Machine Learning
Deep Learning

dsai.asia

Asian Data Science and Artificial Intelligence Master's Program

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.

Outline

1. Introduction
2. Neural network intuition
3. Efficient computation
4. Backpropagation
5. Avoiding overfitting
6. CNNs for image classification
7. Conclusion

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

1. Introduction
2. Neural network intuition
3. Efficient computation
4. Backpropagation
5. Avoiding overfitting
6. CNNs for image classification
7. Conclusion

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).

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.

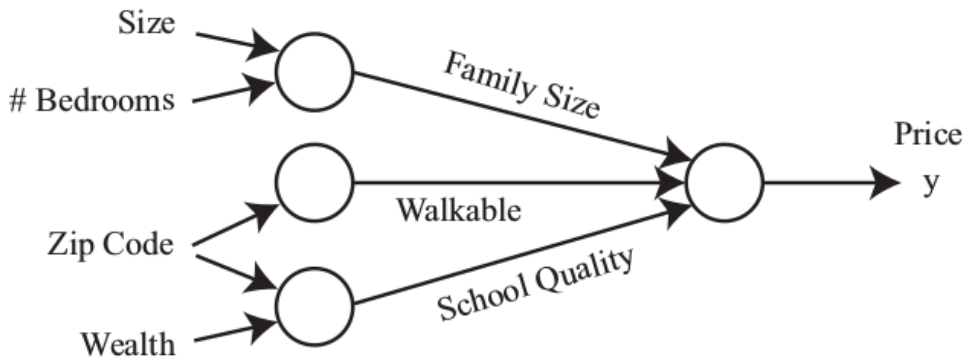
The postal code could be used to compute how "walkable" the neighborhood is.

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

Neural network intuition

Composing units into networks

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.

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

1. Introduction
2. Neural network intuition
3. Efficient computation
4. Backpropagation
5. Avoiding overfitting
6. CNNs for image classification
7. Conclusion

Efficient computation

Calculating activations

Consider calculating hidden unit activations. We have

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

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

1. Introduction
2. Neural network intuition
3. Efficient computation
4. Backpropagation
5. Avoiding overfitting
6. CNNs for image classification
7. Conclusion

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.

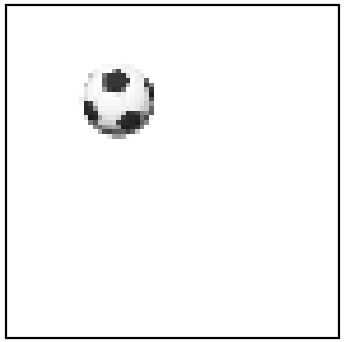
The general procedure is called backpropagation.

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

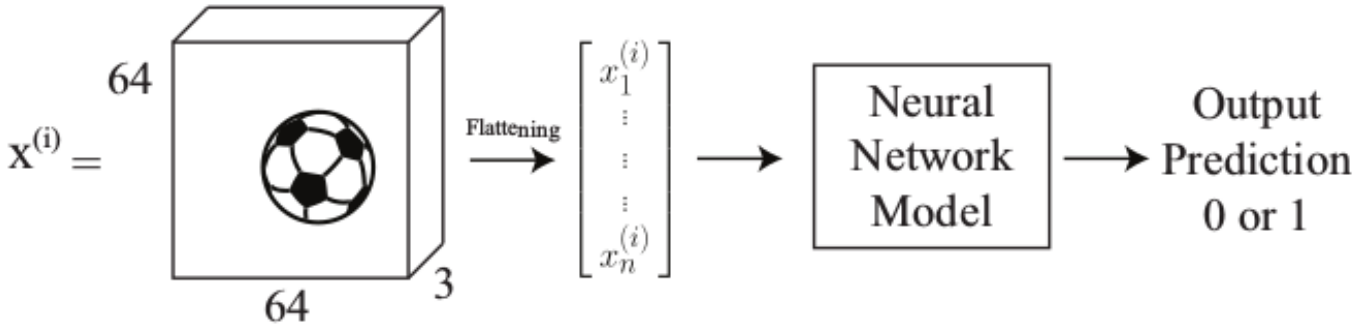
Backpropagation

Flattening the input

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

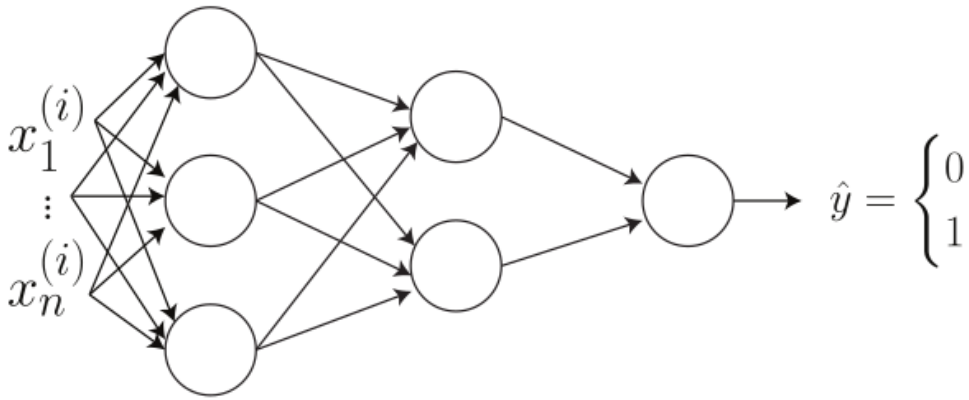
Backpropagation

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.

Backpropagation

Parameter update

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

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

Backpropagation

Parameter update

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

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

we can reuse the part

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

For the remaining terms, we have

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

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

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

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

1. Introduction
2. Neural network intuition
3. Efficient computation
4. Backpropagation
5. Avoiding overfitting
6. CNNs for image classification
7. Conclusion

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.

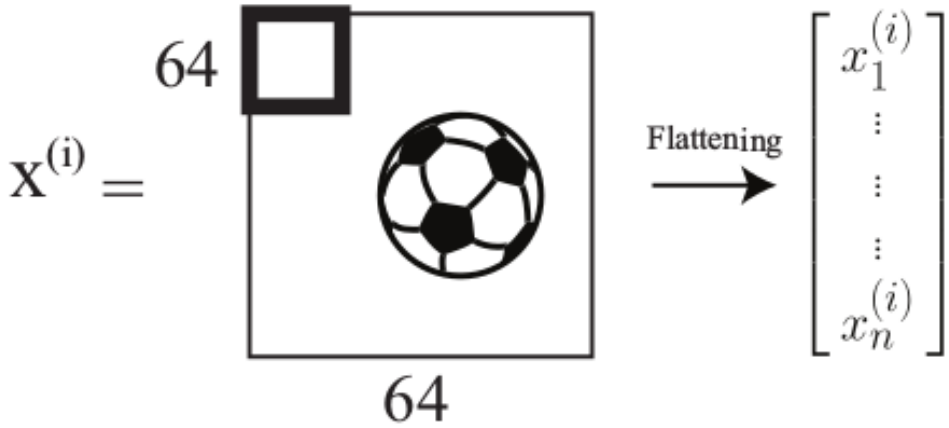
If we used a single logistic regression model for the $64 \times 64 \times 3 = 12{,}288$ parameters of the model, we'd have to train with soccer balls in all possible positions in the image.

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):

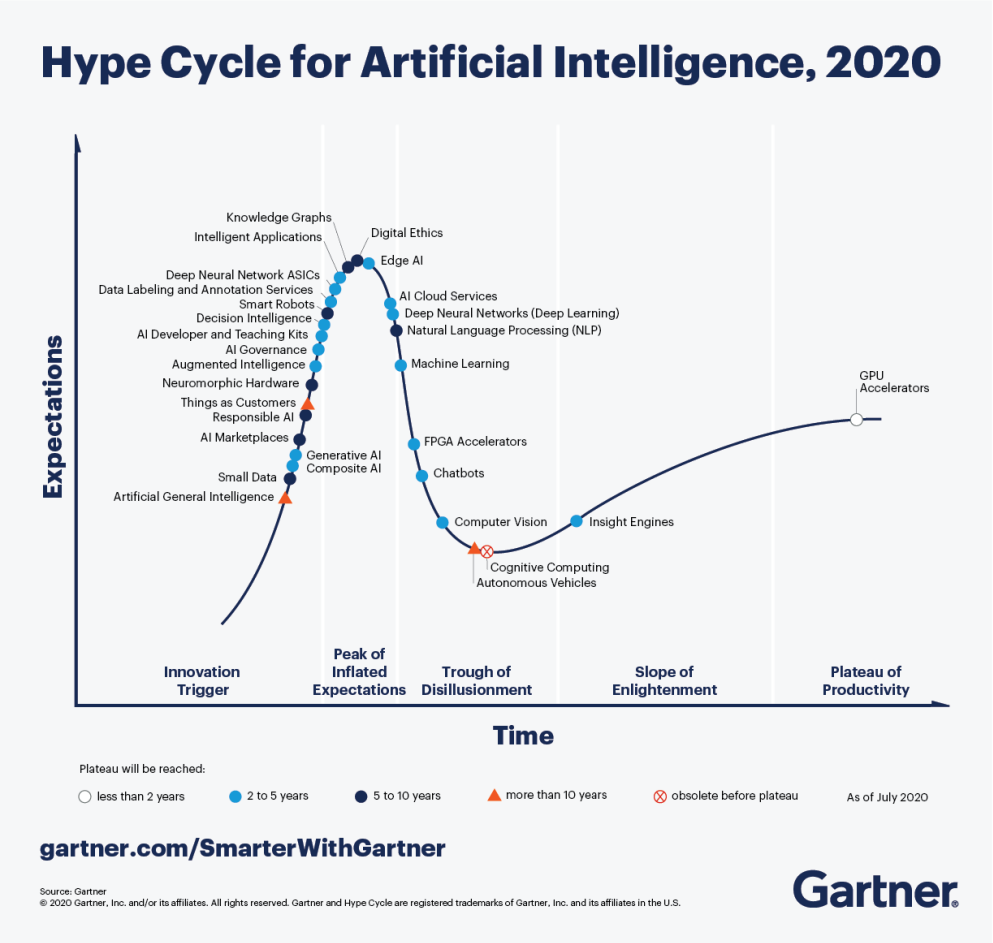
- Inspired by Hubel and Weisel's model of the responses of neurons in the cat's visual cortex to visual stimuli.
- Inspired by Fukushima's Neocognitron (1988).

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

1. Introduction
2. Neural network intuition
3. Efficient computation
4. Backpropagation
5. Avoiding overfitting
6. CNNs for image classification
7. Conclusion

CNNs for image classification

Convolutional neural networks (CNNs) are suited for a variety of tasks that involve forming a big picture view of data through hierarchical synthesis of local spatial and/or temporal relationships.

The most obvious type of data we need to do this with is images. In computer vision, the three primary tasks are

- Image classification
- Object detection
- Image segmentation
- Object tracking

These problems vexed computer vision researchers for decades, but due to recent deep learning methods we may consider the first two problems solved!

Among these four applications, those involving supervised machine learning are classification, detection, and segmentation.

Detection is really just classification:

- We may classify each subwindow of an image sequentially using a sweep window.
- We may look for "interesting" locations first using some kind of interest operator.
- Or, we may classify all possible locations in parallel.

Similarly, segmentation is just classification:

- We may classify each pixel as a member of the same segment as its neighbors or a different segment.
- In semantic segmentation, we attach a specific category label to each pixel of the image.

The key in all of these cases is to be able to classify an image or a patch of an image or a pixel of an image.

We'll thus first look at the state of the art in image classification then see how detection and segmentation can be done.

Convolutional neural networks (CNNs) are good for data with a known grid-like topology:

- Time series form 1-D grids.
- Images form 2-D grids.

Mathematically: convolution is an operation on two functions with a real-valued (possibly vector-valued) argument.

Continuous convolution for smoothing (Goodfellow et al., 2016)

We might use convolution for smoothing a 1-D function such as continuous noisy measurements $x(t)$ of the linear position of a spaceship:

$$s(t) = \int x(a)w(t-a)da$$

$$s(t) = (x * w)(t)$$

$w(\cdot)$ in the smoothing example:

- Should be a probability density function to make it a weighted average.
- Should be 0 for negative arguments (the future)

There are many other applications of continuous convolution.

Now we move to discrete convolution.
$x(t)$ is assumed to be a discrete sample from some underlying continuous function taken at regular spatial or temporal intervals.

We'll call $x(t)$ the input and $w(t)$ the kernel.

The output is usually called a feature map.

The continuous integral becomes a discrete sum:

$$s(t) = (x * w)(t) = \sum_{a=-\infty}^{\infty} x(a)w(t-a)$$

In a ML application, the input is an arbitrary multidimensional array of data and the kernel is a multidimensional array of parameters adapted by the learning algorithm.

Multidimensional arrays are called tensors.

The theoretically infinite functions are assumed 0 everywhere but in the finite set of points we have stored.

Example: a two-dimensional image $I(\cdot,\cdot)$ would best be processed by a two-dimensional kernel $K(\cdot,\cdot)$:

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n).$$

Convolution is commutative, so we can equivalently write

$$S(i,j) = (I * K)(i,j) = (K * I)(i,j) = \sum_m \sum_n I(i-m, j-n) K(m,n).$$

This version is easier to implement, as the loop is over the smaller number of non-zero values in $K(\cdot,\cdot)$.

Note that in both $I * K$ and $K * I$, as the index into the input increases, the input into the kernel decreases, and vice versa.

We have flipped the kernel.

Flipping the kernel is necessary mathematically for commutativity, but is not necessary for the actual information processing. Therefore, we may use cross-correlation

$$S(i,j) = (K \star I)(i,j) = \sum_m \sum_n I(i+m, j+n) K(m,n),$$

in which we don't flip the kernel. You will often see cross-correlation called convolution in machine learning libraries.

In this class, when we say "convolution" we will normally mean cross correlation!

2D convolution 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:

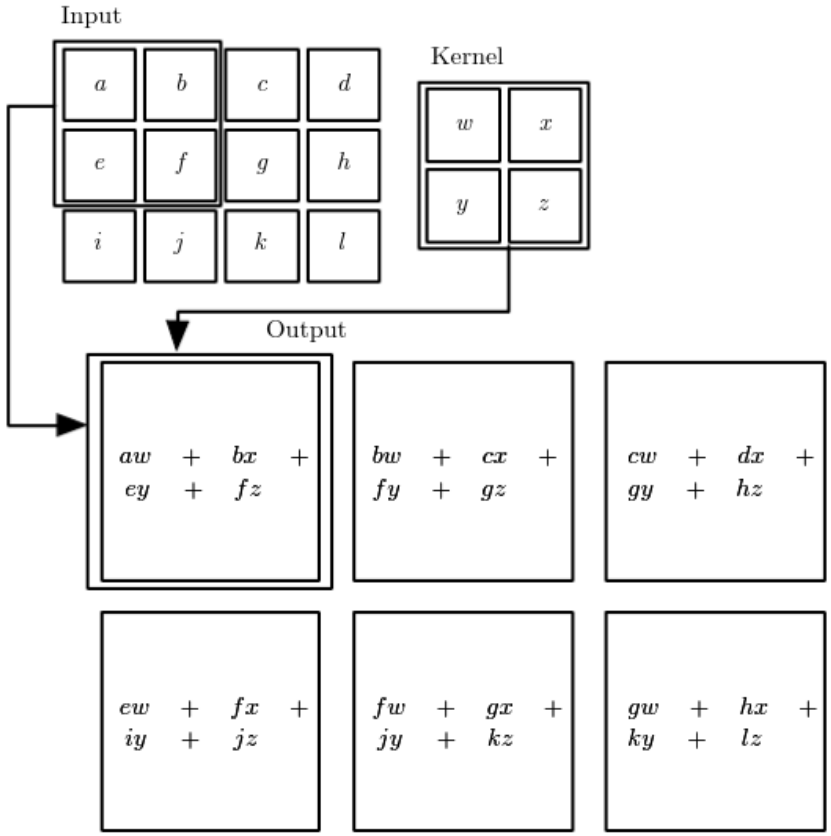
$$\begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} \quad \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

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.

Example below. Note that we only use the valid parts of the convolution where the kernel fully overlaps the valid part of the input.



Goodfellow et al. (2016), Figure 9.1

There are three main important ideas behind the success of convolutional neural networks:

- Sparse interactions
- Parameter sharing
- Equivariant representations

(Aside: a benefit of convolutions is that they can be applied to an input of variable size.)

Sparse interactions:

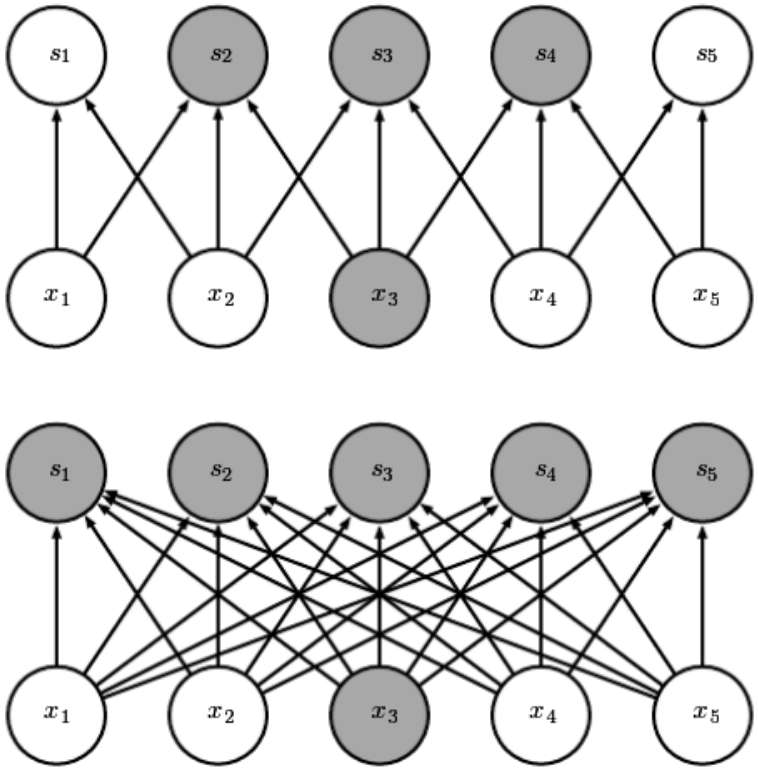
- A fully connected layer has dense interactions (every unit interacts with every unit in the previous layer)
- A convolutional layer has sparse interactions (each unit in the output feature map interacts only with a few elements of the input, via the small kernel.

Fewer parameters means lower memory requirements and higher statistical efficiency.

The specific organization of a convolution also means it can be implemented efficiently.

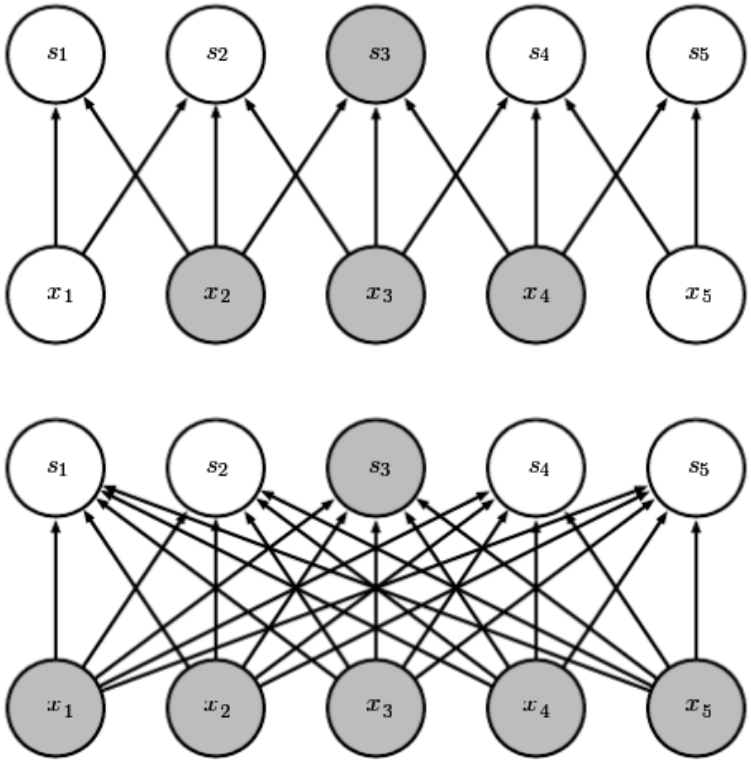
Dense connections from m inputs to n outputs requires $O(mn)$ time. With a kernel of size k, we only need $O(kn)$ time.

Sparse interaction example. Unit x_3 only affects units s_2, s_3, and s_4 in the output feature map. Compare to dense interaction below.



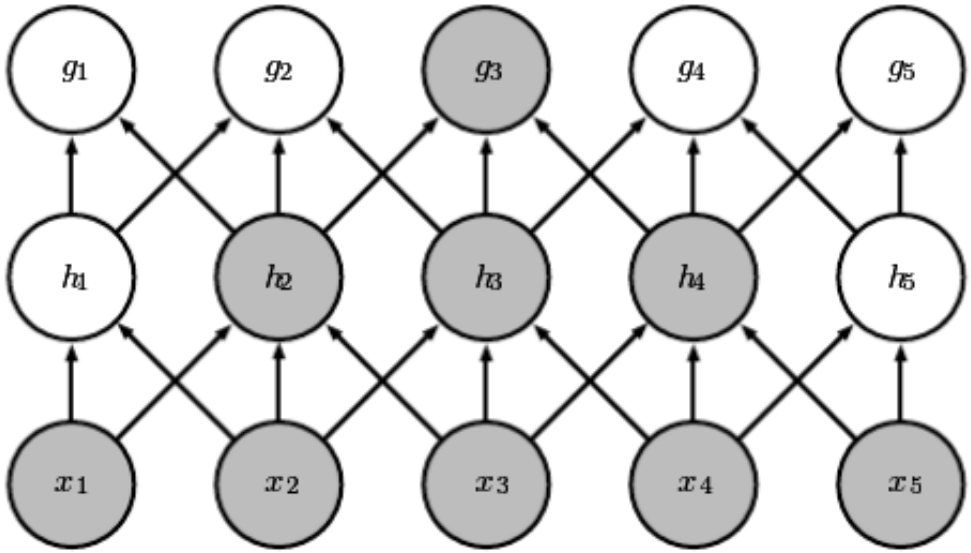
Goodfellow et al. (2016), Figure 9.2

Looking from above, unit s_3 in the output feature map has a receptive field including x_2, x_3, and x_4.



Goodfellow et al. (2016), Figure 9.3

When convolutional layers are formed into hierarchies, a unit in a deep layer (g_3) can be indirectly connected to all or most of the input.



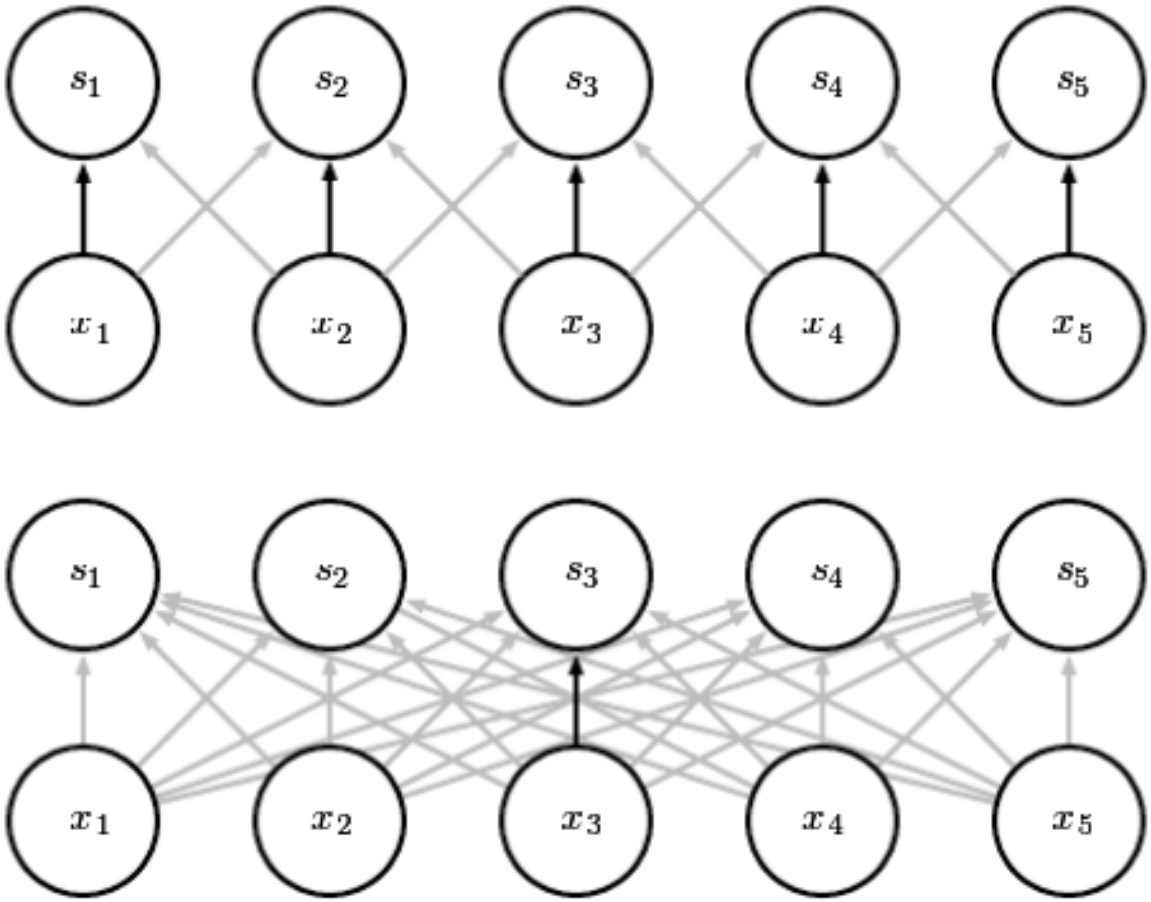
Goodfellow et al. (2016), Figure 9.4

Parameter sharing:

- Sharing means using the same parameter for more than one interaction in a model.
- Shared parameters are also often called tied weights.
- In a convolution, each weight in the kernel is reused at every position in the input.

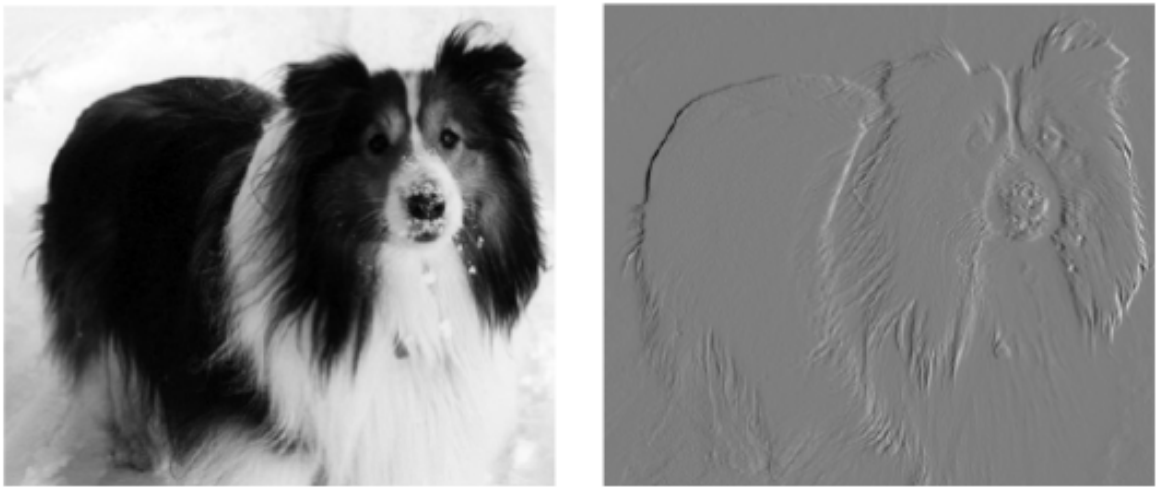
Parameter sharing increases statistical efficiency.

With parameter sharing (top), one parameter is used many times. Without parameter sharing, one parameter is used only once.



Goodfellow et al. (2016), Figure 9.5

The representational power of convolution with shared parameters is clear with the following example of vertical edge detection using a 2-element kernel (-1,1):



Goodfellow et al. (2016), Figure 9.6

Equivariance:

- A function $f(x)$ is equivariant to function g if $f(g(x)) = g(f(x))$.
- Convolution is equivariant to translation.
- If f is a convolution operation and g is a translation operation, we can see that the convolution of a translated version of input x is the same as the translation of the convolution of f and the input x.

How is equivariance useful?

For time series: we get the same response to the same event, regardless of when the event occurs.

For images: we get the same response to a local pattern, regardless of where in the image it occurs.

Note that convolution is NOT equivariant to scale, rotation, etc.

CNNs for image classification

Activation / rectification

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

However, a hierarchy of purely linear transformations would ultimately be equivalent to a single linear transformation, which would not give powerful pattern matching capabilities.

Generally, then, the result of one convolution in the hierarchy should be transformed by a nonlinearity.

ReLU is the most popular nonlinear activation function.

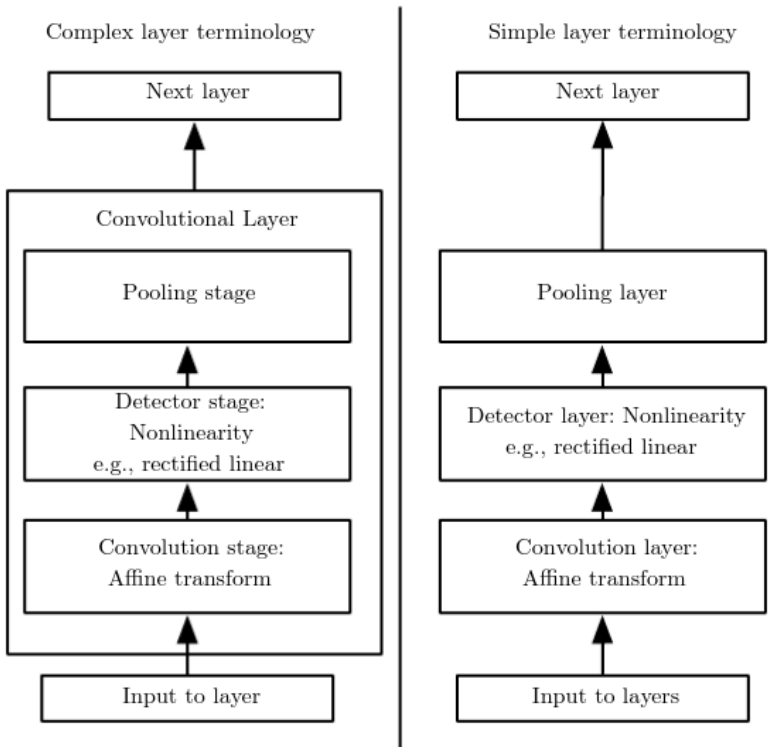
Hyperbolic tangent and the logistic sigmoid are also possible but lead to slower learning, according to Krizhevsky et al. (AlexNet).

The resulting feature map may possibly be downscaled by a pooling operation before it is convolved with higher-level filters.

CNNs for image classification

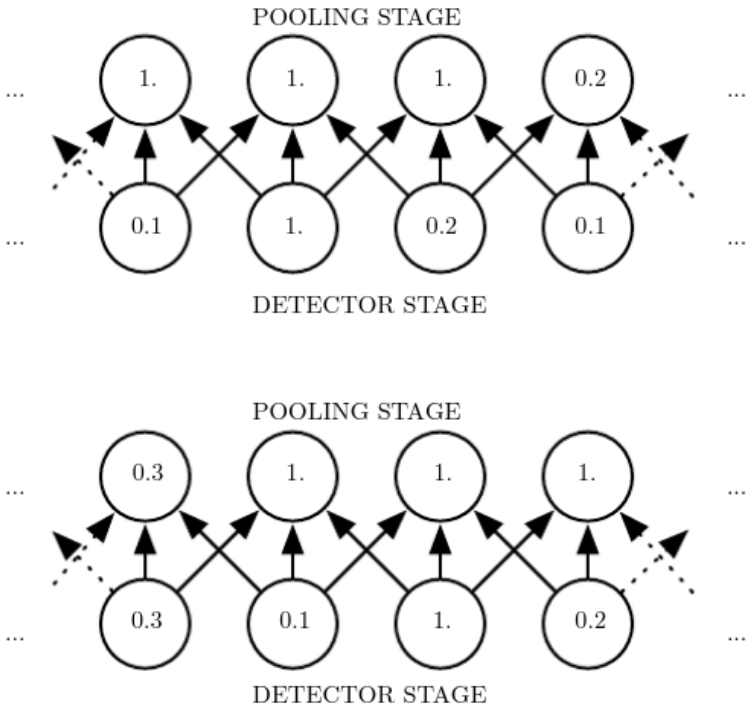
Pooling

The structure of a CNN usually contains several "macro" layers consisting of a convolution, a nonlinearity especially ReLU, then pooling. Sometimes a "layer" means all three operations, and sometimes each operation is treated separately.



Goodfellow et al. (2016), Figure 9.7

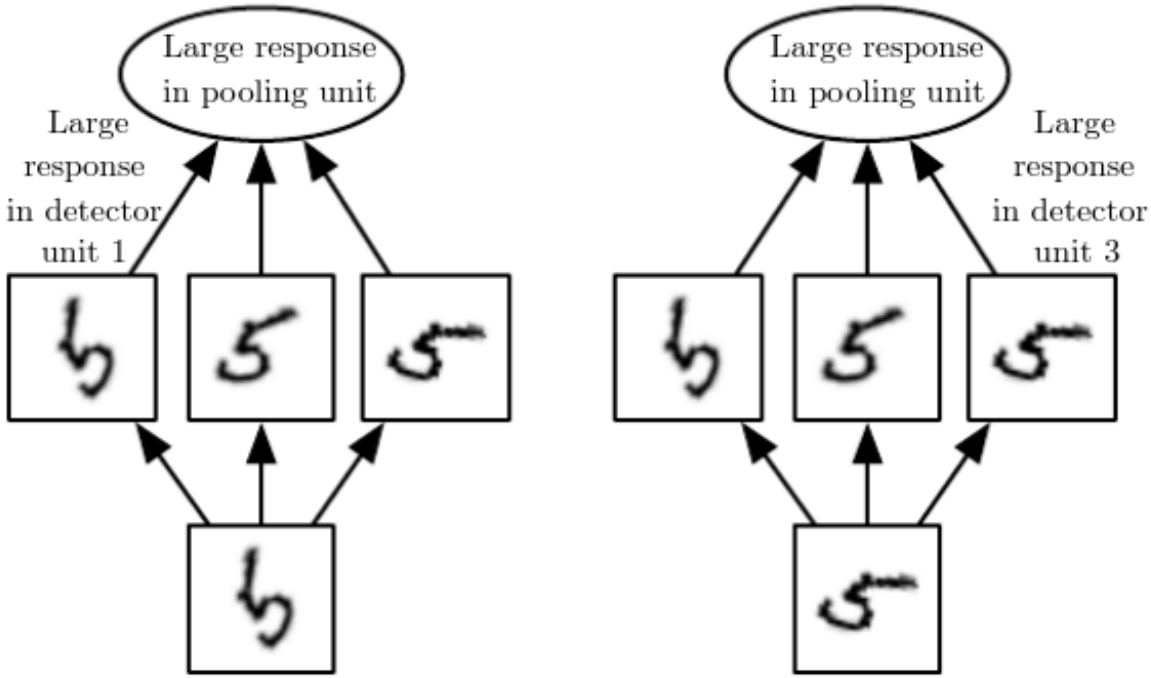
Pooling makes the output feature map approximately invariant to small translations of the input.



Goodfellow et al. (2016), Figure 9.8

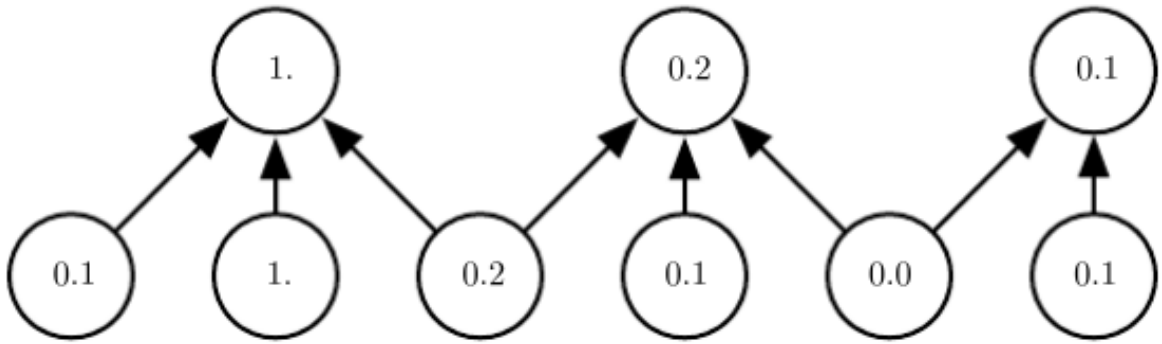
This is good when we want to know whether a feature is in the input, without knowing precisely where it is.

Pooling over features enables invariance to more complex transformations of the input, such as rotation:



Goodfellow et al. (2016), Figure 9.9

Usually, pooling is combined with downsampling, which reduces the computational and statistical burden on the next layer.



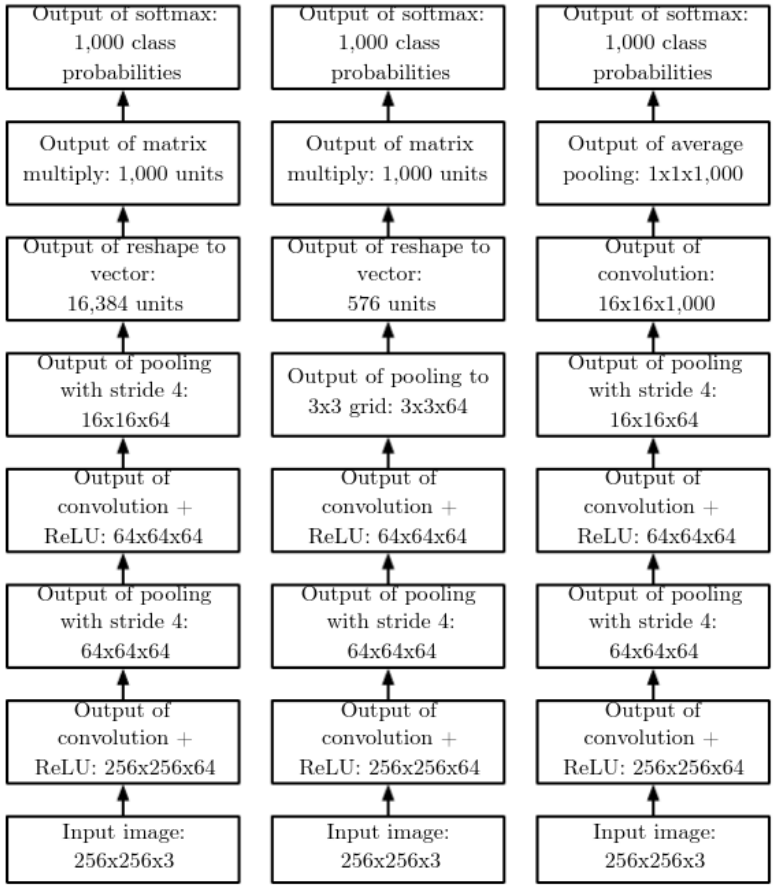
Goodfellow et al. (2016), Figure 9.10

CNNs for image classification

Putting it together

Some sample CNN architectures, as examples. Note that practical networks are deeper, more variable, and have branches.

Generally, the kinds of processing we want to do on spatial and temporal data fit the convolve + pool processing approach, but it may not always be so. Convolution and pooling may cause underfitting.



Goodfellow et al. (2016), Figure 9.11

In general, the input, kernel, and output are all tensors of arbitrary number of dimensions.

Images: 2D data with 3 channels (red, green, blue), which may be combined into mini-batches, giving a 4D tensor as the input.

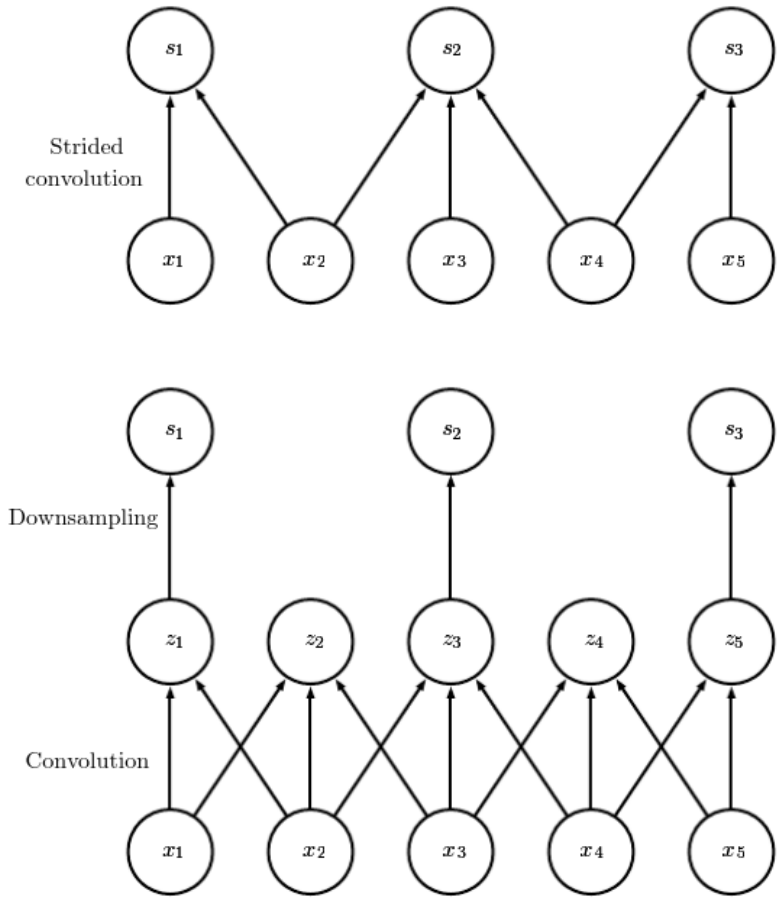
The kernel applied to these data might be a 4D tensor $\mathtt{K}_{i,j,k,l}$ giving strength of connection between input channel i, output channel j, and offset (k, l) between the output and input unit. Note that every feature map j connects with every channel i in the input.

The output would then be a 3D tensor $\mathtt{Z}_{i,j,k}$, where i represents the output channel or feature map and (j, k) indicates the spatial location.

CNNs for image classification

Refinements: downsampling via stride

We may also downsample during the convolution operation, using a stride not equal to 1. Example stride of 2:



Goodfellow et al. (2016), Figure 9.12

Stride is important:

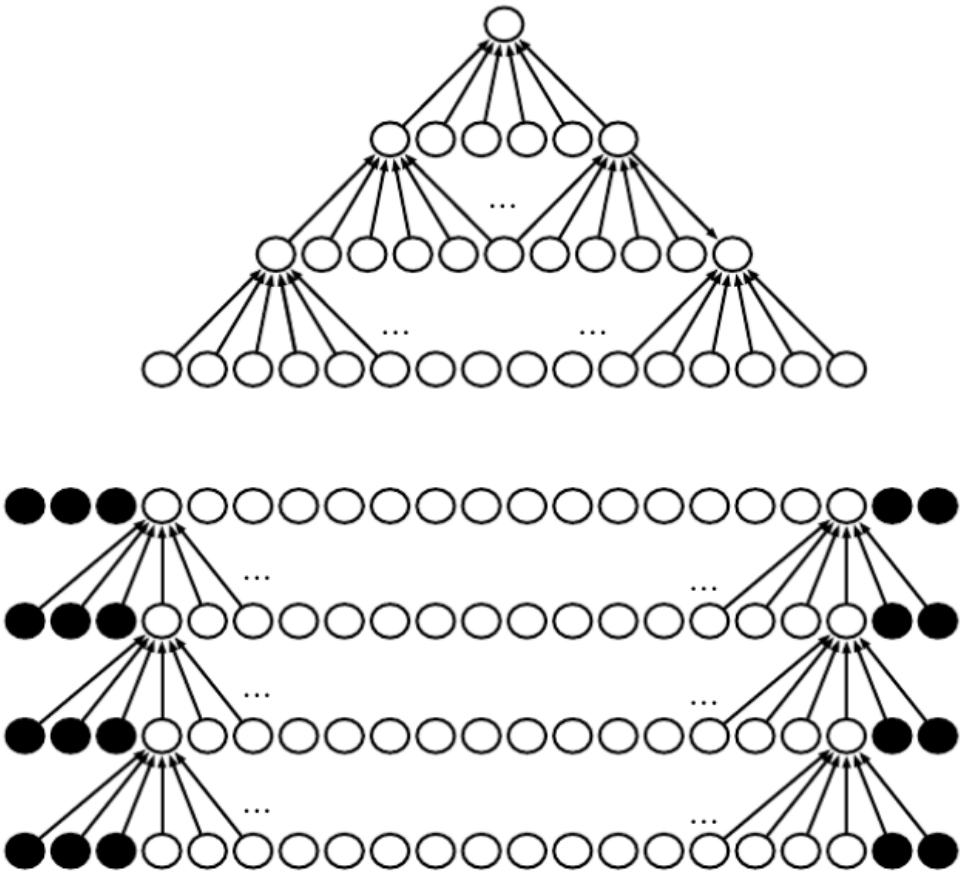
- Typically, the convolution operation is applied at every pixel of the input (stride $= 1$).
- When spatial resolution is less important or neighboring receptive fields overlap significantly, we may skip some pixels of the input (stride > 1).
- Most architectures use a stride of 1 for 3×3 convolutions 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

Refinements: padding

We may also pad the image so that the valid convolution at each layer has the same size as the input to the layer:

Usually, the padding is 0.



Goodfellow et al. (2016), Figure 9.13

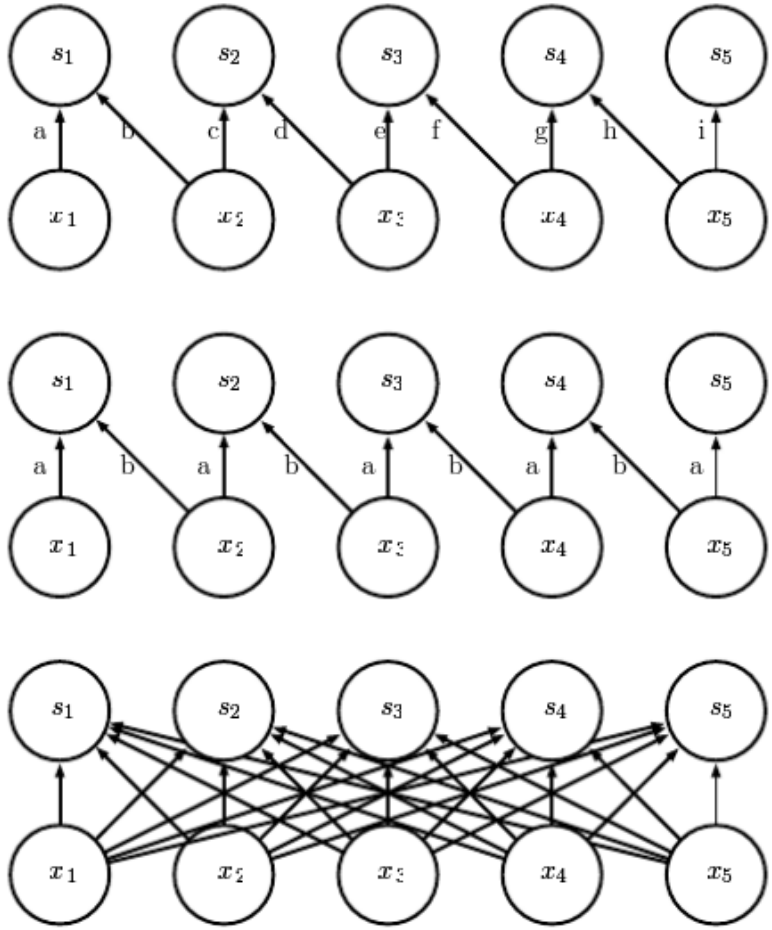
Padding is important:

- 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).

In unshared convolution, we perform the same dot product of a set of weights over a local region in the input, but do not use a shared kernel.

This makes sense when we need local sparse computation but have no reason to apply the same operation throughout the input.

Comparison of local connections, convolution, and full connection:

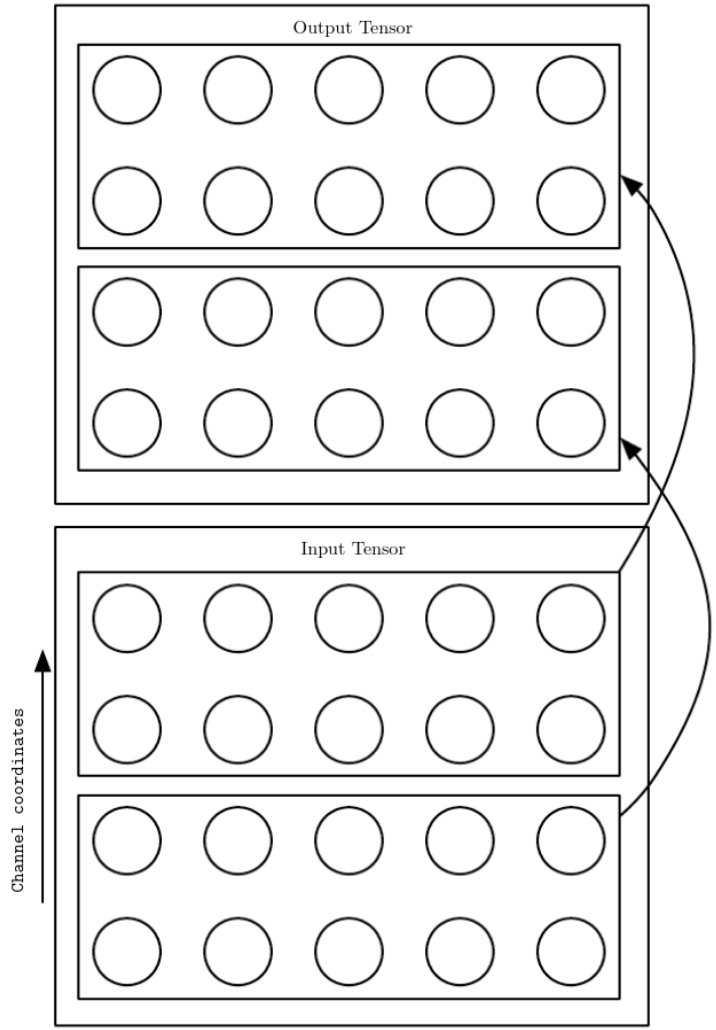


Goodfellow et al. (2016), Figure 9.14

CNNs for image classification

Refinements: organizing by channel

We may also want separate kernels for each input channel:

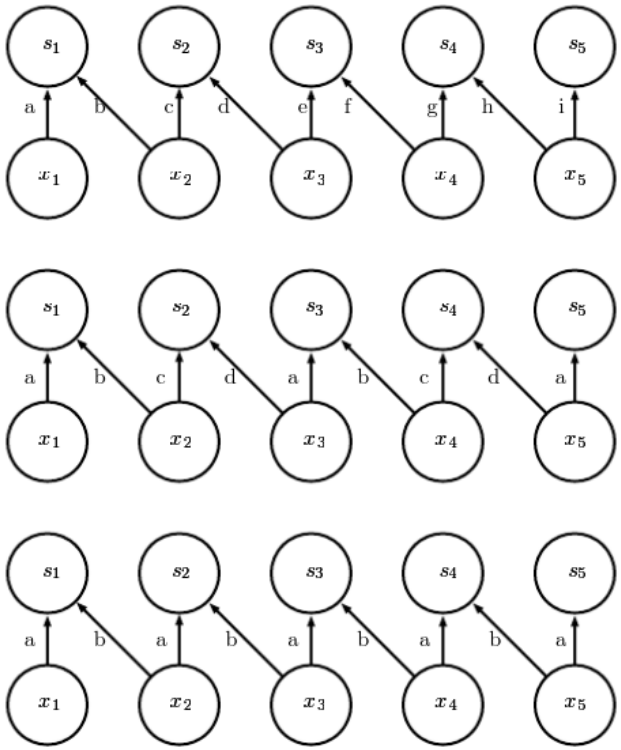


Goodfellow et al. (2016), Figure 9.15

CNNs for image classification

Refinements: tiled convolution

Another variation is tiled convolution in which we combine the idea of having separate weights for neighboring positions in the feature map but also sharing parameters:



Goodfellow et al. (2016), Figure 9.16

Transpose convolution means multiplication by the transpose of the matrix defined by convolution. It is used for

- Backpropagating error derivatives through a convolutional layer
- Reconstructing input units from hidden units in an autoencoder

Suppose we have a convolution tensor $\mathtt{K}$ applied to multichannel tensor $\mathtt{V}$ with stride s: $c(\mathtt{K}, \mathtt{V}, s)$.

Suppose the loss function is $\mathtt{J}(\mathtt{V}, \mathtt{K})$.

Forward propagation gives us $\mathtt{Z} = c(\mathtt{K}, \mathtt{V}, s)$.

$\mathtt{Z}$ is propagated forward, then during backpropagation, we receive tensor $\mathtt{G}$ where

$$\mathtt{G}_{i,j,k} = \frac{\partial}{\partial \mathtt{Z}_{i,j,k}} \mathtt{J}(\mathtt{V}, \mathtt{K}).$$

The derivative of $\mathtt{J}$ with respect to weight i, j, k, l in $\mathtt{K}$ is

$$g(\mathtt{G}, \mathtt{V}, s)_{i,j,k,l} = \frac{\partial}{\partial \mathtt{K}_{i,j,k,l}} J(\mathtt{V}, \mathtt{K}) = \sum_{m,n} \mathtt{G}_{i,m,n} \mathtt{V}_{j,(m-1)\times s+k,(n-1)\times s+l}.$$

The deltas to be backpropgated to previous layers are

$$h(\mathtt{K}, \mathtt{G}, s)_{i,j,k} = \frac{\partial}{\partial \mathtt{V}_{i,j,k}} J(\mathtt{V}, \mathtt{K})$$

$$= \sum_{\substack{l,m \\ \text{s.t.}(l-1)\times s+m=j}} \sum_{\substack{n,p \\ \text{s.t.}(n-1)\times s+p=k}} \sum_{q} \mathtt{K}_{q,i,m,p} \mathtt{G}_{q,l,n}.$$

How the bias term is done varies.

For locally connected layers, each unit would have its own bias.

For convolutional layers, usually, each channel of the output has a separate bias shared across all locations.

Sometimes, each element in the output will have its own bias. This is useful for example when we use 0 padding and border elements receive less input than interior elements.

Besides classification, convolutional networks can also be used to produce a structured output, itself a multidimensional tensor such as an image.

Besides images, the input data could be a single channel audio waveform, or a sequence of feature vectors corresponding to samples over time, or a 3D volume, or a color video,

To make convolution efficient, we may use parallel hardware (GPUs).

In some cases, kernels may be separable, e.g., a 2D convolution is the composition of two 1D convolutions in the different dimensions.

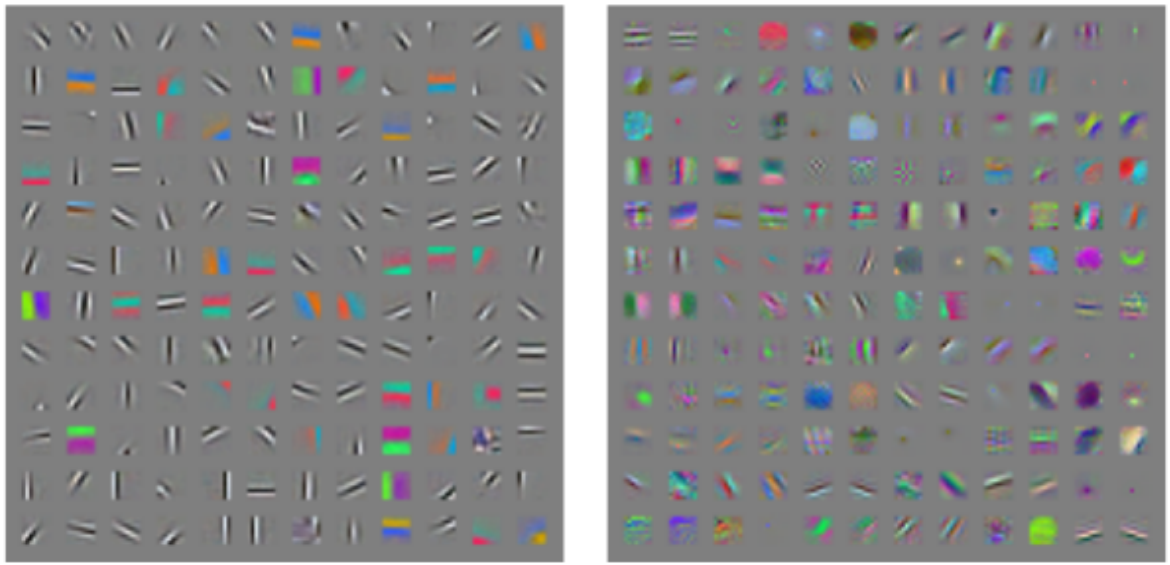
How to obtain convolution kernels without expensive backpropagation?

- Random kernels
- Design by hand (e.g., edge detectors or Gabor filters)
- Learn with an unsupervised criterion, e.g., k-means clustering of image patches in the training set.

CNNs for image classification

Connections with brain science

There is an interesting correspondence between low-level processing in well-trained convolutional neural networks for visual tasks and the kinds of visual feature extractors known to be present in the mammalian visual system:

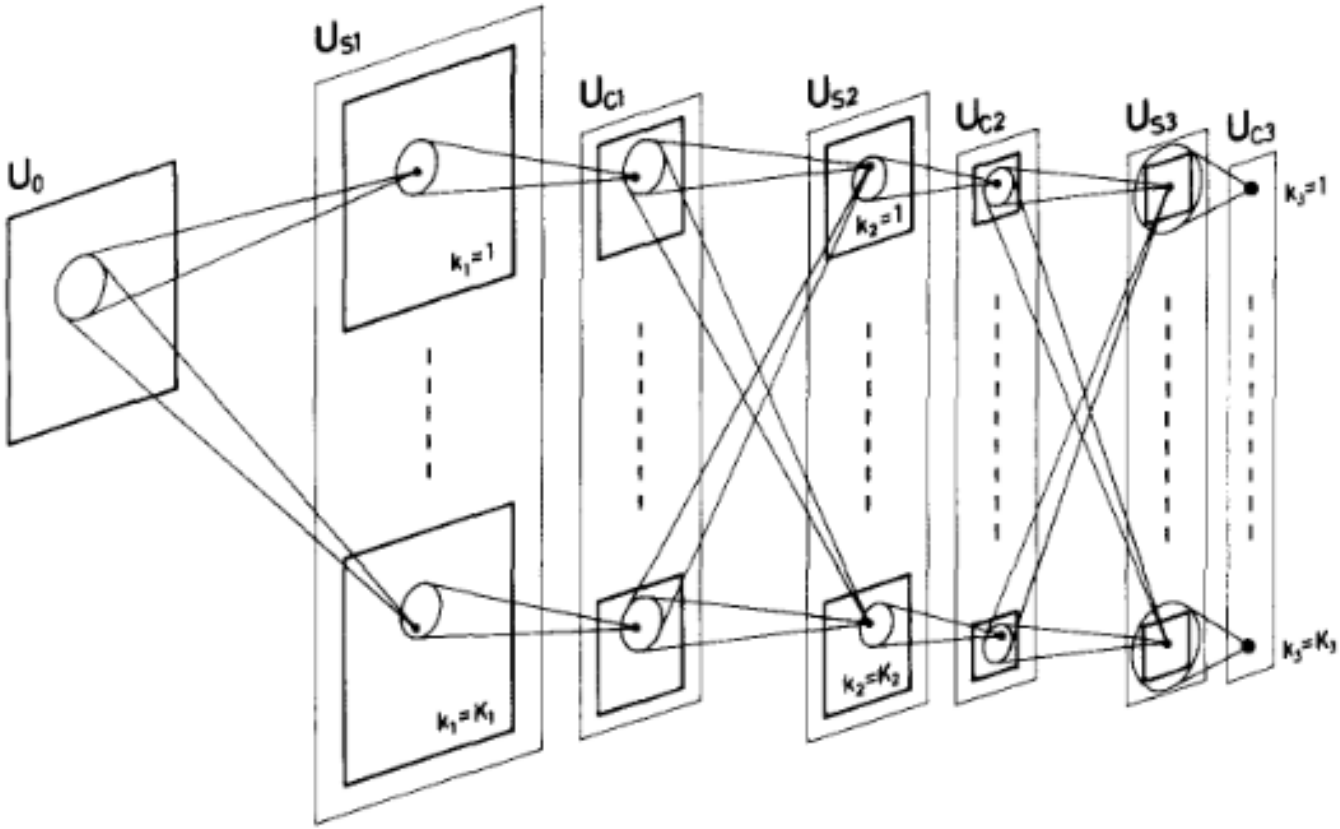


Goodfellow et al. (2016), Figure 9.19

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

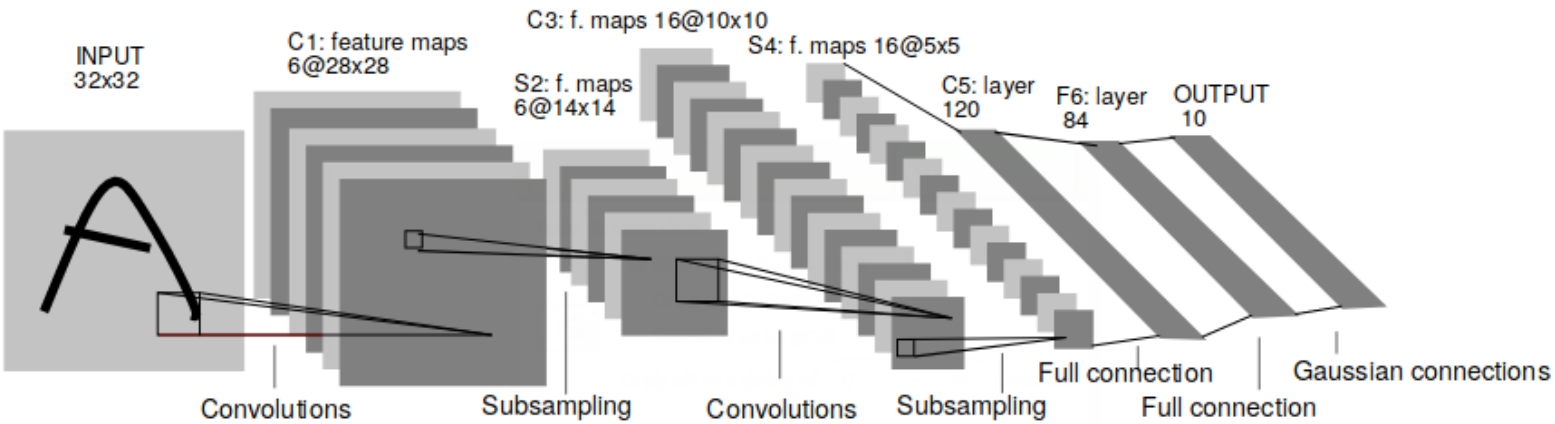
CNNs for image classification

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 approaches to visual object recognition such as HOG (histograms of oriented gradients).

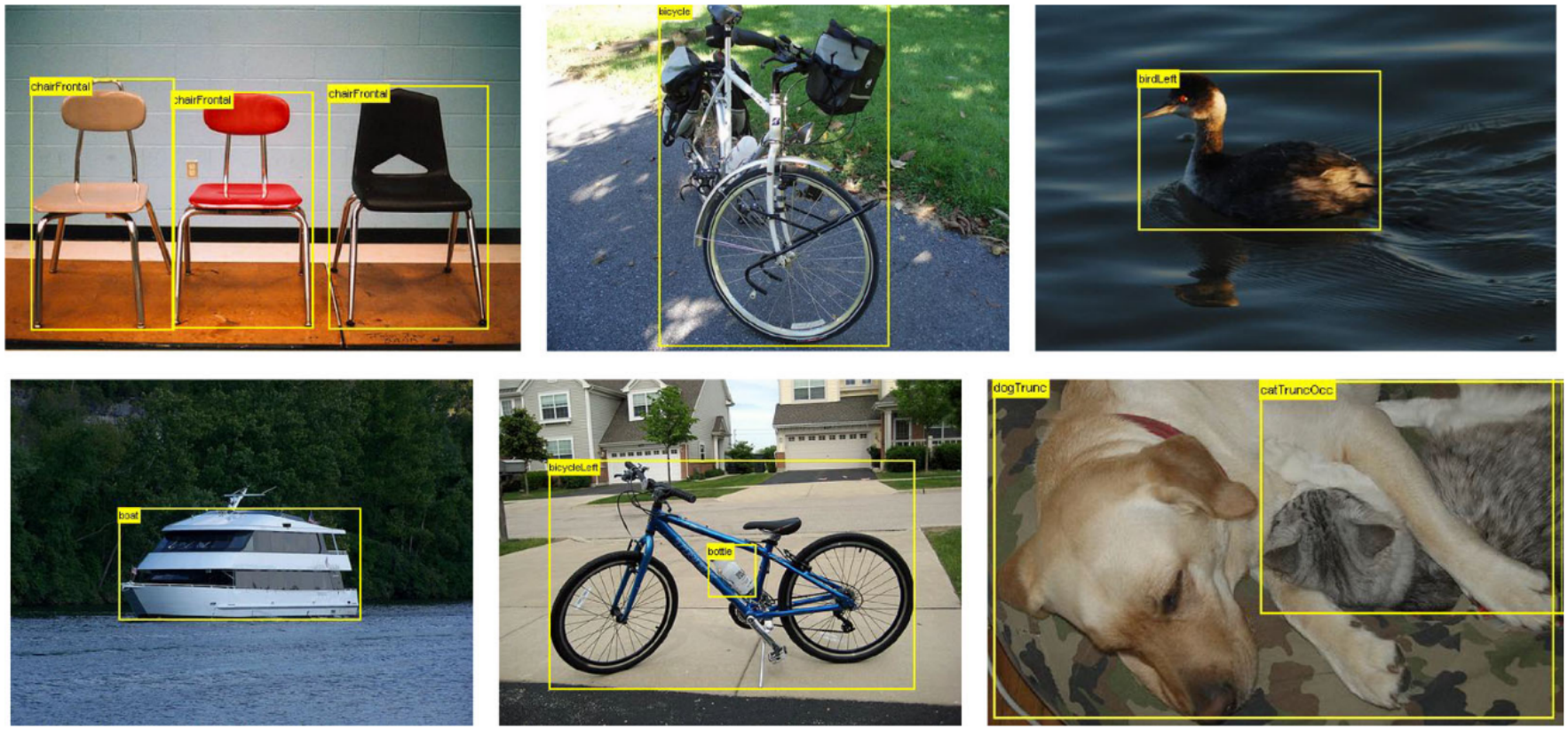
Some research on neural networks continued, but most vision researchers ignored CNNs.

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

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

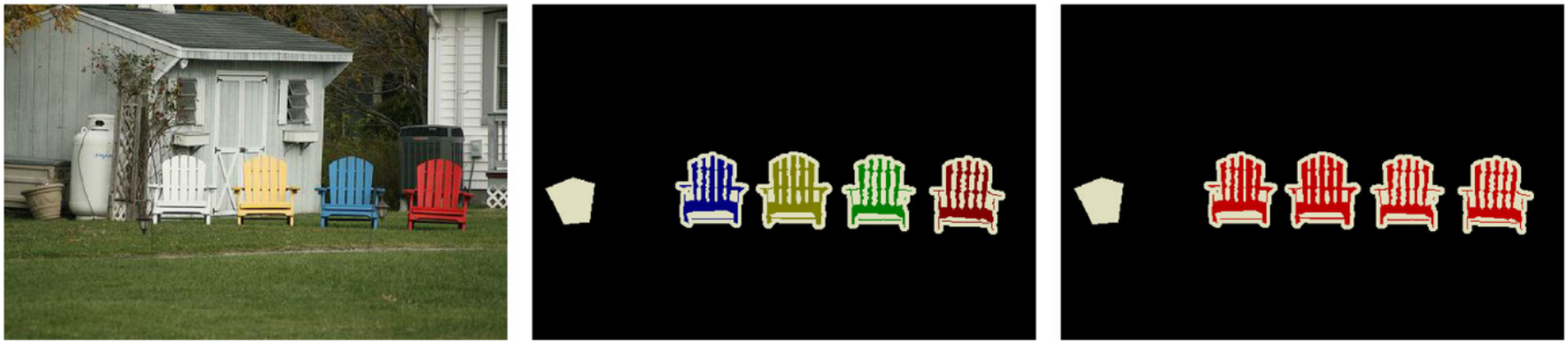
"Simple" feature based methods like HOG could not cope with VOC.

Sample VOC object detection and classification images:

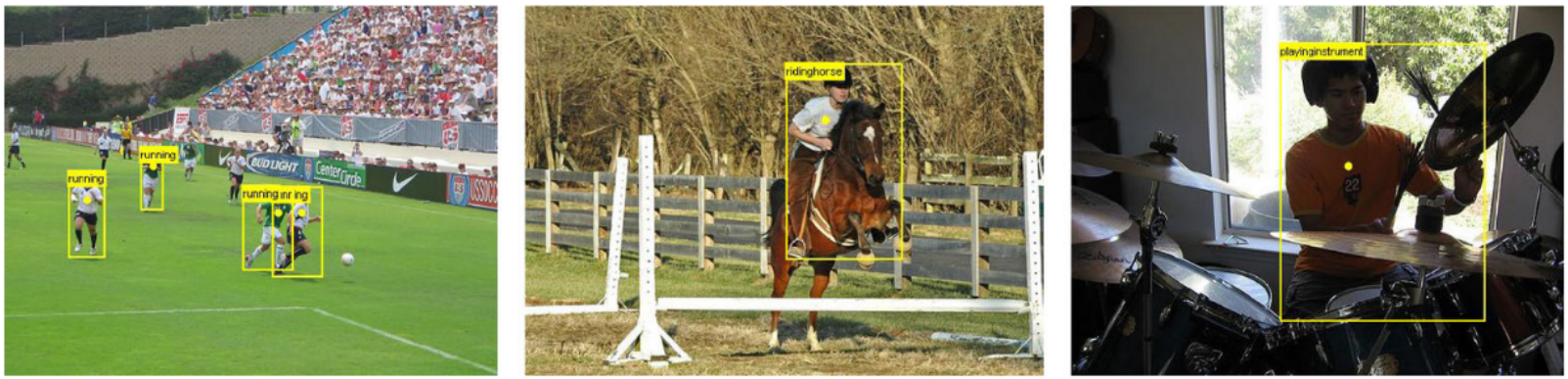


Everingham et al. (2015), Fig. 1a

Sample VOC segmentation and action classification images:



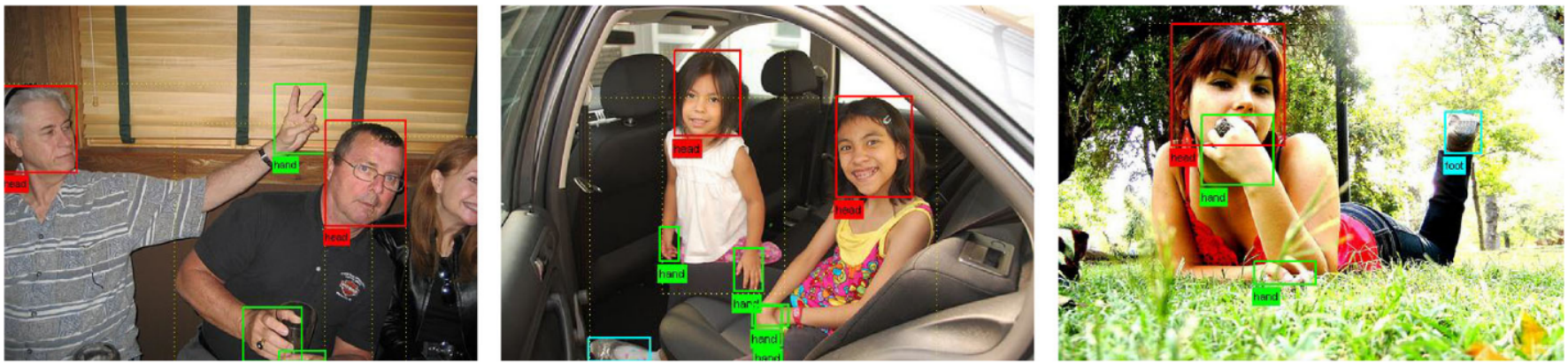
(b) Segmentation



(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.

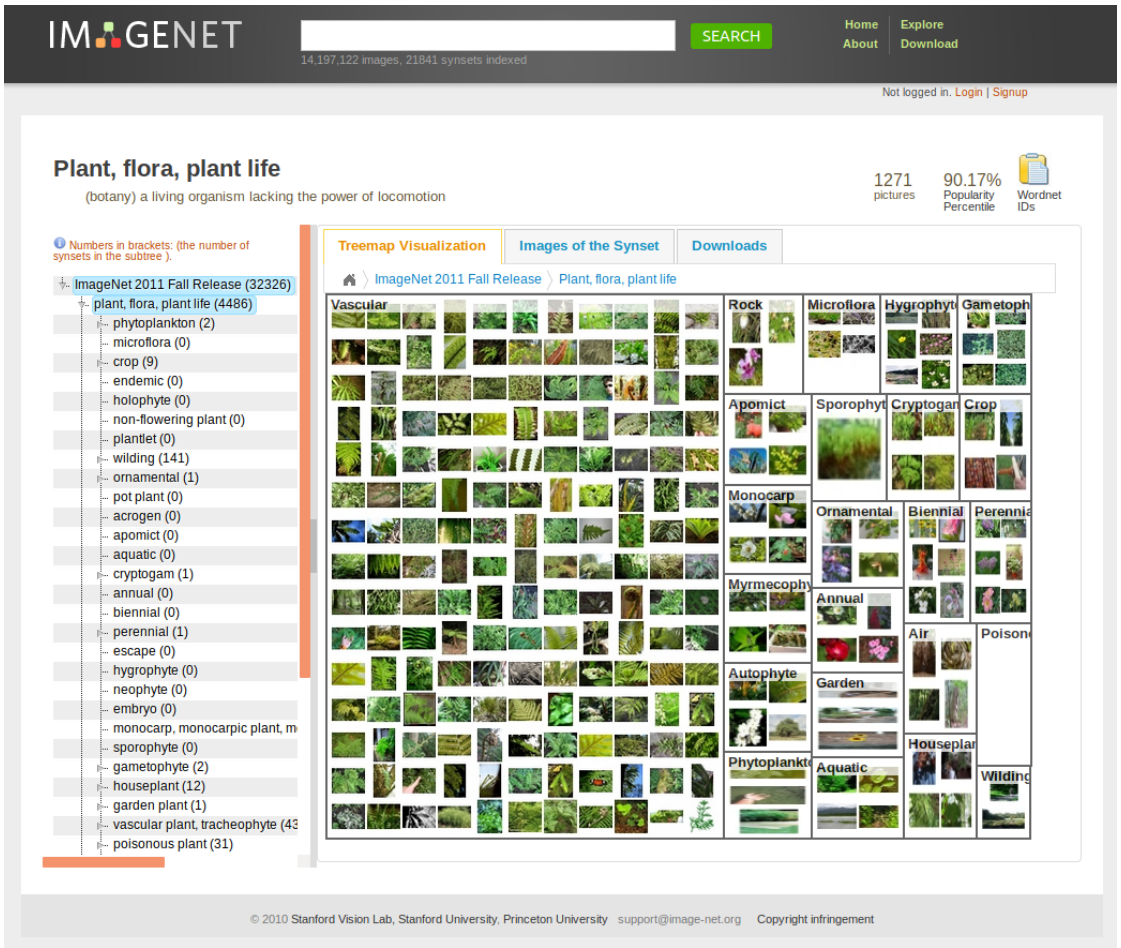
Li failed to get much funding for the project in the beginning but found that Amazon Mechanical Turk could be used to get humans to label images relatively cheaply.

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.

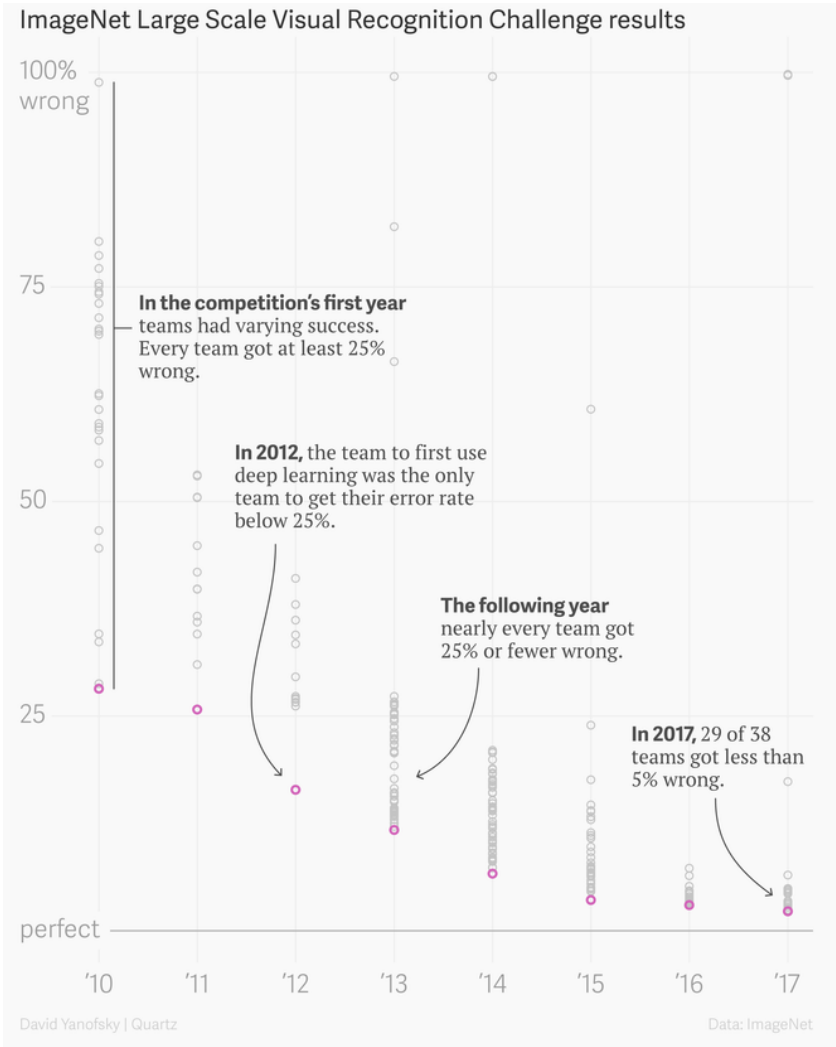
Samples from ImageNet:



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] Recall 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.

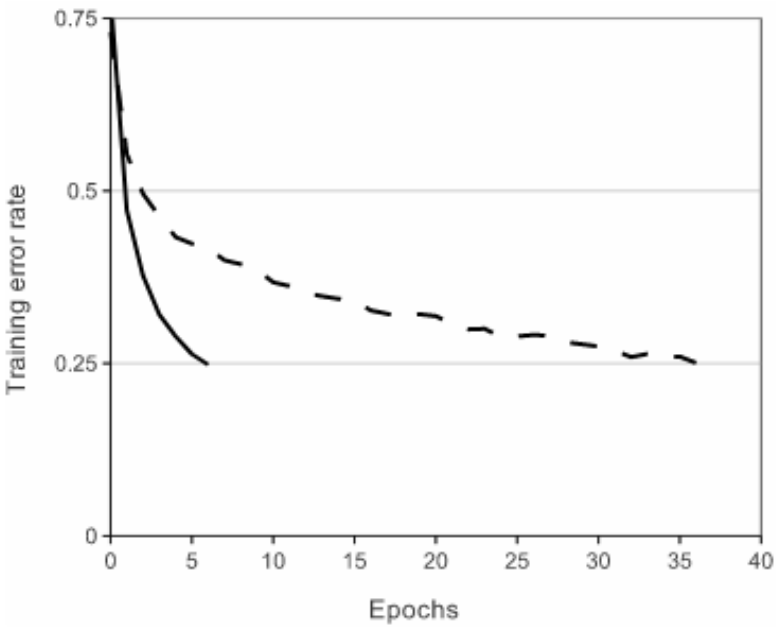
Training required a highly efficient implementation of the learning and runtime operations on GPUs with C++ and CUDA.

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

AlexNet (2012)



Krizhevsky et al. (2012), Fig. 1

The authors were among the first to exploit the benefits of ReLU over other nonlinear activation functions.

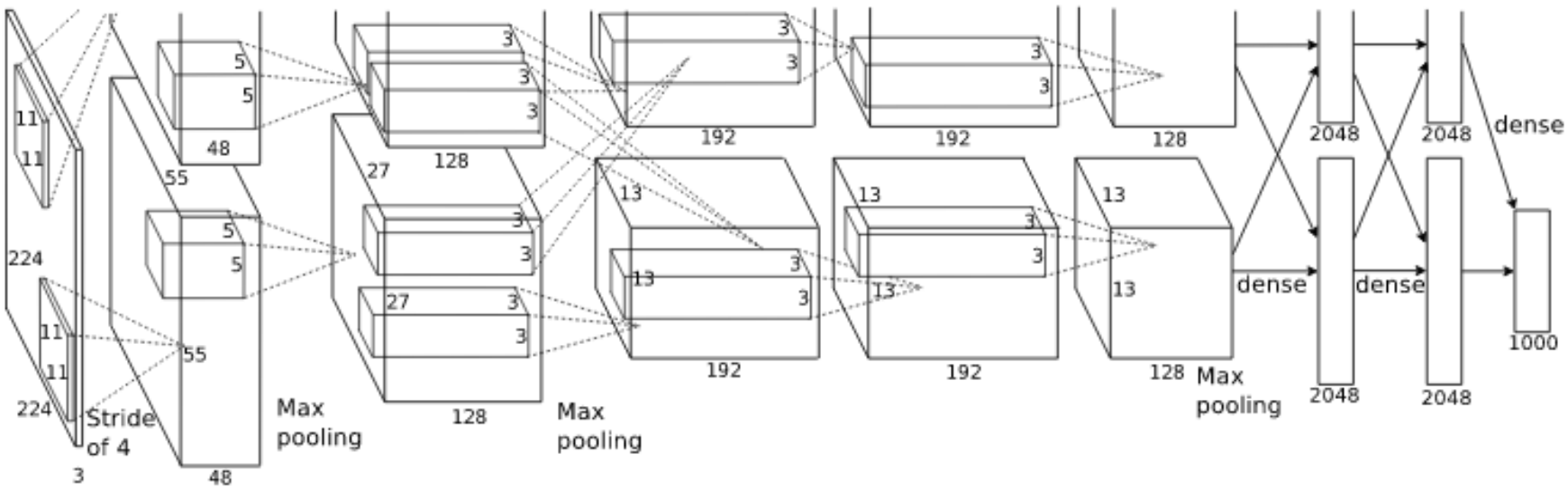
ReLU models learn much faster than tanh models. tanh "saturates" at large absolute values (learning depends on the slope of the activation function).

Training a 4-layer network on CIFAR 10: solid line shows error rate with ReLU, dashed line shows error rates with tanh.

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

CNNs for image classification

AlexNet (2012)

After ReLU, the next most important technique used is local response normalization, which reduces errors by more than 1%.

Non-saturating activation functions means we can have very large activations for some inputs.

Response normalization reduces large activations while preserving relative relationships between different feature maps.

Another term for this is local inhibition, a property seen in real neural circuits.

Letting $a^i_{x,y}$ represent the ReLU activation feature map i at location (x, y), the normalized response is

$$b^i_{x,y} = a^i_{x,y} \left(k + \alpha \sum_{j=\max(0,i-n/2)}^{\min(N-1,i+n/2)} \left(a^j_{x,y}\right)^2 \right)^{\beta}$$

We are normalizing each unit relative to its n neighboring features.

Another technique is overlapping pooling.

Before AlexNet, pooling usually used a stride equal to the width of the pooling region, so that neighboring pooled units did not have overlapping receptive fields.

The authors find, however, that overlapping pooling is effective. They use a stride of 2 and a pooling region of size 3x3.

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) fully connected 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.

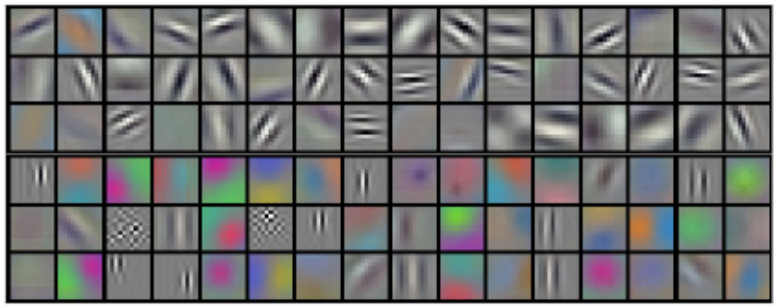
Training parameters:

- 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).

CNNs for image classification

AlexNet (2012)

The first convolutional layer learns representations reminiscent of neurons in visual cortex:



Krizhevsky et al. (2012), Fig. 3

This is after 90 epochs through the 1.2 million images in ImageNet.

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

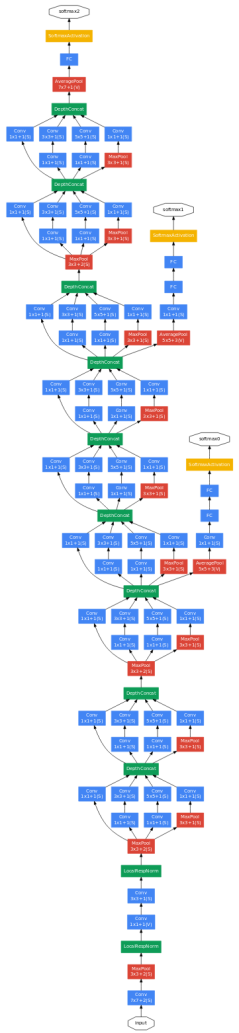
GoogLeNet (2014)

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)

Before Inception, accuracy was achieved by larger networks with more feature maps and more parameters.

Overfitting is avoided by aggressive data augmentation and dropout.

Inception idea: can we have deeper, wider networks with a fixed computational budget?

Goal: 1.5 billion multiply-adds at inference time.

This was achieved with a 22-layer (27 including pooling layers) model with a modular structure.

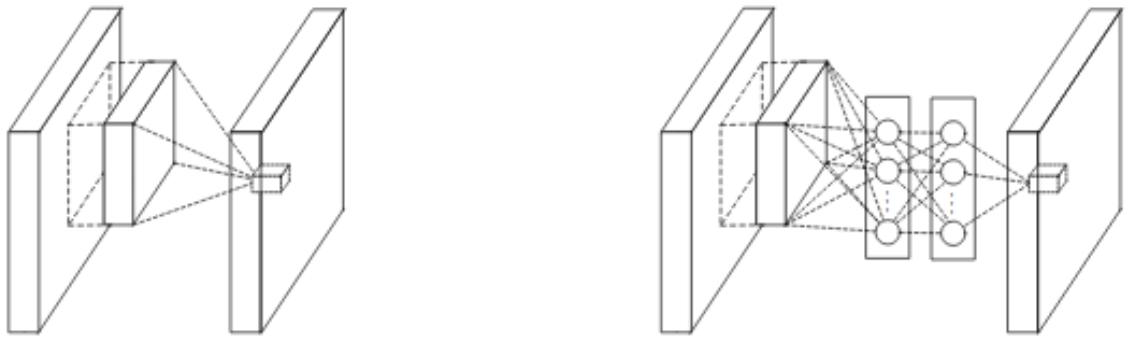
The module idea was inspired by "Network in Network."

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

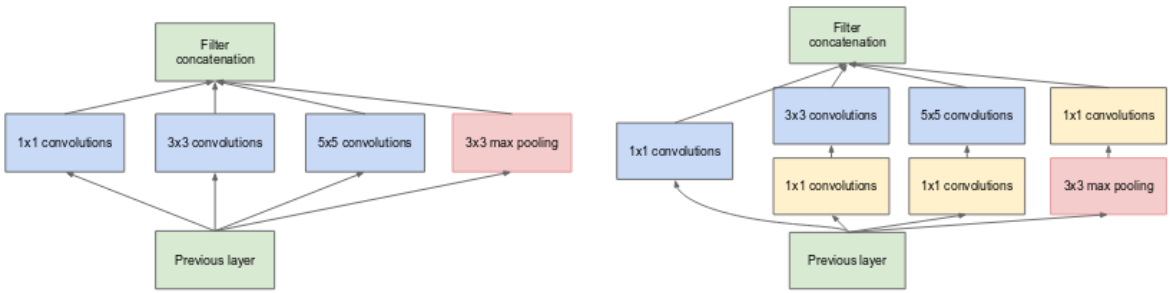
CNNs for image classification

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

CNNs for image classification

GoogLeNet (2014)

1x1 convolutions occur both before larger 3x3 and 5x5 convolutions, and after them.

Before: a reduction step that reduces the number of feature maps the 3x3 or 5x5 operates over, making them more efficient.

After: a project step where redundancy is removed and output dimensionality is reduced.

Besides an approximation to network-in-network, the architecture is intended to approximate theoretically optimal construction of large, sparse, locally connected layers that progressively cluster correlated features at previous layers.

Proportion of 3x3 and 5x5 convolutions increases at later layers.

Pooling layers are used, but unpooled features are also propagated, to get both spatial accuracy and translation invariance.

Auxiliary classifiers are used at intermediate layers to increase their discrimination ability.

Training was CPU-only, with SGD, Polyak averaging, multiple models at prediction time, and aggressive data augmentation.

The model was also used for detection in ILSVRC 2014, applying selective search for region proposals and GoogLeNet to classify those regions, without bounding box regression.

CNNs for image classification

GoogLeNet (2014)

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

Inception v3 (2016)

After 2014, research on image classification network continued at a fast pace.

Several labs continued to improve their models to perform better at ILSVRC.

Szegedy and colleagues increased GoogLeNet's size to 5 billion multiply-adds, trying to use that budget as efficiently as possible.

Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., and Wojna, Z. (2016), Rethinking the Inception architecture for computer vision, *CVPR*.

Some principles:

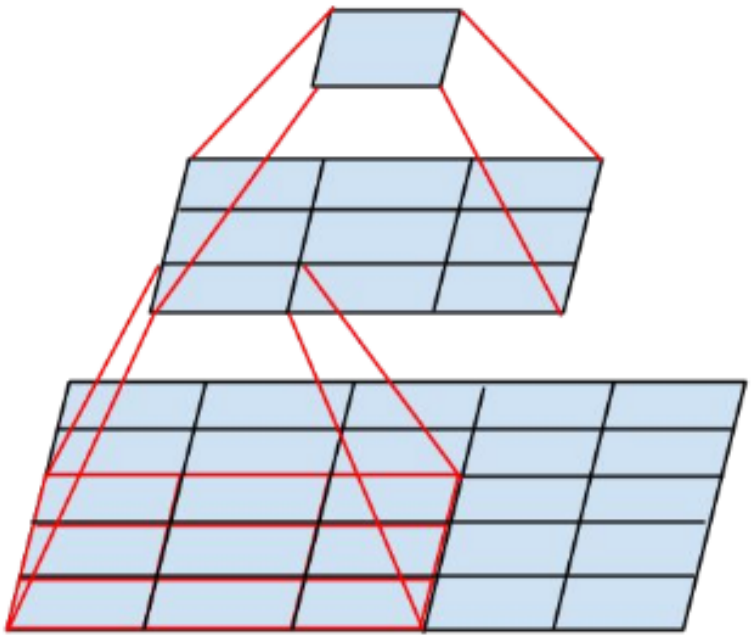
- Avoid information bottlenecks early in processing. Decrease dimensionality gradually as we move deeper.
- Keep dimensionality high for local processing.
- Reduce dimensionality before performing spatial aggregation.
- Balance the width and the depth of the network. Increased capacity should be achieved by increasing both depth and width.

CNNs for image classification

Inception v3 (2016)

The authors discuss a variety of ways of reducing computation without reducing representational capacity.

Example: replacing a 5x5 convolution with a hierarchy of two 3x3s:



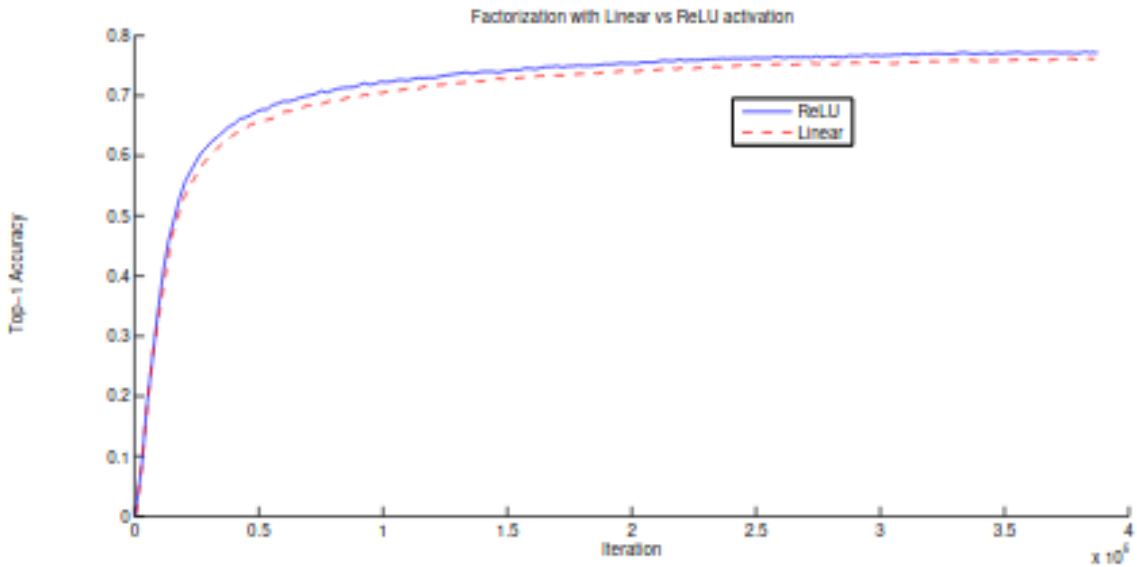
Szegedy et al. (2016), Figure 1

CNNs for image classification

Inception v3 (2016)

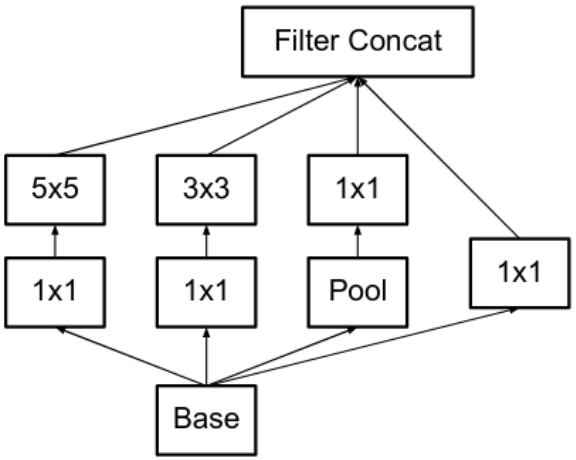
When we factorize convolutions in this way, should we apply ReLU to every layer or only at the end?

Experiments show that ReLU at every step is better than linear activations in the factored layers:

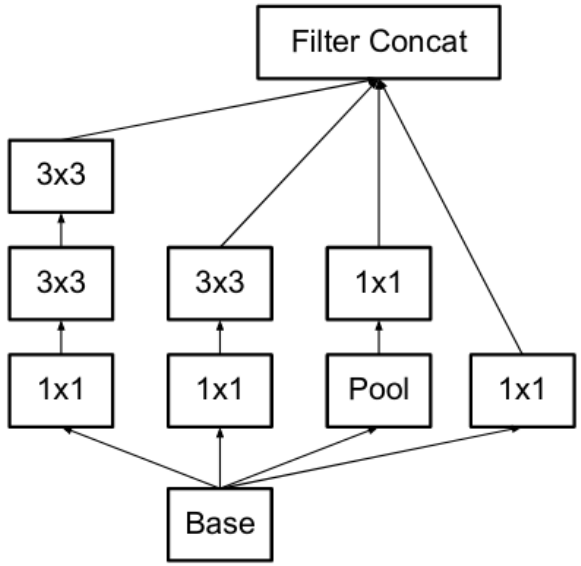


Szegedy et al. (2016), Figure 2

The 5x5 factorization gives a new structure for the basic Inception module:



Szegedy et al. (2016), Figure 4



Szegedy et al. (2016), Figure 5

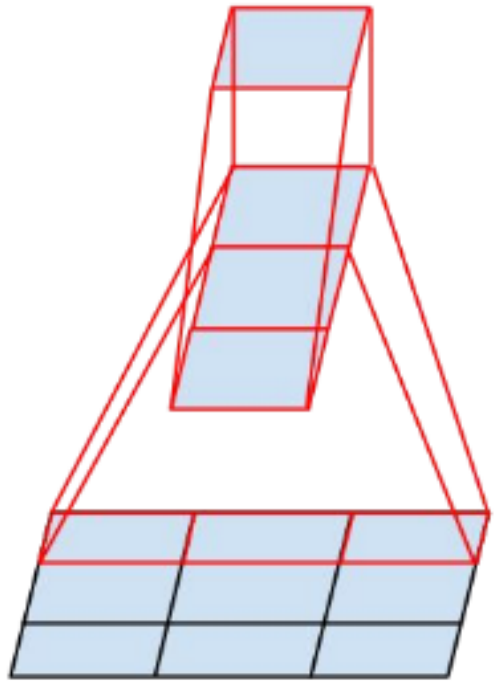
CNNs for image classification

Inception v3 (2016)

Another way of factorizing: replacing a 3x3 with a 3x1 followed by a 1x3.

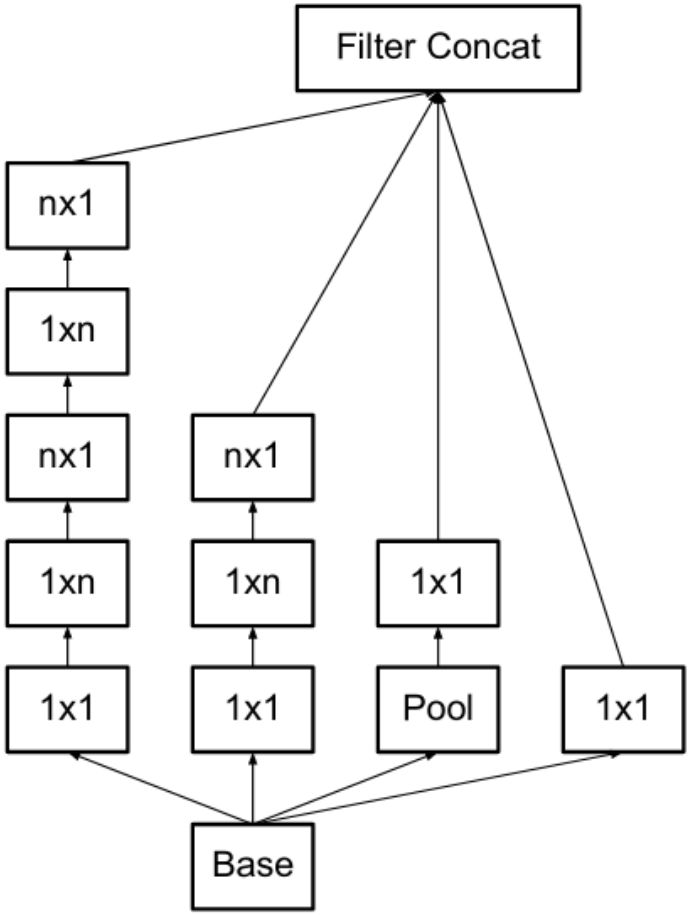
This process can go on; any nxn convolution can be factored into a nx1 followed by a 1xn.

The authors find this is not very useful in early layers, but is effective for medium-sized grids of 12-20 units wide.



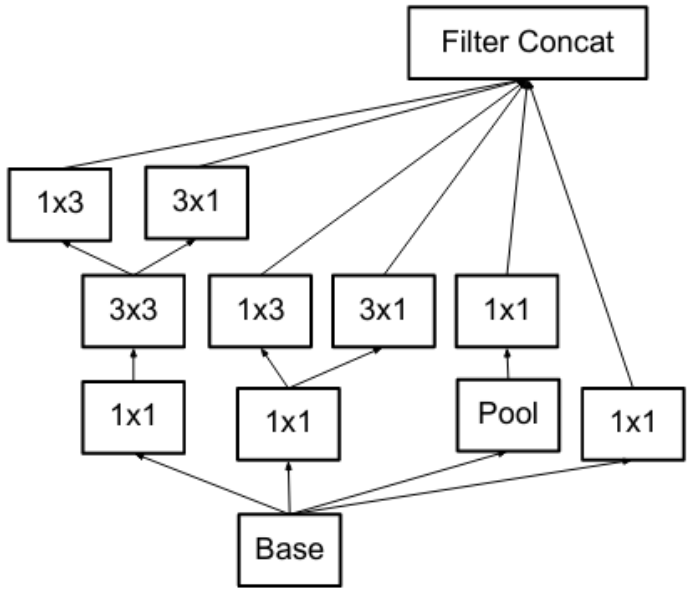
Szegedy et al. (2016), Figure 3

Inception module for medium-sized grids:



Szegedy et al. (2016), Figure 6

Inception module for coarsest grids:



Szegedy et al. (2016), Figure 7

CNNs for image classification

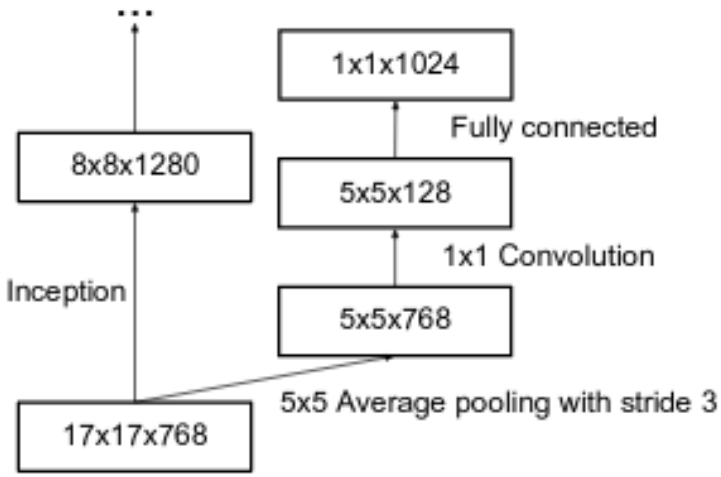
Inception v3 (2016)

The side classifiers from GoogLeNet turn out not to be as effective as first thought.

Removing one of the two side classifiers does not adversely affect performance.

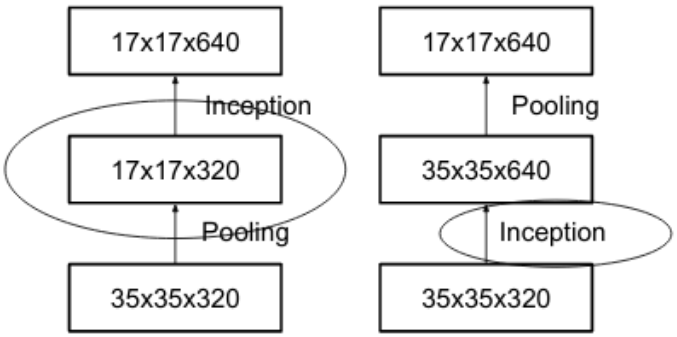
The side classifier seems to work as a regularizer rather than helping to create more discriminative features in early layers.

Auxiliary classifier on top of the last 17x17 feature map:



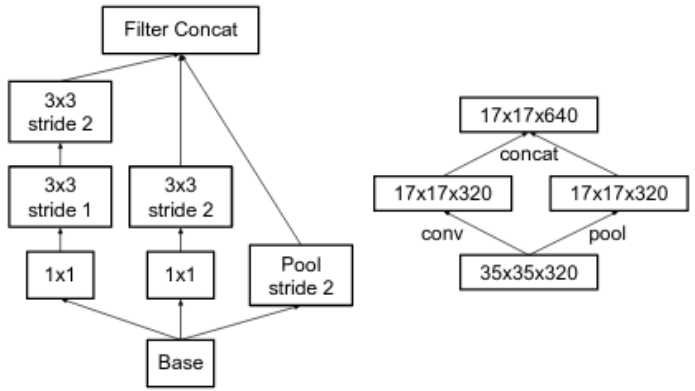
Szegedy et al. (2016), Figure 8

There are multiple ways of reducing grid size. The left structure introduces a bottleneck. The right structure avoids the bottleneck but introduces more computation.



Szegedy et al. (2016), Figure 9

Alternative: perform pooling with stride 2 and convolution with stride 2 in parallel:



Szegedy et al. (2016), Figure 10

Putting all these ideas together resulted in Inception-v2, a 42-layer extension of GoogLeNet.

Additional features leading to Inception-v3:

- RMSprop optimizer
- Label smoothing: rather than one-hot ground truth, we mix the one-hot distribution with the prior distribution over the classes.
- Batch normalization of FC layers and conv layers in the side network.

Result was the highest single-crop top-1 accuracy on ILSVRC to date.

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.

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.

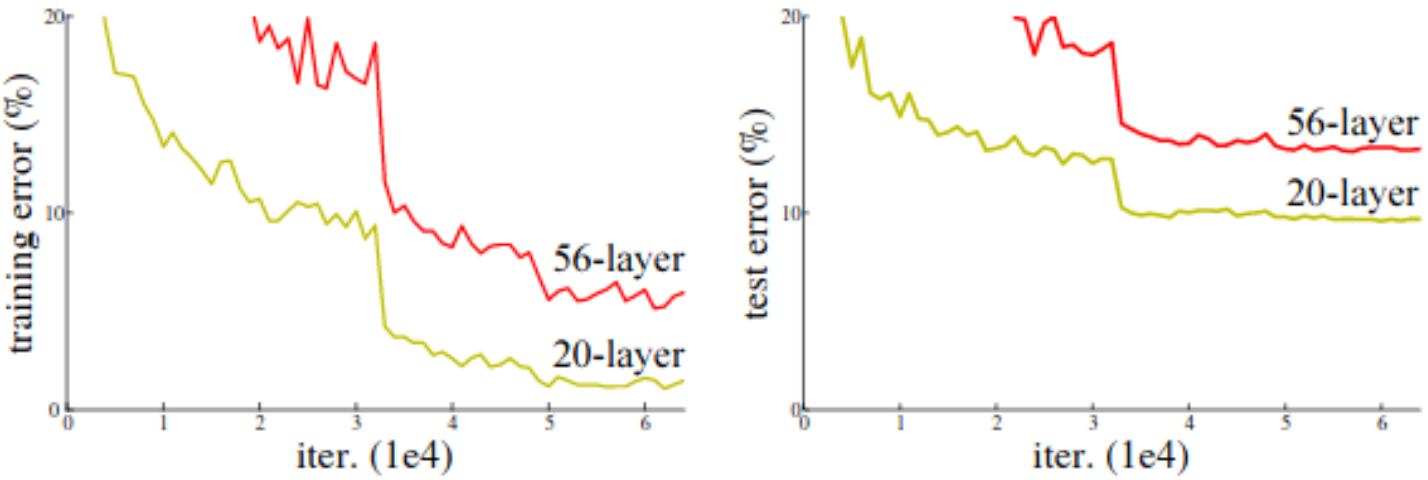
ResNet pushes the notion that "more depth is better" to the extreme.

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.

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

CNNs for image classification

ResNet (2015)



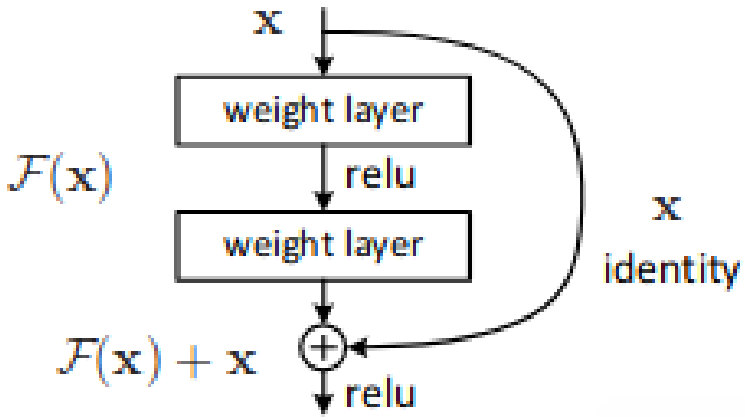
He et al. (2016), Fig. 1

As we are talking about training error, the degredation is not overfitting.

A deeper model should be at least as good as its shallow cousin (think about constructing a deep network from a shallow one by adding identity mappings).

Degredation is due to the vanishing integrity of the training signal as the network gets deeper.

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 $\mathcal{H}(\mathrm{x})$ 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)

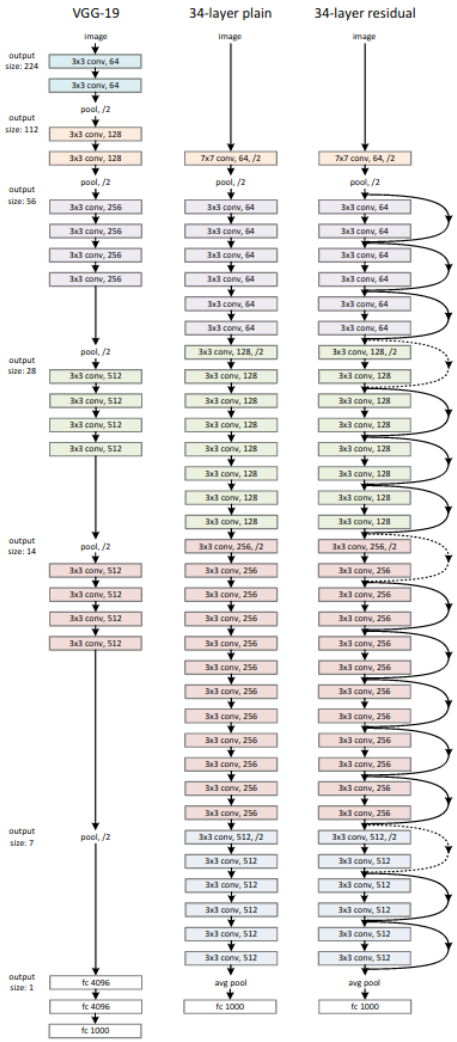
When a subnetwork changes the dimensionality of input x, then a dimensionality changing mapping is used instead of the identity mapping.

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

ResNet (2015)



He et al. (2016), Fig. 3

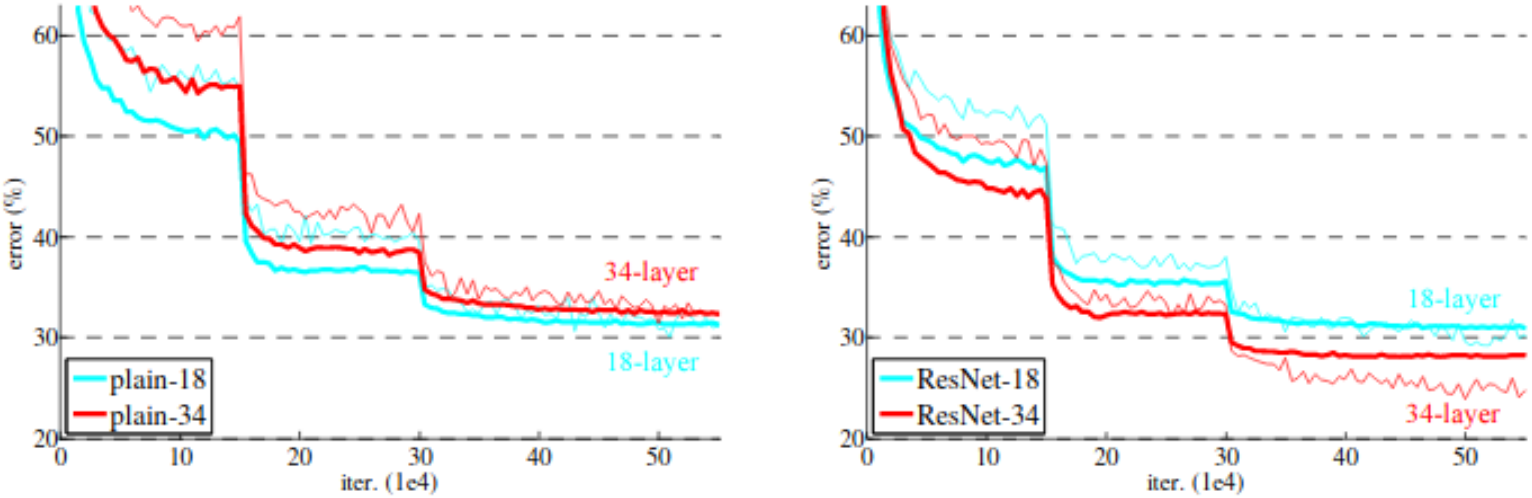
Baseline model (middle) is a 34-layer network

- Mostly 3×3 convolutions
- Equal-size mappings always have the same number of filters
- Mappings that downsample do so by half, with double the number of filters

Shortcut connections in residual version (right) are performed by 1×1 convolutions with stride of 2.

CNNs for image classification

ResNet (2015)



He et al. (2016), Fig. 4

Left: plain networks with 18 or 34 layers. Right: residual networks with 18 or 34 layers.

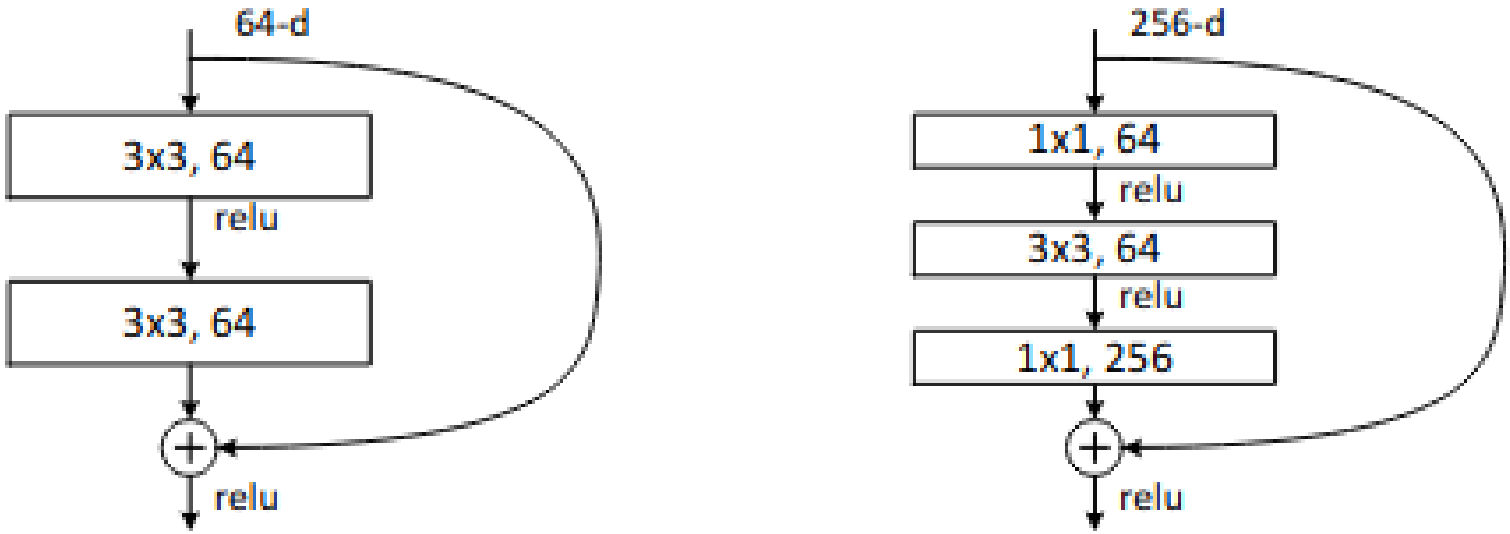
Residual layers give better convergence for the small network and better accuracy for the larger network. Only the larger residual model pushes training error lower than validation error.

Identity or projection? Experiments show not much impact of either choice.

The authors decided to use identity for same-size mappings and projection for decreased-size mappings.

CNNs for image classification

ResNet (2015)



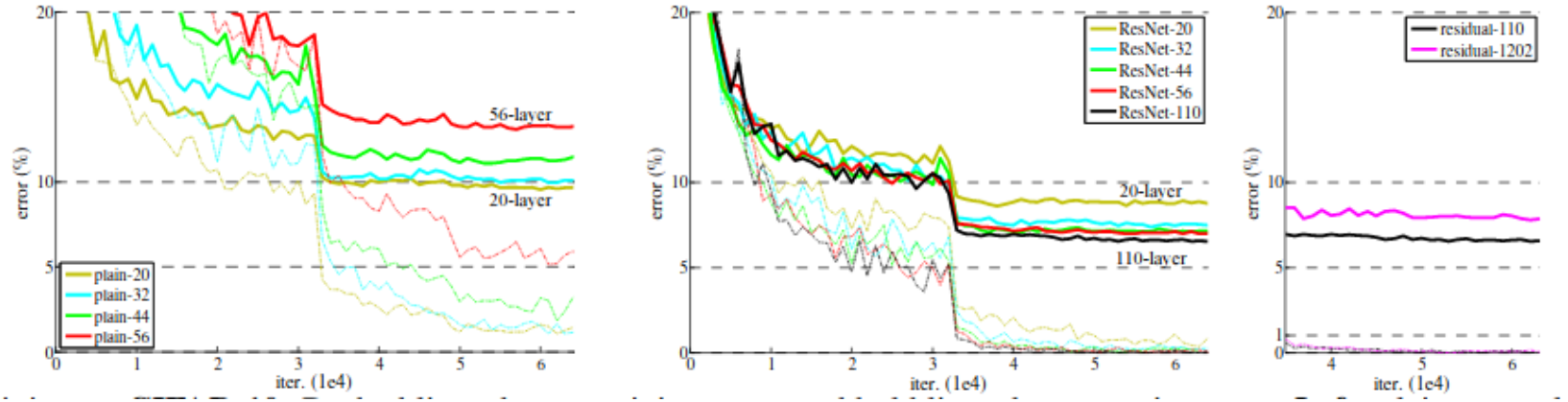
He et al. (2016), Fig. 5

Ordinary residual block (left) and bottleneck residual block (right).

Bottlenecks do not seem to hurt performance and are more economical in terms of calculations and number of parameters, so are used in the 152-layer ResNet.

CNNs for image classification

ResNet (2015)



He et al. (2016), Fig. 6

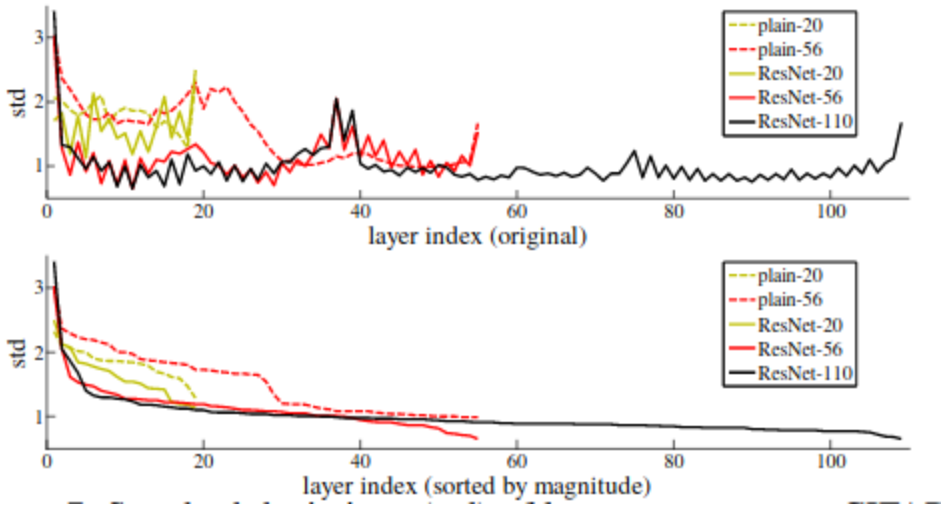
CIFAR results. Left: plain networks perform worse with more layers, both training and test.

Middle: residual networks perform better with more layers, both training and test.

Right: extremely large residual networks with more than 1000 layers learn well but exhibit overfitting.

CNNs for image classification

ResNet (2015)



He et al. (2016), Fig. 7

Top: standard deviation of layer output responses after BN but before skip connection adding and ReLU.

Bottom: same data, sorted.

Residual layers are less active than plain layers.

CNNs for image classification

Inception-ResNet (2016)

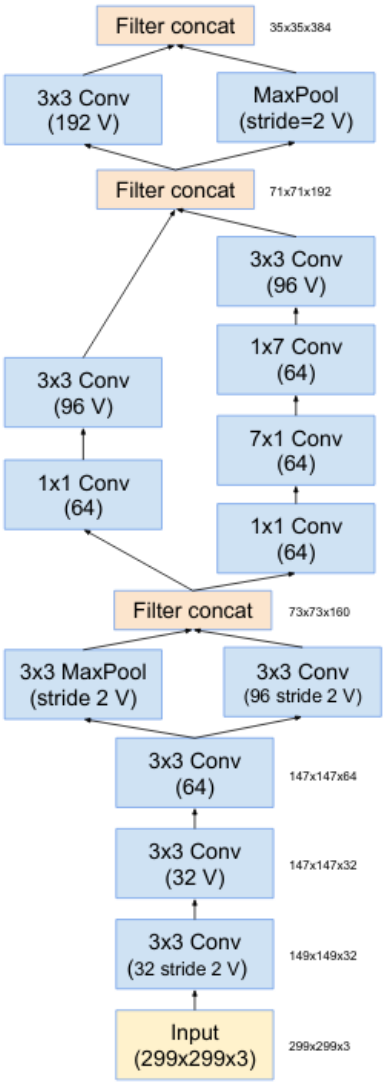
The Inception group, inspired by the success of ResNet in ILSVRC 2015 and other competitions, considered whether residual connections can improve Inception.

Szegedy, C., Ioffe, S., Vanhoucke, V., and Alemi, A. (2017), Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning. *Association for the Advancement of Artificial Intelligence Conference on AI (AAAI)*.

A version of Inception-ResNet achieved a 3.08% top-5 error rate on ILSVRC 2016 (but was beaten by Trimps-Soushen with 2.99%).

CNNs for image classification

Inception-ResNet (2016)



Szegedy et at. (2017), Fig. 3

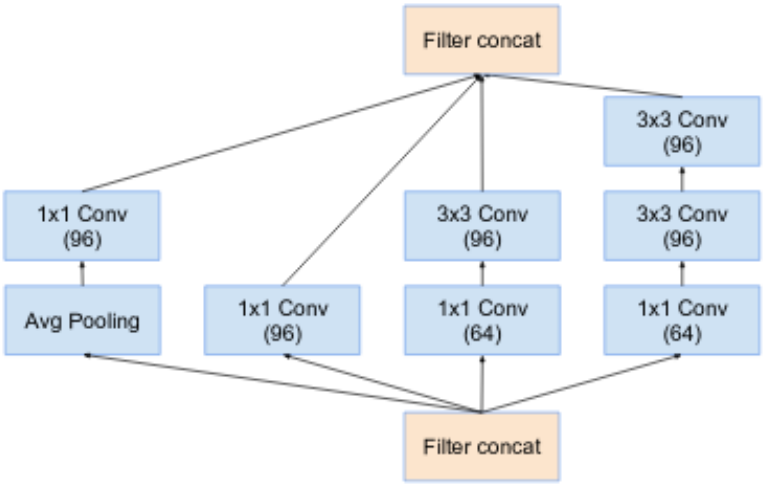
Stem network used at the input of Inception v4 and Inception-ResNet v2.

No "V" means the input is same-padded so that output is same size as input.

"V" means valid only.

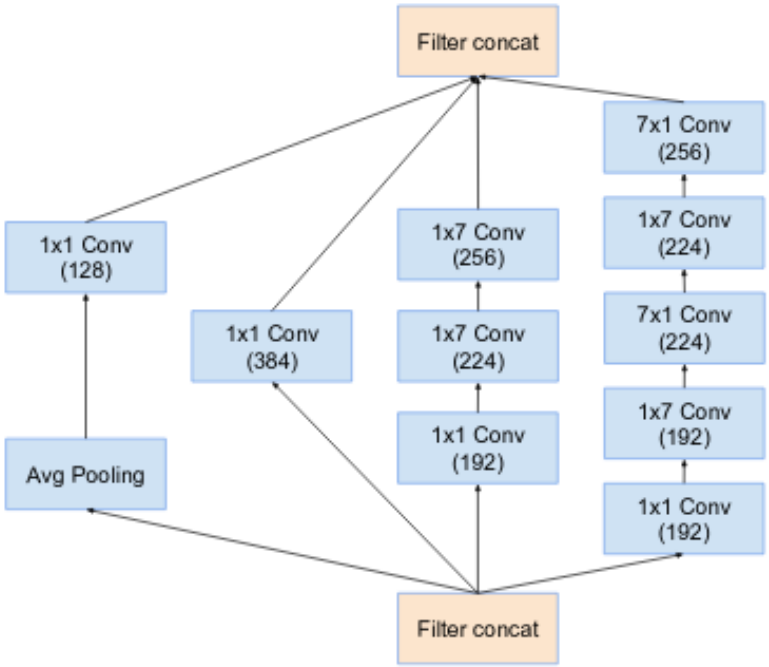
CNNs for image classification

Inception-ResNet (2016)



Szegedy et at. (2017), Fig. 4

35×35 Inception module for Inception v4 (Inception A).

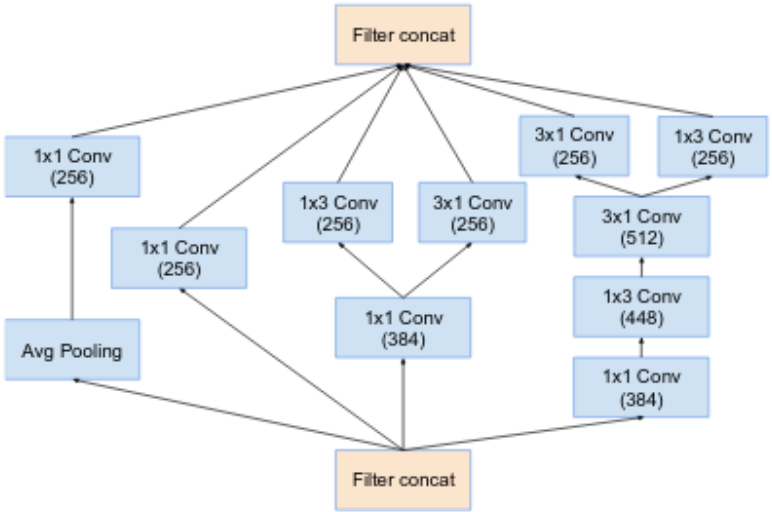


Szegedy et at. (2017), Fig. 5

17×17 Inception module for Inception v4 (Inception B).

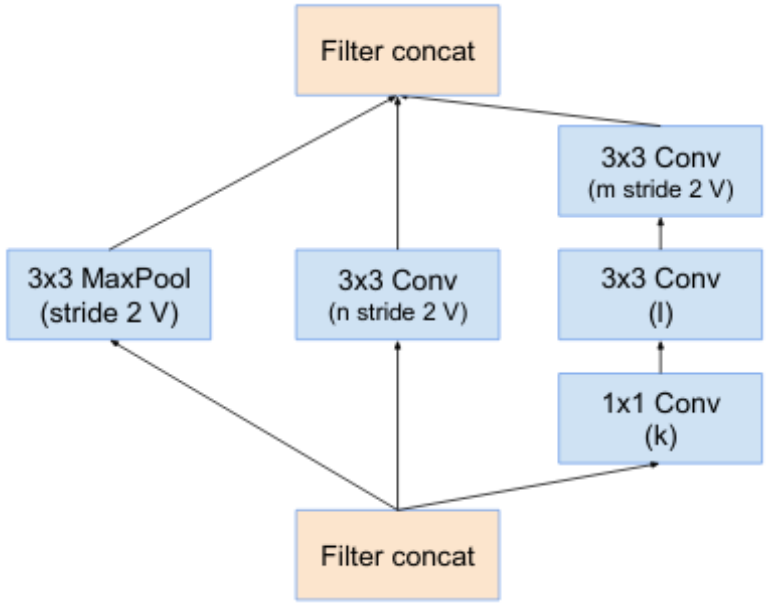
CNNs for image classification

Inception-ResNet (2016)



Szegedy et at. (2017), Fig. 6

8×8 Inception module for Inception v4 (Inception C).

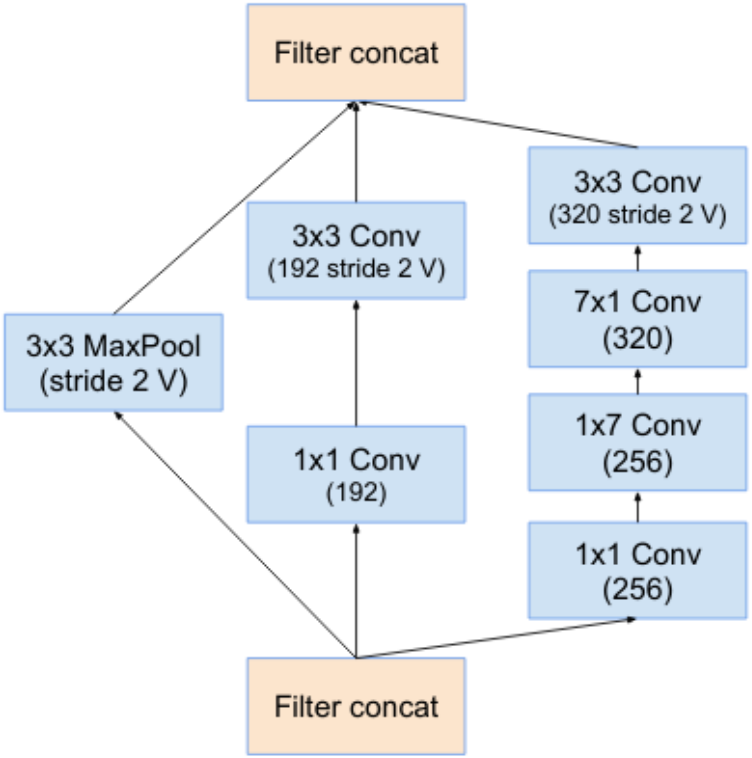


Szegedy et at. (2017), Fig. 7

Module for reduction from 35×35 to 17 × 17 (Reduction-A).

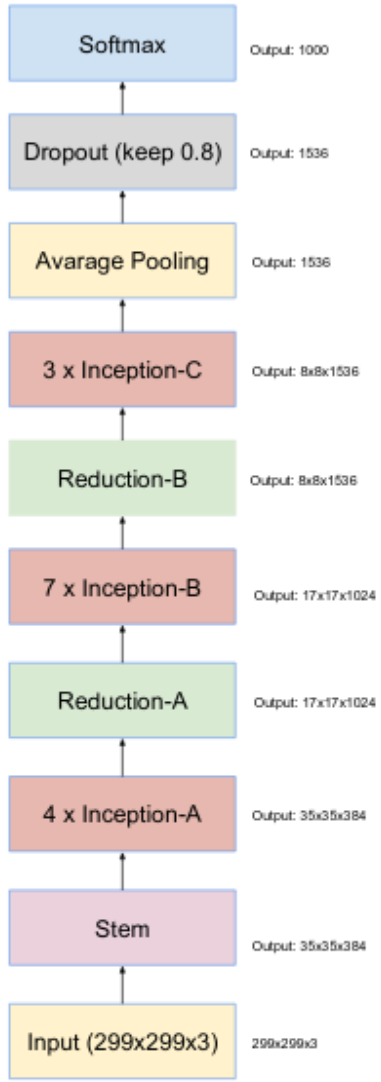
CNNs for image classification

Inception-ResNet (2016)



Szegedy et at. (2017), Fig. 8

Module for reduction from 17×17 to 8×8 (Reduction-B).

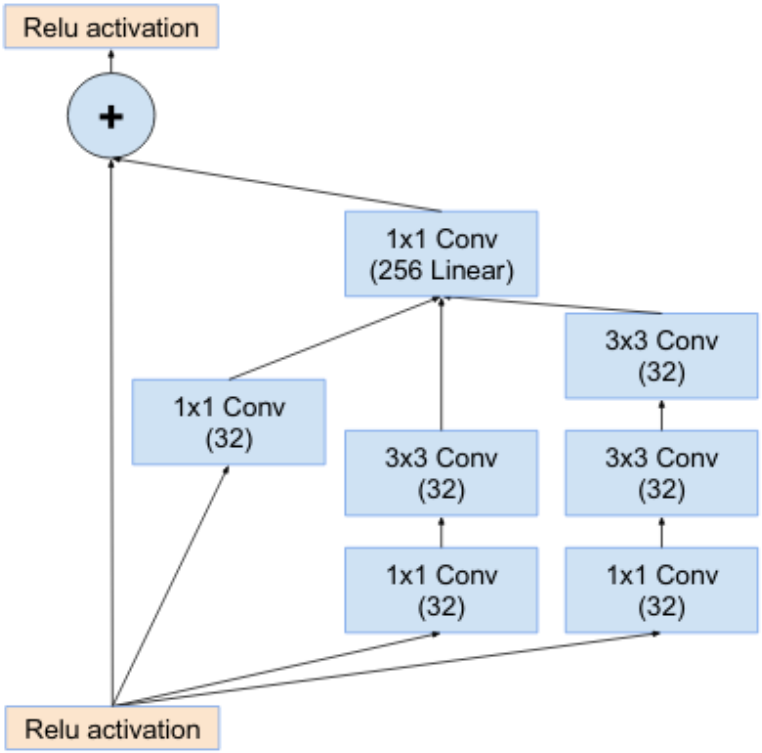


Complete Inception v4 model.

Szegedy et at. (2017), Fig. 9

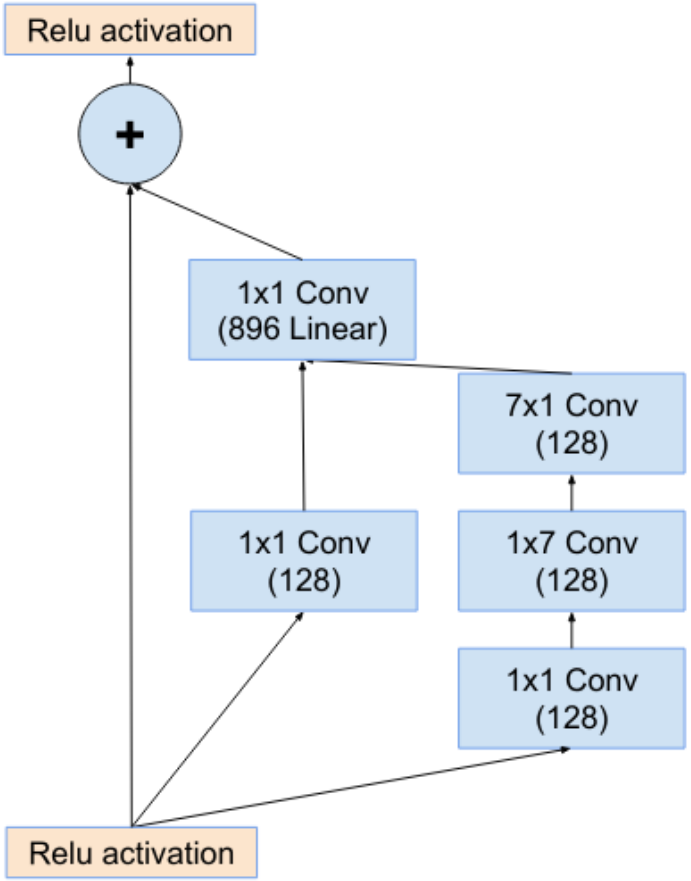
CNNs for image classification

Inception-ResNet (2016)



Szegedy et at. (2017), Fig. 10

35×35 Inception ResNet module (Inception-ResNet-A).

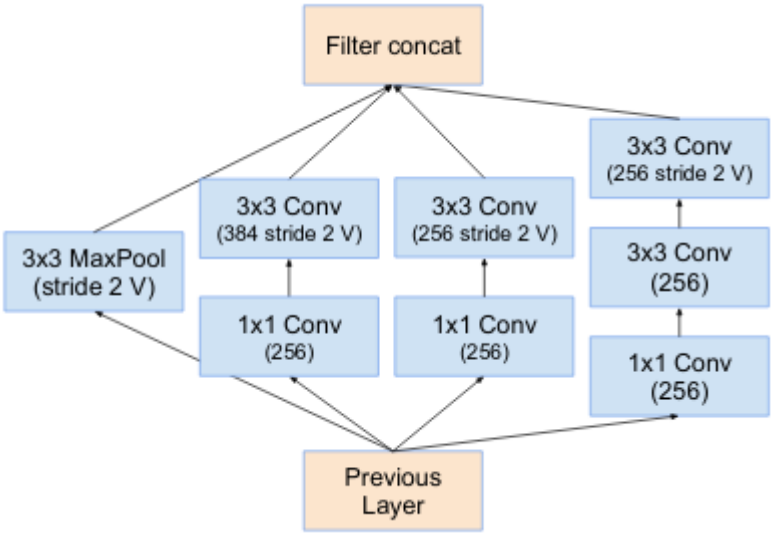


Szegedy et at. (2017), Fig. 11

17×17 Inception ResNet module (Inception-ResNet-B).

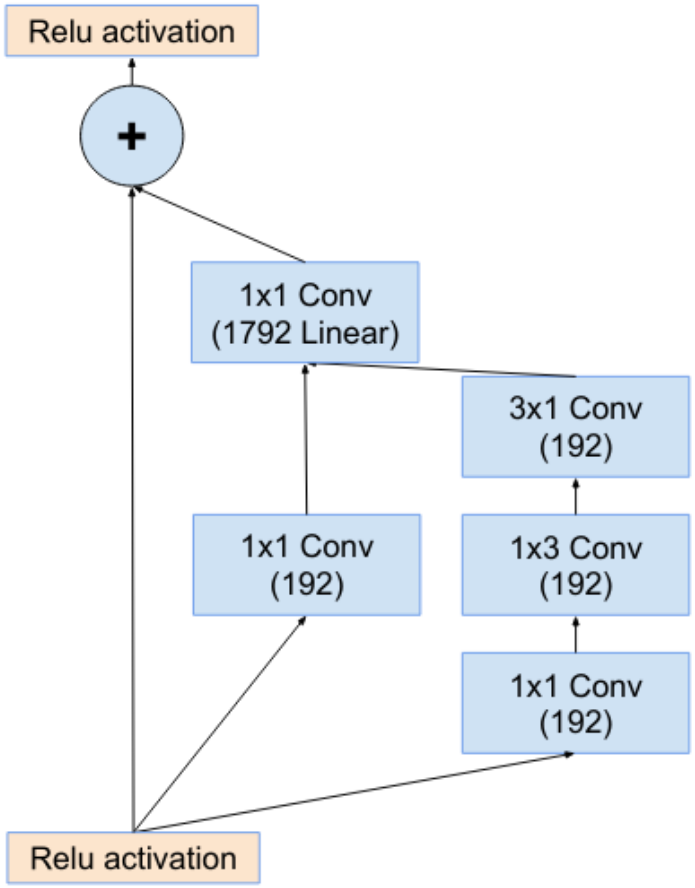
CNNs for image classification

Inception-ResNet (2016)



Szegedy et at. (2017), Fig. 12

17×17 to 8×8 reduction (Reduction-B) for Inception-ResNet-v1.

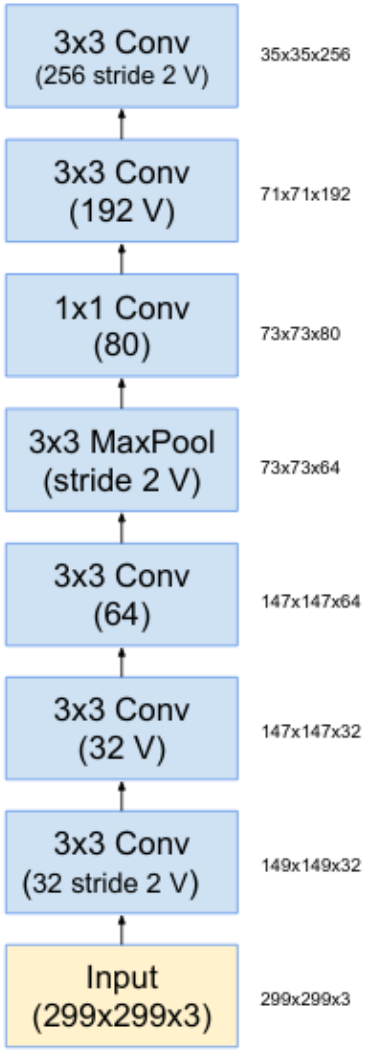


Szegedy et at. (2017), Fig. 13

8×8 Inception-ResNet module (Inception-ResNet-C).

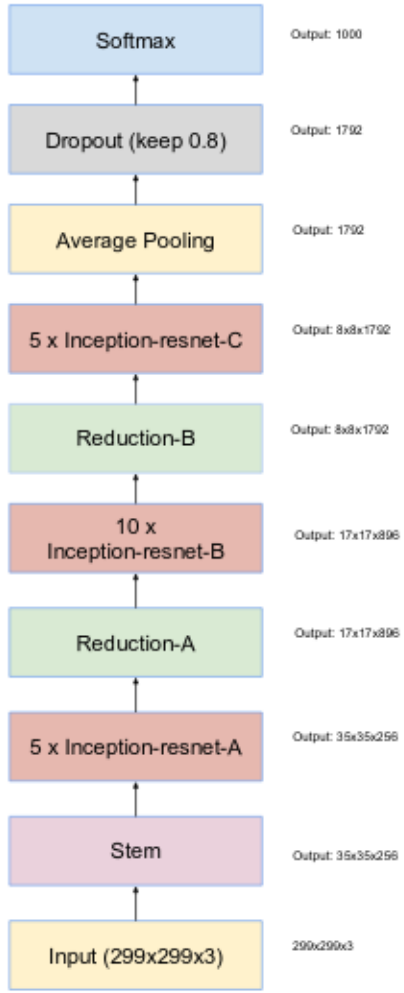
CNNs for image classification

Inception-ResNet (2016)



Stem of Inception-ResNet-v1.

Szegedy et at. (2017), Fig. 14

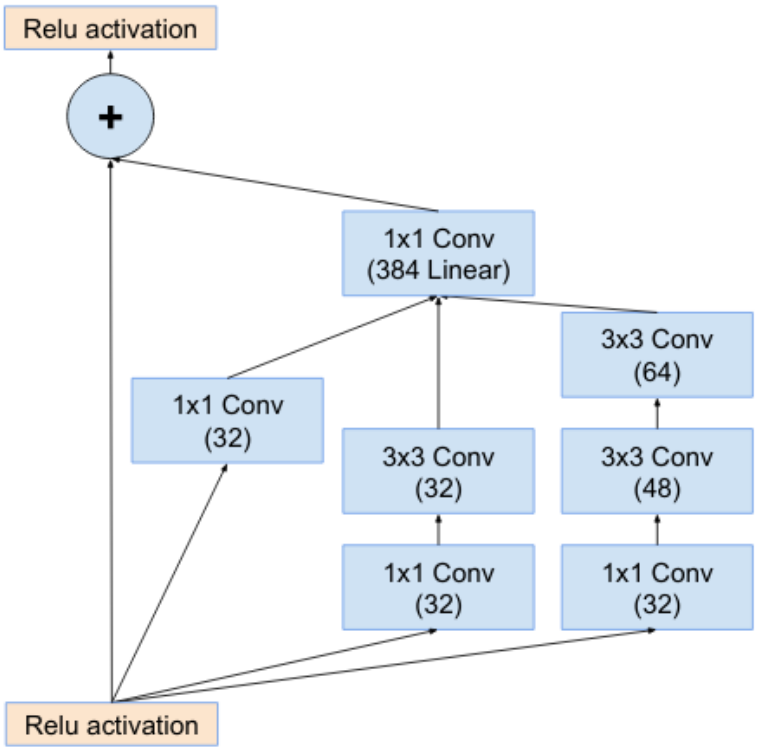


Complete Incpetion-ResNet-v1 and Inception-ResNet-v2 architectures.

Szegedy et at. (2017), Fig. 15

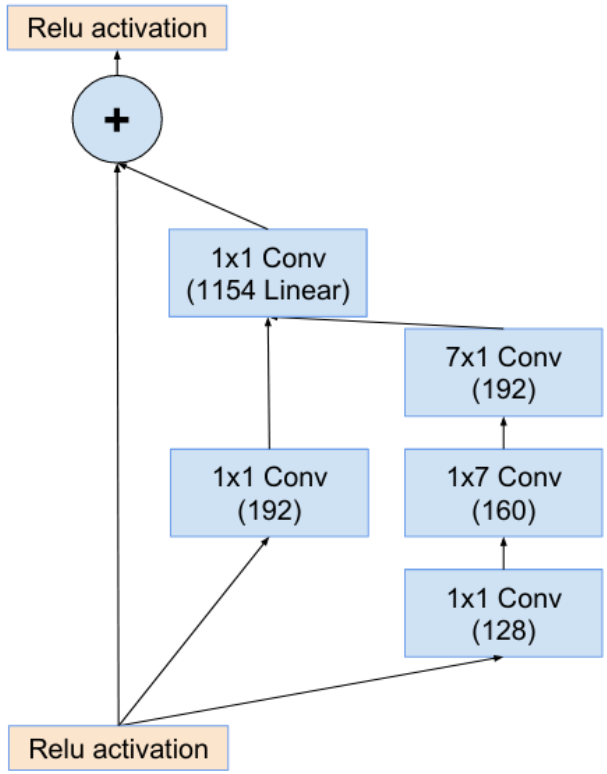
CNNs for image classification

Inception-ResNet (2016)



Szegedy et at. (2017), Fig. 16

35×35 module for Inception-ResNet-v2 (Incpetion-ResNet-A).

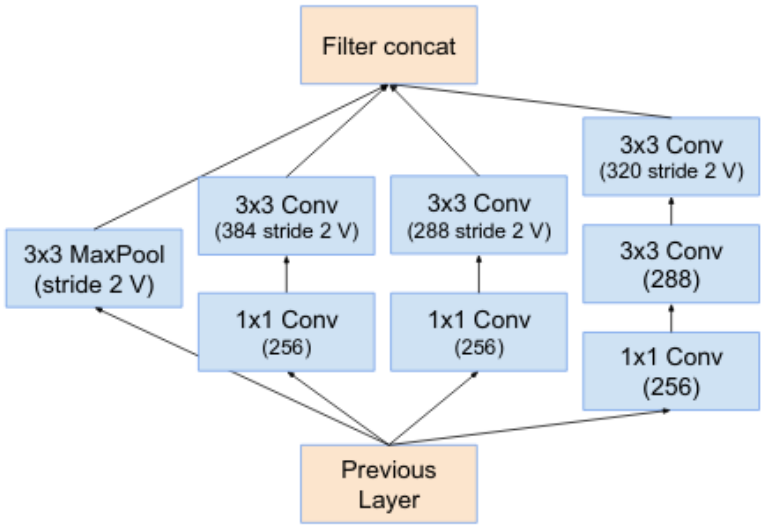


Szegedy et at. (2017), Fig. 17

17×17 module for Inception-ResNet-v2 (Inception-ResNet-B).

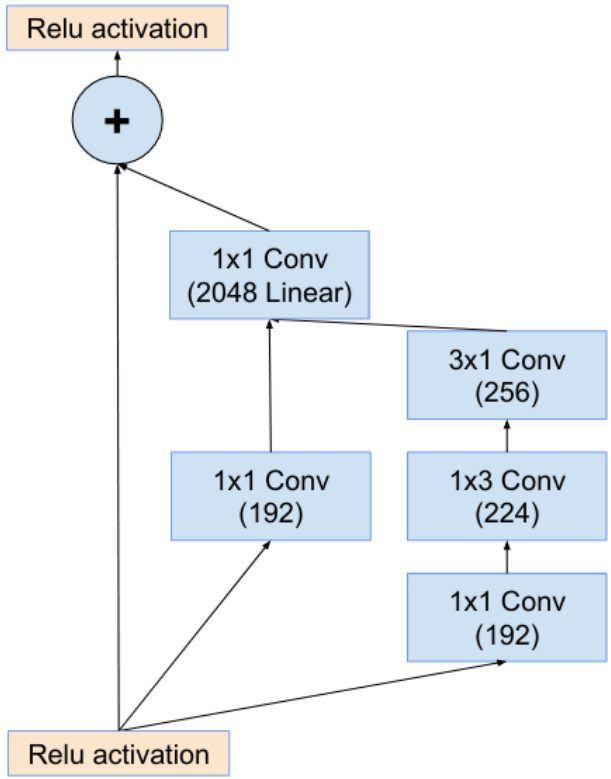
CNNs for image classification

Inception-ResNet (2016)



Szegedy et at. (2017), Fig. 18

17×17 reduction to 8×8 module for Inception-ResNet-v2 (Incpetion-ResNet-A).



Szegedy et at. (2017), Fig. 19

8×8 module for Inception-ResNet-v2 (Inception-ResNet-C).

Conclusion from Inception-ResNet: training of Inception models is faster with residual connections.

Besides techniques for exploiting residual connections in Inception modules, the paper introduces the idea of residual scaling.

Residual scaling was prompted by the finding that with large numbers of filters, regardless of learning rate, later layers started to produce only zeros due to unstable updates.

Weighting the residuals added to a module's output by 0.1–0.3 before adding improved stability of training.

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

Squeeze and excitation (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*.

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

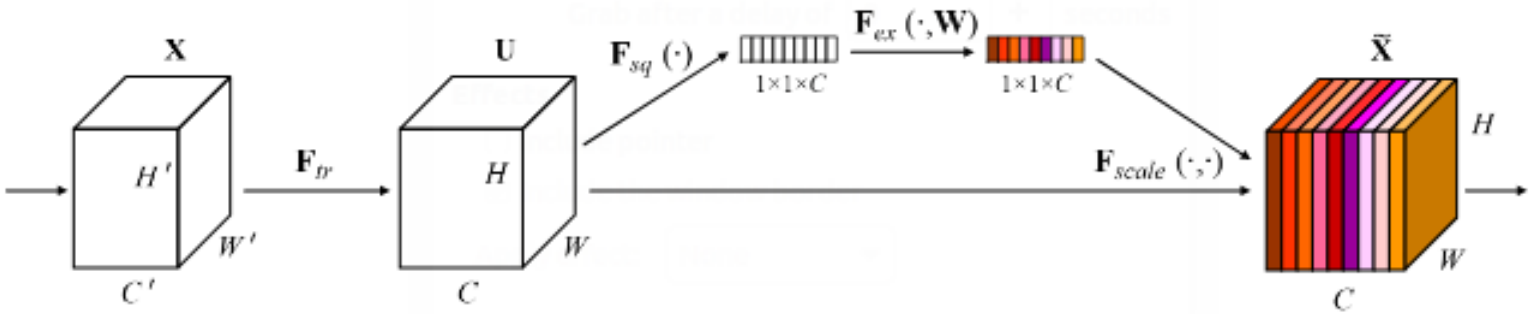
CNNs for image classification

Squeeze and excitation (ILSVRC 2017)

Basic idea: reweight (scale) channels globally according to informativeness. The technique is called feature recalibration.

We have a transformation $\mathtt{F} : \mathcal{X} \mapsto \mathcal{U}$ with $\mathcal{X} = \mathbb{R}^{H' \times W' \times C'}$ and $\mathcal{U} = \mathbb{R}^{H \times W \times C}$.

A squeeze operation aggregates feature map $\mathtt{U} \in \mathcal{U}$ across dimensions $H \times W$ to produce a channel descriptor vector z. It's a simple average of the elements in each channel (no parameters).



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

Excitation is more complicated...

A squeeze is followed by an excitation operation that performs a sample-specific reweighting of the channels of $\mathtt{U}$.

$$\mathsf{s} = \mathsf{F}_{ex}(\mathsf{z}, \mathtt{W}) = \sigma(g(\mathsf{z}, \mathtt{W})) = \sigma(\mathtt{W}_2\delta(\mathtt{W}_1\mathsf{z}))$$

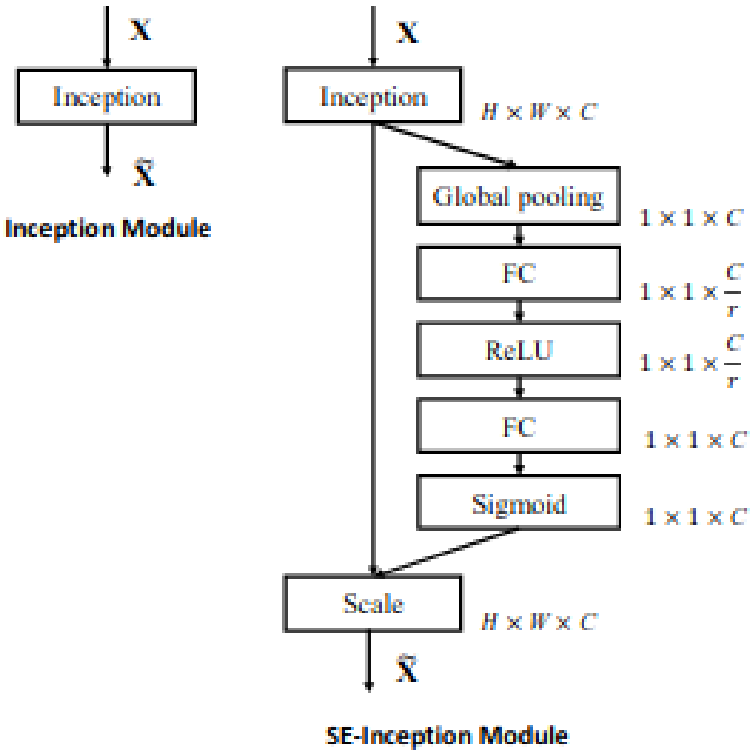
$\delta(\cdot)$ is ReLU, and $\sigma(\cdot)$ is the logistic sigmoid.

$\mathtt{W}_1$ performs dimensionality reduction, and $\mathtt{W}_2$ performs dimensionality expansion.

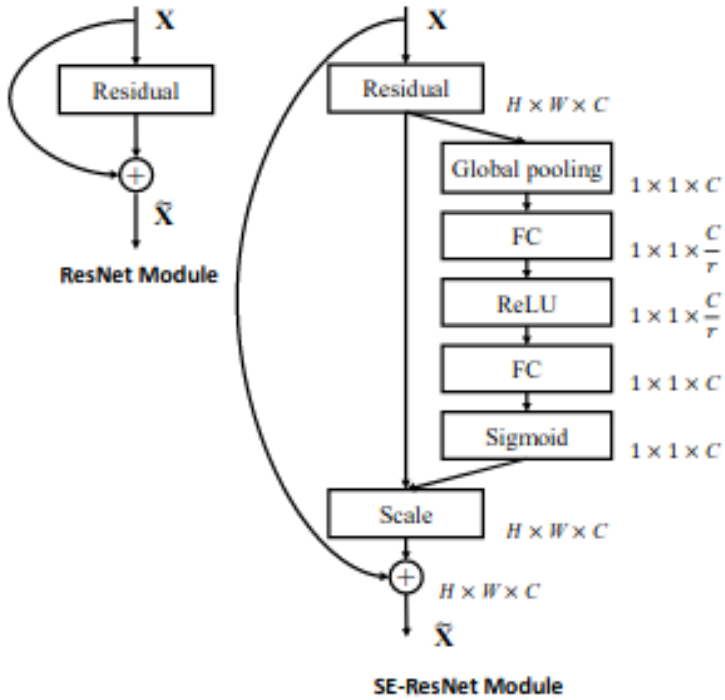
The elements of s are then used to scale $\mathtt{U}$.

For SE to be useful, we can see that $\delta(\mathtt{W}_1\mathsf{z})$ should be a low-dimensional coding of the global activity of each feature map based on which $\mathtt{W}_2$ will decide which feature maps to weight more or less actively.

SE can be applied to simple CNNs like AlexNet easily. The authors also show how to apply SE to Inception modules and ResNet.



Hu, Shen, and Sun (2018), Fig. 2



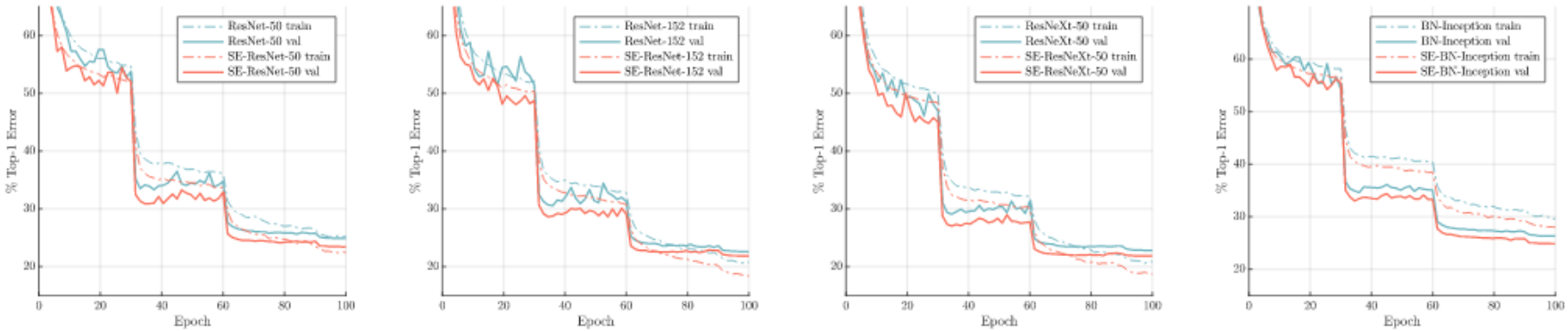
Hu, Shen, and Sun (2018), Fig. 3

The approach improves the single-crop ImageNet performance of several models:

	original		re-implementation			SENet		
	top-1 err.	top-5 err.	top-1 err.	top-5 err.	GFLOPs	top-1 err.	top-5 err.	GFLOPs
ResNet-50 [13]	24.7	7.8	24.80	7.48	3.86	$23.29_{(1.51)}$	$6.62_{(0.86)}$	3.87
ResNet-101 [13]	23.6	7.1	23.17	6.52	7.58	$22.38_{(0.79)}$	$6.07_{(0.45)}$	7.60
ResNet-152 [13]	23.0	6.7	22.42	6.34	11.30	$21.57_{(0.85)}$	$5.73_{(0.61)}$	11.32
ResNeXt-50 [19]	22.2	-	22.11	5.90	4.24	$21.10_{(1.01)}$	$5.49_{(0.41)}$	4.25
ResNeXt-101 [19]	21.2	5.6	21.18	5.57	7.99	$20.70_{(0.48)}$	$5.01_{(0.56)}$	8.00
VGG-16 [11]	-	-	27.02	8.81	15.47	$25.22_{(1.80)}$	$7.70_{(1.11)}$	15.48
BN-Inception [6]	25.2	7.82	25.38	7.89	2.03	$24.23_{(1.15)}$	$7.14_{(0.75)}$	2.04
Inception-ResNet-v2 [21]	$19.9^{\dagger}$	$4.9^{\dagger}$	20.37	5.21	11.75	$19.80_{(0.57)}$	$4.79_{(0.42)}$	11.76

Hu, Shen, and Sun (2018), Table 2

The performance improvement is immediate and consistent:



Hu, Shen, and Sun (2018), Fig. 4

SE also improves the performance of smaller networks aimed at mobile devices:

	original		re-implementation				SENet			
	top-1 err.	top-5 err.	top-1 err.	top-5 err.	MFLOPs	Params	top-1 err.	top-5 err.	MFLOPs	Params
MobileNet [64]	29.4	-	28.4	9.4	569	4.2M	$25.3_{(3.1)}$	$7.7_{(1.7)}$	572	4.7M
ShuffleNet [65]	32.6	-	32.6	12.5	140	1.8M	$31.0_{(1.6)}$	$11.1_{(1.4)}$	142	2.4M

Hu, Shen, and Sun (2018), Table 3

Lessons to be learned from ILSVRC entries:

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

Outline

1. Introduction
2. Neural network intuition
3. Efficient computation
4. Backpropagation
5. Avoiding overfitting
6. CNNs for image classification
7. Conclusion

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.