ONE SHOT LEARNING: THE OMGNIGLOT CHALLENGE

TEAM MEMBERS

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

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

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

Shreyasi Prasad
Department of CSE
shreyap@iitk.ac.in

Nikita Chauhan
Department of CSE
nikitach@iitk.ac.in

Mukund soni
Materials Science and Engineerinng
mukunds@iitk.ac.in

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

Saksham Gupta
Mechanical Engineering
gsaksham@iitk.ac.in

May 25, 2020

BRAIN AND COGNITIVE SOCIETY

1 Replication

1.1 Objective

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.

1.2 Literature Survey

1.2.1 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 following parameters:

k gives the number of strokes and is sampled from a multinomial P(k) estimated from the empirical frequencies.

S gives 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. The number of substrokes n_i is sampled from empirical frequency P(n_i | k) specific to total strokes. Each sub-stroke (s_{ij}) is modeled as a uniform cubic b-spline, which can be decomposed into three variables { x_{ij} , y_{ij} , z_{ij} }. Here z_{ij} is the identity of sub-stroke.The five control points x_{ij} are sampled from a Gaussian P(x_{ij} | z_{ij}) = N($_{zij}$, Σ_{zij}), and they live in an abstract space. The type-level scale y_{ij} of this space, relative to the image frame, is sampled from P(y_{ij} | z_{ij}) = Gamma(a_{zij} , b_{zij}).

R defines 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, $\xi_i \in$ {Independent, Start, End, Along},

with probabilities θ_R.Each type has its own dimensionality:

Independent relations, where the position of stroke i does not depend on previous strokes. The variable J_i is drawn from P(J_i), a multinomial over a 2D image grid that depends on index i. Since the position L_i has to be real-valued, P(L_i |J_i) is then sampled uniformly at random from within the image cell J_i.

Start relations, where stroke i starts at the beginning of a previous stroke u_i, sampled uniformly at random from $u_i \in \{1, ..., \text{i}-1\}$.

End relations, where stroke i starts at the end of a previous stroke u_i, sampled uniformly at random from $u_i \in \{1, ..., \text{i}-1\}$.

Along relations, where stroke i begins along previous stroke $u_i \in \{1, ..., \text{i}-1\}$ at sub-stroke $v_i \in \{1, ..., n_{ui}\}$ at type-level spline coordinate τ_i, each sampled uniformly at random.

1.2.2 Generating character-tokens

For token-level generation we use control points $x_i{}_j^m$ and scale $y_i{}_j^m$ which are sampled from x_{ij} and y_{ij} using the Gaussians P($x_i{}_j^m$|x_{ij}) = N(x_{ij},σ_x^2) and P($y_i{}_j^m$|y_{ij}) = N(y_{ij},σ_y^2) respectively.

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, this is calculated as:

P(L_i^m|R_i^m,T_1^m, ..., $T_i^m{}_{-1}$)= N(g(R_i^m,T_1^m, ..., $T_i^m{}_{-1}$), Σ_L) where the function g(·) locates the stroke at an appropriate position depending on the relation. For Independent relations, g(·) = L_i where L_i is the type-level global location. For End relations (and analogously Start), g(·) = end($T_{u_i}^m$)

Finally, some noise (blurring σ_b^m and inking ϵ^m) is added to the character image based on some sampled distribution to get the final binary image.

1.2.3 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).

1.3 Implementation

1.3.1 Generating character-types

First we create function to generate stroke:

def generate-stroke(i, n_i) :
. z_{i1} <- P(z_{i1})
. for j in {2,...,n_i}:
. z_{ij} <- P($z_{ij}|z_{i(}j-i)$)
. for j in 1,...,n_i :
. x_{ij} <- P($x_{ij}|z_{ij}$)
. y_{ij} <- P($y_{ij}|z_{ij}$)
. s_{ij} = x_{ij}, y_{ij}, z_{ij}
. S_i = {s_{i1},...,s_{ini}}
. return S_i

Thereafter we create a function to generate character type:

def generate-type :
. k <- P(k)
. for i in range(k):
. n_i <- P(n_i| k)
. S_i= generate-stroke(i, n_i)
. ξ_i<- P(ξ_i)
. R_i <- P(R_i | ξ_i, S_1, S_2,...,$S_(i-1)$)
. Ψ= { k, S, R}
. return generate-token(Ψ)

1.3.2 Generating character-tokens

x^m=Control Points
y^m= Token-level scale
R^m= Token-level Relations
L^m= Starting Location
σ_b^m= Bluring Noise
ϵ_b^m= Probability of pixel being inked

In token-level generation for every stroke, we first directly copy the relations R^m from the type-level relations and we sample the starting location L^m for every stroke using a Gaussian function, then for every sub-stroke we sample the control points x^m and token level scale y_m using Gaussian functions.

Then we generate the sub-stroke from x^m and y^m using the cubic spline. The starting point of a new sub-stroke is the last point of the previous one. This way we get a stroke and place its beginning at the point given by L^m for this stroke. We repeat this process for all the strokes and ultimately get a character.

Then blurring noise σ_b^m is added, Blurring is accomplished through a convolution with a Gaussian filter. The amount of noise is itself a random variable, sampled from a uniform distribution. Lastly the second noise is added, the second noise process stochastically flips pixels with probability ϵ_b^m.

1.3.3 Learning to learn motor programs

Data will be randomly split into a 30 alphabet "background" set and a 20 alphabet "evaluation" set constrained such that the background set included the six most common alphabets as determined by Google hits, including Latin, Greek, Japanese, Korean, Hebrew, and Tagalog.

For learning the GMM model for clustering the primitives, we use EM algorithm for learning it's parameters.

For learning other parameters which are mean or covariance estimates of their distributions, we use Maximum Likelihood estimates of mean and variance to estimate them from their empirical frequencies (although we will also be using MAP estimates at some place, but we will assume flat prior there).

We will be using PyMC library so that we don't have to reinvent the wheel for implementing most of the algorithms for learning.

1.4 Progress

The original paper on the 'Omniglot Challenge' and several other relevant materials related to Bayesian Program Learning has been covered. Different parts of the classification task on the 'Omniglot' dataset like generating Character types, Character tokens and Learning-to-Learn Motor programs is being implented in python.

This section has allows to learn several skills related to probabilistic programming and Bayesian Program Learning.The future targets include completing the implementation of the Classification task and then improve upon it and also try and implement other tasks in python.

2 Own-Model

2.1 Objective

The objective of the Own-Model subsection was to create custom one-shot models as well as our own generative model. This is different from BPL in the sense that it was a completely probabilistic model of prediction as well as generation, whereas ours is based on a neural network function, which is based on learning features from the image data right from scratch.

We had also aimed to build a custom model on the lines of Bayesian Program Learning, mainly the generative model, but we haven't yet decided on how we would go ahead with that task of ours.

2.2 Literature Survey

One-shot learning represents a still-open challenge in computer vision to learn from one or a very few number of training examples, contrast to the traditional machine-learning models which uses thousands or tens of thousands of training examples in order to learn.

2.3 Progress

We started with the one-shot classification problem by deciding to train a simple convolutional neural network on the background set in the Omniglot dataset. The images were downsized to a resolution of 28x28. They were then fed into a network with the following architecture-

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1          [-1, 32, 26, 26]             320
       BatchNorm2d-2          [-1, 32, 26, 26]              64
              ReLU-3          [-1, 32, 26, 26]               0
            Conv2d-4          [-1, 64, 24, 24]          18,496
       BatchNorm2d-5          [-1, 64, 24, 24]             128
              ReLU-6          [-1, 64, 24, 24]               0
            Linear-7                 [-1, 80]       2,949,200
              ReLU-8                 [-1, 80]               0
            Linear-9                [-1, 964]          78,084
================================================================
Total params: 3,046,292
Trainable params: 3,046,292
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.00
Forward/backward pass size (MB): 1.35
Params size (MB): 11.62
Estimated Total Size (MB): 12.97
----------------------------------------------------------------
```

Such a model had been previously trained on the MNIST dataset to achieve a 99 percent accuracy. We did not have any idea of how it would perform.

We used the ADAM optimizer, initially with a learning rate of 0.0001, to find out the model was not learning anything. But as I decreased the learning rate to 0.000004, it started learning slowly, and a considerable speed was observed. After training for approximately 200 epochs, he model gave us a 93 percent test accuracy.

2.4 Future

We will now move to be replace the last layer of the model, to an identity mapping, so that we can use the 80 unit linear layer, and the pre-trained model to find how it performs while using a character from another alphabet.

This will be accomplished using image pairs. The 80 unit layer output will be taken, and we will use the Cosine Similarity distance function to check the similarity between two images.

The similarity measure is the measure of how much alike two data objects are. The similarity is subjective and is highly dependent on the domain and application. There are many metrics to calculate a distance between 2 points (x1, y1) and (x2, y2) in xy-plane.

Some of them are

1.Euclidean Distance: It is calculated as the square root of the sum of differences between each point.

2.Chebyshev Distance: It is calculated as the maximum of the absolute difference between the elements of the vectors.

3.Cosine Similarity: It measures the cosine angle between the two vectors.

4.Manhattan Distance: It is calculated as the sum of absolute distances between two points.

5.Pearson correlation coefficient: It measures linear correlation between two variables X and Y. It has a value between +1 and 1, where 1 is total positive linear correlation, 0 is no linear correlation, and 1 is total negative linear correlation (that the value lies between -1 and 1 is a consequence of the Cauchy–Schwarz inequality).

Using these metric function inside vgg16 score function one by one to calculate the average error produced by them, cosine similarity function gives the least error.

We will now build a neural network model in which instead of using a direct metric function we will try to develop our own a non-linear function which could probably give lower error rates.

3 Comparison

3.1 Objective

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.

3.2 Literature survey

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).

3.2.1 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.

3.2.2 GAN -

Generative Adversarial Networks(Goodfellow et. al) are dubbed as one of the coolest models in Machine Learning. Two neural networks contest with each other in a game. Given a training set, this technique learns to generate new data with the same statistics as the training set. Many SOTA GAN models like infoGAN, styleGAN, cycleGAN, have had a great influence in Machine Learning

3.3 Implementation

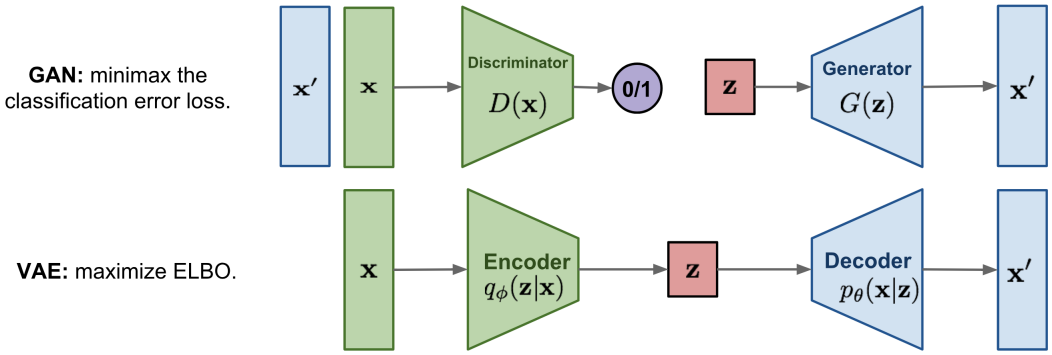
The two models(VAE and GAN) were implemented as:

3.3.1 VAE -

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.

3.3.2 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



3.4 Progress

Initially, models were built using VAE and GAN initially on the MNIST dataset. A basic model was built using VAE that tries to recreate the inputs. Attempts are being made to use different error functions, batch sizes and encoder-decoder models to improve the accuracy of the model and also try to feed data to the decoder to generate completely new objects based on what we have trained our network with.

The GAN model was initially implemented using numpy. This model performed quite well on the MNIST dataset with only 1 labelled image. But didn't work well with the full 10 labelled MNIST. This led to implementation of a Deep Convolutional GAN(DCGAN). This model worked well with Full MNIST dataset. But not quite well with Omniglot dataset due to smaller latent space. Next tasks include implementing GANs that work with very small dataset.

The section allows us to learn handling of datasets in python, using tensorflow and google colab for deep learning tasks.

4 Memory Augmented Neural Networks

4.1 Objective

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

4.2 Literature survey

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.

4.3 Implementation

We divided ourselves to work on two different approaches to solve the same low-shot problem.

4.3.1 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

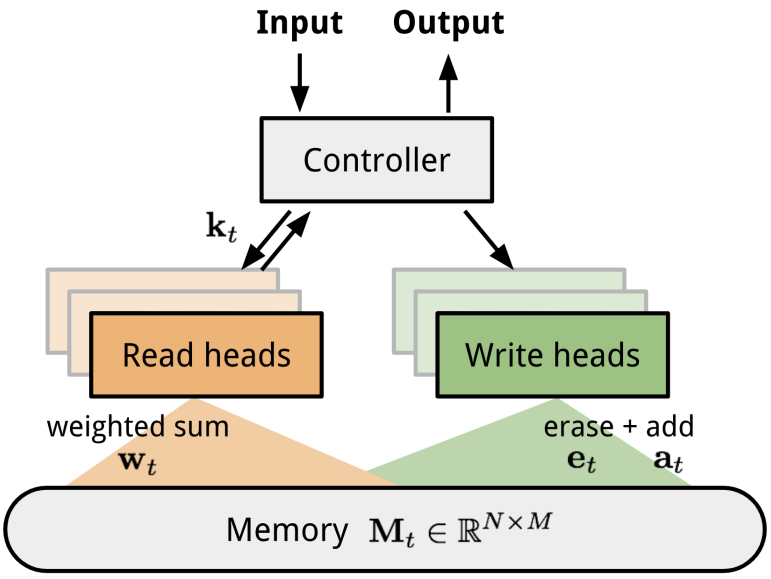
Input: Meta-training set $\mathscr{D}_{meta-train}$, Learner M with parameters θ, Meta-Learner R with parameters Θ.

$\Theta_0 \leftarrow$ random initialization

for $d = 1, n$ **do**
  $D_{train}, D_{test} \leftarrow$ random dataset from $\mathscr{D}_{meta-train}$
  $\theta_0 \leftarrow c_0$ ▷ Intialize learner parameters
  for $t = 1, T$ **do**
    $\mathbf{X}_t, \mathbf{Y}_t \leftarrow$ random batch from D_{train}
    $\mathcal{L}_t \leftarrow \mathcal{L}(M(\mathbf{X}_t; \theta_{t-1}), \mathbf{Y}_t)$ ▷ Get loss of learner on train batch
    $c_t \leftarrow R((\nabla_{\theta_{t-1}} \mathcal{L}_t, \mathcal{L}_t); \Theta_{d-1})$ ▷ Get output of meta-learner using Equation 2
    $\theta_t \leftarrow c_t$ ▷ Update learner parameters
  end for
  $\mathbf{X}, \mathbf{Y} \leftarrow D_{test}$
  $\mathcal{L}_{test} \leftarrow \mathcal{L}(M(\mathbf{X}; \theta_T), \mathbf{Y})$ ▷ Get loss of learner on test batch
  Update Θ_d using $\nabla_{\Theta_{d-1}} \mathcal{L}_{test}$ ▷ Update meta-learner parameters
end for

4.3.2 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



4.4 Progress

LSTM- Currently we are working on using LSTM as an optimizer network for our NN model such that it has a faster Gradient Descent. This optimization will then be used for few-shot learning tasks with the omniglot dataset. Future works include implementing MAML and other optimization based algorithms.
MANN- We are using an addressing scheme called Least recently used access(LRUA) for MANN, so here we are perfoming content based addressing for reading and write to the least recently used location, this design in turn perform better at one-shot learning.

References

[1] George Kour and Raid Saabne. Real-time segmentation of on-line handwritten arabic script. In *Frontiers in Handwriting Recognition (ICFHR), 2014 14th International Conference on*, pages 417–422. IEEE, 2014.

[2] George Kour and Raid Saabne. Fast classification of handwritten on-line arabic characters. In *Soft Computing and Pattern Recognition (SoCPaR), 2014 6th International Conference of*, pages 312–318. IEEE, 2014.

[3] Guy Hadash, Einat Kermany, Boaz Carmeli, Ofer Lavi, George Kour, and Alon Jacovi. Estimate and replace: A novel approach to integrating deep neural networks with existing applications. *arXiv preprint arXiv:1804.09028*, 2018.

[4] Brenden M. Lake, Ruslan Salakhutdinov, Joshua B. Tenenbaum The Omniglot challenge: a 3-year progress report *arXiv:1902.03477 [cs.AI]*, 2019

[5] Brenden M. Lake, Ruslan Salakhutdinov, Joshua B. Tenenbaum Supplementary Material for Human-level concept learning through probabilistic program induction *DOI: 10.1126/science.aab3050*, 2015