ASSIGNMENT 3: CNN

Rubén Martínez Sánchez de la Torre

Assigment Definition:

In this assignment, we will compare different Deep Convolutional Neural Networks in terms of:

- **"number of parameters"**
- **"inference time"**
- **"performance".**

You will be required to construct the following networks: 1. AlexNet 2. VGG-16 3. Your custom Deep CNN (CNN_5x5_Net): use only 5x5 convolutional kernels (hint: in VGG-16, all convolutional kernels were 3x3)

- **Check the number of parameters vs inference time (by generating random noise image and feed-forward through the networks)**

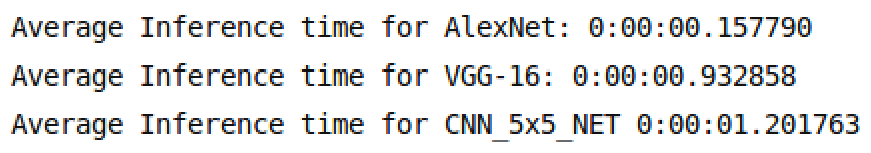
  You can find the whole notebook attached with this document and in the following Jovian link and :

  https://jvn.io/rubensilver/dac35e1a7479468e90af9e5d114ca732

  AlexNet: `Total params: 23,981,450 Trainable params: 23,981,450 Non-trainable params: 0`

  VGG-16: `Total params: 134,301,514 Trainable params: 134,301,514 Non-trainable params: 0`

  CNN_5x5_Net: `Total params: 160,453,450 Trainable params: 160,453,450 Non-trainable params: 0`



```
Average Inference time for AlexNet: 0:00:00.157790
Average Inference time for VGG-16: 0:00:00.932858
Average Inference time for CNN_5x5_NET 0:00:01.201763
```

- **Explain the results: why one was faster than the another, what was the key point?**

  In the previous section we can see that AlexNet is the network with the smallest number of parameters and therefore with the fastest inference time. The VGG-16 network has more parameters than AlexNet but less than CNN_5x5_Net and we can observe that its inference time is greater than the inference time of Alexnet but less than the inference time of CNN_5x5_Net.

  The key point was the number of trainable parameters of each network.

The formula to get the **size of the output feature map** after applying a convolutional kernel over an input image / input feature map is:

$FMsize = \frac{W - K + 2P}{S} + 1$ where **W** is the size of weight or the height of the input image or feature map, **K** is the kernel size that will be applied over the image, **P** is the padding and **S** is the strides.

For example, if you have an input image of 12x12x3 and you're going to apply a kernel size 5x5x3 using padding = 0 and strides = 1, the size of the output feature map will be:

$$FMsize = \frac{12 - 5 + 2*0}{1} + 1 = 8$$

Using the formula above the original 12x12x3 image will become a 8x8x1 image.

The **number of multiplications needed** will be 5x5x3=75 multiplications every time the kernel moves and as we have to move the kernel 8x8 times the final number of multiplications will be 5x5x3x8x8= 4800.

Each kernel is going to learn to detect a visual feature and if we want to train, for example, 64 kernels of 5x5 over an image of 3 channels then the number of weights will be: 32 * 5 * 5 * 3 = 2400 and **the memory needed** to store the output feature map will be 8 * 8 * 32 = 2048. We could multiply this value by 4 to get the raw number of bytes because each floating point number is composed by 4 bytes or by 8 if we are using double precision.

If we set **S= 1** and we pad the input image / input feature map with 0's so that $P = \frac{(K-1)}{2}$ then we are going **to preserve the input size**.

If we use **smaller kernel sizes** we will have smaller receptive fields and we will capture smaller and complex features in the images.

If we use **wider convolutions**, I mean, layers with more kernels it will be easier for the GPU to process a single big chunk of data instead of a lot of smaller ones (that would be the case of having more layers with fewer kernels per layer).

- **Add "Batch Normalization" and/or "Dropout" layers to the networks (AlexNet, VGG-16, Your custom CNN_5x5_Net)**

  This code can be found in the attached notebook.

  When we train a Deep Neural Network we usually normalize the input data so that it follows a normal distribution with zero mean and unitary variance to assure that all input data is in the same range and to prevent early saturation in non linear function among other benefits.

  But as the distribution of the activations is going to be hanging during the training process we are going to have a problem in the intermediate layers. Each layer is going to learn to adapt itself to a new distribution each train step. This situation is known as **internal covariate shift** and it will slow down the training process.

To address that problem we are going to use a **Batch Normalization layer** which is a technique to normalize the input of each layer to prevent the internal covariate shift.

This layer should be applied before of the non linearity of the current layer according to the original paper:

"We add the BN transform immediately before the non-linearity, by normalizing x = Wu+b. We could have also normalized the layer inputs u, but since u is likely the output of another nonlinearity, the shape of its distribution is likely to change during training, and constraining its first and second moments would not eliminate the covariate shift. In contrast, Wu + b is more likely to have a symmetric, non-sparse distribution, that is “more Gaussian”; normalizing it is likely to produce activations with a stable distribution."

The process to compute the output of this layer is:

* Compute the mean and variance of the inputs of the layer where BN is being applied:

Batch mean → $\mu=\frac{1}{n}\cdot\sum_{i=1}^{n} x_i$

Batch variance → $\sigma^2=\frac{1}{n}\cdot\sum_{i=1}^{n}(x_i-\mu)^2$

* Normalize the inputs of the layer:

$$\overline{x}_i=\frac{x_i-\mu}{\sqrt{\sigma^2+\varepsilon}}$$

* Get the output of the BN layer scaling and shifting the normalized inputs:

$y_i=\gamma\cdot\overline{x}_i+\beta$ where γ and β are learned during training.

When we use a **Dropout layer**, at each training step, each neuron will have a probability 1 -p to be removed in that step or a probability p to be kept. The main benefit of this method is to prevent overfitting.

I have found a paper whose link is https://arxiv.org/pdf/1905.05928.pdf that proposes a layer composed by a **BN layer followed by a Dropout layer** to speed up the training process:

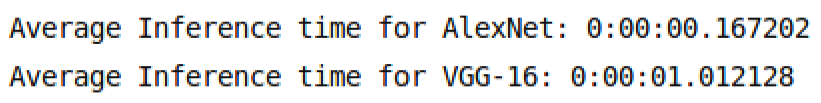
```
from tensorflow.keras.layers import BatchNormalization, Dropout

def Independent_Components_layer(inputs, p):

   x = BatchNormalization(inputs)

   x = Dropout(p)(x)

   return x
```

- **Check how does inference time is changing after adding those layers Note: Framework: we will be using Keras in this assignment.**

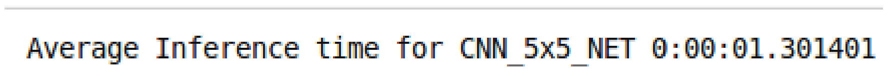
  AlexNet: **`Total params:` `24,019,722` `Trainable params:` `24,000,586` `Non-trainable params:` `19,136`**

  VGG-16: **`Total params:` `134,351,178` `Trainable params:` `134,326,346` `Non-trainable params:` `24,832`**

  `CNN_5x5_Net:` `Total params:` `160,503,114` `Trainable params:` `160,478,282` `Non-trainable params:` `24,832`



```
Average Inference time for AlexNet: 0:00:00.167202
Average Inference time for VGG-16: 0:00:01.012128
```



```
Average Inference time for CNN_5x5_NET 0:00:01.301401
```