Homework 4 (due January 9, 2018) – More about ConvNets

I) Working with a pretrained network

Download the archive `https://cvml.ist.ac.at/courses/DLWT_W17/data/ResNet-L50.zip` (91MB)

It contains a pretrained ResNet with 50 layers:

- `ResNet-L50.meta` contains the graph definition
- `ResNet-L50.ckpt` contains the variable values

You can now load the graph, using

```
saver = tf.train.import_meta_graph("ResNet-L50.meta")
```

1) Make yourself familiar with the graph, e.g.

- find out how many nodes the graph contains? (use e.g. `tf.get_default_graph().get_operations()`)
- how many nodes in the graph are of the type `Placeholder` (e.g. for each node, check it's `type` attribute)
- what's the largest number of inputs that any node has (use e.g. the `inputs` attribute) and which node is that?

You can access specific graph nodes by their names, e.g.

```
pythonvar =  tf.Graph.get_tensor_by_name(tf.get_default_graph(),'var:0')
```

finds the node `var:0` in the graph and assigns it to the python variable `pythonvar`

Use this mechanism to obtain access to

- the *input* layer of the graph (named `images:0`)
- the *output* layer of the graph (name `prob:0`)
- the *logits* layer thats is the input to the softmax layer (find out its name from the inputs to `prob`)

2) load network weights

To fill the variables in the graph with their pretrained values, create a session `sess` and run

```
saver.restore(sess, "ResNet-L50.ckpt")
```

3) making predictions

- load the image `https://cvml.ist.ac.at/courses/DLWT_W17/data/zebra.jpg`
- make sure that the range of pixel values is [0,1]. If they are [0,255], divide all values by 255.
- evaluate the network output when feeding the image as input.
- the resulting probability vector should be very peaked at index `340`.

Download the file `https://cvml.ist.ac.at/courses/DLWT_W17/data/synset.py`. If contains a list `synset` that contains all class names.

- which class does the index `340` correspond to?

II) Adversarial examples

We will use the trained network from above to observe the phenomenon of *adversarial examples* (in simplified form): small perturbations to an image that are (almost) invisible to the human eye can have a strong effect on the classifier outputs.

4) gradient of outputs w.r.t. the image

Perform gradient descent to minimize the 340th *logit* value with respect to the input *image*. As the input is not a variable but a placeholder, the easiest is to write the procedure yourself:

- use `tf.gradients` to compute the gradient
- update the image by a multiple of the gradient in python

- make sure that the image remains within the [0,1] range, e.g. using `np.clip`

Use a learning rate of $\eta = 0.1$. How many iterations does it take for the predicted class to change? What class appears? What other classes appear if you keep running for a while?

Look at the image after its label first changed. Can you see a difference to the original? Compute the pixelwise difference and visualize it.

You could also take the derivative of the *prob* outputs instead of the *logits*. What happens if you do? Why?

5) forcing a class

Modify your code such that you can enforce any class to be predicted for the image, e.g. by performing gradient *ascent* on the logit value of the target class.

E.g., how many gradient steps does it take until the image is classified as class 859 ('toaster')?

Look at the results image. Can you see a difference to the original? Compute the pixelwise difference and visualize it.

III) Residual Networks

6) training data

Download the data file

`https://cvml.ist.ac.at/courses/DLWT_W17/data/hw4-train-data.npy`

(beware, 147MB!) and the labels file

`https://cvml.ist.ac.at/courses/DLWT_W17/data/hw4-train-labels.npy`

You can load these into python using numpy's `load` command. The training data is a 4-dimensional tensor of size 50000x32x32x3. The first dimension ranges over 50000 images in RGB format, each of which is of size 32x32x3 in unsigned 8-bit integer format. The class labels are integers between 0 and 99. Use random 90% of the data as training set and the rest as validation data.

Hint: you should convert the images to floating point format before feeding them into tensorflow (or use `tf.image.convert_image_dtype`). You might also want to subtract the per-channel mean value and rescaling the entries, as network work typically best with data in the range [-1,1] rather than [0,255].

7) residual blocks

Write a python routine that creates a *residual block* to the tensorflow graph.

- input: a tensor of size *[N,W,H,C]* and a scalar S
- output: a tensor of size *[N,W/S,H/S,C*S]*

Each *residual block* consists of

- a *functional path* that acts on the inputs
  - 1 ReLU layer
  - 1 convolution layer with $C * S$ filters of size $3x3$, applied with stride S and 'SAME' padding
  - 1 drop-out layer (see `tf.layers.dropout`) with drop-out probability 0.3
  - 1 ReLU layer
  - 1 convolution layer with $C * S$ filters of size $3x3$, applied with stride 1 and 'SAME' padding
- a *shortcut path* that also acts of the inputs
  - if S=1, this does nothing to the input
  - if S>1, this is a $1x1$ convolution with stride S and $C * S$ filters
- the overall output of the block is the sum of the two paths

Test your implementation by creating a placeholder of size $[None, 32, 32, 3]$ that is fed into networks consisting of variables number of blocks, each with S having a value of either 1 or 2.

Note: everything should fit together nicely, there should not be issues of incompatible dimensions.

8) building a ResNet

Build a ResNet with the following architecture:

- 1 convolution layer with 16 filters of size $3x3$, applied with stride 1 and 'SAME' padding
- 2 residual blocks with $S = 1$
- 1 residual block with $S = 2$
- 2 residual blocks with $S = 1$
- 1 residual block with $S = 2$
- 3 residual blocks with $S = 1$
- 1 average pooling layer (for each channel separately, all values are averaged)
- 1 fully connected layer with 100 channels
- softmax outputs

Train the network on the data from 6) using cross-entropy loss. Choose hyperparameters (optimizer, learning rate, batch size, ...) in a reasonable way.

After each epoch, output the value of the objective and the validation error.

Hint: if training is too slow, e.g. if you don't have a GPU, do the same but use only a subset of the data.

9) analysis

Build a network with the same architecture as above, but without the shortcuts paths (i.e there's only a sequence of the functional blocks).

Train in the same way as before and compare the behavior of the objective and the validation error. What do you observe?

10) (optional) learning rates

ResNet training can benefit a lot from changing the learning rate over the course of the training. Adjust your implementation to reduce the learning rate by a factor of 10 every T epochs (e.g. consult `https://www.tensorflow.org/api_docs/python/tf/train/exponential_decay`). Try to find a value of T that leads to fast convergence and high classification accuracy.

11) (optional) batch normalization

ResNets often use batch normalization layers each convolutional layer in the residual blocks. Adjust your implementation to include this (e.g. consult `https://www.tensorflow.org/api_docs/python/tf/layers/batch_normalization`) and study the effect on convergence speed and classification accuracy.

Hint: beware that batch normalization needs special training operations to make sure that the running averages are computed across batches,

11) (optional) more layers

The above ResNet architecture follows a 3-3-3 scheme, where the values indicate the number of residual blocks and each '-' means a change of the number of channels (triggered by an $S = 2$ instead of $S = 1$).

Try deeper architectures, e.g. 5-5-5, 10-10-10 or even 100-100-100 (if you have the compute power).

Hand-in requirements

1. upload your code to 4),5),7),8) with reasonable parameters to the IST *git* server
2. pick exactly **one** of your trained Residual Networks and use it to predict labels for the data available at:

`https://cvml.ist.ac.at/courses/DLWT_W17/data/hw4-test-data.npy`

Beware that networks that include dropout and/or batch normalization layers should behave differently between training and prediction time, and you will have to specify which phase you are in.

3. write the resulting class predictions to a file "hw4-test-labels.txt" (one number 0-99 per line in the same order as the test examples) and upload it.