C18 Machine Vision and Robotics

Computer Vision

Introduction

Dr Andrea Vedaldi
4 lectures, Michaelmas Term

For lecture notes, tutorial sheets, and updates see

http://www.robots.ox.ac.uk/~vedaldi/teach.html

Notes, handout and tutorial sheet

C18 materials

Look for materials in Weblearn or at

http://www.robots.ox.ac.uk/~vedaldi/teach.html

C18 Computer Vision

Overview

Prof. Andrea Vedaldi (4 lectures)
- Lecture 1: Matching, indexing, and retrieval
- Lecture 2: Convolutional neural networks
- Lecture 3: Backpropagation and automated differentiation
- Lecture 4: Applications

Prof. Victor Prisacariu (4 lectures)
- 3D vision

C18 Machine Vision and Robotics

Computer Vision

C18 materials

Notes, handout and tutorial sheet

A convolutional neural network primer

For the Oxford C18 and ARB Big Data course

Andrea Vedaldi
vedaldi@robots.ox.ac.uk

Version 1.0 August 2018

For lecture notes, tutorial sheets, and updates see

http://www.robots.ox.ac.uk/~vedaldi/teach.html
The Internet: 50 billion images and counting...

It may not contain the picture you just took...

.. but it likely contains a similar one!

All Souls College, Oxford
Goal: search a large collection for an image of the \textbf{same object}

Define a similarity function between images

\[ F(I_1, I_2) = \text{confidence that the object is the same} \]
Viewpoint and visibility

Handling a variable viewpoint
- As viewpoint changes pixels “move around” or even appear/disappear
- We need to match corresponding pixels before we can compare them

Matching and transformation
Matching can be seen as transforming or warping an image to another

Why do pixel values differ so much?

Nuisance factors:
- Viewpoint
- Visibility
- Illumination
- Camera
- Noise

Image similarity (I)
Compare images as vectors of pixels

\[ F(I_1, I_2) = -‖I_1 - I_2‖^2 \]
Matching can be seen as transforming or warping an image to another.
Similarity transformations

If the camera rotates around and translates along the optical axis, the image transforms according to a similarity: scale, rotation, and translation.

\[
\begin{bmatrix}
x' \\
y'
\end{bmatrix} = sR(\theta)\begin{bmatrix}
x \\
y
\end{bmatrix} + \begin{bmatrix}
t_x \\
t_y
\end{bmatrix} R(\theta) = \begin{bmatrix}
\cos \theta & -\sin \theta \\
\sin \theta & \cos \theta
\end{bmatrix}
\]
For pure camera rotation or if the object is planar, then the image transforms with an homography (approximated as an affine transformation).

\[
\begin{bmatrix}
x' 
y'
\end{bmatrix} = \begin{bmatrix}
a_{11} & a_{12} 
a_{21} & a_{22}
\end{bmatrix} \begin{bmatrix}
x 
y
\end{bmatrix} + \begin{bmatrix}
t_x 
t_y
\end{bmatrix}
\]

Comparing local features using normalisation
Consider corresponding feature frames \(f\) and \(f'\).
Then normalisation undoes the effect of a viewpoint change.
After normalisation, pixels are in correspondence (matched) and can be compared directly.
In practice, one compares descriptors rather than pixels. Descriptors:
- handle residual distortions, noise, illumination;
- make the representation more compact.

The most important example is the SIFT descriptor.

For each pair of image features
- Extract and normalize the corresponding image patches
- Compute their descriptor vectors
- Compare descriptors using the Euclidean distance

**Question:** how do we get the features in the first place?

Exhaustive matching

Exhaustive approach:
- Extract all possible features (all circles or all ellipses) from both images
- Test all feature pairs for possible matches

Testing all features guarantees that, if the “same feature” is visible in both images, then the corresponding patches are considered for matching.

The cost of exhaustive matching is $O(N_1 N_2)$ where $N_i$ is the number of features extracted from image $I_i$.

Even after sampling the search space, the number of all possible features $N_i$ is very large (~$10^6$).

Exhaustive matching is just too expensive.

We need a method to select a small subset of features to match.
A detector is a rule that selects a small subset of features for matching.

The key is co-variance: the selection mechanism must pick the “same” (i.e. corresponding) features after an image transformation.

Example of a co-variant detection rule: “pick all the dark blobs”.

A feature extracted by the Harris-Affine detector independently from different frames of a video.

Note that the feature seems “glued on” the scene.

In practice, descriptors are computed in a region surrounding the feature. This is because the feature “visual anchors” (e.g. blobs) look the same and would be confused during matching.
Matching all local features

Step 0: get an image pair

- The left image has \( m \) features \((f_1, d_1), \ldots, (f_m, d_m)\)
- Right image has \( n \) features \((f'_1, d'_1), \ldots, (f'_n, d'_n)\)

Step 1: detect local features \( f \) and extract descriptors \( d \)

- Local matching
  - So far we have detected and then matched local features.
  - This is because normalisation is only possible if features are unoccluded and approximately planar.
  - Small features are much more likely to satisfy such assumptions.
  - On the contrary, the image as a whole is non-planar and contains plenty of self-occlusions.

- Global matching
  - However, our goal is to compare images as a whole, not just individual patches.
  - Next, we will see how to build a global similarity score from patch-level local comparisons.
Step 2: match each descriptor to its closest one

Matching all local features

Match the $i$-th left feature to its right nearest-neighbour $\text{nn}(i)$, where:

$$\text{nn}(i) = \arg\min_{j=1, \ldots, m} \|d_i - d'_j\|^2$$

Step 3: reject ambiguous matches using the 2nd-nn test

Matching all local features

Accept a match $i \rightarrow \text{nn}(i)$ only if it is at least a fraction $\tau = 0.9$ away from other possible matches:

$$\|d_i - d'_{\text{nn}(i)}\|^2 < \tau \arg\min_{j \neq \text{nn}(i)} \|d_i - d'_j\|^2$$

Step 4: geometric verification

The final step is to test whether matches are consistent with an overall image transformation. Inconsistent matches are rejected (see RANSAC).

RANSAC: optimization robust to outliers

Input: $M$ tentative feature matches $(x_1, x'_1), \ldots, (x_M, x'_M)$.

Output: affine transformation $(A^*, T^*)$ with the largest number of inlier matches:

$$(A^*, T^*) = \arg\max_{A,T} \left\{ i : \|x'_i - Ax_i - T\| < \epsilon \right\}$$

1. Repeat a large number of times:
   A. Randomly sample a minimal subset of matches sufficient to estimate $(A, T)$.
   B. Find inliers, i.e. other matches that are compatible with $(A, T)$.
2. Return $(A^*, T^*)$ as the pair $(A, T)$ with the largest number of inliers.
By counting number of verified local feature matches

\[ F(I_1, I_2) = \# \text{ of matches after geometric verification} \]

Matching local features
Global geometric verification
Indexing using visual words
Evaluating retrieval systems

From image matching to image search

Our matching strategy can be used to search a handful of images exhaustively. However, this is far too slow to search a database of a billion or more images such as Flickr, FaceBook, or the Internet.

Example:
- \( L \) images in the database e.g. \( 10^6 - 10^{10} \) (FaceBook)
- \( N \) features per image (incl. query) e.g. \( 10^3 \) (SIFT detector)
- \( D \) dimensional feature descriptor e.g. \( 10^2 \) (SIFT descriptor)
- Exhaustive search cost: \( O(N^2 LD) \) \( 10^{11} - 10^{15} \) ops = 100 days - 300 years
- Memory footprint: \( O(NLD) \) 1TB - 1PB

Goal: develop a method to search a million or more images on a single computer in under a second (and many more on computer clusters).

Issues:
- memory footprint
- matching cost (time)
- precision and recall

The inverted index

Used by Google to search the Web instantaneously

Inverted index
- For each word, lists all documents containing it as pairs (DocID, WordCount)
- Efficient query resolution: given a word, return the corresponding list

Indexing images
- Image = document
- Word = ?

The key is to understand how to extract "words" from images

Inverted Files for Text Search Engines

Fig. 3. Complete document-level inverted file for the Keeper database. The entry for each term \( t \) is composed of the frequency \( f_t \) and a list of pairs, each consisting of a document identifier \( d \) and a document frequency \( f_{d,t} \). Also shown are the \( W_d \) values as computed for the cosine measure shown in Equation 1.
The visual vocabulary is obtained by forming K clusters of example descriptors \((d_1, \ldots, d_M)\). Here \(M\) may be in the order of a 1M, and \(K\) in the order of 10-100K.

The K cluster means \((\mu_1, \ldots, \mu_K)\) are randomly initialised. Then the K-means algorithm alternates two steps:

1. Find for each descriptor \(d_i\) the index \(\pi(d_i)\) of its closest mean:
   \[
   \pi(d_i) = \arg\min_k \|d_i - \mu_k\|^2
   \]
2. Recompute each mean \(\mu_k\) from the descriptor assigned to it:
   \[
   \mu_k = \text{average}\{d_i : \pi(d_i) = k\}
   \]

Once the means are trained, new descriptors \(d\) are quantised by mapping them to the closest mean:

\[
\pi(d) = \arg\min_k \|d - \mu_k\|^2
\]

Visual word examples. Each row is an equivalence class of patches mapped to the same cluster by K-means.
From local features to visual words

Two steps:
- **Extraction.** Extract local features and compute corresponding descriptors as before.
- **Quantisation.** Then map the descriptors to the K-means cluster centres to obtain the corresponding visual words.

**A simple but efficient global image descriptor**

Histogram of visual words

The **histogram of visual words** is the vector of the number of occurrences of the K visual words in the image:

$$h_k = | \{ d_i : \pi(d_i) = k \} |$$

If there are $K$ visual words then $h \in \mathbb{R}^K$.

The vector $h$ is a **global image descriptor**.

**Comparing histograms**

**Cosine similarity**

Comparing histograms

Histogram of visual words can be compared as vectors.

The relative distribution of visual words is more informative than their absolute number of occurrences.

This intuition is captured by the **cosine similarity**, which computes the angle of the L2-normalised histograms.

$$F(I_1, I_2) = \cos \theta = \left\langle \frac{h_1}{\|h_1\|}, \frac{h_2}{\|h_2\|} \right\rangle$$
Image similarity (III)

By comparing bag-of-words descriptors

\[
F(I_1, I_2) = \langle h_1, h_2 \rangle
\]

Search as sparse matrix multiplication

**Goal:** given a query vector \( h \), quickly compute its similarity with all the \( L \) vectors \( h_1, h_2, h_3, \ldots, h_L \) in the database (one per indexed image).

Express this as a vector-matrix multiplication:

\[
\begin{bmatrix}
0 & 0.1 & 0.2 & 0 & \ldots & 0 & \ldots & 0.1 \\
0 & 0.1 & 0 & \ldots & 0 & \ldots & 0 \\
0.2 & 0 & 0 & \ldots & 0 & \ldots & 0 \\
0.1 & 0 & 0.3 & \ldots & 0.1 & \ldots & 0.1 \\
0.1 & 0.1 & 0 & \ldots & 0 & \ldots & 0.2 \\
0.01 & 0.1 & 0 & \ldots & \ldots & \ldots & 0 \\
\end{bmatrix}
\begin{bmatrix}
h_1 \\
h_2 \\
h_3 \\
\vdots \\
h_L \\
\end{bmatrix}
\]

The naive multiplication cost is \( O(KL) \), where \( K \) is the number of visual words and \( L \) is the database size.

However, histograms are often highly sparse. If only a fraction \( \rho \ll 1 \) of entries is non-zero, then the cost reduces to \( O(\rho K L) \) or even \( O(\rho^2 K L) \).

The space required is also only \( O(\rho K L) \).

Summary: image indexing and retrieval

Given a query image \( I \), we search the database by combining the two similarities:

1. The **fast but unreliable** cosine similarity to obtain a short list of \( M \approx 100 \) possible matches.
2. The **slow but reliable** geometric verification to rerank the top \( M \) matches.

Demo

http://www.robots.ox.ac.uk/~vgg/demo/
We now have a system that can match a given picture to a large database of images (e.g. Wikipedia).

Besides speed, a good retrieval system must have two fundamental properties:

1. **Precision**, i.e., the ability to return only images that match the query.
2. **Recall**, i.e., the ability to return all the images that match the query.

Assess the quality of a ranked result list

**Precision-recall curves**

Consider all images up to rank $r$ in the list:

- **Precision** $@r$: fraction of correct results in the top $r$.
- **Recall** $@r$: fraction of relevant database images that are contained in the top $r$.

The **Average-Precision** (AP) is (roughly) the area under the PR curve.

AP is a single number summarising the overall quality of the result list.

A benchmark usually has 1) a large image database and 2) a number of test queries for which the correct answer (relevant/irrelevant images) is known.

The retrieval system is evaluated in term of **mean average precision** (mAP), which is the mean AP of the test queries.

### Evaluation of a retrieval system

We now have a system that can match a given picture to a large database of images (e.g. Wikipedia).

Besides speed, a **good retrieval system** must have two fundamental properties:

1. **Precision**, i.e., the ability to return only images that match the query.
2. **Recall**, i.e., the ability to return all the images that match the query.

### Evaluation of an image retrieval system

A benchmark usually has 1) a large image database and 2) a number of test queries for which the correct answer (relevant/irrelevant images) is known.

The retrieval system is evaluated in term of **mean average precision** (mAP), which is the mean AP of the test queries.
Dataset content
- ~ 5K images of Oxford
- An optional additional set of confounder (irrelevant) images
- 58 test queries

We would like to build a predictor function that can tell if an image $x$ contains a certain object (say a "bicycle").

We learn this function automatically from example images that do and do not contain the object.

In the simplest case, the function is a linear predictor $F(x)$:
- Images are interpreted as (high-dimensional) vectors.
- $F(x)$ projects vectors linearly to a score for the positive hypothesis.
- The sign of $F(x)$ is used as prediction.

![Diagram showing linear predictor](image)

Beyond vector data
A linear predictor applies to vector data.

However, we want to process images, text, videos, or sounds that are not necessarily vectors.

For this, we use a representation function $\Phi$, which maps the data to vectors.

Non-linear classification
Representations are used even if the data $x$ is already a vector.

They realize a non-linear classifier function which can be significantly more expressive.
Meaningful representations

Representations must be built to help the linear classifier perform discrimination.

The goal is to map the semantic similarity between data points to a corresponding vector similarity.

A good representation is:
- invariant to nuisance factors
- sensitive to semantic variations

Semantic similarity → Vector similarity (distance)

embedding space $\mathbb{R}^d$

The perceptron

An early neural network by Rosenblatt (1957)

The perceptron maps an input vector $x$ to a probability value $y$.

For example, $y$ could be the probability that $x$ is an image of a "bicycle" instead of something else.

How

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

Convolutional networks

Learning via SGD

Evaluation
The sigmoid activation function

Makes the perceptron non-linear

The activation function in the perceptron is a sigmoid:

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

The sigmoid converts real scores the range \((-\infty, +\infty)\) into probability values in the range \((0, 1)\).

It has several remarkable properties, such as the following identity for its derivative:

\[ \frac{dS}{dz} = S(z)(1 - S(z)) = S(z)S(-z) \]

Training the perceptron: least square

Regard the perceptron as a parametric function from an input space \(X\) to an output space \(Y\):

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:

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

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

The perceptron is a function \(f(x; w, b)\) parametrized by a weight vector \(w\) and a bias \(b\).

The function:

1. Maps a vector \(x\) to a scalar score using the linear function \(\langle w, x \rangle + b\).
2. Transforms the score into a probability value by applying the sigmoid function \(S(z)\).

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 \(1\) and including \(b\) in \(w\).

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 \(y\) can be expressed as:
  \[ P(y_i | x_i, w) = \frac{f(x_i, w)^{y_i} (1 - f(x_i, w))^{1 - y_i}}{\text{denom}} \]
- 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.

Cross-entropy loss

Better than least square for classification problems

The sigmoid activation function

The perceptron as a parametric function

Perceptron = linear classifier + sigmoid
The activation function in the perceptron is a sigmoid:
\[ S(z) = \frac{1}{1 + \exp(-z)} \]

The sigmoid converts real scores the range \((-\infty, +\infty)\) into probability values in the range \((0, 1)\).

The sigmoid function

\[ S(z) = \begin{cases} 1 & z \geq 0 \\ \frac{1}{1 + e^{-z}} & \text{otherwise} \end{cases} \]

It has several remarkable properties, such as the range \((0, 1)\).
In 1959, Hubel & Wiesel (Nobel Prize in Physiology and Medicine in 1981) conducted seminal experiments on the visual cortex of mammals.

They discovered the existence of neurons that respond to specific orientations and locations in the retina.

These neurons form a local and (statistically) translation invariant image operator.
Variables in CNNs are usually **tensors**, i.e. multi-dimensional array.

Conventionally, the dimensions are $N \times C \times U_1 \times \ldots \times U_D$ where
- $N$ is the **batch size**, i.e. the number of data samples represented by the tensor.
- $C$ is the number of channels.
- $U_1 \times \ldots \times U_D$ are the **spatial dimensions**.

The number of spatial dimensions $D$ can vary:
- $D = 2$ is used to represent 2D data such as images.
- $D = 3$ is used to represent 3D data such as volumes.

In general, it is possible to assign any meaning to the dimensions (e.g. time), as required by the application.

A **color image** can be interpreted as a tensor with $C = 3$ (color) channels, one for each of the R, G, and B color components.

More in general, any $C \times H \times W$ tensor can be interpreted as a $H \times W$ **field** of $C$-dimensional feature vectors.

The meaning of the feature channels is often not obvious.

---

**Tensor indexing**

Tensor elements $x_{ncu}$ are identified via indexes, one for each dimension:
- $n$ is the sample index in the batch
- $c$ is the feature channel index
- $u$ is the spatial index

The spatial index $u$ is in fact a **multi-index**, a shorthand notation for $u = (u_1, \ldots, u_D)$

Indexes are **0-based**:
- $0 \leq n < N$
- $0 \leq c < C$
- $0 \leq u < U = (U_1, \ldots, U_D)$

Generally, whenever you see a spatial multi-index, just pretend there is only one spatial dimension ($D = 1$). The extension to $D > 1$ is almost always trivial.

**Linear convolution**

A linear filter $f$ computes the weighted summation of a window of the input tensor $x$.

Key properties:
- **Linearity**: the operation is linear in the input and the filter parameters.
- **Locality**: the operator looks at a small window of data at a time.
- **Translation invariance**: all windows are processed using the same filter weights.
A linear filter $f$ computes the weighted summation of a window of the input tensor $x$.

**Key properties:**
- **Linearity:** the operation is linear in the input and the filter parameters.
- **Locality:** the operator looks at a small window of data at a time.
- **Translation invariance:** all windows are processed using the same filter weights.

The filter has one channel for each input tensor channel.

**As a neural network operator**

A convolutional layer is an operator that takes an input tensor $x$, a filter bank $f$, and a bias vector $b$ and produces as output a new tensor $y$.

**Dimensions:**
- The batch size $N$ is the same for input and output.
- Input and filters have the same number of channels $C$.
- The number of output channels $K$ is the same as the number of filters in the bank.
- The output dimension $O$ is given by $O = I - F + 1$.

Recall that $O = (O_1, O_2)$, $F = (F_1, F_2)$, and $I = (I_1, I_2)$ as we are using the multi-index shorthand.

**Padding and downsampling**

Padding extends a tensor $x$ with a border $P$ filled with zeros.

Downsampling retain one every $S$ pixels in a tensor, where $S$ is called the stride.

Padding and downsampling can be interpreted as additional layers before and after standard convolution:
Activation functions

Activation functions are scalar non-linear functions $S(z)$ that are applied element-wise to an input tensor $x$ to generate an output tensor $y$ (with the same dimensions).

$S(n_{cv}) = S(z_{ncu}) = \begin{cases} 
0 & \text{ReLU,} \\
\log(1 + e^z) & \text{soft ReLU,} \\
\epsilon z + (1 - \epsilon) \max\{0, z\} & \text{leaky ReLU,} \\
(1 + e^{-z})^{-1} & \text{sigmoid,} \\
\tanh(z) & \text{hyperbolic tangent,} 
\end{cases}$

Parameter-less non-linear filters

Pooling

The max pooling operator is similar to linear filter, operating transitively on $F = (F_1, F_2)$ sized windows.

$y_{ncv} = \max_{0 \leq u < F_1} x_{ncu}$

CNN layers summary

Deep convolutional neural networks

A deep convolutional neural network is a chain of several layers.

The typical pattern is to alternate linear convolution and non-linear activation, usually ReLU.

The other typical pattern is to gradually reduce the spatial resolution (via downsampling) and increase the number of feature channels.

Max-pooling is often used, in combination with downsampling, to reduce resolution further.
AlexNet: a CNN for image classification

The perceptron

Evaluation

Convolutional networks

Learning via SGD

Learning a CNN

The output is a 1000 \times 1 \times 1 tensor.

Each entry represents the score for the hypothesis that the image contains one out of a 1000 possible classes (defined in ImageNet).

Class scores are converted into probabilities by using the softmax layer (multi-class generalization of the sigmoid).

Class scores

Softmax

\[ y_c = \frac{e^{y_c}}{\sum_{c=0}^{C-1} e^{y_c}} \]

The goal is to optimize this energy over the model parameters \( w \).

Given a dataset \((x_1, y_1), (x_2, y_2), \ldots, (x_N, y_N)\), the total error is obtained by averaging the cross-entropy loss.

\[ E(w) = \frac{1}{N} \sum_{i=1}^{N} E_i(w), \quad E_i(w) = \ell(c_i, \Phi(x_i)) \]

\[ w^* = \arg\min_w E(w) \]
A CNN classifier has millions of parameters. Hence, learning requires massive amounts of data.

ImageNet is a large collection of labelled images. The standard subset (ILSVRC12) contains:
- 1,000 object classes
- ~1,000 example images for each class
- 1.2M training images in total

Without ImageNet (or a similar dataset) it would have been impossible to develop modern deep neural networks for computer vision.

The objective function is an average over $N = 1.2$M data points, and so is the gradient. The cost of a single gradient descent update is too large to be practical.

Stochastic gradient

Approximate the gradient by sampling a single data point (or a small batch of size $N' \ll N$). Perform the gradient update using the approximation.

Momentum

SGD can be accelerated by denoising the gradient estimate using a moving average. This average is called momentum.

Epochs & mini-batches

In practice, the data is visited not randomly, but in random order (without repetitions). A full pass is called an epoch.

Gradients are estimated by averaging mini-batches of 10-1000 examples. This takes advantage of parallel hardware such as GPUs.

Annealing schedule

The learning rate $\eta_t$ is gradually reduced over time, usually by a factor 10 when no progress is observed.

This allows SGD to slow down and more accurately land on an optimum as the latter is approached.

Time required

On a fast GPU, it is possible to process ~1k images per second for AlexNet.

An epoch thus lasts for 20 minutes. 40-100 epochs are required, requiring 13-33 hours (faster training requires tricks such as batch normalization).

On a CPU, this could be 100 x slower (four months).

Some networks are much slower (10 - 50 x).
Evaluating deep networks

General approach
Evaluation is not dissimilar to any other machine learning method, such as SVMs or the perceptron. Evaluation must always be done on a held-out validation or test set. This is because we need to test generalization, not just model fitting. Most benchmarks provide validation data for this purpose. Evaluation can use the same loss used for training. However, it is not uncommon to evaluate with respect to other, more meaningful losses as well.

Top-k error
For classification problems, there are two popular losses.

**Classification error:** the percentage of incorrectly classified image in the validation set.

**Top-k error:** the percentage of images whose ground truth class is not contained in the top-k more likely classes according to the model.

The top-k error requires the network to estimate confidences. Top-1 is the same as the classification error.

\[
E(\Phi) = \frac{1}{|\mathcal{D}_{\text{validation}}|} \sum_{(x, c) \in \mathcal{D}_{\text{validation}}} \text{err}(\Phi(x), c)
\]

The need for gradients
In order to train a neural network we minimize the average prediction error

\[
\text{argmin}_{w_1, \ldots, w_8} E(w_1, \ldots, w_8)
\]

In order to do so, we require the gradients of the error with respect to all parameters:
An efficient algorithm to compute the gradients

![Diagram of a neural network with backpropagation](image)

Backpropagation

Chain rule: scalar version

vec operator

The *vec operator* rearranges the elements of a tensor as a column vector, unrolling the tensor dimensions.

The order of unrolling is not essential, but a consistent convention must be used. PyTorch uses the *row major* convention.

By reshaping tensors in this manner, a tensor layer $y = f(x)$ can be thought of as a vector layer $\text{vec } y = f(\text{vec } x)$. 

- vec operator rearranges the elements of a tensor as a column vector, unrolling the tensor dimensions.
- The order of unrolling is not essential, but a consistent convention must be used. PyTorch uses the *row major* convention.
- By reshaping tensors in this manner, a tensor layer $y = f(x)$ can be thought of as a vector layer $\text{vec } y = f(\text{vec } x)$. 

An application of the vec operator is to allow using the standard notion of Jacobian matrix as derivative of a tensor function:

1. Using vec converts a tensor function $y = f(x)$ to a vector function $\text{vec } x = \text{vec } f(x)$
2. The derivative of a vector function is the Jacobian matrix.
3. The Jacobian matrix contains the derivative of each element of the output vector vec $y$ with respect to each element of the input vector vec $x$.

The (unbearable) size of tensor derivatives

The size of these Jacobian matrices is huge. Example:

Scalar

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

Unless the output is a scalar

Now the Jacobian reduces to a gradient and has the same size as $x$. Example:
Assume that $x_n$ is a scalar (e.g. loss)

Backpropagation

113

$\frac{d}{dx}f_n$ $\times$ $\frac{d}{dx}f_{n-1}$ $\times$ $\ldots$ $\times$ $\frac{d}{dx}f_2$ $\times$ $\frac{d}{dx}f_1$ $\times$ $\frac{d}{dx}x_0$

$p_{n-1}$ $\times$ $\ldots$ $\times$ $\frac{d}{dx}f_1$ $\times$ $\frac{d}{dx}x_0$

compute this first!

small
too large

Backpropagation

114

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

$\frac{d}{dx}(f_{n-1} \circ f_n)$ $\times$ $\ldots$ $\times$ $\frac{d}{dx}f_2$ $\times$ $\frac{d}{dx}x_0$

$p_{n-2}$ $\times$$\ldots$ $\times$ $\frac{d}{dx}f_1$ $\times$ $\frac{d}{dx}x_0$

small
too large

Backpropagation

115

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

$\frac{d}{dx}f_n$ $\times$ $\ldots$ $\times$ $\frac{d}{dx}f_2$ $\times$ $\frac{d}{dx}f_1$ $\times$ $\frac{d}{dx}x_0$

$p_{n-2}$ $\times$$\ldots$ $\times$ $\frac{d}{dx}f_1$ $\times$ $\frac{d}{dx}x_0$

small
too large

Backpropagation

116

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

$\frac{d}{dx}(f_{n-1} \circ f_n)$ $\times$ $\ldots$ $\times$ $\frac{d}{dx}f_2$ $\times$ $\frac{d}{dx}x_0$

$p_{n-2}$ $\times$$\ldots$ $\times$ $\frac{d}{dx}f_1$ $\times$ $\frac{d}{dx}x_0$

small
The function as a layer
The forward function evaluates a layer as normal.
The backward function computes the product of the Jacobian matrix evaluated at \( x \).

The backward function is a sort of “reverse layer”.

\( f^{BP} \) also be interpreted as computing the gradient of the projected function \( h(x) = (p, f(x)) \):

\[
\mathbf{p}' = \mathbf{p} \cdot \frac{d \text{vec } f}{d \text{vec } x}
\]

Note that \( \mathbf{p}' = f^{BP}(\mathbf{x}, \mathbf{p}) \) is a function of two arguments.

BP-reverse function \( f^{BP} \)

The key step is the calculation of the vector-Jacobian product

\[
\mathbf{p}' = \mathbf{p} \cdot \frac{d \text{vec } f}{d \text{vec } x}
\]

The result \( \mathbf{p}' \) is a vector that has the same size as \( \mathbf{x} \), so not too large.

However, the Jacobian matrix is still too large to explicitly compute.

Thus, we provide ad-hoc reverse functions \( f^{BP} \) for each layer that directly computes the product without computing the Jacobian explicitly.

This is possible by exploiting layer-specific optimizations.

Example calculation

Backpropagation for the sigmoid layer

Assume that \( \mathbf{x} \) is a vector (otherwise use vec).

Let \( y = f(x) \) be the sigmoid activation layer:

\[
f(x) = \begin{bmatrix}
\sigma(x_1) \\
\sigma(x_2) \\
\vdots \\
\sigma(x_C)
\end{bmatrix}, \quad \sigma(x) = \frac{e^x}{e^x + e^{-x}}.
\]

The Jacobian is then given by:

\[
\sigma' = \begin{bmatrix}
\sigma'(x_1) & 0 & \cdots & 0 \\
0 & \sigma'(x_2) & \cdots & 0 \\
\vdots & \vdots & \ddots & \vdots \\
0 & 0 & \cdots & \sigma'(x_C)
\end{bmatrix}
\]

The output of \( f^{BP} \) is then given by

\[
\mathbf{p} = \frac{df}{dx} = \begin{bmatrix}
p_1\sigma(x_1) & p_2\sigma(x_2) & \cdots & p_C\sigma(x_C)
\end{bmatrix}.
\]

The output of \( f^{BP} \) is the product of the Jacobian matrix evaluated at \( \mathbf{x} \).

Interpretation of the \( f^{BP} \) function

Computing gradients

So what are these vectors \( \mathbf{p} \) computed in the backward pass?

Each \( \mathbf{p} \) is the gradient of the scalar network output \( z \) with respect to the corresponding variable \( \mathbf{x} \).

When we want to emphasize this fact we use the following notation for such gradients:

\[
\mathbf{p} = \frac{dz}{dx} \quad \text{or even just} \quad \mathbf{p} = d\mathbf{x}
\]
Each layer provides a forward and backward function

For each fundamental layer, deep learning toolboxes provide:

- A forward function implementation.
- A backward function implementation.

With AutoDiff (see later) the invocation of the backward function is generally implicit (automatic).

However, you may need to code one to implement new layers or to optimize them.

The compute graph is a mechanism to keep track of the calculations in a program.

It can be used to automatically deduce which computations are required to compute the gradients.

These computations can then be added to the graph and the process repeated to obtain higher-order derivatives.

Modern machine learning toolboxes provide AutoDiff.

This means that calculations can be performed as normal in a programming language.

Underneath, the toolbox builds a compute graph.

Eventually, gradients can be requested.
C18 Machine Vision and Robotics

Computer Vision

Lecture 4: Applications

Dr Andrea Vedaldi
4 lectures, Michaelmas Term

For lecture notes, tutorial sheets, and updates see
http://www.robots.ox.ac.uk/~vedaldi/teach.html

Semantic image segmentation
Label individual pixels

Face analysis
Detection, verification, recognition, emotion, 3D fitting

Text spotting
Detection, word recognition, character recognition

E.g. VGG-Face

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

Object detection
Extract individual object instances

Architectures
Segmentation
Detection
Tracking

Neural network architectures
Evolution
AlexNet (2012)

5 convolutional layers
3 fully-connected layers
Evolution of Neural Network Architectures

133

|-----------|----------------|---------------|-------------------|

134

|-----------|----------------|---------------|-------------------|------------------|

135

|-----------|----------------|---------------|-------------------|------------------|-------------------|-----------------|

136

|-----------|-------------------|-------------------|---------------|----------------|------------------------|------------------------|-------------------------|


### Accuracy

3 x more accurate in 3 years

### Speed

5 x slower

**Remark:** 101 ResNet layers same size/speed as 16 VGG-VD layers

**Reason:** far fewer feature channels (quadratic speed/space gain)

**Moral:** optimize your architecture

### Model size

Num. of parameters is about the same

**Remark:** 101 ResNet layers same size/speed as 16 VGG-VD layers

**Reason:** far fewer feature channels (quadratic speed/space gain)

**Moral:** optimize your architecture
Semantic image segmentation

Label individual pixels

![Image segmentation examples]

Receptive field

The part of the image looked at by a neuron

**Receptive Field (RF) of a neuron**
- The subset of the image affecting the value of a neuron

**Small vs large RFs**
- Small RF: spatially specific, but can only account for small visual structures
- Large RF: spatially a-specific, but can account for large visual structure

**How to make the RF large**
- Use large filters
- Chain several filters
- Interleave downsampling along the chain
  - E.g. downsampling 2x increases the RF size 2x.

Convolutional vs fully connected layers

Comparing the receptive fields

<table>
<thead>
<tr>
<th>Convolutional layers</th>
<th>Fully connected layers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Neurons are spatially selective, can be used to localize things.</td>
<td>Neurons are global, do not characterize well position.</td>
</tr>
</tbody>
</table>

Which one is more useful for pixel level labelling?

A fully connected layer is just a large filter

The filter support fills the entire input tensor
Fully-convolutional neural networks

Dense evaluation
- Apply the whole network convolutional
- Computes a vector of class probabilities at each pixel

Downsampling
- For efficiency, the input data is substantially down sampled in the network
- The output is fairly low resolution (e.g. 1/32 of original)

The object detection problem
The goal of object detection is to simultaneously classify, enumerate, and localise known object types in an image.

A key challenge is that the number of object instances is not known a priori.
CNNs compute a fixed number of image features. A new computational mechanism is needed in order to detect a variable number of objects.

Region-based CNN (R-CNN) use a region proposal algorithm to extract a large number of potential object regions, and then a CNN to assess each one of them.

A region proposal algorithm produces a shortlist of regions that are likely to contain whole objects.

The Selective Search method by [van de Sande, Uijlings et al.]:
- Uses hierarchical segmentation based on colour uniformity and image edges.
- Produces about ~ 2000 regions / image with a > 95% probability of hitting any relevant object in the image.
Run an SVM or similar on top

Classification of a region

The feature vector is then classified by means of a linear predictor (or a multi-layer perceptron). There are \( C + 1 \) possible object types, including “no object” (background).

Region adjustment

Bounding-box regression

A second linear regression is used to refine the bounding box location. In the example, the person’s legs were not included in the proposal, but regression can fix this mistake.

Positive and negative training regions

Based on overlap with ground truth bounding box

At the time of introduction (2013)

Despite its conceptual simplicity, at the time of introduction R-CNN was substantially better than all existing methods.

This is due to the power of the CNN classifier.

Importantly, the CNN is pre-trained on the ImageNet data (1M images) for classification (using only image-level labels), then fine-tuned on PASCAL VOC data (5K images) for object detection (using region-level labels).

R-CNN results on PASCAL VOC

<table>
<thead>
<tr>
<th>Method</th>
<th>VOC 2007</th>
<th>VOC 2010</th>
</tr>
</thead>
<tbody>
<tr>
<td>DPM v5 (Girshick et al. 2011)</td>
<td>33.7%</td>
<td>29.6%</td>
</tr>
<tr>
<td>UVA sel. search (Uijlings et al. 2013)</td>
<td>35.1%</td>
<td></td>
</tr>
<tr>
<td>Regionlets (Wang et al. 2013)</td>
<td>41.7%</td>
<td>39.7%</td>
</tr>
<tr>
<td>SegDPM (Fidler et al. 2013)</td>
<td>40.4%</td>
<td></td>
</tr>
<tr>
<td>R-CNN (TorontoNet)</td>
<td>54.2%</td>
<td>50.2%</td>
</tr>
<tr>
<td>R-CNN (TorontoNet) + bbox regression</td>
<td>58.5%</td>
<td>53.7%</td>
</tr>
<tr>
<td>R-CNN (VGG-VD)</td>
<td>62.1%</td>
<td></td>
</tr>
<tr>
<td>R-CNN (ONet) + bbox regression</td>
<td>66.0%</td>
<td>62.9%</td>
</tr>
</tbody>
</table>
Integrate more of the blocks as CNN components

R-CNNs as a complex CNN

Accelerating R-CNN

The Spatial Pooling (SP) layer

The SP layer extracts a feature vector for each of the \( R \) regions. The output are thus \( R \) tensor of size \( 1 \times 1 \times C \).

Alternatively, this can be seen as a single \( 1 \times 1 \times C \times R \) tensor.
The Spatially Pyramid Pooling Layer

SP with multiple subdivisions

SPP is similar to SP, but pools features in the tiles of a grid-like subdivision of the region. The resulting feature vector captures the spatial layout of the original region.

Fast and Faster R-CNN performance

Both faster and better!

Detection mAP on PASCAL VOC 2007, with VGG-16 pre-trained on ImageNet.

<table>
<thead>
<tr>
<th>Method</th>
<th>Time / image</th>
<th>mAP (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-CNN</td>
<td>~50s</td>
<td>66.0</td>
</tr>
<tr>
<td>Fast R-CNN</td>
<td>~2s</td>
<td>66.9</td>
</tr>
<tr>
<td>Faster R-CNN</td>
<td>198ms</td>
<td>69.9</td>
</tr>
</tbody>
</table>

Example detections


Same parameters

Chair
Box refinement
Background
Box refinement
Potted plant
Box refinement
**PASCAL VOC Leaderboards**

Detection challenge (comp4: train on own data)

http://tinyurl.com/h7uzkov

2014 4 x improvement in accuracy 2016

**Tracking 1/2: select & track**

Draw a bounding box first, then track it automatically

**Tracking 2/2: detect & track**

Track pre-programmed objects (e.g. faces) fully automatically (no manual selection required)
Example of specific trackers

1. Bus
2. Cat
3. Bunny
4. Monkey
5. Plane
6. Bike
7. Fox
8. Car
9. Tiger

Tracking flavours

Select & track
Open ended, but requires manual input

Detect & track
Restricted to the object the program knows, but fully automatic

Track pretty much anything
Cheap to track something new, but still requires manual input

Typical applications: people, faces, cars
New objects can be learned, but at a cost

Open-ended tracking

Problem: Track an arbitrary object with the sole input of a single bounding box in the first frame of the video

Challenge: The tracker must be object-agnostic and learn what we mean from a single example

Tracking via iterated detection

Learn the object in one frame, seek it in the next

Repeat at times $t = 0, 1, 2, 3, ...$

- At frame $t$ learn a model of the object vs background
- At frame $t+1$ use the model to find the new object location

How our tracker works

Descriptor computation
A neural network $\phi$ maps each image window to a visual descriptor.

Two images of different sizes:
- small: exemplar at time $t$
- big: search area at time $t+1$

Descriptor matching
Computes the descriptor similarity at all translated sub-windows.

<table>
<thead>
<tr>
<th>ImageNet Video</th>
<th>Official task is object detection from video - can be easily adapted to arbitrary object tracking</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Almost 4,500 videos and 1,200,000 bounding boxes!</td>
</tr>
<tr>
<td></td>
<td>30 classes: mostly animals (~75%) and some vehicles (~25%)</td>
</tr>
</tbody>
</table>

C18 Computer Vision

Recap
Prof. Andrea Vedaldi (4 lectures)
- Lecture 1: Matching, indexing, and retrieval
- Lecture 2: Convolutional neural networks
- Lecture 3: Backpropagation and automated differentiation
- Lecture 4: Applications

Prof. Victor Prisacariu (4 lectures)
- 3D vision