Assignment 5– CMPUT 328
Classification on CIFAR-10 Dataset

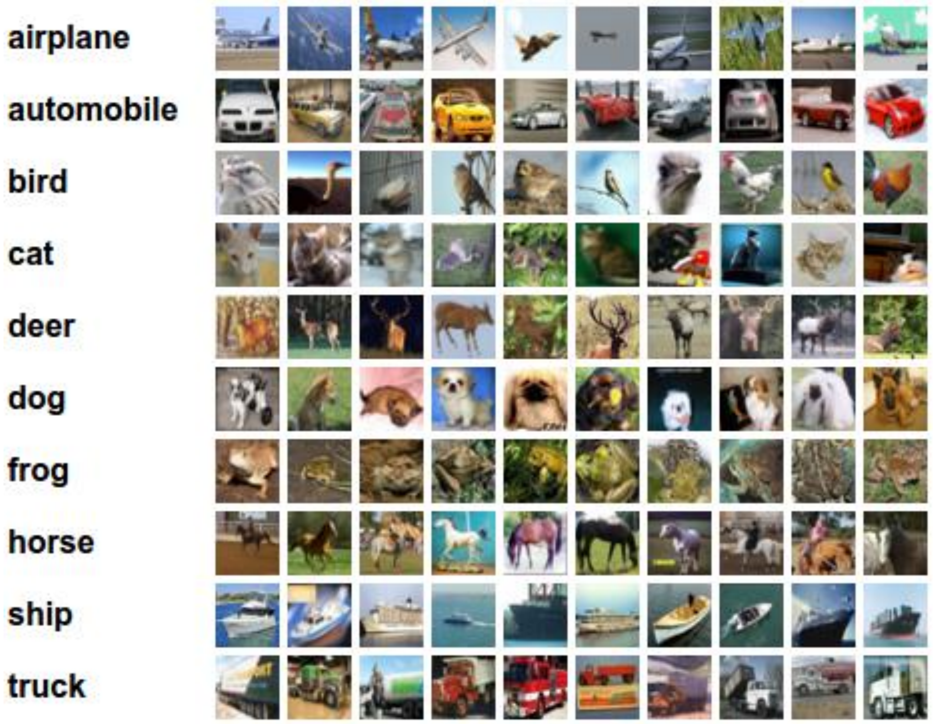
Classification on CIFAR-10 dataset [7 Marks]:

Your task in this assignment is to do classification on CIFAR-10 dataset

CIFAR-10 dataset:

Contains 32x32x3 RGB images. There are 50000 images in the train set and 10000 images in the test set. There are no validation set.

Each image in CIFAR-10 dataset belongs to one of the ten classes: airplane, automobile, bird, cat, deer, dog, frog, horse, ship, truck.



For more information, see https://www.cs.toronto.edu/~kriz/cifar.html

A. Tasks:

In this assignment, you are going to improve the Convolutional Neural Network (implemented in assignment 4) to do classification on CIFAR-10 dataset.

The network architecture in this assignment is not fixed. **You will design the network architecture yourself.**

However, there are some requirements that your network architecture must satisfy:

a) Your network must have at least 5 layers (a layer means convolution or fully connected layer) into an activation function like Relu, Softmax, Tanh or sigmoid..., (input layer, max pooling or strided convolution layer used to downsample activation map doesn't count). The number of layers in your network will be very close to the number of activation functions)

b) Must have at least 1 convolution layer
c) Must have at least 1 fully connected layer at the end
d) Must have at least 1 max pooling or strided convolution layer (i.e. Convolution with stride > 1)

e) Must use batch normalization (**will be explained in section B**)
f) Must use dropout (**will be explained in section B**)
g) Must use skip connection (**will be explained in section B**)

If your network architecture doesn't satisfy any of the above 4 requirements, you will lose mark.

In addition to the above requirement, to do well in this assignment, you may have to:

- Use He/Xavier initialization (**will be explained in section B**)
- Regularization
- Try different activation functions (relu, tanh, sigmoid, elu...)
- Try different optimizer (For example: Vanilla SGD, Adam, RMSProp, AdaDelta...)
- Try different loss function (For example: cross entropy, square error)
- Annealing the learning rate (**will be explained in section B**)

B. Explanation for the terms mentioned above:

1. Batch Normalization:

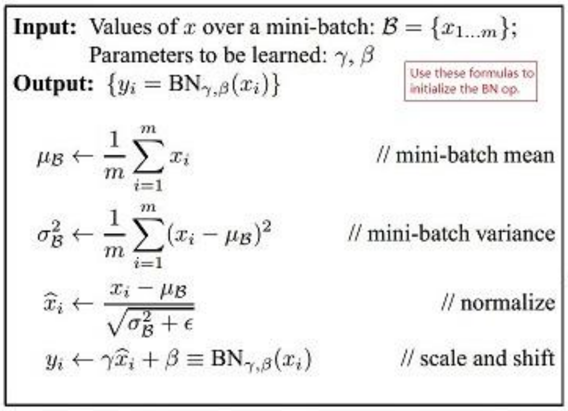
A tradition technique that is used frequently to reduce the effect of internal covariate shift in Deep Neural Network by normalizing the input into each activation function to have a zero mean and unit variance.

Batch Normalization do this by computing the mean and variance of each batch during training time, then subtract the mean from the input and divide the result by variance. A linear transformation is applied after to allow the network to restore the original signal to the activation function.

At test time, Batch Normalization use the mean and variance of the whole population set to normalize the input into activation function instead of using batch statistics. The mean and variance of the whole population are approximated by a moving mean and variance during training that is updated during training time.

More information can be found at: https://arxiv.org/abs/1502.03167 (the original paper), https://gab41.lab41.org/batch-normalization-what-the-hey-d480039a9e3b, https://wiki.tum.de/display/lfdv/Batch+Normalization

Full Batch Normalization algorithm:



Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1\ldots m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \text{BN}_{\gamma,\beta}(x_i)\}$

Use these formulas to initialize the BN op.

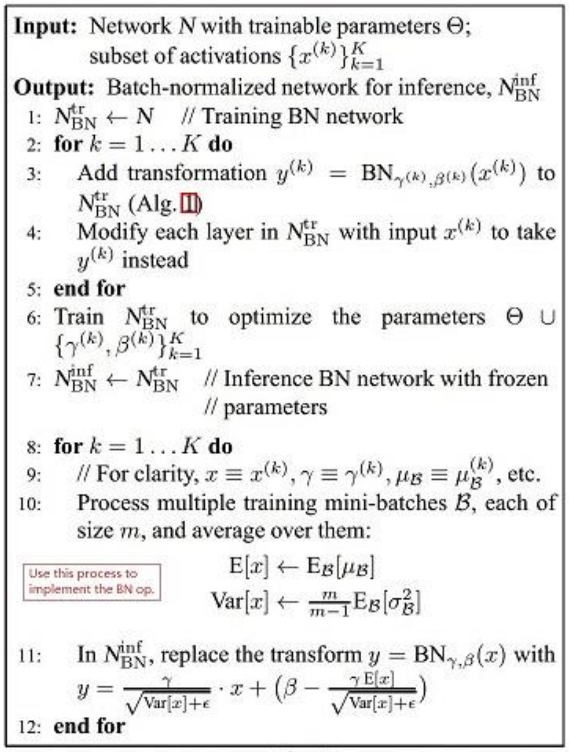
$$\mu_\mathcal{B} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma_\mathcal{B}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_\mathcal{B})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_\mathcal{B}}{\sqrt{\sigma_\mathcal{B}^2 + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \text{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

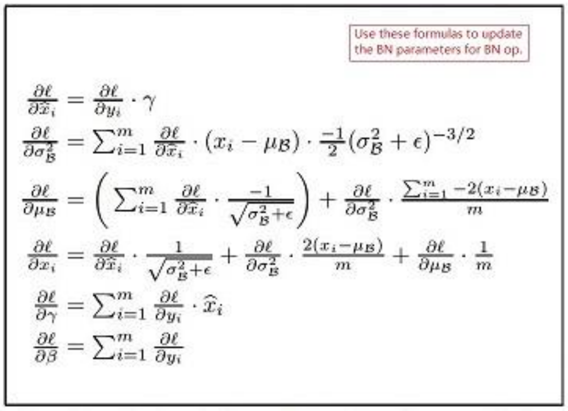
Alg. 1



```
Input: Network $N$ with trainable parameters $\Theta$;
       subset of activations $\{x^{(k)}\}_{k=1}^{K}$
Output: Batch-normalized network for inference, $N_{\text{BN}}^{\text{inf}}$
  $N_{\text{BN}}^{\text{tr}} \leftarrow N$    // Training BN network
  for $k = 1 \ldots K$ do
      Add transformation $y^{(k)} = \text{BN}_{\gamma^{(k)},\beta^{(k)}}(x^{(k)})$ to
      $N_{\text{BN}}^{\text{tr}}$ (Alg. 1)
      Modify each layer in $N_{\text{BN}}^{\text{tr}}$ with input $x^{(k)}$ to take
      $y^{(k)}$ instead
  end for
  Train $N_{\text{BN}}^{\text{tr}}$ to optimize the parameters $\Theta \cup \{\gamma^{(k)}, \beta^{(k)}\}_{k=1}^{K}$
  $N_{\text{BN}}^{\text{inf}} \leftarrow N_{\text{BN}}^{\text{tr}}$    // Inference BN network with frozen
                                  // parameters
  for $k = 1 \ldots K$ do
      // For clarity, $x \equiv x^{(k)}, \gamma \equiv \gamma^{(k)}, \mu_\mathcal{B} \equiv \mu_\mathcal{B}^{(k)}$, etc.
      Process multiple training mini-batches $\mathcal{B}$, each of
      size $m$, and average over them:
          $\text{E}[x] \leftarrow \text{E}_\mathcal{B}[\mu_\mathcal{B}]$
          $\text{Var}[x] \leftarrow \frac{m}{m-1}\text{E}_\mathcal{B}[\sigma_\mathcal{B}^2]$
      In $N_{\text{BN}}^{\text{inf}}$, replace the transform $y = \text{BN}_{\gamma,\beta}(x)$ with
      $y = \frac{\gamma}{\sqrt{\text{Var}[x]+\epsilon}} \cdot x + \left(\beta - \frac{\gamma\,\text{E}[x]}{\sqrt{\text{Var}[x]+\epsilon}}\right)$
  end for
```

Use this process to implement the BN op.

Alg. 2



Use these formulas to update the BN parameters for BN op.

$$\frac{\partial \ell}{\partial \widehat{x}_i} = \frac{\partial \ell}{\partial y_i} \cdot \gamma$$

$$\frac{\partial \ell}{\partial \sigma_\mathcal{B}^2} = \sum_{i=1}^{m} \frac{\partial \ell}{\partial \widehat{x}_i} \cdot (x_i - \mu_\mathcal{B}) \cdot \frac{-1}{2}(\sigma_\mathcal{B}^2 + \epsilon)^{-3/2}$$

$$\frac{\partial \ell}{\partial \mu_\mathcal{B}} = \left(\sum_{i=1}^{m} \frac{\partial \ell}{\partial \widehat{x}_i} \cdot \frac{-1}{\sqrt{\sigma_\mathcal{B}^2+\epsilon}}\right) + \frac{\partial \ell}{\partial \sigma_\mathcal{B}^2} \cdot \frac{\sum_{i=1}^{m} -2(x_i-\mu_\mathcal{B})}{m}$$

$$\frac{\partial \ell}{\partial x_i} = \frac{\partial \ell}{\partial \widehat{x}_i} \cdot \frac{1}{\sqrt{\sigma_\mathcal{B}^2+\epsilon}} + \frac{\partial \ell}{\partial \sigma_\mathcal{B}^2} \cdot \frac{2(x_i-\mu_\mathcal{B})}{m} + \frac{\partial \ell}{\partial \mu_\mathcal{B}} \cdot \frac{1}{m}$$

$$\frac{\partial \ell}{\partial \gamma} = \sum_{i=1}^{m} \frac{\partial \ell}{\partial y_i} \cdot \widehat{x}_i$$

$$\frac{\partial \ell}{\partial \beta} = \sum_{i=1}^{m} \frac{\partial \ell}{\partial y_i}$$

Formula set 1

Note:

- Batch Normalization should always be applied after a matrix multiplication and before an activation function.
- Batch Normalization are usually not applied for the first layer after the input layer as the statistics of the input layer can be computed easily from the dataset to do a simple normalization instead of batch normalization.
- Batch Normalization should NEVER be applied to the output layer of a network. Doing so will force the output of a network to follow a certain distribution which is usually not true.

Batch Normalization in Tensorflow:

There are many high level API functions in Tensorflow that can be used to implement Batch Normalization in Tensorflow (A simple Google search would yield plenty results). One thing to notice is that Batch Normalization training and testing phase behave differently. Therefore the high level Batch Normalization functions of Tensorflow require additional placeholder that has boolean type called “is_training” which should have value of True during training and False during testing.

2. Dropout:

Dropout is a regularization technique used to prevent overfitting in training Deep Neural Networks. Dropout drops random unit (along with their connections) from the neural network

during training. This prevents units from co-adapting too much.
Dropout has somewhat fallen out of favor since Batch Normalization has similar regularization effect. However, it almost never hurt if Dropout is used correctly in a neural network.
Dropout are usually used at the layer just before the output layer of a neural network. Dropout probability is usually 0.5 or 0.2 (correspond to dropout_kept_prob of 0.5 or 0.8)
Note that dropout also have different training and test time behaviors. In Tensorflow, during training phase, dropout kept probability is set to a value that is less than 1.0 but during testing that value must be set to 1.0
For more informiaton: https://www.tensorflow.org/get_started/mnist/pros (check Dropout section), https://www.cs.toronto.edu/~hinton/absps/JMLRdropout.pdf (original paper)

3. Skip connections:
In Deep Learning, skip connections are simply connection from an ealier layer in the network to a later layer. This allows better gradients for very deep networks during training and helps improving performancing.
More information: https://arxiv.org/abs/1512.03385,
https://stats.stackexchange.com/questions/56950/neural-network-with-skip-layer-connections

An example of a skip connection:

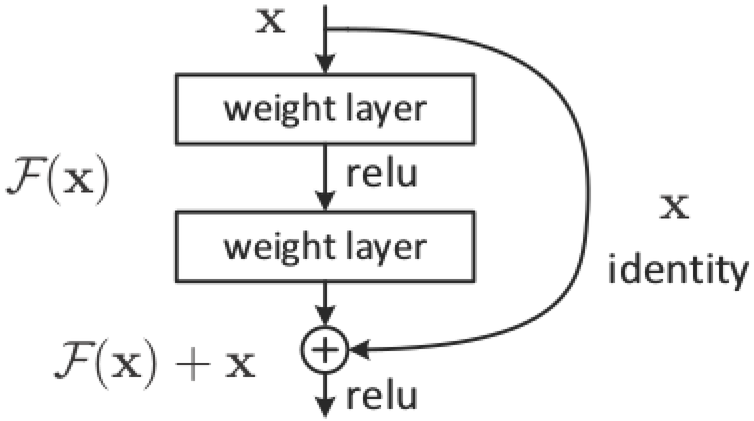

x
weight layer
$\mathcal{F}(\mathbf{x})$
relu
weight layer
x
identity
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
relu

Figure 2. Residual learning: a building block.

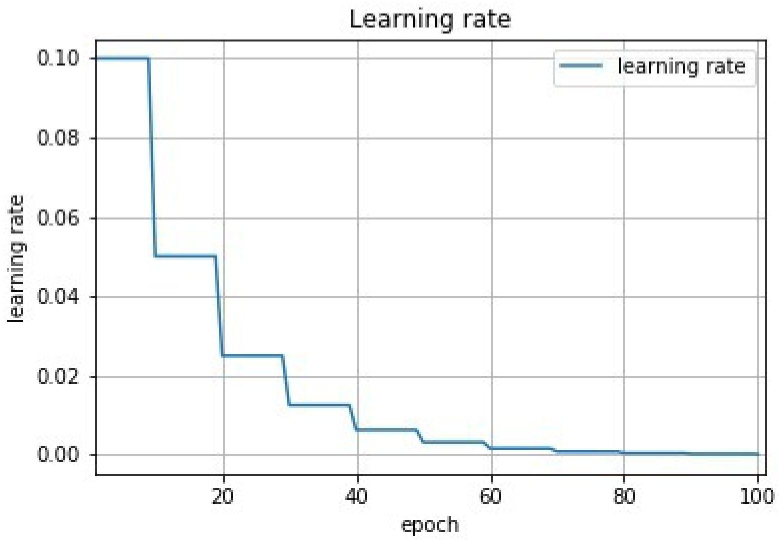
In Tensorflow, a simple implementation of skip connection is just adding the input into an activation function by the output of an earlier layer.

4. Use He/Xavier initialization:
See: http://andyljones.tumblr.com/post/110998971763/an-explanation-of-xavier-initialization
In Tensorflow, you can access this kind of initialization by using
https://www.tensorflow.org/api_docs/python/tf/contrib/layers/xavier_initializer

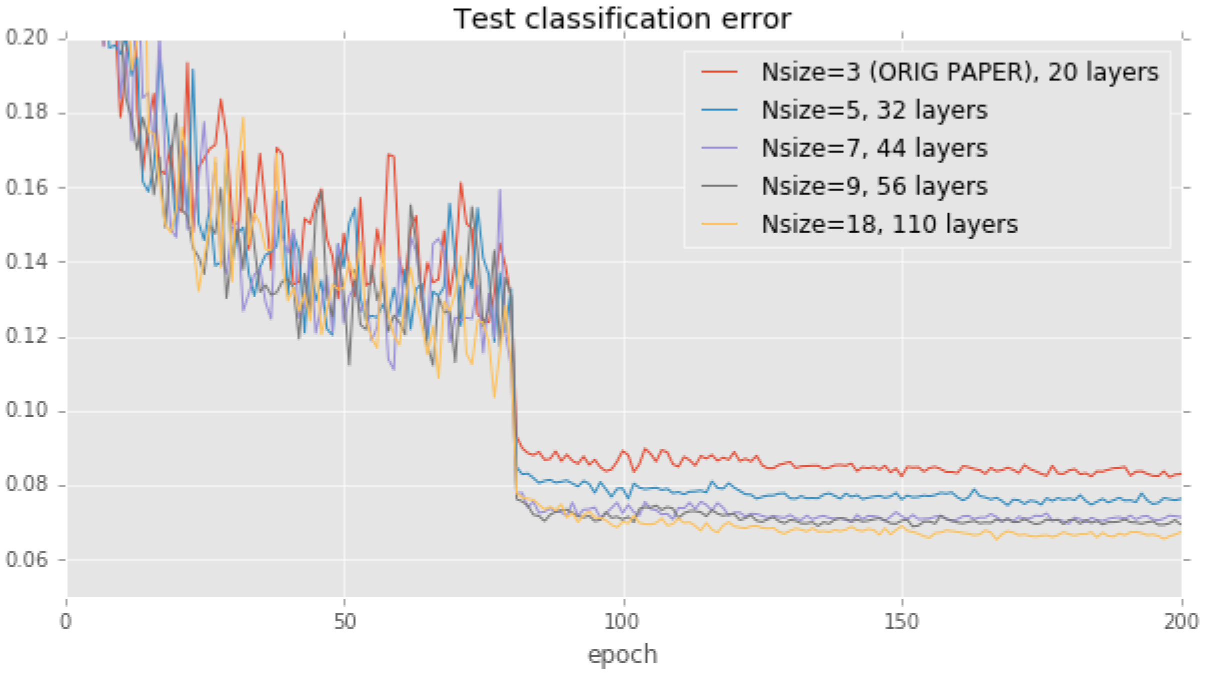
5. Learning Rate Annealing:
During training deep network, it's a good idea to drop your learning rate overtime. Most optimizer assume the learning rate decreasing overtime as a necessary condition for convergence. Decreasing the learning rate overtime allow your network to search on a finer scale of the parameters space and allow improvement of performance.

How much to drop the learning rate and when to drop is important. Drop the learning rate too soon or too much and your network will be very slow to train. A common scheme is to halve the learning rate after a few epochs similar to a staircase function. Another scheme is progressively reduce the learning rate after every iteration.

An example of a staircase learning rate:



An example on effect of learning rate annealing (the part where test error drop dramatically is the part where the learning rate is dropped significantly):



More information: http://cs231n.github.io/neural-networks-3/#anneal
Tensorflow learning rate annealing documentation:
https://www.tensorflow.org/versions/r0.12/api_docs/python/train/decaying_the_learning_rate

C. What to do:

You are given some code. There are 4 files in the code:

a) **cifar10**: This part provides utilities on Cifar10 dataset via the class Cifar10. You don't have to modify this code. Some example usage:

```
cifar10_train = Cifar10(batch_size=100, one_hot=True, test=False, shuffle=True)   # Load Cifar10 train set with batch size 100 and one hot label.
cifar10_test = Cifar10(batch_size=100, one_hot=False, test=True, shuffle=False)  # Load Cifar10 test set
cifar10_test_images, cifar10_test_labels = cifar10_test._images, cifar10_test._labels  # Get all images and labels of the test set.
batch_x, batch_y = cifar10_train.get_next_batch()  # get the next batch of Cifar10 train set
```

b) main: Part used as main entry point for the program. You don't need to modify this file. However, there is one thing you must notice inside this function. You can set TRAIN = False at the beginning to skip the training phase. This is useful when you saved your model and just want to do testing.

c) ops: Part contains some utilities function for your network architecture creation. For example, there is already an example batch normalization wrapper utility function there which you are recommended to use. Here you can also implement functions that create a convolution or fully connected layer. These functions should allocate all variables and compute the result of the convolution/fully connected layer automatically from the input for you.

d) net.py: You will need to implement your network architecture, training code and testing code here in this file.

def net(input, is_training)

Function that return the output of your network based on the input. is_training is a boolean placeholder that should be set to True in training phase and False in testing phase.

The architecture of the network you design inside this function must satisfy the 4 requirements mentioned in section A. For each requirement that is not satisfied, you lose 10% of your mark on this assignment.

def train()

Write your training code here. This should be similar to Assignment 1. However, in this assignment, **you should and probably have to add code to save your model after training in order to satisfy the time constraint in this assignment.** Training the model in this assignment will be quite slow. If you do both training and testing at once, your program will exceed the time limit. The best way to do this is: train your model, save your model. In test time, load your model and predict labels.

Note: This function will NOT be called during marking time!

def test()

Your testing code to generate labels for test images using your trained model should go here. During marking time the **train()** function will NOT be called. Therefore, if you don't want to split your training and testing, you must put your training code here.

COLLABORATION POLICY: This must be your own work. Do not share or look at the code of other people (whether they are inside or outside the class). Do not search for or copy the code from the Internet. You can talk to others that are in the class about solution ideas (but not so detailed that you are verbally sharing or hearing about or seeing the code). You must cite whom you talked to in the comments of your programs.

D. SUBMISSION: You need to submit one file: Submit 2 files: **net.py** and **ops.py** to eclass.
If you have a saved model that will be loaded during test time, submit that model too (IMPORTANT!).

DUE DATE: The due date is Friday, October 19by 11:59 pm. The assignment is to be submitted online on eclass. For late submissions' rules please, check the course information on eclass

E. MARKING:
2 Marks Describe your program for TAs (Monday, Oct 22 – Tuesday, Oct 23 – Thursday, Oct 15) TAs can select five random questions based on your code, results, and algorithms. The time to present will be in your lab section in the week when this lab is due. Note that you must present this part to your TA in your designated lab section. *You will not get any mark for this part if you don't present in your own lab section.*

5 Marks Your final score depends on the correct implementation of algorithms and it must work on CIFAR10 datasets.

1. Base mark:
Your algorithm will get mark based on its accuracy on Cifar-10 dataset: 5 Marks

- <= 0.60: 0
- >= 0.80: 5
- 0.60 <= accuracy < 0.80: scaled

2. Time constraint:
Your whole program (testing part) should not run for more than **200s (With Google colab on GPU)**
3. Architecture requirements:
As mentioned above, your network architecture must satisfy these requirements:

a) Your network must have at least 5 layers (a layer means convolution or fully connected layer) into an activation function like relu, softmax, tanh or sigmoid..., (input layer, output layer, max pooling or strided convolution layer used to downsample activation map doesn't count. The number of layers in your network will be very close to the number of activation functions)
b) Must have at least 1 convolution layer
c) Must have at least 1 fully connected layer at the end
d) Must have at least 1 max pooling or strided convolution layer (i.e. Convolution with stride > 1)
h) Must use batch normalization
i) Must use dropout
j) Must use skip connection

For every requirement that is not satisfied, you lose 10% of your mark.