Convolutional Neural Networks

CNNs

- A convolutional neural network (CNN, or ConvNet) is a class of deep, feed-forward artificial neural networks that has successfully been applied to analyzing visual imagery.
- Convolutional networks were inspired by biological processes in which the connectivity pattern between neurons is inspired by the organization of the animal visual cortex. Individual cortical neurons respond to stimuli only in a restricted region of the visual field known as the receptive field. The receptive fields of different neurons partially overlap such that they cover the entire visual field.
- In Figure 1, a ConvNet is able to recognize scenes and the system is able to suggest relevant captions (“a soccer player is kicking a soccer ball”) while Figure 2 shows an example of ConvNets being used for recognizing everyday objects, humans and animals. Lately, ConvNets have been effective in several Natural Language Processing tasks (such as sentence classification) as well.

a soccer player is kicking a soccer ball

a street sign on a pole in front of a building

a couple of giraffe standing next to each other

figure 1

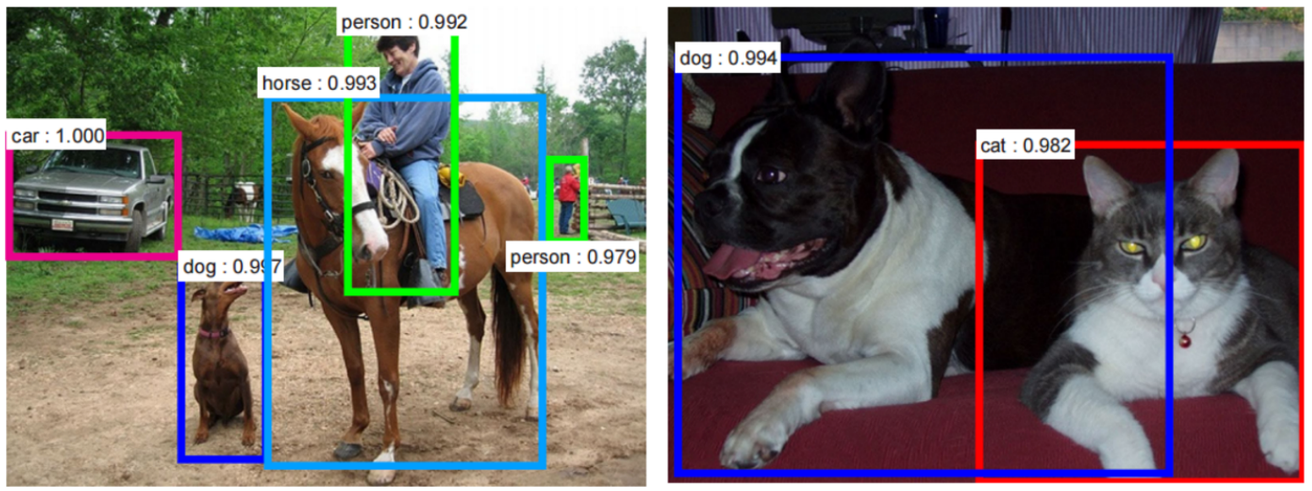
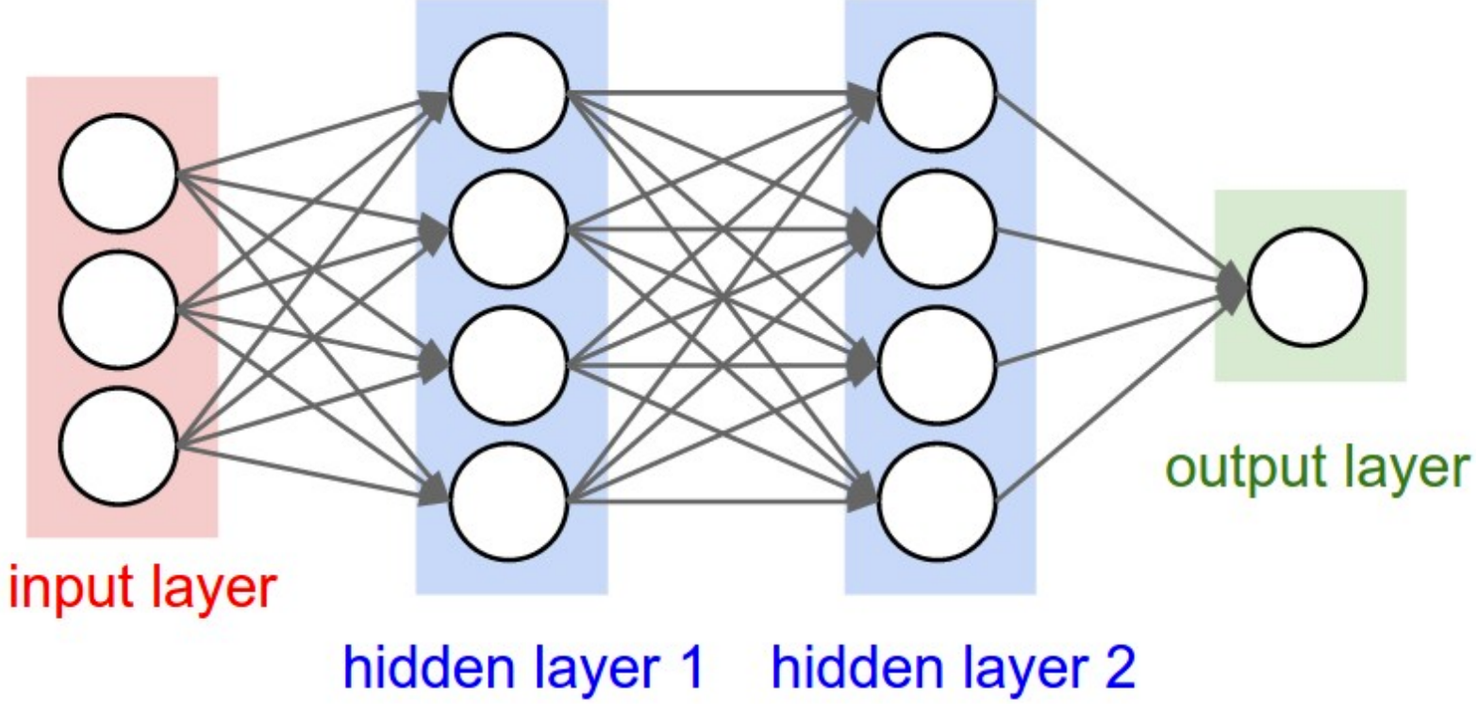


figure 2

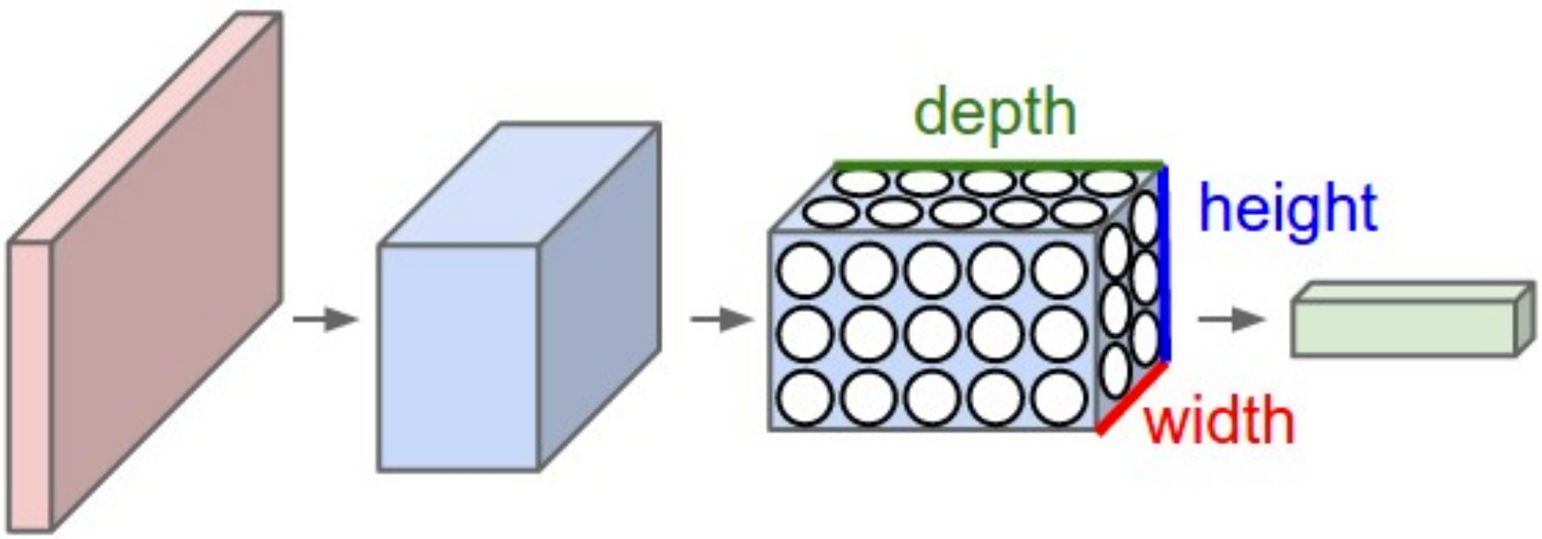
CNNs

- Convolutional Neural Networks are very similar to ordinary Neural Networks: they are made up of neurons that have learnable weights and biases.
- So what does change? ConvNet architectures make the explicit assumption that the inputs are images, which allows us to encode certain properties into the architecture. These then make the forward function more efficient to implement and vastly reduce the amount of parameters in the network.
- Regular Neural Nets: Neural Networks receive an input (a single vector), and transform it through a series of hidden layers. Each hidden layer is made up of a set of neurons, where each neuron is fully connected to all neurons in the previous layer, and where neurons in a single layer function completely independently and do not share any connections. The last fully-connected layer is called the “output layer” and in classification settings it represents the class scores.



CNNs

- Regular Neural Nets don't scale well to full images: images are only of size 32x32x3 (32 wide, 32 high, 3 color channels), so a single fully-connected neuron in a first hidden layer of a regular Neural Network would have 32*32*3 = 3072 weights.But an image of more respectable size, e.g. 200x200x3, would lead to neurons that have 200*200*3 = 120,000 weights. Moreover, we would almost certainly want to have several such neurons, so the parameters would add up quickly! Clearly, this full connectivity is wasteful and the huge number of parameters would quickly lead to overfitting.
- 3D volumes of neurons: Convolutional Neural Networks take advantage of that the input consists of images and the layers of a ConvNet have neurons arranged in 3 dimensions: width, height, depth. For example, the input images are an input volume of activations, and the volume has dimensions 32x32x3 . The neurons in a layer will only be connected to a small region of the layer before it, instead of all of the neurons in a fully-connected manner. Moreover, the final output layer would have dimensions 1x1x10, because the end of the ConvNet architecture reduces the full image into a single vector of class scores, arranged along the depth dimension.

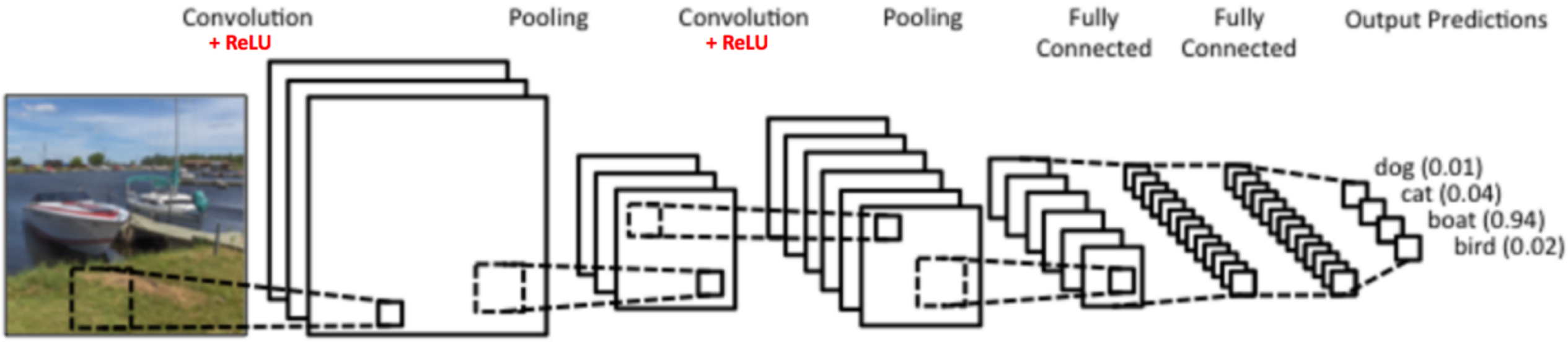


The red input layer holds the image, so its width and height would be the dimensions of the image, and the depth would be 3 (Red, Green, Blue channels).

LeNet Architecture

LeNet was one of the very first convolutional neural networks. This pioneering work by Yann LeCun was named LeNet5 after many previous successful iterations since the year 1988. At that time the LeNet architecture was used mainly for character recognition tasks such as reading zip codes, digits, etc.

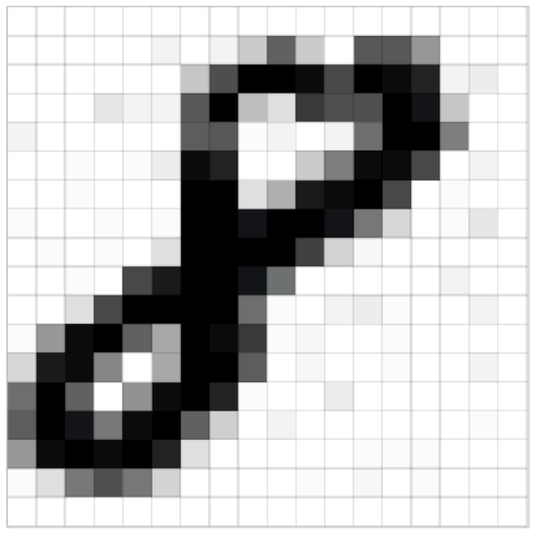
How the LeNet architecture learns to recognize images?



- classifies an input image into four categories: dog, cat, boat or bird
- receive a boat image as input and the network correctly assigns the highest probability for boat (0.94) among all four categories
- There are four main operations in the ConvNet
  - Convolution
  - Non Linearity (ReLU)
  - Pooling or Sub Sampling
  - Classification (Fully Connected Layer)

Input

every image can be represented as a matrix of pixel values.



```
  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   1  12   0  11  39 137  37   0 152 147  84   0   0   0
  0   0   1   0   0   0  41 160 250 255 235 162 255 238 206  11  13   0
  0   0   0  16   9   9 150 251  45  21 184 159 154 255 233  40   0   0
 10   0   0   0   0   0 145 146   3  10   0  11 124 253 255 107   0   0
  0   0   3   0   4  15 236 216   0   0  38 109 247 248 169   0  11   0
  1   0   2   0   0   0 253 253  23  62 224 241 255 164   0   5   0   0
  6   0   0   4   0   3 252 250 228 255 255 234 112  28   0   2  17   0
  0   2   1   4   0  21 255 253 251 255 172  31   8   0   1   0   0   0
  0   0   4   0 163 225 251 255 229 120   0   0   0   0   0  11   0   0
  0   0  21 162 255 255 254 255 126   6   0  10  14   6   0   0   9   0
  3  79 242 255 141  66 255 245 189   7   8   0   0   5   0   0   0   0
 26 221 237  98   0  67 251 255 144   0   8   0   0   7   0   0  11   0
125 255 141   0  87 244 255 208   3   0   0  13   0   1   0   1   0   0
145 248 228 116 235 255 141  34   0  11   0   1   0   0   0   1   3   0
 85 237 253 246 255 210  21   1   0   1   0   0   6   2   4   0   0   0
  6  23 112 157 114  32   0   0   0   0   2   0   8   0   7   0   0   0
  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
```

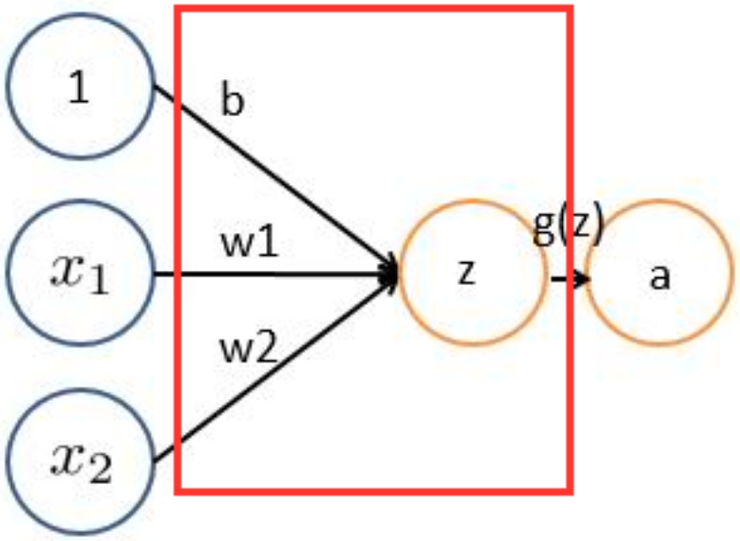
Channel is a conventional term used to refer to a certain component of an image. An image from a standard digital camera will have three channels – red, green and blue – you can imagine those as three 2d-matrices stacked over each other (one for each color), each having pixel values in the range 0 to 255.

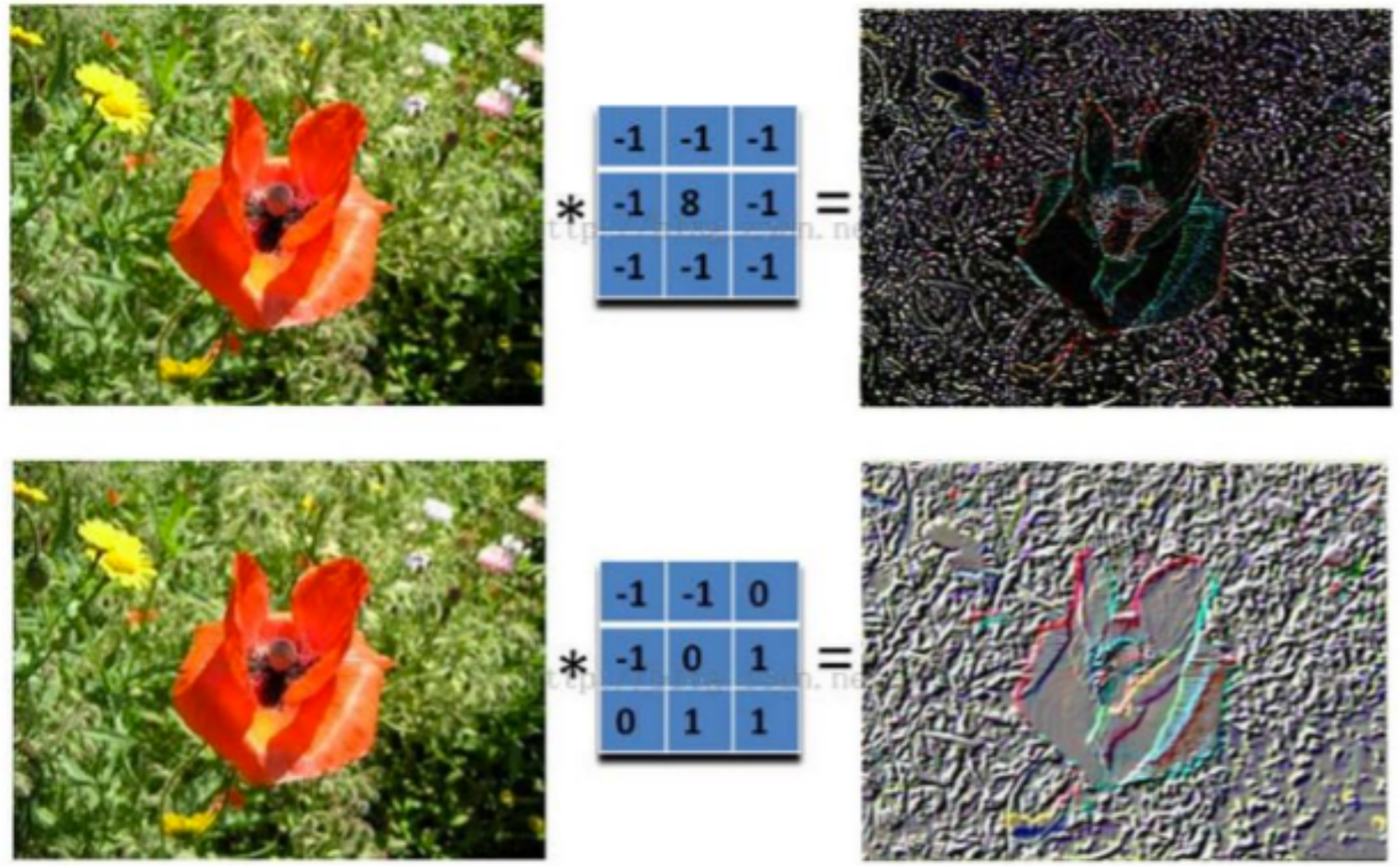
A grayscale image, on the other hand, has just one channel and a single 2d matrix representing an image. The value of each pixel in the matrix will range from 0 to 255 – zero indicating white and 255 indicating black.

For example, INPUT [32x32x3] will hold the raw pixel values of the image, in this case an image of width 32, height 32, and with three color channels R,G,B.

Convolutional layers

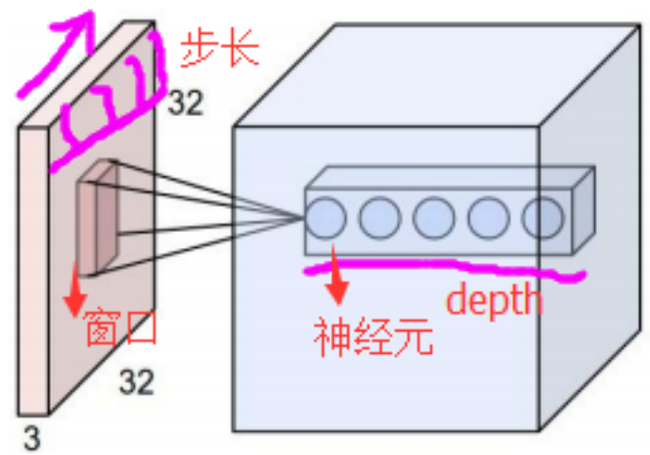
- The primary purpose of a ConvNet is to extract features from the input image. Convolution preserves the spatial relationship between pixels by learning image features using small squares of input data, and it apply a convolution operation to the input, passing the result to the next layer.
- CONV layer will compute the output of neurons that are connected to local regions in the input, each computing a dot product between their weights and a small region they are connected to in the input volume.
- Every neutron acts as a filter for the presence of specific features or patterns present in the original image. The filter is shifted multiple times and applied at different image positions until the entire image has been covered in detail (the filter can correct, if necessary, e.g. for scale, translation, rotation angle, color, transformation, opacity, out of focus, deviations of specific features present in the original image).





input filter (neurons with fixed weights) output

Convolution Operation

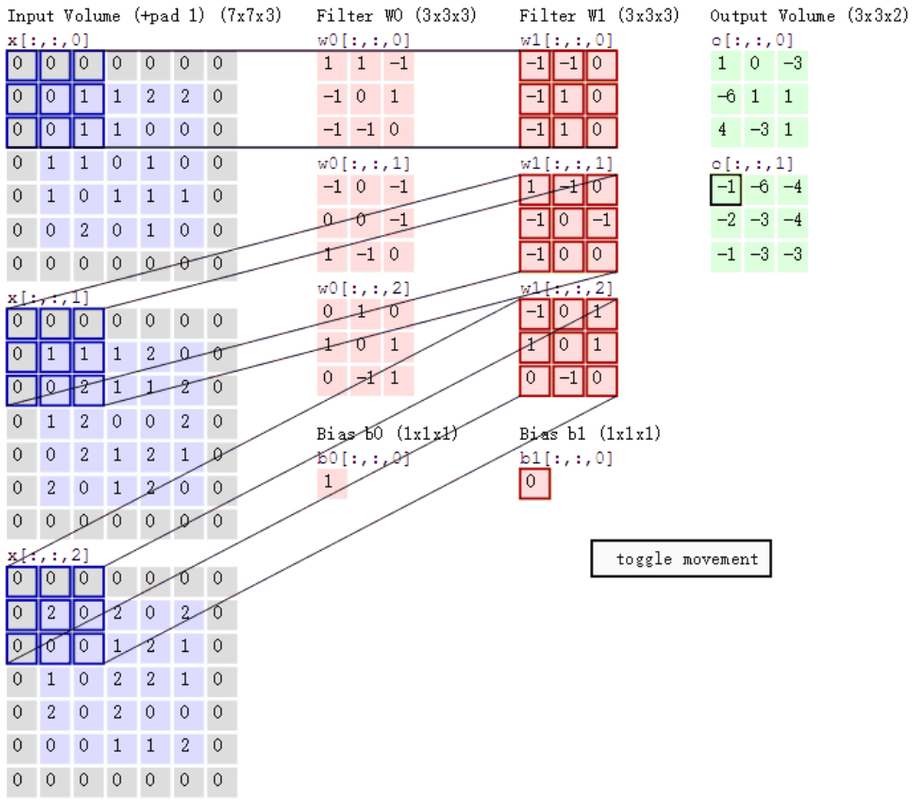


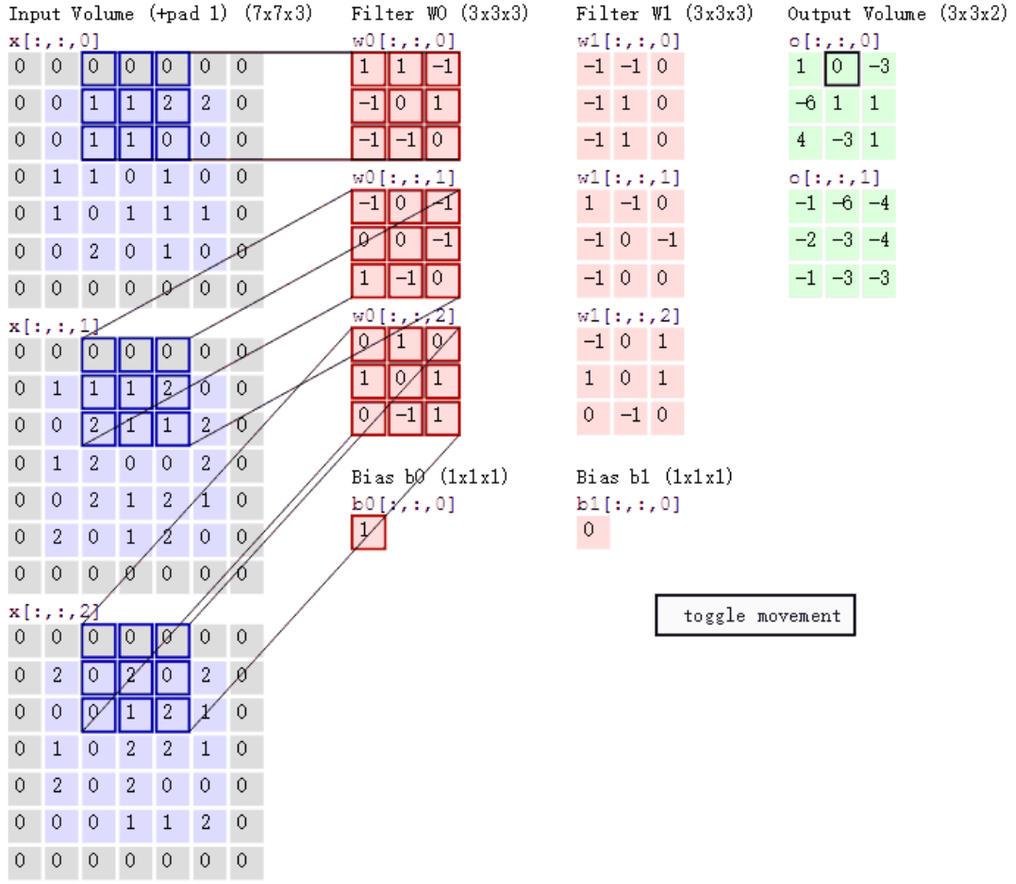
A Filter uses a mall squares of input data for convolution operation (sum(wixi bi)n) and slides over the input image until the entire image has been covered to produce a feature map.

The size of the Feature Map (Convolved Feature) is controlled by three parameters:

- Depth: Depth corresponds to the number of filters we use for the convolution operation.
- Stride: Stride is the number of pixels by which we slide our filter matrix over the input matrix. When the stride is 1 then we move the filters one pixel at a time. When the stride is 2, then the filters jump 2 pixels at a time as we slide them around. Having a larger stride will produce smaller feature maps.
- Zero-padding: Sometimes, it is convenient to pad the input matrix with zeros around the border, so that we can apply the filter to bordering elements of our input image matrix. A nice feature of zero padding is that it allows us to control the size of the feature maps. Adding zero-padding is also called wide convolution, and not using zero-padding would be a narrow convolution.

Convolution Operation

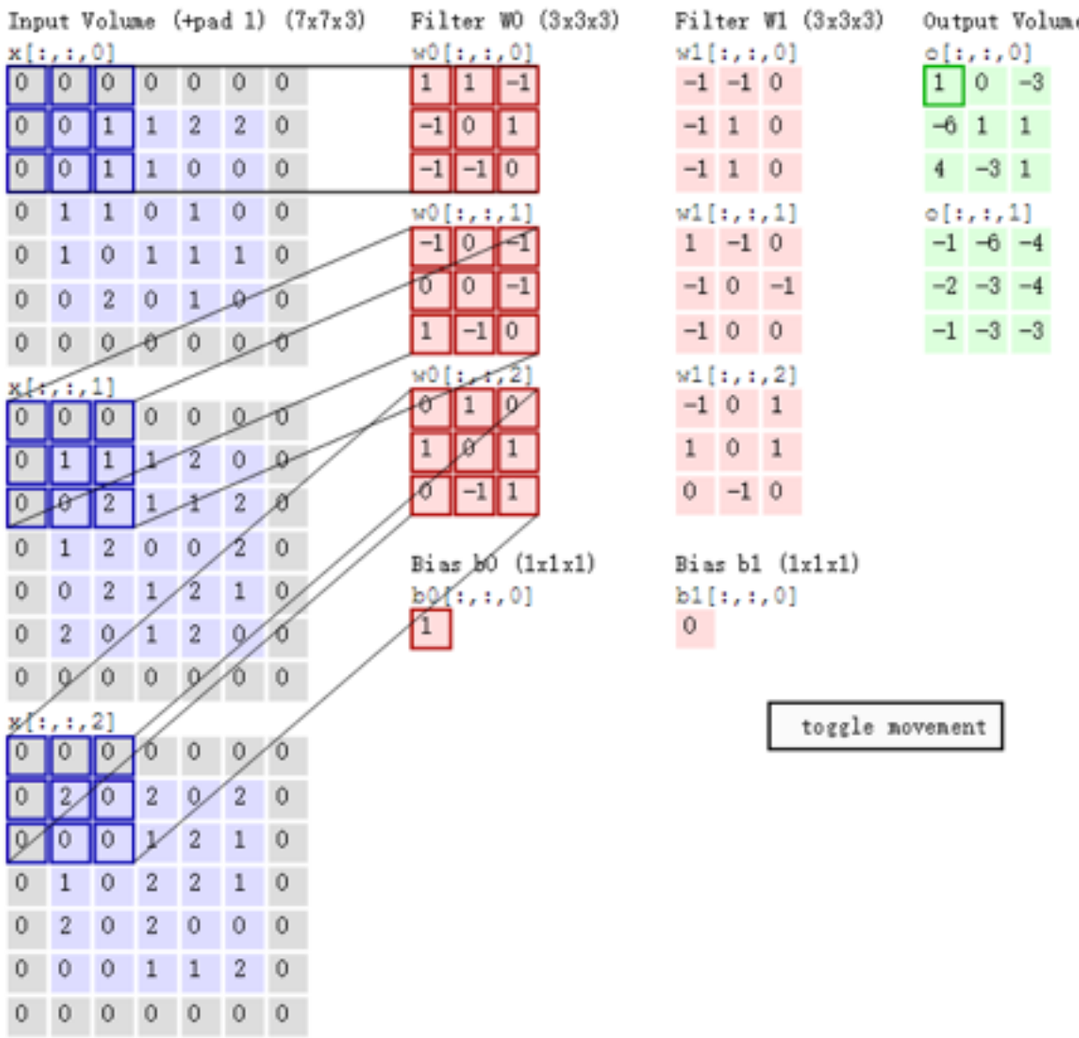




- input image: 5*5*3
- zero-padding=1: input changes to 7*7*3 (width: 7, height: 7, and with three color channels R,G,B)
- depth=2: two filters (filter w0, fiber w1)
- stride=2: the filters jump 2 pixels at a time as we slide them around. fitter size:3*3
- output: 3*3*2

Convolution Operation

- Local regions detection
- Parameter Sharing: because we want to capture the same feature at different places in an image, one filter slides over the input image to produce different outputs, but constrains the neurons in each depth slice to use the same weights and bias.



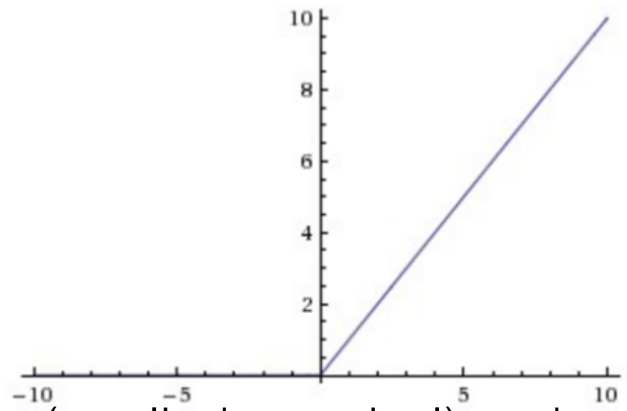
wx + b=

1* 0 + 1*0 + -1*0
+
-1*0 + 0*0 + 1*1
+
-1*0 + -1*0 + 0*1
+
-1*0 + 0*0 + -1*0
+
0*0 + 0*1 + -1*1
+
1*0 + -1*0 + 0*2
+
0*0 + 1*0 + 0*0
+
1*0 + 0*2 + 1*0
+
0*0 + -1*0 + 1*0
+
1
=
1

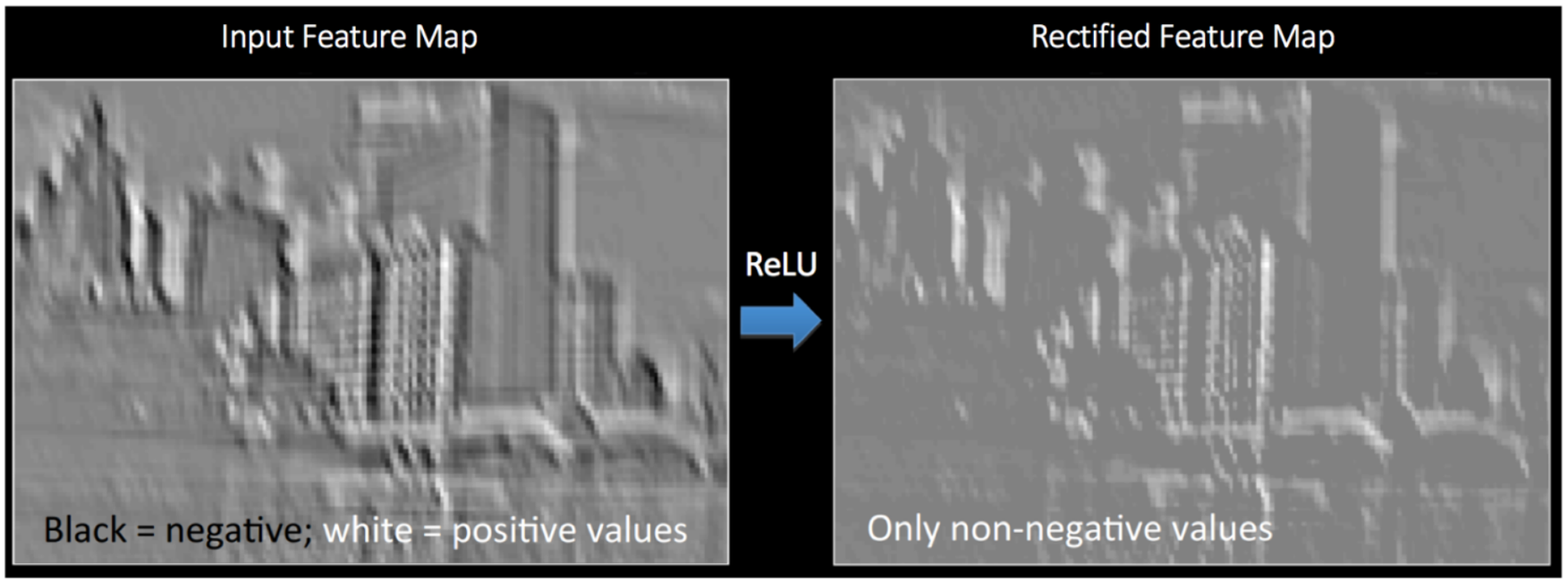
Relu Activation Layer

ReLU stands for Rectified Linear Unit and is a non-linear operation.

Output = Max(zero, Input)



ReLU is an element wise operation (applied per pixel) and replaces all negative pixel values in the feature map by zero. The purpose of ReLU is to introduce non-linearity in a ConvNet, since most of the real-world data we would want our ConvNet to learn would be non-linear (Convolution is a linear operation – element wise matrix multiplication and addition, so we account for non-linearity by introducing a non-linear function like ReLU).

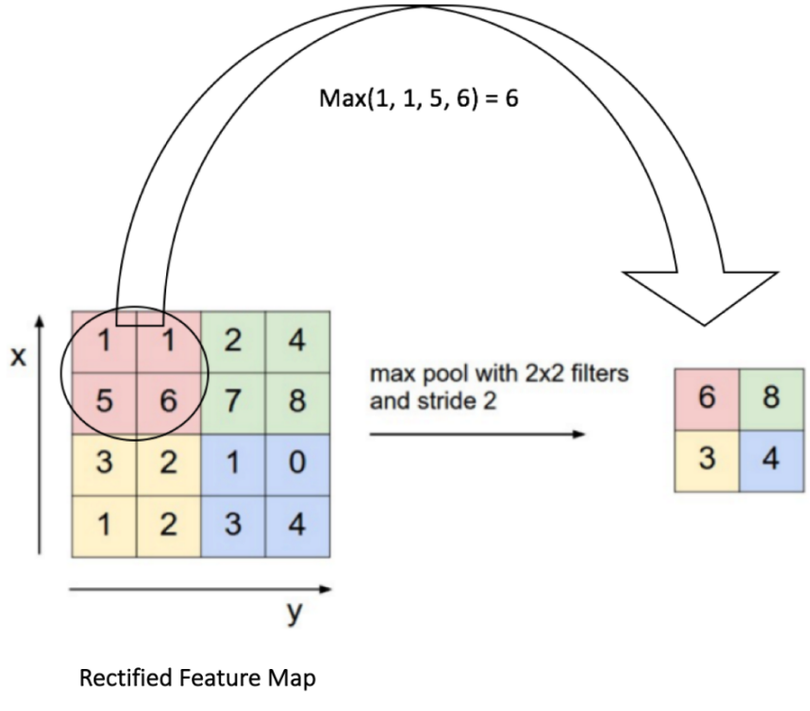


Other non linear functions such as tanh or sigmoid can also be used instead of ReLU, but ReLU has been found to perform better in most situations.

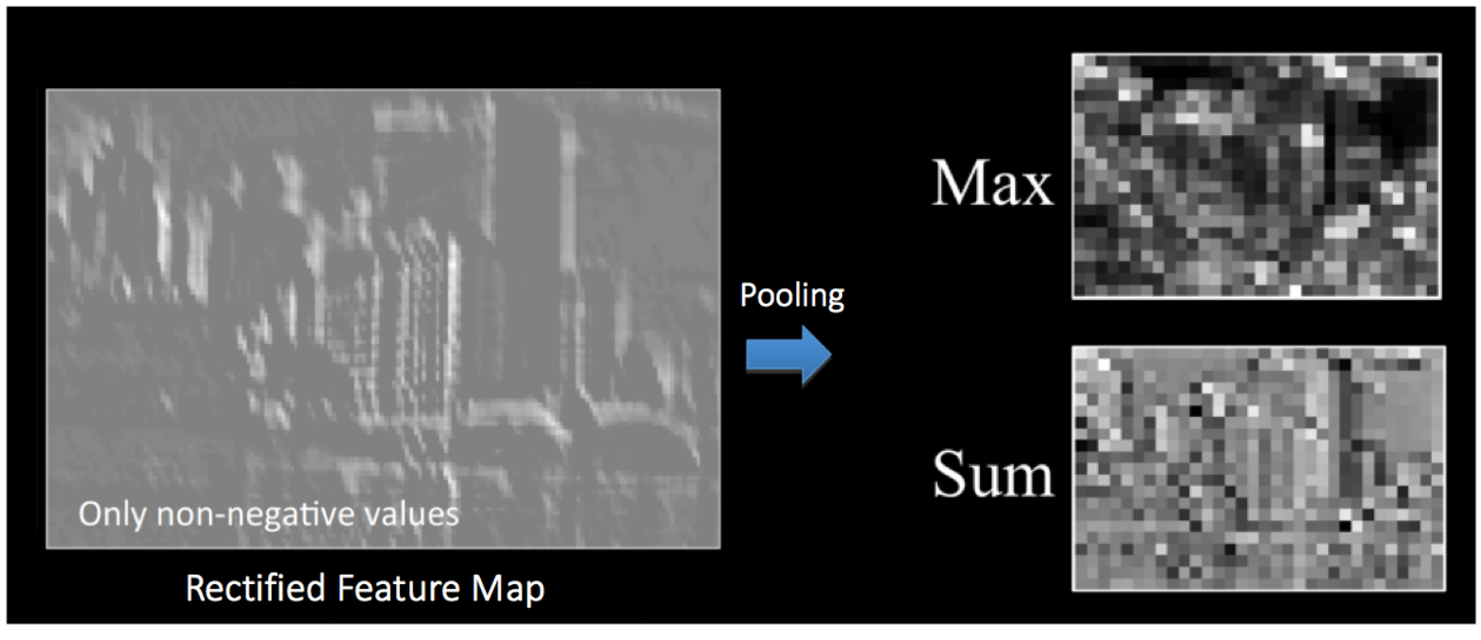
Pooling Layer

Spatial Pooling (also called subsampling or downsampling) reduces the dimensionality of each feature map but retains the most important information. Spatial Pooling can be of different types: Max, Average, Sum etc.

- In case of Max Pooling, we define a spatial neighborhood (for example, a 2×2 window) and take the largest element from the rectified feature map within that window. Instead of taking the largest element we could also take the average (Average Pooling) or sum of all elements in that window. In practice, Max Pooling has been shown to work better.



Rectified Feature Map

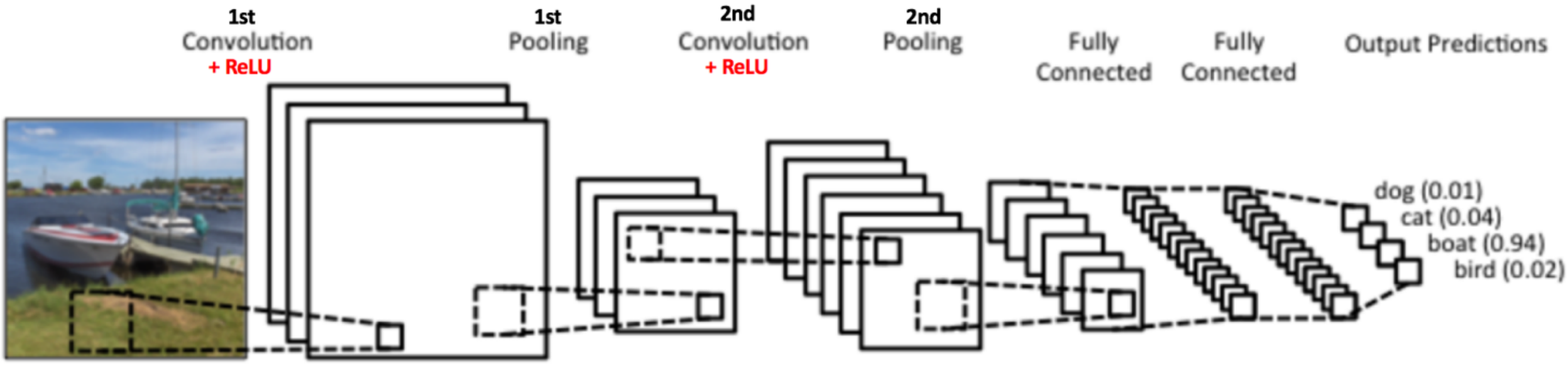


Pooling Layer

The function of Pooling is to progressively reduce the spatial size of the input representation.

- makes the input representations (feature dimension) smaller and more manageable.
- reduces the number of parameters and computations in the network, therefore, controlling overfitting
- makes the network invariant to small transformations, distortions and translations in the input image (a small distortion in input will not change the output of Pooling – since we take the maximum / average value in a local neighborhood).
- helps us arrive at an almost scale invariant representation of our image (the exact term is “equivariant”). This is very powerful since we can detect objects in an image no matter where they are located.

CNN Basics

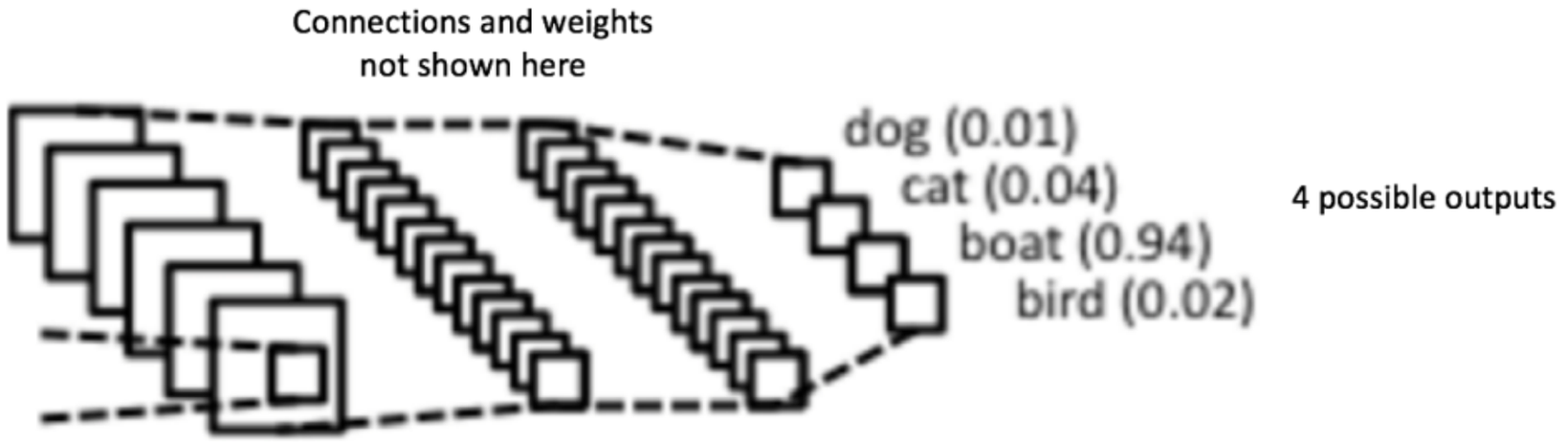


Convolution, ReLU and Pooling layers are the basic building blocks of any CNN. We have two sets of Convolution, ReLU & Pooling layers – the 2nd Convolution layer performs convolution on the output of the first Pooling Layer using six filters to produce a total of six feature maps. ReLU is then applied individually on all of these six feature maps. We then perform Max Pooling operation separately on each of the six rectified feature maps.

- Together these layers extract the useful features from the images, introduce non-linearity in our network and reduce feature dimension while aiming to make the features somewhat equivariant to scale and translation.

Fully Connected Layer

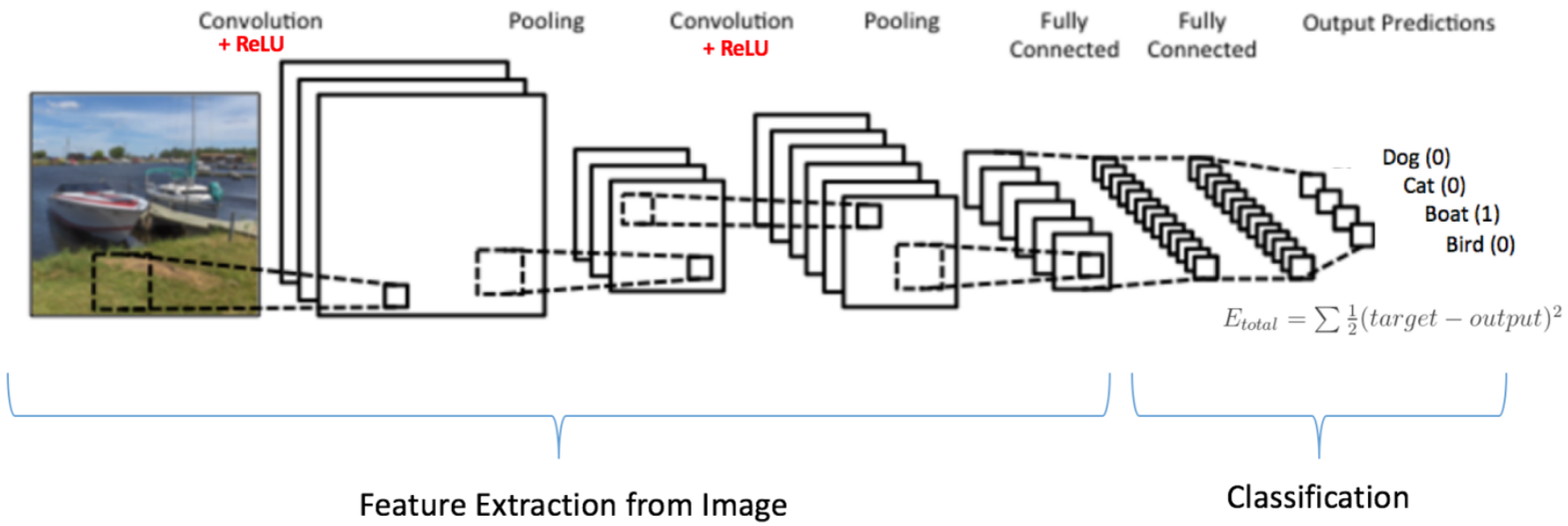
The Fully Connected layer is a traditional Multi Layer Perceptron that uses a softmax activation function in the output layer .The term “Fully Connected” implies that every neuron in the previous layer is connected to every neuron on the next layer.
The output from the convolutional and pooling layers represent high-level features of the input image. The purpose of the Fully Connected layer is to use these features for classifying the input image into various classes based on the training dataset.



Apart from classification, adding a fully-connected layer is also a cheap way of learning non-linear combinations of these features. Most of the features from convolutional and pooling layers may be good for the classification task, but combinations of those features might be even better.
The sum of output probabilities from the Fully Connected Layer is 1. This is ensured by using the Softmax as the activation function in the output layer of the Fully Connected Layer. The Softmax function takes a vector of arbitrary real-valued scores and squashes it to a vector of values between zero and one that sum to one.

Training using Backpropagation

since the input image is a boat, the target probability is 1 for Boat class and 0 for other three classes, i.e.
Input Image = Boat
Target Vector = [0, 0, 1, 0]



The overall training process of the Convolution Network may be summarized as below:

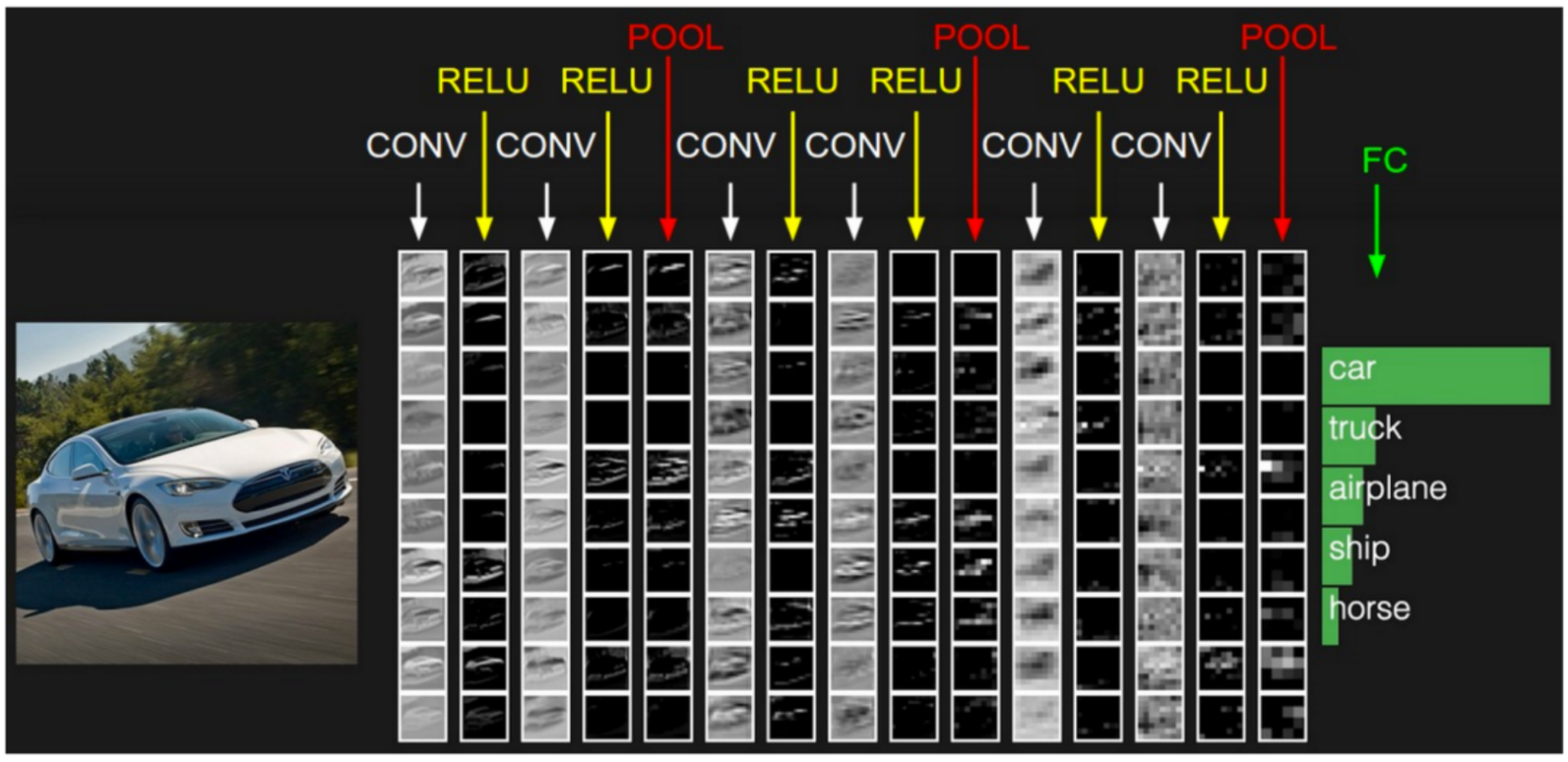
- Step1: We initialize all filters and parameters / weights with random values
- Step2: The network takes a training image as input, goes through the forward propagation step (convolution, ReLU and pooling operations along with forward propagation in the Fully Connected layer) and finds the output probabilities for each class.
  - Lets say the output probabilities for the boat image above are [0.2, 0.4, 0.1, 0.3]
  - Since weights are randomly assigned for the first training example, output probabilities are also random.
- Step3: Calculate the total error at the output layer (summation over all 4 classes)
  - Total Error = ∑ ½ (target probability – output probability) ^{2}
- Step4: Use Backpropagation to calculate the gradients of the error with respect to all weights in the network and use gradient descent to update all filter values / weights and parameter values to minimize the output error.
  - The weights are adjusted in proportion to their contribution to the total error.
  - When the same image is input again, output probabilities might now be [0.1, 0.1, 0.7, 0.1], which is closer to the target vector [0, 0, 1, 0].
  - This means that the network has learnt to classify this particular image correctly by adjusting its weights / filters such that the output error is reduced.
  - Parameters like number of filters, filter sizes, architecture of the network etc. have all been fixed before Step 1 and do not change during training process – only the values of the filter matrix and connection weights get updated.
- Step5: Repeat steps 2-4 with all images in the training set.

Training using Backpropagation

The above steps train the ConvNet – this essentially means that all the weights and parameters of the ConvNet have now been optimized to correctly classify images from the training set.

When a new (unseen) image is input into the ConvNet, the network would go through the forward propagation step and output a probability for each class (for a new image, the output probabilities are calculated using the weights which have been optimized to correctly classify all the previous training examples). If our training set is large enough, the network will (hopefully) generalize well to new images and classify them into correct categories.

In the example above we used two sets of alternating Convolution and Pooling layers. Please note however, that these operations can be repeated any number of times in a single ConvNet. In fact, some of the best performing ConvNets today have tens of Convolution and Pooling layers! Also, it is not necessary to have a Pooling layer after every Convolutional Layer. As can be seen in the Figure below, we can have multiple Convolution + ReLU operations in succession before having a Pooling operation.



Tensorflow

- TensorFlow™ is an open source software library for numerical computation using data flow graphs.
- Nodes in the graph represent mathematical operations, while the graph edges represent the multidimensional data arrays (tensors) communicated between them. The flexible architecture allows you to deploy computation to one or more CPUs or GPUs in a desktop, server, or mobile device with a single API.
- TensorFlow was originally developed by researchers and engineers working on the Google Brain Team within Google's Machine Intelligence research organization for the purposes of conducting machine learning and deep neural networks research, but the system is general enough to be applicable in a wide variety of other domains as well.

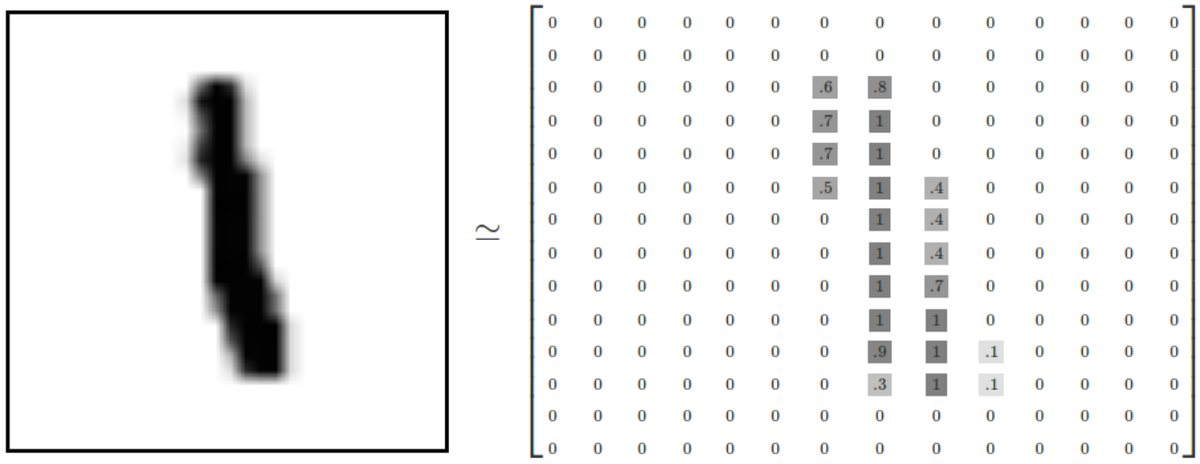
Documentation: https://www.tensorflow.org/overview

installation: https://anaconda.org/conda-forge/tensorflow

Tensorflow

MNIST is a simple computer vision dataset. It consists of images of handwritten digits like these:

Each image is 28 pixels by 28 pixels. We can interpret this as a vector of 28x28 = 784 numbers

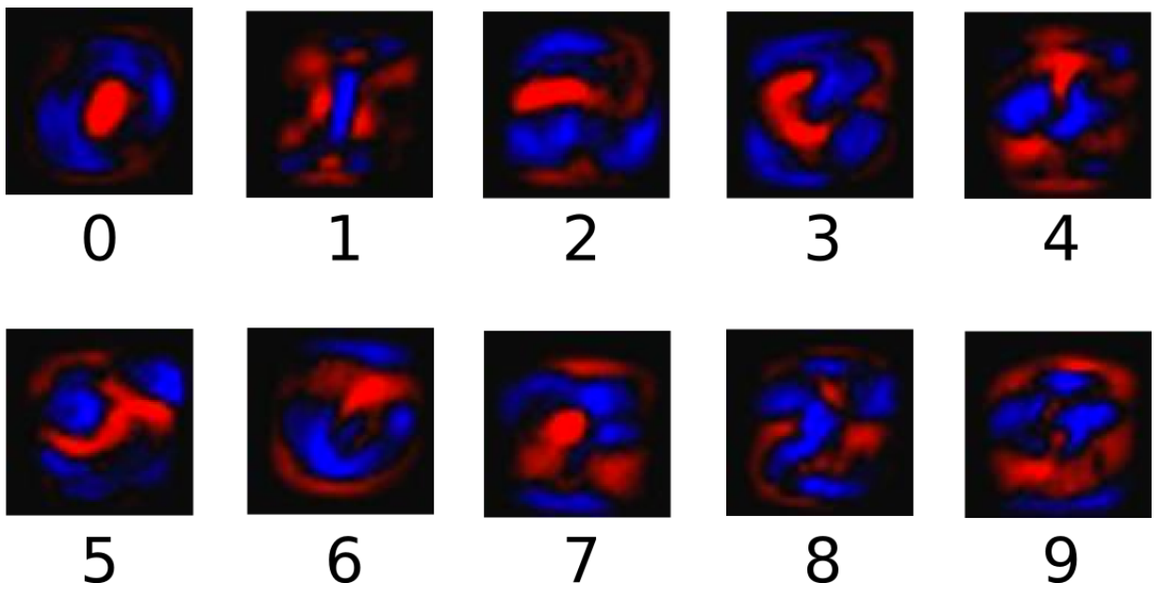


- mnist.train.images is a tensor (an n-dimensional array) with a shape of [55000, 784]. The first dimension is an index into the list of images and the second dimension is the index for each pixel in each image. Each entry in the tensor is a pixel intensity between 0 and 1, for a particular pixel in a particular image.
- A one-hot vector is a vector which is 0 in most dimensions, and 1 in a single dimension. In this case, the nth digit will be represented as a vector which is 1 in the nth dimension. For example, 3 would be [0,0,0,1,0,0,0,0,0,0]. Consequently, mnist.train.labels is a [55000, 10] array of floats.

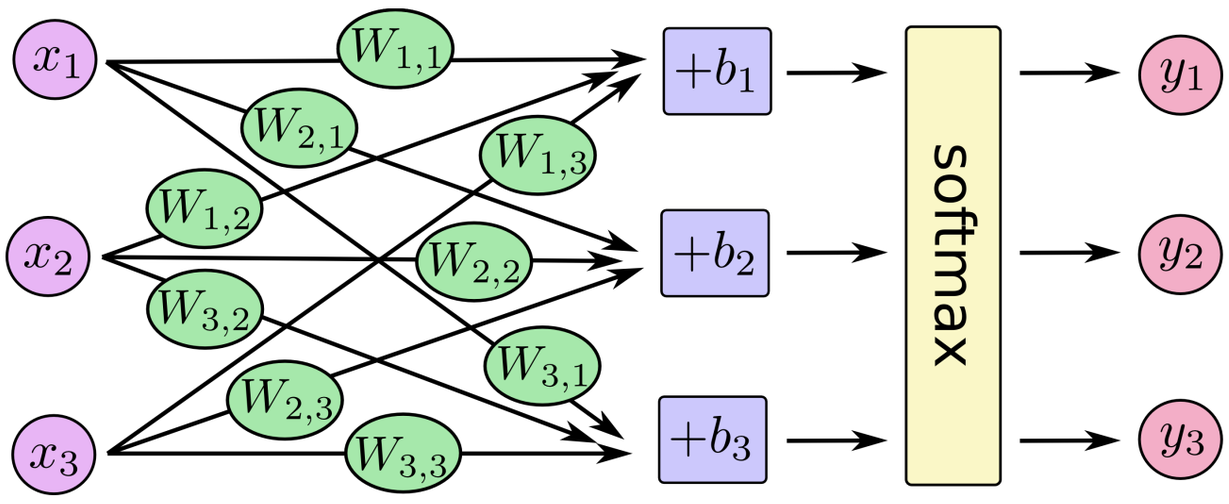
Softmax Regressions

Every image in MNIST is of a handwritten digit between zero and nine. So there are only ten possible things that a given image can be. We want to be able to look at an image and give the probabilities for it being each digit. For example, our model might look at a picture of a nine and be 80% sure it's a nine, but give a 5% chance to it being an eight and a bit of probability to all the others because it isn't 100% sure.

A softmax regression has two steps:
do a weighted sum of the pixel intensities. The weight is negative if that pixel having a high intensity is evidence against the image being in that class, and positive if it is evidence in favor.



Red represents negative weights, while blue represents positive weights.



$$\text{evidence}_i = \sum_j W_{i,\,j} x_j + b_i$$

$$y = \text{softmax}(\text{evidence})$$

$$\text{softmax}(x) = \text{normalize}(\exp(x))$$

$$\text{softmax}(x)_i = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$$

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \text{softmax}\left(\begin{bmatrix} W_{1,1} & W_{1,2} & W_{1,3} \\ W_{2,1} & W_{2,2} & W_{2,3} \\ W_{3,1} & W_{3,2} & W_{3,3} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} \right)$$

Training

In order to train our model, we need to define what it means for the model to be good.
The cost, or the loss, represents how far off our model is from our desired outcome.
We try to minimize that error, and the smaller the error margin, the better our model is.

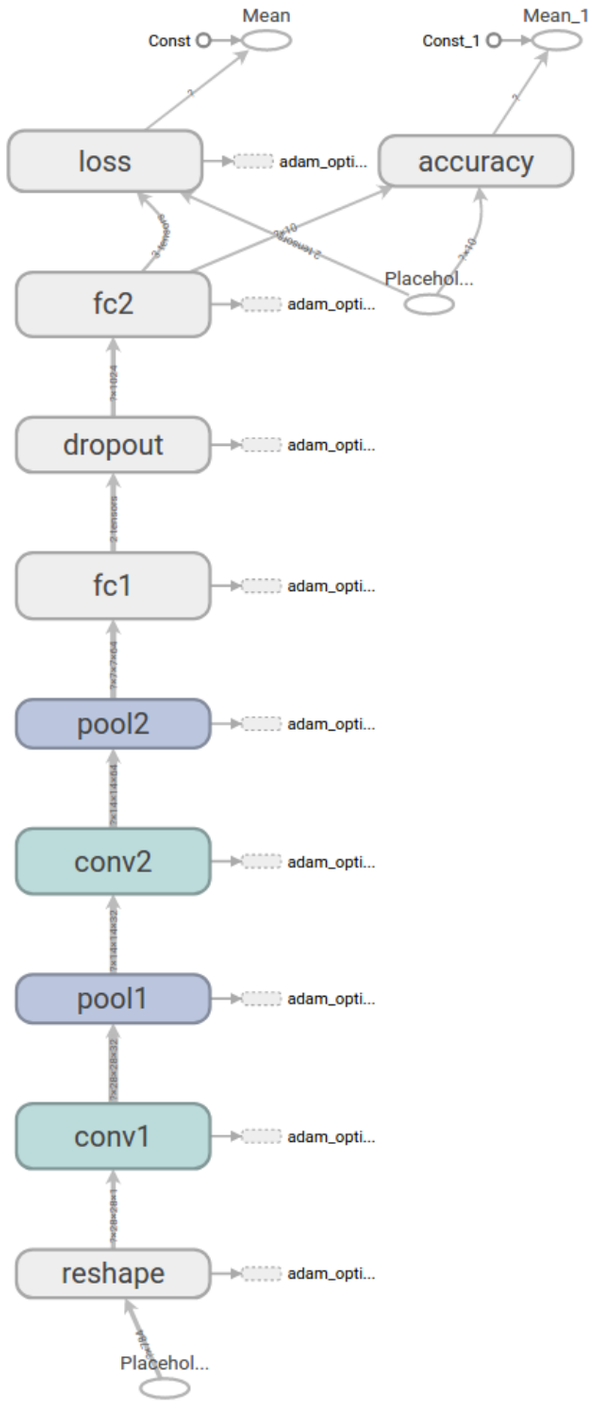
Minimize cross-entropy:

$$H_{y'}(y) = -\sum_i y'_i \log(y_i)$$

Where y is our predicted probability distribution, and y′ is the true distribution (the one-hot vector with the digit labels)

Use the backpropagation algorithm to minimize cross_entropy using the gradient descent algorithm with a learning rate

Build a Multilayer Convolutional Network



Keras: The Python Deep Learning library

Keras is a high-level neural networks API, written in Python and capable of running on top of TensorFlow, CNTK, or Theano. It was developed with a focus on enabling fast experimentation.

Use Keras if you need a deep learning library that:

- Allows for easy and fast prototyping (through user friendliness, modularity, and extensibility).
- Supports both convolutional networks and recurrent networks, as well as combinations of the two.
- Runs seamlessly on CPU and GPU.

Documentation: https://keras.io/getting_started/

installation: https://anaconda.org/conda-forge/keras

Layers

- Reshape
  - # as first layer in a Sequential model
  - model = Sequential()
  - model.add(Reshape((3, 4), input_shape=(12,)))
- Dense: densely-connected NN layer
  - model.add(Dense(units=64, input_dim=100))
- Activation
  - model.add(Activation('relu'))
- Dropout
  - Model.add(Dropout())
- Con2D
  - filters: the number output of filters in the convolution
  - kernel_size: specifying the width and height of the 2D convolution window.
  - strides: specifying the strides of the convolution along the width and height.
  - padding: one of "valid" or "same" .
  - data_format:channels_last corresponds to inputs with shape (batch, height, width, channels) while channels_first corresponds to inputs with shape (batch, channels, height, width). default: “channels_last".
- Flatten: Flattens the input
  - model = Sequential()
  - # now: model.output_shape == (None, 64, 32, 32)
  - model.add(Flatten())
  - # now: model.output_shape == (None, 65536)
- MaxPooling2D
  - model.add(MaxPooling2D(pool_size=(2, 2), strides=None, padding='valid', data_format=None))
- pool_size: :factors by which to downscale (vertical, horizontal).
- strides: Strides values. If None, it will default to pool_size.
- padding: One of "valid" or "same".

Compilation

```
# For a multi-class classification problem
model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# For a binary classification problem
model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['accuracy'])

# For a mean squared error regression problem
model.compile(optimizer='rmsprop',
              loss='mse')

# For custom metrics
import keras.backend as K

def mean_pred(y_true, y_pred):
    return K.mean(y_pred)

model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['accuracy', mean_pred])
```

Compilation

Different optimizers:

SGD: Stochastic gradient descent optimizer.

keras.optimizers.SGD(lr=0.01, momentum=0.0, decay=0.0, nesterov=False)

- lr: float >= 0. Learning rate.
- momentum: float >= 0. Parameter updates momentum.
- decay: float >= 0. Learning rate decay over each update.
- nesterov: boolean. Whether to apply Nesterov momentum.

RMSprop:Divide the gradient by a running average of its recent magnitude

keras.optimizers.RMSprop(lr=0.001, rho=0.9, epsilon=1e-08, decay=0.0)

lr: float >= 0. Learning rate.

rho: float >= 0.

epsilon: float >= 0. Fuzz factor.

decay: float >= 0. Learning rate decay over each update.

Adagrad:Adaptive Subgradient Methods for Online Learning and Stochastic Optimization

keras.optimizers.Adagrad(lr=0.01, epsilon=1e-08, decay=0.0)

Adadelta:an adaptive learning rate method

keras.optimizers.Adadelta(lr=1.0, rho=0.95, epsilon=1e-08, decay=0.0)

Adamax:A Method for Stochastic Optimization

keras.optimizers.Adamax(lr=0.002, beta_1=0.9, beta_2=0.999, epsilon=1e-08, decay=0.0)

Nadam:Nesterov Adam optimizer.

keras.optimizers.Nadam(lr=0.002, beta_1=0.9, beta_2=0.999, epsilon=1e-08, schedule_decay=0.004)

TFOptimizer:Wrapper class for native TensorFlow optimizers.

keras.optimizers.TFOptimizer(optimizer)