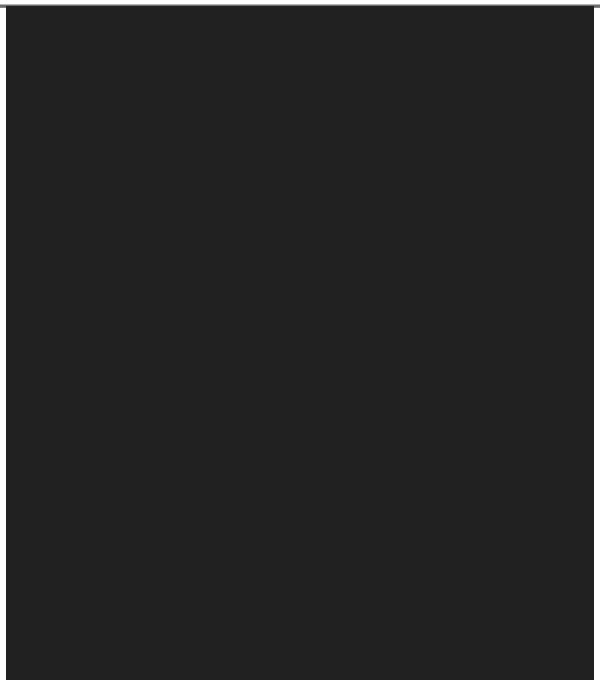


DL-Ops

Assignment 1 - ResNet and Layer-wise pretraining

Debonil Ghosh
Roll No: M21AIE225
Executive MTech
Artificial Intelligence
Indian Institute of Technology, Jodhpur

Problem Statement:

Question 1 [80 marks]
Train ResNet18 on Tiny ImageNet dataset (download from here) with X as the optimizer for
classification task. Plot curves for training loss, training accuracy, validation accuracy and report
the final test accuracy. Here consider accuracy as top-5 accuracy.
1. Use CrossEntropy as the final classification loss function [10 marks]
2. Use Triplet Loss with hard mining as the final classification loss function [30 marks]
3. Use Central Loss as the final classification loss function [40 marks]
Compare the performance of different models and analyze the results in the report.

Note - The code for ResNet18 architecture and the loss functions needs to be implemented
from scratch. Directly importing from the library is not allowed and 0 marks will be awarded for
that.
X = Adam, if last digit of your roll no. is odd
X = SGD, if last digit of your roll no. is even

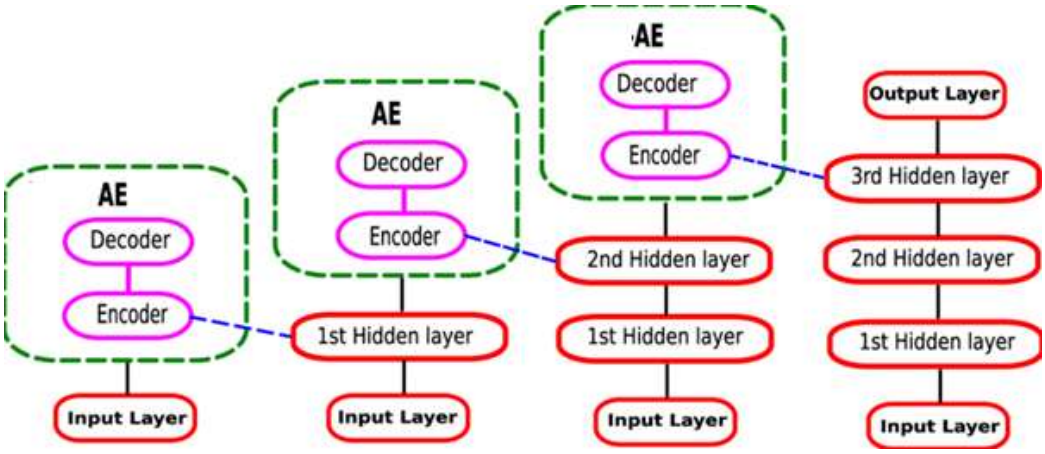
Question 2 [20 marks]
Implement a multi-layered classifier where weights of each layer is calculated greedily using
layer-wise pretraining with the help of auto-encoders on STL-10 dataset. Train a classifier having
X structure (excluding input and output layers) for classification task on the test set.
1. Report the classification accuracy on the test set and plot loss curves on the training and
evaluation set.
2. Report the class-wise accuracy of each class.
3. Plot t-sne for this model (use embeddings from layer X[3]) . Use the first 500 images of
each class from the test dataset for this visualization.
X = [1024,1200,728,512,128], if last digit of your roll no. is odd
X = [1024,1000,500,256, 128,64], if last digit of your roll no. is even
Reference - Slides [page number 11 to 14]

Solution of Question 2:

Greedy layer-wise pretraining is a technique that was commonly used in the early days of deep learning to improve the training of deep neural networks. The idea behind this technique is to pretrain each layer of a deep neural network using unsupervised learning, before fine-tuning the entire network using supervised learning.

The basic idea is to stack multiple layers of neural networks and train each layer greedily to learn a better representation of the input data. The first layer is trained using an unsupervised learning algorithm, to learn a set of features that capture low-level patterns in the input data. Once the first layer is trained, the second layer is added, and the first layer's output is used as the input to the second layer. This process is repeated for each subsequent layer until the entire network is trained.

By pretraining each layer in an unsupervised manner, the weights in each layer are initialized to values that are closer to the optimal values. This can help **to avoid the problem of vanishing gradients** that can occur in deep networks when training from scratch.



After pretraining, the entire network is fine-tuned using supervised learning to optimize the weights for the specific task at hand. This can be done using backpropagation, where the gradients of a loss function with respect to the weights are computed and used to update the weights in each layer.

While greedy layer-wise pretraining was a popular technique in the past, it has become less common in recent years as other techniques, such as batch normalization and skip connections, have been developed to improve the training of deep neural networks. However, it can still be useful in certain cases, such as when working with limited amounts of labelled data.

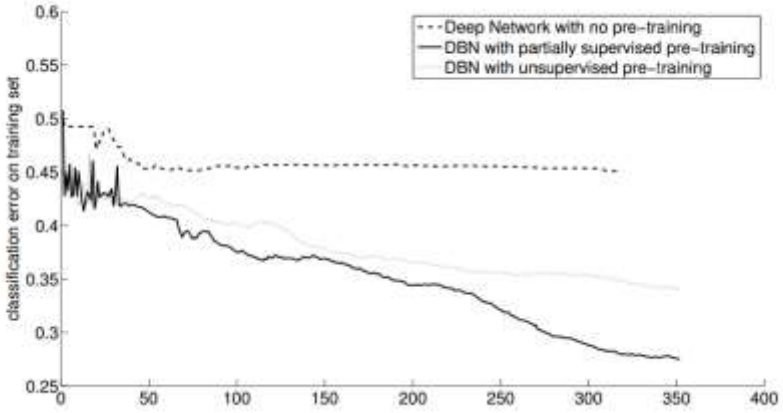


Image source : https://www.cs.cmu.edu/~bhiksha/courses/deeplearning/Fall.2016/notes/hefny_Greedy.pdf

Given Dataset:

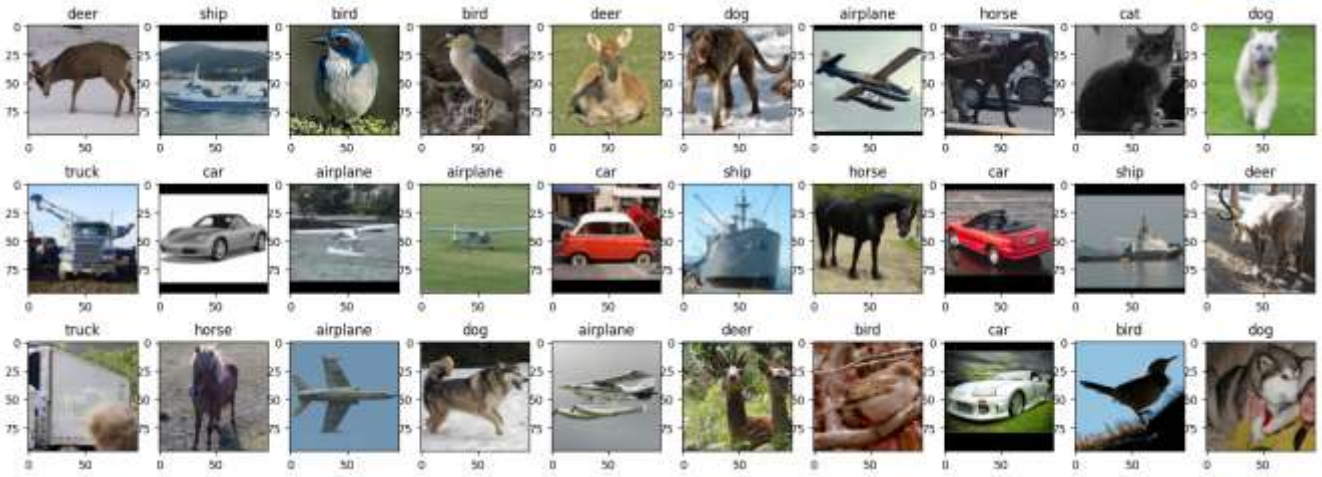
Dataset Name: STL 10

Training Set: 5000 (500×10 pre-defined folds) Colour (3 channels) 96×96 Images

Testing Set: 8000 (800×10 pre-defined folds) Colour (3 channels) 96×96 Images

Classes: 10 classes: airplane, bird, car, cat, deer, dog, horse, monkey, ship, truck

Data Samples:



Model Architecture

For Auto encoder part

```
autoencoder(
  (encoder): Sequential(
    (0): Linear(in_features=9216, out_features=1024, bias=True)
    (1): ReLU()
    (2): Linear(in_features=1024, out_features=1200, bias=True)
    (3): ReLU()
    (4): Linear(in_features=1200, out_features=728, bias=True)
    (5): ReLU()
    (6): Linear(in_features=728, out_features=512, bias=True)
    (7): ReLU()
    (8): Linear(in_features=512, out_features=128, bias=True)
  )
  (decoder): Sequential(
    (0): Linear(in_features=128, out_features=512, bias=True)
    (1): ReLU()
    (2): Linear(in_features=512, out_features=728, bias=True)
    (3): ReLU()
    (4): Linear(in_features=728, out_features=1200, bias=True)
    (5): ReLU()
    (6): Linear(in_features=1200, out_features=1024, bias=True)
    (7): ReLU()
    (8): Linear(in_features=1024, out_features=9216, bias=True)
    (9): Sigmoid()
  )
)
```

For Classifier part

```
image_classifier(
  (encoder): Sequential(
    (0): Linear(in_features=9216, out_features=1024, bias=True)
    (1): ReLU()
    (2): Linear(in_features=1024, out_features=1200, bias=True)
    (3): ReLU()
    (4): Linear(in_features=1200, out_features=728, bias=True)
    (5): ReLU()
    (6): Linear(in_features=728, out_features=512, bias=True)
    (7): ReLU()
    (8): Linear(in_features=512, out_features=128, bias=True)
  )
  (fc): Sequential(
    (0): Linear(in_features=128, out_features=10, bias=True)
    (1): ReLU()
  )
)
```

Hyper Parameters for Auto Encoder:

- *Loss Function: MSE Loss*
- *Optimiser: Adam*
- *Learning Rate: 0.001*
- *Number of epochs: 50*

Hyper Parameters for Classifier:

- Loss Function: Cross Entropy Loss
- Optimiser: Adam
- Learning Rate: 0.001
- Number of epochs: 20

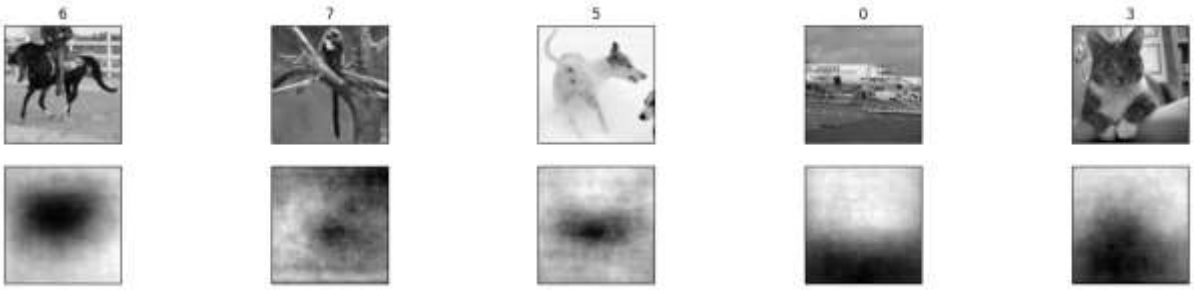
Auto Encoder Training

Log:

Epoch: 1 Training Loss: 5.791006, Test Loss: 52.439228, Training acc: 0.000000, Test acc: 0.000000,
Epoch: 2 Training Loss: 5.040079, Test Loss: 50.626852, Training acc: 0.000000, Test acc: 0.000000,
Epoch: 3 Training Loss: 4.983098, Test Loss: 50.500002, Training acc: 0.000000, Test acc: 0.000000,
Epoch: 4 Training Loss: 4.864150, Test Loss: 44.460203, Training acc: 0.000000, Test acc: 0.000000,
Epoch: 5 Training Loss: 4.343365, Test Loss: 43.248069, Training acc: 0.000000, Test acc: 0.000000,
Epoch: 6 Training Loss: 4.266812, Test Loss: 42.935669, Training acc: 0.000000, Test acc: 0.000000,
Epoch: 7 Training Loss: 4.258879, Test Loss: 42.887688, Training acc: 0.000000, Test acc: 0.000000,
Epoch: 8 Training Loss: 4.262150, Test Loss: 42.721353, Training acc: 0.000000, Test acc: 0.000000,
Epoch: 9 Training Loss: 4.228788, Test Loss: 41.894421, Training acc: 0.000000, Test acc: 0.000000,
Epoch: 10 Training Loss: 4.055733, Test Loss: 38.596042, Training acc: 0.000000, Test acc: 0.000000,
Epoch: 11 Training Loss: 3.763021, Test Loss: 36.924113, Training acc: 0.000000, Test acc: 0.000000,
Epoch: 12 Training Loss: 3.691430, Test Loss: 36.973801, Training acc: 0.000000, Test acc: 0.000000,
Epoch: 13 Training Loss: 3.686795, Test Loss: 36.533810, Training acc: 0.000000, Test acc: 0.000000,
Epoch: 14 Training Loss: 3.673056, Test Loss: 36.641978, Training acc: 0.000000, Test acc: 0.000000,
Epoch: 15 Training Loss: 3.655363, Test Loss: 36.534924, Training acc: 0.000000, Test acc: 0.000000,
Epoch: 16 Training Loss: 3.655831, Test Loss: 36.520660, Training acc: 0.000000, Test acc: 0.000000,
Epoch: 17 Training Loss: 3.655898, Test Loss: 36.369693, Training acc: 0.000000, Test acc: 0.000000,
Epoch: 18 Training Loss: 3.633159, Test Loss: 36.328144, Training acc: 0.000000, Test acc: 0.000000,

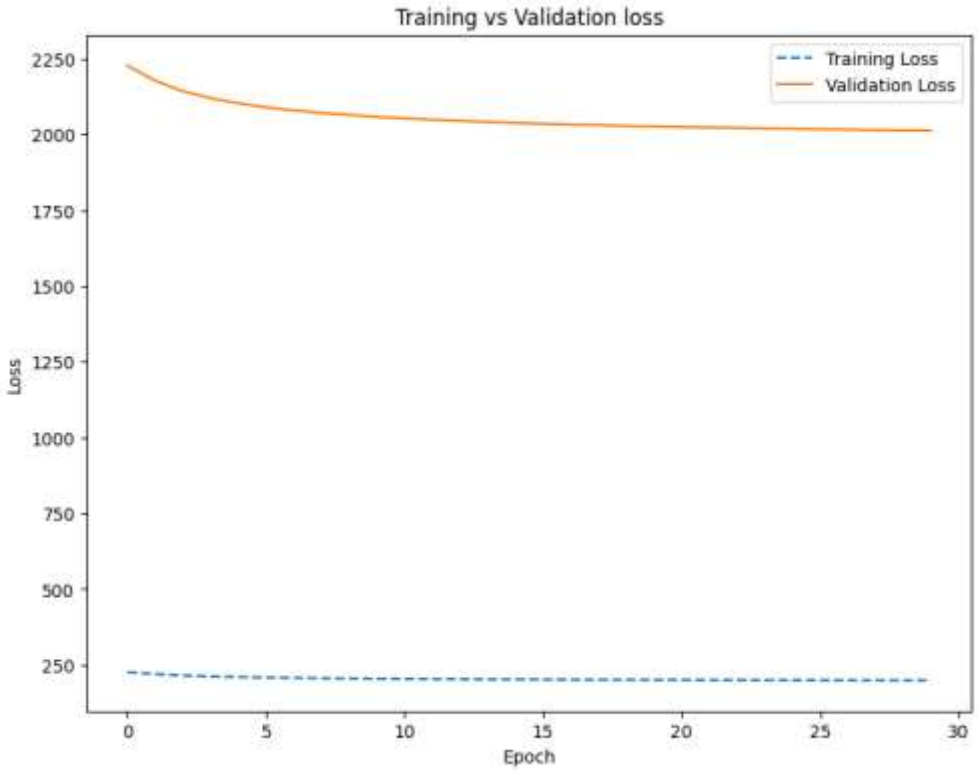
Epoch: 19 Training Loss: 3.625808, Test Loss: 36.168855, Training acc: 0.000000, Test acc: 0.000000,
Epoch: 20 Training Loss: 3.625555, Test Loss: 36.134638, Training acc: 0.000000, Test acc: 0.000000,
Epoch: 21 Training Loss: 3.617460, Test Loss: 36.310866, Training acc: 0.000000, Test acc: 0.000000,
Epoch: 22 Training Loss: 3.616085, Test Loss: 36.016114, Training acc: 0.000000, Test acc: 0.000000,
Epoch: 23 Training Loss: 3.574844, Test Loss: 35.385124, Training acc: 0.000000, Test acc: 0.000000,
Epoch: 24 Training Loss: 3.577289, Test Loss: 35.084840, Training acc: 0.000000, Test acc: 0.000000,

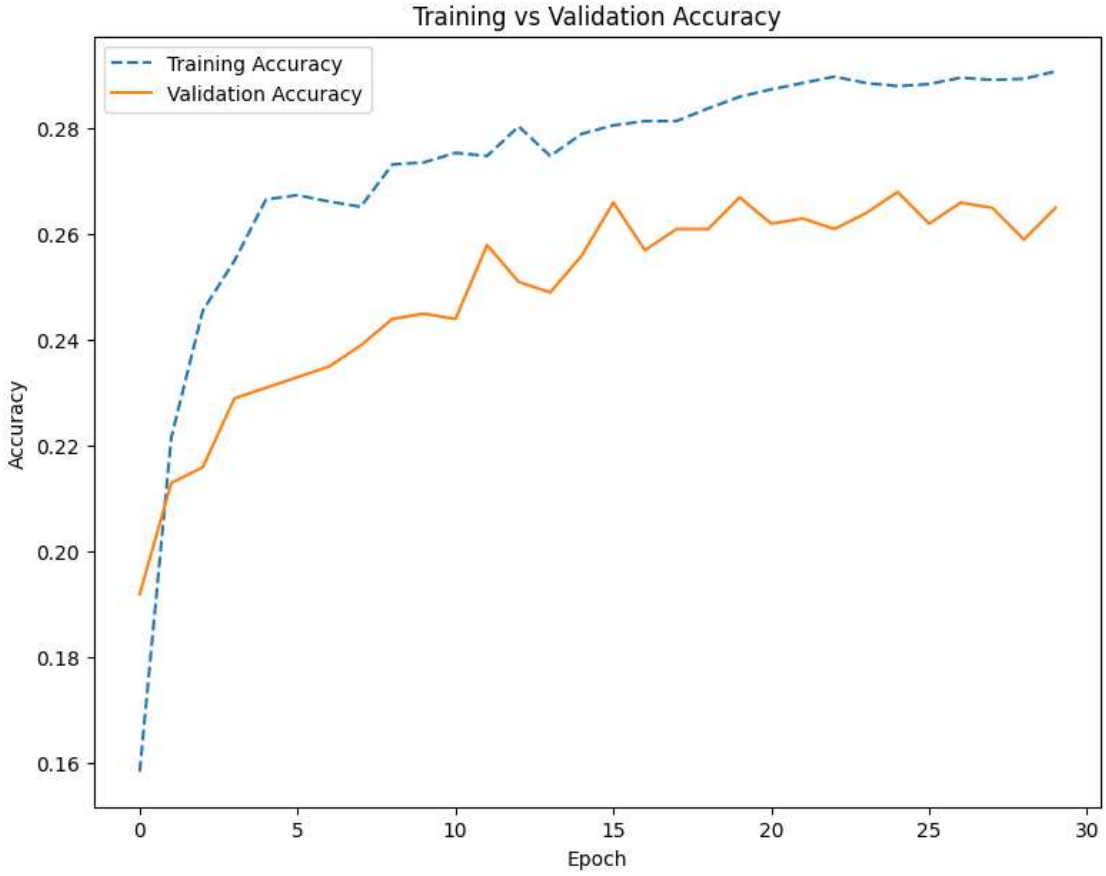
AE Sample prediction after training



Training Log of Classifier:

Epoch: 1 Training Loss: 226.134165, Test Loss: 2227.391720, Training acc: 0.158400, Test acc: 0.192000,
Epoch: 2 Training Loss: 219.504177, Test Loss: 2177.054644, Training acc: 0.221600, Test acc: 0.213000,
Epoch: 3 Training Loss: 214.973242, Test Loss: 2142.748594, Training acc: 0.245600, Test acc: 0.216000,
Epoch: 4 Training Loss: 211.981044, Test Loss: 2119.860172, Training acc: 0.255000, Test acc: 0.229000,
Epoch: 5 Training Loss: 209.860574, Test Loss: 2102.726221, Training acc: 0.266600, Test acc: 0.231000,
Epoch: 6 Training Loss: 208.242755, Test Loss: 2089.500904, Training acc: 0.267400, Test acc: 0.233000,
Epoch: 7 Training Loss: 207.009069, Test Loss: 2079.362869, Training acc: 0.266200, Test acc: 0.235000,
Epoch: 8 Training Loss: 205.961483, Test Loss: 2070.981264, Training acc: 0.265200, Test acc: 0.239000,
Epoch: 9 Training Loss: 205.147539, Test Loss: 2064.410448, Training acc: 0.273200, Test acc: 0.244000,
Epoch: 10 Training Loss: 204.397602, Test Loss: 2057.743311, Training acc: 0.273600, Test acc: 0.245000,
Epoch: 11 Training Loss: 203.774241, Test Loss: 2053.155184, Training acc: 0.275400, Test acc: 0.244000,
Epoch: 12 Training Loss: 203.218104, Test Loss: 2049.008846, Training acc: 0.274800, Test acc: 0.258000,
Epoch: 13 Training Loss: 202.769477, Test Loss: 2044.994593, Training acc: 0.280400, Test acc: 0.251000,
Epoch: 14 Training Loss: 202.340623, Test Loss: 2041.218996, Training acc: 0.274800, Test acc: 0.249000,
Epoch: 15 Training Loss: 201.971613, Test Loss: 2037.982702, Training acc: 0.279000, Test acc: 0.256000,
Epoch: 16 Training Loss: 201.655735, Test Loss: 2035.289049, Training acc: 0.280600, Test acc: 0.266000,
Epoch: 17 Training Loss: 201.339532, Test Loss: 2032.493591, Training acc: 0.281400, Test acc: 0.257000,
Epoch: 18 Training Loss: 201.044175, Test Loss: 2031.008005, Training acc: 0.281400, Test acc: 0.261000,
Epoch: 19 Training Loss: 200.779714, Test Loss: 2028.291225, Training acc: 0.283800, Test acc: 0.261000,
Epoch: 20 Training Loss: 200.546949, Test Loss: 2026.812077, Training acc: 0.286000, Test acc: 0.267000,
Epoch: 21 Training Loss: 200.314089, Test Loss: 2024.442196, Training acc: 0.287400, Test acc: 0.262000,
Epoch: 22 Training Loss: 200.107319, Test Loss: 2023.317814, Training acc: 0.288600, Test acc: 0.263000,
Epoch: 23 Training Loss: 199.906536, Test Loss: 2021.675825, Training acc: 0.289800, Test acc: 0.261000,
Epoch: 24 Training Loss: 199.729853, Test Loss: 2020.110607, Training acc: 0.288600, Test acc: 0.264000,
Epoch: 25 Training Loss: 199.580452, Test Loss: 2018.836737, Training acc: 0.288000, Test acc: 0.268000,
Epoch: 26 Training Loss: 199.432682, Test Loss: 2017.655134, Training acc: 0.288400, Test acc: 0.262000,
Epoch: 27 Training Loss: 199.256521, Test Loss: 2015.921354, Training acc: 0.289600, Test acc: 0.266000,
Epoch: 28 Training Loss: 199.072124, Test Loss: 2014.232635, Training acc: 0.289200, Test acc: 0.265000,
Epoch: 29 Training Loss: 198.940572, Test Loss: 2013.534307, Training acc: 0.289400, Test acc: 0.259000,
Epoch: 30 Training Loss: 198.803510, Test Loss: 2013.034582, Training acc: 0.290800, Test acc: 0.265000,


Training vs Validation loss
Training Loss
Validation Loss
Loss
2250
2000
1750
1500
1250
1000
750
500
250
0
5
10
15
20
25
30
Epoch


Training vs Validation Accuracy
Training Accuracy
Validation Accuracy
Accuracy
0.28
0.26
0.24
0.22
0.20
0.18
0.16
0
5
10
15
20
25
30
Epoch

Accuracy and Confusion Matrix:

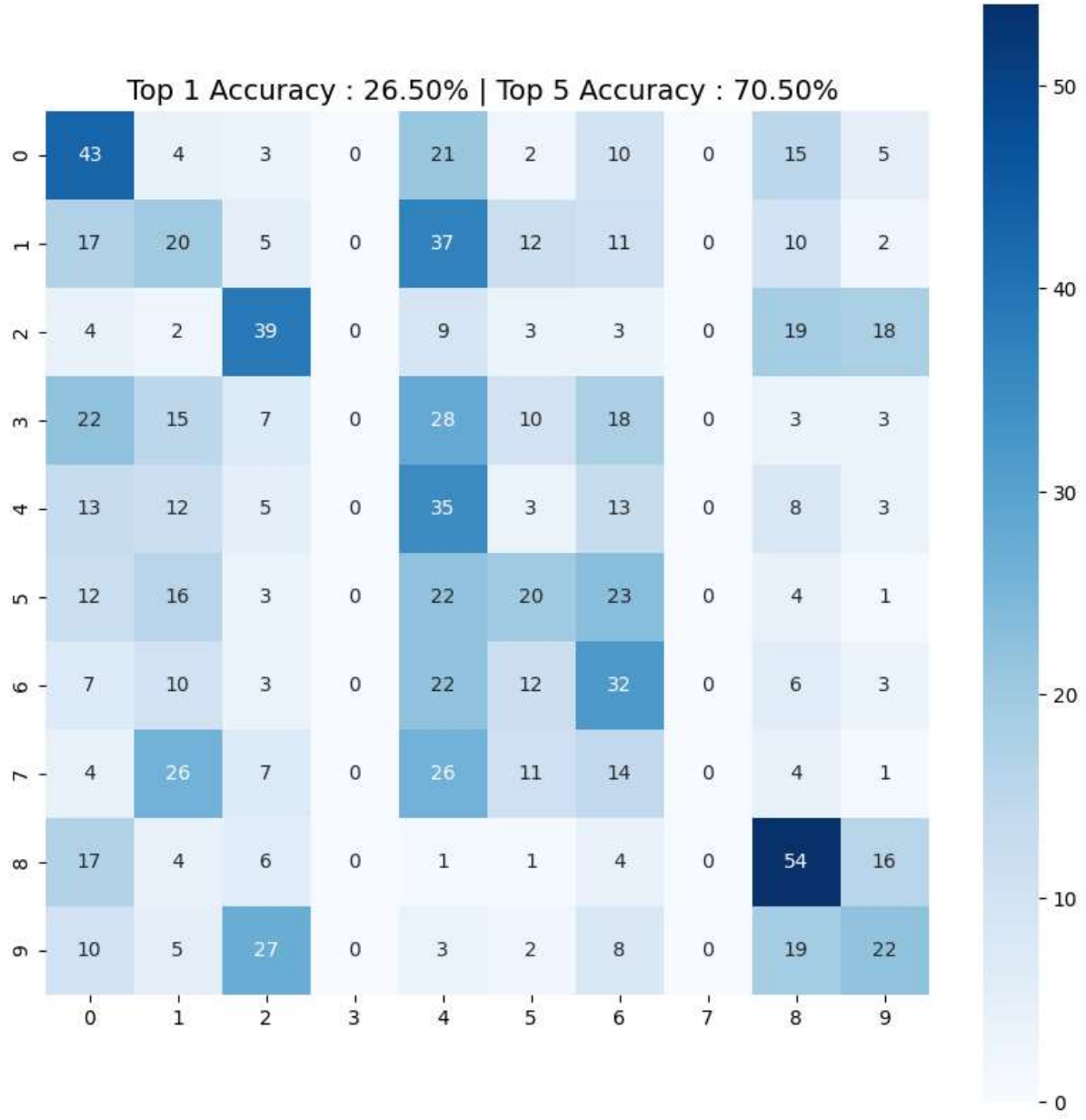
Top 1 Accuracy: 26.500%

Top 5 Accuracy: 70.5%

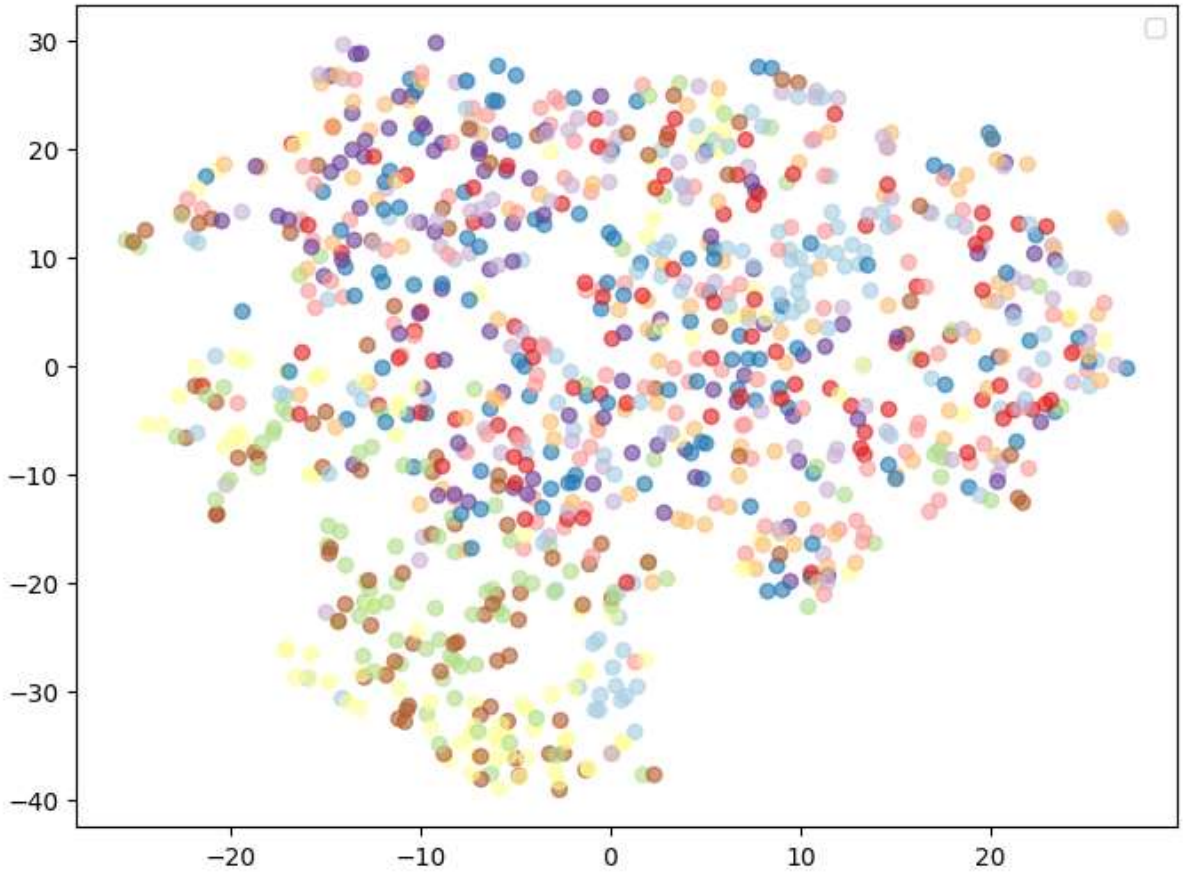
Classwise Accuracy Score:

[0.41747573 0.1754386 0.40206186 0. 0.38043478 0.1980198

0.33684211 0. 0.52427184 0.22916667]



TSNE Plot for the embeddings:



Observation and Conclusion

It can be seen from the results that model did not performed well and reasons for are following:

- Plain Feedforward network unable to extract spatial information from complex colour images. For initial few layer, Convolution Blocks may be used to overcome this issue.
- Lower number of epochs runs due to constrain of computing resources.

Solution of Question 1:

Resnet18

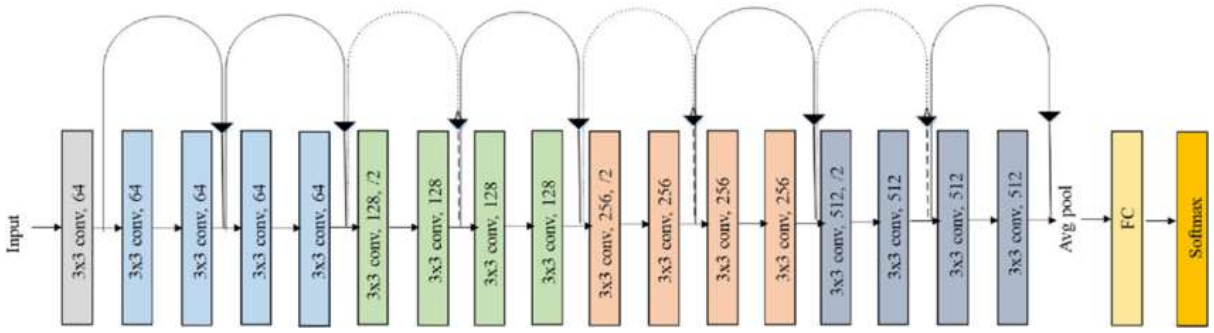
ResNet-18 is a convolutional neural network architecture that was introduced in 2015 as part of the ResNet (Residual Network) family of models. It was developed by Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun from Microsoft Research.

The main innovation of ResNet-18 is the use of residual connections, which allow the network to better learn the underlying mapping between the input and output. Instead of trying to learn the mapping directly, the network learns the residual mapping, which is the difference between the input and the desired output. This residual mapping is then added back to the input to obtain the final output.

ResNet-18 consists of 18 layers, including a convolutional layer, four residual blocks, and a fully connected layer. Each residual block contains two convolutional layers with batch normalization and ReLU activation, followed by the addition of the input and the residual mapping. The last layer of the network is a fully connected layer that produces the final output.

ResNet-18 has been widely used for a variety of computer vision tasks, such as image classification, object detection, and semantic segmentation, and has achieved state-of-the-art performance on many benchmark datasets.

Resnet18 Model Architecture:



Original ResNet-18 Architecture

```
ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
  (layer1): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
```

```
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  )
  (1): BasicBlock(
    (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  )
)
(layer2): Sequential(
  (0): BasicBlock(
    (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (downsample): Sequential(
      (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2), bias=False)
      (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (1): BasicBlock(
    (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  )
)
(layer3): Sequential(
  (0): BasicBlock(
    (conv1): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (downsample): Sequential(
      (0): Conv2d(128, 256, kernel_size=(1, 1), stride=(2, 2), bias=False)
      (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (1): BasicBlock(
    (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  )
```

```
  )
  (layer4): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
  (fc): Linear(in_features=512, out_features=10, bias=True)
)
```

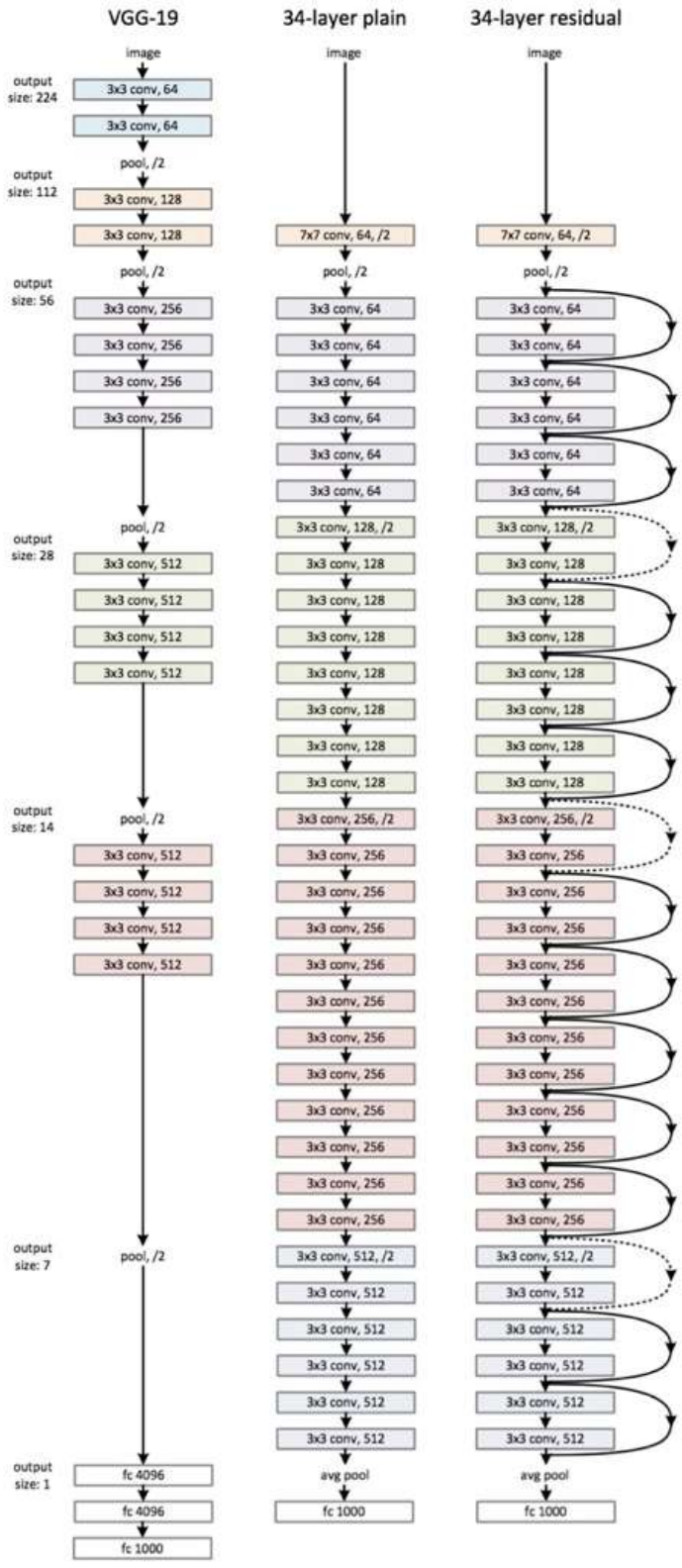


Figure 3. Example network architectures for ImageNet. **Left**: the VGG-19 model [41] (19.6 billion FLOPs) as a reference. **Middle**: a plain network with 34 parameter layers (3.6 billion FLOPs). **Right**: a residual network with 34 parameter layers (3.6 billion FLOPs). The dotted shortcuts increase dimensions. **Table 1** shows more details and other variants.

ResNet 34 from original paper

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

Sizes of outputs and convolutional kernels for ResNet 34

One of the problems ResNets solve is the famous known vanishing gradient. This is because when the network is too deep, the gradients from where the loss function is calculated easily shrink to zero after several applications of the chain rule. This result on the weights never updating its values and therefore, no learning is being performed.

With ResNets, the gradients can flow directly through the skip connections backwards from later layers to initial filters.

Given Dataset:

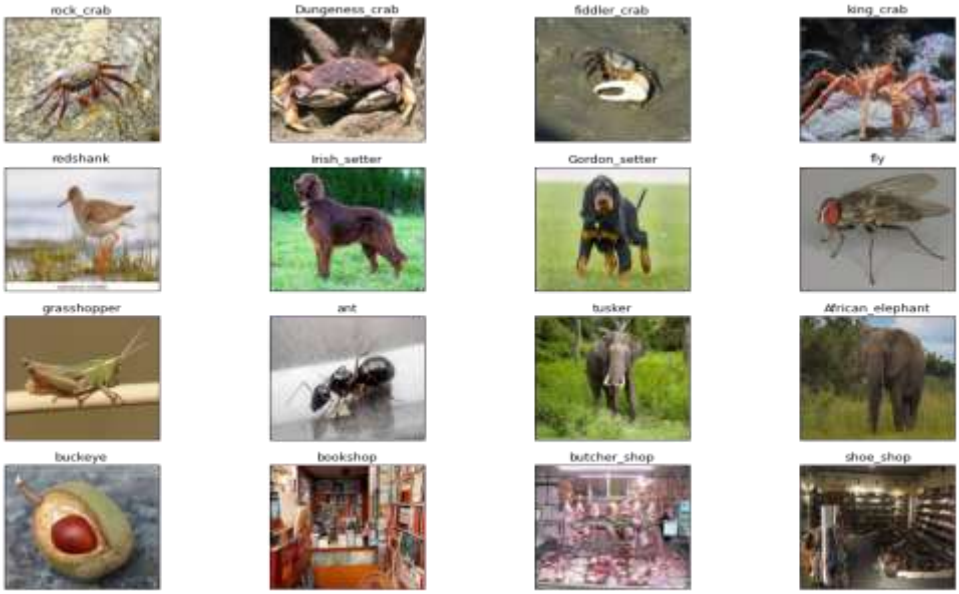
Tiny ImageNet

Training Set: 100000 (500 for each class) Colour (3 channels) 64×64 Images

Testing Set: 100000 (500 for each class) Colour (3 channels) 64×64 Images

Classes: 200 classes

Data Samples:



- Optimizer X = Adam, as last digit of my roll no(M21AIE225) is odd

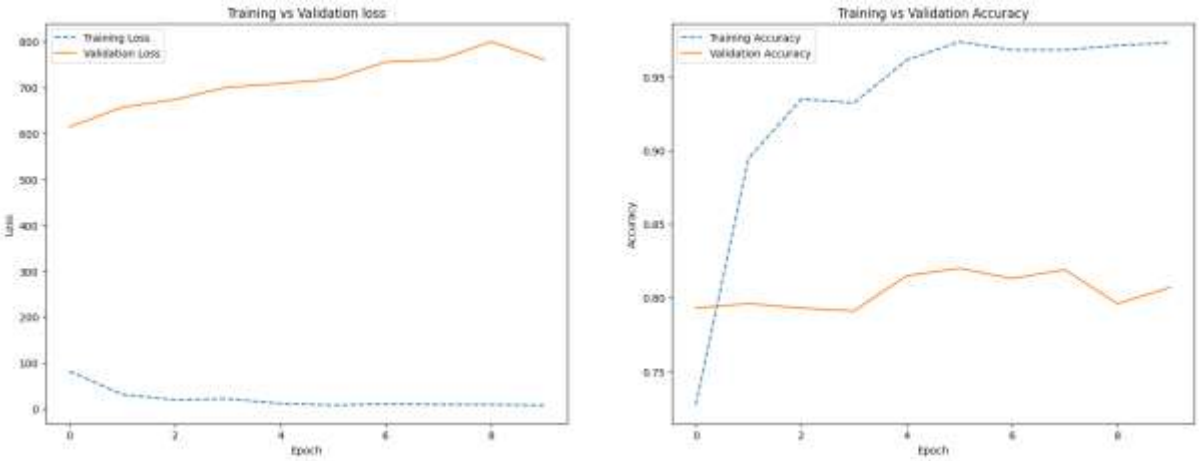
Adam is an optimization algorithm that computes adaptive learning rates for individual weights during training. It maintains a running estimate of the first and second moments of the gradients of the weights, denoted by m and v, respectively. These estimates are initialized as vectors of zeros and updated using a combination of a moving average and bias correction. The optimizer computes bias-corrected estimates of m and v as m_hat = m / (1 - β1^t) and v_hat = v / (1 - β2^t), where β1 and β2 are hyperparameters controlling the exponential decay rates of the estimates, and t is the iteration number. Finally, the optimizer updates the weights using a learning rate α and the bias-corrected estimates of m and v as follows: θ = θ - α * m_hat / (sqrt(v_hat) + ε), where ε is a small constant added to the denominator to avoid division by zero.

- Loss function = Cross Entropy

Model Training Log:

Epoch: 1 Training Loss: 82.078650, Test Loss: 614.433289, Training acc: 0.727000, Test acc: 0.793000,
Epoch: 2 Training Loss: 31.350838, Test Loss: 657.145262, Training acc: 0.894600, Test acc: 0.796000,
Epoch: 3 Training Loss: 19.408739, Test Loss: 673.955321, Training acc: 0.934800, Test acc: 0.793000,
Epoch: 4 Training Loss: 21.767490, Test Loss: 700.918734, Training acc: 0.932200, Test acc: 0.791000,
Epoch: 5 Training Loss: 11.681993, Test Loss: 709.099174, Training acc: 0.961400, Test acc: 0.815000,
Epoch: 6 Training Loss: 7.955046, Test Loss: 718.438625, Training acc: 0.973800, Test acc: 0.820000,
Epoch: 7 Training Loss: 10.227830, Test Loss: 755.967617, Training acc: 0.968200, Test acc: 0.813000,
Epoch: 8 Training Loss: 9.303071, Test Loss: 760.519803, Training acc: 0.968200, Test acc: 0.819000,
Epoch: 9 Training Loss: 9.134530, Test Loss: 800.109386, Training acc: 0.971200, Test acc: 0.796000,
Epoch: 10 Training Loss: 7.732154, Test Loss: 761.471570, Training acc: 0.973200, Test acc: 0.807000,

Model Training Losses and Accuracies:



Accuracy:

Top 1 Accuracy: 80.700%
Top 5 Accuracy: 98.8%

Class wise Accuracy Score:

[0.89320388 0.78070175 0.90721649 0.69811321 0.82608696 0.58415842

0.82105263 0.80645161 0.9223301 0.84375   ]

Q. 1.2. Triplet Loss with hard mining as the final classification loss function:

Triplet Loss:

Triplet Loss is a loss function, where the goal is to learn a mapping from the input space to a metric space, such that similar inputs are mapped close to each other and dissimilar inputs are mapped far apart.

The Triplet Loss function involves selecting triplets of examples (anchor, positive, and negative), and then minimizing the distance between the anchor and positive examples while maximizing the distance between the anchor and negative examples. This can be expressed as:

L_triplet = max(0, d(a,p) - d(a,n) + margin)

where d(a,p) is the distance between the anchor (a) and positive (p) examples, d(a,n) is the distance between the anchor (a) and negative (n) examples, and margin is a hyperparameter that determines the minimum difference between the distances.

In order to make the Triplet Loss function more effective for classification tasks, hard mining can be used. Hard mining involves selecting the hardest negative example for each anchor-positive pair, i.e., the negative example that has the smallest distance to the anchor among all negative examples. This makes the loss function more discriminative, as it focuses on the most difficult examples to classify.

Therefore, using Triplet Loss with hard mining as the final classification loss function can be an effective way to learn a metric space that is optimized for classification tasks.

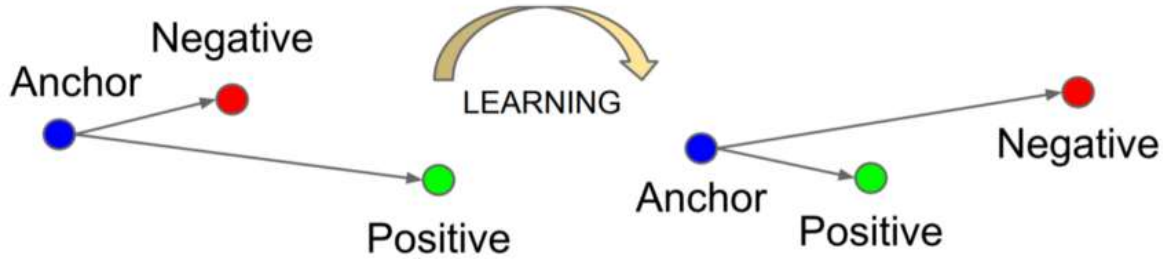


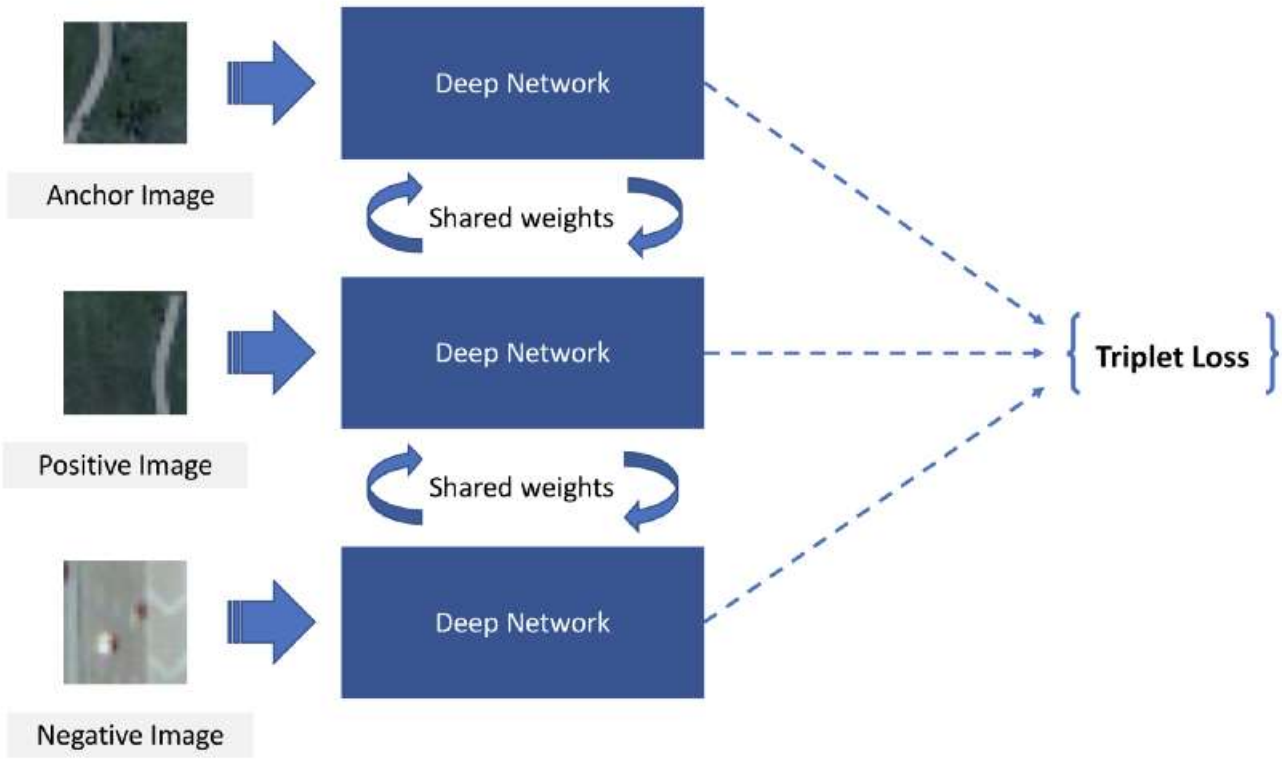
Image Source: medium.com

$$\sum_{i}^{N} \left[\|f(x_i^a) - f(x_i^p)\|_2^2 - \|f(x_i^a) - f(x_i^n)\|_2^2 + \alpha \right]$$

Mathematical Equation of Triplet Loss Function.

- ***f(x)*** takes ***x*** as an input and returns a vector ***w.***
- ***i*** denotes *i'th* input.
- Subscript ***a*** indicates ***Anchor*** image, ***p*** indicates ***Positive*** image, ***n*** indicates ***Negative*** image.

Our objective is to *minimize the above equation*, which implicitly means:-
Minimizing first term → distance between Anchor and Positive image.
Maximizing(since it has negative sign before it) second term → distance between Anchor and Negative image.



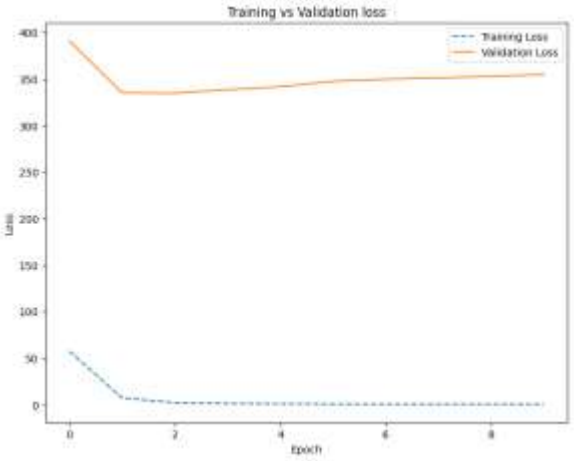
Triplet Loss architecture

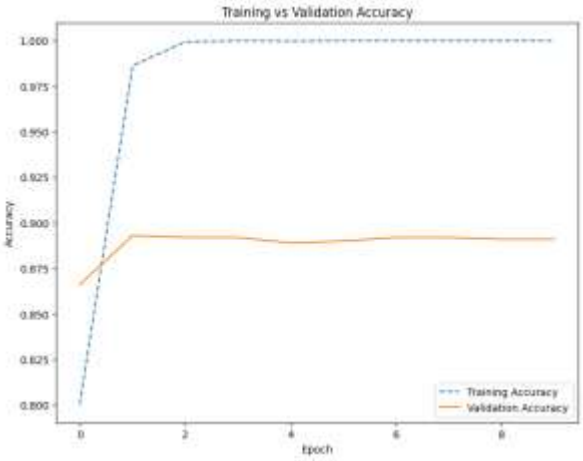
Training and Observations:

Model Training Log:

Epoch: 1 Training Loss: 57.339447, Test Loss: 391.216218, Training acc: 0.800400, Test acc: 0.866000,
Epoch: 2 Training Loss: 7.036367, Test Loss: 335.199594, Training acc: 0.986200, Test acc: 0.893000,
Epoch: 3 Training Loss: 2.061597, Test Loss: 334.692627, Training acc: 0.999400, Test acc: 0.892000,
Epoch: 4 Training Loss: 1.033667, Test Loss: 338.609815, Training acc: 1.000000, Test acc: 0.892000,
Epoch: 5 Training Loss: 0.771824, Test Loss: 341.696292, Training acc: 0.999800, Test acc: 0.889000,
Epoch: 6 Training Loss: 0.554433, Test Loss: 347.637147, Training acc: 1.000000, Test acc: 0.890000,
Epoch: 7 Training Loss: 0.439116, Test Loss: 349.825233, Training acc: 1.000000, Test acc: 0.892000,
Epoch: 8 Training Loss: 0.407432, Test Loss: 351.264626, Training acc: 1.000000, Test acc: 0.892000,
Epoch: 9 Training Loss: 0.375267, Test Loss: 352.931619, Training acc: 1.000000, Test acc: 0.891000,
Epoch: 10 Training Loss: 0.298735, Test Loss: 354.932994, Training acc: 1.000000, Test acc: 0.891000,

Model Training Losses and Accuracies:



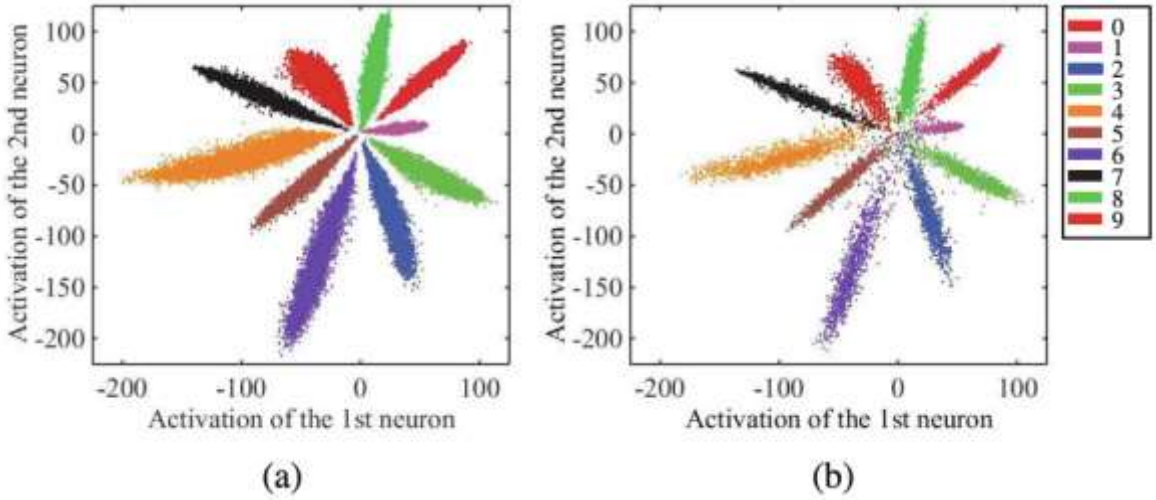


Accuracy:

Top 1 Accuracy: 89.100%
Top 5 Accuracy: 99.6%
Classwise Accuracy Score:
[0.91262136 0.90350877 0.96907216 0.77358491 0.81521739 0.82178218
 0.91578947 0.94623656 0.95145631 0.90625]

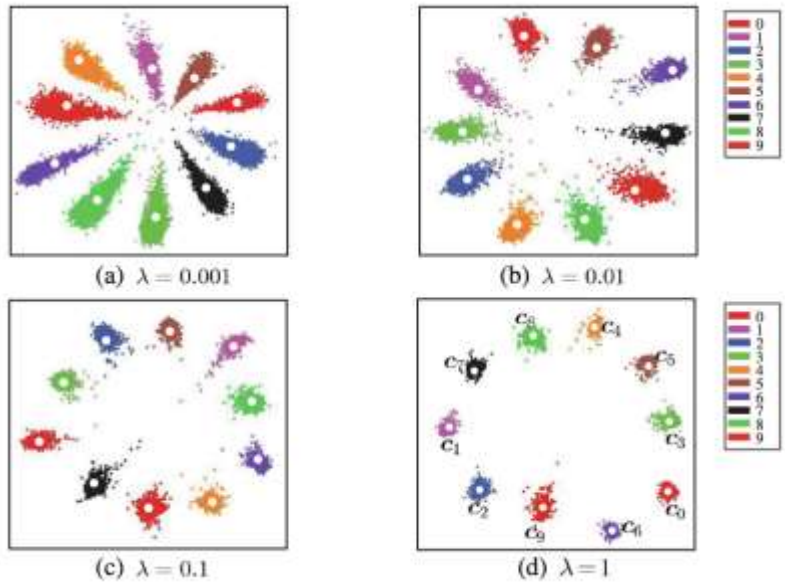
Q. 1.3. Center Loss:

Center loss is a strategy for constructing widely-separated classes.A common problem with ordinary supervised learning is that the latent features for the classes can end up being tightly grouped. This can be undesirable, because a small change in the input can cause an example to shift from one side of the class boundary to the other.



This plot from the paper shows how the activations of 2 neurons display the (a) training data and (b) testing data. Especially near the coordinate (0,0), there is an undesirable mixing of the 10 classes, which contributes to a larger error on the test set.

By contrast, the center loss is a regularization strategy that encourages the model to learn widely-separated class representations. The center loss augments the standard supervised loss by adding a penalty term proportional to the distance of a class's examples from its center.



The center loss includes a hyperparameter λ which controls the strength of the regularization. Increasing λ increases the separation of the class centers.

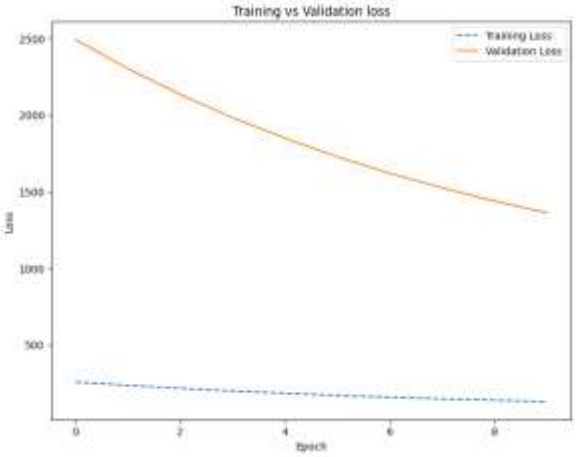
These images were taken from Yandong Wen et al., "A Discriminative Feature Learning Approach for Deep Face Recognition" (2016).

Training and Observations:

Model Training Log:

Epoch: 1 Training Loss: 256.169827, Test Loss: 2491.599083, Training acc: 0.088800, Test acc: 0.106000,
Epoch: 2 Training Loss: 234.733407, Test Loss: 2301.831961, Training acc: 0.150600, Test acc: 0.166000,
Epoch: 3 Training Loss: 215.314672, Test Loss: 2132.591963, Training acc: 0.234600, Test acc: 0.258000,
Epoch: 4 Training Loss: 198.007964, Test Loss: 1981.519461, Training acc: 0.313200, Test acc: 0.327000,
Epoch: 5 Training Loss: 183.040474, Test Loss: 1846.614122, Training acc: 0.390400, Test acc: 0.379000,
Epoch: 6 Training Loss: 169.069120, Test Loss: 1725.814939, Training acc: 0.466200, Test acc: 0.438000,
Epoch: 7 Training Loss: 157.340216, Test Loss: 1618.575931, Training acc: 0.525200, Test acc: 0.484000,
Epoch: 8 Training Loss: 146.545958, Test Loss: 1522.536397, Training acc: 0.575000, Test acc: 0.532000,
Epoch: 9 Training Loss: 137.393952, Test Loss: 1436.814308, Training acc: 0.617800, Test acc: 0.567000,
Epoch: 10 Training Loss: 129.242647, Test Loss: 1360.561848, Training acc: 0.652200, Test acc: 0.601000,

Model Training Losses and Accuracies:



Confusion Matrix and Accuracy:

Top 1 Accuracy: 60.100%
Top 5 Accuracy: 94.6%
Classwise Accuracy Score:
[0.66019417 0.53508772 0.88659794 0.41509434 0.5 0.45544554
0.55789474 0.64516129 0.85436893 0.51041667]

References:

1. Lectures of DL Ops class by Prof Pratik Majumder
2. Implementation Session by Teaching Assistants
3. Blogs from Internet