```
try:
  import google.colab
  IN_COLAB = True
except:
  IN_COLAB = False

if IN_COLAB:
  !pip install wget
  #!pip install split-folders
  import wget
  import zipfile
  #import split-folders

import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
import tensorflow as tf
import keras
from keras.layers import Dense, Flatten, Normalization, Dropout, Conv2D, MaxPooling2D, RandomFlip, RandomRotation, RandomZoom, BatchNormalization, Activati
from keras.models import Sequential
from keras.losses import SparseCategoricalCrossentropy, CategoricalCrossentropy
from keras.callbacks import EarlyStopping
from keras.utils import np_utils
from keras import utils
import os
import matplotlib.pyplot as plt
```

```
    Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
    Requirement already satisfied: wget in /usr/local/lib/python3.9/dist-packages (3.2)
```

Data Pipelines and Dealing with Larger Data Efficiently

When dealing with larger amounts of data in our neural networks we need some tools to manage the data pipeline. We have started to look at a couple of these in the image generators and the datasets from directories. We can build on the dataset specifically to create data pipelines that both apply any data preparation steps and load our data efficiently. The TensorFlow documentation has a great article on the details of pipeline speed with timed examples and visuals of the process, Efficiently loading data from disk.

Note: in the project stuff I suggested using either generators or datasets for performance reasons, as I read more I found that the speed difference between the two has actually become really large, with datasets being much faster. We'll focus only on those here. Some articles said it is up to 30 times faster, which is pretty massive. Even if the difference isn't near that much, it is substantial, so the datasets will be more efficient for larger data.

```
# Helper to plot loss
def plot_loss(history):
  plt.plot(history.history['loss'], label='loss')
  plt.plot(history.history['val_loss'], label='val_loss')
  plt.legend()
  plt.grid(True)
  plt.show()

def plot_acc(history):
  plt.plot(history.history['accuracy'], label='accuracy')
  plt.plot(history.history['val_accuracy'], label='val_accuracy')
  plt.legend()
  plt.grid(True)
  plt.show()
```

```
# Download and Unzip Data
ROOT_DIR = "/content/simpsons_dataset"

def bar_custom(current, total, width=80):
    print("Downloading: %d%% [%d / %d] bytes" % (current / total * 100, current, total))

zip_name = "simpsons.zip"

url = "https://jrssbcrsefilesnait.blob.core.windows.net/3950data1/simpsons.zip"

if not os.path.exists(zip_name):
    wget.download(url, zip_name, bar=bar_custom)

if not os.path.exists("/content/simpsons_dataset"):
    with zipfile.ZipFile(zip_name, 'r') as zip_ref:
            zip_ref.extractall()
!rm -rf "simpsons_dataset/simpsons_dataset"
```

Parameters

```
BATCH_SIZE = 8
BUFFER = 150
VAL_SPLIT = .2
IMG_SIZE = (256, 256, 3)
IM_SIZE = (256, 256)
EPOCHS = 20
SEED = 123
```

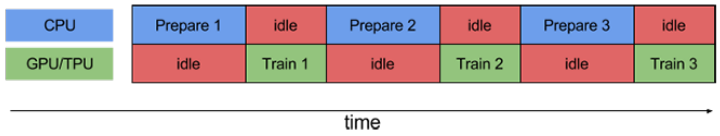
✓ 14m 57s completed at 2:53 PM

```
labels = []
for path, directories, files in label_walk:
    for directory in directories:
        labels.append(directory)
labels.sort()
print(labels)
NUM_CLASSES = len(labels)
print(NUM_CLASSES)

checkpoint_callback = tf.keras.callbacks.ModelCheckpoint("logs/weights.{epoch:02d}-{val_loss:.2f}.hdf5", monitor='val_accuracy', verbose=2, save_best_only=
```

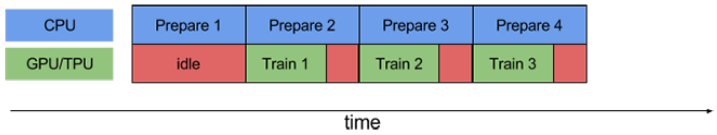
```
['abraham_grampa_simpson', 'agnes_skinner', 'apu_nahasapeemapetilon', 'barney_gumble', 'bart_simpson', 'carl_carlson', 'charles_montgomery_burns', 'ch
42
```

TF.Data Pipelines

The data pipeline setup in tensorflow is a little different than what we are used to in sklearn. Here, the big thing that our pipeline is going to do for us is to offer greater efficiency. Neural networks are generally used with very large datasets that are too large to fint in memory, like a dataframe that we are used to using. The tensorflow pipeline works to pull the data off of disk and into memory in a way that efficiently uses resources. The pipeline that we create will do a better job of ensuring that data is being prepared while the model is training, and is prepped and ready to go without delay, which can really matter with large datasets. The way that it does this is to do a better job of batching and processing the data, to ensure that we waste as little time idle as possible. Recall that when training models we normally use the GPU (graphics processor) becuase it is way faster, but most of the work outside of the fit method still relies on the CPU. The data pipelines work to make sure that the CPU can process and prepare data in parallel with the GPU doing the fitting, so as soon as one batch is finished processing, another is ready to go. Less delay means more of the run time spent processing rather than waiting.



From: **To:**



We have already used these datasets to load data, now we will look at some things that we can do to make the data pipeline more efficient when the data gets larger. These pipelines are a little more focused on the logistics of moving data around than on doing data cleanup like imputation that we often did in the sklearn stuff. When we are dealing with something like images, that type of data prep isn't really relevant. If we were applying this to structured data, cleanup steps could be applied through the .map() function, or we could just clean it before starting the modelling.

Construct the Datasets

We start by making the dataset objects - here we will load in image data from a directory structure, there are other methods to create datasets from other stuff like arrays.

```
train_ds = tf.keras.utils.image_dataset_from_directory(ROOT_DIR, validation_split=VAL_SPLIT, subset="training", seed=SEED, batch_size=None, shuffle=True, l
val_ds = tf.keras.utils.image_dataset_from_directory(ROOT_DIR, validation_split=VAL_SPLIT, subset="validation", batch_size=None, seed=SEED, shuffle=True, l
```

```
Found 20933 files belonging to 42 classes.
Using 16747 files for training.
Found 20933 files belonging to 42 classes.
Using 4186 files for validation.
```

Setup Preprocessing Pipeline

Now that we have the basics of the dataset created, we can work on the pipeline to deliver that data in the form we want it. There are several things that we can do to our dataset, some that we'll focus on here are:

- Cache - preload some data to speed the process.
- Map - apply a function to all the data, usefull to apply transformations like normalization.
- Batch - creates batches of data.
- Prefetch - retreives data early to eliminate delays.
- Shuffle - pulls data randomly to create batches.

Pipeline Syntax

The creation of the entire pipeline is simple from a coding perspective, we just chain all of the functions that we are going to use onto the dataset.

One key thing is that the things in the pipeline are done in order, sometimes this can matter. There are lots of potential combinations of actions that could be put together into a pipeline, so we can't make universal rules on the order, there are a few guidelines that we can use though:

- Prefetch should be last.

Vectorization

Vectorization is something that we've used before, but not discussed in detail. In short, vectorization is applying some action to an entire dataset without doing a loop. For example, if we wanted to take the square root of every item in a list we could write a loop that looks at each item and takes its square root, or we could use something like **map(lambda x:x*x,numbers)** to do the same thing. The end result is the same, but the system is able to compute the vector operation much more quickly. We saw an example of this in the neural network made from scratch - there are operations on the large matricies of weights calculated at each step of the process, those operations are done with vector operations like the dot product rather than a loop for efficiency. The same thing applies here, we want to do as much of the data prep as possible with vector operations. The python language has very fast internal operations for doing these vector operations efficiently, while things in a loop need to be done the "standard" way.

As a bit of a side note, this is an illustration of one of the examples of a downside of using an interpreted language like Python - i.e. a language that goes immediately from code to output, like what happens in our notebook when we click run. If you are using something like C++, the code is first compiled before it runs, or translated into machine code, the actual instructions that the computer understands. These compilers that do the translation are often quite smart, and will look to do things like optimize the code for efficiency. This can include profiling what a loop does and transforming it into a more efficient calculation. In general, code that is compiled is far faster than code that is interpreted, however that extra step makes development more cumbersome - the ability to just run one box of code at a time while we are creating models is really convinient. It is common for parts of the things that we call in Python, including Tensorflow itself, to have the performance critical bits to be written in a language like C++ for speed, then endpoints are provided so we can easily use it in Python. For something like machine learning this works quite well, most of the development is kind of the back and forth exploration and experimentation that benefits from the easy to update notebooks, while most of the processing time ends up being dominated by the training step, which can benefit from being implemented in another language that is faster, and isn't constrained by the code we write in Python.

Map and Batch

Mapping works here like it works anywhere else, we can apply a function to the entire dataset. For dataprep, this is useful, as if we need to do a transformation we can build it into the pipeline here. A big thing here is that we want to work on the data with vectorized operations, not loops. This map function will apply itself to the dataset, so we can do things like normalization here if we choose.

Batches work like we are used to, we can set the size of the data chuncks used in the training process. We set the dataset generator above to not batch the data so we can do it here.

```
rescale = tf.keras.Sequential([
    tf.keras.layers.Rescaling(1./255)
])


#train_ds = train_ds.map(lambda x, y: (rescale(x), y))
train_ds = train_ds.batch(BATCH_SIZE)

#val_ds = val_ds.map(lambda x, y: (rescale(x), y))
val_ds = val_ds.batch(BATCH_SIZE)
```

Cache and Shuffle

Cache just pulls data into memory early, so there is less delay to load it. In addition, you can optionally specify a file location for the cache location - while this probably won't help us, in a server environment you may have a RAM Disk, which is exactly what it sounds like, so you could potentially cache from a regular disk to the super fast ram disk. Here we won't cache, as the dataset is huge and this causes out-of-memory errors on Colab - if we had more RAM we could do this.

Shuffle just randomizes the order of the data, the buffersize controls how many items are shuffled at once. This is useful to ensure that the batches are not all from the same class, which can cause problems. The larger the size, the more random the shuffle, but the longer it takes and the more memory is consumed to do the shuffle.

```
train_ds = train_ds.shuffle(buffer_size=BUFFER)
```

Prefetch

Prefetch is what forces the system to do its data preparation work in parallel with the modelling work, so both the CPU and GPU can stay busy. We allow the autotune (below) to control how much is prepared in advance. This ensures that the training step of one batch, done in the GPU, is set to overlap with the processing and preparation of another batch, done by CPU. This means that the time between batches that the GPU is fitting is minimized, since we are loading and fitting many, many, many batches, making a small improvement here will really add up over the entire training process.

Autotune

One lesser known fact about neural networks is that T-Pain has done a large amount of research into efficiently loading data, even developing a tool predicably named autotune. In addition to making the voice of T-Pain resemble that of an angel, autotune works to make our data pipeline more efficient by monitoring some metrics on performance as the dataset works, and automatically making adjustments to improve things. The ramp-up process for the autotune to learn can impose some performance penalties on the early steps of training as the algorithm is analyzing the data, but once it learns an optimal set of values the efficiency will improve. This makes the autotune tool good for larger and longer training times, as the inital tuning time will become negligable as training progresses.

```
TPAIN = tf.data.AUTOTUNE
train_ds = train_ds.prefetch(buffer_size=TPAIN)
val_ds = val_ds.prefetch(buffer_size=TPAIN)
```

Finishing Pipelines and Handling Datasets

We can finish up by mirroring the steps above on our validation dataset.

Managing Resources

One thing that we can do with these datasets is set the parameters to limit resource use. Resource usage can be monitored on colab by clicking the RAM/CPU icon towards the top right, on a computer you could use the activity manager, task manager, or any other program that monitors RAM usage.

- GPU Ram is exceeded - likely can be addressed by the batch size and model size. One batch is processed at a time, so a smaller batch means less memory usage. It isn't the only factor, doing "stuff" (e.g. the matrix multiplication) with the massive matricies of weights also takes up memory, but that is somewhat harder to pinpoint other than smaller models use less RAM.
- System RAM is exceeded - this could be many things, the most likely cause is that too much data was loaded in for some operation that causes the RAM usage to spike.
    - Shuffle and prefetch options both load more data at a time the higher the limit is, we can lower them to limit RAM usage.

Unfortunately the autotune won't assure us that memory use limits don't get hit.

Memory Usage

This is one of the (likely) few times we really need to monitor RAM usage. In general, your computer is capable of swapping - moving stuff in and out of RAM and back to disk on the fly to ensure everything works. Here, swapping data to disk is so much slower that it is essentially impossible, so if we put 16.1GB in 16GB of RAM, everything dies. The most likely culprits are things that try to do something to all the data at once, it is likely that with larger datasets such actions aren't even possible. We can use the .map() function in the dataset to apply things to the dataset in a managed way. There are models, like some of the larger image or NLP ones, that can't run on normal consumer hardware because the RAM requirements on the GPU are too large. We can see something similar on a smaller scale with smart robotic toys, the ones that use something like a Raspberry Pi or other small computer as a programmable brain. Some of these can be AI driven by loading a neural network model that was trained elsewhere (your laptop) on that makes the speed and steering decisions based on what it sees and senses; that model can only be so large though as the small single board computers are limited in their capacity.

Performance Tuning

Since training models is highly dependent on speed, both for practicality concerns and to be able to test more possible models, eliminating bottlenecks and tuning performance is a massive topic.

Pulling Data

Getting some example data out of our dataset is a little different because they aren't a basic data structure like a dataframe or an array, so we can't just say "give me item 7". We need to approach getting data from the dataset similarly to how it provides did to a fit method, we ask for some data and the dataset produces one batch for us. We can do this with the "as_numpy_iterator" method, which returns an iterator. The iterator, well, iterates over the dataset, so to get some more data we can ask it for the next() piece of data.

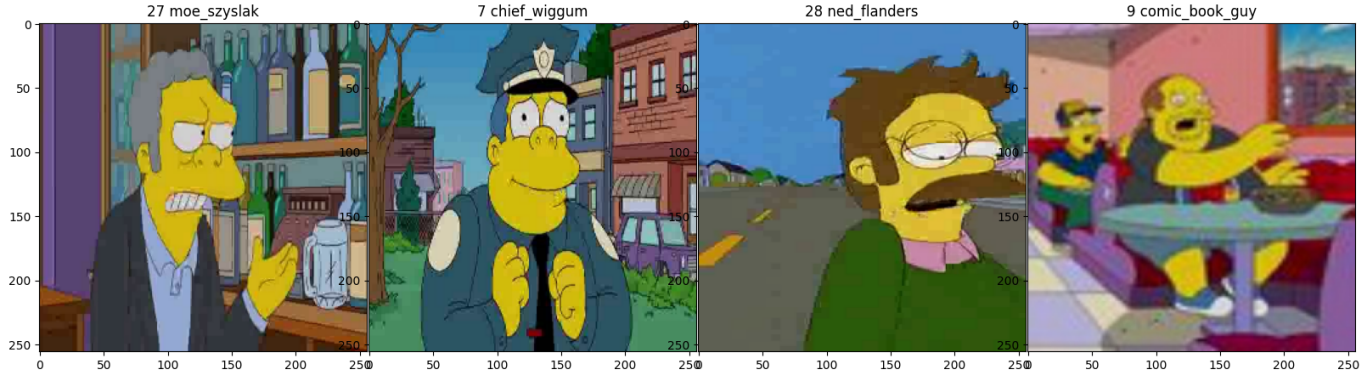
The dataset .next() call returns a batch of data in a 5D array. The first dimension is data vs label, data is 0, labels are 1. The next is the individual item index, and the last 3 are the dimensions of the image - height x width x color depth.

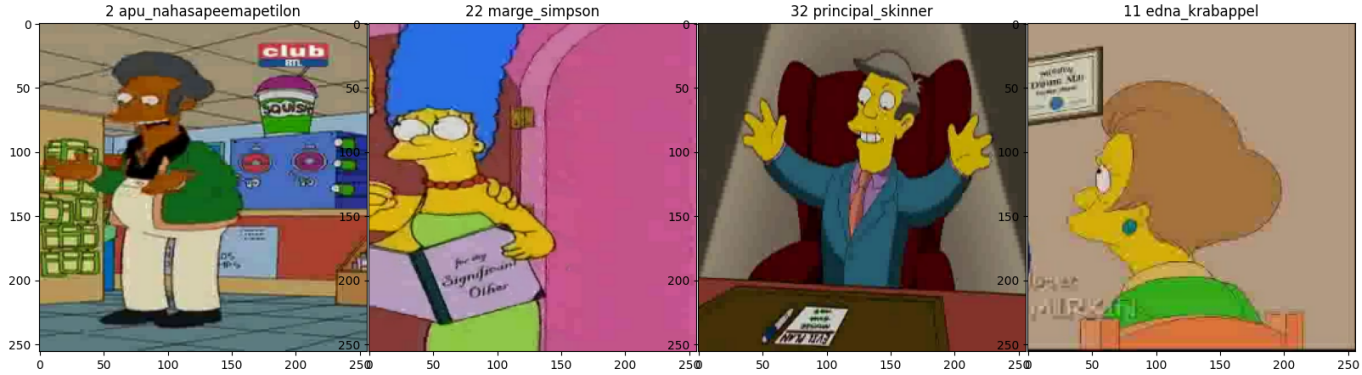
```
# Grab Some Data
some_data = train_ds.as_numpy_iterator()
sample_data = some_data.next()
plt.imshow(sample_data[0][0].astype("int"))
tmp_ind = np.argmax(sample_data[1][0])
print("Label Index:", tmp_ind, " Label:", labels[tmp_ind])
#print(sample_data)
```

```
Label Index: 27  Label: moe_szyslak
```


```
fig, ax = plt.subplots(2, 4, figsize=(20, 20))
plt.subplots_adjust(wspace=0.005)
for i in range(8):
    ax[i//4, i%4].imshow(sample_data[0][i].astype("int"))
    tmp_ind = np.argmax(sample_data[1][i].astype("int"))
    tmp_label = str(tmp_ind)+" "+labels[tmp_ind]
```

```
ax[i//4, i%4].set_title(tmp_label)
```





Modelling

Once the data pipelines are setup, using them is the same as always. Our datasets will handle all the things that we setup above all on their own, and will provide data to the fit method as it requires it. Since this training process may take a while, we will also write a checkpoint callback to save the weights every time we improve the model. The wonky stuff in the file name just assigns each set of weights saved with a label of their epoch and accuracy, a common way to log multiple sets of weights. As well, the adam optimizer gave me quite poor results on this one, so I tried a different one, rmsprop. RMSprop is another of the common optimizer algorithms and is generally good and efficient, just like adam. The effective difference is that rmsprop tends to change direction more quickly, while adam is more aggressive about "going down the gradient descent slope" quickly. Given that I previously got stuck in a minima (details below), that this change improves things makes some sense.

Advanced Activation

Some of activation functions outside of the standards like ReLU require some special considerations. ReLU is the most commonly used activation function on the hidden layers, as it generally performs well and is computationally efficient. The Keras docs have a note in there that the "advanced activation functions" like PReLU and LeakyReLU should not be added as an argument to a regular layer, but should be added as their own layer. This is because these advanced activations have something that is learned in training. The end effect in the structure and

functionality of the model doesn't change, it is simply to allow the way things are implemented in tensorflow to function correctly. If you recall the scratch-made neural network that we started with, those activations were were also added as a separate layer there.

```
#### Create Datasets
# I'm going to remake them so the map cna be applied without breaking the visualization
train_ds = tf.keras.utils.image_dataset_from_directory(ROOT_DIR, validation_split=VAL_SPLIT, subset="training", seed=SEED, batch_size=None, shuffle=True, la
val_ds = tf.keras.utils.image_dataset_from_directory(ROOT_DIR, validation_split=VAL_SPLIT, subset="validation", batch_size=None, seed=SEED, shuffle=True, la

TPAIN = tf.data.AUTOTUNE

train_ds = train_ds.map(lambda x, y: (rescale(x), y))
train_ds = train_ds.batch(BATCH_SIZE)
train_ds = train_ds.shuffle(buffer_size=BUFFER)
train_ds = train_ds.prefetch(buffer_size=TPAIN)

val_ds = val_ds.map(lambda x, y: (rescale(x), y))
val_ds = val_ds.batch(BATCH_SIZE)
val_ds = val_ds.prefetch(buffer_size=TPAIN)
```

```
Found 20933 files belonging to 42 classes.
Using 16747 files for training.
Found 20933 files belonging to 42 classes.
Using 4186 files for validation.
```

```
model = tf.keras.Sequential([
    tf.keras.layers.Conv2D(64, (3,3), padding="same",  input_shape=(256, 256, 3)),
    tf.keras.layers.LeakyReLU(),
    tf.keras.layers.MaxPooling2D(),
    tf.keras.layers.Conv2D(128, (3,3), padding="same"),
    tf.keras.layers.LeakyReLU(),
    tf.keras.layers.MaxPooling2D(),
    tf.keras.layers.Conv2D(256, (3,3), padding="same"),
    tf.keras.layers.LeakyReLU(),
    tf.keras.layers.MaxPooling2D(),
    tf.keras.layers.Conv2D(512, (3,3), padding="same"),
    tf.keras.layers.LeakyReLU(),
    tf.keras.layers.MaxPooling2D(),
    tf.keras.layers.Dropout(.2),

    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(512, kernel_regularizer="l2"),
    tf.keras.layers.LeakyReLU(),
    tf.keras.layers.Dropout(.3),
    tf.keras.layers.Dense(128, kernel_regularizer="l2"),
    tf.keras.layers.LeakyReLU(),
    tf.keras.layers.Dense(NUM_CLASSES, activation="softmax")
])
model.compile(
  #optimizer=tf.optimizers.Adam(learning_rate=0.001),
  optimizer="rmsprop",
  #loss=tf.losses.SparseCategoricalCrossentropy(from_logits=True),
  loss="categorical_crossentropy",
  metrics=['accuracy']
  )
model.summary()
```

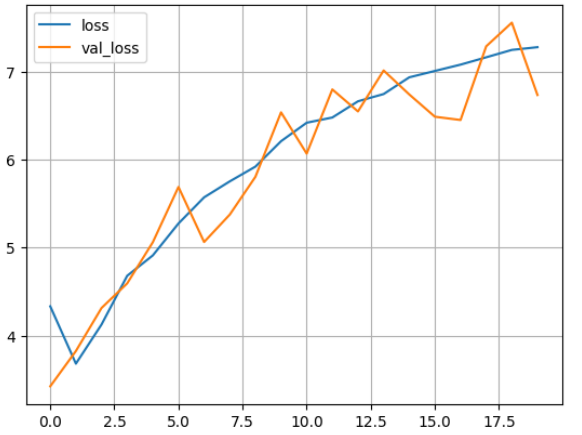
```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_2 (Conv2D)           (None, 256, 256, 64)      1792

 leaky_re_lu (LeakyReLU)     (None, 256, 256, 64)      0

 max_pooling2d_1 (MaxPooling  (None, 128, 128, 64)     0
 2D)

 conv2d_3 (Conv2D)           (None, 128, 128, 128)     73856

 leaky_re_lu_1 (LeakyReLU)   (None, 128, 128, 128)     0

 max_pooling2d_2 (MaxPooling  (None, 64, 64, 128)      0
 2D)

 conv2d_4 (Conv2D)           (None, 64, 64, 256)       295168

 leaky_re_lu_2 (LeakyReLU)   (None, 64, 64, 256)       0

 max_pooling2d_3 (MaxPooling  (None, 32, 32, 256)      0
 2D)

 conv2d_5 (Conv2D)           (None, 32, 32, 512)       1180160

 leaky_re_lu_3 (LeakyReLU)   (None, 32, 32, 512)       0

 max_pooling2d_4 (MaxPooling  (None, 16, 16, 512)      0
 2D)

 dropout_2 (Dropout)         (None, 16, 16, 512)       0

 flatten_1 (Flatten)         (None, 131072)            0

 dense_1 (Dense)             (None, 512)               67109376

 leaky_re_lu_4 (LeakyReLU)   (None, 512)               0
```

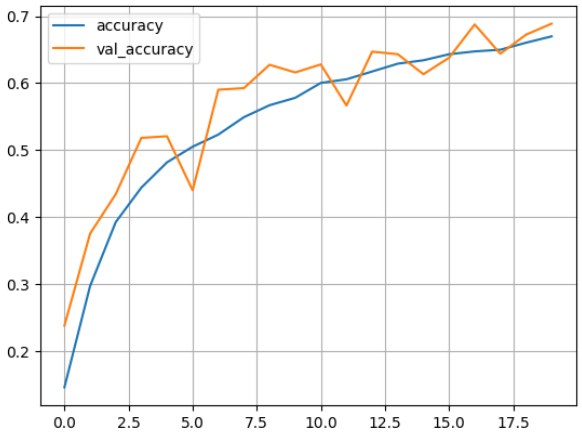
```
 dropout_3 (Dropout)         (None, 512)               0

 dense_2 (Dense)             (None, 128)               65664

 leaky_re_lu_5 (LeakyReLU)   (None, 128)               0

 dense_3 (Dense)             (None, 42)                5418

=================================================================
Total params: 68,731,434
Trainable params: 68,731,434
Non-trainable params: 0
_________________________________________________________________
```

```
# Train
log_m1 = model.fit(
    train_ds,
    validation_data=val_ds,
    epochs=EPOCHS,
    callbacks=[checkpoint_callback]
)
plot_loss(log_m1)
plot_acc(log_m1)
```

```
Epoch 1/20
2093/2094 [============================>.] - ETA: 0s - loss: 4.3348 - accuracy: 0.1456
Epoch 1: saving model to logs/weights.01-3.43.hdf5
2094/2094 [==============================] - 33s 14ms/step - loss: 4.3349 - accuracy: 0.1456 - val_loss: 3.4263 - val_accuracy: 0.2377
Epoch 2/20
2092/2094 [============================>.] - ETA: 0s - loss: 3.6825 - accuracy: 0.2969
Epoch 2: saving model to logs/weights.02-3.83.hdf5
2094/2094 [==============================] - 30s 14ms/step - loss: 3.6835 - accuracy: 0.2968 - val_loss: 3.8272 - val_accuracy: 0.3751
Epoch 3/20
2093/2094 [============================>.] - ETA: 0s - loss: 4.1285 - accuracy: 0.3923
Epoch 3: saving model to logs/weights.03-4.32.hdf5
2094/2094 [==============================] - 30s 14ms/step - loss: 4.1286 - accuracy: 0.3923 - val_loss: 4.3161 - val_accuracy: 0.4341
Epoch 4/20
2093/2094 [============================>.] - ETA: 0s - loss: 4.6816 - accuracy: 0.4438
Epoch 4: saving model to logs/weights.04-4.60.hdf5
2094/2094 [==============================] - 30s 14ms/step - loss: 4.6818 - accuracy: 0.4438 - val_loss: 4.5955 - val_accuracy: 0.5182
Epoch 5/20
2091/2094 [============================>.] - ETA: 0s - loss: 4.9129 - accuracy: 0.4818
Epoch 5: saving model to logs/weights.05-5.06.hdf5
2094/2094 [==============================] - 30s 14ms/step - loss: 4.9134 - accuracy: 0.4816 - val_loss: 5.0630 - val_accuracy: 0.5205
Epoch 6/20
2090/2094 [============================>.] - ETA: 0s - loss: 5.2770 - accuracy: 0.5049
Epoch 6: saving model to logs/weights.06-5.69.hdf5
2094/2094 [==============================] - 30s 14ms/step - loss: 5.2758 - accuracy: 0.5049 - val_loss: 5.6904 - val_accuracy: 0.4398
Epoch 7/20
2094/2094 [==============================] - ETA: 0s - loss: 5.5726 - accuracy: 0.5230
Epoch 7: saving model to logs/weights.07-5.07.hdf5
2094/2094 [==============================] - 30s 14ms/step - loss: 5.5726 - accuracy: 0.5230 - val_loss: 5.0653 - val_accuracy: 0.5901
Epoch 8/20
2093/2094 [============================>.] - ETA: 0s - loss: 5.7544 - accuracy: 0.5490
Epoch 8: saving model to logs/weights.08-5.38.hdf5
2094/2094 [==============================] - 30s 14ms/step - loss: 5.7540 - accuracy: 0.5491 - val_loss: 5.3764 - val_accuracy: 0.5925
Epoch 9/20
2090/2094 [============================>.] - ETA: 0s - loss: 5.9207 - accuracy: 0.5672
Epoch 9: saving model to logs/weights.09-5.81.hdf5
2094/2094 [==============================] - 30s 14ms/step - loss: 5.9216 - accuracy: 0.5669 - val_loss: 5.8056 - val_accuracy: 0.6273
Epoch 10/20
2091/2094 [============================>.] - ETA: 0s - loss: 6.2102 - accuracy: 0.5778
Epoch 10: saving model to logs/weights.10-6.54.hdf5
2094/2094 [==============================] - 30s 14ms/step - loss: 6.2103 - accuracy: 0.5780 - val_loss: 6.5380 - val_accuracy: 0.6159
Epoch 11/20
2091/2094 [============================>.] - ETA: 0s - loss: 6.4210 - accuracy: 0.6002
Epoch 11: saving model to logs/weights.11-6.07.hdf5
2094/2094 [==============================] - 30s 14ms/step - loss: 6.4201 - accuracy: 0.6002 - val_loss: 6.0690 - val_accuracy: 0.6278
Epoch 12/20
2092/2094 [============================>.] - ETA: 0s - loss: 6.4792 - accuracy: 0.6056
Epoch 12: saving model to logs/weights.12-6.80.hdf5
2094/2094 [==============================] - 30s 14ms/step - loss: 6.4793 - accuracy: 0.6057 - val_loss: 6.7983 - val_accuracy: 0.5664
Epoch 13/20
2092/2094 [============================>.] - ETA: 0s - loss: 6.6634 - accuracy: 0.6174
Epoch 13: saving model to logs/weights.13-6.55.hdf5
2094/2094 [==============================] - 30s 14ms/step - loss: 6.6635 - accuracy: 0.6172 - val_loss: 6.5482 - val_accuracy: 0.6469
Epoch 14/20
2090/2094 [============================>.] - ETA: 0s - loss: 6.7455 - accuracy: 0.6290
Epoch 14: saving model to logs/weights.14-7.01.hdf5
2094/2094 [==============================] - 30s 14ms/step - loss: 6.7457 - accuracy: 0.6289 - val_loss: 7.0137 - val_accuracy: 0.6431
Epoch 15/20
2091/2094 [============================>.] - ETA: 0s - loss: 6.9362 - accuracy: 0.6340
Epoch 15: saving model to logs/weights.15-6.74.hdf5
2094/2094 [==============================] - 30s 14ms/step - loss: 6.9358 - accuracy: 0.6340 - val_loss: 6.7412 - val_accuracy: 0.6132
Epoch 16/20
2091/2094 [============================>.] - ETA: 0s - loss: 7.0086 - accuracy: 0.6430
Epoch 16: saving model to logs/weights.16-6.49.hdf5
2094/2094 [==============================] - 30s 14ms/step - loss: 7.0081 - accuracy: 0.6430 - val_loss: 6.4882 - val_accuracy: 0.6378
Epoch 17/20
2091/2094 [============================>.] - ETA: 0s - loss: 7.0807 - accuracy: 0.6472
Epoch 17: saving model to logs/weights.17-6.45.hdf5
2094/2094 [==============================] - 30s 14ms/step - loss: 7.0799 - accuracy: 0.6473 - val_loss: 6.4510 - val_accuracy: 0.6875
Epoch 18/20
2091/2094 [============================>.] - ETA: 0s - loss: 7.1632 - accuracy: 0.6496
Epoch 18: saving model to logs/weights.18-7.29.hdf5
2094/2094 [==============================] - 30s 14ms/step - loss: 7.1631 - accuracy: 0.6498 - val_loss: 7.2877 - val_accuracy: 0.6438
```

```
Epoch 19/20
2094/2094 [==============================] - ETA: 0s - loss: 7.2472 - accuracy: 0.6602
Epoch 19: saving model to logs/weights.19-7.56.hdf5
2094/2094 [==============================] - 30s 14ms/step - loss: 7.2472 - accuracy: 0.6602 - val_loss: 7.5566 - val_accuracy: 0.6722
Epoch 20/20
2092/2094 [============================>.] - ETA: 0s - loss: 7.2786 - accuracy: 0.6697
Epoch 20: saving model to logs/weights.20-6.73.hdf5
2094/2094 [==============================] - 30s 14ms/step - loss: 7.2780 - accuracy: 0.6697 - val_loss: 6.7349 - val_accuracy: 0.6887
```

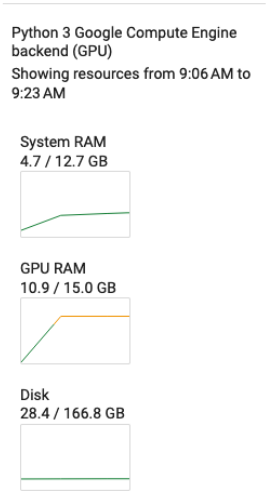




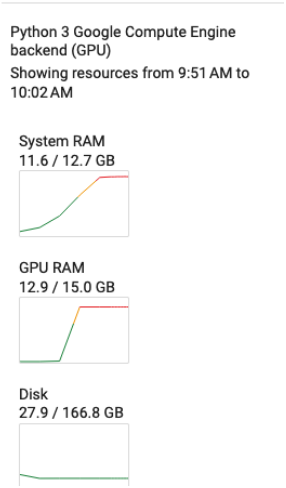
Setting up a Pipeline

These pipelines result in a balance of memory usage vs. speed - we can load a bunch more stuff in memory to make it faster, or we can endure losses in speed to save memory. Ideally, in a real world setting, we have some system (real or virtual) that we are using to train our models and we want to set things up to utilize those resources as fully as possible. On colab, the RAM we are allocated isn't massive, so we may not be able to lean really heavily on some of the functionality here to preload our data. If we were doing a similar task in a "real" scenario, it would be pretty easy for us to allocate or buy something like 64GB of RAM and really utilize it to ensure that both the CPU and GPU have work to do all the time.

Since RAM is likely the limiting factor when we are using colab, configuration of the pipeline also allows us to tailor the memory usage of our model to fit the constraints. For example, this is a snapshot of the RAM usage of this code with the buffer for shuffle set to 8, the prefetch suing autotune, and the batch size set to 8. We could bump these up a bit, and since the resource usage tends to flatten out once everything starts up it won't be too difficult to get our resource usage pretty close to the limit.



When I increased the shuffle buffer to 250, the resources now look like:



Maximizing the batch size when there is a lot of data to process will generally make the training process faster as there is less wasted capacity in the GPU on each batch. For larger datasets this is probably a larger concern than what we saw when we first touched on batch size - that we likely want it small for the best results. Even if a smaller batch performs better in a vacuum, faster epochs mean we can train and experiment more, which will likely outweigh any improvements that a small batch may bring. If we are dealing with images, a "large batch" probably won't be that large anyway. So, tl;dr, if using a large and slow model, fill the GPU with the largest batch it can take.

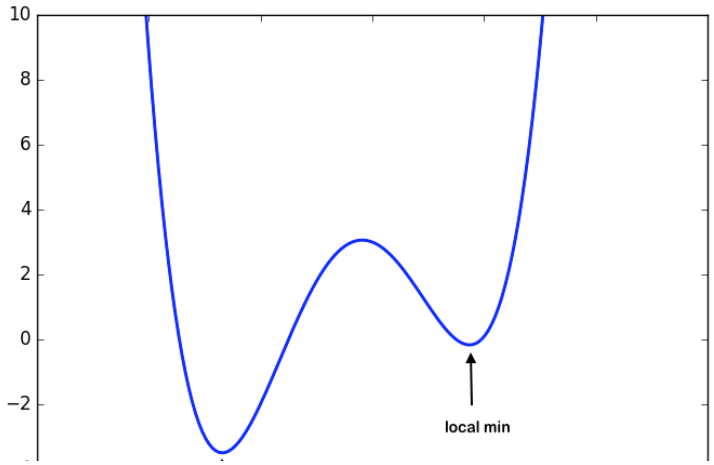
Local Minima - a Happy Accident!

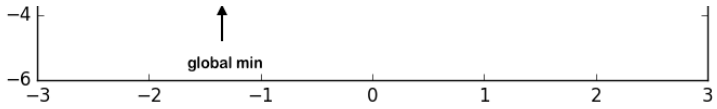
While I was putting this together I had one run that gave a really dramatic example of a local minima in the loss. Look at the loss value through the epochs here - our loss gets small, then explodes, then starts to get smaller again (though we don't have enough epochs here to see where we end up):

```
Epoch 1/10
524/524 [==============================] - ETA: 0s - loss: 229.0139 - accuracy: 0.3891
Epoch 1: saving model to logs/weights.01-4.25.hdf5
524/524 [==============================] - 113s 168ms/step - loss: 229.0139 - accuracy: 0.3891 - val_loss: 4.2456 - val_accuracy: 0.4909
Epoch 2/10
524/524 [==============================] - ETA: 0s - loss: 3.3890 - accuracy: 0.5023
Epoch 2: saving model to logs/weights.02-2.88.hdf5
524/524 [==============================] - 83s 155ms/step - loss: 3.3890 - accuracy: 0.5023 - val_loss: 2.8798 - val_accuracy: 0.4909
Epoch 3/10
524/524 [==============================] - ETA: 0s - loss: 2.5760 - accuracy: 0.5023
Epoch 3: saving model to logs/weights.03-2.49.hdf5
524/524 [==============================] - 83s 156ms/step - loss: 2.5760 - accuracy: 0.5023 - val_loss: 2.4869 - val_accuracy: 0.4909
Epoch 4/10
524/524 [==============================] - ETA: 0s - loss: 86751.5938 - accuracy: 0.3566
Epoch 4: saving model to logs/weights.04-18854.94.hdf5
524/524 [==============================] - 83s 155ms/step - loss: 86751.5938 - accuracy: 0.3566 - val_loss: 18854.9375 - val_accuracy: 0.4909
Epoch 5/10
524/524 [==============================] - ETA: 0s - loss: 4044.5935 - accuracy: 0.2694
Epoch 5: saving model to logs/weights.05-376.61.hdf5
524/524 [==============================] - 83s 155ms/step - loss: 4044.5935 - accuracy: 0.2694 - val_loss: 376.6081 - val_accuracy: 0.4899
Epoch 6/10
524/524 [==============================] - ETA: 0s - loss: 177.7321 - accuracy: 0.4238
Epoch 6: saving model to logs/weights.06-110.54.hdf5
524/524 [==============================] - 83s 155ms/step - loss: 177.7321 - accuracy: 0.4238 - val_loss: 110.5406 - val_accuracy: 0.4909
Epoch 7/10
524/524 [==============================] - ETA: 0s - loss: 102.2869 - accuracy: 0.4922
Epoch 7: saving model to logs/weights.07-98.14.hdf5
524/524 [==============================] - 83s 155ms/step - loss: 102.2869 - accuracy: 0.4922 - val_loss: 98.1358 - val_accuracy: 0.4858
Epoch 8/10
524/524 [==============================] - ETA: 0s - loss: 95.6780 - accuracy: 0.4958
Epoch 8: saving model to logs/weights.08-93.52.hdf5
524/524 [==============================] - 83s 155ms/step - loss: 95.6780 - accuracy: 0.4958 - val_loss: 93.5241 - val_accuracy: 0.4909
Epoch 9/10
524/524 [==============================] - ETA: 0s - loss: 91.1005 - accuracy: 0.4993
Epoch 9: saving model to logs/weights.09-88.64.hdf5
524/524 [==============================] - 83s 155ms/step - loss: 91.1005 - accuracy: 0.4993 - val_loss: 88.6371 - val_accuracy: 0.4893
Epoch 10/10
524/524 [==============================] - ETA: 0s - loss: 85.7476 - accuracy: 0.4986
Epoch 10: saving model to logs/weights.10-82.76.hdf5
524/524 [==============================] - 82s 154ms/step - loss: 85.7476 - accuracy: 0.4986 - val_loss: 82.7637 - val_accuracy: 0.4907
<keras.callbacks.History at 0x7fc4d49302e0>
```

+ Code + Text

There is a pretty good chance our visualized gradient descent process looked something like this:





Automated Machine Learning

Many of the things that we need to do to create ML models is somewhat automatable, particularly the parts around fitting and tuning a model. Experimenting with things like number of neurons in a neural network or different combinations of hyperparameters for any model is something that can be automated and many tools exist to do this. The goal of these tools is generally to make the process of creating machine learning models more accessible to people who don't have a lot of experience with the process. Current tools, and likely ones that will exist in the near future, aren't smart enough to totally replace all the work we need to do, but they can replace parts of it and they are getting better all the time. This is conjecture, but I'd anticipate that true breakthroughs in the capabilities of automated machine learning will explode as more and more services are moved to cloud services. The corporate market, filled with large amounts of data and few people who really analyze it, makes for a promising market, and the semi-standardization in the data that will come as more customers use cloud tools offered by one a few providers (AWS, Azure, Google) solves one of the biggest challenges to automating the process - variation. Companies forecasting sales, predicing loan defaults, or segmenting customers are all doing the same fundamental task, if the data is highly consistent coming out of the same systems, it makes it much easier to design an automated ML workflow that needs little to no human intervention. The machine learning bits can just be another service that the cloud providers can sell to their clients.

These automated ML systems are normally a nice and friendly wrapper around the same core libraries that we use to build things from scratch.

AutoKeras

We can take a peek at one automated machine learning library, AutoKeras. We won't spend a lot of time on it as it doesn't really do anything new or different, it just does the work for us.

```
if IN_COLAB:
    !pip install autokeras
    import autokeras as ak
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Requirement already satisfied: autokeras in /usr/local/lib/python3.9/dist-packages (1.1.0)
Requirement already satisfied: pandas in /usr/local/lib/python3.9/dist-packages (from autokeras) (1.4.4)
Requirement already satisfied: tensorflow>=2.8.0 in /usr/local/lib/python3.9/dist-packages (from autokeras) (2.12.0)
Requirement already satisfied: packaging in /usr/local/lib/python3.9/dist-packages (from autokeras) (23.0)
Requirement already satisfied: keras-nlp>=0.4.0 in /usr/local/lib/python3.9/dist-packages (from autokeras) (0.4.1)
Requirement already satisfied: keras-tuner>=1.1.0 in /usr/local/lib/python3.9/dist-packages (from autokeras) (1.3.4)
Requirement already satisfied: numpy in /usr/local/lib/python3.9/dist-packages (from keras-nlp>=0.4.0->autokeras) (1.22.4)
Requirement already satisfied: tensorflow-text in /usr/local/lib/python3.9/dist-packages (from keras-nlp>=0.4.0->autokeras) (2.12.0)
Requirement already satisfied: absl-py in /usr/local/lib/python3.9/dist-packages (from keras-nlp>=0.4.0->autokeras) (1.4.0)
Requirement already satisfied: requests in /usr/local/lib/python3.9/dist-packages (from keras-tuner>=1.1.0->autokeras) (2.27.1)
Requirement already satisfied: protobuf<=3.20.3 in /usr/local/lib/python3.9/dist-packages (from keras-tuner>=1.1.0->autokeras) (3.20.3)
Requirement already satisfied: kt-legacy in /usr/local/lib/python3.9/dist-packages (from keras-tuner>=1.1.0->autokeras) (1.0.4)
Requirement already satisfied: google-pasta>=0.1.1 in /usr/local/lib/python3.9/dist-packages (from tensorflow>=2.8.0->autokeras) (0.2.0)
Requirement already satisfied: tensorboard<2.13,>=2.12 in /usr/local/lib/python3.9/dist-packages (from tensorflow>=2.8.0->autokeras) (2.12.0)
Requirement already satisfied: astunparse>=1.6.0 in /usr/local/lib/python3.9/dist-packages (from tensorflow>=2.8.0->autokeras) (1.6.3)
Requirement already satisfied: typing-extensions>=3.6.6 in /usr/local/lib/python3.9/dist-packages (from tensorflow>=2.8.0->autokeras) (4.5.0)
Requirement already satisfied: keras<2.13,>=2.12.0 in /usr/local/lib/python3.9/dist-packages (from tensorflow>=2.8.0->autokeras) (2.12.0)
Requirement already satisfied: tensorflow-io-gcs-filesystem>=0.23.1 in /usr/local/lib/python3.9/dist-packages (from tensorflow>=2.8.0->autokeras) (0.3
Requirement already satisfied: jax>=0.3.15 in /usr/local/lib/python3.9/dist-packages (from tensorflow>=2.8.0->autokeras) (0.4.7)
Requirement already satisfied: flatbuffers>=2.0 in /usr/local/lib/python3.9/dist-packages (from tensorflow>=2.8.0->autokeras) (23.3.3)
Requirement already satisfied: h5py>=2.9.0 in /usr/local/lib/python3.9/dist-packages (from tensorflow>=2.8.0->autokeras) (3.8.0)
Requirement already satisfied: opt-einsum>=2.3.2 in /usr/local/lib/python3.9/dist-packages (from tensorflow>=2.8.0->autokeras) (3.3.0)
Requirement already satisfied: wrapt<1.15,>=1.11.0 in /usr/local/lib/python3.9/dist-packages (from tensorflow>=2.8.0->autokeras) (1.14.1)
Requirement already satisfied: termcolor>=1.1.0 in /usr/local/lib/python3.9/dist-packages (from tensorflow>=2.8.0->autokeras) (2.2.0)
Requirement already satisfied: tensorflow-estimator<2.13,>=2.12.0 in /usr/local/lib/python3.9/dist-packages (from tensorflow>=2.8.0->autokeras) (2.12.
Requirement already satisfied: libclang>=13.0.0 in /usr/local/lib/python3.9/dist-packages (from tensorflow>=2.8.0->autokeras) (16.0.0)
Requirement already satisfied: gast<=0.4.0,>=0.2.1 in /usr/local/lib/python3.9/dist-packages (from tensorflow>=2.8.0->autokeras) (0.4.0)
Requirement already satisfied: setuptools in /usr/local/lib/python3.9/dist-packages (from tensorflow>=2.8.0->autokeras) (67.6.1)
Requirement already satisfied: six>=1.12.0 in /usr/local/lib/python3.9/dist-packages (from tensorflow>=2.8.0->autokeras) (1.16.0)
Requirement already satisfied: grpcio<2.0,>=1.24.3 in /usr/local/lib/python3.9/dist-packages (from tensorflow>=2.8.0->autokeras) (1.53.0)
Requirement already satisfied: python-dateutil>=2.8.1 in /usr/local/lib/python3.9/dist-packages (from pandas->autokeras) (2.8.2)
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.9/dist-packages (from pandas->autokeras) (2022.7.1)
Requirement already satisfied: wheel<1.0,>=0.23.0 in /usr/local/lib/python3.9/dist-packages (from astunparse>=1.6.0->tensorflow>=2.8.0->autokeras) (0.
Requirement already satisfied: scipy>=1.7 in /usr/local/lib/python3.9/dist-packages (from jax>=0.3.15->tensorflow>=2.8.0->autokeras) (1.10.1)
Requirement already satisfied: ml-dtypes>=0.0.3 in /usr/local/lib/python3.9/dist-packages (from jax>=0.3.15->tensorflow>=2.8.0->autokeras) (0.0.4)
Requirement already satisfied: markdown>=2.6.8 in /usr/local/lib/python3.9/dist-packages (from tensorboard<2.13,>=2.12->tensorflow>=2.8.0->autokeras)
Requirement already satisfied: tensorboard-plugin-wit>=1.6.0 in /usr/local/lib/python3.9/dist-packages (from tensorboard<2.13,>=2.12->tensorflow>=2.8.
Requirement already satisfied: google-auth-oauthlib<0.5,>=0.4.1 in /usr/local/lib/python3.9/dist-packages (from tensorboard<2.13,>=2.12->tensorflow>=2
Requirement already satisfied: werkzeug>=1.0.1 in /usr/local/lib/python3.9/dist-packages (from tensorboard<2.13,>=2.12->tensorflow>=2.8.0->autokeras)
Requirement already satisfied: google-auth<3,>=1.6.3 in /usr/local/lib/python3.9/dist-packages (from tensorboard<2.13,>=2.12->tensorflow>=2.8.0->autok
Requirement already satisfied: tensorboard-data-server<0.8.0,>=0.7.0 in /usr/local/lib/python3.9/dist-packages (from tensorboard<2.13,>=2.12->tensorfl
Requirement already satisfied: charset-normalizer~=2.0.0 in /usr/local/lib/python3.9/dist-packages (from requests->keras-tuner>=1.1.0->autokeras) (2.0
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.9/dist-packages (from requests->keras-tuner>=1.1.0->autokeras) (3.4)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.9/dist-packages (from requests->keras-tuner>=1.1.0->autokeras) (2022.12.7)
Requirement already satisfied: urllib3<1.27,>=1.21.1 in /usr/local/lib/python3.9/dist-packages (from requests->keras-tuner>=1.1.0->autokeras) (1.26.15
Requirement already satisfied: tensorflow-hub>=0.8.0 in /usr/local/lib/python3.9/dist-packages (from tensorflow-text->keras-nlp>=0.4.0->autokeras) (0.
Requirement already satisfied: cachetools<6.0,>=2.0.0 in /usr/local/lib/python3.9/dist-packages (from google-auth<3,>=1.6.3->tensorboard<2.13,>=2.12->
Requirement already satisfied: pyasn1-modules>=0.2.1 in /usr/local/lib/python3.9/dist-packages (from google-auth<3,>=1.6.3->tensorboard<2.13,>=2.12->t
Requirement already satisfied: rsa<5,>=3.1.4 in /usr/local/lib/python3.9/dist-packages (from google-auth<3,>=1.6.3->tensorboard<2.13,>=2.12->tensorflo
Requirement already satisfied: requests-oauthlib>=0.7.0 in /usr/local/lib/python3.9/dist-packages (from google-auth-oauthlib<0.5,>=0.4.1->tensorboard<
Requirement already satisfied: importlib-metadata>=4.4 in /usr/local/lib/python3.9/dist-packages (from markdown>=2.6.8->tensorboard<2.13,>=2.12->tenso
Requirement already satisfied: MarkupSafe>=2.1.1 in /usr/local/lib/python3.9/dist-packages (from werkzeug>=1.0.1->tensorboard<2.13,>=2.12->tensorflow>
Requirement already satisfied: zipp>=0.5 in /usr/local/lib/python3.9/dist-packages (from importlib-metadata>=4.4->markdown>=2.6.8->tensorboard<2.13,>=
Requirement already satisfied: pyasn1<0.5.0,>=0.4.6 in /usr/local/lib/python3.9/dist-packages (from pyasn1-modules>=0.2.1->google-auth<3,>=1.6.3->tens
Requirement already satisfied: oauthlib>=3.0.0 in /usr/local/lib/python3.9/dist-packages (from requests-oauthlib>=0.7.0->google-auth-oauthlib<0.5,>=0.
```

Model

We can create the model, it should be easy. The auto model wants the labels to be in the original format, so I setup a couple of new datasets to do it without the categorical bit.

```
clf = ak.ImageClassifier(overwrite=True, max_trials=1)
train_ds = tf.keras.utils.image_dataset_from_directory(ROOT_DIR, validation_split=VAL_SPLIT, subset="training", seed=SEED, batch_size=None, shuffle=True, la
val_ds = tf.keras.utils.image_dataset_from_directory(ROOT_DIR, validation_split=VAL_SPLIT, subset="validation", batch_size=None, seed=SEED, shuffle=False, 
train_ds = train_ds.shuffle(buffer_size=BUFFER)
train_ds = train_ds.prefetch(TPAIN)
val_ds = val_ds.prefetch(TPAIN)
# Feed the tensorflow Dataset to the classifier.
clf.fit(train_ds, validation_data=val_ds, epochs=EPOCHS)
```

```
Trial 1 Complete [00h 07m 27s]
val_loss: 3.1317670345306396

Best val_loss So Far: 3.1317670345306396
Total elapsed time: 00h 07m 27s
Epoch 1/20
524/524 [==============================] - 23s 38ms/step - loss: 3.8603 - accuracy: 0.0995 - val_loss: 3.1924 - val_accuracy: 0.1058
Epoch 2/20
524/524 [==============================] - 21s 37ms/step - loss: 3.1732 - accuracy: 0.1036 - val_loss: 3.1692 - val_accuracy: 0.1061
Epoch 3/20
524/524 [==============================] - 21s 38ms/step - loss: 3.1634 - accuracy: 0.1068 - val_loss: 3.1671 - val_accuracy: 0.1061
Epoch 4/20
524/524 [==============================] - 21s 37ms/step - loss: 3.1585 - accuracy: 0.1069 - val_loss: 3.1753 - val_accuracy: 0.1056
Epoch 5/20
524/524 [==============================] - 21s 37ms/step - loss: 3.1523 - accuracy: 0.1070 - val_loss: 3.1548 - val_accuracy: 0.1061
Epoch 6/20
524/524 [==============================] - 21s 37ms/step - loss: 3.1426 - accuracy: 0.1072 - val_loss: 3.1523 - val_accuracy: 0.1061
Epoch 7/20
524/524 [==============================] - 21s 37ms/step - loss: 3.1337 - accuracy: 0.1074 - val_loss: 3.1480 - val_accuracy: 0.1061
Epoch 8/20
524/524 [==============================] - 21s 38ms/step - loss: 3.1323 - accuracy: 0.1073 - val_loss: 3.1455 - val_accuracy: 0.1061
Epoch 9/20
524/524 [==============================] - 21s 37ms/step - loss: 3.1295 - accuracy: 0.1076 - val_loss: 3.1436 - val_accuracy: 0.1061
Epoch 10/20
524/524 [==============================] - 21s 37ms/step - loss: 3.1266 - accuracy: 0.1075 - val_loss: 3.1433 - val_accuracy: 0.1061
Epoch 11/20
524/524 [==============================] - 21s 38ms/step - loss: 3.1271 - accuracy: 0.1076 - val_loss: 3.1348 - val_accuracy: 0.1061
Epoch 12/20
524/524 [==============================] - 21s 37ms/step - loss: 3.1230 - accuracy: 0.1075 - val_loss: 3.1319 - val_accuracy: 0.1061
Epoch 13/20
524/524 [==============================] - 21s 37ms/step - loss: 3.1214 - accuracy: 0.1076 - val_loss: 3.1327 - val_accuracy: 0.1061
Epoch 14/20
524/524 [==============================] - 21s 38ms/step - loss: 3.1176 - accuracy: 0.1076 - val_loss: 3.1676 - val_accuracy: 0.1054
Epoch 15/20
524/524 [==============================] - 21s 37ms/step - loss: 3.1183 - accuracy: 0.1077 - val_loss: 3.1353 - val_accuracy: 0.1061
Epoch 16/20
524/524 [==============================] - 21s 38ms/step - loss: 3.1182 - accuracy: 0.1075 - val_loss: 3.1371 - val_accuracy: 0.1061
Epoch 17/20
524/524 [==============================] - 21s 37ms/step - loss: 3.1165 - accuracy: 0.1076 - val_loss: 3.1331 - val_accuracy: 0.1061
Epoch 18/20
524/524 [==============================] - 21s 37ms/step - loss: 3.1151 - accuracy: 0.1075 - val_loss: 3.1330 - val_accuracy: 0.1061
Epoch 19/20
524/524 [==============================] - 21s 38ms/step - loss: 3.1144 - accuracy: 0.1075 - val_loss: 3.1315 - val_accuracy: 0.1061
Epoch 20/20
524/524 [==============================] - 21s 37ms/step - loss: 3.1134 - accuracy: 0.1077 - val_loss: 3.1280 - val_accuracy: 0.1061
WARNING:absl:Found untraced functions such as _jit_compiled_convolution_op, _jit_compiled_convolution_op while saving (showing 2 of 2). These function
<keras.callbacks.History at 0x7f092811e490>
```

Bob's Your Uncle

The automation does save us the time of making a model, deciding on a structure, and picking hyperparameters, but that definitely isn't all the work and I'd say not really the hardest part of the process. The work in getting and prepping the data to go into the model, automated or not, is probably more challenging and complex than making a model to do the predicting. If you're designing complex models to do chatGPTesque breakthroughs this may not hold true, but these automodels aren't doing anything that complex. This is why I think that automated machine learning will be most useful in the corporate world, where the data is (often) highly consistent and the models are relatively simple.