BRAIN AND COGNITIVE SOCIETY

IIT KANPUR

The Omniglot Challenge

Som Tambe - Department of Aerospace Engineering

Nikita Chauhan - Department of Computer Science Engineering

Anmol Pabla - Department of Mechanical Engineering

Mohit Kulkarni - Department of Mathematics

Vaibhav Thakkar - Department of Electrical Engineering

Mentor: Shashi Kant

Department of Electrical Engineering

Developing *one-shot learning* models with greater scalability, using deep learning techniques

Som Tambe
somvt@iitk.ac.in
Department of Aerospace Engineering

Nikita Chauhan
nikitach@iitk.ac.in
Department of Computer Science & Engineering

Introduction

The first thing that strikes our mind, or what is popularly taught in various courses around the world, when we see neural networks, that they are analogous to neurons in our brains. This analogy is not completely true. Generally, deep neural networks, or any machine learning model, requires large datasets to generalize trends/patterns across a wide variety of samples.

Humans, when given a set of completely new objects, can classify the similar objects given a reference object, without requiring hundreds-of-thousands of examples. Past efforts to counter these problems include the Bayesian Program Learning [1].

The BPL paper hand-engineered various parts of the model to serve to the similarity and generation tasks with the Omniglot dataset [2]. We aim to create much more generalized models, which can serve for a wide variety of datasets, and tend to be less hand-engineered towards a particular target.

Models

The articles below enlist all the study and implementation we have done yet. All references have been mentioned at the end of the document, and mentioned wherever used.

1 Enhanced learning based on stroke data

Preface

The implementation for this model has been carried on by Som Tambe. The study is not yet complete yet, partly due to time constraints and due to unexpected results.

Approach

For this model, I use the Omniglot Dataset [2]. The dataset contains 964 different characters, with 20 examples per class. This accounts to much lesser than what standard datasets have, for instance MNIST [3] has about 6000 examples per class. Ergo, I have chosen the dataset (also because of the fact it has become a standard for benchmarking one-shot learning models) .

Algorithms used

The algorithms have been described in the table below-
An example spline is shown in Fig. 1.

This helps us make all the strokes to a constant length of 25. I use the splprep module from scipy.interpolate, which allows me to interpolate it into parametric curve (since strokes are not perfect functions, therefore we use parametric representation).

Algorithm 1 Conversion of stroke data to 25-splines

```
procedure SPLINE($d$)                                    ▷ where $d$ is a single part stroke data
    while no error do
        if len($d$)$\geqslant$ 5 then
            return spline($d, 3$)                        ▷ spline with degree=3
        else
            if 1 ¡ len($d$)$\leqslant$ 5 then
                return spline($d, 1$)                    ▷ degree=1
            end if
            if len($d$)=1 then
                return repetition($d$)                   ▷ 25 times
            end if
        end if
    end while
    while throws error do
        $d \leftarrow d + \mathcal{N}(0, 1)$             ▷ Add some noise
        procedure SPLINE($d$)
        end procedure
    end while
end procedure
```

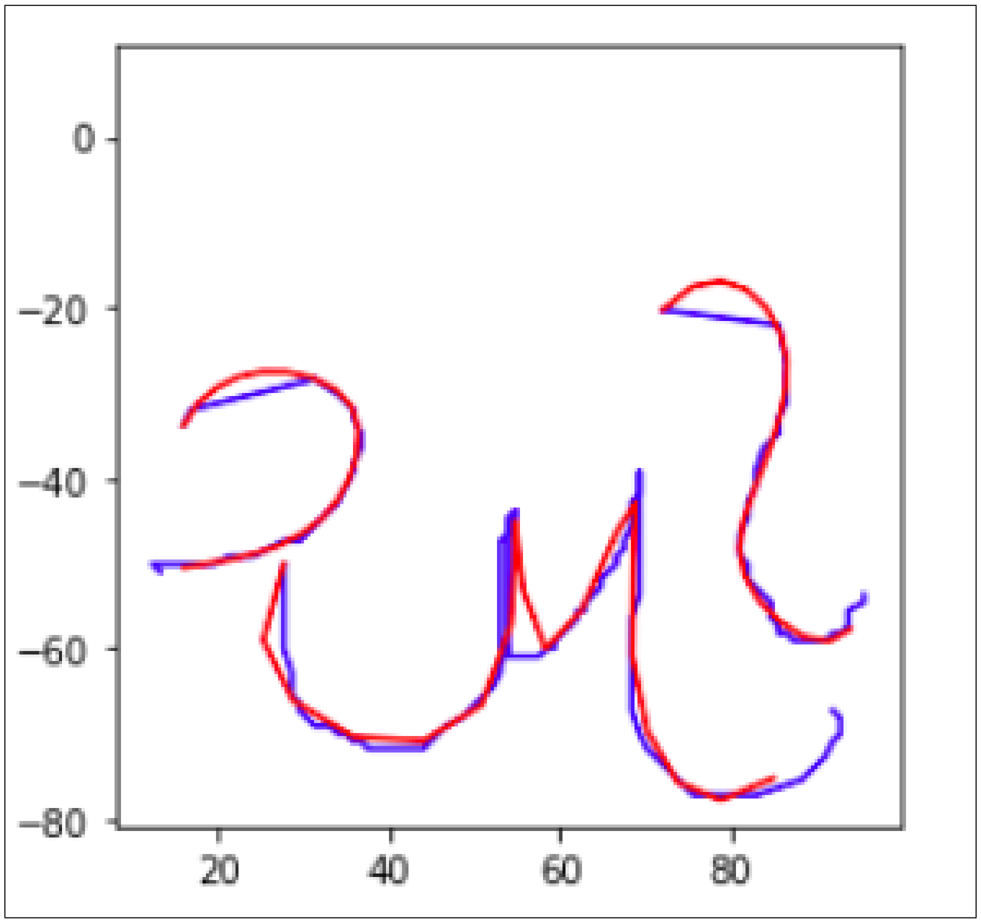


Figure 1: In blue: Real stroke. In red: 25-length stroke obtained from constructing a spline.

I had also been suggested the use of Ramer-Douglas-Peucker [5] (**RDP**) algorithm, which helps decimate a curve to a fewer point representation, by varying the value of epsilon iteratively. We decided not to use this after comparing the results of scipy.interpolate and RDP .

Errors were observed when completely vertical lines were sub-parts of strokes. I thus add some extra gaussian noise in each coordinate $\sim \mathcal{N}(0,1)$.

Next thing we do is contruct a Variational Autoencoder [4] . We also use the reparameterization trick mentioned in [4] . This is to construct a 2-dimensional latent space visualization of how the strokes are distributed in space.

Another reason we use a VAE is that there exist only $\sim$ 48000 strokes in total. Therefore using a VAE to an ordinary autoencoder gives us more stochasticity in terms of the generated latent vector, therefore making training better, and faster.

Algorithm 2 Using a VAE to find the 2-d latent space representation of strokes

```
    procedure ENCODE($r$)
2:      $d \leftarrow (d - \mu_d)/\sigma_d$                ▷ Normalizing the spline
        $d \leftarrow d.view(50)$                          ▷ Flatten the coordinates
4:      $\mu_z, \sigma_z \leftarrow encoder(d)$            ▷ Mean, variance of latent vector
        $z \leftarrow \mathcal{N}(\mu_z, \sigma_z)$        ▷ Sample from a normal distribution
6:      $d' \leftarrow decoder(z)$                         ▷ Generated coordinates
        Calculate loss                                     ▷ KLD & recombination
8:      Backpropogate derivatives
        Update network
10: end procedure
```

After training this VAE network, we first observe the decoder outputs. Fig 5 shows the results.

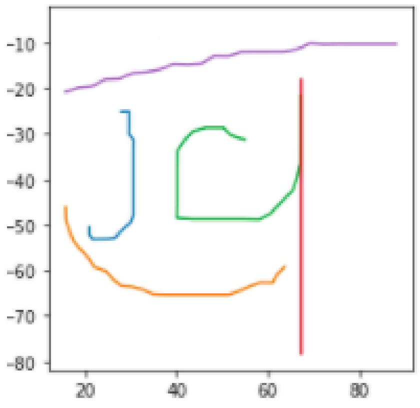


Figure 2: Original

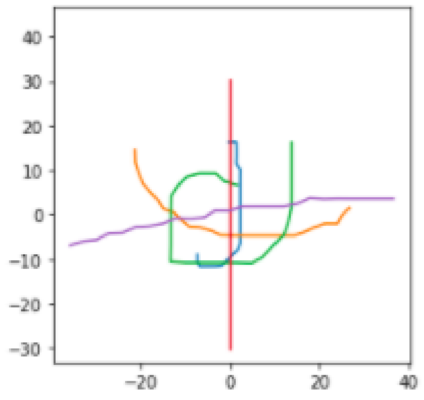


Figure 3: Original normalized

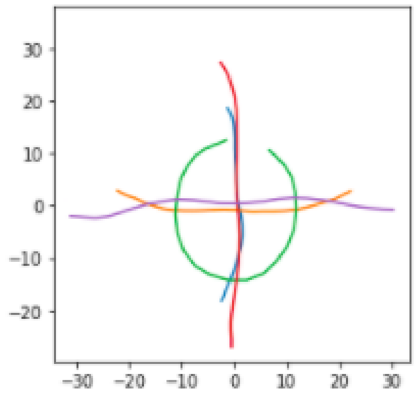


Figure 4: Decoder output

Figure 5: VAE results. Fig 3 was fed to the VAE network, Fig 4 was the output. Notice how similar do the normalized input and the output look. This is a first. Also notice, strokes are denoted with different colours.

These results have been obtained after training for 150 epochs. The loss had started converging well before 150 epochs, and after randomly testing characters, we finally concluded the training was complete.

We discard the decoder network after training (although I store the weights). The encoder gives us all that is required. We now proceed to the further part of the model.

We decide to plot the 2-dimensional representation of the obtained stroke representation. Here is what we got, shown in Fig 6 .

By plotting the 2 dimensional plot, I aimed to get noticeable clusters which would help me identify how many different categories of parts exist. This would help me prepare one hot vectors of characters, given their stroke data, which would denote the parts present in the characters.

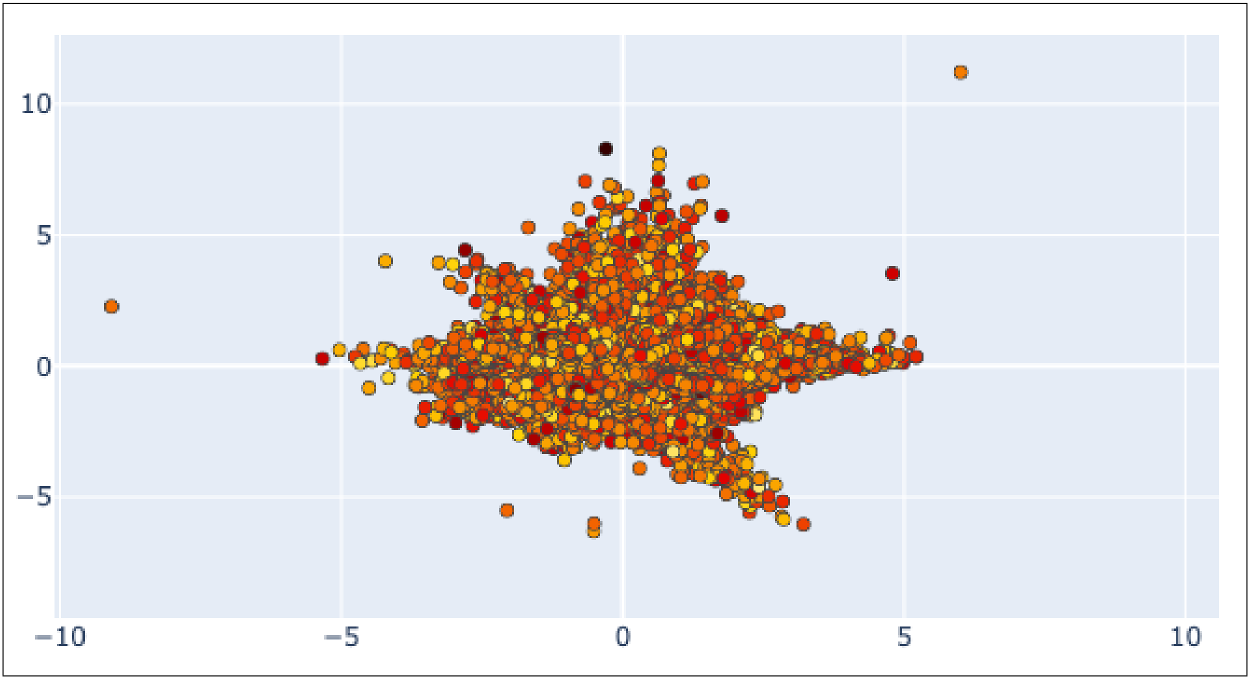


Figure 6: 2-dimensional representation of latent space.

By performing supervised learning on the characters and their respective one-hot vectors produced by us by above clustering methods using another convolutional neural network, we can obtain a network which would help us retrieve stroke information given any new character, since any new character would contain some similar parts in them.

I encountered a problem in clustering. As visible in Fig 6 , there are no noticeable clusters. On further zooming in, you can find out there do exist clusters, but they do not have discrete boundaries, rather they share their boundaries with other clusters, indicating similarity between strokes.

To check if clusters are formed in higher dimensional representations, I trained the VAE network with latent dimensions 3, which would give us a 3 dimensional representation of stroke data. Fig 7 shows us our 3d plot in a blue colour.

When you zoom in and rotate the 3d plot, there a few more clusters visible compared to the 2d plot. Though this does not help us in our task, we have atleast been successful in observing few clusters, which means we are still on the right track. Great news for us (still not that great though)!

Now we proceed with clustering, using the 2d representation. We use K-Means clustering here, and the results of clustering are displayed in Fig 8 .

I make one-hot encoded vectors for each of the characters in the dataset. I then try to train the CNN. The network does not train properly, even after the losses converging. The reasons for this have been studied upon and mentioned in the future scope of the project.

Technical Stuff

I have consistently used PyTorch [6] for all the deep learning related part of the project. I have mentioned the use of the SciPy library before for the use of spline creation and interpolation of data. Matplotlib was used along with Plotly for plotting of graphs. Numpy was used very frequently for handling of tensors on base levels.

Stroke data was obtained from the authors repository of the Omniglot dataset in a .mat file. We converted that to usable format using the scipy.io module.

The Variational Autoencoder model was made of only Linear layers (vanilla flavor). Encoder consisted of $[50, 1000, 1000, 2 * \mathit{dimension}]$ units layerwise, decoder mirrored the encoder. We used the reparametrization trick [4] to generate the latent vector.

PyTorch supports two types of dataloaders - Map-style and Iterable form. Since data was in dictionary format here, mainly non-serial order, I use the torch.utils.data.IterableDataset base class for creating custom datasets.

For the lastmost network, as I have mentioned earlier, I use a CNN (plain, nothing residual) . The convolutional block is described below, with a ReLU activation between each layer. There are three Conv2d layers, each having feature maps $[128, 128, 128]$, with stride 1 and 3×3 kernels. Then I use a Maxpooling layer with stride 2 and kernel

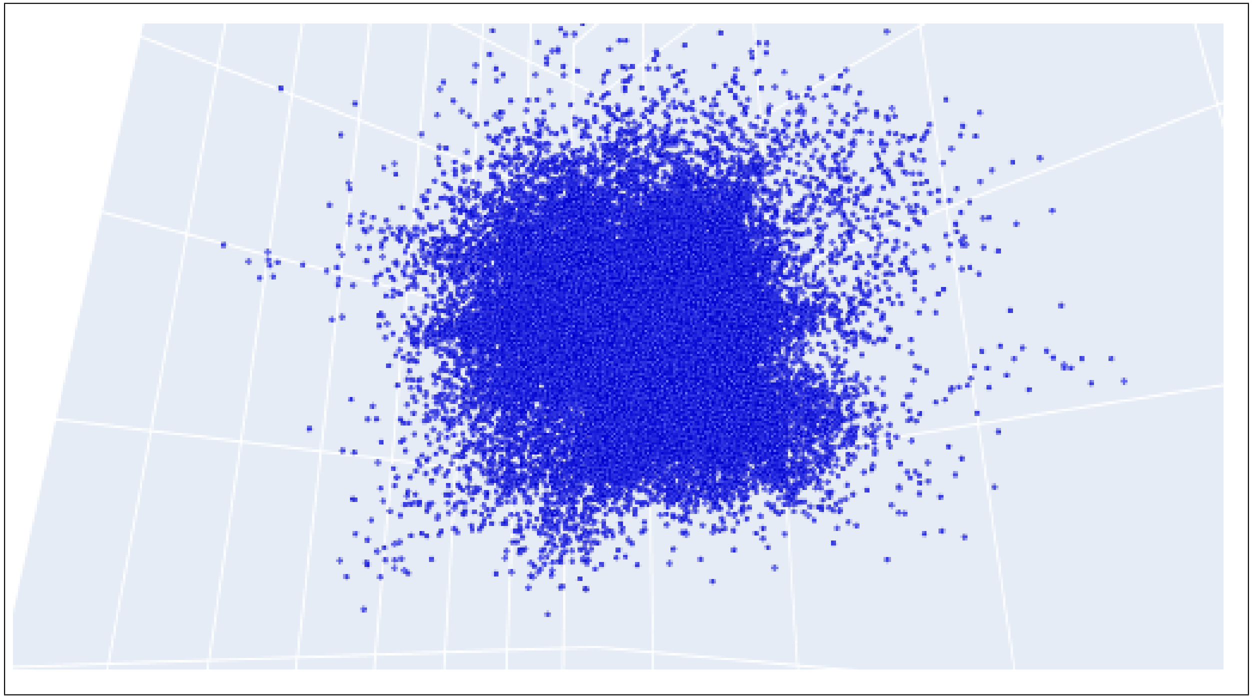


Figure 7: 3-dimensional representation of latent space.

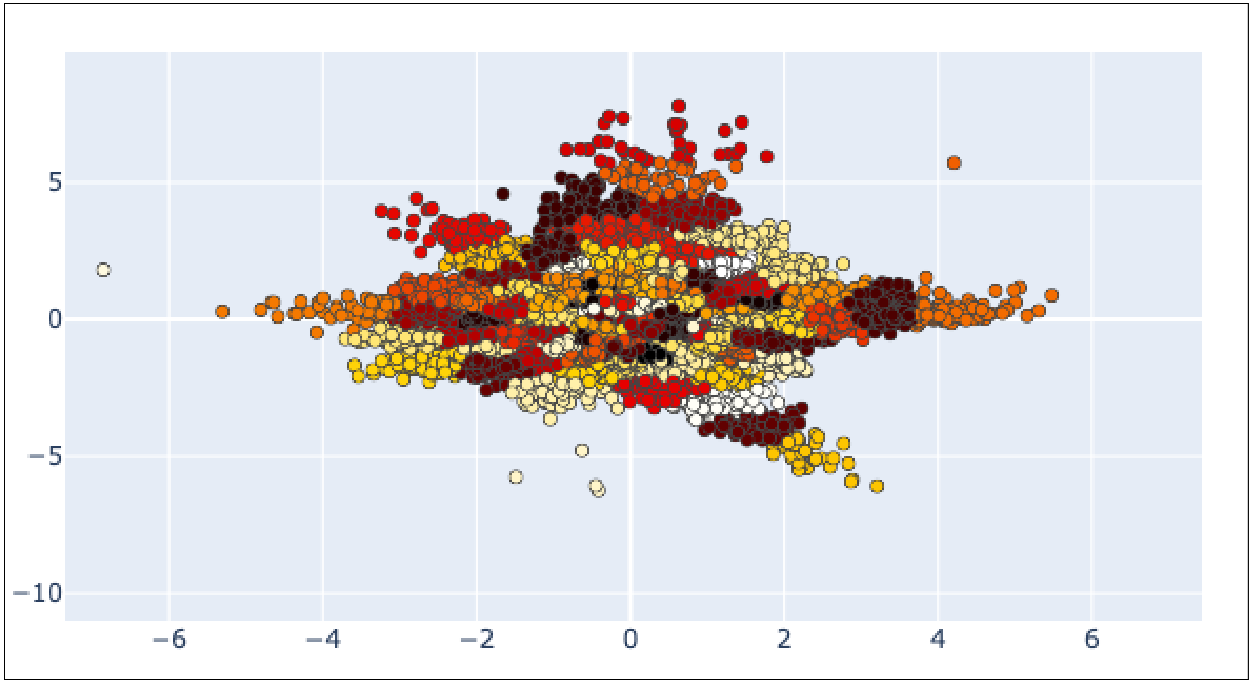


Figure 8: 2-dimensional clusters.

size 2×2. Fully connected layers follow with these many units: $[3000, 300]$. I use the Adam optimizer [8] with a learning rate of 10^{-3}.

Future Scope

We realize now the main problem lies in clustering. We hypothesised that characters of the same class should have had same one-hot vectors. This was supposedly proven false, and we observed variation across one-hot vectors across different instances of the same character (after we clustered into 300 strokes). This caused unsuccessful training of the last CNN we were supposed to train.

Now how do we rectify this problem?
There exist variations in VAE models which define much better latent space clusters. [9] introduces a discriminator network which matches the aggregated posterior of the hidden variable to an arbitrary prior distribution. This ensures that generating samples from any part of the pre-defined distribution would ensure results.

But we think the problem does not lie here. The problem lies in the fact that we were taking strokes as a whole. There exist sub-part in strokes too, which act like primitives. These primitives (for example a left-half curve, a simple vertical line) form the larger strokes. Therefore, using these primitives for clustering would ensure better results, as there may be a very large number of strokes which exist (combinations of primitives). This has also been observed by [1], who constructed their model on the basis of these primitives.

Another improvement we can make is making the use of heat-maps, as popularly used by posenet [10]. But the main problem we would encounter in heat-maps would be the large number of primitives that exist, which would make it computationally expensive.

After obtaining correct clusters, and therefore one-hot vectors for characters, we would expect our CNN to train successfully.

We can then use this CNN for similarity tasks, as well as classifcation tasks.

The latest model repo can be found here -
https://github.com/SomTambe/omniglot-bcs

2 One-Shot Learning using Siamese Network

Preface

The implementation for this model has been carried on by Nikita Chauhan. The model has been completed but the performance of the model is not good.

Approach

I have trained the model on a subset of the Omniglot dataset, it is one of the most popular dataset used for one-shot learning. It is specially designed to compare and contrast the learning abilities of humans and machines. This dataset consists of handwritten characters of 50 languages (alphabets) with 1623 total characters. There are only 20 samples for each character, each drawn by a distinct individual. The dataset is divided into 2 sets: background set and evaluation set. Background set contains 30 alphabets (964 characters) and is used to train the model whereas the remaining 20 alphabets are for pure evaluation purposes only.

To develop a model for one-shot image classification, I have implemented a neural network that can discriminate between the class-identity of image pairs, which is the standard verification task for image recognition. The verification model learns to identify input pairs according to the probability that they belong to the same class or different classes. This model can then be used to evaluate new images, exactly one per novel class, in a pairwise manner against the test image.

The pairing with the highest score according to the verification network is then awarded the highest probability for the one-shot task. First, I have trained it on the training set so that the verification model can learn the features to become sufficient to confirm or deny the identity of characters from one set of alphabets, then it ought to be sufficient for other alphabets, provided that the model has been exposed to a variety of alphabets to encourage variance amongst the learned features.

Model Architecture and Training

For this work, we first implemented basic architecture of Siamese Networks and Matching Networks using sigmoid Activation function. The Siamese network is an architecture with two parallel layers. In this architecture, instead of a model learning to classify its inputs using classification loss functions, the model learns to differentiate between

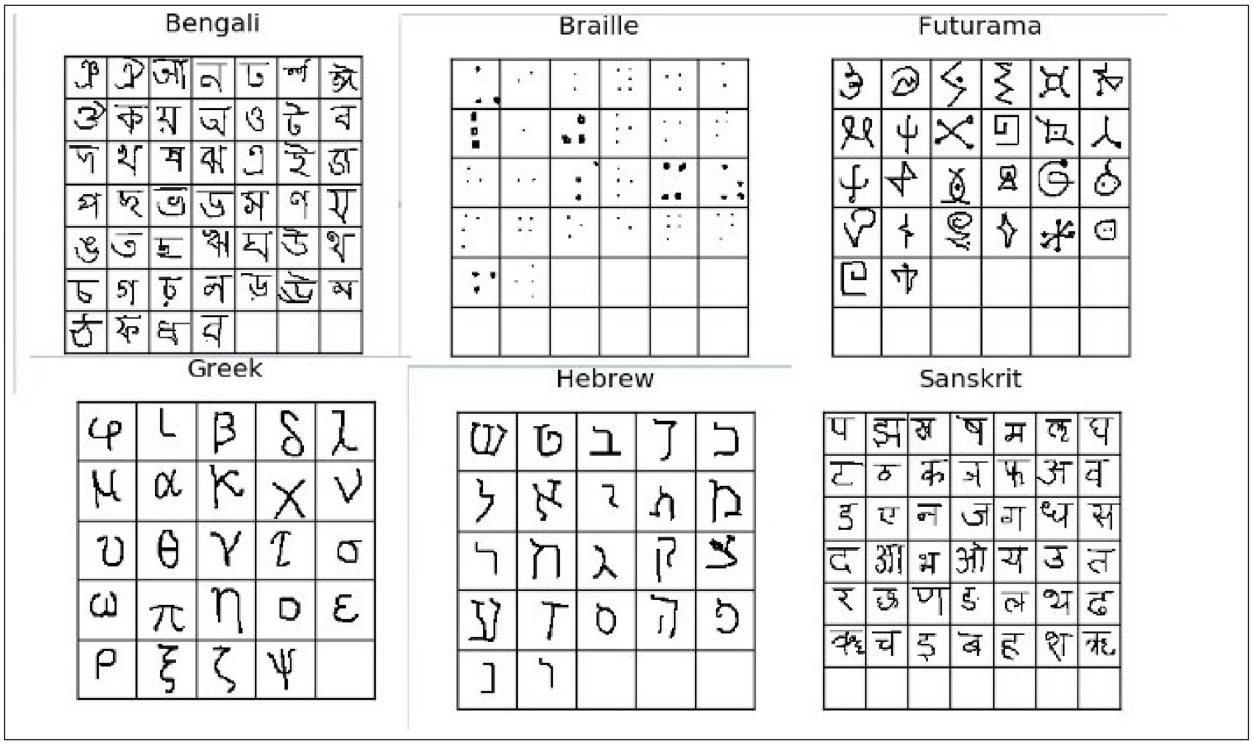


Figure 9: The Omniglot dataset contains a variety of different images from alphabets across the world.

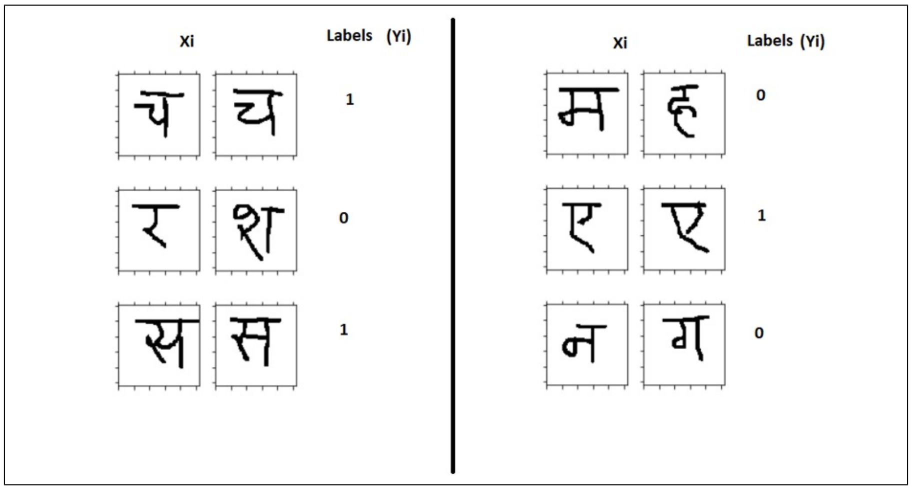


Figure 10: Sample of 6 data points

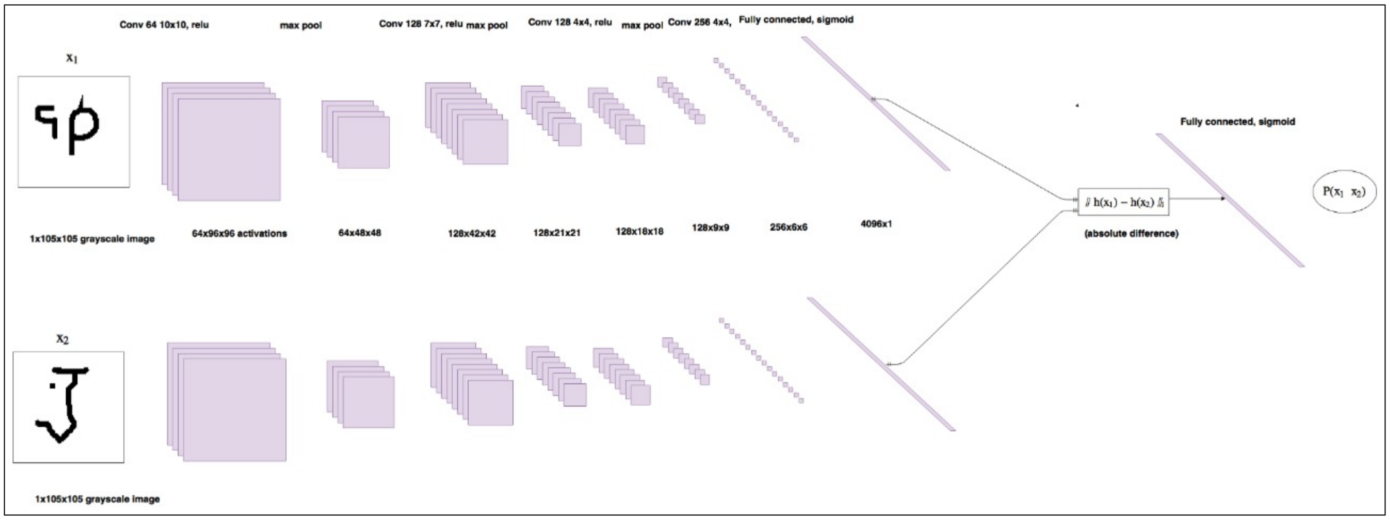


Figure 11: A high level architecture

two given inputs. It compares two inputs based on a similarity metric, and checks whether they are the same or not. Similar to any deep learning architecture, a Siamese network also has two phases: Training and Testing Phase.

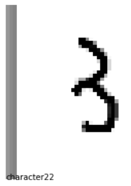
But usually for a One-shot learning approach (as we won't have a lot of data points) we train the model architecture on a different dataset and test it for our less amount of dataset.

For the feature extraction,I have used transfer learning(i.e pre-trained network on imagenet), that allows the network to train faster as it doesn't learn from scratch. And the units in the final convolutional layer are flattened into a single vector. The vgg16 is followed by a fully-connected layer, and then one more layer computing the induced distance metric between each siamese twin, which is given to a single sigmoidal output unit.

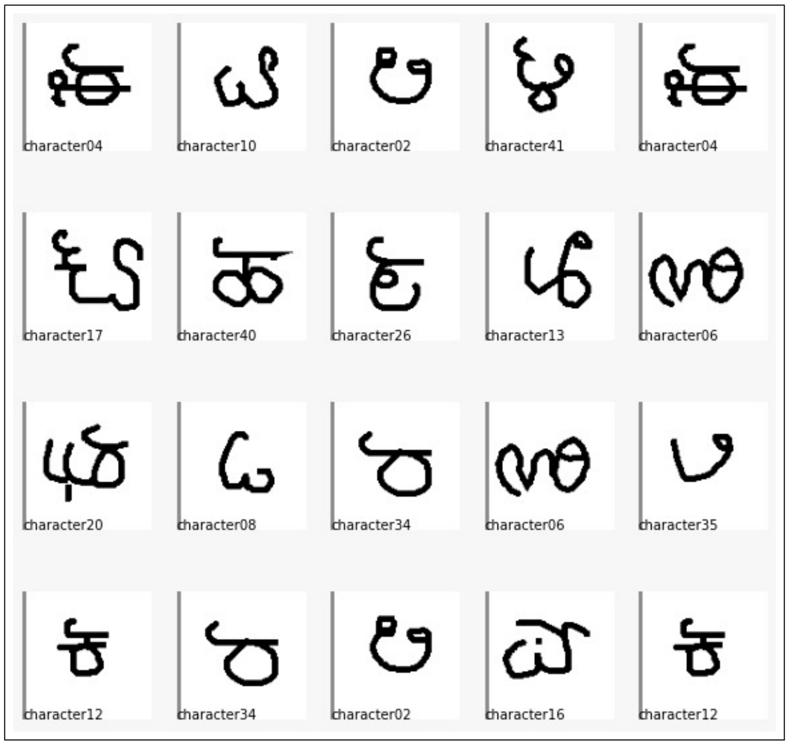
After optimizing the siamese network to master the verification task, then I tried to demonstrate the discriminative potential of our learned features at one-shot learning. To empirically evaluate one-shot learning performance, Lake developed a 20-way within-alphabet classification task in which an alphabet is first chosen from among those reserved for the evaluation set, along with twenty characters taken uniformly at random. Two of the twenty drawers are also selected from among the pool of evaluation drawers. These two drawers then produce a sample of the twenty characters. Each one of the characters produced by the first drawer are denoted as test images and individually compared against all twenty characters from the second drawer, with the goal of predicting the class corresponding to the test image from among all of the second drawer's characters. An individual example of a one-shot learning trial is depicted in Figure 12.

Conclusion

I have trained the model using a tensorflow backend in Keras.I have not been able to reproduce the results reported by the authors (>90% in the evaluation set). I was able to get results in the order of 50%. Probably this difference is because I didn't implement many of the performance enhancing tricks from the original paper, like layerwise learning rates/momentum, data augmentation with distortions, bayesian hyperparemeter optimization and I also probably trained for less epochs. Currently I'm working on how to improve the performance of the model.



character22



character04
character10
character02
character41
character04
character17
character40
character26
character13
character06
character20
character08
character34
character06
character35
character12
character34
character02
character16
character12

Figure 12

Implementing SOTA ML Models for Text Generation on the 'Omniglot' dataset

Anmol Pabla
apabla@iitk.ac.in
Department of Mechanical Engineering

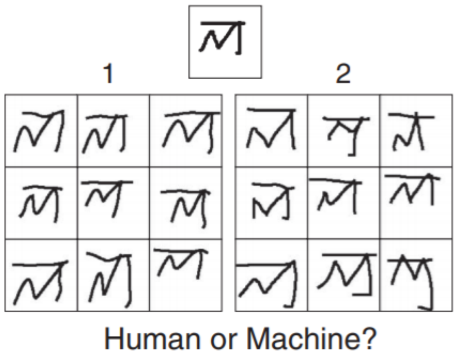
Mohit Kulkarni
mohitmk@iitk.ac.in
Department of Mathematics

Introduction

The main objective of this section is comparing Bayesian Program Learning Model with SOTA Machine Learning models for tasks other than classification on the 'Omniglot' dataset. The link to the Github Repo is:

`https://github.com/bcs-iitk/omniglot-project/tree/master`

The task for generation in the original Omniglot Challenge compared the creative outputs produced by humans and machines through "visual Turing tests", where naive human judges tried to identify the machine, given paired examples of human and machine behavior. In the most basic task, judges compared the drawings from nine humans asked to produce a new instance of a concept given one example with nine new examples drawn by the BPL/ML model.

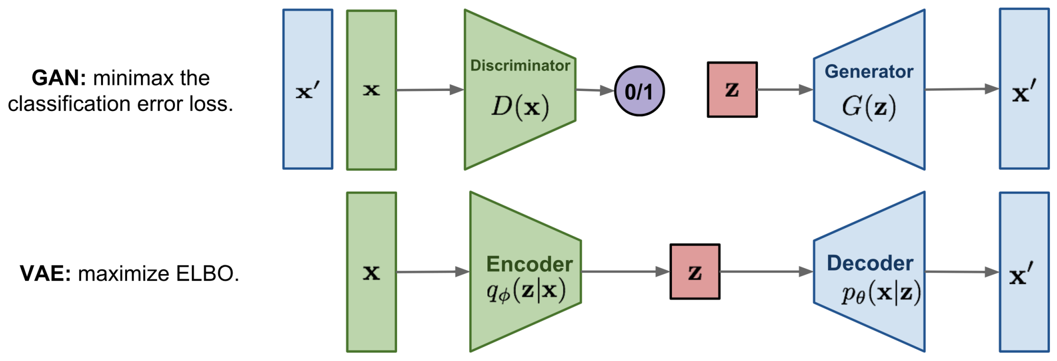


The judges were now asked to distinguish between examples drawn using BPL/ML model and those drawn by humans. The model was evaluated based on the accuracy of the judges, which is termed their identification (ID) level: Ideal model performance is 50% ID level, is indicating that they cannot distinguish the model's behavior from humans; worst-case performance is 100% suggesting the judges can clearly determine the images drawn by the model against a human.

The major demerit of the BPL model is that it is very specific to the Omniglot dataset and would not work for other images. The target of the comparison sub-team is to create SOTA generative ML models that give similar results to the BPL model.

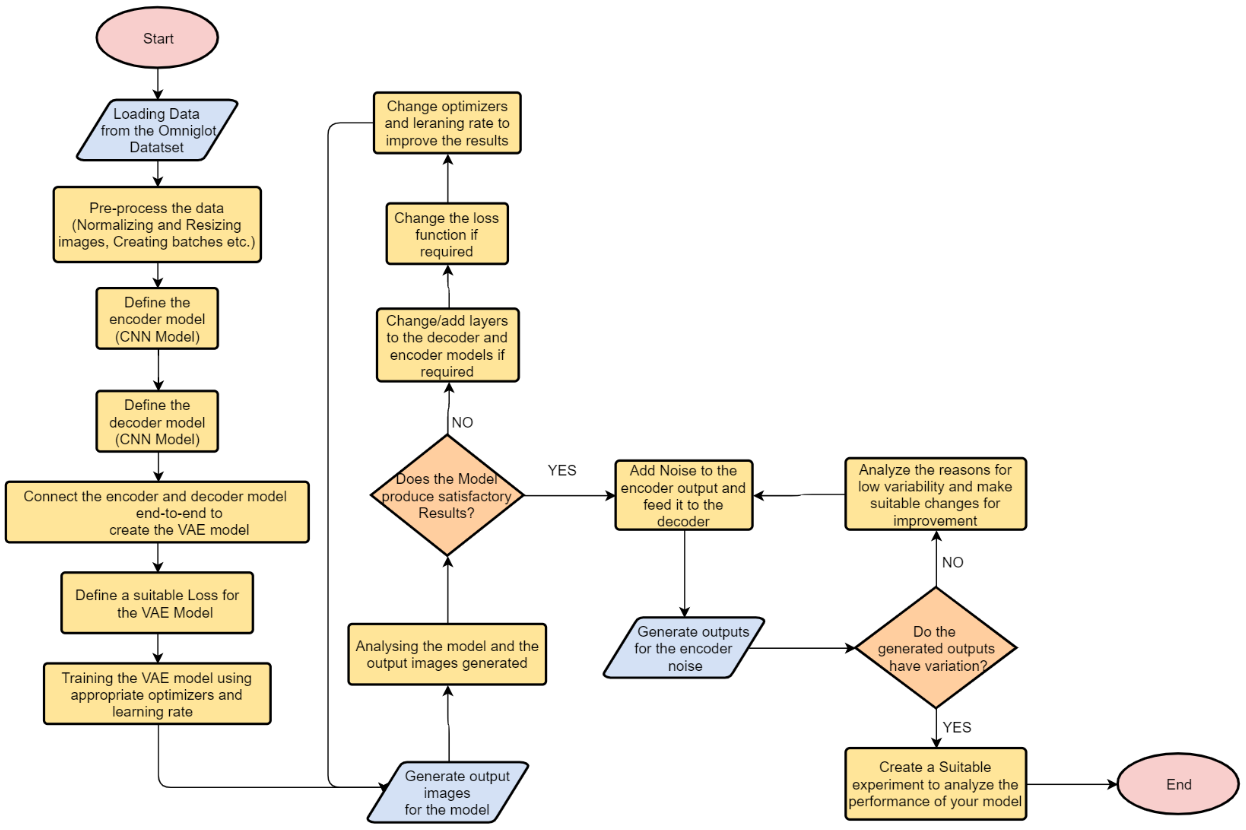
Implemented Models

In the last few years, deep learning based generative models have gained more and more interest. Among these deep generative models, two major families stand out and deserve a special attention: Generative Adversarial Networks(GANs) and Variational Autoencoders (VAEs). The VAE architecture is intuitive and simple to understand, it is a deep learning technique for learning latent representations. They have also been used to draw images, achieve state-of-the-art results in semi-supervised learning. Generative Adversarial Networks(Goodfellow et. al) are dubbed as one of the coolest models in Machine Learning. A GAN can be trained to generate images from random noises.



VAE -

Autoencoders are a type of neural network that can be used to learn efficient codings of input data. Given some inputs, the network first applies a series of transformations that map the input data into a lower dimensional space. This part of the network is called the encoder. Then, the network uses the encoded data to try and recreate the inputs. This part of the network is the decoder. Using a general autoencoder, we don't know anything about the coding that's been generated by our network. We could compare different encoded objects, but it's unlikely that we'll be able to understand what's going on. This means that we won't be able to use our decoder for creating new objects. For this reason we use Variational AutoEncoders. In VAEs, we simply tell our network what we want this distribution to look like.


Start
Loading Data from the Omniglot Datatset
Pre-process the data (Normalizing and Resizing images, Creating batches etc.)
Define the encoder model (CNN Model)
Define the decoder model (CNN Model)
Connect the encoder and decoder model end-to-end to create the VAE model
Define a suitable Loss for the VAE Model
Training the VAE model using appropriate optimizers and learning rate
Generate output images for the model
Analysing the model and the output images generated
Does the Model produce satisfactory Results?
NO
Change/add layers to the decoder and encoder models if required
Change the loss function if required
Change optimizers and leraning rate to improve the results
YES
Add Noise to the encoder output and feed it to the decoder
Generate outputs for the encoder noise
Do the generated outputs have variation?
NO
Analyze the reasons for low variability and make suitable changes for improvement
YES
Create a Suitable experiment to analyze the performance of your model
End

Implementation -

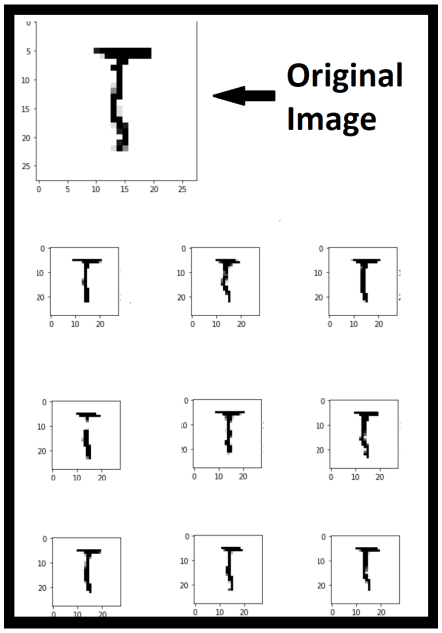
We feed the images into the encoder where it passes through several layers and it ultimately leads to two layers, representing mean and standard deviation which is then used to create a distribution. Then, the network uses the encoded data to try and recreate the inputs using the decoder. Usually, we will constrain the network to produce latent vectors having entries that follow the unit normal distribution. Then, in the final step when trying to generate data we simply feed data to the decoder from this distribution and the decoder will return us completely new objects that appear just like the objects our network has been trained with. It can be more clearly understood using the flowchart above which describes the overall process.

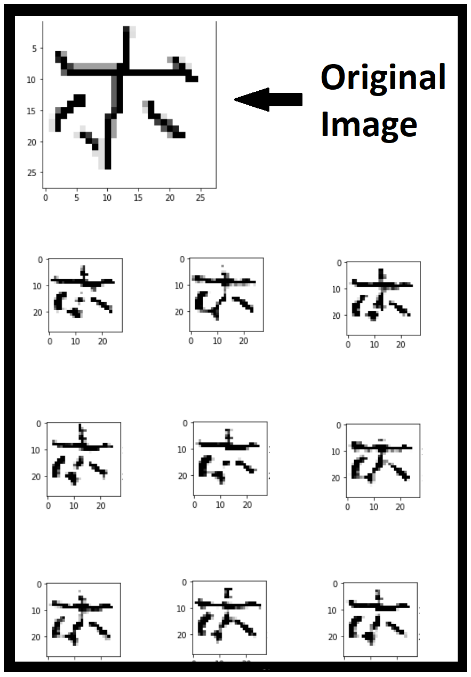
GAN -

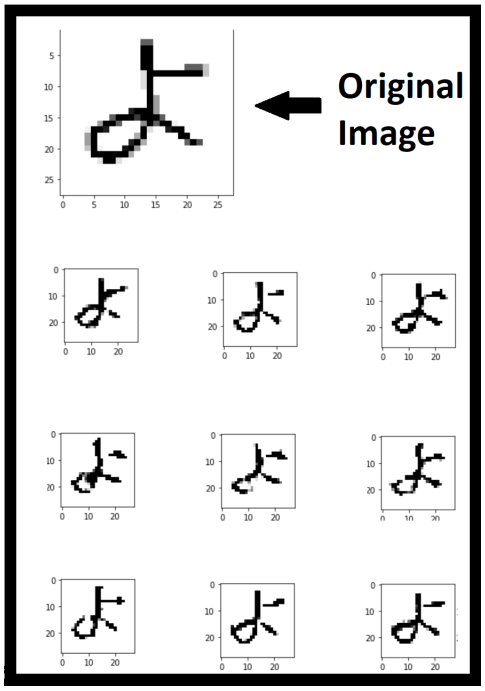
GANs are basically two neural networks Generative network, which generates candidates, and discriminative network, which evaluates them. The generative model is typically a deconvolutional neural network, and the discriminator is a convolutional neural network. The generative network tries to fool the discriminator network by producing images similar to input space. We Started of by using Deep Convolutional GANs for omniglot generative task but found it to be quickly overfitting to the omniglot Dataset due to very less number of samples per class. Data Augmentation was used to no avail

Results

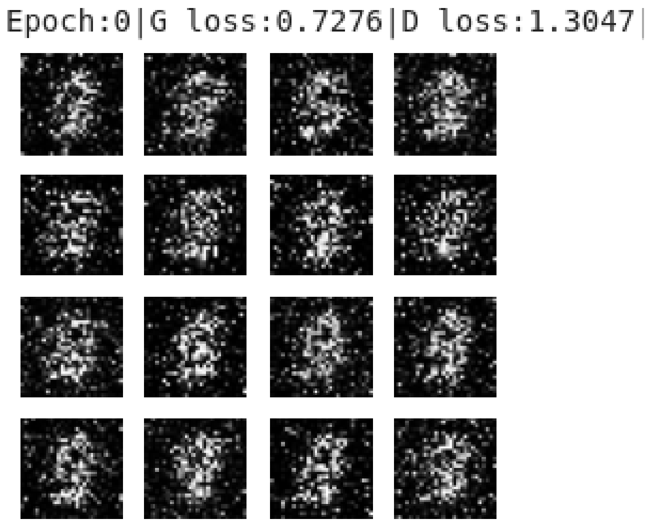
The VAE model was first implemented on the MNIST dataset and then on the Omniglot dataset, the model uses ELBO loss. Several steps were taken to improve the results and also add variability to the output generated by the model, these steps include data augmentation and adding noise to the encoder output and then feeding it to the decoder. The output images are 28x28px and have decent accuracy and variability, therefore it can be expected that the model would give satisfactory results when the original task is performed with human judges. Some of the images generated by the model for three characters are given below:

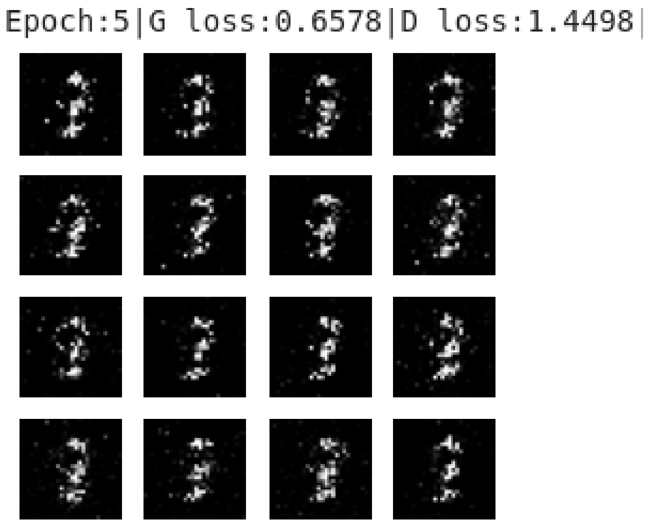






DCGAN was first implemented to the MNIST dataset and then tried on the Omniglot Dataset. We found that after about 5 epochs, the network starts overfitting rapidly.





Epoch:14|G loss:0.1507|D loss:2.1553|

We then tried to find GAN architectures specially built for few shot learning tasks or more specifically for the omniglot dataset. MetaGANs was found to be promising in this category where the generative model is based on another implemented paper MAML. Current work is implementation of MAML paper. This can then be generalised towards MetaGANs.

Future Targets and Discussion

The main target for the future would be to perform the original task for our model with human judges and participants and compare the results with the BPL model. We would attempt to improve the results for both the models and also try and implement other SOTA ML models for generation.

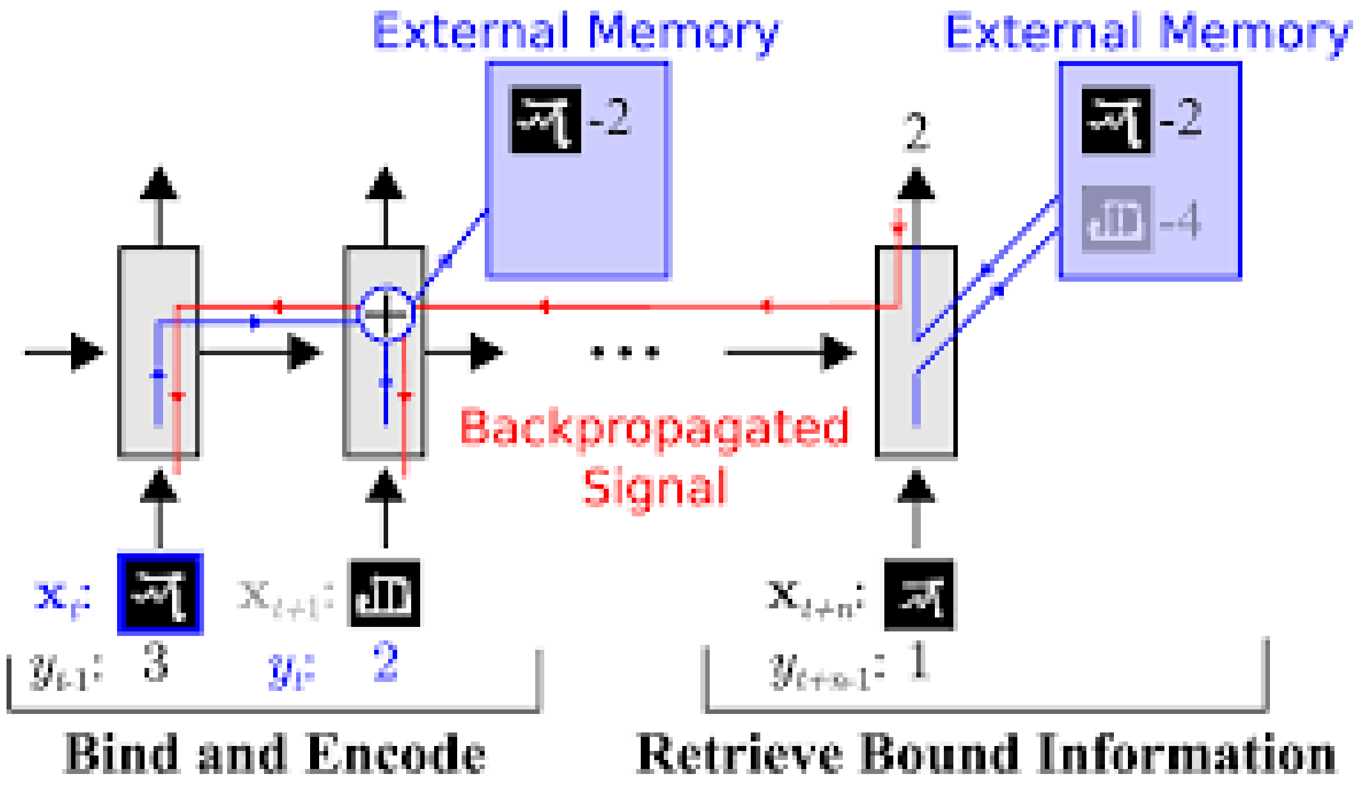
Memory Augmented Neural Network for Few Shot Classification on 'Omniglot' Dataset

Mohit Kulkarni
`mohitmk@iitk.ac.in`
`Department of Mathematics`

The main objective of this subteam was to implement the paper Meta Learning with memory augmented neural network(Santoro et. al).:

`https://github.com/bcs-iitk/omniglot-project/tree/master`

Memory augmented neural networks are based on Neural Turing machines which serve as external memory for our neural network model. The network has the ability to read from selected memory locations and ability to write to selected locations, which makes it a perfect candidate for low-shot predictions.



Implementation

LSTM Based

In this approach we try to use basic deep learning models to work for low latent dataset. Optimization based approaches such as given by Ravi & Larochelle is to efficiently update the learner's parameters using a small support set so that the learner can adapt to the new task quickly. The meta-learner is modelled by a LSTM because 1. There is similarity between the gradient-based update in backpropagation and the cell-state update in LSTM. 2. Knowing a history of gradients benefits the gradient update. Similar to Momentum in ADAM optimizer. Other papers such as MAML also use this type of optimization based update to solve meta-learning problem

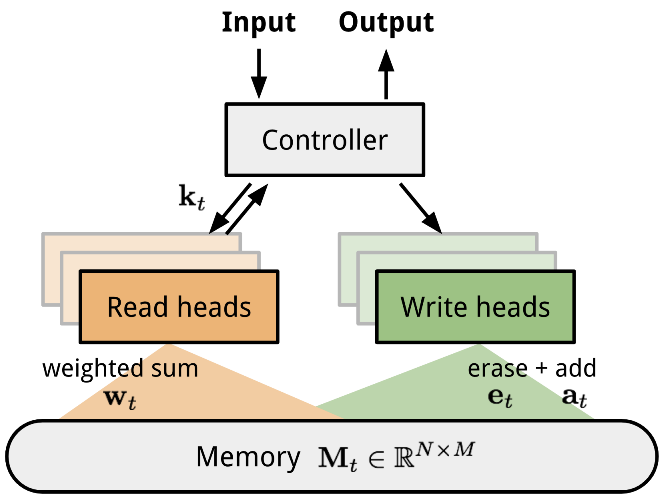
Algorithm 1 Train Meta-Learner

Input: Meta-training set $\mathscr{D}_{meta-train}$, Learner M with parameters θ, Meta-Learner R with parameters Θ.

```
 1: $\Theta_0 \leftarrow$ random initialization
 2:
 3: for $d = 1, n$ do
 4:     $D_{train}, D_{test} \leftarrow$ random dataset from $\mathscr{D}_{meta-train}$
 5:     $\theta_0 \leftarrow c_0$                                          ▷ Intialize learner parameters
 6:
 7:     for $t = 1, T$ do
 8:         $\mathbf{X}_t, \mathbf{Y}_t \leftarrow$ random batch from $D_{train}$
 9:         $\mathcal{L}_t \leftarrow \mathcal{L}(M(\mathbf{X}_t; \theta_{t-1}), \mathbf{Y}_t)$          ▷ Get loss of learner on train batch
10:         $c_t \leftarrow R((\nabla_{\theta_{t-1}} \mathcal{L}_t, \mathcal{L}_t); \Theta_{d-1})$   ▷ Get output of meta-learner using Equation 2
11:         $\theta_t \leftarrow c_t$                                  ▷ Update learner parameters
12:     end for
13:
14:     $\mathbf{X}, \mathbf{Y} \leftarrow D_{test}$
15:     $\mathcal{L}_{test} \leftarrow \mathcal{L}(M(\mathbf{X}; \theta_T), \mathbf{Y})$               ▷ Get loss of learner on test batch
16:     Update $\Theta_d$ using $\nabla_{\Theta_{d-1}} \mathcal{L}_{test}$              ▷ Update meta-learner parameters
17:
18: end for
```

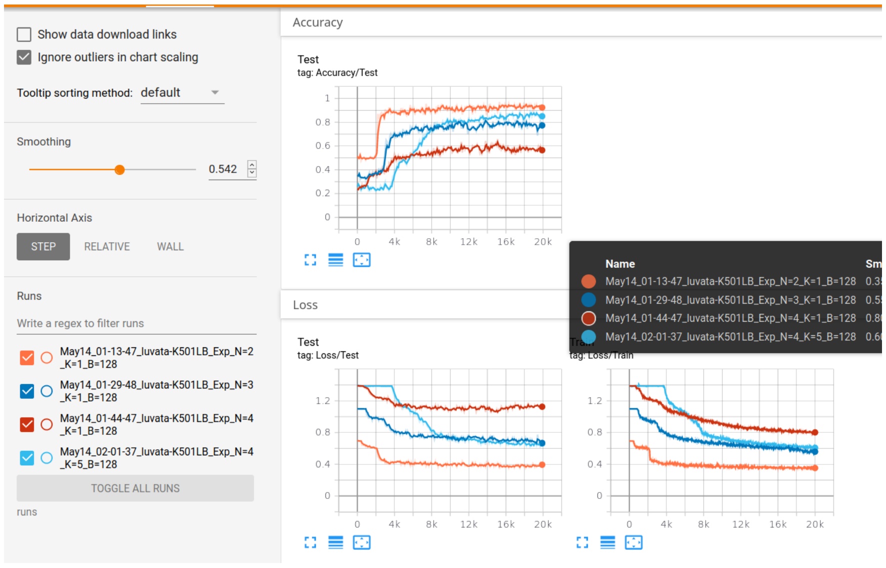
NTM and MANN Based

NTM is composed of a neural network which has the ability to store and retrieve information from the memory. It basically has three components- controller, memory and read and write heads. Memory augmented neural network (MANN) is a variant of NTM which is extensively used for one-shot learning it is designed to make NTM perform better at one-shot learning tasks. NTM uses content and location based lookup to address the memory while MANN uses an access module called LRUA.



Results

We ran the Memory augmented model for various parameters of N(Number of classes) and K(number of sampled images per class). Below tensorboard image shows the Test Accuracy, Test loss and train loss. We can see that our model has a slightly lower accuracy than what was given in the original paper, This can be accounted for the unavailability of the correct train and test split modes. Using a constant batch size of 128, we can see that our model gives a accuracy of 56% on N=4 and K=1 training.



Runs:
- May14_01-13-47_luvata-K501LB_Exp_N=2_K=1_B=128
- May14_01-29-48_luvata-K501LB_Exp_N=3_K=1_B=128
- May14_01-44-47_luvata-K501LB_Exp_N=4_K=1_B=128
- May14_02-01-37_luvata-K501LB_Exp_N=4_K=5_B=128

Name	Sm
May14_01-13-47_luvata-K501LB_Exp_N=2_K=1_B=128	0.3
May14_01-29-48_luvata-K501LB_Exp_N=3_K=1_B=128	0.5
May14_01-44-47_luvata-K501LB_Exp_N=4_K=1_B=128	0.8
May14_02-01-37_luvata-K501LB_Exp_N=4_K=5_B=128	0.6

Future Targets and Discussion

The next main target would be to try and improve on the existing model. this can be done in various ways

- Use of bi-directional LSTM networks
- Try to make the optimization process faster using other optimization models such as MAML

MAML is a paper that is very interesting and would definitely be the next step in few-shot learning process.

Implementation of Bayesian Program Learning Model in Python to Develop One-Shot Learning for the ’Omniglot’ Dataset

Anmol Pabla
apabla@iitk.ac.in
Department of Mechanical Engineering

Vaibhav Thakkar
vaithakk@iitk.ac.in
Department of Electrical Engineering

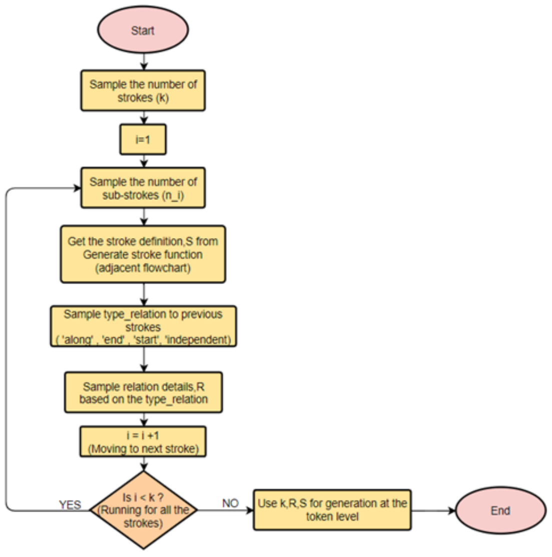
Introduction

Bayesian program learning (BPL) is one of the most used approaches to one-shot learning. It attempts to learn tasks using much lesser amount of data than conventinal ML models. The main objective of this section is the Replication of BPL in python for ’Omniglot’ dataset for tasks like classification and generation. The Omniglot dataset contains of several handwritten characters from about 50 different languages. This section’s main target is to replicate the BPL Model in Python for the Omniglot Dataset. The link to the Github Repo is:

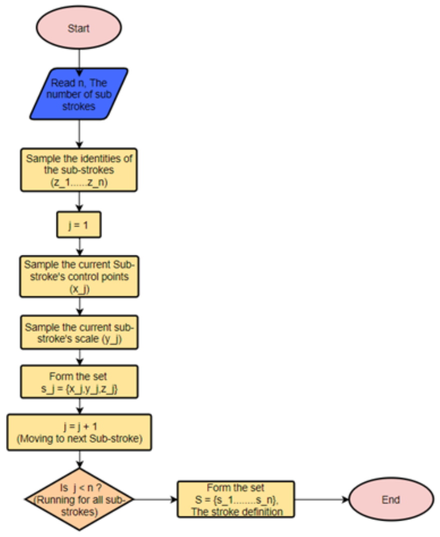
`https://github.com/bcs-iitk/omniglot-project/tree/master`
This section is divided mainly into three parts:

Generating Character Types-

In this section, the model generates new character types based on the information it has collected from the Omniglot dataset. A character type Ψ is defined by the parameters: **k**, the number of strokes; **S** the set of strokes, each stroke is defined by pressing the pen down and terminated by lifting the pen up, each stroke (S_i) is further divided into sub-strokes which are distinguished by the pauses while writing a stroke and by **R**, the relation set which defines how each stroke is related to other strokes i.e. R_i defines how the beginning of stroke S_i is related to $\{S_1,...,S_{i-1}\}$. Relations are classified into four types, $_i \in \{Independent, Start, End, Along\}$. The flowchart for the processes of Type Generation and stroke generation given below would help further understanding:


Start
Sample the number of strokes (k)
i=1
Sample the number of sub-strokes (n_i)
Get the stroke definition,S from Generate stroke function (adjacent flowchart)
Sample type_relation to previous strokes ('along' , 'end' , 'start', 'independent)
Sample relation details,R based on the type_relation
i = i +1 (Moving to next stroke)
Is i < k ? (Running for all the strokes)
YES
NO
Use k,R,S for generation at the token level
End

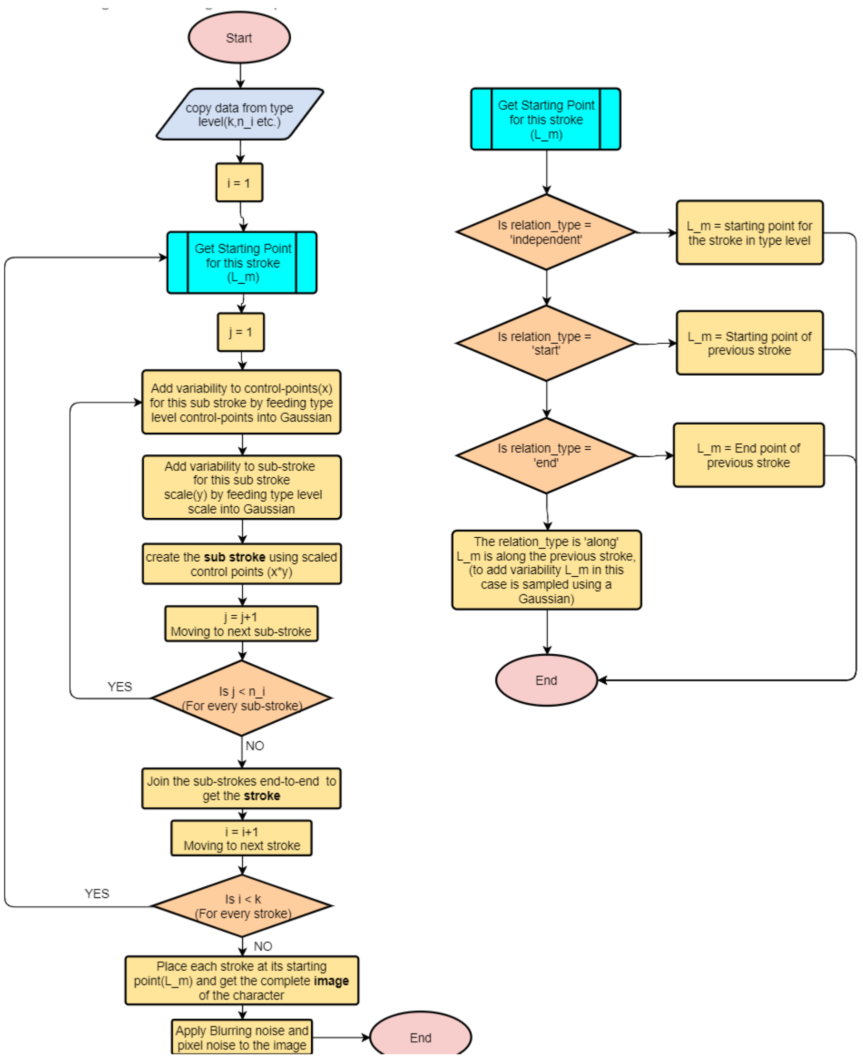
Flowchart for Type-Level Generation


Start
Read n, The number of sub strokes
Sample the identities of the sub-strokes (z_1......z_n)
j = 1
Sample the current Sub-stroke's control points (x_j)
Sample the current sub-stroke's scale (y_j)
Form the set s_j = {x_j,y_j,z_j}
j = j + 1 (Moving to next Sub-stroke)
Is j < n ? (Running for all sub-strokes)
Form the set S = {s_1........s_n}, The stroke definition
End

Flowchart for Stroke Generation

Generating Character Tokens-

For token-level generation we use control points x_{ij}^{m} and scale y_{ij}^{m} which are sampled from x_{ij} and y_{ij} obtained form the type level. This causes them to be slightly noisier than their type-level counterparts to generate the spline which represents the sub-stroke, several sub-strokes are combined to form strokes which are then combined based on spatial relations R_m which is directly copied from the type level relations to form characters, the beginning of stroke is placed at L_m, the starting location of stroke.Finally, some noise is added to the character image to get the final binary image.



Start
copy data from type level(k,n_i etc.)
i = 1
Get Starting Point for this stroke (L_m)
j = 1
Add variability to control-points(x) for this sub stroke by feeding type level control-points into Gaussian
Add variability to sub-stroke for this sub stroke scale(y) by feeding type level scale into Gaussian
create the sub stroke using scaled control points (x*y)
j = j+1 Moving to next sub-stroke
YES
Is j < n_i (For every sub-stroke)
NO
Join the sub-strokes end-to-end to get the stroke
i = i+1 Moving to next stroke
YES
Is i < k (For every stroke)
NO
Place each stroke at its starting point(L_m) and get the complete image of the character
Apply Blurring noise and pixel noise to the image
End
Get Starting Point for this stroke (L_m)
Is relation_type = 'independent'
L_m = starting point for the stroke in type level
Is relation_type = 'start'
L_m = Starting point of previous stroke
Is relation_type = 'end'
L_m = End point of previous stroke
The relation_type is 'along' L_m is along the previous stroke, (to add variability L_m in this case is sampled using a Gaussian)
End

Flowchart for token generation

Learning to Learn motor programs-

The aim of this part is to learn various parameters of BPL framework that are required for generating token types and then generating characters from the token type generated.

Learning Primitives Primitives are partitions/family/clusters of substrokes used in the data set, the assumption is that each partition/cluster of substrokes has a multivariate Gaussian Distribution with it's own mean and covariance matrix. This model is known as the Gaussian Mixture Model (GMM), it's parameters are learn using EM algorithm, which is a standard algorithm with implementations available in various libraries. After partitioning, the parameters μ_z and Σ_z are estimated for each partition, which are used for sampling control points $x|z$, similarly α_z and β_z for sampling the scale $y|z$.

But the above model does not incorporate the order of substrokes, so we also have a Dicrete markov model for that which has transition probability $P(z_i|z_j)$, which is learned by empirical count of each substroke.

Learning Start positions For each 'Independent' stroke you can have a different starting position, so for that we have a probability associated with each cell that tells how likely it is that the cell is the starting position for a stroke, this is a multinomial model for which the parameters are estimated by empirical frequency.

There are some complex hyperparameters for this, like how small should be a cell, smoothing and aggregating threshold that will be learn by cross validation set.

Learning Relations and Token variability The relation parameter θ_R is estimated by empirical frequency count only, which was found to be $(34\%, 5\%, 11\%, 50\%)$ for (Independent, start, end, along) respectively by the authors.

For other parameters $(\Sigma_l, \sigma_x, \sigma_y, \sigma_\tau)$ which are mainly the variance parameters for adding noise in generating character tokens, I found the explanation for learning these parameters very vague in the paper, will try to understand it from their code-base.

Learning Image Parameters Image parameters are a quadruple $(r_x, r_y, t_x, t_y) \sim N([1, 1, 0, 0], \Sigma_A)$, which are also fit in an empirical way, for we only need to estimate covariance matrix but for doing that we need to ensure the mean vector is [1,1,0,0] so we translate each image's centre of mass to the average centre of mass of all images, and similarly scale each image to average range of all images.

Hyperparameters a and b are fixed, (a=0.5, b=6).

Results

BPL is a fairly new concept, the sub-team covered a fair amount of theory for implementation of a BPL model to the Omniglot dataset. The basic structure for the BPL class has been implemented and some of the functions for the class have also been implemented. Major portion of the token generation part have been completed, some changes are yet to be made to ensure that the token generation function is compatible with the type level and the basic class structure.

Future Targets

The primary target would be to complete the BPL model by implementing its code in Python. We would then perform the original tasks mentioned in the Omniglot Challenge using human judges and try to compare the results of the BPL model with the Deep Learning models.
A major demerit of BPL is that it is very specific to the dataset it is implemented upon, we would try to look into possible ways to improve this.

References

1.] *Lake, B. M., Salakhutdinov, R., and Tenenbaum, J. B. (2015).* Human-level concept learning through probabilistic program induction. Science, 350(6266), 1332-1338.

2.] *Lake, B. M., Salakhutdinov, R., and Tenenbaum, J. B. (2019).* The Omniglot Challenge: A 3-Year Progress Report. Preprint available on arXiv:1902.03477

3.] *LeCun, Yann, et al.* (1999): The MNIST Dataset Of Handwritten Digits.

4.] *Diederik P. Kingma and Max Welling* (2019), "An Introduction to Variational Autoencoders", Foundations and TrendsR in Machine Learning: Vol. xx, No. xx, pp 1–18. DOI: 10.1561/XXXXXXXXX.

5.] *Urs Ramer*, "An iterative procedure for the polygonal approximation of plane curves", Computer Graphics and Image Processing, 1(3), 244–256 (1972) doi:10.1016/S0146-664X(72)80017-0

6.] *Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, Soumith Chintala*, "PyTorch: An Imperative Style, High-Performance Deep Learning Library ", NeurIPS 2019, arXiv:1912.01703

7.] *Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, CJ Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E.A. Quintero, Charles R Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors.* (2020) SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. Nature Methods, in press.

8.]*DP Kingma, JA Ba*, A method for stochastic optimization - arXiv:1412.6980

9.] *A. Makhzani, J. Shlens, N. Jaitly, I. Goodfellow, and B. Frey.* Adversarial autoencoders. arXiv preprint arXiv:1511.05644, 2015.

10.] *Zhe Cao, Gines Hidalgo, Tomas Simon, Shih-En Wei, Yaser Sheikh*, OpenPose: Realtime Multi-Person 2D Pose Estimation using Part Affinity Fields - arXiv:1812.08008