Computer Vision 2: Exercise 6 (4.6.2019)

Regularization

We will learn about regularization techniques, such as weight decay, dropout, and data augmentation.

Model capacity, underfitting, and overfitting

Figure 1 illustrates the concepts of training and generalization error of a machine learning model as a function of the *model capacity.* The textbook "Deep learning" by Goodfellow, Bengio, and Courville[1] states:

> Informally, a models capacity is its ability to fit a wide variety of functions. Models with low capacity may struggle to fit the training set. Models with high capacity can overfit by memorizing properties of the training set that do not serve them well on the test set.

The capacity of a CNN for example is higher the more parameters it has. As seen from Figure 1, as the capacity increases, the error on the training data set can be reduced. But this does not always result in a low generalization error (error on unseen or testing data): the difference between the two errors is called the generalization gap. The optimal capacity is indicated by the red vertical bar: here, the generalization gap is minimized.

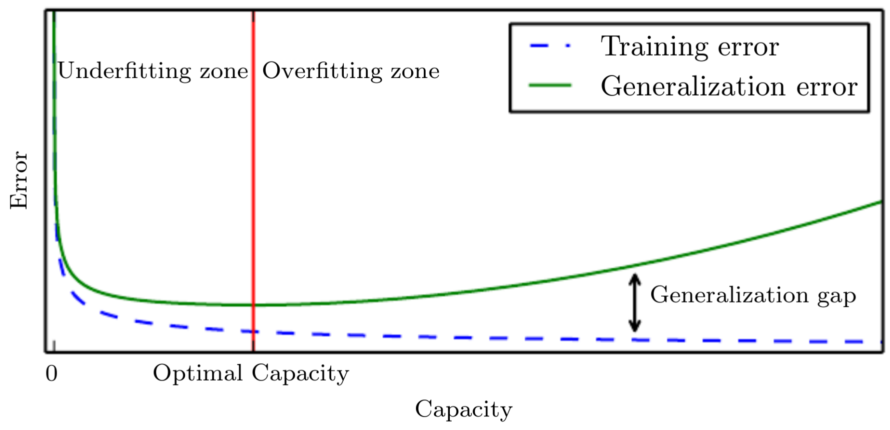

Figure 1: Training and generalization as a function of model capacity. Image from Goodfellow et al.

Goodfellow et al. define regularization as follows:

> Regularization is any modification we make to a learning algorithm that is intended to reduce its generalization error but not its training error.

Applying regularization, we are expecting two things to happen: 1) the training error will increase, and 2) the generalization gap will decrease. To see if 1) happens, we monitor the training loss. To monitor the generalization gap, we add a validation stage to our training process.

Typical regularization techniques

In the following subsections, we look at a few common regularization techniques. You can read the subsections in any order. For more in-depth information and more regularization techniques, read Chapter 7 "Regularization for deep learning" of the textbook "Deep learning".

[1] `http://www.deeplearningbook.org/`

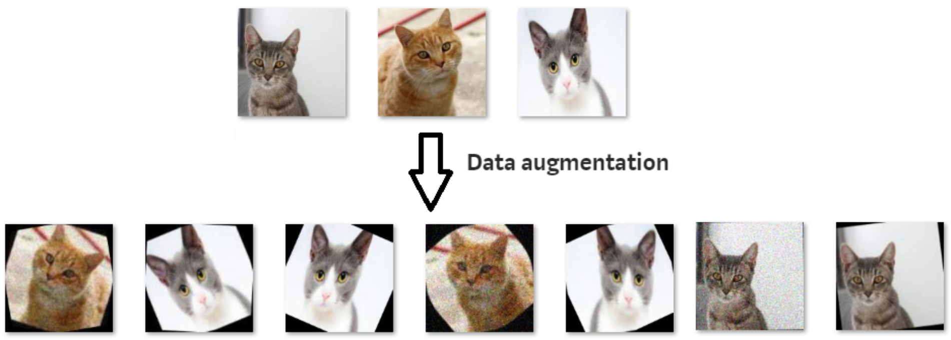

Figure 2: Data augmentation modifies an input x_i such that the target y_i is preserved, or is modified similarly as the input. Image by Thomas Himblot.

Dropout

We applied dropout in the previous exercise. A dropout layer will randomly set to zero some of the entries in its activation feature map, making them unavailable to the following network layers. This will decrease the reliance of the network on particular features (as they are not always available), making it more robust. Another way to interpret dropout is as model averaging[2].

In Tensorflow, dropout can be implemented by using a `tf.layers.dropout` layer. This layer takes as an input a feature map, and a boolean tensor, usually a placeholder, which controls whether dropout is enabled or not. The boolean placeholder should be assigned `True` during training to enable dropout, and `False` during validation/testing to disable dropout.

- Implement dropout in your network for at least one layer. It is often most useful to apply dropout near the output layer of the network, e.g., to disable some of the features that are used to predict a final output (saliency map).

Data augmentation

One of the most efficient ways to improve a machine learning model is to use more data for training. Data augmentation refers to the process of applying *label-preserving transformations* on the existing training data to generate more training examples. A label-preserving transform is a transform applied to an input sample x_i which will not change the corresponding target value y_i. For example, if x_i is an image with a cat, so that $y_i =$"cat", flipping the image x_i from left to right will not change the label.

If the labels are images, i.e., we are solving a prediction task where we predict a value in y_i for each pixel in the input x_i, we can also sometimes apply the same transformation to x_i and y_i both and achieve the same transformation in the target as in the input: see Figure 2. For example, if x_i is an image, and y_i is an eye fixation map, if we flip *both* x_i and y_i from left to right, we obtain a new training sample.

You can get very creative with data augmentation. For images, it is possible for example to flip images, rotate them, change the brightness of the image, take a random crop from the image, etc.

- Implement some data augmentation for your training dataset. For example, implement the following augmentation: for each pair of image and fixation map, with probability 0.5, flip both the image and the fixation map from left to right.

- You can use functions from the Python `skimage` toolbox to do many operations on images to achieve data augmentation.

[2]See Srivastava et al. (2014): "Dropout: A Simple Way to Prevent Neural Networks from Overfitting" for further information.

- Only apply data augmentation on the training dataset, not on the validation or testing datasets.

- **Remember:** only do types of data augmentation that makes sense for the problem you are solving. Adding irrelevant data is not helpful!

  For example, for a classification task, flipping an image upside down is easy, but consider: will upside down images ever appear in the application? Not necessarily.

  Another example: for eye fixation prediction, if you change the input images' colors radically, it may affect where people would look at in the image. But you have no way to modify the eye fixation map appropriately, so this data augmentation technique is not helpful.

Parameter norm penalties

Usually in deep learning, we are optimizing a loss function of the form $L(\mathcal{D};\theta)$ where $\mathcal{D}$ is our training dataset and θ are the model parameters. Adding parameter penalties modifies the loss function to

$$L(\mathcal{D};\theta) + \alpha J(\theta),$$

where $\alpha > 0$ is regularization strength parameter, and $J(\theta)$ is some non-negative penalty function applied on the parameters.

Adding the term $J(\theta)$ limits the capacity of the model, and thus helps avoid overfitting (Figure 1). It becomes costly to choose parameters that result in a large penalty, so it is less likely the learning process will converge to such parameter values.

Different choices for J lead to different solutions being preferred. Two of the most common choices are

1. L^2 norm penalty, also called weight decay, where $J(\theta) = \frac{1}{2}\|\theta\|_2$, and
2. L^1 norm penalty, $J(\theta) = \|\theta\|_1 = \sum_i |\theta_i|$

The L^1 tends to prefer sparse solutions, where some parameters become exactly zero. This is true especially for large regularization strength α. L^2 penalty tends to penalize for large parameter values.

In Tensorflow, L^2 and L^1 regularization can be implemented in various different ways. A simple example is shown below.

```
alpha = 1e-5
my_regularizer = tf.contrib.layers.l2_regularizer(alpha)
# Define all layers similar as follows
my_conv = tf.layers.conv2d(my_input, filters=32, kernel_size=(3,3),
    kernel_regularizer=my_regularizer)
# Later in loss function
l2_loss = tf.losses.get_regularization_loss() # gets regularization
    losses from all layers in network
# Final loss is your usual data loss plus the L2 loss!
loss = tf.mean_squared_error(output, x) + l2_loss
```

- Implement weight decay(L^2 penalty) for the convolutional kernels in your network. Start with a small regularization strenght such as $\alpha = 10^{-5}$.