Transfer Learning

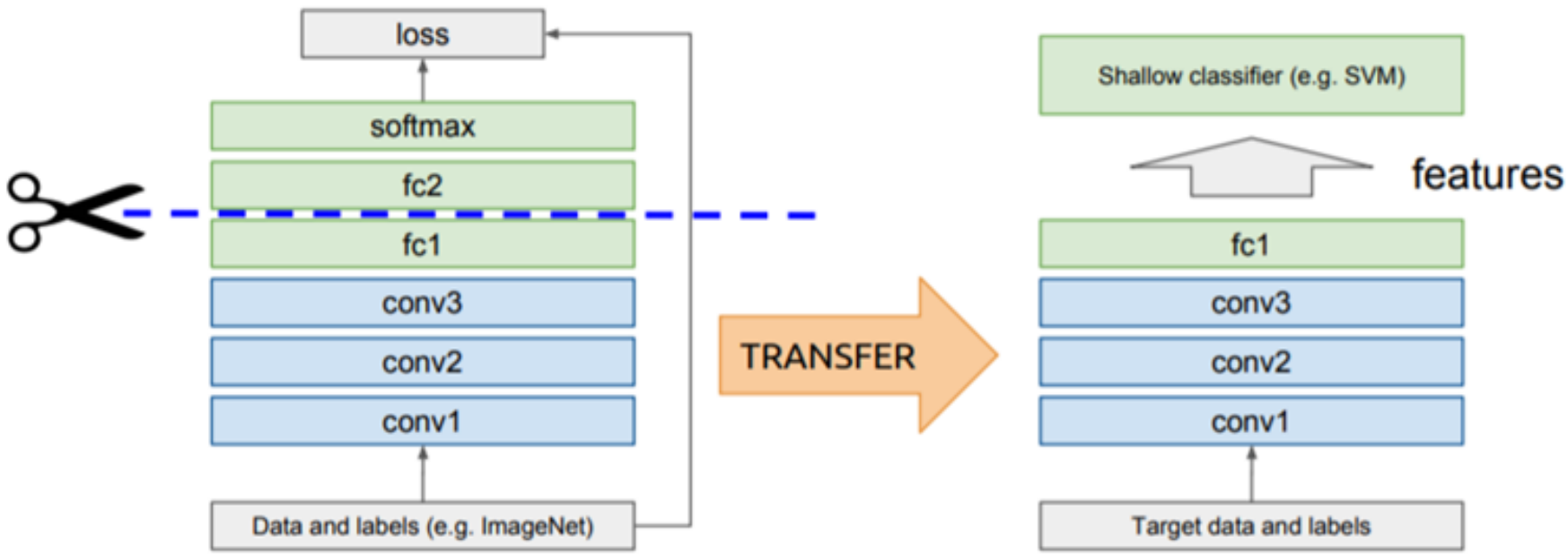
Transfer Learning

- Transfer Learning is a technique in which a model that is trained on a certain task can be used to make predictions for another task based on its learning (weights).

Transfer Learning

- The key idea here is to just leverage the pre-trained model's weighted layers to extract features but not to update the weights of the model's layers during training with new data for the new task.

Idea: use outputs of one or more layers of a network trained on a different task as generic feature detectors. Train a new shallow model on these features.



Fine Tuning

- This is a more involved technique, where we do not just replace the final layer (for classification/regression), but we also selectively retrain some of the previous layers.

Freeze or fine-tune?

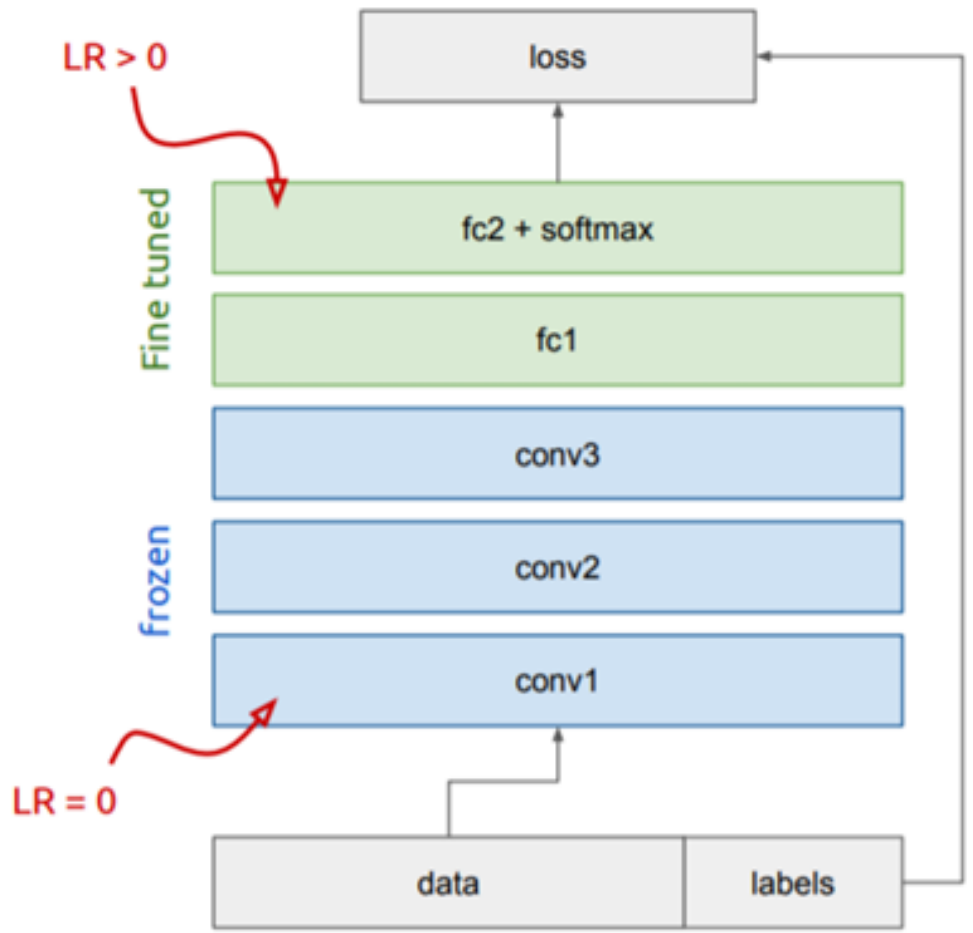
Bottom *n* layers can be frozen or fine tuned.

- **Frozen**: not updated during backprop
- **Fine-tuned**: updated during backprop

Which to do depends on target task:

- **Freeze**: target task labels are scarce, and we want to avoid overfitting
- **Fine-tune**: target task labels are more plentiful

In general, we can set learning rates to be different for each layer to find a tradeoff between freezing and fine tuning



Normalization

- Normalization has become an important part of deep neural networks that compensates for the unbounded nature of certain activation functions such as ReLU, ELU etc.
- With these activation functions, the output layer is not constrained within a bounded range (such as [-1,1] for $tanh$), rather they can grow as high as the training allows it.
- To limit the unbounded activation from increasing the output layer values, normalization is used just before the activation function.

Local Response Normalization (LRN)

- Was first introduced in AlexNet architecture where the activation function used was *ReLU* as opposed to the more common $tanh$ and $sigmoid$ at that time.

- LRN is a **non-trainable layer** that square-normalizes the pixel values in a feature map within a local neighbourhood.

Local Response Normalization (LRN)

- This layer is useful when we are dealing with ReLU neurons. Why?

- Because ReLU neurons have unbounded activations and we need LRN to normalize that. We want to detect high frequency features with a large response. If we normalize around the local neighborhood of the excited neuron, it becomes even more sensitive as compared to its neighbors.

Batch Normalization (BN)

- A technique that normalizes the mean and variance of each of the features at every level of representation during training.
- The technique involves normalization of the features across the examples in each mini-batch.
- Empirically it appears to stabilize the gradient (less exploding or vanishing values) and batch-normalized models appear to overfit less.
- In fact, batch-normalized models seldom even use dropout.

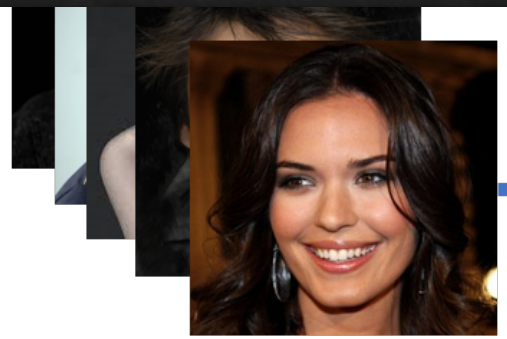
Batch Normalization (BN)

- Batch normalization is applied to individual layers (optionally, to all of them) and works as follows: In each training iteration, for each layer, we first compute its activations as usual. Then, we normalize the activations of each node by subtracting its mean and dividing by its standard deviation estimating both quantities based on the statistics of the current minibatch.

Generative Adversarial Network (GAN)

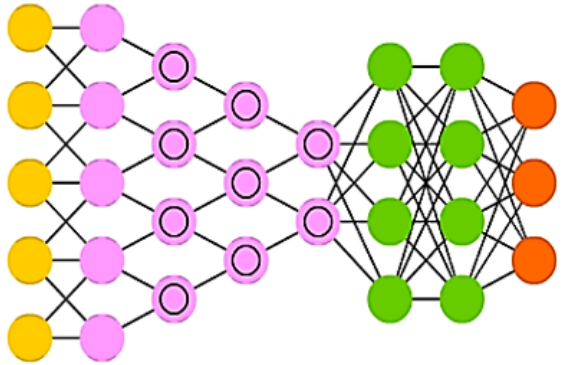
IDEA

- GAN attempts to *combine the discriminatory model and the generative model* by randomly generating the data through the generative model, then letting the discriminative model evaluate the data and use the result to improve the next output.



Discriminator

Deep Convolutional Network (DCN)

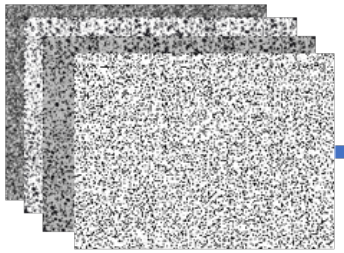


Fake

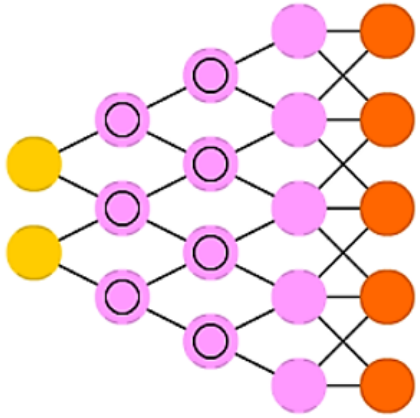
Real

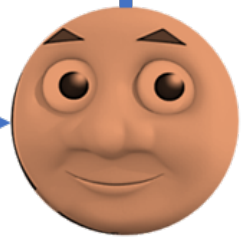
Generator

Random noise



Deconvolutional Network (DN)





Generated faces

INPUT

- In its most basic form, a GAN takes random noise as its input. The generator then transforms this noise into a meaningful output. By introducing noise, we can get the GAN to produce a wide variety of data, sampling from different places in the target distribution.
- Experiments suggest that the distribution of the noise doesn't matter much, so we can choose something that's easy to sample from, like a uniform distribution. For convenience the space from which the noise is sampled is usually of smaller dimension than the dimensionality of the output space

TRAINING

- Training a GAN has two parts:
- **Part 1:** The Discriminator is trained while the Generator is idle. In this phase, the network is only forward propagated and no back-propagation is done. The Discriminator is trained on real data for n epochs, and see if it can correctly predict them as real. Also, in this phase, the Discriminator is also trained on the fake generated data from the Generator and see if it can correctly predict them as fake.

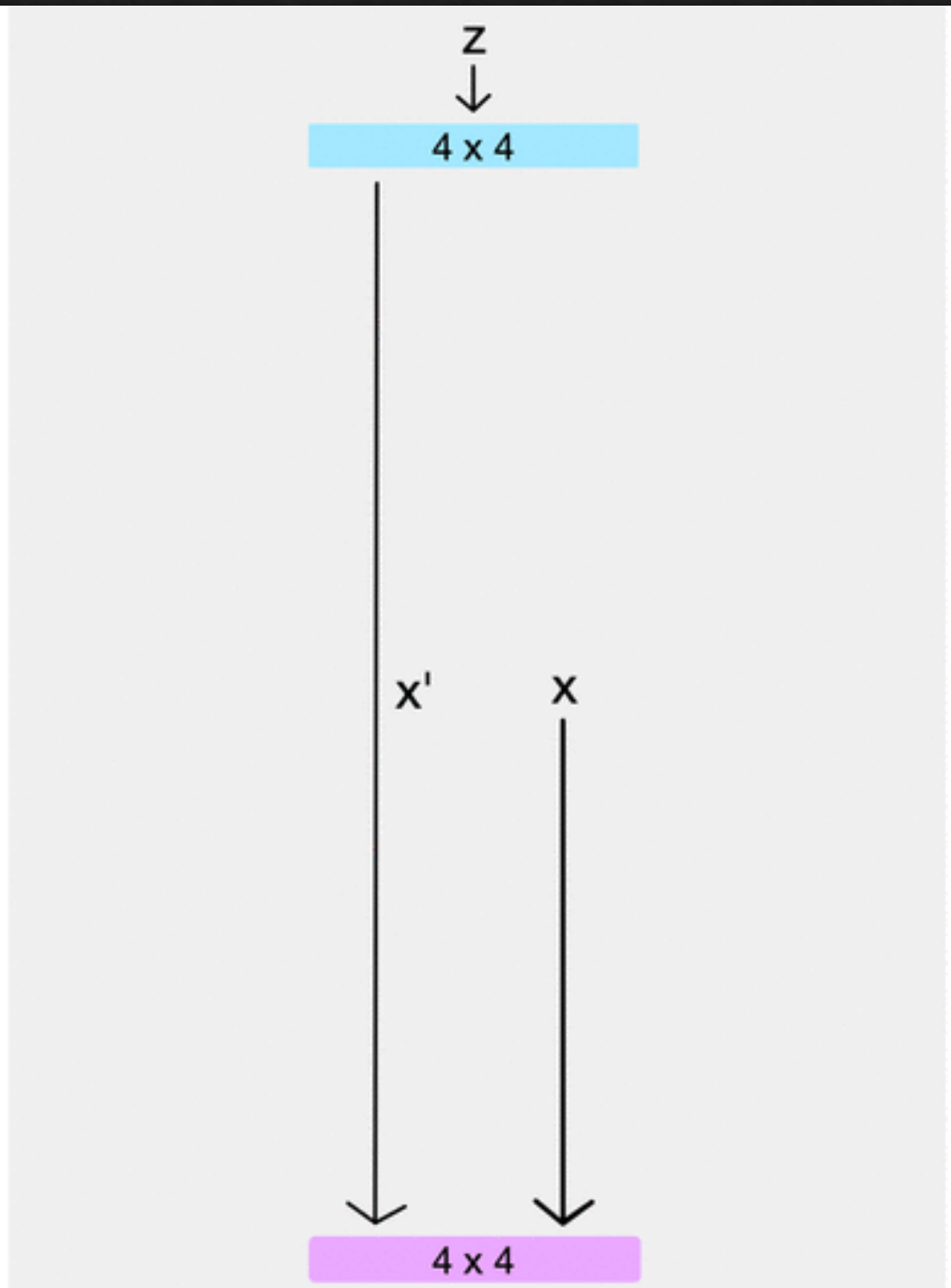
TRAINING

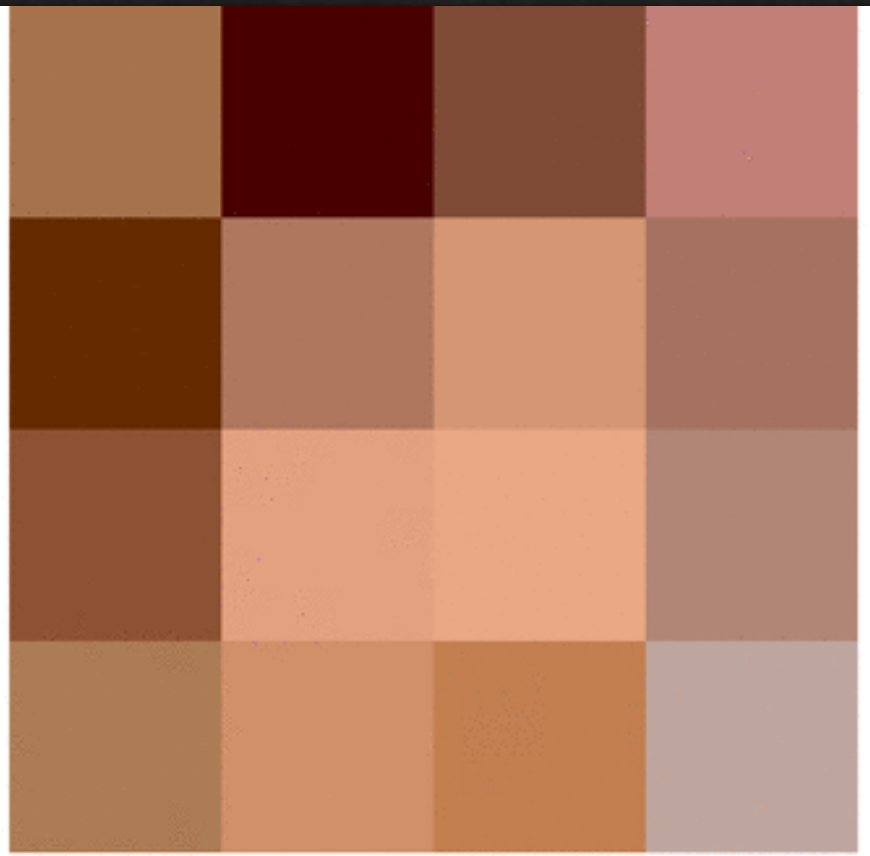
- **Part 2:** The Generator is trained while the Discriminator is idle. After the Discriminator is trained by the generated fake data of the Generator, we can get its predictions and use the results for training the Generator and get better from the previous state to try and fool the Discriminator.

TRAINING

We train the generator with the following procedure:

1. Sample random noise.
2. Produce generator output from sampled random noise.
3. Get discriminator "Real" or "Fake" classification for generator output.
4. Calculate loss from discriminator classification.
5. Backpropagate through both the discriminator and generator to obtain gradients.
6. Use gradients to change only the generator weights.





Training time: 0 days

4x4 resolution

Generator

Discriminator

z = random code

x = real image

x' = generated image

INCEPTION

INCEPTION MODULES

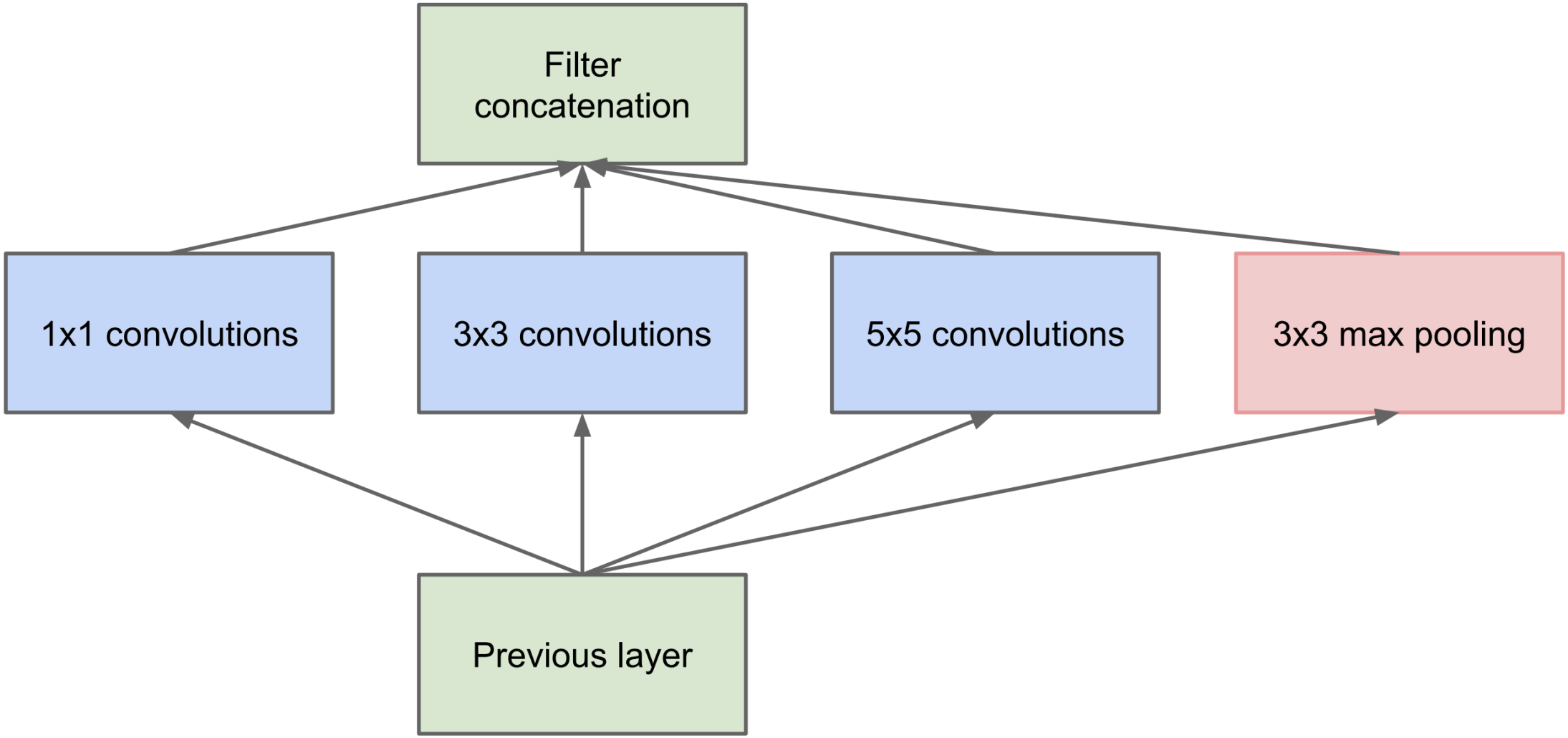
- Inception Modules are used in Convolutional Neural Networks to allow for more efficient computation and deeper Networks through a **dimensionality reduction**.

- The modules were designed to solve the problem of computational expense, as well as overfitting, among other issues. The solution, in short, is to take multiple kernel filter sizes within the CNN, and rather than stacking them sequentially, ordering them to operate on the same level.

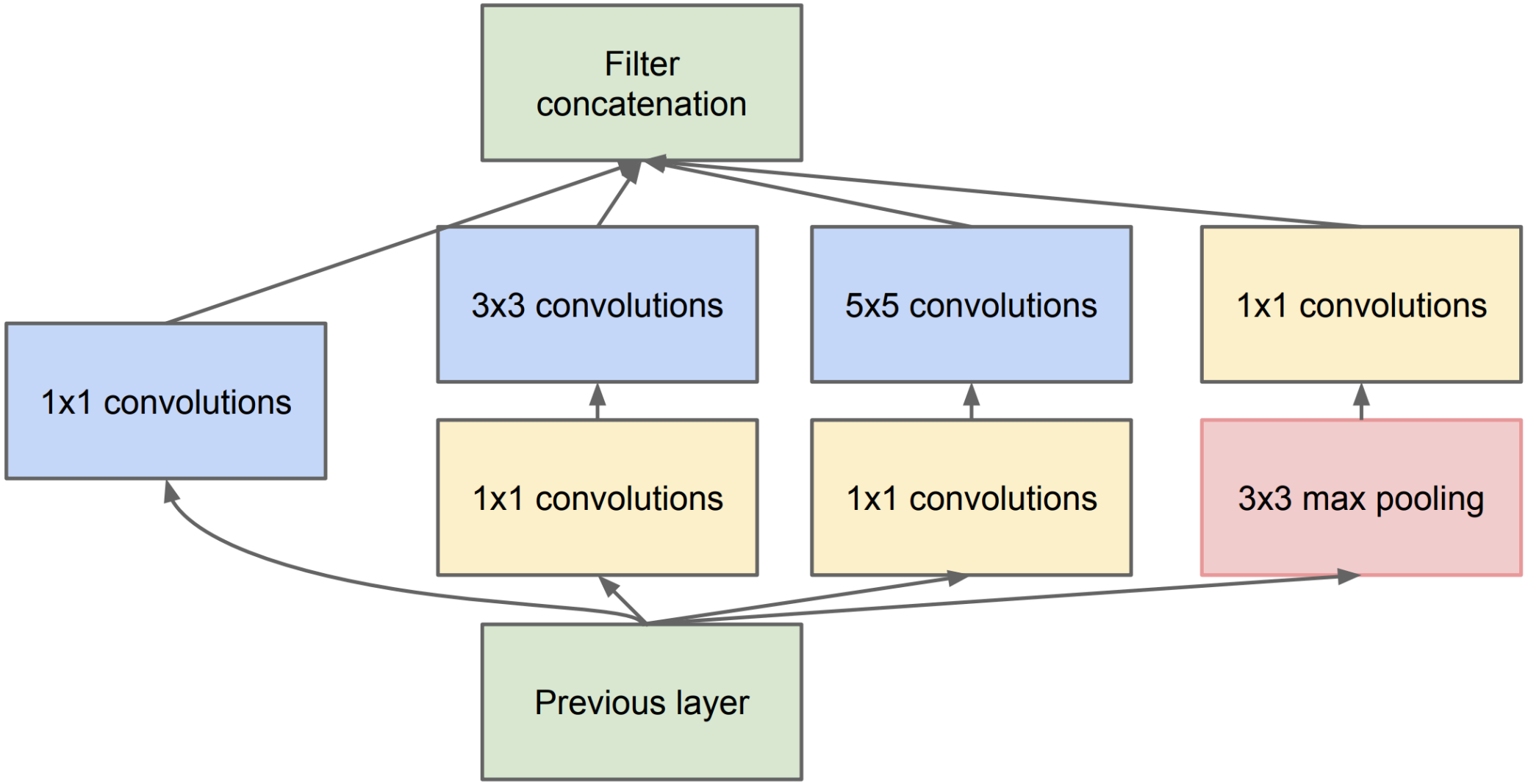
HOW IT WORKS?

- The most simplified version of an inception module works by performing a convolution on an input with NOT one, but THREE different sizes of filters (1x1, 3x3, 5x5). Also, max pooling is performed. Then, the resulting outputs are concatenated and sent to the next layer.

- By structuring the CNN to perform its convolutions on the same level, the network gets progressively wider, not deeper.

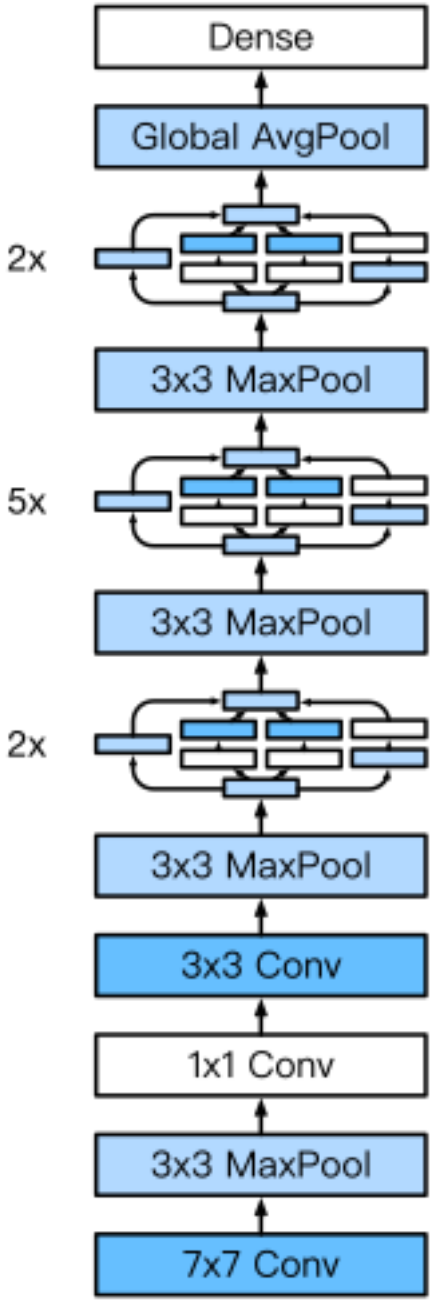


(a) Inception module, naïve version



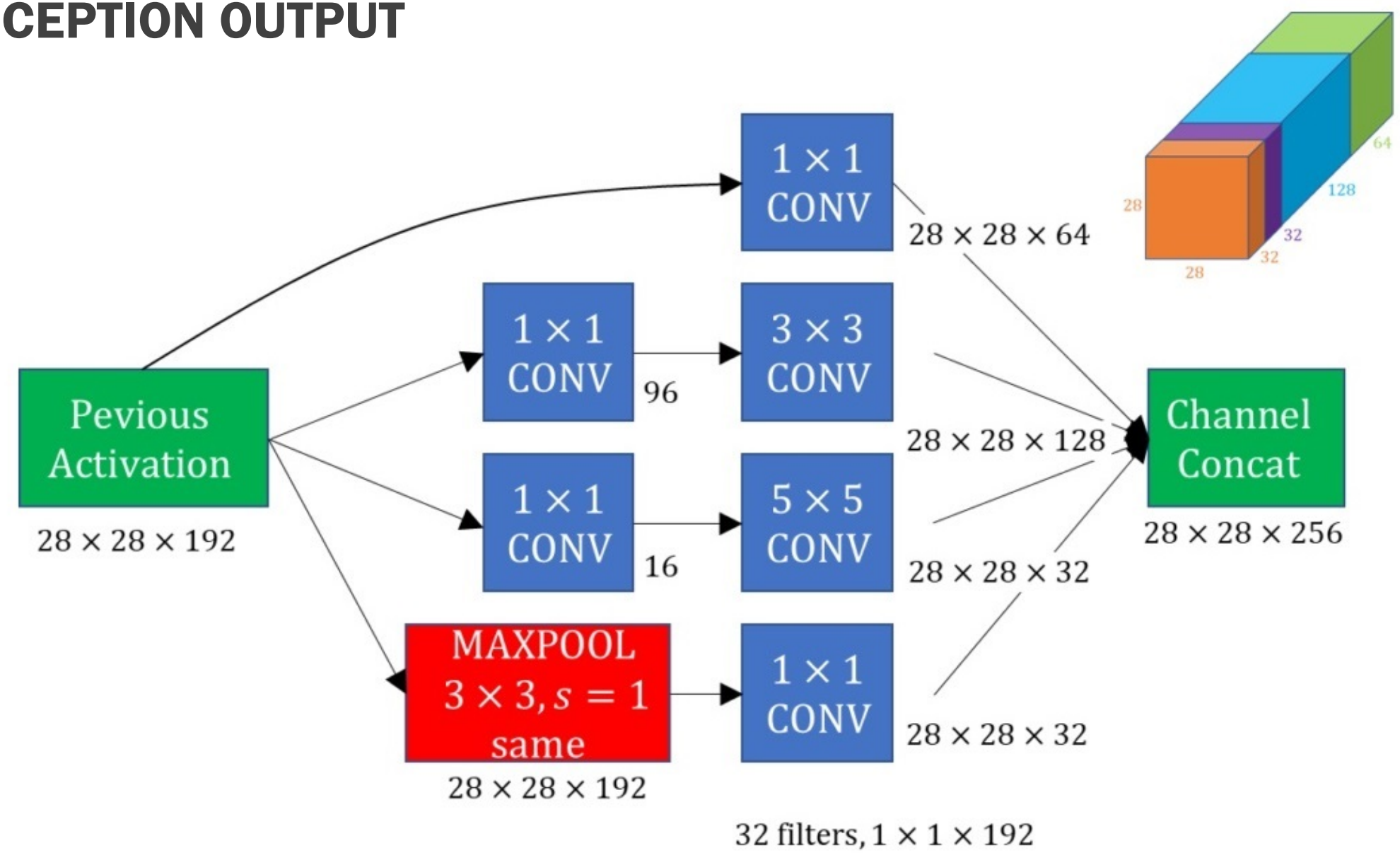
(b) Inception module with dimension reductions

GOOGLENET



- GoogLeNet uses a stack of a total of **9 inception** blocks and global average pooling to generate its estimates. Maximum pooling between inception blocks reduced the dimensionality.

INCEPTION OUTPUT



DOES INCEPTION EVEN COMPLETE TRAINING AND START PREDICTING?

1X1 CONVOLUTIONS COMES TO THE RESCUE!

1X1 CONVOLUTION

- A 1x1 convolution simply maps an input pixel with all it's channels to an output pixel, not looking at anything around itself. It is often used to reduce the number of depth channels.

  - input (256 depth) -> 1x1 convolution (64 depth) -> 4x4 convolution (256 depth)
  - input (256 depth) -> 4x4 convolution (256 depth)

  The bottom one is about ~3.7x slower

EXAMPLE

- Input ($28 \times 28 \times 192$)
- Let's apply 32, 5×5 filters

How many multiplication in required?

$$28 \times 28 \times 192 \times 5 \times 5 \times 32 \; = \sim 120 \; Million!$$

EXAMPLE

- Input ($28\times28\times192$)
- **First, let's apply 16, 1×1 filters**
- Then, apply 32, 5×5 filters

How many multiplication in required?

$$28\times28\times192\times1\times1\times16 = \sim2.4\ Million$$

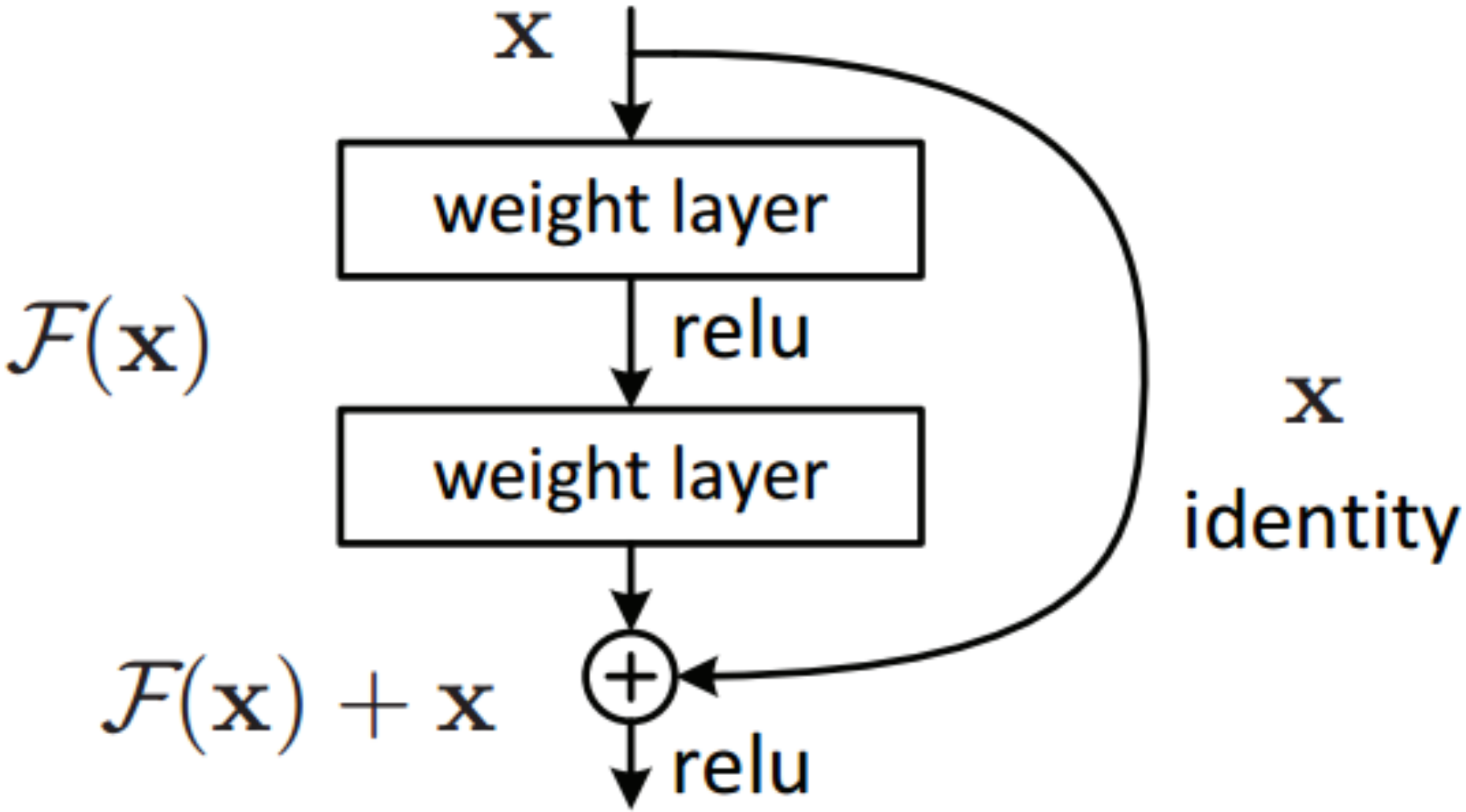
$$28\times28\times16\times5\times5\times32 = \sim10\ Million$$

$$Total = \sim12\ Million$$

- The $28\times28\times16$ intermediate output is what is called **Bottle-neck** layer.

RESIDUAL NETWORKS (RESNET)

- Deep Residual Network is almost similar to the networks which have convolution, pooling, activation and fully-connected layers stacked one over the other. The only construction to the simple network to make it a residual network is the ***identity connection*** between the layers.


x
weight layer
relu
weight layer
$\mathcal{F}(\mathbf{x})$
x
identity
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
relu

RESNET IDEA

- Input is x
- In a convolutional neural network we try to learn the output $(H(x))$
- In a residual network we try to learn the residual which is $(R(x) = H(x) - x)$

STANDARD 2-LAYER MODULE

```
def Unit(x,filters):

out = BatchNormalization()(x)
out = Activation("relu")(out)
out = Conv2D(filters=filters, kernel_size=[3, 3], strides=[1, 1], padding="same")(out)

out = BatchNormalization()(out)
out = Activation("relu")(out)
out = Conv2D(filters=filters, kernel_size=[3, 3], strides=[1, 1], padding="same")(out)

return out
```

RESNET MODULE

```
def Unit(x,filters):


res = x
out = BatchNormalization()(x)
out = Activation("relu")(out)
out = Conv2D(filters=filters, kernel_size=[3, 3], strides=[1, 1], padding="same")(out)

out = BatchNormalization()(out)
out = Activation("relu")(out)
out = Conv2D(filters=filters, kernel_size=[3, 3], strides=[1, 1], padding="same")(out)

out = keras.layers.add([res,out])

return out
```