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.

**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

**The perceptron**

**Convolutional networks**

**Learning via SGD**

**Evaluation**
The perceptron

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.

Perceptron = linear classifier + sigmoid

The perceptron as a parametric function

The perceptron as a parametric function

The sigmoid activation function

The sigmoid activation function

The sigmoid activation function

The sigmoid activation function

The sigmoid activation function

The sigmoid activation function

The sigmoid activation function
Training the perceptron: least square

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

$$x \rightarrow y = S(\langle w, x \rangle + b)$$

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:

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

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

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:
  $$P(y_i | x_i, w) = \frac{\exp(\langle w, x \rangle + b)}{1 + \exp(\langle w, x \rangle + b)}$$

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

The empirical negative log-likelihood is obtained by averaging the negative log-likelihood over all the training data points:

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

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

Softmax layer

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

The vector of scores $x^c$ is mapped to a vector of probabilities $x^c$ using the **softmax** operator, which is a generalisation of the sigmoid.

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

$$x_1 = \frac{e^{x^1}}{e^{x^1} + e^{x^2}} = \frac{e^{x^1}}{e^{x^1} + e^{z = x^2 - x^1}} = \frac{1}{1 + e^{-z}} = S(\langle w, x \rangle + b)$$
The log-likelihood and objective function for a multi-class perceptron are given by:

\[ -\log P(y_i = c | x_i, W) = -\log \sum_{q=1}^{C} e^{w_i^T x_i + b_q} = -w_i^T x_i - b_c + \log \sum_{q=1}^{C} e^{w_i^T x_i + b_q} \]

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

\[ E(W) = \frac{1}{N} \sum_{i=1}^{N} \left( -w_i^T x_i - b_c + \log \sum_{q=1}^{C} e^{w_i^T x_i + b_q} \right) \]

Perceptrons can also be chained, resulting in a so-called \textit{deep neural network}. Depth refers to the fact that the function decomposes as a long ("deep") chain of simpler perception-like functions.

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.

Hubel and Wiesel 1959

The discovery of oriented cells in the visual cortex
Variables in CNNs are usually tensors, i.e. multi-dimensional array.

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

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.

Multiple input channels

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.

Multiple output channels and filter banks

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 bank of filters is used to generated multiple output channels, one per filter.
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.

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).

\[
S(x_{nc,v}) = b_k + \sum_{i=0}^{C-1} \sum_{u=0}^{F-1} f_{kci} \cdot x_{n,c,v+u}
\]

Pooling can use other operators, for example average:

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

Pooling can use other operators, for example average:
### CNN layers summary

<table>
<thead>
<tr>
<th>Layer</th>
<th>Expression</th>
<th>Dimensions</th>
</tr>
</thead>
<tbody>
<tr>
<td>CNN layers</td>
<td>$y_{akc} = b_k + \sum_{c=0}^{C-1} \sum_{u=0}^{F-1} f_{ku} \cdot x_{n,c,v+u}$</td>
<td>$O = I - F + 1$</td>
</tr>
<tr>
<td></td>
<td>$y_{mcu} = \max{0, x_{mcu}}$</td>
<td>$K = C$, $O = I$</td>
</tr>
<tr>
<td></td>
<td>$y_{muv} = \max{0, x_{muv}}$</td>
<td>$O = I - F + 1$</td>
</tr>
</tbody>
</table>

### 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

**A long sequence of layers**

<table>
<thead>
<tr>
<th>Layer</th>
<th>Input</th>
<th>Expression</th>
<th>Dimensions</th>
</tr>
</thead>
<tbody>
<tr>
<td>AlexNet</td>
<td>$3 \times 244 \times 244$</td>
<td>$1000 \times 1 \times 1$</td>
<td>vector of C scores</td>
</tr>
<tr>
<td></td>
<td>$64 \times 27 \times 27$</td>
<td>$256 \times 23 \times 23$</td>
<td>$384 \times 13 \times 13$</td>
</tr>
</tbody>
</table>

### AlexNet contains 8 blocks, each formed by:
- A linear convolution operator (with padding/downsampling)
- A ReLU operator (except for $f_8$)
- An optional max pooling operator (with padding/downsampling)

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

$$p_i = \frac{e^{y_i}}{\sum_{j=1}^{K} e^{y_j}}$$
Learning a CNN

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).

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 $\text{err}$ 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.

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.

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

An efficient algorithm to compute the gradients is backpropagation.

The chain rule in scalar version is:

$$\nabla E = \left( \frac{dE}{dw_1}, \ldots, \frac{dE}{dw_8} \right)$$
Chain rule: scalar version

A composition of $n$ functions

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

Derivative obtained using the chain rule

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

Derivative of tensor-valued functions

Using vec() and matrix notation

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:

$$\text{vec} \begin{bmatrix} y_{00} & y_{01} \\ y_{10} & y_{11} \end{bmatrix} = \begin{bmatrix} y_{00} \\ y_{01} \\ y_{10} \\ y_{11} \end{bmatrix}$$

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

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 $\text{vec} \; x = \text{vec} \; f(\text{vec} \; x)$
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 $\text{vec} \; x$. 

### Chain rule: tensor version

Using vec() and matrix notation
The (unbearable) size of tensor derivatives

The size of these Jacobian matrices is huge. Example:

\[ \frac{d \text{vec } f}{d \text{vec } x} \]

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 reduces to a gradient and has the same size as \( x \). Example:

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

Backpropagation

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

Backpropagation

$\begin{align*}
& x_n, f_n, x_{n-1}, \ldots, f_2, x_1, f_1, x_0 \\
& \frac{d \text{vec} f_n \cdots f_1}{d \text{vec} x_1} \times \frac{d \text{vec} f_1}{d \text{vec} x_0} \\
& p_{n-2} \times \text{small} \quad \text{too large}
\end{align*}$

Key reduction operation

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

$\begin{align*}
p' &= p \cdot \frac{d \text{vec} f}{d \text{vec} x}
\end{align*}$

The result $p'$ is a vector that has the same size as $x$, 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.

Interpretation of the $f^{BP}$ function

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

The backward function $f^{BP}$ computes the product $p'$ of a vector $p$ with respect to the layer Jacobian matrix evaluated at $x$.

The backward function can be interpreted as:

- Forming the projected function $h(x) = \langle p, f(x) \rangle$. This function has a scalar output.
- Computing the derivative (aka gradient) of this projected function.

Proof: use the fact that the derivative operator is linear and hence can be swapped with the inner product.

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

\[
p = \frac{dz}{dx} \quad \text{or even just} \quad p = dx
\]

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 many 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.

The graph is more commonly shown the other way around, with the forward direction left to right.
**Backpropagation network**

**Conv, ReLU, MP and their transposed blocks**

**forward**

- $x_0$ → $x_1$ → $x_2$ → $x_3$ → ...

**backward**

- $\delta_0$ → $\delta_1$ → $\delta_2$ → $\delta_3$ → ...

**Conv, ReLU, MP and their transposed blocks**

- $x_0$ → $x_1$ → $x_2$ → $x_3$ → ...

**backward**

- $\delta_0$ → $\delta_1$ → $\delta_2$ → $\delta_3$ → ...

**Sufficient statistics and bottlenecks**

**forward**

- $x_0$ → $x_1$ → $x_2$ → $x_3$ → ...

**backward**

- $\delta_0$ → $\delta_1$ → $\delta_2$ → $\delta_3$ → ...

**AutoDiff**

**Automatic backpropagation**

```
import torch

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

**AIMS Big Data Course**

**Introduction to deep learning**

**Part 3: Applications**
Semantic image segmentation
Label individual pixels

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

Detection, verification, recognition, emotion, 3D fitting

Text spotting
Detection, word recognition, character recognition

Object detection
Extract individual object instances

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
AlexNet (2012)
5 convolutional layers
3 fully-connected layers

AlexNet (2012)
VGG-M (2013)
VGG-VD-16 (2014)
GoogLeNet (2014)
Neural network architectures

**Evolution**

- **AlexNet** (2012)
- **VGG-M** (2013)
- **VGG-VD-16** (2014)
- **GoogLeNet** (2014)

16 convolutional layers
50 convolutional layers
152 convolutional layers

**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


