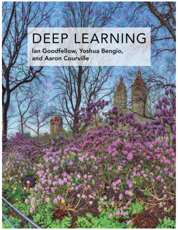
Neural networks and deep learning

Deep Learning by Goodfellow et al. defines 'deep learning' as algorithms enabling "the computer to learn complicated concepts by building them out of simpler ones".



This is implemented using *neural networks* that consist of hierarchically organized simple processing units, *neurons*.

This field had a series of huge successes starting in ∼2012: classifying images, generating images, playing Go, writing text, folding proteins, etc. Known together as the 'deep learning revolution'.

This lecture is about *feed-forward* neural networks for image classification.

We have already spent one entire lecture talking about a neural network classifier!

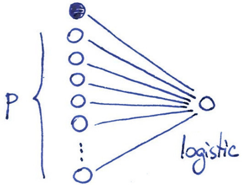
Logistic regression revisited

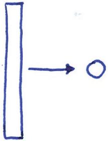
The loss function of logistic regression:

$$\mathcal{L} = -\sum_i \Big[y_i \log h(\mathbf{x}_i) + (1 - y_i) \log \big(1 - h(\mathbf{x}_i)\big) \Big]$$

where

$$h(\mathbf{x}) = \frac{1}{1 + e^{-\beta^\top \mathbf{x}}} = \frac{1}{1 + e^{-\mathbf{W}_1 \mathbf{x}}}.$$

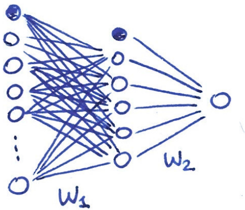


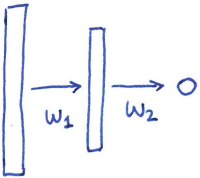


Here coefficients are called *weights*.

A hidden layer

Logistic regression is a linear network that has an *input layer* and an *output layer*. We now add a *hidden layer*:





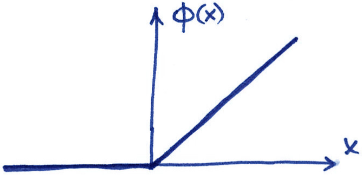
$$\mathcal{L} = -\sum_i \left[y_i \log h(\mathbf{x}_i) + (1 - y_i) \log \left(1 - h(\mathbf{x}_i)\right) \right]$$

Linear: $h(\mathbf{x}) = \dfrac{1}{1 + e^{-\mathbf{W}_2 \mathbf{W}_1 \mathbf{x}}}$ Nonlinear: $\dfrac{1}{1 + e^{-\mathbf{W}_2 \phi(\mathbf{W}_1 \mathbf{x})}}$

Activation function

We want to use some nonlinear *activation function* ϕ that is easy to work with. The most common choice:

$$\phi(x) = \max(0, x).$$



Such neurons are called *rectified linear units (ReLU)*.

Universal approximation theorems

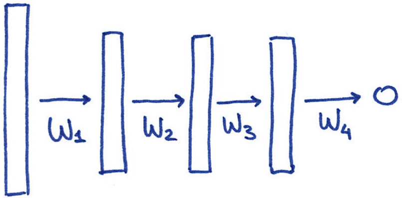
Any continuous function f can be arbitrarily well approximated by a neural network with one hidden layer (for any given non-polynomial activation function ϕ, such as e.g. logistic or rectifier):

$$f(\mathbf{x}) \approx \mathbf{W}_2 \phi(\mathbf{W}_1 \mathbf{x})$$
$$f \approx \mathbf{W}_2 \circ \phi \circ \mathbf{W}_1$$

However, this has little practical relevance because the hidden layer may need to be prohibitively large and/or the training may be prohibitively difficult (note that the loss function is not convex!).

Going deeper

What we had above was a *shallow* network. Here is a deeper one:



$$\mathcal{L} = -\sum_i \Big[y_i \log h(\mathbf{x}_i) + (1 - y_i) \log \left(1 - h(\mathbf{x}_i)\right) \Big]$$

$$h(\mathbf{x}) = \frac{1}{1 + e^{-\mathbf{W}_4 \phi\left(\mathbf{W}_3 \phi\left(\mathbf{W}_2 \phi(\mathbf{W}_1 \mathbf{x})\right)\right)}}$$

Gradient

Logistic regression gradient (see Lecture 5):

$$\nabla \mathcal{L} = -\sum (y_i - h(\mathbf{x}_i)) \nabla (\boldsymbol{\beta}^\top \mathbf{x}_i) = -\sum (y_i - h(\mathbf{x}_i)) \mathbf{x}_i.$$

The gradient for the deep network:

$$\nabla \mathcal{L} = -\sum (y_i - h(\mathbf{x}_i)) \nabla \Big[\mathbf{W}_4 \phi \Big(\mathbf{W}_3 \phi (\mathbf{W}_2 \phi (\mathbf{W}_1 \mathbf{x}_i)) \Big) \Big].$$

Let us write this term down with indices:

$$z = \sum_a W_{4a} \phi \Big[\sum_b W_{3ab} \phi \Big(\sum_c W_{2bc} \phi \big(\sum_d W_{1cd} x_{id} \big) \Big) \Big].$$

Now we need to use the chain rule: if $h(x) = f(g(x))$, then $h'(x) = f'(g(x)) g'(x)$.

Chain rule and backpropagation

$$z = \sum_a W_{4a}\phi\Big[\sum_b W_{3ab}\phi\Big(\sum_c W_{2bc}\phi\big(\sum_d W_{1cd}x_{id}\big)\Big)\Big]$$

$$\frac{\partial z}{\partial W_{4a}} = \phi\Big[\sum_b W_{3ab}\phi\Big(\sum_c W_{2bc}\phi\big(\sum_d W_{1cd}x_{id}\big)\Big)\Big]$$

$$\frac{\partial z}{\partial W_{3ab}} = W_{4a}\frac{\partial\phi(\dots)}{\partial(\dots)}\phi\Big(\sum_c W_{2bc}\phi\big(\sum_d W_{1cd}x_{id}\big)\Big)$$

$$\frac{\partial z}{\partial W_{2bc}} = \sum_a W_{4a}\frac{\partial\phi(\dots)}{\partial(\dots)}W_{3ab}\frac{\partial\phi(\dots)}{\partial(\dots)}\phi\big(\sum_d W_{1cd}x_{id}\big)$$

$$\frac{\partial z}{\partial W_{1cd}} = \sum_a W_{4a}\frac{\partial\phi(\dots)}{\partial(\dots)}\sum_b W_{3ab}\frac{\partial\phi(\dots)}{\partial(\dots)}W_{2bc}\frac{\partial\phi(\dots)}{\partial(\dots)}x_{id}$$

Backpropagation allows efficient computation of all these derivatives using a *backward pass* through the network.

Stochastic gradient descent

Using backpropagation, we can compute the gradient (partial derivatives with respect to each weight) and use gradient descent.

Two notes:

1. In practice, gradient descent algorithm is often used with some modifications: *momentum*, adaptive learning rates (e.g. Adam), etc.
2. Gradient descent requires summation over all training samples at each step. In practice, training data are split into *batches* and are processed one ny one: *stochastic gradient descent (SGD)*. One sweep through the entire training dataset is called an *epoch*.

Multiclass classification

If there are K classes, then the last *softmax* layer has K output neurons:

$$P(y = k) = \frac{e^{z_k}}{\sum_i e^{z_i}},$$

where z_i are pre-nonlinearity activations: $\mathbf{z} = \mathbf{W}_L \phi(\mathbf{W}_{L-1}\phi(\ldots))$.

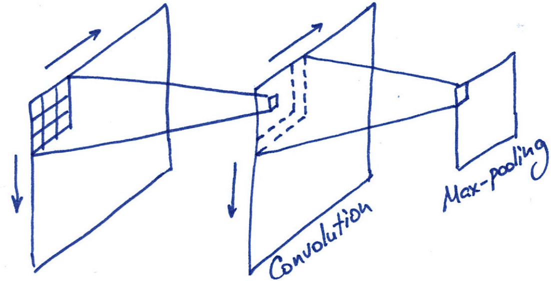
The *cross-entropy* loss function can be written as

$$\mathcal{L} = -\sum_{i=1}^{n}\sum_{k=1}^{K} Y_{ik} \log P(y_i = k),$$

where $Y_{ik} = 1$ if $y_i = k$ and 0 otherwise.

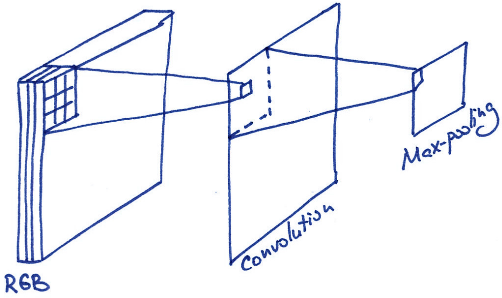
Convolutional neural networks

Convolutional neural networks (CNN) use *weight sharing* to build translation invariance into the model.



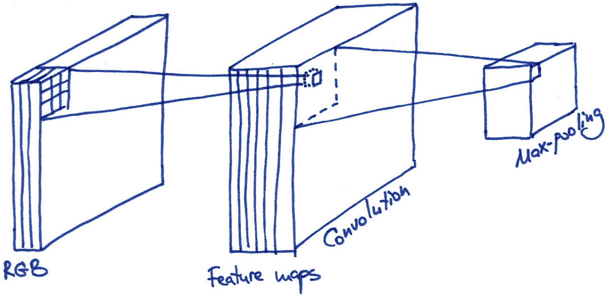
Convolutional neural networks:

The input image has three input channels:



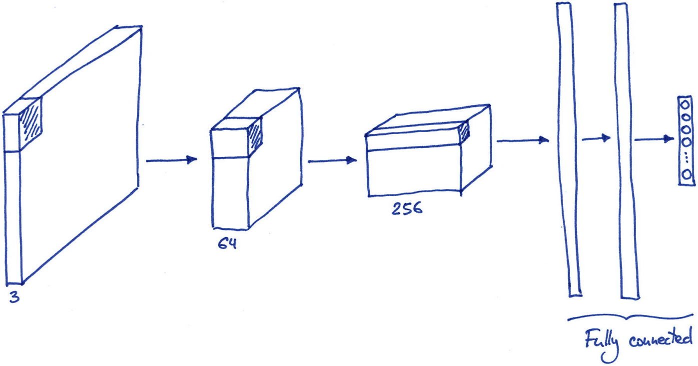
Convolutional neural networks

Several *feature maps*:



Convolutional neural networks

Standard CNN architecture:



What do CNNs learn?

First layer:

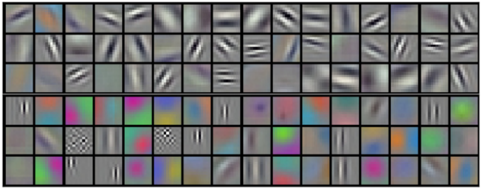


Figure 3: 96 convolutional kernels of size $11 \times 11 \times 3$ learned by the first convolutional layer on the $224 \times 224 \times 3$ input images. The

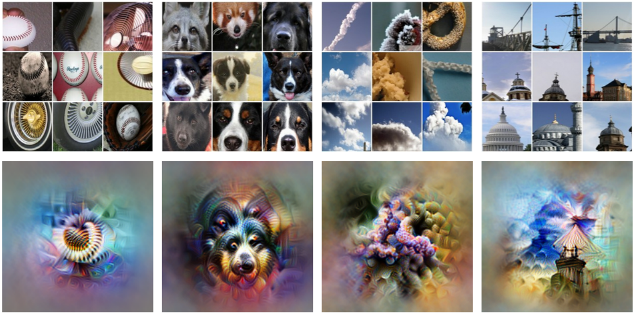
Krizhevsky et al. 2012

What do CNNs learn?

Hidden layer:

Dataset Examples show us what neurons respond to in practice

Optimization isolates the causes of behavior from mere correlations. A neuron may not be detecting what you initially thought.



Baseball—or stripes? *mixed4a, Unit 6*

Animal faces—or snouts? *mixed4a, Unit 240*

Clouds—or fluffiness? *mixed4a, Unit 453*

Buildings—or sky? *mixed4a, Unit 492*

Olah et al. 2017

Historical remarks

Deep neural networks, CNNs, and backpropagation were invented in the 1960/1970s. Why did it take until 2010s for them to become popular?

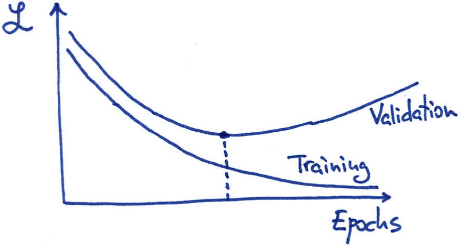
- Computing power (GPUs);
- Large labeled datasets (such as ImageNet);
- Optimization/initialization/normalization/regularization tricks.

It is obvious that one can take a network and run gradient descent. What is not obvious, is (a) whether it will avoid getting stuck in a useless local minimum, and (b) whether it will not hopelessly overfit.

Overfitting and regularization

Ridge (L_2) regularization: $\lambda||\mathbf{W}_l||^2$ on each layer l. This is also called *weight decay* (see Lecture 4).

Another method is called *early stopping*:



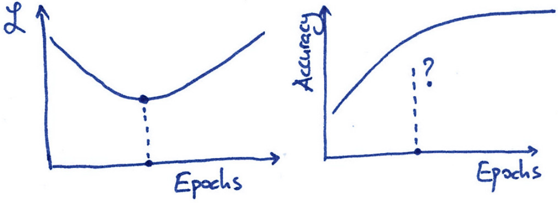
Remark: for linear regression one can show that early stopping penalizes smaller singular values stronger, as does the ridge penalty.

Overparametrization

Modern neural networks are typically used in the overparametrized regime, i.e. they can perfectly or near-perfectly overfit training data. This can be shown by training them using randomly shuffled labels: they can still achieve zero training loss.

At the same time, generalization performance can be high: ‘benign’ overfitting due to *implicit* regularization.

Sometimes one sees overfitting in the test loss but not in the test accuracy:



Overparametrization

When model complexity is gradually increased, the optimal performance is often achieved far beyond the interpolation threshold: