Going Deep: Neural Networks, Part III
Statistics 650/750
Week 15 Tuesday

Christopher Genovese and Alex Reinhart

07 Dec 2017

Announcements

- As announced in emails, homework must be submitted by 5 Dec (today) to guarantee grading by 10 Dec, giving you time to revise. Homework submitted after 5 Dec cannot be revised.
- Final homework submissions 12 Dec, final Challenge 2 submissions 14 Dec. No revisions after these dates.
- Notes on line in `documents` repository `weekFR.*`.
- It's been a pleasure having you all in this class.

Activity Redux

There appeared to be some difficulty last time with the activity, so we will spend a little time on it in a different way.

Reminder:

1. In layer ℓ for $\ell = 1, \ldots, L$, we have n_ℓ nodes.
2. W_ℓ is the $n_{\ell-1} \times n_\ell$ matrix with $W_{\ell,jk}$ being the weight of node j in layer $\ell - 1$ to node k in layer ℓ.
3. b_ℓ is the vector of biases for layer ℓ.
4. The *weighted input* vector is $z_\ell = W_\ell^T a_{\ell-1} + b_\ell$ with *activation* $a_\ell = \phi(z_\ell)$.

A look back at forward

For layers $\ell = 1, \ldots, L$, we have:

$$\begin{aligned} a_0 &= x \\ a_\ell &= \phi(W_\ell^T a_{\ell-1} + b_\ell) \\ &= \phi(z_\ell). \end{aligned}$$

In code, this recursion looks like:

```r
forward <- function(network, input) {
    L <- network$L
    z <- vector("list", L)
    a <- vector("list", L)

    activations <- input
    for ( ell in 1:L ) {
        z <- network$W[ell,,] %*% activations + network$b[ell,]
        activations <- network$phi[ell](z)

        z[[ell]] <- z
        a[[ell]] <- activations
    }
    return( list(output=a[[L]], a=a, z=z, input=input, L=L) )
}
```

Here, we used a simple method to access the network's attributes, but other (possibly better) ways are possible.

Back Propagation

The four main backpropagation equations are:

$$\begin{aligned}
\delta_L &= \frac{\partial C}{\partial a_L} \star \phi'(z_L) \qquad (1)\\
&= (y - a_L(x)) \star \phi'(z_L)\\
\delta_{\ell-1} &= (W_\ell \delta_\ell) \star \phi'(z_{\ell-1}) \qquad (2)\\
\frac{\partial C}{\partial b_\ell} &= \delta_\ell \qquad (3)\\
\frac{\partial C}{\partial W_\ell} &= a_{\ell-1} \odot \delta_\ell. \qquad (4)
\end{aligned}$$

The first two equations give a simple recurrence relation. The last two equations tell us how to get the gradient from the results.

Start `backprop` by treating all the quantities you need as local variables in your function. Write the code that way and then worry about the references.

```r
backprop <- function(network, input, output) {
    predicted <- forward(network, input)
    residual <- (...)
    L <- network$L
    a <- predicted$a
    z <- predicted$z
    delta <- vector("list", L)

```

Stochastic Gradient Descent

Recall that $\theta = (W_1, \ldots, W_\ell, b_1, \ldots, b_L)$.

Start by choosing initial (typically random) values for the weights and biases (θ).
Select an initial learning rate and a method for changing it as the algorithm proceeds.
Until stopping conditions are met, do:

1. Shuffle the training set and divide into m mini-batches of designated size.
2. For $k = 1, \ldots, m$, get the gradients of C_k using `backprop()` and set:

$$\theta = \theta - \eta * \frac{\partial C_k}{\partial \theta},$$

   where C_k is the cost function for mini-batch k.

3. Make any scheduled adjustments to the learning rate.

Try it.

What's Deep about Deep Learning?

We've seen that single-hidden-layer FF neural networks can approximate arbitrarily complicated functions, if given enough nodes.

What would we gain or lose by having more hidden layers? And is there a qualitative difference between say 2-10 hidden layers and 1000, if we control for the total number of model parameters?

And if we do use more hidden layers, what effect will this have on training? Will backpropogation with stochastic gradient descent still work? Will it still be efficient?

We will look at these questions. But as a preview it's worth considering why *depth* per se is viewed as desirable:

- Efficiency of representation

  Universality is not enough. Flat computer example.

- "Modularity" of representation

  Specialized layers (convolutional, pooling, softmax, ...) help design.

It is also worth considering types of networks beyond feed-forward (DAG) style. An important example of that is **recurrent** neural networks, where a node's output at one time can influence that node (or others) at a later time.

Training Deep Networks with Backprop/SGD

If we create a deep network (e.g., FF neural net with many hidden layers) and train it with a data set (using backprop/SGD), we will often find that the deep network performs not mch better than the single-hidden-layer network. Why?

One reason this happens is because the gradients of $\frac{\partial C}{\partial \theta}$ depend on a **product** across layers of terms that can vary substantially in magnitude. As a consequence, some layers end up learning much faster than others, for which the parameters move relatively slowly. Variants of this problem are called: the **vanishing gradient problem** or the **exploding gradient problem**, with the former more common with sigmoidal activation functions.

Unstable gradients are just one challenge to training. Others include multi-modality and the sensitivity of performance to tuning parameters.

As a result, a variety of computational and training techniques have been applied to improve and speed up the process. Among these is a *regularlization* technique called **dropout** (see the original paper here), where randomly selected nodes are ignored during a portion of training. That is, their activations are excluded from the forward propogation, and the any weight/bias updates during the backward propogation are not applied. This has the effect of making the network's output less sensitive to the parameters for any node, reducing overfitting and improving generalizability.

New Types of Layers

So far, we have stuck with the standard representation of the nodes in a feed-forward network. But for capturing representations of diverse kinds of data – such as images and text – it will be useful to have new types of layers.

Current approaches to building deep learning models focus on combining and tuning layers of different types in a way that is adapted to the data, the outputs, and the types of internal representations that might perform well for the task.

Convolutional Layers/Networks

Convolutional neural networks mimic the structure of the human visual system, which uses a complex hierarchy of decomposition and recombination to process complex visual scenes quickly and effectively.

Recall: one dimensional convolution

$$(a \star f)_k = \sum_j f_j a_{k-j}$$

This generalizes to multiple dimensions and inspires the filtering steps to be used below.

The key ideas behind convolutional neural networks are:

- Visual fields as inputs and **local receptive fields** for modes
- Capturing multiple features through sub-layers of the hidden layer, usually called **feature maps**.
- **Shared parameters** within each sublayer (feature map).
- **Pooling**

Local Receptive Fields

```
   Input layer                     A Feature Map
                                   within Hidden Layer
........................
........................
........................
........................            . . .
........................            . . .
........................            . . .
........................
........................
........................
........................
........................
........................
```

Key parameters:

- Filter size (size of local receptive field)
- Stride (how much filter is shifted)

This structure is repeated for each sub-layer (feature map) in the hidden layer.

Shared Weights and Biases

Each sub-layer (feature map) in the hidden layer **shares** the same weights and biases:

$$a_{\ell m,jk} = \phi(b_m + \sum_r \sum_s W_{\ell m,rs} a_{\ell-1,j+r,k+s}$$

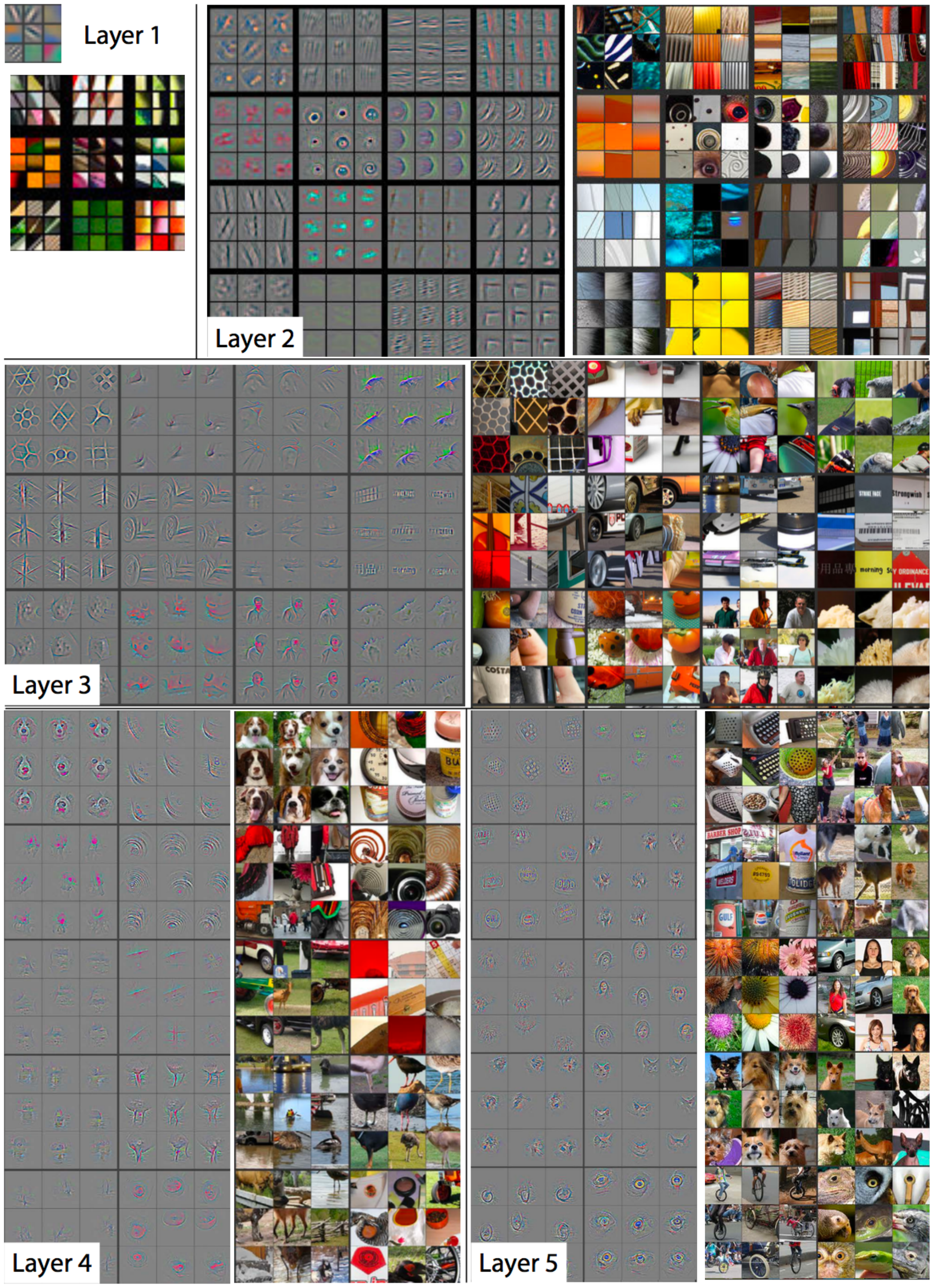
where $a_{\ell m,jk}$ is the output activation of node (j,k) in feature map m in layer ℓ, $W_{\ell m}$ is the matrix of weights that captures the local receptive field – which is sometimes called the **filter**, and $a_{\ell-1}$ is the matrix of activations from the previous layer.

Do you notice the convolution here? We can write this as

$$a_{\ell m} = \phi(b + W_{\ell m} \star a_{\ell-1}).$$

Because of this sharing, all the hidden neurons in a given feature map **detect the same feature** but at different parts of the visual field. A combination of feature maps thus decomposes an input image into a meaningful intermediate representation.

The following picture shows the layers of a fully-trained convolutional network from Zeiler and Fergus (2013) (paper here).



Pooling Layers

A **pooling** layer in the convolutional network is designed to compress the information in the feature map, primarily as a form of dimension reduction. This is a layer without weights and biases, though it may have a tuning parameter that specifies how much compression is done.

Pooling operates on output of the nodes in a feature map, producing a feature map with fewer nodes.

```
Feature map output         Pooled Feature map

1 1 2 2 . . . .
1 1 2 2 . . . .
. . 3 3 . . . .              1 2 . .
. . 3 3 . . . .              . 3 . .
. . . . . . . .              . . . .
. . . . . . . .              . . . 4
. . . . . . 4 4
. . . . . . 4 4
```

Examples:

- Max pooling – groups of nodes are reduced to one by taking the **maximum** of their output activations as the activation of the resulting node.
- ℓ^2 pooling – groups of nodes are reduced to one by taking the root-mean-square (ℓ^2 norm) of their output activations.

Sample Network

1. Input layer (32×32)
2. Convolutional ($4 \times 16 \times 16$)
   - 4×4 local receptive fields
   - Stride 2
   - Four feature maps
3. Max pooling ($4 \times 4 \times 4$)
   - 4×4 blocks
4. Fully-connected output layer

The backpropogation equations can be generalized to convolutional networks and applied here. In this case, dropout would typically be applied only to the fully connected layer.

A few common tricks for training:

- Use rectified linear units ($\phi(x) = \max(x, 0)$) in place of sigmoidal units.
- Add some small regularization (like a ridge term) to the loss function
- "Expand" the training data in various ways For example, with images, the training images can be shifted randomly several times by one or a few pixels in each (or random) directions.
- Use an ensemble of networks for classification problems

Softmax Layers

The softmax function is a mapping from n-vectors to a discrete probability distribution on $1 \ldots n$:

$$\psi_i(z) = \frac{e^{z_i}}{\sum_j e^{z_j}}$$

(Q: Why is this giving a probability distribution? What inputs give a probability of 1 for selecting 3?)
A **softmax output layer** in a neural network allows us to compute probability distributions.
If the output layer has n nodes, we obtain

$$a_{L,j} = \psi_j(z_L),$$

where z_L is the weighted input vector in the last layer.

More

Other types of networks

Recurrent Neural Networks, Long Short-Term-Memory Networks, Belief Nets, ...

Deep Learning Frameworks

- Tensor Flow (C++, Python, Java)
- DeepLearning4j (JVM)
- Keras
- Cortex (clojure)
- Theano (python no longer developed)
- Torch/PyTorch (Lua, C++/Python)
- Microsoft Cognitive Toolkit/CNTK (C++, Python)
- MXNET (C++, Python, R, JVM)
- Caffe (primarily for convolutional)