Computer Vision 2: Exercise 5 (28.5.2019)

CNNs with image outputs

We will learn about

- how to predict images as outputs from a CNN
- using dropout regularization
- how to save prediction outputs to disk

A CNN for eye fixation prediction

In problems such as eye fixation prediction the output of a CNN should be an image rather than a single class label. Other problems where an image output is needed include semantic segmentation and instance segmentation. Figure 1 shows the general architecture of the network we will implement[1]. Briefly, the main idea of the network structure is that it concatenates features from multiple layers together to predict a saliency map, instead of only using low-level or high-level features.

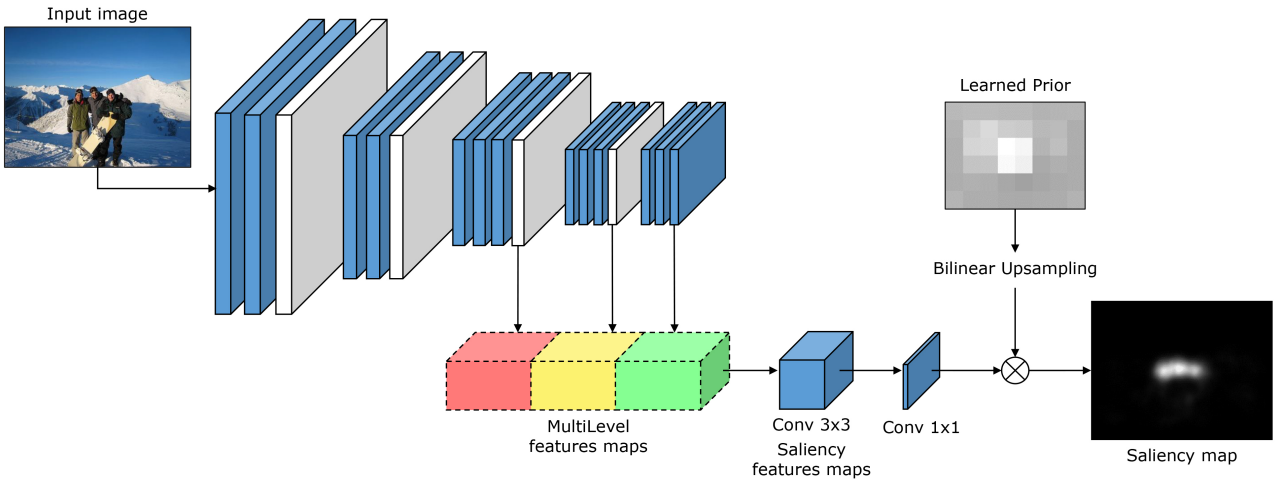

Figure 1: A multi-level network for saliency prediction. Blue layers denote convolutional layers. Gray layers are pooling layers.

Getting started: reading data Download the starting script `ex05_start.py` from Moodle. This file contains some basics to get you started with this exercise:

- reading of training data from a directory (modify to fit your path structure),
- implementation of the loss function, discussed in more detail later, and
- a skeleton of a training loop.

You will find some commented lines in the middle of the file where you can add your own work.

[1] You can refer to the paper: Cornia, M., Baraldi, L., Serra, G., and Cucchiara, R.: "A Deep Multi-Level Network for Saliency Prediction", In Proceedings of the 23rd International Conference on Pattern Recognition, 2016. The paper is available at `https://arxiv.org/pdf/1609.01064.pdf` for further information.

Base network. We set up the base network architecture shown on the left hand side of Figure 1, next to the input image.

1. Define convolution and max pooling layers to recreate the base network. Use the configuration based on VGG16-D shown in Table 1. You can either initialize the weights randomly, or use transfer learning. **If you use transfer learning, initialize the biases to zero.**

   Note that the configuration used here is different from the one reported in the paper. This is because using the full configuration can require very large amounts of GPU memory to run. If you still run out of memory, try to reduce your batch size or remove the `conv_4_1` block and use only two feature maps in the following tasks.

Table 1: Modified VGG-16D structure for use in the base network. Changes to original VGG-16D structure are highlighted in red. All layers have ReLU activation, and use padding: same.

Block	Layer	Parameters
conv1	conv3-64	3x3 kernel, 64 filters
	conv3-64	3x3 kernel, 64 filters
pool1	maxpool	2 x 2, stride 2
conv2	conv3-128	3x3 kernel, 128 filters
	conv3-128	3x3 kernel, 128 filters
pool2	maxpool	2 x 2, stride 2
conv3	conv3-256	3x3 kernel, 256 filters
	conv3-256	3x3 kernel, 256 filters
	conv3-256	3x3 kernel, 256 filters
pool3	maxpool	2 x 2, stride 1
conv4	conv3-512	3x3 kernel, 512 filters

Multilevel feature maps and saliency map output. Next, we implement a multilevel feature map shown at the bottom of Figure 1. This feature map should concatenates together the feature maps output by layers `pool3`, `pool4`, and `conv4_1`. This is possible since due to the set up of the network shown in Table 1, they have the same height H and width W.

Mathematically, we have three feature maps f_1, f_2 and f_3, which each have a size of $[B, H, W, s_i]$, where B is the batch size, and s_i is the number of layers in the ith feature map f_i. We concatenate these together, and obtain a concatenated feature map f which has a size $[B, H, W, s]$ where $s = \sum_{i=1}^{3} s_i$ – the original feature maps are concatenated along the last dimension.

1. Use `tf.concat` to concatenate the outpus of `pool3`, `pool4`, and `conv_4_1` along the last dimension to obtain the multilevel feature map.

2. Add a dropout regularizer after the multileve feature map concatenation. The dropout will be set to randomly sample which features will be set to zero. The idea is to modify the learning process so the network will not rely too much on a single feature. This is a type of regularization, making the training task more challenging with the hope of improving generalization to unseen data.

Dropout can be implemented by using `tf.layers.dropout`. Use a keep ratio of 0.5. Create a placeholder called `is_training` with a boolean data type and scalar shape, and use it in the dropout layer. We have to do this so we can control when dropout will be enabled: we want to enable it during training, but not during testing or validation!

Dropout can easily be enabled or disabled by providing a `True` or `False` value, respectively, for the placeholder when calling `session.run`.

3. Add a 3x3 convolution layer with 64 filters (padding: same, activation: ReLU) that processes the multilevel feature after dropout to get the "saliency feature maps" indicated in Figure 1.

4. Add a 1x1 convolution with 1 filter (activation: ReLU) which produces a raw saliency map that is on the left hand side of the $\otimes$ node in Figure 1.

The loss function The loss function is already implemented in the starting code[2]. The idea is to include a mean squared error term, but weight the pixels. This is done because a typical saliency map has many more dark (non-salient) pixels than salient pixels. Without weighting, the network would learn to predict an all zero saliency map, which would get a decent loss but completely ignore the task we want to solve! The weighting we apply will give higher emphasis to bright pixels (with target value close to 1).

Given an input image x, the raw predicted saliency map from the network is denoted as $\phi(x)$. Because of the ReLU activation, all values of $\phi(x)$ are known to be non-negative. This map is normalized to the range $[0, 1]$ by $\phi(x)/\max_i \phi(x)_i$. This is compared to the target saliency map y by calculating the pixelwise L_2-norm. The values for each pixel i are weighted by a term $\frac{1}{\alpha - y_i}$, where $\alpha > 1$ is a weighting constant. We use $\alpha = 1.01$. So a dark pixel ($y_i = 0$) will receive a weight of ≈ 0.99, and a bright pixel ($y_i = 1$) a weight of 100.

The loss for the pair (x, y) of input and target is

$$\frac{1}{N} \sum_{i=1}^{N} \frac{1}{\alpha - y_i} \left\| \frac{\phi(x)_i}{\max_i \phi(x)_i} - y_i \right\|_2 ,$$

where N is the number of pixels in the maps.

Training the network In the starting code, some code for generating batches is already available. Take some time to study it to understand how it works.

1. Add any summaries you are interested to see during training, at least the loss value.

2. Add the code to train the network by calling the minimization operation.

3. Add any other functionality you want: validation, saving of parameters, other summaries (e.g., images), ...

4. Start training and verify that it works: loss value starts to decrease, no NaN losses, etc.

You can work on the final task before training the network further. Note that it can take **a lot** of time/training batches for the network to converge. For example, after 10000 batches of size 32, there were still improvements in the loss value.

[2]Note the loss presented here differs from the one presented in the paper. There is an error in the paper, as confirmed by the authors: `https://github.com/marcellacornia/mlnet/issues/8`

Saving outputs to disk Because of the max pooling layers, the outputs predicted by the network are smaller than the original saliency maps. We can upsample them back to size 180 by 320, and then save them to disk.

1. After training is complete, add code that 1) loads the testing images from disk, and 2) feeds them through the network, either one at a time or in batches. You will not need to calculate the loss value, or run the training operation, but you must turn off dropout.

2. Add code that upsamples the images to the desired size and saves it to disk. Remember to make sure that you know which testing images you are working with so you can name the saved image appropriately. Example:

```
from skimage.transform import resize
my_saliency = sess.run(predicted_saliency, feed_dict={input_image
    : my_image, is_training: False})
upsampled_saliency = resize(my_saliency, output_shape=(180,320,1)
# save image to disk, e.g., using imageio.imwrite
```

Optional extra tasks

If you are done with all other tasks, there are a number of extra tasks you can try. They are listed below in no particular order, you can work on any one you want in any order you like!

- Read from the paper about learnable prior. Implement this by adding a `tf.Variable` into your own network. Remember to add the extra regularization term to the loss.

- Add a validation step to your training: read the validation data and periodically evaluate the loss on it and log it to a summary.

- Read up on image summaries and add them to your code so you can visualize the inputs, targets, and predictions during training: `https://www.tensorflow.org/api_docs/python/tf/summary/image`

- Add saving of the model parameters.