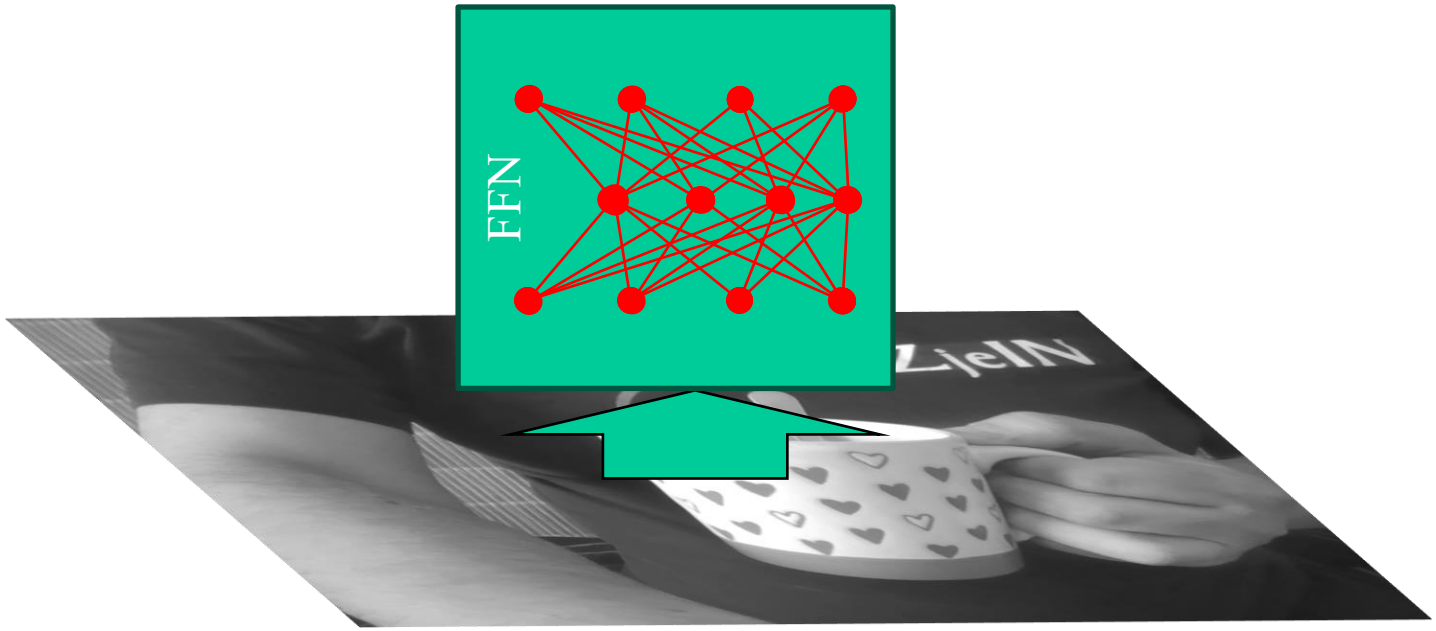
Convolutional autoencoder and classifier

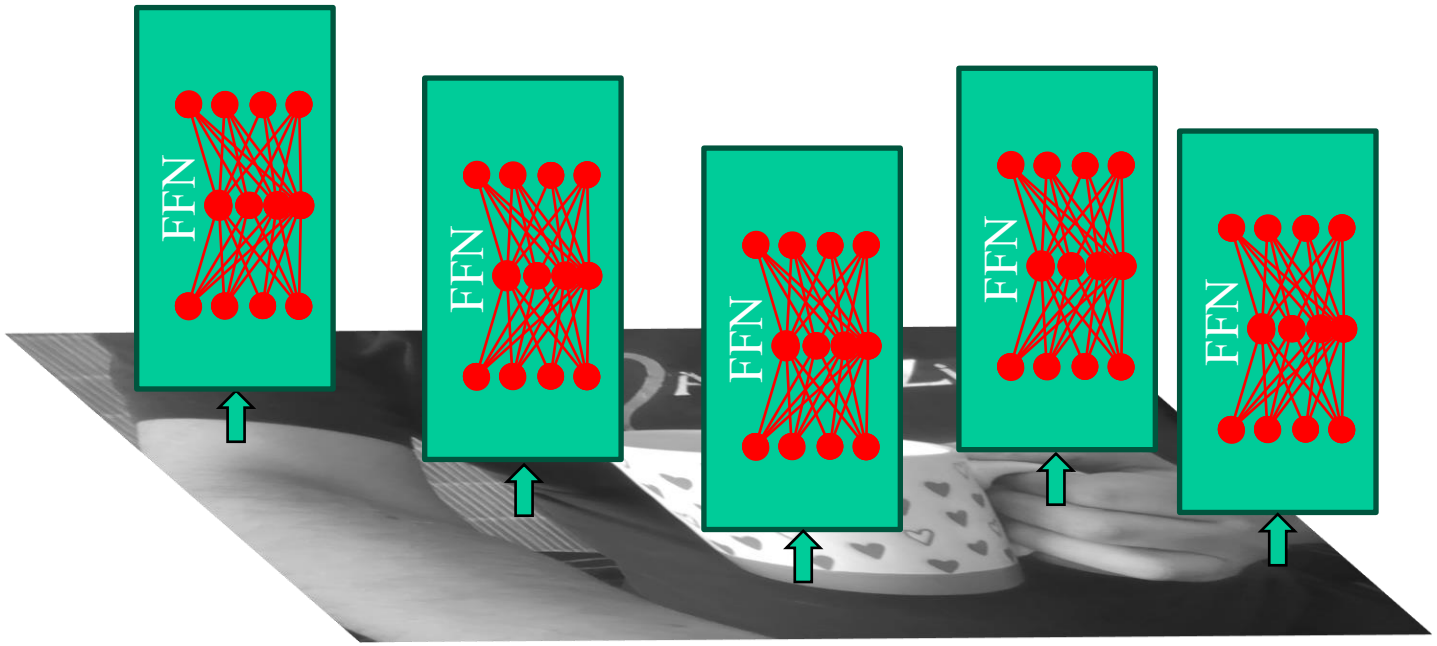
Andrej Lúčny

Autoencoder. Feature vectors, latent space. Encoder: Image Classifier. Decoder: Image Generator. Backbones transforming images to feature maps (VGG, ResNet).

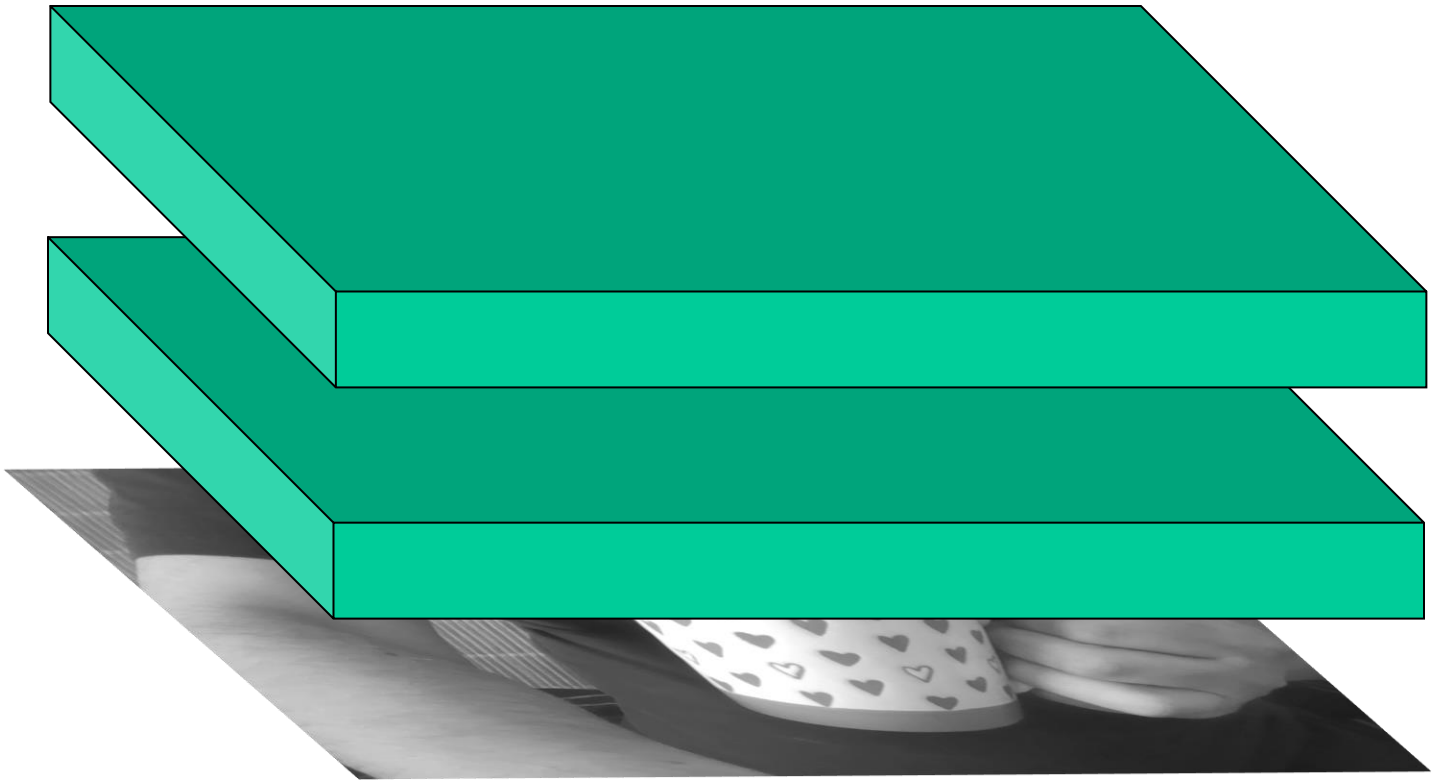
`https://github.com/andylucny/OAI`



Although the Perceptron (Feed Forward Network) is a universal approximator, in practice it is not directly applicable to image processing



However, we can process the image with smaller, parallel and locally operating perceptrons that share weights (1×1 kernels) and possibly also cooperate with neighbors (3×3 kernels)...



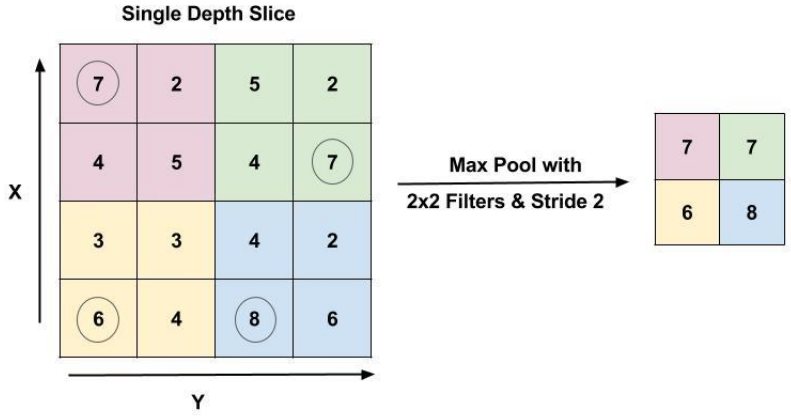
... which corresponds to two convolutional layer blocks arranged one after the other, which convert the color channels of the image into features and those into other features

Dimension reduction

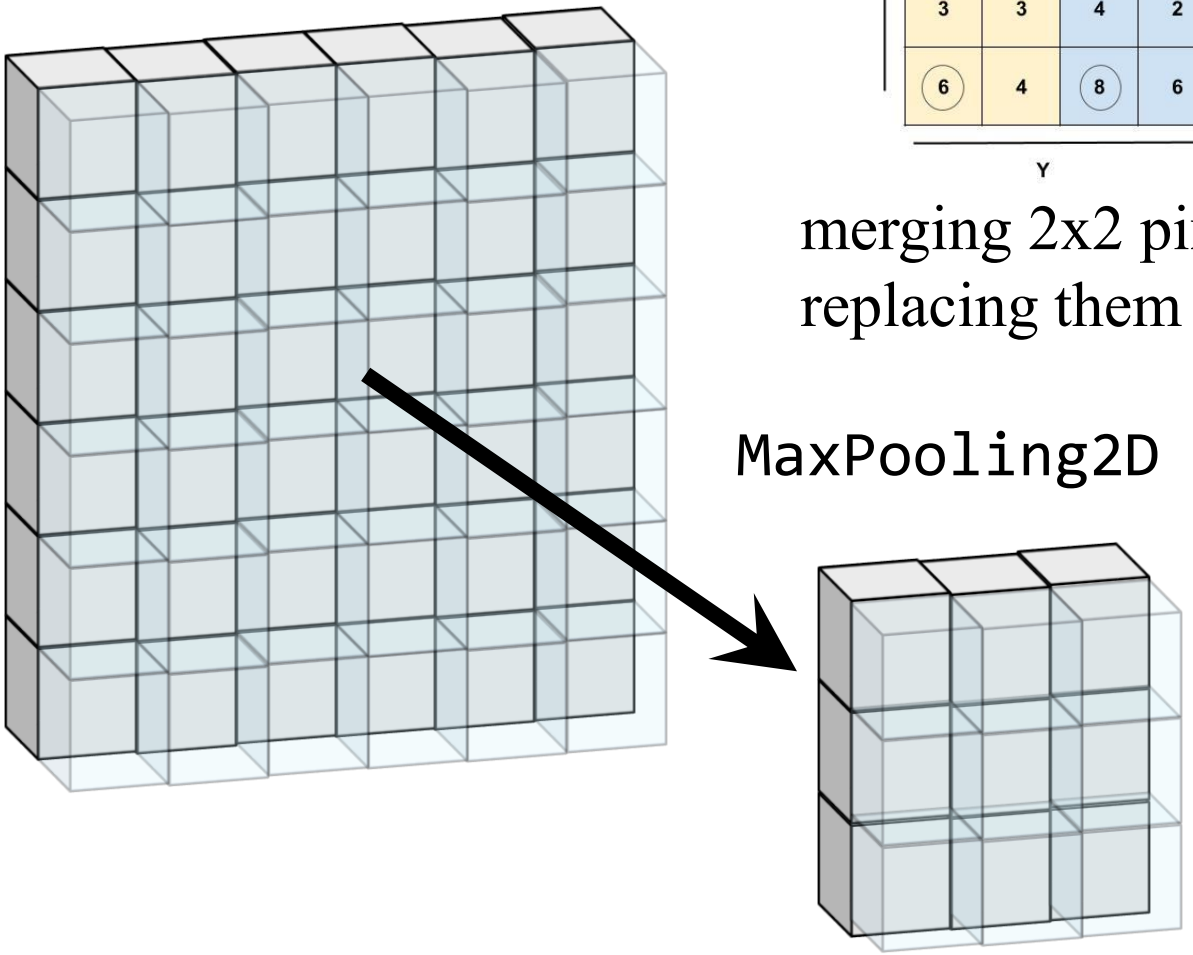
- Our plan was to process the image (which is a high-dimensional input) into something much smaller in dimension, but without losing information value

- Once it's small enough, the perceptron can handle it.

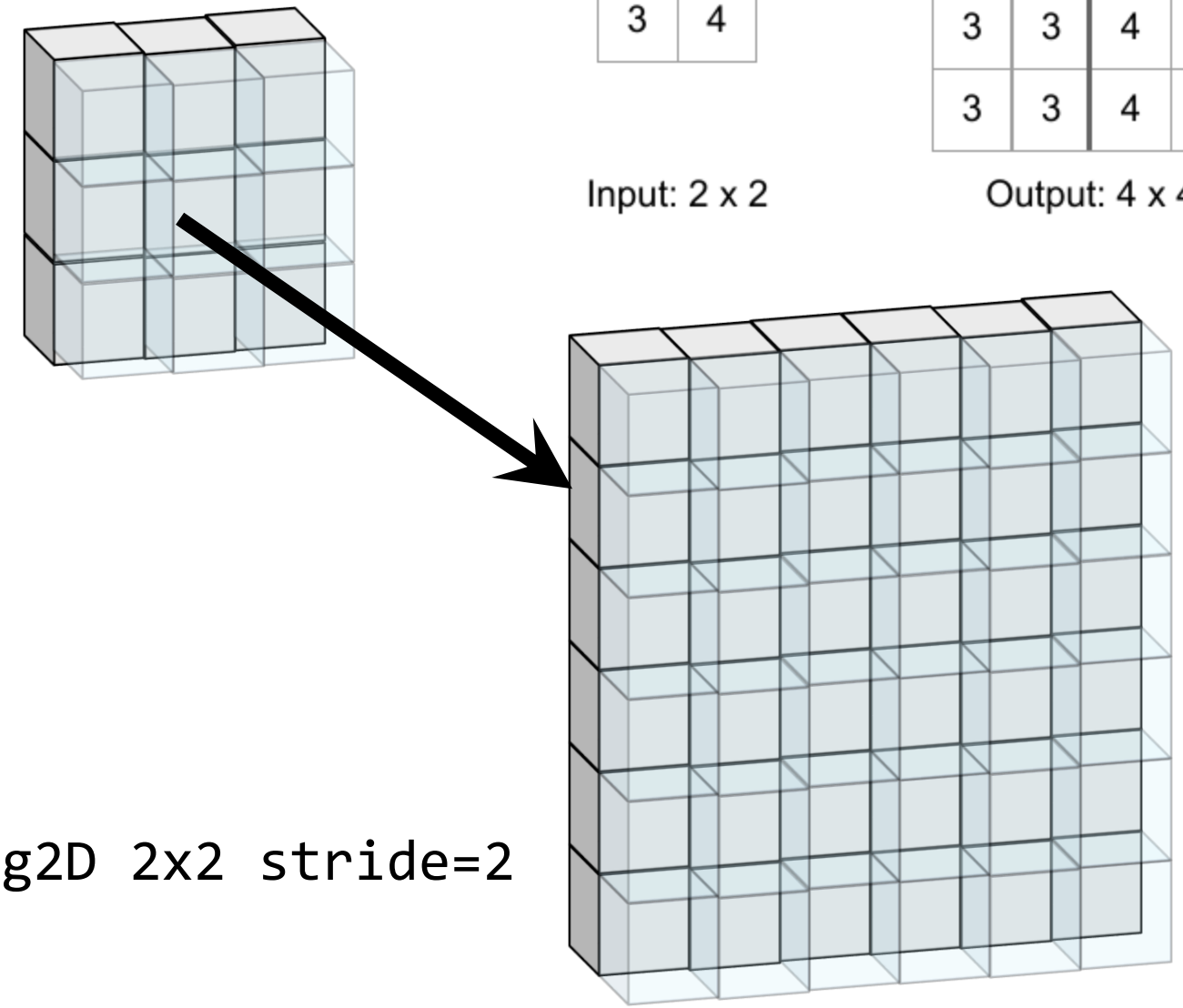
Dimension reduction via MaxPool



merging 2x2 pixels with step 2 and replacing them with the maximum



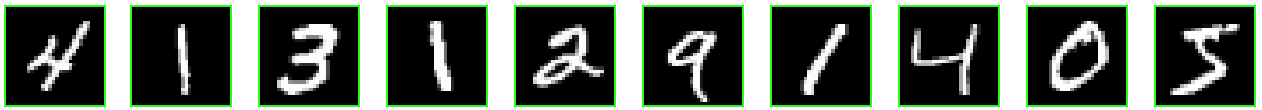
Dimension expansion



`UpSampling2D 2x2 stride=2`

Autoencoder

We use a dataset of unannotated images

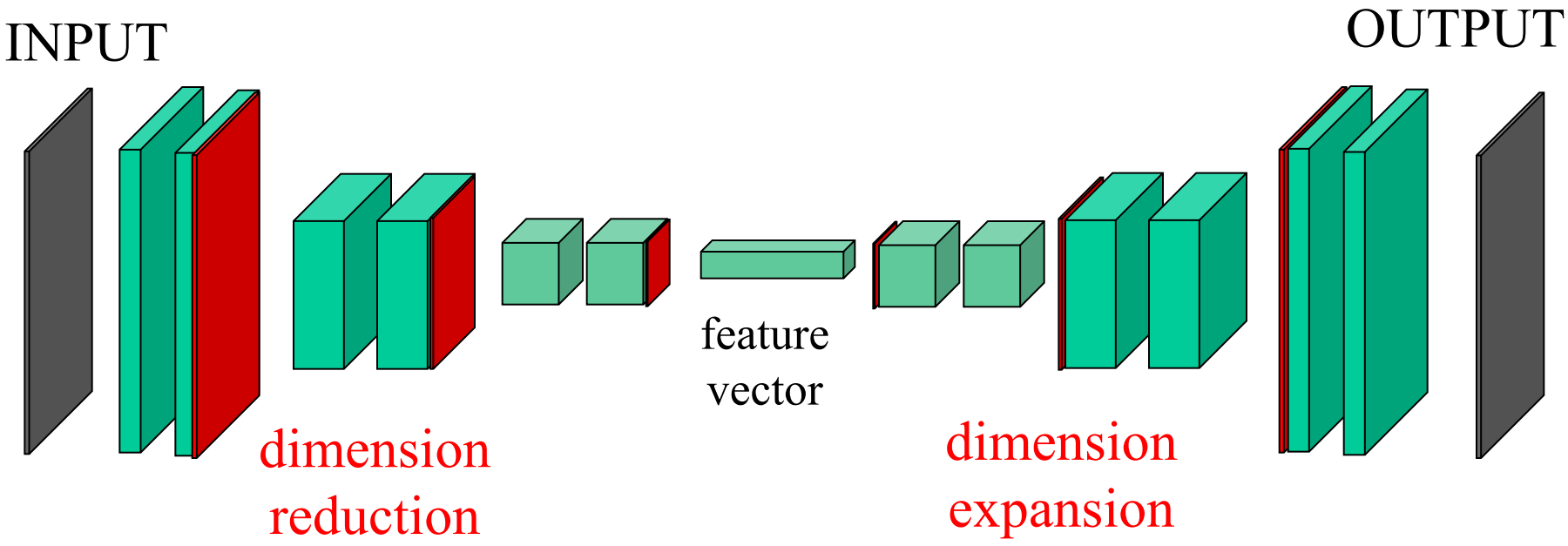


60000 images 28×28

We transform images into low-dimensional features

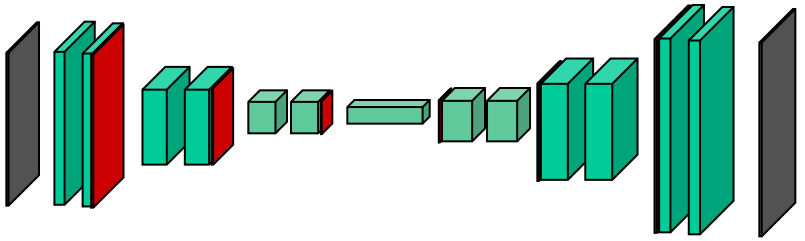
Good features are those that we can not only encode (extract) but also decode (generate the original image from).

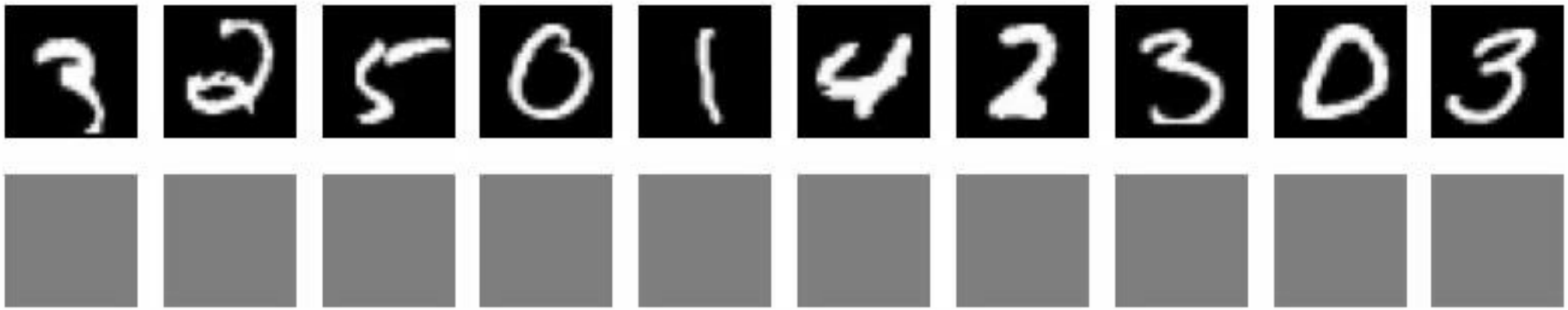
Convolutional autoencoder



We aim:
INPUT = OUTPUT

Autoencoder

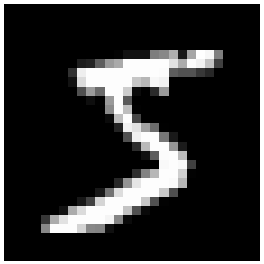




Autoencoder

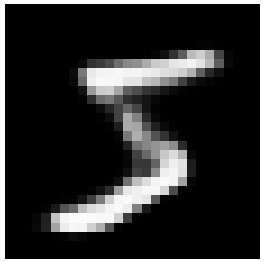
LATENT SPACE

INPUT




```
tensor([0.5128, 0.4216, 0.4599, 0.5000, 0.4729, 0.3759, 0.4845, 0.4800, 0.4545,
        0.4549, 0.3755, 0.3943, 0.4592, 0.4887, 0.3720, 0.4864, 0.4326, 0.3576,
        0.2825, 0.4351, 0.6167, 0.3217, 0.3943, 0.5935, 0.3106, 0.3975, 0.6174,
        0.5864, 0.4706, 0.3994, 0.6045, 0.4558, 0.3407, 0.5280, 0.4200, 0.3513,
        0.4281, 0.4669, 0.4538, 0.4297, 0.5300, 0.4694, 0.4335, 0.4635, 0.4034,
        0.5179, 0.3048, 0.2911, 0.5669, 0.3968, 0.2865, 0.5514, 0.3497, 0.3256,
        0.4478, 0.5840, 0.3743, 0.4684, 0.5981, 0.3716, 0.4457, 0.5995, 0.4222,
        0.3256, 0.3229, 0.5112, 0.3323, 0.3461, 0.4971, 0.3396, 0.3261, 0.4653])
```


OUTPUT

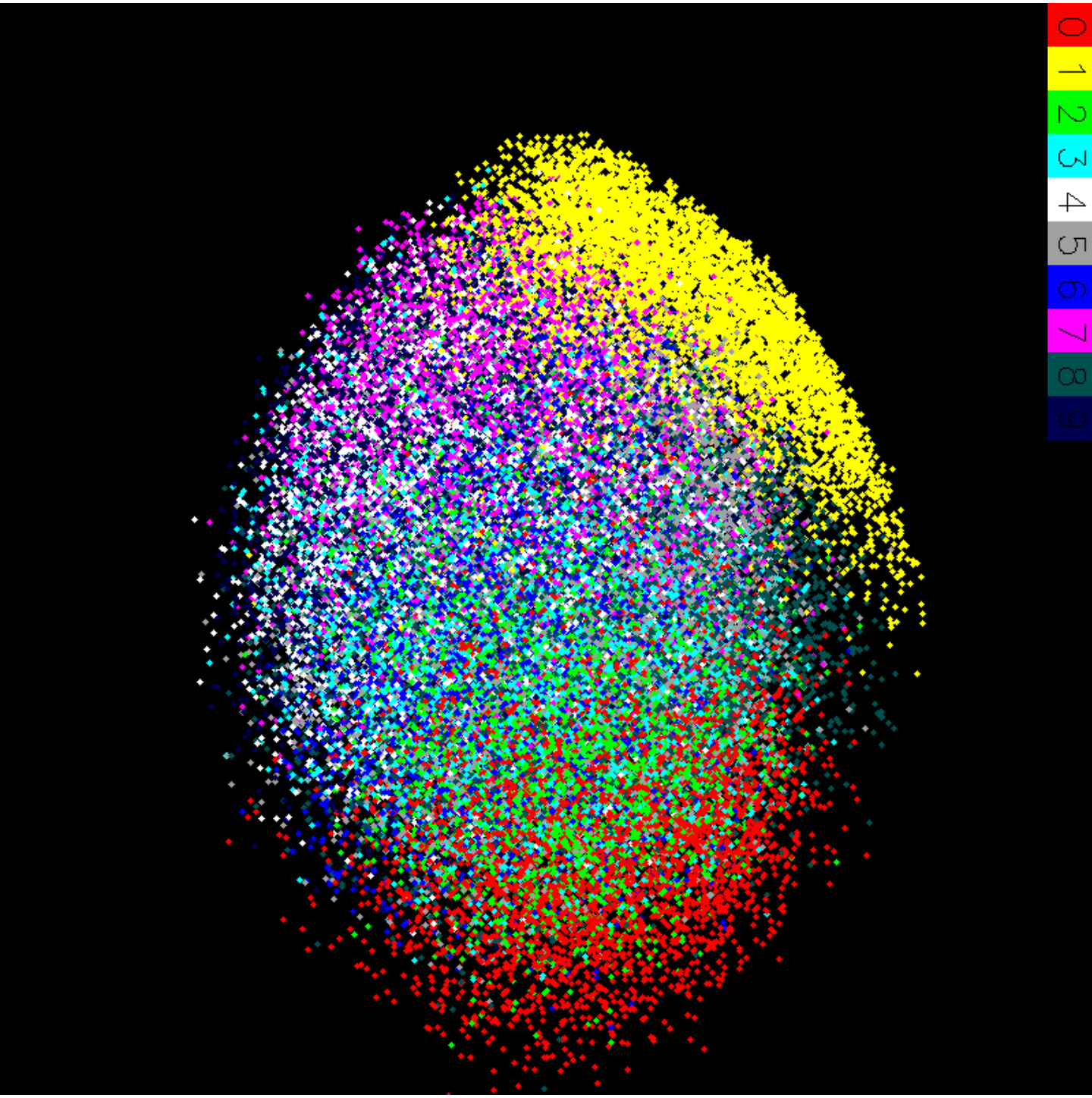


feature vector

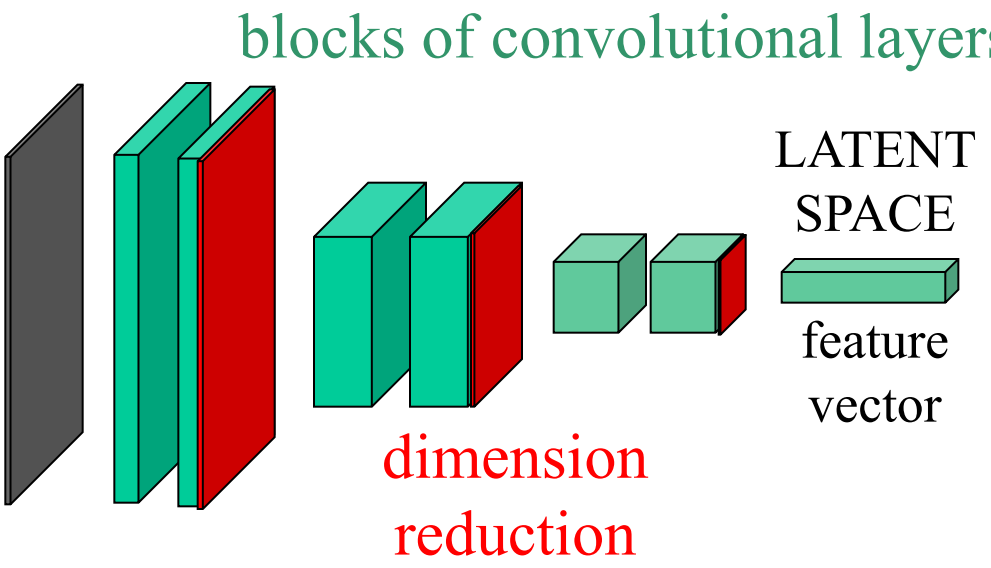
Fundamental properties of Features

- Feature vectors can be represented as points in latent space
- Although only a finite number of features correspond to samples from the dataset, all points in the latent space correspond to some instance of the data
- The latent space is not sparse and is uniformly continuous

LATENT SPACE

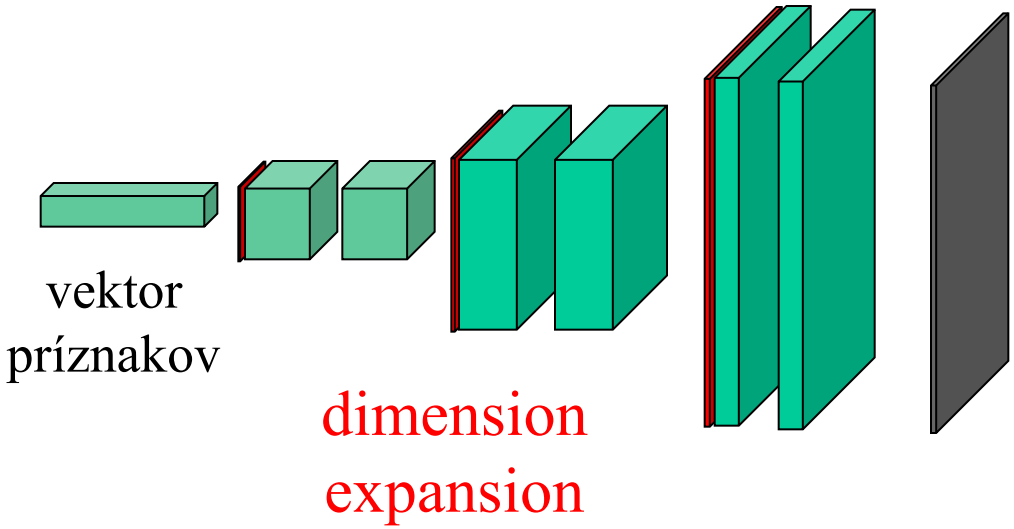


Encoder (Extractor)



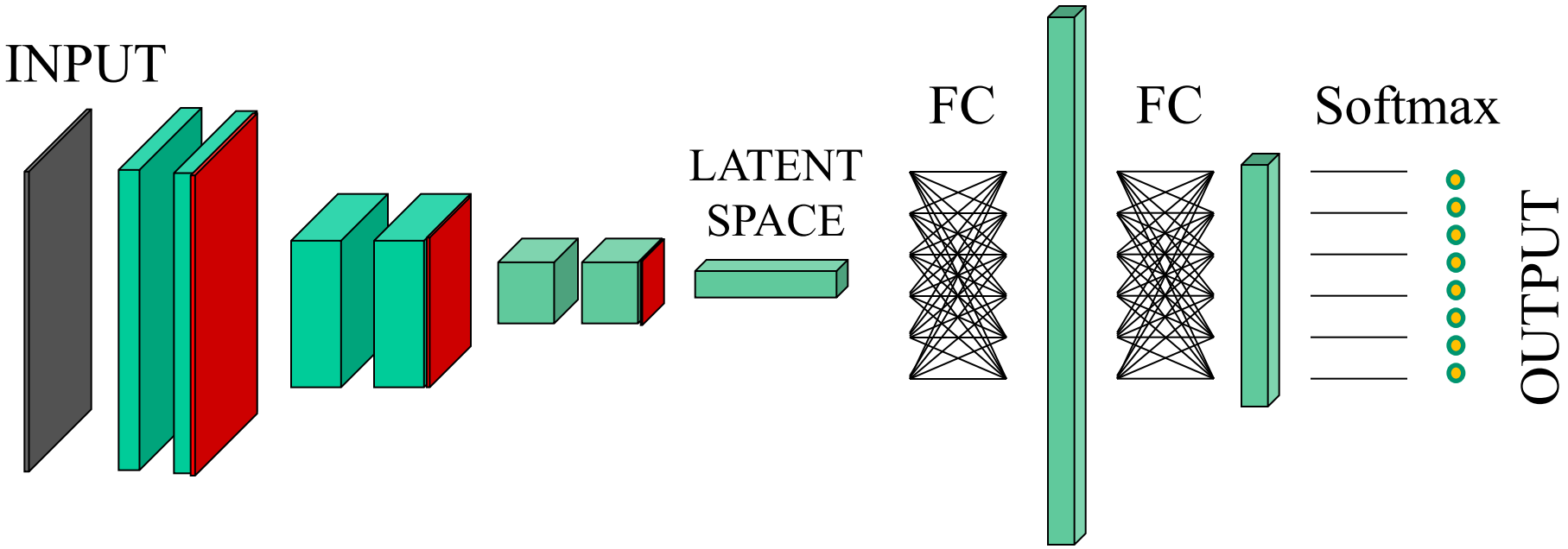
Encoder (the first half of the autoencoder) transforms image into features (a point in latent space)

Decoder (Generator)



Decoder (the second half of the autoencoder) transforms (any) features into an image

Classifier



logit

Softmax transforms whatever to probabilities

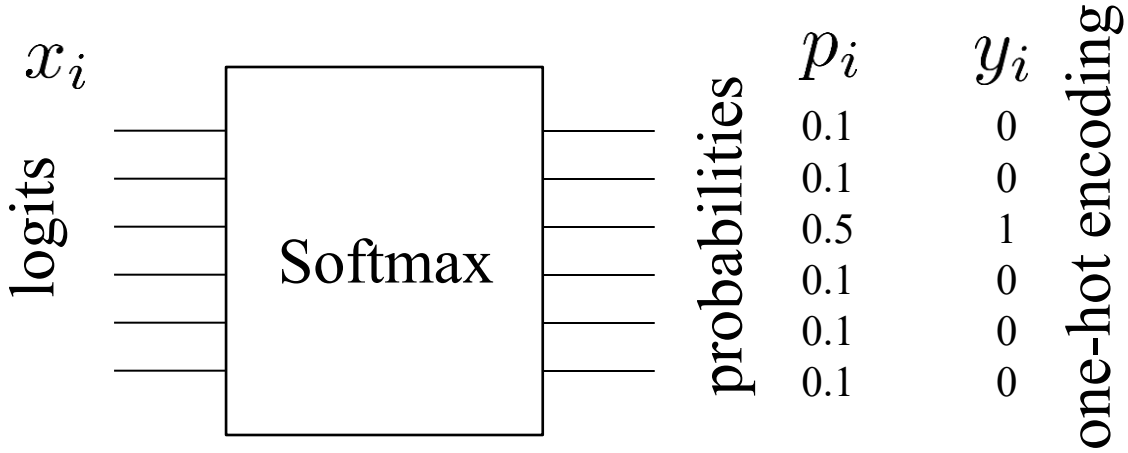
$$softmax_i(x) = \frac{e^{x_i}}{\sum_k e^{x_k}}$$

```
def softmax(x):
    e_x = np.exp(x - np.max(x))
    return e_x / e_x.sum(axis=0)
```

Softmax only makes sense when the categories are mutually exclusive. For example, if we have a wheel and a bicycle, we use Sigmoid

Cross Entropy Loss

$$loss(x, y) = -\sum_i y_i \log \frac{e^{x_i}}{\sum_j e^{x_j}}$$



gradients $\frac{\partial L}{\partial x_i} = \begin{cases} p_i & \text{when } y_i = 0 \\ p_i - 1 & \text{when } y_i = 1 \end{cases}$

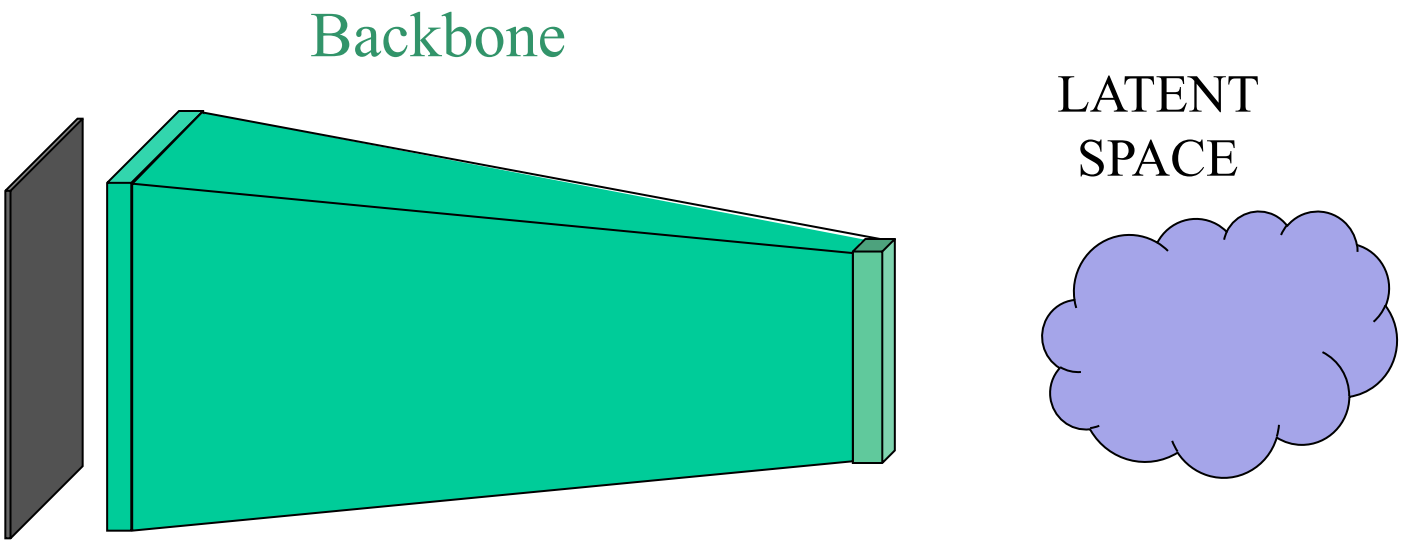
Important principle

- When we can justify how to make a DNN gradually and in parts
  (we make an autoencoder, cut off the decoder, connect a perceptron, ...),
  we can train it directly in its final form, all at once and as a whole
  (end-to-end system)

Backbones

- However, it may be faster to take a suitable ready-made feature extractor, train a perceptron for it, for example, and only then retrain the entire model (Fine tuning)
- Backbones: VGG, ResNet, MobileNet, ...
- ZOO: each backbone architecture has multiple pre-trained models available (torchvision)

Backbone

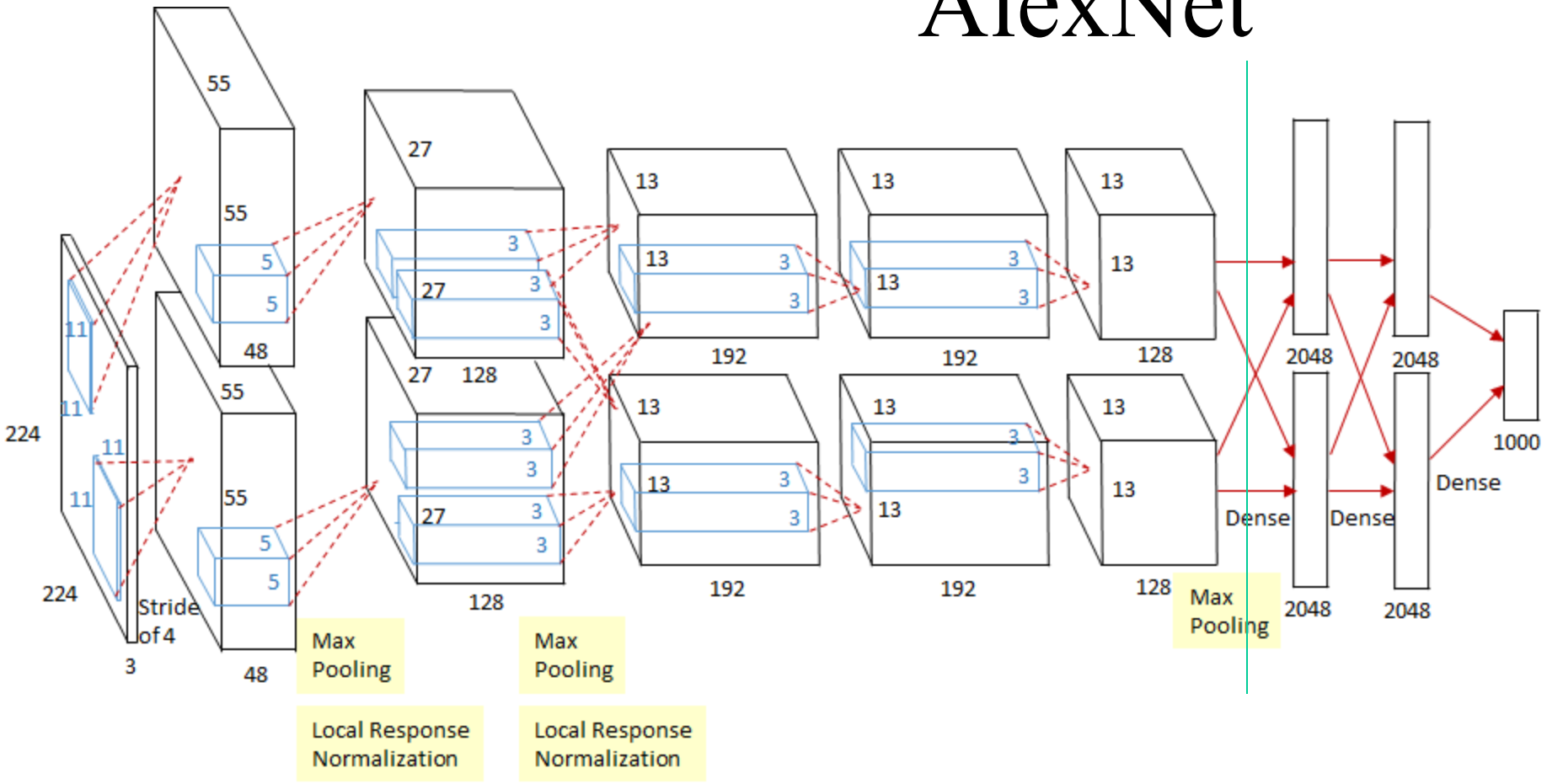


- transforms images into general-purpose features

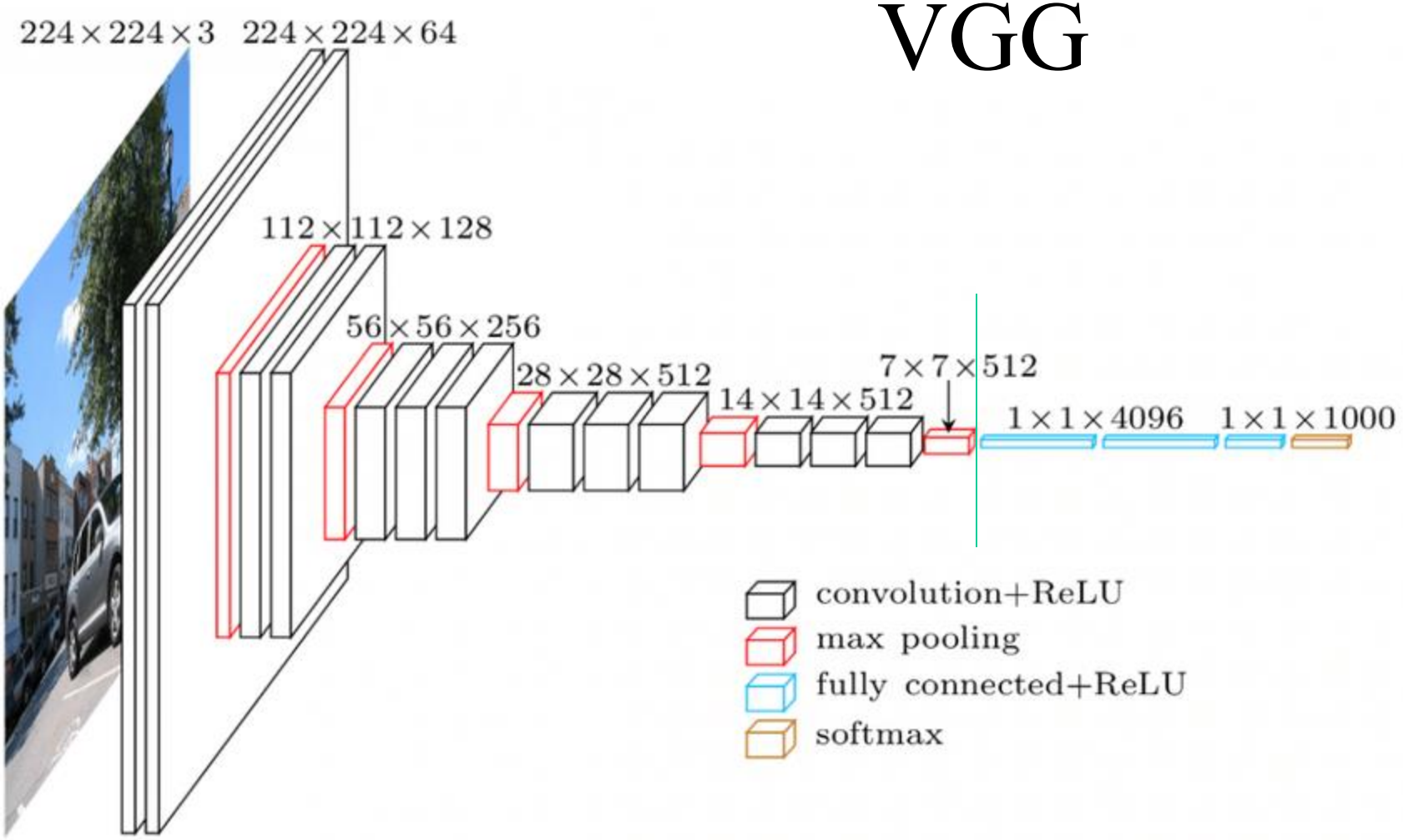
Backbones

- AlexNet
- VGG (VGG16, VGG19)
- ResNet (ResNet50, ResNet18)
- DarkNet
- Inception (Inception v3)
- Xception
- MobileNet

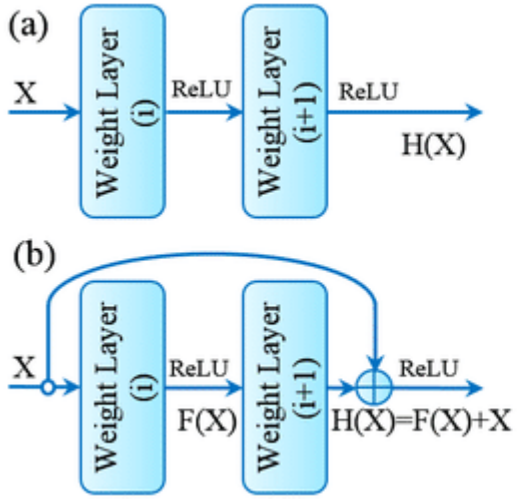
AlexNet

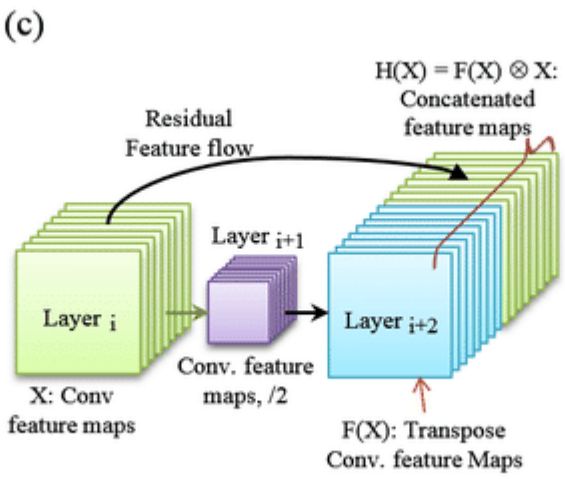

55
55
48
55
55
48
224
224
11
11
11
11
Stride of 4
3
27
27
128
27
27
128
3
5
13
13
192
192
128
2048
2048
1000
Dense
Dense
Dense
2048
2048
Max Pooling
Max Pooling
Local Response Normalization
Local Response Normalization

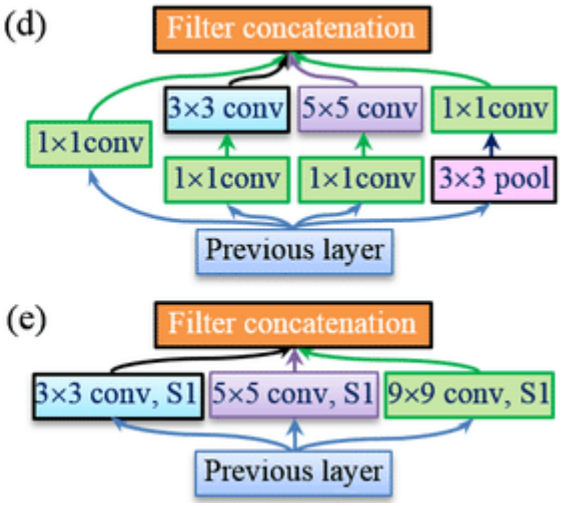
VGG

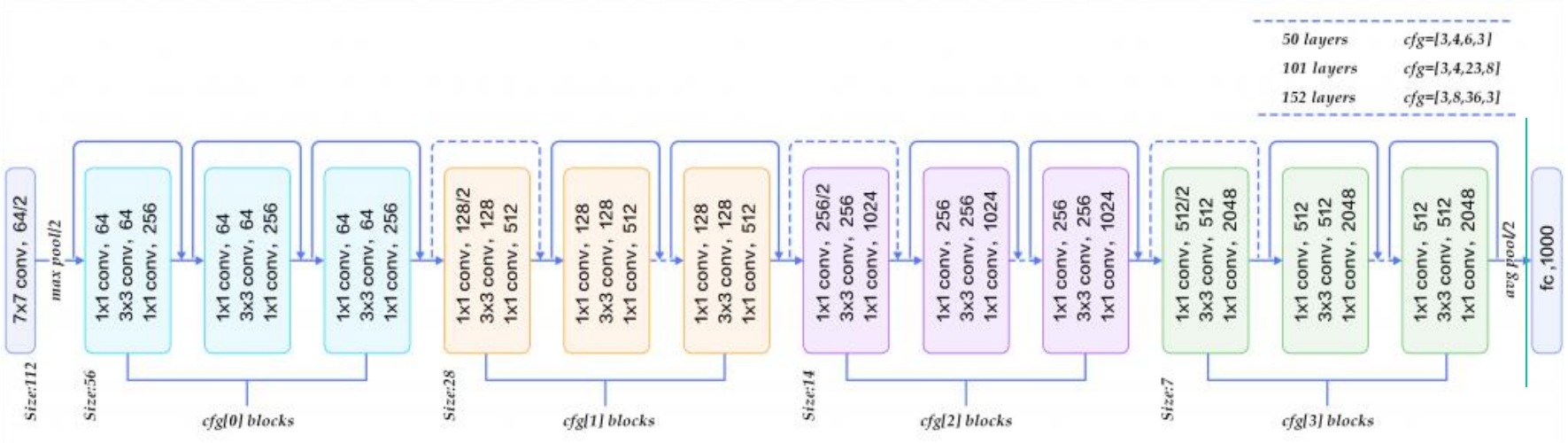

224 × 224 × 3
224 × 224 × 64
112 × 112 × 128
56 × 56 × 256
28 × 28 × 512
14 × 14 × 512
7 × 7 × 512
1 × 1 × 4096
1 × 1 × 1000
convolution+ReLU
max pooling
fully connected+ReLU
softmax

ResNet

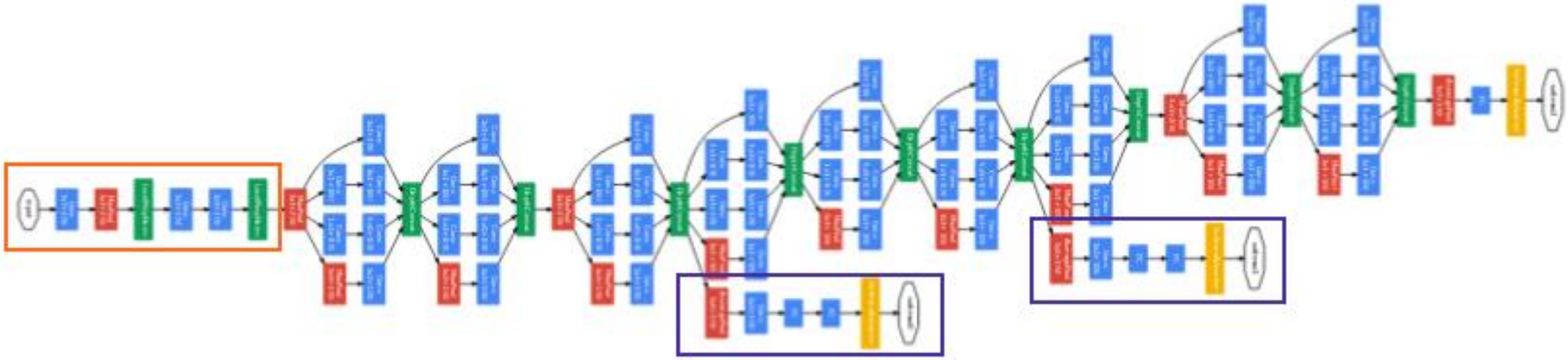

(a)
X
Weight Layer (i)
ReLU
Weight Layer (i+1)
ReLU
H(X)
(b)
X
Weight Layer (i)
ReLU
F(X)
Weight Layer (i+1)
ReLU
H(X)=F(X)+X


(c)
H(X) = F(X) ⊗ X:
Concatenated feature maps
Residual Feature flow
Layer i+1
Layer i
Layer i+2
X: Conv feature maps
Conv. feature maps, /2
F(X): Transpose Conv. feature Maps

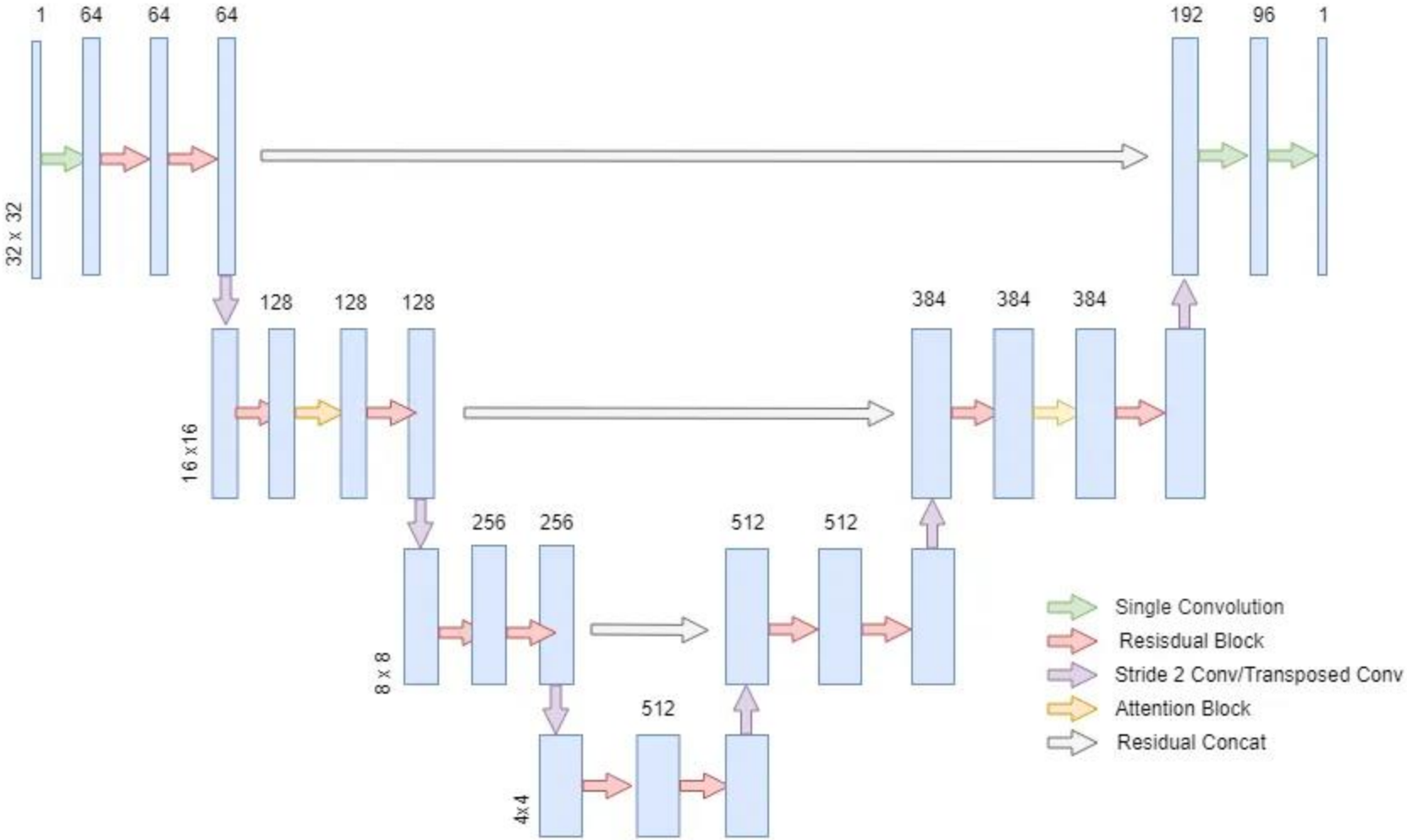

(d)
Filter concatenation
3×3 conv
5×5 conv
1×1conv
1×1conv
1×1conv
1×1conv
3×3 pool
Previous layer
(e)
Filter concatenation
3×3 conv, S1
5×5 conv, S1
9×9 conv, S1
Previous layer


50 layers cfg=[3,4,6,3]
101 layers cfg=[3,4,23,8]
152 layers cfg=[3,8,36,3]
7x7 conv, 64/2
max pool/2
1x1 conv, 64
3x3 conv, 64
1x1 conv, 256
1x1 conv, 128/2
3x3 conv, 128
1x1 conv, 512
1x1 conv, 256/2
3x3 conv, 256
1x1 conv, 1024
1x1 conv, 512/2
3x3 conv, 512
1x1 conv, 2048
avg pool/2
fc ,1000
Size:112
Size:56
Size:28
Size:14
Size:7
cfg[0] blocks
cfg[1] blocks
cfg[2] blocks
cfg[3] blocks

Inception V1



U-Net


1
64
64
64
192
96
1
32 x 32
128
128
128
384
384
384
16 x16
256
256
512
512
8 x 8
512
4x4
Single Convolution
Resisdual Block
Stride 2 Conv/Transposed Conv
Attention Block
Residual Concat

Pretrained model

- Each backbone also has its own Zoo – a collection of models trained on various datasets, e.g. COCO, PascalVOC ...
- Training a network from a pre-trained model, behind which we include our (custom) output layers, is called **transfer learning**

Transfer learning

- It is likely that the model for classifying a car will be very similar to the model for a bicycle
- Therefore, when creating the second one, we can start from the first one rather than from scratch
- From the model trained for a certain dataset, for example COCO or VOC, we will keep only the backbone and distribute it as a pre-trained model

Fine tuning

- We connect a perceptron to the pre-trained model and train only the weights of the perceptron to obtain, for example, a custom classifier

- Such training is also much faster. However, it relies only on ready-made (general-purpose) features

- Therefore, it makes sense to try to continue training, while also allowing the backbone weights to be changed. This phase of training is called Fine tuning