Data representations
Using linear predictors on non-vectorial data

A linear predictor can be used to classify vector data. The question is how such a predictor can be applied to images, text, videos, or sounds.

This is solved by an encoder, which maps the data to a vectorial representation:

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

Meaningful representation

Semantic similarity
Vector similarity (distance)

\[ \Phi(x) \text{ is invariant to nuisance factors, sensitive to semantic variations} \]
The perceptron

Elements of convolutional neural networks

Learning a deep neural network with backpropagation

Applications
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):

$$f(x, w) = P(y = 1 | x, w)$$

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

$$\sum w_i x_i + b = z$$

$$P(y = 1 | x, w, b) = S(z) = \frac{1}{1 + e^{-z}}$$

Activation function (sigmoid)

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.

Perceptron as a scoring function

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 $S$.

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 $Y$:

$$f(x; w) = S((w, x) + b) = \frac{1}{1 + e^{-w_1 x_1 - \cdots - w_0 x_0 - b}}$$

The parameters $(w, b)$ of the perceptron are learned empirically by fitting the function to example data $(x_1, y_1), (x_2, y_2), \ldots, (x_N, y_N)$. This can be done by solving a least-square problem:

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

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

Perceptron

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.

The perceptron

Elements of convolutional neural networks

Learning a deep neural network with backpropagation

Applications

Discovery of oriented cells in the visual cortex

[Hubel and Wiesel 59]
The data manipulated by a CNN has the form of 3D tensors. These are interpreted as discrete vector fields \( x \), 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):

\[
\begin{align*}
y_{v'u'c'} &= b_{c'} + \sum_{\delta=1}^{H} \sum_{\zeta=1}^{W} \sum_{c=1}^{C} x_{v'\delta, u'\zeta, c' c} f_{\delta \zeta c c'} \\
x \rightarrow y &= F \ast x + b
\end{align*}
\]

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

Linear convolution

input features

a bank of 2 filters

2-dimensional output features

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

Non-linear functions applied to each element of a tensor

Activation functions

Convolution, activation, convolution, activation, ...

A deep convolutional neural network chains several filtering & non-linear activation function sequences.
The non-linear activation functions are essential. Why?
Downsampling by a factor $S$ amounts to keeping only one every $S$ pixels, discarding the others.

Filter banks oftter incorporate, or are followed by, $2 \times$ output downsampling.

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

Overall, as depth increases the volume of the tensors decreases, but slowly.

Padding virtually extends the input tensor with zeros.

A detail: padding

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.

By far, the most common variant is max pooling.

Variants

- max pooling
- sum pooling
- $L^2$-sum pooling

By far, the most common variant is max pooling.
CNN layers summary

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
- Here the spatial resolution is $1 \times 1$ (tensors are mere vectors), but there can be thousands of channels

Label probabilities and softmax
- The last feature vector has one entry per class label which is converted to a vector of label probabilities by the softmax operator:

$$[\text{softmax}(x)]_i = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$$

The perceptron

Elements of convolutional neural networks

Learning a deep neural network with backpropagation

Applications
Learning a CNN

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:

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

uniform distribution

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:

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

Backpropagation

Computing derivatives using the chain rule

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
- ~ 90 epochs
- days—weeks of training
- requires processing ~150 images/sec

1K classes
- ~ 1K training images per class
- ~ 1M training images
Chain rule: scalar version

A composition of $n$ functions

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

\[ \frac{dx_n}{dx_0} = \frac{df_n}{dx_{n-1}} \times \frac{df_{n-1}}{dx_{n-2}} \times \ldots \times \frac{df_2}{dx_1} \times \frac{df_1}{dx_0} \]

Derivative obtained using the chain rule

Vector representation of tensor-valued functions

3D tensors

\[ \text{vec} \ y \]

\[ \text{vec} \ f \]

\[ \text{vec} \ x \]

Derivative of tensor-valued functions

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

\[ \text{vec} \ y \]

\[ \text{vec} \ f \]

\[ \text{vec} \ x \]

The vec operator allows us to use a familiar matrix notation for the derivatives
Chain rule: tensor version

Using vec() and matrix notation

\[
\begin{align*}
  & f_n & f_{n-1} & \ldots & f_2 & f_1 \\
  & x_0 & x_{n-1} & \ldots & x_1 & x_0 \\
  \end{align*}
\]

\[
\frac{d \vec{f}_n}{d \vec{x}_{n-1}} \times \frac{d \vec{f}_{n-1}}{d \vec{x}_{n-2}} \times \ldots \times \frac{d \vec{f}_2}{d \vec{x}_1} \times \frac{d \vec{f}_1}{d \vec{x}_0}
\]

The (unbearable) size of tensor derivatives

The size of these Jacobian matrices is huge. Example:

\[
\begin{align*}
  \text{vec } y & \rightarrow \text{vec } f \rightarrow \text{vec } x \\
  \frac{\partial \text{vec } f}{\partial \text{vec } x^T} & \rightarrow \text{vec } x^T \\
\end{align*}
\]

275 B elements

1 TB of memory required!!

The size of these Jacobian matrices is huge. Example:

\[
\begin{align*}
  \text{vec } y & \rightarrow \text{vec } f \rightarrow \text{vec } x^T \\
  \frac{\partial \text{vec } f}{\partial \text{vec } x^T} & \rightarrow \text{vec } x^T \\
\end{align*}
\]

275 B elements

1 TB of memory required!!

Unless the output is a scalar

Scalar

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

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

\[
\begin{align*}
  \text{vec } y & \rightarrow \text{vec } x \\
  \frac{\partial \text{vec } f}{\partial \text{vec } x^T} & \rightarrow \text{vec } x^T \\
\end{align*}
\]

524K elements

32 x 32 x 512

Just 2MB of memory

Backpropagation

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

\[
\begin{align*}
  \frac{df_n}{d \text{vec } x_{n-1}} & \times \frac{d \text{vec } f_{n-1}}{d \text{vec } x_{n-2}} \times \ldots \times \frac{d \text{vec } f_2}{d \text{vec } x_1} \times \frac{d \text{vec } f_1}{d \text{vec } x_0} \\
\end{align*}
\]

compute this first!

Just 2MB of memory

1 x 1 x 1

Small explicitly compute

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

Backpropagation

The "BP-reversed" layer

The $f^{BP}$ function computes the derivative of $f$ projected onto $p$. 

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

The projected function derivative.

The function $f$ projects onto $p$.

Implicit $\approx$ explicit

Explicitly compute small

Uber matrices do not explicitly compute small

Projected function derivative
Anatomy of a building block

**forward (eval)**

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

**backward (backprop)**

\[ dzdx = vl\_nnconv(x, W, b, dzdy) \]
BP induces a “reversed” network

The perceptron

Elements of convolutional neural networks

Learning a deep neural network with backpropagation

Applications

Semantic image segmentation

Label individual pixels
Face analysis
Detection, verification, recognition, emotion, 3D fitting

E.g. VGG-Face

Object detection
Extract individual object instances

boat: 0.853
person: 0.993
person: 0.981
person: 0.972
person: 0.907

Text spotting
Detection, word recognition, character recognition

E.g. SynthText and VGG-Text
http://zeus.robots.ox.ac.uk/textsearch/#/search/

Demo

Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation