CS 446 Machine Learning Fall 2016 Oct 04, 2016

Neural Networks

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

Overview

- Introduction
- Learning rules
- Over-training prevention
- Input-output coding
- Auto-associative networks
- Convolutional Neural Networks
- Recurrent Neural Networks

1 Introduction

Neural networks can be thought of as a robust approach to approximating real-valued, discrete-values, and vector-valued target functions. They're particularly effective for complex and hard to interpret input data, and have had a lot of recent success in handwritten character recognition, speech recognition, object recognition, and some NLP problems.

We can write neural networks as functions such that

$$\text{NN} : X \to Y,$$

where X can be a continuous space $[0,1]^n$ or a discrete space $\{0,1\}^n$ and $Y = [0,1]$ or $\{0,1\}$, correspondingly. In this way, it can be thought of as a classifier, but it can also be used to approximate other real-valued functions.

Neural networks themselves were named after – and inspired by – biological systems. However, there is actually very little connection to this architecture and anything we know (thought we don't know a lot) about a real neural system. In essence, a neural network is a machine learning algorithm with a specific architecture.

We are currently on rising part of a wave of interest in neural network architectures, after a long downtime from the mid-nineties, for multiple reasons. The wave came back in the last five years or so, because of better computer architecture (GPUs, parallelism) and a lot more data than before. Tiny algorithmic changes were made since the late-eighties for the optical character recognition problem (OCR), but the recent change has been driven by the architecture.

Interestingly, one emerging perspective on neural networks is that of intermediate representations. In the past, neural networks were thought of as one of of the family of function approximators (perceptron, boosting, decision trees, etc.). Now, there is a belief that the hidden layers – that is, intermediate neural network representations – that age generated during learning may be meaningful. Ideas are being developed on the value of these intermediate representations for transfer learning etc.

1.1 Basic units

In a linear function, we're interested in the basic unit $o_i = w \cdot x$: the dot product of the weights and the input that gives an output. In neural networks, however, we want to introduce non-linearity to increase expressivity. If all units were linear, stacking them together would still be a linear function, and thus we add no expressivity.

One way to add this non-linearity is to take the sign of the dot product, such that $o_i = \text{sgn}(w \cdot x)$. However, this unit would not be differentiable and thus would be inappropriate for gradient descent.

In neural networks, we must propagate error from the top of the network to the bottom. To do so using gradient descent, we must use threshold units that are differentiable.

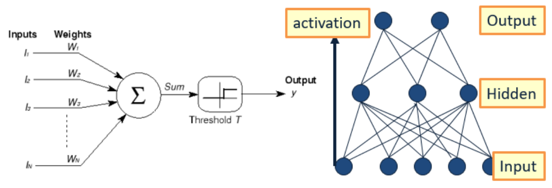


Figure 1: Linear units in the network

One option for introducing differentiable non-linearity is with a smooth non-linear approximation $o_i = [1 + \exp(-w \cdot x)]^{-1}$.

As shown in Figure 2, it looks quite similar to a step function. This key idea was invented in 1970s, though not originally in the context of neural networks.

The basic operation is the linear sum. The net input to a unit is defined as

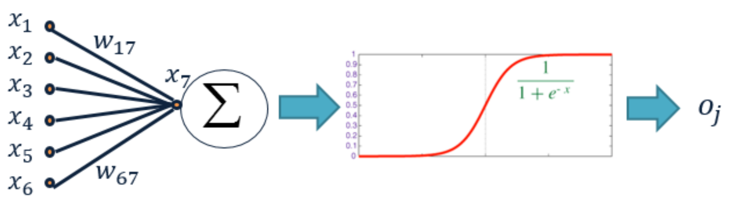


Figure 2: A differentiable threshold unit

$\text{net}_j = \sum w_{ij} x_i$, and the output of a unit is given by

$$o_j = \frac{1}{1 + \exp(-(\text{net}_j - T_j))},$$

where T_j is the threshold. This threshold is something we don't necessarily have to carry with us, as it behaves like a weight.

1.2 History: neural computation

In 1943, McCollough and Pitts showed that linear threshold units were expressive, could be used to compute logical functions, and – by properly setting weights – could be used to build basic logic gates

- AND: $w_{ij} = T_j/n$
- OR: $w_{ij} = T_j$
- NOT: use negative weight

Given these basic gates, arbitrary logic circuits, finite-state machines, and computers can be built. Also, since DNF and CNF are universal representations, any Boolean function could be specified using a two layer network (with negation).

Learning came just a bit later. In 1949, Hebb suggested that if two units are both active (firing) then the weights between them should increase:

$$w_{ij} = w_{ij} + R o_i o_j,$$

where R is a constant called learning rate, acting upon the product of activations.

Following that, in 1959, Rosenblatt suggested that when a target output value is provided for a single neuron with fixed input, it can incrementally change weights and learn to produce the output using the perceptron learning rule. This led to the perceptron learning algorithm, which is really the basic learning machinery that we use in machine learning.

1.3 Learning Rules

If the neuron sees x_i as input, and the output it produces is o_j, given the target output t_j for the output unit, the perceptron learning algorithm updates weights according to

$$w_{ij} \leftarrow w_{ij} + R(t_j - o_j)x_i.$$

Specifically, perceptron updates in a mistake-driven way. If the output is correct, the weights won't be changed. Only if the output is wrong, we update the weights for all inputs which are 1.

About the same time, Widrow and Hoff developed a slightly different update rule, which is called the Widrow-Hoff rule. Essentially, an least-mean-square error is defined, such that

$$\text{Err}(\mathbf{w}^{(j)}) = \frac{1}{2}\sum_{d \in D}(t_d - o_d)^2,$$

where $o_d = \sum_i w_{ij}x_i = \mathbf{w}^{(j)} \cdot \mathbf{x}$ is the output of linear unit on example d, and t_d is the target output for example d. They suggested to use gradient descent to determine the weight vector that minimizes the error.

These update rules are related to gradient descent. Even multiplicative update rules, which are not directly related to gradient descent, can be thought of as exponential gradient descent. The gradient descent is happening in an exponential way, rather than additive.

2 Learning with a Multi-Layer Perceptron

2.1 Intuition

So far, everything we've done so far has involved modifying the feature space; we start with a space that a linear function cannot express, and move to a representation that can be expressed linearly.

One question that can be asked is: can the learning algorithm learn this expressive representation directly?

Decision trees are one family of algorithms that do this, and multi-layer neural networks is another. By stacking several layers of threshold elements, where each layer uses the output as the previous as input, a multi-layer neural network can overcome the expressivity limitation of a single threshold element.

2.2 Learning

It is easy to learn the top layer of a network, as it is just a linear unit; the feedback (truth) given at the top layer, the weights of the layer below can be updated using either the perceptron update rule or gradient descent, depending on which loss function is applied.

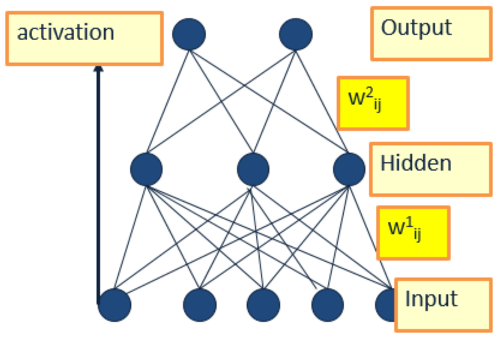


Figure 3: Stack of several layers of threshold elements

The intuition for why downstream weights can be updated in this way is given by the chain rule

If y is a function of x, and z is a function of y, then z is a function of x

To differentiate z relative to x, then, we must also differentiate that intermediate function, such that

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y}\frac{\partial y}{\partial z}$$

In the case of neural networks, all the activation units are differentiable, as is the output of the network. Thus, if we can define an error function (eg. sum of squares) that is a differentiable function of the output, we can evaluate the derivatives of this error with respect to the weights, and find weights that minimize the error using gradient descent or other methods. This method of propegating error from the top layers to lower layers is called backpropagation.

3 Backpropagation

The backpropagation algorithm learns the weights for a multi-layer network, given a network with a fixed set of units and interconnections. The error function used here is the squared error (LMS). Every other error function could work, but here the learning rules are developed according to LMS. Since there could be multiple output units, we define the error as the sum over all the network output units

$$\mathrm{Err}(\mathbf{w}) = \frac{1}{2} \sum_{d \in D} \sum_{k \in K} (t_{kd} - o_{kd})^2,$$

where D is the set of training examples and K is the set of output units.

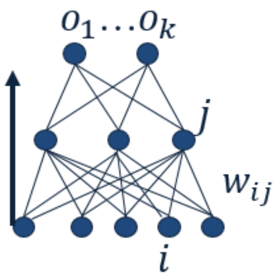


Figure 4: A multi-layer network with k output units

This is used to derive the (global) learning rule which performs gradient descent in the weight space in an attempt to minimize the error function

$$\Delta w_{ij} = -R \frac{\partial \mathrm{E}}{\partial w_{ij}}$$

.

3.1 Derivation of the Learning Rule

For each training example d, every weight w_{ij} is updated incrementally by adding to it Δw_{ji}

$$\Delta w_{ij} = -R \frac{\partial E_d}{\partial w_{ij}}$$

where R is the learning rate, and E_d is the error on training example d, summed over all output units in the network

$$E_d(\mathbf{w}) = \frac{1}{2} \sum_{k \in K} (t_k - o_k)^2.$$

Here t_k is the target output value of unit k for training example d, and o_k is the output of unit k given training example d.

Notice that w_{ij} can only influence the output through net_j, such that

$$\mathrm{net}_j = \sum w_{ij} x_{ij}$$

where x_{ij} is the ith input to unit j (thus x_{ij} is from the previous layer of unit j).

Therefore, we can use the chain rule to write

$$\frac{\partial E_d}{\partial w_{ij}} = \frac{\partial E_d}{\partial \text{net}_j}\frac{\partial \text{net}_j}{\partial w_{ij}} = \frac{\partial E_d}{\partial \text{net}_j} x_{ij}.$$

Now our task is to derive a convenient expression for $\frac{\partial E_d}{\partial \text{net}_j}$. We consider two cases: the case where unit j is an output unit in the network, and the case where unit j is a hidden unit.

3.1.1 Derivation of Learning Rules for Output Unit Weights

Just as w_{ij} can only influence the rest of the network only through net_j, net_j can influence the network only through o_j. Therefore, we can again invoke the chain rule to write

$$\frac{\partial E_d}{\partial \text{net}_j} = \frac{\partial E_d}{\partial o_j}\frac{\partial o_j}{\partial \text{net}_j}.$$

Recall that $E_d = \frac{1}{2}\sum_{k\in K}(t_k - o_k)^2$, thus

$$\frac{\partial E_d}{\partial o_j} = \frac{\partial}{\partial o_j}\frac{1}{2}\sum_{k\in K}(t_k - o_k)^2$$

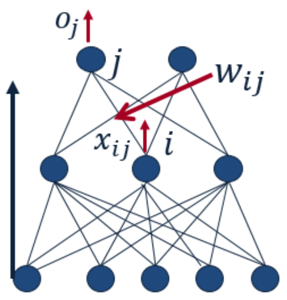


Figure 5: Output unit weights

The derivatives $\frac{\partial}{\partial o_j}(t_k - o_k)^2$ will be zero for all output units k except when $k = j$. Therefore,

$$\frac{\partial E_d}{\partial o_j} = \frac{\partial}{\partial o_j}(t_j - o_j)^2 = \frac{1}{2}2(t_j - o_j)\frac{\partial(t_j - o_j)}{\partial o_j}$$

Next, consider the sigmoid function $y = \frac{1}{1+\exp(-(x-T))}$, its derivative w.r.t. x is given by

$$\frac{\partial y}{\partial x} = \frac{\exp(-(x-T))}{(1+\exp(-(x-T)))^2} = y(1-y)$$

Since $o_j = \frac{1}{1+\exp(-(\text{net}_j - T_j))}$ is a sigmoid function, we have

$$\frac{\partial o_j}{\partial \text{net}_j} = o_j(1-o_j)$$

Then, we have

$$\frac{\partial E_d}{\partial \text{net}_j} = -(t_j - o_j)o_j(1-o_j)$$

Hence, the learning rule of weights of output units can be written as

$$\Delta w_{ij} = -R\frac{\partial E_d}{\partial w_{ij}} = R(t_j - o_j)o_j(1-o_j)x_{ij}$$

or

$$\Delta w_{ij} = R\delta_j x_{ij}$$

where $\delta_j = (t_j - o_j)o_j(1-o_j)$ is dependent on the output and its feedback.

3.1.2 Derivation of Learning Rules for Hidden Unit Weights

Now we already know how to update the output layer, we need to figure out how to propagate the error to hidden units. In the case of where j is a hidden unit in the network, the derivation of the training rule for w_{ij} must take into account the indirect ways in which w_{ij} can influence the network outputs and thus E_d. For this reason, we will find it useful to refer to the set of all units immediately downstream of unit j in the network (i.e., all units whose direct inputs include the output of unit j). We denote this set of units by downstream(j).

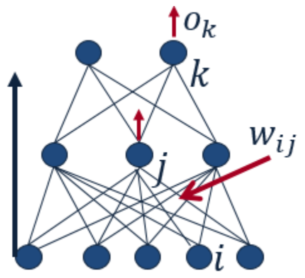


Figure 6: Hidden unit weights

Notice that net$_j$ can influence the network outputs only through the units in downstream(j). Therefore, we have

$$\frac{\partial E_d}{\partial \text{net}_j} = \sum_{k \in \text{downstream}(j)} \frac{\partial E_d}{\partial \text{net}_k}\frac{\partial \text{net}_k}{\partial \text{net}_j}$$

$$= \sum_{k \in \text{downstream}(j)} -\delta_k \frac{\partial \text{net}_k}{\partial \text{net}_j}$$

$$
\begin{aligned}
&= \sum_{k\in \text{downstream}(j)} -\delta_k \frac{\partial \text{net}_k}{o_j}\frac{\partial o_j}{\partial \text{net}_k} \\
&= \sum_{k\in \text{downstream}(j)} -\delta_k w_{jk} o_j(1-o_j)
\end{aligned}
$$

Using δ_j to denote $-\frac{\partial E_d}{\partial \text{net}_j}$, we have

$$\delta_j = o_j(1-o_j)\sum_{k\in \text{downstream}(j)} \delta_k w_{jk}$$

and hence the learning rule of weights of hidden units can be written as

$$\Delta w_{ij} = -R\frac{\partial E_d}{\partial w_{ij}} = -R\frac{\partial E_d}{\partial \text{net}_j}x_{ij} = R\delta_j x_{ij}$$

Basically, what we have is an incrementing algorithm. We started by determining the error for the output units. Then, backpropagate this error layer by layer through the network, changing weights appropriately in each layer.

3.2 The Backpropagation Learning Rule

Now let's summarize what we have done. It is described for three layers, but exactly the same is going to work for k layers.

First, create a fully connected three layer network and initialize weights. Then, go example by example, until all examples produce the correct output within ϵ (or other criteria).

```
For each example in the training set
    Compute the network output for this example $o_k$
    Compute the error between the output and the target value
        $\delta_k = (t_k - o_k)o_k(1-o_k)$
    For each output unit $k$, compute error term
        $\delta_j = o_j(1-o_j)\sum_{k\in \text{downstream}(j)} \delta_k w_{jk}$
    For each hidden unit, compute error term
        $\Delta w_{ij} = R\delta_j x_{ij}$
    Update network weights $w_{ij}$
        $w_{ij} \leftarrow w_{ij} + \Delta w_{ij}$
```

The algorithm described above updates weights incrementally, one example at a time, just like stochastic gradient descent. In principle, you may want to do it in batches, but that would complicate the derivations. Conceptually, they are the same algorithm.
The same algorithm holds for more hidden layers. Once we describe the second layer, exactly the same thing will work with more layers.

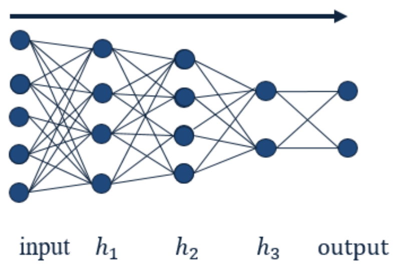


Figure 7: More hidden layers

4 Training

It is important to remember that there is no guarantee of convergence: the algorithm may oscillate or reach a local minima. In practice, many large networks can be trained on large amounts of data requiring many hours of computation time.

As in all gradient algorithms driven algorithms, and important question is termination criteria: number of epochs, threshold on training set error, no decrease in error, increased error on a validation set, etc.

To avoid local minima, one useful technique is to use several trials with different random initial weights with majority or voting.

4.1 Over-training and over-fitting

Running too many epochs may over-train the network and result in over-fitting (improved result on training, decrease in performance on test set).

We have talked about some standard techniques to avoid over-training, and some of them you have experimented with.

Keeping a hold-out validation set and test accuracy after every epoch is going to work. You can also maintain weights for best performing network on the validation set and return it when performance decreases significantly beyond that. To avoid losing training data to validation, use k-fold cross-validation to determine the average number of epochs that optimizes validation performance and train on the full data set using this many epochs to produce the final results.

4.2 Network Architecture

Apart from parameters and tuning methodologies, there is also question on what should be the architecture of the network: how many hidden layers and in what arrangement.

Since it was known that a single hidden layer is enough to approximate any function, it used to be the case that few hidden layers were used. However, using too few hidden units might prevent the system from adequately fitting the data and learning the concept, while using too many hidden units leads to over-fitting.

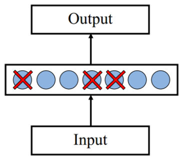
Various modern systems train very deep networks, which is not a simple issue because the gradients are going to be smaller and smaller as you go down. This vanishing gradient problem is difficult to deal with.

As with the number of layers, there is no theory behind the size of the layers themselves, and the cross-validation method is one way to approximate number of hidden units in a layer.

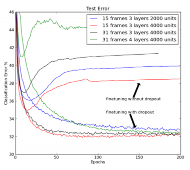
Another approach to prevent over-fitting is weight-decay: all weights are multiplied by some fraction in (0,1) after every epoch. In this way, smaller weights and less complex hypothesis are encouraged. Equivalently, we can change the error function to include a term for the sum of the squares of the weights in the network. These are general techniques that can be applied to other algorithms.

4.3 Dropout Training

Previously, we've discussed that having weights of value zero simplifies the learned hypothesis function, which should reduce overfitting. *Dropout training* simulates the notion of zero weights by eliminating some of the hidden units while training. In dropout training, each time an example is read, some hidden units are removed with probability p, and the network is trained and propagates error as if those weights were not there.



(a) Dropout Training



(b) Dropout of 50% of the hidden units and 20% of the input units

Experiments showed that if dropout scheme is used, a more robust result can be obtained.

Though dropout training was introduced in the context of neural networks, it can be applies to all learning algorithms; rather than changing the architecture of the network, dropout can be thought of as a change in the input.

Given a set of examples, a learning algorithm will determine which features are important. The weights for those important features will then be large and thus dominate the prediction. In the scheme of dropout learning, some features are randomly dropped as examples are read, forcing the learning algorithm to attend to all features.

In practice, it turns out that this idea is effective and often much stronger than other known regularizers.

5 Inputs, Outputs, and Hidden Layers

5.1 Input / Output Coding

What should the inputs and outputs to the network be?

Typically, each output of the network can be thought of independently as a real or binary value.

Where inputs are concerned, people prefer binary values over categorical, since it is difficult to encode categorical relations $(5 > 4)$ in a way the network can understand. In the past, it was common to binarize inputs, but these quickly became very high dimensional. In NLP, for example, inputs could have a million dimensions and cause backpropagation to be extremely slow.

It is currently common practice to encode inputs in relatively few units, where these encodings are produced by a different process. For example, though words can be encoded as sparse, high-dimensional bit-vectors, it is common to use dense, real-valued representations as input to a network.

Sparse representations can be used with neural networks, but only in conjunction with dimensionality reduction methods.

Assume m examples, each with a million features ($n = 10^6$; input matrix is $m \times n$). It is not possible to run backpropogation on this input, but by multiplying the input by a normally distributed random matrix of size $n \times 300$, you produce a dense $m \times 300$ matrix: that is, each example has gone from a sparse 10^6 element vector to a sense 300 element vector.

One of the reasons that it works is that what you had was a very sparse million-dimension vector with just a few 1s. These 1s choose which column in the random matrix that are summed up. Although some meaning is lost along the way, similarities between any two vectors are maintained.

5.2 Hidden Layer Representation

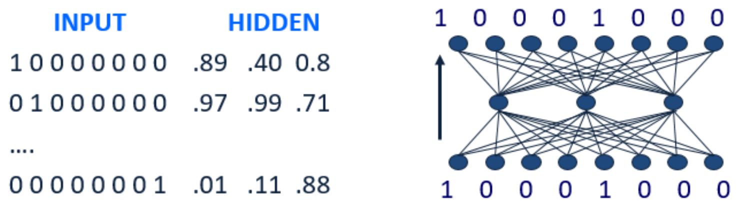
In essence, backpropagation (until the last layer) is a form of learning features over inputs, and the last layer is just a linear learning algorithm over these new features.

Sometimes backpropagation will define new hidden layer features that are not explicit in the input representation, but which capture properties of the input instances that are most relevant to learning the target function. Since the last layer is a linear function over these features, trained hidden units can be seen as newly constructed features that re-represent the examples so that they are linearly separable.

5.3 Auto-Associative Networks

Backpropogation can be used to generate this hidden layer feature representation, as is the case in *auto-associative networks*, where the output must reproduce the input, and the item of interest is the hidden layer between them.

For example, assume numbers 1 to 8 are encoded as 8-bit vectors with only one bit on (eg., 2 encoded as 01000000), and a hidden layer is three nodes. If the



INPUT	HIDDEN
1 0 0 0 0 0 0 0	.89 .40 0.8
0 1 0 0 0 0 0 0	.97 .99 .71
....	
0 0 0 0 0 0 0 1	.01 .11 .88

Figure 9: An auto-associative network with 8 inputs, 3 hidden units and 8 outputs

network is trained in this way, the hidden layer becomes a binary encoding of eight numbers; learning, here, is a compression mechanism.

Stated more generally, given examples $\mathbf{x}$, an auto-associative network learns to produce $\mathbf{x}$ as output, where a hidden layer is of lower dimensionalty. The learned representation is thus more compact, and can be used to chain auto-associative networks, as shown in Figure 10. In such a network, the reconstruction layer is dropped after optimization and a new layer is added. These kinds of tricks have been found to be useful for computer vision, and more generally people have found that the final layer of a network can be transferred; using the final layer of one network can be helpful for slightly different problems. This is something interesting that has not been completely explored yet.

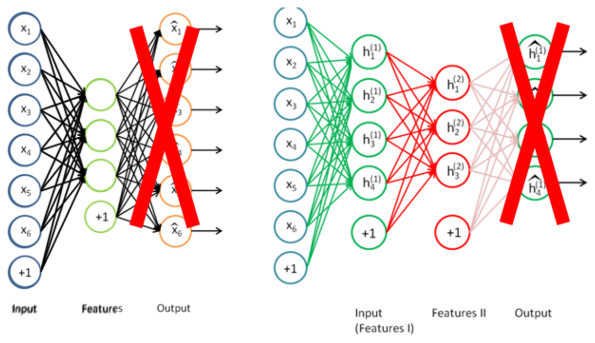


Figure 10: Stacking Auto-encoder

6 Receptive Fields

Consider the problem of encoding input into mathematical models. Humans have sensory elements – eyes and ears – to encode information from the environment. Neural networks also require eyes and ears – things to encode information – in order to process images and sentences. This is a big challenge and there are different ways to handle this, for different tasks and different types of data. However, no ideal, one-size-fits-all solution exists.

In neural network jargon, the input connections can be referred to as *receptive fields*. This term is borrowed from biology, and refers to the individual sensory neuron for which a stimulus will trigger the neuron to fire. 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.

Consider using a neural network to predict whether an image contains a face. Receptive fields should provide expressive features from the raw image data, converting the image to inputs that the neural network can use. One approach

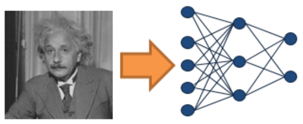
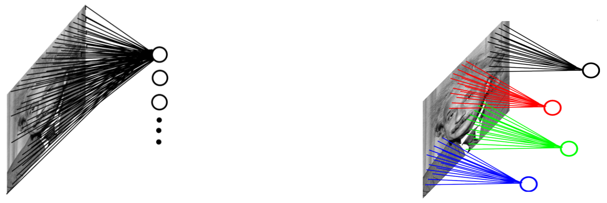


Figure 11: Task with image inputs

to do so would be to design 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 (spatial correlations are lost).



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

Rather than giving all the image data 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 12b 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, this approach is also not ideal.

7 Convolutional layers

These days, people commonly create filters to capture different patterns in the input space. For images, these filters are matrices. Each of the filters scans

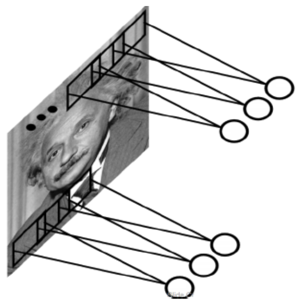


Figure 13: Convolutional layer

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. Using the filters, not only the spatial correlations are preserved, desired properties of the image can also be obtained. This idea can be generalized to other problems – such as text – but this idea also lies at the heart of convolutional neural networks.

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

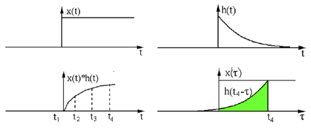


Figure 14: An example of convolution

the functions is flipped. In two dimensions, the idea is the same; flip one matrix and slide it on the other matrix. In the example below, the image is convolved

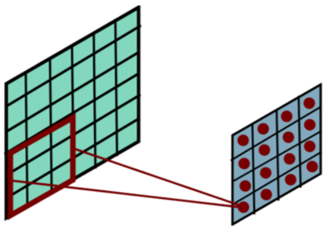


Figure 15: Convolution in 2D

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 in another image. Keep doing this for all blocks of size 3-by-3 over the whole image. 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, the complexity of the convolution operator is $n \log n$. For two-dimensions, each convolution takes $MN \log MN$ time, where the size of input is MN.

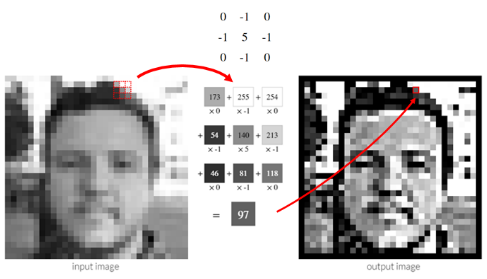


Figure 16: Example: sharpen kernel

7.2 Convolution and Pooling

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

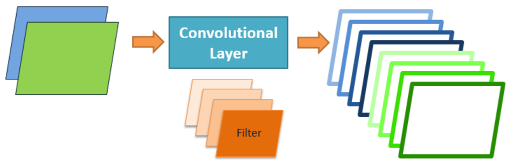


Figure 17: Convolutional layer

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

Now the sizes of the outputs depend on the sizes of the inputs. People in the community are actually using something very simple called a *pooling layer*, which is a layer that reduces input of different sizes to a fixed size. There are

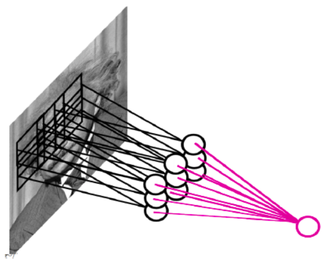


Figure 18: Pooling

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.

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

7.3 Convolutional Neural Networks

Combined, the convolution and pooling operations are said to constitute a single convolutional stack.

1. Convolve an input with a filter $\rightarrow$ produce outputs of variable sizes
2. Use pooling to shrink outputs to a single, desired size

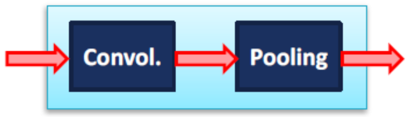


Figure 19: One-stage convolutional net

We can then combine these stacks as often 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.

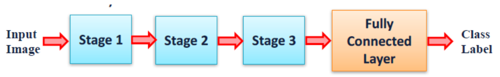


Figure 20: Convolutional Neural Network

7.4 Training a ConvNet

Remember in backpropagation we started from the error terms in the last layer, and passed them back to the previous layers, one by one. The same procedure from backpropagation applies here.

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

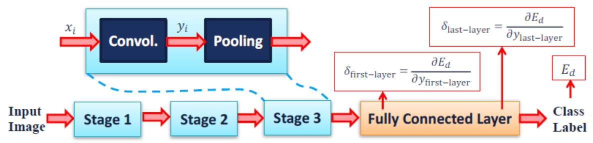

x_i
Convol.
y_i
Pooling
$\delta_{\text{last-layer}} = \frac{\partial E_d}{\partial y_{\text{last-layer}}}$
$\delta_{\text{first-layer}} = \frac{\partial E_d}{\partial y_{\text{first-layer}}}$
E_d
Input Image
Stage 1
Stage 2
Stage 3
Fully Connected Layer
Class Label

Figure 21: Backpropagation for ConvNets

detailed in the lecture slides.

7.5 Example of ConvNets

To get more intuition about ConvNets, let's look at the following example of identifying whether there is a car in an image. 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? We are training

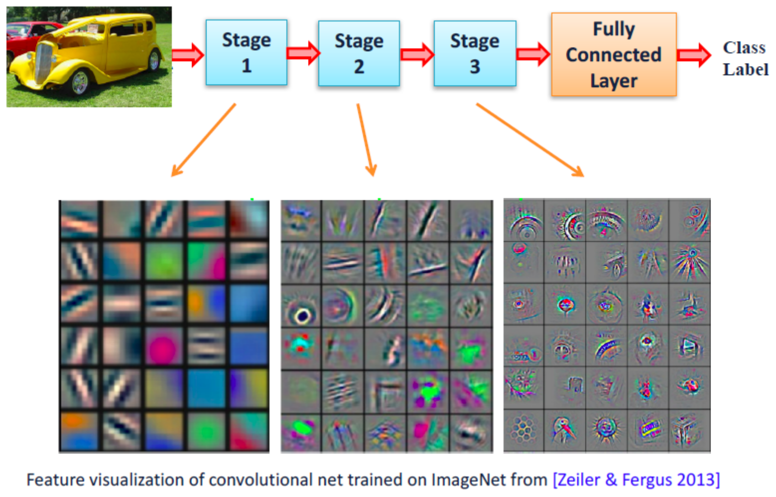

Stage 1
Stage 2
Stage 3
Fully Connected Layer
Class Label
Feature visualization of convolutional net trained on ImageNet from [Zeiler & Fergus 2013]

Figure 22: Example of identifying a car with ConvNet

the filters. Our task is to identify whether a car is in the image, but at each stage, there are multiple 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.

7.6 History

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

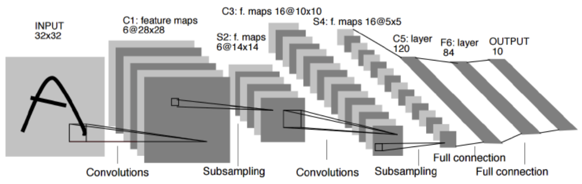


Figure 23: Example system: LeNet

read zip codes, digits, etc.

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

8 Neural Network Depth

It used to be the case that people preferred small, shallow networks, since it was known that even a single hidden layer could approximate any function. Recently, however, it's common to use deeper networks. Consider Figure 24; the error decreases as depth increases in recent years. Though deeper networks are

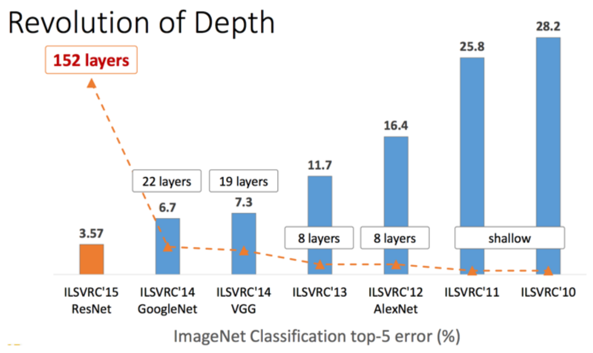


Figure 24: Revolution of Depth

generally more accurate, it is not clear theoretically why this is the case.

9 Recurrent Neural Networks

In the feed-forward neural network architecture, there are no cycles, because error must be propagated backwards. In principle RNNs have cycles, but in practice these cycles are broken. An RNN is a digraph that has cycles, which

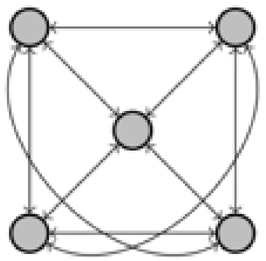


Figure 25: RNN is a digraph

can act as memory; the hidden states can carry information about a potentially unbounded number of previous inputs. In practice we essentially we break the cycles in an RNN by unwrapping it across time.

Consider the cyclic representation in Figure 26. Assume that there is a time

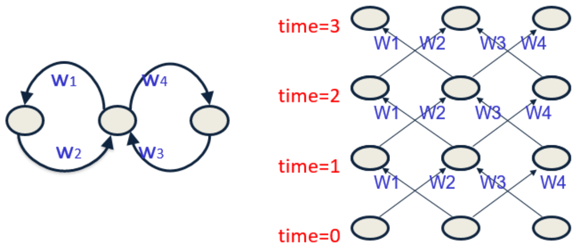
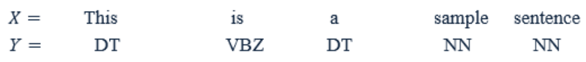


Figure 26: Cyclic representation of a neural network

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.

9.1 An NLP Example

Training a general RNNs can be hard. Here we will focus on a special family of RNNs that predict on chain-like input. Consider the task of Part-of-Speech (POS) tagging Given a sentence of words, the RNN should output a POS tag for each of the word in the sentence.



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

Figure 27: POS tagging words in a 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.

9.2 Chain RNN

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

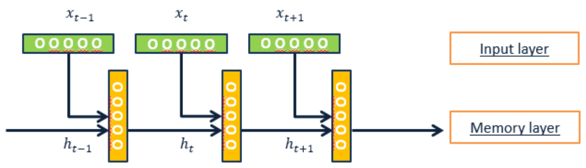


Figure 28: An RNN with a chain-like structure

As shown in Figure 28, the x_t's are 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 the current word. Each input is combined with a current hidden state, and another hidden state is produced. 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 step t, which in this example refers to the words in the sentence.

The structure shown in the above figure is the same structure being applied multiple times (three in the figure). 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.

9.3 Bi-Directional RNNs

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 Figure 29 Using the same idea, the model can be made further

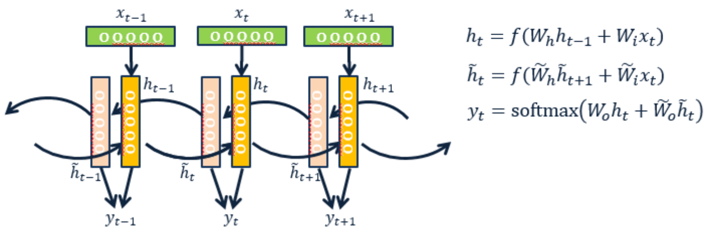

x_{t-1}
x_t
x_{t+1}
h_{t-1}
h_t
h_{t+1}
$\tilde{h}_{t-1}$
$\tilde{h}_t$
$\tilde{h}_{t+1}$
y_{t-1}
y_t
y_{t+1}
$h_t = f(W_h h_{t-1} + W_i x_t)$
$\tilde{h}_t = f(\widetilde{W}_h \tilde{h}_{t+1} + \widetilde{W}_i x_t)$
$y_t = \text{softmax}(W_o h_t + \widetilde{W}_o \tilde{h}_t)$

Figure 29: An RNN with a bi-directional structure

complicated, like the stack of bi-directional networks shown in the below figure.

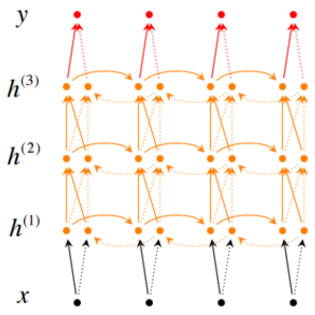


Figure 30: stack of bi-directional networks

9.4 Training

In the POS tagging task, each word is represented as a vector of fixed size. Consider initializing each word with a random weight. Now, the representation for each word is a set of parameters we must train. These input representations are then multiplied by a matrix to get the hidden state from the previous state, which is multiplied by another matrix to get the next hidden state. This process is how we transfer from one hidden state to the next; the matrices involved are more parameters that must be trained.

Given these hidden states, we multiply them with a matrix, apply the softmax function, and produce a distribution over the output labels. This final matrix is

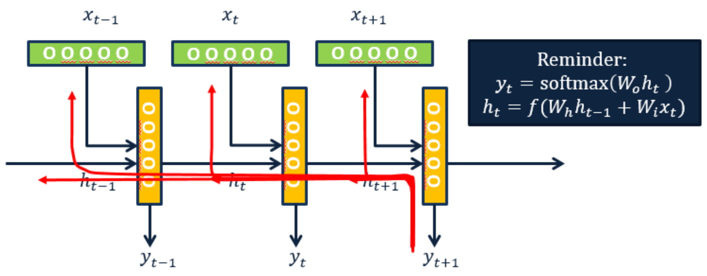


Figure 31: Training an RNN

also another set of parameters that must be learned. 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 (words 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.

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

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

9.5 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. 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).