In [28]:
```
import torch
import my_utils as mu
import pandas as pd
import torch.utils.data as data
from torch import nn
```

Elliot Linsey Neural Network Exam ECS659P June 2022

Q1.A

We use models with more than one hidden layer in order to extract more features from our input data and model more complex interactions. Also, we can introduce non-linear relationships using activation functions that are more accurately able to model data, an example being the relu function which can be seen in Multi-Layer Perceptrons (MLPs). In making a model deeper, we learn high-level features from our data in an incremental manner, whereas with a shallow network we may only recognise lines and edges, with a deep network we can recognise concepts such as 'face' or 'dog'. Deeper neural networks are also better approximators, it is far easier to learn functions using deeper networks than with shallow ones.

Q1.B

One technique used is batch normalisation. This normalises per channel by subtracting the mean and dividing by the standard deviation to give us a mean of 0 and standard deviation of 1. This is then rescaled channelwise using the parameter gamma, and an offset beta is also added. Both these parameters can be learned and optimised during the training process. What batch normalisation provides is extra stability to the neural network, allowing it to be trained longer as well as with a higher learning rate. It also makes it less sensitive to weight initialisation methods.

Another technique is the skip connection seen in resnet blocks. This is where you add the input feature to the output of your block, so y = F(x) + x. This allows a direct path for propagating gradients during backpropagation. As the model gets deeper, it becomes less able to learn features, skip connections allow the output of one layer to skip ahead and be fed as input into the next layer to prevent this.

Q2

In [29]:
```
inp2 = torch.rand(16,4,128,128)
```

In [30]:
```
conv1 = nn.Conv2d(4,256,5,stride=1,padding=0)
conv2 = nn.Conv2d(64,64,3,stride=1,padding=0)
fl = nn.Flatten()
```

In [31]:
```
ex1 = conv1(inp2)
ex1.size()
```

Out[31]:
```
torch.Size([16, 256, 124, 124])
```

weights = 256x4x5x5

In [32]:
```
inp3 = torch.rand(16,64,124,124)
```

In [33]:
```
ex2 = conv2(inp3)
ex2.size()
```

Out[33]:
```
torch.Size([16, 64, 122, 122])
```

weights = 4x(64x64x3x3)

In [34]:
```
inp4 = torch.rand(16,256,122,122)
```

In [35]:
```
ex3 = fl(inp4)
ex3.size()
```

Out[35]:
```
torch.Size([16, 3810304])
```

weights = 3810304 x 20

In [39]:
```
print('weights with no avg pooling ' + str((256*4*5*5)+(4*(64*64*3*3))+(3810304*20)))
```

```
weights with no avg pooling 76379136
```

In [40]:
```
print('weights with avg pooling ' + str((256*4*5*5)+(4*(64*64*3*3))+(256*20)))
```

```
weights with avg pooling 178176
```

The first weights tensor is [256x4x5x5]

The second weights tensor is 4x[64x64x3x3]

This leaves us with dimensions of [256,122,122] when reconstituted

Flattened to a linear classifier this becomes [3810304,20]

weights = (256x4x5x5)+(4x(64x64x3x3))+(3810304x20) = 76379136

The question mentions an avg pooling layer which is not used. I'm assuming that if this was used prior to flattening we would instead result with:

[256,1,1] after being reconstituted and pooled

This results in a linear classifier of [256,20]

The weights then equal (256x4x5x5)+(4x(64x64x3x3))+(256x20) = 178176

Q3

This appears to be a type of CNN with a skip connection, due to the expand_channels with a 1x1 CNN layer. However, there are some issues with this. Firstly, identity appears to be taking the place of 'x', however he first applies the stem function to 'x' before assigning 'identity' to it. In a skip connection, identity should be untouched and defined as 'identity = x' before the stem function.

Also, the order of batch normalisation, relu and convolution is not optimal. A resnet block consists of convolution followed by batch normalisation followed by relu.

After this he defines 'x = identity', in a skip connection it should be 'x = x + identity', otherwise all the calculations you have done are discarded when x becomes identity.

After this he follows it by a relu then the last convolution, it should be the other way around with convolution followed by relu or perhaps remove the last convolutional layer.

Q4.A

The VGG net utilises a convolution arch to determine the number of convolutional layers and the number of output channels. The creators of VGG found that narrow convolutions of size 3x3 worked optimally, therefore I will set the kernel sizes to this, although later this can be changed and evaluated. Due to the large number of classes, it is most likely best to use deep and narrow convolutional kernel sizes. Then, utilising the convolutional arch I would run tests starting with only 1 convolutional layer to limit the number of parameters. I could see how high an accuracy I could achieve by using N number of VGG blocks using only one convolutional layer and increasing the number of output channels, then I would investigate the difference in accuracy when I start to add another convolutional layer to each VGG block, one at a time. I could also experiment with max pooling kernels and stride to reduce the feature maps even further.

When there is a number of N blocks I have identified that achieves 90% accuracy, it could be useful to utilise a gridsearch mechanism to further investigate parameters such as learning rate, convolutional kernels and stride/padding, as well as different optimisers like SGD or Adam.

Q4.B

A traditional RNN has a form of short-term memory and is not very good at remembering long-term information due to vanishing gradients. A version that allows greater long-term memory is that of Long Short Term Memory or LSTM. This adds a forget gate, input gate, output gate and memory cell to the RNN.

The forget gate determines how much of the previous memory cell information is contained, for example if the previous information stored in the memory cell was 'Wednesday', but we encounter a word further along of 'Thursday', the forget gate will determine how much of the original 'Wednesday' information in the memory cell should be forgotten.

The input gate determines how much information is taken into the candidate memory cell. It essentially determines how important a word is and whether it should be remembered or not, for example it will determine how much of the current input 'Thursday' should be remembered.

The candidate memory cell is a result of applying a tanh activation function to a matrix multiplication of the input matrix and weights plus the previous hidden state matrix and weights.

When put together, the memory cell is the result of the forget gate elementwise multiplying the previous memory cell plus the input gate elementwise multiplying the candidate memory cell. Therefore, if the forget gate has low values and the input gate has high values, the memory cell will be updated with the new candidate memory cell information.

Both the input gate and forget gate are fully-connected layers with activation functions (such as sigmoid or tanh).