Idiomatic Programmer

Learning Keras

Handbook 3: Computer Vision Training and Deployment

Andrew Ferlitsch, Google AI

Version August 2019

The Idiomatic Programmer - Learning Keras

Handbook 3 - Computer Vision Training and Deployment

Part 10 - Computer Vision Training Preparation and Hyperparameters
Part 11 - Computer Vision Training and Deployment
Part 12 - Transfer Learning & Prebuilt Models
Part 13 - Production Environment

Part 10 - Training Preparation and Hyperparameters

As a Googler, one of my duties is to educate software engineers on how to use machine learning. I already had experience creating online tutorials, meetups, conference presentations, and coursework for coding school, but I am always looking for new ways to effectively teach.

Welcome to my latest approach, the idiomatic programmer. My audience are software engineers who are proficient in non-AI frameworks, such as Angular, React, Django, etc. You should know at least the basics of Python. It's okay if you still struggle with what is a compression, what is a generator; you still have some confusion with the weird multi-dimensional array slicing, and this thing about which objects are mutable and non-mutable on the heap. For this tutorial it’s okay.

You have a desire (or requirement) to become a machine learning engineer. What does that mean? A machine learning engineer (MLE) is an applied engineer. You don't need to know statistics (really you don't!), you don't need to know computational theory. If you fell asleep in your college calculus class on what a derivative is, that's okay, and if somebody asks you to do a dot product between two matrices you'd look them in the eyes and say why?

Your job is to learn the knobs and levers of a framework, and apply your skills and experience to produce solutions for real world problems. That's what I am going to help you with.

Overview

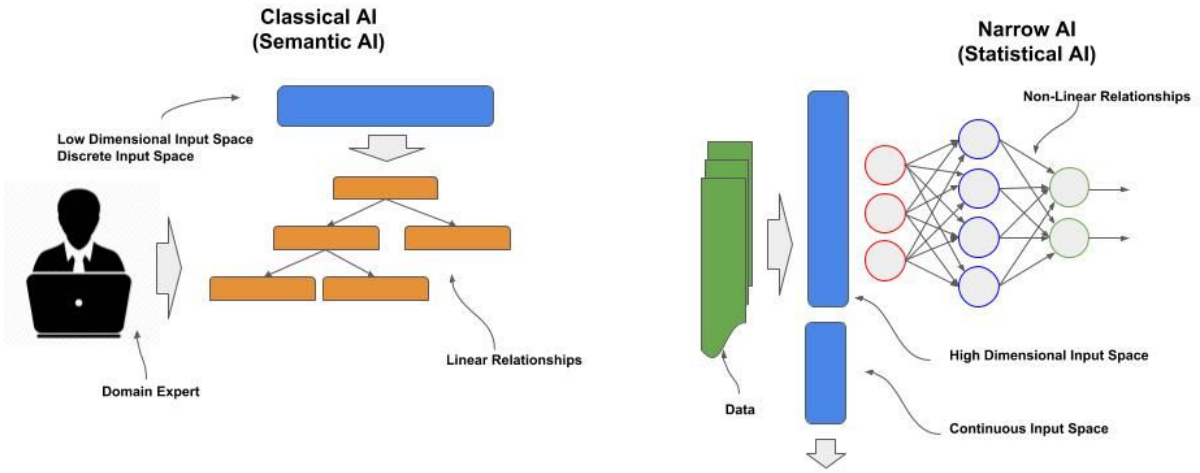
In this part, we will cover best practices to prepare for training and hyperparameter tuning for a computer vision (image recognition) model.

Let’s briefly revisit what a model is. Prior to machine learning, software developers designed and coded algorithms, incorporating theirs and others’ domain expertise for the problem the algorithm will solve. The algorithm was primarily based on human engineering, with some machine assistance. The types of algorithms humans design can be characterized as:

- Having a low dimensional input space,
- Discrete input space with finite state changes, and
- A linear relationship between the output behaviors and the input space.

A model is an algorithm discovered by machine learning, which can be characterized as:

- Having high dimensional input space,
- A continuous input space, and
- A non-linear relationship between the output behaviors and the input space.



So what does this mean?

High (vs. Low) Dimensionality

Humans are very good at solving problems in low dimensional space. Low dimensionality is when there are just a few distinct inputs. For example, classifying an image as a polygon shape (e.g., triangle, rectangle, etc) using just two features: points and lines. Let's start with perfect shapes in black on a white background. As an engineer, you would say the pixel value would determine if that pixel is a point (i.e., black). You would then write a set of rules looking at adjacent pixels, and if an adjacent pixel is black, then that's part of a line. When you're done scanning through the pixels, you code another set of rules for classifying the shape by counting the number of lines, the order in which they are connected and length of the line.

Wow, that was easy! Now let's say that the lines can be different shades of gray and the background may not be perfectly white. This becomes a third feature. As an engineer, you select a threshold where any pixel above or below the value is classified as black or white. That wasn't too hard. Let's introduce a fourth feature and say the polygons are hand-drawn and can be a little less than perfect. Now things become complicated in that you have to modify the adjacency rules for lines and line length being some threshold within what was otherwise perfect. Let's introduce a fifth feature in that the shape can be any color and the background any

different color. As an engineer, you add a preprocessing step to find the mode pixel value and a threshold range around the value, and you call that the background color.

By now you're not correctly recognizing 100% of the time, but you're getting 98.5% and you feel it's good enough. Let's now add in more features, such as an arbitrary background, endpoints of lines having some separation, the lines being of different colors, the polygon having different orientations. At some point, no matter how many rules you code, your accuracy will plummet. Why? At some point, you've introduced nonlinearities that exceed your ability to write rules for exceptions, edge cases, etc and you start overfitting to these special cases, which undos the generalization.

High dimensionality is when the number of inputs exceeds our ability to abstract how these inputs collectively contribute to a solution. As in our extremely simple problem above, once we go to a high number of features, it would be impossible for an engineer to reliably code a set of rules. Prior to deep learning, this is what engineers did and we referred to these as expert systems (aka business intelligence). But they were costly to build, constantly needed updating/tuning, and generally only achieved at most 70% reliability.

Continuous (vs. Discrete) Input Space

Humans are very good at solving problems when the input space is discrete. A discrete input space is when we have a fixed number of features, such as was described in our polygon recognition problem. A continuous input space is when the input space does not consist of a fixed number of features, but an unbounded set. In other words, when we cannot predetermine the features.

Let's take fruit as an example. You probably could come up with an exhaustive list of types of fruits and their varieties, their shapes, textures and colors. But this is actually a continuous input space. Let's just simply start with the fact that growers continuously breed (engineer) new varieties, which you could not anticipate. Let's consider all the environmental conditions that the image may appear in: a kitchen, a garden, a commercial field, a warehouse, a grocery store, a kid's school lunch --at some point, you could not anticipate all the environments. Let's consider all the conditions of the fruit that may appear in the image:flowering stage, ripening stage, sliced, prepared, rottening, insect damaged, diseased, partially eaten --at some point, you could not anticipate all the conditions.

Non-Linearity

Before machine learning, most algorithms solved problems that were either all or very close to having some form of a linear relationship between the inputs and outputs. Classical use of linear and logistic regression and decision trees were used on problems that had low dimensionality in the input space and a high level of linearity, where the linearity could be expressed as a polynomial equation.

Algorithms which have high nonlinearity cannot be expressed as a polynomial equation. Instead, the algorithm needs to be decomposed into segments of linearity and then combined through activation functions. With large datasets and an increase in computing power, deep learning methods can be used to train a model to learn this decomposition and relationships (activations) between within the decomposition, which are typically represented as filters, weights and biases.

Training Steps

When training a model, the following steps are typically followed:

1. Split the dataset into training, evaluation (validation) and test sets.
2. Repeat
    a. Set hyperparameters such as batch size and learning rate.
    b. Repeat:
        i. Shuffle the training data.
        ii. Feed the training data through the neural network (epoch).
        iii. Measure accuracy of the model on the train and eval data per epoch.
        iv. Monitor for convergence.
    c. If targets for rate of convergence and accuracy are sufficient, stop training.
3. Measure final accuracy with the test data.

Let's now explore these steps in greater detail.

Splitting the Dataset

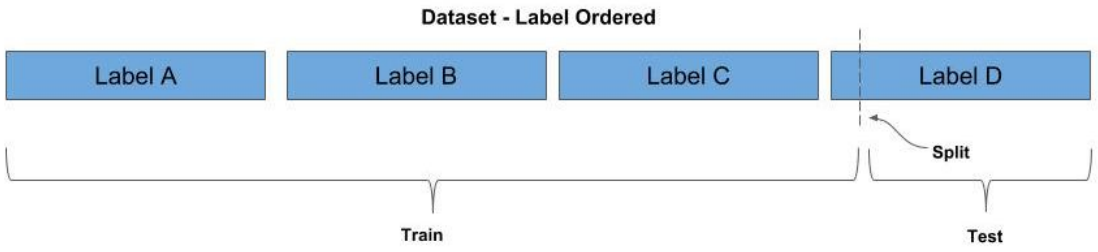
Prior to training the model, the dataset needs to be split into training and test data, where a larger portion of the dataset will be used only for training, and a smaller portion only for testing. The latter, test, is sometimes referred to as the holdout set.

In some cases, a portion of the training data is further split off into an evaluation or eval dataset, which is sometimes referred to as validation data. When the training data is large, typically a prior split is made for the eval data. The same eval data is then used at the end of each epoch to estimate what the accuracy will be on the test data (to be discussed further down); thus the eval data is never used as part of the training. When the training data is small, a different random portion of the training data is used for validation per epoch, which is referred to as cross-validation or rotation-estimation. In this case, eval data is used during training, but not on the specific epoch that it is used to estimate the accuracy on the test data.

Shuffling

When a dataset is split, both the training and test data should have the same probability distribution of the labels (classes). For example, if 10% of the data is of label A, then both the training and test data should contain 10% of label A.

A common mistake is to make an arbitrary split of the dataset. It's not uncommon for datasets to be sequentially ordered by labels. For example, consider a dataset where all the data for label A comes first, then followed by label B, then label C, and so forth. If you made an arbitrary split, say the first 80% of the data is training and the last 20% is test, the training and test will not have the same probability distribution. Worse yet, it's possible that some labels in the test data won't appear in the training data (not learned) and some labels in training data won't appear in test (not verified). The example below depicts this situation.



Example of a bad split between training and test

The common practice is to randomly shuffle the dataset prior to splitting. If the dataset is sufficiently large, a random shuffle should produce a probability distribution of the labels that is the same in the training and test data.

When the dataset is too small to maintain an equal probability distribution between train and test with a random shuffle, it is a common practice to stratify the data prior to randomly shuffling. In this case, the data is partitioned into bins by label (stratify), and then random selections are made from the bin according to the probability distribution for the training data. The remaining unselected data becomes the test data.

In-Memory

If your [preprocessed] image data is in-memory and stored either in a Python list or numpy array, the image data can be randomly shuffled efficiently. The code examples below demonstrate how to randomly shuffle a Python list.

In this first example, the image data (x) and corresponding label (y) are represented as tuples:

```
import random
# assume the dataset appears like [ (x,y), (x,y),...], where x is the image data
# and y is the corresponding label

# randomly shuffle the elements in the list dataset in place
random.shuffle(dataset)
```

In this second example, the image data (x) and corresponding label (y) are represented by separate lists. Note that in this case, we use a `random.seed()`. Since both the x and y lists will be randomly shuffled separately, we need both to have the same random sequence to align. We accomplish this using the same seed prior to shuffling the x and y data.

```
import random
# assume the dataset appears as [x, x, x, x] and [y, y, y, y], where x is the image
# data and y is the corresponding label

# pick an arbitrary number for a seed
seed = 101

# seed the random sequence and shuffle the x (image) data
random.seed(seed)
random.shuffle(x_data)

# reset the same seed to get the identical random sequence and shuffle the y
# (label) data
random.seed(seed)
random.shuffle(y_data)
```

The code examples below demonstrate how to randomly shuffle a numpy multi-dimensional array, when the image and corresponding labels are paired as tuples:

```
import numpy as np
# assume the dataset appears like [ (x,y), (x,y),...], where x is the image data
# and y is the corresponding label

# randomly shuffle the elements in the list dataset
np.random.shuffle(dataset)
```

The code examples below demonstrate how to randomly shuffle a numpy multi-dimensional array, when the image and corresponding labels are in separate arrays:

```
import numpy as np
# assume the dataset appears as [x, x, x, x] and [y, y, y, y], where x is the image
# data and y is the corresponding label

# pick an arbitrary number for a seed
seed = 101

# seed the random sequence and shuffle the x (image) data
np.random.seed(seed)
np.random.shuffle(x_data)

# reset the same seed to get the identical random sequence and shuffle the y
# (label) data
np.random.seed(seed)
np.random.shuffle(y_data)
```

File Listing

If your processed image data is in a file listing, like a CSV or JSON format, you can make an indirect index in memory to the file listing, and then randomly shuffle the in-memory index. The code below demonstrates using an indirect index to randomly shuffle the dataset as a file listing in a CSV file.

```
import random

# open the CSV file and count (using sum) the number of lines, which equals the
# number of samples
with open(csv_file) as f:
        nimages  = sum(1 for line in f)
        # subtract one from the total count if the first line in CSV file is a
header
        if header:
                nimages -= 1

# create a sequential index between 0 and nimages-1
index = [i for i in range(nimages)]

# now randomly sort the index
random.shuffle(index)
```

The code below demonstrates using an indirect index to randomly shuffle the dataset as a file listing in a JSON file, where the files are a list ([]) of objects, and each object has the key ‘image’.

```
dataset = json.load(json_file)
nimages = dataset['image']

# create a sequential index between 0 and nimages-1
index = [i for i in range(nimages)]

# now randomly sort the index
random.shuffle(index)
```

Keras ImageDataGenerator

In **Keras**, an image dataset can be shuffled and split into training and eval (validation) with the `ImageDataGenerator` class. This class is used to ingest a dataset for feeding a neural network during training. In the code example below:

1. The variable `model` refers to a Keras model that has already been compiled.
2. `x_train` and `y_train` refers to a processed dataset for training (i.e., already split), where `x_train` typically is a numpy array, with each element a preprocessed image, and `y_train` typically is a numpy array, with each element the corresponding one-hot encoded label.
3. The variable `datagen` is a generator, which is instantiated by `ImageDataGenerator` for feeding the neural network.
4. The method `flow()` sequentially moves through the training data (`x_train`, `y_train`) in a specified batch size, which is specified as 32 in this example.
5. The parameter `shuffle` is set to `True`, which causes the training data to be shuffled by the generator, at the beginning of each epoch.
6. Each batch of training data is fed through the neural network using the model's `fit()` method.

```
from keras.preprocessing.image import ImageDataGenerator

# x_train and y_train assume the image data and labels have been preprocessed and
# split into training and test data.

# instantiate an Image Data generator object
datagen = ImageDataGenerator()

# the number of batches in an epoch
nbatches = len(x_train) // 32
# the number of epochs (training passes over the entire training data)
epochs = 10

for epoch in range(epochs):
```

```
    batches = 0
    # Use generator to create batches
    for x_batch, y_batch in datagen.flow(x_train, y_train, batch_size=32,
                                         shuffle=True):
        model.fit(x_batch, y_batch)
        batches += 1
        if batches == nbatches:
            break
```

In the code below, unlike above, the `x_train` data has not been normalized/standardized. In this case, the parameter `rescale` is passed to the instantiation of `ImageDataGenerator`, which will result in the corresponding `flow()` method normalizing the pixel data (i.e., dividing by 255) while the data is being fed:

```
from keras.preprocessing.image import ImageDataGenerator

# x_train and y_train assume the image data and labels have been resized for the
# CNN and split into training and test data, but the data has not been normalized.

# instantiate an Image Data generator object and specify normalizing the image data
datagen = ImageDataGenerator(rescale=1./255)

# feed (train) the neural network
```

In the code below, the `rescale` parameter is replaced with `featurewise_std_normalization`, which will standardize the data. Since standardization requires calculating the mean and standard deviation of the pixel data across the entire training set, a call is made to the `fit()` method for the corresponding calculation.

```
from keras.preprocessing.image import ImageDataGenerator

# instantiate an Image Data generator object and specify standardizing the image
# data
datagen = ImageDataGenerator(featurewise_std_normalization=True)

# calculate the mean/stddev for standardization
datagen.fit(x_train)

# feed (train) the neural network
```

In the code below, the parameter validation_split is added to use 10% of the training data as validation data on each epoch. Additionally, we do a single call for training per epoch (vs. manually batch feeding it), by passing the `datagen.flow()` generator to the `fit_generator()`:

```
from keras.preprocessing.image import ImageDataGenerator

x_train and y_train assume the image data and labels have been resized for the
# CNN and split into training and test data, but the data has not been normalized.

# instantiate an Image Data generator object
datagen = ImageDataGenerator(rescale=1./255, validation_split=0.1)

# the number of epochs (training passes over the entire training data)
epochs = 10

for epoch in range(epochs):
    # Use generator to create batches
    model.fit_generator(datagen.flow(x_data, y_data, batch_size=32, shuffle=True))
```

If the dataset is smaller, such as under a thousand, a random shuffle is less likely to produce an equal distribution of the classes (labels) between the training and test data. In these cases, the common practice is to stratify the distribution of the data. In stratification, one maintains data grouped by label, but randomly draws from each group in proportion to its distribution when creating a batch. For example, if the batch size is 32, and label A represents 25% and label B represents 75%, then a stratification method would draw at random 8 samples of label A and 24 samples of label B.

Split Percentages

When training a neural network, one splits the processed image dataset into training data and test data. That is, a portion of the preprocessed image dataset is set aside to test the accuracy of the trained model, and is not used as part of the training. Generally only a small percentage should be set aside, and the larger the image dataset, the smaller that percentage can be. Below is a general rule of thumb of the percentage to set aside for test data, based on the number of images in the dataset:

Less than	1000	20%
Less than	10000	10%
Greater than	10000	5%

The code example below splits a dataset into 80% training and 20% test using `numpy`. In this example, the dataset is presumed to be randomly shuffled. A `pivot` point in the dataset is calculated where all the elements prior to the pivot are training data and all the elements after the pivot are test.

```
import random
import numpy as np

# Make a fake dataset of 100 samples of 10 classes (labels)
# x are the images
x = [[_,_] for _ in range(100)]
# y are the corresponding labels, randomly chosen
y = [ random.randint(1,10) for _ in range(100)]

# percent of dataset for training
percent = 0.2

# find the pivot point in dataset to split
pivot = int(len(x) * (1 - percent))

# presumed to be randomly shuffled
x_train = x[0:pivot]
y_train = y[0:pivot]
x_test  = x[pivot:]
y_test  = y[pivot:]

print("Train", len(x_train), len(y_train)) # will output Train 80 80
print("Test ", len(x_test),  len(y_test))  # will output Test  20 20
```

The code example below splits a dataset into 80% training and 20% test using the `train_test_split()` method from **scikit-learn**. In this example, with the parameter `shuffle` set to `True`, the dataset will be randomly shuffled prior to splitting.

```
import random
from sklearn.model_selection import train_test_split

# Make a fake dataset of 100 samples of 10 classes (labels)
# x are the images
x = [[_,_] for _ in range(100)]
# y are the corresponding labels, randomly chosen
y = [ random.randint(1,10) for _ in range(100)]

# Split the dataset (x, y) into 80% training and 20% test
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2,
                                                    shuffle=True)

print("Train", len(x_train), len(y_train)) # will output Train 80 80
print("Test ", len(x_test),  len(y_test))  # will output Test  20 20
```

In the code example below using **Keras,** we specify the percentage split between training and

evaluation data at the time we instantiate the `ImageDataGenerator` object, while the actual shuffle and split is deferred until feeding.

```
# instantiate an Image Data generator object
# and preset splitting the data into 80% train and 20% test
datagen = ImageDataGenerator(validation_split=0.2)
# ...
```

Hyperparameter Tuning

Let's start by explaining the difference between learned parameters and hyperparameters. The learned parameters are parameters that are learned during training. For neural networks, these typically are the weights on each neural network connection, the biases on each node, and for convolutional neural networks, the filters in each convolutional layer. These learned parameters stay as part of the model when the model is done training.

Hyperparameters are parameters used to train the model, but not part of the trained model itself. That is, once trained the hyperparameters no longer exist. Hyperparameters are used to improve the training of the model, such as for:

1. How long does it take to train the model?
2. How fast does the model converge?
3. Does it find the global optima?
4. How accurate is the model?
5. How overfitted is the model?

Another perspective of hyperparameters is that they are a means to measure cost and quality of developing the model. We will cover the above and other questions as we go more into the hyperparameters.

Epochs

The most basic hyperparameter is the number of epochs --though this is now being more commonly replaced with steps. The epochs hyperparameter is the number of times you will pass the entire training data through the neural network during training. Training is very expensive in compute time. It includes both the forward feed to pass the training data through and the backward propagation to update (train) the model's parameters. For example, if a full pass of the data (epoch) takes 15 minutes and we run 100 epochs, the training time will take 25 hrs.

Early presumptions on training were that the more times you feed the training data into the model, the better the accuracy. What we've found, particularly on larger and more complex networks, is that there is a point where the accuracy will degrade. Today, we now look for

convergence on an acceptable local optima for the purpose of how the model will be used in an application. If we overtrain the neural network, the following can happen:

1. The neural network becomes overfitted to the training data, showing increasing accuracy on the training data, but degrading accuracy on the test data.
2. In deeper neural networks, the layers will learn in a non-uniform manner and have different convergence rates. Thus, as some layers are working towards convergence, others may have convergence and thus start diverging.
3. Continued training may cause the neural network to pop out of one local optima and start converging on another local optima that is less accurate.

It's understandable if you're struggling with the concepts of convergence and optimas -- we will gradually go into more detail as we discuss hyperparameters.

Let's start with a simple convnet model in Keras using the CIFAR-10 dataset to demonstrate the concept of convergence and then diverging. In the code below, we have intentionally left out methods which prevent overfitting, like dropout or batch normalization.

```
import keras
from keras import Sequential
from keras.layers import Conv2D, MaxPooling2D, Dropout, Flatten, Dense
from keras.datasets import cifar10

# load the Keras builtin CIFAR-10 dataset
(x_train, y_train), (x_test, y_test) = cifar10.load_data()

# Get the shape of each image (should be 32x32)
height = x_train.shape[1]
width  = x_train.shape[2]

# Next we need to normalize the pixel data
x_train = x_train / 255.0
x_test  = x_test  / 255.0

# Next we need to one-hot encode the labels
num_classes = 10
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test  = keras.utils.to_categorical(y_test,  num_classes)

# Our simple CovNet model
model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3),
                 activation='relu',
                 input_shape=(height, width, 3)))
model.add(Conv2D(64, (3, 3), activation='relu'))
```

```
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dense(num_classes, activation='softmax'))

model.compile(loss=keras.losses.categorical_crossentropy,
              optimizer=keras.optimizers.Adadelta(),
              metrics=['accuracy'])

# Train the model
epochs=20
batch_size=32
model.fit(x_train, y_train,
          batch_size=batch_size,
          epochs=epochs,
          verbose=1,
          validation_data=(x_test, y_test))
```

Below are the stats for the first six epochs. You can see with each pass there is a steady reduction in loss, which means the neural network is getting closer to fitting the data. Additionally, the accuracy on the training data is going up from 48.75% to 88.3% and on the validation data from 61.26% to 69.23%.

```
Train on 50000 samples, validate on 10000 samples
Epoch 1/20
50000/50000 [==============================] - 61s 1ms/step - loss: 1.4311 - acc:
0.4875 - val_loss: 1.1157 - val_acc: 0.6126
Epoch 2/20
50000/50000 [==============================] - 59s 1ms/step - loss: 0.9873 - acc:
0.6536 - val_loss: 1.1588 - val_acc: 0.5883
Epoch 3/20
50000/50000 [==============================] - 59s 1ms/step - loss: 0.8072 - acc:
0.7197 - val_loss: 1.0325 - val_acc: 0.6475
Epoch 4/20
50000/50000 [==============================] - 59s 1ms/step - loss: 0.6512 - acc:
0.7736 - val_loss: 1.0646 - val_acc: 0.6469
Epoch 5/20
50000/50000 [==============================] - 58s 1ms/step - loss: 0.4972 - acc:
0.8293 - val_loss: 0.9752 - val_acc: 0.6908
Epoch 6/20
50000/50000 [==============================] - 58s 1ms/step - loss: 0.3479 - acc:
0.8830 - val_loss: 1.0822 - val_acc: 0.6923
```

Let's now look at epochs 11 thru 20. You can see that we've hit a 100% on the training data, which means we are tightly fitted to the training data. On the other hand, our accuracy on the

validation data plateaued at 69.96%. Thus, after six epochs, there was no improvement from continued training, and we can conclude that by epoch 7 the model was overfitted to the training data.

```
Epoch 11/20
50000/50000 [==============================] - 58s 1ms/step - loss: 0.0198 - acc:
0.9944 - val_loss: 2.0663 - val_acc: 0.6915
Epoch 12/20
50000/50000 [==============================] - 59s 1ms/step - loss: 0.0112 - acc:
0.9970 - val_loss: 2.1746 - val_acc: 0.6939
Epoch 13/20
50000/50000 [==============================] - 59s 1ms/step - loss: 0.0077 - acc:
0.9978 - val_loss: 2.2837 - val_acc: 0.6907
Epoch 14/20
50000/50000 [==============================] - 60s 1ms/step - loss: 0.0048 - acc:
0.9988 - val_loss: 2.4035 - val_acc: 0.6916
Epoch 15/20
50000/50000 [==============================] - 60s 1ms/step - loss: 0.0025 - acc:
0.9994 - val_loss: 2.4452 - val_acc: 0.6980
Epoch 16/20
50000/50000 [==============================] - 59s 1ms/step - loss: 0.0017 - acc:
0.9996 - val_loss: 2.6432 - val_acc: 0.6912
Epoch 17/20
50000/50000 [==============================] - 59s 1ms/step - loss: 9.8118e-04 -
acc: 0.9998 - val_loss: 2.6411 - val_acc: 0.6973
Epoch 18/20
50000/50000 [==============================] - 59s 1ms/step - loss: 8.6766e-05 -
acc: 1.0000 - val_loss: 2.6870 - val_acc: 0.6994
Epoch 19/20
50000/50000 [==============================] - 58s 1ms/step - loss: 2.3760e-05 -
acc: 1.0000 - val_loss: 2.7450 - val_acc: 0.6991
Epoch 20/20
50000/50000 [==============================] - 59s 1ms/step - loss: 1.5005e-05 -
acc: 1.0000 - val_loss: 2.7683 - val_acc: 0.6996
```

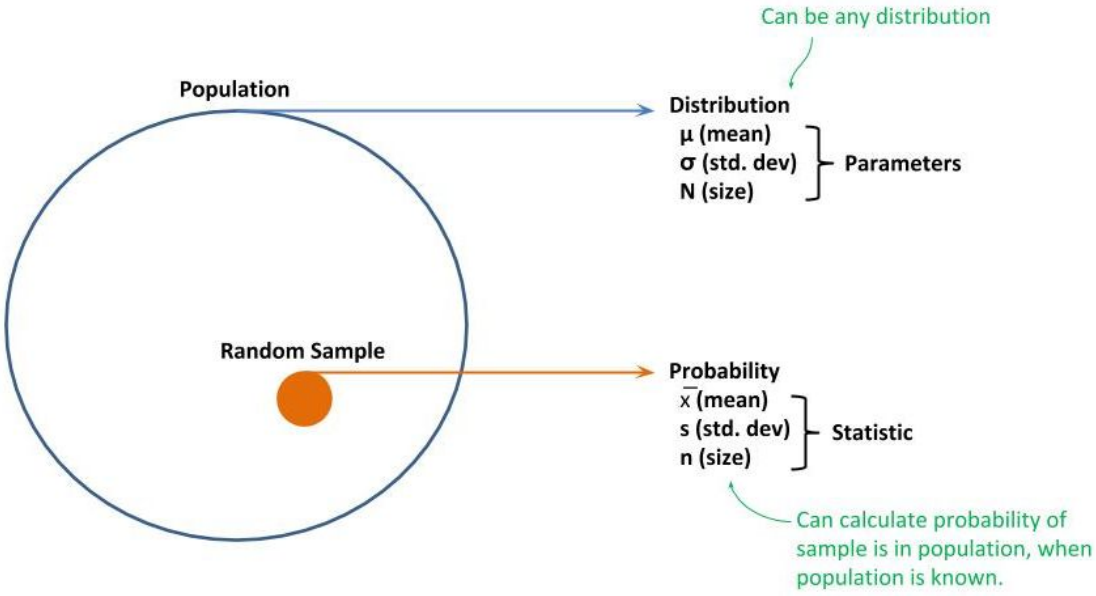
The values of the loss function for the training and validation data also indicate that the model is overfitting. The loss function between epochs 11 and 20 for the training data continues to get smaller, but for the corresponding validation data it had plateaued and then gets worse (i.e., diverging).

Steps

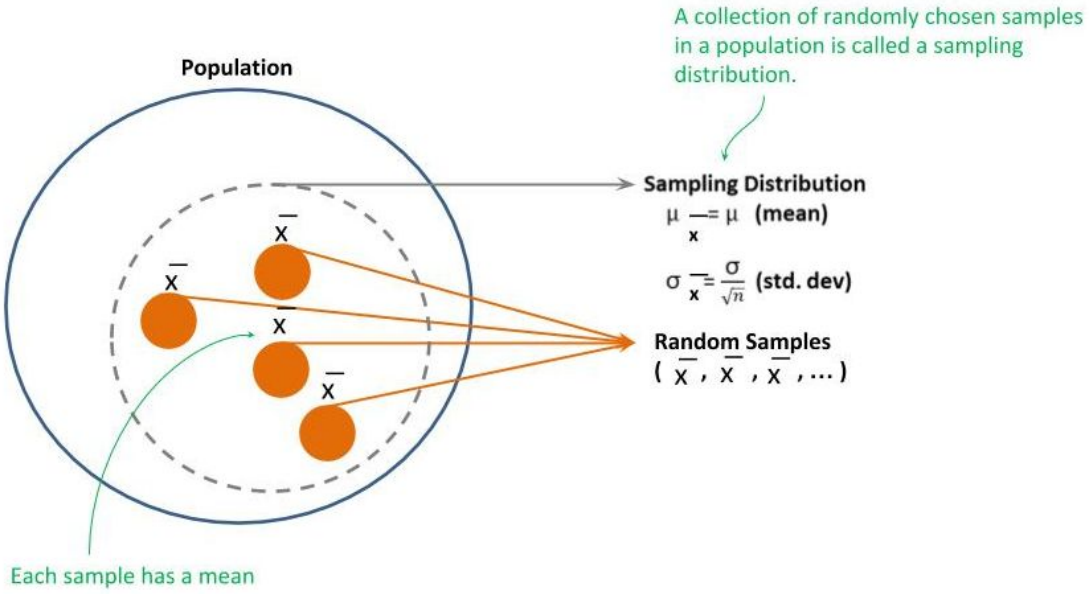
Over the course of the last few years, a lot of insights have been learned by researchers and practitioners on how neural networks learn. These insights have lead to focusing on regularization, normalization, augmentation, and sampling distributions. In shorter words, we can improve accuracy and reduce training time by changing the sampling distribution of the training dataset.

In epochs, we think of a sequential draw of batches from our training data. Even though we randomly shuffle the training data at the start of each epoch, the sampling distribution is still the same.

Let's now think of the entire population of what we want to recognize. In statistics, we call this the population distribution, as depicted in the figure below.



But we will never have a dataset that is the actual entire population distribution. Instead, we have some sample, which we refer to as a sampling distribution of the population distribution, as depicted in the figure below.



Another way to improve our model is to additionally learn the best sampling distribution to train the model with. While our dataset may be fixed, we can do a number of methods to alter the distribution, and thus learn the sampling distribution that best fits training the model. These methods include:

1. Regularization / Dropout
2. Batch Normalization
3. Data Augmentation

From this perspective, we no longer see feeding the neural network as sequential passes over the training data, but as making random draws from a sampling distribution. In this context, steps refers to the number of batches (draws) we will make from the sampling distribution of our training data.

When we add dropout layers to the neural networks, we are randomly dropping activations on a per sample basis. In addition to reducing overfitting of a neural network, we are also changing the distribution.

With batch normalization, we are minimizing covariance shift between our batches of training data (samples). Like using standardization on our input, the activations are rescaled using standardization (i.e., subtract the batch mean and divide by the batch standard deviation). This normalization reduces the fluctuations in updates to parameters in the model --referred to as adding more stability to the training. In addition, this normalization mimics drawing from a sampling distribution that is more representative of the population distribution.

With data augmentation, we create new samples by modifying existing samples within a set of parameters, from which we randomly select the modification, which also contributes to changing the distribution.

With batch normalization, regularization/dropout and data augmentation, no two epochs will have the same sampling distribution. In this case, the practice now is to limit the number of random draws from each new sampling distribution, which is referred to as steps. For example, if steps is set to 1000, then per epoch, only a 1000 random batches will be selected and feed into the neural network for training.

In Keras, we can specify both the number of epochs and steps as parameters to the `fit()` or `fit_generator()` methods:

```
model.fit(x_train, y_train,
          batch_size=32,
          epochs=10,
          steps_per_epoch=1000)
```

Batch Size

To understand how to set batch size, you should have a basic understanding of the three types of gradient descent algorithms; wherein the gradient descent algorithm is the means by which the model parameters are updated (learned) during training.

Stochastic Gradient Descent

In stochastic gradient descent (SGD), the model is updated after each sample is fed through during training. Since each sample is randomly selected, the variance between samples can result in large swings in the gradient. A benefit to this is that during training one is less likely to converge on a local optima, and more likely to find the global optima to converge on. Another benefit is that the rate of change in loss can be monitored in real-time, which may aid in algorithms that do auto-hyperparameter tuning. The downside is that this is more computationally expensive per epoch.

Batch Gradient Descent

In batch gradient descent, the error loss per sample is calculated as each sample is fed through during training, but the updating of the model is done at the end of each epoch (i.e., after the entire training data is passed through). As a result, the gradient is smoothed out since it's calculated across the loss of all the samples, instead of a single sample. A benefit to this is that this is less computational expensive per epoch and the training more reliably converges. The

downside is that the model may converge on a less accurate local optima, and an entire epoch needs to be run to monitor performance data.

Mini-Batch Gradient Descent

The mini-batch gradient descent method is a tradeoff between stochastic and batch gradient descent. Instead of one sample or all samples, the neural network is fed in mini-batches which are a subset of the entire training data. The smaller the mini-batch side, the more the training will resemble stochastic gradient descent, while larger batch sizes will resemble batch gradient descent.

For certain models and datasets, stochastic gradient descent (SGD) works best. In general, it's a common practice to use the trade-off of mini-batch gradient descent. The hyperparameter `batch_size` is the size of the mini-batch. Due to hardware architectures, the most time/space efficient batch sizes are multiples of 8, such as 8, 16, 32 and 64. The `batch_size` that is most commonly tried first is 32. One generally never sees a batch size greater than 128.

In Keras, you can specify the batch_size either in the model `fit()` or `fit_generator()` methods:

```
# model is a compiled keras model (Model or Sequential class).
model.fit(x_train, y_train, batch_size=32)
```

Alternately, if using the `ImageDataGenerator`, batch_size can be specified in the corresponding `flow()/flow_from_directory()/flow_from_dataframe()` methods:

```
# randomly rotate images +/- 20 degrees
datagen = ImageDataGenerator(rotation_range=20)

# train the model
model.fit_generator(datagen.flow(x_train, y_train, batch_size=32), epochs=10)
```

Learning Rate

The `learning rate` is generally the most influential of the hyperparameters. It can have a significant impact on the length of time to train a neural network, whether the neural network converges on a local optima, and whether it converges on the best (global) local optima.

When doing updates to the model parameters during the backward propagation pass, the gradient descent algorithm is used to derive a value to add/subtract to the parameters in the model from the loss function for that pass. These additions and subtractions could result in large

swings in parameter values. If a model has and continues to have large swings in parameter values, the model will be ‘all over the map’ and never converge.

What is convergence? This is when there is a steady reduction in the loss (referred to as minimizing the loss) and a steady increase or plateau in the accuracy per pass. If you observe big swings in the amount of loss and/or accuracy, then the training of your model is not converging. If the training is not converging, it won’t matter how many epochs you run, it will never finish training.

The learning rate provides us with a means to control the degree that the model parameters are updated. In the basic method, the learning rate is a fixed coefficient between 0 and 1 that is multiplied against the value to add/subtract, to reduce the amount being added or subtracted. These smaller increments add more stability during the training and increase the likelihood of convergence.

Small vs Large Learning Rate

If we use a very small learning rate, like 0.001, we will eliminate large swings in the model parameters during updates. This will generally guarantee that the training will converge on a local optima. But there is a drawback. First, the smaller we make the increments, the more passes of the training data (epochs) will be needed to minimize the loss. That means more time to train. Second, the smaller the increments the less likely the training will explore other local optimas, which might be more accurate than the one that the training is converging on; instead, it may converge on poor local optima or get stuck on a saddle point.

A large learning rate, like 0.1, likely will cause big jumps in the model parameters during updates. In some cases, it might initially lead to faster convergence (less epochs). The drawback is that even if you are initially converging fast, the jumps may overshoot and start causing the convergence to swing back and forth, or hop across different local optima. At very high learning rates, the training may start to diverge (i.e., increasing loss).

There are a lot of factors of what will be the best learning rate at different times during the training. In best practice the rate will range between 0.1 and 10e-5.

Below is a basic formula of how a weight is adjusted by multiplying the learning rate by the amount calculated to add/subtract (gradient):

```
weight += -learning_rate * gradient
```

Decay

A common practice has been to start with a slightly larger learning rate, and then gradually decay the learning rate. The larger learning rate would at first explore different local optima to converge on and make some initial deep swings into the respective local optimas. The rate of convergence and minimizing the loss function on the initial updates can be used to hone in on the best (good) local optima. From that point, the learning rate is gradually decayed. As the learning rate decays, it is less likely for swings out of the good local optima to occur and the steadily decreasing learning rate will tune the convergence to approach the minimal point; albeit, the smaller and smaller learning rate will increase training time. So the decay becomes a trade-off between small increases in final accuracy and the overall training time.

Below is a basic formula of how decay is added to the calculation of updating the weights, where on each update, the learning rate is reduced by the decay amount.

```
weight += -learning_rate * gradient
learning_rate -= decay
```

Momentum

Another practice is to accelerate or decelerate the rate of change based on prior changes. If we have large jumps in convergence, we risk jumping out of the local optima, so we may want to decelerate the learning rate; while if we have small to no changes in convergence, we may want to accelerate the learning rate to hop over a saddle point. Typically values for momentum range from 0.5 to 0.99.

```
velocity = (momentum * velocity) - (learning_rate * gradient)
weight += velocity
```

Adaptive Learning Rate

There are many popular algorithms that dynamically adapt the learning rate:

- Adadelta
- Adagrad
- Adam
- AdaMax
- AMSGrad
- Momentum
- Nadam
- Nesterov
- RMSprop

The explanation of these are beyond the scope of this section. For Keras, these learning rate algorithms are specified when the optimizer is defined for minimizing the loss function.

```
from keras import optimizers

# instantiate an optimizer
optimizer = optimizers.RMSprop(lr=0.001, rho=0.9, epsilon=None, decay=0.0)

# compile the model, specifying the loss function and optimizer
model.compile(loss='mean_squared_error', optimizer=optimizer)
```

Validation Data

An alternative to splitting the dataset into training and test data is to have an additional validation dataset. In this version -- train, validation, and test --, the test data contains the holdout data that is not used during training and only for final evaluation.

The validation data is drawn from the training data. When the training dataset is large, the validation data is pre-drawn from the training data and set aside. On smaller training datasets, the validation data is randomly drawn and removed from the training data on a per epoch basis, which is referred to as cross-validation.

Typically, the validation data will be 5 to 10% of the training data. The remaining training data is then fed through the neural network, and a training accuracy is determined. Subsequently, a forward pass (prediction) is made of the validation data and a validation accuracy is determined.

This method provides additional insight while training; whereby after each epoch the validation data gives a likely indicator of what the accuracy would be with the test data. Along with the rate of reduction in the loss function, convergence of the training accuracy, the validation accuracy adds more information about the progress of the training. For example, if the training starts to overfit the model, we would expect to see an increasing gap between the training and validation accuracy.

Feeding

To train a neural network, the training data is fed through the neural network in a forward pass either as individual samples (stochastic), batches (mini-batch) or an epoch (batch) at a time, followed by a backward propagation pass to update the parameters of the model. This process is referred to as feeding the neural network, and the mechanisms for feeding the neural network are called feeders.

Keras has several feeders which can be used with image data.

In-Memory Feeder

If your memory resources are sufficient to hold the entire preprocessed image data in memory, the most efficient (fastest) way to feed a neural network for training is to use an in-memory feeder. Typically, the preprocessed image data is stored as a contiguous region of memory. The feeder uses an indirect index to each preprocessed image in memory, by which the feeder can randomly shuffle, draw batches, and do transforms without moving memory.


```
# the mini-batch size
batch_size = 32

# the number of epochs
epochs=10

# once the model is compiled and the image data has been preprocessed,
# the fit method will start training the model, where x_train and y_train
# are the preprocessed image data and one-hot encoded labels
model.fit(x_train, y_train, batch_size=batch_size, epochs=epochs)
```

In-Memory Generator

The `fit_generator()` method is similar to the `fit()` method, but uses a generator, which is constructed with the `ImageDataGenerator` class. This generator enables image augmentation, where augmented images are generated on-the-fly as the neural network is being fed. The `flow()` method of the data generator then constructs mini-batches which consist of a combination of the original images and augmented versions.

```
# randomly rotate images +/- 20 degrees
datagen = ImageDataGenerator(rotation_range=20)

# train the model using the flow() method of the generator to create mini-batches
model.fit_generator(datagen.flow(x_train, y_train, batch_size=32), epochs=10)
```

On-Disk Generator

If your memory resources are not sufficient to hold the entire preprocessed dataset in memory, then you will need to continuously feed the neural network by drawing batches from the on-disk storage of the preprocessed (or original) image data.

The drawback is that for each mini-batch, the generator must re-read the preprocessed (or original) data from the disk.

In the code below, we first create a generator using the `ImageDataGenerator` class, which we will use in conjunction with the `flow_from_directory()` method. This method reads the image data in its original image format. We will need to pass parameters to do some preprocessing on-the-fly each time an image is read from disk. In this example we add normalization of the pixel data by setting the `rescale` parameter to divide each pixel value by 255, which will normalize the pixels (for 8 bits per channel). The `flow_from_directory()` will then create mini-batches of size 32 and resize each preprocessed image to fit an input vector of 128x128x3 (for three channels --RGB).

```
# normalize the pixel data (rescale) as its read from memory, and
# randomly rotate images +/- 20 degrees
datagen = ImageDataGenerator(rescale=1/255., rotation_range=20)

# train the model using the flow_from_directory() method
model.fit_generator(datagen.flow_from_directory('dataset', target_size=(128,128),
                    batch_size=32), epochs=10)
```

Hybrid Mixed In-Memory and On-Disk

Continuously re-reading each image from on-disk is the least efficient method to feed the neural network for training. Instead, one can consider using a hybrid approach. Let's revisit the concept of a sampling distribution, which approximates the distribution of a population. Let's say you have 16GB of memory to hold data, and the preprocessed dataset is 64GB. What we do in a hybrid feeding, is that we take a large segment of the preprocessed data (8BG in our example) at a time, which has been stratified. We then are going to repeatedly feed the same segment to the neural network as epochs. But each time, we do image augmentation such that each epoch is a unique sampling distribution of the entire preprocessed image dataset. Below is a description of the steps to do a hybrid in-memory/on-disk feeding:

1. Create a stratified index to the preprocessed image data on disk.
2. Partition the stratified index into partitions based on the available memory to hold a segment in memory.
3. For each segment:

a. Repeat for a specified number of epochs
    i. Randomly shuffle the segment per epoch.
    ii. Randomly apply image augmentation to create a unique sampling distribution per epoch.
    iii. Feed the mini-batches to the neural network.

Test Data

The test data, also known as the hold-out set, is the portion of the dataset that was not used in training. These are samples that the model never saw during training. Once the model has been trained with the training data, a forward pass is made through the trained model (referred to as evaluation) to predict the likely accuracy the model will have when deployed and predicting (inference) on samples it has not seen before.

Generally, the accuracy of the test data is slightly less than that of the training data. If there is a large difference between the training accuracy (high) and the test accuracy (low), then the model is overfitted to the training data.

In **Keras**, once the model has been trained, we use the `evaluate()` method to obtain the accuracy on the test data.

```
# Evaluate the model using the test (hold-out) data
score = model.evaluate(x_test, y_test)
print("Test Accuracy", score[1])
```

Below is an example output of the above example code:

```
# example output
Test accuracy: 0.6996
```

Next

In the second part, we will cover training in more detail, as well as checkpointing, and deployment.

Part 11 - Training and Deployment

Overview

In this part, we will cover best practices for training and deploying a computer vision (image recognition) model.

Training Cycle

Let's start by discussing the training cycle, before we do a deep dive. The general phases of training are:

- Pretraining - find best hyperparameters for training. Repeat:
    - Set initial hyperparameters.
    - Run a partial training session.
    - Examine rate of changes in accuracy and loss on training and validation data.
    - If acceptable, stop the pre-training phase.

- Set training objectives:
    - Accuracy on validation data.
    - Rate of progression in reducing loss on training data.
    - Plan number of epochs or steps to meet objectives.

- Training:
    - Start full training.
    - If objectives are met, early stop and checkpoint model
    - If diverging from objectives, terminate training

Pretraining

The purpose of the pretraining phase is to find the best hyperparameter settings to initiate a full training run. Let's revisit what goes wrong when the wrong hyperparameters are used during training:

- It takes an excessive amount of time to train the model.
- The model gets stuck on a saddle point and never converges.
- The model becomes overfitted and diverges.

Weights/Biases Initialization

When training starts, all the weights and biases need an initial value. Those initial values can impact how long and how accurate the model will be, even when everything else is right. If we initialize the weights to zero, then all the updates to the weights in the neural network during training will be the same --i.e., the neurons will be symmetric. As a result, the network is the same as a single neuron.

For biases, this is not the case, and the current practice is to initialize them to zero.

For the weights, they are initialized by some random distributions. A uniform random distribution will generally produce a good result. But practices have shown that a uniform distribution can result in significant variance in the final accuracy. In the past, using a uniform random distribution, several instances of the model would be trained in parallel with different uniform random distributions.

In 2010, a paper by Xavier Glorot demonstrated that drawing a random distribution from a Gaussian distribution with a mean of zero resulted more consistently in obtaining the best accuracy. The specific distribution, known now as Xavier initialization, works best when the activation function for hidden units is a tanh, which was the convention of the time. In 2015, a variant of Xavier known as He, was found to work best when the activation function for the hidden units was a ReLU or Leaky ReLU.

Keras supports a large variety of random initializers. By default, the weights are initialized by a uniform random distribution and the biases are set to zero. Since CNN uses ReLU and Leaky ReLU activations for the convolutional and dense layers, the best practice today is to initialize the weights using the He initialization.

```
from keras import initializers

# initialize the weights using He initialization when activation function is ReLU
model.add(Dense(128, kernel_initializer='he_normal'))
model.add(ReLU())
```

Initial Learning Rate

The learning rate is considered to have the most impact on training your model and the most uncertainty of what the initial rate should be. It is highly dependent on the type of image data and the distribution of images (samples) in the dataset. The best practice is that the initial learning rate will be between 0.1 and 10e-5. But how do you pick the number? Does one just guess and try and try again? Sort of. Today, we do what's called grid search. One will train several instances of the model for short number of epochs at different learning rates and look at

the rate of change in the loss and accuracy of the validation data. Generally, a grid search is used on a set of learning rates, where the set is typically a logarithmic scale between 0.1 and 10e-5, such as: `[ 0.1, 0.01, 0.001, 0.0001, 0.00001 ]`

For each learning rate, an instance of the model is trained for a small number of epochs. This is typically 5 or 10 epochs. The loss vs. accuracy is then plotted per learning rate. One then looks for the learning rate that sustains a reduction in loss and increase in accuracy vs. a sharp increase in loss. When plotted, once tends to see a U shape. The ideal best learning rate will either be at the bottom of the U shape or on the near the bottom on the left-side (converging).

What you're trying to achieve here is to find the highest initial learning rate that:

- Makes the fastest route to convergence on initial training start,
- Prevents the training from being stuck on a saddle point, but:
- Not so large, that it either:
    - Bounces around different local optima, or
    - Dives into a poor local optima.

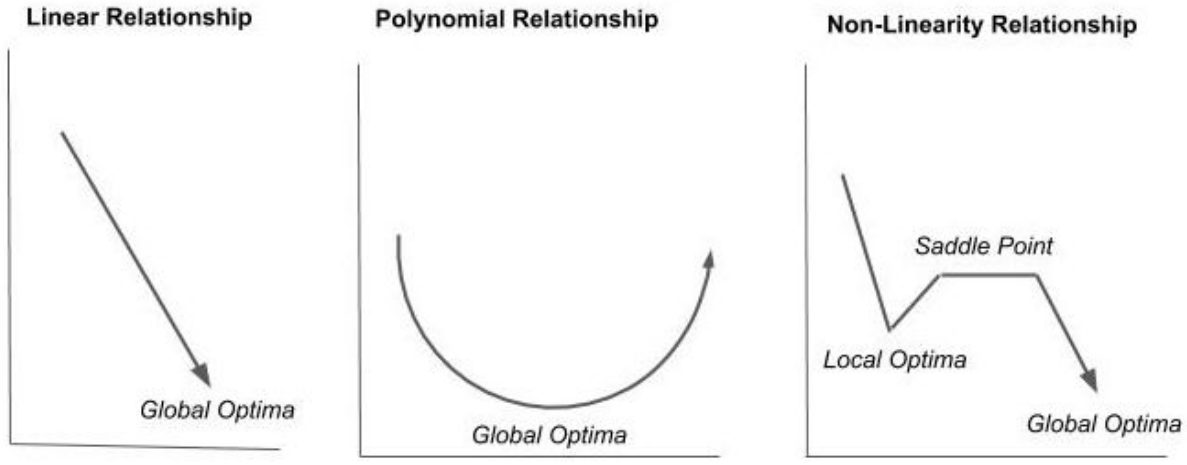
On the other hand, one does not want to pick an initial learning rate on the lowest-end that:

- Increases training time to converge,
- Gets stuck or a saddle point, or
- Does not explore for better local optimas.

The concept of gradient descent is a measurement of the slope of the rate of change. If we have a function, say f(x) where the relationships between the input and the outputs is linear, we would expect the slope to be a straight line, as depicted below. In this case, the learning rate would only affect how long it would take to find the global optima, but since there is only one optima, we would find it nonetheless.

On the other hand, if f(x) is a polynomial function, we expect the slope to look more like a curved bowl. Now the learning rate would affect whether we find the global optima. That is, too high of a rate and we bounce around the curved bowl. A tiny learning rate, and we are (almost) guaranteed to descend eventually to the global optima, but it may take a long time.

Linear and polynomial functions are easily solvable with the classical modeling techniques of linear algebra, such as linear/logistic regression/classifiers and CART analysis. What deep learning and neural networks provide is the ability to find solutions, and hence very good or the actual global optima of real-world problems that have high non-linearity.



Gradient Descent - Slope of Rage of Change

In the simplified third example depicted above, the function f(x) is comprised of segments of non-linearity. In addition to a global optima, we depict a local optima which the training could dive into and converge on, overlooking the global optima. Likewise is a saddle point. This is a plateaued region of the slope. If our learning rate is too small, one might bounce back and forth on the plateau and never escape.

Decay and Momentum

In addition to the initial learning rate, optimizer algorithms employ a variety of techniques to adapt the learning rate as training proceeds. The most basic is decay. The assumption here is that one starts with a somewhat high learning rate to speed up diving down into a good optima, but then gradually decay the learning rate to prevent overlooking better local optima, and if a good optima, not to bounce out of it. As the learning rate is decayed, the rate of reducing the loss will also decrease, which will lengthen the overall learning. It's a trade-off between overall time to train the model vs. converging on a global (or otherwise good) optima.

Momentum is another strategy used in conjunction with decay. In this case, instead of using a fixed decay rate, the previous rate of change is used to dynamically increase or decrease the decay rate.

The details and options of optimizers is outside the scope of this handbook. In Keras, optimizers can be specified in two ways.

As a string value to the parameter `optimizer` in the `compile()` method. In this case, the selected optimizer is configured with default settings.

```
model.compile(optimizer='rmsprop', loss='categorical_crossentropy',
```

```
              metrics=['accuracy'])
```

An optimizer can also be specified from the `optimizer` class, and configured when the optimizer object is instantiated.

```
from keras import optimizers

optimizer = optimizers.RMSProp(lr=0.01)
model.compile(optimizer=optimizer, loss='categorical_crossentropy',
              metrics=['accuracy'])
```

Checkpointing and Early Stop

Checkpointing

Checkpointing is periodically saving the learned model parameters and current hyperparameter values during training. There are two reasons for doing this:

- To be able to resume training of a model without restarting the training (i.e., where it left off).
- Identify a past point training that the model gave the best results.

In the former case (resume training), for resource management one may split the training across sessions. For example, one might reserve (or be authorized) one hour a day for training. At the end of the one hour training each day, the training is checkpointed. The following day, training is resume by restoring from the checkpoint.

Why wouldn't saving the model's weights and biases be enough? In neural networks, some of the hyperparameter values will dynamically change, such as the learning rate and decay. One would want to resume at the same hyperparameter values at the time the training was paused.

In another scenario, one may implement continuous learning as a part of a continuous build and integration process. In this scenario, new labeled images are continuously added to the training data, and one only wants to incrementally retrain the model vs. retraining from scratch on each build cycle.

In the later case (find best result), during training the model may have trained past the best optima, and started to diverge and/or overfit. In this case, one would not want to start retraining from scratch with fewer epochs (or other hyperparameter changes), but instead identify the epoch that achieved the best results, and restore (set) the learned model parameters to those that were checkpointed at the end of that epoch.

Checkpointing occurs at the end of an epoch, but should one checkpoint after each epoch? Probably not. That can be expensive space wise. Let's presume that the model has 25 million parameters (e.g., ResNet-50), where each parameter is a 32-bit floating point value (4 bytes). Each checkpoint would then require 100Mb to save. After 10 epochs, that would already be one gigabyte of disk space.

One generally only checkpoints after each epoch if the number of model parameters is small and/or the number of epochs is small. In the code example below, a checkpoint is instantiated with the `ModelCheckpoint` class. The parameter `filepath` is the file path of where to write the checkpoint to. The filepath can either be a complete file path or a formatted file path. In the former case, the checkpoint file would be overwritten each time. In the case below, we used the format syntax `epoch:02d' to generate a unique file for each checkpoint, based on the epoch number. For example, if it's the third epoch, the file would be 'mymodel-03.h5'. Also note that checkpoints are written in a HDF5 file format.

```
from keras.callbacks import ModelCheckpoint

# Create a unique checkpoint file per checkpoint using the formatting option
# {epoch:02d}
filepath = "mymodel-{epoch:02d}.hdf5"

# Create a checkpoint
checkpoint = ModelCheckpoint(filepath)

# Train the model and use the callbacks parameter to enable the checkpoint
model.fit(x_train, y_train, epochs=epochs, batch_size=batch_size,
          callbacks=[checkpoint])
```

A model can then be subsequently restored from a checkpoint using the `load_model()` method:

```
from keras.models import load_model

# restore a model from a saved checkpoint
model = load_modal('mymodel-03.h5')
```

For models with larger number of parameters and/or number of epochs, one may choose to save a checkpoint on every nth epoch (e.g., every 4th epoch) with the parameter `period`. In the code example below, a checkpoint is saved on every 4th epoch:

```
from keras.callbacks import ModelCheckpoint
```

```
# Create a unique checkpoint file per checkpoint using the formatting option
# {epoch:02d}
filepath = "mymodel-{epoch:02d}.hdf5"

# Create a checkpoint for every 4th epoch
checkpoint = ModelCheckpoint(filepath, period=4)

# Train the model and use the callbacks parameter to enable the checkpoint
model.fit(x_train, y_train, epochs=epochs, batch_size=batch_size,
          callbacks=[checkpoint])
```

Alternately, one may choose to save the current best checkpoint with the parameters `save_best_only` set to `True` and `monitor` to the measurement to base the decision on. For example, if `monitor` is set to `val_acc`, it will only write a checkpoint if the valuation accuracy is higher than the last saved checkpoint. If the parameter is set to `val_loss`, it will only write a checkpoint if the valuation loss is lower than the last saved checkpoint.

```
from keras.callbacks import ModelCheckpoint

# Create a unique checkpoint file per checkpoint using the formatting option
# {epoch:02d}
filepath = "mymodel-{epoch:02d}.hdf5"

# Create a checkpoint for every 4th epoch
checkpoint = ModelCheckpoint(filepath, save_best_only=True, monitor='val_acc')

# Train the model and use the callbacks parameter to enable the checkpoint
model.fit(x_train, y_train, epochs=epochs, batch_size=batch_size,
          callbacks=[checkpoint])
```

Early Stopping

An early stop is setting a condition upon which training is terminated earlier than the set limits (e.g., number of epochs). This is generally set to conserve resources and/or prevent overtraining when a goal objective is reached, such as a level of accuracy, convergence on evaluation loss, etc. For example, one might set a training for 20 epochs, which average 30 minutes each --for a total of 10 hours. But if the objective is met after 8 epochs, it would be ideal to terminate the training, saving 6 hours of resources.

An early stop is specified in a manner similar to a checkpoint. An `EarlyStopping` object is instantiated with the configured with target goal, and passed to the `callbacks` parameter of the `fit()` method.

In the code example below, training will be stopped early if the valuation loss stops reducing from the previous epoch:

```
from keras.callbacks import EarlyStopping

# set an early stop (termination of training) when the valuation loss has stopped
# reducing (default setting).
earlystop = EarlyStopping(monitor='val_loss')

# Train the model and use early stop to stop training early if the valuation loss
# stops decreasing
model.fit(x_train, y_train, epochs=epochs, batch_size=batch_size,
          callbacks=[earlystop])
```

In addition to monitoring the valuation loss for early stop, one can alternately monitor the valuation accuracy with the parameter setting `monitor="val_acc"`. There are some additional parameters for fine tuning to prevent inadvertent early stop, such as on a saddle point where more training will overcome. The parameter `patience` specifies a minimum number of epochs without improvement before early stop, and the parameter `min_delta` specifies a minimum threshold to determine if the model improved or not. In the code example below, the training will stop early if there is no improvement in the valuation loss after three epochs.

```
from keras.callbacks import EarlyStopping

# set an early stop (termination of training) when the valuation loss has stopped
# reducing for three epochs.
earlystop = EarlyStopping(monitor='val_loss', patience=3)

# Train the model and use early stop to stop training early if the valuation loss
# stops decreasing
model.fit(x_train, y_train, epochs=epochs, batch_size=batch_size,
          callbacks=[earlystop])
```

Model Saving/Restoring

Save

In Keras, we can save both the model and the trained parameters (i.e., weights and biases). The model and weights can be saved separately or together. The `save()` method will save both the weights/biases and the model to a specified file in HDF5 file format. Below is an example:

```
# Train a model
model.fit(x_train, y_train, epochs=epochs, batch_size=batch_size)

# Save the model and trained weights and biases.
model.save('mymodel.h5')
```

The trained weights/biases and the model can be saved separately. For example, one may want to create and save a model, which is not trained, and then use this pre-built model to train several different versions of the same model; whereby the saved model can be reused as a pre-built model, and the saved (learned) model parameters can be used for transfer learning.

When saving a model only, the model architecture can be saved either in JSON or YAML format. In the example below, the method `to_json()` returns the model architecture as a JSON object in a string format (i.e., not a dictionary format). The JSON string representation of the model architecture is then written to a file (e.g., model.json)

```
import json

# Save the model in JSON string format
json_string = model.to_json()

# Write the JSON string to a file
with open('model.json', 'w') as f:
         f.write(s)
```

In the code example below, the learned model weights and biases, after training, are saved in a HDF5 format:

```
# Train a model
model.fit(x_train, y_train, epochs=epochs, batch_size=batch_size)

# Save the trained (learned) model parameters
model.save_weights('mymodel-weights.h5')
```

Restore

In Keras, we can restore a model architecture and/or the model parameters (i.e., weights and biases). Restoring a model architecture is generally done for loading a pre-built model, while loading both the model architecture and model parameters is generally done for transfer learning. One should note that loading the model and model parameters is not the same as checkpointing, in that one is not restoring the current state of hyperparameters and this method therefore should not be used for continuous learning.

```
from keras.models import load_model

# load a pre-trained model
model = load_model('mymodel.h5')
```

In the code example below, the model architecture is loaded from a previous saved model architecture (i.e., pre-built model) as a JSON string.

```
from keras.models import model_from_json

# Read the JSON string from a file
with open('mymodel.json', 'r') as f:
         s = f.read()

# Load the model architecture
model = model_from_json(s)
```

In the code example below, the trained weights/biases for a model are loaded into the corresponding pre-built model, using the `load_weights()` method. Having different sets of trained weights/biases for the same pre-built model can be advantages for transfer learning. In this case, separate instances of the pre-built model are trained for different categories of images (e.g., medical, nature, etc). In this case, when doing transfer learning, one selects the set of trained weights/biases which corresponds to the category of the new image data to train a new model instance.

```
from keras.models import load_weights
from keras.models import model_from_json

# Read the JSON string from a file
with open('mymodel.json', 'r') as f:
         s = f.read()

# Load the model architecture
model = model_from_json(s)

# Load the trained weights for the model
model.load_weights('mymodel-weights.h5')
```

Hyperparameter Search

Hyperparameter search is the process of finding the optimal (or near) values for initializing the hyperparameters for training. Depending on the tools and methodology, the process maybe assisted or automated. In both cases, a process will take a range of hyperparameter values, a distribution within that range, and generate some set of combinations. For each combination, a training instance is executed for some short-length training session. Generally, depending on the compute resources, a plurality of training instances are executed in parallel.

Once all the hyperparameter search training sessions have completed, one then selects the best combination by:

- In the assist case, each combination is displayed with a corresponding set of metrics at the time of the short length training session completed. These typically are:
    - Validation Accuracy, Recall, Precision, F1
    - Validation Loss, Rate of Change
- In the automated case, the algorithm will process the metrics for each combination and make a determination which combination will provide the best results in a full training run.

Hyperparameters to Tune

The most influential hyperparameter is the learning rate. In general, the following is a rank order of how influential a hyperparameter will be to tuning for optimizing the training of a model (time and performance):

- Learning Rate
- Dropout (or other Regularization)
- Batch Size
- Optimizer Algorithm

The following are common practices when evaluating hyperparameter tuning:

1. Good: Expect a big drop in training loss from the first epoch by the third epoch.
2. Bad : The training loss stays plateaued, or decreases in tiny amounts.
3. Bad : The train loss shows NaN.
4. Good: The validation accuracy, while modestly less, parallels the train accuracy.
5. Bad : The validation accuracy stays plateaued while the training accuracy decreases.
6. Bad : The validation accuracy increases while the training accuracy decreases.

7. Good: The validation loss steadily decreases.
8. Bad : The validation loss stays plateaued.
9. Bad : The validation loss bounces back (decreases) and forth (increases).

Common ranges for hyperparameter search are:

Learning Rate: [0.1, 0.01, 0.001, 0.0001, 0.00001]
Dropout : 0.10, 0.25, 0.50
Batch Size : 32, 64, 128
Optimizer : Adam, Adagrad, RMSprop

In general, one would not try all combinations; in that the number of combinations would be too high. In our example above, we would have 5 x 3 x 2 x 3 = 90. Generally, a more modest number of combinations are tried; whereby the combinations are selected at random.

Homebrew Search

Hyperparameter search is still considered a black magic art. Before using an assist or automated hyperparameter search tool, I recommend one gets familiar with the process by performing a homebrewed search for awhile build up one's personal insight.

The code example below is a simple homebrewed hyperparameter assisted search, which one can practice with and expand on. The code consists of the following:

- model_fn() - Constructs a model instance, and compiles it. The dropout layer in the model is configurables, as well as the optimizer and learning rate.

- train_fn() - Does a short duration training (specified by number of epochs) on the train/validation data. Will output train and validation loss and accuracy on each epoch.

- hyper_search(): Does the combination selection and invokes making a model instance and short training session per combination.

```
from keras import optimizers
import random

def model_fn(learning_rate, optimizer, dropout):
        ''' make an instance of the model '''
        # ADD CODE to construct the model here
        # set dropout rates based on the dropout parameter.
        # common convention is to ½ the dropout amount on each subsequent dropout
        # layer.
```

```
        # select the optimizer and set the learning rate
        if optimizer == 'adam':
                opt = optimizers.Adam(lr=learning_rate)
        elif optimizer == 'adagrad':
                opt = optimizers.Adagrad(lr=learning_rate)
        elif optimizer == 'rmsprop':
                opt = optimizers.RMSprop(lr=learning_rate)
        elif optimizer == 'sgd':
                opt = optimizers.SGD(lr=learning_rate)

        # compile the model
        model.compile(loss='categorical_crossentropy', optimizer=opt,
              metrics=['accuracy'])

        # return the model
        return model

def train_fn(model, nepochs, batch_size, x_train, y_train, x_val, y_val):
        ''' train the model for a fixed number of epochs '''

        # create a feeder and add some basic image augmentation
        datagen = ImageDataGenerator(horizontal_flip=True, vertical_flip=True,
                                     rotation=30)

        # train the model for the short number of epochs
        model.fit_generator(datagen.flow(x_train, y_train, batch_size=batch_size,
                            shuffle=True),
                            steps_per_epoch=len(x_train) / batch_size,
                            epochs=nepochs, verbose=1,
                            validation_data=(x_val, y_val))

def hyper_search(nepochs, ncombos, x_train, y_train, x_val, y_val):
        ''' Perform a Hyperparameter Search'''
        # the hyperparameter ranges to search from
        learning_rates = [ 0.1, 0.01, 0.001, 0.0001, 0.00001 ]
        dropouts = [ 0.10, 0.25, 0.25 ]
        batch_sizes = [ 32, 128 ]
        optimizers = [ 'adam', 'adagrad', 'rmsprop' ]

        # Generate the specified (ncombos) random combinations
        for n in range(ncombos):
                learning_rate = random.choice(learning_rates)
                dropout = random.choice(dropouts)
                batch_size = random.choice(batch_sizes)
                optimizer = random.choice(optimizers)

                # Construct the model instance
```

```
                model = model_fn(learning_rate, optimizer, dropout)

                # Do the short run training session
                train_fn(model, nepochs, batch_size, x_train, y_train, x_val,
                         y_val)
```

Hyperparameter Assist and Automation Tools

There are a number of packages for doing automated or assisted search for the best hyperparameters settings when training a model in Keras. The three most popular packages currently are (in no specific order):

- Talos (Assist)
- Hyperas (Assist)
- scikit-learn's GridSearchCV (Automated)

Deployment

Prerequisite

To deploy a trained model for use in an application, one should have first saved the model architecture and trained weights/biases, such as with the `model.save(filepath)` method (see earlier description). Once saved, one will want to copy (move) the corresponding file to the compute instance where the model will be ran.

Data Preprocessing Pipeline

One needs to create a data preprocessing pipeline that will preprocess the new images in the same way that the images in the training dataset were preprocessed. This includes:

- Resizing the images.
- Padding of the images.
- Number of Channels
- Pixel data type
- Normalization or Standardization

For resizing, the images will need to be the same shape as the input shape of the trained model (e.g., (128, 128, 3)). It's likely that the source of the new images will be in a different height and width shape from the input shape, and in other cases the images maybe different in height and weight amongst them.

Models are typically trained on shapes where height == width (e.g., 128 x 128). If your new images always have height == width, then one can simply resize them to the input shape of the model. When resizing, if the image is being downsampled (made smaller), the interpolation algorithm INTER_AREA typically will produce the best results, while if upsampled (made bigger), the interpolation algorithm INTER_CUBIC will typically produce the best results.

If your images are from a mobile camera, they will likely be either in a portrait or landscape layout, where height != width. If one arbitrarily resized these images into a square (height == width), the original aspect ratio will not be preserved, and the spatial relationships of features in the image will be distorted. In this case, the image needs to be padded so the height == width, prior to resizing.

One also needs to consider the number of channels of the new image. If the model takes three channels (e.g., RGB) and the new image is grayscale, one will need to first convert it to 3-channel RGB image. In the same scenario, if the image is a PNG with an alpha channel for transparency (i.e., 4 channels), the 4th channel needs to be removed.

If the model was trained with images that had 8 bits per pixel per channel (UINT8), and the new image has 16 bits per pixel per channel (UINT16), one needs to change the datatype if the pixels, correspondingly.

The pixel data will then need to be normalized or standardized the same manner that was used in the training of the model. In the case of standardization, the values calculated for the mean and standard deviation from the training data needs to be saved and copied with the saved model to the compute engine, and loaded into the preprocessing pipeline.

Model Loading

While the application is running, the model should only be loaded one time (vs. loading per prediction). It would be very inefficient to continuously load the model, given the very large size of a typical model's weights and biases (e.g., 100Mb+).

Diagnostics

A final note on diagnostics while training. We will show outputs from some training sessions that demonstrate when something is going wrong.

In our example, we use a simple ConvNet with 2M parameters to train the coarse labels for CIFAR-100 dataset, which consists of 20 categories. We use the builtin dataset in Keras and further split the training data into training and validation data, as below:

```
from keras.datasets import cifar100
import numpy as np

# get the train and test data
(x_train, y_train), (x_test, y_test) = cifar100.load_data(label_mode='coarse')

# normalize the pixel data between 0 and 1
x_train = (x_train / 255.0).astype(np.float32)
x_test  = (x_test / 255.0).astype(np.float32)

# convert the labels to categorical
nclasses = np.max(y_train) + 1
y_train = utils.to_categorical(y_train, nclasses)
y_test  = utils.to_categorical(y_test, nclasses)

# further split off from the training data the validation data (10%)
pivot = int(len(x_train) * 0.9)
x_val = x_train[pivot:]
y_val = y_train[pivot:]
x_train = x_train[:pivot]
y_train = y_train[:pivot]
```

We ran several training sessions using an Adam optimizer for ten epochs each, and varied the learning rate. On our first training session, we use an aggressive learning rate of 0.1. From the results below, one sees our accuracy loss from the first epoch to the tenth never actually decreases and our accuracy is stuck in the 5% range. For 20 categories, that what we would expect if the prediction is random. We are not learning anything, but simply bouncing back and forth trying to find an optima to dive down into.

```
Epoch 1/10
352/351 [==============================] - 44s 125ms/step - loss: 15.2716 - acc:
0.0503 - val_loss: 15.3767 - val_acc: 0.0460
Epoch 2/10
352/351 [==============================] - 42s 119ms/step - loss: 15.3049 - acc:
0.0505 - val_loss: 15.3767 - val_acc: 0.0460
Epoch 3/10
352/351 [==============================] - 42s 120ms/step - loss: 15.3052 - acc:
0.0504 -
Epoch 10/10
352/351 [==============================] - 41s 117ms/step - loss: 15.3055 - acc:
0.0504 - val_loss: 15.3767 - val_acc: 0.0460
```

Next, we lower the learning rate by a magnitude to 0.01. We had a substantial drop in the initial loss

from 15 to 3. But, as above, our loss stays essentially the same and the accuracy again is around 5%. While our swings are not as wild while looking for an optima to dive into, the learning rate is still too large to find one.

```
Epoch 1/10
352/351 [==============================] - 43s 123ms/step - loss: 3.0251 - acc:
0.0500 - val_loss: 2.9968 - val_acc: 0.0460
Epoch 2/10
352/351 [==============================] - 40s 114ms/step - loss: 2.9964 - acc:
0.0486 - val_loss: 2.9965 - val_acc: 0.0460
Epoch 3/10
352/351 [==============================] - 40s 114ms/step - loss: 2.9961 - acc:
0.0490 - val_loss: 2.9966 - val_acc: 0.0422
...
Epoch 10/10
352/351 [==============================] - 40s 115ms/step - loss: 2.9959 - acc:
0.0466 - val_loss: 2.9965 - val_acc: 0.0422
```

Next, we lower the learning rate by another magnitude to 0.001. Now, we see some improvement. Our loss is steadily going down and our accuracy is going up. But the rate of decline in the loss function is small. We don't know if we are just going so slow it will take a lot of epochs, or we have dived into a poor local optima.

```
Epoch 1/10
352/351 [==============================] - 43s 123ms/step - loss: 2.5162 - acc:
0.2211 - val_loss: 2.4523 - val_acc: 0.2610
Epoch 2/10
352/351 [==============================] - 40s 114ms/step - loss: 2.1625 - acc:
0.3258 - val_loss: 2.2507 - val_acc: 0.3200
Epoch 3/10
352/351 [==============================] - 40s 114ms/step - loss: 2.0423 - acc:
0.3662 - val_loss: 2.1630 - val_acc: 0.3482
...
Epoch 10/10
352/351 [==============================] - 40s 115ms/step - loss: 1.7465 - acc:
0.4550 - val_loss: 1.9470 - val_acc: 0.4180
```

Next, we double this learning rate to 0.002 and see if we can get an improvement. We don't really see any difference.

```
Epoch 1/10
352/351 [==============================] - 44s 125ms/step - loss: 2.5566 - acc:
0.2064 - val_loss: 2.5235 - val_acc: 0.2322
Epoch 2/10
```

```
352/351 [==============================] - 40s 114ms/step - loss: 2.2015 - acc:
0.3169 - val_loss: 2.2833 - val_acc: 0.2956
Epoch 3/10
...
Epoch 10/10
352/351 [==============================] - 41s 116ms/step - loss: 1.7438 - acc:
0.4550 - val_loss: 1.9590 - val_acc: 0.4070
```

Let's try going the other direction and instead of doubling the learning rate, let's cut it in half to 0.0005. What we see is that the performance degrades. It's likely the best learning rate for this model and dataset is around 0.001.

```
Epoch 1/10
352/351 [==============================] - 44s 125ms/step - loss: 2.5507 - acc:
0.2115 - val_loss: 2.4716 - val_acc: 0.2750
Epoch 2/10
352/351 [==============================] - 41s 116ms/step - loss: 2.2437 - acc:
0.3023 - val_loss: 2.2966 - val_acc: 0.3228
Epoch 3/10
...
Epoch 10/10
352/351 [==============================] - 41s 115ms/step - loss: 1.8893 - acc:
0.4136 - val_loss: 2.0811 - val_acc: 0.3776
```

Next

In the next part, we will cover the principal behind and best practices for transfer learning.

Part 12 - Pre-built Models & Transfer Learning

Overview

In this part, we will cover best practices for using pre-built models for training and transfer learning.

Pre-built Models

The **Keras** framework comes with a number of pre-built models, which you can use either as-is to train a new model, or modify and/or fine-tune for transfer learning. These models are based on best-in-class models for image classification, which have been award winning models in competitions like ImageNet. These model architectures are cited frequently in deep learning research papers.

Documentation on the pre-built Keras models are found here. The pre-built model architectures include:

Sequential CNN
: VGG16, VGG19

Residual CNN
: ResNet

Wide Layer CNN
: ResNeXt, Inception

Advanced CNN
: DenseNet, Xception, NASNet

Mobile
: MobileNet

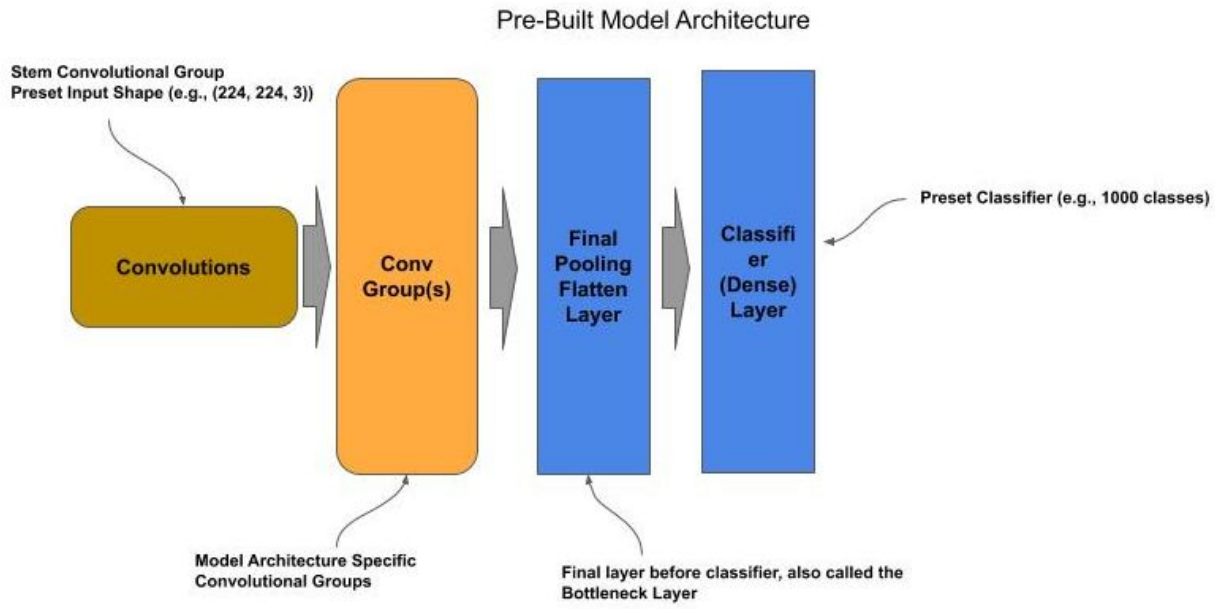
The pre-built Keras models are imported from the `keras.applications` module. Below are some examples:

```
from keras.applications import VGG16
from keras.applications import VGG19
from keras.applications import ResNet50
from keras.applications import InceptionV3
from keras.applications import InceptionResNetV2
from keras.applications import DenseNet121
from keras.applications import DenseNet169
from keras.applications import DenseNet201
from keras.applications import Xception
from keras.applications import NASNetLarge
```

```
from keras.applications import NASNetMobile
from keras.applications import MobileNet
```

The Base Model

By default, the pre-built models are complete but untrained (i.e., the weights and biases are randomly initialized). Each untrained pre-built model is configured for a specific input shape (see documentation), and number of output classes. In most cases the input shape is either (224, 224, 3) or (299, 299, 3). The models will also take input in channel first format as in (3, 224, 224) and (3, 299, 299). In most cases, the number of output classes is 1000, meaning the models can identify 1000 common image labels.



The pre-built models do not have an assigned loss function and optimizer. Prior to using them, one must issue the `compile()` method to assign the loss, optimizer and performance measurements.

```
from keras.applications import ResNet50

# Get a pre-built ResNet50 model
model = ResNet50()

# Compile the model for training
model.compile(loss='categorical_crossentropy', optimizer='adam',
              metrics=['accuracy'])
```

```
# Now train the model
```

Using the pre-built models in the manner above is pretty limited, considering not only is the input size fixed, but so is the number of categories for the classifier, which is 1000. It's unlikely whatever you need to do will use the default configuration.

Next we will explore some ways to configure the pre-built models to perform various tasks.

Pre-Trained

All of the pre-built models come with weights and biases pre-trained from *ImageNet 2012* dataset; which is a dataset of 1.2 million images across 1000 classes. If your need is simply to predict if an image is within the 1000 classes of *ImageNet* dataset, then one can use the pre-trained pre-built models as-is. The mapping of label identifiers to class names can be found in this Github repo. Examples of classes include things like bald eagle, toilet paper, strawberry, and balloon.

The code below uses the pre-built ResNet model pre-trained with ImageNet weights to classify (predict) an image of an elephant, where:

1. The `preprocess_input()` method will preprocess the image according to the method used by the pre-built ResNet model.
2. The `decode_predictions()` method will map label identifiers back to the class name.
3. The pre-built ResNet model is instantiated with Imagenet weights.
4. An image of an elephant is read in by openCV and then resized to (224, 224) to fit the input shape of the model.
5. The image is then preprocessed using the model's `preprocessed_input()` method.
6. The image is then reshaped into a batch.
7. The image is then classified by the model using the `predict()` method.
8. The top 3 predicted labels are then mapped to their class names using `decode_predictions()` and printed. In this example, one might see 'African Elephant' as the top prediction.

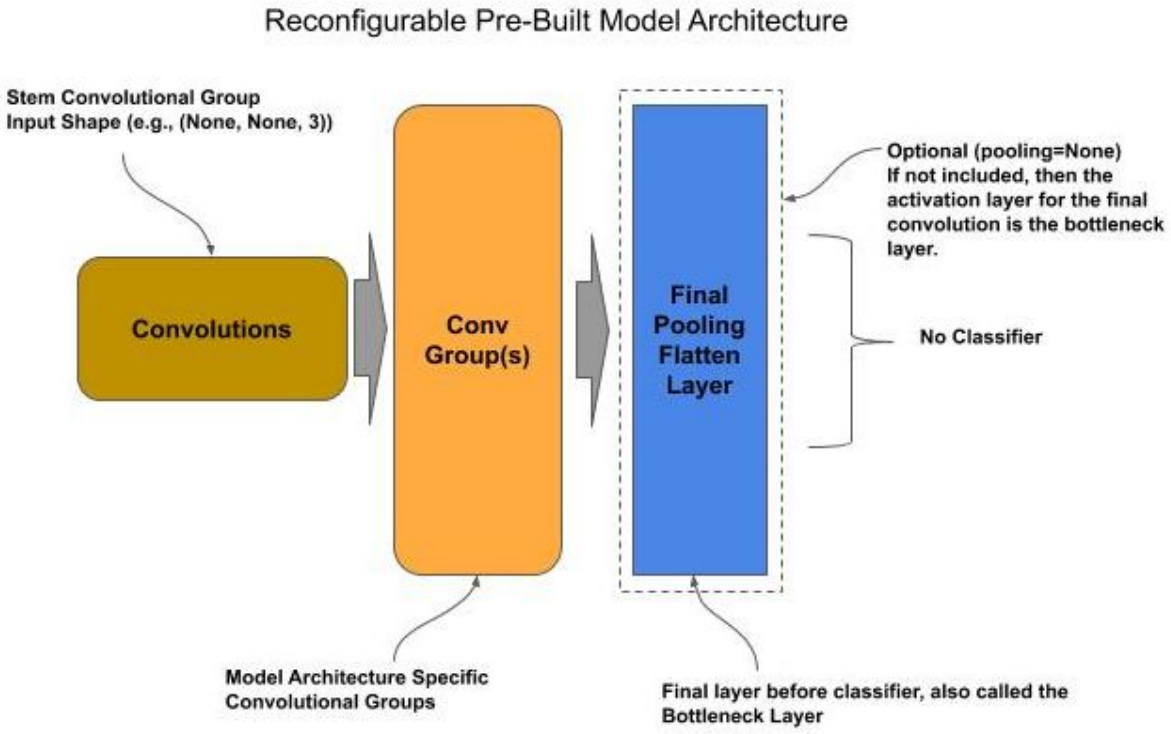
```
from keras.applications import ResNet50
from keras.applications.resnet import preprocess_input, decode_predictions

# Get a pre-built ResNet50 model
model = ResNet50(weights='imagenet')

# Read the image into memory as a numpy array
image = cv2.imread('elephant.jpg', cv2.IMREAD_COLOR)
```

```
# Resize the image to fit the input shape of ResNet model
image = cv2.resize(image, (224, 224), cv2.INTER_LINEAR)

# Preprocess the image using the same image processing used by the pre-built model
image = preprocess_input(image)

# Reshape from (224, 224, 3) to (1, 224, 224, 3) for the predict() method
image = image.reshape((-1, 224, 224, 3))

# Call the predict() method to classify the image
predictions = model.predict(image)

# Display the class name based on the predicted label using the decode function for
# the built-in model.
print(decode_predictions(preds, top=3))
```

New Classifier

The final classifier layer in all the pre-built models can be removed and replaced with a new classifier. The new classifier can then be used to train the pre-built model for a new dataset and set of classes. For example, if you had a dataset of twenty different classes of noodle dishes, you would simply remove the existing classifier layer, replace it with a new twenty node classifier layer, compile the model and train it.

In all the pre-built models, the classifier layer is referred to as the top layer. When instantiating an instance of a pre-built model, you would set the parameter `include_top` to `False` to get an instance without the classifier. Additionally, when the `include_top=False`, one can also reset the input shape of the model with the parameter `input_shape`.



As for the input shape, the documentation for the pre-built models has a limitation on the minimum input shape size. For most models, this is 32 x 32. I generally don't advise using the pre-built models in this manner, because for most of these architectures, the final feature maps before the global average pooling layer (i.e., last layer before classifier) will be 1x1 (single pixel) feature maps --essentially losing all spatial relationships. Though, even down to this size, researchers have found that when used with CIFAR-10 and CIFAR-100, which are 32 x 32 images, they are able to find good hyperparameter settings before advancing to competition-grade (e.g., ImageNet) image datasets.

In the code below, we instantiate a pre-built ResNet model and replace it with a new classifier for 20 classes:

1. Remove the existing classifier with the parameter `include_top`.
2. Set the input shape to (100, 100, 3) with the parameter `input_shape`
3. Retain the final pooling/flattening layer (before classifier) as a global average pooling layer with the parameter `pooling`.
4. Add a dense layer with twenty nodes and a softmax activation function.
5. Compile the model

```
from keras.applications import ResNet50
from keras.layers import Dense
```

```
# Get a pre-built model for input shape (100,100,3) and without the classifier
model = ResNet50(include_top=False, input_shape=(100, 100, 3), pooling='avg')

# Add a classifier for 20 classes
outputs = Dense(20, activation='softmax')(model.output)
model = Model(model.input, outputs)

# Compile the model for training
model.compile(loss='categorical_crossentropy', optimizer='adam',
              metrics=['accuracy'])

# Now train the model
```

For most of these models, the final layer preceding the classifier is a global average pooling layer. This layer acts as both a final pooling layer for the feature maps and a flatten operation (convert to 1D vector). In some cases, one might want to replace this layer with one's own custom final pooling/flatten layer. In this case, one either specifies the parameter `pooling` to `None` or not specify it, which is the default setting.

In the code example below, we replace the final pooling/flattening layer with a flattening (no pooling) layer.

```
from keras.applications import ResNet50
from keras.layers import Dense, Flatten
from keras import Model

# Get a pre-built model for input shape (100,100,3) and without the classifier
model = ResNet50(include_top=False, input_shape=(100, 100, 3), pooling=None)

# Flatten the Feature Maps into a 1D vector
output = Flatten()(model.output)

# Add a classifier for 20 classes
output = Dense(20, activation='softmax')(output)

# Compile the model for training
model = Model(model.input, output)
model.compile(loss='categorical_crossentropy', optimizer='adam',
              metrics=['accuracy'])

# Now train the model
```

Transfer Learning

Transfer learning is where one uses pretrained models for one task and retrains the classifier and/or fine-tune layers for a new task, similar to what was discussed above in the subsection on *New Classifier*.

There are two general approaches to transfer learning:

- Similar Tasks (Image Domains)
- Distinct Tasks

Similar Tasks

When deciding on the approach, we look at the similarity of the source (pre-trained) domain of images and the destination (new) domain. The more similar, the more of the existing bottom layers we can reuse without retraining. For example, if one had a model trained on fruits, it's likely that all of the bottom layers of the pretrained model can be reused without retraining for building a new model to recognize vegetables.

That is, we are assuming that the coarse features learned at the bottom-most layers will be the same for the new classifier, and learning finer details from the coarser details in the middle and final layers, all can be reused as-is, prior to entering the topmost layer(s) for classification.

When the source and destination domains have this high level of similarity, we generally can replace the existing topmost classifier layer with a new classifier layer, freeze the lower layers and train only the classifier layer. Since we don't need to learn the weights/biases for the other layers, we can generally train a model for the new domain with substantially less data and fewer epochs.

While having more data is always better, transfer learning between similar source and destination domains provides the ability to train with substantially smaller datasets. The two best practices for the minimum size of the dataset are:

- Each class (label) is 10% as big as in the source dataset.
- Each class (label) has at least 100 images.

In contrast to the method shown for the *New Classifier*, we modify the code to freeze all the layers preceding the topmost classifier layer prior to training. Freezing prevents the weights/biases of these layer(s) from being updated (retrained) during training of the classifier (topmost) layer. In Keras, each layer has the property `freeze`, which defaults to `True`.

The code example below will:

1. Use a pre-built model with pre-trained weights/biases (ImageNet 2012),
2. Drop the existing classifier from the pre-built model (topmost layer).
3. Freeze the remaining layers.
4. Add a new classifier layer.
5. Train the model through transfer learning.

```
from keras.applications import ResNet50
from keras.layers import Dense
from keras import Model

# Get a pre-trained/pre-built model without the classifier and retain the global
# average pooling # layer following the final convolution (bottleneck) layer
model = ResNet50(include_top=False, pooling='avg', weights='imagenet')

# Freeze the weights of the remaining layer
for layer in model.layers:
    layer.trainable = False

# Add a classifier for 20 classes
output = Dense(20, activation='softmax')(model.output)

# Compile the model for training
model = Model(model.input, output)
model.compile(loss='categorical_crossentropy', optimizer='adam',
              metrics=['accuracy'])

# Now train the model
```

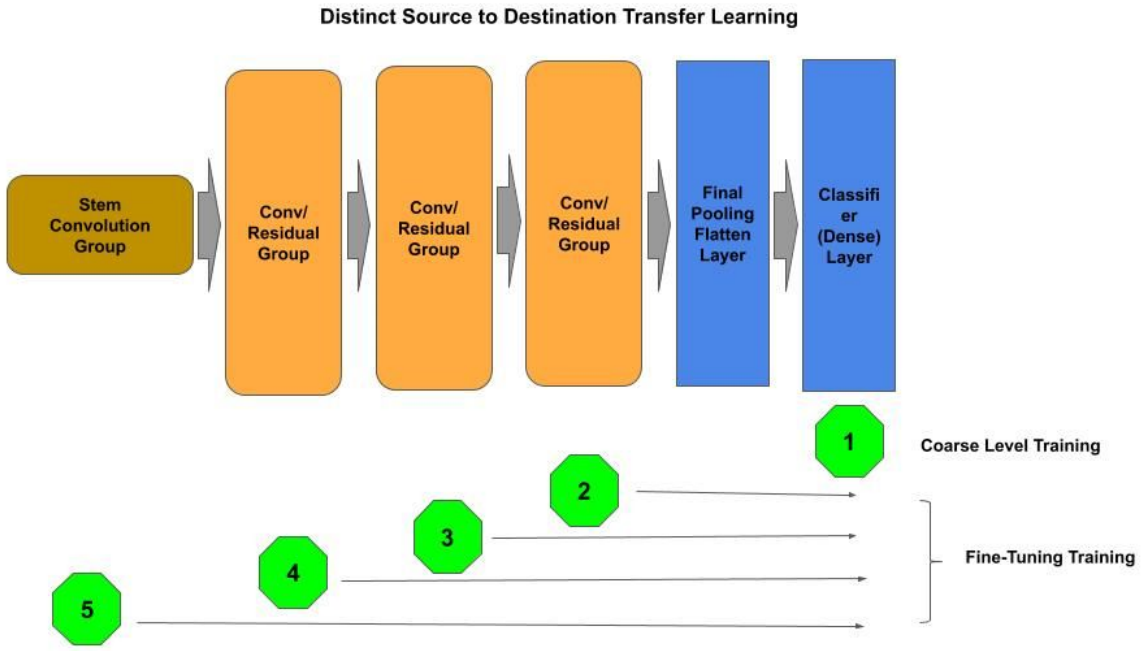
Note in the above code example, we retained the original input shape (i.e., 224, 224, 3). In practice, if we changed the input shape the pre-existing trained weights/biases won't match the resolution of the feature extraction they were trained on. In this case, it's better to handle this as a distinct task case.

Distinct Task

When the source and destination domain of the image datasets are non-similar, one starts with the same steps as in the similar task approach above, but then follows up with fine-tuning the bottom layers. The steps in this approach generally are:

1. Add a new classifier layer and freeze the remaining layer.
2. Train the new classifier layer for the target number of epochs.
3. Repeat for fine-tuning:

    a. Unfreeze the next bottom-most convolutional group (moving in direction of top to bottom)
    b. Train for a few epochs to fine-tune.
4. Once the convolutional groups are fine-tuned:
    a. Unfreeze the convolutional stem group
    b. Train for a few epochs to fine-tune



The code example below demonstrates first a coarse training for the add classifier level, followed by fine-tuning of each convolutional group and finally the stem convolutional group, where:

1. The classifier layer is trained with 50 epochs.
2. Each convolutional group from top-most to bottom-most, along with predecessors is trained for 5 epochs.
3. The stem convolutional (and hence whole model) is trained for an additional 5 epochs.

```
from keras.applications import ResNet50
from keras.layers import Dense
from keras import Model
import keras.layers

# Get a pre-trained/pre-built model without the classifier and retain the global
# average pooling # layer following the final convolution (bottleneck) layer
model = ResNet50(include_top=False, pooling='avg', weights='imagenet')
```

```
# Freeze the weights of the remaining layer
for layer in model.layers:
    Layer.trainable = False

# Add a classifier for 20 classes
output = Dense(20, activation='softmax')(model.output)

# Compile the model for training
model = Model(model.input, output)
model.compile(loss='categorical_crossentropy', optimizer='adam',
              metrics=['accuracy'])

# Train the classifier
model.fit(x_data, y_data, batch_size=32, epochs=50, validation_split=0.2)

stem = None # layer that is the convolutional layer for the stem group
groups = [] # the add layer for each convolutional group
conv2d = [] # the convolutional layers of a group

first_conv2d = True
for layer in model.layers:
        if type(layer) == layers.convolutional.Conv2D:
        # In ResNet50, the first Conv2D is the stem convolutional layer
        if first_conv2d == True:
                stem = layer
                first_conv2d = False
        # Keep list of convolutional layers per convolutional group
        else:
                conv2d.append(layer)
        # Each convolutional group in Residual Networks ends with a Add layer.
        # Maintain list in reverse order (top-most conv group is top of list)
        elif type(layer) == layers.merge.Add:
                groups.insert(0, conv2d)
                conv2d = []

# Unfreeze a convolutional group at a time (from top-most to bottom-most)
# And fine-tune (train) that layer
for i in range(1, len(groups)):
       # Unfreeze the convolutional layers in this conv/residual group
        for layer in groups[i]:
            layer.trainable = True

       # re-compile the model for training
       model.compile(loss='categorical_crossentropy', optimizer='adam',
                     metrics=['accuracy'])
```

```
        # Fine-tune train the convolutional group(s)
        model.fit(x_data, y_data, batch_size=32, epochs=5)

# Unfreeze the stem convolutional and do a final fine-tuning
stem.trainable = True
model.compile(loss='categorical_crossentropy', optimizer='adam',
              metrics=['accuracy'])
model.fit(x_data, y_data, batch_size=32, epochs=5, validation_split=0.2)
```

Note in the above example, when unfreezing layers for fine-tuning, the model must be re-compiled prior to issuing the next training session.

Domain Specific Weights

In the previous examples, we initialized the frozen layers of the model with weights learned from the *ImageNet 2012* dataset. One may desire to have pre-trained weights from a specific domain, other than the *ImageNet 2012*. In the code example below, we first train a ResNet50 pre-built architecture for a specific domain (e.g., produce) and then subsequently use the pretrained domain specific weights and initialization to train another ResNet50 model in a similar domain.

To do so, the code does the following:

1. Instantiate an uninitialized ResNet50 model without the classifier and pooling layer, which we designate as the base model.
2. Save the base model architecture for later reuse in transfer learning (`produce-model.json`).
3. Add a classifier (`Flatten` and `Dense` layers) and train for a specific (source) domain (e.g., produce).
4. Save the weights for the trained model (`produce-weights.h5`)
5. Load the base model architecture (`model-produce.json), which does not contain the classifier layer.
6. Initialize the base model architecture with the pretrained weights for the source domain (`model-produce.h5`).
7. Add a classifier for the new similar domain.
8. Train the model/classifier for the new similar domain.

```
from keras.applications import ResNet50
from keras import Model
from keras.layers import Dense, Flatten
from keras.models import load_model, model_from_json

model = ResNet50(include_top=False, pooling=None, input_shape=(100, 100, 3))
```

```
# save the base model
base_model = model.to_json()# Write the JSON string to a file
with open('produce-model.json', 'w') as f:
        f.write(base_model)

# Add classifier
output = Flatten(name='bottleneck')(model.output)
output = Dense(20, activation='softmax')(output)

# do training here

# save the model weights
model.save_weights('produce-weights.h5')

# Read the JSON string for the base model from a file
with open('produce-model.json', 'r') as f:
        base_model = f.read()

# Reuse the base model and trained weights
model = model_from_json(base_model)
model.load_weights('produce-weights.h5')

# Add classifier
output = Flatten(name='bottleneck')(model.output)
output = Dense(20, activation='softmax')(output)

# Compile the model
model = Model(model.input, output)
model.compile(loss='categorical_crossentropy', optimizer='adam',
metrics=['accuracy'])

# train the new model for a new dataset
```

Domain Transfer Weight Initialization

Another form of transfer learning is the transfer of domain specific weights to use as weight initialization in a model one will otherwise fully retrain. In this case, one is trying to improve on using an initializer based on a random weight distribution algorithm (e.g., Xavier for tanh and He-Normal for ReLU activation functions). After the hyperparameters have been selected, it's common practice to start training several instances of the model in parallel with different instances of the weight initialization. The rate of convergence and accuracy of the validation data are then monitored to identify instances that may converge on poorer local optima, which are then terminated. As training proceeds, eventually all but one of the training sessions is terminated, with the final training session running to completion.

The assumption is that the differences in the weight initialization may cause some training sessions to prematurely dive down into a poorer local optima. So one hedges their bet by initiating several training sessions, each with a different random distribution of the weight initialization.

Transferring of domain specific weights is a one-shot weight initialization approach. The presumption is to generate a set of weight initialization that is generalized enough that model training will lead to the best local (or global) optima. Ideally during initial training, the weights of the model will::

- Point in the general right direction for convergence.
- Be over generalized to prevent diving into an arbitrary local optima.
- Be used as the initialization weights for a single (one-shot) training session which will converge on the best local optima.

The steps for this form of weight initialization are:

1. Instantiate a model, with a random weight distribution (Xavier, He-Normal, etc).
2. Use high level of dropout and/or regularization to prevent fitting to the data.
3. Run a few epochs.
4. Save the weights.
5. Start a full training session using the saved weights.

```
from keras.applications import ResNet50
from keras import Model
from keras.layers import Dense, Flatten

# Instantiate base model with default weight initialization (i.e., He-Normal)
model = ResNet50(include_top=False, pooling=None, input_shape=(100, 100, 3))

# save the base model
base_model = model.to_json()# Write the JSON string to a file
with open('model.json', 'w') as f:
        f.write(base_model)

# Add a dropout layer and classifier to the base ResNet Model
output = layers.Dropout(0.75)(model.output)
output = layers.Dense(20, activation='softmax')(output)
model  = Model(model.input, output)

# do pre-training here for a few epochs (e.g., 5 epochs)

# save the model weights as the weight initializer
```

```
model.save_weights('weights-init.h5')

# Read the JSON string for the base model from a file
with open('model.json', 'r') as f:
        base_model = f.read()

# Reuse the base model
model = model_from_json(base_model)

# Initialize the weights using domain transfer weight initialization
model.load_weights('weights-init.h5')

# Add classifier without Dropout
output = Dense(20, activation='softmax')(model.output)

model = Model(model.input, output)
model.compile(loss='categorical_crossentropy', optimizer='adam',
metrics=['accuracy'])

# train the new model
```

Negative Transfer

In some cases one will find that transfer learning results in lower accuracy than training from scratch. That is, when using a pre-trained model to train a new model, the overall performance during training is less than what it would be if the model was not pre-trained.

This is referred to as negative transfer. In this case, the source and destination domains are so distinct that the learned weights for the source domain cannot be reused on the destination domain. Additionally, when the weights are reused the the model will not converge, and quite possibly diverge.

In general, one can usually spot negative transfer within five to ten epochs.

Next

In the next part, we will cover moving from exploration into a production environment with tf.keras and Tensorflow 2.0

Part 13 - Production

Overview

In this part, we will cover migrating from a preparation/exploratory phase to a production environment. This part will cover updating and replacing parts of your code from pure keras to **tf.keras** and **Tensorflow (TF) 2.0** ecosystem for a production environment.

tf.keras

The **tf.keras** is Tensorflow's integration of the Keras layers for constructing models using the Sequential and Functional API. This version of the API is optimized for Tensorflow and will execute (training) at faster speeds than pure Keras. The current best practice for using Keras in a production environment is to use tf.keras implementation.

Tensorflow (TF) 2.0

The *Tensorflow 2.0* ecosystem provides high speed performance in feeding, distributed training, quantization of models and model serving (in conjunction with *TFX Serving*). The current best practice for using Keras in a production environment is to use the TF 2.0 ecosystem.

Migrating Models to tf.keras

Migrating your pure Keras models to tf.keras is fairly straightforward for TF 1.14 version, while a few extra steps are required for TF 2.0, which is currently in alpha as of this writing. We will first cover the migration to TF 1.14.

The first step is to update your Tensorflow implementation to the last pre-2.0 release, which is version 1.14, using the `pip` on the command line (terminal or console).

```
cmd> pip install -U tensorflow
```

From a python shell (or notebook), you can verify that the current installed version of Tensorflow is 1.14 as follows:

```
import tensorflow as tf
# should output '1.13.1'
print(tf.__version__)
```

You generally only need to make the following changes:

- Change all the `import/from keras.xxx` statements to `import/from tensorflow.keras.xxx`.
- When checkpointing, tf.keras will default to the Tensorflow checkpointing format (yaml). Add the parameter `save_format='h5'` to save to Keras checkpointing format (HDF5).

In the example below is a simple DNN multi-class classifier that is trained with the Keras builtin dataset for MNIST.

```
# imports as pure Keras
from keras import Sequential, Input, optimizers
from keras.layers import Dense, Flatten
from keras.datasets import mnist
from keras.utils import to_categorical
import numpy as np

# Keras builtin dataset
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# Data preprocessing
x_train = ((x_train / 255.0).astype(np.float)).reshape(-1, 28, 28, 1)
x_test  = ((x_test  / 255.0).astype(np.float)).reshape(-1, 28, 28, 1)

# Label encoding
y_train = to_categorical(y_train)
y_test  = to_categorical(y_test)

# Build a DNN multi-class classifier
model = Sequential()
model.add(Flatten(input_shape=(28, 28, 1)))
model.add(Dense(512, activation='relu'))
model.add(Dense(512, activation='relu'))
model.add(Dense(10, activation='softmax'))

# Compile the model
model.compile(loss='categorical_crossentropy', optimizer=optimizers.Adam(),
              metrics=['accuracy'])

# Train the model
model.fit(x_train, y_train, epochs=10, batch_size=32,
          validation_data=(x_test, y_test))
```

In the example below, we made the following changes to migrate the model from pure Keras to tf.keras:

- Add import of tensorflow
- Import model classes from tf.keras

```
# imports as tf.keras
import tensorflow as tf
from tensorflow.keras import Sequential, Input, optimizers
from tensorflow.keras.layers import Dense, Flatten
from tensorflow.keras.datasets import mnist
from tensorflow.keras.utils import to_categorical
import numpy as np

# Keras builtin dataset
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# Data preprocessing
x_train = ((x_train / 255.0).astype(np.float)).reshape(-1, 28, 28, 1)
x_test  = ((x_test  / 255.0).astype(np.float)).reshape(-1, 28, 28, 1)

# Label encoding
y_train = to_categorical(y_train)
y_test  = to_categorical(y_test)

# Build a DNN multi-class classifier
model = Sequential()
model.add(Flatten(input_shape=(28, 28, 1)))
model.add(Dense(512, activation='relu'))
model.add(Dense(512, activation='relu'))
model.add(Dense(10, activation='softmax'))

# Compile the model
model.compile(loss='categorical_crossentropy', optimizer=Adam(),
metrics=['accuracy'])

# Train the model
model.fit(x_train, y_train, epochs=10, batch_size=32, validation_data=(x_test,
          y_test))
```

Tensorflow Dataset API

In the Tensorflow production environment, if the dataset is in-memory and small, then the recommendation is to use `numpy` multi-dimensional arrays as inputs, as in the above examples.

If you use large datasets, or distributed training, the recommendation is to use the Tensorflow Dataset API. The Dataset API provides the ability to create in-memory iterators using `tf.data.Dataset` which provides both maximum efficiency for feeding a neural network, as well as support for distributed training.

In the updated example below, we:

- Convert `x_train` and `y_train` to a `tf.data.Dataset` (dataset) using `tf.data.Dataset.from_tensor_slices()`.
- Convert `x_test` and `y_test` to a `tf.data.Dataset` (test_dataset).
- Set parameters for the dataset iterators: batch size = 32 (`.batch(32)`), shuffle (`.shuffle(512)`) and repeat (`.repeat()`).
- Update the `fit()` method using the dataset iterators.

Note, that we add setting the number of steps per epoch. We do this because `tf.data.dataset` is an iterator (not an array), and thus we need to tell the iterator how many batches there are in the training data. In this case, we divide the length of the training data by the `batch_size` to calculate the number of steps.

```
# imports as tf.keras
import tensorflow as tf
from tensorflow.keras import Sequential, Input, optimizers
from tensorflow.keras.layers import Dense, Flatten
from tensorflow.keras.datasets import mnist
from tensorflow.keras.utils import to_categorical
import numpy as np

# Keras builtin dataset
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# Data preprocessing
x_train = ((x_train / 255.0).astype(np.float)).reshape(-1, 28, 28, 1)
x_test  = ((x_test  / 255.0).astype(np.float)).reshape(-1, 28, 28, 1)

# Label encoding
y_train = to_categorical(y_train)
y_test  = to_categorical(y_test)

# Set the batch_size
```

```
# TF dataset iterator (training)
dataset = tf.data.Dataset.from_tensor_slices((x_train, y_train))
dataset = dataset.batch(batch_size)
dataset = dataset.shuffle(512)
dataset = dataset.repeat()

# TF dataset iterator (test)
test_dataset = tf.data.Dataset.from_tensor_slices((x_test, y_test))
test_dataset = test_dataset.batch(batch_size)

# Build a DNN multi-class classifier
model = Sequential()
model.add(Flatten(input_shape=(28, 28, 1)))
model.add(Dense(512, activation='relu'))
model.add(Dense(512, activation='relu'))
model.add(Dense(10, activation='softmax'))

# Compile the model
model.compile(loss='categorical_crossentropy', optimizer=tf.train.AdamOptimizer(),
              metrics=['accuracy'])

# Train the model
model.fit(dataset, epochs=10, steps_per_epoch=len(x_train)//batch_size,
          validation_data=test_dataset, validation_steps=len(x_test)//batch_size)
```

Repeat()

The `repeat()` method specifies the number of times to repeat feeding the entire dataset (epoch) during training. If no value is specified, then by default it will repeat forever. When using in conjunction with the Keras `fit()` method, where one specifies the number of epochs, no value should be specified for `repeat()`, as shown above.

Shuffle()

The `shuffle()` method specifies shuffling the dataset per epoch when feeding the dataset. By default, the dataset is not shuffled, thus each pass will see the same ordering of images in the batches. The `shuffle()` method takes a single parameter `buffer_size`. This specifies the number of elements (i.e., images) to randomly shuffle in place. A value of 1 would be no shuffling. If the dataset fits entirely in memory, then a value equal to the size of the dataset (i.e., len(x_train)) would do a uniform shuffling across the entire dataset. Otherwise, the dataset is shuffled a segment at a time. For example, a `shuffle(512)` will shuffle 512 images at a time.

Batch()

The `batch()` method specifies the batch size when feeding. In other words, the tf.data.dataset iterator will return this many examples (e.g., images) from the training data per iteration.

Prefetch()

The prefetch() method is used when training on multi-core CPU or in conjunction with a CPU+GPU. Image preprocessing steps and feeding during training are the I/O intensive operations and generally are most efficiently done on the CPU, while the matrix operations during training are most efficient on the GPU.

By default, the image preprocessing/feeding and training are done serially, which can lead to I/O bottlenecking on the GPU. When prefetch is set to 1, another thread will be run on the CPU to be generating the next mini-batch while the current mini-batch is being fed and trained. Below is a code example of setting the prefetch attribute; note how the methods can be chained together:

```
dataset = dataset.batch(32).shuffle(512).repeat().pretch(1)
```

Note, in our earlier example we only specified `batch()` for the test data. Since the test data is only forward feed once (no backward propagation), there is no need to specify `shuffle()` or `repeat()`.

TFRecords

Another alternate method to feeding data from disk to a model while training is to use Tensorflow's `TFRecord` file format. This format is a binary format that was originally designed for efficient serialization of structured data using Google's protocol buffer definitions. The fine details of the format are beyond the scope of this section; instead we will cover just how the format has been utilized for images for training CNNs.

The format has similarities to both a Python dictionary and JSON objects. A sample (e.g., image) and corresponding metadata are encapsulated within a `tf.Example` class object. A `tf.Example` object consists of a list of one or more `tf.train.Feature` entries. Each feature entry can be of one of three data types:

- `tf.train.ByteList`
- `tf.train.FloatList`
- `tf.train.Int64List`

The `tf.train.ByteList` is used for sequences of bytes or a string. A `tf.train.FloatList` is used for 32-bit (single precision) or 64-bit (double precision) float point numbers. A

`tf.train.Int64List` is used for both 32-bit and 64-bit signed and unsigned integers, and booleans.

There are several common practices for encoding image data into `TFRecord` format, of which share the following in common:

- A Feature entry for encoding of the image data.
- A Feature entry for the image shape (for reconstruction).
- A Feature entry for the corresponding label.

Below is a generic example of defining a `tf.train.Example` for encoding an image, where the /entries here/ are the dictionary entries for the image data and corresponding metadata:

```
example = tf.train.Example(features = { /entries here/ })
```

Let's start with the basics. In the code below we create a `TFRecord` object for an image which has not been decoded --i.e., in the compressed on-disk format. The benefit with this approach is that we use the least amount of disk space when `TFRecord` is stored. The drawback is that each time we read the `TFRecord` from disk while feeding the neural network during training, the image data must be uncompressed --which is a time vs. space trade-off.

In the code example below, we define a function for converting an on-disk image file (parameter `path`) and corresponding label (parameter `label`) as follows:

- The on-disk image is first read in and uncompressed into a raw bitmap using OpenCV method `cv2.imread()` to obtain the shape of the image (i.e., rows, columns, channels).
- The on-disk image is read in a second time using `tf.gfile.FastGFile()` in its original compressed format. Note, the `tf.gfile.FastGFile()` is equivalent to a `open()`, but if the image is stored on GCS bucket, the method is optimized for I/O read/write performance.
- A `tf.train.Example()` is instantiated with three dictionary entries for the features object:
    - image - a `BytesList` for the uncompressed (original on-disk data) image data.
    - label - a `Int64List` for the label value.
    - shape - a `Int64List` for the tuple (rows, height, channels) shape of the image.

In our example, if we assume that the size of the on-disk image is 24K bytes, then the size of the `TFRecord` file will be about 25K bytes.

```
import tensorflow as tf
import numpy as np
import sys
```

```
import cv2

def TFRecordImage(path, label):
        ''' The original compressed version of the image '''

        # read in (and uncompress) the image to get its shape
        image = cv2.imread(path)
        shape = image.shape

        # read in the image a second time for the original bytes (not uncompressed)
        with tf.gfile.FastGFile(path, 'rb') as fid:
            disk_image = fid.read()

        # make the record
        return tf.train.Example(features = tf.train.Features(feature = {
        'image': tf.train.Feature(bytes_list = tf.train.BytesList(value =
                                  [disk_image])),
        'label': tf.train.Feature(int64_list = tf.train.Int64List(value =
                                  [label])),
        'shape': tf.train.Feature(int64_list = tf.train.Int64List(value =
                                  [shape[0], shape[1], shape[2]]))
        }))

example = TFRecordImage('example.jpg', 0)
# output would be something like: 25,000
print(example.ByteSize())
```

In the next code example, we now store the uncompressed image data in the `TFRecord`. This has the benefit of only reading the image from disk once, and it does not need to be uncompressed each time the `TFRecord` is read from disk during training. The drawback is that the size of the record will be substantially larger than the on-disk version of the image. In the above example, assuming a 95% JPEG compression, the size of the `TFRecord` would be 500K bytes. Note, in the `BytesList` encoding of the image data, the `np.uint8` data format is retained.

```
def TFRecordImageUncompressed(path, label):
        ''' The uncompressed version of the image '''

        # read in (and uncompress) the image
        image = cv2.imread(path)
        shape = image.shape

        # make the record
        return tf.train.Example(features = tf.train.Features(feature = {
        'image': tf.train.Feature(bytes_list = tf.train.BytesList(value =
```

```
                                [image.tostring()])),
        'label': tf.train.Feature(int64_list = tf.train.Int64List(value =
                                [label])),
        'shape': tf.train.Feature(int64_list = tf.train.Int64List(value =
                                [shape[0], shape[1], shape[2]]))
        }))

example = TFRecordImageUncompressed('example.jpg', 0)
# output would be something like: 500,000
print(example.ByteSize())
```

In our final code example, we first normalize the pixel data (i.e., dividing by 255) and store the normalized image data. The advantage to this method is that we do not need to normalize the pixel data each time the `TFRecord` is read from disk during training. The drawback is that now the pixel data is stored as a `np.float32`, which is four times bigger than the corresponding `np.uint8`. Assuming the same above image example, the size of the `TFRecord` will now be 2M bytes.

```
def TFRecordImageNormalized(path, label):
        ''' The normalized version of the image '''

        # read in (uncompress) the image and normalize the pixel data
        image = (cv2.imread(path) / 255.0).astype(np.float32)
        shape = image.shape

        # make the record
        return tf.train.Example(features = tf.train.Features(feature = {
        'image': tf.train.Feature(bytes_list = tf.train.BytesList(value =
                                [image.tostring()])),
        'label': tf.train.Feature(int64_list = tf.train.Int64List(value =
                                [label])),
        'shape': tf.train.Feature(int64_list = tf.train.Int64List(value =
                                [shape[0], shape[1], shape[2]]))
        }))

example = TFRecordImageNormalized('example.jpg', 0)
# output should be something like: 2,000,000
print(example.ByteSize())
```

Now that we have constructed a `TFRecord` in memory, the next step is to write the record to disk. To maximize the efficiency of writing to and reading back from on-disk storage, the records are serialized to a string format for storing in Google's `protocol buffer` format. In the code example below, the `tf.python_io.TFRecordWriter` is an object which will write a serialized record to a file in Google's `protocol buffer` format. It is also a common convention when

writing a `TFRecord` to disk, to use the suffix `.tfrecord` in the file name.

```
# Create a writer for writing a TFRecord in protocol buffer format
with tf.python_io.TFRecordWriter('example.tfrecord') as writer:
        writer.write(example.SerializeToString())
```

A on-disk `.tfrecord` file may contain multiple `TFRecord` objects. Below is a code example of writing multiple serialized `TFRecords` to a `.tfrecord` file:

```
with tf.python_io.TFRecordWriter('example.tfrecord') as writer:
        # examples is a list of TFRecords, one per image
        for example in examples:
            writer.write(example.SerializeToString())
```

The next code example demonstrates how to read each `TFRecord` from a `.tfrecord` file in sequential order. In the example below we assume that the file example.tfrecord contains multiple serialized `TFRecords`. The `tf.python_io.record_interator()` creates an iterator object, that when used in a for statement will read into memory each serialized `TFRecord` in sequential order. The method `ParseFromString()` is used to deserialize the data into an in-memory `TFRecord` format.

```
# create an iterator for iterating through TFRecords in sequential order
iterator = tf.python_io.tf_record_iterator('example.tfrecord')
for record in iterator:
        # each record is read in as a serialized string
        example = tf.train.Example()
        # convert the serialized string to a TFRecord
        example.ParseFromString(record)
```

Alternatively, we can read and iterate through a set of `TFRecord` from a `.tfrecord` file using the `tf.data.TFRecordDataset` class. In the code example below, we:

- Instantiate a `tf.data.TFRecordDataset` object as an iterator for the on-disk records.
- Define the dictionary `feature_description` to specify how to deserialize the serialized `TFRecord`.
- Define the helper function `_parse_function()` for taking a serialized `TFRecord` (proto) and deserializing it using the dictionary `feature_description`.
- Use the `map()` method to iteratively deserialize each `TFRecord`.

```
# create an iterator for the on-disk dataset
```

```
dataset = tf.data.TFRecordDataset('example.tfrecord')

# create a dictionary description for deserializing a TFRecord
feature_description = {
    'image': tf.FixedLenFeature([], tf.string),
    'label': tf.FixedLenFeature([], tf.int64),
    'shape': tf.FixedLenFeature([], tf.int64),
}

def _parse_function(proto):
    ''' parse the next serialized TFRecord using the feature description '''
    return tf.parse_single_example(proto, feature_description)

parsed_dataset = dataset.map(_parse_function)
```

If we print `parsed_dataset`, the output should be:

```
<MapDataset shapes: {image: (), shape: (), label: ()}, types: {image: tf.string,
shape: tf.int64, label: tf.int64}>
```

tf.keras Saving/Restoring Model Weights

In Keras, model weights are saved in a HDF5 file format. In the tf.keras implementation, by default they are saved in Tensorflow's checkpoint format.

```
# Build a DNN multi-class classifier
model = Sequential()
model.add(Flatten(input_shape=(28, 28, 1)))
# lines deleted for brevity
model.add(Dense(10, activation='softmax'))

# Compile the model
model.compile(loss='categorical_crossentropy', optimizer=Adam(),
              metrics=['accuracy'])

# Train the model
model.fit(dataset, epochs=10, steps_per_epoch=len(x_train)//32,
          validation_data=val_dataset, validation_steps=len(x_test)//32)

# save the model weights in TF checkpoint format.
model.save_weights('my-model-weights')

# later restore the weights
```

```
mode.load_weights('my-model-weights')
```

To save in Keras HDF5 file format style, add the parameter `save_format='h5'` to the `save_weights()` method:

```
# save the model weights in Keras HDF5 format
model.save_weights('my-model-weights.h5', save_format='h5')

# later restore the weights
mode.load_weights('my-model-weights.h5')
```

Checkpointing and other Callbacks

When using tf.keras, one uses the tf.keras.callbacks methods for specifying callbacks (such as checkpointing or early stopping) in place of pure Keras methods for callbacks. The builtin tf.keras callbacks are:

- `tf.keras.callbacks.ModelCheckpoint` - for saving checkpoints during training.
- `tf.keras.callbacks.EarlyStopping` - for early stopping during training.
- `tf.keras.callbacks.LearningRateScheduler` - fine-grain control of dynamically changing the learning rate during training.
- `tf.keras.callbacks.TensorBoard` - Monitor the training using TensorBoard.

In the code example below, a checkpoint callback is declared for each epoch, where the checkpoint data will be stored in a file with prefix 'weights.' followed by the epoch number.

```
# create a checkpoint for each epoch
checkpoint = tf.keras.callbacks.ModelCheckpoint('weights.{epoch:02}')

model.fit(dataset, epochs=10, steps_per_epoch=len(x_train)//32,
          validation_data=val_dataset, validation_steps=len(x_test)//32,
          callbacks=[checkpoint])
```

tf.keras Saving/Restoring the Model

In Tensorflow 2.0, the recommendation is to save the tf.keras model and weights (when trained) in Tensorflow's `SavedModel` format. This format is compatible across all of Tensorflow components, including *TFX Serving.* By default, for backwards compatibility, the `tf.keras.save()` and `tf.keras.load_model()` methods.

To save in the `SavedModel` format and restore, one uses the `tf.keras.experimental.export_saved_model()` and `tf.experimental.load_from_saved_model()`, respectively.

```
# Save a (un)trained tf.keras model in SavedFormat
tf.keras.experimental.export_saved_model(model, 'my_saved_model')

# restore (load) a tf.keras model in SavedFormat.
model = tf.keras.experimental.load_from_saved_model('my_saved_model')
```

Converting Keras models to tf.estimators for Distributed Training (TF 1.1X)

Tensorflow's 1.1X (e.g., 1.13) Estimator-based models provide the following benefits:

- Estimator-based models can be trained on a local host or on a distributed multi-server environment without changing the model.
- Estimator-based models can be trained on CPUs, GPUs, or TPUs without recoding your model.

lc

- Create and compile a tf.keras model.
- Convert the tf.keras model to an Estimator model using `tf.keras.estimator.model_to_estimator()` method
- Define an input function for feeding the estimator during training.
    - In this example, we assume the training data is in a numpy multi-dimensional array, so we use the `tf.estimator.inputs.numpy_input_fn()` method.
    - Set the parameters for the training image data (`x`) and labels (`y`) to the corresponding numpy arrays `x_train` and `y_train`.
    - Set the number of epochs to 10 and set reshuffling the training data after each epoch.
- Call the estimator method `train()` to train the model, feeding 2000 mini-batches (`steps`) per epoch.

```
from tensorflow.keras import Sequential, Input, optimizers
from tensorflow.keras.layers import Dense, Flatten

# Build a DNN multi-class classifier
model = Sequential()
model.add(Flatten(input_shape=(28, 28, 1)))
# ... deleted for brevity

# Compile the model
```

```
model.compile(loss='categorical_crossentropy', optimizer=tf.train.AdamOptimizer(),
              metrics=['accuracy'])

# Create an estimator model from the compiled model
estimator = tf.keras.estimator.model_to_estimator(model)

# Make an input function for the estimator
train_input_fn = tf.estimator.inputs.numpy_input_fn(
    x=x_train,
    y=y_train,
    num_epochs=10,
    shuffle=True)

# Train the estimator version of the model
estimator.train(input_fn=train_input_fn, steps=2000)
```

tf.distribute.Strategy for Distributed Training (TF 2.0)

In Tensorflow 2.0, a tf.keras model can be used as-is for distributed training (w/o converting to tf.estimator), using `tf.distribute.Strategy`. There are five distributed strategies to choose from. For the purpose of this handbook, we will only cover the most common strategy `tf.distribute.Strategy`. This strategy will mirror (replicate) copies of the model to each compute device (e.g., GPU), feeds the replicated models with data parallelism, and implements a parameter server for updating the parameters on each model during backward probagation.

In the code example below, we:

- Set a `tf.distribute.Strategy` for `MirroredStrategy`.
- Use the `with` directive when building (constructing) and compiling the model.
- Then train the model (not required to be within the `mirrored_strategy.scope()`

```
mirrored_strategy = tf.distribute.MirroredStrategy()
with mirrored_strategy.scope():
  model = ... # build the model
  model.compile(...) # compile the model

model.fit(...)
```

Building Data Preprocessing into the Model (TF 2.0)

Another recommendation of TF 2.0, is to build the data preprocessing into the graph. Prior to TF 2.0, data preprocessing (e.g., normalization) occurred upstream from the model and was ran on the CPU. If the CPU was not sufficient in speed for feeding the data to the GPUs, the GPUs would be starved waiting for the next batch of data, and not run at their full capacity. TF 2.0 introduced new components to move data preprocessing into the graph, which are:

- Builtin data preprocessing as graph ops using Tensorflow Transform component (tft.transform).
- The @tf.function decorator for converting Python code into graph ops using `AutoGraph`.
- Subclassing of layers to add data preprocessing to the graph as a pre-stem operation.

Some of the benefits of moving data preprocessing into the graph are:

- Keeping the GPUs from being I/O bottleneck so they run at full capacity.
- Not having to re-implement the data preprocessing pipeline for prediction (inference), since it is built into the graph.
- The graph optimization is applied to the transform graph operations to further improve their speed.

Let's start by showing a basic template for subclassing layers and then explain it:

```
class NewLayer(layers.Layer):
    def __init__(self):
        super(NewLayer, self).__init__()
        self.my_vars = blash, blah

    def build(self, input_shape):
        """ Handler for building the layer """
        self.kernel = blah, blah

    def call(self, inputs):
        """ Handler for layer object as callable """
        outputs = do something with inputs
        return outputs
```

The first line in the above template class NewLayer(layers.Layer) indicates we want to create a new class object named NewLayer which is subclassed (derived) from the tf.keras layers class. This will give us a custom layer definition.

init() method (template)

This is the initializer (constructor) for the class object instantiation. We use the initializer to initialize layer specific variables.

build() method (template)

This method handles the building of the layer when the model is compiled. A typical action is to define the shape of the kernel (trainable parameters) and initialization of the kernel.

call() method (template)

This method handles calling the layer as a callable (function call) for execution in the graph.

Subclassing a Custom Layer

In the code below, we subclass a custom layer for doing preprocessing of the input, and where the preprocessing is converted to graph operations in the model.

The first line in the code class `Normalize(layers.Layer)` indicates we want to create a new class object named `Normalize` which is subclassed (derived) from the tf.keras layers class.

init() method (custom example)

Since we won't have any constants or variables to preserve, we don't have any need to add anything to this method.

build() method (custom example)

Our custom layer won't have any trainable parameters. We will tell the compile process to not set up any gradient descent updates on the kernel during training by setting the layers class variable self.kernel to None.

call() method (custom example)

This is where we add our preprocessing. The parameter inputs is the input tensor to the layer during training and prediction. A TF tensor object implements polymorphism to overload operators. We use the overloaded division operator, which will broadcast the division operation across the entire tensor --thus each element will be divided by 255.0.

Finally, we add the decorator @tf.function to tell **TensorFlow AutoGraph** to convert convert the Python code in this method to graph operations in the model.

```
class Normalize(layers.Layer):
    """ Custom Layer for Preprocessing Input """
    def __init__(self):
        """ Constructor """
        super(Normalize, self).__init__()

    def build(self, input_shape):
        """ Handler for Input Shape """
        self.kernel = None

    @tf.function
    def call(self, inputs):
        """ Handler for layer object is callable """
        inputs = inputs / 255.0
        return inputs
```

Let's build a model to train on the MNIST dataset. We will keep it really basic:

1. Use the Functional API method for defining the model.
2. Make the first layer of our model the custom preprocessing layer.
3. The remaining layers are a basic DNN for MNIST.

```
# Create the input vector for 28x28 MNIST images
inputs = Input((28, 28))

# The first layer is the preprocessing layer, which is bound to the input vector
x = Normalize()(inputs)

# Next layer, we flatten the preprocessed input into a 1D vector
x = Flatten()(x)

# Create a hidden dense layer of 128 nodes
x = Dense(128, activation='relu')(x)

# Create an output layer for classifying the 10 digits
outputs = Dense(10, activation='sigmoid')(x)

# Instantiate the model
model = Model(inputs, outputs)

# Compile the model
model.compile(loss='sparse_categorical_crossentropy', optimizer='adam', metrics=['acc'])
```

We will get the tf.keras builtin dataset for MNIST. The dataset is pre-split into train and test data. The data is separated into numpy multi-dimensional arrays for images and labels. The image data is not preprocessed --i.e., all the values are between 0 and 255. The label data is not one-hot-encoded --hence why we compiled with `loss='sparse_categorical_crossentropy'`

```
from tensorflow.keras.datasets import mnist

# Load the train and test data into memory
(x_train, y_train), (x_test, y_test) = mnist.load_data()
```

Let's now train the model (with the preprocessing built into the model graph) on the unpreprocessed MNIST data.

```
model.fit(x_train, y_train, epochs=10, batch_size=32, validation_split=0.1,
verbose=1)
```