The Internet: 50 billion images and counting...

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

.. but it likely contains a similar one!
Goal: search a large collection for an image of the same object

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

Define a similarity function between images:

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

Image similarity (I)

Compare images as vectors of pixels

\[ F(I_1, I_2) = -‖ I_1 - I_2 ‖^2 \]

Why do pixel values differ so much?

Nuisance factors

- Viewpoint
- Visibility
- Illumination
- Camera
- Noise

Matching and transformation

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

\[
\begin{bmatrix}
x' \\
y'
\end{bmatrix} = sR(\theta) \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \end{bmatrix} \quad R(\theta) = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix}
\]
Homography/affine transformations

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 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 ($\sim 10^6$).

Exhaustive matching is just too expensive.

We need a method to select a small subset of features to match.
Co-variant feature detectors

A detector is a rule that selects a small subset of features for matching. The key is co-vari ance: 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”.

Co-variant detector types

Similarity | Affine
--- | ---

Properties of a detector
- repeatability
- generality
- speed

Benefits of increased covariance
- handle more general motions / objects

Cons of increased covariance
- less robust
- slower

Discriminability and support

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

**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 0: get an image pair**

The left image has \( m \) features \((f_1, d_1), \ldots, (f_m, d_m)\)

Right image has \( n \) feature \((f'_1, d'_1), \ldots, (f'_n, d'_n)\)

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

number of matches: 0

The left image has \( m \) features \((f_1, d_1), \ldots, (f_m, d_m)\)

Right image has \( n \) feature \((f'_1, d'_1), \ldots, (f'_n, d'_n)\)
Matching all local features

**Step 2:** match each descriptor to its closest one

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

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

Matching all local features

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

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

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

Matching all local features

**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

Indexing using visual words

Global geometric verification

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. \( 10^2 \) (~SIFT detector)
- \( D \) dimensional feature descriptor: e.g. \( 10^2 \) (~SIFT descriptor)
- Exhaustive search cost: \( O(N^2 LD) \)
- Memory footprint: \( O(NLD) \)

10^11 - 10^10 ops = 100 days - 300 years

Exhaustive search cost: \( O(N^2 LD) \)

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

Inverted index

For each word, lists all documents containing it as pairs \( \langle \text{DocID, WordCount} \rangle \)

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
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:
   \[
   \pi(d_i) = \arg\min_{k=1,\ldots,K} \|d_i - \mu_k\|^2
   \]

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:

\[
\pi(d) = \arg\min_{i=1,\ldots,K} \|d - \mu_i\|^2
\]
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**

This is also called a **bag of visual words** because it does not remember the relative positions of the features, just the number of occurrences.

Hence, h **discards spatial information.**

**Pros:** more invariant to viewpoint changes and other nuisance factors.

**Cons:** less discriminative.

Comparing histograms

**Cosine similarity**

The 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.
By comparing bag-of-words descriptors

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

Image similarity (III)

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:

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

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.

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.
**Example benchmark: Oxford 5K**

http://www.robots.ox.ac.uk/~vgg/data/oxbuildings/

Dataset content
- ~ 5K images of Oxford
- An optional additional set of confounder (irrelevant) images
- 58 test queries

---

**Linear predictors**

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.

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

---

**Data representations**

**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

**Embedding Space** $\mathbb{R}^d$

**Vector Similarity (distance)**

**The perceptron**

An early neural network by Rosenblatt (1957)

**What**
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.

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

Learning via SGD

Convolutional networks

Evaluation
Training the perceptron: least square

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

$$y = S((w,x) + b)$$

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

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

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 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:

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

Cross-entropy loss

Better than least square for classification problems

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:
  $$P(y_i | x_i, w) = f(x_i, w)/(1 - f(x_i, w))^{1-y_i}$$
- 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.
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 neuron 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.

Tensor elements $x_{n,c,u}$ 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.

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.

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.

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** 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**.
### Activation functions

**The non-linearity in deep networks**

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).

\[
\begin{align*}
\text{Sigmoid} & : z = \frac{1}{1 + e^{-z}} \\
\text{Tanh} & : z = \frac{e^z - e^{-z}}{e^z + e^{-z}} \\
\text{ReLU} & : z = \max(0, z) \\
\text{Leaky ReLU} & : z = (1 - \epsilon) \max(0, z) + \epsilon z \\
\text{Soft ReLU} & : z = \log(1 + e^z) \\
\text{Sigmoid} & : z = \frac{1}{1 + e^{-z}} \\
\text{Hyperbolic tangent} & : z = \frac{e^z - e^{-z}}{e^z + e^{-z}}
\end{align*}
\]

### CNN layers summary

**Expression**

\[
y_{ncw} = \sum_{c=0}^{C-1} \sum_{u=0}^{F-1} f_{kc} \cdot x_{ncw} + b_k
\]

**Dimensions**

\[N \times C \times I \rightarrow N \times K \times O\]

### Pooling

**Parameter-less non-linear filters**

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

\[
y_{ncw} = \max_{0 \leq u < F} x_{n,v+w}
\]

Pooling can use other operators, for example **average**:

\[
y_{ncw} = \frac{1}{F_1 \cdot F_2} \sum_{0 \leq u < F} x_{n,v+w}
\]

### Deep convolutional neural networks

**A long sequence of layers**

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.
The perceptron

Learning via SGD

Convolutional networks

AlexNet: a CNN for image classification

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.

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

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

\[
w^* = \arg\min_w E(w)
\]
Learning a CNN

ImageNet benchmark data

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.

Stochastic gradient descent

The objective function is an average over $N = 1.2M$ data points, and so is the gradient. The cost of a single gradient descent update is way 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).
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.

In order to train a neural network we minimize the average prediction error

$$\arg\min_{w_1, \ldots, w_8} E(w_1, \ldots, w_8)$$

$$\nabla E = \left( \frac{dE}{dw_1}, \ldots, \frac{dE}{dw_8} \right)$$

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

Backpropagation

Chain rule: scalar version

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.

Reshaping tensors into vectors

A composition of $n$ functions

Derivative obtained using the chain rule

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

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 \( \text{vec } x = f(\text{vec } x) \)

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 \( \text{vec } y \)
with respect to each element of the input
vector \( \text{vec } x \).

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

114

115

116
The \( f \)BP function as a layer

The forward function \( f \) evaluates a layer \( y = f(x) \) as normal.

The \( f \)BP function also computes the product \( p' \) of a vector \( p \) with respect to the layer’s Jacobian matrix evaluated at \( x \).

The \( f \)BP function is a sort of “reverse layer”.

The \( f \)BP also be interpreted as computing the gradient of the projected function \( h(x) = p \cdot f(x) \). In fact:

\[
p \cdot \frac{df}{dx} = \frac{d(p \cdot f(x))}{dx} = p \cdot \frac{df}{dx}
\]

Note that \( p' = fBP(x, p) \) is a function of two arguments.

Example calculation

Backpropagation for the sigmoid layer

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

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

\[
f(x) = \begin{bmatrix}
\sigma(x_1) \\
\sigma(x_2) \\
\vdots \\
\sigma(x_C)
\end{bmatrix},
\sigma(x) = \frac{e^x}{e^x + 1}.
\]

The Jacobian is the diagonal matrix:

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

The output of \( fBP \) is then given by:

\[
p \cdot \frac{df}{dx} = \begin{bmatrix}
\sigma(x_1) & \sigma(x_1) & \cdots & \sigma(x_1) \\
\sigma(x_2) & \sigma(x_2) & \cdots & \sigma(x_2) \\
\vdots & \vdots & \ddots & \vdots \\
\sigma(x_C) & \sigma(x_C) & \cdots & \sigma(x_C)
\end{bmatrix}.
\]

Leveraging sparsity is the key to efficiency!

Interpretation of the \( f \)BP function

Computing gradients

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

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

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

\[
p = \frac{dz}{dx} \quad \text{or even just} \quad p = dx
\]
Implementation in machine learning toolboxes

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.

Compute graphs

Keeping track of calculations for automatic differentiation

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.

AutoDiff

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.

import torch

# Define two random inputs, both requiring grads
x0 = torch.randn(1, 10, 18, 18, requires_grad=True)
x1 = torch.randn(1, 3, 20, 20, requires_grad=True)

# Get a convolutional layer. Implicitly this contains a parameter tensor conv.weight with requires_grad=True
conv = torch.nn.Conv2d(3, 10, 3)

# Intermediate calculations
x2 = conv(x0)
x3 = torch.nn.ReLU()(x2) + x1

# Obtain a scalar output by summing everything
x4 = x3.sum()

# Invoke AutoGrad to compute gradients
x4.backward()

# Print gradient shapes (just to check)
print(x0.grad.shape) 
print(x1.grad.shape) 
print(conv.weight.grad.shape)
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

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/
Object detection

Extract individual object instances

Neural network architectures

Evolution
AlexNet (2012)

5 convolutional layers

3 fully-connected layers
<table>
<thead>
<tr>
<th>Evolution</th>
<th>Neural network architectures</th>
<th>133</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Evolution</th>
<th>Neural network architectures</th>
<th>134</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Evolution</th>
<th>Neural network architectures</th>
<th>135</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Evolution</th>
<th>Neural network architectures</th>
<th>136</th>
</tr>
</thead>
</table>

---


**Accuracy**

3 × more accurate in 3 years

**Speed**

5 × slower

**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

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.

Comparing the receptive fields
Convolutional vs fully connected layers

Convolutional layers
Neurons are spatially selective, can be used to localize things.

Fully connected layers
Neurons are global, do not characterize well position.

Which one is more useful for pixel level labelling?

The filter support fills the entire input tensor

A fully connected layer is just a large filter

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

class predictions


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

Downsampling
- For efficiency, the input data is substantially downsampled 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.

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.

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

A region proposal is slightly dilated to capture some visual context and then cropped and resized in order to be passed to a CNN.

The cropped and resize region is passed through a CNN to extract a corresponding feature vector (or image representation).
Run an SVM or similar on top

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.

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-CNNs as a complex CNN

Integrate more of the blocks as CNN components

R-CNN can be improved substantially in three ways:
- By integrating all blocks in an end-to-end trainable CNN
- By accelerating region-specific computations
- By replacing region proposal generation with something better

Accelerating R-CNN

Problem: The fundamental bottleneck is evaluating the CNN from scratch for each image region.

Solution: Compute all the convolutional features just once, and then crop directly the resulting feature map.

Only the fully-connected layers are evaluated for each region.

How: Spatial pool layer.

The Spatial Pooling (SP) layer

The SP layer extracts a feature vector for each of the $R$ regions.

The output are thus $R$ tensors 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>
PASCAL VOC Leaderboards

Detection challenge (comp4: train on own data)

http://tinyurl.com/h7uzkov

2014  2016
4 x improvement in accuracy

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

- bus
- cat
- bunny
- monkey
- plane
- bike
- fox
- car
- 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

At frame $t$

TRAINING

- learn a model of the object vs background

At frame $t+1$

TESTING

- use the model to find the new object location

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

Describe and match
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.

ImageNet Video
Training data

Official task is object detection from video - can be easily adapted to arbitrary object tracking.

Almost 4,500 videos and 1,200,000 bounding boxes!

30 classes: mostly animals (~75%) and some vehicles (~25%).

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