Going Deep
Statistics 650/750
Week 15 Thursday

Christopher Genovese and Alex Reinhart

06 Dec 2018

Announcements

- Notes on line in `documents` repository `weekFR.*`.

What's Deep about Deep Learning?

Deep learning is the buzzword of the moment, and an enormous amount of attention is focused on applying and developing them.

We call a multi-layer network **deep** if it has more than one hidden layer, and commonly used models have hundreds of layers.

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?
- 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 backpropagation with stochastic gradient descent still work? Will it still be efficient? If not, how should we train these networks.
- Are there useful network designs beyond feed-forward networks?

The Benefits of Depth

Think of any complicated system – your laptop, your car, a nuclear reactor – it is composed of many interconnected modules, each serving a focal purpose.

Take the computer as an example. In principle, we can arrange all the circuitry in your laptop in a single layer of logic gates. (Such a layer of binary circuits has a *universality property* and can compute any computable function.)

But such an arrangement is harder in several ways, including:

- It is hard to design.
- It is hard to understand its function given it's inputs and outputs.
- It requires many more gates/elements than a more modular design.

Consider for example how much easier it is to create a binary adder with multiple layers. Even for a simple task like computing the parity of a binary string requires exponentially more elements in a shallow circuit than a deep circuit.

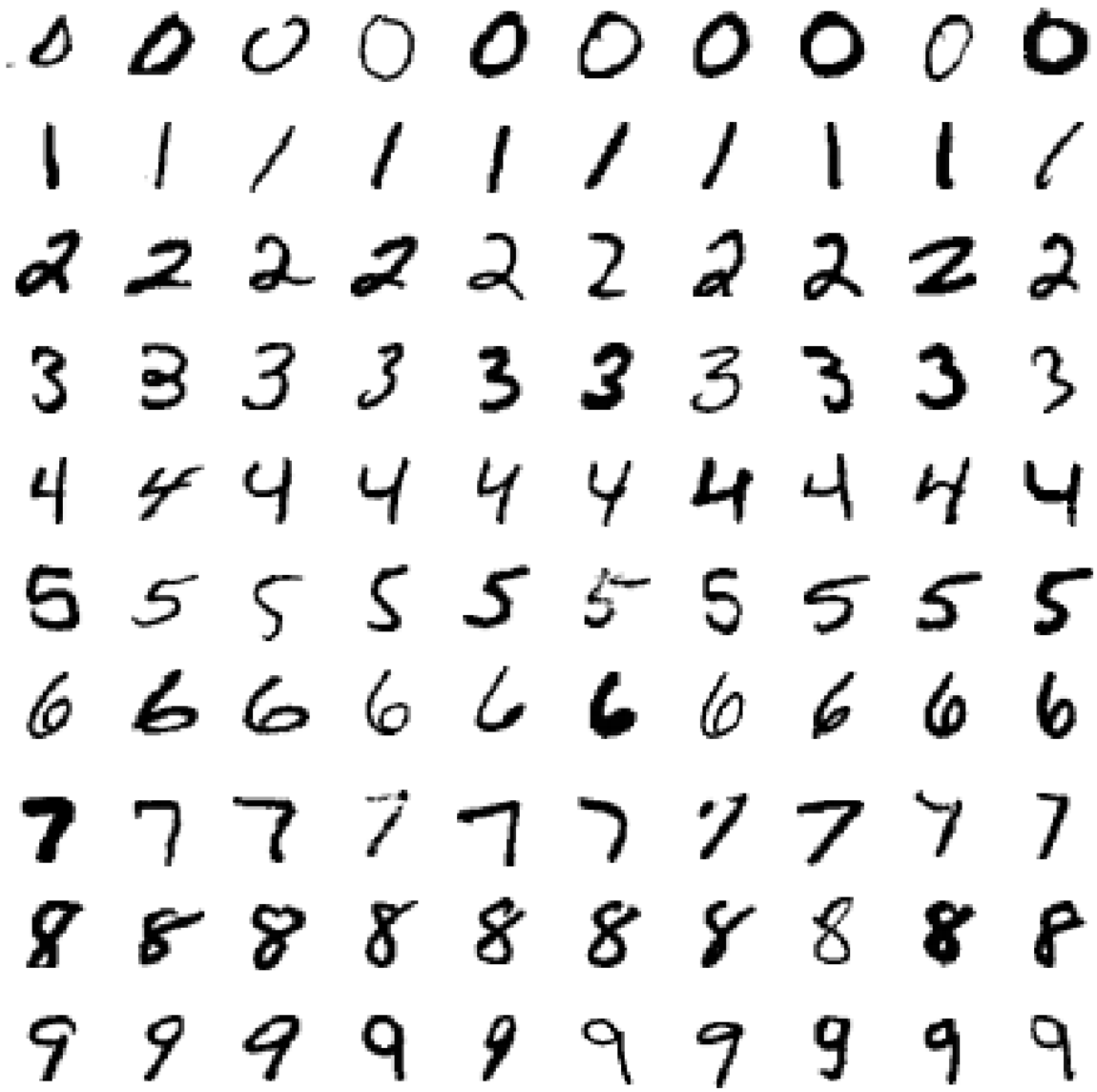
This is a useful analogy for the benefits of depth in neural network models. Deep networks:

- Are easier to design.
- Generate intermediate representations of the inputs that make it easier (though not easy) to understand the networks function.
- Require many fewer nodes than a shallow network of comparable power.

Of course, these benefits are not without costs. Most notably: **deep networks can be hard to train**.

Training Deep Networks is Hard

Consider the "Hello, World" of neural networks: the MNIST hand-written digits, a collection of 70,000 28×28 images.



A network to recognize these digits might have 784 inputs (one per pixel) and 10 outputs (one per digit).

Fitting this with a standard feed forward network and SGD on half the data gives 97% correct classification with one hidden layer, almost 96% with two hidden layers, and the rate drops a bit and oscillates in the same range as the number of layers increases. Depth is not helping.

We can look at how the parameters are changing in the network during fitting, and it reveals a key fact: for any number of hidden layers, **the earlier layers change more slowly than the later layers**. The problem gets worse as we add more layers.

This is called the **vanishing gradient problem**. A dual issue – called the **exploding gradient problem** can also occur. In either case, the underlying problem is the *instability* of the gradient in deep networks.

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.

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

Tricks of the Trade

Faster Optimizations

The venerable **gradient descent** algorithm moves generally downhill towards a (local) minimum of the cost function

$$\theta' \leftarrow \theta - \eta \frac{\partial C}{\partial \theta}.$$

But this can produce several problems: going slowly in steep valleys, getting stuck in local minima, moving over a valley, et cetera..

Momentum

Gradient descent has no memory. Instead, maintain an exponentially-weighted running average of the gradient and accumulate these.

$$\begin{aligned} m &\leftarrow \beta m - \eta \nabla C(\theta + \beta m) \\ \theta &\leftarrow \theta + m \end{aligned}$$

(This is Nesterov momentum, which is commonly used.) The β parameter is (inversely) related to "friction" in the movement of the parameter. This tends to move significantly faster than gradient descent alone, by roughly a factor of $1/(1+\beta)$.

Adaptive

Adaptive optimization algorithms like ADAM combine the ideas of momentum with a dynamic transformation that speeds up the motion on the edge of flat valleys. This requires tuning of several hyperparameters and is easy to get wrong. But when it is suitable, it allegedly perfoms well.

Regularization

Smoothing the cost function.

Dropout

A simple, interesting, and effective regularization technique is called **dropout** (see the original paper here), where randomly selected nodes are ignored during training. During each step of training, each node (excluding output nodes) is completely ignored during that step with probability p. After training, outputs are scaled by a factor $1 - p$ to stay in the training range.

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.

Data Augmentation

Transfer Learning

Learning from previously trained models

New Layer Types

The networks we have seen so far are **fully connected**: each node transmits its output to the inputs of *all* nodes in the next layer. And all of these edges (and nodes) have their own weights (and biases).

But for many kinds of data – especially images and text – new architectures turn out to be useful, even essential.

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

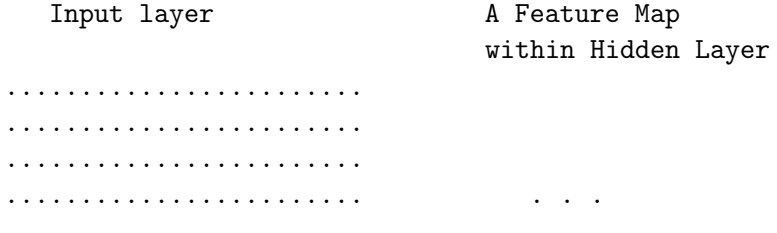
$$(a \star f)_k = \sum_j f_j a_{k-j}$$

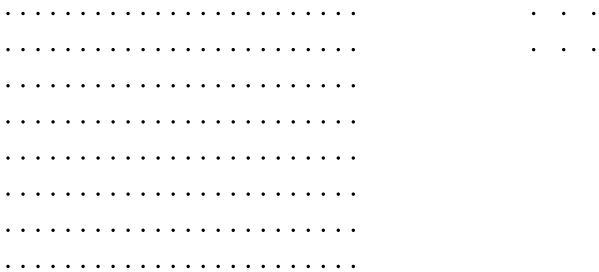
This generalizes to multiple dimensions and inspires the filtering steps to be used below.

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** – dimension reduction

Local Receptive Fields





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 $1 \dots n$:

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

(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.

Other Types of Networks

Recurrent Networks

Adding memory and time.

Autoencoders

Finding efficient representations of a data set.

Deep Learning Frameworks

- Tensor Flow (C++, Python, Java)

  Popular and powerful. Python/numpy is main interface. Other languages exists but are still catching up.

  Computational graph abstraction Good for RNNs

- MXNET (C++, Python, R, JVM:Clojure/Scala/Java)

  Easy structure, many languages, easy to extend, well supported and actively developed

- DeepLearning4j (JVM)

  Works with JVM languages, actively developed, increasing capabilities

- Keras (High Level Wrapper)

- Caffe (Python, C++/CUDA)

  Can specify declaratively (less coding) Good for feedforward, tools for tuning Good python interface Primarily for convolutional

- Torch/PyTorch (Lua, C++/Python)

  Modular, third-party packages, easy to read code, pretrained models

  More coding, not great for RNNs

- Theano (python) Seemingly no longer developed

  high-level wrappers Computational graph abstraction Good for RNNs

- Microsoft Cognitive Toolkit/CNTK (C++, Python)

Model Zoos

VGG. LeNet-*, AlexNet, ResNet, GoogLeNet, ...