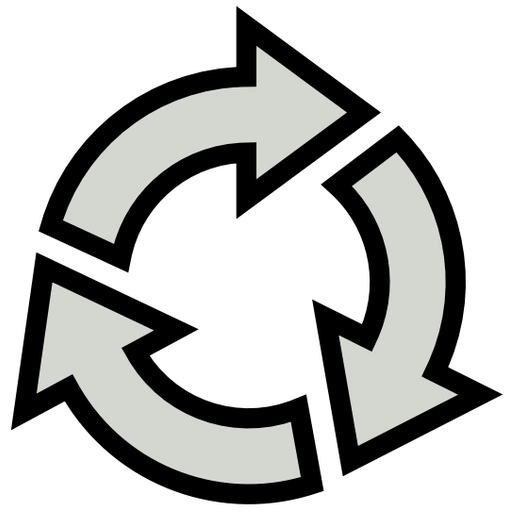
# Practical deep learning

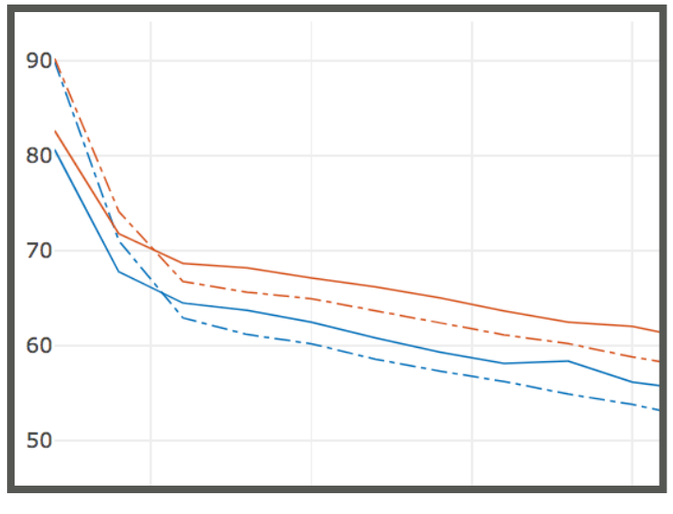
Collect and label data

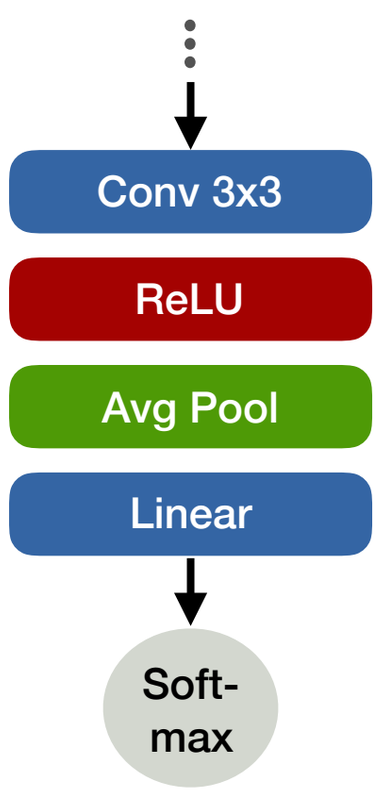
Look at your data

Design and train your model



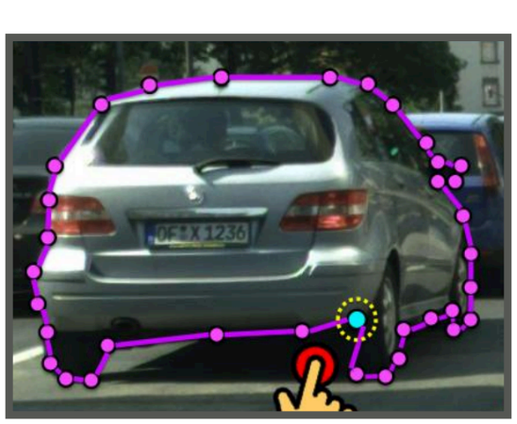
Evaluate your model on unseen data



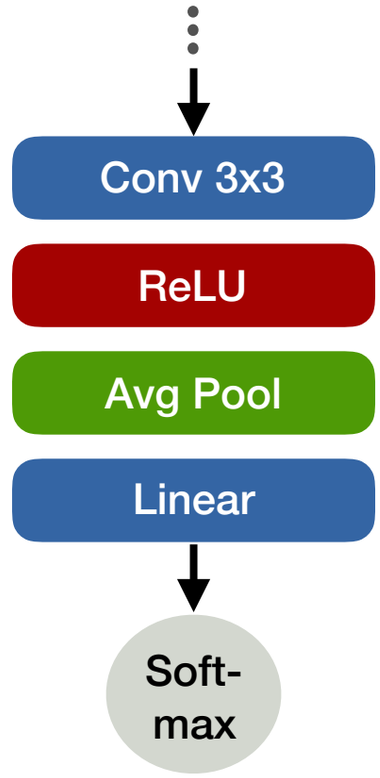
# Graduate student descent

Collect and label data

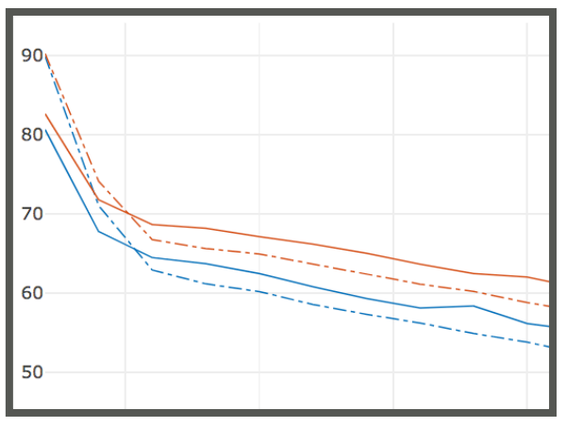


Look at your data

Design and train your model




Evaluate your model on unseen data



# Looking at your data

# Looking at your data

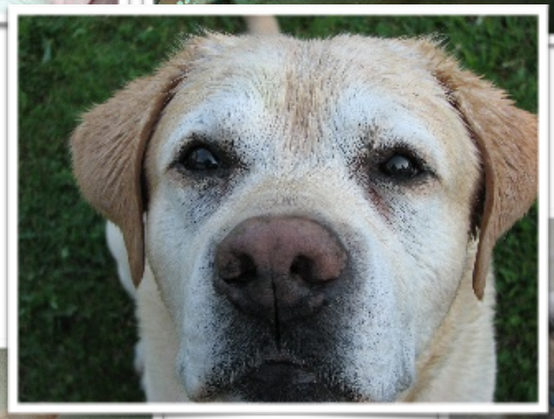
- E.g. images
  - Random ones
  - Smallest / largest file size
  - Try solving the task manually

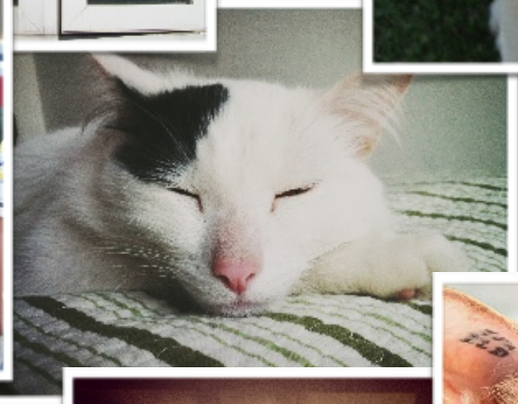
# Random images





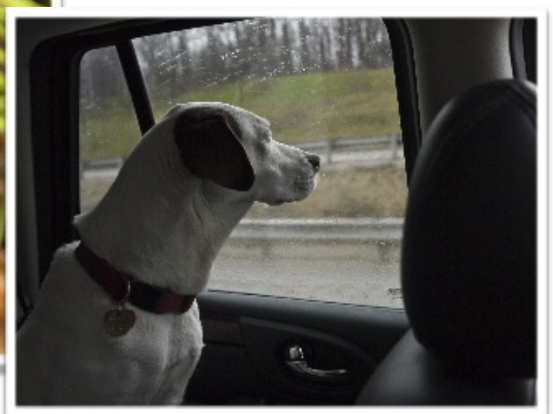








# Largest file size


# Smallest file size

# Solving the task manually













# Training, validation, and test sets

# Dataset

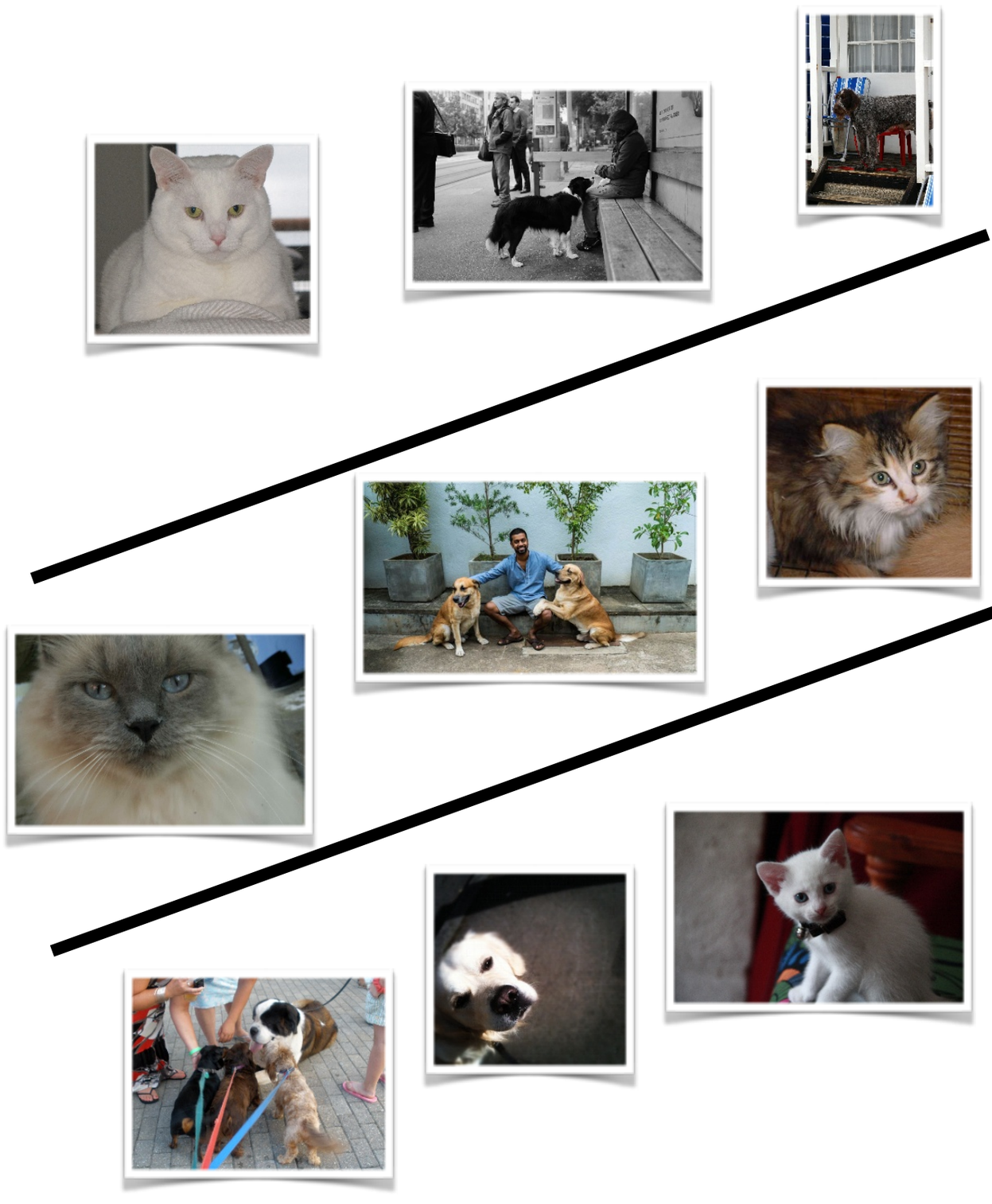
- Training set
  - Learn model parameters
- Validation set
  - Learn hyper-parameters
- Test set
  - Measure generalization performance





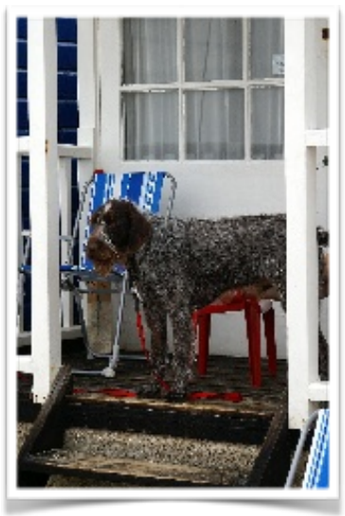

# Why split the data?

- Overfitting
- Goal: Learn a model that works well in the real world
- Optimization objective: Learn a model that works well in training data



# Training set

- Used to train all parameters of the model
- Model will work very well on training set
- Size: 60-80% of data



# Validation set

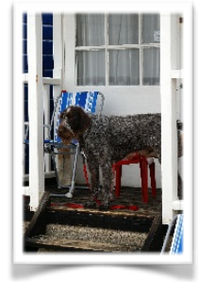
- Used to determine how well the model works
- Used to tune model and hyper-parameters
- Size: 10-20% of data

# Testing set

- Used to measure performance of model on unseen data
- Used exactly once
- Size: 10-20% of data

# How to split the data?

- Random sampling without replacement

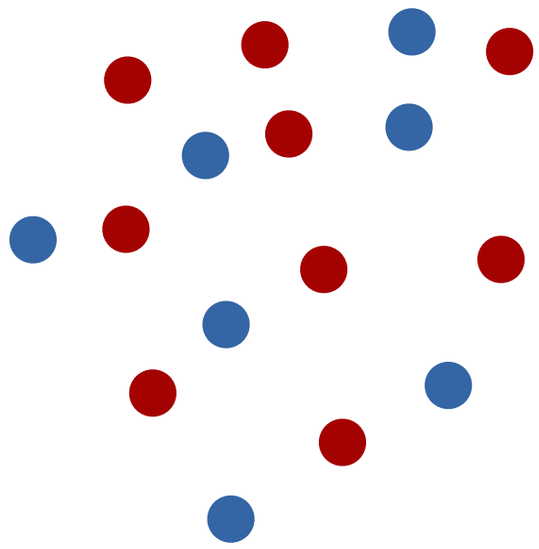







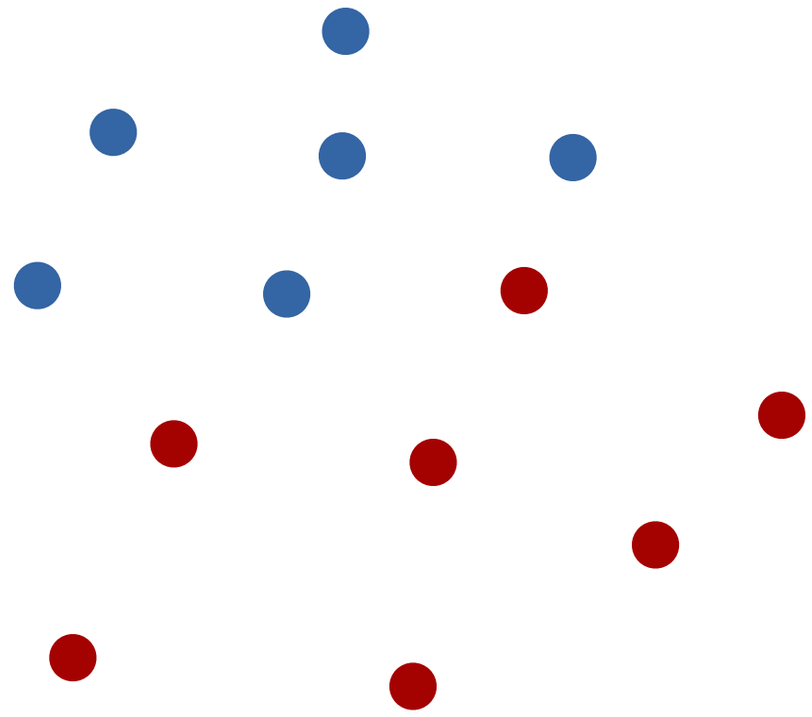

# Distribution of data

Low dimensions



$$D_{data} \approx D_{train} \approx D_{valid} \approx D_{test}$$

High dimensions



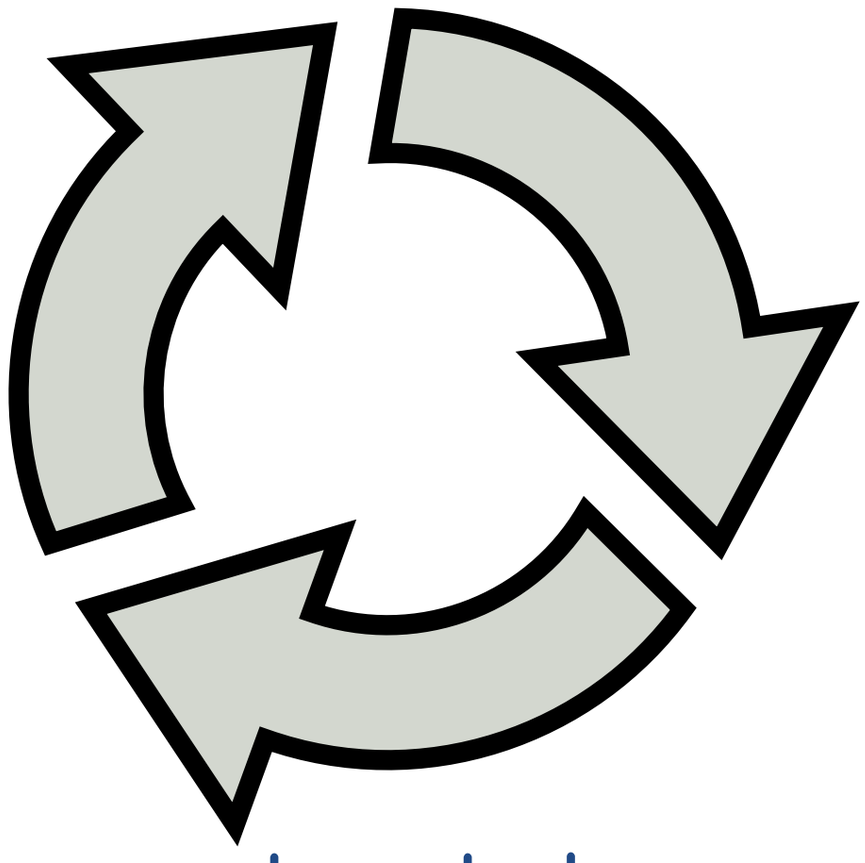
$$D_{data} \neq D_{train} \neq D_{valid} \neq D_{test}$$

# Graduate student descent

Look at your
data / model output

semi-
automated

manual



Evaluate
your model on
validation set

Design and
train your model

automated

# 03

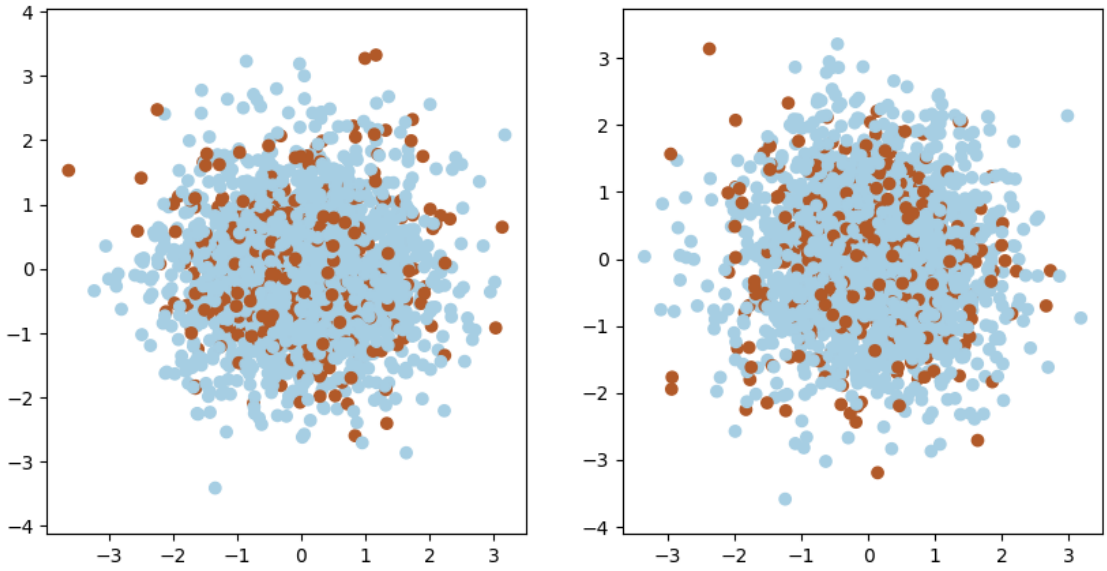
January 23, 2024

[1]:
```
%pylab inline
import numpy as np
from sklearn import linear_model
cls = linear_model.SGDClassifier(max_iter=1000)

# Some sklearn versions spam warnings
import warnings
warnings.simplefilter(action='ignore', category=FutureWarning)
```

```
%pylab is deprecated, use %matplotlib inline and import the required libraries.
Populating the interactive namespace from numpy and matplotlib
```

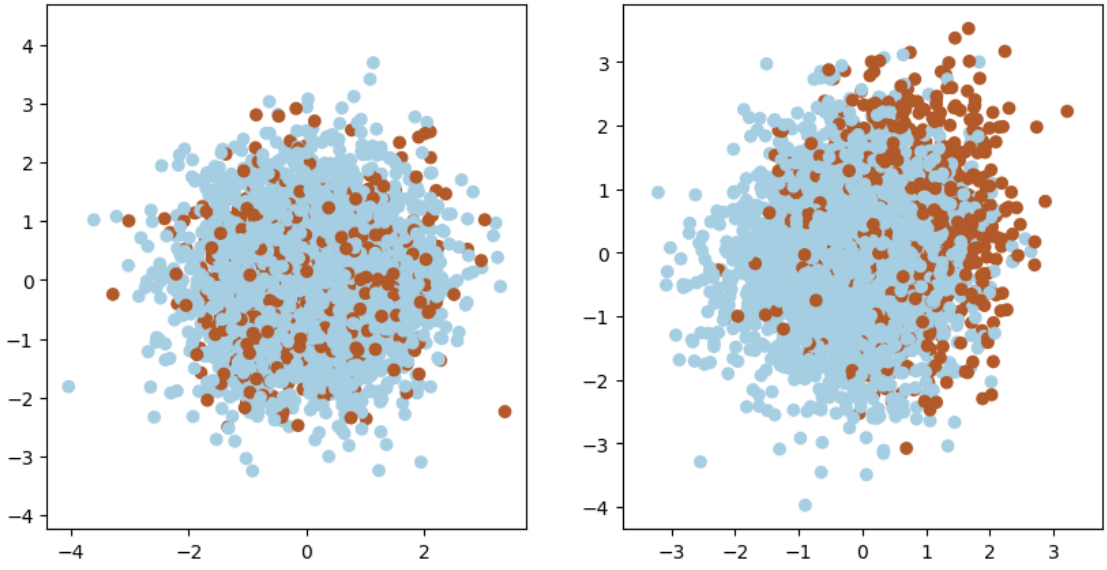
[2]:
```
def separate_points(N=2000, d=2):
    X = np.random.normal(0,1,size=(N, d))
    y = np.random.rand(N) >= 0.8

    # Fit the first classifier
    cls.fit(X, y)
    # Let's project all points along the decision boundary of the first␣
 ↪classifier
    p1 = cls.coef_ / np.linalg.norm(cls.coef_)
    d1 = X.dot(p1.T)
    score = cls.score(X, y)

    cls.fit(X - d1*p1, y)
    p2 = cls.coef_ / np.linalg.norm(cls.coef_)
    d2 = X.dot(p2.T)

    figure(figsize=(10,5))
    subplot(1,2,1)
    scatter(*X[:,:2].T, c=y.flat, cmap='Paired')
    axis('equal')
    subplot(1,2,2)
    scatter(d1.flat, d2.flat, c=y.flat, cmap='Paired')
    axis('equal')

    print( score )
separate_points(2000, 2)
```

0.802



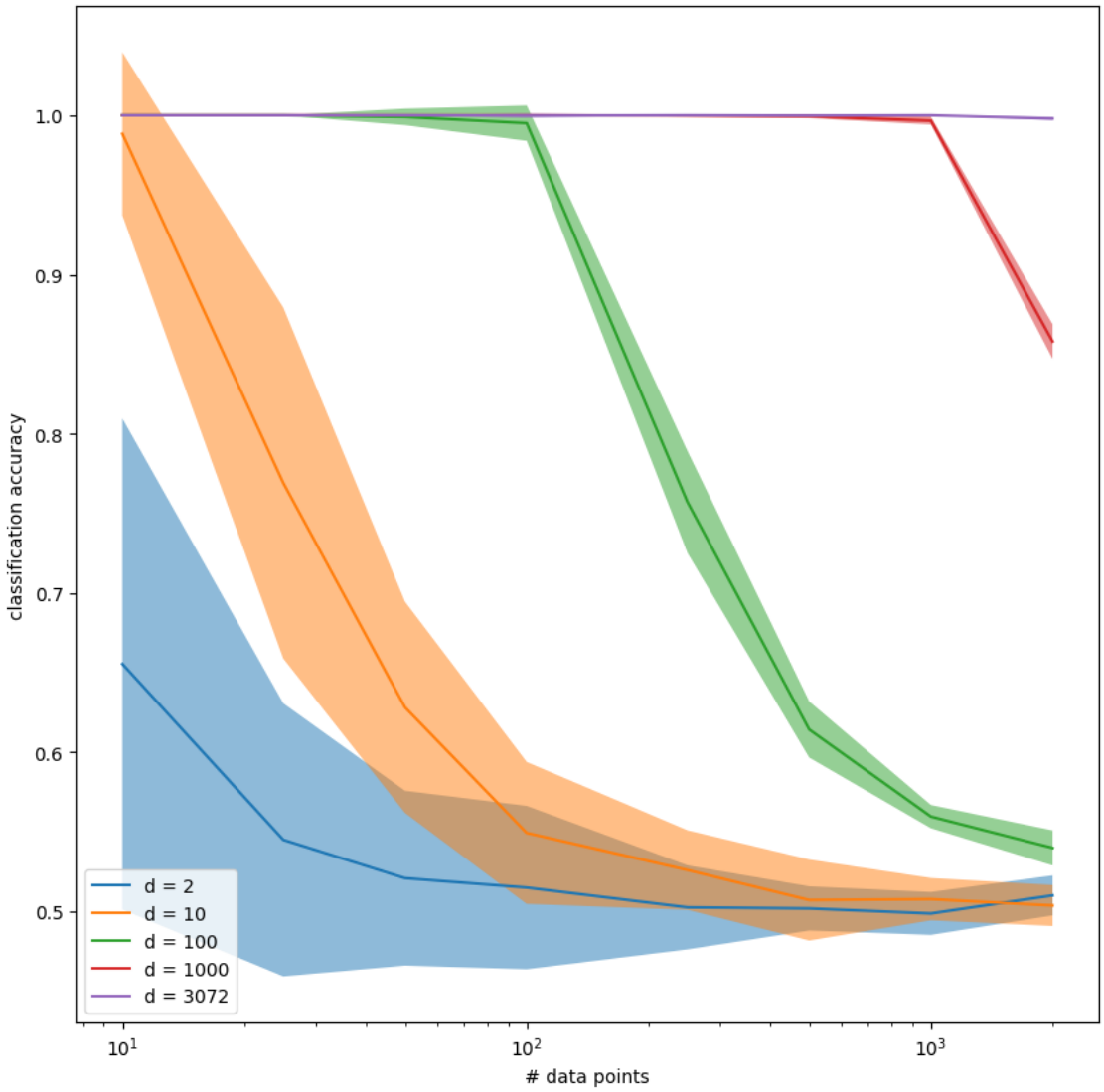
[3]:
```
separate_points(4000, 1000)
```

0.81525



[4]:
```
def plot_sep(d=2, N=[10,25,50,100,250,500,1000,2000]):
    S = []
    mnS, mxS = [], []
```

```
    for n in N:
        s = []
        for it in range(5000 // n + 1):
            X = np.random.normal(0,1,size=(n, d))
            y = np.random.rand(n)
            y = y > np.median(y)

            # Fit the first classifier
            s.append( cls.fit(X, y).score(X,y) )
        S.append(np.mean(s))
#         mnS.append(np.min(s))
#         mxS.append(np.max(s))
        mnS.append(np.mean(s) - np.std(s))
        mxS.append(np.mean(s) + np.std(s))
    fill_between(N, mnS, mxS, alpha=0.5)
    plot(N,S, label='d = %d'%d)
    xlabel('# data points')
    ylabel('classification accuracy')
    xscale('log')
    legend()

figure(figsize=(10,10))
plot_sep(2)
plot_sep(10)
plot_sep(100)
plot_sep(1000)
plot_sep(32*32*3)
```




[5]:
```
224*224*3
```

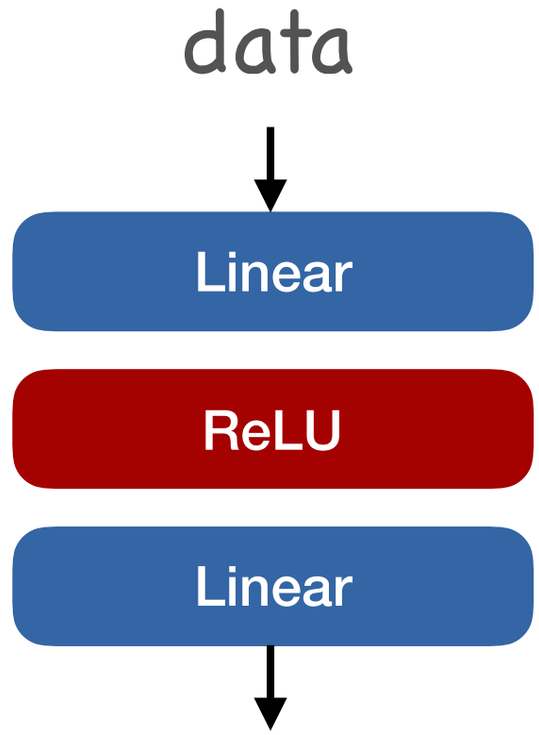
[5]: 150528

[ ]:
```

```

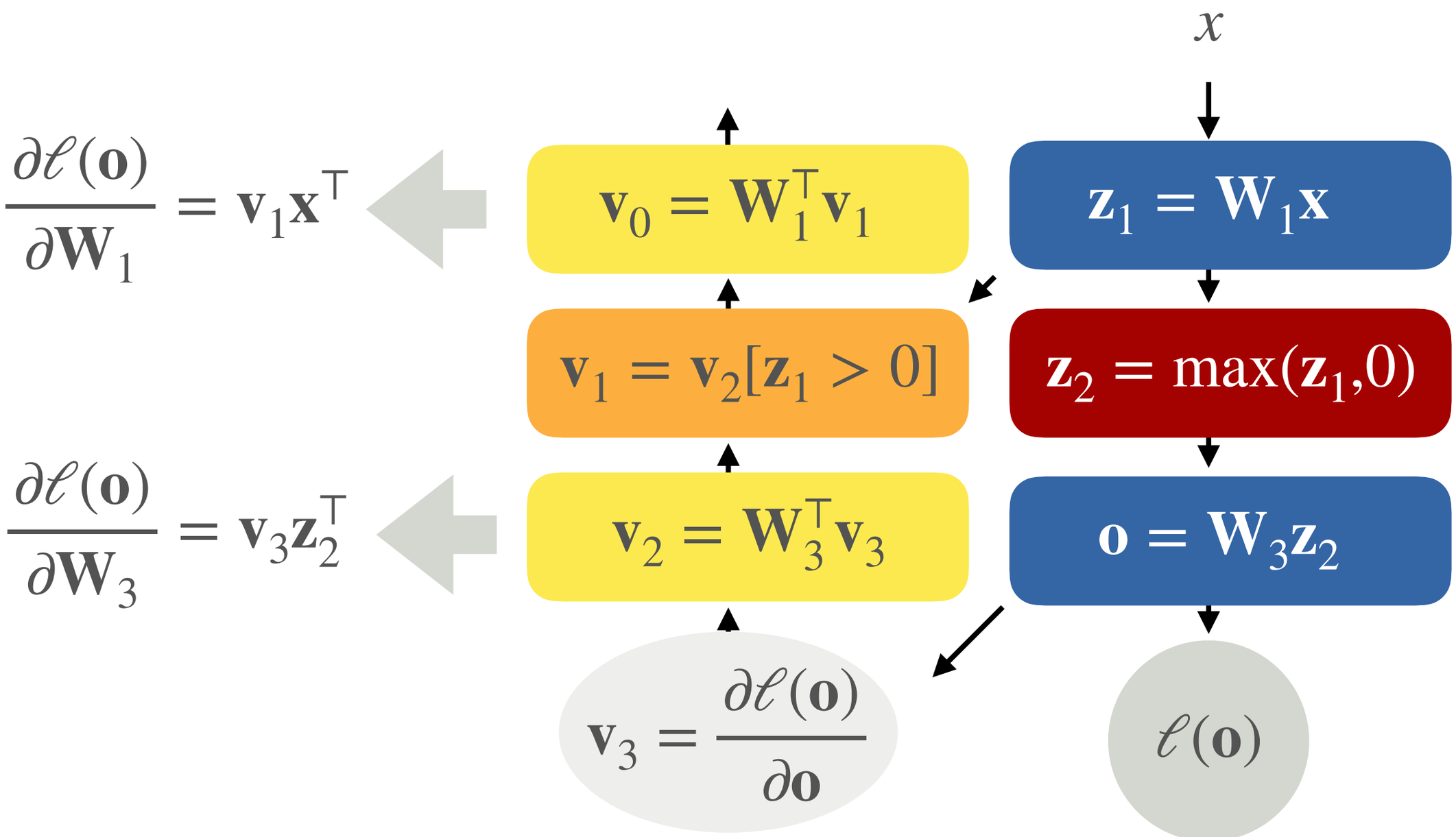
# Network initialization

# Network initialization

- What do we set the initial parameters to?



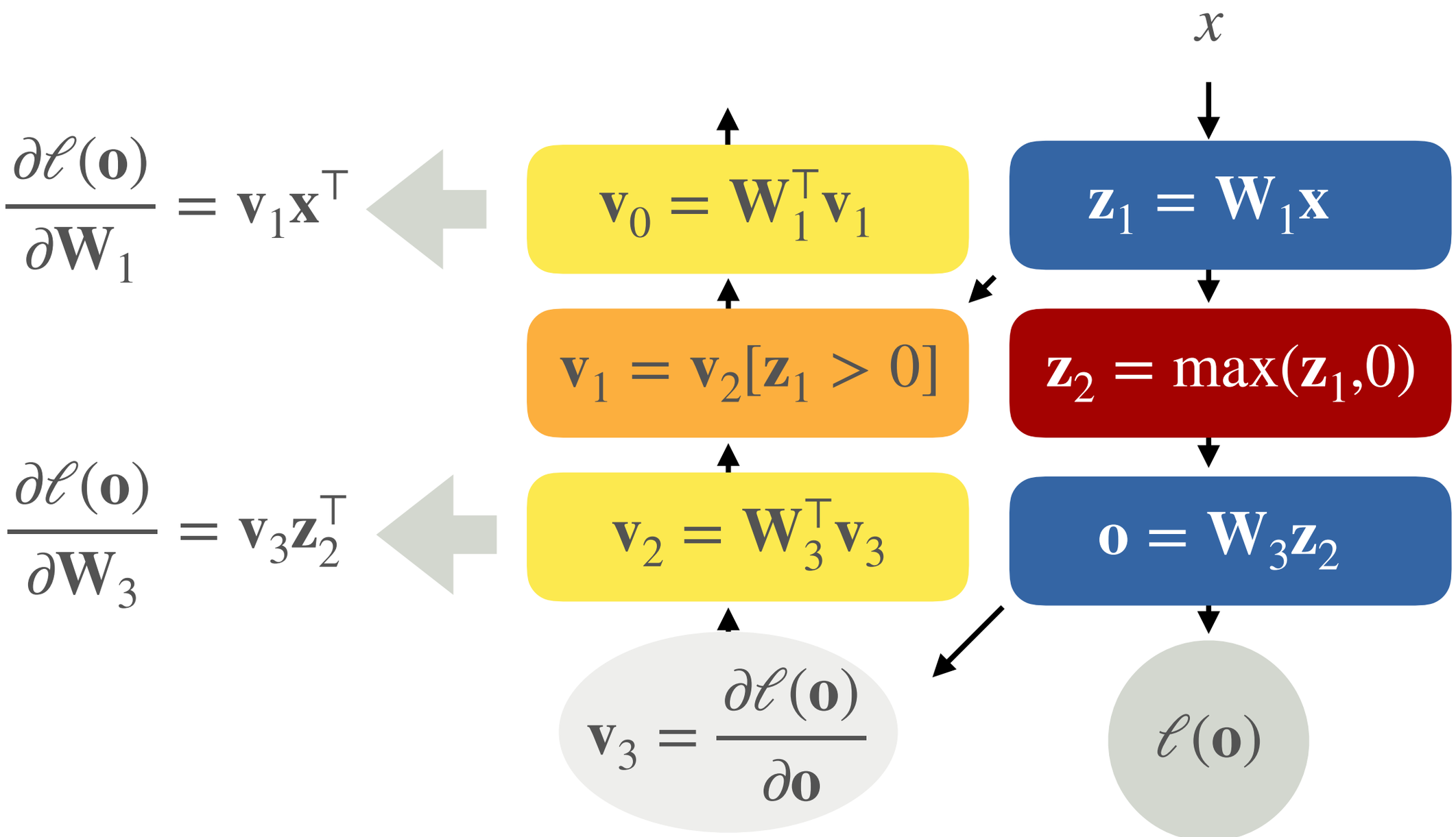
# Idea 1: All zero



# Idea 1: All zero

- Does not work
  - No gradient
- Saddle point

# Idea 2: constant



# Idea 2: constant

- Does not break symmetries

# Solution

- Random initialization

$$\mathbf{W}_i = \mathcal{N}(\mu_i, \sigma_i^2 \mathbf{I})$$

# Random initialization

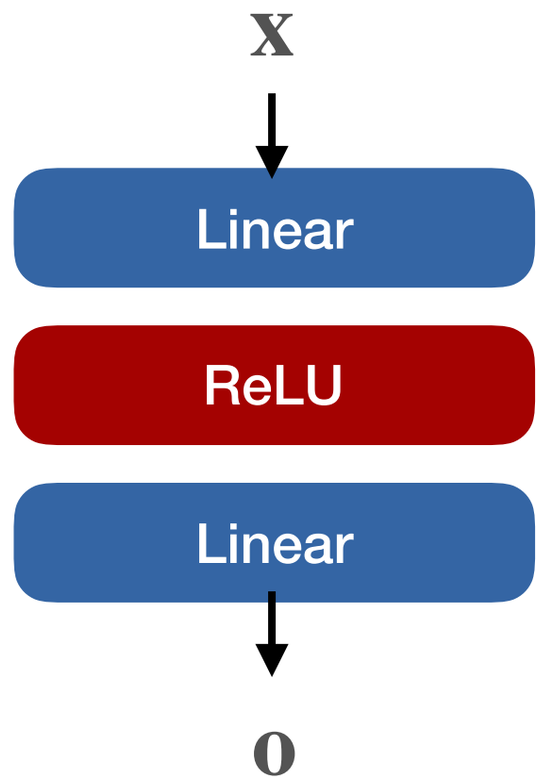
# Random initialization

- Initialize weights
  - Normal distribution
  - Uniform distribution

$$\mathbf{W}_1 = \mathcal{N}(\mu_1, \sigma_1^2\mathbf{I})$$

- What should $\mu_i$ and $\sigma_i$ be?
  - For simplicity $\mu_i$ = 0 and bias = 0

$$\mathbf{W}_3 = \mathcal{N}(\mu_3, \sigma_3^2\mathbf{I})$$

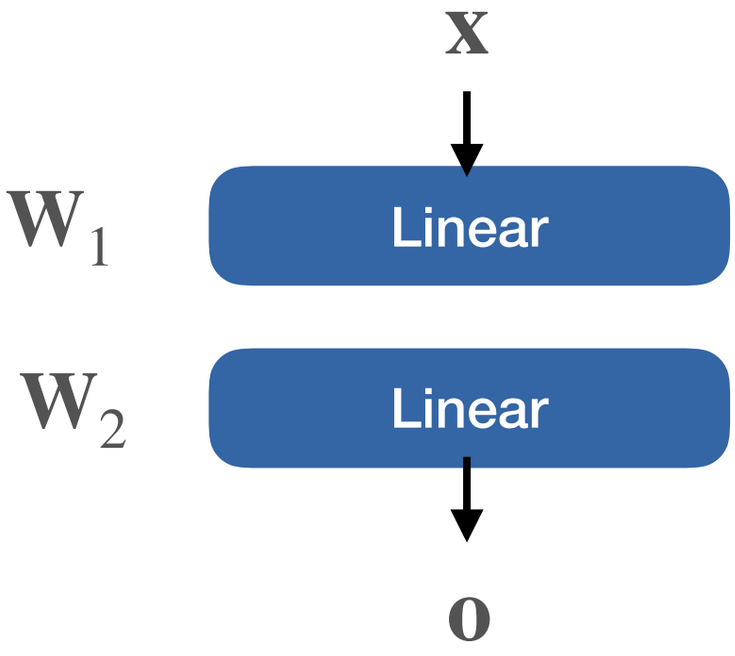


# Scaling matters

$$\mathbf{o} = \mathbf{W}_2\mathbf{W}_1\mathbf{x}$$

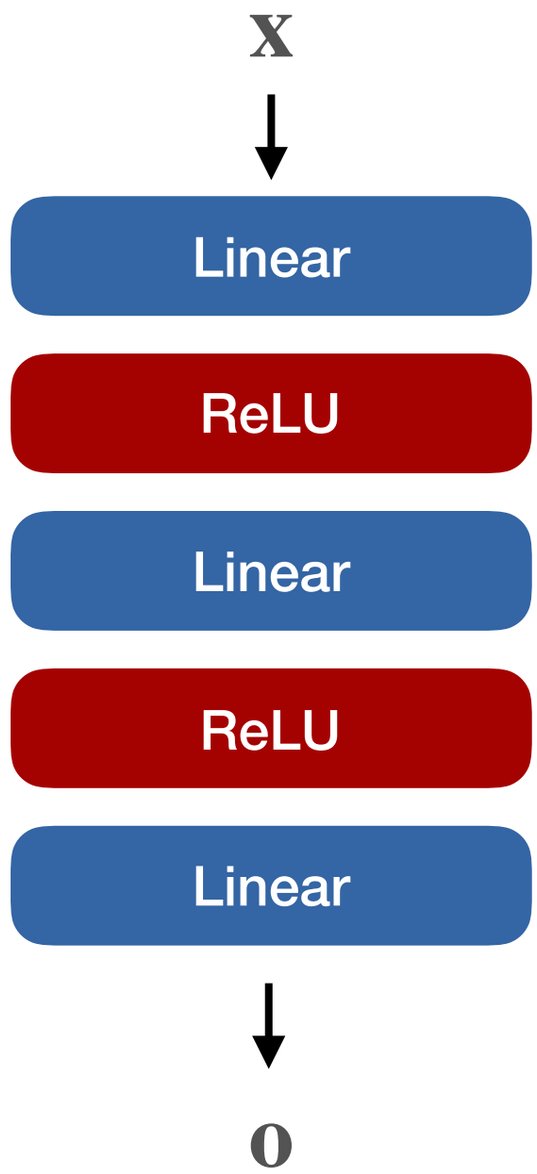
$$\frac{\partial \ell(\mathbf{o})}{\partial \mathbf{W}_1} = \left(\mathbf{W}_2^\top \frac{\partial \ell(\mathbf{o})}{\partial \mathbf{o}}\right) \mathbf{x}^\top$$

$$\frac{\partial \ell(\mathbf{o})}{\partial \mathbf{W}_2} = \frac{\partial \ell(\mathbf{o})}{\partial \mathbf{o}} \left(\mathbf{W}_1\mathbf{x}\right)^\top$$



# How do we scale the initialization?

- By hand
  - A lot of tuning
- Automatically
  - A lot of math



# Xavier and Kaiming initialization

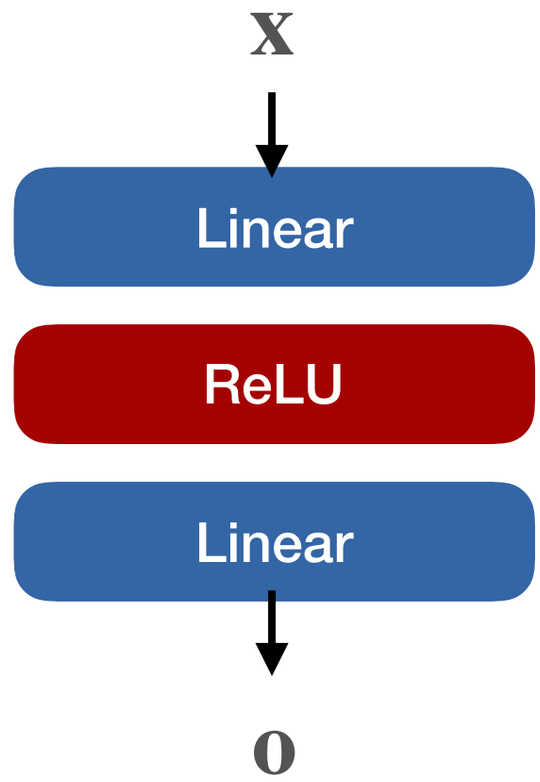
# Xavier and Kaiming initialization

- Strategy to set variance $\sigma^2$ of Normal initialization

- All activations are of similar scale

$$\mathbf{W}_1 \sim \mathcal{N}(\mu_1, \sigma_1^2 \mathbf{I})$$

$$\mathbf{W}_3 \sim \mathcal{N}(\mu_3, \sigma_3^2 \mathbf{I})$$



# Random matrix multiplication

$$\mathbf{a}^\top \mathbf{x} \sim \mathcal{N}(\mu_a \sum_i \mathbf{x}_i, \|\mathbf{x}\|^2 \sigma_a^2) \quad \text{for} \quad \mathbf{a} \sim \mathcal{N}(\mu_a, \sigma_a^2 \mathbf{I})$$ [1]

[1] for derivation see: https://en.m.wikipedia.org/wiki/Multivariate_normal_distribution

# Random matrix multiplication

$$\mathbf{z}_i = \mathbf{W}_{i-1}\mathbf{z}_{i-1} \sim \mathcal{N}(0,\ \|\mathbf{z}_{i-1}\|^2\sigma^2_{W_{i-1}}\mathbf{I}) \quad \text{for} \quad \mathbf{W}_{i-1} \sim \mathcal{N}(0,\ \sigma^2_{W_{i-1}}\mathbf{I})$$

# Random ReLU

$$\mathbf{z}_{i+1} = \max(\mathbf{z}_i, 0) \qquad \text{for} \qquad \mathbf{z}_i \sim \mathcal{N}(0, \sigma_i^2 \mathbf{I})$$

$$\mathbb{E}[\|\mathbf{z}_{i+1}\|^2] = \frac{1}{2} n_{\mathbf{z}_i} \sigma_i^2$$

# Putting things together

$$\mathbf{z}_i \sim \mathcal{N}(0,\, \sigma_i^2 \mathbf{I})$$

$$\|\mathbf{z}_{i+1}\|^2 \approx \mathbb{E}[\|\mathbf{z}_{i+1}\|^2] = \frac{1}{2} n_{\mathbf{z}_i} \sigma_i^2$$

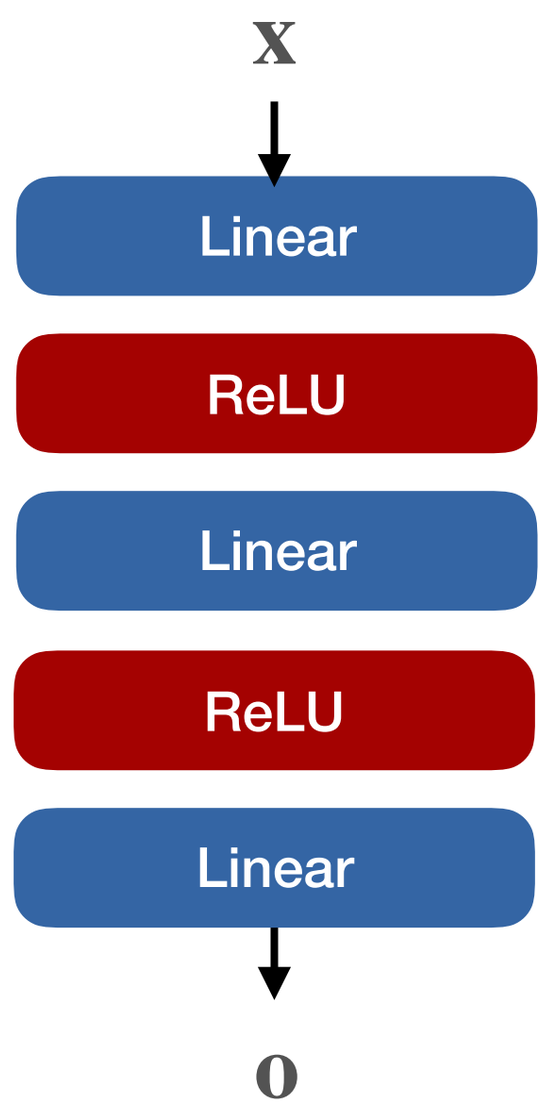
$$\mathbf{z}_{i+2} \sim \mathcal{N}(0,\ \underbrace{\|\mathbf{z}_{\mathbf{i+1}}\|^2 \sigma_{W_{i+1}}^2}_{\sigma_{i+2}^2} \mathbf{I})$$

$$\sigma_{i+2} = \frac{1}{\sqrt{2}} \sigma_{W_{i+1}} \sigma_i \sqrt{n_{\mathbf{z}_i}}$$

$$\sigma_i = \prod_{k=0}^{(i-1)/2} \left( \frac{1}{\sqrt{2}} \sigma_{W_{2k+1}} \sqrt{n_{\mathbf{z}_{2k}}} \right) \sigma_x$$

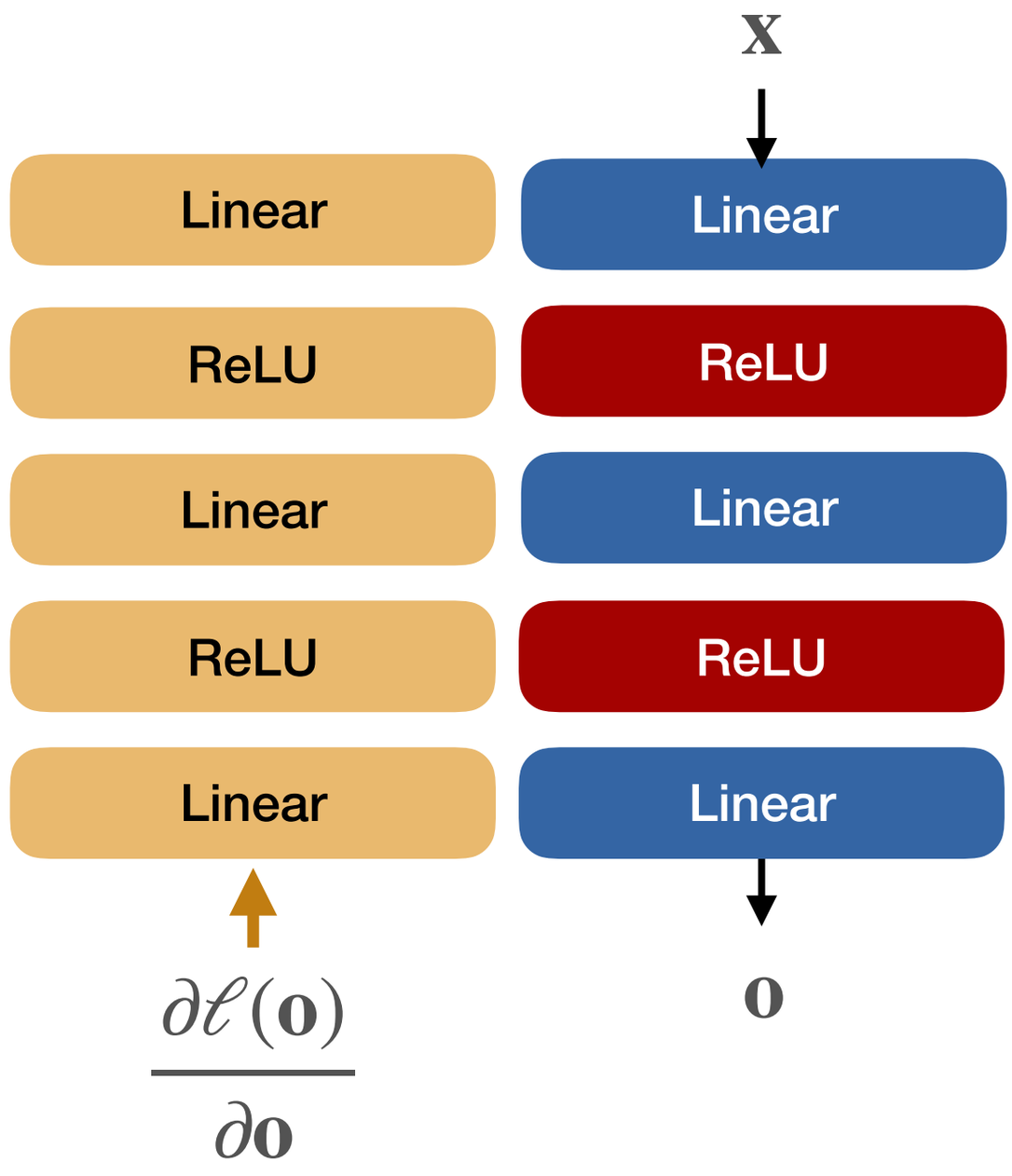
# Randomly initialized network

$$\sigma_i = \prod_{k=0}^{(i-1)/2} \left( \frac{1}{\sqrt{2}} \sigma_{W_{2k+1}} \sqrt{n_{\mathbf{z}_{2k}}} \right) \sigma_x$$



# Variance of back-propagation graph

$$\hat{\sigma}_i = \prod_{k=i/2}^{(N-1)/2} \left( \frac{1}{\sqrt{2}} \sigma_{W_{2k+1}} \sqrt{n_{\mathbf{z}_{2k+2}}} \right) \hat{\sigma}_N$$



# Xavier initialization

$$\sigma_i = \prod_{k=0}^{(i-1)/2} \left( \frac{1}{\sqrt{2}} \sigma_{W_{2k+1}} \sqrt{n_{\mathbf{z}_{2k}}} \right) \sigma_x$$

- Try to keep both activations and gradient magnitude constant

$$\hat{\sigma}_i = \prod_{k=i/2}^{(N-1)/2} \left( \frac{1}{\sqrt{2}} \sigma_{W_{2k+1}} \sqrt{n_{\mathbf{z}_{2k+2}}} \right) \hat{\sigma}_N$$

- $\sigma_W = \sqrt{2} \sqrt{\frac{2}{n_{\mathbf{z}_i} + n_{\mathbf{z}_{i+1}}}}$

# Kaiming initialization

- Try to keep either activation or gradient magnitude constant

$$\sigma_i = \prod_{k=0}^{(i-1)/2} \left( \frac{1}{\sqrt{2}} \sigma_{W_{2k+1}} \sqrt{n_{\mathbf{z}_{2k}}} \right) \sigma_x$$

$$\hat{\sigma}_i = \prod_{k=i/2}^{(N-1)/2} \left( \frac{1}{\sqrt{2}} \sigma_{W_{2k+1}} \sqrt{n_{\mathbf{z}_{2k+2}}} \right) \hat{\sigma}_N$$

- $\sigma_W = \sqrt{2} \frac{1}{\sqrt{n_{\mathbf{z}_i}}}$

- $\sigma_W = \sqrt{2} \frac{1}{\sqrt{n_{\mathbf{z}_{i+1}}}}$

# Initialization in practice

- Xavier (default) is often good enough
- Initialize last layer to zero

# 07

January 23, 2024

[1]:
```
import torch
```

[2]:
```
class ConvNet(torch.nn.Module):
    class Block(torch.nn.Module):
        def __init__(self, n_input, n_output, stride=1):
            super().__init__()
            self.net = torch.nn.Sequential(
              torch.nn.Conv2d(n_input, n_output, kernel_size=3, padding=1,␣
 ↪stride=stride),
              torch.nn.ReLU(),
              torch.nn.Conv2d(n_output, n_output, kernel_size=3, padding=1),
              torch.nn.ReLU()
            )
            torch.nn.init.xavier_normal_(self.net[0].weight)
            torch.nn.init.constant_(self.net[0].bias, 0.1)

        def forward(self, x):
            return self.net(x)

    def __init__(self, layers=[32,64,128], n_input_channels=3):
        super().__init__()
        L = [torch.nn.Conv2d(n_input_channels, 32, kernel_size=7, padding=3,␣
 ↪stride=2),
             torch.nn.ReLU(),
             torch.nn.MaxPool2d(kernel_size=3, stride=2, padding=1)]
        c = 32
        for l in layers:
            L.append(self.Block(c, l, stride=2))
            c = l
        self.network = torch.nn.Sequential(*L)
        self.classifier = torch.nn.Linear(c, 1)

        # Initialize the weights
        torch.nn.init.zeros_(self.classifier.weight)
        torch.nn.init.xavier_normal_(self.network[0].weight)
        torch.nn.init.constant_(self.network[0].bias, 0.1)
```

```
    def forward(self, x):
        # Compute the features
        z = self.network(x)
        # Global average pooling
        z = z.mean(dim=[2,3])
        # Classify
        return self.classifier(z)[:,0]
```

[3]:
```
net = ConvNet()
print(net.network[0].weight, net.network[0].bias)
print()
print(net.classifier.weight, net.classifier.bias)
```

```
Parameter containing:
tensor([[[[ 1.7270e-02, -7.3183e-04, -3.1007e-02,  …,  7.2382e-02,
           -5.6650e-02,  3.4035e-02],
          [ 3.4279e-02, -1.5811e-02, -4.0903e-02,  …, -3.3977e-02,
            1.0024e-02,  5.9173e-02],
          [ 3.1137e-02,  1.8381e-02, -4.6479e-02,  …, -3.4519e-02,
            3.6153e-02,  5.8218e-02],
          …,
          [ 3.8489e-02,  7.1153e-03, -1.7250e-03,  …,  2.4888e-02,
           -1.6550e-02, -5.4573e-04],
          [ 3.4834e-02, -3.4391e-02, -5.1466e-02,  …, -4.7134e-02,
            8.0856e-02,  6.8783e-02],
          [ 1.4183e-02, -5.3272e-02,  2.7214e-03,  …,  4.6807e-02,
            6.0609e-02, -4.9352e-02]],

         [[-3.0341e-02, -6.1002e-02, -2.6696e-02,  …,  1.3062e-02,
            1.6921e-02, -1.7774e-02],
          [ 2.1336e-02, -4.2838e-02,  1.3234e-02,  …, -7.4801e-03,
           -2.2554e-02, -1.7629e-02],
          [-4.9333e-03, -2.8675e-02, -1.5096e-02,  …, -9.5340e-02,
           -3.4195e-02,  2.3809e-02],
          …,
          [ 3.1219e-02,  1.5745e-02,  1.8433e-02,  …, -1.8063e-02,
            2.0708e-02,  1.0663e-02],
          [ 5.7711e-02,  2.4035e-02,  9.3086e-02,  …, -4.5895e-02,
           -6.3147e-03, -1.3131e-04],
          [-3.4656e-02, -1.7938e-03,  6.3285e-03,  …,  5.0900e-02,
            3.8086e-02, -4.9981e-03]],

         [[-5.7595e-02, -2.1504e-02,  4.8310e-04,  …, -5.5216e-03,
           -3.3110e-03,  3.4074e-02],
          [ 3.9680e-02,  3.7819e-02, -3.8671e-02,  …,  5.2865e-02,
            1.9275e-02,  1.3183e-04],
          [-9.3193e-03, -3.3895e-02, -1.3461e-02,  …,  7.6623e-03,
```

```
    4.1645e-02,  2.7737e-03],
  …,
  [-6.5633e-03, -1.6800e-02, -3.5289e-03,  …,  1.4955e-02,
    1.1166e-02,  1.4390e-02],
  [ 5.0660e-02,  5.2829e-02, -4.4883e-02,  …, -9.9083e-03,
    2.5047e-02,  6.5250e-03],
  [-8.2853e-02,  2.0762e-02, -2.3080e-02,  …,  1.8657e-02,
   -4.0069e-02,  5.6833e-02]]],


[[[ 3.0312e-02,  1.9735e-02, -8.3784e-02,  …,  6.7218e-03,
    1.0337e-02, -2.0585e-02],
  [-5.8684e-03,  4.4530e-02, -1.9908e-02,  …,  5.2928e-02,
   -3.9958e-02,  3.7817e-02],
  [ 2.7240e-03, -4.6326e-02, -2.6096e-02,  …, -9.5495e-03,
    2.2732e-02,  4.7033e-02],
  …,
  [ 4.7201e-02, -2.0678e-02,  6.4564e-03,  …, -3.6370e-03,
   -1.6052e-02,  5.2273e-02],
  [-7.8522e-03, -5.0071e-02,  6.7837e-02,  …, -6.8882e-03,
   -1.5945e-02, -2.9675e-03],
  [ 8.1658e-03, -4.1443e-02, -7.6495e-02,  …, -2.9860e-04,
    8.8739e-03,  4.1826e-02]],

 [[ 6.1974e-02,  2.8197e-03,  3.2834e-02,  …, -2.6804e-02,
   -7.3290e-03,  1.1866e-02],
  [-3.7422e-02,  2.5091e-02, -1.4149e-02,  …,  1.0261e-02,
    3.2109e-02,  3.0673e-02],
  [-7.2544e-02, -1.3210e-02, -2.8296e-02,  …, -3.2223e-02,
   -3.4223e-02, -2.7663e-03],
  …,
  [ 2.5300e-02,  1.2554e-02,  1.0254e-02,  …, -3.4622e-02,
    9.3363e-03, -3.3402e-03],
  [ 6.1115e-02, -2.8740e-02,  5.0647e-02,  …,  3.3570e-02,
   -3.7666e-02,  3.1693e-02],
  [ 9.9463e-03,  2.2477e-02, -2.1866e-03,  …,  8.5332e-03,
    5.3147e-02,  2.5073e-02]],

 [[-7.1589e-02, -6.2781e-03, -2.2707e-02,  …, -7.1620e-03,
   -5.6453e-02,  2.0606e-03],
  [ 7.0756e-03,  3.0717e-02, -1.0585e-02,  …,  2.0466e-02,
   -1.2539e-02,  6.0194e-02],
  [ 1.7780e-02,  3.7930e-02,  3.0591e-02,  …,  3.8881e-02,
    3.5865e-02,  2.0826e-02],
  …,
  [-4.3162e-02, -3.2457e-02, -2.3963e-03,  …,  1.7215e-02,
   -3.0654e-02,  6.1037e-02],
  [-2.2126e-02, -1.4697e-03,  1.8692e-02,  …,  5.8771e-03,
```

```
    5.1038e-02,  6.9203e-02],
  [-4.7190e-02,  7.4458e-02, -2.2988e-02,  …, -3.9607e-02,
   -6.4576e-03,  1.0107e-02]]],


[[[-1.0469e-02, -1.7985e-02,  8.3147e-03,  …, -4.1685e-02,
   -7.2816e-03,  5.2073e-02],
  [ 2.0040e-02, -2.8831e-02, -3.9398e-02,  …,  4.6675e-02,
    4.0492e-03, -6.2253e-03],
  [ 5.7879e-03, -5.0133e-02, -5.7173e-04,  …,  3.3450e-03,
    2.1345e-02, -1.7006e-03],
  …,
  [ 6.8814e-03, -1.2141e-02,  3.0120e-02,  …, -8.7471e-03,
    2.6608e-02, -3.0677e-03],
  [-3.1705e-02, -3.8473e-02,  2.1697e-02,  …, -1.5960e-02,
   -5.2818e-02, -3.8368e-03],
  [-4.9731e-02, -8.2384e-03, -8.2883e-04,  …, -9.9182e-03,
   -3.4472e-02,  9.5803e-03]],

 [[-3.4683e-02,  2.1258e-04,  3.9984e-03,  …, -2.3746e-02,
    2.0702e-02, -3.4651e-02],
  [ 4.9876e-02,  2.3990e-03,  3.9380e-02,  …, -2.0188e-03,
   -2.8576e-02,  4.9345e-02],
  [ 1.8118e-02, -1.1511e-03,  8.3344e-03,  …, -1.4987e-02,
    5.5184e-02, -4.8711e-02],
  …,
  [ 2.5114e-02, -2.2418e-02, -1.4019e-02,  …, -3.9997e-03,
    4.1805e-02,  3.5255e-02],
  [-3.6014e-02, -6.8018e-04,  1.6488e-02,  …, -3.8227e-02,
   -3.2243e-02, -7.1064e-02],
  [ 5.9795e-02,  7.5698e-02,  2.7621e-02,  …,  2.5830e-02,
   -2.3608e-03,  2.2370e-02]],

 [[ 1.3785e-02, -6.2701e-03,  2.5712e-02,  …, -1.2512e-02,
    1.9808e-03, -5.2850e-02],
  [-4.0397e-03,  9.0568e-03, -9.0059e-02,  …, -1.4699e-02,
    5.1221e-02, -2.6680e-02],
  [-1.3011e-02, -4.3516e-02, -1.9160e-02,  …,  2.3676e-02,
    2.3806e-02, -3.6848e-02],
  …,
  [ 4.1114e-02,  5.0811e-02, -3.1632e-02,  …, -4.7123e-02,
   -9.7130e-03, -8.8099e-03],
  [-3.9926e-02, -3.0552e-02, -5.3696e-02,  …,  6.7025e-03,
   -2.2567e-02, -8.7240e-04],
  [ 8.1035e-03,  5.1553e-03,  2.2702e-02,  …, -2.4104e-02,
   -2.5652e-03, -7.1151e-02]]],
```

```
…,


[[[ 4.2128e-04, -3.6085e-02,  1.0412e-02,  …,  4.9810e-02,
   -1.2295e-02, -7.8702e-03],
  [-1.9585e-03, -9.1903e-03,  1.5473e-02,  …, -1.3587e-02,
   -5.4549e-02,  1.7739e-02],
  [ 3.3826e-02,  6.8026e-03, -1.0630e-02,  …,  2.0466e-02,
    3.4522e-02,  1.3296e-02],
  …,
  [-1.6260e-03,  6.6798e-04, -2.2252e-02,  …, -3.2502e-02,
   -2.8506e-02, -2.6194e-03],
  [ 1.0688e-01, -3.3566e-02, -2.2241e-02,  …, -1.6185e-02,
   -6.1541e-03,  7.2378e-04],
  [ 1.2457e-02,  8.0554e-03,  3.3645e-02,  …, -3.2195e-02,
   -7.4916e-03, -1.1952e-02]],

 [[ 3.1327e-02,  1.8129e-02,  3.6721e-03,  …,  7.7060e-03,
   -1.7012e-02,  4.8286e-02],
  [ 2.4806e-02,  6.8878e-02, -6.1908e-02,  …, -1.5746e-02,
   -1.2506e-02,  5.0871e-02],
  [ 2.0487e-02, -3.8297e-03,  1.9064e-02,  …, -2.9599e-02,
   -4.5969e-02, -7.6089e-02],
  …,
  [ 6.3154e-02, -1.0776e-03, -4.2670e-02,  …,  7.5058e-03,
   -5.9743e-02,  1.5269e-02],
  [-1.2461e-02,  3.6127e-02, -1.9113e-02,  …, -4.0617e-02,
    3.4519e-02, -2.6409e-02],
  [-1.4306e-02,  1.2585e-03,  3.1119e-02,  …,  3.8380e-02,
    1.5414e-02,  6.8117e-03]],

 [[ 1.0122e-02, -2.6863e-02,  2.0506e-02,  …,  4.2748e-02,
    1.2057e-02,  1.4551e-02],
  [-1.1920e-02,  5.0190e-03,  7.3298e-03,  …,  8.0210e-03,
   -3.0813e-02, -2.7648e-02],
  [-9.3097e-03,  5.8606e-02, -9.6205e-02,  …, -2.1899e-02,
   -7.8621e-02,  1.0599e-04],
  …,
  [ 6.8920e-03, -9.4265e-03, -2.7177e-02,  …,  1.9282e-02,
    9.0126e-02, -8.5844e-03],
  [ 1.0911e-02, -2.0499e-02,  6.8522e-02,  …, -5.0500e-02,
    5.4342e-02,  4.7019e-02],
  [ 5.5456e-02,  2.2103e-03,  7.0158e-02,  …,  4.0282e-02,
   -9.2080e-03,  2.0980e-02]]],


[[[ 7.8945e-03,  6.3346e-02, -3.8348e-02,  …,  1.8074e-02,
    6.7733e-03, -7.4155e-02],
```

```
  [-4.0311e-02,  7.5571e-02,  2.0998e-02,  …, -1.5845e-02,
   -1.7667e-02, -2.4792e-02],
  [ 1.1830e-02,  5.1968e-02,  4.3802e-02,  …, -6.1078e-02,
   -7.3165e-02, -4.8460e-02],
  …,
  [-4.7456e-02, -1.7781e-02, -2.8179e-02,  …, -1.0197e-02,
   -2.6407e-02,  8.3755e-03],
  [ 3.4731e-02,  1.9920e-03, -1.7169e-03,  …,  9.6292e-03,
    4.2919e-02,  7.4485e-03],
  [-1.1433e-02,  1.0171e-02, -5.7195e-02,  …,  3.8149e-02,
   -5.8774e-04, -4.0581e-03]],

 [[ 5.8451e-03,  1.3633e-02, -4.6927e-03,  …, -5.1043e-02,
   -3.6188e-02, -3.4600e-02],
  [ 2.3242e-02,  3.5638e-02,  1.6548e-02,  …,  1.1957e-02,
    8.4894e-03, -2.2695e-02],
  [-1.1045e-02, -3.1804e-02, -1.3691e-02,  …, -2.1453e-02,
    4.5871e-02, -1.0960e-02],
  …,
  [-2.5247e-02, -2.1553e-02,  2.4125e-02,  …,  8.3183e-02,
    2.1369e-02, -1.8380e-02],
  [ 9.3596e-03,  1.6879e-02,  1.8336e-02,  …, -7.1493e-02,
    3.5498e-02,  5.5757e-02],
  [-1.5180e-02, -8.7149e-03,  3.3943e-02,  …, -6.2982e-03,
   -1.5851e-02,  2.7976e-03]],

 [[ 2.5917e-02,  5.0543e-02, -2.8627e-02,  …, -4.9950e-02,
   -3.2251e-02, -1.1888e-02],
  [ 2.0599e-02, -8.3955e-02, -4.4174e-02,  …, -3.9893e-02,
    4.1910e-02,  3.5731e-02],
  [ 2.3033e-02,  2.1210e-02, -2.6593e-02,  …, -4.9481e-02,
   -6.9238e-03, -3.4365e-02],
  …,
  [ 4.7847e-02,  1.2076e-02, -4.1541e-03,  …,  1.6007e-02,
   -3.5096e-03,  3.7028e-02],
  [-1.2383e-03,  6.1084e-02, -4.5421e-02,  …, -8.9091e-03,
    3.0989e-02,  6.4288e-02],
  [ 2.2258e-02, -1.4001e-02, -4.4644e-03,  …, -2.3183e-02,
   -4.6317e-02,  8.5237e-03]]],


[[[ 1.6528e-03, -1.2499e-02,  3.0952e-02,  …,  3.1003e-02,
   -6.1820e-02, -8.0819e-02],
  [-2.0791e-02,  4.1725e-03, -3.9585e-02,  …, -2.6372e-03,
    2.8532e-02, -2.4089e-05],
  [-1.2268e-02, -4.8644e-03, -4.2005e-02,  …, -4.6175e-03,
    1.4548e-02,  5.8493e-02],
  …,
```

```
           [ 3.8000e-02, -2.7958e-03, -1.6598e-02,  …,  5.8346e-02,
             1.2790e-02,  1.2083e-02],
           [ 5.4728e-03, -2.9331e-03,  4.9359e-03,  …, -9.7022e-03,
             2.5282e-02, -5.5388e-03],
           [ 4.4741e-02, -5.7182e-02,  1.4514e-02,  …,  1.0064e-01,
            -4.7624e-02, -4.5235e-02]],

          [[-3.8993e-02,  1.5706e-02, -1.1157e-02,  …, -9.2559e-02,
            -6.1555e-03,  1.2815e-02],
           [ 1.7850e-02, -4.6040e-03, -2.3204e-02,  …, -1.5972e-02,
             4.9309e-02, -3.3457e-02],
           [ 9.6365e-03,  1.3227e-02,  4.2185e-02,  …, -2.4377e-02,
            -2.0152e-02,  1.1091e-02],
           …,
           [-1.3281e-02,  2.6287e-03, -9.2315e-04,  …,  2.7528e-02,
             1.3573e-02,  6.2306e-02],
           [ 3.2790e-02,  4.4430e-03,  2.7772e-02,  …,  4.6722e-02,
             6.3781e-02,  2.8058e-02],
           [ 1.5616e-02, -3.7544e-02,  4.2796e-02,  …,  4.6488e-02,
            -2.0198e-02, -1.6001e-02]],

          [[ 4.9031e-02,  2.2490e-03, -1.8706e-02,  …,  4.6293e-02,
            -1.7358e-02,  8.7689e-03],
           [ 7.1945e-02,  7.9511e-03, -2.8067e-02,  …, -3.1480e-02,
             1.1430e-02, -5.5074e-02],
           [-8.2491e-03,  6.3296e-03,  4.7554e-02,  …,  2.7065e-02,
             2.8531e-02, -1.8962e-02],
           …,
           [-4.0945e-02, -6.9874e-02,  8.3975e-02,  …, -9.2623e-03,
            -6.0930e-03,  1.3908e-02],
           [-3.4397e-02, -6.5461e-02, -1.8450e-02,  …,  1.6276e-02,
            -2.3134e-02,  3.9372e-02],
           [-2.2640e-02,  1.9279e-02, -6.3605e-02,  …, -1.7532e-02,
            -6.1585e-03,  1.3110e-02]]]], requires_grad=True) Parameter
containing:
tensor([0.1000, 0.1000, 0.1000, 0.1000, 0.1000, 0.1000, 0.1000, 0.1000, 0.1000,
        0.1000, 0.1000, 0.1000, 0.1000, 0.1000, 0.1000, 0.1000, 0.1000, 0.1000,
        0.1000, 0.1000, 0.1000, 0.1000, 0.1000, 0.1000, 0.1000, 0.1000, 0.1000,
        0.1000, 0.1000, 0.1000, 0.1000, 0.1000], requires_grad=True)

Parameter containing:
tensor([[0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
0., 0., 0., 0., 0., 0.,
         0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
0., 0., 0., 0., 0., 0.,
         0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
0., 0., 0., 0., 0., 0.,
         0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
```

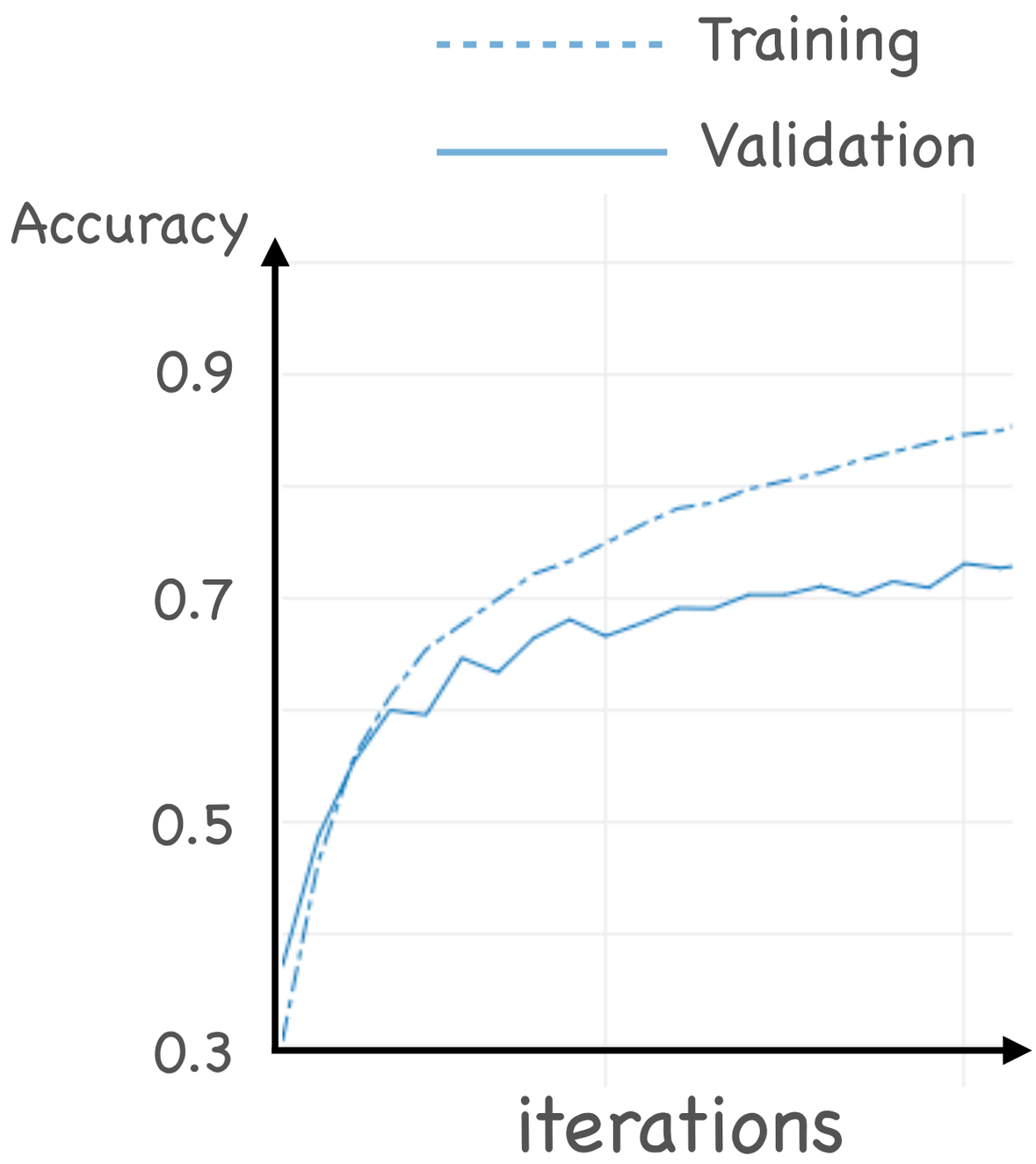
```
0., 0., 0., 0., 0., 0.,
         0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
0., 0., 0., 0., 0., 0.,
         0., 0., 0., 0., 0., 0., 0., 0.]], requires_grad=True) Parameter
containing:
tensor([-0.0601], requires_grad=True)
```

[ ]:

# Optimization

# Optimization

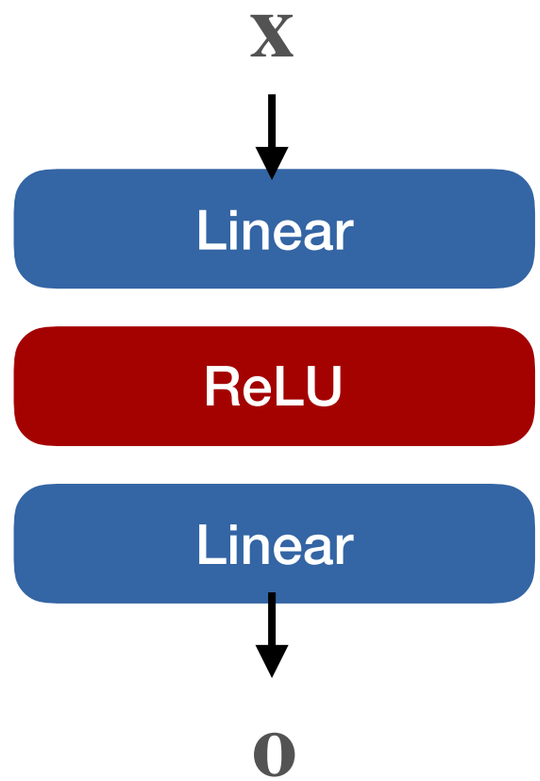
- Stochastic Gradient Descent
- Convergence speed
- Training accuracy
- Generalization performance



# Input normalization

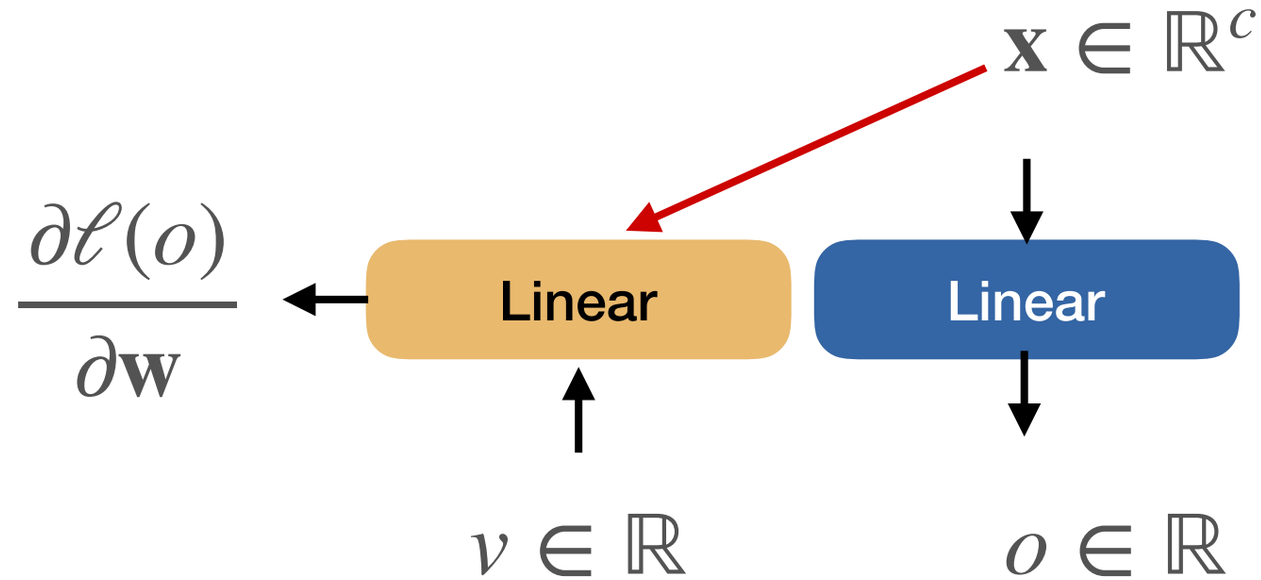
# Input normalization

- Input: $\mathbf{x}_i$
- Apply affine transformation $\hat{\mathbf{x}}_i = \alpha \mathbf{x}_i + \beta$



# Gradients of uncentered inputs: A simple example

- Input vector $\mathbf{x}$
- Output scalar $o$
- $\dfrac{\partial \ell(o)}{\partial \mathbf{w}} = v\mathbf{x}^\top$



# Mean subtraction

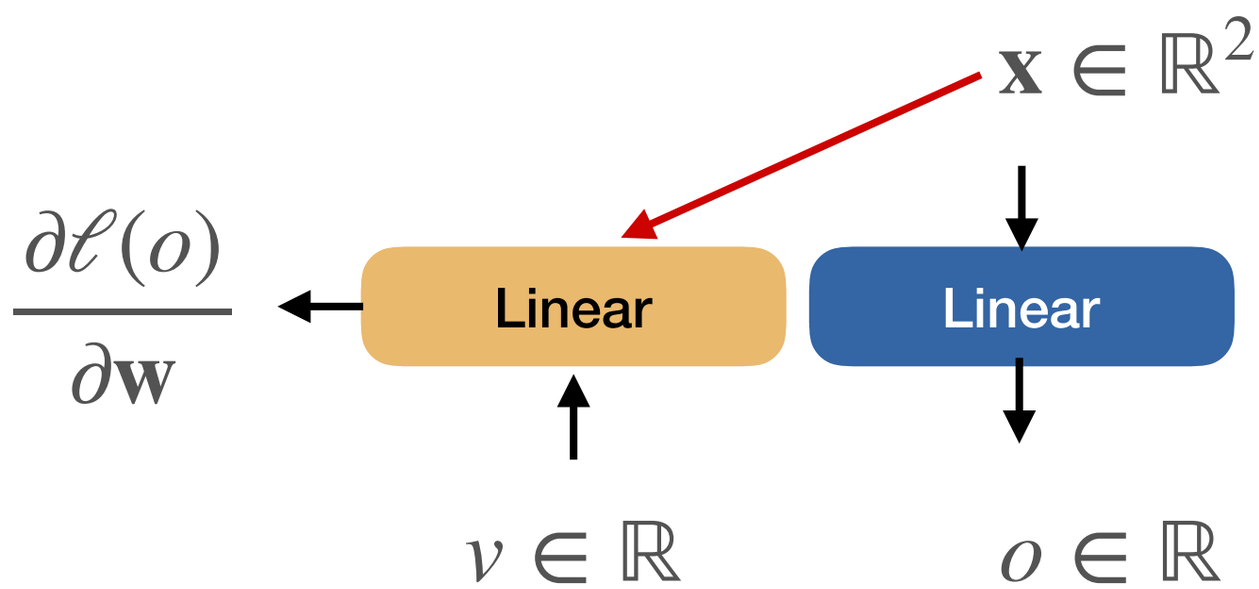
- Input: $\mathbf{x}_i$

- Apply affine transformation $\hat{\mathbf{x}}_i = \mathbf{x}_i - \mu_{\mathbf{x}}$

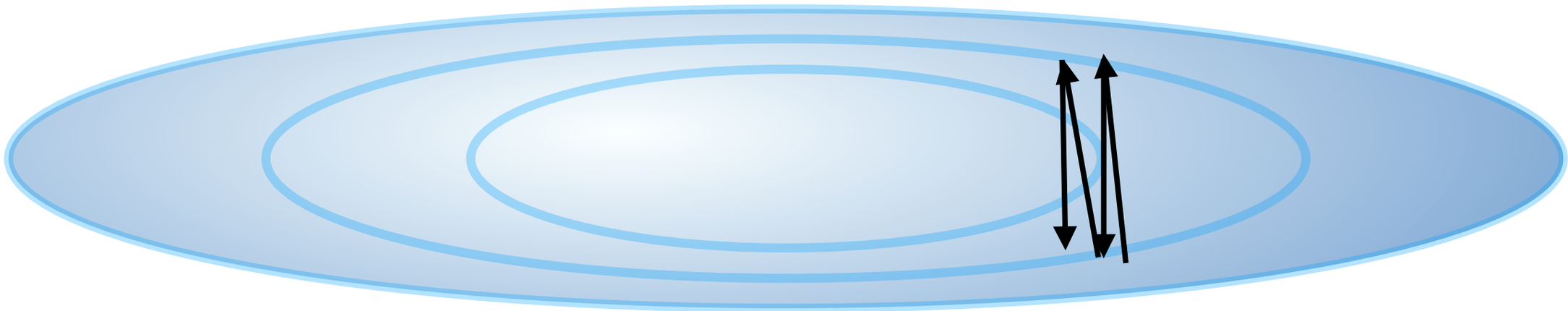
# Gradients of unnormalized inputs: A simple example

$$|\mathbf{x}[0]| \ll |\mathbf{x}[1]|$$

- Input vector $\mathbf{x}$
- Output scalar $o$
- $\frac{\partial \ell(o)}{\partial \mathbf{w}} = v\mathbf{x}^\top$



# Gradients of unnormalized inputs: A simple example



# Input normalization

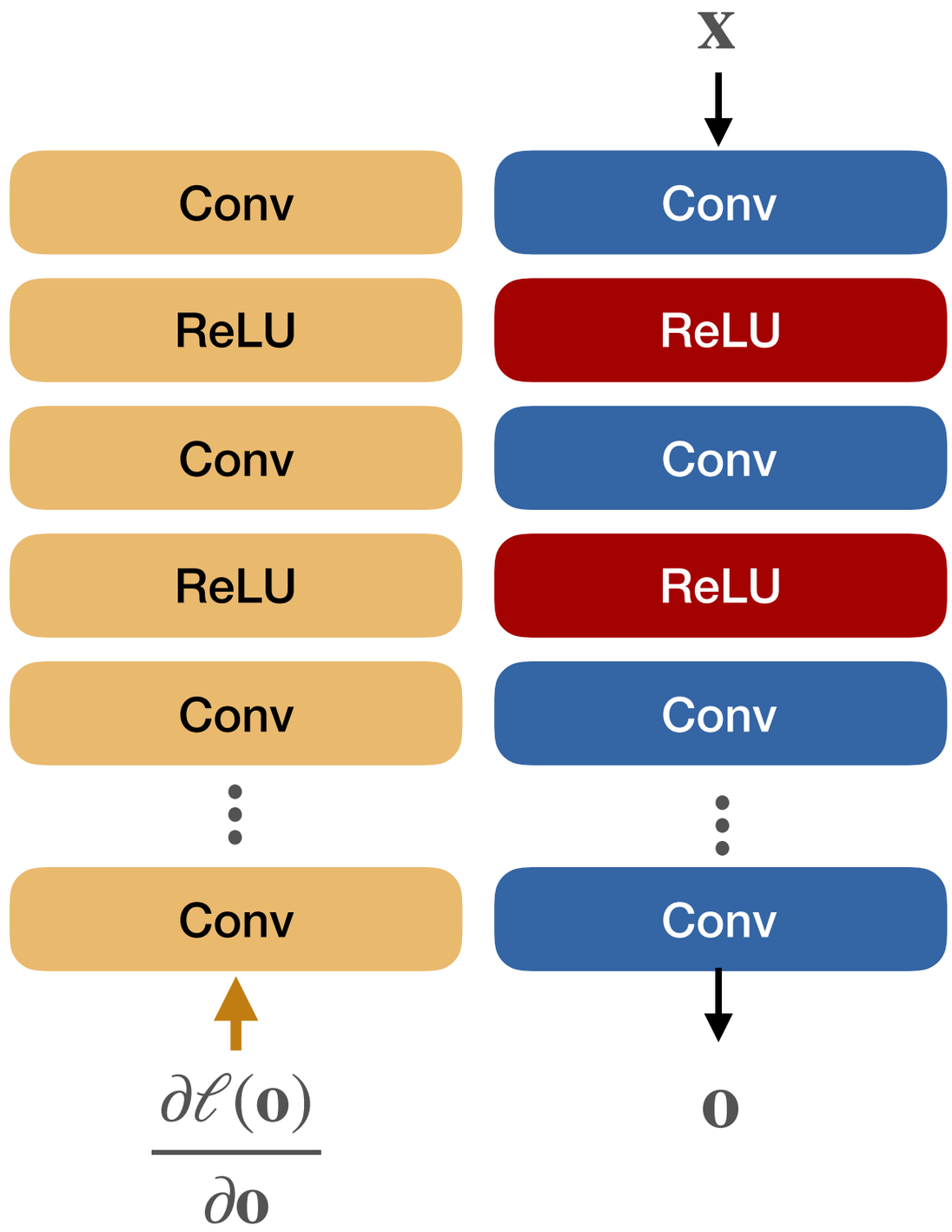
- Input: $\mathbf{x}_i$

- Apply affine transformation $\hat{\mathbf{x}}_i = (\mathbf{x}_i - \mu_{\mathbf{x}})/\sigma_{\mathbf{x}}$

# Vanishing and exploding gradients

# Exploding gradients

- Weight of one layer grows
  - Gradient of all other layers grow
  - Weights of other layers grow
- Bad feedback loop



# Exploding gradients

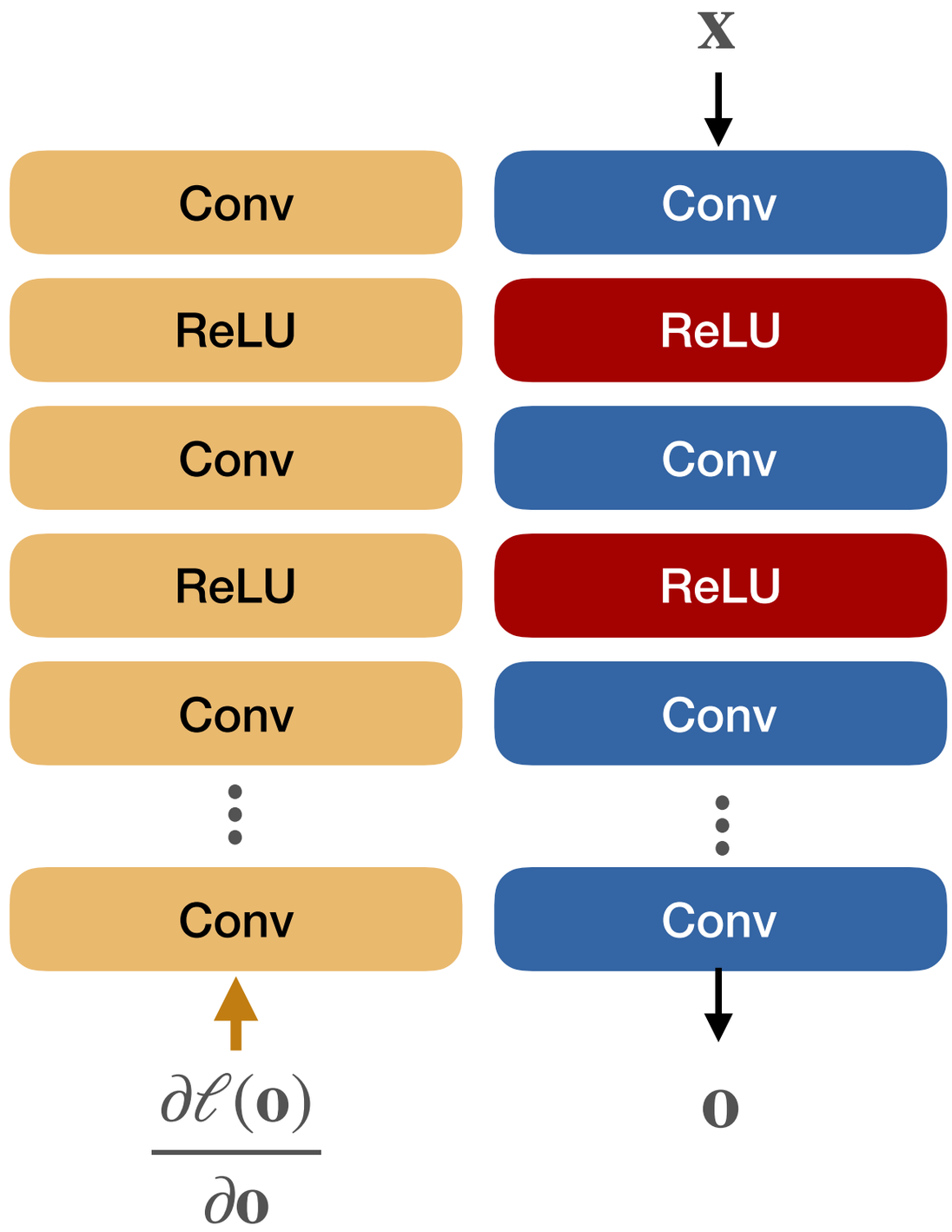
- Not a bit issue for most networks
- An issue for recurrent networks, and networks that share parameters across layers

# Detecting exploding gradients

- Plot loss
- Plot weight and gradient magnitude per layer

# Vanishing gradients

- Weight of one layer shrink
  - Gradient of all other layers shrink
  - Weights of other layers stay constant
- No progress



# Vanishing gradient

- Big issue for larger networks
- Issue for recurrent networks and weights tied across layers

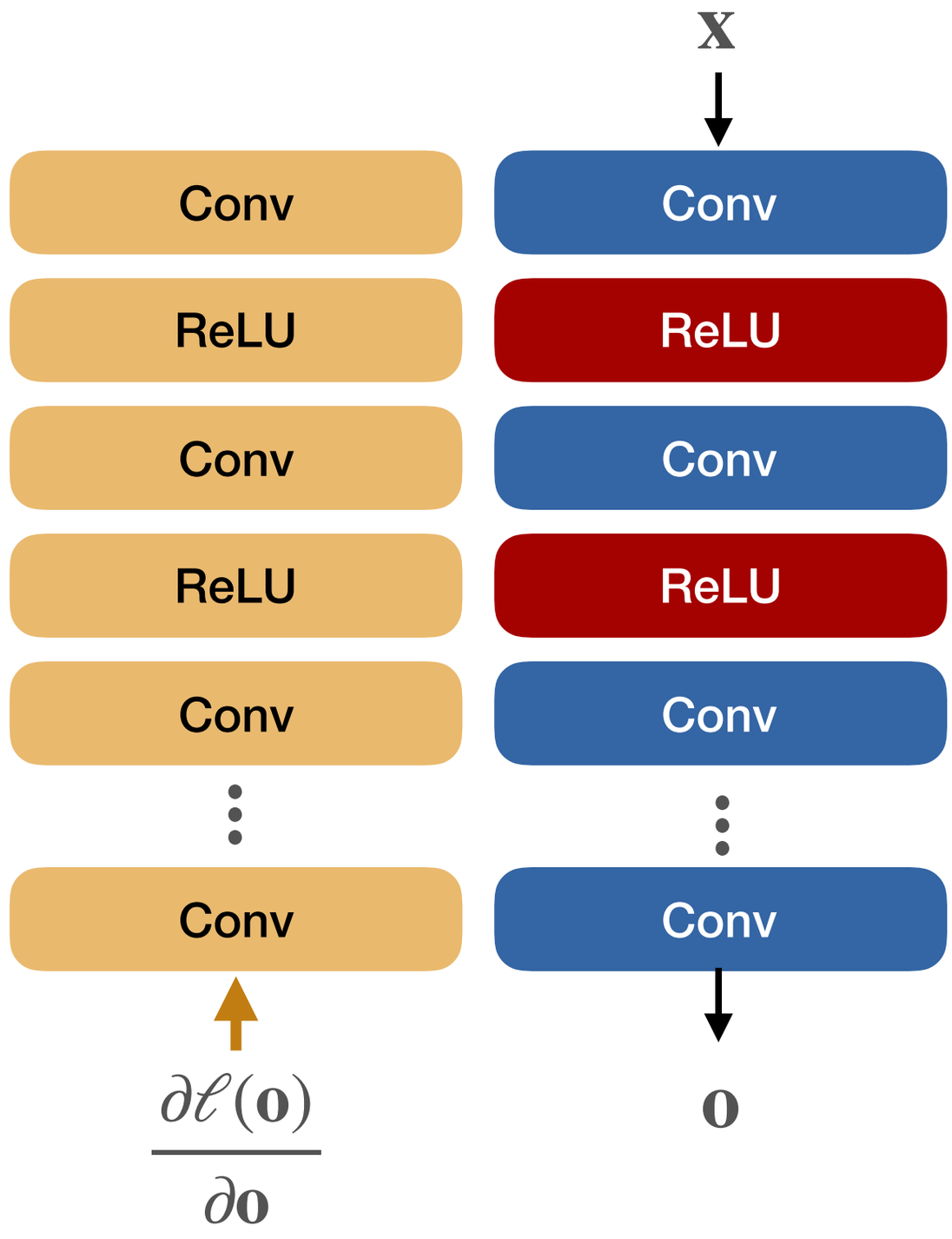
# Detecting vanishing gradients

- Plot loss
- Plot weight and gradient magnitude per layer

# Normalization

# Normalization

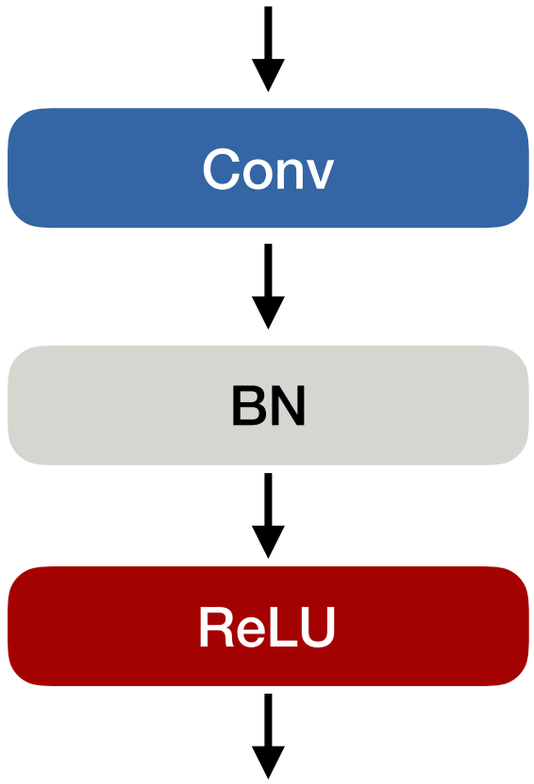
- How to prevent vanishing (or exploding) gradients?



# Batch normalization

# Batch normalization

- Make activations zero mean and unit variance




S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In ICML, 2015

# Batch normalization

- Normalize by channel-wise mean $\mu_c$ and standard deviation $\sigma_c$

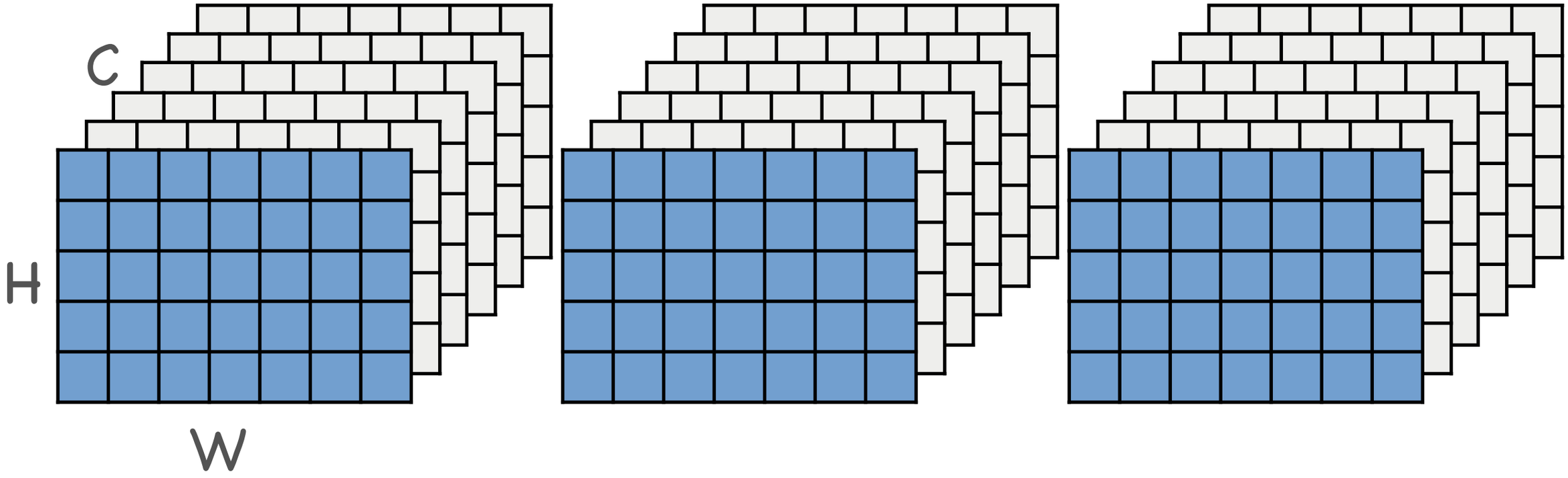
$$\mathbf{Z} \in \mathbb{R}^{B \times W \times H \times C}$$

$$\frac{\mathbf{Z}_{k,x,y,c} - \mu_c}{\sigma_c}$$

$$\mu_c = \frac{1}{BWH} \sum_{k,x,y} \mathbf{Z}_{k,x,y,c}$$

$$\sigma_c^2 = \frac{1}{BWH} \sum_{k,x,y} (\mathbf{Z}_{k,x,y,c} - \mu_c)^2$$

# What does batch normalization do?



# What does batch normalization do?

- The good:
  - Regularizes the network
  - Handles badly scaled weights
- The bad:
  - Mixes gradient information between samples

$$\mathbf{Z} \in \mathbb{R}^{B \times W \times H \times C}$$

$$\frac{\mathbf{Z}_{k,x,y,c} - \mu_c}{\sigma_c}$$

$$\mu_c = \frac{1}{BWH} \sum_{k,x,y} \mathbf{Z}_{k,x,y,c}$$

$$\sigma_c^2 = \frac{1}{BWH} \sum_{k,x,y} (\mathbf{Z}_{k,x,y,c} - \mu_c)^2$$

# Batch norm and batch size

- Large batch sizes work better
- More stable mean and standard deviation estimates

$$\mathbf{Z} \in \mathbb{R}^{B \times W \times H \times C}$$

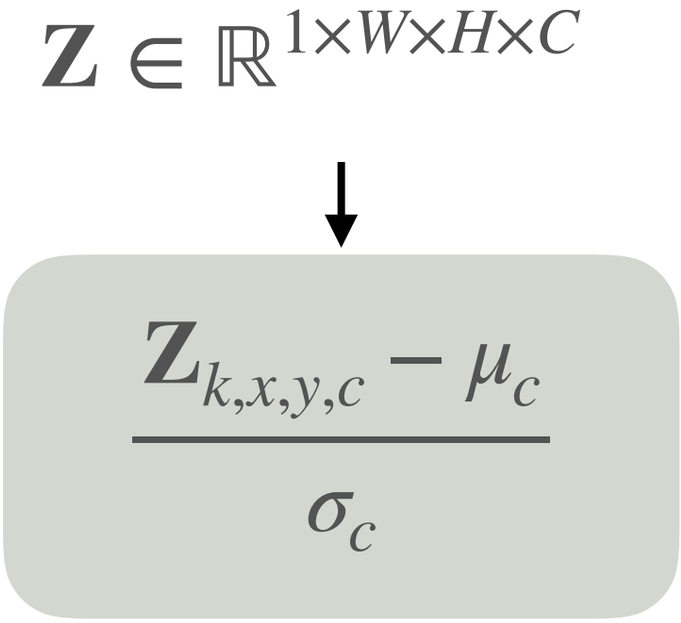
$$\frac{\mathbf{Z}_{k,x,y,c} - \mu_c}{\sigma_c}$$

$$\mu_c = \frac{1}{BWH} \sum_{k,x,y} \mathbf{Z}_{k,x,y,c}$$

$$\sigma_c^2 = \frac{1}{BWH} \sum_{k,x,y} (\mathbf{Z}_{k,x,y,c} - \mu_c)^2$$

# Batch norm at test time

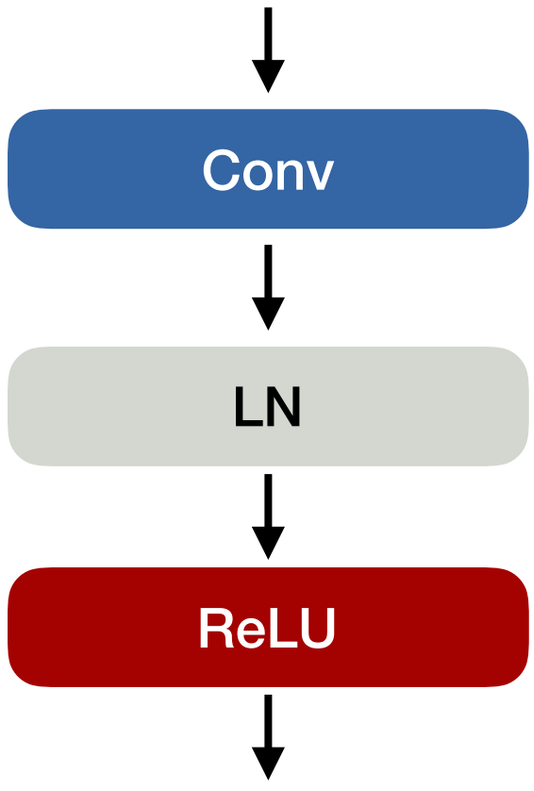
- Compute mean and standard deviation on training set using running average



# Layer normalization

# Layer normalization

- Make activations zero mean and unit variance without collecting statistics across batches




Layer Normalization, Ba, J., Kiros, J. R. and Hinton, G., arXiv preprint arXiv: 1607.06450, 2016

# Layer normalization

- Normalize by image-wise mean $\mu_k$ and standard deviation $\sigma_k$

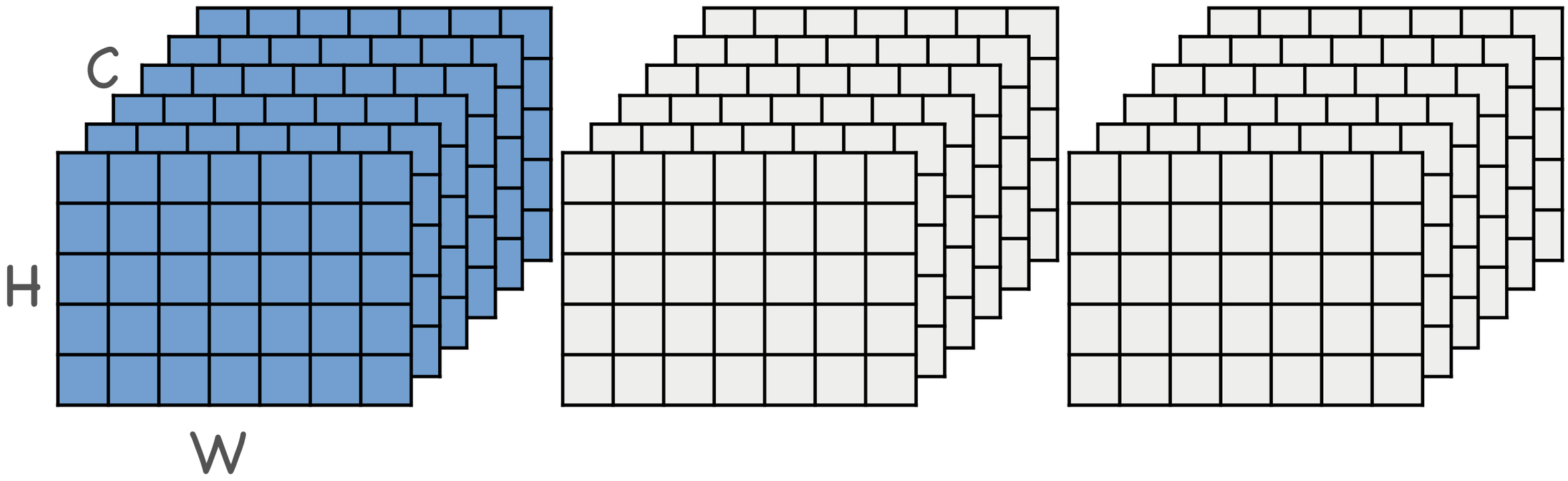
$$\mathbf{Z} \in \mathbb{R}^{B \times W \times H \times C}$$

$$\frac{\mathbf{Z}_{k,x,y,c} - \mu_k}{\sigma_k}$$

$$\mu_k = \frac{1}{WHC} \sum_{x,y,c} \mathbf{Z}_{k,x,y,c}$$

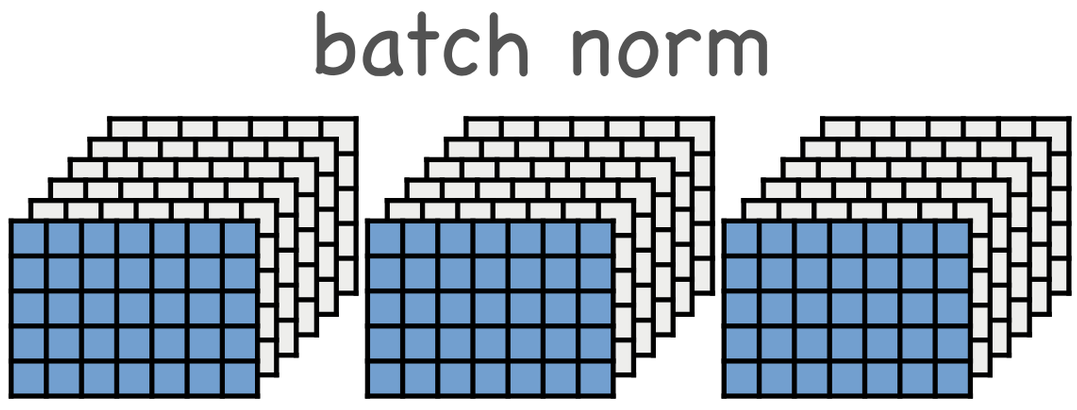
$$\sigma_k^2 = \frac{1}{WHC} \sum_{x,y,c} (\mathbf{Z}_{k,x,y,c} - \mu_k)^2$$

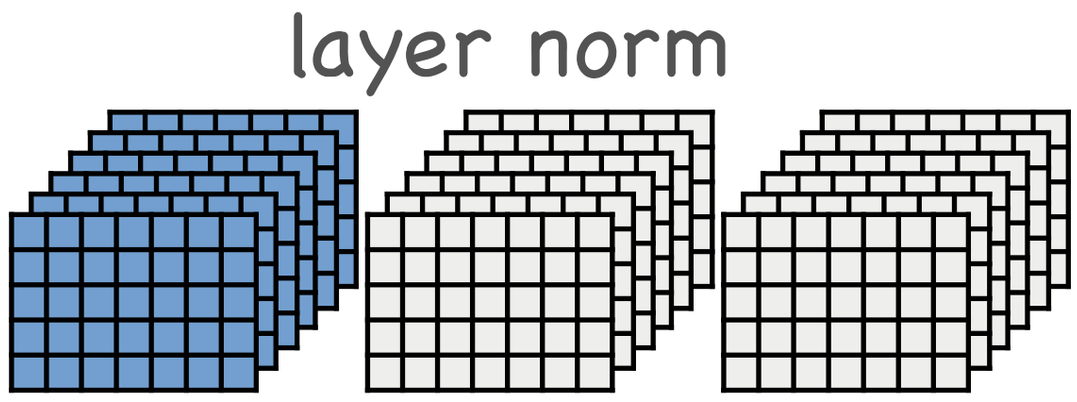
# What does layer normalization do?



# Comparison to batch norm

- No summary statistics
  - Training and testing are the same
- Works well for sequence models
- Does not scale activations individually

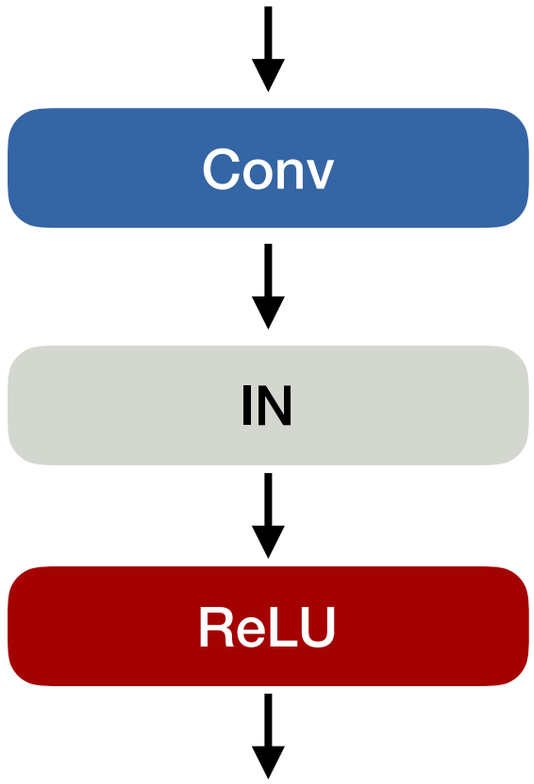


# Instance normalization

# Instance normalization

- Batch norm per input




Ulyanov, Dmitry, Andrea Vedaldi, and Victor Lempitsky. "Instance normalization: The missing ingredient for fast stylization." arXiv 2016.

# Instance normalization

- Normalize by spatial mean $\mu_{kc}$ and standard deviation $\sigma_{kc}$

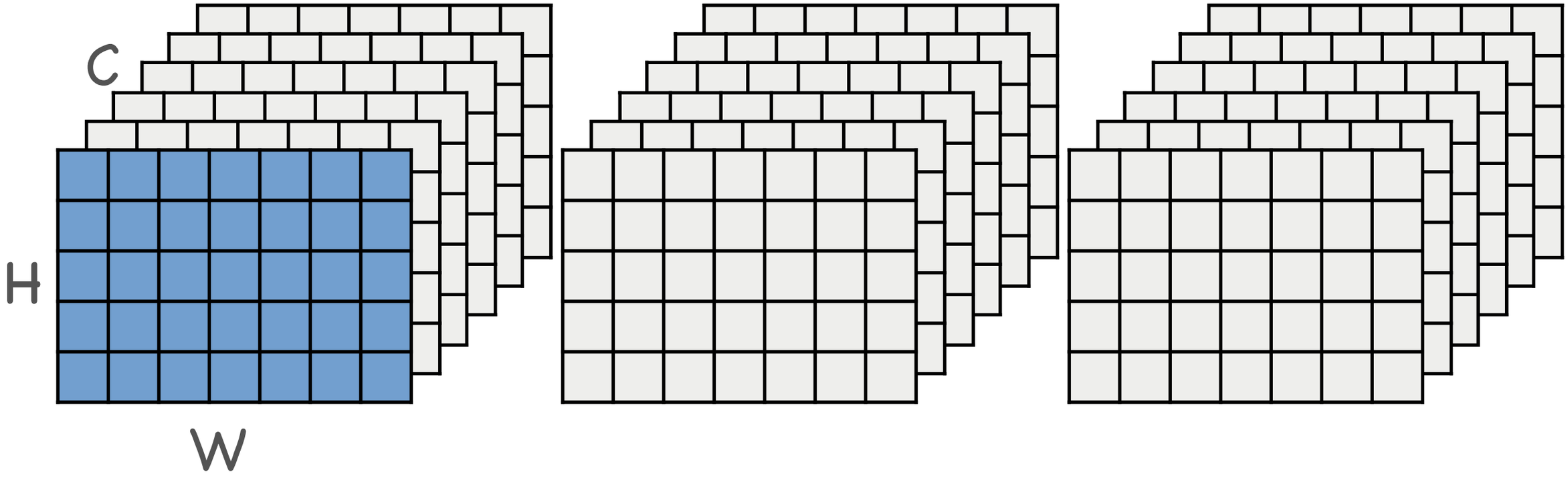
$$\mathbf{Z} \in \mathbb{R}^{B \times W \times H \times C}$$

$$\frac{\mathbf{Z}_{k,x,y,c} - \mu_{kc}}{\sigma_{kc}}$$

$$\mu_{kc} = \frac{1}{WH} \sum_{x,y} \mathbf{Z}_{k,x,y,c}$$

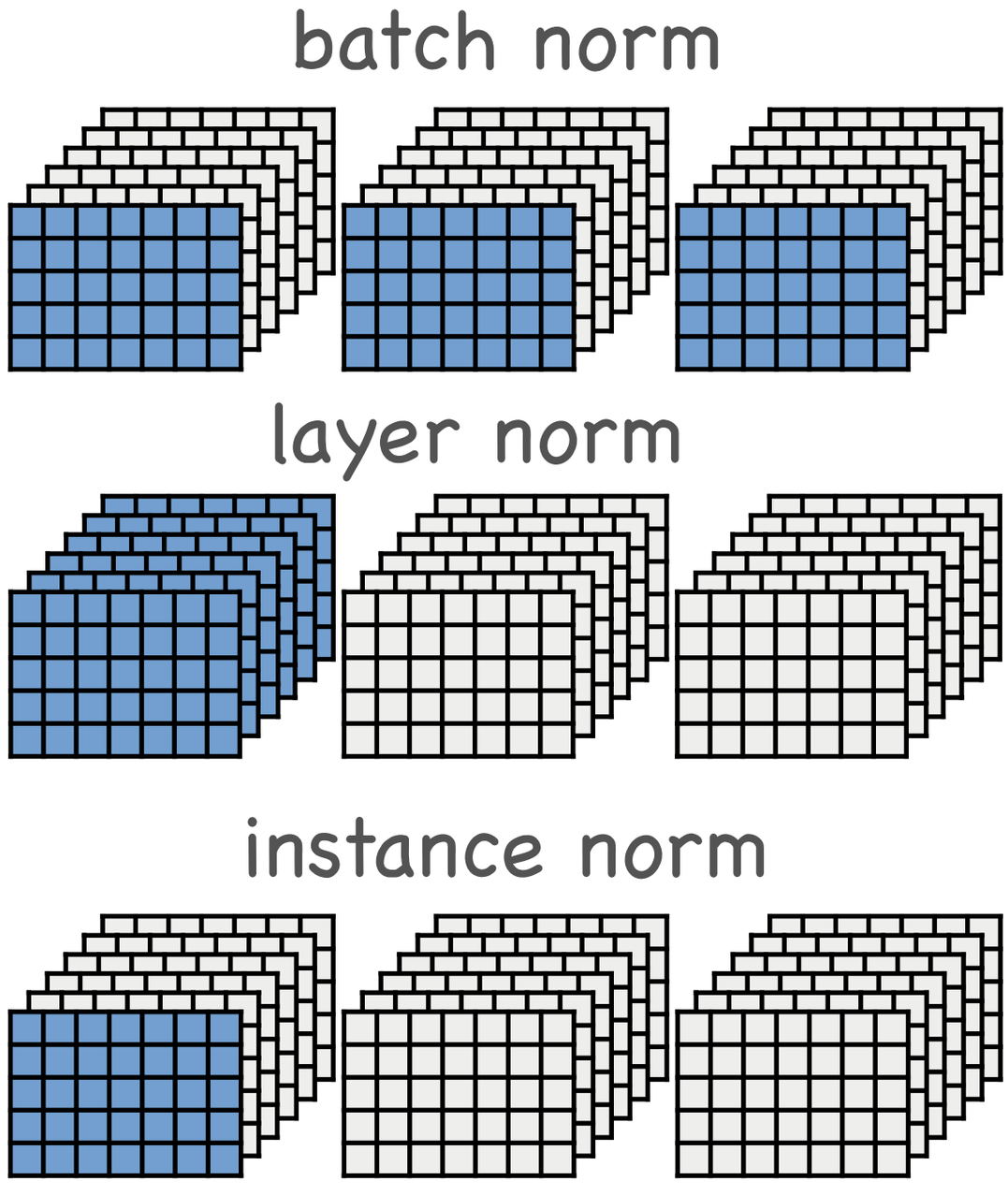
$$\sigma_{kc}^2 = \frac{1}{WH} \sum_{x,y} (\mathbf{Z}_{k,x,y,c} - \mu_{kc})^2$$

# What does instance normalization do?



# Comparison to batch norm

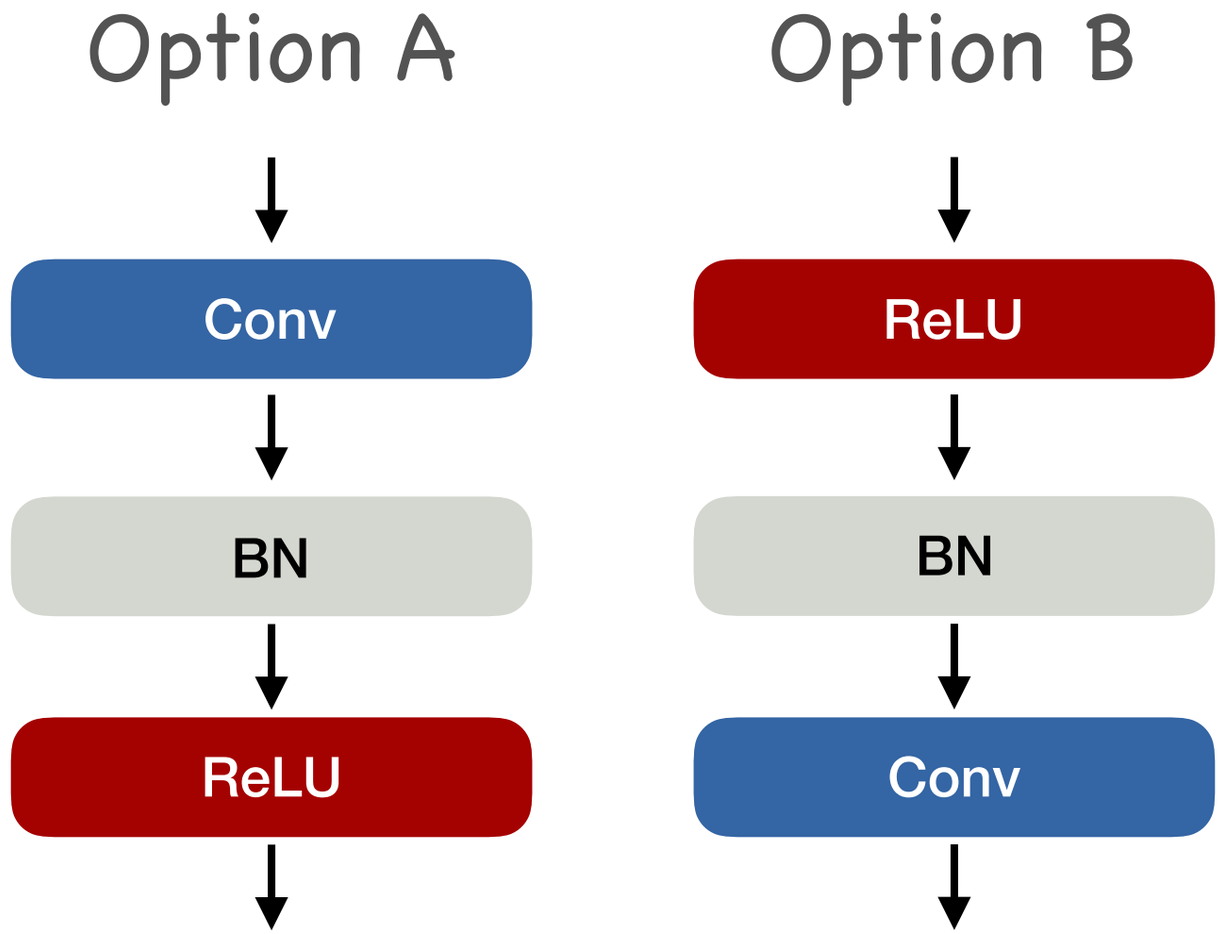
- No summing over batches
- Works well for graphics applications
- Not used much in recognition
  - Unstable statistics



# Where to add normalization?

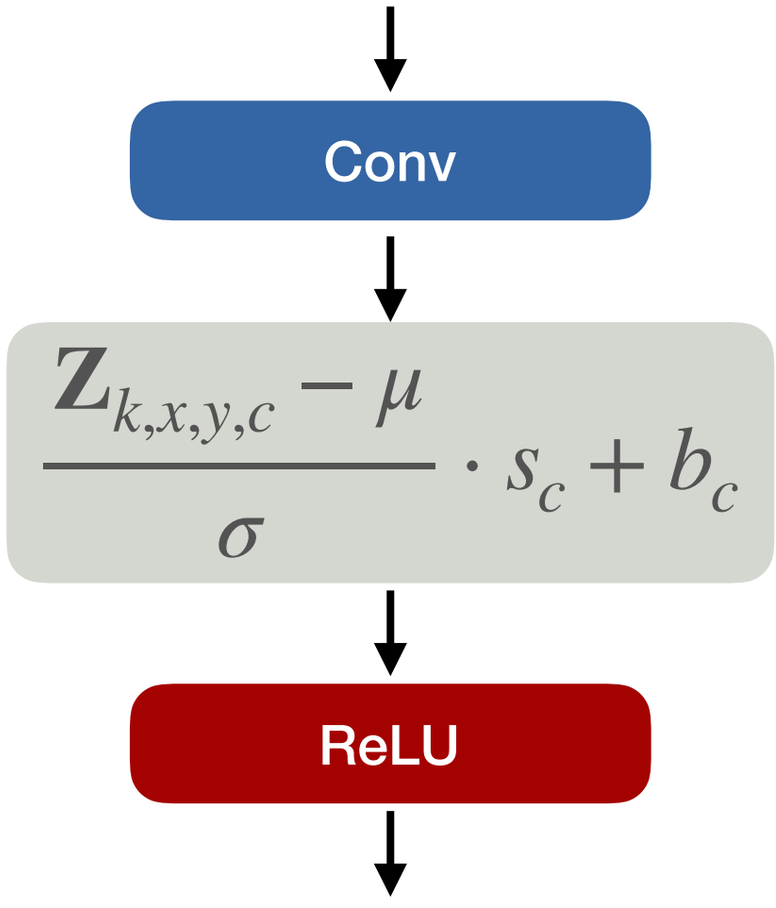
# Where to add normalization?

- Option A
  - After convolution
- Option B
  - After ReLU (non-linearity)



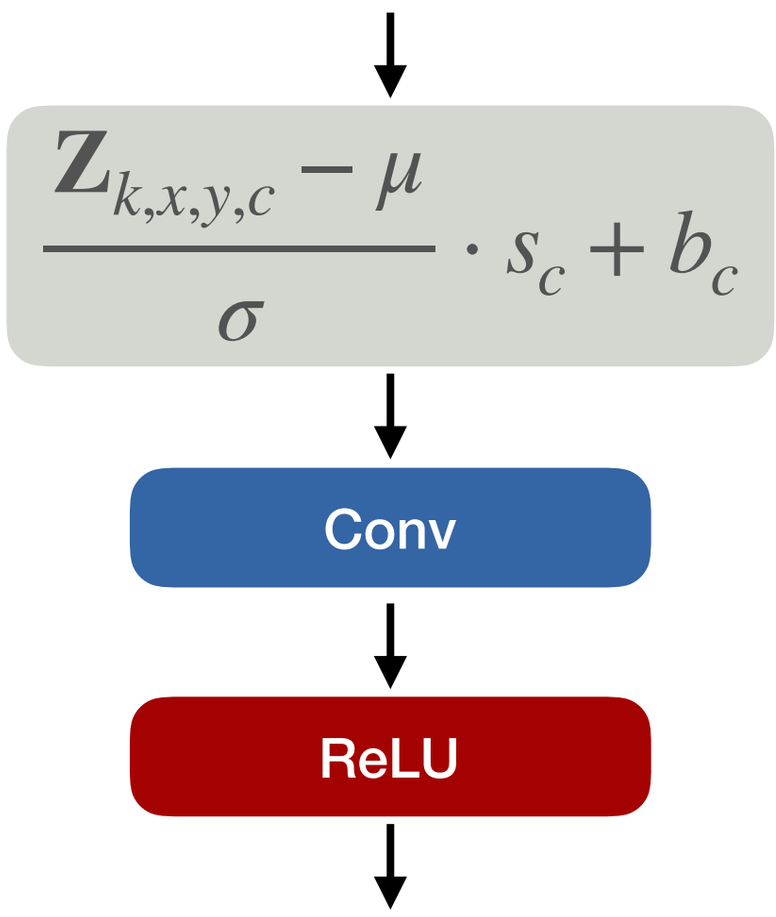
# Option A

- No bias in conv
- Activations are zero mean
  - Half of activations zeroed out in ReLU
- Solution:
  - Learn a scale $s_c$ and bias $b_c$ after norm



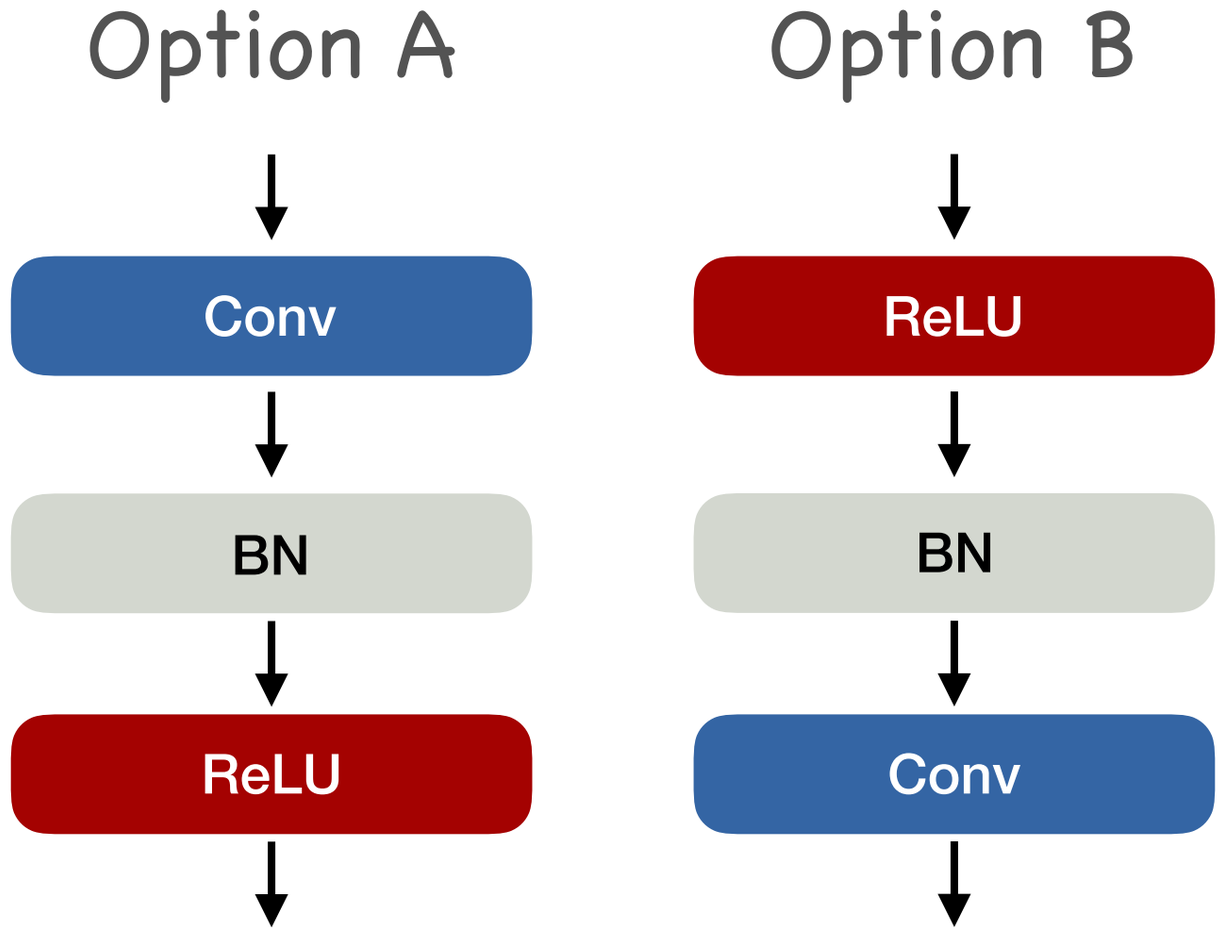
# Option B

- Scale $s_c$ and bias $b_c$ optional



# Where to add normalization?

- Both work
- Option A is more popular
- Option B is easier
  - Scale and bias optional
  - Conv unchanged



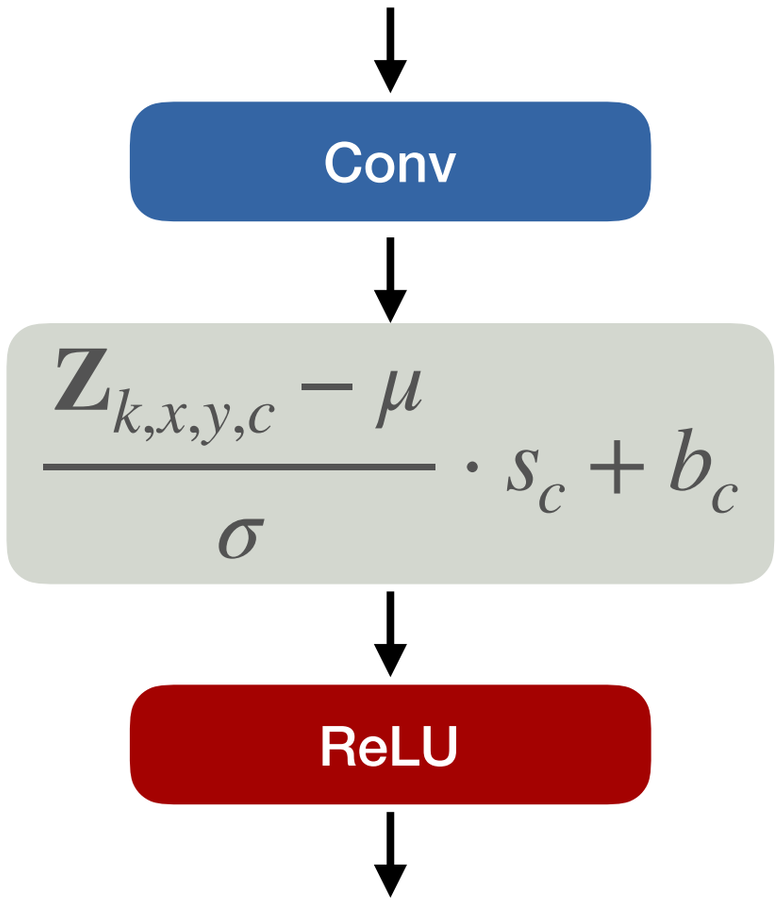
# Where not to add batch norm?

- After fully connected layers

- Mean and standard deviation estimates too unstable

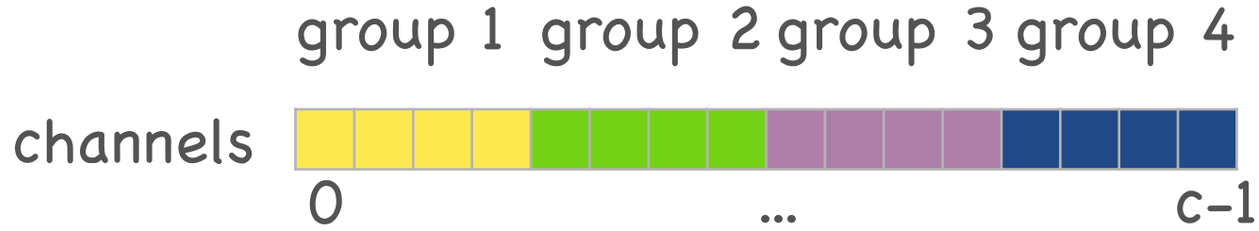
# Why does normalization work?

- Regularizes the network
- Handles badly scaled weights
- Single parameter to learn scale $s_c$



# Group normalization and local response normalization

# Group normalization




- Normalize groups of G channels together

$$\mathbf{Z} \in \mathbb{R}^{B \times W \times H \times C}$$

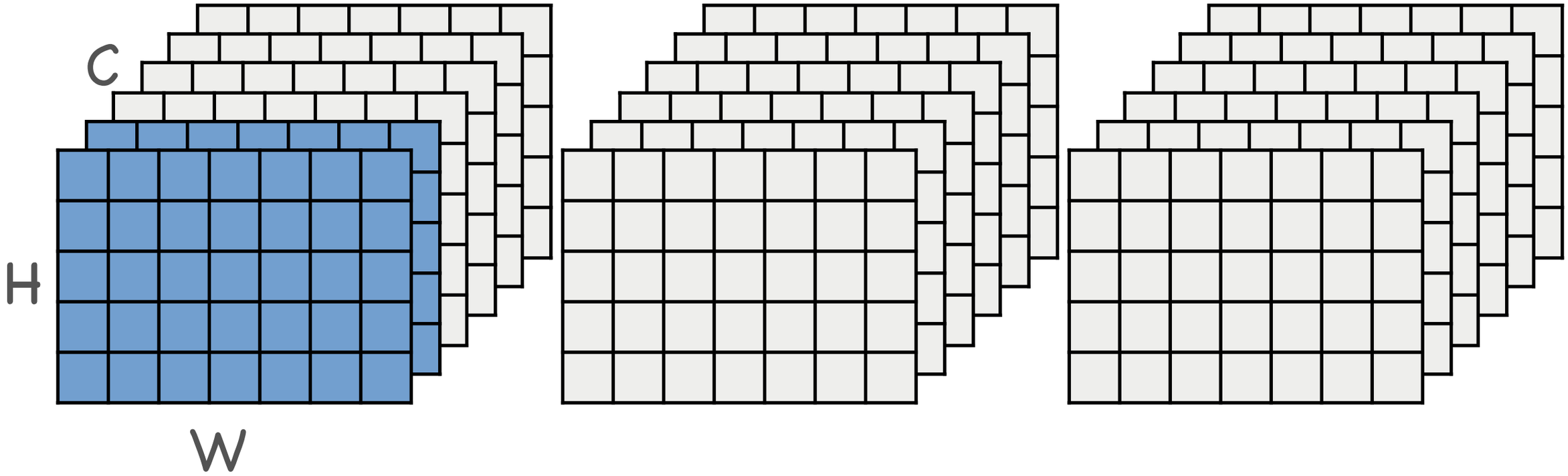
$$\frac{\mathbf{Z}_{k,x,y,c} - \mu_{kg}}{\sigma_{kg}} \qquad g = \lfloor c/G \rfloor$$

$$\mu_{kg} = \frac{1}{WHG} \sum_{c=gG}^{(g+1)G-1} \sum_{x,y} \mathbf{Z}_{k,x,y,c}$$

$$\sigma_{kg}^2 = \frac{1}{WHG} \sum_{c=gG}^{(g+1)G-1} \sum_{x,y} (\mathbf{Z}_{k,x,y,c} - \mu_{kg})^2$$

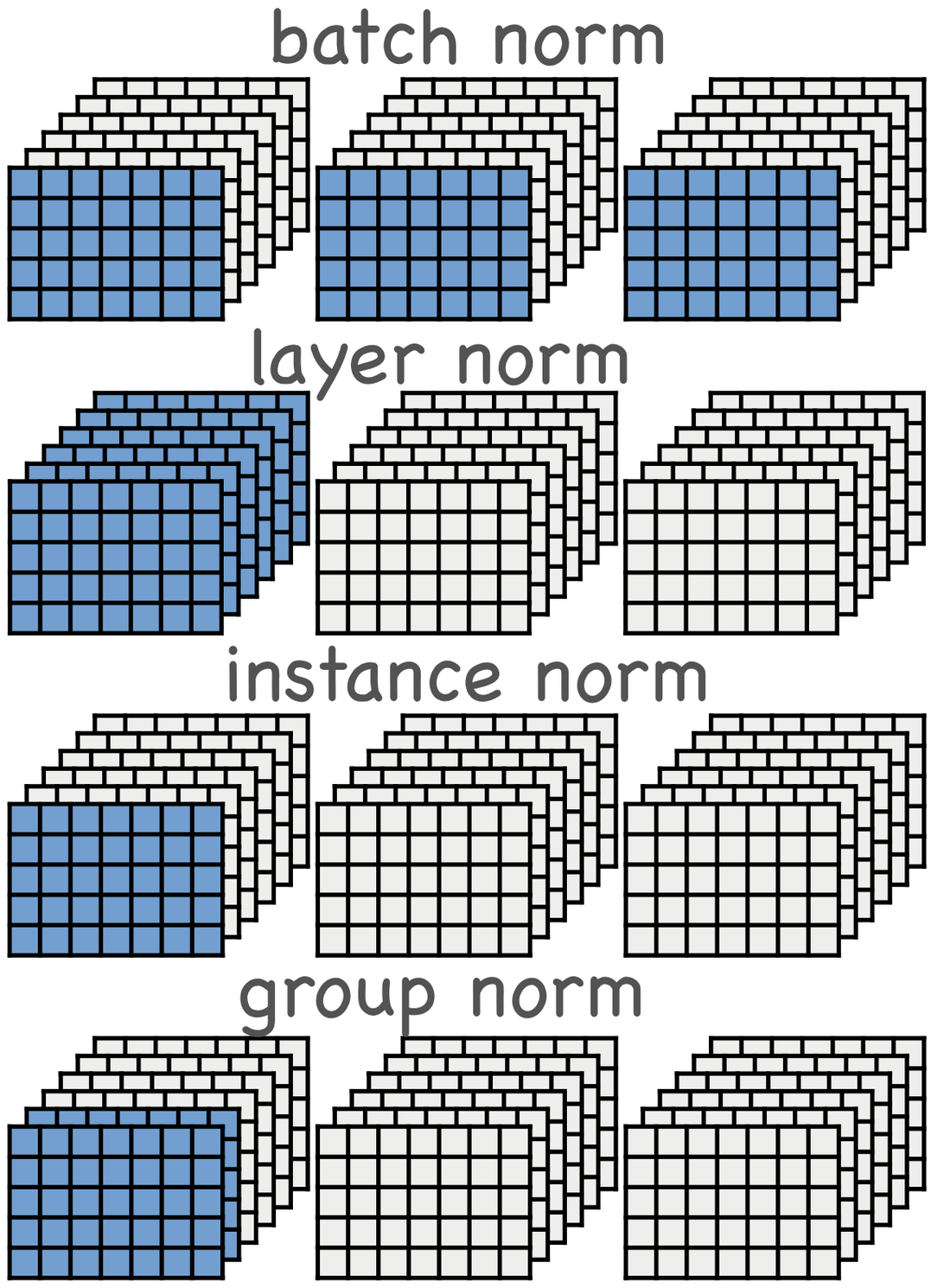
Yuxin Wu, and Kaiming He. "Group normalization." ECCV. 2018.

# What does group normalization do?



# Comparison to other norms

- More stable statistics than instance norm
  - G=C
- Not all channels tied as in layer norm
  - G=1



# Local response normalization


- "Generalization" of group norm
- Parameters $\alpha$ and $\beta$

$$\mathbf{Z} \in \mathbb{R}^{B \times W \times H \times C}$$

$$\mathbf{Z}_{k,x,y,c} \left( \gamma + \frac{\alpha}{n} \sum_{c'=c-\frac{n}{2}}^{c+\frac{n}{2}} \mathbf{Z}^2_{k,x,y,c} \right)^{-\beta}$$

Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. "Imagenet classification with deep convolutional neural networks." NIPS 2012

# Differences between LRN and GN

- Group norm
  - Normalize over all spatial locations
  - Subtract mean
  - Scale and bias transformation
- Local response normalization
  - More flexible parametrization


$$\mathbf{Z} \in \mathbb{R}^{B \times W \times H \times C}$$

$$\mathbf{Z}_{k,x,y,c} \left( \gamma + \frac{\alpha}{n} \sum_{c'=c-\frac{n}{2}}^{c+\frac{n}{2}} \mathbf{Z}^2_{k,x,y,c} \right)^{-\beta}$$

# 16

January 23, 2024

[1]:
```
%pylab inline
import torch
import sys
sys.path.append('..')
sys.path.append('../..')
from data import load
device = torch.device('cuda') if torch.cuda.is_available() else torch.
 ↪device('cpu')
print('device = ', device)
```

```
%pylab is deprecated, use %matplotlib inline and import the required libraries.
Populating the interactive namespace from numpy and matplotlib
device =  cuda
```

[2]:
```
class ConvNet(torch.nn.Module):
    class Block(torch.nn.Module):
        def __init__(self, n_input, n_output, stride=1):
            super().__init__()
            self.net = torch.nn.Sequential(
              torch.nn.Conv2d(n_input, n_output, kernel_size=3, padding=1,␣
 ↪stride=stride, bias=False),
              torch.nn.BatchNorm2d(n_output),
              torch.nn.ReLU(),
              torch.nn.Conv2d(n_output, n_output, kernel_size=3, padding=1,␣
 ↪bias=False),
              torch.nn.BatchNorm2d(n_output),
              torch.nn.ReLU()
            )

        def forward(self, x):
            return self.net(x)

    def __init__(self, layers=[32,64,128], n_input_channels=3):
        super().__init__()
        L = [torch.nn.Conv2d(n_input_channels, 32, kernel_size=7, padding=3,␣
 ↪stride=2, bias=False),
             torch.nn.BatchNorm2d(32),
```

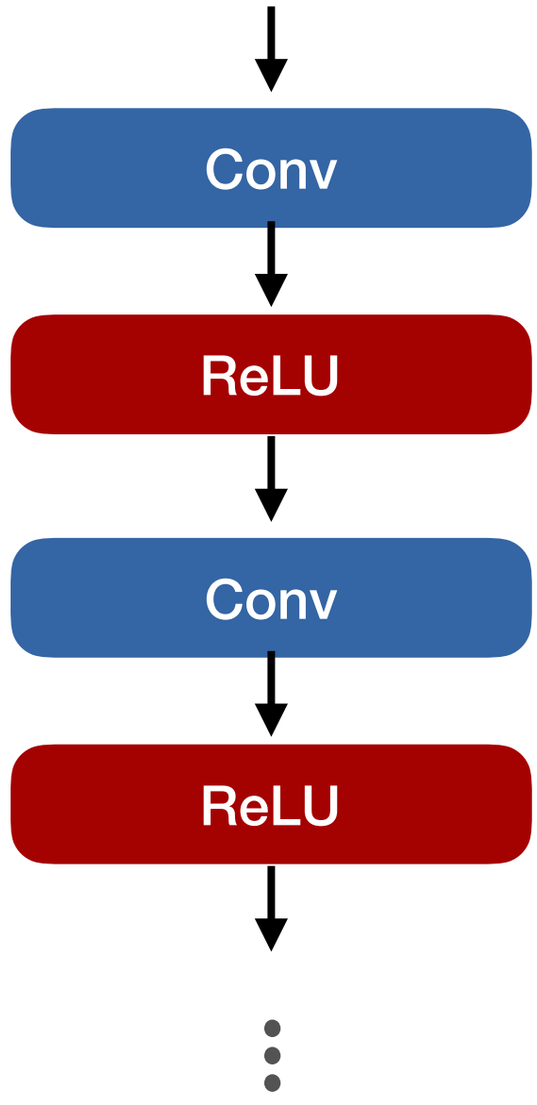
```
            torch.nn.ReLU(),
            torch.nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
           ]
        c = 32
        for l in layers:
            L.append(self.Block(c, l, stride=2))
            c = l
        self.network = torch.nn.Sequential(*L)
        self.classifier = torch.nn.Linear(c, 1)

    def forward(self, x):
        # Compute the features
        z = self.network(x)
        # Global average pooling
        z = z.mean(dim=[2,3])
        # Classify
        return self.classifier(z)[:,0]

net = ConvNet()
net.train()
print( net.training )
net.eval()
print( net.training )
```

```
True
False
```

# Residual connections

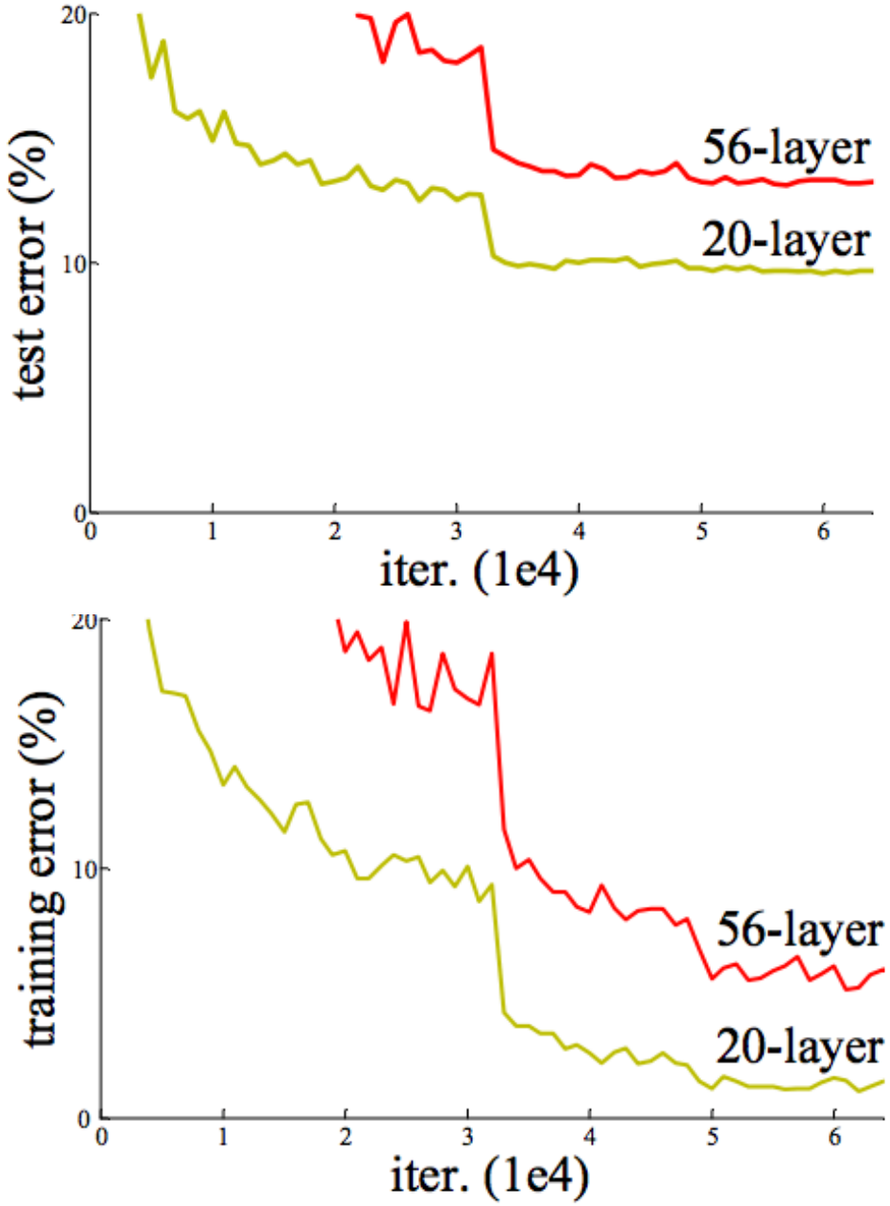
# Deep networks

- Without normalization
  - Max depth 10-12
- With normalization
  - Max depth 20-30



# What happens to deeper networks?

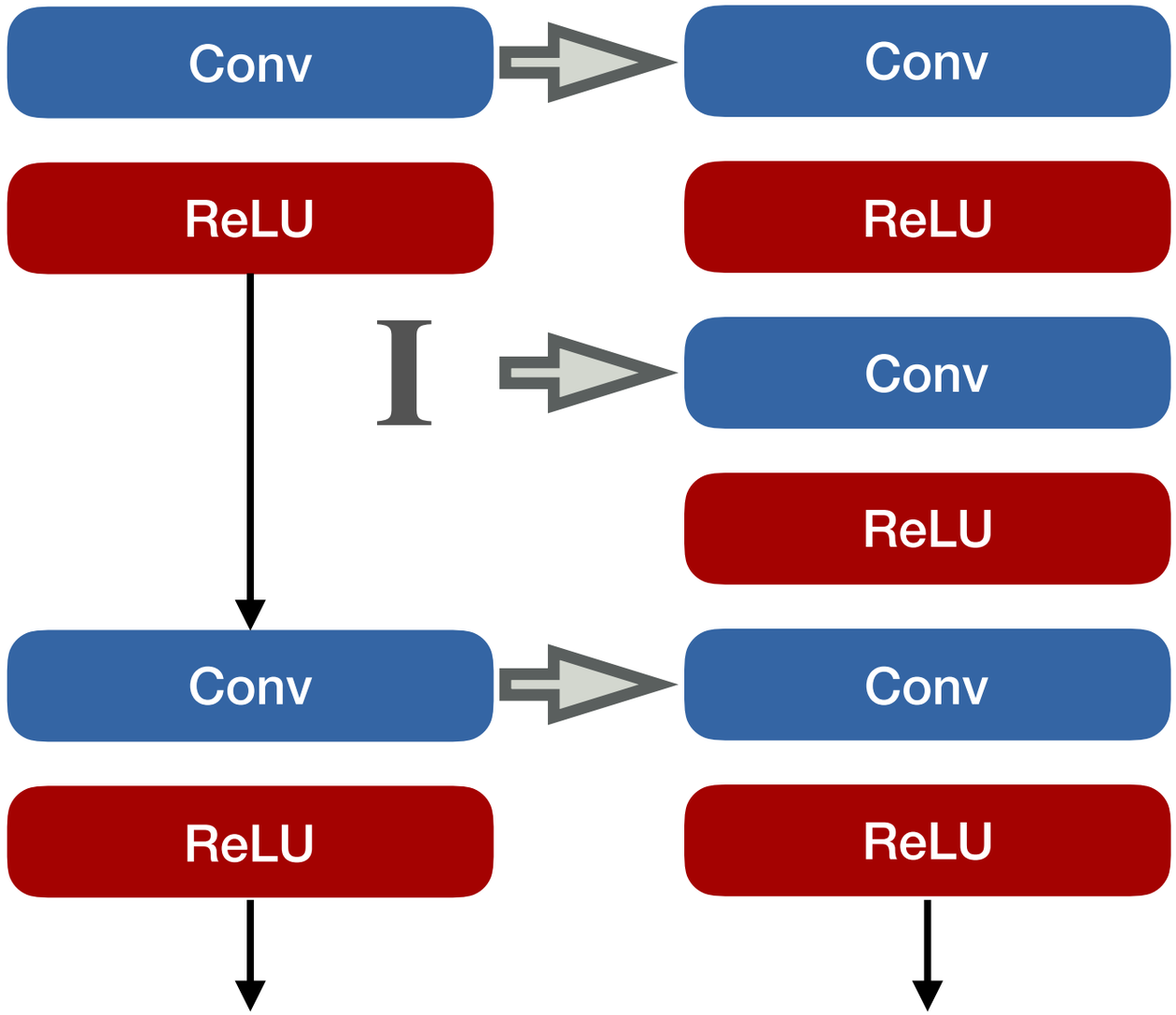
- It does not train well




[Figure source: Kaiming He et al., "Deep Residual Learning for Image Recognition", CVPR 2016]

# What happens to deeper networks?

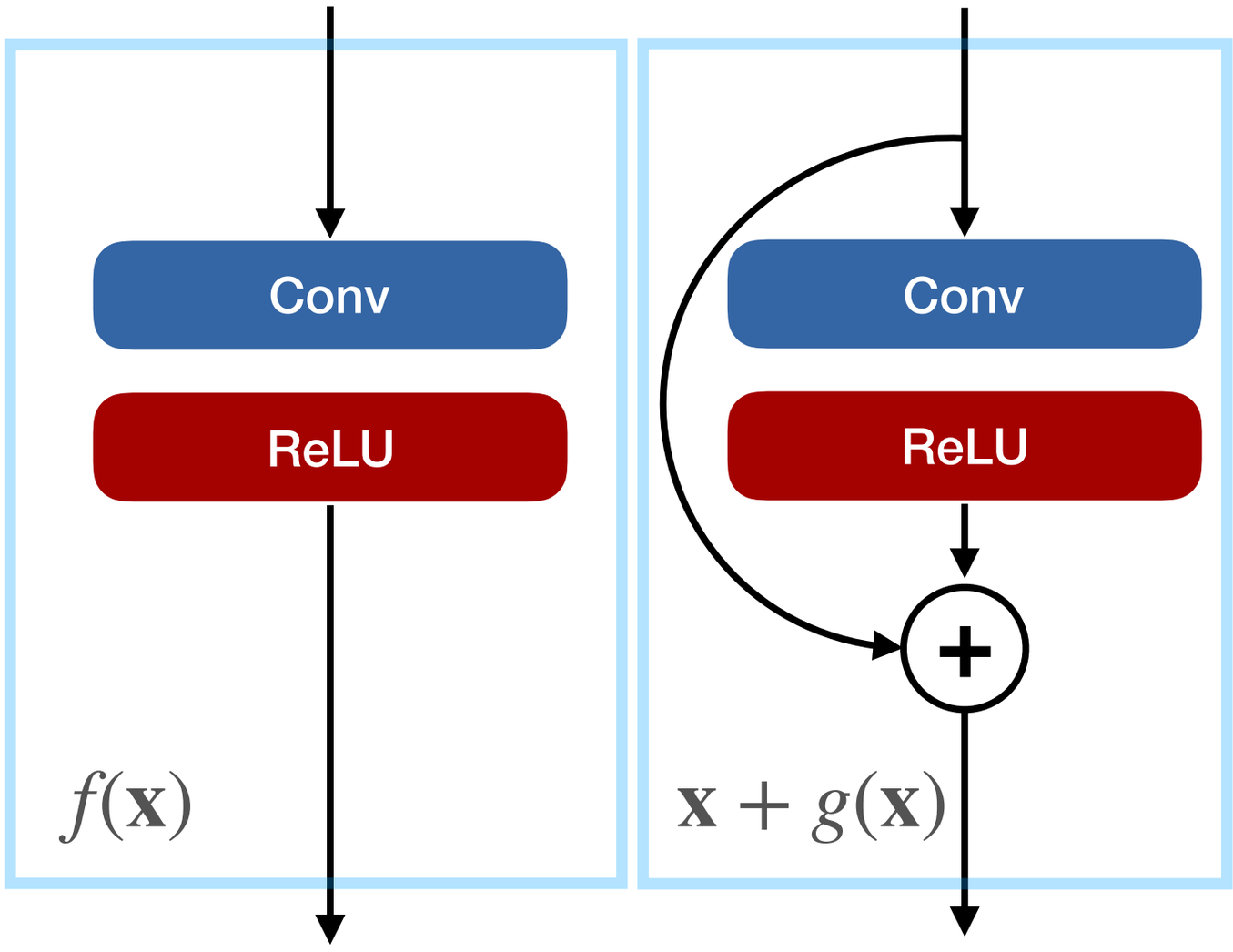
- Training a shallower network and adding identity layers works better



# Solution: Residual connections

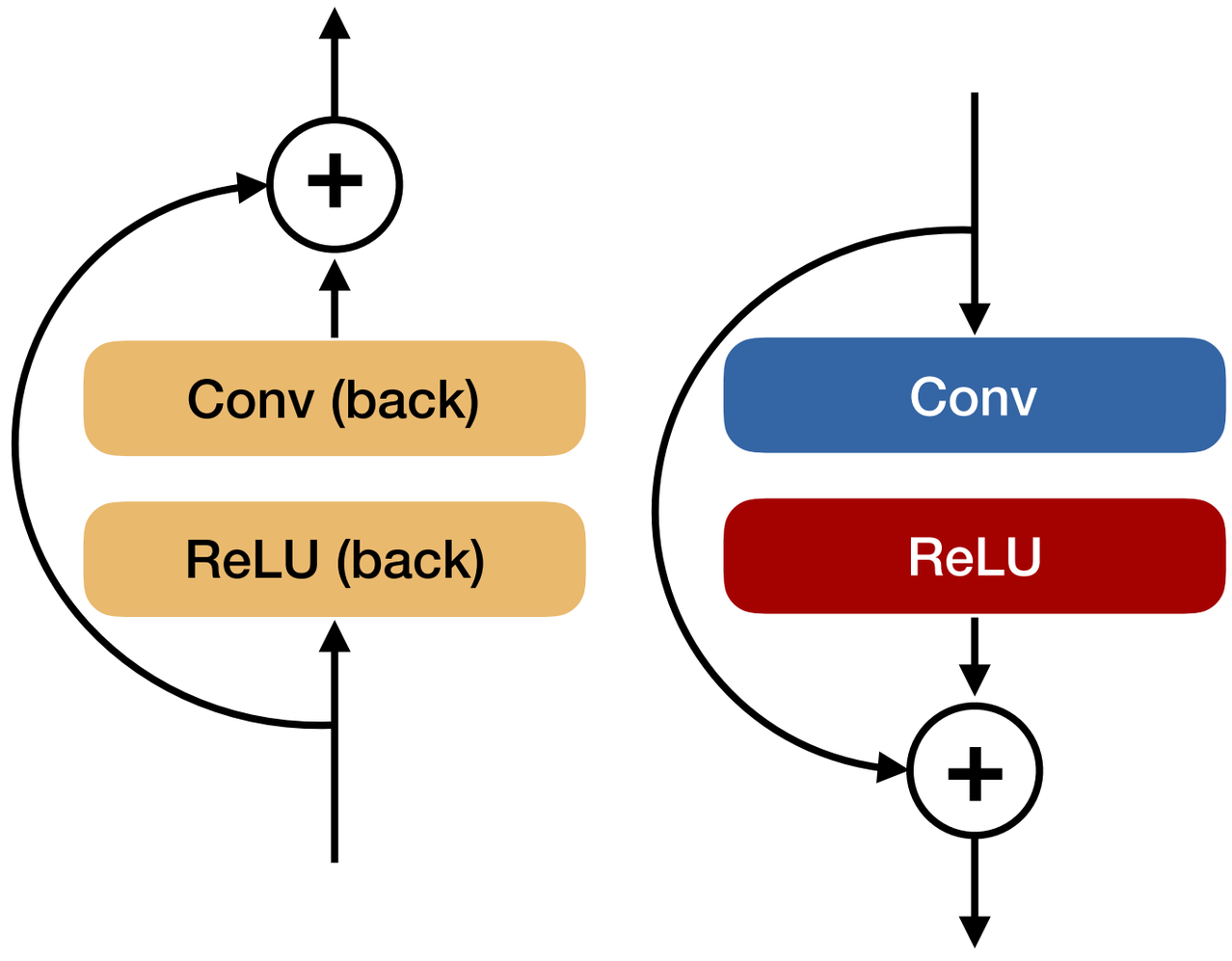
- Parametrize layers as

$$f(\mathbf{x}) = \mathbf{x} + g(\mathbf{x})$$

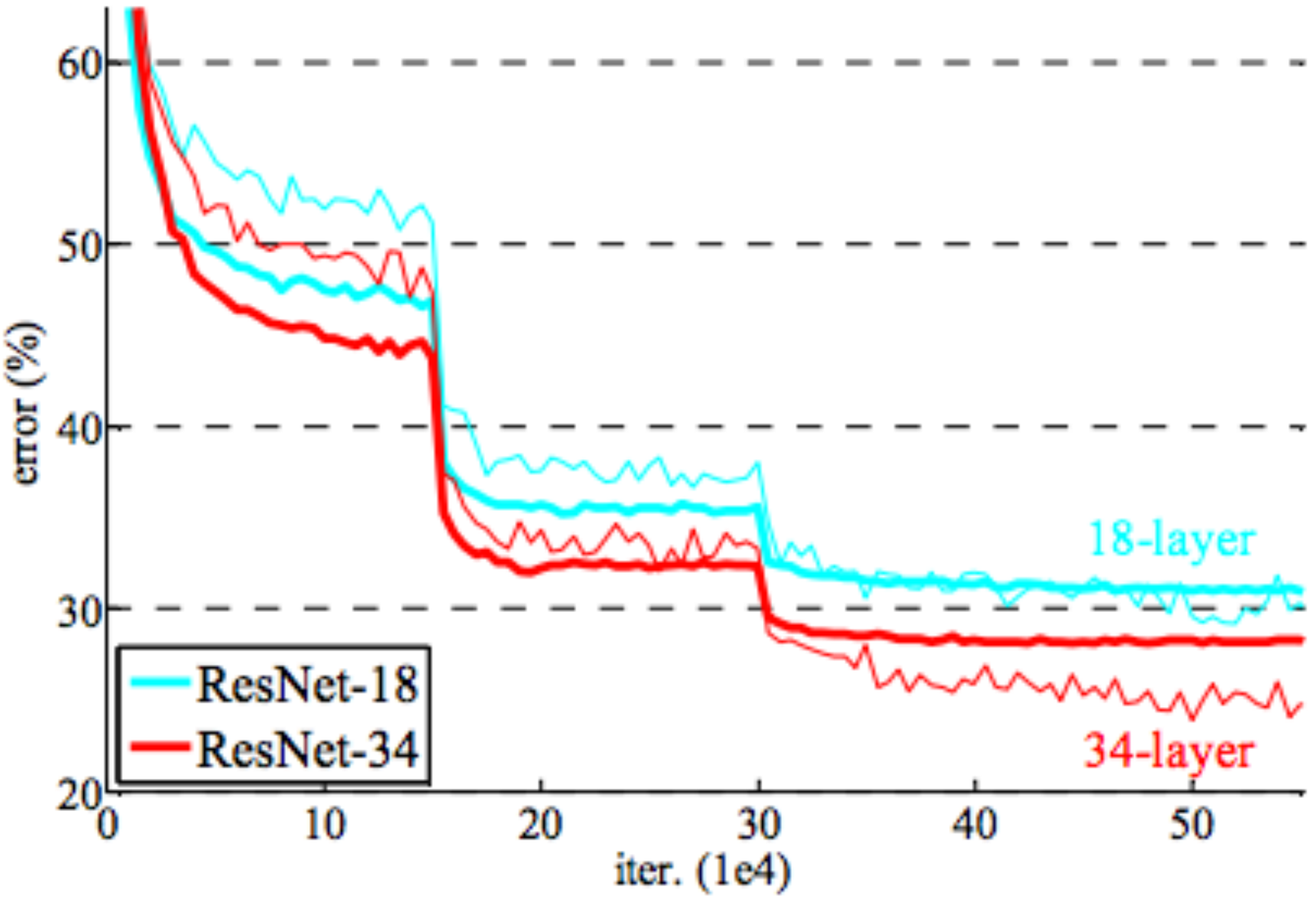


# Fun fact

- Backward graph is symmetric



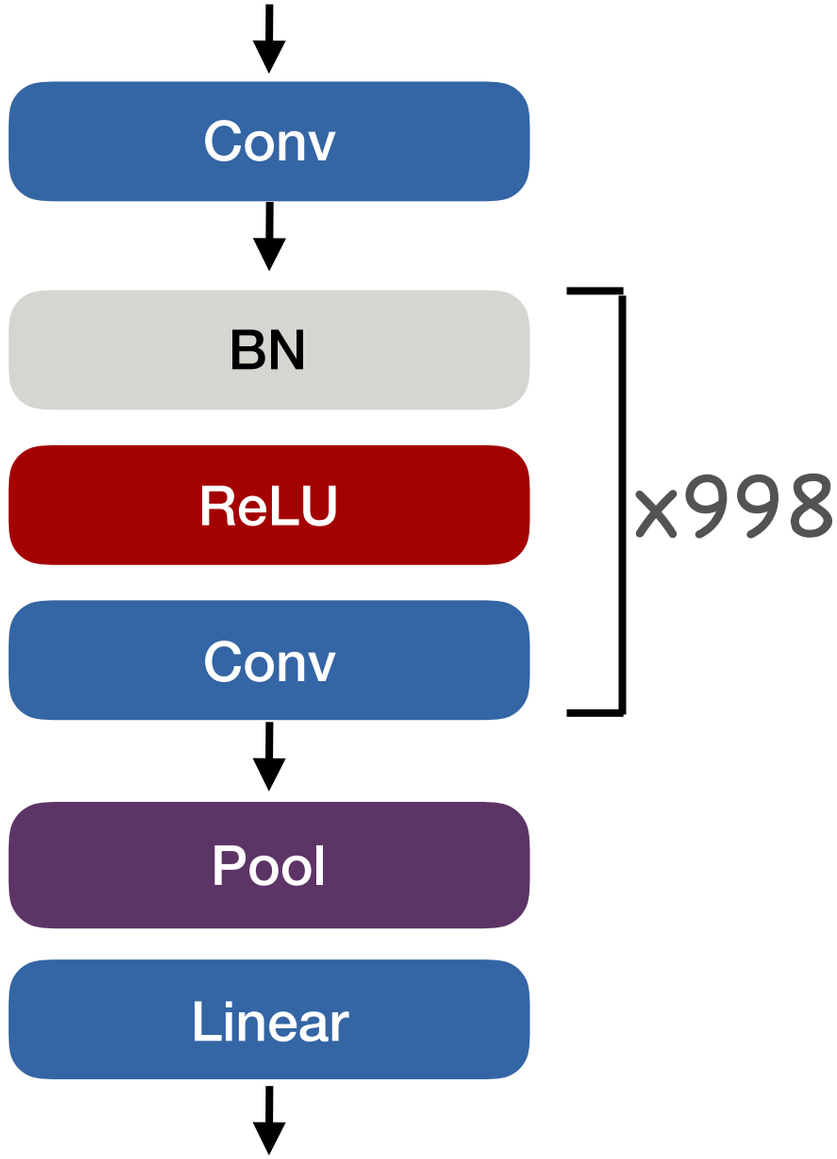
# Residual Networks




[Figure source: Kaiming He et al., "Deep Residual Learning for Image Recognition", CVPR 2016]

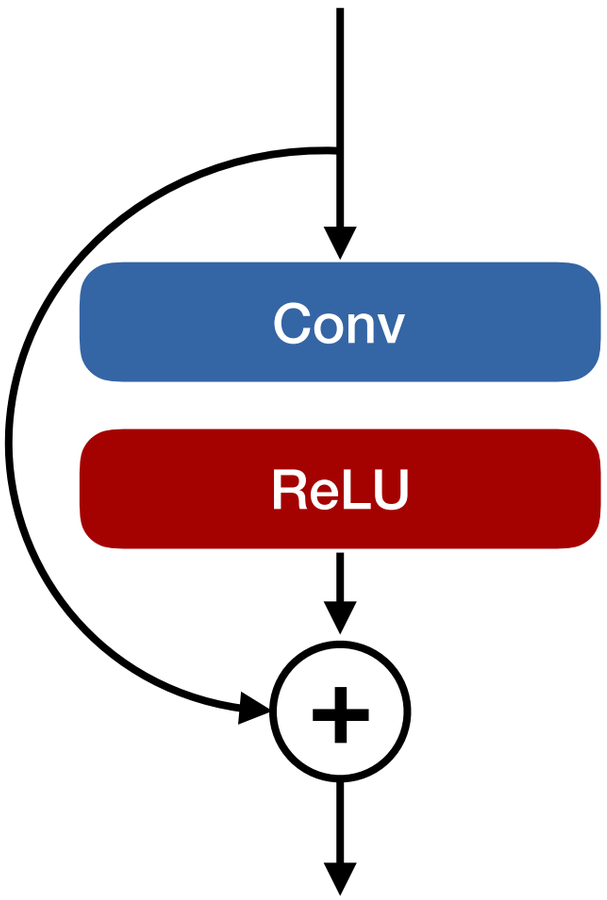
# How well do residual connections work?

- Can train networks of up to 1000 layers



# Why do residual connection work? - Practical answer

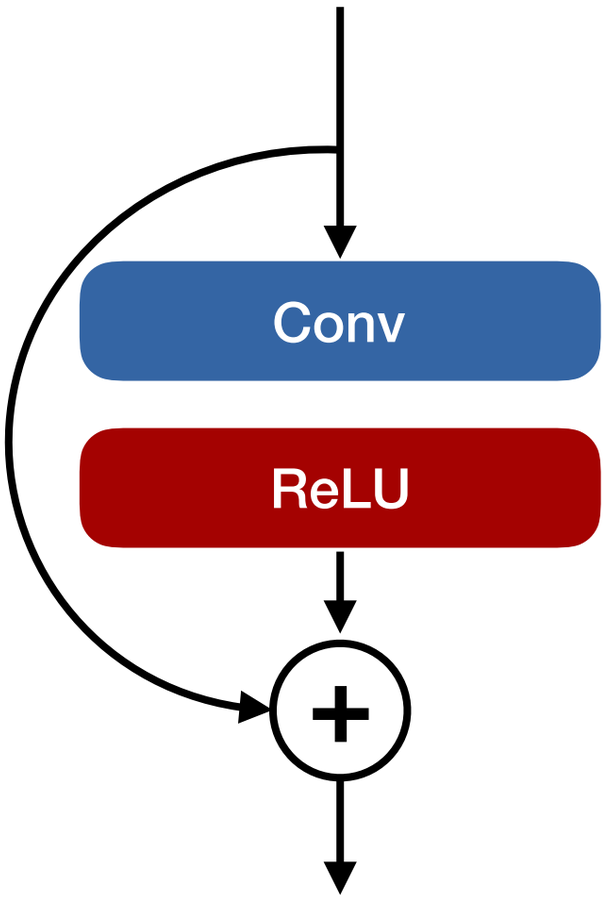
- Gradient travels further without major modifications (vanishing)
- Reuse of patters
  - Only update patterns
  - Dropping some layers does not even hurt performance
- Weights $\rightarrow 0$
  - Model $\rightarrow$ identity




[Gao Huang et al., "Deep Networks with Stochastic Depth", ECCV 2016]

# Why do residual connection work? - Theoretical answer

- Without ReLU
  - Invertible functions
- Very wide
  - SGD find global optimum




[Moritz Hardt and Tengyu Ma, “Identity matters in deep learning”, ICLR 2017]

[Simon S. Du, et al., "Gradient Descent Finds Global Minima of Deep Neural Networks", ICML 2019]

# Residual connections - Summary

- Used in most modern networks
- Allow for much deeper networks

# 18

January 23, 2024

[1]:
```
%pylab inline
import torch
import sys
sys.path.append('..')
sys.path.append('../..')
from data import load
device = torch.device('cuda') if torch.cuda.is_available() else torch.
 ↪device('cpu')
print('device = ', device)
```

```
%pylab is deprecated, use %matplotlib inline and import the required libraries.
Populating the interactive namespace from numpy and matplotlib
device =  cuda
```

[2]:
```
class ConvNet(torch.nn.Module):
    class Block(torch.nn.Module):
        def __init__(self, n_input, n_output, stride=1):
            super().__init__()
            self.net = torch.nn.Sequential(
              torch.nn.Conv2d(n_input, n_output, kernel_size=3, padding=1,␣
 ↪stride=stride, bias=False),
              torch.nn.BatchNorm2d(n_output),
              torch.nn.ReLU(),
              torch.nn.Conv2d(n_output, n_output, kernel_size=3, padding=1,␣
 ↪bias=False),
              torch.nn.BatchNorm2d(n_output),
              torch.nn.ReLU()
            )
            self.downsample = None
            if stride != 1 or n_input != n_output:
                self.downsample = torch.nn.Sequential(torch.nn.Conv2d(n_input,␣
 ↪n_output, 1),
                                                      torch.nn.
 ↪BatchNorm2d(n_output))

        def forward(self, x):
            identity = x
```

```
            if self.downsample is not None:
                identity = self.downsample(x)
            return self.net(x) + identity

    def __init__(self, layers=[32,64,128], n_input_channels=3):
        super().__init__()
        L = [torch.nn.Conv2d(n_input_channels, 32, kernel_size=7, padding=3,␣
↪stride=2, bias=False),
             torch.nn.BatchNorm2d(32),
             torch.nn.ReLU(),
             torch.nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
            ]
        c = 32
        for l in layers:
            L.append(self.Block(c, l, stride=2))
            c = l
        self.network = torch.nn.Sequential(*L)
        self.classifier = torch.nn.Linear(c, 1)

    def forward(self, x):
        # Compute the features
        z = self.network(x)
        # Global average pooling
        z = z.mean(dim=[2,3])
        # Classify
        return self.classifier(z)[:,0]

net = ConvNet()
net.train()
print( net.training )
net.eval()
print( net.training )
```

```
True
False
```

[ ]:

# Optimization algorithms

# Stochastic Gradient Descent with Momentum

- Default optimizer
- Works well in most cases
- Tune learning rate

for n epochs
  for $B_i$ batches

$$\mathbf{g} := \mathbb{E}_{\mathbf{x},y \in B_i}\left[\frac{\partial \ell(\mathbf{x}, y \mid \theta)}{\partial \theta}\right]$$

$$\mathbf{v} := \rho \mathbf{v} + \mathbf{g}$$

$$\theta := \theta - \epsilon \mathbf{v}$$

# RMSProp

- Very specialized
  - Auto-tunes learning rate
  - Momentum optional
  - Doesn't play nice with momentum
  - Works well on some reinforcement learning problems

$$\mathbf{m} := \mathbf{v} := 0$$

for n epochs

for $B_i$ batches

$$\mathbf{g} := \mathbb{E}_{\mathbf{x}, y \in B_i}\left[\frac{\partial \ell(\mathbf{x}, y \mid \theta)}{\partial \theta}\right]$$

$$\mathbf{m} := \alpha \mathbf{m} + (1 - \alpha)\mathbf{g}^2$$

$$\mathbf{v} := \rho \mathbf{v} + \frac{\mathbf{g}}{\sqrt{\mathbf{m} + \varepsilon}}$$

$$\theta := \theta - \epsilon \mathbf{v}$$

Tijmen Tieleman and Geoffrey Hinton. "Lecture 6.5-RMSProp: Divide the gradient by a running average of its recent magnitude." Neural networks for machine learning 4.2, 2012

# ADAM

- Less learning rate tuning
  - Works well on small networks and problems
  - Trains well, generalizes worse
  - Mathematically not correct

$$\mathbf{v} := \mathbf{m} := 0$$

for n epochs

for $B_i$ batches

$$\mathbf{g} := \mathbb{E}_{\mathbf{x}, y \in B_i}\left[\frac{\partial \ell(\mathbf{x}, y \mid \theta)}{\partial \theta}\right]$$

$$\mathbf{v} := \beta_1 \mathbf{v} + (1 - \beta_1)\mathbf{g}$$

$$\mathbf{m} := \beta_2 \mathbf{m} + (1 - \beta_2)\mathbf{g}^2$$

$$s := \epsilon \frac{\sqrt{1 - \beta_2^{\text{step}}}}{1 - \beta_1^{\text{step}}}$$

$$\theta := \theta - s \frac{\mathbf{v}}{\sqrt{\mathbf{m}} + \varepsilon}$$

Kingma, D. P., & Ba, J. L.. Adam: a Method for Stochastic Optimization. ICLR 2015

# What optimizer to use?

- Large models and data
  - SGD with momentum
- Small models and data
  - ADAM

# Learning rate

# Optimization algorithms

- Hyper parameters
- Learning rate
- Momentum
- Batch size

# What learning rate it use?

- Rule of thumb: Largest LR that trains
- Train for a few epochs and measure validation accuracy

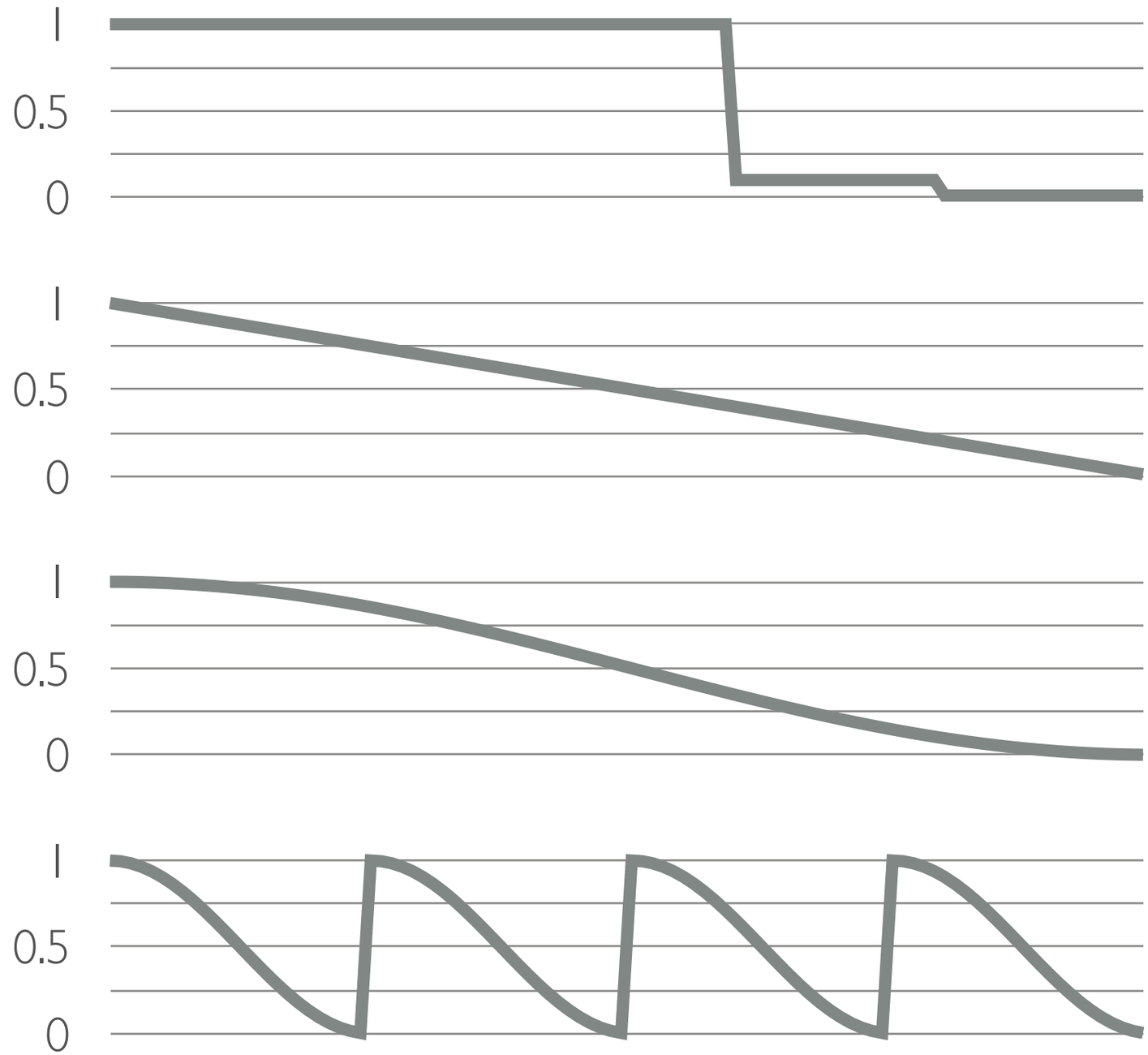
# Learning rate vs batch size

- **Linear Scaling Rule**:
  When the minibatch size is multiplied by $k$ , multiply the learning rate by $k$ .

Priya Goyal et al., "Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour", arXiv 2017

# Learning rate schedules

- Step schedule
- Linear schedule
- Cosine schedule
- Cyclical schedules



# Summary

- Use close to the largest LR that trains

- Step schedule

# 21

January 23, 2024

[1]:
```
%pylab inline
import torch
import sys
sys.path.append('..')
sys.path.append('../..')
from data import load
device = torch.device('cuda') if torch.cuda.is_available() else torch.
 ↪device('cpu')
print('device = ', device)
```

```
%pylab is deprecated, use %matplotlib inline and import the required libraries.
Populating the interactive namespace from numpy and matplotlib
device =  cuda
```

[2]:
```
class ConvNet(torch.nn.Module):
    class Block(torch.nn.Module):
        def __init__(self, n_input, n_output, stride=1):
            super().__init__()
            self.net = torch.nn.Sequential(
              torch.nn.Conv2d(n_input, n_output, kernel_size=3, padding=1,␣
 ↪stride=stride, bias=False),
              torch.nn.BatchNorm2d(n_output),
              torch.nn.ReLU(),
              torch.nn.Conv2d(n_output, n_output, kernel_size=3, padding=1,␣
 ↪bias=False),
              torch.nn.BatchNorm2d(n_output),
              torch.nn.ReLU()
            )
            self.downsample = None
            if stride != 1 or n_input != n_output:
                self.downsample = torch.nn.Sequential(torch.nn.Conv2d(n_input,␣
 ↪n_output, 1, stride=stride),
                                                      torch.nn.
 ↪BatchNorm2d(n_output))

        def forward(self, x):
            identity = x
```

```
            if self.downsample is not None:
                identity = self.downsample(x)
            return self.net(x) + identity

    def __init__(self, layers=[32,64,128], n_input_channels=3):
        super().__init__()
        L = [torch.nn.Conv2d(n_input_channels, 32, kernel_size=7, padding=3,␣
 ↪stride=2, bias=False),
             torch.nn.BatchNorm2d(32),
             torch.nn.ReLU(),
             torch.nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
            ]
        c = 32
        for l in layers:
            L.append(self.Block(c, l, stride=2))
            c = l
        self.network = torch.nn.Sequential(*L)
        self.classifier = torch.nn.Linear(c, 1)

    def forward(self, x):
        # Compute the features
        z = self.network(x)
        # Global average pooling
        z = z.mean(dim=[2,3])
        # Classify
        return self.classifier(z)[:,0]

model = ConvNet()
model.train()
print( model.training )
model.eval()
print( model.training )
```

```
True
False
```

[4]:
```
n_epochs = 100

train_data = load.get_dogs_and_cats(resize=(128,128), batch_size=32,␣
 ↪is_resnet=True)
optimizer = torch.optim.Adam(model.parameters())
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=30, gamma=0.1)
scheduler = torch.optim.lr_scheduler.MultiStepLR(optimizer, [50,75], gamma=0.1)
scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer, 'max')

# Transfer the data to a GPU (optional)
model = model.to(device)
```

```
# Construct the loss and accuracy functions
loss = torch.nn.BCEWithLogitsLoss()
accuracy = lambda o, l: ((o > 0).long() == l.long()).float()

# Train the network
for epoch in range(n_epochs):
    model.train()
    accuracies = []
    for it, (data, label) in enumerate(train_data):
        # Transfer the data to a GPU (optional)
        data, label = data.to(device), label.to(device)

        # Produce the output
        o = model(data)

        # Compute the loss and accuracy
        loss_val = loss(o, label.float())
        accuracies.extend(accuracy(o, label).detach().cpu().numpy())

        # Take a gradient step
        optimizer.zero_grad()
        loss_val.backward()
        optimizer.step()
        break
#     scheduler.step()
    scheduler.step(np.mean(accuracies))
    print( 'epoch = ', epoch, 'optimizer_lr', optimizer.param_groups[0]['lr'],␣
 ↪'accuracy', np.mean(accuracies))
```

```
epoch =  0 optimizer_lr 0.001 accuracy 0.46875
epoch =  1 optimizer_lr 0.001 accuracy 0.65625
epoch =  2 optimizer_lr 0.001 accuracy 0.96875
epoch =  3 optimizer_lr 0.001 accuracy 0.96875
epoch =  4 optimizer_lr 0.001 accuracy 1.0
epoch =  5 optimizer_lr 0.001 accuracy 1.0
epoch =  6 optimizer_lr 0.001 accuracy 1.0
epoch =  7 optimizer_lr 0.001 accuracy 1.0
epoch =  8 optimizer_lr 0.001 accuracy 1.0
epoch =  9 optimizer_lr 0.001 accuracy 1.0
epoch =  10 optimizer_lr 0.001 accuracy 1.0
epoch =  11 optimizer_lr 0.001 accuracy 1.0
epoch =  12 optimizer_lr 0.001 accuracy 1.0
epoch =  13 optimizer_lr 0.001 accuracy 1.0
epoch =  14 optimizer_lr 0.001 accuracy 1.0
epoch =  15 optimizer_lr 0.0001 accuracy 1.0
epoch =  16 optimizer_lr 0.0001 accuracy 1.0
```

```
epoch =  17 optimizer_lr 0.0001 accuracy 1.0
epoch =  18 optimizer_lr 0.0001 accuracy 1.0
epoch =  19 optimizer_lr 0.0001 accuracy 1.0
epoch =  20 optimizer_lr 0.0001 accuracy 1.0
epoch =  21 optimizer_lr 0.0001 accuracy 1.0
epoch =  22 optimizer_lr 0.0001 accuracy 1.0
epoch =  23 optimizer_lr 0.0001 accuracy 1.0
epoch =  24 optimizer_lr 0.0001 accuracy 1.0
epoch =  25 optimizer_lr 0.0001 accuracy 1.0
epoch =  26 optimizer_lr 1e-05 accuracy 1.0
epoch =  27 optimizer_lr 1e-05 accuracy 1.0
epoch =  28 optimizer_lr 1e-05 accuracy 1.0
epoch =  29 optimizer_lr 1e-05 accuracy 1.0
epoch =  30 optimizer_lr 1e-05 accuracy 1.0
epoch =  31 optimizer_lr 1e-05 accuracy 1.0
epoch =  32 optimizer_lr 1e-05 accuracy 1.0
epoch =  33 optimizer_lr 1e-05 accuracy 1.0
epoch =  34 optimizer_lr 1e-05 accuracy 1.0
epoch =  35 optimizer_lr 1e-05 accuracy 1.0
epoch =  36 optimizer_lr 1e-05 accuracy 1.0
epoch =  37 optimizer_lr 1.0000000000000002e-06 accuracy 1.0
epoch =  38 optimizer_lr 1.0000000000000002e-06 accuracy 1.0
epoch =  39 optimizer_lr 1.0000000000000002e-06 accuracy 1.0
epoch =  40 optimizer_lr 1.0000000000000002e-06 accuracy 1.0
epoch =  41 optimizer_lr 1.0000000000000002e-06 accuracy 1.0
epoch =  42 optimizer_lr 1.0000000000000002e-06 accuracy 1.0
epoch =  43 optimizer_lr 1.0000000000000002e-06 accuracy 1.0
epoch =  44 optimizer_lr 1.0000000000000002e-06 accuracy 1.0
epoch =  45 optimizer_lr 1.0000000000000002e-06 accuracy 1.0
epoch =  46 optimizer_lr 1.0000000000000002e-06 accuracy 1.0
epoch =  47 optimizer_lr 1.0000000000000002e-06 accuracy 1.0
epoch =  48 optimizer_lr 1.0000000000000002e-07 accuracy 1.0
epoch =  49 optimizer_lr 1.0000000000000002e-07 accuracy 1.0
epoch =  50 optimizer_lr 1.0000000000000002e-07 accuracy 1.0
epoch =  51 optimizer_lr 1.0000000000000002e-07 accuracy 1.0
epoch =  52 optimizer_lr 1.0000000000000002e-07 accuracy 1.0
epoch =  53 optimizer_lr 1.0000000000000002e-07 accuracy 1.0
epoch =  54 optimizer_lr 1.0000000000000002e-07 accuracy 1.0
epoch =  55 optimizer_lr 1.0000000000000002e-07 accuracy 1.0
epoch =  56 optimizer_lr 1.0000000000000002e-07 accuracy 1.0
epoch =  57 optimizer_lr 1.0000000000000002e-07 accuracy 1.0
epoch =  58 optimizer_lr 1.0000000000000002e-07 accuracy 1.0
epoch =  59 optimizer_lr 1.0000000000000004e-08 accuracy 1.0
epoch =  60 optimizer_lr 1.0000000000000004e-08 accuracy 1.0
epoch =  61 optimizer_lr 1.0000000000000004e-08 accuracy 1.0
epoch =  62 optimizer_lr 1.0000000000000004e-08 accuracy 1.0
epoch =  63 optimizer_lr 1.0000000000000004e-08 accuracy 1.0
epoch =  64 optimizer_lr 1.0000000000000004e-08 accuracy 1.0
```

```
epoch =  65 optimizer_lr 1.0000000000000004e-08 accuracy 1.0
epoch =  66 optimizer_lr 1.0000000000000004e-08 accuracy 1.0
epoch =  67 optimizer_lr 1.0000000000000004e-08 accuracy 1.0
epoch =  68 optimizer_lr 1.0000000000000004e-08 accuracy 1.0
epoch =  69 optimizer_lr 1.0000000000000004e-08 accuracy 1.0
epoch =  70 optimizer_lr 1.0000000000000004e-08 accuracy 1.0
epoch =  71 optimizer_lr 1.0000000000000004e-08 accuracy 1.0
epoch =  72 optimizer_lr 1.0000000000000004e-08 accuracy 1.0
epoch =  73 optimizer_lr 1.0000000000000004e-08 accuracy 1.0
epoch =  74 optimizer_lr 1.0000000000000004e-08 accuracy 1.0
epoch =  75 optimizer_lr 1.0000000000000004e-08 accuracy 1.0
epoch =  76 optimizer_lr 1.0000000000000004e-08 accuracy 1.0
epoch =  77 optimizer_lr 1.0000000000000004e-08 accuracy 1.0
epoch =  78 optimizer_lr 1.0000000000000004e-08 accuracy 1.0
epoch =  79 optimizer_lr 1.0000000000000004e-08 accuracy 1.0
epoch =  80 optimizer_lr 1.0000000000000004e-08 accuracy 1.0
epoch =  81 optimizer_lr 1.0000000000000004e-08 accuracy 1.0
epoch =  82 optimizer_lr 1.0000000000000004e-08 accuracy 1.0
epoch =  83 optimizer_lr 1.0000000000000004e-08 accuracy 1.0
epoch =  84 optimizer_lr 1.0000000000000004e-08 accuracy 1.0
epoch =  85 optimizer_lr 1.0000000000000004e-08 accuracy 1.0
epoch =  86 optimizer_lr 1.0000000000000004e-08 accuracy 1.0
epoch =  87 optimizer_lr 1.0000000000000004e-08 accuracy 1.0
epoch =  88 optimizer_lr 1.0000000000000004e-08 accuracy 1.0
epoch =  89 optimizer_lr 1.0000000000000004e-08 accuracy 1.0
epoch =  90 optimizer_lr 1.0000000000000004e-08 accuracy 1.0
epoch =  91 optimizer_lr 1.0000000000000004e-08 accuracy 1.0
epoch =  92 optimizer_lr 1.0000000000000004e-08 accuracy 1.0
epoch =  93 optimizer_lr 1.0000000000000004e-08 accuracy 1.0
epoch =  94 optimizer_lr 1.0000000000000004e-08 accuracy 1.0
epoch =  95 optimizer_lr 1.0000000000000004e-08 accuracy 1.0
epoch =  96 optimizer_lr 1.0000000000000004e-08 accuracy 1.0
epoch =  97 optimizer_lr 1.0000000000000004e-08 accuracy 1.0
epoch =  98 optimizer_lr 1.0000000000000004e-08 accuracy 1.0
epoch =  99 optimizer_lr 1.0000000000000004e-08 accuracy 1.0
```

[ ]:

# Open Problem: Pruning and compression

# How do we train a small network?

- Idea 1:
  - Randomly initialize and train network
- Idea 2:
  - Train a larger network and make it small

# Network distillation

- Train an ensemble of large networks
  - Train a small network to mimic it's output (with cross entropy)
  - Important: Reduce confidence of ensemble prediction (soft targets)

Hinton, Geoffrey, Oriol Vinyals, and Jeff Dean. "Distilling the knowledge in a neural network." arXiv 2015.

# Why does distillation work?

- Dark knowledge
- Networks learn about (visual) relationships of classes
- Boost training signal

# Network pruning / factorization

- Train a wide network (many channels)
  - Remove channels/weights that are used the least
    - 90% of parameters can be removed after training
    - Training the small network is challenging

H Li, A Kadav, I Durdanovic, H Samet, HP Graf, "Pruning Filters for Efficient ConvNets", ICLR 2017
S Han, J Pool, J Tran, W Dally, "Learning both Weights and Connections for Efficient Neural Network" NIPS 2015

# Possible explanation: Lottery ticket hypothesis

- Not all initializations are created even
- Train network

A randomly-initialized, dense neural network contains a subnetwork that is initialized such that—when trained in isolation—it can match the test accuracy of the original network after training for at most the same number of iterations.

Jonathan Frankle, Michael Carbin, "The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks", ICLR 2019

# Lottery ticket hypothesis

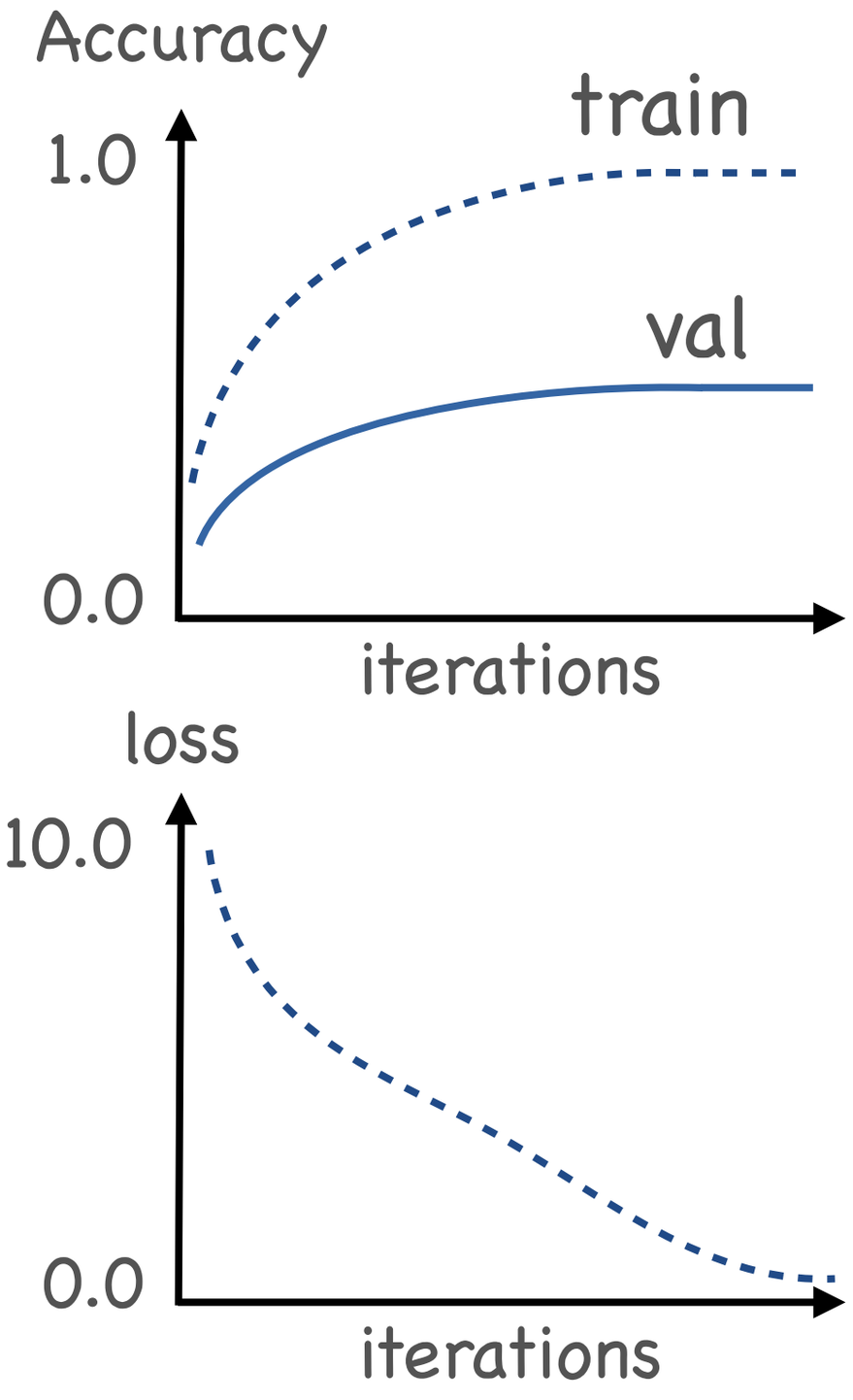
- Very nice idea
- Likely not the full story

Zhuang Liu et al., "Rethinking the Value of Network Pruning", ICLR 2019

# Overfitting and how to detect it

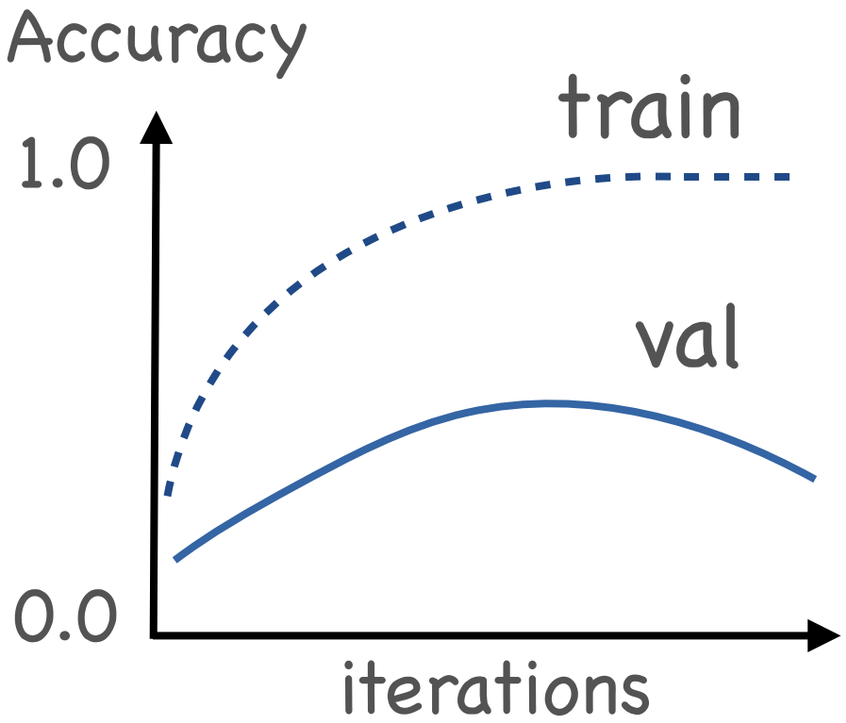
# Overfitting

- Fit model to training set
- Model does not work on unseen data (e.g. validation set)

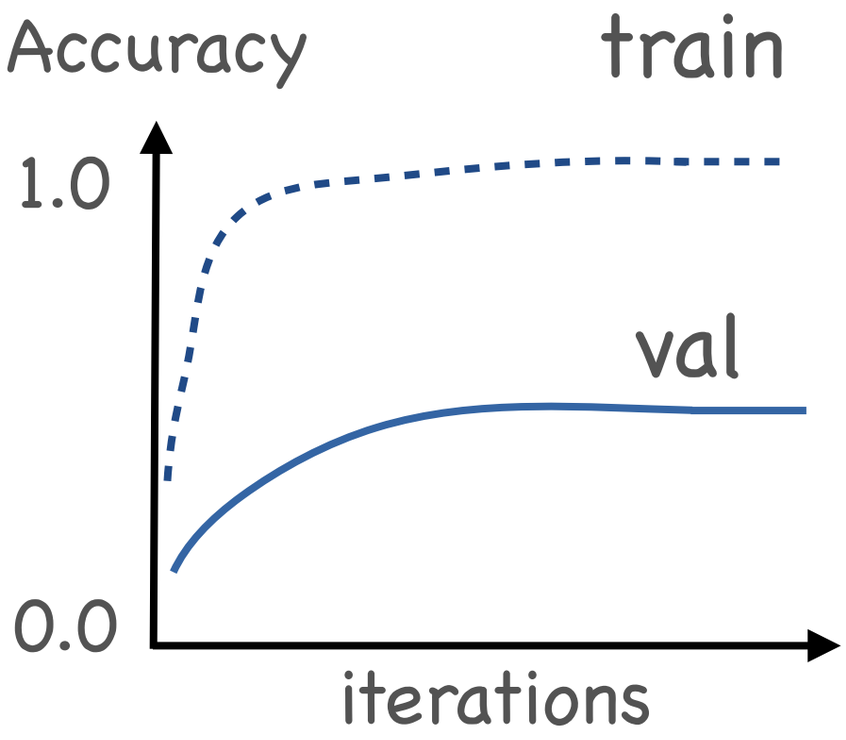


# How to detect overfitting?

- Plot training and validation accuracy

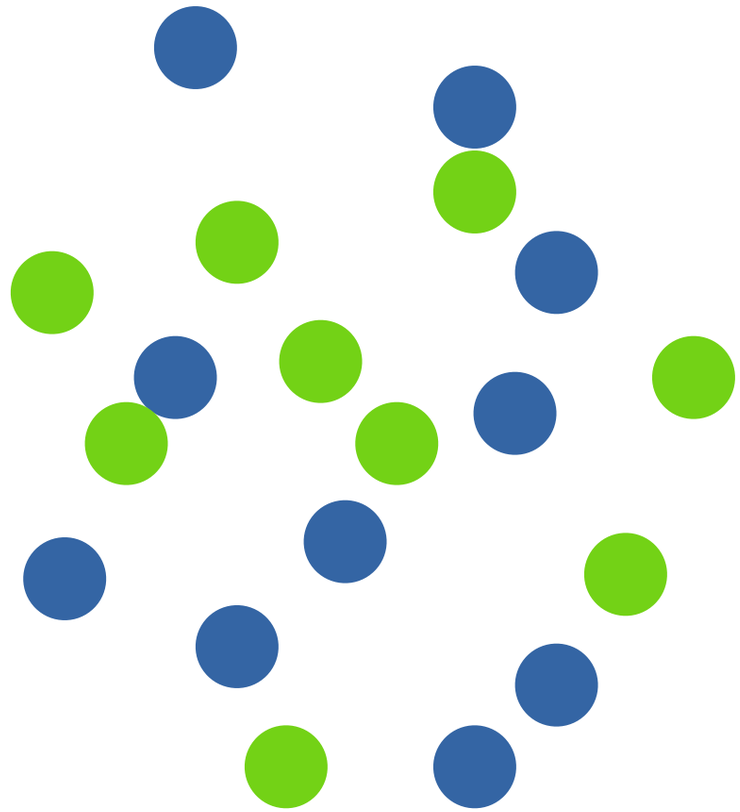


# Is overfitting always bad?



# Why do we overfit?

- Sampling bias
  - Optimization fits patterns that only exist in training set



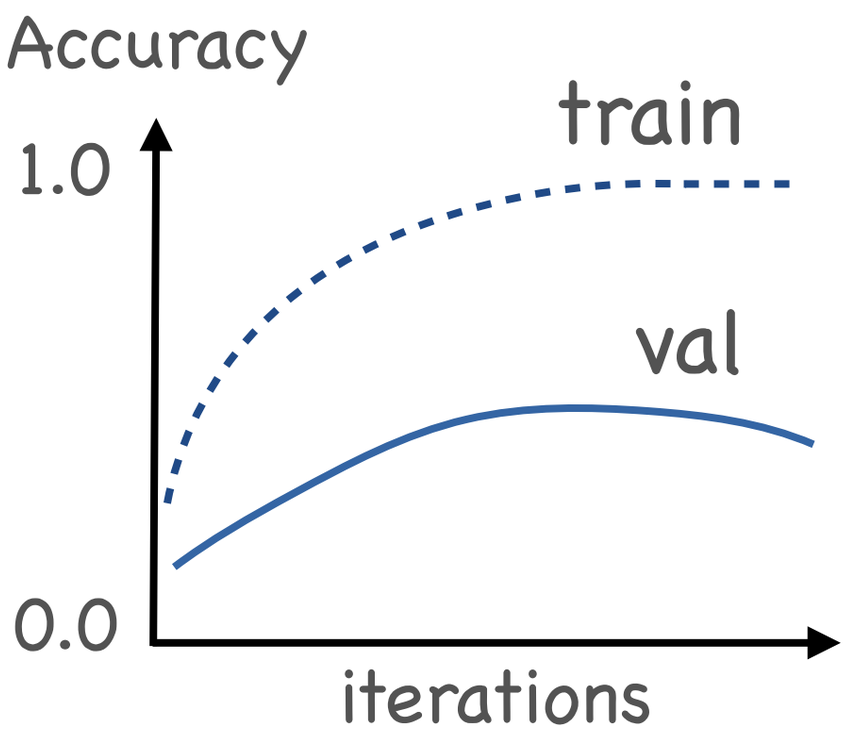
# Do we overfit with infinite training data?

# How do we prevent overfitting?

- Collect more data
- Make the model simpler
  - Regularize
- Transfer learning

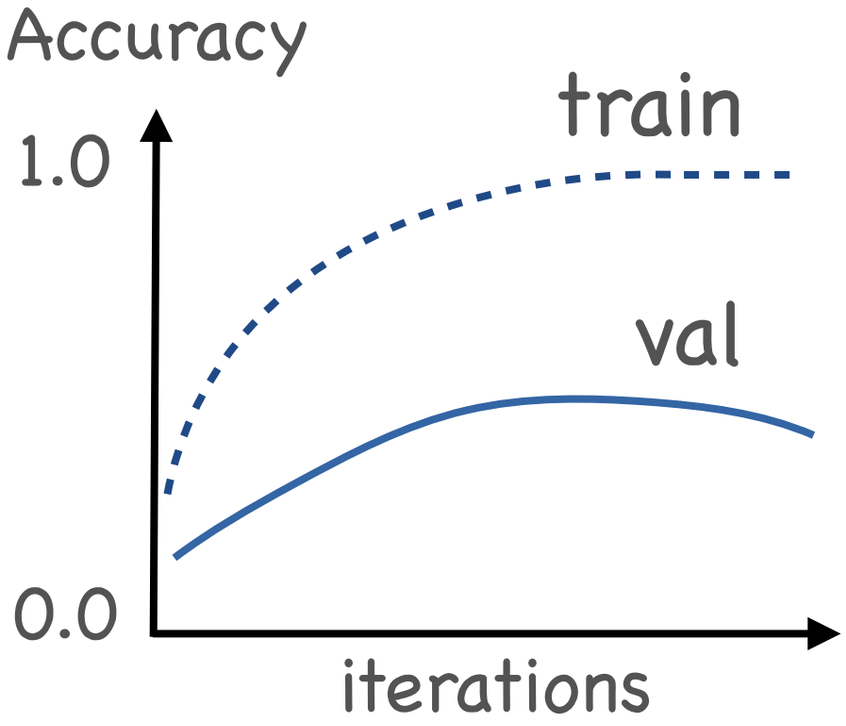
# Early stopping

# Training and overfitting



# Early stopping in practice

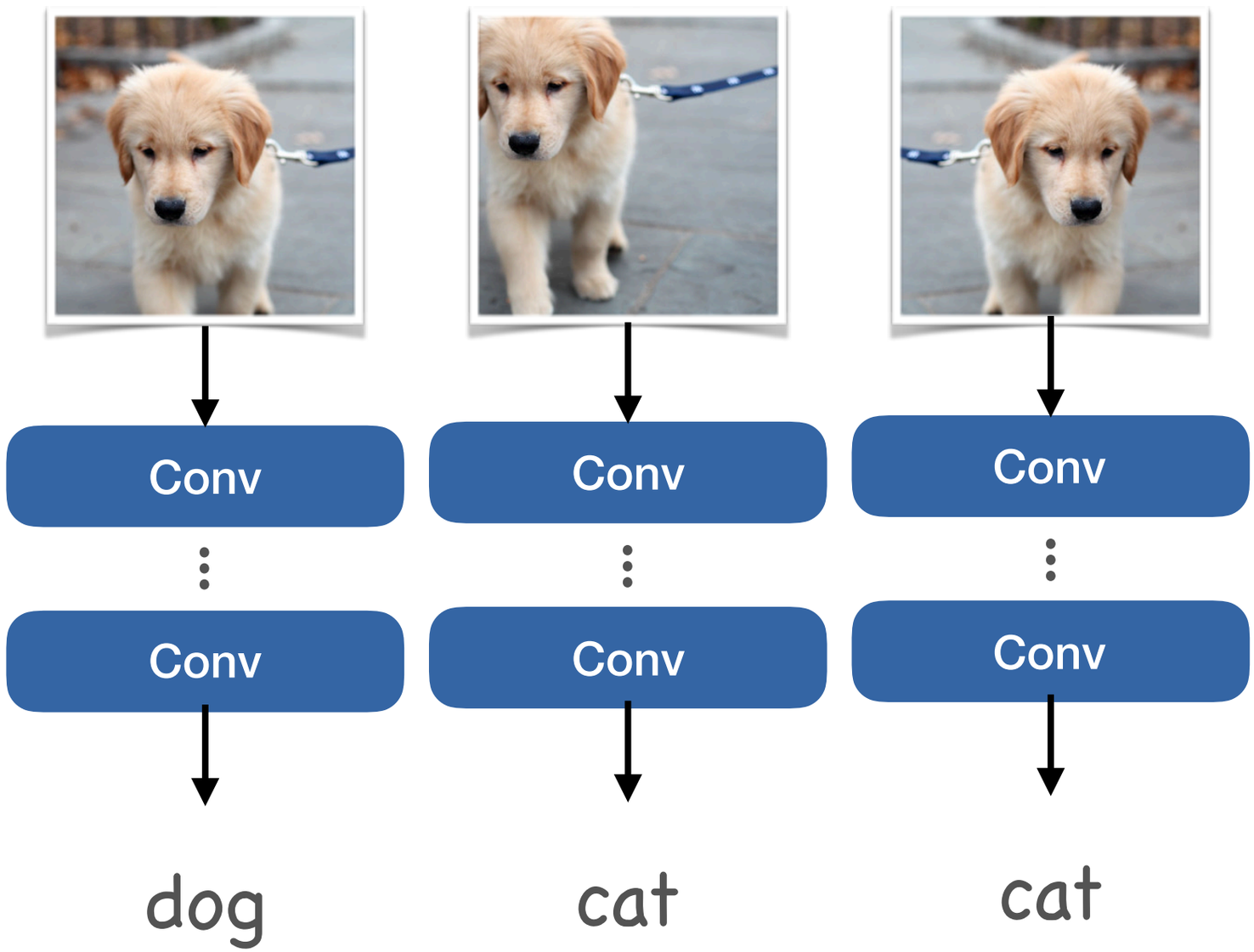
- No need for stop button
- Measure validation accuracy periodically
- Save your model periodically



# Data augmentation

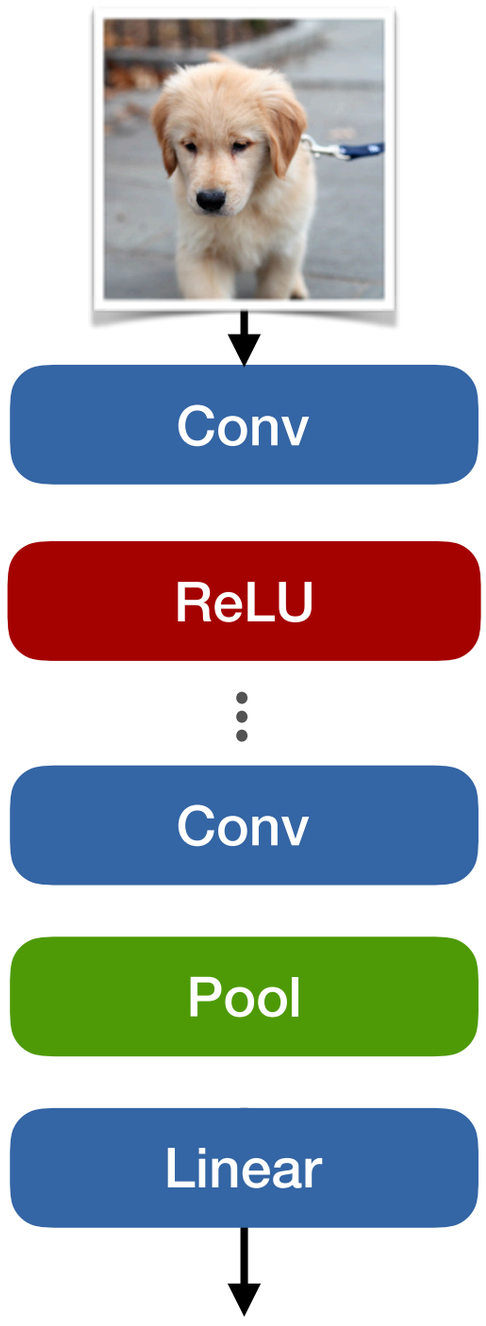
# Signs of overfitting

- Does not capture invariances in data



# How to capture invariances?

- Build them into the model
  - Convolutions
  - All-convolutional models
- Build them into the data
  - Data augmentation



# Data augmentation

- Capture invariances in data
  - (Randomly) transform data during training
- Reuse a label

(  , dog)

(  , dog)

(  , dog)

(  , dog)

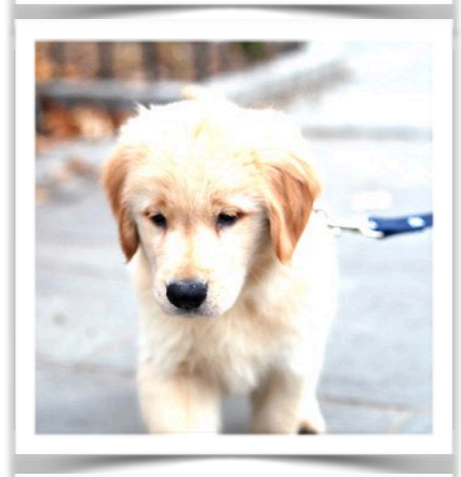
# Image augmentations

flip

saturation

shift

brightness



scale

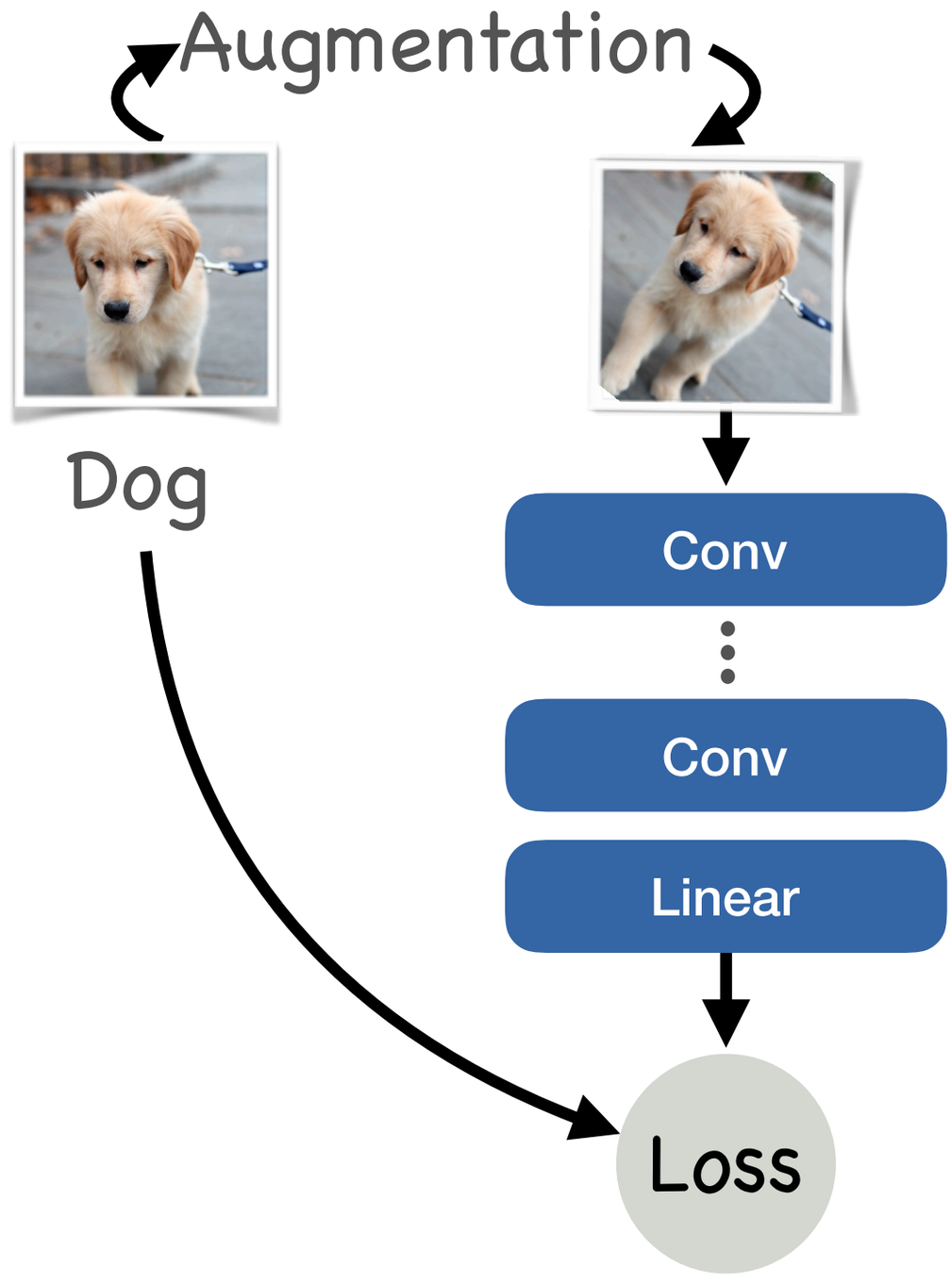
rotate

tint/hue

# Training with data augmentation

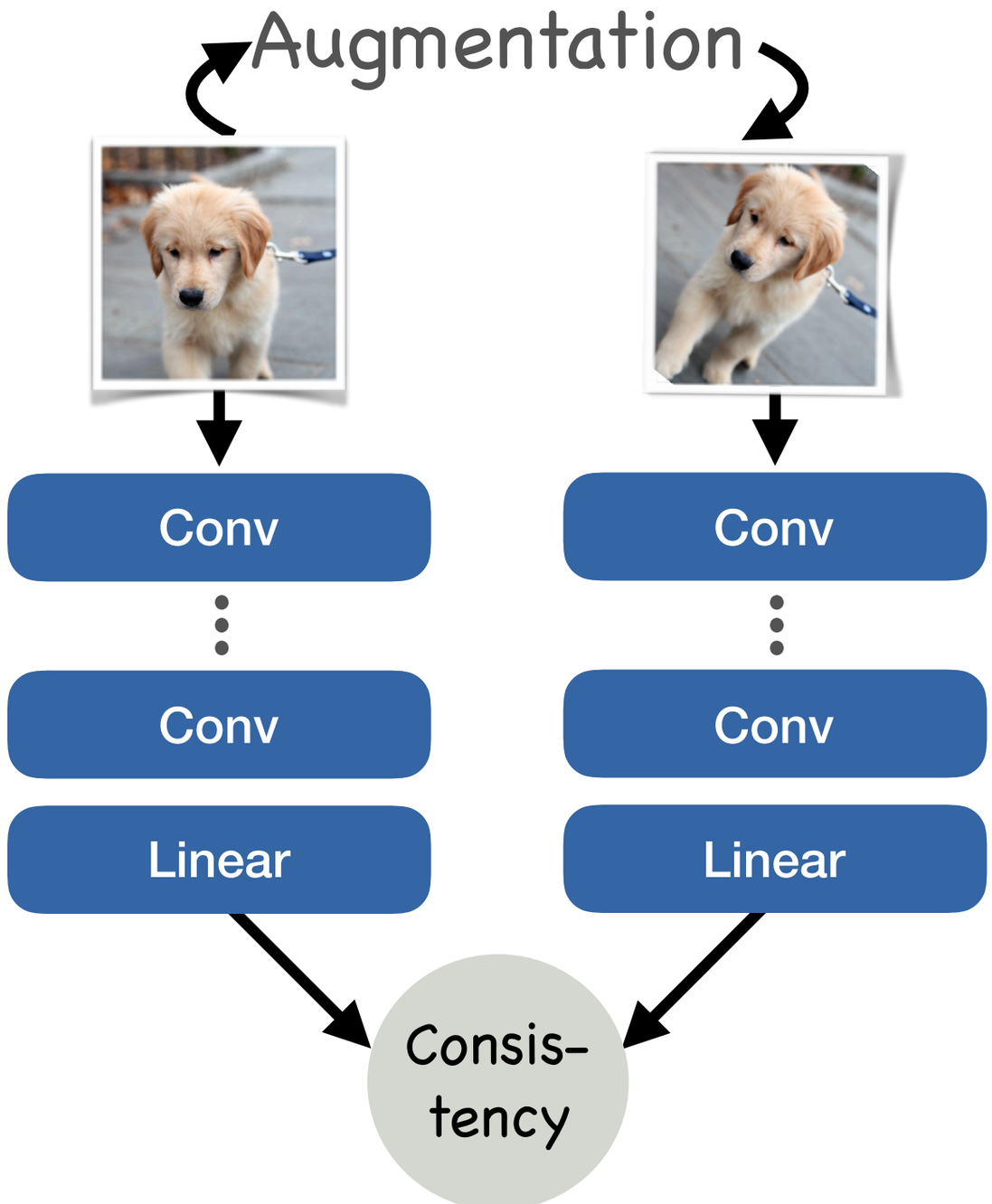
- (Randomly) augment every single iteration

- Network never sees exact same data twice



# Unsupervised data augmentation

- Captures invariances on unseen and unlabeled data




Xie, Dai, Hovy, Luong, Le,
"Unsupervised Data Augmentation",
arXiv 2019

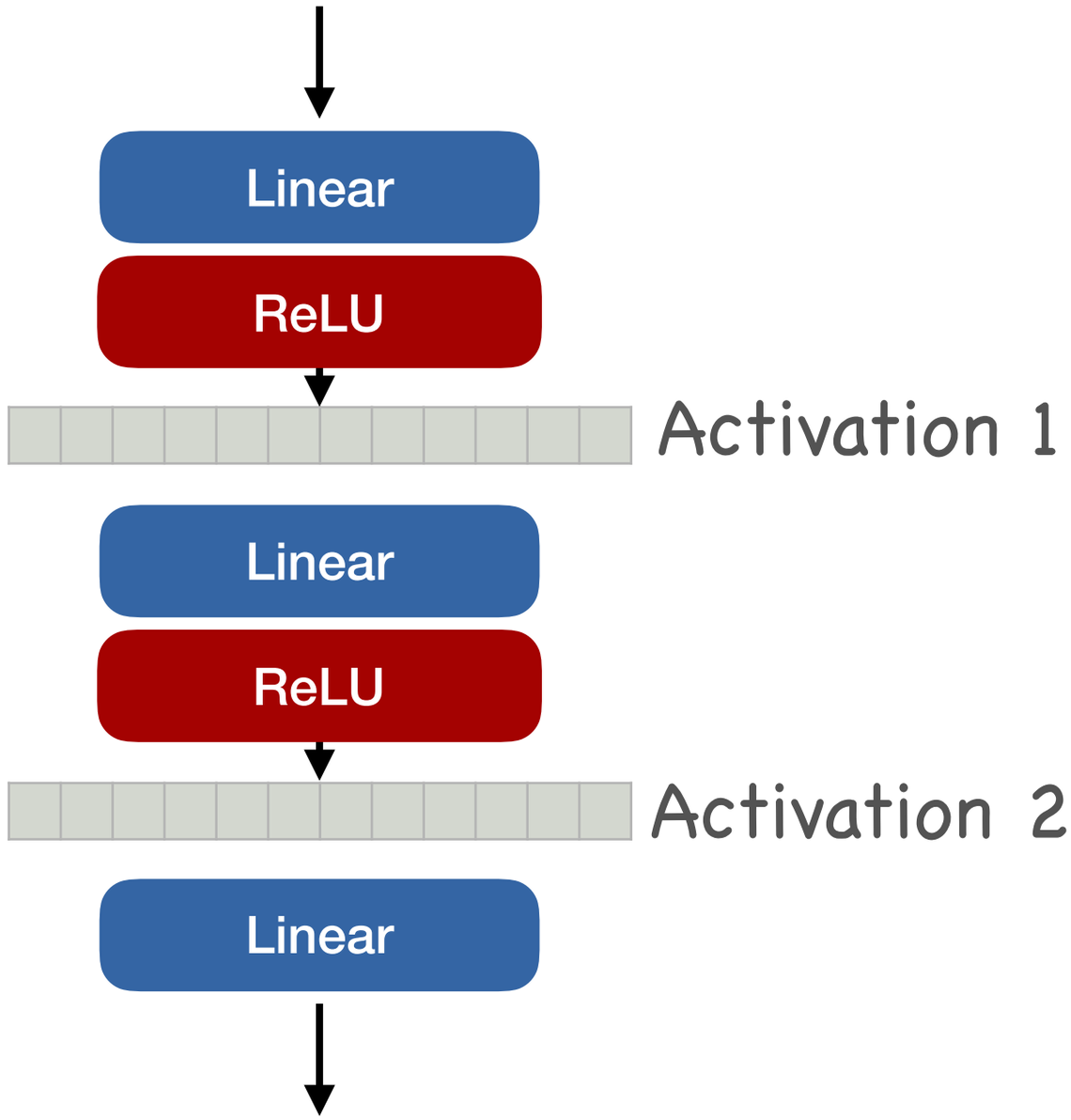
# Data augmentation

- Always use data augmentation if possible
- Some augmentations require augmentation of labels
  - e.g. for dense prediction tasks

# Dropout

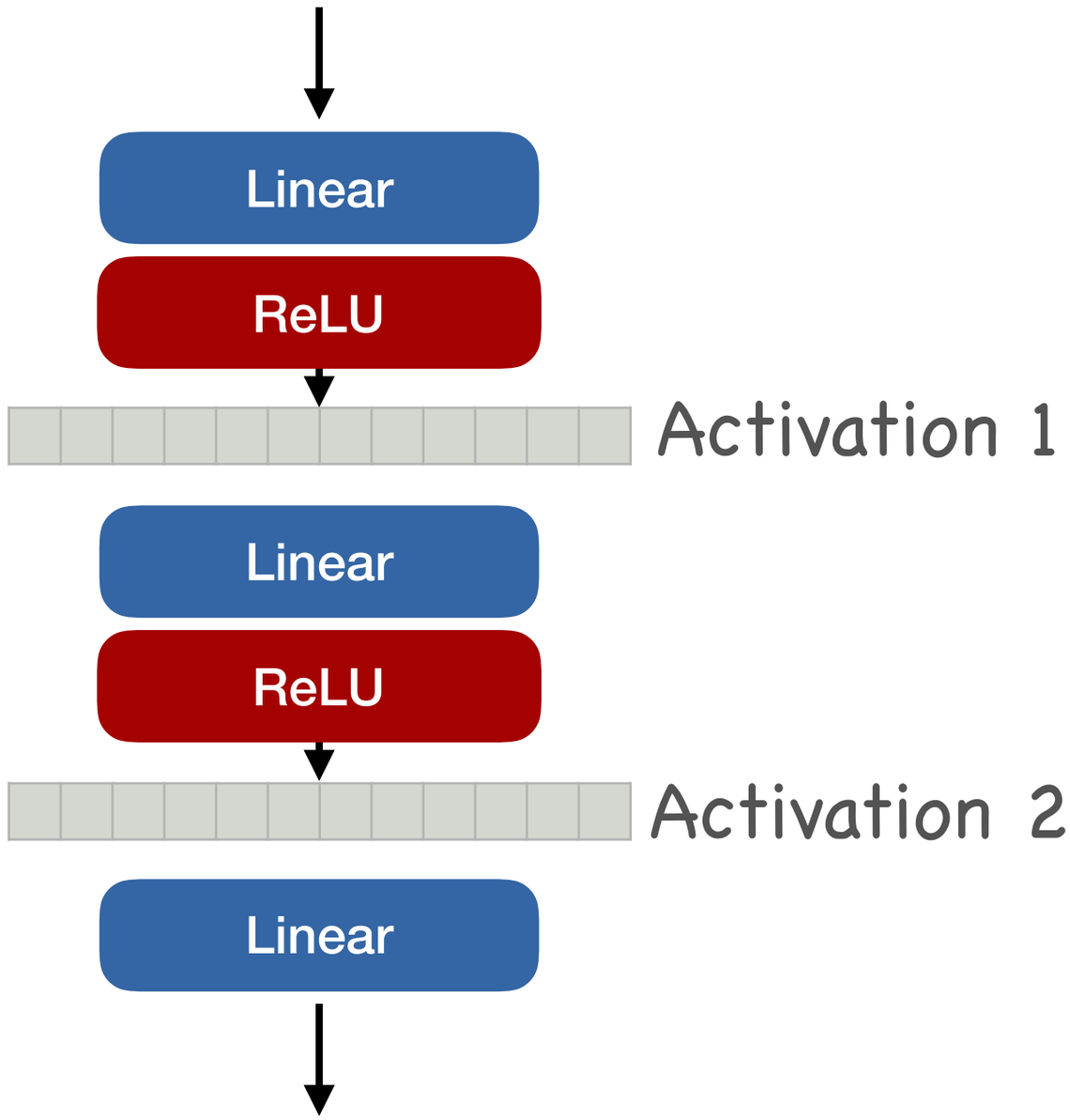
# Overfitting in deep networks

- Overfitting
  - Exploit patterns that exist in training data, but not in the validation / test data
- Not all activations overfit



# Overfitting in deep networks

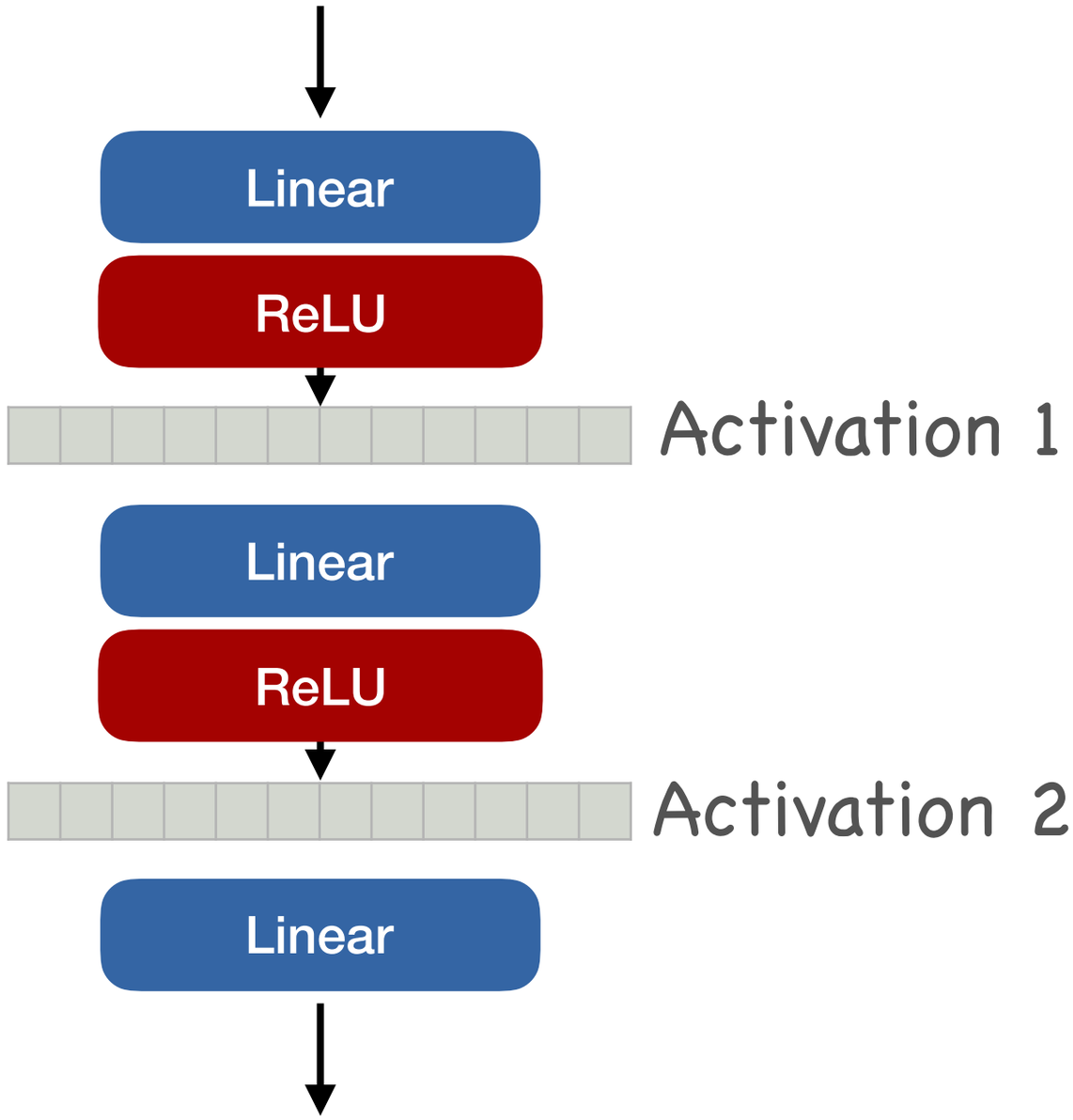
- Deeper layers overfit more
- Rely on overfit activations from previous layers



# Preventing overfitting in deep networks

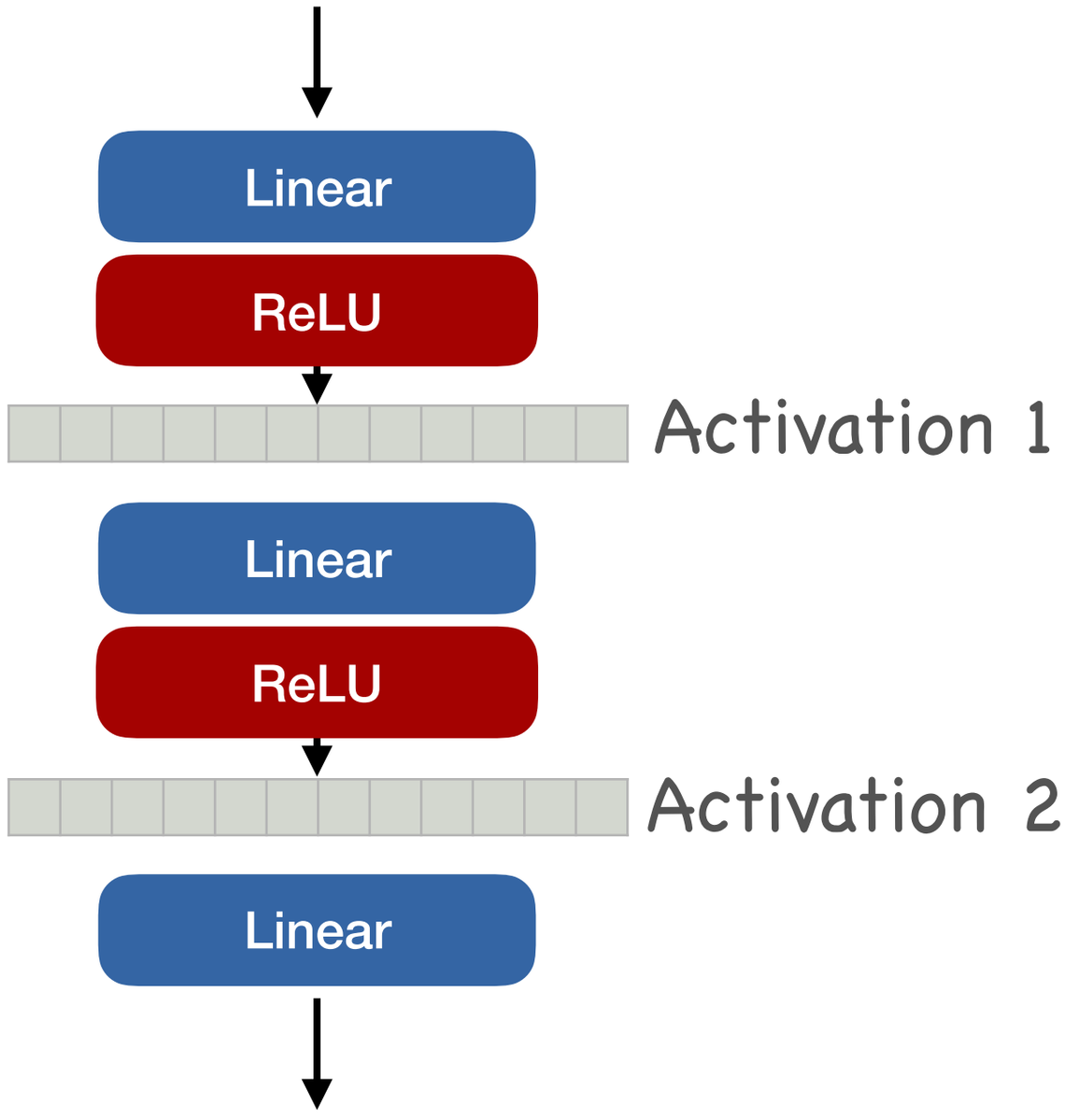
- Reduct reliance on specific activations in previous layer

- Randomly remove activations



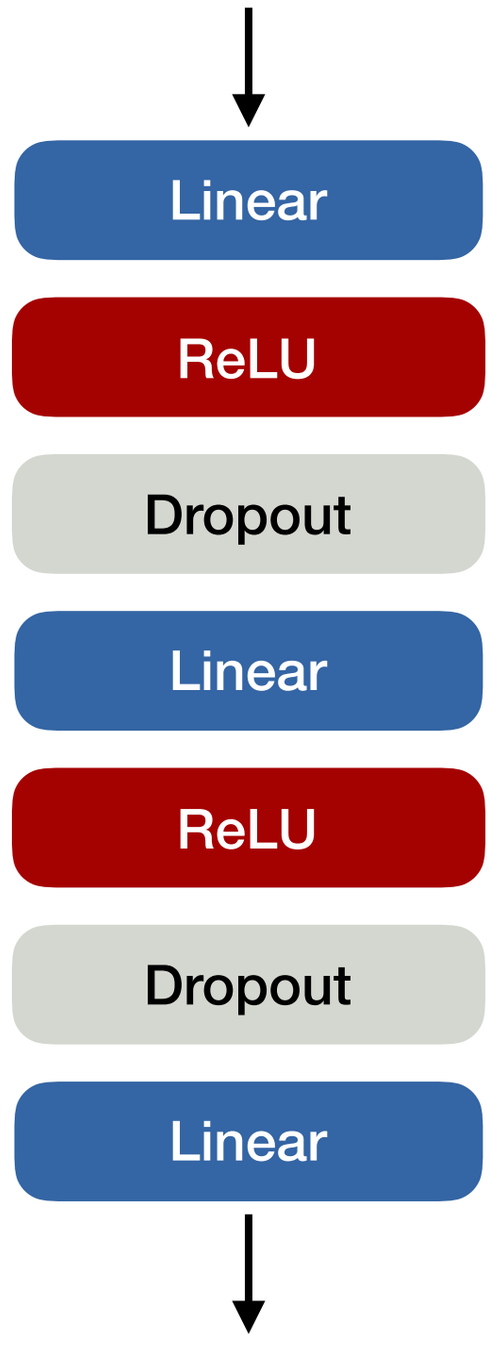
# Dropout

- During training
  - With probability $\alpha$ set activation $a_l(i)$ to zero
- During evaluation
  - Use all activations, but scale by $1 - \alpha$



# Dropout in practice

- A separate layer torch.nn.Dropout
- During training
  - With probability $\alpha$ set activation $a_l(i)$ to zero
  - Scale activations by $\frac{1}{1-\alpha}$
- During evaluation identity

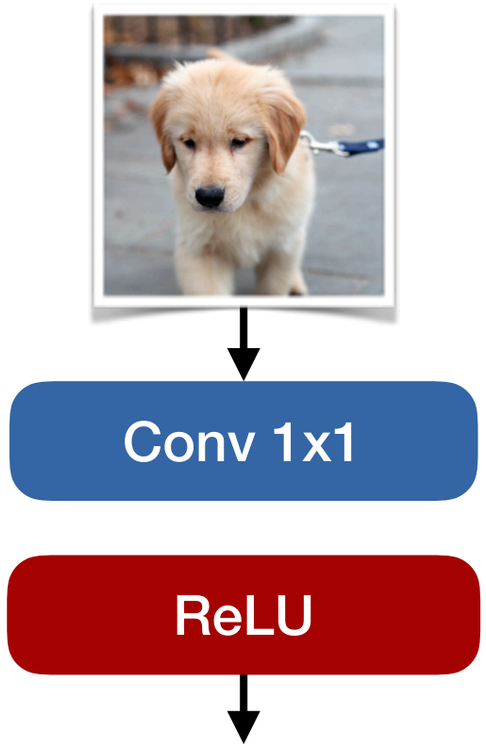


# Where to add dropout?

- Before any large fully connected layer

- Before some 1x1 convolutions

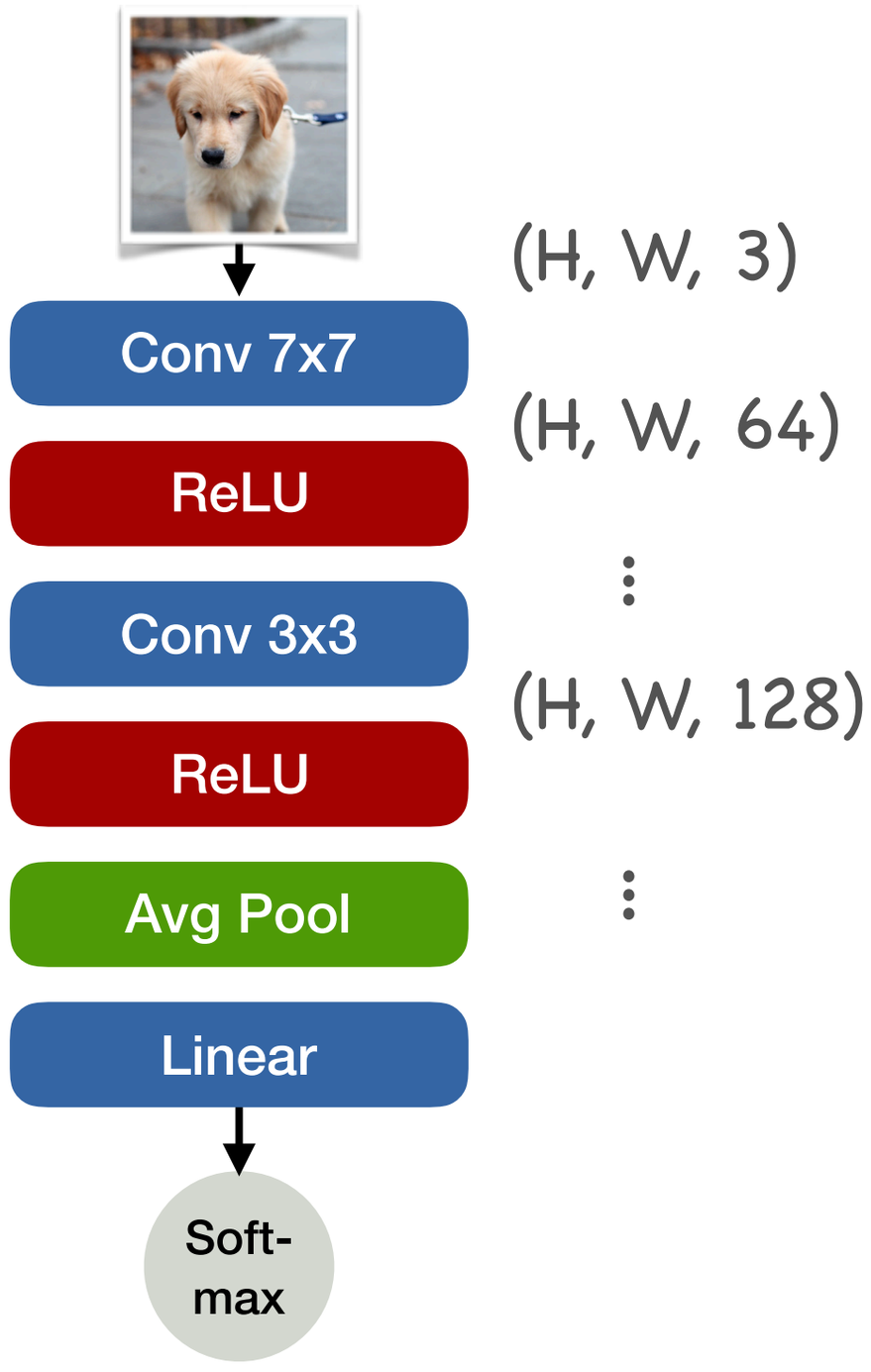
- Not before general convolutions



# Weight decay

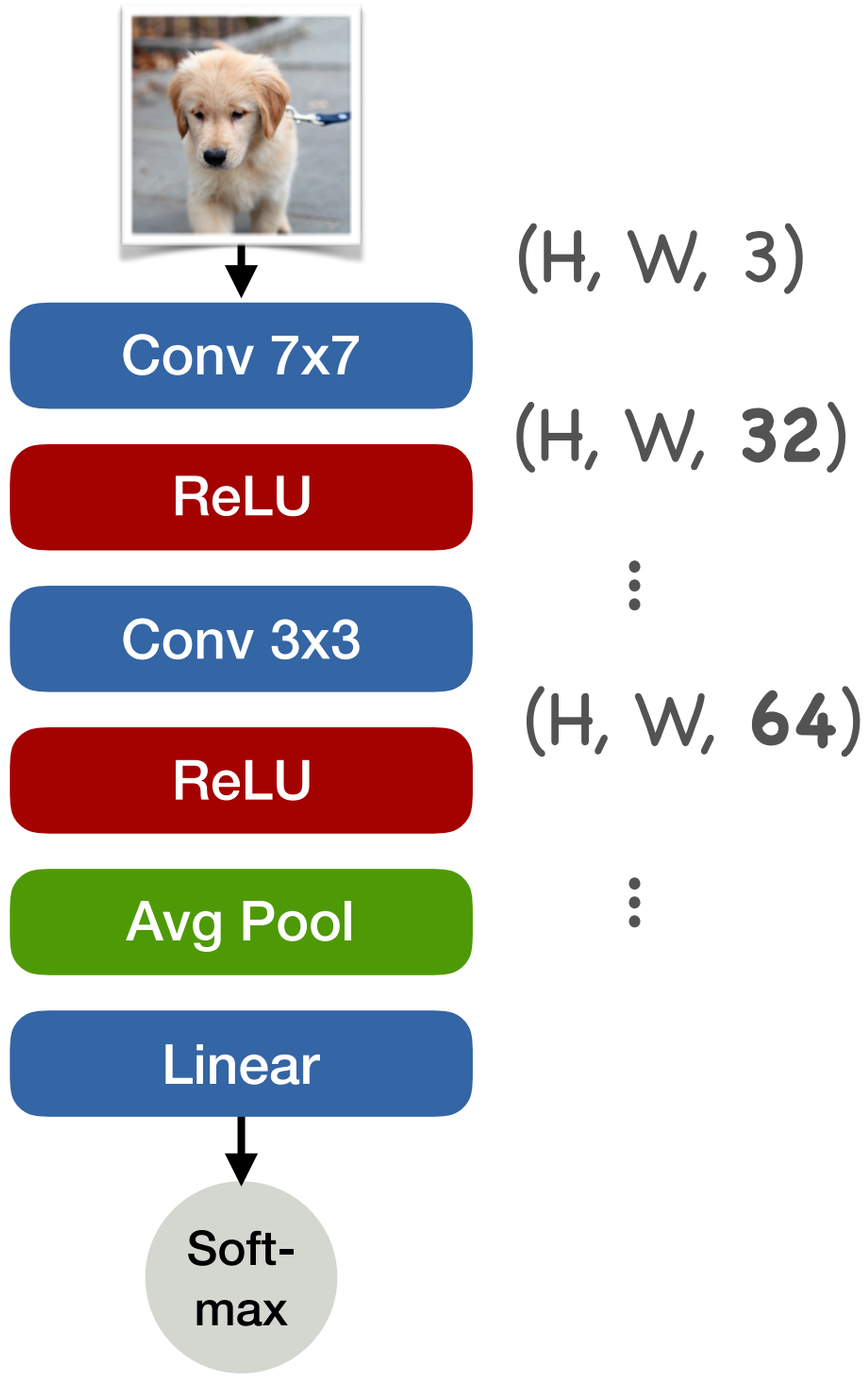
# Simpler models

- Traditional wisdom
  - Simpler model = less overfitting



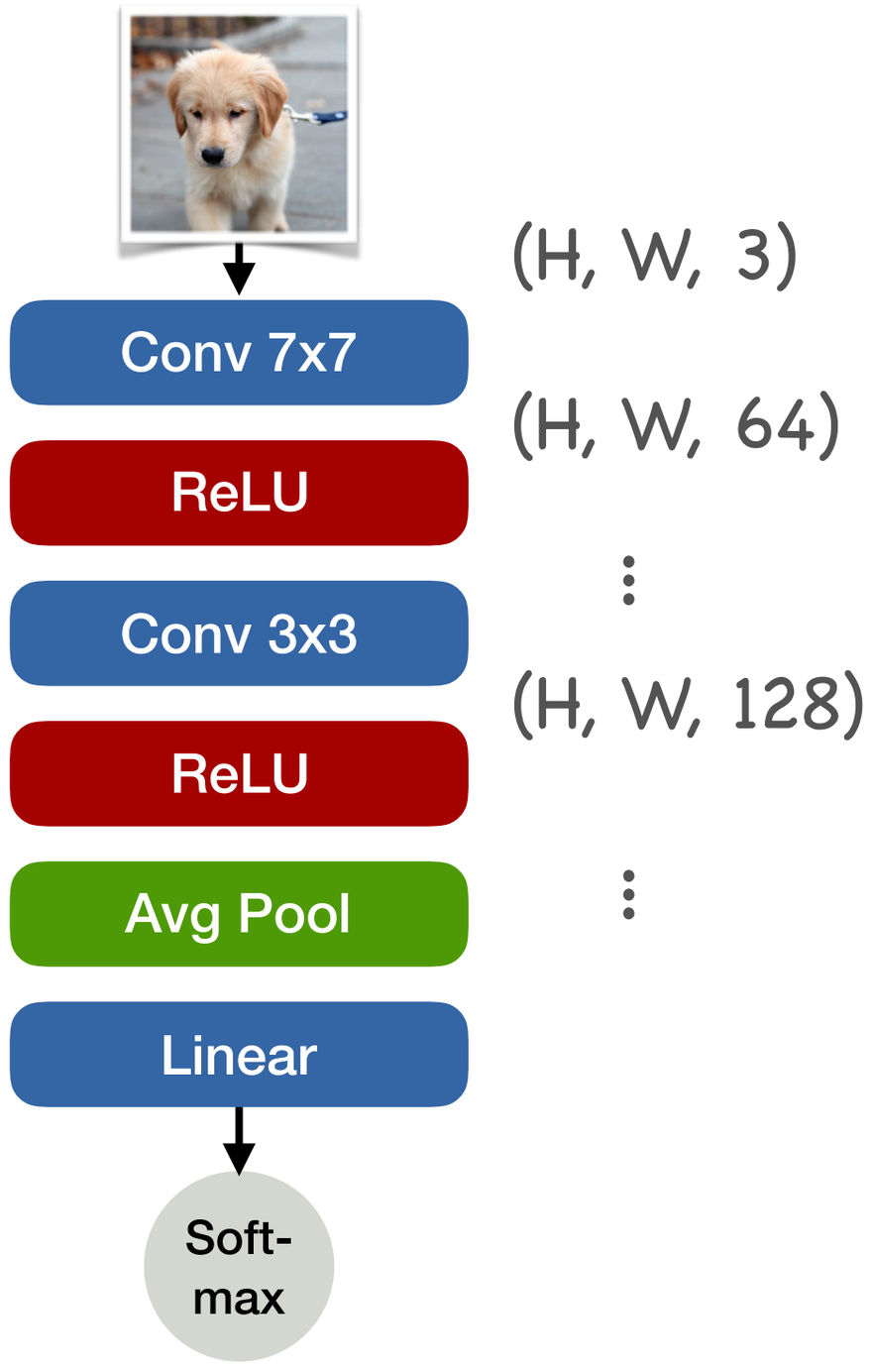
# Idea 1: Smaller model

- Overfits less
- Fits less
  - Worse generalization



# Idea 2: Big model with regularization

- Weight decay
  - Keep weights small (L2 norm)
- Works sometimes
  - Keep weight at same magnitude

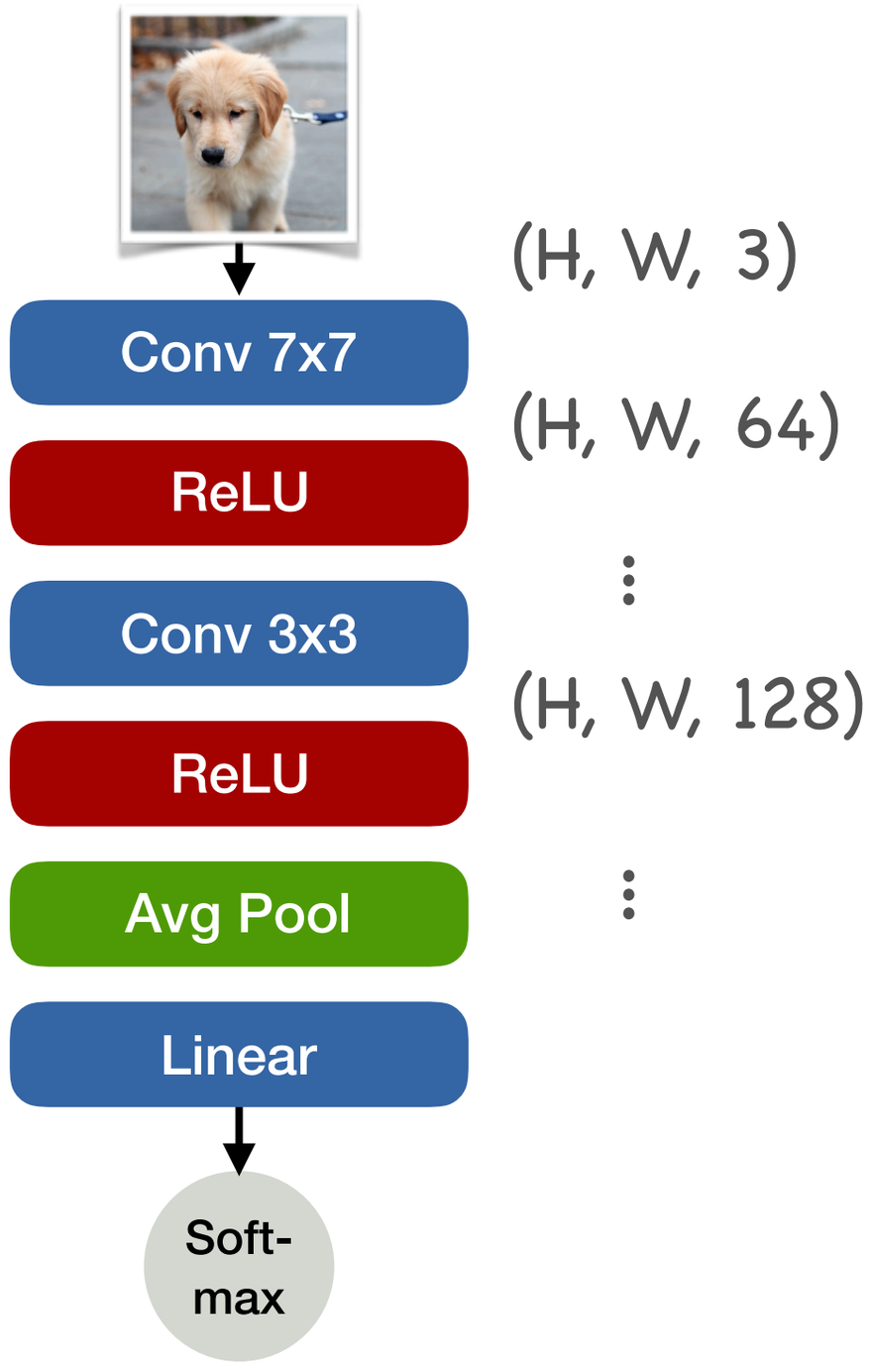


# How to use weight decay?

- Parameter in optimizer,
  e.g. torch.optim.SGD or
  torch.optim.Adam
  - weight_decay
  - Use 1e-4 as default

# Other reasons to use weight decay

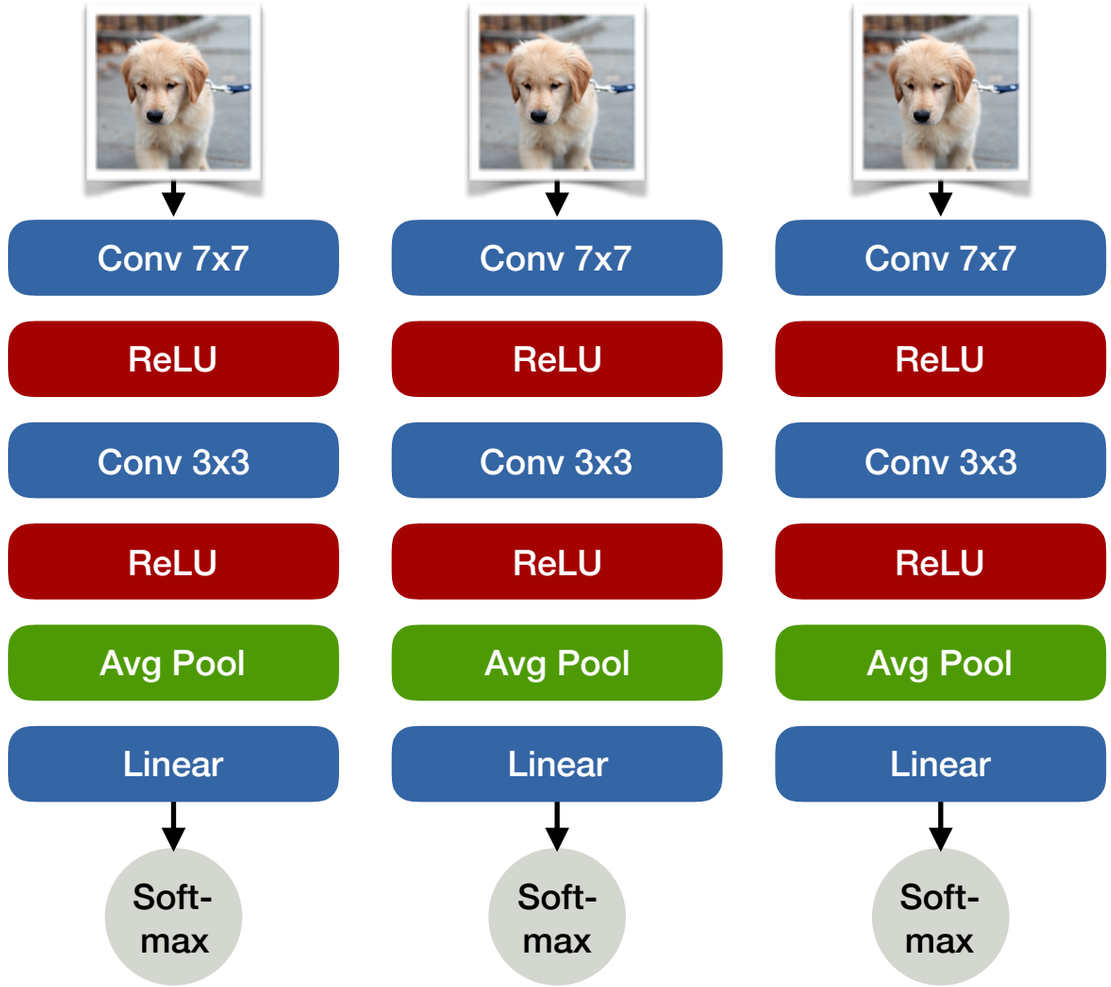
- Network weights cannot grow infinitely large
  - Helps handle exploding gradients



# Ensembles

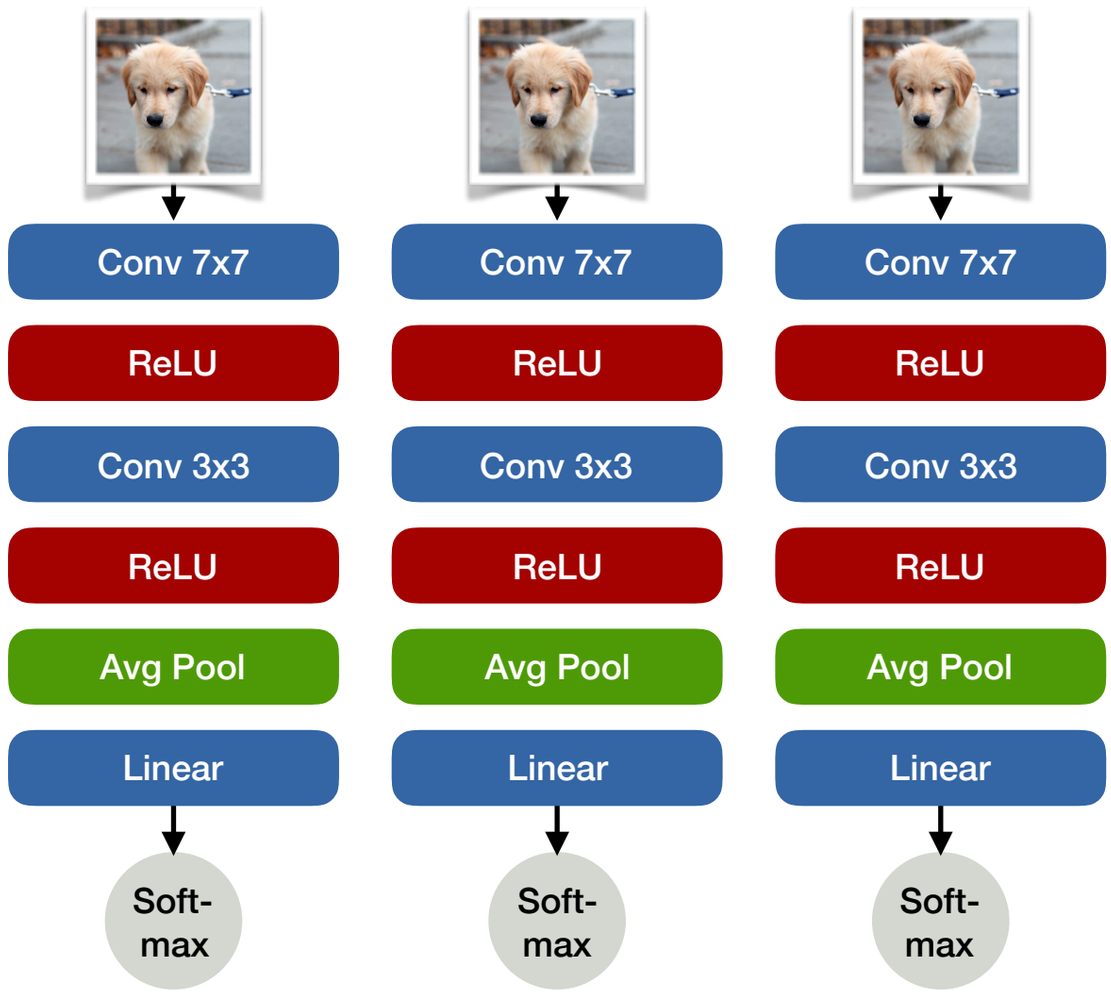
# Ensembles

- Train multiple models
  - Average predictions of multiple models



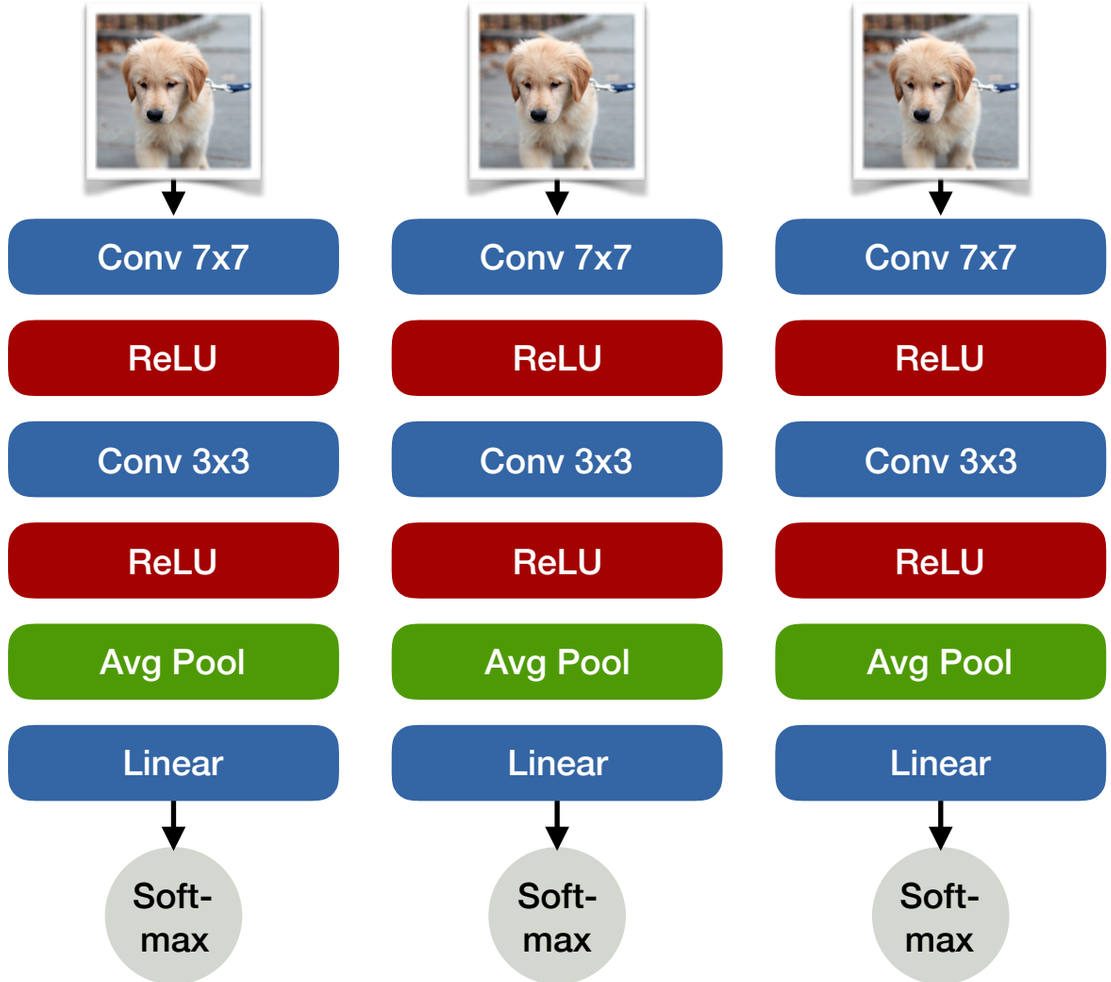
# Ensembles

- Pre-deep learning
  - Use different subsets of training data
- Deep learning
  - Use different random initializations / data augmentation
  - Different local minima



# Why do ensembles work?

- Fewer parameters / model
- Each model overfits in its own way
- Usually a 1-3% accuracy boost on most tasks
  - longer training



# Why do we average predictions?

- For a convex loss function
- loss of average prediction < average loss of individual models

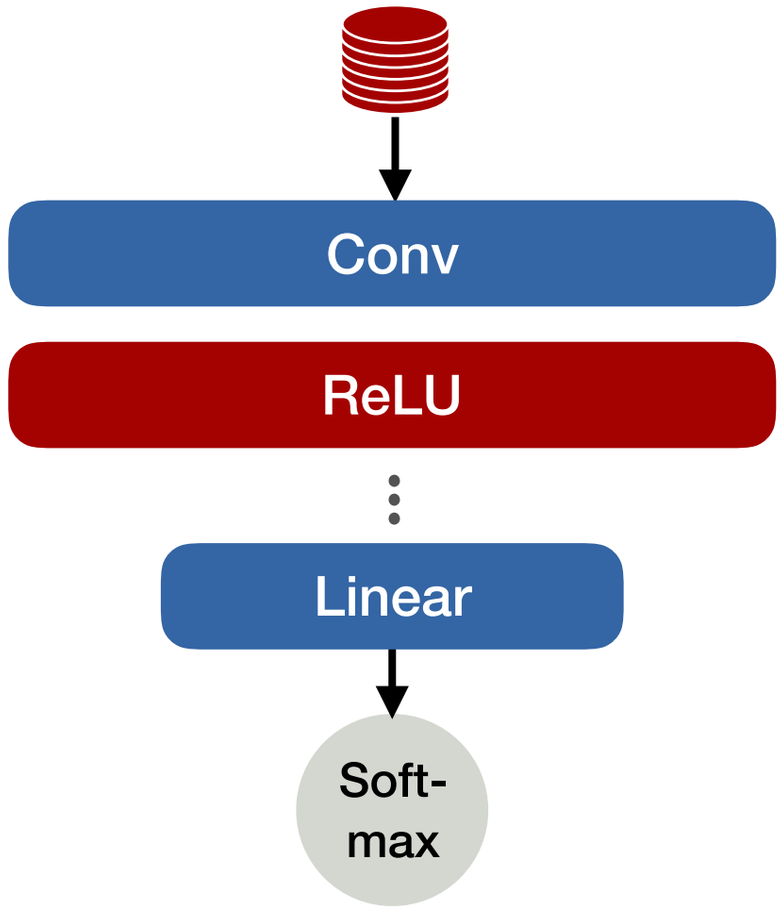
# When to use ensembles?

- If you have the compute power
- If you really need the last bit of accuracy
  - e.g. production, competitions

# Transfer learning

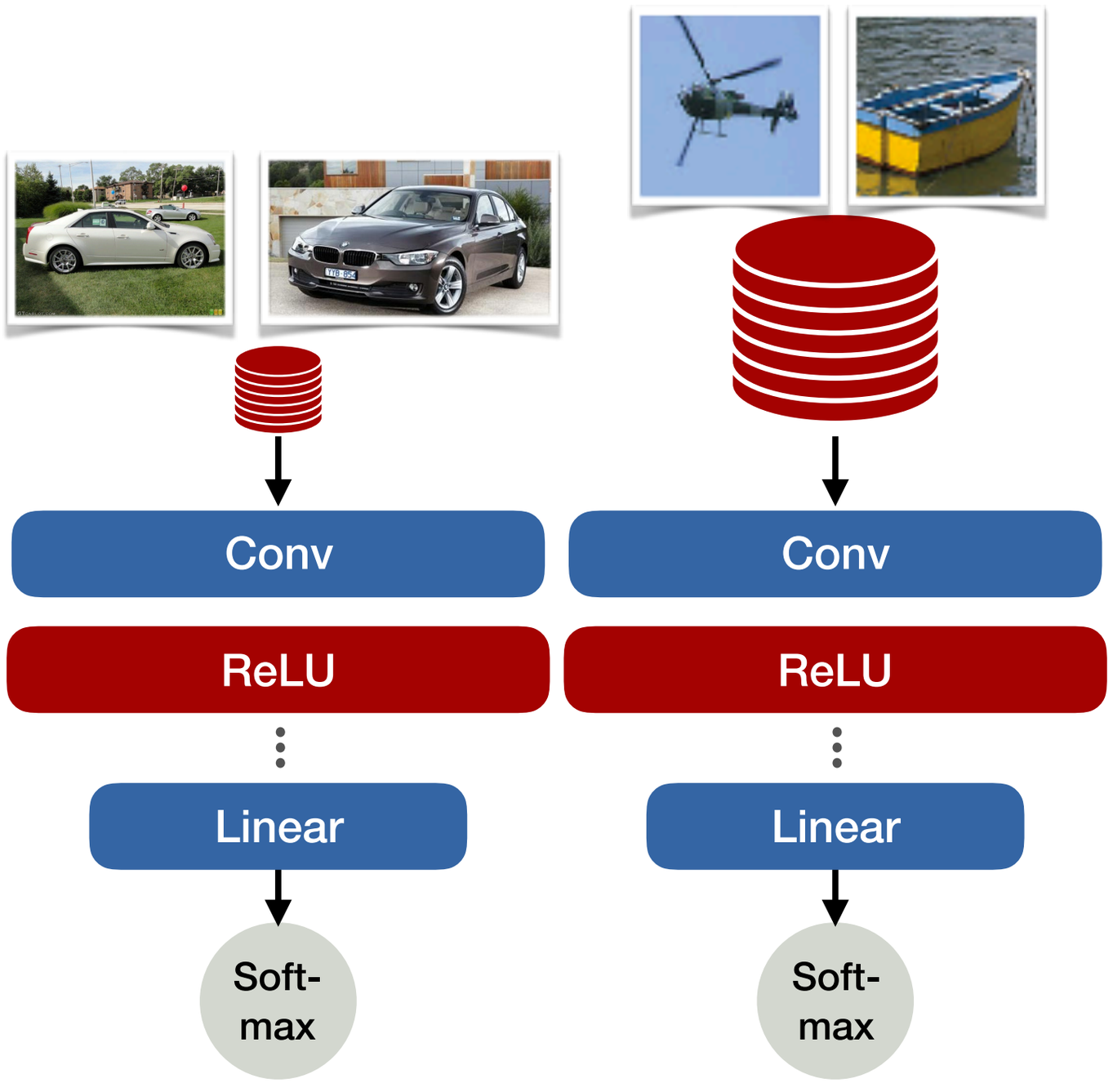
# Training on small datasets

- How to training
  - a large model
  - on a small dataset?



# Solution: Pre-training / fine-tuning

- Train model on large dataset (Pre-training)
  - on related task
- Copy model
- Continue training on small dataset (Fine-tuning)



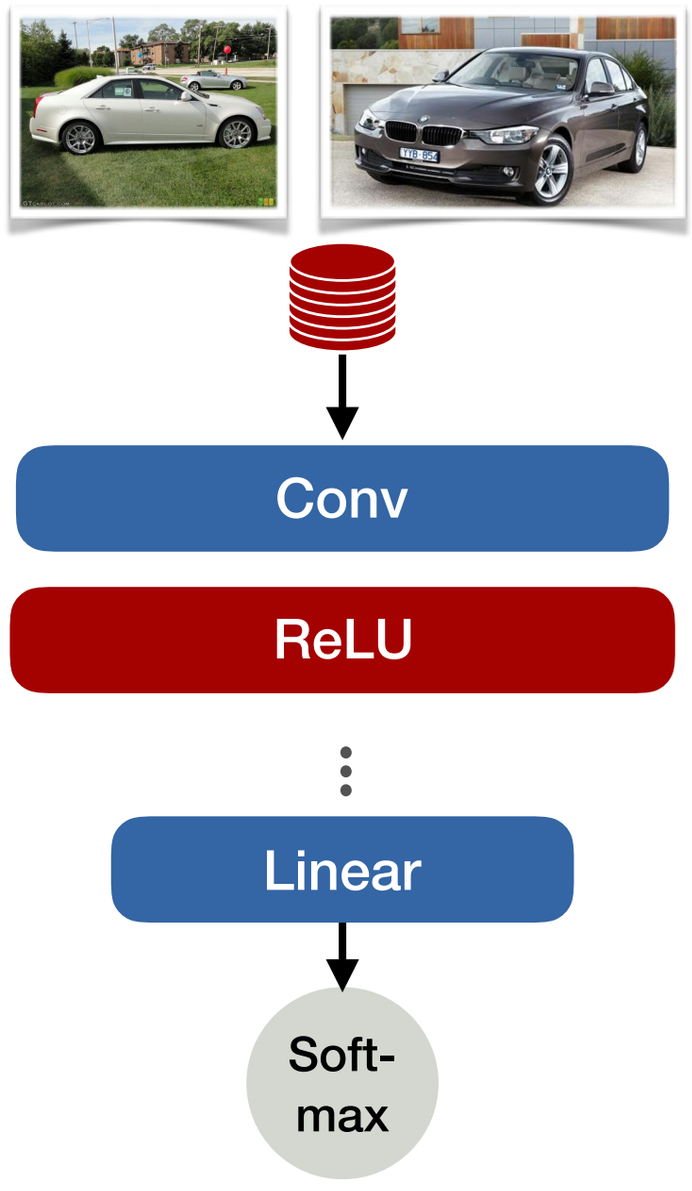
# Pre-training

- Computer vision
  - Supervised
    - e.g. ImageNet
- Natural Language Processing
  - Self-supervised
    - Unlabeled text


WIKIPEDIA
The Free Encyclopedia

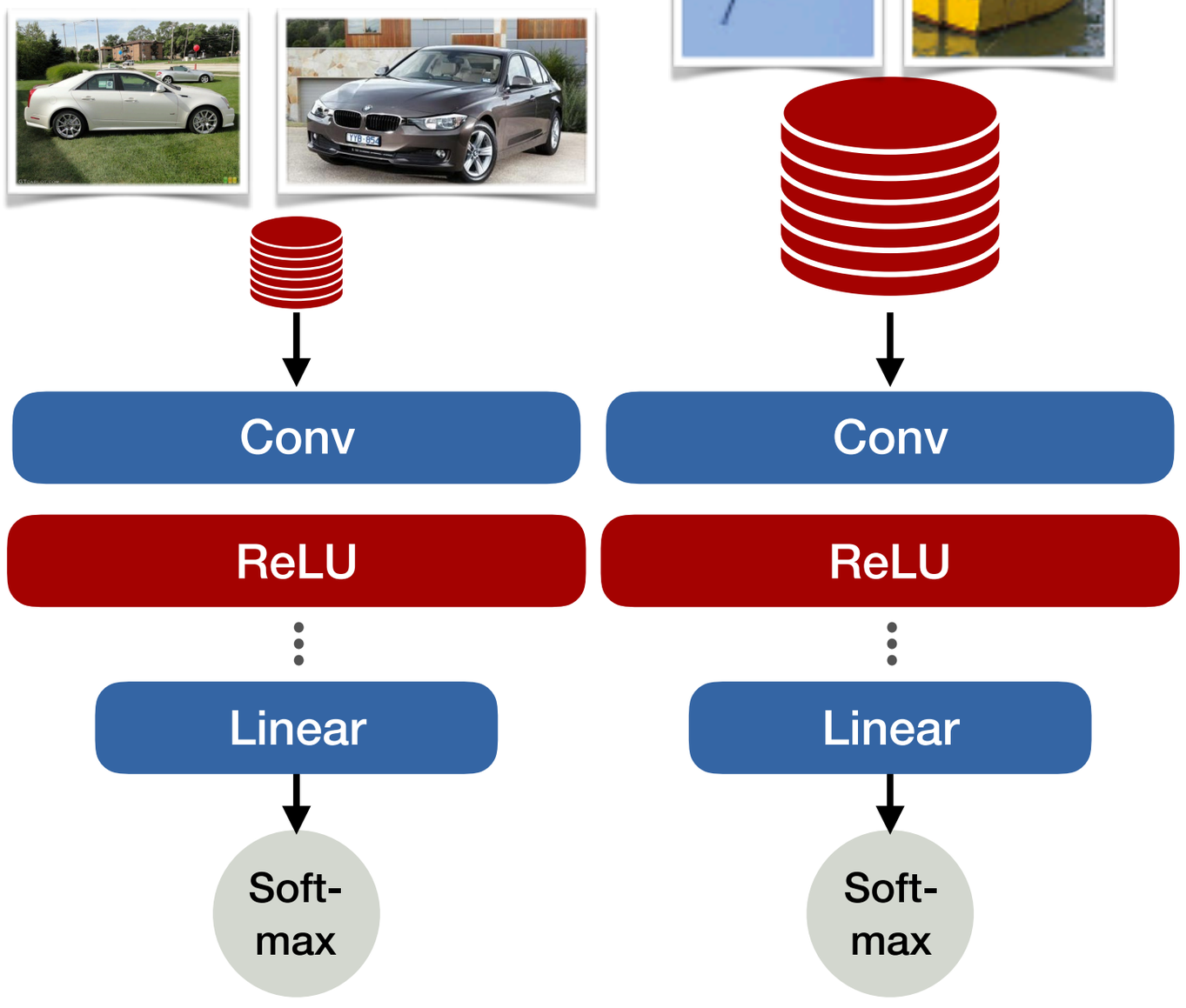
# Pre-training / fine-tuning in practice

- Download a pre-trained model
  - Model-zoo
  - Directly for author
- Run a few training iterations on small dataset



# Why does transfer learning work?

- Similar inputs
  - e.g. Images, Text, …
  - Transfer between tasks
- Good initialization
  - Learned weights are tuned well



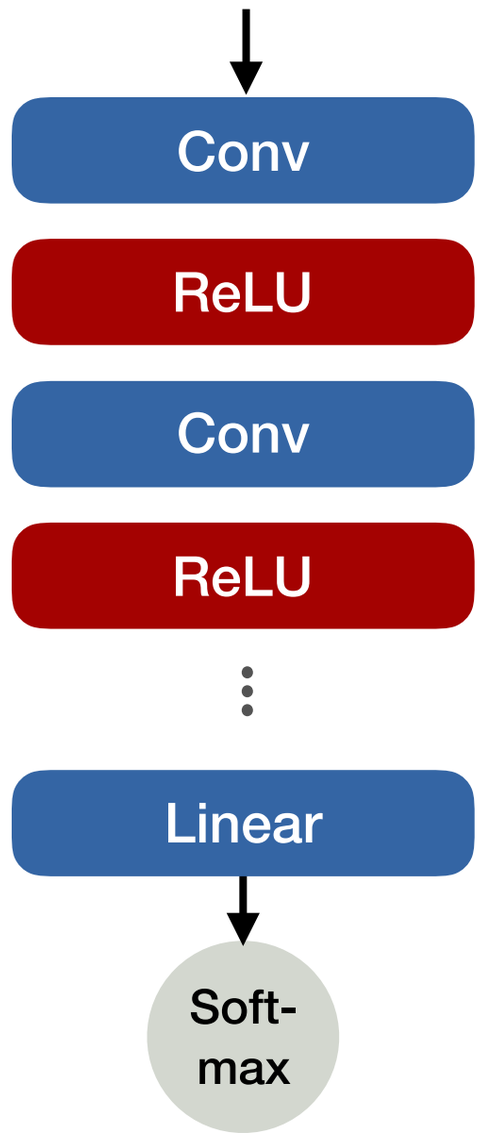
# When to use transfer learning?

- Whenever possible
- In early experiments
- Large pre-trained model exists

# Open Problem: Understanding generalization

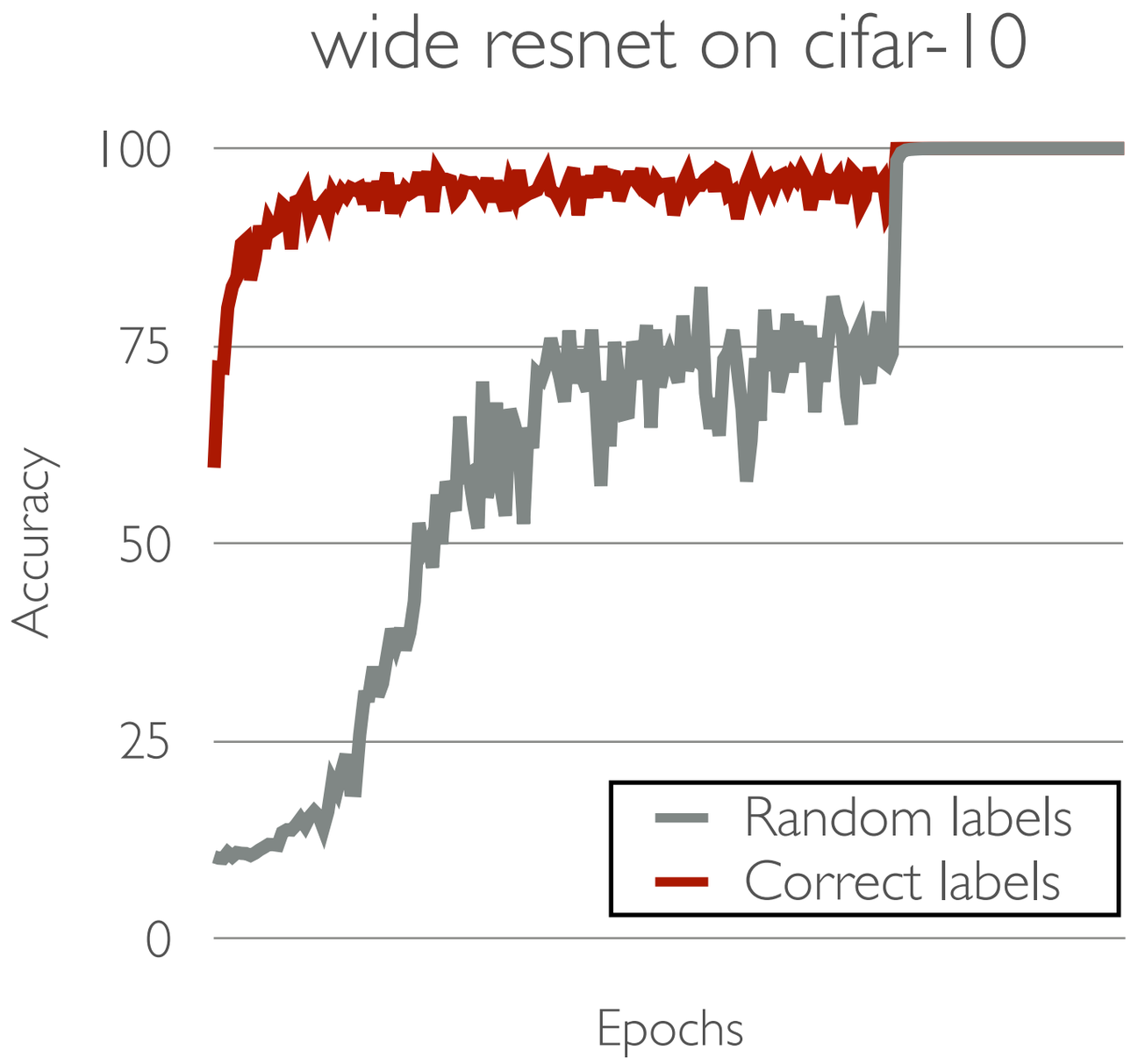
# Generalization in deep learning

- Standard wisdom
  - Bigger/wider models overfit more



# Deep networks are big enough to remember all training data

- Deep networks easily fit random labels
- Memorize all data
- Works even for random noise inputs




Understanding deep learning requires rethinking generalization, Zhang etal. 2017

# Why does SGD still work?

- SGD gradually minimizes objective

- Prefers solutions close to initialization

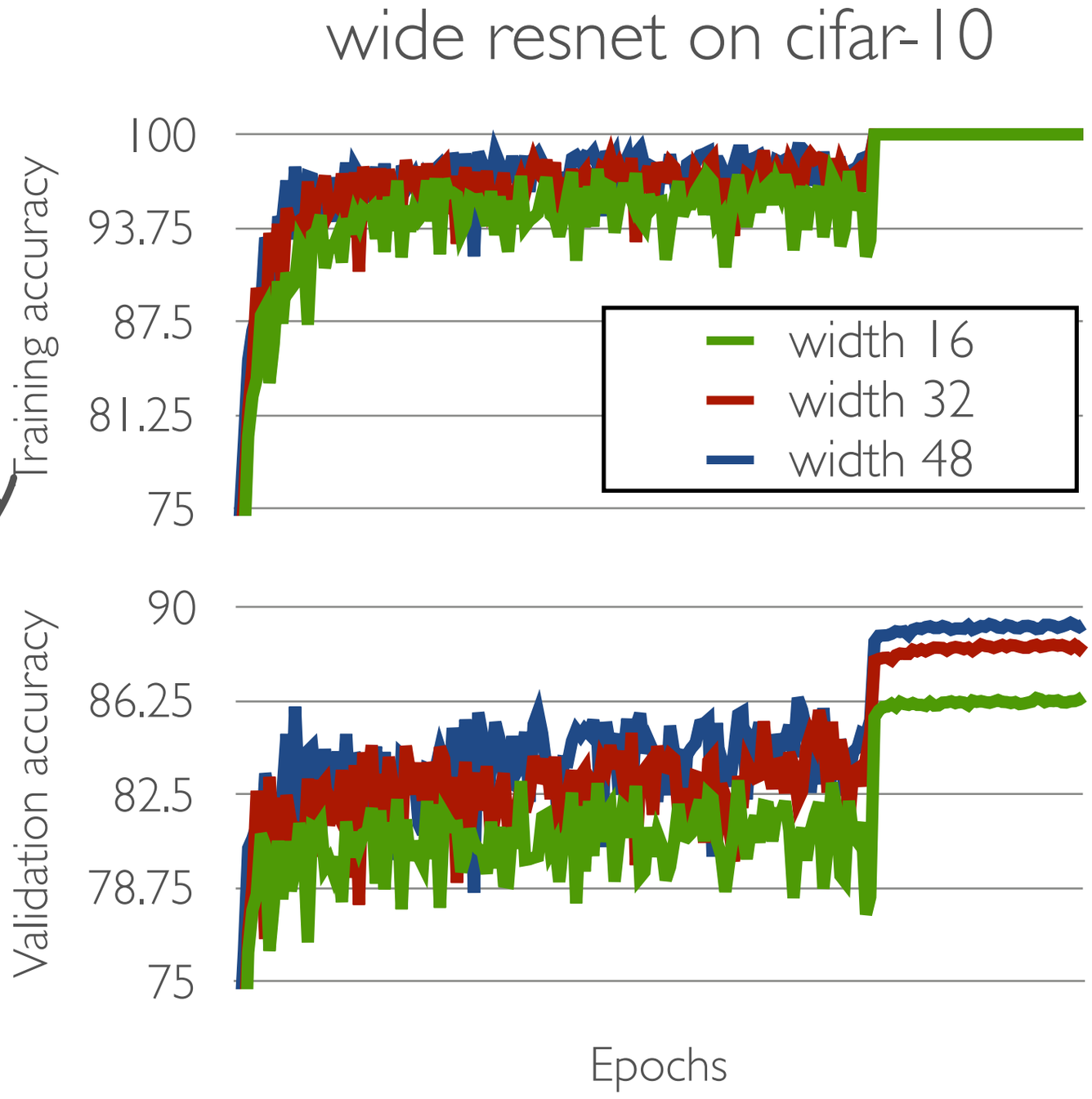
- Implicitly regularizes

- Random labels take SGD on a longer path

Exploring generalization in Deep Learning, Neyshabur etal. 2017

# Larger networks overfit less

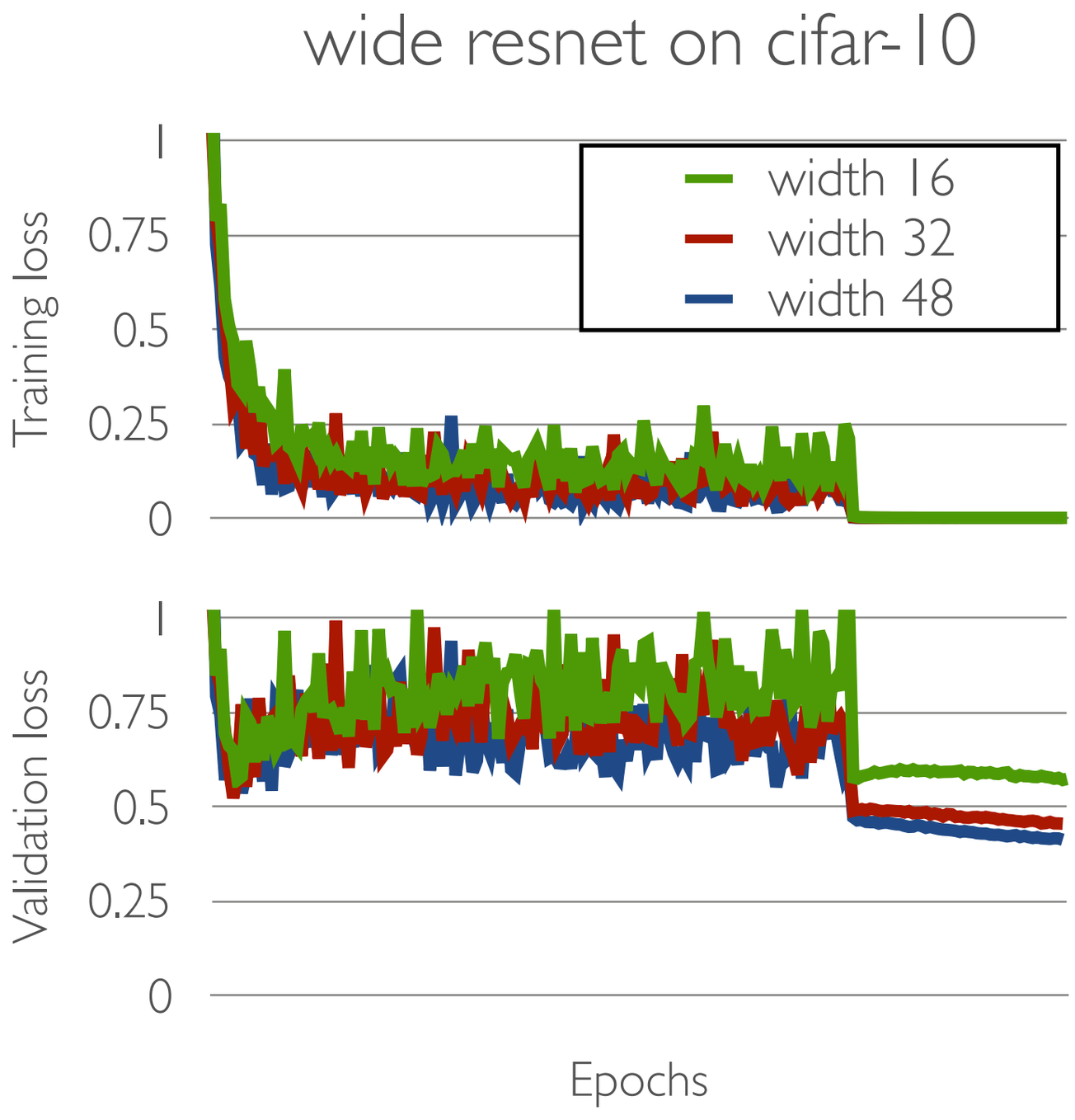
- Without data augmentation
- 100% training accuracy
- Larger models generalize better
- Hence overfit less




Understanding deep learning requires rethinking generalization, Zhang etal. 2017

# Larger networks overfit less

- All models overfit massively on loss (log likelihood)




On Calibration of Modern Neural Networks, Guo etal. 2017

# Larger networks overfit less

- Do we need a new learning theory?
- Do we need new intuitions?

# In summary

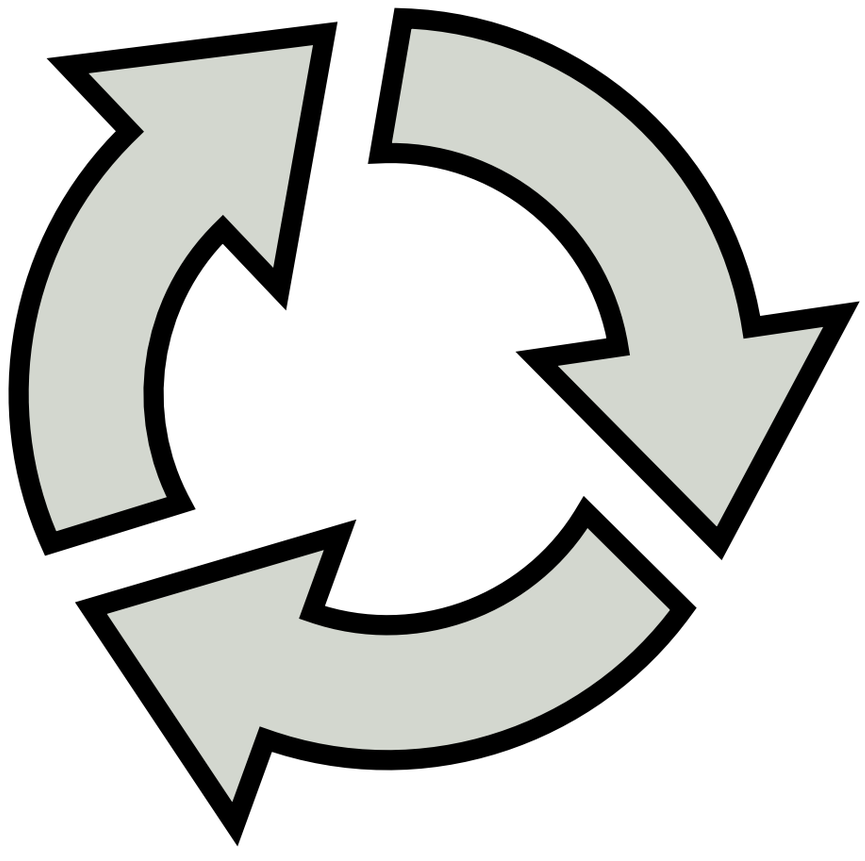
- Models can overfit, but do not with SGD and data augmentation
    - Implicit regularization
        - How to do make it explicit?
        - Overfitting is dependent on learning algorithms (e.g. Adam overfits more)
- How can we measure overfitting?

# Summary, a practical guide to deep network optimization

# Graduate student descent

semi-automated

Look at your data / model output



manual

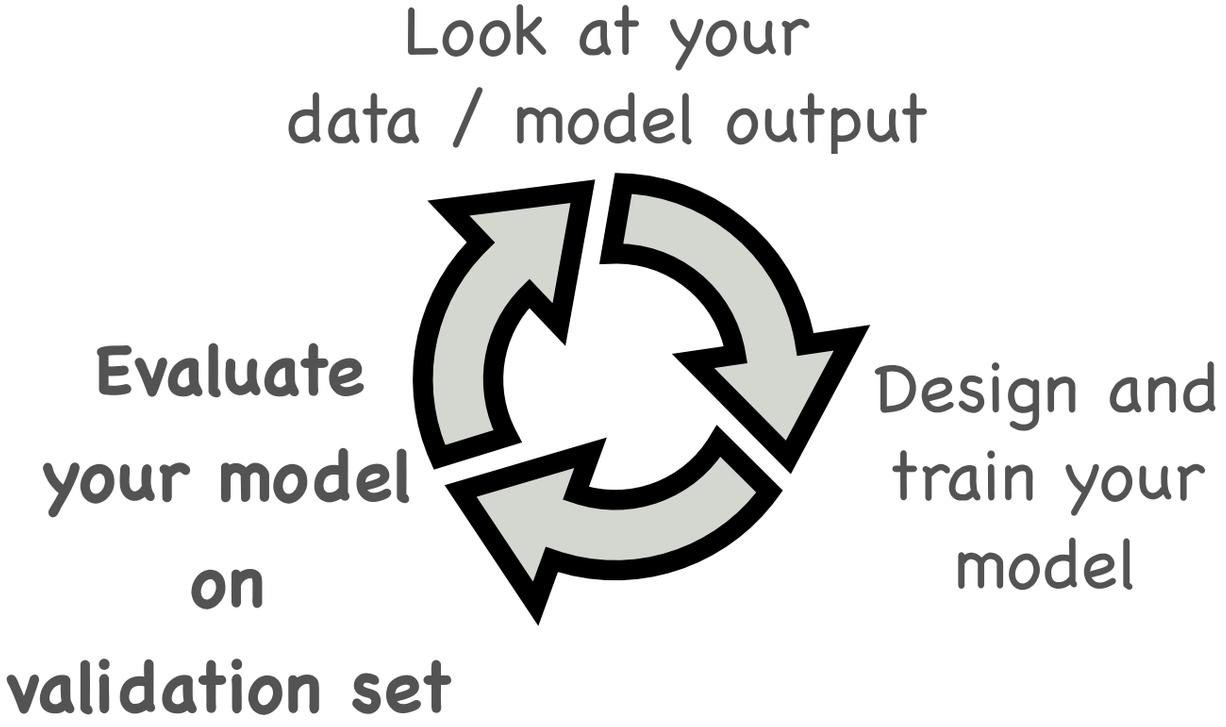
Design and train your model

Evaluate your model on validation set

automated

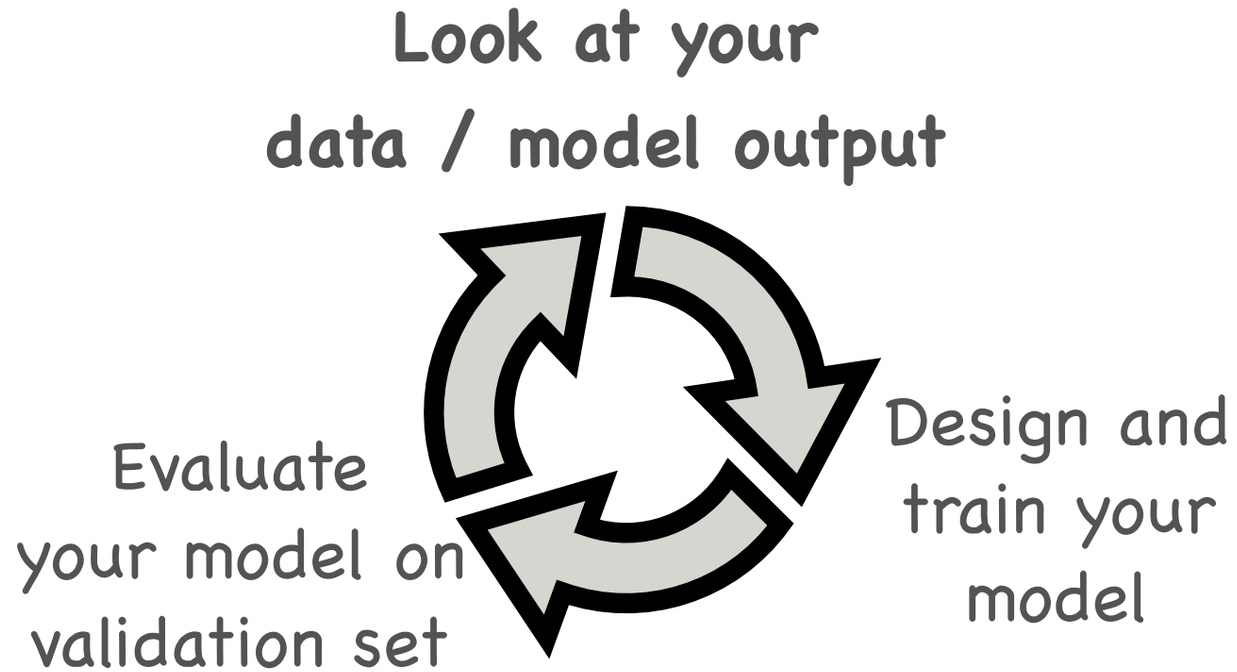
# Evaluation on validation set

- Run during training
- Every epoch or n iterations
- Log in TensorBoard



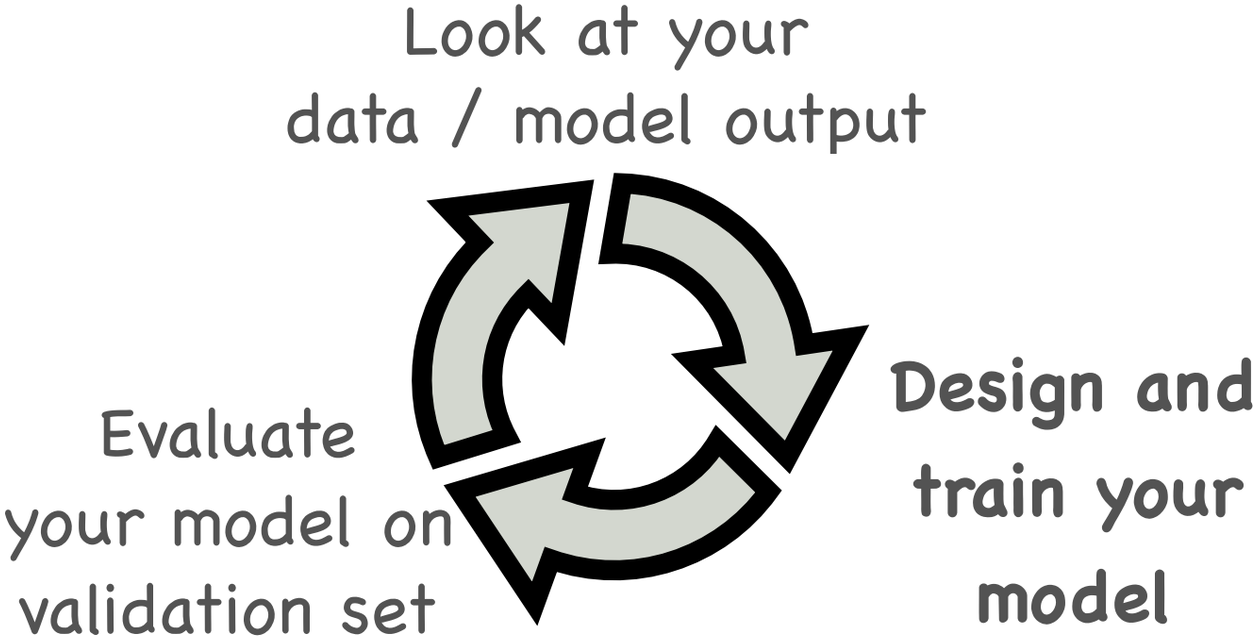
# Look at your data / model output

- Run during training
  - Every epoch or n iterations
- Log in TensorBoard
- Select same training and validation images



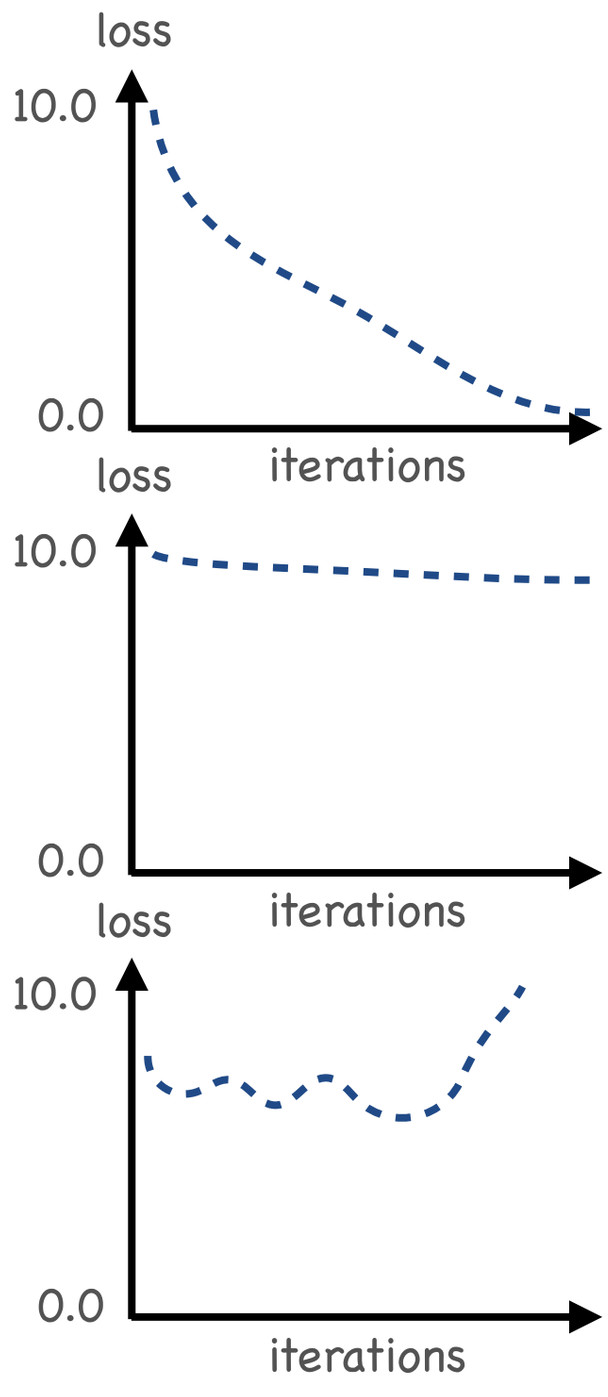
# Design and train your model

- Mostly manual work



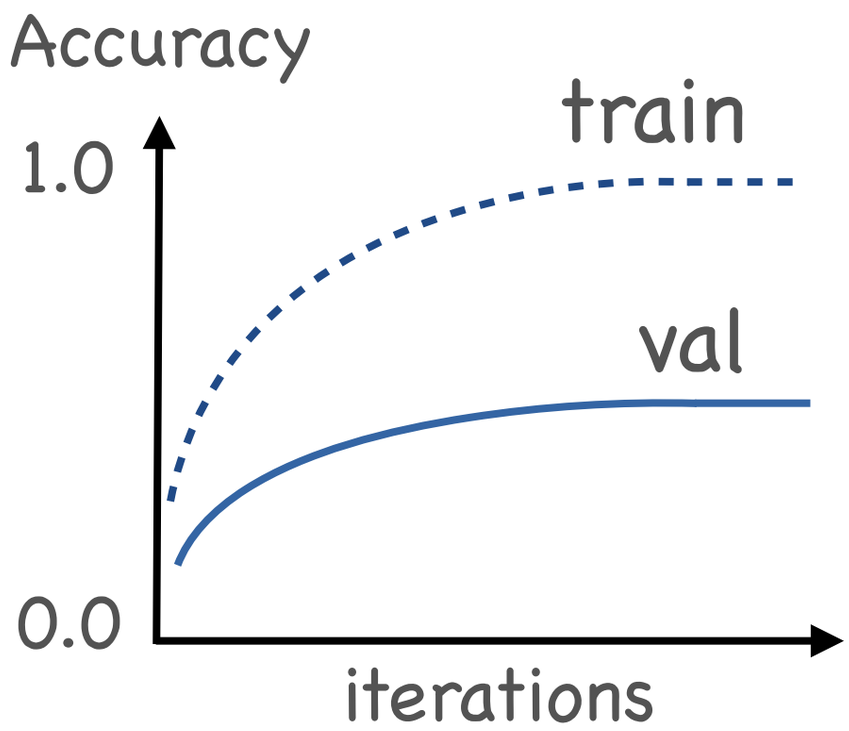
# Design and train your model

- Network does not train
  - Vanishing or exploding gradients?
    - Fix initialization and learning rate
  - Slow training
    - Add normalization
    - Residual connections
- Iterate until model trains



# Design and train your model

- Network overfits to training data
  - Add data augmentation
  - Early stopping
  - Try a pre-trained network
  - Collect more data
- Iterate until model generalizes well



# Design and train your model

- Network fits training and validation data well
- Stop graduate student descent
- Take a break
- Evaluate on test set