9.027289896678925, 16.01248299986557)

In [32]: math.sqrt(512/2)

Out[32]: 16.0
```

He et al. (2015): Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification

What about ReLU?

- Scaling the values of the weight matrix by this number will cause each individual ReLU layer to have a standard deviation of 1 on average.
- This is called “Kaiming initialization” or “He initialization”, depending on the framework.

```
In [34]: def kaiming(m,h):
             return torch.randn(m,h)*math.sqrt(2./m)

In [35]: x = torch.randn(512)

         for i in range(100):
             a = kaiming(512, 512)
             x = relu(a @ x)
         x.mean(), x.std()

Out[35]: (tensor(0.2789), tensor(0.4226))
```

https://towardsdatascience.com/weight-initialization-in-neural-networks-a-journey-from-the-basics-to-kaiming-954fb9b47c79

In practise

- Improper weight initialization can cause vanishing gradients (slow learning or no learning at all) or exploding gradients.
- The choice of weight initialization strategy depends on the choice of activation function:
  - For sigmoid and tanh, try Xavier (sometimes called Glorot).
  - For ReLU, use Kaiming (sometimes called He).
  - If none of these work, you will have to experiment on your own. Use a normal distribution with zero mean and adjust the variance until your model starts converging.
- Don't initialize with zeros or constants!
- **Note:** In Keras both Xavier and Kaiming come in two variants, where samples are drawn either from a normal distribution or a uniform distribution. The choice shouldn't make a huge difference in practise.

Batch normalization

Motivation

- We have seen that **small changes to the weights amplify** as the network becomes deeper, which leads to saturation and vanishing gradients.
- What causes this problem is something called Internal Covariance Shift, which you can think of as instabilities (i.e., changes) in the distribution of nonlinearity inputs as the network trains.
- **Traditional solution:** Use ReLUs, careful weight initialization, and small learning rates.
- **New solution:** Make the distributions stable by enforcing zero mean and unit variance.
- This is **batch normalization**

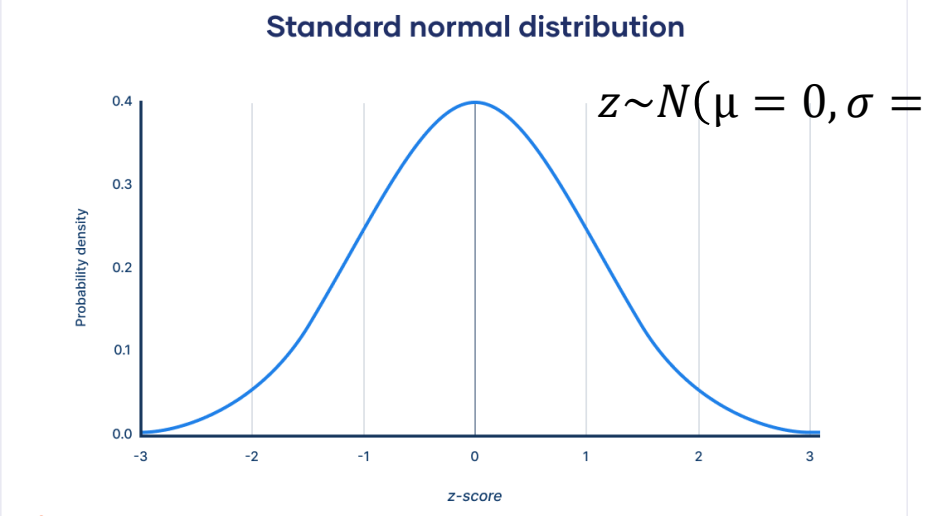
Ioffe & Szegedy (2015): Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift

Basic idea

- Consider a batch of activations at some layer.
- To make each dimension zero mean and unit variance, simply normalize each input dimension of x using the standard score:

$$z = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}(x)}} = \frac{x - \mu}{\sigma}$$

- where $\mathrm{E}[x] = \mu$ is the mean of x, and $\mathrm{Var}(x) = \sigma^2$ is the variance of x.
- Since we don't know the true mean and variance, we estimate them from the current batch.

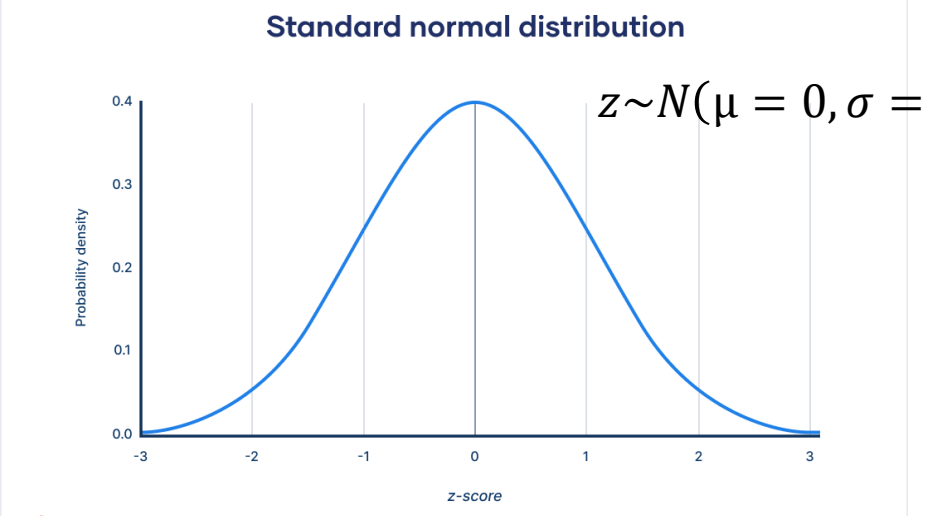


Basic idea

- Consider a batch of activations at some layer.
- To make each dimension zero mean and unit variance, simply normalize each input dimension of x using the standard score:

$$z = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}(x)}} = \frac{x - \mu}{\sigma}$$

- where $\mathrm{E}[x] = \mu$ is the mean of x, and $\mathrm{Var}(x) = \sigma^2$ is the variance of x.
- Since we don't know the true mean and variance, we estimate them from the current batch.



Recall our two rules of thumb to avoid vanishing/exploding gradients:

1. The mean of the activations in any given layer should be zero.
2. The variance of the activations should stay the same across every layer.

With standard normalization, we can easily get

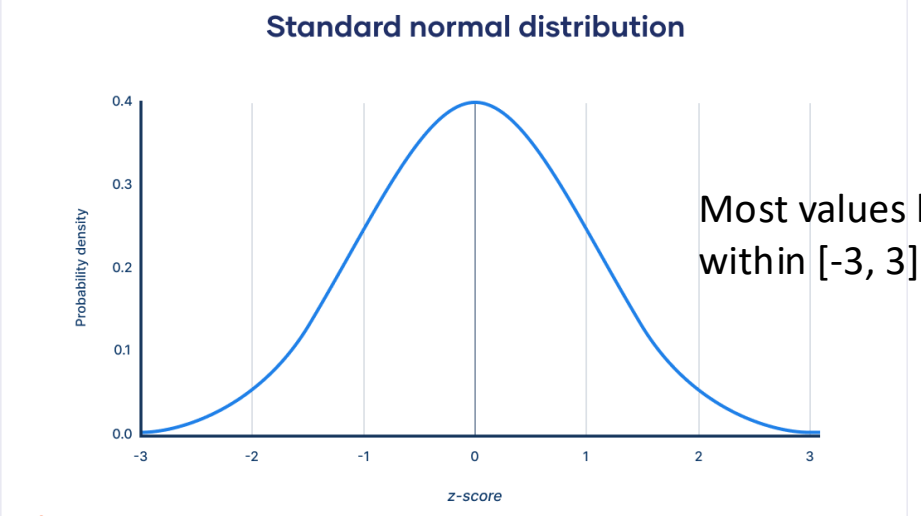
1. $\mathrm{E}\left[z^{(j+1)}\right] = 0$
2. $\mathrm{Var}\left(z^{(j+1)}\right) = 1$

Basic idea

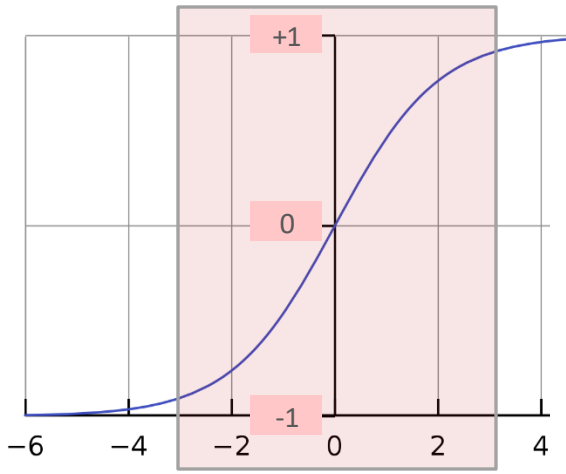
- Consider a batch of activations at some layer.
- To make each dimension zero mean and unit variance, simply normalize each input dimension of x using the standard score:

$$z = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}(x)}} = \frac{x - \mu}{\sigma}$$

- where $\mathrm{E}[x] = \mu$ is the mean of x, and $\mathrm{Var}(x) = \sigma^2$ is the variance of x.
- Since we don't know the true mean and variance, we estimate them from the current batch.



Most values lie within [-3, 3]



Example: tanh
Output has zero mean and constant σ (also, we avoid saturation and utilize non-linear range)

Batch normalization

- Input: $X \in \mathbb{R}^{N \times D}$
- N is the number of observations in a batch (i.e., the batch size).
- D is the dimensionality of the data (i.e., number of features such as pixels).
- Then X is normalized during training as

$$Z_{ij} = \frac{X_{ij} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}$$

- where

$$\mu_j = \frac{1}{N}\sum_{i=1}^{N} X_{ij} \text{ for } j = 1, \ldots, D$$

(per pixel mean if X is an image)

$$\sigma_j^2 = \frac{1}{N}\sum_{i=1}^{N} (X_{ij} - \mu_j)^2 \text{ for } j = 1, \ldots, D$$

(per pixel variance if X is an image)

Batch normalization

- Input: $X \in \mathbb{R}^{N \times D}$
- N is the number of observations in a batch (i.e., the batch size).
- D is the dimensionality of the data (i.e., number of features such as pixels).
- Then X is normalized during training as

$$Z_{ij} = \frac{X_{ij} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}$$

- Note that simply normalizing each input of a layer may change what the layer can represent.
- For instance, normalizing the inputs of a sigmoid would constrain them to the linear regime of the nonlinearity. To address this, we **make sure that the transformation inserted in the network can represent the identity transform**.

Batch normalization

- Input: $X \in \mathbb{R}^{N \times D}$
- N is the number of observations in a batch (i.e., the batch size).
- D is the dimensionality of the data (i.e., number of features such as pixels).
- Then X is normalized during training as

$$Z_{ij} = \frac{X_{ij} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}$$

- To accomplish this, we introduce, for each dimension, a pair of **learnable parameters** γ_j, β_j, which scale and shift the normalized value:

$$Y_{ij} = \gamma_j Z_{ij} + \beta_j$$

- Learning $\gamma_j = \sigma_j$ and $\beta_j = \mu_j$ will recover the identity function ($Y_{ij} = X_{ij}$).

Batch normalization at test time

- During training, estimates of mean and variance depend on mini-batch:

$$\mu_j = \frac{1}{N}\sum_{i=1}^{N} X_{ij} \text{ for } j = 1, \ldots, D$$

$$\sigma_j^2 = \frac{1}{N}\sum_{i=1}^{N} (X_{ij} - \mu_j)^2 \text{ for } j = 1, \ldots, D$$

$$Z_{ij} = \frac{X_{ij} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}$$

- At test time

$$\mu_j = (\text{running}) \text{ average of values seen during training}$$
$$\sigma_j^2 = (\text{running}) \text{ average of values seen during training}$$

- During testing, batch norm becomes a linear operator.
- Can be fused with the previous fully-connected or conv layer

Batch normalization in ConvNets

Fully-connected neural networks:

Input:

$X: N \times D$

Normalize:

$\mu, \sigma : 1 \times D$
$\gamma, \beta : 1 \times D$ (learnable)

$Y_{ij} = \gamma_j(X_{ij} - \mu_j)/\sigma_j + \beta_j$

Separate mean and variance for each feature dimension (D).

***N* is the batch size**

Convolutional neural networks:

Input:

$X: N \times C \times H \times W$

Normalize:

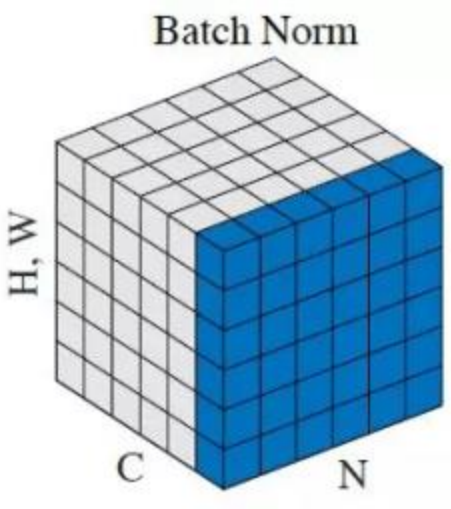
$\mu, \sigma : 1 \times C \times 1 \times 1$
$\gamma, \beta : 1 \times C \times 1 \times 1$ (learnable)

$Y_{ijmn} = \gamma_j(X_{ijmn} - \mu_j)/\sigma_j + \beta_j$

Separate mean and variance for each channel (C).

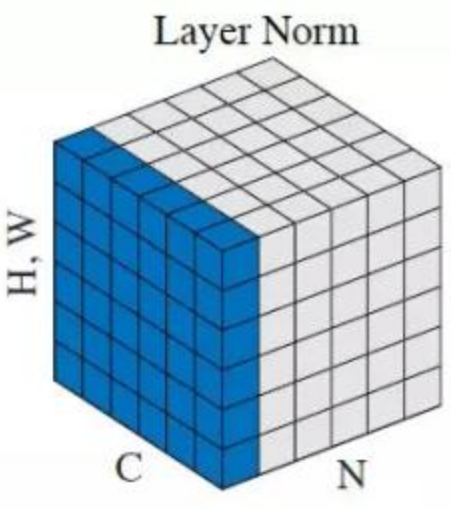
Other variants

$\mu, \sigma : 1 \times C \times 1 \times 1$



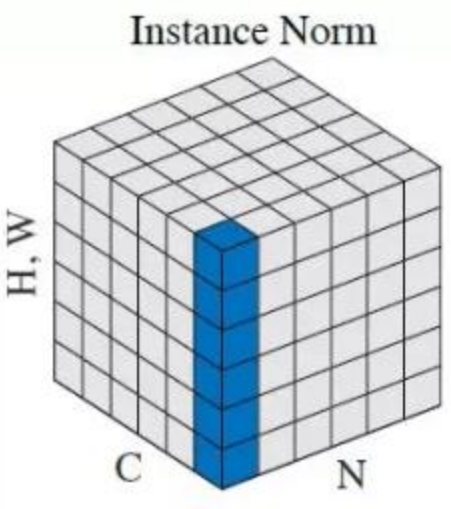
https://arxiv.org/abs/1502.03167

$\mu, \sigma : N \times 1 \times 1 \times 1$

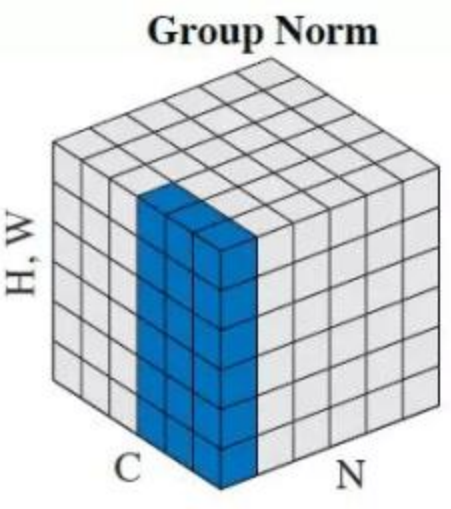


https://arxiv.org/abs/1607.06450

$\mu, \sigma : N \times C \times 1 \times 1$



https://arxiv.org/abs/1701.02096

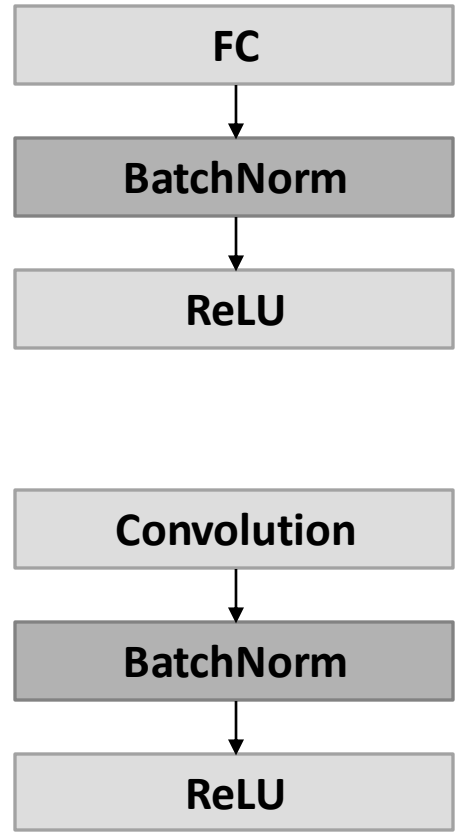


https://arxiv.org/abs/1803.08494

The batch normalization layer

```
mobilenet_full.summary()

WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/ker
Instructions for updating:
Please use `rate` instead of `keep_prob`. Rate should be set to `r
Downloading data from https://github.com/fchollet/deep-learning-mo
17227776/17225924 [==============================] - 13s 1us/step
Model: "mobilenet_1.00_224"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_2 (InputLayer)         (None, 224, 224, 3)       0
_________________________________________________________________
conv1_pad (ZeroPadding2D)    (None, 225, 225, 3)       0
_________________________________________________________________
conv1 (Conv2D)               (None, 112, 112, 32)      864
_________________________________________________________________
conv1_bn (BatchNormalization (None, 112, 112, 32)      128
_________________________________________________________________
conv1_relu (ReLU)            (None, 112, 112, 32)      0
_________________________________________________________________
conv_dw_1 (DepthwiseConv2D)  (None, 112, 112, 32)      288
_________________________________________________________________
conv_dw_1_bn (BatchNormaliza (None, 112, 112, 32)      128
_________________________________________________________________
conv_dw_1_relu (ReLU)        (None, 112, 112, 32)      0
_________________________________________________________________
conv_pw_1 (Conv2D)           (None, 112, 112, 64)      2048
_________________________________________________________________
conv_pw_1_bn (BatchNormaliza (None, 112, 112, 64)      256
_________________________________________________________________
conv_pw_1_relu (ReLU)        (None, 112, 112, 64)      0
_________________________________________________________________
conv_pad_2 (ZeroPadding2D)   (None, 113, 113, 64)      0
_________________________________________________________________
conv_dw_2 (DepthwiseConv2D)  (None, 56, 56, 64)        576
_________________________________________________________________
conv_dw_2_bn (BatchNormaliza (None, 56, 56, 64)        256
_________________________________________________________________
conv_dw_2_relu (ReLU)        (None, 56, 56, 64)        0
```

Usually inserted **after** fully-connected or convolutional layers, and **before** nonlinearity.



Batch norm enables higher learning rates

- A too-high learning rate may result in
  - gradients that explode
  - model getting stuck in poor local minima
  - dead ReLUs
- By normalizing activations throughout the network, batch normalization **prevents small changes to the parameters from amplifying** into larger and suboptimal changes in activations in gradients; for instance, it **prevents the training from getting stuck in the saturated regimes of nonlinearities**.
- Batch normalization also **makes training more resilient to the parameter scale**. Normally, large learning rates may increase the scale of layer parameters, which then amplify the gradient during backpropagation and lead to the **model explosion**. However, with Batch Normalization, backpropagation through a layer is unaffected by the scale of its parameters.

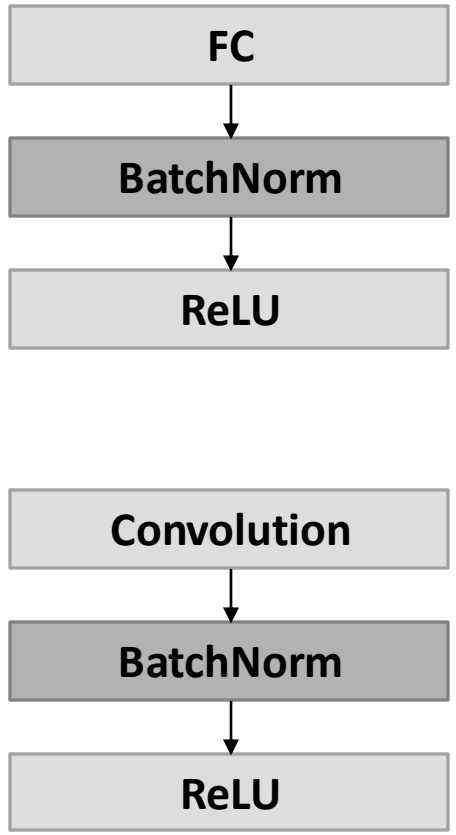
Batch norm regularizes the model

- When training with batch normalization, a training example is seen in conjunction with other examples in the mini-batch, and the training network **no longer producing deterministic values for a given training example**.
- Experiments have found this effect to be advantageous to the generalization of the network.
- Means that **instead of using dropout** to reduce overfitting (see later lecture), we can use batch normalization.

In practise

- Always use batch normalization
- Insert BatchNorm layers before non-linearities
- Makes deep networks **much** easier to train!
- Improves gradient flow (i.e., prevents vanishing gradients)
- Allows higher learning rates, faster convergence
- Networks become more robust to initialization
- Acts as regularization during training
- Zero overhead at test-time: can be fused with FC/conv!

- Behaves differently during training and testing: this is a very common source of bugs! (Read more here).



Summary

- **Activation functions:** Use ReLU
- **Data preprocessing:** Subtract mean
- **Weight initialization:** Use Xavier/Kaiming
- **Batch normalization:** Use always
- **Next time**
  - **Optimizer:** Use SGD+Momentum or Adam
  - **Regularization:** Early stopping and data augmentation almost always a good idea
  - **Hyperparameter search:** Course-to-fine search and monitor loss curves
  - **Transfer learning:** Almost always a good idea

Further reading + online videos/demos

- Optimization:
  - https://medium.com/analytics-vidhya/optimization-algorithms-for-deep-learning-1f1a2bd4c46b
  - http://ruder.io/optimizing-gradient-descent/
  - https://lilianweng.github.io/lil-log/2019/03/14/are-deep-neural-networks-dramatically-overfitted.html
  - https://medium.com/inveterate-learner/deep-learning-book-chapter-8-optimization-for-training-deep-models-part-i-20ae75984cb2
- Stanford CS231:
  - http://cs231n.github.io/neural-networks-2/
  - http://cs231n.github.io/neural-networks-3/
  - http://cs231n.github.io/transfer-learning/