Nirbhay Sharma (B19CSE114)

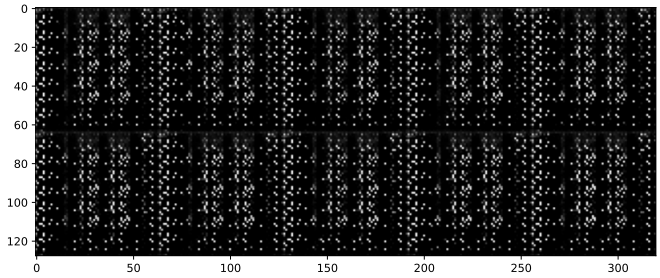
Deep Learning assignment - 3

Note: - along side the code for que2 and que4, the generated log files are also attached to show that the codes were indeed run and the results are indeed reproduced.

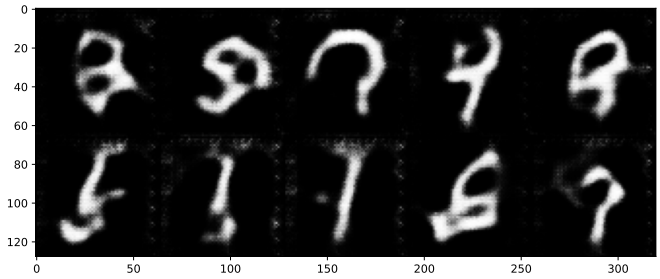
que-1

- For training the GAN the generator and discriminator are prepared and the gan is trained using the discriminator and generator loss with epochs = 15 and batch_size = 64 and learning rate = $2e^{-4}$ and the initial random vector has size of 256
- results of the training

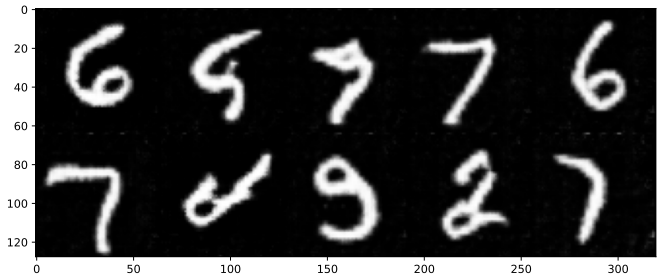
after first epoch



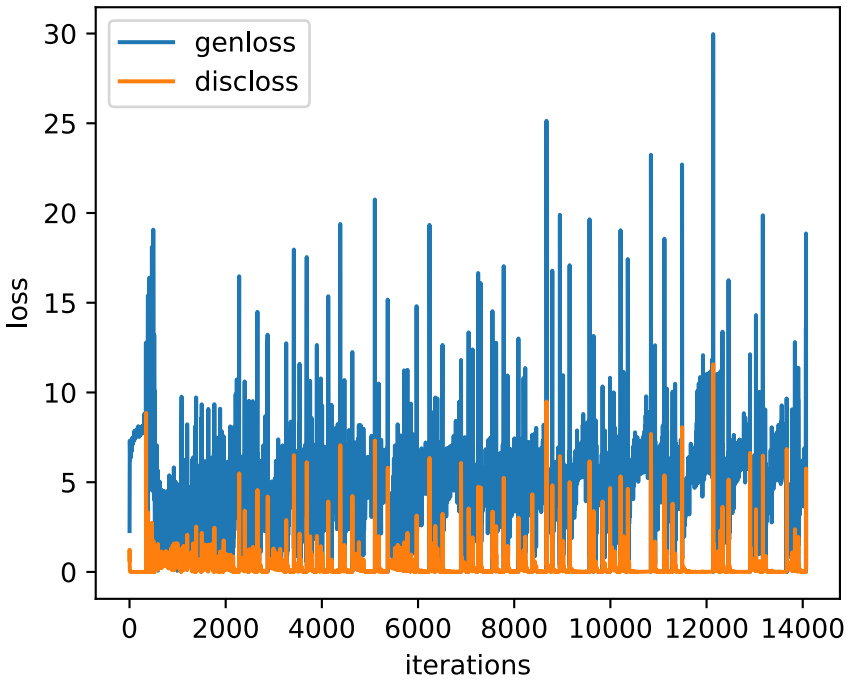
after $\frac{n}{2}$ = 7 epochs



after n epochs



- after the first epoch none of the images is identifiable since the network requires some time to adapt to the distribution
- after 7 epochs the images are somewhat clear and still some of the images are not visible since the model has start converging here it tries to generate images but they are still blurr
- at 15 th epoch the images are much clearer and are easily identifiable like we can clearly identify 6, 7 5 etc.
- generator discriminator loss function is as follows



[bonus]

- no, in the above GAN i.e. DCGAN we have no control on what the generator is generating since we are just passing the random vector to the generator and in turn it is generating the images according to the distribution.
- so suppose we want to generate only images of 4 then the above generator would not be able to do that since it has only knowledge of distribution and not the knowledge of the class labels and hence the gan needs to be modified with providing the class information as well to the generator
- to modify the above GAN we need to convert it to the conditional GAN which takes in random noise and class labels information as well and generate the images of what we are intended to generate

que-2

code-reference: github

code-explanation & algorithm-explanation

there are various files in the code named

- **omniglog_dataset.py** - This file contains the code for general dataloader for downloading and loading the omniglot dataset and it does it by making a class for dataset and the format of class is generally

```
class dataset():
  def __init__(self):
    pass

  def __len__(self):
```

```
    pass

  def __getitem__(self,idx):
    pass
```

using these dunder methods the dataset is loaded efficiently at runtime

- **parset_util.py** - This file contains the code for managing the command line arguments and for that purpose they use argparse, argparse is sometimes very helpful in orgainizing the commandline arguments and this is what this file is doing
- **protonet.py** - This file contains the code for the protonet model which is basically a CNN architecture with Conv2d, Maxpool, Relu, BatchNorm layers and all the layers are combined to make final protonet model which is basically an encoder which takes in the input image and convert it into embeddings of z_dim size so output from the protonet models is of the shape [batch_size, z_dim]
- **prototypical_batch_sampler.py** - This file contains the code for prototypical batch sampler and it basically returns an iterator which yields the batches at a particular index
- **prototypical_loss.py** - This file contains the loss for training the prototypical network which is as follows we are given input_images which is our query set and we are give target which is our ground truth classes and we are provided with support vectors which are the vector representations of support images that we chosen initially now we find the prototype or representative of each class which is a vector v^k for class k, and is calculated as

$$v^{(k)} = \frac{1}{N}\sum_{i=1}^{N} f_\phi(x_i^{(k)})$$

so suppose there are three classes and we have support vectors for each support vector then the final prototype vector for each class is $v^{(1)}, v^{(2)}, v^{(3)}$ so then we compute the distance of each of the query samples with the prototype vector to get class probabilities using the distance function like eucledian function and softmax function

calculating distance for x with each class (k)

$$d(f_\phi(x), v^{(k)}) = \sqrt[2]{f_\phi(x) - v^{(k)}}$$

computing the softmax (class probabilities) and the ground truth is K (-ve sign in the distance is to keep the class probability high once the distance starts reducing)

$$P(y = K/x) = \frac{e^{-d(f_\phi(x), v^{(k)})}}{\sum_{k'=1}^{C} e^{-d(f_\phi(x), v^{(k')})}}$$

and now penalize this using NLL loss i.e $-log(P(y = K/x))$, so overall loss function becomes (C represents total number of classes and Q represents total query samples)

$$L = \sum_{i=1}^{Q} \frac{1}{CQ} [d(f_\phi(x_i), v^{(k)}) + log(P(y = K/x_i))]$$

and this loss is getting backpropagated

- **train.py** - The train.py contains the basic training pipeline of importing the dataset, chossing the optimizer, loss etc and train the model

Algorithm

- This simple algorithm is done in 5 steps
- First calculate the embeddings of the query images using the protonet architecture
- similarly calculate the embeddings for support images
- find the prototypical vector representing each class with the help of support vectors obtained from support images
- calculate distance and find the loss function
- backpropagate the loss function to train the weights for the prototypical architectures
- the loss function ensures that the embeddings distance from their respective prototypical vector is as small as possible and also tries to penalize for wrong class probability prediction

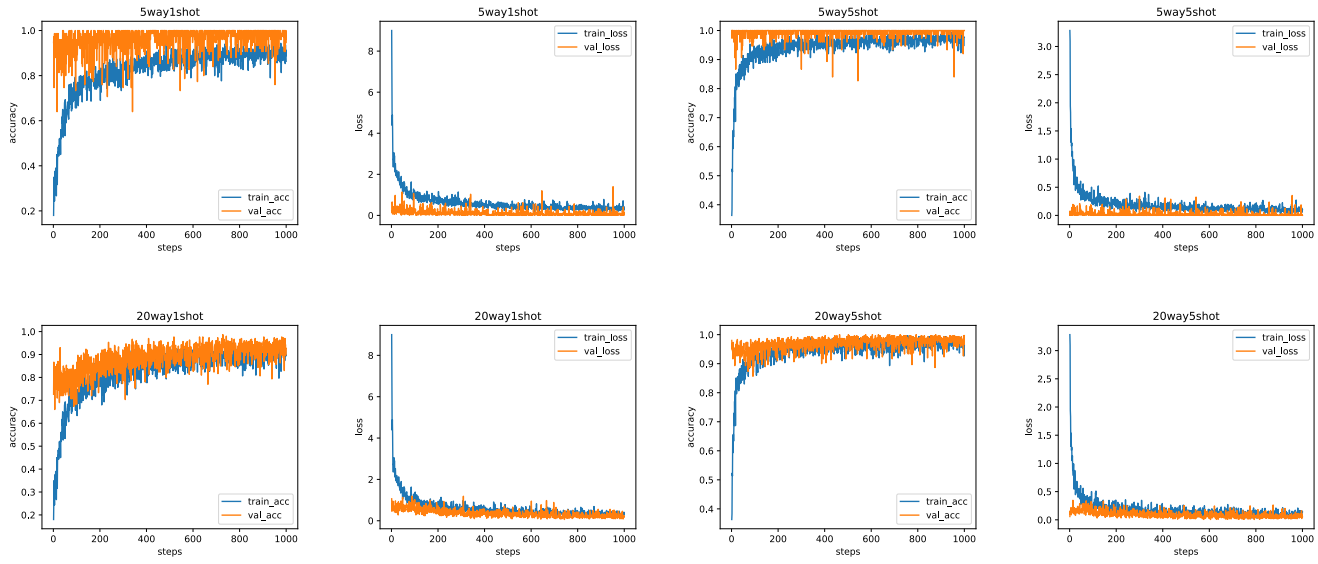
results

accuracy on test set

Impelmentation	5way1shot	5way5shot	20way1shot	20way5shot
paper	98.8	99.7	96	98.9
reproduced	97.06	99.31	91.47	97.61

plots

loss accuracy plots for all the algorithms

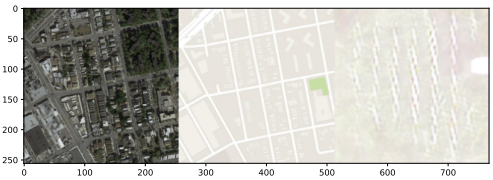


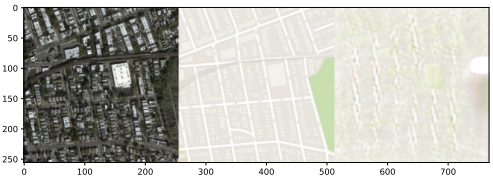
results analysis

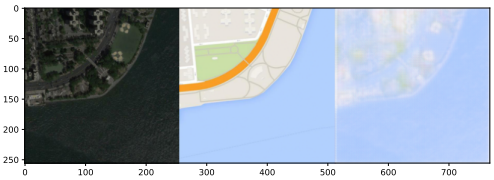
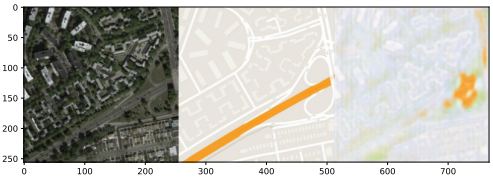
- the results are very good and are matching with the official paper, the differences in accuracies is maybe due to the different versions of pytorch but the overall results are promising and are comparable to what is reported in paper
- now coming to the plots, the plots also looks very promising and are created on all the 4 methods (20way1shot, 20way5shot, 5way1shot, 5way5shot) with validation and train accuracy and validation and train loss, the loss curve is perfectly decreasing in all the methods and the accuracy curve is perfectly increasing in all the methods
- moreover the validation and training, loss and accuracy goes hand in hand and which also shows that the model does not overfit

que-3

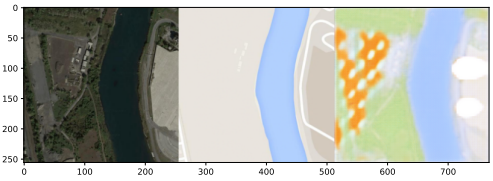
- sample sat2map images from the test set are presented below ([input_image, ground_truth, generated_image])



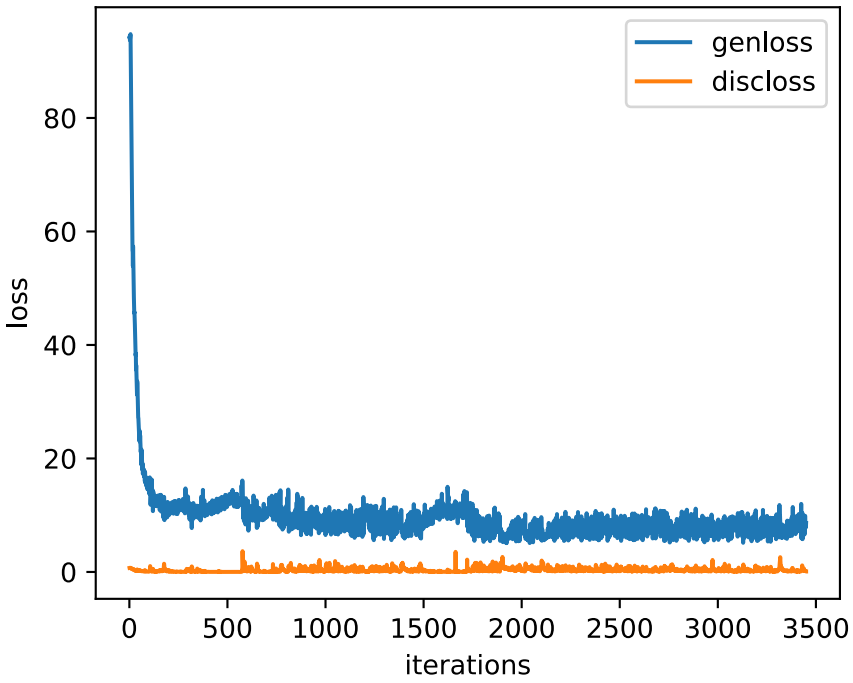




- a single set of 10 good generated test samples

comments on above generated images

- as we can see from the above images the model is converging in a good way just some slight error is there, like as observed from the images we can see that it generates blue color very well and where ever a river was there it generates blue color very well but just some noise is generated also which is due to orange color and it looks like that model kind of learns to generate orange color so it is generating them in some of the images, but overall result looks promising
- generator-discriminator loss is presented below



que-4

code-reference: github

code-explanation & algorithm explanation

DARTS (Differentiable Architecture Search) is a method introduced by the authors in the paper "DARTS" where they have proposed an approach for the deep learning architecture search, apart from the NAS/AUTOML techiniques, DARTS uses gradient based methods by continuous relaxation and optimization techniques and combined they form the architecture parameters α which represents the search architecture, DARTS contains the following techniques

1. search space - based on the fact that the CNN architectures are basically repeatation of some particular blocks / cells and on the same line the authors try to search for that particular cell/ block , repeatation of which will make neural network work well and so their search space is now confined to searching a cell, cell according to the authors is DAG (directed acyclic graph) consisting orderd sequence of N nodes connected to each other in DAG fashion and the output of one node depends on the output of previous nodes in the graph, each edge (i,j) has some associated operation $o^{(i,j)}$ (operations include convolution, maxpool etc) applied on some input x^i and hence the output of the nodes is considered as

$$x^{(j)} = \sum_{i<j} o^{(i,j)} x^{(i)}$$

2. continuous relaxation and optimization - Consider O be a set of candidate operations (convolution,batchnorm) etc, where it's each value i.e. $o(.)$ is applied on input $x^{(i)}$ and hence in order to make search space continuous the authors have relaxed the choice of operation using softmax using the equation

$$o^{(i,j)}(x) = \sum_{o \in O} \frac{e^{\alpha_o^{(i,j)}}}{\sum_{o' \in O} e^{\alpha_{o'}^{(i,j)}}} o(x)$$

after relaxation the weight need to be learning using some optimization tecniques and it does so using gradient descent algorithm which include

$$min_\alpha L_{val}(w^*(\alpha), \alpha)$$

$$w^*(\alpha) = argmin_w L_{train}(w, \alpha)$$

where w^* represents weight came after minimizing the training loss over previous w and architecture α, the summary of the algorithm is shown below

Algorithm 1: DARTS – Differentiable Architecture Search

Create a mixed operation $\bar{o}^{(i,j)}$ parametrized by $\alpha^{(i,j)}$ for each edge (i, j)
while *not converged* **do**
1. Update architecture α by descending $\nabla_\alpha \mathcal{L}_{val}(w - \xi \nabla_w \mathcal{L}_{train}(w, \alpha), \alpha)$ ($\xi = 0$ if using first-order approximation)
2. Update weights w by descending $\nabla_w \mathcal{L}_{train}(w, \alpha)$

Derive the final architecture based on the learned α.

the final architecture is obtained and tested on the test set and to test it they basically discard the weights and retrain the whole model on train set and test on test set.

- As far as code is concerned we had various files in CNN folder such as genotypes files which contains the genes, which are the building blocks of the cells the genes / genotypes includes convolution operations with various kernel_size (3×3, 5×5) and maxpool operations and various types of convolutions such as separable convolutions etc
- the architect files contains the whole code for optimization and backward propagations
- in model_search.py file all the code for creating the cell and network is present and the cells constitues of genotypes and in this manner the whole network is created
- the utils.py file contains the utilities for saving and loading the models as well as finding the accuracy and saving chekpoints etc
- the train_search.py file contains the pipeline for training, the pipeline is same to generate an architecture and test it on test set etc.

how to run the code

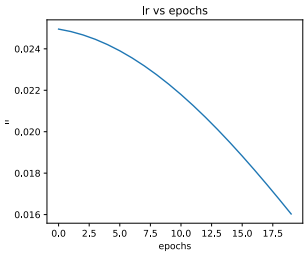
- the code is written in earlier version of pytorch and hence to run it in newer versions we need to do the following changes
- in train_seach.py file we need to replace all .data[0] with .item() and in newer version .item() is used to extract the numerical value from the tensor
- in utils.py file the 37 th line needs to be modified as follows (correct_k = correct[:k].flatten().float().sum(0)) , instead of correct[:k].view(-1) we need to use .flatten()
- and finally run the train_search file using python3 train_search.py command

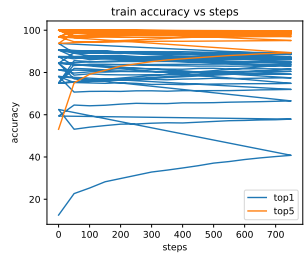
results

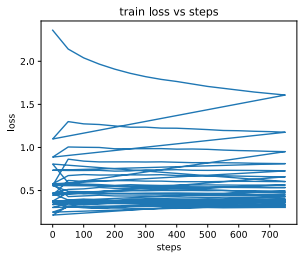
accuracy on cifar10 test set

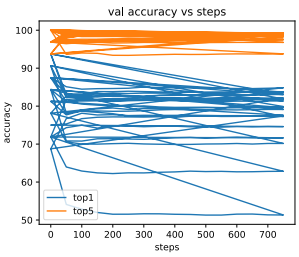
Implementation	top-1	error %
paper	97%	3 %
reproduced	83.81%	16.1 %

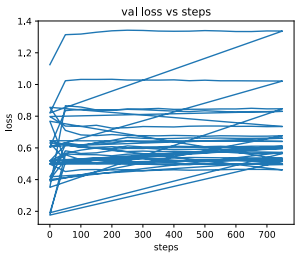
plots











result analysis

- In the official implementation the code was ran for around 300 epochs, due to the computation constraint the code needs to be run at 20 epochs and that too took around 3GPU Days to run so the reproduced accuracy is lower than the reported in the paper
- Moreover the curves looks good, by looking at the curves it looks like that the search has started with low accuracy and high loss and gradually it optimizes by choosing the model and at the end the accuracies kind of starts increasing slowly but they are increasing anyways so more number of epochs is recommended to get the best result
- the learning rate vs epochs curve is also looking good, the search actually reduces the learning rate with the course of epochs so that the search does not overfit.

references

1. https://towardsdatascience.com/few-shot-learning-with-prototypical-networks-87949de03ccd

---