Springboard Data Science Career Track Capstone Two – Apparel Image Classification

Rose Zdybel

Introduction

You are browsing the internet for an on-line item such as apparel. The browser finds images matching your criteria. How does the underlying mechanism work that locates relevant items? For example, how does it locate black pants or a white dress? The goal of this project is to explore the basic underlying mechanism used in image classification. We use an apparel image dataset.

Image classification is a well-known Machine Learning application. Its applications include general web browser searches such as Google images, retailer applications for identifying objects from user snapshots, facial recognition, and object recognition for autonomous vehicles.

Here we focus on an application in the apparel realm. It is inspired by understanding the underlying mechanism used by an application such as Stitch Fix , the on-line apparel site that allows a user to indicate styles they like and then matches clothing selections to the user's style. (NOTE: Stitch Fix does use personal consultants, in addition to any automation they may use).

Since image classification is a well-known problem, pre-trained models are available, and can be further customized for specific datasets. However, here, we do not use a pre-trained model, since the objective is two-fold:

- To get experience in the challenges from training a model from scratch
- To get the best-possible predictive capabilities from the model.

In this project, we use a subset of Kaggle's Apparel images dataset, which includes images for 24 predefined categories.

EDA

In this section we discuss exploratory data analysis (EDA) for the dataset, focusing on categories and image sizes.

Overview

The entire dataset consists of 11385 images and is composed of the categories and image counts listed in the table below. They are shown alphabetically on the left and sorted in descending image count order on the right. Categories highlighted in yellow are used in this project.

Category (Alphabetical)	Image Count		Category (Sorted by Image Count)	Image Count
black_dress	450		black_pants	871
black_pants	871		white_dress	818
black_shirt	715		red_dress	800
black_shoes	766		blue_pants	798
black_shorts	328		black_shoes	766
blue_dress	502		blue_shirt	741
blue_pants	798		black_shirt	715
blue_shirt	741		red_shoes	610
blue_shoes	523		white_shoes	600
blue_shorts	299		blue_shoes	523
brown_pants	311		blue_dress	502
brown_shoes	464		brown_shoes	464
brown_shorts	40		green_shoes	455
green_pants	227		black_dress	450
green_shirt	230		black_shorts	328
green_shoes	455		brown_pants	311
green_shorts	135		red_pants	308
red_dress	800		blue_shorts	299
red_pants	308		white_pants	274
red_shoes	610		green_shirt	230
white_dress	818		green_pants	227
white_pants	274		green_shorts	135
white_shoes	600		white_shorts	120
white_shorts	120		brown_shorts	40

The dataset was designed for multi-label image classification. However, here we treat it as a single label/multi-class problem, with the anticipation that predictive capabilities can be improved further by using a multi-label model in the future.

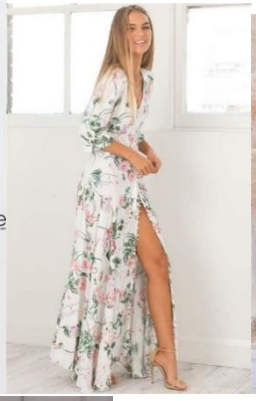
Sample Data

Examples of images from selected categories are shown in the following subsections.

Red_dress




White_dress






Black_shirt



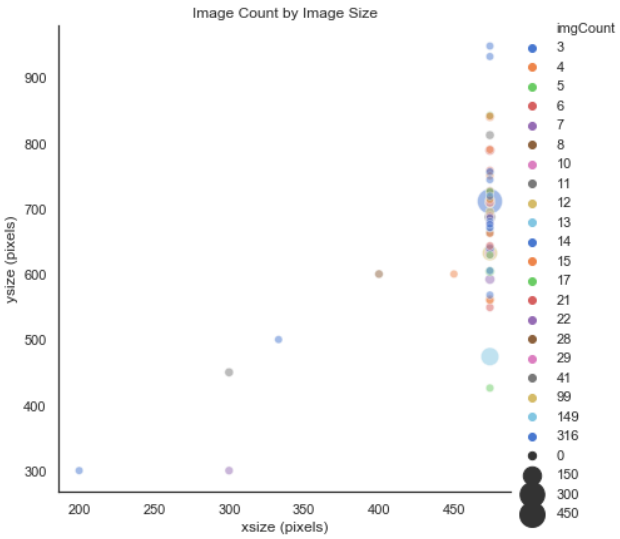

Image size

From the samples shown above, it is apparent that images sizes are not consistent. Image size can affect the predictive accuracy and training time of a model. Therefore, we experiment with different image sizes. We choose square sizes to experiment with, even though non-square could be used as well.

The five most common image sizes are:

xsize	ysize	imgCount
474	711	316
474	474	149
474	632	99
474	687	41
474	631	29

The scatter plot below further emphasizes that most images have an X-dimension of 474 pixels. The color and size of the marks indicate counts.



We initially used an image size of 474x474 but ran into training time and memory issues. We then experimented with smaller sizes (all square): 150x150, 200x200, 300x300. We ended up focusing on two sizes: 150x150 and 300x300.

NOTE: Ultimately the results indicate we should consider smaller sizes as well.

Approach and Model Development

We use a convolutional neural network (CNN) model, which is a commonly used machine learning model for image classification (google CNN development history).

We initially explored various CNN configurations and architectural complexity: we varied the block configuration (one or more convolution layers plus a pool layer) in the model, the number of blocks, and the number of nodes per layer. The results influenced our approach which is described in the *Initial model exploration* section. We then took a systematic approach, described in the second subsection.

Initial model exploration and issues

The initial experimentation was done on a laptop. The issues encountered are categorized below. We discuss tactics in the next subsection.

- Compute resource limitations
    - Memory shortage for larger image sizes and larger number of parameters.
- Elongated Training time for some parameters and model configurations
    - GPU not available on local laptop.
- Loss function not improving over time (Gradient not descending)
    - We suspect due to poor learning rate (LR) choice.
- Model Accuracy

Issue Tactics

In this section we discuss the tactics we took to address the issues. The table below lists each tactic and its anticipated impact: negative (-), positive (+), or unknown (+/-) impact. NOTE: this is impact, not values; for example, a reduction in training time is a positive impact.

Figure 1: Anticipated impact of tactics: (-) negative; (+) positive; (+/-) unknown. ***NOTE****: this is impact, not values; for example, a reduction in training time is a positive impact*

Tactic	Training time	Model Stability	CPU resources	Memory resources	Accuracy
Colab Env	+		+	+	
Data set size reduction	+				-
Image size reduction options	+			+	+/-
Optimizer/LR options		+			+/-

Environment

- Use Google Colab as the platform, rather than local laptop
    - Colab can make more **memory** available
    - Colab can be configured for GPU (which was unavailable on local laptop)
        - Use of GPU usually reduces **training time**.
    - From the website:
        - *Colaboratory, or "Colab" for short, allows you to write and execute Python in your browser, with*
            - *Zero configuration required*
            - *Free access to GPUs*
            - *Easy sharing*

Goals

- Reduce training time
    - Increase CPU throughput using GPU
- Alleviate memory constraint present on laptop

Sample Data Set Size

To reduce the training time, we use a subset of the data: of the original 24 classes, we use the 10 having the greatest number of samples.

Goal

- Reduce training time

Image Size

Larger image sizes increase training time, since the convolution must be applied a greater number of times. Image size can also impact predictive capability. Therefore, we explore the effect various image re-sizing the input images has on predictive capability; We explore the impact of various sizes.

Goal

- Reduce training / prediction time
- Improve predictive capability

Optimizers and Learning Rate

In neural networks, learning rate can be one of the most important parameters, but involves a trade-off between training time and model convergence.

Optimizers, such as Adam/AMSGrad automatically adjust learning rates during training; however, the user must supply an initial rate. Other optimizers, such as SGD can be configured with a user-specified learning-rate scheduler.

The choice of optimizer can also affect training time. We explore the impact of various optimizers and learning rates.

Goal

- Improve loss function gradient
- Reduce convergence time
- Balance training time/model accuracy tradeoff

Systematic "Experiments": Setup and Parameterization

We 'experiment' with a variety of model architectures and parameter configurations. In this section we describe the details. We refer to each model/parameter combination as an 'experiment'.

Setup

Data splits

Data split used for Train/Validation/Test Data is 70/15/15:

Train Data: obviously used for training the model

Validation Data: used for early stopping in which the loss does not decrease within a certain number of epochs, specified as the 'patience' parameter in the Early stopping callback. We use 8 epochs.

Test (Holdout) Data: used to measure model performance.

Metric (Model performance)

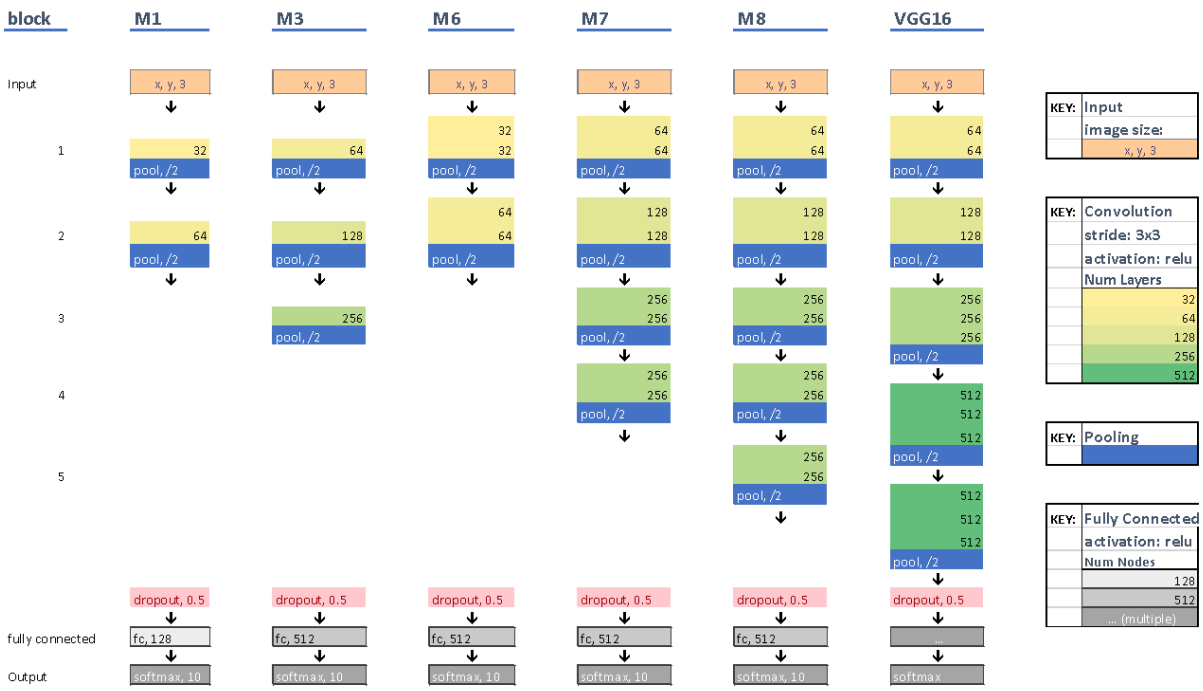
The metric used to measure model performance is prediction accuracy for the test data.

Model configuration

The model architectures used in the final comparison are summarized in the table below; they were chosen after initial exploration on a variety of models, and therefore the gaps in numbering.

The table shows: the size of the input; the block configurations consisting of the size of the convolution layer(s) and pooling layer; the size of the hidden fully connected layer; and the output layer. As a comparison, we show the configuration of the award-winning VGG16 network (VGG16 Overview). **NOTE**: we did not implement VGG16 due to time constraints and leave that for future work consideration.

Figure 2: Model Configurations Summary



Optimizers

From web posts and discussions (e.g., gradient descent discussion, heuristic experiment discussion), two popular optimizers are Stochastic Gradient Descent (SGD) and Adam, each having variations. Here we consider SGD with a learning rate decay (LR scheduler tutorial), and Keras Adam optimizer with the AMSGrad option. The AMSGrad (AMSGrad paper) option addresses a convergence issue that can occur with basic Adam.

NOTE: in this report, we use AMSGrad or Adam/AMSGrad to refer to the Keras Adam optimizer with the AMSGrad option.

Pros and cons summary of the optimizers:

- Stochastic Gradient Descent (SGD):
    - Slower to converge than Adam
        - (i.e., can be slower to train)
    - Can provide better results than alternatives in some cases
    - Option for LR-scheduler implementation
- Adam and its variations (e.g., AMSGrad option)
    - built-in mechanism that dynamically adjusts learning rates
    - faster to converge than SGD
        - (i.e., can be faster to train)
    - competitive results with SGD

Learning Rate (LR)

We initially tried using the Adam default learning rate of 0.01. However, the loss function did not improve after the first few epochs for some of the initial models, so we experiment with the use of lower values.

We ultimately use LR-finder (LR-Finder Tutorial) to determine the optimal LR for both Adam/AMSGrad and SGD.

Early stopping callback

We implement early stopping with max of 250 epochs and patience of 8 epochs; all model configurations hit early stopping.

Results

We ran experiments for various combinations of parameters, similar to a python ML grid search. However, due to time constraints, we did not run all combinations; instead, we ran the most promising combinations based on initial experimental results. The option/values grid is shown below.

Table 1: Parameter Grid

OPTION	VALUES
Optimizer	SGD , AMSGRAD(AMSGrad)
Learning Rate	.01, 0.001, .0001
Models	M1, M3, M6, M7, M8
Image Size (“pixels”)	150x150, 300x300

The table below shows results for the experimental combinations we tried:

optimizer	lr	pixels	M1	M3	M6	M7	M8
Sgd	0.01	150	0.8879	0.9228	0.8796	0.9449	0.9421
Sgd	0.01	300	None	None	None	None	0.9357
Sgd	0.001	150	None	None	None	None	None
Sgd	0.001	300	None	None	None	None	None
Sgd	0.0001	150	None	None	None	None	None
Sgd	0.0001	300	None	None	None	None	None
amsgrad	0.01	150	None	None	None	None	None
amsgrad	0.01	300	None	None	None	None	None
amsgrad	0.001	150	0.9136	0.9476	0.9274	0.9449	0.9513
amsgrad	0.001	300	0.909	0.9403	0.9081	0.943	0.9338
amsgrad	0.0001	150	None	0.9292	None	0.943	None
amsgrad	0.0001	300	None	None	None	None	None

Results by parameters

We show results for various combinations of parameters (by freezing the best in class for the other parameters). A reminder that the comparison metric is test data prediction accuracy.

Optimizer Comparison

AMSGrad did better or the same as SGD for all experiments.

Frozen:

Option	Value
Pixels	150
LR	SGD (.01); AMSGrad (.001)

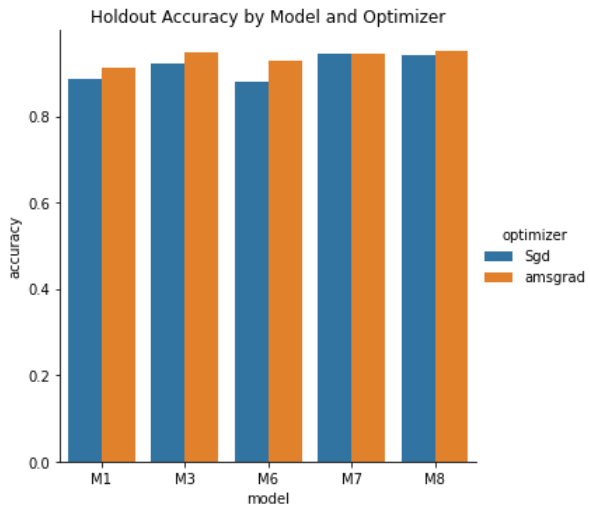
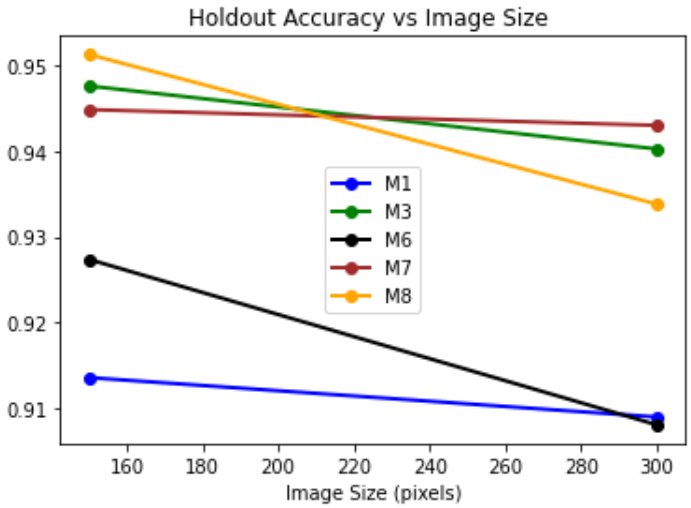


Image Size (Pixels) Comparison

The smaller (150x150) image size performed better than the larger (300x300) image size. NOTE: This implies that we should consider smaller sizes in future experiments, to see if they perform even better.

Frozen:

Option	Value
Optimizer	AMSGrad
LR	.001

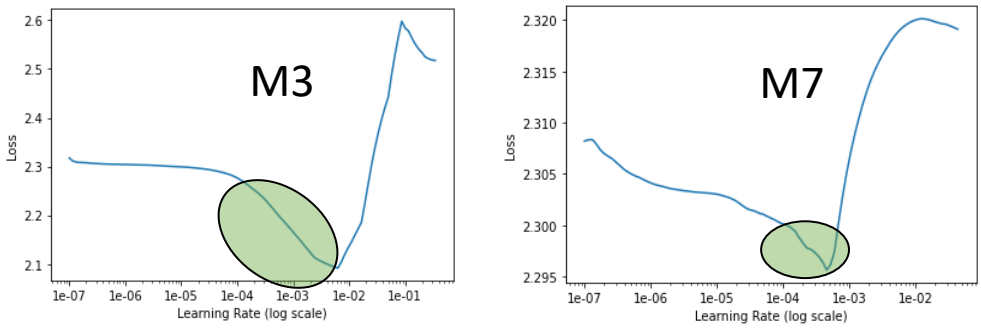


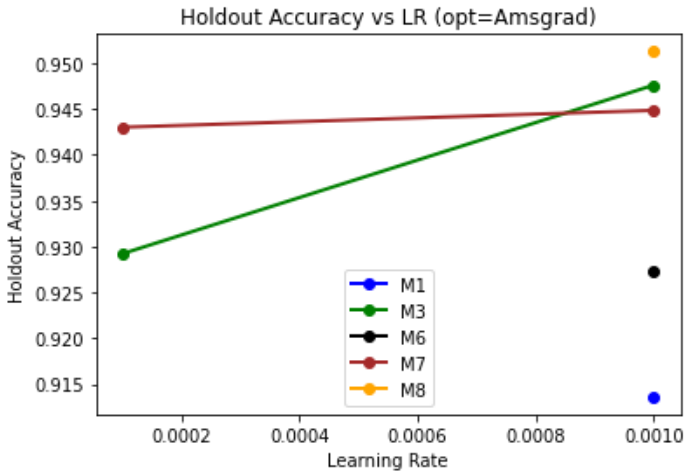
Learning Rate Comparison

We used LR-finder to identify the best LR for each optimizer/model combination. Therefore, we did not run all combinations of LRs. Instead, we spot-checked the LR-finder recommendations by doing two comparisons and verified that the LR-finder recommendations were reasonable. For AMSGrad, LR-finder recommended a learning rate of around 0.001 for all the models. The graph below confirms that it is better than 0.0001 for both the models we cross-checked. As mentioned previously, an LR of 0.01 was unstable for some models during initial experimentation.

Configuration:

Option	Value
Optimizer	AMSGrad
Models	M3, M7
Pixels	150x150
LRs	0.0001, 0.001





Best Model

In this section we discuss the best model and its predictive capabilities. Again, the metric used is highest predictive accuracy for the test data.

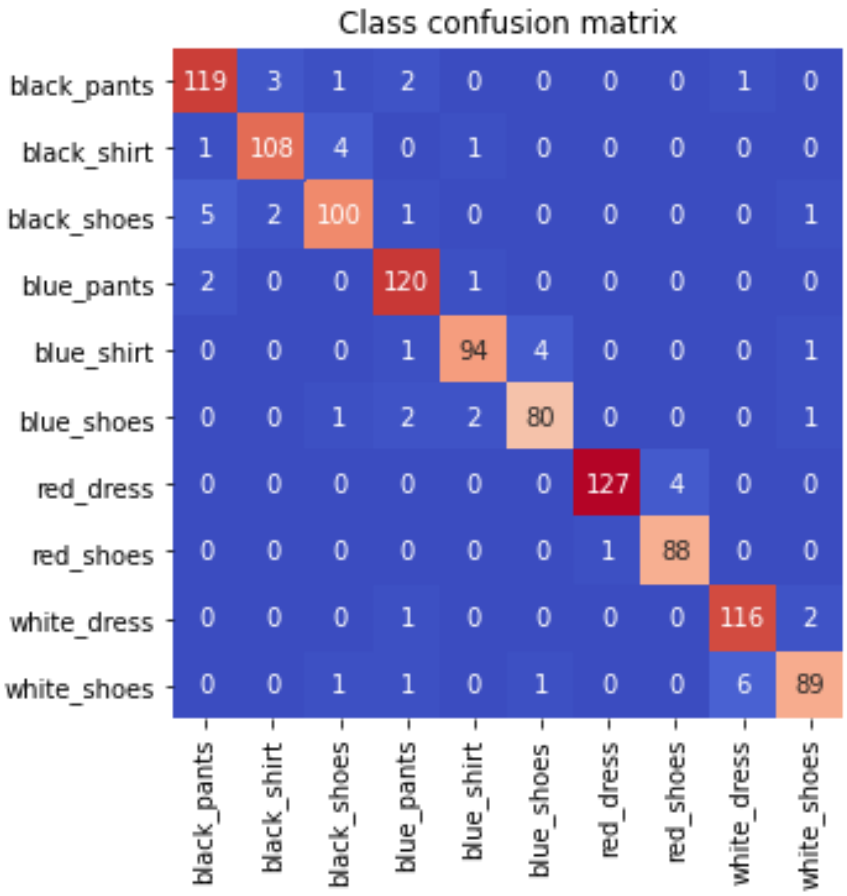
Parameters

The highest test data accuracy score of 0.951 was achieved with Model 8 (M8), an image size of 150x150, and the AMSGrad optimizer with a learning rate of 0.001 . The parameters are summarized in the table below. Since the best model was the one with the most blocks/layer, this implies we should consider models with additional blocks/layers.

Test Data Accuracy	0.951
Model	M8
Image Size	150 x 150
Optimizer	Amsgrad
Learning Rate	.001

Confusion Matrix

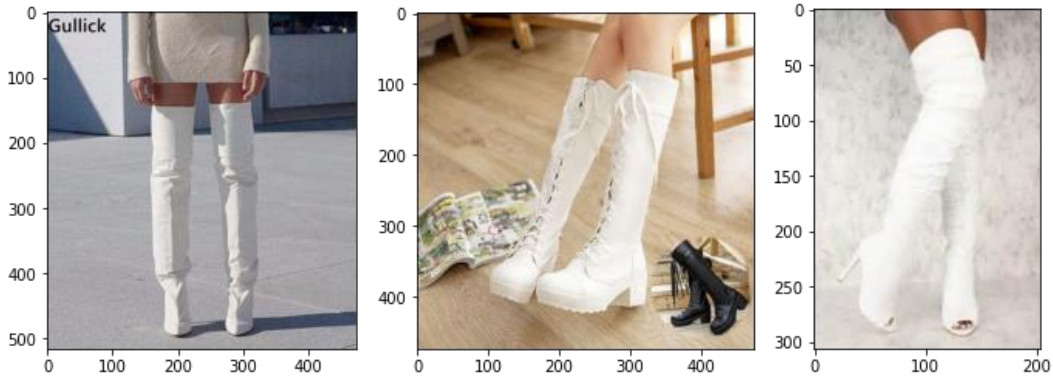
The confusion matrix shown below indicates the most problematic cases were predicting 5 black_shoes images as black_pants and predicting 6 white_shoes images as white_dress. We discuss misclassified images in the next section.



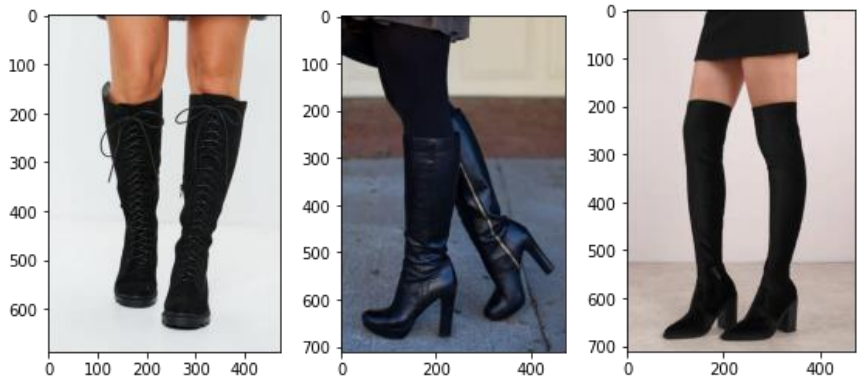
Mis-Classified image examples

The examples below imply that the model was having problems with high boots. For high white_boots it appears decorative ones have features similar to white_dresses. Likewise, some high black_boots have features similar to black_pants.

White_shoes predicted as white_dress



Black_shoes predicted as black_pants



Conclusion

- We use the Kaggle Apparel Image dataset to classify images
- We explore training a convolutional neural network from scratch
- Use Colab due to local compute resource limitations
- Explore the effect of influential parameters
- Develop a model that gives predictive capability of 95% accuracy.
- Future Work
    - Investigate the use of a multi-class model
        - Potential for better predictive capability
    - Investigate a wider range of parameter values
        - Image sizes

        - Try smaller sizes
    - Alternative Optimizers and options
- Investigate models with more blocks/layers
- Investigate the use of one or more pre-trained models

Appendix A - Model Configuration Samples

M1

- blocks: 2
- conv/block: 1
- Dense nodes: 128

M1 Definition

```
Model1= tf.keras.models.Sequential([
  # Note the input shape is the desired size of the image (e.g, 150x150) with 3 bytes color
   # This is the first convolution
   tf.keras.layers.Conv2D(32, (3,3), activation='relu', input_shape=input_shape),
   tf.keras.layers.MaxPooling2D(2, 2),
   # The second convolution
   tf.keras.layers.Conv2D(64, (3,3), activation='relu'),
   tf.keras.layers.MaxPooling2D(2,2),
   # Flatten the results to feed into a DNN
   tf.keras.layers.Flatten(),
   tf.keras.layers.Dropout(0.5),
   # 128 neuron hidden layer
   tf.keras.layers.Dense(128, activation='relu'),
   tf.keras.layers.Dense(num_classes, activation='softmax')
])
```

M1.summary()

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_9 (Conv2D)            (None, 148, 148, 32)      896
_________________________________________________________________
max_pooling2d_7 (MaxPooling2 (None, 74, 74, 32)        0
_________________________________________________________________
conv2d_10 (Conv2D)           (None, 72, 72, 64)        18496
_________________________________________________________________
max_pooling2d_8 (MaxPooling2 (None, 36, 36, 64)        0
_________________________________________________________________
flatten_3 (Flatten)          (None, 82944)             0
_________________________________________________________________
dropout_2 (Dropout)          (None, 82944)             0
_________________________________________________________________
dense_6 (Dense)              (None, 128)               10616960
_________________________________________________________________
dense_7 (Dense)              (None, 10)                1290
=================================================================
Total params: 10,637,642
Trainable params: 10,637,642
Non-trainable params: 0
```

M6

- blocks: 2
- conv/block: 2
- Dense nodes: 512

M6 definition

```
model6 = tf.keras.models.Sequential([
  # Note the input shape is the desired size of the image (e.g, 150x150) with 3 bytes color
   # This is the first convolution
   tf.keras.layers.Conv2D(32, (3,3), activation='relu', input_shape=input_shape),
   tf.keras.layers.Conv2D(32, (3,3), activation='relu'),
   tf.keras.layers.MaxPooling2D(2, 2),
   # The second convolution
   tf.keras.layers.Conv2D(64, (3,3), activation='relu'),
   tf.keras.layers.Conv2D(64, (3,3), activation='relu'),
   tf.keras.layers.MaxPooling2D(2,2),
   # Flatten the results to feed into a DNN
   tf.keras.layers.Flatten(),
   tf.keras.layers.Dropout(0.5),
   # 512 neuron hidden layer
   tf.keras.layers.Dense(512, activation='relu'),
   tf.keras.layers.Dense(num_classes, activation='softmax')
])
```

M6.summary()

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_2 (Conv2D)            (None, 148, 148, 32)      896
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 146, 146, 32)      9248
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 73, 73, 32)        0
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 71, 71, 64)        18496
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 69, 69, 64)        36928
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 34, 34, 64)        0
_________________________________________________________________
flatten_1 (Flatten)          (None, 73984)             0
_________________________________________________________________
dropout (Dropout)            (None, 73984)             0
_________________________________________________________________
dense_2 (Dense)              (None, 512)               37880320
_________________________________________________________________
dense_3 (Dense)              (None, 10)                5130
=================================================================
Total params: 37,951,018
Trainable params: 37,951,018
Non-trainable params: 0
```