Refactoring of Neural Network Models for Hyperparameter Optimization in Serverless Cloud

Alex Kaplunovich†
Department of Computer Science
University of Maryland
Baltimore, Maryland, USA
akaplun1@umbc.edu

Yelena Yesha†
Department of Computer Science
University of Maryland
Baltimore, Maryland, USA
yeyesha@umbc.edu

ABSTRACT

Machine Learning and Neural Networks in particular have become hot topics in Computer Science. The recent 2019 Turing award to the forefathers of Deep Learning and AI - Yoshua Bengio, Geoffrey Hinton, and Yann LeCun proves the importance of the technology and its effect on science and industry. However, we have realized that even nowadays, the state of the art methods require several manual steps for neural network hyperparameter optimization. Our approach automates the model tuning by refactoring the original Python code using open-source libraries for processing. We were able to identify hyperparameters by parsing the original source and analyzing it. Given these parameters, we refactor the model, add the state of the art optimization library calls, and run the updated code in the Serverless Cloud. Our approach has proven to eliminate manual steps for an arbitrary TensorFlow and Keras tuning. We have created a tool called OptPar which automatically refactors an arbitrary Deep Neural Network optimizing its hyperparameters. Such a transformation can save hours of time for Data Scientists, giving them an opportunity to concentrate on designing their Machine Learning algorithms.

KEYWORDS

Neural Networks, Refactoring, Hyperparameter, Automation, Optimization, Tuning, Machine Learning, Serverless, Cloud, AWS

1 Introduction

Although Machine Learning is becoming more and more popular, we have seen very few attempts to refactor the actual code. Instead, many key industry players (Amazon, Google or Microsoft) either create models under the hood (invisible to the practitioners), or require Data Scientists to implement several time consuming manual steps to tune models.

 ACM ISBN 978-1-4503-7963-2/20/05…$15.00
https://doi.org/10.1145/3387940.3392268

The first approach does not give developers the flexibility to create, develop, test and modify their algorithms. Usually, the generated code is not accessible to the end-user. It is just a black box. The concept "No Data Science experience required, we will do the job for you" is too risky in our opinion.

The second approach, requires the practitioners to identify hyperparameters and their ranges manually. Such an approach is used by many Machine Learning tools including AWS's Sagemaker, Google's Datalab or Azure's Machine Learning Studio. Moreover, users have to manually specify the parameters for each of their models. Such an approach definitely requires extra time and can be automated. OptPar was designed to address the problem.

We decided that it will be very beneficial to refactor a code automatically to improve the model performance.

2 Approach

Our tool was designed to work with an arbitrary neural network source. It parses Python TensorFlow and Keras code, identifies the hyperparameters and refactors the model.

2.1 Hyperparameter Identification

Hyperparameters are values passed to Keras methods (learning rate, batch size, dropout, number of neurons, etc.). They affect the behavior of a Machine Learning model and need tuning. Modern parsers provide very reliable, flexible and robust way to work with source code. In order to find the parameters, we will need to provide a metadata corresponding to Keras API. In Python each parameter can be passed by name or by index. We will be handling both cases.

We have created a metadata csv file containing information about Keras layer methods, hyperparameter locations and acceptable ranges. Such a file is static, and needs to be loaded just once before our system starts processing a model.

Metadata provides a snapshot of Keras layers, activations, optimizers and initializers. It contains details about the API parameters and their values, names and locations. It was created from the Keras online documentation [4]. Each row corresponds to one hyperparameter and contains the following columns:
Layer, ParamName, Hptype, TupleSize, ParamOrder, Base, Type, Min, Max, Int, and Set.

We can specify the parameter types (Integer, Float, String or Set) as well as desired parameter ranges. They will be used to find the ideal set of parameters via Bayesian optimization.

2.2 Model Parsing

Our tool is using ast (Abstract Syntax Trees) event Python parser implementing the node visitor pattern. It allows us to specify the events of interest, such as method call, return statement, assignment or expression. It allows us to parse every method's call and identify the hyperparameters for our model. Later on, we will update the parameter references in the code to use them during model tuning.

Model parsing starts from a single *ast.parse* call, creating a parse tree from the model source code. Once we were able to create a tree, we pass it to the method visit of the ast.NodeTransformer object. During that call, we can process events for every Python construct via visit_* methods for each of the language construct (method call, declaration, assignment, expression, return statement, etc.).

```
class
GetNeuralNetorkTree(ast.NodeTransformer):
    def __init__(self, tree,
updateParams=True):
        self.tree = tree
        self.updateParams = updateParams

    def visit_Print(self, node):
        new_node =
ast.Expr(value=ast.Call(func=ast.Name(id
='print', ctx=ast.Load()),
args=node.values,

keywords=[], starargs=None,
kwargs=None))
        ast.copy_location(new_node,
node)
        return new_node

    def visit_Num(self, node):
        return node

    def visit_Call(self, node):
        return node
```

Figure 1: Parsing a model and updating a node (visitor pattern)

Figure 1 demonstrates how can we parse our code and change the tree nodes. If a visit_* method returns an original node, it stays unchanged. If we need to change a source code, we just create a new ast node (as in the method visit_Print) and return that node from the method. The resulting parse tree will contain the updated nodes. Moreover, using the astunparse library, we will be able to generate the code from the modified parse tree.

Such an approach allows us to have full control of the generated code, updating only the nodes we need for the events of interest, keeping other code intact.

2.3 Model Refactoring

If a parameter matches the metadata definitions (see section 2.1), we update the model code to parametrize the hyperparameter. Figures 2 and 3 demonstrate how we update model parameters. We replace constant or passed variable parameters with the params['param_id'] which will be used during the model tuning step.

```
model = Sequential()
# Step 1 - Convolution
model.add(Conv2D(32, (3, 3), 3,
input_shape = (64, 64, 3), activation
='relu'))
# Step 2 - Pooling
model.add(MaxPooling2D(pool_size = (2,
2)))
# Adding a second convolutional layer
model.add(Conv2D(32, 3, 3, activation
='relu'))
model.add(MaxPooling2D(pool_size = (2,
2)))
# Step 3 - Flattening
model.add(Flatten())
# Step 4 - Full connection
model.add(Dense(64, activation ='relu'))
model.add(Dense(1, activation ='sigmoid'))
# Compiling the CNN
model.compile(optimizer ='adam', loss
='binary_crossentropy', metrics =
['accuracy'])
```

Figure 2: Original Machine Learning Model

Here is the refactored code, replacing found parameters (Figure 3).

```
classifier = Sequential()
classifier.add(Conv2D(params['filters0'],
(int(params['kernel_size2_0']),
int(params['kernel_size2_1'])),
int(params['strides2']), input_shape=(64,
64, 3), activation=params['activation3']))
classifier.add(MaxPooling2D(pool_size=int(
params['pool_size4'])))
classifier.add(Conv2D(params['filters5'],
int(params['kernel_size6']),
int(params['strides7']),
activation=params['activation8']))
classifier.add(MaxPooling2D(pool_size=(int
(params['pool_size10_0']),
int(params['pool_size10_1']))))
classifier.add(Flatten())
classifier.add(Dense(params['units10'],
activation=params['activation11']))
classifier.add(Dense(params['units12'],
activation=params['activation13']))
classifier.compile(optimizer=params['optim
izer14'], loss=params['loss15'],
metrics=['accuracy'])
```

Figure 3: Updated Machine Learning Model

Along with the refactored model, we create an object space dictionary, containing the substituted parameters and their ranges in the format of hyperopt library (see Figure 4).

The refactored code can be used for tuning of the hyperparameters. The biggest value of our method is – it works for an arbitrary machine learning model, providing automatic detection of the parameter space. As you will see in the future sections, the resulting model can be easily tuned using the state of the art Bayesian optimization methods. Unlike other options, our approach does not require any manual steps.

```
space = {
'filters0':hp.choice('filters0',
[32,64,128,256]),
'kernel_size2_0':hp.quniform('kernel_size2
_0', 2.0,4.0, 1),
'kernel_size2_1':hp.quniform('kernel_size2
_1', 2.0,4.0, 1),
'strides2':hp.quniform('strides2',
2.0,4.0, 1),
'activation3':hp.choice('activation3',
["softmax", "elu", "selu", "softplus",
"softsign", "relu", "tanh", "sigmoid",
"hard_sigmoid", "linear"]),
'pool_size4':hp.quniform('pool_size4',
2.0,4.0, 1),
'filters5':hp.choice('filters5',
[32,64,128,256]),
'kernel_size6':hp.quniform('kernel_size6',
2.0,4.0, 1),
'strides7':hp.quniform('strides7',
2.0,4.0, 1),
'activation8':hp.choice('activation8',
["softmax", "elu", "selu", "softplus",
"softsign", "relu", "tanh", "sigmoid",
"hard_sigmoid", "linear"]),
'pool_size10_0':hp.quniform('pool_size10_0
', 2.0,4.0, 1),
'pool_size10_1':hp.quniform('pool_size10_1
', 2.0,4.0, 1),
'units10':hp.choice('units10', [64, 128,
256, 512]),
}
```

Figure 4: Generated hyperparameter space for the model

It is important to mention that parsing and detecting parameters is not always straight forward. Our tool handles all kinds of tricky cases including variable as parameters, object as parameters, inline class creation and class path references. As Figure 5 demonstrates, we parametrize the appropriate.

We can handle tuple parameters as well, since many Keras layers use them (kernel_size, strides, etc). Our tool takes care of all the difficult cases and the generated search space will parametrize the actual place in the code where the parameter is present. For example, in Figure 5, we will parametrize the actual definition of the object ReLU (variable **actv**). Our system will distinguish between constant parameters (passed by value) and object parameters (representing Python classes).

```
model.add(keras.layers.Dense(128,
activation='relu'))
model.add(Dense(128, activation='relu'))
model =
Sequential([Dense(units=24,activation='relu'),.
..])
actv =
keras.layers.ReLU(negative_slope=0.1)
model.add(Dense(128, activation= actv))
model.add(Dense(128, activation=
   ReLU(negative_slope=0.1)))
```

Figure 5: Handling class path, inline objects and variable as parameter in the original model

2.4 Model Tuning

Our tool is using the hyperopt library described in [5] and [6]. It is a state of the art hyperparameter optimization library written by James Bergstra [1]. It has been proven in [2] that randomly chosen parameters are much more efficient than manual or exhaustive grid search. We also would like to avoid a curse of dimensionality [3] since the number of grid search trials is exponential to the number of grid dimensions.

As [6] specifies, hyperopt currently supports three optimization algorithms – Random Search, Tree of Parzen Estimators (TPE) and Adaptive TPE. Our system refactors the original model code to integrate with hyperopt and call its optimization methods.

Our tools implements the following steps:

1. Identify model hyperparameters
2. Parametrize model source code
3. Generate search space
4. Generate a method running the model with passed parameters
5. Call method fmin to find optimal hyperparameters

```
from hyperopt import STATUS_OK, STATUS_FAIL
def modelOptFunction(params=None):
    model = Sequential()
    …
     histName747557 = model.fit
(X_test,y_test,epochs=params['epochs27'],v
erbose=1,validation_split=params['validati
on_split28'],validation_data=None,
shuffle=True)
    return {
        'loss':
histName747557.history['loss'][(len(histNa
me747557.history['loss']) - 1)],
        'status': STATUS_OK,
    }
```

Figure 6: Handling class path, inline objects and variables

Steps 1 through 3 were implemented in Section 2.3. Let's discuss the remaining refactoring steps implemented by our tool. The generated hyperparameter search space (Figure 4) defines the detected parameters. Now we just need to pass them to a function. Our tool generates that function automatically.

Figure 6 demonstrates the generated method, containing our model definition and execution (Step 4). The output of the method, containing status and loss will be used by the hyperopt to choose future parameters for the next round of optimization.

The following Step 5 generates the code that will be running our model and finding the optimal paramaters. Hyperopt provides a method fmin to accomplish that. The method takes the following parameters:

1. Model execution method
2. Search space
3. Optimization algorithm (tpe.suggest, rand.suggest or anneal.suggest)
4. Number of evaluations to perform
5. Trials object keeping the execution history

Our refactoring tool optPar generates the call to fmin automatically.

```
from hyperopt import fmin, Trials
start = datetime.datetime.now()
trials = Trials()
best = fmin(modelOptFunction, space,
algo=tpe.suggest, max_evals=50,
trials=trials)
print ('best: ', best)
print ('trials: ', trials)
outParams(best, space)
end = datetime.datetime.now()
```

Figure 7: Generating the fmin call to tune the parameters

Figure 7 shows the generated code. After the execution, the code returns the optimal hyperparameters along with the timing results. We should also catch the exceptions to make sure our code does not fail if some unexpected error occurs.

2.5 Serverless Cloud Integration

Our tool is fully integrated into the AWS Cloud ecosystem. It can be automatically triggered by placing the model code into S3 bucket. A Lambda function will refactor the model in real time. Cloud integration gives us many more benefits, including access to Monitoring tools and saving results in safe and secure locations for future analytics.

Figure 8 demonstrates the architecture of our system. It provides security, scalability and continuous monitoring of our system via Cloud Trail and Cloud Watch services. Moreover, given increasing popularity of Machine Learning and Data Science, our system will have access to all existing and new AWS cloud services.

We believe that event-triggered serverless architectures will be used more and more in the future. They save money, provide scalability and do not require any idle server time or advanced server procurement. Serverless applications are already widely used enterprise-wide becoming the new IT Industry standard.

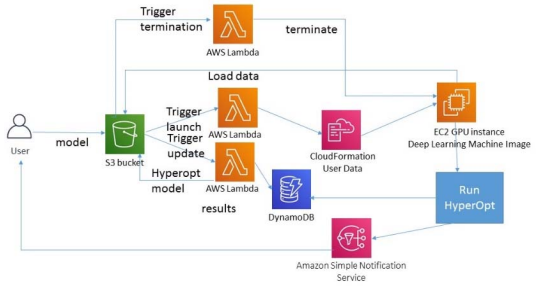

Figure 8: OptPar serverless Architecture

3 Conclusion

Although data science is extremely popular, very few tools attempt to refactor machine learning model code. We have proven that it is possible and that we can accomplish a lot parsing and updating AI models.

Our tool OptPar can automatically refactor an arbitrary machine learning model to tune hyperparameters. It helps to avoid time consuming manual steps and improve Data Scientist productivity. The tool can also be used just for hyperparameters identification, allowing the practitioners to edit detected parameters or utilize other optimization frameworks.

We anticipate that more machine learning refactoring tools will be created in the future, transforming original models or adding some desired features. Moreover, some data science methodologies can be applied to existing models to achieve certain goals. In the end of the day, a Machine Learning Model is code which can be refactored.

REFERENCES

[1] James Bergstra et.al. "Random Search for Hyper-Parameter Optimization", Journal of Ma-chine Learning Research 13, 2012

[2] BERGSTRA, YAMINS, AND COX "Making a Science of Model Search", Proceeding 30th International Conference on Machine Learning, Atlanta, Georgia, USA, 2013

[3] Francis Bach "Breaking the Curse of Dimensionality with Convex Neural Networks", 4/2017, Journal of Machine Learning Research 18 (2017) 1-53

[4] Keras online documentation, https://keras.io/models/about-keras-models/

[5] James Bergstra, Dan Yamins, David D. Cox "Hyperopt: A Python Library for Optimizing the Hyperparameters of Machine Learning Algorithms", THE 12th PYTHON IN SCIENCE CONF, (SCIPY 2013)

[6] Hyperopt github repository, https://github.com/hyperopt/hyperopt