Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

Advances in Neural Network Architectures

François Pitié

Assistant Professor in Media Signal Processing
Department of Electronic & Electrical Engineering, Trinity College Dublin

[4C16/5C16] Deep Learning and its Applications — 2022/2023

In this handout are covered some of the important advances in network architectures between 2012 and 2015.

These advances try to address some of the major difficulties in DNN's, including the problem of vanishing gradients when training deeper networks.

Transfer Learning

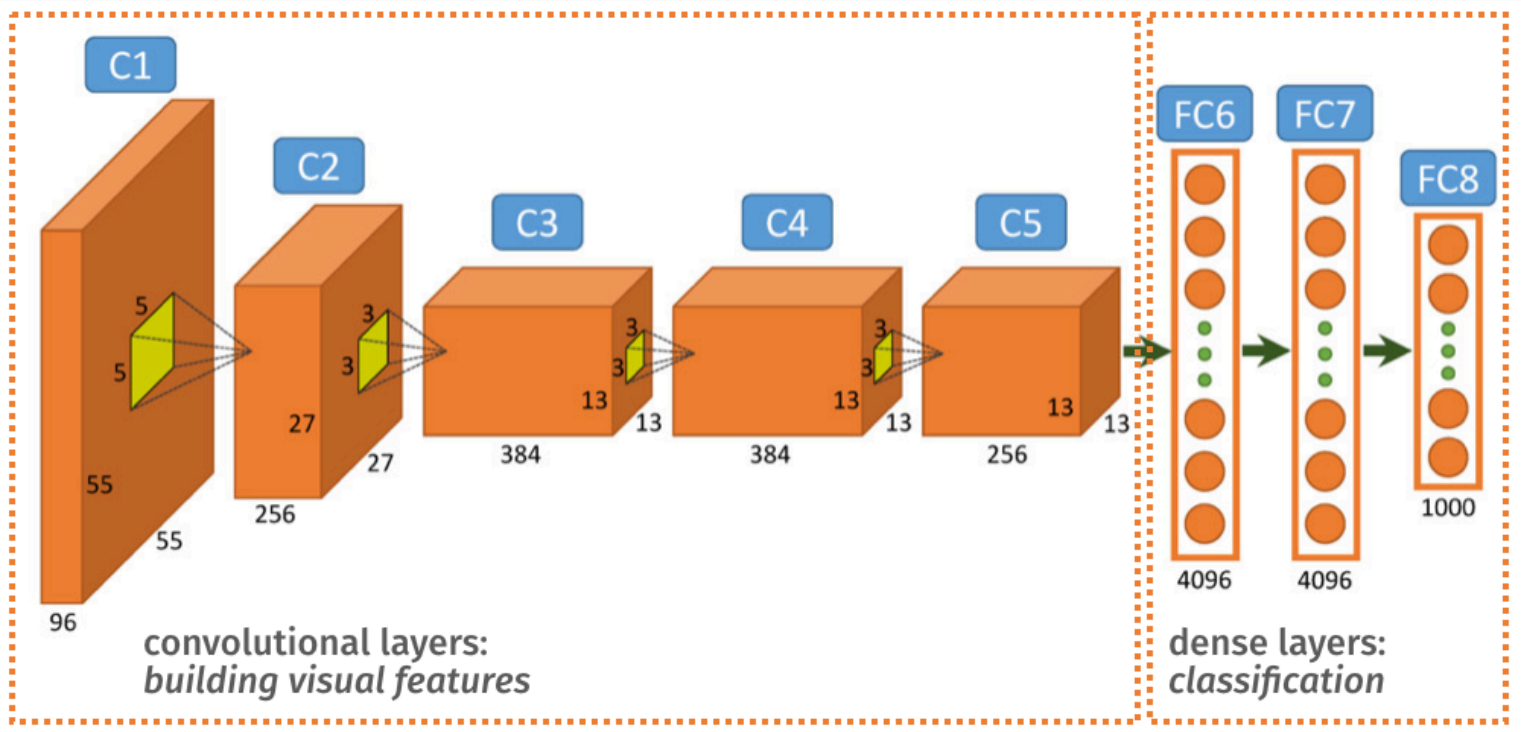
Reusing off-the-shelf networks

Say you are asked to develop a DNN application that can recognise pelicans on images.

Training a state of the art CNN network from scratch can necessitate weeks of training and hundreds of thousands of pictures. This would be unpractical in your case because you can only source a thousand images.

What can you do? You can reuse parts of existing networks.

Recall the architecture of AlexNet (2012):



Broadly speaking the convolutional layers (up to C5) build visual features whilst the last dense layers (FC6, FC7 and FC8) perform classification based on these visual features.

AlexNet (and any of the popular off-the-shelf networks such as VGG, ResNet or GoogLeNet) was trained on millions of images and thousands of classes. The network is thus able to deal with a great variety of problems and the trained filters produce very generic features that are relevant to most visual applications.

Therefore AlexNet's visual features could be very effective for your particular task and maybe there is no need to train new visual features: just reuse these existing ones.

The only task left is to design and train the classification part of the network (eg. the dense layers).

Your application looks like this: copy/paste a pre-trained network, cut away the last few layers and replace them with your own specialised network.

Depending on the amount of training data available to you, you may decide to only redesign the last layer (*ie.* FC8), or a handful of layers (*eg.* C5, FC6, FC7, FC8). Keep in mind that redesigning more layers will necessitate more training data.

If you have enough samples, you might want to allow backpropagation to update some of the imported layers, so as to fine tune the features for your specific application. If you don't have enough data, you probably should freeze the values of the imported weights.

In Keras you can freeze the update of parameters using the `trainable=False` argument. For instance:

```
currLayer = Dense(32, trainable=False)(prevLayer)
```

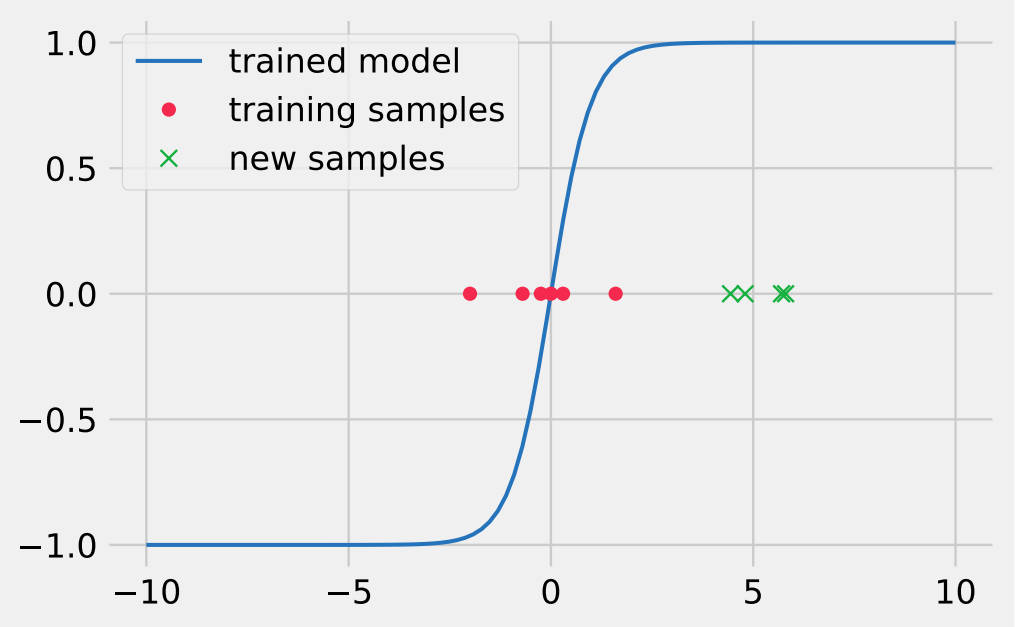
In most image based applications you should first consider reusing off-the-shelf networks such as VGG, GoogLeNet or ResNet. It has been shown (see link below) that using such generic visual features yield state of the art performances in most applications.

Razavian et al. “CNN Features off-the-shelf: an Astounding Baseline for Recognition”. 2014. [https://arxiv.org/abs/1403.6382]

Transfer learning and domain adaption

Transfer learning and vanishing gradients

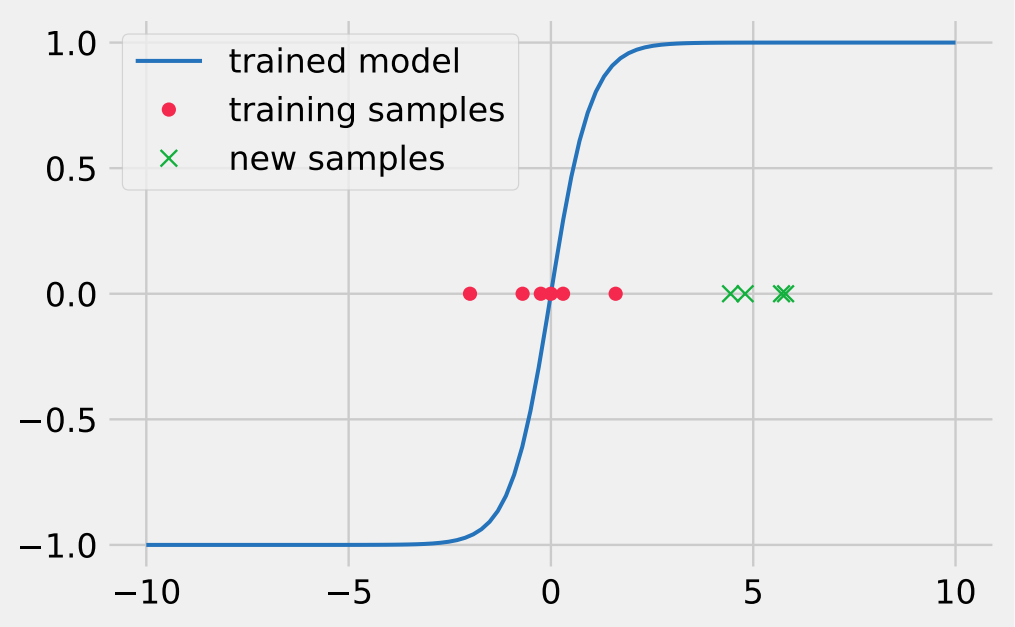
Let's see why re-using networks on new training sets can be difficult. Consider a single neuron and assume a $\tanh$ activation function:



$f(x_i, w) = \tanh(x_i + w)$. The training samples are images taken on a sunny day. The input values x_i (red dots) are centred about 0 and the estimated w is $\hat{w} = 0$.

Transfer learning and vanishing gradients

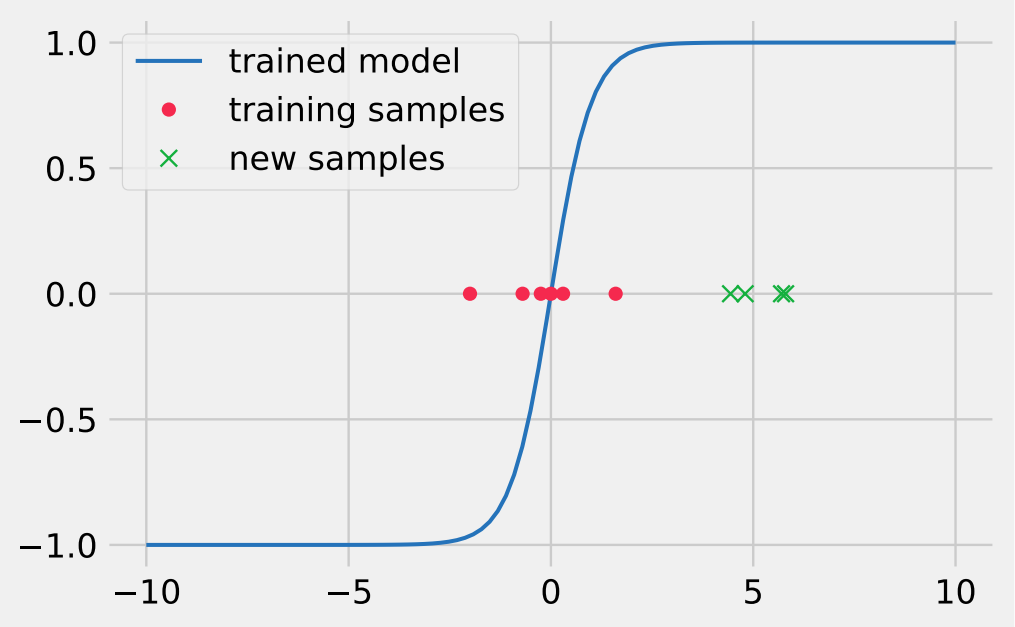
Let's see why re-using networks on new training sets can be difficult. Consider a single neuron and assume a $\tanh$ activation function:



We want to fine tune the training with new images taken on cloudy days. The new samples values x_i (green crosses) are centred around 5. For that input range the derivative of $\tanh$ is almost zero (see graph).

Transfer learning and vanishing gradients

Let's see why re-using networks on new training sets can be difficult. Consider a single neuron and assume a $\tanh$ activation function:



...which means we have a problem of vanishing gradients. It will be difficult to update the network weights.

Normalisation Layers

It is thus critical for the input data to be in the correct value range. To cope with possible shifts of value range between datasets we can use a Normalisation Layer, whose purpose it to scale the data according to the training set statistics.

Denoting x_i an input value of the normalisation layer, The output x_i' after normalisation is defined as follows:

$$x_i' = \frac{x_i - \mu_i}{\sigma_i}$$

where μ_i and σ_i are computed off-line based on the input data statistics.

Normalisation Layers

Batch Normalisation (BN) is a particular type of normalisation layer where the rescaling parameters μ and σ are chosen as follows.

For training, μ_i and σ_i are set as the mean value and standard deviation of x_i over the mini-batch. That way the distribution of the values of x_i' after BN is 0 centred and with variance 1.

For evaluation, μ_i and σ_i are averaged over the entire training set.

BN can help to achieve higher learning rates and be less careful about optimisation considerations such as initialisation or Dropout.

> Sergey Ioffe, Christian Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift." (2015) [https://arxiv.org/abs/1502.03167]

Going Deeper

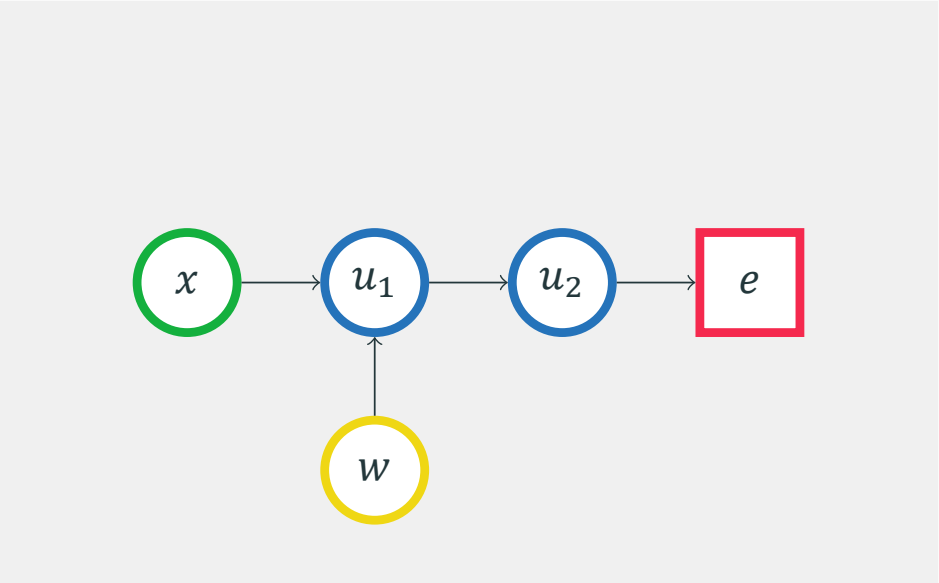
Going Deeper

There has been a general trend in recent years to design deeper networks. Deeper network are known to produce more complex features and tend to generalise better.

Training deep networks is however difficult. One key recurring issue being the problem of vanishing gradients.

Going Deeper

Recall the problem of vanishing gradients on this simple network:

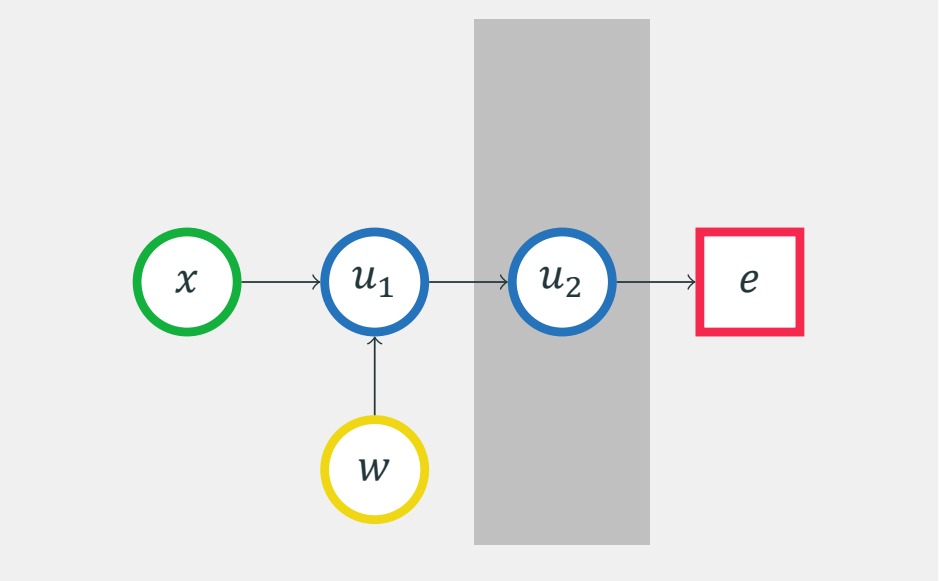


$$\frac{\partial e}{\partial w} = \frac{\partial e}{\partial u_2}\frac{\partial u_2}{\partial u_1}\frac{\partial u_1}{\partial w}$$

During the gradient descent, we evaluate $\frac{\partial e}{\partial w}$, which is a product of the intermediate derivatives. If any of these is zero, then $\frac{\partial e}{\partial w} \approx 0$.

Going Deeper

Recall the problem of vanishing gradients on this simple network:

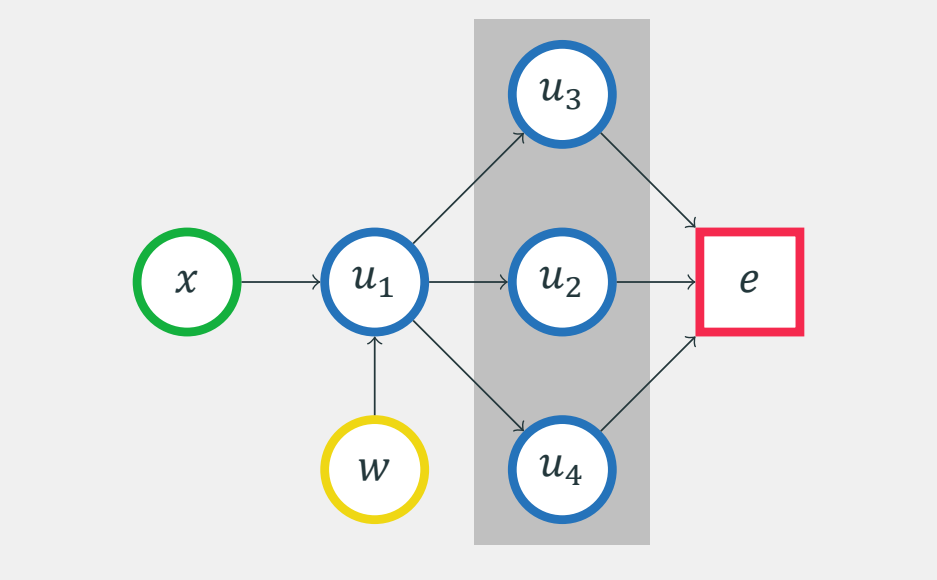


$$\frac{\partial e}{\partial w} = \frac{\partial e}{\partial u_2}\frac{\partial u_2}{\partial u_1}\frac{\partial u_1}{\partial w}$$

Now consider the layer containing u_2 ...

Going Deeper

Recall the problem of vanishing gradients on this simple network:

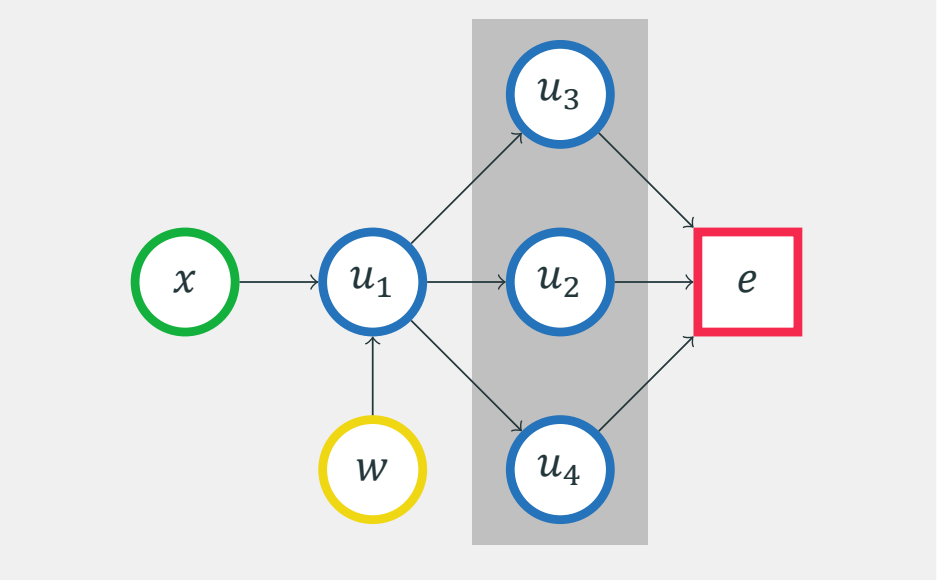


$$\frac{\partial e}{\partial w} = \frac{\partial e}{\partial u_2}\frac{\partial u_2}{\partial u_1}\frac{\partial u_1}{\partial w} + \frac{\partial e}{\partial u_4}\frac{\partial u_4}{\partial u_1}\frac{\partial u_1}{\partial w} + \frac{\partial e}{\partial u_3}\frac{\partial u_3}{\partial u_1}\frac{\partial u_1}{\partial w}$$

...and replace it with a network of 3 units in parallel (u_3 , u_2 , u_4).

Going Deeper

Recall the problem of vanishing gradients on this simple network:



$$\frac{\partial e}{\partial w} = \frac{\partial e}{\partial u_2}\frac{\partial u_2}{\partial u_1}\frac{\partial u_1}{\partial w} + \frac{\partial e}{\partial u_4}\frac{\partial u_4}{\partial u_1}\frac{\partial u_1}{\partial w} + \frac{\partial e}{\partial u_3}\frac{\partial u_3}{\partial u_1}\frac{\partial u_1}{\partial w}$$

It is now less likely to $\frac{\partial e}{\partial w} \approx 0$ as all three terms need to be null.

So a simple way of mitigating vanishing gradients is to avoid a pure sequential architecture and introduce parallel paths in the network.

This is what was proposed in GoogLeNet (2014) and ResNet (2015).

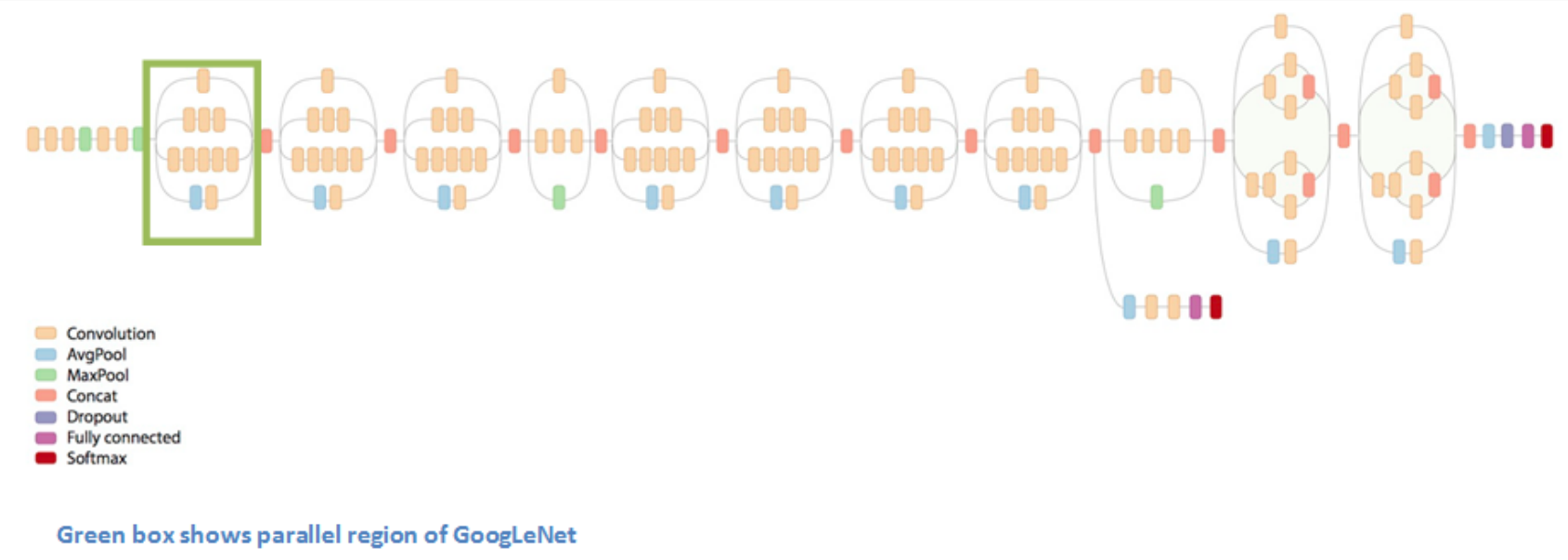
GoogLeNet: Inception

GoogLeNet was the winner of ILSVRC 2014 (the annual competition on ImageNet) with a top 5 error rate of 6.7% (human error rate is around 5%).

The CNN is 22 layer deep (compared to the 16 layers of VGG).

> Szegedy et al. “Going Deeper with Convolutions”, CVPR 2015. (paper link: https://goo.gl/QTCe66)

GoogLeNet: Inception

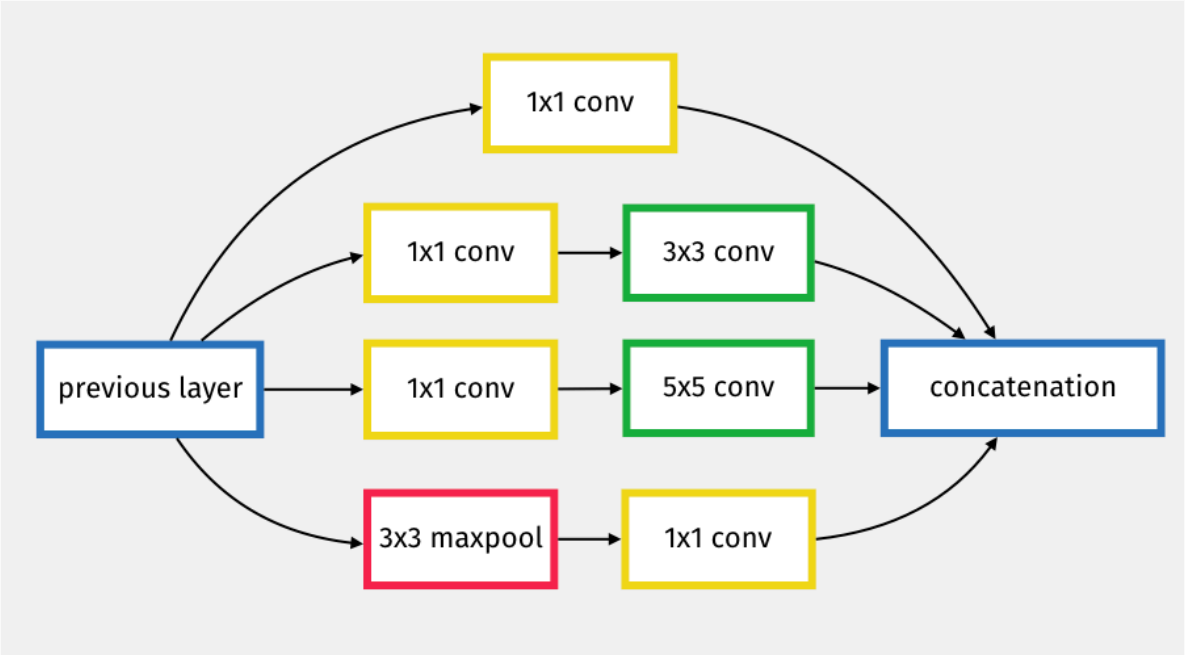


Green box shows parallel region of GoogLeNet

The architecture resembles the one of VGG, except that instead of a sequence of convolution layers, we have a sequence of inception layers (eg. green box).

GoogLeNet: Inception

An inception layer is a sub-network (hence the name inception) that produces 4 different types of convolutions filters, which are then concatenated (see this video: [https://youtu.be/VxhSouuSZDY]).



The inception network creates parallel paths that help with the vanishing gradient problem and allow for a deeper architecture.

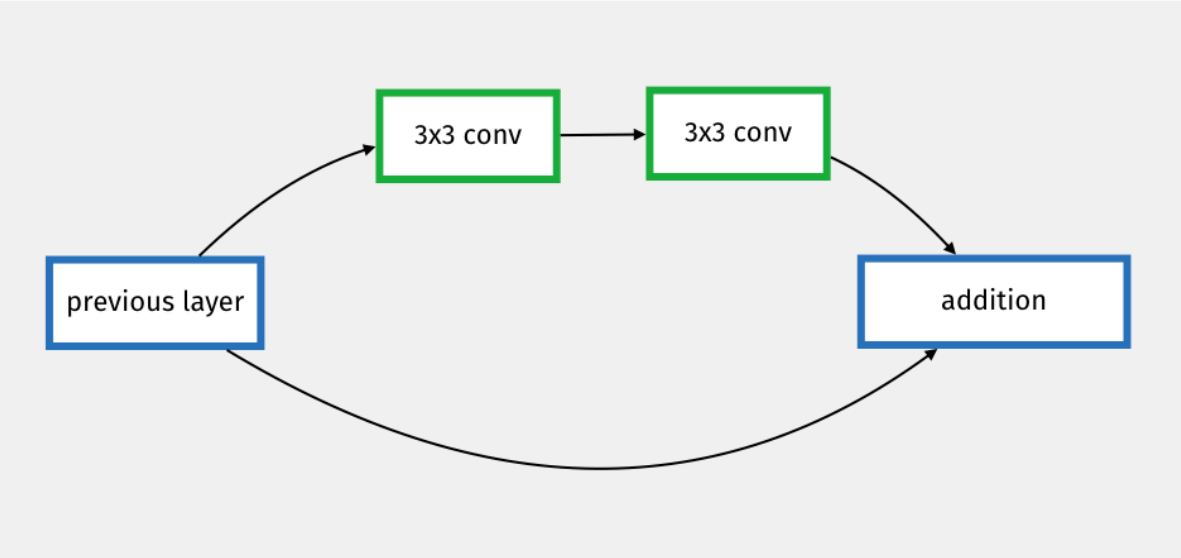
ResNet: Residual Network

ResNet is a 152 (yes, 152!!) layer network architecture developed at Microsoft Research that won ILSVRC 2015 with an error rate of 3.6% (better than human performance).

Kaiming He et al (2015). “Deep Residual Learning for Image Recognition”. [https://goo.gl/Zs6G6X]

ResNet: Residual Network

Similarly to GoogLeNet, at the heart of ResNet is the idea of creating parallel connections between deeper layers and shallower layers. The connection is simply done by adding the result of a previous layer to the result after 2 convolutions layers:



The idea is very simple but allows for a very deep and very efficient architecture.

Generative Adversarial Networks

So far, we have mainly looked as discriminative models, which try to estimate the likelihood $P(Y|X)$. For instance logitistic regression would be a discriminative classifier.

Another kind of application is to look at generative models, which model the probability distributions of the data itself, ie. $P(X)$ or $P(X|Y)$.

The objective is to learn how to synthesise new data, that is similar to the training data.

Example

Given a dataset of faces, and you would like to generate new realistic looking faces.

It is not immmediately obvious how this can be done using the kind of approach we've adopted so far. We have a number of issues to solve:

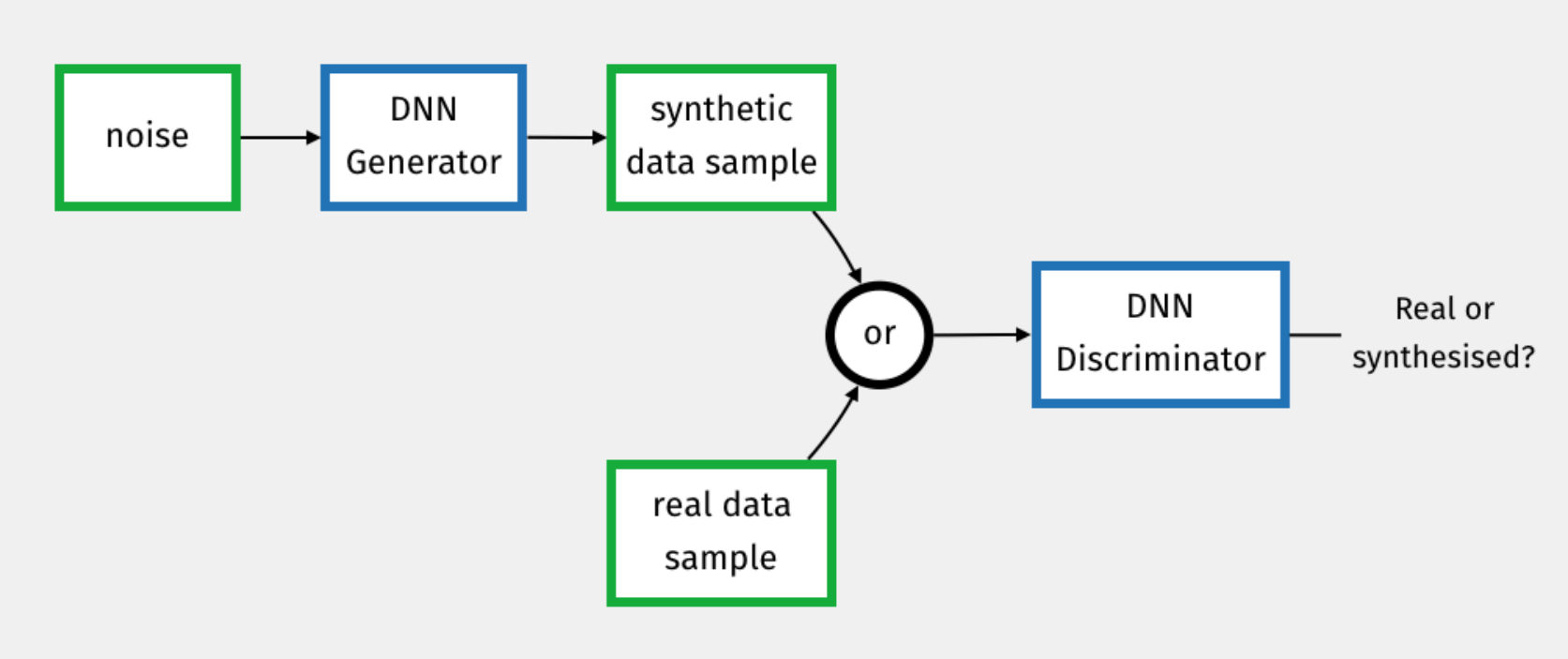
A. How do you generate the new data?

B. Then, what would be the loss function? If we generate data that is different from any sample from our existing dataset, how do we measure how realistic this is?

Two main solutions are Generative Adversarial Networks (GANs) (see next slides) and **Variational Autoencoders** (VAEs) (see specific hand-out).

Ian Goodfellow et al. “Generative Adversarial Network” (2014). [https://arxiv.org/abs/1406.2661]

GAN's architecture (see below) is made of two NN: a Generator Network, that is responsible for generating fake data, and a Discriminator Network that is responsible for detecting whether data is fake or real.



Typically the **generator** network takes an input noise and transforms that noise into a sample of a target distribution. This is our photo portrait generator: it transforms a random seed into a photo portrait of a random person.

The loss function for that generator is the **discriminator** network that in our case can classify between fake and real photos.

It is thus an arms race where each network is trying to outdo the other.

Both problems taken independently are hard because the generator is missing a loss function and the discriminator is missing data but by addressing both problem at the same time in a single architecture, we can solve for both problems.

A GAN can be trained by alternating the phases of training:

1. Freeze the Generator and train the Discriminator (eg. 1+ epochs)
2. Freeze the Discriminator and train the Generator (eg. > 1 epochs)
3. Repeat steps 1. and 2.

The issue is that the convergence is a bit complicated...

For instance, if the Generator is perfect, there is no flaw to be learned for the discriminator, and the discriminator will be a bit random, and at the next Generator will not be put in check anymore by the discreminator.

Thus training a GAN network is particularly difficult but the benefits are spectacular.

Example of GAN generating pictures of fake celebrities (see 2018 ICLR NVidia paper here [https://goo.gl/AgxRhp])