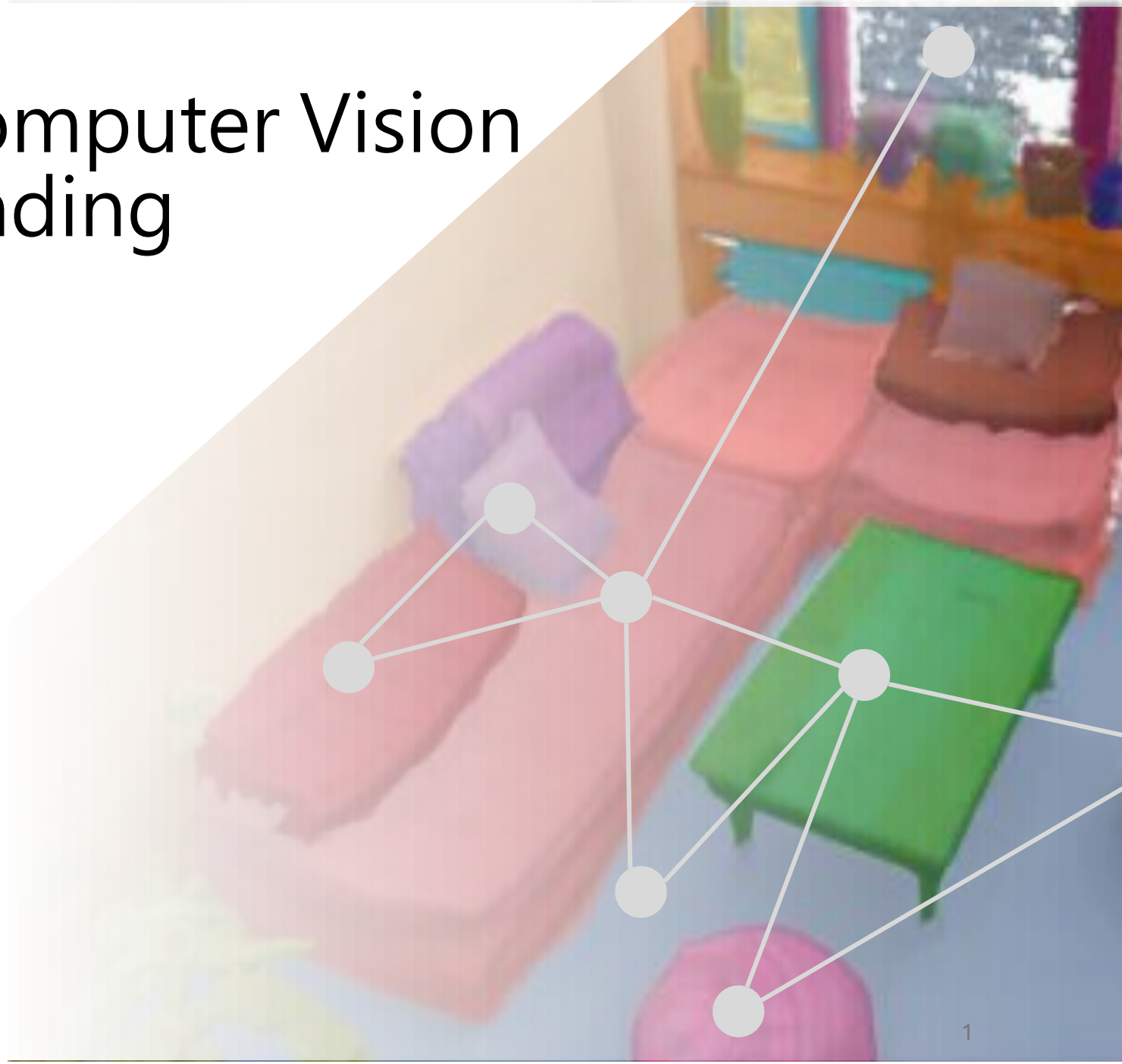
Deep Learning for Computer Vision and Scene Understanding



Dr. Helisa Dhamo
Research Scientist in 3D Vision
Huawei Noah's Ark Lab London

Lecture slides

Deep Learning for 2D Computer Vision

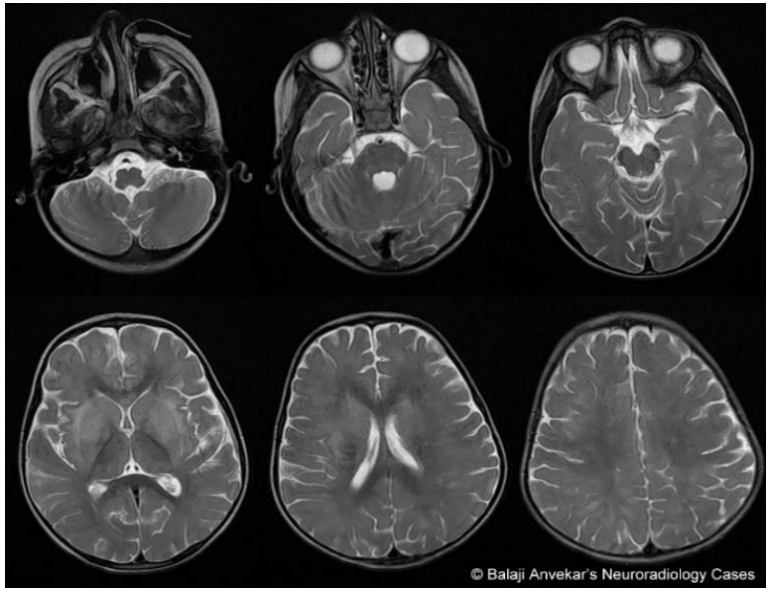
Lecture 2

Recap: Neural Networks

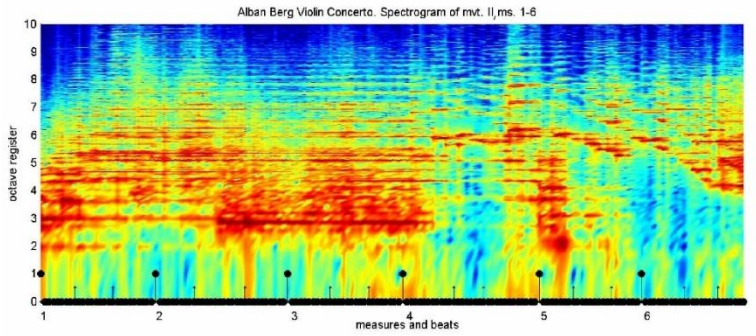
- Have a perceptron as the building block (layer)
- Composed of a number of such layers
- Use activations to introduce non-linearity
- Are trained using gradient descent and backpropagation

Motivation

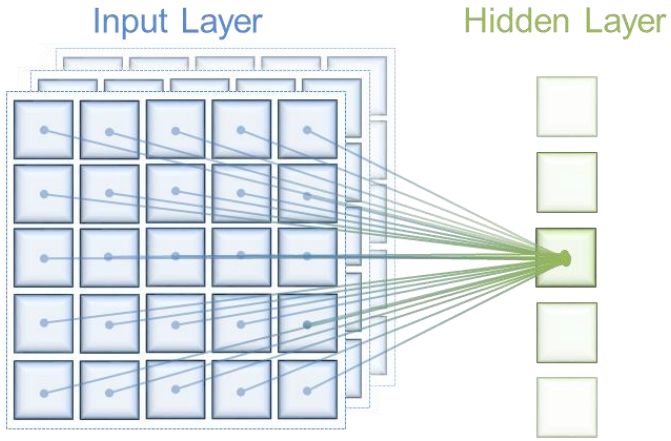
Data from the real world



Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident,



From ANNs to CNNs

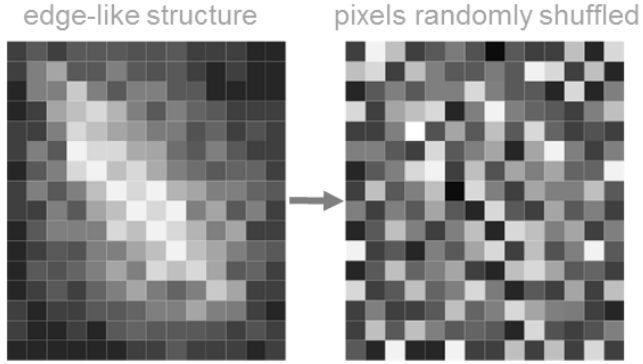


e.g. $5 \cdot 5 \cdot 3$ $(input\ size)$ $\cdot 5$ $(neurons)$

Some issues with ANNs:

- They don't scale well with high-dimensional inputs, increasing number of neurons, layers, etc.

- They do not take advantage of the input structure and ignore the local arrangement of neurons. For structured inputs (e.g. images), this is a disadvantage.



- Similar weight patterns would need to be learned for different regions, where similar features repeat. Need for large datasets that cover all variations.

Towards CNNs

Convolutional Neural Networks (CNNs) are a category of multi-layer NNs with learnable weights and biases, designed such that they tackle common problems of ANNs.

- **Assumption:**
  Inputs are structured, e.g. images

- **Key idea:**
  Invariance to shifts, scale and small distortions using
  - local weighted connections, i.e. local receptive field
  - shared weights across spatial locations
  - spatial sub-sampling

- **Main operations:**
  - Dot products with weights ($W^T x$) and addition of biases b
  - Non-linearities
  - $\max(\cdot)$ functions

CNN Architecture

Network architecture generally composed by:

- Neurons arranged in three dimensions: width, height and depth.
- Alternating convolutional (followed by a non-linear activation function) and sub-sampling layers to produce features at different levels of abstraction.
- Fully-connected layers that act as the final classifiers.

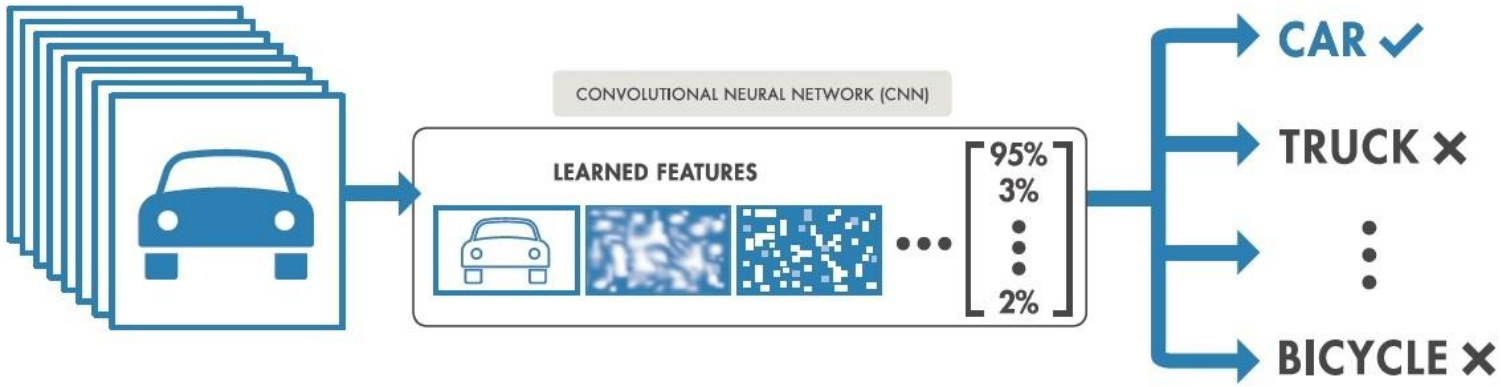


Image source: https://de.mathworks.com/discovery/convolutional-neural-network

CNN Architecture

Network architecture generally composed by:

- Neurons arranged in three dimensions: width, height and depth.
- Alternating convolutional (followed by a non-linear activation function) and sub-sampling layers to produce features at different levels of abstraction.
- Fully-connected layers that act as the final classifiers.

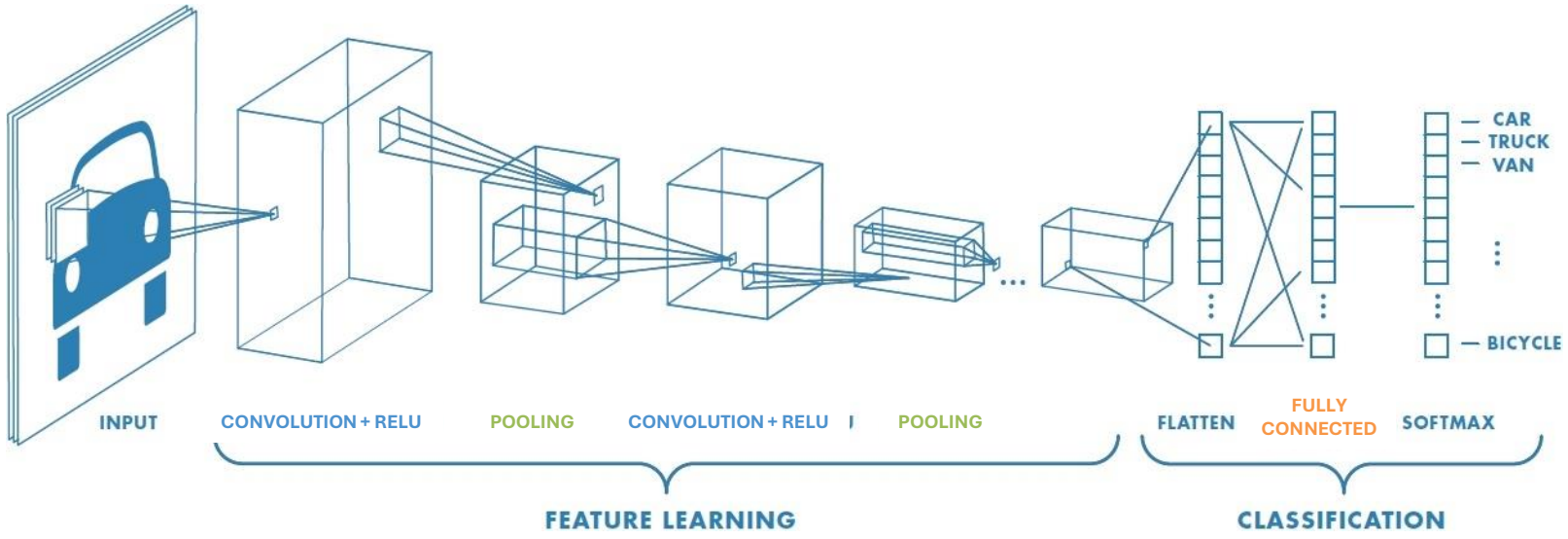
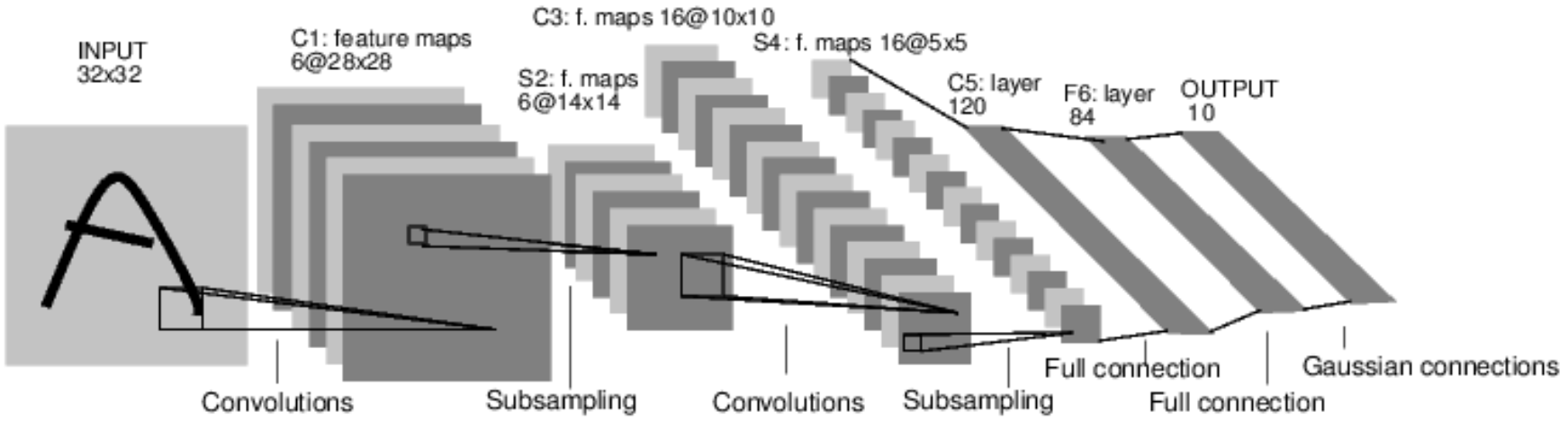


Image source: https://de.mathworks.com/discovery/convolutional-neural-network

LeNet – 5

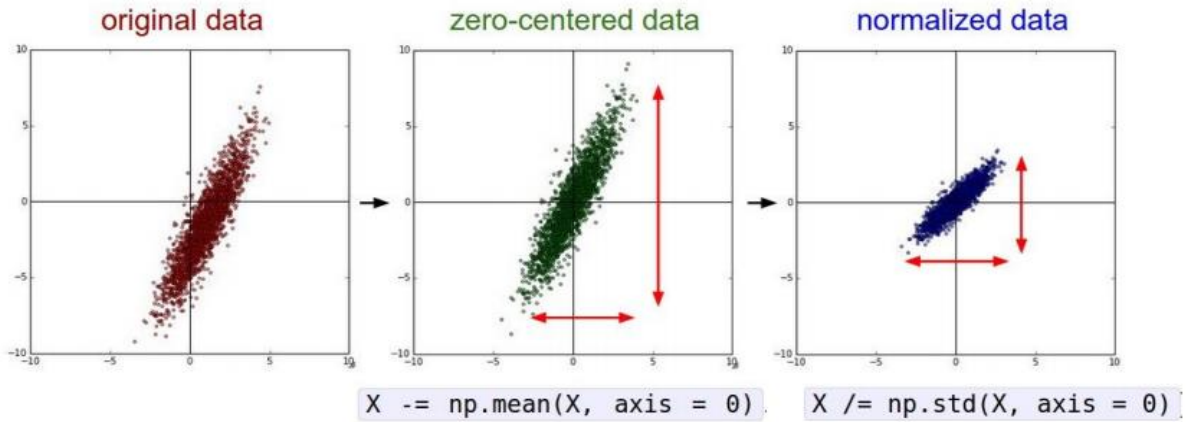
- First successful CNN architecture
- Introduced in 1998 for handwritten digit recognition
- Trained with back-propagation and gradient descent



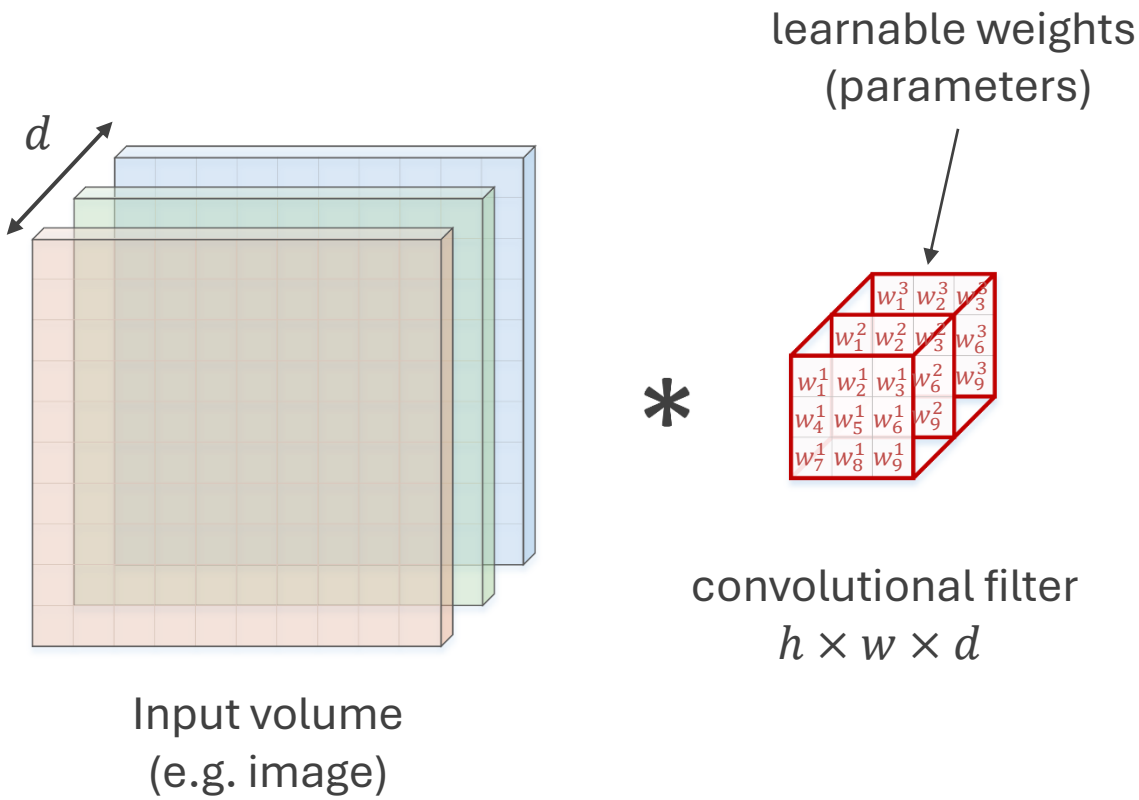
- LeCun et al. "Gradient-based learning applied to document recognition." *Proceedings of the IEEE* 86.11 (1998): 2278-2324.

Basic Layers Input Layer

- Data usually undergo some pre-processing before training
- PyTorch expects images in [batch_size, n_channels, height, width]
  - mean subtraction, unit standard deviation
  - augmentations (= small transformations, e.g. rotation, flipping, cropping, multiplicative color factor)

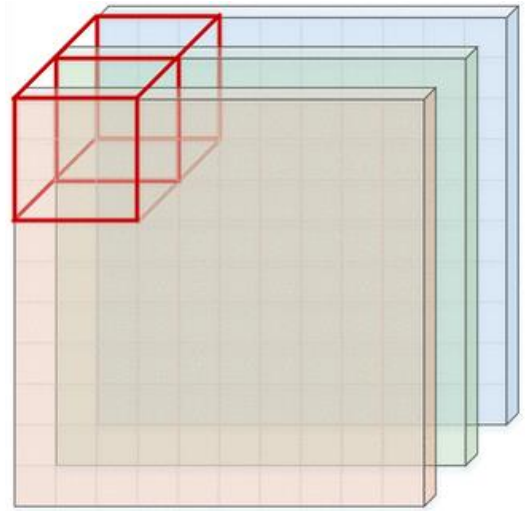


Basic Layers Convolutional Layer



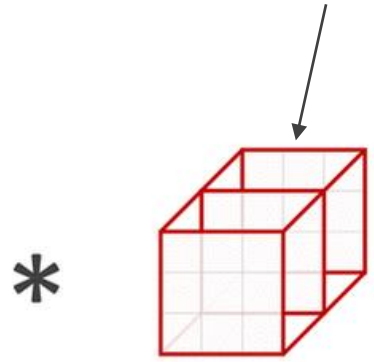
Basic Layers Convolutional Layer

slide filter
over input



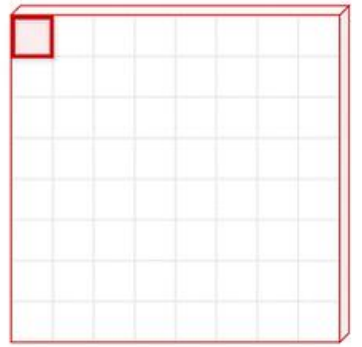
Input volume
(e.g. image)
$10 \times 10 \times 3$

learnable weights
(parameters)



convolutional filter
$h \times w \times d$

$3 \times 3 \times 3$

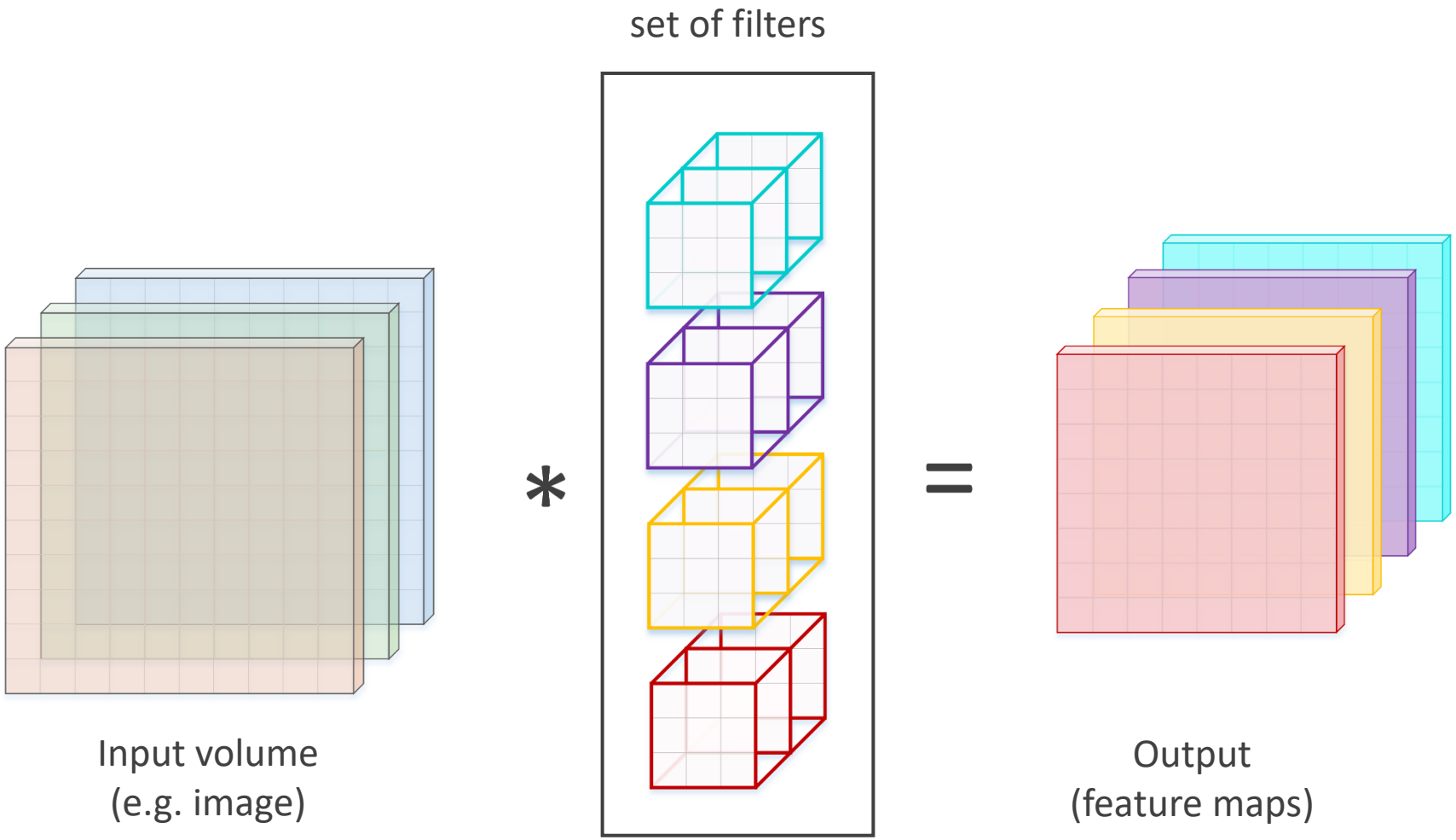


Output
(feature map)
$8 \times 8 \times 1$

The filter slides *spatially*, but operates (**dot product**) on *all* dimensions

Why output resolution is 8 x 8 and not 10 x 10 ?

Basic Layers Convolutional Layer



learning multiple (different) filters → produce several feature maps → multitude of features

Basic Layers Convolutional Layer

- Convolution in PyTorch

```
torch.nn.Conv2d(
    in_channels,
    out_channels,
    kernel_size,
    stride=1,
    padding=0,
    dilation=1,
    groups=1,
    bias=True,
    padding_mode='zeros‘
)
```

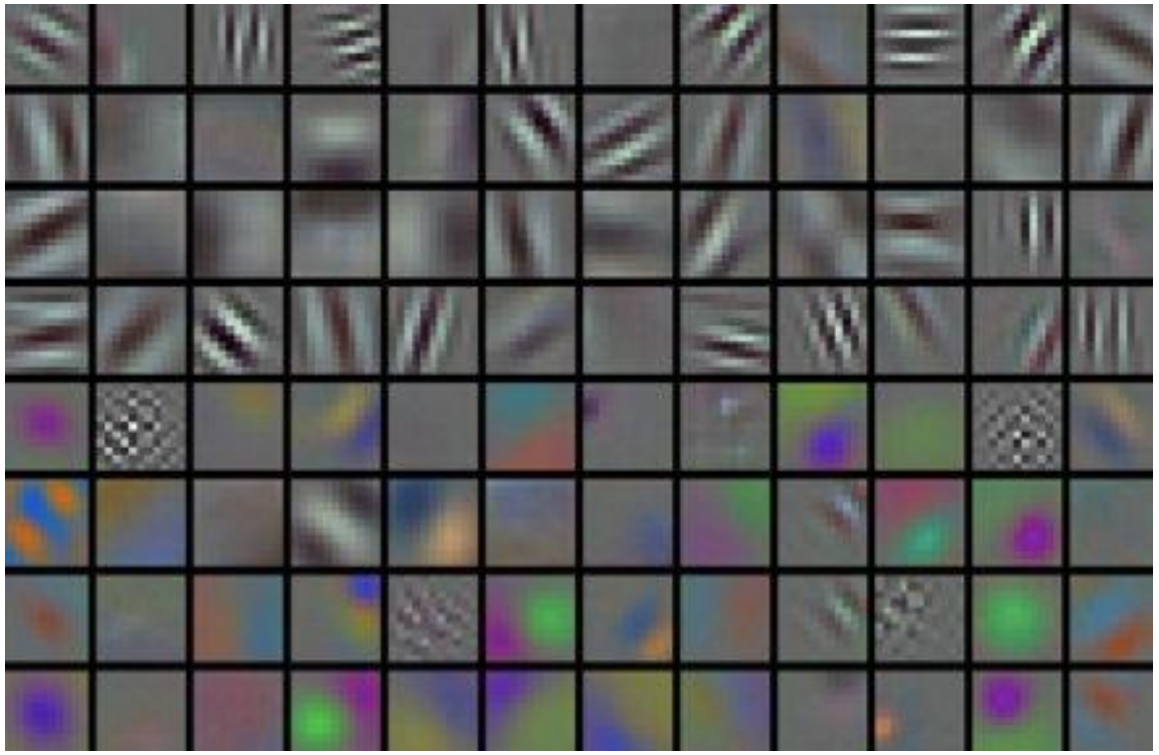
Common filter sizes are 3×3, 5×5, etc.
1×1 is also possible because it operates in depth too.

Single or tuple kernel_size, stride, padding, dilation (single makes it the same for H and W)

Stride n: take every n-th pixel, skip the rest

Basic Layers Convolutional Layer

First-layer filters from **AlexNet** (visualization of 96 $[11 \times 11 \times 3]$ filters):



First-layer learned features include basic elements, such as edges, blobs, colors, etc.

Parameter sharing thus appears to be reasonable: detecting e.g. an edge is important at any position of the input image.

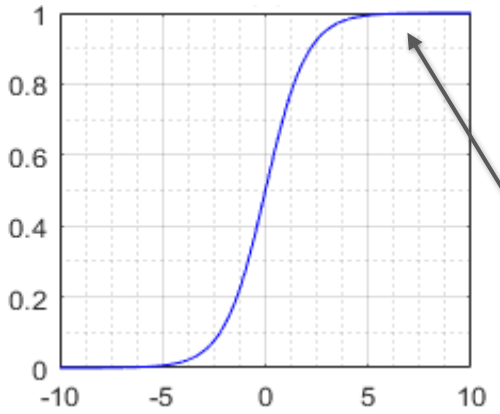
- Krizhevsky et al. "Imagenet classification with deep convolutional neural networks." *NIPS* 2012.

Basic Layers Activation function

Convolutions are typically followed by **non-linear** activation functions, that act per neuron.

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

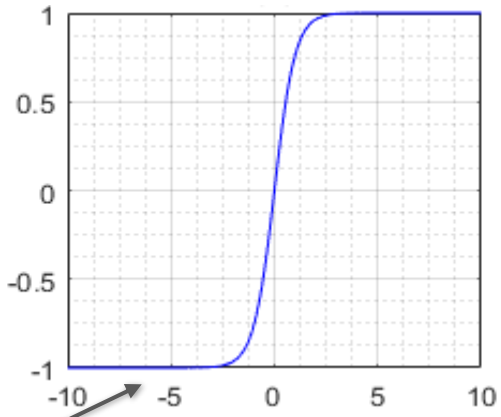
`torch.nn.Sigmoid()`



Output range: [0,1]

Tanh: $\tanh(x) = 2\sigma(2x) - 1$

`torch.nn.Tanh()`



Output range: [-1,1]

However, sigmoid and tanh functions saturate, making gradients very small while flowing backwards through the network → weight updates get killed.

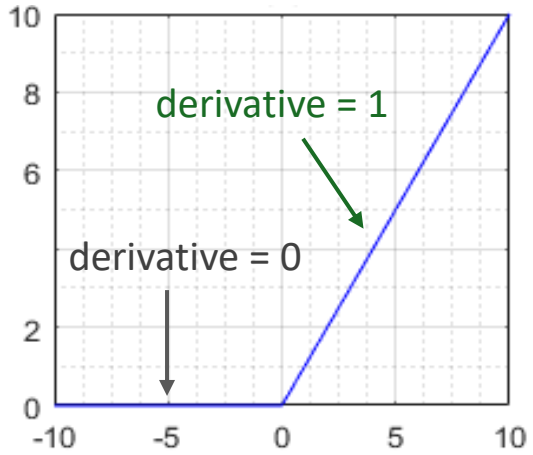
Basic Layers Activation function

Convolutions are typically followed by **non-linear** activation functions, that act per neuron.

Rectified Linear Unit (ReLU)

$$f(x) = \max(0, x)$$

`torch.nn.ReLU(inplace=False)`



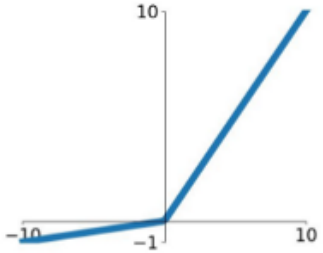
- Was introduced to CNNs in the past few years (2012)
- Simply thresholds at zero
- Sparse activation
- Computationally efficient
- Non-saturating → speeds up convergence

Basic Layers Activation function

"Dying ReLU" : neurons might be driven into an irreversible state due to specific weight updates (e.g. high gradient flow) and thus stay inactive for all inputs.

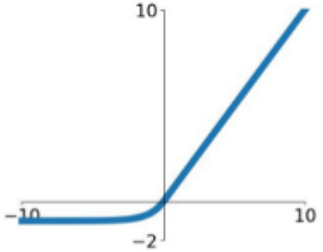
Other variations of ReLU:

- "Leaky" ReLU $f(x) = \begin{cases} x, & if\ x > 0 \\ ax, & otherwise \end{cases}$




```
torch.nn.LeakyReLU(negative_slope=0.01, inplace=False)
```

- Exponential (ELU) $f(x) = \begin{cases} x, & if\ x \geq 0 \\ a(e^x - 1), & otherwise \end{cases}$



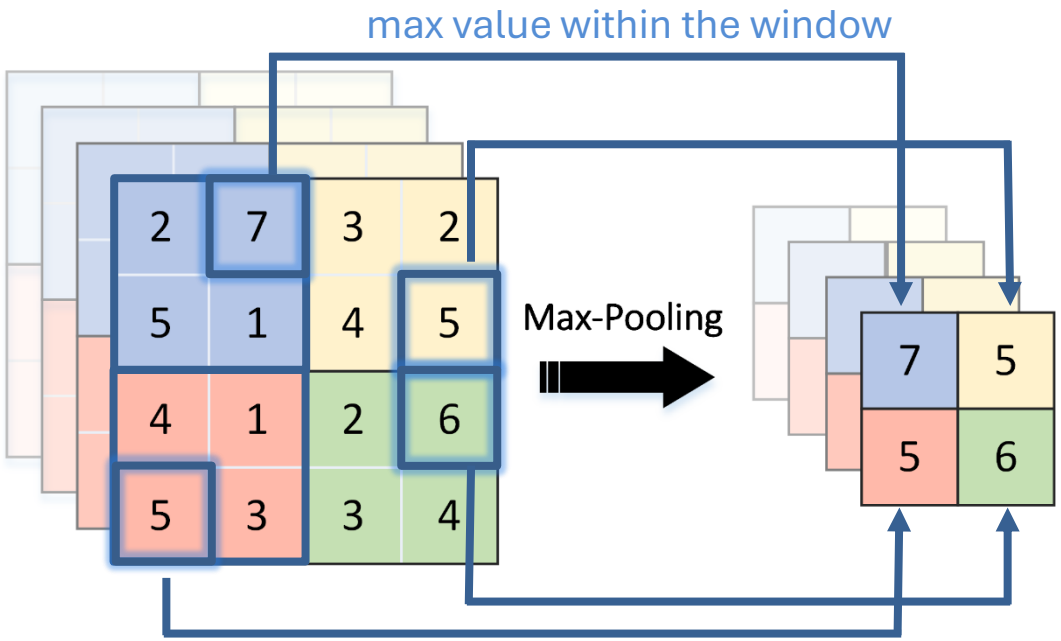

```
torch.nn.Elu(alpha=1.0, inplace=False)
```

He et al. "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification." *ICCV* 2015.
Clevert et al. "Fast and accurate deep network learning by exponential linear units (elus)." *arXiv:1511.07289* (2015).

Basic Layers

Pooling Layer

Performs an element-wise operation on the feature maps, on each channel independently. **Usually $\max(\cdot)$ or $\text{avg}(\cdot)$.**



Example: Max-pooling

The window size is 2×2, applied with a stride of 2 (common case)

Basic Layers

Pooling Layer

- Parameter-free layer
- Typically used for feature map spatial sub-sampling (with stride > 1).
- Controls the capacity of the network by reducing the resolution
- Introduces some invariance to small transformations of the input, because precise spatial information is lost

- **Hyperparameters:**
  - width w and height h of window
  - stride s
    overlapping of sliding window occurs if $s < w$ *or* $s < h$

Size of resulting pooled maps:

$$h_{out} = \frac{h_{in} - h}{s} + 1$$

$$w_{out} = \frac{w_{in} - w}{s} + 1$$

Basic Layers

Pooling Layer

- (Max)Pooling in PyTorch

```
torch.nn.MaxPool2d(
    kernel_size,
    stride=None,
    padding=0,
    dilation=1,
    return_indices=False,
    ceil_mode=False
)
```

There exist more pooling layers:

`torch.nn.AvgPool2d()`,

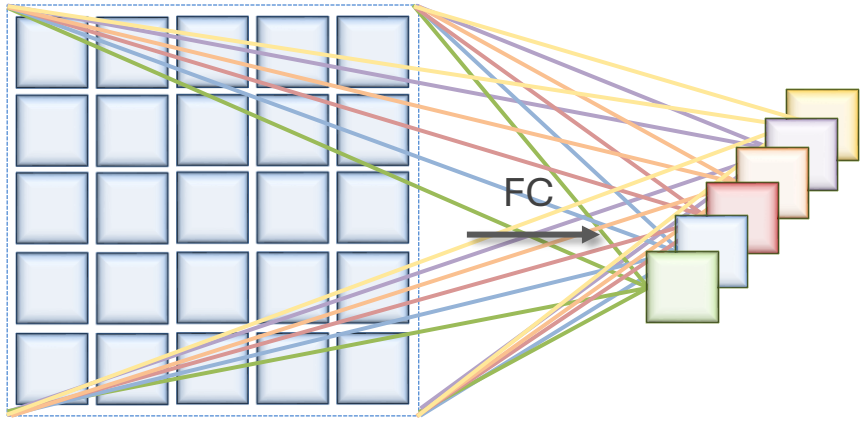
`torch.nn.MaxPool3d()`,

`torch.nn.AdaptiveAvgPool1d()`
(specify the output size – kernel size and stride are adapted accordingly),

…

Basic Layers Fully-connected Layer

- Fully-connected layers follow the principle of the typical ANN weighted connections: each neuron in the output connects to all neurons of the input
- Usually added as the last layers of the network / output layer
- Guarantee a full receptive field



Input $h \times w \times d$
(here: $5 \times 5 \times 1$)

Output $1 \times 1 \times n$
n being the only hyperparameter
(here: $n = 6$)

Can be also seen as a convolutional layer with a set of filters of the **same size** as the input volume, i.e. n filters of size $h \times w \times d$

Basic Layers Fully-connected Layer

- To construct a fully-connected layer in PyTorch:

```
torch.nn.Linear(
    in_features,
    out_features,
    bias=True
)
```

```
Input shape: [batch_size, in_features]
Output shape: [batch_size, out_features]
Weight shape: [in_features, out_features]
```

- Other possible options

Using `torch.nn.Conv2d` with the size of `filters` matching that of the `inputs, and no padding`

PyTorch Define a Neural Network Model

- ➔ Extend the `Module` class of `torch.nn`
- ➔ Implement the constructor and the forward member function

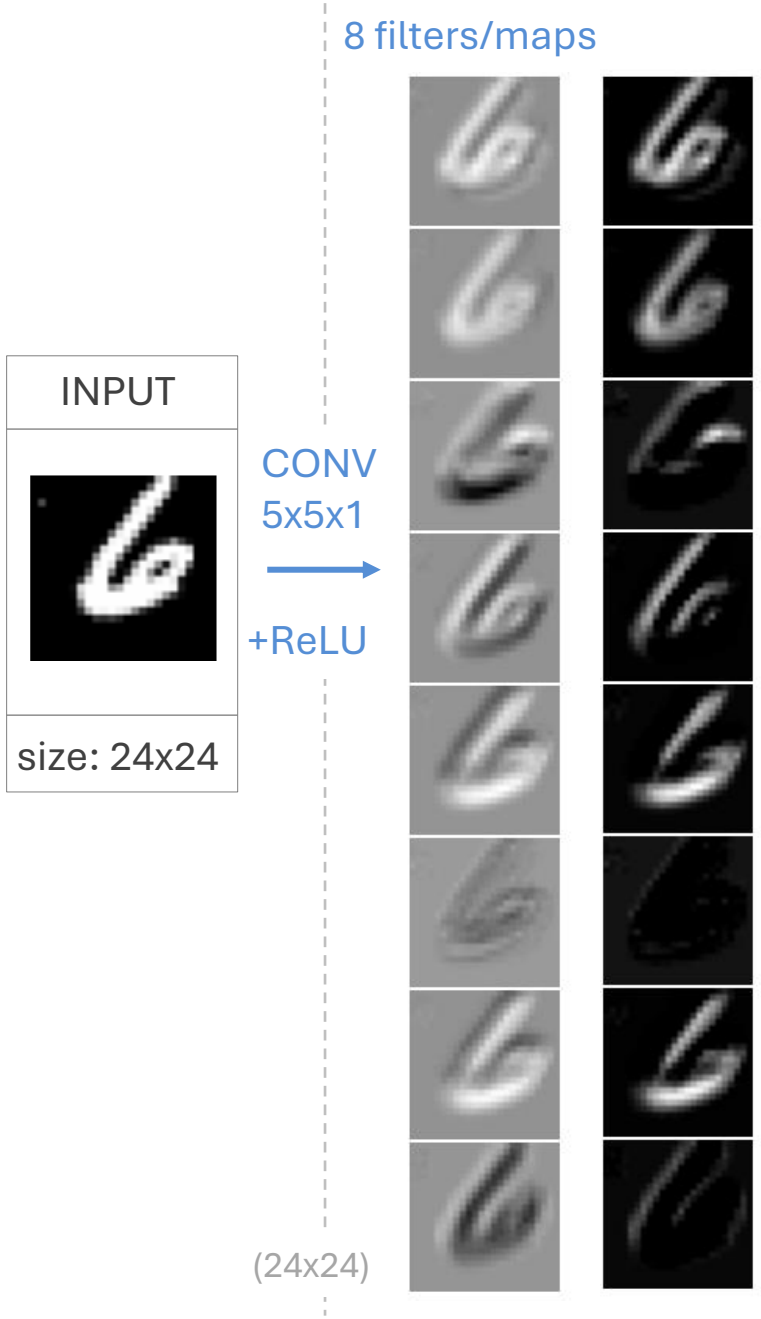
```
import torch.nn as nn

class ConvNet(nn.Module):
    def __init__(self):
        super(ConvNet, self).__init__()

        self.pool = nn.AvgPool2d(2, 2)
        self.conv = nn.Conv2d(3, 16, 3)
        self.fc = nn.Linear(16 * 5 * 5, 60)
        self.relu = nn.Relu()

    def forward(self, x):
        x = self.pool(self.relu(self.conv(x)))
        x = x.view(-1, 16 * 5 * 5)
        x = self.fc(x)
        return x
```

PyTorch Define a Neural Network Model

- ➔ Extend the `Module` class of `torch.nn`
- ➔ Implement the constructor and the forward member function
  - ➔ Alternatively, use `torch.nn.Sequential`

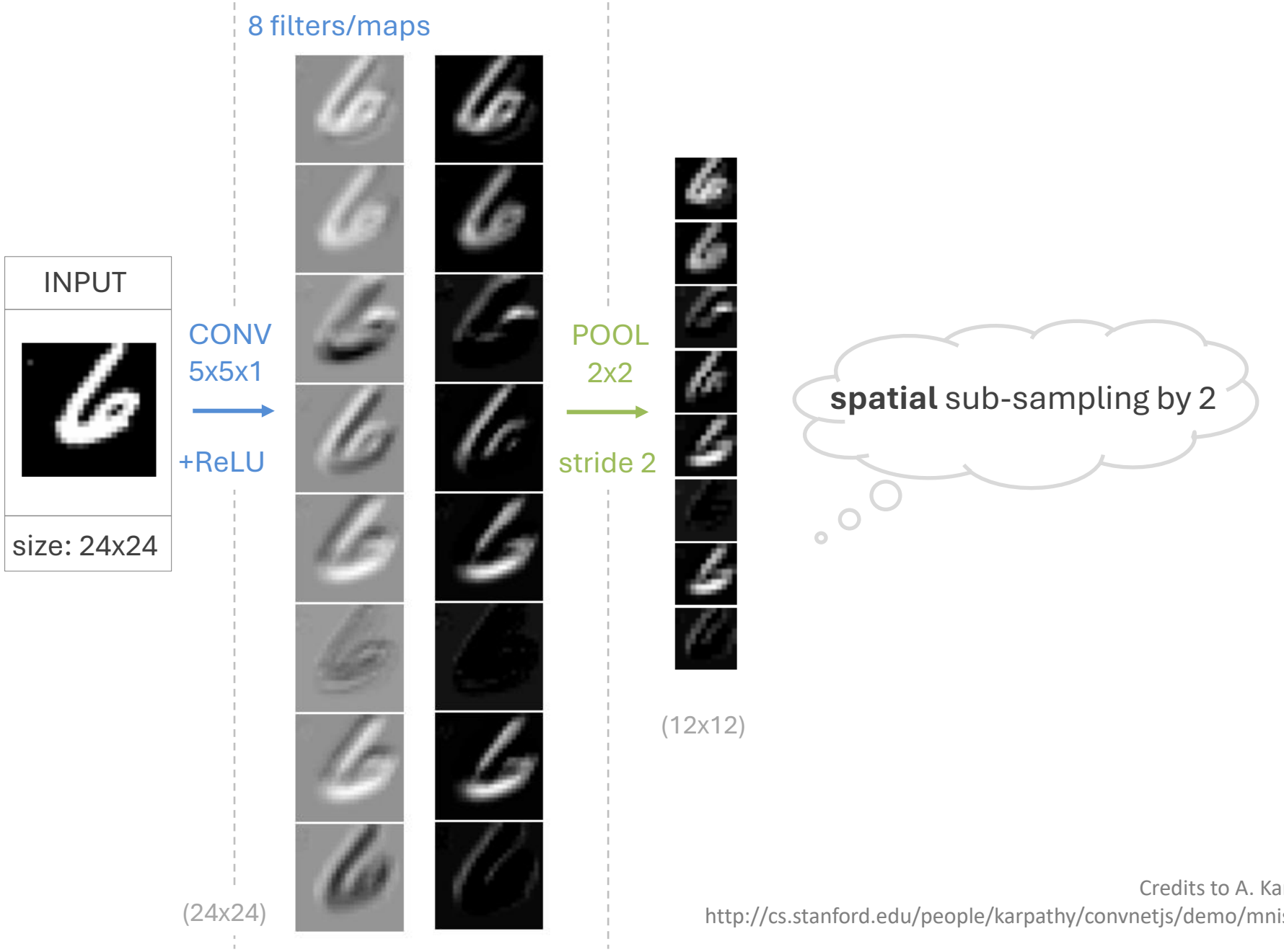
```
import torch.nn as nn

class ConvNet(nn.Module):
    def __init__(self):
        super(ConvNet, self).__init__()

        self.backbone = nn.Sequential(nn.Conv2d(3, 16, 3),
                                      nn.Relu(),
                                      nn.AvgPool2d(2, 2))

        self.fc = nn.Linear(16 * 5 * 5, 60)

    def forward(self, x):
        x = self.backbone(x)
        x = x.view(-1, 16 * 5 * 5)
        x = self.fc(x)
        return x
```

No need to have a name for each layer

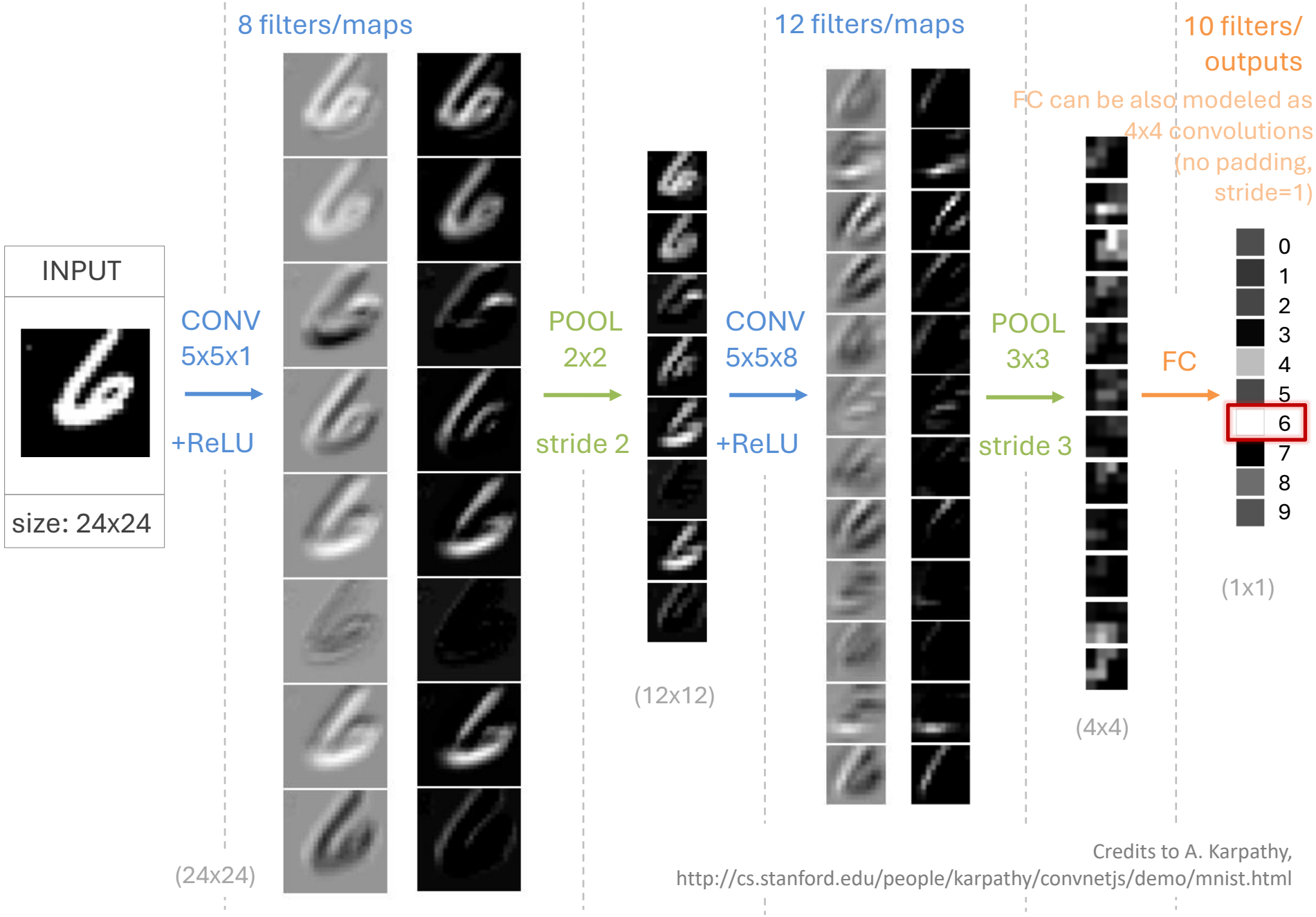


Notice how feature maps activate on different basic structures, depending on the corresponding filter

Credits to A. Karpathy,
http://cs.stanford.edu/people/karpathy/convnetjs/demo/mnist.html



Credits to A. Karpathy,
http://cs.stanford.edu/people/karpathy/convnetjs/demo/mnist.html

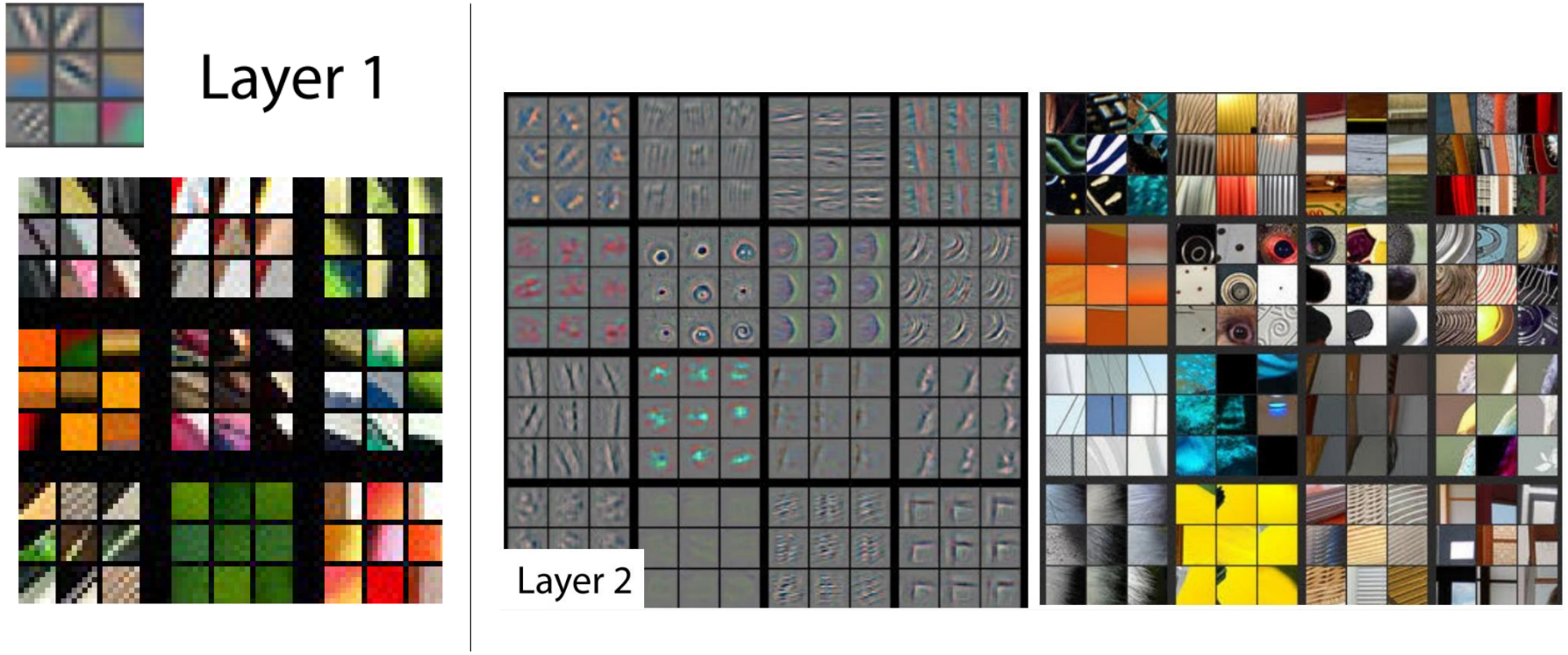


Credits to A. Karpathy,
http://cs.stanford.edu/people/karpathy/convnetjs/demo/mnist.html

Layer Visualization

Patterns which excite neurons the most become more complex

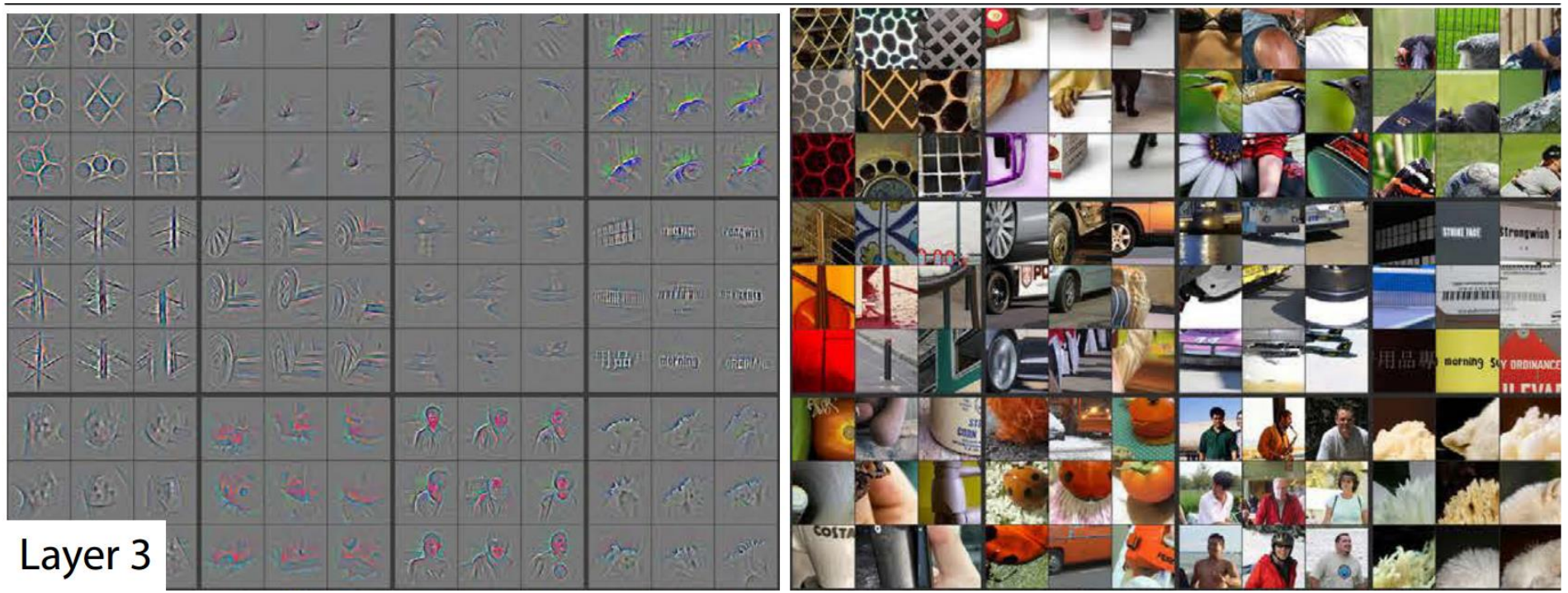
as we go deeper (features are extracted in a hierarchical manner)



Zeiler, M. D., & Fergus, R. (2014, September). Visualizing and understanding convolutional networks. In *European conference on computer vision* (pp. 818-833). Springer International Publishing.

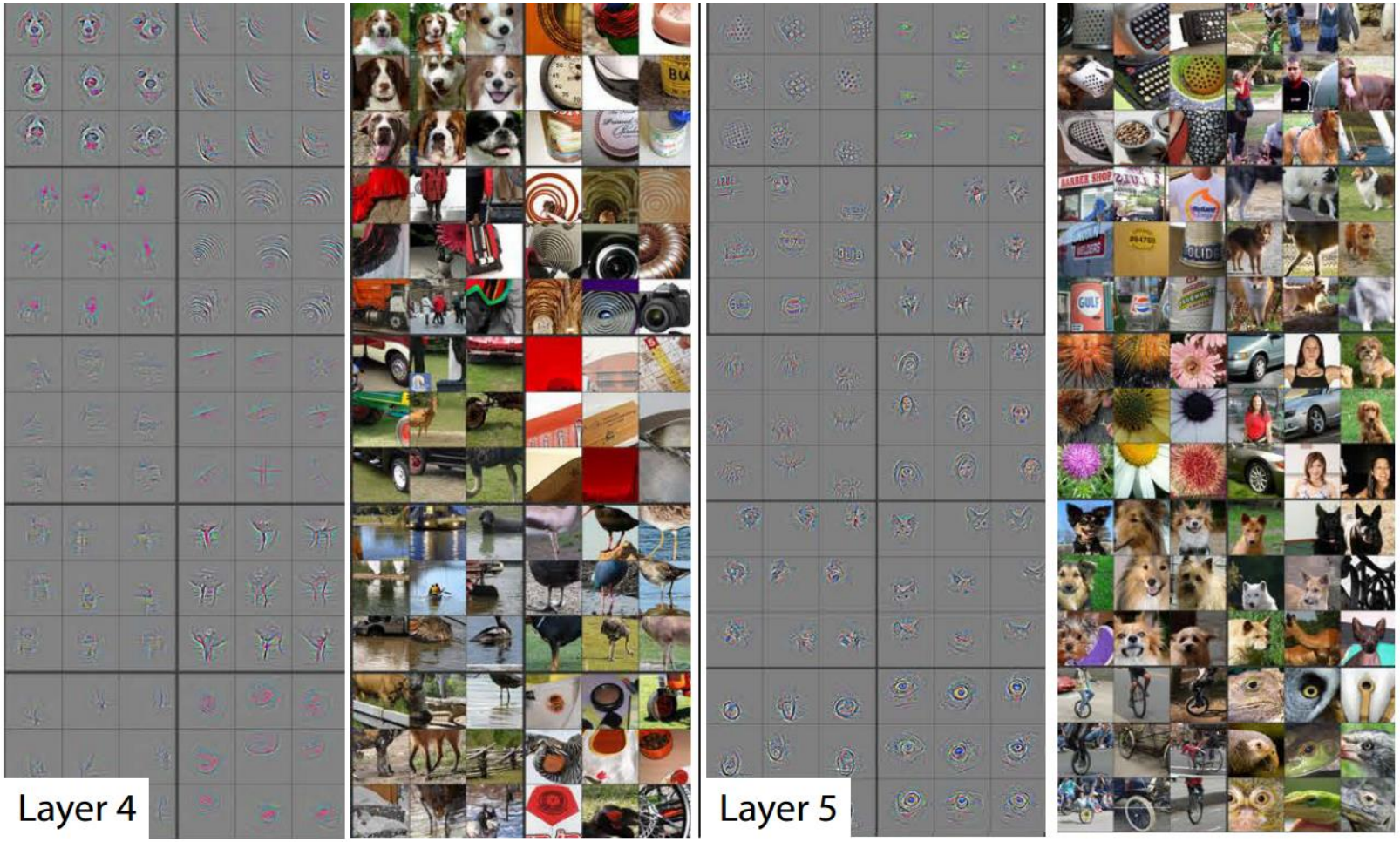
Layer Visualization

The patterns also increase in spatial size, because of the increasing receptive field of the network, i.e. the area in an input image that a given neuron “sees” at each layer.



Layer Visualization

The deeper we go → the more powerful the representations we learn



Weight Initialization

- Weights must be initialized before we start training. PyTorch layers have default weight initializations, e.g. sample from uniform distribution, stddev ~ kernel_size
- To have an initialization different from default:

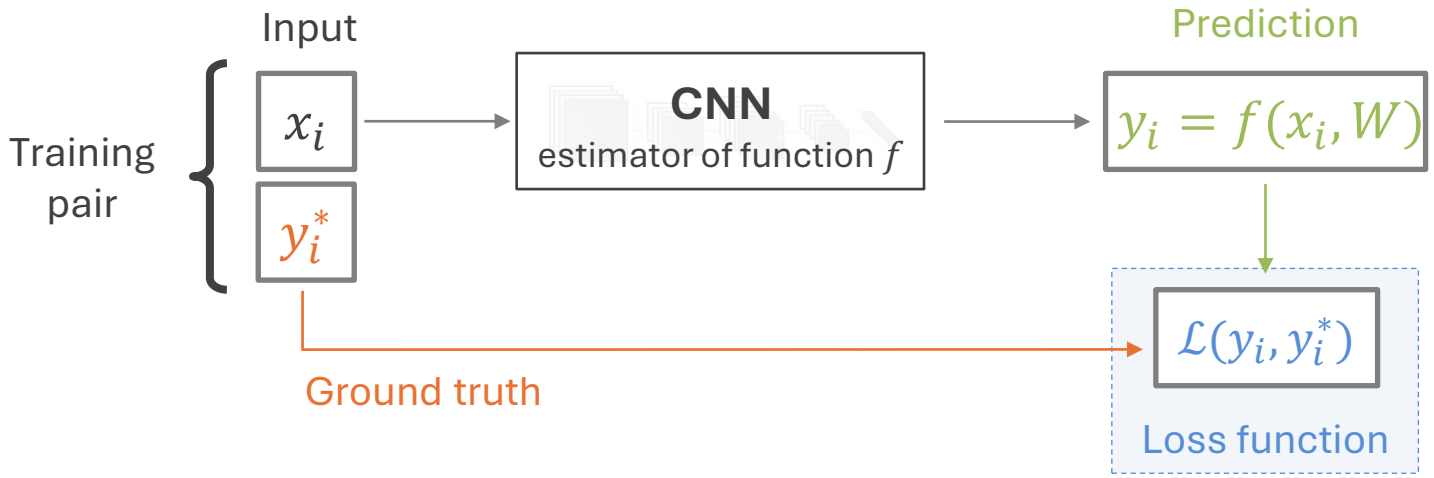
```
def weights_init(m):
    if isinstance(m, nn.Conv2d): # init all conv weights with xavier
        xavier(m.weight.data)
        xavier(m.bias.data)

model.apply(weights_init)
```

- Give each neuron a unique weight, resulting in a distinct update. Randomly draw weights from a Gaussian distribution (multi-dimensional) and one of the following options:
  - zero mean and constant standard deviation (e.g. 0.01)
  - Xavier initialization (keep the scale of gradients approximately the same in all layers by scaling with the right deviation $\sqrt{\frac{2}{n_{in} + n_{out}}}$
- Biases can be initialized with zeros (symmetry broken)

Glorot and Bengio, “Understanding the difficulty of training deep feedforward neural networks”. International conference on artificial intelligence and statistics (2010)

Loss function

- The weights of a CNN can be learned on different problems (classification or regression ones) by minimizing a specified objective, i.e. loss function.
- It simply measures how well the CNN performs on the task.
- To do this, a “loss layer” receives the output of the CNN (prediction) and compares it to the ground truth of the given input.
- The loss over the entire dataset must be written as the mean of the individual losses of the samples.



Loss function

- The weights of a CNN can be learned on different problems (classification or regression ones) by minimizing a specified objective, i.e. loss function.
- It simply measures how well the CNN performs on the task.
- To do this, a “loss layer” receives the output of the CNN (prediction) and compares it to the ground truth of the given input.
- The loss over the entire dataset must be written as the mean of the individual losses of the samples.

- **Example: Image classification**
  - the ground truth is the labeled category for the image
  - the prediction is a vector of scores, referred to as `logits`, which represent the confidences that the input belongs to each of the existing categories.

Loss function

Case: Classification

Softmax operation `torch.nn.Softmax()` or `torch.nn.softmax(logits)`

(from logistic outputs y_i to probabilities $\hat{y}_i$)

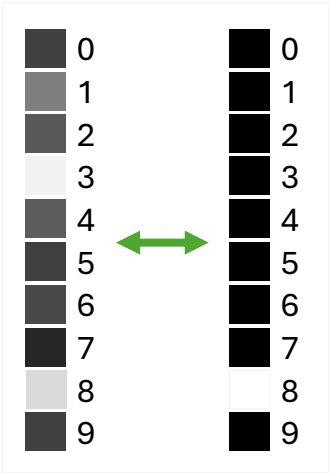
$$\hat{y}_i(x) = \frac{\exp(y_i)}{\sum_{k=1}^{C} \exp(y_k)}, \text{ for the } i^{th} \text{ class } (i = 1, 2, \dots, C)$$

Cross entropy (Minimize the negative log likelihood per sample)
(generalization of logistic regression to multiple mutually exclusive classes)

$$\mathcal{L}(\boldsymbol{y}^*, \hat{\boldsymbol{y}}) = -\sum_{i=1}^{C} y_i^* \cdot \log(\hat{y}_i)$$

One-hot class labels
{0,1}

normalized probabilities
(i.e. after softmax)



Loss function

Case: Classification

```
torch.nn.NLLLoss(weight=None, size_average=None, ignore_index=100,
reduce=None, reduction='mean')
```

- Expects the log softmax of the logits as input

```
torch.nn.CrossEntropyLoss(weight=None, size_average=None,
ignore_index=-100, reduce=None, reduction='mean')
```

- Combines `nn.LogSoftmax()` and `nn.NLLLoss()` in the same class:
- Therefore, you should pass the logits directly

- Check documentation carefully before using (does it assume softmax, logsoftmax, logits)?

Loss function

Case: Regression

Typically used losses: $\mathcal{L}_1$ or $\mathcal{L}_2$ norm of the difference between prediction $\boldsymbol{y}$ and ground truth $\boldsymbol{y}^*$ per sample.

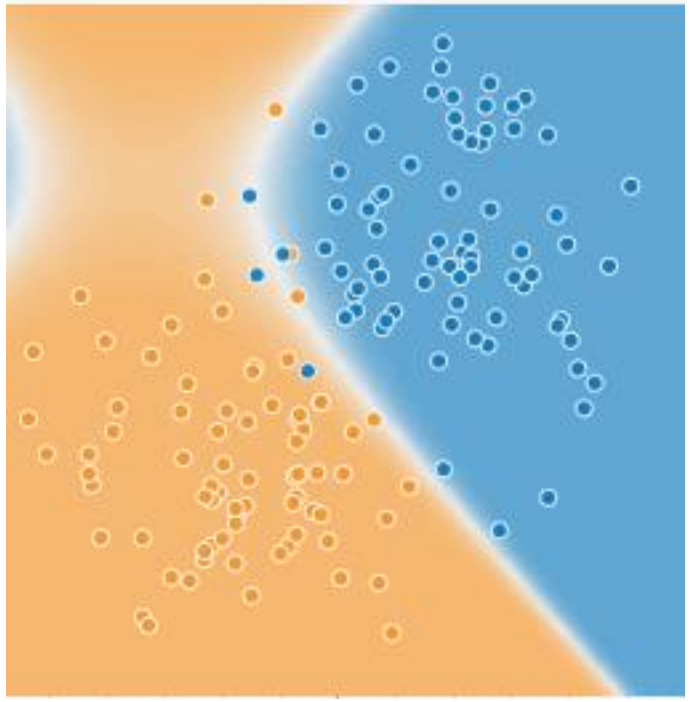
$$\mathcal{L}_1(\boldsymbol{y}, \boldsymbol{y}^*) = |\boldsymbol{y} - \boldsymbol{y}^*|_1 = \frac{1}{n}\sum_{i=1}^{n} |y_i - y_i^*|$$

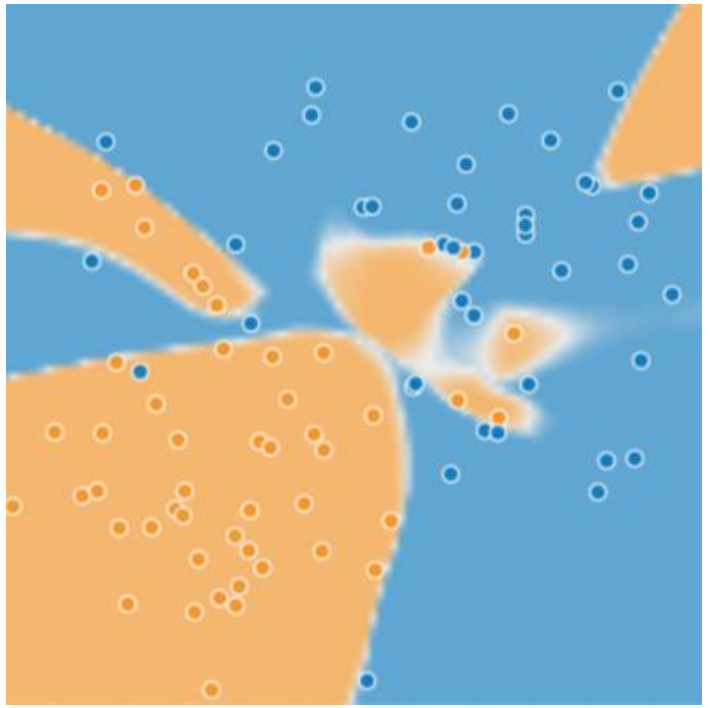
$$\mathcal{L}_2(\boldsymbol{y}, \boldsymbol{y}^*) = \|\boldsymbol{y} - \boldsymbol{y}^*\|_2 = \frac{1}{n}\sum_{i=1}^{n} (y_i - y_i^*)^2$$

Note that $\boldsymbol{y}, \boldsymbol{y}^*$ can have arbitrary dimensions depending on the task, e.g. vectors of regressed points or entire prediction maps.
n would then be the number of points or pixels respectively

```
torch.nn.L1Loss(size_average=None, reduce=None, reduction='mean')
torch.nn.MSELoss(size_average=None, reduce=None, reduction='mean')
```

Regularization

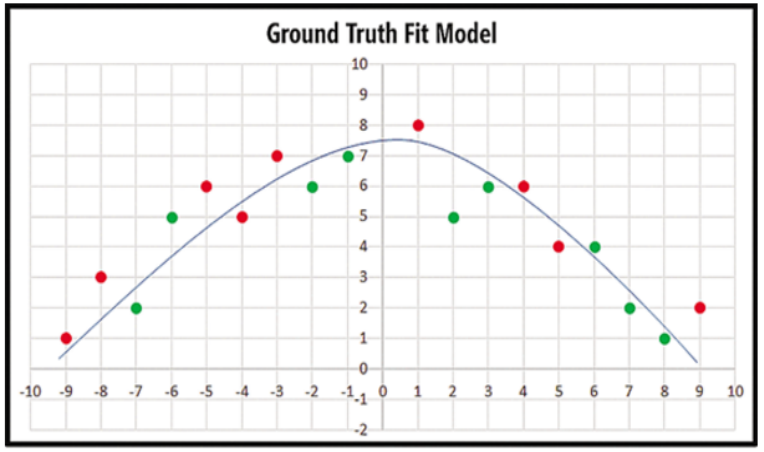


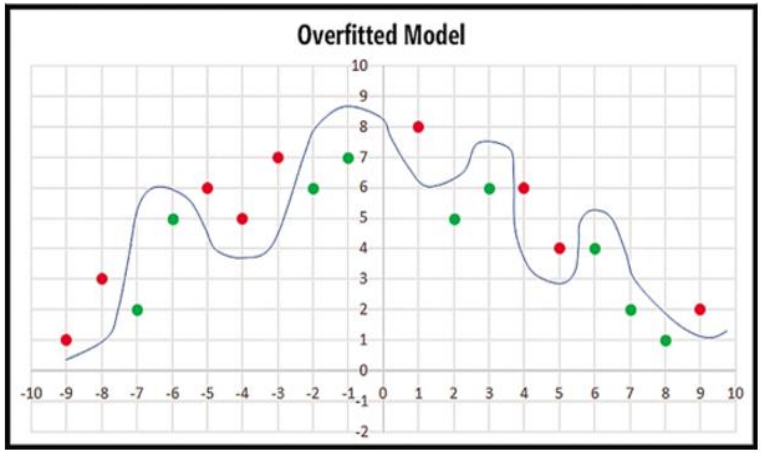


The **loss function** measures how well the model fits the data.
Regularization measures the model complexity.

Regularization

It includes methods for better generalization to unseen data, i.e. preventing over-fitting to the training samples.





In an over-fitted model, the predicted curve is not “regular”
Weights have very large or very small values

- Weight regularization
- Dropout
- Batch normalization

Regularization

It includes methods for better generalization to unseen data, i.e. preventing over-fitting to the training samples.

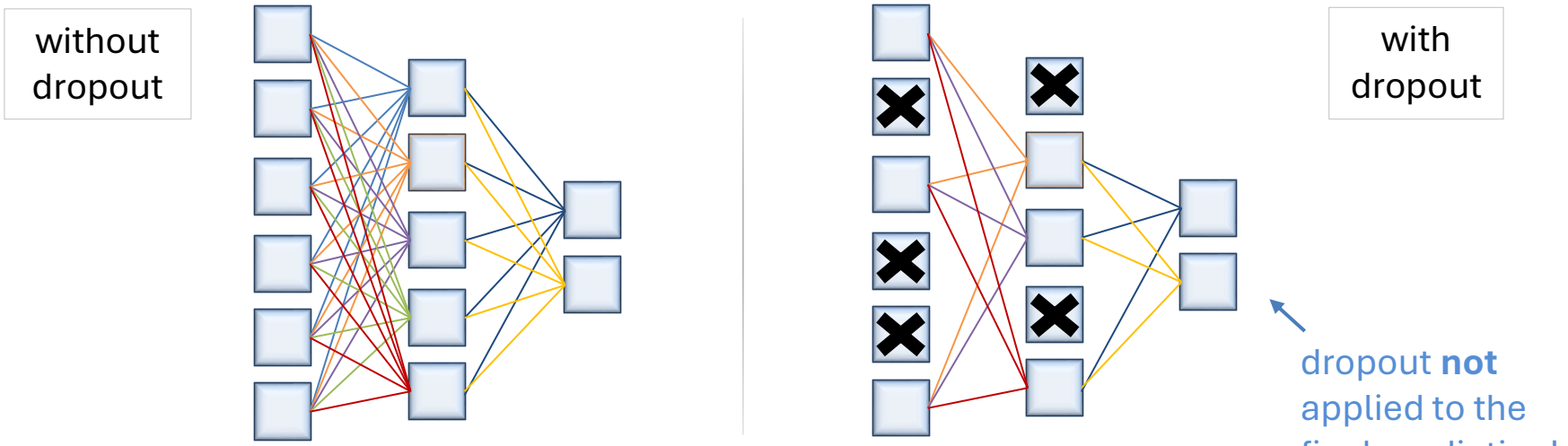
- L2 regularization
  - Add a penalty term $\frac{1}{2}aw^2$ to each weight
    a is the regularization strength (typically small, e.g. order of 10^{-4})
  - Favors weight "diffusion"
  - Weight update through gradient descent: $w_{t+1} = w_t - aw_t$ (linear decay)
  - In practice: add another term (regularizer) to the loss

$$\mathcal{L} = \mathcal{L} + a\sum_{i=1}^{K} w_i^2$$

ALL learnable parameters in the network!

Regularization

- Dropout `torch.nn.Dropout(p=0.5, inplace=False)`
  - Randomly “dropping out” neurons of a layer (with probability p, usually 0.5) at each iteration of training
  - This effectively means making them inactive (setting to zero) so that they do not contribute in forward/backward passes
  - Neurons do not learn to rely on the presence of other specific neurons
  - Usually applied before the last fully-connected layer(s)

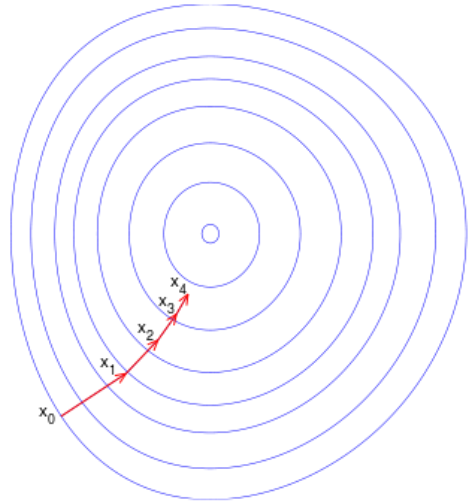


Srivastava et al. "Dropout: a simple way to prevent neural networks from overfitting." *Journal of Machine Learning Research* 15.1 (2014)

Optimization Methods

For a training iteration t and the current state of parameters denoted as w_t, an update is performed as:

$$w_{t+1} = w_t + \Delta w_t$$



- There exist a variety of **first-order** solvers, popular for training CNNs.
- Optimization (finding local minima) based on gradient descent, i.e. taking a step proportional to the negative of the gradient Δw_t.

In Pytorch: `package torch.optim`
(implements a diverse set of optimization algorithms)

Optimization Methods

Stochastic Gradient Descent (SGD)

```
torch.optim.SGD(params, lr=<required
parameter>, momentum=0, dampening=0, weight_decay=0, nesterov=False)
```

Follow the negative gradient for a "mini-batch" of samples $\Delta w_t = -\lambda g_t$

- Requires manual setting of learning
- Manual annealing: decrease learning rate, if validation curve "plateaus" to prevent parameters from oscillating near local minima

SGD with momentum (`momentum != 0`)

Keep in memory previous weight updates $\Delta w_t = \rho \Delta w_{t-1} - \lambda g_t$

- Accelerates SGD progress when gradient points in the same direction as before and dampens oscillations

Optimization Methods

- AdaGrad
  `torch.optim.Adagrad(params,lr=0.01,lr_decay=0, weight_decay=0, initial_accumulator_value=0, eps=1e-10)`
  Adapt the learning rate per parameter, plus annealing: $\Delta w_t = -\frac{\lambda g_t}{\sqrt{\sum_{\tau=1}^{t} g_\tau^2}}$

  shrinking the learning rate over time

  - Global learning rate λ
  - Dynamic rate **per parameter**, inversely proportional to the sum of all past gradient magnitudes (accumulated till iteration t)
  - Helpful in sparse problems, e.g. natural language processing
  - Weakness: rapidly decaying learning rate
    (thus also sensitive to the choice of λ and/or weight initialization)

- RMSprop
  `torch.optim.RMSprop(params, lr=0.01, alpha=0.99, eps=1e-08, weight_decay=0, momentum=0, centered=False)`

  $$\Delta w_t = -\frac{\lambda g_t}{\sqrt{E[g^2]_t + \varepsilon}}$$

  (or $RMS[g]_t$)

  Tackle previous problem, by gradient accumulation in a fixed window, with an exponentially decaying average: $\boldsymbol{E}[g^2]_t = \rho \boldsymbol{E}[g^2]_{t-1} + (1-\rho) g_t^2$ and

  - Decay constant ρ similar to momentum
  - Denominator gradients do not accumulate infinitely

Optimization Methods

- Adam (Adaptive Moment Estimation)
  `torch.optim.Adam(params, lr=0.001, betas=(0.9, 0.999), eps=1e-08, weight_decay=0, amsgrad=False)`
- Additionally to the past *squared* gradients, also keep an exponentially decaying average of past gradients:

1st moment (mean)

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t \quad \rightarrow \quad \widehat{m_t} = m_t/(1-\beta_1^t)$$

2nd moment (variance)

$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \quad \rightarrow \quad \underbrace{\widehat{v_t} = v_t/(1-\beta_2^t)}_{\text{“bias correction”}}$$

$$\Delta w_t = -\frac{\lambda}{\sqrt{\widehat{v}_t + \varepsilon}} m_t$$

- Adds momentum to RMSprop
- Suggested decay: $\beta_1 = 0.9, \ \beta_2 = 0.999$. Initial averages: zeros

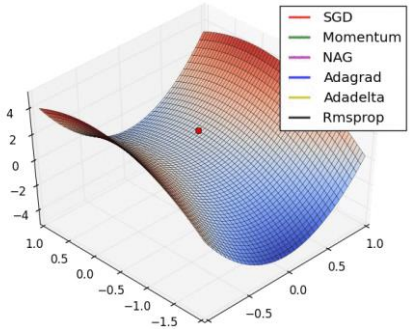
Kingma and Ba. "Adam: A method for stochastic optimization." *ICLR'15*.

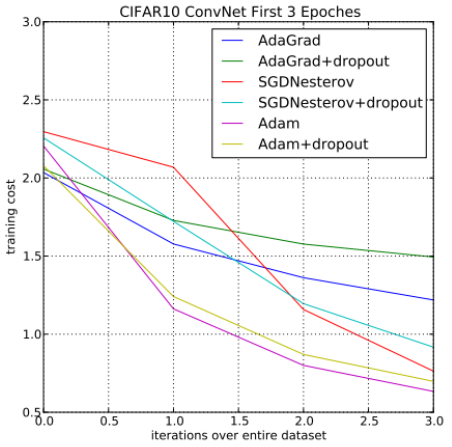
Optimization Methods

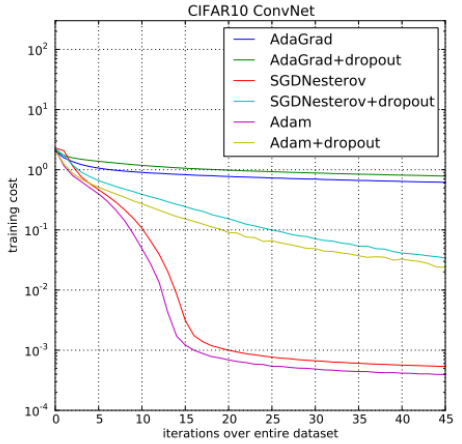
Summary:

- Adaptive learning rate optimizers
  - do not depend strongly on setting the learning rate
  - are good for sparse input data
  - Adam is a safe option

- SGD with momentum
  - is often used
  - requires the tuning and annealing of learning rate
  - when well-tuned, it usually converges fast and at better final values.

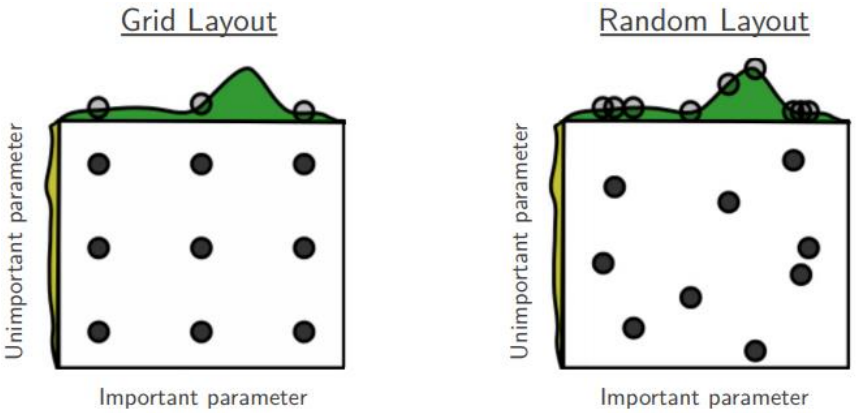






Hyperparameter tuning

- Network architecture
  - Number and types of layers (capacity)
  - Kernel sizes, number of channels
- Optimizer
  - Learning rate
  - Decay schedule
  - Update type
- Regularization
  - L2 weight decay
  - Dropout

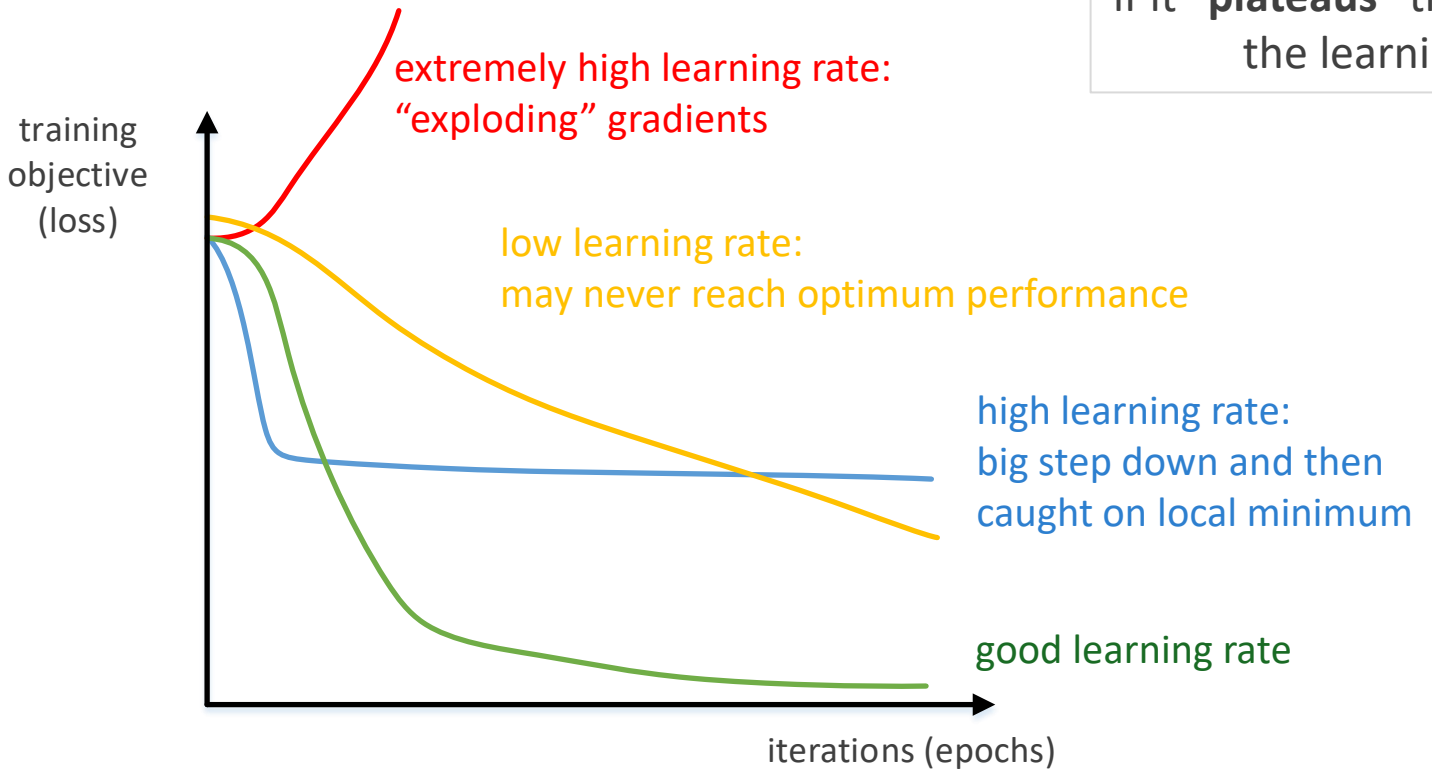


Bergstra, James, and Yoshua Bengio. "Random search for hyper-parameter optimization." *Journal of Machine Learning Research* 13.Feb (2012): 281-305.

How to train your network

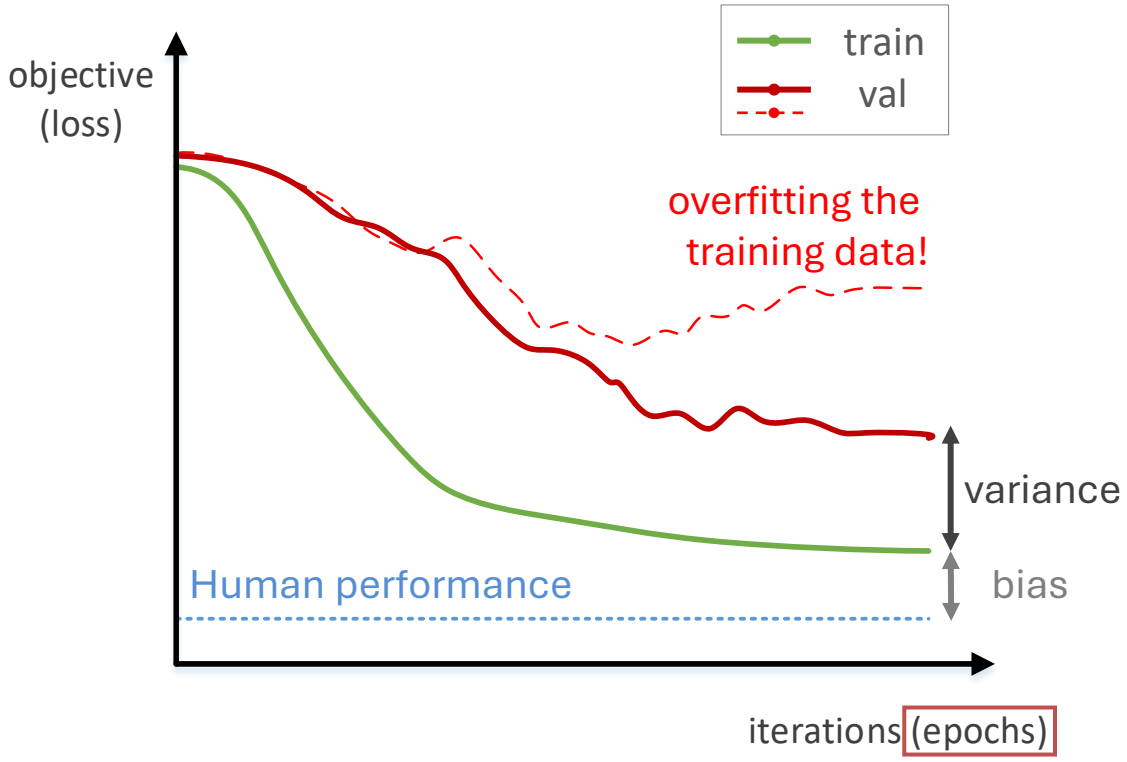
Looking for the right learning rate...

As for the validation curve, if it "**plateaus**" then decrease the learning rate



How to train your network

Validation vs Training (when data comes from the same distribution)



If **bias** is high:
Train a bigger model
(increase capacity)
or train longer

If **variance** is high:
Try more data,
augmentations,
regularization (e.g. dropout)

If **overfitting**:
Try more data or
early stopping

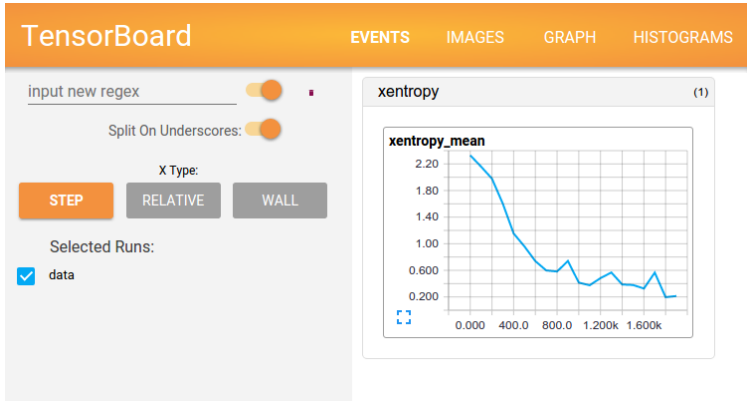
It makes sense to check the validation loss over the entire validation set

How to train your network

Best practices & sanity checks:

- Vanishing gradients
  (early layers do not train because gradients vanish towards 0 → use ReLU)
- Exploding gradients – weights too large
  (batch normalization, lower learning rate)
- Overfit a small subset of data.
  Turn off regularization.
  Achieve training error close to zero on a tiny set.
  (i.e. the network is able to “learn by heart” the given task,
  as a way to ensure a bug-free implementation)
- Make sure the initial loss has a reasonable value.
  (e.g. a **10-class** classification problem should yield a starting softmax
  cross entropy loss of approximately ~ln(1/10) = 2.302 for the correct class)

TensorBoard: Visualizing Learning



TensorFlow Interface that allows for

- Plotting losses metrics during execution
- Monitoring the training process
- Now also works with PyTorch: `pip install tensorboardX`

- **Summary operations**, e.g. loss, images

```
writer = SummaryWriter('logs')
```
← will create an event file

```
for train_step in range(num_train_steps):
    ...
    pred = model(x)
    loss = compute_loss(gt, pred)
    writer.add_scalar('loss', loss, train_step)
    writer.add_image('image', x, train_step)
    ...
```

update the event file

- Navigate to http://localhost:6006 after running `tensorboard --logdir logs`

Saving / Restoring weights

Model weights are stored in checkpoint files

- A dictionary of the form:

```
layer.weight: tensor
```

```
model = MyModelClass(*args)
PATH = ‘/my/checkpoint/path/checkpoint.pt’
```

- Saving
  `torch.save(model.state_dict(), PATH)`

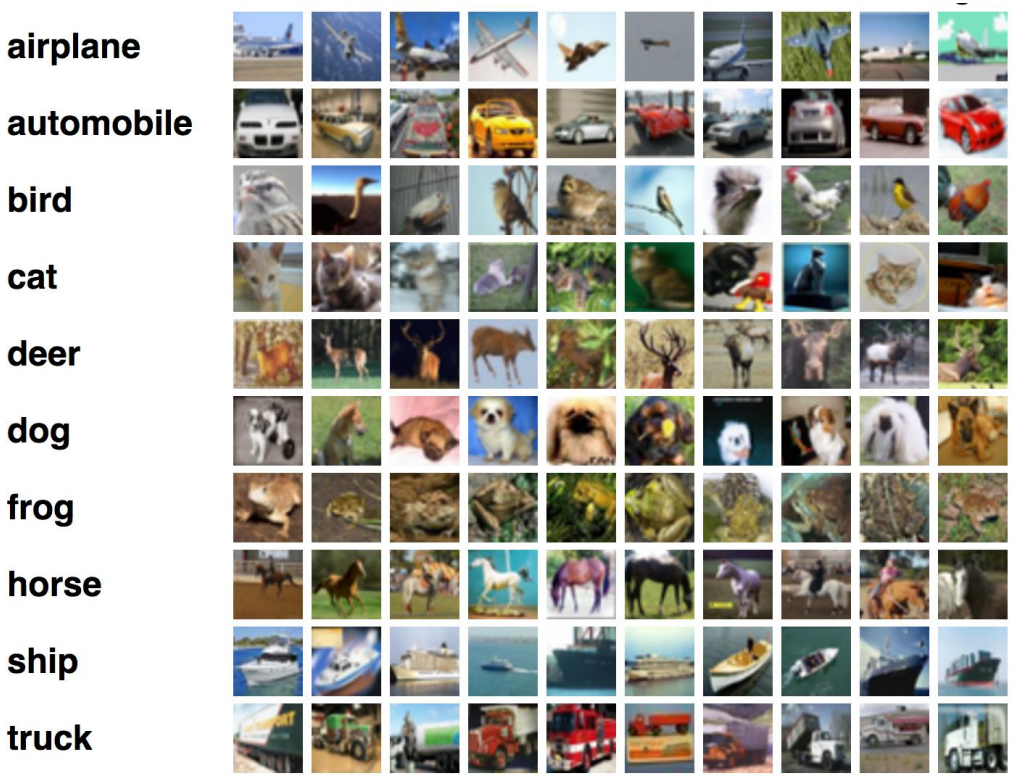
- Restoring
  `model.load_state_dict(torch.load(PATH))`

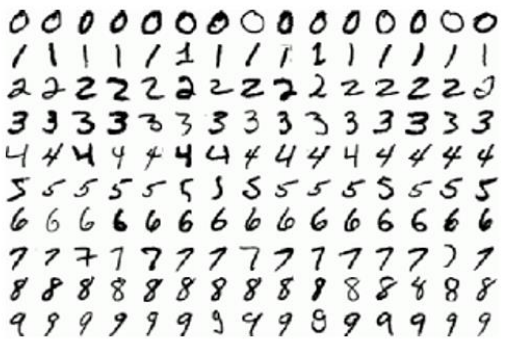
- Parameter names in the checkpoint must match those in the graph for successful restoring!

Break ☺

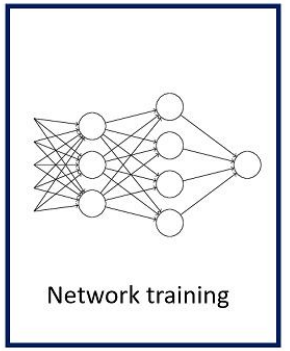
CNNs for Computer Vision Tasks

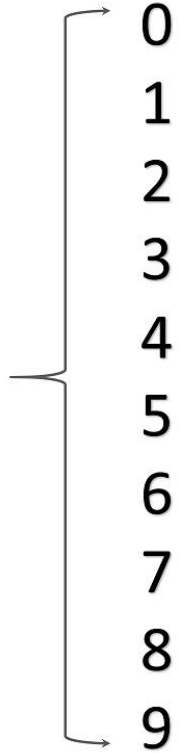
Image Classification



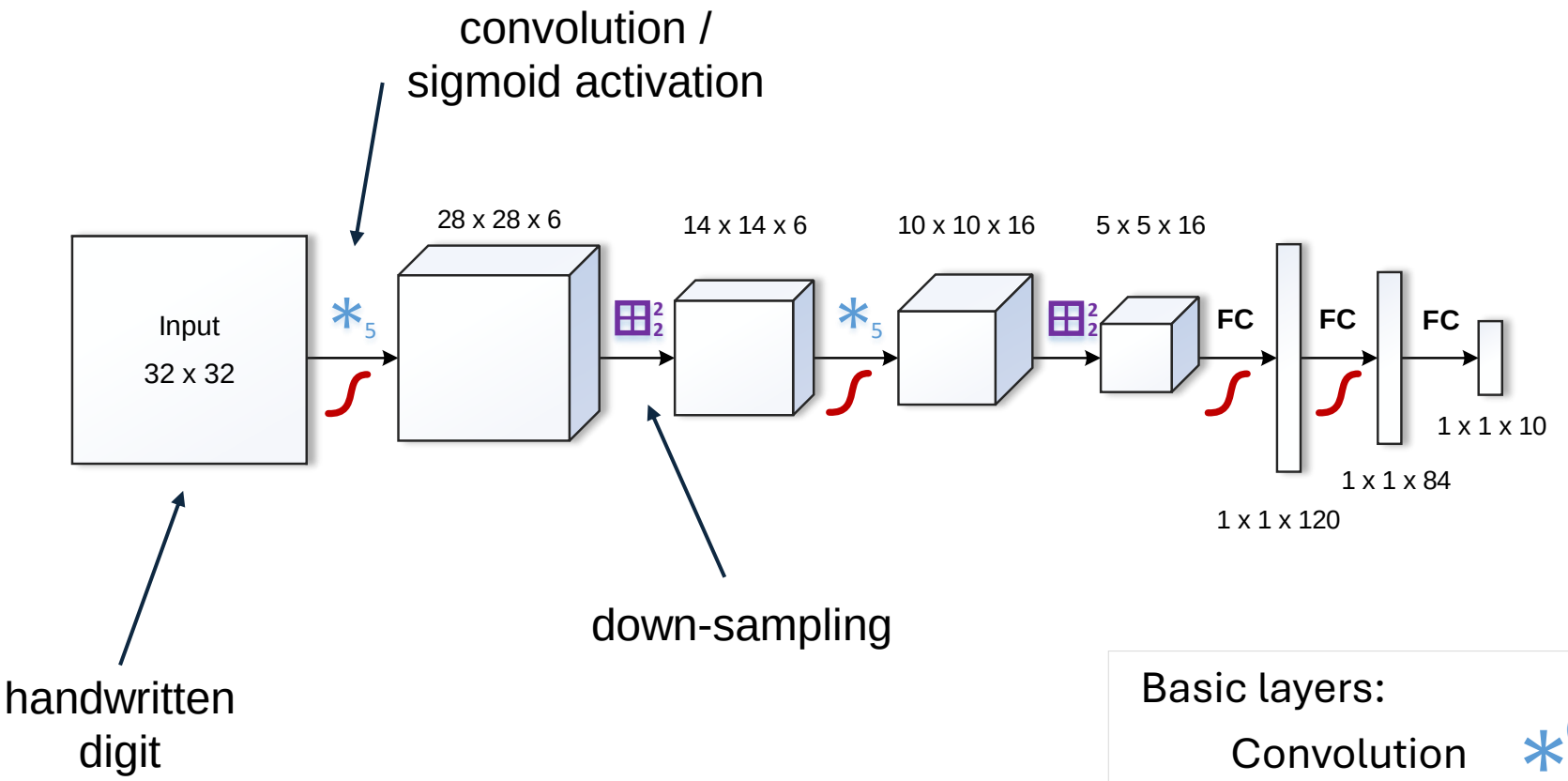


Data & Labels





(Recap) LeNet – 5



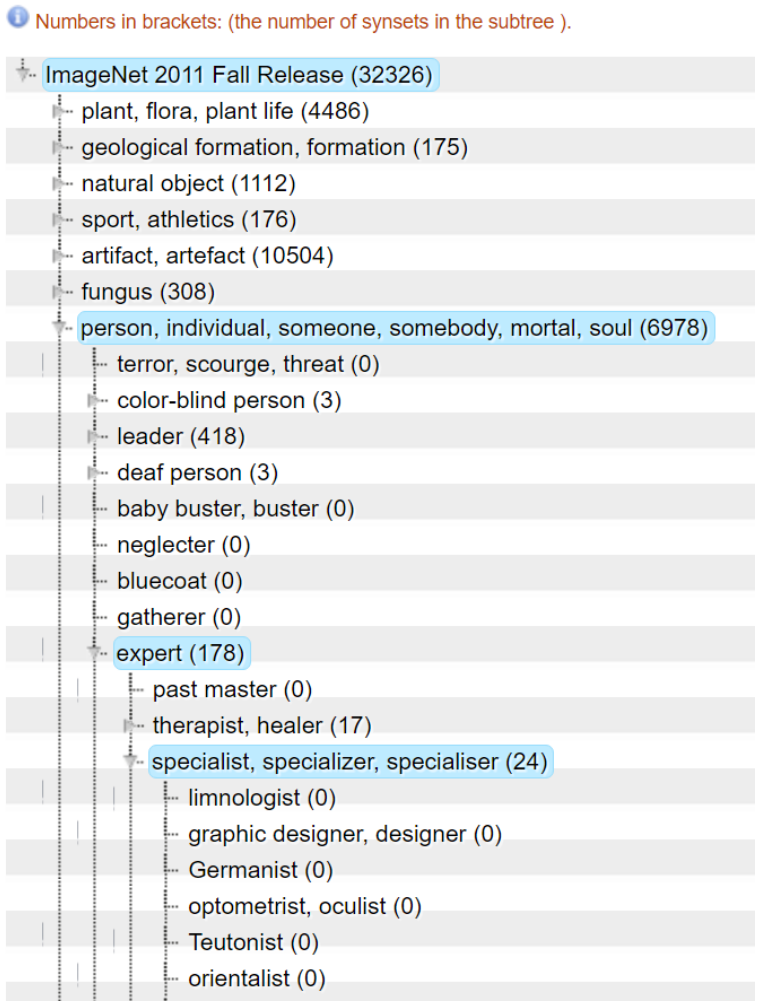
Basic layers:

- Convolution
- Pooling
- Activation

ImageNet

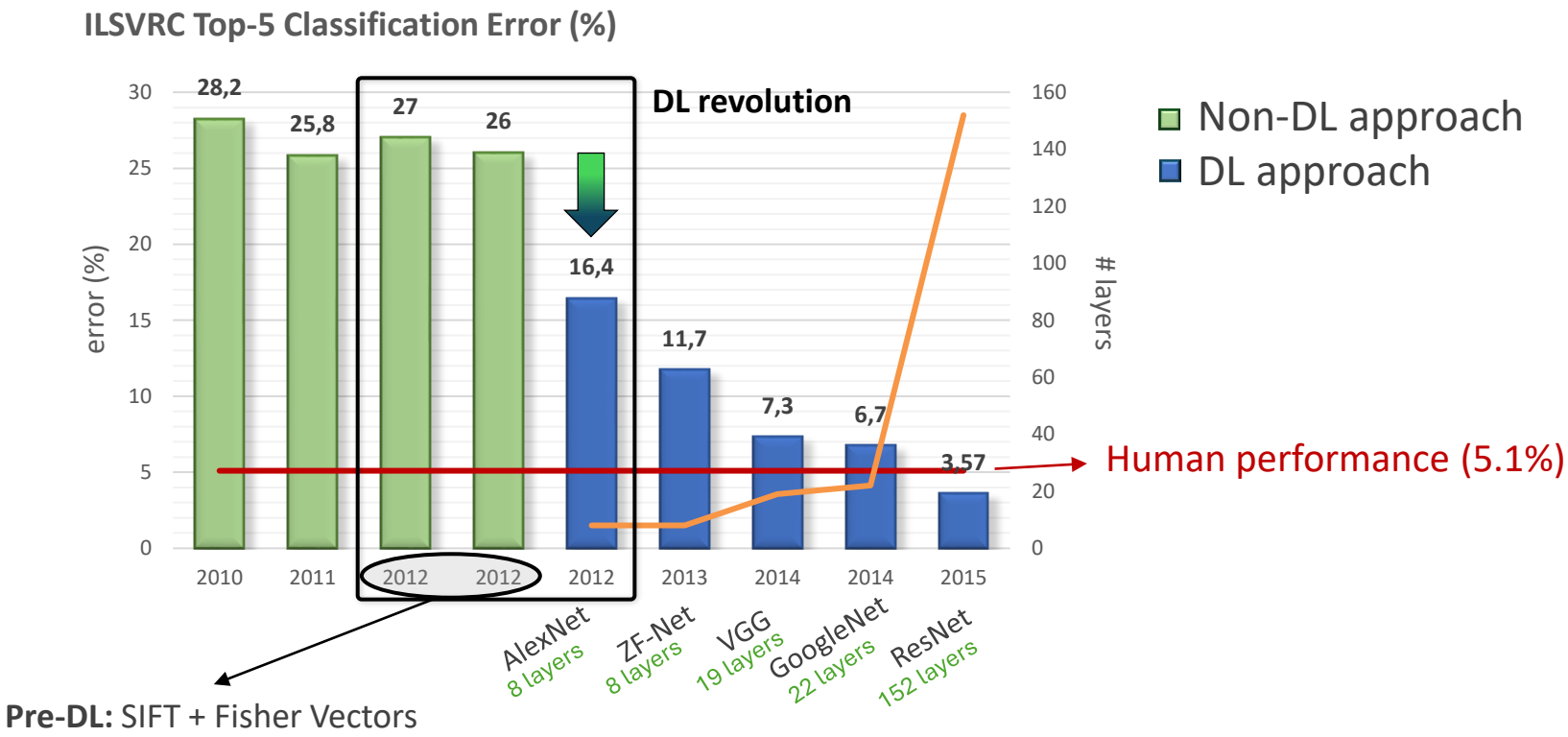
- Huge collection of images (over 10 million)
- Hierarchical structure from WordNet
- Annotated with 1000 classes
- Yearly: Large Scale Visual Recognition Challenge (ILSVRC)
- Since 2012: CNNs win everything

IMAGENET

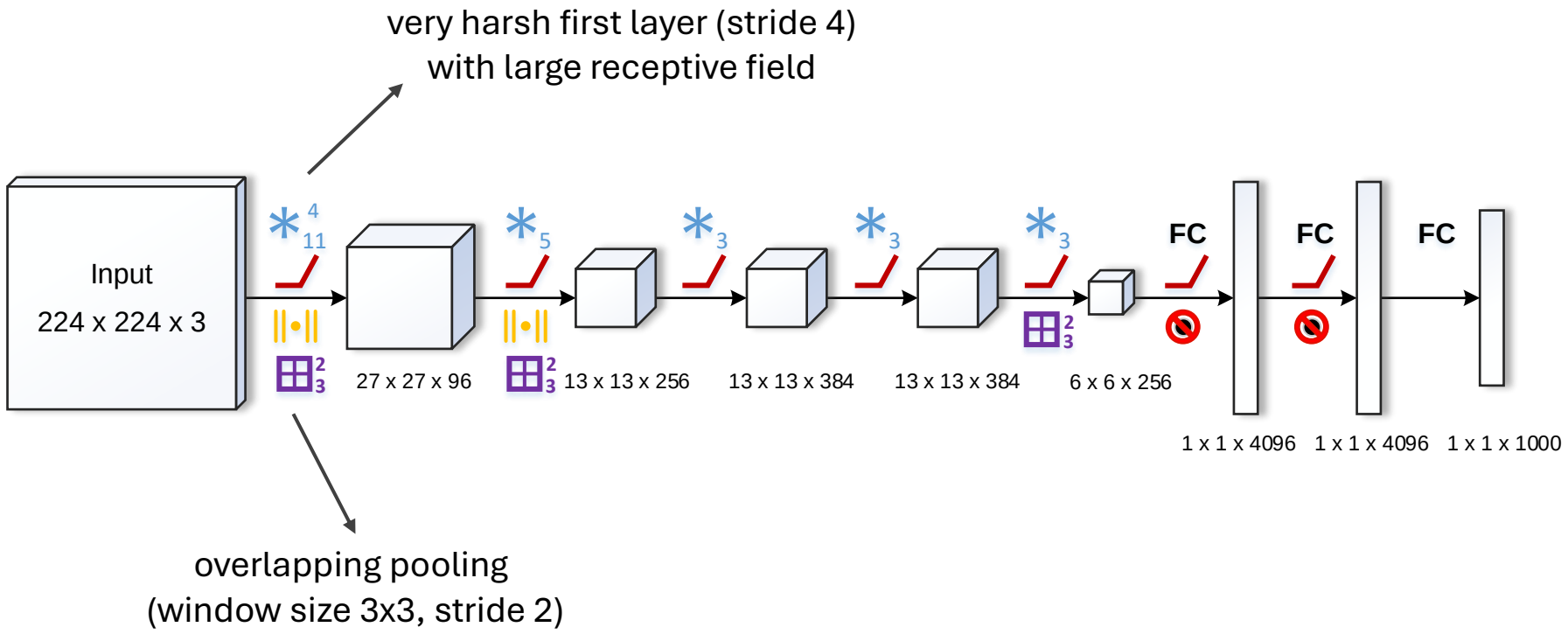


ImageNet Challenge

IMAGENET



AlexNet (2012)



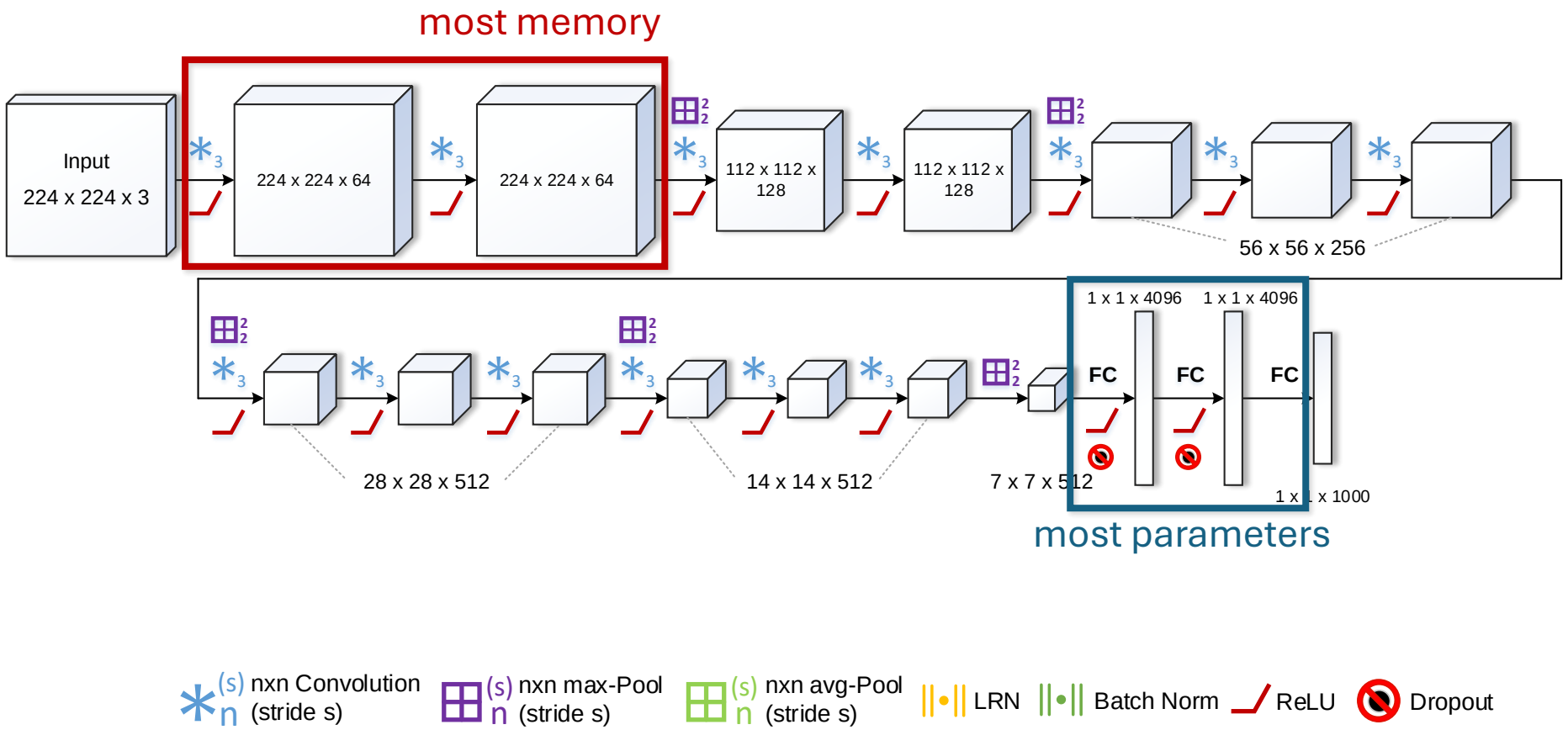
Krizhevsky et al., “Imagenet classification with deep convolutional neural networks”, NIPS12

AlexNet (2012)

- Made modern CNNs popular
- Trained in parallel on two GPUs with custom code
- Introduced:
  - ReLU instead of sigmoid
  - Dropout
  - Regularization (data augmentation, weight decay)
  - Local Response Normalization (LRN)
- 60 million parameters

Krizhevsky et al., “Imagenet classification with deep convolutional neural networks”, NIPS12

VGG-16 (2014)



Simonyan et al., “Very deep convolutional networks for large-scale image recognition”, arXiv preprint arXiv:1409.1556(2014).

VGG-16 (2014)

- VGG “very deep” networks
  - Experiments with 11 to 19 convolutional layers
- Repeating small (3x3) convolutions

Why?

A: More non-linearities increase network‘s discriminative capability

Why small filters?

A: Fewer parameters per layer

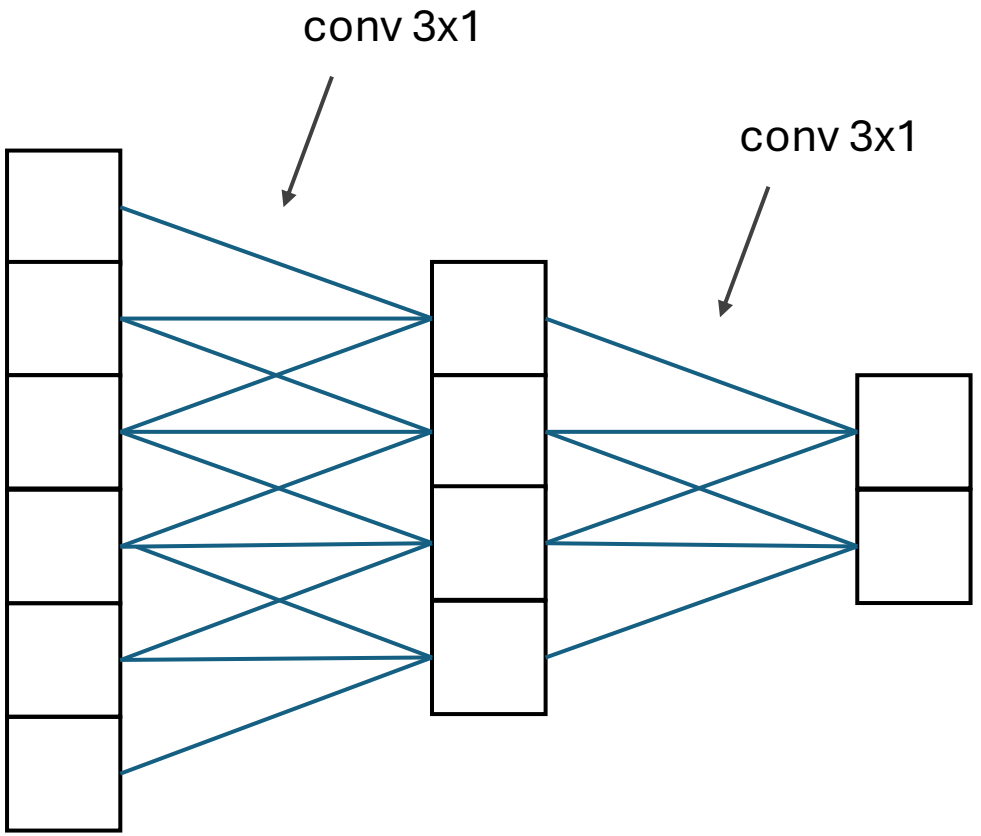
ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

133+ million parameters
(double compared to AlexNet)

Simonyan et al., “Very deep convolutional networks for large-scale image recognition”, arXiv preprint arXiv:1409.1556(2014).

(Background) Receptive field

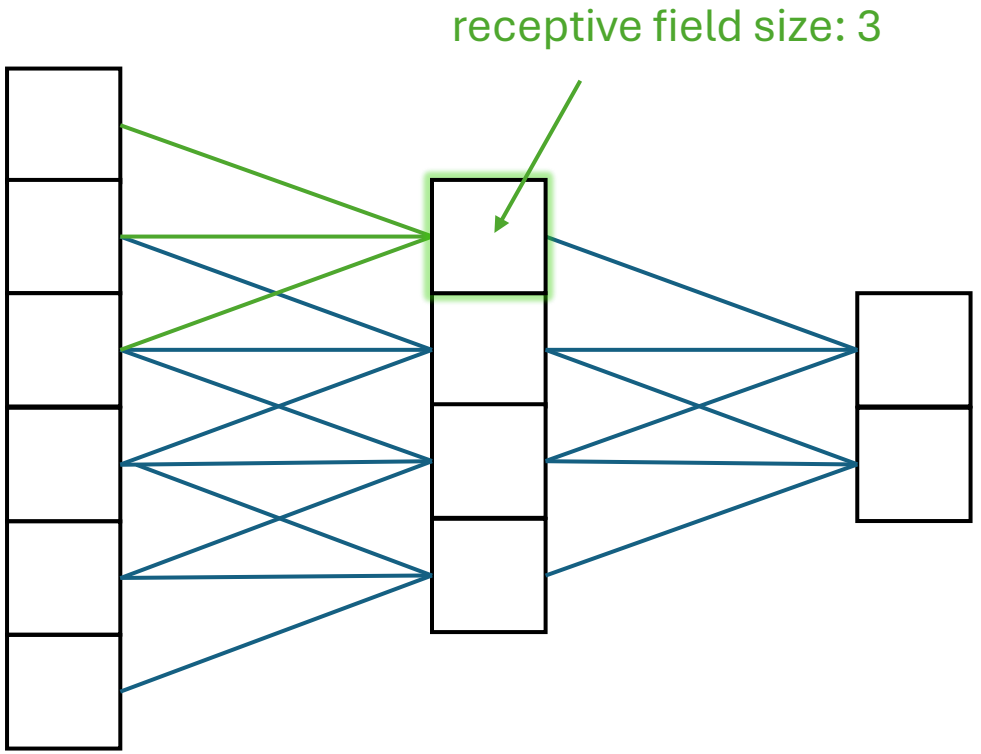
1D Example



Size of the maximal area in an earlier layer
(traceable back to the input layer)
that a neuron can see and receive inputs from

(Background) Receptive field

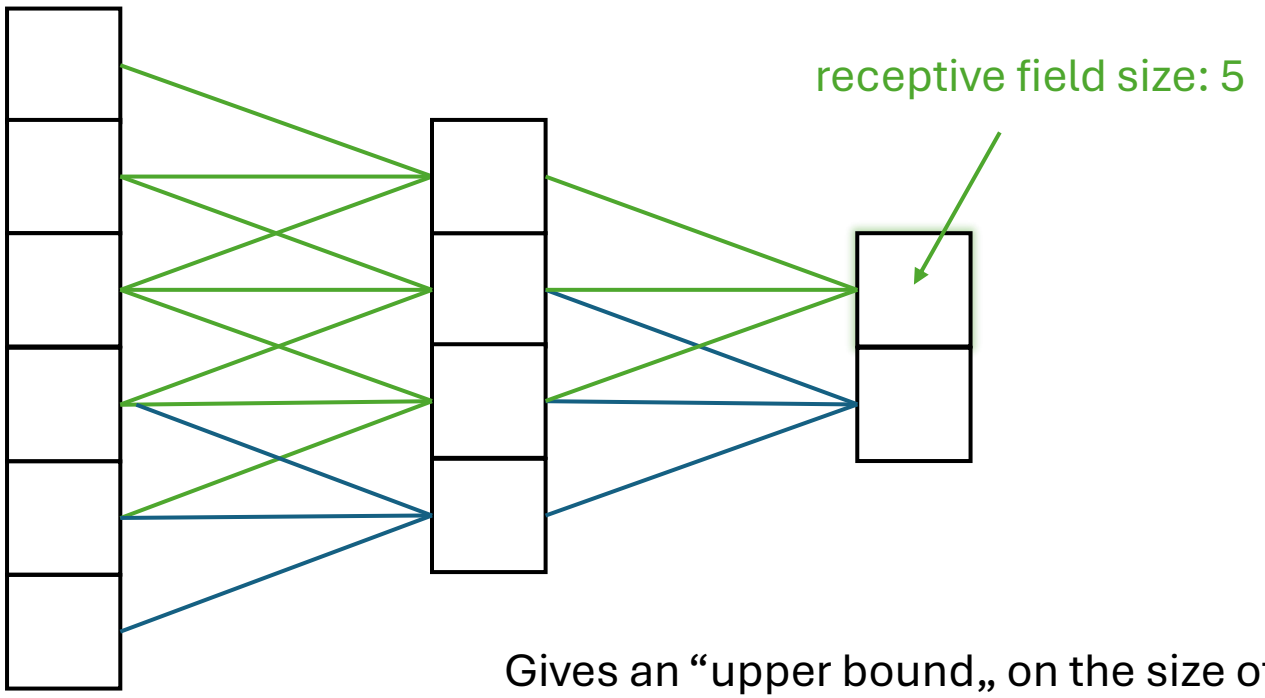
1D Example



Size of the maximal area in an earlier layer
(traceable back to the input layer)
that a neuron can see and receive inputs from

(Background) Receptive field

1D Example

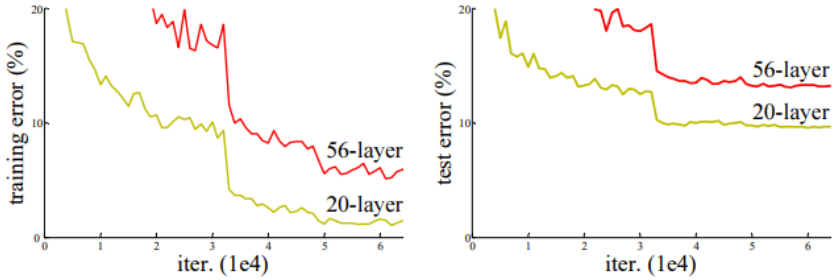


Gives an “upper bound„ on the size of concepts/patterns the CNN can learn.

Thus, with the network’s increasing depth, the receptive field typically increases too.

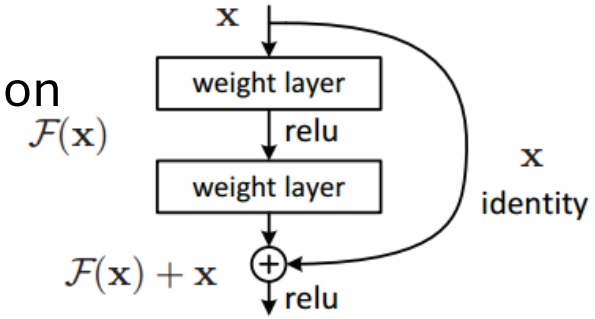
ResNet (2015)

He et al., “Deep Residual Learning for Image Recognition”, CVPR (2015).



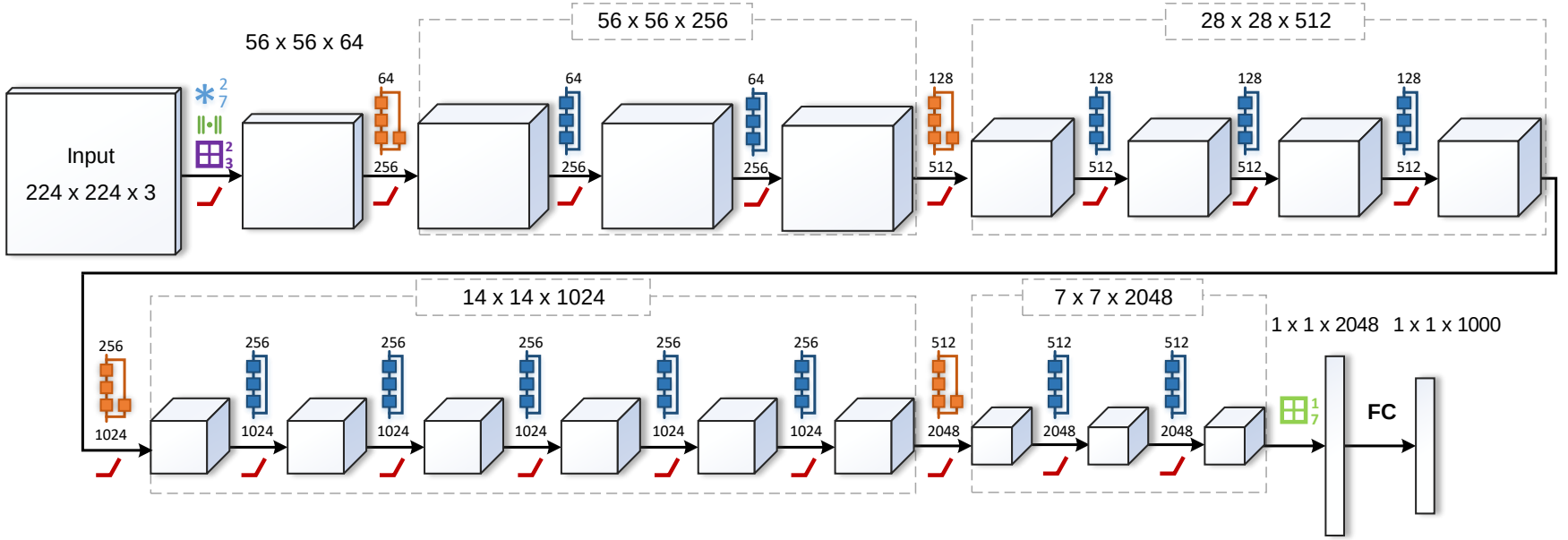
Training error increased with more layers.
This is not over-fitting, but likely an *optimization* problem.

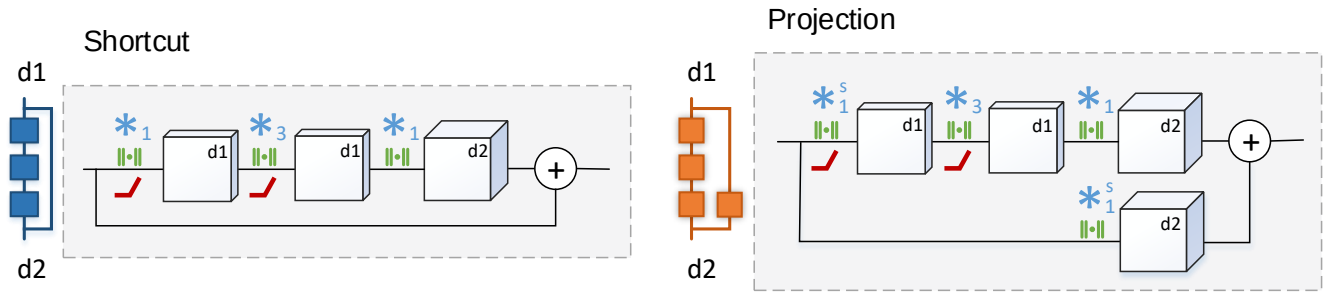
- Idea: Residual Learning
  “It is easier to optimize a residual mapping than the original unreferenced mapping.”
- Identity mapping: parameter-free shortcut (*skip*) connection
  - Output becomes: $\mathcal{F}(x) + x$
- Allows for ultra deep networks (up to 1000 layers!)



ResNet (2015)

He et al., “Deep Residual Learning for Image Recognition”, CVPR (2015).

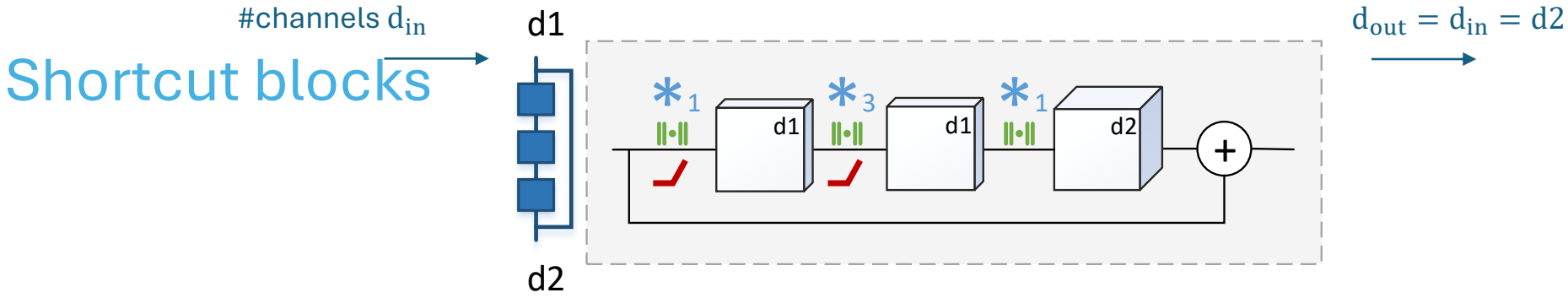




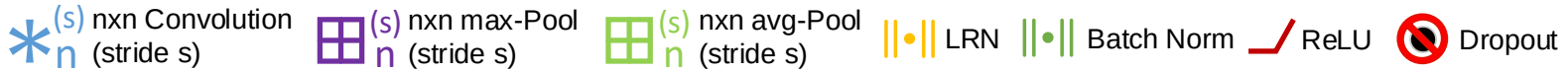
(s) nxn Convolution (stride s)
(s) nxn max-Pool (stride s)
(s) nxn avg-Pool (stride s)
LRN
Batch Norm
ReLU
Dropout

ResNet (2015)

He et al., “Deep Residual Learning for Image Recognition”, CVPR (2015).



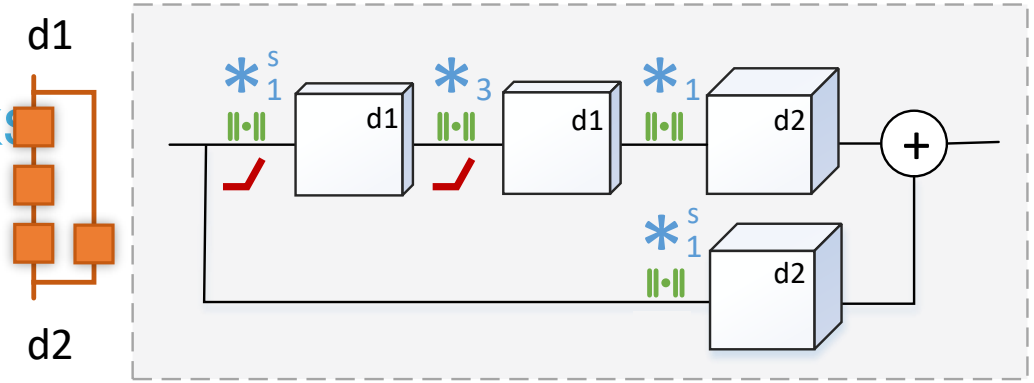
- It is the building block instead of simple convolutions
- The first 1x1 conv (“bottleneck„) reduces the number of filters, the second increases it again (d1 < d2). ReLU follows after the sum.
- Makes 3x3 convolutions less heavy on resources.
- Each convolution is followed by batch normalization.



ResNet (2015)

He et al., “Deep Residual Learning for Image Recognition”, CVPR (2015).

Projection blocks



Projection blocks are used when sizes need to change, by adding a convolution on the “skip” branch to handle this.

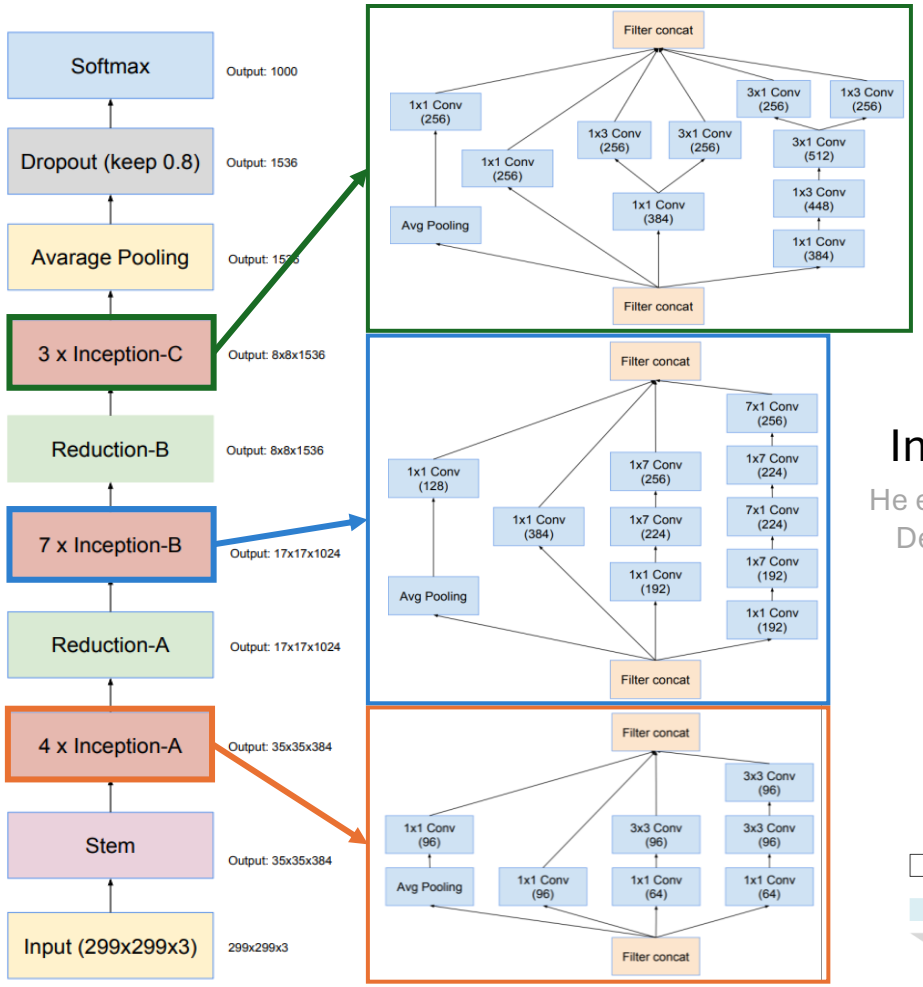
- Used for downsampling
  First convolution in each branch is with stride (no pooling!)
- Also used to change the number of channels (when $d_{out} \neq d_{in}$)

(s) n nxn Convolution (stride s) | (s) n nxn max-Pool (stride s) | (s) n nxn avg-Pool (stride s) | LRN | Batch Norm | ReLU | Dropout

… and beyond

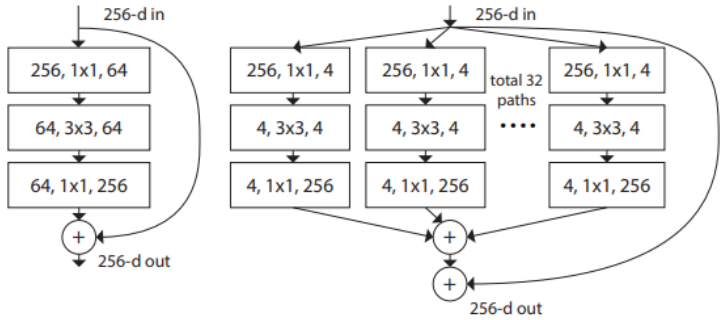
Inception-v4

Szegedy et al., “Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning”, AAAI’17



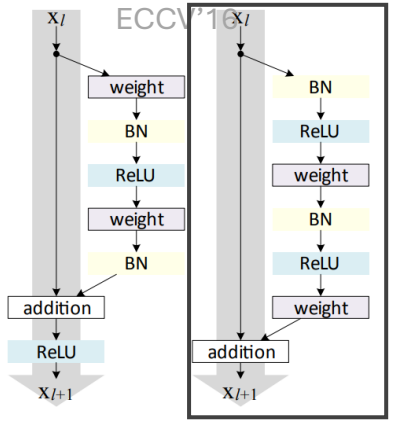
ResNeXt

Xie et al., “Aggregated Residual Transformations for Deep Neural Networks”, CVPR’17



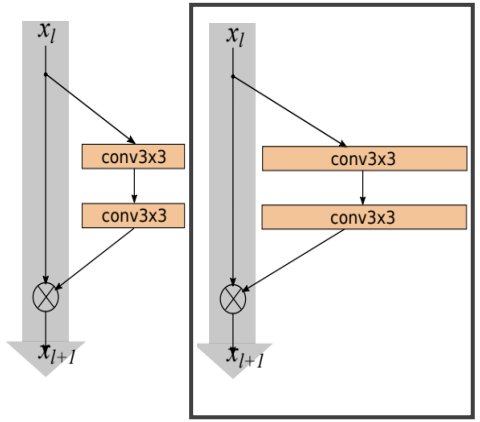
Improved res-block

He et al., “Identity Mappings in Deep Residual Networks ”, ECCV’16



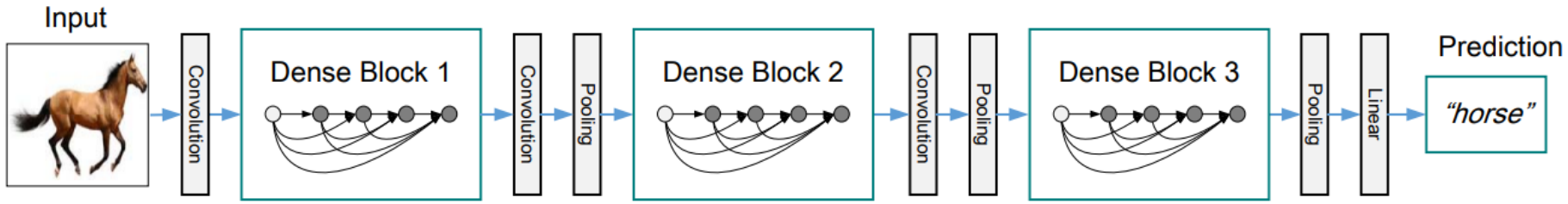
Wide ResNet

Zagoruyko and Komodakis, “Wide Residual Networks ”, arXiv:1605.07146



DenseNet (2017)

Huang et al., “Densely Connected Convolutional Networks”, CVPR (2017).



- Dense block :
  every layer is connected to
  every other layer (feed-forward)
- Concatenation of feature maps
- Layers are narrow (small #channels)
- Alleviate vanishing gradients
- Strengthen feature propagation
- Reduce #parameters

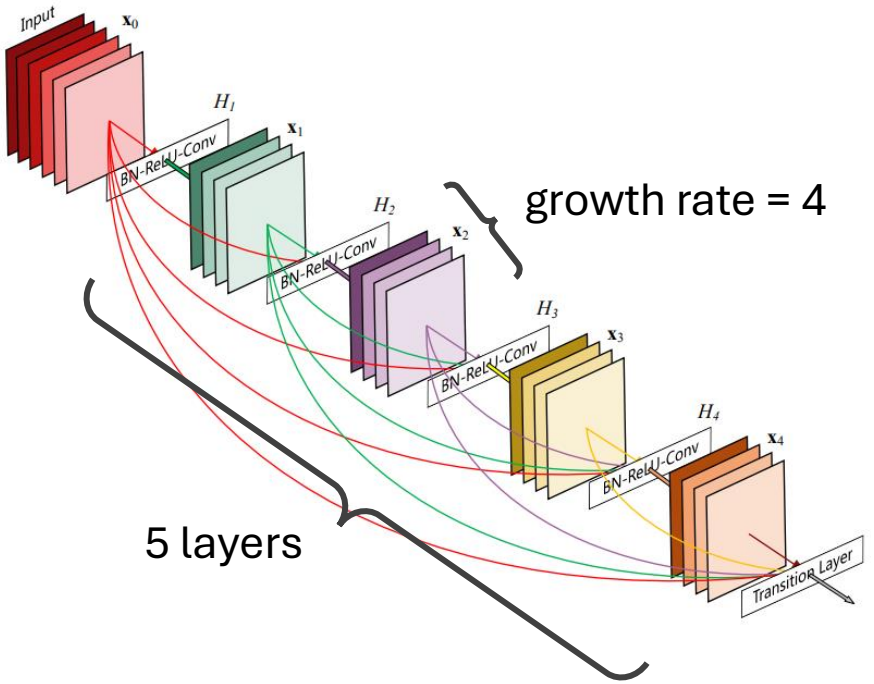
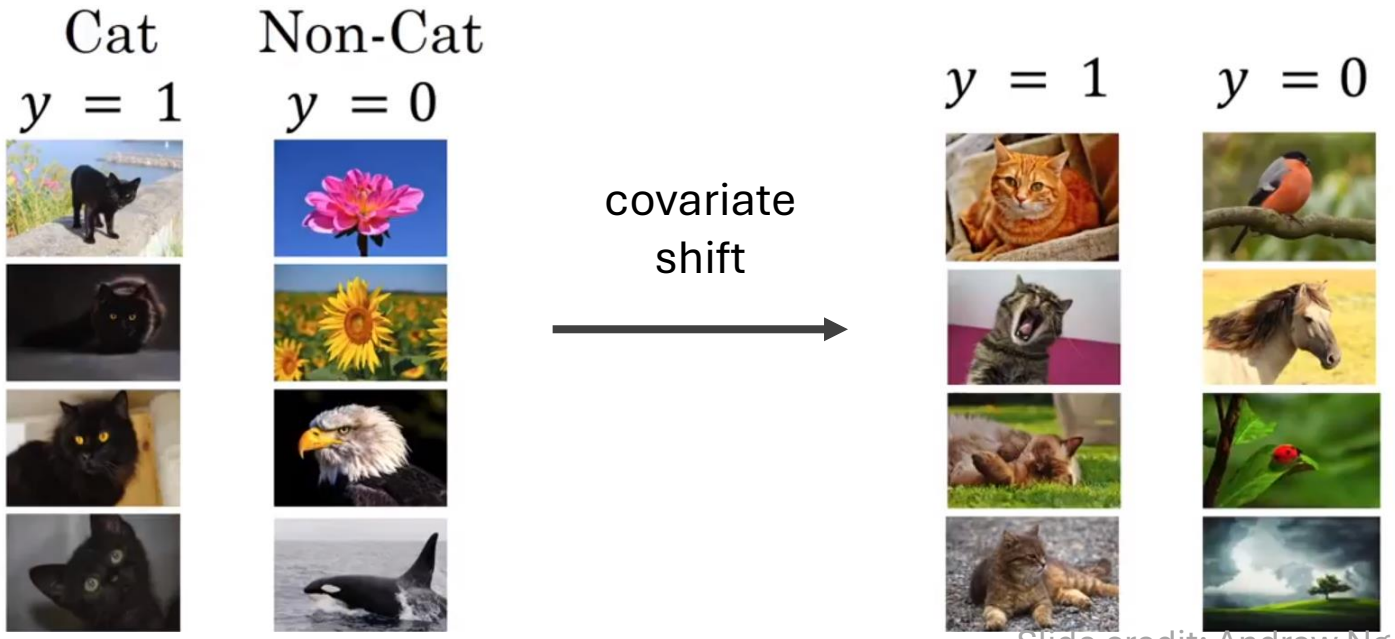


Image classification - torchvision example

https://colab.research.google.com/drive/1-vdPU1KNRk81ikF6-1Qn9e8CLpr3T-O8?usp=sharing

Batch Normalization

- Motivation



Slide credit: Andrew Ng

If a model trained for cat/non-cat classification has seen only black cats,
it will not perform well when the distribution changes (i.e. other colors of cats)
Good practice: Normalize inputs. Why not also normalize activations?

Batch Normalization

Idea: Normalize each activation over the current batch

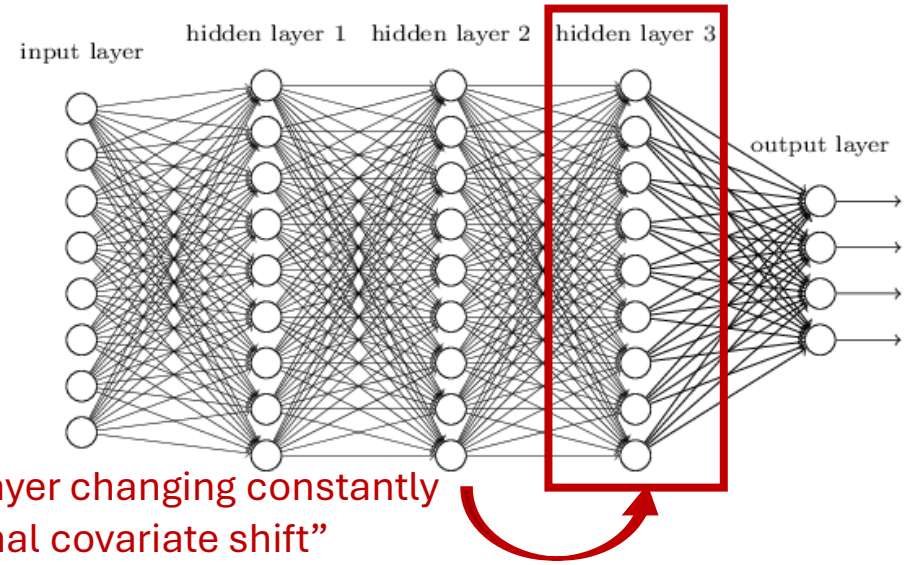
To address the problem of internal covariate shift, for every neuron activation x individually over the batch $x^{(1)} \dots x^{(m)}$ do:

1. Mean: $\mu = \frac{1}{m}\sum_{i=1}^{m} x^{(i)}$
2. Variance: $\sigma^2 = \frac{1}{m}\sum_{i=1}^{m}\left(x^{(i)} - \mu\right)^2$
3. Normalize: $\hat{x}^{(i)} = \frac{x^{(i)}-\mu}{\sqrt{\sigma^2+\epsilon}}$

   Now, the batch follows a fixed distribution with zero mean and unit variance

4. Scale and Shift: $y^{(i)} = \gamma\hat{x}^{(i)} + \beta$

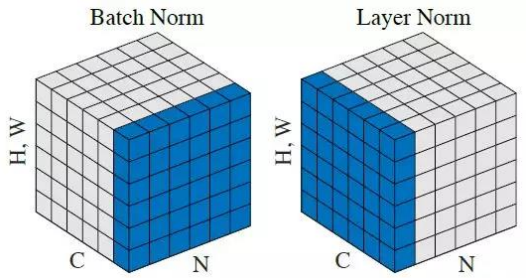
   Learn these parameters (γ, β) for every activation



Ioffe et al. "Batch normalization: Accelerating deep network training by reducing internal covariate shift"arXiv 1502.03167 (2015)

Batch Normalization

- What does it do:
  - Reduces the amount that the (previous layer) distribution shifts around
  - Allows layers to learn *independently*
  - Mean subtraction cancels biases of previous layer

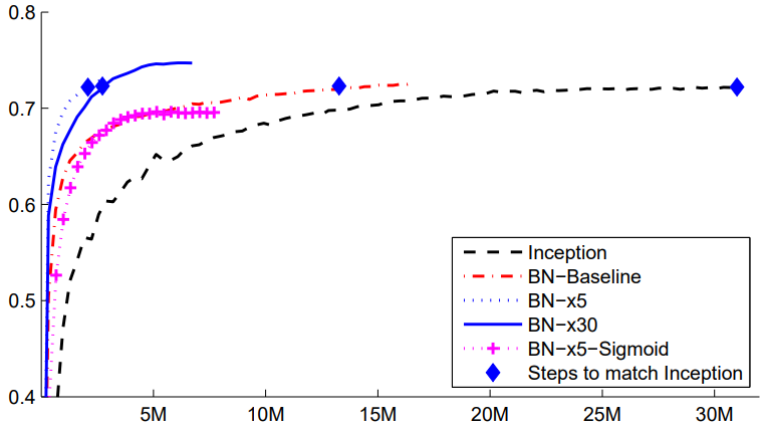


- How to:
  - Usually applied directly after each convolution
  - During *training*:
    - calculate and apply mini-batch statistics (μ, σ)
    - update population statistics $(\mu_{train}, \sigma_{train})$ via a *moving average* over mini-batches
  - During *testing*:
    - often just one image (no batch) → no statistics
    - use the estimated population statistics $(\mu_{train}, \sigma_{train})$ to normalize the activations

Ioffe et al. "Batch normalization: Accelerating deep network training by reducing internal covariate shift“ arXiv 1502.03167 (2015)

Batch Normalization

- The effect:
  - Allows increasing learning rate
    (because the range of activation values is controlled)
  - Slight regularization effect
    (because of adding some noise)
  - Reduces need for dropout
  - Solves vanishing gradients
  - Faster convergence
    (because the layers learn faster)



Ioffe et al. "Batch normalization: Accelerating deep network training by reducing internal covariate shift“
arXiv 1502.03167 (2015)

Batch Normalization

- In PyTorch:

```
torch.nn.BatchNorm1d()
torch.nn.BatchNorm2d()
...
```

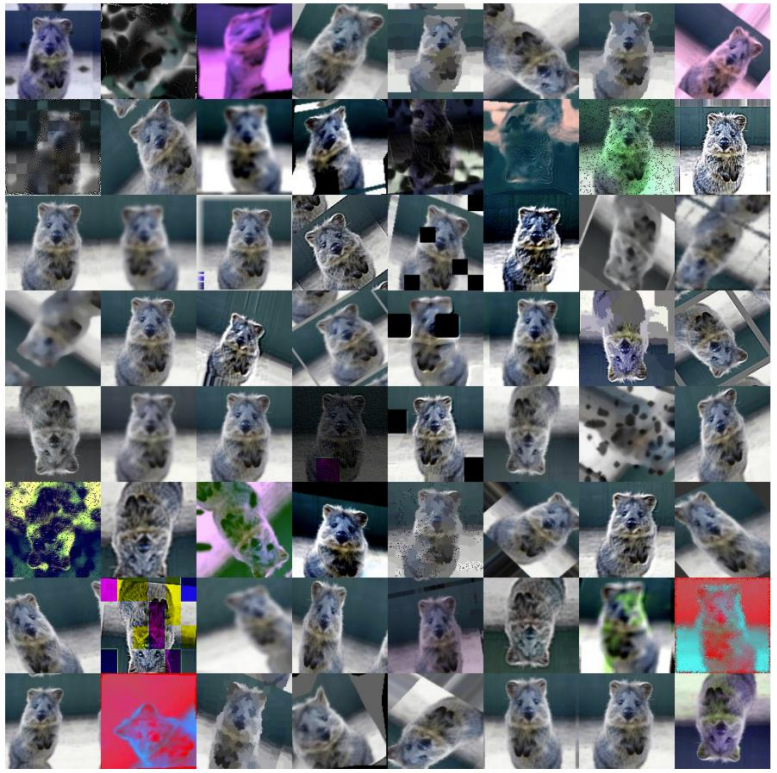
Important:
take care of the mode: `model.train()` or `model.eval()`

Different behaviors for batch norm!

Data Augmentation

Random combination of

- Rotations (e.g. in [-20, 20] degrees)
- Scaling [sc_min, sc_max]
- Translations (crops)
- Color jitter (e.g. a multiplicative in $\mathbb{R}^3$ or additive/brightness factor)
- Contrast change
- …



https://github.com/aleju/imgaug

Data Augmentation

Random combination of

- Rotations (e.g. in [-20, 20] degrees)
- Scaling [sc_min, sc_max]
- Translations (crops to fixed/input size)
- Color jitter (e.g. a multiplicative in $\mathbb{R}^3$ or additive/brightness factor)
- Contrast change
- …

Just for training!

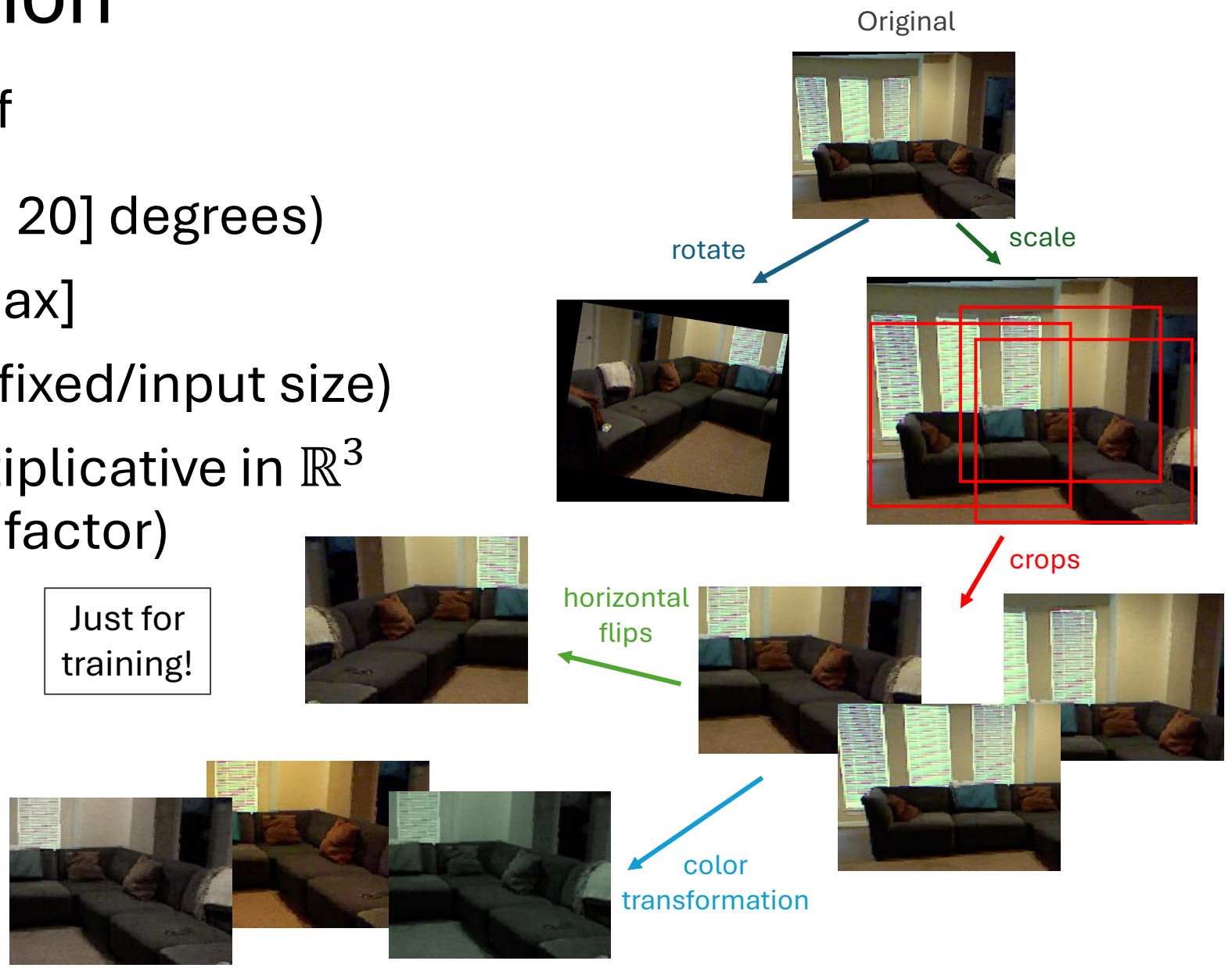


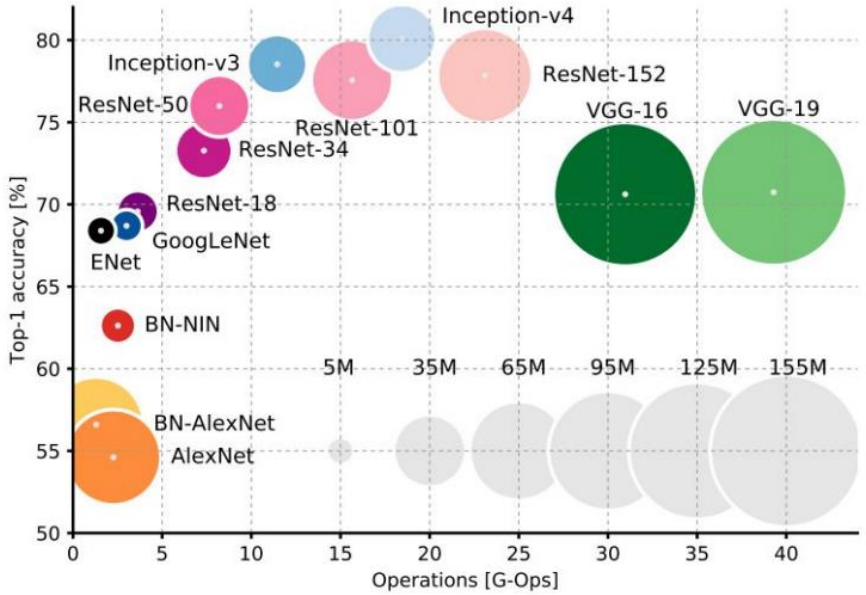
Image classification : Take home message

"Don't be a hero" – *Andrej Karpathy*
(i.e. start with an architecture that is proven to work!)

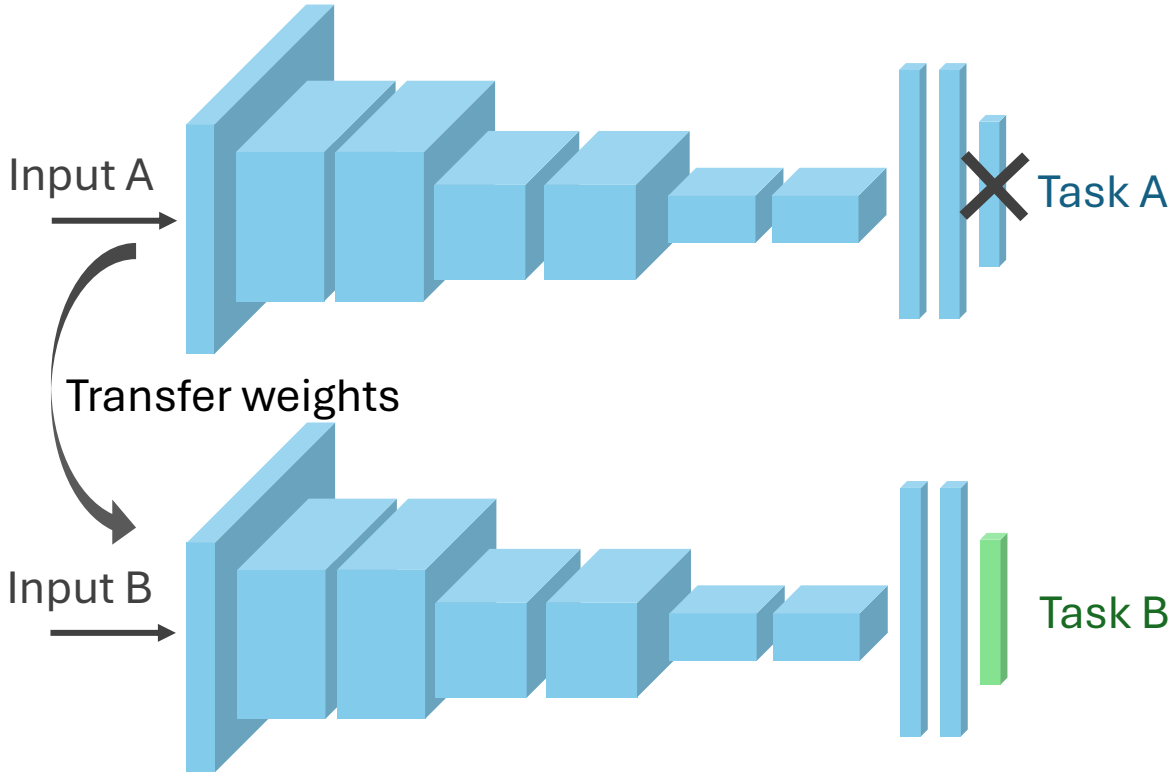
Trends:

- go deeper
- keep #parameters low
- DAG-style (multiple connections)
- improve gradient flow

Comparison of architectures



Transfer Learning

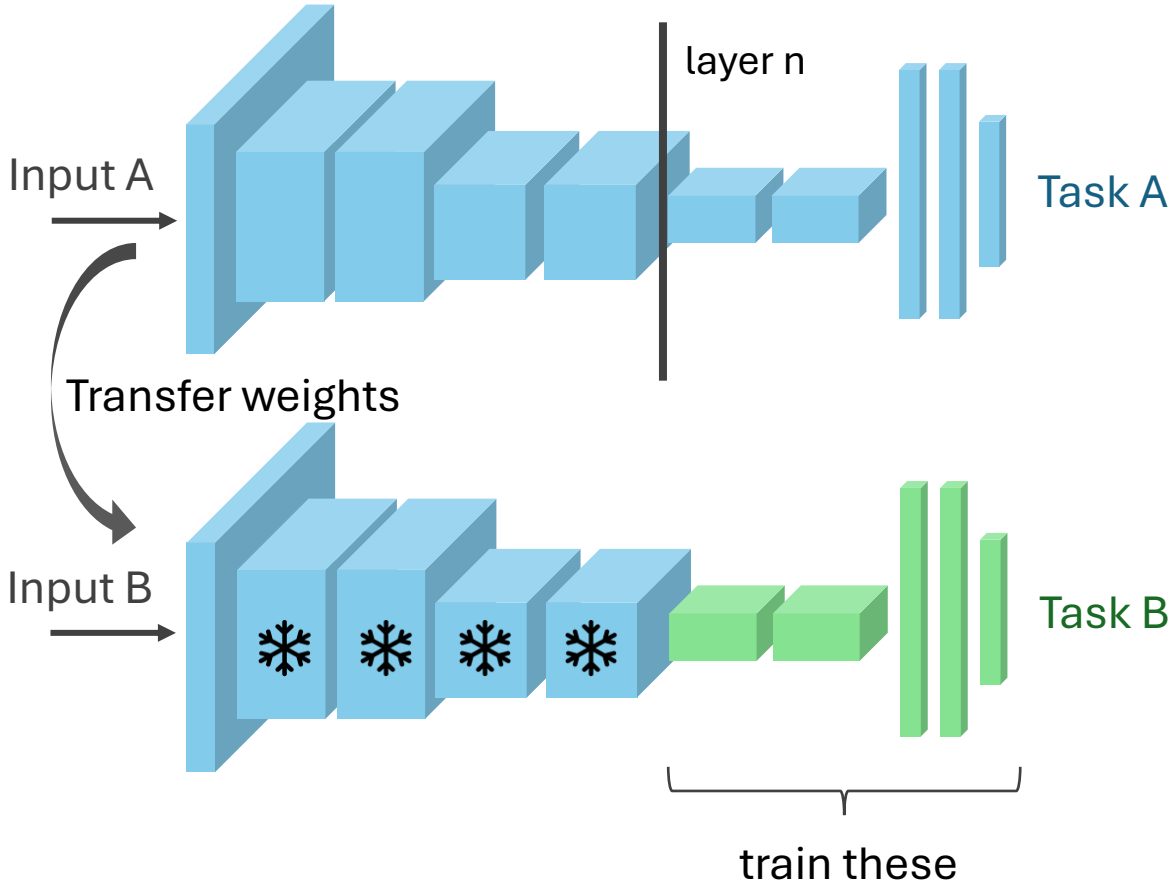


Transfer Learning
is useful when we have
a lot of data in the problem
we are transferring from and
few data for the problem we
are transferring to.

Task A and Task B might have
different output dimensions

- Chop off prediction layer
- Replace with new layer
  with correct dimensions
- Re-initialize
  new layer only

Transfer Learning



It is also possible to re-initialize several layers or add new layers.

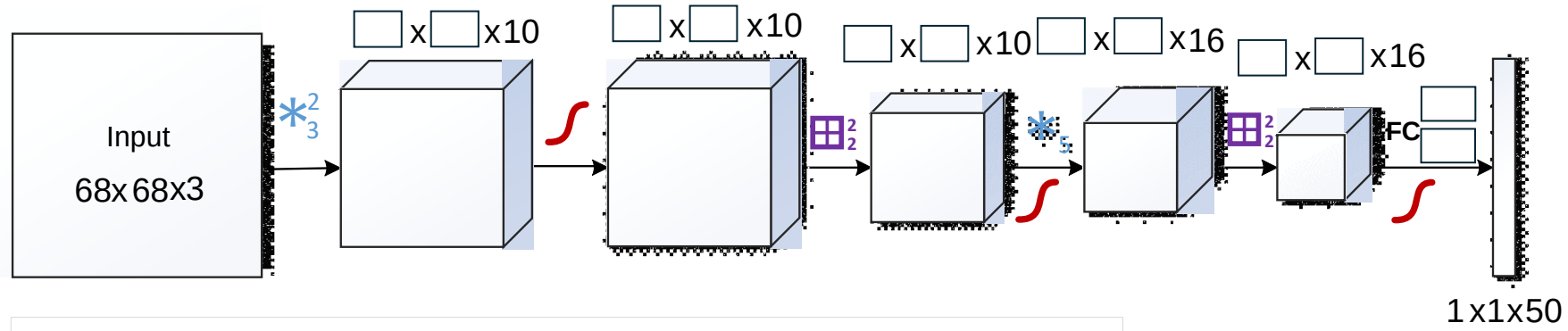
This depends on how different task B is and the amount of available data.

Example:
pre-training on ImageNet
finetuning on new data

Exercise

No need to send the solution back to me, but you can ask questions!

Define the missing dimensions for the feature maps and layer weights below.
How does padding / no padding affect the result?



Basic layers:

Convolution $*_{n}^{(s)}$ *s*- stride, *n*-kernel size

Pooling $\boxplus_{n}^{(s)}$

Activation

Fully-connected $FC_{output\_dim}^{input\_dim}$