[SPOILERS] Using Convolutional Neural Networks to Identify Game of Thrones Content

Machine Learning Engineer Nanodegree

Capstone Project

Minh Hoang September 29, 2017

I. Definition

Project Overview

In the domain of computer vision, image recognition remains to one of the most highly researched areas. Competitions such as the ImageNet Large-Scale Visual Recognition Challenge contniue to encourage researchers innovate and create new neural network architectures to improve performance on image recognition problems. In 2012, for the first time a convolutional neural network (CNN) named AlexNet[1] proposed by Alex Krizhevskey et al. was used to win the ImageNet competition. Obtaining a top 5 test error (the rate the model does not output the correct label with its top 5 predictions for a given image) of 15.4%, the performance of AlexNet blew the competition away, with the next best entry having a top 5 test error of 26.2%. Since then, the most common and effective technique used to tackle image recognition problems are convolutional neural networks. Many other award winning architectures followed, from VGG Net[2] to GoogleNet[3] from recently ResNet[4] by Microsoft, all containing different tweaks and modifications but built from the common foundation of convolutional neural networks. Thanks to Nvidia, as GPU (Graphics Processing Unit) computing power has ramped up over the years, it has allowed the computer vision community to take advantage and develop even deeper and complex CNNs.

My motivation for my capstone project is driven by what I see on my phone on a daily basis. In the digital world we live in today, there are billions of images on the the likes of social media that can serve as training data for experimenting and solving problems with image recognition and CNNs.

Problem Statement

In this day and age, machine learning techniques have a massive influence on social media platforms. For instance, Instagram and YouTube are fueled by recommendation systems and discover algorithms with the purpose of bringing relevant and specific content tailored to the users' interests. These algorithms work extremely well, almost to a fault in some situations. For example, let's say you are obsessed with a TV series but you haven't seen the latest episode or whole thing yet. Social media platforms like Instagram will know that you are particularly interested in this specific TV series, and therefore will bring content to you revolving around it, which is fine until you see something that you have not seen yet or other various spoilers that potentially ruins a plot or storyline of the TV series for you. Inspired by the recent episode leaks in the latest season of Game of Thrones, a massive problem with browsing social media is potentially seeing content that contain spoilers that you don't intentionally want to see. This problem is not just limited to Instagram, it can also be a problem browsing YouTube, Facebook, Twitter, and pretty much any type of social media outlet.

The simple solution to this problem is to just give the user the ability to add certain keywords, such as "Game of Thrones", to a filter that will block images that are relevant to the keywords. The question arises that how will this filter be able to discern which images are related to the certain keywords in the filter? With the strides made in the field of image recognition, it is completely plausible to build and train a CNN that will, given a batch of images, would be able to recognize and determine which images contain Game of Thrones

content. The performance of this CNN can be measured simply by how accurate it is when determining whether or not the image is related to Game of Thrones.

Convolutional neural networks have been used to solve a wide variety of problems dealing with image recognition. The solution model will be developed using the deep learning library framework Keras in combination with TensorFlow backend. This neural network will have several convolutional layers with different filter sizes that the input image will pass through in order to determine the low and high level features of the images. These features obtained from the convolution layers will be flattened into a vector and will be pass through a fully connected layer which will ultimately decide whether or not the image contains content relating to Game of Thrones using a softmax probability. The CNN will be trained using 40,000 images and validated using a separate 10,000 images. The training and validation image datasets will be compiled by obtaining images from the Instagram API. The 40,000 training images and 10,000 validation images will be fed into the neural network for 50 epochs. The performance of the CNN will be measured by the accuracy of its ability to predict the correct label for an image and its loss. After each epoch, the weights will be optimized using the Adam[6] optimizer to minimize the overall loss of the model output.

Metrics

My proposed CNN architecture will have 2 outputs: Not GoT (images not relating to Game of Thrones) and GoT (images confirmed being related to Game of Thrones). A softmax function will be applied to the output to assign probabilities for each output. With 2 different categorical outputs, I will be using categorical crossentropy loss to calculate the loss of the output of the model. The loss is represented by the formula below:

$$H(p, q) = -\sum_x p(x) \log(q(x))$$

where p is equal to the true label and q is equal to the predicted labels. The categorical crossentropy loss will measure the probability error in the classification task. Generally, a low loss value will indicate that the model is performing well in terms of classifying an image while a high loss value will indicate that the model is not doing a good job classifying an image to the correct label.

Additionally, we will be using accuracy as another performance metric for the model, which calculates how often the predictions generated from the model matches the true labels from the dataset.

$$accuracy = \frac{\#\ correct\ predictions}{\#\ total\ predictions}$$

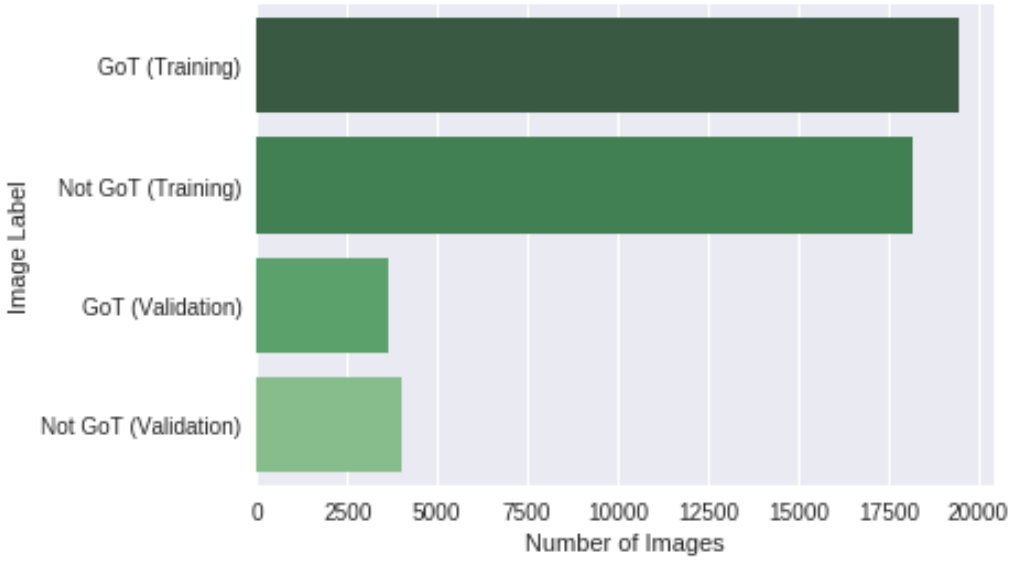
II. Analysis

Data Exploration

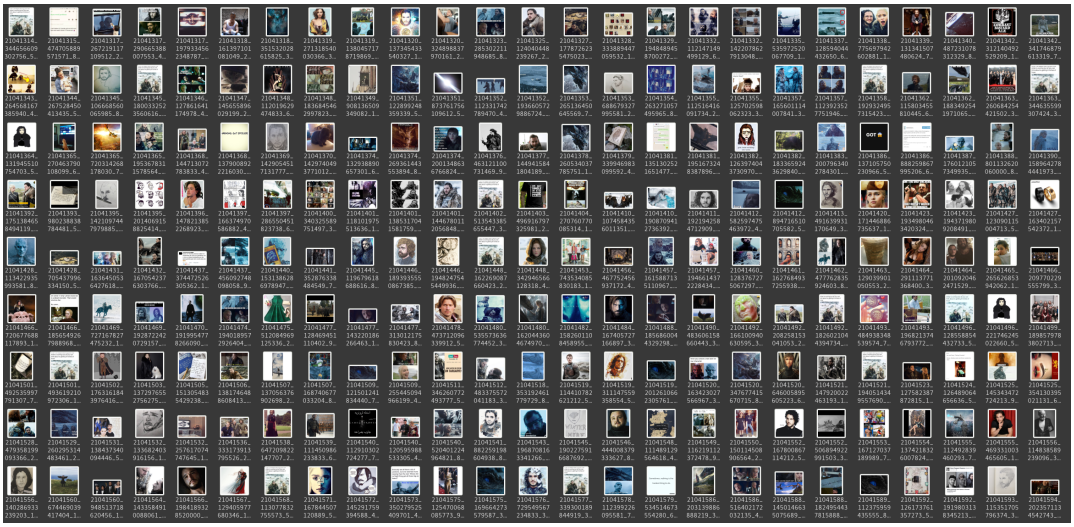
Since the problem revolves around social media platforms, I have chosen Instagram as the main source of images for my dataset. For this project, I will be creating my own dataset by scraping Instagram's API for images using the Python Instragram Scraper[5]. This tool is built using Python and can fetch and download images using either a user account name or a hashtag. Because I will be needing a large amount of images, I decided to collect the images by using hashtags. The dataset will contain images relating to Game of Thrones (by scraping the Instagram API for images under relevant Game of Thrones hashtags). I will be using my own Instagram account as a sandbox environment, so the dataset will also be mixed with other images relevant to my Discover feed such as cars, basketball, shoes, and other things unrelated to Game of Thrones.

Images will be obtained from the Instagram API using hashtags to create the necessary datasets for the model.

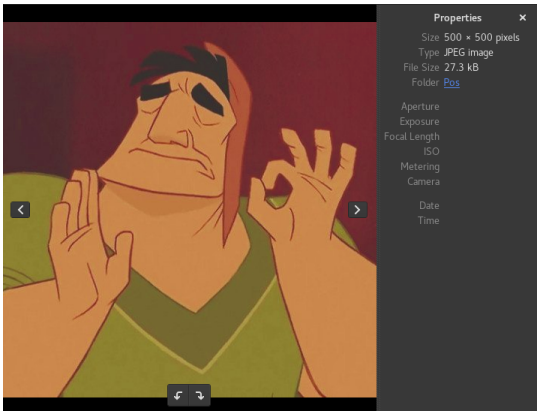
Dataset	Number of Images	Hashtags Used
Training	19,466	#gameofthrones, #asoiaf, #jonsnow, #cersei
Training	18,195	#nike, #nba, #cars
Validation	3,625	#gameofthrones, #asoiaf, #jonsnow, #cersei
Validation	3,992	#nike, #nba, #cars
Testing	888	#gotspoilers
Testing	886	#boostvibes, #dubnation



A total of 37,661 images for training and 7,617 images for validation were collected. The Figure above shows the distribution of images labeled "GoT versus images labeled"Not GoT". It is important, especially in the training dataset, to have balanced distribution of class labels. If there was an imablance of classes in our dataset, the model will not properly learn during training. In addition, 1,774 more images will be collected to be used as a final test set. This dataset will be used to test the performance of my model. Since the model has never seen this dataset, it will be a great indicator of the performance of my CNN. All images have 3 color channels, and will be resized to 224x224 to be used as input for the CNN. A sample of the image dataset can be seen below:



After inspecting the datasets, I've noticed a couple of things to keep in mind. Because the images are being pulled straight from Instagram using hashtags, there will be instances of the same images. Hopefully the size of this dataset will help ease this issue, but if signs of overfitting are seen during the intial training of the benchmark model, data augmentation will be considered as an option to relieve this issue. Another issue is that some images under Game of Throne related hashtags have nothing to do with Game of Thrones, like the image below, which is actually from the animated film *The Emperor's New Groove*.

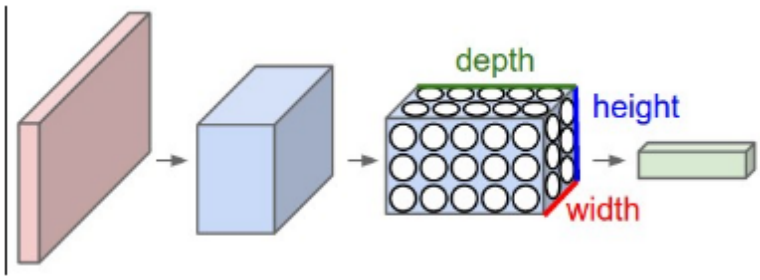


For the sake of time, I have decided to leave these images in the dataset. Because the goal is to identify Game of Throne images for the sake of filtering, there is a higher tolerance for false positives (images falsely flagged for Game of Thrones related content) rather than false negatives (images falsely flagged as unrelated to Game of Thrones). The main thing we should be worry about is Game of Throne related images in the Negative labeled images in our dataset, and so far after inspection, the Negative labeled images are unrelated to Game of Thrones content.

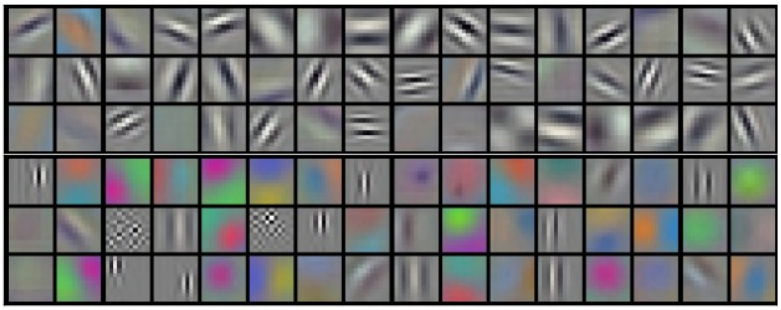
Algorithms and Techniques

A standard neural network takes a single vector as an input and is transform through a series of hidden layers, which are made of fully connected neurons with its own weight in each connection. If I were to use a standard neural network for this specific problem, that would require flattening the 224x224x3 input image. So in a single fully connected neuron in the first hidden layer of a standard neural network, there would be 150,528 weights (224 multiplied by 224 multiplied by 3). Keep in mind, this is the number of parameters for one neuron, so the parameters would quickly stack up as we add more neurons and more hidden layers. The huge amount of parameters would take a very long time to train and could also result in overfitting as well. For these reasons, convolutional neural networks have became the popular alternative for solving image recognition problems.

The main advantage of using CNNs for image recognition tasks is the fact that it is able to take 3 dimensions (width x height x depth) as an input. This means that a CNN is able to take the 224x224x3 image as input and there is no need to flatten the dimensions like you would with a standard neural network. Each convolutional layer will take a 3D input and transform it into a 3D output of neuron activations, which are also known as the filters or feature maps. This is where the model learns high and low level features of the input image. The figure below, taken from Stanford's CS231n course notes[9], illustates the general operation of a CNN.



A convolutional layer works essentially by sliding a "viewing window" across the entire area of an input image from left to right. This "viewing window"" is known as the filter (the size of the filter is a parameter that will be tuned) and is essentially an array of numbers that make up weights of a given filter. As the filter slides (also known as the stride which is a tunable parameter) across the image, it is performing element wise multiplication with the weights on the filter and the pixels in the area of the input image the filter is currently viewing. Convolution layers will have a stack of filters, and each filter will have their respective weights which will ultimately make of the feature maps of the convolution layer. The figure below, also taken from Stanford's CS231n course notes[9], is an example of the type of edges and characteristics the feature maps may pick up:



The model will use these stacked feature maps of low and high level characteristics to help determine the correct class of the input image.

The plan is to use multiple convolution layers to create a stack of feature maps with a 3x3 filter size with stride 2 with zero padding. I will using stride 2 with the purpose of reducing the dimensionality of the input data instead of using max pooling layers. As the neural network proceeds, the feature map size of each convolution layer will increase (16, 32, 64, 128, 256). The number of layers and feature maps is still to be determined as I begin testing. The final convolution layer will be followed by a flattening layer that will reshape the data for the fully connected layers. The weights in each layer will be initialized using the Xavier Initializer[7], which ensures that the intial distribution of the weights will approximately be the same and will improve the rate of convergence at the optimal loss during training. Dropout layers, which turns off a fraction of the neurons during training, will be added after the fully connected layer to minimize overfitting. In addition, Batch normalization will be added each layer to provide regulization. The batch normalization algorithm[8] can be summarized as:

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Furthermore, the non-linear ReLU activation will be used after each layer with exception to the final output layer. The formula for ReLU activation[9] can be seen below:

$$f(x) = \max(0, x)$$

Basically what the ReLU activation does is thresholds the output at 0, where negative outputs will become 0 and postive outputs will remain. This activation function has been increasingly popular due to the fact that it helps prevent the problem of vanishing gradients.

The final output layer (2 classes: Not GoT and GoT), which will use a softmax function, given in the formula above, to squash the outputs and assign probabilities to each output to determine the final classification.

$$P(y_i \mid x_i; W) = \frac{e^{f_{y_i}}}{\sum_j e^{f_j}}$$

During training, the categorical cross entropy loss will be minimized using the Adam optimizer. The details of the algorithm for Adam is outline in the image below, taken from the original paper[6].

Algorithm 1: *Adam*, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation. g_t^2 indicates the elementwise square $g_t \odot g_t$. Good default settings for the tested machine learning problems are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. All operations on vectors are element-wise. With β_1^t and β_2^t we denote β_1 and β_2 to the power t.

Require: α: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters θ
Require: θ_0: Initial parameter vector
  $m_0 \leftarrow 0$ (Initialize 1st moment vector)
  $v_0 \leftarrow 0$ (Initialize 2nd moment vector)
  $t \leftarrow 0$ (Initialize timestep)
  while θ_t not converged **do**
    $t \leftarrow t + 1$
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)
    $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
    $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
    $\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
    $\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
    $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
  end while
  return θ_t (Resulting parameters)

The Adam optimizer takes advantage of a per parameter adaptive learning rate, which avoids the expensive task of tuning learning rates during training.

Benchmark

As a benchmark model, I will be implementing a CNN with 5 convolution layers similar to AlexNet[1] using Keras[10], a deep learning library built on top of TensorFlow[11]. Since AlexNet is considered one of the pioneers of using CNN to tackle image recognition problems, it will serve as a great starting point and benchmark model in regards to the development of my own proposed CNN architecture. The benchmark model will received the same treatment during training as my proposed solution model, and will be trained on 37,661 training images and validated on 7,617 validation images for 50 epochs. The loss of the model will be calculated using the categorical crossentropy loss function. This loss function will be minimized and weights will be optimized using the Adam optimizer. Accuracy of true labels versus predicted labels will be used the performance metric. The validation loss and validation accuracy (true label versus label prediction) will be used as benchmark results as I develop and train the solution model. Comparing the validation loss and validation accuracy of the benchmark model versus the solution model, I will be able to determine whether or not the solution model is underperforming and if the solution model needs to be improved.

The architecture for the benchmark model can be seen in the code chunk below (see `GoT_CNN_benchmark_model.ipynb` for more):

```
model = Sequential()

model.add(Conv2D(filters=16, kernel_size=3, strides=2, padding='same',
                 input_shape=(224,224,3)))
model.add(BatchNormalization())
model.add(Activation('relu'))

model.add(Conv2D(filters=32, kernel_size=3, strides=2, padding='same'))
model.add(BatchNormalization())
model.add(Activation('relu'))

model.add(Conv2D(filters=64, kernel_size=3, strides=2, padding='same'))
model.add(BatchNormalization())
model.add(Activation('relu'))

model.add(Conv2D(filters=128, kernel_size=3, strides=2, padding='same'))
model.add(BatchNormalization())
model.add(Activation('relu'))

model.add(Conv2D(filters=256, kernel_size=3, strides=2, padding='same'))
model.add(BatchNormalization())
model.add(Activation('relu'))

model.add(Flatten())

model.add(Dense(1000))
model.add(BatchNormalization())
model.add(Activation('relu'))
model.add(Dropout(0.5))

model.add(Dense(2, activation='softmax'))
```

Initially, I planned to train the model for 50 epochs. However, during the actual training, the training loss continued to drop and accuracy continued to increase after each epoch (reached about 94% training accuracy) but it appears the model stopped learning as the validation loss and validation accuracy plateaued around 10 epochs (76% validation accuracy).

```
Epoch 6/50
37632/37661 [============================>.] - ETA: 0s - loss: 0.4190 - acc: 0.8035Epoch 00005:
 val_loss improved from 0.53453 to 0.52361, saving model to saved_models/benchmark_model.hdf5
37661/37661 [==============================] - 114s - loss: 0.4191 - acc: 0.8035 - val_loss: 0.5
236 - val_acc: 0.7394
Epoch 7/50
37632/37661 [============================>.] - ETA: 0s - loss: 0.3508 - acc: 0.8429Epoch 00006:
 val_loss did not improve
37661/37661 [==============================] - 113s - loss: 0.3508 - acc: 0.8428 - val_loss: 0.7
514 - val_acc: 0.6948
Epoch 8/50
37632/37661 [============================>.] - ETA: 0s - loss: 0.2740 - acc: 0.8817Epoch 00007:
 val_loss did not improve
37661/37661 [==============================] - 114s - loss: 0.2741 - acc: 0.8816 - val_loss: 0.6
650 - val_acc: 0.7537
Epoch 9/50
37632/37661 [============================>.] - ETA: 0s - loss: 0.2029 - acc: 0.9160Epoch 00008:
 val_loss did not improve
37661/37661 [==============================] - 113s - loss: 0.2030 - acc: 0.9160 - val_loss: 0.7
169 - val_acc: 0.7558
Epoch 10/50
37632/37661 [============================>.] - ETA: 0s - loss: 0.1502 - acc: 0.9398Epoch 00009:
 val_loss did not improve
37661/37661 [==============================] - 113s - loss: 0.1502 - acc: 0.9398 - val_loss: 0.7
505 - val_acc: 0.7529
```

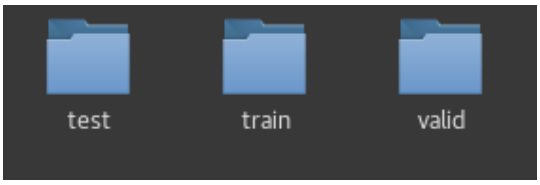
I decided to stop the training early since this is a sign that the model may be overfitting the data. This will serve as the benchmark result as I continue to develop and refine the final solution model.

III. Methodology

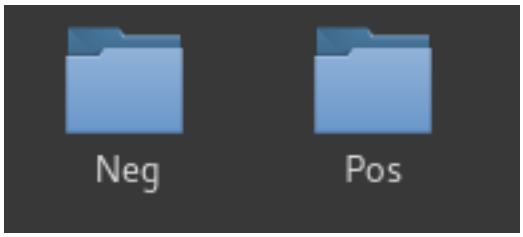
Data Preprocessing

Every image in the datasets will be resized to 224x224x3 (Width x Height x Depth) and rescaled (by dividing each pixel by 255) using Keras[10] **ImageDataGenerator**. Since Batch Normalization will be used at each layer of the CNN, centering and normalization of the data will not be necessary during pre-processing.

Each type of dataset will have its own directory folder shown below.



Within each dataset directory folder, the images are separatated into different folders that represent their true labels. Neg represents the "0" label (Not GoT) and Pos represents the "1" label (GoT) shown below.



The **flow_from_directory** method of **ImageDataGenerator** will automatically create a 2D one-hot encoded labels (categorical) for our images based on the directory and their subdirectory folder names and will also shuffle the dataset. This process can be implemented in the code below.

```
from keras.preprocessing.image import ImageDataGenerator, array_to_img,
                                      img_to_array, load_img

batch_size=16

train_aug = ImageDataGenerator(rescale=1.0/255)
```

```
test_aug = ImageDataGenerator(rescale=1.0/255)

train_generator = train_aug.flow_from_directory('GoT-images/train',
                                                target_size=(224,224),
                                                batch_size=batch_size,
                                                shuffle=True,
                                                class_mode='categorical')

validation_generator = test_aug.flow_from_directory('GoT-images/valid',
                                                    target_size=(224,224),
                                                    batch_size=batch_size,
                                                    shuffle=True,
                                                    class_mode='categorical')
```

Implementation

For the initial implementation (version 1.0), the CNN architecture will be made up of 5 convolutional layers, all with zero padding. The first convolutional layer will have 24 filters, and each filter will be 6x6 in size and a stride of 1 to maximize the initial amount of information learned from the input image. The second convolutional layer following will have 48 filters, and each filter will be 5x5 in size with stride 2. Next will be another convolutional layer with 64 filters,, with a filter size of 4x4 and stride 2. The last convolutional layer will have 128 filters, with a filter size of 3x3 and stride 2. After each convolutional layer, a batch normalization layer will be added for regularization and followed by the non-linear ReLU activation. Futhermore, each convolutional layer will also have a max pooling layer with a pool size of 2, which subsamples the output of the convolution layer and further reduces the dimensions of the input image. After the input image passes through all the convolutional layers, it will be flattened into a single vector and fed into a fully connected layer with 200 neurons, followed by a final output layer with 2 classes. The final outputs will be squashed using a softmax function and a probability will be assigned to each class. In order to prevent overfitting, I've also added a dropout layer after every layer (except the final output layer) with varying keep probabilities.

This CNN will be implementated using Keras[10] and it's **Sequential** model API (refer to `GoT_CNN_1.0.ipynb` for more details).

```
from keras.layers import Conv2D, Activation, Dropout, Flatten, Dense
from keras.models import Sequential
from keras.layers.normalization import BatchNormalization

# Version 1.0

model = Sequential()

model.add(Conv2D(filters=24, kernel_size=6, strides=1, padding='same',
          input_shape=(224,224,3)))
model.add(BatchNormalization())
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=2))
model.add(Dropout(0.2))


model.add(Conv2D(filters=48, kernel_size=5, strides=2, padding='same'))
model.add(BatchNormalization())
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=2))
```

```
model.add(Dropout(0.3))

model.add(Conv2D(filters=64, kernel_size=4, strides=2, padding='same'))
model.add(BatchNormalization())
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=2))
model.add(Dropout(0.4))

model.add(Conv2D(filters=128, kernel_size=3, strides=2, padding='same'))
model.add(BatchNormalization())
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=2))
model.add(Dropout(0.5))


model.add(Flatten())

model.add(Dense(200))
model.add(BatchNormalization())
model.add(Activation('relu'))
model.add(Dropout(0.7))


model.add(Dense(2, activation='softmax'))
```

The **Sequential** model is very user friendly, and allows for fast prototyping and experimentation of hyperparameters. The model object is simply initiated and layers are sequentially added.

Next, the model will be compiled using categorical cross entropy loss and accuracy metrics. During training, the weights will be updated using the Adam optimizer with the default learning rate of 0.001. Keras especially makes this part easy for the user, and is performed with a single line of code.

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

The model will be trained for 50 epochs. During training, the model will be fed images created by the **ImageGenerator** objects created earlier. The code for implementing the training process is outline below.

```
from keras.callbacks import ModelCheckpoint, EarlyStopping


epochs = 50

# number of training images
nb_training_size = 37661.0

# number of validation images
nb_validation_size = 7617.0

# saves best model to filepath
checkpointer = ModelCheckpoint(filepath='saved_models/ver1_0.hdf5',
                               verbose=1, save_best_only=True)

# stops training if validation loss stops improving
early_stop = EarlyStopping(monitor='val_loss', patience=5)
```

```
model.fit_generator(train_generator,
                    steps_per_epoch=nb_training_size//batch_size,
                    epochs=epochs,
                    validation_data=validation_generator,
                    validation_steps=nb_validation_size//batch_size,
                    verbose=1,
                    callbacks=[checkpointer, early_stop])
```

Keras also offers a handful of useful callbacks objects that can be used during the training process. For this specific model, I will be using the **ModelCheckpoint** callback, which saves the best model to file during training, and the **EarlyStopping** callback, which stops training if the validation loss ceases to improve for 5 epochs.

Because of the extensive computational power required for training a CNN of this size, I will be using EC2 GPU cloud computing on Amazon Web Services (px2.large) to train the model. The training/validation loss and accuracy will be monitored and will determine if any changes to model architecture or tuning of hyperparameters is needed.

The initial implementation was trained for 12 epochs before being stopped early due to the plateau of validation. At the last epoch, the model finished with a training accuracy of 76% and validation accuracy of 72% (the validation accuracy of benchmark model outperformed the initial model at 76%)

```
2352/2353 [============================>.] - ETA: 0s - loss: 0.5317 - acc: 0.7354Epoch 00005: val_loss improved fro
m 0.50295 to 0.46847, saving model to saved_models/ver1_0.hdf5
2353/2353 [==============================] - 891s - loss: 0.5317 - acc: 0.7353 - val_loss: 0.4685 - val_acc: 0.7798
Epoch 7/50
2352/2353 [============================>.] - ETA: 0s - loss: 0.5257 - acc: 0.7401Epoch 00006: val_loss did not impr
ove
2353/2353 [==============================] - 895s - loss: 0.5257 - acc: 0.7401 - val_loss: 0.6519 - val_acc: 0.6723
Epoch 8/50
2352/2353 [============================>.] - ETA: 0s - loss: 0.5180 - acc: 0.7477Epoch 00007: val_loss did not impr
ove
2353/2353 [==============================] - 892s - loss: 0.5179 - acc: 0.7477 - val_loss: 0.5552 - val_acc: 0.7286
Epoch 9/50
2352/2353 [============================>.] - ETA: 0s - loss: 0.5066 - acc: 0.7548Epoch 00008: val_loss did not impr
ove
2353/2353 [==============================] - 905s - loss: 0.5066 - acc: 0.7549 - val_loss: 0.6362 - val_acc: 0.6670
Epoch 10/50
2352/2353 [============================>.] - ETA: 0s - loss: 0.5006 - acc: 0.7581Epoch 00009: val_loss did not impr
ove
2353/2353 [==============================] - 906s - loss: 0.5007 - acc: 0.7581 - val_loss: 0.5422 - val_acc: 0.7365
Epoch 11/50
2352/2353 [============================>.] - ETA: 0s - loss: 0.4938 - acc: 0.7622Epoch 00010: val_loss did not impr
ove
2353/2353 [==============================] - 904s - loss: 0.4938 - acc: 0.7623 - val_loss: 0.6024 - val_acc: 0.7071
Epoch 12/50
2352/2353 [============================>.] - ETA: 0s - loss: 0.4914 - acc: 0.7621Epoch 00011: val_loss did not impr
ove
2353/2353 [==============================] - 904s - loss: 0.4913 - acc: 0.7620 - val_loss: 0.5705 - val_acc: 0.7200

<keras.callbacks.History at 0x7f19f87113c8>
```

Refinement

After the intial results of the model, it appears there is still room for improvement. To increase the generalization strength of the model, data augmentation will be performed on our training dataset. Using data augmentation, the original images in the training data set will be sheared, zoomed into, and flipped horizontally so that the model does not see the same specific image multiple times during training. This method is especially useful in cases where data is limited. It is important to note that data augmentation only needs to be performed on the training dataset in order to determine how robust the model is when being validated with the validation dataset. Data augmentation can be implemented also using Keras' **ImageDataGenerator** with a few tweaks and is outline in the code below.

```
from keras.preprocessing.image import ImageDataGenerator, array_to_img,
                                      img_to_array, load_img

batch_size=32
```

```
# Data Augmentation added to ImageDataGenerator of training dataset
train_aug = ImageDataGenerator(rescale=1.0/255,
                               shear_range=0.2,
                               zoom_range=0.2,
                               horizontal_flip=True)

test_aug = ImageDataGenerator(rescale=1.0/255)
```

Furthermore, I have decided increase the number of filters of each convolutional layer. In addition, the dropout layers and max pooling layers are being removed from the convolutional layers because I was concern that the model is potentially losing information during the dropout/max pooling steps. Instead, I am replacing the max pooling layers with additional convolutional layers with stride 2 as the main source of dimension reduction. Also, an additional fully connected layer has been added, both with 1000 neurons each. Collectively, all these refinements have been made in order to increase the learning strength of the model. The new implementation of the CNN can be seen in the code below (can also be found in `GoT_CNN_2.0.ipynb`).

```
from keras.layers import Conv2D, Activation, Dropout, Flatten, Dense
from keras.models import Sequential
from keras.layers.normalization import BatchNormalization

# Version 2.0

model = Sequential()

model.add(Conv2D(filters=64, kernel_size=3, strides=1, padding='same',
                 input_shape=(224,224,3)))
model.add(BatchNormalization())
model.add(Activation('relu'))
model.add(Conv2D(filters=64, kernel_size=3, strides=2, padding='same'))
model.add(BatchNormalization())
model.add(Activation('relu'))


model.add(Conv2D(filters=128, kernel_size=3, strides=1, padding='same'))
model.add(BatchNormalization())
model.add(Activation('relu'))
model.add(Conv2D(filters=128, kernel_size=3, strides=2, padding='same'))
model.add(BatchNormalization())
model.add(Activation('relu'))


model.add(Conv2D(filters=256, kernel_size=3, strides=1, padding='same'))
model.add(BatchNormalization())
model.add(Activation('relu'))
model.add(Conv2D(filters=256, kernel_size=3, strides=2, padding='same'))
model.add(BatchNormalization())
model.add(Activation('relu'))


model.add(Conv2D(filters=512, kernel_size=3, strides=1, padding='same'))
model.add(BatchNormalization())
model.add(Activation('relu'))
model.add(Conv2D(filters=512, kernel_size=3, strides=2, padding='same'))
model.add(BatchNormalization())
```

```
model.add(Activation('relu'))


model.add(Conv2D(filters=512, kernel_size=3, strides=1, padding='same'))
model.add(BatchNormalization())
model.add(Activation('relu'))
model.add(Conv2D(filters=512, kernel_size=3, strides=2, padding='same'))
model.add(BatchNormalization())
model.add(Activation('relu'))

model.add(Conv2D(filters=1024, kernel_size=3, strides=2, padding='same'))
model.add(BatchNormalization())
model.add(Activation('relu'))

model.add(Flatten())

model.add(Dense(1000))
model.add(BatchNormalization())
model.add(Activation('relu'))
model.add(Dropout(0.7))

model.add(Dense(1000))
model.add(BatchNormalization())
model.add(Activation('relu'))
model.add(Dropout(0.5))

model.add(Dense(2, activation='softmax'))
```

The second version of the CNN model implementation was also trained on an AWS EC2 instance. The first round of training was unfortunately interrupted, and was only able to train the model for 30 epochs. I was able save the current weights as a checkpoint and resumed training for another 22 epochs for a total of 52 training epochs. The results can be seen below.

```
Epoch 18/25
1175/1176 [============================>.] - ETA: 1s - loss: 0.1749 - acc: 0.9303Epoch 00017: val_loss did not impr
ove
1176/1176 [==============================] - 1546s - loss: 0.1748 - acc: 0.9304 - val_loss: 0.4403 - val_acc: 0.829
7
Epoch 19/25
1175/1176 [============================>.] - ETA: 1s - loss: 0.1692 - acc: 0.9340Epoch 00018: val_loss did not impr
ove
1176/1176 [==============================] - 1547s - loss: 0.1692 - acc: 0.9340 - val_loss: 0.4395 - val_acc: 0.835
9
Epoch 20/25
1175/1176 [============================>.] - ETA: 1s - loss: 0.1634 - acc: 0.9361Epoch 00019: val_loss did not impr
ove
1176/1176 [==============================] - 1547s - loss: 0.1633 - acc: 0.9361 - val_loss: 0.4583 - val_acc: 0.832
4
Epoch 21/25
1175/1176 [============================>.] - ETA: 1s - loss: 0.1528 - acc: 0.9402Epoch 00020: val_loss did not impr
ove
1176/1176 [==============================] - 1545s - loss: 0.1528 - acc: 0.9401 - val_loss: 0.4469 - val_acc: 0.836
5
Epoch 22/25
1175/1176 [============================>.] - ETA: 1s - loss: 0.1431 - acc: 0.9436Epoch 00021: val_loss did not impr
ove
1176/1176 [==============================] - 1546s - loss: 0.1431 - acc: 0.9436 - val_loss: 0.4853 - val_acc: 0.838
5
```

The model ended with a training accuracy of 94% and a validation accuracy of 84%, which is a nice improvement from both the benchmark model (76%) and the first version (72%).

IV. Results

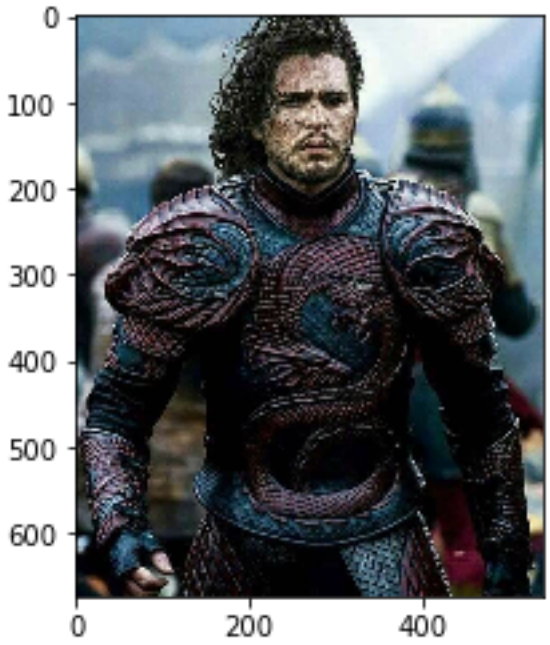
Model Evaluation

The second version of the CNN model implementation appears very promising, and as a final step, the performance will be tested using the test dataset (1,774 images) that was set aside when the data was first collected. The testing process can be found in the `GoT_CNN_Final.ipynb` notebook. Evaluating the final model with the test dataset, the model yielded a loss of 0.59 and an accuracy of 81%.

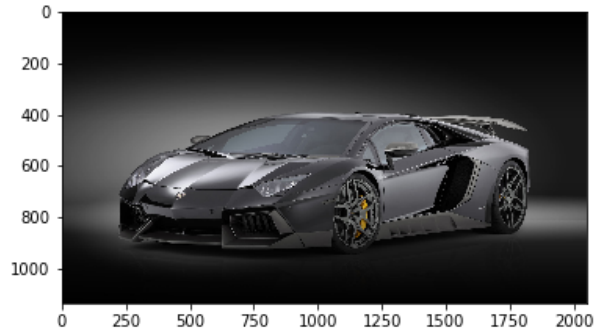
Considering that the CNN architecture was built from scratch, the final implementation of the CNN model performed nicely even with unseen test data with 81% of predictions being correct (a slight drop off from the validation of 84% earlier). This model clearly outperforms the benchmark model (76%) along with the initial implementation (72%). For real world deployment, I would prefer the model to be able to at least be 90% accurace, but this current model is a nice start, and can be built upon with room for more improvements and experimentation.

V. Conclusion

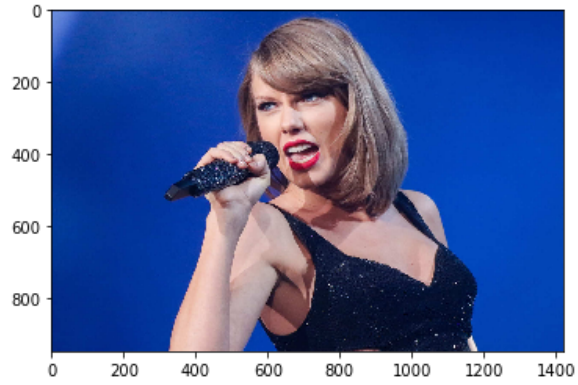
To get a better sense of the final model's potential shortcomings and what type of images are being classified as Game of Throne related images, I've included a function in the `GoT_CNN_Final.ipynb` notebook that takes an image from a filepath and uses the final CNN model (Version 2.0) to predict the outcome and plots the image. Here are 17 sample images that were fed into this function:



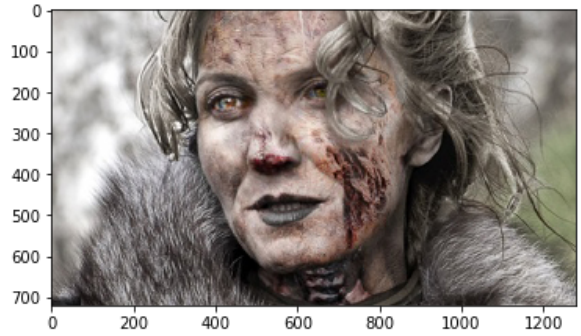
Beware: Game of Thrones content detected!



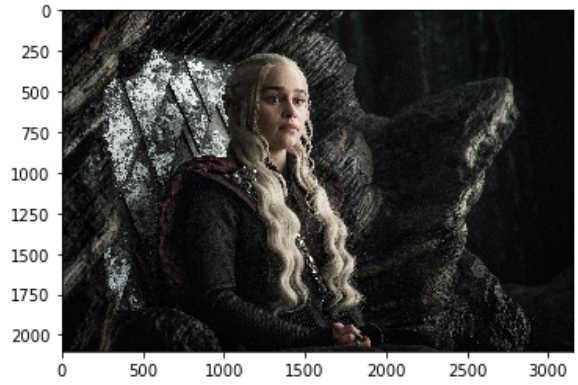

```
This image is not related to Game of Thrones.
```



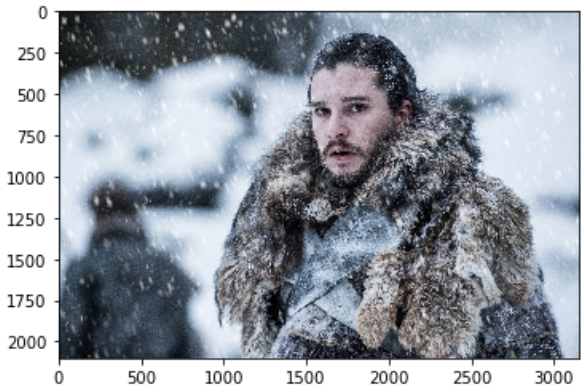

```
Beware: Game of Thrones content detected!
```




```
Beware: Game of Thrones content detected!
```

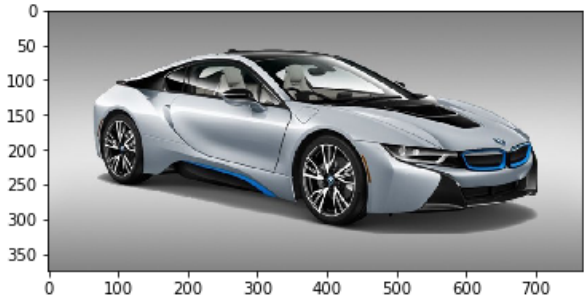



```
Beware: Game of Thrones content detected!
```



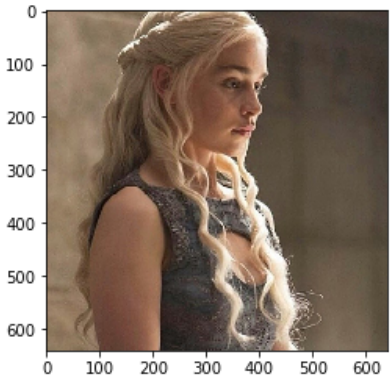
```
Beware: Game of Thrones content detected!
```

```
Beware: Game of Thrones content detected!
```



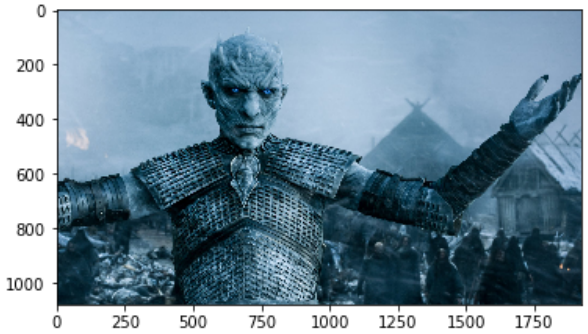
```
This image is not related to Game of Thrones.
```

```
Beware: Game of Thrones content detected!
```



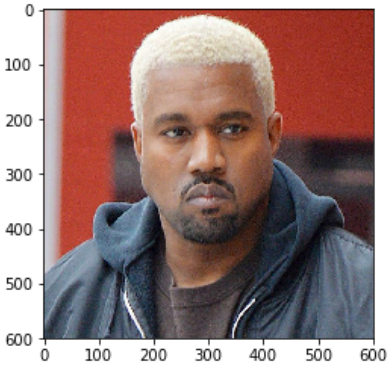
```
Beware: Game of Thrones content detected!
```

```
This image is not related to Game of Thrones.
```



```
Beware: Game of Thrones content detected!
```

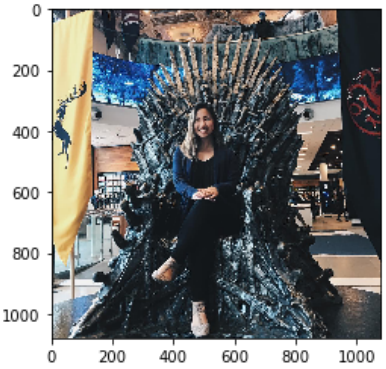
```
Beware: Game of Thrones content detected!
```



```
Beware: Game of Thrones content detected!
```

```
This image is not related to Game of Thrones.
```

```
This image is not related to Game of Thrones.
```



```
Beware: Game of Thrones content detected!
```

Out of the 17 sample images, 14 images were correctly classified (82%). The highlight from the model performance is that all the true positives were correct, meaning that all images relating to Game of Thrones were correctly classified as Game of Thrones related content. Another interesting aspect of the model's classifications is that it was able to successfully identify fan fiction images as being related to Game of Thrones, as seen in the image of Catelyn Stark as an undead (something that happened in the book, but not in the show) and Jon Snow in the Targaryen armor. The model was even able to identify Game of Thrones content in my own picture that I took myself of my girlfriend sitting on the Iron Throne (last image), which was on display at the AT&T Store in San Francisco. This shows that the model is able to learn the characteristics that are associated with Game of Thrones, and is robust enough to correctly identify images that did not even happen in the show, which is great for means of filtering potential spoilers.

For the lowlights of the model, it appears that images of people with blond hair will result in the model classifying the image as Game of Thrones related content, which can be seen in the incorrect labels of the images of blond Kanye West and Taylor Swift. This is a problem that I suspected would happen, due to the bulk of images of Daenerys Targaryen, a popular Game of Thrones character with notable blond hair, so this is a characteristic that the model has learn to associate with Game of Thrones. In addition, I threw in a Lord of the Rings image to test and as suspected, it was also classified as a Game of Thrones related image. In this image, the characteristics of medievil times, such as wielding swords and wearing armor, is something the model has learn to associate with Game of Thrones. However, as discussed earlier, there is a higher tolerance for false positives, and as long as all the true positives were correctly classified, this final model would be suitable for the task of filtering Game of Thrones related images for the sake of avoiding spoilers.

Improvements

There are improvements that can be made that would help decrease the number of false positives when the model is classifying images. For example, the "Negative" portion of the training dataset was fairly limited (used only hashtags for nike, nba, cars), and if this portion of the training data set was expanded to a larger variety of subject areas (possibly adding more medievil images NOT relating to Game of Thrones), the model would learn more features and characteristics that would help it further distinguish Game of Thrones related images during training, making it more robust in terms of avoiding false positives. Usually for image recognition tasks, or any machine learning task, larger datasets will help with model performace. In addition, another way we could improve overall performance is through transfer learning with a state-of-art CNN architecture, such as GoogleNet[3], which is the technique of taking a pretained model archictecture and just re-training the last classification layer tailored to the specific problem that is trying to be solved. Because the model is already pre-trained with weights and only the last classification layer has to be trained, this reduces the computational resources and time required for training. Furthermore, GoogleNet is known for its "Inception" modules, that help decrease the memory size of the model in terms of parameters, and would ultimately improve the scalability of the model.

Reflection

Inspired by the ongoing problem of unwarranted spoilers showing up on my Instagram feed, I wanted to see if a model could be implemented to help filter out images that contain Game of Thrones content. With convolutional neural networks being heavily used for image recognition tasks and often yielding great results, I knew that a CNN would be a perfect fit for the task at hand. The end to end process started with downloading images using the Instagram API and creating a benchmark model to test against. Once the benchmark result was obtained, I experimented with data augmentation and a CNN model architecture from scratch, and trained/fine-tuned the model until optimal results were obtained. With a final test accuracy of 81%, I believe the model does a pretty good job at classify images relating to Game of Thrones content and exceptionally well at avoiding false negatives. Although the model suffers from occassional false positives, as seen in the previous section, the main task of filtering images with any possible spoilers relating to Game of Thrones allows for a higher tolerance for false positives.

Unlike other machine learning projects I've worked on, this is the first one where there was no dataset available to use, and I would have to to gather and create one myself. I learned so much from the process of scraping the Instagram API, and has inspired to make use of data from other social media platforms, such as Twitter, for future projects.

In addition, this is the first project where my personal laptop would not computationally sufficient enough to train the model. This is where I would have to turn to AWS and their GPU computing power on their EC2 instances. I wanted to continue to make more improvements to my model by adding more data or possibly added more layers, but this would lead to a greater computational load. Because I had to rent an EC2 instance and pay by the hour, it was sort of a bottleneck and it was not feasible to make these types of further adjustments due to the amount of time it would require to train the model. To avoid this difficult resource bottleneck in the future, I plan to build my own deep learning workstation equipped with a Nvidia 1080 GTX Ti. Having my own deep learning workstation would allow me to train my models freely without worrying about hourly costs. This freedom alone would be worth the investment, and would allow me to work on as many deep learning projects as I wanted!

References

[1] Krizhevsky, A., Sutskever, I., and Hinton, G. E. ImageNet classification with deep convolutional neural networks. In NIPS, pp. 1106–1114, 2012.

[2] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. arXiv:1409.1556, 2014.

[3] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In CVPR, 2015.

[4] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. arXiv preprint arXiv:1512.03385, 2015.

[5] R. Arcega, Instagram Scraper, https://github.com/rarcega/instagram-scraper

[6] Kingma, Diederik P. and Ba, Jimmy. Adam: A Method for Stochastic Optimization. arXiv:1412.6980 [cs.LG], December 2014.

[7] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in International conference on artificial intelligence and statistics, 2010, pp. 249–256.

[8] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In ICML, 2015.

[9] A. Karpathy, Stanford CS231n Course Notes, http://cs231n.github.io/

[10] Keras: The Python Deep Learning Library, https://keras.io/

[11] TensorFlow, https://www.tensorflow.org/