Systematic Approach to Convolutional Neural Network Hyper-Parameter Optimization

Ethan Blagg
UCCS
Colorado Springs, Colorado
eblagg@uccs.edu
https://github.com/ethanblagg/

***Abstract*—Neural network hyper-parameterization is seen by many as a dark art. While the inner workings may not be understood by a network designer, intelligent decisions can be made. A systematic process which informs the design direction will result in a more accurate network, with few unforeseen problems.**

Keywords—machine learning, neural networks, convolution, memory, intelligence, optimization, patterns, categorization, probability, algorithms, data preprocessing, parameterization, hyper-parameterization, tensorflow, keras

I. Introduction

Convolutional neural networks are often used in computer vision. A convolutional neural network combines fully connected layers with convolutional layers, to achieve the task of object recognition and classification [3]. Fully connected layers mimic an animal neocortex. Convolutional layers apply filters to recognize patterns that re-occur in data. Many combinations of these layers exist, depending on the data and goals of the designer. [7]

In order to obtain a network that is accurate, the designer must also choose values for a number of hyper-parameters. Hyper-parameters are parameters which change the design and behavior of a neural network. These include learning rate, number of epochs, batch size, activation function, weights, dropout, and optimizer. There are many other hyper-parameters, however these are some of the most impactful. [8]

Hyper-parameter optimization can seem like a black box when dealing with a new network [4]. Many rules of thumb have been developed to create a starting point for a new designer [6]. Part of the reason rules of thumb are often used is the difficulty in understanding hyper-parameters and why they have the effects that they do. Often, the relationship between hyper-parameters and input data can be quite complex. This paper will discuss a systematic approach to hyper-parameter optimization. While the examples shown here are simple, they can be expanded to include training time, number of epochs, network structure, and more.

II. Problem Statement

While initially setting hyper-parameters by rules of thumb may work well for getting a functional neural network, true optimization takes both time and insight to how hyper-parameters interact with each other. Without a systematic approach, optimization may have unexpected results when changing parameters. For example, a designer may find that increasing parameter A improves the results and that independently decreasing parameter B also improves results. But when A is increase at the same time B is decreased, performance degrades.

This paper will approach 3 hyperparameters, and will run simulations over combinations of them. There is some variance in accuracy of unique neural networks when trained different times with the exact same hyper-parameters. Therefore each model with a set of hyper-parameters will be trained twice, and the final accuracy will be averaged between the two results.

Accuracy can come in many forms for a neural network. Two meaningful metrics for any neural network are loss and accuracy. Loss is a metric which shows how far removed from the correct result a network is, at a certain point in training. It is found by comparing a known correct output to the network's predicted output, for some input. A low number indicates an accurate model.

The second metric, accuracy, is a higher level metric showing the percent of time that the model was correct in its prediction.

The network used for this paper consisted of 3 convolution layers each with a max-pooling layer, followed by 2 fully connected layers with a dropout layer between them. The max-pooling layers server to decrease the resolution of the data flowing through the network, to move towards abstractions which help to classify the object in the image. The dropout layer drops a random number of connections between layers with some specified likelihood. For this layer, pool size was fixed at 2 and dropout rate was fixed at 0.05 for all iterations. The output of the network was set of one-hot logits, where each logit represented a unique category of fruit.

Tensorflow 2.0.0 in Python 3 was the software used to create, train, and test the neural network [9]. Network

training across sets of hyperparameters was done using bash shell scripting, passing hyperparameters values in as options. Results were logged, and data was output to a .csv file for analysis.

Analysis was performed using a Jupyter notebook with matplotlib heavily used. Key parts of the source code are listed in Appendix A. All source code used for this paper is available on Ethan Blagg's Github [2].

The inputs to the model were images of fruit, provided by Mihai Oltean and Horea Muresan via a Github repository [9]. The dataset contained 61,476 images for training, and 20,618 images for testing. There were 120 classes of fruit, many distinguishing varieties of the same fruit. For example, there were separate categories for Golden-Red apples and Pink-Lady apples. Image size was 100x100 pixels.

III. Simulation Results

The hyper-parameters used as simulation variables were learning rate, filter size, and optimizer. Learning rate is the rate at which the network is updated after a batch of training. Larger numbers can increase the speed of training a network, but can prevent a model from ever finding a "stable solution". Smaller number can slow the training speed, and can cause a model to become trapped in a "local minimum", preventing it from finding an ideal solution.

Filter size indicates the initial filter size for a convolutional layer. If filter size is F, then the actual filter is F x F. Additionally, F is the size of the filter in the first convolutional layer. Filter size for layer 2 is 2 x F, and filter size for layer 3 is 4 x F.

The optimizer variable is which optimization algorithm was used. Each choice uses the loss output of the network to "back-propagate" through the network and update its weights. The choices were Poisson, Categorical Cross Entropy, and Mean Squared Error.

Figures 1 and 2 display a rough idea of how hyper-parameters effect a model's accuracy and loss. Note that there is a clustering of good results below the 0.01 learning rate.

In order to better distinguish the results within this clustering, we can drop the 0.01 hyper-parameter from the plot. Now note the difference in the worst case results according to the legend, between Figure 1 and Figure 3.

With very little effort, we can trim hyper-parameters until we get a plot as seen in Figure 4. A similar process can be executed for the loss metric. This process provides a rapid and systematic process for understanding how multiple hyper-parameters interact with each other. The designer can immediately "trim the fat" in certain areas. The hyper-parameter choice for a network can be made from a refined set.

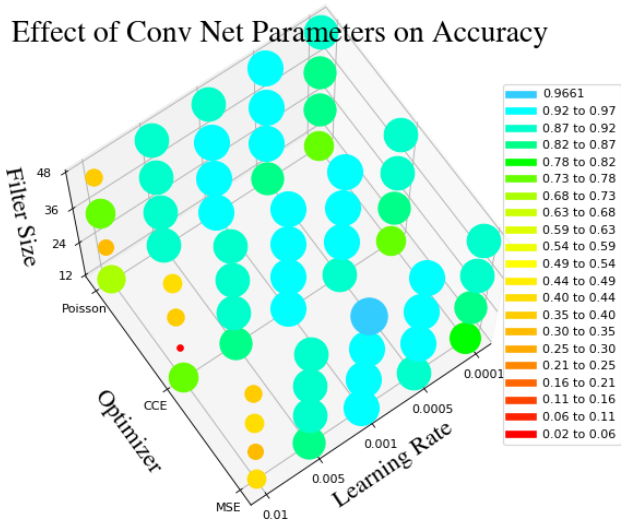

FIGURE 1. VISUALIZATION OF HYPER-PARAMETER EFFECT

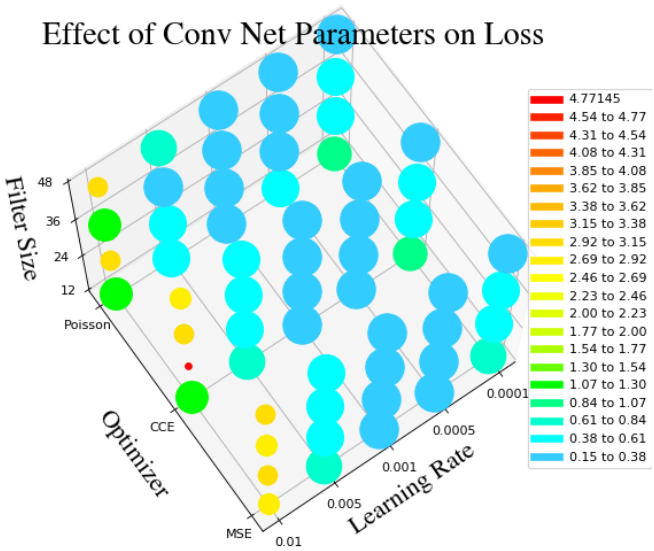

FIGURE 2. VISUALIZATION OF HYPER-PARAMETER EFFECT

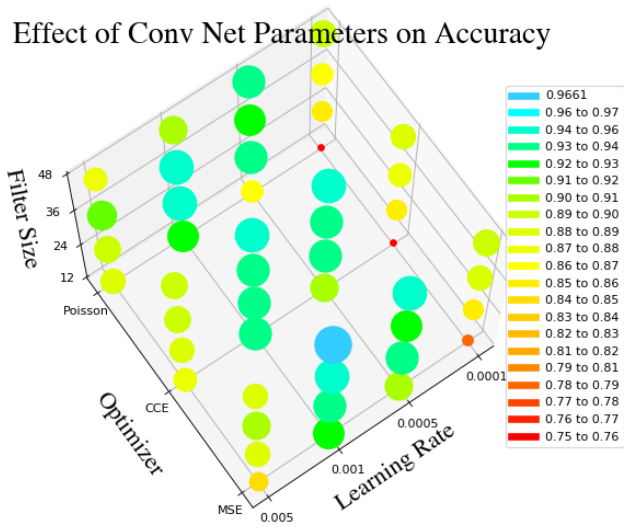

FIGURE 3. TUNING THE VISUALIZATION

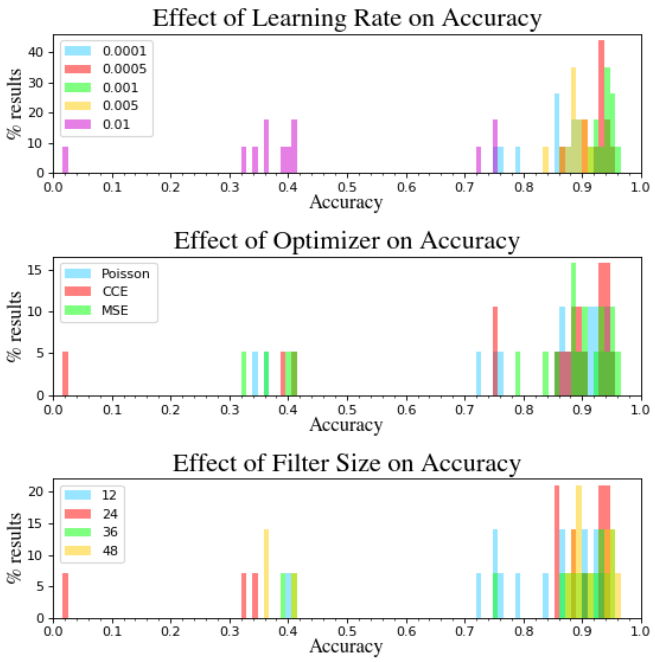

FIGURE 5. COMPARISON OF DIFFERENT VALUES OF A HYPER-PARAMETER

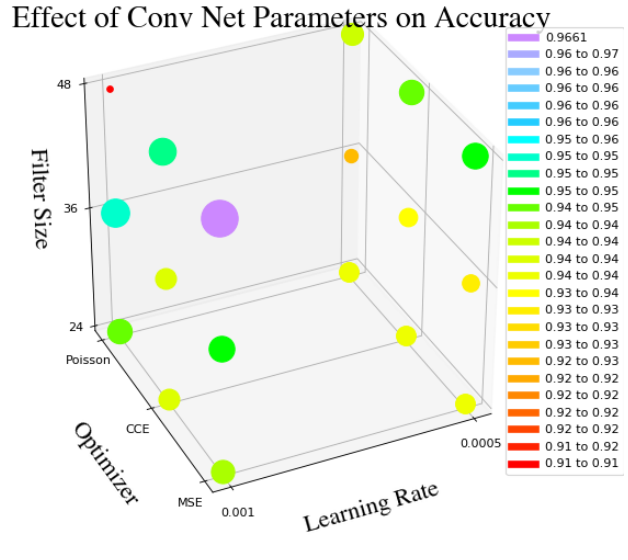

FIGURE 4. FINE TUNING OVER MULTIPLE DIMENSIONS

Another way to use visualization to better understand the effect of hyper-parameters is to display data with histograms. Figures 5 & 6 show the results for accuracy and loss, when a hyper-parameter is held constant. Each color on a plot represents a constant value for a hyper-parameter, and the percentage of results for an accuracy or loss value is represented by a bar at that value.

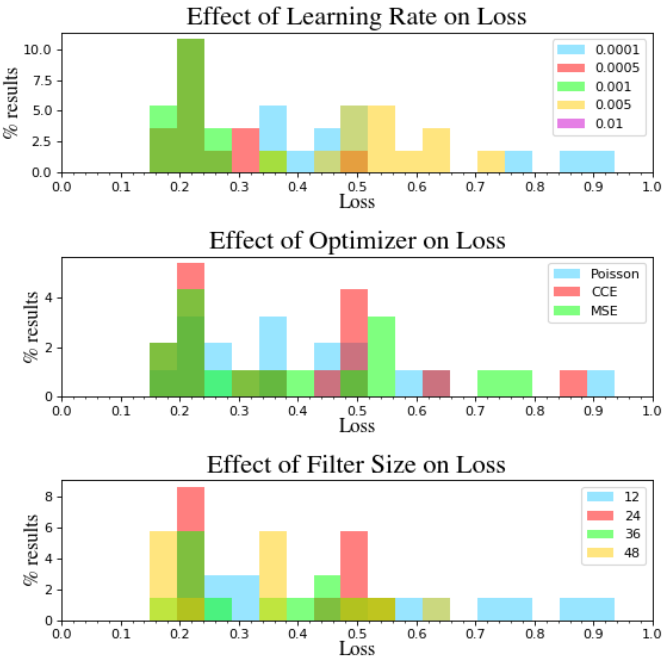

FIGURE 6. COMPARISON OF DIFFERENT VALUES OF A HYPER-PARAMETER

Again, by trimming the results based on the 3d plots, we can quickly gain insight into which hyper-parameters have the best results. A trimmed plot for accuracy is shown below, using the same trimming as in Figure 4.

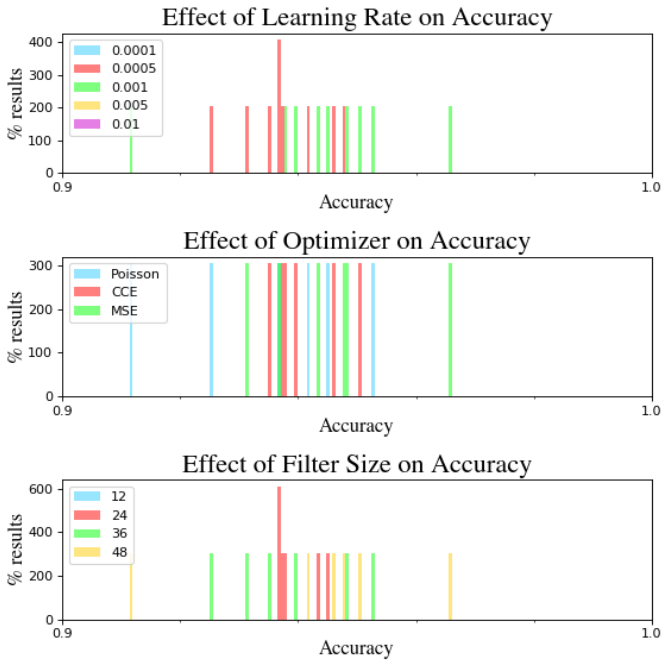

FIGURE 7. COMPARISON OF DIFFERENT VALUES OF A HYPER-PARAMETER

CONCLUSION

Through the use of a systematic process, a data scientist can tune neural network hyper-parameters for better results. While this process can take time in the data gathering stage, the results can ensure the network designer creates a network that bets meets the prescribed needs of the task.

A further development of the methods shown in this paper would likely include a comparison of learning rate schedules, and network layer order and organization [5]. A comparison of number of epochs agains learning rate will also likely have large impact on the final result of a network.

REFERENCES

1. R. Bartyzal, "Multi-label image classification with Inception net," Medium, 16-Apr-2017. [Online]. Available: https://towardsdatascience.com/multi-label-image-classification-with-inception-net-cbb2ee538e30. [Accessed: 29-Feb-2020].
2. E. Blagg, "ConvNetHyperParameterOptimization," GitHub. [Online]. Available: https://github.com/ethanblagg/ConvNetHyperParameterOptimization. [Accessed: 04-May-2020].
3. J. Hawkins and S. Blakeslee, On intelligence: New York: Times Books/Henry Holt, 2008.
4. J. Heaton, "Applications of Deep Neural Networks," Playlist: Applications of Deep Neural Networks, 31-Aug-2016. [Online]. Available: https://www.youtube.com/watch?v=gfOcjfIc0Nc&list=PLjy4p-07OYzt4467lXAD6Fxexg-UUHfZT.
5. C. Olah, "Conv Nets: A Modular Perspective," Conv Nets: A Modular Perspective - colah's blog, 18-Jul-2014. [Online]. Available: https://colah.github.io/posts/2014-07-Conv-Nets-Modular/. [Accessed: 29-Feb-2020].
6. C. Ranjan, "Rules-of-thumb for building a Neural Network," Medium, 02-Aug-2019. [Online]. Available: https://towardsdatascience.com/17-rules-of-thumb-for-building-a-neural-network-93356f9930af. [Accessed: 26-Mar-2020].
7. S. Lau, "A Walkthrough of Convolutional Neural Network - Hyperparameter Tuning," Medium, 10-Jul-2017. [Online]. Available: https://towardsdatascience.com/a-walkthrough-of-convolutional-neural-network-7f474f91d7bd. [Accessed: 04-May-2020].
8. P. Radhakrishnan, "What are Hyperparameters ? and How to tune the Hyperparameters in a Deep Neural Network?," Medium, 18-Oct-2017. [Online]. Available: https://towardsdatascience.com/what-are-hyperparameters-and-how-to-tune-the-hyperparameters-in-a-deep-neural-network-d0604917584a. [Accessed: 04-May-2020].
9. "TensorFlow," TensorFlow. [Online]. Available: https://www.tensorflow.org/. [Accessed: 26-Mar-2020].
10. Horea94, "Fruit-Images-Dataset," GitHub, 30-Apr-2020. [Online]. Available: https://github.com/Horea94/Fruit-Images-Dataset. [Accessed: 04-May-2020].

APPENDIX A: RELEVANT SOURCE CODE

All source code used for this paper is available at https://github.com/ethanblagg/ConvNetHyperParameterOptimization

Main tensorflow body:

```
import tensorflow as tf
from Ethanet_0 import Ethanet
import datetime
import os
import numpy as np
import argparse
import pathlib                      # glob

# ==================== Fetch args ========================================
parser = argparse.ArgumentParser(description='Provide optional inputs for scripting')

parser.add_argument('-r', '--learning_rate', type=float, default = 3e-3, help='Learning Rate for training')
parser.add_argument('-e', '--epochs', type=int, default=1, help='Number of training Epochs')
parser.add_argument('-b', '--batch_size', type=int, default=32, help='Batch Size')
parser.add_argument('-f', '--init_filter_size', type=int, default=12, help='First Conv Layer Filter Size')
parser.add_argument('-v', '--verbose', action='store_true', default=False, help='Print verbose information')

loss_group = parser.add_mutually_exclusive_group()
loss_group.add_argument('-p', '--poisson', action='store_true')
loss_group.add_argument('-c', '--categorical_cross_entropy', action='store_true')
loss_group.add_argument('-m', '--mean_squared_error', action='store_true')

args = parser.parse_args();          # x & y lengths for images to be resized to\



# ==================== Constants / Variables ============================
if args.poisson:
    loss_type = tf.keras.losses.Poisson()
elif args.categorical_cross_entropy:
    loss_type = tf.keras.losses.CategoricalCrossentropy()
elif args.mean_squared_error:
    loss_type = tf.keras.losses.MeanSquaredError()

learning_rate = args.learning_rate
num_epochs = args.epochs
batch_size = args.batch_size
loss = loss_type = tf.keras.losses.CategoricalCrossentropy()
filt_size = args.init_filter_size
verbose = args.verbose
image_size = 100
IMG_WIDTH = image_size
IMG_HEIGHT = image_size

param_str = "_lr={}_e={}".format(learning_rate, num_epochs)
model_time_identifier = datetime.datetime.now().strftime("%Y-%m-%d--%H-%M-%S") + param_str
log_dir = '../log/' + model_time_identifier                  # Directory where logs are saved
data_dir = '../data/'
train_data_dir_str = data_dir + 'training_data_fruit/'
test_data_dir_str = data_dir + 'testing_data_fruit/'
train_cache_str = data_dir + 'training_data_cache.tfcache'
test_cache_str = data_dir + 'testing_data_cache.tfcache'

train_data_dir = pathlib.Path(train_data_dir_str)
test_data_dir = pathlib.Path(test_data_dir_str)

# ==================== Setup ============================================
print('\nRunning in {}'.format(os.getcwd()))
print('Log identifier:  {}'.format(model_time_identifier))
print('Training epochs: {}'.format(num_epochs))
print('Learning rate:   {}'.format(learning_rate))
print('')


def get_label(file_path):
    # convert the path to a list of path components
    parts = tf.strings.split(file_path, os.path.sep)
    # The second to last is the class-directory
    return parts[-2] == CLASS_NAMES

def decode_img(img):
    # convert the compressed string to a 3D uint8 tensor
    img = tf.image.decode_jpeg(img, channels=3)
    # Use `convert_image_dtype` to convert to floats in the [0,1] range.
    img = tf.image.convert_image_dtype(img, tf.float32)
    # resize the image to the desired size.
    return tf.image.resize(img, [IMG_WIDTH, IMG_HEIGHT])

def process_path(file_path):
    label = get_label(file_path)
    # load the raw data from the file as a string
    img = tf.io.read_file(file_path)
```

```
    img = decode_img(img)
    return img, label


def prepare_for_training(ds, cache=True, shuffle_buffer_size=1000):
    # This is a small dataset, only load it once, and keep it in memory.
    # use `.cache(filename)` to cache preprocessing work for datasets that don't
    # fit in memory.
    if cache:
        if isinstance(cache, str):
            ds = ds.cache(cache)
        else:
            ds = ds.cache()

    ds = ds.shuffle(buffer_size=shuffle_buffer_size)

    # Repeat forever
    #ds = ds.repeat()

    ds = ds.batch(batch_size)

    # `prefetch` lets the dataset fetch batches in the background while the model
    # is training.
    ds = ds.prefetch(buffer_size=tf.data.experimental.AUTOTUNE)

    return ds


CLASS_NAMES = np.array([item.name for item in train_data_dir.glob('*') if item.name != "LICENSE.txt"])
if verbose:
    print(CLASS_NAMES)
    print(len(CLASS_NAMES))


train_list_ds = tf.data.Dataset.list_files(train_data_dir_str + '*/*')
test_list_ds = tf.data.Dataset.list_files(test_data_dir_str + '*/*')
if verbose:
    for f in train_list_ds.take(5):
        print(f.numpy())

    for f in test_list_ds.take(5):
        print(f.numpy())

print("Number of training records: ", tf.data.experimental.cardinality(train_list_ds))

# Set `num_parallel_calls` so multiple images are loaded/processed in parallel.

train_labeled_ds = train_list_ds.map(process_path, num_parallel_calls=tf.data.experimental.AUTOTUNE)
test_labeled_ds  = test_list_ds.map(process_path, num_parallel_calls=tf.data.experimental.AUTOTUNE)
train_ds = prepare_for_training(train_labeled_ds, cache=train_cache_str)
test_ds = prepare_for_training(test_labeled_ds, cache=test_cache_str)


if verbose:
    for image, label in train_labeled_ds.take(1):
        print("Image shape: ", image.numpy().shape)
        print("Label: ", label.numpy())


print('\nCompiling Ethanet')
model = Ethanet(filt_size=filt_size, num_logits=len(CLASS_NAMES))                                       # Create the model
model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=learning_rate),                             # Compile the model
              loss=tf.keras.losses.CategoricalCrossentropy(from_logits='true'),
              metrics=['accuracy'])

print('Connecting to Tensorboard')
tensorboard_cbk = tf.keras.callbacks.TensorBoard(log_dir=log_dir,                    # tensorboard callback
                                                 histogram_freq=1,                  # How often to log histogram visualizations
                                                 embeddings_freq=1,                 # How often to log embedding visualizations
                                                 update_freq='epoch')               # How often to write logs (default: once per epoch)

print('Training model')
history = model.fit(train_ds, #train_ds.batch(batch_size),                                                  # train the model
                    epochs=num_epochs,
                    #validation_data=,
                    shuffle=True,
                    callbacks=[tensorboard_cbk])


print('Evaluating model')
print('final_output_line:  ', end='')
model.evaluate(test_ds,
               callbacks=[tensorboard_cbk])


print('\n\n\n')
```