Measure Feature Map Similarity

This notebook is an enhanced version of a notebook in the Keras examples: [Simple MNIST convnet]

Convolutional Neural Network (CNN) architectures use *feature maps* to capture aspects of an image.

Since the set of feature maps is the complete inventory of features of an image found by a CNN, a well-trained model should not have redundant feature maps- the feature maps should all be different. This notebook introduces a measurement of similarity across feature maps with the aim of avoiding redundant feature maps.

We will train a simple CNN against the standard MNIST stroke-digit dataset and will demonstrate how the mean similarity of feature maps slowly drops during training. We will also display feature map activations against an original MNIST image to illuminate how feature map similarity is a good measurement of the quality of a CNN model.

```
import random
import numpy as np
from tensorflow import keras
from tensorflow.keras import layers
```

```
# Model / data parameters
num_classes = 10
input_shape = (28, 28, 1)

# the data, split between train and test sets
(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()

# Scale images to the [0, 1] range
x_train = x_train.astype("float32") / 255
x_test = x_test.astype("float32") / 255
# Make sure images have shape (28, 28, 1)
x_train = np.expand_dims(x_train, -1)
x_test = np.expand_dims(x_test, -1)
# convert class vectors to binary class matrices
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)
```

Build the model

The model in the original notebook is broken out into two models:

1. a sub-model which emits the output of the CNN

2. a parent model for training purposes

This allows us to extract feature maps and measure similarity in a callback function.

Remember, a Model is also a Layer.

```python
cnn_model = keras.Sequential(
    [
        layers.InputLayer(input_shape=input_shape),
        layers.Conv2D(32, kernel_size=(3, 3), activation="relu"),
        layers.MaxPooling2D(pool_size=(2, 2)),
        layers.Conv2D(64, kernel_size=(3, 3), activation="relu"),
    ]
    , name='CNN_sub_model'
)
cnn_model.summary()

model = keras.Sequential(
    [
        layers.InputLayer(input_shape=input_shape),
        cnn_model,
        layers.MaxPooling2D(pool_size=(2, 2)),
        layers.Flatten(),
        layers.Dropout(0.5),
        layers.Dense(num_classes, activation="softmax"),
    ],
    name='Parent_model'
)

model.summary()
```

```
Model: "CNN_sub_model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 26, 26, 32)        320
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 13, 13, 32)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 11, 11, 64)        18496
=================================================================
Total params: 18,816
Trainable params: 18,816
Non-trainable params: 0
_________________________________________________________________
Model: "Parent_model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
CNN_sub_model (Sequential)   (None, 11, 11, 64)        18816
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 5, 5, 64)          0
_________________________________________________________________
flatten (Flatten)            (None, 1600)              0
```

```
_________________________________________________________________
dropout (Dropout)            (None, 1600)              0
_________________________________________________________________
dense (Dense)                (None, 10)                16010
=================================================================
Total params: 34,826
Trainable params: 34,826
Non-trainable params: 0
_________________________________________________________________
```

Log image similarities during training

Add a Callback that fetches the final set of feature maps generated by the model, and calculates the average similarity of a random subset of pairs of feature maps.

There are various ways to calculate similarity. This multiplies the two feature maps together and counts the resulting "high" valued cells.

```
img_array = x_test[0:1]

def similarity_multiply(img1, img2):
    # norm both to 0->1, multiply to produce 0->1
    min1 = np.min(img1)
    min2 = np.min(img2)
    base1 = np.max(img1) - min1
    base2 = np.max(img2) - min2
    if base1 == 0:
        base1 = 0.0001
    if base2 == 0:
        base2 = 0.0001
    norm1 = (img1 - min1) / base1
    norm2 = (img2 - min2) / base2
    mult = norm1 * norm2
    correlated = mult > np.mean(mult)

    percentage = sum(correlated.flatten()) / len(img1.flatten())
    return percentage

# While training, capture and log the mean similarity of the feature map pairs.
# This network only has 64 fmaps, so it's ok to just check every pair.
# This is using when training the complete network, but calls predict()
# on the sub-network to fetch the feature maps.

class LogSimilarities(keras.callbacks.Callback):
    def __init__(self, cnn_model, img_array, simfunc):
        super(LogSimilarities, self).__init__()
        self._cnn_model = cnn_model
        self._img_array = img_array
        self._simfunc = simfunc
```

```
        self._simfunc = simfunc

    def on_epoch_end(self, epoch, logs=None):
        maps = self._cnn_model.predict(self._img_array)[:, :, :, :]
        preds = []
        for i in range(maps.shape[3]):
            preds.append(maps[0, :, :, i])
        sims = []
        for i in range(maps.shape[3]):
            for j in range(i + 1, maps.shape[3]):
                measure = self._simfunc(preds[i], preds[j])
                sims.append(measure)
        avg = sum(sims)/len(sims)
        if logs:
            if 'similarity' not in logs:
                logs['similarity'] = []
            logs['similarity'].append(avg)
        else:
            print('Epoch: ' + epoch + ', mean similarity: ' + avg)

```

Train the model

```
batch_size = 512
epochs = 30

simfunc = similarity_multiply
logsim = LogSimilarities(cnn_model, img_array, simfunc=simfunc)

model.compile(loss="categorical_crossentropy", optimizer="adam", metrics=["accurac

history = model.fit(x_train, y_train, batch_size=batch_size, epochs=epochs, valida
          callbacks=[logsim])
```

```
Epoch 1/30
106/106 [==============================] - 3s 8ms/step - loss: 0.7420 - accura
Epoch 2/30
106/106 [==============================] - 1s 5ms/step - loss: 0.1904 - accura
Epoch 3/30
106/106 [==============================] - 1s 5ms/step - loss: 0.1375 - accura
Epoch 4/30
106/106 [==============================] - 1s 6ms/step - loss: 0.1104 - accura
Epoch 5/30
106/106 [==============================] - 1s 6ms/step - loss: 0.0943 - accura
Epoch 6/30
106/106 [==============================] - 1s 6ms/step - loss: 0.0852 - accura
Epoch 7/30
106/106 [==============================] - 1s 6ms/step - loss: 0.0741 - accura
Epoch 8/30
106/106 [==============================] - 1s 6ms/step - loss: 0.0701 - accura
Epoch 9/30
106/106 [==============================] - 1s 6ms/step - loss: 0.0641 - accura
```

```
Epoch 10/30
106/106 [==============================] - 1s 6ms/step - loss: 0.0610 - accura
Epoch 11/30
106/106 [==============================] - 1s 5ms/step - loss: 0.0574 - accura
Epoch 12/30
106/106 [==============================] - 1s 6ms/step - loss: 0.0551 - accura
Epoch 13/30
106/106 [==============================] - 1s 6ms/step - loss: 0.0523 - accura
Epoch 14/30
106/106 [==============================] - 1s 5ms/step - loss: 0.0496 - accura
Epoch 15/30
106/106 [==============================] - 1s 6ms/step - loss: 0.0486 - accura
Epoch 16/30
106/106 [==============================] - 1s 6ms/step - loss: 0.0470 - accura
Epoch 17/30
106/106 [==============================] - 1s 6ms/step - loss: 0.0438 - accura
Epoch 18/30
106/106 [==============================] - 1s 6ms/step - loss: 0.0432 - accura
Epoch 19/30
106/106 [==============================] - 1s 6ms/step - loss: 0.0416 - accura
Epoch 20/30
106/106 [==============================] - 1s 6ms/step - loss: 0.0398 - accura
Epoch 21/30
106/106 [==============================] - 1s 6ms/step - loss: 0.0397 - accura
Epoch 22/30
106/106 [==============================] - 1s 5ms/step - loss: 0.0388 - accura
Epoch 23/30
106/106 [==============================] - 1s 6ms/step - loss: 0.0381 - accura
Epoch 24/30
106/106 [==============================] - 1s 6ms/step - loss: 0.0368 - accura
Epoch 25/30
106/106 [==============================] - 1s 6ms/step - loss: 0.0345 - accura
Epoch 26/30
106/106 [==============================] - 1s 6ms/step - loss: 0.0333 - accura
Epoch 27/30
106/106 [==============================] - 1s 6ms/step - loss: 0.0333 - accura
Epoch 28/30
106/106 [==============================] - 1s 6ms/step - loss: 0.0322 - accura
Epoch 29/30
106/106 [==============================] - 1s 6ms/step - loss: 0.0321 - accura
Epoch 30/30
```

Analyze Similarity of Feature Maps

```
!pip install --force-reinstall -qq git+https://github.com/LanceNorskog/keract.git
import keract
from sklearn.preprocessing import MinMaxScaler

```

```
  Building wheel for keract (setup.py) ... done
ERROR: tensorflow 2.5.0 has requirement numpy~=1.19.2, but you'll have numpy 1.2
ERROR: datascience 0.10.6 has requirement folium==0.2.1, but you'll have folium 
ERROR: albumentations 0.1.12 has requirement imgaug<0.2.7,>=0.2.5, but you'll ha
```

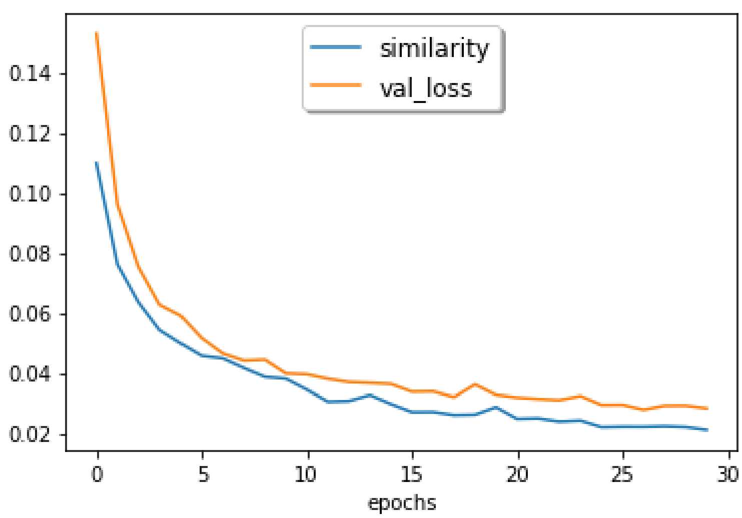
Plot mean similarity over epochs

Let's plot the **similarity** value gathered during training. This is the mean similarity between all pairs of the 64 feature maps generated by the CNN network.

```
import matplotlib.pyplot as plt

def plot_similarity_stats(history):
    fig, ax = plt.subplots(nrows=1, ncols=1)
    ax.plot(history['similarity'], label='similarity')
    ax.plot(history['val_loss'], label='val_loss')
    legend = ax.legend(loc='upper center', shadow=True, fontsize='large')
    ax.set(xlabel='epochs', title='')

plot_similarity_stats(history.history)
```



This chart demonstrates how the drop in similarity tracks the improvement of the CNN (val_loss). CNN feature maps will slowly become decorrelated during a stable training cycle. Also notice how the similarity continues to drop as the network overtrains (val_loss starts increasing).

Visualize the Feature Maps

```
def plot_heatmaps(img_array, fmap_i, fmap_j, similarity=None):
    sim_label = ''
    if similarity:
        sim_label = "{:.2f}".format(similarity)
    feature_maps = np.zeros((1, fmap_i.shape[0], fmap_i.shape[1], 3), dtype='float
    feature_maps[0, :, :, 0] = fmap_i
```

```
    feature_maps[0, :, :, 1] = fmap_j
    feature_maps[0, :, :, 2] = fmap_i[:,:] * fmap_j[:,:]
    activations = {sim_label: feature_maps}
    fig, axes = plt.subplots(1, 3, figsize=(12, 12))
    keract.display_heatmaps_1(activations, img_array, in_fig=fig, in_axes=axes)
```

Calculate and display similarity over all pairs of feature maps. Keep the image pair with the maximum and minimum similarity.

Phillipe Remy's "Keract" library provides a very handy toolkit for fetching all of the feature maps generate for an image. It also will adorn the original image with data from a feature map to create a "heatmap", which superimposes the feature map onto the original image used to make the prediction.

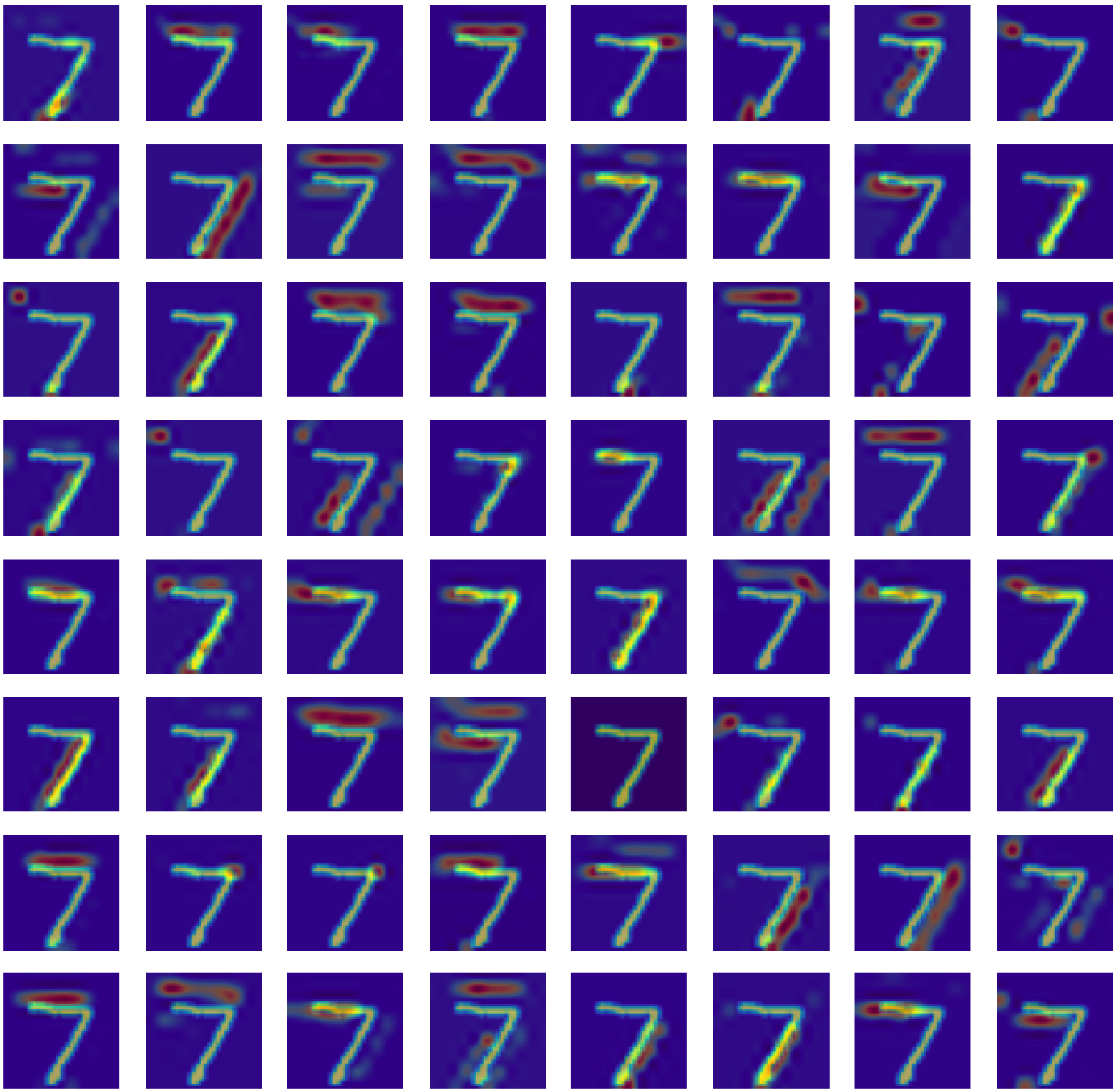
```
maps = cnn_model.predict(img_array)[:, :, :, :]
preds = []
for i in range(maps.shape[3]):
    preds.append(maps[0, :, :, i])
preds = np.asarray(preds)
scaler = MinMaxScaler()
scaler.fit(preds.reshape(-1, 1))


sims = []
x = 0
min_i = -1
min_j = -1
max_i = -1
max_j = -1
min_sim = 100000
max_sim = -1

for i in range(len(preds)):
    for j in range(i + 1, len(preds)):
        measure = simfunc(preds[i], preds[j])
        top_i = np.max(preds[i])
        top_j = np.max(preds[j])
        ratio = np.max([top_i, top_j])/np.min([top_i, top_j])
        if measure < min_sim and ratio < 3:
            min_sim = measure
            min_i = i
            min_j = j
        if measure > max_sim:
            max_sim = measure
            max_i = i
            max_j = j
        sims.append(measure)
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:24: RuntimeWarning:
```

```
activations = {'': maps}
fig, axes = plt.subplots(8, 8, figsize=(12, 12))
axes[3][2].grid(color='r', linestyle='-', linewidth=2)
keract.display_heatmaps_1(activations, img_array, in_fig=fig, in_axes=axes)
```



These 64 heatmaps are "features" or "aspects" of what the CNN notices about the handwritten digit '7'. There are several different measurements of horizontal and diagonal strokes.

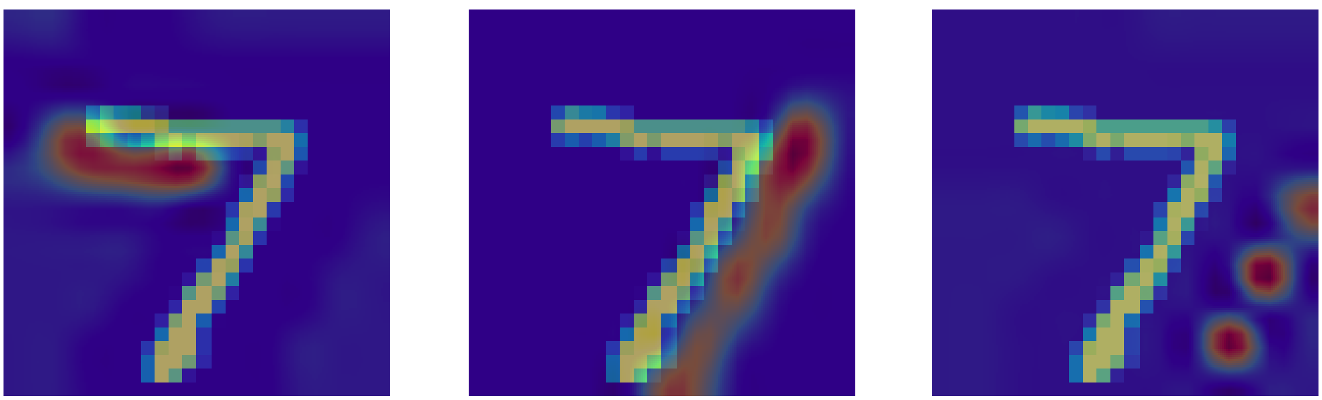
Note:

> These images are a great demonstration of the "translation invariance" property of Convolutional Neural Networks. As the image is processed by a stack of Conv2D layers, activations "slide across" the image. Different input images with the same features in different places in the image can activate the same feature map. This is why a feature map might "light up" next to the handwritten stroke rather than on it: the handwritten digits are all roughly the same size, but they are placed differently inside the image. The feature maps pick an "average" placement for a horizontal or diagonal stroke.

Similar Feature Maps

Next we will display the most similar pair of feature maps above, and then multiply the two feature maps together in Hadamard (cell-wise) mode to demonstrate their correlation. The left and middle images are the two feature maps, the rightmost image is the two feature maps multiplied together. This is the core idea of the similarity measure.

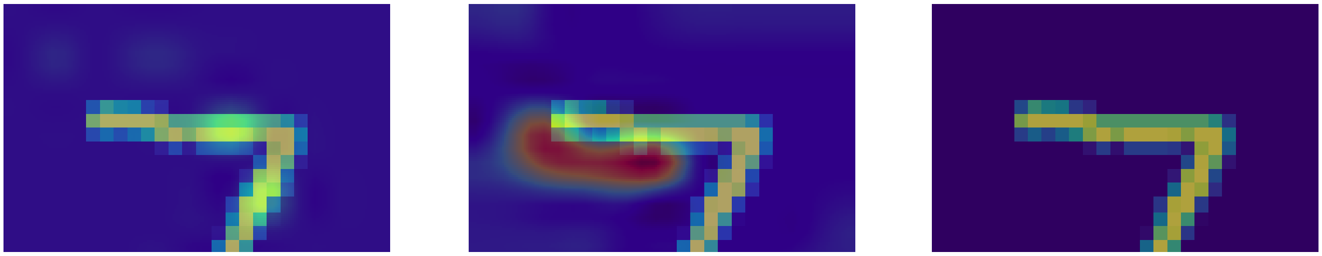
```
plot_heatmaps(img_array, preds[max_i], preds[max_j])
```



Here we do the same with the least similar feature maps. Since they have no common areas, there no activated areas on the right.

The rightmost image has a darker background because high and low activations are exaggerated by multiplying.

```
plot_heatmaps(img_array, preds[min_i], preds[min_j])
```



Conclusion

It is clear from this demonstration that feature map similarity is a useful measurement of the quality of a convolutional neural network: as the network improves, the mean similarity will drop.

The reason for this is simple: good feature maps are decorrelated. Feature maps are *independent captures* of features (parts of images) that happen over and over in the input images. If multiple feature maps describe the same feature, then processing power is being wasted. The **descriptive bandwidth** of the feature maps is optimized when no two feature maps describe the same feature.

Based on this insight, it should be possible to improve a CNN by measuring feature map similarity and providing feedback via the loss function. This is the core idea behind Wedge Dropout.

```
1
```