TAMPERE UNIVERSITY OF TECHNOLOGY

SIGNAL INTERPRETATION

Lecture 6: ConvNets

February 16, 2016

Heikki Huttunen

heikki.huttunen@tut.fi

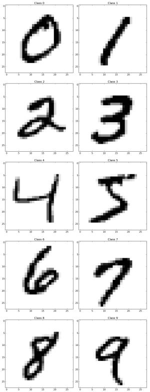
Department of Signal Processing
Tampere University of Technology

CONVNETS

Continued from previous slideset

Convolutional Network: Example

- Let’s train a convnet with the famous MNIST dataset.
- MNIST consists of 60000 training and 10000 test images representing handwritten numbers from US mail.
- Each image is 28×28 pixels and there are 10 categories.
- Generally considered an easy problem: Logistic regression gives over 90% accuracy and convnet can reach (almost) 100%.
- However, 10 years ago, the state of the art error was still over 1%.



Convolutional Network: Example

```
# Training code (modified from mnist_cnn.py at
       Keras examples)
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers.core import Dense, Dropout,
       Activation, Flatten
from keras.layers.convolutional import
       Convolution2D, MaxPooling2D

# We use the handwritten digit database "MNIST".
# 60000 training and 10000 test images of
# size 28x28
(X_train, y_train), (X_test, y_test) = mnist.
       load_data()

num_featmaps = 32    # This many filters per layer
num_classes = 10     # Digits 0,1,...,9
num_epochs = 50      # Show all samples 50 times
w, h = 5, 5          # Conv window size
```

```
model = Sequential()

# Layer 1: needs input_shape as well.
model.add(Convolution2D(num_featmaps, w, h,
          input_shape=(1, 28, 28),
          activation = 'relu'))

# Layer 2:
model.add(Convolution2D(num_featmaps, w, h,
       activation = 'relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

# Layer 3: dense layer with 128 nodes
# Flatten() vectorizes the data:
# 32x10x10 -> 3200
# (10x10 instead of 14x14 due to border effect)
model.add(Flatten())
model.add(Dense(128, activation = 'relu'))
model.add(Dropout(0.5))

# Layer 4: Last layer producing 10 outputs.
model.add(Dense(num_classes, activation='softmax'))

# Compile and train
model.compile(loss='categorical_crossentropy',
       optimizer='adadelta')
model.fit(X_train, Y_train, nb_epoch=100)
```

Convolutional Network: Training Log

- The code runs for about 5-10 minutes on a GPU.
- On a CPU, this would take 1-2 hours (1 epoch ≈ 500 s)

```
Using gpu device 0: Tesla K40m
Using Theano backend.
Compiling model...
Model compilation took 0.1 minutes.
Training...
Train on 60000 samples, validate on 10000 samples
Epoch 1/10
60000/60000 [================] — 31s — loss: 0.2193 — acc: 0.9322 — val_loss: 0.0519 — val_acc: 0.9835
Epoch 2/10
60000/60000 [================] — 31s — loss: 0.0807 — acc: 0.9758 — val_loss: 0.0398 — val_acc: 0.9863
Epoch 3/10
60000/60000 [================] — 31s — loss: 0.0581 — acc: 0.9825 — val_loss: 0.0322 — val_acc: 0.9898
Epoch 4/10
60000/60000 [================] — 31s — loss: 0.0500 — acc: 0.9851 — val_loss: 0.0276 — val_acc: 0.9913
Epoch 5/10
60000/60000 [================] — 31s — loss: 0.0430 — acc: 0.9872 — val_loss: 0.0287 — val_acc: 0.9906
Epoch 6/10
60000/60000 [================] — 31s — loss: 0.0387 — acc: 0.9882 — val_loss: 0.0246 — val_acc: 0.9922
Epoch 7/10
60000/60000 [================] — 31s — loss: 0.0352 — acc: 0.9897 — val_loss: 0.0270 — val_acc: 0.9913
Epoch 8/10
60000/60000 [================] — 31s — loss: 0.0324 — acc: 0.9902 — val_loss: 0.0223 — val_acc: 0.9928
Epoch 9/10
60000/60000 [================] — 31s — loss: 0.0294 — acc: 0.9907 — val_loss: 0.0221 — val_acc: 0.9926
Epoch 10/10
60000/60000 [================] — 31s — loss: 0.0252 — acc: 0.9922 — val_loss: 0.0271 — val_acc: 0.9916
Training (10 epochs) took 5.8 minutes.
```

Save and Load the Net

- The network can be saved to disk in two parts:
  - Network topology as JSON or YAML: `model.to_json()` or `model.to_yaml()`. The resulting string can be written to disk using `.write()` of a file object.
  - Coefficients are saved in HDF5 format using `model.save_weights()`. HDF5 is a serialization format similar to `.mat` or `.pkl`
- Alternatively, the net can be pickled, although this is not recommended.
- Read back to memory using `model_from_json` and `load_weights`


```yaml
class_mode: categorical
layers:
- W_constraint: null
  W_regularizer: null
  activation: relu
  activity_regularizer: null
  b_constraint: null
  b_regularizer: null
  border_mode: valid
  dim_ordering: th
  init: glorot_uniform
  input_shape: !!python/tuple [1, 28, 28]
  name: Convolution2D
  nb_col: 5
  nb_filter: 32
  nb_row: 5
  subsample: &id001 !!python/tuple [1, 1]
- W_constraint: null
  W_regularizer: null
  activation: relu
  activity_regularizer: null
  b_constraint: null
  b_regularizer: null
  border_mode: valid
  dim_ordering: th
  init: glorot_uniform
  name: Convolution2D
  nb_col: 5
  nb_filter: 32
  nb_row: 5
  subsample: *id001
- border_mode: valid
  dim_ordering: th
  name: MaxPooling2D
  pool_size: &id002 !!python/tuple [2, 2]
  strides: *id002
- {name: Dropout, p: 0.25}
- {name: Flatten}
```


Part of network definition in YAML format.

Network Structure

- It is possible to look into the filters on the convolutional layers.

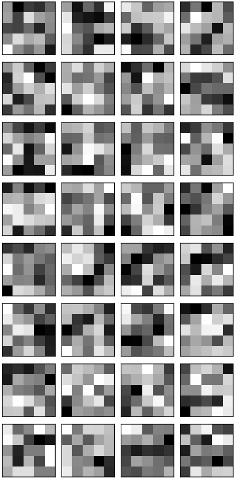
```
# First layer weights (shown on the right):
weights = model.layers[0].get_weights()[0]
```

- The second layer is difficult to visualize, because the input is 32-dimensional:

```
# Zeroth layer weights:
>>> model.layers[0].get_weights()[0].shape
(32, 1, 5, 5)
# First layer weights:
>>> model.layers[1].get_weights()[0].shape
(32, 32, 5, 5)
```

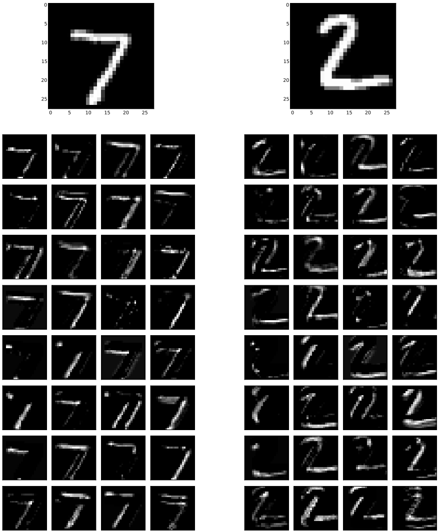
- The dense layer is the 5th (conv → conv → maxpool → dropout → flatten → dense).

```
# Fifth layer weights map 3200 inputs to 128 outputs.
# This is actually a matrix multiplication.
>>> model.layers[5].get_weights()[0].shape
(3200, 128)
```



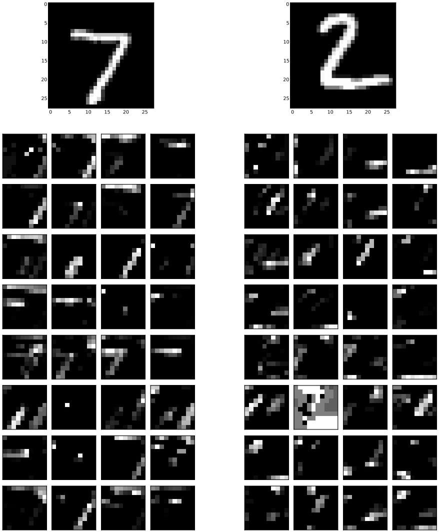
Network Activations

- The layer outputs are usually more interesting than the filters.
- These can be visualized as well.
- For details, see Keras FAQ.



Second Layer Activations

- On the next layer, the figures are downsampled to 12x12.
- This provides *spatial invariance*: The same activation results although the input would be slightly displaced.



DEEP LEARNING HIGHLIGHTS OF 2015-2016

2015 was Full of Breakthroughs: Let's See Some of Them

Feature Selection

- One of the benefits of logistic regression is its ability of *feature selection*.
- More specifically, LR can choose the most essential set of good features and discard the rest.
- Helps in high-dimensional cases.
- Improves performance by removing "confusers"; *i.e.,* measurements which have no predictive value (or may even degrade performance).

Traditional Approaches to Feature Selection

[fragile]

- **Variance based selection**: Retain features with high variance.
  - Simple to implement; poor performance as variance may not be related to feature importance.
  - `sklearn.feature_selection.VarianceThreshold`
- **Statistics based selection**: Apply statistical tests for dependence between features and labels, *e.g.,* chi-squared test.
  - Good if the assumptions are correct (often not the case).
  - `sklearn.feature_selection.SelectKBest`.

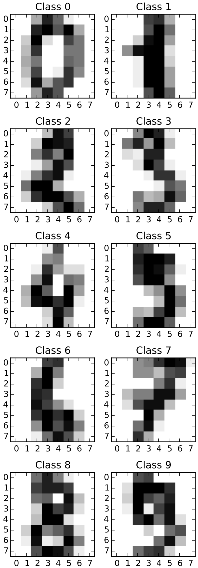
Traditional Approaches to Feature Selection

- **Recursive selection**: Progressively add or remove features that seem to improve performance the most.
  - Forward selection starts with empty set and adds variables one by one.
  - Backward elimination starts with full set and removes variables one by one.
  - There are also hybrid versions that alternate between addition and removal.
  - `sklearn.feature_selection.RFECV` implements recursive feature elimination with cross-validated scoring.

Example

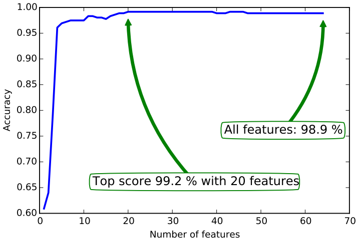
- Consider an example of classifying the *digits* dataset (`sklearn.datasets.load_digits`).
- We use LDA classifier with recursive feature elimination.
- For simplicity, we consider only two classes (zeros and ones).

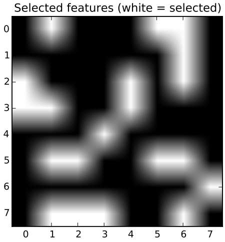
```
from sklearn.datasets import load_digits
from sklearn.lda import LDA
from sklearn.feature_selection import RFECV

digits = load_digits()

# Use only classes 0 and 1
X = digits.data[digits.target < 2, :]
y = digits.target[digits.target < 2]

# Select features
rfecv = RFECV(estimator=LDA())
rfecv.fit(X, y)

# Scores and feature sets are here
scores = rfecv.grid_scores_
mask = rfecv.support_.reshape(8, 8)
```



Example

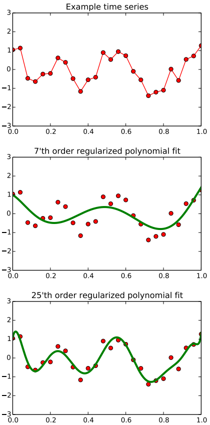
- The smallest high scoring set of features consists of 20 pixels.
- There are also larger sets with equal score.
- However, using all features will give a lower score.





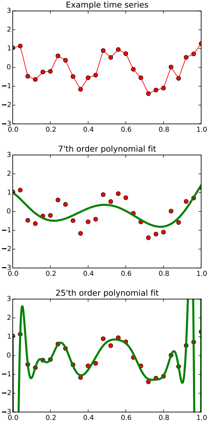
Regularization

- A more recent approach to feature selection is *regularization*.
- The traditional use is in ill-posed cases (*e.g.*, fewer samples than dimensions), or to prevent *overfitting*.
- Regularization adds a penalty term to the fitting error.
- The model is encouraged to use small coefficients.
- Large coefficients are expensive, so the model can afford to fit only to the major trends.
- On the right, the high order model has a good expression power, but does still not follow the noise patterns.



Overfitting

- Generalization is also related to *overfitting*.
- On the right, a polynomial model is fitted to a time series to minimize the error between the samples (red) and the model (green).
- As the order of the polynomial increases, the model starts to follow the data very faithfully.
- Low-order models do not have enough expression power.
- High-order models are over-fitting to noise and become "unstable" with crazy values near the boundaries.



Sparsity

- Regularization also enables the design of *sparse* classifiers.
- In this case, the penalty term is designed such that it favors **zero** coefficients.
- A zero coefficient for a linear operator is equivalent to discarding the corresponding feature altogether.
- The plots illustrate the model coefficients without regularization, with traditional regularization and sparse regularization.
- The importance of sparsity is twofold: The model can be used for feature selection, but often also generalizes better.