Project Report on

Optimization of Deep Learning in Big Data Scenarios

Submitted as partial fulfillment for the award of

Post Graduate Diploma

in

BIG DATA ANALYTICS

Session Feb '19 by

Aditya Ghate (PRN 190240125017)

Manas Shukla (PRN 190240125029)

Mukul Sharma (PRN 190240125031)

Under the guidance of

Dr. Shridhar Dattatray Page

(PG-DBDA) from C-DAC ACTS (Pune)

Centre for Development of Advance Computing (C-DAC)
ACTS, Pune

Students’ Declaration

We hereby declare that the work being presented in this report entitled “Optimization of Deep Learning in Big Data Scenarios” is an authentic record of our own work carried out under the supervision of Dr. Shridhar Dattatray Page

The matter embodied in this report has not been submitted by us for the award of any other degree / diploma.

Dated: ____________

____

Aditya Ghate (PG-DBDA, ACTS Pune)

____

Manas Shukla (PG-DBDA, ACTS Pune)

____

Mukul Sharma (PG-DBDA, ACTS Pune)

This is to certify that the above statement made by the candidate(s) is correct to the best of my knowledge.

____________________	____________________
Signature of Guide	**Signature of Co-Coordinator**
Name:	**Name:**
Date:	**Date:**

ACKNOWLEDGEMENT

I Aditya Ghate (PRN No. 190240125017), (Manas Shukla PRN No. 190240125029), and Mukul Sharma (PRN No. 190240125031) have taken efforts for this project. However, it would not have been possible without the kind support and help of many individuals and organizations. I would like to extend my sincere thanks to all of them. I am highly indebted to all the faculty of CDAC ACTS, Pune for their guidance and constant supervision as well as for providing necessary information regarding the project & also for their support in completing the project. I would like to express my special gratitude and thanks to our Coordinator, Namrata Ailawar and to our Project mentor Dr. Shridhar Dattatray Page for giving us support and time. Our thanks and appreciations also go to my colleague in developing the project and people who have willingly helped me out with their abilities.

Name: Aditya Ghate

Date:

Name: Manas Sharma

Date:

Name: Mukul Sharma

Date:

Table of Contents

List of Figures

Abstract

The technological advancement and research in the field of Machine Learning and Deep Computer Vision is moving up at fast pace. Neural Networks are powerful and flexible models that work well for many difficult learning tasks in image, speech and natural language understanding. Neural Networks have recently gained popularity and wide practical applications. However, to get good results with neural networks, it is critical to pick the right network topology, which has always been a difficult manual task. Therefore, despite their success, Neural Networks are hard to design. Google's recent project promises to help solve this task automatically with a meta-AI which will design the topology for neural network architecture. Google, however, did not offer documentation or examples of how to use this new wonderful technology. We liked the idea and came up with a practical implementation that other people can follow, using it as an example in our Project. This is similar in concept to AlphaGo, for instance. In this project, we use a recurrent network to generate the model descriptions of neural networks and train this RNN with reinforcement learning to maximize the expected accuracy of the generated architectures on a validation set. Our method, starting from scratch, can design a novel network architecture that rivals the best human-invented architecture in terms of test set accuracy. We have managed to achieve a validation accuracy of 95.7% on the MNIT standard dataset by training it based on the architecture suggested by the Controller Network.

1. Introduction

The last few years have seen much success of deep neural networks in many challenges applications, be it speech recognition (Hinton, et al., 2012), image recognition (LeCun, et al.) or machine translation (Luong, et al., 2014). These approaches brought with them a paradigm shift from feature designing to designing the overall architecture of the neural network. Although with recent developments the task has become relatively easier to comprehend, designing neural nets for tasks like image or speech recognition require significant architectural expertise.

Neural Architecture Search(NAS), is one such methodology to automate this process of designing neural networks by finding the optimal set of tuning parameters. NAS owes its growing research interest to the increasing prominence of deep learning models as of late. As of now several methods to design neural networks have emerged. Over the past couple of years, the community has seen different search methods proposed including the following:

Reinforcement Learning:

- Neural Architecture Search with Reinforcement Learning (Zoph, et al., 2016)
- NASNet (Zoph, et al., 2017)
- ENAS (Pham, et al., 2018)

Evolutionary Learning:

- Hierarchical Evo (Liu, et al., 2017)
- AmoebaNet (Real, et al., 2018)

Sequential Model based Optimization (SMBO):

- PNAS (Liu, et al., 2017)

Bayesian Optimization:

- Auto-Keras (Jin, et al., 2018)
- NASBOT (Kandasamy, et al., 2018)

Gradient based Optimization:

- SNAS (Xie, et al., 2018)
- DARTS (Liu, et al., 2018)

In this proposed study we have worked on an Efficient Neural Architecture Search(ENAS) which employs Reinforcement Learning on a Recurrent Neural Network(RNN) to build a set of Convolutional Neural Networks(CNNs).

We have based our approach on Reinforcement Learning, wherein the parent Neural Network reviews the potency of the child Neural Network and makes adjustments to its architecture, with the aim of achieving the maximum cumulative reward (the model accuracy in this case). Various parameters such as the number of layers, weights, regularization and activation methods, etc. can be tweaked in order to improve efficiency.

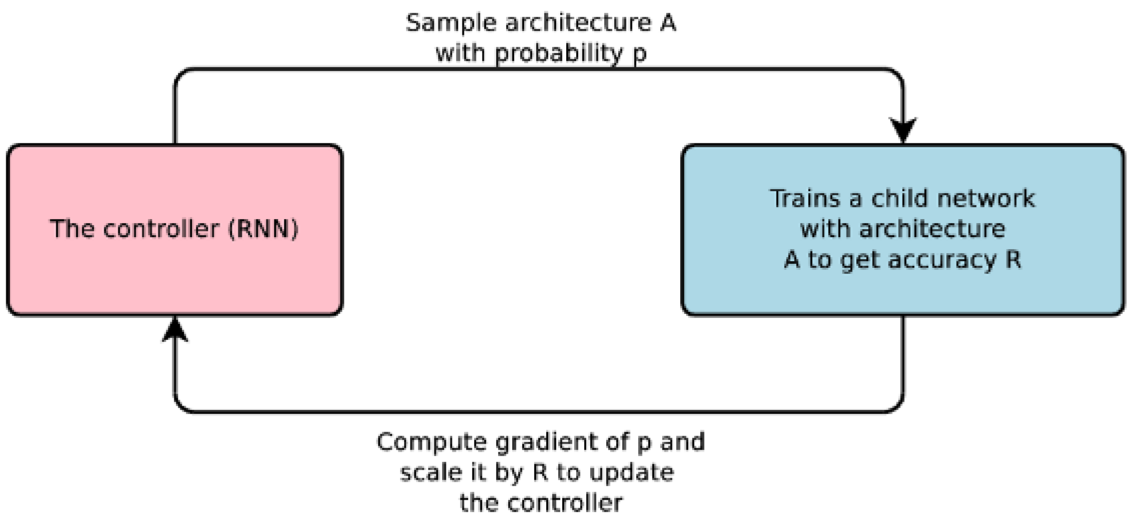


Figure 1: Image from Google Blog

The advantage of automating this task is to remove the ML expertise along with some level of guesswork required to find the optimal model parameters. Not to mention this would significantly reduce the time required to train the model, as this is one of the most time intensive tasks.

Neural Architecture Search (NAS) with Reinforcement Learning is a method for finding good neural networks architecture. In this proposed study, we have applied Reinforcement Learning to an RNN so that it could choose the best CNN to correctly identify numbers from images of handwritten digits under the guise of maximizing accuracy as the reward.

2. Objective

The objective of our project is to define a Controller (LSTM Cell or RNN) that looks for a novel Child Model (desired CNN for image classification) in a search space (all possible different architectures or child models that can possibly be generated) through a search strategy (a method to generate architecture of child models) and pick up the best child model that maximizes the reward or validation accuracy in this case.

The Controller network controls or directs the building of the child model's architecture by 'generating a set of instructions' (or, more rigorously, making decisions or sampling decisions) using a certain search strategy. These decisions can range from deciding the type of the layer (Convolutional, Max Pooling, Dense, etc.), the number of layers, the neurons required at each level, the type of activation function to use at each layer and so on.

Based on the above mentioned parameters, we try to define a state which will be fed into the Controller Network. This state is iterated upon for every episode, with the aim to find out the child model with maximum validation accuracy. Since we rely on an RNN for the controller network, every new state is calculated taking into consideration the previous states as well. Of all the models generated, the one with the highest validation accuracy is termed as the best fit for the image classification task.

3. Literature Review

Hyper parameter optimization is an important research topic in machine learning, and is widely used in practice (Bergstra, et al.) Despite their success, these methods are still limited in that they only search models from a fixed-length space. In other words, it is difficult to ask them to generate a variable-length configuration that specifies the structure and connectivity of a network. In practice, these methods often work better if they are supplied with a good initial model. (Snoek, et al., 2015) There are Bayesian optimization methods that allow to search non fixed length architectures (Mendoza, et al., 2016), but they are less general and less flexible than the method proposed in this project.

Modern neuro-evolution algorithms, e.g., (Wierstra, et al.); (Floreano, et al.); (Rhoades, et al.), on the other hand, are much more flexible for composing novel models, yet they are usually less practical at a large scale. Their limitations lie in the fact that they are search-based methods, thus they are slow or require many heuristics to work well.

Neural Architecture Search has some parallels to program synthesis and inductive programming, the idea of searching a program from examples (Summers, et al.). In machine learning, probabilistic program induction has been used successfully in many settings, such as learning to solve simple Q&A (Y, et al.), sort a list of numbers (Reed, et al., 2015), and learning with very few examples (Lake, et al., 2015).

The controller in Neural Architecture Search is auto-regressive, which means it predicts hyper parameters one a time, conditioned on previous predictions. This idea is borrowed from the decoder in end-to-end sequence to sequence learning (Sutskever, et al., 2014). Unlike sequence to sequence learning, this method optimizes a non-differentiable metric, which is the accuracy of the child network. It is therefore similar to the work on BLEU optimization in Neural Machine Translation (Ranzato, et al., 2016). Unlike these approaches, this method learns directly from the reward signal without any supervised bootstrapping.

More closely related is the idea of using a neural network to learn the gradient descent updates for another network (Andrychowicz, et al., 2016) and the idea of using reinforcement learning to find update policies for another network (Li, et al., 2016).

4. Methodology

4.1 Data and Pre-Processing

To train our model we have used the MNIST database (Yann LeCun)of handwritten digits, which has a training set of 55,000 images and a test set of 10,000 examples. The images are in black and white with a pixel size of 28 x 28. Each image can be classified into either of the 10 classes, representing the whole numbers from 0 to 9.

4.2 Pseudo Algorithm

Controller directs the building of the child model's architecture using a certain search strategy. These decisions for the controller are things like what types of operations (convolutions, pooling etc.) to perform at a particular layer of the child model. Using these decisions, many child models are built within a search space.

Particular child models are then trained to convergence using stochastic gradient descent to minimize the expected loss function between the predicted class and ground truth class (for an image classification task). This is done for a specified number of epochs, then, a validation accuracy is obtained from this trained model.

Then, we update the controller's parameters using REINFORCE, a policy-based reinforcement learning algorithm, to maximize the expected reward function which is the validation accuracy. This parameter update hopes to improve the controller in generating better decisions that give higher validation accuracies.

This entire process is just one epoch, say a controller epoch. We then repeat this for a specified number of controller epochs.

Of all the child models generated, the one with the highest validation accuracy is the optimum Neural Network for the image classification task. This entire problem is essentially a reinforcement learning framework with the archetypal elements:

- **Agent**: The Controller
- **Action:** The decisions taken to build the child network
- **Reward:** Validation accuracy from the child network

The aim of this reinforcement learning task is to maximize the reward (validation accuracy) from the actions taken (decisions taken to build child model architecture) by the agent (controller).

A pseudo algorithm for the entire training is written below:

```
- CONTROLLER_EPOCHS = Some_Value
- CHILD_EPOCHS = Some_Other_Value
- Build controller network

- for i in CONTROLLER_EPOCHS:

      1. Generate a child model
      2. Train this child model for CHILD_EPOCHS
      3. Obtain val_acc
      4. Update controller parameters

- Get child model with the highest val_acc
- Train this child model for CHILD_EPOCHS
```

4.3 The Model

The network we are built here consists of a controller and the actual neural network that we are trying to optimize. The Controller is an RNN tensorflow with NAS cells and special reinforcement learning methods for training and getting rewards.

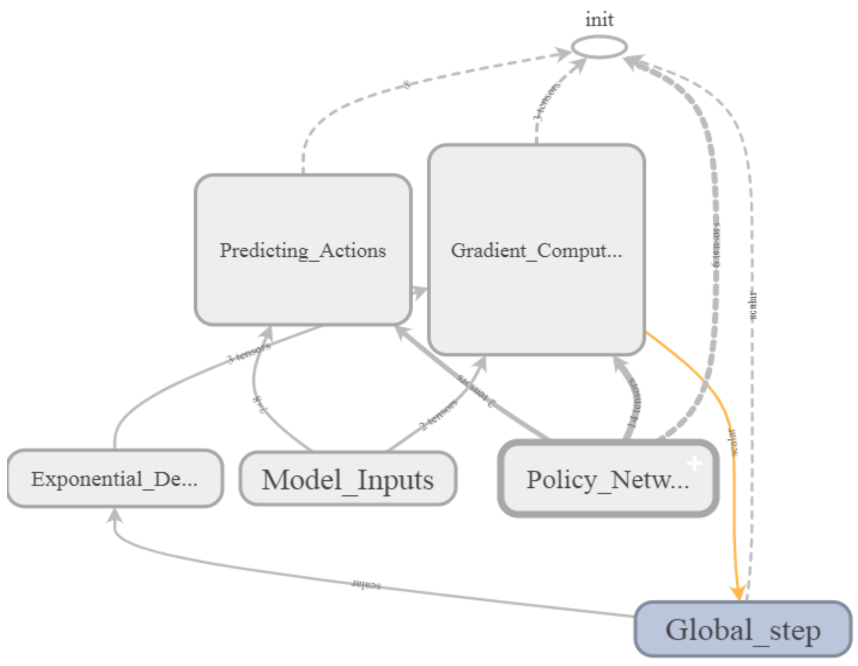


Figure 2: Model Overview(Tensorboard Graph)

We will define “rewards” as maximizing the accuracy of the desired neural network and train the Controller to improve this outcome. The controller should generate Actions to modify the architecture of CNN. Specifically, Actions can modify filters: the dimensionality of the output space, kernel_size (integer, specifying the length of the 1D convolution window), pool_size (integer, representing the size of the pooling window) and dropout_rate per layer.

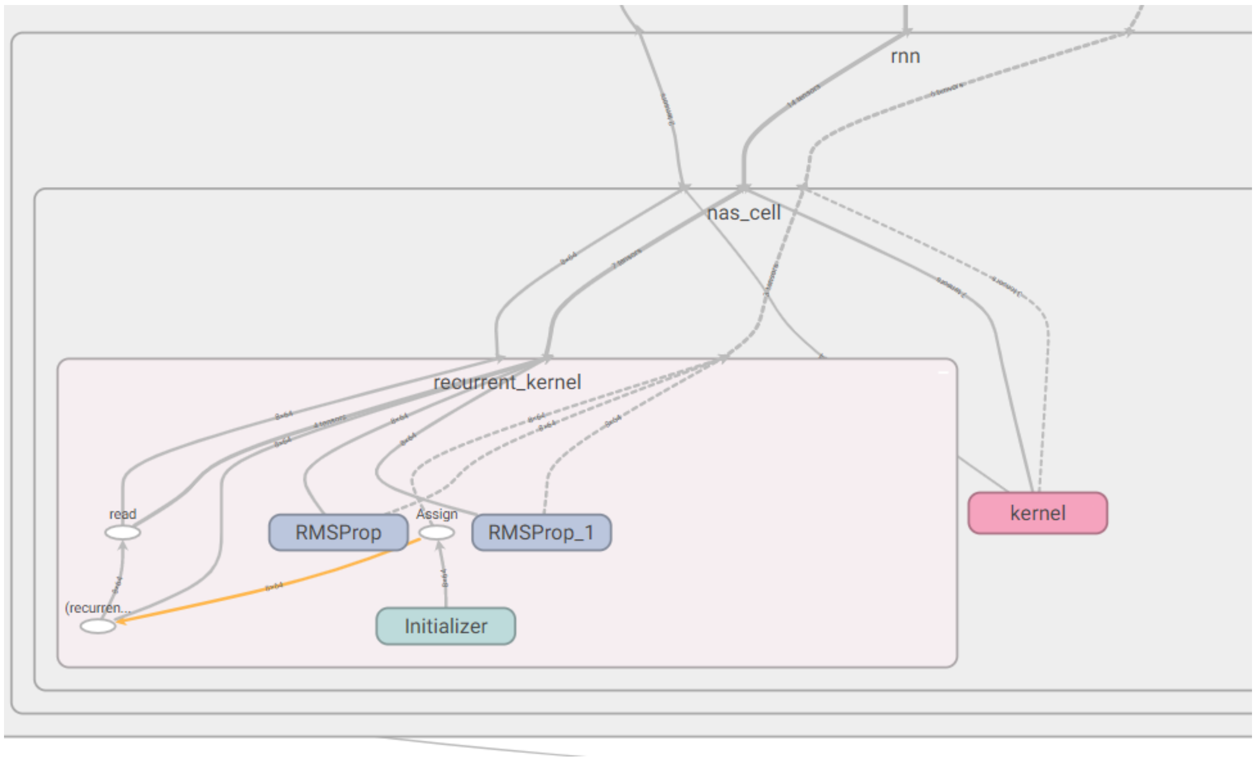


Figure 3: Expanded view of a NAS Cell(Tensorboard)

A policy network function maps the states along with the actions to be taken and returns the next action to the controller, which in this case is the architectural details for the CNN. This entire process is termed as one episode. The training is repeated for a substantial number of episodes, with the rewards being accumulated as either +0.01 or -0.01 based on the comparison between relative validation accuracies.

4.4 Details of the Architecture Search Space

All convolutions employ Rectified Linear Units non-linearity. Weights were initialized by the Xavier initialization algorithm. All CNN models employ a 1D - Convolutional layer, a Max Pooling layer, a Dropout layer twice to get 6 layers overall before being flattened into the final layer. This gives us a list of 8 parameters which is fed as the current state of the network into the Policy Network.

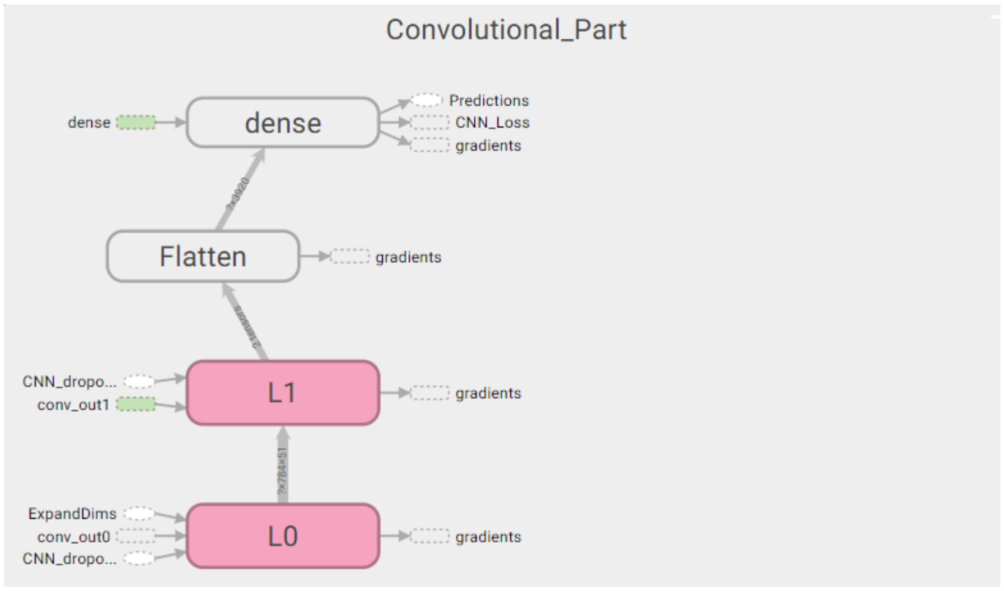


Figure 4: Architecture of CNN layers(Tensorboard)

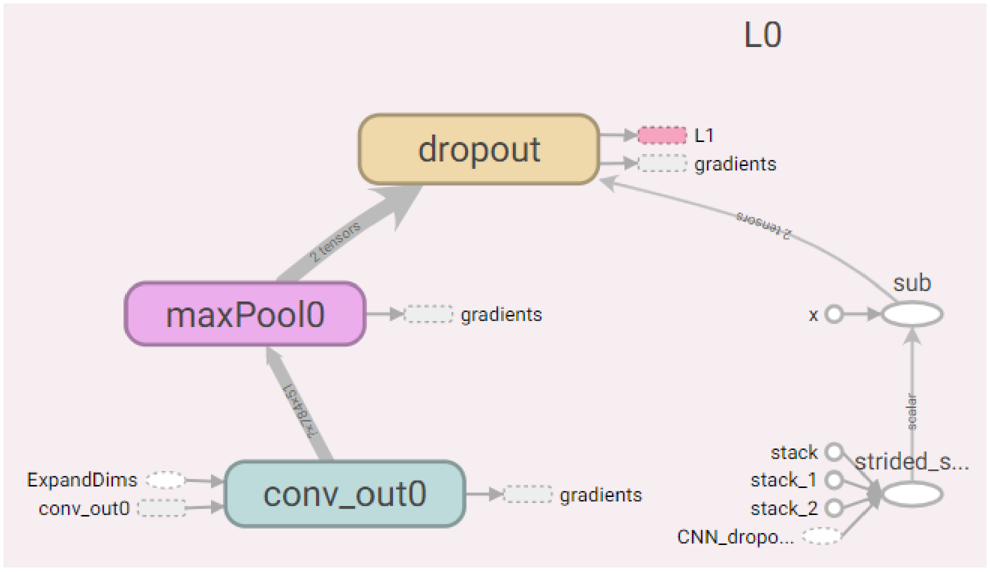
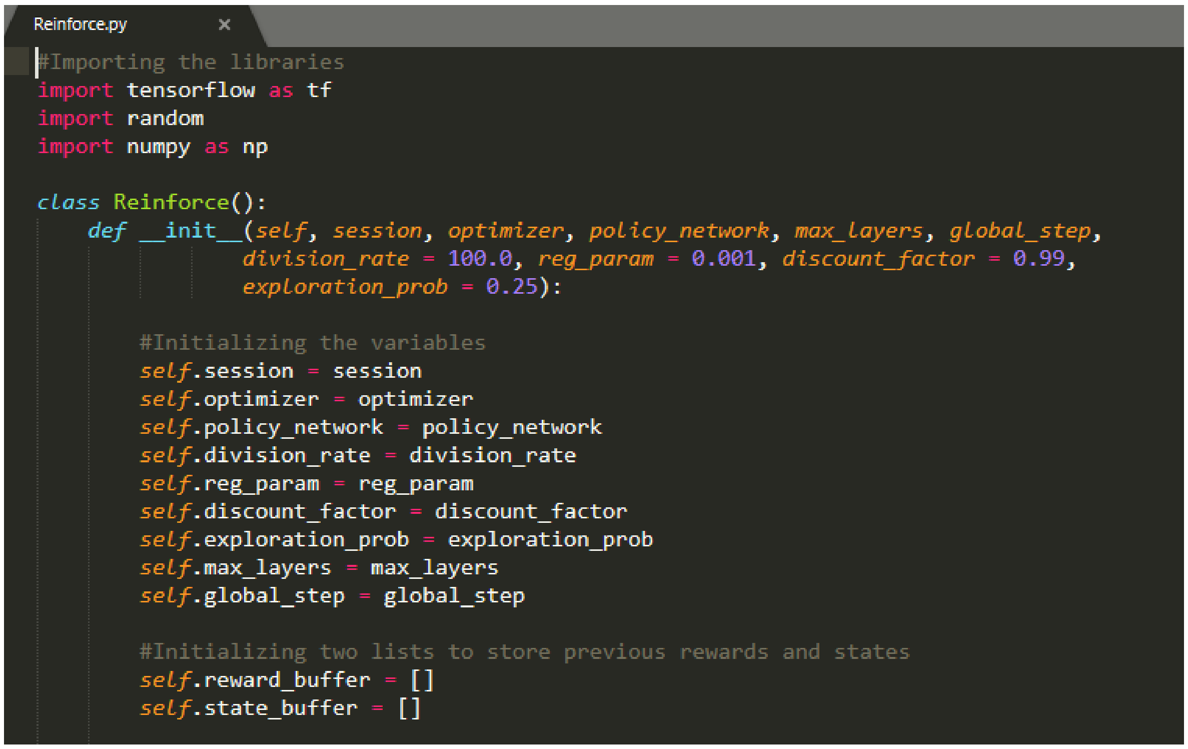


Figure 5: L0 expanded from previous figure

4.5 Implementation

For the Controller, a method for policy network based on NASCell was defined. This network takes, as inputs, the current state (in this task, state and action are the same things) and maximum number of searching layers and outputs new Action to update the desired neural network.

To allow hyper parameter tuning we put our code into a Reinforce class.



Reinforce.py

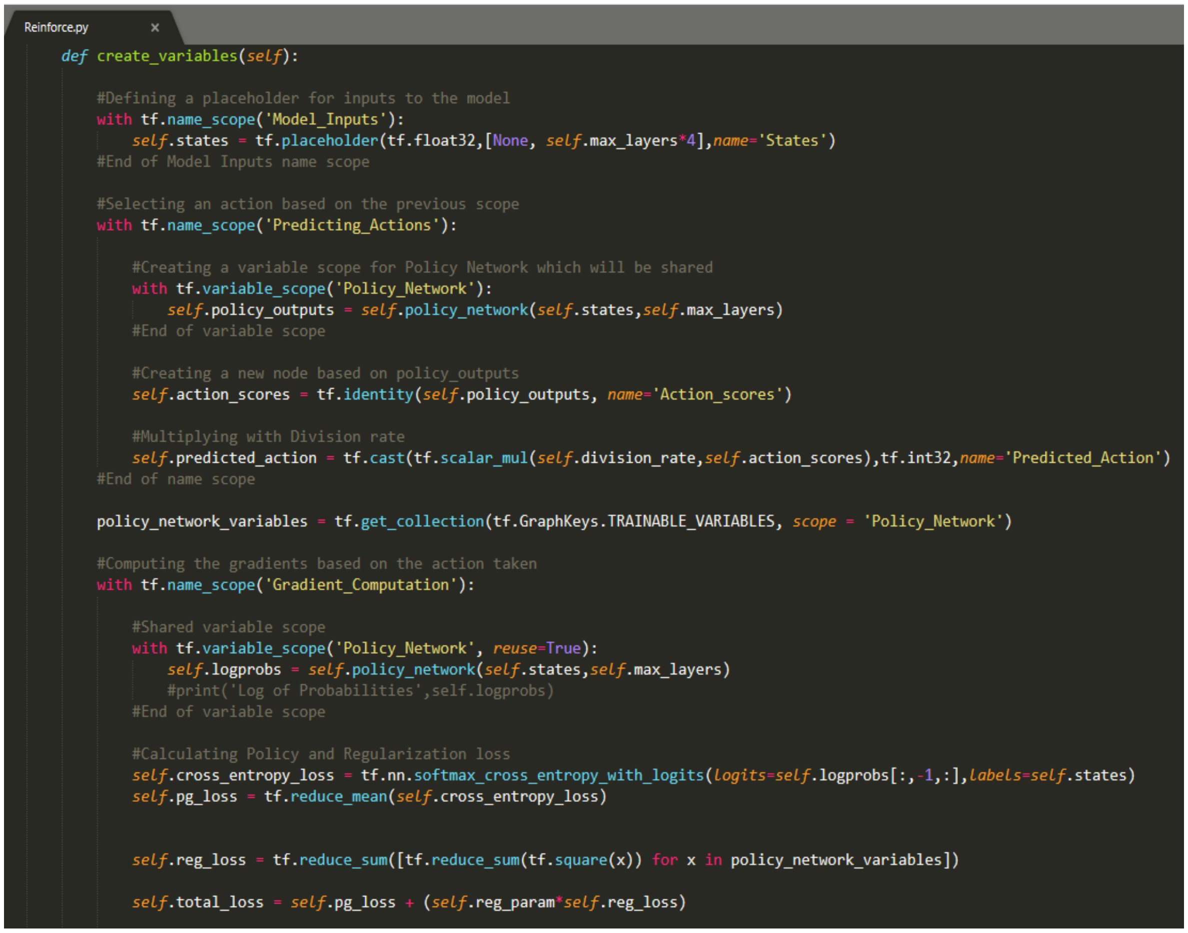
```
#Importing the libraries
import tensorflow as tf
import random
import numpy as np

class Reinforce():
    def __init__(self, session, optimizer, policy_network, max_layers, global_step,
                division_rate = 100.0, reg_param = 0.001, discount_factor = 0.99,
                exploration_prob = 0.25):

        #Initializing the variables
        self.session = session
        self.optimizer = optimizer
        self.policy_network = policy_network
        self.division_rate = division_rate
        self.reg_param = reg_param
        self.discount_factor = discount_factor
        self.exploration_prob = exploration_prob
        self.max_layers = max_layers
        self.global_step = global_step

        #Initializing two lists to store previous rewards and states
        self.reward_buffer = []
        self.state_buffer = []
```

Figure 6: Reinforce class

To instantiate the class, we then pass the following arguments:

- **session** and **optimizer**: TensorFlow session and RMSPROP optimizer, which will be initialized separately
- **policy_network**: Method to map states and actions
- **max_layers**: The maximum number of layers
- **division_rate**: Normal distribution values of each neuron from -1.0 to 1.0
- **reg_param**: Parameter for regularization
- **exploration**: The probability of generating random action

Since Tensorflow was used for this project, a graph oriented methodology had to be followed. In order to create the nodes and variables in the graph, a method called create_variables was defined.



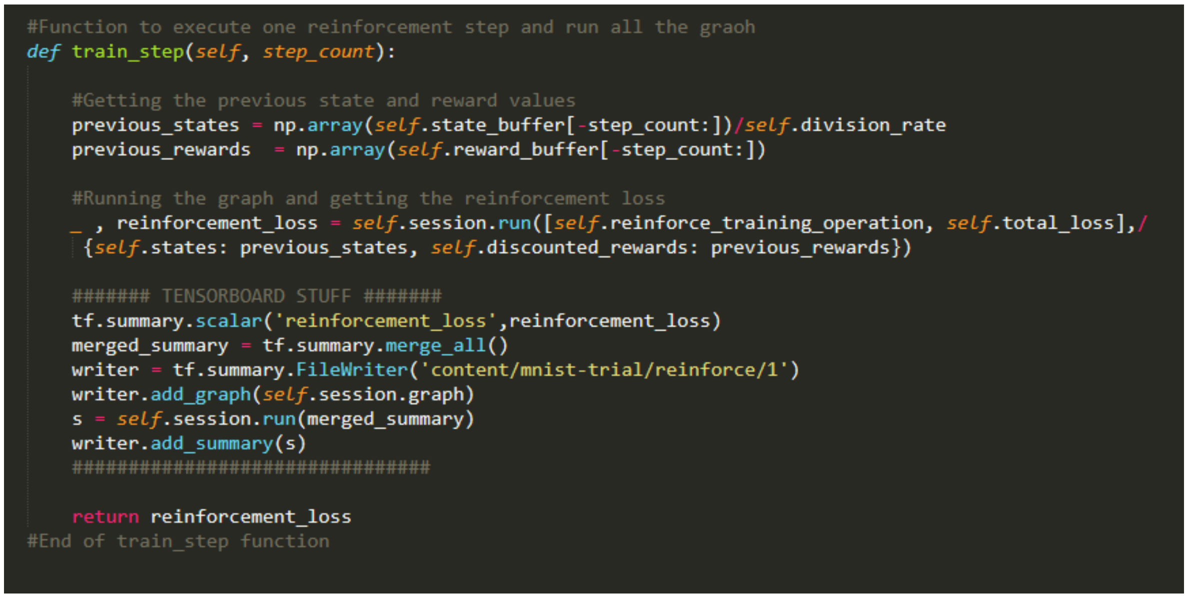
```
    def create_variables(self):

        #Defining a placeholder for inputs to the model
        with tf.name_scope('Model_Inputs'):
            self.states = tf.placeholder(tf.float32,[None, self.max_layers*4],name='States')
        #End of Model Inputs name scope

        #Selecting an action based on the previous scope
        with tf.name_scope('Predicting_Actions'):

            #Creating a variable scope for Policy Network which will be shared
            with tf.variable_scope('Policy_Network'):
                self.policy_outputs = self.policy_network(self.states,self.max_layers)
            #End of variable scope

            #Creating a new node based on policy_outputs
            self.action_scores = tf.identity(self.policy_outputs, name='Action_scores')

            #Multiplying with Division rate
            self.predicted_action = tf.cast(tf.scalar_mul(self.division_rate,self.action_scores),tf.int32,name='Predicted_Action')
        #End of name scope

        policy_network_variables = tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES, scope = 'Policy_Network')

        #Computing the gradients based on the action taken
        with tf.name_scope('Gradient_Computation'):

            #Shared variable scope
            with tf.variable_scope('Policy_Network', reuse=True):
                self.logprobs = self.policy_network(self.states,self.max_layers)
                #print('Log of Probabilities',self.logprobs)
            #End of variable scope

            #Calculating Policy and Regularization loss
            self.cross_entropy_loss = tf.nn.softmax_cross_entropy_with_logits(logits=self.logprobs[:,-1,:],labels=self.states)
            self.pg_loss = tf.reduce_mean(self.cross_entropy_loss)


            self.reg_loss = tf.reduce_sum([tf.reduce_sum(tf.square(x)) for x in policy_network_variables])

            self.total_loss = self.pg_loss + (self.reg_param*self.reg_loss)
```

Figure 7: Method to instantiate nodes of the TensorFlow graph

After computing the initial gradients, the gradient descent method is launched and gradient value is multiplied to the discounted reward.

Every Action depends on the previous state, but sometimes, for more effective training, we can generate random actions to avoid local minimums.

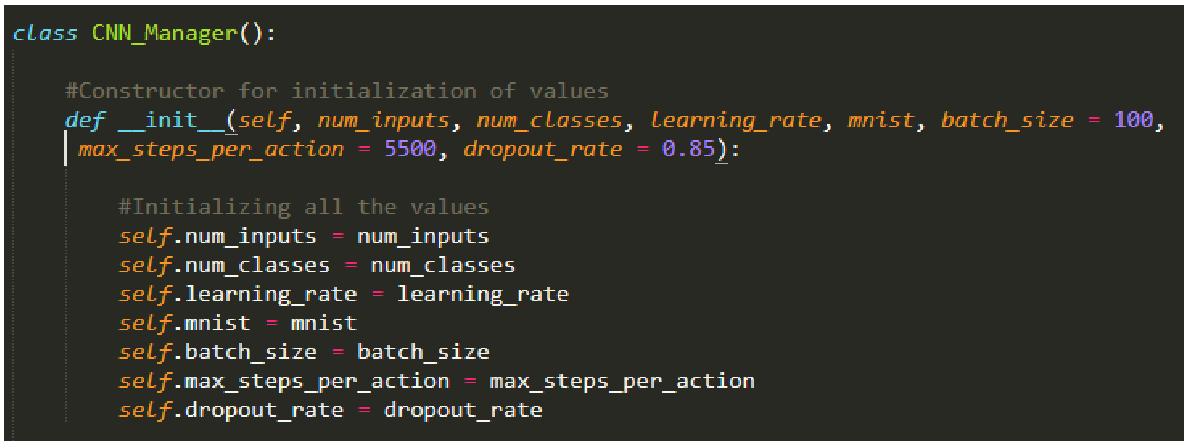
In each cycle, our network will generate an Action, get rewards and after that, take a training step. The implementation of the training step includes store_rollout and train_step methods below:



```
#Function to execute one reinforcement step and run all the graoh
def train_step(self, step_count):

    #Getting the previous state and reward values
    previous_states = np.array(self.state_buffer[-step_count:])/self.division_rate
    previous_rewards  = np.array(self.reward_buffer[-step_count:])

    #Running the graph and getting the reinforcement loss
    _ , reinforcement_loss = self.session.run([self.reinforce_training_operation, self.total_loss],/
     {self.states: previous_states, self.discounted_rewards: previous_rewards})

    ####### TENSORBOARD STUFF #######
    tf.summary.scalar('reinforcement_loss',reinforcement_loss)
    merged_summary = tf.summary.merge_all()
    writer = tf.summary.FileWriter('content/mnist-trial/reinforce/1')
    writer.add_graph(self.session.graph)
    s = self.session.run(merged_summary)
    writer.add_summary(s)
    ################################

    return reinforcement_loss
#End of train_step function
```

Figure 8: Training Step of Reinforcement class

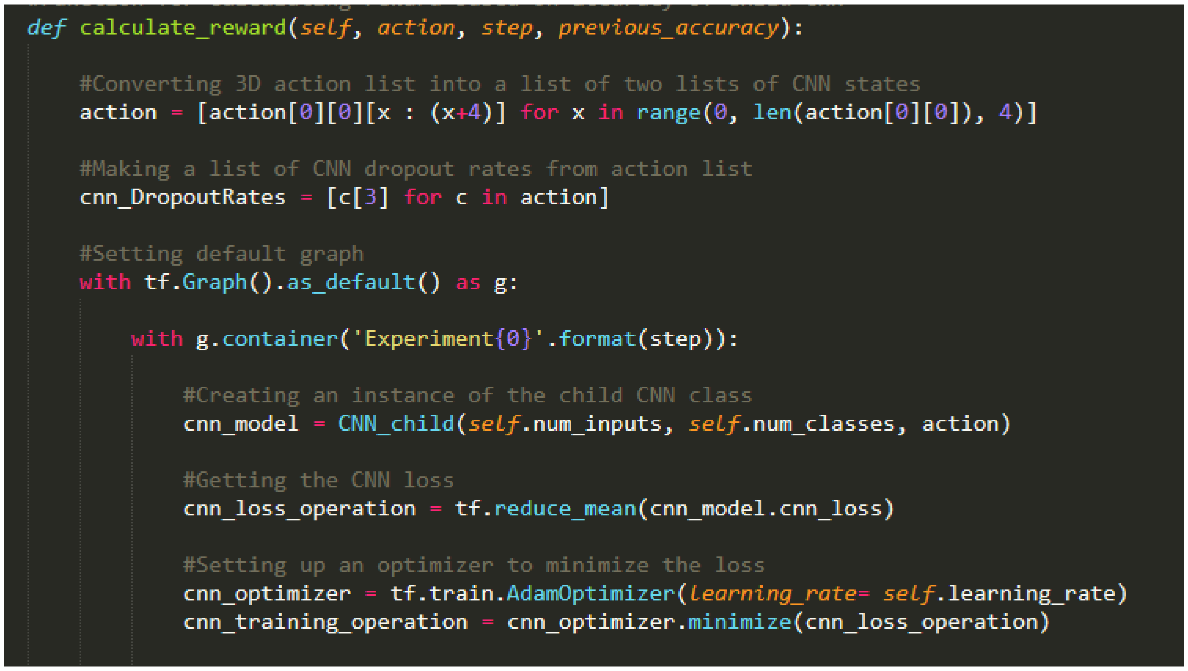
Defining rewards for each Action / State is accomplished by generating a new CNN network with new architecture per Action, training it and assessing its accuracy. Since this process generates a lot of CNN networks, a separate CNN Manager class was developed.



```
class CNN_Manager():

    #Constructor for initialization of values
    def __init__(self, num_inputs, num_classes, learning_rate, mnist, batch_size = 100,
     max_steps_per_action = 5500, dropout_rate = 0.85):

        #Initializing all the values
        self.num_inputs = num_inputs
        self.num_classes = num_classes
        self.learning_rate = learning_rate
        self.mnist = mnist
        self.batch_size = batch_size
        self.max_steps_per_action = max_steps_per_action
        self.dropout_rate = dropout_rate
```

Figure 9: CNN Manager to handle CNN Networks

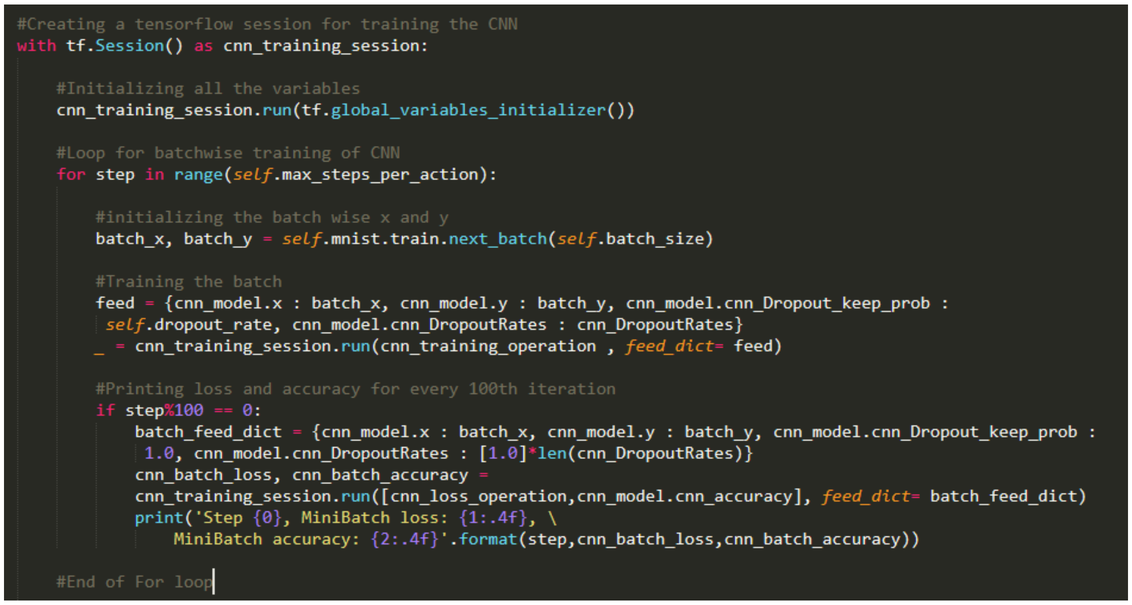
Further, a batch size of 100 with hyper parameters for every layer in “action” was train and cnn_drop_rate - list of dropout rates for every layer was extracted.



```
def calculate_reward(self, action, step, previous_accuracy):

    #Converting 3D action list into a list of two lists of CNN states
    action = [action[0][0][x : (x+4)] for x in range(0, len(action[0][0]), 4)]

    #Making a list of CNN dropout rates from action list
    cnn_DropoutRates = [c[3] for c in action]

    #Setting default graph
    with tf.Graph().as_default() as g:

        with g.container('Experiment{0}'.format(step)):

            #Creating an instance of the child CNN class
            cnn_model = CNN_child(self.num_inputs, self.num_classes, action)

            #Getting the CNN loss
            cnn_loss_operation = tf.reduce_mean(cnn_model.cnn_loss)

            #Setting up an optimizer to minimize the loss
            cnn_optimizer = tf.train.AdamOptimizer(learning_rate= self.learning_rate)
            cnn_training_operation = cnn_optimizer.minimize(cnn_loss_operation)
```

Figure 10: Function to calculate rewards based on accuracy comparison

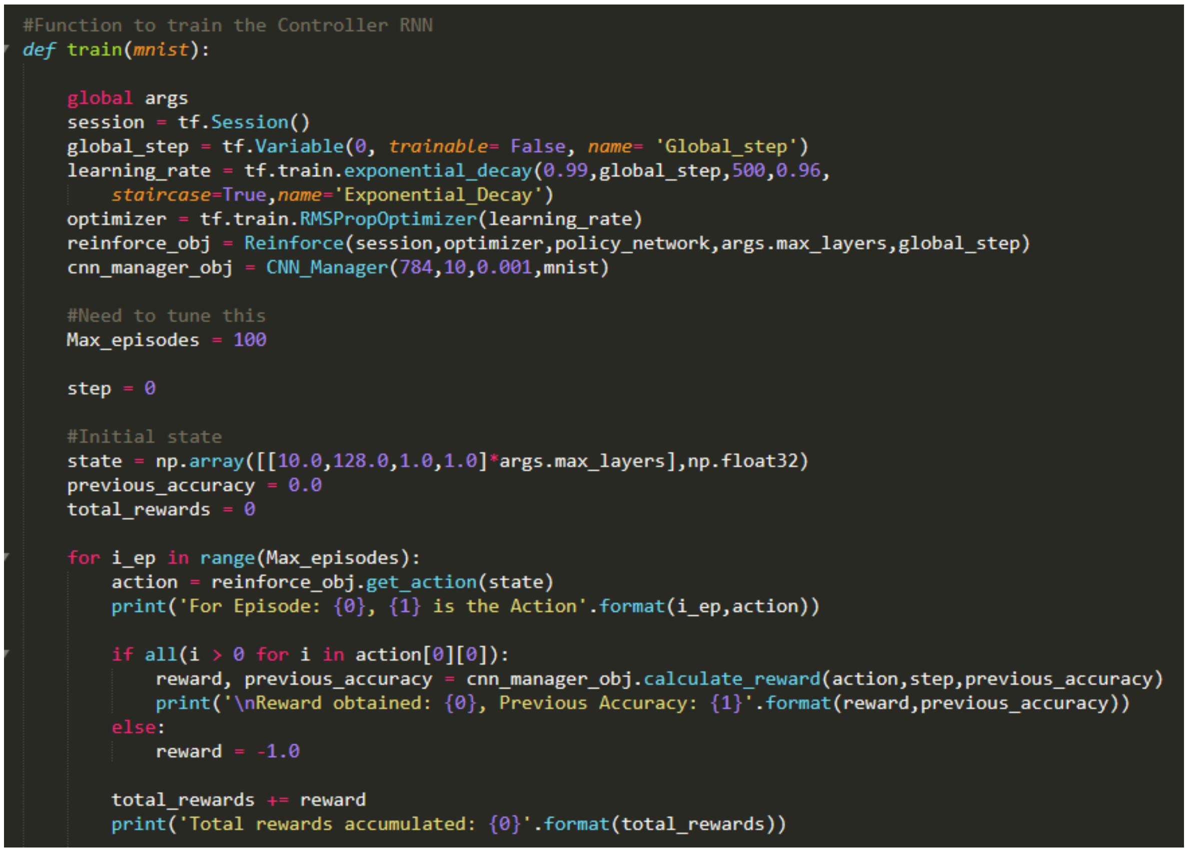
We defined a convolution neural model with CNN class. It can be any class that is able to generate the neural model by some action. We created a separate container to avoid confusion in TF graph. After creating a new CNN model, we can train it and get a reward.



```
#Creating a tensorflow session for training the CNN
with tf.Session() as cnn_training_session:

    #Initializing all the variables
    cnn_training_session.run(tf.global_variables_initializer())

    #Loop for batchwise training of CNN
    for step in range(self.max_steps_per_action):

        #initializing the batch wise x and y
        batch_x, batch_y = self.mnist.train.next_batch(self.batch_size)

        #Training the batch
        feed = {cnn_model.x : batch_x, cnn_model.y : batch_y, cnn_model.cnn_Dropout_keep_prob :
         self.dropout_rate, cnn_model.cnn_DropoutRates : cnn_DropoutRates}
        _ = cnn_training_session.run(cnn_training_operation , feed_dict= feed)

        #Printing loss and accuracy for every 100th iteration
        if step%100 == 0:
            batch_feed_dict = {cnn_model.x : batch_x, cnn_model.y : batch_y, cnn_model.cnn_Dropout_keep_prob :
             1.0, cnn_model.cnn_DropoutRates : [1.0]*len(cnn_DropoutRates)}
            cnn_batch_loss, cnn_batch_accuracy =
            cnn_training_session.run([cnn_loss_operation,cnn_model.cnn_accuracy], feed_dict= batch_feed_dict)
            print('Step {0}, MiniBatch loss: {1:.4f}, \
                MiniBatch accuracy: {2:.4f}'.format(step,cnn_batch_loss,cnn_batch_accuracy))

    #End of For loop
```

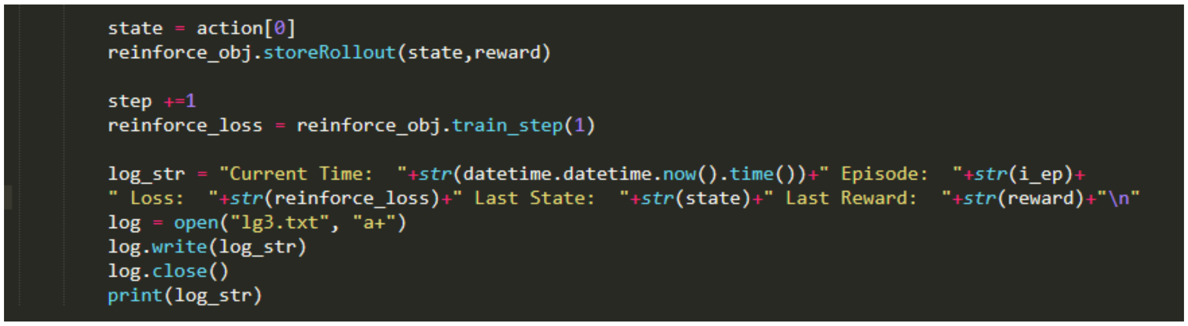
Figure 11: CNN layers

As defined, the reward improves accuracy on all test datasets; for MNIST it is 10000 examples. Now we proceed with finding the best architecture for MNIST. First, the architecture for the number of layers is optimized. For simplicity, the maximum number of layers was set to 2. Although, we can set this value to be higher, but every layer needs a lot of computing power.



```python
#Function to train the Controller RNN
def train(mnist):

    global args
    session = tf.Session()
    global_step = tf.Variable(0, trainable= False, name= 'Global_step')
    learning_rate = tf.train.exponential_decay(0.99,global_step,500,0.96,
        staircase=True,name='Exponential_Decay')
    optimizer = tf.train.RMSPropOptimizer(learning_rate)
    reinforce_obj = Reinforce(session,optimizer,policy_network,args.max_layers,global_step)
    cnn_manager_obj = CNN_Manager(784,10,0.001,mnist)

    #Need to tune this
    Max_episodes = 100

    step = 0

    #Initial state
    state = np.array([[10.0,128.0,1.0,1.0]*args.max_layers],np.float32)
    previous_accuracy = 0.0
    total_rewards = 0

    for i_ep in range(Max_episodes):
        action = reinforce_obj.get_action(state)
        print('For Episode: {0}, {1} is the Action'.format(i_ep,action))

        if all(i > 0 for i in action[0][0]):
            reward, previous_accuracy = cnn_manager_obj.calculate_reward(action,step,previous_accuracy)
            print('\nReward obtained: {0}, Previous Accuracy: {1}'.format(reward,previous_accuracy))
        else:
            reward = -1.0

        total_rewards += reward
        print('Total rewards accumulated: {0}'.format(total_rewards))
```

Figure 12: Main training logic



```python
        state = action[0]
        reinforce_obj.storeRollout(state,reward)

        step +=1
        reinforce_loss = reinforce_obj.train_step(1)

        log_str = "Current Time:  "+str(datetime.datetime.now().time())+" Episode:  "+str(i_ep)+
        " Loss:  "+str(reinforce_loss)+" Last State:  "+str(state)+" Last Reward:  "+str(reward)+"\n"
        log = open("lg3.txt", "a+")
        log.write(log_str)
        log.close()
        print(log_str)
```

Figure 13: Logging and Reinforcement model training

The initial state was chosen as [[10.0, 128.0, 1.0, 1.0] *args.max_layers] to expedite the training process. After each iteration the state was sent in as an input to the RNN to obtain the next action. In this context, action and state have been used synonymously. This is the reason the reinforcement training is being conducted in steps of 1.

After 100 cycles, we get the following architecture:

- Input layer: 784 nodes (MNIST images size)
- First convolution layer: 15 x 17
- First max-pooling layer: 5
- Second convolution layer: 34 x 9
- Second max-pooling layer: 5
- Output layer: 10 nodes (number of class for MNIST)

5. Results and Discussion

After training the model for 250 episodes, the loss determined by Markov's decision process was observed to be converging. In other words, the state output by the RNN almost sees a variation of ±1 data points based on the Gradient Descent method.

The loss function over time was visualized using Google Tensorboard and is as follows:

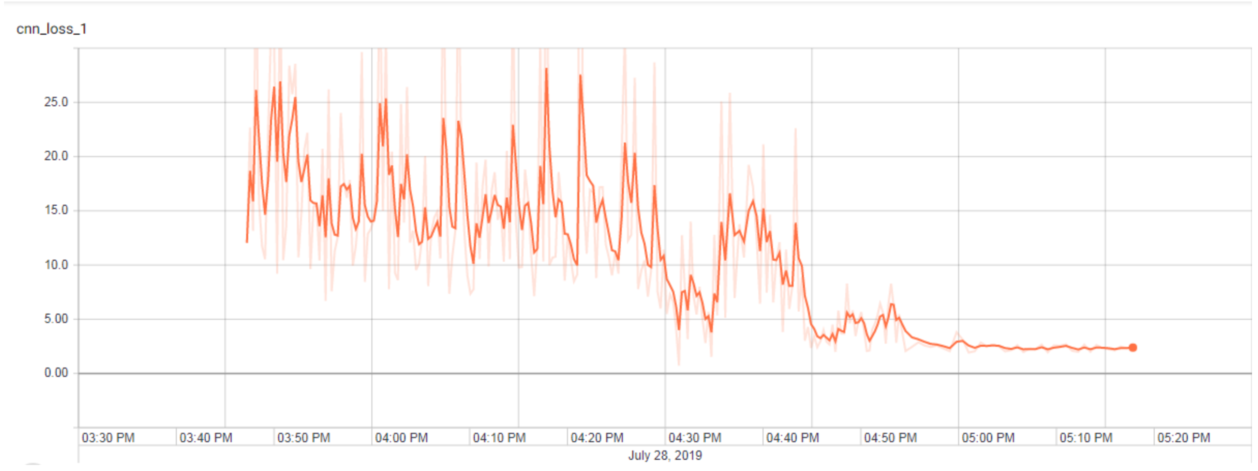


Figure 14: Convergence of the Loss function(Tensorboard)

The final Reinforcement loss value was observed to be around 2.458 and can be seen as asymptotically propagating. We were able to achieve a validation accuracy of 95.7% on 250 trial episodes. We believe further increment of the number of episodes in training can improve the accuracy but at the trade-off of heavy computation power. The CNN accuracy over time can be visualized as follows:

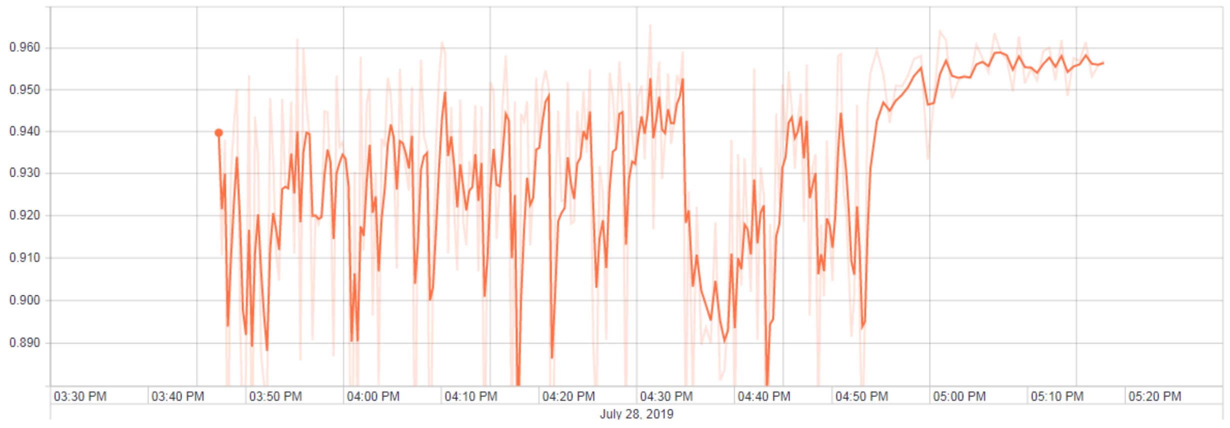


Figure 15: Validation accuracy over time(Tensorboard)

6. Conclusion and Future Scope

In this study, the aim was to implement Neural Architecture Search based on a Recurrent Neural Network as the controller. Using an RNN as the parent neural network gives the flexibility of giving a variable state over different number of episodes. Also, every previously calculated state is discounted by the reward obtained based on the accuracy of the child CNN. Tensorflow framework was used to create the model graph. NAS training requires heavy computing power and a Tesla T4 GPU on Google Colab was used for the same. Upon a 250 trial episodes, we were able to achieve a validation accuracy of 95.7% with the loss function converging at a value of 2.458. Upon further research, the NAS network can be used to completely eliminate the ML expertise and make the process of training a deep neural network model more intuitive.

7. Bibliography

Andrychowicz Marcin [et al.] Learning to learn by gradient descent by gradient descent [Online]. - 2016. - https://arxiv.org/abs/1606.04474.

Bergstra James and Bengio Yoshua Random Search for Hyper-Parameter Optimization [Online]. - http://jmlr.csail.mit.edu/papers/volume13/bergstra12a/bergstra12a.pdf.

Floreano Dario, Nolfi Stefano and Husbands Phil Evolutionary Robotics [Online]. - https://www.researchgate.net/publication/226668077_Evolutionary_Robotics.

Hinton Geoffrey [et al.] Deep Neural Networks for Acoustic Modeling [Online]. - April 27, 2012. - https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/38131.pdf.

Jin Haifeng, Song Qingquan and Hu Xia Auto-Keras: An Efficient Neural Architecture Search System [Online]. - June 27, 2018. - https://arxiv.org/abs/1806.10282.

Kandasamy Kirthevasan [et al.] Neural Architecture Search with Bayesian Optimisation and Optimal Transport [Online]. - 2018. - https://papers.nips.cc/paper/7472-neural-architecture-search-with-bayesian-optimisation-and-optimal-transport.pdf.

Lake Brenden M. [et al.] Human-level concept learning through probabilistic program induction [Online]. - 2015. - https://web.mit.edu/cocosci/Papers/Science-2015-Lake-1332-8.pdf.

LeCun Yann and Bengio Yoshua Convolutional Networks for Images, Speech, and Timw-Series [Online]. - https://www.researchgate.net/profile/Yann_Lecun/publication/2453996_Convolutional_Networks_for_Images_Speech_and_Time-Series/links/0deec519dfa2325502000000.pdf.

Li Ke and Malik Jitendra Learning to Optimize [Online]. - 2016. - https://arxiv.org/abs/1606.01885.

Liu Hanxiao [et al.] Hierarchical Representations for Efficient Architecture Search [Online]. - Nov 01, 2017. - https://arxiv.org/abs/1711.00436.

Liu Hanxiao, Simonyan Karen and Yang Yiming DARTS: Differentiable Architecture Search [Online]. - June 24, 2018. - https://arxiv.org/abs/1806.09055.

Liu Huiquan [et al.] A-to-I RNA editing is developmentally regulated and generally adaptive for sexual reproduction in Neurospora crassa [Online]. - Feb 16, 2017. - https://www.pnas.org/content/early/2017/08/23/1702591114.short.

Luong Minh-Thang [et al.] Addressing the Rare Word Problem in Neural Machine Translation [Online]. - October 30, 2014. - https://arxiv.org/abs/1410.8206.

Mendoza Hector [et al.] Towards Automatically-Tuned Neural Networks [Online]. - 2016. - https://ml.informatik.uni-freiburg.de/papers/16-AUTOML-AutoNet.pdf.

Pham Hieu [et al.] Efficient Neural Architecture Search via Parameter Sharing [Online]. - Feb 9, 2018. - https://arxiv.org/abs/1802.03268.

Ranzato Marc'Aurelio, Zaremba Wojciech and Auli Michael Sequence Level Training with Recurrent Neural Networks [Online]. - 2016. - https://www.researchgate.net/publication/319769905_Sequence_Level_Training_with_Recurrent_Neural_Networks.

Real Esteban [et al.] Regularized Evolution for Image Classifier Architecture Search [Online]. - Feb 5, 2018. - https://arxiv.org/abs/1802.01548.

Reed Scott and Freitas Nando de Neural Programmer-Interpreters [Online]. - 2015. - https://arxiv.org/abs/1511.06279.

Rhoades Galena K., Stanley Scott M. and Markman Howard J. Working with Cohabitation in Relationship Education and Therapy [Online]. - https://www.ncbi.nlm.nih.gov/pmc/articles/PMC2897720/.

Snoek Jasper [et al.] Scalable Bayesian Optimization Using Deep Neural Networks [Online]. - Feb 19, 2015. - https://arxiv.org/abs/1502.05700.

Summers David A., Ashworth Clark D. and Feldman-Summer Shirley Judgment Processes and Interpersonal Conflict Related to Societal Problem Solutions [Online]. - https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1559-1816.1977.tb01337.x.

Sutskever Ilya, Vinyals Oriol and Le Quoc V. Sequence to Sequence Learning with Neural Networks [Online]. - 2014. - https://arxiv.org/abs/1409.3215.

Wierstra Daan, Glasmachers Tobias and Sun Yi Natural Evolution Strategies [Online]. - http://www.jmlr.org/papers/volume15/wierstra14a/wierstra14a.pdf.

Xie Sirui [et al.] SNAS: stochastic neural architecture search [Online]. - Sept 28, 2018. - https://openreview.net/forum?id=rylqooRqK7.

Y Liang [et al.] Role of Candida albicans Aft2p transcription factor in ferric reductase activity, morphogenesis and virulence. [Online]. - https://www.yeastgenome.org/reference/S000134574.

Yann LeCun Corinna Cortes, Christopher J.C. Burges THE MNIST DATABASE [Online]. - http://yann.lecun.com/exdb/mnist/.

Zoph Barret [et al.] Learning Transferable Architectures for Scalable Image Recognition [Online]. - July 21, 2017. - arXiv:1707.07012.

Zoph Barret and Le Quoc V. Neural Architecture Search with Reinforcement Learning [Online]. - 2016. - https://arxiv.org/abs/1611.01578.