Predicting Image Similarity using Siamese Networks

In my previous post, I mentioned that I want to use Siamese Networks to predict image similarity from the INRIA Holidays Dataset. The Keras project on Github has an example Siamese network that can recognize MNIST handwritten digits that represent the same number as similar and different numbers as different. This got me all excited and eager to try this out on the Holidays dataset, which contains 1491 photos from 500 different vacations.

My Siamese network is somewhat loosely based on the architecture in the Keras example. The main idea behind a Siamese network is that it takes two inputs which need to be compared to each other, so we reduce it to a denser and hopefully more "semantic" vector representation and compare it using some standard vector arithmetic. Each input undergoes a dimensionality reduction transformation implemented as a neural network. Since we want the two images to be transformed in the same way, we train the two networks using shared weights. The output of the dimensionality reduction is a pair of vectors, which are compared in some way to yield a metric that can be used to predict similarity between the inputs.

The Siamese network I built is shown in the diagram below. It differs from the Keras example in two major ways. First, the Keras example uses Fully Connected Networks (FCNs) as the dimensionality reduction transformation component, whereas I use a Convolutional Neural Network (CNN). Second, the example computes the Euclidean distance between the two output vectors, and attempts to minimize the contrastive loss between them to produce a number in the [0,1] range that is thresholded to return a binary similar/dissimilar prediction. In my case, I use a FCN that combines the output vectors using element-wise dot product, use cross-entropy as my loss function, and predict a 0/1 to indicate similar/dissimilar.

![Siamese Network Diagram](image)

For the CNN, I tried various different configurations. Unfortunately, I started running out of memory on the g2.2xlarge instance when I started trying large CNNs, and ended up migrating to a p2.xlarge. Even then, I had to either cut down the size of the input image or the network complexity, and eventually settled on a LeNet configuration for my CNN, which seemed a bit underpowered for the data. For the current configuration, shown in 02-holidays-siamese-network notebook, the network pretty much refused to learn anything. In other tries, the best test set accuracy I was able to get was about 60%, but all of them involved compromising on the input size or the complexity of the CNN, so I gave up and started looking at other approaches.

I have had success with transfer learning in the past, where you take large networks pre-trained on some external corpus such as ImageNet, chop off the classification head, and expose the vector from the layer prior to the head layer(s). So the pre-trained network acts as the vectorizer or dimension reducer component. I used the following pre-trained networks that are available in Keras applications, to generate vectors from. The code to do this can be found in the 03-pretrained-nets-vectorizers notebook.

- VGG-16
- VGG-19
- ResNet
- InceptionV3
- xCeption
The diagram above shows the general setup of this approach. The first step is to just run the predict method on the pre-trained models to generate the vectors for each image. These vectors then need to be combined and fed to another classifier component. Some strategies I tried were element-wise dot product, absolute difference and squared (Euclidean) distance. In case of dot product, corresponding elements of the two vectors that are both high end up becoming higher, and elements that differ end up getting smaller. In case of absolute and squared differences, elements that are different tend to become larger. In case of squared difference, large differences are highlighted better than small differences.

The classifier component (shown as FCN in my previous diagram) can be any kind of classifier, including non neural network based ones. As a baseline, I tried several common classifiers from the Scikit-Learn and XGBoost packages. You can see the code in the 04-pretrained-vec-dot-classifier, 05-pretrained-vec-l1-classifier, and 06-pretrained-vec-l2-classifier notebooks. The resulting accuracies for each (vectorizer, merge strategy, classifier) combination on the held out test set are summarized below.

Generally speaking, XGBoost seems to do the best across all merge strategies and vectorization schemes. Among these, Inception and ResNet vectors seem to be the best overall. We also now have a pretty high baseline for accuracy, about 96.5% for Inception vectors merged using dot product and classified with XGBoost. The code for this can be found in the 07-pretrained-vec-nn-classifier notebook. The figure below shows the accuracies for different merge strategies for ResNet and Inception.

The next step was to see if I could get even better performance by replacing the classifier head with a neural network. I
ended up using a simple 3 layer FCN that gave a 95.7% accuracy with Inception vectors and using dot product for a merge strategy. Not quite as good as the XGBoost classifier, but quite close.

Finally, I decided to merge the two approaches. For the vectorization, I chose a pre-trained Inception network with its classification head removed. Input to this network would be images, and I would use the Keras ImageDataGenerator to augment my dataset, using the mechanism I described in my previous post. I decided to keep all the pre-trained weights fixed. For the classification head, I decided to start with the FCN I trained in the previous step and fine tune its weights during training. The code for that is in the 08‑holidays‑siamese‑finetune notebook.

![Diagram of the approach](image)

Unfortunately, this did not give me the stellar results I was hoping for, my best result was about 88% accuracy in similarity prediction. In retrospect, it may make sense to experiment with a simpler pre-trained model such as VGG and fine tune some of the later layer weights instead of keeping them all frozen. There is also a possibility that my final network is not getting the benefits of a fine tuned model from the previous steps. One symptom is that the accuracy after the first epoch is only around 0.6 - I would have expected it to be higher with a well trained model. In another project where a similar thing happened, a colleague discovered that I was doing extra normalization with ImageDataGenerator that I hadn't been doing with the vectorization step - this doesn't seem to be the case here though.

Overall, I got the best results from the transfer learning approach, with Inception vectors, dot product merge strategy and XGBoost classifier. Nice thing about transfer learning is that it is relatively cheap in terms of resources compared to the fine tuning or even the from-scratch training approach. While XGBoost does take some time to train, you can do the whole thing on your laptop. This is also true if you replace the XGBoost classifier with an FCN. You can also do inline Image Augmentation (i.e, without augmenting and saving) using the Keras ImageDataGenerator if you use the random_transform call.

*Edit 2017‑08‑09:* seovchinnikov on Github has run some further experiments on his own datasets, where he has achieved 98% accuracy using feature fusion (code). See here for the full discussion.

Posted by Sujit Pal at 8:20 AM

Labels: deep‑learning, image‑processing, similarity