Source: Python Reinforcement Learning Projects- by Yang Wenzhuo; Rajalingappaa Shanmugamani; Sean SaitoPublished by [Packt Publishing](https://learning.oreilly.com/library/publisher/packt-publishing/), 2018

**Generating a Deep Learning Image Classifier**

Some of the models that human researchers have designed and engineered have also gained popularity, including AlexNet, Inception, VGGNet, ResNet, and DenseNet; some of them are now the go-to standard for their respective tasks. However, the better the model gets, the more complex the architecture becomes, especially with the introduction of residual connections between convolutional layers. The task of designing a high-performance neural network has thus become a very arduous one. Hence the question arises: is it possible for an algorithm to learn how to generate neural network architectures?

It is possible to train a neural network to generate neural networks that perform well on a given task. In this chapter, we will examine **Neural Architecture Search** (referred to as **NAS** henceforth), a novel framework developed by Barret Zoph and Quoc V. Le from the Google Brain team that uses deep reinforcement learning to train a Controller to produce child networks that learn to accomplish tasks. We will learn how policy gradient methods (REINFORCE in particular) can train such a Controller. We will then implement a Controller that uses NAS to generate child networks that train on CIFAR-10 data.

NAS – lets see how the framework learns to generate other neural networks to complete tasks using a popular reinforcement learning scheme called **REINFORCE**, which is a type of policy gradient algorithm.

**Generating and training child networks**

Research on algorithms that generate neural architectures has been around since the 1970's. What sets NAS apart from previous works is its ability to cater to large-scale deep learning algorithms and its formulation of the task as a reinforcement learning problem. More specifically, the agent, which we will refer to as the Controller, is a recurrent neural network that generates a sequence of values. You can think of these values as a sort of genetic code of the child network that defines its architecture; it sets the sizes of each convolutional kernel, the length of each kernel, the number of filters in each layer, and so on. In more advanced frameworks, the values also determine the connections between layers to generate residual layers:

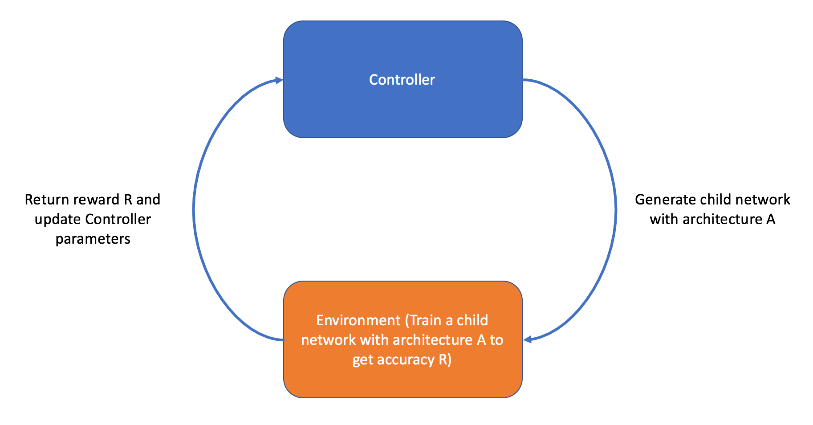
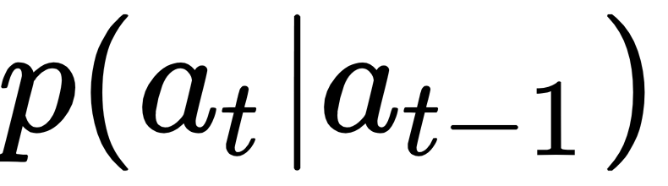
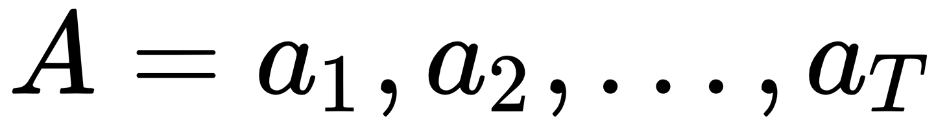


Figure 1: Overview of the NAS framework

Moreover, each value of the genetic code that the Controller outputs counts as an action, *a*, that is sampled with probability, *p*. Because the Controller is a recurrent neural network, we can represent the *t*th action as . Once we have a list of actions, —where *T* is some predefined parameter that sets the maximum size of the genetic code—we can generate the child network with the specified architecture, *A*:

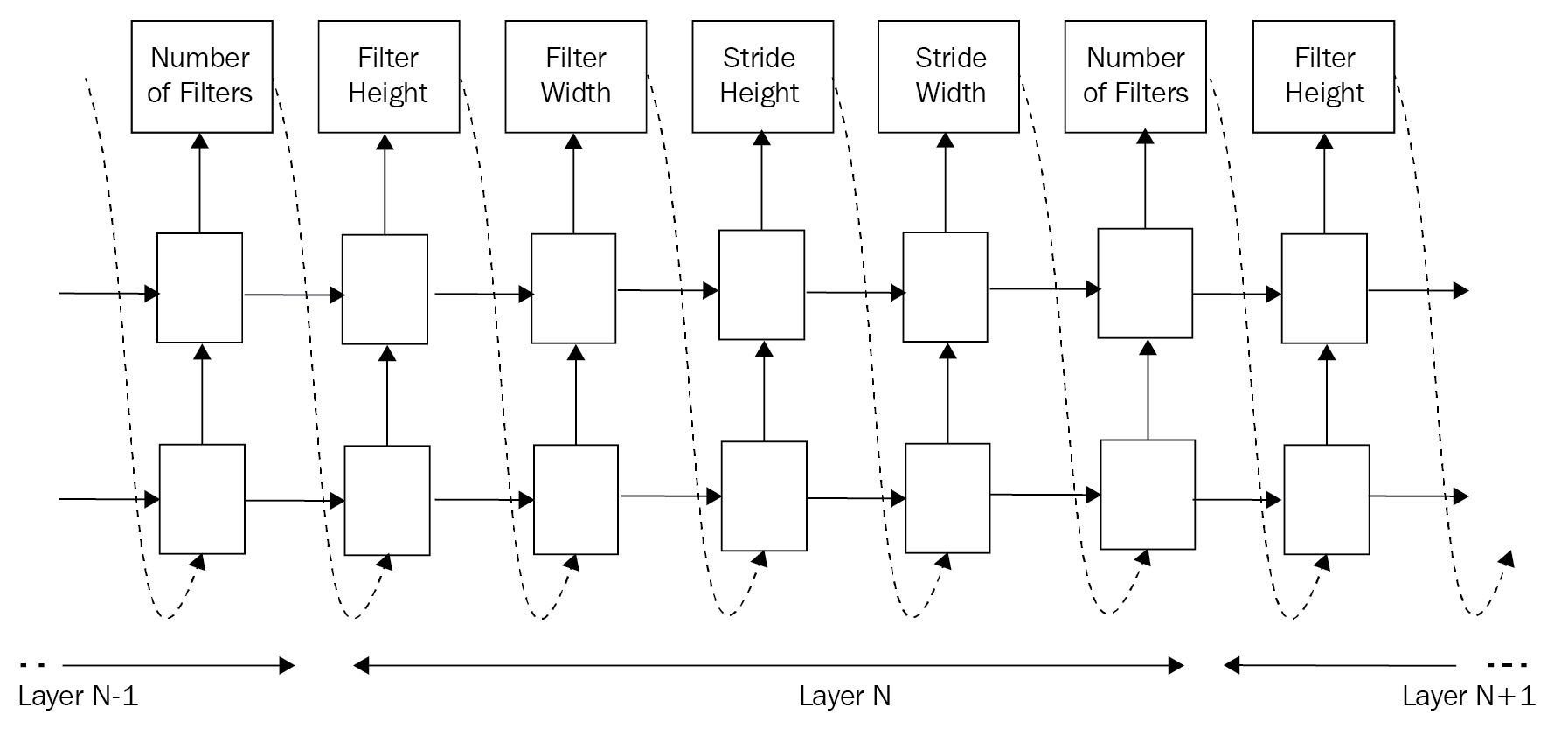
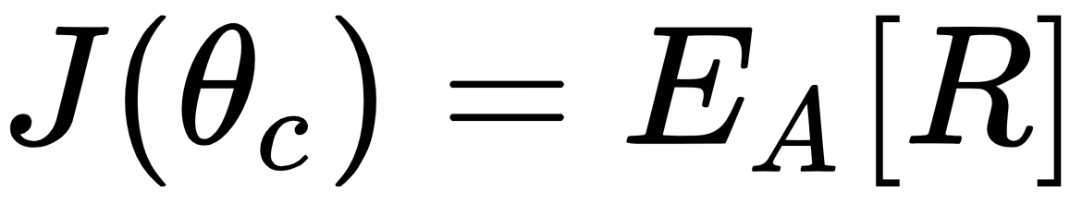
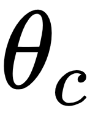
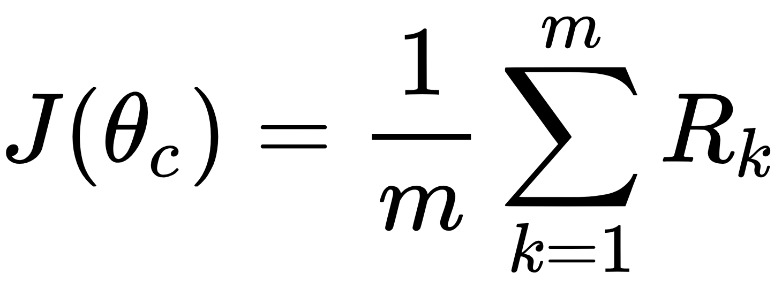


Figure 2: The architecture of the Controller

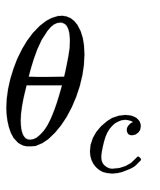
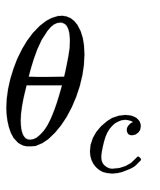
Once the Controller generates a child network, we train it on a given task until either some termination criteria is met (for example, after a specified number of epochs). We then evaluate the child network on the validation set to produce some validation accuracy, *R*. The validation accuracy acts as the reward signal for the Controller. So, the objective of the Controller is to maximize the expected reward:

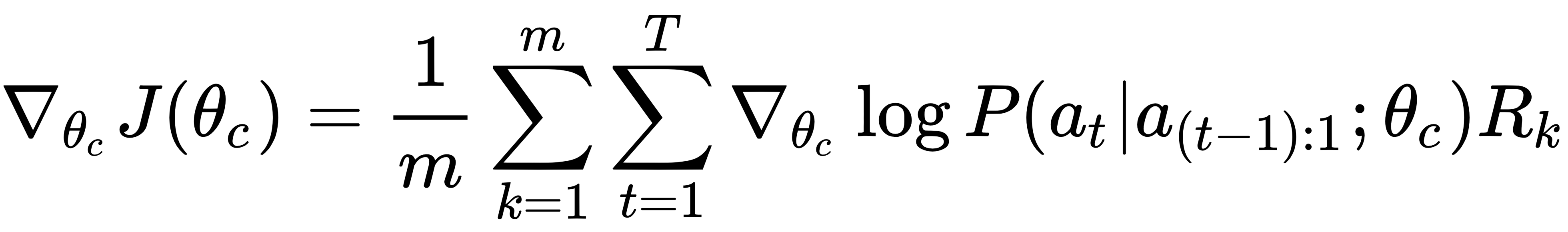


Here, *J* is the reward function (also referred to as the fit function),  is the parameters of the Controller, and the right-hand side of the equation is the expectation of the reward given a child network architecture, *A*. In practice, this expectation is calculated by averaging the rewards over *m*child network models that the Controller produces in one batch:

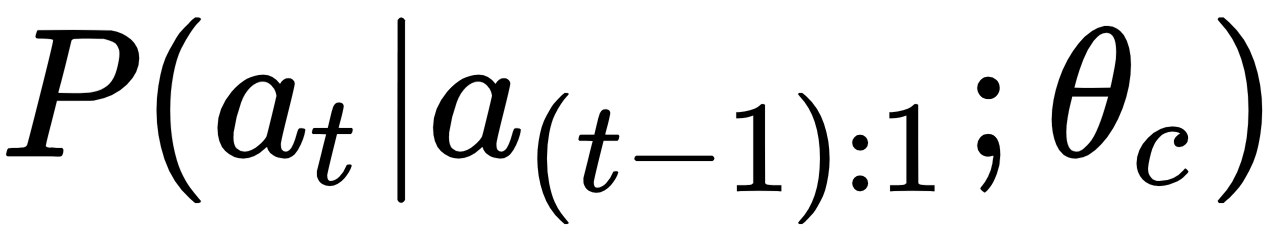
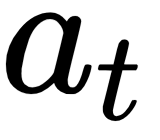
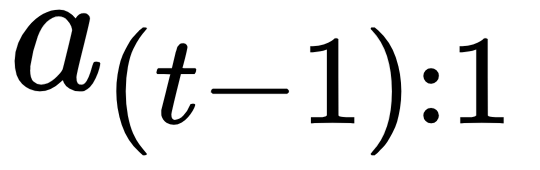
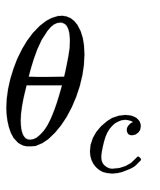
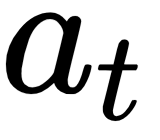
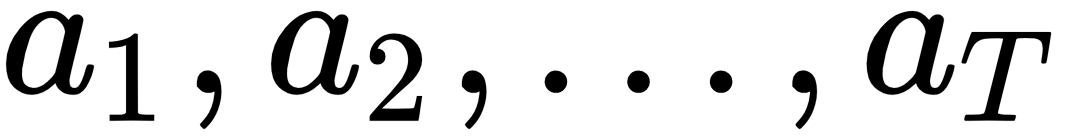


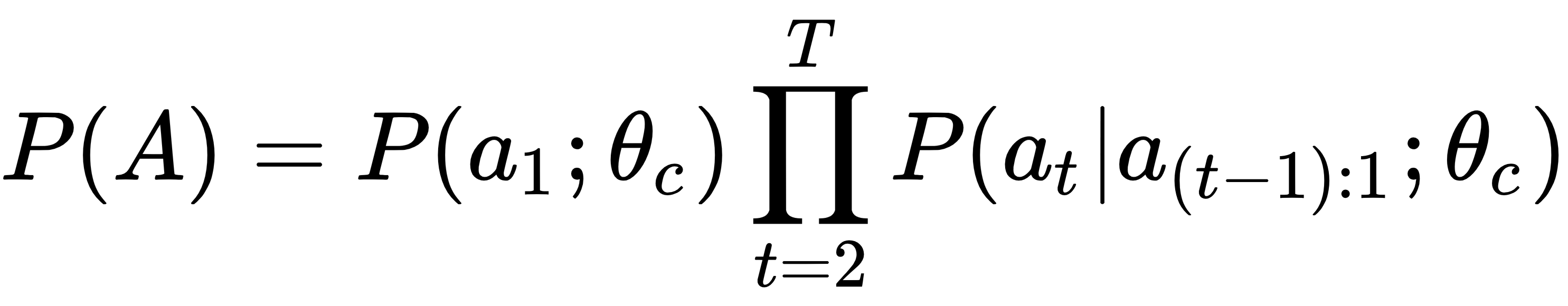
**Training the Controller**

How do we use this reward signal to update the Controller? Remember, this reward signal is not differentiable like a loss function in supervised learning; we may not backpropagate this through the Controller on its own. Instead, we employ a policy gradient method called **REINFORCE** to iteratively update the Controller parameters, . In REINFORCE, the gradient of the reward function, *J*, with respect to the parameters of the Controller, , is defined as follows:

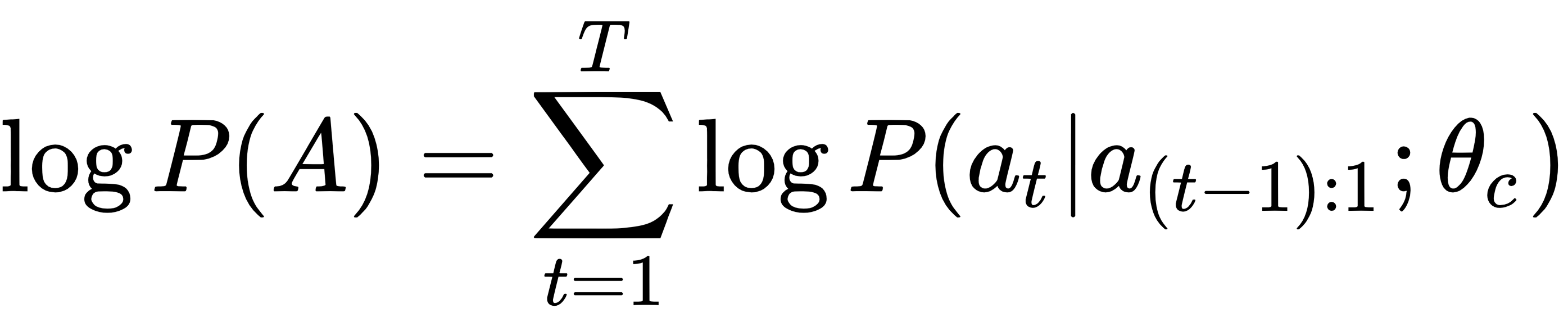


You may recall seeing a similar expression in [Chapter 6](https://learning.oreilly.com/library/view/python-reinforcement-learning/9781788991612/857fa187-2524-4682-ae75-ab6b15e00a50.xhtml), *Learning to Play Go*. Indeed, this is the policy gradient method that AlphaGo and AlphaGo Zero use to update the weights of their reinforcement learning policy networks. We briefly introduced the method then, but we will go a bit more in-depth here.

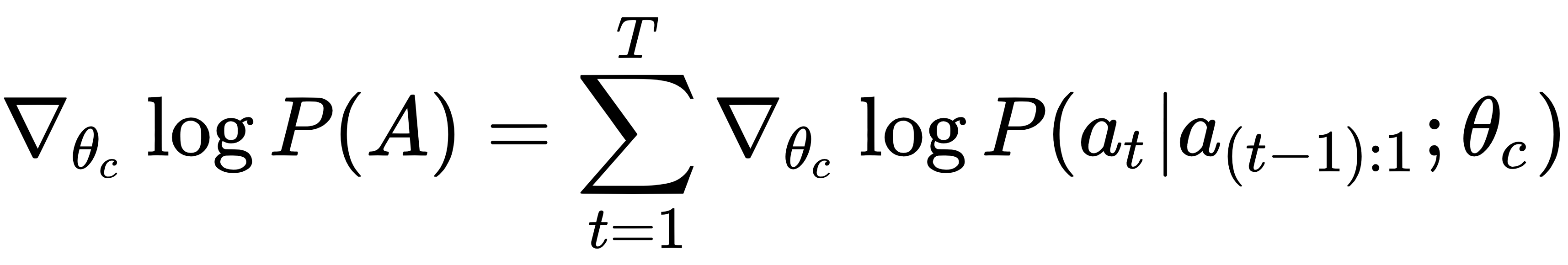
Let's break the preceding equation down. On the right-hand side, we would like to represent the probability of choosing some architecture, *A*. In particular,   
 represents the probability that the Controller takes action  given all the previous actions, , and the parameters of the Controller, . Again, action  corresponds to the *t*th value in the genetic sequence that represents the child network's architecture. The joint probability of choosing all actions, , can be formulated as follows:



By transforming this joint probability to the log space, we can turn the product into a sum of probabilities:

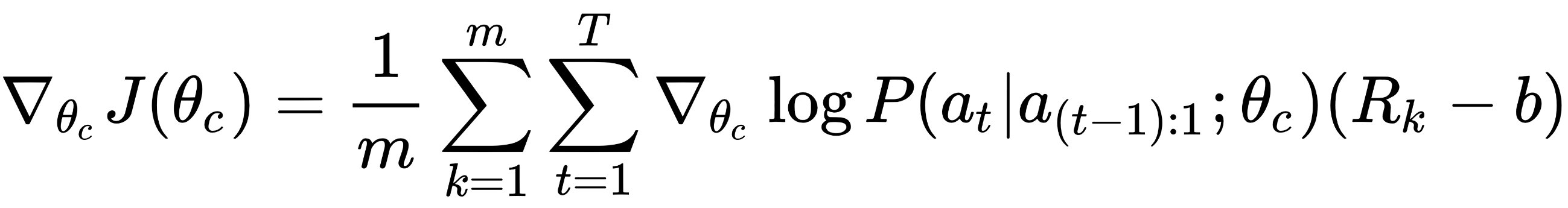


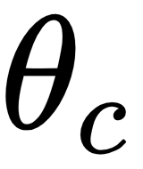
In general, we want to maximize this log conditional probability for taking some action. In other words, we want to increase the likelihood of the Controller generating a particular sequence of genetic codes. Hence we perform gradient ascent on this objective with respect to the Controller's parameters by taking the derivative of the log probability of sampling architecture *A*:



But how do we update the Controller parameters so that better architectures are generated? This is where we make use of the reward signal, *R*. By multiplying the preceding with the reward signal, we can control the size of the policy gradient. In other words, if a particular architecture achieved high validation accuracy (with the highest possible being 1.0), the gradients for that policy will be relatively strong and the Controller will learn to produce similar architectures. On the other hand, smaller validation accuracies will mean smaller gradients, which helps the Controller ignore those architectures.

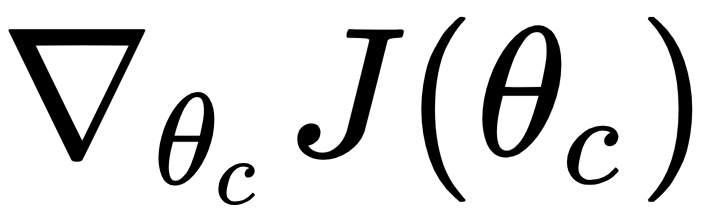
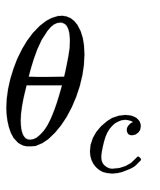
One problem with the REINFORCE algorithm is that the reward signal *R* can have high variance, which can lead to unstable training curves. To reduce the variance, it is common to subtract the reward with some value, *b*, which we refer to as the baseline function. In Zoph et al., the baseline function is defined as the exponential moving average of the past rewards. Hence our REINFORCE policy gradient is now defined as follows:

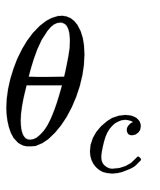


Once we have this gradient, we apply the usual backpropagation algorithm to update the Controller parameters, .

# Training algorithm

The training steps for the Controller is as follows:

* For each episode, do the following:
  1. Generate m child network architectures
  2. Train child networks on given task and obtain m validation accuracies
  3. Calculate 
  4. Update 

In Zoph et al., the training procedure is done with several copies of the Controller. Each Controller is parameterized by , which itself is stored in a distributed manner among multiple servers, which we call parameter servers.

In each episode of training, the Controller creates several child architectures and trains them independently. The policy gradient calculated as a result is then sent to the parameter servers to update the Controller's parameters:

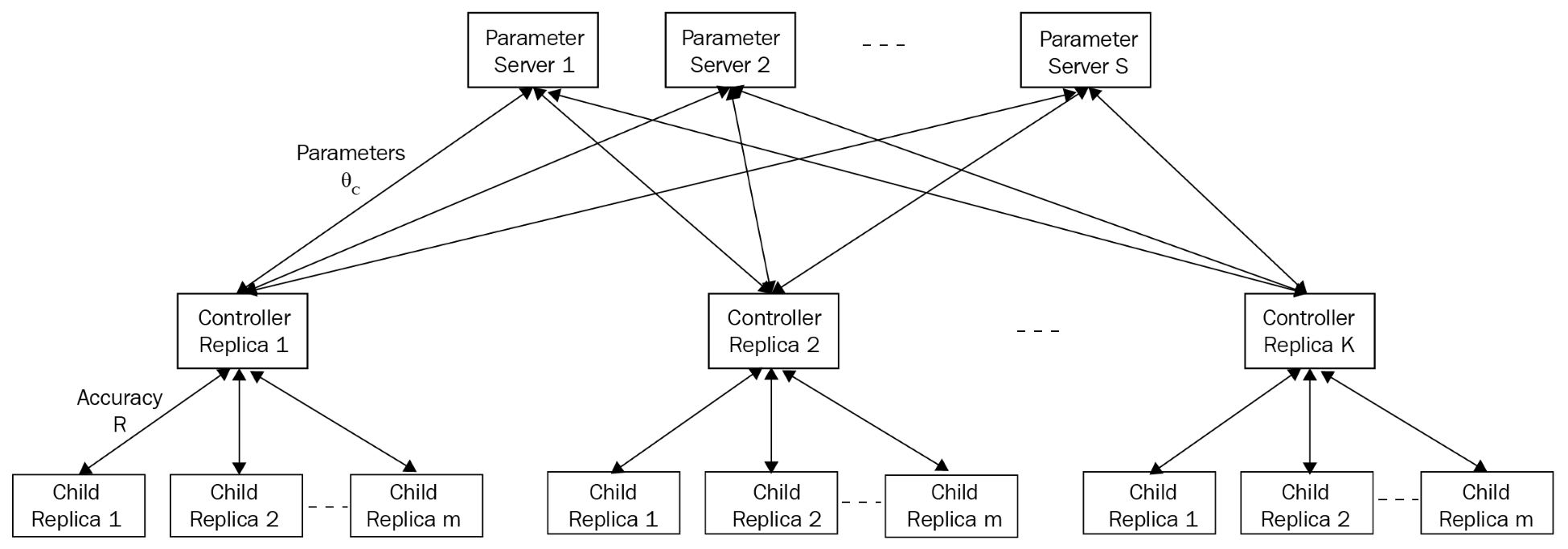


Figure 3: The training architecture

The Controller's parameters are shared among a number of parameter servers. Moreover, multiple copies of the Controller are trained in parallel, each one calculating rewards and gradients for its respective batches of child network architectures.

This architecture allows the Controller to be trained quickly given enough resources. For our purposes, however, we will stick to one Controller that generates m child network architectures. Once we have trained the Controller for a specified number of episodes, we calculate the test accuracy by choosing the child network architecture that had the best validation accuracy and measuring its performance on the test set.

# Implementing NAS

In this section, we will implement NAS. In particular, our Controller is tasked with generating child network architectures that learn to classify images from the CIFAR-10 dataset. The architecture of the child network will be represented by a list of numbers. Every four values in this list represent a convolutional layer in the child network, each describing the kernel size, stride length, number of filters, and the pooling window size in the subsequent pooling layer. Moreover, we specify the number of layers in a child network as a hyper-parameters. For example, if our child network has three layers, its architecture is represented as a vector of length 12. If we have an architecture represented as [3, 1, 12, 2, 5, 1, 24, 2], then the child network is a two-layer network where the first layer has kernel size of 3, stride length of 1, 12 filters, and a max-pooling window size of 2, and the second layer has kernel size of 5, stride length of 1, 24 filters, and max-pooling window size of 2. We set the activation function between each layer as ReLU. The final layer involves flattening the last convolutional layer output and applying a linear layer with the number of classes as its width, followed by a Softmax activation. The following sections will walk you through the implementation.

**child\_network.py**

We will first implement our child network module. This module contains a class called ChildCNN, which constructs a child network given some architecture configuration, which we call cnn\_dna. As mentioned previously, cnn\_dna is simply a list of numbers, with each value representing a parameter of its respective convolutional layer. In our config.py, we specify the max number of layers a child network can have. For our implementation, each convolutional layer is represented by four parameters, where each corresponds to the kernel size, stride length, number of filters, and subsequent max-pooling window size.

Our ChildCNN is a class that takes the following parameters in its constructor:

* cnn\_dna: The network architecture
* child\_id: A string that simply identifies the child network architecture
* beta: Weight parameter for L2 regularization
* drop\_rate: Dropout rate

import logging  
  
import tensorflow as tf  
  
logger = logging.getLogger(\_\_name\_\_)  
  
class ChildCNN(object):  
  
 def \_\_init\_\_(self, cnn\_dna, child\_id, beta=1e-4, drop\_rate=0.2, \*\*kwargs):  
 self.cnn\_dna = self.process\_raw\_controller\_output(cnn\_dna)  
 self.child\_id = child\_id  
 self.beta = beta  
 self.drop\_rate = drop\_rate  
 self.is\_training = tf.placeholder\_with\_default(True, shape=None, name="is\_training\_{}".format(self.child\_id))  
 self.num\_classes = 10

We also implement a helper function called proces\_raw\_controller\_output(), which parses cnn\_dna that the Controller outputs:

def process\_raw\_controller\_output(self, output):  
 """  
 A helper function for preprocessing the output of the NASCell  
 Args:  
 output (numpy.ndarray) The output of the NASCell  
  
 Returns:  
 (list) The child network's architecture  
 """  
 output = output.ravel()  
 cnn\_dna = [list(output[x:x+4]) for x in range(0, len(output), 4)]  
 return cnn\_dna

Finally, we include the build method, which builds our child network using the given cnn\_dna. You will notice that, although we are letting the Controller decide the architecture of our child network, we are still hardcoding several things, such as the activation function, tf.nn.relu, and the way we initialize the kernels. The fact that we are adding a max-pooling layer after each convolutional layer is also hardcoded. A more sophisticated NAS framework would also let the Controller decide these components of the architecture as well, with the trade off being longer training time:

def build(self, input\_tensor):  
 """  
 Method for creating the child neural network  
 Args:  
 input\_tensor: The tensor which represents the input  
  
 Returns:  
 The tensor which represents the output logit (pre-softmax activation)  
  
 """  
 logger.info("DNA is: {}".format(self.cnn\_dna))  
 output = input\_tensor  
 for idx in range(len(self.cnn\_dna)):  
 # Get the configuration for the layer  
 kernel\_size, stride, num\_filters, max\_pool\_size = self.cnn\_dna[idx]  
 with tf.name\_scope("child\_{}\_conv\_layer\_{}".format(self.child\_id, idx)):  
 output = tf.layers.conv2d(output,  
 # Specify the number of filters the convolutional layer will output  
 filters=num\_filters,  
 # This specifies the size (height, width) of the convolutional kernel  
 kernel\_size=(kernel\_size, kernel\_size),  
 # The size of the stride of the kernel  
 strides=(stride, stride),  
 # We add padding to the image  
 padding="SAME",  
 # It is good practice to name your layers  
 name="conv\_layer\_{}".format(idx),  
 activation=tf.nn.relu,  
 kernel\_initializer=tf.contrib.layers.xavier\_initializer(),  
 bias\_initializer=tf.zeros\_initializer(),  
 kernel\_regularizer=tf.contrib.layers.l2\_regularizer(scale=self.beta))

Each convolutional layer is followed by a max-pooling layer and a dropout layer:

# We apply 2D max pooling on the output of the conv layer  
 output = tf.layers.max\_pooling2d(  
 output, pool\_size=(max\_pool\_size, max\_pool\_size), strides=1,  
 padding="SAME", name="pool\_out\_{}".format(idx)  
 )  
 # Dropout to regularize the network further  
 output = tf.layers.dropout(output, rate=self.drop\_rate, training=self.is\_training)

Finally, after several blocks of convolutional, pooling, and dropout layers, we flatten the output volume and a fully connected layer:

# Lastly, we flatten the outputs and add a fully-connected layer  
 with tf.name\_scope("child\_{}\_fully\_connected".format(self.child\_id)):  
 output = tf.layers.flatten(output, name="flatten")  
 logits = tf.layers.dense(output, self.num\_classes)  
  
 return logits

The argument to our build method is an input tensor, which has by default a shape of (32, 32, 3), which is the CIFAR-10 data shape. The reader is free to tweak the architecture of this network, including adding a few more fully connected layers or inserting batch normalization layers in between convolutions.

# cifar10\_processor.py

This module contains code for processing CIFAR-10 data, which we use to train our child networks. In particular, we construct an input data pipeline using TensorFlow's native tf.data.Dataset API. Those who have used TensorFlow for some time may be more familiar with creating tf.placeholder tensors and feeding data via sess.run(..., feed\_dict={...}). However, this is no longer the preferred way of feeding data into the network; in fact, it is the slowest way to train a network, for the repetitive conversions from data in numpy format to a native TensorFlow format cause significant computational overhead. tf.data.Dataset alleviates this problem by turning the input pipeline into TensorFlow operations that are part of the symbolic graph. In other words, the data is converted into tensors right from the get-go. This allows for a much smoother input pipeline that can speed up training.

*Refer to this official tutorial (*[*https://www.tensorflow.org/guide/datasets\_for\_estimators*](https://www.tensorflow.org/guide/datasets_for_estimators)*) for more information on the tf.data.Dataset API.*

The cifar10\_processor.py contains a single method to create CIFAR-10 data into tensors. We first implement a helper function for creating a tf.data.Dataset object:

import logging  
  
import numpy as np  
import tensorflow as tf  
from keras.datasets import cifar10  
from keras.utils import np\_utils  
  
logger = logging.getLogger(\_\_name\_\_)  
  
def \_create\_tf\_dataset(x, y, batch\_size):  
 return tf.data.Dataset.zip((tf.data.Dataset.from\_tensor\_slices(x),  
 tf.data.Dataset.from\_tensor\_slices(y))).shuffle(500).repeat().batch(batch\_size)

In the main data processor function, we first load CIFAR-10 data. We use the keras.datasets API to do this (run pip install keras in your Terminal if you don't have Keras):

def get\_tf\_datasets\_from\_numpy(batch\_size, validation\_split=0.1):  
 """  
 Main function getting tf.Data.datasets for training, validation, and testing  
  
 Args:  
 batch\_size (int): Batch size  
 validation\_split (float): Split for partitioning training and validation sets. Between 0.0 and 1.0.  
 """  
 # Load data from keras datasets api  
 (X, y), (X\_test, y\_test) = cifar10.load\_data()  
  
 logger.info("Dividing pixels by 255")  
 X = X / 255.  
 X\_test = X\_test / 255.  
  
 X = X.astype(np.float32)  
 X\_test = X\_test.astype(np.float32)  
 y = y.astype(np.float32)  
 y\_test = y\_test.astype(np.float32)  
  
 # Turn labels into onehot encodings  
 if y.shape[1] != 10:  
 y = np\_utils.to\_categorical(y, num\_classes=10)  
 y\_test = np\_utils.to\_categorical(y\_test, num\_classes=10)  
  
 logger.info("Loaded data from keras")  
  
 split\_idx = int((1.0 - validation\_split) \* len(X))  
 X\_train, y\_train = X[:split\_idx], y[:split\_idx]  
 X\_valid, y\_valid = X[split\_idx:], y[split\_idx:]

We then turn these NumPy arrays into TensorFlow tensors, which we can feed directly to our network. What actually happens in our \_create\_tf\_dataset helper function? We use the tf.dataset.Dataset.from\_tensor\_slices() function to turn the data and the labels, both of which are NumPy arrays, into TensorFlow tensors. We then create the native dataset by zipping these tensors. The shuffle, repeat, and batch functions after zipping the data and labels define how we want the input pipeline to work. In our case, we are shuffling the input data, repeating the dataset once we reach the end, and batching the data with a given batch size. We also calculate the number of batches that each dataset has and return them:

train\_dataset = \_create\_tf\_dataset(X\_train, y\_train, batch\_size)  
valid\_dataset = \_create\_tf\_dataset(X\_valid, y\_valid, batch\_size)  
test\_dataset = \_create\_tf\_dataset(X\_test, y\_test, batch\_size)  
  
# Get the batch sizes for the train, valid, and test datasets  
num\_train\_batches = int(X\_train.shape[0] // batch\_size)  
num\_valid\_batches = int(X\_valid.shape[0] // batch\_size)  
num\_test\_batches = int(X\_test.shape[0] // batch\_size)  
  
return train\_dataset, valid\_dataset, test\_dataset, num\_train\_batches, num\_valid\_batches, num\_test\_batches

And with that, we have an optimized input data pipeline that is much faster than using feed\_dict.

# controller.py

The controller.py module is where everything comes together. We will implement the Controller, which handles training each child network as well as its own parameter updates. We first implement a helper function that calculates an exponential moving average of a list of numbers. We use this as the baseline function for our REINFORCE gradient calculation, as mentioned previously, to calculate the exponential moving average of the past rewards:

import logging  
  
import numpy as np  
import tensorflow as tf  
  
from child\_network import ChildCNN  
from cifar10\_processor import get\_tf\_datasets\_from\_numpy  
from config import child\_network\_params, controller\_params  
  
logger = logging.getLogger(\_\_name\_\_)  
  
def ema(values):  
 """  
 Helper function for keeping track of an exponential moving average of a list of values.  
 For this module, we use it to maintain an exponential moving average of rewards  
   
 Args:  
 values (list): A list of rewards   
  
 Returns:  
 (float) The last value of the exponential moving average  
 """  
 weights = np.exp(np.linspace(-1., 0., len(values)))  
 weights /= weights.sum()  
 a = np.convolve(values, weights, mode="full")[:len(values)]  
 return a[-1]

Next, we define our Controller class:

class Controller(object):  
  
 def \_\_init\_\_(self):  
 self.graph = tf.Graph()  
 self.sess = tf.Session(graph=self.graph)  
 self.num\_cell\_outputs = controller\_params['components\_per\_layer'] \* controller\_params['max\_layers']  
 self.reward\_history = []  
 self.architecture\_history = []  
 self.divison\_rate = 100  
 with self.graph.as\_default():  
 self.build\_controller()

There are several attributes to note: self.num\_cell\_outputs refers to the number of values that our **recurrent neural network** (**RNN**) should output and corresponds to the length of the child network architecture configuration. self.reward\_history and self.ar chitecture\_history are simply buffers that allow us to keep track of rewards and child network architectures that the RNN generated.

# Method for generating the Controller

We next implement a method for generating the Controller, which we call build\_controller. The first step in constructing our Controller is defining the input placeholders. We create two of these—one is for the child network DNA, which is fed as input to the RNN for generating a new child network DNA, and the second is a list for storing discounted rewards when calculating the gradients for REINFORCE:

def build\_controller(self):  
 logger.info('Building controller network')  
 # Build inputs and placeholders  
 with tf.name\_scope('controller\_inputs'):  
 # Input to the NASCell  
 self.child\_network\_architectures = tf.placeholder(tf.float32, [None, self.num\_cell\_outputs],   
 name='controller\_input')  
 # Discounted rewards  
 self.discounted\_rewards = tf.placeholder(tf.float32, (None, ), name='discounted\_rewards')

We then define the output tensors of our RNN (to be implemented here). Note that the outputs of the RNN are small, in the range of (-1, 1). So, we multiply the output by 10 in order to create the child network DNA:

# Build controller  
with tf.name\_scope('network\_generation'):  
 with tf.variable\_scope('controller'):  
 self.controller\_output = tf.identity(self.network\_generator(self.child\_network\_architectures),   
 name='policy\_scores')  
 self.cnn\_dna\_output = tf.cast(tf.scalar\_mul(self.divison\_rate, self.controller\_output), tf.int32,  
 name='controller\_prediction')

We then define the loss function and optimizer. We use RMSPropOptimizer as our backpropagation algorithm, where the learning rate decays exponentially. Rather than calling optimizer.minimize(loss) as is usually done with other neural network models, we call the compute\_gradients method to obtain gradients for calculating REINFORCE gradients:

# Set up optimizer  
self.global\_step = tf.Variable(0, trainable=False)  
self.learning\_rate = tf.train.exponential\_decay(0.99, self.global\_step, 500, 0.96, staircase=True)  
self.optimizer = tf.train.RMSPropOptimizer(learning\_rate=self.learning\_rate)  
  
# Gradient and loss computation  
with tf.name\_scope('gradient\_and\_loss'):  
 # Define policy gradient loss for the controller  
 self.policy\_gradient\_loss = tf.reduce\_mean(tf.nn.softmax\_cross\_entropy\_with\_logits(  
 logits=self.controller\_output[:, -1, :],  
 labels=self.child\_network\_architectures))  
 # L2 weight decay for Controller weights  
 self.l2\_loss = tf.reduce\_sum(tf.add\_n([tf.nn.l2\_loss(v) for v in  
 tf.trainable\_variables(scope="controller")]))  
 # Add the above two losses to define total loss  
 self.total\_loss = self.policy\_gradient\_loss + self.l2\_loss \* controller\_params["beta"]  
 # Compute the gradients  
 self.gradients = self.optimizer.compute\_gradients(self.total\_loss)  
  
 # Gradients calculated using REINFORCE  
 for i, (grad, var) in enumerate(self.gradients):  
 if grad is not None:  
 self.gradients[i] = (grad \* self.discounted\_rewards, var)

Finally, we apply the REINFORCE gradients on the Controller parameters:

with tf.name\_scope('train\_controller'):  
 # The main training operation. This applies REINFORCE on the weights of the Controller  
 self.train\_op = self.optimizer.apply\_gradients(self.gradients, global\_step=self.global\_step)  
  
logger.info('Successfully built controller')

The actual Controller network is created via the network\_generator function. As mentioned, the Controller is a recurrent neural network with a special kind of cell. However, we don't have to implement this from scratch, as the developers behind TensorFlow have already implemented a custom tf.contrib.rnn.NASCell. We simply need to use this to construct our recurrent neural network and obtain the outputs:

def network\_generator(self, nas\_cell\_hidden\_state):  
 # number of output units we expect from a NAS cell  
 with tf.name\_scope('network\_generator'):  
 nas = tf.contrib.rnn.NASCell(self.num\_cell\_outputs)  
 network\_architecture, nas\_cell\_hidden\_state = tf.nn.dynamic\_rnn(nas, tf.expand\_dims(  
 nas\_cell\_hidden\_state, -1), dtype=tf.float32)  
 bias\_variable = tf.Variable([0.01] \* self.num\_cell\_outputs)  
 network\_architecture = tf.nn.bias\_add(network\_architecture, bias\_variable)  
 return network\_architecture[:, -1:, :]

# Generating a child network using the Controller

Now, we implement a method that generates a child network using the Controller:

def generate\_child\_network(self, child\_network\_architecture):  
 with self.graph.as\_default():  
 return self.sess.run(self.cnn\_dna\_output, {self.child\_network\_architectures: child\_network\_architecture})

Once we generate our child network, we call the train\_child\_network function to train it. This function takes child\_dna and child\_id and returns the validation accuracy that the child network achieves. First, we instantiate a new tf.Graph() and a new tf.Session() so that the child network is separated from the Controller's graph:

def train\_child\_network(self, cnn\_dna, child\_id):  
 """  
 Trains a child network and returns reward, or the validation accuracy  
 Args:  
 cnn\_dna (list): List of tuples representing the child network's DNA  
 child\_id (str): Name of child network  
  
 Returns:  
 (float) validation accuracy  
 """  
 logger.info("Training with dna: {}".format(cnn\_dna))  
 child\_graph = tf.Graph()  
 with child\_graph.as\_default():  
 sess = tf.Session()  
  
 child\_network = ChildCNN(cnn\_dna=cnn\_dna, child\_id=child\_id, \*\*child\_network\_params)

We then define the input data pipeline, which uses the tf.data.Dataset creator we implemented here. In particular, we use tf.data.Iterator to create a generator that yields a batch of input tensors every time we call iterator.get\_next(). We initialize an iterator for the training and validation datasets respectively. The batch of input tensors contains the CIFAR-10 images and the corresponding labels, which we unpack at the end:

# Create input pipeline  
train\_dataset, valid\_dataset, test\_dataset, num\_train\_batches, num\_valid\_batches, num\_test\_batches = \  
 get\_tf\_datasets\_from\_numpy(batch\_size=child\_network\_params["batch\_size"])  
  
# Generic iterator  
iterator = tf.data.Iterator.from\_structure(train\_dataset.output\_types, train\_dataset.output\_shapes)  
next\_tensor\_batch = iterator.get\_next()  
  
# Separate train and validation set init ops  
train\_init\_ops = iterator.make\_initializer(train\_dataset)  
valid\_init\_ops = iterator.make\_initializer(valid\_dataset)  
  
# Build the graph  
input\_tensor, labels = next\_tensor\_batch

The input\_tensor becomes the argument to the child network's build method. We then define all the TensorFlow operations needed for training, including the prediction, loss, optimizer, and accuracy operations:

# Build the child network, which returns the pre-softmax logits of the child network  
logits = child\_network.build(input\_tensor)  
  
# Define the loss function for the child network  
loss\_ops = tf.nn.softmax\_cross\_entropy\_with\_logits\_v2(labels=labels, logits=logits, name="loss")  
  
# Define the training operation for the child network  
train\_ops = tf.train.AdamOptimizer(learning\_rate=child\_network\_params["learning\_rate"]).minimize(loss\_ops)  
  
# The following operations are for calculating the accuracy of the child network  
pred\_ops = tf.nn.softmax(logits, name="preds")  
correct = tf.equal(tf.argmax(pred\_ops, 1), tf.argmax(labels, 1), name="correct")  
accuracy\_ops = tf.reduce\_mean(tf.cast(correct, tf.float32), name="accuracy")  
  
initializer = tf.global\_variables\_initializer()

We then train the child network. Notice that when calling sess.run(...), we are no longer passing an argument for the feed\_dict parameter. Instead, we are simply calling the operations we want to run (loss\_ops, train\_ops, and accuracy\_ops). This is because the inputs are already represented as tensors in the child network's graph:

# Training  
sess.run(initializer)  
sess.run(train\_init\_ops)  
  
logger.info("Training child CNN {} for {} epochs".format(child\_id, child\_network\_params["max\_epochs"]))  
for epoch\_idx in range(child\_network\_params["max\_epochs"]):  
 avg\_loss, avg\_acc = [], []  
  
 for batch\_idx in range(num\_train\_batches):  
 loss, \_, accuracy = sess.run([loss\_ops, train\_ops, accuracy\_ops])  
 avg\_loss.append(loss)  
 avg\_acc.append(accuracy)  
  
 logger.info("\tEpoch {}:\tloss - {:.6f}\taccuracy - {:.3f}".format(epoch\_idx,  
 np.mean(avg\_loss), np.mean(avg\_acc)))

Once training finishes, we calculate the validation accuracy and return it:

# Validate and return reward  
 logger.info("Finished training, now calculating validation accuracy")  
 sess.run(valid\_init\_ops)  
 avg\_val\_loss, avg\_val\_acc = [], []  
 for batch\_idx in range(num\_valid\_batches):  
 valid\_loss, valid\_accuracy = sess.run([loss\_ops, accuracy\_ops])  
 avg\_val\_loss.append(valid\_loss)  
 avg\_val\_acc.append(valid\_accuracy)  
 logger.info("Valid loss - {:.6f}\tValid accuracy - {:.3f}".format(np.mean(avg\_val\_loss),  
 np.mean(avg\_val\_acc)))  
  
return np.mean(avg\_val\_acc)

Finally, we implement a method for training the Controller. Due to computational resource constraints, we will not parallelize the training procedure (that is, m child networks trained in parallel per Controller epoch). Instead, we will sequentially generate these child networks and keep track of the mean validation accuracy among them.

# train\_controller method

The train\_controller method is called after we build the Controller. The first step is thus to initialize all the variables and the first state:

def train\_controller(self):  
 with self.graph.as\_default():  
 self.sess.run(tf.global\_variables\_initializer())  
  
 step = 0  
 total\_rewards = 0  
 child\_network\_architecture = np.array([[10.0, 128.0, 1.0, 1.0] \*  
 controller\_params['max\_layers']], dtype=np.float32)

The first child\_network\_architecture is a list that resembles an architecture configuration and will be the argument to NASCell, which would output the first child DNA.

The training procedure consists of two for loops: one for the number of epochs for the Controller, and another for each child network the Controller generates per epoch. In the inner for loop, we generate a new child\_network\_architecture using NASCell and train a child network based on it to obtain a validation accuracy:

for episode in range(controller\_params['max\_episodes']):  
 logger.info('=============> Episode {} for Controller'.format(episode))  
 step += 1  
 episode\_reward\_buffer = []  
  
 for sub\_child in range(controller\_params["num\_children\_per\_episode"]):  
 # Generate a child network architecture  
 child\_network\_architecture = self.generate\_child\_network(child\_network\_architecture)[0]  
  
 if np.any(np.less\_equal(child\_network\_architecture, 0.0)):  
 reward = -1.0  
 else:  
 reward = self.train\_child\_network(cnn\_dna=child\_network\_architecture,  
 child\_id='child/{}'.format("{}\_{}".format(episode, sub\_child)))  
 episode\_reward\_buffer.append(reward)

After we obtain m validation accuracies, we update our Controller using the mean reward and the gradients computed with respect to the last child network's DNA. We also keep track of past mean rewards. Using the ema method implemented previously, we calculate the baseline, which we then subtract from the latest mean reward. We then call self.sess.run([self.train\_op, self.total\_loss]...) to update the Controller and calculate the Controller's loss:

mean\_reward = np.mean(episode\_reward\_buffer)  
  
self.reward\_history.append(mean\_reward)  
self.architecture\_history.append(child\_network\_architecture)  
total\_rewards += mean\_reward  
  
child\_network\_architecture = np.array(self.architecture\_history[-step:]).ravel() / self.divison\_rate  
child\_network\_architecture = child\_network\_architecture.reshape((-1, self.num\_cell\_outputs))  
baseline = ema(self.reward\_history)  
last\_reward = self.reward\_history[-1]  
rewards = [last\_reward - baseline]  
logger.info("Buffers before loss calculation")  
logger.info("States: {}".format(child\_network\_architecture))  
logger.info("Rewards: {}".format(rewards))  
  
with self.graph.as\_default():  
 \_, loss = self.sess.run([self.train\_op, self.total\_loss],  
 {self.child\_network\_architectures: child\_network\_architecture,  
 self.discounted\_rewards: rewards})  
  
logger.info('Episode: {} | Loss: {} | DNA: {} | Reward : {}'.format(  
 episode, loss, child\_network\_architecture.ravel(), mean\_reward))

And that's it! You can find the full implementation of controller.py in the main GitHub repository.

# Testing ChildCNN

Now that we have implemented both child\_network and controller, it would be great to test the training of ChildCNN via our Controller with custom child network configurations. We would like to make sure that, with a sensible architecture, ChildCNN can learn sufficiently.

To do this, first open up your favorite Terminal and start a Jupyter console:

$ ipython  
Python 3.6.4 (default, Jan 6 2018, 11:49:38)  
Type 'copyright', 'credits' or 'license' for more information  
IPython 6.4.0 -- An enhanced Interactive Python. Type '?' for help.

We first configure our logger so we can see the outputs on the Terminal:

In [1]: import sys  
  
In [2]: import logging  
  
In [3]: logging.basicConfig(stream=sys.stdout,  
 ...: level=logging.DEBUG,  
 ...: format='%(asctime)s %(name)-12s %(levelname)-8s %(message)s')  
 ...:  
  
In [4]:

Next, we import the Controller class from controller.py:

In [4]: import numpy as np  
  
In [5]: from controller import Controller  
  
In [6]:

We then handcraft some child network architecture to be passed to the Controller's train\_child\_networkfunction:

In [7]: dna = np.array([[3, 1, 30, 2], [3, 1, 30, 2], [3, 1, 40, 2]])

Finally, we instantiate our Controller and call the train\_child\_network method:

In [8]: controller = Controller()  
  
...  
  
2018-09-16 01:58:54,978 controller INFO Successfully built controller  
  
In [9]: controller.train\_child\_network(dna, "test")  
  
2018-09-16 01:58:59,208 controller INFO Training with dna: [[ 3 1 30 2]  
 [ 3 1 30 2]  
 [ 3 1 40 2]]  
**2018-09-16 01:58:59,605 cifar10\_processor INFO Dividing pixels by 255**  
**2018-09-16 01:59:01,289 cifar10\_processor INFO Loaded data from keras**  
**2018-09-16 01:59:03,150 child\_network INFO DNA is: [[3, 1, 30, 2], [3, 1, 30, 2], [3, 1, 40, 2]]**  
**2018-09-16 01:59:14,270 controller INFO Training child CNN first for 1000 epochs**

If successful, you should be seeing decent accuracy scores after several epochs of training:

**2018-09-16 06:25:01,927 controller INFO Epoch 436: loss - 1.119608 accuracy - 0.663**  
**2018-09-16 06:25:19,310 controller INFO Epoch 437: loss - 0.634937 accuracy - 0.724**  
**2018-09-16 06:25:36,438 controller INFO Epoch 438: loss - 0.769766 accuracy - 0.702**  
**2018-09-16 06:25:53,413 controller INFO Epoch 439: loss - 0.760520 accuracy - 0.711**  
**2018-09-16 06:26:10,530 controller INFO Epoch 440: loss - 0.606741 accuracy - 0.812**

# config.py

The config.py module includes configurations used by the Controller and the child networks. Here, you can adjust several training parameters, such as the number of episodes, the learning rate, and the number of child networks generated by the Controller per epoch. You can also experiment with child network sizes, but do note that the larger the child network, the longer training takes for both the Controller and the child network:

child\_network\_params = {  
 "learning\_rate": 3e-5,  
 "max\_epochs": 100,  
 "beta": 1e-3,  
 "batch\_size": 20  
}  
  
controller\_params = {  
 "max\_layers": 3,  
 "components\_per\_layer": 4,  
 'beta': 1e-4,  
 'max\_episodes': 2000,  
 "num\_children\_per\_episode": 10  
}

Some of these numbers (such as max\_episodes) are arbitrarily chosen. We encourage the reader to tweak these numbers to understand how they affect the training of both the Controller and the child networks.

# train.py

This train.py module acts as our top-level entry to training the Controller:

import logging  
import sys  
  
from .controller import Controller  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 # Configure the logger  
 logging.basicConfig(stream=sys.stdout,  
 level=logging.DEBUG,  
 format='%(asctime)s %(name)-12s %(levelname)-8s %(message)s')  
 controller = Controller()  
 controller.train\_controller()

And there we have it; a neural network that generates other neural networks! Make sure your implementation has the following directory structure:

src  
|-- \_\_init\_\_.py  
|-- child\_network.py  
|-- cifar10\_processor.py  
|-- config.py  
|-- constants.py  
|-- controller.py  
`-- train.py

To execute training, simply run the following command:

**$ python train.py**

If all works well, you should be seeing output like the following:

**2018-09-16 04:13:45,484 src.controller INFO Successfully built controller**   
**2018-09-16 04:13:45,542 src.controller INFO =============> Episode 0 for Controller**   
**2018-09-16 04:13:45,952 src.controller INFO Training with dna: [[ 2 10 2 4 1 1 12 14 7 1 1 1]] 2018-09-16 04:13:45.953482: I tensorflow/core/common\_runtime/gpu/gpu\_device.cc:1484] Adding visible gpu devices: 0**   
**2018-09-16 04:13:45.953530: I tensorflow/core/common\_runtime/gpu/gpu\_device.cc:965] Device interconnect StreamExecutor with strength 1 edge matrix:**   
**2018-09-16 04:13:45.953543: I tensorflow/core/common\_runtime/gpu/gpu\_device.cc:971] 0**   
**2018-09-16 04:13:45.953558: I tensorflow/core/common\_runtime/gpu/gpu\_device.cc:984] 0: N**   
**2018-09-16 04:13:45.953840: I tensorflow/core/common\_runtime/gpu/gpu\_device.cc:1097] Created TensorFlow device (/job:localhost/replica:0/task:0/device:GPU:0 wi th 21618 MB memory) -> physical GPU (device: 0, name: Tesla M40 24GB, pci bus id: 0000:03:00.0, compute capability: 5.2)**   
**2018-09-16 04:13:47,143 src.cifar10\_processor INFO Dividing pixels by 255**   
**2018-09-16 04:13:55,119 src.cifar10\_processor INFO Loaded data from keras**   
**2018-09-16 04:14:09,050 src.child\_network INFO DNA is: [[2, 10, 2, 4], [1, 1, 12, 14], [7, 1, 1, 1]]**   
**2018-09-16 04:14:21,326 src.controller INFO Training child CNN child/0\_0 for 100 epochs**   
**2018-09-16 04:14:32,830 src.controller INFO Epoch 0: loss - 2.351300 accuracy - 0.100**  
**2018-09-16 04:14:43,976 src.controller INFO Epoch 1: loss - 2.202928 accuracy - 0.180**   
**2018-09-16 04:14:53,412 src.controller INFO Epoch 2: loss - 2.102713 accuracy - 0.220**   
**2018-09-16 04:15:03,704 src.controller INFO Epoch 3: loss - 2.092676 accuracy - 0.232**   
**2018-09-16 04:15:14,349 src.controller INFO Epoch 4: loss - 2.092633 accuracy - 0.240**

You should see logging statements for each child network architecture its CIFAR-10 training logs. During CIFAR-10 training, we print the loss and accuracy for each epoch as well as the validation accuracy which we return to the Controller.

biggest advantage of NAS is that one does not need to spend copious amounts of time designing a neural network for a particular problem. This also means that those who are not data scientists can also create machine learning agents as long as they can prepare data. In fact, Google has already productized this framework as Cloud AutoML, which allows anyone to train customized machine learning models with minimum effort. According to Google, Cloud AutoML provides the following benefits:

* Users only need to interact with a simple GUI to create machine learning models.
* Users can have Cloud AutoML annotate their own datasets if they are not labeled already. This is similar to Amazon's Mechanical Turk service.
* Models generated by Cloud AutoML are guaranteed to have high accuracy and fast performance.
* Easy end-to-end pipeline for uploading data, training and validating the model, deploying the model, and creating a REST endpoint for fetching predictions.

Currently, Cloud AutoML can be used for image classification/detection, natural language processing (text classification), and translation.

*For more information on Cloud AutoML, check out their official page here:*[*https://cloud.google.com/automl/*](https://cloud.google.com/automl/)

Another advantage that NAS provides is the ability to generate more compact models than those designed by humans. According to *Efficient Neural Architecture Search via Parameter Sharing* by Hieu Pham et. al., whereas the most recent state-of-the-art neural network for CIFAR-10 classification had 26.2 million parameters, a NAS-generated neural network that achieved comparable test accuracy (97.44% for human-designed network versus 97.35% for the NAS-generated network) only had 3.3 million parameters. Note that older, less-accurate models such as VGG16, ResNet50, and InceptionV3 have 138 million, 25 million, and 23 million parameters respectively. The vast reduction in parameter size allows for more efficient inference time and model storage, both of which are important aspects when deploying models into production.

we have implemented NAS, a framework where a reinforcement learning agent (the Controller) generates child neural networks to complete a certain task. We studied the theory behind how the Controller learns to generate better child network architectures via policy gradient methods. We then implemented a simplified version of NAS that generates child networks that learn to classify CIFAR-10 images.

For more information on related topics, refer to the following list of links:

* NAS with reinforcement learning: <https://arxiv.org/abs/1611.01578>
* Efficient NAS via parameter sharing: <https://arxiv.org/pdf/1802.03268>
* Google Cloud AutoML: <https://cloud.google.com/automl/>
* Awesome Architecture Search—a curated list of papers related to generating neural networks: <https://github.com/markdtw/awesome-architecture-search>

The NAS framework marks an exciting development in the deep learning field, for we have figured out how to automatically design neural network architectures, a decision previously made by humans. There are now improved versions of NAS and other kinds of algorithms that generate neural networks automatically, which we encourage the reader to look into as well.