Convolutional Neural Networks (CNNs)

Outline

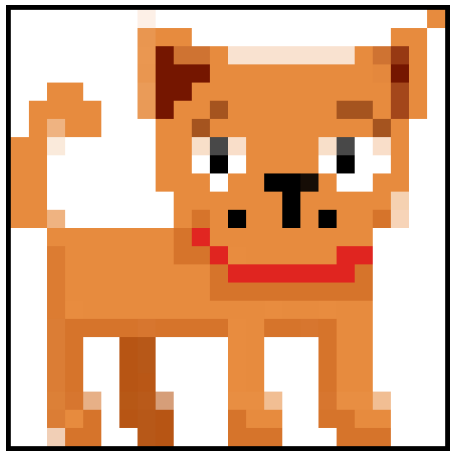
- Remember Fully-Connected Networks
- Image Kernels
- Common Applications
- Locality and Parameter Sharing
- Convolution Hyperparameters (Channels, Size, Stride, Padding, Dilation, etc.)
- The Classic CNN Architecture
- Related Concepts (Transfer Learning and Data Augmentation)

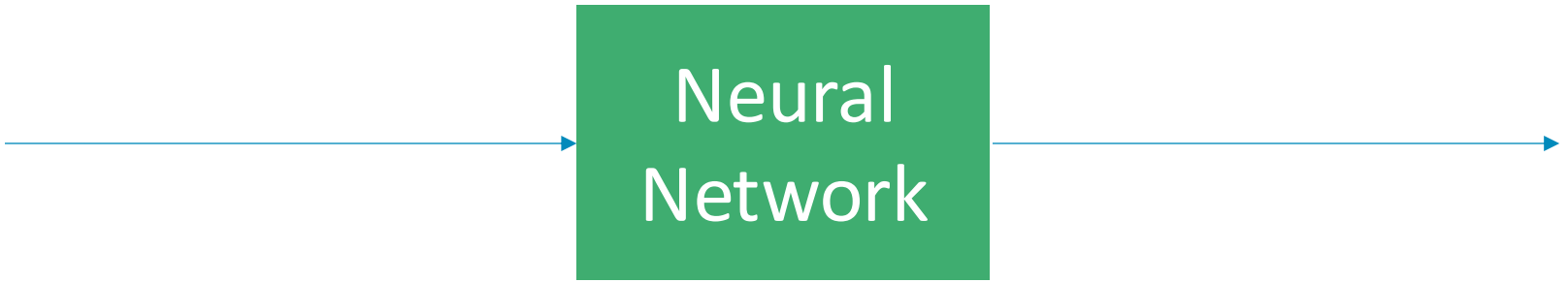
Recap: Overfitting and/or Remedies

- Take five minutes to draw
- Example: loss plots and overfit models

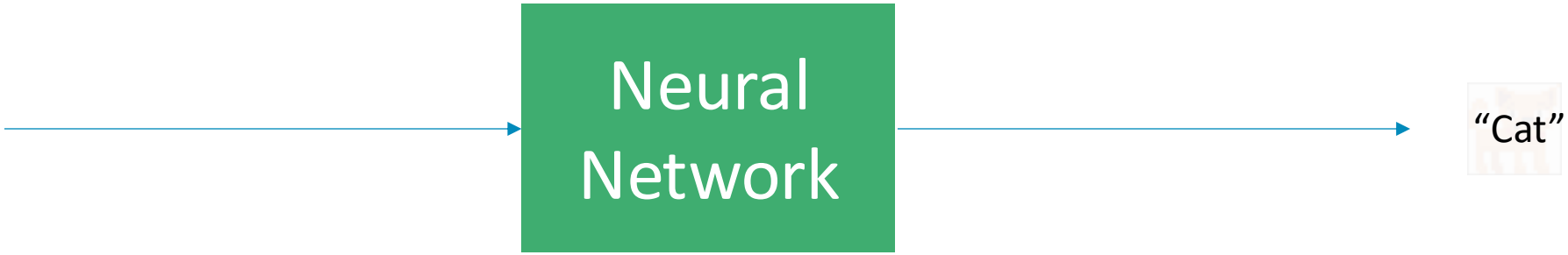
Images by macrovector on Freepik

Scenario: Cat vs Dog

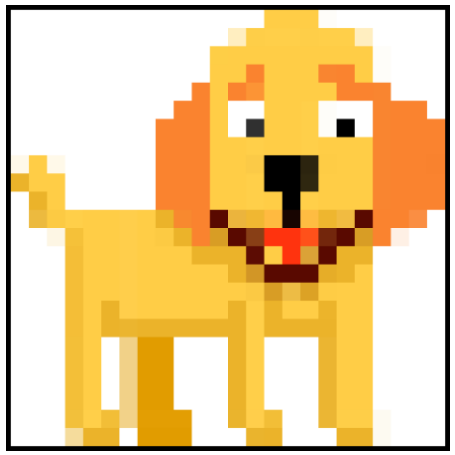


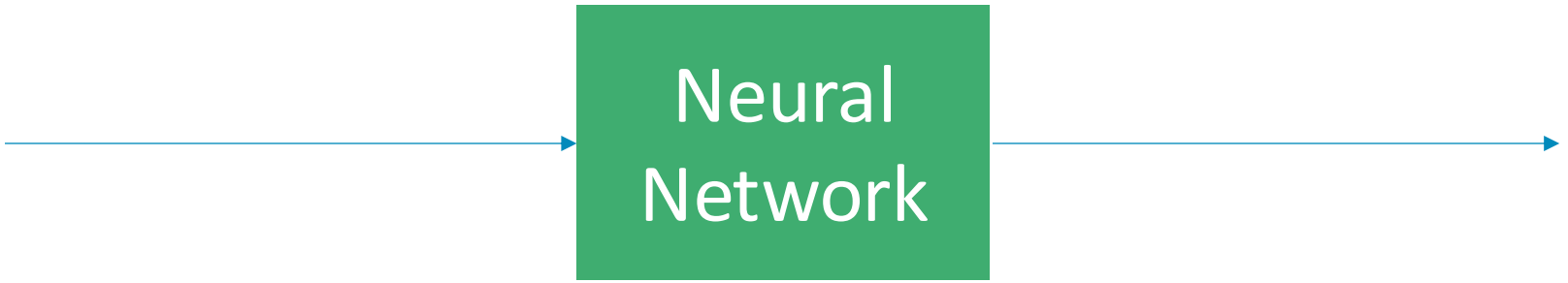


Scenario: Cat vs Dog

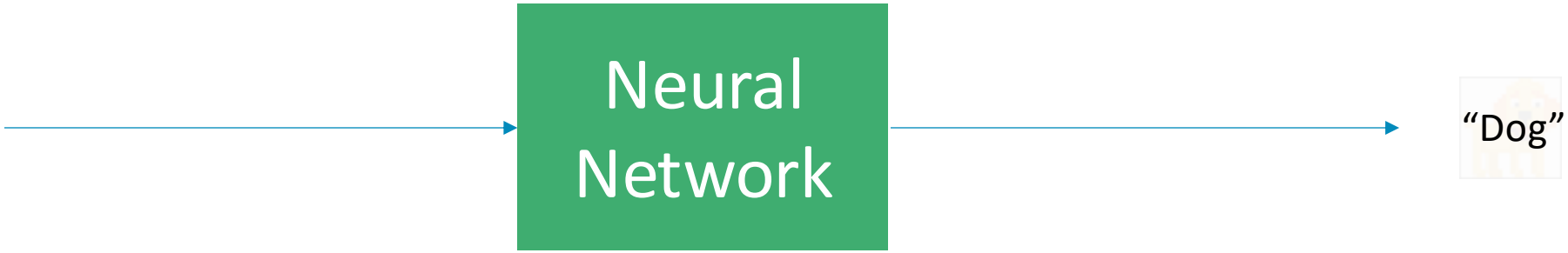


Scenario: Cat vs Dog



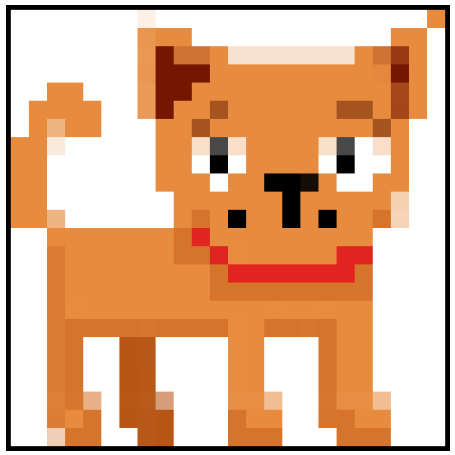


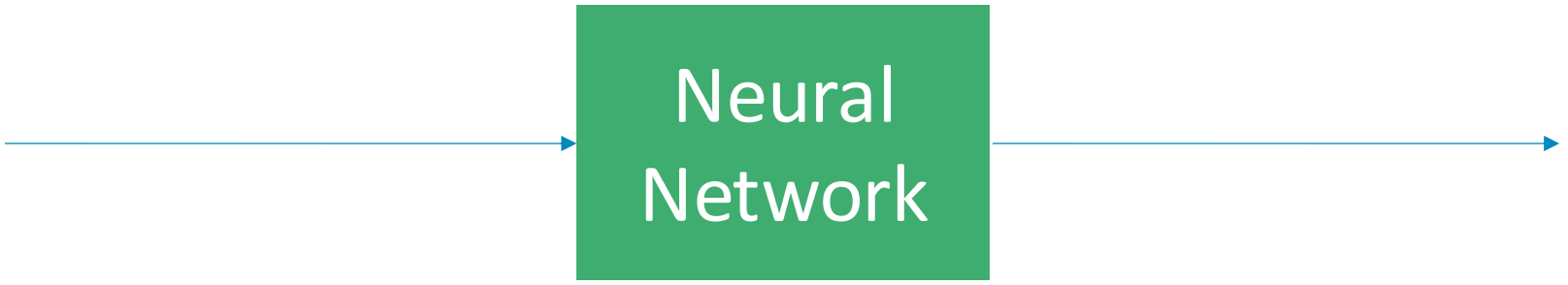
Scenario: Cat vs Dog



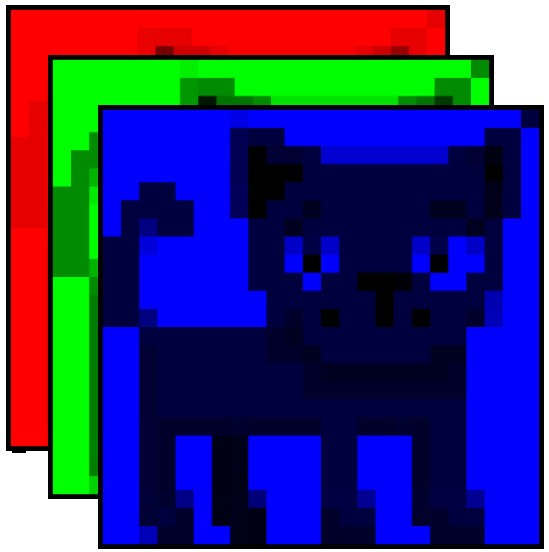
(3, 64, 64)

Scenario: Cat vs Dog





Scenario: Cat vs Dog

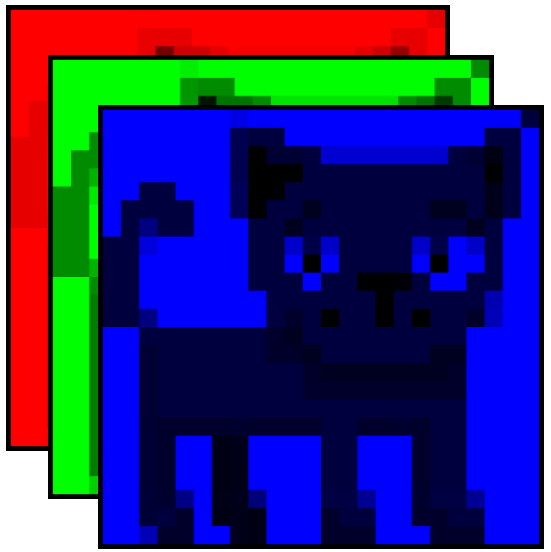


Neural Network

Scenario: Cat vs Dog

(3, 64, 64) → Flatten → (12,288, 1)

input



10 hidden layers

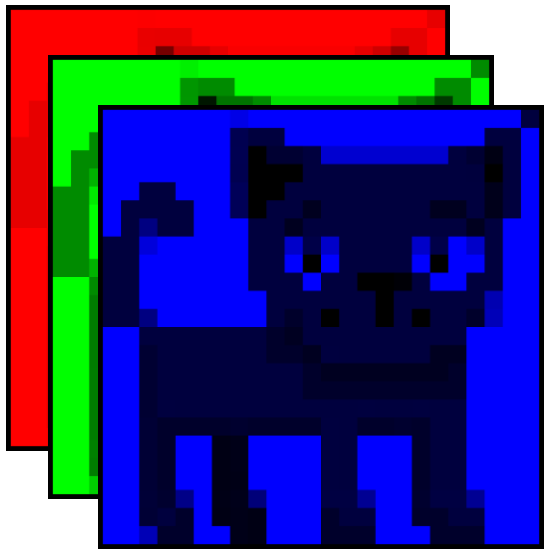
2 output neurons

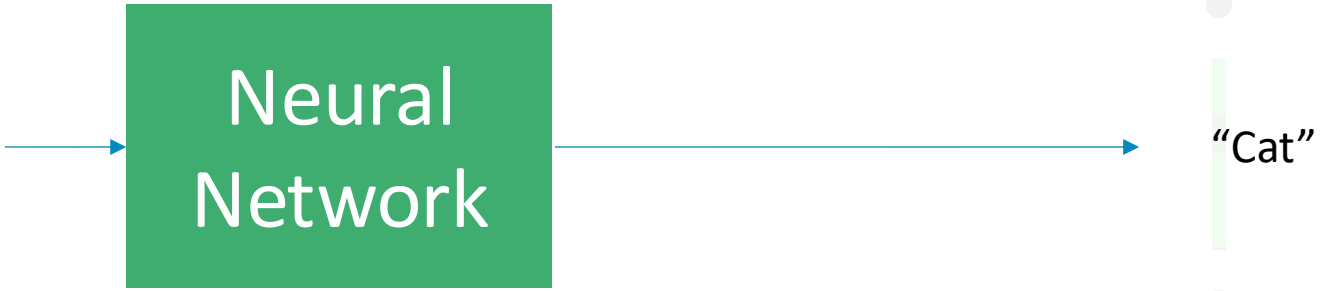
Neural Network

How many parameters?

X → [10] → [2] → Y

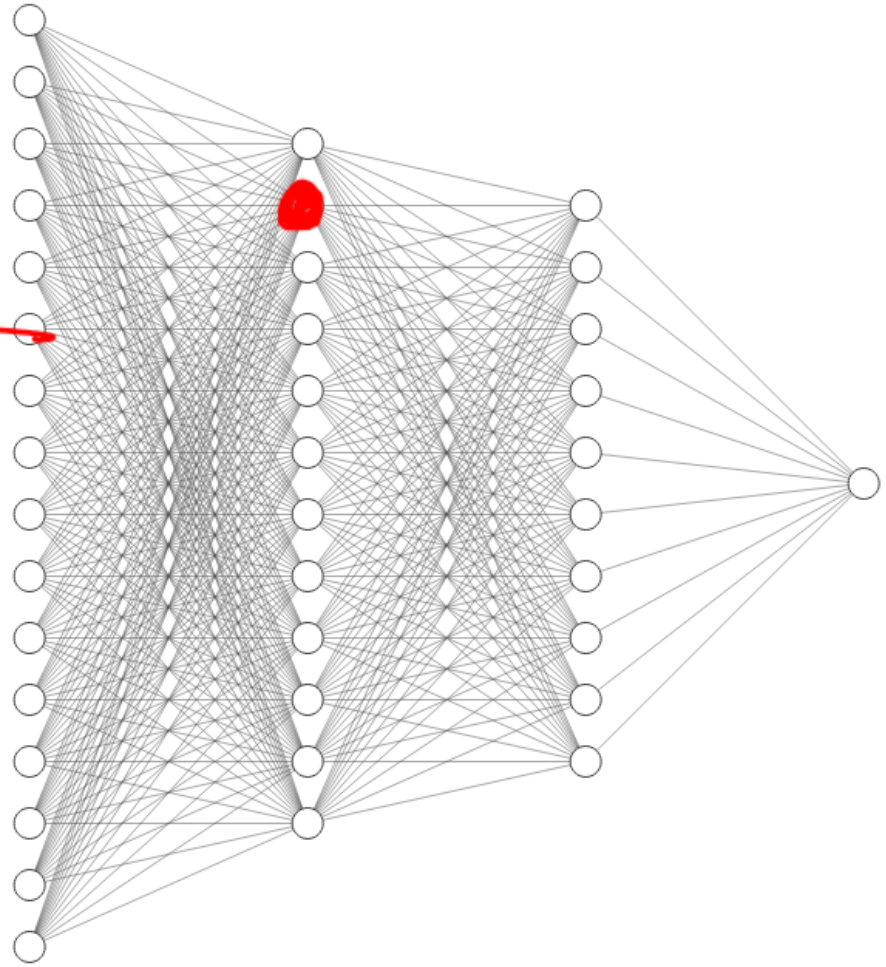
Scenario: Cat vs Dog





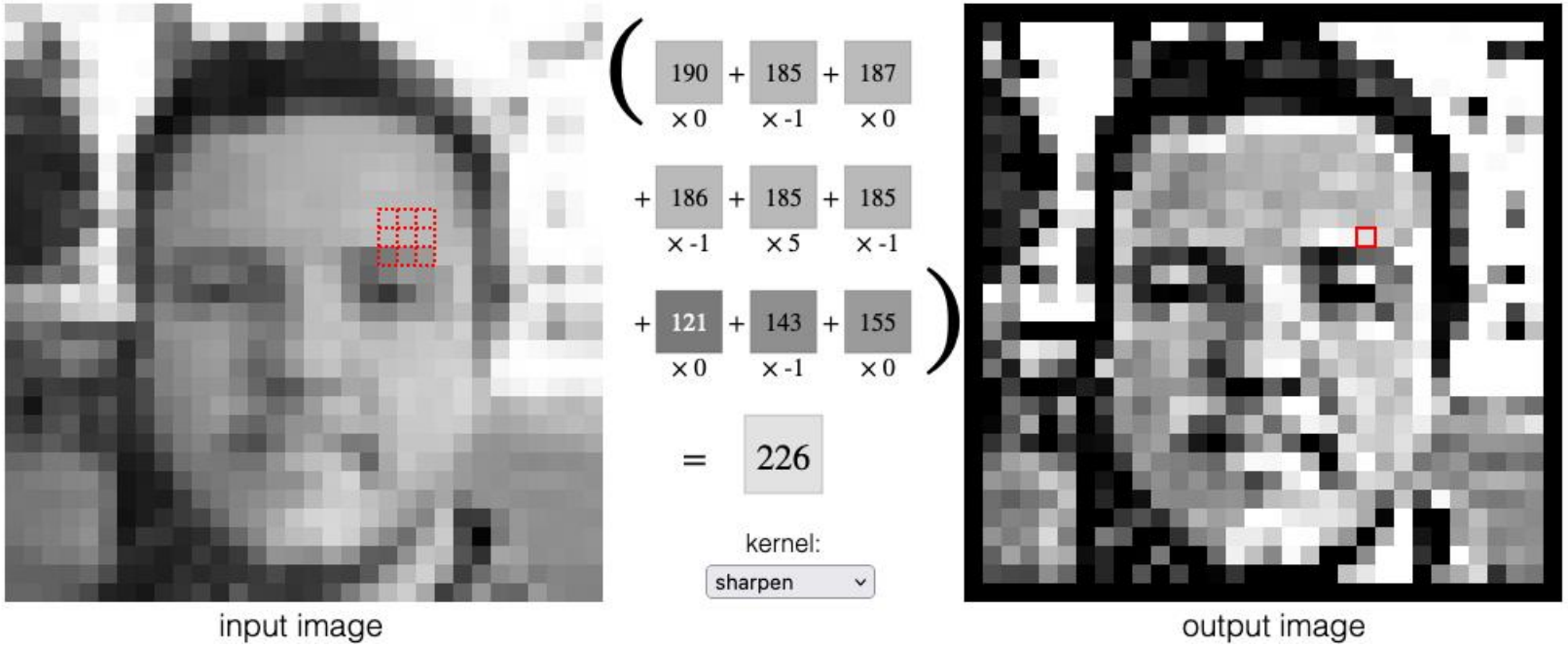
Fully-Connected Networks

- No locality: hidden layers are not given any information about the relationship among inputs
- Lots of parameters (no sharing of information)
- Is not transform invariant: what happens if all pixels are rotated around the center by 5 degrees?



Enter Convolutional Neural Networks (CNNs)

Let’s start by discussing Kernels



https://setosa.io/ev/image-kernels/

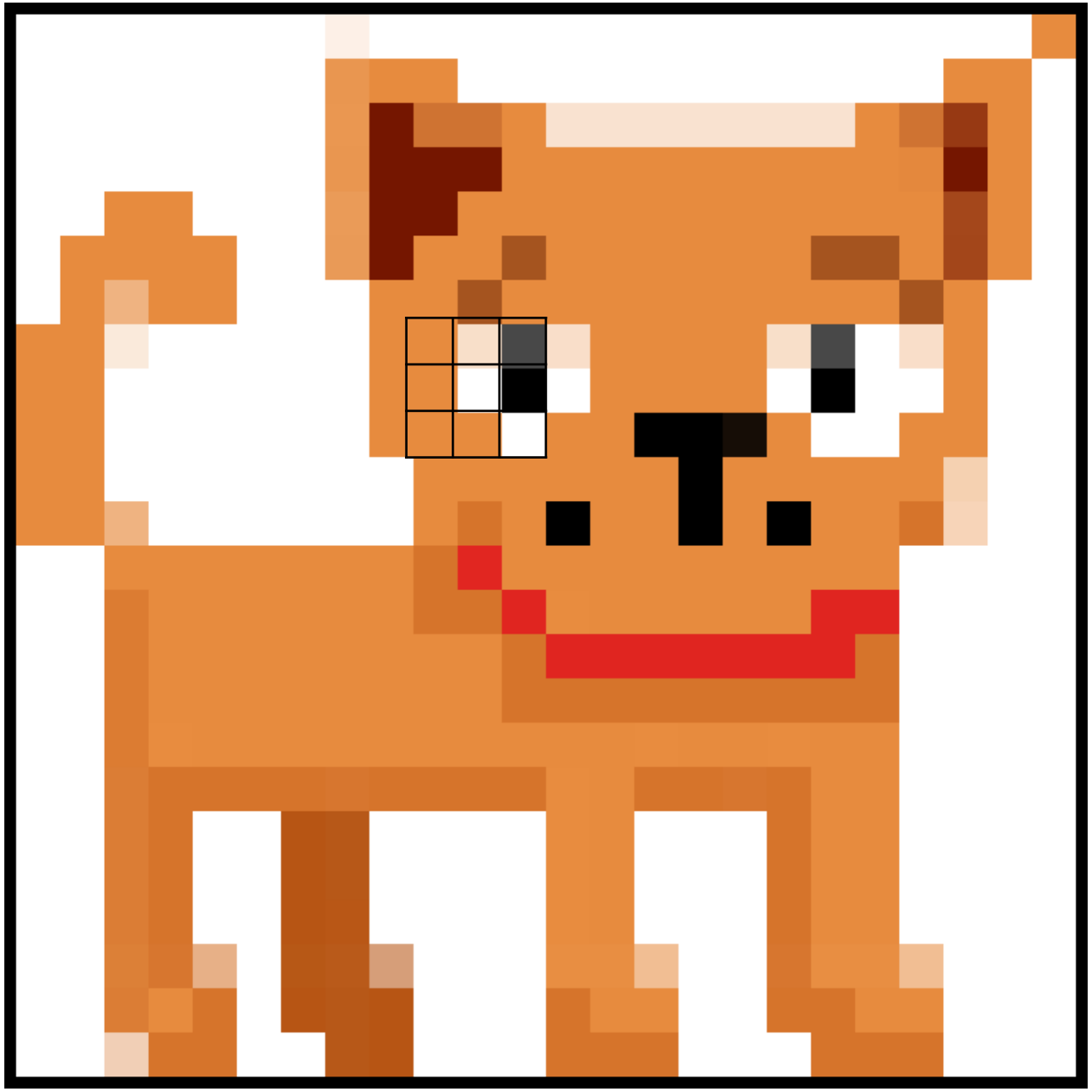
Common Applications

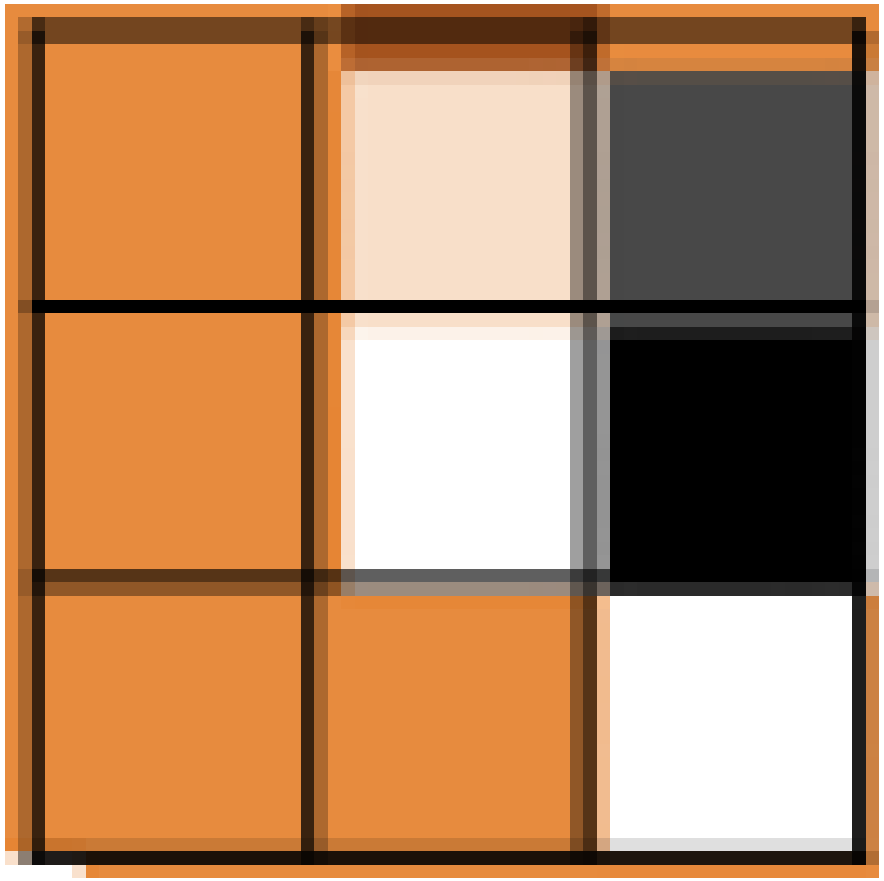
Application characteristics: structure locality to the input data.

- Give an example with input feature locality. Gaze detection
- Given an example without input feature locality. PPP Loans

- Examples
  - Image classification
  - Stock market predictions (and other time series data)
  - Object localization (where is a person's nose in an image)
  - Object detection (output all object classes and locations; YOLO and SSD)
  - Image and instance segmentation (label every pixel)
  - Pose detection from video

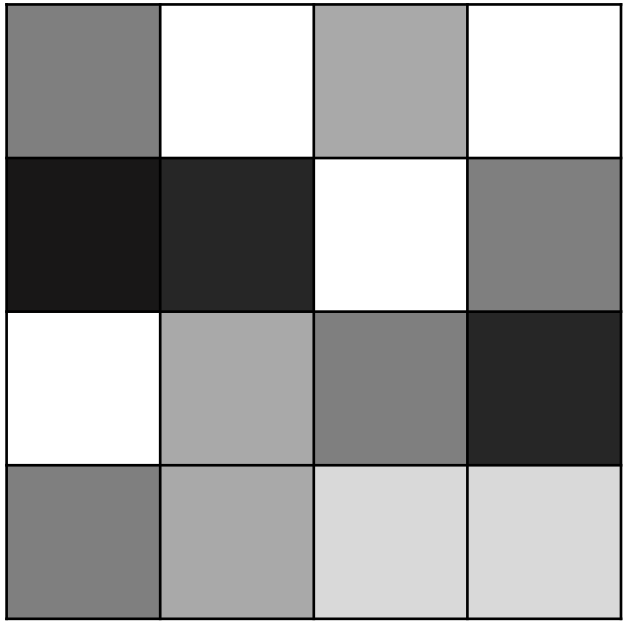
Convolution Layers

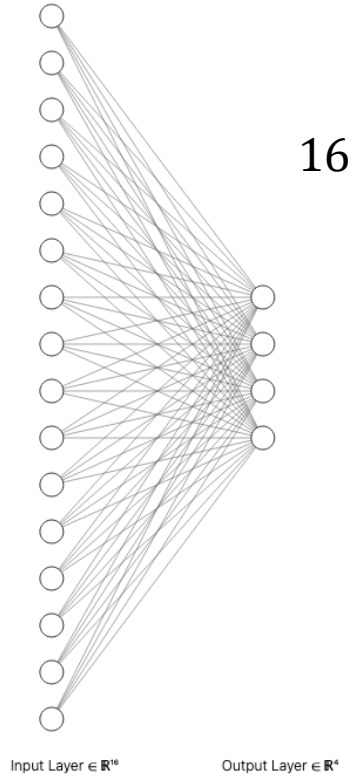




No Parameter Sharing with Fully Connected Layer

Consider a 4x4 input and four output neurons





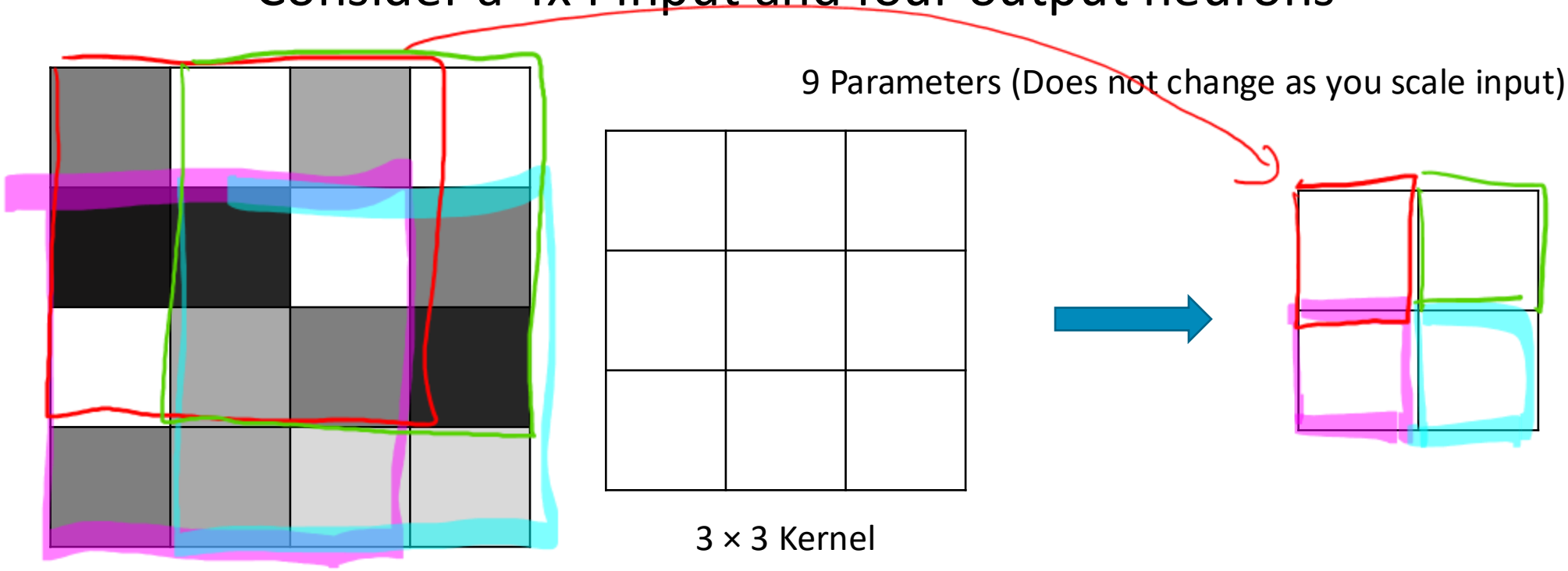
$16 \times 4 = 64$ parametesr

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \\ x_9 \\ x_{10} \\ x_{11} \\ x_{12} \\ x_{13} \\ x_{14} \\ x_{15} \\ x_{16} \end{bmatrix}^T \quad \begin{bmatrix} w_{1,1} & w_{1,2} & w_{1,3} & w_{1,4} & w_{1,5} & w_{1,6} & w_{1,7} & w_{1,8} & w_{1,9} & w_{1,10} & w_{1,11} & w_{1,12} & w_{1,13} & w_{1,14} & w_{1,15} & w_{1,16} \\ w_{2,1} & w_{2,2} & w_{2,3} & w_{2,4} & w_{2,5} & w_{2,6} & w_{2,7} & w_{2,8} & w_{2,9} & w_{2,10} & w_{2,11} & w_{2,12} & w_{2,13} & w_{2,14} & w_{2,15} & w_{2,16} \\ w_{3,1} & w_{3,2} & w_{3,3} & w_{3,4} & w_{3,5} & w_{3,6} & w_{3,7} & w_{3,8} & w_{3,9} & w_{3,10} & w_{3,11} & w_{3,12} & w_{3,13} & w_{3,14} & w_{3,15} & w_{3,16} \\ w_{4,1} & w_{4,2} & w_{4,3} & w_{4,4} & w_{4,5} & w_{4,6} & w_{4,7} & w_{4,8} & w_{4,9} & w_{4,10} & w_{4,11} & w_{4,12} & w_{4,13} & w_{4,14} & w_{4,15} & w_{4,16} \end{bmatrix}^T = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix}$$

$(16 \times 1)^T$ $(4 \times 16)^T$

Parameter Sharing

Consider a 4x4 input and four output neurons



$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \\ x_9 \\ x_{10} \\ x_{11} \\ x_{12} \\ x_{13} \\ x_{14} \\ x_{15} \\ x_{16} \end{bmatrix}^T$$

$(16 \times 1)^T$

$$\begin{bmatrix} k_{1,1} & k_{1,2} & k_{1,3} & 0 & k_{2,1} & k_{2,2} & k_{2,3} & 0 & k_{3,1} & k_{3,2} & k_{3,3} & 0 & 0 & 0 & 0 & 0 \\ 0 & k_{1,1} & k_{1,2} & k_{1,3} & 0 & k_{2,1} & k_{2,2} & k_{2,3} & 0 & k_{3,1} & k_{3,2} & k_{3,3} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & k_{1,1} & k_{1,2} & k_{1,3} & 0 & k_{2,1} & k_{2,2} & k_{2,3} & 0 & k_{3,1} & k_{3,2} & k_{3,3} & 0 \\ 0 & 0 & 0 & 0 & 0 & k_{1,1} & k_{1,2} & k_{1,3} & 0 & k_{2,1} & k_{2,2} & k_{2,3} & 0 & k_{3,1} & k_{3,2} & k_{3,3} \end{bmatrix}^T = \begin{bmatrix} y_{1,1} & y_{1,2} \\ y_{2,1} & y_{2,2} \end{bmatrix}$$

$(4 \times 16)^T$

Fully Connected vs. Convolutional

Consider a 4x4 input and four output neurons

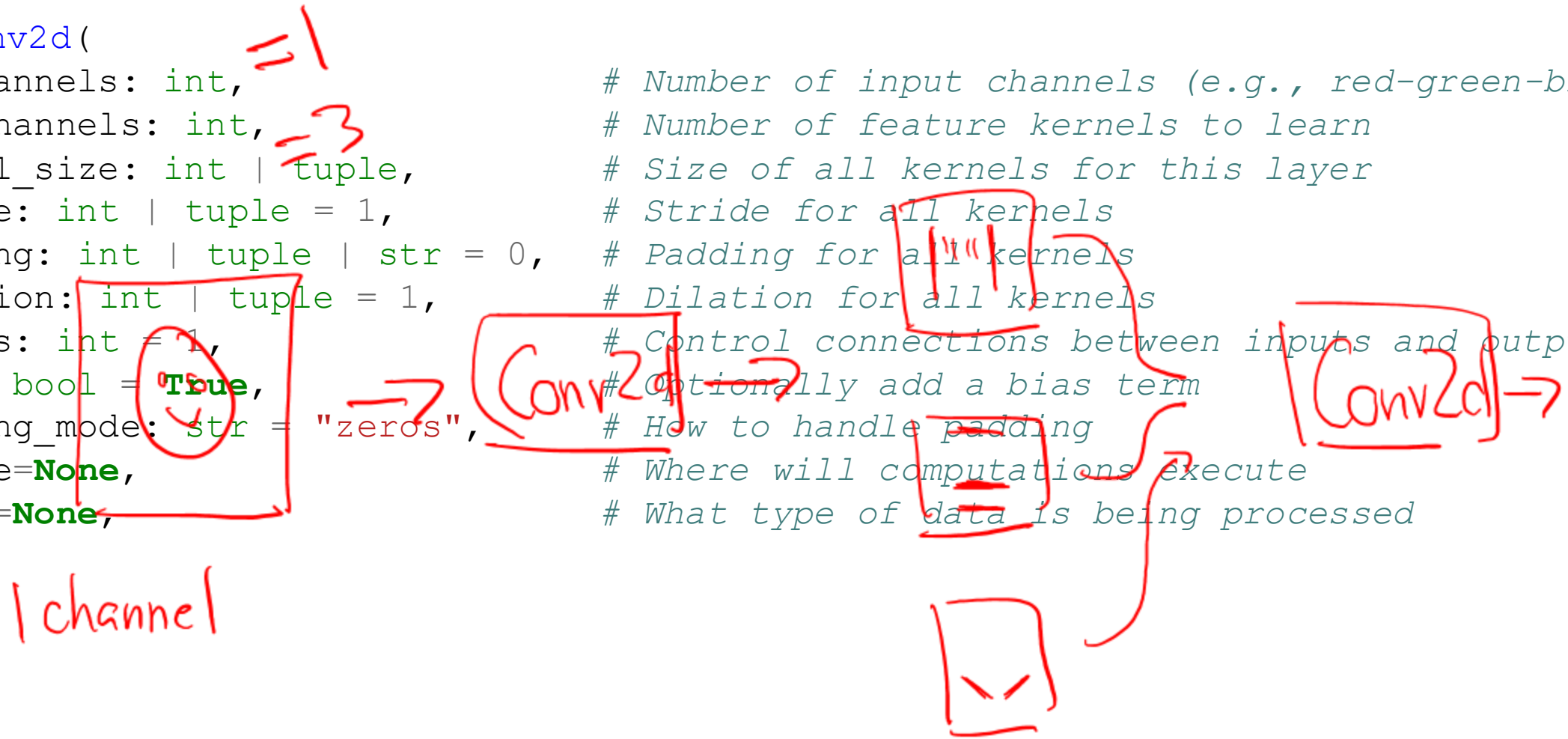
$$\begin{bmatrix} w_{1,1} & w_{1,2} & w_{1,3} & w_{1,4} & w_{1,5} & w_{1,6} & w_{1,7} & w_{1,8} & w_{1,9} & w_{1,10} & w_{1,11} & w_{1,12} & w_{1,13} & w_{1,14} & w_{1,15} & w_{1,16} \\ w_{2,1} & w_{2,2} & w_{2,3} & w_{2,4} & w_{2,5} & w_{2,6} & w_{2,7} & w_{2,8} & w_{2,9} & w_{2,10} & w_{2,11} & w_{2,12} & w_{2,13} & w_{2,14} & w_{2,15} & w_{2,16} \\ w_{3,1} & w_{3,2} & w_{3,3} & w_{3,4} & w_{3,5} & w_{3,6} & w_{3,7} & w_{3,8} & w_{3,9} & w_{3,10} & w_{3,11} & w_{3,12} & w_{3,13} & w_{3,14} & w_{3,15} & w_{3,16} \\ w_{4,1} & w_{4,2} & w_{4,3} & w_{4,4} & w_{4,5} & w_{4,6} & w_{4,7} & w_{4,8} & w_{4,9} & w_{4,10} & w_{4,11} & w_{4,12} & w_{4,13} & w_{4,14} & w_{4,15} & w_{4,16} \end{bmatrix}$$

$$\begin{bmatrix} k_{1,1} & k_{1,2} & k_{1,3} & 0 & k_{2,1} & k_{2,2} & k_{2,3} & 0 & k_{3,1} & k_{3,2} & k_{3,3} & 0 & 0 & 0 & 0 & 0 \\ 0 & k_{1,1} & k_{1,2} & k_{1,3} & 0 & k_{2,1} & k_{2,2} & k_{2,3} & 0 & k_{3,1} & k_{3,2} & k_{3,3} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & k_{1,1} & k_{1,2} & k_{1,3} & 0 & k_{2,1} & k_{2,2} & k_{2,3} & 0 & k_{3,1} & k_{3,2} & k_{3,3} & 0 \\ 0 & 0 & 0 & 0 & 0 & k_{1,1} & k_{1,2} & k_{1,3} & 0 & k_{2,1} & k_{2,2} & k_{2,3} & 0 & k_{3,1} & k_{3,2} & k_{3,3} \end{bmatrix}$$

Convolution Hyperparameters

Convolution Layers

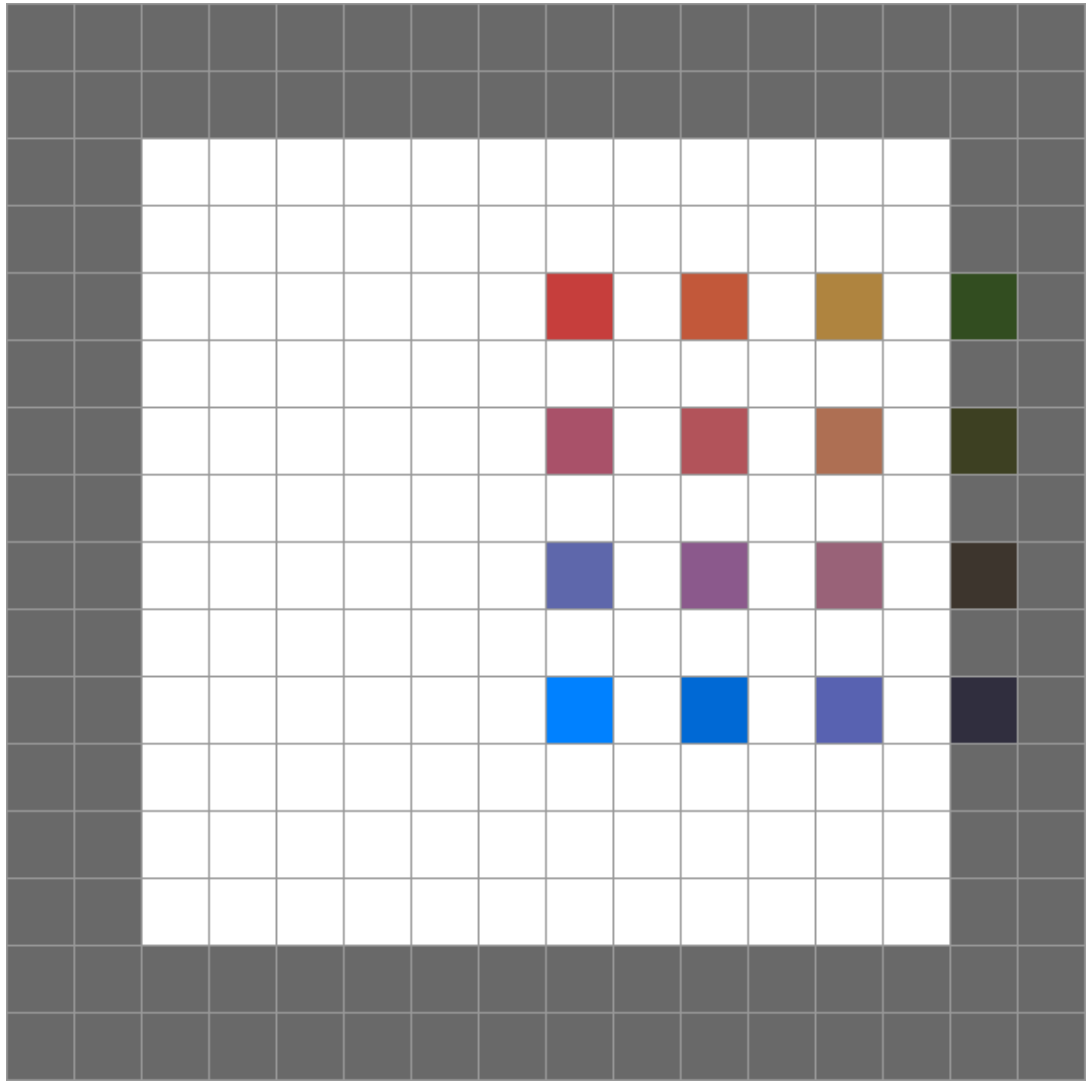
```
CLASS Conv2d(
    in_channels: int,                # Number of input channels (e.g., red-green-blue)
    out_channels: int,               # Number of feature kernels to learn
    kernel_size: int | tuple,        # Size of all kernels for this layer
    stride: int | tuple = 1,         # Stride for all kernels
    padding: int | tuple | str = 0,  # Padding for all kernels
    dilation: int | tuple = 1,       # Dilation for all kernels
    groups: int = 1,                 # Control connections between inputs and outputs
    bias: bool = True,               # Optionally add a bias term
    padding_mode: str = "zeros",     # How to handle padding
    device=None,                     # Where will computations execute
    dtype=None,                      # What type of data is being processed
):
    …
```



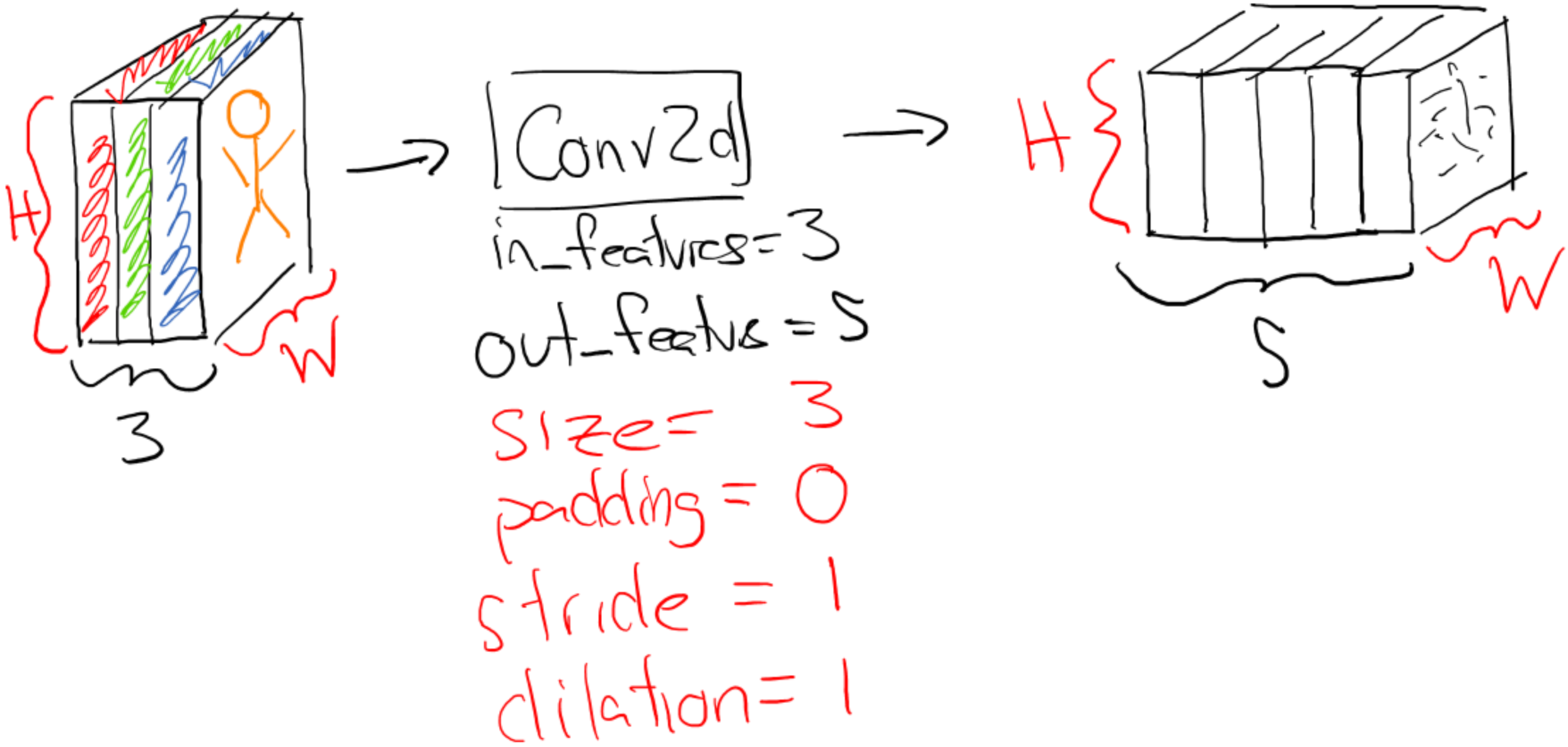
Convolution Layers

```
CLASS Conv2d(
    in_channels: int,                   # Number of input channels (e.g., red-green-blue)
    out_channels: int,                  # Number of feature kernels to learn
    kernel_size: int | tuple,           # Size of all kernels for this layer
    stride: int | tuple = 1,            # Stride for all kernels
    padding: int | tuple | str = 0,     # Padding for all kernels
    dilation: int | tuple = 1,          # Dilation for all kernels
    groups: int = 1,                    # Control connections between inputs and outputs
    bias: bool = True,                  # Optionally add a bias term
    padding_mode: str = "zeros",        # How to handle padding
    device=None,                        # Where will computations execute
    dtype=None,                         # What type of data is being processed
):
     …
```

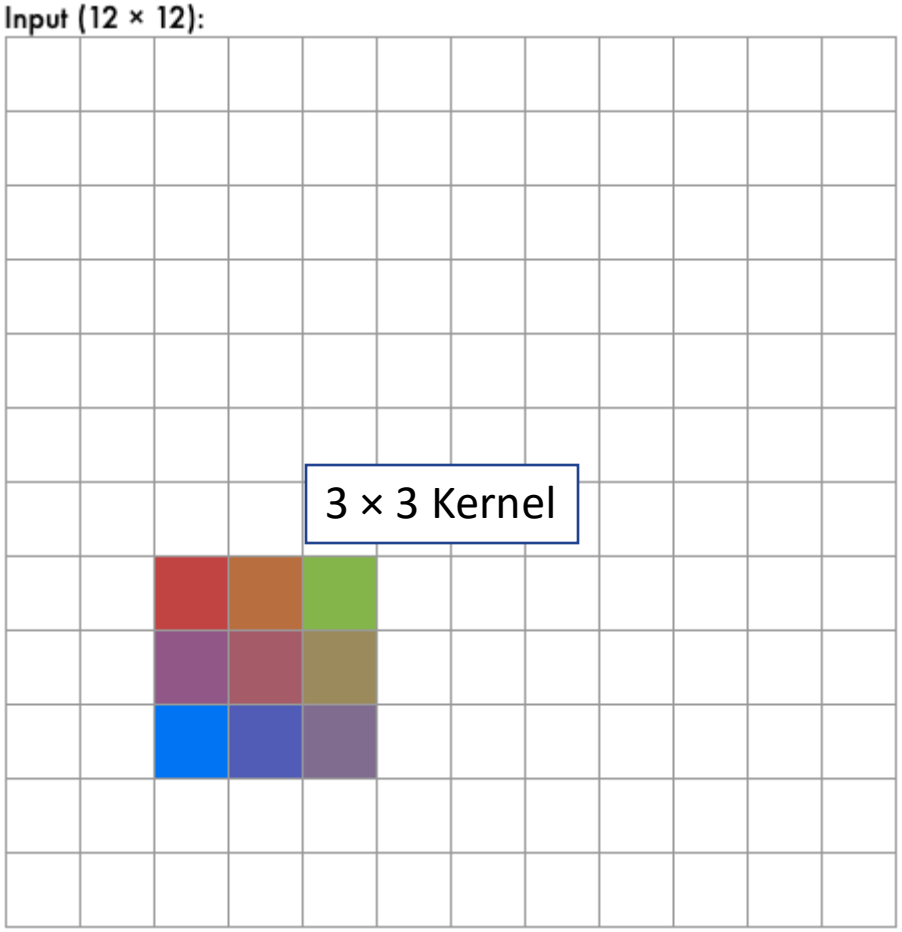
Visualization Helps: Convolution Visualizer

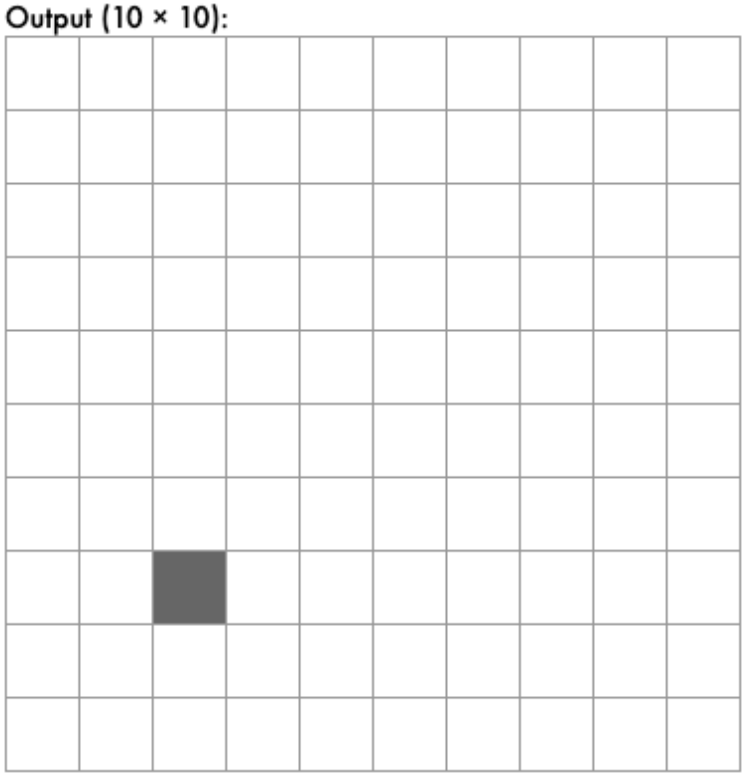


Convolution Hyperparameters: Filters/Channels (C)



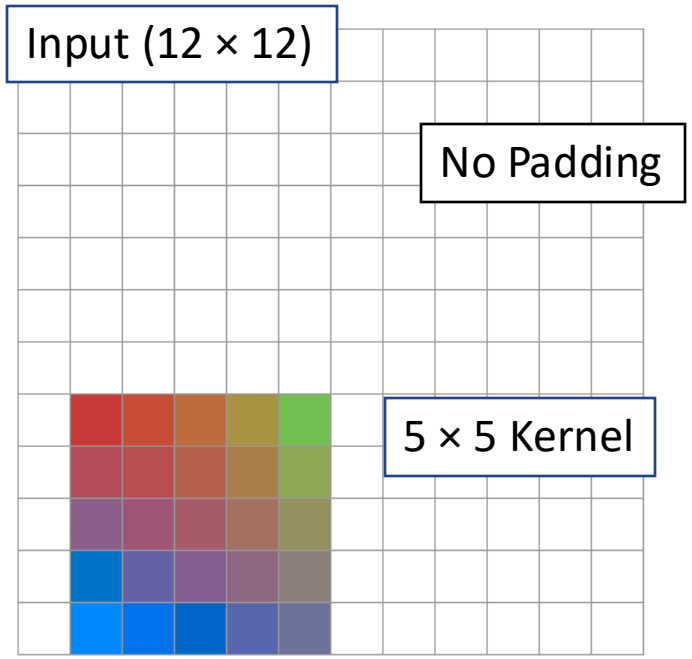
Convolution Hyperparameters: Kernel Size (K)

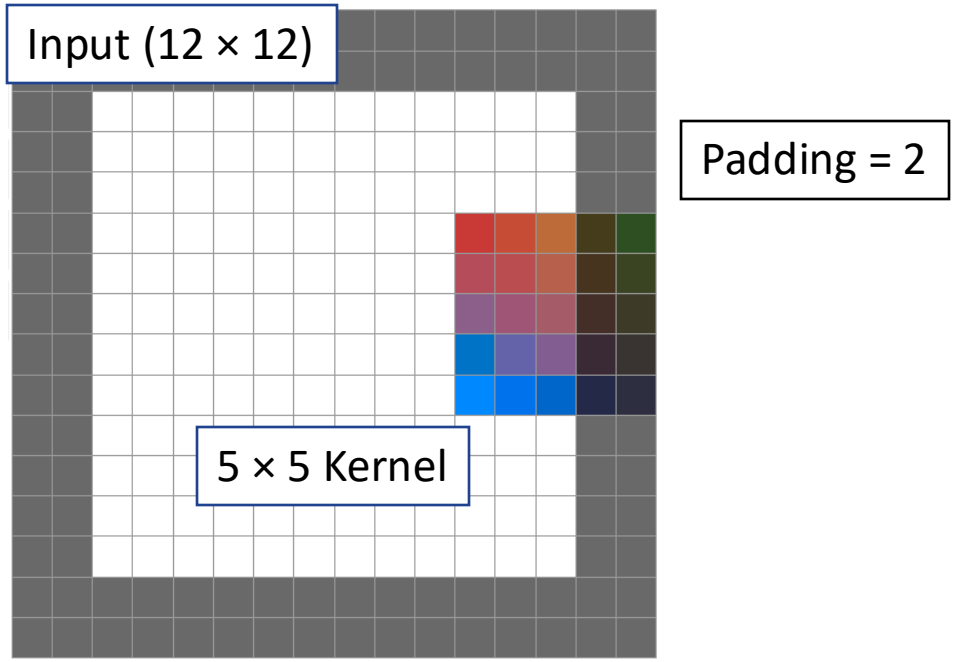


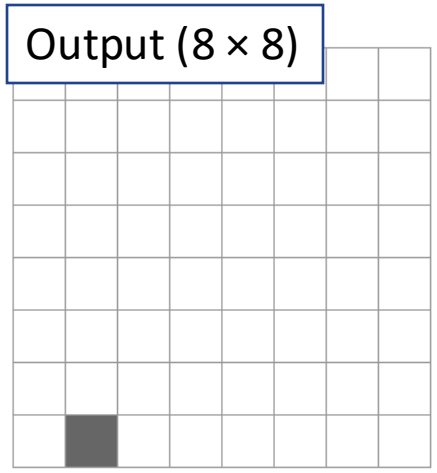


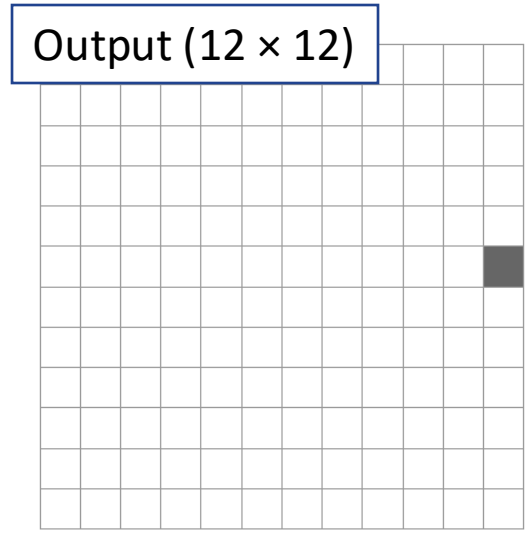
Convolution Visualizer

Convolution Hyperparameters: Padding (P)



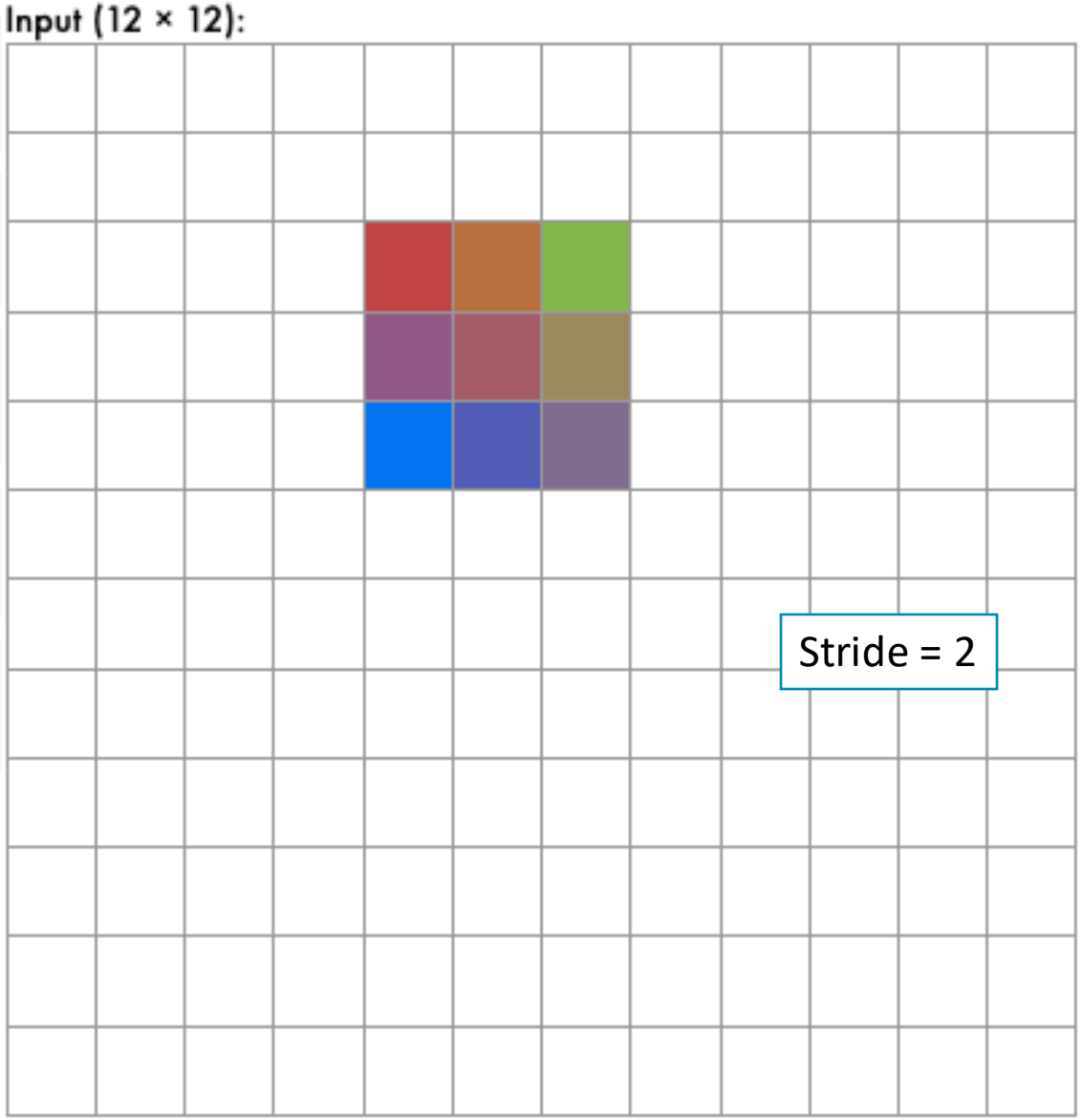


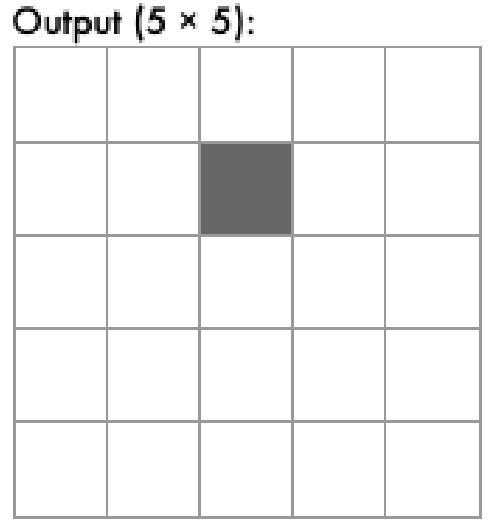




Convolution Visualizer

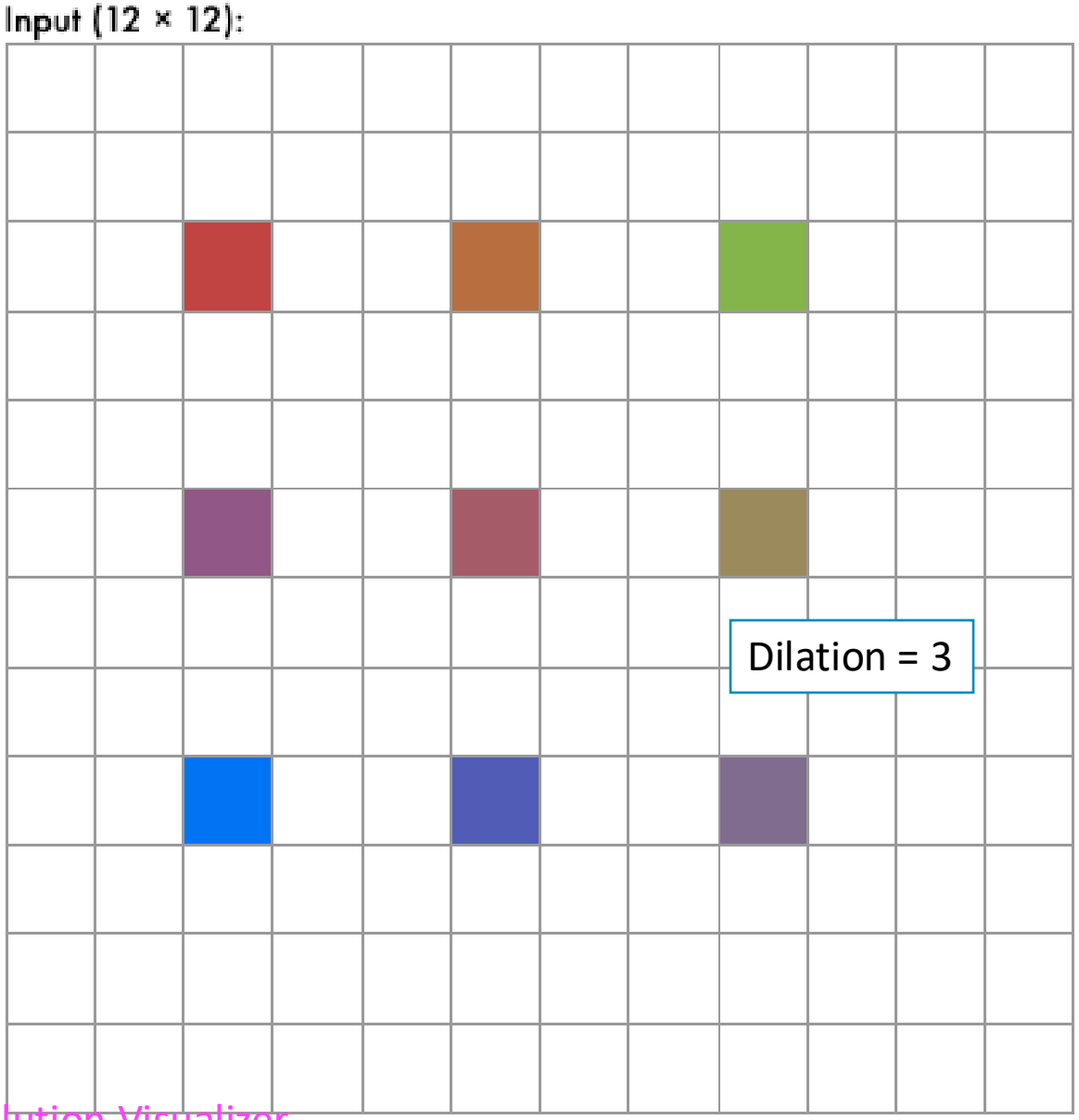
Convolution Hyperparameters: Stride (S)

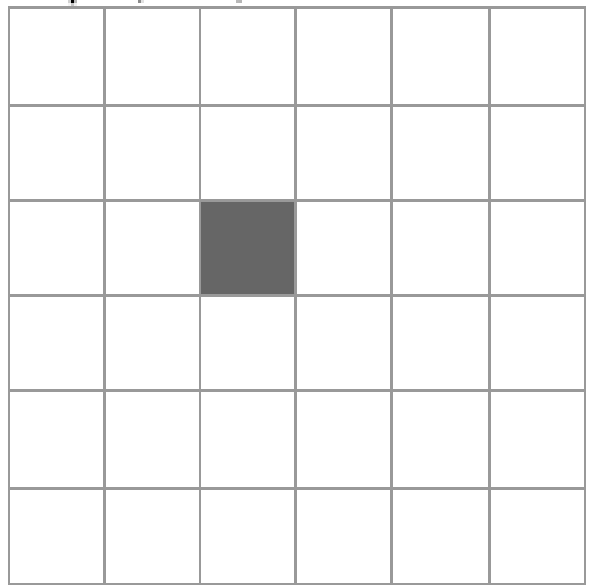




Convolution Visualizer

Convolution Hyperparameters: Dilation (D)





Convolution Visualizer

Computing Output Sizes

- Input shape: $(N, C_{in}, H_{in}, W_{in})$
- Output shape: $(N, C_{out}, H_{out}, W_{out})$

$$H_{out} = \left\lfloor \frac{H_{in} + 2P - D(K-1) - 1}{S} + 1 \right\rfloor$$

$$W_{out} = \left\lfloor \frac{W_{in} + 2P - D(K-1) - 1}{S} + 1 \right\rfloor$$

- Technically, you can have different values of padding, dilation, kernel size, and stride for each dimension

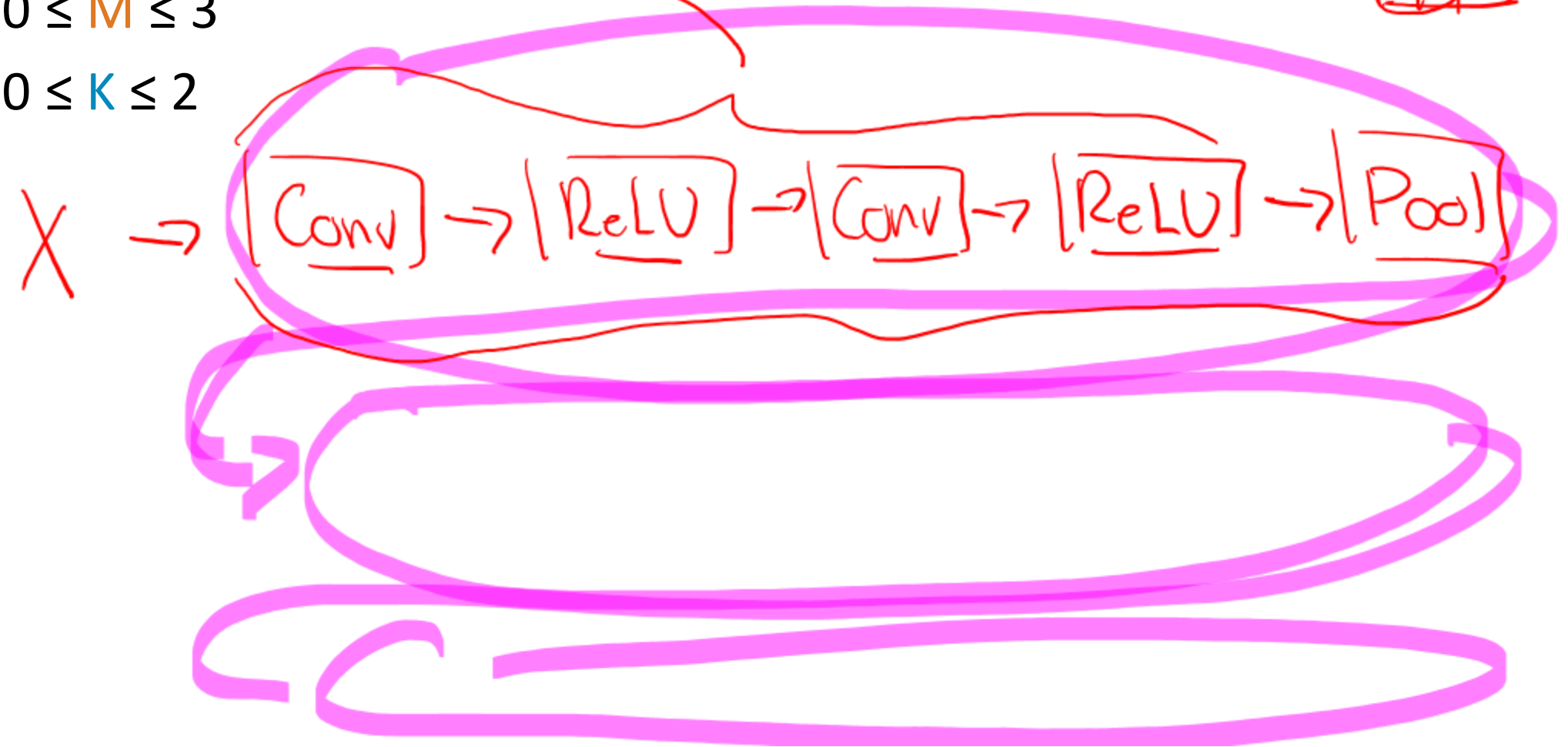
ConvNet Shape Calculator

The “Classic” (Old) CNN

AlexNet

Input -> [[Conv -> ReLU] x N -> Pool] x M -> [FC -> ReLU] x K -> FC -> Output

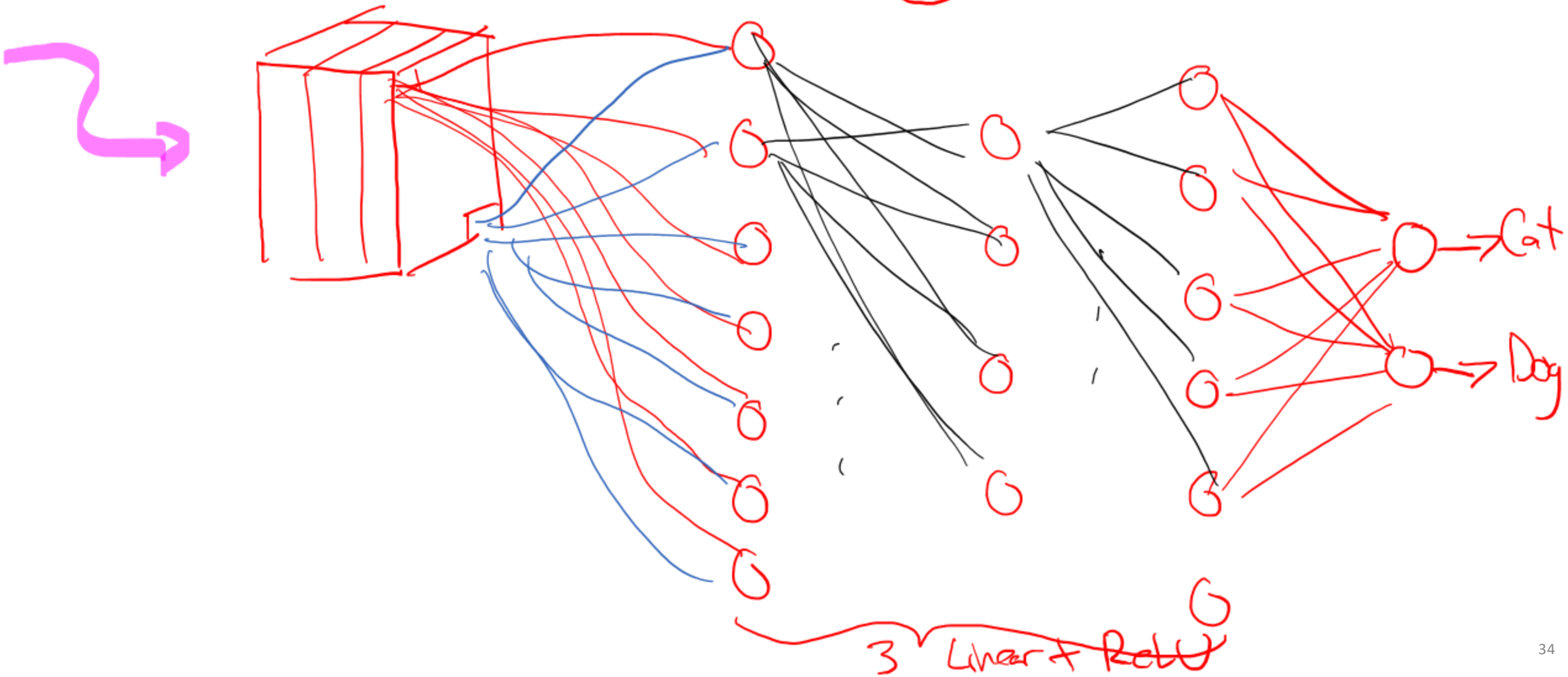
- $0 \leq N \leq 3$
- $0 \leq M \leq 3$
- $0 \leq K \leq 2$



Feature Detection

The “Classic” (Old) CNN

Input -> [[Conv -> ReLU] x N -> Pool] x M -> [FC -> ReLU] x K -> FC -> Output



```python
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()

        # Early CNNs had the following structure:
        #    X -> [[Conv2d -> ReLU] x N -> MaxPool2d] x M
        #      -> [Linear -> ReLU] x K -> Linear
        #   Where
        #     0 ≤ N ≤ 3
        #     0 ≤ M ≤ 3
        #     0 ≤ K < 3
        #
        # The "[[Conv2d -> ReLU] x N -> MaxPool2d] x M" part extracts
        # useful features, and the "[Linear -> ReLU] x K -> Linear" part
        # performs the classification.

        # Here is a network with N = 2, M = 2, K = 2

        # First [Conv2d -> ReLU] group of M=1
        self.conv1 = nn.Conv2d(3, 32, 3)
        self.conv2 = nn.Conv2d(32, 64, 3)

        # Second [Conv2d -> ReLU] group of M=1
        self.conv3 = nn.Conv2d(64, 128, 3)
        self.conv4 = nn.Conv2d(128, 256, 3)

        # First [Conv2d -> ReLU] group of M=2
        self.conv5 = nn.Conv2d(256, 128, 3)
        self.conv6 = nn.Conv2d(128, 64, 3)

        # Second [Conv2d -> ReLU] group of M=2
        self.conv7 = nn.Conv2d(64, 32, 3)
        self.conv8 = nn.Conv2d(32, 16, 3)

        # First [Linear -> ReLU] group of K=1
        # 64 comes from the shape and number of filters in the last conv layer
        self.fc1 = nn.Linear(64, 32)

        # Second [Linear -> ReLU] group of K=1
        self.fc2 = nn.Linear(32, 16)

        # And finally, the output linear layer
        self.fc3 = nn.Linear(16, num_classes)
```

```python
    def forward(self, X):
        # print("Input:", X.shape)
        X = F.relu(self.conv1(X))
        # print("After conv1:", X.shape)
        X = F.relu(self.conv2(X))
        # print("After conv2:", X.shape)
        X = F.max_pool2d(X, 2)
        # print("After max_pool:", X.shape)
        X = F.relu(self.conv3(X))
        # print("After conv3:", X.shape)
        X = F.relu(self.conv4(X))
        # print("After conv4:", X.shape)
        X = F.max_pool2d(X, 1)
        # print("After max_pool:", X.shape)
        X = F.relu(self.conv5(X))
        # print("After conv5:", X.shape)
        X = F.relu(self.conv6(X))
        # print("After conv6:", X.shape)
        X = F.max_pool2d(X, 1)
        # print("After max_pool:", X.shape)
        X = F.relu(self.conv7(X))
        # print("After conv7:", X.shape)
        X = F.relu(self.conv8(X))
        # print("After conv8:", X.shape)
        X = F.max_pool2d(X, 1)
        # print("After max_pool:", X.shape)
        X = X.view(X.shape[0], -1)
        # print("After reshape:", X.shape)
        X = F.relu(self.fc1(X))
        # print(X.shape)
        X = F.relu(self.fc2(X))
        # print(X.shape)
        X = self.fc3(X)
        # print(X.shape)

        return X
```

```
        # First [Conv2d -> ReLU] group of M=1
        self.conv1 = nn.Conv2d(3, 32, 3)
        self.conv2 = nn.Conv2d(32, 64, 3)

        # Second [Conv2d -> ReLU] group of M=1
        self.conv3 = nn.Conv2d(64, 128, 3)
        self.conv4 = nn.Conv2d(128, 256, 3)

        # First [Conv2d -> ReLU] group of M=2
        self.conv5 = nn.Conv2d(256, 128, 3)
        self.conv6 = nn.Conv2d(128, 64, 3)

        # Second [Conv2d -> ReLU] group of M=2
        self.conv7 = nn.Conv2d(64, 32, 3)
        self.conv8 = nn.Conv2d(32, 16, 3)

        # First [Linear -> ReLU] group of K=1
        # 64 comes from the shape and number of filters in the las
```

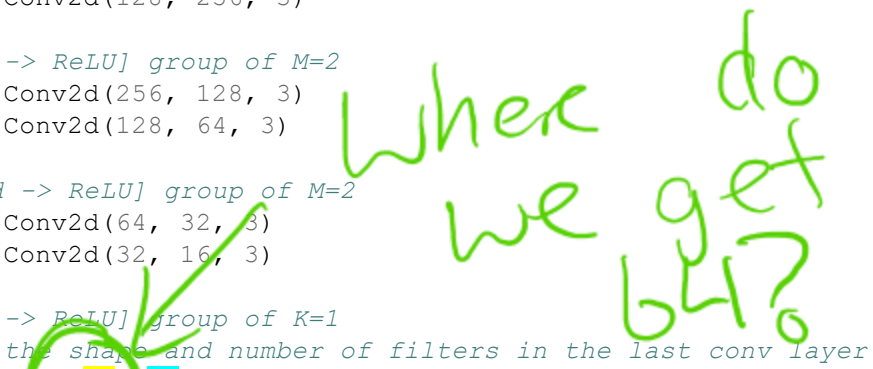
```
        # Second [Conv2d -> ReLU] group of M=2
        self.conv7 = nn.Conv2d(64, 32, 3)
        self.conv8 = nn.Conv2d(32, 16, 3)

        # First [Linear -> ReLU] group of K=1
        # 64 comes from the shape and number of filters in the las
        self.fc1 = nn.Linear(64, 32)

        # Second [Linear -> ReLU] group of K=1
        self.fc2 = nn.Linear(32, 16)

        # And finally, the output linear layer
        self.fc3 = nn.Linear(16, num_classes)
```

```
def forward(self, X):
    # print("Input:", X.shape)
    X = F.relu(self.conv1(X))
    # print("After conv1:", X.shape)
    X = F.relu(self.conv2(X))
    # print("After conv2:", X.shape)
    X = F.max_pool2d(X, 2)
    # print("After max_pool:", X.shape)
    X = F.relu(self.conv3(X))
    # print("After conv3:", X.shape)
    X = F.relu(self.conv4(X))
    # print("After conv4:", X.shape)
    X = F.max_pool2d(X, 1)
    # print("After max_pool:", X.shape)
    X = F.relu(self.conv5(X))
    # print("After conv5:", X.shape)
    X = F.relu(self.conv6(X))
    # print("After conv6:", X.shape)
    X = F.max_pool2d(X, 1)
    # print("After max_pool:", X.shape)
    X = F.relu(self.conv7(X))
    # print("After conv7:", X.shape)
    X = F.relu(self.conv8(X))
    # print("After conv8:", X.shape)
```

```
    # print("After conv6:", X.shape)
    X = F.max_pool2d(X, 1)
    # print("After max_pool:", X.shape)
    X = F.relu(self.conv7(X))
    # print("After conv7:", X.shape)
    X = F.relu(self.conv8(X))
    # print("After conv8:", X.shape)
    X = F.max_pool2d(X, 1)
    # print("After max_pool:", X.shape)
    X = X.view(X.shape[0], -1)
    # print("After reshape:", X.shape)
    X = F.relu(self.fc1(X))
    # print(X.shape)
    X = F.relu(self.fc2(X))
    # print(X.shape)
    X = self.fc3(X)
    # print(X.shape)

    return X
```

flatten

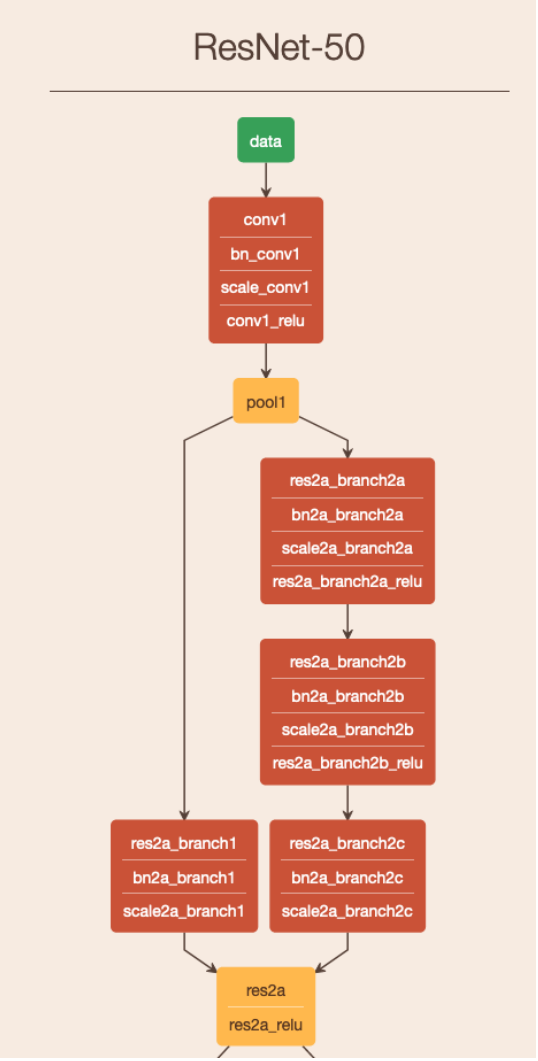
```python
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()

        # Early CNNs had the following structure:
        #    X -> [[Conv2d -> ReLU] x N -> MaxPool2d] x M
        #      -> [Linear -> ReLU] x K -> Linear
        #   Where
        #     0 ≤ N ≤ 3
        #     0 ≤ M ≤ 3
        #     0 ≤ K < 3
        #
        # The "[[Conv2d -> ReLU] x N -> MaxPool2d] x M" part extracts
        # useful features, and the "[Linear -> ReLU] x K -> Linear" part
        # performs the classification.

        # Here is a network with N = 2, M = 2, K = 2

        # First [Conv2d -> ReLU] group of M=1
        self.conv1 = nn.Conv2d(3, 32, 3)
        self.conv2 = nn.Conv2d(32, 64, 3)

        # Second [Conv2d -> ReLU] group of M=1
        self.conv3 = nn.Conv2d(64, 128, 3)
        self.conv4 = nn.Conv2d(128, 256, 3)

        # First [Conv2d -> ReLU] group of M=2
        self.conv5 = nn.Conv2d(256, 128, 3)
        self.conv6 = nn.Conv2d(128, 64, 3)

        # Second [Conv2d -> ReLU] group of M=2
        self.conv7 = nn.Conv2d(64, 32, 3)
        self.conv8 = nn.Conv2d(32, 16, 3)

        # First [Linear -> ReLU] group of K=1
        # 64 comes from the shape and number of filters in the last conv layer
        self.fc1 = nn.Linear(64, 32)

        # Second [Linear -> ReLU] group of K=1
        self.fc2 = nn.Linear(32, 16)

        # And finally, the output linear layer
        self.fc3 = nn.Linear(16, num_classes)
```




```python
    def forward(self, X):
        # print("Input:", X.shape)
        X = F.relu(self.conv1(X))
        # print("After conv1:", X.shape)
        X = F.relu(self.conv2(X))
        # print("After conv2:", X.shape)
        X = F.max_pool2d(X, 2)
        # print("After max_pool:", X.shape)
        X = F.relu(self.conv3(X))
        # print("After conv3:", X.shape)
        X = F.relu(self.conv4(X))
        # print("After conv4:", X.shape)
        X = F.max_pool2d(X, 1)
        # print("After max_pool:", X.shape)
        X = F.relu(self.conv5(X))
        # print("After conv5:", X.shape)
        X = F.relu(self.conv6(X))
        # print("After conv6:", X.shape)
        X = F.max_pool2d(X, 1)
        # print("After max_pool:", X.shape)
        X = F.relu(self.conv7(X))
        # print("After conv7:", X.shape)
        X = F.relu(self.conv8(X))
        # print("After conv8:", X.shape)
        X = F.max_pool2d(X, 1)
        # print("After max_pool:", X.shape)
        X = X.view(X.shape[0], -1)
        # print("After reshape:", X.shape)
        X = F.relu(self.fc1(X))
        # print(X.shape)
        X = F.relu(self.fc2(X))
        # print(X.shape)
        X = self.fc3(X)
        # print(X.shape)

        return X
```

Exploring More Modern Architectures

- 2010: ImageNet Challenge Starts
- 2011: Shallow approaches win with lower accuracy
- 2013: 8-Layer network, AlexNet, wins with 63% (60M parameters)
- 2014: VGG-16 with 74% (138M parameters)
- 2015: ResNet-152 with 78% (66.8M parameters)
- 2016: Inception with 80% (55.8M parameters)
- 2017: NASNET with 82% (88.9M parameters)
- 2018: ResNeXt with 85% (829M parameters)
- 2019: EfficientNet with 87% (19M parameters)
- 2020: EfficientNet-L2 with 90% (480M parameters)
- 2021: ViT-G/14 with 90.5% (1843M parameters)
- 2022: CoCa with 91% (2100M parameters)
- 2023: Basic-L with 91.1% (2440M parameters)
- (Nothing better in 2024 or 2025 to date)



https://dgschwend.github.io/netscope/quickstart.html

https://paperswithcode.com/sota/image-classification-on-imagenet

Data Augmentation

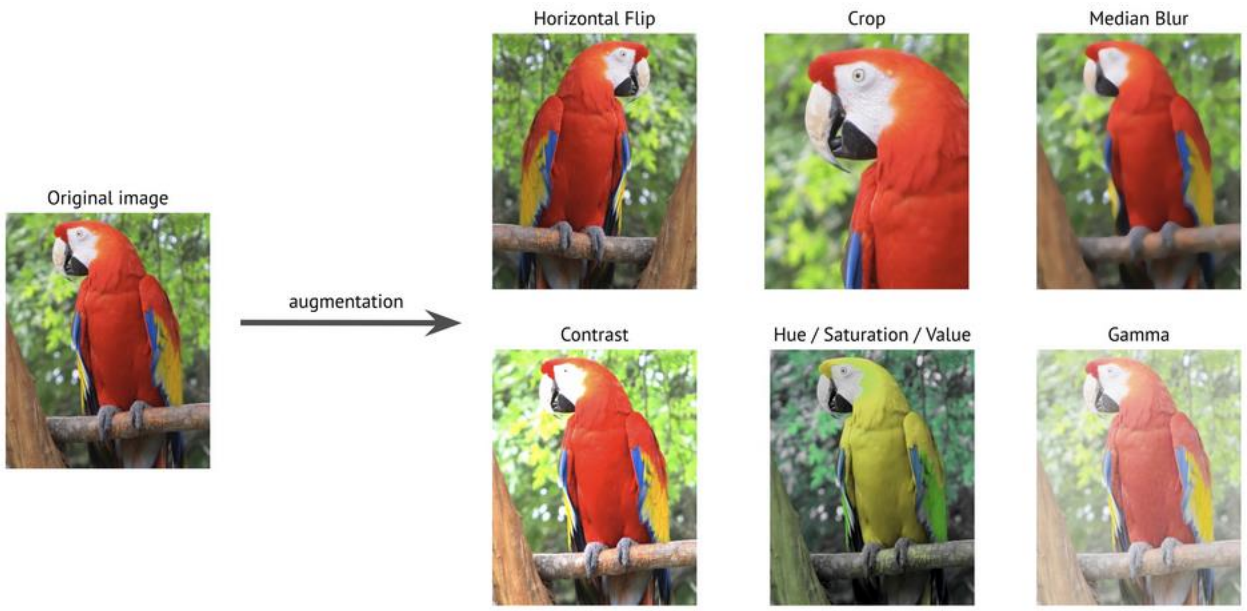
https://albumentations.ai/

Original

Mirrored

Rotated

Brighter

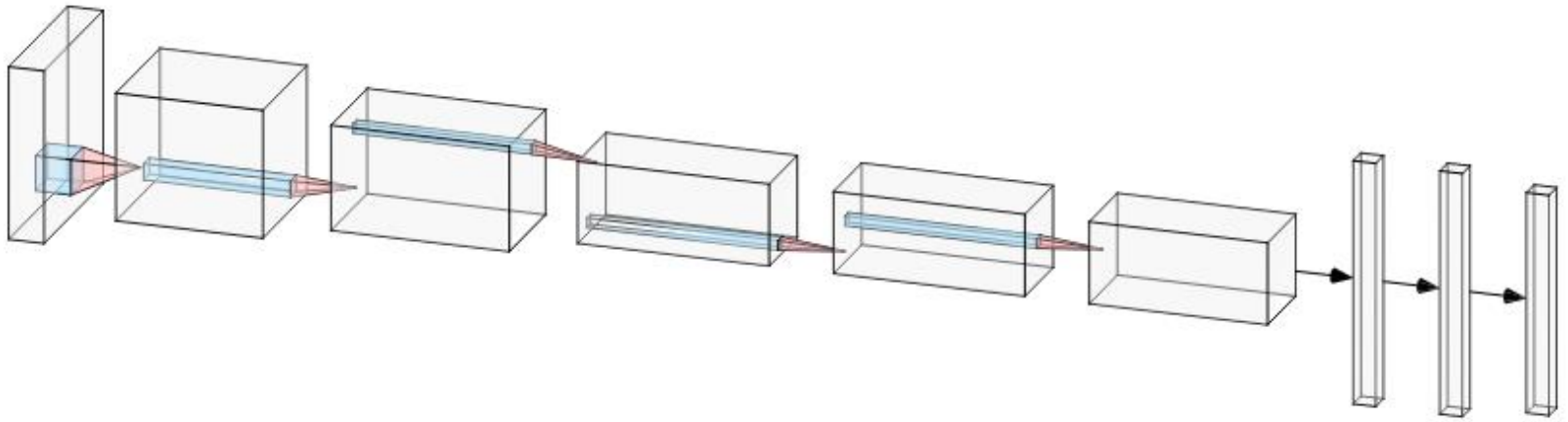



```
for epoch in range(num_epochs):
    model.train()
    for X, y in train_loader:
        yhat = model(X)
        loss = criterion(y, yhat)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    model.eval()
    with torch.no_grad():
        for X, y in valid_loader:
            yhat = model(X)
            loss = criterion(y, yhat)
            metric = metrics(y, yhat, model)
```

Transfer Learning

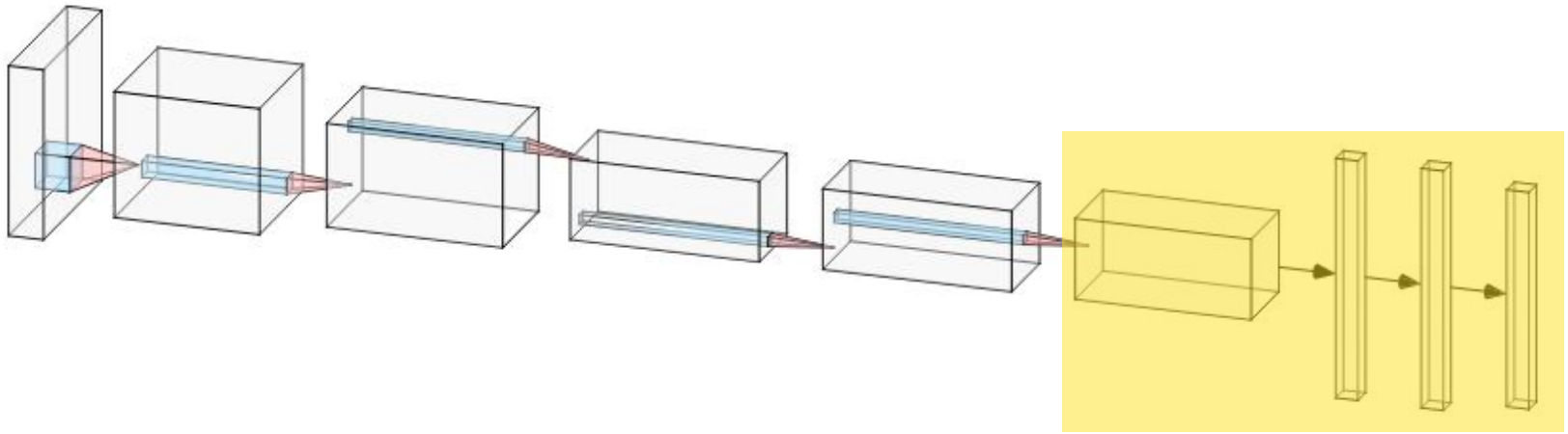
A process to more quickly train a network for a new problem

1. Train a neural network for hours/days/weeks/months
2. Save and share the learned parameters
3. Replace parameters in the final layers with random values
4. “Finetune” parameters on a new problem



Transfer Learning, Robotics Example

- Train a ResNet-18 model on the ImageNet task ("…more than 14 million images have been hand-annotated by the project to indicate what objects are pictured and in at least one million of the images, bounding boxes are also provided. ImageNet contains more than 20,000 categories." – Wikipedia)



- Replace the final layers with randomly initialized layers
- Finetune the network on robot navigation

Summary

- Some input data has local relationships among values
- Fully connected networks must learn this relationship from scratch
- Convolutions leverage such relationships by default
- Use convolutional layers when your input data exhibits locality
- Default to using data augmentation and transfer learning

Resources / Links

- Image Kernels explained visually
- CS 230 - Convolutional Neural Networks Cheatsheet
- CS231n Convolutional Neural Networks for Visual Recognition
- A guide to convolution arithmetic for deep learning
- Intuitively Understanding Convolutions for Deep Learning
- A technical report on convolution arithmetic in the context of deep learning
- What Are Convolutional Neural Networks?
- Computing Receptive Fields of Convolutional Neural Networks