Machine Learning
Deep Learning

dsai.asia

Asian Data Science and Artificial Intelligence Master's Program

Co-funded by the
Erasmus+ Programme
of the European Union

Readings

Readings for these lecture notes:

- Goodfellow, I., Bengio, Y., and Courville, A. (2016), *Deep Learning*, MIT Press, Chapter 6.
- Bishop, C. (2006), *Pattern Recognition and Machine Learning*, Springer, Chapters 3, 4, 6, 7.
- Hastie, T., Tibshirani, R., and Friedman, J. (2016), *Elements of Statistical Learning: Data Mining, Inference, and Prediction*, Springer, Chapters 2, 3, 4, 12.
- Ng, A. (2017), *Deep Learning*, Lecture note set for CS229, Stanford University.

These notes contain material © Bishop (2006), Hastie et al. (2016), Goodfellow et al. (2016), and Ng (2017).

Outline

Introduction

Deep learning involves the training of so-called neural networks.

In this unit, we'll study how neural networks work and understand basic neural network learning algorithms.

Then we'll briefly cover some of the modern neural network architectures that are generating so much layperson interest in AI today.

Outline

To begin with, we'll consider the univariate regression setting where we want to learn a function $f : x \mapsto y$.

A simple neural network can represent $f(x)$ by a single neuron or unit that computes

$$f(x) = \max(ax + b, 0)$$

for some fixed coefficients a and b.

This particular unit is called a ReLU (rectified linear unit).

Neural network intuition

Composing units into networks

By stacking such units so that one passes its output to another, we can model increasingly complex functions.

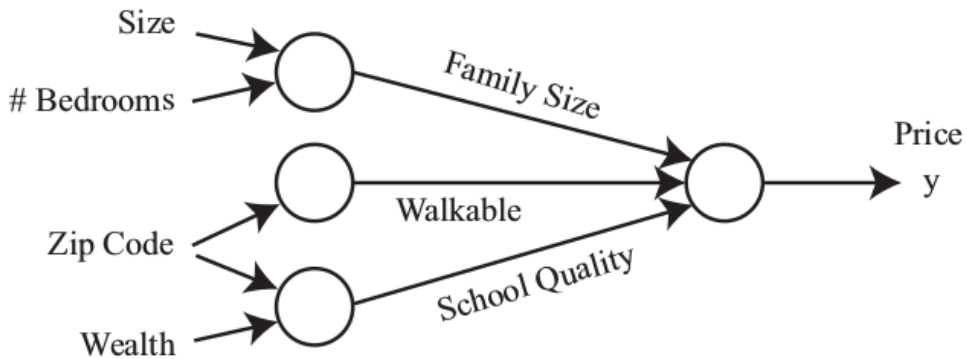
From Ng's course notes, consider the example in which we want to predict house price given size, number of bedrooms, postal code, and wealth of the neighborhood the house is in.

We could build the model up incrementally, having it compute a "family size" variable based on the house size and number of bedrooms.

The postal code could be used to compute how "walkable" the neighborhood is.

Combining the zip code with neighborhood wealth might predict "school quality."

We might finally decide that the price depends on these three derived features: family size, walkable, and school quality:



Ng (2017), CS229 deep learning lecture notes.

Neural network intuition

Composing units into networks

Another example: loan application underwriting.

One important attribute of a loan applicant is his/her income.

But high income is meaningless if the applicant's debt is very high.

Loan underwriters combine these two features into a higher level feature, debt-to-income ratio.

43% is considered a "magic" tolerable debt-to-income ratio for a family.

See any problems with this architecture as we described it so far?

Luckily, we don't need to solve those problems, as neural network learning is end-to-end learning.

That means the network figures out for itself what intermediate features are best for the task at hand.

Neural network intuition

Hidden units

Intermediate units between the raw inputs and the output are called hidden units.

Example suppose we have:

- Four inputs x_1, x_2, x_3, and x_4
- Three hidden units
- A single output y

The goal of the network will be to find intermediate features that will best predict each $y^{(i)}$ from the corresponding $x^{(i)}$.

It may be difficult to understand the "meaning" of the intermediate features thus induced.

Neural networks are therefore some times called black boxes.

Here are the terms we've seen so far, and some new ones:

- A neuron or unit applies some function to its inputs to generate an output.
- Units may be composed into neural networks.
- Input features are sometimes represented by units called input units organized into a input layer.
- One or more outputs comprise the output layer.
- Intermediate units are called hidden units and may be organized into zero or more hidden layers.

Suppose we have an input layer composed of features $x_1, x_2, \ldots$

Ng uses the notation $a_i^{[j]}$ to indicate the activation of the ith unit in the jth layer.

$a_1^{[1]}$ is the output of the first hidden unit in the first hidden layer.

$a_1^{[2]}$ is the output of the first unit in the second layer (the output layer in a network with only one hidden layer).

To unify the notation, we let $a_i^{[0]} = x_i$, i.e., we treat the input as layer 0.

We saw a simple ReLU network already.

Logistic regression can also be treated as a simple neural network with one output unit and no hidden units:

$$g(\mathrm{x}) = \frac{1}{1 + e^{-\mathrm{w}^\top \mathrm{x}}}$$

is written in standard neural network notation as two steps:

1. Calculate the linear response $z = \mathrm{w}^\top \mathrm{x} + b$.
2. Calculate the activation function $a = \sigma(z)$ where $\sigma(z) = 1/(1 + e^{-z})$.

Usually, $g(z)$ is nonlinear. The most common activation functions:

- The sigmoid $g(z) = \frac{1}{1+e^{-z}}$
- ReLU (the default activation function in modern neural networks) $g(z) = \max(z, 0)$
- Hyperbolic tangent $g(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$

The full calculation proceeds as follows.

For the first hidden unit in the first hidden layer, we calculate

$$z_1^{[1]} = \mathtt{W}_1^{[1]\top} \mathrm{x} + b_1^{[1]}$$

and

$$a_1^{[1]} = g(z_1^{[1]}),$$

where $\mathtt{W}$ is a matrix or parameters or weights.

We repeat for each unit in each layer to get the final output layer.

Neural network intuition

What remains?

Now that we understand the feed-forward computation of a neural network, we'll talk about

- Efficient execution of the feed-forward computation
- The gradient descent process used to learn the weights `W`.

Outline

Efficient computation

Calculating activations

Consider calculating hidden unit activations. We have

$$\begin{aligned} z_1^{[1]} &= \mathtt{W}_1^{[1]\top} \mathrm{x} + b_1^{[1]} \quad \text{and} \quad a_1^{[1]} = g(z_1^{[1]}) \\ &\vdots \qquad\qquad\qquad \vdots \qquad\qquad \vdots \\ z_4^{[1]} &= \mathtt{W}_4^{[1]\top} \mathrm{x} + b_4^{[1]} \quad \text{and} \quad a_4^{[1]} = g(z_4^{[1]}) \end{aligned}$$

Depending on the "deepness" of our model, for a single input, we may be doing an operation like this for hundreds or thousands of units.

Code to implement this procedure using `for` loops and the like will run very slowly, especially if implemented in a bytecode based language like Python or Java.

Efficient computation

BLAS library

What is needed is the ability to perform matrix algebra with a single library call or instruction that is highly optimized, using CPU instructions provided for vector operations.

BLAS is a well-known library that does this and is used by numpy:

```
ldd /usr/lib/python3/dist-packages/numpy/core/multiarray.cpython-35m-x86_64-linux-gnu.so
linux-vdso.so.1 =>  (0x00007fff40fde000)
libblas.so.3 => /usr/lib/libblas.so.3 (0x00007f9a14579000)
libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6 (0x00007f9a14270000)
libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0 (0x00007f9a14053000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f9a13c89000)
/lib64/ld-linux-x86-64.so.2 (0x00007f9a14b69000)
libopenblas.so.0 => /usr/lib/libopenblas.so.0 (0x00007f9a11bf5000)
libgfortran.so.3 => /usr/lib/x86_64-linux-gnu/libgfortran.so.3 (0x00007f9a118ca000)
libquadmath.so.0 => /usr/lib/x86_64-linux-gnu/libquadmath.so.0 (0x00007f9a1168b000)
libgcc_s.so.1 => /lib/x86_64-linux-gnu/libgcc_s.so.1 (0x00007f9a11475000)
```

Alternatively, we may parallelize computation by recruiting GPU resources.

Efficient computation

Vectorized activation calculation

For an entire layer, to use vectorized computation, we need to perform the operation in one operation:

$$z^{[1]} = W^{[1]\top} x + b^{[1]}$$

This can be implemented in a single Python statement:

```
W1 = np.matrix(np.random.normal(0,1,(3,4)))
b1 = np.matrix(np.random.normal(0,1,(4,1)))
x = np.matrix(np.random.normal(0,1,(3,1)))
z1 = W1.T * x + b1
```

This would run much faster than the equivalent doubly-nested `for` loop.

Efficient computation

Vectorized activation calculation

Then to calculate $a^{[1]}$ as a vector operation, we can hopefully use vectorized functions in our implementation language.

If we have the sigmoid function, for example, we implement

$$g(z_i^{[1]}) = \frac{1}{1 + e^{-z_i^{[1]}}}$$

as

```
def g(z):
    return 1 / (1 + exp(-z))
z = np.matrix([[1,2,3]]).T
a = g(z)
```

This will run much faster than executing the `exp()` function within a Python loop.

Efficient computation

Vectorizing over training examples

Next, consider performing a calculation over many training examples.

Performing the operation

$$z^{[1]} = W^{[1]\top} x^{(i)} + b^{[1]}$$

inside a loop for every training example i would be slower than the vectorized operation

$$z^{[1]} = W^{[1]\top} X^{\top} + b^{[1]}.$$

Note: despite different dimensions of $W^{[1]\top} X^{\top}$ and $b^{[1]}$, some languages like Python allow broadcasting of the addition operation horizontally:

```
>>> W = np.matrix([[1,2,3],[2,3,4]])
>>> b = np.matrix([[2,3]]).T
>>> W+b
matrix([[3, 4, 5],
        [5, 6, 7]])
```

Outline

Backpropagation

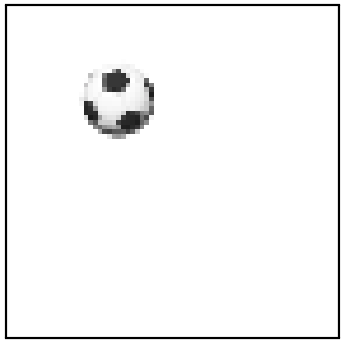
Introduction

Now that we understand how the feedforward computation of a neural network works and how to use optimized vector operations, let's consider how to train a neural network.

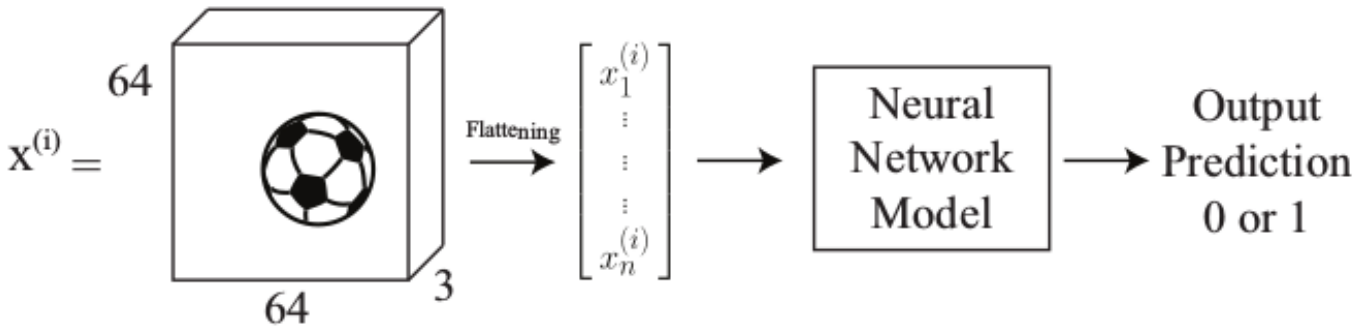
The general procedure is called backpropagation.

We'll use Ng's (2017) example of classifying an image as containing a soccer ball or not.

First we'll scale the image to a standard size, for example 64 $\times$ 64.



Then we'll flatten the 64 $\times$ 64 $\times$ 3 elements of the input to a 12,288-element vector and present to our neural network:



Ng (2017), CS 229 Lecture notes on deep learning

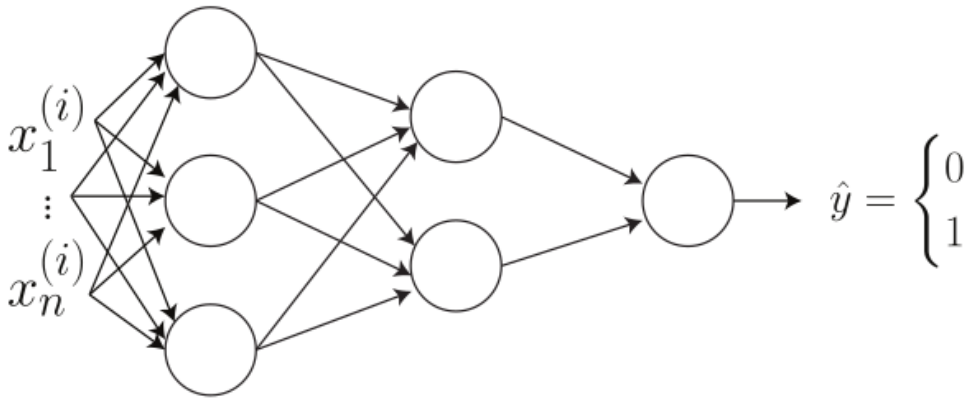
Backpropagation

Model, architecture, parameters

Some terminology a neural network model consists of

- The network architecture (number of layers, units, type of units, connectivity between units),
- The parameters (the values of the weights $\mathrm{W}^{[i]}$ and $\mathrm{b}^{[i]}$.

In the forthcoming analysis, we'll assume a 3-layer network architecture



Ng (2017), Deep learning lecture notes for CS 229.

The architecture consists of two fully connected layers followed by a single logistic sigmoid output. Now we consider how to learn the $3n + 14$ parameters of the model through backpropagation.

First step: set initial values of the parameters.

Initializing to 0 would be a bad idea, because the output of each layer would be identical for every unit, and the gradients backpropagated later would also be identical.

Solution: randomly initialize parameters to small values close to 0, e.g.,

$$w_{jk}^{[i]} \sim \mathcal{N}(0, 0.1).$$

A better method in practice is called Xavier/He initialization:

$$w_{jk}^{[i]} \sim \mathcal{N}\left(0, \sqrt{\frac{2}{n^{[i]} + n^{[i-1]}}}\right),$$

where $n^{[i]}$ is the number of units in layer i. This encourages the variance of the outputs of a layer to be similar to the variance of the inputs.

Note that for ReLU hidden units, the recommendation for the bias weights is to use a small positive (even constant) initial value.

This ensures that the unit's output is initially positive for most training examples.

In the case of a single logistic sigmoid at the output layer, after forward propagation, we have a predicted value $\hat{y}$.

Neural network parameter update rules are usually derived in terms of backpropagating an error or loss.

If we have an objective function such as maximum likelihood, we can convert to a loss by negating it.

We thus get the log loss function for a network with a single logistic sigmoid output:

$$\mathcal{L}(\hat{y}, y) = -\left[(1-y)\log(1-\hat{y}) + y \log \hat{y}\right].$$

Note that it is easy to do the same for a linear output (Gaussian distribution for y) or softmax ouptut (multinomial distribution for y).[1]

[1] See Goodfellow et al. (2016) Section 6.2.2.4 for discussion of other output types.

Now, to update the parameters in layer l, we update using gradient descent on the log loss:

$$\begin{aligned} \mathbf{W}^{[l]} &\leftarrow \mathbf{W}^{[l]} - \alpha \frac{\partial \mathcal{L}}{\partial \mathbf{W}^{[l]}} \\ \mathbf{b}^{[l]} &\leftarrow \mathbf{b}^{[l]} - \alpha \frac{\partial \mathcal{L}}{\partial \mathbf{b}^{[l]}} \end{aligned}$$

First we consider the weights at the output layer. We have

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{[3]}} &= -\frac{\partial}{\partial \mathbf{W}^{[3]}}\left((1-y)\log(1-\hat{y}) + y\log\hat{y}\right) \\
&= -(1-y)\frac{\partial}{\partial \mathbf{W}^{[3]}}\log\left(1 - g(\mathbf{W}^{[3]}\mathbf{a}^{[2]} + \mathbf{b}^{[3]})\right) \\
&\quad -y\frac{\partial}{\partial \mathbf{W}^{[3]}}\log\left(g(\mathbf{W}^{[3]}\mathbf{a}^{[2]} + \mathbf{b}^{[3]})\right) \\
&= \frac{(1-y)g'(\mathbf{W}^{[3]}\mathbf{a}^{[2]} + \mathbf{b}^{[3]})\mathbf{a}^{[2]\top}}{1 - g(\mathbf{W}^{[3]}\mathbf{a}^{[2]} + \mathbf{b}^{[3]})} \\
&\quad -\frac{yg'(\mathbf{W}^{[3]}\mathbf{a}^{[2]} + \mathbf{b}^{[3]})\mathbf{a}^{[2]\top}}{g(\mathbf{W}^{[3]}\mathbf{a}^{[2]} + \mathbf{b}^{[3]})}
\end{aligned}$$

Continuing, we note that for this model, $g(z)$ is the logistic sigmoid.

Let's replace $g(z)$ with $\sigma(z)$ and $g'(z)$ with $\sigma'(z)$ to make this clear.

We also recall that $\sigma'(z) = \sigma(z)(1 - \sigma(z))$, allowing us to simplify the expression:

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{[3]}} &= \dots \\
&= (1-y)\sigma(\mathbf{W}^{[3]}\mathbf{a}^{[2]} + \mathbf{b}^{[3]})\mathbf{a}^{[2]\top} - y(1 - \sigma(\mathbf{W}^{[3]}\mathbf{a}^{[2]} + \mathbf{b}^{[3]}))\mathbf{a}^{[2]\top} \\
&= (1-y)a^{[3]}\mathbf{a}^{[2]\top} - y(1 - a^{[3]})\mathbf{a}^{[2]\top} \\
&= (a^{[3]} - y)\mathbf{a}^{[2]\top}.
\end{aligned}$$

What about the weights for layer 2?

We can use the chain rule from calculus. When we have a function $f(z)$ where $z = g(x)$, we can write

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial g}\frac{\partial g}{\partial x}$$

In our case we have

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{[2]}} = \frac{\partial \mathcal{L}}{\partial \mathbf{a}^{[3]}}\frac{\partial \mathbf{a}^{[3]}}{\partial \mathbf{z}^{[3]}}\frac{\partial \mathbf{z}^{[3]}}{\partial \mathbf{a}^{[2]}}\frac{\partial \mathbf{a}^{[2]}}{\partial \mathbf{z}^{[2]}}\frac{\partial \mathbf{z}^{[2]}}{\partial \mathbf{W}^{[2]}}.$$

To evaluate this expression, let's try to reuse what we already have for $\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{[3]}}$ first:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{[3]}} = \frac{\partial \mathcal{L}}{\partial a^{[3]}} \frac{\partial a^{[3]}}{\partial \mathbf{z}^{[3]}} \frac{\partial \mathbf{z}^{[3]}}{\partial \mathbf{W}^{[3]}} = (a^{[3]} - y)\mathbf{a}^{[2]}$$

we can reuse the part

$$\frac{\partial \mathcal{L}}{\partial \mathbf{z}^{[3]}} = \frac{\partial \mathcal{L}}{\partial a^{[3]}} \frac{\partial a^{[3]}}{\partial \mathbf{z}^{[3]}} = a^{[3]} - y.$$

For the remaining terms, we have

$$\begin{aligned} \frac{\partial \mathbf{z}^{[3]}}{\partial \mathbf{a}^{[2]}} &= \mathbf{W}^{[3]} \\ \frac{\partial \mathbf{a}^{[2]}}{\partial \mathbf{z}^{[2]}} &= g'(\mathbf{z}^{[2]}) \\ \frac{\partial \mathbf{z}^{[2]}}{\partial \mathbf{W}^{[2]}} &= \mathbf{a}^{[1]}. \end{aligned}$$

Putting the chain rule terms together in an order appropriate for vector calculations, we obtain

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{[2]}} = \text{diag}(g'(\mathbf{z}^{[2]}))\mathbf{W}^{[3]}(a^{[3]} - y)\mathbf{a}^{[1]\top}.$$

The calculation is also similar for the bias weights, except that in place of $\mathbf{a}^{[1]}$ we have 1.

What about the weights for layer 1?

We want

$$\frac{\partial \mathcal{L}}{\partial w_{ij}^{[1]}}.$$

We can readily see that $w_{ij}^{[1]}$ affects all of the second layer activations $\mathrm{a}^{[2]}$.

In this case, the applicable more general chain rule, when $y = f(\mathrm{u})$ and $\mathrm{u} = g(\mathrm{x})$ is

$$\frac{\partial y}{\partial x_i} = \sum_j \frac{\partial y}{\partial u_j} \frac{\partial u_j}{\partial x_i}.$$

In our case, the generalized chain rule gives

$$\frac{\partial \mathcal{L}}{\partial w_{ij}^{[1]}} = \sum_k \frac{\partial \mathcal{L}}{\partial a_k^{[2]}} \frac{\partial a_k^{[2]}}{\partial w_{ij}^{[1]}}.$$

Expanding the two terms within the summation, we obtain

$$\frac{\partial \mathcal{L}}{\partial w_{ij}^{[1]}} = \sum_k \frac{\partial \mathcal{L}}{\partial a^{[3]}} \frac{\partial a^{[3]}}{\partial z^{[3]}} \frac{\partial z^{[3]}}{\partial a_k^{[2]}} \frac{\partial a_k^{[2]}}{\partial z_k^{[2]}} \frac{\partial z_k^{[2]}}{\partial a_j^{[1]}} \frac{\partial a_j^{[1]}}{\partial z_j^{[1]}} \frac{\partial z_j^{[1]}}{\partial w_{ij}^{[1]}}.$$

Getting complicated, right?

The key to solving the problem efficiently is realizing that the term

$$\frac{\partial \mathcal{L}}{\partial a^{[3]}} \frac{\partial a^{[3]}}{\partial z^{[3]}} \frac{\partial z^{[3]}}{\partial a_k^{[2]}} \frac{\partial a_k^{[2]}}{\partial z_k^{[2]}}$$

has already been calculated, in the process of determining

$$\frac{\partial \mathcal{L}}{\partial w_{jk}^{[2]}}!$$

Really, go back and check. Now we see that we generally want to reuse terms that look like $\frac{\partial \mathcal{L}}{\partial z_i^{[l]}}$.

Let's therefore define

$$\delta_i^{[l]} = \frac{\partial \mathcal{L}}{\partial z_i^{[l]}}.$$

We then obtain the backpropagation algorithm for our sample network...

Backpropagation (fully-connected network)

Given example (x, y):

$\mathsf{a}^{[0]} \leftarrow \mathsf{x}$.

for $l = 1..L$ do

$\mathsf{z}^{[l]} \leftarrow \mathsf{W}^{[l]}\mathsf{a}^{[l-1]} + \mathsf{b}^{[l]}$.

$\mathsf{a}^{[l]} \leftarrow g^{[l]}(\mathsf{z}^{[l]})$.

$\boldsymbol{\delta}^{[L]} \leftarrow \frac{\partial \mathcal{L}}{\partial \mathsf{z}^{[L]}}$.

for $l = L..1$ do

$\frac{\partial \mathcal{L}}{\partial \mathsf{W}^{[l]}} \leftarrow \boldsymbol{\delta}^{[l]}\mathsf{a}^{[l-1]\top}$.

$\frac{\partial \mathcal{L}}{\partial \mathsf{b}^{[l]}} \leftarrow \boldsymbol{\delta}^{[l]}$.

if $l > 1$ then

$\boldsymbol{\delta}^{[l-1]} \leftarrow \mathrm{diag}(g'(\mathsf{z}^{[l-1]}))\mathsf{W}^{[l]}\boldsymbol{\delta}^{[l]}$

Backpropagation

Stochastic vs. batch gradient descent

Thus far, the derviation was for a single (x, y) pair.

What about batch gradient descent? We would use the rule

$$\mathrm{W}^{[l]} \leftarrow \mathrm{W}^{[l]} - \alpha \frac{\partial J}{\partial \mathrm{W}^{[l]}},$$

where J is the cost function

$$J = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}^{(i)}$$

and $\mathcal{L}^{(i)}$ is the loss for a single example.

Stochastic gradient descent will be more noisy but will usually converge faster than batch gradient descent.

Since batch gradient descent is more accurate but slower to get moving than stochastic gradient descent, a common compromise is mini-batch gradient descent.

The mini-batch is a compromise between the accuracy of batch and the the speed of stochastic gradient descent. We split the data set into partitions $B_1, B_2, \ldots$ (or repeatedly sample uniformly from the full training set) and let

$$J_i = \frac{1}{|B_i|} \sum_{j \in B_i} \mathcal{L}^{(j)}.$$

Another common optimization is called momentum.

The problem is that sometimes noisy data make us jump back and forth across valleys in the loss function, leading to slow convergence.

With momentum, we remember the last update to each parameter and use it to calculate a moving average of the gradient over time.

We use the following update rule:

$$\begin{aligned} \mathtt{V}_{dW^{[l]}} &\leftarrow \beta \mathtt{V}_{dW^{[l]}} + (1-\beta)\frac{\partial J}{\partial \mathtt{W}^{[l]}} \\ \mathtt{W}^{[l]} &\leftarrow \mathtt{W}^{[l]} - \alpha \mathtt{V}_{dW^{[l]}} \end{aligned}$$

The momentum term β will encourage the optimization to accelerate gradually toward the minimum.

For a more sophisticated version of momentum that is adaptive, take a look at the famous ICLR 2015 paper on Adam:

Kingma, D.P. and Ba, J.L. (2015), Adam: A Method for Stochastic Optimization. In *International Conference on Learning Representations*.

Adam usually outperforms other optimization methods such as SGD with momentum or RMSProp when applied to large NN models.

Outline

Avoiding overfitting

Overfitting

Neural networks are universal approximators.

Roughly speaking, given enough training data and sufficient model complexity, backpropgation can learn any function to an arbitrary level of accuracy.

But when model complexity is too high for the training set size, we observe overfitting: training accuracy is high, but validation set/test set accuracy is substantially lower.

Solutions:

- Decrease model complexity (remove hidden units, remove layers)
- Collect more training data
- Regularization

Avoiding overfitting

Regularization

Let w denote a single vector containing the set of all parameters in our model (every $w_{ij}^{[l]}$ and $b_i^{[l]}$).

Let J be the model cost function.

L2 regularization adds a new term to the cost function:

$$\begin{aligned} J_{L2} &= J + \frac{\lambda}{2}\|\mathrm{w}\|^2 \\ &= J + \frac{\lambda}{2}\mathrm{w}^\top \mathrm{w} \end{aligned}$$

Why do this?

- $\lambda = 0$ gives us unregularized parameter learning.
- Big λ encourages solutions with small w, especially, as many 0 elements as possible.

Forcing less useful parameters to 0 decreases model complexity and will reduce overfitting.

How does the regularizer affect the learning rule?

Previously, we had

$$\mathsf{w} \leftarrow \mathsf{w} - \alpha \frac{\partial J}{\partial \mathsf{w}}.$$

Now we have

$$\begin{aligned} \mathsf{w} &\leftarrow \mathsf{w} - \alpha \frac{\partial J}{\partial \mathsf{w}} - \alpha \frac{\lambda}{2} \frac{\partial \mathsf{w}^\top \mathsf{w}}{\partial \mathsf{w}} \\ &= (1 - \alpha\lambda)\mathsf{w} - \alpha \frac{\partial J}{\partial \mathsf{w}} \end{aligned}$$

Think about what this means (α and λ are both small positive reals).

On each update, we make all the weights' magnitudes a little smaller. Only the most important weights will survive.

Avoiding overfitting

Parameter sharing

Let's return to the soccer ball image recognition problem.

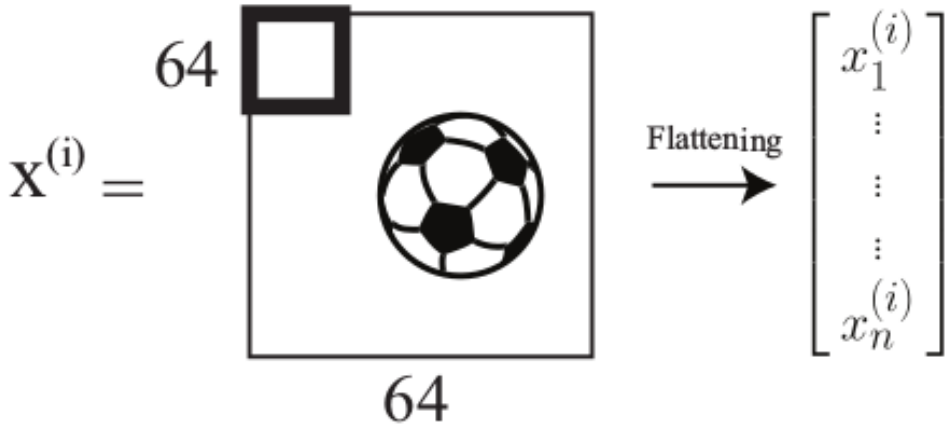
If we used a single logistic regression model for the $64 \times 64 \times 3 = 12{,}288$ parameters of the model, we'd have to train with soccer balls in all possible positions in the image.

Such a model would fail if it encountered a ball in a position it had never seen during training.

One solution is parameter sharing: each unit in the hidden layer looks at a different overlapping sub-region of the image, but these units share the same weights.

Avoiding overfitting

Parameter sharing



Ng (2017), CS229 lecture notes

We might draw $\boldsymbol{\theta}$ from $\mathbb{R}^{4\times4\times3}$, meaning we have one weight for each of the R, G, and B pixel intensities in a 4×4 region.

To learn θ:

- We might slide the region of interest over each positive training image to create many positive 48-element training vectors.
- Or we might represent each element in the "convolution" as a separate unit but tie their weights together during backpropagation.

The convolution style of weight sharing is the basic idea of the convolutional neural network (CNN):

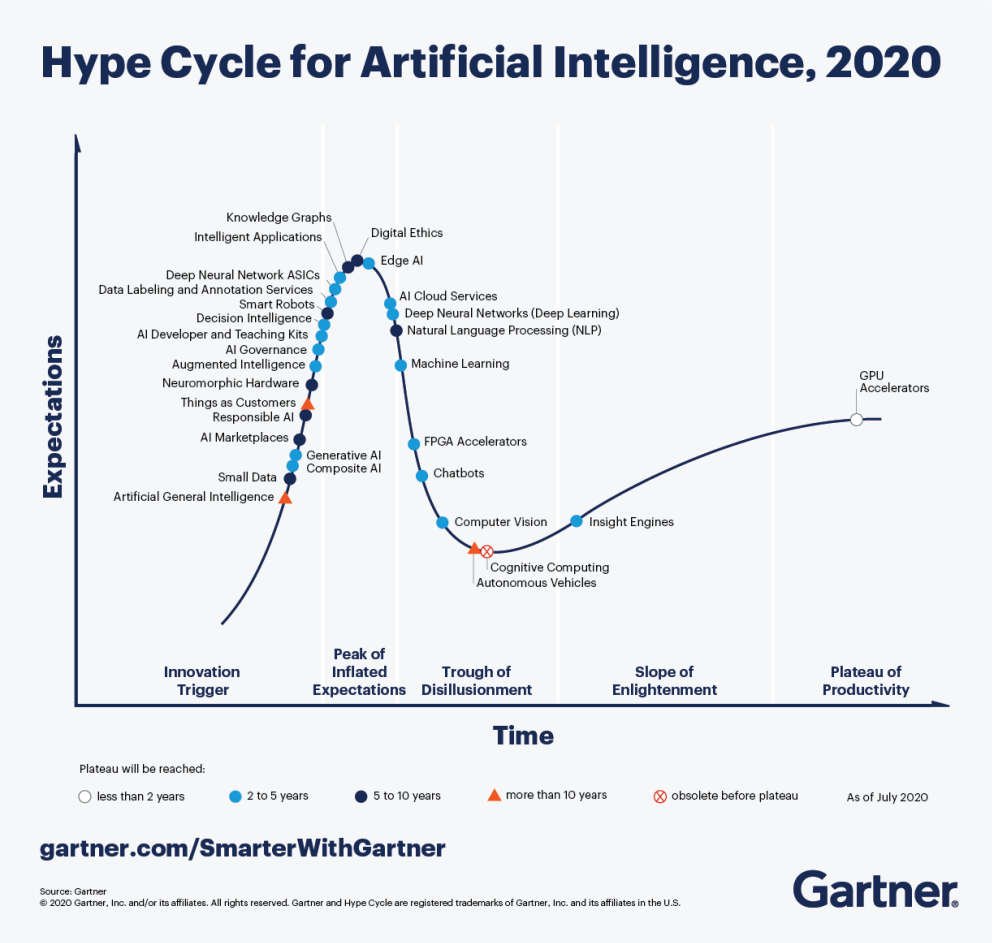
- Inspired by Hubel and Weisel's model of the responses of neurons in the cat's visual cortex to visual stimuli.
- Inspired by Fukushima's Neocognitron (1988).

Avoiding overfitting

Parameter sharing

CNNs:

- Applied successfully to handwritten character recognition by LeCun and colleagues (LeNet 5, 1998).
- Won the ImageNet Large Scale Visual Recognition Challenge in 2012 and every ImageNet competition since then.
- Jump started the recent round of hype in machine learning and AI!



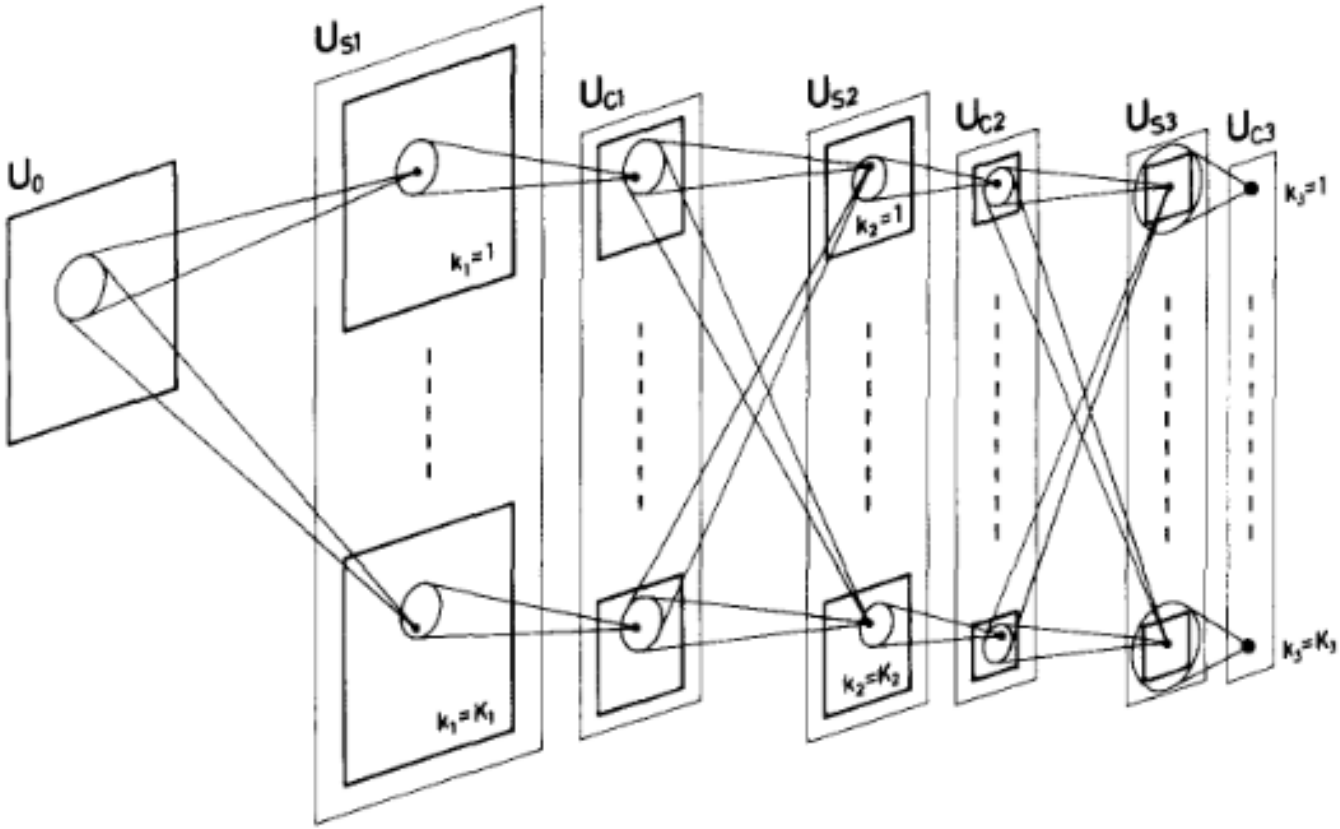
Gartner hype cycle (2020)

Outline

CNNs for image classification

Origins

The story of the CNN begins in the 1980s with Fukushima's Neocognitron, a hierarchical nerual network designed in primitive mimicry of the hierarchical processing in the primate visual cortex.



Fukushima (1980), Fig. 2

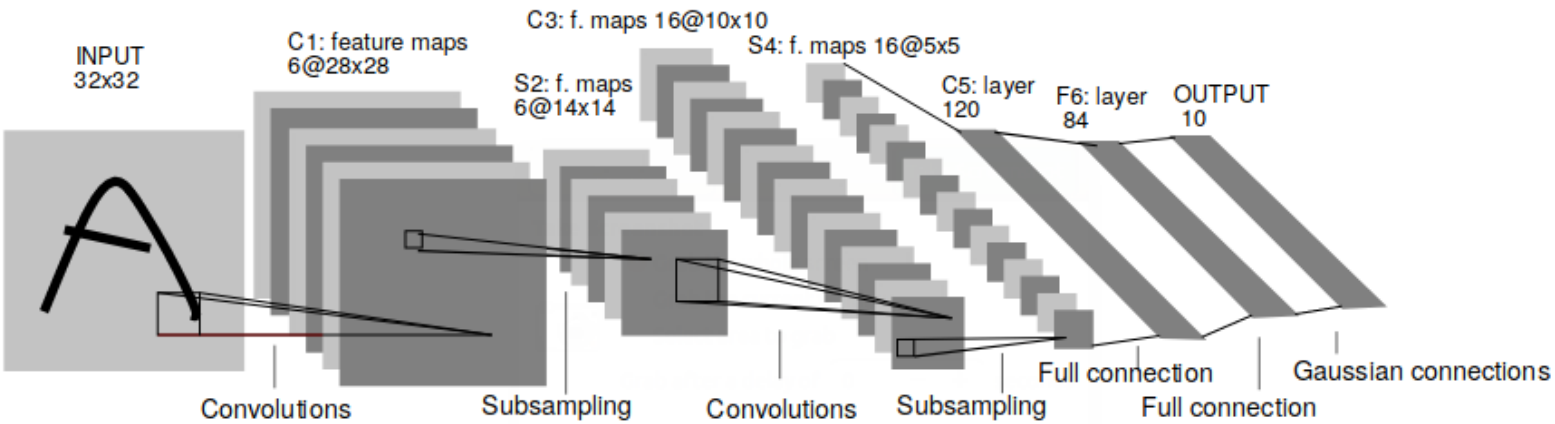
CNNs for image classification

Origins

Geoff Hinton was one of the rediscoverers of backpropagation in 1986.

Hinton's postdoc Yann LeCun went on to create the first practical modern convolutional neural networks for OCR at AT&T Research.[2]

LeCun and colleagues' LeNet-5 (1998) had the world's best performance at recognizing handwritten digits for several years.



LeCun et al. (1998), Fig. 2

[2] Today, Yann LeCun is chief AI scientist at Facebook.

CNNs for image classification

Origins

The 2000s were a golden era for feature-based methods like HOG.

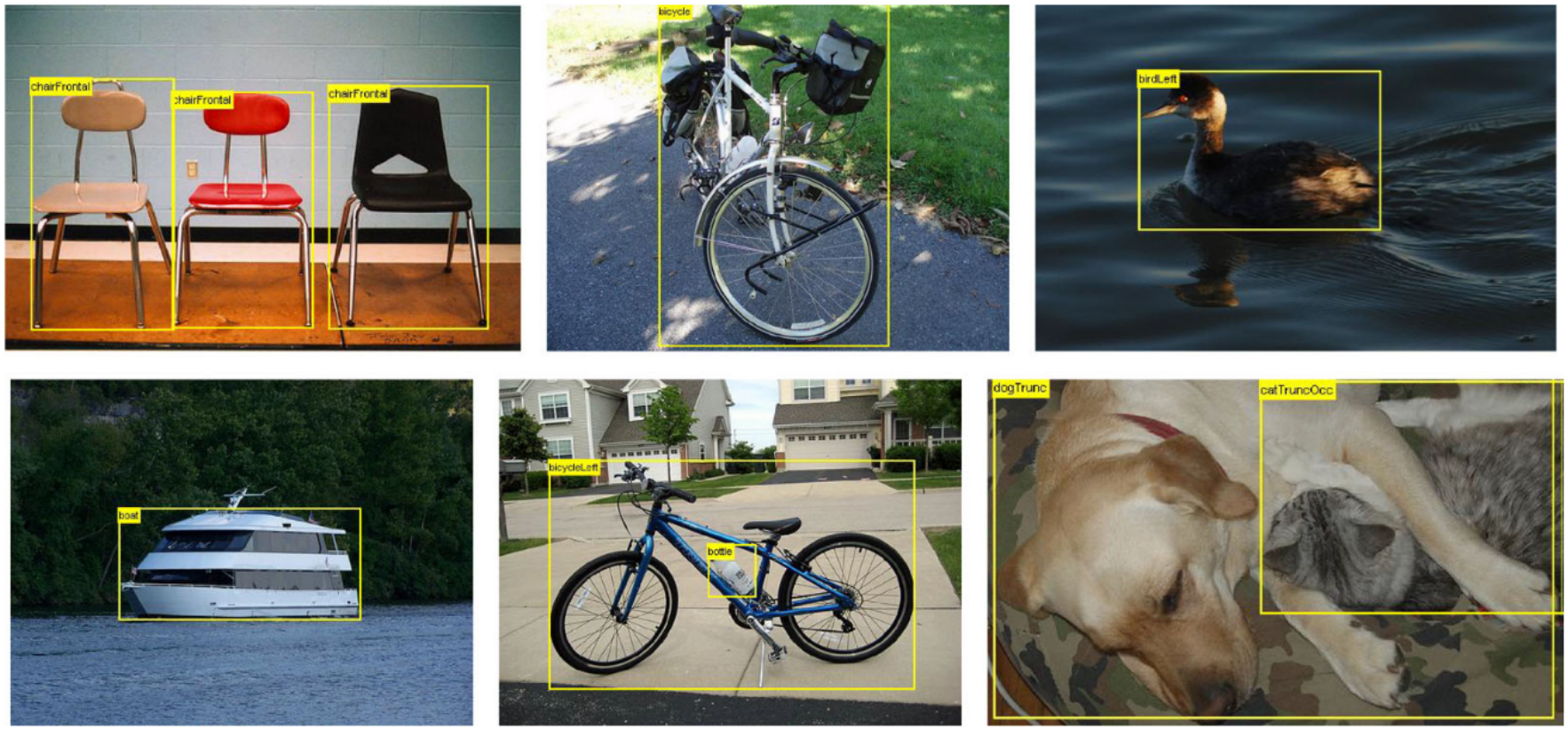
Some research continued, but most vision researchers ignored CNNs.

The main development pushing work forward was the emergence of standardized large-scale datasets.

2006: the PASCAL Visual Object Classification (VOC) challenge started with 20 object categories and ran with more difficult datasets each year until 2012.

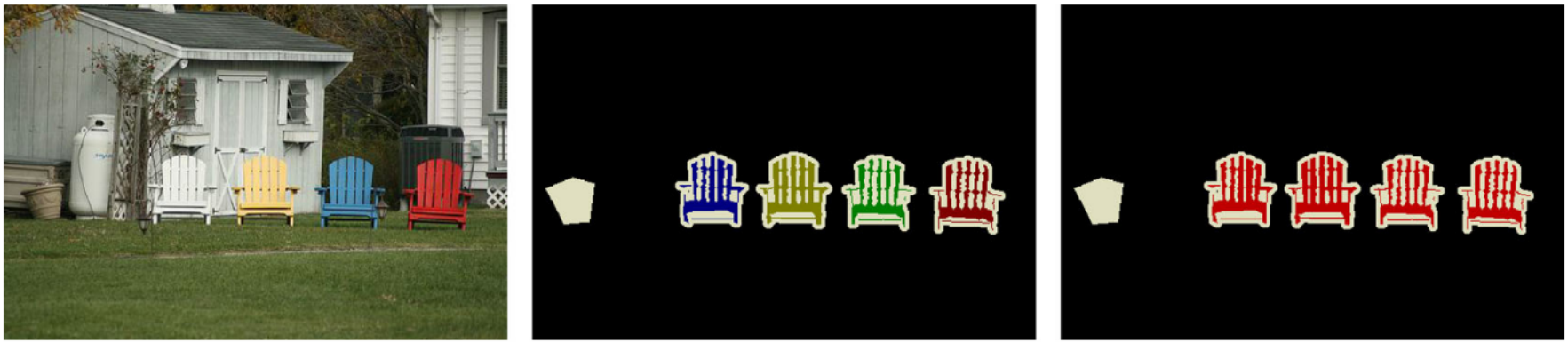
"Simple" feature based methods like HOG could not cope with VOC.

Sample VOC object detection and classification images:

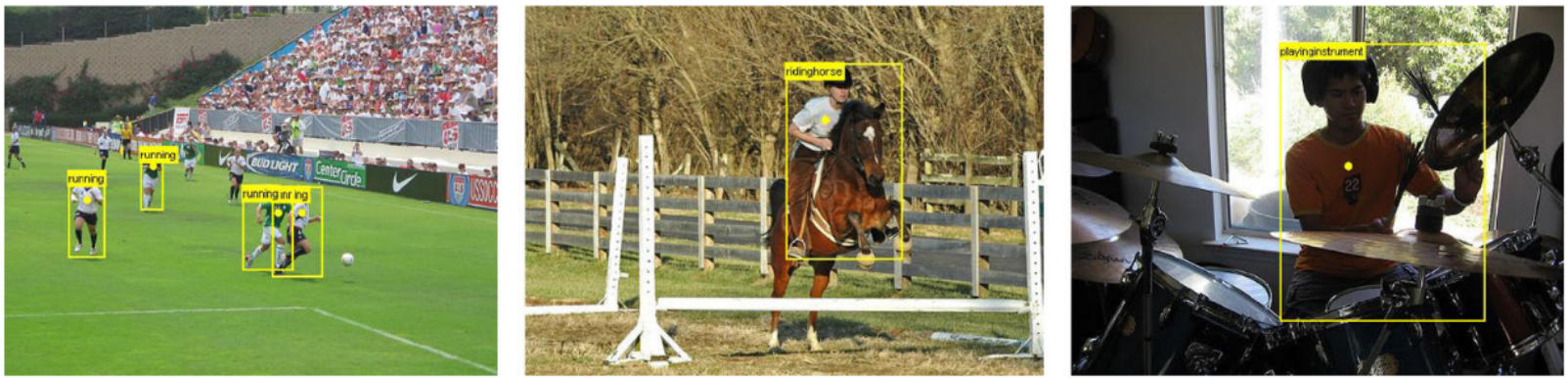


Everingham et al. (2015), Fig. 1a

Sample VOC segmentation and action classification images:



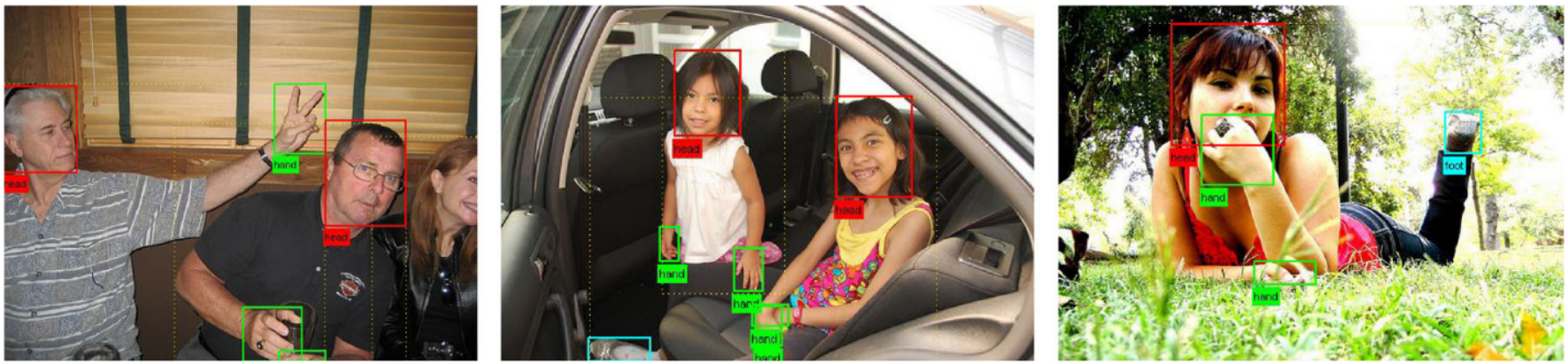
(b) Segmentation



(c) Action classification

Everingham et al. (2015), Fig. 1b-c

Sample VOC person layout images:



Everingham et al. (2015), Fig. 1d

The yearly VOC challenges helped push progress forward, but in retrospect it is clear that the dataset was too small.

CNNs for image classification

Origins

Another competition, ImageNet, had even greater influence than VOC.

2007: Fei-Fei Li, then at Princeton and later at Stanford, undertook a new effort to create a visual version of George Miller's WordNet, to be called ImageNet.

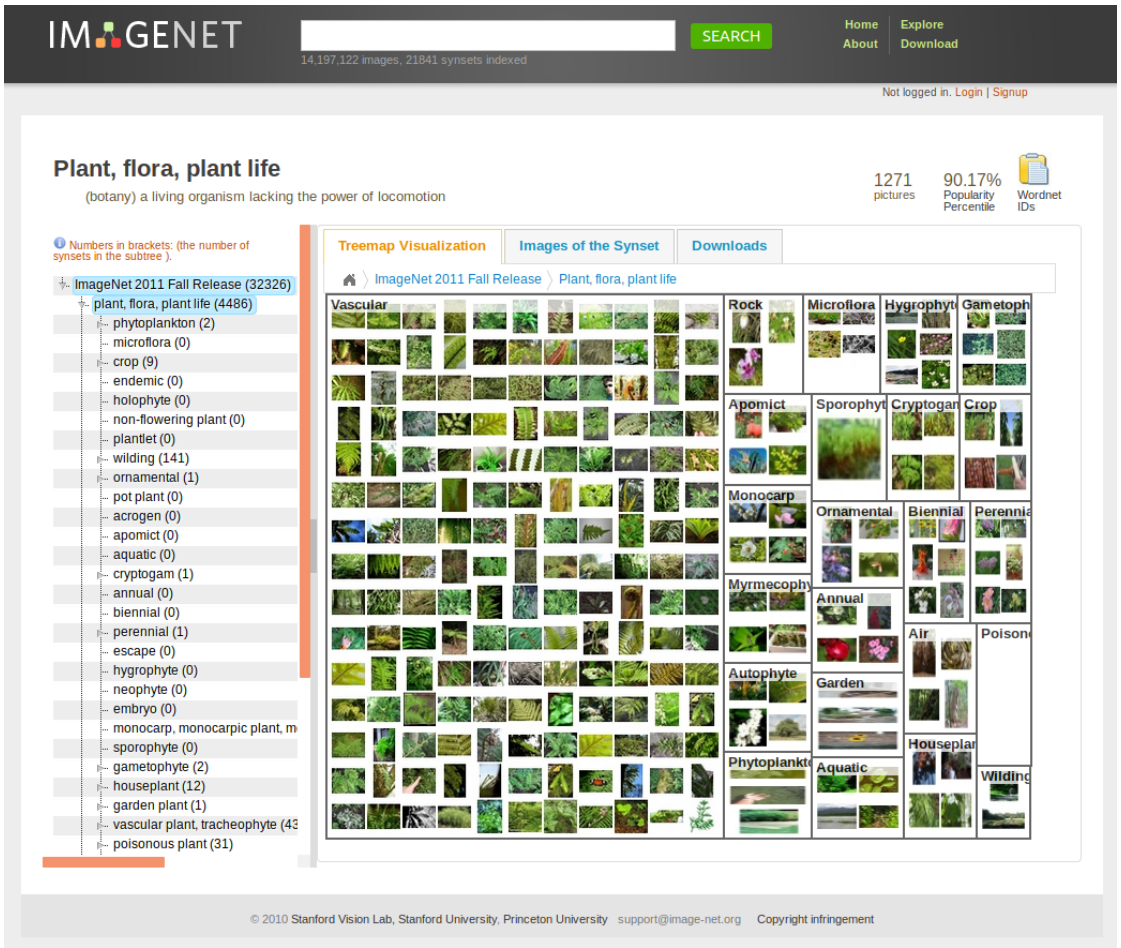
Li failed to get much funding for the project in the beginning but found that Amazon Mechanical Turk could be used to get humans to label images relatively cheaply.

2009: ImageNet was released at CVPR in a poster session then joined forces with PASCAL VOC for a 2010 competition: the ImageNet Large-Scale Visual Recognition Challenge (ILSVRC).

Eventually, the dataset had over 15 million images over 22,000 categories.

The 2010 contest dataset comprised 1.2 million images over 1000 categories.

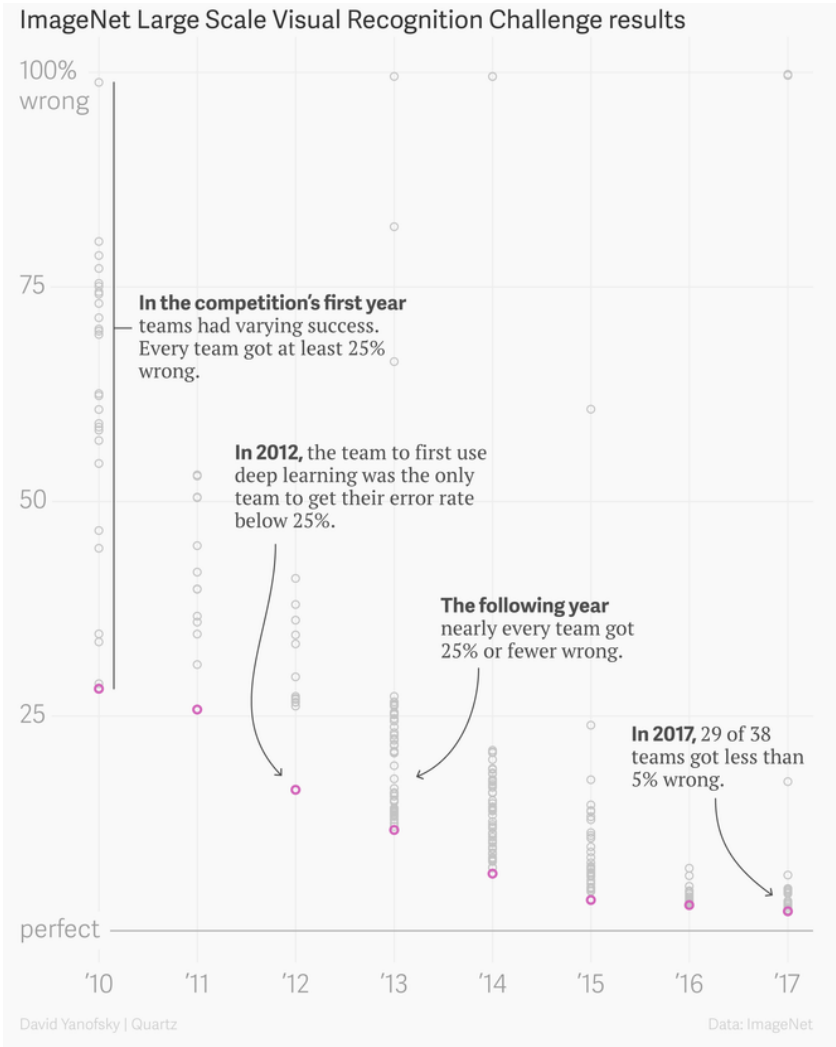
Samples from ImageNet:



http://image-net.org/explore

CNNs for image classification

Origins



https://qz.com/1034972/the-data-that-\
changed-the-direction-of-ai-research-\
and-possibly-the-world/

2010–2011: ImageNet competitors all had error rates over 25%.

2012: Krizhevsky, Sutskever, and Hinton submit AlexNet, sparking today's explosion of interest in AI and deep learning.

CNNs for image classification

AlexNet (2012)

Krizhevsky et al. begin with some provacative points:

- Nature is extremely diverse. We need extremely large datasets if we hope to learn that diversity.
- Learning from millions of images requires models with large capacity.
- CNN capacity can be controlled by varying their depth and breadth.
- The number of parameters in a CNN is smaller than that of similarly-sized fully connected models.
- CNNs decrease the number of parameters by making assumptions that seem to be correct: relevant features' statistics are stationary, and dependencies between pixels are mostly local.

These factors give us hope that CNNs may have the capacity to learn large datasets with relatively few parameters.[3]

[3] We will see in the next couple weeks that fewer parameters generally means lower VC dimension which in turn generally means better generalization.

CNNs for image classification

AlexNet (2012)

The authors trained what as of 2012 was the largest CNN ever, with 60 million parameters.

Training required a highly efficient implementation of the learning and runtime operations on GPUs with C++ and CUDA.

Training time was 5–6 days on two GTX 580 3GB GPUs.

Current version of the toolkit the authors built is available as open source: `https://code.google.com/archive/p/cuda-convnet2/`

CNNs for image classification

Aside: why convolutions?

2D convolution and cross-correlation operations give us the linear response of a 2D filter applied to the pixels in a local region of an image.

See any of many examples on YouTube, such as `https://www.youtube.com/watch?v=_iZ3Q7VXiGI` !

The simplest example is edge detection using, for example, Sobel filters:

$$\begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} \quad \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

[In computer vision, we often call cross-correlation "convolution."]

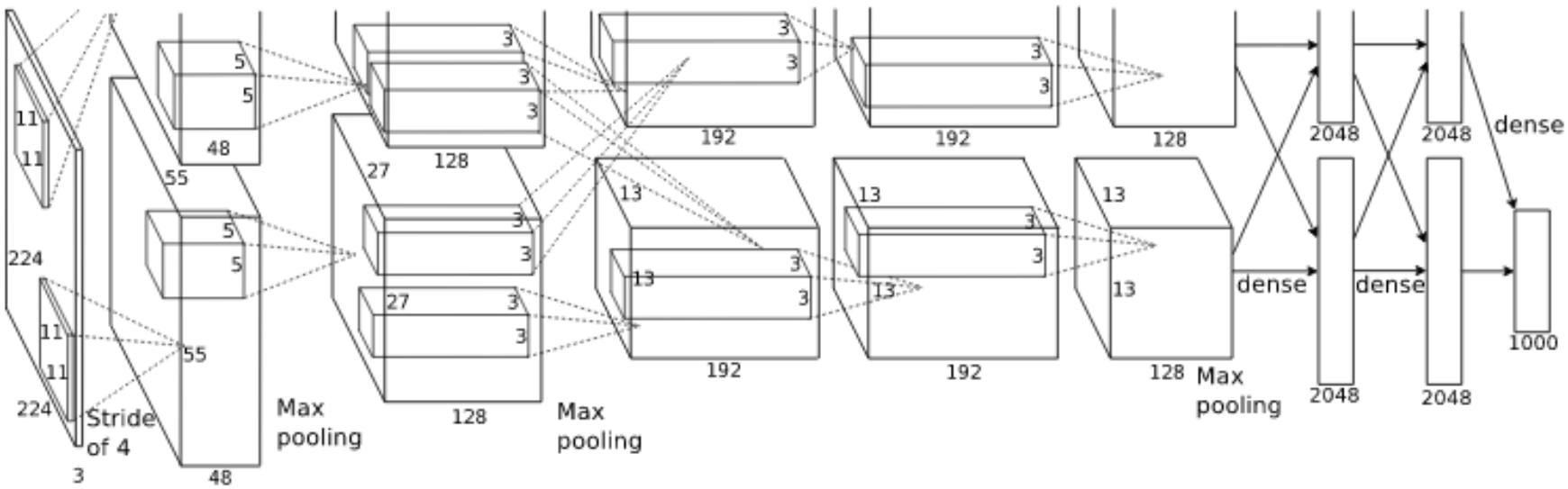
One of the insights of the CNN is to perform convolutions hierarchically.

The result of one convolution is transformed by a nonlinearity and possibly downscaled to obtain a feature map that is then convolved with higher-level filters.

CNNs for image classification

AlexNet (2012)

AlexNet begins with a preprocessing step in which the input image is scaled to 256 pixels in the shortest dimension then is cropped to 256×256.



Krizhevsky, Sutskever, and Hinton, (2012), Fig. 2

Five convolutional layers with ReLU activations (Nair and Hinton, 2010) and max pooling layers are followed by three fully-connected layers.

CNNs for image classification

AlexNet (2012)

AlexNet layers:

- $224 \times 224 \times 3$ input
- Convolution with 96 kernels of size $11 \times 11 \times 3$ with a stride of 4 pixels
- ReLU + local response normalization + overlapping max pooling
- Convolution with 256 kernels of size $5 \times 5 \times 48$ with a stride of 1.
- ReLU + local response normalization + overlapping max pooling
- 384 kernels of size $3 \times 3 \times 256$ with a stride of 1
- ReLU only (no normalization or pooling)
- 384 kernels of size $3 \times 3 \times 192$
- ReLU only
- 256 kernels of size $3 \times 3 \times 192$.
- ReLU only
- 4096 fully connected units
- 4096 fully connected units
- 1000 fully connected softmax units

How do the convolutional layers work?

A convolution may apply to the input or a feature map from a previous layer.

Typically, convolutions apply to all of the feature maps in the previous layer: this is called convolution over volume.

- For the input: three maps (R, G, and B) or one map (grayscale).
- For an inner layer: the number of kernels. This would be, e.g., 96 for AlexNet's second convolutional layer, except that AlexNet splits into two separate hierarchies so it is 48 for each stream.

Padding is important:

- Without padding, the border would shrink after each convolution, and information at the image border would be lost.
- In most cases, we should add padding necessary so that the output feature map has the same size as the input feature map when the stride is 1.
- The most common choice for padding seems to be 0-padding.
- Matt likes copy-border padding, but most libraries do not implement it.
- Most libraries do implement "reflect" padding which seems to work well (and is better than zero-padding).

Stride is also important:

- Typically, the convolution operation is applied at every pixel of the input (stride $= 1$).
- When spatial resolution is less important or neighboring receptive fields overlap significantly, we may skip some pixels of the input (stride > 1).
- Most architectures use a stride of 1 for 3×3 convolutions and a stride of 1–3 for 5×5 convolution.
- The trend: smaller kernels and more layers $\rightarrow$ small strides (11×11 convolutions as in AlexNet are not often seen).

CNNs for image classification

AlexNet (2012)

How to avoid overfitting when we have 60 million parameters?

For images, data augmentation using various image transformations makes sense.

AlexNet authors begin with a 256×256 image then sample 224×224 patches from the original.

- Training time: 2048 random patches including translation, horizontal reflections, and global random intensity transformations per image.
- Test time: four corner patches plus the center patch are processed, and the output layer is averaged over the 5 samples.

The second trick is dropout at the (first two) convolutional layers:

- Training time: each output in the feature map is set to 0 with probability 0.5. Zeroed outputs do not get any backpropagated error.
- Test time: all units are used but output is multipled by 0.5.

CNNs for image classification

AlexNet (2012)

Training parameters:

- Stochastic gradient descent
- Batch size 128 examples
- Momentum 0.9
- Weight decay 0.0005
- Weights initialized with samples from $\mathcal{N}(0, 0.01)$
- Biases initialized to 1 for most layers (to place ReLU in the positive region) and 0 for other layers (first and third convolutional layer).

The result: top-5 error rate dropped from 26% (2011) to 15.3%.

CNNs for image classification

ZFNet (2013)

In the 2013 ILSVRC, a "tweaked" version of AlexNet reduced the top-5 error rate to 12%.

CNNs for image classification

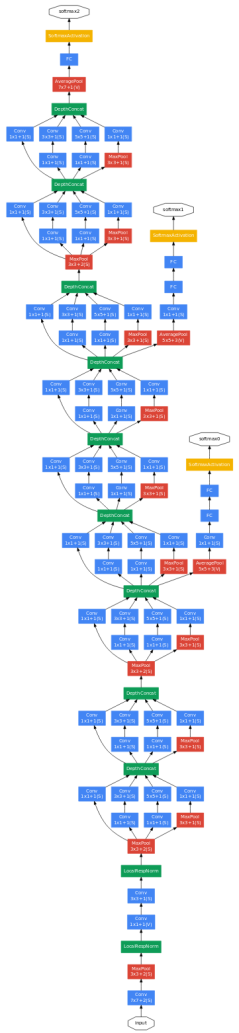
GoogLeNet (2014)

In the 2014 ILSVRC, Google's entry achieved a further huge improvement to a 6.7% top-5 error rate.

Principles: smaller convolutions, more layers, inception modules.

AlexNet's 60 million parameters reduced to 4 million.

Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A. (2015), Going deeper with convolutions. *IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1–9.



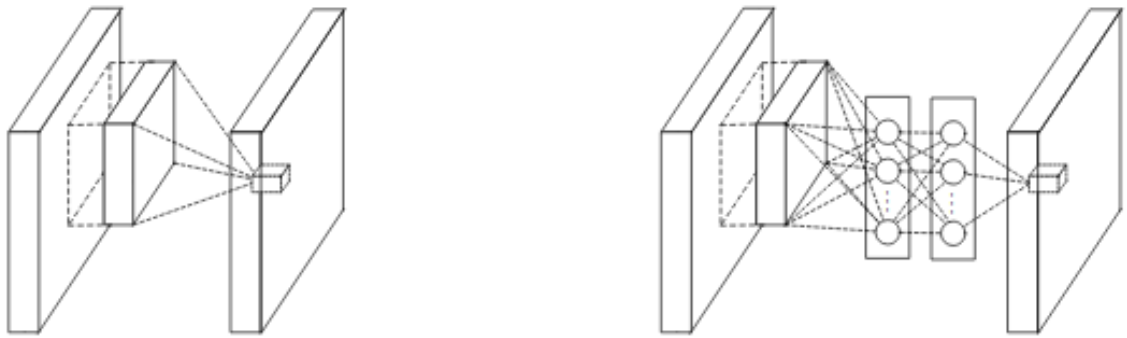
Szegedy et al. (2014), Fig. 3

CNNs for image classification

GoogLeNet (2014)

In 2013, Lin, Chen, and Yan at NUS had introduced the concept of Network-In-Network:

- In place of simple convolutions, local operations are performed by small multilayer perceptrons.
- The entire module is then scanned over the input like a convolution to produce a new feature map.



(a) Linear convolution layer (b) Mlpconv layer

Lin, Chen, and Yan (2013), Fig. 1

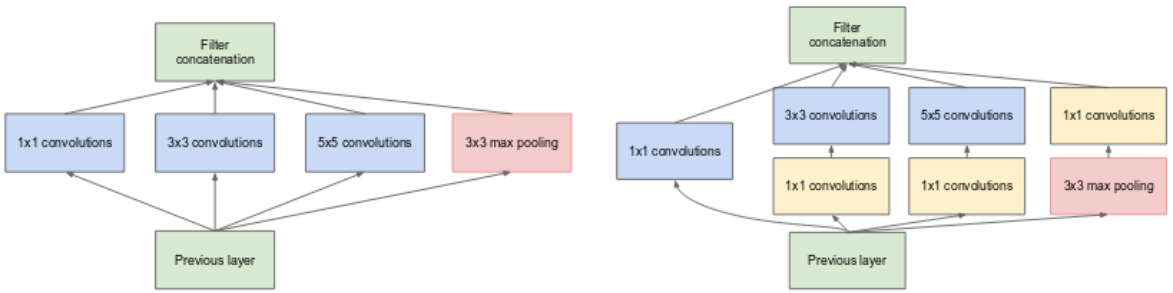
CNNs for image classification

GoogLeNet (2014)

Inception modules simplify the NIN concept, replacing the MLP with a single $1 \times 1 \times D$ convolution followed by ReLU, which can be implemented with standard CNN tools.

The 1×1 convolutions also aim to capture some of the theoretical work suggesting that extracting and combining sparse clusters of features over the image is optimal.

An inception module thus combines 1×1, 3×3, and 5×5 convolutions all feeding a single aggregating feature map.



(a) Inception module, naïve version (b) Inception module with dimension reductions

Szegedy et al. (2014), Fig. 2

GoogLeNet Image classification setup:

- 7 different models with different training pattern sampling.
- Test images are scaled to four sizes (256, 288, 320, and 352).
- For each scaled image, the left, center, and right or top, center, and bottom squares are taken.
- For each such image, 5 224×224 crops (4 corners plus center) and the entire region scaled to 224×224 are taken.
- For each crop, we take the original and horizontally flipped image.
- Total test images per input: $4 \times 3 \times 6 \times 2 = 144$.

CNNs for image classification

VGG (2014)

The 2nd place entry in 2014 was VGG (Simonyan and Zisserman, 2014).

Important features:

- 3×3 filters only
- 16–19 layers
- Otherwise similar to AlexNet
- 138 million parameters
- Tested on multiple crops through additional convolutional steps rather than averaging multiple crops

We learn that a deeper network with smaller convolutions is better than a shallower network with larger convolutions.

CNNs for image classification

ResNet (2015)

The 2015 ILSRVC winner was ResNet from Microsoft Research.[4]

He, K., Zhang, X., Ren, S., and Sun, J. (2016), Deep Residual Learning for Image Recognition, In *IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 770–778.

ResNet pushes the notion that "more depth is better" to the extreme.

The problem faced by very deep networks is degredation: though adding more layers improves training error to a point, eventually, training error starts to increase.

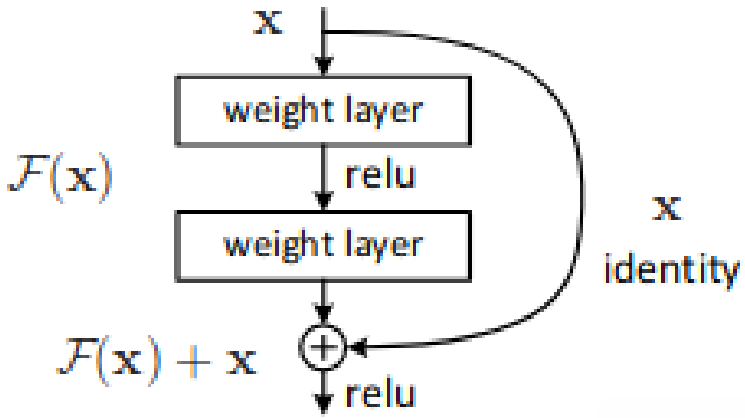
As we are talking about training error, the degradation is not overfitting — it is a purely due to the vanishing integrity of the training signal as the network gets deeper.

[4] Note, though, that Baidu had an entry that beat ResNet, but the entry was disqualified for cheating.

CNNs for image classification

ResNet (2015)

To overcome degredataion in very deep networks, ResNet uses the concept of residual learning:



He et al. (2016), Fig. 2

To learn a mapping $\mathcal{H}(\mathrm{x})$, we let intermediate layers learn another mapping $\mathcal{F}(\mathrm{x}) = \mathcal{H}(\mathrm{x}) - \mathrm{x}$ then compute $\mathcal{F}(\mathrm{x})$ at the output.

Residual learning can be implemented with shortcut connections that add the input to the output of the subnetwork.

CNNs for image classification

ResNet (2015)

When a subnetwork changes the dimensionality of input x, then a dimensionality changing mapping is used instead of the identity mapping.

He et al. demonstrate that

- Very deep networks without shortcut connections fail to learn the training set as well as similar networks with shortcut connections.
- A 152-layer network with shortcut connections can learn on ImageNet and CIFAR and won ILSVRC 2015 (3.56% top-5 error).

CNNs for image classification

ILSVRC 2016

The classification challenge continued in 2016 and 2017.

In 2016, the Trimps-Soushen team (sponsored by the Chinese Ministry of Public Security) won with 2.99% top-5 error rate.

Trimps-Soushen used a fusion strategy combining the results of various Inception and ResNet models, weighted by their accuracy.

CNNs for image classification

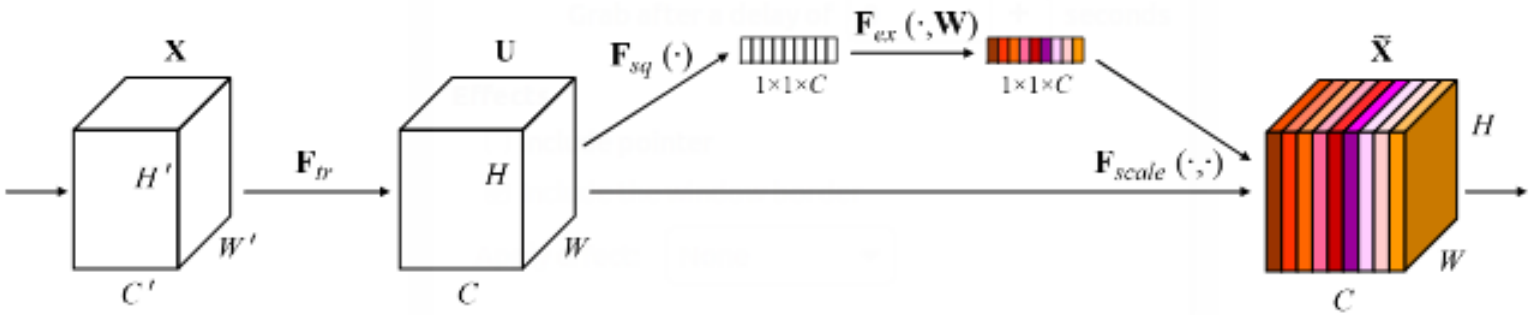
ILSVRC 2017

The final recognition challenge was won with 2.25% error rate by a team from Momenta (a Chinese self-driving car company) and Oxford.[5]

The model is called a squeeze-and-excitation network (SENet).

Hu, J., Shen, L., and Sun, G. (2018), Squeeze-and-excitation networks. In *IEEE Computer Society Conference on Computer Vision and Pattern Recognition*.

Basic idea: reweight (scale) channels globally according to informativeness.



Hu, Shen, and Sun (2018), Fig. 1

[5] Trained Caffe models available at `https://github.com/hujie-frank/SENet`

Lessons to be learned from ILSVRC entries:

- Use small convolutions
- Go deeper
- Reduce dimensionality and number of parameters when possible
- Combine multiple models
- Learn residuals rather than direct mappings
- Amplify informative channels

Some informative exercises:

- Derive the backpropagation rule for convolutional layers
- Derive the backpropagation rule for a softmax loss layer

Hint 1: Assume you already have $\delta_{ijk}^{[l]}$, equal to $\frac{\partial \mathcal{L}}{z_{ijk}}$, the net input to the (i, j)-th element of feature map k in layer l. Figure out what is $\frac{\partial \mathcal{L}}{\partial w_{mnok}^{[l]}}$, the weight for to the (m, n, o)-th weight for feature map k in layer l.

Hint 2: refer to the lecture notes on the GLM and multinomial distribution. The posterior estimate for a GLM is the softmax

$$p(y = i \mid \mathrm{x}; \Theta) = \frac{e^{\boldsymbol{\theta}_i^\top \mathrm{x}}}{\sum_{j=1}^{k} e^{\boldsymbol{\theta}_j^\top \mathrm{x}}},$$

and the log likelihood is

$$\ell(\Theta) = \sum_{i=1}^{m} \log \prod_{l=1}^{k} \left(\frac{e^{\boldsymbol{\theta}_l^\top \mathrm{x}^{(i)}}}{\sum_{j=1}^{k} e^{\boldsymbol{\theta}_j^\top \mathrm{x}^{(i)}}} \right)^{\delta(y^{(i)}=l)}.$$

Outline

Conclusion

This has been a very brief introduction to deep learning.

Hopefully you get the main ideas and can see how to extend them to new problems.

Want to go further?

- Read up on the literature, beginning with AlexNet then moving forward.
- Learn one or more frameworks such as Caffe, Tensorflow, Darknet, or Torch.
- A nice resource: `https://adeshpande3.github.io/The-9-Deep-Learning-Papers-You-Need-To-Know-About.html`
- Take the DS&AI Computer Vision course.
- Take the DS&AI Recent Trends in Machine Learning (Deep Learning) course.