AIMS Big Data
Lectures 3: Deep learning 1 of 2
Andrea Vedaldi
MT 2016

For slides and up-to-date information:
http://www.robots.ox.ac.uk/~vedaldi/teach.html

Linear predictor

\[ F(x) = \langle w, x \rangle \]

Data representations

Using linear predictors on non-vectorial data

An encoder maps the data into a vectorial representation

Allows linear predictors to be applied to images, text, sound, videos, ...

\[ F(x) = \langle w, \Phi(x) \rangle \]

Meaningful representation

Semantic similarity

Vector similarity (distance)

embedding space \( \mathbb{R}^d \)

\( \Phi \) is invariant to nuisance factors, sensitive to semantic variations
Learning predictors

- labelled data \((x_1, y_1), (x_2, y_2), \ldots\)
- encoder \(\Phi\)
- learning large-scale optimiser
- predictor parameters \(w^*\)

\[ w^* = \arg\min_{w} E(w) \]

Good representations

- Main desiderata
  - Powerful: meaningful similarity / untangles factors
  - Cheap: fast to evaluate (can be computed on the fly)
  - Compact: small code (takes little RAM, disk, IO)

- Others
  - Easy to learn (when not hand-crafted)
  - Easy to implement

- Deep learning \(\Phi\) predictor
  - Handcrafted features
  - Kernel embedding
  - Metric learning
  - Deep learning

- Handcrafted features \(\Phi\) predictor
  - Kernel embedding

- Handcrafted features \(\Phi\) predictor
  - Metric learning

- Handcrafted features \(\Phi\) predictor
  - Deep learning
Histogram of Oriented Gradients

[Hog 1999, Dalal & Triggs 2005]

HOG captures the local gradient (edge) orientations in the image

---

Bag of visual words

[Sivic & Zisserman 2003, Csurka et al. 2004, Nowak et al. 2006]

BoVW construction
- Extract local descriptor densely
- Quantise descriptors
- Form histogram

Note that BoVW discards spatial information

---

BoVW intuition

Discarding spatial information gives lots of invariance
Visual words represent “iconic” image fragments

---

Fisher Vector (FV)

[Perronnin et al. ECCV 201, Sharma Hussain Jurie ECCV 2010, Sanchez et al. 2103]

\[
\Phi = \begin{bmatrix} v_1^T \ u_1^T \ v_2^T \ u_2^T \ \vdots \ v_K^T \ u_K^T \end{bmatrix}
\]

FV encoding

\[
v_k = \frac{1}{M\sqrt{2\pi}} \sum_{i=1}^{M} \gamma_k(x_i) \frac{x_i - \mu_k}{\sigma_i}
\]

\[
u_k = \frac{1}{M\sqrt{2\pi}} \sum_{i=1}^{M} \gamma_k(x_i) \left( \frac{x_i - \mu_k}{\sigma_i} - 1 \right)^2
\]

association strength

Gaussians \((\mu_k, \Sigma_k)\)
Some fundamental ideas

Local and translation invariant operators
- gradients, filters, visual words

Untangling
- sparsity, quantisation

Pooling
- max, sum, spatial pooling

Deep learning
- Φ predictor

Handcrafted features
- kernel embedding

Metric learning

Learned representation

The perceptron and the importance of convolution

Elements of convolutional neural networks

Learning a deep neural network with backpropagation

Applications

Perceptron
An early neural network by [Rosenblatt 57]

The perceptron maps a data vector $x$ to a posterior probability value $y$ (for example the probability that $x$ is an image of a bicycle as opposed to something else):

$$f(x, w) \rightarrow P(y = 1 \mid x, w)$$

The perceptron computes this probability by weighing the vector elements, summing them, and then applying a non-linear activation function:

$$S(z) = \frac{1}{1 + e^{-z}}$$

The activation function in the perceptron is a sigmoid, which converts the range $(-\infty, +\infty)$ into probability values $(0, 1)$.

This is how real scores are converted into probabilities.
**Functional form**

The perceptron:
1. Maps a vector \( x \) to a real score using an affine projection \((w, b)\).
2. Transforms the score into a probability value by applying the sigmoid \( S \).

Note: there usually is a constant bias term \( b \) added to the score. This can be implemented by extending the input vector with a constant element equal to 1.

\[
f(x; w) = S(\langle w, x \rangle + b) = \frac{1}{1 + e^{-\langle w, x \rangle - b}}
\]

**Perceptron**

**Learning from example data: non-linear least-square regression**

The perceptron can be seen as a parametric function from an input space \( X \) to an output space \( Y \):

\[
E(w) = \frac{1}{N} \sum_{i=1}^{N} (S(\langle w, x \rangle + b) - y_i)^2
\]

This problem is non-linear due to the activation function \( S \). It needs to be solved by an iterative method such as gradient descent.

**Probabilistic loss function**

Given the probabilistic nature of the perceptron output, usually the fitting criterion is not least square, but maximum log-likelihood. The log-likelihood is computed as follows:

- The posterior probability of the 0-1 label \( y_i \) can be expressed as:
  \[ P(y_i | x_i, w) = f(x_i, w)^{y_i} (1 - f(x_i, w))^{1-y_i} \]

- The negative log-likelihood of the parameters is:
  \[- \log P(y_i | x_i, w) = -y_i \log f(x_i, w) - (1 - y_i) \log(1 - f(x_i, w)) \]

- The empirical negative log-likelihood is obtained by averaging the negative log-likelihood over all the training data points:
  \[ E(w) = -\frac{1}{N} \sum_{i=1}^{N} y_i \log f(x_i, w) + (1 - y_i) \log(1 - f(x_i, w)) \]

Just like the squared objective of least square, this objective function can be minimised by using an iterative method such as gradient descent.

**Multi-class perceptron**

Multiple perceptrons can be combined to predict more than two classes.
Each perceptron computes the score \( x^2 \) for a class hypothesis \( c = 1, \ldots, C \).

The vector of scores \( x^1 \) is mapped to a vector of probabilities \( x^3 \) using the softmax operator, which is a generalisation of the sigmoid.
Softmax = sigmoid for 2 classes

In the binary case, the softmax is the same as the sigmoid:

\[
\frac{e^{x_1}}{e^{x_1} + e^{x_2}} = \frac{e^z}{1 + e^{-z}} = \frac{1}{1 + e^{-z}} = S((w, x) + b)
\]

The log-likelihood and objective function for a multi class perceptron are given by:

\[
E(W) = \frac{1}{N} \sum_{i=1}^{N} \left( -w_{y_i}^T x_i - b_{y_i} + \log \sum_{c=1}^{C} e^{w_c^T x_i + b_c} \right)
\]

This loss function is sometimes called cross-entropy. It measures the discrepancy between
- the empirical posterior distributions \(Q_i(y_i|x_i)\) and
- the predicted posterior distributions \(P_i(y_i|x_i, w, b)\).

Multi-class perceptron

Learning from example data

Deep architectures

Multi-layer perceptron (MLP)

The perceptron and the importance of convolution

Elements of convolutional neural networks

Learning a deep neural network with backpropagation

Applications
A signal processing notation

The data manipulated by a CNN has the form of 3D tensors. These are interpreted as discrete vector fields \( \mathbf{x} \), assigning a feature vector \((x_{uv1}, \ldots, x_{uvC})\) at each spatial location \((v,u)\).

A colour image is a simple example of a vector field with 3D features (RGB):
Linear convolution

With a bank of 3D filters

\[ y_{v'w'c'} = b_{c'} + \sum_{v=1}^{H} \sum_{w=1}^{W} \sum_{c=1}^{C} x_{v' + v - 1,w' + w - 1,c} f_{v'w'c'} \]

Linear convolution applies a bank of linear filters \( F \) to the input tensor \( x \).
- Input tensor \( x = H \times W \times K \) array
- Filter bank \( F = H \times W \times K \times Q \) array
- Output tensor \( y = (H - H + 1) \times (W - W + 1) \times Q \) array

As a neural network

A bank of 256 filters (learned from data)
Each filter is 1D (it applies to a grayscale image)
Each filter is 16 \( \times \) 16 pixels
Activation functions
Non-linear functions applied to each element of a tensor

\[ y = \frac{1}{1 + e^{-x}} \] (sigmoid)
\[ y = \tanh(x) \] (hyperb. tan)
\[ y = \max\{0, x\} \] (ReLU)
\[ y = \log(1 + e^x) \] (Soft ReLU)
\[ y = cx + (1 - c) \max\{0, x\} \] (Leaky ReLU)

A deep convolutional neural networks chains several filtering + non-linear activation function sequences.

The non-linear activation functions are essential. Why?

Multiple layers
Convolution, gating, convolution, ...

Downsampling
by a factor S amounts to keeping only one every S pixels, discarding the others.

Filter banks often incorporate, or are followed by, 2x output downsampling.

Downsampling is often matched with an increase in the number of feature channels.

As depth increases the volume of the tensors decreases, but slowly.

Padding
Padding virtually extends the input tensor with zeros

\[ \text{padding pad } P \]
\[ \text{convolution } (F, b) \]
\[ \text{downsampling stride } S \]
Local contrast normalisation

Normalise image/feature patches

\[
y_{ijq} = \frac{x_{ijq} - \mu_{ijq}}{\sigma_{ijq}}
\]

\[
\mu_{ijq} = \frac{1}{|N(i,j)|} \sum_{(u,v) \in N(i,j)} y_{uvq}
\]

\[
\sigma_{ijq}^2 = \frac{1}{|N(i,j)|} \sum_{(u,v) \in N(i,j)} (y_{uvq} - \mu_{ijq})^2
\]

Example

It has a local equalising effect:

\[
\begin{align*}
0 & \rightarrow 0 \\
1 & \rightarrow 0.5 \\
-1 & \rightarrow 0.5 \\
0 & \rightarrow 0
\end{align*}
\]

Local feature normalisation

Across feature channels rather than spatially

Operates at each spatial location independently

\[
y_{jk} = x_{jk} \left( \kappa + \alpha \sum_{q \in G(k)} x_{jq}^2 \right)^{-\beta}
\]

Normalise groups G(k) of feature channels

Groups are usually defined in a sliding window manner

Spatial pooling

Local translation invariance

Pooling computes the average or max of a feature response in a small image spatial neighbourhood.

It is applied to each feature channel independently and in parallel.
Spatial pooling

Variants

- **max pooling**
  
  \[ y_{ijk} = \max_{(u,v) \in N(i,j)} x_{uvk} \]

- **sum pooling**
  
  \[ y_{ijk} = \sum_{(u,v) \in N(i,j)} x_{uvk} \]

- **L^2-sum pooling**
  
  \[ y_{ijk} = \sqrt{\sum_{(u,v) \in N(i,j)} x_{uvk}^2} \]

By far, the most common variant is **max pooling**.

Feature pooling

Across feature channels, not in space

Pooling across feature channels (filter outputs) can achieve invariance. L2 pooling, in particular, is invariant to the sign of the edge filter too.

CNN layers summary

- **Linear convolution**
  
  \[ y = F \ast x + b \]

- **ReLU**
  
  \[ y = \max\{0, x\} \]

- **Max pooling**
  
  \[ y_{ijk} = \max_{pq \in \Omega_i} x_{pqk} \]

Many common CNN architectures can be obtained by just three layer types:
- Linear convolution
- ReLU
- Max pooling

Other common layers include:
- Cross-feature normalisation such as LRN.
- Within-feature normalisation such as contrast and batch normalisation.
- Parametric and pyramid spatial pooling (see next lecture).
- Sum and stacking (for branching network topologies).

A typical CNN design

From left to right
- decreasing spatial resolution
- increasing feature dimensionality

"Fully-connected" layers
- same as convolutional, but with 1 × 1 spatial resolution
- contain most of the parameters
Convolutional layers

Each block \( c_1, c_2, \ldots, f_8 \) is in turn a composition of convolution, downsampling, ReLU, and max pooling. Downsampling and max pooling are optional.

The perceptron and the importance of convolution

Elements of convolutional neural networks

Learning a deep neural network with backpropagation

Applications

Learning a CNN

Learning CNNs classifiers

Challenge
- many parameters, prone to overfitting

Key ingredients
- very large annotated data
- heavy regularisation (dropout)
- stochastic gradient descent
- GPU(s)

Training time
- \( \sim 90 \) epochs
- days—weeks of training
- requires processing \( \sim 150 \) images/sec

\[
\arg\min \ E(\mathbf{w}_1, \mathbf{w}_2, \ldots, \mathbf{w}_8)
\]

Stochastic gradient descent (with momentum, dropout, …)
Stochastic gradient descent

The objective function is an average over many data points:

\[ E(w) = \frac{1}{N} \sum_{i=1}^{N} E_i(w) \]

Key idea: approximate the gradient sampling a point at a time:

\[ w_{t+1} = w_t - \eta_t \nabla E_i(w_t), \quad i \sim U(\{1, 2, \ldots, N\}) \]

Details:
- **Epochs**: all points are visited sequentially, but in random order
- **Validation**: evaluate \( E(w_t) \) on an held-out validation set to diagnose objective decrease
- **Learning rate** \( \eta_t \): is decreased tenfold once the objective \( E(w_t) \) stops decreasing.
- **Momentum**: the gradient estimate is smoothed by using a moving average:

\[ m_{t+1} = 0.9 m_t + \eta_t \nabla E_i(w_t), \quad w_{t+1} = w_t - m_{t+1} \]

Chain rule: scalar version

\[ x_n = (f_n \circ f_{n-1} \circ \ldots \circ f_2 \circ f_1)(x_0) \]

\[ \frac{dx_n}{dx_0} = \frac{df_n}{dx_n-1} \times \frac{df_{n-1}}{dx_{n-2}} \times \ldots \times \frac{df_2}{dx_1} \times \frac{df_1}{dx_0} \]
CNN layers as tensor valued functions

E.g. linear convolution = bank of 3D filters

\[ y = F \star x + b \]

<table>
<thead>
<tr>
<th></th>
<th>height</th>
<th>width</th>
<th>channels</th>
<th>instances</th>
</tr>
</thead>
<tbody>
<tr>
<td>input x</td>
<td>H</td>
<td>W</td>
<td>C</td>
<td>1 or N</td>
</tr>
<tr>
<td>filters F</td>
<td>( H_f )</td>
<td>( W_f )</td>
<td>C</td>
<td>K</td>
</tr>
<tr>
<td>output y</td>
<td>( H - H_f + 1 )</td>
<td>( W - W_f + 1 )</td>
<td>K</td>
<td>1 or N</td>
</tr>
</tbody>
</table>

Vector representation of tensor-valued functions

Derivative of tensor-valued functions

Derivative (Jacobian): every output element w.r.t. every input element!

Using vec() and matrix notation

Chain rule: tensor version

The vec operator allows us to use a familiar matrix notation for the derivatives
The (unbearable) size of tensor derivatives

The size of these Jacobian matrices is huge. Example:

\[ \begin{array}{c}
\text{vec } y \\
\end{array} \quad \begin{array}{c}
\text{vec } f \\
\end{array} \quad \begin{array}{c}
\text{vec } x \\
\end{array} \]

\[ \frac{\partial \text{vec } f}{\partial \text{vec } x^T} \]

275 B elements

32 \times 32 \times 512

1 TB of memory required !!

32 \times 32 \times 512

Scalar

This is always the case if the last layer is the loss function

Now the Jacobian has the same size as \( x \). Example:

\[ \begin{array}{c}
\text{vec } x^T \\
\end{array} \quad \begin{array}{c}
\text{vec } x \\
\end{array} \]

\[ \begin{array}{c}
\text{vec } y \\
\end{array} \]

524K elements

32 \times 32 \times 512

Just 2MB of memory

1 \times 1 \times 1

Backpropagation

Assume that \( x_n \) is a scalar (e.g. loss)

\[ df_n \]

\[ d \text{vec } x_{n-1}^T \]

\[ f_n \]

\[ f_{n-1} \]

\[ f_2 \]

\[ f_1 \]

\[ x_0 \]

\[ x_1 \]

\[ \ldots \]

\[ x_n \]

\[ \frac{d \text{vec } f_n}{d \text{vec } x_{n-1}^T} \times \frac{d \text{vec } f_{n-1}}{d \text{vec } x_{n-2}^T} \times \ldots \times \frac{d \text{vec } f_2}{d \text{vec } x_1^T} \times \frac{d \text{vec } f_1}{d \text{vec } x_0^T} \]

compute this first !

small explicitly compute

uber matrices do not explicitly compute

Backpropagation

Assume that \( x_n \) is a scalar (e.g. loss)

\[ df_n \circ f_{n-1} \]

\[ d \text{vec } x_{n-2}^T \]

\[ f_n \]

\[ f_{n-1} \]

\[ f_2 \]

\[ f_1 \]

\[ x_0 \]

\[ x_1 \]

\[ \ldots \]

\[ x_n \]

\[ \frac{d \text{vec } f_n \circ f_{n-1}}{d \text{vec } x_{n-2}^T} \times \ldots \times \frac{d \text{vec } f_2}{d \text{vec } x_1^T} \times \frac{d \text{vec } f_1}{d \text{vec } x_0^T} \]

small explicitly compute

uber matrices do not explicitly compute
Assume that $x_n$ is a scalar (e.g. loss).

Backpropagation

$\mathbf{x}_n \mathbf{x}_{n-1} \ldots \mathbf{x}_1 \mathbf{x}_0$

$\frac{df_n \circ \ldots \circ f_2}{d \text{vec } x_1} \times \frac{d \text{vec } f_1}{d \text{vec } x_0}$

small explicitly compute
uber matrix
do not explicitly compute

The “BP-reversed” layer

Projected function derivative

The $f_{BP}$ function computes the derivative of $f$ projected onto $\mathbf{p}$.

Anatomy of a building block
Anatomy of a building block

**forward (eval)**

\[
y = vl\_nnconv(x, W, b)
\]

**backward (backprop)**

\[
dzdx = vl\_nnconv(x, W, b, dzdy)
\]

Backpropagation network

**BP induces a “reversed” network**

where \( dx_i = \frac{df_i \circ \cdots \circ f_{i+1}}{d \text{vec} x_i} \)

Note: the BP network is linear in \( dx_1, \ldots, dx_n, dx_0 \). Why?
BP induces a “transposed” network

Backpropagation network

Conv, ReLU, MP and their transposed blocks

Sufficient statistics and bottlenecks

Usually much less information is needed

The perceptron and the importance of convolution

Elements of convolutional neural networks

Learning a deep neural network with backpropagation

Applications
Applications

Label individual pixels
Semantic image segmentation

Face analysis
Detection, verification, recognition, emotion, 3D fitting
E.g. VGG-Face

Text spotting
Detection, word recognition, character recognition
E.g. SynthText and VGG-Text
http://zeus.robots.ox.ac.uk/textsearch/#/search/
Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation

Demo