1. Read the dataset

- The CIFAR10 data can be easily accessible from torchvision library. Torchvision provides many built-in datasets with the module named *'torchvision.datasets'*. Therefore, to get access to CIFAR10 dataset, we can use *'torchvision.datasets.CIFAR10()'*[1]:
    - The dataset contains both training and test set: when argument 'train' is True, you can have access to training set, otherwise, you would get access to test one
    - The number of samples of training set is 50k, while that of test one is 10k data
    - There are 10 classes in total for classification
    - The module *'torchvision.datasets.CIFAR10()'* also allows us to incorporate data transformation via argument 'transform' with *'torchvision.transforms'* module, which would make it convenient for data conversion to tensor (applicable for pytorch) and data normalization
- For train set, we can split it into training and validation set by using *'torch.utils.data.random_split()'*:
    - The splitting is necessary to validate the performance when training a model
    - Random seed should be fixed to get consistent results when performing model tuning
- After we access to training, validation and test set, then we can load/read data with *'torch.utils.data.DataLoader()'* primitive, which allows us to set desired batch size and to shuffle the data

2. Create the model

- The model follows the required architecture with a backbone which has n number of blocks. Each block is consisted of k number of convolutional layers which parallelly receives the input and produce the outputs
- Structure of each block includes:
    a. The **k** convolutional layers **parallelly** receive the input, not sequentially
    b. The output of each convolutional layers is normalized with *'torch.nn.BatchNorm2d()'*, then sent to a *ReLU()* activation function
    c. The output from **(b)** is combined together by a linear equation (or Multilayer Perceptron – MLP), where the weight 'a' is computed by the formula:
    a = g(SpatialAveragePool(X)W)
        i. Spatial Average Pool function used here is *'torch.nn.AdaptiveAvgPool2d()'*
        ii. For g (activation function), the *Sigmoid()* is chosen
        iii. The weight 'a' is calculated by an MLP of 3 linear layers, with the number of final outputs is equal to k
    d. The max pooling layer *'torch.nn.MaxPool2d()'* at the end to reduce the spatial dimension of the data
- Structure of the backbone:
    - Contain n number of blocks, each of which use the output of the previous block as an input
    - The output from the final block is sent to a classifier, which contains:
        - First: a step to compute a mean feature with Spatial Average Pool (*'torch.nn.AdaptiveAvgPool2d()'*)
        - Second: an MLP of 3 layers, which contains a *ReLU()* activation and a Dropout layer (*'torch.nn.Dropout()'*) between every 2 layers
            - The output of the classifier should be equal to the number of classes, which is 10
    - Also, a function to initialize the weights for each layer within each blocks of a backbone is needed
        - The *'torch.nn.init.xavier_normal_()'* is chosen

3. Create the loss and optimizer

- The loss function chosen in this case is Cross Entropy Loss ('torch.nn.CrossEntropyLoss()')
- The selected optimizer is Stochastic Gradient Descent ('torch.optim.SGD'):
    - Learning rate: initially go with 0.1
    - Weight decay: initially go with 0.0001

4. Write training script to train the model

[1] *'torch.datasets.CIFAR', related load/read codes and how to visual images are taken reference from this document:* *https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html*

- First, build a function to train a model in one epoch, aiming to minimize the loss (Cross Entropy Loss) with the chosen optimizer
    - This could be referred to the 'my_utils' material provided
- Second, build a function to train a model across all epochs
    - This could be referred to the 'my_utils' material provided
    - We can add some modification to return/print out the training loss, training and validation accuracy after done training each epoch
- Then, start training:
    - Choose batch size: initially go with 128
    - Initialize the model:
        - Decide number of blocks within the backbone: initially go with 5 blocks
        - For each block, choose number of convolutional layers and number of output channels: block 1 – (2, 64), block 2 – (3, 128), block 3 – (4, 256), block 4 – (5, 512), block 5 – (5, 512)
    - Choose number of epochs for training and run training the model
- After training, we could try to optimize the performance, since if the model is constructed and trained properly, it is almost certain that the model would face an issue of overfitting. Therefore, optimization is needed in order to overcome such a concern and improve the performance on validation/test set. Below is a summary of different experiments:
    - Learning rate

Learning rate	**Impact on accuracy**	**Other impacts**
0.03	positive impact on validation accuracy	Slow to converge
0.1	positive impact on validation accuracy	Reasonable time to converge
0.2	No improvement seen	Cannot converge

    - Add batch normalization: validation accuracy increases to ~75% - 80%
    - Add regularization

Weight decay	**Impact on accuracy**	**Other impacts**
0.01	No improvement seen	Cannot converge
0.001	Reduce training and validation accuracy	Take some time to converge
0.0001	Good impact on validation accuracy	Reasonable time to converge

    - Add dropout

Dropout rate	**Impact on accuracy**	**Other impacts**
0.2 – 0.3	Seem to be good for overfitting but not too obvious	
0.1	No impact seen	
Otherwise	Possible cause underfitting	Very long time to converge

    - Data augmentation[2]

Augmentation method	**Impact on accuracy**	**Other impacts**
Random horizontal flip Random crop Random rotation(a)	Very positive impact on training and validation accuracy, good for overfitting	Fast converge
Random horizontal flip Random crop Random rotation Gaussian blur	Good for accuracy improvement and overfitting but no big difference with (a)	Take a bit longer to converge
Random horizontal flip Random crop Random rotation Gaussian blur	Good for accuracy improvement and overfitting but no big difference with (a)	Take a bit longer to converge

[2] Data augmentation is inspired and taken reference from these sources:
https://towardsdatascience.com/a-comprehensive-guide-to-image-augmentation-using-pytorch-fb162f2444be
https://pytorch.org/vision/main/transforms.html

Random affine Random invert Random posterize Color jitter	The validation accuracy seems a bit unstable with increasing epochs, especially when accuracy > 80%	

- Batch size: affect the variation in accuracy and time/epochs to converge

Batch size	**Impact on accuracy**	**Other impacts**
64	Good for overfitting	Long time to converge
128	Good for overfitting	Reasonable training time
256	Create variability in accuracy	Short time for training

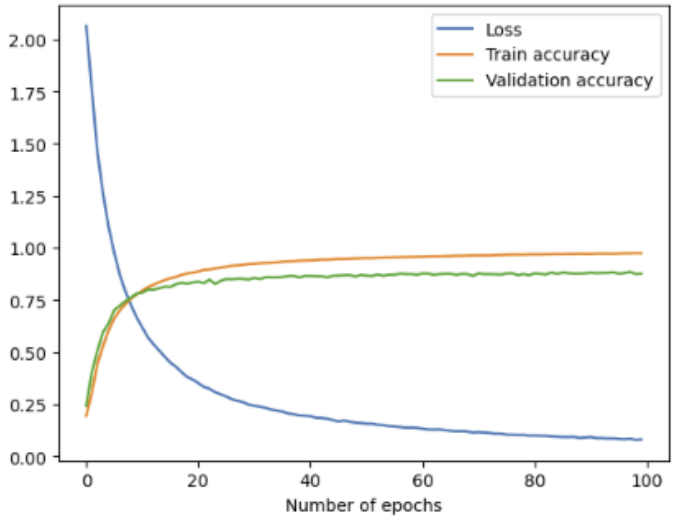
- Increase sample size with augmentation

Increase training data	**Impact on accuracy**	**Other impacts**
No, augment the 40k training data and use it for training	Good for overfitting but still struggle to reach 85% validation accuracy	Fast to converge
Yes, combine 40k normally transformed training data and 40k augmented training data	Better for overfitting, can help to reach over 85% accuracy on validation set more easily	Very fast converge

- Final model is as below:
  - Selection of the model properties:

Model properties	Selection
Batch size	128
Number of blocks	5
(Number of convolution layers, number of output channels) per block	(2, 64), (3,128), (4, 256), (5, 512), (5, 512)
Batch Normalization	Yes
Dropout rate	0.3 (30%)
Learning rate	0.1
Weight decay	0.0001
Data augmentation	Horizontal flip, crop, rotation
Number of epochs	100
Increase training samples?	Yes (from increase from 40k to 80k)

  - Loss curve and accuracy of training and validation set:



5. Final model accuracy

- Final accuracy on test set is 87.04% (> 85%)

```
In [16]:
1 test_acc = evaluate_accuracy(model, testloader)
2 print('Accuracy on test set is: ', test_acc)

Accuracy on test set is:  0.8704
```