Image recognition

deep learning 2

Raoul Grouls, 26 November 2024

Recap

Data

$$X = \{\overrightarrow{x_1}, \ldots, \overrightarrow{x_n} \,|\, \vec{x} \in \mathbb{R}^d\}$$

$$y = \{y_1, \ldots, y_n \,|\, y \in \{0,1\}\}$$

Trainable Weights W, b

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

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

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

Optimize $w \leftarrow w - \eta \dfrac{\partial Loss}{\partial W}$

Recap

$$X \rightarrow f_1 \circ \sigma \circ \ \ldots \ \rightarrow Loss(y, \hat{y})$$

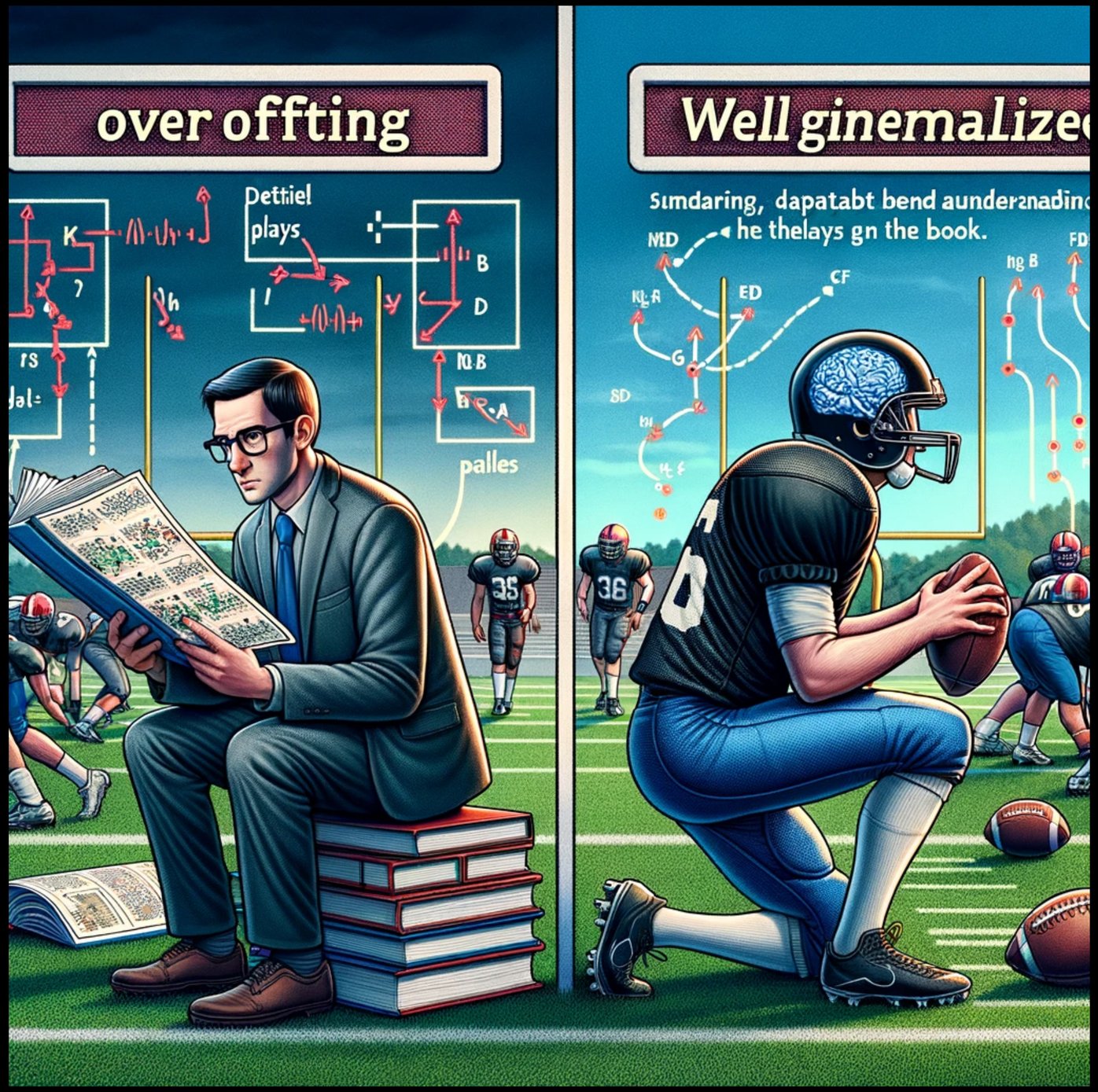
$$\leftarrow \quad w \leftarrow w - \eta \frac{\partial Loss}{\partial W} \quad \leftarrow$$

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.



Regularization

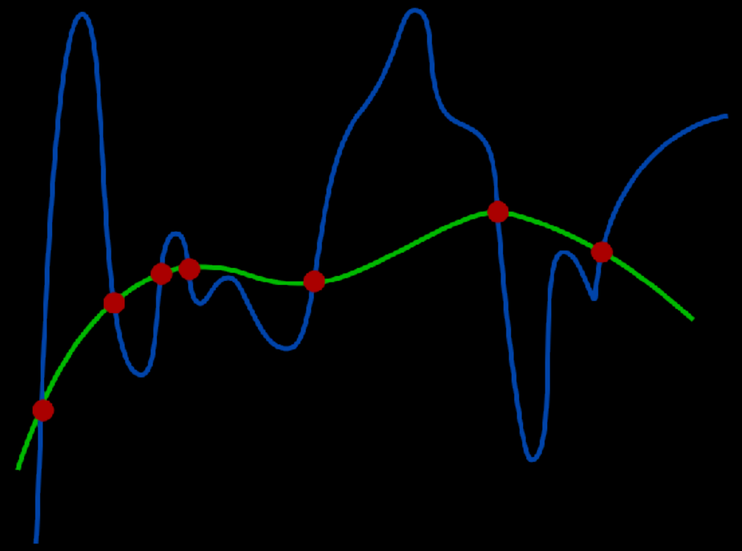
$$J(y, \hat{y} \mid W) = Loss(y, \hat{y}) + \eta \cdot Penalty(W)$$

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.

```python
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:

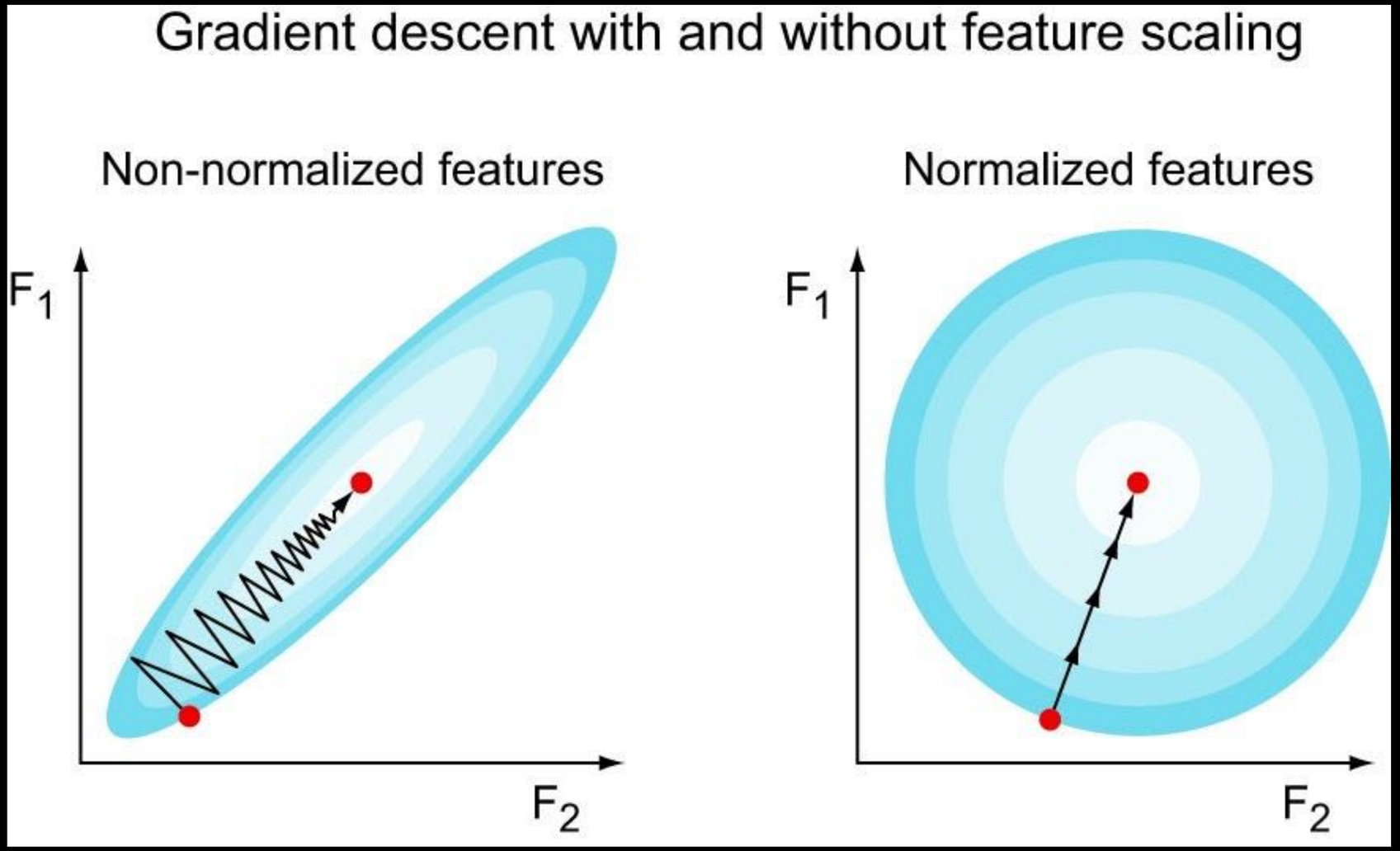
- All methods that are not directly added to the optimization problem.
- Examples include early stopping, introducing noise (eg with random batches, dropout), and discarding outliers.
- Ubiquitous in machine learning, including *batched stochastic gradient descent* and *ensemble methods* like random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

```python
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

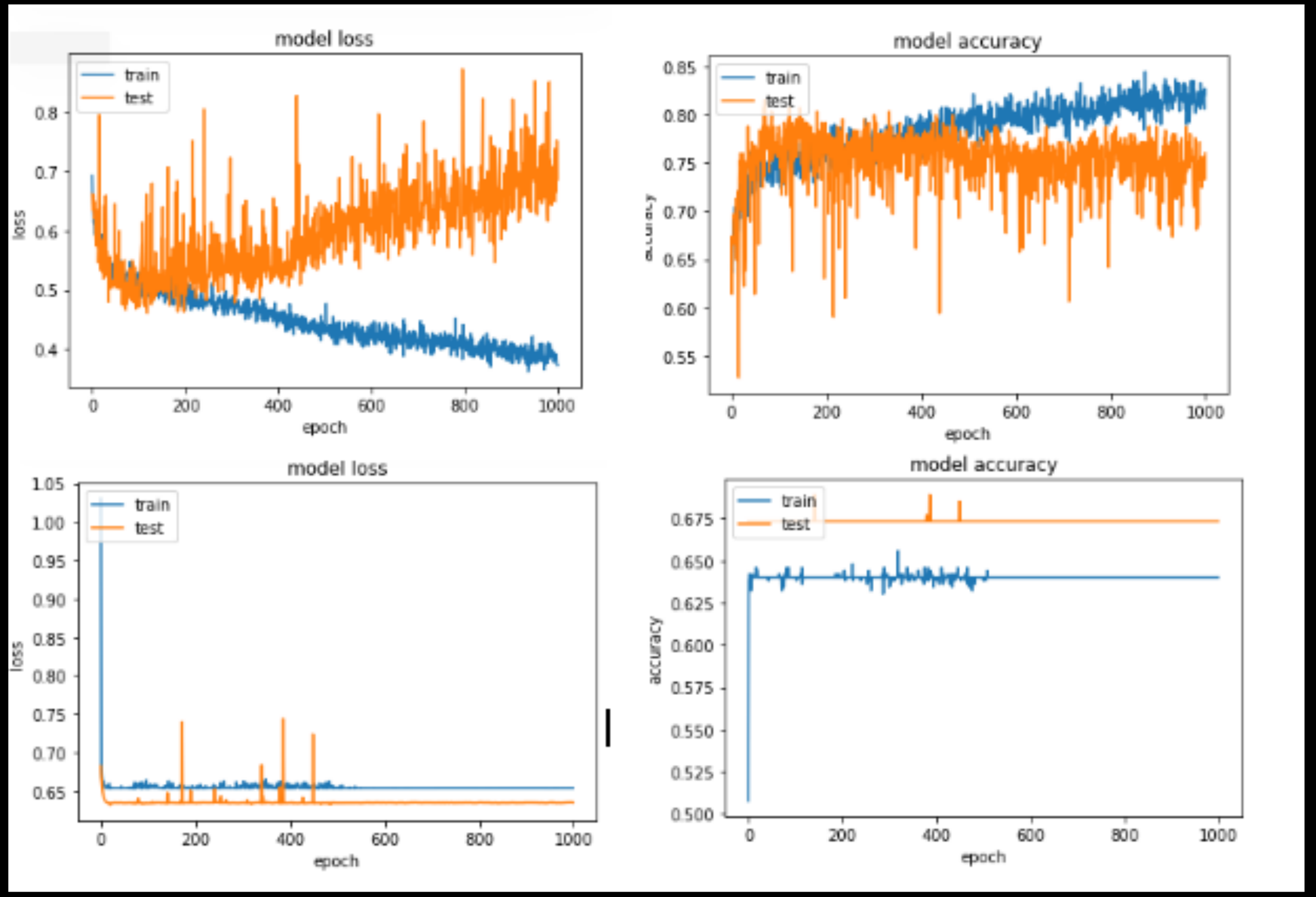
Normalization



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.

Spotting overfitting & underfitting



Dropout

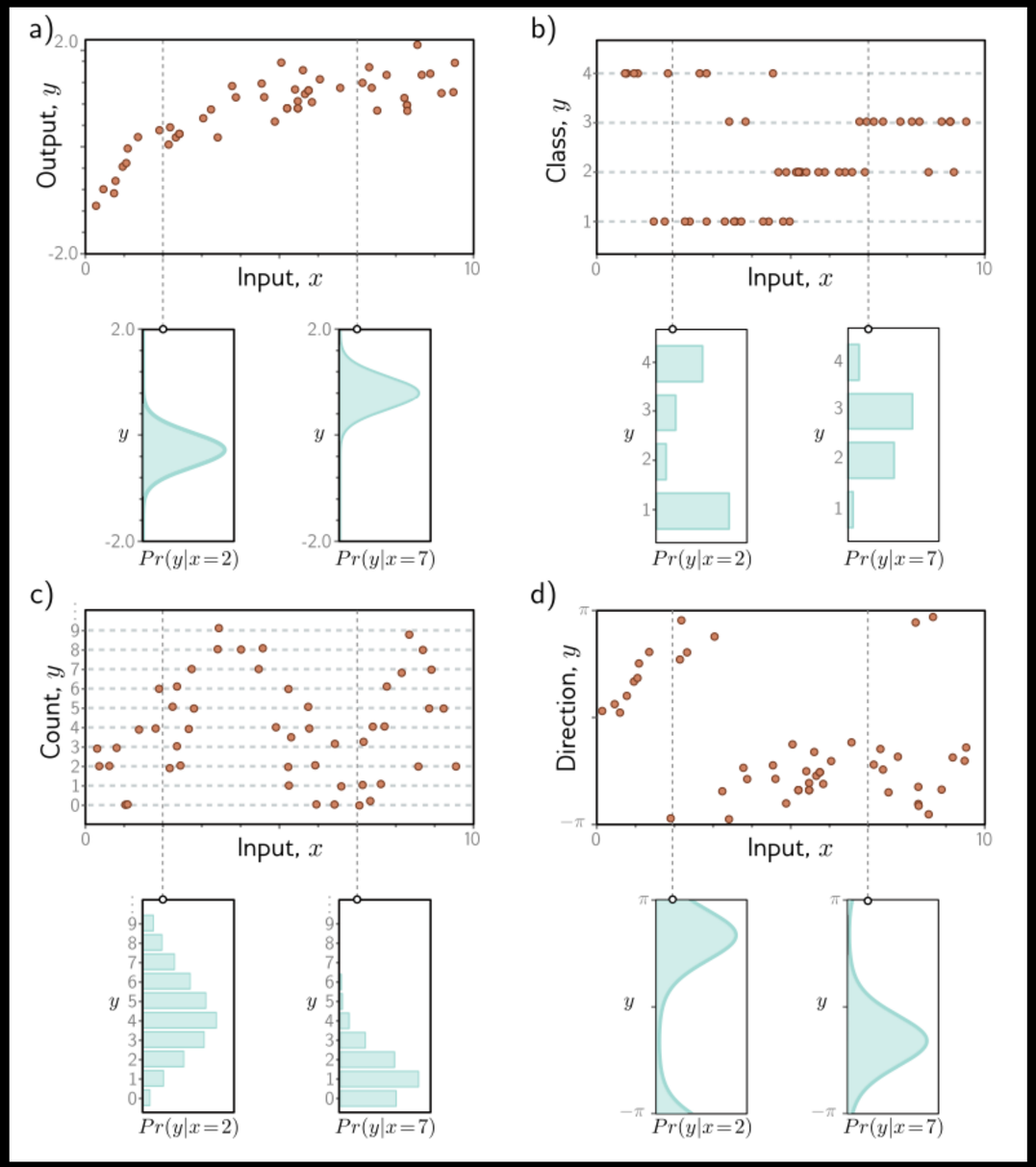
- `torch.nn.Dropout(p=0.1)`
- Randomly drop weights in order to avoid overspecialised parameters
- $\hat{w}_j = \begin{cases} w_j & \text{with } p(\theta) \\ 0 & \text{otherwise} \end{cases}$
- If you are juggling with a group and everyone might drop a ball at random, you will be more alert


OVEATAICE SOTSION
ABSIABENCE DICCE
ABSSENT COLLUBBE

Losses

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

Predicting outputs



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.

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

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

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

Convolutions

Convolutions - the motivation

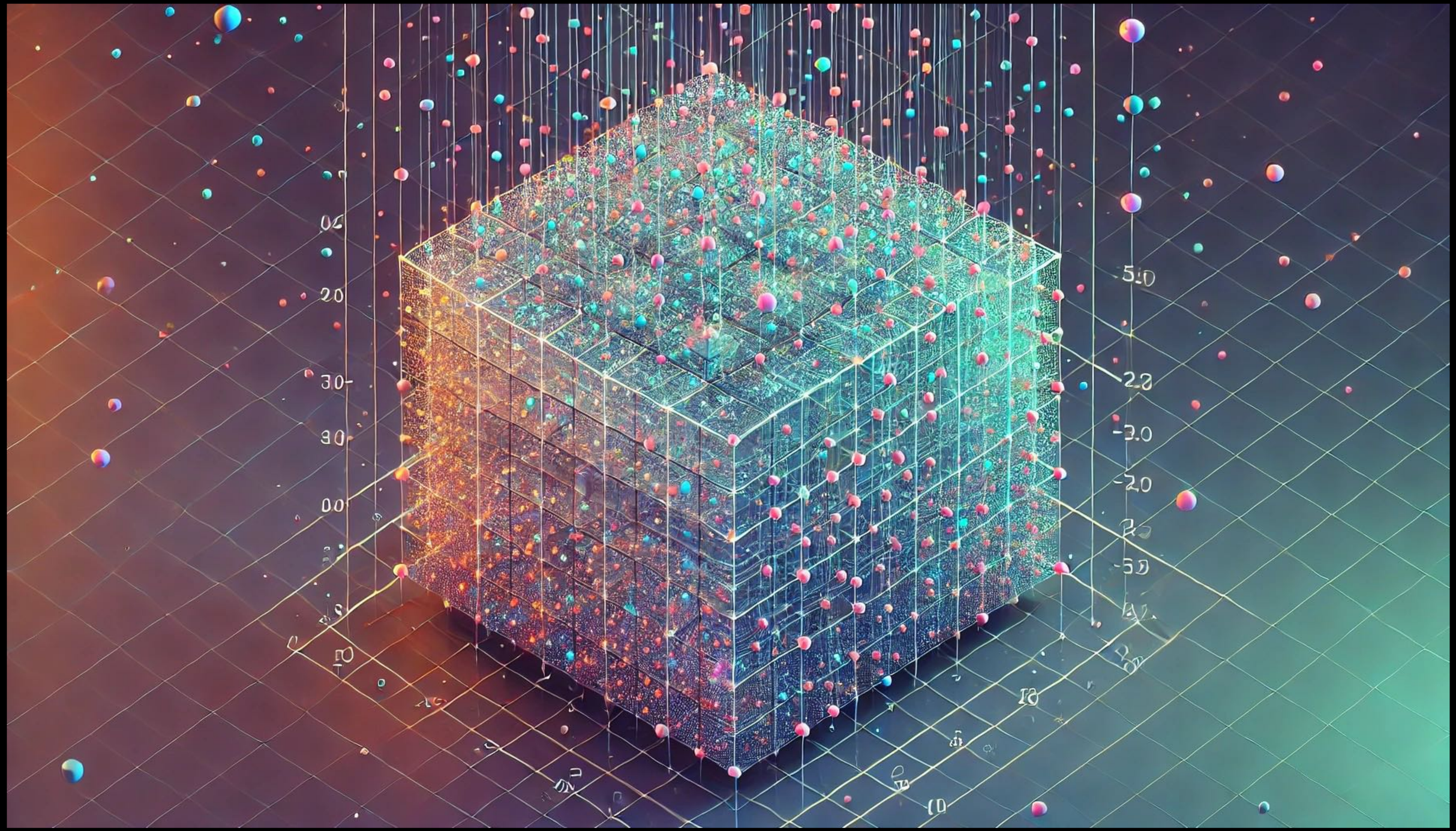
An image of size 28x28 has 784 pixels

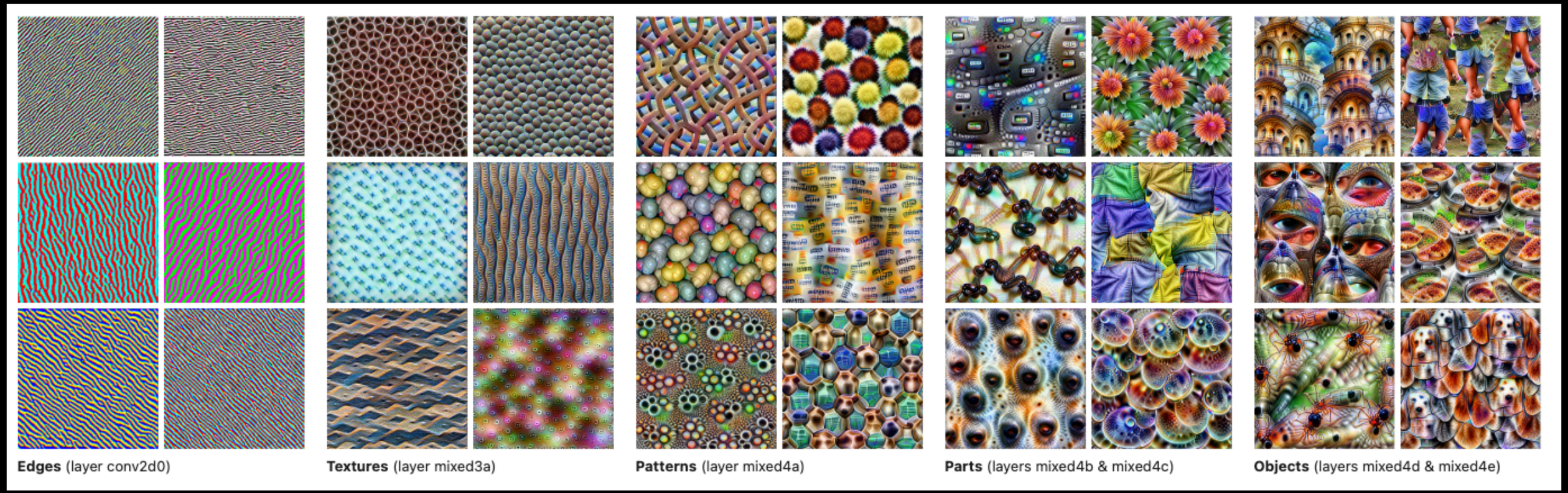
With 256x256 you are already at 65536 pixels

Treating every pixels as a feature will blow up the amount of parameters for your model:

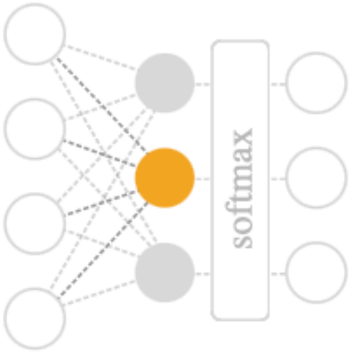
Assuming you halve the dimension, a batch with dimensions (32, 65536) will need a (65536, 32000) weight matrix.

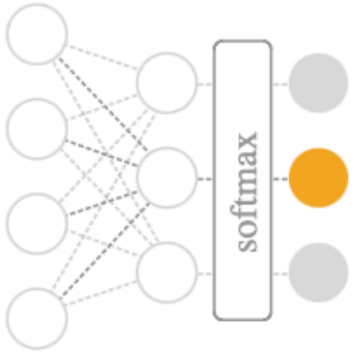
That are over 2×10^9 parameters, just for the first layer…





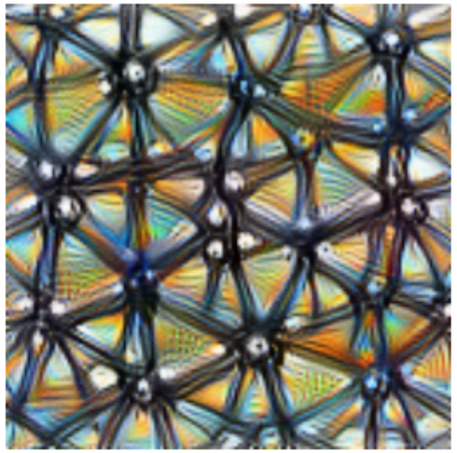
https://distill.pub/2017/feature-visualization/





Neuron

`layer`$_n$`[x,y,z]`



Channel

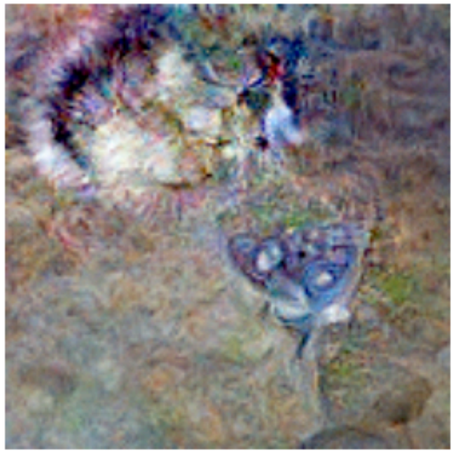
`layer`$_n$`[:,:,z]`

Layer/DeepDream

`layer`$_n$`[:,:,:]`^{2}

Class Logits

`pre_softmax[k]`

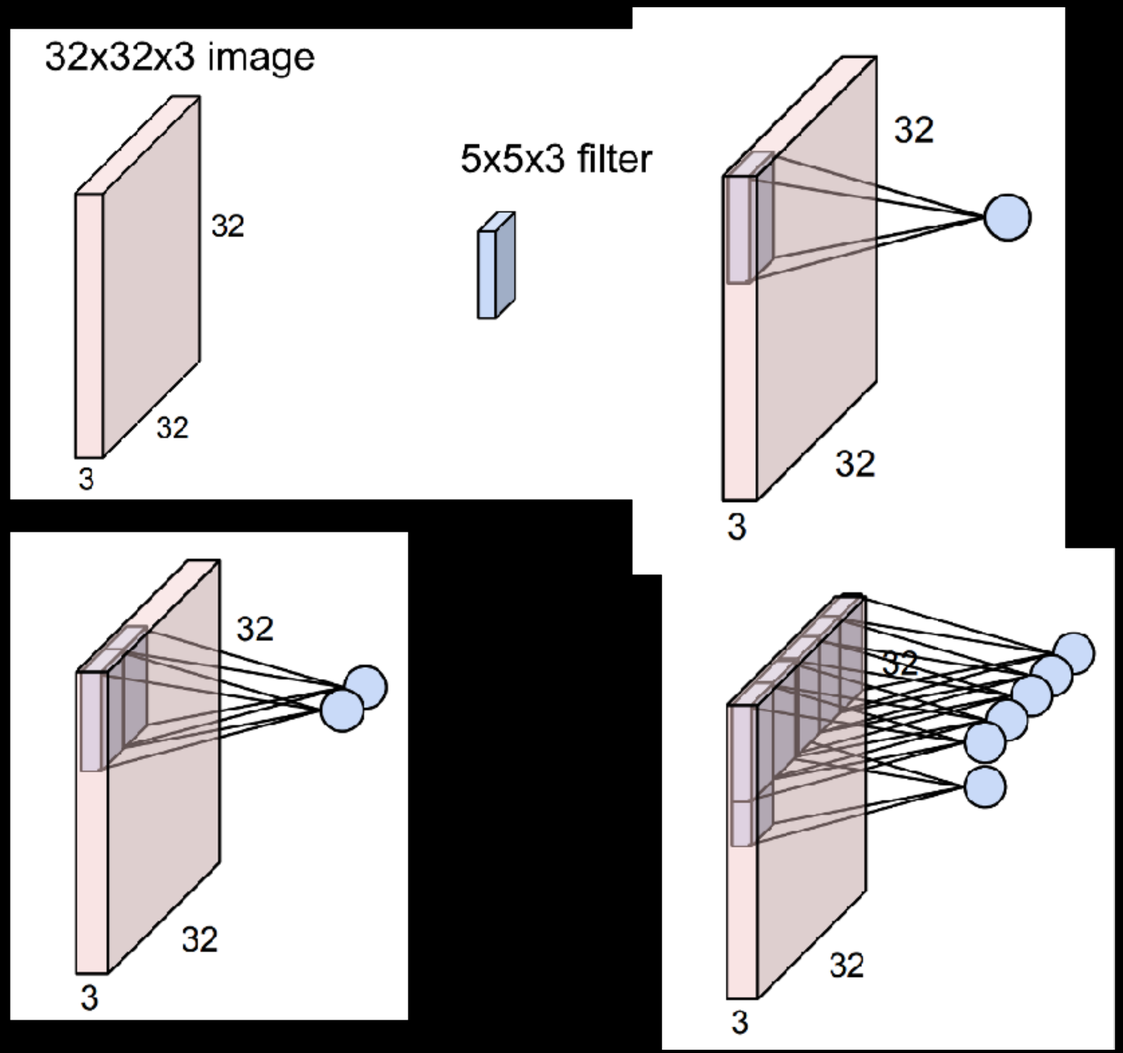


Class Probability

`softmax[k]`

Convolutions

- Take a filter of size (5x5x3)
- This needs just 75 parameters
- Slide it over the image, like a scanner

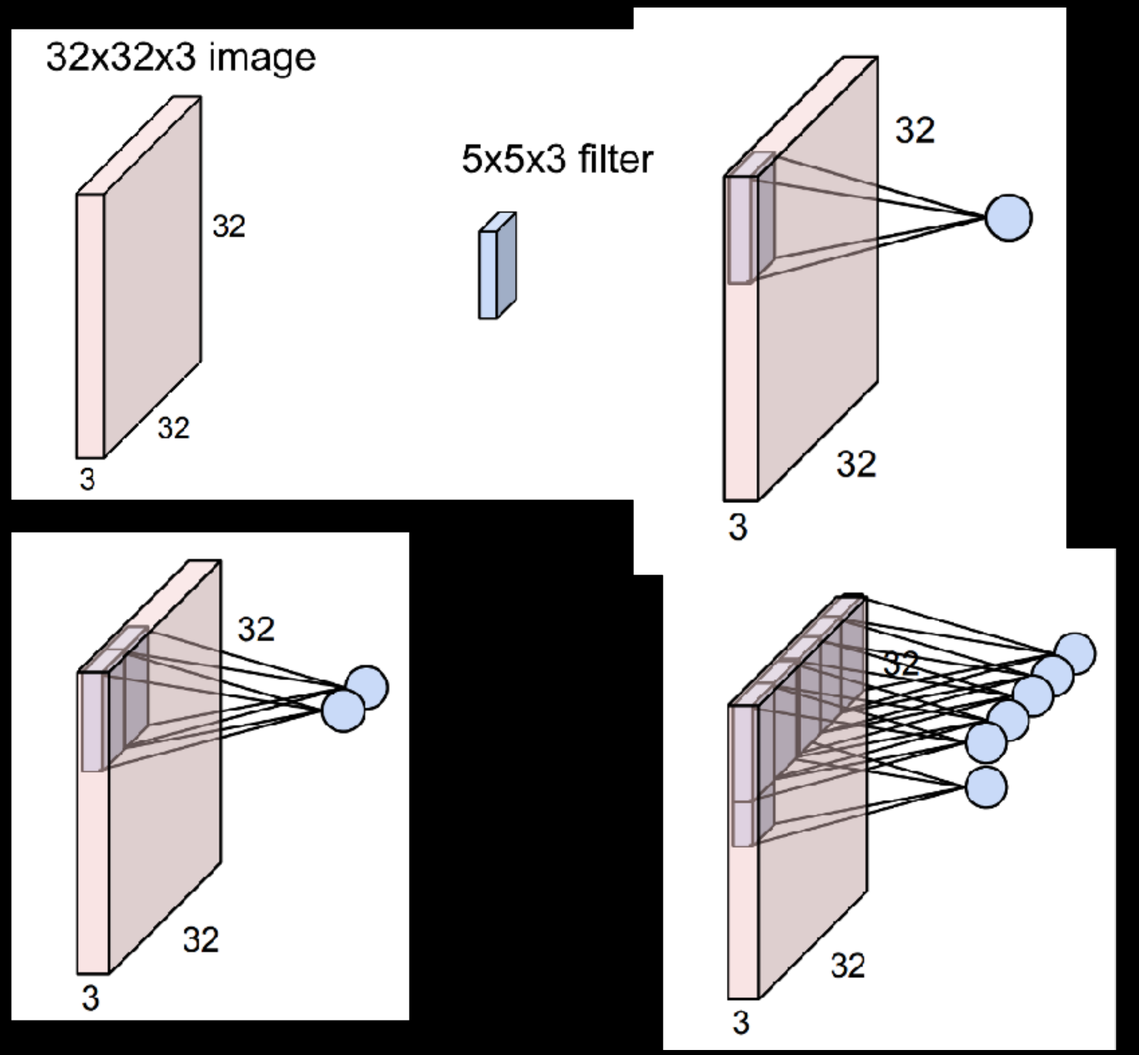


Convolutions

- Calculate the dot product between the filter and the image
- E.g.,

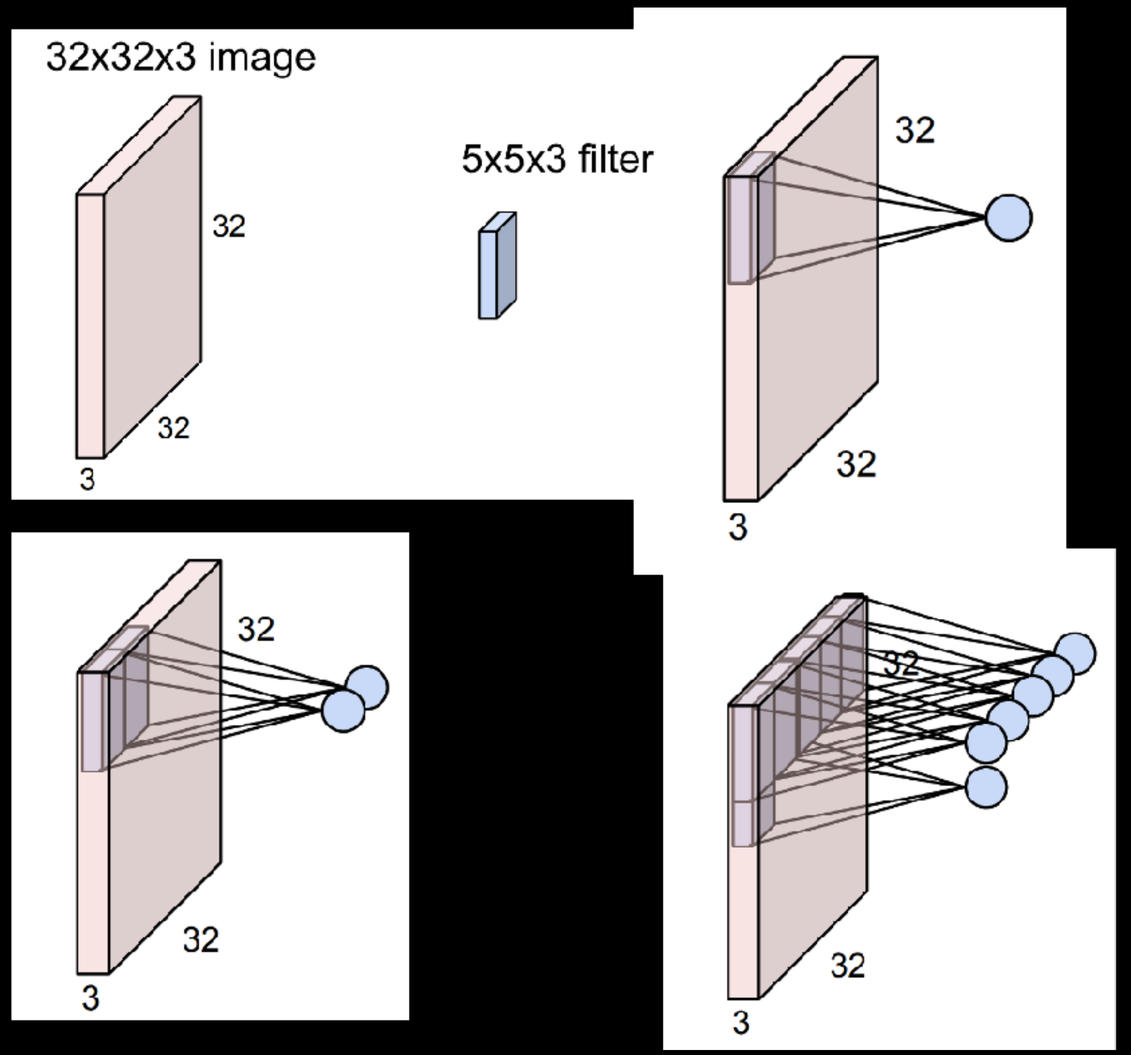
$$\begin{bmatrix} 1 & 10 \\ 2 & 20 \end{bmatrix} \cdot \begin{bmatrix} 1 & 2 \\ 2 & 0 \end{bmatrix} =$$

$$(1 * 1) + (10 * 2) + (2 * 2) + 0 = 25$$



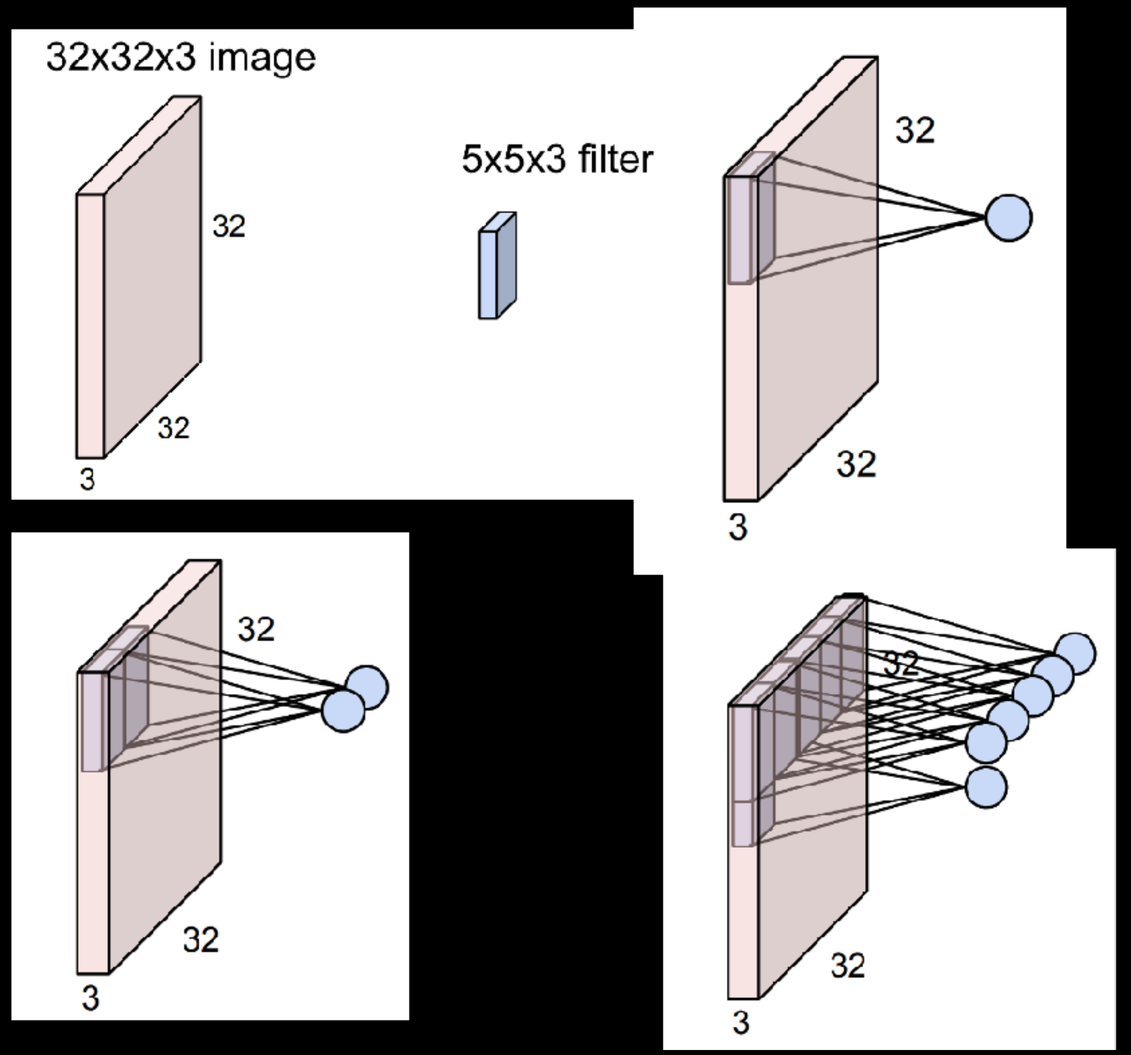
Convolutions

- So, with a 5x5 filter, every 5x5 image slice is reduced to a single number
- This number contains weighted information about it's neighborhood
- We call the result an activation map.



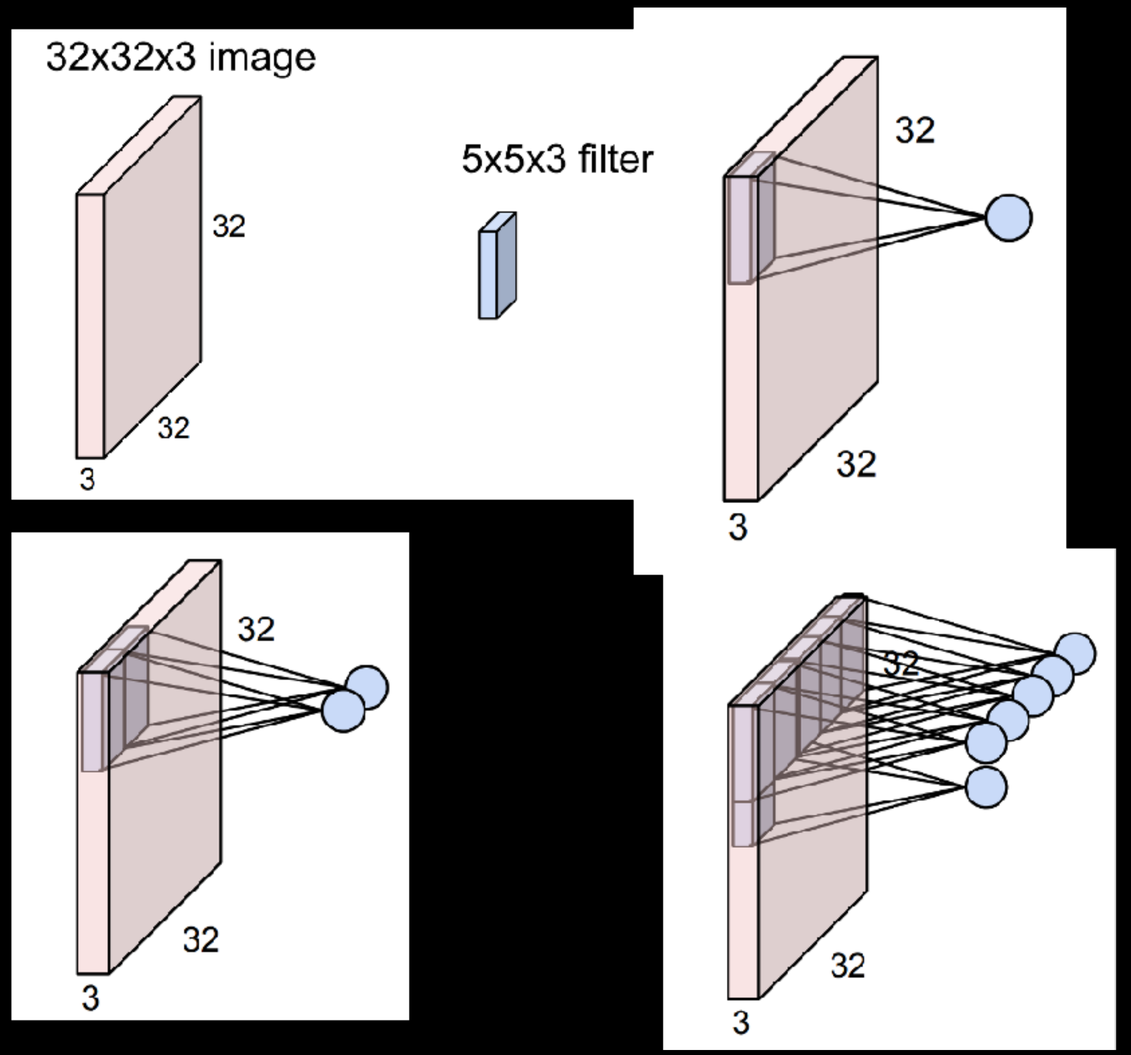
Convolutions

- The filter goes through the depth of the input
- Parameters of the filter are: width, height, depth
- In the first case, the depth is 3, representing the RGB colors

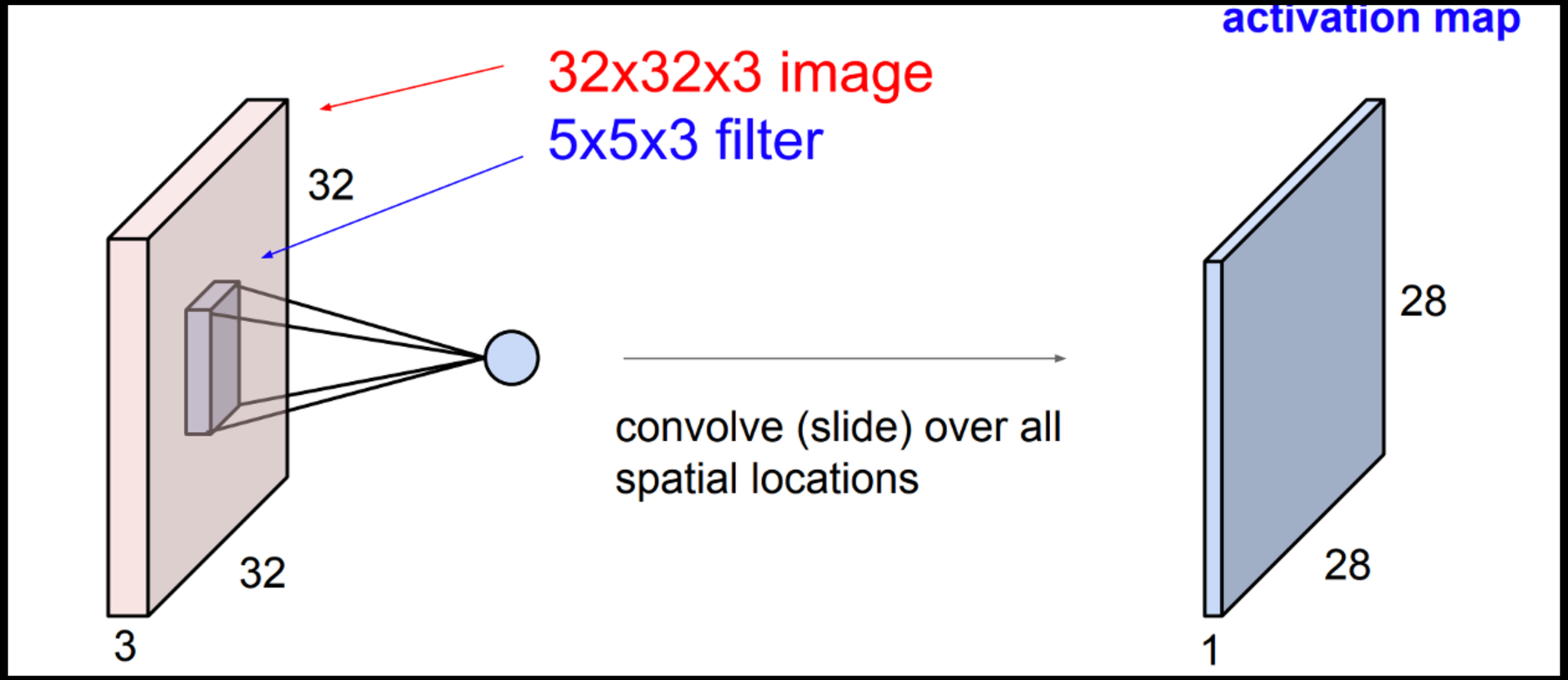


Convolutions

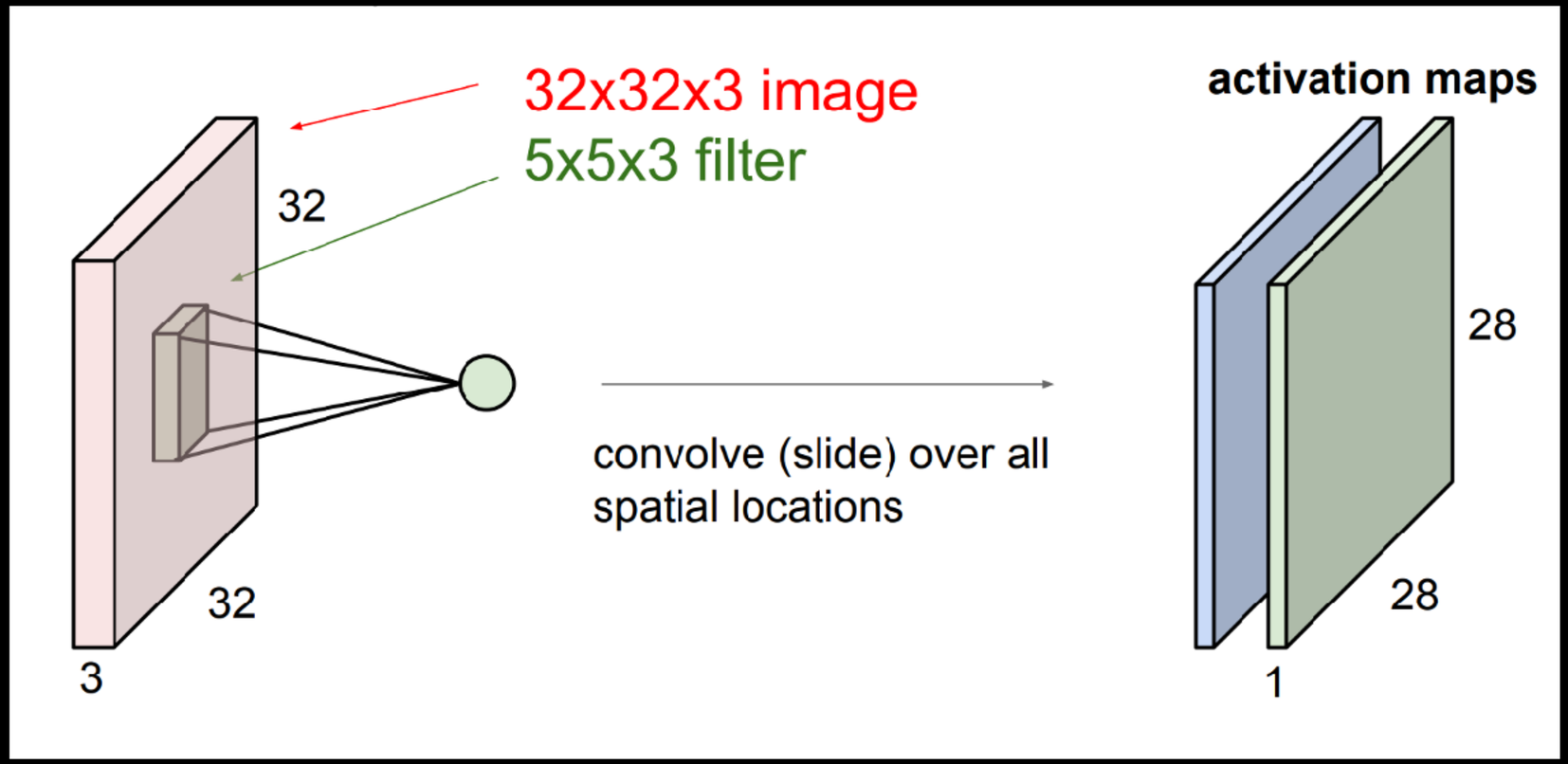
- Later on, the channel represents the amount of filters the model has available.
- Typically, you might start with a matrix with dimensions like (256, 256, 3)
- After a few layers of convolutions, you end up with dimensions like (5, 5, 512) which means there are 512 activation maps



Convolutions

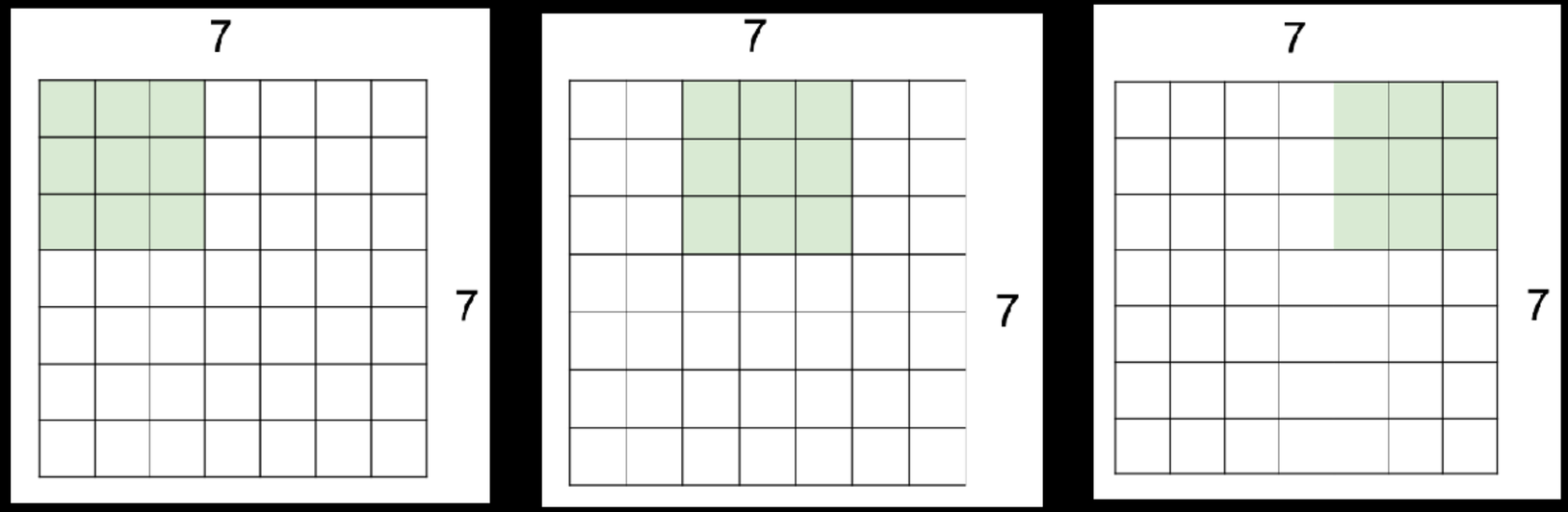


Convolutions



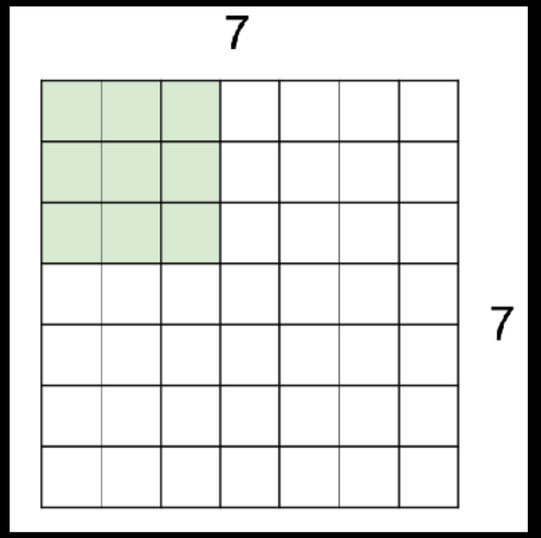
Stride

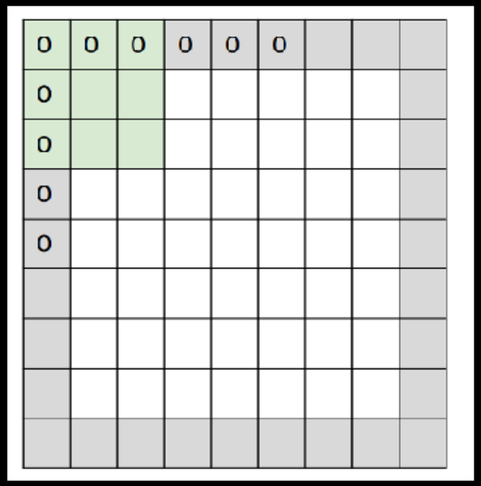
- How many pixels to step while sliding the filter
- Note that this shrinks the size if stride > 1



Padding

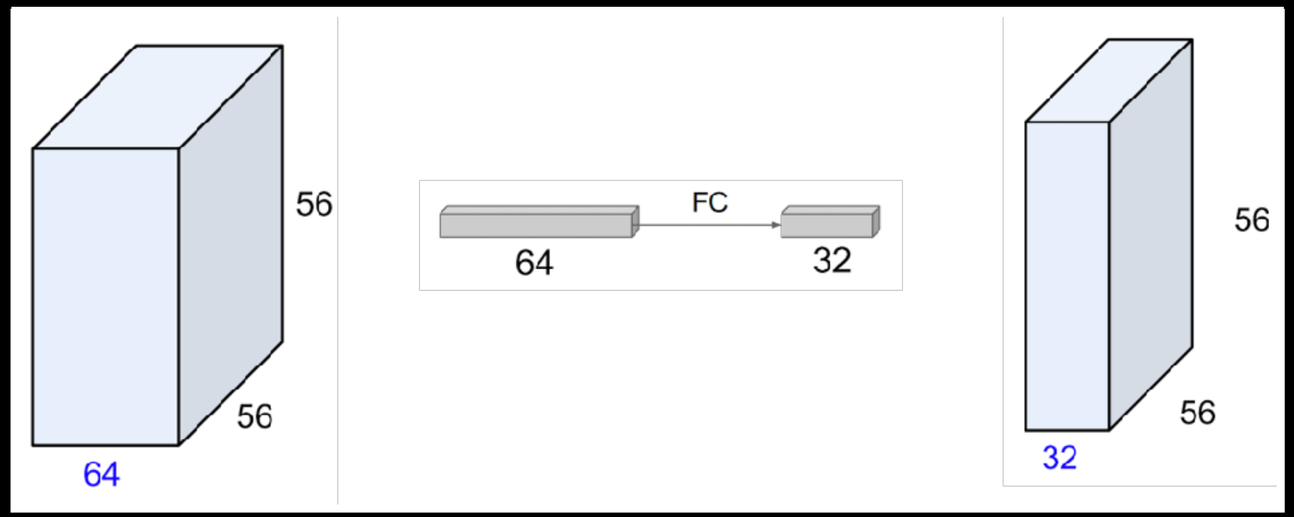
- no padding: ignore the last pixel if the filter does not fit
- add a row of zeros to make the slide fit.





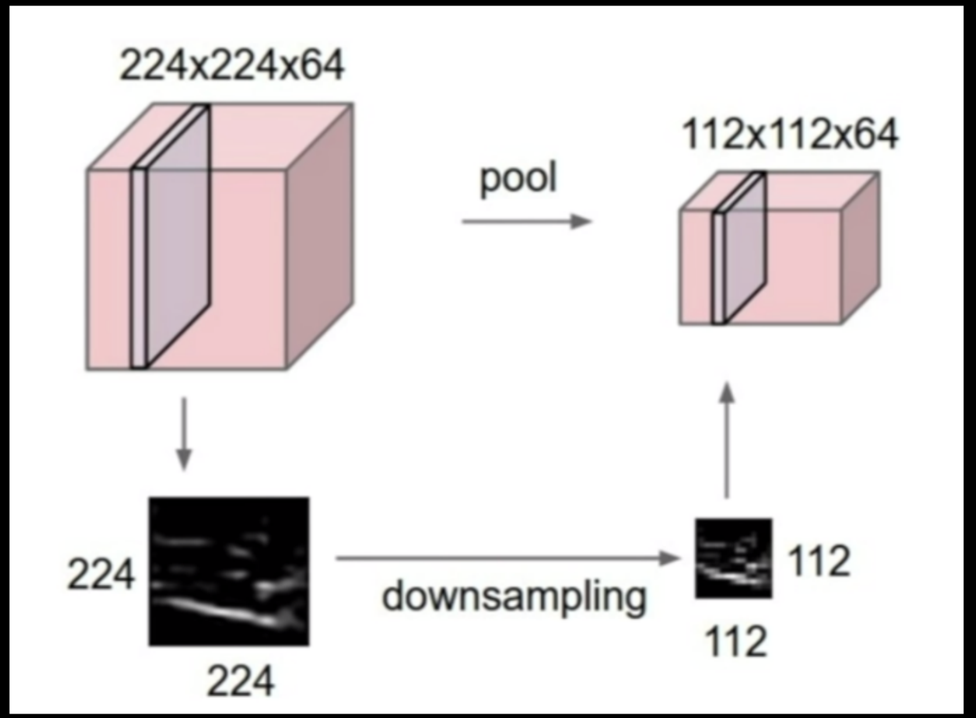
1x1 convolutions

- While the 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**.



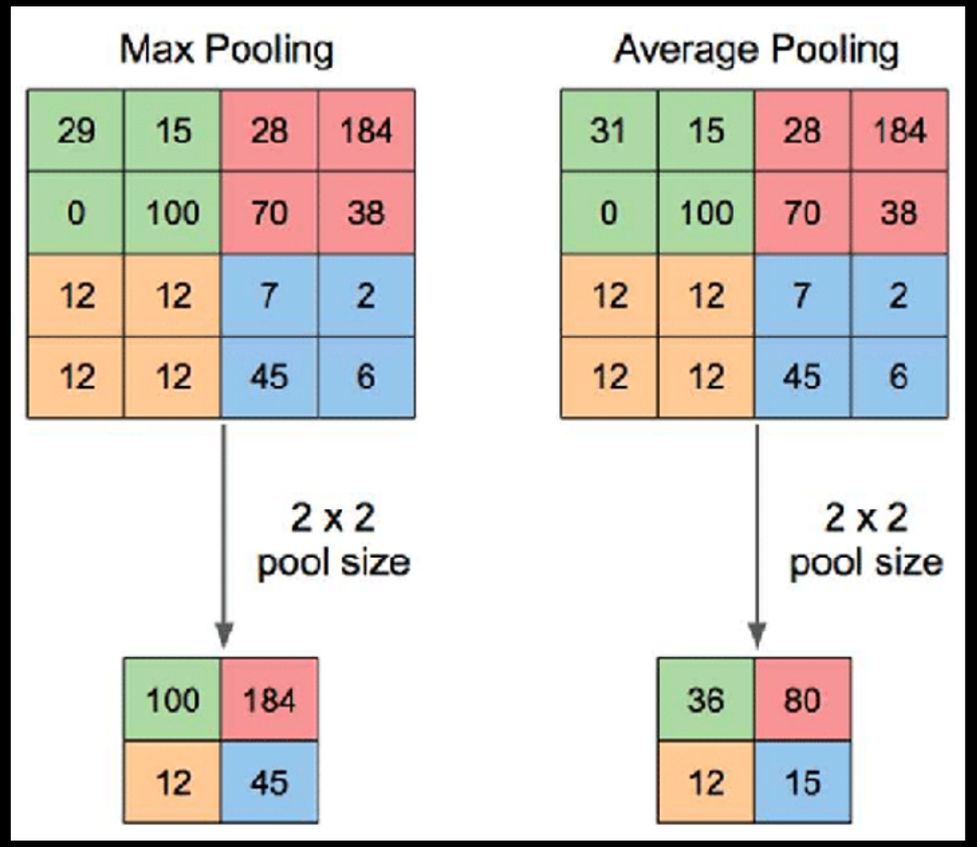
Pooling

- A way to downsample the input
- Convolutions with stride 2 also downsample, but pooling has no learnable parameters



Pooling

- max pooling: focus on the highest value
- average pooling: smooth the pixels



```python
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)
```

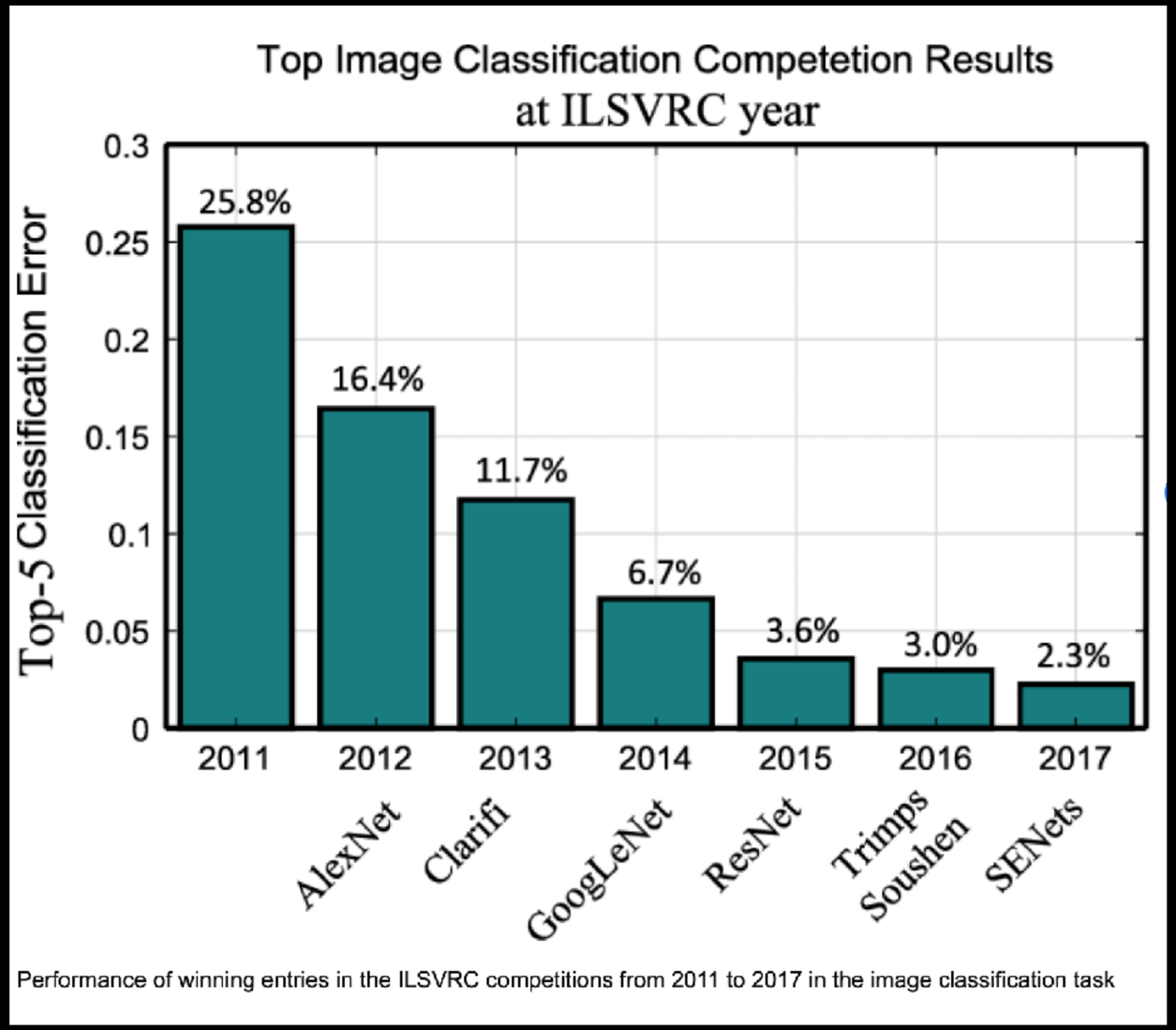
mltrainer v0.1.129 - imagemodels.py

mltrainer v0.1.129 - imagemodels.py

```python
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"Caluclated 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
```

Imagenet Challenge

- Human performance is about 5%



Performance of winning entries in the ILSVRC competitions from 2011 to 2017 in the image classification task

AlexNet

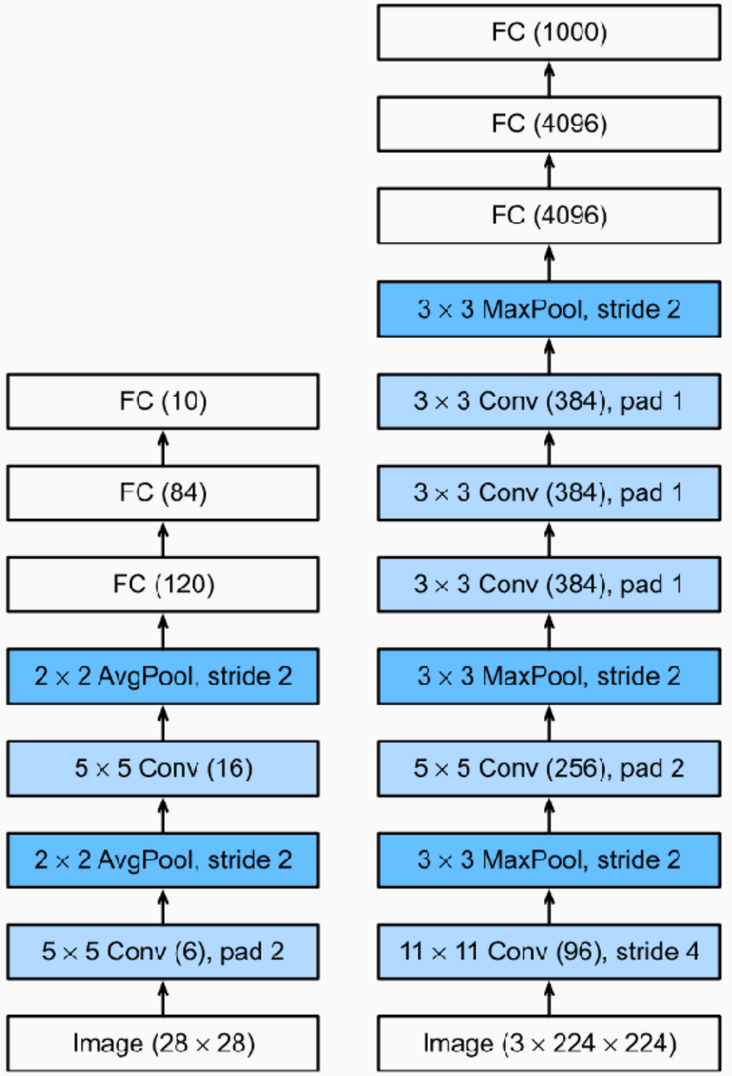


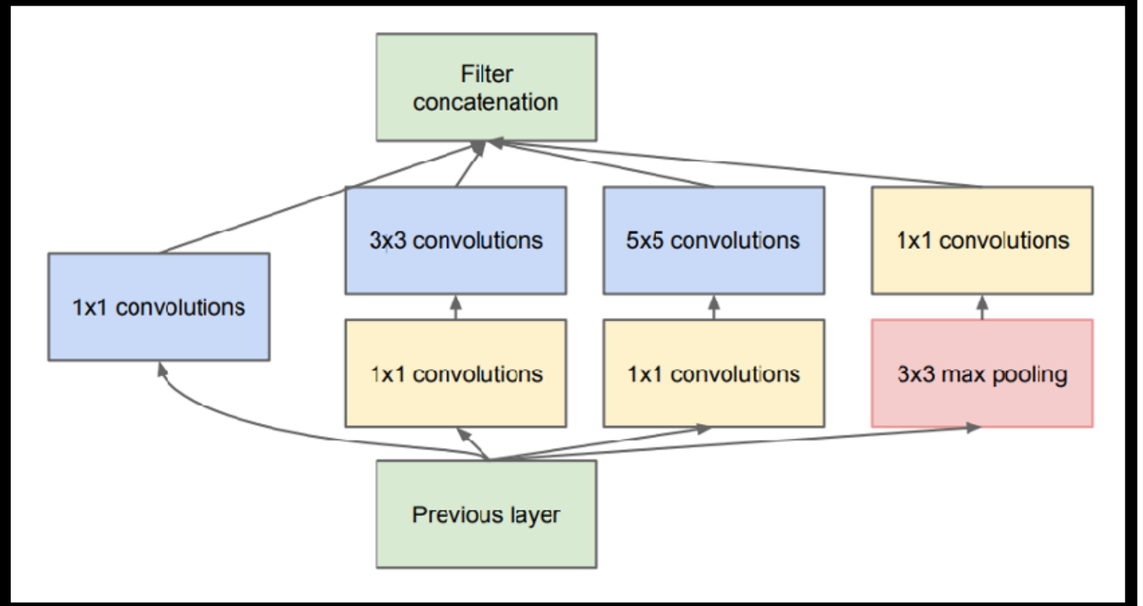
Fig. 7.1.2 From LeNet (left) to AlexNet (right).

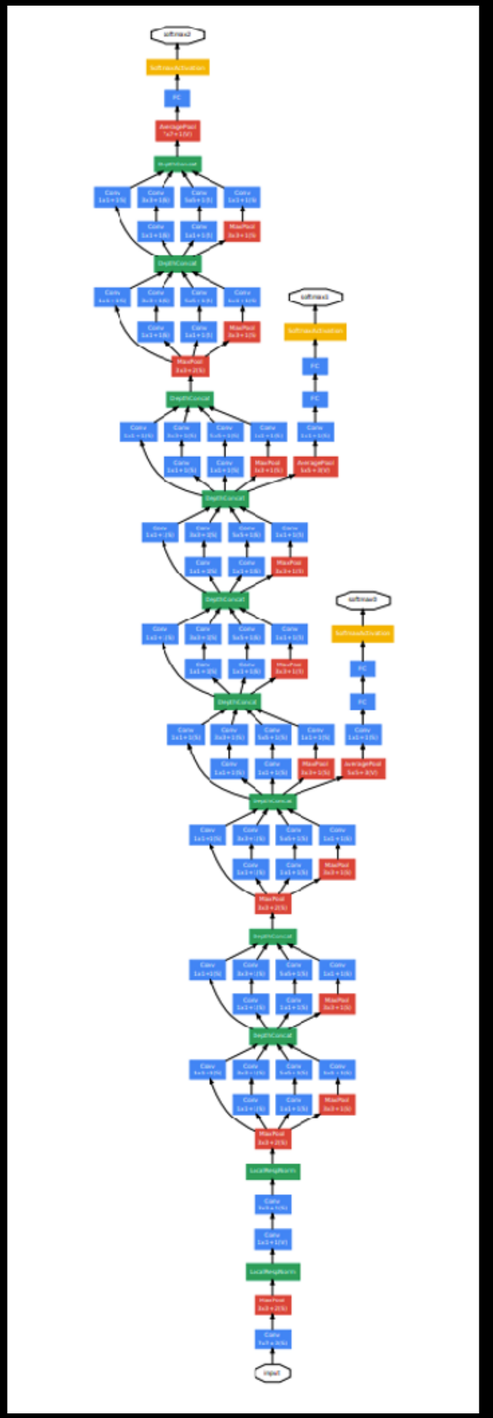
VGG

Changes in:
Filter size
channel numbers
depth
138 million parameters

GoogleNet

Parallel filters
Bottleneck layers
6.7 million parameters





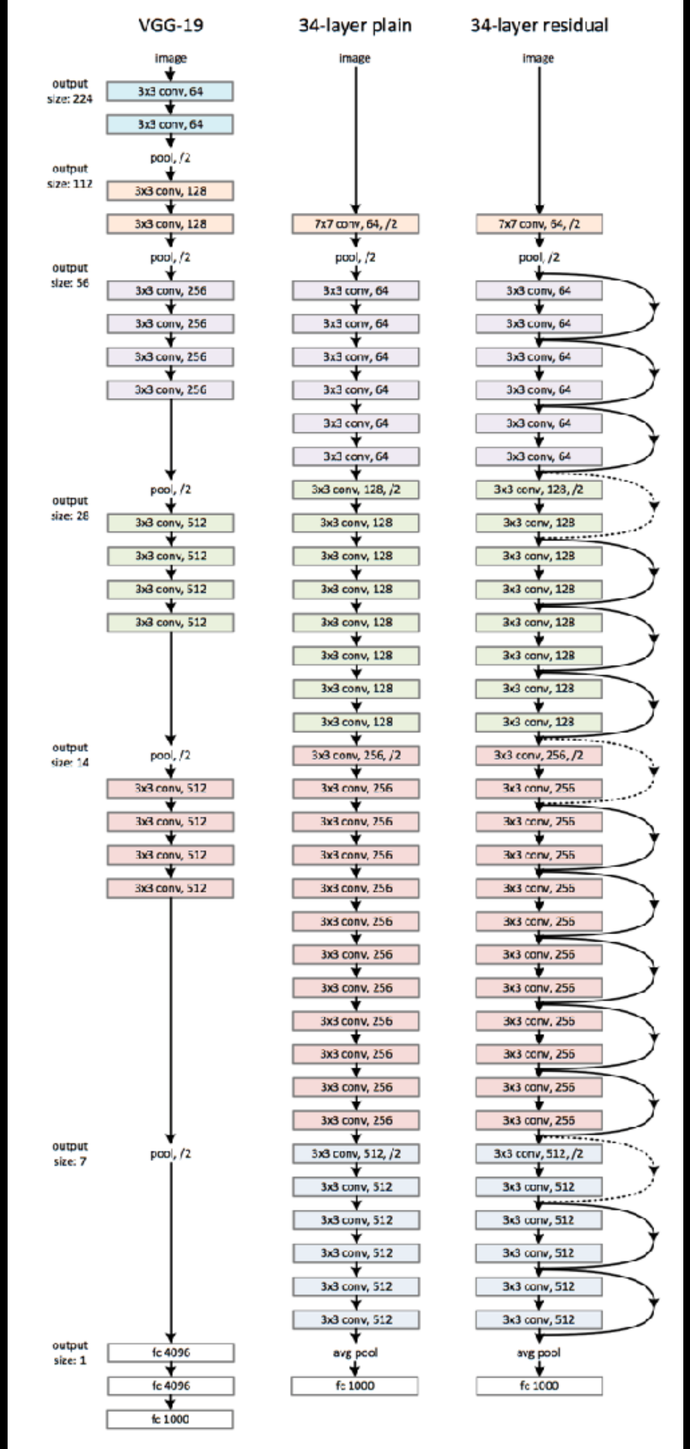
ResNet

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).



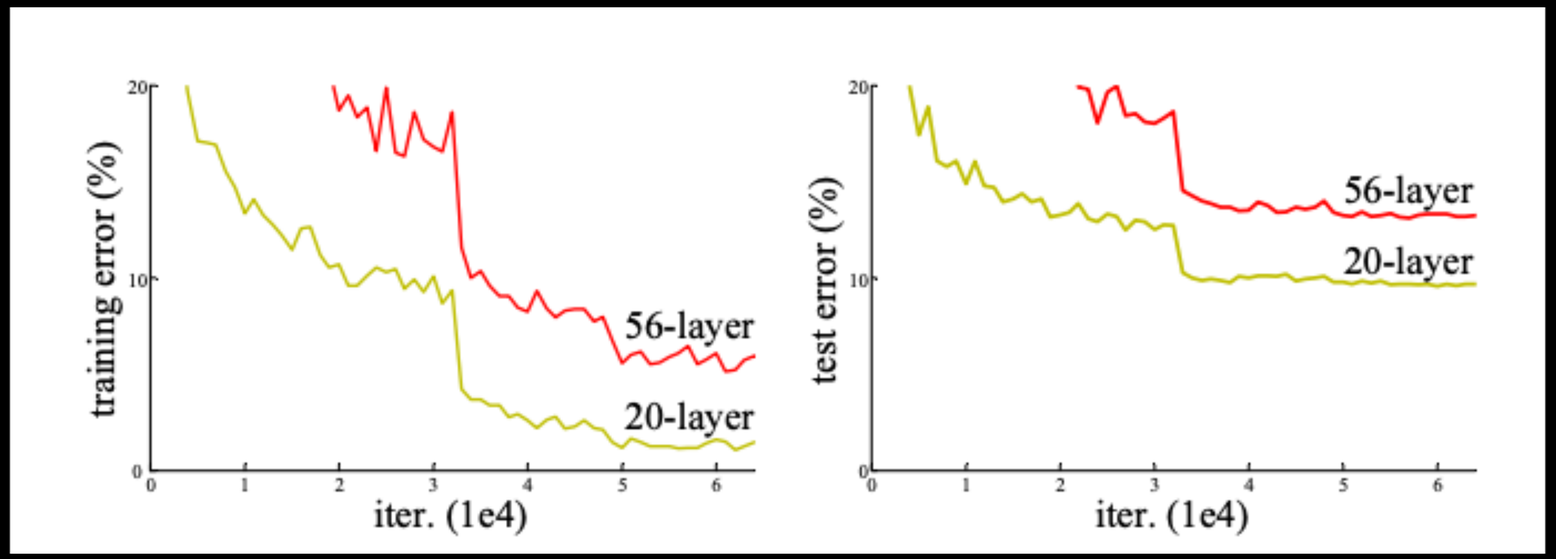
ResNet

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

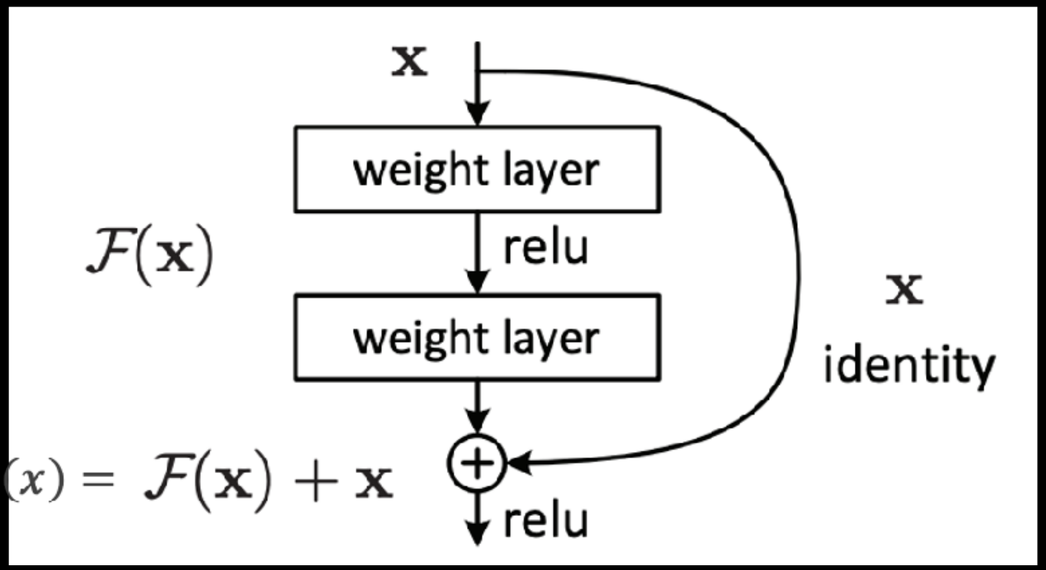


ResNet

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

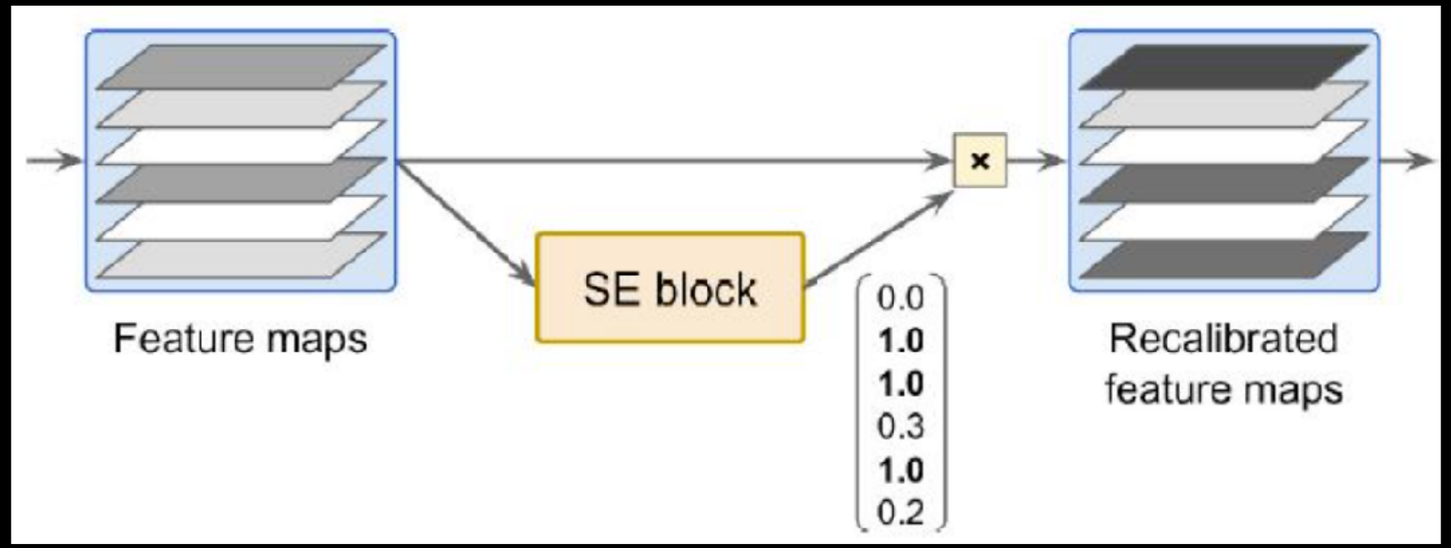


ResNet

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	3×3 max pool, stride 2				
		$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

SEnet

- 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.



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.