Learning to Design Convolutional Neural Networks with Reinforcement Learning

Aashima Arora
Department of Computer Science
Columbia University
aa3917@columbia.edu

Jason Krone
Department of Computer Science
Columbia University
jpk2151@columbia.edu

Abstract

Developing novel and effective convolutional neural network (CNN) architectures currently requires significant expertise and considerable trial and error. In this paper, we explore the use of reinforcement learning to automate the design of CNN architectures using validation accuracy as our reward function. Futhermore, we show that it is relatively simple to modify the reward function to account for computational requirements, such as the maximum number of parameters that can be used in the network. We intentionally constrain our architecture search space to decrease training time, determine the values of the reward function, and allow for a complete analysis of sample complexity and performance.

1 Introduction

State-of-the-art convolutional architectures have become increasingly complex over the recent years as researchers search for better performing models. This trend can be seen in the winning architectures of the ImageNet competition. The first deep neural network to win the competition, AlexNet, was a relatively simple 8-layer architecture. Soon thereafter, the networks became deeper with VGG and more complicated as skip connections were added and multiple convolutional operations were applied at each layer in the ResNet and Inception architectures respectively. In addition, the performance gains and designs of these networks continue to become more incremental and less intuitive over time. For these reasons, it is likely that future advances in neural network architectures will be driven by more complex building blocks and require a high volume of experiments to discover. Therefore, it makes sense to automate architecture design by extending the paradigm of end-to-end learning to include the optimization of network architectures.

There are a number of candidate methods for automating architecture discovery. Bayesian optimization, evolution strategies, and reinforcement learning are all capable of optimizing black box functions, such as validation accuracy. In this work, we choose the Reinforcement Learning framework because it has demonstrated success in designing state-of-the-art architectures for the ImageNet dataset. Please see our related works section for a more detailed discussion of this result and alternative methods.

To provide context we give a brief review of the reinforcement learning problem and describe how architecture search fits into this formulation. Reinforcement learning is the process of training an agent to maximize reward in an environment. More technically, the aim is to learn the optimal policy for selecting actions to take in a Markov Decision Process (MDP). A MDP is defined as $\langle S, A, P, r, \rho_0, \gamma, T \rangle$ where S is a set of states, A is a set of actions, $P : S \times A \times S \rightarrow \mathbb{R}_+$ is a transition function, $r : S \times A \rightarrow [R_{min}, R_{max}]$ is a reward function, $\gamma \in [0, 1]$ is a discount factor, and T is a time horizon. The policy $\pi : S \times A \rightarrow \mathbb{R}_+$ is trained to maximize the expected discounted return $\eta(\pi_\theta) = \mathbb{E}_\tau \left[\sum_{t=0}^{T} \gamma^t r(s_t, a_t) \right]$, where τ denotes a trajectory $\tau = (s_0, a_0, \dots)$ sampled according to π_θ with $s_0 \sim \rho_0(s_0), a_t \sim \pi_\theta(a_t \mid s_t)$, and $s_{t+1} \sim P(s_{t+1} \mid s_t, a_t)$. In the MDP corresponding to our implementation of architecture search, each action a defines a neural

network architecture and A is our architecture search space. The set of states is $S = \{s_{init}, s_{trained}\}$, where s_{init} is an initial state in which no architecture has been trained and $s_{trained}$ is a state in which an architecture has been trained. The reward for any architecture and initial state $R : (a, s_{init}) \rightarrow R_+$ is a function of the validation accuracy of the trained neural network with architecture a. Given an initial state s_{init} and an architecture a, the environment will always transition to the trained state i.e. $P(s_{trained} \mid s_{init}, a) = 1.0$. Our objective in this framework is to learn an optimal policy π^*_θ that selects the architecture(s) with the highest validation accuracy.

It is the case that many convolutional architectures such as VGG, ResNet, and Inception have a modular design. These networks repeat a pattern of operations, which we refer to as a convolutional "cell". For instance, the VGG cell contains three 3x3 convolutions followed by a maxpool. Taking inspiration from [2], we define our search space over cells rather than entire architectures. This approach has two advantages over predicting which operation to apply at every layer: 1. it reduces the search space and saves time 2. it reduces the likelihood of "overfitting" to the validation set. In the approach section, we describe exactly how a cell is defined and how a network is constructed given a cell architecture.

In addition to searching for the convolution cell that achieves maximum validation accuracy on MNIST, we show it is simple to modify the reward function such that architecture search finds the best architecture with fewer than n parameters. This extension could be particularly useful when searching for architectures to be run on embedded devices, which often have specific memory constraints. Moreover, our approach is flexible and can accommodate using other metrics, such as maximum inference time, in place of parameter count. To the best of our knowledge this is a new extension of architecture search that has not been explored in other publications.

2 Related Work

The process of manually designing machine learning models is difficult because the search space of all possible models can be combinatorially large. Hence, the process of designing networks often takes a significant amount of time and experimentation by those with significant machine learning expertise. Recently, the idea of using certain types of neural networks (LSTM RNN's) to automatically generate neural network architectures consisting of convolutional cells has been attracting a lot of attention. Evolutionary algorithms and reinforcement learning have shown great promise among many algorithms that have been studied in this aspect. Recent works such as Zoph & Le(2016)[1] focus on learning architectures for large academic datasets like ImageNet and COCO datasets that vastly outperform state-of-the-art-models.

The design of our search-action space is inspired from LSTMs, and Neural Architecture Search Cell[1]. The modular structure of the convolutional cell is also related to previous methods on ImageNet such as VGG, Inception, and ResNet. We constrain the search to finding a good convolutional cell design, and simply stack it to handle inputs of arbitrary spatial dimensions and filter depth. The controller in NASnet is auto-regressive, which means it predicts hyperparameters one a time, conditioned on previous predictions. Eventually the controller learns to assign high probability to areas of architecture space that achieve better accuracy on the validation dataset, and low probability to areas of architecture space that score poorly. Thus, the controller learns directly from the reward signal.

Our work is most inspired by two consecutive works of Zoph & Le. Zoph & Le(2016)[1] used reinforcement learning to train a recurrent network that generates descriptions of neural networks to minimize validation error, and found convolutional and LTSM architectures that performed competitively in CIFAR-10 and Penn Treebank datasets, respectively. Using this approach, Zoph et al. (2017)[2] learn a convolutional cell on the CIFAR-10 dataset that can be transferred to the ImageNet dataset. The architecture obtained by stacking these convolutional cells is called NasNet. A key element of NasNet is to design the search space S to generalize across problems of varying complexity and spatial scales. Applying NasNet directly on the ImageNet dataset would be very expensive and require months to complete an experiment. However, if the search space is properly constructed, architectural elements can transfer across datasets. By stacking together more of this cell, they achieve better top-1 accuracy than the best human-invented architectures with less computation.

Q-Learning has also been used to automate the network design process. Baker et al.(2017)[3] train a learning agent to sequentially choose CNN layers using Q-learning with ϵ-greedy exploration

strategy and experience replay. They beat existing networks designed with only standard convolution, pooling and fully-connected layers. But reinforcement learning is more popular in comparison to other approaches. Cai et al. (2017)[4] train a reinforcement learning agent to grow the depth or layer width of a neural network, allowing previously learned weights to be reused. Li & Malik, 2016[5] apply the idea of using reinforcement learning to find update policies for another network. Another related work is the idea of learning to learn or meta-learning (Thrun & Pratt, 2012)[6], a general framework of using information learned in one task to improve a future task.

In addition, there has been some related work in hyperparameter optimization (Bergstra et al., 2011; Bergstra & Bengio, 2012; Snoek et al., 2012; 2015; Saxena & Verbeek, 2016). It is difficult to use them generate variable-length outputs that specify the network configuration and have been observed to work provided a good initial model (Bergstra & Bengio, 2012; Snoek et al., 2012; 2015). Then, there are Bayesian optimization methods that allow to search non fixed length architectures (Bergstra et al., 2013; Mendoza et al., 2016), but they are less general and less flexible.

3 Approach

Our work extends the Neural Architecture Search (NAS) framework put forward in [1]. In this framework, a policy network π_θ predicts a mini-batch of cell architectures A_{mini}, which in turn define a min-batch of child networks C_{mini}. The child networks are then trained until convergence on the MNIST dataset and their validation accuracies are used as the reward to update π_θ. Pseudo code for this training loop is given below.

Pseudo Code: NAS Training Loop

---

for $k = 0, 1, 2, \ldots$ **do**

1. Sample a mini-batch of architectures A_{mini} from π_θ
2. Train child networks C_{mini} on MNIST and report reward R_{mini}
3. Form a training dataset $D = (A_{mini}, R_{mini})$
4. Compute policy update $\theta_{k+1} = \text{argmax}_\theta L^{CLIP}_{\theta_k}(\theta)$ by taking K steps of minibatch SGD (with Adam) using proximal policy optimization (PPO)

3.1 Search Space

We experiment with two search spaces A_{small} and A_{large}. A_{small} is a relatively small search space where a convolutional cell is defined by two operations $\langle op_1, op_2 \rangle$ applied consecutively, and the list of operations is as follows:

- 1x3 then 3x1 conv
- 3x3 conv
- 3x3 depthwise-seperable conv
- 5x5 conv
- 7x7 conv
- 2x2 max pooling
- 2x2 average pooling
- identity

The advantage of using A_{small} is that there are only $8 \times 8 = 64$ possible child networks. As a result, we can pre-compute the exact values of the reward function (child network validation accuracies) prior to training π_θ. Then during the training procedure, we can use these pre-computed rewards to train π_θ more quickly. As we show in the experiments section, even on this small search space, NAS takes hundreds of samples to converge. With our limited compute (1 GPU), it was infeasible for us to train hundreds of child network architectures. Pre-computing the reward function allows us to both run experiments in a shorter amount of time and obtain accurate results. We provide more information on the pre-computed reward function in the experiments section.

While A_{small} is useful for analyzing the sample complexity and performance of NAS, it does not demonstrate the ability NAS to generate complex architectures and learn without exploring the entire search space. For this reason, we also experiment with a much larger search space A_{large}, which is a subset of the search spaced used in [2]. In A_{large}, a convolutional cell is defined as B convolutional

blocks, where a block takes the form given in figure 1. Each convolutional block requires selecting four parameters $\langle h_a, h_b, op_1, op_2 \rangle$, where

- h_a is the hidden layer input to op_1
- h_b is the hidden layer input to op_2
- op_1 is an operation from the list given above
- op_2 is an operation from the list given above

As illustrated in figure 1, the representations produced by $op_1(h_a)$ and $op_2(h_b)$ in block B_i are combined using filter-wise concatenation to form a new hidden layer h_i. The hidden layer h_i can be selected as the value for the parameters h_a, h_b in the following blocks. This structure allows for cells in this search space to form skip connections between layers within a cell.

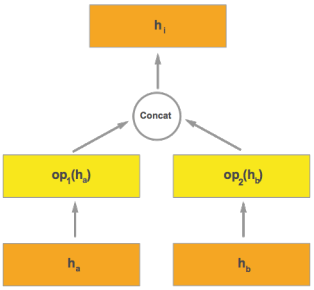

Figure 1: A_{large} cell-block structure

3.2 Controller Network

We use a LSTM with one hundred hidden units to approximate our policy π_θ. The outputs of the LSTM are softmax probability distributions over the operations, and or hidden states, that define the convolutional cell in our search space. The softmax distribution produced at time-step t is fed as the input to the LSTM at time-step $t + 1$ to condition future parameters on the parameters selected so far. To sample a child network, we sample the values of the parameters that define a cell according to their respective softmax probabilities. The controller is trained to maximize the validation accuracy of the sampled child networks using Proximal Policy Optimization (PPO). Becuase cells architectures in A_{small} and A_{large} differ, our controller architectures for A_{small} and A_{large} differ as well. The controllers for A_{small} and A_{large} are detailed below:

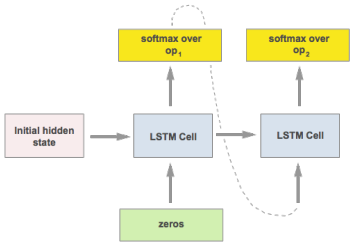

Figure 2: Controller architecture for A_{small}

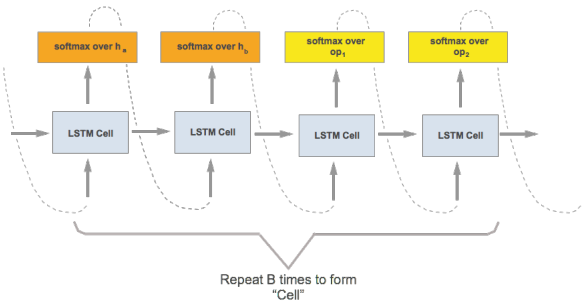

Figure 3: Controller architecture for A_{large}

3.3 Child Network

Given a cell architecture a, the corresponding child network C is defined by the sequence: a-maxpool-a-maxpool-fc-softmax where maxpool is a 2x2 max pooling layer and fc is a 4096 hidden unit fully

connected layer with dropout. We follow the convention set by VGG and double the number of filters used in the convolutional cell whenever the spatial dimension is reduced by a factor of two. Specifically, we use 64 filters in the first convolutional cell and 128 filters in the second convolutional cell. This child network structure is illustrated in figure 4 below. Child networks are trained on the MNIST dataset and evaluated on a held out validation set. Although it is beyond the scope of this work, it is likely that stacking a greater number of convolutional cells could increase performance and allow the cell architecture to "scale" to larger datasets such as ImageNet.

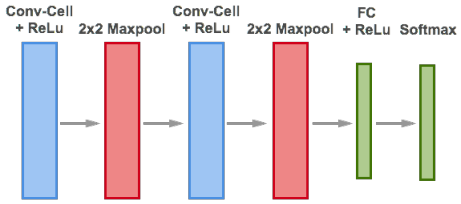

Figure 4: Child network architecture

4 Experiments

In this section, we discuss the three NAS experiments we ran on the MNIST dataset. There are two factors that differentiate these experiments. Namely, the search space used in the experiment (A_{small} or A_{large}) and the reward function used in the experiment (r_{val} or r_{param}). We explicitly define both of these reward functions in the subsection below. Our experiments are as follows:

1. NAS applied to A_{small} using the r_{val} reward function
2. NAS applied to A_{small} using the r_{param} reward function
3. NAS applied to A_{large} using the r_{val} reward function

To analyze the performance of NAS, we plot of the mean, min, max, and standard deviation of the reward over training episodes. Here an episode corresponds to one iteration of the NAS training loop wherein we sample a mini-batch of 20 child network architectures. For both of our experiments on the A_{small} search space, we pre-compute the values of our reward function prior to training the NAS controller to dramatically speed up training time.

4.1 Reward functions

The validation accuracy reward function, r_{val}, is defined as $r_{val} = Val(a)$, where $a \in A_{small}$ and $Val(a)$ denotes the validation accuracy of the child network with cell architecture a on the MNIST validation set.

The maximum parameter reward function, r_{param}, is meant to constrain the search space such that NAS finds the best architecture with fewer than n parameters. Let P_a be the total number of trainable parameters in the child network with cell architecture $a \in A_{small}$. Then r_{param} is defined as follows:

$$r_{param} = \begin{cases} -100.0 & P_a \geq n \\ r_{val} & P_a < n \end{cases}$$

4.2 Dataset

All of the child networks in our experiments are trained and evaluated on the MNIST dataset. We choose the MNIST dataset because it is both a standard computer vision benchmark and a relatively small dataset, which allows us to minimize training time. The MNIST dataset contains $70,000$ black and white images of hand written digits. We use $54,000$ images for the training set, $6,000$ images for the validation set, and $10,000$ images for the test set. We train each child network on the training split for 10 epochs using a negative log likelihood loss and the ADAM optimizer with a learning rate

of 0.0001. We found empirically that after 10 epochs the validation accuracy of the child network converged.

4.3 Results for A_{small} using r_{val}

The reward plot for A_{small} using r_{val} is given in the figure below. There are two takeaways from this figure. Firstly, we see that NAS is successful in finding the best cell architecture in the search space; this is illustrated in the graph by the min and mean validation accuracy (reward) converging to the maximum validation accuracy. Secondly, the plot demonstrates that NAS has poor sample complexity (requires many samples to converge). As shown in the graph, convergence occurs after approximately 200 episodes, which would require training of $4,000$ child networks if the values of the reward function were not pre-computed. The two cells with the highest validation accuracy found by NAS on this search space are $\langle$5x5 conv, 1x3 then 3x1 conv$\rangle$ and $\langle$3x3 seperable conv, 7x7 conv$\rangle$. We give the validation and test accuracies for the child networks corresponding to these cells in table 1.

Table 1: Performance of best cells for A_{small} and r_{val}

	Validation Accuracy	**Test Accuracy**
$\langle$5x5 conv, 1x3 then 3x1 conv$\rangle$	98.85	98.88
$\langle$3x3 seperable conv, 7x7 conv$\rangle$	98.85	98.2

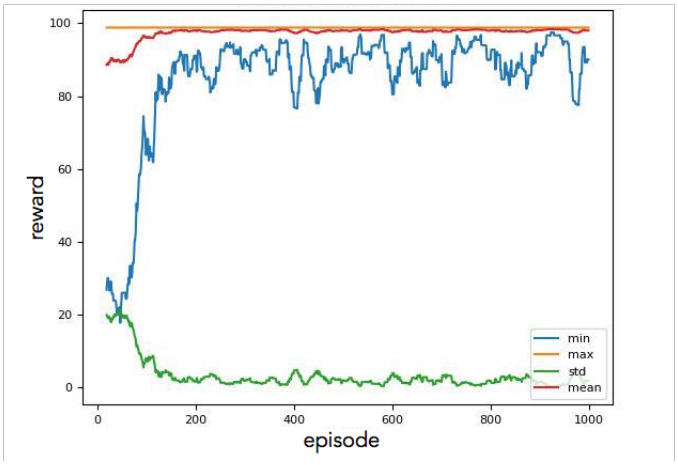

Figure 5: Small search space validation reward plot

4.4 Results for A_{small} using r_{param}

As discussed above, the maximum parameter reward function associates all architectures that have more than n parameters with a negative reward. This negative reward discourages the controller from selecting architectures with more than n parameters. We choose $n = 150,000$ as our parameter limit because it partitions the search space such that only cells with at most one convolutional operation from the set $\{$3x3 seperable conv, 3x3 conv$\}$ are associated with positive reward.

The reward plot for this experiment is given in figure 6. It is clear this graph that finding the best architecture with fewer than $150,000$ parameters is more difficult than simply finding the best architecture with no parameter limit. While the reward plot for r_{val} converges after 200 episodes, the reward plot for r_{param} does not fully converge within 1000 episodes. This is evidenced by the fact that, even after 1000 episodes, the standard deviation of r_{param} is high and the minimum reward appears to still be increasing. That said, the controller does quickly learn **not** to predict architectures with negative reward as you can see by the rapid increase in reward from -25 to

50 within the first 100 episodes. In this experiment, NAS was unable to find the best architecture with fewer than $150,000$ paramters, $\langle$3x3 conv, max$\rangle$. However, it was able to find the second best architecture, $\langle$3x3 conv, avg$\rangle$. We give the parameter counts and validation accuracies for both of these architectures in the following table.

Table 2: Performance of best and second best cells for A_{small} and r_{param}

	Parameter Count	**Validation Accuracy**
$\langle$3x3 conv, max$\rangle$	$117,972$	98.61
$\langle$3x3 conv, avg$\rangle$	$117,972$	98.01

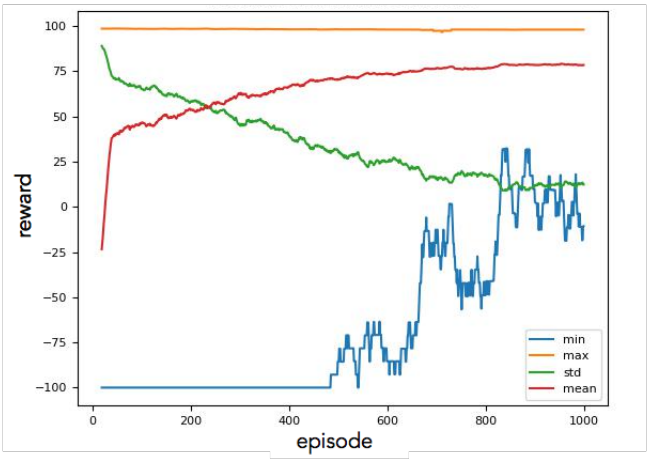

Figure 6: Small search space maximum parameter reward plot

4.5 Results for A_{large} and r_{val}

While the experiments on the search space A_{small} allow us to analyze the performance of NAS in detail, they do not illustrate how NAS can be used to generate novel and complex cell architectures. For this reason, we also experiment with A_{large}. Unfortunately, we did not have the compute necessary to train NAS on enough samples for the reward plot to show any signs of improvement on A_{large}. For this reason we do not give a reward plot for this experiment. Instead, we report the performance of the best cell NAS has found to-date and provide a digram of this cell architecture in figure 7.

Table 3: Performance of best cell for A_{large} and r_{val}

	Validation Accuracy	**Test Accuracy**
Complex convolutional cell	41.41	42.97

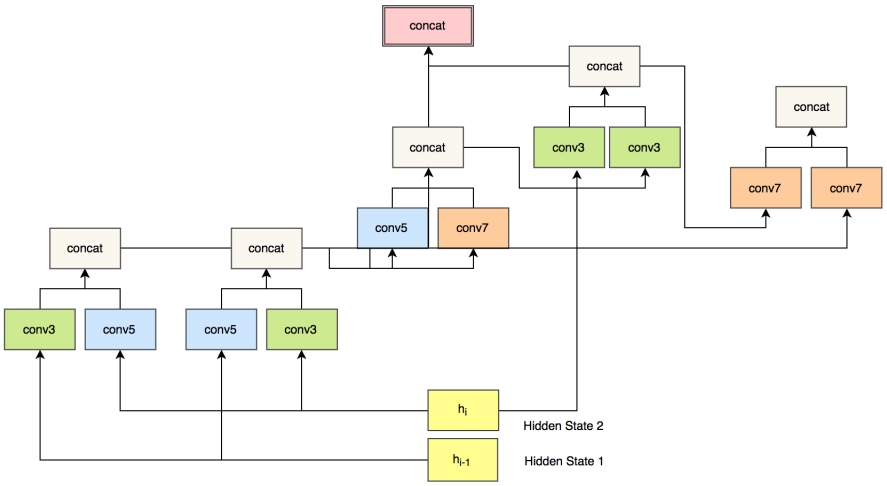

Figure 7: Complex convolutional cell architecture

5 Conclusion

In this work, we demonstrate that it is possible to use NAS to find high performing convolutional cells on the MNIST dataset. In addition, we show that it is relatively simple to modify the reward function to account for computational requirements. Although, NAS succeeds at finding architectures that perform well on MNIST, the number of samples necessary to find these architectures is very large. We see the poor sample complexity of NAS as the main limitation associated with this method. Even on the relatively small MNIST dataset, NAS required thousands of samples to converge, and we believe that tens of thousands of samples would be required for convergence on larger dataset such as CIFAR and ImageNet. Therefore, reducing the sample complexity of NAS by developing more data efficient reinforcement learning algorithms is an important area for future work. In addition, we think it would be interesting to apply the NAS framework to design other neural network components such as attention or memory mechanisms.

References

[1] Zoph and Le. “Neural Network architecture search with reinforcement learning”. In: *ICLR*. 2016.

[2] Zoph et Al. “Learning Transferable Architectures for Scalable Image Recognition”. In: *arXiv:1707.07012*. 2017.

[3] Baker et Al. “Designing neural network architectures using reinforcement learning”. In: *ICLR*. 2017.

[4] Cai et Al. “Efficient Architecture Search by Network Transformation”. In: *arXiv:1707.04873*. 2017.

[5] Li and Malik. “Learning to Optimize”. In: *arXiv:1606.01885*. 2016.

[6] Sebastian Thrun and Lorien Pratt. *Learning to Learn*. 1998.

[7] Liu Et Al. “Progressive Neural Architecture Search”. In: *arXiv:1712.00559*. 2017.

Appendix

5.1 Contributions

The project two components: 1. the controller LSTM 2. child network generation. Jason Krone was responsible for all code related to controller LSTM, and Aashima Arora was responsible for all of the code necessary for child network generation.

We list the files written by each group member below:

- Jason Krone : controller.py, storage.py, main.py, visualize.py
- Aashima Arora : child.py, child-complex.py

All of the code involved in this project was written by Jason and Aashima and not taken from the internet. We list our scource code in the following pages.

5.2 Code

Controller

1. controller.py

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.autograd import Variable


class ControllerLSTM(nn.Module):
    def __init__(self, hidden_dim=100, out_dims=[7, 7, 2, 2, 7, 7, 2, 3, 7, 7, 2]):
        '''
        default out dims assume 3 blocks per cell
        '''
        super(ControllerLSTM, self).__init__()
        self.hidden_dim = hidden_dim
        self.out_dims    = out_dims

        self.lstm = nn.LSTMCell(max(out_dims), hidden_dim)
        for weight in self.lstm.parameters():
            weight.data.uniform_(-0.1, 0.1)

        self.hidden2out_layers = nn.ModuleList([nn.Linear(hidden_dim, out_dim) for out_

    def init_hidden(self, batch_size):
        ''' use weight.new so that parameters on correct device cpu or gpu'''
        weight = next(self.parameters()).data
        return (Variable(weight.new(batch_size, self.hidden_dim).zero_()),
                Variable(weight.new(batch_size, self.hidden_dim).zero_()))

    def forward(self, hidden, is_volatile=False, in_fun=F.softmax, out_fun=F.softmax):
        '''
        computes over discrete distributions of out_dim sizes
        returns shape: [len(out_dims), max(out_dims)]
        note: zero pads forward(out_dims[i]) if out_dims[i] < max(out_dims)
        '''
        N = len(self.out_dims)
        max_out_dim = max(self.out_dims)
        # get the batch size, this can be dependant on input
        batch_size = hidden[0].size()[0]
        hx, cx = hidden
        weight = next(self.parameters()).data

        x = Variable(weight.new(batch_size, max_out_dim).zero_(), volatile=is_volatile)
        outputs = Variable(weight.new(batch_size*N, max_out_dim).zero_())
```

```
        for i, (out_dim, hidden2out) in enumerate(zip(self.out_dims, self.hidden2out_la
            # move forward one time, step
            hx, cx = self.lstm(x, (hx, cx))
            out = hidden2out(hx.view(batch_size, -1))
            outputs[i*batch_size:(i+1)*batch_size, 0:out_dim] = out_fun(out)
            # pad input to LSTM at next time step with zeros
            x_in = in_fun(out)
            x = Variable(x_in.data.new(batch_size, max_out_dim).zero_())
            x[:, 0:out_dim] = x_in
        return outputs

    def act(self, hidden, deterministic=False):
        '''
        selects len(out_dims) actions, which specificy a cnn architecture
        returns shape: [batch_size*len(out_dims), 1]
        '''
        weight = next(self.parameters()).data
        probs = self(hidden, is_volatile=True)
        if deterministic is False:
            # multinomial only takes input with two dims
            actions = probs.multinomial()
        else:
            actions = probs.max(1, keepdim=True)[1]
        return actions

    def evaluate_actions(self, hidden, actions, is_volatile):
        '''
        returns the joint log probabilities and entropy of the given actions under the
        '''
        batch_size = hidden[0].size()[0]
        probs = self(hidden, is_volatile=is_volatile)
        log_probs = self(hidden, is_volatile=is_volatile, in_fun=F.softmax, out_fun=F.l
        log_probs = log_probs.view(hidden[0].size()[0], -1)
        # get probs of the given actions actions
        action_log_probs = log_probs.gather(1, actions)
        action_log_probs = action_log_probs.prod(1)
        dist_entropy = -(log_probs * probs).sum(-1).mean()
        return action_log_probs, dist_entropy


if __name__ == '__main__':
    lstm = ControllerLSTM()
    hidden = lstm.init_hidden(2)
    print('out shape:', lstm(hidden).size())
    actions = lstm.act(hidden)
    print('act non-deterministic shape:', actions.size())
    print('act deterministic shape:', lstm.act(hidden, deterministic=True).size())
    print('evaluate actions shape:', lstm.evaluate_actions(hidden, actions, True))
```

2. storage.py

```
import torch
from torch.utils.data import Dataset, DataLoader, TensorDataset
from torch.utils.data.sampler import RandomSampler
from torch.autograd import Variable
import numpy as np


class Rollouts(object):
    def __init__(self, actions, rewards, batch_size, max_iter, use_cuda):
        self.rewards = rewards
        self.actions = actions
        self.batch_size = batch_size
        self.use_cuda = use_cuda
        self.max_iter = max_iter
```

```
        self.cur_iter  = 0

    def get_mini_batch(self):
        n, m = self.actions.size()
        batch_idxs = np.random.choice(n, self.batch_size)
        rewards_batch = torch.Tensor(self.batch_size)
        actions_batch = torch.LongTensor(self.batch_size, m)
        if self.use_cuda:
            rewards_batch = rewards_batch.cuda()
            actions_batch = actions_batch.cuda()

        for i, j in enumerate(batch_idxs):
            rewards_batch[i] = self.rewards[j]
            actions_batch[i] = self.actions[j]
        return (actions_batch, rewards_batch)

    def __iter__(self):
        return self

    def __next__(self):
        return self.next()

    def next(self):
        if self.cur_iter < self.max_iter:
            self.cur_iter += 1
            return self.get_mini_batch()
        else:
            raise StopIteration


class Storage(object):
    def __init__(self, num_updates, num_trajectories, num_steps, batch_size, action_sha
        self.num_updates = num_updates
        self.num_trajectories = num_trajectories
        self.num_steps = num_steps
        self.batch_size = batch_size
        self.action_names = action_names

        N = num_updates * num_trajectories
        self.rewards = torch.zeros(N, 1)
        self.actions = torch.zeros(N, action_shape)
        # smallest empty index
        self.rollout_idx = 0

        self.batch_size = batch_size
        M = num_steps * batch_size * num_updates
        self.loss  = torch.zeros(M, 1)
        self.policy_loss = torch.zeros(M, 1)
        self.policy_entropy = torch.zeros(M, 1)
        self.loss_idx = 0

    def cuda(self):
        self.rewards = self.rewards.cuda()
        self.actions = self.actions.cuda()
        self.loss = self.loss.cuda()
        self.policy_loss = self.policy_loss.cuda()
        self.policy_entropy = self.policy_entropy.cuda()

    def insert_rollout(self, actions, rewards):
        start, end = self.rollout_idx, self.rollout_idx + self.num_trajectories
        self.actions[start:end].copy_(actions)
        self.rewards[start:end] = rewards
        self.rollout_idx += self.num_trajectories

    def get_numpy_rollouts(self):
```

```
        actions = self.actions[0:self.rollout_idx].numpy()
        rewards = self.rewards[0:self.rollout_idx].numpy()
        rewards = np.squeeze(rewards)
        return actions, rewards

    def get_numpy_losses(self):
        loss = self.loss[0:self.loss_idx].numpy()
        policy_loss = self.policy_loss[0:self.loss_idx].numpy()
        policy_entropy = self.policy_entropy[0:self.loss_idx].numpy()
        return loss, policy_loss, policy_entropy

    def insert_loss(self, loss, policy_loss, policy_entropy):
        start, end = self.loss_idx, self.loss_idx + self.batch_size
        self.loss[start:end].copy_(loss)
        self.policy_loss[start:end].copy_(policy_loss)
        self.policy_entropy[start:end].copy_(policy_entropy)
        self.loss_idx += self.batch_size

    def write_to_file(self, rollout_path, loss_path):
        # save as csv of rollouts in logdir
        rewards = self.rewards.numpy()[0:self.rollout_idx]
        actions = self.actions.numpy()[0:self.rollout_idx]
        rollout_data = np.concatenate((actions, rewards), axis=1)
        np.savetxt(rollout_path, rollout_data, delimiter=',')

        # save csv of loss data in logdir
        loss = self.loss.numpy()[0:self.loss_idx]
        policy_loss = self.policy_loss.numpy()[0:self.loss_idx]
        policy_entropy = self.policy_entropy.numpy()[0:self.loss_idx]
        loss_data = np.concatenate((loss, policy_loss, policy_entropy), axis=1)
        np.savetxt(loss_path, loss_data, delimiter=',')

    def load_from_file(self, rollout_path, loss_path):
        rollout_data = np.genfromtxt(rollout_path, delimiter=',')
        actions, rewards = rollout_data[:, 0:2], rollout_data[:, 2]
        self.rollout_idx = actions.shape[0]
        self.actions[0:self.rollout_idx] = torch.Tensor(actions)
        self.rewards[0:self.rollout_idx] = torch.Tensor(rewards)

        loss_data = np.genfromtxt(loss_path, delimiter=',')
        loss, policy_loss, policy_entropy = loss_data[:, 0], loss_data[:, 1], loss_data
        self.loss_idx = loss.shape[0]
        self.loss[0:self.loss_idx] = torch.Tensor(loss)
        self.policy_loss[0:self.loss_idx] = torch.Tensor(policy_loss)
        self.policy_entropy[0:self.loss_idx] = torch.Tensor(policy_entropy)


def get_reward_matrix(operations, max_params):
    f = open('params_table.log', 'r')
    lines = [l.strip() for l in f.readlines()]
    operation2idx = {op : idx for idx, op in enumerate(operations)}
    accuracy_mtrx = np.zeros((8, 8))
    for l in lines:
        split = l.split('␣')
        op1, op2 = split[1], split[3]
        params, accuracy = int(split[4]), float(split[6])
        if params > max_params:
            accuracy = -100.0
        idx1, idx2 = operation2idx[op1], operation2idx[op2]
        accuracy_mtrx[idx1][idx2] = accuracy
    flat = accuracy_mtrx.flatten()
    var  = Variable(torch.from_numpy(flat).cuda().type(torch.FloatTensor))
    return var
```

```
if __name__ == '__main__':
    storage = Storage(10, 4, 5, 2, 2)
    # test rollout insert
    actions = torch.Tensor([[1, 1], [2, 2], [3, 3], [4, 4]])
    rewards = torch.Tensor([5, 6, 7, 8])
    print('rollout no insert:', storage.actions, storage.rewards)
    storage.insert_rollout(actions, rewards)
    print('rollout 1 insert:', storage.actions, storage.rewards)
    storage.insert_rollout(actions, rewards)
    print('rollout 2 insert:', storage.actions, storage.rewards)
    # test loss insert
    loss = torch.Tensor([0.1, 0.1])
    policy_loss = torch.Tensor([0.2, 0.2])
    policy_entropy = torch.Tensor([0.3, 0.3])
    print('loss no insert:', storage.loss, storage.policy_loss, storage.policy_entropy)
    storage.insert_loss(loss, policy_loss, policy_entropy)
    print('loss 1 insert:', storage.loss, storage.policy_loss, storage.policy_entropy)
    storage.insert_loss(loss, policy_loss, policy_entropy)
    print('loss 2 insert:', storage.loss, storage.policy_loss, storage.policy_entropy)
    storage.write_to_file('temp_rollout.csv', 'temp_loss.csv')

    rollouts = Rollouts(actions, rewards, 2, 10)
    for i, (a, r) in enumerate(rollouts):
        print('i:', i)
        print('action:', a)
        print('reward:', r)
```

3. main.py

```
import torch
import torch.optim as optim
from torch.autograd import Variable
import argparse
import copy
import os
import logging
import numpy as np
from datetime import datetime
from visdom import Visdom

from visualize import visdom_plot
from controller import ControllerLSTM
from storage import *
from child import *


DIR = os.path.dirname(__file__)

parser = argparse.ArgumentParser()
parser.add_argument('--lr', type=float, default=0.00035, help='Learning rate step-size'
parser.add_argument('--e', type=float, default=0.00001, help='Entropy penalty')
parser.add_argument('--alpha', type=float, default=0.05, help='Decay rate for running m
parser.add_argument('--ppo_clip', type=float, default=0.2, help='Clip hyperparameter')
parser.add_argument('--entropy_coef', type=float, default=0.0001, help='Entropy penalty
parser.add_argument('--seed', type=int, default=1, help='Random seed to be used')
parser.add_argument('--num_blocks', type=int, default=1, help='Number of blocks per con
parser.add_argument('--ops_per_block', type=int, default=2, help='Number of operations
parser.add_argument('--num_updates', type=int, default=1000, help='Number of updates to
parser.add_argument('--num_steps', type=int, default=5, help='Number of mini batch batc
parser.add_argument('--batch_size', type=int, default=10, help='Mini batch size for PPO
parser.add_argument('--num_trajectories', type=int, default=20, help='Number of archite
parser.add_argument('--log_dir', type=str, default=os.path.join(DIR, '../log'), help='l
parser.add_argument('--vis_rate', type=int, default=1, help='number of updates before u
parser.add_argument('--checkpoint_dir', type=str, default=os.path.join(DIR, '../checkpo
parser.add_argument('--checkpoint_rate', type=int, default=5, help='number of iteration
```

```
parser.add_argument('--checkpoint', type=str, default=None, help='checkpoint to run')
parser.add_argument('--use_cuda', type=bool, default=False, help='should use cuda if av
parser.add_argument('--true_reward', action='store_true', default=False, help='Train ar
parser.add_argument('--max_params', type=float, default=float('inf'), help='Maximum num
parser.add_argument('--many_hidden', action='store_true', default=False, help='Use a hi

args = parser.parse_args()
args.cuda = torch.cuda.is_available() and args.use_cuda
print("CUDA", args.cuda)
print("max params:", args.max_params)
torch.manual_seed(args.seed)
if args.cuda:
    torch.cuda.manual_seed(args.seed)

FILE_PREFIX = datetime.today().strftime('%Y%m%d_%H%M%S')+'_lr_'+str(args.lr)+'_e_'+str(
              '_num_blocks_'+str(args.num_blocks)+'_num_updates_'+str(args.num_updates
              '_ppo_clip_'+str(args.ppo_clip)+'_alpha_'+str(args.alpha)+'_enrtopy_coef
              '_restored_'+str(args.checkpoint != None)+'_true_reward_'+str(args.true_r
              '_batch_size_'+str(args.batch_size)+'_num_trajectories_'+str(args.num_t

def cuda_safe(var):
    if args.cuda:
        return var.cuda()
    else:
        return var

OPERATIONS = ['twoconv', 'conv3', 'conv5', 'conv7', 'max', 'avg', 'identity', 'sep']
REWARD_MATRIX = cuda_safe(get_reward_matrix(OPERATIONS, args.max_params))
print('reward matrix:', REWARD_MATRIX)
ACTION2IDX = cuda_safe(Variable(torch.FloatTensor([len(OPERATIONS), 1])))

# taken from https://github.com/pytorch/examples/blob/master/word_language_model/main.p
def repackage_hidden(h):
    """Wraps hidden states in new Variables, to detach them from their history."""
    if type(h) == Variable:
        return Variable(h.data)
    else:
        return tuple(repackage_hidden(v) for v in h)

def ppo_loss(controller, old_controller, hidden, actions, rewards, baseline):
    advantage = cuda_safe(Variable(rewards - baseline.data))
    # create a new actions variable so that input can be used for a gradient, need acti
    action_log_prob, dist_entropy = controller.evaluate_actions(hidden, Variable(action
    # no gradient will be produced from old_action_log_prob,
    old_action_log_prob, _ = old_controller.evaluate_actions(hidden, actions, is_volati
    # create a new variable to detch data from the graph
    ratio = torch.exp(action_log_prob - Variable(old_action_log_prob.data))
    surr1 = ratio * advantage
    surr2 = torch.clamp(ratio, 1.0 - args.ppo_clip, 1.0 + args.ppo_clip) * advantage
    action_loss = torch.min(surr1, surr2).mean()
    loss = action_loss - dist_entropy * args.entropy_coef
    return loss, action_loss, dist_entropy


def train(controller, optimizer, baseline, i):
    # plotting
    viz = Visdom()
    win = [None]*5
    # storage
    action_shape = args.ops_per_block * args.num_blocks
    storage = Storage(args.num_updates, args.num_trajectories, args.num_steps, args.bat
    old_controller = copy.deepcopy(controller)
    while i < args.num_updates:
        # collect partial trajectories on policy
        hidden = controller.init_hidden(args.num_trajectories)
```

```
        actions = controller.act(hidden).view(args.num_trajectories, args.num_blocks*ar
        rewards = rewards_for_actions(actions)
        baseline = args.alpha*rewards.mean() + (1.0 - args.alpha)*baseline
        # store trajectories
        storage.insert_rollout(actions.data, rewards.data)
        rollouts = Rollouts(actions.data, rewards.data, args.batch_size, args.num_steps
        for j, (actions_batch, rewards_batch) in enumerate(rollouts): # sample mini_bat
            hidden = controller.init_hidden(args.batch_size)
            loss, policy_loss, policy_entropy = ppo_loss(controller, old_controller, hi
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
            storage.insert_loss(loss.data, policy_loss.data, policy_entropy.data)
        old_controller.load_state_dict(controller.state_dict())
        if (i + 1) % args.checkpoint_rate == 0 and args.true_reward:
            save_checkpoint(controller, optimizer, baseline, i)
            rollout_path = os.path.join(args.log_dir, FILE_PREFIX+'_rollouts.csv')
            loss_path = os.path.join(args.log_dir, FILE_PREFIX+'_loss.csv')
            storage.write_to_file(rollout_path, loss_path)
        if (i + 1) % args.vis_rate == 0:
            win = visdom_plot(viz, win, storage, FILE_PREFIX)
        i += 1
    rollout_path = os.path.join(args.log_dir, FILE_PREFIX+'_rollouts.csv')
    loss_path = os.path.join(args.log_dir, FILE_PREFIX+'_loss.csv')


def rewards_for_actions(actions):
    # probably have to train models in batches of 5
    rewards = None
    if args.true_reward == False:
        actions_temp = cuda_safe(actions.type(torch.FloatTensor))
        idxs = cuda_safe(actions_temp.matmul(ACTION2IDX).type(torch.LongTensor))
        rewards = REWARD_MATRIX.gather(0, idxs)
    else:
        rewards = []
        actions_numpy = actions.data.cpu().numpy()
        actions_numpy = np.split(actions_numpy, args.num_trajectories / 10)
        for a_batch in actions_numpy:
            a1s, a2s = a_batch[:, 0], a_batch[:, 1]
            op1, op2 = [OPERATIONS[a] for a in a1s], [OPERATIONS[a] for a in a2s]
            temp = get_accuracy(op1, op2)
            rewards_batch, params_batch = zip(*temp)
            rewards += rewards_batch
        rewards = Variable(cuda_safe(torch.Tensor(rewards)))
    return rewards


def save_checkpoint(model, optimizer, baseline, epoch):
    save_model = model
    if args.cuda:
        save_model = copy.deepcopy(model).cpu()
    checkpoint_path = os.path.join(args.checkpoint_dir, str(epoch) +'_'+ FILE_PREFIX +
    data = {'epoch':epoch,'baseline' : baseline, 'model_state_dict':save_model.state_di
    torch.save(data, checkpoint_path)


def load_checkpoint(model, optimizer):
    # get most recent file in directory
    logging.info('loading␣from␣checkpoint:␣'+args.checkpoint)
    data = torch.load(args.checkpoint)
    epoch = data['epoch']
    baseline = data['baseline']
    model.load_state_dict(data['model_state_dict'])
    optimizer.load_state_dict(data['optim_state_dict'])
    return model, optimizer, baseline, epoch
```

```
if __name__ == '__main__':
    controller = ControllerLSTM(out_dims=[8, 8])
    controller = cuda_safe(controller)
    optimizer = optim.Adam(controller.parameters(), lr=args.lr)

    if args.checkpoint != None:
        controller, optimizer, baseline, epoch = load_checkpoint(controller, optimizer)
    else:
        baseline = 0.0
        epoch = 0
    controller.train() # set to training mode
    train(controller, optimizer, baseline, epoch)
```

4. visualize.py

```
import glob
import json
import os
from textwrap import wrap

import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from collections import Counter
from scipy.signal import medfilt
matplotlib.rcParams.update({'font.size': 8})


def load_data(storage, smoothing_window=20):
    actions, rewards = storage.get_numpy_rollouts()
    loss, policy_loss, policy_entropy = storage.get_numpy_losses()

    if smoothing_window > 1:
        rewards_smoothed = pd.Series(rewards).rolling(smoothing_window, min_periods=smo

    data_dict = {
       'rewards' : rewards,
       'rewards_smoothed' : rewards_smoothed,
       'actions' : actions,
       'loss' : loss,
       'policy_loss' : policy_loss,
       'policy_entropy' : policy_entropy
    }
    return data_dict

color_defaults = [
    '#1f77b4',  # muted blue
    '#ff7f0e',  # safety orange
    '#2ca02c',  # cooked asparagus green
    '#d62728',  # brick red
    '#9467bd',  # muted purple
    '#8c564b',  # chestnut brown
    '#e377c2',  # raspberry yogurt pink
    '#7f7f7f',  # middle gray
    '#bcbd22',  # curry yellow-green
    '#17becf'   # blue-teal
]

def plot_action_heatmap(viz, win, storage, title, num_prev_rollouts=1):
    window = storage.num_trajectories*num_prev_rollouts
    actions = storage.actions.cpu().numpy().astype(int)[storage.rollout_idx - window: s
```

```
    actions_tup = [tuple(a.tolist()) for a in actions]
    actions, counts = zip(*Counter(actions_tup).items())
    num_actions = len(storage.action_names)
    arange_actions = np.arange(num_actions)
    counts_grid = np.zeros((num_actions, num_actions))
    for acts, c in zip(actions, counts):
        a1, a2 = acts
        counts_grid[a1, a2] = c
    fig, ax = plt.subplots()
    ax.set_title('\n'.join(wrap(title, 60)))
    plt.xticks(arange_actions, storage.action_names)
    plt.yticks(arange_actions, storage.action_names)
    cax = ax.imshow(counts_grid, cmap='hot', interpolation='nearest')
    cticks = sorted(np.unique(counts_grid))
    cbar = fig.colorbar(cax, ticks=cticks)
    plt.draw()
    image = np.fromstring(fig.canvas.tostring_rgb(), dtype=np.uint8, sep='')
    image = image.reshape(fig.canvas.get_width_height()[::-1] + (3, ))
    image = np.transpose(image, (2, 0, 1))
    plt.close(fig)
    return viz.image(image, win)


def visdom_plot(viz, win, storage, title):
    data_dict  = load_data(storage)
    rewards, rewards_smoothed = data_dict['rewards'], data_dict['rewards_smoothed']
    x = np.arange(rewards.size)

    if x is None or rewards is None:
        return win

    # plot rewards
    fig = plt.figure()
    plt.plot(x, rewards, label='without␣smoothing')#, label="{}".format(name))
    plt.plot(x, rewards_smoothed, label='with␣smoothing')
    plt.xlabel('Number␣of␣Timesteps')
    plt.ylabel('Validation␣Accuracy')
    plt.title('\n'.join(wrap(title, 60)))
    plt.legend(loc=4)
    plt.show()
    plt.draw()
    image1 = np.fromstring(fig.canvas.tostring_rgb(), dtype=np.uint8, sep='')
    image1 = image1.reshape(fig.canvas.get_width_height()[::-1] + (3, ))
    image1 = np.transpose(image1, (2, 0, 1))
    plt.close(fig)
    win[0] = viz.image(image1, win[0])

    # plot min/max/mean/std rewards per episode
    rewards = rewards.reshape(-1, storage.num_trajectories)
    episodes = np.arange(rewards.shape[0])
    min_rewards = np.min(rewards, axis=1)
    max_rewards = np.max(rewards, axis=1)
    std_rewards = np.std(rewards, axis=1)
    mean_rewards = np.mean(rewards, axis=1)
    # apply smoothing
    mean_rewards = pd.Series(mean_rewards).rolling(20).mean()
    min_rewards = pd.Series(min_rewards).rolling(20).mean()
    max_rewards = pd.Series(max_rewards).rolling(20).mean()
    std_rewards = pd.Series(std_rewards).rolling(20).mean()

    fig = plt.figure()
    plt.plot(episodes, min_rewards, label='min')
    plt.plot(episodes, max_rewards, label='max')
    plt.plot(episodes, std_rewards, label='std')
    plt.plot(episodes, mean_rewards, label='mean')
```

```
    plt.xlabel('Episodes')
    plt.ylabel('Validation Accuracy')
    plt.title('\n'.join(wrap(title, 60)))
    plt.legend(loc=4)
    plt.show()
    plt.draw()
    image2 = np.fromstring(fig.canvas.tostring_rgb(), dtype=np.uint8, sep='')
    image2 = image2.reshape(fig.canvas.get_width_height()[::-1] + (3, ))
    image2 = np.transpose(image2, (2, 0, 1))
    plt.close(fig)
    win[1] = viz.image(image2, win[1])

    # plot average loss, policy_loss
    N = storage.num_steps*storage.batch_size
    loss, policy_loss = data_dict['loss'], data_dict['policy_loss']
    loss, policy_loss = loss.reshape(-1, N), policy_loss.reshape(-1, N)
    loss, policy_loss = np.mean(loss, axis=1), np.mean(policy_loss, axis=1)
    loss_x = np.arange(loss.size)
    fig = plt.figure()
    plt.ylim(ymin=0, ymax=100)
    plt.plot(loss_x, loss, label='loss')
    plt.plot(loss_x, policy_loss, label='policy loss')
    plt.xlabel('Episodes')
    plt.title('\n'.join(wrap(title, 60)))
    plt.legend(loc=4)
    plt.show()
    plt.draw()
    image3 = np.fromstring(fig.canvas.tostring_rgb(), dtype=np.uint8, sep='')
    image3 = image3.reshape(fig.canvas.get_width_height()[::-1] + (3, ))
    image3 = np.transpose(image3, (2, 0, 1))
    plt.close(fig)
    win[2] = viz.image(image3, win[2])

    # plot policy entropy
    policy_entropy = data_dict['policy_entropy']
    policy_entropy = np.mean(policy_entropy.reshape(-1, N), axis=1)
    fig = plt.figure()
    plt.plot(loss_x, policy_entropy, label='policy entropy')
    plt.xlabel('Episodes')
    plt.title('\n'.join(wrap(title, 60)))
    plt.legend(loc=4)
    plt.show()
    plt.draw()
    image4 = np.fromstring(fig.canvas.tostring_rgb(), dtype=np.uint8, sep='')
    image4 = image4.reshape(fig.canvas.get_width_height()[::-1] + (3, ))
    image4 = np.transpose(image4, (2, 0, 1))
    plt.close(fig)
    win[3] = viz.image(image4, win[3])
    # plot average KL divergence between policies over episodes
    # plot action heatmap
    if storage.rollout_idx > storage.num_trajectories:
        win[4] = plot_action_heatmap(viz, win[4], storage, title)
    return win
```

Child Networks

Code - Smaller Search Space Child Network(child.py)

```
# coding: utf-8

# In[ ]:

import torch
```

```
import torchvision
import torch.nn as nn
import torch.optim as optim
import torchvision.datasets as dsets
import torch.utils.data as data
import torchvision.transforms as transforms
import torch.nn.functional as F
import torch.multiprocessing as mp
from multiprocessing import set_start_method
import copy
import numpy as np
import argparse
from torch.autograd import Variable
parser = argparse.ArgumentParser()
parser.add_argument('--op1', type=str, help='first.conv.operation.in.block')
parser.add_argument('--op2', type=str, help='second.conv.operation.in.block')
#torch.manual_seed(1)
# In[ ]:

# Hyper Parameters
batch_size = 128
valid_size=0.1
num_classes = 10


# In[ ]:

# MNIST Dataset
train_dataset = dsets.MNIST(root='./data/',
                            train=True,
                            transform=transforms.Compose([
                            transforms.ToTensor(),
                            transforms.Normalize((0.1307,), (0.3081,))]),
                            download=True
                            )


valid_dataset = dsets.MNIST(root='./data/',
                            train=False,
                            transform=transforms.Compose([
                            transforms.ToTensor(),
                            transforms.Normalize((0.1307,), (0.3081,))
                            ]),
                            download=True,)


test_dataset = dsets.MNIST(root='./data/',
                           train=False,
                           transform=transforms.Compose([
                            transforms.ToTensor(),
                            transforms.Normalize((0.1307,), (0.3081,))
                            ]))


# In[ ]:

num_train = len(train_dataset)
indices = list(range(num_train))
split = int(np.floor(valid_size * num_train))
train_idx, valid_idx = indices[split:], indices[:split]
train_sampler = data.sampler.SubsetRandomSampler(train_idx)
valid_sampler = data.sampler.SubsetRandomSampler(valid_idx)

# In[ ]:
```

```
# Data Loader (Input Pipeline)
train_loader = torch.utils.data.DataLoader(dataset=train_dataset,
                                           batch_size=batch_size,
                                           sampler=train_sampler
                                           )


valid_loader = torch.utils.data.DataLoader(dataset=valid_dataset,
                                           batch_size=batch_size,
                                           sampler=valid_sampler)


test_loader = torch.utils.data.DataLoader(dataset=test_dataset,
                                          batch_size=batch_size,
                                          shuffle=False)

# In[ ]:

class SeparableConv2d(nn.Module):

    def __init__(self, in_channels, out_channels, dw_kernel,
                         dw_stride, dw_padding, bias=False):
        super(SeparableConv2d, self).__init__()
        self.depthwise_conv2d = nn.Conv2d(in_channels, in_channels,
                                          dw_kernel,
                                          stride=dw_stride,
                                          padding=dw_padding,
                                          bias=bias,
                                          groups=in_channels)
        self.pointwise_conv2d = nn.Conv2d(in_channels, out_channels, 1,
                                          stride=1, bias=bias)

    def forward(self, x):
        x = self.depthwise_conv2d(x)
        x = self.pointwise_conv2d(x)
        return x


# In[ ]:

class Conv(nn.Module):

    def __init__(self, in_channels, out_channels,kernel_size, bias=False):
        super(Conv, self).__init__()
        self.conv = nn.Conv2d(in_channels, out_channels,kernel_size,1,1)
        self.filter = kernel_size
        self.pad = nn.ZeroPad2d((3, 0, 3, 0))

    def forward(self, x):
        (_, C, H, W) = x.data.size()
        if H <= 1 or W <= 1:
            return x
        if H < self.filter:
            x = self.pad(x)
        x = self.conv(x)
        x = F.relu(x)
        return x


# In[ ]:

class TwoConv(nn.Module):

    def __init__(self, in_channels, out_channels, bias=False, dw_padding=1):
        super(TwoConv, self).__init__()
```

```
        self.conv_0 = nn.Conv2d(in_channels, out_channels,(1,3), 1,
                                dw_padding, bias=bias)
        self.conv_1 = nn.Conv2d(out_channels, out_channels,(3,1), 1,
                                0, bias=bias)

    def forward(self, x):
        x = self.conv_0(x)
        x = self.conv_1(x)
        x = F.relu(x)
        return x


# In[ ]:

class TwoSeparables(nn.Module):

    def __init__(self, in_channels, out_channels, dw_kernel, dw_stride,
                       bias=False):
        super(TwoSeparables, self).__init__()
        self.separable_0 = SeparableConv2d(in_channels, in_channels,
                                           dw_kernel, 1,1, bias=bias)
        self.bn_0 = nn.BatchNorm2d(in_channels, eps=0.001, momentum=0.1,
                                           affine=True)
        self.separable_1 = SeparableConv2d(in_channels, out_channels,
                                           dw_kernel, 1, 1, bias=bias)
        self.bn_1 = nn.BatchNorm2d(out_channels, eps=0.001, momentum=0.1,
                                           affine=True)

    def forward(self, x):
        x = self.separable_0(x)
        x = self.bn_0(x)
        x = F.relu(x)
        x = self.separable_1(x)
        x = self.bn_1(x)
        x = F.relu(x)
        return x


# In[ ]:

class NormalCell(nn.Module):

    def identity_func(self,x):
        return x

    def apply_conv_op(self,op,in_channels,out_channels):
        if 'twoconv' in op:
            return TwoConv(in_channels,out_channels),out_channels
        if 'conv3' in op:
            return Conv(in_channels, out_channels,3),out_channels
        if 'conv5' in op:
            return Conv(in_channels, out_channels,5),out_channels
        if 'conv7' in op:
            return Conv(in_channels, out_channels,7),out_channels
        if 'sep' in op:
            return TwoSeparables(in_channels, out_channels, 3, 1,
            bias=False),out_channels
        if 'max' in op:
            return nn.Sequential(nn.MaxPool2d(3, stride=1, padding=1),
            nn.ReLU()),in_channels
        if 'avg' in op:
            return nn.Sequential(nn.AvgPool2d(3, stride=1, padding=1),
            nn.ReLU()),in_channels
        if 'identity' in op:
            return self.identity_func,in_channels
```

```
    def __init__(self, op_left, op_right,
                 in_channels=1, out_channels=64
                ):
        super(NormalCell, self).__init__()

        self.comb_iter_left,in_channels = self.apply_conv_op(op_left,
                                                in_channels,out_channels)
        self.channel_num = in_channels
        self.comb_iter_right,_= self.apply_conv_op(op_right,
                                                in_channels,out_channels)

    def forward(self, x):
        (_, C, H, W) = x.data.size()
        if H <= 1 or W <= 1:
            return x
        x = self.comb_iter_left(x)
        (_, C, H, W) = x.data.size()
        if H <= 1 or W <= 1 or C != self.channel_num:
            return x
        x = self.comb_iter_right(x)
        return x


# In[ ]:

class NasNet(nn.Module):

    def __init__(self, cell_list, num_classes=10):
        super(NasNet, self).__init__()
        self.num_classes = num_classes
        self.normal_hidden = [cell_list[0][0], cell_list[0][1]]
        self.normal_op = [cell_list[1][0],cell_list[1][1]]
        self.cell_0 = NormalCell(self.normal_op[0],self.normal_op[1],
                                 in_channels=1,out_channels=64)
        self.cell_1 = NormalCell(self.normal_op[0],self.normal_op[1],
                                 in_channels=64,out_channels=128)
        #self.cell_2 = NormalCell(self.normal_op[0],self.normal_op[1],
                        in_channels=128,out_channels=256)
        #self.cell_3 = NormalCell(self.normal_op[0],self.normal_op[1],
                                 in_channels=256,out_channels=512)
        #self.cell_4 = NormalCell(self.normal_op[0],self.normal_op[1],
                                 in_channels=512,out_channels=512)
        self.register_parameter('weight', None)
        self.fc = nn.Linear(1024,self.num_classes)
        self.reduction_cell = nn.MaxPool2d(2,stride=None,padding=0)
        #self.dropout_0 = nn.Dropout2d()


    def reset_parameters(self, input,size):
        self.weight = nn.Parameter(input.data.new(1024,size))

    def features(self, x):
        x = self.cell_0(x)
        (_, C, H, W) = x.data.size()
        if H <= 1 or W <= 1:
            return x
        x = self.reduction_cell(x)
        x = F.relu(x)
        #x = F.relu(F.max_pool2d(x, 2))
        x = self.cell_1(x)
        (_, C, H, W) = x.data.size()
        if H <= 1 or W <= 1:
            return x
        x = self.reduction_cell(x)
```

```
        x = F.relu(x)
        #x = self.cell_2(x)
        #(_, C, H, W) = x.data.size()
        #if H <= 1 or W <= 1:
        #    return x
        #x = self.reduction_cell(x)
        #x = F.relu(x)
        #x = self.cell_3(x)
        #x = self.cell_4(x)
        #(_, C, H, W) = x.data.size()
        #if H <= 1 or W <= 1:
        #    return x
        #x = self.reduction_cell(x)
        #x = F.relu(x)
        return x

    def classifier(self, x):
        (_, C, H, W) = x.data.size()
        size = C*H*W
        x = x.view(-1 ,size)
        if self.weight is None:
            self.reset_parameters(x,size)
        x = F.linear(x,self.weight)
        x = F.relu(x)
        x = F.dropout(x, training=self.training)
        x = self.fc(x)
        return x

    def forward(self, x):
        x = self.features(x)
        x = self.classifier(x)
        return F.log_softmax(x)

#Train Parameters
log_interval = 15
epochs = 10


# In[ ]:


def train(epoch,model,f):
    correct = 0
    model.train()
    for batch_idx, (data, target) in enumerate(train_loader):
        data, target = data.cuda(), target.cuda()
        data, target = Variable(data), Variable(target)
        optimizer = optim.Adam(model.parameters(), lr=1e-4)
        #optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)
        optimizer.zero_grad()
        output = model(data)
        loss = F.nll_loss(output, target)
        #loss = nn.CrossEntropyLoss()(output, target)
        loss.backward()
        optimizer.step()
        #print (list(model.parameters())[0].grad)
        prediction = output.data.max(1)[1]
        if batch_idx % log_interval == 0:
            correct += prediction.eq(target.data.view_as(prediction)).cpu().sum()
            accuracy = 100. *correct /batch_size
        log_string = 'Train␣Epoch:␣{}␣[{}/{}'
        '({:.0f}%)]\tLoss:␣{:.6f}\tAccuracy:␣{:.3f}'.format(
                epoch, batch_idx * len(data), len(train_sampler),
                100. * batch_idx / len(train_sampler), loss.data[0],accuracy)
            f.write(log_string)
```

```
            f.write('\n')
            f.flush()
            correct = 0


def evaluate_model(model,f):
    model.eval()
    validation_loss = 0
    correct = 0
    accuracy = 0
    for data, target in valid_loader:
        data, target = data.cuda(), target.cuda()
        data, target = Variable(data, volatile=True), Variable(target)
        output = model(data)
                loss = F.nll_loss(output, target, size_average=False).data[0]
         #Sum batch loss
        validation_loss += loss
         # get the index of the max log-probability
        pred = output.data.max(1, keepdim=True)[1]
        correct += pred.eq(target.data.view_as(pred))
                            .cpu().sum()

    validation_loss /= len(valid_loader.dataset)
    log_string = '\nValidation␣set:␣Average␣loss:'
        '{:.4f},␣Accuracy:␣{}/{}␣({:.0f}%)\n'.format(
    validation_loss, correct, len(valid_sampler),
    100. * correct / len(valid_sampler))''
    accuracy = 100. * correct / len(valid_sampler)
    f.write(log_string)
    f.write('\n')
    f.flush()
    return accuracy

def test_model(model,f):
    model.eval()
    test_loss = 0
    correct = 0
    accuracy = 0
    for data, target in test_loader:
        data, target = data.cuda(), target.cuda()
        data, target = Variable(data, volatile=True), Variable(target)
        output = model(data)
        loss = F.nll_loss(output, target, size_average=False).data[0]
        #Sum batch loss
        test_loss += loss
        # get the index of the max log-probability
        pred = output.data.max(1, keepdim=True)[1]
        correct += pred.eq(target.data.view_as(pred)).cpu().sum()

    size = len(test_loader.dataset)
    test_loss /= size
    log_string = '\nTest␣set:␣Average␣loss:'
    '{:.4f},␣Accuracy:␣{}/{}␣({:.0f}%)\n'.format(
    test_loss, correct, size,
    100. * correct / size)
        accuracy = 100. * correct / size
        f.write(log_string)
        f.write('\n')
    f.flush()



def get_param_size(model):
    params = 0
    for p in model.parameters():
```

```
        tmp = 1
        for x in p.size():
            tmp *= x
            params += tmp
    return params


# In[ ]:
def train_model(epochs,model,index,return_dict):
    f = open('train'+str(index)+'.log','w+')
    val_accuracy= []
    #for p in model.parameters():
    #    print(p.size())
    for epoch in range(1,epochs+1):
        train(epoch,model,f)
        accuracy = evaluate_model(model,f)
        val_accuracy.append(accuracy)
        test_model(model,f)
    accuracy = str(max(val_accuracy))
    f.write('Accuracy:␣'+ accuracy + '\n')
    f.flush()
    return_dict[index] = (accuracy,get_param_size(model))

def get_accuracy(op1,op2):
    try:
        set_start_method('spawn')
    except RuntimeError:
        pass
    manager = mp.Manager()
    return_dict = manager.dict()
    models = []
    #op1 = ['identity','conv3','conv5','max','identity']
    #op2 = ['twoconv','conv5','conv5','avg','identity']
    num_processes = len(op1)
    for i in range(num_processes):
        cell_list = [[0, 1], [op1[i], op2[i]]]
        val_accuracy = []
        #f.write('op1: ' + op1[i] + ', op2: ' + op2[i] + '\n')
        #f.flush()
        model = NasNet(cell_list)
        model.cuda()
        model.share_memory()
        models.append(model)

    processes = []
    for rank in range(num_processes):
        p = mp.Process(target=train_model, args=(epochs,models[rank],rank,return_dict))
        p.start()
        processes.append(p)

    for p in processes:
        p.join()
    return return_dict.values()

# In[ ]:
```

Code - Larger Search Space Child Network(child-complex.py)

```
# coding: utf-8

# In[ ]:

import torch
import torchvision
import torch.nn as nn
```

```
import torch.optim as optim
import torchvision.datasets as dsets
import torch.utils.data as data
import torchvision.transforms as transforms
import torch.nn.functional as F
import copy
import numpy as np
import argparse
import pdb

from torch.autograd import Variable
parser = argparse.ArgumentParser()
parser.add_argument('--op1', type=str, help='first.conv.operation.in.block')
parser.add_argument('--op2', type=str, help='second.conv.operation.in.block')
torch.manual_seed(1)
# In[ ]:

# Hyper Parameters
batch_size = 128
valid_size=0.1
num_classes = 10
f = open('train.log','w+')

# In[ ]:

# MNIST Dataset
train_dataset = dsets.MNIST(root='./data/',
                            train=True,
                            transform=transforms.Compose([
                            transforms.ToTensor(),
                            transforms.Normalize((0.1307,), (0.3081,))]),
                            download=True
                            )


valid_dataset = dsets.MNIST(root='./data/',
                            train=False,
                            transform=transforms.Compose([
                            transforms.ToTensor(),
                            transforms.Normalize((0.1307,), (0.3081,))
                            ]),
                            download=True,
                            )


test_dataset = dsets.MNIST(root='./data/',
                           train=False,
                           transform=transforms.Compose([
                            transforms.ToTensor(),
                            transforms.Normalize((0.1307,), (0.3081,))
                            ]))


# In[ ]:

num_train = len(train_dataset)
indices = list(range(num_train))
split = int(np.floor(valid_size * num_train))
train_idx, valid_idx = indices[split:], indices[:split]
train_sampler = data.sampler.SubsetRandomSampler(train_idx)
valid_sampler = data.sampler.SubsetRandomSampler(valid_idx)

# In[ ]:

# Data Loader (Input Pipeline)
```

```
train_loader = torch.utils.data.DataLoader(dataset=train_dataset,
                                           batch_size=batch_size,
                                           sampler=train_sampler
                                           )


valid_loader = torch.utils.data.DataLoader(dataset=valid_dataset,
                                           batch_size=batch_size,
                                           sampler=valid_sampler)


test_loader = torch.utils.data.DataLoader(dataset=test_dataset,
                                          batch_size=batch_size,
                                          shuffle=False)

# In[ ]:

class SeparableConv2d(nn.Module):

    def __init__(self, in_channels, out_channels, dw_kernel, dw_stride,
                         dw_padding, bias=False):
        super(SeparableConv2d, self).__init__()
        self.depthwise_conv2d = nn.Conv2d(in_channels, in_channels,
                                          dw_kernel,
                                          stride=dw_stride,
                                          padding=dw_padding,
                                          bias=bias,
                                          groups=in_channels)
        self.pointwise_conv2d = nn.Conv2d(in_channels, out_channels, 1,
                                          stride=1, bias=bias)

    def forward(self, x):
        x = self.depthwise_conv2d(x)
        x = self.pointwise_conv2d(x)
        return x


# In[ ]:

class Conv(nn.Module):

    def __init__(self, in_channels, out_channels,kernel_size, bias=False):
        super(Conv, self).__init__()
        self.conv = nn.Conv2d(in_channels, out_channels,kernel_size,1,1)
        self.filter = kernel_size
        self.pad = nn.ZeroPad2d((3, 0, 3, 0))

    def forward(self, x):
        (_, C, H, W) = x.size()
        if H <= 1 or W <= 1:
            return x
        #if H < self.filter:
        #    x = self.pad(x)
        x = self.conv(x)
        x = F.relu(x)
        return x


# In[ ]:

class TwoConv(nn.Module):

    def __init__(self, in_channels, out_channels, bias=False, dw_padding=1):
        super(TwoConv, self).__init__()
        self.conv_0 = nn.Conv2d(in_channels, out_channels,(1,3), 1,
```

```
                        dw_padding, bias=bias)
        self.conv_1 = nn.Conv2d(out_channels, out_channels,(3,1),
                        1,0, bias=bias)

    def forward(self, x):
        x = self.conv_0(x)
        x = self.conv_1(x)
        x = F.relu(x)
        return x


# In[ ]:

class TwoSeparables(nn.Module):

    def __init__(self, in_channels, out_channels, dw_kernel, dw_stride,
                         bias=False):
        super(TwoSeparables, self).__init__()
        self.separable_0 = SeparableConv2d(in_channels, in_channels,
                        dw_kernel, 1,1, bias=bias)
        self.bn_0 = nn.BatchNorm2d(in_channels, eps=0.001, momentum=0.1,
                        affine=True)
        self.separable_1 = SeparableConv2d(in_channels, out_channels, 3, 1,
                        1, bias=bias)
        self.bn_1 = nn.BatchNorm2d(out_channels, eps=0.001, momentum=0.1,
                        affine=True)

    def forward(self, x):
        x = self.separable_0(x)
        x = self.bn_0(x)
        x = F.relu(x)
        x = self.separable_1(x)
        x = self.bn_1(x)
        x = F.relu(x)
        return x


# In[ ]:

class NormalCell(nn.Module):
    def apply_conv_op(self,op,in_channels,out_channels):
        if 'twoconv' in op:
            return TwoConv(in_channels,out_channels),out_channels
        if 'conv3' in op:
            return Conv(in_channels, out_channels,3),out_channels
        if 'conv5' in op:
            return Conv(in_channels, out_channels,5),out_channels
        if 'conv7' in op:
            return Conv(in_channels, out_channels,7),out_channels
        if 'sep' in op:
            return TwoSeparables(in_channels, out_channels, 3,
            1,bias=False),out_channels
        if 'max' in op:
            return nn.Sequential(nn.MaxPool2d(3, stride=1, padding=1),
            nn.ReLU()),in_channels
        if 'avg' in op:
            return nn.Sequential(nn.AvgPool2d(3, stride=1, padding=1),
            nn.ReLU()),in_channels
        if 'identity' in op:
            return self.Identity(),in_channels


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

```
    def identity_func(self,x):
        return x

    def cuda(self):
        return self.identity_func

def __init__(self, hidden_left, hidden_right,op_left, op_right,
            in_channels_left=1, out_channels_left=64,
            in_channels_right=1, out_channels_right=64,
            ):
    super(NormalCell, self).__init__()

    self.hleft = hidden_left
    self.hright = hidden_right
    self.op_left = op_left
    self.op_right = op_right
    self.in_channels_left = in_channels_left
    self.in_channels_right = in_channels_right
    self.out_channels_right = out_channels_right
    self.out_channels_left = out_channels_left

def forward(self, x,x_prev):
    hidden_out = []
    hidden_out.append(x)
    hidden_out.append(x_prev)

    x_comb_iter_0_left = x
    (_, CL, HL, WL) = x_comb_iter_0_left.size()
    input_size = HL
    (x_op_left,filter_left) = self.apply_conv_op(self.op_left[0],
                    CL,self.out_channels_left)
    x_output_left = x_op_left.cuda()(x_comb_iter_0_left)
    x_comb_iter_0_right = x_prev
    (_, CR, HR, WR) = x_comb_iter_0_right.size()
    (x_op_right,filter_right) = self.apply_conv_op(self.op_right[0],
                    CR,self.out_channels_right)
    x_output_right = x_op_right.cuda()(x_comb_iter_0_right)

    (_, CL, HL, WL) = x_output_left.size()
    (_, CR, HR, WR) = x_output_right.size()

    pad = abs(HL - HR)
    if HL > HR:
        x_output_right = nn.ZeroPad2d((pad,0,pad,0))(x_output_right)
    else:
        x_output_left = nn.ZeroPad2d((pad,0,pad,0))(x_output_left)

    #print('x_output_left0',x_output_left.size())
    #print('x_output_right0',x_output_right.size())
    x_comb_iter_0 = torch.cat([x_output_left,x_output_right],1)
    (_,_,S,_) = x_comb_iter_0.size()
    pad = abs(input_size - S)
    x_comb_iter_0 = nn.ZeroPad2d((pad,0,pad,0))(x_comb_iter_0)
    #print('x_comb_iter_0',x_comb_iter_0.size())
    hidden_out.append(x_comb_iter_0)


    x_comb_iter_1_left = x
    (_, CL, HL, WL) = x_comb_iter_1_left.size()
    (x_op_left,filter_left) = self.apply_conv_op(self.op_left[1],
                    CL,self.out_channels_left)
    x_output_left = x_op_left.cuda()(x_comb_iter_1_left)
    x_comb_iter_1_right = x_prev
    (_, CR, HR, WR) = x_comb_iter_1_right.size()
```

```
        (x_op_right, filter_right) = self.apply_conv_op(self.op_right[1],
                        CR, self.out_channels_right)
        x_output_right = x_op_right.cuda()(x_comb_iter_1_right)

        (_, CL, HL, WL) = x_output_left.size()
        (_, CR, HR, WR) = x_output_right.size()

        pad = abs(HL - HR)
        if HL > HR:
            x_output_right = nn.ZeroPad2d((pad,0,pad,0))(x_output_right)
        else:
            x_output_left = nn.ZeroPad2d((pad,0,pad,0))(x_output_left)

        #print('x_output_left1',x_output_left.size())
        #print('x_output_right1',x_output_right.size())
        x_comb_iter_1 = torch.cat([x_output_left,x_output_right],1)
        (_,_,S,_) = x_comb_iter_1.size()
        pad = abs(input_size - S)
        x_comb_iter_1 = nn.ZeroPad2d((pad,0,pad,0))(x_comb_iter_1)
        #x_comb_iter_1 = x_output_left + x_output_right
        #print('x_comb_iter_1',x_comb_iter_1.size())
        hidden_out.append(x_comb_iter_1)


        x_comb_iter_2_left = hidden_out[self.hleft[2]]
        (_, CL, HL, WL) = x_comb_iter_2_left.size()
        (x_op_left, filter_left) = self.apply_conv_op(self.op_left[2],
                        CL, self.out_channels_left)
        x_output_left = x_op_left.cuda()(x_comb_iter_2_left)
        x_comb_iter_2_right = hidden_out[self.hright[2]]
        (_, CR, HR, WR) = x_comb_iter_2_right.size()
        (x_op_right, filter_right) = self.apply_conv_op(self.op_right[2],
                        CR, self.out_channels_right)
        x_output_right = x_op_right.cuda()(x_comb_iter_2_right)

        (_, CL, HL, WL) = x_output_left.size()
        (_, CR, HR, WR) = x_output_right.size()

        pad = abs(HL - HR)
        if HL > HR:
            x_output_right = nn.ZeroPad2d((pad,0,pad,0))(x_output_right)
        else:
            x_output_left = nn.ZeroPad2d((pad,0,pad,0))(x_output_left)

        #print('x_output_left2',x_output_left.size())
        #print('x_output_right2',x_output_right.size())
        x_comb_iter_2 = torch.cat([x_output_left,x_output_right],1)
        #x_comb_iter_2 = x_output_left + x_output_right
        (_,_,S,_) = x_comb_iter_2.size()
        pad = abs(input_size - S)
        x_comb_iter_2 = nn.ZeroPad2d((pad,0,pad,0))(x_comb_iter_2)
        #print('x_comb_iter_2',x_comb_iter_2.size())
        hidden_out.append(x_comb_iter_2)


        x_comb_iter_3_left = hidden_out[self.hleft[3]]
        (_, CL, HL, WL) = x_comb_iter_3_left.size()
        (x_op_left, filter_left) = self.apply_conv_op(self.op_left[3],
                        CL, self.out_channels_left)
        x_output_left = x_op_left.cuda()(x_comb_iter_3_left)
        x_comb_iter_3_right = hidden_out[self.hright[3]]
        (_, CR, HR, WR) = x_comb_iter_3_right.size()
        (x_op_right, filter_right) = self.apply_conv_op(self.op_right[3],
                        CR, self.out_channels_right)
        x_output_right = x_op_right.cuda()(x_comb_iter_3_right)
```

```
        (_, CL, HL, WL) = x_output_left.size()
        (_, CR, HR, WR) = x_output_right.size()

        pad = abs(HL - HR)
        if HL > HR:
            x_output_right = nn.ZeroPad2d((pad,0,pad,0))(x_output_right)
        else:
            x_output_left = nn.ZeroPad2d((pad,0,pad,0))(x_output_left)

        #print('x_output_left3',x_output_left.size())
        #print('x_output_right3',x_output_right.size())
        x_comb_iter_3 = torch.cat([x_output_left,x_output_right],1)
        (_,_,S,_) = x_comb_iter_3.size()
        pad = abs(input_size - S)
        x_comb_iter_3 = nn.ZeroPad2d((pad,0,pad,0))(x_comb_iter_3)
        #x_comb_iter_3 = x_output_left + x_output_right
        #print('x_comb_iter_3',x_comb_iter_3.size())

        x_comb_iter_4_left = hidden_out[self.hleft[4]]
        (_, CL, HL, WL) = x_comb_iter_4_left.size()
        (x_op_left,filter_left) = self.apply_conv_op(self.op_left[4],
                        CL,self.out_channels_left)
        x_output_left = x_op_left.cuda()(x_comb_iter_4_left)
        x_comb_iter_4_right = hidden_out[self.hright[4]]
        (_, CR, HR, WR) = x_comb_iter_4_right.size()
        (x_op_right,filter_right) = self.apply_conv_op(self.op_right[4],
                        CR,self.out_channels_right)
        x_output_right = x_op_right.cuda()(x_comb_iter_4_right)

        (_, CL, HL, WL) = x_output_left.size()
        (_, CR, HR, WR) = x_output_right.size()

        pad = abs(HL - HR)
        if HL > HR:
            x_output_right = nn.ZeroPad2d((pad,0,pad,0))(x_output_right)
        else:
            x_output_left = nn.ZeroPad2d((pad,0,pad,0))(x_output_left)

        #print('x_output_left4',x_output_left.size())
        #print('x_output_right4',x_output_right.size())
        x_comb_iter_4 = torch.cat([x_output_left,x_output_right],1)
        (_,_,S,_) = x_comb_iter_4.size()
        pad = abs(input_size - S)
        x_comb_iter_4 = nn.ZeroPad2d((pad,0,pad,0))(x_comb_iter_4)
        #x_comb_iter_4 = x_output_left + x_output_right
        #print('x_comb_iter_4',x_comb_iter_4.size())

        x_out = torch.cat([x_comb_iter_0, x_comb_iter_1, x_comb_iter_2,
                        x_comb_iter_3, x_comb_iter_4], 1)
        #print('X OUT',x_out.size())
        return x_out


# In[ ]:

class NasNet(nn.Module):

    def __init__(self, cell_list, num_classes=10):
        super(NasNet, self).__init__()
        self.num_classes = num_classes
        self.normal_hidden_left = [cell_list[0][0], cell_list[1][0],
                        cell_list[2][0],cell_list[3][0], cell_list[4][0]]
        self.normal_hidden_right = [cell_list[0][1], cell_list[1][1],
                        cell_list[2][1],cell_list[3][1], cell_list[4][1]]
```

```
        self.normal_op_left = [cell_list[0][2], cell_list[1][2],
                               cell_list[2][2], cell_list[3][2], cell_list[4][2]]
        self.normal_op_right = [cell_list[0][3], cell_list[1][3],
                               cell_list[2][3], cell_list[3][3], cell_list[4][3]]
        self.cell_0 = NormalCell(self.normal_hidden_left,
                               self.normal_hidden_right,self.normal_op_left,
                                 self.normal_op_right,in_channels_left=1,
                                 in_channels_right=1,
                              out_channels_left=64,out_channels_right=64)

        self.cell_1 = NormalCell(self.normal_hidden_left,
                               self.normal_hidden_right,self.normal_op_left,
                                 self.normal_op_right,in_channels_left=64,
                              in_channels_right=64,
                              out_channels_left=128,out_channels_right=128)

        #self.cell_2 = NormalCell(self.normal_op[0],self.normal_op[1],
                in_channels=128,out_channels=256)
        #self.cell_3 = NormalCell(self.normal_op[0],self.normal_op[1],
                in_channels=256,out_channels=512)
        #self.cell_4 = NormalCell(self.normal_op[0],self.normal_op[1],
                in_channels=512,out_channels=512)
        self.register_parameter('weight', None)
        self.fc = nn.Linear(1024,self.num_classes)
        self.reduction_cell = nn.MaxPool2d(2,stride=None,padding=0)
        #self.dropout_0 = nn.Dropout2d()


    def reset_parameters(self, input,size):
        self.weight = nn.Parameter(input.data.new(1024,size))

    def features(self, x):
        img = x
        x_cell_0 = self.cell_0(img,img)
        (_, C, H, W) = x_cell_0.data.size()
        if H <= 1 or W <= 1:
            return x_cell_0
        x_red_0 = self.reduction_cell(x_cell_0)
        x_red_0 = F.relu(x_red_0)
        #x = F.relu(F.max_pool2d(x, 2))
        x_cell_1 = self.cell_1(x_red_0,x_red_0)
        (_, C, H, W) = x_cell_1.data.size()
        if H <= 1 or W <= 1:
            return x
        x_red_1 = self.reduction_cell(x_cell_1)
        x_red_1 = F.relu(x_red_1)
        #x = self.cell_2(x)
        #(_, C, H, W) = x.data.size()
        #if H <= 1 or W <= 1:
        #    return x
        #x = self.reduction_cell(x)
        #x = F.relu(x)
        #x = self.cell_3(x)
        #x = self.cell_4(x)
        #(_, C, H, W) = x.data.size()
        #if H <= 1 or W <= 1:
        #    return x
        #x = self.reduction_cell(x)
        #x = F.relu(x)
        return x_red_1

    def classifier(self, x):
        (_, C, H, W) = x.data.size()
        size = C*H*W
        x = x.view(-1 ,size)
```

```
        if self.weight is None:
            self.reset_parameters(x,size)
        x = F.linear(x,self.weight)
        x = F.relu(x)
        x = F.dropout(x, training=self.training)
        x = self.fc(x)
        return x

    def forward(self, x):
        x = self.features(x)
        x = self.classifier(x)
        return F.log_softmax(x)

#Train Parameters
log_interval = 15
epochs = 10


# In[ ]:


def train(epoch,model):
    correct = 0
    model.train()
    for batch_idx, (data, target) in enumerate(train_loader):
        data, target = data.cuda(), target.cuda()
        data, target = Variable(data), Variable(target)
        optimizer = optim.Adam(model.parameters(), lr=1e-4)
        #optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)
        optimizer.zero_grad()
        output = model(data)
        loss = F.nll_loss(output, target)
        #loss = nn.CrossEntropyLoss()(output, target)
        loss.backward(retain_graph=True)
        optimizer.step()
        #print (list(model.parameters())[0].grad)
        prediction = output.data.max(1)[1]
        if batch_idx % log_interval == 0:
            correct += prediction.eq(target.data.view_as(prediction)).cpu().sum()
            accuracy = 100. *correct /batch_size
            log_string = 'Train␣Epoch:␣{}␣[{}/{}␣({:.0f}%)]'
            '\tLoss:␣{:.6f}\tAccuracy:␣{:.3f}'.format(
                epoch, batch_idx * len(data), len(train_sampler),
                100. * batch_idx / len(train_sampler), loss.data[0],accuracy)
            f.write(log_string)
            f.write('\n')
            f.flush()
            correct = 0
        del loss, output

def evaluate_model(model):
    model.eval()
    validation_loss = 0
    correct = 0
    accuracy = 0
    for data, target in valid_loader:
        data, target = data.cuda(), target.cuda()
        data, target = Variable(data, volatile=True), Variable(target)
        output = model(data)
        loss = F.nll_loss(output, target, size_average=False).data[0]
        #Sum batch loss
        validation_loss += loss
        # get the index of the max log-probability
        pred = output.data.max(1, keepdim=True)[1]
        correct += pred.eq(target.data.view_as(pred)).cpu().sum()
```

```
    validation_loss /= len(valid_loader.dataset)
    log_string = '\nValidation set: Average loss:'
    '{:.4f}, Accuracy: {}/{} ({:.0f}%)\n'.format(
    validation_loss, correct, len(valid_sampler),
    100. * correct / len(valid_sampler))
    accuracy = 100. * correct / len(valid_sampler)
    f.write(log_string)
    f.write('\n')
    f.flush()
    return accuracy

def test_model(model):
    model.eval()
    test_loss = 0
    correct = 0
    accuracy = 0
    for data, target in test_loader:
        data, target = data.cuda(), target.cuda()
        data, target = Variable(data, volatile=True), Variable(target)
        output = model(data)
        loss = F.nll_loss(output, target, size_average=False).data[0]
        #Sum batch loss
        test_loss += loss
        # get the index of the max log-probability
        pred = output.data.max(1, keepdim=True)[1]
        correct += pred.eq(target.data.view_as(pred)).cpu().sum()

    size = len(test_loader.dataset)
    test_loss /= size
    log_string = '\nTest set: Average loss: {:.4f},'
    'Accuracy: {}/{} ({:.0f}%)\n'.format(
    test_loss, correct, size,
    100. * correct / size)
    accuracy = 100. * correct / size
    f.write(log_string)
    f.write('\n')
    f.flush()



def get_param_size(model):
    params = 0
    for p in model.parameters():
        tmp = 1
        for x in p.size():
            tmp *= x
            params += tmp
    return params


# In[ ]:
def train_model(epochs,model):
    val_accuracy= []
    #for p in model.parameters():
    #    print(p.size())
    for epoch in range(1,epochs+1):
        train(epoch,model)
        accuracy = evaluate_model(model)
        val_accuracy.append(accuracy)
        test_model(model)
    accuracy = str(max(val_accuracy))
    f.write('Accuracy: '+ accuracy + '\n')
    f.flush()
    return accuracy,get_param_size(model)
```

```
def get_accuracy(cell_list):
        OPERATIONS = ['twoconv', 'conv3', 'conv5', 'conv7',
        'conv3', 'conv5', 'conv7', 'conv3']
        val_accuracy = []
        f.flush()
        op_cell_list = []
        for i in range(len(cell_list)):
            item = cell_list[i]
            new_item = []
            new_item.append(int(item[0]))
            new_item.append(int(item[1]))
            new_item.append(OPERATIONS[int(item[2])])
            new_item.append(OPERATIONS[int(item[3])])
            op_cell_list.append(new_item)

        print(op_cell_list)
        f.write(str(op_cell_list))
        model = NasNet(op_cell_list)
        model.cuda()
        return train_model(epochs,model)
```