

Q1-

In some cases where we don't have enough GPU memory, and also the convergence rate becomes very slow and less accurate, we can use *gradient accumulation* technique.

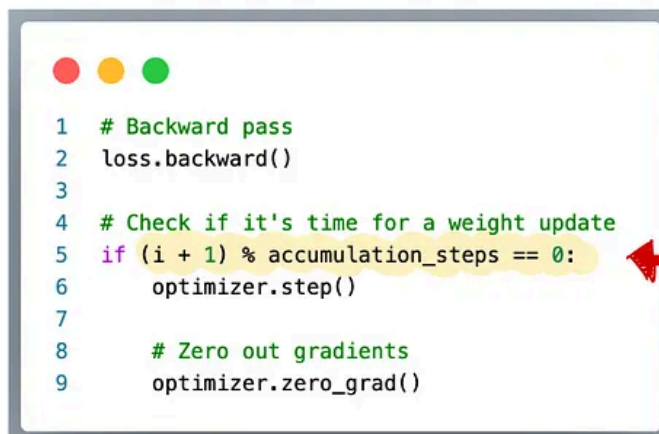
The main idea of this technique is to “NOT update the model weight each step!”. First we just keep the batch size as small as it can fit in GPU memory, and then we set the *accumulation step* parameter which defines the number of steps that we should wait before updating the weights. But how?

In this scenario we calculate the gradients and store them in a variable since we get the accumulation step then we use all the gradients to update weights, let's decompose it by an example:

Assume that we set the accumulation step equal to 5. In the first iteration we pass the mini-batch through the model then we backpropagate and calculate the gradient.(we don't update the weights yet!). We store the gradient of the first step in a variable like G for example. We repeat this process for step 2, the only difference is after calculating the gradient of the second step, we **Add** this value to the previous step! That's why it's called accumulation! The steps 3 and 4 are exactly the same as step 2. In step 5, after calculating the gradient, and adding to the G, we now want to update the weights using G. now we can pass the value G to the optimizer for updating the weights, like below:

$$V_t = V_{t-1} - lr * \left(\sum_{i=0}^N grad_i \right)$$

We can implement this in Pytorch by just not calling the optimizer.step() in every training step, simply we can set the accumulation step and call the .step() function in each accumulation step only. Here is the code example.



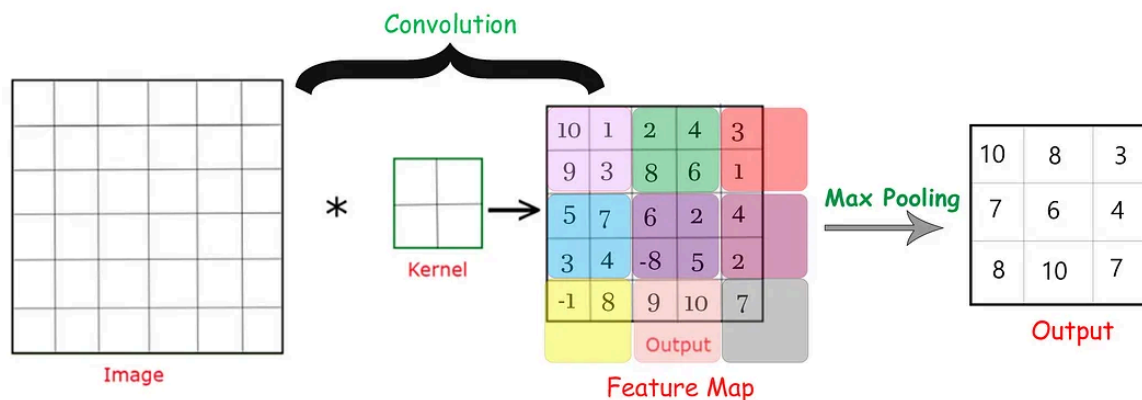
```
1 # Backward pass
2 loss.backward()
3
4 # Check if it's time for a weight update
5 if (i + 1) % accumulation_steps == 0:
6     optimizer.step()
7
8 # Zero out gradients
9 optimizer.zero_grad()
```

Q2- [Question2](#) link to answer.

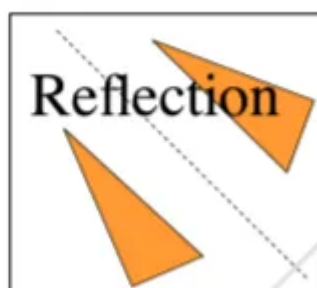
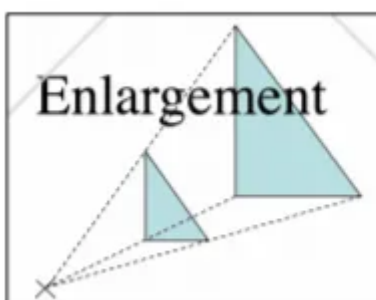
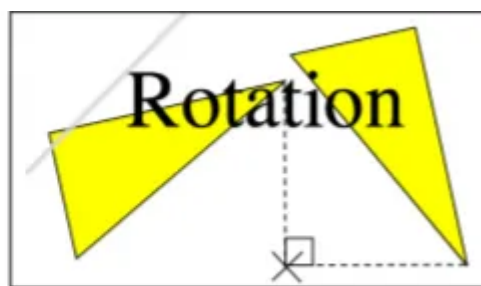
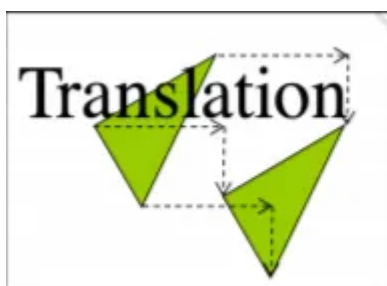
Q3-

The pooling layers are used in CNNs for several reasons. First where the size of the images are very large and need to be downsized. But there is not only downsizing, another

important job for pooling layers is to keep important features from the image pixels. Let's see how everything is happening.



The above image shows the process of one convolution layer with kernel 2*2, and max pooling layer with size 2*2 applying on the featuremap. After performing the convolution step on input, we have the feature map with some pixel values. In this case the input size is 6*6, (which is very small, often the image sizes are more than 700*700). We performed the convolution on the input and the output size now is 5*5. ($n-k+1$ the formula of the output dimensions of convolution with stride = 1). The point is that the dimension of the input image is reduced by only one pixel in each axis! Since the computational cost of the convolution is high and we can't use several convolution layers to extract features and downsize the images, we use *pooling layers* instead. For example with pooling size 2*2, we roll on all dimensions of the image with a stride equal to the pooling axis, then aggregate it and make output. In case of *max pooling* we take the max of the pixels. In addition to downsizing the image, we extract the important features of each area of image by choosing the max value of that area. There are other types of pooling layers like *average pooling* which means we take the mean value of the area instead of the max, and it holds an overall information of that area in the output instead of only important features, but we mainly use max pooling since we need important features. Another benefit of pooling layers is removing *invariances* like shift, rotation and scale.



By keeping only important pixels in an area, we can achieve this feature.

But we must be careful when using pooling layers!

Since we only keep the max pixels for the next layers, the output of max pooling may cause the over-smoothing, which means we lost a lot of information of previous layers which may cause the *Error* and poor result. Due to this, we don't use these layers frequently in the network and also choosing the size and stride of the pooling layers matter!

Q4-

The concept of transfer learning in neural networks is the idea of using a pre-trained model which was trained on some prior related task, for a new task. Intuitively it's like to change a part of your brain with another piece of someone else's brain! But how is it going to help us with the concept of CNN?

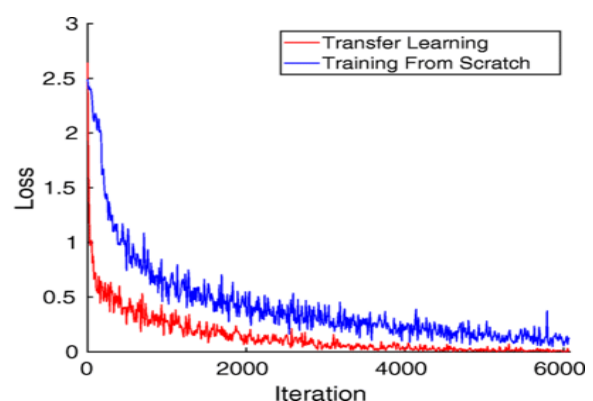
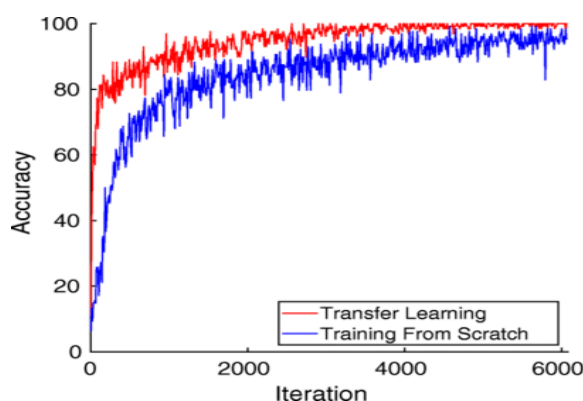
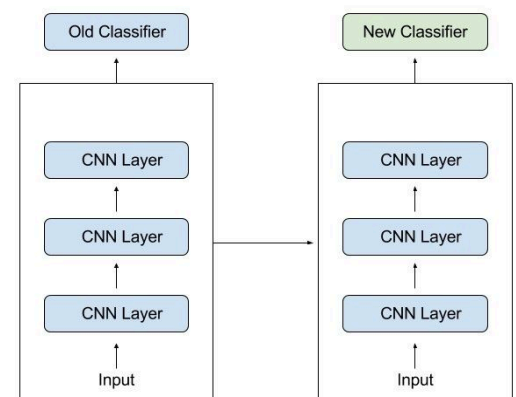
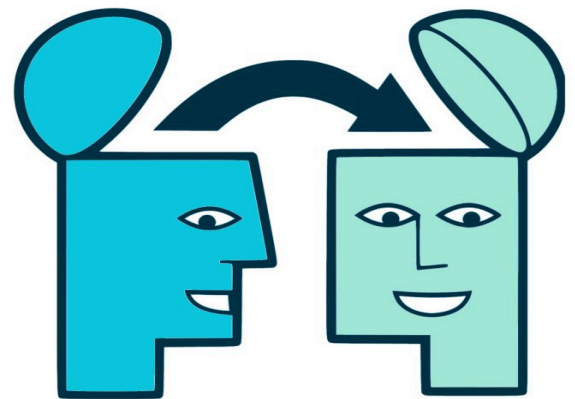
In the field of Computer vision, due to working with image data, the training data is often really huge and also needs to perform more computational tasks to extract features. So training new models with high performances needs lots of requirements like more GPU and CPU power, more training data, and also a lot of time for training. So it may not efficient to always train a model from scratch.

Fortunately, some tasks in the vision are related and really close to one another, for example you have a model trained to detect the cat in the image. So you can use this model to detect another similar object like a dog with little training data and epochs. As we can see in the below

image, we can use the same CNN layers to overcome two different classification tasks. The main idea is changing the *Decoder* part of the model and training it with some training data to achieve the goal! The main question is, “Is transfer learning always a better option?”

The questions is No, this is not always the best solution but in the most of the times(98%) using an strong and deep pretrained model is always preferred instead of training a new model from scratch, the time and computational efficiency of using a pre-trained model makes the bottom of the scale heavier!

As an example of using transfer learning in CNNs, we can mention some famous architectures like ResNet, VGG, InceptionNet, etc. the ResNet model was trained on ImageNet dataset, and used for image classification tasks, and its widely used in transfer learning for image classification.



Q5-

- True. due to the word 'Verification' the task is a binary decision in which an image corresponding to a specific individual person, so we only compare the image with that persons image, but in case of 'Recognition', we are trying to find a person information in a certain database, so we compare it with other images in database.
- False. The model could barely overfit in this dataset due it can not generalize on the task, the best ratio between number of images per person is having tens or hundreds of images per person.
- True. In earlier layers in CNNs, the model captures the base features of the image like lines and simple shapes, and by moving forward to the deep, the feature maps become more generalized and close to the input image. So it's more likely for a neuron in layer 4 to activate with a cat image than a neuron in layer 1.