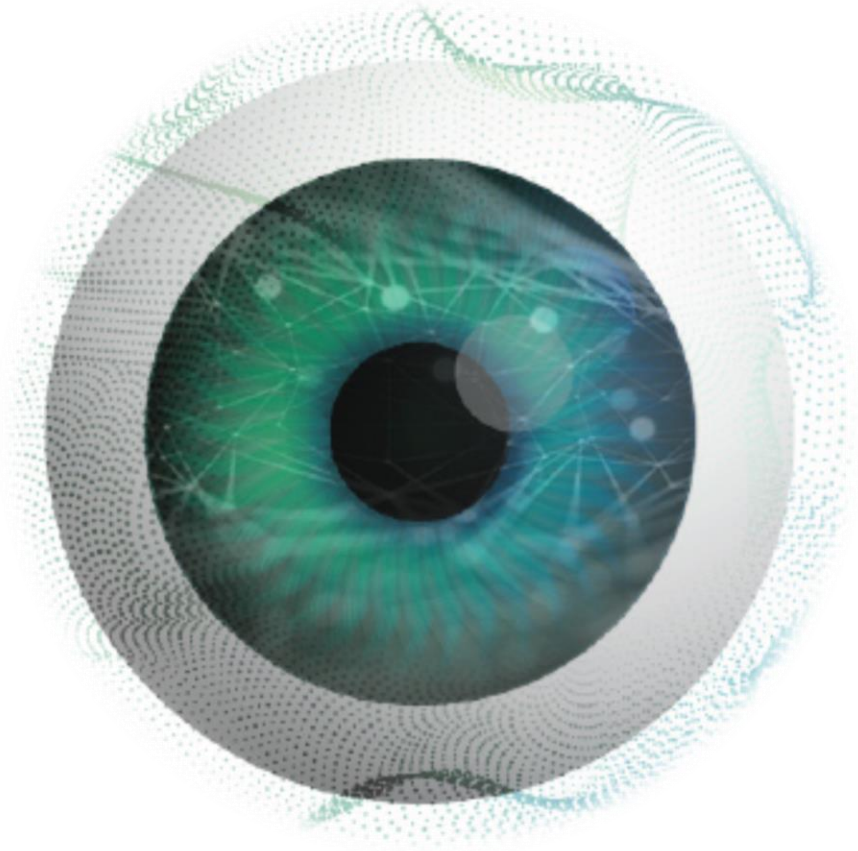
# DEEP LEARNING FOR VISUAL RECOGNITION

Lecture 6 – Training Convolutional Neural Networks

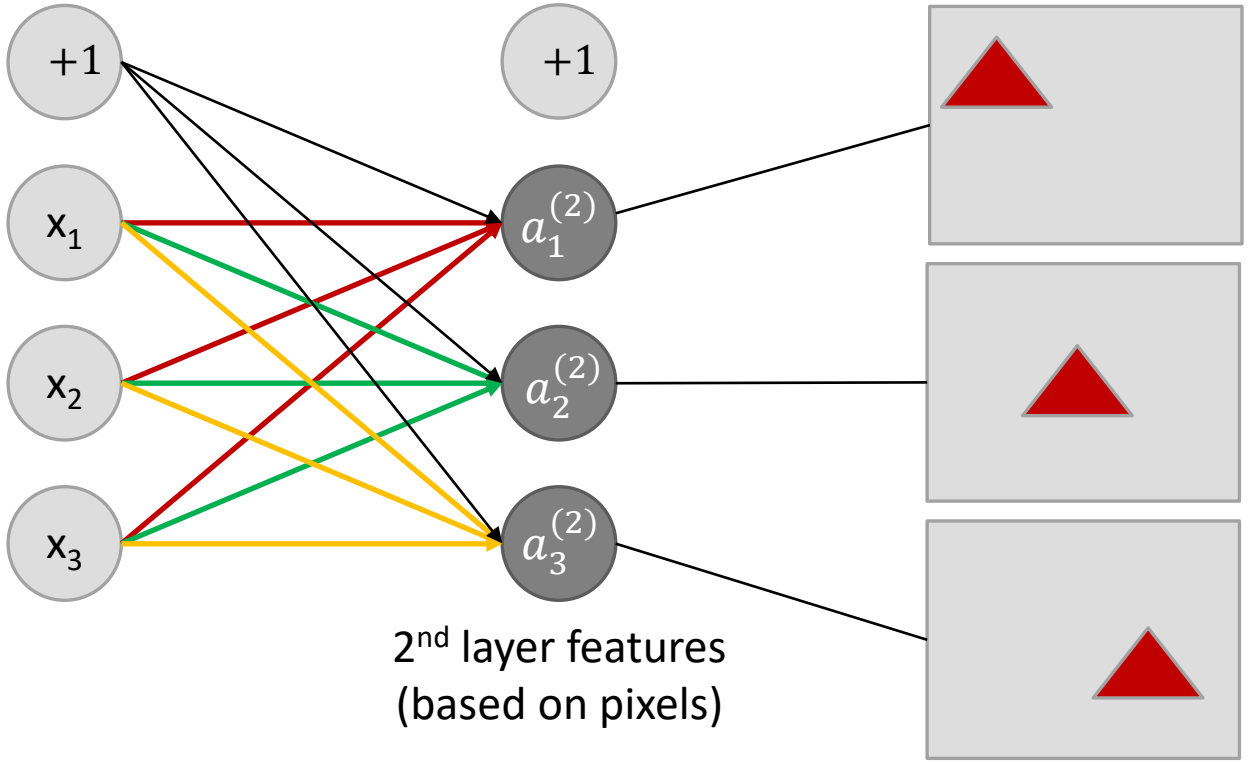


**Henrik Pedersen, PhD**
External lecturer
Department of Computer Science
Aarhus University
hpe@cs.au.dk

# Today's agenda

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

# Where we are now

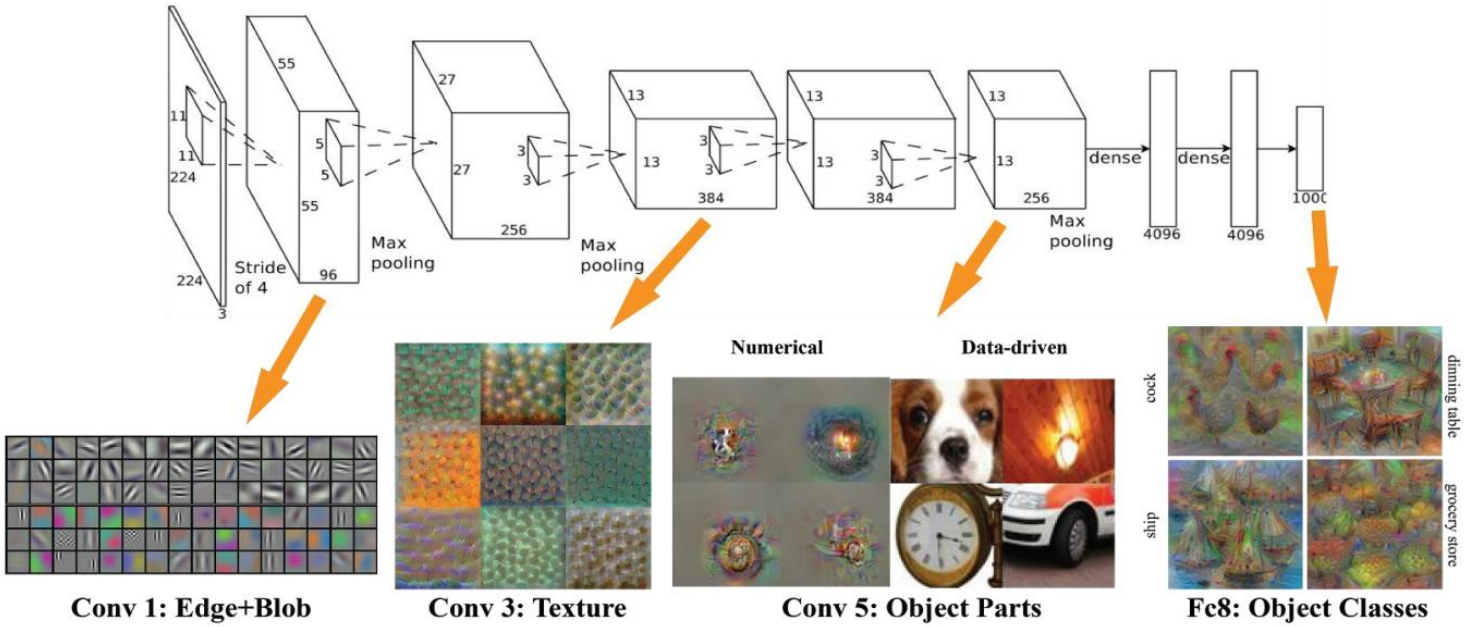



**Fully connected networks:**
Not well suited for images

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

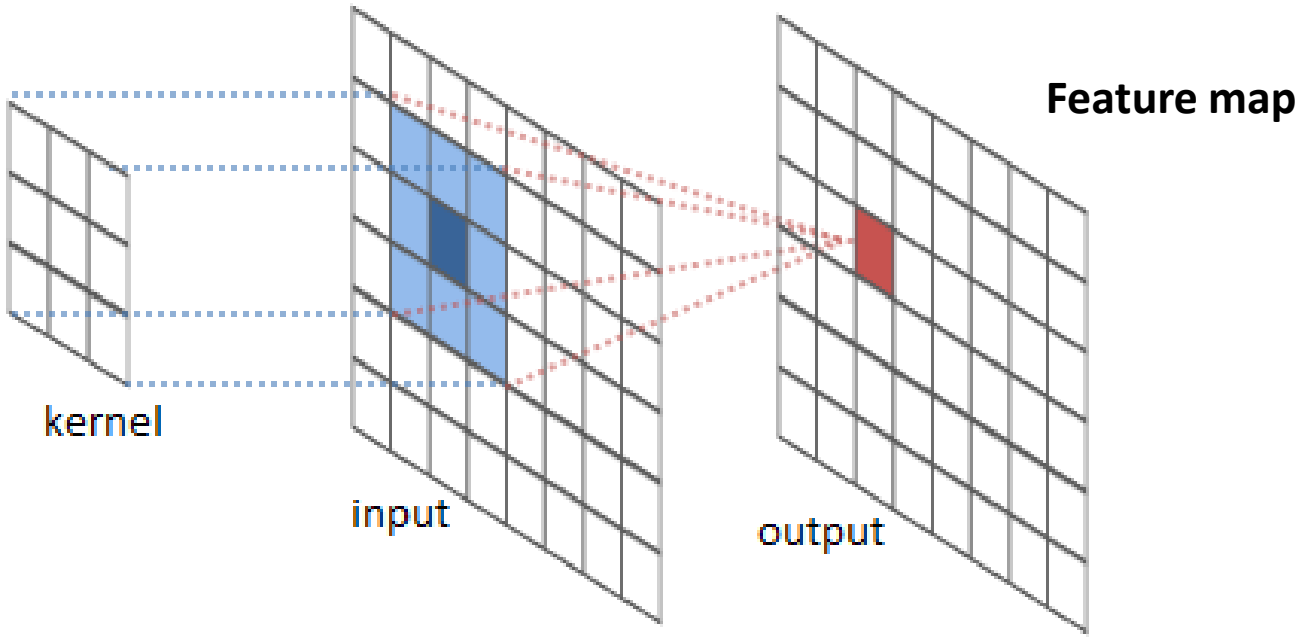
# Where we are now

- 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

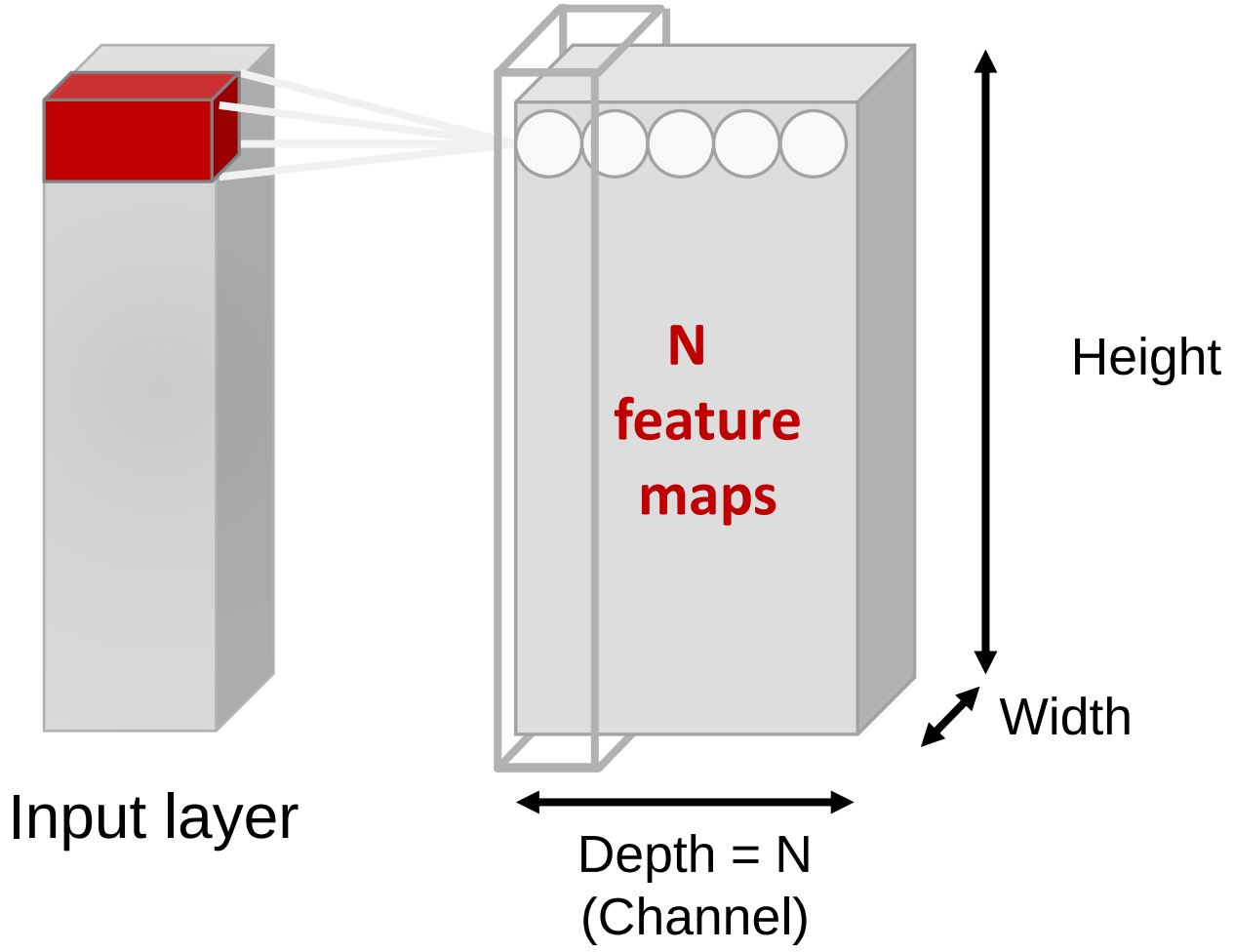
**One output channel per filter**, referred to as a **feature map**.

The filter connectivity is

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

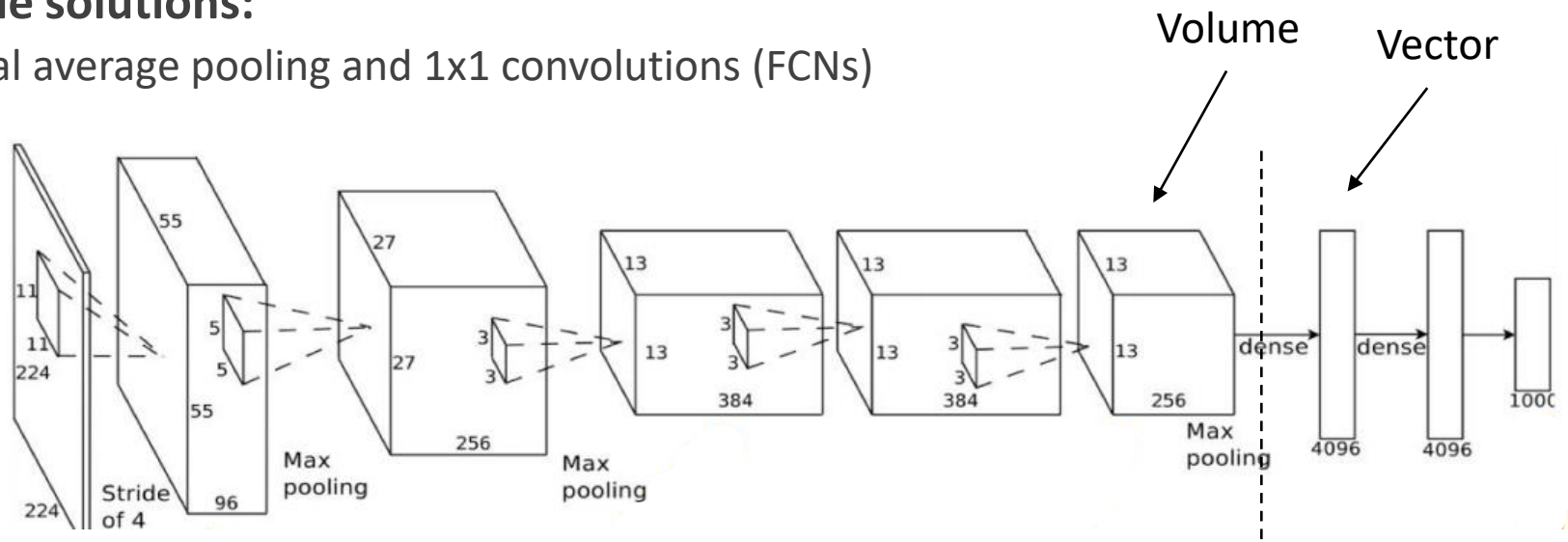
Advantages over fully connected neural networks:

- Sparse interactions
- Parameter sharing
- Translation invariance

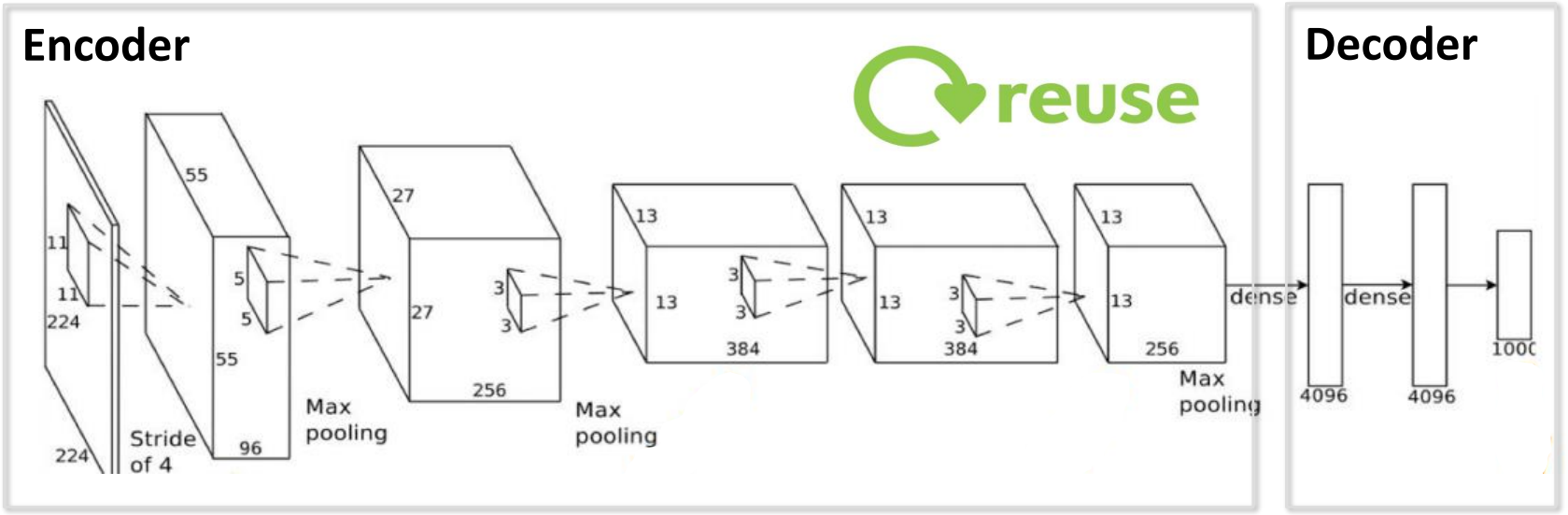


# Where we are now

- 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)



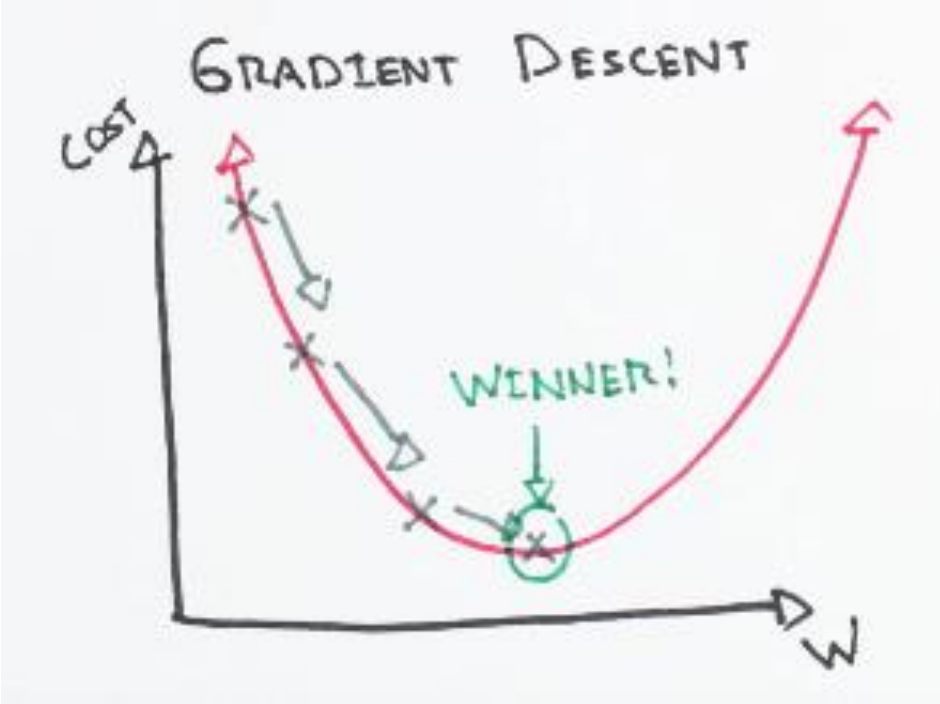
# Where we are now




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

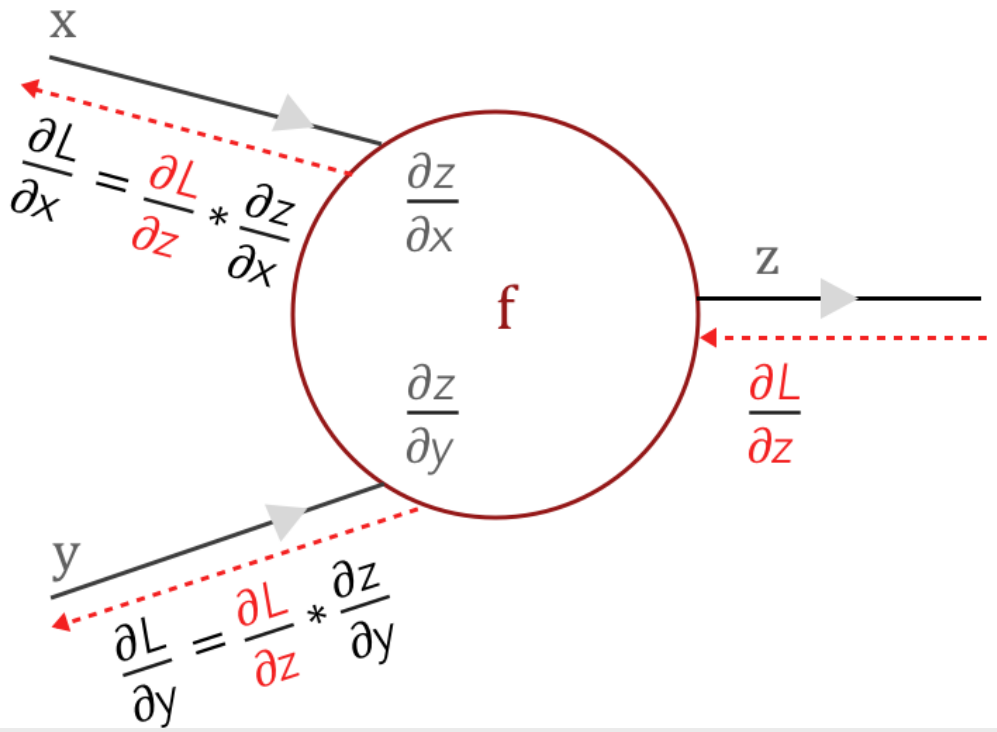
# Where we are now

- 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.

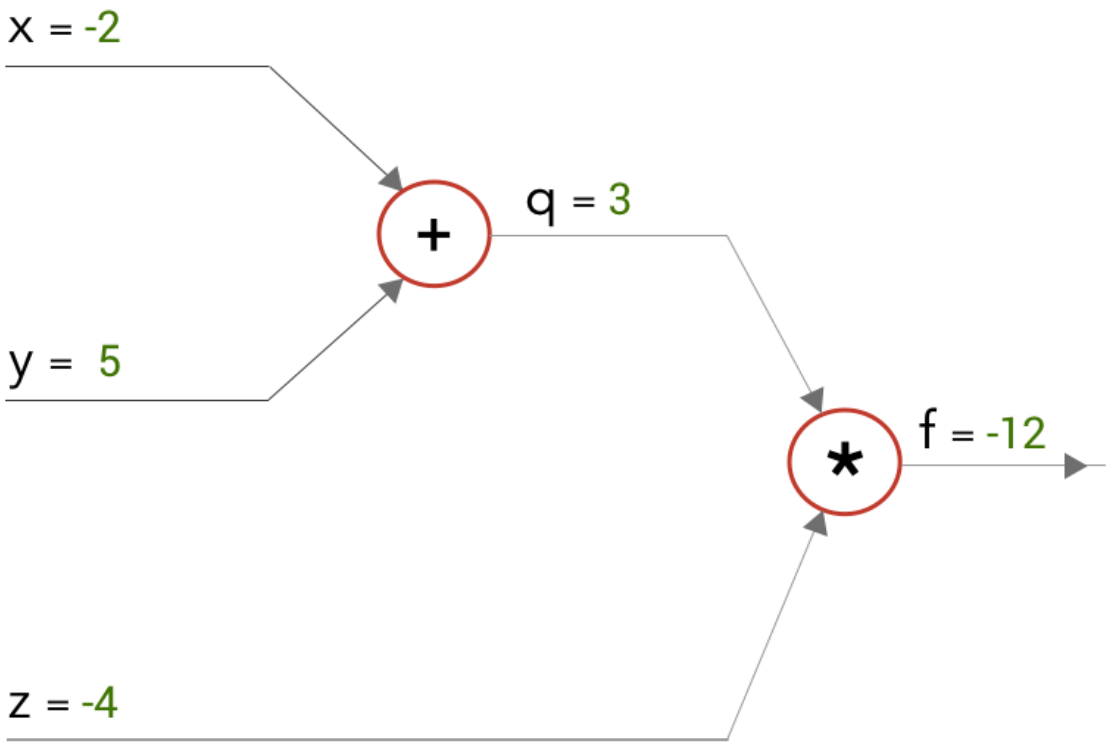



$\frac{\partial z}{\partial x}$ & $\frac{\partial z}{\partial y}$ *are local gradients*

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

# Computational graphs – example

- Consider the equation $f(x, y, z) = (x + y)z$
- Our goal is to **calculate the gradients**: ∂f/∂x, ∂f/∂y and ∂f/∂z.
- To make it simpler, let us split it into two equations.
  - $q = x + y$
  - $f = q * z$
- Now, let us draw a computational graph for it with values x = -2, y = 5, z = 4.
- **Forward pass** results in f = - 12.
- Now, let's do the **backward pass** to calculate the gradients.



# Computational graphs – example

- Use the chain rule to compute ∂f/∂x and ∂f/∂y:

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

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

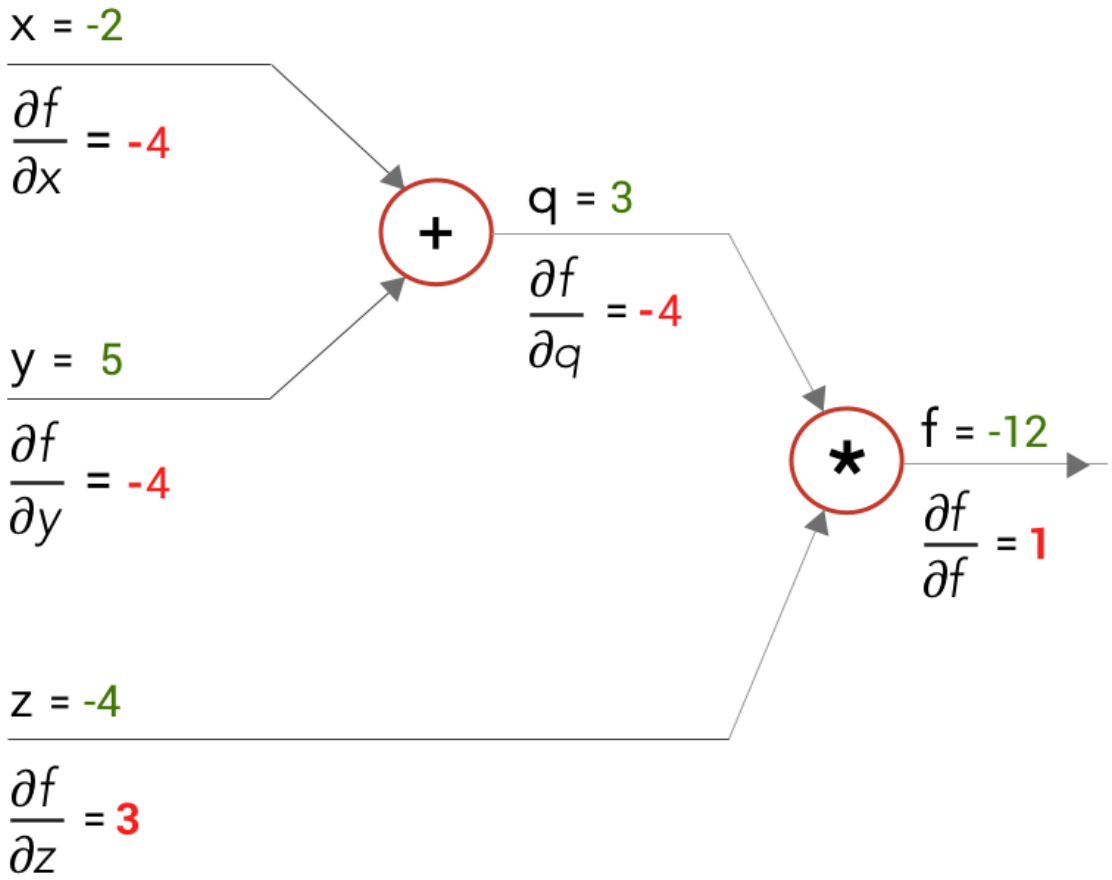
$f = q * z$

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

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

$q = x + y$

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



# 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

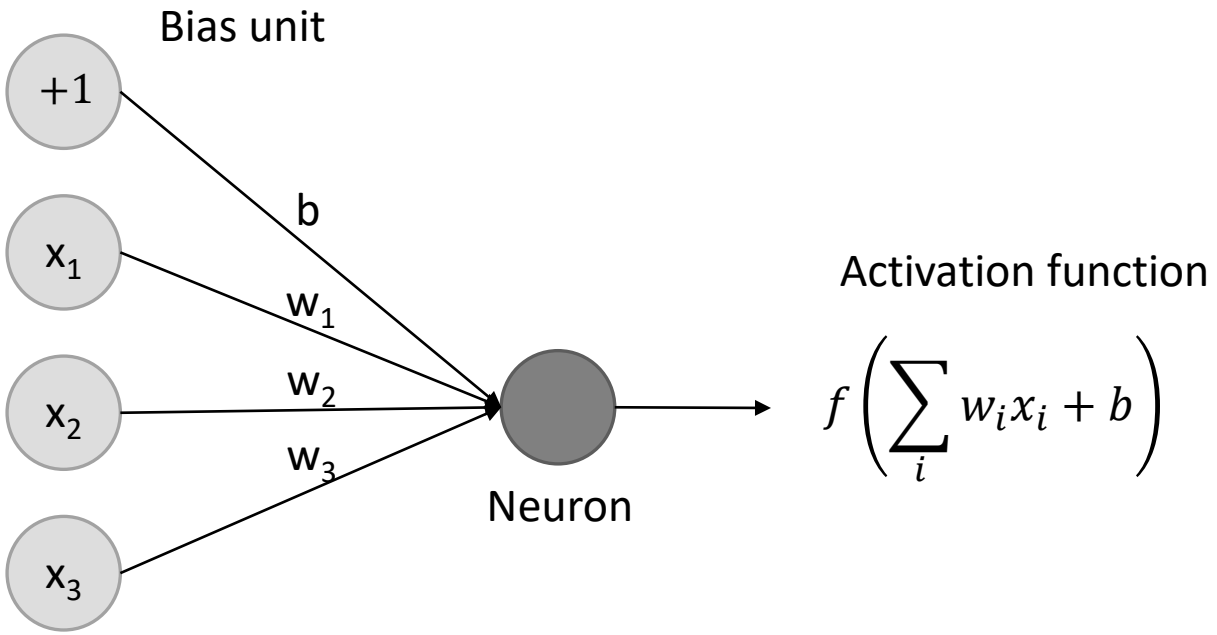
# Next: Training ConvNets

- 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

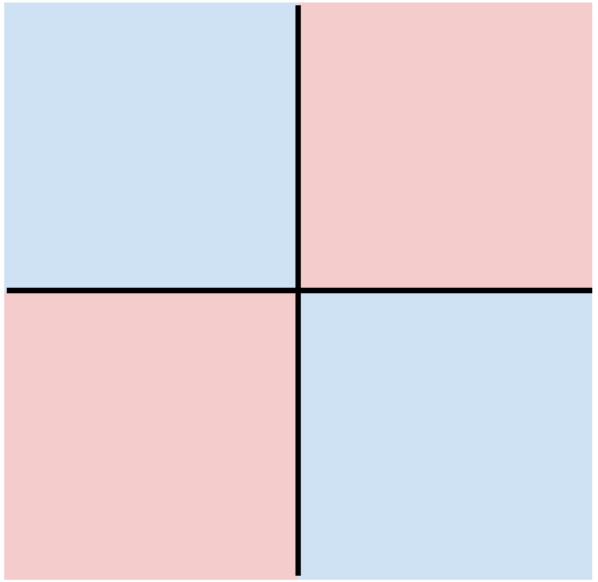
Activation functions are necessary to make our model non-linear, and we need non-linearity because …



# Hard cases for linear classifier

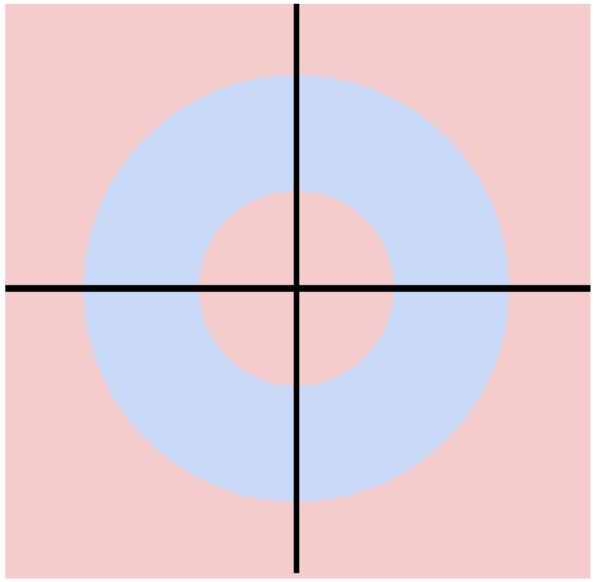
**Class 1**:
First and third quadrants

**Class 2**:
Second and fourth quadrants



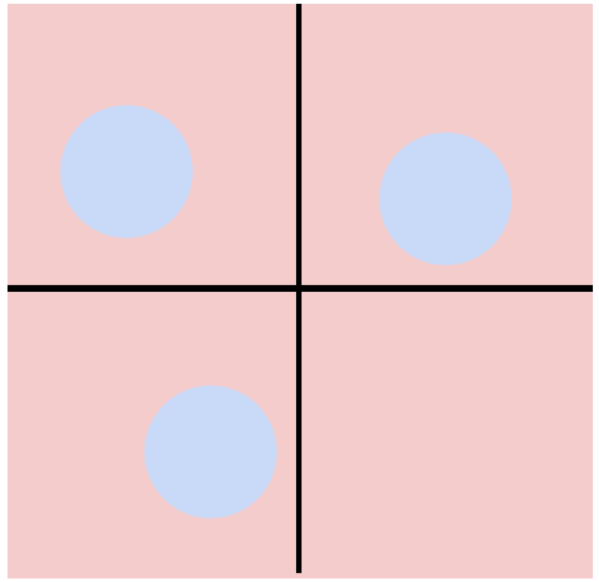
**Class 1**:
1 <= L2 norm <= 2

**Class 2**:
Everything else



**Class 1**:
Three modes

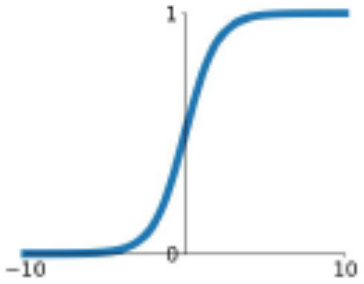
**Class 2**:
Everything else



# Examples of activation functions

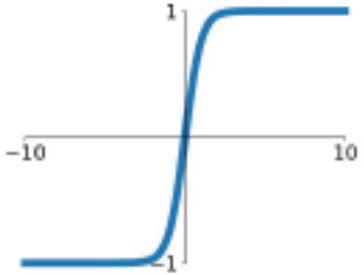
**Sigmoid**

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



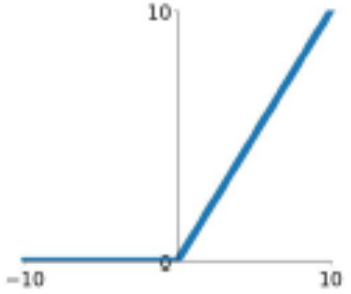

**tanh**

$$\tanh(x)$$



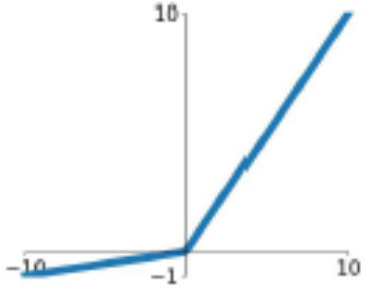

**ReLU**

$$\max(0, x)$$




**Leaky ReLU**

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

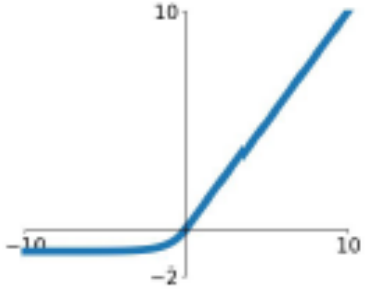



**Maxout**

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

**ELU**

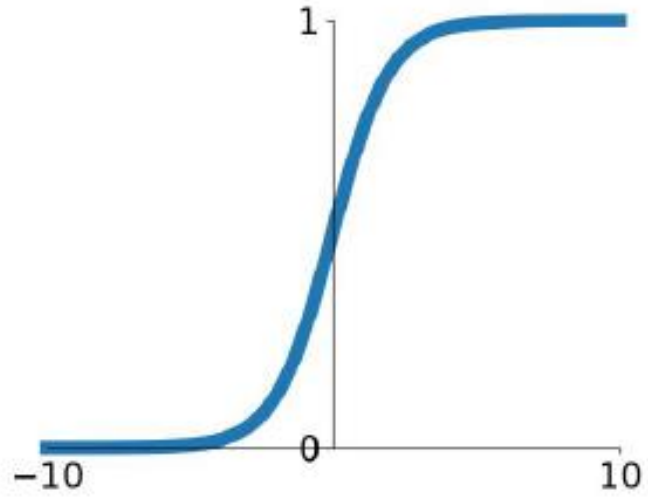
$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



# Sigmoid

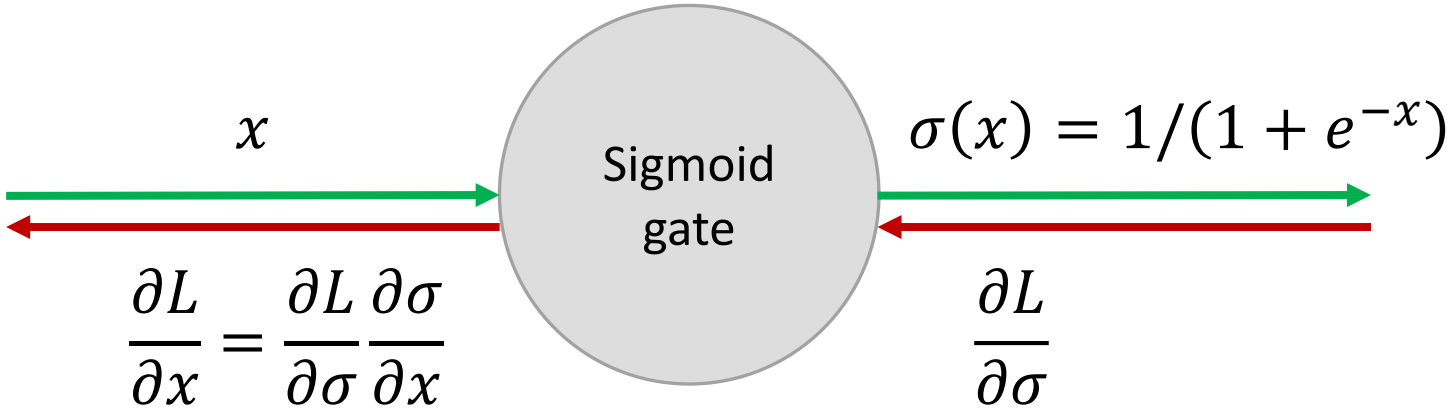
- Squashes numbers to range [0, 1].
- Useful for modelling probabilities (e.g., logistic regression).
- Historically popular, because it has a nice interpretation as a saturating “firing rate” of a neuron.
- **3 problems**
  1. Saturated neurons “kill the gradients” (see Lecture 4).
  2. Sigmoid outputs are not zero-centered
  3. exp() is a bit computational expensive

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



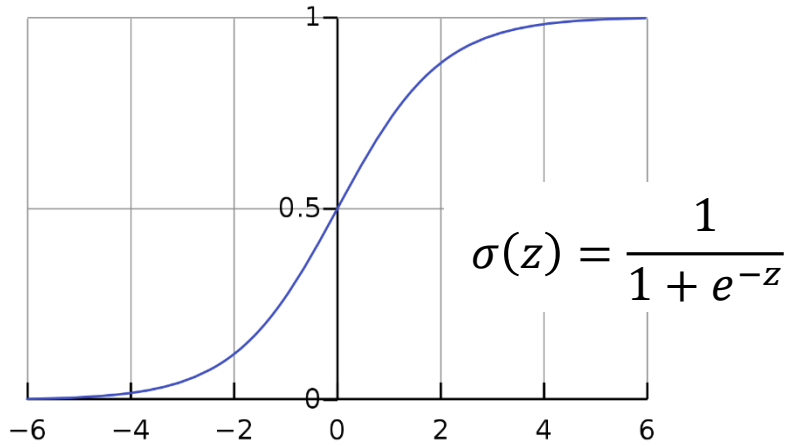

Sigmoid

# Saturated neurons “kill the gradients”



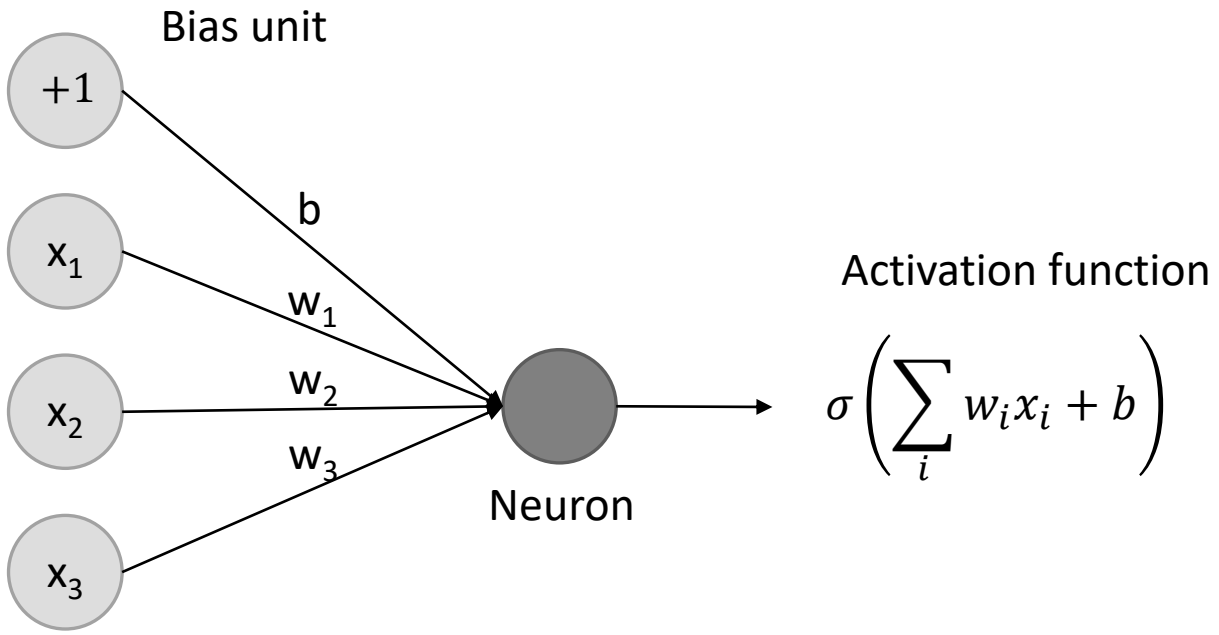

Saturation occurs when:

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

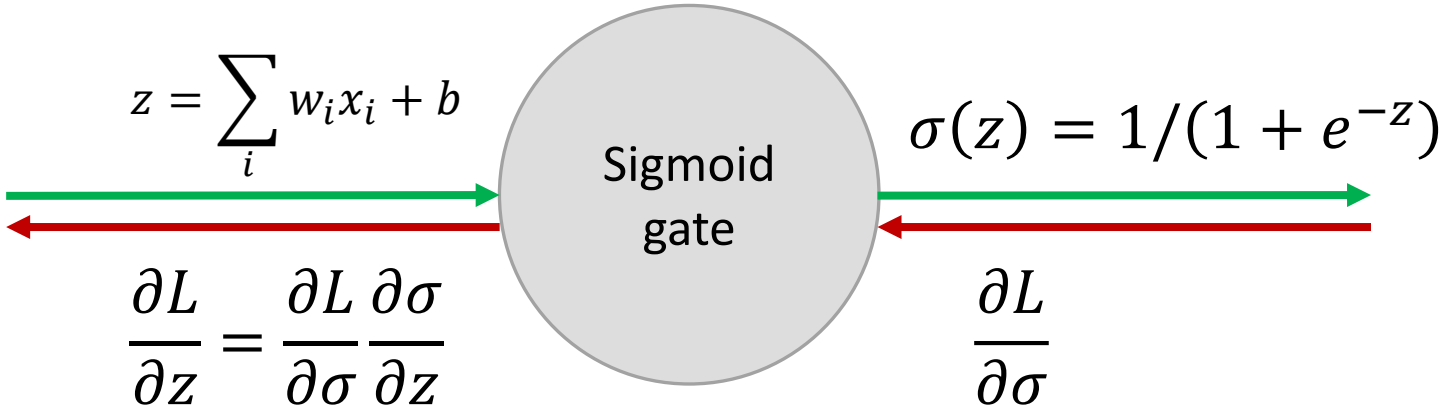


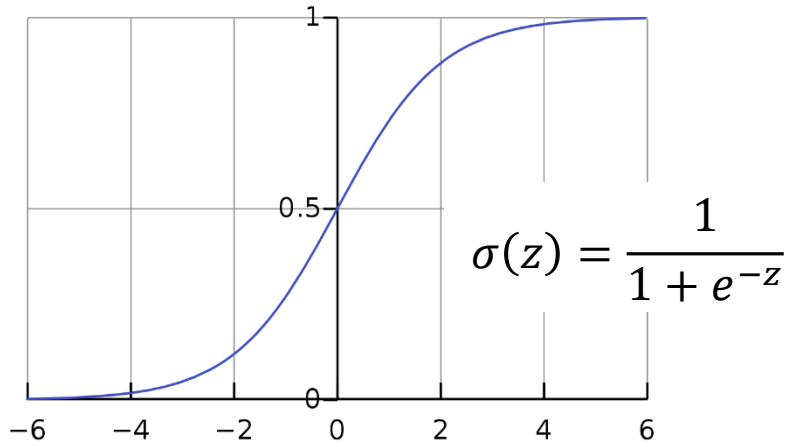
# Sigmoid outputs are not zero-centered

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

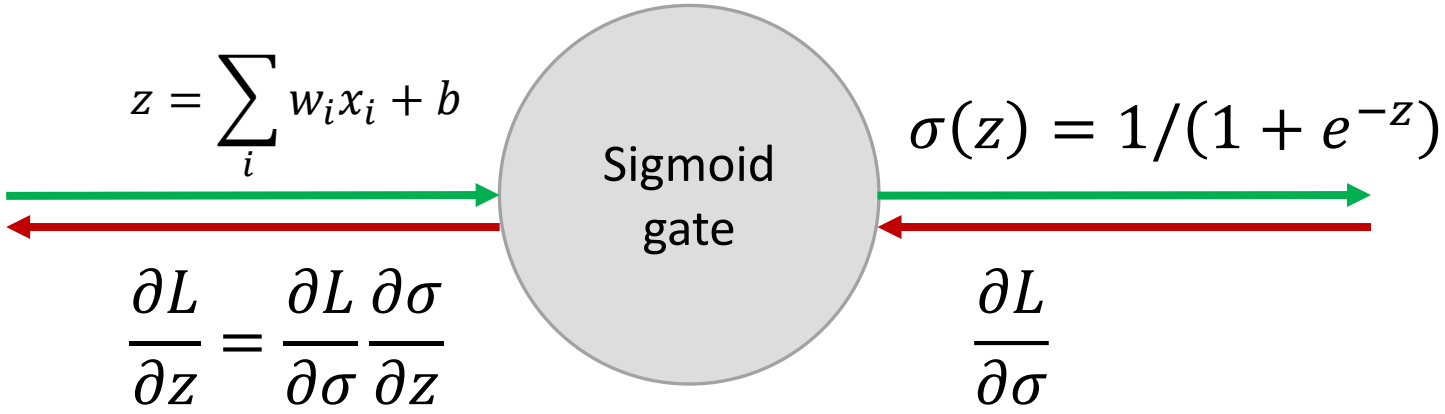


# Sigmoid outputs are not zero-centered



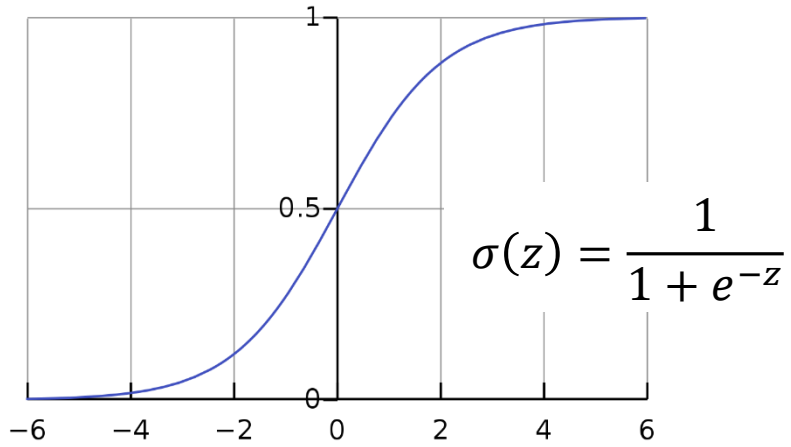



# Sigmoid outputs are not zero-centered

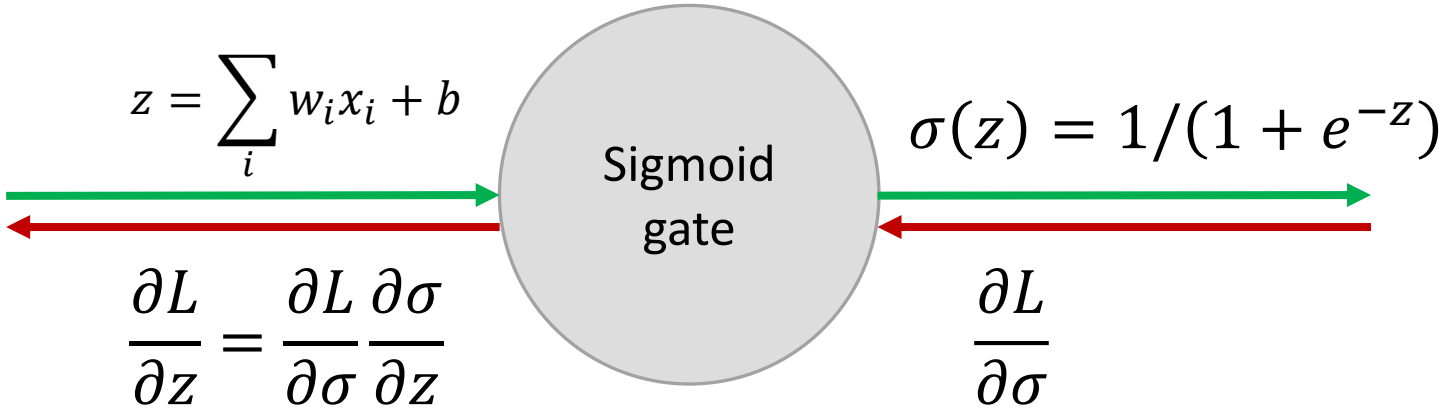



Applying the chain rule again, we get:

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



# Sigmoid outputs are not zero-centered



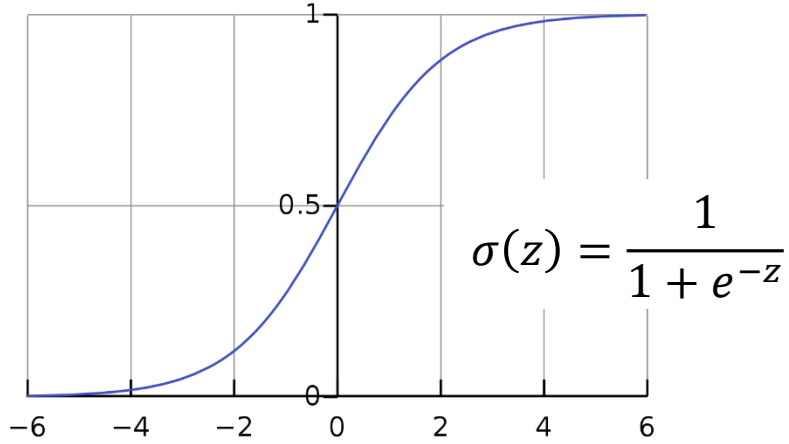

**Observation:** $\frac{\partial \sigma}{\partial z}$ is always positive

Applying the chain rule again, we get:

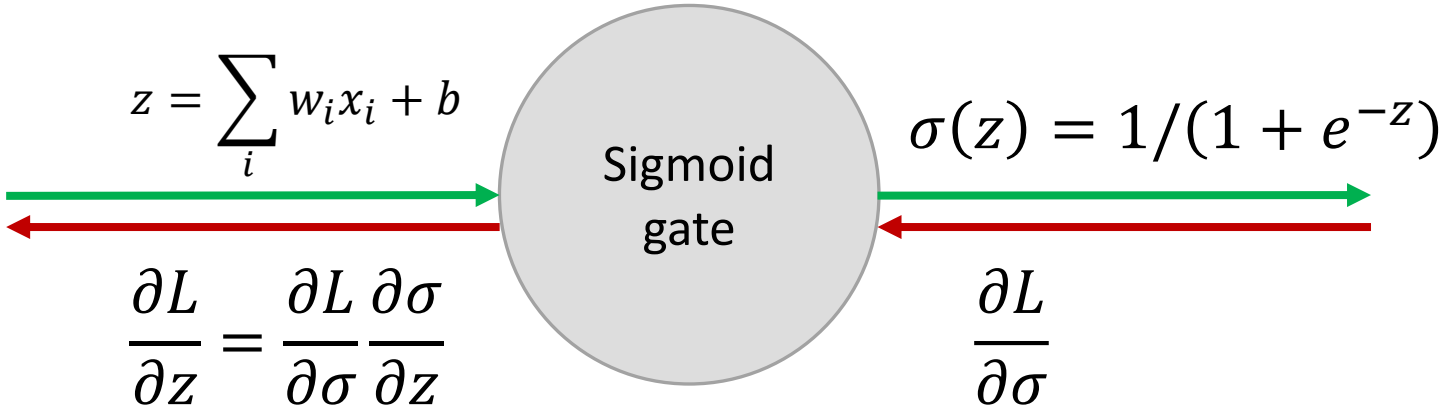
$$\frac{\partial L}{\partial w_i} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial w_i} = \frac{\partial L}{\partial \sigma}\frac{\partial \sigma}{\partial z}x_i$$

Positive (according to our assumption)

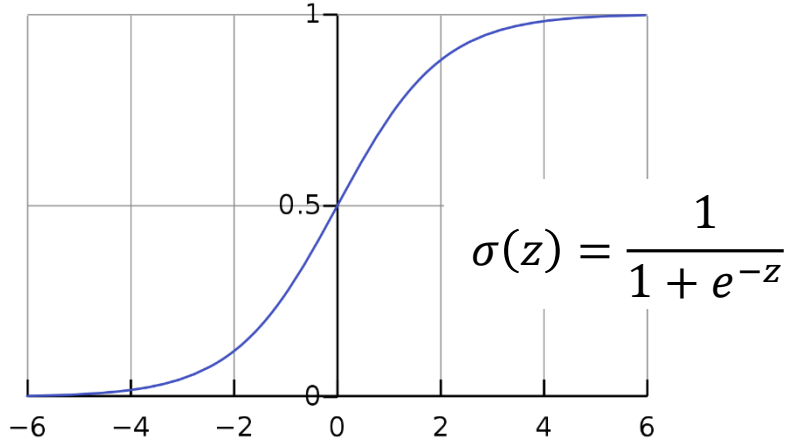
Always positive



# Sigmoid outputs are not zero-centered




**Observation:** $\frac{\partial \sigma}{\partial z}$ is always positive



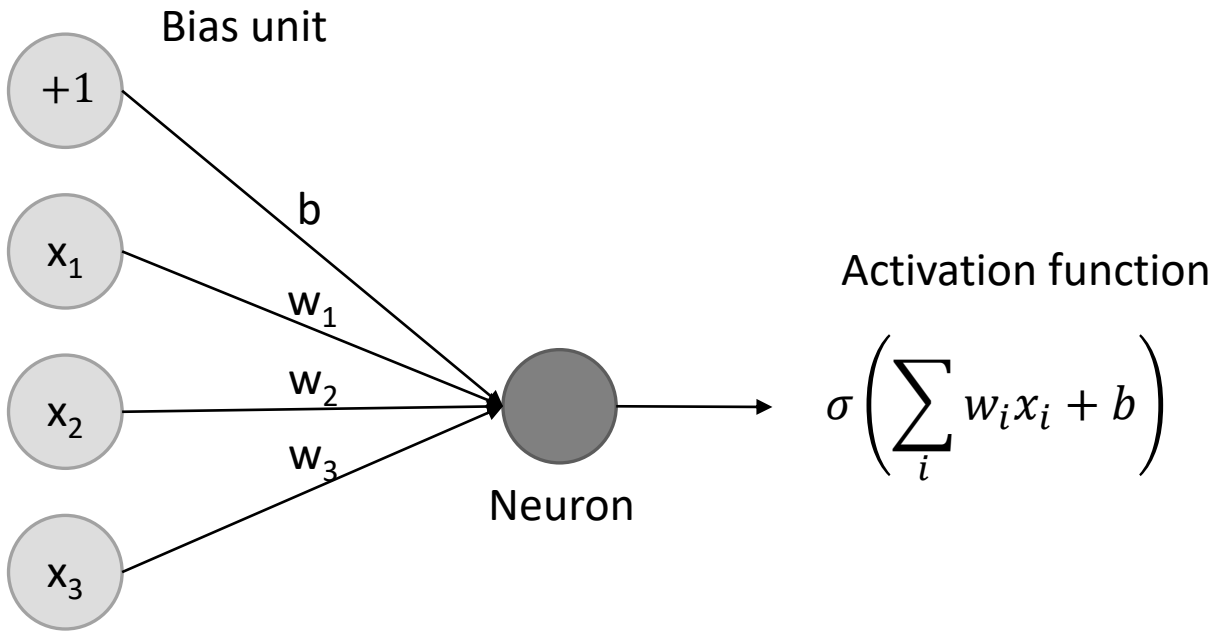

Applying the chain rule again, we get:

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

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 $\sigma$).
- **Q:** What can we say about the gradients of the loss w.r.t. **w**?
- **A:** They are all positive or all negative (at least for a single element. Mini-batches help…).

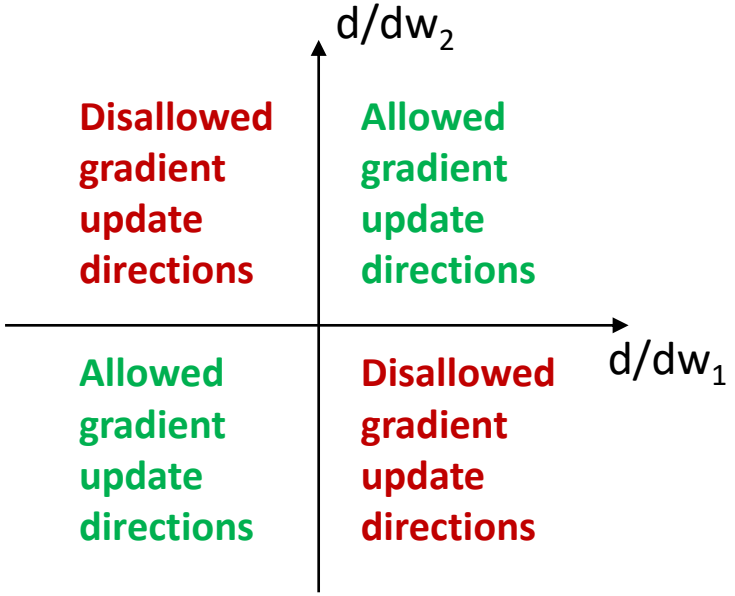


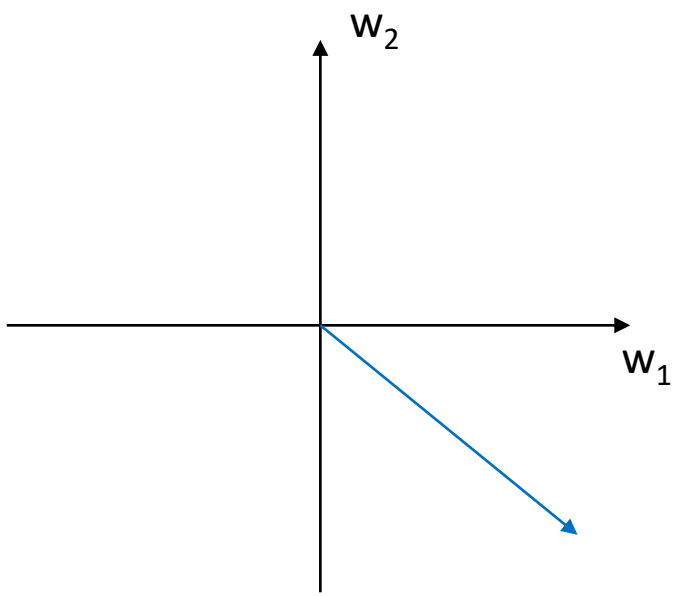
# Sigmoid outputs are not zero-centered

- Consider what happens when the input **x** to a neuron is always positive (as if produced by a $\sigma$).
- **Q:** What can we say about the gradients of the loss w.r.t. **w**?
- **A:** They are all positive or all negative (at least for a single element. Mini-batches help…).
- **Implications:**
  - If the data coming into a neuron is always positive (e.g., x > 0 elementwise), then the gradient on 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.

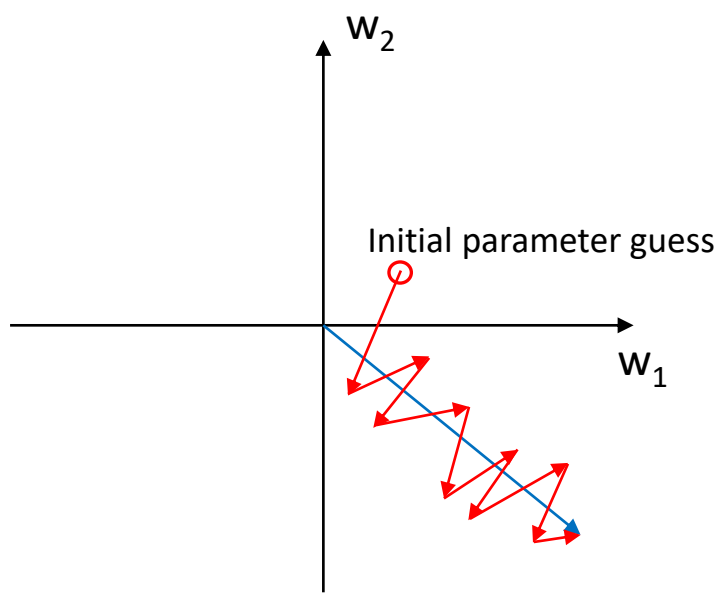
# Sigmoid outputs are not zero-centered

- Consider what happens when the input **x** to a neuron is always positive (as if produced by a $\sigma$).
- **Q:** What can we say about the gradients of the loss w.r.t. **w**?
- **A:** They are all positive or all negative (at least for a single element. Mini-batches help…).



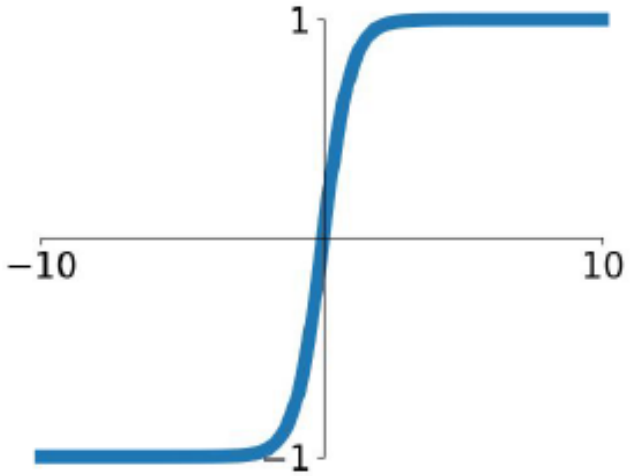

So what happens if the optimal weight points in a disallowed gradient direction?




Zig-zagging = instability

# Tanh

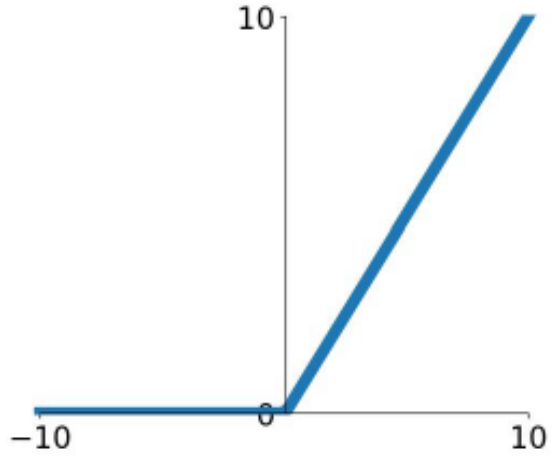
- Hyperbolic tangent
- Squashes numbers to range [-1, 1].
- Outputs are zero-centered (nice)
- Still kills gradient when saturated




tanh(x)

# ReLU

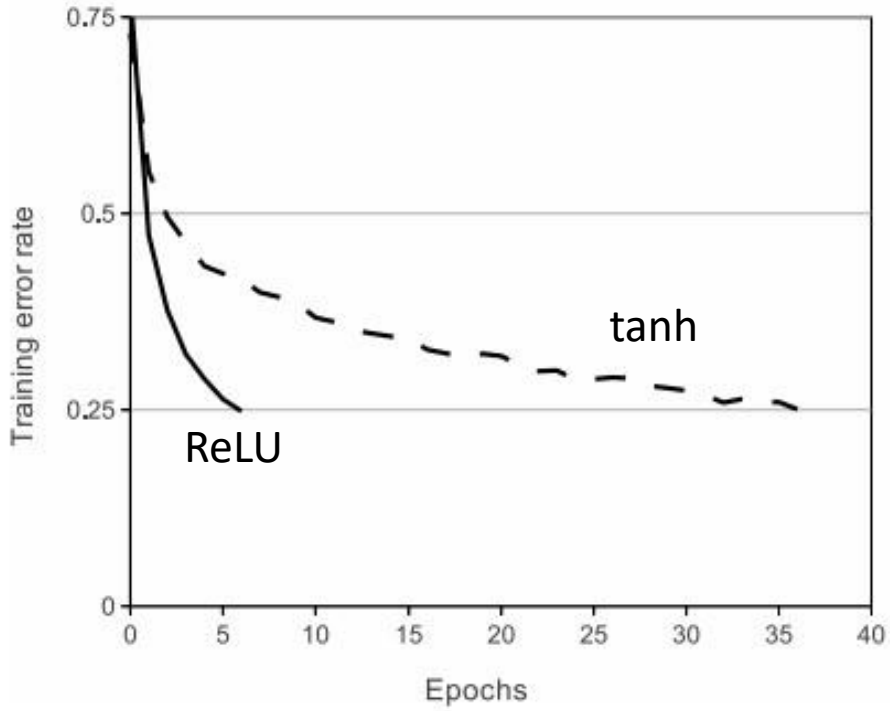
- Computes $f(x) = \max(0, x)$
- Does not saturate (in +region)
- Very computationally efficient
- Converges much faster than sigmoid/tanh in practice (e.g. 6x) [Krizhevsky et al., 2012]




**ReLU**
(Rectified Linear Unit)

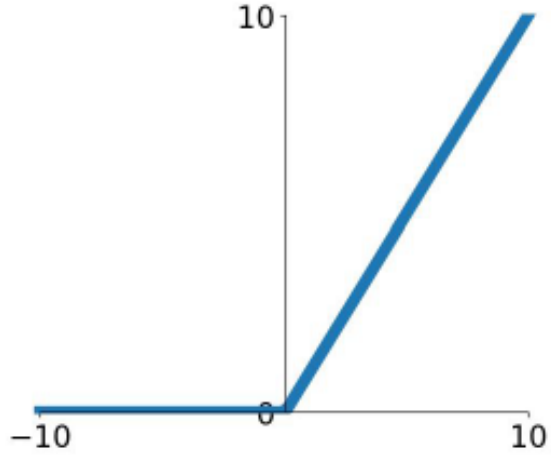
# ReLU

- 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

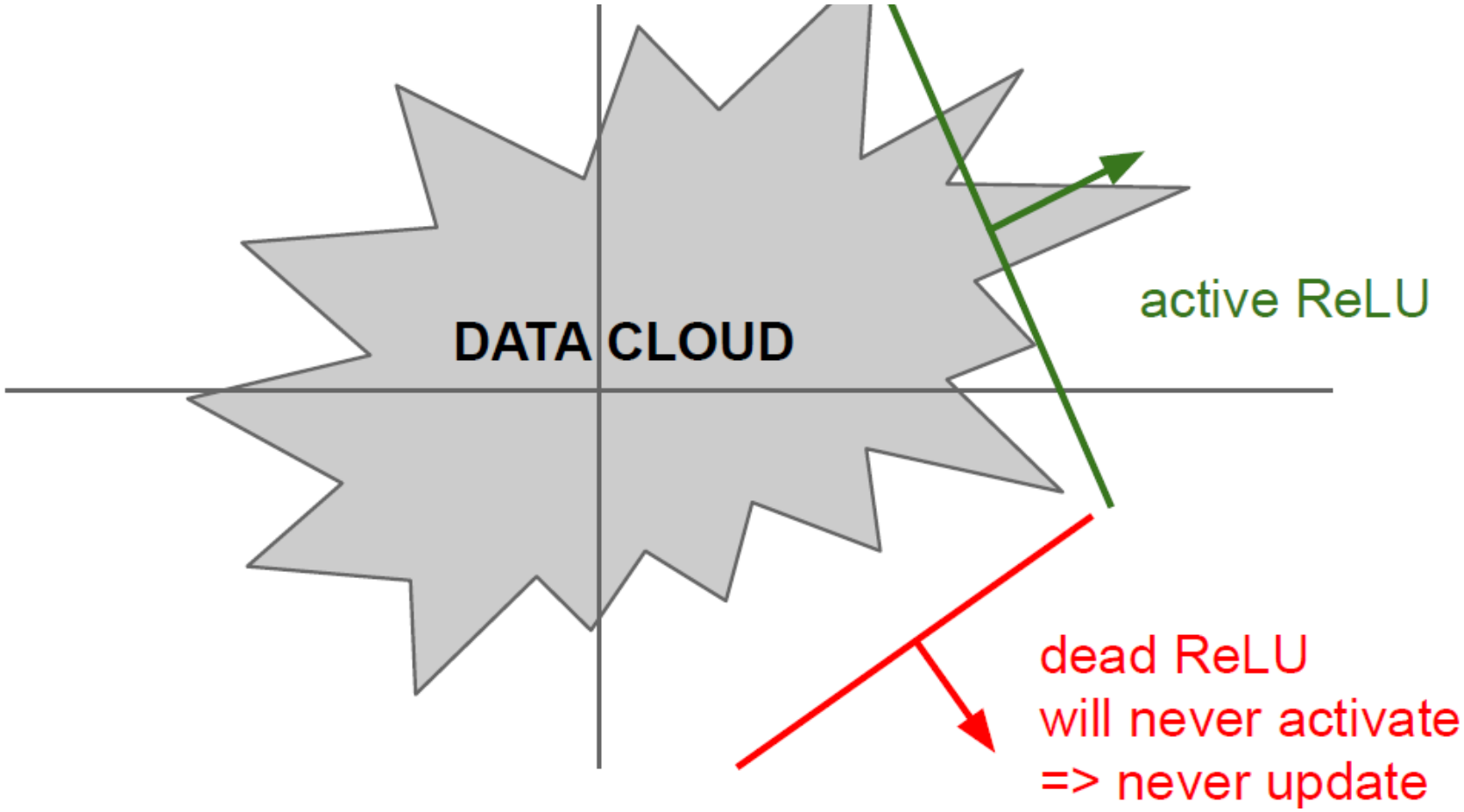
- Computes $f(x) = \max(0, x)$
- Does not saturate (in +region)
- Very computationally efficient
- Converges much faster than sigmoid/tanh in practice (e.g. 6x) [Krizhevsky et al., 2012]
- Not zero-centered output
- No gradient flow when x < 0 causing the neuron to "die":
  - A large gradient flowing 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)

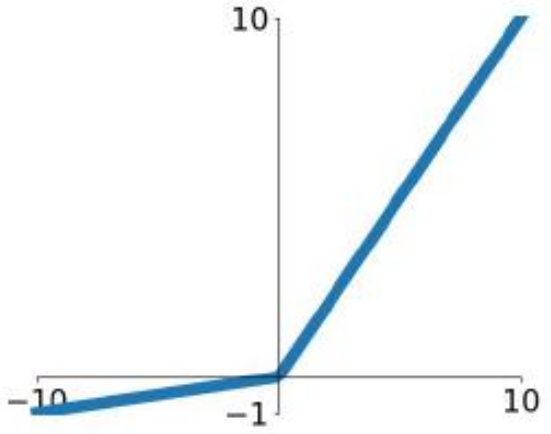
# Dead ReLU

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

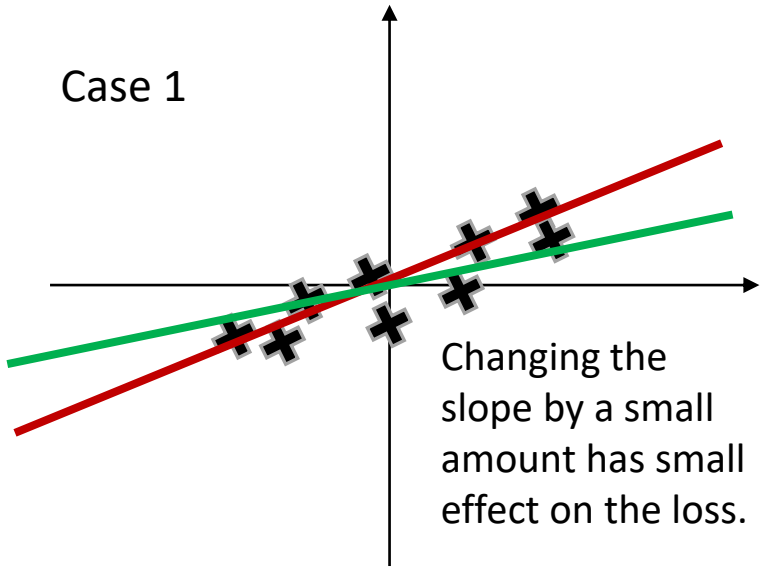
- Use **ReLU**, but be careful with your learning rates.
- Try out **Leaky ReLU** (or more exotic alternatives not covered here, like Maxout or ELU)
- Try out **tanh** but don't expect much.
- Don't use **sigmoid**, except maybe at the output layer, if you want values that reflect probabilities.

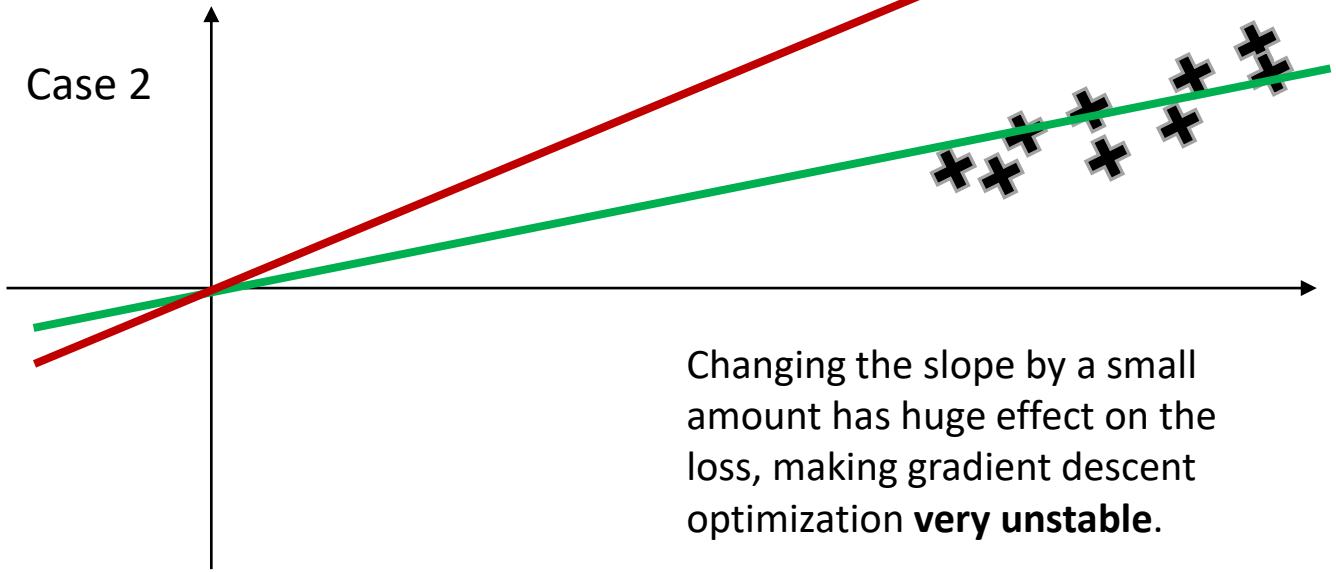
# Data preprocessing

ALSO CALLED NORMALIZATION OR ZERO-CENTERING

# Why we prefer zero-centered data

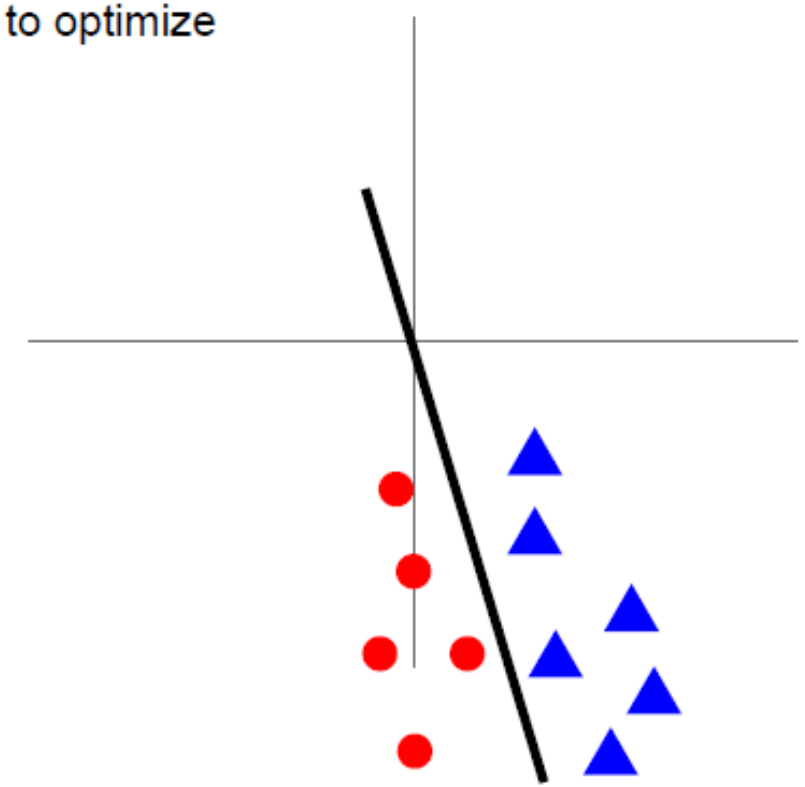
- Your task is to fit a straight line to the datapoints shown below.
- **Q:** Which of the two cases is going to be the more challenging one? Why?
- **A:** Case 2 is harder, because the datapoints lie “far” away from the origin. Small changes in the slope (weights) cause large changes in the loss. The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



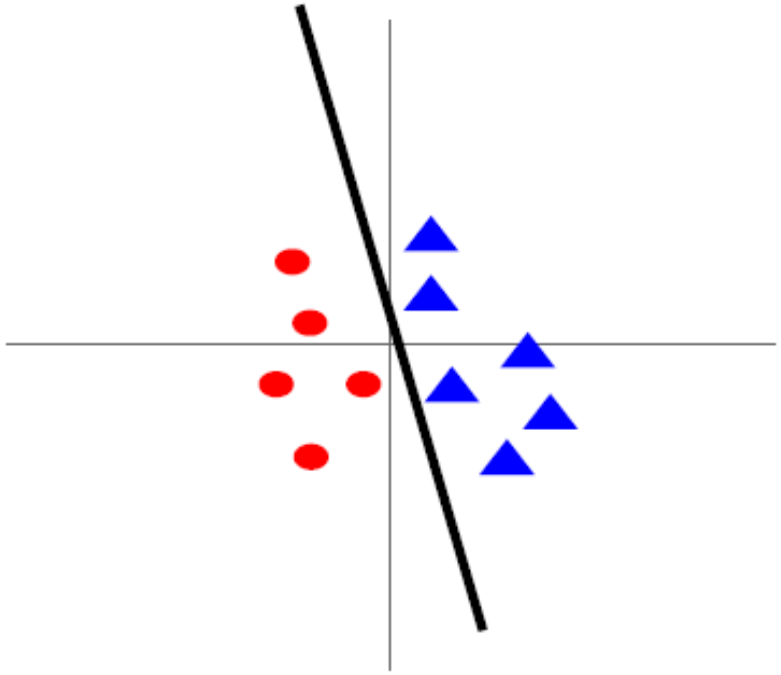



# Classification example

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

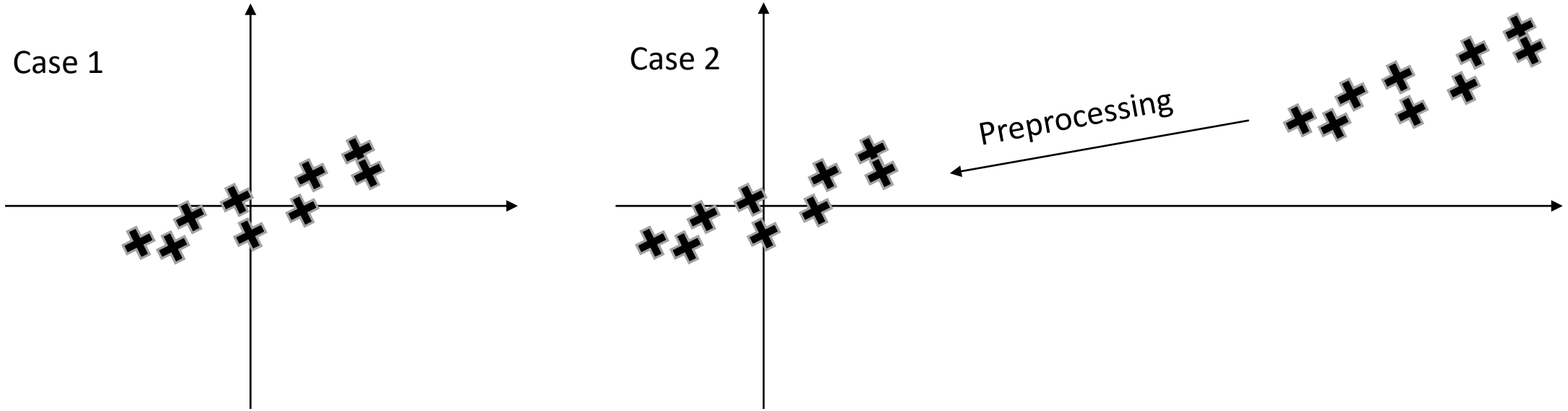


**After normalization**: less sensitive to small changes in weights; easier to optimize



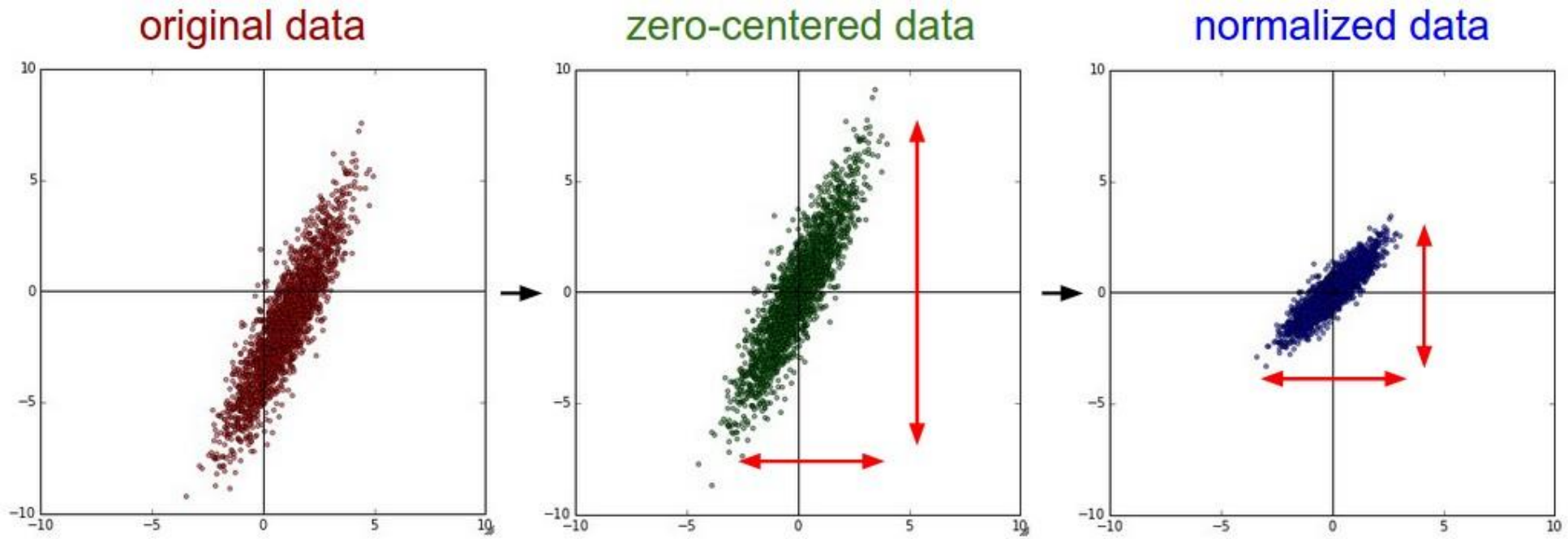
# Why we prefer zero-centered data

- **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**).



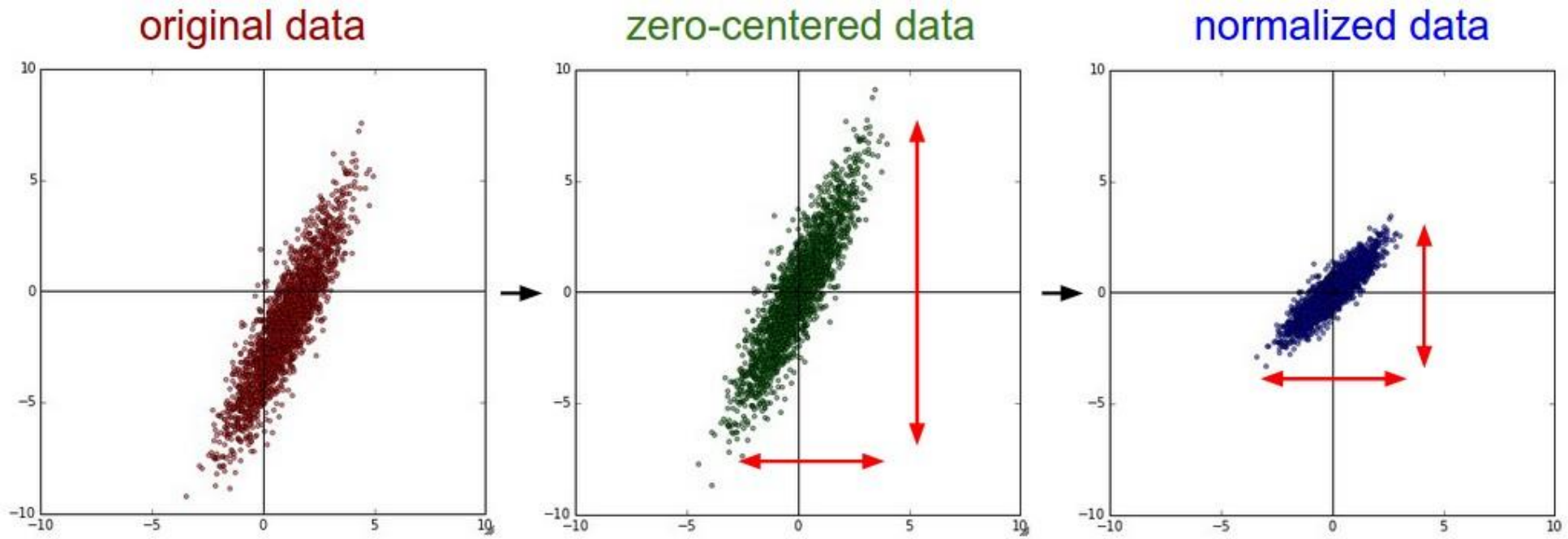
# Mean subtraction and normalization

- **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)`.



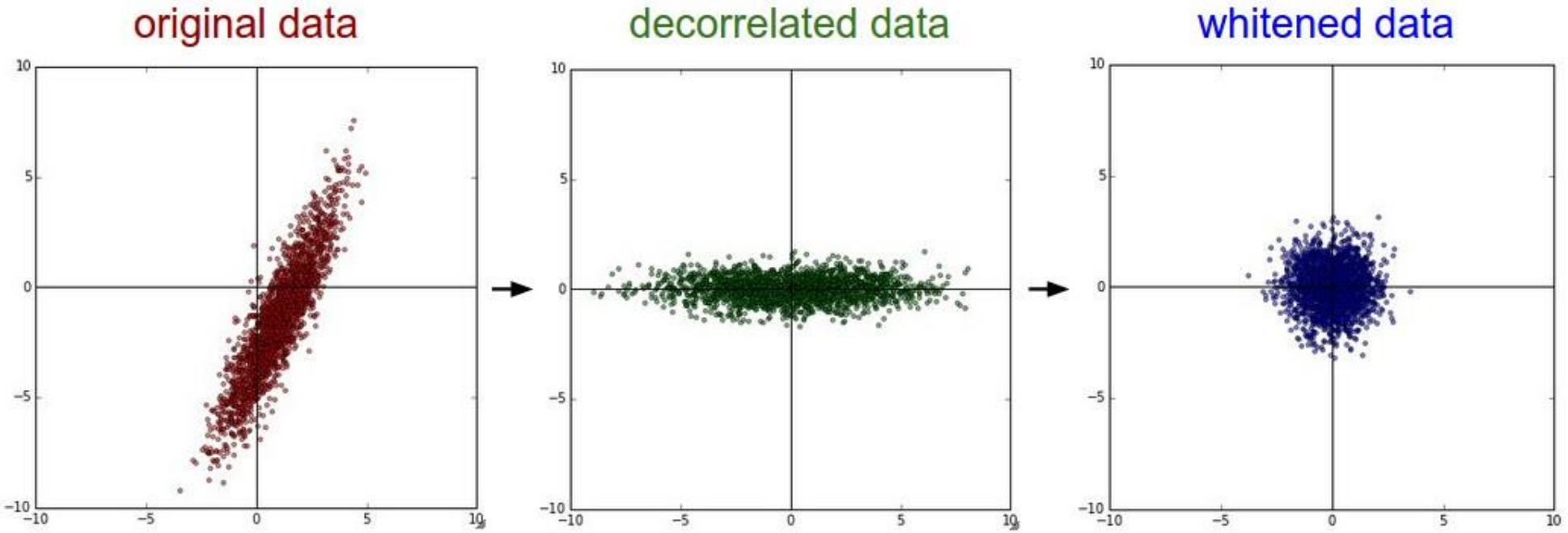
# Mean subtraction and normalization

- **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 todays neural networks.
- **PCA – Principal Component Analysis:** Basically 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 every pixel 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

- In practice for images: Center only
- 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

- Lets start with what we should not do (also see Lecture 4).
- With proper data normalization it is reasonable to **assume that approximately half of the weights 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.
- This turns out to be a mistake, 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**.
- In other words, there is **no source of asymmetry between neurons** if their weights are initialized to be the same.
- Note: Its safe to initialize the biases to zero.

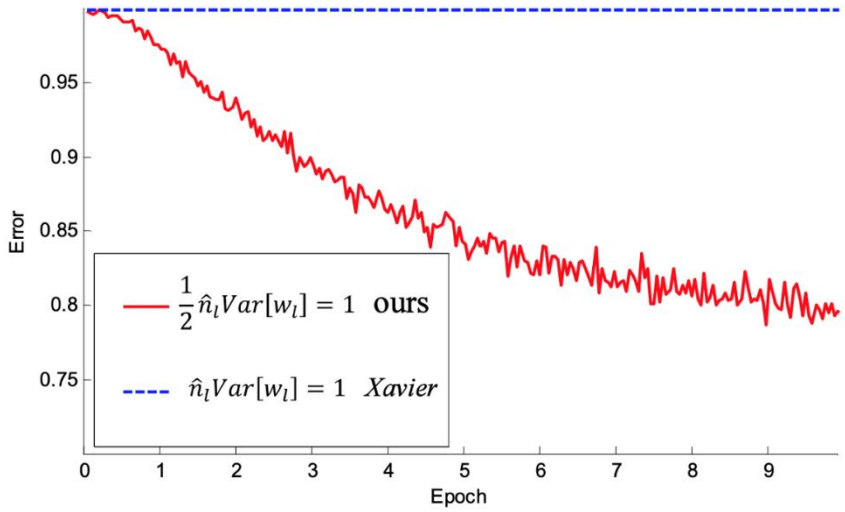
# Small random numbers

- 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)
```

# 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:

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

**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.

# 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:

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

**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 (mean 0 and a standard deviation 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 a mean of 0 and a standard deviation of 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

```
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

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 (mean 0 and a standard deviation 1), the layer outputs got so big that even the computer wasn't able to recognize their standard deviation and mean as numbers.

Initializing the weights with a normal distribution with a mean of 0 and a standard deviation of 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, …).

# 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))
```

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:

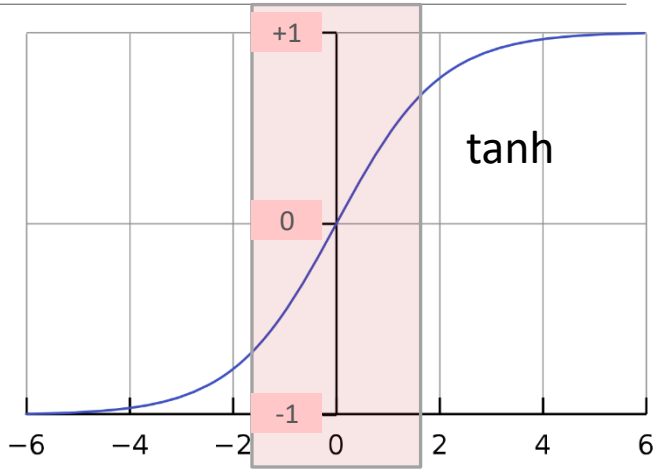
- 6 layers
- 4096 dimensional input
- 16 neurons in each layer
- tanh activation

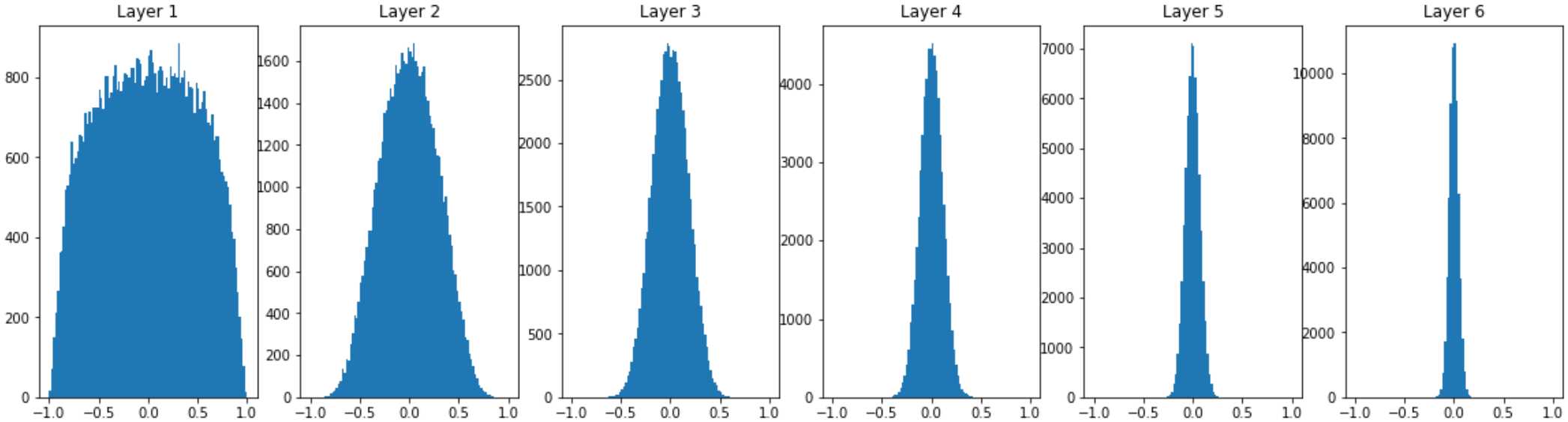
Initialize weights with small random values:

- 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.



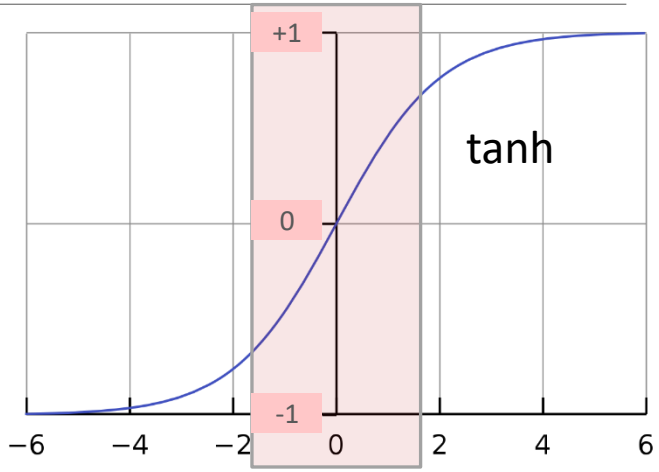
# Visualizing activation statistics

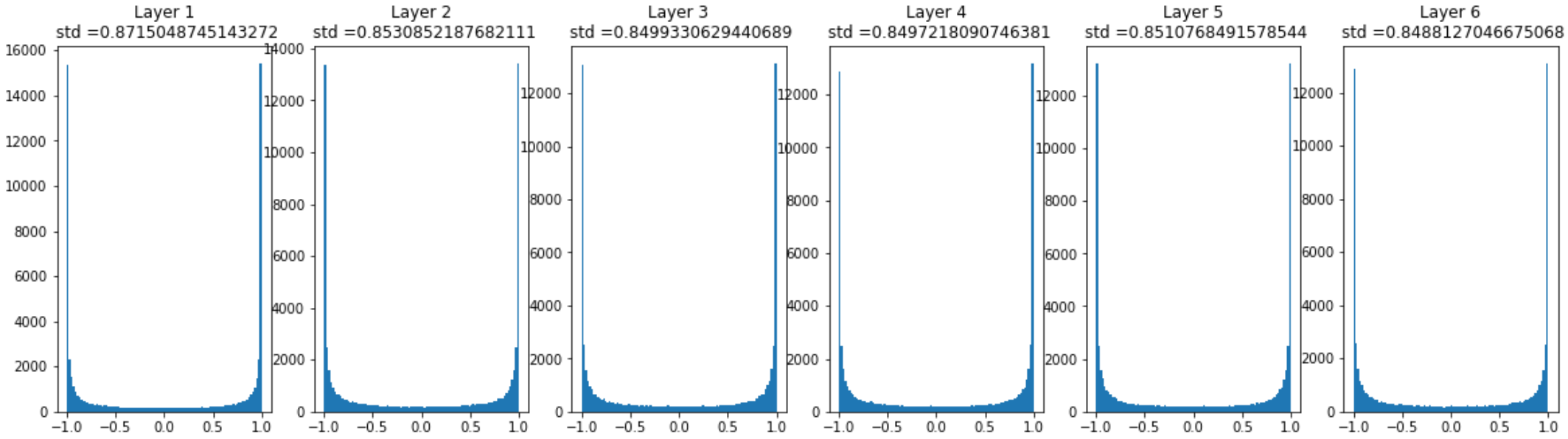
```
import numpy as np

# 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…



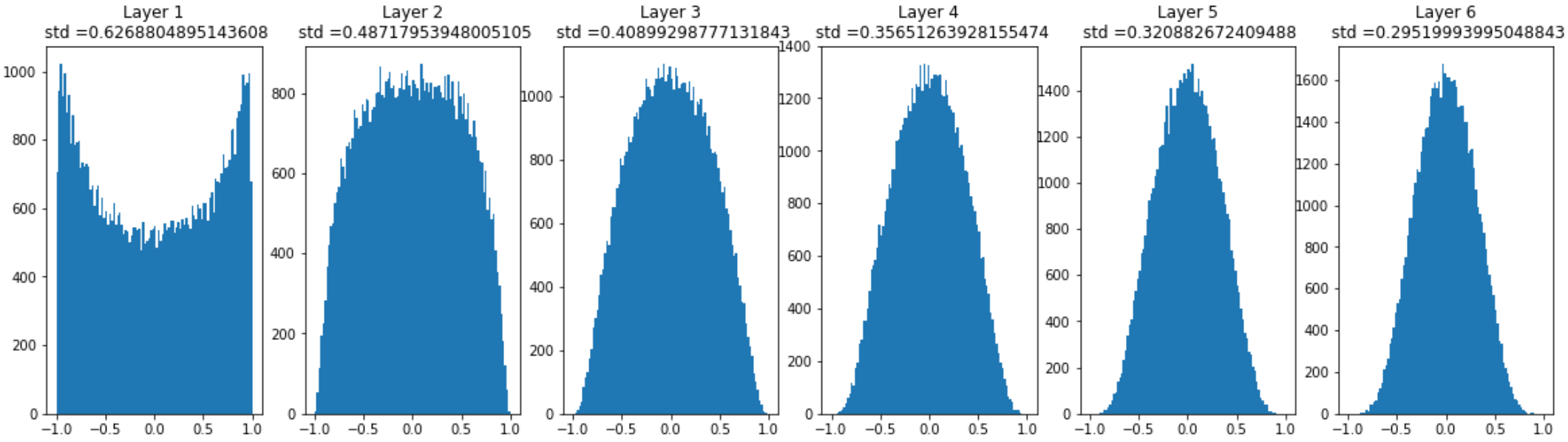
# Visualizing activation statistics

```
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.
- For conv layers, $D_{in}$ is kernel_size$^2$ * input_channels
- Why does it work?
- See derivation here: https://www.deeplearning.ai/ai-notes/initialization/



# 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 replace each input dimension of x with its 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.

# 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)

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

(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** $\gamma_j$, $\beta_j$, 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

$\mu_j$ = (running) average of values seen during training

$\sigma_j^2$ = (running) 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).*

**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)

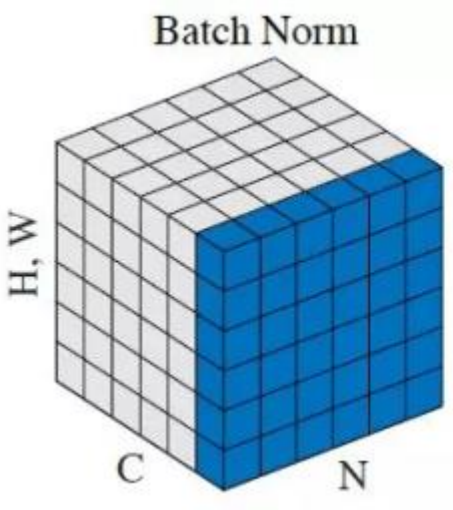
$Y_{ijmn} = \gamma_j(X_{ijmn} - \mu_j)/\sigma_j + \beta_j$

*Separate mean and variance for each color channel (C).*

***N* is the batch size**

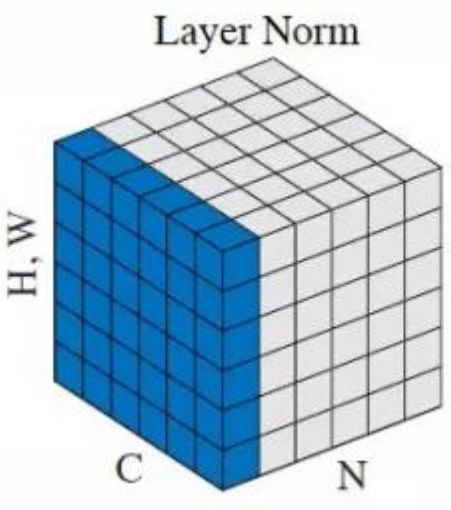
# Other variants

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



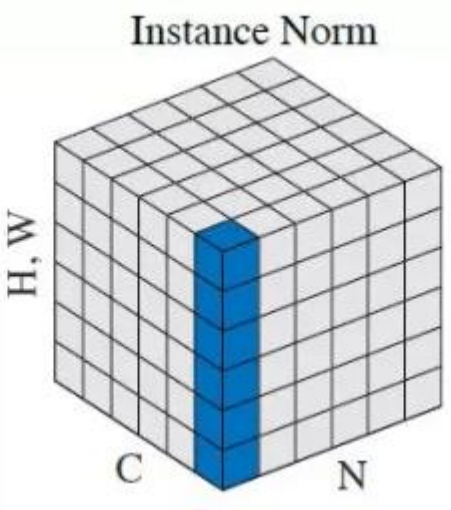

https://arxiv.org/abs/1502.03167

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

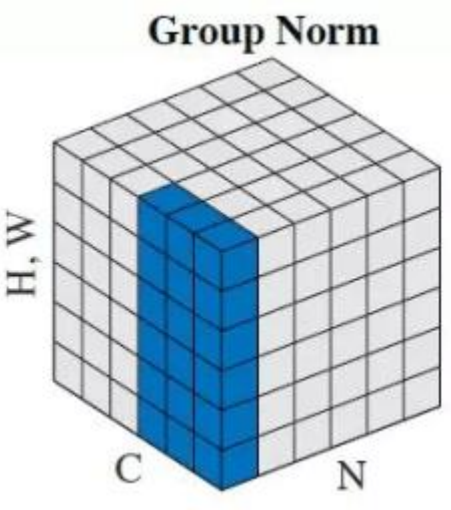



https://arxiv.org/abs/1607.06450

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




https://arxiv.org/abs/1701.02096



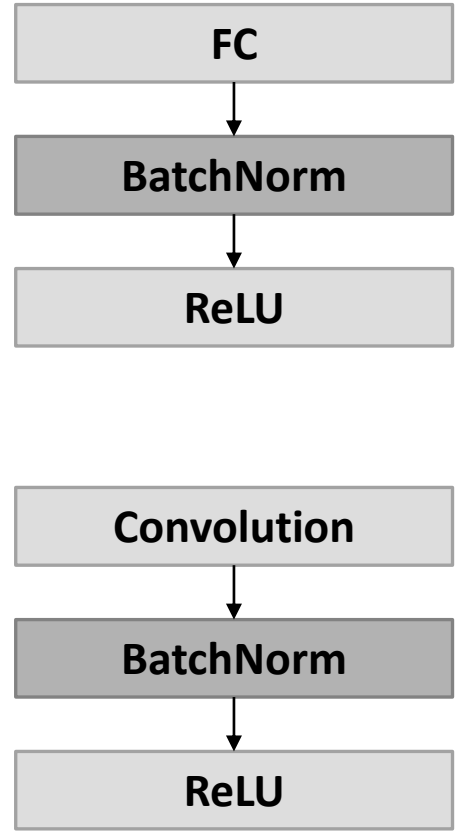

https://arxiv.org/abs/1803.08494

# The batch normalization layer

```
mobilenet_full.summary()
```

```
WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/kera
Instructions for updating:
Please use `rate` instead of `keep_prob`. Rate should be set to `ra
Downloading data from https://github.com/fchollet/deep-learning-moc
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

- In traditional deep networks, too-high learning rate may result in the gradients that explode or vanish, as well as getting stuck in poor local minima.
- 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.

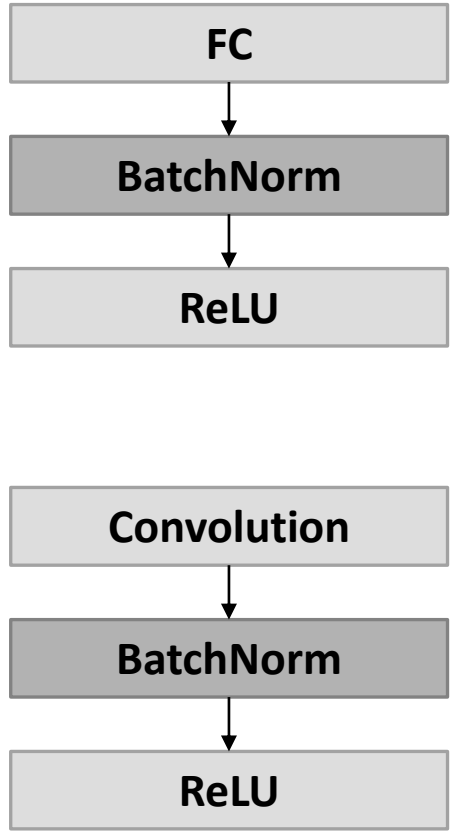
# Batch norm regularizes the model

- 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), 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
- 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 conv!

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



# Stochastic Gradient Descent (SGD) and momentum

# Recall: Gradient descent

- Gradient descent is a way to **minimize a loss function** $J(w)$ of a model $h_w(x)$, parameterized by a set of parameters $w$.
- It works by updating the parameters in the opposite direction of the gradient $\nabla J(w)$ of the loss function w.r.t. to the parameters $w$:

$$w = w - \alpha \nabla J(w)$$

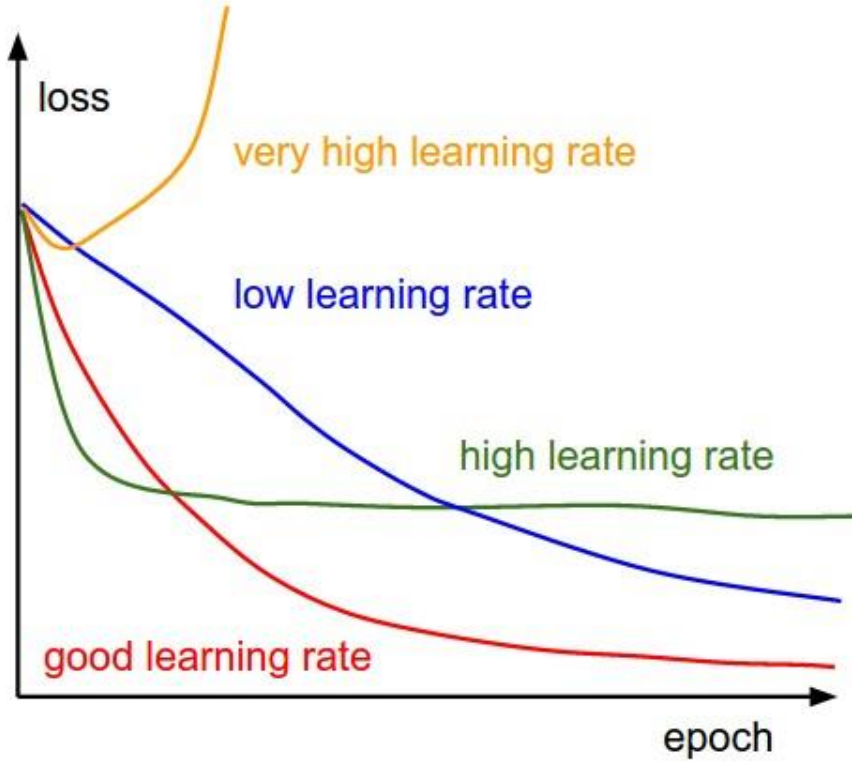
- The gradient is sometimes written $\nabla_w J(w)$.
- The learning rate $\alpha$ determines the size of the steps we take to reach a (local) minimum.
- In other words, we follow the direction of the slope of the surface created by the loss function downhill until we reach a valley.

# Gradient descent variants

- There are basically two variants of gradient descent.
- **Batch gradient descent:** Vanilla gradient descent, aka batch gradient descent, computes the gradient of the loss function w.r.t. to the parameters for the **entire training dataset**.
- **Stochastic gradient descent (SGD):** Mini-batch gradient descent, aka stochastic gradient descent, estimates the gradient using a small, random set of *n* samples, called a **mini-batch**.
- Notes on SGD:
  - *n* is commonly referred to as the **batch size**
  - Increasing *n* reduces the variance of the parameter updates, which can lead to more stable convergence.
  - On the other hand, decreasing *n* creates random fluctuations the help SGD avoid local minima.
  - Deep learning libraries can make use of highly optimized matrix optimizations (on the GPU) that make computing the gradient w.r.t. a mini-batch very efficient. It is a common practise to use batch sizes of powers of 2 to offer better run-time with GPUs.
  - Some define SGD as $n = 1$ and mini-batch gradient as $n > 1$. We will not distinguish between the two.

# Challenges with standard SGD

- Choosing a proper learning rate can be difficult.
- **Learning rate schedulers** try to adjust the learning rate during training, but have to be defined in advance and are unable to adapt to a dataset's characteristics.
- The same learning rate applies to all parameters, ignoring the possibility that some features are more important than others.



# Challenges with standard SGD

- Choosing a proper learning rate can be difficult.
- **Learning rate schedulers** try to adjust the learning rate during training, but have to be defined in advance and are unable to adapt to a dataset's characteristics.
- The same learning rate applies to all parameters, ignoring the possibility that some features are more important than others.
- **SGD has trouble navigating ravines**, i.e. areas where the surface curves much more steeply in one dimension than in another, which are common around local optima.

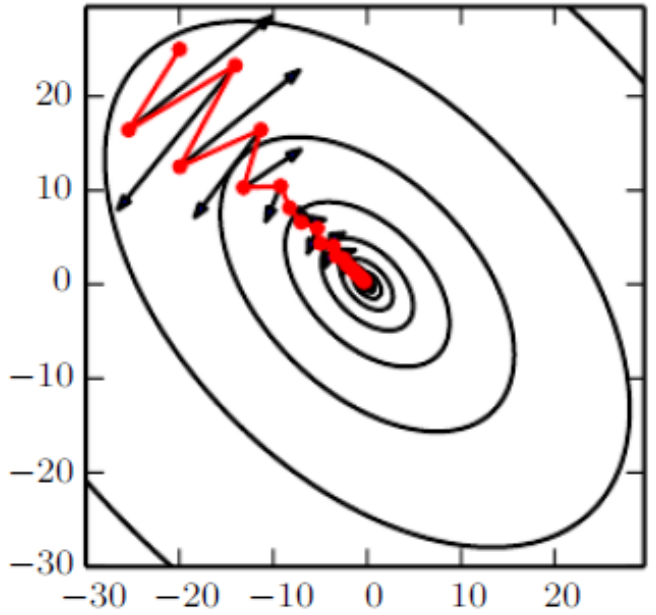
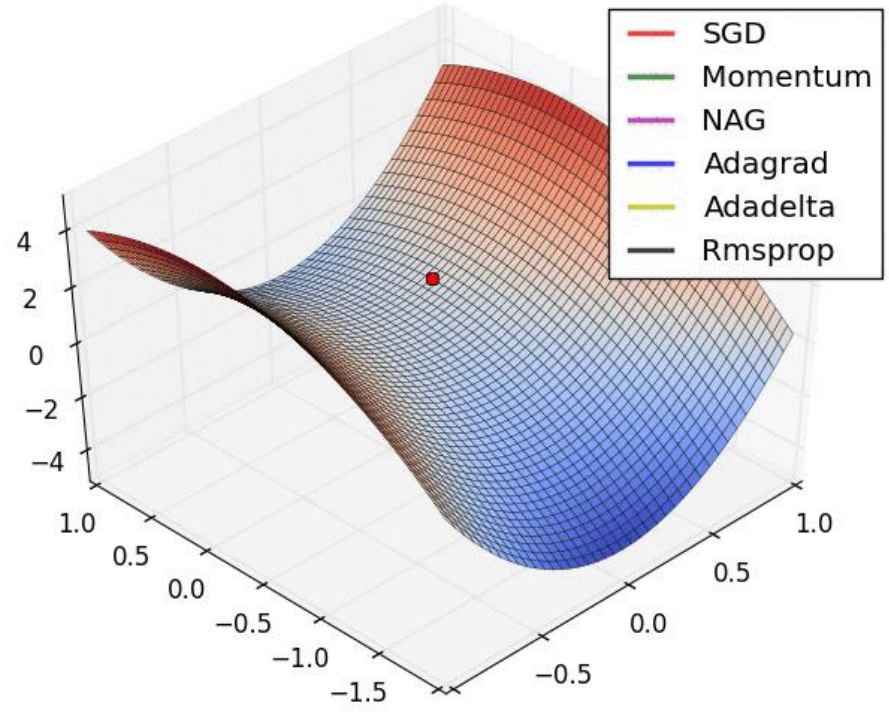



Figure 8.5 (book)

# Challenges with standard SGD

- Choosing a proper learning rate can be difficult.
- **Learning rate schedulers** try to adjust the learning rate during training, but have to be defined in advance and are unable to adapt to a dataset's characteristics.
- The same learning rate applies to all parameters, ignoring the possibility that some features are more important than others.
- **SGD has trouble navigating ravines**, i.e. areas where the surface curves much more steeply in one dimension than in another, which are common around local optima.
- Another key challenge of minimizing highly non-convex error functions common for neural networks is **avoiding getting trapped** in their numerous suboptimal local minima and/or saddle points (i.e., points where one dimension slopes up and another slopes down, and there is zero gradient).




Link to GIF animation:
http://ruder.io/content/images/2016/09/saddle_point_evaluation_optimizers.gif

# Challenges with standard SGD

- Choosing a proper learning rate can be difficult.
- **Learning rate schedulers** try to adjust the learning rate during training, but have to be defined in advance and are unable to adapt to a dataset's characteristics.
- The same learning rate applies to all parameters, ignoring the possibility that some features are more important than others.
- **SGD has trouble navigating ravines**, i.e. areas where the surface curves much more steeply in one dimension than in another, which are common around local optima.
- Another key challenge of minimizing highly non-convex error functions common for neural networks is **avoiding getting trapped** in their numerous suboptimal local minima and/or saddle points (i.e., points where one dimension slopes up and another slopes down, and there is zero gradient).

These problems become more likely in higher dimensions.

# Momentum

- SGD has trouble navigating ravines, i.e. areas where the surface curves much more steeply in one dimension than in another, which are common around local optima.
- In these scenarios, SGD oscillates across the slopes of the ravine while only making hesitant progress along the bottom towards the local optimum as shown in Image 2 below:

Image 2: SGD without momentum

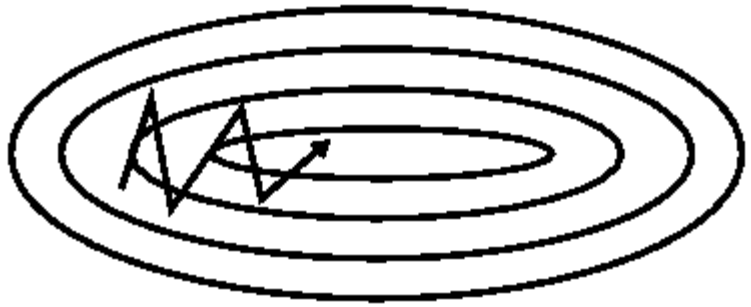


Image 3: SGD with momentum

- Momentum is a method that **helps accelerate SGD in the relevant direction and dampens oscillations** as can be seen in Image 3.

http://ruder.io/optimizing-gradient-descent/

# Momentum – update equation

- Momentum dampens oscillations by adding a fraction of the update vector of the past time step to the current update vector:

$$v_0 = 0$$
$$v_t = \gamma v_{t-1} + \nabla J(w)$$
$$w = w - \alpha v_t$$

The momentum term $\gamma$ is usually set to 0.9 or a similar value.

Image 2: SGD without momentum

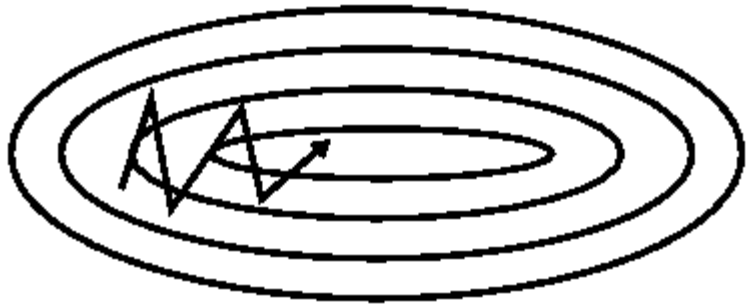


Image 3: SGD with momentum

- The momentum algorithm accumulates an exponentially decaying moving average of past gradients and continues to move in their direction.

# Momentum – intuition

- You can think of momentum as **pushing a ball down a hill**. The ball accumulates momentum as it rolls downhill, becoming faster and faster on the way (until it reaches its terminal velocity if there is air resistance, i.e. $\gamma < 1$). The same thing happens to our parameter updates: The momentum term increases for dimensions whose gradients point in the same directions and reduces updates for dimensions whose gradients change directions. As a result, we gain faster convergence and reduced oscillation.

Image 2: SGD without momentum

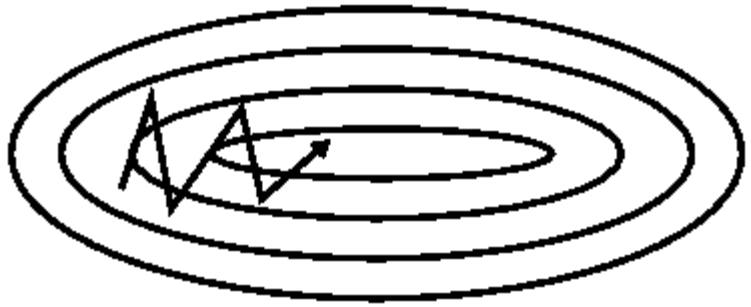
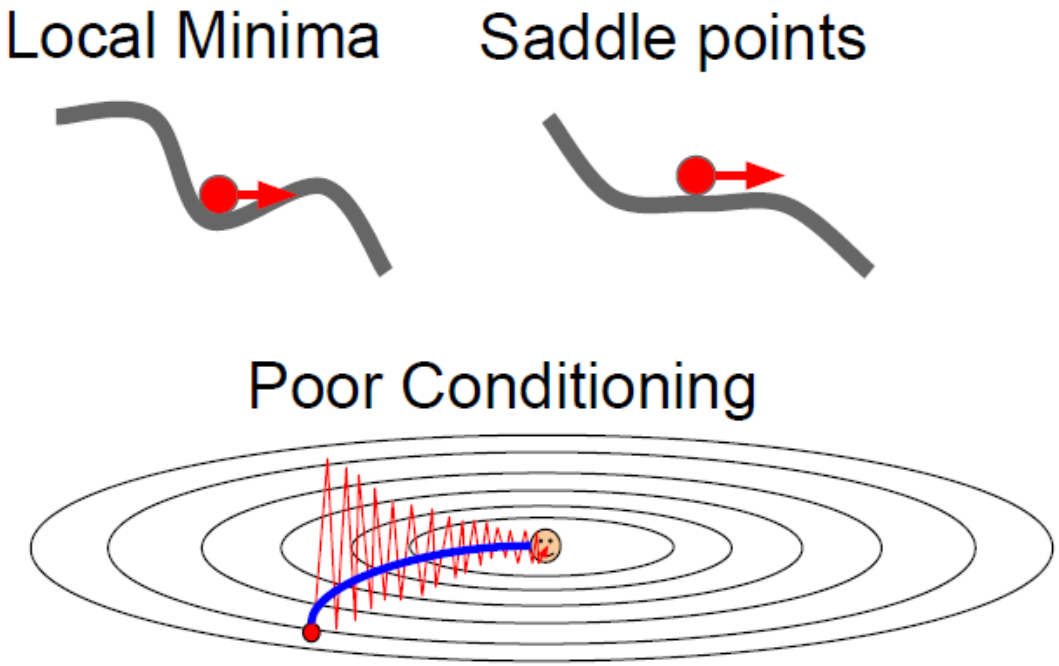


Image 3: SGD with momentum

http://ruder.io/optimizing-gradient-descent/

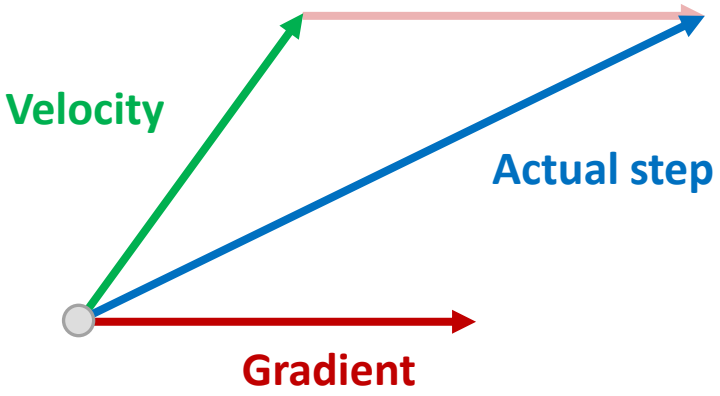
# Momentum

- The velocity/momentum that we have built up also helps us escape local minima and saddle points (think of rolling ball).
- Also, the zig-zagging associated with SGD (formally explained by poor conditioning of the Hessian matrix) is dampened.
- The explanation of the latter is that the oscillations in the sensitive direction cancel each other out over time, because the momentum algorithm accumulates an exponentially decaying moving average of past gradients.



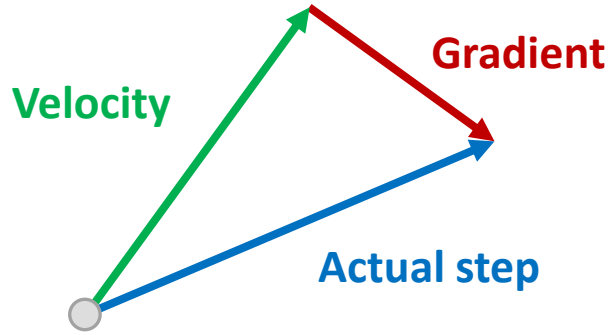
# Nestorov momentum

**Momentum update:**




$$v_0 = 0$$
$$v_t = \gamma v_{t-1} + \nabla J(w)$$
$$w = w - \alpha v_t$$

**Nestorov momentum:**
(look ahead)

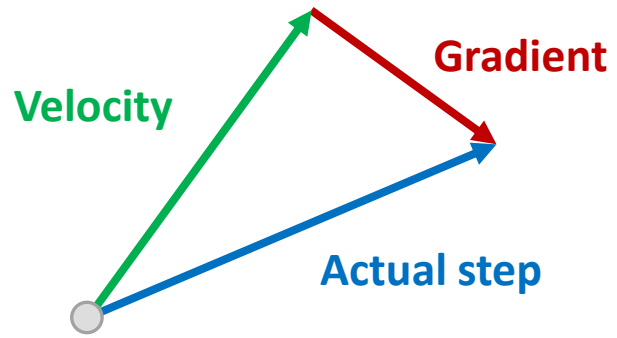



$$v_0 = 0$$
$$v_t = \gamma v_{t-1} + \nabla J(w + \gamma v_{t-1})$$
$$w = w - \alpha v_t$$

# Nestorov momentum

**Intuition:** A ball that rolls down a hill, blindly following the slope, is highly unsatisfactory. We'd like to have **a smarter ball**, a ball that has a notion of where it is going so that it **knows to slow down before the hill slopes up again**. Nestorov momentum does this by looking ahead in time. In practise, it usually works slightly better than conventional momentum.

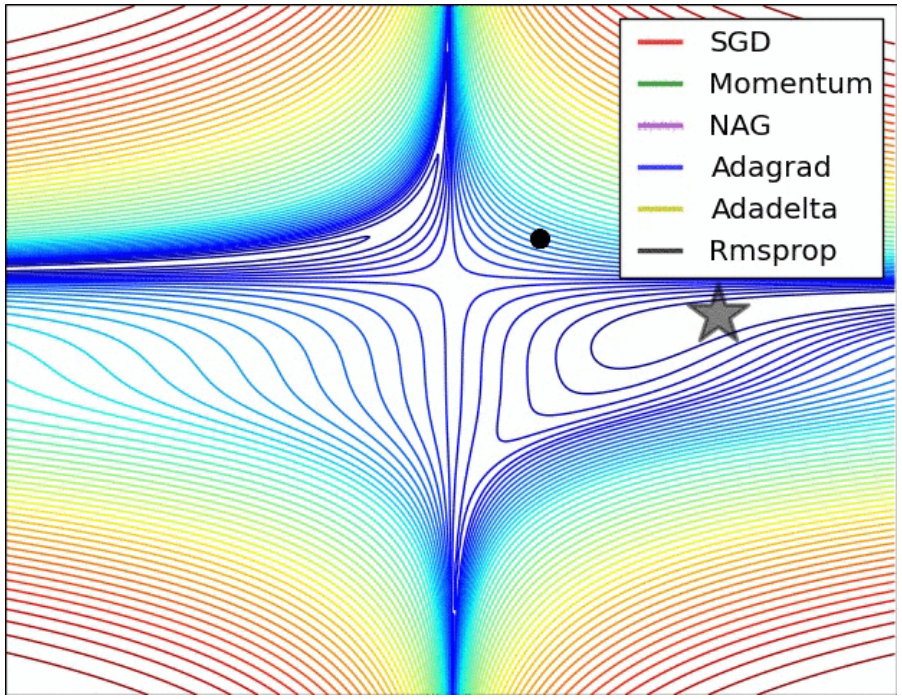
**Nestorov momentum:**
(look ahead)



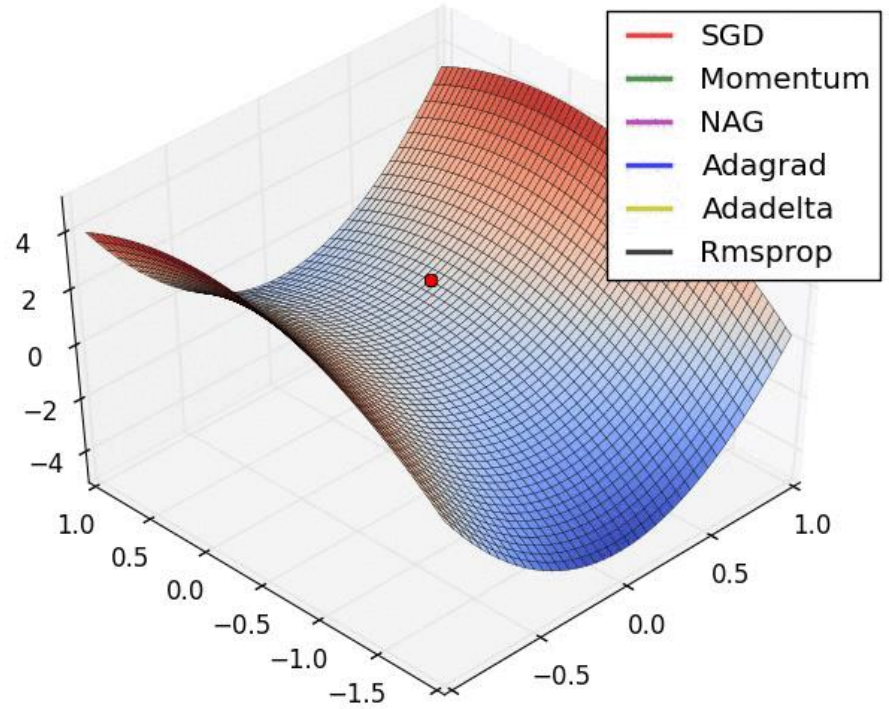

$$v_0 = 0$$
$$v_t = \gamma v_{t-1} + \nabla J(w + \gamma v_{t-1})$$
$$w = w - \alpha v_t$$

# Nestorov momentum

Look for NAG: Nestorov Accelerated Gradient




Link to GIF animation:
http://ruder.io/content/images/2016/09/contours_evaluation_optimizers.gif




Link to GIF animation:
http://ruder.io/content/images/2016/09/saddle_point_evaluation_optimizers.gif

# In practise

- Always use momentum with SGD.
- Setting the momentum term ($\gamma$) between 0.9 and 0.99 usually works.
- Start with high learning rate and decay over time.

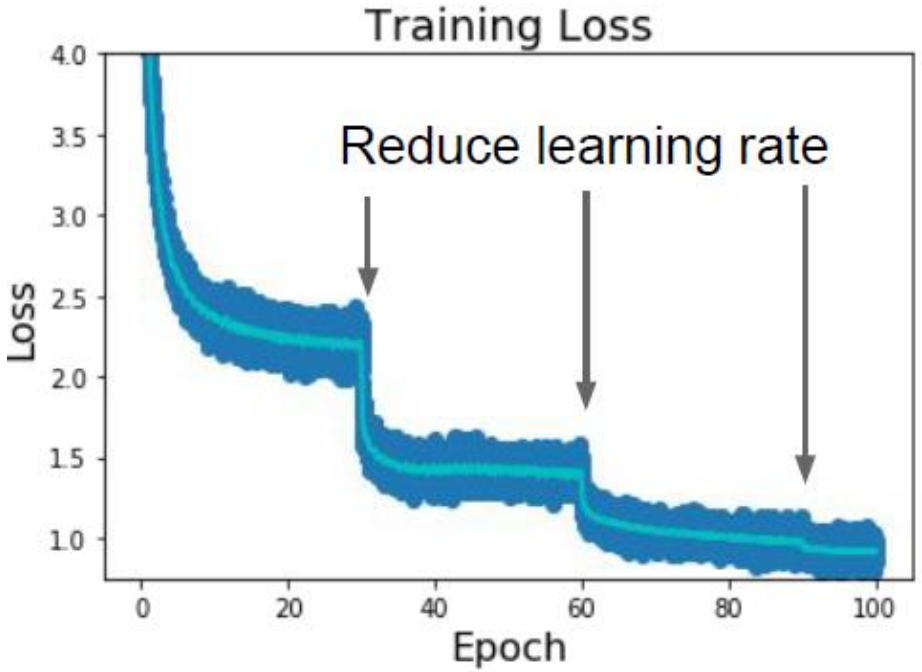
# Learning rate decay and cycling

# Learning rate decay

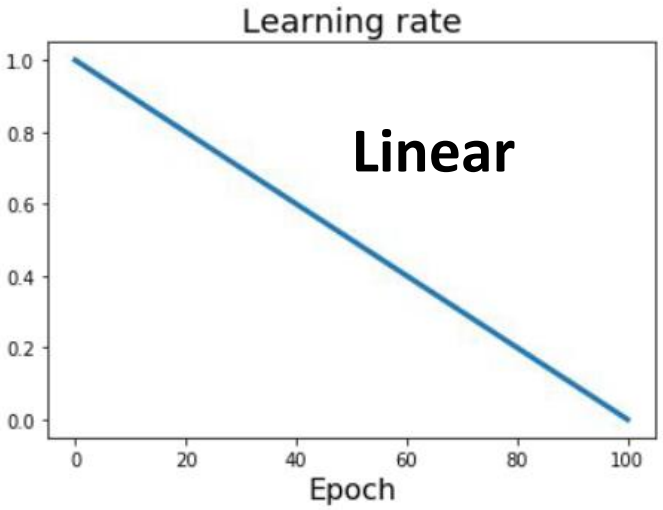
- In training deep networks, it is usually helpful to decay the learning rate over time.
- **Intuition**
  1. Start with high learning rate to **explore loss landscape**.
  2. Then slowly decay learning rate to **tune in on a local minimum**.
- Knowing when and how much to decay the learning rate **can be tricky**:
  - Decay it slowly and you'll be wasting computation bouncing around chaotically with little improvement for a long time.
  - But decay it too aggressively and the system will cool too quickly, unable to reach the best position it can.

# Step decay

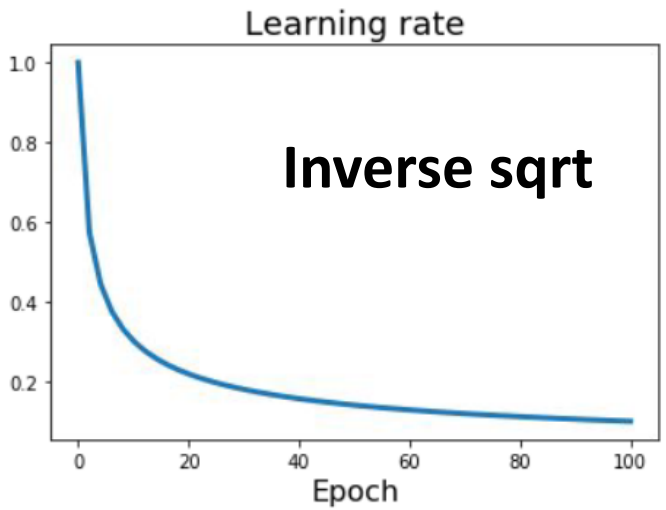
- Reduce the learning rate by some factor every few epochs.
- Typical values might be reducing the learning rate by a half every 5 epochs, or by 0.1 every 20 epochs.
- These numbers depend heavily on the type of problem and the model.
- One heuristic you may see in practice is to watch the validation error while training with a fixed learning rate, and reduce the learning rate by a constant (e.g. 0.5) whenever the validation error stops improving.



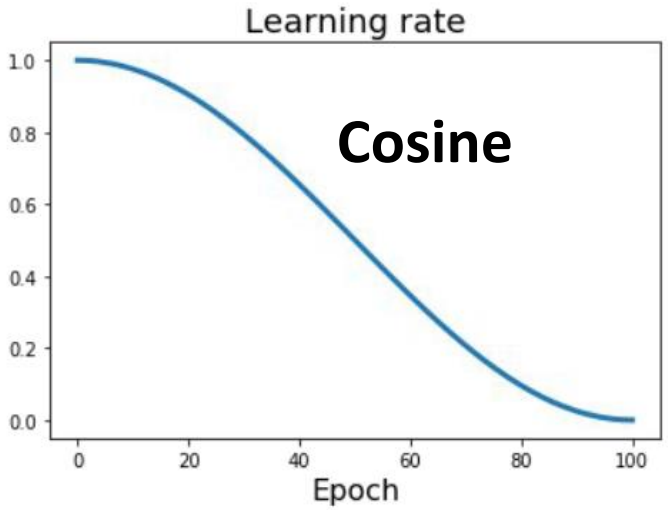
# Other variants of learning rate decay




$\alpha = \alpha_0(1 - t/T)$



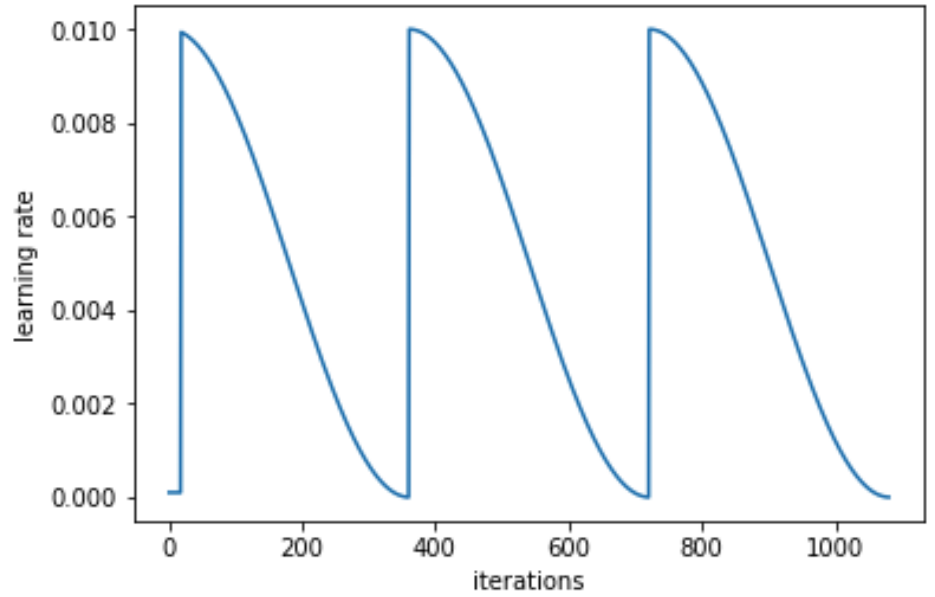

$\alpha = \alpha_0/\sqrt{t}$




$\alpha = \frac{1}{2}\alpha_0(1 + \cos(t\pi/T))$

# Cyclical Learning Rate (CLR)

- The learning rate is the most important hyperparameter to tune for training deep neural networks.
- CLR is a method for setting the learning rate, which practically eliminates the need to experimentally find the best values and schedule for the global learning rates.
- Instead of monotonically decreasing the learning rate, **this method lets the learning rate cyclically vary between reasonable boundary values**.
- Training with cyclical learning rates instead of fixed values **achieves improved classification accuracy** without a need to tune and often in fewer iterations.

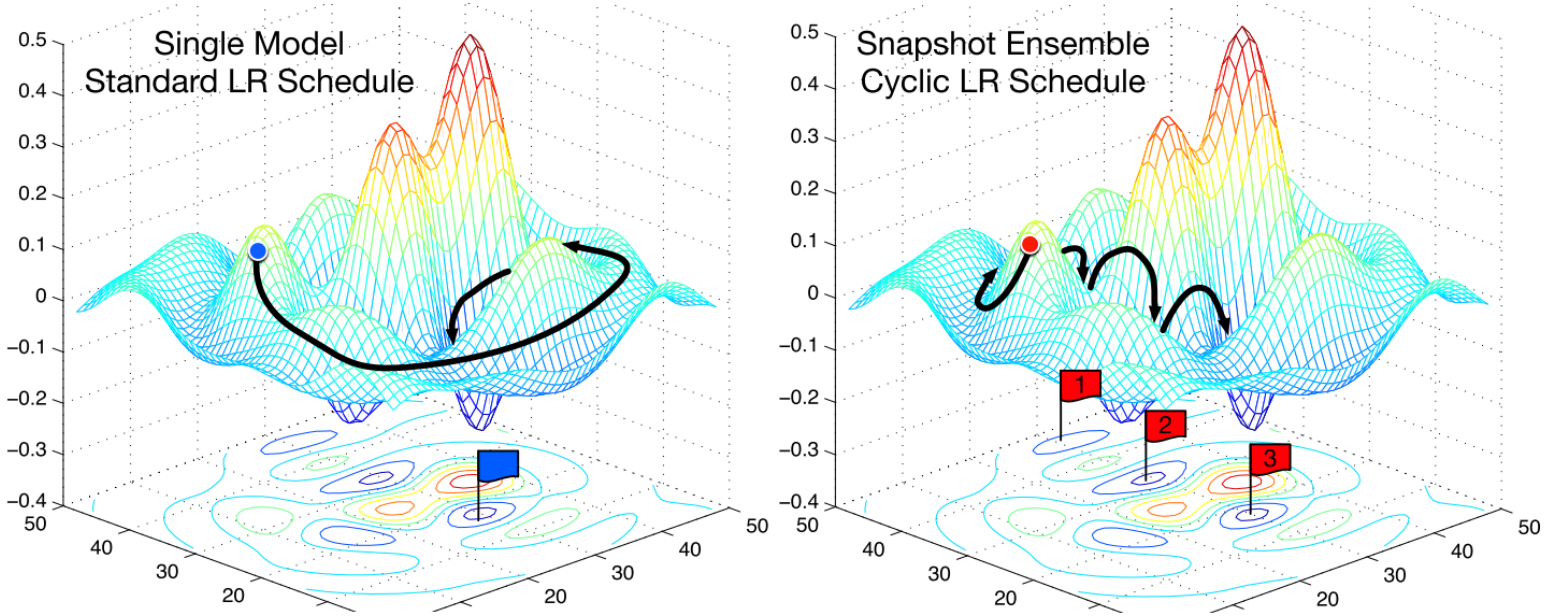



Smith (2015): Cyclical Learning Rates for Training Neural Networks

https://medium.com/@lipeng2/cyclical-learning-rates-for-training-neural-networks-4de755927d46

# Cyclical Learning Rate (CLR)

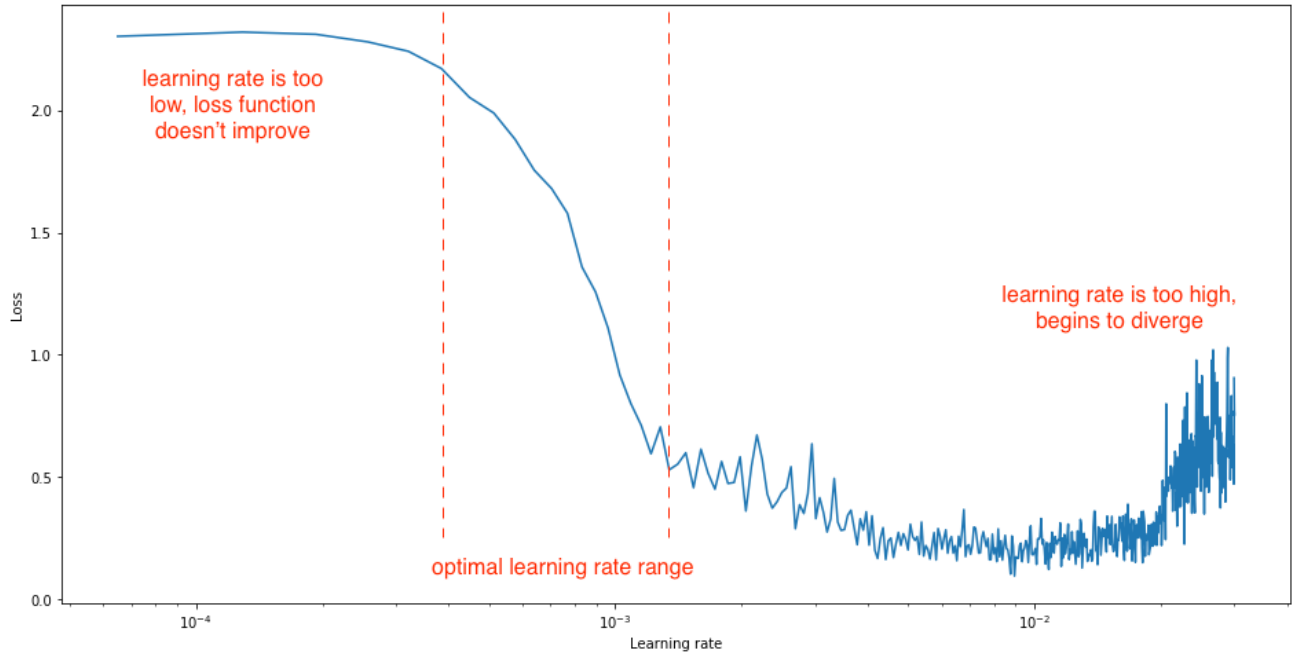
- The rationale is that increasing the learning rate **will force the model to jump to a different part of the weight space if the current area is "spikey".**
- In other words, it will force to find another local minimum if the current minimum is not robust, and make the model generalize better to unseen data. Below is an illustration of CLR schedule with three resets.




https://medium.com/@lipeng2/cyclical-learning-rates-for-training-neural-networks-4de755927d46

# Cyclical Learning Rate (CLR)

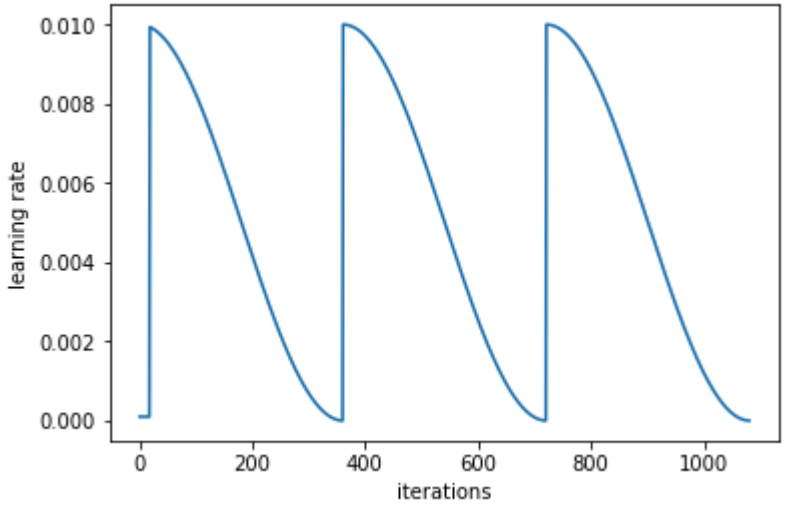
- **Step 1: Learning rate range test**
- Start training and increase learning rate linearly after each batch, and calculate the loss.
- Then display loss as a function of learning rate.
- By manual inspection, select a range of suitable learning rates (look for strongest downward slope).




See Jeremy Howard's guide: https://youtu.be/Egp4Zajhzog?t=1674

# Cyclical Learning Rate (CLR)

- **Step 2: Actual training**
- Train our model where the learning rate cyclically varies between the bounds that were found in step 1.
- You can combine with learning rate decay.



# CLR in practise

- Fast.ai is the name of an online course *and* a deep learning framework built on top of PyTorch.
- Jeremy Howard from fast.ai uses CLR a lot.
- Jeremy's approach (see next slide):
  1. Download a pretrained convolutional encoder and add your own decoder (classifier)
  2. Train only the decoder for a few cycles (with cyclic learning rate). This is called transfer learning, and the purpose is just to get the weights of the decoder "approximately right".
  3. "Unfreeze" the encoder and run "learning rate range test" (see step 1 on previous slide)
  4. Plot loss vs. learning rate and pick suitable learning rate range
  5. Fine-tune your model for a couple of cycles (train all layers, including convolutional base).
- See demo here (Jupyter notebook).
- See brief explanation here (Youtube video – watch about 3 minutes).

# CLR in practise

- Jeremy's approach:
  1. Download a pretrained convolutional encoder and add your own decoder (classifier)
  2. Train only the decoder for a few cycles (with cyclic learning rate). This is *transfer learning*.
  3. "Unfreeze" the encoder and run learning rate finder (see step 1 on previous slide)
  4. Plot loss vs. learning rate and pick suitable learning rate range
  5. Train your model for a another few cycles (train all layers, including convolutional base). This is called *fine-tuning*.

```
learn = create_cnn(data, models.resnet34, metrics=error_rate)
```

Then run `fit_one_cycle` 4 times and see how we go. And we have a 2% error rate. So that's pretty good. Sometimes it's easy for me to recognize a black bear from a grizzly bear, but sometimes it's a bit tricky. This one seems to be doing pretty well.

```
learn.fit_one_cycle(4)
```

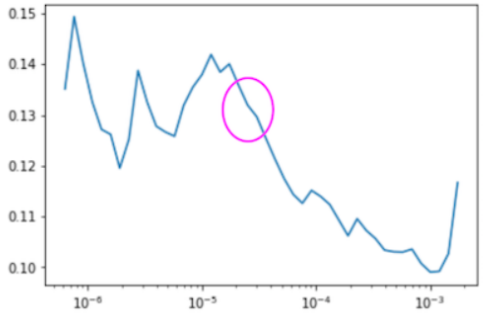
```
Total time: 00:54
epoch  train_loss  valid_loss  error_rate
1      0.710584    0.087024    0.021277    (00:14)
2      0.414239    0.045413    0.014184    (00:13)
3      0.306174    0.035602    0.014184    (00:13)
4      0.239355    0.035230    0.021277    (00:13)
```

```
learn.unfreeze()
```

Then we run the learning rate finder and plot it (it tells you exactly what to type). And we take a look.

```
learn.lr_find()
```

```
learn.recorder.plot()
```

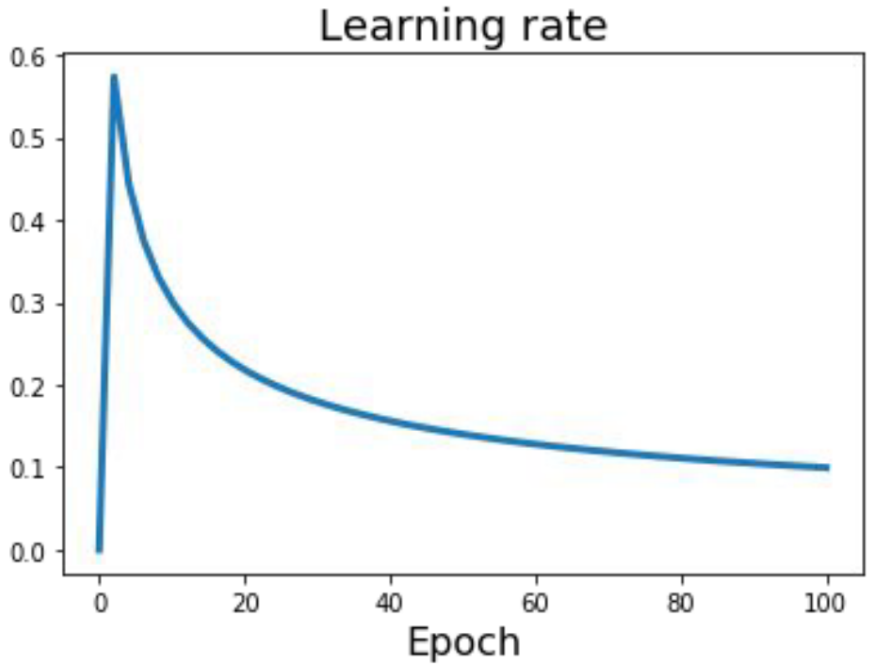



```
learn.fit_one_cycle(2, max_lr=slice(3e-5,3e-4))
```

```
Total time: 00:28
epoch  train_loss  valid_loss  error_rate
1      0.107059    0.056375    0.028369    (00:14)
2      0.070725    0.041957    0.014184    (00:13)
```

# Linear warmup

- In some cases, deep neural networks can suffer from a sort of "early over-fitting".
- This can happen if your data is skewed or includes a cluster of related, strongly-featured observations.
- What happens then is that your model's initial training can skew badly toward those features – or worse, toward incidental features that aren't truly related to the topic at all.
- Warm-up is a way to reduce the primacy effect of the early training examples.
- Without it, you may need to run a few extra epochs to get the convergence desired, as the model un-trains those early superstitions.




Goyal et al (2017): Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour

# Per-parameter adaptive learning rate methods

# Motivation

- The approaches we've discussed so far **manipulate the learning rate globally and equally for all parameters**.
- Tuning the learning rates is an expensive process, so much work has gone into devising methods that can **adaptively tune the learning rates, and even do so per parameter**.
- Many of these methods may still require other hyperparameter settings, but the argument is that they are well-behaved for a broader range of hyperparameter values than the raw learning rate.
- In this section we highlight some common adaptive methods you may encounter in practice:
  - Adagrad
  - RMSProp
  - Adam

# Adagrad

- Adagrad adapts the learning rate to the parameters, performing **smaller updates** (i.e. low learning rates) for parameters associated with frequently occurring features, and **larger updates** (i.e. high learning rates) for parameters associated with infrequent features.
- Adagrad has been shown to improve the robustness of SGD.
- Basic idea:
  - Add element-wise scaling of the gradient based on the historical sum of squares in each dimension
  - "Per-parameter learning rates" or "adaptive learning rates"

$$G_{0,i} = 0$$
$$g_{t,i} = \nabla J(w_{t,i})$$
$$G_{t,i} = G_{t-1,i} + g_{t,i}^2$$
$$w_{t+1,i} = w_{t,i} - \frac{\alpha}{\sqrt{G_{t,i}+\varepsilon}} g_{t,i}$$

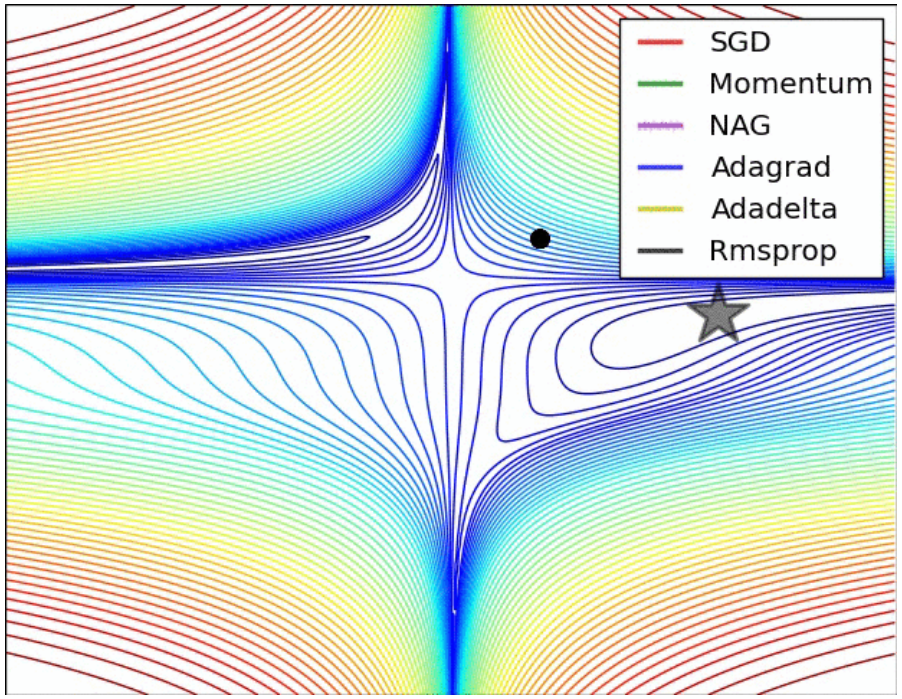
$g_{t,i}$ is the partial derivative of *i*'th parameter at time *t.*

$G_{t,i}$ is the sum of the squared partial derivative of *i*'th up to time *t.*

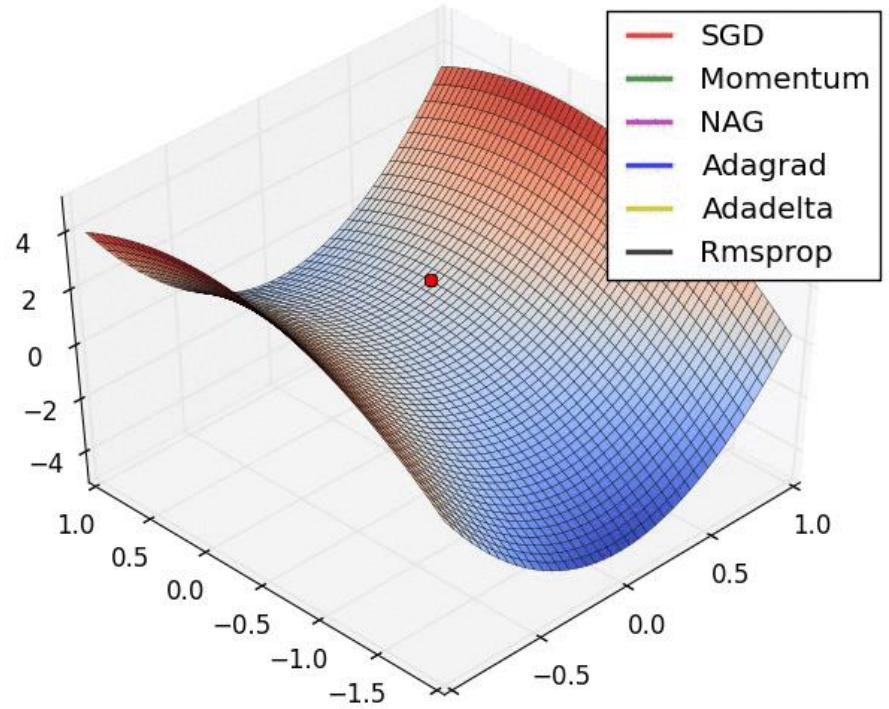
$\varepsilon$ is a smoothing term to avoid division by 0 (usually $10^{-8}$ to $10^{-7}$).

# Adagrad

Progress along “steep” directions is damped, progress along “flat” directions is accelerated




Link to GIF animation:
http://ruder.io/content/images/2016/09/contours_evaluation_optimizers.gif




Link to GIF animation:
http://ruder.io/content/images/2016/09/saddle_point_evaluation_optimizers.gif

# RMSProp

- Adagrad's main weakness is its accumulation of the squared gradients in the denominator.
- Since every added term is positive, **the accumulated sum keeps growing during training**.
- This in turn **causes the learning rate to shrink and eventually become infinitesimally small**, at which point the algorithm is no longer able to acquire additional knowledge.
- Both RMSProp and Adadelta (not covered here) aim to resolve this flaw.

Adagrad:

$$G_{0,i} = 0$$
$$g_{t,i} = \nabla J(w_{t,i})$$
$$G_{t,i} = G_{t-1,i} + g_{t,i}^2$$
$$w_{t+1,i} = w_{t,i} - \frac{\alpha}{\sqrt{G_{t,i}+\varepsilon}} g_{t,i}$$

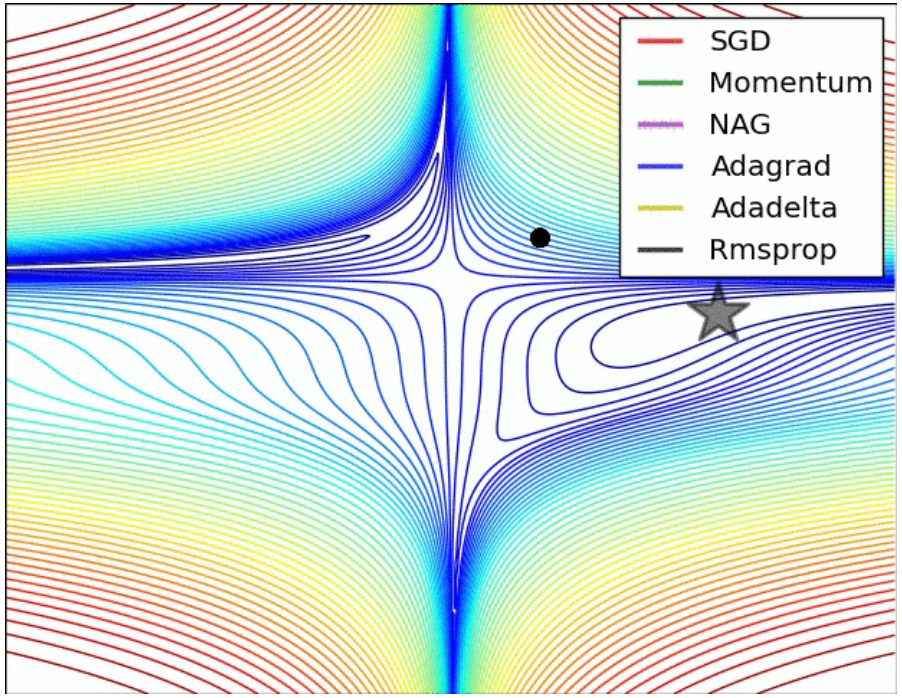
RMSprop:

$$G_{0,i} = 0$$
$$g_{t,i} = \nabla J(w_{t,i})$$
$$G_{t,i} = \gamma G_{t-1,i} + (1-\gamma) g_{t,i}^2$$
$$w_{t+1,i} = w_{t,i} - \frac{\alpha}{\sqrt{G_{t,i}+\varepsilon}} g_{t,i}$$

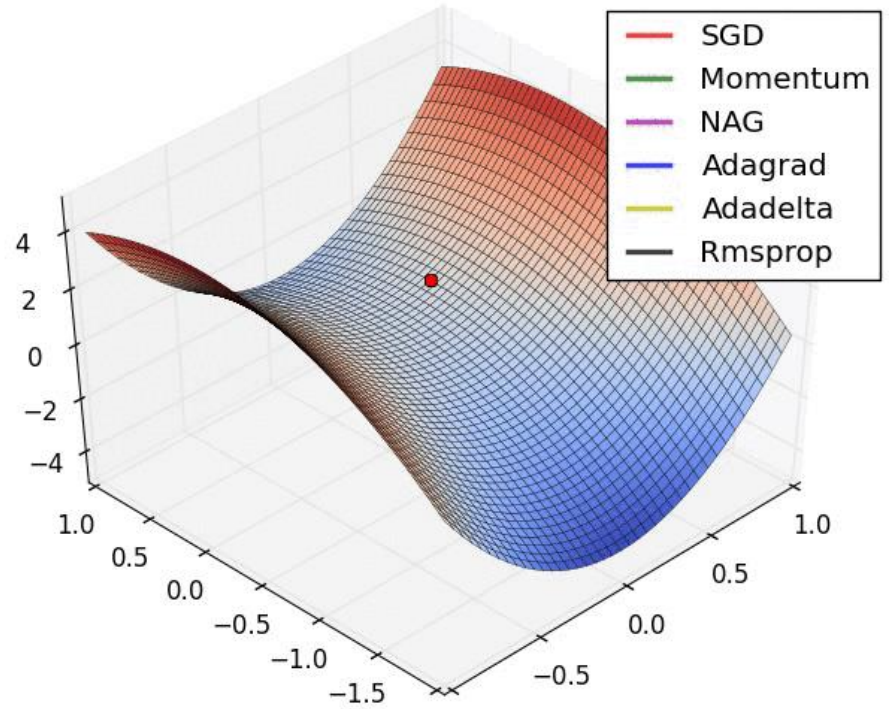
$\gamma$ is a decay rate set to around 0.9

# RMSProp

Slightly faster convergence compared to Adagrad




Link to GIF animation:
http://ruder.io/content/images/2016/09/contours_evaluation_optimizers.gif




Link to GIF animation:
http://ruder.io/content/images/2016/09/saddle_point_evaluation_optimizers.gif

# Adam

- Adaptive Moment Estimation (Adam) is another method that computes adaptive learning rates for each parameter.
- **Sort of like RMSProp with momentum:** In addition to storing an exponentially decaying average of past squared gradients, like RMSprop, Adam also keeps an exponentially decaying average of past gradients, similar to momentum.
- **Intuition:** Whereas momentum can be seen as a ball running down a slope, Adam behaves like a heavy ball with friction, which thus prefers flat minima in the error surface.

Almost Adam:

$$m_{0,i}, v_{0,i} = 0$$
$$m_{t,i} = \beta_1 m_{t-1,i} + (1-\beta_1) g_{t,i}$$
$$v_{t,i} = \beta_2 v_{t-1,i} + (1-\beta_2) g_{t,i}^2$$
$$w_{t+1,i} = w_{t,i} - \frac{\alpha}{\sqrt{v_{t,i}+\varepsilon}} m_{t,i}$$

Adam: Add bias correction

$$\hat{m}_{t,i} = \frac{m_{t,i}}{1-\beta_1^t} \text{ and } \hat{v}_{t,i} = \frac{v_{t,i}}{1-\beta_2^t}$$
$$w_{t+1,i} = w_{t,i} - \frac{\alpha}{\sqrt{\hat{v}_{t,i}+\varepsilon}} \hat{m}_{t,i}$$

Kinga & Ba (2014): Adam: A Method for Stochastic Optimization

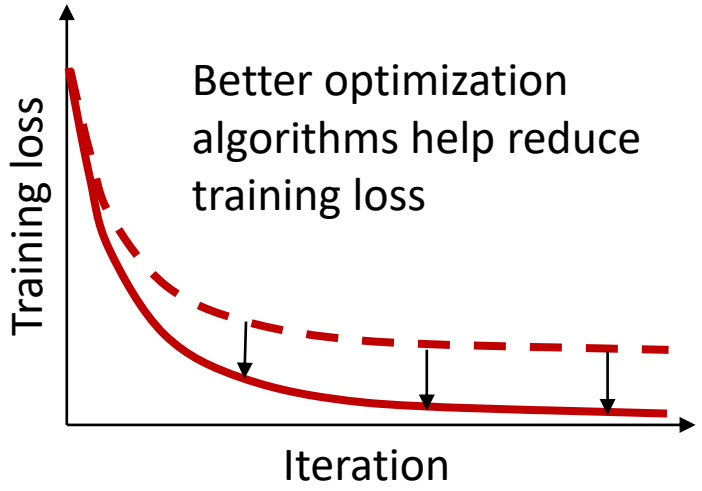
# In practise

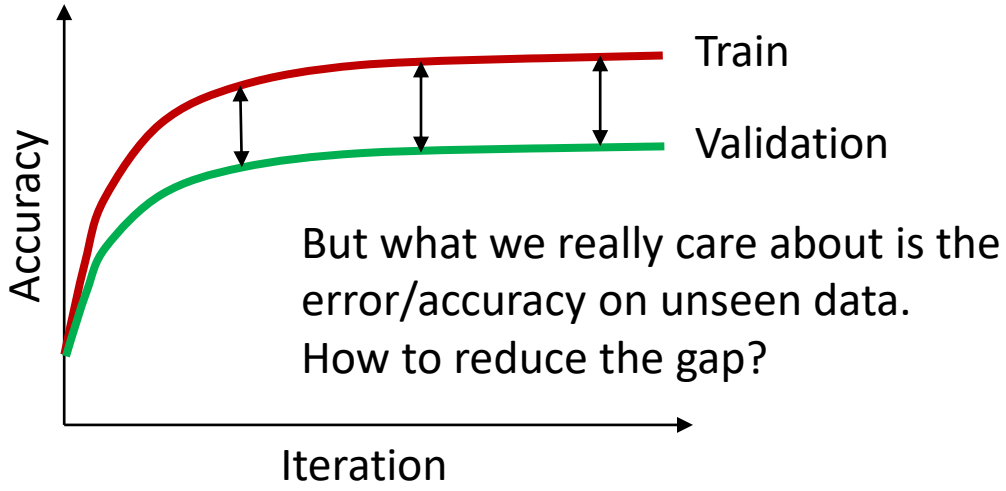
- **Adam** is a good default choice in many cases; it often works ok even with constant learning rate
- **SGD + Momentum** can outperform Adam but may require more tuning of learning rate and learning rate schedule

# Regularization

# What is regularization?

- A central problem in machine learning is how to make an algorithm that will perform well not just on the training data, but also on new inputs (**unseen data**).
- Many strategies used in machine learning are **explicitly designed to reduce the test/validation error**, possibly at the expense of increased training error. These strategies are known collectively as **regularization**.
- The overall purpose is to **reduce overfitting**.



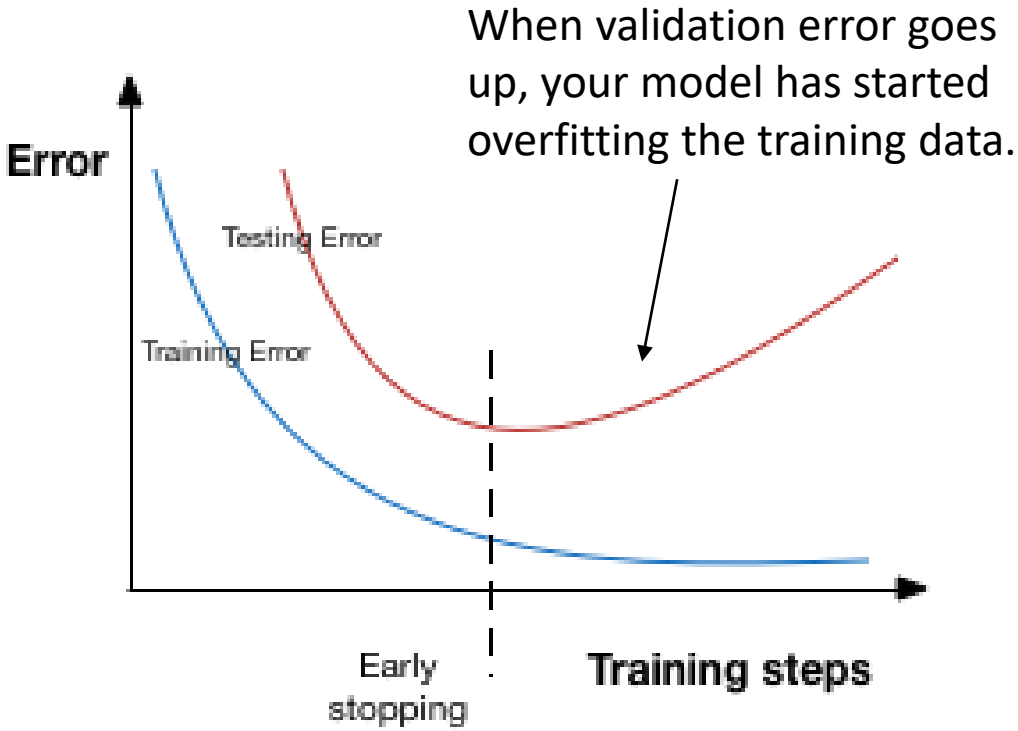



# Overview

- Early stopping
- Model ensembles
- Dropout
- Data augmentation
- Noise robustness (Cutout, Mixup and Label smoothing)
- Multi-task learning
- Regularization terms

# Early stopping

- Always do this!
- In early stopping, we keep one part of the training set as the validation set.
- **When performance on the validation set is getting worse, we immediately stop the training on the model.**
- Alternatively, train for a long time, but always keep track of the **model snapshot** that worked best on the validation set.
- Early stopping can be automated (e.g., using callback in Keras), but in practise its often easier to just inspect the loss curves manually.



# Model ensembles

- **Basic idea:** Train multiple (possibly independent) models, and at test time **average their predictions**.
- In practice, this gives a few percent extra performance (see for instance AlexNet paper).
- There are a few approaches to forming an ensemble:
  - Same model, different initializations.
  - Top models discovered using, say, cross-validation.
  - Different checkpoints of a single model (see for instance "snapshot ensemble" paper referenced below).
  - Running average of parameters during training.
- Snapshot ensembles: Train 1, get M for free

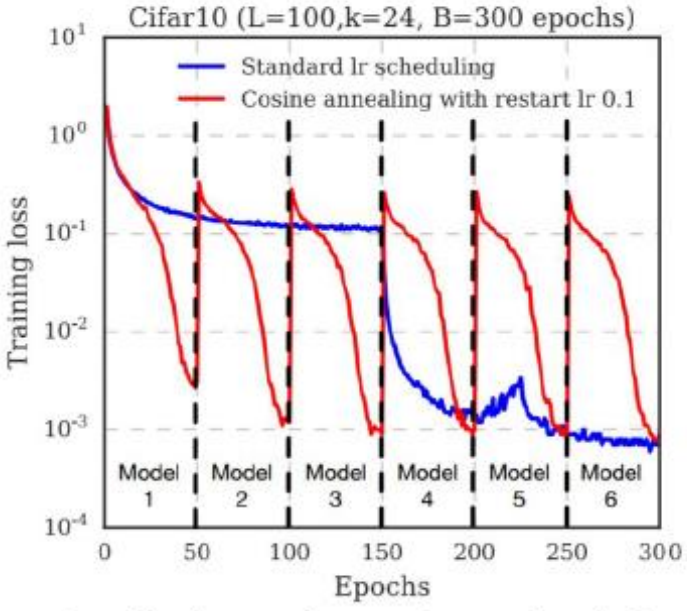
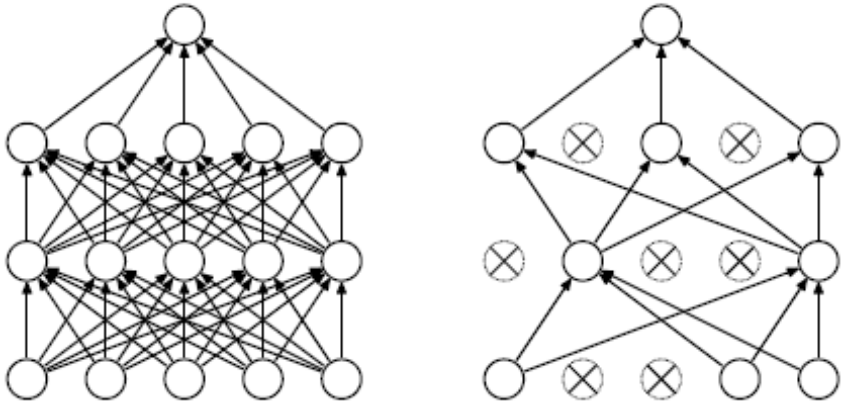



Figure 2 (Snapshot ensemble paper): Training loss of 100-layer DenseNet on CIFAR10 using standard learning rate (blue) and M = 6 cosine annealing cycles (red). The intermediate models, denoted by the dotted lines, form an ensemble at the end of training

# Dropout

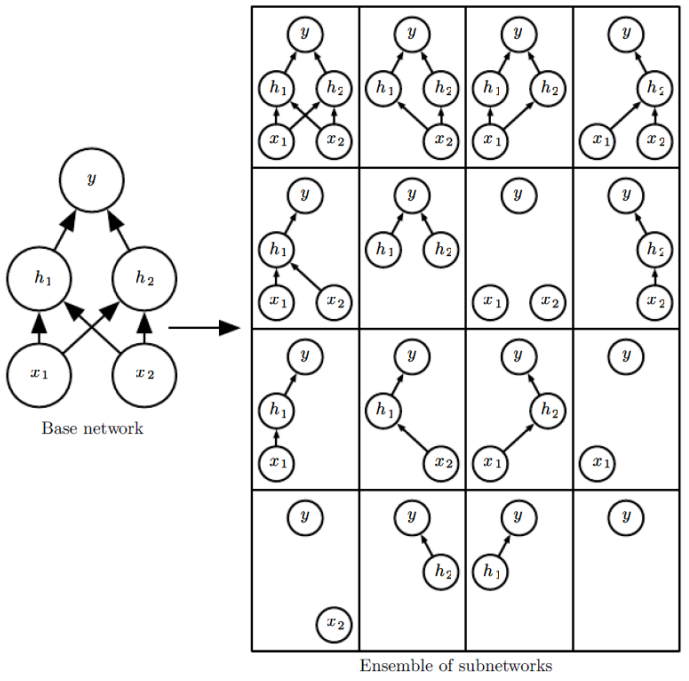
- Dropout provides a computationally inexpensive but powerful method of regularization.
- Dropout can be thought of as turning your neural network into a **giant ensemble of smaller models** during training, and combining all these models into one model at test time.
- **Basic idea:** In each forward pass, randomly set some neurons to zero.
- Probability $p$ of dropping is a hyperparameter ($p = 0.5$ is common)



(a) Standard Neural Net (b) After applying dropout.

Figure 1: Dropout Neural Net Model. **Left**: A standard neural net with 2 hidden layers. **Right**: An example of a thinned net produced by applying dropout to the network on the left. Crossed units have been dropped.

Figure 7.6 (book)



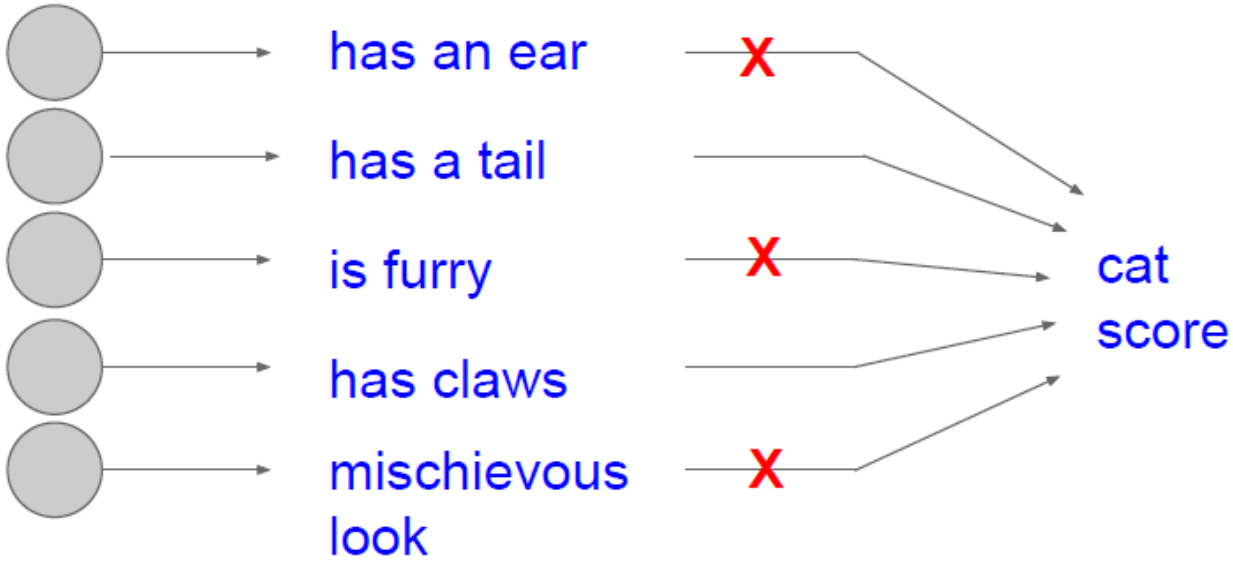

Srivastava et al. (2014): Dropout: A Simple Way to Prevent Neural Networks from Overfitting

# Dropout

- How can this possibly be a good idea?
- Another interpretation: Dropout prevents neurons from co-adapting too much. In other words, it forces the network to learn a redundant representation, which eventually makes the model better at generalizing (by reducing overfitting).

With dropout the network cannot rely on any single neuron to measure the “cat-ness”.

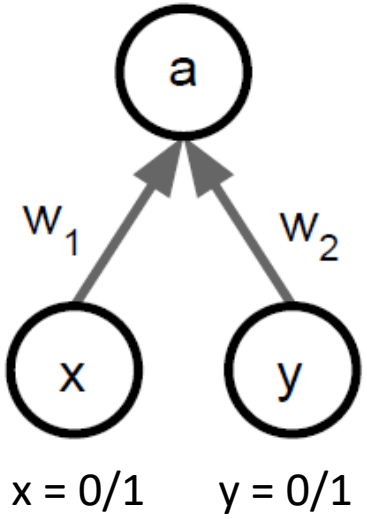
Instead it is forced to distribute its idea of cat-ness across many different neurons.




https://towardsdatascience.com/simplified-math-behind-dropout-in-deep-learning-6d50f3f47275

# Dropout at test time

- Dropout makes our output random!
- Want to “average out” the randomness at test-time.
- At test time all neurons are active always => We must scale the activations so that for each neuron: output at test time = expected output at training time
- **Solution:** At test time, perform **correction** by multiplying output by dropout probability (*p*).




**During training ($p = \frac{1}{2}$):**

$$\mathrm{E}[a] = \frac{1}{4}(w_1 x + w_2 y) + \frac{1}{4}(w_1 x + 0y) + \frac{1}{4}(0x + w_2 y) + \frac{1}{4}(0x + 0y) = \frac{1}{2}(w_1 x + w_2 y)$$

**At test time (without correction):**

$$\mathrm{E}[a] = w_1 x + w_2 y$$

**At test time (with correction):**

$$\mathrm{E}[a] = p(w_1 x + w_2 y) = \frac{1}{2}(w_1 x + w_2 y)$$

**Same expectation during training and at test time**

# Data augmentation

- The best way to make a machine learning model generalize better is to **train it on more data**.
- In practice, the amount of data we have is limited.
- One way to get around this problem is to **create fake data** and add it to the training set.
- For most visual recognition tasks, it is reasonably straightforward to create new fake data.
- **Examples:**
  - Keras ImageDataGenerator class (https://keras.io/preprocessing/image/)
  - Albumentations (https://github.com/albumentations-team/albumentations)

# Data augmentation – types of variation

Viewpoint variation

Scale variation

Deformation

Occlusion

Illumination conditions

Background clutter

Intra-class variation

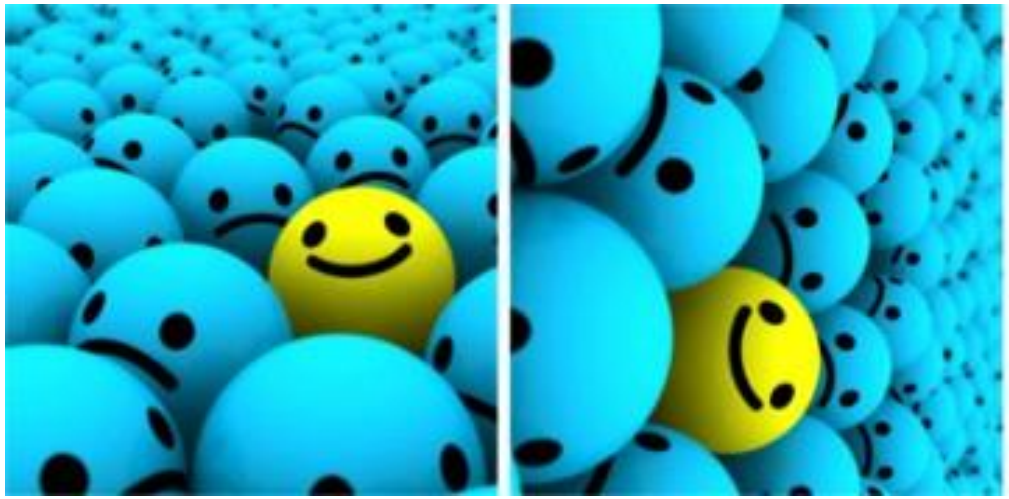
# Data augmentation - examples

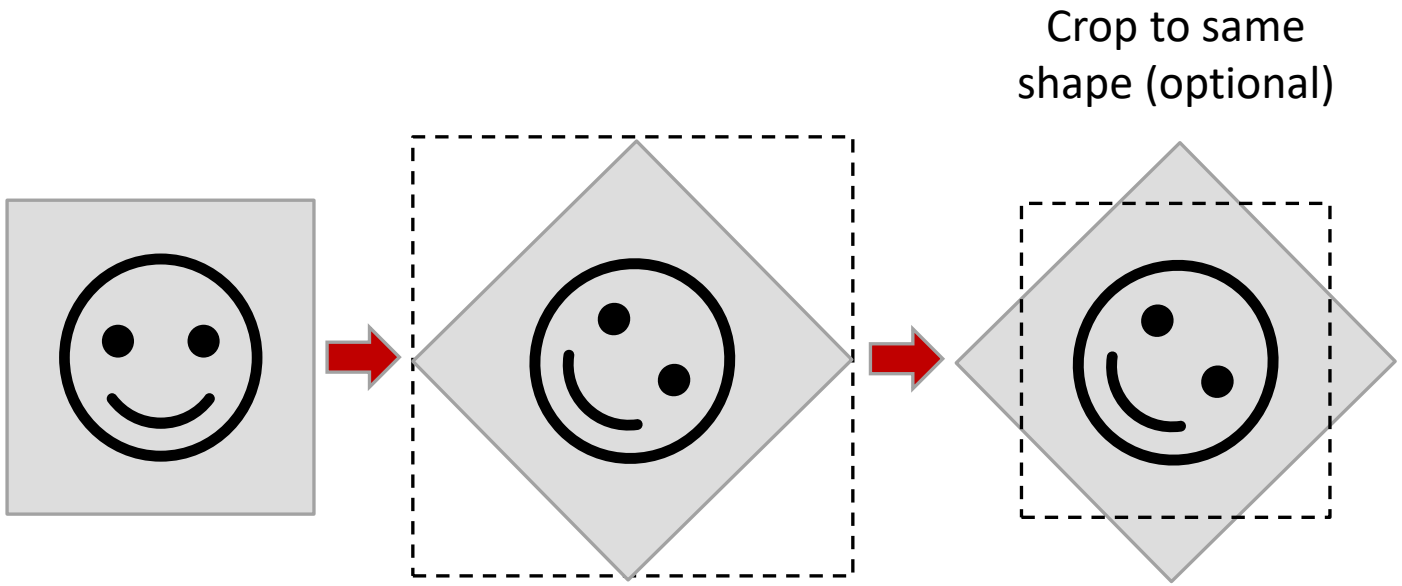
- **Flip -** flip images horizontally and vertically.
- Be careful only to flip images, if the original class properties are preserved (so not MNIST).

https://nanonets.com/blog/data-augmentation-how-to-use-deep-learning-when-you-have-limited-data-part-2/

# Data augmentation - examples

- **Rotation**
- Note that image dimensions may not be preserved after rotation.






https://nanonets.com/blog/data-augmentation-how-to-use-deep-learning-when-you-have-limited-data-part-2/

# Data augmentation - examples

- **Scaling**
- The image can be scaled outward or inward.

https://nanonets.com/blog/data-augmentation-how-to-use-deep-learning-when-you-have-limited-data-part-2/

# Data augmentation - examples

- **Crop -** randomly sample a section from the original image and resize this section to the original image size.

https://nanonets.com/blog/data-augmentation-how-to-use-deep-learning-when-you-have-limited-data-part-2/

# Data augmentation - examples

- **Translation -** just involves moving the image along the X or Y direction (or both).
- Very useful as most objects can be located at almost anywhere in the image.

https://nanonets.com/blog/data-augmentation-how-to-use-deep-learning-when-you-have-limited-data-part-2/

# Data augmentation - examples

- **Gaussian noise –** adding just the right amount of noise can enhance the learning capability.
- Overfitting usually happens when your neural network tries to learn **high frequency features** (patterns that occur a lot) that may not be useful. Gaussian noise, which has zero mean, essentially has data points in all frequencies, effectively distorting the high frequency features.

https://nanonets.com/blog/data-augmentation-how-to-use-deep-learning-when-you-have-limited-data-part-2/

# Data augmentation – filling empty gaps

- **Filling in the empty gaps** - after we perform spatial transformations, we need to preserve our original image size. Since our image does not have any information about things outside it's boundary, we need to make some assumptions.
- Deep learning frameworks have some standard ways with which you can decide on how to fill the unknown space. They are defined as follows:

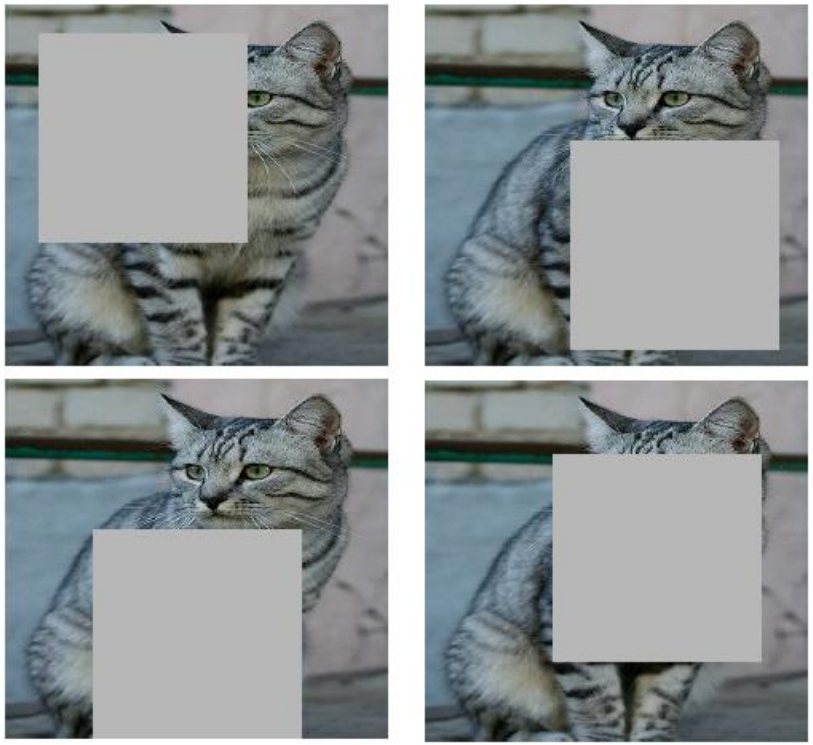
Constant value | Extent edge value | Reflect values | Symmetric | Wrap

# Noise robustness

- A common pattern in regularization is:
  - **Training:** Add some kind of randomness
  - **Test:** Marginalize over the noise (means to average out the noise…)
- Examples: Dropout and data augmentation
- Other examples
  - Cutout
  - Mixup
  - Label smoothing

# Noise robustness

- Cutout:
  - **Training:** Set random image region to zero (or constant value)
  - **Test:** Use full image
- Teaches network to become **invariant to occlusion.**

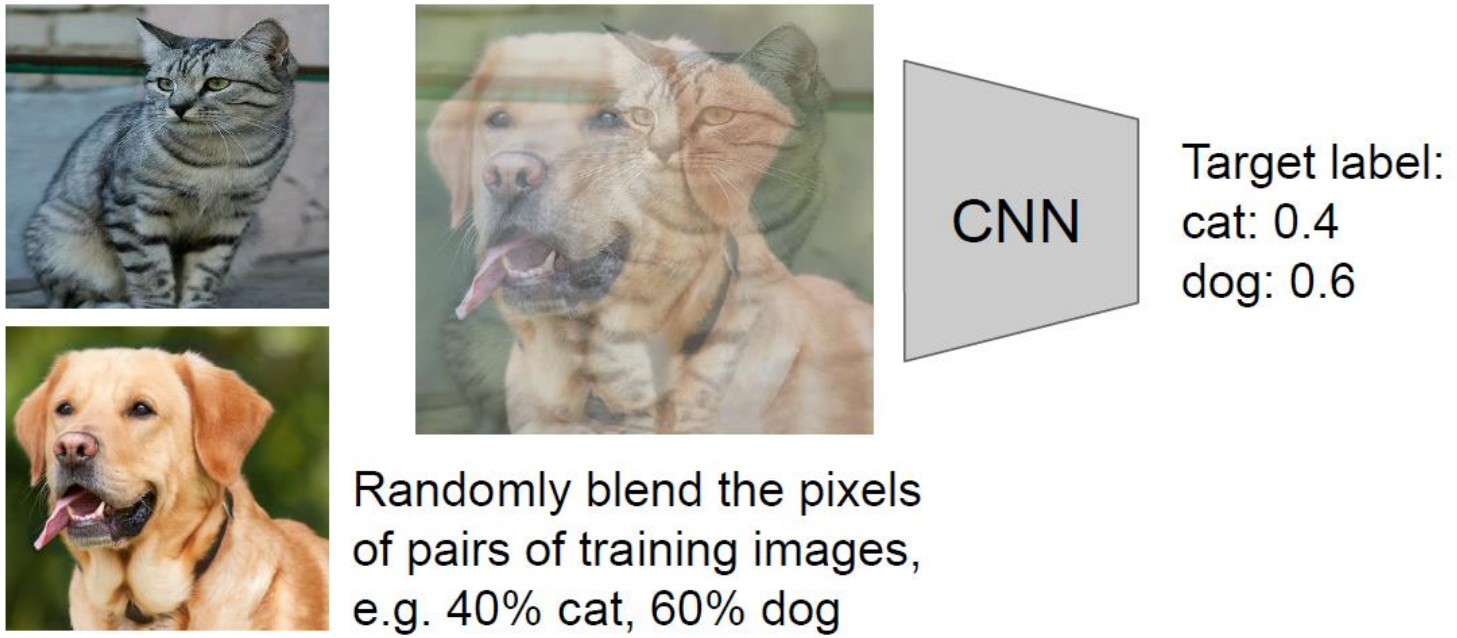


Works very well for small datasets like CIFAR, less common for large datasets like ImageNet

https://arxiv.org/abs/1708.04552

# Noise robustness

- Mixup:
  - **Training:** Train on random blends of images
  - **Test:** Use original image
- Mixup can be understood as a form of data augmentation that encourages the model to behave linearly in-between training examples.
- This linear behaviour reduces the amount of undesirable oscillations when predicting outside the training examples (unseen data).




https://arxiv.org/abs/1710.09412

# Noise robustness

- Label Smoothing:
  - **Training:** Train with slightly noisy labels
  - **Test:** Use original images
- The problem with cross-entropy loss is the hard targets. The model has to produce large *logit* value for the correct label. It encourages the differences between the largest logit and all others to become large, and this, combined with the bounded gradient reduces the ability of the model to adapt, resulting in a model **too confident about its predictions**.
- This is particularly bad if the dataset has some amount of mistakes in the *y* labels (which most data sets have). All of this, in turn, can **lead to overfitting**.
- The assumption underlying Label Smoothing is that for some small constant *e*, the training set label y is correct with probability 1 − *e*, and otherwise any of the other possible labels might be correct. Label smoothing regularizes a model based on a softmax with k output values by replacing the hard 0 and 1 classification targets with targets of $e/(k-1)$ and $1 - e$, respectively.

https://medium.com/@nainaakash012/when-does-label-smoothing-help-89654ec75326

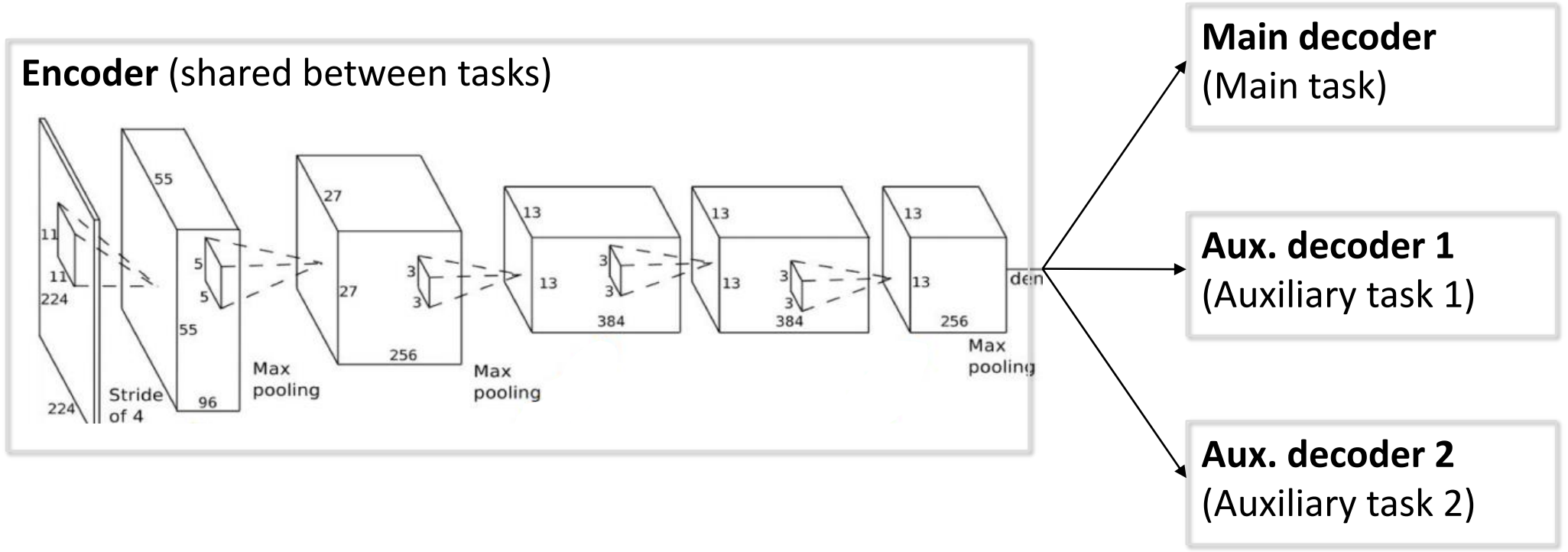
Szegedy (2015): Rethinking the Inception Architecture for Computer Vision

# Multi-task learning

- We generally train a single model or an ensemble of models to perform our desired task by **focusing on optimizing for a particular metric**.
- By being laser-focused on our single task, we ignore information that might help us do even better on the metric we care about. Specifically, this information comes from the training signals of related tasks.
- By **sharing representations between related tasks**, we can enable our model to generalize better on our original task.
- This approach is called Multi-Task Learning (MTL).

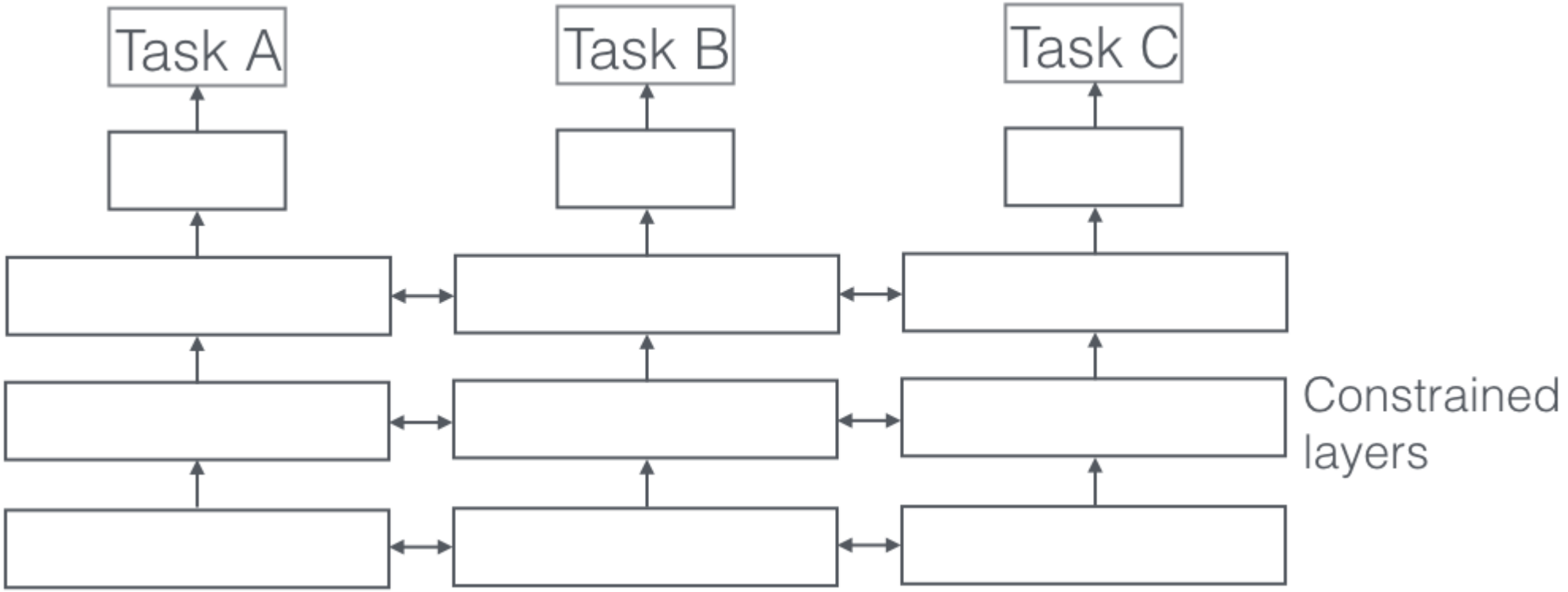
# Multi-task learning

- **Hard parameter sharing -** train the same encoder to solve multiple tasks (with multiple decoders)
- At test time only use the decoder of your main task.
- Hard parameter sharing greatly reduces the risk of overfitting.



# Multi-task learning

- **Soft parameter sharing -** In soft parameter sharing on the other hand, each task has its own model with its own parameters. The distance between the parameters of the model is then regularized in order to encourage the parameters to be similar.



# Multi-task learning

- **Why does it work?**
- MTL effectively increases the sample size that we are using for training our model.
- As all tasks are at least somewhat noisy, when training a model on some task A, our aim is to learn a good representation for task A that ideally ignores the data-dependent noise and generalizes well.
- As different tasks have different noise patterns, a **model that learns two tasks simultaneously is able to learn a more general representation**.
- Learning just task A bears the risk of overfitting to task A, while learning A and B jointly enables the model to obtain a better representation F through **averaging the noise patterns**.

# Multi-task learning: examples

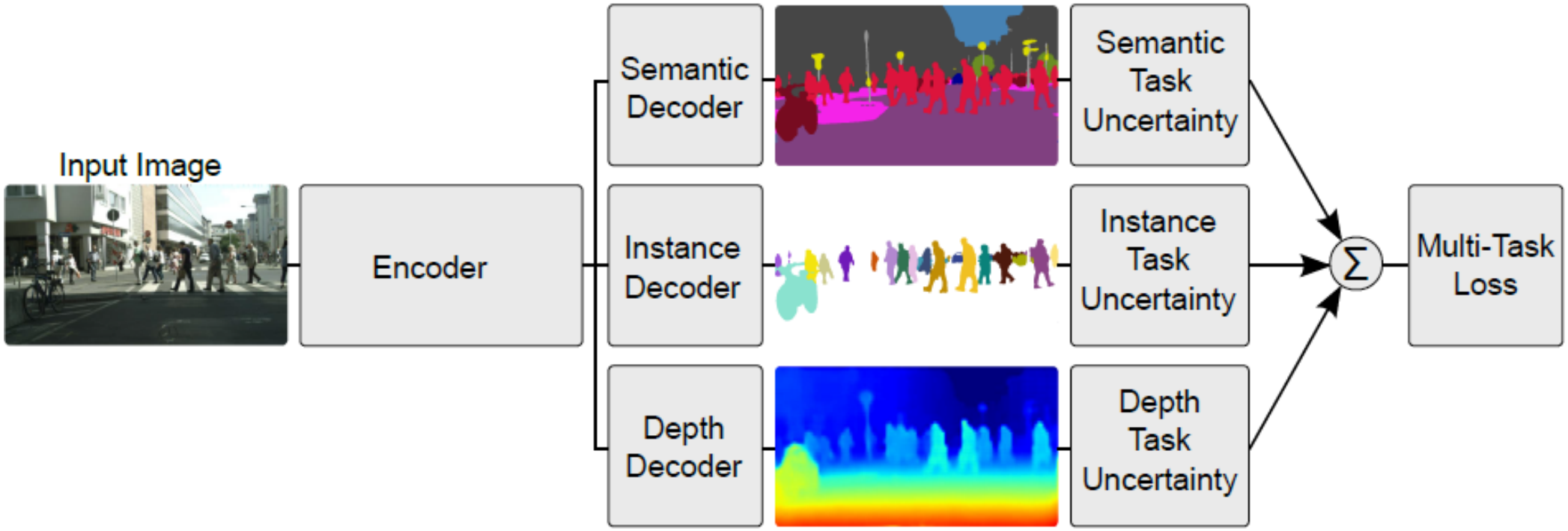
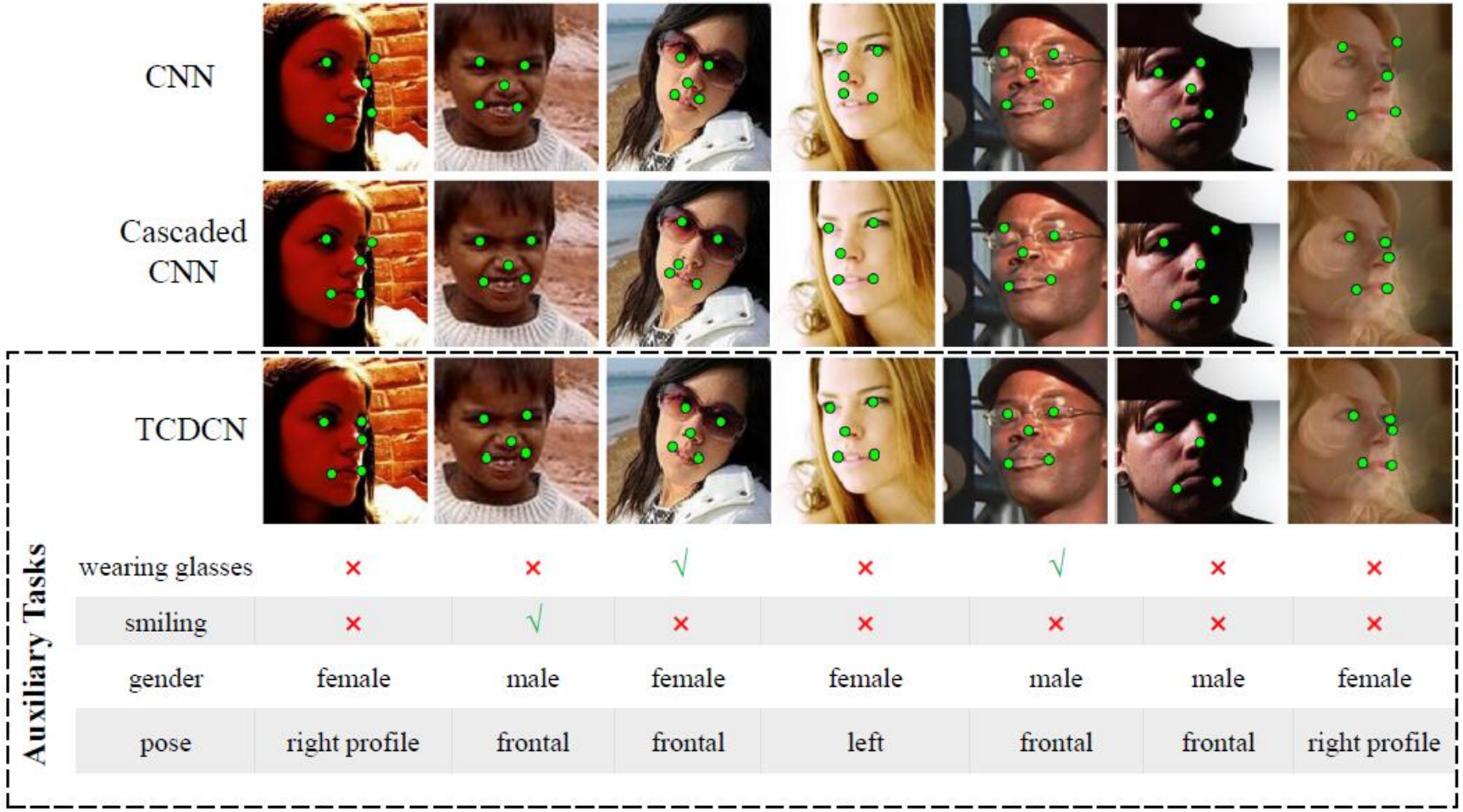



Figure 1: **Multi-task deep learning.** We derive a principled way of combining multiple regression and classification loss functions for multi-task learning. Our architecture takes a single monocular RGB image as input and produces a pixel-wise classification, an instance semantic segmentation and an estimate of per pixel depth. Multi-task learning can improve accuracy over separately trained models because cues from one task, such as depth, are used to regularize and improve the generalization of another domain, such as segmentation.

Kendall et al: Multi-Task Learning Using Uncertainty to Weigh Losses for Scene Geometry and Semantics

# Multi-task learning: examples




Zhang et al: Facial Landmark Detection by Deep Multi-task Learning

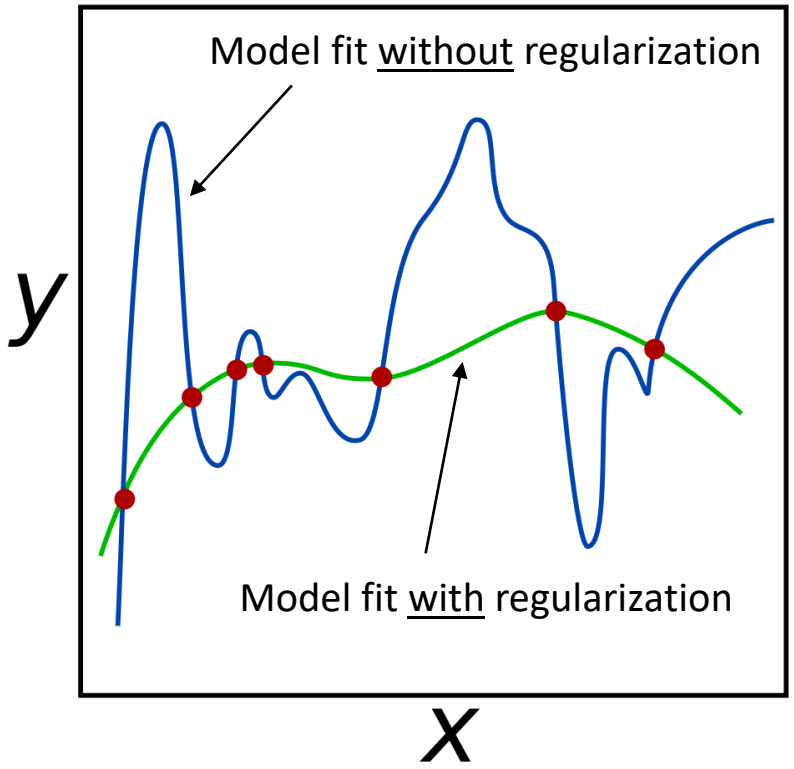
# And of course: Weight decay

- Extend the data loss $J(w)$ with an extra term

$$J_{reg}(w) = J(w) + \lambda R(w)$$

where R(w) is either the L1-norm or the L2-norm of $w$.

- The last term as called the *regularization term*, and $\lambda$ is the regularization parameter.
- $\lambda$ determines the trade-off between minimizing the data loss and minimizing the model parameters $w$.
- By keeping the weights small, the regularization term makes the model simpler to **avoid overfitting**.
- So weight decay pushes against fitting the data *too* well so **we don't fit noise in the data**.



# Effect of weight decay

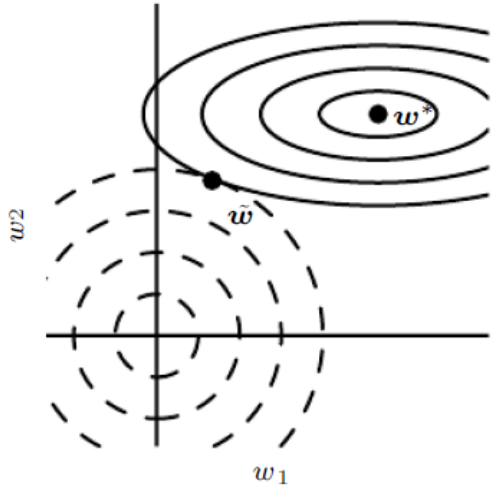
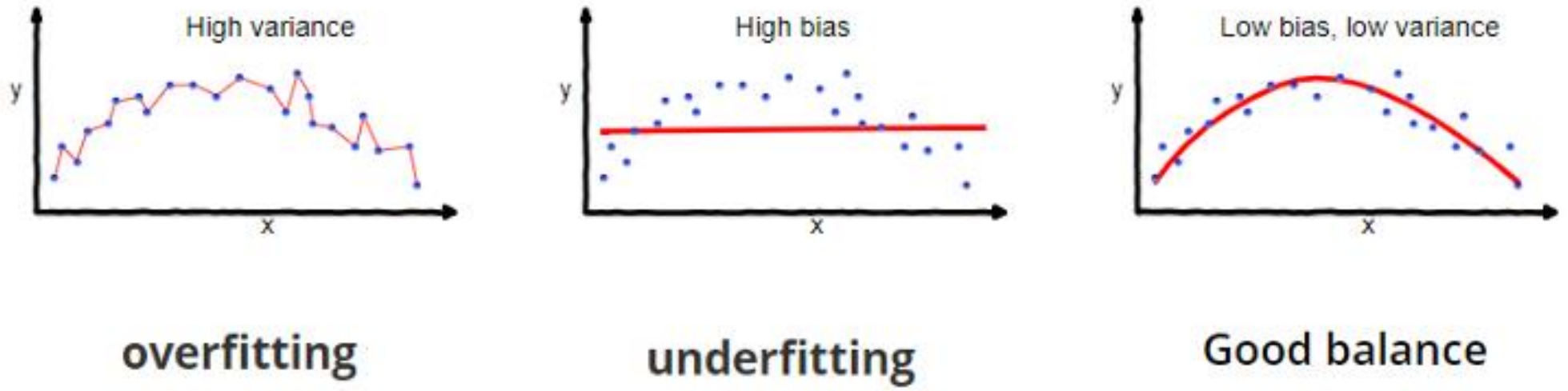



Figure 7.1: An illustration of the effect of $L^2$ (or weight decay) regularization on the value of the optimal $\boldsymbol{w}$. The solid ellipses represent contours of equal value of the unregularized objective. The dotted circles represent contours of equal value of the $L^2$ regularizer. At the point $\tilde{\boldsymbol{w}}$, these competing objectives reach an equilibrium. In the first dimension, the eigenvalue of the Hessian of $J$ is small. The objective function does not increase much when moving horizontally away from $\boldsymbol{w}^*$. Because the objective function does not express a strong preference along this direction, the regularizer has a strong effect on this axis. The regularizer pulls $w_1$ close to zero. In the second dimension, the objective function is very sensitive to movements away from $\boldsymbol{w}^*$. The corresponding eigenvalue is large, indicating high curvature. As a result, weight decay affects the position of $w_2$ relatively little.

# Side-note: Bias-variance tradeoff

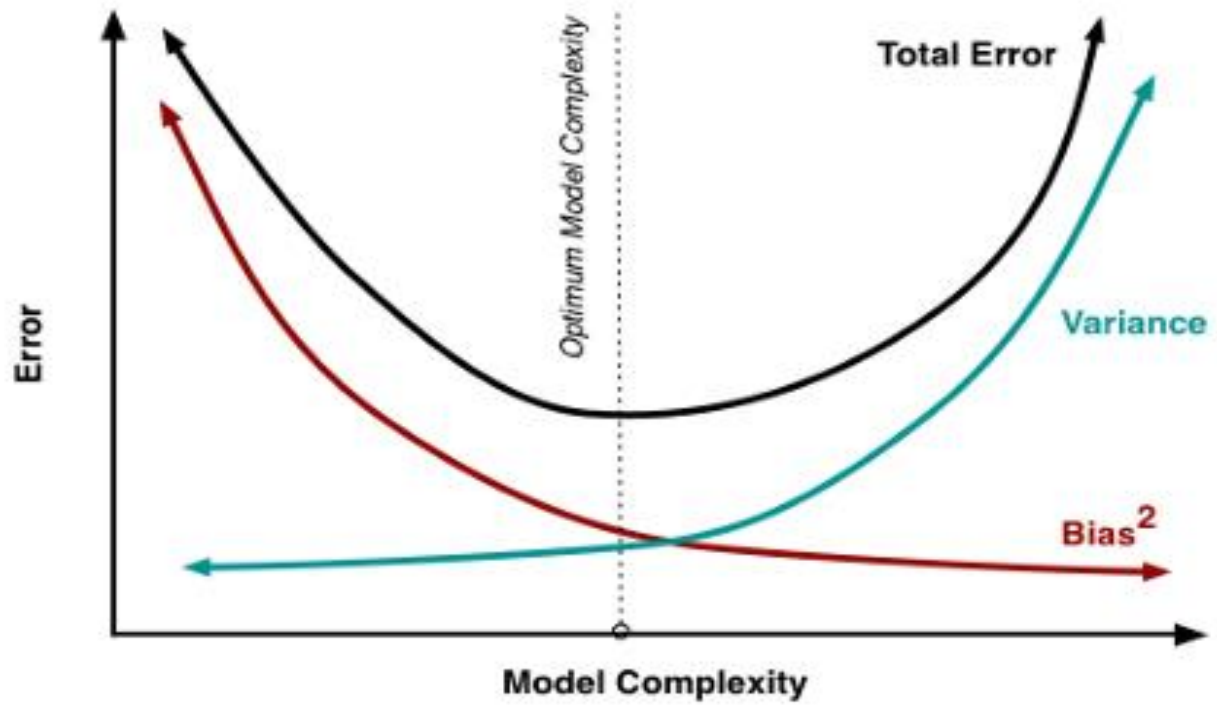
- **Bias** results from erroneous assumptions in the model. High bias can cause a model to miss the relevant relations between features and target outputs (underfitting).
- **Variance** results from high sensitivity to small fluctuations in the training set. High variance can cause a model to mimic random noise in the training data, rather than the true distribution of the data (overfitting).




https://towardsdatascience.com/understanding-the-bias-variance-tradeoff-165e6942b229

# Side-note: Bias-variance tradeoff

- If our model is too simple and has very few parameters then it may have high bias and low variance. On the other hand if our model has large number of parameters then it's going to have high variance and low bias. So we need to find the right/good balance without overfitting and underfitting the data. This is essentially the bias-variance tradeoff.




https://towardsdatascience.com/understanding-the-bias-variance-tradeoff-165e6942b229

# In practise

- Always use early stopping and batch normalization.
- Regularization terms are often included by default.
- Consider dropout for large fully-connected layers.
- Data augmentation almost always a good idea.
- Try cutout, mixup, label smoothing especially for small classification datasets.
- Try multi-task learning if your dataset has training data to solve more than one task.

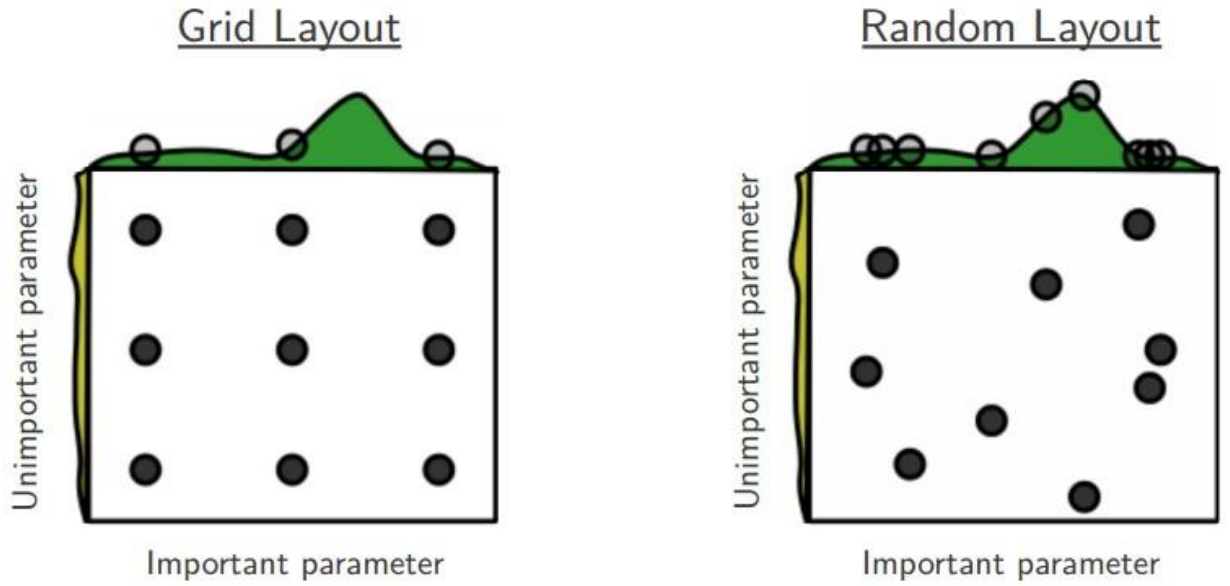
# Hyperparameter tuning

# Motivation

- As we've seen, training neural networks can involve **many hyperparameter settings**.
- The most common hyperparameters include:
  - the initial learning rate
  - learning rate decay schedule (such as the decay constant)
  - regularization strength (L2 penalty, dropout strength)
  - batch size
- There are many more relatively less sensitive hyperparameters, for example in per-parameter adaptive learning methods, the setting of momentum and its schedule, etc.
- **How do we find the optimal hyperparameters?**
- Really nice guide: https://towardsdatascience.com/guide-to-choosing-hyperparameters-for-your-neural-networks-38244e87dafe

# General recommendations

- Performing hyperparameter search can take many hours/days/weeks.
- **Use a single validation set of respectable size.**
- **Search for hyperparameters on log scale:** A typical sampling of the learning rate would be $10^{-6}$, $10^{-5}$, $10^{1}$. Some parameters (e.g. dropout) are instead usually searched in the original scale.
- **Prefer random search to grid search** (its usually faster and easier to spot important parameters).




Bergstra & Bengio: Random Search for Hyper-Parameter Optimization

# General recommendations

- **Careful with best values on border.** Sometimes it can happen that you're searching for a hyperparameter (e.g. learning rate) in a bad range. Once we receive the results, it is important to double check that the final learning rate is not at the edge of this interval, or otherwise you may be missing more optimal hyperparameter setting beyond the interval.
- **Stage your search from coarse to fine.** In practice, it can be helpful to first search in coarse ranges, and then depending on where the best results are turning up, narrow the range.
- **Start with 1 epoch or less (coarse).** Also, it can be helpful to perform the initial coarse search while only training for 1 epoch or even less, because many hyperparameter settings can lead the model to not learn at all, or immediately explode with infinite cost.
- **… then add more epochs (fine).** The second stage could then perform a narrower search with 5 epochs, and the last stage could perform a detailed search in the final range for many more epochs (for example).

Bengio (2012): Practical recommendations for gradient-based training of deep architectures

# Possible strategy

- Step 1: Check initial loss

  - Just a sanity check
  - Does it seem reasonable?

# Possible strategy

- Step 1: Check initial loss
- Step 2: Overfit a small sample

- Try to train to 100% training accuracy on a small sample of training data (~5-10 minibatches)
- This should always be possible if 1) your model has enough capacity, and 2) you remember to set regularization to zero.
- Fiddle with architecture, learning rate, batch size, weight initialization
- Loss not going down? Learning rate too low, bad initialization, batch size too low
- Loss explodes to Inf or NaN? Learning rate too high, bad initialization

# Possible strategy

- Step 1: Check initial loss
- Step 2: Overfit a small sample
- Step 3: Find learning rate that makes loss go down

- Use the architecture from the previous step, use all training data, turn on small weight decay
- Find a learning rate that makes the loss drop significantly within ~100 iterations
- Good learning rates to try: 1e-1, 1e-2, 1e-3, 1e-4

# Possible strategy

- Step 1: Check initial loss
- Step 2: Overfit a small sample
- Step 3: Find learning rate that makes loss go down
- Step 4: Course grid, train for 1-5 epochs

- Choose a few values of learning rate and weight decay around what worked from Step 3, train a few models for ~1-5 epochs.
- Good weight decay to try: 1e-4, 1e-5, 0

# Possible strategy

- Step 1: Check initial loss
- Step 2: Overfit a small sample
- Step 3: Find learning rate that makes loss go down
- Step 4: Course grid, train for 1-5 epochs
- Step 5: Refine grid, train longer

- Pick best models from Step 4, train them for longer (~10-20 epochs) without learning rate decay

# Possible strategy

- Step 1: Check initial loss
- Step 2: Overfit a small sample
- Step 3: Find learning rate that makes loss go down
- Step 4: Course grid, train for 1-5 epochs
- Step 5: Refine grid, train longer
- **Step 6: Look at loss curves**

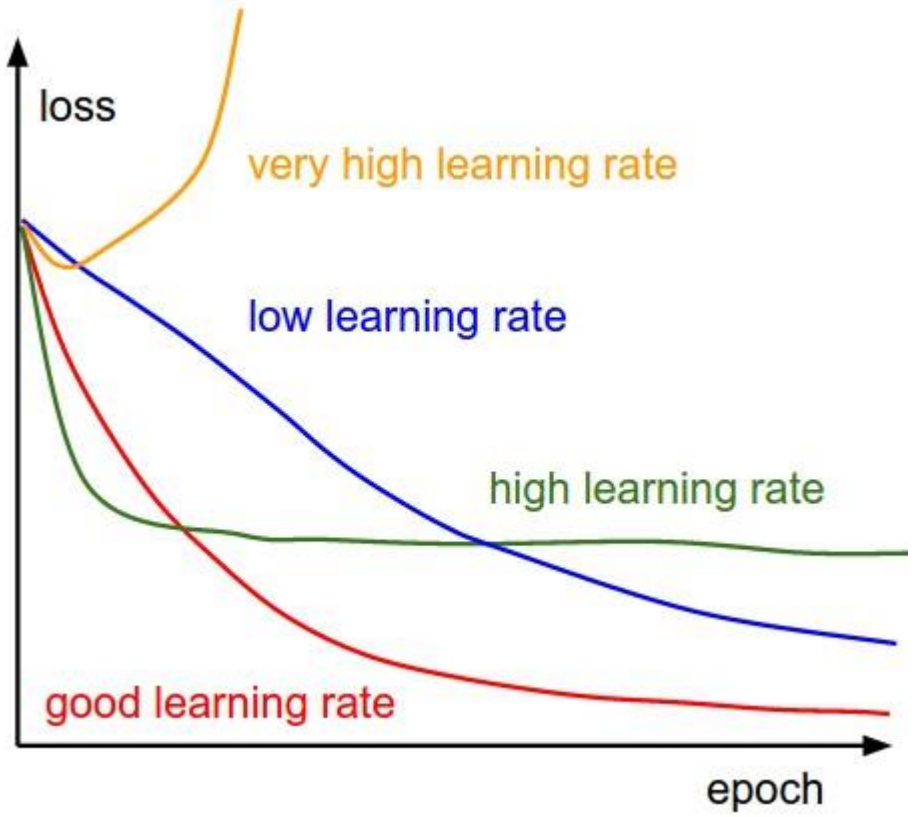
- This is the final and most important step (explained in more detail in the next section)
- If the loss curves don't look right, you'll need to go back an repeat one or more of the previous steps.

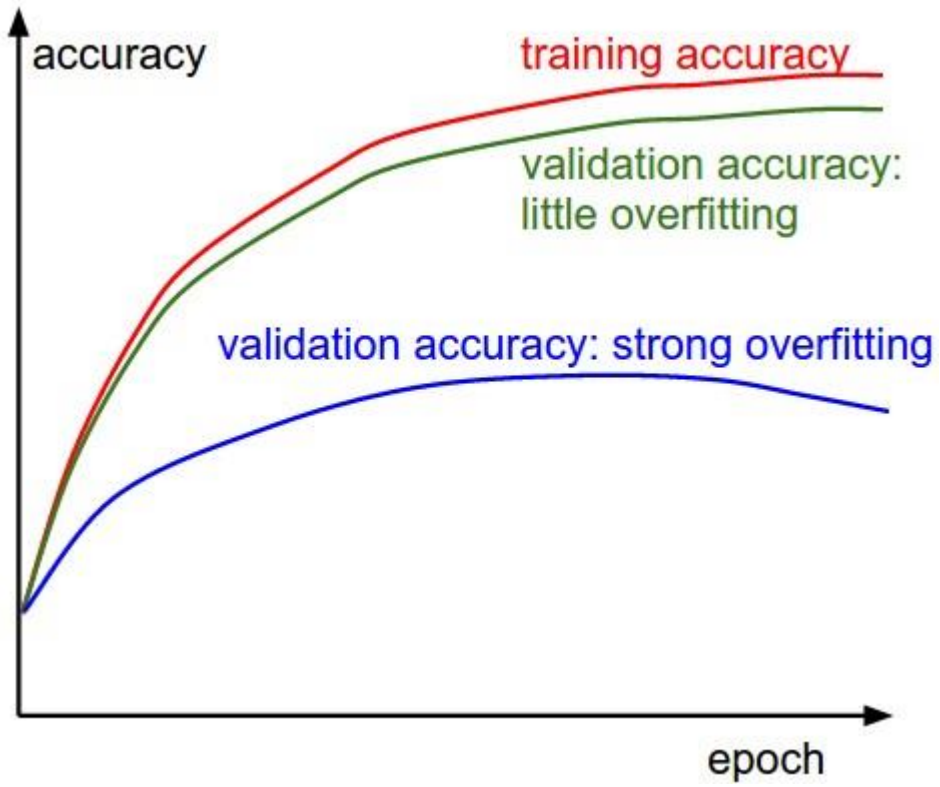
# Babysitting the learning process

# What to monitor during training?

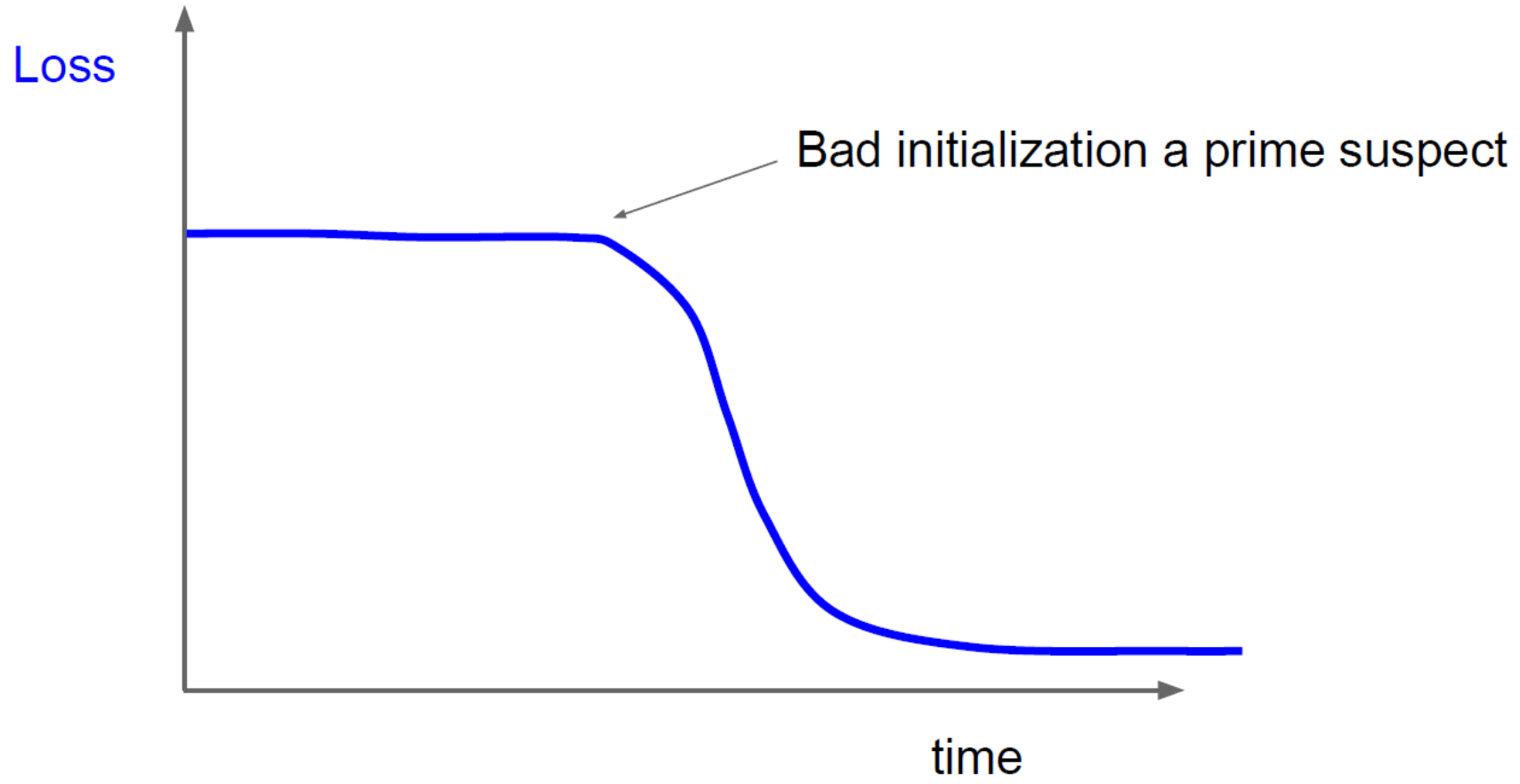
- There are multiple useful quantities you should monitor during training of a neural network.
- The most important ones are the **loss curves** (including validation error or accuracy).
- These curves are the window into the training process and should be utilized to get intuitions about different hyperparameter settings and how they should be changed for more efficient learning.
- Other useful things to monitor include
  - Activation / Gradient distributions per layer
  - First-layer visualizations

# Quick summary

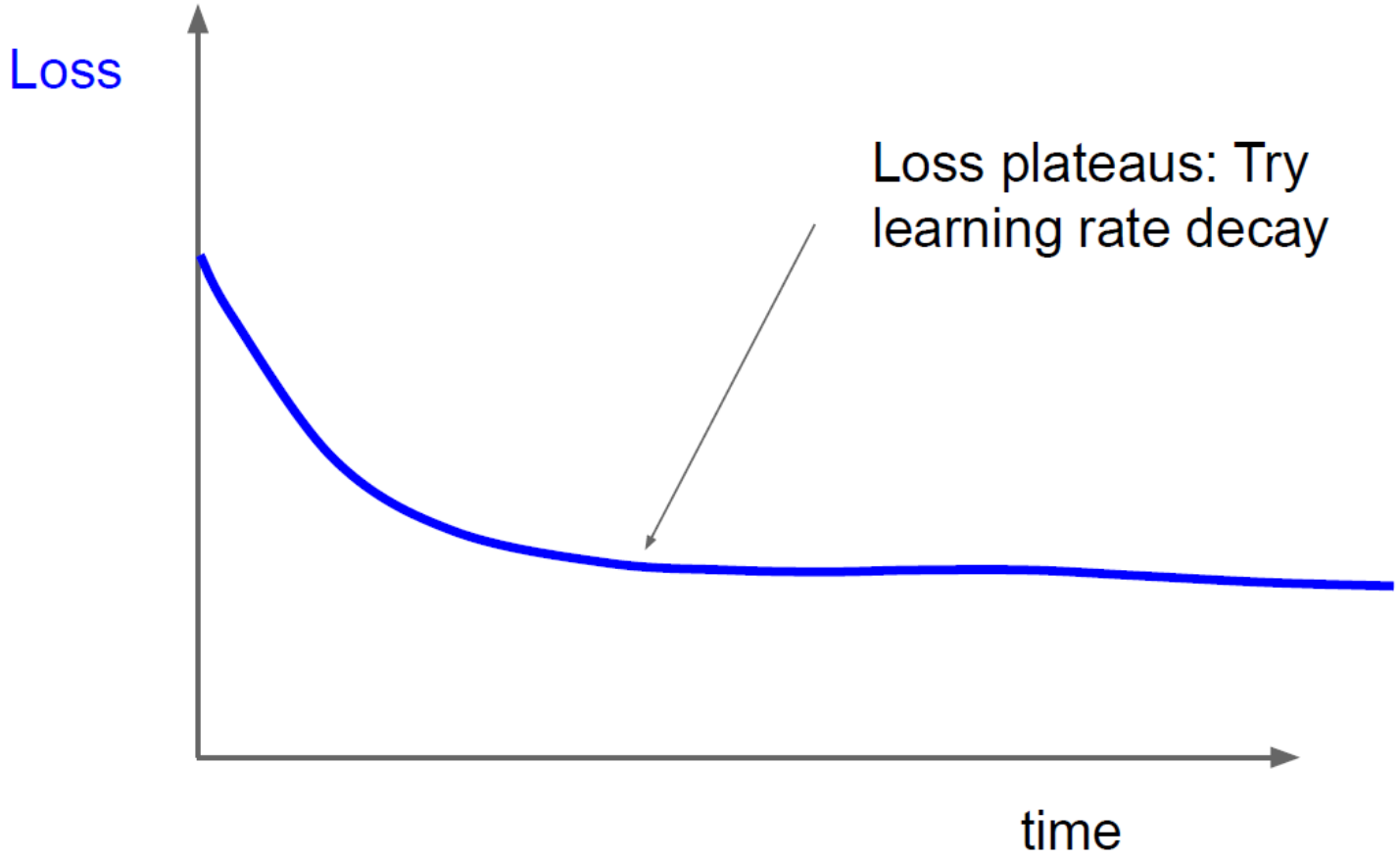


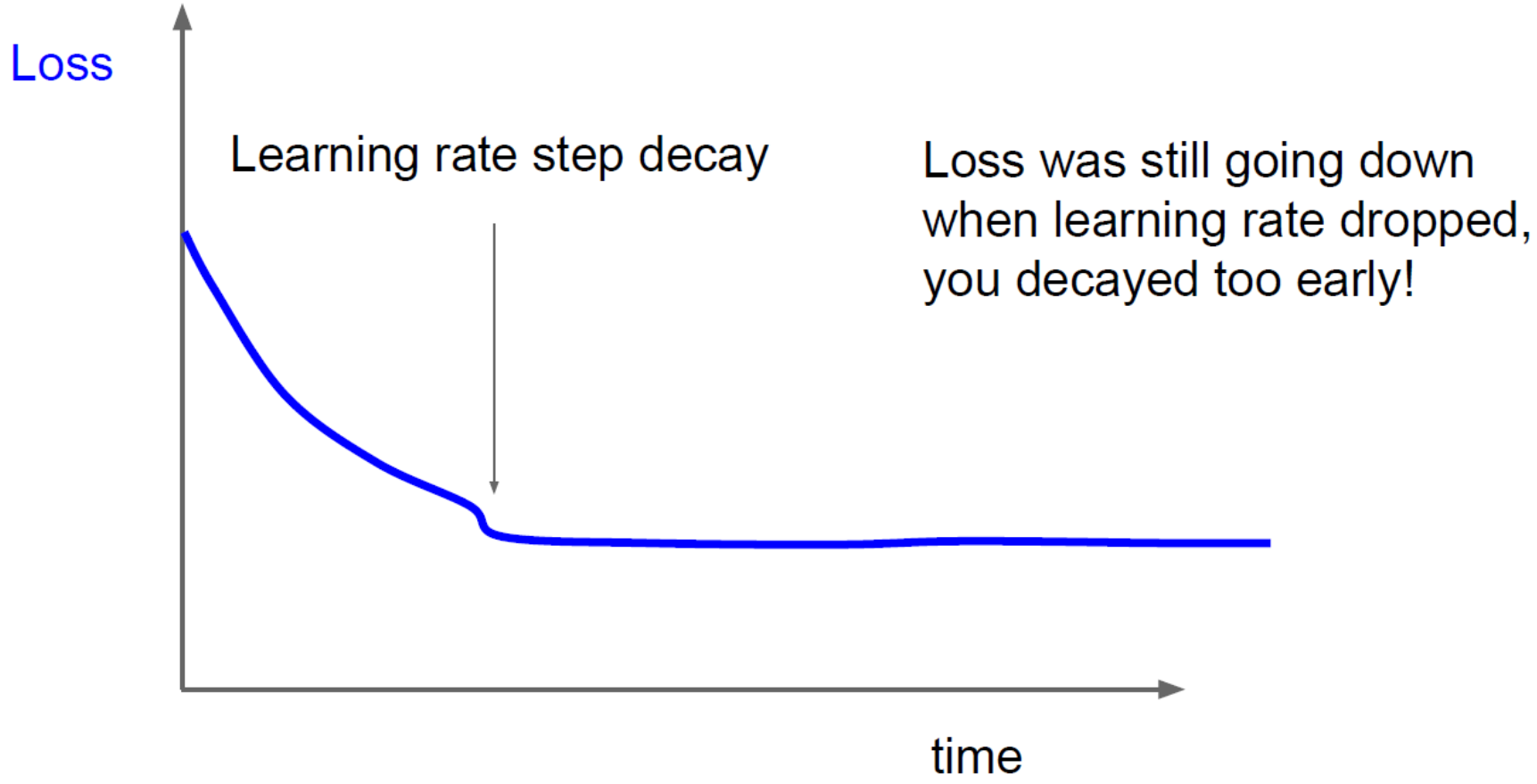
# Loss curves



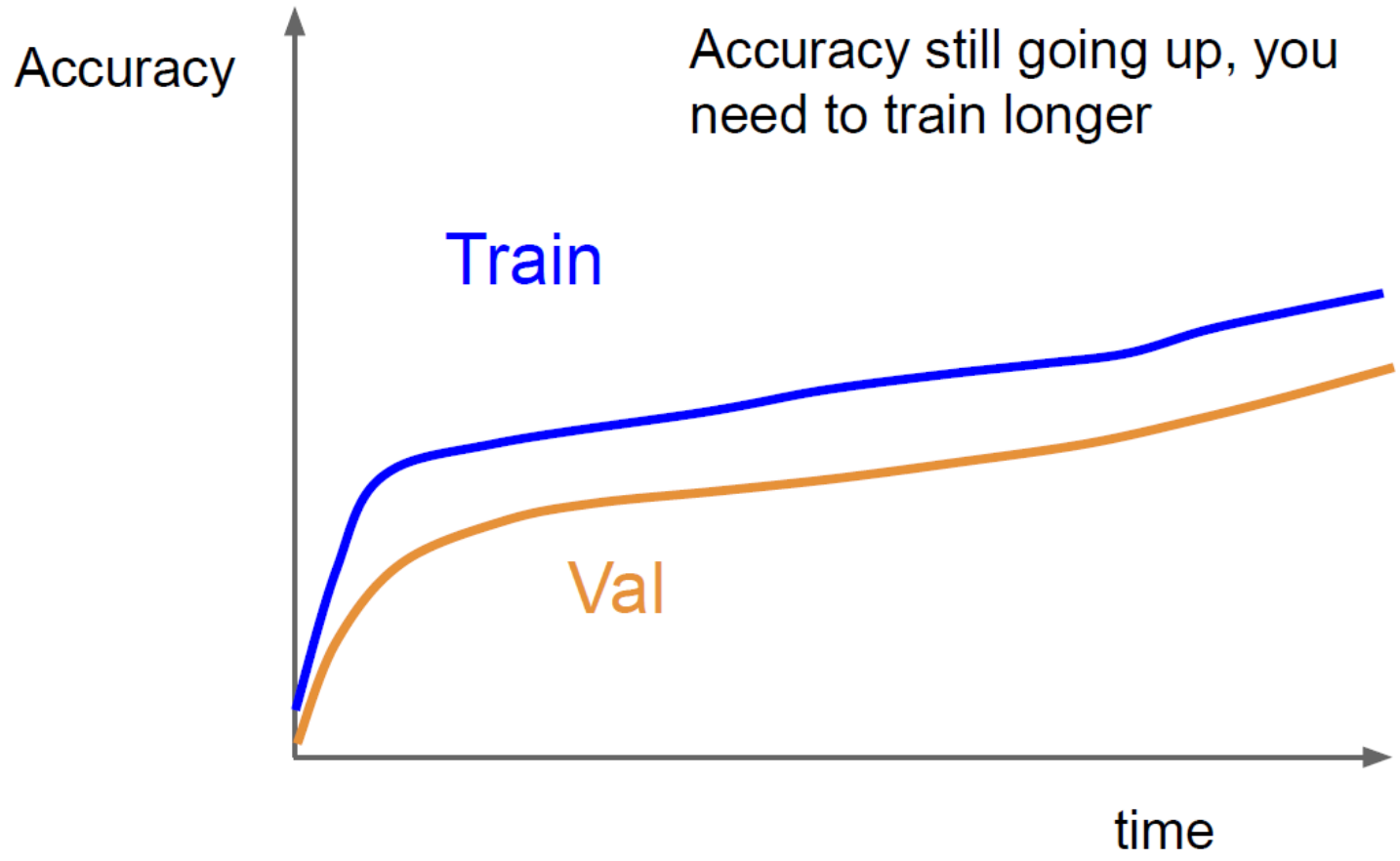
# Loss curves



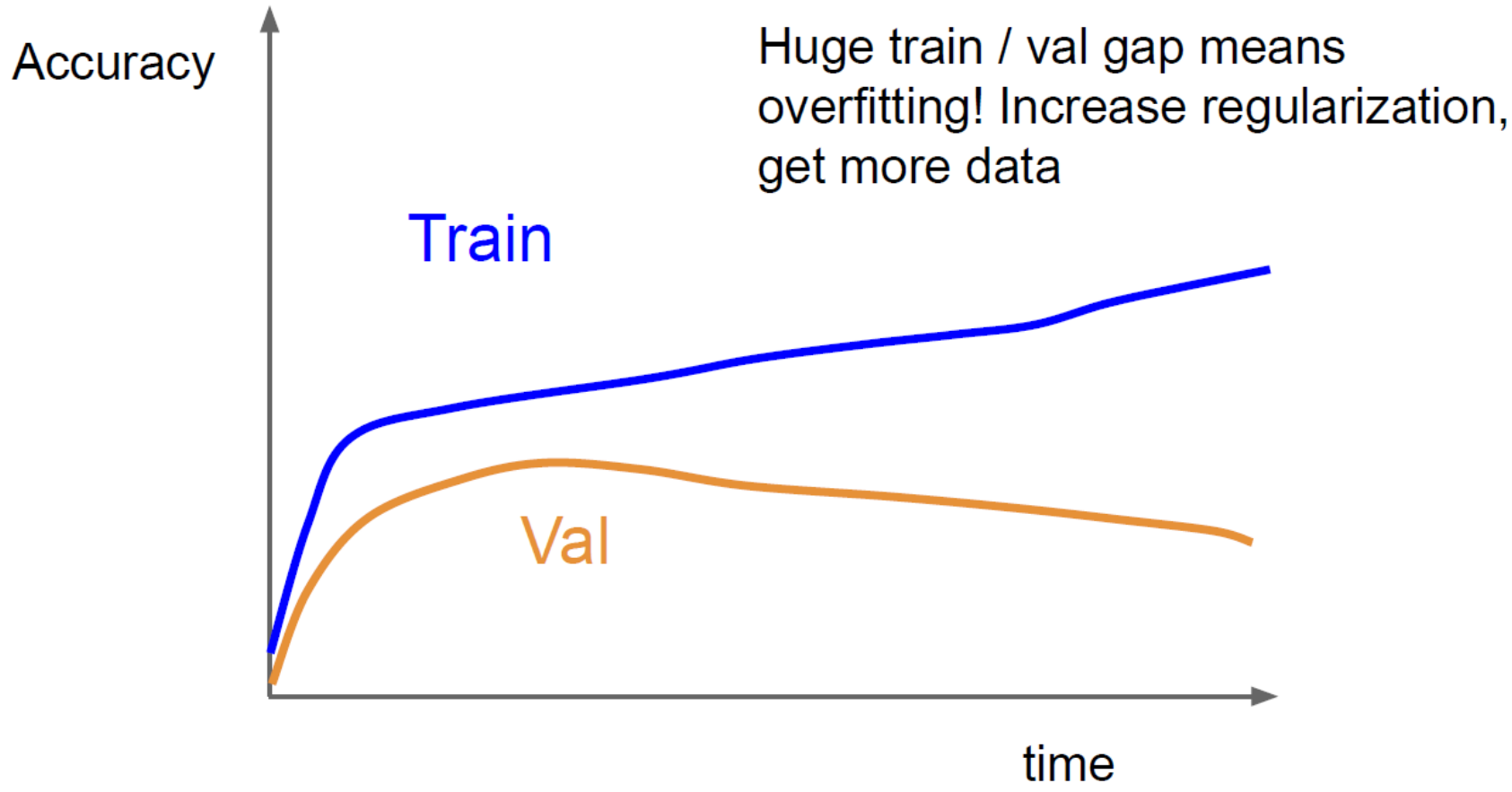
# Loss curves


Loss
Learning rate step decay
Loss was still going down
when learning rate dropped,
you decayed too early!
time

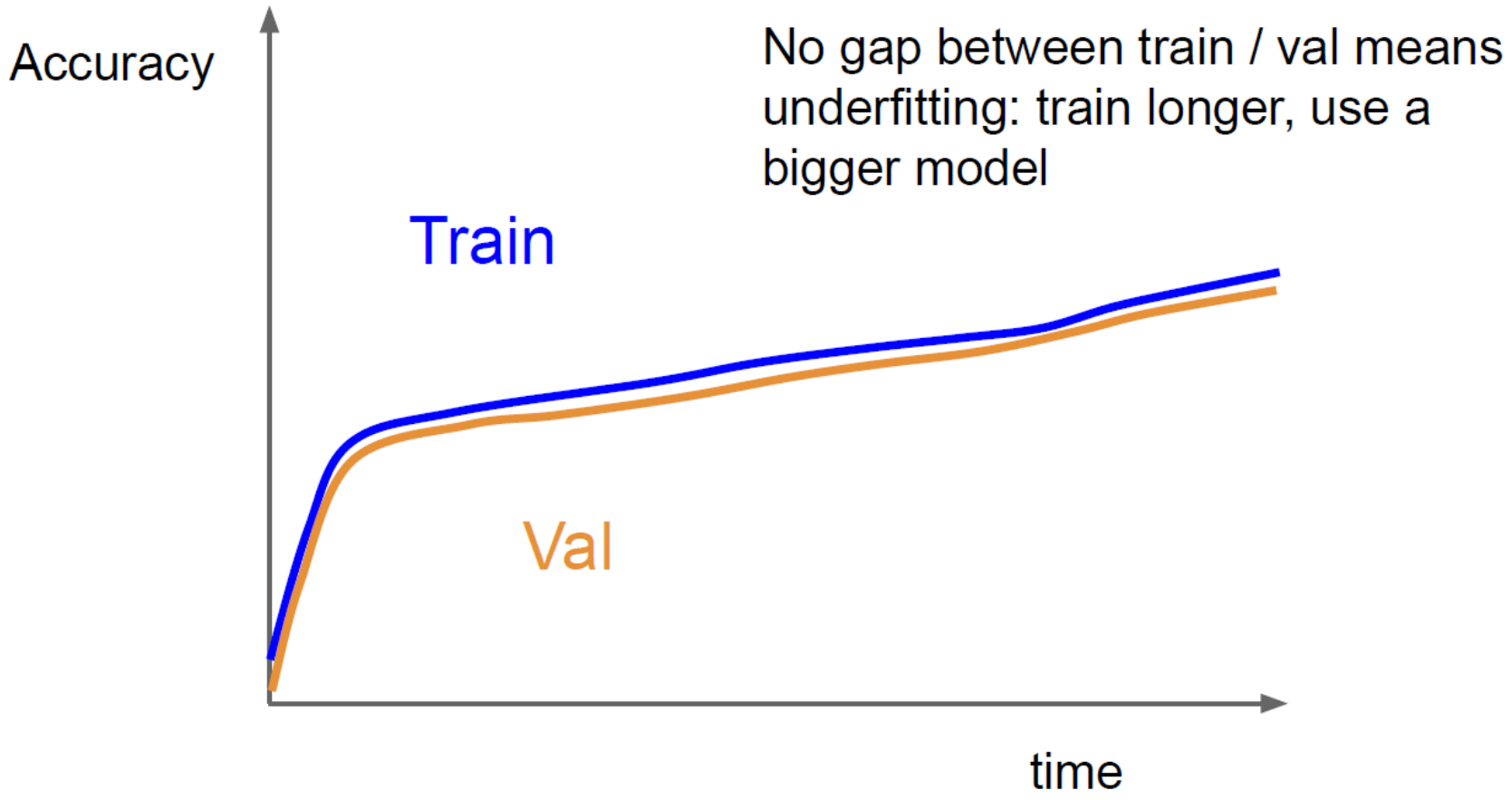
# Train vs validation curves



# Train vs validation curves

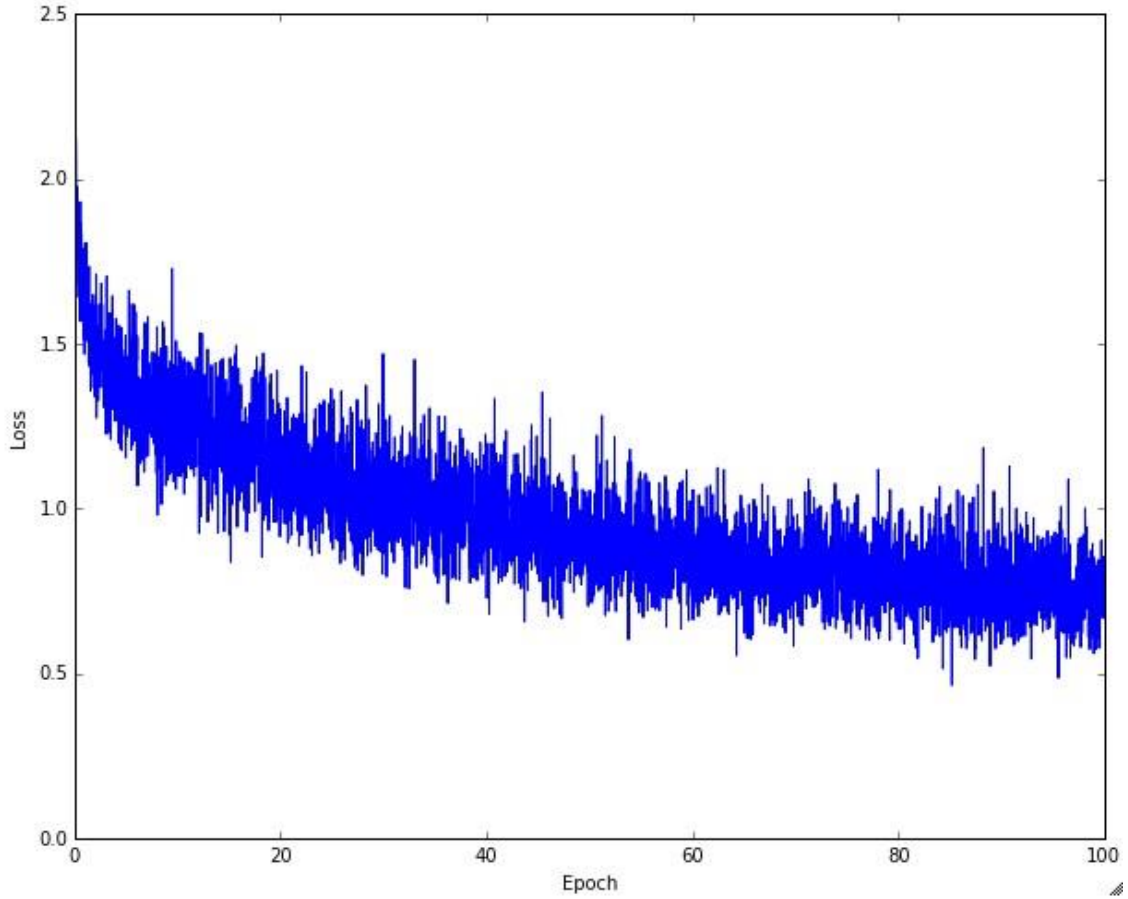


# Train vs validation curves



# Noisy curves

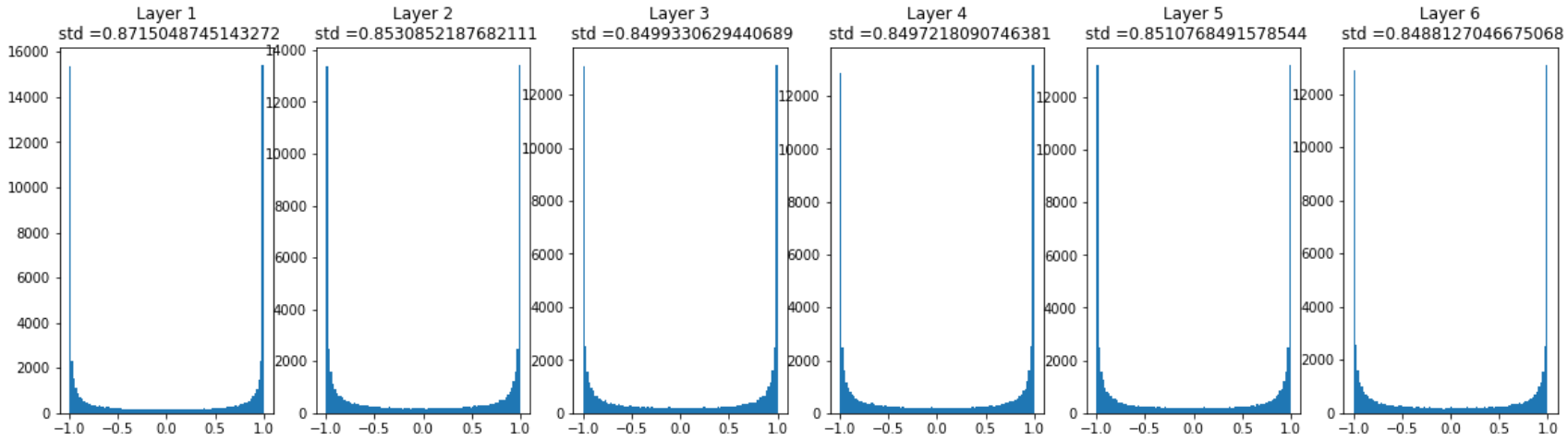
- In practise the curves are always somewhat noisy.
- Look at the overall trend
- This curve looks okay
- If its too noisy, try increasing the batch size
- Why? Because the loss is an average over a mini-batch, and if the batch size is small, the average becomes noisy.



# Activation / Gradient distributions per layer

- An incorrect initialization can slow down or even completely stall the learning process due to **vanishing gradients**.
- This issue can be diagnosed by plotting activation/gradient histograms for all layers of the network.
- Intuitively, it is not a good sign to see any strange distributions - e.g. with tanh neurons we would like to see a distribution of neuron activations between the full range of [-1,1], instead of seeing all neurons outputting zero, or all neurons being completely saturated at either -1 or 1.

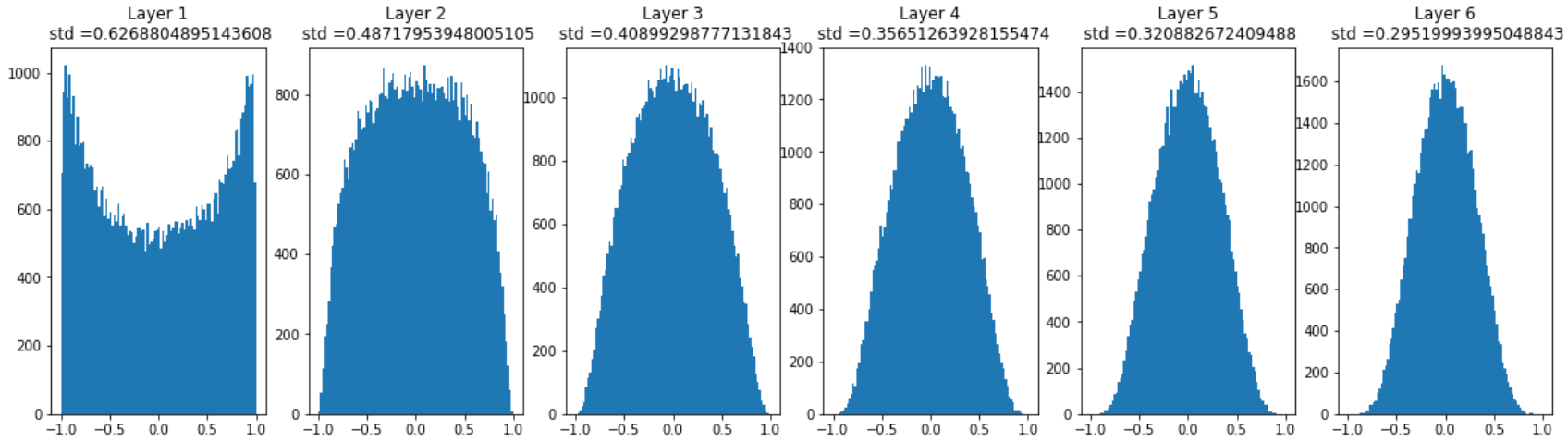
Not ideal



# Activation / Gradient distributions per layer

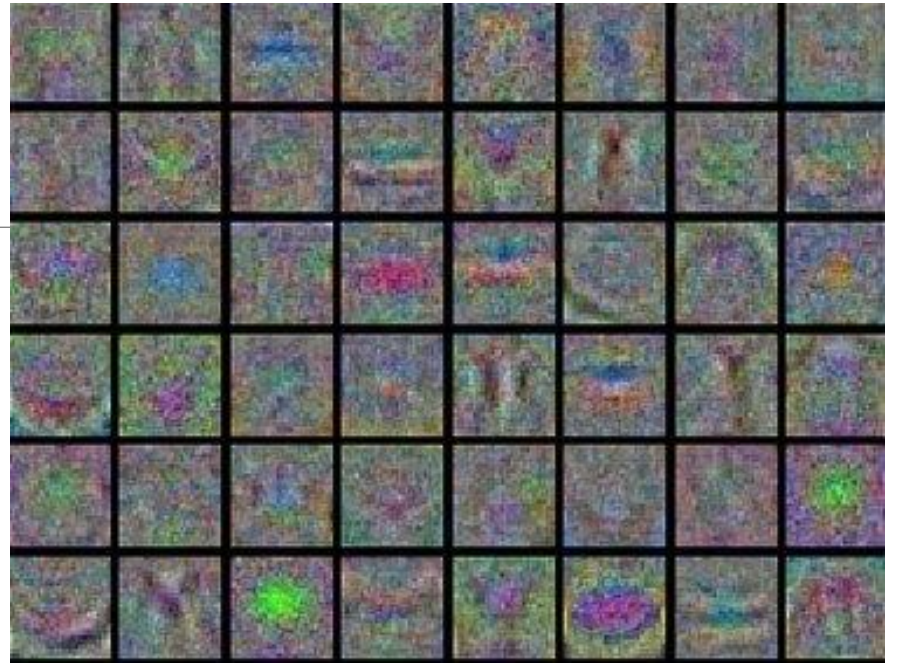
- An incorrect initialization can slow down or even completely stall the learning process due to **vanishing gradients**.
- This issue can be diagnosed by plotting activation/gradient histograms for all layers of the network.
- Intuitively, it is not a good sign to see any strange distributions - e.g. with tanh neurons we would like to see a distribution of neuron activations between the full range of [-1,1], instead of seeing all neurons outputting zero, or all neurons being completely saturated at either -1 or 1.

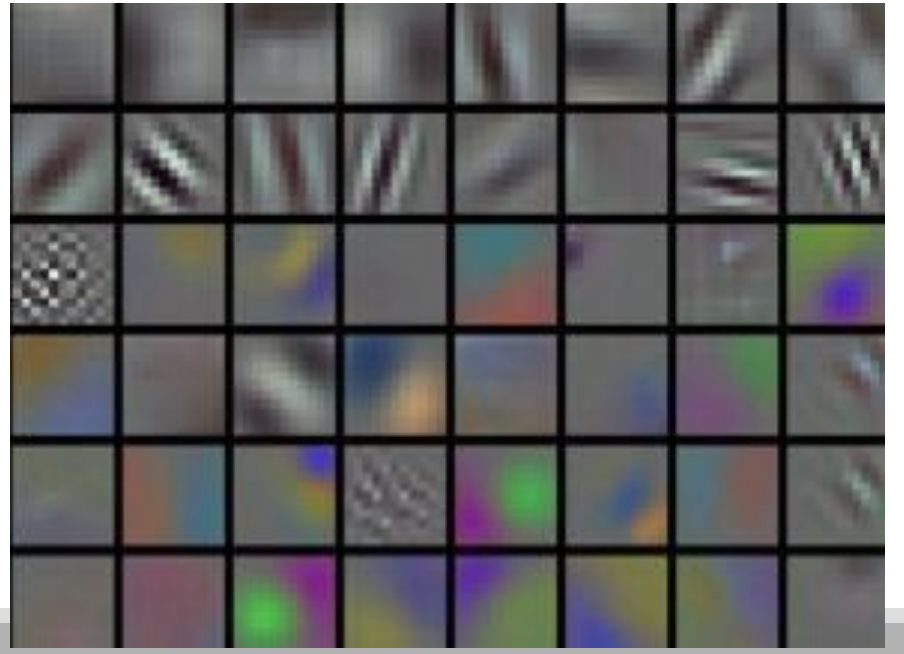
Much better



# First-layer visualizations

- Lastly, when one is working with image pixels it can be helpful and satisfying to plot the first-layer features visually:
- **Top**: Noisy features could indicate
  - un-converged network
  - improperly set learning rate
  - very low weight regularization penalty
- **Bottom:** Nice, smooth, clean and diverse features are a good indication that the training is proceeding well.





# Summary

- **Activation functions:** Use ReLU
- **Data preprocessing:** Subtract mean
- **Weight initialization:** Use Xavier/Kaiming
- **Batch normalization:** Use always
- **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 (next lecture):** 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/