15-381: Deep Learning

J. Zico Kolter

December 1, 2015

Outline

Some brief history

Machine learning with neural networks

Training neural networks

Advanced models and architectures

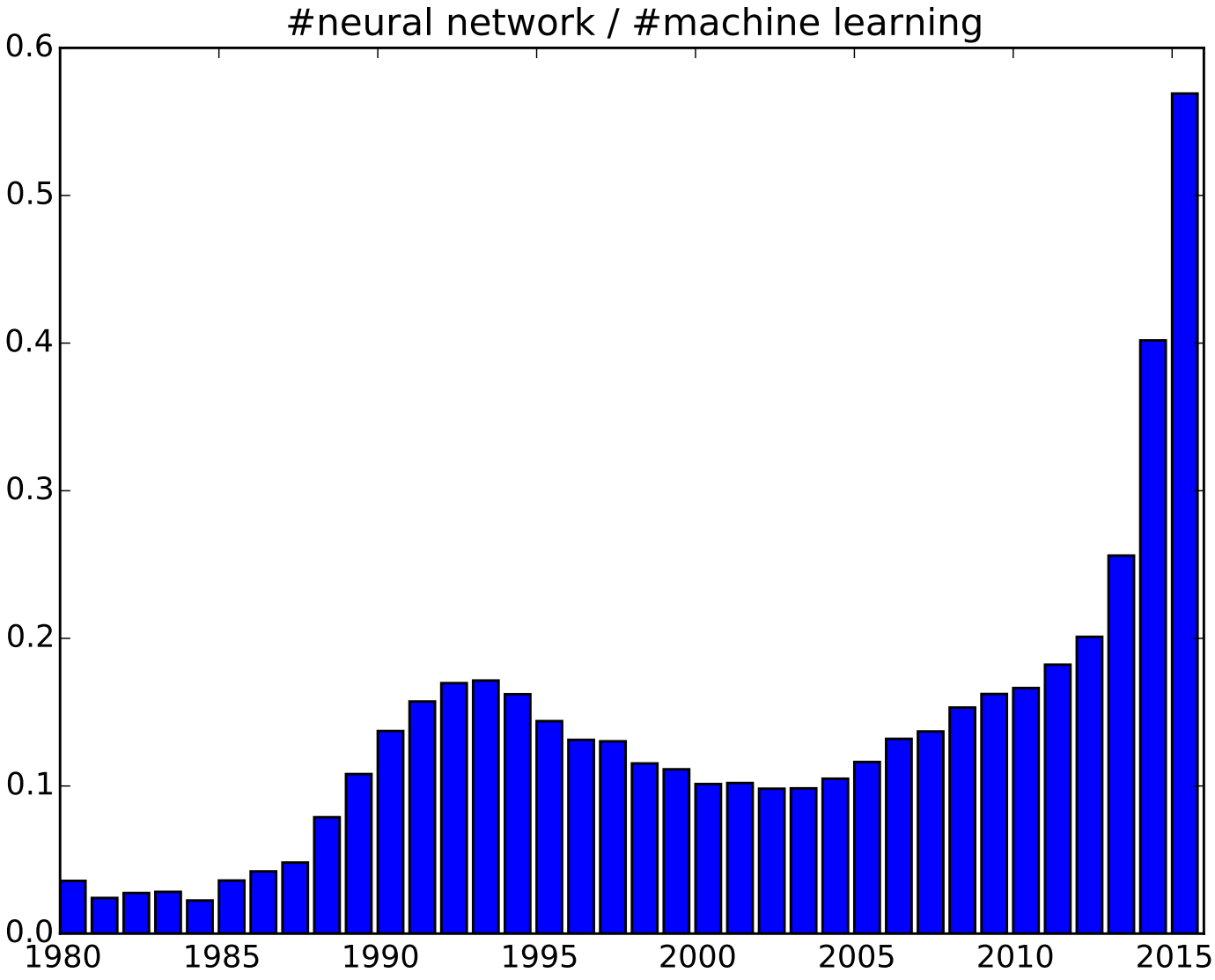
Outline

Some brief history

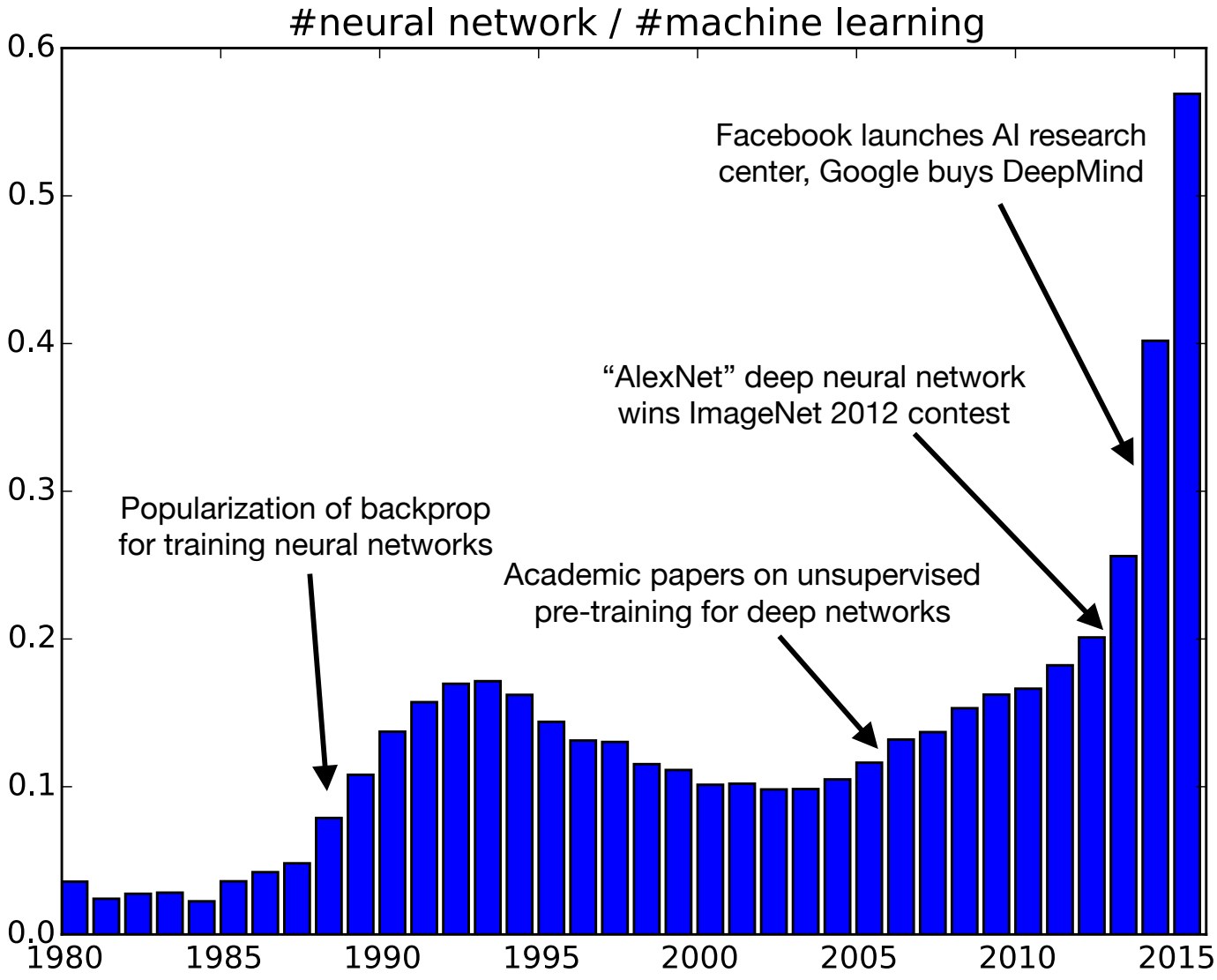
Machine learning with neural networks

Training neural networks

Advanced models and architectures



Google scholar counts of papers containing “neural network” divided by count of papers containing “machine learning”



A highly non-exhaustive list of some of the important events that impacted this trend

Outline

Some brief history

Machine learning with neural networks

Training neural networks

Advanced models and architectures

Recall supervised learning setup

Input features $x^{(i)} \in \mathbb{R}^n$

Outputs $y^{(i)} \in \mathcal{Y}$ ($\mathbb{R}$, $\{-1, +1\}$, $\{1, \ldots, p\}$

Model parameters $\theta \in \mathbb{R}^k$

Hypothesis function $h_\theta : \mathbb{R}^n \to \mathcal{Y}$

Loss function $\ell : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}_+$

Machine learning optimization problem

$$\underset{\theta}{\text{minimize}} \quad \sum_{i=1}^{m} \ell(h_\theta(x^{(i)}), y^{(i)})$$

(possibly plus some additional regularization)

We mainly considered the linear hypothesis class

$$h_\theta(x^{(i)}) = \theta^T \phi(x^{(i)})$$

for some set of non-linear features $\phi : \mathbb{R}^n \to \mathbb{R}^k$

(Note: previously, we just directly included the non-linear features in $x^{(i)}$, but here we separate them for clarity)

Example

$$x^{(i)} = [\text{temperature for day } i]$$

$$\phi(x^{(i)}) = \begin{bmatrix} 1 \\ x^{(i)} \\ {x^{(i)}}^2 \\ \vdots \end{bmatrix}$$

Challenges with linear models

Linear models crucially depend on choosing “good” features

Some “standard” choices: polynomial features, radial basis functions, random features (surprisingly effective)

But, many specialized domains required highly engineered special features

- E.g., computer vision tasks used Haar features, SIFT features, every 10 years or so someone would engineer a new set of features

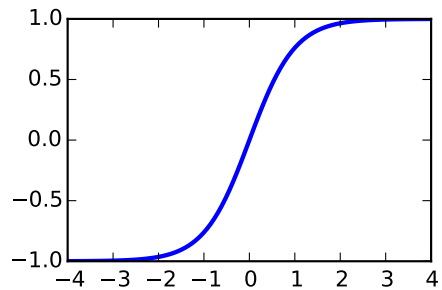
Neural networks

Neural networks are a simply a machine learning algorithm with a more complex hypothesis class, directly incorporating non-linearity (in the parameters)

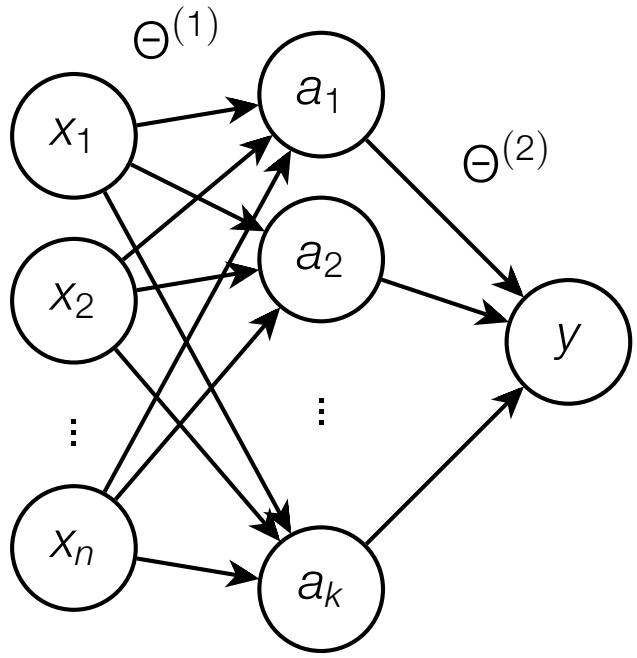
Example: neural network with one hidden layer

$$h_\theta(x) = \Theta^{(2)} f(\Theta^{(1)} x)$$

where $\Theta^{(1)} \in \mathbb{R}^{k \times n}$, $\Theta^{(2)} \in \mathbb{R}^{1 \times k}$ and f is some non-linear function applied elementwise to a vector (common choice is "tanh" function $\tanh(x) = (1 - e^{-2x})/(1 + e^{-2x})$)



Architectures are often shown graphically



Middle layer *a* is referred to as the *hidden layer*, there is nothing in the data that prescribes what values these should take, left up to the algorithm to decide

Viewed another way: neural networks are like linear classifiers where the features themselves are *also* learned

Pros

- No need to manually engineer good features, just let the machine learning algorithm handle this part

- It turns out that a 3-layer network is a *universal function approximator*, any non-linear function can be represented with a 3-layer network with a large enough hidden layer

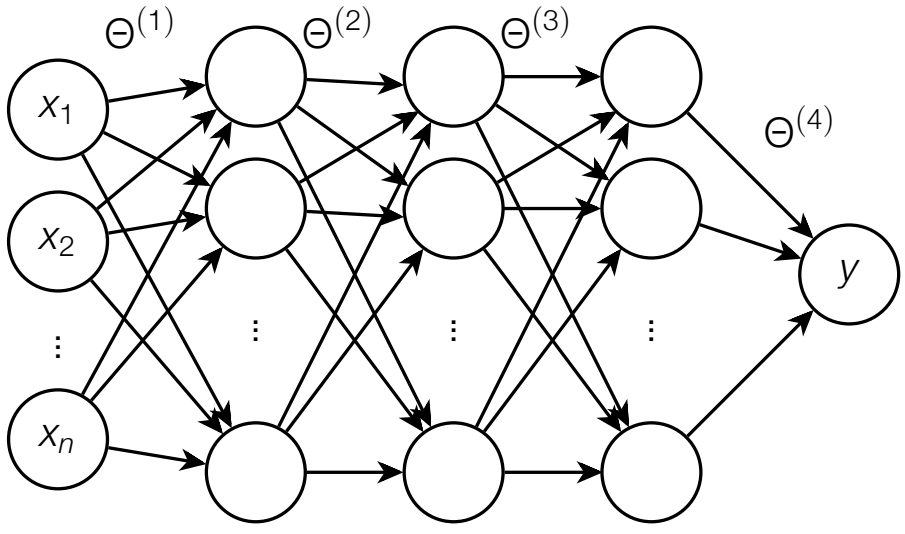
Cons

- Minimizing loss on training data is no longer a convex optimization problem in parameters θ

- Still need to engineer a good architecture (more on this shortly)

Deep learning

"Deep" neural networks typically refer to networks with multiple hidden layers



Note: original term "deep learning" referred to any machine learning architecture with multiple layers, including several probabilistic models, etc, but most work these days focuses on neural networks

Motivation from circuits: many functions can be represented more compactly using deep networks than one-hidden layer networks (e.g. parity function would require (2^n) hidden units in 3-layer network, $O(n)$ units in $O(\log n)$-layer network)

Motivation from neurobiology: brain appears to use multiple levels of interconnected neurons to process information (but careful, neurons in brain are not just non-linear functions)

In practice: works better for many domains

Outline

Some brief history

Machine learning with neural networks

Training neural networks

Advanced models and architectures

Optimizing neural network parameters

How do we optimize the parameters for the machine learning loss minimization problem with a neural network

$$\underset{\theta}{\text{minimize}} \quad \sum_{i=1}^{m} \ell(h_\theta(x^{(i)}), y^{(i)})$$

now that this problem is non-convex?

Just do exactly what we did before: initialize with random weights and run stochastic gradient descent

Now have the possibility of local optima, and function can be harder to optimize, but we won't worry about all that because the resulting models still often perform better than linear models

Stochastic gradient descent, repeat:

- Select some example i

$$\theta := \theta - \alpha \nabla_\theta \ell \left(h_\theta(x^{(i)}), y^{(i)} \right), \quad i = 1, \ldots, m$$

So how do we compute the gradient with respect to all the parameters in a neural network (i.e., all weights $\Theta^{(1)}$, $\Theta^{(2)}$, . . .)?

Backpropagation

Backpropagation is a method for computing all the necessary gradients using one “forward pass” (just computing all the values at layers), and one “backward pass” (computing gradients backwards in the network)

The equations sometimes look complex, but it’s just an application of the chain rule of calculus

First, some notation that will make things a bit easier:

- Activations: $a^{(i)}$ values at hidden layer i (with convention that $a^{(1)} = x$)

- Linear activations: $z^{(i)} = \Theta^{(i)} a^{(i)}$, so $a^{(i+1)} = f(z^{(i)})$

Let's treat everything as scalars for now, and consider a network with two hidden layers:

$$\begin{aligned}
\frac{\partial}{\partial \Theta^{(1)}} \ell(h_\theta(x), y) &= \frac{\partial}{\partial \Theta^{(1)}} \ell(\Theta^{(3)} f(\Theta^{(2)} f(\Theta^{(1)} x)), y) \\
&= \ell'(z^{(3)}, y) \Theta^{(3)} \frac{\partial}{\partial \Theta^{(1)}} f(\Theta^{(2)} f(\Theta^{(1)} x)) \\
&= \ell'(z^{(3)}, y) \Theta^{(3)} f'(z^{(2)}) \Theta^{(2)} \frac{\partial}{\partial \Theta^{(1)}} f(\Theta^{(1)} x) \\
&= \ell'(z^{(3)}, y) \Theta^{(3)} f'(z^{(2)}) \Theta^{(2)} f'(z^{(1)}) a^{(1)}
\end{aligned}$$

By the same procedure

$$\frac{\partial}{\partial \Theta^{(2)}} \ell(h_\theta(x), y) = \ell'(z^{(3)}, y) \Theta^{(3)} f'(z^{(2)}) a^{(2)}$$

If you want to compute gradients with respect to all the parameters $\Theta^{(1)}, \ldots, \Theta^{(L)}$, we can "reuse" parts of this computation

Let

$$g^{(L)} = \ell'(z^{(L)})\Theta^{(L)}$$
$$g^{(i)} = g^{(i+1)}f'(z^{(i)})\Theta^{(i)}$$

then

$$\frac{\partial}{\partial \Theta^{(i)}}\ell(h_\theta(x), y) = g^{(i+1)}f'(z^{(i)})a^{(i)}$$

It takes just slightly more advanced calculus, but it turns out the general matrix/vector case is *exactly* the same, just being careful with the ordering/size of matrix multiplication

The full backpropagation algorithm:

$$
\begin{aligned}
g^{(L)} &= \Theta^{(L)^T} \ell'(z^{(L)}) \\
g^{(i)} &= \Theta^{(i)^T} (g^{(i+1)} \cdot f'(z^{(i)})) \\
\nabla_{\Theta^{(i)}} \ell(h_\theta(x), y) &= (g^{(i+1)} \cdot f'(z^{(i)})) a^{(i)^T}
\end{aligned}
$$

where $\cdot$ denotes elementwise multiplication

You'll often see this written out element by element (e.g., in R+N Ch 20), and slight differences depending on whether you also apply f to the last layer, but it's the same algorithm

Gradients can get somewhat tedious to derive by hand, especially for the more complex models that follow

Fortunately, a lot of this work has already been done for you

Tools like Theano (`http://deeplearning.net/software/theano/`), Torch (`http://torch.ch/`) all let you specify the network structure and then automatically compute all gradients (and use GPUs to do so)

Autograd package for Python (`https://github.com/HIPS/autograd`) lets you compute the derivative of (almost) any arbitrary function using numpy operations using automatic backprop

What's changed since the 80s?

All these algorithms (and most of the extensions in later slides), were developed in the 80s or 90s

So why are these just becoming more popular in the last few years?

- More data
- Faster computers
- (Some) better optimization techniques

Unsupervised pre-training (Hinton et al., 2006): "Pre-train" the newtork have the hidden layers recreate their input, one layer at a time, in an unsupervised fashion

- This paper was partly responsible for re-igniting the interest in deep neural networks, but the general feeling now is that it doesn't help much

Dropout (Hinton et al., 2012): During training and computation of gradients, randomly set about half the hidden units to zero (a different randomly selected set for each stochastic gradient step)

- Acts like regularization, prevents the parameters for overfitting to particular examples

Different non-linear functions (Nair and Hinton, 2010): Use non-linearity $f(x) = \max\{0, x\}$ instead of $f(x) = \tanh(x)$

Outline

Some brief history

Machine learning with neural networks

Training neural networks

Advanced models and architectures

Convolutional neural networks

One of the biggest successes for neural networks has come in computer vision, using convolutional neural networks

In traditional neural networks, images are treated as unstructured vectors $x \in \mathbb{R}^{W \cdot H}$, and we learn arbitrary transformations i.e., $f(\Theta x)$

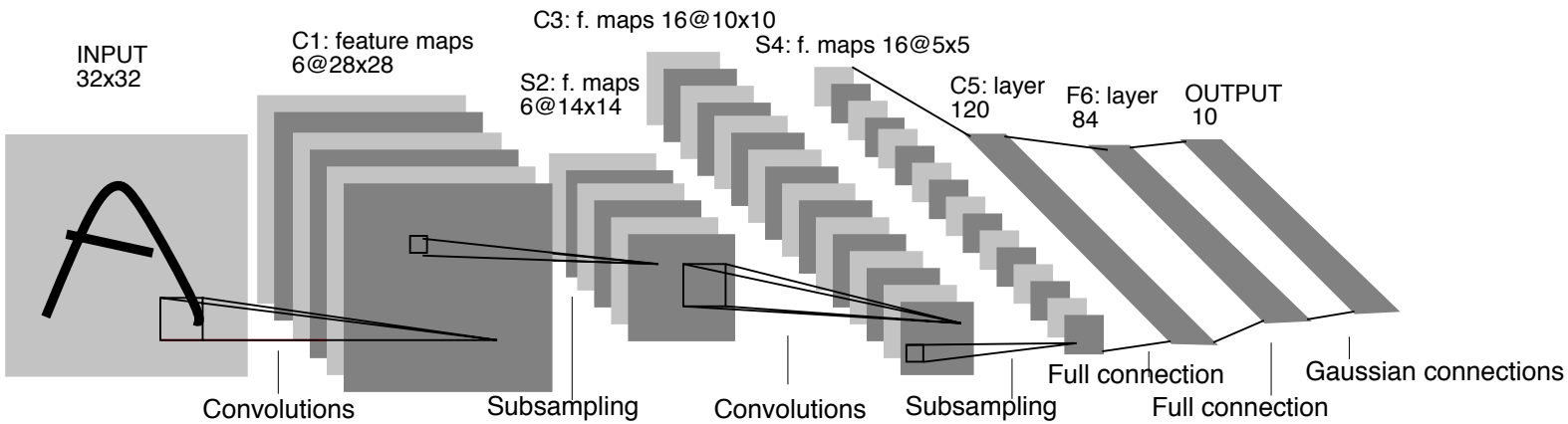
But this doesn't seem like a good model, slightly shifting/scaling image winds up with a very different input vector, so we need to learn all these invariances in our parameters

Basic idea of convolutional neural networks: parameters are elements of a (set of) convolutional filters applied to the image

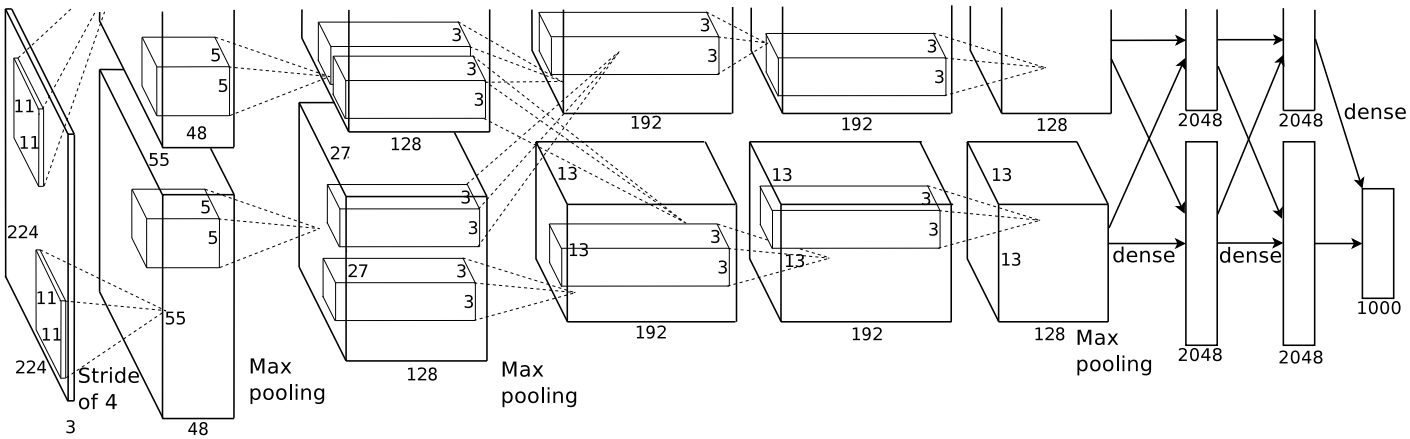
$a^{(i+1)} = f(a^{(i)} * \Theta^{(i)})$

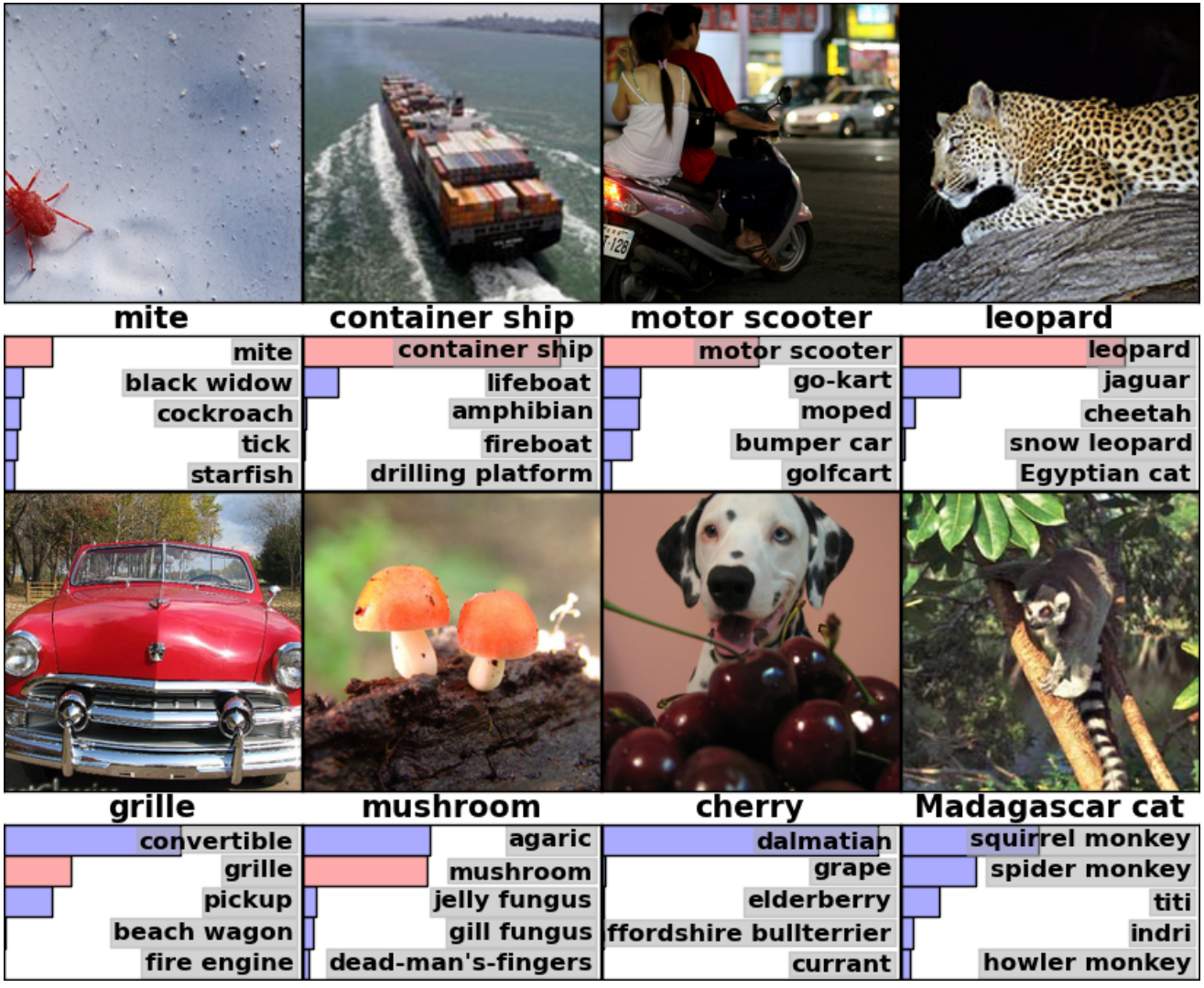
The function *f* also does downsampling "max-pooling" to produce lower dimensional images at later layers

Lenet-5 (LeCun et al., 1998) architecture, 1% error on MNIST classification (compare to 10% for linear classifier)



“AlexNet” (Krizhevsky et al., 2012), work ImageNet 2012 competition with a Top-5 error rate of 15.3% (next best system with highly engineered features based upon SIFT got 26.1% error)





Some classification results from AlexNet

Google Deep Dream software: adjust input images (by gradient descent) to strengthen the activations that are present in an image

Recurrent neural networks

Framework for dealing with sequence data: hidden units for elements in sequence are fed into hidden units for subsequent elements

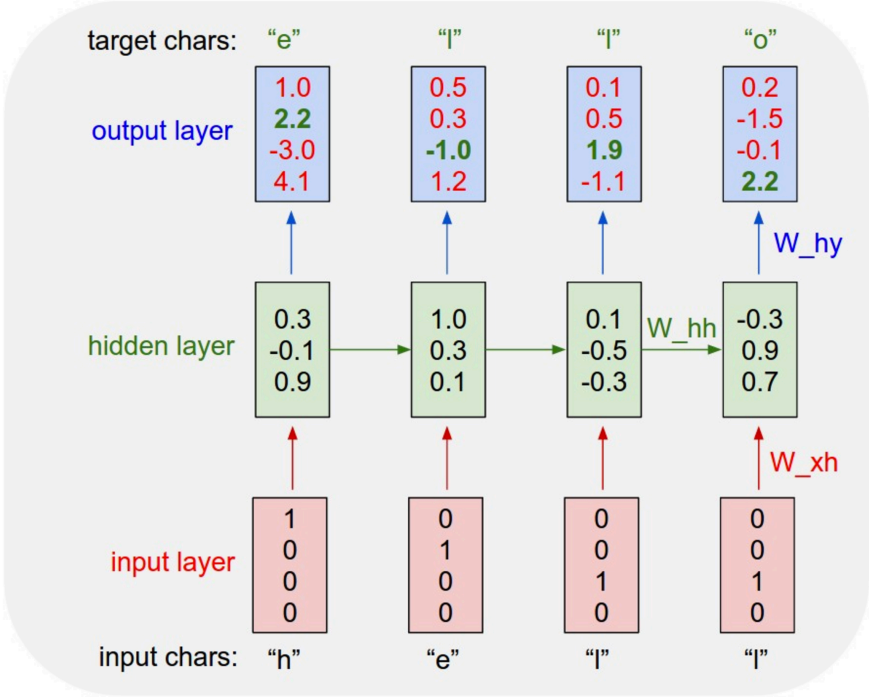


Figure and next examples from Karpathy, 2015,
http://karpathy.github.io/2015/05/21/rnn-effectiveness/.

Recurrent neural network trained character-by-character level on the linux source code

Some random source code sampled from the resulting model

```
/*
 * Increment the size file of the new incorrect UI_FILTER group information
 * of the size generatively.
 */
static int indicate_policy(void)
{
  int error;
  if (fd == MARN_EPT) {
    /*
     * The kernel blank will coeld it to userspace.
     */
    if (ss->segment < mem_total)
      unblock_graph_and_set_blocked();
    else
      ret = 1;
    goto bail;
  }
  segaddr = in_SB(in.addr);
  selector = seg / 16;
  setup_works = true;
  ...
```

Deep reinforcement learning

Instead of maintaining a separate Q-value for each state/action pair, we can use a deep neural network (or any other class of functions) to represent the Q function

Q-Learning update rule becomes

$$\theta := \theta - \alpha(R + \gamma \max_{a'} Q(s', a'; \theta) - Q(s, a; \theta))\nabla_\theta Q(s, a; \theta) \quad (1)$$

where θ are the parameters (i.e. network weights) that specify our representation of the Q function

DeepMind paper shows deep learning to play Atari video games (Mihn et al., 2013)