

# Image recognition

## deep learning 2

Raoul Grouls, 25 November 2025

# Recap

$$X = \{\vec{x}_1, \dots, \vec{x}_n \mid \vec{x} \in \mathbb{R}^d\}$$

Data

$$y = \{y_1, \dots, y_n \mid y \in \{0,1\}\}$$

Trainable Weights  $W, b$

$$\text{(Non)linearity} \quad f(X) = WX + b \quad \sigma(X) = \max(0, X)$$

Predict

$$\hat{y} = f_n \circ \sigma \circ f_{n-1} \circ \dots \circ \sigma \circ f_1$$

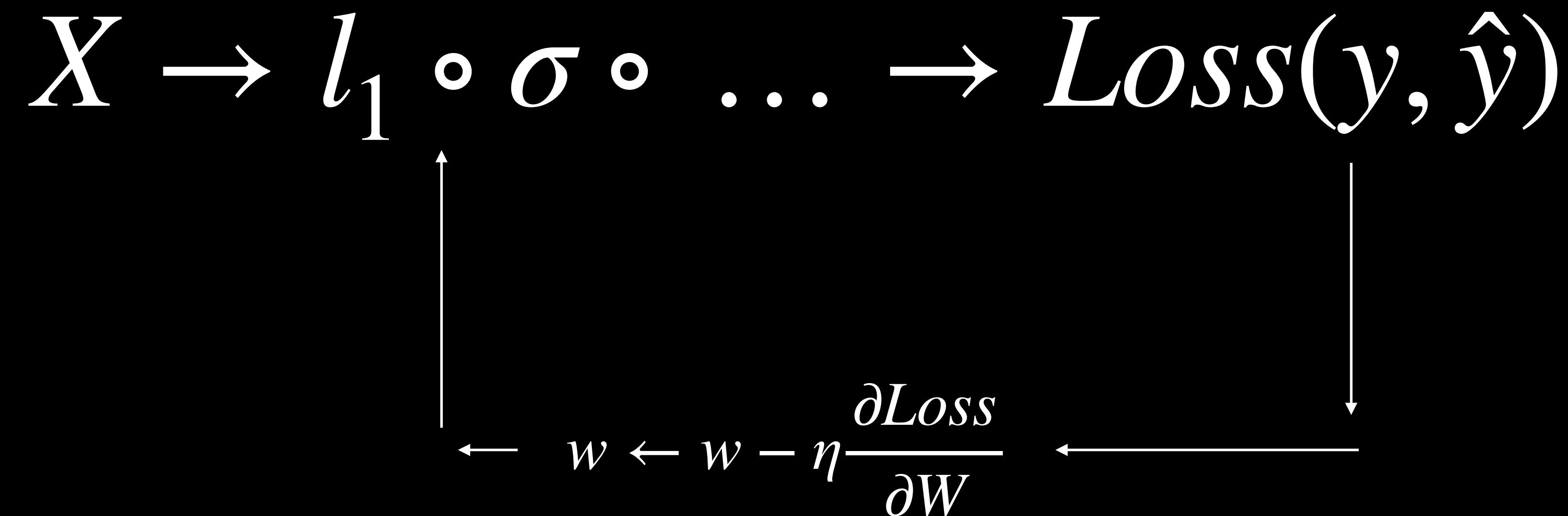
Loss

$$Loss(y, \hat{y})$$

Optimize

$$w \leftarrow w - \eta \frac{\partial Loss}{\partial W}$$

# Training

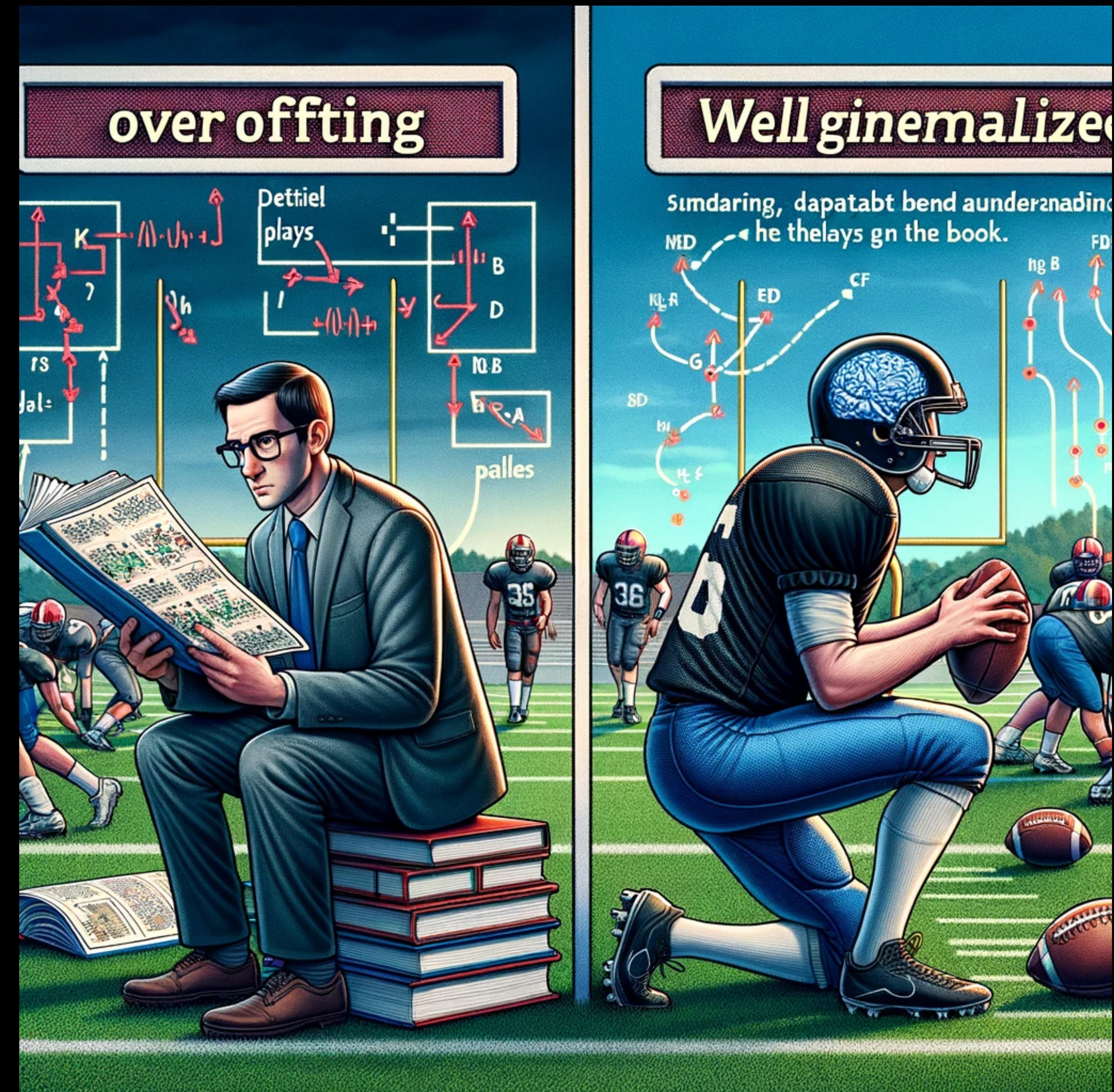


# Training & regularization

# Overfitting

A model that corresponds too closely to a particular set of data, therefore failing to fit to unseen data.

- the error / performance metrics for the training data are very good, but bad on unseen data.
- Overfit models tend to have higher complexity (number of learnable parameters) than is warranted by the data complexity
- Small changes in the data can lead to large changes in the model.

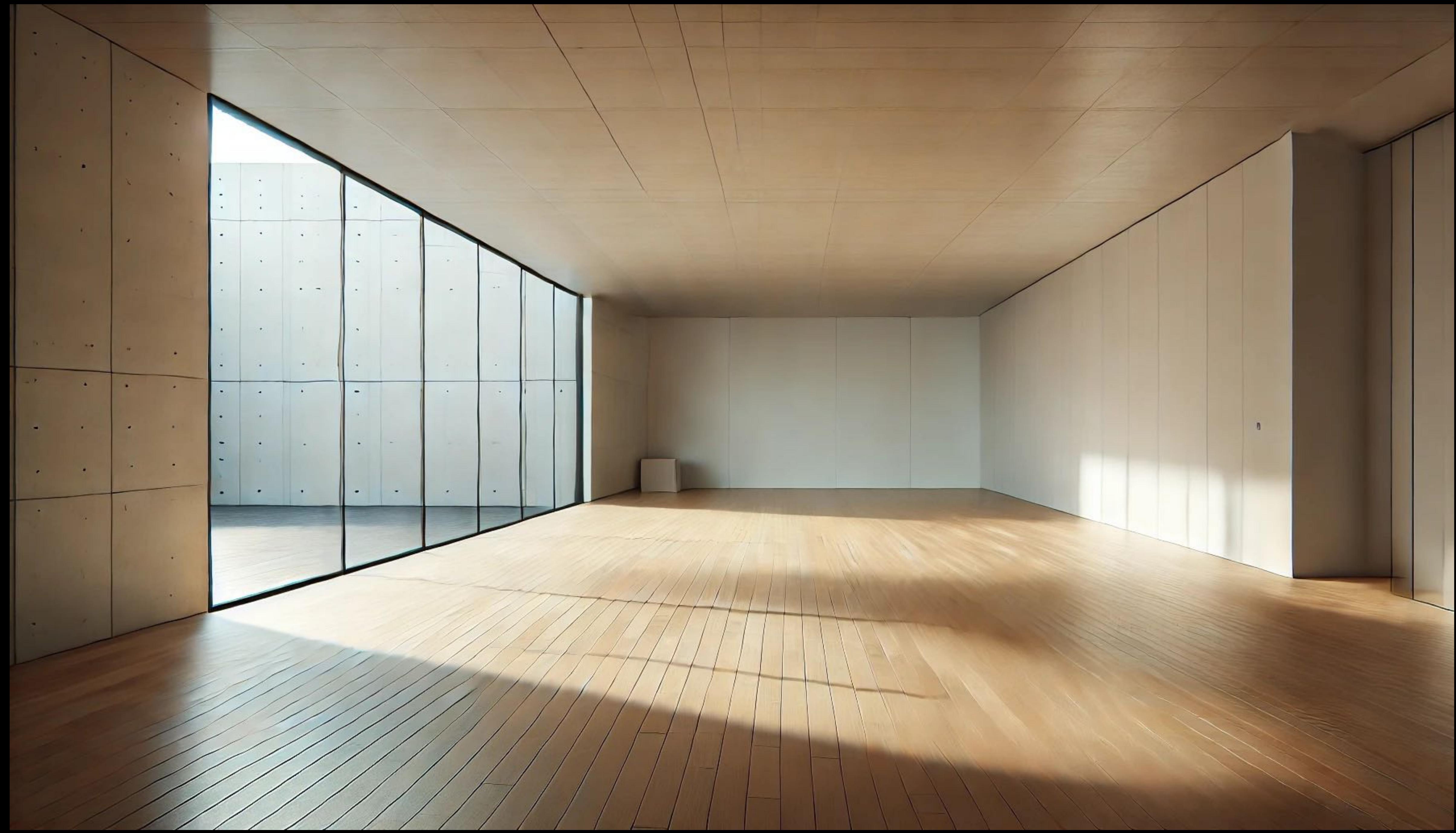


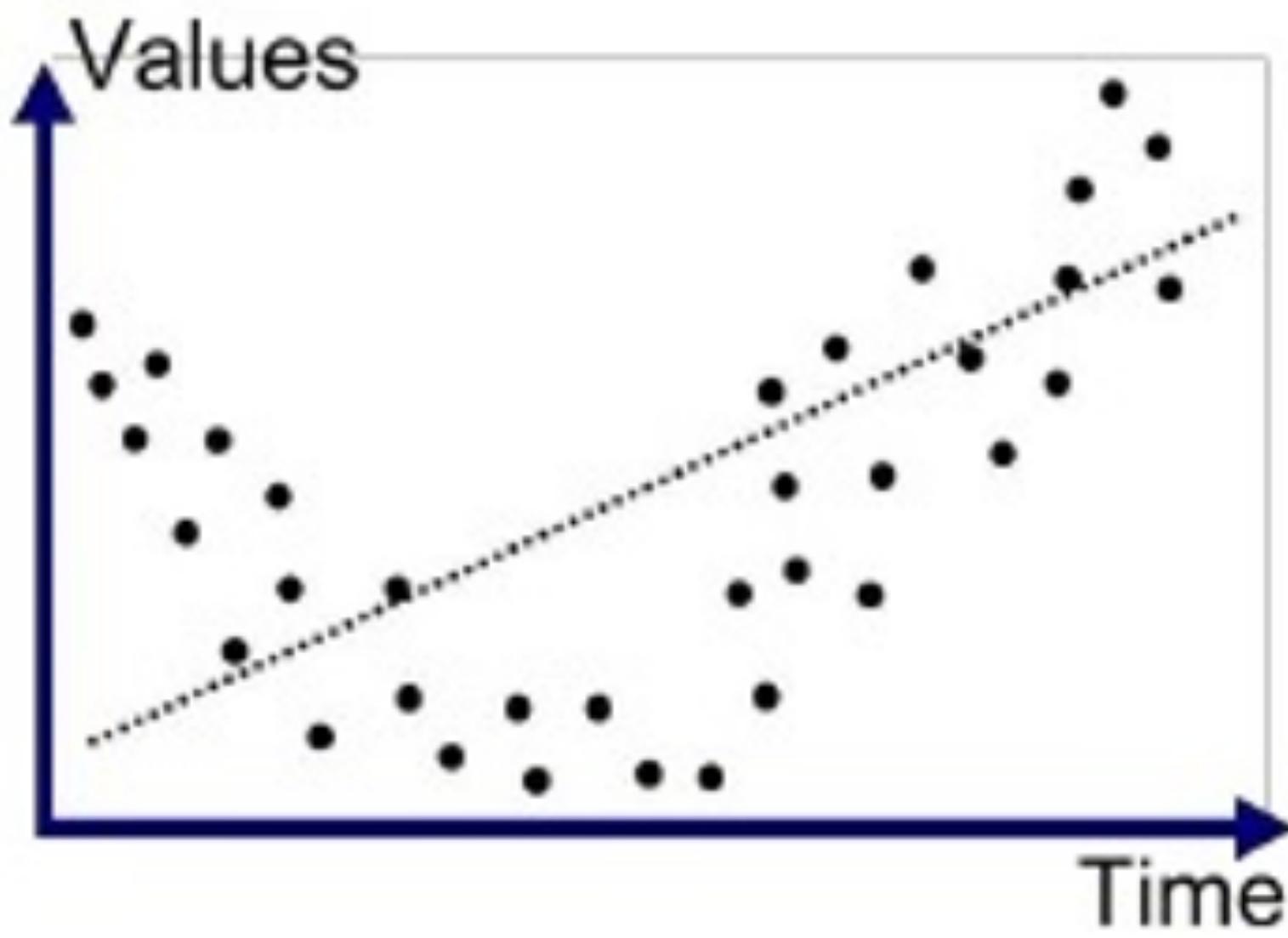


# Underfitting

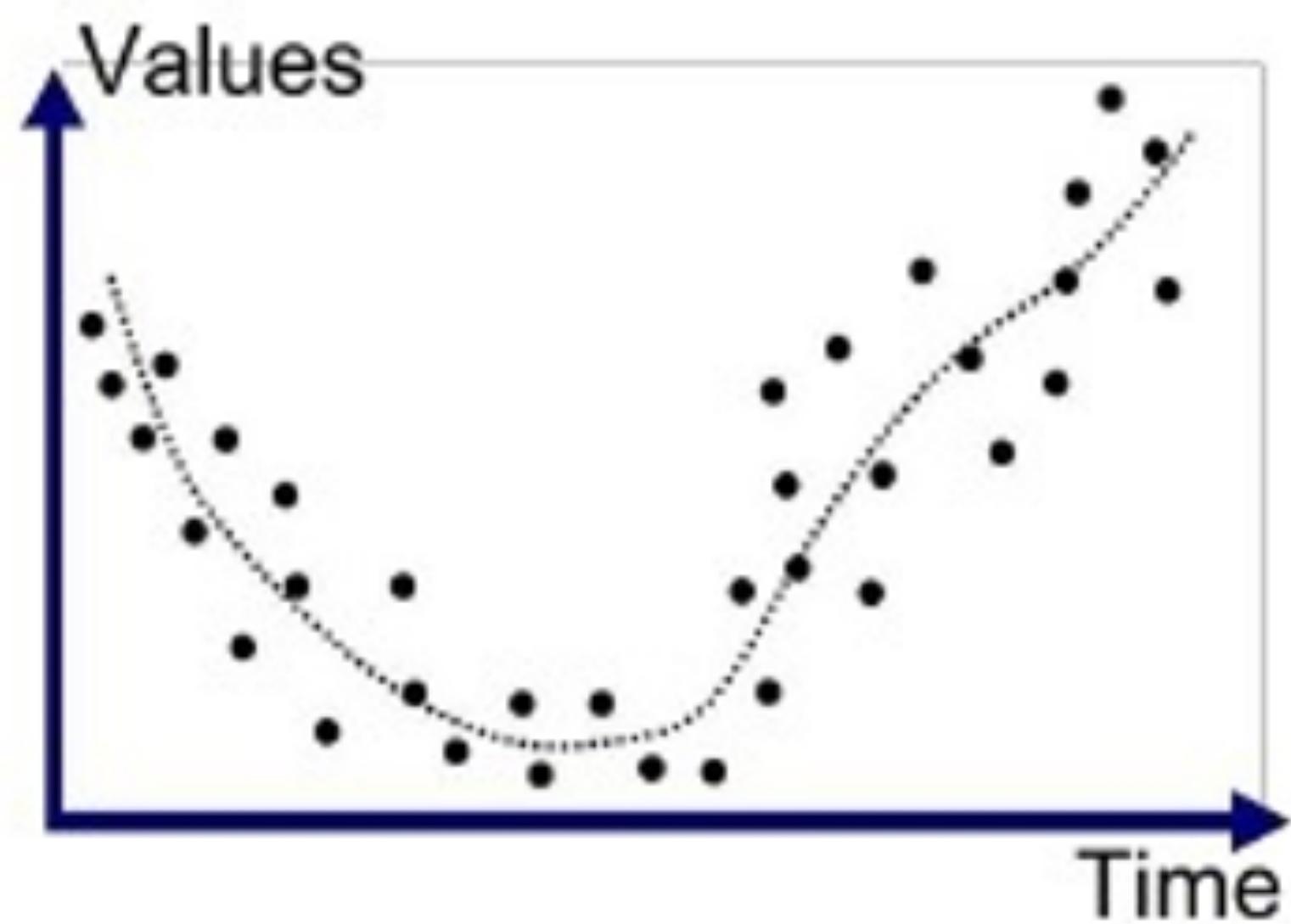
A model that is **too simplistic** to capture the underlying patterns in data, therefore failing to perform well on both training and unseen data.

- The error/performance metrics are poor on both training and test data.
- Underfit models tend to have **lower complexity** (fewer learnable parameters) than is warranted by the data complexity.
- Small changes in the data typically lead to **minimal changes** in the model predictions.

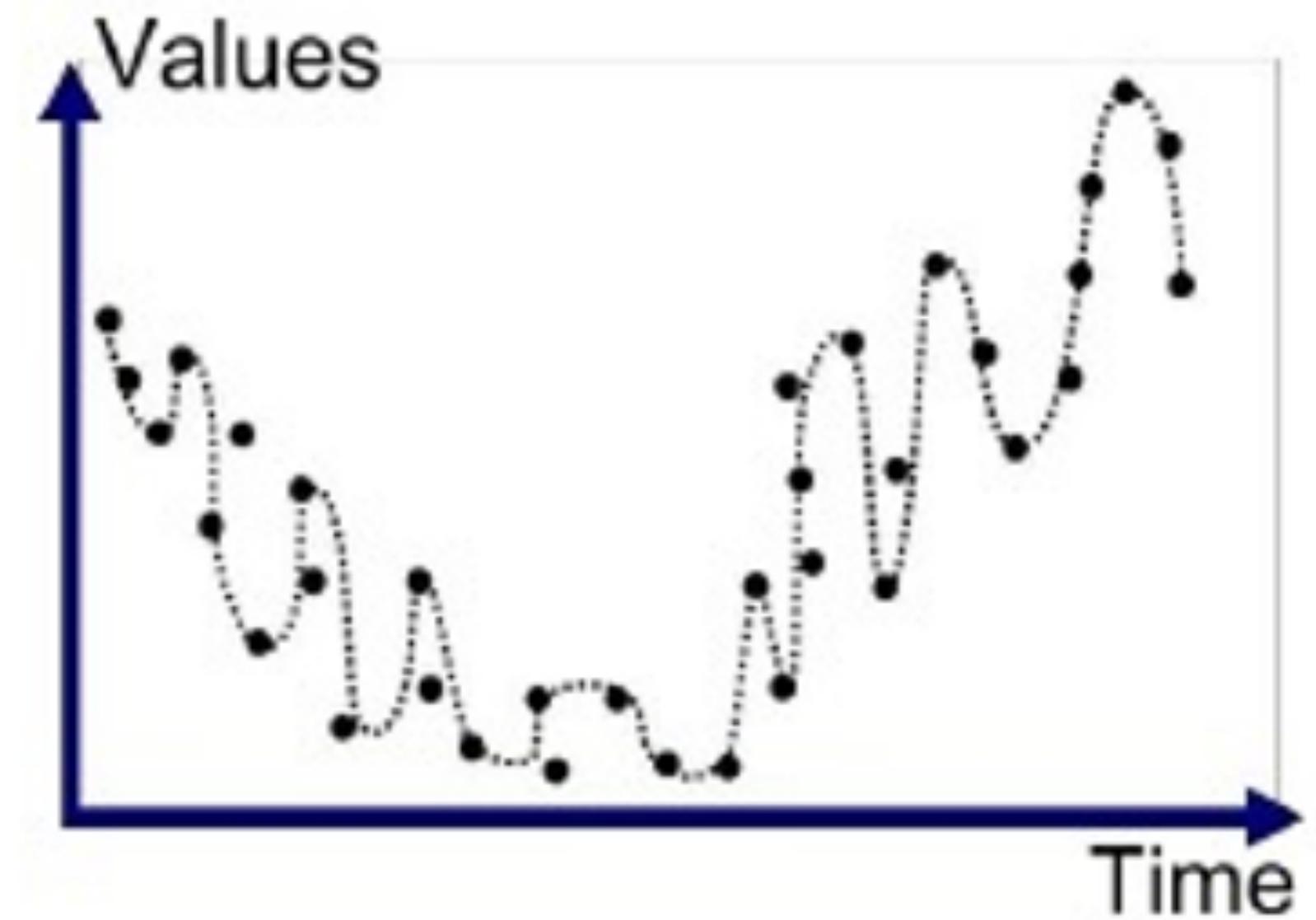




Underfitted



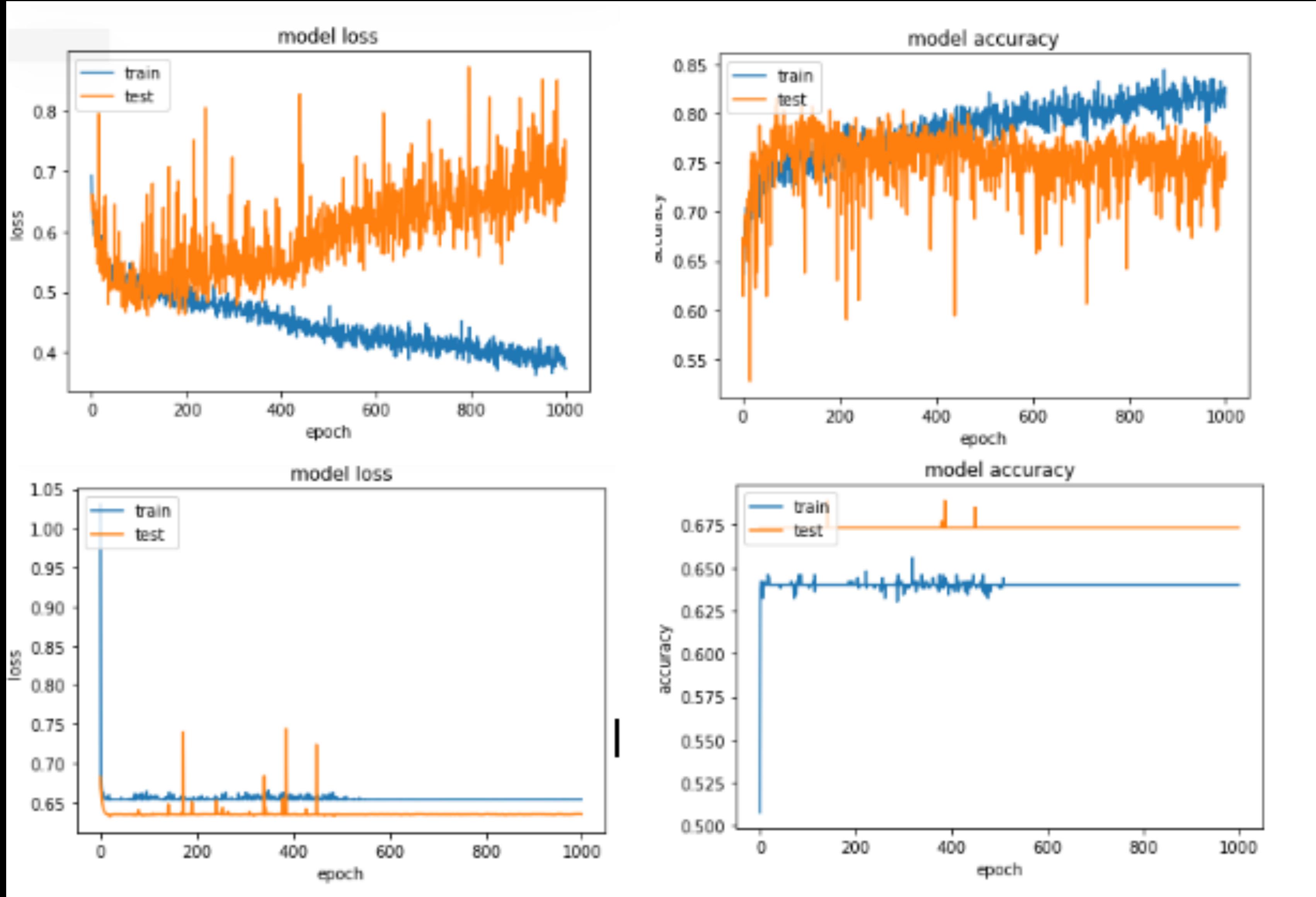
Good Fit/R robust



Overfitted



# Spotting overfitting & underfitting



# Regularization

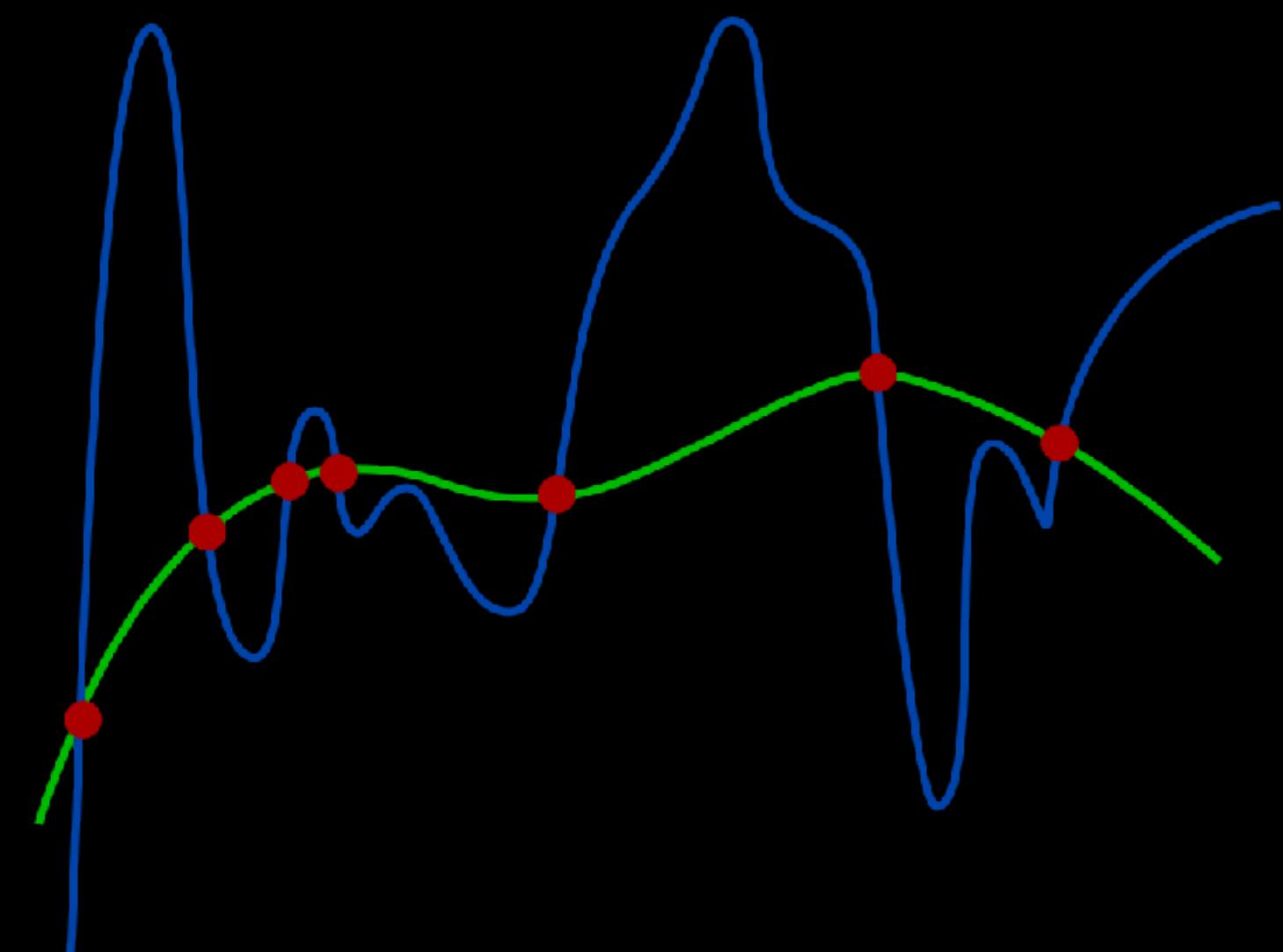
A process used to enforce simpler models to prevent overfitting, even though the model has the capacity to be overly complex.

## Explicit Regularization:

- Involves directly adding a term to the optimization problem (a main idea in SVM)
- These terms can include priors, penalties, or constraints.
- Imposes a cost on the optimization function.
- Eg in PyTorch, this happens when you set `weight_decay > 0` in the optimizer.

$$J(y, \hat{y} | W) = Loss(y, \hat{y}) + \eta \cdot Penalty(W)$$

```
class TrainerSettings(FormattedBase):  
    epochs: int  
    metrics: List[Callable]  
    logdir: Path  
    train_steps: int  
    valid_steps: int  
    reporttypes: List[ReportTypes]  
    optimizer_kwargs: Dict[str, Any] = {"lr": 1e-3, "weight_decay": 1e-5}  
    scheduler_kwargs: Optional[Dict[str, Any]] = {"factor": 0.1, "patience": 10}  
    earlystop_kwargs: Optional[Dict[str, Any]] = {  
        "save": False,  
        "verbose": True,  
        "patience": 10,  
    }
```



# Regularization

## Implicit Regularization:

- All methods that are not directly added to the optimization problem.
- Examples include early stopping, introducing noise (eg with random batches, dropout), and discarding outliers.
- Ubiquitous in machine learning, including *batched stochastic gradient descent* and *ensemble methods* like random forests and gradient boosted trees.

```
class EarlyStopping:  
    def __init__(  
        self,  
        log_dir: Path,  
        patience: int = 7,  
        verbose: bool = False,  
        delta: float = 0.0,  
        save: bool = False,  
    ) -> None:  
        """  
  
        Args:  
            log_dir (Path): location to save checkpoint to.  
            patience (int): How long to wait after last time validation loss improved.  
            | | | | | Default: 7  
            verbose (bool): If True, prints a message for each validation loss improvement. Default: False  
            delta (float): Minimum change in the monitored quantity to qualify as an improvement. Default: 0.0  
        """
```

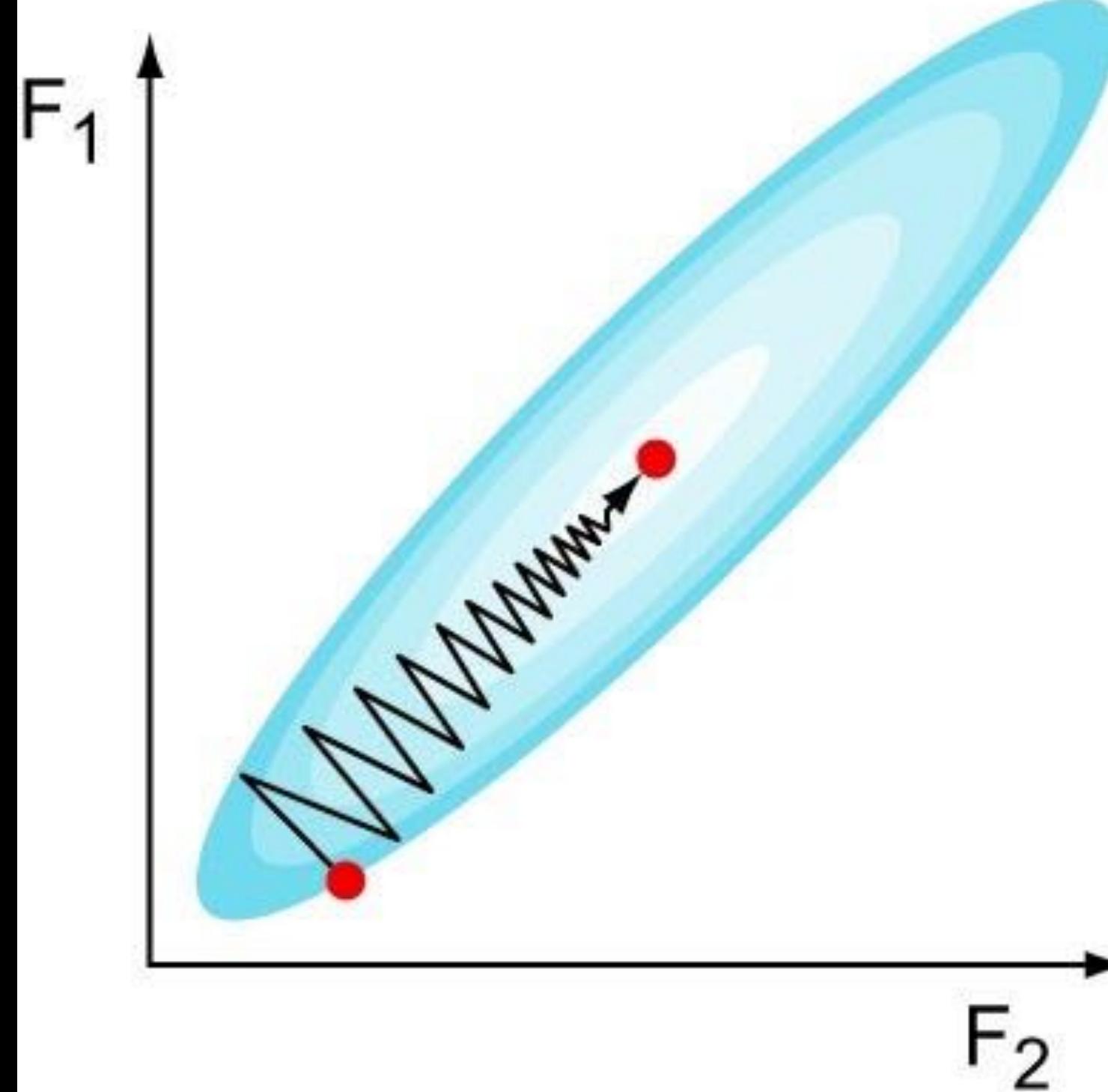
## Winning setup in MADS hackathon

- With hard voting classifier on:
  - Transformer model trained on SMOTE
  - Transformer model trained on manual upsampled dataset
  - BiGRU with upsampled data on only training set
  - BiGRU with upsampled data on both sets

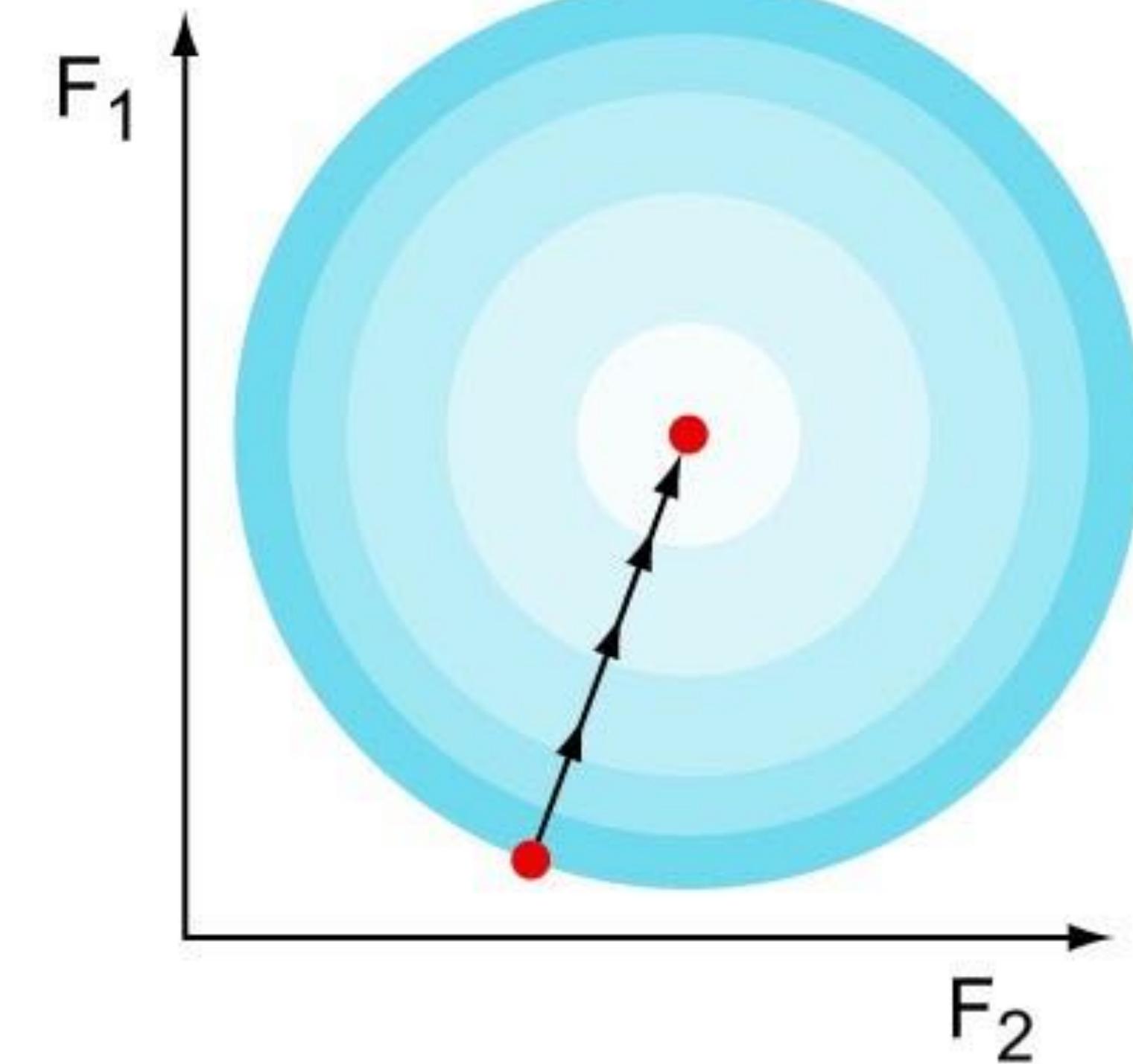
# Normalization

Gradient descent with and without feature scaling

Non-normalized features



Normalized features



# Normalization Layers

`nn.BatchNorm1d`

Applies Batch Normalization over a 2D or 3D input.

`nn.BatchNorm2d`

Applies Batch Normalization over a 4D input.

`nn.BatchNorm3d`

Applies Batch Normalization over a 5D input.

`nn.LazyBatchNorm1d`

A `torch.nn.BatchNorm1d` module with lazy initialization.

`nn.LazyBatchNorm2d`

A `torch.nn.BatchNorm2d` module with lazy initialization.

`nn.LazyBatchNorm3d`

A `torch.nn.BatchNorm3d` module with lazy initialization.

`nn.GroupNorm`

Applies Group Normalization over a mini-batch of inputs.

`nn.SyncBatchNorm`

Applies Batch Normalization over a N-Dimensional input.

`torch.nn`

+ Containers

Convolution Layers

Pooling layers

Padding Layers

Non-linear Activations (weighted sum, nonlinearity)

Non-linear Activations (other)

Normalization Layers

Recurrent Layers

Transformer Layers

Linear Layers

Dropout Layers

Sparse Layers

Distance Functions

Loss Functions

Vision Layers

Shuffle Layers

DataParallel Layers (multi-GPU, distributed)

+ Utilities

Quantized Functions

+ Lazy Modules Initialization

# Dropout

- `torch.nn.Dropout(p=0.1)`
- Randomly drop weights in order to avoid overspecialised parameters
- $\hat{w}_j = \begin{cases} w_j & \text{with } p(\theta) \\ 0 & \text{otherwise} \end{cases}$
- If you are juggling with a group and everyone might drop a ball at random, you will be more alert

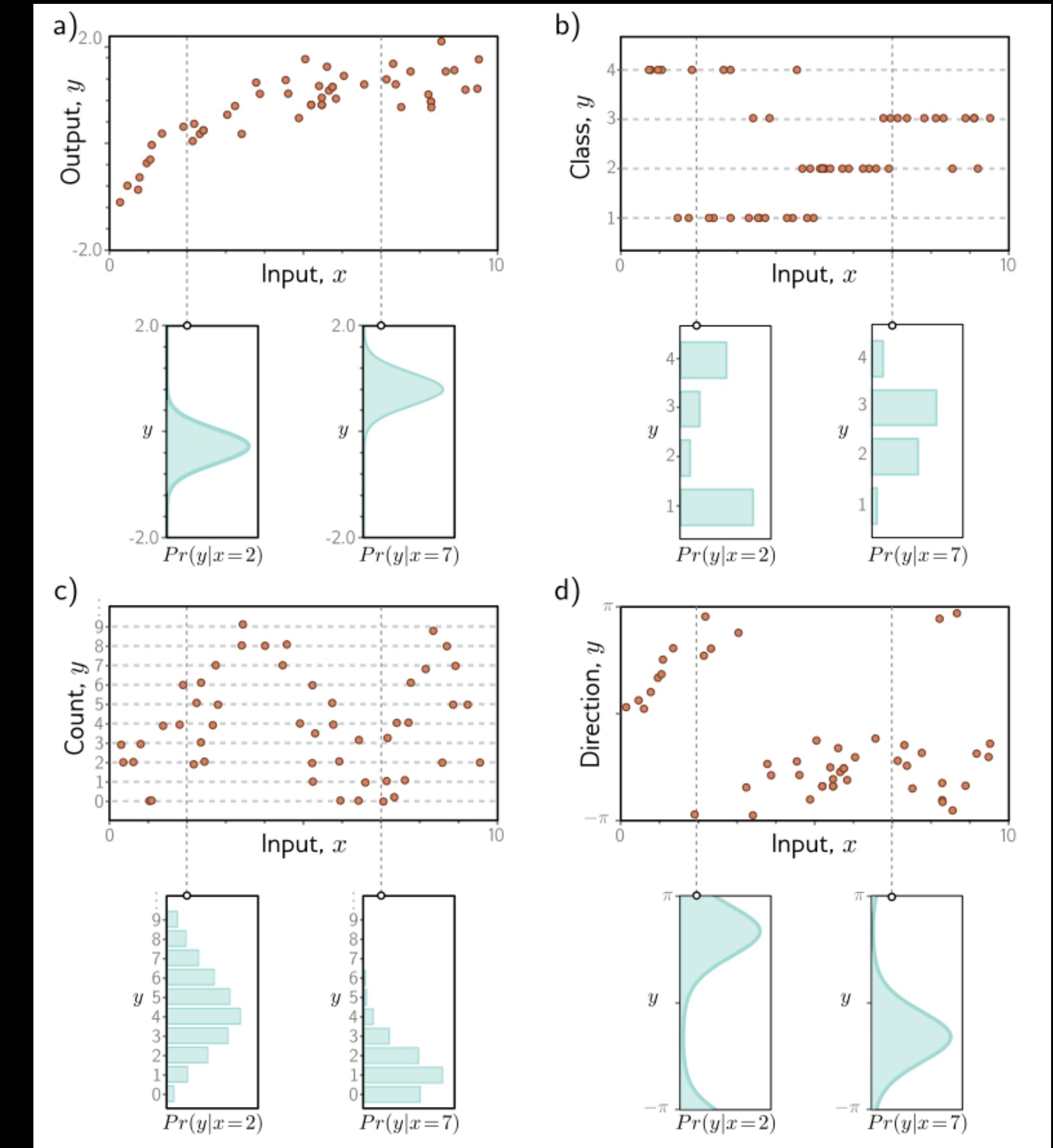




# Predicting outputs

# Losses

- Machine learning tries to find a mapping  $f: X \rightarrow y$
- The model predicts a conditional probability distribution  $P(y|x)$
- The loss function quantifies the mismatch between prediction  $\hat{y}$  and  $y$



# Losses

- For regression, we can assume a normal distribution
- We can simplify the pdf of the normal distribution
- See notebook ‘losses’ for more loss functions.

$$P(y | \mu, \sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp \frac{-(y - \mu)^2}{2\sigma^2}$$

$$P(y | \mu, \sigma) = -(y - \hat{y})^2$$

$$\mathcal{L} = \sum_{i=1}^N (y_i - \hat{y}_i)^2$$

# Convolutions

$O(n^2)$

# Width x Height



28x28



100x100



200x200



400x400

## Width x Height

## Features



28x28

784



100x100

10.000



200x200

40.000



400x400

160.000

Width x Height

Features

Weights



28x28

784

614.656



100x100

10.000

100.000.000



200x200

40.000

1.600.000.000



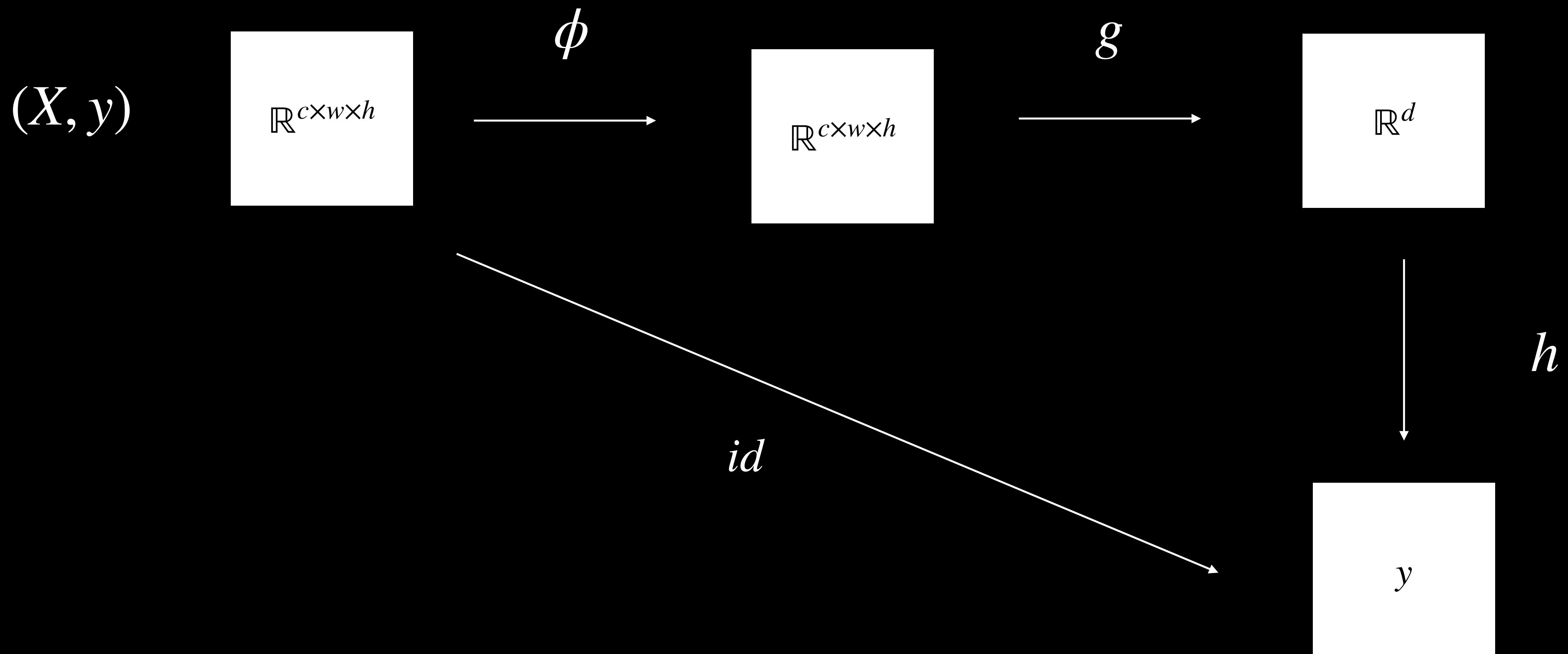
400x400

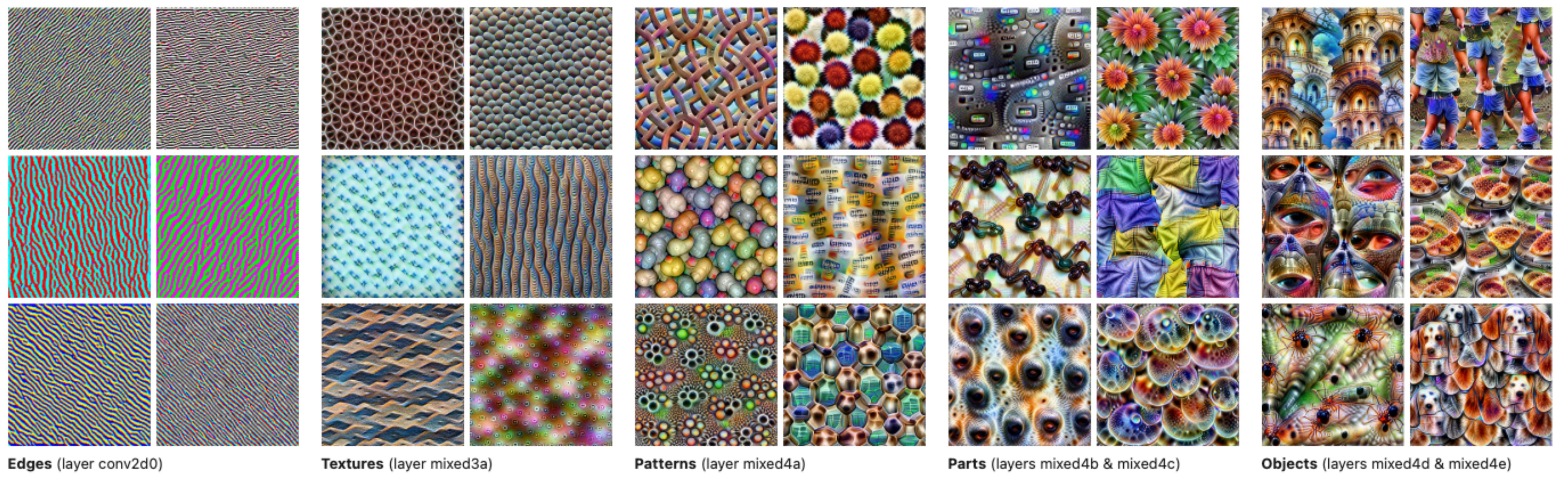
160.000

25.600.000.000

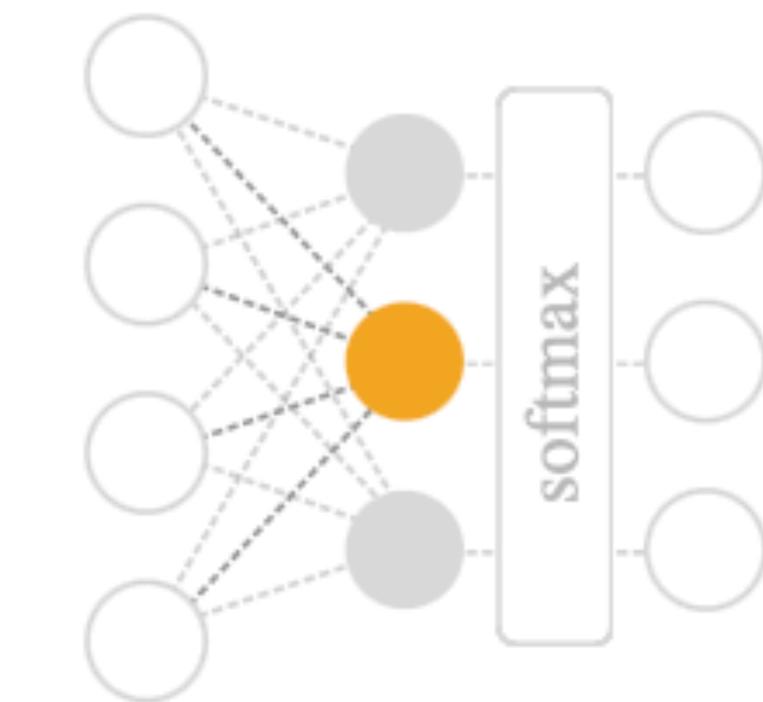
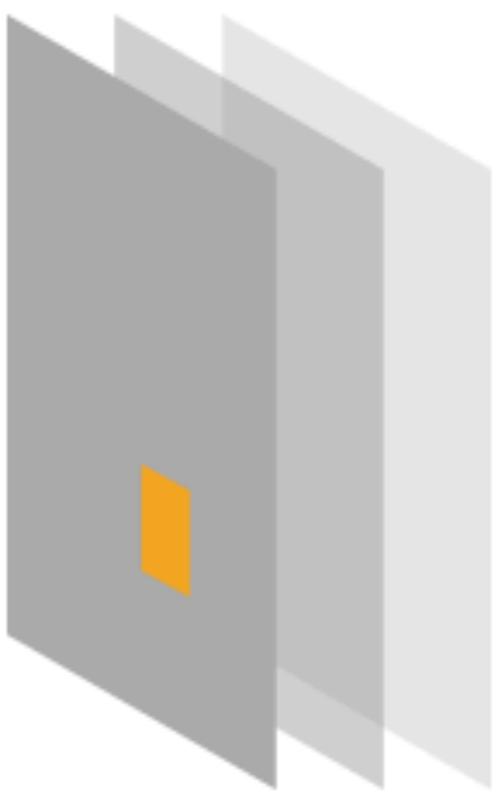


# Supervised



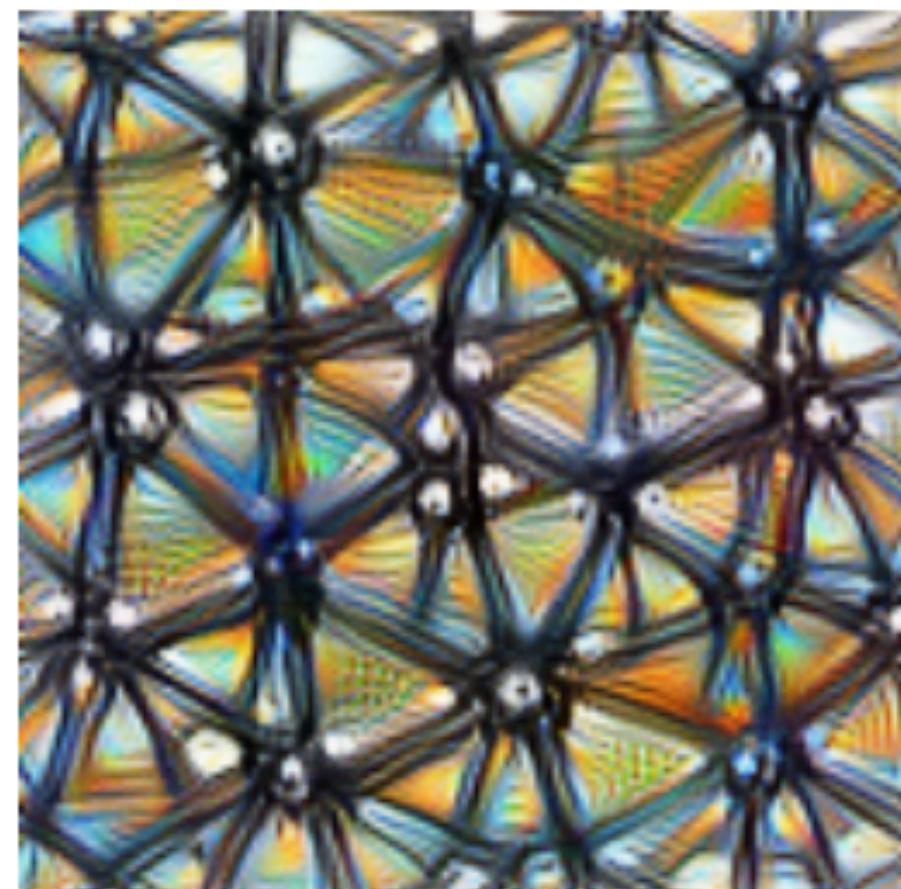


<https://distill.pub/2017/feature-visualization/>



**Neuron**

```
layern[x,y,z]
```



**Channel**

```
layern[:, :, z]
```



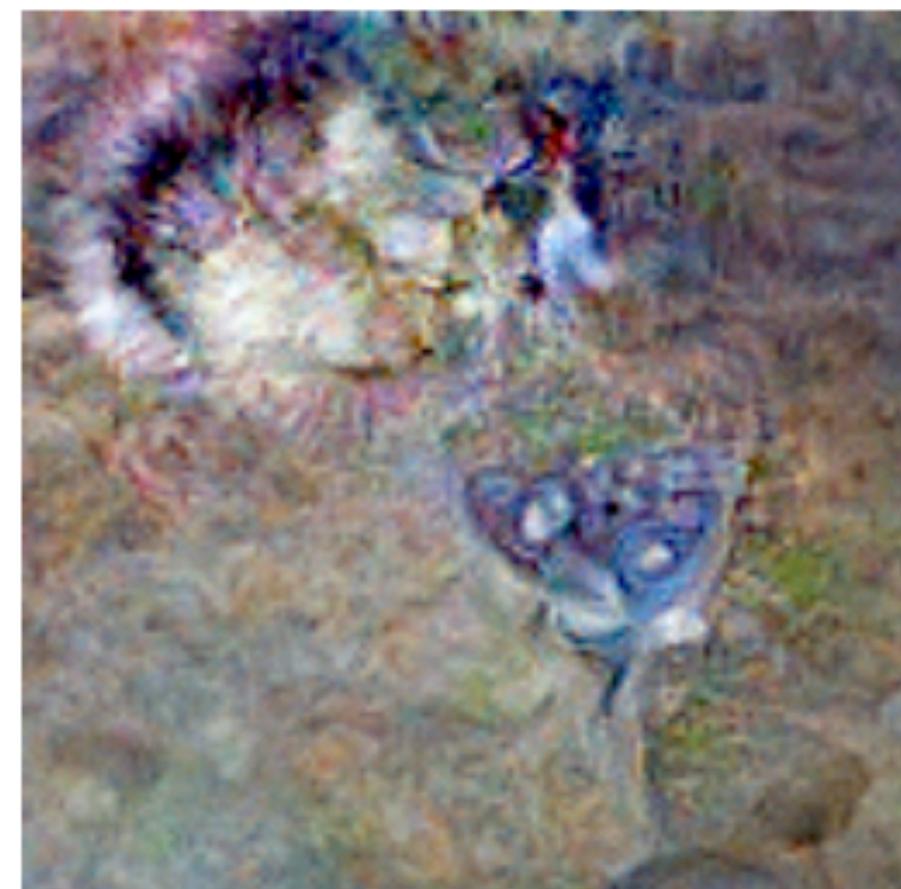
**Layer/DeepDream**

```
layern[:, :, :]2
```



**Class Logits**

```
pre_softmax[k]
```



**Class Probability**

```
softmax[k]
```

# Convolutions

# Convolutions

	1			
1	-1	1		
	-1			
1	-1			

0	0	0
0	1	0
0	0	0


# Convolutions

0	1	0	0	0
1	-1	1	0	0
0	0	0	0	0
0	-1	0	0	0
0	1	-1	0	0

0	0	0
0	1	0
0	0	0


# Convolutions

0	1	0	0	0
1	-1	1	0	0
0	0	0	0	0
0	-1	0	0	0
0	1	-1	0	0

0	0	0
0	1	0
0	0	0


# Convolutions

0	1	0	0	0
1	-1	1	0	0
0	0	0	0	0
0	-1	0	0	0
0	1	-1	0	0

0	0	0
0	1	0
0	0	0

-1		

# Convolutions

0	1	0	0	0
1	-1	1	0	0
0	0	0	0	0
0	-1	0	0	0
0	1	-1	0	0

0	0	0
0	1	0
0	0	0

-1	1	

# Convolutions

0	1	0	0	0
1	-1	1	0	0
0	0	0	0	0
0	-1	0	0	0
0	1	-1	0	0

0	0	0
0	1	0
0	0	0

-1	1	0

# Convolutions

0	1	0	0	0
1	-1	1	0	0
0	0	0	0	0
0	-1	0	0	0
0	1	-1	0	0

0	0	0
0	1	0
0	0	0

-1	1	0
0		

# Convolutions

0	1	0	0	0
1	-1	1	0	0
0	0	0	0	0
0	-1	0	0	0
0	1	-1	0	0

0	0	0
0	1	0
0	0	0

-1	1	0
0	0	

# Convolutions

0	1	0	0	0
1	-1	1	0	0
0	0	0	0	0
0	-1	0	0	0
0	1	-1	0	0

0	0	0
0	1	0
0	0	0

-1	1	0
0	0	0

# Convolutions

0	1	0	0	0
1	-1	1	0	0
0	0	0	0	0
0	-1	0	0	0
0	1	-1	0	0

0	0	0
0	1	0
0	0	0

-1	1	0
0	0	0
-1		

# Convolutions

0	1	0	0	0
1	-1	1	0	0
0	0	0	0	0
0	-1	0	0	0
0	1	-1	0	0

0	0	0
0	1	0
0	0	0

-1	1	0
0	0	0
-1	0	

# Convolutions

0	1	0	0	0
1	-1	1	0	0
0	0	0	0	0
0	-1	0	0	0
0	1	-1	0	0

0	0	0
0	1	0
0	0	0

-1	1	0
0	0	0
-1	0	0

# Convolutions

0	1	0	0	0
1	-1	1	0	0
0	0	0	0	0
0	-1	0	0	0
0	1	-1	0	0

0	0	0
0	1	0
0	0	0

-1	1	0
0	0	0
-1	0	0

Identity kernel

# Convolutions

0.4	0.1	-0.1	-0.7	0
0.5	0.4	0.1	-0.1	0.6
0.1	0.7	0	0.4	0.2
0.9	-1	0.4	0.1	-0.1
0.1	0.2	-0.9	-0.1	0

0.2	0.3	0.2
0.2	0.5	0.2
-0.2	0.3	0.2

Learnable

$\hat{y}$	$\hat{y}$	$\hat{y}$
$\hat{y}$	$\hat{y}$	$\hat{y}$
$\hat{y}$	$\hat{y}$	$\hat{y}$

# Convolutions

x	x	x	x	x
x	x	x	x	x
x	x	x	x	x
x	x	x	x	x
x	x	x	x	x

w	w	w
w	w	w
w	w	w

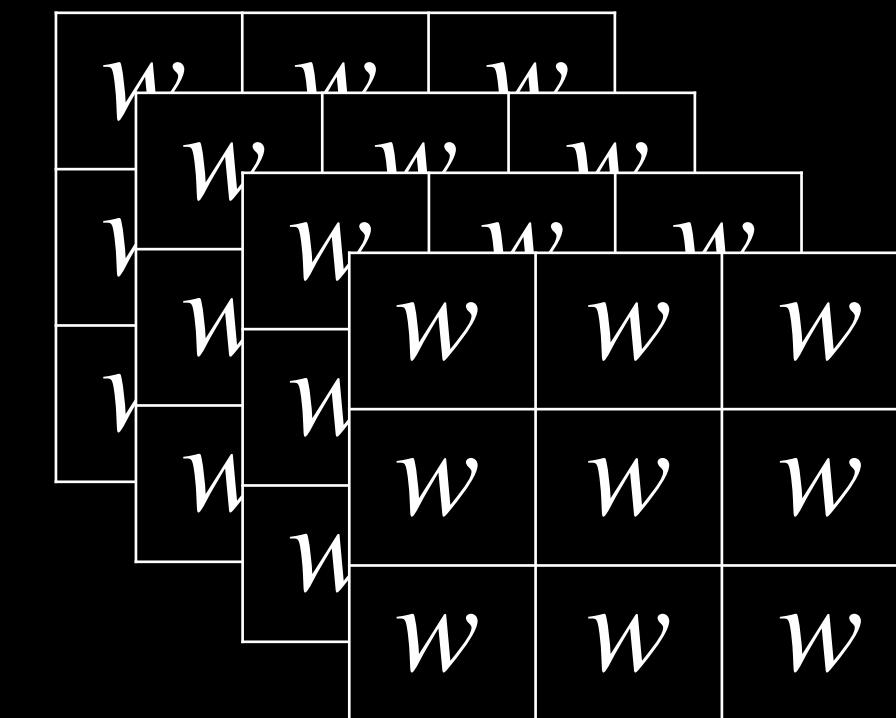
Learnable

$\hat{y}$	$\hat{y}$	$\hat{y}$
$\hat{y}$	$\hat{y}$	$\hat{y}$
$\hat{y}$	$\hat{y}$	$\hat{y}$

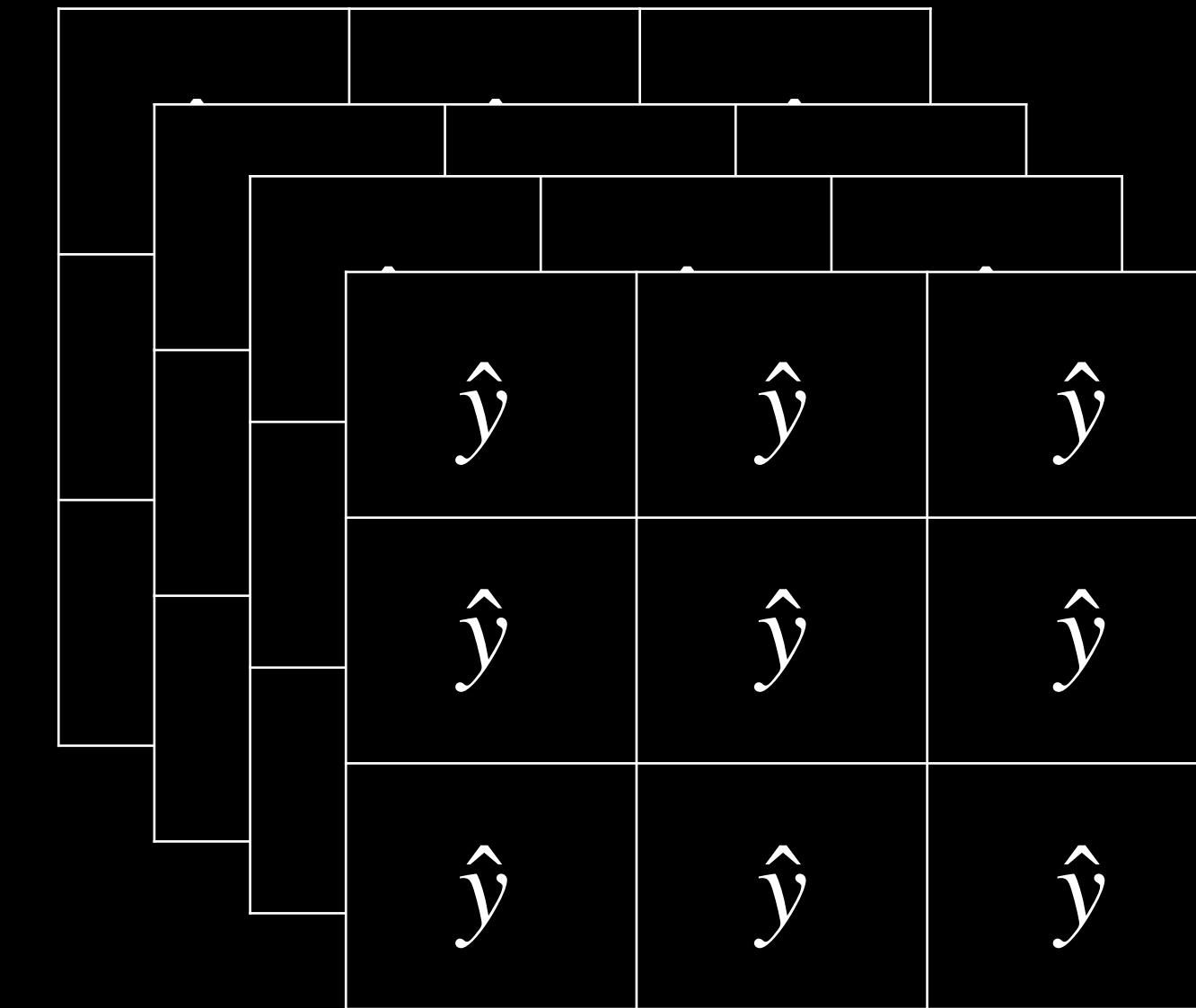
# Convolutions

x	x	x	x	x
x	x	x	x	x
x	x	x	x	x
x	x	x	x	x
x	x	x	x	x

Input



4 ( $1 \times 3 \times 3$ ) kernels

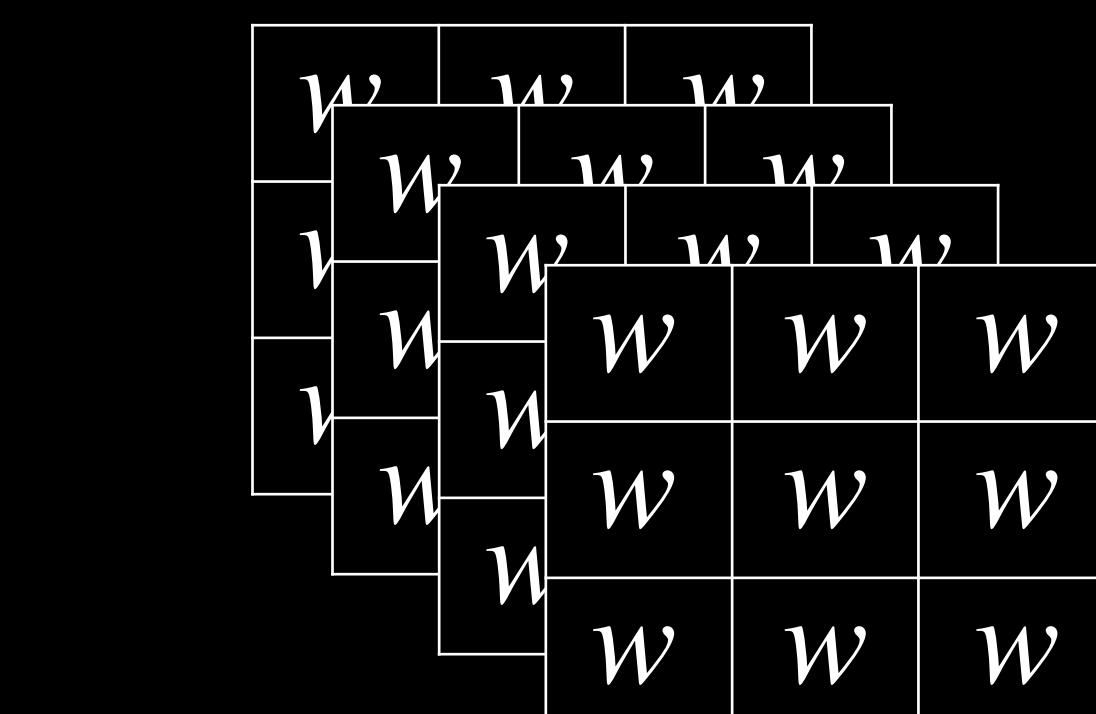


4 feature maps

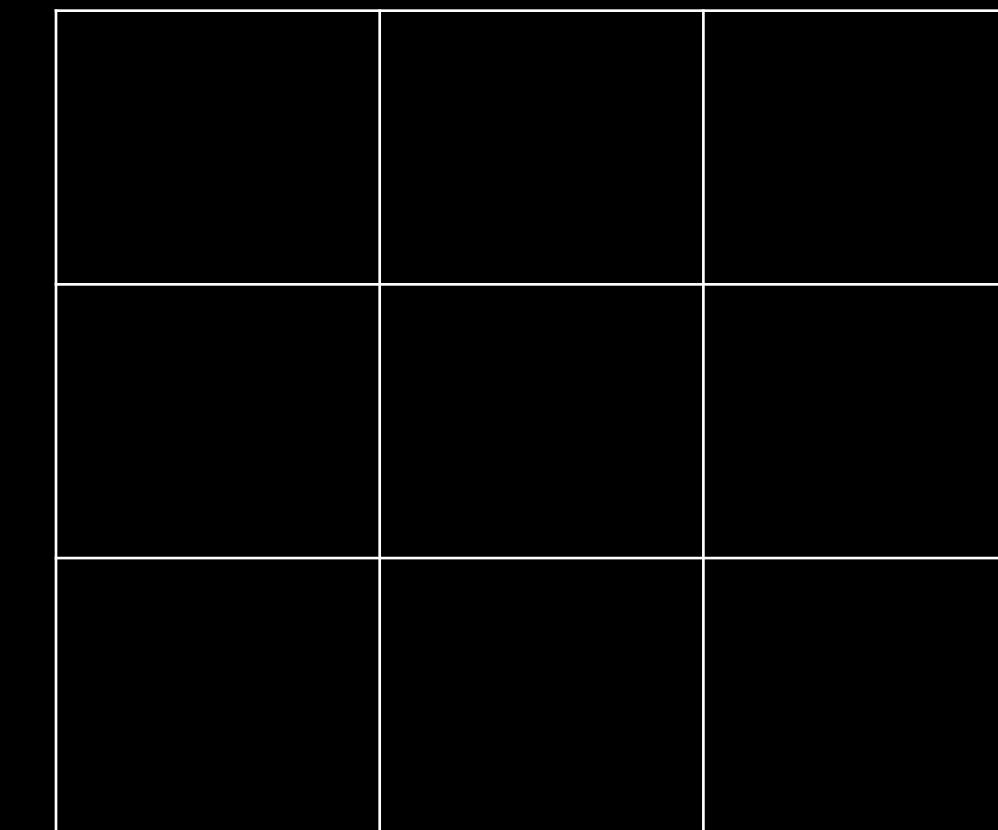
# Convolutions

1	1	0	0	0
0	1	1	0	0
1	0.5	1	0	0
1	-1	1	0	0
0	0	0	0	0
0	0	0	0	0
0	0	-1	0	0
0	1	-1	0	0

4 feature maps

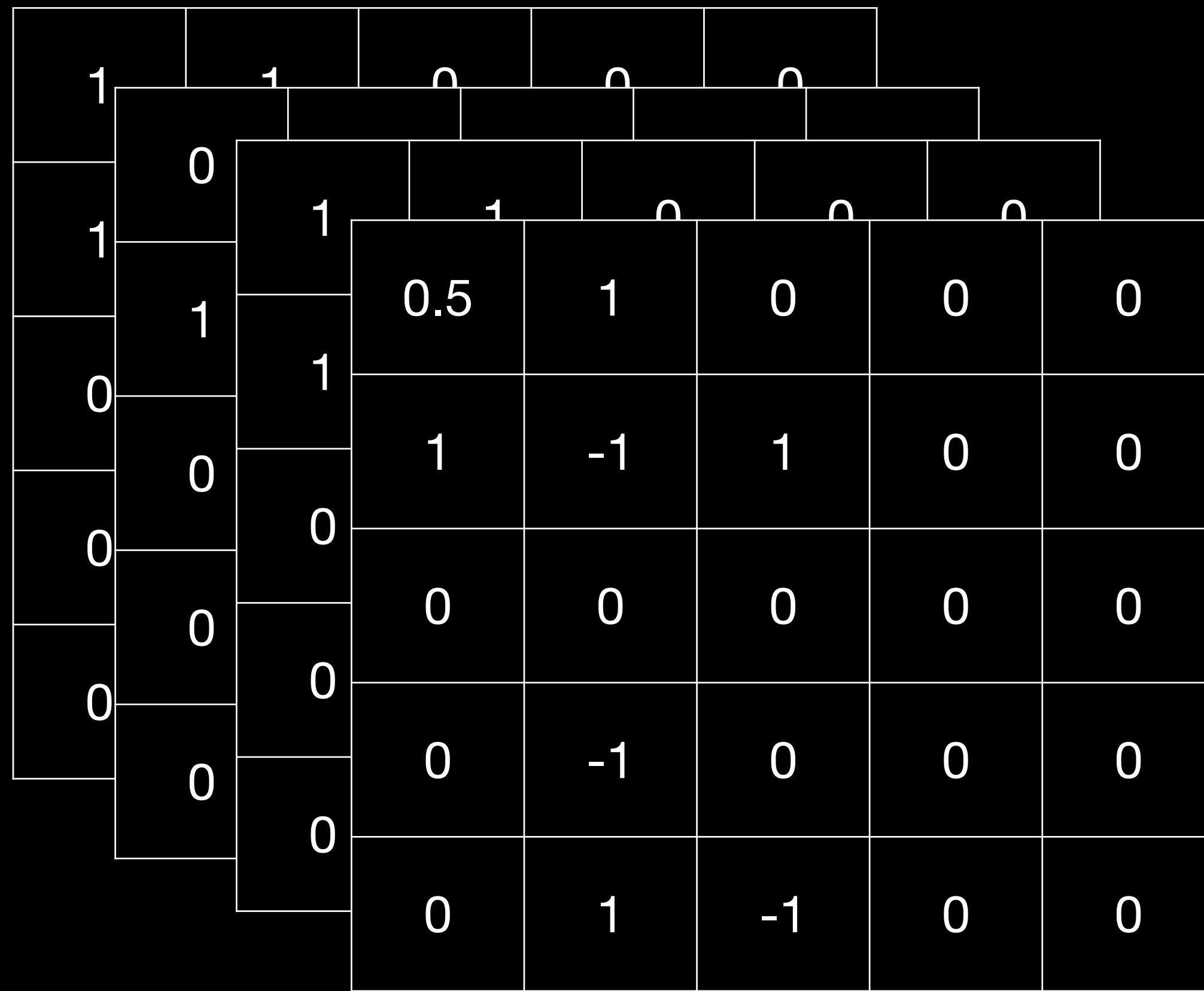


1 ( $4 \times 3 \times 3$ ) kernel

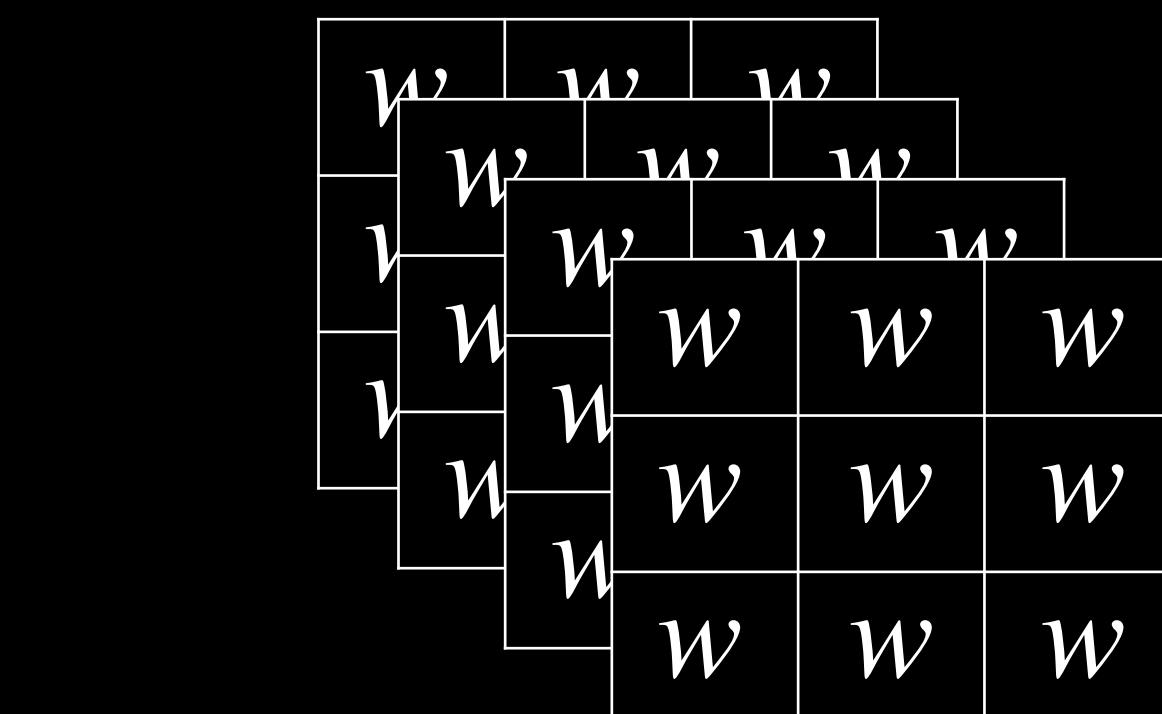


1 feature map

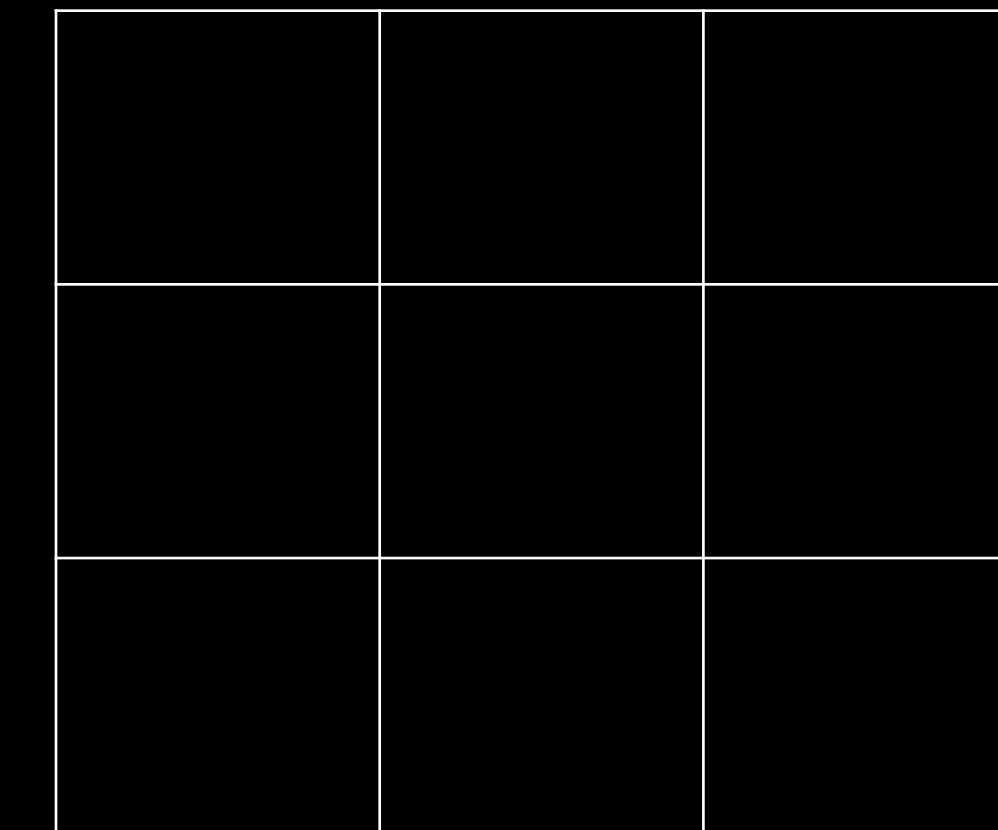
# Convolutions



$n(c \times w \times h)$  feature maps



$m(c \times k \times k)$  kernel

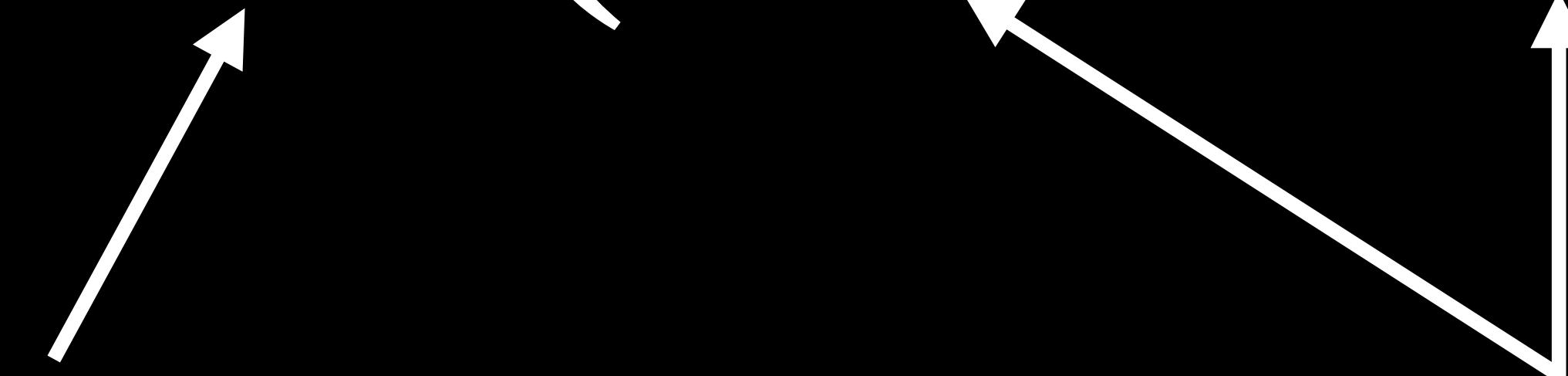


$m(1 \times w \times h)$  feature map

$O(k)$

# Neural networks

$$\sigma(wx + b)$$



Non-linear transformation

Linear transformation

# Convolutions

$$\sigma(w * x)$$

The diagram illustrates the components of a convolution operation. At the top, the mathematical expression  $\sigma(w * x)$  is shown in white. Below it, two arrows point upwards towards the expression: one from the left pointing to the  $w$  term, labeled "Non-linear transformation", and one from the right pointing to the  $x$  term, labeled "Linear transformation".

# Convolutions

## Elements of a Convolution

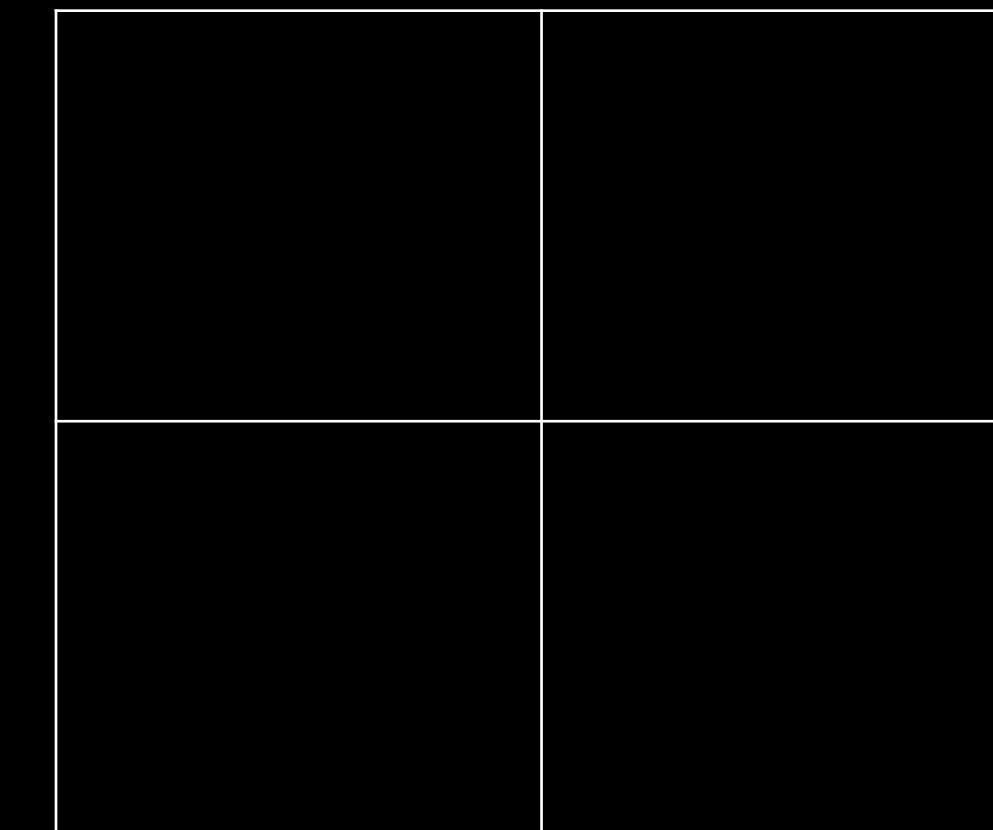
- Kernel or filter
- Stride
- Padding

# Convolutions

Kernelsize =  $4 \times 4$

0	1	0	0	0
1	-1	1	0	0
0	0	0	0	0
0	-1	0	0	0
0	1	-1	0	0

0	0	0	0
0	1	1	0
0	1	1	0
0	0	0	0



# Convolutions

Stride = 2



0	1	0	0	0
1	-1	1	0	0
0	0	0	0	0
0	-1	0	0	0
0	1	-1	0	0

0	0	0
0	1	0
0	0	0


# Convolutions

Stride = 2



0	1	0	0	0
1	-1	1	0	0
0	0	0	0	0
0	-1	0	0	0
0	1	-1	0	0

0	0	0
0	1	0
0	0	0


# Convolutions

Padding = (1,1)

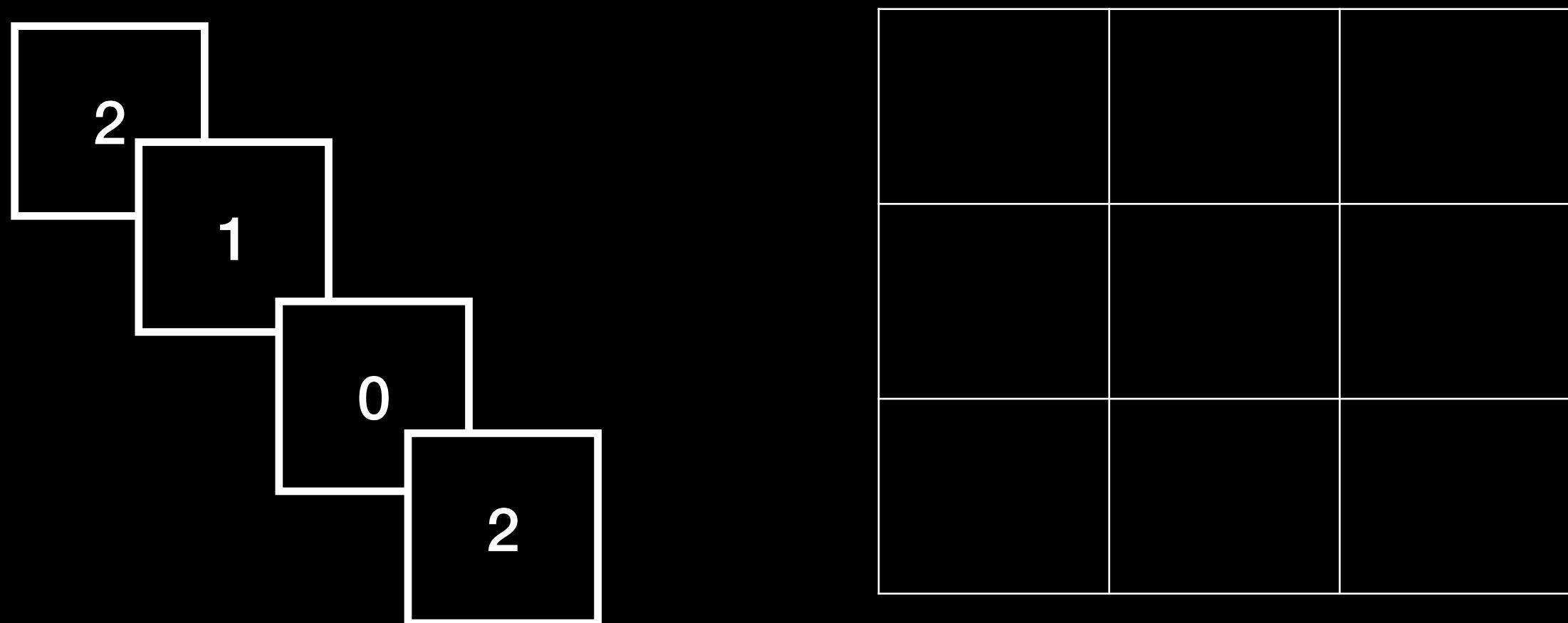
0	0	0	0	0	0	0
0	X	X	X	X	X	0
0	X	X	X	X	X	0
0	X	X	X	X	X	0
0	X	X	X	X	X	0
0	X	X	X	X	X	0
0	0	0	0	0	0	0

0	0	0
0	1	0
0	0	0


# Convolutions

$1 \times 1$  kernel

1		1	0	0	0
0					
1		1	1	0	0
1		0.5	1	0	0
1		1	-1	1	0
0		0	0	0	0
0		0	0	0	0
0		0	-1	0	0
0		0	1	-1	0
0		0			

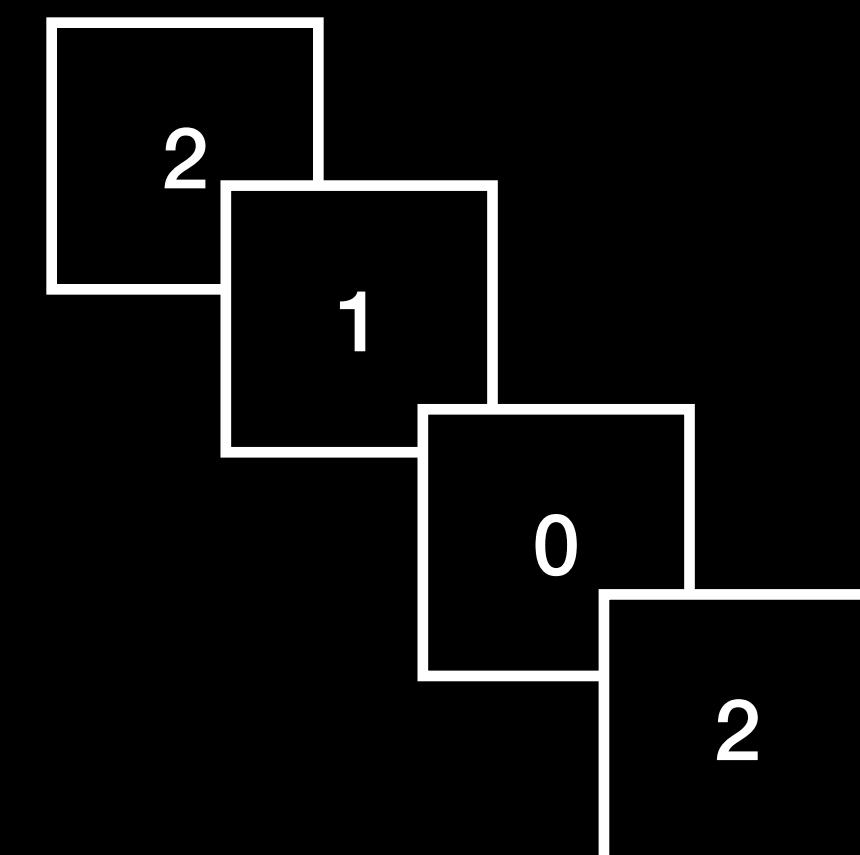


4 feature maps

# Convolutions

$1 \times 1$  kernel

1	1	0	0	0	0	0
0	0	1	1	0	0	0
1	1	0.5	1	0	0	0
1	1	1	-1	1	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	-1	0	0	0
0	0	1	-1	-1	0	0



$$1 \times 2 + 0 \times 1 + 1 \times 0 + 0.5 \times 2 = 3$$

3				

$n(c \times w \times h)$  feature maps

$m(c \times k \times k)$  kernel

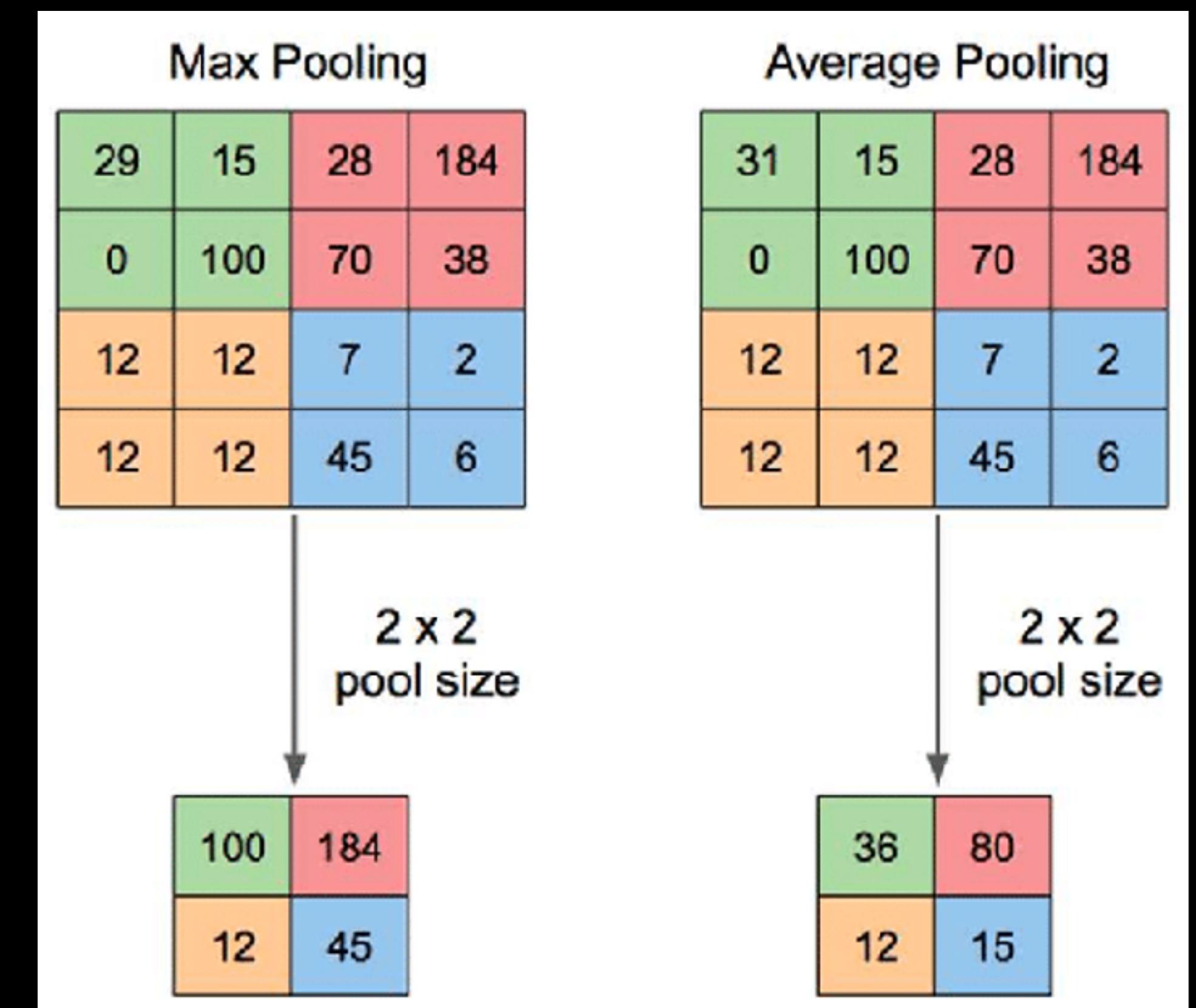
$m(1 \times w \times h)$  feature map

# 1x1 convolutions

- While they can not capture spatial patterns, they will capture patterns along the **depth dimension**
- When configured to reduce the depth, they **reduce dimensionality**
- We can add extra activations, adding more **non-linearities**.

# Pooling

- A way to downsample the input
- no learnable parameters
- **max pooling:** focus on the highest value
- **average pooling:** smooth the pixels



```
class ConvBlock(nn.Module):
    def __init__(self, in_channels, out_channels, kernel_size):
        super().__init__()
        self.conv = nn.Sequential(
            nn.Conv2d(
                in_channels, out_channels, kernel_size=kernel_size, stride=1, padding=1
            ),
            nn.ReLU(),
            nn.Conv2d(
                out_channels, out_channels, kernel_size=kernel_size, stride=1, padding=1
            ),
            nn.ReLU(),
        )

    def forward(self, x):
        return self.conv(x)
```

mltrainer v0.1.129 - imagemodels.py

```
class CNNblocks(nn.Module):
    def __init__(self, config: CNNConfig) → None:
        super().__init__()
        input_channels = config.input_channels
        kernel_size = config.kernel_size
        hidden = config.hidden
        # first convolution
        self.convolutions = nn.ModuleList(
            [
                ConvBlock(input_channels, hidden, kernel_size),
            ]
        )

        # additional convolutions
        pool = config.maxpool
        num_maxpools = 0
        for i in range(config.num_layers):
            self.convolutions.extend(
                [ConvBlock(hidden, hidden, kernel_size), nn.ReLU()]
            )
            # every two layers, add a maxpool
            if i % 2 == 0:
                num_maxpools += 1
                self.convolutions.append(nn.MaxPool2d(pool, pool))

        # let's try to calculate the size of the linear layer
        # please note that changing stride/padding will change the logic
        matrix_size = (config.matrixshape[0] // (pool**num_maxpools)) * (
            config.matrixshape[1] // (pool**num_maxpools)
        )
        print(f"Calculated matrix size: {matrix_size}")
        print(f"Calculated flatten size: {matrix_size * hidden}")

        self.dense = nn.Sequential(
            nn.Flatten(),
            nn.Linear(matrix_size * hidden, hidden),
            nn.ReLU(),
            nn.Linear(hidden, config.num_classes),
        )

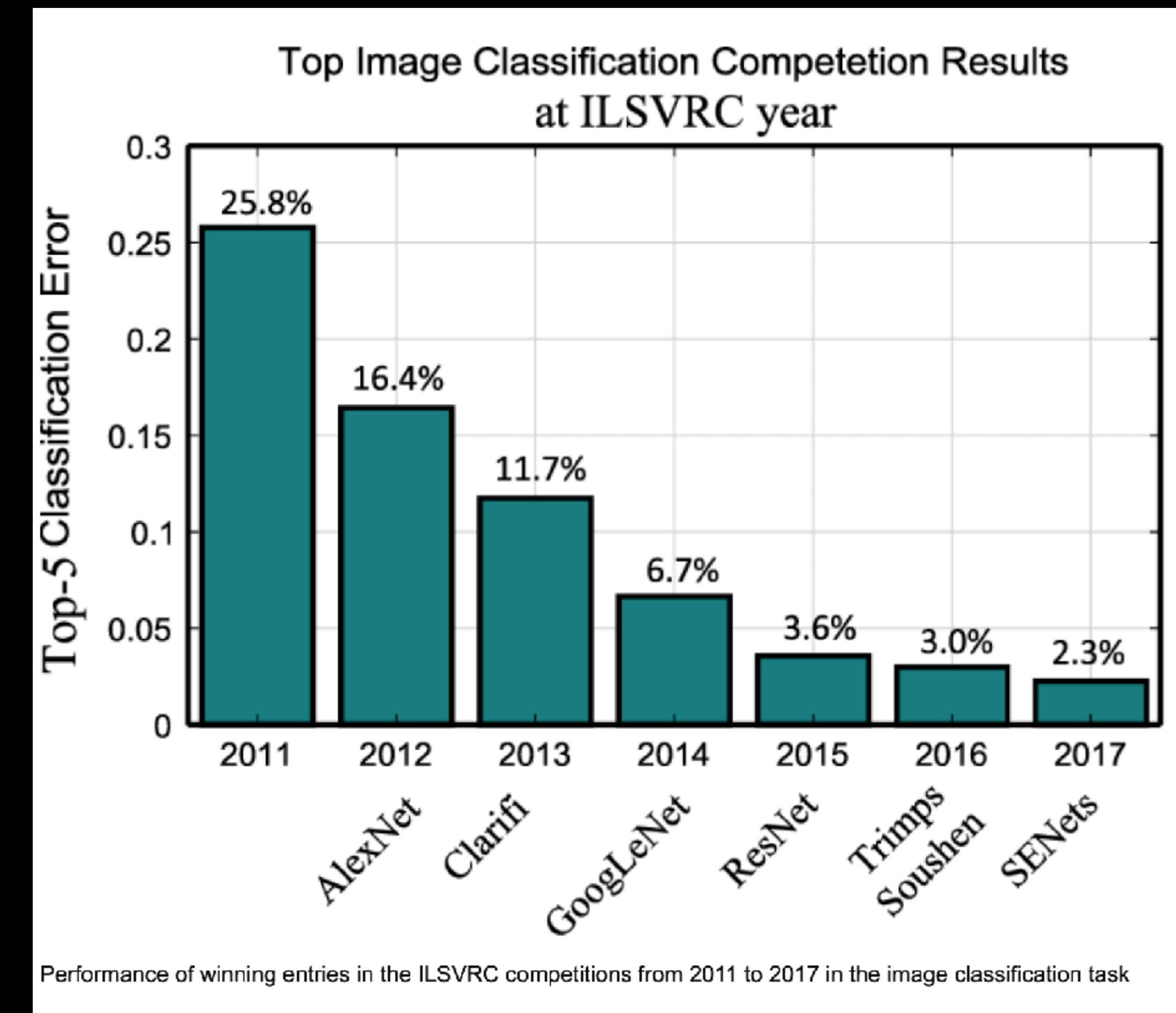
    def forward(self, x: torch.Tensor) → torch.Tensor:
        for conv in self.convolutions:
            x = conv(x)
        x = self.dense(x)
        return x
```

mltrainer v0.1.129 -  
imagemodels.py

# Overview of image architectures

# Imagenet Challenge

- Human performance is about 5%



2012

# AlexNet

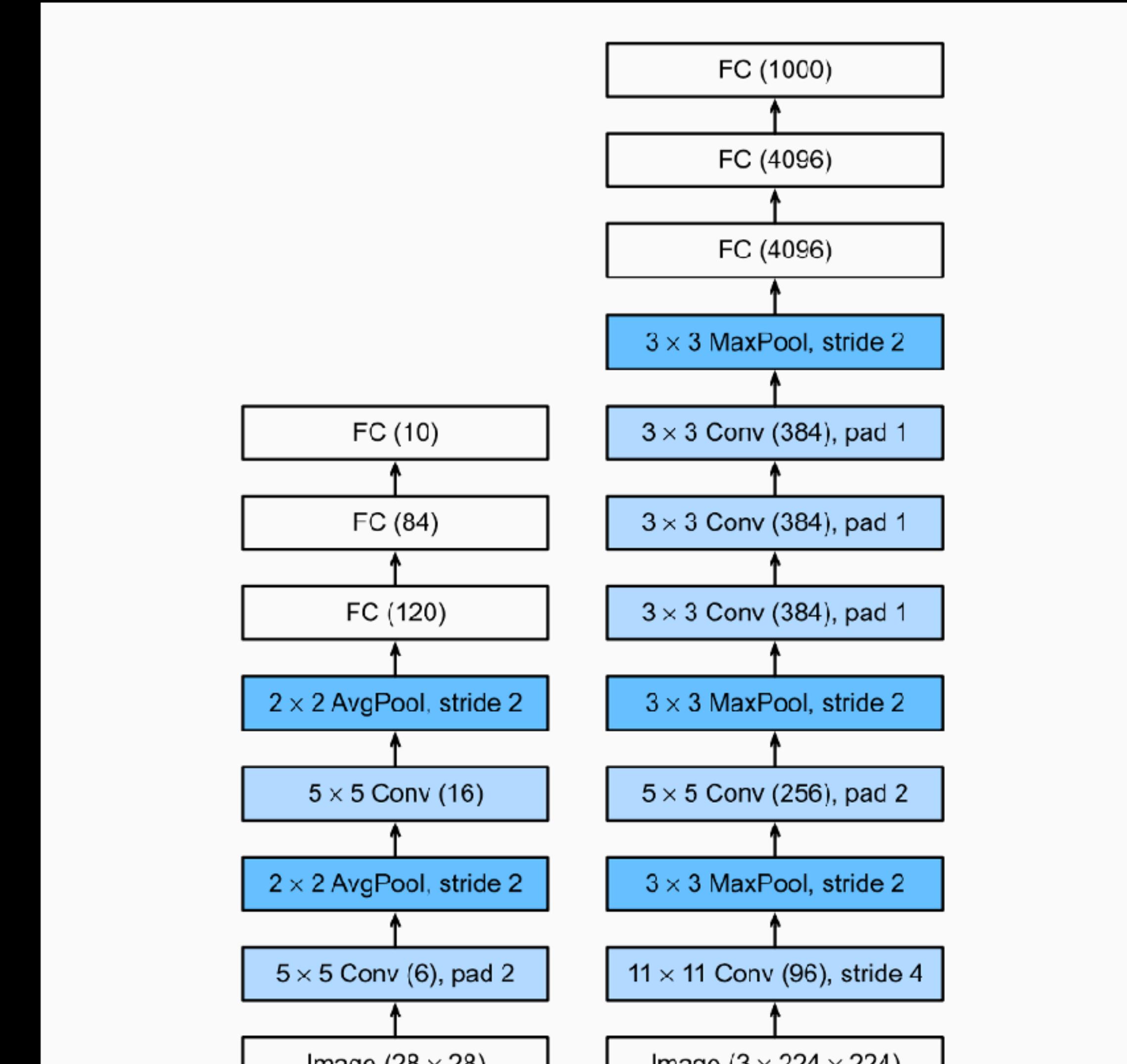
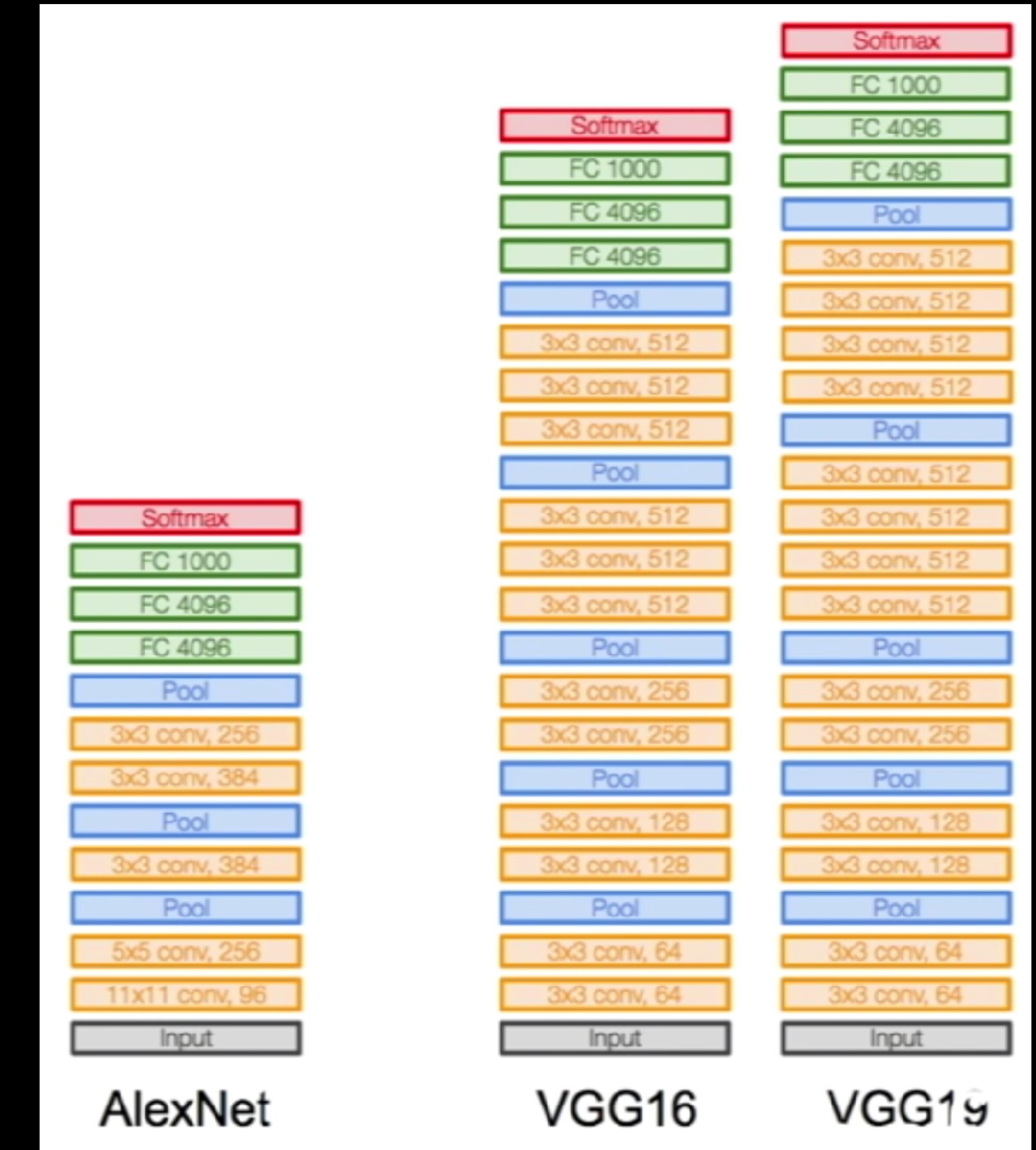


Fig. 7.1.2 From LeNet (left) to AlexNet (right).

# 2013

# VGG

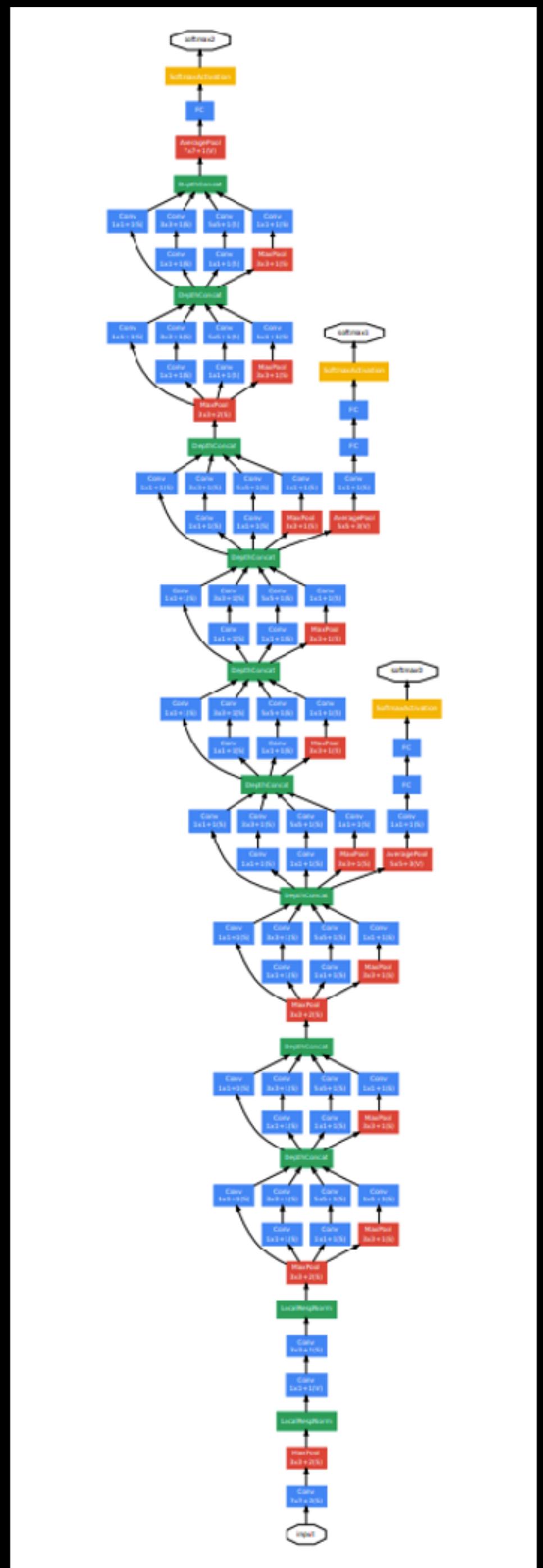
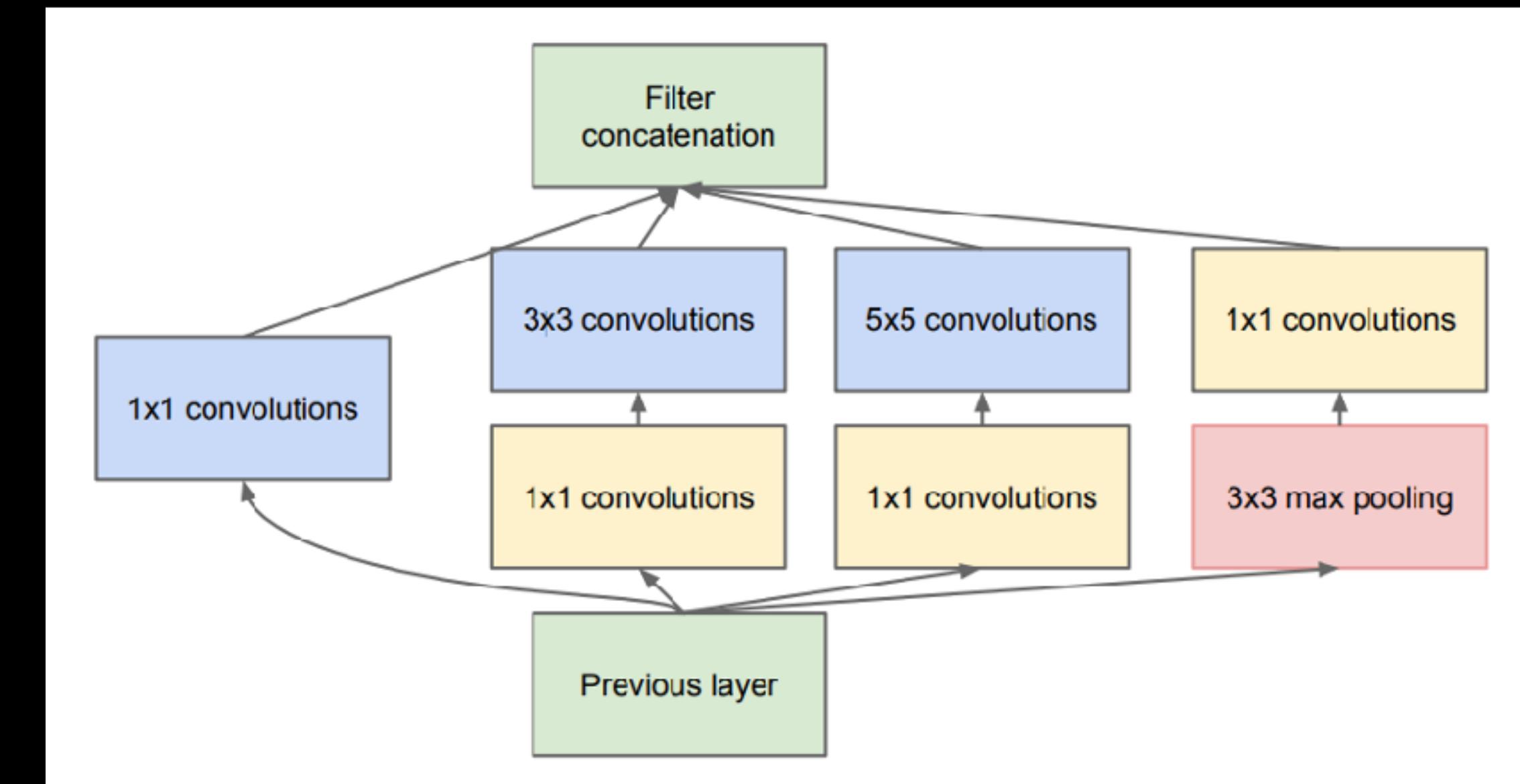
Changes in:  
**Filter size**  
**channel numbers**  
**depth**  
**138 million parameters**



# 2014

# GoogleNet

**Parallel filters**  
**Bottleneck layers**  
**6.7 million parameters**



# 2015

# ResNet

batch normalization

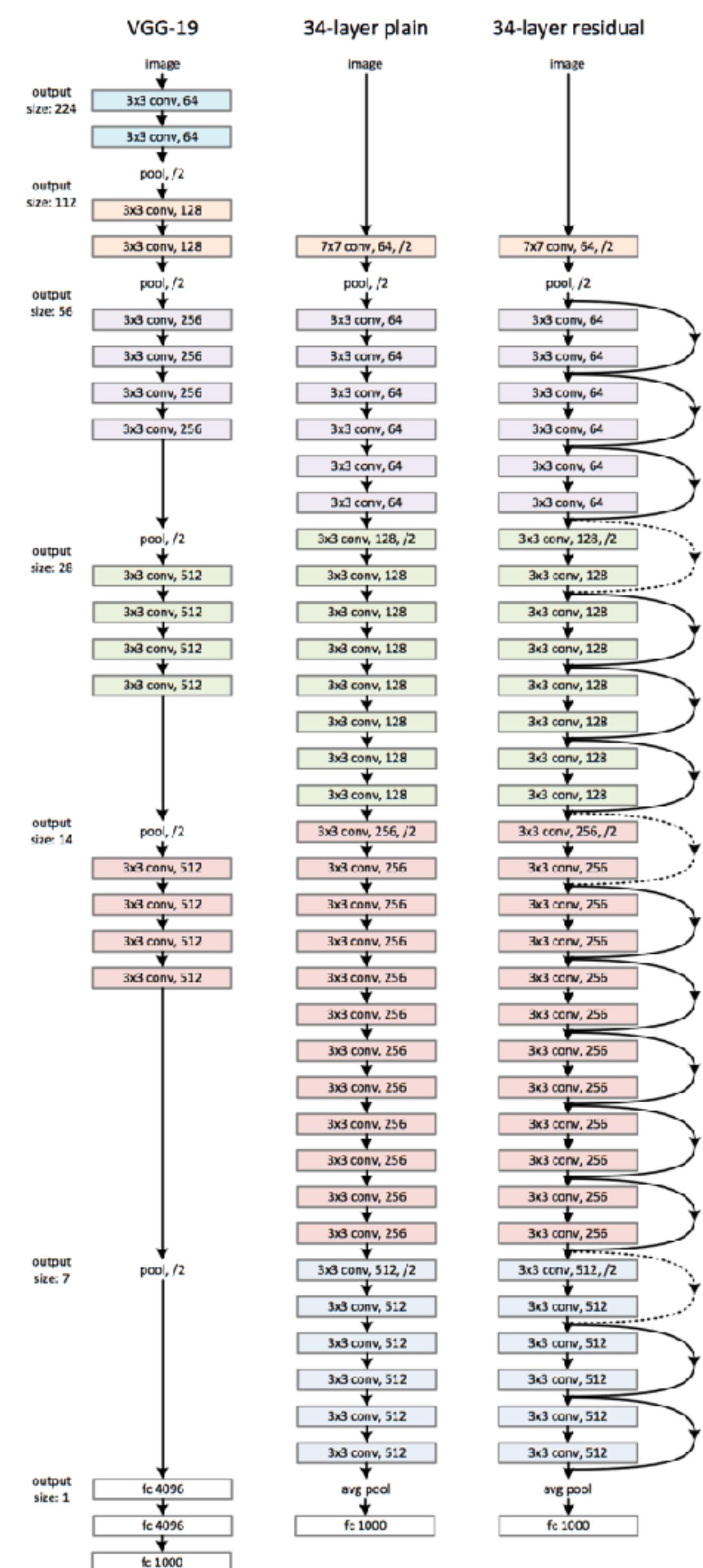
Residual blocks

11 million parameters for ResNet18

Dataset: ImageNet, 15milj images, 1000 classes

Paper: <https://arxiv.org/pdf/1512.03385>

He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 770-778).



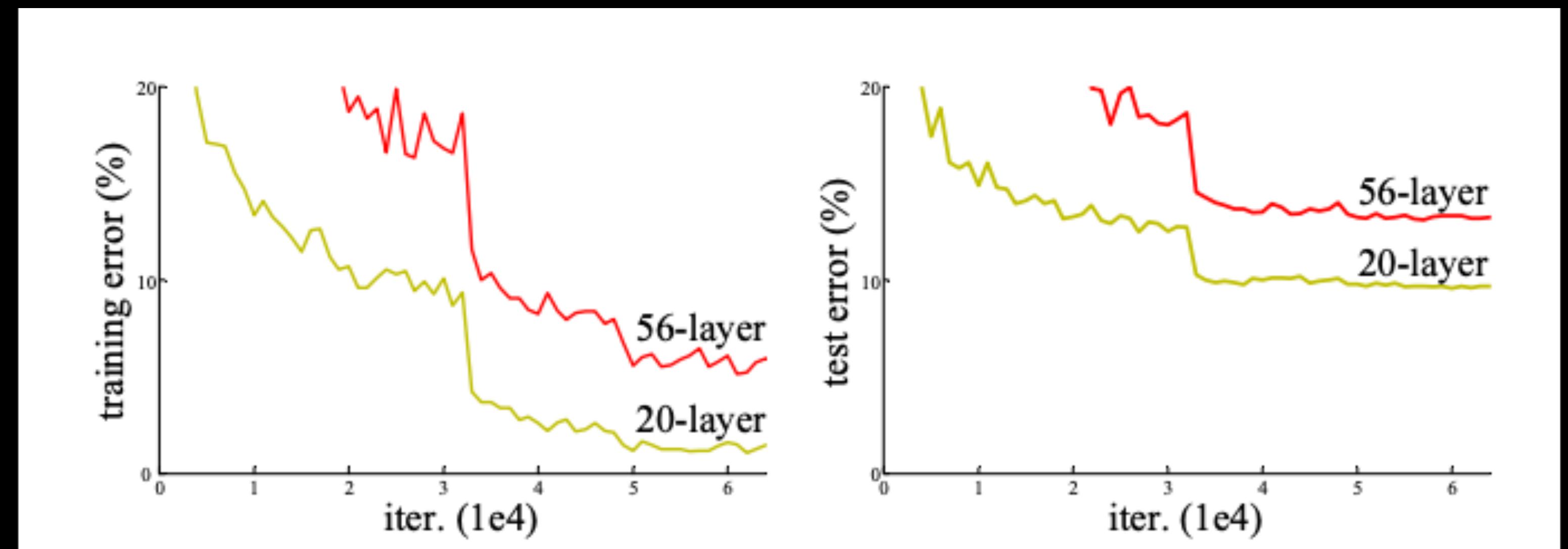
# ResNet

**What they noticed:** a deeper net (56 layers) was performing worse than a shallower net (20 layers), but not because of overfitting

**Hypothesis:** a deeper net is harder

**Idea:** don't learn the mapping, but learn the residual

**Residual:** what you need to change from the previous representation to achieve the mapping

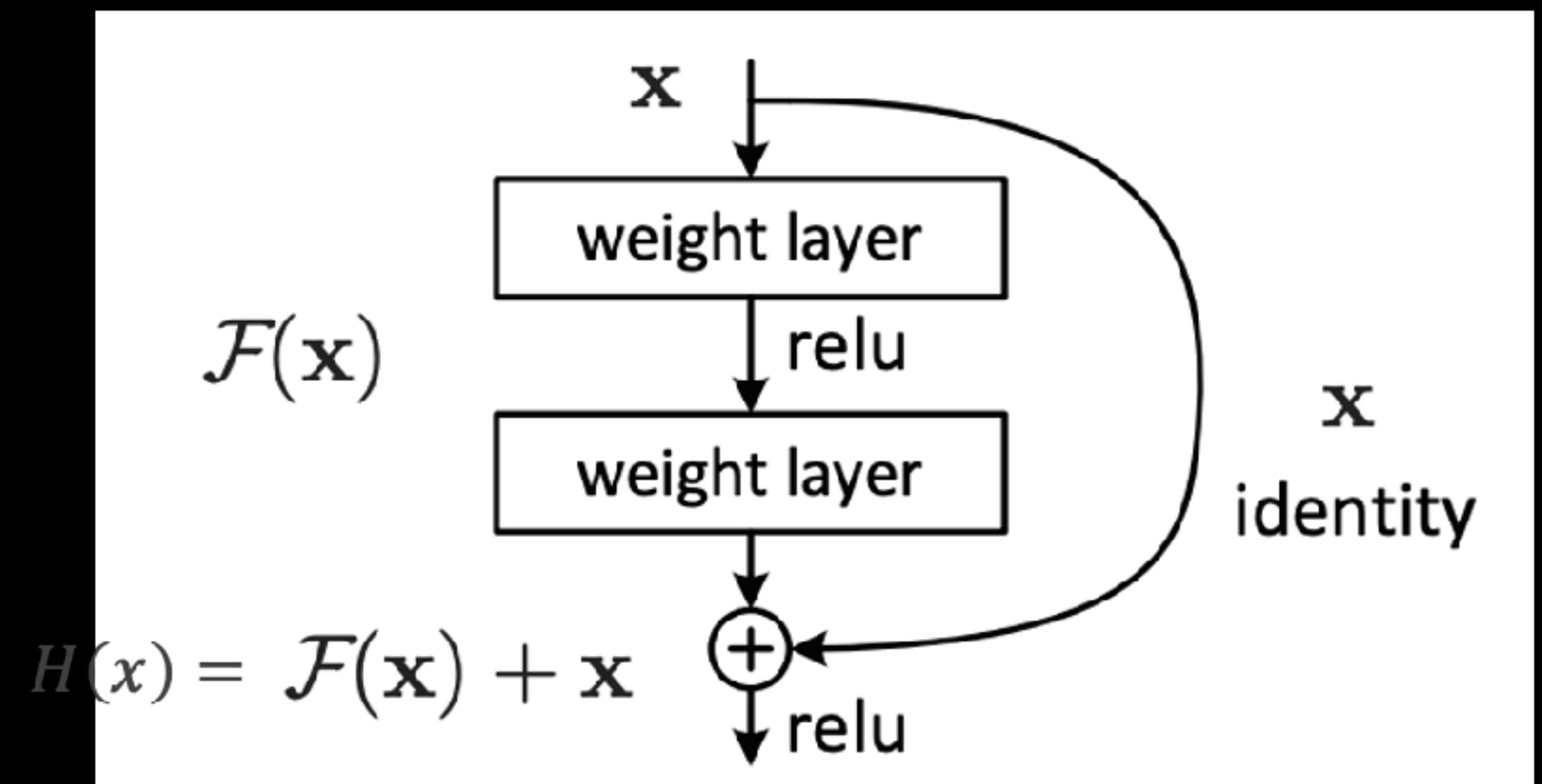


# ResNet

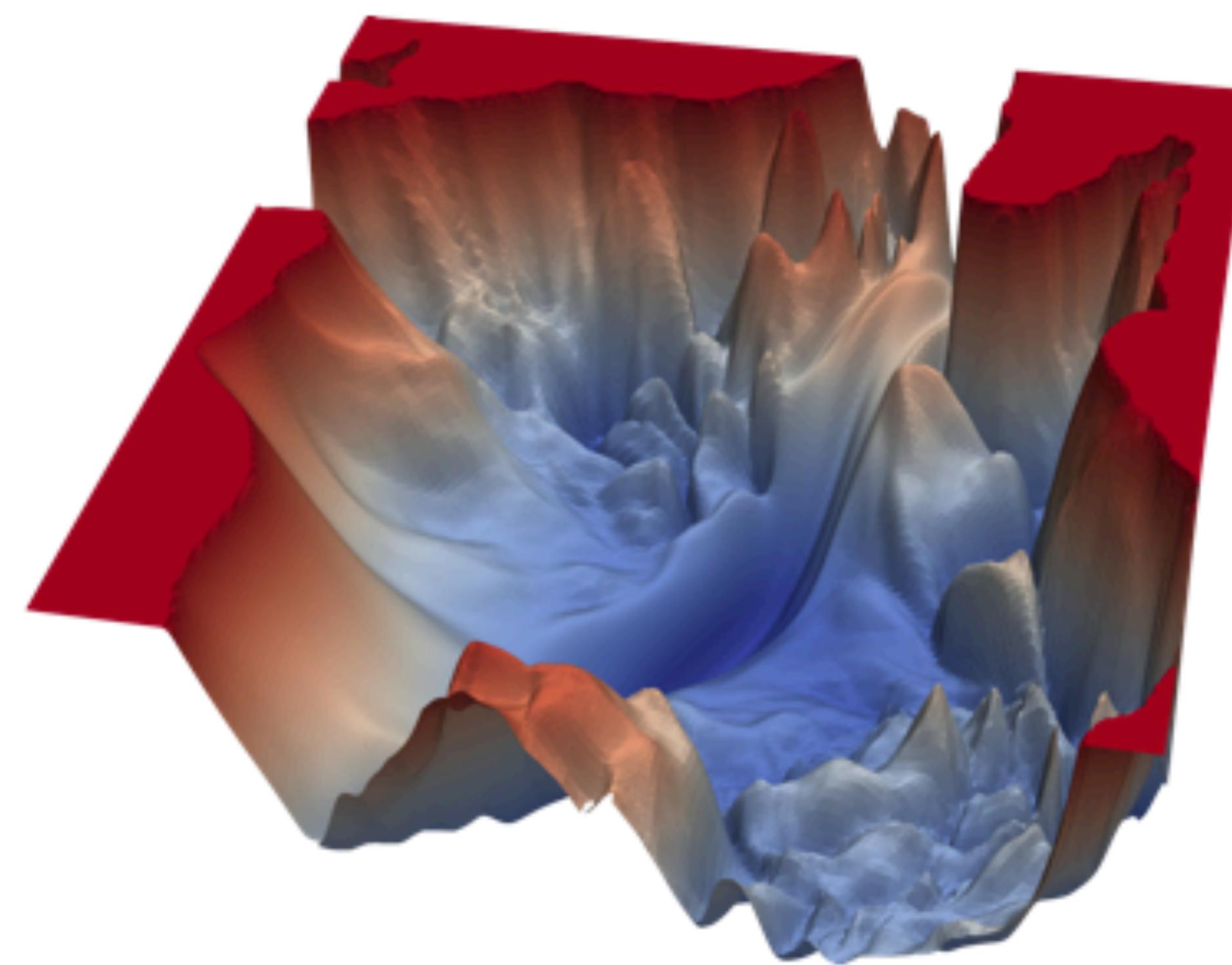
Learn the residual, instead of the full mapping

E.g., “use the same settings, but just change  $x$ ” is easier than starting from scratch.

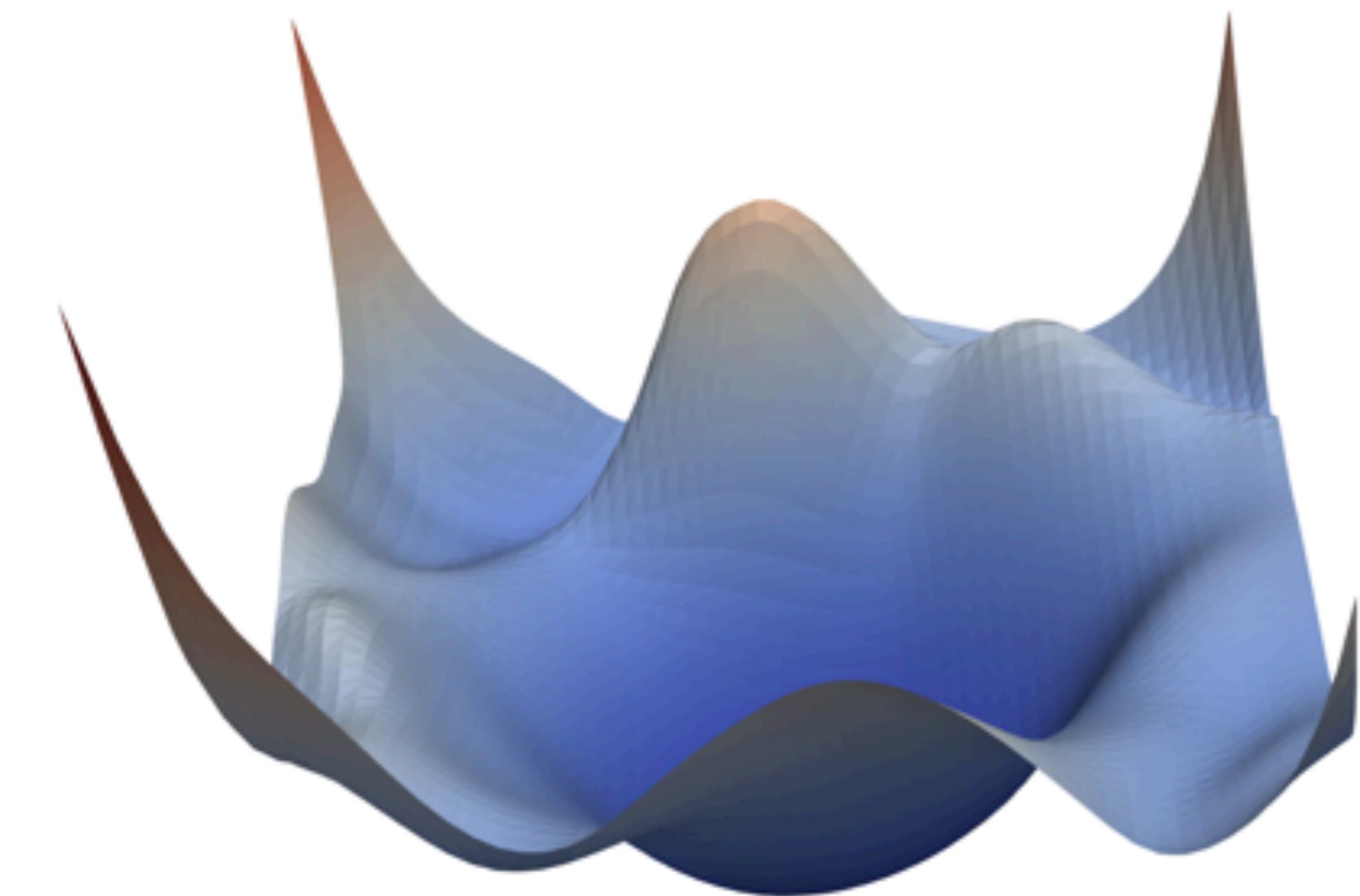
It is easier for the gradient to flow back



No residual connections



With residual connections



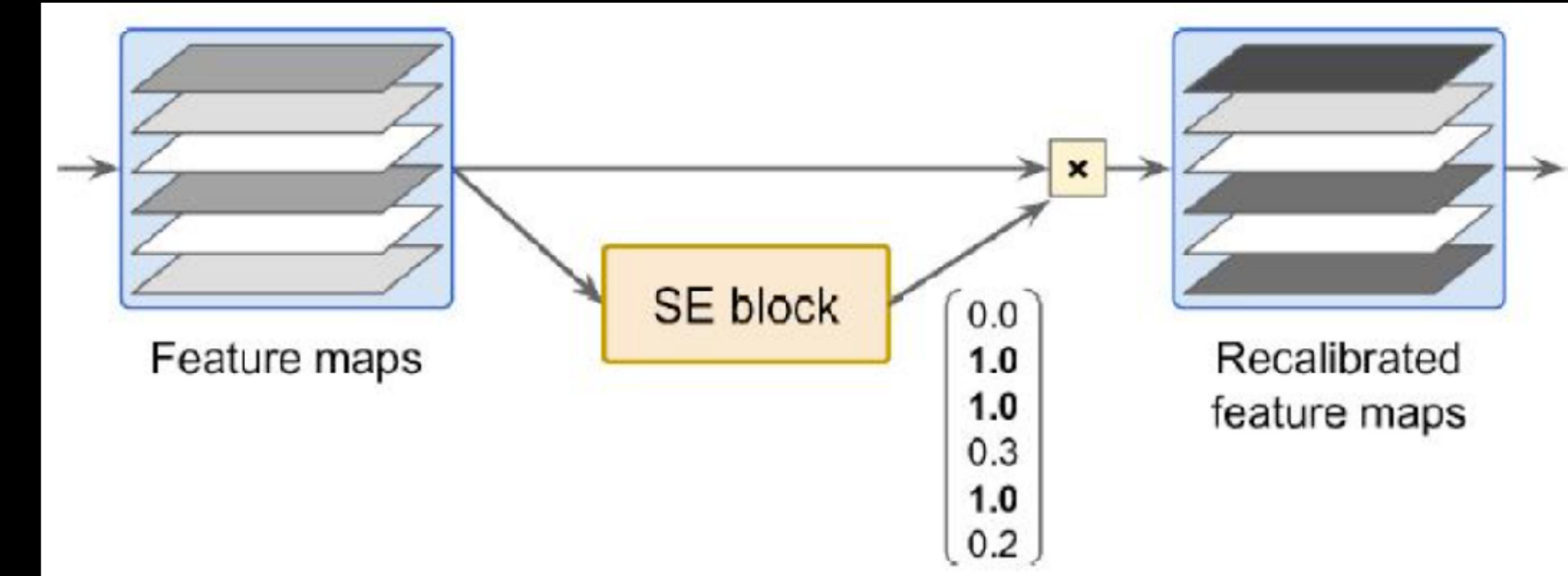
Same general network architecture

# ResNet

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112			7×7, 64, stride 2		
conv2_x	56×56	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix} \times 2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix} \times 3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix} \times 3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix} \times 3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix} \times 3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix} \times 2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix} \times 4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix} \times 4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix} \times 4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix} \times 8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix} \times 2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix} \times 6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix} \times 6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix} \times 23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix} \times 36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix} \times 2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix} \times 3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix} \times 3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix} \times 3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix} \times 3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		$1.8 \times 10^9$	$3.6 \times 10^9$	$3.8 \times 10^9$	$7.6 \times 10^9$	$11.3 \times 10^9$

2017

# SEnet



- Used **in combination with** other architectures
- Improves featuremaps: which features are most likely to be activated together?
  - E.g., if you see a left eye, you expect a right eye. Even if it is hidden in the image.

2017

# SEnet

- **global average pool**: calculates the average of each activation map
- **Squeeze**: reduce dimensionality, typically by a fraction of 16
- **Excite**: restore the amount of dimensions.

The process can be compared to writing an abstract of a text.

