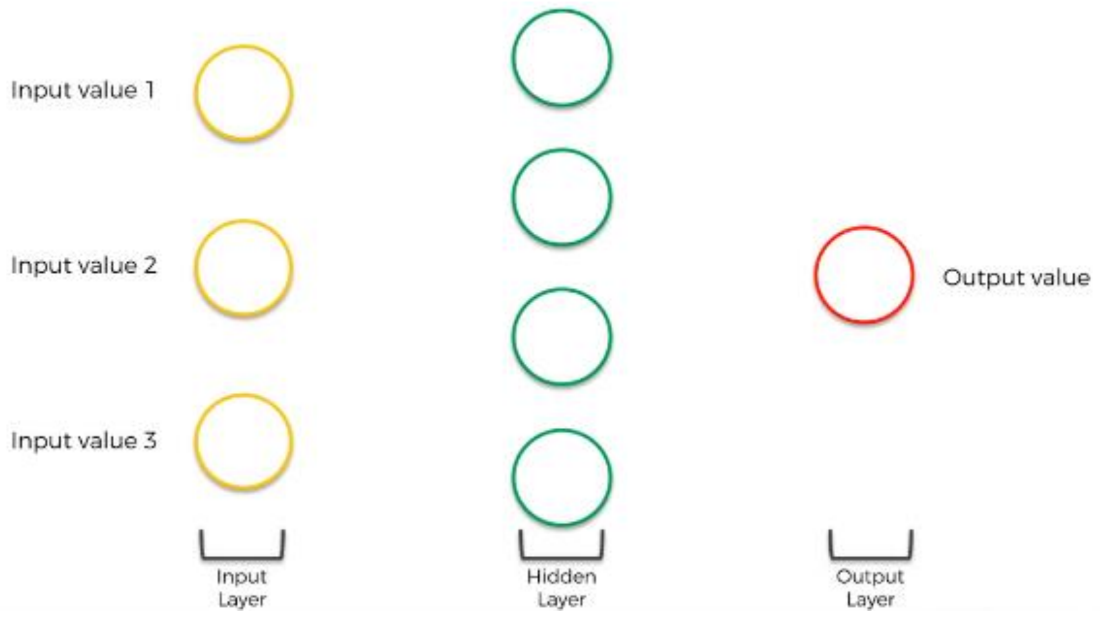
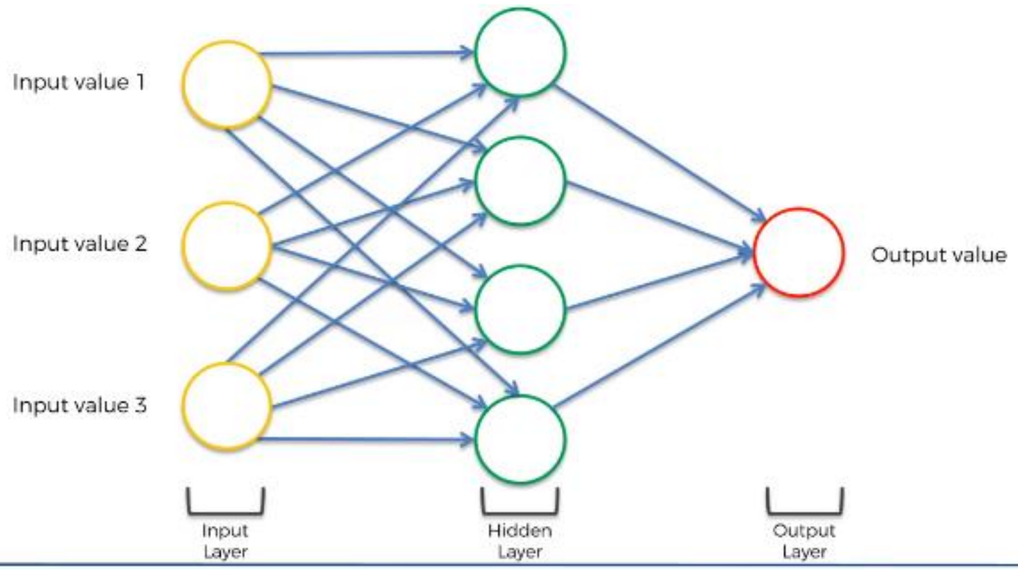
Understanding Neural Networks. From neuron to RNN, CNN, and Deep Learning

Neural Networks is one of the most popular machine learning algorithms at present. It has been decisively proven over time that neural networks outperform other algorithms in accuracy and speed. With various variants like CNN (Convolutional Neural Networks), RNN(Recurrent Neural Networks), AutoEncoders, Deep Learning etc. neural networks are slowly becoming for data scientists or machine learning practitioners what linear regression was one for statisticians. It is thus imperative to have a fundamental understanding of what a Neural Network is, how it is made up and what is its reach and limitations. This post is an attempt to explain a neural network starting from its most basic building block a neuron, and later delving into its most popular variations like CNN, RNN etc.

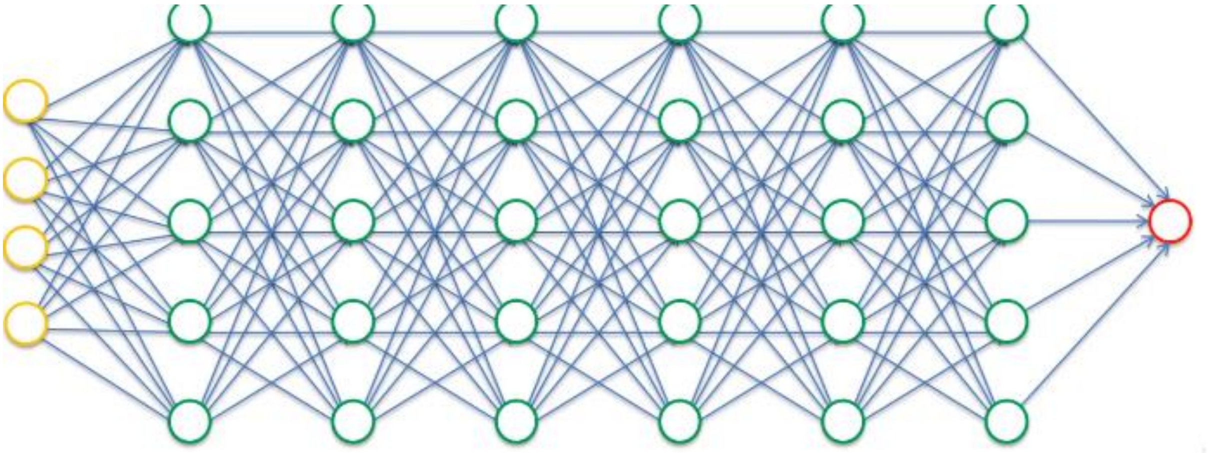


Input --→ hiiden layer--→ o/p

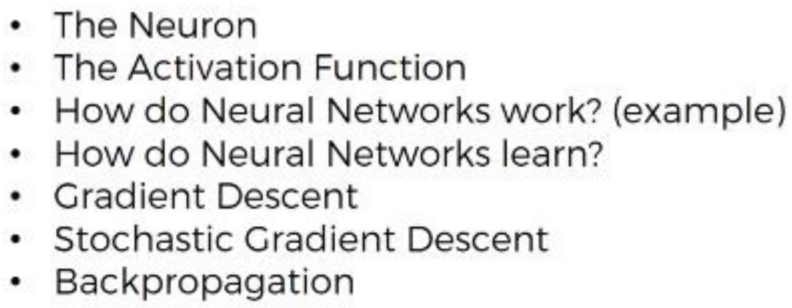


When number of hidden layer is 1 then it is called as shallow learning

But as number of hidden layer grows thenit is called as deep learning



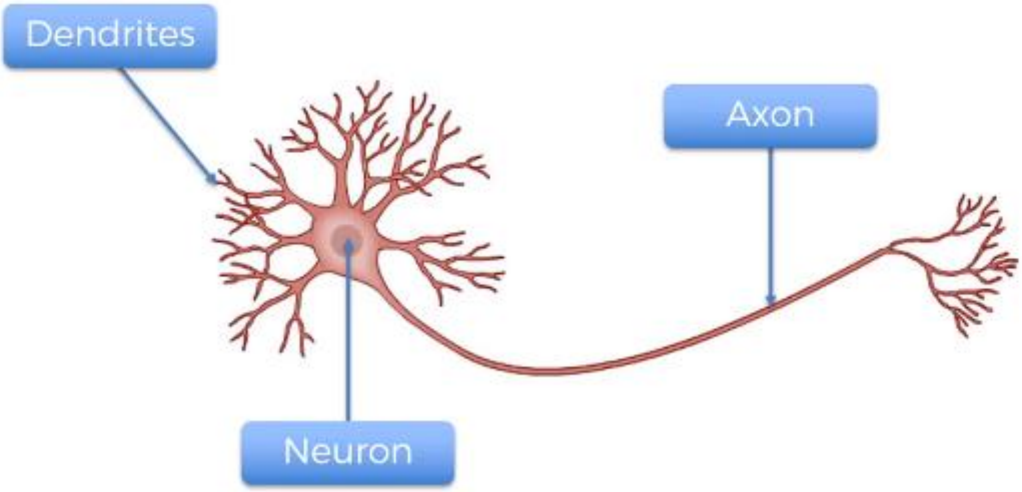
We will look into

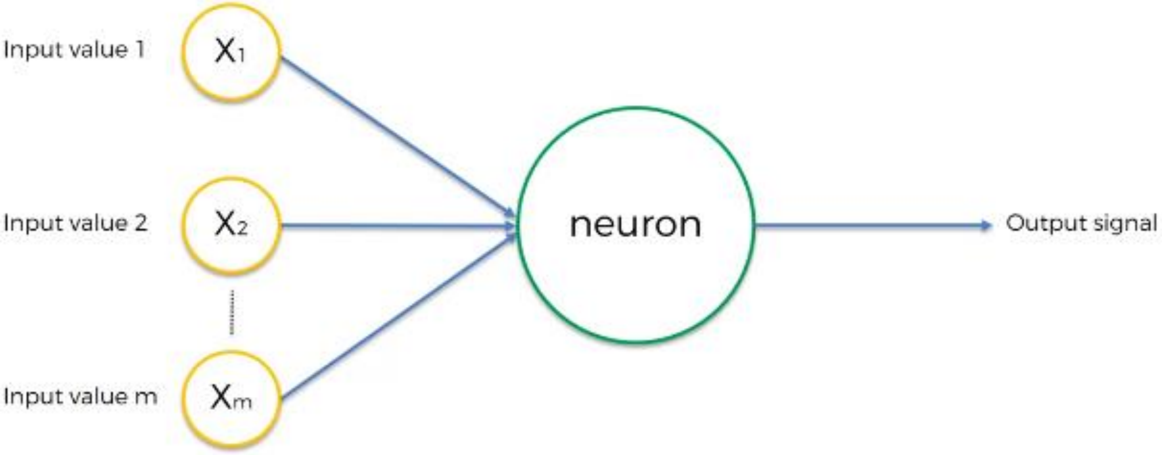


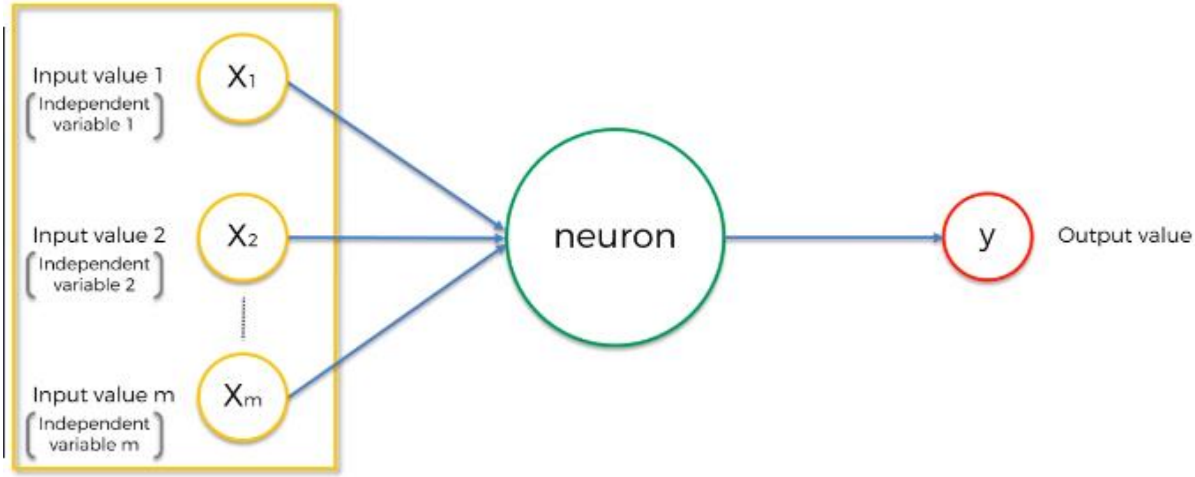
- The Neuron
- The Activation Function
- How do Neural Networks work? (example)
- How do Neural Networks learn?
- Gradient Descent
- Stochastic Gradient Descent
- Backpropagation

What is a Neuron?

As the name suggests, neural networks were inspired by the neural architecture of a human brain, and like in a human brain the basic building block is called a Neuron. Its functionality is similar to a human neuron, i.e. it takes in some inputs and fires an output. In purely mathematical terms, a neuron in the machine learning world is a placeholder for a mathematical function, and its only job is to provide an output by applying the function on the inputs provided.







Multiple independent variables are given as i/p then it is normalized /processed/regularized and then given as i/p to next layer. Finally we get o/p

Extra info about normalization

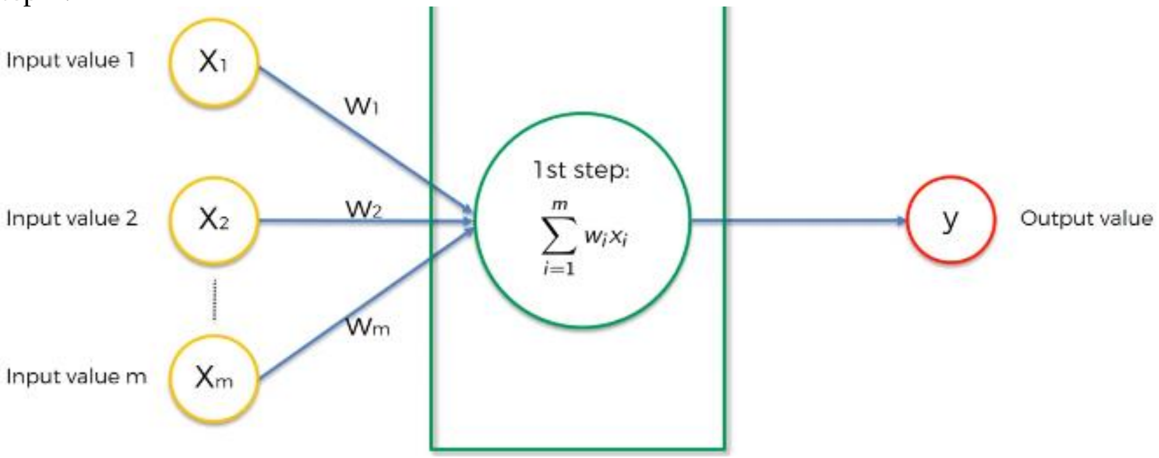
Link:

http://yann.lecun.com/exdb/publis/pdf/lecun-98b.pdf

o/p value can be a single continuous variable/categorical variable/Boolean variable

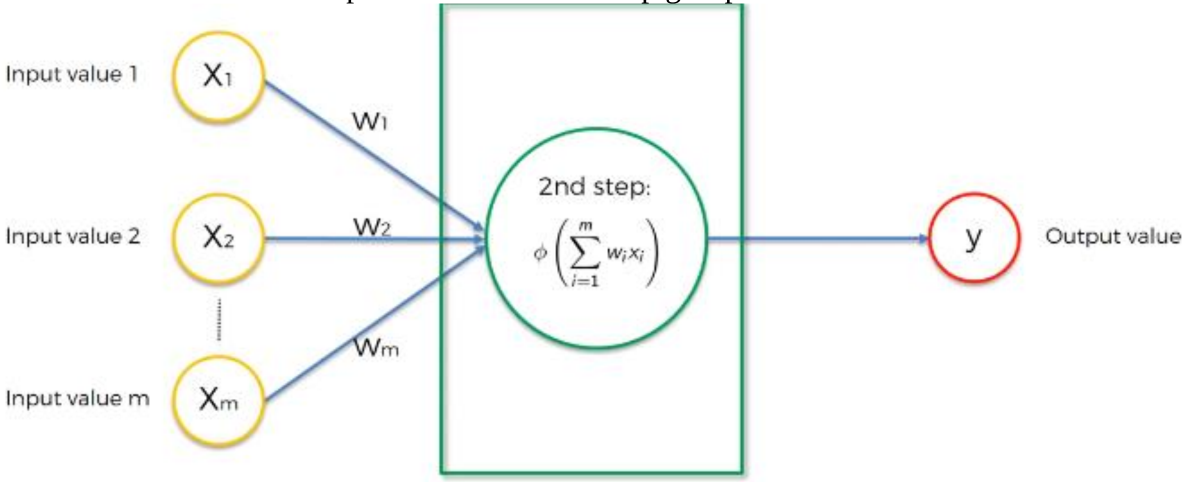
On i/p variables weights are applied. These weights are called as synupse. These are very important. Which signal is important, which signal will be passed further all these are decided by weights

Step 1:



Then in neuron with the help of actified function i/p gets processed

Then in neuron with the help of actified function i/p gets processed

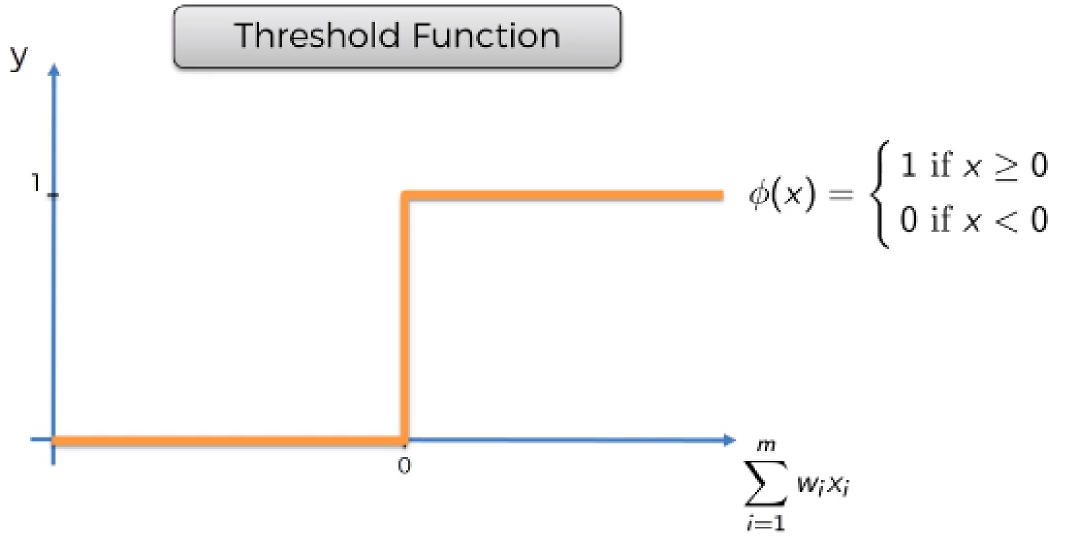


The function used in a neuron is generally termed as an activation function. There are many but There have been 4 major activation functions tried to date, Threshold(step), sigmoid, tanh, and ReLU. Each of these is described in detail below.

ACTIVATION FUNCTIONS

Threshold(Step) function

A step function is defined as

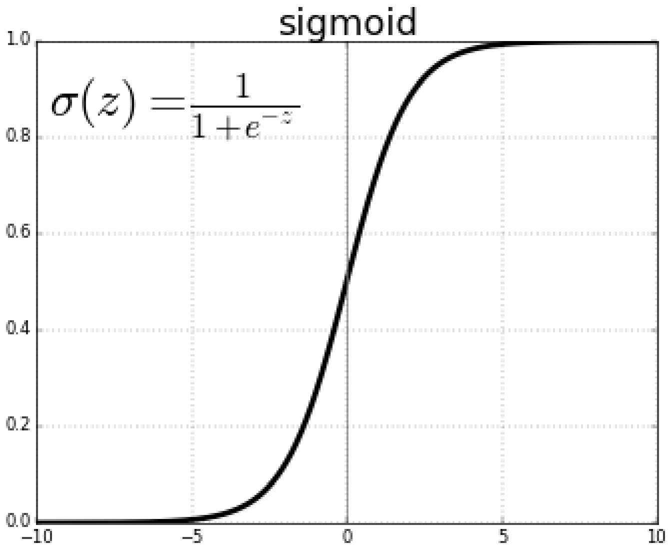


Where the output is 1 if the value of x is greater than equal to zero and 0 if the value of x is less than zero. As one can see a step function is non-differentiable at zero. At present, a neural network uses back propagation method along with gradient descent to calculate weights of different layers. Since the step function is non-differentiable at zero hence it is not able to make progress with the gradient descent approach and fails in the task of updating the weights.

To overcome, this problem sigmoid functions were introduced instead of the step function.

Sigmoid Function

A sigmoid function or logistic function is defined mathematically as



The value of the function tends to zero when z or independent variable tends to negative infinity and tends to 1 when z tends to infinity. It needs to be kept in mind that this function represents an approximation of the behavior of the dependent variable and is an assumption. Now the question arises as to why we use the sigmoid function as one of the approximation functions. There are certain simple reasons for this.

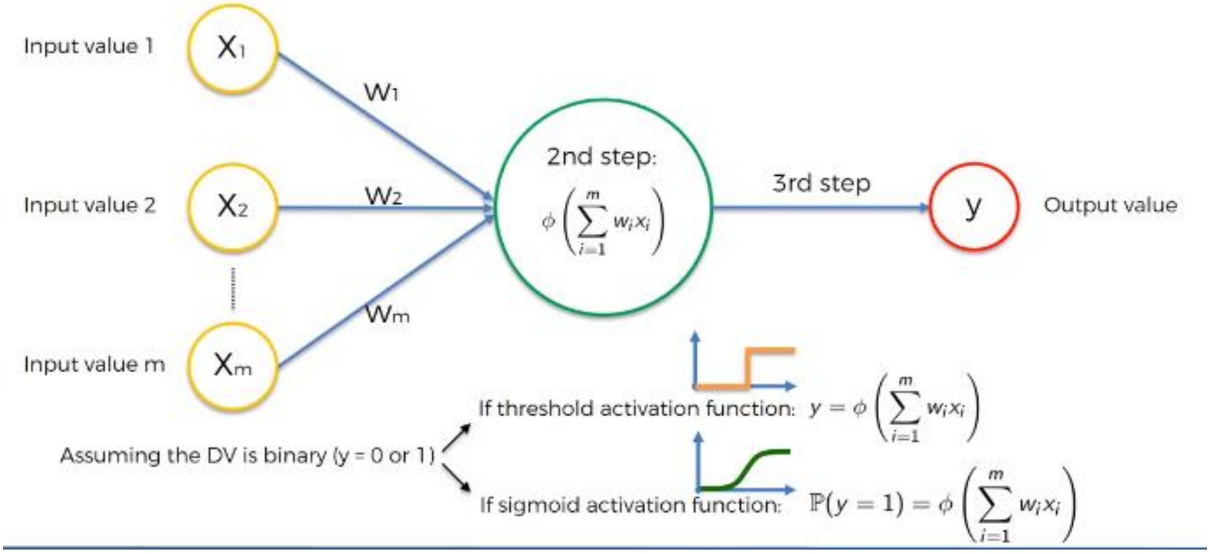
1. It captures non-linearity in the data. Albeit in an approximated form, but the concept of non-linearity is essential for accurate modeling.

2. The sigmoid function is differentiable throughout and hence can be used with gradient descent and backpropagation approaches for calculating weights of different layers

3. The assumption of a dependent variable to follow a sigmoid function inherently assumes a Gaussian distribution for the independent variable which is a general distribution we see for *a lot of randomly occurring events and this is a good generic distribution to start with.*

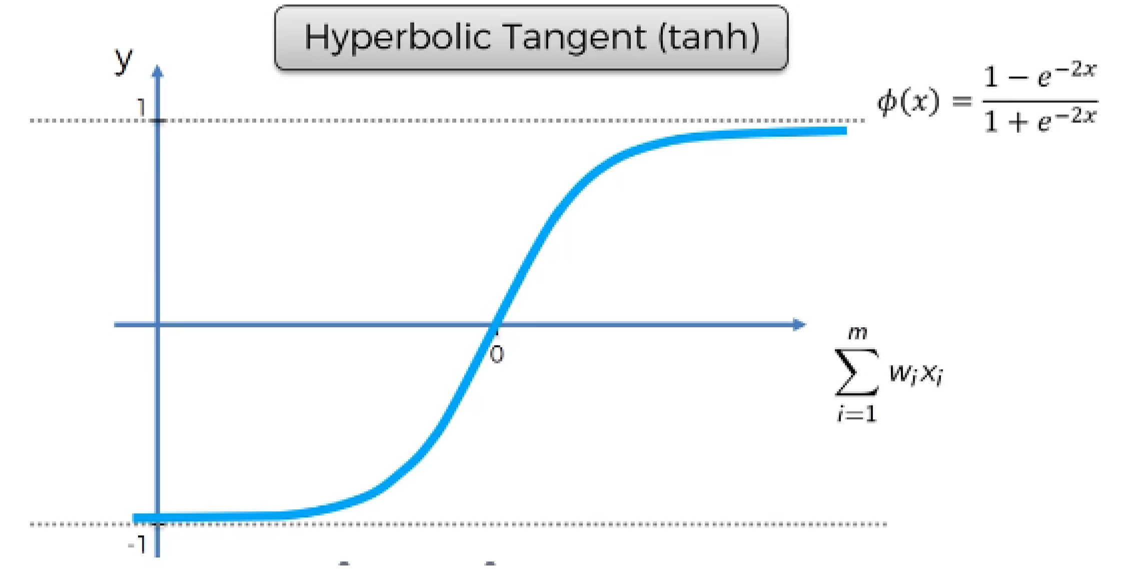
However, a sigmoid function also suffers from a problem of vanishing gradients. As can be seen from the picture a sigmoid function squashes it's input into a very small output range [0,1] and has very steep gradients. Thus, there remain large regions of input space, where even a large change produces a very small change in the output. This is referred to as the problem of vanishing gradient. This problem increases with an increase in the number of layers and thus stagnates the learning of a neural network at a certain level.

If the o/p variable is 0 or 1 then which function u will use?


Input value 1
X_1
W_1
Input value 2
X_2
W_2
Input value m
X_m
W_m
2nd step:
$\phi\left(\sum_{i=1}^{m} w_i x_i\right)$
3rd step
y
Output value
If threshold activation function: $y = \phi\left(\sum_{i=1}^{m} w_i x_i\right)$
Assuming the DV is binary (y = 0 or 1)
If sigmoid activation function: $\mathbb{P}(y = 1) = \phi\left(\sum_{i=1}^{m} w_i x_i\right)$

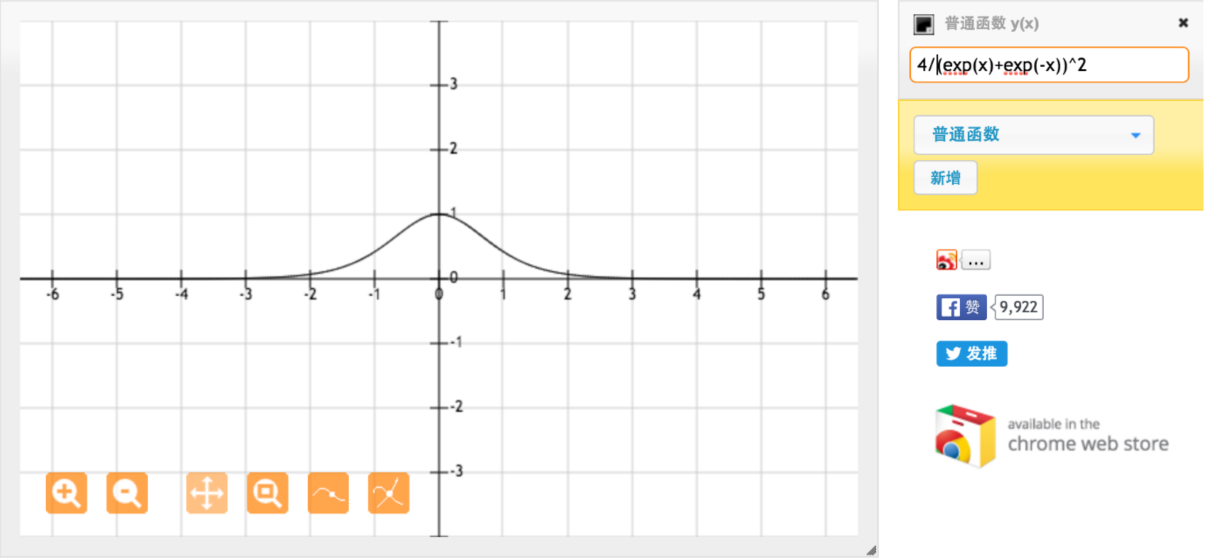
Tanh Function

The tanh(*z*) function is a rescaled version of the sigmoid, and its output range is [– 1,1] instead of [0,1]. [1]



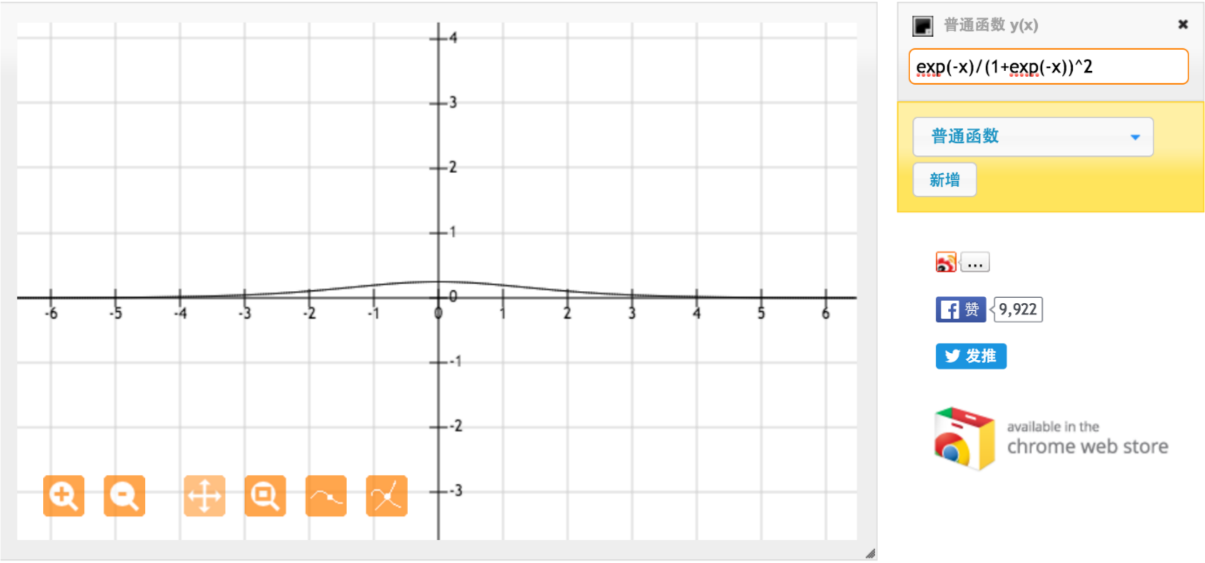
The general reason for using a Tanh function in some places instead of the sigmoid function is because since data is centered around 0, the derivatives are higher. A higher gradient helps in a better learning rate. Below attached are plotted gradients of two functions tanh and sigmoid. [2]

For tanh function, for an input between [-1,1], we have derivative between [0.42, 1].


普通函数 y(x)

4/(exp(x)+exp(-x))^2

For sigmoid function on the other hand, for input between [0,1], we have derivative between [0.20, 0.25]



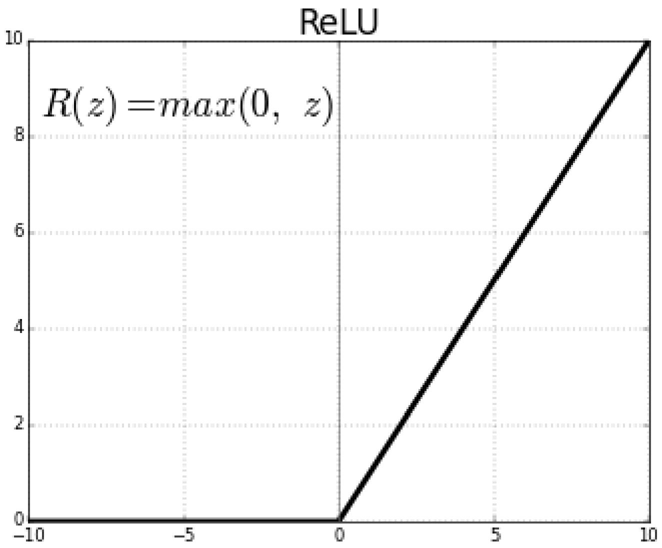
exp(-x)/(1+exp(-x))^2

As one can see from the pictures above a Tanh function has a higher range of derivative than a Sigmoid function and thus has a better learning rate. However, the problem of vanishing gradients still persists in Tanh function.

ReLU Function

The Rectified Linear Unit is the most commonly used activation function in deep learning models. The function returns 0 if it receives any negative input, but for any positive value x, it returns that value back. So, it can be written as f(x)=max (0, x).

Graphically it looks like this [3]

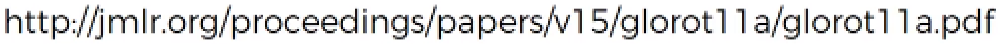


The Leaky ReLU is one of the most well-known. It is the same as ReLU for positive numbers. But instead of being 0 for all negative values, it has a constant slope (less than 1.).

That slope is a parameter the user sets when building the model, and it is frequently called α. For example, if the user sets α=0.3, the activation function is.`f(x) = max (0.3*x, x)` *This has the theoretical advantage that, by being influenced, by* `x` *at all values, it may make more complete use of the information contained in x.*

There are other alternatives, but both practitioners and researchers have generally found an insufficient benefit to justify using anything other than ReLU. In general practice as well, ReLU has found to be performing better than sigmoid or tanh functions.

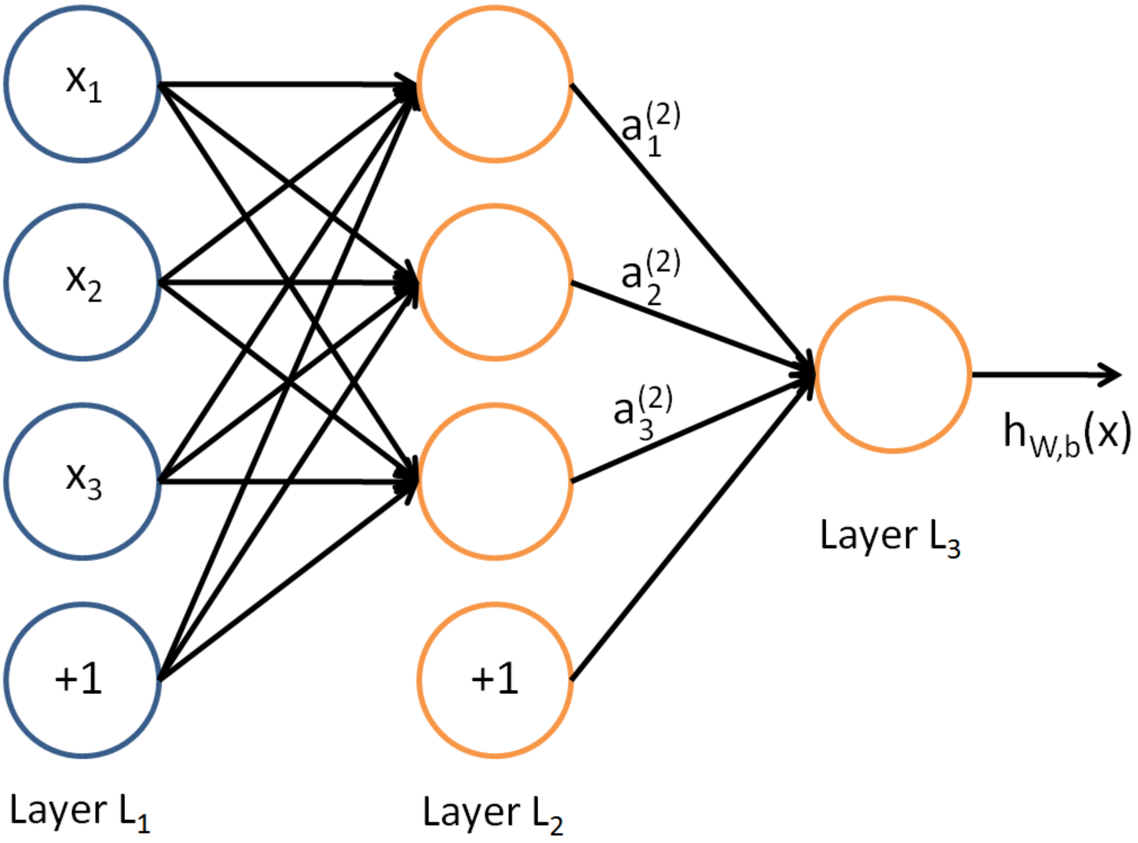
Extra info for activation functions


http://jmlr.org/proceedings/papers/v15/glorot11a/glorot11a.pdf

Neural Networks

Till now we have covered neuron and activation functions which together for the basic building blocks of any neural network. Now, we will dive in deeper into what is a Neural Network and different types of it. I would highly suggest people, to revisit neurons and activation functions if they have a doubt about it.

Before understanding a Neural Network, it is imperative to understand what is a layer in a Neural Network. A layer is nothing but a collection of neurons which take in an input and provide an output. Inputs to each of these neurons are processed through the activation functions assigned to the neurons. For example, here is a small neural network.



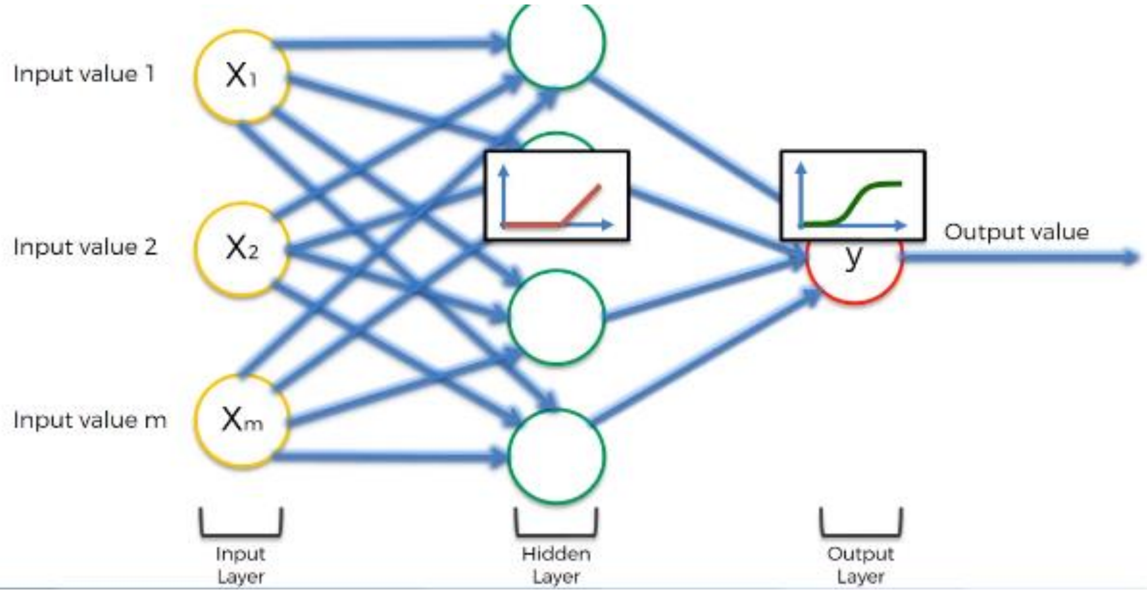
The leftmost layer of the network is called the ***input layer****, and the rightmost layer the* ***output layer*** *(which, in this example, has only one node). The middle layer of nodes is called the* ***hidden layer*** *because its values are not observed in the training set. We also say that our example neural network* *has 3 **input units** (not counting the bias unit), 3 **hidden units**, and 1 **output unit** [4]*

Any neural network has 1 input and 1 output layer. The number of hidden layers, for instance, differ between different networks depending upon the complexity of the problem to be solved.

Another important point to note here is that each of the hidden layers can have a different activation function, for instance, hidden layer1 may use a sigmoid function and hidden layer2 may use a ReLU, followed by a Tanh in hidden layer3 all in the same neural network. Choice of the activation function to be used again depends on the problem in question and the type of data being used.

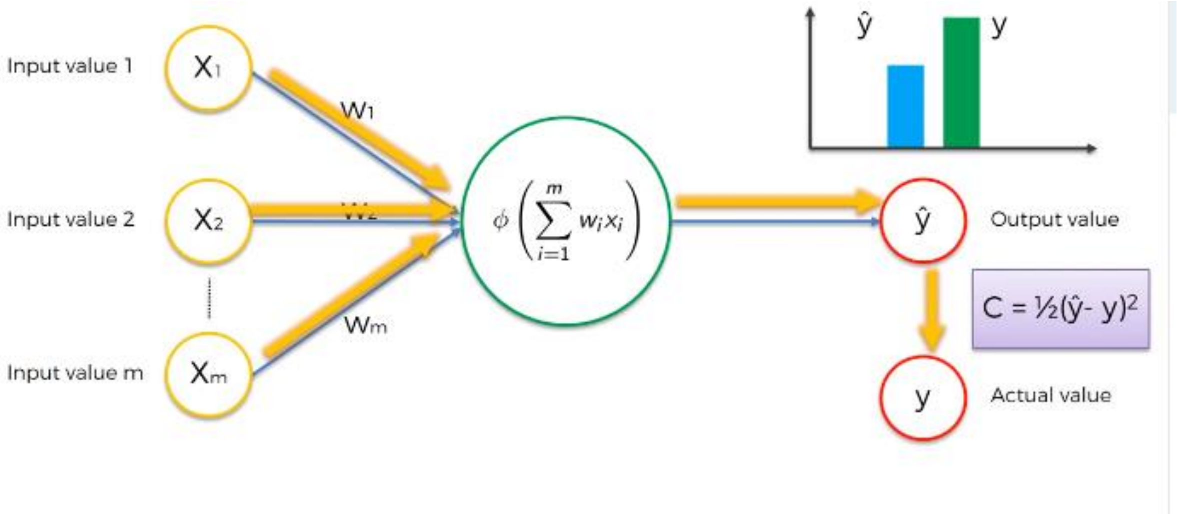
Now for a neural network to make accurate predictions each of these neurons learn certain weights at every layer. The algorithm through which they learn the weights is called back propagation, the details of which are beyond the scope of this post.

A neural network having more than one hidden layer is generally referred to as a Deep Neural Network.

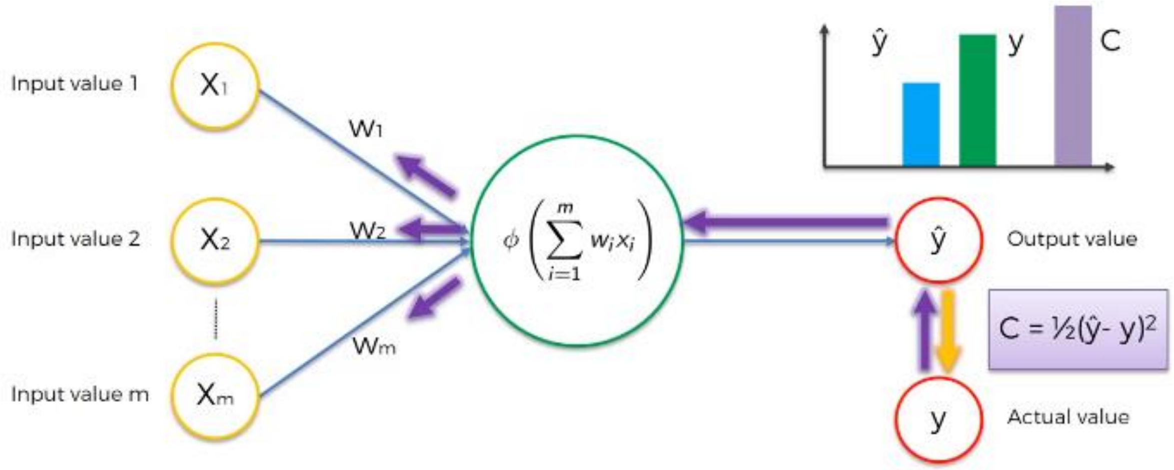


This combination is very common combination

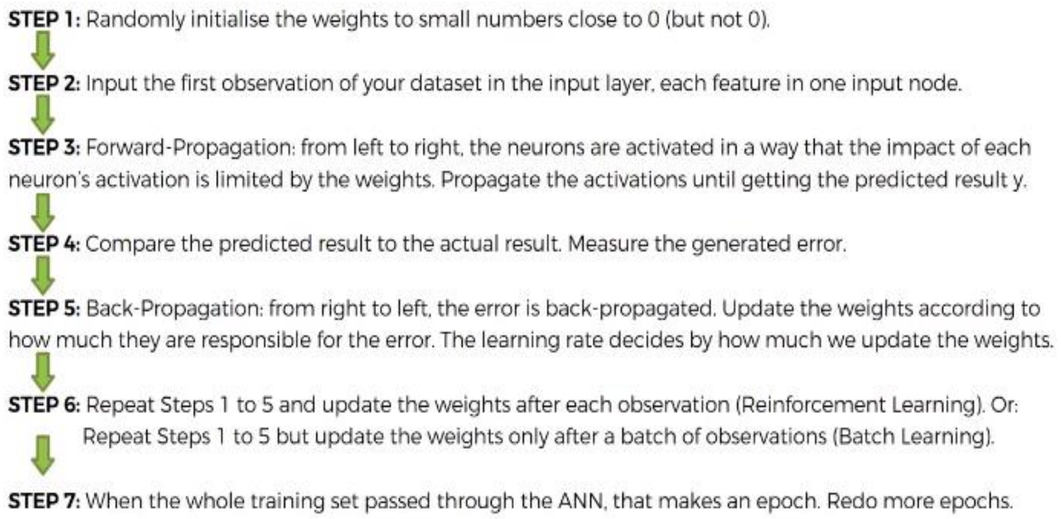
We apply rectifier function and in the output layer we apply sigmoid function



Backpropogation



Step by step method



STEP 1: Randomly initialise the weights to small numbers close to 0 (but not 0).

STEP 2: Input the first observation of your dataset in the input layer, each feature in one input node.

STEP 3: Forward-Propagation: from left to right, the neurons are activated in a way that the impact of each neuron's activation is limited by the weights. Propagate the activations until getting the predicted result y.

STEP 4: Compare the predicted result to the actual result. Measure the generated error.

STEP 5: Back-Propagation: from right to left, the error is back-propagated. Update the weights according to how much they are responsible for the error. The learning rate decides by how much we update the weights.

STEP 6: Repeat Steps 1 to 5 and update the weights after each observation (Reinforcement Learning). Or: Repeat Steps 1 to 5 but update the weights only after a batch of observations (Batch Learning).

STEP 7: When the whole training set passed through the ANN, that makes an epoch. Redo more epochs.

To create a neural network model

2 ways by designing graph or by using Sequential class.

```
import keras

#This is needed to create neural network model

from keras.models import Sequential

#this is used to add layers in the network

from keras.layers import Dense


# Initialising the ANN

classifier = Sequential()
```

Steps

Now we need to add layer in the model.

# Adding the input layer and the first hidden layer

classifier.add(Dense(output_dim = 6, init = 'uniform', activation = 'relu', input_dim = 11))

Following parameters are used in dense function

1. Output dimensions.
   The value of o/p dimensions can be decided based on experience we find the value.
   Usually we may use different parameters and models with k-fold-cross validation and find out optimum value. But for the beginning by experience we use value which id nearby average value of (11 independent variable+1 depenedent variable)/2=6
   So we will use 6). This output_dim also specify how many nodes we are adding in the hidden layer.
2. Init- randomly initialize the weight. We will use uniform function which will initialize weights uniformly near zero but not zero.
3. Activation function – relu
4. Input_dim – is compulsory here as this is the first node we have added.

We will add one more layer. As in deep learning number of hidden layers should be more than 1. # Adding the second hidden layer

classifier.add(Dense(output_dim = 6, init = 'uniform', activation = 'relu'))

To add output layer which is last layer of our data

# Adding the output layer

classifier.add(Dense(output_dim = 1, init = 'uniform', activation = 'sigmoid'))

since we have only one independent variable. So output_dim=1.

But if o/p variable is categorical and has more than one category then output_dim=number of different values in categorical variable and activation function need to be changed to softmax

This is sigmoid function which is used if number of categorical variables are more than 2.

Now we need to compile neural network

# Compiling the ANN

classifier.compile(optimizer = 'adam', loss = 'binary_crossentropy', metrics = ['accuracy'])

parameters

1. Optimizer—Its gradientdescent function used to optimize the weights. The better choice is stochastic gradientdescent, one of the popular algorithm for stochastic gradient descent is adam.
2. Loss- adam algorithm is based on loss function to calculate optimum weights.
   Some of the loss functions we have used

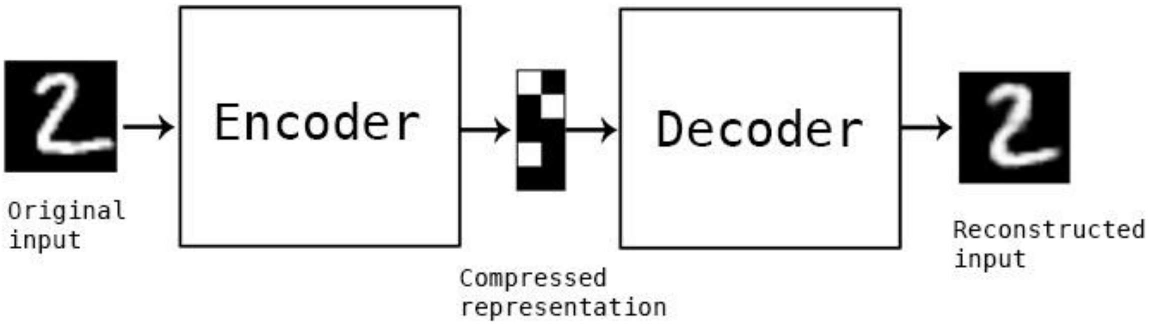
Building Autoencoders in Keras

We will see autoencoders, and we will cover code examples of the following models:

- a simple autoencoder based on a fully-connected layer
- a sparse autoencoder
- a deep fully-connected autoencoder
- a deep convolutional autoencoder
- an image denoising model
- a sequence-to-sequence autoencoder
- a variational autoencoder

Note: all code examples have been updated to the Keras 2.0 API on March 14, 2017. You will need Keras version 2.0.0 or higher to run them.

What are autoencoders?



"Autoencoding" is a data compression algorithm where the compression and decompression functions are 1) data-specific, 2) lossy, and 3) *learned automatically from examples* rather than engineered by a human. Additionally, in almost all contexts where the term "autoencoder" is used, the compression and decompression functions are implemented with neural networks.

1) Autoencoders are data-specific, which means that they will only be able to compress data similar to what they have been trained on. This is different from, say, the MPEG-2 Audio Layer III (MP3) compression algorithm, which only holds assumptions about "sound" in general, but not about specific types of sounds. An autoencoder trained on pictures of faces would do a rather poor job of compressing pictures of trees, because the features it would learn would be face-specific.

2) Autoencoders are lossy, which means that the decompressed outputs will be degraded compared to the original inputs (similar to MP3 or JPEG compression). This differs from lossless arithmetic compression.

3) Autoencoders are learned automatically from data examples, which is a useful property: it means that it is easy to train specialized instances of the algorithm that will perform well on a specific type of input. It doesn't require any new engineering, just appropriate training data.

To build an autoencoder, you need three things: an encoding function, a decoding function, and a distance function between the amount of information loss between the compressed representation of your data and the decompressed representation (i.e. a "loss" function). The encoder and decoder will be chosen to be parametric functions (typically neural networks), and to be differentiable with respect to the distance function, so the parameters of the encoding/decoding functions can be optimize to minimize the reconstruction loss, using Stochastic Gradient Descent. It's simple! And you don't even need to understand any of these words to start using autoencoders in practice.

Are they good at data compression?

Usually, not really. In picture compression for instance, it is pretty difficult to train an autoencoder that does a better job than a basic algorithm like JPEG, and typically the only way it can be achieved is by restricting yourself to a very specific type of picture (e.g. one for which JPEG does not do a good job). The fact that autoencoders are data-specific makes them generally impractical for real-world data compression problems: you can only use them on data that is similar to what they were trained on, and making them more general thus requires *lots* of training data. But future advances might change this, who knows.

What are autoencoders good for?

They are rarely used in practical applications. In 2012 they briefly found an application in greedy layer-wise pretraining for deep convolutional neural networks [1], but this quickly fell out of fashion as we started realizing that better random weight initialization schemes were sufficient for training deep networks from scratch. In 2014, batch normalization [2] started allowing for even deeper networks, and from late 2015 we could train arbitrarily deep networks from scratch using residual learning [3].

Today two interesting practical applications of autoencoders are **data denoising** (which we feature later in this post), and **dimensionality reduction for data visualization**. With appropriate dimensionality and sparsity constraints, autoencoders can learn data projections that are more interesting than PCA or other basic techniques.
For 2D visualization specifically, t-SNE (pronounced "tee-snee") is probably the best algorithm around, but it typically requires relatively low-dimensional data. So a good strategy for visualizing similarity relationships in high-dimensional data is to start by using an autoencoder to compress your data into a low-dimensional space (e.g. 32 dimensional), then use t-SNE for mapping the compressed data to a 2D plane. Note that a nice parametric implementation of t-SNE in Keras was developed by Kyle McDonald and is available on Github. Otherwise scikit-learn also has a simple and practical implementation.

So what's the big deal with autoencoders?

Their main claim to fame comes from being featured in many introductory machine learning classes available online. As a result, a lot of newcomers to the field absolutely love autoencoders and can't get enough of them. This is the reason why this tutorial exists!

Otherwise, one reason why they have attracted so much research and attention is because they have long been thought to be a potential avenue for solving the problem of unsupervised learning, i.e. the learning of useful representations without the need for labels. Then again, autoencoders are not a true unsupervised learning technique (which would imply a different learning process altogether), they are a *self-supervised* technique, a specific instance of *supervised learning* where the targets are generated from the input data. In order to get self-supervised models to learn interesting features, you have to come up with an interesting synthetic target and loss function, and that's where problems arise: merely learning to reconstruct your input in minute detail might not be the right choice here. At this point there is significant evidence that focusing on the reconstruction of a picture at the pixel level, for instance, is not conductive to learning interesting, abstract features of the kind that label-supervized learning induces (where targets are fairly abstract concepts "invented" by humans such as "dog", "car"...). In fact, one may argue that the best features in this regard are those that are the *worst* at exact input reconstruction while achieving high performance on the main task that you are interested in (classification, localization, etc).

In self-supervized learning applied to vision, a potentially fruitful alternative to autoencoder-style input reconstruction is the use of toy tasks such as jigsaw puzzle solving, or detail-context matching (being able to match high-resolution but small patches of pictures with low-resolution versions of the pictures they are extracted from). The following paper investigates jigsaw puzzle solving and makes for a very interesting read: Noroozi and Favaro (2016) Unsupervised Learning of Visual Representations by Solving Jigsaw Puzzles. Such tasks are providing the model with built-in assumptions about the input data which are missing in traditional autoencoders, such as *"visual macro-structure matters more than pixel-level details"*.

Figure from Noroozi and Favaro (2016)

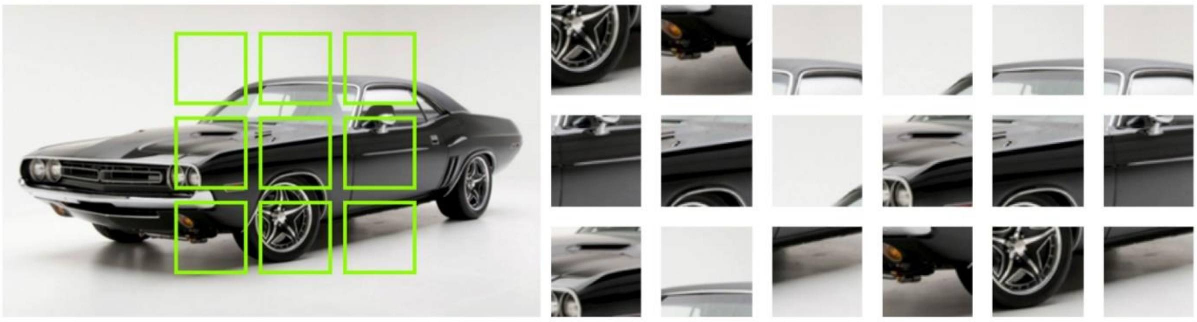


Fig. 1: What image representations do we learn by solving puzzles? Left: The image from which the tiles (marked with green lines) are extracted. Middle: A puzzle obtained by shuffling the tiles. Some tiles might be directly identifiable as object parts, but their identification is much more reliable once the correct ordering is found and the global figure emerges (Right).

Let's build the simplest possible autoencoder

We'll start simple, with a single fully-connected neural layer as encoder and as decoder:

```
from keras.layers import Input, Dense
from keras.models import Model

# this is the size of our encoded representations
encoding_dim = 32  # 32 floats -> compression of factor 24.5, assuming the input is
784 floats

# this is our input placeholder
input_img = Input(shape=(784,))
# "encoded" is the encoded representation of the input
encoded = Dense(encoding_dim, activation='relu')(input_img)
# "decoded" is the lossy reconstruction of the input
decoded = Dense(784, activation='sigmoid')(encoded)

# this model maps an input to its reconstruction
autoencoder = Model(input_img, decoded)
```

Let's also create a separate encoder model:

```
# this model maps an input to its encoded representation
encoder = Model(input_img, encoded)
```

As well as the decoder model:

```
# create a placeholder for an encoded (32-dimensional) input
encoded_input = Input(shape=(encoding_dim,))
```

```
# retrieve the last layer of the autoencoder model
decoder_layer = autoencoder.layers[-1]
# create the decoder model
decoder = Model(encoded_input, decoder_layer(encoded_input))
```

Now let's train our autoencoder to reconstruct MNIST digits.

First, we'll configure our model to use a per-pixel binary crossentropy loss, and the Adadelta optimizer:

```
autoencoder.compile(optimizer='adadelta', loss='binary_crossentropy')
```

Let's prepare our input data. We're using MNIST digits, and we're discarding the labels (since we're only interested in encoding/decoding the input images).

```
from keras.datasets import mnist
import numpy as np
(x_train, _), (x_test, _) = mnist.load_data()
```

We will normalize all values between 0 and 1 and we will flatten the 28x28 images into vectors of size 784.

```
x_train = x_train.astype('float32') / 255.
x_test = x_test.astype('float32') / 255.
x_train = x_train.reshape((len(x_train), np.prod(x_train.shape[1:])))
x_test = x_test.reshape((len(x_test), np.prod(x_test.shape[1:])))
print x_train.shape
print x_test.shape
```

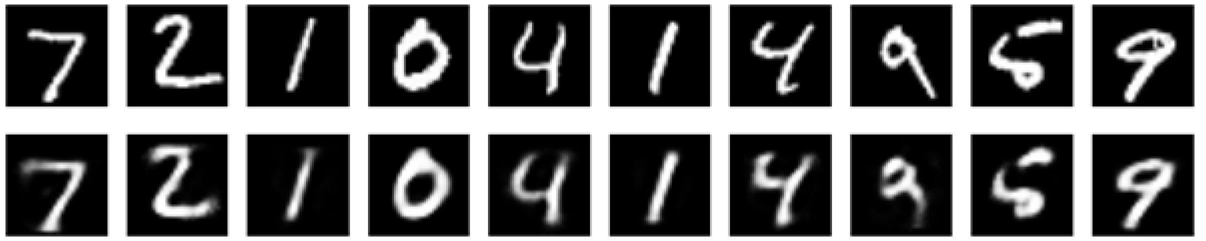
Now let's train our autoencoder for 50 epochs:

```
autoencoder.fit(x_train, x_train,
                epochs=50,
                batch_size=256,
                shuffle=True,
                validation_data=(x_test, x_test))
```

After 50 epochs, the autoencoder seems to reach a stable train/test loss value of about `0.11`. We can try to visualize the reconstructed inputs and the encoded representations. We will use Matplotlib.

```
# encode and decode some digits
# note that we take them from the *test* set
encoded_imgs = encoder.predict(x_test)
decoded_imgs = decoder.predict(encoded_imgs)
# use Matplotlib (don't ask)
import matplotlib.pyplot as plt

n = 10  # how many digits we will display
plt.figure(figsize=(20, 4))
for i in range(n):
    # display original
    ax = plt.subplot(2, n, i + 1)
    plt.imshow(x_test[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)

    # display reconstruction
    ax = plt.subplot(2, n, i + 1 + n)
    plt.imshow(decoded_imgs[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
plt.show()
```

Here's what we get. The top row is the original digits, and the bottom row is the reconstructed digits. We are losing quite a bit of detail with this basic approach.



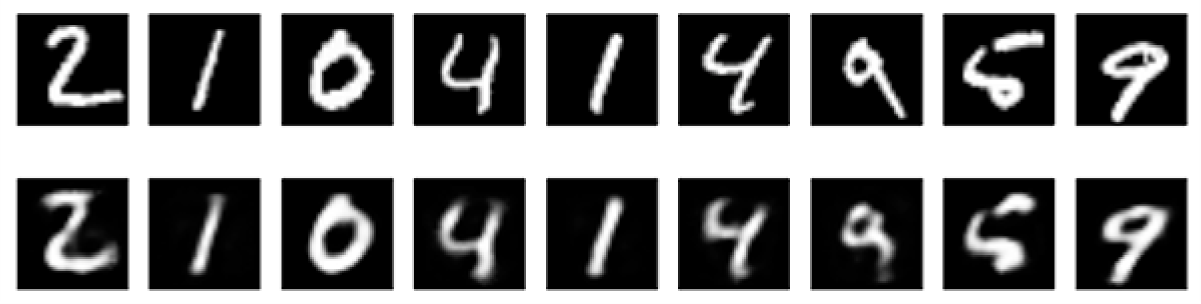
Adding a sparsity constraint on the encoded representations

In the previous example, the representations were only constrained by the size of the hidden layer (32). In such a situation, what typically happens is that the hidden layer is learning an approximation of PCA (principal component analysis). But another way to constrain the representations to be compact is to add a sparsity contraint on the activity of the hidden representations, so fewer units would "fire" at a given time. In Keras, this can be done by adding an `activity_regularizer` to our `Dense` layer:

```
from keras import regularizers

encoding_dim = 32

input_img = Input(shape=(784,))
# add a Dense layer with a L1 activity regularizer
encoded = Dense(encoding_dim, activation='relu',
                activity_regularizer=regularizers.l1(10e-5))(input_img)
decoded = Dense(784, activation='sigmoid')(encoded)

autoencoder = Model(input_img, decoded)
```

Let's train this model for 100 epochs (with the added regularization the model is less likely to overfit and can be trained longer). The models ends with a train loss of `0.11` and test loss of `0.10`. The difference between the two is mostly due to the regularization term being added to the loss during training (worth about 0.01).

Here's a visualization of our new results:



They look pretty similar to the previous model, the only significant difference being the sparsity of the encoded representations. `encoded_imgs.mean()` yields a value `3.33` (over our 10,000 test images), whereas with the previous model the same quantity was `7.30`. So our new model yields encoded representations that are twice sparser.

Deep autoencoder

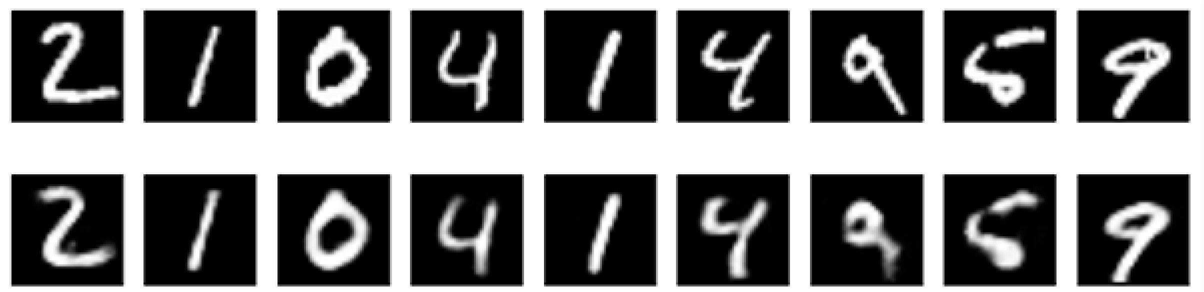
We do not have to limit ourselves to a single layer as encoder or decoder, we could instead use a stack of layers, such as:

```
input_img = Input(shape=(784,))
encoded = Dense(128, activation='relu')(input_img)
encoded = Dense(64, activation='relu')(encoded)
encoded = Dense(32, activation='relu')(encoded)

decoded = Dense(64, activation='relu')(encoded)
decoded = Dense(128, activation='relu')(decoded)
decoded = Dense(784, activation='sigmoid')(decoded)
```

Let's try this:

```
autoencoder = Model(input_img, decoded)
autoencoder.compile(optimizer='adadelta', loss='binary_crossentropy')

autoencoder.fit(x_train, x_train,
                epochs=100,
                batch_size=256,
                shuffle=True,
                validation_data=(x_test, x_test))
```

After 100 epochs, it reaches a train and test loss of ~0.097, a bit better than our previous models. Our reconstructed digits look a bit better too:



Convolutional autoencoder

Since our inputs are images, it makes sense to use convolutional neural networks (convnets) as encoders and decoders. In practical settings, autoencoders applied to images are always convolutional autoencoders --they simply perform much better.

Let's implement one. The encoder will consist in a stack of `Conv2D` and `MaxPooling2D` layers (max pooling being used for spatial down-sampling), while the decoder will consist in a stack of `Conv2D` and `UpSampling2D` layers.

```
from keras.layers import Input, Dense, Conv2D, MaxPooling2D, UpSampling2D
from keras.models import Model
from keras import backend as K

input_img = Input(shape=(28, 28, 1))  # adapt this if using `channels_first` image
data format

x = Conv2D(16, (3, 3), activation='relu', padding='same')(input_img)
x = MaxPooling2D((2, 2), padding='same')(x)
x = Conv2D(8, (3, 3), activation='relu', padding='same')(x)
x = MaxPooling2D((2, 2), padding='same')(x)
x = Conv2D(8, (3, 3), activation='relu', padding='same')(x)
encoded = MaxPooling2D((2, 2), padding='same')(x)

# at this point the representation is (4, 4, 8) i.e. 128-dimensional

x = Conv2D(8, (3, 3), activation='relu', padding='same')(encoded)
x = UpSampling2D((2, 2))(x)
x = Conv2D(8, (3, 3), activation='relu', padding='same')(x)
x = UpSampling2D((2, 2))(x)
x = Conv2D(16, (3, 3), activation='relu')(x)
x = UpSampling2D((2, 2))(x)
decoded = Conv2D(1, (3, 3), activation='sigmoid', padding='same')(x)

autoencoder = Model(input_img, decoded)
```

```
autoencoder.compile(optimizer='adadelta', loss='binary_crossentropy')
```

To train it, we will use the original MNIST digits with shape `(samples, 3, 28, 28)`, and we will just normalize pixel values between 0 and 1.

```
from keras.datasets import mnist
import numpy as np

(x_train, _), (x_test, _) = mnist.load_data()

x_train = x_train.astype('float32') / 255.
x_test = x_test.astype('float32') / 255.
x_train = np.reshape(x_train, (len(x_train), 28, 28, 1))  # adapt this if using `channels_first` image data format
x_test = np.reshape(x_test, (len(x_test), 28, 28, 1))  # adapt this if using `channels_first` image data format
```

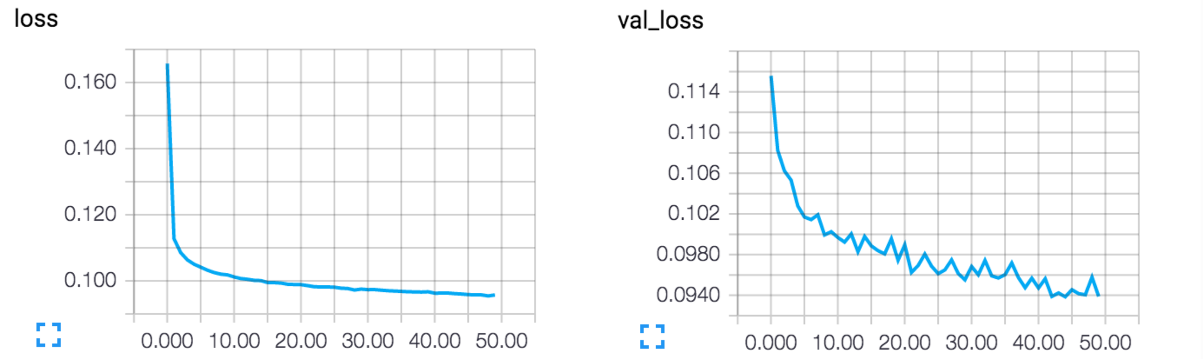
Let's train this model for 50 epochs. For the sake of demonstrating how to visualize the results of a model during training, we will be using the TensorFlow backend and the TensorBoard callback. First, let's open up a terminal and start a TensorBoard server that will read logs stored at `/tmp/autoencoder`.

```
tensorboard --logdir=/tmp/autoencoder
```

Then let's train our model. In the `callbacks` list we pass an instance of the `TensorBoard` callback. After every epoch, this callback will write logs to `/tmp/autoencoder`, which can be read by our TensorBoard server.

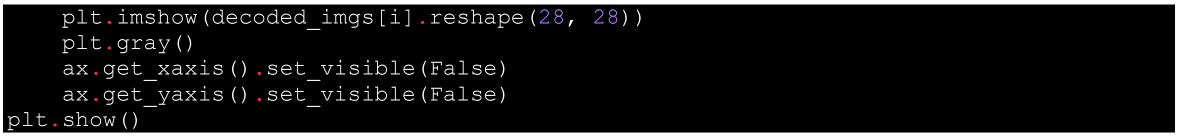
```
from keras.callbacks import TensorBoard

autoencoder.fit(x_train, x_train,
                epochs=50,
                batch_size=128,
                shuffle=True,
                validation_data=(x_test, x_test),
                callbacks=[TensorBoard(log_dir='/tmp/autoencoder')])
```

This allows us to monitor training in the TensorBoard web interface (by navighating to `http://0.0.0.0:6006`):

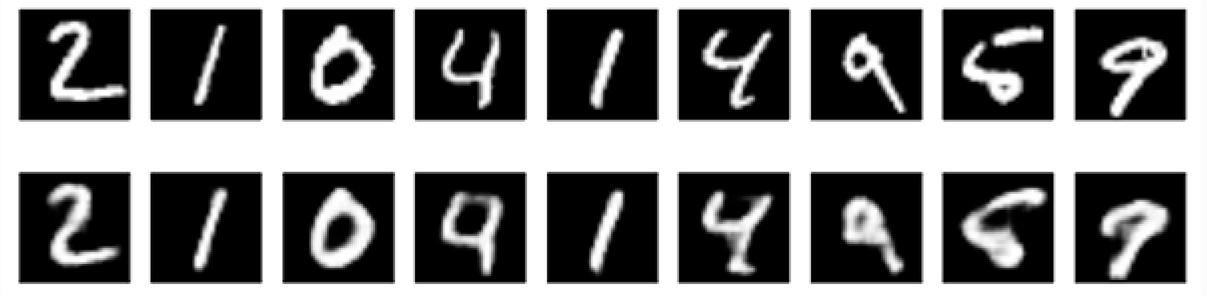


The model converges to a loss of 0.094, significantly better than our previous models (this is in large part due to the higher entropic capacity of the encoded representation, 128 dimensions vs. 32 previously). Let's take a look at the reconstructed digits:

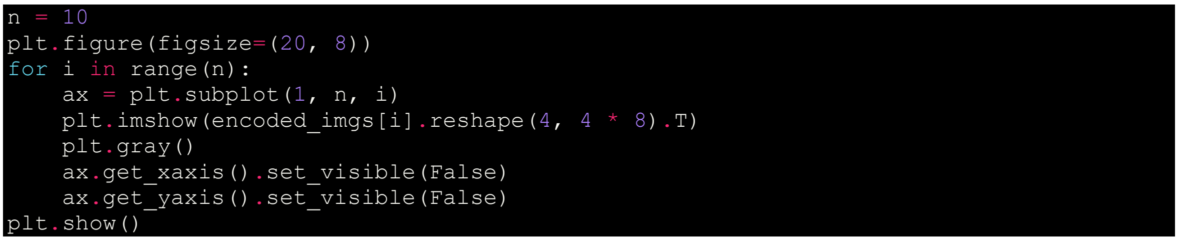
```
decoded_imgs = autoencoder.predict(x_test)

n = 10
plt.figure(figsize=(20, 4))
for i in range(n):
    # display original
    ax = plt.subplot(2, n, i)
    plt.imshow(x_test[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)

    # display reconstruction
    ax = plt.subplot(2, n, i + n)
```



```
    plt.imshow(decoded_imgs[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
plt.show()
```



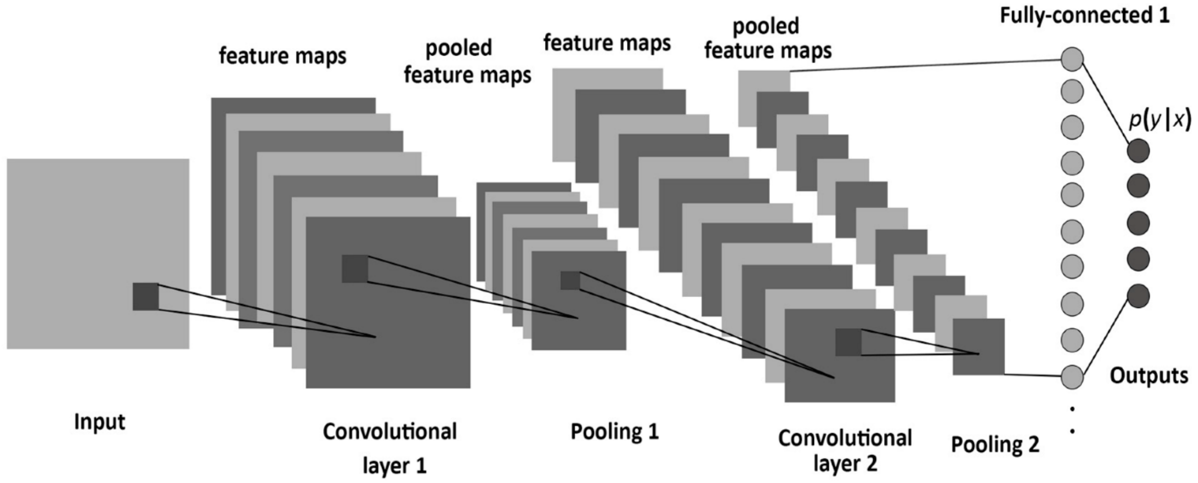
We can also have a look at the 128-dimensional encoded representations. These representations are 8x4x4, so we reshape them to 4x32 in order to be able to display them as grayscale images.



```
n = 10
plt.figure(figsize=(20, 8))
for i in range(n):
    ax = plt.subplot(1, n, i)
    plt.imshow(encoded_imgs[i].reshape(4, 4 * 8).T)
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
plt.show()
```

Convolutional Neural Networks (CNN)

Convolutional Neural Networks (CNN) is one of the variants of neural networks used heavily in the field of Computer Vision. It derives its name from the type of hidden layers it consists of. The hidden layers of a CNN typically consist of convolutional layers, pooling layers, fully connected layers, and normalization layers. Here it simply means that instead of using the normal activation functions defined above, convolution and pooling functions are used as activation functions.



To understand it in detail one needs to understand what convolution and pooling are. Both of these concepts are borrowed from the field of Computer Vision and are defined below.

Convolution*: Convolution operates on two signals (in 1D) or two images (in 2D): you can think of one as the "input" signal (or image), and the other (called the kernel) as a "filter" on the input image, producing an output image (so convolution takes two images as input and produces a third as output). [5]*

In layman terms it takes in an input signal and applies a filter over it, essentially multiplies the input signal with the kernel to get the modified signal. Mathematically, a convolution of two functions f and g is defined as

$$(f * g)(i) = \sum_{j=1}^{m} g(j) \cdot f(i - j + m/2)$$

which, is nothing but dot product of the input function and a kernel function.

In case of Image processing, it is easier to visualize a kernel as sliding over an entire image and thus changing the value of each pixel in the process.

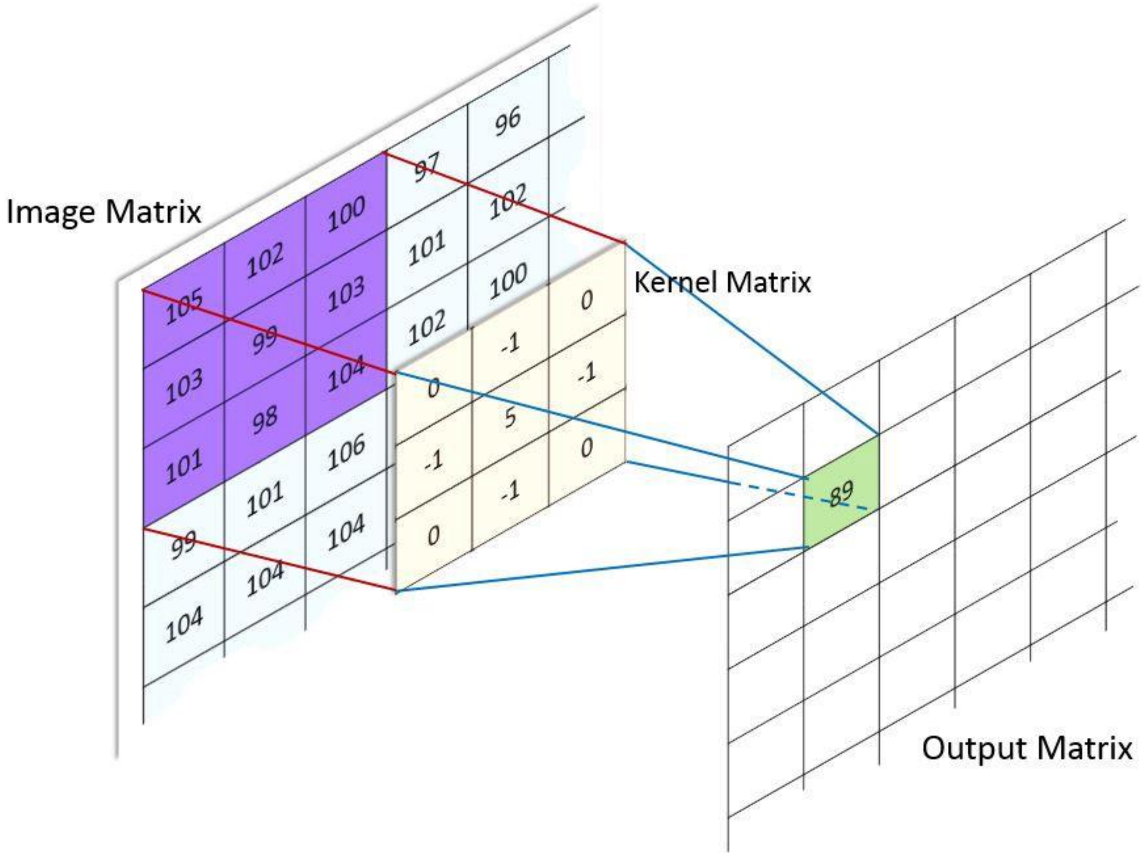
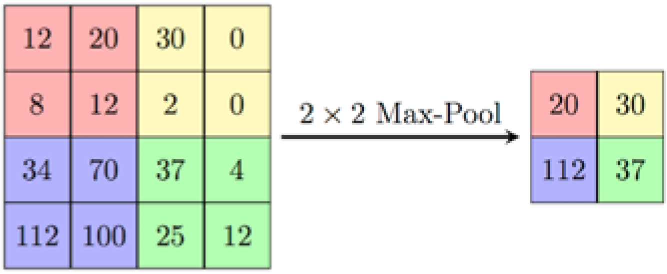


Image Credit: Machine Learning Guru [6]

Pooling: Pooling is a **sample-based discretization process**. The objective is to down-sample an input representation (image, hidden-layer output matrix, etc.), reducing its dimensionality and allowing for assumptions to be made about features contained in the sub-regions binned.

There are 2 main types of pooling commonly known as max and min pooling. As the name suggests max pooling is based on picking up the maximum value from the selected region and min pooling is based on picking up the minimum value from the selected region.



12	20	30	0
8	12	2	0
34	70	37	4
112	100	25	12

20	30
112	37

Thus as one can see A Convolutional Neural Network or CNN is basically a deep neural network which consists of hidden layers having convolution and pooling functions in addition to the activation function for introducing non-linearity.

A more detailed explanation can be found at

http://colah.github.io/posts/2014-07-Conv-Nets-Modular/

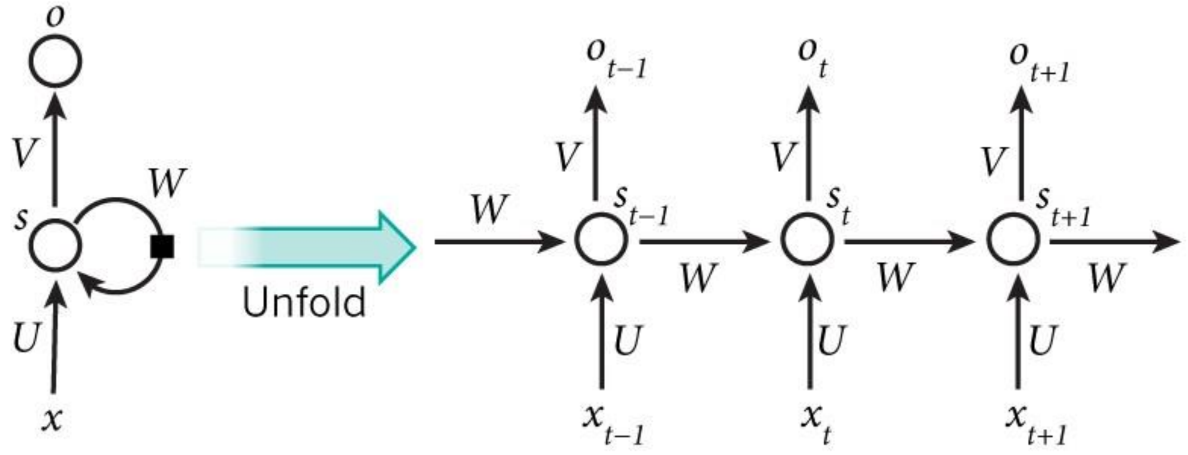
Recurrent Neural Networks (RNN)

Recurrent Neural Networks or RNN as they are called in short, are a very important variant of neural networks heavily used in Natural Language Processing. In a general neural network, an input is processed through a number of layers and an output is produced, with an assumption that two successive inputs are independent of each other.

This assumption is however not true in a number of real-life scenarios. For instance, if one wants to predict the price of a stock at a given time or wants to predict the next word in a sequence it is imperative that dependence on previous observations is considered.

RNNs are called recurrent *because they perform the same task for every element of a sequence, with the output being depended on the previous computations. Another way to think about RNNs is that they have a "memory" which captures information about what has been calculated so far. In theory,* *RNNs can make use of information in arbitrarily long sequences, but in practice, they are limited to looking back only a few steps. [7]*

Architecture wise, an RNN looks like this. One can imagine it as a multilayer neural network with each layer representing the observations at a certain time t.



RNN has shown to be hugely successful in natural language processing especially with their variant LSTM, which are able to look back longer than RNN. If you are interested in understanding LSTM, I would certainly encourage you to visit

http://colah.github.io/posts/2015-08-Understanding-LSTMs/

In this article I have tried to cover neural networks from a theoretical standpoint, starting from the most basic structure, a neuron and covering up to the most popular versions of neural networks. The aim of this write up was to make readers understand how a neural network is built from scratch, which all fields it is used and what are its most successful variations.

I understand that there are many other popular versions which I will try to cover in subsequent posts. Please feel free to suggest a topic if you want it to be covered earlier.

Reference

1. http://ufldl.stanford.edu/wiki/index.php/Neural_Networks

2. https://stats.stackexchange.com/questions/101560/tanh-activation-function-vs-sigmoid-activation-function

3. https://www.kaggle.com/dansbecker/rectified-linear-units-relu-in-deep-learning

4. http://ufldl.stanford.edu/tutorial/supervised/MultiLayerNeuralNetworks/

5. https://www.cs.cornell.edu/courses/cs1114/2013sp/sections/S06_convolution.pdf

6. http://machinelearninguru.com/computer_vision/basics/convolution/image_convolution_1.html

7. http://www.wildml.com/2015/09/recurrent-neural-networks-tutorial-part-1-introduction-to-rnns/