convolutional-neural-networks-cnns/) — please refer to it if you have any questions on the architecture or are simply looking for more detail. If you're looking to design your own models, you'll want to pick up a copy of my book, ***Deep Learning for Computer Vision with Python (https://pyimagesearch.com/deep-learning-computer-vision-python-book/)***.

Ensure you've used the ***"Downloads"*** section at the bottom of this blog post to grab the source code + example images. From there, open up the `smallervggnet.py` file in the `pyimagesearch` module to follow along:

→ **Launch Jupyter Notebook on Google Colab**

Multi-label classification with Keras

```
1. | # import the necessary packages
2. | from tensorflow.keras.models import Sequential
3. | from tensorflow.keras.layers import BatchNormalization
4. | from tensorflow.keras.layers import Conv2D
5. | from tensorflow.keras.layers import MaxPooling2D
6. | from tensorflow.keras.layers import Activation
7. | from tensorflow.keras.layers import Flatten
8. | from tensorflow.keras.layers import Dropout
9. | from tensorflow.keras.layers import Dense
10. | from tensorflow.keras import backend as K
```

On **Lines 2-10**, we import the relevant Keras modules and from there, we create our `SmallerVGGNet` class:

→ **Launch Jupyter Notebook on Google Colab**

Multi-label classification with Keras

```
12. | class SmallerVGGNet:
13. |     @staticmethod
14. |     def build(width, height, depth, classes, finalAct="softmax"):
15. |         # initialize the model along with the input shape to be
16. |         # "channels last" and the channels dimension itself
17. |         model = Sequential()
18. |         inputShape = (height, width, depth)
19. |         chanDim = -1
20. |
21. |         # if we are using "channels first", update the input shape
22. |         # and channels dimension
23. |         if K.image_data_format() == "channels_first":
24. |             inputShape = (depth, height, width)
25. |             chanDim = 1
```

Our class is defined on **Line 12**. We then define the `build` function on **Line 14**, responsible for assembling the convolutional neural network.

The `build` method requires four parameters — `width` , `height` , `depth` , and `classes` . The `depth` specifies the number of channels in an input image, and `classes` is the number (integer) of categories/classes (not the class labels themselves). We'll use these parameters in our training script to instantiate the model with a `96 x 96 x 3` input volume.

The optional argument, `finalAct` (with a default value of `"softmax"`) will be utilized ***at the end*** of the network architecture. Changing this value from softmax to sigmoid will enable us to perform multi-label classification with Keras.

Keep in mind that this behavior is ***different*** than our original implementation of `SmallerVGGNet` in our previous post — we are adding it here so we can control whether we are performing simple classification or multi-class classification.

From there, we enter the body of `build` , initializing the `model` (**Line 17**) and defaulting to `"channels_last"` architecture on **Lines 18 and 19** (with a convenient switch for backends that support `"channels_first"` architecture on **Lines 23-25**).

Let's build the first `CONV => RELU => POOL` block:

→ **Launch Jupyter Notebook on Google Colab**

Multi-label classification with Keras

```
27. |         # CONV => RELU => POOL
28. |         model.add(Conv2D(32, (3, 3), padding="same",
29. |             input_shape=inputShape))
30. |         model.add(Activation("relu"))
31. |         model.add(BatchNormalization(axis=chanDim))
32. |         model.add(MaxPooling2D(pool_size=(3, 3)))
33. |         model.add(Dropout(0.25))
```

Our `CONV` layer has `32` filters with a `3 x 3` kernel and `RELU` activation (Rectified Linear Unit). We apply batch normalization, max pooling, and 25% dropout.

Dropout is the process of randomly disconnecting nodes from the *current* layer to the *next* layer. This process of random disconnects naturally helps the network to reduce overfitting as no one single node in the layer will be responsible for predicting a certain class, object, edge, or corner.

From there we have two sets of `(CONV => RELU) * 2 => POOL` blocks:

→ Launch Jupyter Notebook on Google Colab

Click here to download the source code to this post

Multi-label classification with Keras

```
        # (CONV => RELU) * 2 => POOL
        model.add(Conv2D(64, (3, 3), padding="same"))
        model.add(Activation("relu"))
        model.add(BatchNormalization(axis=chanDim))
        model.add(Conv2D(64, (3, 3), padding="same"))
        model.add(Activation("relu"))
        model.add(BatchNormalization(axis=chanDim))
        model.add(MaxPooling2D(pool_size=(2, 2)))
        model.add(Dropout(0.25))

        # (CONV => RELU) * 2 => POOL
        model.add(Conv2D(128, (3, 3), padding="same"))
        model.add(Activation("relu"))
        model.add(BatchNormalization(axis=chanDim))
        model.add(Conv2D(128, (3, 3), padding="same"))
        model.add(Activation("relu"))
        model.add(BatchNormalization(axis=chanDim))
        model.add(MaxPooling2D(pool_size=(2, 2)))
        model.add(Dropout(0.25))
```

Notice the numbers of filters, kernels, and pool sizes in this code block which work together to progressively reduce the spatial size but increase depth.

These blocks are followed by our only set of `FC => RELU` layers:

→ Launch Jupyter Notebook on Google Colab

Multi-label classification with Keras

```
55.         # first (and only) set of FC => RELU layers
56.         model.add(Flatten())
57.         model.add(Dense(1024))
58.         model.add(Activation("relu"))
59.         model.add(BatchNormalization())
60.         model.add(Dropout(0.5))
61.
62.         # softmax classifier
63.         model.add(Dense(classes))
64.         model.add(Activation(finalAct))
65.
66.         # return the constructed network architecture
67.         return model
```

Fully connected layers are placed at the end of the network (specified by `Dense` on **Lines 57 and 63**).

Line 64 is ***important*** for our multi-label classification — `finalAct` dictates whether we'll use `"softmax"` activation for single-label classification or `"sigmoid"` activation **in the case of today's multi-label classification.** Refer to **Line 14** of this

Multi-label classification with Keras

Click here to download the source code to this post

```
$ python
>>> import os
>>> labels = []
>>> imagePath = "dataset/red_dress/long_dress_from_macys_red.png"
>>> l = label = imagePath.split(os.path.sep)[-2].split("_")
>>> l
['red', 'dress']
>>> labels.append(l)
>>>
>>> imagePath = "dataset/blue_jeans/stylish_blue_jeans_from_your_favorite_store.png"
>>> l = label = imagePath.split(os.path.sep)[-2].split("_")
>>> labels.append(l)
>>>
>>> imagePath = "dataset/red_shirt/red_shirt_from_target.png"
>>> l = label = imagePath.split(os.path.sep)[-2].split("_")
>>> labels.append(l)
>>>
>>> labels
[['red', 'dress'], ['blue', 'jeans'], ['red', 'shirt']]
```

As you can see, the `labels` list is a "list of lists" — each element of `labels` is a 2-element list. The two labels for each list is constructed based on the file path of the input image.

We're not quite done with preprocessing:

→ **Launch Jupyter Notebook on Google Colab**

Multi-label classification with Keras

```
# scale the raw pixel intensities to the range [0, 1]
data = np.array(data, dtype="float") / 255.0
labels = np.array(labels)
print("[INFO] data matrix: {} images ({:.2f}MB)".format(
    len(imagePaths), data.nbytes / (1024 * 1000.0)))
```

Our `data` list contains images stored as NumPy arrays. In a single line of code, we convert the list to a NumPy array and scale the pixel intensities to the range `[0, 1]` .

We also convert labels to a NumPy array as well.

From there, let's binarize the labels — the below block is ***critical*** for this week's multi-class classification concept:

→ **Launch Jupyter Notebook on Google Colab**

Multi-label classification with Keras

```
# binarize the labels using scikit-learn's special multi-label
# binarizer implementation
```

```
75. | print("[INFO] class labels:")
76. | mlb = MultiLabelBinarizer()
77. | labels = mlb.fit_transform(labels)
78. |
79. | # loop over each of the possible class labels and show them
80. | for (i, label) in enumerate(mlb.classes_):
81. |     print("{}. {}".format(i + 1, label))
```

Click here to download the source code to this post

In order to binarize our labels for multi-class classification, we need to utilize the scikit-learn library's **MultiLabelBinarizer (http://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MultiLabelBinarizer.html)** class. You *cannot* use the standard `LabelBinarizer` class for multi-class classification. **Lines 76 and 77** fit and transform our human-readable labels into a vector that encodes which class(es) are present in the image.

Here's an example showing how `MultiLabelBinarizer` transforms a tuple of `("red", "dress")` to a vector with six total categories:

→ **Launch Jupyter Notebook on Google Colab**

Multi-label classification with Keras

```
1.  | $ python
2.  | >>> from sklearn.preprocessing import MultiLabelBinarizer
3.  | >>> labels = [
4.  | ...     ("blue", "jeans"),
5.  | ...     ("blue", "dress"),
6.  | ...     ("red", "dress"),
7.  | ...     ("red", "shirt"),
8.  | ...     ("blue", "shirt"),
9.  | ...     ("black", "jeans")
10. | ... ]
11. | >>> mlb = MultiLabelBinarizer()
12. | >>> mlb.fit(labels)
13. | MultiLabelBinarizer(classes=None, sparse_output=False)
14. | >>> mlb.classes_
15. | array(['black', 'blue', 'dress', 'jeans', 'red', 'shirt'], dtype=object)
16. | >>> mlb.transform([("red", "dress")])
17. | array([[0, 0, 1, 0, 1, 0]])
```

One-hot encoding transforms categorical labels from a single integer to a vector. The same concept applies to **Lines 16 and 17** except this is a case of two-hot encoding.

Notice how on **Line 17** of the Python shell (not to be confused with the code blocks for `train.py`) two categorical labels are "hot" (represented by a *"1"* in the array), indicating the presence of each label. In this case "dress" and "red" are hot in the array (**Lines 14-17**). All other labels have a value of "0".

array (**Lines 14-17**). All other labels have a value of "0".

Click here to download the source code to this post

Let's construct the training and testing splits as well as initialize the data augmenter:

→ Launch Jupyter Notebook on Google Colab

Multi-label classification with Keras

```
83. # partition the data into training and testing splits using 80% of
84. # the data for training and the remaining 20% for testing
85. (trainX, testX, trainY, testY) = train_test_split(data,
86.     labels, test_size=0.2, random_state=42)
87.
88. # construct the image generator for data augmentation
89. aug = ImageDataGenerator(rotation_range=25, width_shift_range=0.1,
90.     height_shift_range=0.1, shear_range=0.2, zoom_range=0.2,
91.     horizontal_flip=True, fill_mode="nearest")
```

Splitting the data for training and testing is common in machine learning practice — I've allocated 80% of the images for training data and 20% for testing data. This is handled by scikit-learn on **Lines 85 and 86**.

Our data augmenter object is initialized on **Lines 89-91**. Data augmentation is a best practice and a most-likely a "must" if you are working with less than 1,000 images per class.

Next, let's build the model and initialize the Adam optimizer:

→ Launch Jupyter Notebook on Google Colab

Multi-label classification with Keras

```
93. # initialize the model using a sigmoid activation as the final layer
94. # in the network so we can perform multi-label classification
95. print("[INFO] compiling model...")
96. model = SmallerVGGNet.build(
97.     width=IMAGE_DIMS[1], height=IMAGE_DIMS[0],
98.     depth=IMAGE_DIMS[2], classes=len(mlb.classes_),
99.     finalAct="sigmoid")
100.
101. # initialize the optimizer (SGD is sufficient)
102. opt = Adam(lr=INIT_LR, decay=INIT_LR / EPOCHS)
```

On **Lines 96-99** we build our `SmallerVGGNet` model, noting the `finalAct="sigmoid"` parameter indicating that we'll be performing **multi-label classification**.

From there, we'll compile the model and kick off training (this could take a while depending on your hardware).

→ **Launch Jupyter Notebook on Google Colab**

Multi-label classification with Keras

```
104. | # compile the model using binary cross-entropy rather than
105. | # categorical cross-entropy -- this may seem counterintuitive for
106. | # multi-label classification, but keep in mind that the goal here
107. | # is to treat each output label as an independent Bernoulli
108. | # distribution
109. | model.compile(loss="binary_crossentropy", optimizer=opt,
110. |     metrics=["accuracy"])
111. |
112. | # train the network
113. | print("[INFO] training network...")
114. | H = model.fit(
115. |     x=aug.flow(trainX, trainY, batch_size=BS),
116. |     validation_data=(testX, testY),
117. |     steps_per_epoch=len(trainX) // BS,
118. |     epochs=EPOCHS, verbose=1)
```

2020-06-12 Update: *Formerly, TensorFlow/Keras required use of a method called* `.fit_generator` *in order to accomplish data augmentation. Now, the* `.fit` *method can handle data augmentation as well, making for more-consistent code. This also applies to the migration from* `.predict_generator` *to* `.predict`*. Be sure to check out my articles about* ***fit and fit_generator (https://pyimagesearch.com/2018/12/24/how-to-use-keras-fit-and-fit_generator-a-hands-on-tutorial/)*** *as well as* ***data augmentation (https://pyimagesearch.com/2019/07/08/keras-imagedatagenerator-and-data-augmentation/)****.*

On **Lines 109 and 110** we compile the model using **binary cross-entropy** rather than **categorical cross-entropy**.

This may seem counterintuitive for multi-label classification; however, the goal is to treat each output label as an *independent Bernoulli distribution* and *we want to penalize each output node independently*.

From there we launch the training process with our data augmentation generator (**Lines 114-118**).

After training is complete we can save our model and label binarizer to disk:

→ **Launch Jupyter Notebook on Google Colab**

Multi-label classification with Keras

```
# save the model to disk
print("[INFO] serializing network...")
model.save(args["model"], save_format="h5")

# save the multi-label binarizer to disk
print("[INFO] serializing label binarizer...")
f = open(args["labelbin"], "wb")
f.write(pickle.dumps(mlb))
f.close()
```

2020-06-12 Update: *Note that for TensorFlow 2.0+ we recommend explicitly setting the `save_format="h5"` (HDF5 format).*

From there, we plot accuracy and loss:

→ **Launch Jupyter Notebook on Google Colab**

Multi-label classification with Keras

```
130. # plot the training loss and accuracy
131. plt.style.use("ggplot")
132. plt.figure()
133. N = EPOCHS
134. plt.plot(np.arange(0, N), H.history["loss"], label="train_loss")
135. plt.plot(np.arange(0, N), H.history["val_loss"], label="val_loss")
136. plt.plot(np.arange(0, N), H.history["accuracy"], label="train_acc")
137. plt.plot(np.arange(0, N), H.history["val_accuracy"], label="val_acc")
138. plt.title("Training Loss and Accuracy")
139. plt.xlabel("Epoch #")
140. plt.ylabel("Loss/Accuracy")
141. plt.legend(loc="upper left")
142. plt.savefig(args["plot"])
```

2020-06-12 Update: *In order for this plotting snippet to be TensorFlow 2+ compatible the `H.history` dictionary keys are updated to fully spell out "accuracy" sans "acc" (i.e., `H.history["val_accuracy"]` and `H.history["accuracy"]`). It is semi-confusing that "val" is not spelled out as "validation"; we have to learn to love and live with the API and always remember that it is a work in progress that many developers around the world contribute to.*

Accuracy + loss for training and validation is plotted on **Lines 131-141**. The plot is saved as an image file on **Line 142**.

In my opinion, the training plot is just as important as the model itself. I typically go through a few iterations of training and viewing the plot before I'm satisfied to share with you on the blog.

Click here to download the source code to this post

I like to save plots to disk during this iterative process for a couple reasons: (1) I'm on a headless server and don't want to rely on X-forwarding, and (2) I don't want to forget to save the plot (even if I am using X-forwarding or if I'm on a machine with a graphical desktop).

Recall that we changed the matplotlib backend on **Line 3** of the script up above to facilitate saving to disk.

Training a Keras network for multi-label classification

Don't forget to use the ***"Downloads"*** section of this post to download the code, dataset, and pre-trained model (just in case you don't want to train the model yourself).

If you want to train the model yourself, open a terminal. From there, navigate to the project directory, and execute the following command:

→ **Launch Jupyter Notebook on Google Colab**

Multi-label classification with Keras

```
$ python train.py --dataset dataset --model fashion.model \
   --labelbin mlb.pickle
Using TensorFlow backend.
[INFO] loading images...
[INFO] data matrix: 2165 images (467.64MB)
[INFO] class labels:
1. black
2. blue
3. dress
4. jeans
5. red
6. shirt
[INFO] compiling model...
[INFO] training network...
Epoch 1/30
54/54 [==============================] - 2s 35ms/step - loss: 0.3184 - accuracy: 0.8774 - val_loss: 1.1824 - val_accuracy: 0.6251
Epoch 2/30
54/54 [==============================] - 2s 37ms/step - loss: 0.1881 - accuracy: 0.9427 - val_loss: 1.4268 - val_accuracy: 0.6255
Epoch 3/30
54/54 [==============================] - 2s 38ms/step - loss: 0.1551 - accuracy: 0.9471 - val_loss: 1.0533 - val_accuracy: 0.6305
```

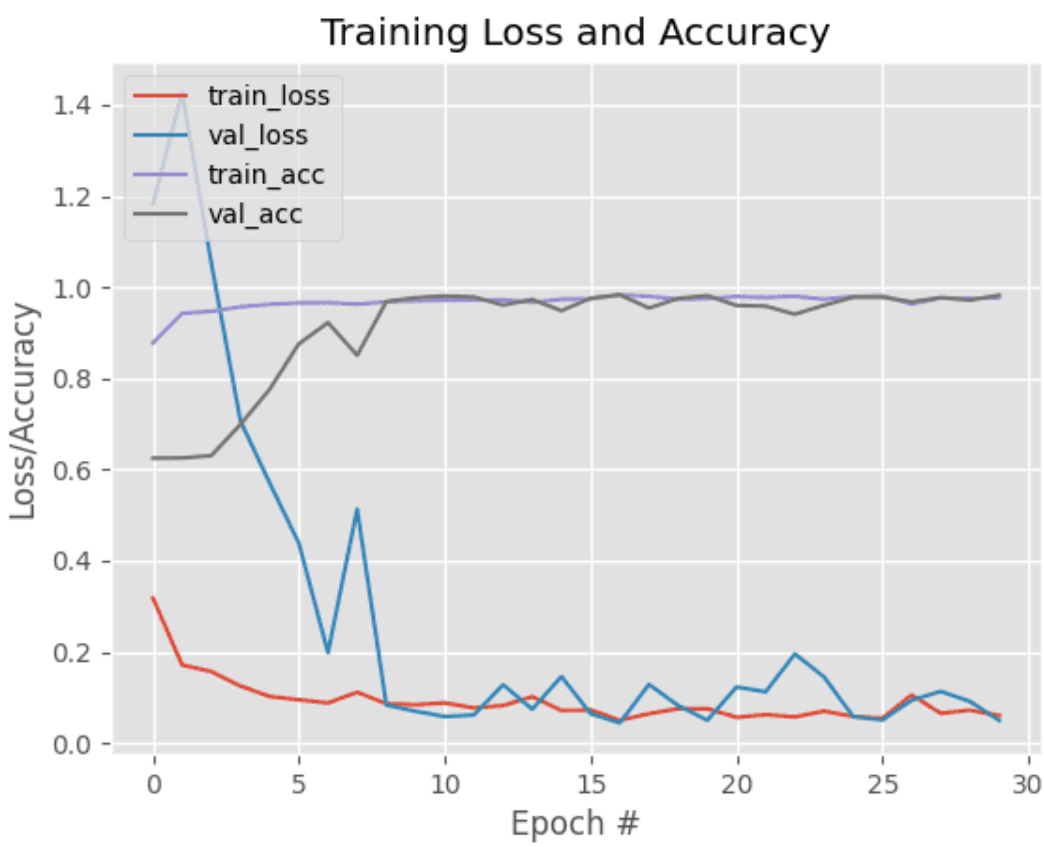
```
 ...
 Epoch 28/30
 54/54 [==============================] - 2s 41ms/step - loss: 0.0656 - accuracy: 0.9763 -
val_loss: 0.1[illegible] - val_accuracy: 0.9[illegible]
 Epoch 29/30
 54/54 [==============================] - 2s 40ms/step - loss: 0.0801 - accuracy: 0.9751 -
val_loss: 0.0916 - val_accuracy: 0.9715
 Epoch 30/30
 54/54 [==============================] - 2s 37ms/step - loss: 0.0636 - accuracy: 0.9770 -
val_loss: 0.0500 - val_accuracy: 0.9823
 [INFO] serializing network...
 [INFO] serializing label binarizer...
```

Click here to download the source code to this post

As you can see, we trained the network for 30 epochs, achieving:

- **97.70%** multi-label classification accuracy on the training set
- **98.23%** multi-label classification accuracy on the testing set

The training plot is shown in **Figure 3:**



(https://pyimagesearch.com/wp-content/uploads/2018/05/keras_multi_label_plot.png)

Figure 3: Our Keras deep learning multi-label classification accuracy/loss graph on the training and validation data

and validation data.

Click here to download the source code to this post

Applying Keras multi-label classification to new images

Now that our multi-label classification Keras model is trained, let's apply it to images *outside* of our testing set.

This script is quite similar to the `classify.py` script in my **previous post (https://pyimagesearch.com/2018/04/16/keras-and-convolutional-neural-networks-cnns/)** — be sure to look out for the multi-label differences.

When you're ready, open create a new file in the project directory named `classify.py` and insert the following code (or follow along with the file included with the ***"Downloads"***):

→ Launch Jupyter Notebook on Google Colab

Multi-label classification with Keras

```
1. | # import the necessary packages
2. | from tensorflow.keras.preprocessing.image import img_to_array
3. | from tensorflow.keras.models import load_model
4. | import numpy as np
5. | import argparse
6. | import imutils
7. | import pickle
8. | import cv2
9. | import os
10. |
11. | # construct the argument parse and parse the arguments
12. | ap = argparse.ArgumentParser()
13. | ap.add_argument("-m", "--model", required=True,
14. |     help="path to trained model model")
15. | ap.add_argument("-l", "--labelbin", required=True,
16. |     help="path to label binarizer")
17. | ap.add_argument("-i", "--image", required=True,
18. |     help="path to input image")
19. | args = vars(ap.parse_args())
```

On **Lines 2-9** we `import` the necessary packages for this script. Notably, we'll be using Keras and OpenCV in this script.

Then we proceed to parse our three required command line arguments on **Lines 12-19**.

From there, we load and preprocess the input image:

→ **Launch Jupyter Notebook on Google Colab**

Click here to download the source code to this post

Multi-label classification with Keras

```
# load the image
image = cv2.imread(args["image"])
output = imutils.resize(image, width=400)

# pre-process the image for classification
image = cv2.resize(image, (96, 96))
image = image.astype("float") / 255.0
image = img_to_array(image)
image = np.expand_dims(image, axis=0)
```

We take care to preprocess the image in the *same manner* as we preprocessed our training data.

Next, let's load the model + multi-label binarizer and classify the image:

→ **Launch Jupyter Notebook on Google Colab**

Multi-label classification with Keras

```
31. # load the trained convolutional neural network and the multi-label
32. # binarizer
33. print("[INFO] loading network...")
34. model = load_model(args["model"])
35. mlb = pickle.loads(open(args["labelbin"], "rb").read())
36.
37. # classify the input image then find the indexes of the two class
38. # labels with the *largest* probability
39. print("[INFO] classifying image...")
40. proba = model.predict(image)[0]
41. idxs = np.argsort(proba)[::-1][:2]
```

We load the `model` and multi-label binarizer from disk into memory on **Lines 34 and 35**.

From there we classify the (preprocessed) input `image` (**Line 40**) and extract the top two class labels indices (**Line 41**) by:

- Sorting the array indexes by their associated probability in descending order

- Grabbing the first two class label indices which are thus the top-2 predictions from our network

You can modify this code to return more class labels if you wish. I would also suggest

You can modify this code to return more class labels if you wish. I would also suggest thresholding the probabilities and only returning labels with > *N%* confidence.

Click here to download the source code to this post

From there, we'll prepare the class labels + associated confidence values for overlay on the output image:

→ **Launch Jupyter Notebook on Google Colab**

Multi-label classification with Keras

```
43. # loop over the indexes of the high confidence class labels
44. for (i, j) in enumerate(idxs):
45.     # build the label and draw the label on the image
46.     label = "{}: {:.2f}%".format(mlb.classes_[j], proba[j] * 100)
47.     cv2.putText(output, label, (10, (i * 30) + 25),
48.         cv2.FONT_HERSHEY_SIMPLEX, 0.7, (0, 255, 0), 2)
49.
50. # show the probabilities for each of the individual labels
51. for (label, p) in zip(mlb.classes_, proba):
52.     print("{}: {:.2f}%".format(label, p * 100))
53.
54. # show the output image
55. cv2.imshow("Output", output)
56. cv2.waitKey(0)
```

The loop on **Lines 44-48** draws the top two multi-label predictions and corresponding confidence values on the `output` image.

Similarly, the loop on **Lines 51 and 52** prints the all the predictions in the terminal. This is useful for debugging purposes.

Finally, we show the `output` image on the screen (**Lines 55 and 56**).

Keras multi-label classification results

Let's put `classify.py` to work using command line arguments. You do not need to modify the code discussed above in order to pass new images through the CNN. Simply use the **command line arguments (https://pyimagesearch.com/2018/03/12/python-argparse-command-line-arguments/)** in your terminal as is shown below.

Let's try an image of a red dress — notice the three command line arguments that are processed at runtime:

→ **Launch Jupyter Notebook on Google Colab**

Click here to download the source code to this post

Multi-label classification with Keras

```
$ python classify.py --model fashion.model --labelbin mlb.pickle \
   --image examples/example_01.jpg
Using TensorFlow backend.
[INFO] loading network...
[INFO] classifying image...
black: 0.00%
blue: 3.58%
dress: 95.14%
jeans: 0.00%
red: 100.00%
shirt: 64.02%
```

(https://pyimagesearch.com/wp-content/uploads/2018/04/keras_multi_label_output_01.png)

Figure 4: The image of a red dress has correctly been classified as *"red"* and *"dress"* by our Keras multi-label classification deep learning script.

Success! Notice how the two classes (*"red"* and *"dress"*) are marked with high confidence.

Now let's try a blue dress:

→ Launch Jupyter Notebook on Google Colab

Click here to download the source code to this post

Multi-label classification with Keras

```
$ python classify.py --model fashion.model --labelbin mlb.pickle \
   --image examples/example_02.jpg
Using TensorFlow backend.
[INFO] loading network...
[INFO] classifying image...
black: 0.03%
blue: 99.98%
dress: 98.50%
jeans: 0.23%
red: 0.00%
shirt: 0.74%
```

(https://pyimagesearch.com/wp-content/uploads/2018/04/keras_multi_label_output_02.png)

Figure 5: The *"blue"* and *"dress"* class labels are correctly applied in our second test of our Keras multi-label image classification project.

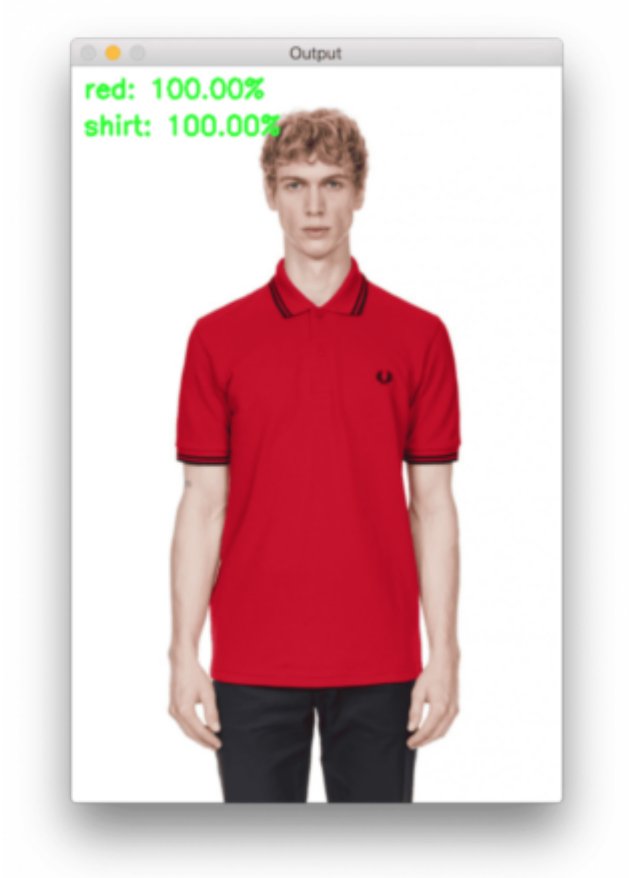
A blue dress was no contest for our classifier. We're off to a good start, so let's try an image of a red shirt:

→ Launch Jupyter Notebook on Google Colab

Click here to download the source code to this post

Multi-label classification with Keras

```
$ python classify.py --model fashion.model --labelbin mlb.pickle \
   --image examples/example_03.jpg
Using TensorFlow backend.
[INFO] loading network...
[INFO] classifying image...
black: 0.00%
blue: 0.69%
dress: 0.00%
jeans: 0.00%
red: 100.00%
shirt: 100.00%
```



(https://pyimagesearch.com/wp-content/uploads/2018/04/keras_multi_label_output_03.png)

Figure 6: With 100% confidence, our deep learning multi-label classification script has correctly classified this red shirt.

The red shirt result is promising.

How about a blue shirt?

→ Launch Jupyter Notebook on Google Colab

Click here to download the source code to this post

Multi-label classification with Keras

```
$ python classify.py --model fashion.model --labelbin mlb.pickle \
   --image examples/example_04.jpg
Using TensorFlow backend.
[INFO] loading network...
[INFO] classifying image...
black: 0.00%
blue: 99.99%
dress: 22.59%
jeans: 0.08%
red: 0.00%
shirt: 82.82%
```

(https://pyimagesearch.com/wp-content/uploads/2018/04/keras_multi_label_output_04.png)

Figure 7: Deep learning + multi-label + Keras classification of a blue shirt is correctly calculated.

blue shirt is correctly calculated.

Our model is very confident that it sees blue, but slightly less confident that it has encountered a shirt. That being said, this is still a correct multi-label classification!

Click here to download the source code to this post

Let's see if we can fool our multi-label classifier with blue jeans:

→ **Launch Jupyter Notebook on Google Colab**

Multi-label classification with Keras

```
$ python classify.py --model fashion.model --labelbin mlb.pickle \
   --image examples/example_05.jpg
Using TensorFlow backend.
[INFO] loading network...
[INFO] classifying image...
black: 0.00%
blue: 100.00%
dress: 0.01%
jeans: 99.99%
red: 0.00%
shirt: 0.00%
```

(https://pyimagesearch.com/wp-content/uploads/2018/04/keras_multi_label_output_05.png)

Figure 8: This deep learning multi-label classification result proves that blue jeans can be correctly classified as both *"blue"* and *"jeans"*.

[Click here to download the source code to this post]

Let's try black jeans:

→ **Launch Jupyter Notebook on Google Colab**

Multi-label classification with Keras

```
$ python classify.py --model fashion.model --labelbin mlb.pickle \
   --image examples/example_06.jpg
Using TensorFlow backend.
[INFO] loading network...
[INFO] classifying image...
black: 100.00%
blue: 0.00%
dress: 0.01%
jeans: 100.00%
red: 0.00%
shirt: 0.00%
```

(https://pyimagesearch.com/wp-content/uploads/2018/04/keras_multi_label_output_06.png)

Figure 9: Both labels, *"jeans"* and *"black"* are correct in this Keras multi-label classification deep learning

experiment.

Click here to download the source code to this post

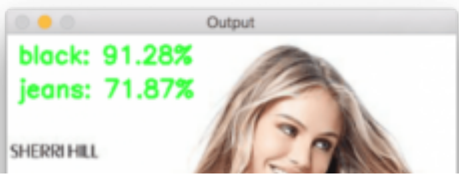
I can't be 100% sure that these are denim jeans (they look more like leggings/jeggings to me), but our multi-label classifier is!

Let's try a final example of a black dress (`example_07.jpg`). While our network has learned to predict *"black jeans"* and *"blue jeans"* along with both *"blue dress"* and *"red dress"*, can it be used to classify a *"black dress"*?

→ **Launch Jupyter Notebook on Google Colab**

Multi-label classification with Keras

```
$ python classify.py --model fashion.model --labelbin mlb.pickle \
    --image examples/example_07.jpg
Using TensorFlow backend.
[INFO] loading network...
[INFO] classifying image...
black: 91.28%
blue: 7.70%
dress: 5.48%
jeans: 71.87%
red: 0.00%
shirt: 5.92%
```



[Click here to download the source code to this post]

[(https://pyimagesearch.com/wp-content/uploads/2018/04/keras_multi_label_output_07.png)]

Figure 10: What happened here? Our multi-class labels are incorrect. Color is marked as *"black"* but the classifier had a higher confidence that this was an image of *"jeans"* than a *"dress"*. The reason is that our neural network never saw this combination in its training data. See the *"Summary"* below for further explanation.

Oh no — a blunder! Our classifier is reporting that the model is wearing black jeans when she is actually wearing a black dress.

What happened here?

Why are our multi-class predictions incorrect? To find out why, review the summary below.

What's next? We recommend PyImageSearch University

Summary

In today's blog post you learned how to perform multi-label classification with Keras.

Performing multi-label classification with Keras is straightforward and includes two primary steps:

1. Replace the *softmax activation* at the end of your network with a *sigmoid activation*
2. Swap out *categorical cross-entropy* for *binary cross-entropy* for your loss function

From there you can train your network as you normally would.

The end result of applying the process above is a multi-class classifier.

You can use your Keras multi-class classifier to predict *multiple labels* with just a *single* forward pass.

However, there is a difficulty you need to consider:

You need training data for *each combination* of categories you would like to predict.

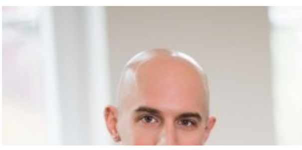
Just like a neural network cannot predict classes it was never trained on, your neural network cannot predict multiple class labels for combinations it has never seen. The reason for this behavior is due to activations of neurons inside the network.

If your network is trained on examples of both (1) black pants and (2) red shirts and now you want to predict *"red pants"* (where there are no *"red pants"* images in your dataset), the neurons responsible for detecting *"red"* and *"pants"* will fire, but since the network has never seen this combination of data/activations before once they reach the fully-connected layers, your output predictions will very likely be incorrect (i.e., you may encounter *"red"* or *"pants"* but very unlikely both).

Again, **your network cannot correctly make predictions on data it was never trained on** (and you shouldn't expect it to either). Keep this caveat in mind when training your own Keras networks for multi-label classification.

I hope you enjoyed this post!

To be notified when future posts are published here on PyImageSearch, *just enter your email address in the form below!*



About the Author

Hi there, I'm Adrian Rosebrock, PhD. All too often I see