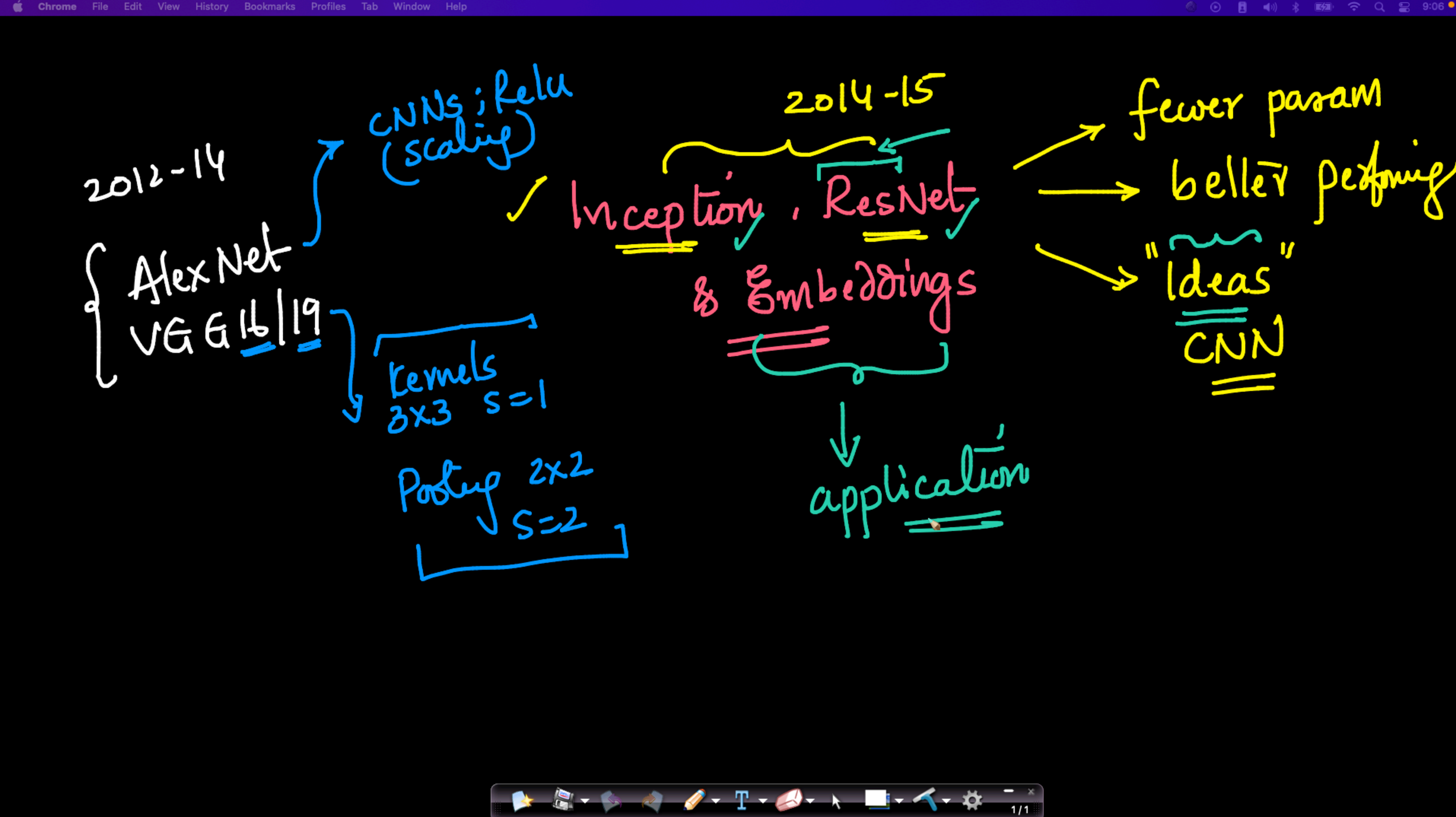

2012-14
AlexNet
VGG16|19
CNNs; Relu
(scaling)
Kernels
3x3 s=1
Pooling 2x2
s=2
2014-15
Inception, ResNet
& Embeddings
fewer param
better perform
"Ideas"
CNN
application

- Original Paper: https://arxiv.org/pdf/1409.4842.pdf

Why this network is important?

- **This Network is important for two reasons:**
- First, the model architecture is tiny compared to AlexNet and VGGNet.
  - The authors are able to obtain such a dramatic drop in network architecture size (while still increasing the depth of the overall network) by removing fully-connected layers and instead using global average pooling.
  - Most of the weights in a CNN can be found in the dense FC layers –
    - if these layers can be removed, the memory savings are massive.

Model Name	Number of params	Top 1 Acc	Top 5 Acc
Alexnet	60M	63.3	84.6
VGG16	138M	74.4	91.9
VGG19	144M	74.5	92.0
GoogleNet	11.2M	74.8	92.2

- Secondly, the Szegedy et al. paper makes usage of a **network in network or micro-architecture** when constructing the overall macro-architecture.
  - Up to this point, we have seen only sequential neural networks where the output of one network feeds directly into the next.
  - We are now going to see micro-architectures, small building blocks that are used inside the rest of the architecture,
    - where the output from one layer can split into a number of various paths and be rejoined later.
  - Specifically, Szegedy et al. contributed the Inception module to the deep learning community, a building block that fits into a

Vincent Vanhoucke
Google Inc.

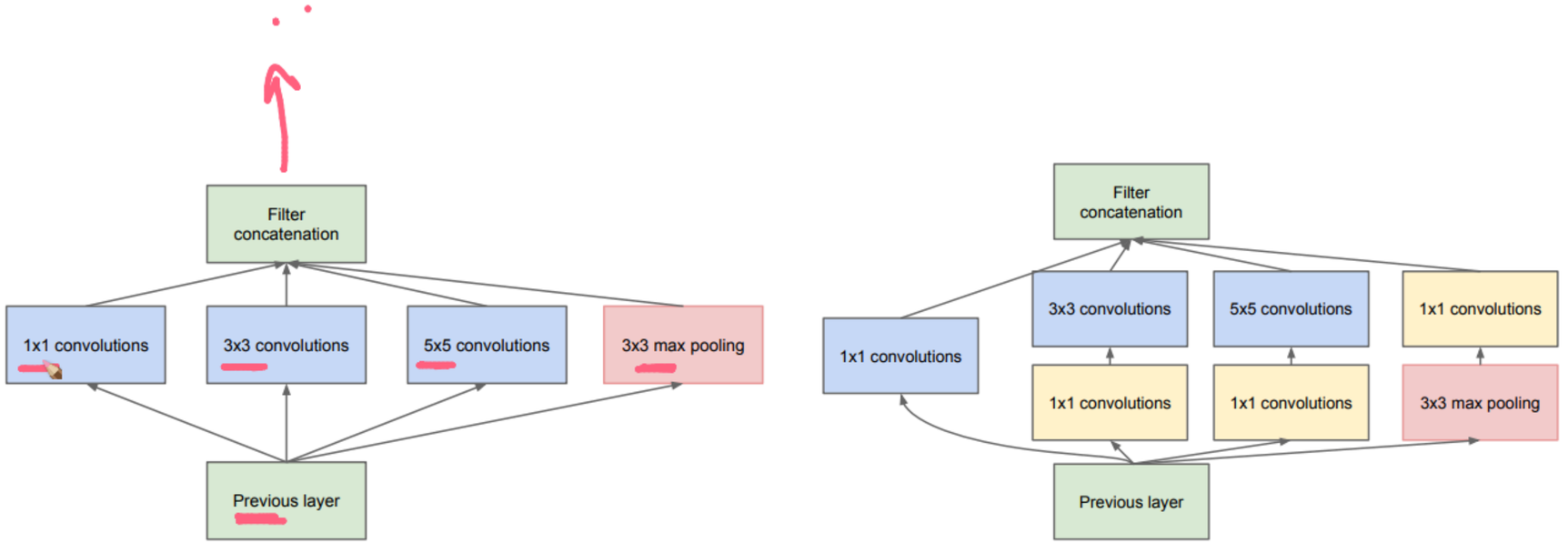
Andrew Rabinovich
Google Inc.

Abstract

We propose a deep convolutional neural network architecture codenamed Inception, which was responsible for setting the new state of the art for classification and detection in the ImageNet Large-Scale Visual Recognition Challenge 2014 (ILSVRC14). The main hallmark of this architecture is the improved utilization of the computing resources inside the network. This was achieved by a carefully crafted design that allows for increasing the depth and width of the network while keeping the computational budget constant. To optimize quality, the architectural decisions were based on the Hebbian principle and the intuition of multi-scale processing. One particular incarnation used in our submission for ILSVRC14 is called GoogLeNet, a 22 layers deep network, the quality of which is assessed in the context of classification and detection.

1 Introduction

In the last three years, mainly due to the advances of deep learning, more concretely convolutional networks [10], the quality of image recognition and object detection has been progressing at a dramatic pace. One encouraging news is that most of this progress is not just the result of more powerful hardware, larger datasets and bigger models, but mainly a consequence of new ideas, algorithms and improved network architectures. No new data sources were used, for example, by the top entries in the ILSVRC 2014 competition besides the classification dataset of the same competition for detection purposes. Our G arameters

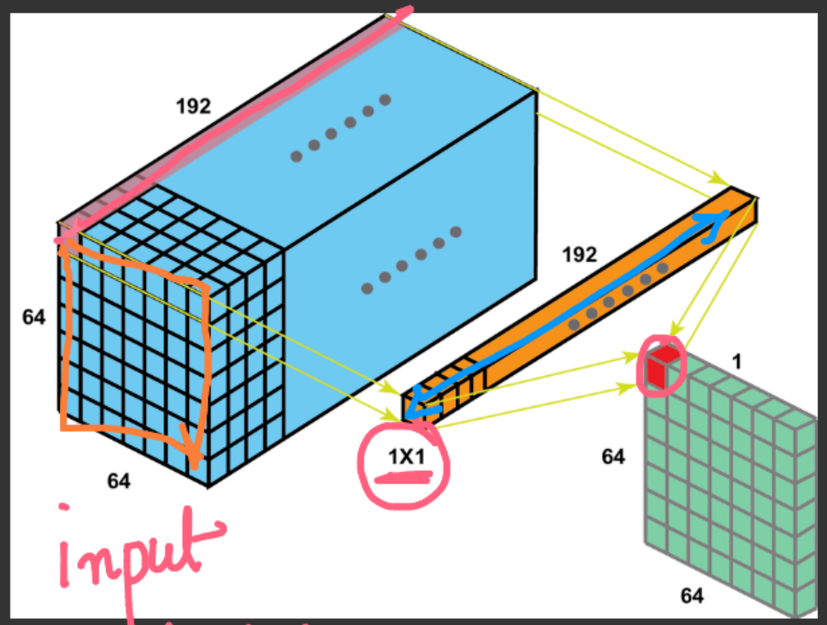


(a) Inception module, naïve version (b) Inception module with dimension reductions

Figure 2: Inception module

increase in the number of outputs from stage to stage. Even while this architecture might cover the optimal sparse structure, it would do it very inefficiently, leading to a computational blow up within a few stages.

- For example, multiplying the three channels of an RGB image by three learnable weights
- and then **adding them up produces a linear combination of the channels**
- And why would we need this?
- It is useful for **adjusting the number of channels of the data.** and they are also **less computationally expensive**



input
64x64x192

64x64x1

1x1 conv

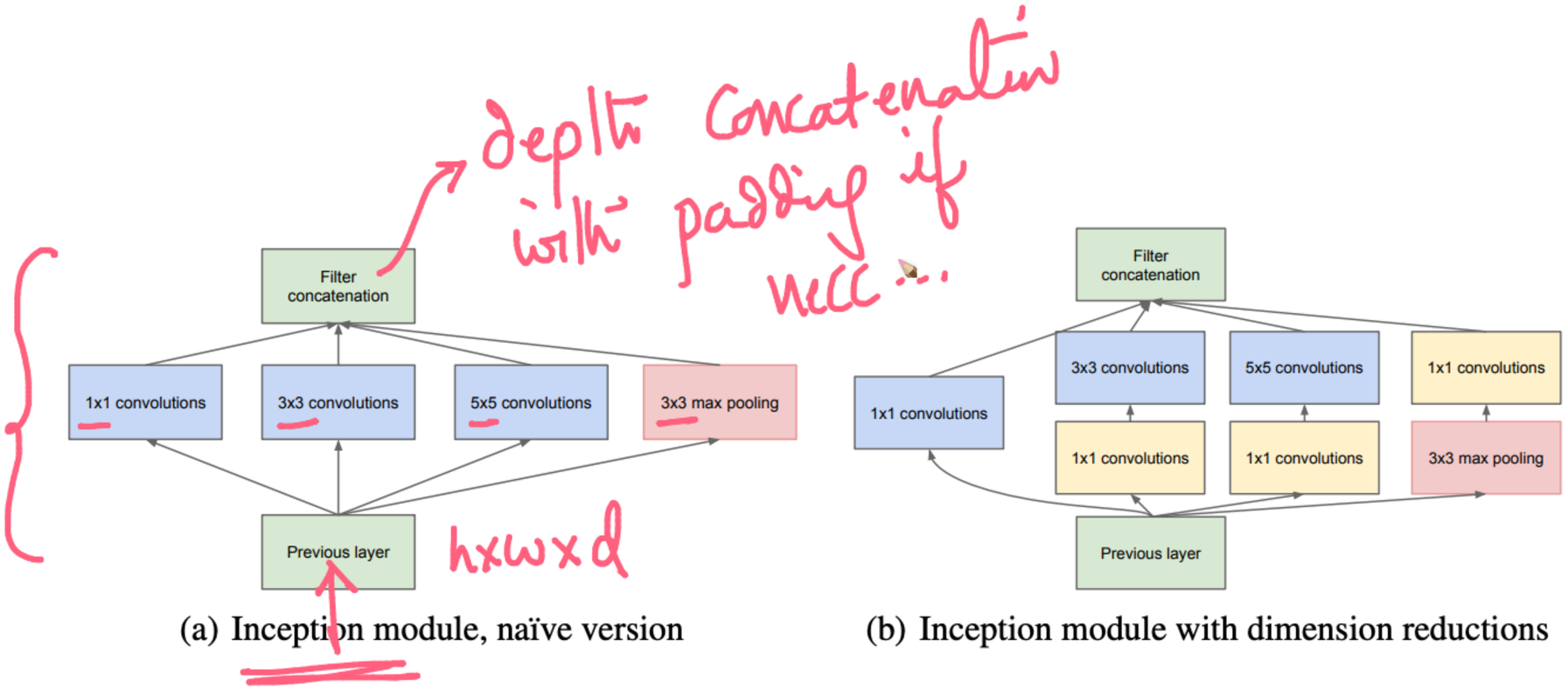
- less param
- GI. Avg. pooling
  64x64 → 1
  Coll. wish

vs

1x1-conv
Collapsing the depth with learnable weights.

- Till now we have looked into 2/3 models
- As the **number of layer is increasing, Accuracy is also increasing**
- So why not have a 500 layer model, wont it attain 99 % acc?

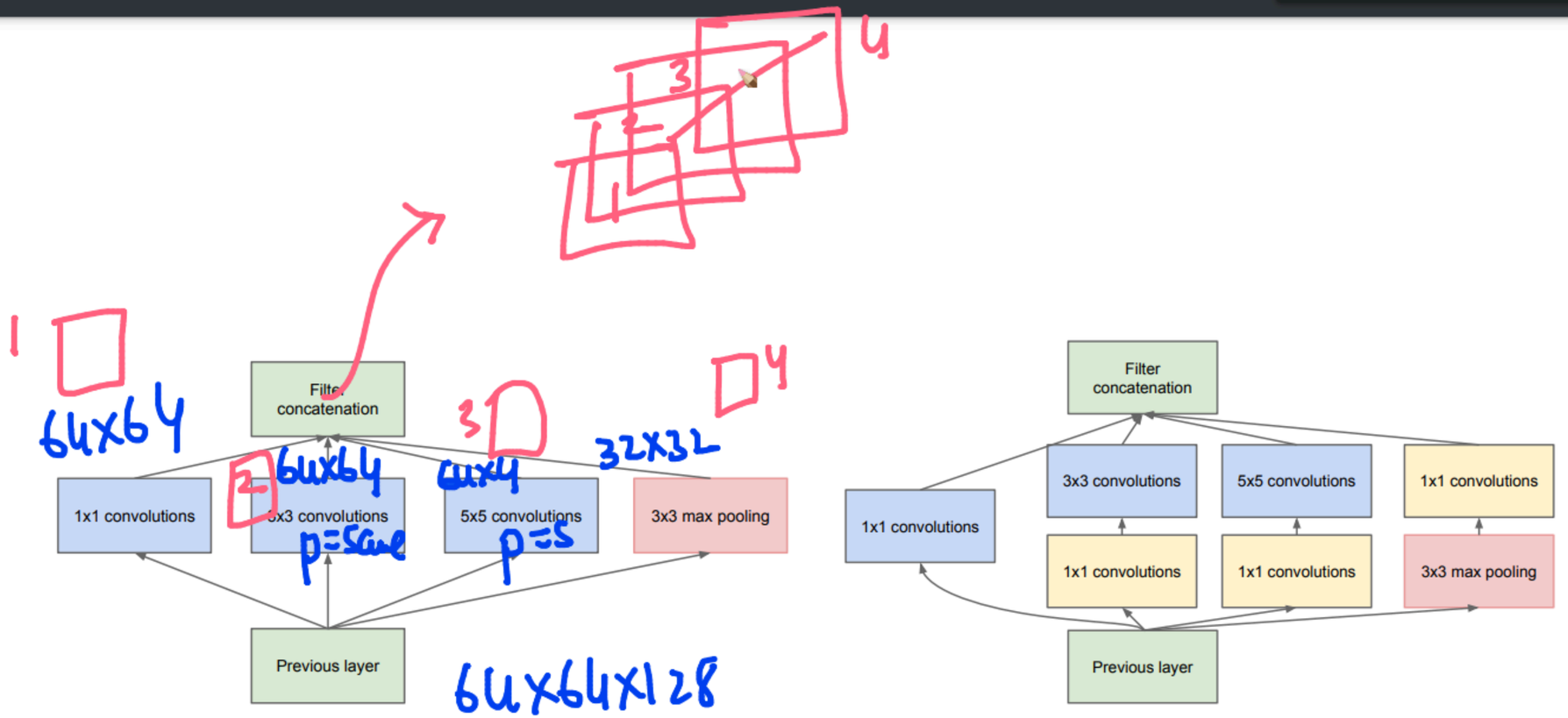
Why we need to increase the depth of network ?



(a) Inception module, naïve version (b) Inception module with dimension reductions

Figure 2: Inception module

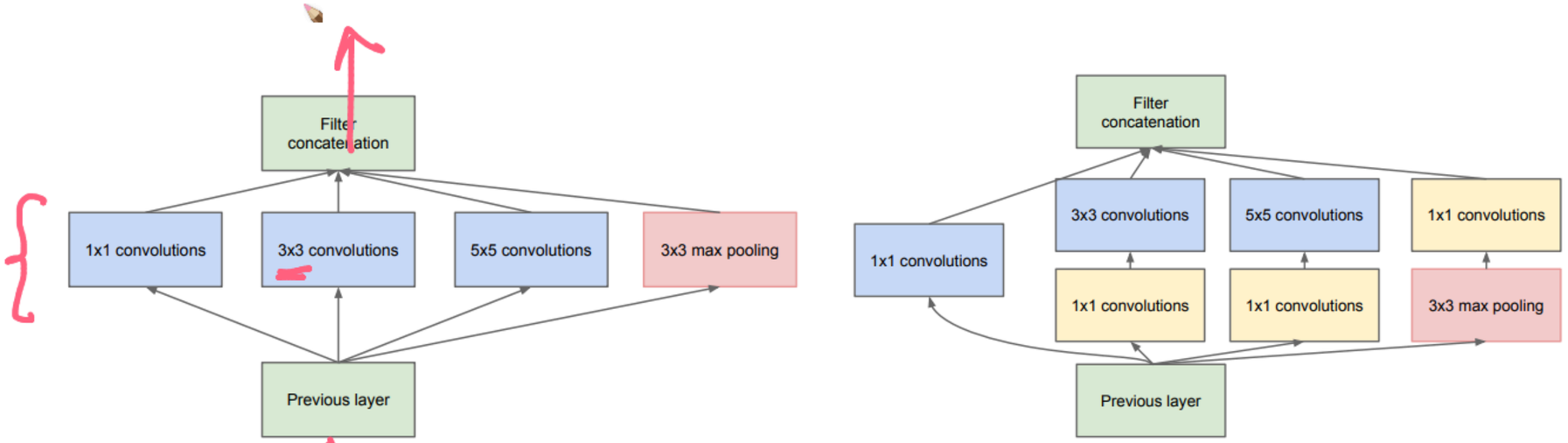
increase in the number of outputs from stage to stage. Even while this architecture might cover the optimal sparse structure, it would do it very inefficiently, leading to a computational blow up within a few stages.



(a) Inception module, naïve version (b) Inception module with dimension reductions

Figure 2: Inception module

increase in the number of outputs from stage to stage. Even while this architecture might cover the optimal sparse structure, it would do it very inefficiently, leading to a computational blow up within a few stages.

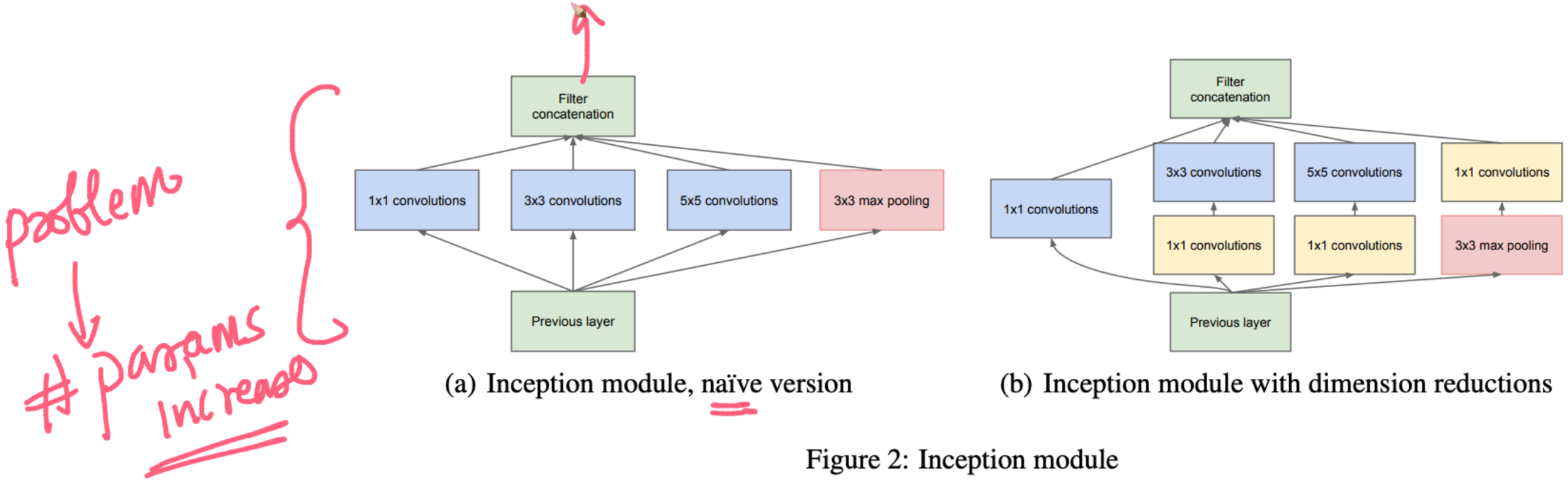


(a) Inception module, naïve version

(b) Inception module with dimension reductions

Figure 2: Inception module

increase in the number of outputs from stage to stage. Even while this architecture might cover the optimal sparse structure, it would do it very inefficiently, leading to a computational blow up within a few stages.

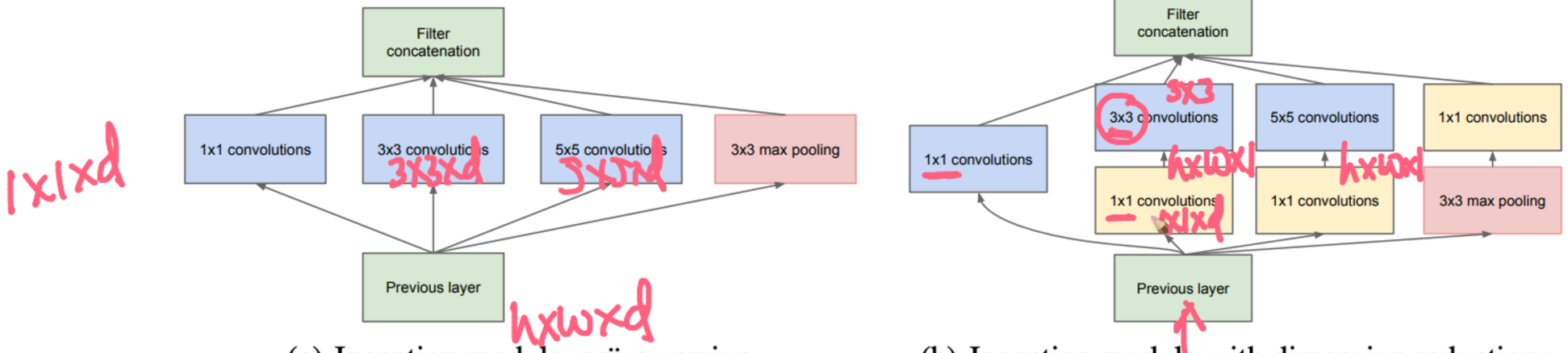


(a) Inception module, naïve version

(b) Inception module with dimension reductions

Figure 2: Inception module

increase in the number of outputs from stage to stage. Even while this architecture might cover the optimal sparse structure, it would do it very inefficiently, leading to a computational blow up within a few stages.

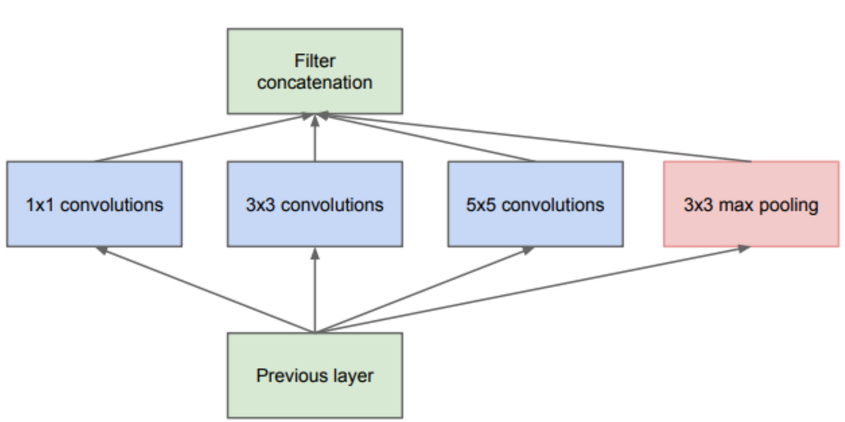


(a) Inception module, naïve version (b) Inception module with dimension reductions

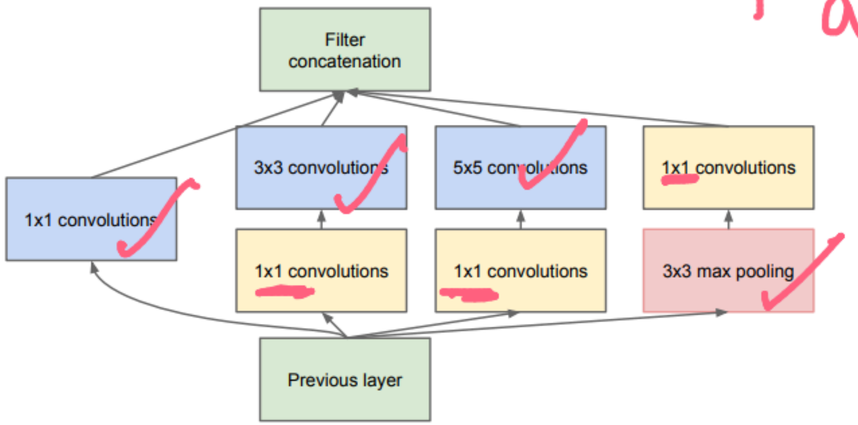
Figure 2: Inception module

increase in the number of outputs from stage to stage. Even while this architecture might cover the optimal sparse structure, it would do it very inefficiently, leading to a computational blow up within a few stages.

Smaller Tr-time
#params
are fewer



(a) Inception module, naïve version

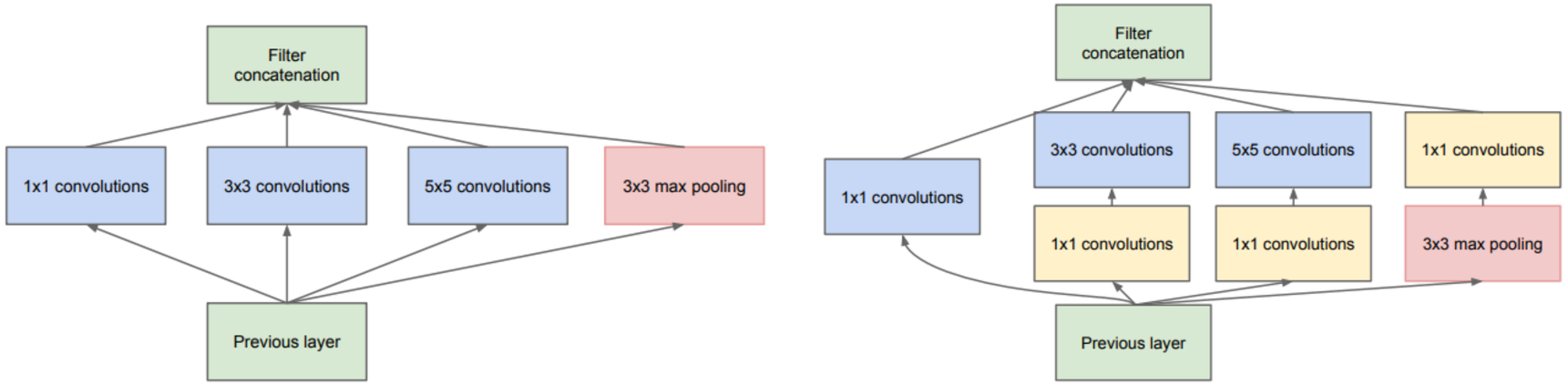


(b) Inception module with dimension reductions

Figure 2: Inception module

increase in the number of outputs from stage to stage. Even while this architecture might cover the optimal sparse structure, it would do it very inefficiently, leading to a computational blow up within a few stages.

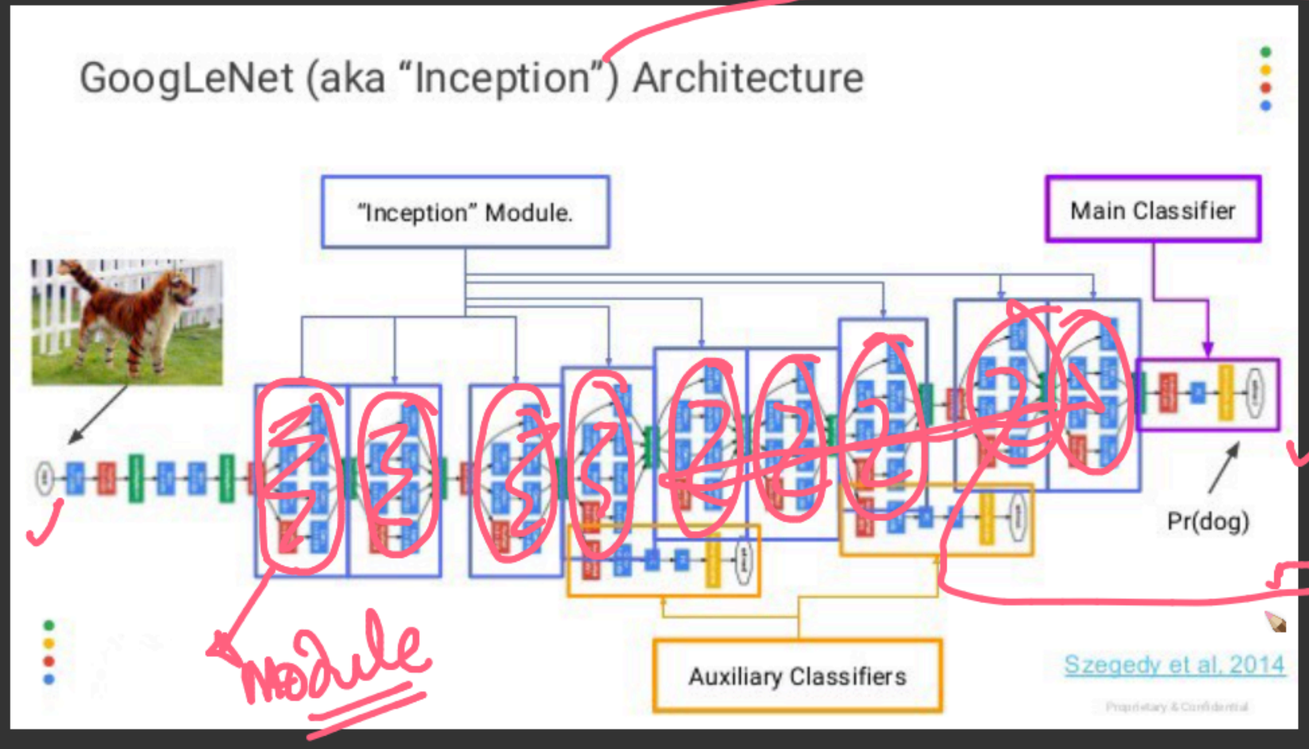
1x1 ConV



(a) Inception module, naïve version (b) Inception module with dimension reductions

Figure 2: Inception module

increase in the number of outputs from stage to stage. Even while this architecture might cover the optimal sparse structure, it would do it very inefficiently, leading to a computational blow up within a few stages.

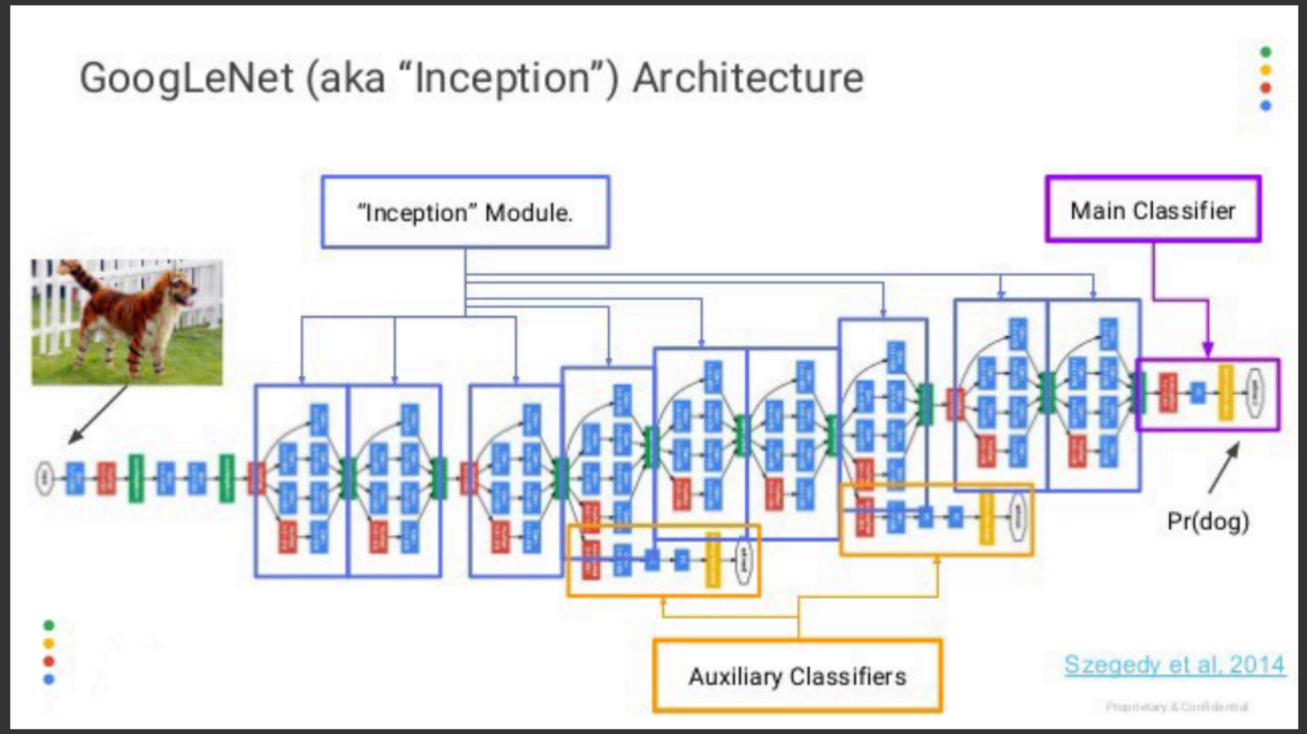


- The network is 22 layers deep when counting only layers with parameters (or 27 layers if we also count pooling).
- Please refer the link to see code of full model architecture: https://www.analyticsvidhya.com/blog/2018/10/understanding-inception-network-from-scratch/

Description of Architecture:

Specifically take note of how the Inception module branches into four distinct paths from the input layer.

- The first branch in the Inception module simpl



Key ideas
- Inception module
  - use multiple kernels
  - 1x1 kernels to reduce #params
- → Inception module as a building block
- → Gl. Avg-pooling

- The network is 22 layers deep when counting only layers with parameters (or 27 layers if we also count pooling).
- Please refer the link to see code of full model architecture: https://www.analyticsvidhya.com/blog/2018/10/understanding-inception-network-from-scratch/

Description of Architecture:

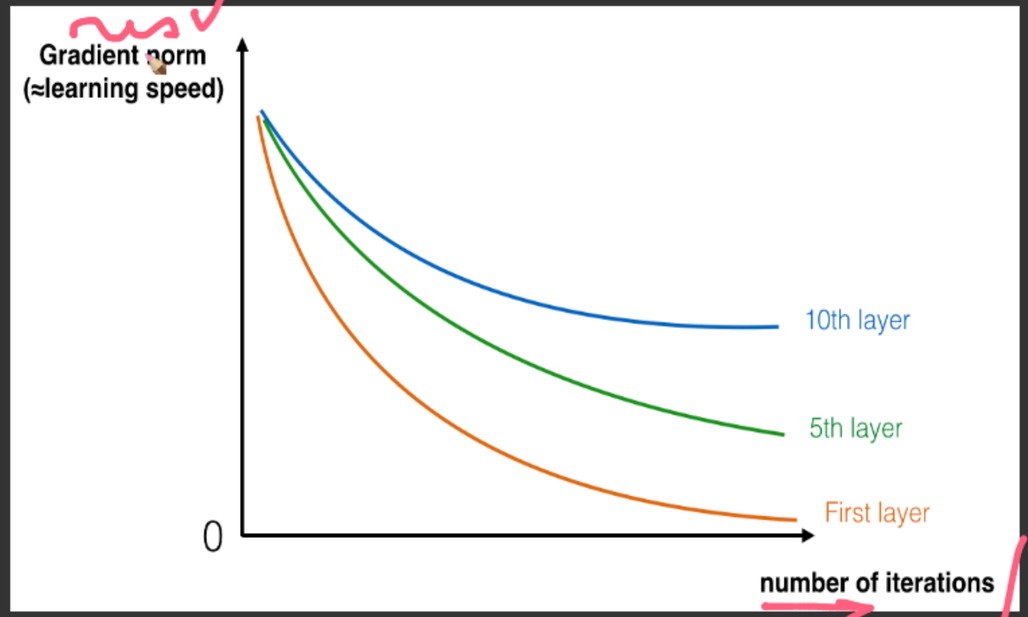
Specifically take note of how the Inception module branches into four distinct paths from the input layer.

- The first branch in the Inception module simpl

weight matrix on each step, and

- thus the gradient can decrease exponentially quickly to zero (or, in rare cases, grow exponentially quickly and "explode" to take very large values).

During training, you might therefore see the magnitude (or norm) of the gradient for the earlier layers descrease to zero very rapidly as training proceeds:



$$\nabla_{W_1} L = \begin{bmatrix} \vdots \end{bmatrix}$$

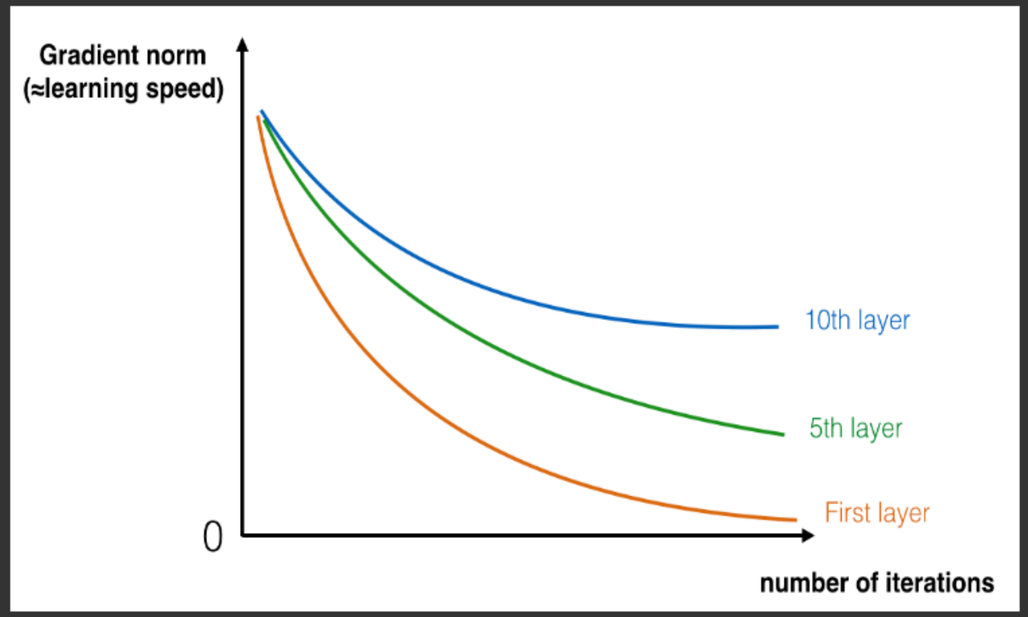
Figure 1 : **Vanishing gradient** The speed of learning decreases very rapidly for the early layers as the network trains

How we are going to solve this issue ?

You are now going to solve this problem by building a Residual Network!

weight matrix on each step, and
- thus the gradient can decrease exponentially quickly to zero (or, in rare cases, grow exponentially quickly and "explode" to take very large values).

During training, you might therefore see the magnitude (or norm) of the gradient for the earlier layers descrease to zero very rapidly as training proceeds:



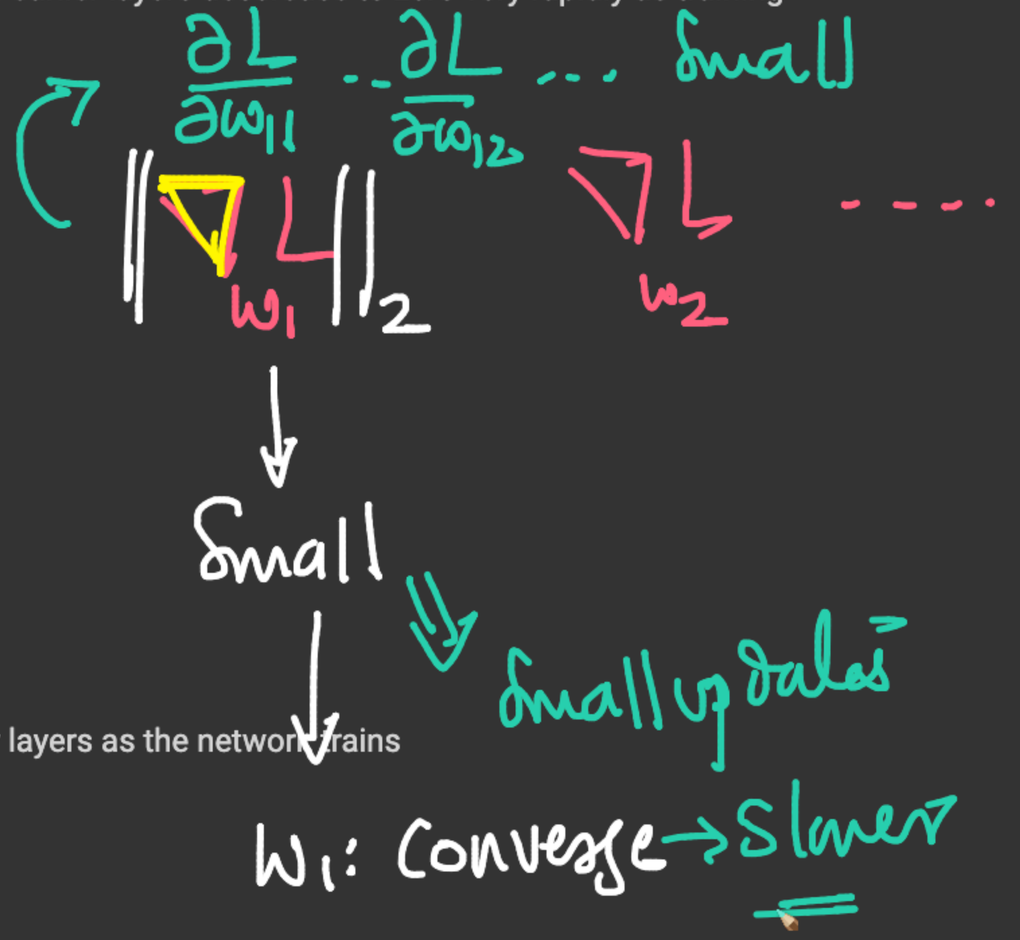


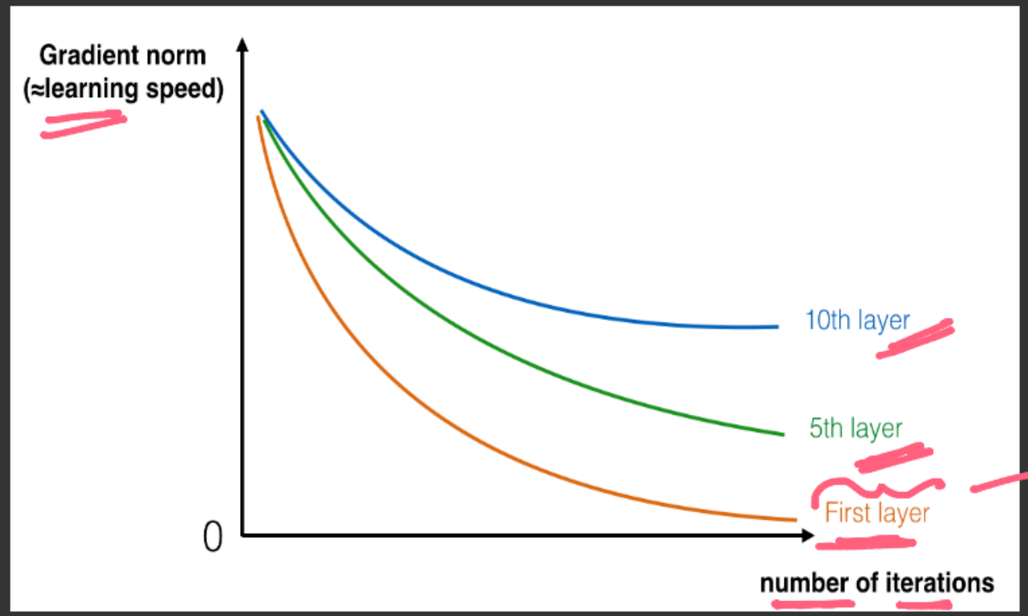
Figure 1 : **Vanishing gradient** The speed of learning decreases very rapidly for the early layers as the network trains

How we are going to solve this issue ?

You are now going to solve this problem by building a Residual Network!

weight matrix on each step, and
- thus the gradient can decrease exponentially quickly to zero (or, in rare cases, grow exponentially quickly and "explode" to take very large values).

During training, you might therefore see the magnitude (or norm) of the gradient for the earlier layers descrease to zero very rapidly as training proceeds:



50, 100, 150

1st layer weights are not changing much

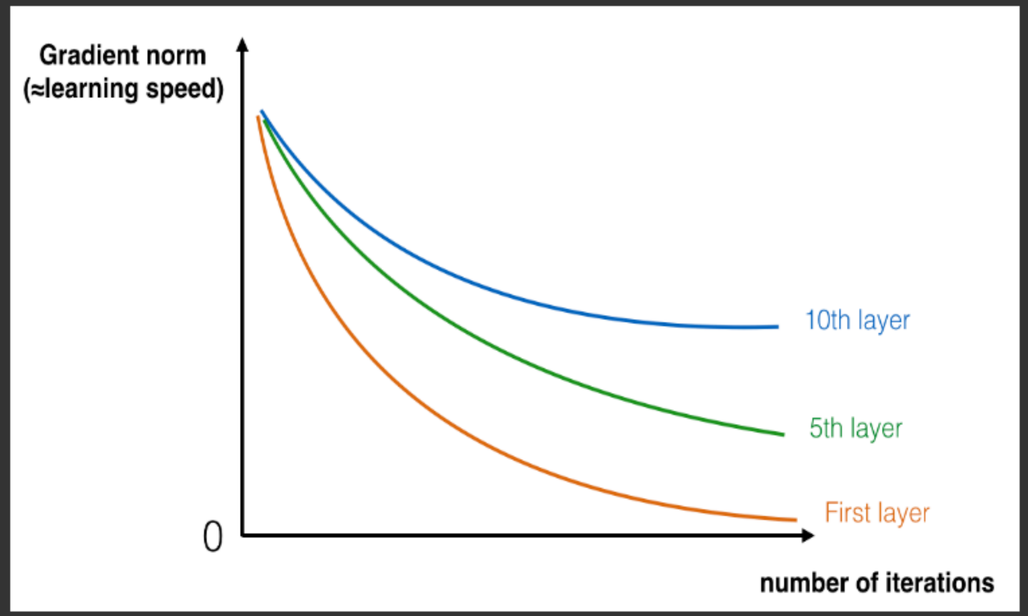
Figure 1 : **Vanishing gradient** The speed of learning decreases very rapidly for the early layers as the network trains

How we are going to solve this issue ?

You are now going to solve this problem by building a Residual Network!

weight matrix on each step, and
- thus the gradient can decrease exponentially quickly to zero (or, in rare cases, grow exponentially quickly and "explode" to take very large values).

During training, you might therefore see the magnitude (or norm) of the gradient for the earlier layers descrease to zero very rapidly as training proceeds:



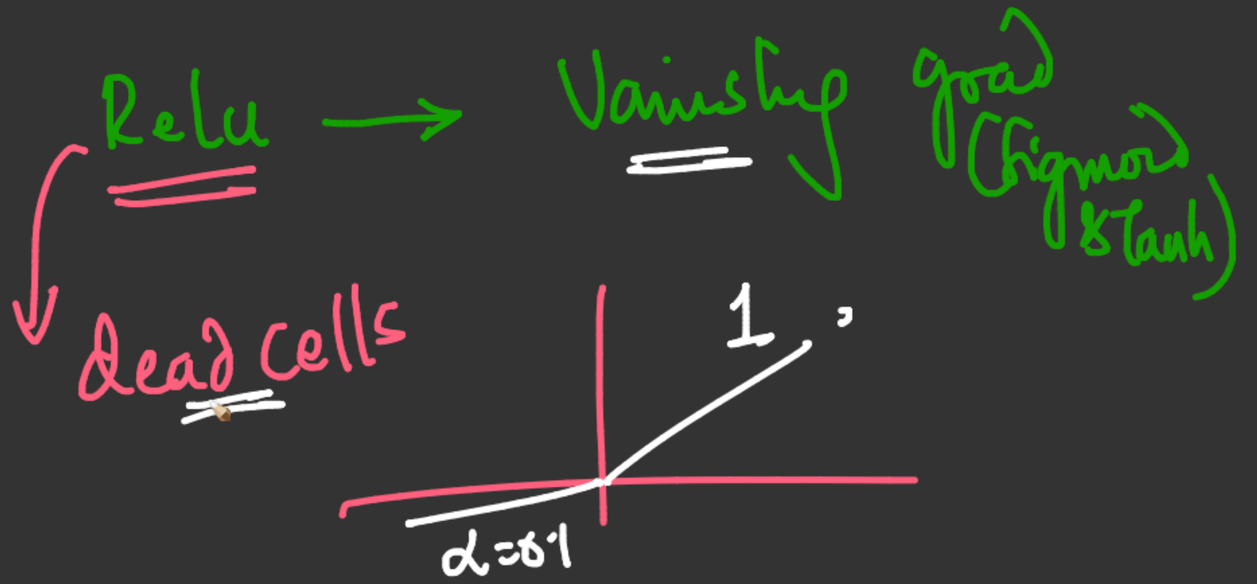


Figure 1 : **Vanishing gradient** The speed of learning decreases very rapidly for the early layers as the network trains

How we are going to solve this issue ?

You are now going to solve this problem by building a Residual Network!

weight matrix on each step, and
- thus the gradient can decrease exponentially quickly to zero (or, in rare cases, grow exponentially quickly and "explode" to take very large values).

During training, you might therefore see the magnitude (or norm) of the gradient for the earlier layers descrease to zero very rapidly as training proceeds:

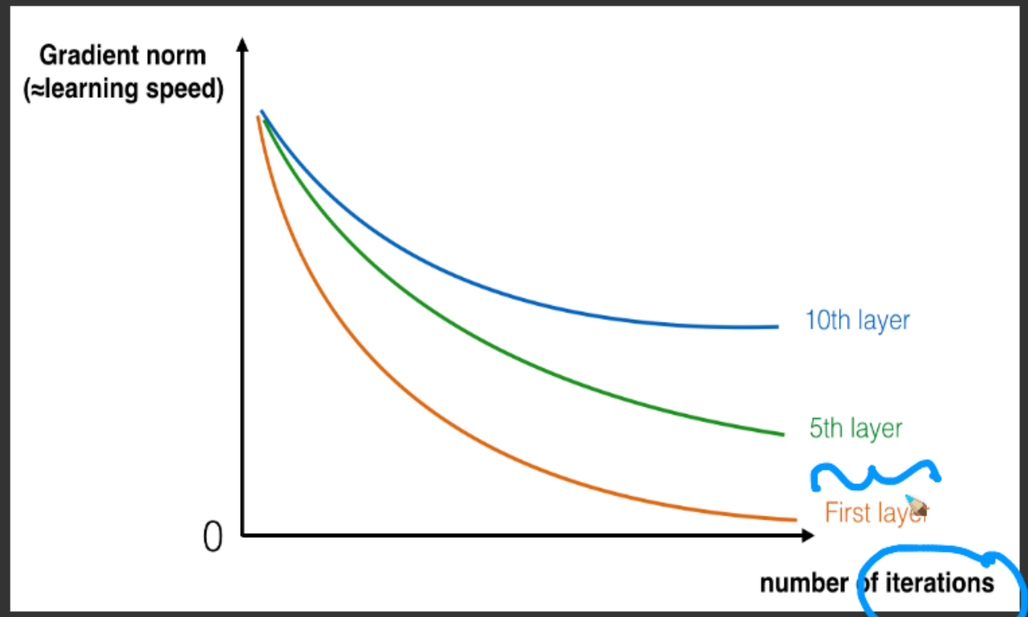


Figure 1 : **Vanishing gradient** The speed of learning decreases very rapidly for the early layers as the network trains

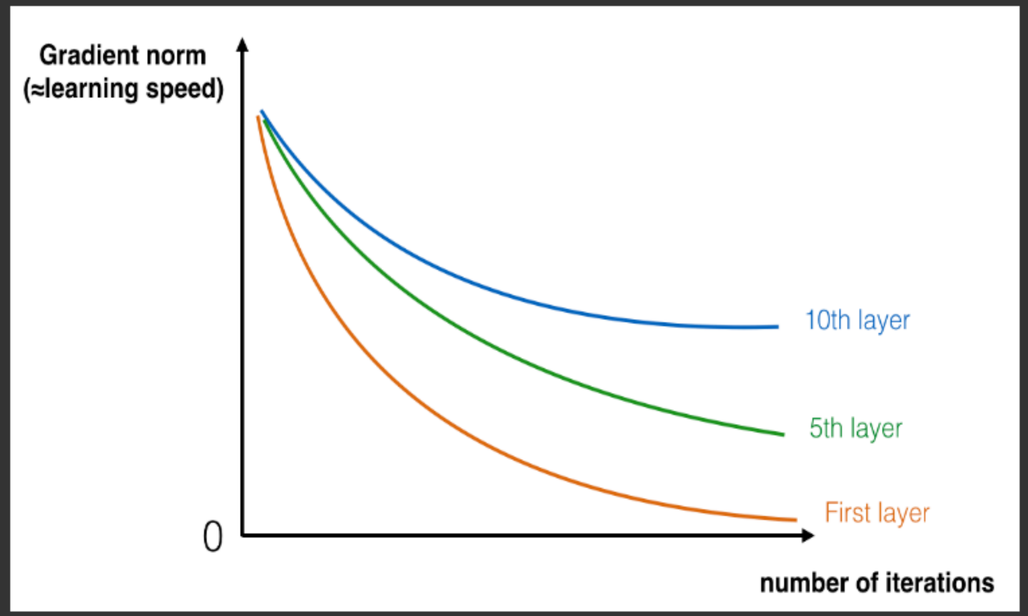
How we are going to solve this issue ?

You are now going to solve this problem by building a Residual Network!

weight matrix on each step, and

- thus the gradient can decrease exponentially quickly to zero (or, in rare cases, grow exponentially quickly and "explode" to take very large values).

During training, you might therefore see the magnitude (or norm) of the gradient for the earlier layers descrease to zero very rapidly as training proceeds:



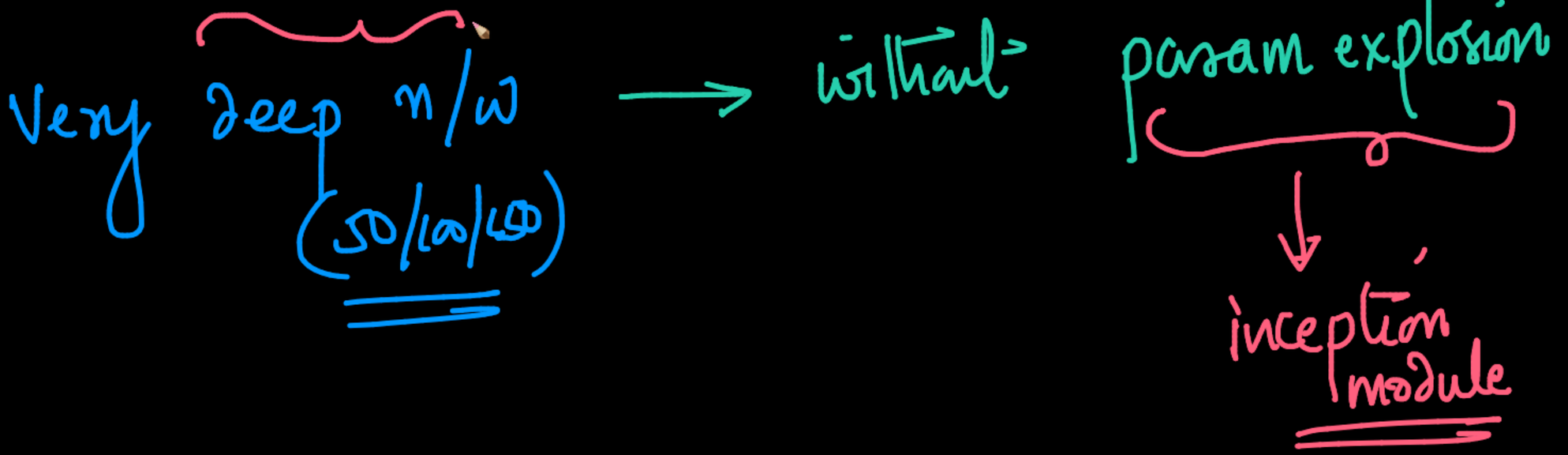
27 - layers n/w → 2014

{ 50/100/200

Figure 1 : **Vanishing gradient** The speed of learning decreases very rapidly for the early layers as the network trains

How we are going to solve this issue ?

You are now going to solve this problem by building a Residual Network!



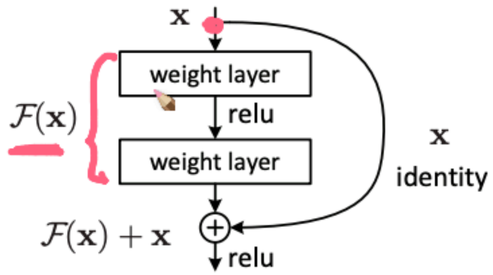


Figure 2. Residual learning: a building block.

are comparably good or better than the constructed solution (or unable to do so in feasible time).

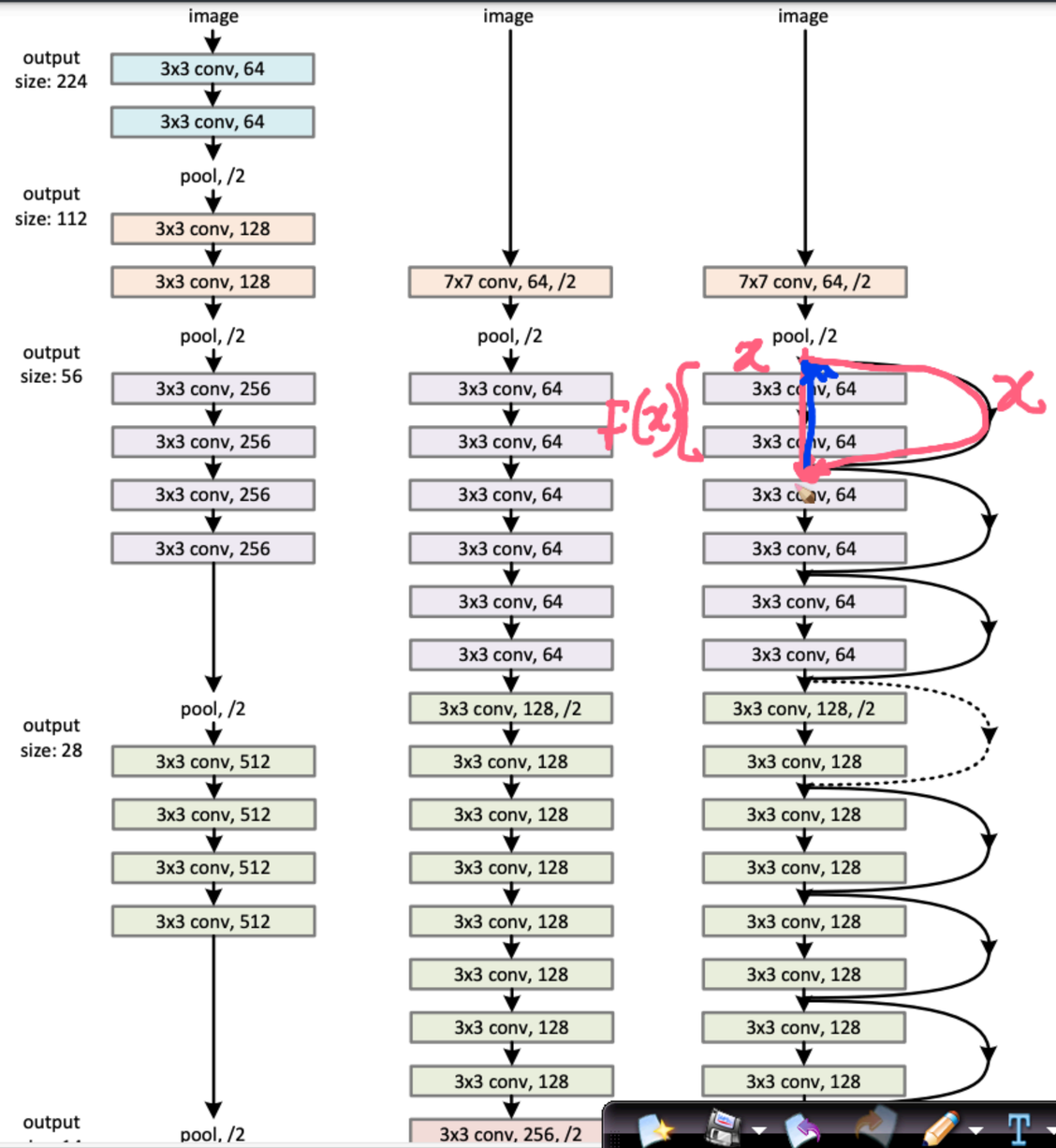
In this paper, we address the degradation problem by introducing a *deep residual learning* framework. Instead of hoping each few stacked layers directly fit a desired underlying mapping, we explicitly let these layers fit a residual mapping. Formally, denoting the desired underlying mapping as $\mathcal{H}(\mathbf{x})$, we let the stacked nonlinear layers fit another mapping of $\mathcal{F}(\mathbf{x}) := \mathcal{H}(\mathbf{x}) - \mathbf{x}$. The original mapping is recast into $\mathcal{F}(\mathbf{x}) + \mathbf{x}$. We hypothesize that it is easier to optimize the residual mapping than to optimize the original, unreferenced mapping. To the extreme, if an identity mapping were optimal, it would be easier to push the residual to zero than to fit an ide

ImageNet *test* set, and *won the 1st place in the ILSVRC 2015 classification competition*. The extremely deep representations also have excellent generalization performance on other recognition tasks, and lead us to further *win the 1st places on: ImageNet detection, ImageNet localization, COCO detection, and COCO segmentation* in ILSVRC & COCO 2015 competitions. This strong evidence shows that the residual learning principle is generic, and we expect that it is applicable in other vision and non-vision problems.

2. Related Work

Residual Representations. In image recognition, VLAD [18] is a representation that encodes by the residual vectors with respect to a dictionary, and Fisher Vector [30] can be formulated as a probabilistic version [18] of VLAD. Both of them are powerful shallow representations for image retrieval and classification [4, 48]. For vector quantization, encoding residual vectors [17] is shown to be more effective than encoding original vectors.

In low-level vision and computer graphics, for solving Partial Differential Equations (PDEs), the widely used system as subprob-



insert shortcut connections (Fig. 3, right) which turn the network into its counterpart residual version. The identity shortcuts (Eqn.(1)) can be directly used when the input and output are of the same dimensions (solid line shortcuts in Fig. 3). When the dimensions increase (dotted line shortcuts in Fig. 3), we consider two options: (A) The shortcut still performs identity mapping, with extra zero entries padded for increasing dimensions. This option introduces no extra parameter; (B) The projection shortcut in Eqn.(2) is used to match dimensions (done by 1×1 convolutions). For both options, when the shortcuts go across feature maps of two sizes, they are performed with a stride of 2.

3.4. Implementation

Our implementation for ImageNet follows the practice in [21, 41]. The image is resized with its shorter side randomly sampled in $[256, 480]$ for scale augmentation [41]. A 224×224 crop is randomly sampled from an image or its horizontal flip, with the per-pixel mean subtracted [21]. The standard color augmentation in [21] is used. We adopt batch normalization (BN) [16] right after each convolution and before activation, following [16]. We initialize the weights [illegible]ain/residual nets from scratch. We

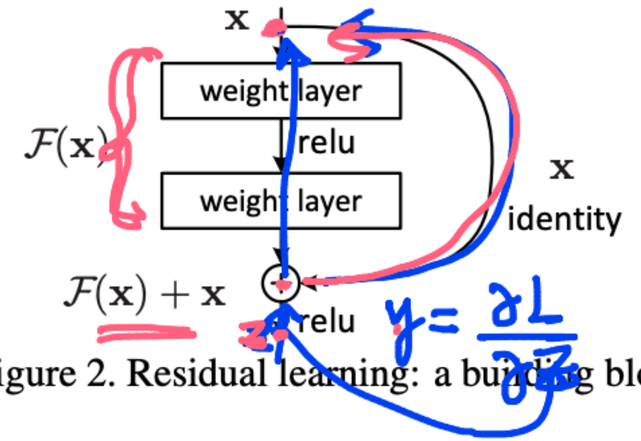


Figure 2. Residual learning: a building block.

are comparably good or better than the constructed solution (or unable to do so in feasible time).

In this paper, we address the degradation problem by introducing a *deep residual learning* framework. Instead of hoping each few stacked layers directly fit a desired underlying mapping, we

ImageNet *test* set, and *won the 1st place in the ILSVRC 2015 classification competition*. The extremely deep representations also have excellent generalization performance on other recognition tasks, and lead us to further *win the 1st places on: ImageNet detection, ImageNet localization, COCO detection, and COCO segmentation* in ILSVRC & COCO 2015 competitions. This strong evidence shows that the residual learning principle is generic, and we expect that it is applicable in other vision and non-vision problems.

2. Related Work

Residual Representations. In image recognition, VLAD [18] is a representation that encodes by the residual vectors

ary, and Fisher Vector [30] can be

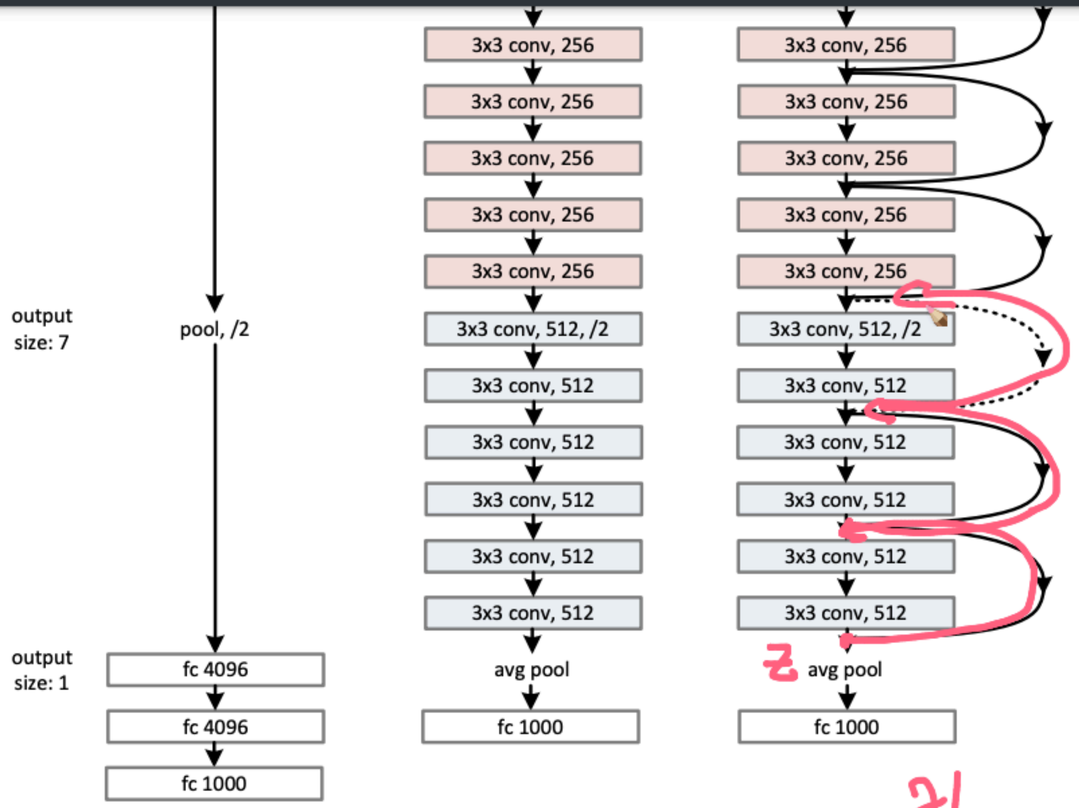


Figure 3. Example network architectures for ImageNet. **Left**: the VGG-19 model [41] (19.6 billion FLOPs) as a reference. **Middle**: a plain network with 34 parameter layers (3.6 billion FLOPs). **Right**: a residual network with 34 parameter layers (3.6 billion FLOPs). The dotted shortcuts increase dimensions. **Table 1** shows more details and other variants.

convolutional form as in [41, 13], and average the scores at multiple scales (images are resized such that the shorter side is in $\{224, 256, 384, 480, 640\}$).

4. Experiments

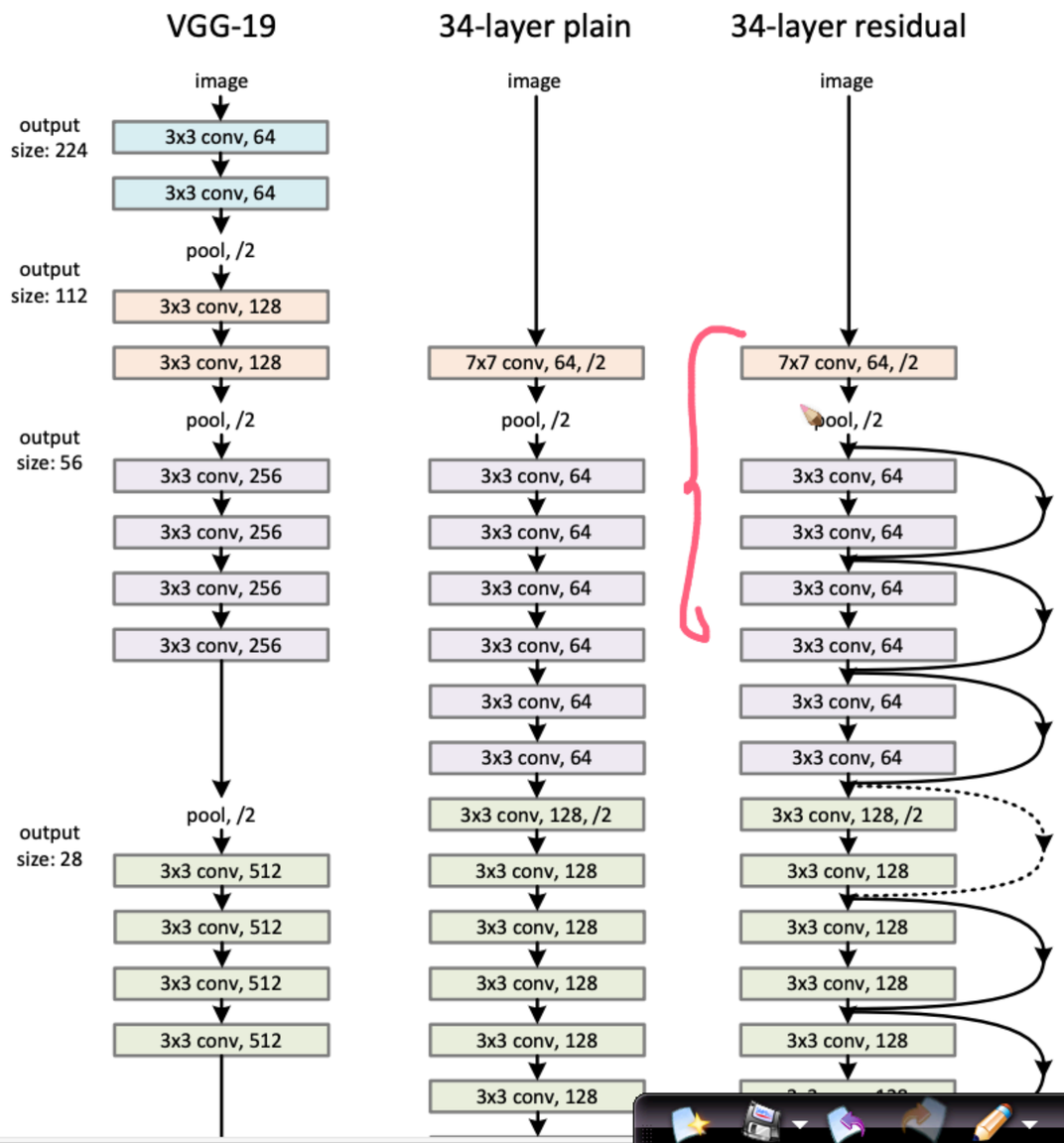
4.1. ImageNet Classification

We evaluate our method on the ImageNet 2012 classification dataset [36] that consists of 1000 classes. The models are trained on the 1.28 million training images, and evaluated on the 50k validation images. We also obtain a final result on the 100k test images, reported by the test server. We evaluate both top-1 and top-5 error rates.

Plain Networks. We first evaluate 18-layer and 34-layer plain nets. The 34-layer plain net is in Fig. 3 (middle). The 18-layer plain net is of a similar form. See Table 1 for detailed architectures.

The results in Table 2 show that the deeper 34-layer plain net has higher validation error than the shallower 18-layer plain net. To reveal the reasons, in Fig. 4 (left) we compare their training/validation errors during the training pro-

ed the degradation problem - the



Residual Network. Based on the above plain network, we insert shortcut connections (Fig. 3, right) which turn the network into its counterpart residual version. The identity shortcuts (Eqn.(1)) can be directly used when the input and output are of the same dimensions (solid line shortcuts in Fig. 3). When the dimensions increase (dotted line shortcuts in Fig. 3), we consider two options: (A) The shortcut still performs identity mapping, with extra zero entries padded for increasing dimensions. This option introduces no extra parameter; (B) The projection shortcut in Eqn.(2) is used to match dimensions (done by 1×1 convolutions). For both options, when the shortcuts go across feature maps of two sizes, they are performed with a stride of 2.

3.4. Implementation

Our implementation for ImageNet follows the practice in [21, 41]. The image is resized with its shorter side randomly sampled in $[256, 480]$ for scale augmentation [41]. A 224×224 crop is randomly sampled from an image or its horizontal flip, with the per-pixel mean subtracted [21]. The standard color augmentation in [21] is used. We adopt batch

large values).

During training, you might therefore see the magnitude (or norm) of the gradient for the earlier layers descrease to zero very rapidly as training proceeds:

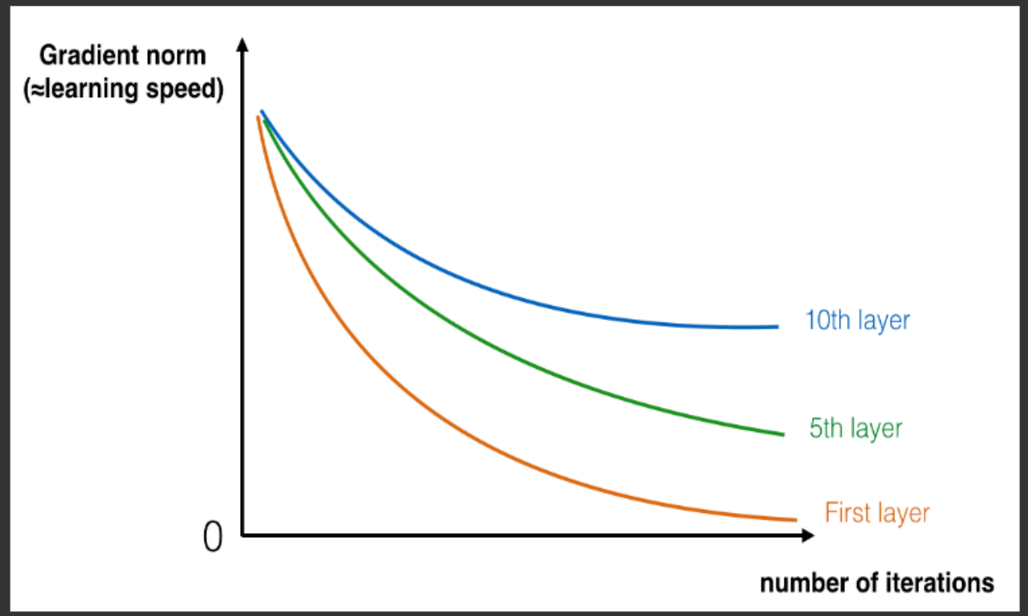
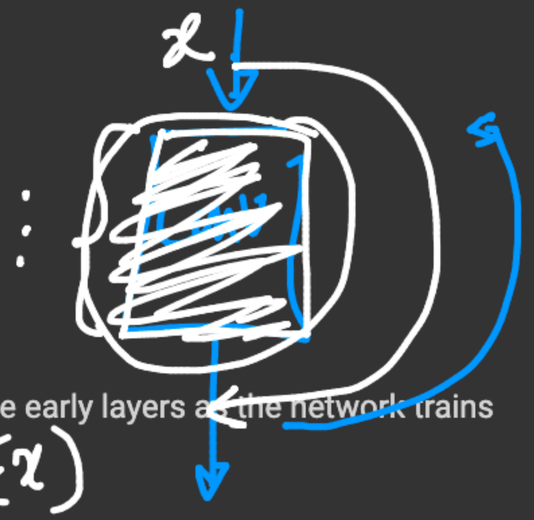


Figure 1 : **Vanishing gradient** The speed of learning decreases very rapidly for the early layers as the network trains

Can work as a form of regularization.

Res-conn → Vanishing gradient problem



is more or less taken care of even if you have lost layers

How we are going to solve this issue ?

You are now going to solve this problem by building a Residual Network!

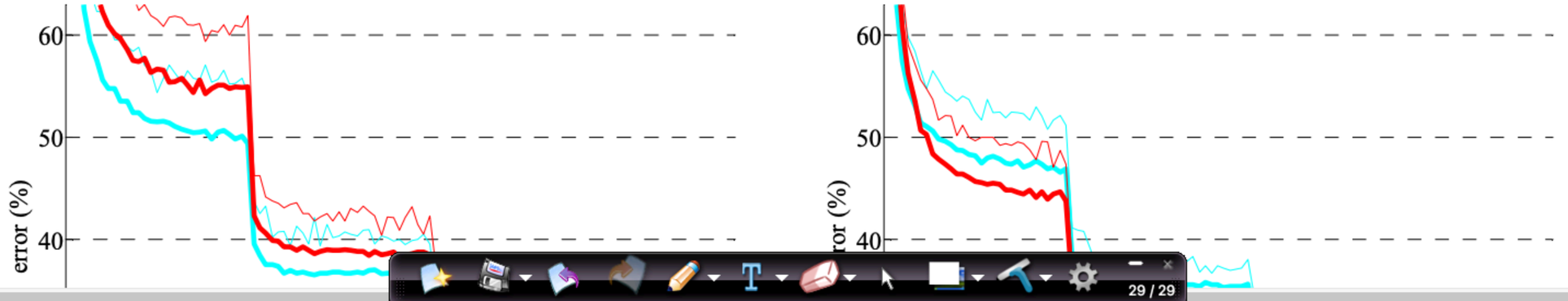
ResNet

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				
		$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix} \times 2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix} \times 3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix} \times 3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix} \times 3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix} \times 3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix} \times 2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix} \times 4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix} \times 4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix} \times 4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix} \times 8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix} \times 2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix} \times 6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix} \times 6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix} \times 23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix} \times 36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix} \times 2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix} \times 3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix} \times 3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix} \times 3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix} \times 3$
	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

Table 1. Architectures for ImageNet. Building blocks are shown in brackets (see also Fig. 5), with the numbers of blocks stacked. Down-sampling is performed by conv3_1, conv4_1, and conv5_1 with a stride of 2.

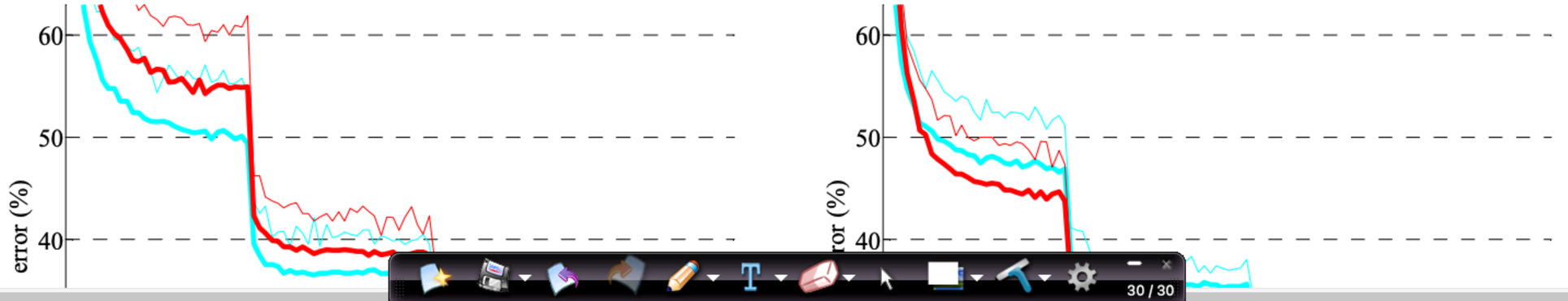
layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
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

Table 1. Architectures for ImageNet. Building blocks are shown in brackets (see also Fig. 5), with the numbers of blocks stacked. Downsampling is performed by conv3_1, conv4_1, and conv5_1 with a stride of 2.



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

Table 1. Architectures for ImageNet. Building blocks are shown in brackets (see also Fig. 5), with the numbers of blocks stacked. Down-sampling is performed by conv3_1, conv4_1, and conv5_1 with a stride of 2.



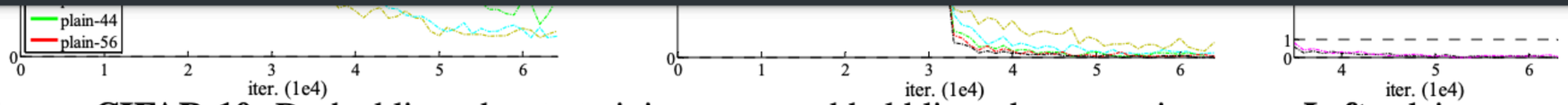


Figure 6. Training on **CIFAR-10**. Dashed lines denote training error, and bold lines denote testing error. **Left**: plain networks. The error of plain-110 is higher than 60% and not displayed. **Middle**: ResNets. **Right**: ResNets with 110 and 1202 layers.

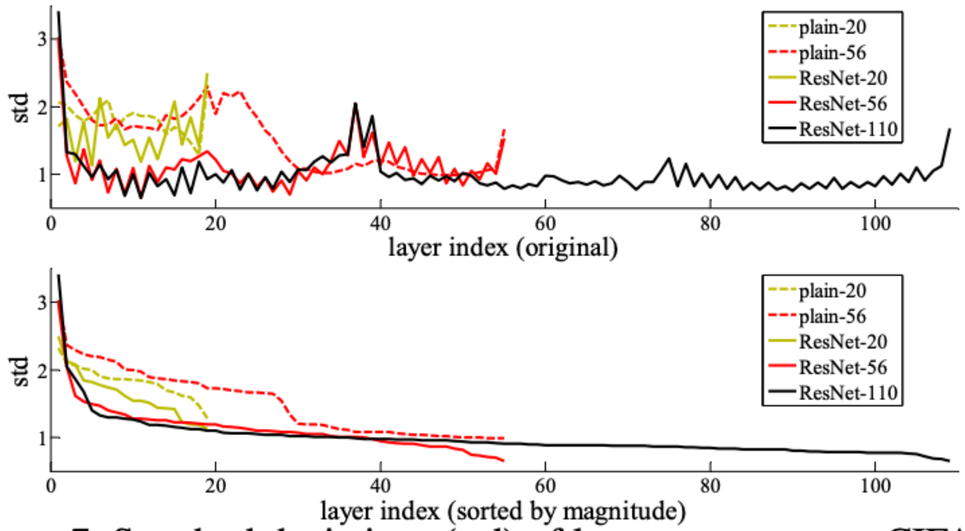


Figure 7. Standard deviations (std) of layer responses on CIFAR-10. The responses are the outputs of each 3×3 layer, after BN and before nonlinearity. **Top**: the layers are shown in their original order. **Bottom**: the responses are ranked in descending order.

training data	07+12	07++12
test data	VOC 07 test	VOC 12 test
VGG-16	73.2	70.4
ResNet-101	**76.4**	**73.8**

Table 7. Object detection mAP (%) on the PASCAL VOC 2007/2012 test sets using **baseline** Faster R-CNN. See also Table 10 and 11 for better results.

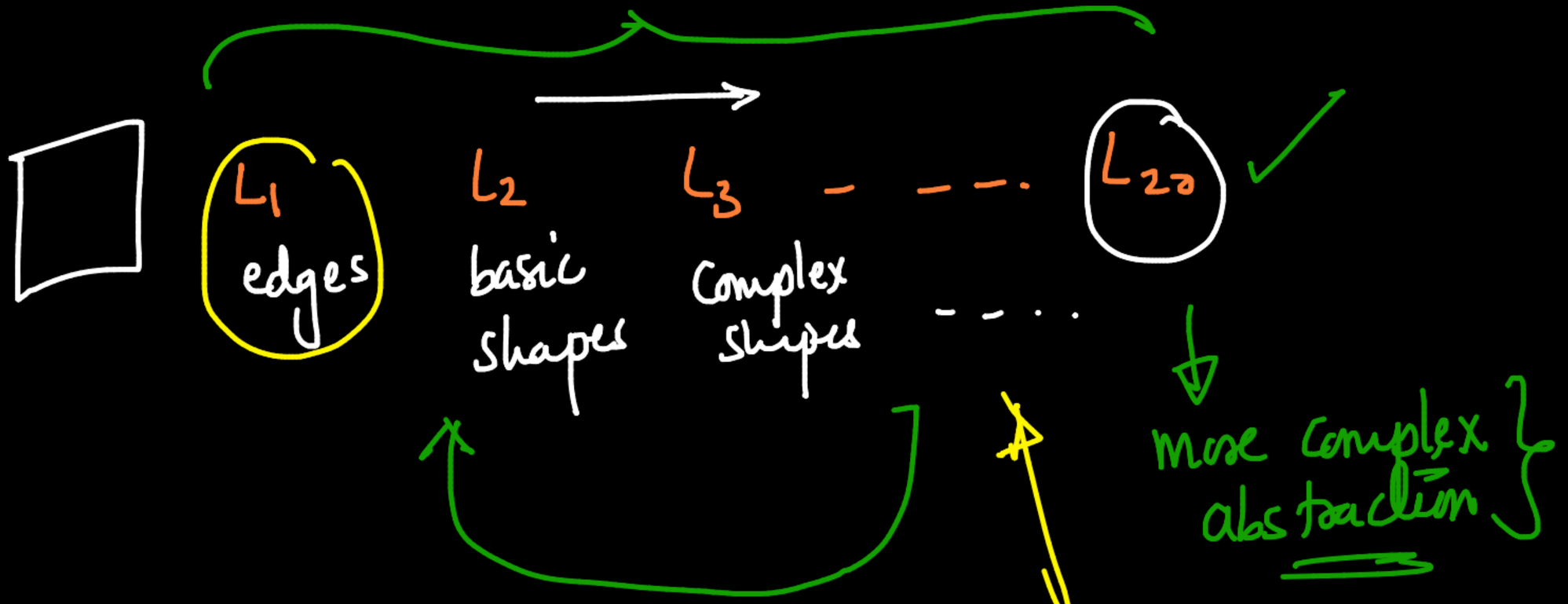
metric	mAP@.5	mAP@[.5, .95]
VGG-16	41.5	21.2
ResNet-101	**48.4**	**27.2**

Table 8. Object detection mAP (%) on the COCO validation set using **baseline** Faster R-CNN. See also Table 9 for better results.

networks such as FitNet [35] and Highway [42] (Table 6), yet is among the state-of-the-art results (6.43%, Table 6).

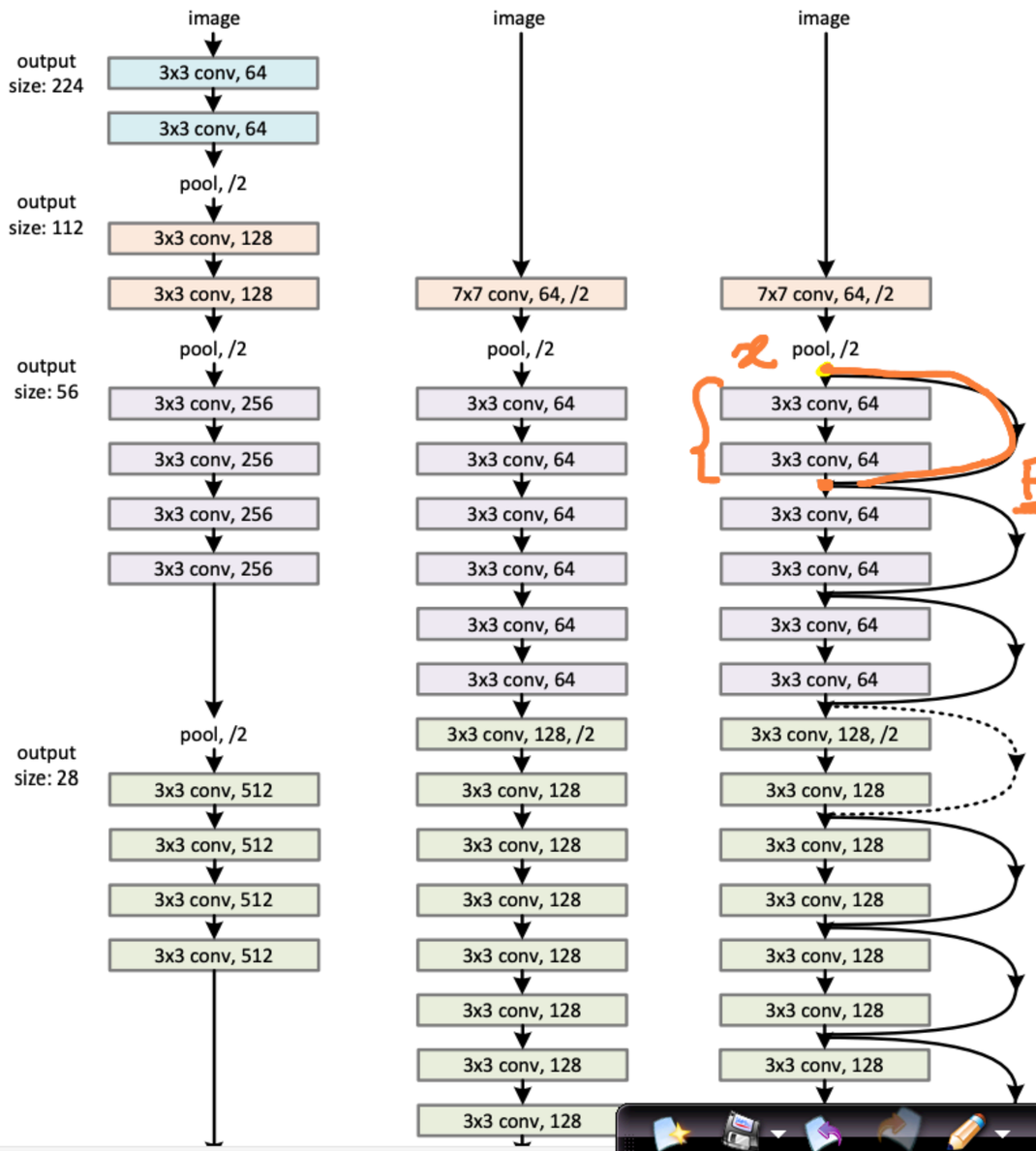
have similar training error. We argue that this is because of overfitting. The 1202-layer network may be unnecessarily large (19.4M) for this small dataset. Strong regularization [illegible]ropout [14] is applied to obtain the

Q



Malik:

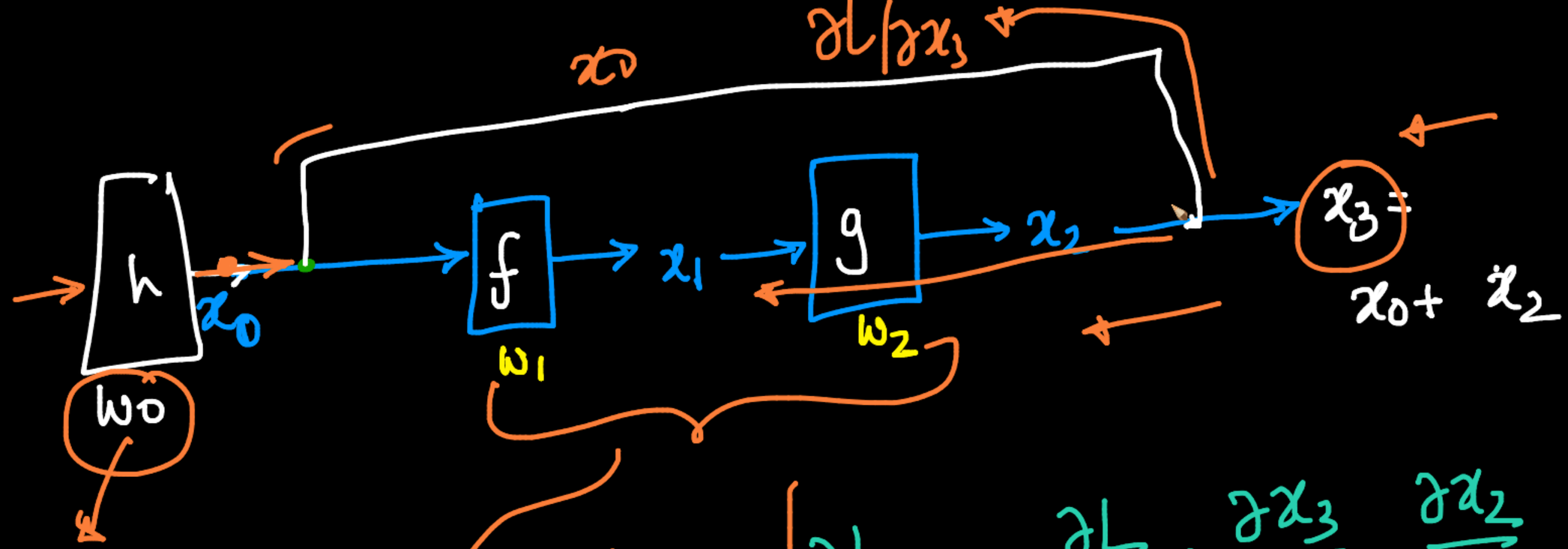
$f(g(x))$ vs $f(g(h(i(x))))$



insert shortcut connections (Fig. 3, right) which turn the network into its counterpart residual version. The identity shortcuts (Eqn.(1)) can be directly used when the input and output are of the same dimensions (solid line shortcuts in Fig. 3). When the dimensions increase (dotted line shortcuts in Fig. 3), we consider two options: (A) The shortcut still performs identity mapping, with extra zero entries padded for increasing dimensions. This option introduces no extra parameter; (B) The projection shortcut in Eqn.(2) is used to match dimensions (done by 1×1 convolutions). For both options, when the shortcuts go across feature maps of two sizes, they are performed with a stride of 2.

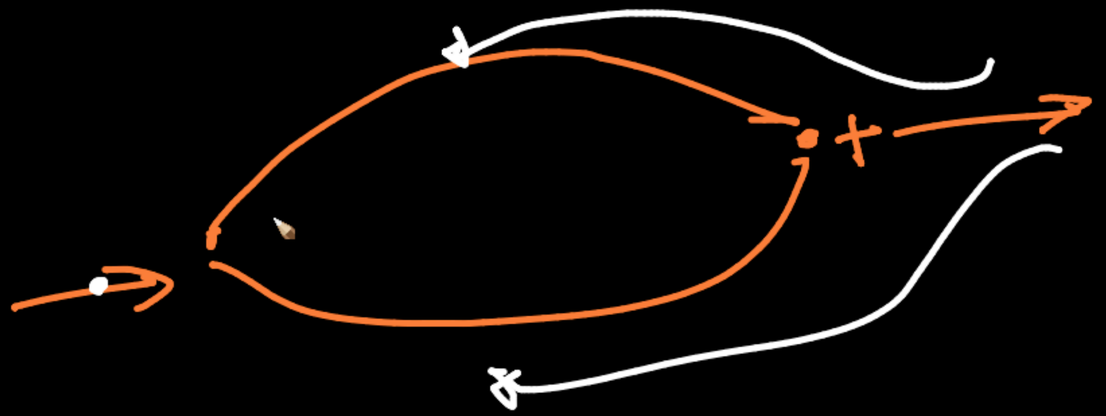
3.4. Implementation

Our implementation for ImageNet follows the practice in [21, 41]. The image is resized with its shorter side randomly sampled in [256, 480] for scale augmentation [41]. A 224×224 crop is randomly sampled from an image or its horizontal flip, with the per-pixel mean subtracted [21]. The standard color augmentation in [21] is used. We adopt batch normalization (BN) [16] right after each convolution and before activation, following [16]. We initialize the weights



$$\frac{\partial L}{\partial w_0} = \frac{\partial x_0}{\partial w_0} \cdot \left(1 + \boxed{\quad}\right) \frac{\partial L}{\partial x_3}$$

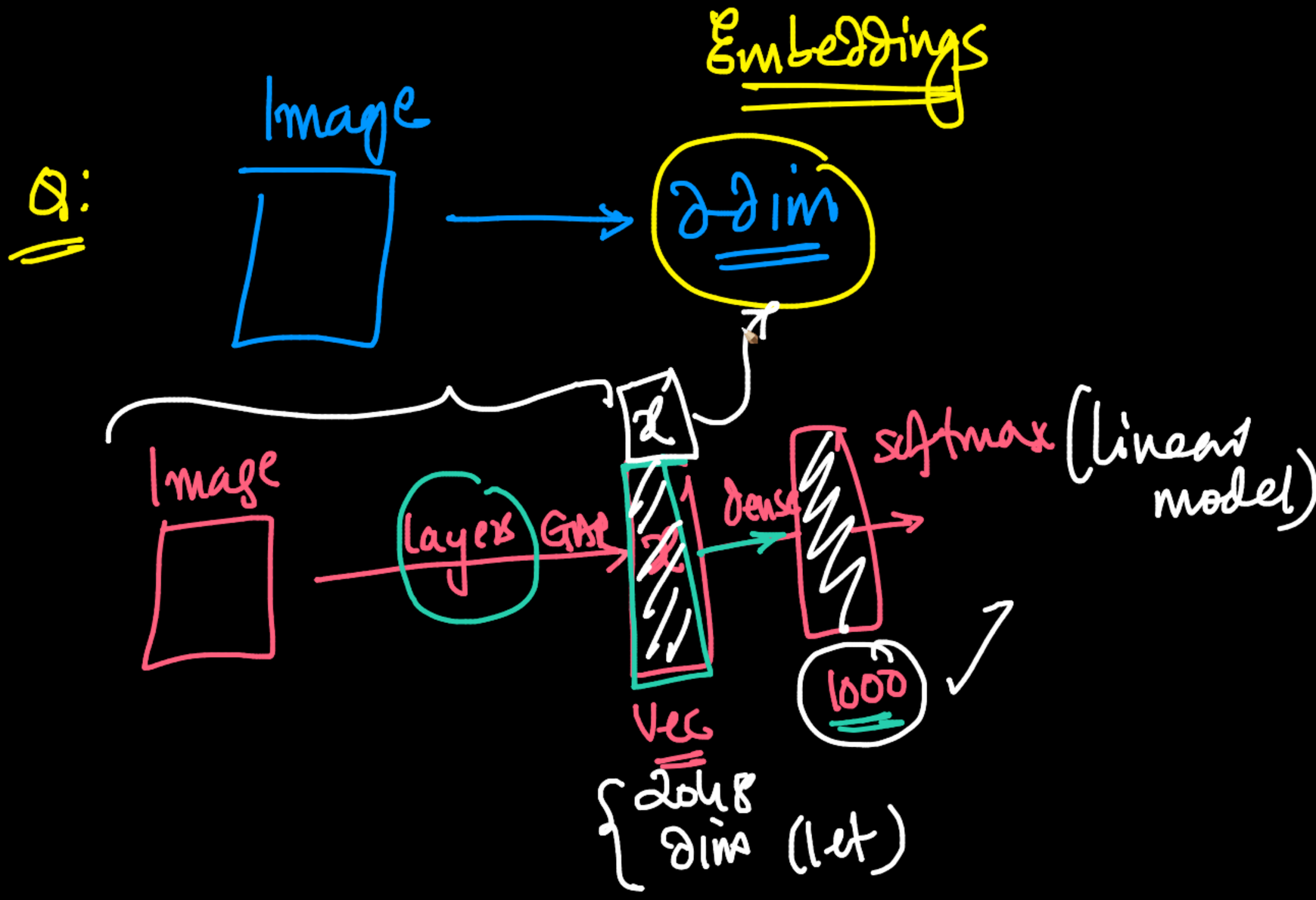
$$\frac{\partial L}{\partial w_2} = \frac{\partial L}{\partial x_3} \cdot \frac{\partial x_3}{\partial x_2} \frac{\partial x_2}{\partial w_2}$$



Q

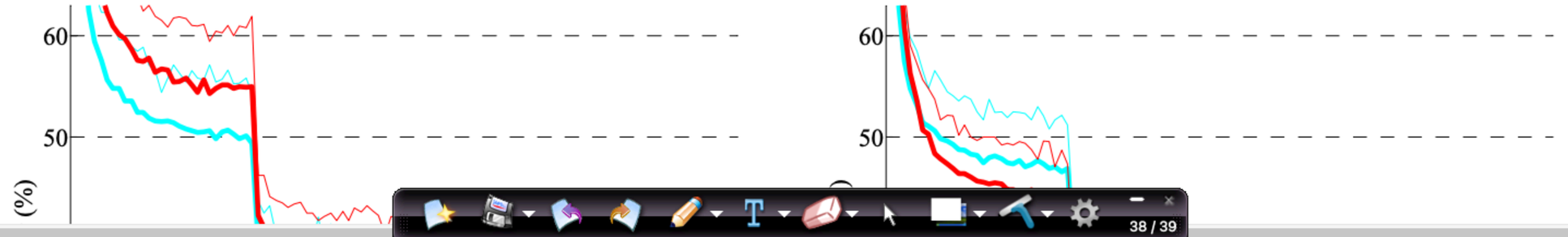
NN

$f(g(h(x)))$



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

Table 1. Architectures for ImageNet. Building blocks are shown in brackets (see also Fig. 5), with the numbers of blocks stacked. Down-sampling is performed by conv3_1, conv4_1, and conv5_1 with a stride of 2.



```
conv2_block2_out (Activation)  (None, 56, 56, 256)  0          ['conv2_block2_add[0][0]']

conv2_block3_1_conv (Conv2D)   (None, 56, 56, 64)   16448      ['conv2_block2_out[0][0]']

conv2_block3_1_bn (BatchNormal  (None, 56, 56, 64)  256        ['conv2_block3_1_conv[0][0]']
ization)

conv2_block3_1_relu (Activatio  (None, 56, 56, 64)  0          ['conv2_block3_1_bn[0][0]']
```

Feature Extraction :

Let's define a function to extract image features given an image and a model.

```
def extract_features(img_path, model):
    input_shape = (224, 224, 3)
    img = image.load_img(img_path,
                         target_size=(input_shape[0], input_shape[1]))
    img_array = image.img_to_array(img)
    expanded_img_array = np.expand_dims(img_array, axis=0)
    preprocessed_img = preprocess_input(expanded_img_array)
    features = model.predict(preprocessed_img)
    flattened_features = features.flatten()
    normalized_features = flattened_features / norm(flattened_features)
    return normalized_features
```

Im → Model → 2-dim

2048x1x1

Let's see the feature length the model generates by running on an example image. If you don't have the usual cat image available locally, let's download it!

```
8677
```

Now, let's run the extraction over the entire dataset and time it.

```
[ ] BATCH_SIZE = 128
    generator = tf.keras.utils.image_dataset_from_directory(root_dir,
                                                shuffle=False,
                                               batch_size=BATCH_SIZE,
                                                image_size=(224,224))

    num_images = len(generator.file_paths)
    num_epochs = int(math.ceil(num_images / BATCH_SIZE))

    start_time = time.time()
    feature_list = []
    feature_list = model.predict(generator, num_epochs)
    end_time = time.time()
```

batch → Model → 128
128
8-8 ms

```
Found 8677 files belonging to 101 classes.
```

```
[ ] for i, features in enumerate(feature_list):
        feature_list[i] = features / norm(features)

    feature_list = feature_list.reshape(len(feature_list), -1)

    print("Num images   = ", len(generator.file_paths))
    print("Shape of feature_list = ", feature_list.shape)
    print("Time taken in min = ", (end_time - start_time)/60)
```

```
                                        image_size=(224,224))

num_images = len(generator.file_paths)
num_epochs = int(math.ceil(num_images / BATCH_SIZE))

start_time = time.time()
feature_list = []
feature_list = model.predict(generator, num_epochs)
end_time = time.time()
```


```
Found 8677 files belonging to 101 classes.
```

```
for i, features in enumerate(feature_list):
    feature_list[i] = features / norm(features)

feature_list = feature_list.reshape(len(feature_list), -1)

print("Num images   = ", len(generator.file_paths))
print("Shape of feature_list = ", feature_list.shape)
print("Time taken in min = ", (end_time - start_time)/60)
```

```
Num images   =  8677
Shape of feature_list =  (8677, 2048)
Time taken in min =  16.577463599046073
```

```
[ ] # import matplotlib.pyplot as plt
    # class_names = generator.class_names
    # plt.figure(figsize=(10, 10))
    # for images, labels in generator.take(2):
    #   for i in range(9):
```

```
                                        image_size=(224,224))

num_images = len(generator.file_paths)
num_epochs = int(math.ceil(num_images / BATCH_SIZE))

start_time = time.time()
feature_list = []
feature_list = model.predict(generator, num_epochs)
end_time = time.time()
```

```
Found 8677 files belonging to 101 classes.
```

```
for i, features in enumerate(feature_list):
    feature_list[i] = features / norm(features)

feature_list = feature_list.reshape(len(feature_list), -1)

print("Num images   = ", len(generator.file_paths))
print("Shape of feature_list = ", feature_list.shape)
print("Time taken in min = ", (end_time - start_time)/60)
```

```
Num images   =  8677
Shape of feature_list =  (8677, 2048)
Time taken in min =  16.577463599046073
```

```
[ ] # import matplotlib.pyplot as plt
    # class_names = generator.class_names
    # plt.figure(figsize=(10, 10))
    # for images, labels in generator.take(2):
    #   for i in range(9):
```

Image → 2-2 im → Unit-vec

~9000

mist: ✓

 → KNN

Image; → $x_i \in$ 2048 dim vec

↓

k-NN (brute force)

Approx NN

```
for dimensions in pca_dimensions:
    pca = PCA(n_components=dimensions)
    pca.fit(feature_list)
    feature_list_compressed = pca.transform(feature_list[:])
    # Calculate accuracy over the compressed features
    accuracy, t = calculate_accuracy(feature_list_compressed[:])
    pca_time.append(t)
    pca_accuracy.append(accuracy)
    print("For PCA Dimensions = ", dimensions, ",\tAccuracy = ", accuracy, "%",
          ",\tTime = ", pca_time[-1])
```

```
For PCA Dimensions =  1 ,     Accuracy =  15.43 % ,   Time =  5.8796515464782715
For PCA Dimensions =  2 ,     Accuracy =  26.15 % ,   Time =  6.158180236816406
For PCA Dimensions =  3 ,     Accuracy =  32.4 % ,    Time =  7.299118995666504
For PCA Dimensions =  4 ,     Accuracy =  40.07 % ,   Time =  6.617396116256714
For PCA Dimensions =  5 ,     Accuracy =  45.93 % ,   Time =  6.806637763977051
For PCA Dimensions =  10 ,    Accuracy =  65.93 % ,   Time =  14.823916912078857
For PCA Dimensions =  20 ,    Accuracy =  77.81 % ,   Time =  13.174192667007446
For PCA Dimensions =  50 ,    Accuracy =  85.21 % ,   Time =  23.598203897476196
For PCA Dimensions =  75 ,    Accuracy =  86.38 % ,   Time =  40.408857345581055
For PCA Dimensions =  100 ,   Accuracy =  86.91 % ,   Time =  37.94637584686279
For PCA Dimensions =  150 ,   Accuracy =  87.03 % ,   Time =  51.85892105102539
For PCA Dimensions =  200 ,   Accuracy =  87.05 % ,   Time =  63.817588806152344
```

- As is visible stats, there is little improvement in accuracy after increasing beyond a feature-length of 150 dimensions.
- With almost 20 times fewer dimensions (150) than the original (2,048), this offers drastically higher speed and less time on almost any search algorithm, while achieving similar (and sometimes slightly better) accuracy.
- Hence, 150 would be an ideal feature-length for this dataset.
- This also means that the first 150 dimensions contain the most information about the dataset.

```
for dimensions in pca_dimensions:
    pca = PCA(n_components=dimensions)
    pca.fit(feature_list)
    feature_list_compressed = pca.transform(feature_list[:])
    # Calculate accuracy over the compressed features
    accuracy, t = calculate_accuracy(feature_list_compressed[:])
    pca_time.append(t)
    pca_accuracy.append(accuracy)
    print("For PCA Dimensions = ", dimensions, ",\tAccuracy = ", accuracy, "%",
          ",\tTime = ", pca_time[-1])
```

```
For PCA Dimensions =  1 ,      Accuracy =  15.43 % ,   Time =  5.8796515464782715
For PCA Dimensions =  2 ,      Accuracy =  26.15 % ,   Time =  6.158180236816406
For PCA Dimensions =  3 ,      Accuracy =  32.4 % ,    Time =  7.299118995666504
For PCA Dimensions =  4 ,      Accuracy =  40.07 % ,   Time =  6.617396116256714
For PCA Dimensions =  5 ,      Accuracy =  45.93 % ,   Time =  6.806637763977051
For PCA Dimensions =  10 ,     Accuracy =  63.93 % ,   Time =  14.823916912078857
For PCA Dimensions =  20 ,     Accuracy =  77.81 % ,   Time =  13.174192667007446
For PCA Dimensions =  50 ,     Accuracy =  85.21 % ,   Time =  23.598203897476196
For PCA Dimensions =  75 ,     Accuracy =  86.38 % ,   Time =  40.408857345581055
For PCA Dimensions =  100 ,    Accuracy =  86.91 % ,   Time =  37.94637584686279
For PCA Dimensions =  150 ,    Accuracy =  87.03 % ,   Time =  51.85892105102539
For PCA Dimensions =  200 ,    Accuracy =  87.05 % ,   Time =  63.817588806152344
```

- As is visible stats, there is little improvement in accuracy after increasing beyond a feature-length of 150 dimensions.
- With almost 20 times fewer dimensions (150) than the original (2,048), this offers drastically higher speed and less time on almost any search algorithm, while achieving similar (and sometimes slightly better) accuracy.
- Hence, 150 would be an ideal feature-length for this dataset.
- This also means that the first 150 dimensions contain the most information about the dataset.

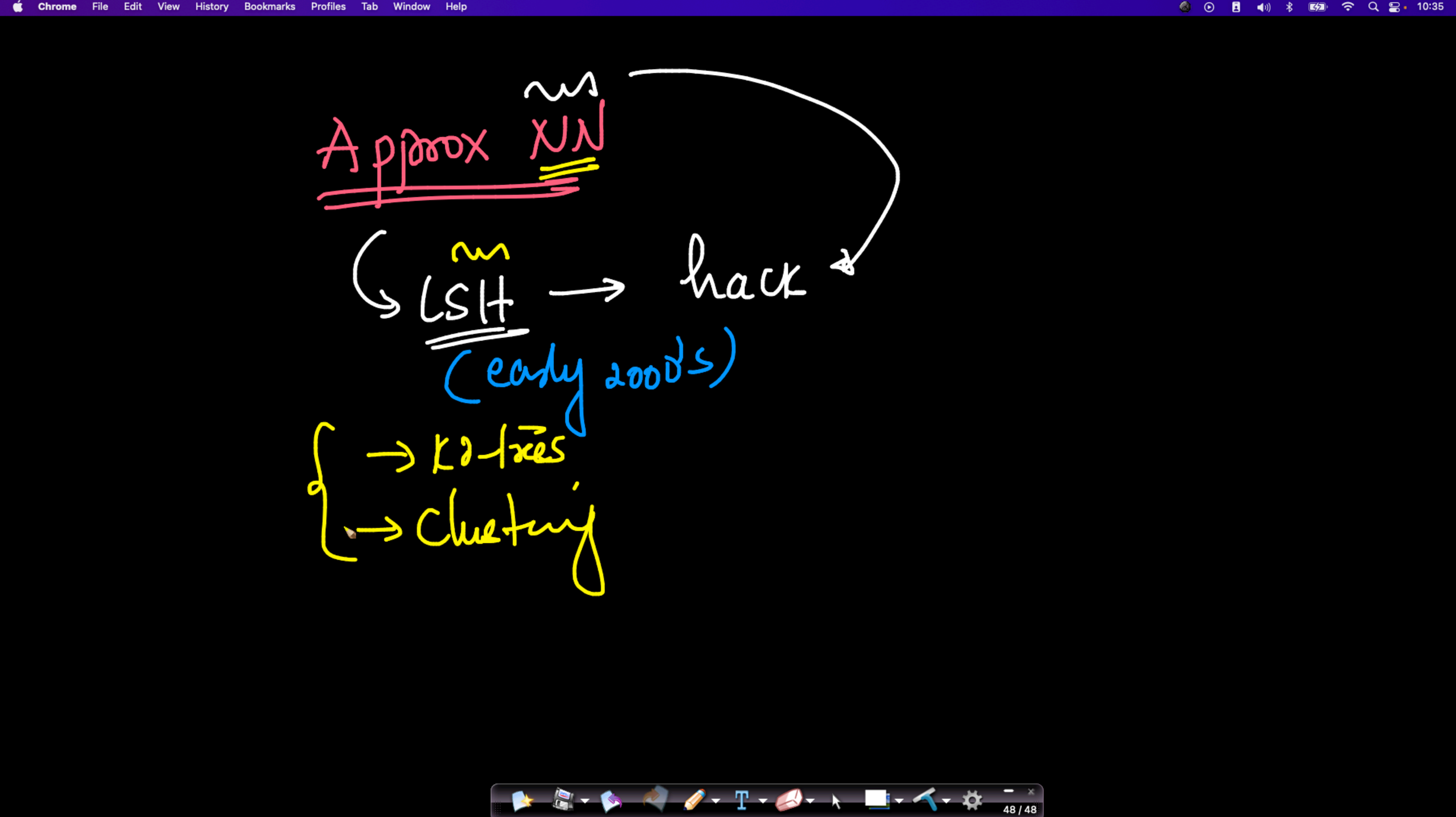
```
for dimensions in pca_dimensions:
    pca = PCA(n_components=dimensions)
    pca.fit(feature_list)
    feature_list_compressed = pca.transform(feature_list[:])
    # Calculate accuracy over the compressed features
    accuracy, t = calculate_accuracy(feature_list_compressed[:])
    pca_time.append(t)
    pca_accuracy.append(accuracy)
    print("For PCA Dimensions = ", dimensions, ",\tAccuracy = ", accuracy, "%",
          ",\tTime = ", pca_time[-1])
```

```
For PCA Dimensions =  1 ,     Accuracy =  15.43 % ,   Time =  5.8796515464782715
For PCA Dimensions =  2 ,     Accuracy =  26.15 % ,   Time =  6.158180236816406
For PCA Dimensions =  3 ,     Accuracy =  32.4 % ,    Time =  7.299118995666504
For PCA Dimensions =  4 ,     Accuracy =  40.07 % ,   Time =  6.617396116256714
For PCA Dimensions =  5 ,     Accuracy =  45.93 % ,   Time =  6.806637763977051
For PCA Dimensions =  10 ,    Accuracy =  63.93 % ,   Time =  14.823916912078857
For PCA Dimensions =  20 ,    Accuracy =  77.81 % ,   Time =  13.174192667007446
For PCA Dimensions =  50 ,    Accuracy =  85.21 % ,   Time =  23.598203897476196
For PCA Dimensions =  75 ,    Accuracy =  86.38 % ,   Time =  40.408857345581055
For PCA Dimensions =  100 ,   Accuracy =  86.91 % ,   Time =  37.94637584686279
For PCA Dimensions =  150 ,   Accuracy =  87.03 % ,   Time =  51.85892105102539
For PCA Dimensions =  200 ,   Accuracy =  87.05 % ,   Time =  63.817588806152344
```

→ Brute for KNN

→ PCA + brute force KNN

- As is visible stats, there is little improvement in accuracy after increasing beyond a feature-length of 150 dimensions.
- With almost 20 times fewer dimensions (150) than the original (2,048), this offers drastically higher speed and less time on almost any search algorithm, while achieving similar (and sometimes slightly better) accuracy.
- Hence, 150 would be an ideal feature-length for this dataset.
- This also means that the first 150 dimensions contain the most information about the dataset.


Approx NN
LSH → hack
(early 2000's)
→ k-d trees
→ clustering

PCA + Annoy:

- Annoy (Approximate Nearest Neighbors Oh Yeah) is a C++ library with Python bindings for searching nearest neighbors.
- Synonymous with speed, it was released by Spotify and is used in production to serve their music recommendations.
- To use annoy, install it using pip: pip3 install annoy

FAISS: FB

SOTA: SCANN (Google)

```
[ ] !pip3 install Annoy
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Collecting Annoy
  Downloading annoy-1.17.0.tar.gz (646 kB)
     |████████████████████████████████| 646 kB 28.9 MB/s
Building wheels for collected packages: Annoy
  Building wheel for Annoy (setup.py) ... done
  Created wheel for Annoy: filename=annoy-1.17.0-cp37-cp37m-linux_x86_64.whl size=394608 sha256=847ebda234dcd2bdd6c73398026399a0a412dc4583
  Stored in directory: /root/.cache/pip/wheels/4f/e8/1e/7cc9ebbfa87a3b9f8ba79408d4d31831d67eea918b679a4c07
Successfully built Annoy
Installing collected packages: Annoy
Successfully installed Annoy-1.17.0
```

```
[ ] from annoy import AnnoyIndex
```

First, we build a search index with two hyperparameters - the number of dimensions of the dataset, and the number of trees.

```
[ ] # Time the indexing for Annoy
```

```
        t.add_item(i, feature)
    endtime = time.time()
    print(endtime - starttime)
    t.build(40)  # 40 trees
    t.save('data/caltech101index.ann')
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:2: FutureWarning: The default argument for metric will be removed in future v

0.3510735034942627
True
```

Annoy on one image: Now let's find out the time it takes to search the 5 nearest neighbors of one image.

```
[ ] random_image_index = 1001
```

```
[ ] u = AnnoyIndex(150)
    %timeit u.get_nns_by_vector(feature_list_compressed[random_image_index], 5, include_distances=True)
    indexes = u.get_nns_by_vector(feature_list_compressed[random_image_index],
                                  5,
                                  include_distances=True)
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:1: FutureWarning: The default argument for metric will be removed in future v
  """Entry point for launching an IPython kernel.
The slowest run took 100.68 times longer than the fastest. This could mean that an intermediate result is being cached.
100000 loops, best of 5: 19.9 µs per loop
```

- Now **THAT is blazing fast!** To put this in perspective, for such a modestly sized dataset, this can serve almost 15000 requests on a single

Conclusion:

Feature extraction using RESNET50 on Caltech101 dataset:

- Bruteforce + 2048 length feature 59.8ms
- Bruteforce + PCA(150) 5.3ms
- Annoy + PCA(150) 19us

Exercise:

Try to extract features from other pretrained models such as Inception net, Vgg19 etc and compare the time and accuracy of your models.

L3 : CNN Under the hood.ipynb

Agenda

- In this lecture we will explain how forward & backward propagation works in CNN from scratch using numpy and scipy(for mathematical operations) to classify Images from the MNIST dataset.

About MNIST dataset:

- MNIST is a dataset consisting of 70,000 black-and-white images of handwritten digits.
- Each image is 28 x 28 (= 784) pixels.
- Total of 10 classes (numbers from 0 to 9)
- Each pixel is encoded as an integer from 1 (white) to 255 (black):
  - the higher this value the darker the color.
  - 60,000 images are used as train set and 10,000 as test set.