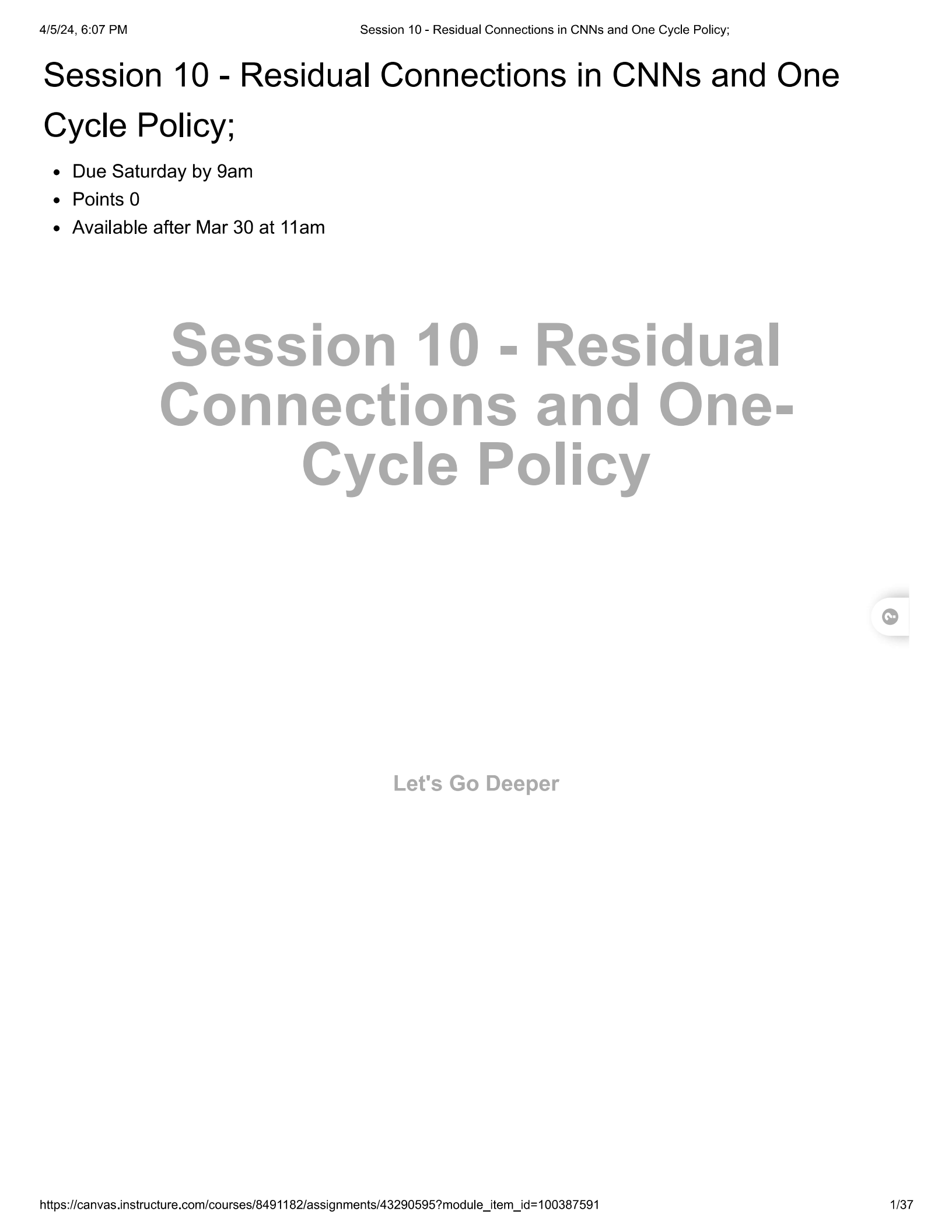
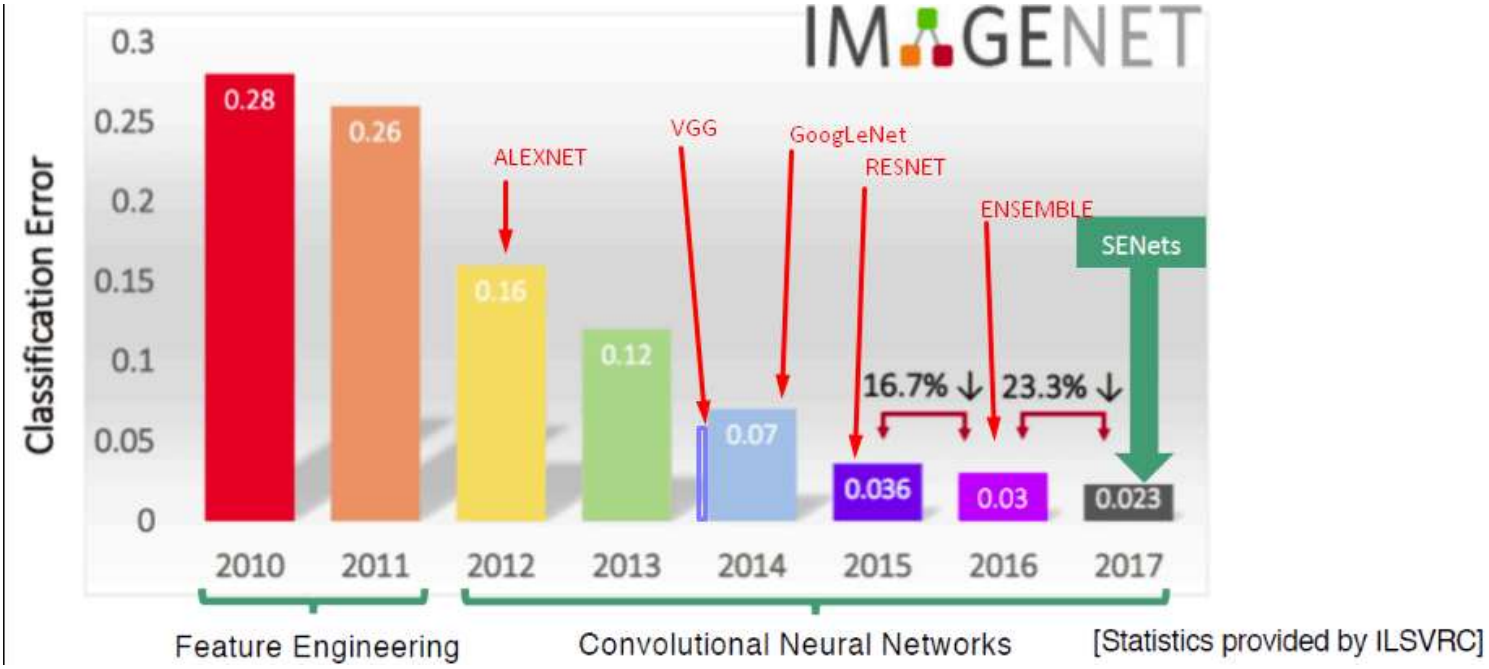
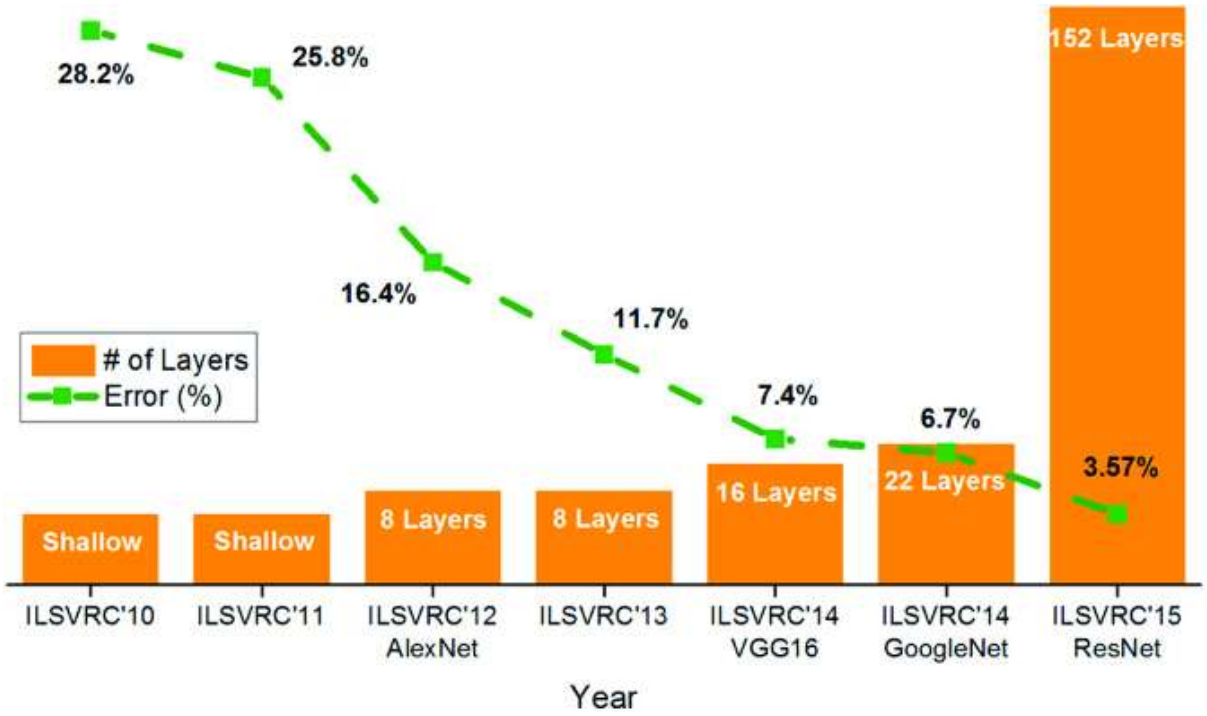
Session 10 - Residual Connections in CNNs and One Cycle Policy;

- Due Saturday by 9am
- Points 0
- Available after Mar 30 at 11am





There is a general trend of going deeper as years pass by!



But when we add more layers, though we still keep 3 transformation blocks, still, we add a lot of receptive fields!

Way beyond the size of the image or the object sizes in it!

What was the receptive field again?

The receptive field in a Deep Neural Network (DNN) refers to the portion of the input space that a particular neuron is "sensitive" to, or in other words, the region of the input that influences the neuron's output. In Convolutional Neural Networks (CNNs), the receptive field is determined by the spatial extent of the filters and the strides of the convolutional operations. Larger receptive fields lead to neurons that have a wider field of view, allowing them to capture more complex relationships in the input data.

RF is more than the size of the image!

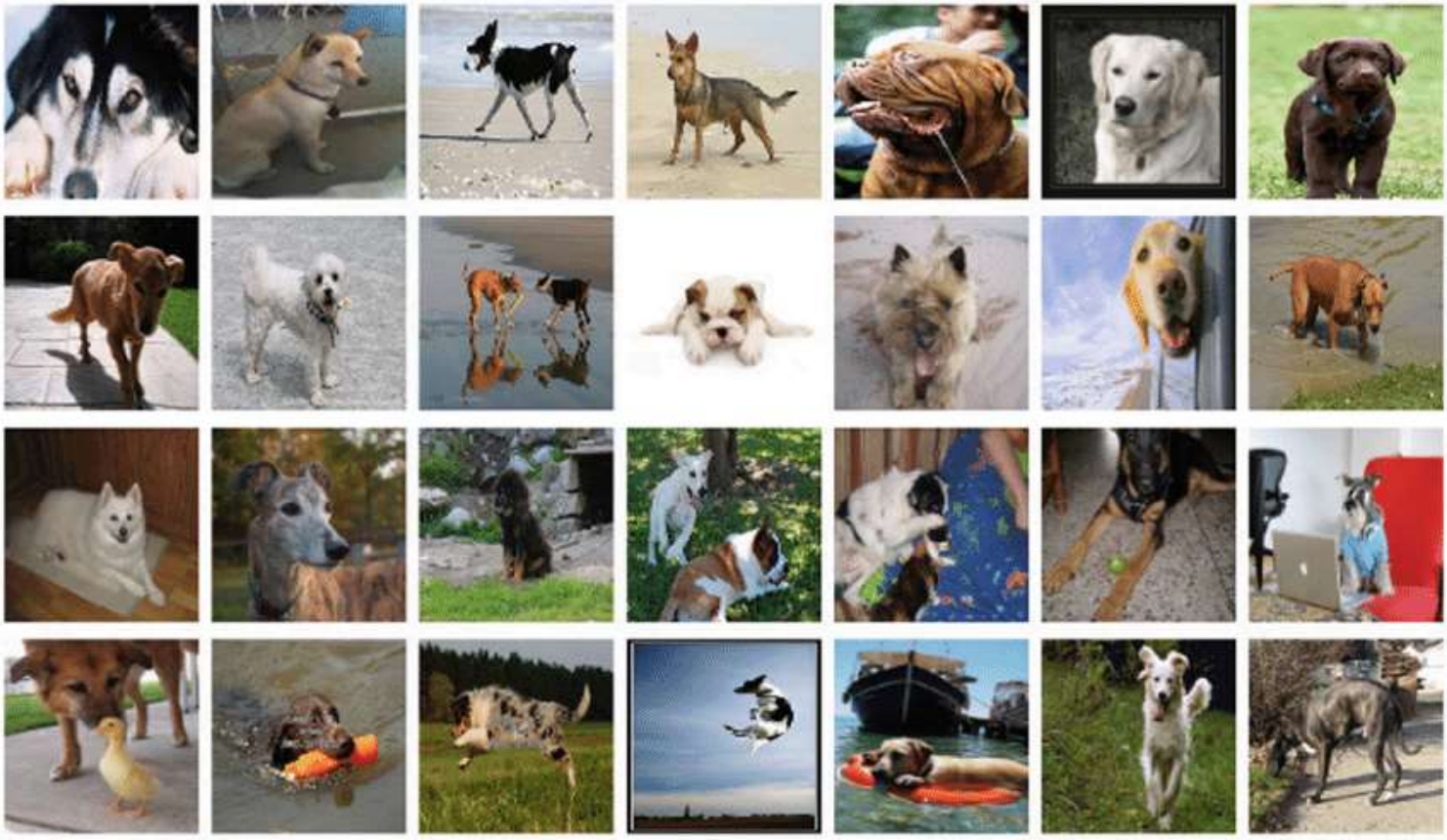
Let's look at some questions

What receptive field value do we actually need?

This is linked to the size of the objects:

We want to be able to identify the smallest and the largest dog. A very high RF will b able to capture the large dog, but completely miss out on the smaller one, and vice versa!

Sometimes the images might not be able to cover the whole object as well!

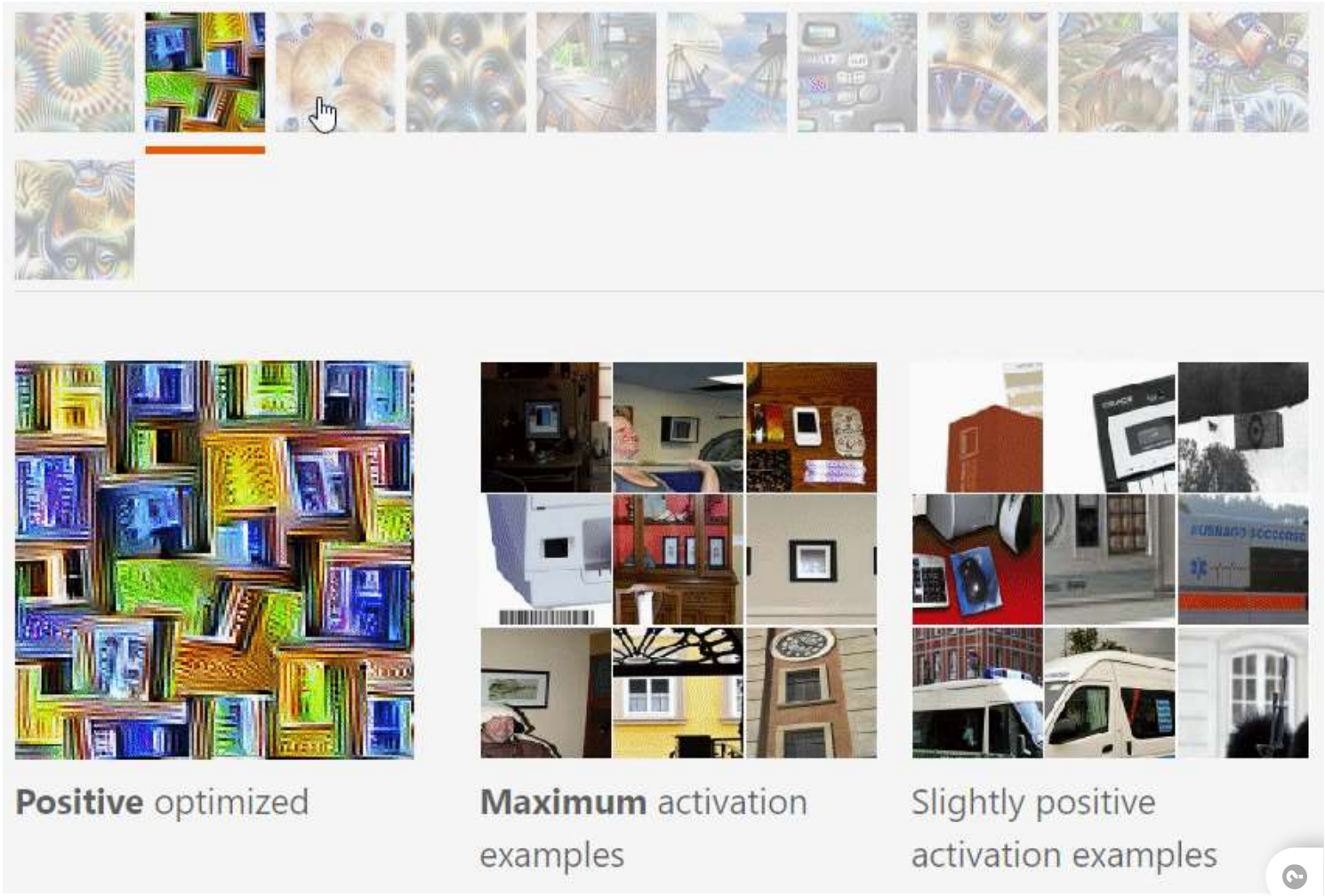


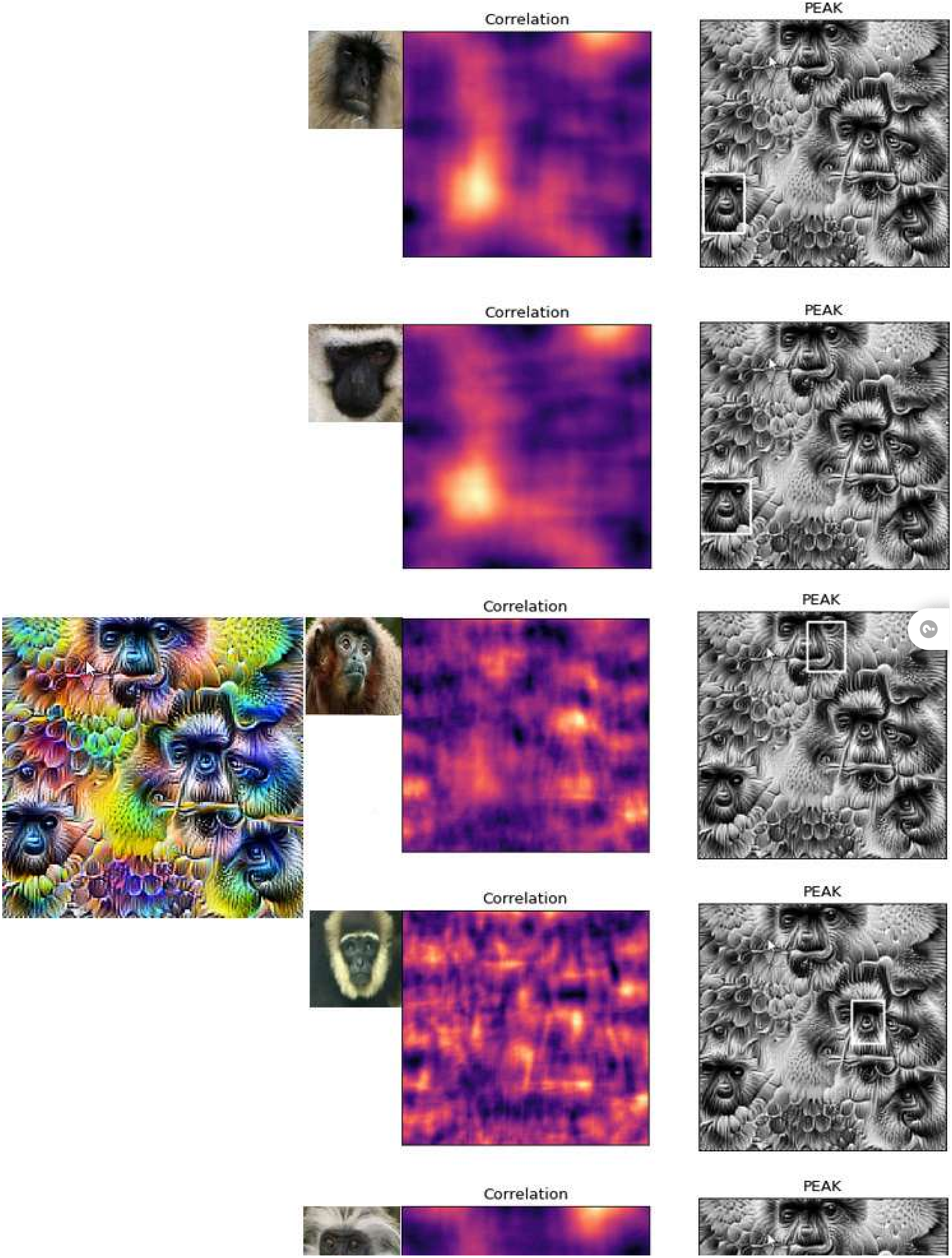
But the missing parts of these dogs might be available/guessed from other images.

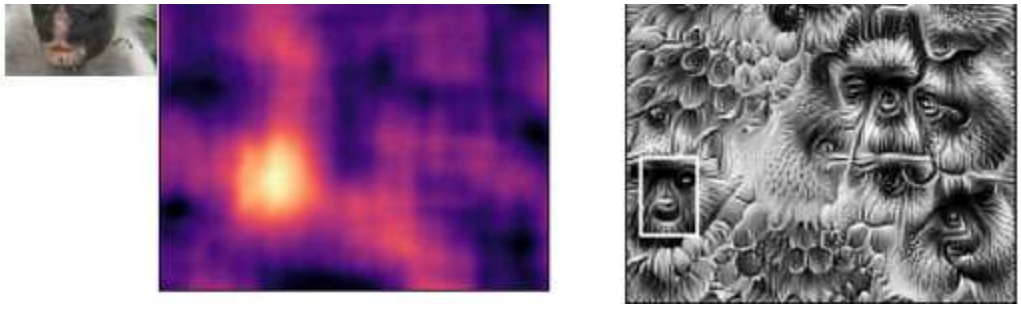
Most times, there are just too many variants (shape/size/structure) of the same objects

So we need to handle different sizes and variants

HENCE A LARGE RECEPTIVE FIELD THAT CAN HOLD **DIFFERENT VARIANTS** **(https://distill.pub/2017/feature-visualization/)** AS A TEMPLATE


Positive optimized
Maximum activation examples
Slightly positive activation examples


Correlation
PEAK
Correlation
PEAK
Correlation
PEAK
Correlation
PEAK
Correlation
PEAK



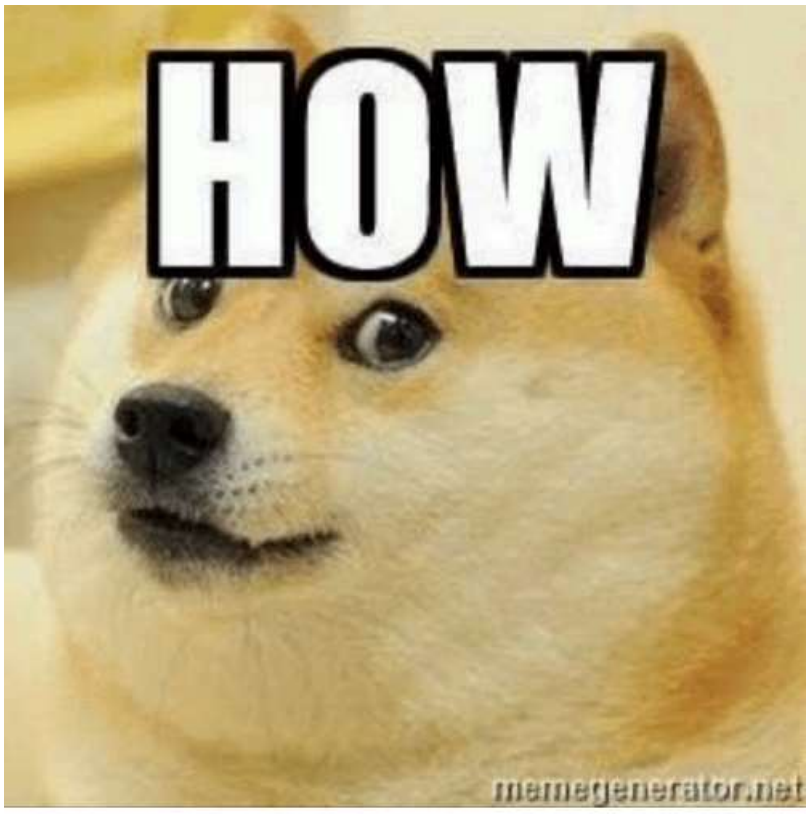
VGG Architecture

VGG was an audacious attempt to go deeper. BN wasn't invented, so they couldn't go deeper, but deep enough to be 2nd in the ISLRVC competition.

But, VGG has 1 receptive field, and not many

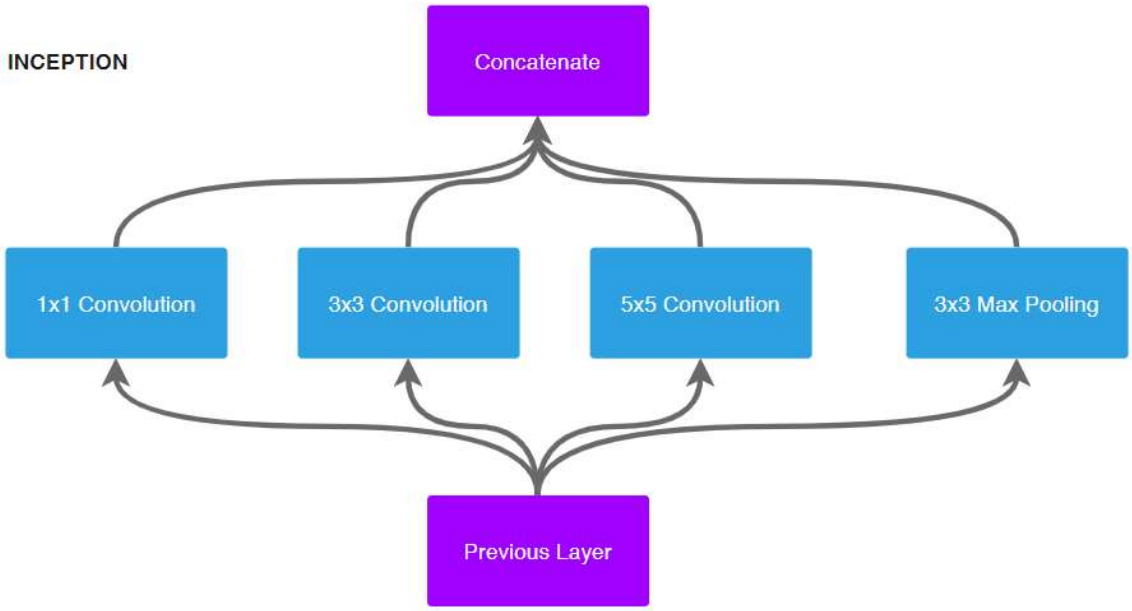
Many Receptive Fields

But

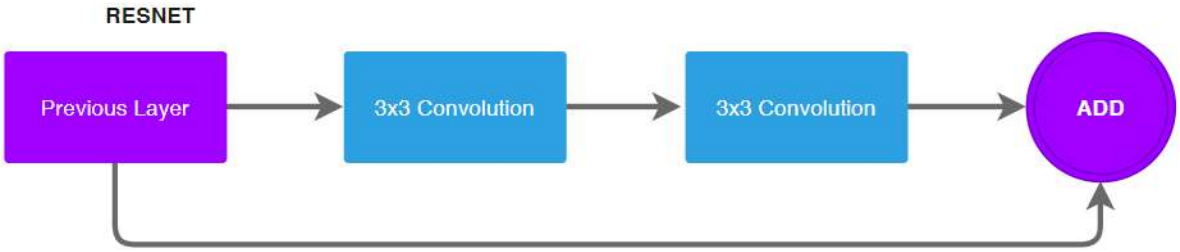


This is where the "group" concept we covered in the last session comes into play.

Let's look at **Inception Block**



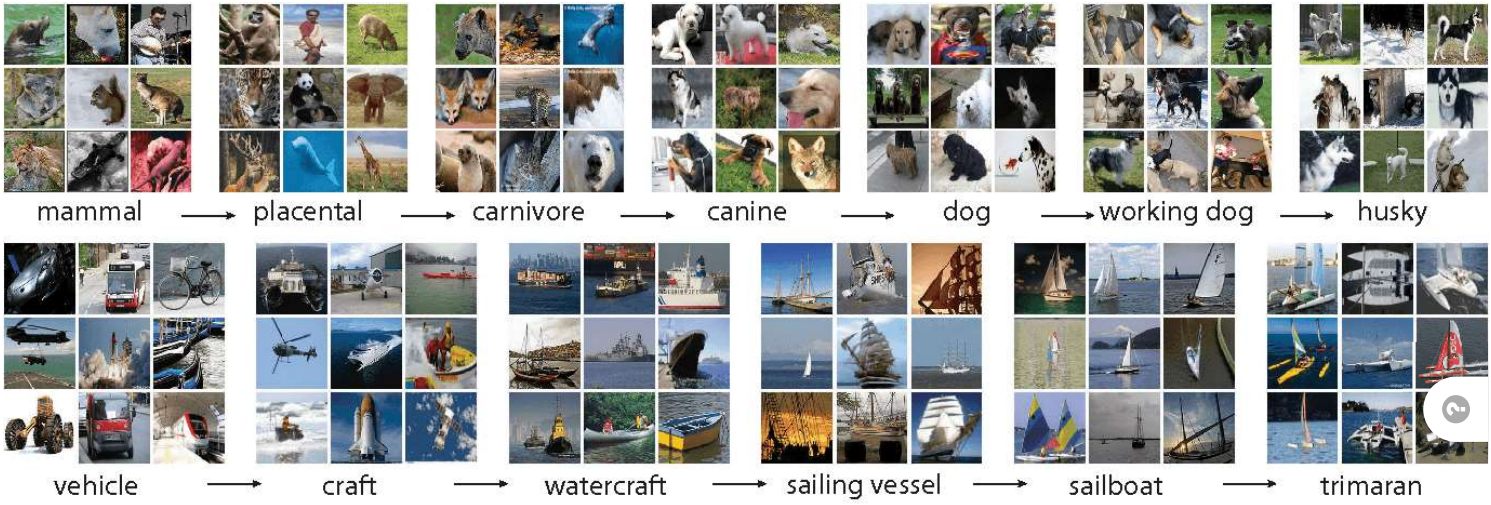
Isn't it wonderful how this simple architecture allows us to target many receptive fields?



Beyond Image Size Receptive Fields

But what happens when we increase our Receptive field, how do they "look" or what are they looking for?

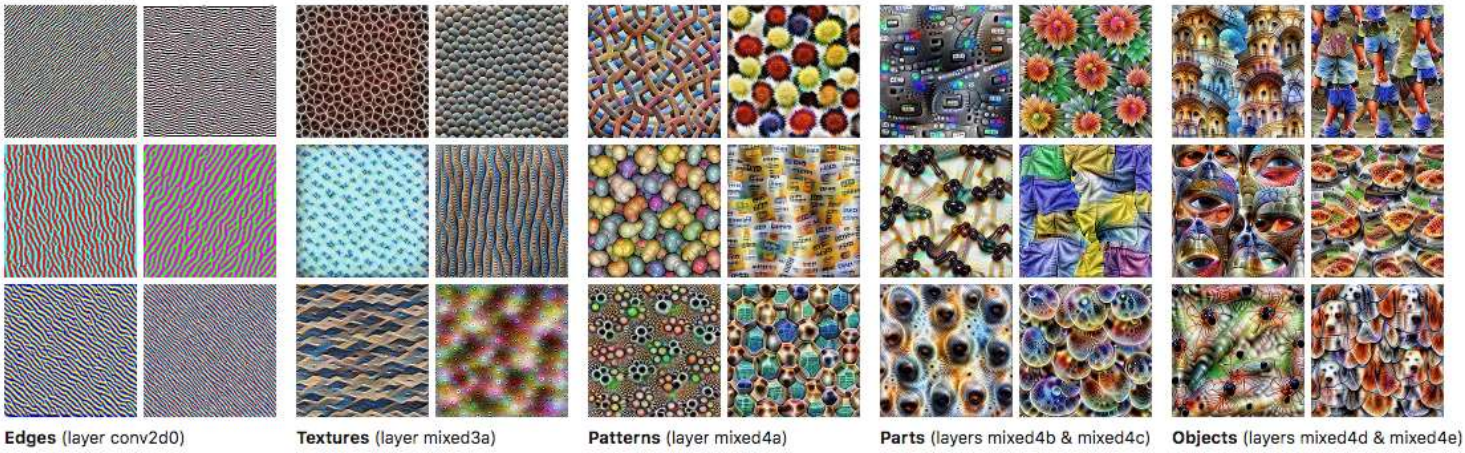
Well, this is what data looks like:



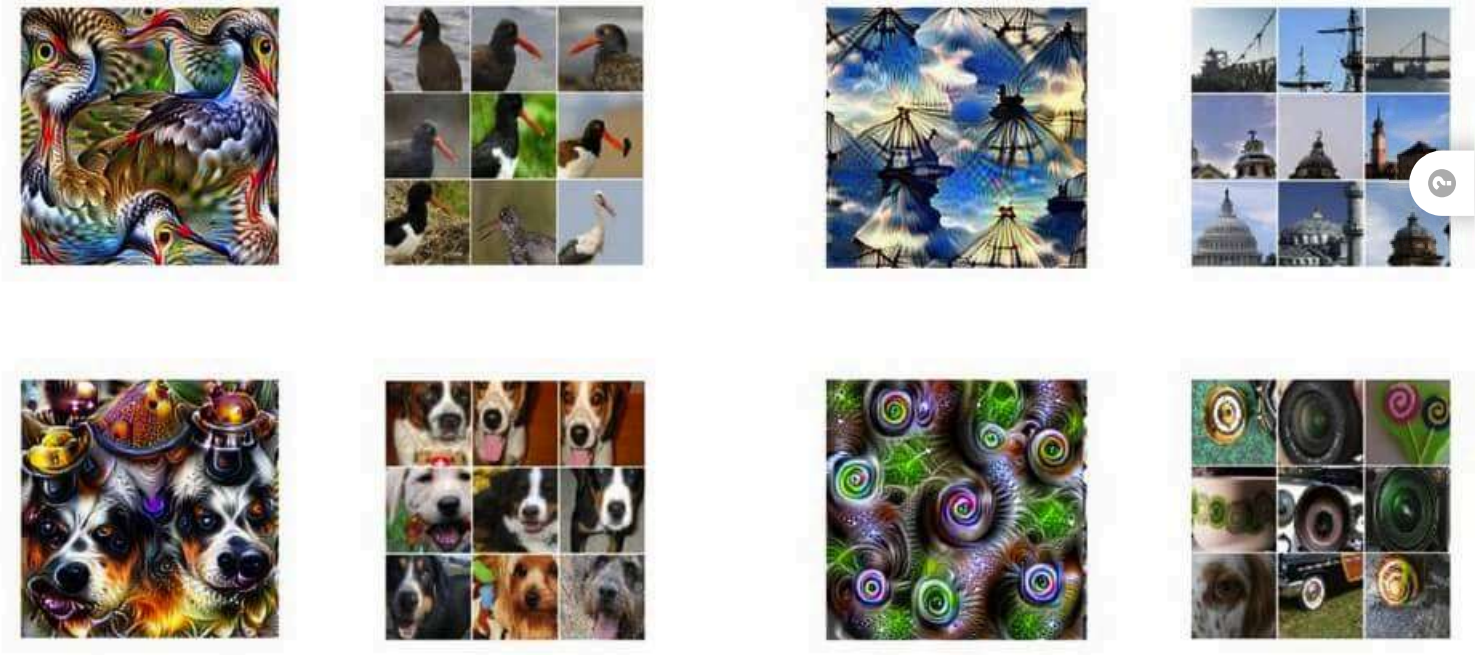
Varied in size, different but sometimes semantic backgrounds.

When we target higher receptive fields our target become

Edges >> Textures >> Patterns >> Parts >> Objects >> O



Object templates capture the same object in different sizes and with different backgrounds!

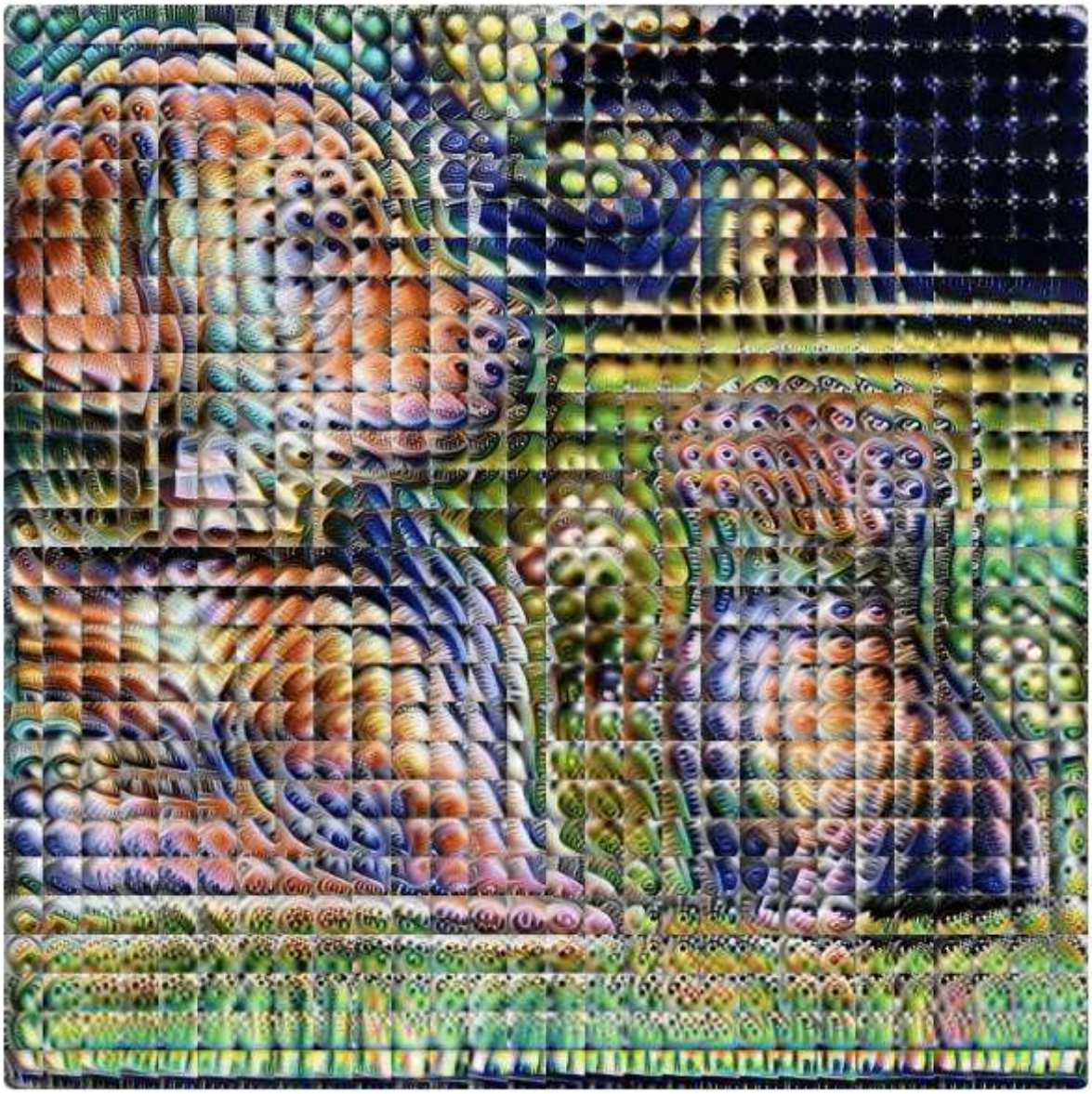


Change in "Current Views" As we do down the layers

But it is even more interesting to see what happens on the micro level at each layer!

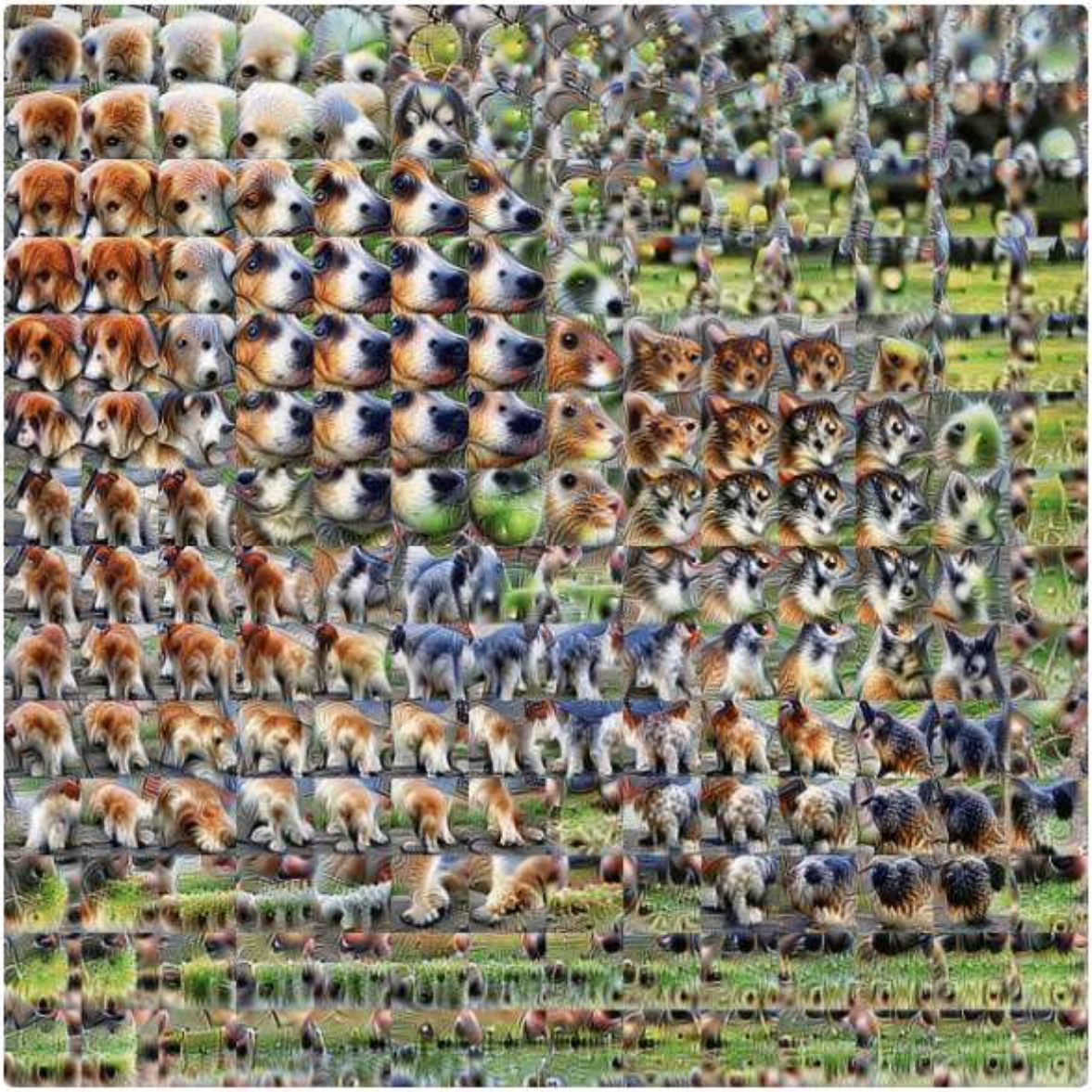
Let's start with this cute image:

After a few Conv layers

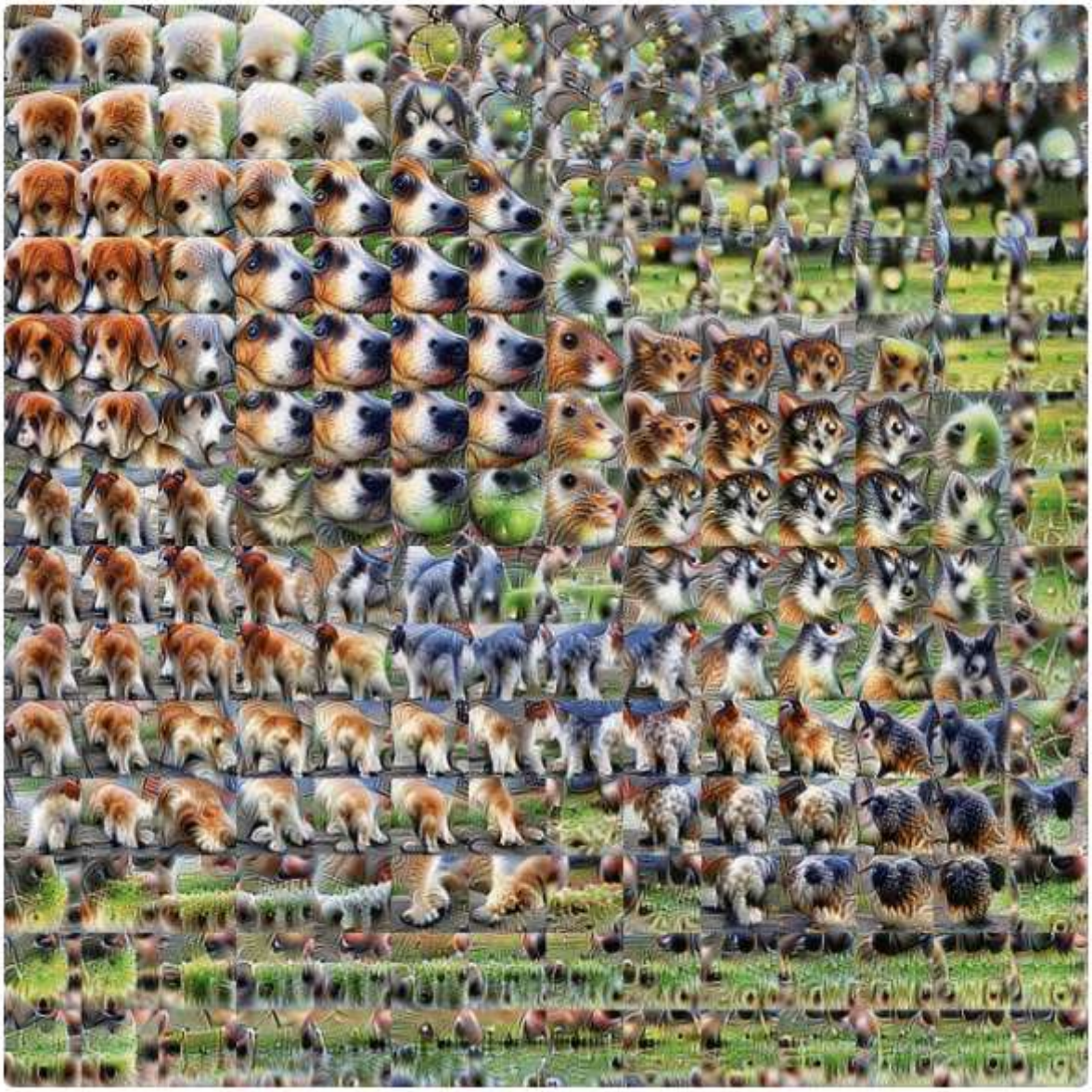


Before we proceed, what exactly are we looking at above?

After a few more Conv layers

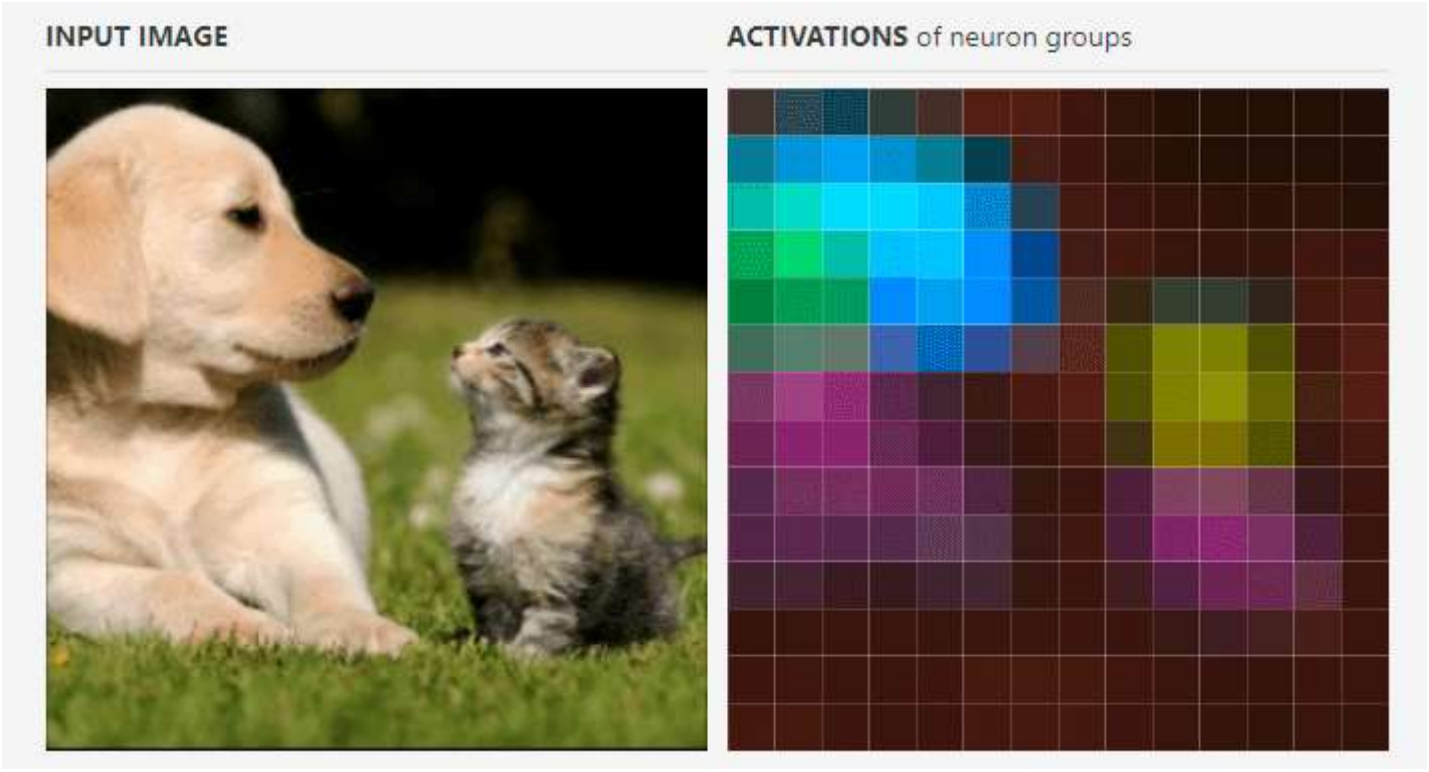


After even more Conv layers



After a few more Conv layers... but now we are getting closer to SoftMax

And closer to SoftMax



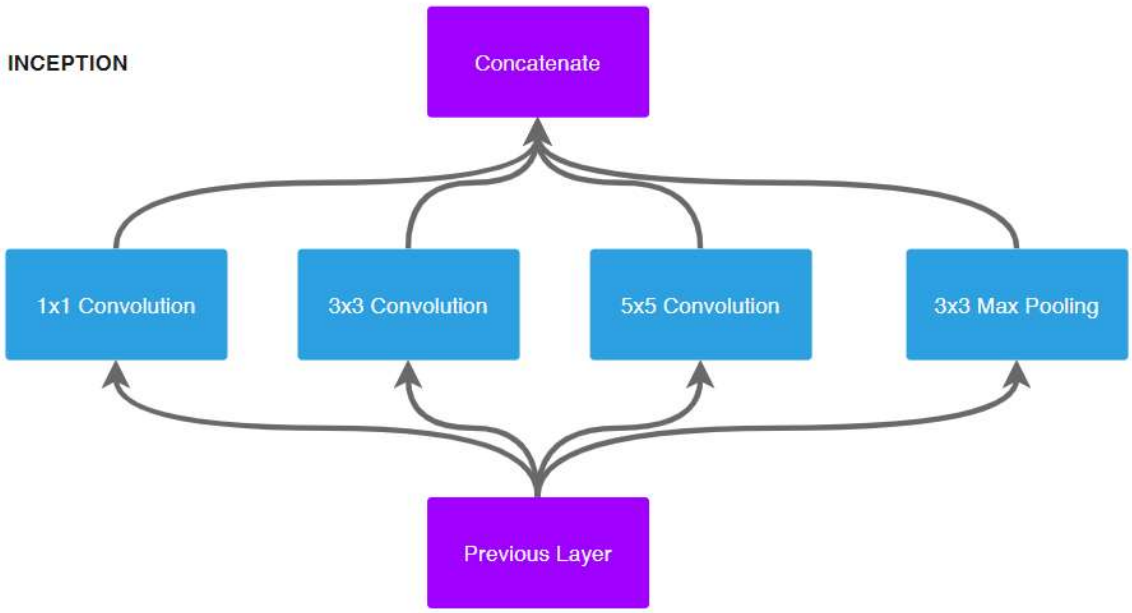
Inception

Inception V1

- **Salient parts** in the image can have an extremely **large variation** in size. For instance, an image with a dog can be either of the following, as shown below. The area occupied by the dog 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**, and a **smaller kernel** is preferred f information that is distributed more **locally.**
- Very deep networks are prone to overfitting. It is also hard to pass gradient updates through the entire network
- Naively stacking large convolution operations is **computationally expensive**.

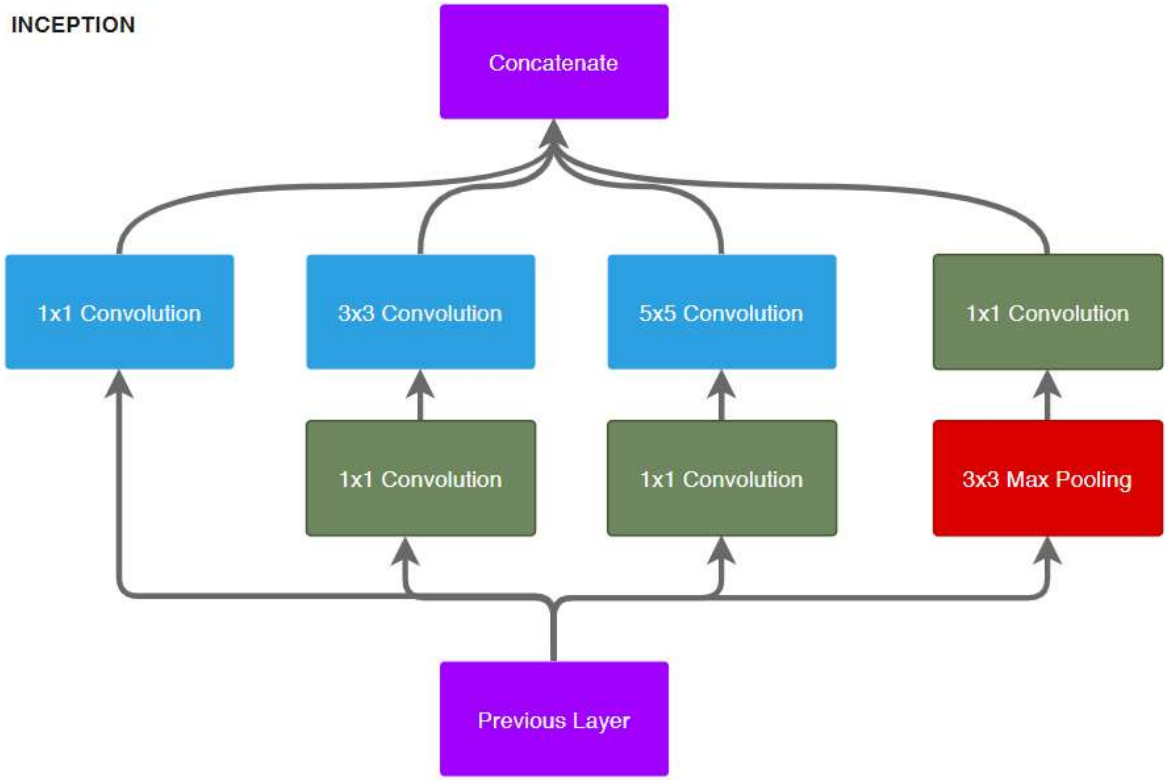
Inception V1 solves this through:

- with multiple-sized filters operating on the same level
- going wider to capture different GRF contexts
- not using addition, but concatenation to make sure GRF links are maintained till the SoftMax layer

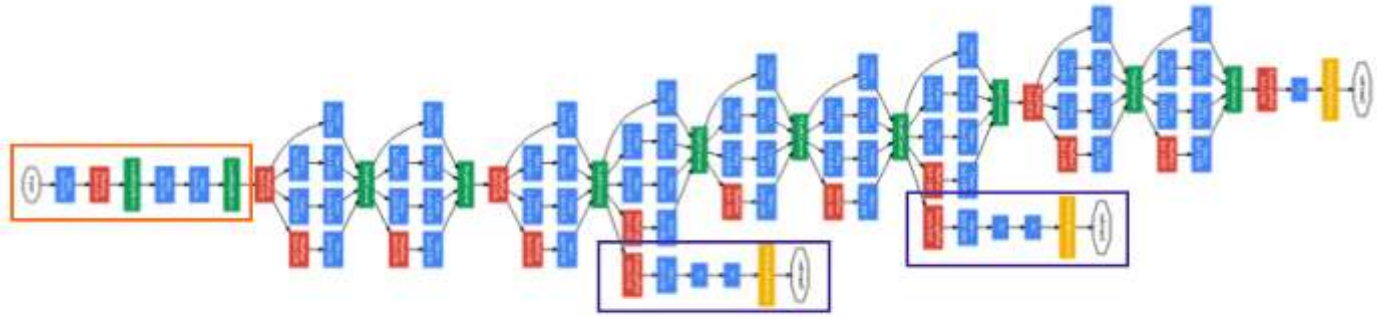


But this was expensive (think of the next 3x3s, 5x5s which would now need a large number of channels to properly convolve).

To make it cheaper, the authors **limit** the number of **input channels** by adding an **ex**
1x1 convolution before the 3x3 and 5x5 convolutions.



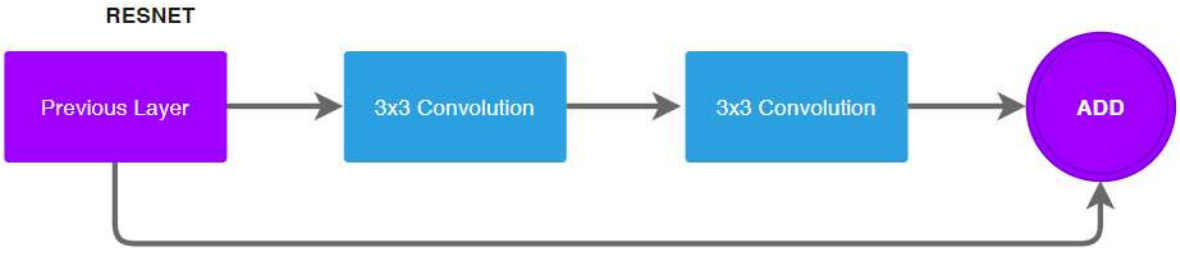
- GoogLeNet has 9 such inception modules stacked linearly.
- It is 22 layers deep (27, including the pooling layers).
- It uses global average pooling at the end of the last inception module.
- it is a pretty **deep classifier**. As with any very deep network, it is subject to the **vanishing gradient problem**.
- To prevent the **middle part** of the network from "**dying out**", the authors introduced **two auxiliary classifiers** (The purple boxes in the image). They essentially applied softmax to the outputs of two of the inception modules and computed an **auxiliary loss** over the same labels. The **total loss function** is a **weighted sum** of the **auxiliary loss** and the **real loss**. The weight value used in the paper was 0.3 for each auxiliary loss.
- auxiliary loss is purely used for training purposes and is ignored during inference.



The total loss used by the inception net during training.

total_loss = real_loss + 0.3 * aux_loss_1 + 0.3 * aux_loss_2

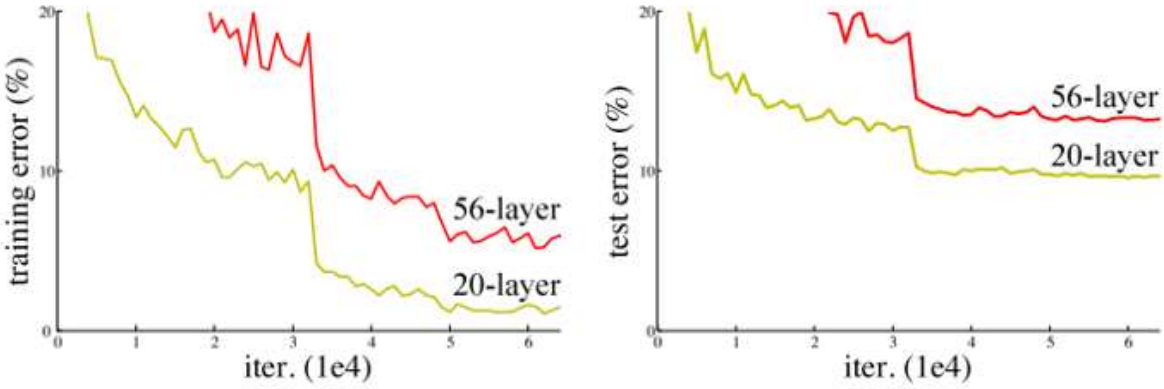
ResNet V1 & V2



Since AlexNet, the state-of-the-art CNN architecture is going deeper and deeper. While AlexNet had only 5 convolutional layers, the VGG network and GoogleNet (also codenamed Inception_v1) had 19 and 22 layers respectively.

However, increasing network depth does not work by simply stacking layers together. Deep networks are hard to train because of the notorious vanishing gradient problem — as the gradient is back-propagated to earlier layers, repeated multiplication may make the gradient infinitely small. As a result, as the network goes deeper, its performance gets saturated or even starts degrading rapidly.

BEFORE RESNET



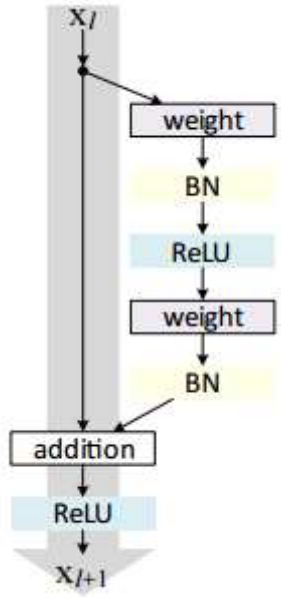
AFTER RESNET

Table 3. Classification error (%) on the CIFAR-10/100 test set using the original Residual Units and our pre-activation Residual Units.

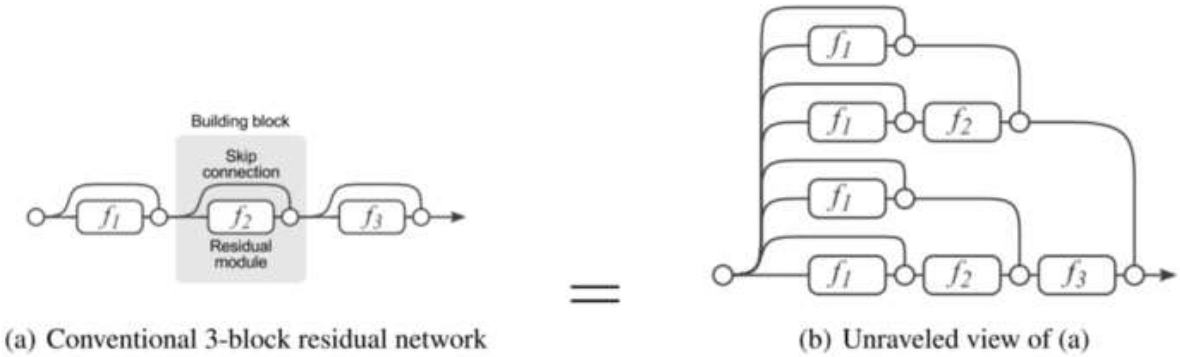
dataset	network	baseline unit	pre-activation unit
CIFAR-10	ResNet-110 (1layer skip)	9.90	8.91
	ResNet-110	6.61	6.37
	ResNet-164	5.93	5.46
	ResNet-1001	7.61	4.92
CIFAR-100	ResNet-164	25.16	24.33
	ResNet-1001	27.82	22.71

The authors of the ResNet paper argue, that the stacking layers shouldn't degrade the network performance, because we could simply stack identity mappings (layers that don't do anything) upon the current network, and the resulting architecture would perform the same.

This indicates that the deeper model should not produce a training error higher than its shallower counterparts.



If you think about it, ResNet can be considered as an ensemble of Smaller networks!



(a) Conventional 3-block residual network (b) Unraveled view of (a)

WHERE IS THE RESIDUE?

But why call it residual? Where is the residue? It's time we let the mathematicians within us to come to the surface. Let us consider a neural network block, whose input is *x* and we would like to learn the true distribution *H(x)*. Let us denote the difference (or the residual) between this as

$$F(x) = Output - Input = H(x) - x$$

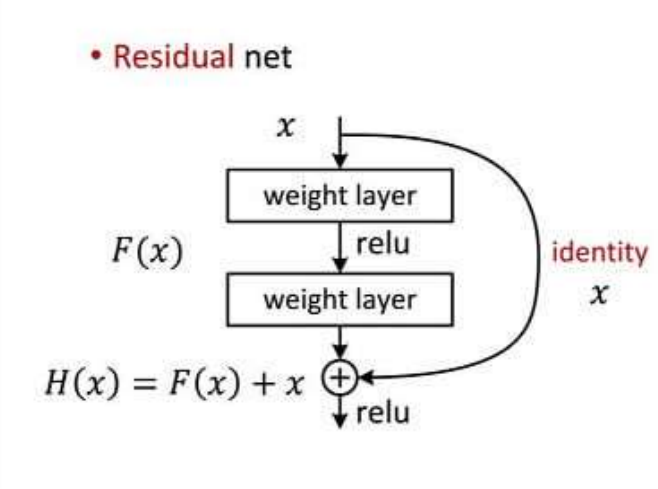
Rearranging it, we get,

$$H(x) = F(x) + x$$

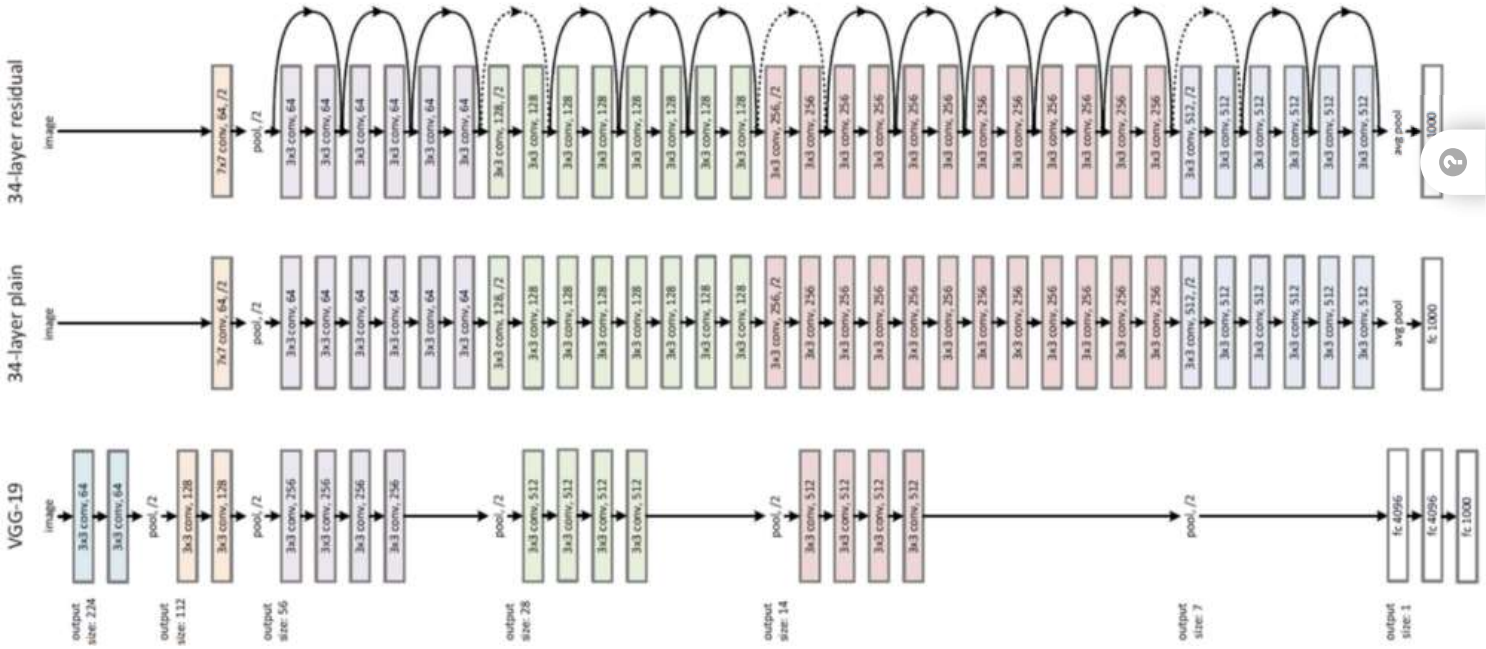
THE CORE IDEA

The core idea of ResNet is introducing a so-called "identity shortcut connection" that skips one or more layers.

A residual block is displayed as the following:



The residual unit obtains ***F(x)*** by processing ***x*** with two weight layers. Then it adds ***x*** to ***F(x)*** to obtain ***H(x)***.



For ResNet, there are **two kinds of residual connections:**

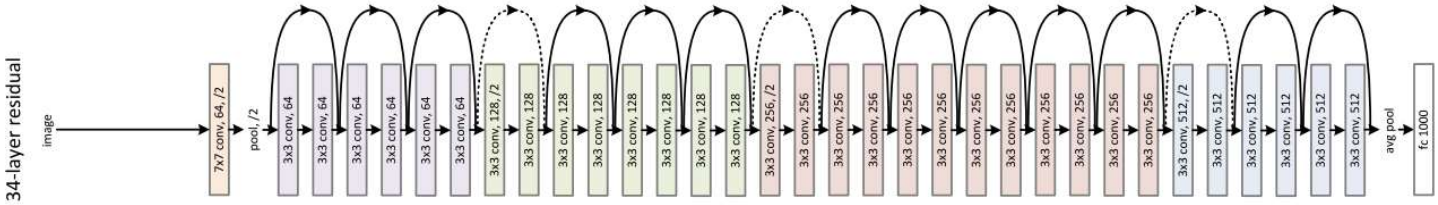
1. the identity shortcuts (x) can be directly added with the input and output are of the same dimensions
2. when the dimensions change (input is larger than residual output, but we need to add them). The default way of solving this is to use a 1x1 Conv with a stride of 2. **Yes, half of the pixels will be ignored.**

François Chollet, Author of Keras .

He says: They do this because it's what you should be doing. Residual connections with different shapes should be handled via a learned linear transformation between the two tensors, e.g. a 1x1 convolution with appropriate strides and border_mode, or for Dense layers, just matrix multiplication.

RESNET MODEL WALKTHROUGH

ResNet34



R18	- Regular	- 2, 2, 2, 2
R34	- Regular	- 3, 4, 6, 3

We'll go through ResNet34.

- It has 1 convolution layer of 7x7 sized kernel (64), with a stride of 2
- It is followed by MaxPooling. In fact, ResNet has only 1 MaxPooling operation!
- It is followed by 4 ResNet blocks (config: 3, 4, 6, 3)
- The channels are constant in each block (64, 128, 256, 512 respectively). Each block has only 3x3 kernels.
- The channel size is constant in each block
- Except for the first block, each block starts with a 3x3 kernel of stride 2 (this handles MaxPooling)
- The dotted lines are 1x1 convs with stride 2

SideNote:

The accuracy of convolutional networks evaluated on ImageNet is **vastly underestimated**. We find that when the mistakes of the model as assessed by human subjects and considered correct when four out of five humans agree with the model's prediction, the **top-1 error of a ResNet-101** trained on ImageNet... decreases from **22.69% to 9.47%**. Similarly, the top-5 error decreases from **6.44% to 1.94%**. (This is true for other models as well). **Ref (https://eccv2018.org/openaccess/content_ECCV_2018/papers/Pierre_Stock_ConvNets_and_**

ResNet Bottleneck Blocks

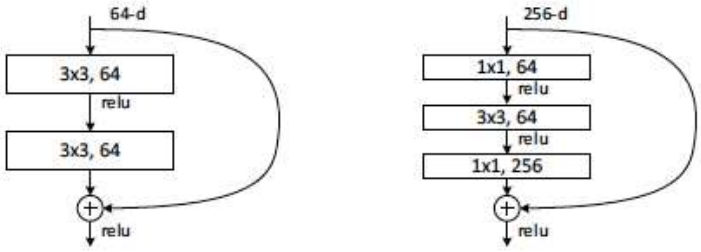
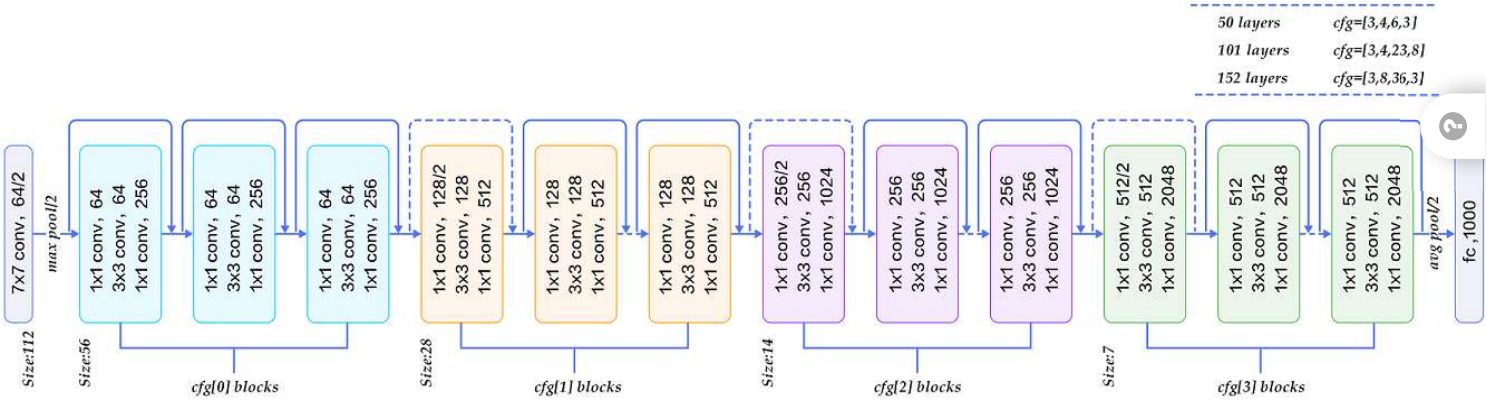


Figure 5. A deeper residual function $\mathcal{F}$ for ImageNet. Left: a building block (on 56×56 feature maps) as in Fig. 3 for ResNet-34. Right: a "bottleneck" building block for ResNet-50/101/152.

deeper non-bottleneck ResNets also gain accuracy from increasing depth but are not as economical as the bottleneck ResNets. So the usage of bottleneck designs is mainly due to practical considerations.



R18 - Regular - 2, 2, 2, 2

R34 - Regular - 3, 4, 6, 3

R50 - Bottleneck - 3, 4, 6, 3

R101 - Bottleneck - 3, 4, 23, 3

R152 - Bottleneck - 3, 8, 36, 3

ResNet V3

Aggregated Transformations

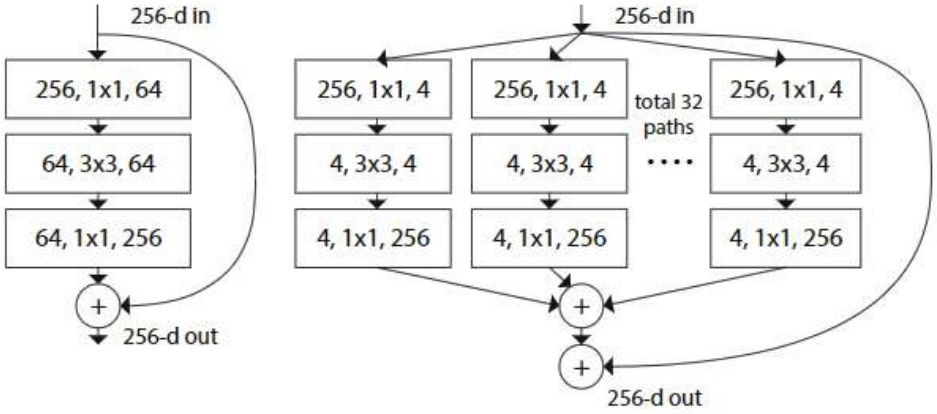


Figure 1. **Left**: A block of ResNet [14]. **Right**: A block of ResNeXt with cardinality = 32, with roughly the same complexity. A layer is shown as (# in channels, filter size, # out channels).

We'll limit ourselves to ResNet V1 or V2.

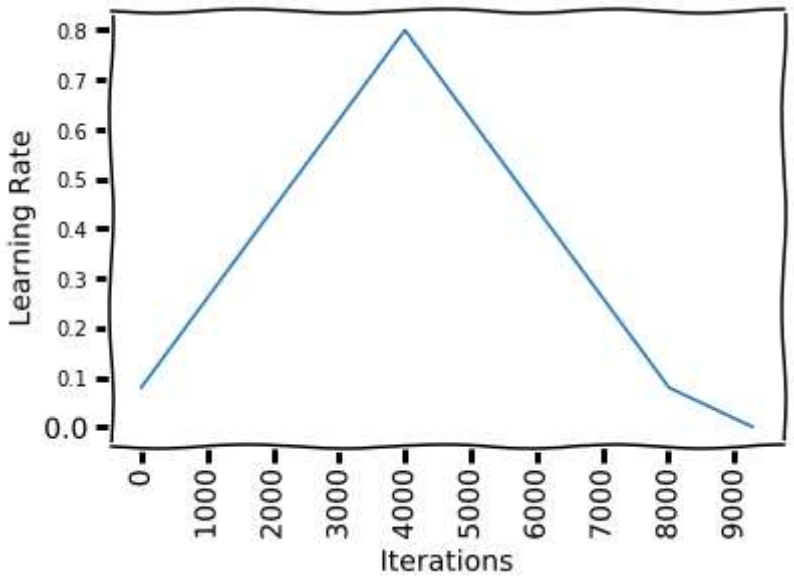
One Cycle Policy

In the paper " **A disciplined approach to neural network hyper-parameters: Part 1 — learning rate,**

batch size, momentum, and weight decay (https://arxiv.org/abs/1803.09820)", Leslie Smith describes the approach to setting hyper-parameters(namely learning rate, momentum, and weight decay) and batch size. In particular, he suggests a 1 Cycle policy to apply learning rates.

The author recommends doing one cycle of learning rate of 2 steps of equal length. We choose the maximum learning rate using a range test. We use a lower learning rate as 1/5th or 1/10th of the maximum learning rate. We go from a lower learning rate to a higher learning rate in Step 1 and back to a lower learning rate in Step 2. We pick this cycle length slightly less than the total number of epochs to be trained. In the last remaining iterations, we annihilate the learning rate way below the lower learning rate value(1/10 th or 1/100 th).

The motivation behind this is that, during the middle of learning when the learning rate is higher, the learning rate works as a regularization method and keeps the network from overfitting. This helps the network to avoid steep areas of loss and land better flatter minima.

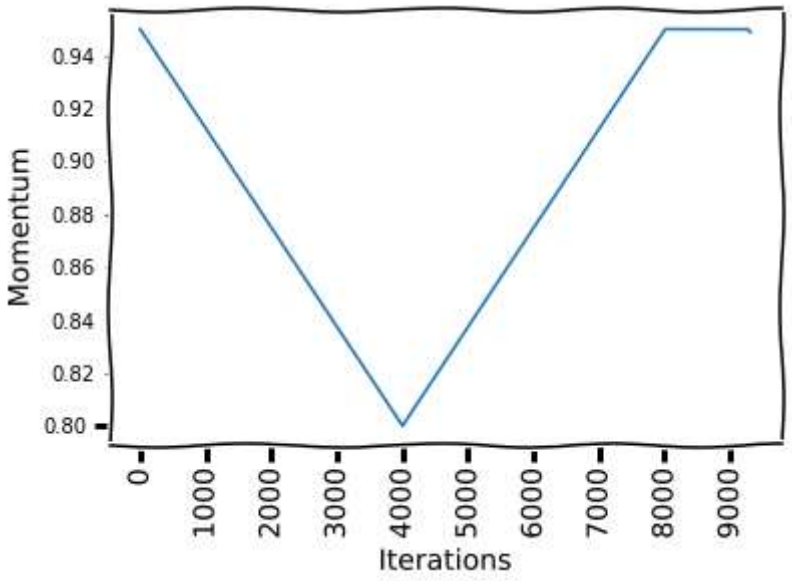


As in the figure, We start at a learning rate of 0.08 and make a step of 41 epochs to reach a learning rate of 0.8, then make another step of 41 epochs where we go back to a learning rate of 0.08. Then we make another 13 epochs to reach 1/10th of the lower

learning rate bound(0.08). With CLR 0.08–0.8, batch size 512, momentum 0.9, and Resnet-56, we got ~91.30% accuracy in 95 epochs on CIFAR-10.

Momentum and learning rate are closely related. It can be seen in the weight update equation for SGD that the momentum has a similar impact as the learning rate on weight updates.

The author found in their experiments that reducing the momentum when the learning rate is increasing gives better results. This supports the intuition that in that part of the training, we want the SGD to quickly go in new directions to find better minima, so the new gradients need to be given more weight.



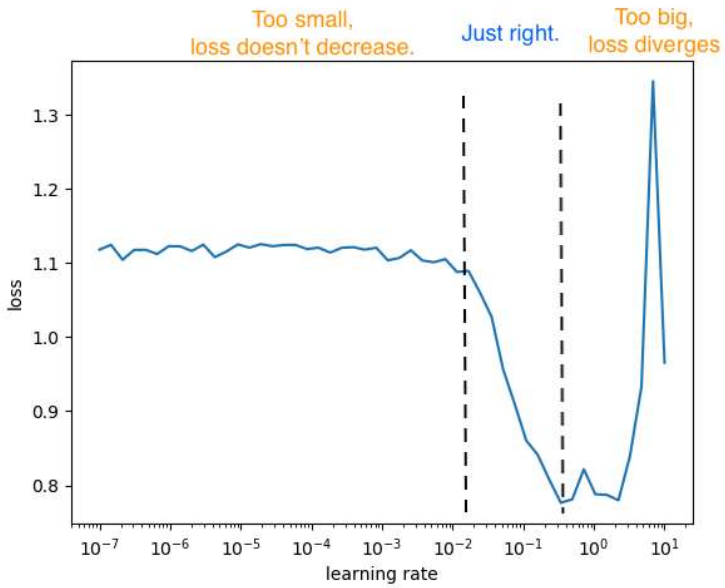
In practice, we choose 2 values for momentum. As in One Cycle, we do 2 step cycle of momentum, wherein in step 1 we reduce momentum from higher to lower bound, and in step 2 we increase momentum from lower to higher bound. According to the paper, this cyclic momentum gives the same final results, but this saves the time and effort of running multiple full cycles with different momentum values.

With the One Cycle Policy and cyclic momentum, I could replicate the results mentioned in the paper. Where the model achieved 91.54% accuracy in 9310 iterations while using one cycle with learning rates 0.08–0.8 and momentum 0.95–0.80 with Resnet-56 and

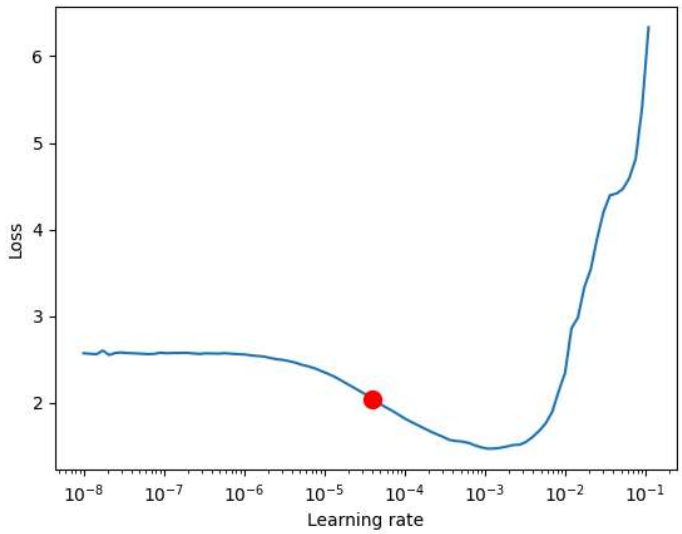
batch size of 512, while without CLR it requires around 64k iterations to achieve this accuracy. (Paper achieved 92.0 ± 0.2 accuracies) .

Steps:

Maximum LR should be picked using the **Learning Rate Finder (https://github.com/davidtvs/pytorch-lr-finder)**



It is recommended to not pick the learning rate that achieves the lowest loss, but instead something in the middle of the sharpest downward slope (redpoint).



The minimum can be set as 10 times smaller than Max LR

Annealing LR should be 10/100 times smaller than the minimum LR.

Don't forget to reset it!

Does it provide us with higher accuracy in practice? NO

Assignment

1. Write a **custom** (https://myrtle.ai/learn/how-to-train-your-resnet-8-bag-of-tricks/) ResNet architecture for CIFAR10 that has the following architecture:
    1. PrepLayer - Conv 3x3 s1, p1) >> BN >> RELU [64k]
    2. Layer1 -
        1. X = Conv 3x3 (s1, p1) >> MaxPool2D >> BN >> RELU [128k]
        2. R1 = ResBlock((Conv-BN-ReLU-Conv-BN-ReLU))(X) [128k]
        3. Add(X, R1)
    3. Layer 2 -
        1. Conv 3x3 [256k]
        2. MaxPooling2D
        3. BN
        4. ReLU
    4. Layer 3 -
        1. X = Conv 3x3 (s1, p1) >> MaxPool2D >> BN >> RELU [512k]
        2. R2 = ResBlock((Conv-BN-ReLU-Conv-BN-ReLU))(X) [512k]
        3. Add(X, R2)
    5. MaxPooling with Kernel Size 4

    6. FC Layer
    7. SoftMax
2. Uses One Cycle Policy such that:
    1. Total Epochs = 24
    2. Max at Epoch = 5
    3. LRMIN = FIND
    4. LRMAX = FIND
    5. NO Annihilation
3. Uses this transform -RandomCrop 32, 32 (after padding of 4) >> FlipLR >> Followed by CutOut(8, 8)
4. Batch size = 512
5. Use ADAM and CrossEntropyLoss
6. Target Accuracy: 90%
7. NO score if your code is not modular. Your collab must be importing your GitHub package, and then just running the model. I should be able to find the custom_resnet.py model in your MASTER GitHub repo that you'd be training.

2. Once done, proceed to answer the Assignment-Solution page.

Video

STUDIO

GM