

Deep Nets

Deep learning (with deep networks) simply mean that we have a neural network with lots of layers. With an abundance of layers comes an abundance of training parameters, so a couple of extra issues come up:

- Training from scratch becomes harder and harder. In particular, because there are an infinite number of ways to extract “features” from data, we need extra constraints on how we train. Typically, that will come in the form of regularization and sparsity.
- If you think the number of training parameters is large, just wait- We’ll need a LOT of data to be sure that we’ve trained the network properly. For example, the digit recognition problem below is using six thousand images. This makes them harder to deal with on your computer, and brings up issues in database management- that would take us far afield.

The notes below have been summarized from many different places, but mostly from Matlab.

Dealing with a Lot of Data

In these notes, we’ll be discussing ways of building a neural network whose training will require a lot of data. In particular, the convolutional neural network was designed for image processing, so we need a lot (on the order of thousands) of images.

It would be difficult to deal with all this data manually, so Matlab has some special processing commands to deal with it. Matlab uses what they call an “image data store”. That is, if you have a file folder filled with subfolders labeled by type (like **trains**, **planes**, **cats**, and so on), then Matlab will automatically go through all of them, take note of all the files stored, and will automatically take a listing of the categories. Below is a sample of how that’s done.

I’ve assumed that the Matlab tar file has been downloaded from the Hinton group of researchers in Toronto. This is the CIFAR-10 data set which is about 175 MB.

`www.cs.toronto.edu/~kriz/index.html`

Convolutional Neural Nets

CNN is a “convolutional neural network” designed to work with images. It contains several layers and computational nodes to produce an output.

You can train a CNN to do image analysis tasks, like scene classification or object detection and identification.

To understand how CNNs work, we’ll look at three important ideas:

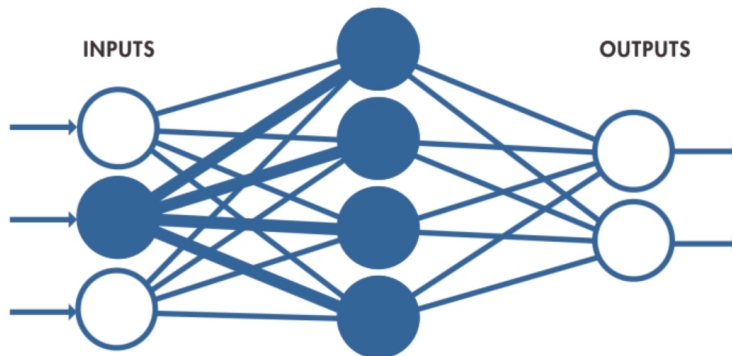
- Local receptive fields.

- Shared weights and biases.
- Activation and Pooling.

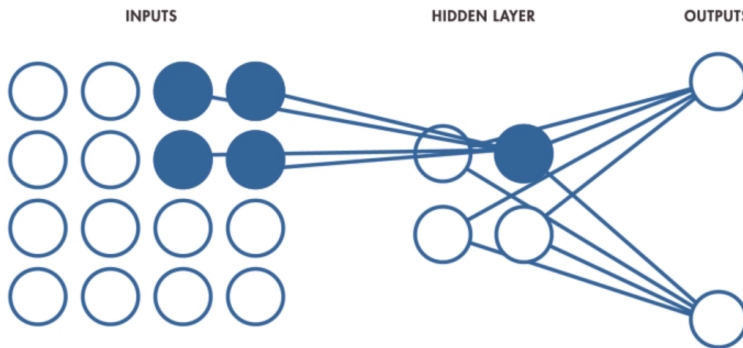
Finally, we'll discuss three ways to train a CNN for image analysis. But first, local receptive fields.

Local Receptive Fields

In a standard neural network, each input neuron is fully connected to the hidden layer, as shown below for the middle neuron:



In a convolution neural net, only a small part of the image is connected forward, as shown below. The sample image consists of four subimages (each 4×4). Each of the four sub-blocks is then connected to its own neuron in the hidden layer.

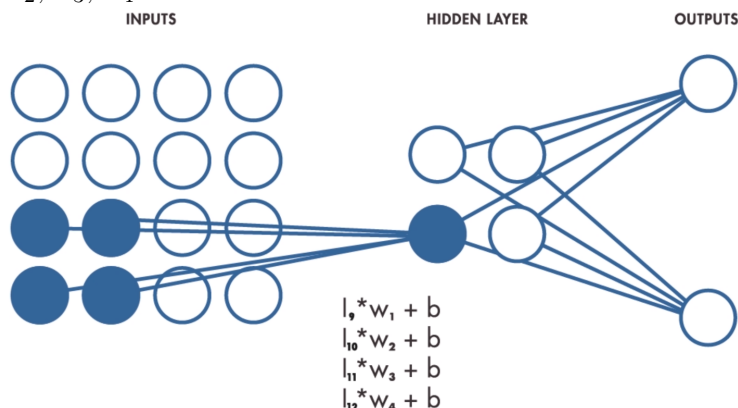


Why does this work well? The main reason is so that objects become translation invariant- If we're looking for a cat in the image, for example, we would need to scan the image to see where an object might be before we try to classify that object. In terms of vocabulary, we might say that these sub-images form the *local receptive fields* and the mapping from the inputs to the hidden layer is called a *feature mapping*. Convolution is used to implement the process efficiently.

Shared Weights and Biases

We note that in a standard neural network, there are different weights for each input. In a convolutional neural network, the weights (and bias) are identical. The image below illus-

trates this point- No matter which 4×4 block we are currently working on, the weights are w_1, w_2, w_3, w_4 and bias b .

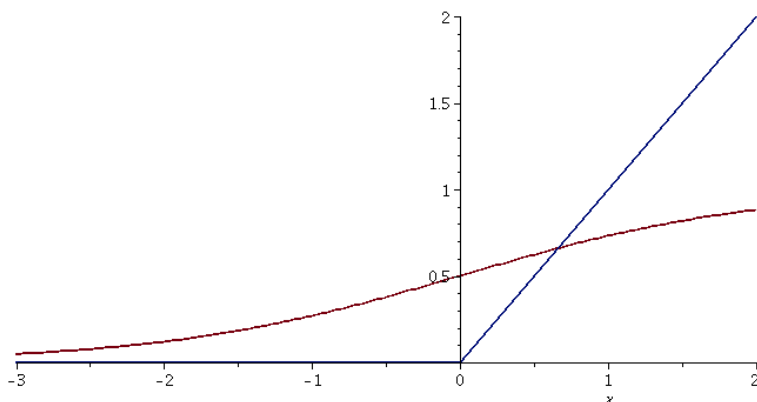


Activation and Pooling

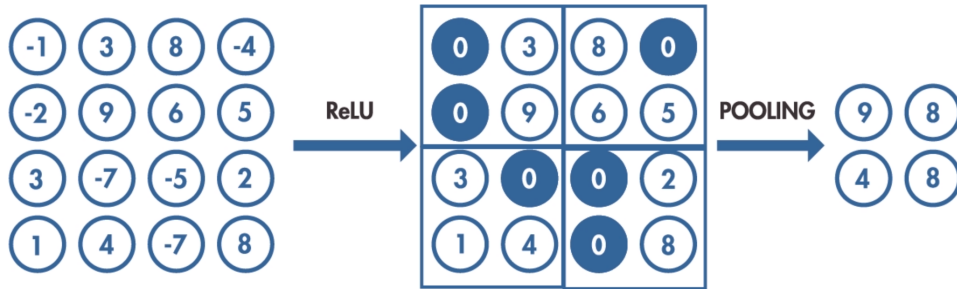
In a standard neural network, we use the sigmoidal function for the transfer function (or activation function), $\sigma(r)$, and this is typically $\sigma(r) = \frac{1}{1 + e^{-r}}$. It was found that we don't necessarily require smooth functions, and researchers have found some good results using the so-called "ReLU", or "Rectified Linear Function":

$$\text{ReLU}(r) = \begin{cases} 0 & \text{if } r < 0 \\ r & \text{if } r \geq 0 \end{cases}$$

Plotting the two, we can see the similarities and differences. There are some challenges in using this transfer function that we'll discuss later.



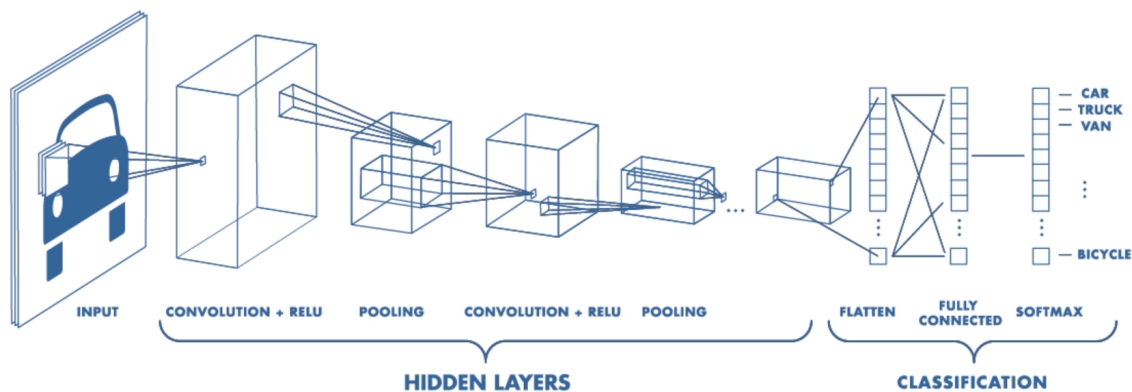
Following the activation, it is common to perform what's called "pooling", where we take a block of values and replace it by a single value. The size of the block can be specified, and how to compute the value can be specified. In Matlab, the pooling can be done by either taking the maximum value of the block, as shown below with a 4×4 block, or it can be the average of the values.



As we see, the main consequence of pooling is in dimensionality reduction.

Putting the layers together

We can put the layers together to create the full convolutional neural network- Here's one example:



Once the Network has been defined...

There are three basic ways of training and working with the CNN.

- Train from Scratch (most challenging)

This is the most challenging. You may need hundreds of thousands of labeled images for training. The training can also take a very long time- The example below took a modern PC about 6 hours to train.

- Transfer learning.

In transfer learning, you begin with a network that has been trained (presumably for related tasks). For example, we might download **AlexNet** from the internet, which has already been pretrained with thousands of images. We might then focus training on "Trucks versus Cars" from that. This would then just be fine tuning your parameters instead of starting from scratch.

Use a pre-trained CNN. For a new task, fine-tune the network weights.

- Feature Extraction.

This may be the most interesting part. We'll show below that it is possible to break training up into pieces, where each piece is trained to try to do general feature extraction. Afterward, we'll put the pieces together and "fine tune" the weights using the full problem. (See the section on "Autoencoders and Deep Learning").

A Basic Training Session

This example assumes we have the CIFAR-10 data set downloaded. See me if you need help with that- Matlab has some scripts that make it easy, but including them now would lead us too far astray. For now, let's go ahead and look at a sample training session that builds the convolution neural network from scratch and then trains it.

First, we define where the data set is using an "image data store"- this is a new feature in Matlab that helps us manage a large database of images.

```
categories = {'deer','dog','frog','cat'};

rootFolder = 'cifar10Train';
imds = imageDatastore(fullfile(rootFolder, categories), ...
    'LabelSource', 'foldernames');
```

Unlike a feed forward network, we'll define each layer separately for the convolution network, then put them together. The images here are $32 \times 32 \times 3$ (color).

```
%% Set up the convolution neural network

varSize = 32;
conv1 = convolution2dLayer(5,varSize,'Padding',2,'BiasLearnRateFactor',2);
conv1.Weights = single(randn([5 5 3 varSize])*0.0001);
```

In this case we have a two dimensional convolution with a $5 \times 5 \times 3$ filter (the filter is 5×5 , but since there are three frames of color, the filter also has a depth of 3.) The weights of the filter are initialized randomly.

Next, we'll define a fully connected layer and initialize the weights.

```
fc1 = fullyConnectedLayer(64,'BiasLearnRateFactor',2);
fc1.Weights = single(randn([64 576])*0.1);
```

And now a second connected layer, and initialize the weights.

```
fc2 = fullyConnectedLayer(4,'BiasLearnRateFactor',2);
fc2.Weights = single(randn([4 64])*0.1);
```

Now is the key command: We define the layers of the neural network below. You can read what they are- First, the image is input, followed by a convolution layer, followed by a pooling layer, followed by the transfer function layer. That's followed by a second convolution layer, followed by a transfer function, followed by pooling, followed by another convolution layer, etc. Quite a deep net!

```
layers = [
    imageInputLayer([varSize varSize 3]);
    conv1;
    maxPooling2dLayer(3,'Stride',2);
    reluLayer();
    convolution2dLayer(5,32,'Padding',2,'BiasLearnRateFactor',2);
    reluLayer();
    averagePooling2dLayer(3,'Stride',2);
    convolution2dLayer(5,64,'Padding',2,'BiasLearnRateFactor',2);
    reluLayer();
    averagePooling2dLayer(3,'Stride',2);
    fc1;
    reluLayer();
    fc2;
    softmaxLayer()
    classificationLayer()];

opts = trainingOptions('sgdm', ...
    'InitialLearnRate', 0.001, ...
    'LearnRateSchedule', 'piecewise', ...
    'LearnRateDropFactor', 0.1, ...
    'LearnRateDropPeriod', 8, ...
    'L2Regularization', 0.004, ...
    'MaxEpochs', 10, ...
    'MiniBatchSize', 100, ...
    'Verbose', true);

%% Actual training:

tic
[net, info] = trainNetwork(imds, layers, opts);
toc

rootFolder = 'cifar10Test';
imds_test = imageDatastore(fullfile(rootFolder, categories), ...
    'LabelSource', 'foldernames');
```

```

%% Sample test
labels = classify(net, imds_test);

ii = randi(4000);
im = imread(imds_test.Files{ii});
imshow(im);
if labels(ii) == imds_test.Labels(ii)
    colorText = 'g';
else
    colorText = 'r';
end
title(char(labels(ii)), 'Color', colorText);

%% Error
% This could take a while if you are not using a GPU
confMat = confusionmat(imds_test.Labels, labels);
confMat = confMat./sum(confMat,2);
mean(diag(confMat))

```

Transfer Learning

In this example, we download the pre-trained AlexNet to classify the CIFAR-10 data. Here's the code with a few comments.

There are a few things you need to do if you haven't downloaded AlexNet before, but you only need to do that once, then the following command will work. Go ahead and try it first:

```
net=alexnet;
```

To see what the layers look like, type the following. You can leave the semicolon off of the end and the layers will print out. We'll note that we need to change a few parameters.

```
layers = net.Layers
```

```
rootFolder = 'cifar10Train';
```

```
categories = {'Deer','Dog','Frog','Cat'};
```

```

imds = imageDatastore(fullfile(rootFolder, categories), 'LabelSource', 'foldernames');
imds = splitEachLabel(imds, 500, 'randomize') % we only need 500 images per class
imds.ReadFcn = @readFunctionTrain;

```

```
% We'll take out a few layers, add a few layers:
```

```

layers = layers(1:end-3);
layers(end+1) = fullyConnectedLayer(64, 'Name', 'special_2');
layers(end+1) = reluLayer;
layers(end+1) = fullyConnectedLayer(4, 'Name', 'fc8_2 ');
layers(end+1) = softmaxLayer;
layers(end+1) = classificationLayer()

% We'll add in some fine-tuning parameters (this takes some experience to know)
layers(end-2).WeightLearnRateFactor = 10;
layers(end-2).WeightL2Factor = 1;
layers(end-2).BiasLearnRateFactor = 20;
layers(end-2).BiasL2Factor = 0;

opts = trainingOptions('sgdm', ...
    'LearnRateSchedule', 'none',...
    'InitialLearnRate', .0001,...
    'MaxEpochs', 20, ...
    'MiniBatchSize', 128);

% Train it!
convnet = trainNetwork(imds, layers, opts);

% Test it! First, define where the test data is:

rootFolder = 'cifar10Test';
testDS = imageDatastore(fullfile(rootFolder, categories), 'LabelSource', 'foldernames');
testDS.ReadFcn = @readFunctionTrain;

% Now test it:
[labels,err_test] = classify(convnet, testDS, 'MiniBatchSize', 64);

% confusion:
confMat = confusionmat(testDS.Labels, labels);
confMat = confMat./sum(confMat,2);
mean(diag(confMat))

```

Autoencoders for Deep Learning

The basic idea behind using autoencoders for deep learning, is that we can use regularization techniques to encode the raw input data into “features” in a much lower dimensional space that will make it easier to train a deep network.

First we'll take the raw data down to some intermediate dimensions- In the example

below, we'll take 784 dimensions down to 100 first, then the second network will take 100 dimensions down to 50. We then train the final network with 50 dimensional input (and 10 dimensional output, since we'll have 10 classes).

The nice thing about this process is that we can break up the feature encoding into smaller pieces, then put the trained networks together for what will hopefully then be an easy network to train.

In this example, we classify the handwritten digits 0-9 (the MNIST data set), where each digit is a 28×28 array representing a small image.

First we'll load the data using Matlab's built-in commands for this task.

(Warning: These commands might take approximately 20 minutes or so to complete.)

```
[XTrainImages,tTrain]=digitTrainCellArrayData;
```

The output, `xTrainImages`, is a cell array where each cell is a 28×28 gray scale image. To see the 20th image, we would type:

```
imshow(xTrainImages{20})
```

The input layer is $28 \times 28 = 784$, so we'll first bring the number of nodes to 100. Here is the full command:

```
autoenc1=trainAutoencoder(xTrainImages, 100, ...  
'MaxEpochs',400,...  
'L2WeightRegularization',0.0004,...  
'SparsityRegularization',4,...  
SparsityProportion,0.15,...  
'ScaleData',false);
```

```
figure()  
plotWeights(autoenc1);
```

```
feat1 = encode(autoenc1,xTrainImages);
```

Notice that `feat1` represents the data at the hidden layer (100 dimensional), which we now use as training input to the next autoencoder!

```
hiddenSize2 = 50;  
autoenc2 = trainAutoencoder(feat1,hiddenSize2, ...  
    'MaxEpochs',100, ...  
    'L2WeightRegularization',0.002, ...  
    'SparsityRegularization',4, ...  
    'SparsityProportion',0.1, ...  
    'ScaleData', false);
```

```
feat2 = encode(autoenc2,feat1);
```

Once again, `feat2` represents the data at the new hidden layer, this time in 50 dimensional space. This will be used as input to train our last layer- a “softmax” layer that will actually use the target information as well.

```
softnet = trainSoftmaxLayer(feat2,tTrain,'MaxEpochs',400);
```

Now the three networks have been trained. Matlab has a nice feature called “stack” which will automatically stack the networks correctly, then we’ll view the result to be sure everything looks OK.

```
stackednet = stack(autoenc1,autoenc2,softnet);  
view(stackednet)
```

Now we’ll load some test data and check the confusion matrix. We’ll need to change the format of our input data slightly. Recall that the first network could use the cell structure, but in the new net, we’ll use a matrix.

```
% Get the number of pixels in each image  
imageWidth = 28;  
imageHeight = 28;  
inputSize = imageWidth*imageHeight;  
  
% Load the test images  
[xTestImages,tTest] = digitTestCellArrayData;  
  
% Turn the test images into vectors and put them in a matrix  
xTest = zeros(inputSize,numel(xTestImages));  
for i = 1:numel(xTestImages)  
    xTest(:,i) = xTestImages{i}(:);  
end  
  
y = stackednet(xTest);  
plotconfusion(tTest,y);
```

Finally, we can fine-tune the network (which we really should have done before using the sacred test data...)

```
% Turn the training images into vectors and put them in a matrix  
xTrain = zeros(inputSize,numel(xTrainImages));  
for i = 1:numel(xTrainImages)  
    xTrain(:,i) = xTrainImages{i}(:);  
end  
  
% Perform fine tuning  
stackednet = train(stackednet,xTrain,tTrain);
```

```
y = stackednet(xTest);  
plotconfusion(tTest,y);
```

At the end, we get approximately 98% classification accuracy! Good job!