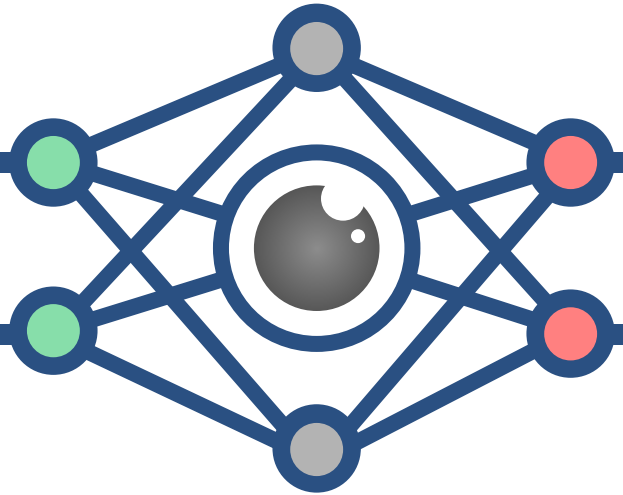
CS3485

Deep Learning for Computer Vision



Lec 10: Inception Net and what CNNs learn

Announcements

- Quiz should be available on Canvas!

(Tentative) Lecture Roadmap

Basics of Deep Learning

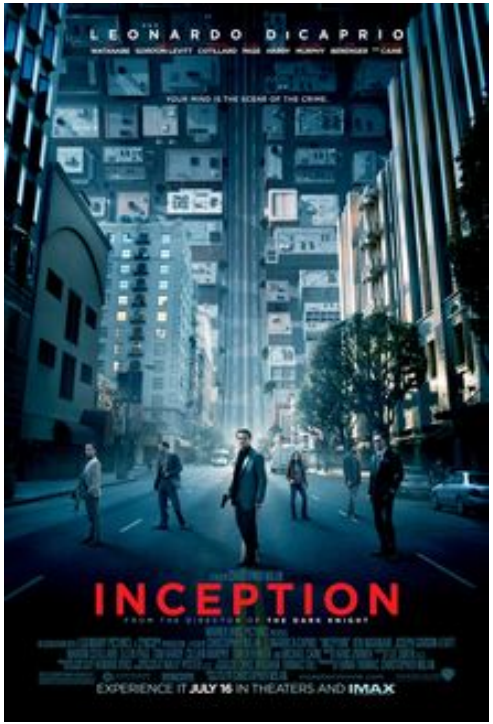
- Intro to Computer Vision
- Linear Classifiers and Perceptron
- Multilayer Perceptron
- Optimization in Deep Learning
- Pytorch I – MLPs
- Pytorch II – Images and Regularization
- Convolutional Neural Networks
- Data Augmentation and Deep CNNs
- Transfer Learning and Residual Nets
- Inception Net and what CNNs learn
- Adversarial Examples and Self-supervision

Deep Learning and Computer Vision in Practice

- Intro to MLOps
- Intro to Object Detection
- Fast Object Detection
- Intro to Image Segmentation
- Applications of Detection and Segmentation
- Autoencoders
- Image Generation with GANs
- Advanced GANs
- The Attention Mechanism
- Transformers and ChatGPT
- Image Generation by Prompt
- Misc. Topics

Going Deeper with Convolutions

- In the previous lectures we saw that the most straightforward way of improving the performance of deep neural networks is by increasing their depth.
- However, this will result in
  - The vanishing gradient problem,
  - A dramatic increase in network parameters and use of computational resources.
- Besides ResNets, another architecture that overcame those issues is called the Inception Network, from the paper “Going Deeper with Convolutions” from 2015.
- The network was named after the (great) movie of same name and has the same “going deeper” premise.



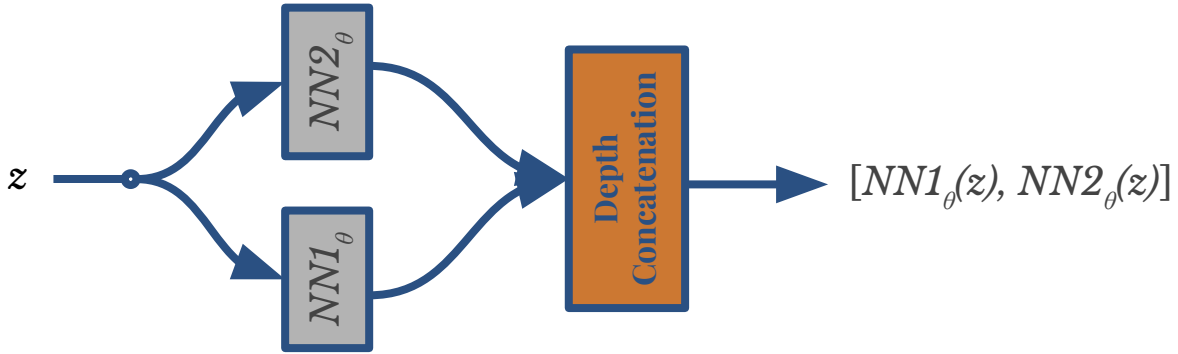
Why the Inception Network?

- Important parts in the image for classification can have **extremely large variation in size**.
- For instance, an image with a dog can be either of the options on the right. The area where the dog is in is different in each image.

- Because of this huge variation in the location of the information, choosing **the right kernel size** for the convolution operation becomes tough:
  - A **larger kernel** is preferred for information that is distributed more **globally** (like the first image above)
  - A **smaller kernel** is preferred for information that is distributed more **locally** (like the last one).
- However, naively stacking large convolution operations (large kernels) is **computationally expensive**.

Branching Networks

- The solution the InceptionNet found for those issues use **branching networks** that generalize Residual Layers.
- It recurs to a **depth concatenation** step, that takes the tensors coming from two different branches and outputs a tensor with its channels concatenated. For example:

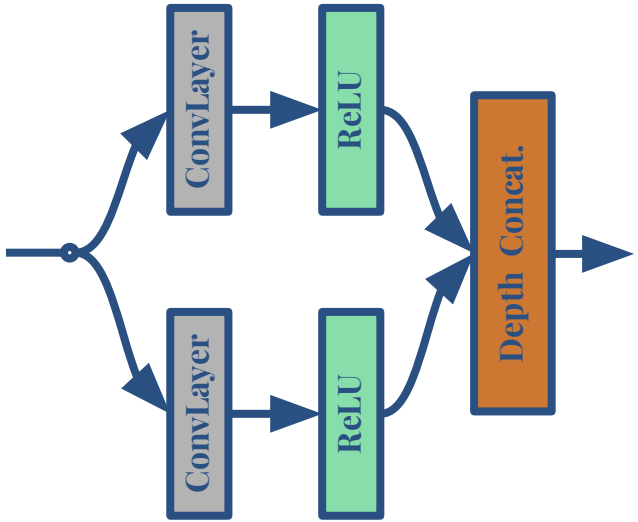


- Note that the other dimensions (height and width) of the output tensors from $NN1_\theta$ and $NN2_\theta$ **have to be the same**, so the concatenation operation can make sense.

Branching Networks in PyTorch

- In PyTorch, we can easily code up branching networks via the function `torch.cat()` that concatenates two tensors.
- An example of custom layer that uses concatenation is in the following example:

```
import torch.nn as nn

class BranchingLayer(nn.Module):
    def __init__(self, in_ch, out_ch1, out_ch2, kernel_size):
        super().__init__()
        self.conv1 = nn.Conv2d(in_ch, out_ch1, kernel_size)
        self.relu1 = nn.ReLU()
        self.conv2 = nn.Conv2d(in_ch, out_ch2, kernel_size)
        self.relu2 = nn.ReLU()

    def forward(self, x):
        x1 = self.relu1(self.conv1(x))
        x2 = self.relu2(self.conv2(x))
        return torch.cat([x1, x2], dim=1)
```

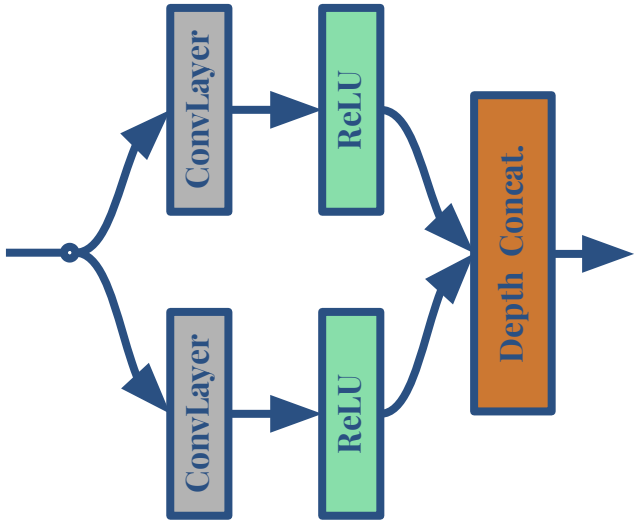


Branching Networks in PyTorch

- In PyTorch, we can easily code up branching networks via the function `torch.cat()` that concatenates two tensors.
- An example of custom layer that uses concatenation is in the following example:

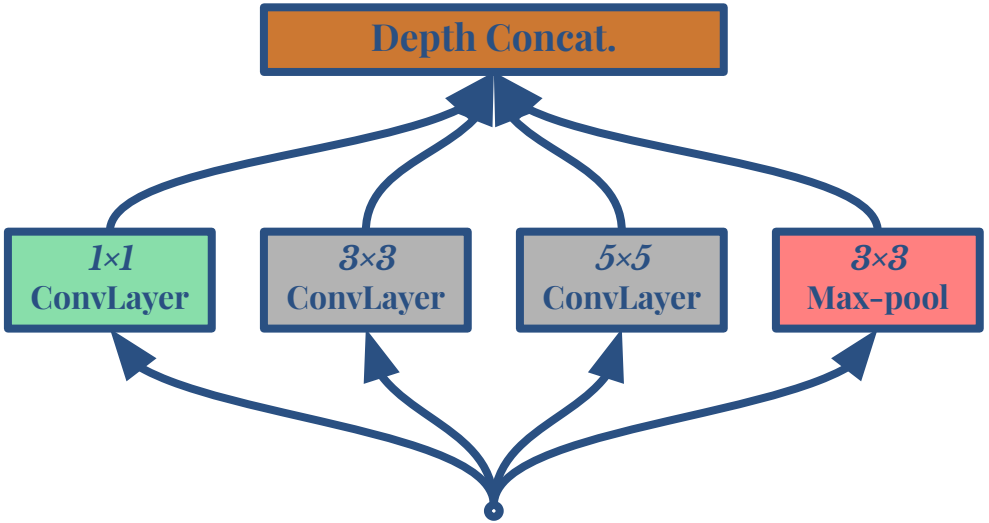
```
import torch.nn as nn

class BranchingLayer(nn.Module):
    def __init__(self, in_ch, out_ch1, out_ch2, kernel_size):
        super().__init__()
        self.conv1 = nn.Conv2d(in_ch, out_ch1, kernel_size)
        self.relu1 = nn.ReLU()
        self.conv2 = nn.Conv2d(in_ch, out_ch2, kernel_size)
        self.relu2 = nn.ReLU()

    def forward(self, x):
        x1 = self.relu1(self.conv1(x))
        x2 = self.relu2(self.conv2(x))
        return torch.cat([x1, x2], dim=1)
```

The filters are in the tensor's second dimension. Remember: (N, C, H, W).



The Inception Module (Naive)

- Using branching networks, why not not have filters with **multiple sizes** operate on the same level instead of stacking them up?
- With that, we'd avoid the expensive staking and could also use global and local ConvLayers **on the same input**.
- The Inception Module (on the right) was then created with that exact purpose. It has:
  - A more local set of filters in the 3×3 ConvLayer,
  - A more global set in the 5×5 one.
  - A Max-pool layer (which is global).
- The 1×1 ConvLayer serves another purpose.

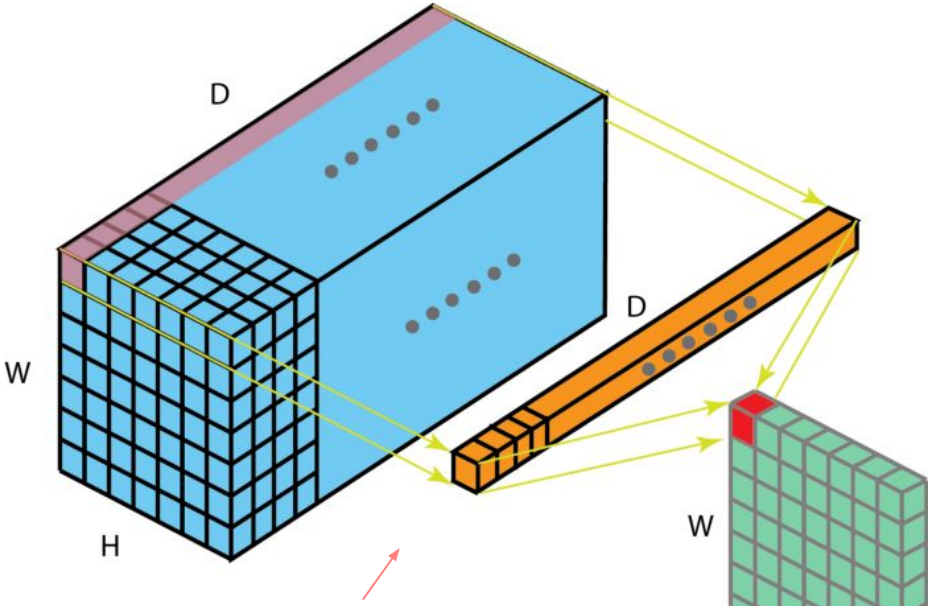


The *1×1* Convolution

- An important piece in the Inception module is the *1×1* Convolution.
- We use it when we want to **reduce the depth and keep the height × width** of the feature maps.
- This effect of channel down-sampling is called "**Dimensionality Reduction**".
- For example, in PyTorch, we can create such a module as following:

```
nn.Conv2d(in_ch, out_ch, kernel_size=1)
```

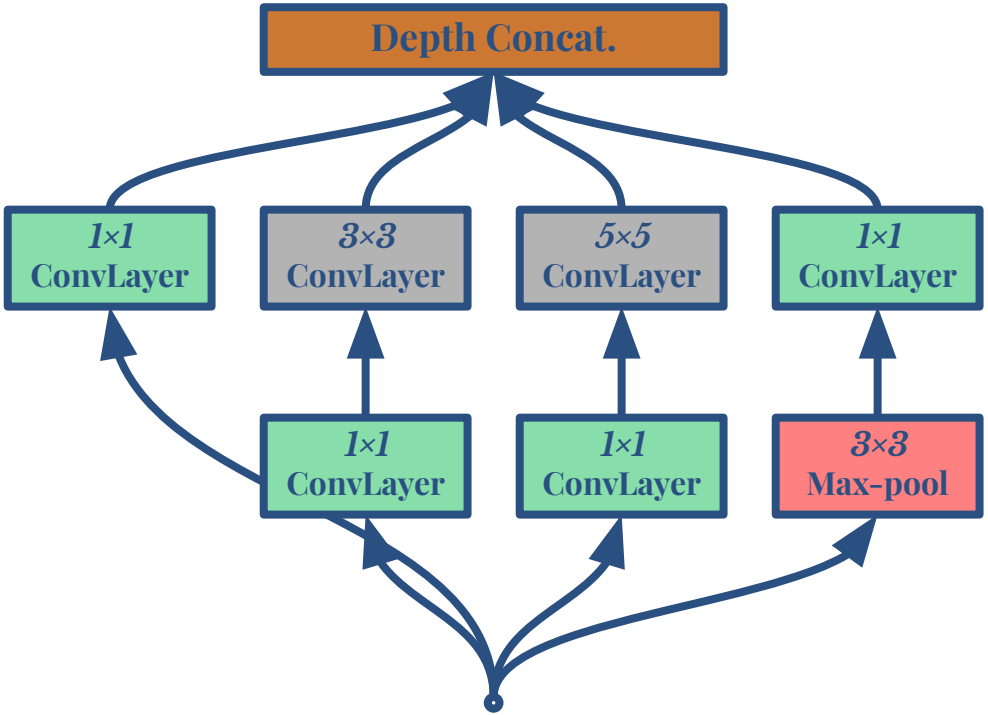
where the feature map's depth changes from `in_ch` to `out_ch`.



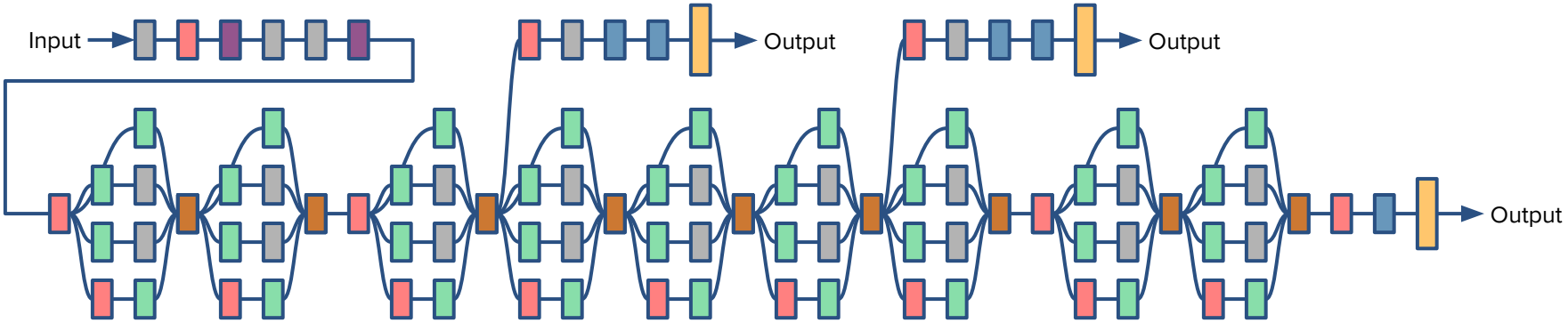
This is the execution of **one** *1×1* Convolution filter, usually you will have more than just one filter.

The Inception Module (Improved)

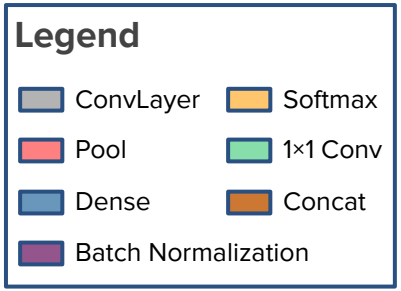
- This filter dimensionality reduction is very handy **for larger convolution layers**, such as the 5×5 ones, that require many weights to be learned for each filter.
- It is so much so that Inception Net uses many 1×1 convolutions on its final inception module.
- By reducing the number of channels in each tensor before the “real” ConvLayers, we can control the network’ computational expense.



The Inception Network

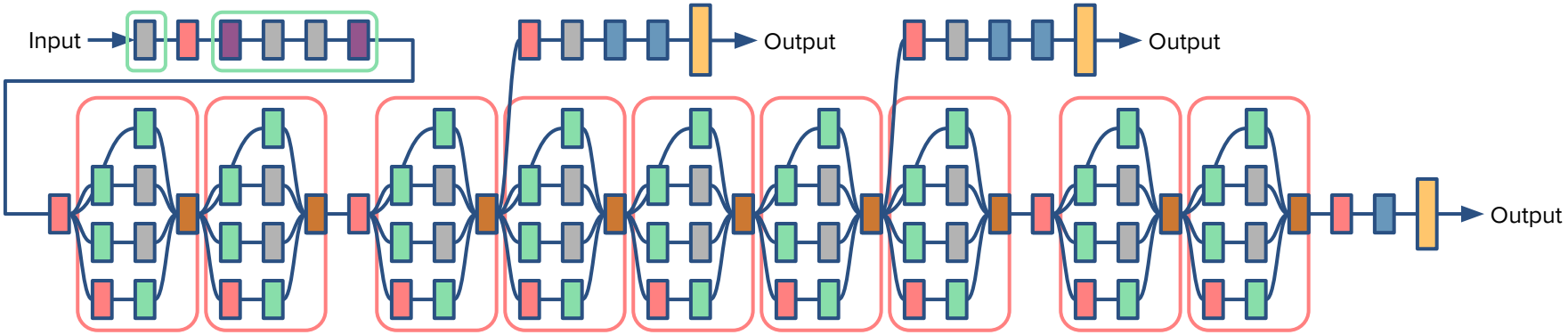


- The InceptionNet (also called Inception v1 and **GoogLeNet***) is shown above.
- During training, it has *three classifiers*, all with the same output data. During testing, only the rightmost classifier is used.
- This is done to train the inner layers and avoid vanishing gradients.



*The name is also an hommage to one of Deep Learning's pioneers, Yann **Le**Cun

The Inception Network



- The network uses 9 Inception modules in three different stages, separated by max-pooling layers.
- Furthermore, it has 2 “preprocessing” convolutional layer stages to prepare the data for the upcoming inception modules.
- In total it is 22 layers deep (27, including the pooling layers)

Legend

ConvLayer	Softmax
Pool	1×1 Conv
Dense	Concat
Batch Normalization	

Versions of Inception and Performance

- This network (Inception v1), when trained on ImageNet achieved a pretty good **9.2% Top-5 error rate**.
- Inception v2 and Inception v3 were presented in the same paper and added a number of tricks which increased v1's accuracy and reduced the computational complexity.
- Most of the important upgrades concerned the Inception Module itself, for example:
  - In v2, they factorize the 5×5 convolution to two stacked 3×3 convolution operations to improve computational speed. A 5×5 convolution is 2.78 times more expensive than a 3×3 convolution.
  - Furthermore, they factorize $n\times n$ convolutions to a combination of $1\times n$ and $n\times1$ convolutions (a 3×3 conv. is equivalent to performing a 1×3 conv followed by a 3×1 one). This makes the overall cost 33% cheaper.
  - In v3, they included (factorized) 7×7 convolutions.
- This lead to a **5.6% Top-5 error rate** (for Inception v3) in ImageNet.
- Furthermore, there is the Inception V4 and Inception-ResNet architectures.

Inception Network in PyTorch

- Just like the VGG and ResNet architectures, PyTorch also makes the Inception v3 network available to use with pre-trained ImageNet weights.
- This is done as follows (which analogous to VGG and ResNet):

```
from torchvision.models import inception_v3, Inception_V3_Weights
model = inception_v3(weights=Inception_V3_Weights.IMAGENET1K_V1)
```

- And the model's summary:

```
from torchsummary import summary
summary(model.to(device), (3, 299, 299))
```

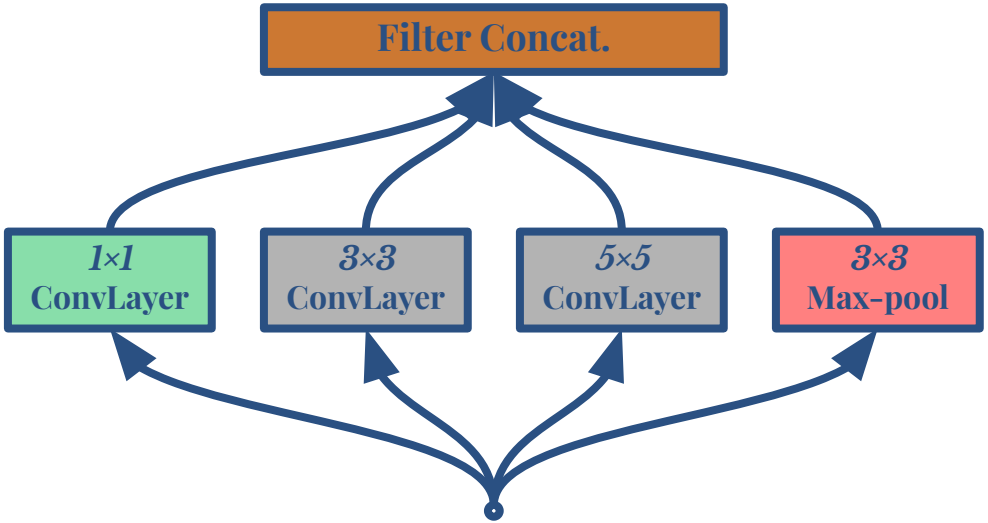
One should keep in mind that the Inception Networks were originally trained on 299×299 RGB images.

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1         [-1, 32, 149, 149]             864
               (...)                      (...)             (...)
      InceptionE-303           [-1, 2048, 8, 8]               0
AdaptiveAvgPool2d-304           [-1, 2048, 1, 1]               0
         Dropout-305           [-1, 2048, 1, 1]               0
          Linear-306                 [-1, 1000]       2,049,000
================================================================
Total params: 27,161,264
Trainable params: 27,161,264
Non-trainable params: 0
----------------------------------------------------------(...)
```

Relatively fewer weights than the VGGs

Exercise (*In pairs*)

- In PyTorch, code up the naive inception block on the right. Remember that the tensor sizes at the depth concatenation should be the same. How would you do it for each layer in this block? Call this module `InceptionModule`. *Hint*: to try out if your network "works", try to print its summary as you go:




```
from torchsummary import summary

model = InceptionModule(1, 64, 128, 32)
summary(model.to(device),(1,28,28))
```

Solution

- Here is the solution for the previous exercise:

```
import torch.nn as nn
import torch
from torchsummary import summary
device = 'cuda' if torch.cuda.is_available()
                else 'cpu'

class ConvLayer(nn.Module):
   def __init__(self, in_channels,
                      out_channels,
                      **kwargs):
       super().__init__()
       self.relu = nn.ReLU()
       self.conv = nn.Conv2d(in_channels,
                             out_channels,
                             **kwargs)

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

```
class InceptionModule(nn.Module):
   def __init__(self, in_ch, out_ch_1x1, out_ch_3x3, out_ch_5x5):
       super().__init__()
       self.branch1 = ConvLayer(in_ch, out_ch_1x1,
                                kernel_size=1, padding=0)

       self.branch2 = ConvLayer(in_ch, out_ch_3x3,
                                kernel_size=3, padding=1)

       self.branch3 = ConvLayer(in_ch, out_ch_5x5,
                                kernel_size=5, padding=2)

       self.branch4 = nn.MaxPool2d(kernel_size=(3, 3),
                                   stride=1, padding=1)

   def forward(self, x):
       return torch.cat([self.branch1(x), self.branch2(x),
                         self.branch3(x), self.branch4(x)], dim=1)
```

- Note that the ConvLayer step should **always have a ReLU** after the convolutional layer.

Solution (Cont.)

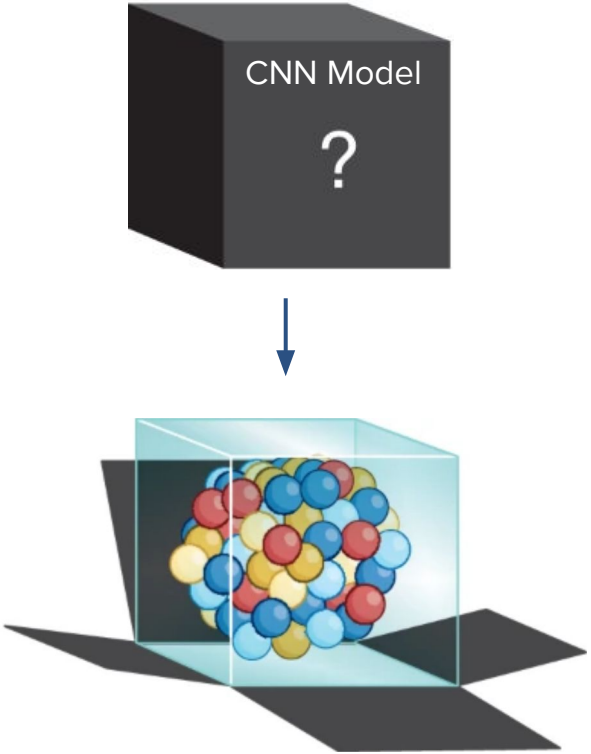
- Finally, on the right, we have the summary of the network:

```
from torchsummary import summary

model = InceptionModule(1, 64, 128, 32)
summary(model.to(device),(1,28,28))
```

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1          [-1, 64, 28, 28]             128
              ReLU-2          [-1, 64, 28, 28]               0
         ConvLayer-3          [-1, 64, 28, 28]               0
            Conv2d-4         [-1, 128, 28, 28]           1,280
              ReLU-5         [-1, 128, 28, 28]               0
         ConvLayer-6         [-1, 128, 28, 28]               0
            Conv2d-7          [-1, 32, 28, 28]             832
              ReLU-8          [-1, 32, 28, 28]               0
         ConvLayer-9          [-1, 32, 28, 28]               0
        MaxPool2d-10           [-1, 1, 28, 28]               0
================================================================
Total params: 2,240
Trainable params: 2,240
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.00
Forward/backward pass size (MB): 4.03
Params size (MB): 0.01
Estimated Total Size (MB): 4.04
----------------------------------------------------------------
```

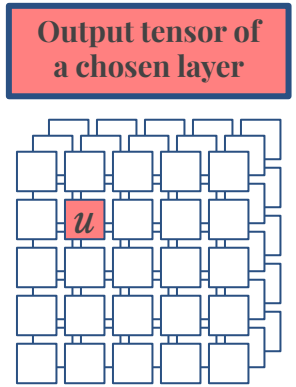
What do CNNs learn?

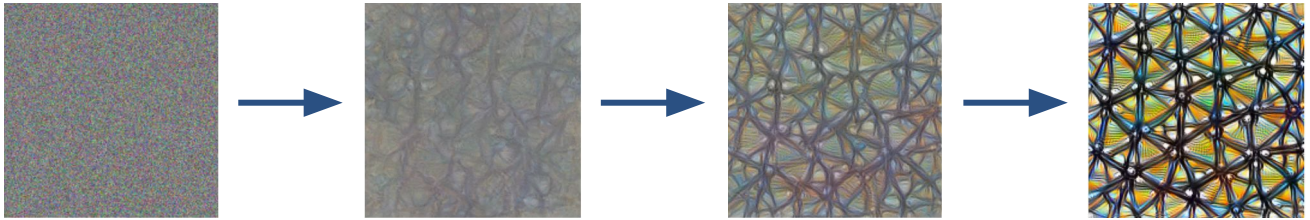
- So far, we saw that Deep Convolutional Nets are very good at learning visual features that improve image classification.
- But **what do they learn after all**? What is behind the black box that is Deep Learning?
- In fact, there is a growing sense that neural networks need to be **interpretable to humans**, so it can be used more efficiently and securely.
- The field of **Network Interpretability** or **Explainable AI** has formed in response to such concerns.
- We'll use Inception v1/GoogLeNet trained on ImageNet as our running example to understand this learning.



Understanding Feature Learning via Optimization

- A way to see what the network is learning is via **feature visualization***, i.e., we'll visualize what features it in fact learned.
- This means that we'll:
  - Choose a certain unit u in a certain layer we want to understand,
  - Find the input image I that causes u to have maximum activation.
- To find I, we use **optimization**: starting from a image with random pixel values, change it via Gradient Descent so to maximize the output of u.

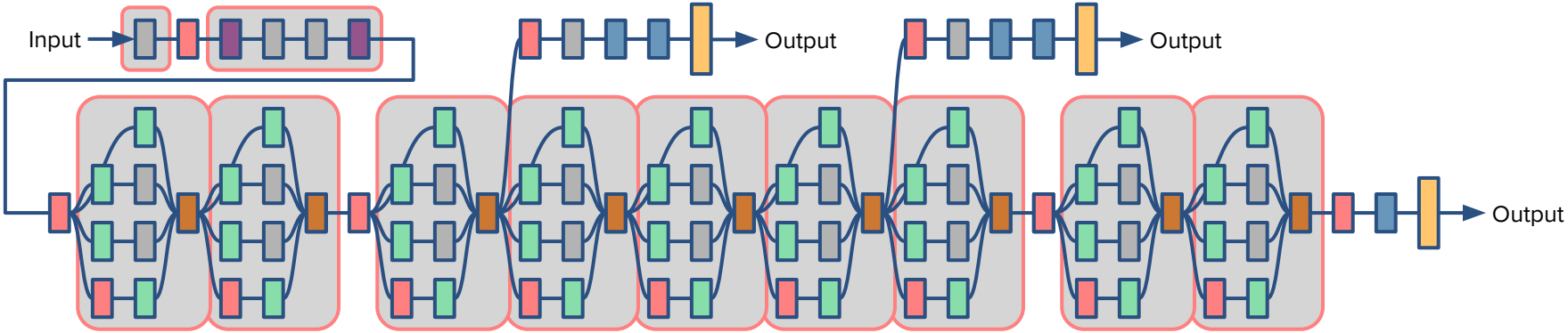




*The images shown here were taken from this (very good) article on feature visualization in deep nets.

What happens inside the Inception Network?

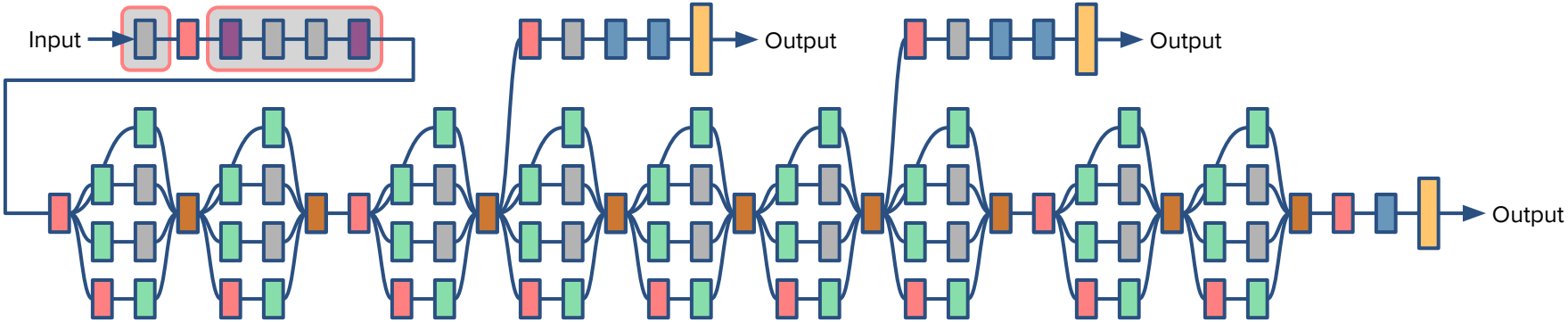
- We can use this optimization process to visualize what GoogLeNet learns at each stage.



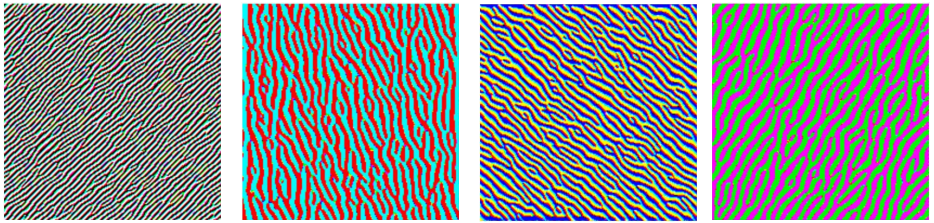
- To do so, we'll divide up the network in *11* stages/blocks and show what images some of the units in each stage most strongly react to.
- This will give us a hint of what these filters (and blocks) learned when trained on the ImageNet database.

What happens inside the Inception Network?

- We can now go block by block:

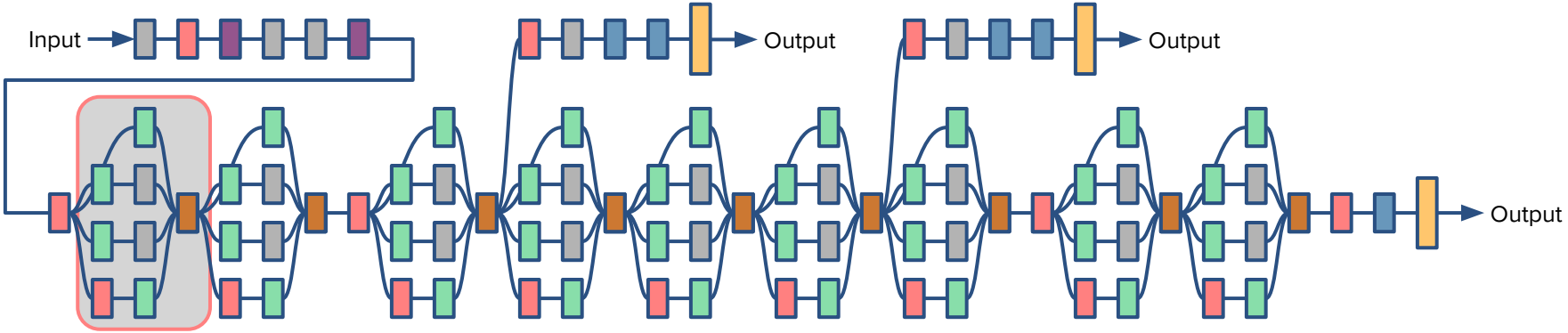


Block 1 and 2: Without any inception layer here (just simple ConvLayers), it mostly reacts to straight-ish meaningless patterns (like *edges*)

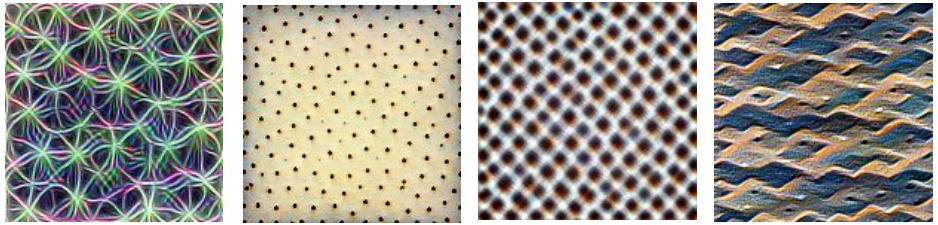


What happens inside the Inception Network?

- We can now go block by block:

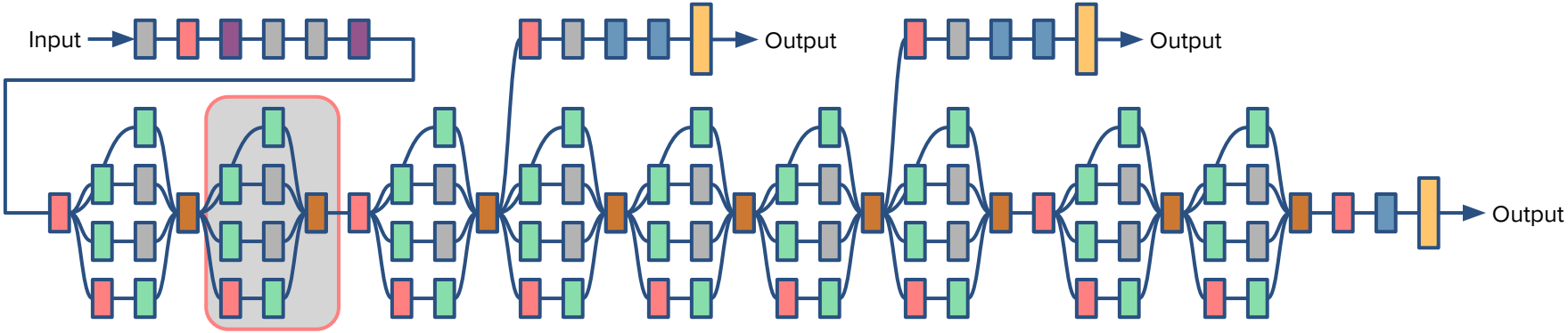


Block 3a: This is the first inception block. It shows some quite interesting *textures* that are very localized.

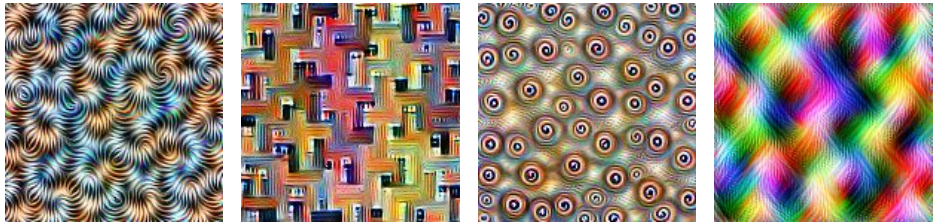


What happens inside the Inception Network?

- We can now go block by block:

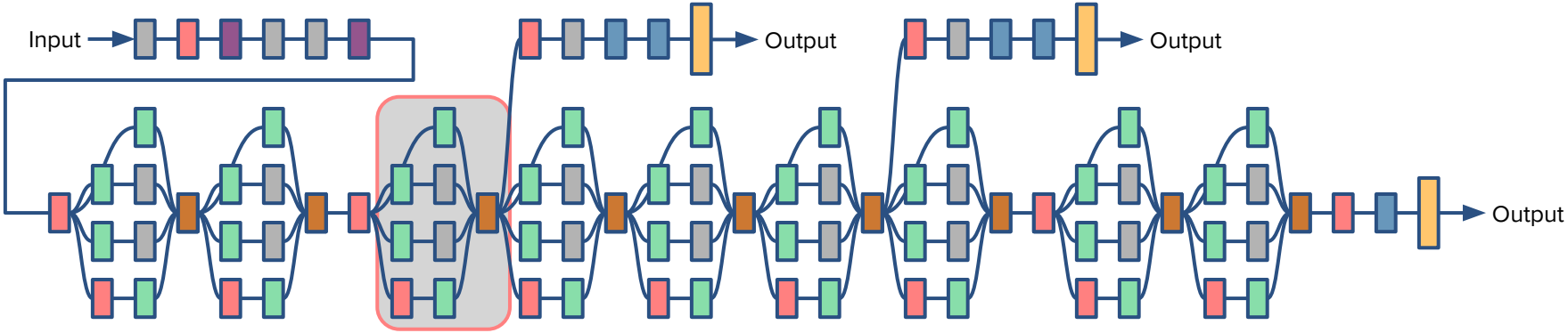


Block 3b: Textures start to become more complex and larger/global and display shapes that look more detailed.

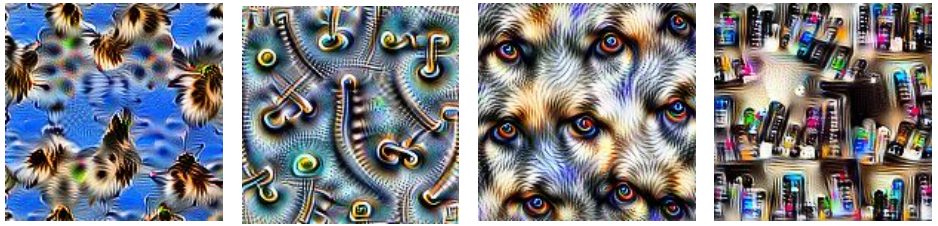


What happens inside the Inception Network?

- We can now go block by block:

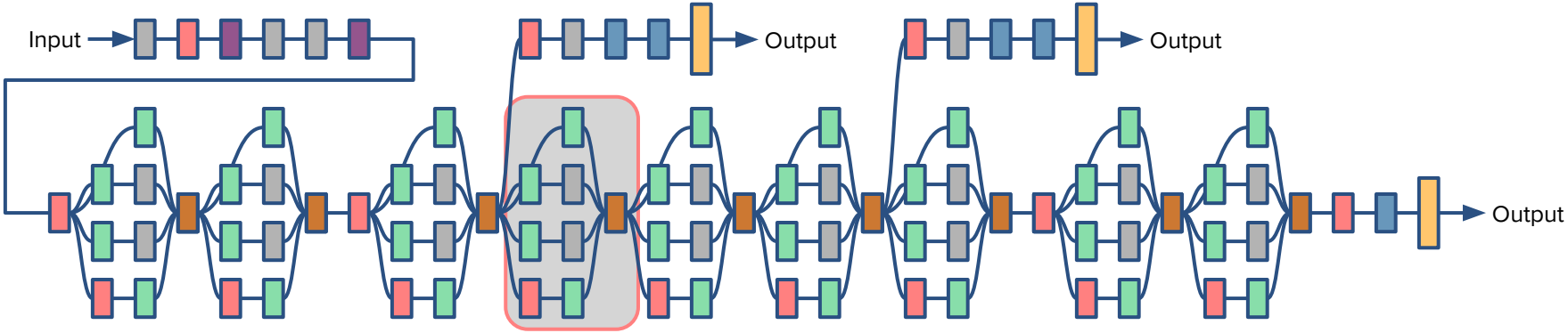


Block 4a: In this layer, which follows a pooling step, we begin to see more complex patterns, and even *object parts*.

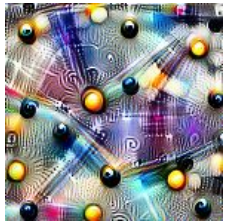


What happens inside the Inception Network?

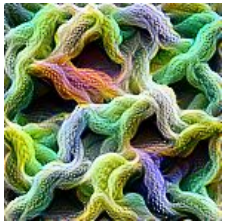
- We can now go block by block:



Block 4b: You can begin to make out parts of objects and visualizations start having more context (like trees in front of sky and ground).


Billiards balls

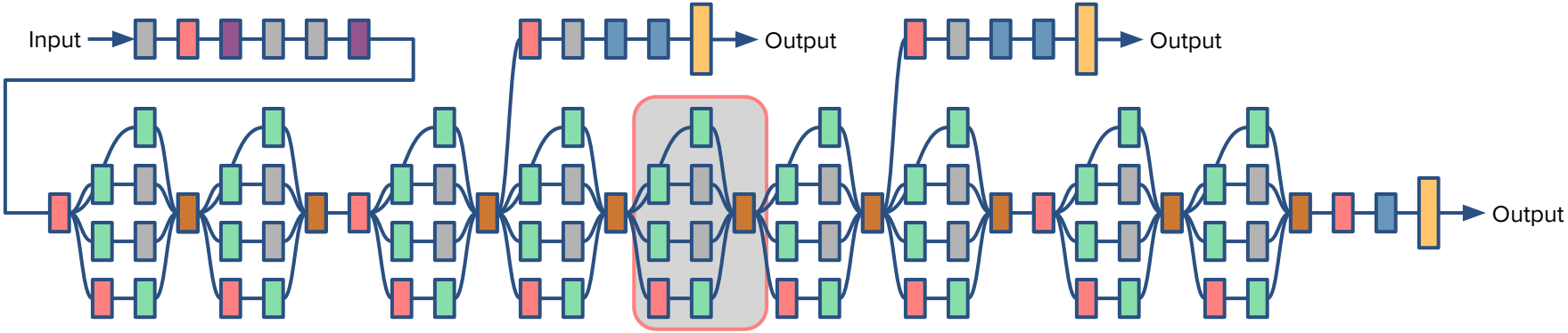

Trees


Fluffy ropes


Architecture

What happens inside the Inception Network?

- We can now go block by block:



Block 4c: In this layer, you can find units responding to specific *objects* like dogs on leashes or wheels.


House

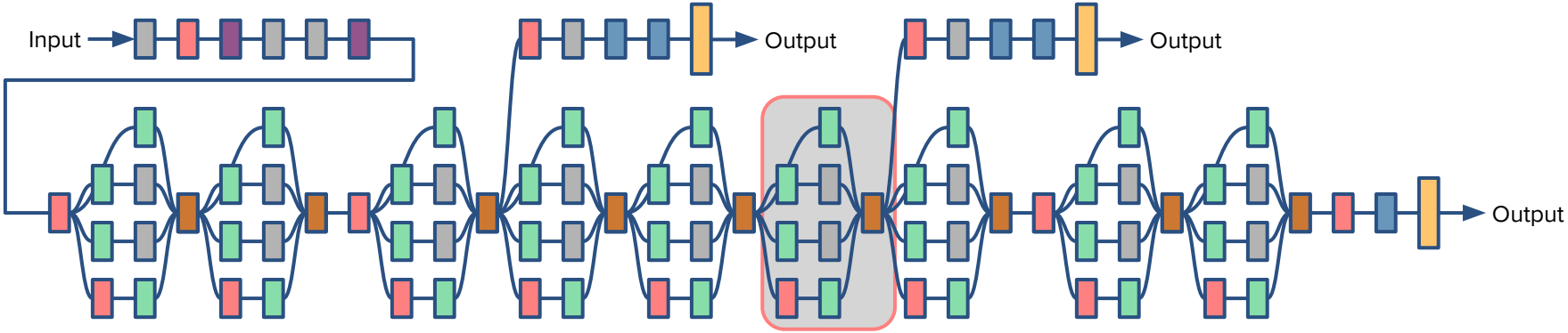

Dogs on leash


Wheels


Palm Trees

What happens inside the Inception Network?

- We can now go block by block:



Block 4d: By this layer we find more sophisticated concepts, like a particular kind of animal snout or snake heads


Dishes

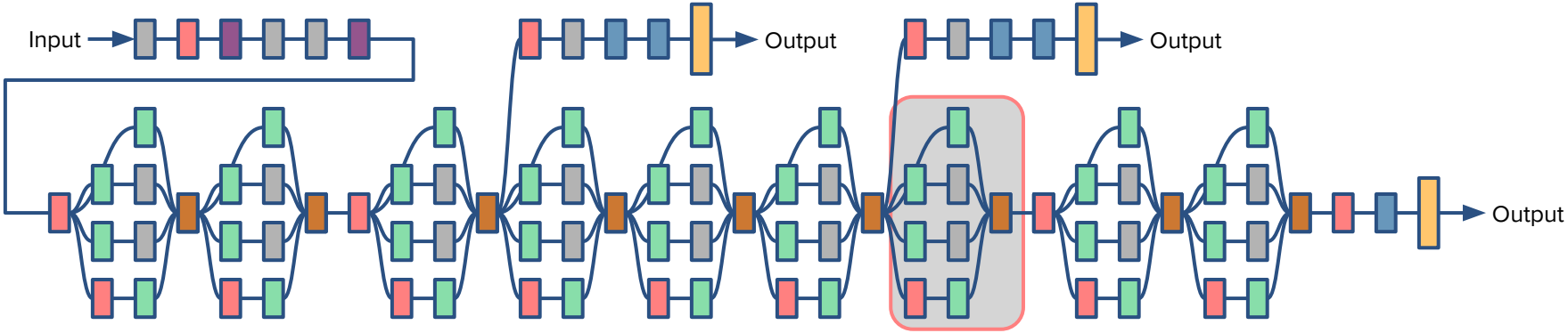

Snake heads


Primates


Dog snouts

What happens inside the Inception Network?

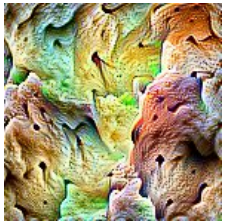
- We can now go block by block:

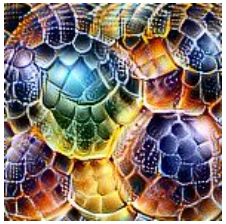


Block 4e: At this level, many neurons start to react to multiple visually similar concepts (like satellite dishes/sombreros, ice cream /bread).


Sombreros

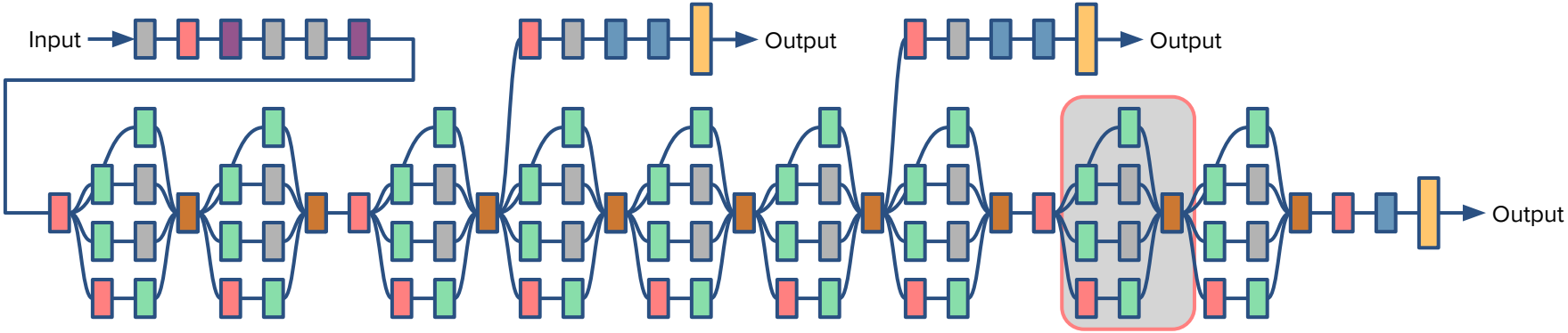

Cat fur


Ice Cream/bread


Turtle shells

What happens inside the Inception Network?

- We can now go block by block:



Block 5a: after another pooling step, visualizations become harder to interpret, since visually similar concepts are getting mixed.


Candles

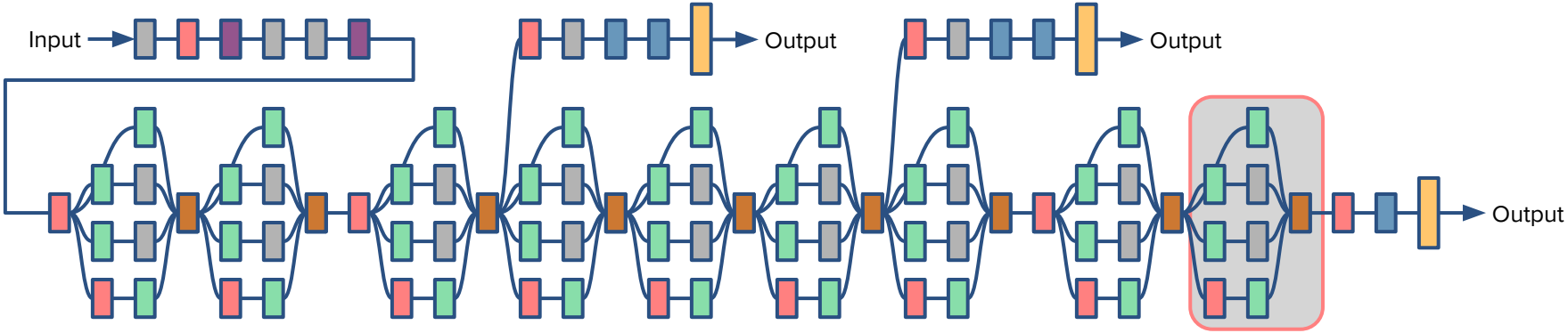

Traffic lights


Brass Instruments

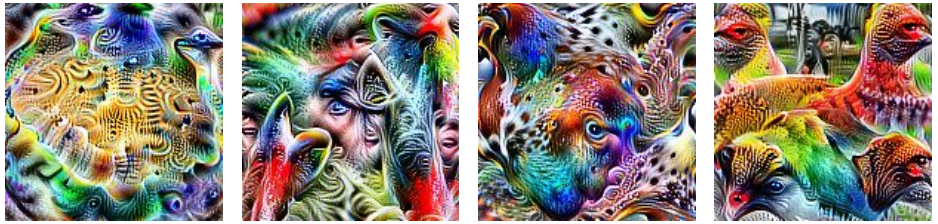

Balls

What happens inside the Inception Network?

- We can now go block by block:

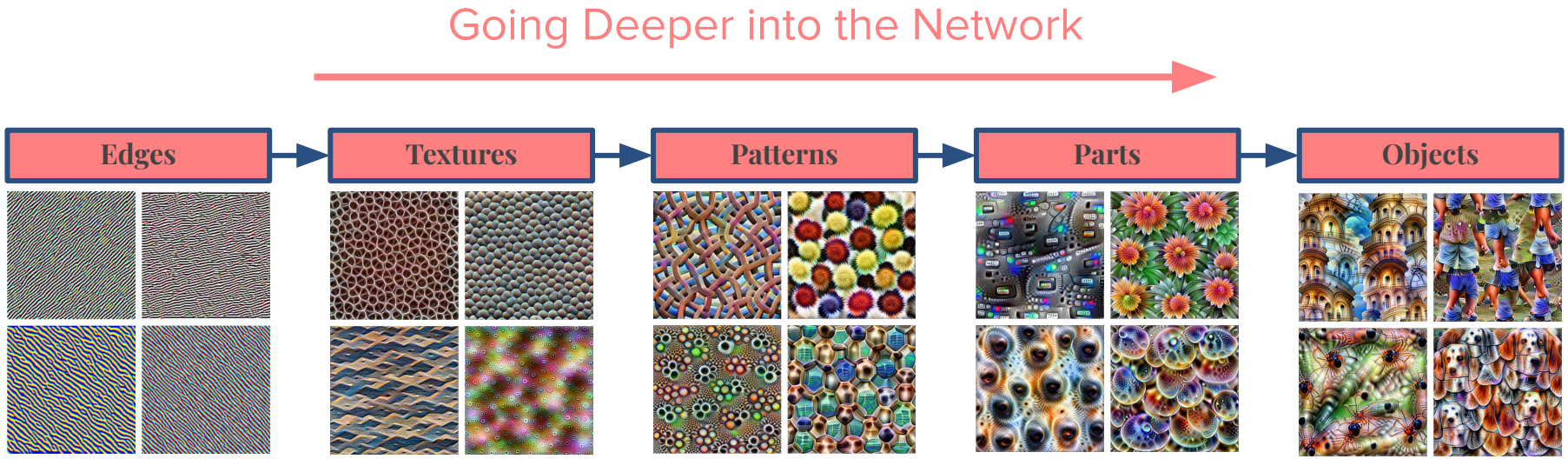


Block 5b: In this layer visualizations become mostly nonsensical collages. Neurons do not seem to correspond to particularly meaningful semantic ideas anymore.



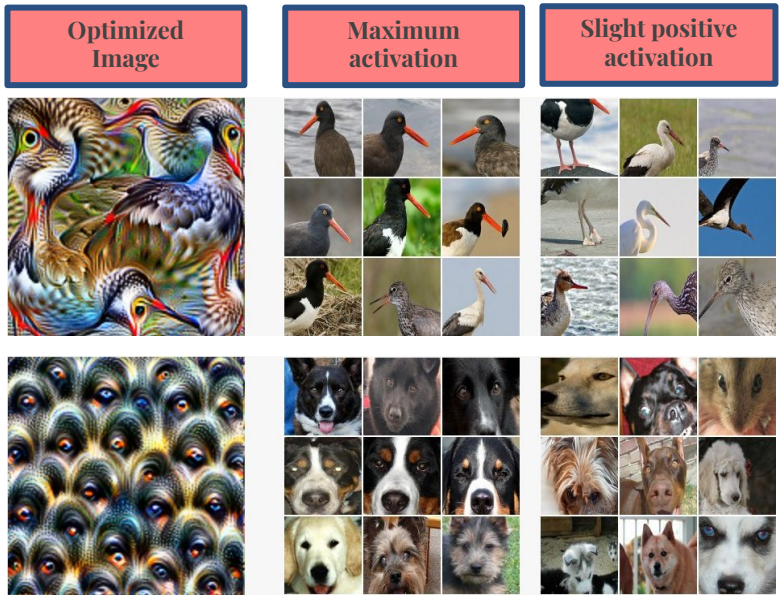
Summary

- In sum, as you go deeper into the network when training it, the layers learn **more and meaningful/less** and **less abstract image features.**
- This process is common in all deep convolutional nets, not just in the Inception.



Images that Activate the Units the Most

- We can also check which images in the dataset activate certain units (4b stage) the most:

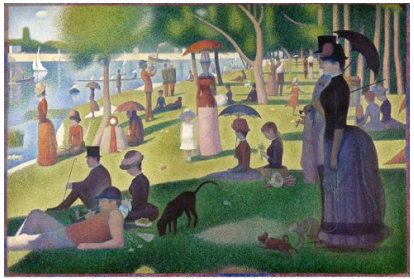


- Note how the units **are specialized in certain image features/kinds**.

DeepDream

- Instead of finding an image that maximally activates a unit of GoogLeNet, we can optimize for a **whole layer** (which represents a certain abstraction level).
- To understand this effect, we can also replace the **initial random image by a real image** and let the optimization start there. This process is called **DeepDream**.
- If we choose to optimize a lower layer (that is sensitive to basic features such as edges) on a real image, it'll insert strokes or ornament-like patterns in it, for example.

Original Image



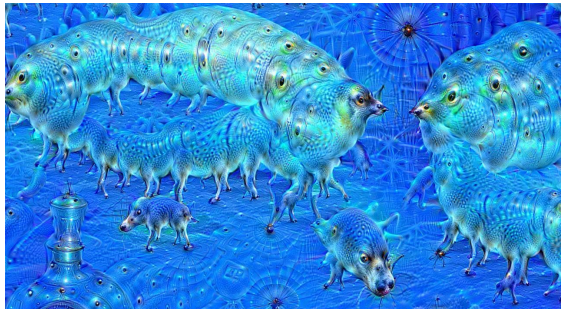
Different "Dreams" by optimizing different early channel in GoogLeNet

DeepDream

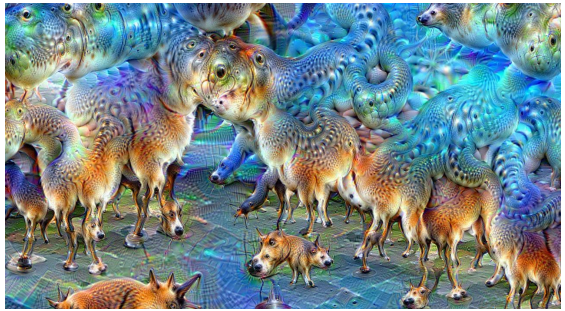
- If we choose higher-level layers to optimize (which identify more sophisticated features in images) **complex features** or even whole objects tend to emerge.
- To make these features more explicit, we can then create a feedback loop: we optimize a layer for an image and give the resulting image and back to the optimization.
- Below, this process is repeated in a network trained on dog images. Note that it “dogfies” everything it finds to the point of getting a lot of nonsense!

Original Image

After 10 DeepDream Iterations

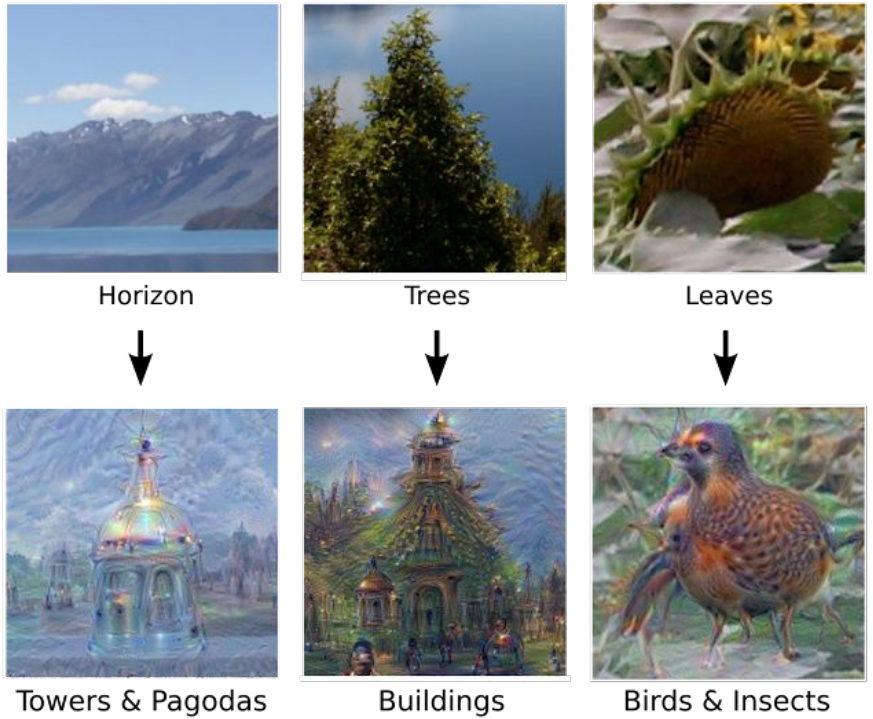


After 50 DeepDream Iterations



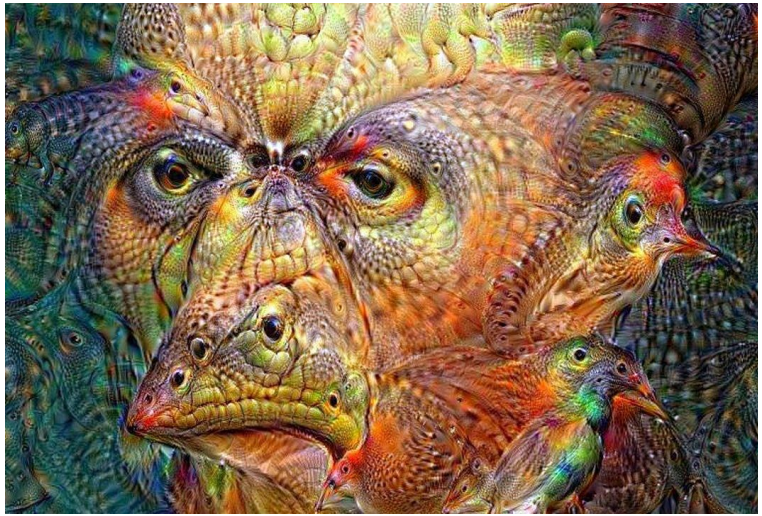
DeepDream

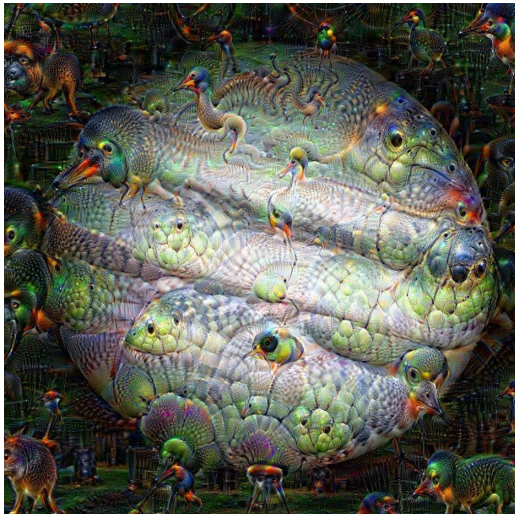
- Repeating this process on the later layers of GoogLeNet trained on ImageNet, DeepDream will approximate features from the original image with features it learned well. For example,
  - Horizon lines tend to get filled with towers and pagodas.
  - Rocks and trees turn into buildings.
  - Birds appear in images of leaves.
- In this sense, DeepDream gives us a qualitative sense of the level of abstraction that a particular layer has achieved in its understanding of images.



DeepDream

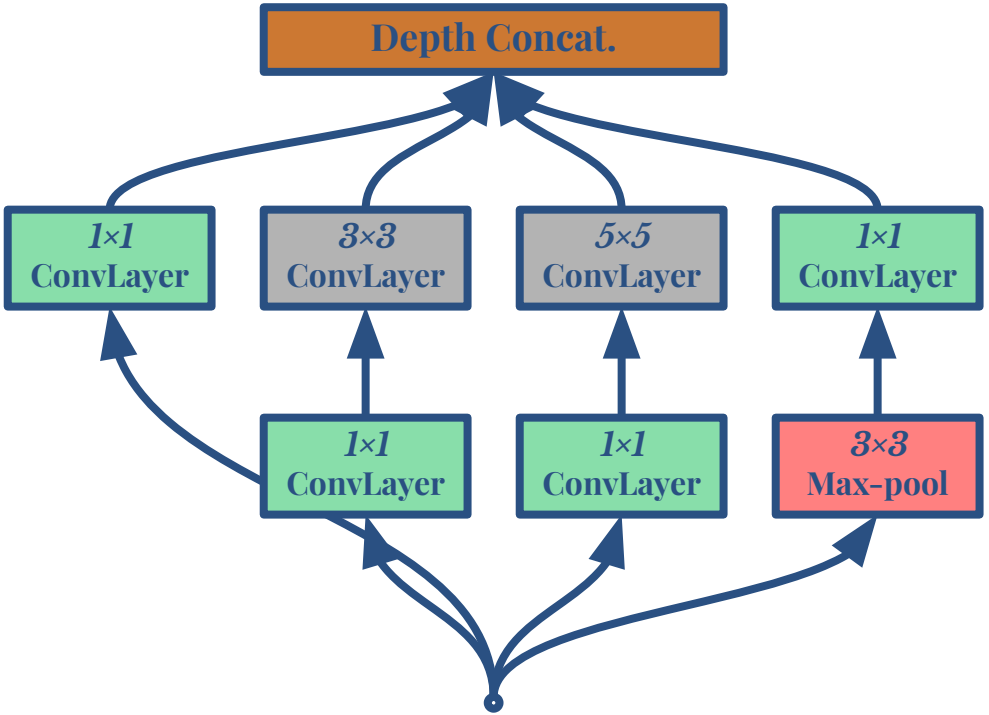
- Besides learning about what deep CNNs learn, we can use DeepDream to produce “exotic” image filters (and obviously have tons of fun)!





Exercise (*In pairs*)

- Implement the improved Inception Module from GoogLeNet (on the right) in PyTorch. Feel free to use the code in here as a base.



Exercise (*In pairs*)

- Implement the improved Inception Module from GoogLeNet (on the right) in PyTorch. Feel free to use the code in here as a base.