Lab 9

Basic Image Processing Algorithms
Fall 2016

Lab 9: Classifying CIFAR images with CNN

- Implement a predefined network architecture
- Play with the network and the training to achive better classification accuracy
- Deadline (if you do not finish until the end of the lab): **23:59, 23/11/2016**; please send it to: bipa2016fall@gmail.com
  with the subject: LAB9
  please **send only your functions**, nothing else:
  - `init_network_ex1.m`
  - `init_network_ex2.m`
  - `init_network_ex3.m` (*this is optional*)

  and please write in the body of your mail the achieved accuracies per exercises

Overview

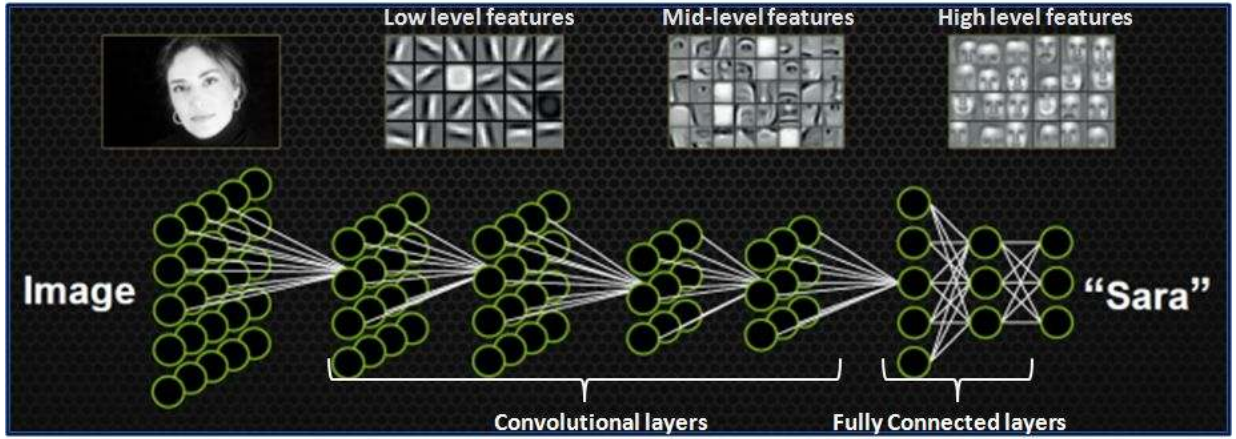
Last time we calculated handcrafted features (like HOG) on the CIFAR-10 images and used these features and the corresponding labels to train an SVM:
images → descriptors → train the SVM model → testing on the test set

Now, we train a CNN model on the images:
images → train the CNN model → testing of the model on the test set

Convolutional Neural Networks

We extract features hierarchically through convolutional layers.

But the parameters of these layers (the convolutional kernels) are not fixed, they are learnt from the data!



Source of the image:
https://devblogs.nvidia.com/parallelforall/accelerate-machine-learning-cudnn-deep-neural-network-library/
H. Lee, R. Grosse, R. Ranganath, and A. Y. Ng. "Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations." In ICML 2009.

What you'll need

You can download everything we will use in one pack from here:
https://users.itk.ppke.hu/kep/Assignments/A9/DL_pack.zip

CIFAR-10 database

MatConvNet (pre-compiled mex files for Win7 x64 are included; *for other platforms only:*

```
addpath matlab
vl_compilenn
vl_setupnn
vl_testnn )
```

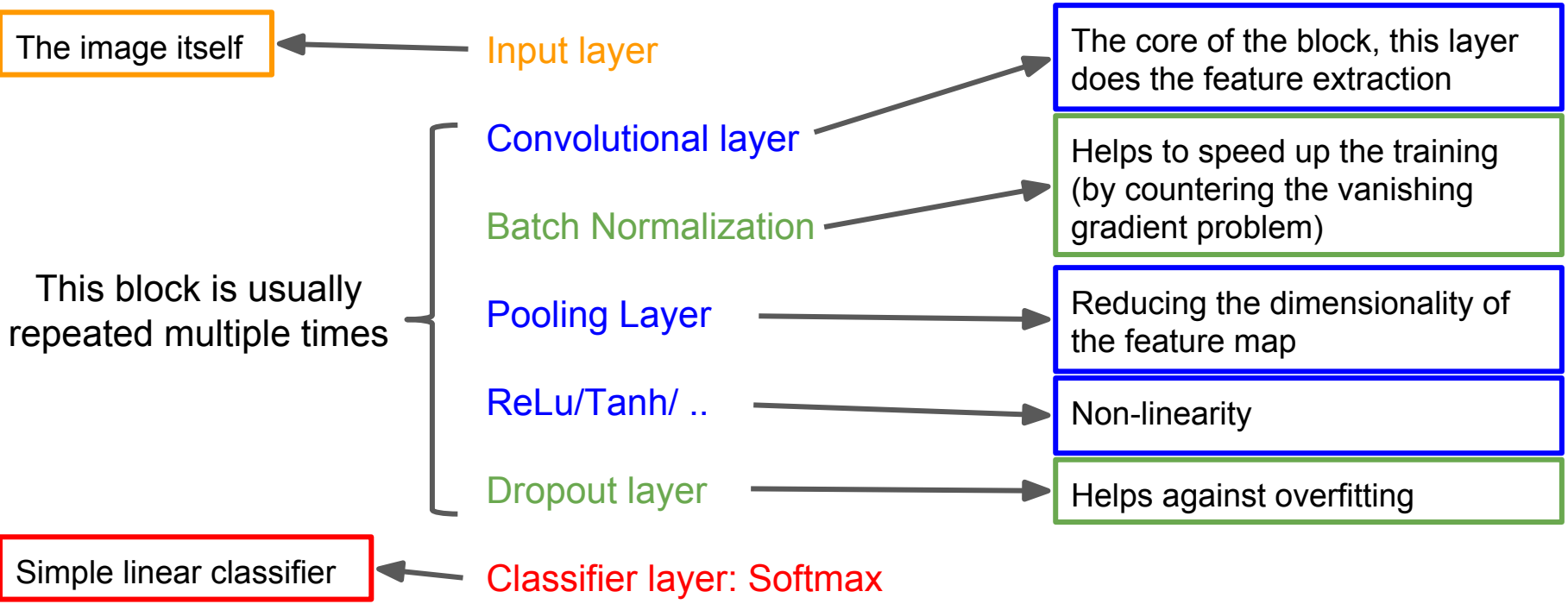
From Matlab run the **`vl_setupnn`** function (<MatConvNet>/matlab/vl_setupnn)

Starting code:

- deep_learning_tutorial.m => this is the main script, run this first to get the baseline result
- cifar_loader => this function loads the CIFAR-10 database in the proper format
- evaluation => evaluates the trained model on the CIFAR-10 test dataset
- get_batch => this function loads and preprocesses the batch data during training
- init_network => builds the CNN network architecture, initializes the weights, sets the hyper-parameters

Standard Parts of a CNN

A CNN consists of layers sequentially built on eachother:



MatConvNet

- Unofficial Matlab toolbox for working with CNNs
- Stores the layers of the network in a cell array of structs:

```
net.layers = {} ;
net.layers{end+1} = struct('type', 'conv', ...
                           'weights', {{f*randn(5,5,1,20, 'single'), zeros(1, 20, 'single')}}, ...
                           'stride', 1, ...
                           'pad', 0) ;
net.layers{end+1} = struct('type', 'pool', ...
                           'method', 'max', ...
                           'pool', [2 2], ...
                           'stride', 2, ...
                           'pad', 0) ;
net.layers{end+1} = struct('type', 'conv', ...
                           'weights', {{f*randn(5,5,20,50, 'single'),zeros(1,50,'single')}}, ...
```

Starting Code

In the starting code a very simple network architecture is initialized, trained and evaluated on the train and test samples of the CIFAR-10 dataset.
The architecture is the following:

Conv (W x H x D x Ch: 3x3x3x4, stride=1, pad=1)
BatchNorm
ReLu
Pool (Max, WxH: 3x3, stride=4)
Conv (3x3x4x4, stride=1, pad=1)
BatchNorm
ReLu
Pool (Avg, 3x3, stride=2)
Conv (4x4x4x8, stride=1, pad=0)
BatchNorm
ReLu
Conv (1x1x8x10, stride=1, pad=0)
BatchNorm
Softmax

Changes of the Data dimension during forward pass:

Input dimension: **batch_size x 32 x 32 x 3**
Output dimension: **batch_size x 32 x 32 x 4**

Input dimension: **batch_size x 32 x 32 x 4**
Output dimension: **batch_size x 8 x 8 x 4**

Input dimension: **batch_size x 8 x 8 x 4**
Output dimension: **batch_size x 8 x 8 x 4**

Input dimension: **batch_size x 8 x 8 x 4**
Output dimension: **batch_size x 4 x 4 x 4**

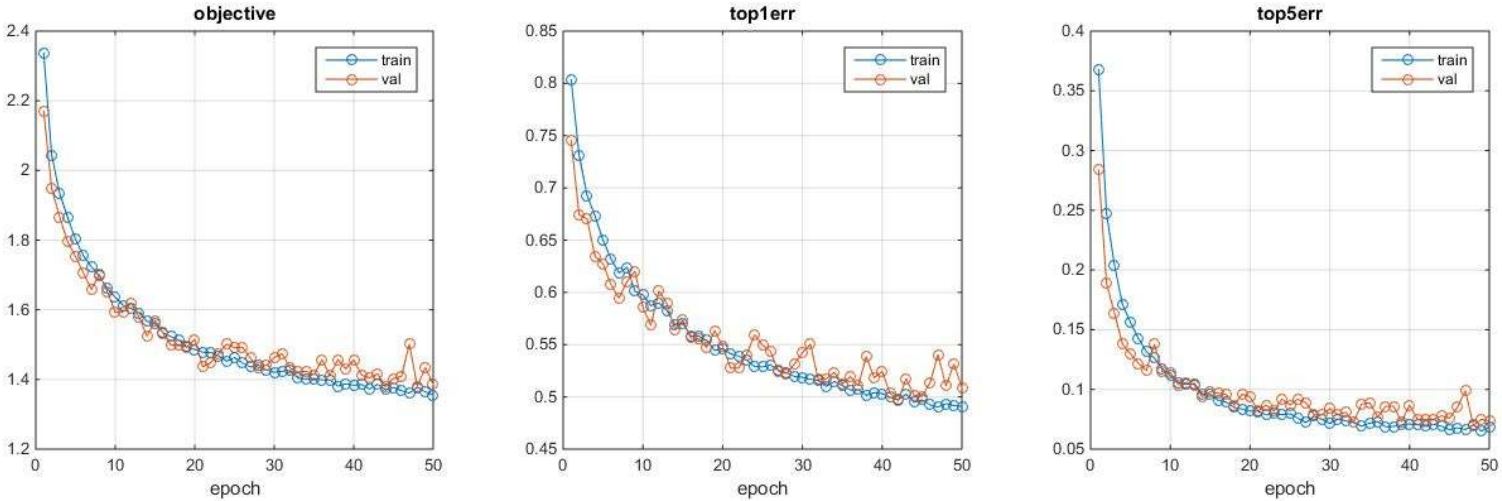
Input dimension: **batch_size x 4 x 4 x 4**
Output dimension: **batch_size x 1 x 1 x 8**

Input dimension: **batch_size x 1 x 1 x 8**
Output dimension: **batch_size x 1 x 1 x 10**

Input dimension: **batch_size x 1 x 1 x 10**
Output dimension: **batch_size x 10**

Starting Code

For training we will use the MatConvNet ‘cnn_train()’ function. This function preforms gradient descent with backpropagation algorithm. During the training it prints and plots the value of the loss function on the train- and validation set, and the top1 and top5 error rates:



Exercise 1

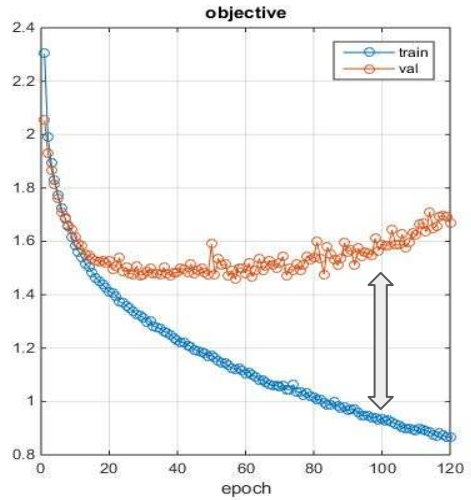
With the current settings the test accuracy is around 50% after training the network.

Your task is to increase this training accuracy:

- by changing the architecture: introducing new layers,
- increasing the dimension of the layers,

Increasing the model size could lead to overfitting!!

We will use dropout layers to fight it.



A sure sign of overfitting if there is a big gap between the train and validation loss.

Exercise 1: Implement a Deeper/Wider Architecture

Implement the following architecture, train it on the CIFAR10 train set and test it on the test set:

Conv (3x3x3x8, stride=1, pad=1)
BatchNorm
ReLu
Pool (Max, WxH: 3x3, stride=2)

Conv (3x3x8x16, stride=1, pad=1)
BatchNorm
ReLu
Pool (Avg, 3x3, stride=2)

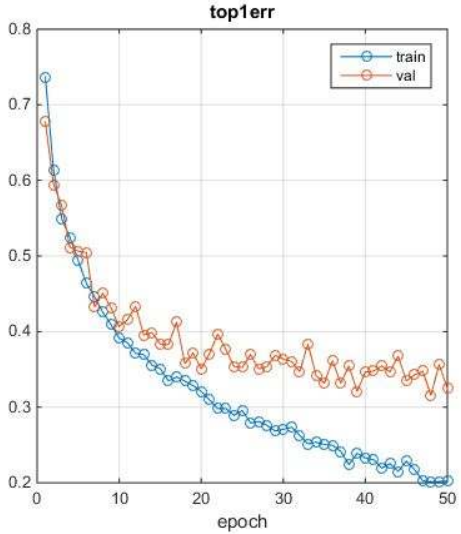
Conv (3x3x16x32, stride=1, pad=1)
BatchNorm
ReLu
Pool (Avg, 3x3, stride=2)

Conv (4x4x32x64, stride=1, pad=0)
BatchNorm
ReLu

Conv (1x1x64x10, stride=1, pad=0)
BatchNorm
Softmax

Expected Accuracy: around 0.6
(*out best result: 0.632*)

Expected Learning Curve:



Exercise 2: Add dropout

With the new network architecture we can experience a little overfitting.

Use Dropout layer to counter it:

Conv (3x3x3x8, stride=1, pad=1)
BatchNorm
ReLu
Pool (Max, WxH: 3x3, stride=2)

Conv (3x3x8x16, stride=1, pad=1)
BatchNorm
ReLu
Pool (Avg, 3x3, stride=2)

Conv (3x3x16x32, stride=1, pad=1)
BatchNorm
ReLu
Pool (Avg, 3x3, stride=2)

Conv (4x4x32x64, stride=1, pad=0)
BatchNorm
ReLu

Conv (1x1x64x10, stride=1, pad=0)
BatchNorm
Softmax

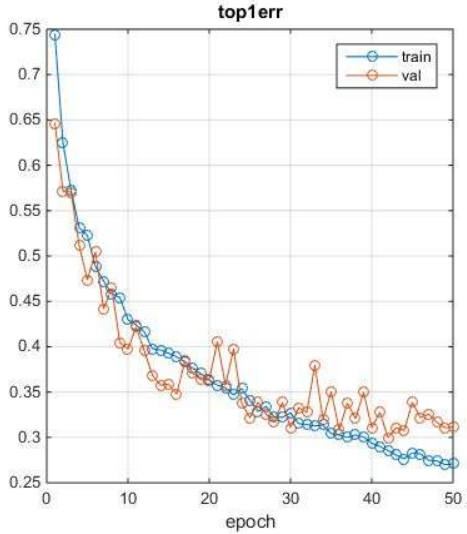
Add dropout layers here (drop_rate= 0.1)

```
net.layers{end+1} = struct('type', 'dropout', 'rate',
drop_rate);
```

Exercise 2: Add dropout

Expected Accuracy: 0.645 (*our best result*)

Expected Learning Curve:



Optional Exercise: Improve the Network to get 70%

Experiment with different architectures

- width (increase/decrease the number of channels in the conv layers)
- depth (add extra layers/layer blocks)
- regularization (play with the dropout and weight decay parameters)

Add more data (we used only 10k out of the 50k in the training set)

Let it learn for a longer time (increase the number of epochs)

Experiment with the learning rate parameter