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

Advances in Neural Network Architectures and Training

François Pitié

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

[4C16/5C16] Deep Learning and its Applications — 2024/2025

In this handout we cover some of the advances in network architecture design and training that have happened since 2012.

The idea is to try to highlight some of the typical components of a modern architecture and training pipeline.

Transfer Learning

Reusing off-the-shelf networks

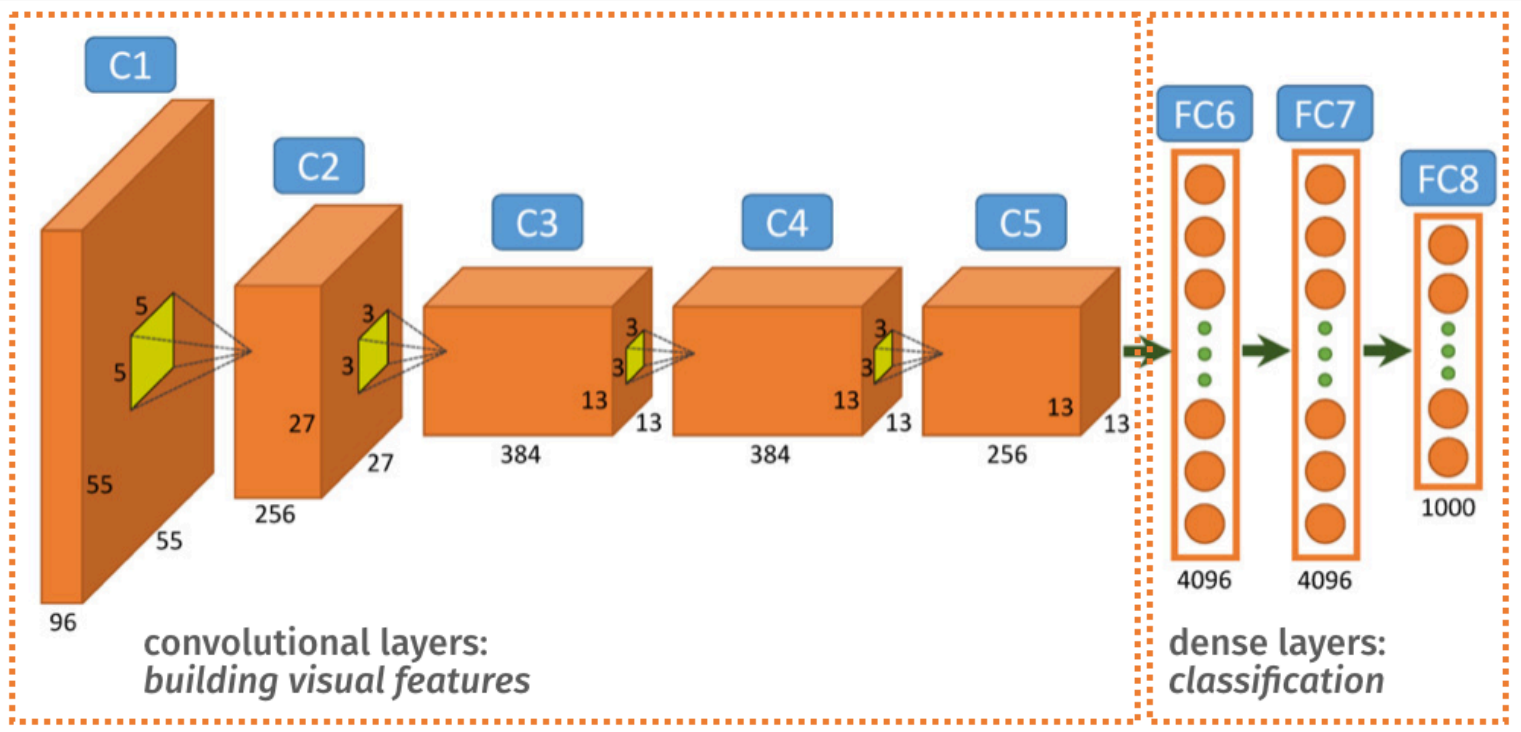
Transfer learning is the idea of reusing knowledge learned from a task to boost performance on a related task.

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 hundreds of thousands of pictures. This is however unpractical 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 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)
```

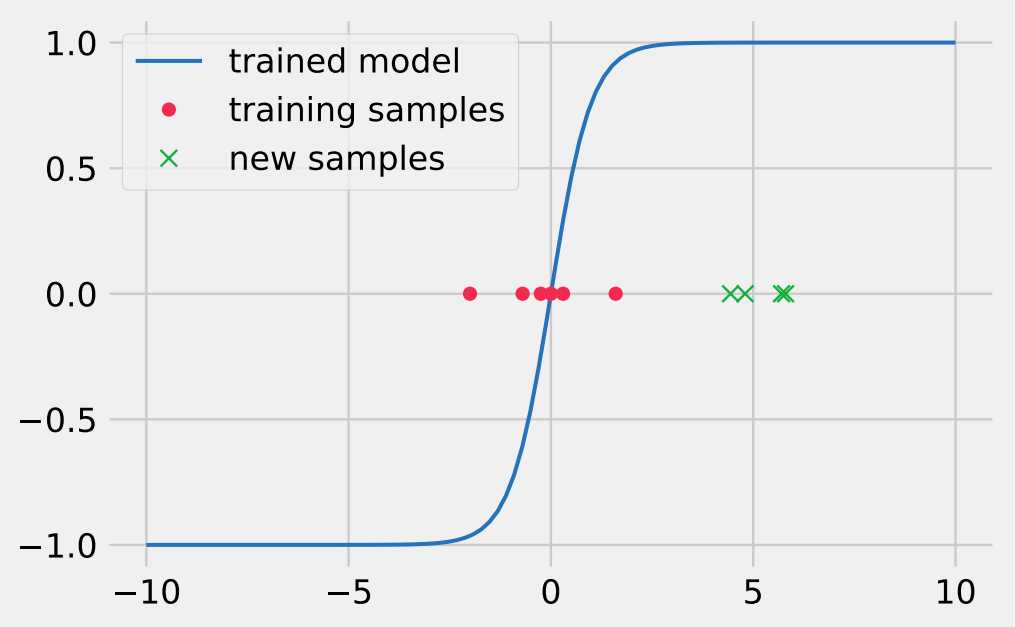
In most image based applications you should first consider reusing off-the-shelf networks such as VGG, GoogLeNet or ResNet. It has been shown (see link below) that using such generic visual features yield state of the art performances in most applications.

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

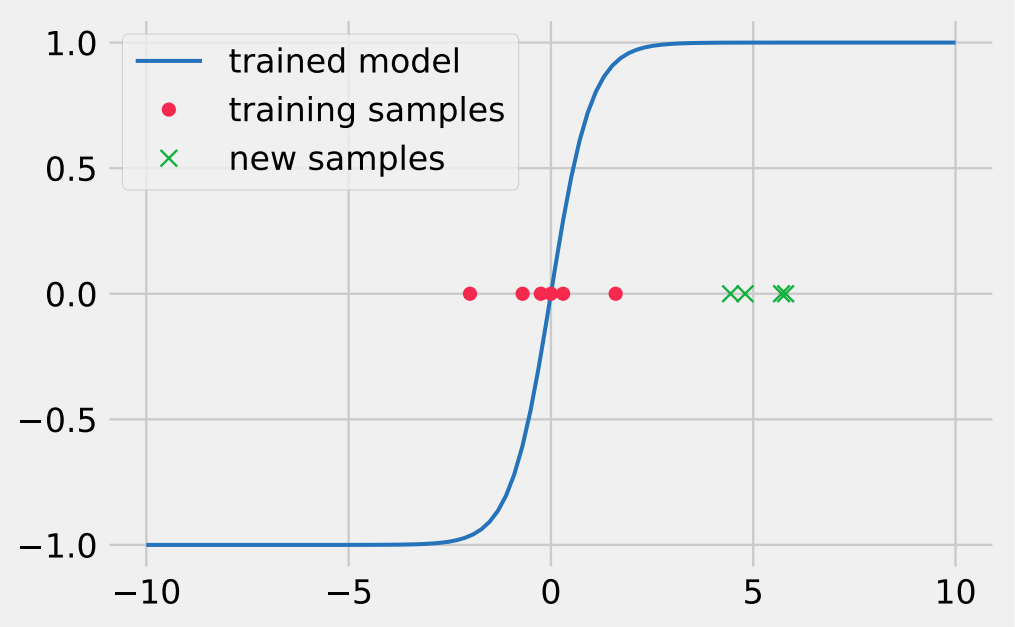
Let's see why re-using networks on new training sets can be difficult. Consider a single neuron and assume a $\tanh$ activation function:



$f(x_i, w) = \tanh(x_i + w)$. 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

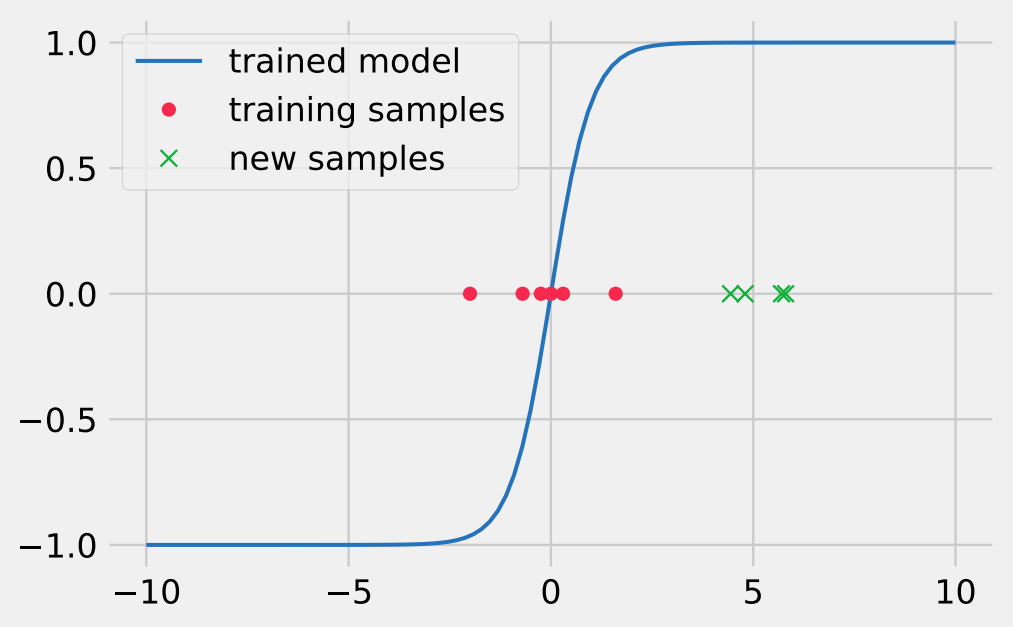
Let's see why re-using networks on new training sets can be difficult. Consider a single neuron and assume a $\tanh$ activation function:



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 set 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

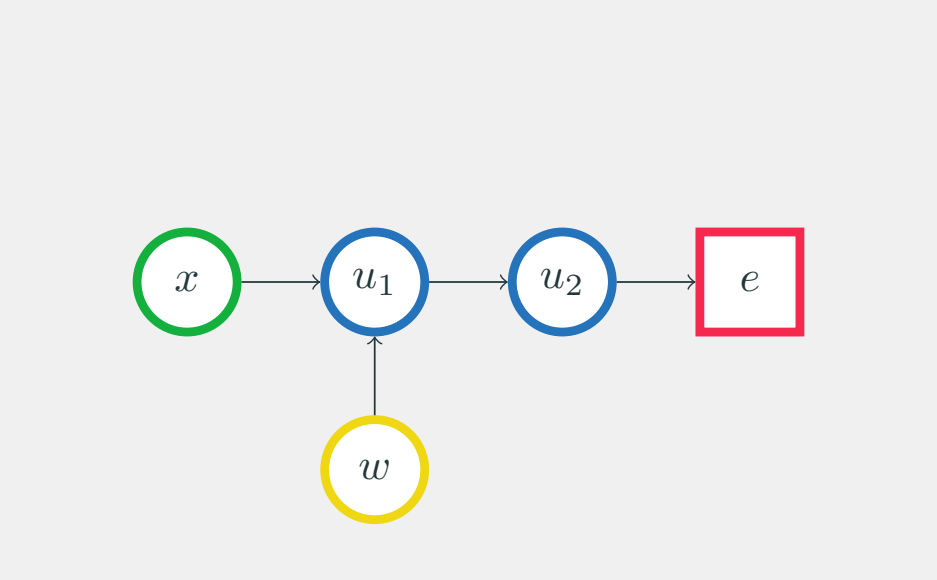
Going Deeper

As the potential for deeper networks to generalise better became evident, a fervent competition to push the boundaries further began after 2012.

The key hurdle in this race was that, because of vanishing gradients, sequential architectures like VGG couldn't be trained for more than 14-16 layers.

Going Deeper

Recall the problem of vanishing gradients on this simple network:

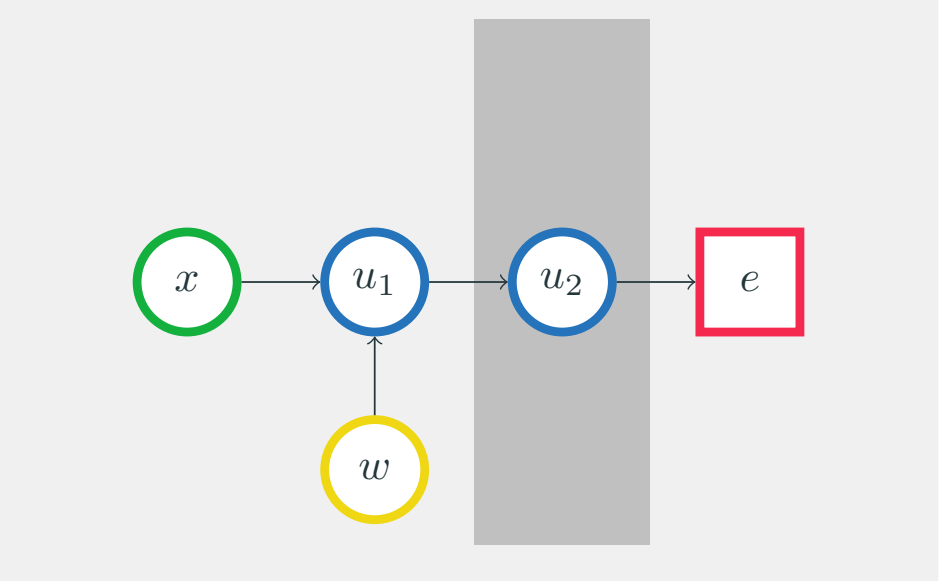


$$\frac{\partial e}{\partial w} = \frac{\partial e}{\partial u_2}\frac{\partial u_2}{\partial u_1}\frac{\partial u_1}{\partial w}$$

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:

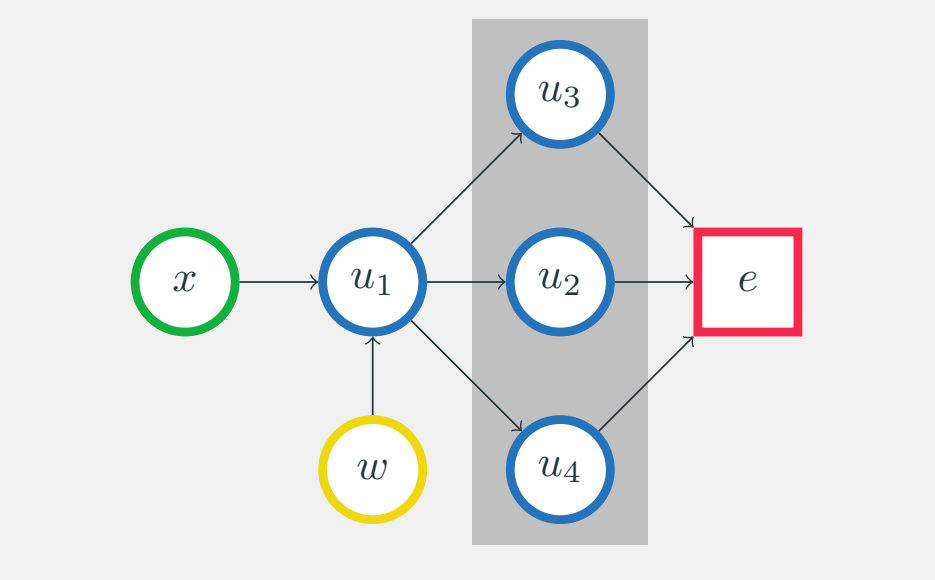


$$\frac{\partial e}{\partial w} = \frac{\partial e}{\partial u_2} \frac{\partial u_2}{\partial u_1} \frac{\partial u_1}{\partial w}$$

Now consider the layer containing u_2 ...

Going Deeper

Recall the problem of vanishing gradients on this simple network:

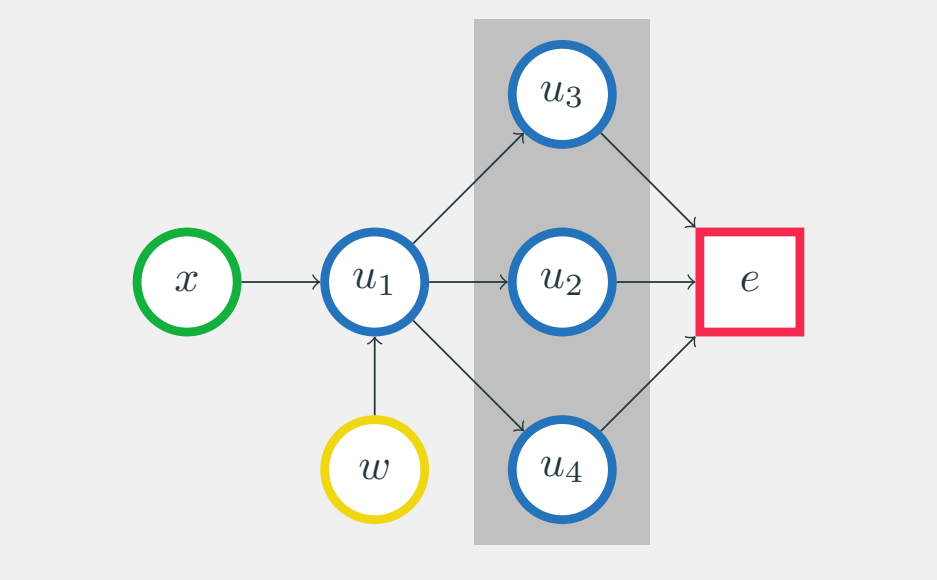


$$\frac{\partial e}{\partial w} = \frac{\partial e}{\partial u_2}\frac{\partial u_2}{\partial u_1}\frac{\partial u_1}{\partial w} + \frac{\partial e}{\partial u_4}\frac{\partial u_4}{\partial u_1}\frac{\partial u_1}{\partial w} + \frac{\partial e}{\partial u_3}\frac{\partial u_3}{\partial u_1}\frac{\partial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

...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).

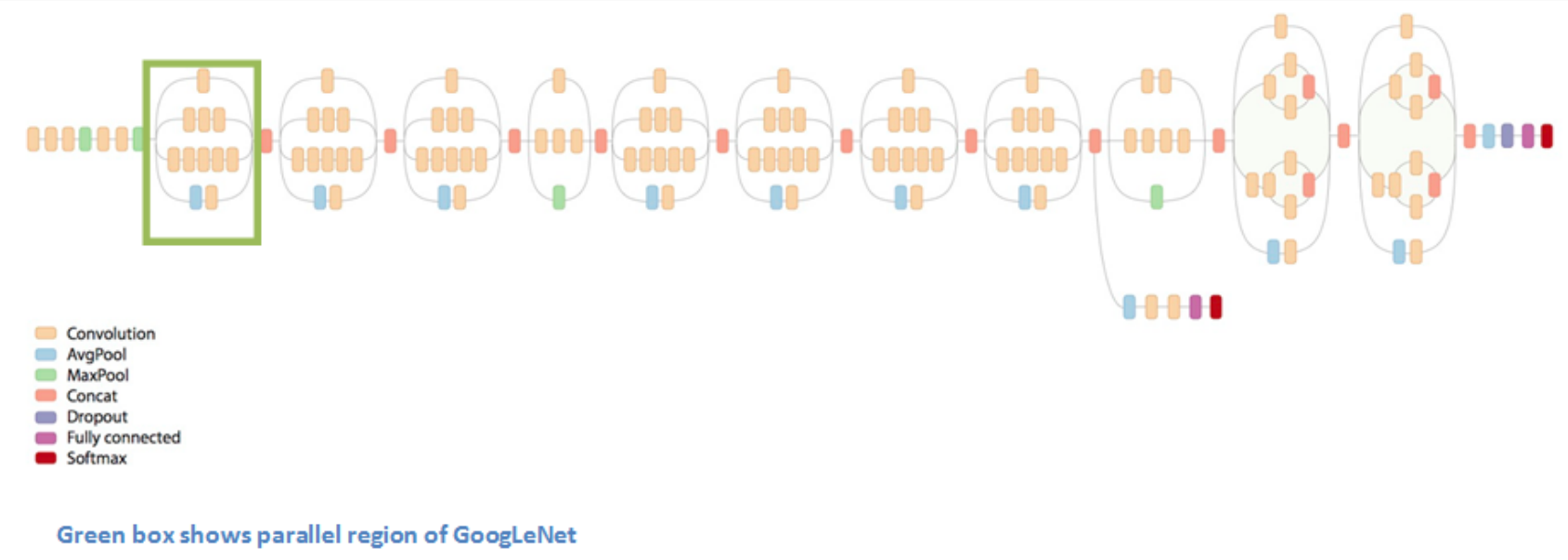
GoogLeNet: Inception

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

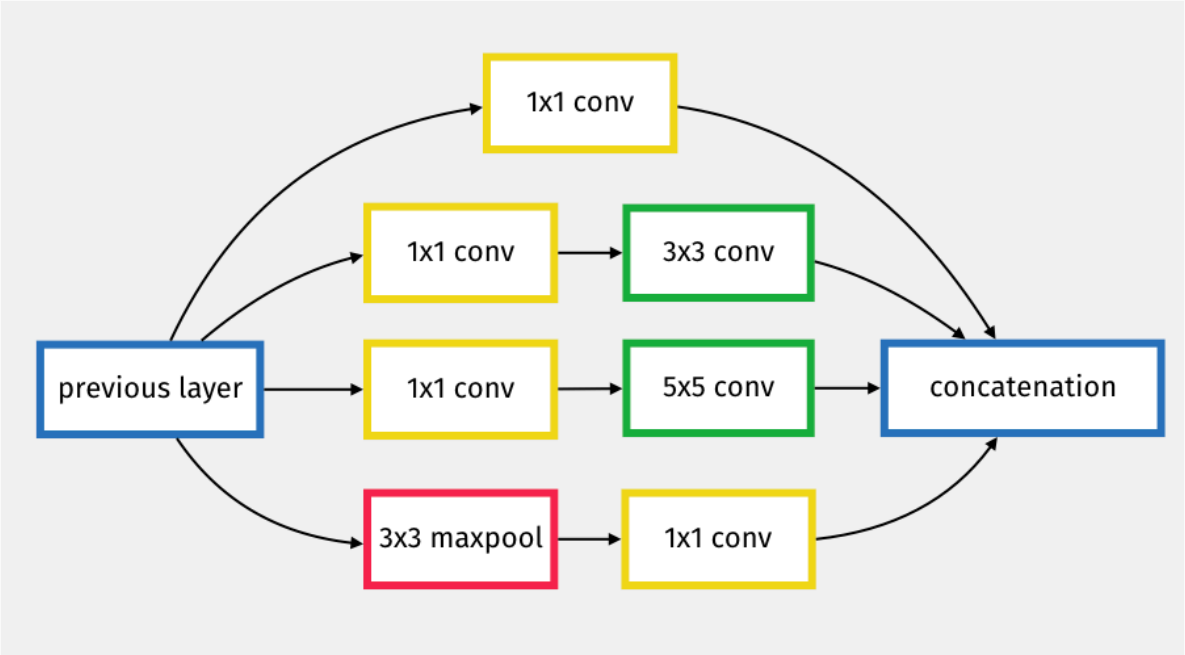


Green box shows parallel region of GoogLeNet

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.

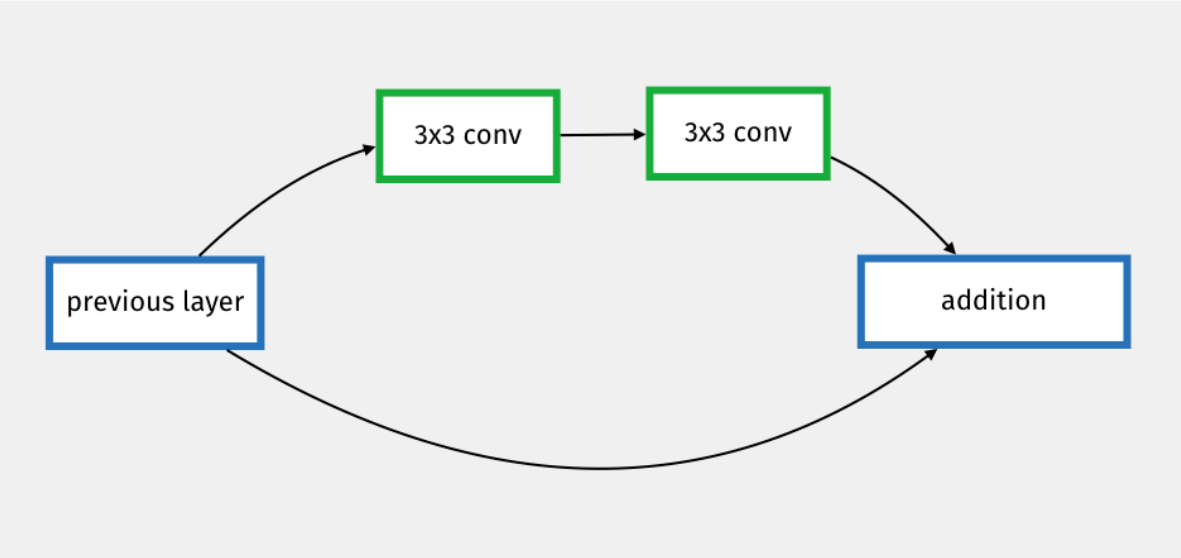
ResNet: Residual Network

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:

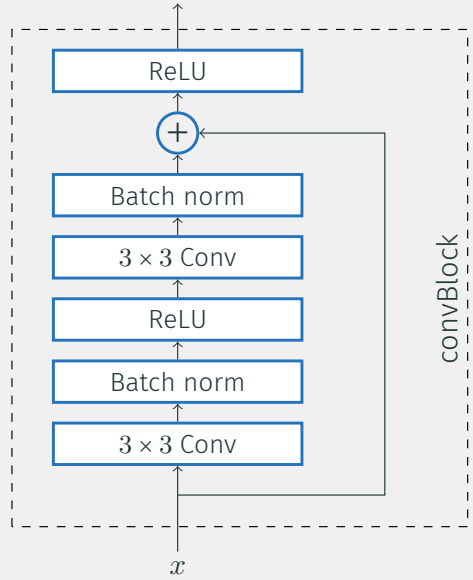


The idea is very simple but allows for a very deep and very efficient architecture.

Putting it Together...

Here is an example of a convolution block implementation that includes a resnet skip connection, batchnorm and He's initialisation:

```
def ConvBlock(x, filters):
  x0 = x
  x = Conv2D(filters, (3, 3), padding='same',
             kernel_initializer='he_normal')(x)
  x = BatchNormalization(x)
  x = Activation('relu')(x)
  x = Conv2D(filters, (3,3), padding='same',
             kernel_initializer='he_normal')(x)
  x = BatchNormalization(x)
  x = add([x, x0])
  x = Activation('relu')(x)
  return x
```

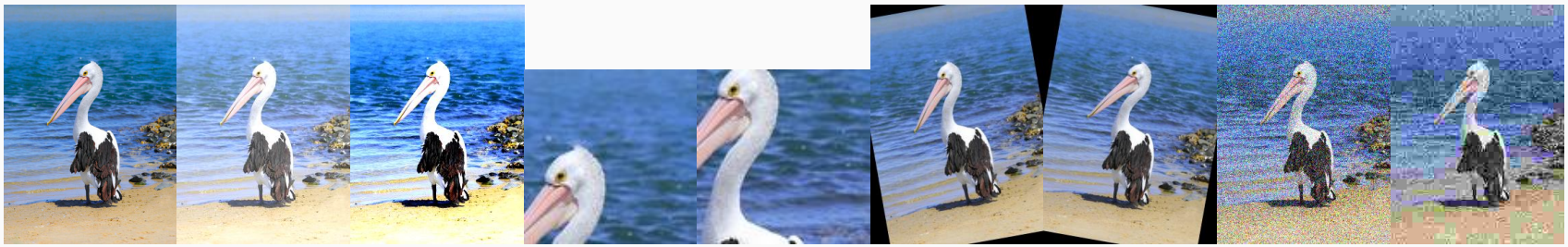


A Modern Training Pipeline

Data Augmentation

The pipeline starts with the dataset. It is often possible to naturally increase your dataset by generating variants of your input data.

With **images**, we can apply image processing operations, such as geometric transformations (eg. crop, flip, rotation, zoom), colour space transformations (eg. brightness or contrast change), blur the image, add noise, include occlusions, JPEG compression, *etc.*



`https://keras.io/api/layers/preprocessing_layers/image_augmentation/`

Data Augmentation

With **text** you can randomly replace words with their synonyms, shuffle sentences, delete words or characters, *etc.*

With **audio**, you can add noise, reverb, compression, *etc.*

Synthetic data can be a good way to expand the original dataset, but be mindful that it is easier to overfit artificial data. Generative DNNs (see later handout), can also be used to synthesise data (*eg.* you can use ChatGTP to generate text for your dataset).

Initialisation

Initialisation needs to be considered carefully. Starting at $w = 0$ is probably not a good idea, as you are likely to be stuck into some special local minimum, with the gradient stuck at zero from the start.

The idea would then be to start at random. We need, however to be careful, and control the output at each layer to avoid a situation where gradients would explode or vanish through the different layers.

Initialisation

For ReLU, a popular initialisation is He's initialisation. For each layer l, the bias b and weights w are initialised as $b_l = 0, w_l \sim \mathcal{N}(0, \sqrt{2/n_{l-1}})$, where n_{l-1} is the number of neurons in prev layer. Using this guarantees stable gradients throughout the network (at least at the start of the training).

`https://keras.io/api/layers/initializers/`

> More Detailed Explanations and very nice Demos
> `https://www.deeplearning.ai/ai-notes/initialization/`

> Kaiming He et al. Delving Deep into Rectifiers
> `https://arxiv.org/abs/1502.01852`

He Initialisation - Derivations

Consider a sequence of conv or dense layers (l):

$$\mathbf{y}_l = \mathbf{W}_l \mathbf{x}_l \quad \text{and} \quad \mathbf{x}_l = \max(\mathbf{y}_{l-1}, 0).$$

Assuming independence and weights and biases with zero mean:

$$\mathrm{Var}[y_l] = n_l \mathrm{Var}[w_l x_l] = n_l \mathrm{Var}[w_l] E[x_l^2]$$

for ReLU, $x_l = 0$ for $y_{l-1} < 0$, thus $E[x_l^2] = \frac{1}{2}\mathrm{Var}[y_{l-1}]$, and

$$\mathrm{Var}[y_l] = \frac{1}{2} n_l \mathrm{Var}[w_l] \mathrm{Var}[y_{l-1}].$$

One way to avoid an increase/decrease of the variance throughout the layers is to set:

$$\mathrm{Var}[w_l] = \frac{2}{n_l},$$

which we can achieve by sampling w_l from $\mathcal{N}(0, \sqrt{2/n_l})$.

Optimisers

As we have seen in earlier handouts, a number of optimisation techniques are available to us for training. Papers in the literature tend to gravitate around Adam or SGD. Adam seems to be the fastest out of the box, but best-in-class models tend to be trained on SGD with momentum, as they find local minima that generalise better than the ones found by Adam.

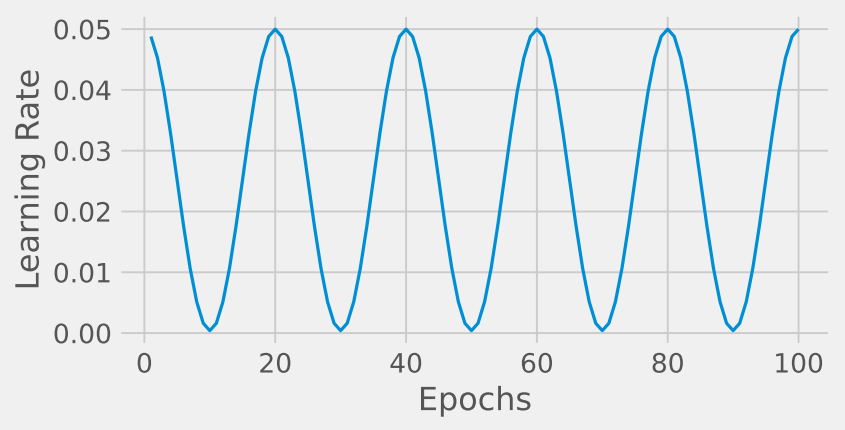
Since then, an improved version of Adam, AdamW was proposed and seems to fix Adam's shortcomings and bring it to the level of SGD.

> Loshchilov and Hutter. "Decoupled Weight Decay Regularization". 2017.
> `https://arxiv.org/abs/1711.05101v3`

Learning Rate Scheduler

A critical aspect of the training regime is the specification of the learning rate scheduling.

In 2017 was popularised the idea of warm restarts, which periodically raise the learning rate to temporary diverge and allow to hop over hills. A variant of this scheme is the cosine annealing schedule:



> Loshchilov and Hutter. “SGDR: Stochastic Gradient Descent with Warm Restarts”. 2017.
> `https://arxiv.org/abs/1608.03983`

> `https://residentmario.github.io/pytorch-training-performance-guide/lr-sched-and-optim.html`

Learning Rate Scheduler

Another idea popularised by fast.ai, is that, as the gradient is initially not very stable, we should starts slow, then go fast, and then slow again to go down that local minimum. This is called a one-cycle learning rate scheduler.

Learning Rate Scheduler

Here is an example on how to setup a cosine annealing schedule with AdamW in Keras:

```
cosine_decay_scheduler = optimizers.schedules.CosineDecay(
     initial_learning_rate, decay_steps, alpha=0.0, name=None)
optimizer = optimizers.AdamW(learning_rate=cosine_decay_scheduler)

model.compile(optimizer=optimizer, ...)
```

Take Away

It is typical for modern convolution networks to enhance the original convolution/activation block with a combination of normalisation layers and residual connections. The new blocks are much more resilient to the vanishing gradient problem, and allows both 1) to go much deeper, and 2) to be efficiently used for transfer learning.

Modern training pipelines typically include some data augmentation step, a dedicated initialisation strategy (eg. He or Xavier), a careful consideration of the optimisation (eg. AdamW) and of the learning rate schedule (eg. cosine annealing), with sometimes a transfer-learning/fine-tuning approach to kick start the training.

Keep in mind that there are no universal truth here. These are popular techniques but they might not be optimal for your problem.