ECE 239AS Project Milestone Report

Hengda Shi **Gaohong Liu** **Jintao Jiang**
Department of Computer Science
University of California, Los Angeles
Los Angeles, CA 90095

Introduction

Our project is to re-implement the Progressive Neural Networks [2] proposed in 2016 by DeepMind. This neural network architecture takes advantage of transfer learning and can also avoid catastrophic forgetting in traditional machine learning tasks. Although in this paper they mainly focused on using this neural network as a function approximator in Reinforcement Learning, it can be applied to Machine Learning schemes as well. Compared to human-level intelligence, neural networks often lack the ability to transfer knowledge from one task to another. Most of the neural networks would act like a black box that takes input and generates desired output values. However, there was no effective method to save the experience learned from one network and propagate it to another network.

One of the choice in transfer learning is Finetuning, which means that a network would be pretrained on one domain, and then we could replace the last fully connected layer with the requirement of our new data, take the trained weights as the initial values and retrain the network with the real data. This classic transfer learning technique can effectively help the neural network obtain a more generative picture of the data distribution. However, if there are multiple tasks and we could like to transfer the knowledge over all tasks, the order in which we learned the task would be hard to determine, and the previously learned information would get destructed as the network gets retrained and retrained again.

Progressive Neural Networks solves the catastrophic forgetting problem by porting the output of each layer into the next neural network in an inductive manner. Those imported outputs from the previous layer would be passed through an adapter layer and eventually merged with the current existing layer output. Each column would be trained inductively meaning that the second column would only be trained after the first column has done training. The weights from all previous columns would be frozen after training so that the features extracted from using the previous neural network can ,therefore, be preserved in the next neural network. The following figure shows an example network from the paper:

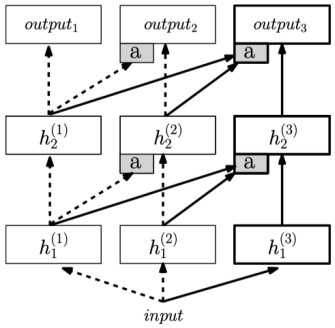

Figure 1: Three column Progressive Neural Network

In figure 1, the grey box represents the adapter layer that receives outputs from the previous layers. Notice that for layer $h_2^{(3)}$, it receives outputs from both layer $h_1^{(1)}$ and $h_1^{(2)}$. It means that each layer

would receive outputs from all previous layers. This relationship is captured by the following formula:

$$h_i^{(k)} = f\left(W_i^{(k)} h_{i-1}^{(k)} + \sum_{j<k} U_i^{(k:j)} h_{i-1}^{(j)}\right), \tag{1}$$

Preliminary Results

We have currently implemented a Progressive Neural Network in PyTorch. Our final goal is to build an A3C [1] with PNN to train on Atari games (or Cartpole if Atari takes too long to train). We will explain the design of the PNN in detail below and show code snippets to match with the output equation of the Progressive Neural Network.

In practice, the adapter layer is more than just a function U. The author uses a single hidden layer MLP to act as the adapter layer and adds non-linearity to the inputs. The inputs are also multiplied by a learned scalar α before feeding into MLP to assign different weights of different inputs. The output equation with adjusted adapter layer would be (biases are omitted for clarity):

$$h_i^{(k)} = \sigma\left(W_i^{(k)} h_{i-1}^{(k)} + U_i^{(k:j)} \sigma(V_i^{(k:j)} \alpha_{i-1}^{(<k)} h_{i-1}^{(<k)})\right), \tag{2}$$

Single Layer of Progressive Neural Network

Going back to the actual implementation, we would construct the basic layer class first in PNN. The construction part of the code is as follows:

```
class PNNLinear(nn.Module):
    def __init__(self, lid, cid, input_dim, output_dim):
        super(PNNLinear, self).__init__()
        self.lid = lid
        self.cid = cid
        self.input_dim = input_dim
        self.output_dim = output_dim

        # basic neural net
        self.w = nn.Linear(self.input_dim, self.output_dim)
        # lateral connection
        self.u = nn.ModuleList()
        # adapter
        self.v = nn.ModuleList()
        # alpha
        self.alpha = []

        # need lateral connection only if
        # 1. it is not the first column
        # 2. it is not the first layer
        if self.cid and self.lid:
            self.u.extend([
                nn.Linear(self.input_dim, self.output_dim)
                for _ in range(self.cid)
            ])
            self.v.extend([
                nn.Linear(self.input_dim, self.input_dim)
                for _ in range(self.cid)
            ])
            self.alpha.extend([
                nn.Parameter(torch.rand(1, dtype=float))
                for _ in range(self.cid)
            ])
```

For the purpose of simplicity, we only construct a linear layer block for PNN even though games like Pong are 2D. Instead of capturing the features from pixels using convolutional layers, we would be using the "ram" version of the Atari game environments provided by OpenAI Gym which has already reduced the dimensionality to 1D by extracting meaningful information from the game.

For each PNN linear layer, `self.w` represents the default fully connected perceptron. `self.u`, `self.v` and `self.alpha` are only needed when the layer is not on the first column and not the first layer. The number of required fully connected networks depends on the number of previous columns. To enable gradient calculation on α, we wrap the torch tensor with `nn.Parameter`. Paper only mentioned that the value of α is initialized to be a small value. We used the random number generator in PyTorch to sample a number in a uniform distribution in $[0, 1)$.

The actual forwarding logic is declared in the `forward` function:

```
    def forward(self, X):
        X = [X] if not isinstance(X, list) else X
        # first part of the equation
        # current column output
        # use the latest input
        cur_out = self.w(X[-1])
        # second part of the equation
        # lateral connections
        # use old inputs from previous columns
        prev_out = sum([
            u(F.relu(v(alpha * x)))
            for u, v, alpha, x in zip(self.u, self.v, self.alpha, X)
        ])
        return F.relu(cur_out + prev_out)
```

The input `X` consists of inputs from the first column to the current column in order. Invoking `self.w(X[-1])` would be corresponding to the $W_i^{(k)} h_{i-1}^{(k)}$ part of the equation. For each previous column outputs, it would be multiplied by the learned scalar α, passed into the adapter layer V, filtered with ReLU activation function, then passed into the function lateral connection layer U. The results of all previous columns would be summed up corresponding to $U_i^{(k:j)} \sigma(V_i^{(k:j)} \alpha_{i-1}^{(<k)} h_{i-1}^{(<k)}))$ part of the equation. Notice that $h_{i-1}^{(<k)}$ is further defined as a vector of all previous column outputs. Therefore, the dot product would eventually be a summation over all results of calculations on previous columns and can match up with equation (1). The final output of one layer would be the summation of `cur_out` and `prev_out` passed through the ReLU activation function.

Progressive Neural Network

We constructed the class of progressive neural network with no columns initially, and the number of layers on each column is fixed so that all layers would be able to receive results from previous columns' layers. The forwarding scheme is defined as follows:

```
    def forward(self, X, cid=-1):
        # first layer pass
        h = [column[0](X[i]) for i, column in enumerate(self.columns)]
        # rest layers pass
        for k in range(1, self.nlayers):
            h = [column[k](h[:i + 1]) for i, column in
            ↪  enumerate(self.columns)]

        # return latest output unless specified
        return h[cid]
```

The first layer has no dependency on previous columns, Each column would only need input for that specific column. However, for the rest of the layers, they would need all inputs for previous columns,

which is the reason we use `h[:i+1]` to slice the list. Unless specified, the return value would be the output on the last column.

One problem with the current setting is that the problem of catastrophic forgetting still exists because the weights would still be updated on the first column if we retrain the network with added columns. Luckily, PyTorch provides an easy way to turn off the gradient calculation on weights so that they will no longer get updated. We defined a `freeze` function to freeze all previous columns after training:

```
    # freeze previous columns
    def freeze(self):
        for column in self.columns:
            for params in column.parameters():
                params.requires_grad = False
```

Conclusion

In this milestone report, we explained the rationale behind the benefits of Progressive Neural Network: its ability to solve the catastrophic forgetting problem in common transfer learning techniques. Freezing weights on previous columns after training essentially preserves the previous experience to apply to the next neural network. In the final report, we would be demonstrating how to combine this function approximator with A3C [1], and the evaluation results on different Atari games. The full code of PNN is included in the Appendix section.

References

[1] MNIH, V., BADIA, A. P., MIRZA, M., GRAVES, A., LILLICRAP, T. P., HARLEY, T., SILVER, D., AND KAVUKCUOGLU, K. Asynchronous methods for deep reinforcement learning. *CoRR abs/1602.01783* (2016).

[2] RUSU, A. A., RABINOWITZ, N. C., DESJARDINS, G., SOYER, H., KIRKPATRICK, J., KAVUKCUOGLU, K., PASCANU, R., AND HADSELL, R. Progressive neural networks. *CoRR abs/1606.04671* (2016).

Supplementary Material

Implementation of Progressive Neural Network

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import torch
import torch.nn as nn
import torch.nn.functional as F


# basic construction block for PNN
class PNNLinear(nn.Module):
    def __init__(self, lid, cid, input_dim, output_dim):
        super(PNNLinear, self).__init__()
        self.lid = lid
        self.cid = cid
        self.input_dim = input_dim
        self.output_dim = output_dim

        # h^(k)_i = \sigma(W^(k)_i h^(k)_{i-1} + U^(k:j)_{i}
        #           \sigma(V^(k:j)_i \alpha^(<k)_{i-1} h^(<k)_{i-1}))

```

```
        # basic neural net
        self.w = nn.Linear(self.input_dim, self.output_dim)
        # lateral connection
        self.u = nn.ModuleList()
        # adapter
        self.v = nn.ModuleList()
        # alpha
        self.alpha = []

        # need lateral connection only if
        # 1. it is not the first column
        # 2. it is not the first layer
        if self.cid and self.lid:
            self.u.extend([
                nn.Linear(self.input_dim, self.output_dim)
                for _ in range(self.cid)
            ])
            self.v.extend([
                nn.Linear(self.input_dim, self.input_dim)
                for _ in range(self.cid)
            ])
            self.alpha.extend([
                nn.Parameter(torch.randn(1, dtype=float))
                for _ in range(self.cid)
            ])

    def forward(self, X):
        X = [X] if not isinstance(X, list) else X
        # first part of the equation
        # current column output
        # use the latest input
        cur_out = self.w(X[-1])
        # second part of the equation
        # lateral connections
        # use old inputs from previous columns
        prev_out = sum([
            u(F.relu(v(alpha * x)))
            for u, v, alpha, x in zip(self.u, self.v, self.alpha, X)
        ])
        return F.relu(cur_out + prev_out)


class PNN(nn.Module):
    # nlayers is the number of layers in one column
    def __init__(self, nlayers):
        super(PNN, self).__init__()
        self.nlayers = nlayers
        self.columns = nn.ModuleList()

    def forward(self, X, cid=-1):
        # first layer pass
        h = [column[0](X[i]) for i, column in enumerate(self.columns)]
        # rest layers pass
        for k in range(1, self.nlayers):
            h = [column[k](h[:i + 1]) for i, column in
            ↪  enumerate(self.columns)]

        # return latest output unless specified
        return h[cid]
```

```
79
80      # sizes contains a list of layers' output size
81      # add a column to the neural net
82      def add(self, sizes):
83          modules = [
84              PNNLinear(lid, len(self.columns), sizes[lid], sizes[lid + 1])
85              for lid in range(self.nlayers)
86          ]
87          self.columns.append(nn.ModuleList(modules))
88
89      # freeze previous columns
90      def freeze(self):
91          for column in self.columns:
92              for params in column.parameters():
93                  params.requires_grad = False
94
95      # return parameters of the current column
96      def parameters(self, cid=None):
97          return super(PNN, self).parameters(
98          ) if cid is None else self.columns[cid].parameters()
```