Cognitive Computer Vision: Exercise 5

CNN for eye fixation prediction: Training

In the previous exercise, we implemented a simple eye fixation prediction system based on the architecture shown in Figure 1. We now set up the training procedure for the network.

- Start from your implemented system from the last exercise, or the model solution for the last exercise provided in Moodle.
- Download the dataset from Moodle. You can find the `.zip` file under the Project link.

Reading images

Download the file `datareader.py` from Moodle. This file provides a simple class that will create a list of input images and fixation maps, and read them from the disk one batch at a time. This way, we do not have to read the entire dataset into memory at once.

Try out the following example to see how this class works. You must to replace the path to point to the location where you saved the training dataset.

```python
from datareader import DataReader
train_data = DataReader('/path/to/ccv/data/train')
batchsize = 10

num_training_images = train_data.num_images
num_batches = train_data.num_batches_of_size(batchsize)
print('Training data has ' + str(num_training_images) + ' images')
print('There are ' + str(num_batches) + ' batches of size ' + str(
    batchsize) + ' (of which one may be smaller)')

images, fixations, filenames = train_data.get_batch(batchsize)
```

Note that the class will produce batches indefinitely. Whenever it runs out of images to read, it will shuffle the list of available images and start over.

Training the network

Refer to the example solutions of earlier exercises in Moodle for hints how to complete the following steps.

- Create `DataReader` objects for both the training and validation datasets. Select a batch size to use.
- Create an optimizer, e.g., `tf.train.GradientDescentOptimizer` or `tf.train.AdamOptimizer` and set a learning rate for it. Set the optimizer to minimize the loss function.
- Write a training loop that runs for a fixed number of batches, and runs an validation step after a specific number of batches. Use the `num_images` property of `DataReader` and your batch size to determine how many batches you can run in the validation to use all data.

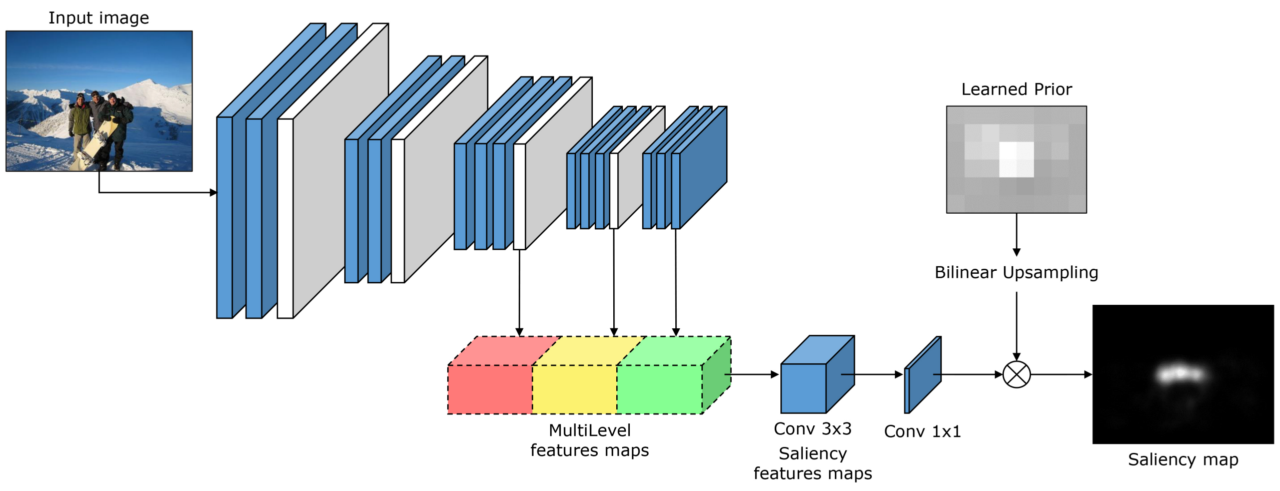

Figure 1: Features from many layers are concatenated into a multilevel feature map. Additional convolutional layers are inserted to predict a saliency map or fixation map. Blue layers indicate convolutional layers. Gray layers indicate pooling layers. Image from authors of [1].

- Set up TensorBoard monitoring for the loss, both training and validation.

Run your code to verify that your network starts training.

Saving checkpoints

When training a CNN, it is useful to be able to save the learned parameters from time to time. Good parameter sets can be recovered later, or training can be resumed after a break. In Tensorflow, we can do this by saving and restoring `tf.Variables`. Here is an example of saving all variables in a model.

```
# First, construct all variables.
# Now add ops to save and restore all the variables.
saver = tf.train.Saver()
with tf.Session() as sess:
  # Possibly some computation with the variables, or training
  # Save variables to disk.
  # Replace b by a variable that specifies the current global step, e.g., training batch number.
  save_path = saver.save(sess, "/my/path/my-model", global_step=b)
```

Here is an example of restoring all variables in a model.

```
# First, construct all variables as before.
# Now add ops to save and restore all the variables.
saver = tf.train.Saver()
with tf.Session() as sess:
  # Read variables from disk for model saved with global_step=50
  saver.restore(sess, "/my/path/my-model-50")
```

`Saver` only restores all variables **already defined in your model**. This means that to read the checkpoint, we must also **first construct the model with all the variables**. This is not a problem since we know how to construct the computational graph for our model.

- Modify your code so that it stores a checkpoint after each validation step.

By default, `Saver` will keep 5 of the most recent checkpoints stored on disk. If you want to keep more checkpoints, you can use the `max_to_keep` argument of the constructor.

Saving outputs

Recall that the `DataReader` also returns the names of the original files for each batch. We can use these names to save the predictions of our model with an appropriate name: for example, if the batch contains an image `1623.jpg`, we can save the prediction output as `1623_prediction.jpg`.

- During every validation step, save the predictions output by your model to disk. Follow these steps:

    1. In `sess.run`, add an operation to run `normalized_output` that will return the predicted output normalized to range $[0, 1]$.
    2. Normalize the output obtained to the range $[0, 255]$ and convert it to `uint8` type for saving. Here is an example of saving an image:

    ```
    import numpy as np
    from PIL import Image
    # Assumed: prediction is a (224, 224, 1) array of type np.
        float32
    p_arr = (prediction * 255.0).astype(np.uint8)
    p_image = Image.fromarray(p_arr[:,:,0])
    p_image.save(prediction_name)
    ```

    3. Save the outputs with appropriate names. Note that the array size is $\begin{bmatrix} B & H & W & 1 \end{bmatrix}$, where B is the batch size. You should loop over the first dimension to save each prediction in the batch separately. You can look at `DataReader.py` for hints.

- Try to compare the saved predictions to the ground truth images using some evaluation metric. For example, you can find the KLD metric implemented in the project instructions in Moodle.

Next steps

We have set up a basic system to predict eye fixations. Of course, a lot can be done to improve the performance of this basic system. If you finish the other tasks for this exercise, you can look into one or several of the following topics to improve your system:

- Add a center bias through a learned prior as shown in Figure 1. Read the relevant parts of [1] for more information, and implement the prior by adding a `tf.Variable`.
- Increase the depth of your CNN by adding more layers and/or more filters per layer.
- Study how to select the training hyperparameters: experiment to find which learning rate produces the best validation loss.
- If you observe a much lower training loss than validation loss, look into different methods for regularization: L2 loss on the parameters, using dropout, etc.
- The simple approach for reading and feeding data we use here is also not so fast. The process is much faster if the next batch of images are read already *while the GPU is processing the current batch*. If you are interested to try this in your project, you can learn more about `tf.Datasets`: `https://www.tensorflow.org/programmers_guide/datasets` and `https://www.tensorflow.org/performance/datasets_performance`.

References

[1] Marcella Cornia, Lorenzo Baraldi, Giuseppe Serra, and Rita Cucchiara. A Deep Multi-Level Network for Saliency Prediction. In *International Conference on Pattern Recognition (ICPR)*, 2016.