CS 446 Machine Learning Fall 2016 Oct 04, 2016

Neural Networks 2

Professor: Dan Roth *Scribe: C. Cheng, C. Cervantes*

Overview

- Convolutional Neural Networks
- Recurrent Neural Networks

1 Introduction

There are different variations of the neural networks algorithms for different structures. Today we are going to see two different architectures of neural networks. Although they are very different, but the underlying ideas are the same.

2 Receptive fields and dealing with image inputs

Before jumping into the two new architectures, let's start with some ideas we didn't address last time which are fundamental to all machine learning problems. It deals with encoding the input to fit into mathematical models. As humans, we have sensory elements, like ears and eyes to sense the environment. That is how we get information from the nature. Neural networks also need eyes and ears. They have to encode information, like images and sentences, to neural networks language. That is a big challenge. People have come up with different ideas to do this for different tasks and different types of data, but still we don't have an ideal solution for it. We will just go through a bunch of ideas.

The receptive field of an individual sensory neuron is the particular region of the sensory space (e.g., the body surface, or the retina) in which a stimulus will trigger the firing of that neuron. For example, in the auditory system, receptive fields can correspond to volumes in auditory space. However, designing proper receptive fields for the input Neurons is a significant challenge.
To be more concrete, consider a task dealing with image inputs. We are using neural networks to predict something from the image, such as whether there is a face in it. Receptive fields should give expressive features from the raw input to the system by converting the image to inputs to the neural network. How

would you design the receptive field for this problem? One approach could be

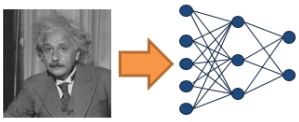

Figure 1: Task with image inputs

designing a filter to tell how "edgy" the picture is, and give the value to the neural network. Based on this encoding, the whole picture, no matter how big it is, is converted to a real-valued signal. Although it might not be an ideal approach to detecting faces, it is a very good starting point.

Another idea is that for every pixel in the input image, give all the pixels to each unit in the input layer. It will work even when you have images with different sizes. However, the problem is that this network does not have any understanding of the local connections between pixels, i.e., spatial correlations are lost.

(a) All image info to all units (b) Image divided into blocks

Rather than giving all the information of the image to all units in the input layer, we could create small blocks within the image. Each unit is then responsible for a certain block in the image. As shown in Figure (b) above, the blocks are disjoint. Inside each block, we still have the problem of losing spatial correlations. Another issue is when we are moving from block to block, the smoothness of moving from pixel to pixel is lost. Therefore, again this approach is not ideal.

3 Convolutional layers

These days, what people commonly do is creating filters to capture different patterns in the input space. For images, these filters are matrices.

Each of the filters scans over the image and creates different outputs. For each filter, there is a different output. For example, a filter can be designed to be sensitive to sharp corners, which is the idea we just mentioned. Using the filters, not only the spatial correlations are preserved, desired properties of the image

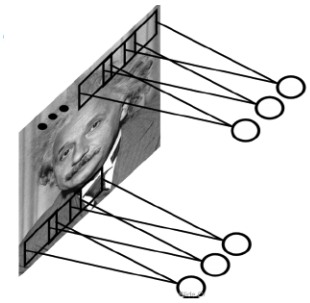

Figure 3: Convolutional layer

can also be obtained. This kind of idea starts the discussion of convolutional neural networks, which are huge in the field of computer vision now.

3.1 Convolutional operator

Convolution is a mathematical operator (denoted by $*$ symbol), in one-dimension it is defined as

$$(x * h)(t) = \int x(\tau)h(t-\tau)d\tau$$

$$(x * h)[n] = \sum_m x[m]h[n-m]$$

for continuous and discrete cases, respectively. In the definition above, x and h are both functions of t (or n). Let's say x is the input, and h is the filter. Convolution of x and h is just an integration of product of x and flipped h.

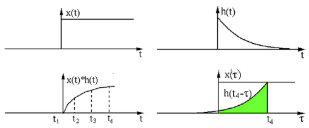

Figure 4: An example of convolution

Convolution is very similar to cross-correlation, except that in convolution one of the functions is flipped.

In two-dimension, the idea is the same. Flip one matrix and slide it on the other matrix.

In the example shown below, the image is convolved with the sharpen kernel matrix. First, flip the matrix both vertically and horizontally. Then, starting from one corner of the image, multiply this matrix element-wise with the matrices representing blocks of pixels in the image. Sum them up, and put it

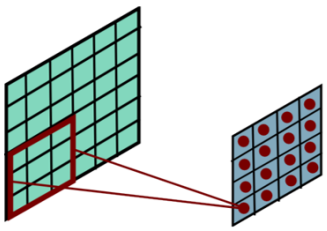

Figure 5: Convolution in 2D

in another image. Keep doing this for all blocks of size 3-by-3 over the whole image.

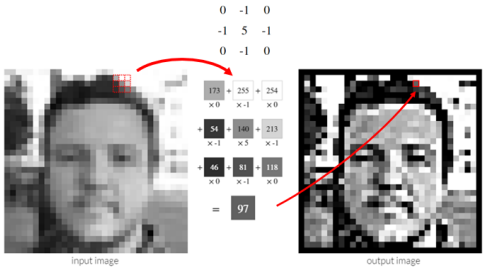

Figure 6: Example: sharpen kernel

To deal with the boundaries, we can either start within the boundaries, or pad zero values around the image. The result is going to be another picture, sharper than the previous one. Likewise, we can design filters for other purposes.

In practice, Fast-Fourier-Transform (FFT) is applied to compute the convolutions. For n inputs, complexity of convolution operator is $n \log n$. For two-dimension, each convolution takes $MN \log MN$ time, where the size of input is MN.

3.2 Convolutional and pooling layer

So far, we have the inputs and the convolutional layer. The convolution of the input (vector/matrix) with weights (vector/matrix) results in a response vector/matrix. We can have multiple filters (four in the example shown in the figure below) in each convolutional layer, each producing an output.

If we have multiple channels in the input, for example, a channel for blue color and a channel for green, for each channel, we will have a set of outputs.

Now the sizes of the outputs depend on the sizes of the inputs. How are we going to handle this? People in the community are actually using something

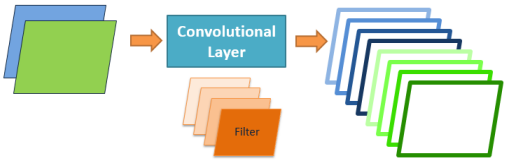

Figure 7: Convolutional layer

very simple that is called pooling layer, which is a layer that reduces input of different sizes to a fixed size.

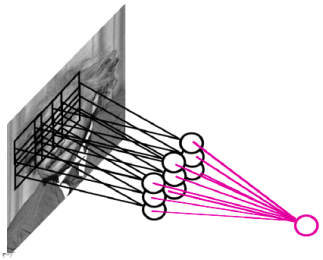

Figure 8: Pooling

There are different variations of pooling. For max pooling, simply take the value of the block with the largest value. One could also take the average value of blocks, or any other combinations of the values. So values can be combined and give you blocks of fixed size.

Different variations are listed as follows:

- Max pooling:
$$h_i[n] = \max_{i \in N(n)} \tilde{h}[i]$$
- Average pooling:
$$h_i[n] = \frac{1}{n} \sum_{i \in N(n)} \tilde{h}[i]$$
- L-2 pooling:
$$h_i[n] = \frac{1}{n} \sqrt{\sum_{i \in N(n)} \tilde{h}^2[i]}$$

4 Convolutional nets

From an input, convolve it with a filter to get a bunch of outputs of variable sizes. Then, do pooling to shrink the size of output to the desired size. This is

what we have so far. Combining convolutional layer and pooling layer gives us one stack in a convolutional network.

Figure 9: One-stage convolutional net

Then, stack the one-stage structure as many time as we want. The size of output depends on the number of features, channels and filters and design choices. We can give an image as input, and get a class label as prediction. This whole thing is a convolutional network.

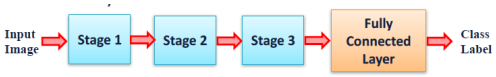

Figure 10: Whole system

4.1 Training a ConvNet

Remember in backprop we started from the error terms in the last stage, and passed them back to the previous layers, one by one. The same procedure from Back-propagation applies here.

For the pooling layer, consider, for example, the case of max pooling. This layer only routes the gradient to the input that has the highest value in the forward pass. Hence, during the forward pass of a pooling layer it is common to keep track of the index of the max activation (sometimes also called the switches) so that gradient routing is efficient during backpropagation. Therefore, we have $\delta = \frac{\partial E_d}{\partial y_i}$.

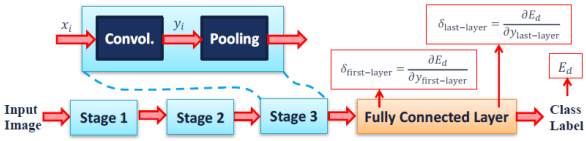

Figure 11: Backpropagation for ConvNets

Derivations are detailed in the lecture slides.

4.2 Example of ConvNets

To get more intuition about ConvNets, let's look at the following example of identifying whether there is a car in the picture.

In the first stage, we have convolutions with a bunch of filters and more sets of convolutions in the following stages. When we are training, what are we really training?

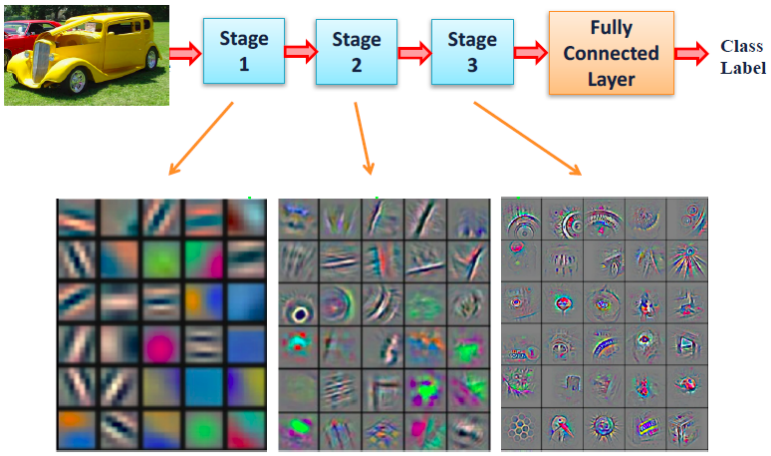

Figure 12: Example of identifying a car with ConvNet

What we are training are the filters. Our task is to identify the existence of a car in the picture, but at each stage, there are bunch of filters. Usually, in the early stages, the filters are more sensitive to more general and less detailed elements of the picture, as shown in the figures above. In later stages, more detailed pieces are favored.

4.3 ConvNet roots

In 1980s, Fukushima designed network with same basic structure but did not train by backpropagation. The first successful applications of Convolutional Networks was done by Yann LeCun in 1990s (LeNet).

The LeNet was used to read zip codes, digits, etc.

There are many variants nowadays, such as GoogLeNet developed in Google, but the core idea is the same.

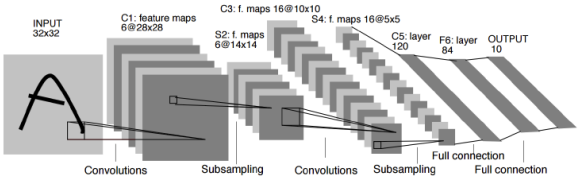

Figure 13: Example system: LeNet

4.4 Depth matters

It used to be the case that people prefer using small and shallow networks, but recently, people are coming up with deeper and deeper networks. As shown in the figure below, the error decreases as depth increases in recent years.

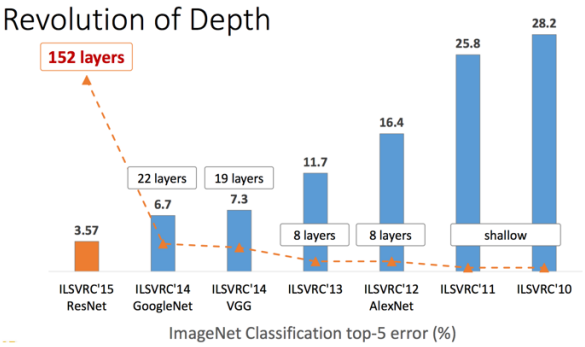

Figure 14: Revolution of Depth

Since expressivity of any function can be achieved by shallow networks, people didn't even think about using deep networks. However, someone happened to use deep networks and it turned out working better, but we have almost zero understanding of why it is happening.

4.5 Practical tips

skipped in class

4.6 Debugging

skipped in class

4.7 CNN on vectors

skipped in class

5 Recurrent neural networks

5.1 Equivalence between RNN and Feed-forward NN

In the feed-forward neural networks architecture, there are no cycles, because we have to propagate the error back. A feed-forward neural network is just a DAG that computes a fixed sequence of non-linear learned transformations to convert an input pattern into an output pattern.

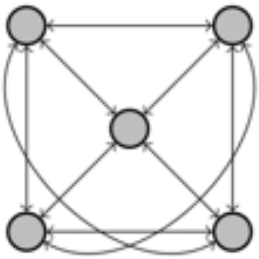

Figure 15: RNN is a digraph

A recurrent neural network is a digraph that has cycles. A cycle can act as a memory. The hidden state of a recurrent net can carry along information about a potentially unbounded number of previous inputs.

The figure below is a cyclic representation of a neural network.

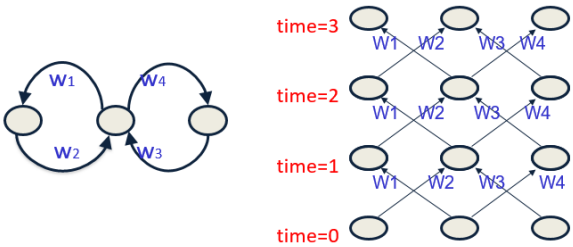

Figure 16: Cyclic representation of a neural network

Assume that there is a time delay of 1 in using each connection. W_1, W_2, W_3, and W_4 are the weights. Starting from the initial states at time=0, keep reusing the same weights, the cycles are unwrapped over time.

5.2 An NLP task

Training a general RNNs can be hard. Here we will focus on a special family of RNNs, that is prediction on chain-like input. For example, POS tagging.

$X =$	This	is	a	sample	sentence
$Y =$	DT	VBZ	DT	NN	NN

Figure 17: POS tagging words in a sentence

Given a sentence of words, the RNN should output a POS tag for each of the word in the sentence. There are several issues we have to handle. First of all, there are connections between labels. For example, verbs tend to appear after adverbs. Second, some sentences tend to be longer than the other ones. We have to handle variable sizes of inputs. Also, there is interdependence between elements of the inputs. The final decision is based on an intricate interdependence of the words on each other.

5.3 Chain RNN

To handle the chain-like input, we can design an RNN with a chain-like structure.

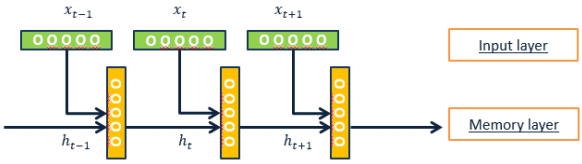

Figure 18: An RNN with a chain-like structure

As shown in the above figure, x_t's are the values obtained from the input space. They are vector representations of the words. Hidden (memory) units are another set of vectors. They are computed from the past memory and current word. Given an input, it is combined with a current hidden state, and another hidden state is got. Each h_t contains information about previous inputs and previous hidden units h_{t-1}, h_{t-2}, etc. They summarize the sentence up to each time. Here, time refers to words in the sentence.

The structure shown in the above figure is the same structure being applied multiple (three in the figure) times. It is not a three-layer stack model. Instead, it is a fixed structure, whose output is applied again to the same structure. It is like applying it multiple times to itself. That is a big difference from the fully-connected feed-forward networks.

Depending on the task, prediction can be made on each word or each sentence. That is really a design choice.

5.4 Bi-directional RNN

Rather than having just one-directional structure, in which the prediction would only depend on previous contexts, you can have bi-directional structures like the one shown in the figure below.

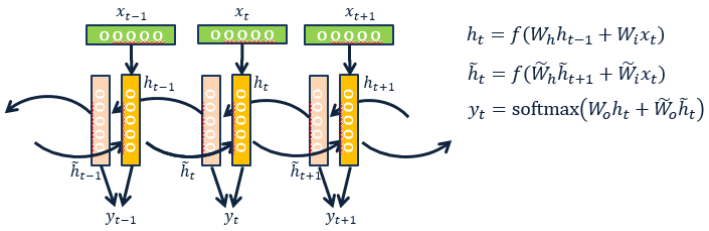

Figure 19: An RNN with a bi-directional structure

Using the same idea, the model can be made further complicated, like the stack of bi-directional networks shown in the below figure.

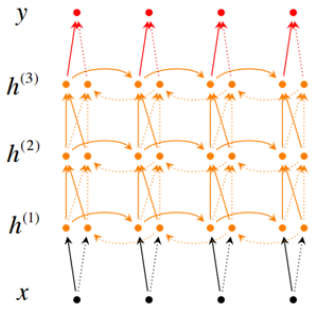

Figure 20: stack of bi-directional networks

5.5 Training RNNs

Before diving into any derivations, we have to be clear with the parameters to be learned. What are we trying to learn here? Consider the POS tagging task, in which we have decided to represent each word with a vector of fixed size. We can initialize them with random weights. The representations for each word is a set of parameters that we have to train. The input representations are then multiplied by a matrix to get the hidden state from the previous state. Then, the hidden state is multiplied by another matrix to get to the next hidden state. This is how we transfer from a hidden state to another hidden state. These matrices that are multiplied are another set of parameters to train. Given the hidden states, multiply them with a matrix and apply the softmax function to

get a distribution over the output labels. This matrix is another parameter we have to train.

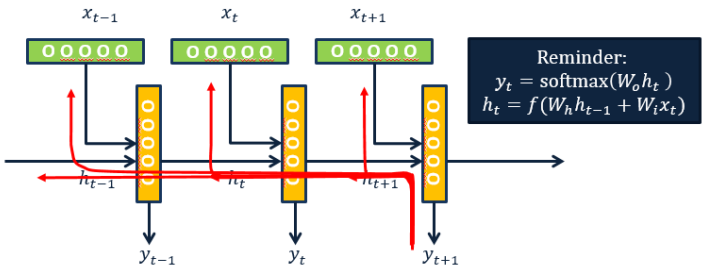

Figure 21: Training an RNN

To summarize, the parameters we have to train include the matrix multiplied to generate outputs, the matrix that gives the hidden state from the previous state, and the matrix that gives the hidden state from the vector representations of the input values.

To actually train the RNN, we need to generalize the same ideas from back-propagation for feed-forward networks.

As a starting point, we first get the total output error $E(\mathbf{y}, \mathbf{t}) = \sum_{t=1}^{T} E_t(y_t, t_t)$, which is computed over "time" across the sentence. Then, we propagate the gradients of this error in the outputs back to the parameters. The gradients w.r.t. matrix W are calculated as

$$\frac{\partial E}{\partial W} = \sum_{t=1}^{T} \frac{\partial E_t}{\partial W}$$

where

$$\frac{\partial E_t}{\partial W} = \sum_{t=1}^{T} \frac{\partial E_t}{\partial y_t} \frac{\partial y_t}{\partial h_t} \frac{\partial h_t}{\partial h_{t-k}} \frac{\partial h_{t-k}}{\partial W}$$

What is a little tricky here is to calculate the gradient of a hidden state w.r.t another previous hidden state. It can actually be calculated as the product of a bunch of matrices.

$$\frac{\partial h_t}{\partial h_{t-1}} = W_h \text{diag}[f'(W_h h_{t-1} + W_i x_t)]$$

$$\frac{\partial h_t}{\partial h_{t-k}} = \prod_{j=t-k+1}^{t} \frac{\partial h_j}{\partial h_{j-1}} = \prod_{j=t-k+1}^{t} W_h \text{diag}[f'(W_h h_{t-1} + W_i x_t)]$$

5.6 Vanishing/exploding gradients

The gradients of the error function depend on the value of $\frac{\partial h_t}{\partial h_{t-k}}$, and the value of this term can get very small or very large, because it is a product of k terms. In such cases, the gradient $\frac{\partial E_t}{\partial W}$ would become super small or large. This phenomenon is called vanishing/exploding gradients. They are quite a prevalent and serious issue.

In an RNN trained on long sequences (e.g. 100 time steps), the gradients can easily explode or vanish. Therefore, RNNs have difficulty dealing with long-range dependencies.

Many methods have been proposed to reduce the effect of vanishing gradients, although it is still a problem. Those approaches include introducing shorter path between long connections, abandoning stochastic gradient descent in favor of a much more sophisticated Hessian-Free (HF) optimization, and adding fancier modules that are robust to handling long memory, e.g., Long Short Term Memory (LSTM).