Neural Networks and Deep Learning- Michael Nielsen CH6 - CNN

2017/11/07 17:03

- Intro
    - deep neural networks are much more powerful but often much harder to train than shallow neural networks
    - in this CH
        - to develop techniques which can be used to train deep networks, and apply them in practice.
        - briefly reviewing recent progress on using deep nets for image recognition, speech recognition, and other applications.
        - take a brief, speculative look at what the future may hold for neural nets, and for artificial intelligence
        - introduce deep convolutional networks
    - the focus
        - is on understanding some of the core principles behind deep neural networks
        - fundamentals, and so to prepare you to understand a wide range of current work.
    - Key techniques in CNN
        - convolutions,
        - pooling,
        - the use of GPUs to do far more training
        - the algorithmic expansion of our training data (to reduce overfitting),
        - the use of the dropout technique (also to reduce overfitting),
        - the use of ensembles of networks,
        - and others.

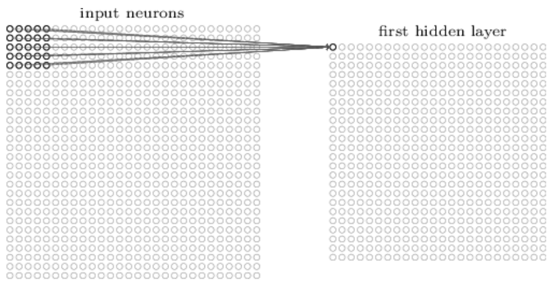
- # Introducing convolutional networks
    - Background
        - Previously, we use fully-connected NN in which every neuron in the network is connected to every neuron in adjacent layers
        - But The weird thing is that such a network architecture

does not take into account **the spatial structure of the images**

- e.g. treat the far and close pixel as the same

- so introduce CNN

- Intro
  - CNN use a special architecture which is **particularly well-adapted to classify images.**
  - Using this architecture makes convolutional networks **fast to train**
- three basic ideas of CNN: *local receptive fields*, *shared weights*, and *pooling*.
  - **Local receptive fields**
    - each neuron in the first hidden layer will be connected to a small region of the input neurons, say, e.g., a 5×5 region
    - That region in the input image is called the ***local receptive field*** for the hidden neuron.

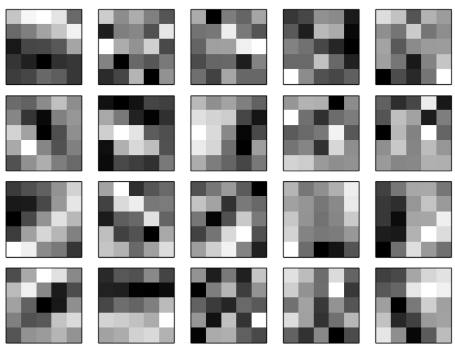


    - parameters
      - stride length
      - local receptive fields size
      - size of the 1st hidden layer: (#input -stride length + 1) ^ 2
        - (28-5+1) ^ 2 = 24 x 24
  - **Shared weights and biases**

- how it is
    - we're going to **use the *same* weights and bias for each of the 24×24 hidden neurons**, i.e. **each neuron has exactly the same weights and bias as another in the same layer,** i.e. one feature map has one set of weights and one bias
- the meaning of shared weights and bias
    - This means that **all the neurons in the first hidden layer detect exactly the same feature, just at different locations in the input image.**
    - why do this
        - it is useful to apply the same feature detector everywhere in the image so that **one layer detects one specific feature**
        - this makes CNN well adapted to the **translation invariance of images:** move a

picture of a cat (say) a little ways, and it's still an image of a cat

- Terms
    - ***feature map:* The map from the input layer to the hidden layer**
    - *shared weights, shared bias*
    - **The shared weights and bias are often said to define a *kernel* or *filter*.**
- To do image recognition we'll need more than one feature map.
    - so **a complete convolutional layer consists of several different feature maps**, i.e. several conv layers
    - e.g. 20 maps



- It's clear there is spatial structure here beyond what we'd expect at random: many of the features have clear sub-regions of light and dark. this shows our network really is learning things related to the spatial structure, even if we don't clearly know what they are.

- If you're interested in following up on that work, I suggest starting with the paper [Visualizing and Understanding Convolutional Networks] by Matthew Zeiler and Rob Fergus (2013)

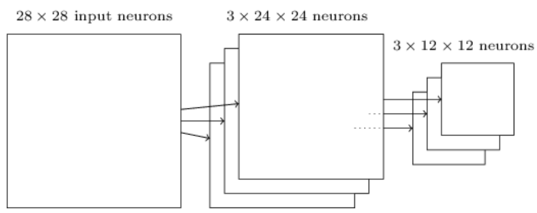
- A big advantage of sharing weights and biases
  - **it greatly reduces the number of parameters involved in a convolutional network.**

- e.g.
  - CNN: (5 x 5 + 1) x 20 = 520, only 1/3
  - fully-con: (28 x 28 + 1) x 20 = 15,700
- intuitively, it seems likely that the use of translation invariance by the convolutional layer will reduce the number of parameters it needs to get the same performance as the fully-connected model.

- **Pooling layers**
  - **where to use**
    - Pooling layers are usually **used immediately after convolutional layers.**
  - what it does
    - **to simplify the information** in the output from the convolutional layer. In detail, **to prepares a condensed feature map** from feature map
    - e.g. max pooling with 2x2 region
      - get a 12 x 12

condensed feature map

- We **apply max-pooling to each feature map separately**

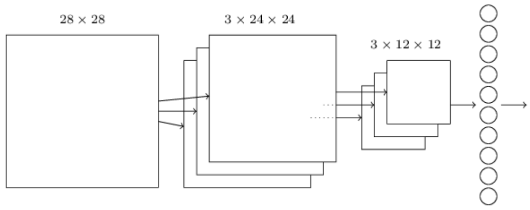


- We can think of max-pooling as a way for the network to ask whether a given feature is found anywhere in a region of the image. It then throws away the exact positional information since its exact location isn't as important as **its rough location relative to other features**. So there're fewer pooled features and thus fewer parameters required in later layers

- L2 pooling
    - works and means the same as max pooling
    - it takes the square root of the sum of the squares of the activations in the n×n region.
- and there're others. If you like, try

them out with a validation set

- **Putting it all together**
    - **the architecture**



    - **elements/hypeparameters in this CNN**
        - **Input layer**
            - begins with 28×28 input neurons, which are used to encode the pixel intensities for the MNIST image.
        - then followed by **a convolutional layer**
            - using a 5×5 local receptive field and 3 feature maps.
            - The result is a layer of 3×24×24 hidden feature neurons.
        - The next step is a max-pooling layer,
            - applied to 2×2 regions, across each of the 3 feature maps.
            - The result is a layer of 3×12×12 hidden feature neurons.
        - Output layer
            - is a fully-connected layer i.e.

this layer connects *every* neuron from the max-pooled layer to every one of the 10 output neurons.

- compare to fully-connected NN
  - This convolutional architecture is quite different
  - But the overall picture is similar: **a network made of many simple units, whose behaviors are determined by their weights and biases.**
  - the overall goal is still the same: **to use training data to train the network's weights and biases so that the network does a good job classifying input digits.**
  - as earlier in the book, we will train our network using s**tochastic gradient descent and backpropagation, but with a little modification to the derivation for convolutional and max-pooling layers of backpropagation**

- ## Convolutional neural networks in practice
  - intro
    - still the MNIST digit classification problem.
    - the code is available on GitHub
    - now we use a machine learning library known as Theano
      - features
        - makes it easy to implement

backpropagation for convolutional neural networks, since it automatically computes all the mappings involved.

- Theano is also quite a bit faster than our earlier code
- it can run code on either a CPU or, if available, a GPU.
  - Running on a GPU provides a substantial speedup which helps make it practical to train more complex networks.

- misc
  - To install Theano, follow the instructions at the project's homepage.
  - to get Theano up and running on a GPU you may find the instructions here
  - If you don't have a GPU available locally, then you may wish to look into Amazon Web Services EC2 G2 spot instances.
  - If you're using a CPU, you may wish to reduce the number of

training epochs for the more complex experiments, or perhaps omit them entirely

- **a fully-connected NN**
  - I obtained a best classification accuracy of 97.80 percent.
    - ???Using the validation data to decide when to evaluate the test accuracy helps avoid overfitting to the test data (see this earlier discussion of the use of validation data).
    - fully-connected NN: 98.04 percent accuracy obtained from a similar network architecture and learning hyper-parameters
  - difference between these two NN
    - previously, regularized the earlier network to help reduce the effects of overfitting.
      - Regularizing the current network does improve the accuracies, but the gain is only small, and so we'll hold off worrying about regularization until

later.

- used sigmoid activations and the cross-entropy cost function, the current network uses a softmax final layer, and the log-likelihood cost function.
  - I haven't made this switch for any particularly deep reason - mostly, I've done it because **softmax plus log-likelihood cost is more common in modern image classification networks.**

- with on convolutional pooling layer
  - That gets us to 98.78 percent accuracy, which is a considerable improvement over any of our previous results. we've reduced our error rate by better than a third, which is a great improvement.
  - **understanding**
    - In this architecture, we can think of **the convolutional and pooling layers as learning about local spatial structur**e in the

input training image
  - while the later, fully-connected layer **learns at a more abstract level, integrating global information from across the entire image.**
  - This is **a common pattern in convolutional neural networks.**

- inserting anther convolutional layer with same parameters after the previous one
  - we're now at 99.06 percent classification accuracy!
  - what does it even mean to apply a second convolutional-pooling layer?
    - you can think of the second convolutional-pooling layer as having as input 12×12 "images", whose "pixels" represent the presence (or absence) of particular localized features in the original input image.
    - These "images" are abstracted and condensed, but still has a lot of spatial structure, and so it makes sense to use a second convolutional-pooling layer.
  - How should neurons in the second convolutional-pooling layer respond to

these multiple input "images", 20 "images"?

- the feature detectors in the second convolutional-pooling layer have access to *all* the features from the previous layer, but only within their particular local receptive field
- just as access to rgb 3 channels in color images

- **Using tanh**
  - Personally, I did not find much advantage in switching to tanh, although I haven't experimented exhaustively
  - tanh can be a little faster than sigmoid. but as ReLU almost alwasys outperforms sigmoid and tanh, we choose ReLU instead.
- **Using rectified linear units with** [l2 regularization]() on the previous two convolutional pooling layer NN
  - classification accuracy: 99.23 percent
    - It's a modest improvement over the sigmoid results (99.06)
    - across all my experiments I found that networks based on **rectified linear units consistently outperformed networks based on sigmoid activation functions**.
  - the reason
    - on the basis of hunches or

heuristic arguments, no clear answers so far

- **Expanding the training data**
  - Another way we may hope to improve our results is by **algorithmically expanding the training data**.
  - A simple way of expanding the training data is
    - **to displace each training image by a single pixel**, either up one pixel, down one pixel, left one pixel, or right one pixel.
    - 50,000 => 250,000
    - But, in fact, expanding the data turned out to **considerably reduce the effect of overfitting**.
    - 99.37 percent training accuracy.
  - in 2003 Simard, Steinkraus and Platt
    - 99.6 percent
    - They did this by rotating, translating, and skewing the MNIST training images
    - and a process of "elastic distortion", a way of emulating the random oscillations hand muscles undergo when a person is writing
  - [Problem]

- It may seem surprising, then, that our network can learn more when all we've done is translate the input data. Can you explain why this is actually quite reasonable?

- **Inserting an extra fully-connected layer**
  - to expand the size of the fully-connected layer, 100 originally
    - I tried with 300 and 1,000 neurons, obtaining results of 99.46 and 99.43percent, respectively. That's interesting, but **not really a convincing win over the earlier result** (99.37 percent).
  - adding an extra fully-connected layer
    - 100-hidden neuron
      - 99.43 percent
      - Again, the expanded net isn't helping so much
    - 300 and 1,000 neurons
      - 99.48 and 99.47 percent.
      - That's encouraging, but still falls short of a

really decisive win.

  - Is it that the expanded or extra fully-connected layers really don't help with MNIST? Or might it be that our network has the capacity to do better, but we're going about learning the wrong way?

- try stronger regularization techniques, dropout , to reduce the tendency to overfit.
  - the basic idea of dropout is **to remove individual activations at random** while training the network. This makes the model **more robust to the loss of individual pieces of evidence**, and thus less likely to rely on particular idiosyncracies of the training data.
  - 0.5 keepprob dropout applied to the two `FullyConnectedLayer`
    - accuracy of 99.60 percent, which is a **substantial improvement** over our earlier results
  - There are two changes worth

noting.

- reduce epochs from 60 to 40
  - dropout reduced overfitting, and so we learned faster.
- the fully-connected hidden layers have 1,000 neurons, not the 100 used earlier.
  - Of course, dropout effectively omits many of the neurons while training, so some expansion is to be expected.

- 300 hidden neurons work very slightly worse

- **Why we only applied dropout to the fully-connected layers**
  - Because the convolutional layers have considerable inbuilt resistance to overfitting
  - The reason is that the **shared weights** mean that convolutional filters are **forced to learn from across the entire image**. This makes them **less likely to pick up on local idiosyncracies** in the training data.

- **Using an ensemble of networks**
  - the basic idea it to **create several neural networks, and then get them to vote to determine the best classification**.
  - e.g. 5 different neural networks using the prescription above
    - 99.67 percent accuracy
  - why this help
    - Even though the networks would all have similar accuracies, they might well make different

errors, due to the different random initializationss so the accurancy might be better

- this kind of ensembling is **a common trick with both neural networks and other machine learning techniques**.

- when the accuracy is high, check the training data
  - Our network is getting near to human performance. Since even a careful human makes the occasional mistake

- **Going further**
  - Rodrigo Benenson has compiled an [informative summary page](), showing progress over the years
    - If you dig through the papers you'll find many interesting techniques, and you may enjoy implementing some of them.
- **Why are we able to train deep net like the above?**
  - problem
    - we saw that the gradient tends to be quite unstable, tending to either vanish or explode. Since the gradient is the signal we use to train, this causes problems.

- How have we avoided those problems?
  - we haven't. Instead, we've **done a few things that help us proceed anyway**.
    - (1) Using convolutional layers greatly reduces #parameters in those layers, making the learning problem much easier;
    - (2) Using more powerful regularization techniques (notably dropout and convolutional layers) to reduce overfitting, which is otherwise more of a problem in more complex networks;
    - (3) Using rectified linear units instead of sigmoid neurons, to speed up training -

empirically, often by a factor of **3-5**;
    - (4) Using GPUs and being willing to train for a long period of time.
  - and other ideas
    - making use of sufficiently large data sets (to help avoid overfitting);
    - using the right cost function (to avoid a learning slowdown);
    - using good weight initializations (also to avoid a learning slowdown, due to neuron saturation);
    - algorithmically expanding the training data.
- **What can be called a deep networks, anyway?**
  - DNN > 2 layers
  - The real breakthrough in deep learning was to realize that it's practical to go

beyond the shallow 1- and 2-hidden layer networks that dominated work until the mid-2000s.

- But beyond that, the number of layers is not of primary fundamental interest. Rather, the use of deeper networks is a tool to use to help achieve other goals - like better classification accuracies.

- **A word on procedure**
    - I've presented a cleaned-up narrative, omitting many experiments - including many failed experiments.
    - Getting a good, working network can involve a lot of trial and error, and occasional frustration. In practice, you should expect to engage in quite a bit of experimentation.
    - read more in how to choose a neural network's hyper-parameters,