Chapter 7 - Automation

"There's a lot of automation that can happen that isn't a replacement of humans but of mind-numbing behavior."

- Stewart Butterfield, Founder (Slack)

We have talked about a variety of techniques in the last few chapters to improve efficiency and boost the quality of deep learning models. These techniques are just a small subset of the available techniques. It is often tedious to decide which ones would work for a problem even for experts. The simplest approach is to try and see which ones produce the best results. For example, between quantization and clustering, which one is preferable? What is the performance impact when both are used together? We have four options: none, quantization, clustering, and both. We would need to train a model with each of these four options to make an informed decision.

Blessed with a large research community, the deep learning field is growing at a rapid pace. Over the past few years, we have seen newer architectures, techniques and training procedures pushing the performance benchmarks higher. Figure 7-1 shows some of the choices we face when working on a deep learning problem in the vision domain for instance. Some of these choices are boolean, others have discrete parameters and still there are the ones with continuous parameters. Some choices even have multiple parameters. For example, horizontal flip is a boolean choice, rotation requires a fixed angle or a range of rotation, and random augment requires multiple parameters.

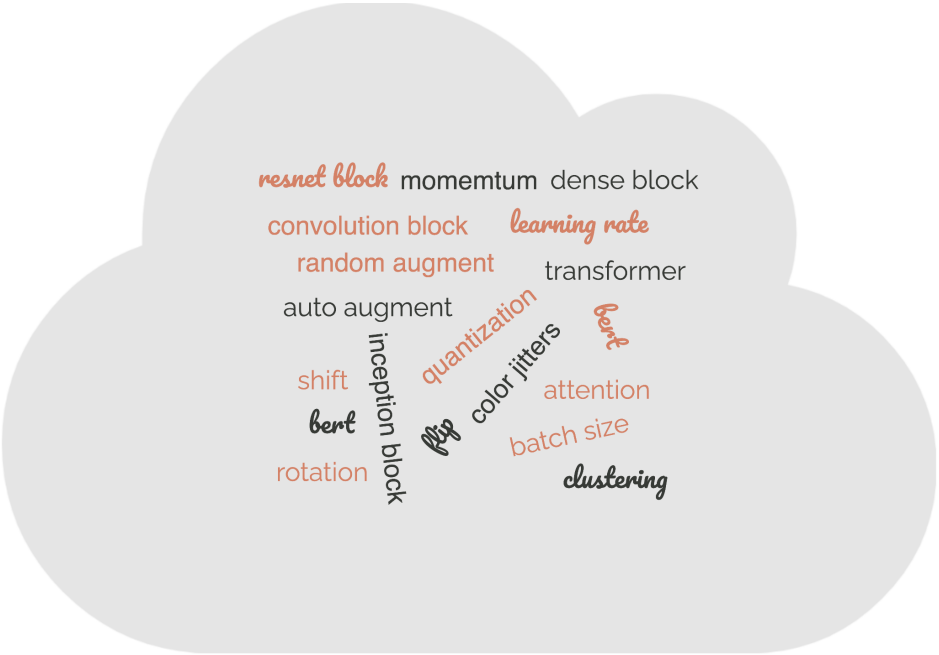


Figure 7-1: The plethora of choices that we face when training a deep learning model in the computer vision domain.

A *Search Space* Λ for *n parameters* is a *n-dimensional* region such that a point in such a region is a set of well-defined values for each of those parameters. The parameters can take discrete or continuous values. It is called a "search" space because we are searching for a point λ^* in Λ which minimizes (or maximizes) an *Evaluation Function* E. Formally, we can define λ^* as

$$\lambda^* = \arg\min_{\lambda \in \Lambda} E(\lambda)$$

Let's understand this using the earlier example for choosing quantization and/or clustering techniques for model optimization. We have a search space Λ which has *two* boolean valued parameters: quantization and clustering. A $$True$$ value means that the technique is turned on and a $$False$$ value means it is turned off. This search space[1] has four possible values such that $\Lambda = \{(False, False), (True, False), (False, True), (True, True)\}$. Let's take another example of a search space with *two* parameters. However, in this example, the second parameter is a continuous valued parameter with domain $[0.0, 1.0]$. $\{(False, 0.1), (True, 0.02), (True, 0.80), ...\}$ are some of the valid points in this search space. As we can see, this search space has infinitely many points because the second parameter can take infinitely many values.

In the context of deep learning, the parameters that influence the process of learning are called *hyperparameters* to differentiate them from model parameters. The performance of deep learning relies on a set of *good* hyperparameters. Some of the commonly tuned hyperparameters are the learning rate and the momentum of the optimization algorithm and the training batch size. Other aspects of the training pipeline like data augmentation, layer and channel configurations can also be parameterized using hyperparameters. For example, when using image data augmentation with rotation, we can treat the angle of rotation as a hyper-parameter. Think of it as tuning a guitar (a model) by turning the knobs (the hyperparameters) until we are satisfied with the sound (model quality and footprint) that each string produces. Unlike the guitar which has a few knobs, the hyperparameter search space can be quite large. Moreover, unlike a guitar knob which is associated with a single string, hyperparameters may influence each other. Hence, we need a sophisticated approach to tune them. Hyperparameter Optimization (HPO) is the process of choosing values for hyperparameters that lead to an optimal model. HPO performs trials with different sets of hyperparameters using the model as a blackbox. The set which performs the best is chosen for full training. In the next section, we'll discuss various approaches for hyperparameter optimization.

Hyperparameter Optimization

Hyperparameter Optimization improves two aspects of the training process: performance and convergence. Hyperparameters like number of filters in a convolution network or

[1] Note that this search space is just choosing if we are applying the techniques. The techniques themselves might also have additional parameters which could be searched as well.

transformation parameters in data augmentation layer contribute to performance improvements while others like learning rate, batch size or momentum are geared towards model convergence. However, they all work in conjunction to produce *better* models *faster*.

Let's say that we are optimizing the validation loss, L, for a given dataset $D(x, y))$ on a model represented by a function f with a set of hyperparameters λ. Further, assume that θ is a set of model parameters. HPO is attempting to find λ^* such that:

$$\lambda^* = \arg\min_{\lambda \in \Lambda} L(f_\lambda(x; \theta), y)$$
$$\text{where } \Lambda \text{ represents the hyperparameter search space}$$

As we stated earlier, Λ with any real valued parameter is an infinitely large search space. A common strategy is to approximate λ^* by picking a finite set of trials, $S = \{\lambda^{(1)}, \lambda^{(2)}, ..., \lambda^{(n)})\}$ such that $S \in \Lambda$. Then, we can approximate the equation X as follows:

$$\lambda^* = \underset{\lambda \in \lambda^{(1)}, \lambda^{(2)}, ..., \lambda^{(n))}}{argmin} L(f_\lambda(x; \theta), y)$$

Now, let's take up a simple exercise to find a λ^* for a given set of trials S.

A Simple Example of Hyperparameter Search

In this exercise, we will train a model with a pair of hyperparameters: layer size and learning rate. The layer size determines the model size and the learning rate is used by the model optimizer to update the model's weights as per gradient descent. We will conduct four trials with four pairs of hyperparameter values. The hyperparameter values which achieve the minimum loss are the winners. Let's start by importing the relevant libraries and creating a random classification dataset with 20 samples, each one assigned to one of the five target classes.

```
import random
import tensorflow as tf
import numpy as np
from tensorflow.keras import layers, losses, optimizers

X = tf.random.uniform((20, 5))
Y = tf.squeeze(
   tf.one_hot(tf.random.uniform((20, 1), 0, 5, tf.int64), 5)
)
```

Now, we choose the trial set, S, which contains four pairs of hyperparameter values.

```
# A set of pairs of layer size and learning rate to run HPO.
S = [
   (5, .01),
   (10, .1),
   (20, .001),
```

```
    (50, .1),
]
```

The *create_model()* function creates a single hidden layer model whose size is determined by the input *size* parameter.

```
def create_model(size):
    return tf.keras.Sequential([
        tf.keras.Input(shape=(5,5)),
        layers.Dense(size, activation='relu'),
        layers.Dense(5, activation='softmax')
    ])
```

Our model, input data and the hyperparameter trial set is ready. Let's go ahead and train the model, each time choosing one item from the trial set. Each model is trained for 2000 iterations. At the end of a trial, we record the minimum loss achieved with the associated hyperparameters.

```
search_results = []
for trial_id, (layer_size, learning_rate) in enumerate(S):
    model = create_model(size=layer_size)
    opt = optimizers.SGD(learning_rate=learning_rate)
    losses = []

    for iteration in range(2000):
        with tf.GradientTape() as tape:
            output = model(X)
            loss = tf.reduce_mean(tf.math.square(Y - output))
            grads = tape.gradient(loss, model.trainable_variables)
            opt.apply_gradients(zip(grads, model.trainable_variables))
            losses.append(loss.numpy())

    min_loss = np.min(losses)
    search_results.append(min_loss)
    fmt = 'Trial: {} learning_rate: {} layer_size: {} loss: {}'
    print(fmt.format(trial_id, learning_rate, layer_size, min_loss))

best_trial_id = np.argmin(search_results)
best_loss = np.min(search_results)

print('\n=============== Search Summary ===============')
print('Best Trial: {} Loss: {}'.format(best_trial_id, best_loss))
```

```
Trial: 0 learning_rate: 0.01 layer_size: 5 loss: 0.15629929304122925
Trial: 1 learning_rate: 0.1 layer_size: 10 loss: 0.12303829193115234
Trial: 2 learning_rate: 0.001 layer_size: 20 loss: 0.1564401090145111
Trial: 3 learning_rate: 0.1 layer_size: 50 loss: 0.11825279891490936

=============== Search Summary ===============
Best Trial: 3 Loss: 0.11825279891490936
```

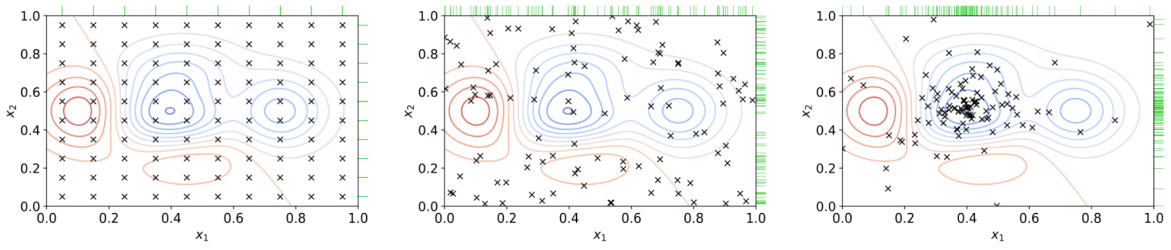
As we can see from the trial results, the last trial #3 achieves the minimum loss value. This exercise demonstrates the essence of HPO which is to perform trials with different parameter values and choose the model with best performance. In this example, we manually picked the pairs of learning rates and layer sizes. Ideally, we would want it

automated so that it doesn't require human intervention. Now, let's talk about a few search strategies starting with Grid Search.

Grid Search

A simple algorithm for automating HPO is Grid Search (also referred to as Parameter Sweep), where the trial set S consists of all the combinations of valid hyperparameters values. Each trial is configured with an element from the trial set. After all the trials are complete, we pick the one with the best results. The trials are independent of each other which makes them a good candidate for parallel execution. For example, the trial set for two hyperparameters x_1 and x_2 where $x_1 \in \{x_1^1, x_1^2, x_1^3\}$ and $x_2 \in \{x_2^1, x_2^2\}$ is $S = \{(x_1^1, x_2^1), (x_1^1, x_2^2), (x_1^2, x_2^1), (x_1^2, x_2^2), (x_1^3, x_2^1), (x_1^3, x_2^2)\}$

Figure 7-2 (a) shows results of grid search trials with two hyperparameters x_1 and x_2. The blue contours mark the positive results while the red ones indicate the trials with high losses. The density of trials is identical in both the regions which indicates that the search doesn't learn from the past results. It also has a couple of additional drawbacks. First, it suffers from the curse of dimensionality where the total number of trials grows quickly for each additional hyperparameter value or a new hyperparameter. Second, it does not differentiate between unimportant and important hyperparameters. *Important* hyperparameters have a larger number of subspaces or subranges than *unimportant* parameters that need to be searched for an optimal value. For example, in the US presidential elections, the swing states are the important parameters because they have many counties (subspaces) whose vote share has substantial variations over the election years. In other states, such counties are far lesser in number. Hence, a model to predict the results of a US presidential election would benefit from a more aggressive focus (search) on the counties in swing states (important parameters) than those in the rest of the states (unimportant parameters).



(a) Grid Search (b) Random Search (c) Bayesian Optimization

Figure 7-2: A comparison of hyperparameter search algorithms for two hyperparameters. The blue contours show the regions with positive results while the red contours indicate poor results. The 'x' marks indicate the trials. The images are sourced under CC BY-SA 4.0 license from Hyperparameter optimization article on wikipedia.

Grid Search has serious limitations for real valued hyperparameters because it requires the practitioners to choose specific values to try out. The maximum choices for such parameters are limited to the chosen set. Alternatively, we can specify ranges for such

hyperparameters and let the search algorithm pick arbitrary values in those ranges for each trial. Let's discuss this strategy next!

Random Search

Random Search (RS) samples trials randomly from the search space. For discrete parameters, it picks a value from the available choices. For real valued parameters, it samples a value randomly within the specified range. For example, given two hyperparameters x_1 and x_2 such that x_1 is real valued in range $[-1, 1]$ and $x_2 \in \{5, 10\}$, RS can generate an arbitrary number of trials. For 5 trials, one possible trial set can be $S = \{(.2, 5), (.12, 5), (-.5, 10), (.75, 5), (-.92, 10)\}$ and for 7 trials, one possible set can be $S = \{(.32, 10), (-.2, 5), (-.5, 10), (.5, 10), (-.72, 10), (.59, 5), (-.99, 10)\}$.

The total number of trials in Random Search are limited by the available computational budget. They can be increased as more resources become available or reduced in resource constrained situations. The likelihood of finding the optimal λ^* increases with the number of trials. In contrast, the Grid Search has a fixed number of maximum trials. If there are K real valued hyperparameters and N total trials, grid search would pick a maximum of $N^{\frac{1}{K}}$ values for each hyperparameter while random search will pick N different values for each hyperparameter. Since each hyperparameter gets a new value per trial, the unimportant parameters do not increase the evaluation cost.

Figure 7-2 (b) shows an example of random search. The trials are randomly spread across the search space. Even though the trials in the blue region perform better than the ones in the red region, the search algorithm makes no effort to search "more" in the blue region. In other words, it doesn't learn from the past trials. Wouldn't it be nice if we could sample more in the favorable regions? The next search strategy does exactly that!

Bayesian Optimization

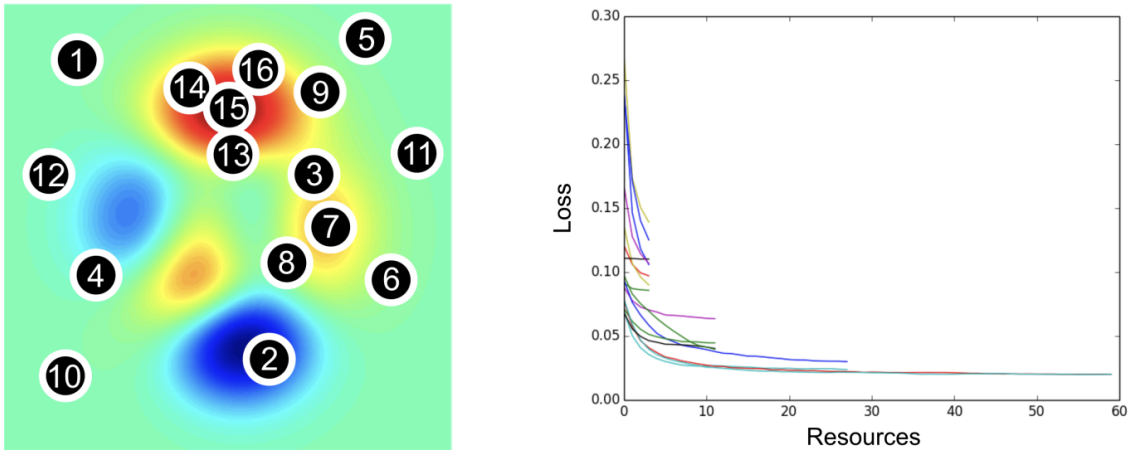
Bayesian Optimization Search (BOS) is a sequential model based search technique where the search is guided by actively estimating the value of the objective function at different points in the search space, and then spawning trials based on the information gathered so far. The objective function is estimated through a *surrogate* function that is initialized with a prior estimate. The trials are created using an *acquisition* function which picks the next trial using the surrogate function, the likelihood of improving on the optimum so far, whether to explore / exploit etc. As the trials progress, both these functions refine their estimates. Take a look at the figure 7-2 (c) which shows that the BOS trials concentrated in the blue region.

The surrogates are typically represented through Gaussian Processes, Random Forests or other statistical models. They estimate the probability that an objective function yields a value y given a set of hyperparameters λ. They are represented as a conditional probability distribution $p(y|\lambda)$. The surrogates are cheaper to compute in comparison to the *blackbox* model (the deep learning model). Hence, they can be executed much more frequently than the blackbox.

There are several choices for activation functions such as Probability of Improvement (PI), Expected Improvement (EI), Upper Confidence Bound (UCB) etc. The PI criterion maximizes the probability of improving over the current best value while the EI criterion estimates the expected improvement over the current best for a hyperparameter set.

Since the method keeps an internal model of how the objective function looks and plans the next trials based on that knowledge, it is a model-based technique. Moreover, since the selection of trials depends on the results of the past trials, this method is sequential. In contrast to the previous search methods, BOS is a guided search. Hence, it requires fewer trials to reach the optimum. However, the sequential nature hampers the search parallelization. Another drawback of BOS is its reliance on statistical distributions to estimate the objective function which introduces noise in the system.

Figure 7-3 (a) shows BOS for a two dimensional search space. It indicates that the search adaptively explores the areas with lower validation errors. This approach is also called Configuration Selection because we are aiming to find optimal hyperparameter values. BOS is likely to reach the optimum configuration faster than Grid and Random searches. Figure 7-3 (b) shows an alternative search approach which evaluates multiple configurations and adaptively allocates more resources to the promising ones. This is called Configuration Evaluation. Let's discuss it in detail in the next section.



(a) Configuration Selection (b) Configuration Evaluation

Figure 7-3: (a) Bayesian Optimization Search on a two dimensional search space. The red areas correspond to lower validation errors. The points are labeled with the iteration number indicating that the algorithm adaptively chooses configurations with lower validation errors. (b) This plot shows the validation error as a function of resources allocated to each configuration. Promising configurations get more resources. Source: Hyperband[2]

[2] Li, Lisha, et al. "Hyperband: A novel bandit-based approach to hyperparameter optimization." *The Journal of Machine Learning Research* 18.1 (2017): 6765-6816.

Hyperband

The configuration selection scenario in figure 7-3 (a) took 16 iterations to reach the optimum hyperparameters. That means the model had to be trained 16 times to finally decide on the hyperparameters for the final training. For large models, this is very expensive in terms of time and resources. Alternatively, we can base the search approach on the budget allocation to cap the resource utilization.

Multi-Armed Bandit based algorithms allocate a finite amount of resources to a set of hyperparameter configurations. The trials for each configuration are run for multiple iterations. All the trials in an iteration are allocated an identical budget. The configurations are promoted to the next iteration based on an evaluation criteria such that some of them are discarded in each iteration. The search stops when we have just one (the best) configuration left. An intuitive way to think about it is to imagine a multiplayer game with multiple levels where a few best performing players are promoted to the next level until we have a winner.

There are various strategies for evaluation criteria. Successive Halving[3] promotes half of the configuration in each iteration to the next based on their performance. The worse performing half is discarded. If we start with N configurations, it takes

Hyperband attempts to resolve the problem of choosing N by choosing brackets where each bracket starts with a pair of (n, r) such that n is the number of configurations and r is the minimum budget for each configuration in the bracket. For large n, r is smaller and vice-versa to ensure that each bracket gets a comparable budget.

Take a look at table 7-1 which shows the changes in the number of configurations (n, r) as the iterations progress for each bracket. In comparison to successive halving, in this example, two-thirds of the configurations are dropped in each iteration and one-third carry forward. The freed resources from the dropped configurations are allocated to the remaining configurations in the bracket. Bracket 0 in the first iteration has $(n, r) = (81, 1)$ which consumes a total of $81x1 = 81$ resources. In the second iteration, the total number of resources remains identical $27x3 = 81$ with each remaining configuration being allocated a larger share of resources. In contrast to the bracket 0, subsequent brackets start with a smaller set of configurations and higher resource allocation per configuration. This ensures that we try successive halves with various values of N.

	Brackets (n, r)				
Iterations	0	1	2	3	4
0	81, 1	27, 3	9, 9	6, 27	5, 81

[3] Jamieson, Kevin, and Ameet Talwalkar. "Non-stochastic best arm identification and hyperparameter optimization." *Artificial intelligence and statistics*. PMLR, 2016.

1	27, 3	9, 9	3, 27	2, 81	
2	9, 9	3, 27	1, 81		
3	3, 27	1, 81			
4	1, 81				

Table 7-1: A demonstration of configuration and resource allocation changes across multiple brackets in a Hyperband. Source: Hyperband

In chapter 3, we trained a model to classify flowers in the *oxford_flowers102* dataset. In the next section, we will retrain the same model but with a twist!

Project: Oxford Flower Classification With Hyperparameter Tuning

Recall that in chapter 3, we trained a ResNet based model to classify *oxford_flowers102* flowers dataset. We used *two* hyperparameters: LEARNING_RATE and DROPOUT_RATE. The learning rate was set to 0.0002 and the dropout rate was 0.2. The model reached the top accuracy of 70% after training for 100 epochs. In this project, we will let the HyperBand choose the best values for these hyperparameters and see if we can do better. We will use the *keras_tuner* package which has an implementation of HyperBand.

The hyperband algorithm requires *two* additional parameters: *max_epochs* and a *factor*. The *max_epochs* parameter is the most amount of resources a trial can consume. The *factor* parameter decides the fraction of configurations that are dropped after each round. For successive halving, this is set to 2. For HyperBand, the recommended factor is 3. We will use the same. Now, let's go on and load the required modules and the dataset.

```
import tensorflow as tf
import tensorflow_datasets as tfds
import keras_tuner as kt
import numpy as np

from matplotlib import pyplot as plt
from tensorflow.keras import applications as apps
from tensorflow.keras import layers, optimizers

train_ds, val_ds, test_ds = tfds.load(
   'oxford_flowers102',
   split=['train', 'validation', 'test'],
   as_supervised=True,
   read_config=tfds.ReadConfig(try_autocache=False)
)
```

Let's resize the dataset splits to the same size. The target size is identical to the project in chapter 3.

```
# Dataset image size
IMG_SIZE = 264
```

```
def resize_image(image, label):
    image = tf.image.resize(image, [IMG_SIZE, IMG_SIZE])
    image = tf.cast(image, tf.uint8)
    return image, label

train_ds = train_ds.map(resize_image)
val_ds = val_ds.map(resize_image)
test_ds = test_ds.map(resize_image)
```

Note that the *create_model()* function here has two additional parameters: *learning_rate* and *dropout_rate* which replace the global LEARNING_RATE and DROPOUT_RATE parameters from chapter 3. We have an additional function *build_hp_model()* here which takes a *hp* parameter that refers to a *keras_tuner. HyperParameters()* object. The *hp* parameter is used to create hyperparameters which are managed by *keras_tuner*. We create two float type hyperparameters: *learning_rate* in range [.0001, .01] and *dropout_rate* in range [.1, .8]. The *build_hp_model()* is called by the tuner to create a model for each trial with the chosen values for the *learning_rate* and *dropout_rate*.

```
DROPOUT_RATE = 0.2
LEARNING_RATE = 0.0002
NUM_CLASSES = 102

def build_hp_model(hp):
    if hp:
        learning_rate = hp.Float(
            "learning_rate",
            min_value=1e-4,
            max_value=1e-2,
            sampling="log"
        )
        dropout_rate = hp.Float(
            "dropout_rate",
            min_value=.1,
            max_value=.8,
            step=.1
        )

    return create_model(learning_rate, dropout_rate)

def create_model(learning_rate=LEARNING_RATE, dropout_rate=DROPOUT_RATE):
    # Initalize the core model
    core_args = dict(input_shape=(IMG_SIZE, IMG_SIZE, 3), include_top=False)
    core = apps.resnet50.ResNet50(**core_args)
    core.trainable = False

    # Setup the top
    model = tf.keras.Sequential([
        layers.Input([IMG_SIZE, IMG_SIZE, 3], dtype = tf.uint8),
        layers.Lambda(lambda x: tf.cast(x, tf.float32)),
        layers.Lambda(lambda x: apps.resnet.preprocess_input(x)),
        core,
        layers.Flatten(),
        layers.Dropout(dropout_rate),
        layers.Dense(NUM_CLASSES, activation='softmax')
    ])
```

```
    adam = optimizers.Adam(learning_rate=learning_rate)

    model.compile(
        optimizer=adam,
        loss='sparse_categorical_crossentropy',
        metrics=['accuracy']
    )
    return model

# model = create_model()
model = build_hp_model(kt.HyperParameters())
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lambda (Lambda)              (None, 264, 264, 3)       0
_________________________________________________________________
lambda_1 (Lambda)            (None, 264, 264, 3)       0
_________________________________________________________________
resnet50 (Functional)        (None, 9, 9, 2048)        23587712
_________________________________________________________________
flatten (Flatten)            (None, 165888)            0
_________________________________________________________________
dropout (Dropout)            (None, 165888)            0
_________________________________________________________________
dense (Dense)                (None, 102)               16920678
=================================================================
Total params: 40,508,390
Trainable params: 16,920,678
Non-trainable params: 23,587,712
_________________________________________________________________
```

Next, we initialize the HyperBand *tuner*. Noteworthy initialization arguments are *hypermodel*, *objective*, *max_epochs* and *factor*. The *hypermodel* argument is set to the *build_hp_model* callback defined earlier. It is used to create a model with the chosen values for the hyperparameters. The *objective* refers to the metric to score the trial quality. Each trial can use a maximum of *max_epochs* resources and

```
tuner = kt.Hyperband(
    hypermodel=build_hp_model,
    objective="val_accuracy",
    max_epochs=10,
    factor=3,
    hyperband_iterations=1,
    overwrite=True,
    directory="hpo",
    project_name="hyperband",
)
tuner.search_space_summary()
```

```
Search space summary
Default search space size: 2
learning_rate (Float)
{'default': 0.0001, 'conditions': [], 'min_value': 0.0001, 'max_value': 0.01, 'step': None,
'sampling': 'log'}
dropout_rate (Float)
```

```
{'default': 0.1, 'conditions': [], 'min_value': 0.1, 'max_value': 0.8, 'step': 0.1,
'sampling': None}
```

We are now ready to start the search. The *search()* method of tuner takes the training and the validation sets to run the search.

```
tds = train_ds.batch(32)
vds = val_ds.batch(256)

tuner.search(tds, validation_data=vds)
tuner.results_summary(num_trials=3)
```

```
Trial 30 Complete [00h 01m 24s]
val_accuracy: 0.6313725709915161

Best val_accuracy So Far: 0.7284313440322876
Total elapsed time: 00h 17m 23s

Results summary
Results in hpo/hyperband
Showing 3 best trials
Trial summary
Hyperparameters:
learning_rate: 0.00026592775384539827
dropout_rate: 0.7000000000000001
tuner/epochs: 10
tuner/initial_epoch: 4
tuner/bracket: 2
tuner/round: 2
tuner/trial_id: 0012
Score: 0.7284313440322876

Trial summary
Hyperparameters:
learning_rate: 0.00026592775384539827
dropout_rate: 0.7000000000000001
tuner/epochs: 2
tuner/initial_epoch: 0
tuner/bracket: 2
tuner/round: 0
Score: 0.7166666388511658

Trial summary
Hyperparameters:
learning_rate: 0.00017821013875888482
dropout_rate: 0.5
tuner/epochs: 4
tuner/initial_epoch: 0
tuner/bracket: 1
tuner/round: 0
Score: 0.7107843160629272
```

It took 17 minutes and a few seconds to execute this search. The best model **reached 72.8% accuracy after 10 epochs** which is **significantly better** than the results in chapter 3 where the model reached 70% accuracy after 100 epochs. The search results suggest that a model **with a dropout rate of 70% converges much faster** than the model with 20% dropout rate and achieves a better accuracy as well.

Table 7-2 shows a breakdown of trials for this run. Note that the bracket ids are in reverse order in contrast to the example in table 7-1. The tuner runs a total of 30 trials in three brackets. The maximum number of epochs for any trial is 10. The trial in bracket 2 reaches the best performance in the last round.

	Brackets (n, r)		
Iterations	2	1	0
0	12, 2	6, 4	4, 10
1	4, 4	2, 10	
2	2, 10		

Table 7-2: A breakdown of trials in the tuner run. (n, r) corresponds to the number of trials and the maximum available resource for the trial in a bracket and iteration. Bracket 2 achieves the best performance in iteration 2.

It would be interesting to do a 100 epoch run with 70% dropout rate and see if we can do better than 72.8% accuracy. We leave that exercise to the reader. The total number of epochs spent in the search are 144. That is about 1.5 training runs. If we go back and look at figure 7-X for BOS, it took 16 runs to converge to the optimum hyperparameters. However, there are other ways to make BOS run quicker by using smaller datasets, early stopping or low resolution inputs etc. Early Stopping can even be applied with the HyperBand to terminate the runs sooner if they do not show improvements for a number of epochs.

The algorithms like HyperBand bring the field of HPO closer to the evolutionary approaches which are based on biological mechanisms like mutation and natural selection. The promotion of better performing trials to the next iteration (round) can be viewed as selection or survival of the fittest. Population Based Training[4] (PBT) incorporate these biological mechanisms to evolve better models. It spawns a fixed number of trials (referred as population) and trains them to convergence. Each trial is trained for a predetermined number of steps with random values for the hyperparameters. Then, every trial's weights and hyperparameters are replaced with those from the *best* trial in the population. This is the exploitation part of the search which selects the fittest model. For exploration, the hyperparameters are slightly perturbed to mimic the mutation aspect of evolution. This process repeats till convergence. PBT can be naturally parallelized because the trials are largely independent of each other until the mutation phase. They work well for adaptive hyperparameters such as learning rate and weight decay.

The hyperparameter search can be extended beyond training parameters to structural parameters that can manipulate the structure of a network. The number of dense units, number of convolution channels or the size of convolution kernels can sometimes be

[4] Jaderberg, Max, et al. "Population based training of neural networks." *arXiv preprint arXiv:1711.09846* (2017).

searched with the techniques that we discussed in this section. However, to truly design a Neural Network from scratch, we need a different approach. The next section dives into the search for neural architectures.

Neural Architecture Search

On a high level, Neural Architecture Search (NAS) is similar to Hyperparameter Search. In both cases, we search for parameters to optimize a blackbox function. However, in the case of NAS, these parameters define the architecture of the model that represents the blackbox function. In the hyperparameter tuning project, we searched for the value of *dropout_rate* which influences the model architecture. In fact, we could use HPO to decide whether adding a dropout layer is a good idea. Neural Architectures are composed of layers stacked on top of each other with a given layer processing the output of the previous layers. However, HPO techniques are insufficient to model this ordered structure because they do not model the concept of *order* well. Another limitation of HPO is the search for variable length architectures. HPO requires all the hyperparameters to be known prior to the start of the search.

In their paper titled "Neural Architecture Search With Reinforcement Learning"[5], Zoph et. al. employed neural networks to search for optimal neural architectures for image classification and language modeling. Their generated models exhibited strong performance on the image and language benchmark datasets. Moreover, their NAS model could generate variable depth *child networks*. Figure 7-4 shows a sketch of their search procedure. It involves a *controller* which samples the search space to generate candidate architectures. The candidates are used as a reference to construct the child networks which are trained on the problem dataset and their performance metrics are fed back to the controller as reward signals. The controller incorporates the rewards signals in its gradient updates. Zoph et. al. modeled NAS as a reinforcement learning (RL) problem where the controller is a recurrent model and the child networks are the players whose rewards are determined by their performance on the target dataset. The controller model learns to generate better architectures as the search game progresses.

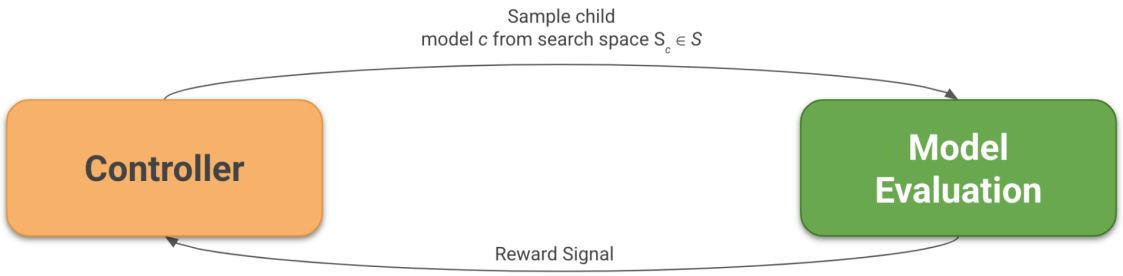


Figure 7-4: An overview of Neural Architecture Search framed as a Reinforcement Learning problem.

[5] Zoph, Barret, and Quoc V. Le. "Neural architecture search with reinforcement learning." *arXiv preprint arXiv:1611.01578* (2016).

A typical NAS recurrent model predicts *one* action in each timestep. The next time step takes the output action from the previous time step as input to generate the *next* action and so on. We can design a recurrent model with a fixed or a variable number of time steps. Figure 7-5 shows a general architecture of the NAS recurrent model. The time step at t_k takes the output of time step t_{k-1} as input. The output of t_k is subsequently fed to the next time step t_{k+1}. The output of each time step is a prediction for the target child architecture. The output is typically fed to a softmax layer to choose from a discrete set of choices.

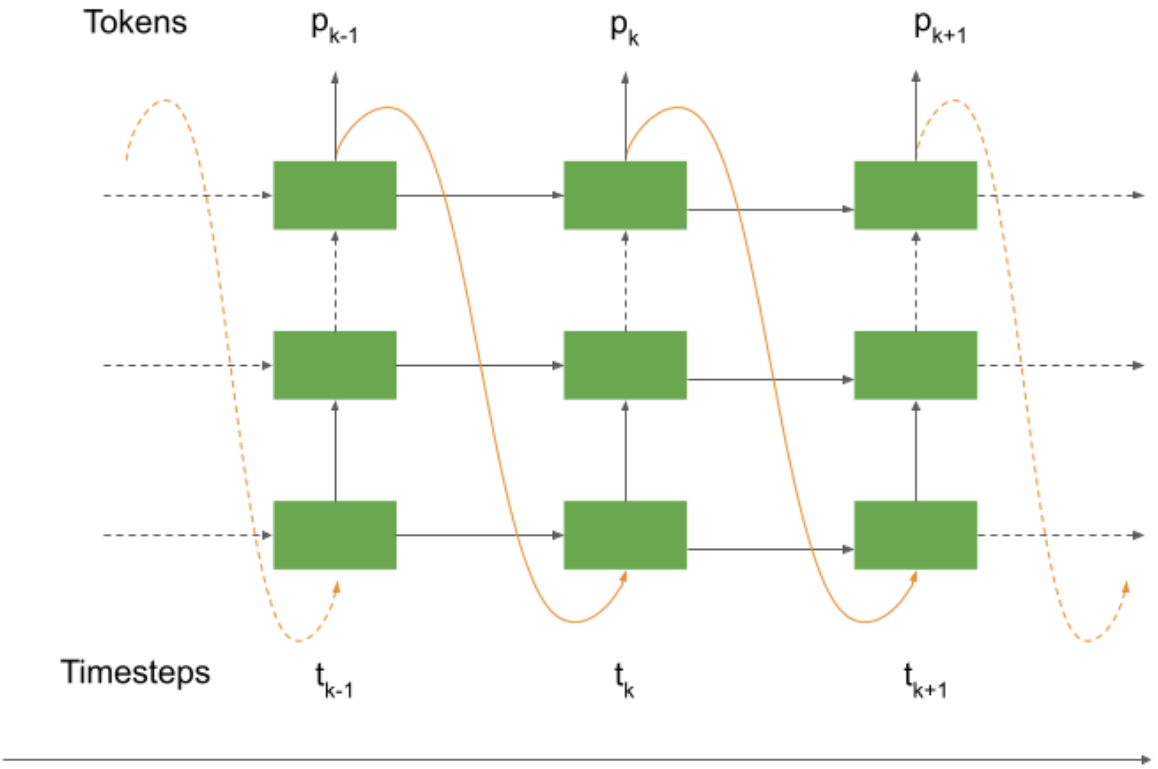


Figure 7-5: The architecture of an RNN controller for NAS. Each time step t outputs a token p. The output token is fed as input to the next time step and so on. The figure also shows multiple recurrent units stacked (vertical stack) on top of each other to learn complex relationships between the time steps.

Zoph et. al. formulated the architectural search as an expectation maximization problem. Given a set of actions $a_{1..T}$ which produce a child network with an accuracy R, an RNN controller maximizes the expected reward (accuracy) $J(\theta_c)$ represented as follows:

$$J(\theta_c) = E_{P(a_{1:T};\theta_c)}[R]$$
$$\text{where } \theta_c \text{ are controller parameters}$$

They used a policy gradient method to iteratively update the controller. The updates are described as follows:

$$\nabla\theta_c J(\theta_c) = \sum_{t=1}^{T} E_{P(a_{1:T};\theta_c)} \left[\nabla\theta_c \log P(a_t|a_{(t-1):1};\theta_c)R\right]$$

In other words, the above equation scales the gradients from Gradient Descent with a reward signal R. The reward signal is based on the accuracy of the child network over the dataset of interest. The policy gradient rule in equation X can be generalized for m child networks in a batch as follows:

$$\frac{1}{m}\sum_{k=1}^{m}\sum_{t=1}^{T} \nabla\theta_c \log P(a_t|a_{(t-1):1};\theta_c)R_k$$

Figure 7-6 shows the architecture of the controller used by Zoph et. al. to generate a simple convolution network. Each timestep outputs a convolution layer parameter such as *number of filters*, *filter height*, *filter width* and other parameters required to describe a convolution layer. It outputs the parameters for the first layer, followed by the parameters for the second layer and so on. The outputs from the timesteps are passed through a softmax classifier to reduce the search space to fewer choices. A subsequent version of this controller added additional parameters for each layer to allow skip connections.

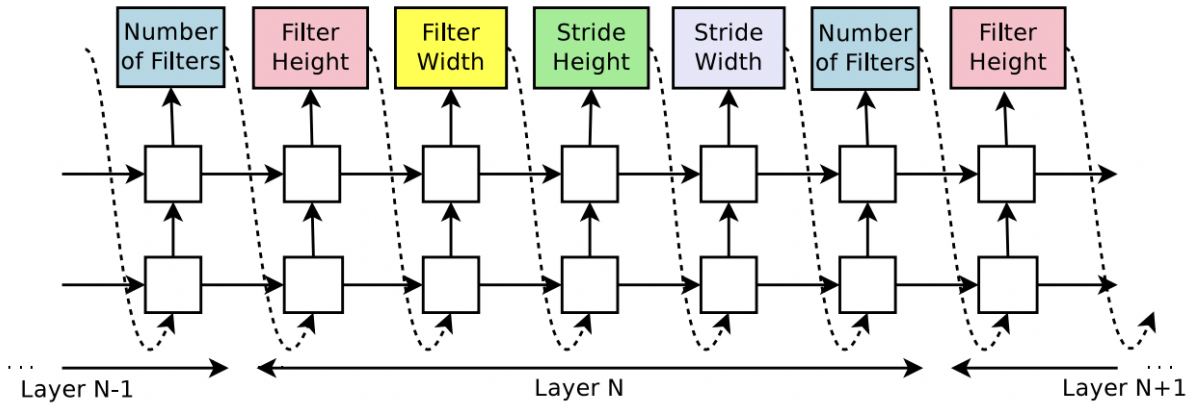


Figure 7-6: The architecture of a controller which predicts convolutional networks. Each timestep predicts a parameter of the convolution layer. The controller predicts these parameters layer by layer. Source: Neural Architecture Search with Reinforcement Learning.

The generated child networks performed at par with the SOTA networks at the time. However, this controller design had two main drawbacks. First, the architecture of the child network is tied closely to the controller. To add an additional layer to the child, we would need to update the controller as well. Second, the controller needs to be scaled with the child networks. For a large child network, a large controller is required which would invariably lead to higher search expenses.

In a follow up paper, Zoph et. al. addressed the above shortcomings with a novel controller architecture called NASNet[6] which predicts the architecture of cells that are used as building

[6] Zoph, Barret, et al. "Learning transferable architectures for scalable image recognition." *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018.

blocks for the child networks. NASNet searches for the cells that are fitted into a hand-designed child network. The reward signal to the controller is still based on the performance of the child network which tunes it to search for cells that result in a high performance child network. NASNet has a much refined search space because it is predicting fewer overall parameters. The predicted cells can be used to design a small, large or a very large child network without any changes to the controller.

NASNet predicts two types of cells: a *Normal* and a *Reduction* cell. A normal cell's output feature map is identical to the input feature map. In contrast, a reduction cell reduces the output feature map to *half*. Figure 7-7 shows two child networks that use these cells as building blocks. The network on the left is smaller which was used to classify the *cifar10* dataset. The larger network on the right is designed to learn from the ImageNet dataset. Both of these networks are largely composed of alternate stacks (of size N) of normal and reduction cells which demonstrates the scalability of NASNet.

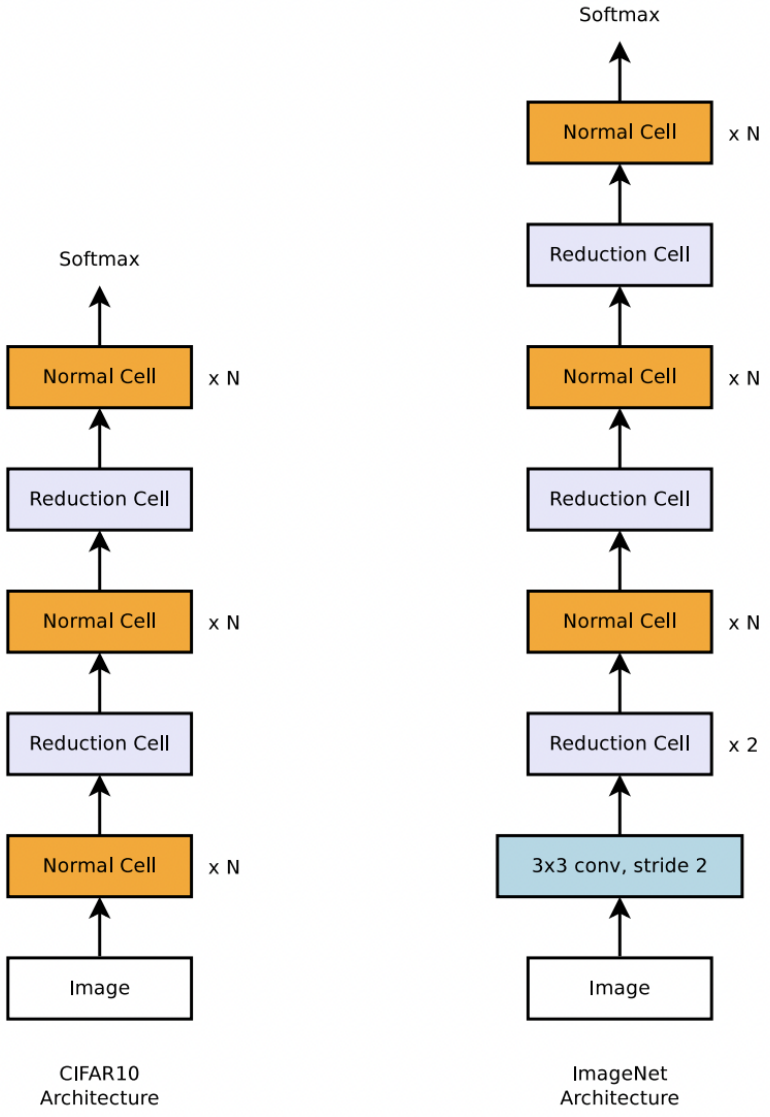


Figure 7-7: The architectures of two networks designed using the Normal and Reduction cells as the building blocks. The larger network stacks a higher number of normal and

reduction cell blocks. Source: Learning transferable architectures for scalable image recognition.

NASNet cells are composed of blocks. A single block corresponds to two *hidden* inputs, two *primitive* operations for the hidden states, and *a combination* operation as shown in figure 7-8 (left). NASNet predicts these five inputs and operations for every block. Each cell contains B such blocks. Hence, for each cell, NASNet predicts $5 \times B$ parameters. Since we predict the design of *two* cells, the total number of predicted parameters is $2 \times 5 \times B$. In the original NASNet paper, the value for B is chosen to be 5. Figure 7-8 (right) shows a predicted block.

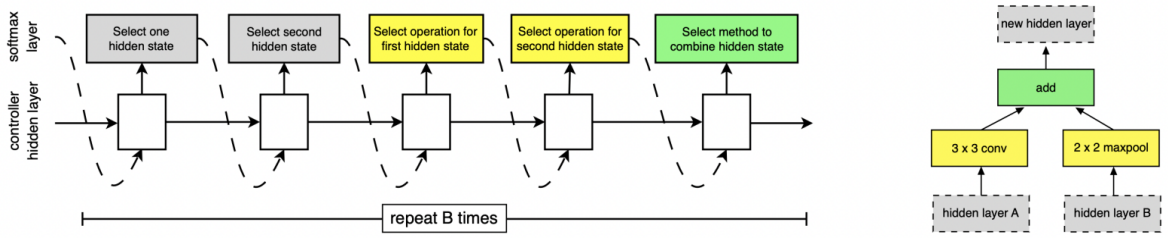


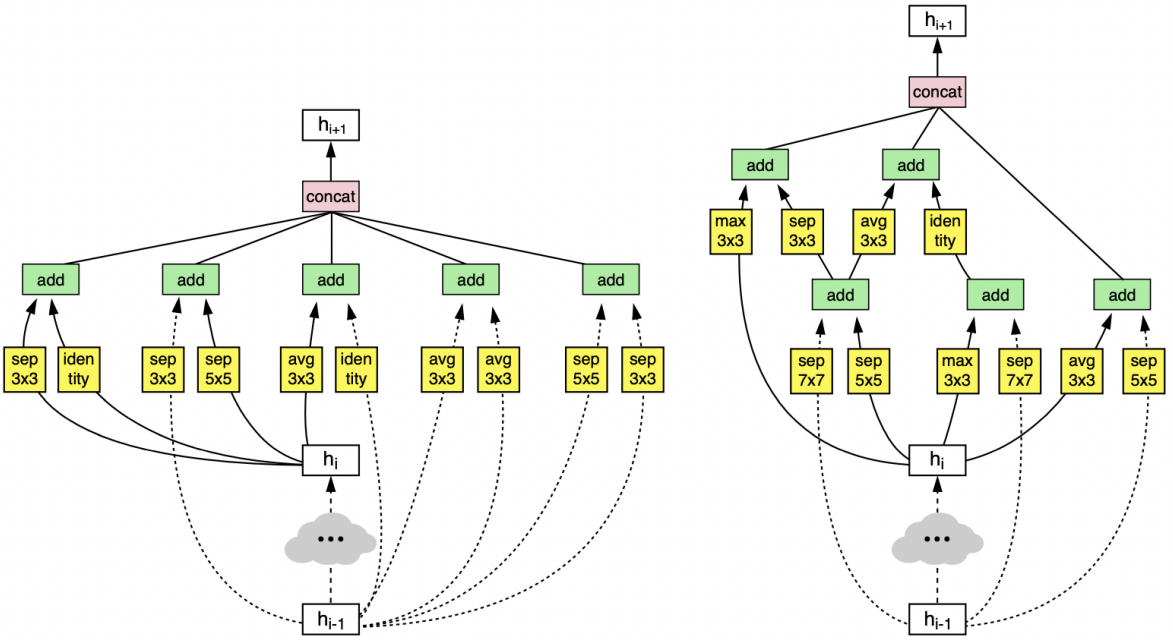
Figure 7-8: The structure of a block used to compose normal and reduction cells. The image on the left shows the timesteps predicting the hidden states, primitive operations and the combinations operations. Right image shows the structure of a block after applying the predictions from NASNet. NASNet selects the *add* operation for combining the output of two predicted primitive operations *3x3 conv* and 2x2 *maxpool*. Source: Learning transferable architectures for scalable image recognition.

The output of a cell is the concatenated output of all the B component blocks. The inputs to the cell are the outputs of the last two cells. The blocks are predicted in a sequential order. The first block chooses the two hidden inputs from the cell inputs. The second block, however, gets to choose its inputs from the two cell inputs and the output of the first block and so on. The final block has 6 input choices: two cell inputs and four block outputs. The freedom to have a block choose the output of previous blocks as input enables the possibility of hierarchical organization of the blocks which could produce more complex cells.

For primitive operations, NASNet chooses from a list of 13 frequently used operations in convolution networks such as regular convolutions, max pooling, average pooling, depthwise separable convolutions etc. These operations have associated parameters such that 3x3 convolution and 7x7 convolution are two different choices for primitive operations. The combination operation has two choices: element wise addition or the concatenation of output of primitive operations. The concatenation operation happens along the filter dimension to keep the feature map intact.

Figure 7-9 shows the Normal and Reduction cells predicted by NASNet with the *cifar10* dataset. The normal cell has a single level where all the B blocks receive cell inputs. The

output of the cell is the concatenated output of the cell blocks. The reduction cell, however, has the blocks organized in a hierarchy such that two blocks receive the outputs of other blocks in the cell as inputs. All the blocks are predicted to use *add* as a combination operation.



Normal Cell *Reduction Cell*

Figure 7-9: The architecture of the Normal and Reduction cells predicted by NASNet for CIFAR-10 dataset. The normal cell has a flat structure while the reduction cell is predicted with a hierarchical structure indicating that it is capable of predicting complex cell structures. Source: Learning transferable architectures for scalable image recognition.

The child networks predicted by NASNet for CIFAR-10 and ImageNet approached or exceeded the contemporary state of the art models. These child networks were smaller and more efficient than the human designed models. However, the key contribution of NASNet was the focus on predicting the components of child networks which enabled the construction of multiscale networks without needing to tweak the controller for a target child.

The early RL based NAS models used the validation accuracy as a primary reward signal R which is perfect for the applications that have sufficient compute resources at their disposal. However, on mobile and edge devices with limited compute capabilities, inference latencies become an important concern. Hence, the reward signal needed to incorporate an indicator for the inference latencies on the target device to search for a Pareto-Optimal model. There were models which used proxy metrics such as FLOPS as an indicator of performance on target devices. But, for the same amount of FLOPS, target latencies vary wildly. Later on, Mnasnet[7] introduced a multiobjective reward function which combined the accuracy and latency metrics. It searched for Pareto optimal child networks by computing their latencies

[7] Tan, Mingxing, et al. "Mnasnet: Platform-aware neural architecture search for mobile." *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2019.

directly on the target devices and weighing the model accuracy based on the latency values. Given a model m, its accuracy $ACC(m)$, its latency $LAT(m)$ and the target latency T, Mnasnet maximizes the following objective:

$$\underset{m}{maximize}\ ACC(m) \times \left[\frac{LAT(m)}{T}\right]^{w}$$

where w is a weight factor defined as,

$$w = \begin{cases} \alpha \text{ if } LAT(m) \leq T \\ \beta \text{ otherwise} \end{cases}$$

such that α and β variables control the reward penalty for latency violation.

In addition to the multiobjective optimization, Mnasnet also adopted mobile optimized convolution operations which provide a good accuracy-latency tradeoff. Overall, it still followed the fundamental design of a RNN based controller similar to its predecessors.

The idea to design block and cell structures by predicting inputs, primitive operations and combination operations naturally fit into evolution based architecture search where a crossover event could be implemented through a random tweak to the configuration of a block. In the paper[8] titled *Regularized Evolution for Image Classifier Architecture Search*, the authors used evolution algorithms to search for the best *Normal* and *Reduction* cells in the NASNet. The architecture resulting from the neuroevolution of these cells was named AmoebaNet-A. The high level architecture of the AmoebaNet-A models is identical to the NASNets as shown in figure 7-7. However, it takes an evolutionary approach to search for the cell architectures. The search is initialized with a population P of models with randomly generated cells. The models are evaluated on the target dataset and their performance is recorded. The best performing model in a random sample S of models from P is selected for mutation. After the mutation, the child's performance is recorded and the oldest model from P is discarded. This process is repeated for C cycles.

There are two main mutations used in the mutation step: *hidden state* mutation and the *operation* mutation. One of these mutations is selected at random in each cycle. A block is randomly picked from a randomly selected cell (normal or reduction) as a mutation candidate. For hidden state mutation, a hidden state is picked at random and is replaced with another hidden state in the cell such that no cycles are created. For *operation* mutation, a primitive operation in the block is replaced with a randomly selected operation from the state space. Figure 7-10 shows the structures of the *Normal* and the *Reduction* cells for AmoebaNet-A after the evolution search. In comparison to the NASNet, the normal AmoebaNet-A cell has three layers. NASNet and AmoebaNet demonstrated comparable

[8] Real, Esteban, et al. "Regularized evolution for image classifier architecture search." *Proceedings of the aaai conference on artificial intelligence*. Vol. 33. No. 01. 2019.

performances on CIFAR-10 and ImageNet datasets. However, a larger version of AmoebaNet-A established a new state of the art performance on ImageNet.

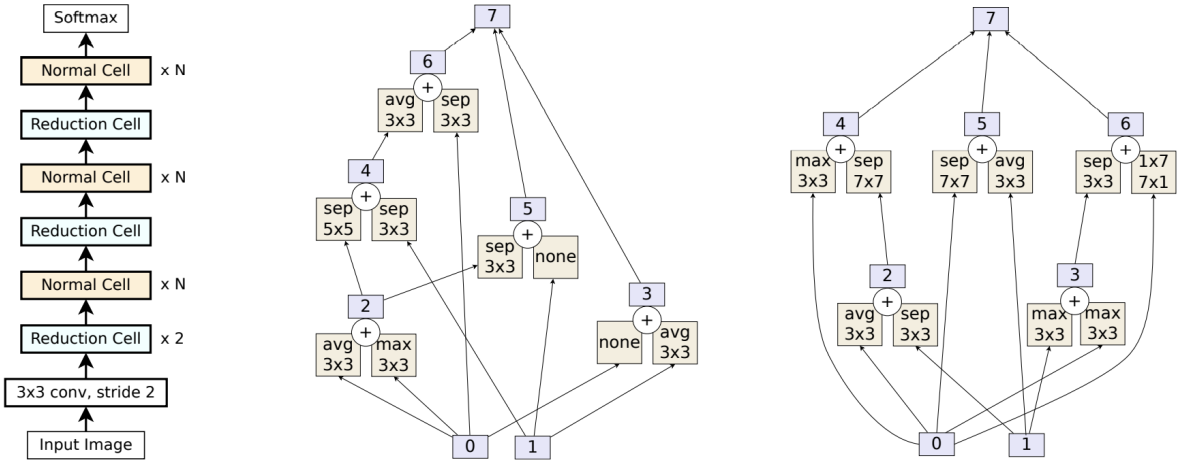


Figure 7-10: The left image shows the architecture of the AmoebaNet-A model. Note that the arrangement of the cell stacks is identical to the NASNet. The two images on the right are the architectures of the cells discovered through the evolution search. Both NASNet and AmoebaNet search the same search space.

Both NASNet and AmoebaNet made significant contributions to the neural architecture search field. NASNet contributed to decoupling the controller and the child architectures in addition to defining a smaller search space for architecture design. AmoebaNet, on the other hand, applied evolutionary search to NASNet search space to evolve novel cell configurations. It is exciting to see these two seemingly separate areas borrowing ideas from each other and pushing the benchmarks even higher. Now, it's time for us to undertake a project to build NASNet from scratch to apply our understanding.

Project: Neural Architecture Search for CIFAR-10 Dataset

Our goal is to build a simplified NASNet and employ it to design normal and reduction cells for a CIFAR-10 classification network. The arrangement of the cells is identical to the network shown in figure 7-7. We have chosen *N=1*. Our primitive operation search space contains *six* operations:

- 3x3 Depthwise Separable Convolution
- 5x5 Depthwise Separable Convolution
- 7x7 Depthwise Separable Convolution
- 3x3 Average Pool
- 3x3 Max Pool
- Identity Operation

The combination operations are identical to the NASNet namely, *add* and *concat*. The cells are single layer cells which means that each cell block only gets to choose from the cell inputs. NASNet, in contrast, allows the cell blocks to choose from the outputs of the previous blocks in the cell which allows the formation of multiple layers in the cell. In this project, we will use the NASCell as a recurrent unit. The choice of the recurrent cell is arbitrary.

We train the RNN for 150 episodes. Each episode involves *three* distinct steps. In the first step, the RNN predicts architectures for normal and reduction cells. To balance the exploration and exploitation aspects of the search, we sample random cell architectures with probability p. The second step involves training the child network and obtaining a reward. The reward is a measure of improvement over the previous performances. It is computed as the difference of the accuracy of the child network in the current episode and the past accuracies. The past accuracies are represented by a moving average with a fixed window size. The third and final step is to train the RNN with the cell architectures and their reward signals.

Let's start with installing the required packages. We need *tensorflow_addons* which contains an implementation of the NASCell.

```
!pip install tensorflow-addons
```

Next, we load the required modules and initialize the random seeds.

```
import random
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
import tensorflow.keras as keras
import tensorflow_addons as tfa
import tensorflow_datasets as tfds
import pandas as pd

from tensorflow.keras import layers, optimizers
from collections import deque
from matplotlib.ticker import MaxNLocator
from math import pow

SEED = 111

tf.random.set_seed(SEED)
np.random.seed(SEED)
random.seed(SEED)
```

We will need the CIFAR-10 dataset to evaluate the performance of the child networks. Let's download and prepare our dataset next.

```
DATASET_PARAMS = dict(
    input_shape=(32, 32, 3),
    n_classes=10,
    baseline_accuracy=.1,
)

train_ds, val_ds = tfds.load(
    'cifar10',
    split=['train[:80%]', 'train[20%:]'],
    as_supervised=True
)
```

We do a 80-20 split of the training set to create training and validation splits. You'll notice a *baseline_accuracy* constant in *DATASET_PARAMS*. It will be used later on to compute the initial reward signal. Next, we define the state space and a few configurations for the child network.

```
CHILD_PARAMS = dict(
    epochs=6,
    batch_size=128,
    learning_rate=0.001,
    train_ds=train_ds,
    val_ds=val_ds,
    rolling_accuracies_window=20,
    max_branch_length=2,
    blocks=5,
    cells=2,
    initial_width=1,
    initial_channels=4
)

STATE_SPACE = [
    dict(name='hidden_state', values=list(range(2)), count=2),
    dict(
        name=primitives,
        values=['sep_3x3', 'sep_5x5', 'sep_7x7', 'avg_3x3', 'max_3x3', 'identity'],
        count=2
    ),
    dict(name='combinations', values=['add', 'concat'], count=1),
]
```

The *STATE_SPACE* has three components to mimic the NASNet search space. The *hidden_state* element can take two values to represent the two input hidden states to a cell. The *primitives* and the combination elements represent the primitive and the combination operation choices. The *count* property of each *STATE_SPACE* element indicates the number of timesteps per block that state appears. For example, each block has two primitive operations and their inputs to choose from the available operations and hidden state inputs. Hence, we have the count=2 for both of them. Their detailed usage will be explained shortly.

```
class CNNCell():
    """
    It composes a cell based on the input configuration.
    Arguments:
        stride: A positive integer to represent the convolution strides.
            Normal cells use stride=1 and reduction cells use stride=2
    """
    def __init__(self, stride=1, channels=64):
        self.channels = channels
        self.stride = stride
        self.kwargs = dict(strides=(1, 1), padding='same')

    def repair_channels(self, inp):
    """
    This method sends the input through a convolution layer with
    a fixed channel size to ensure that the inputs to each cell block
    have identical channel dimensions.
    """
```

```
        return layers.Conv2D(self.channels, 1, padding='same')(inp)

    def repair_branches(self, branches):
    """
    It transforms the input branches to an identical feature space.
    It is useful when a cell receives inputs with different feature spaces.
    """
        (hidden_1, width_1), (hidden_2, width_2) = branches

        if width_1 != width_2:
            hidden_1 = layers.Conv2D(
                self.channels,
                3,
                strides=(2,2),
                padding='same'
            )(hidden_1)
        else:
            hidden_1 = self.repair_channels(hidden_1)

        hidden_2 = self.repair_channels(hidden_2)

        return [hidden_1, hidden_2]

    def reduce_inputs(self, inputs):
    """
    It halves the input feature space using a convolution layer with stride 2
    """
        if self.stride == 2:
            inputs = list(
                map(
                    lambda inp: layers.Conv2D(
                        self.channels,
                        3,
                        strides=(2,2),
                        padding='same'
                    )(inp),
                    inputs
                )
            )

        return inputs

    def apply_op(self, op_id, inp):
    """
    It applies a primitive operation to the input.
    """
        if op_id == 0:
            output = layers.DepthwiseConv2D(3, **self.kwargs)(inp)
        elif op_id == 1:
            output = layers.DepthwiseConv2D(5, **self.kwargs)(inp)
        elif op_id == 2:
            output = layers.DepthwiseConv2D(7, **self.kwargs)(inp)
        elif op_id == 3:
            output = layers.AveragePooling2D(3, **self.kwargs)(inp)
        elif op_id == 4:
            output = layers.MaxPool2D(3, **self.kwargs)(inp)
        elif op_id == 5:
            output = inp
```

```
        return output

    def combination(self, op_id, inp_1, inp_2):
    """
    It applies a combination operation to the inputs.
    """
        if op_id == 0:
            output = inp_1 + inp_2
        elif op_id == 1:
            x = tf.concat([inp_1, inp_2], -1)
            output = self.repair_channels(x)

        return output

    def compose_block(self, block_config, inputs):
        input_1, input_2 = list(map(lambda idx: inputs[idx], block_config[:2]))
        op1, op2 = block_config[2:4]
        combine_op = block_config[-1]

        output = self.combination(
            combine_op,
            self.apply_op(op1, input_1),
            self.apply_op(op2, input_2)
        )

        return output

    def make_cell(self, cell_config, branches):
    """
    It constructs a cell based on the cell_config and the branches.
    A cell_config is 2-D numpy array of shape [5,5] which contains
    the 5 state choices for each of the 5 blocks.
    """
        inputs = self.repair_branches(branches)
        inputs = self.reduce_inputs(inputs)

        blocks = []
        for block in cell_config:
            block_output = self.compose_block(block, inputs)
            blocks.append(block_output)

        x = tf.concat(blocks, -1)
        output = self.repair_channels(x)

        return output
```

The *CNNCell()* class is responsible for the construction of cells given the predicted (or randomly sampled) cell config and the hidden cell inputs. *make_cell()* is the entry method to the class that is called with the *cell_config* and the *branches* arguments. The *cell_config* argument is a numpy array of shape (5, 5) which contains 5 state choices for each of the 5 blocks. Before the cell construction, we standardize the two branch inputs to an appropriate feature space and channel size. First, we project both the branches to identical channel dimensions to support addition combination operation. Next, we project the branches to identical feature space as well to support concatenation combination operation. The two branches could have different feature space if they are coming from different cell types. For

example, the feature space of the previous reduction cell and previous-previous normal cell differ by a factor of 2. Moreover, after every concatenation operation, we repair channels yet again for standardization.

The following code defines a *ChildManager* class which is responsible for spawning child networks, training them, and computing rewards. The *layers* constant defined in the class indicates the stacking order of the cells. Each element of *layers* is a pair of numbers which indicate the cell type and its strides respectively. A normal cell is a type 0 cell with strides of size (1,1). A reduction cell is a type 1 cell with strides of size (2,2). The *initial_channels* and *initial_width* configurations in *CHILD_PARAMS* define the number of channels for each cell. The bottom cell output has 4 channels. The number of channels double with each layer as we move towards the top of the network.

```
class ChildManager():
   def __init__(self):
       self.tds = CHILD_PARAMS['train_ds'].shuffle(
           500,
           reshuffle_each_iteration=True
       ).batch(CHILD_PARAMS['batch_size'])
       self.vds = CHILD_PARAMS['val_ds'].batch(256)

       self.past_accuracies = deque(
           maxlen=CHILD_PARAMS['rolling_accuracies_window']
       )
       self.past_accuracies.append(DATASET_PARAMS['baseline_accuracy'])

       self.layers = [(0, 1), (1, 2), (0, 1), (1, 2), (0, 1)]

   def make_child(self, config):
       """
       Arguments:
           config: It is an array of shape [2, 5, 5]
                   Each row represents a cell: [Normal, Reduction]
                   Each cell contains 5 blocks.
                   Each block contains 5 operations.

       Returns:
           model: A CNN model with the layers laid out based on the input config.
       """
       inp = tf.keras.Input(shape=DATASET_PARAMS['input_shape'], dtype=tf.uint8)
       x = layers.Rescaling(1./255)(inp)

       width = CHILD_PARAMS['initial_width']
       branches = deque(maxlen=CHILD_PARAMS['max_branch_length'])
       branches.append((x, width))
       branches.append((x, width))

       for cell_type, stride in self.layers:
           width *= stride
           cnn_cell = CNNCell(
               stride,
               channels=CHILD_PARAMS['initial_channels']**width
           )

           x = cnn_cell.make_cell(config[cell_type], branches)
```

```
            branches.append((x, width))

        x = layers.GlobalAveragePooling2D()(x)
        output = layers.Dense(DATASET_PARAMS['n_classes'], activation='softmax')(x)

        model = tf.keras.Model(inp, output)
        optimizer = optimizers.Adam(learning_rate=CHILD_PARAMS['learning_rate'])

        model.compile(
            optimizer=optimizer,
            loss='sparse_categorical_crossentropy',
            metrics='accuracy'
        )
        model.summary()

        return model

    def train(self, model):
        history = model.fit(
            self.tds,
            validation_data=self.vds,
            epochs=CHILD_PARAMS['epochs']
        )
        accuracy = max(history.history['val_accuracy'])

        return accuracy

    def get_rewards(self, config):
        model = self.make_child(config)
        accuracy = self.train(model)

        self.past_accuracies.append(accuracy)
        rolling_accuracy = (sum(self.past_accuracies)/len(self.past_accuracies))
        reward = accuracy - rolling_accuracy

        return reward, accuracy
```

The *get_rewards()* method is the main entrypoint into the *ChildManager* class. It is called by the controller to obtain rewards for a sampled or predicted child configuration. This method performs *three* distinct steps. The first step is to construct a child network based on the input configuration. In the second step, the child network is training on the CIFAR-10 dataset. The third step involves computing reward which is the difference between the accuracy and the rolling average of past accuracies over a configured window. The method returns the reward and the latest accuracy.

Moving on, we define a few configurations for the controller to decide the exploitation/exploration ratio, the hidden size of the recurrent cell, the number of training episodes for the controller and the controller learning rate. We also define a variable *STATE_ADDRESSES* that contains the mapping between time steps and the corresponding *(cell, block, state)* tuple.

```
CONTROLLER_PARAMS = dict(
    exploration=0.5,
    hidden_size=32,
    episodes=150,
```

```
    learning_rate=0.001,
)

BLOCK_TIMESTEPS = 5
CELL_TIMESTEPS = BLOCK_TIMESTEPS*CHILD_PARAMS['blocks']
TOTAL_TIMESTEPS = CELL_TIMESTEPS*CHILD_PARAMS['cells']
STATE_ID_TO_STATE_SPACE_ID = {0:0, 1:0, 2:1, 3:1, 4:2}

TIMESTEP_ADDRESS_SPACE = (
    CHILD_PARAMS['cells'],
    CHILD_PARAMS['blocks'],
    len(STATE_ID_TO_STATE_SPACE_ID)
)

TIMESTEP_ADDRESSES = np.stack(
    np.unravel_index(range(TOTAL_TIMESTEPS),TIMESTEP_ADDRESS_SPACE),
    -1
)

STATE_ADDRESSES = list(map(
    lambda x: [x[0], x[1], STATE_ID_TO_STATE_SPACE_ID[x[2]]],
    TIMESTEP_ADDRESSES
))
```

The *Controller* class defined below has *three* entry points: *predict_child()*, *save_trial()* and *train_step()*. The RNN model is a simple model which takes a start state as the input. Each recurrent time step corresponds to a state prediction. NASNet makes 5 predictions for each block. It predicts 5 blocks for each cell. Hence, the total number of predictions is 2x5x5=50 where the number of cells is 2. The input to the RNN is an input id which is chosen from $0, 1$ to indicate the input to the first primitive operation. It is mapped to an embedding through an embedding table to transform it to *hidden_size* dimensions of the RNN cell. A softmax layer is applied to the cell outputs to convert cell outputs to the probabilities of choosing an element in the state space corresponding to the time step.

The *predict_child()* method predicts a child configuration. The *exploration* configuration controls whether the child configuration is generated randomly or it is predicted by the recurrent model. The *save_trial()* method records the trial history. The *train_step()* method trains the recurrent networks for a single step with the historical data. It scales the model gradients with the reward value to accelerate the search in the directions which yield higher rewards.

```
class Controller():
    def __init__(self):
        self.rnn = self.make_rnn()
        self.store = dict(children=[], rewards=[], accuracies=[])
        self.loss_fn = tf.keras.losses.SparseCategoricalCrossentropy()
        self.optimizer = optimizers.Adam(
            learning_rate=CONTROLLER_PARAMS['learning_rate']
        )

    def make_rnn(self):
        start_state = tf.keras.Input(shape=(), dtype=tf.int32)
        rnn_cell = tfa.rnn.NASCell(CONTROLLER_PARAMS['hidden_size'])
```

```
        initial_data = {
            'cell_state': [
                tf.zeros((1, CONTROLLER_PARAMS['hidden_size']), dtype=tf.float32),
                tf.zeros((1, CONTROLLER_PARAMS['hidden_size']), dtype=tf.float32)
            ],
            'embeddings': tf.keras.layers.Embedding(
                len(STATE_SPACE[0]['values']),
                CONTROLLER_PARAMS['hidden_size']
            )
        }

        prev_output = initial_data['embeddings'](start_state)
        prev_cell_state = initial_data['cell_state']
        predictions = []

        print('Controller Timesteps: ', TOTAL_TIMESTEPS)

        for timestep, (cell, block, state) in enumerate(STATE_ADDRESSES):
            prev_output, prev_cell_state = rnn_cell(prev_output, prev_cell_state)

            prediction_layer = tf.keras.layers.Dense(
                len(STATE_SPACE[state]['values']),
                activation='softmax',
                name='softmax_%d'%timestep
            )
            pred_action = prediction_layer(prev_output)
            predictions.append(pred_action)

        model = tf.keras.Model(inputs=start_state, outputs=predictions)
        model.summary()

        return model

    def predict_child(self, start_state):
        if np.random.random() < CONTROLLER_PARAMS['exploration']:
            def sample_state_space(x):
                (_, __, state_id) = x
                return random.randrange(len(STATE_SPACE[state_id]['values']))

            predictions = list(map(
                sample_state_space,
                STATE_ADDRESSES
            ))
            print(
                'Random Predictions:',
                np.array(predictions).reshape(TIMESTEP_ADDRESS_SPACE)
            )
        else:
            predictions = self.rnn([start_state])
            predictions = list(map(lambda x: np.argmax(x), predictions))

            print(
                'RNN Predictions:',
                np.array(predictions).reshape(TIMESTEP_ADDRESS_SPACE)
            )

        return predictions

    def save_trial(self, child, reward, accuracy):
```

```
        self.store['children'].append(child)
        self.store['rewards'].append(reward)
        self.store['accuracies'].append(accuracy)

    def train_step(self):
        child_config = self.store['children'][-1]
        reward = self.store['rewards'][-1]
        start_state = np.array([child_config[0]])

        with tf.GradientTape() as tape:
            outputs = controller.rnn(start_state)
            loss = 0.

            for index, output in enumerate(outputs):
                y_true = [child_config[index]]
                loss += self.loss_fn(y_true, output)

            grads = tape.gradient(loss, controller.rnn.trainable_variables)

            for index, grad in enumerate(grads):
                grads[index] = tf.multiply(grad, reward)

            self.optimizer.apply_gradients(
                zip(grads, controller.rnn.trainable_variables)
            )
```

The next chunk of code puts everything together and runs the search for 150 episodes.

```
controller = Controller()
child_manager = ChildManager()
start_state = np.array([random.randrange(len(STATE_SPACE[0]))])

for episode in range(CONTROLLER_PARAMS['episodes']):
    predictions = controller.predict_child(start_state)

    config = np.array(predictions).reshape(TIMESTEP_ADDRESS_SPACE)

    # Evaluate the child generated by the controller
    reward, accuracy = child_manager.get_rewards(config)
    print(
        'Episode: {} Reward: {} Accuracy: {}'.format(
            episode,
            reward,
            accuracy
        )
    )

    # Store predicted child and its rewards
    controller.save_trial(predictions, reward, accuracy)

    # Train the Controller
    controller.train_step()

    # Update start_state for next episode.
    start_state = np.array([predictions[0]])
    tf.print('Start State:', start_state)
```

```
Episode: 0 Reward: 0.13005000054836274 Accuracy: 0.36010000109672546
```

```
Episode: 1 Reward: 0.29068332711855566 Accuracy: 0.6660749912261963
Episode: 2 Reward: 0.2597874984145164 Accuracy: 0.7217749953269958
Episode: 3 Reward: -0.06779000878334046 Accuracy: 0.37724998593330383
Episode: 4 Reward: 0.1551124890645345 Accuracy: 0.6311749815940857
Episode: 5 Reward: 0.18273214272090366 Accuracy: 0.6892499923706055
                                        xxxxx
Episode: 50 Reward: -0.02466248571872709 Accuracy: 0.6651750206947327
Episode: 51 Reward: -0.009128761291503862 Accuracy: 0.6808249950408936
Episode: 52 Reward: 0.04923374354839327 Accuracy: 0.7421000003814697
Episode: 53 Reward: 0.012139973044395402 Accuracy: 0.7044249773025513
Episode: 54 Reward: 0.023349997401237443 Accuracy: 0.7184000015258789
Episode: 55 Reward: 0.04027997255325322 Accuracy: 0.7440999746322632
                                        xxxxx
Episode: 120 Reward: 0.0738500043749809 Accuracy: 0.7706249952316284
Episode: 121 Reward: -0.049653741717338606 Accuracy: 0.6621749997138977
Episode: 122 Reward: 0.04787125289440153 Accuracy: 0.7609249949455261
Episode: 123 Reward: -0.010424998402595476 Accuracy: 0.7005749940872192
Episode: 124 Reward: 0.022921249270439148 Accuracy: 0.7346749901771545
Episode: 125 Reward: 0.013183763623237588 Accuracy: 0.7234500050544739
                                        xxxxx
Episode: 145 Reward: -0.04218623638153074 Accuracy: 0.6523500084877014
Episode: 146 Reward: 0.009832504391670271 Accuracy: 0.704925000667572
Episode: 147 Reward: -0.012747514247894332 Accuracy: 0.6819999814033508
Episode: 148 Reward: 0.02163251936435695 Accuracy: 0.7150750160217285
Episode: 149 Reward: 0.006060001254081682 Accuracy: 0.701324999332428
```

Figure 7-11 shows the accuracies and the rolling accuracies (window=10) observed in each training episode. The rolling accuracies start to stabilize after episode 60. The model in episode 120 reaches the top accuracy of 77%. The accuracies start to dip again in the final episodes. It would be interesting to see if this trend reverses if we train for more episodes.

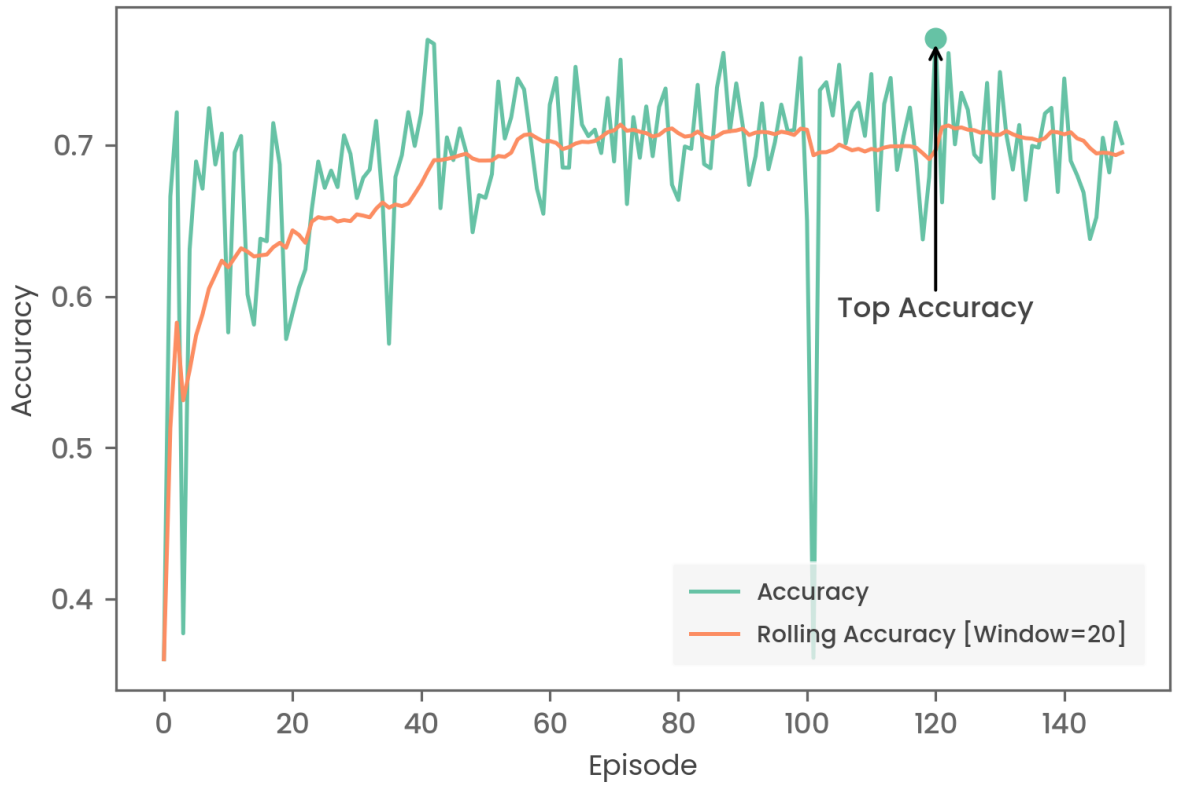


Figure 7-11: The graph showing the accuracies and rolling accuracies for the child networks predicted by the controller network.

Figure 7-12 shows the architectures of the predicted normal and reduction cells. Both the cells have flat hierarchies as enforced by the block input constraints. None of the predicted cells use 5x5 or 7x7 convolution layers.

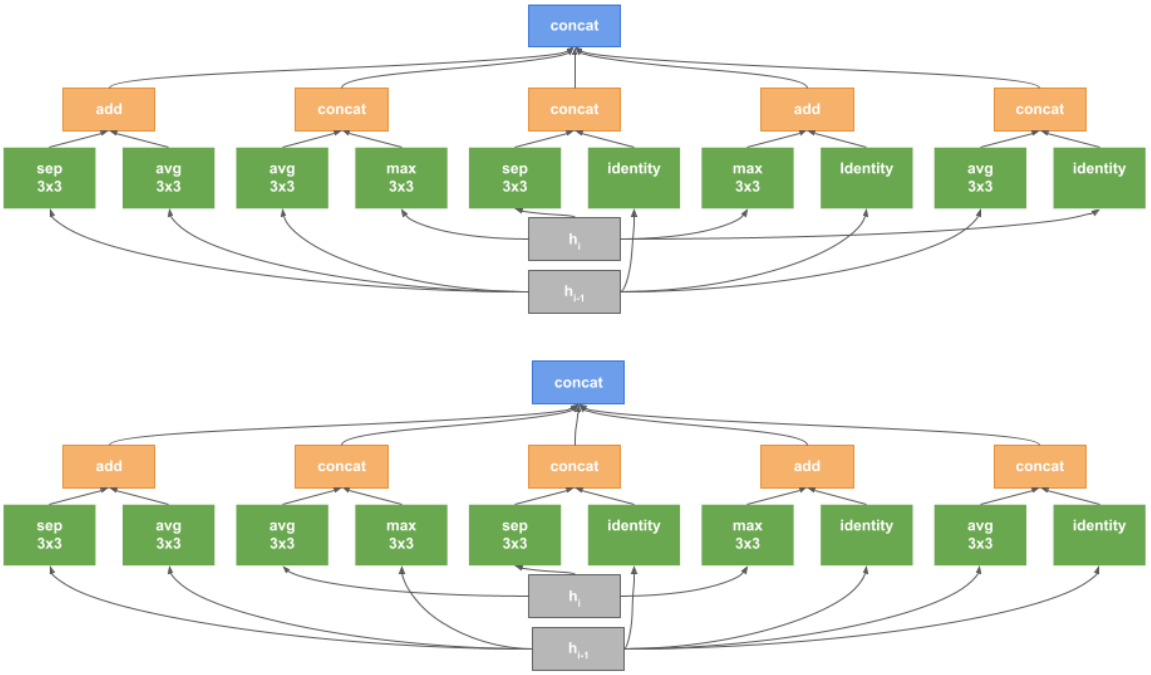


Figure 7-12: The cells predicted by the controller network. On the top is the normal cell. The bottom subgraph shows the architecture of the reduction cells. Both the cells have flat hierarchy because each block gets to choose from the cell inputs h_{i-1} and h_i. This is a deviation from NASNet which allows the past block outputs to be considered for the subsequent block inputs.

The project was a simple exercise to familiarize ourselves with RNN based architecture search. We took a few liberties with the design choices in comparison to the NASNet. These choices were made with the goals to simplify the design and to give ourselves an opportunity to experiment. We encourage the readers to play with other choices and see how they affect the results. It's now time to conclude our discussion on automation with a short introduction to *Automated ML* or *AutoML* in the final section.

Summary

In the past decade, deep learning has made incredible progress. Every couple of years, we see new architectural designs which surpass the current state of the art. Along with the architectures, the approaches to tune the training parameters are evolving as well. In the HPO section, we discussed strategies to search for good hyperparameters which can help to squeeze better performance out of a model. Traditionally, we relied on experts who used their intuition and a fair bit of trial-and-error to tune hyperparameters. However, in a fast paced environment, intuitions become outdated quickly and the trial-and-error approach is suitable for tuning a small number of hyperparameters. It was clear that some kind of automation was needed which led to the growth of the HPO field.

Something similar happened in the data preparation domain where the techniques for data cleaning and data augmentation saw extensive growth. Popular deep learning frameworks such as PyTorch and Tensorflow added support for data cleaning and augmentation but they left the responsibilities of choosing appropriate parameters to the deep learning practitioners.

With the rapid growth of deep learning, several frameworks have popped up to automate hyperparameter tuning. Frameworks such as Keras Tuner, Ray Tune, Talos and Optuna have made it straightforward to configure the training procedure to try out different configurations for the hyperparameters and choose a configuration that works best for the problem. A few other frameworks such as Auto Keras, Auto WEKA, NNI and auto-sklearn have gone even further and adopted the doctrine of AutoML which aims to automate most of the steps involved in the machine learning pipelines to reduce the dependency on ML experts and to promote large-scale adoption of machine learning. An AutoML pipeline assumes all the responsibilities which traditionally required ML experts. Imagine that we are developing an application to identify a flower from its picture. We have access to a flowers dataset (*oxford_flowers102*). As an application developer, with no experience with ML, we would like a model trained on the flowers dataset to integrate into our application. The goal of AutoML is to produce such a model.