Learn OpenCV

Neural Networks : A 30,000 Feet View for Beginners

MAY 2, 2017 BY SATYA MALLICK (HTTPS://WWW.LEARNOPENCV.COM/AUTHOR/SPMALLICK/)

In this article, I am going to provide a 30,000 feet view of Neural Networks. The post is written for absolute beginners who are trying to dip their toes in Machine Learning and Deep Learning.

We will keep this short, sweet and math-free.

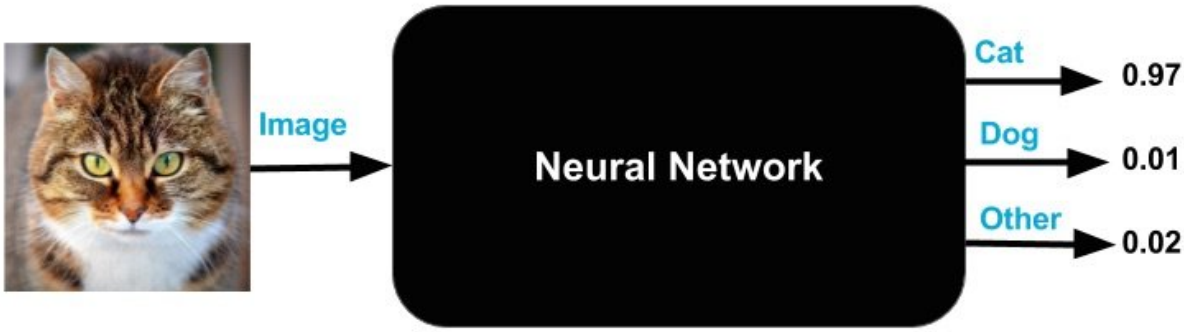
This post is part of the series on Deep Learning for Beginners, which consists of the following tutorials :

1. Neural Networks : A 30,000 Feet View for Beginners
2. Installation of Deep Learning frameworks (Tensorflow and Keras with CUDA support) (/installing-deep-learning-frameworks-on-ubuntu-with-cuda-support/)
3. Introduction to Keras (/deep-learning-using-keras-the-basics/)
4. Understanding Feedforward Neural Networks (/understanding-feedforward-neural-networks/)
5. Image Classification using Feedforward Neural Networks (/image-classification-using-feedforward-neural-network-in-keras/)
6. Image Recognition using Convolutional Neural Network (/image-classification-using-convolutional-neural-networks-in-keras/)
7. Understanding Activation Functions (/understanding-activation-functions-in-deep-learning/)
8. Understanding AutoEncoders using Tensorflow (/understanding-autoencoders-using-tensorflow-python/)
9. Image Classification using pre-trained models in Keras (/keras-tutorial-using-pre-trained-imagenet-models/)
10. Transfer Learning using pre-trained models in Keras (/keras-tutorial-transfer-learning-using-pre-trained-models/)
11. Fine-tuning pre-trained models in Keras (/keras-tutorial-fine-tuning-using-pre-trained-models)
12. More to come . . .

Neural Networks as Black Box

We will start by treating a Neural Networks as a magical black box. You don’t know what’s inside the black box. All you know is that it has one input and three outputs. The input is an image of any size, color, kind etc. The three outputs are numbers between 0 and 1. The outputs are labeled “Cat”, “Dog”,

and “Other”. The three numbers always add up to 1.



(/wp-content/uploads/2017/05/neural-network-as-blackbox.jpg)

Understanding the Neural Network Output

The magic it performs is very simple. If you input an image to the black box, it will output three numbers. A perfect neural network would output (1, 0, 0) for a cat, (0, 1, 0) for a dog and (0, 0, 1) for anything that is not a cat or a dog. In reality, though, even a well trained neural network will not give such clean results. For example, if you input the image of a cat, the number under the label “Cat” could say 0.97, the number under “Dog” could say 0.01 and the number under the label “Other” could say 0.02. The outputs can be interpreted as probabilities. This specific output means that the black box “thinks” there is a 97% chance that the input image is that of a cat and a small chance that it is either a dog or something it does not recognize. Note that the output numbers add up to 1.

This particular problem is called **image classification**; given an image, you can use the label with the highest probability to assign it a class (Cat, Dog, Other).

Understanding the Neural Network Input

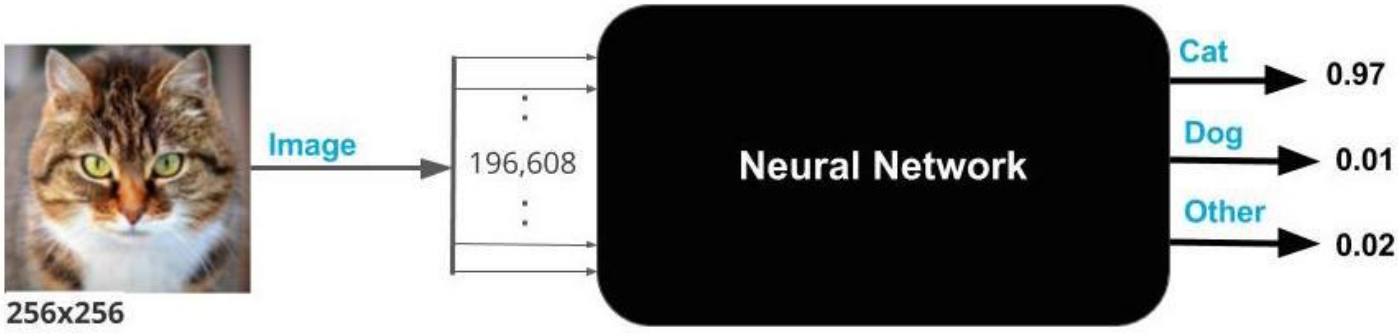
Now, you are a programmer and you are thinking you could use floats and doubles to represent the output of the Neural Network.

How do you input an image?

Images are just an array of numbers. A 256×256 image with three channels is simply an array of

256x256x3 = 196,608 numbers. Most libraries you use for reading the image will read a 256×256 color image into a continuous block of 196,608 numbers in memory.

With this new knowledge, we know the input is slightly more complicated. It is actually 196,608 numbers. Let us update our black box to reflect this new reality.



(/wp-content/uploads/2017/05/neural-network-as-blackbox-2.jpg)

I know what you are thinking. What about images that are not 256×256. Well, you can always convert any image to size 256×256 using the following steps.

1. **Non-Square aspect ratio**: If the input image is not square, you can resize the image so that the smaller dimension is 256. Then, crop 256×256 pixels from the center of the image.
2. **Grayscale image**: If the input image is not a color image, you can create a 3 channel image by copying the grayscale image into three channels.

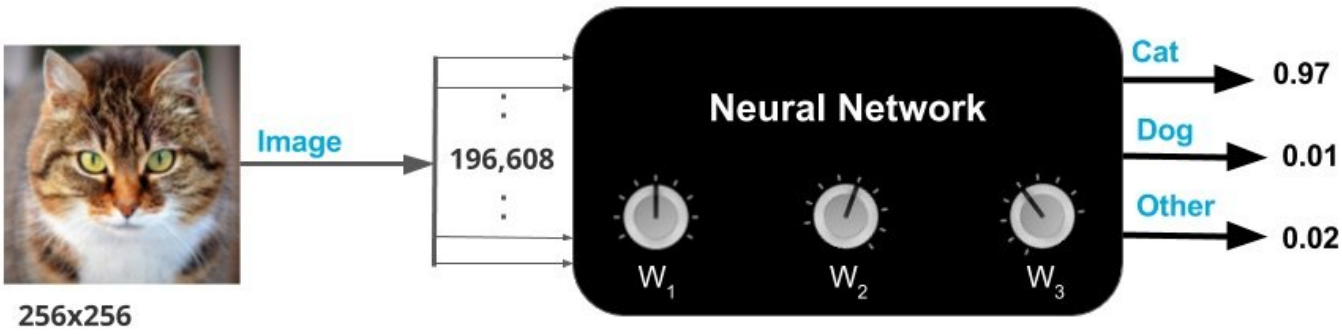
People use many different tricks to convert an image to a fixed size (e.g. a 256×256) image, but since I promised I will keep it simple, I won't go into those tricks. The important thing to note is that any image can be converted into a fixed size image even though we lose some information when we crop and resize an image to that fixed size.

What does it mean to train a Neural Network ?

The black box has knobs that can be used to "tune" it. In technical jargon, these knobs are called

weights. When the knobs are in the right position, the neural network gives the right output more often for different inputs.

Training the neural net simply means finding the right knob settings (or weights).



(/wp-content/uploads/2017/05/neural-network-as-blackbox-3.jpg)

How do you train a Neural Network?

If you had this magical black box but did not know the right knob settings, it would be a useless box.

The good news is that you can find the right knob settings by “training” the Neural Network.

Training a Neural Network is very similar to training a little child. You show the child a ball and tell her that it is a “ball”. When you do that many times with different kinds of balls, the child figures out that it is the shape of the ball that makes it a ball and not the color, texture or size. You then show the child an

egg and ask, “What is this?” She responds “Ball.” You correct them that it is not a ball, but an egg. When this process is repeated several times, the child is able to tell the difference between a ball and an egg.

To train a Neural Network, you show it several thousand examples of the classes (e.g. Cat, Dog, Other) you want it to learn. This kind of training is called **Supervised Learning** because you are providing the Neural Network an image of a class and explicitly telling it that it is an image from that class.

To train a neural network, we, therefore, need three things.

1. **Training data** : Thousands of images of each class and the expected output. For example, for all images of cats in this dataset, the expected output is (1, 0, 0).
2. **Cost function** : We need to know if the current setting is better than the previous knob setting. A cost function sums up the errors made by the neural network over all images in the training set. For example, a common cost function is called **sum of squared errors (SSE)**. If the expected output for an image is a cat, or (1, 0, 0) and the neural network outputs (0.37, 0.5, 0.13), the squared error made by the neural network on this particular image is $(1-0.37)^2 + (0-0.5)^2 + (0-0.13)^2 = 0.6638$. The total cost over all images is simply the sum of squared errors over all images. **The goal of training is to find the knob settings that will minimize the cost function**.
3. **How to update the knob settings**: Finally we need a way to update the knob settings based on the error we observe over all training images.

Training a neural network with a single knob

Let’s say we have a thousand images of cats, a thousand images of dogs, and a thousand images of random objects that are not cats or dogs. These three thousand images are our training set. If our neural network has not been trained, it will have some random knob settings and when you input these three thousand images, the output will be right only one in three times.

For the purpose of simplicity, let’s say our neural network has just one knob. Since we have just one knob, we could test a thousand different knob settings spanning the range of expected knob values and find the best knob setting that minimizes the cost function. This would complete our training.

However, the real world neural networks do not have a single knob. For example, VGG-Net, a popular neural network architecture has 138 million knobs!

Training a neural network with multiple knobs

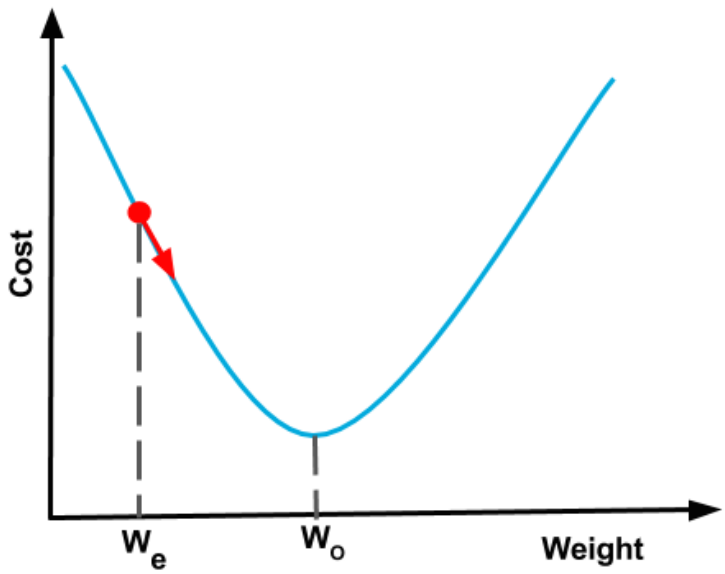
When we had just one knob, we could easily find the best setting by testing all (or a very large number of) possibilities. This quickly becomes unrealistic because even if we had just three knobs, we would have to test a billion settings. Imagine the number of possibilities with something as large as VGG-Net.

Needless to say a brute force search for the optimal knob settings is not feasible.

Fortunately, there is a way out. When the cost function is convex (i.e. shaped like a bowl), there is a principled way to iteratively find the best weight by a method called **Gradient Descent**

Gradient Descent

Let's go back to our Neural Network with just one knob and assume that our current estimate of the knob setting (or weight) is W_e. If our cost function is shaped like a bowl, we could find the slope of the cost function and move a step closer to the optimum knob setting W_o. This procedure is called **Gradient Descent** because we are moving down (descending) the curve based on the slope (gradient). When you reach the bottom of the bowl, the gradient or slope goes to zero and that completes your training. These bowl-shaped functions are technically called convex functions.



(/wp-content/uploads/2017/05/gradient-descent.png)

How do you come up with the first estimate? You can pick a random number.

Note: If you are using popular neural network architectures like GoogleNet or VGG-Net, you can use the weight trained on ImageNet instead of picking random initial weights to get much faster convergence.

Gradient Descent works similarly when there are multiple knobs. For example, when there are two knobs, the cost function is a bowl in 3D. If we place a ball on any part of this bowl, it will roll down to the bottom following the path of the maximum downward slope. This is exactly how gradient descent works. Also, note that if you let the ball roll down at full velocity, it will overshoot the bottom and take much more time to settle down at the bottom compared to a ball that is rolled down slowly in a more controlled manner. Similarly, while training a neural network, we use a parameter called the **learning rate** to control convergence of cost to its minimum.

When we have millions of knobs (weights), the shape of the cost function is a bowl in this higher dimensional space. Even though such a bowl is impossible to visualize, the concept of slope and Gradient Descent works just as well. Therefore, Gradient Descent allows us to converge to a solution thus making the problem tractable.

Backpropagation

There is one piece left in the puzzle. Given our current knob settings, how do we know the slope of the cost function?

First, let's remember that the cost function, and therefore its gradient depends on the difference between true output and the current output for all images in the training set. In other words, every image in the training set contributes to the final gradient calculation based on how badly the Neural Network performs on those images.

The algorithm used for estimating the gradient of the cost function is called **Backpropagation**. We will cover backpropagation in a future post and yes it does involve calculus. You would be surprised though that backpagation is simply repetitive application of the **chain rule** that you might have learned in high school.

Subscribe & Download Code

If you liked this article and would like to receive a free Computer Vision Resource (https://bigvisionllc.leadpages.net/leadbox/143948b73f72a2%3A173c9390c346dc/5649050225344512/) Guide, please subscribe (https://bigvisionllc.leadpages.net/leadbox/143948b73f72a2%3A173c9390c346dc/5649050225344512/).

In our newsletter, we share OpenCV tutorials and examples written in C++/Python, and Computer Vision and Machine Learning algorithms and news. You will also receive free access to all the code I have written for this blog.

Subscribe Now
(https://bigvisionllc.leadpages.net/leadbox/143948b73f72a2%3A173c9390c346dc/5649050225344512/)

Learn OpenCV

Deep learning using Keras – The Basics

SEPTEMBER 25, 2017 BY VIKAS GUPTA (HTTPS://WWW.LEARNOPENCV.COM/AUTHOR/VIKAS/)

```
import keras
Using TensorFlow backend.
```

(/wp-content/uploads/2017/09/import-keras.png)

This post is part of the series on Deep Learning for Beginners, which consists of the following tutorials :

1. Neural Networks : A 30,000 Feet View for Beginners (/neural-networks-a-30000-feet-view-for-beginners/)
2. Installation of Deep Learning frameworks (Tensorflow and Keras with CUDA support) (/installing-deep-learning-frameworks-on-ubuntu-with-cuda-support/)
3. Introduction to Keras
4. Understanding Feedforward Neural Networks (/understanding-feedforward-neural-networks/)
5. Image Classification using Feedforward Neural Networks (/image-classification-using-feedforward-neural-network-in-keras/)
6. Image Recognition using Convolutional Neural Network (/image-classification-using-convolutional-neural-networks-in-keras/)
7. Understanding Activation Functions (/understanding-activation-functions-in-deep-learning/)
8. Understanding AutoEncoders using Tensorflow (/understanding-autoencoders-using-tensorflow-python/)
9. Image Classification using pre-trained models in Keras (/keras-tutorial-using-pre-trained-imagenet-models/)
10. Transfer Learning using pre-trained models in Keras (/keras-tutorial-transfer-learning-using-pre-trained-models/)
11. Fine-tuning pre-trained models in Keras (/keras-tutorial-fine-tuning-using-pre-trained-models)
12. More to come . . .

1. Deep Learning Frameworks

Deep Learning is a branch of AI which uses Neural Networks for Machine Learning. In the recent years, it has shown dramatic improvements over traditional machine learning methods with applications in Computer Vision, Natural Language Processing, Robotics among many others. A very light introduction to Convolutional Neural Networks (a type of Neural Network) is covered in **this article** (/neural-networks-a-30000-feet-view-for-beginners/).

Deep Learning became a household name for AI engineers since 2012 when Alex Krizhevsky (https://scholar.google.com/citations?user=xegzhJcAAAAJ) and his team won the ImageNet challenge. **ImageNet** (http://image-net.org) is a computer vision competition in which the computer is required to correctly classify the image of an object into one of 1000 categories. The objects include different types of animals, plants, instruments, furniture, Vehicles to name a few.

This attracted a lot of attention from the Computer vision community and almost everyone started working on Neural Networks. But at that time, there were not many tools available to get you started in this new domain. A lot of effort has been put in by the community of researchers to create useful libraries making it easy to work in this emerging field. Some popular deep learning frameworks at present are **Tensorflow** (https://www.tensorflow.org/), **Theano** (http://deeplearning.net/software/theano/), **Caffe** (http://caffe.berkeleyvision.org/), **Pytorch** (http://pytorch.org/), **CNTK** (https://www.microsoft.com/en-us/cognitive-toolkit/), **MXNet** (https://mxnet.incubator.apache.org/), **Torch** (http://torch.ch/), **deeplearning4j** (https://deeplearning4j.org/), **Caffe2** (https://caffe2.ai/) among many others.

Keras is a high-level API, written in Python and capable of running on top of TensorFlow, Theano, or CNTK. The above deep learning libraries are written in a general way with a lot of functionalities. This can be overwhelming for a beginner who has limited knowledge in deep learning. Keras provides a simple and modular API to create and train Neural Networks, hiding most of the complicated details under the hood. This makes it easy to get you started on your Deep Learning journey.

Once you get familiar with the main concepts and want to dig deeper and take control of the process, you may choose to work with any of the above frameworks.

2. Keras installation and configuration

As mentioned above, Keras is a high-level API that uses deep learning libraries like Theano or Tensorflow as the backend. These libraries, in turn, talk to the hardware via lower level libraries. For example, if you run the program on a CPU, Tensorflow or Theano use BLAS libraries. On the other

hand, when you run on a GPU, they use CUDA and cuDNN libraries.

If you are setting up a new system, you might want to look at **this article** (/installing-deep-learning-frameworks-on-ubuntu-with-cuda-support/) for installing the most common deep learning frameworks. We will mention only the Keras specific part here.

It is advisable to install everything on virtual environments. If virtual environment is not installed on the system, then check step 5 of the above article.

We will install Theano and Tensorflow as backend libraries for Keras, along with some more libraries which are useful for working with data (h5py) and visualization (pydot, graphviz and matplotlib).

Create virtual environment

Create the virtual environment for either python 2 or python 3, whichever you want to use.

```
mkvirtualenv virtual-py2 -p python2
# Activate the virtual environment
workon virtual-py2
```

Or

```
mkvirtualenv virtual-py3 -p python3
# Activate the virtual environment
workon virtual-py3
```

Install libraries

```
pip install Theano
#If using only CPU
pip install tensorflow
#If using GPU
pip install tensorflow-gpu
pip install keras
pip install h5py pydot matplotlib
```

Also install graphviz

```
#For Ubuntu
sudo apt-get install graphviz

#For MacOs
brew install graphviz
```

Configure Keras

By default, Keras is configured to use Tensorflow as the backend since it is the most popular choice. However, If you want to change it to Theano, open the file ~/.keras/keras.json which looks as shown:

```
{
```

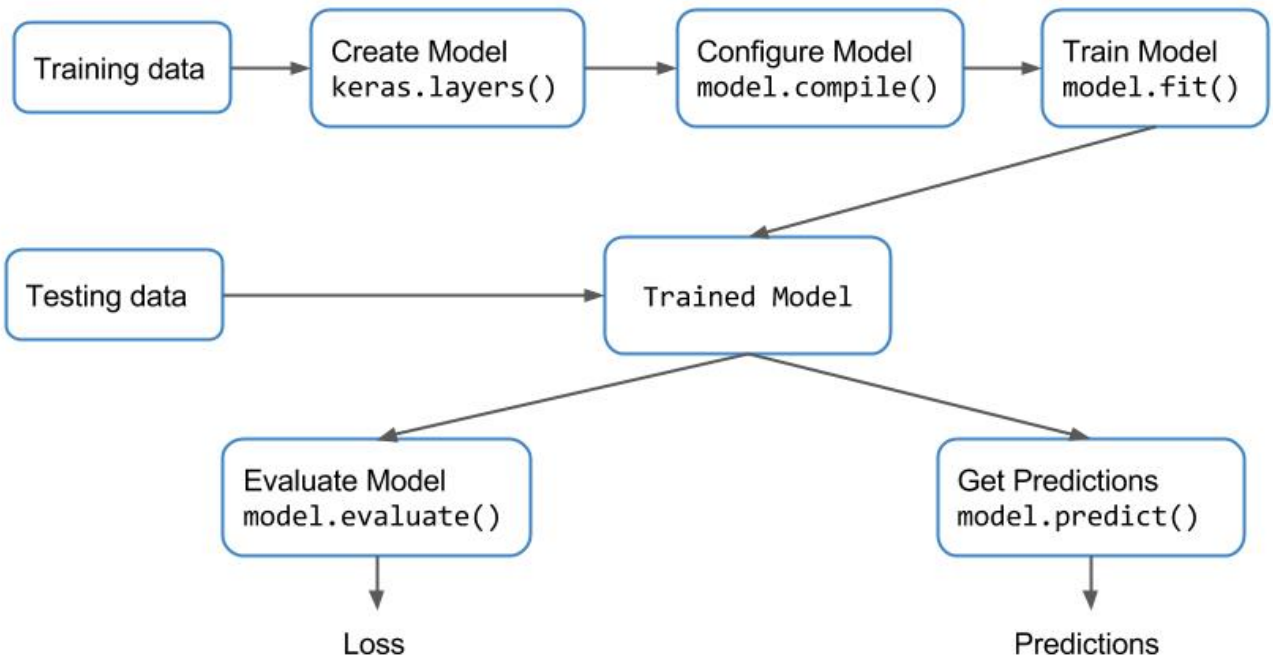
```
{
    "epsilon": 1e-07,
    "floatx": "float32",
    "image_data_format": "channels_last",
    "backend": "tensorflow"
}
```

and change it to

```
{
    "epsilon": 1e-07,
    "floatx": "float32",
    "image_data_format": "channels_first",
    "backend": "theano"
}
```

3. Keras Workflow

Keras provides a very simple workflow for training and evaluating the models. It is described with the following diagram



(/wp-content/uploads/2017/09/keras-workflow.jpg)

Basically, we are creating the model and training it using the training data. Once the model is trained, we take the model to perform inference on test data. Let us understand the function of each of the blocks.

3.1. Keras Layers

Layers can be thought of as the building blocks of a Neural Network. They process the input data and produce different outputs, depending on the type of layer, which are then used by the layers which are connected to them. We will cover the details of every layer in future posts.
Keras provides a number of core layers which include

- Dense layers, also called fully connected layer, since, each node in the input is connected to every node in the output,
- Activation layer which includes activation functions like ReLU, tanh, sigmoid among others,
- Dropout layer – used for regularization during training,
- Flatten, Reshape, etc.

Apart from these core layers, some important layers are

- Convolution layers – used for performing convolution,
- Pooling layers – used for down sampling,
- Recurrent layers,
- Locally-connected, normalization, etc.

We can use the code snippet to import the respective layer

```
1 from keras.layers import Dense, Activation, Conv2D, MaxPooling2D
```

3.2. Keras Models

Keras provides two ways to define a model:

- **Sequential** (https://keras.io/getting-started/sequential-model-guide/), used for stacking up layers – Most commonly used.
- **Functional API** (https://keras.io/getting-started/functional-api-guide/), used for designing complex model architectures like models with multiple-outputs, shared layers etc.

```
1 from keras.models import Sequential
```

For creating a Sequential model, we can either pass the list of layers as an argument to the constructor or add the layers sequentially using the model.add() function.
For example, both the code snippets for creating a model with a single dense layer with 10 outputs are

equivalent.

```
from keras.models import Sequential
from keras.layers import Dense, Activation

model = Sequential([Dense(10, input_shape=(nFeatures,)),
                    Activation('linear') ])
```

```
from keras.models import Sequential
from keras.layers import Dense, Activation

model = Sequential()
model.add(Dense(10, input_shape=(nFeatures,)))
model.add(Activation('linear'))
```

An important thing to note in the model definition is that we need to specify the input shape for the first layer. This is done in the above snippet using the input_shape parameter passed along with the first Dense layer. The shapes of other layers are inferred by the compiler.

3.3. Configuring the training process

Once the model is ready, we need to configure the learning process. This means

- Specify an Optimizer which determines how the network weights are updated
- Specify the type of cost function or loss function.
- Specify the metrics you want to evaluate during training and testing.
- Create the model graph using the backend.
- Any other advanced configuration.

This is done in Keras using the model.compile() function. The code snippet shows the usage.

```
model.compile(optimizer='rmsprop', loss='mse', metrics=['mse', 'mae'])
```

The mandatory parameters to be specified are the optimizer and the loss function.

Optimizers

Keras provides a lot of optimizers to choose from, which include

- Stochastic Gradient Descent (SGD),
- Adam,
- RMSprop,
- AdaGrad,
- AdaDelta, etc.

RMSprop is a good choice of optimizer for most problems.

Loss functions

In a supervised learning problem, we have to find the error between the actual values and the predicted value. There can be different metrics which can be used to evaluate this error. This metric is often called loss function or cost function or objective function. There can be more than one loss function depending on what you are doing with the error. In general, we use

- binary-cross-entropy for a binary classification problem,
- categorical-cross-entropy for a multi-class classification problem,
- mean-squared-error for a regression problem and so on.

3.4. Training

Once the model is configured, we can start the training process. This can be done using the model.fit() function in Keras. The usage is described below.

```
model.fit(trainFeatures, trainLabels, batch_size=4, epochs = 100)
```

We just need to specify the training data, batch size and number of epochs. Keras automatically figures out how to pass the data iteratively to the optimizer for the number of epochs specified. The rest of the information was already given to the optimizer in the previous step.

3.5. Evaluating the model

Once the model is trained, we need to check the accuracy on unseen test data. This can be done in two ways in Keras.

- model.evaluate() – It finds the loss and metrics specified in the model.compile() step. It takes both the test data and labels as input and gives a quantitative measure of the accuracy. It can also be used to perform cross-validation and further finetune the parameters to get the best model.
- model.predict() – It finds the output for the given test data. It is useful for checking the outputs qualitatively.

Now, let's see how to use keras models and layers to create a simple Neural Network.

4. Linear Regression Example

We will learn how to create a simple network with a single layer to perform **linear regression** (https://en.wikipedia.org/wiki/Linear_regression). We will use the **Boston Housing dataset** (https://keras.io/datasets/) available in Keras as an example. Samples contain 13 attributes of houses at different locations around the Boston suburbs in the late 1970s. Targets are the median values of the houses at a location (in k$). With the 13 features, we have to train the model which would predict the price of the house in the test data.

4.1. Training

We use the Sequential model to create the network graph. Then we add a Dense layer with the number of inputs equal to the number of features in the data and a single output. Then we follow the workflow as explained in the previous section. We compile the model and train it using the fit command. Finally, we use the model.summary() function to check the configuration of the model. All keras datasets come with a load_data() function which returns tuples of training and testing data as shown in the code.

```
from keras.models import Sequential
from keras.layers import Dense
from keras.datasets import boston_housing

(X_train, Y_train), (X_test, Y_test) = boston_housing.load_data()

nFeatures = X_train.shape[1]

model = Sequential()
model.add(Dense(1, input_shape=(nFeatures,), activation='linear'))

model.compile(optimizer='rmsprop', loss='mse', metrics=['mse', 'mae'])

model.fit(X_train, Y_train, batch_size=4, epochs=1000)

model.summary()
```

The output of model.summary() is given below. It shows 14 parameters – 13 parameters for the weights and 1 for the bias.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
```

```
dense_1 (Dense)        (None, 1)             14
=================================================================
Total params: 14
Trainable params: 14
Non-trainable params: 0
```

4.2. Inference

After the model has been trained, we want to do inference on the test data. We can find the loss on the test data using the model.evaluate() function. We get the predictions on test data using the model.predict() function. Here we compare the ground truth values with the predictions from our model for the first 5 test samples.

```
model.evaluate(X_test, Y_test, verbose=True)

Y_pred = model.predict(X_test)

print Y_test[:5]
print Y_pred[:5,0]
```

The output is

```
[  7.2  18.8  19.   27.   22.2]
[  7.2  18.26  21.38  29.28  23.72]
```

It can be seen that the predictions follow the ground truth values, but there are some errors in the predictions.

References

https://keras.io (https://keras.io)

Subscribe & Download Code

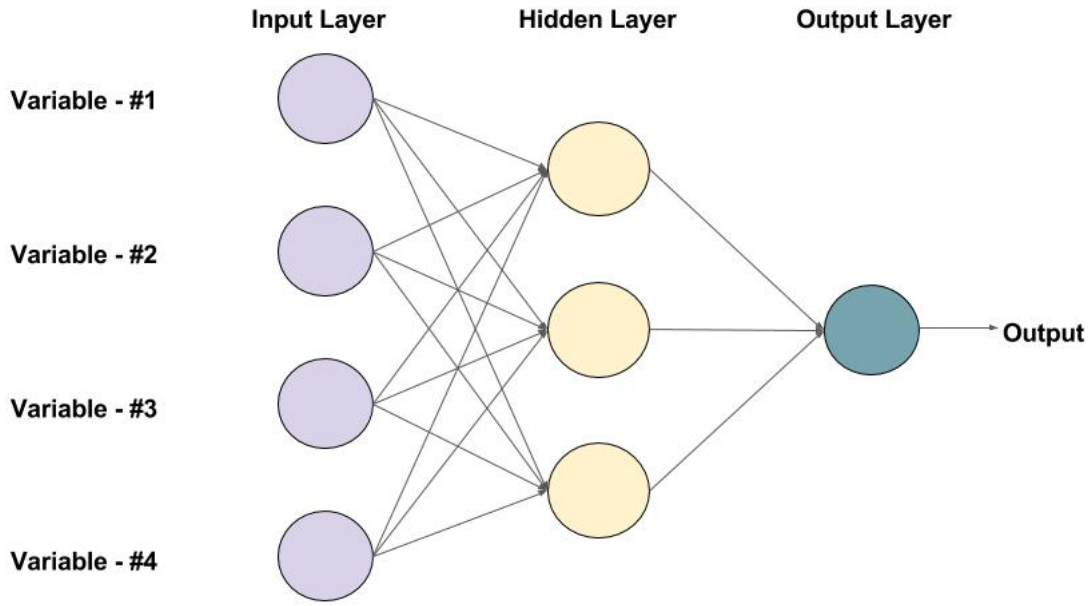
If you liked this article and would like to download code (C++ and Python) and example images used in all the posts of this blog, please subscribe (https://bigvisionllc.leadpages.net/leadbox/143948b73f72a2%3A173c9390c346dc/5649050225344512/) to our newsletter. You will also receive a free Computer Vision Resource (https://bigvisionllc.leadpages.net/leadbox/143948b73f72a2%3A173c9390c346dc/5649050225344512/) Guide. In our newsletter, we share OpenCV tutorials and examples written in C++/Python, and Computer Vision and Machine Learning algorithms and news.

Subscribe Now
(https://bigvisionllc.leadpages.net/leadbox/143948b73f72a2%3A173c9390c346dc/5649050225344512/)

Learn OpenCV

Understanding Feedforward Neural Networks

OCTOBER 9, 2017 BY VIKAS GUPTA (HTTPS://WWW.LEARNOPENCV.COM/AUTHOR/VIKAS/)



An example of a Feed-forward Neural Network with one hidden layer (with 3 neurons)

(/wp-content/uploads/2017/10/mlp-diagram.jpg)

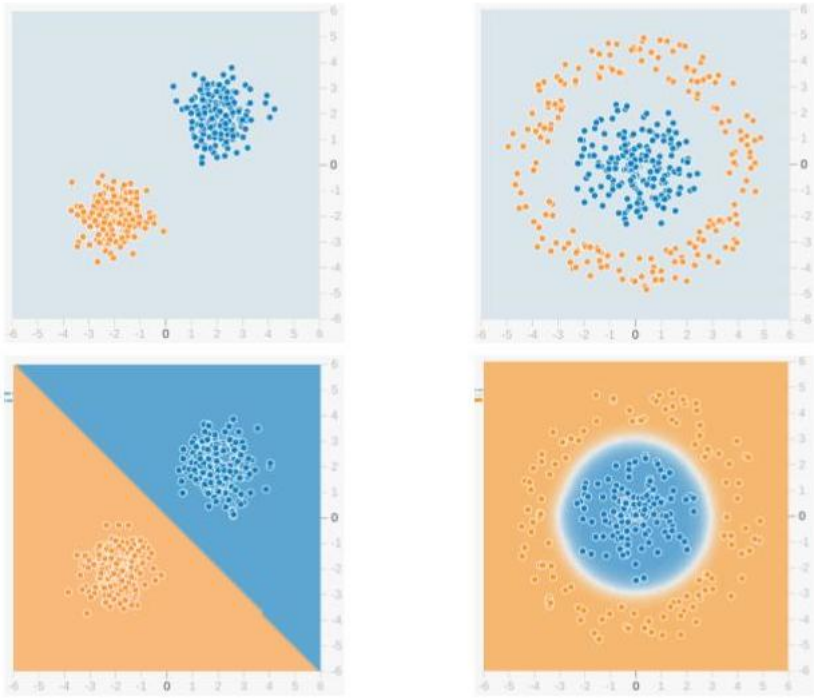
This post is part of the series on Deep Learning for Beginners, which consists of the following tutorials :

1. Neural Networks : A 30,000 Feet View for Beginners (/neural-networks-a-30000-feet-view-for-beginners/)
2. Installation of Deep Learning frameworks (Tensorflow and Keras with CUDA support) (/installing-deep-learning-frameworks-on-ubuntu-with-cuda-support/)
3. Introduction to Keras (/deep-learning-using-keras-the-basics/)
4. Understanding Feedforward Neural Networks
5. Image Classification using Feedforward Neural Networks (/image-classification-using-feedforward-neural-network-in-keras/)
6. Image Recognition using Convolutional Neural Network (/image-classification-using-convolutional-neural-networks-in-keras/)
7. Understanding Activation Functions (/understanding-activation-functions-in-deep-learning/)
8. Understanding AutoEncoders using Tensorflow (/understanding-autoencoders-using-tensorflow-python/)
9. Image Classification using pre-trained models in Keras (/keras-tutorial-using-pre-trained-imagenet-models/)
10. Transfer Learning using pre-trained models in Keras (/keras-tutorial-transfer-learning-using-pre-trained-models/)
11. Fine-tuning pre-trained models in Keras (/keras-tutorial-fine-tuning-using-pre-trained-models)
12. More to come . . .

In this article, we will learn about feedforward Neural Networks, also known as Deep feedforward Networks or Multi-layer Perceptrons. They form the basis of many important Neural Networks being used in the recent times, such as Convolutional Neural Networks (used extensively in computer vision applications), Recurrent Neural Networks (widely used in Natural

language understanding and sequence learning) and so on. We will try to understand the important concepts involved in an intuitive and interactive way, without going into the mathematics involved. If you are interested in diving into deep learning but don't have much background in statistics and machine learning, then this article is a perfect starting point.

We will use the feedforward network to solve a binary classification (https://en.wikipedia.org/wiki/Binary_classification) problem. In Machine Learning, Classification (https://en.wikipedia.org/wiki/Classification) is a type of Supervised Learning method, where the task is to divide the data samples into predefined groups by a Decision Function. When there are only two groups, it is called Binary Classification. The figure given below shows an example. The points in blue belong to one group (or class) and orange points belong to the other. The imaginary line(s) which separate the groups are called Decision Boundaries. The decision function is learned from a set of labeled samples, which is called Training Data and the process of learning the decision function is called Training.

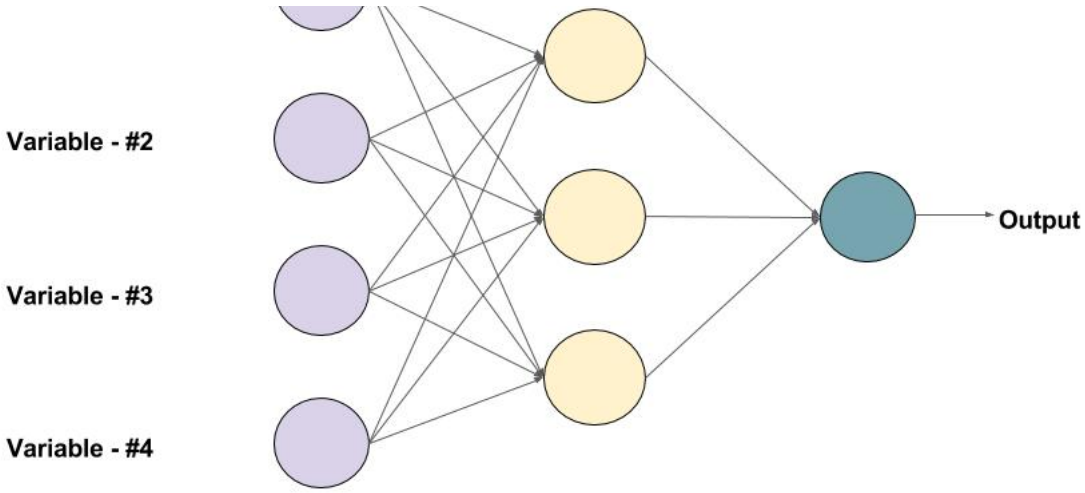


(/wp-content/uploads/2017/10/sample-data-mlp.jpg)

In the above example, the top row shows two different data distributions and the bottom row shows the decision boundary. The left image shows an example of data which is **Linearly Separable** (https://en.wikipedia.org/wiki/Linear_separability). This means that a linear boundary (e.g. a straight line) is enough to separate the data into groups. On the other hand, the image on the right shows an example of data which is not linearly separable. The decision boundary, in this case, has to be circular or polygonal as shown in the figure.

1. Understanding the Neural Network Jargon

Given below is an example of a feedforward Neural Network. It is a directed acyclic Graph which means that there are no feedback connections or loops in the network. It has an input layer, an output layer, and a hidden layer. In general, there can be multiple hidden layers. Each node in the layer is a Neuron, which can be thought of as the basic processing unit of a Neural Network.

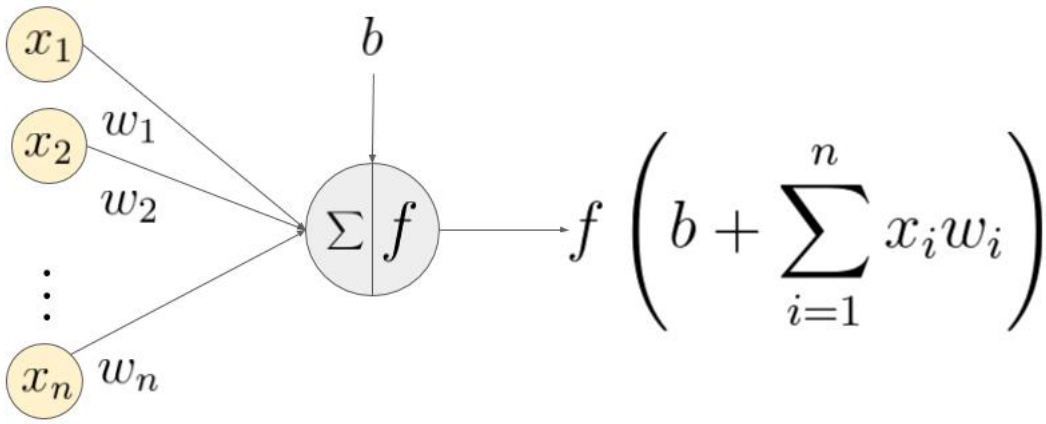


An example of a Feed-forward Neural Network with one hidden layer (with 3 neurons)

(/wp-content/uploads/2017/10/mlp-diagram.jpg)

1.1. What is a Neuron?

An Artifical Neuron is the basic unit of a neural network. A schematic diagram of a neuron is given below.



An example of a neuron showing the input (x_1 - x_n), their corresponding weights (w_1 - w_n), a bias (b) and the activation function f applied to the weighted sum of the inputs.

(/wp-content/uploads/2017/10/neuron-diagram.jpg)

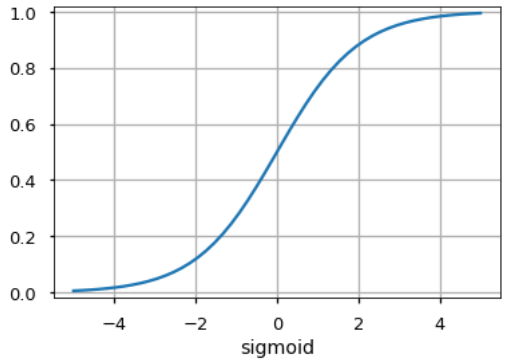
As seen above, It works in two steps – It calculates the weighted sum of its inputs and then applies an activation function to normalize the sum. The activation functions can be linear or nonlinear. Also, there are weights associated with each input of a neuron. These are the parameters which the network has to learn during the training phase.

1.2. Activation Functions

The activation function is used as a decision making body at the output of a neuron. The neuron learns Linear or Non-linear

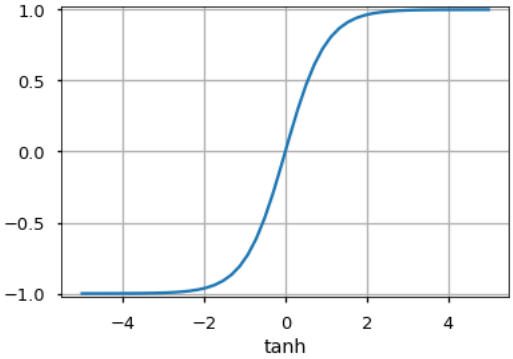
decision boundaries based on the activation function. It also has a normalizing effect on the neuron output which prevents the output of neurons after several layers to become very large, due to the cascading effect. There are three most widely used activation functions

- **Sigmoid** (https://en.wikipedia.org/wiki/Sigmoid_function)
  It maps the input (x axis) to values between 0 and 1.



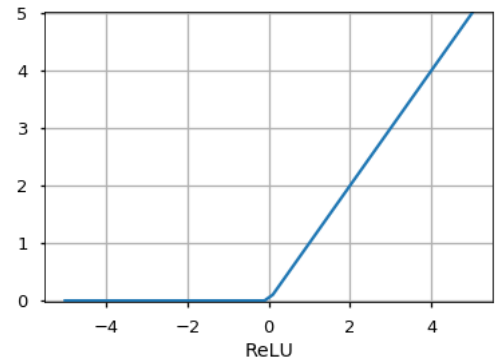
(/wp-content/uploads/2017/10/sigmoid.png)

- **Tanh**
  It is similar to the sigmoid function butmaps the input to values between -1 and 1.



(/wp-content/uploads/2017/10/tanh.png)

- **Rectified Linear Unit (ReLU)** (https://en.wikipedia.org/wiki/Rectifier_(neural_networks))
  It allows only positive values to pass through it. The negative values are mapped to zero.



(/wp-content/uploads/2017/10/relu.png)

There are other functions like the **Unit Step function, leaky ReLU, Noisy ReLU, Exponential LU** etc which have their own

merits and demerits.

1.3. Input Layer

This is the first layer of a neural network. It is used to provide the input data or features to the network.

1.4. Output Layer

This is the layer which gives out the predictions. The activation function to be used in this layer is different for different problems. For a binary classification problem, we want the output to be either 0 or 1. Thus, a sigmoid activation function is used. For a Multiclass classification problem, a **Softmax** (https://en.wikipedia.org/wiki/Softmax_function) (think of it as a generalization of sigmoid to multiple classes) is used. For a regression problem, where the output is not a predefined category, we can simply use a linear unit.

1.5. Hidden Layer

A feedforward network applies a series of functions to the input. By having multiple hidden layers, we can compute complex functions by cascading simpler functions. Suppose, we want to compute the 7th power of a number, but want to keep things simple (as they are easy to understand and implement). You can use simpler powers like square and cube to calculate the higher order functions. Similarly, you can compute highly complex functions by this cascading effect. The most widely used hidden unit is the one which uses a Rectified Linear Unit (ReLU) as the activation function.

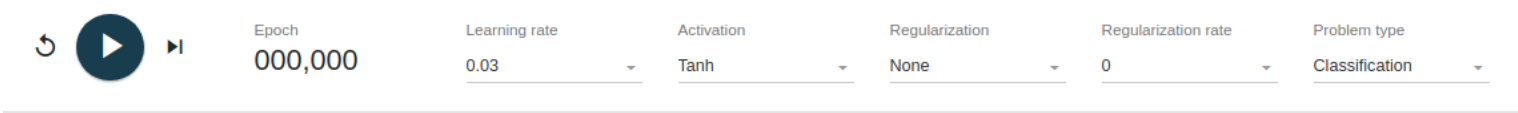
The choice of hidden units is a very active research area in Machine Learning. The type of hidden layer distinguishes the different types of Neural Networks like CNNs, RNNs etc. The number of hidden layers is termed as the depth of the neural network. One question you might ask is exactly how many layers in a network make it deep? There is no right answer to this. In general, deeper networks can learn more complex functions.

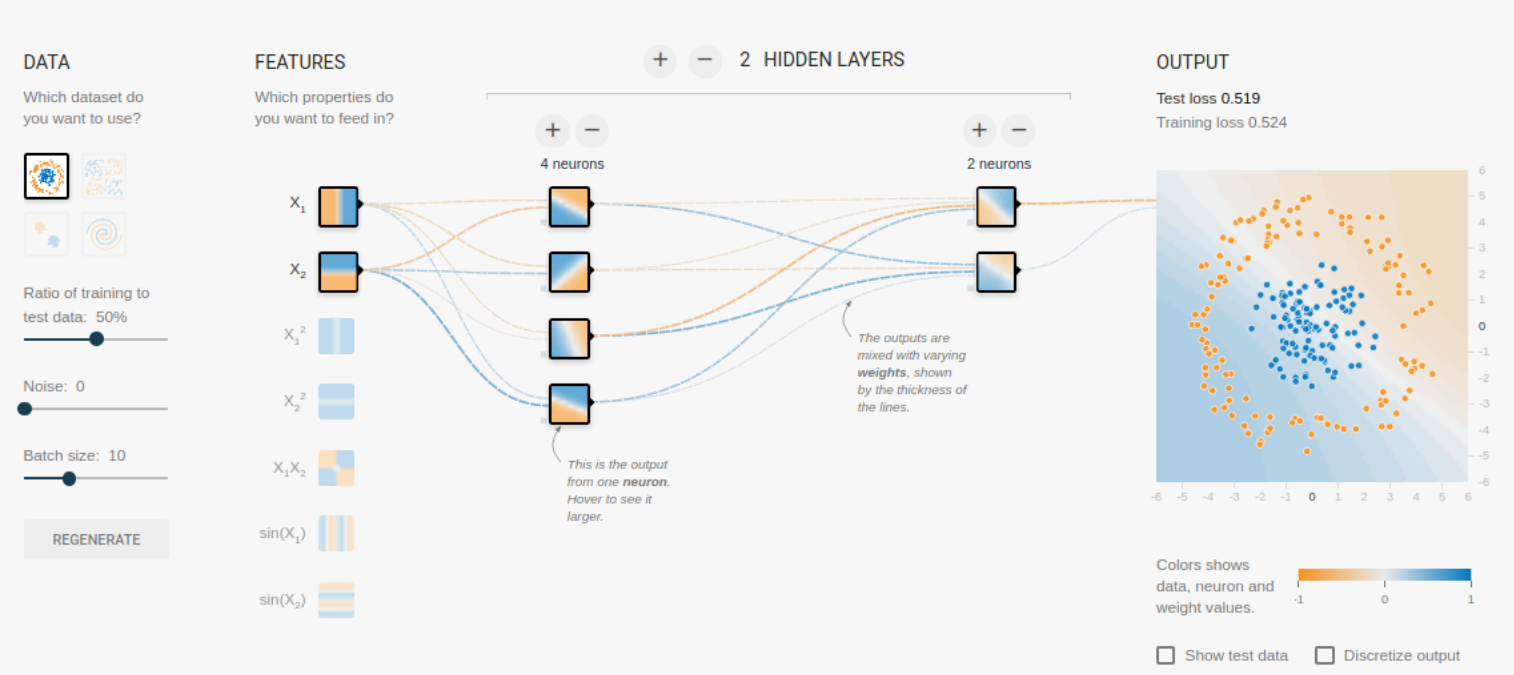
1.6. How does the network learn?

The training samples are passed through the network and the output obtained from the network is compared with the actual output. This error is used to change the weights of the neurons such that the error decreases gradually. This is done using the **Backpropagation** (http://neuralnetworksanddeeplearning.com/chap2.html) algorithm, also called backprop. Iteratively passing batches of data through the network and updating the weights, so that the error is decreased, is known as **Stochastic Gradient Descent (SGD)** (https://en.wikipedia.org/wiki/Stochastic_gradient_descent). The amount by which the weights are changed is determined by a parameter called **Learning rate**. The details of SGD and backprop will be covered in a separate post.

2. Why use Hidden Layers?

To understand the significance of hidden layers we will try to solve the binary classification problem without hidden layers. For this, we will use an interactive platform from Google, playground.tensorflow.org (http://playground.tensorflow.org) which is a web app where you can create simple feedforward neural networks and see the effects of training in real time. You can play around by changing the number of hidden layers, number of units in a hidden layer, type of activation function, type of data, learning rate, regularization parameters etc. Given below is a screenshot of the web page.



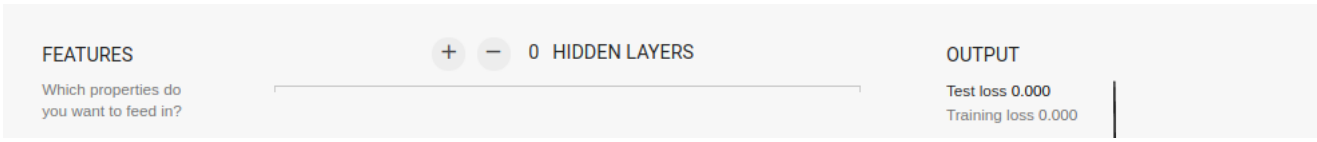


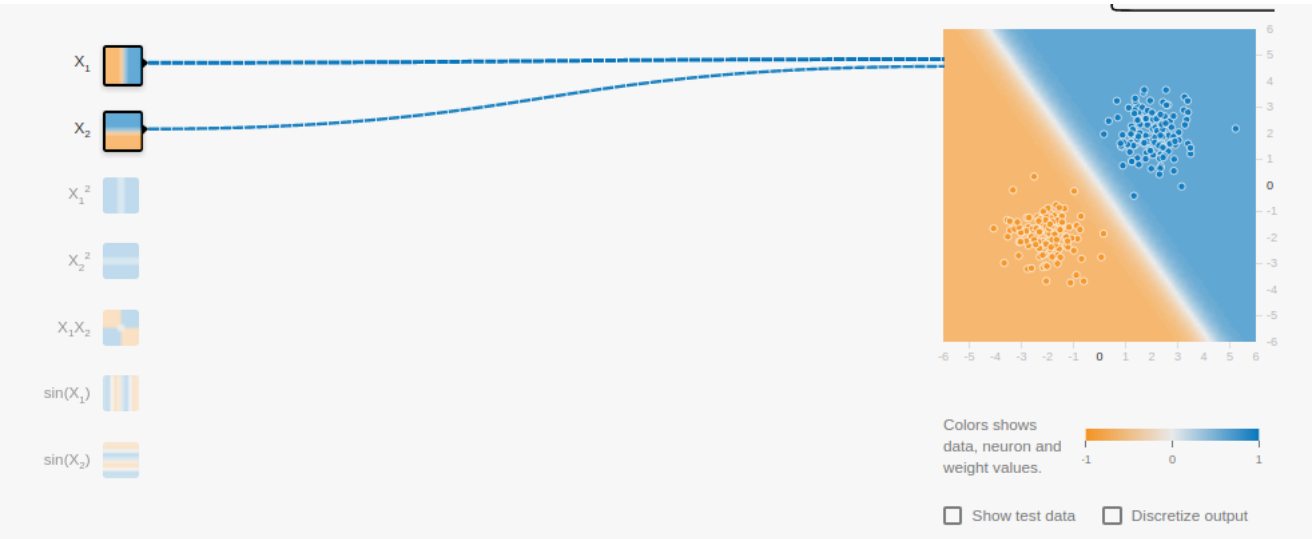
(/wp-content/uploads/2017/10/sample-playground-tensorflow.png)

In the above page, you can select the data and click on the play button to start training. It will show you the learned decision boundary and the loss curves at the top right corner.

2.1. No hidden layer

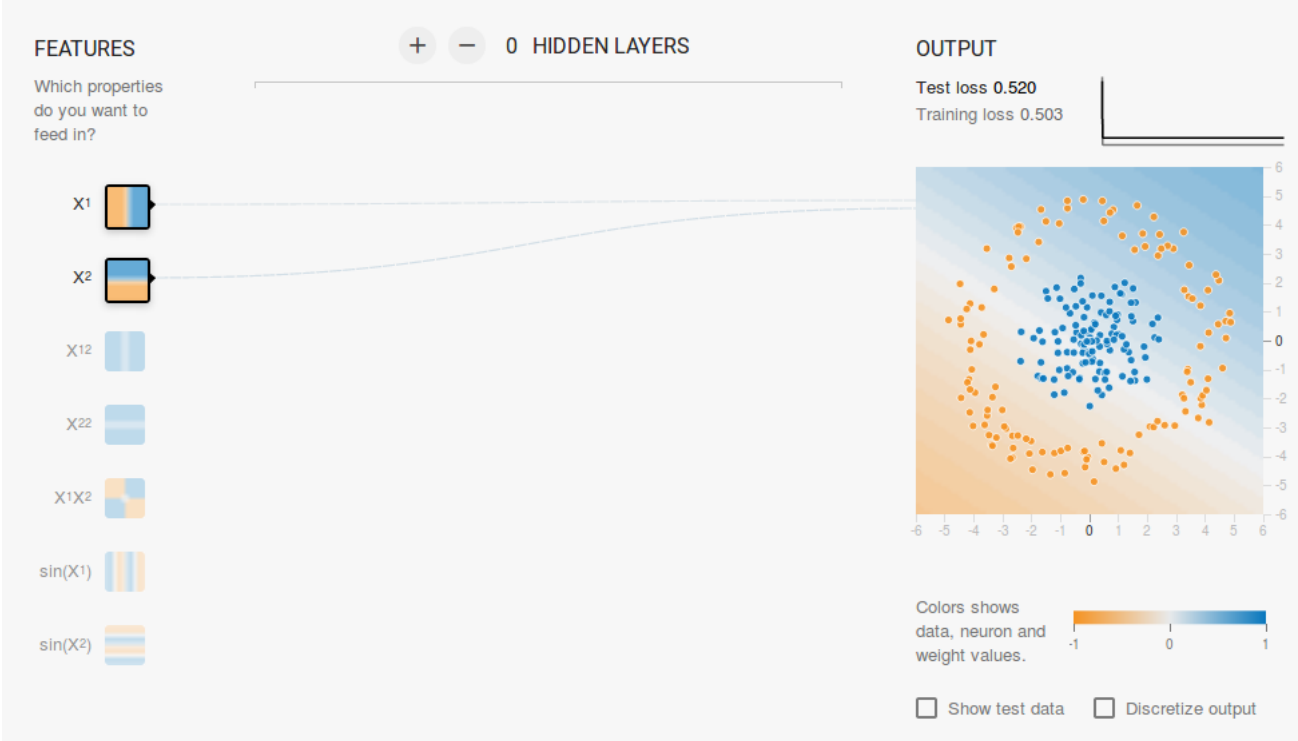
We want a network without a hidden layer which I have created in this **link** (http://playground.tensorflow.org/#activation=relu&batchSize=10&dataset=gauss&regDataset=reg-plane&learningRate=0.1&regularizationRate=0&noise=0&networkShape=&seed=0.75972&showTestData=false&discretize=false Here there are no hidden layers so it becomes a simple neuron, which is capable of learning a linear decision boundary. We can select the type of data from the top left corner. In case of linearly separable data (3rd type), it will be able to learn (when you click the play button) a linear boundary as shown below.





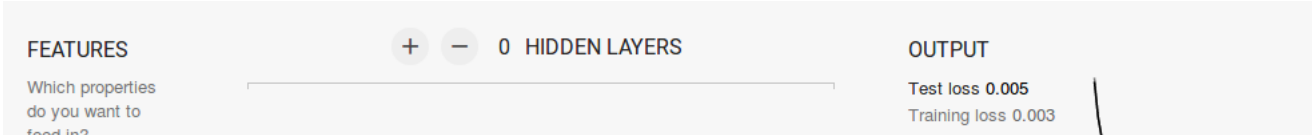
(/wp-content/uploads/2017/10/linear-output.png)

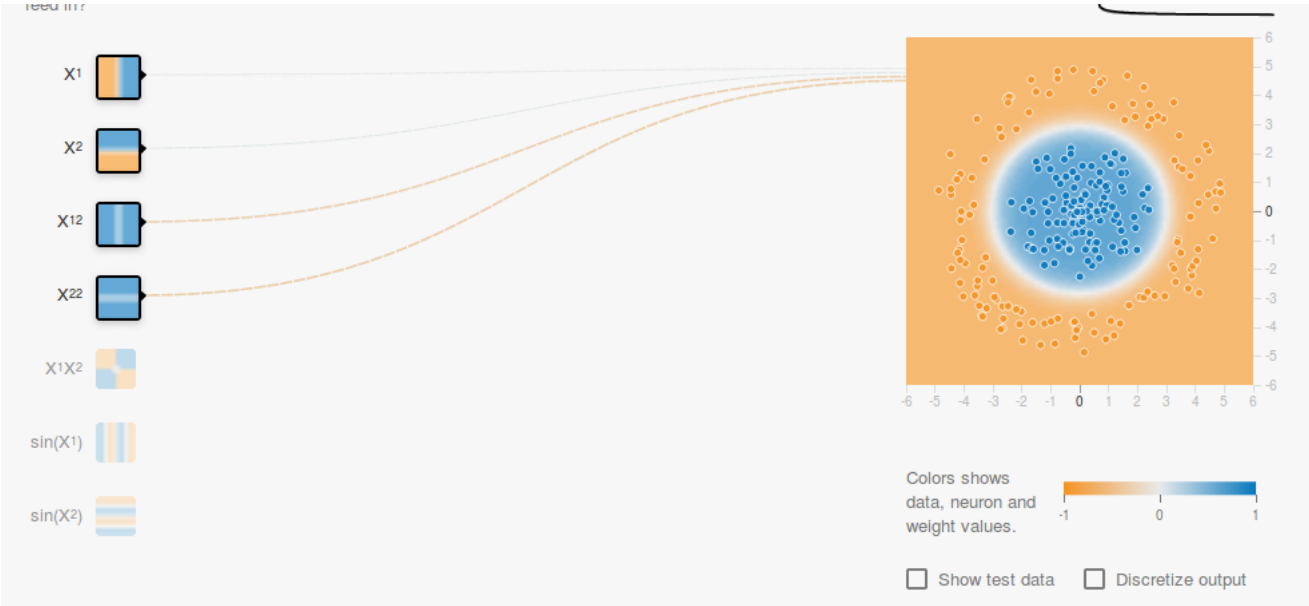
However, if you choose the 1st data it will not be able to learn the circular decision boundary.



(/wp-content/uploads/2017/10/circular-output.png)

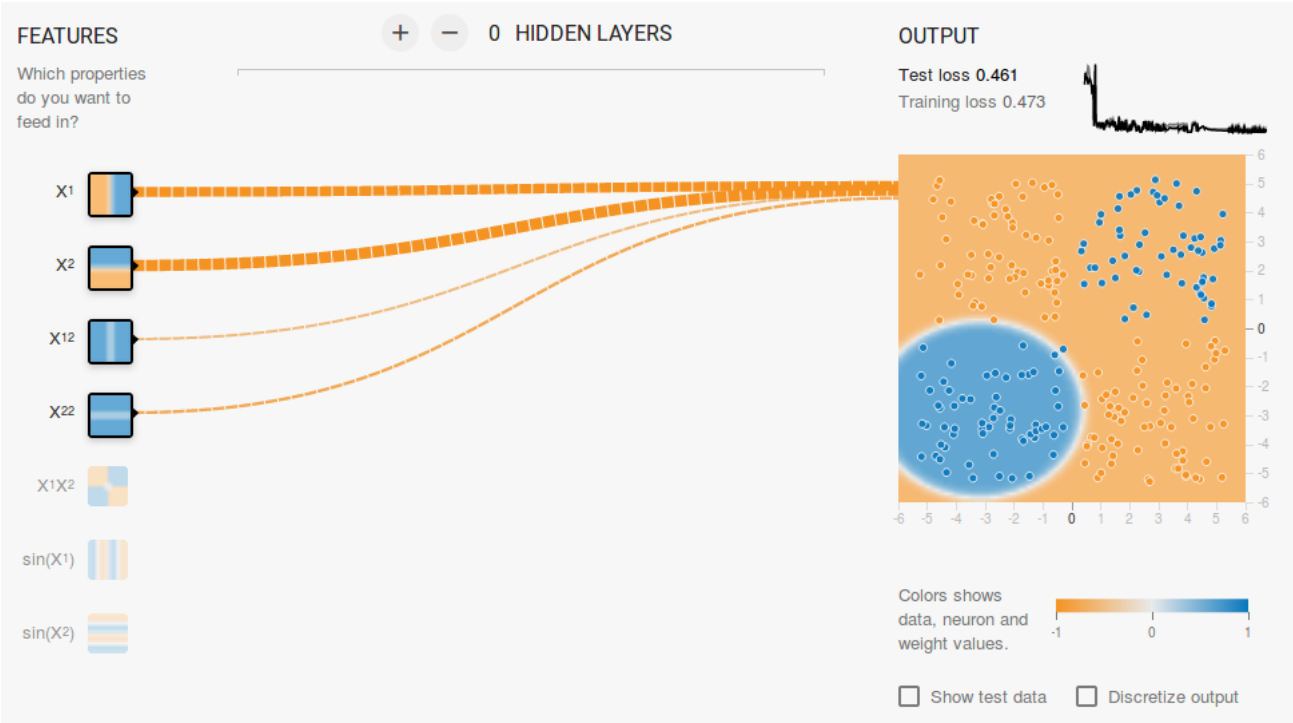
Since the data lies in a circular region, one may say that using squared values of the features as inputs might help. As it turns out, upon training, the neuron will be able to find the circular decision boundary.





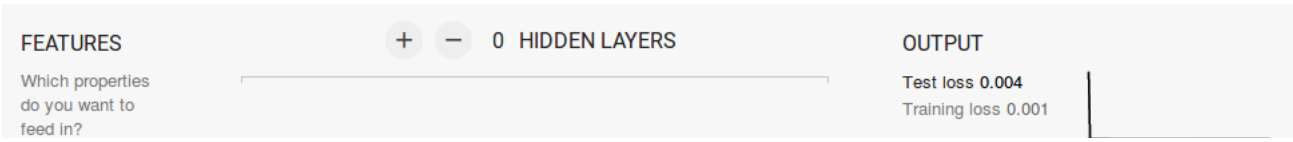
(/wp-content/uploads/2017/10/circular-output-correct.png)

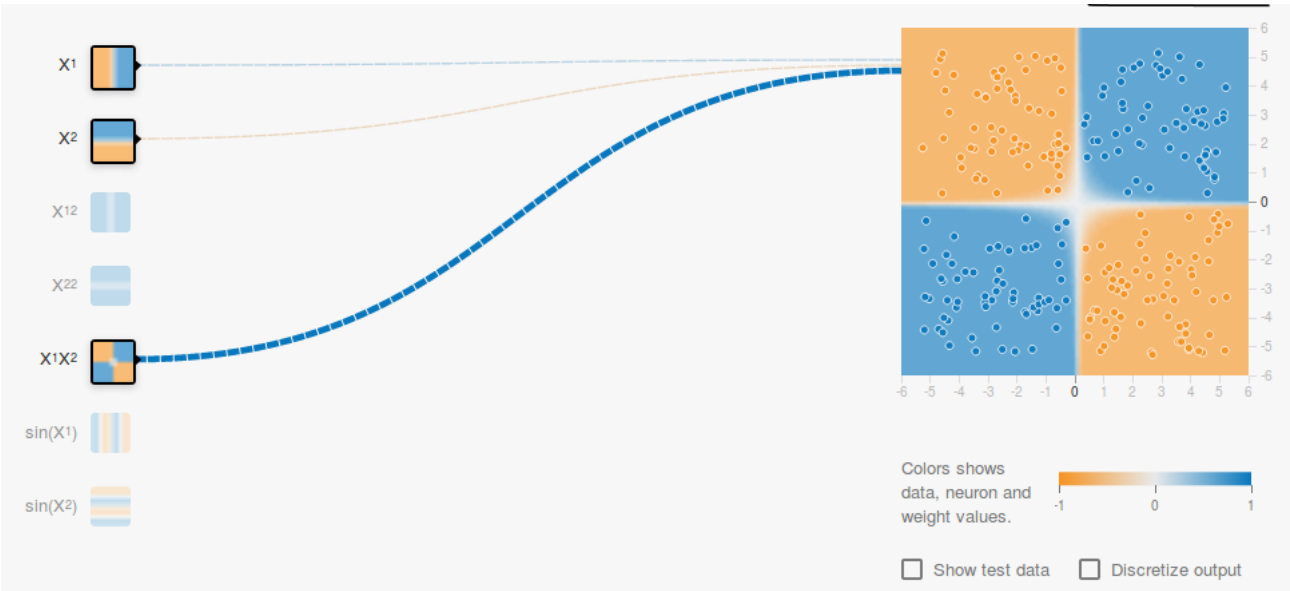
Now, if you select the 2nd data, the same configuration will not be able to learn the appropriate decision boundary.



(/wp-content/uploads/2017/10/parabola-output.png)

Again by intuition, it looks like the decision boundary is a conic section(like a parabola or hyperbola). So, if we include the product of the feature (i.e. X1X2), the neuron is able to learn the desired decision boundary.





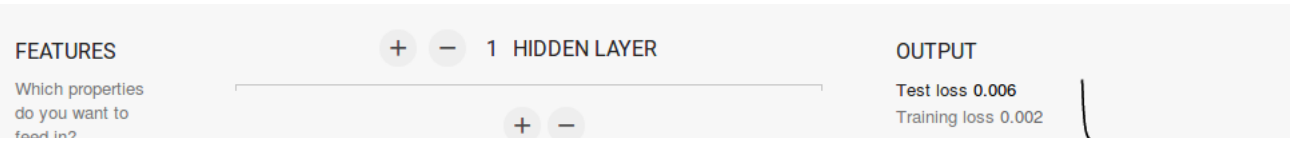
(/wp-content/uploads/2017/10/parabola-output-correct.png)

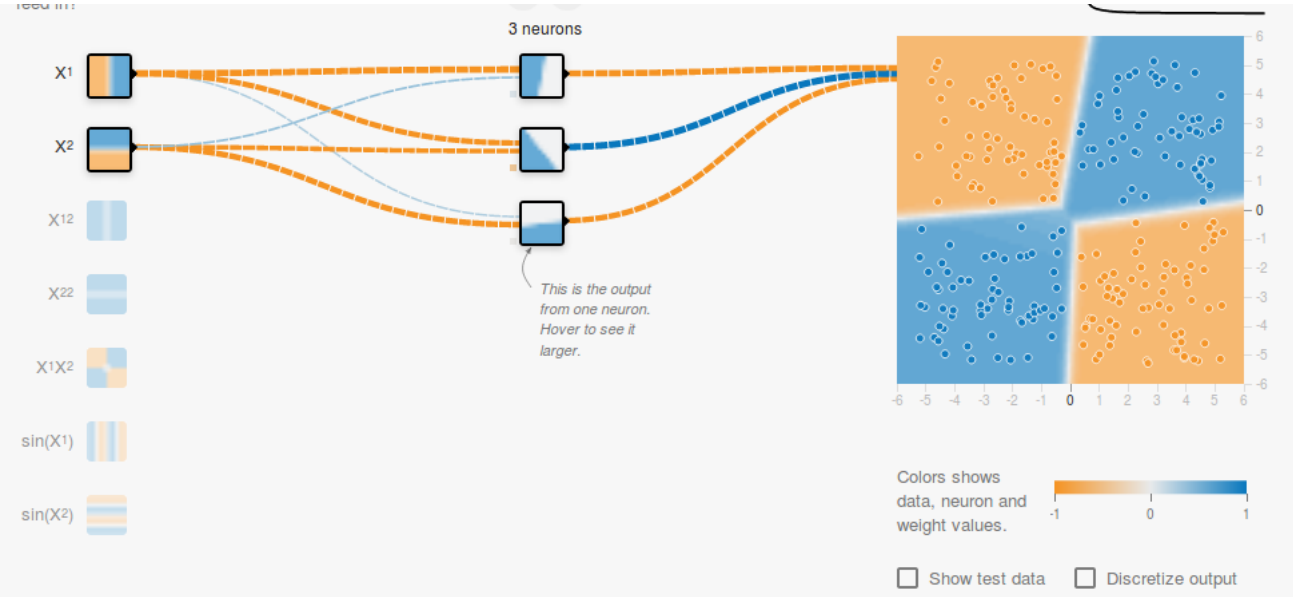
From the above experiment, we observed the following:

- Using a single neuron we can only learn a linear decision boundary
- We had to come up with feature transformations (like square of features or product of features) by visualizing the data. This step can be tricky for data which is not easy to visualize.

2.2. Adding a hidden layer

By adding a hidden layer as shown in this **link** (http://playground.tensorflow.org/#activation=relu&batchSize=10&dataset=xor&regDataset=reg-plane&learningRate=0.1&regularizationRate=0&noise=0&networkShape=3&seed=0.63075&showTestData=false&discretize=fals we can get rid of this feature engineering and have a single network which can learn all the three decision boundaries. A Neural Network with a single hidden layer with nonlinear activation functions is considered to be a **Universal Function Approximator** (https://en.wikipedia.org/wiki/Universal_approximation_theorem) (i.e. capable of learning any function). However, the number of units in the hidden layer is not fixed. The result of adding a hidden layer with just 3 neurons is shown below:



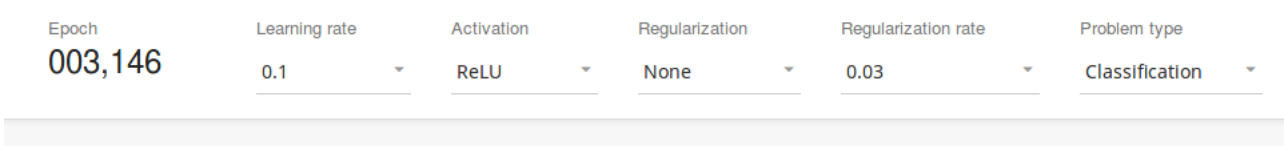


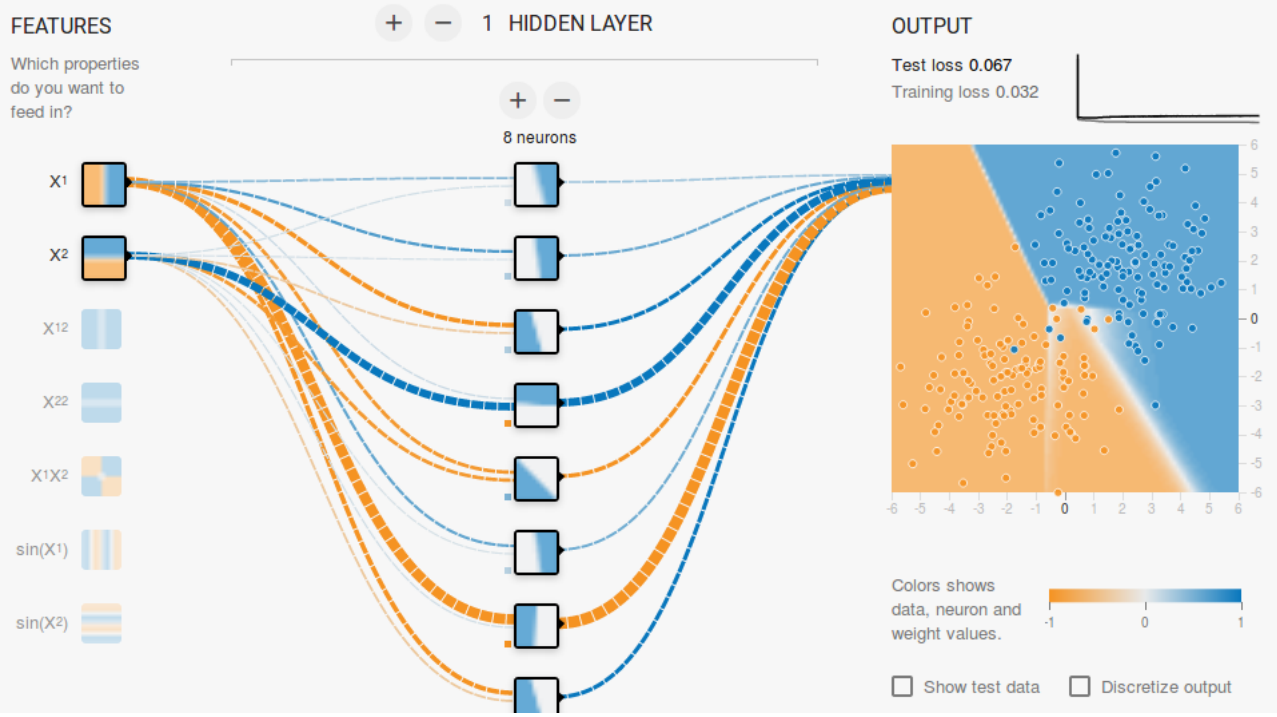
(/wp-content/uploads/2017/10/nonlinear-output-correct.png)

3. Regularization

As we saw in the previous section, a multilayer network can learn nonlinear decision boundaries. However, if there is noise in the data (which is often the case) the network may try to learn the nonlinearity introduced by the noise too, trying to fit the noisy samples. In such cases, the noisy samples should be treated as outliers. In this **link** (http://playground.tensorflow.org/#activation=relu&batchSize=10&dataset=gauss&regDataset=reg-plane&learningRate=0.1&regularizationRate=0.03&noise=30&networkShape=8&seed=0.46428&showTestData=false&discretize=

I have added some noise to the linearly separable data. Also, to demonstrate the idea, I have increased the number of hidden units.

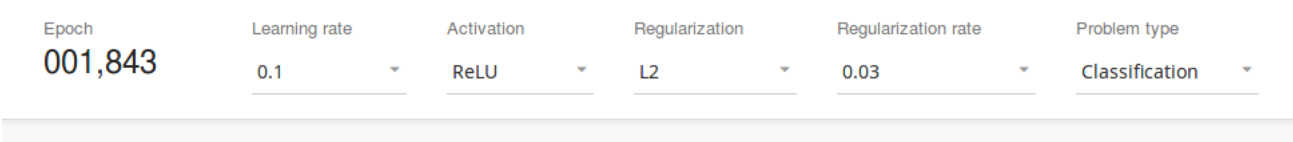


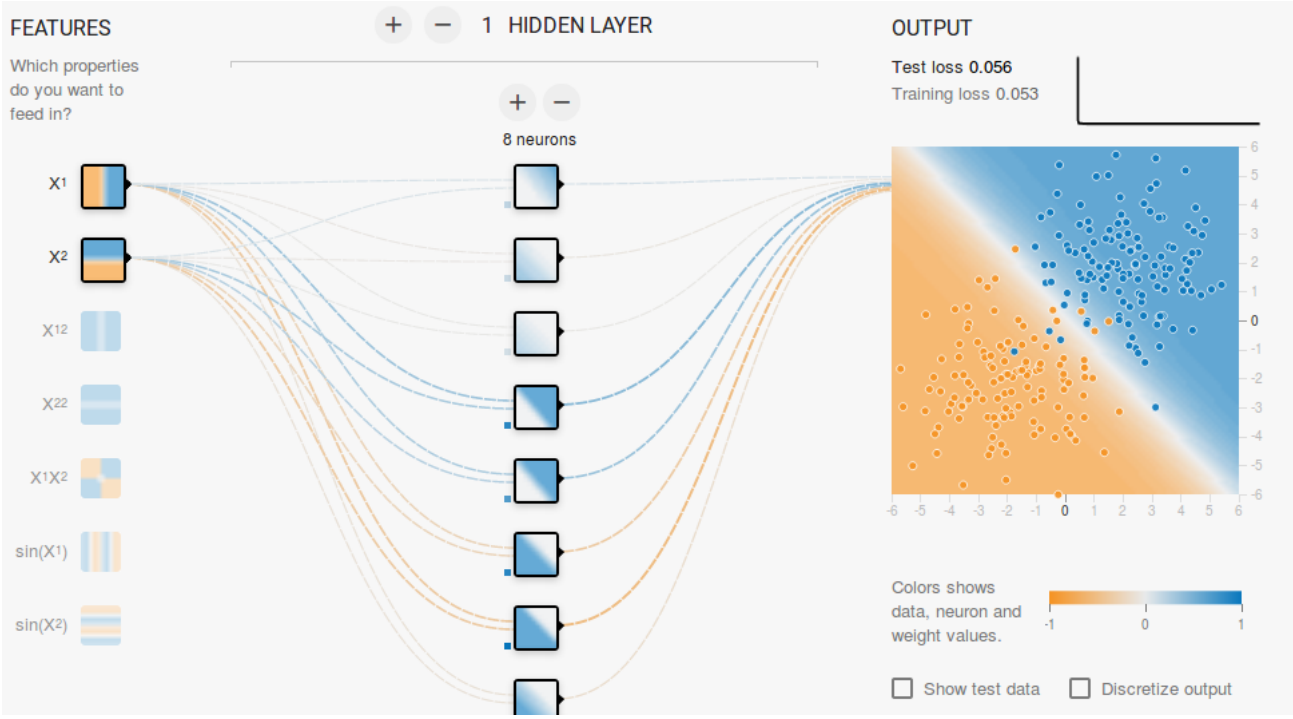


(/wp-content/uploads/2017/10/overfitting-linear-data.png)

In the above figure, it can be seen that the decision boundary is trying very hard to accommodate the noisy samples in order to reduce the error. But as you can see it is being misguided by the noisy samples. In other words, the network will be fragile in the presence of noise. This phenomenon is called **Overfitting**. In such cases, the error on training data might decrease but the network performs badly on unseen data. It can be seen from the loss curves at the top right corner.

The training loss is decreasing but the test loss is increasing. Also, you can see that some weights have become very large (very thick connections or you can see the weights if you hover above the connections). This can be rectified by putting some restrictions on the values of weights (like not allowing the weights to become very high). This is called **Regularization**. We impose restrictions on the other parameters of the network. In a sense, we don't trust the training data fully and want the network to learn "nice" decision boundaries. I have added L2 regularization to the above configuration in this **link** (http://playground.tensorflow.org/#activation=relu&regularization=L2&batchSize=10&dataset=gauss&regDataset=reg-plane&learningRate=0.1&regularizationRate=0.03&noise=30&networkShape=8&seed=0.46428&showTestData=false&discretize= and the output is shown below.





(/wp-content/uploads/2017/10/regulariztion-output.png)
After including L2 regularization, the decision boundary learned by the network is smoother and similar to the case when there was no noise. The effect of regularization can also be seen from the loss curves and the value of the weights.

In the next post, we will learn how to implement a feedforward neural network in Keras for solving a multi-class classification problem and learn more about feedforward networks.

Subscribe & Download Code

If you liked this article and would like to download code (C++ and Python) and example images used in all posts of this blog, please subscribe (https://bigvisionllc.leadpages.net/leadbox/143948b73f72a2%3A173c9390c346dc/5649050225344512/) to our newsletter. You will also receive a free Computer Vision Resource (https://bigvisionllc.leadpages.net/leadbox/143948b73f72a2%3A173c9390c346dc/5649050225344512/) Guide. In our newsletter, we share OpenCV tutorials and examples written in C++/Python, and Computer Vision and Machine Learning algorithms and news.

Subscribe Now
(https://bigvisionllc.leadpages.net/leadbox/143948b73f72a2%3A173c9390c346dc/5649050225344512/)

Learn OpenCV

Image Classification using Feedforward Neural Network in Keras

OCTOBER 23, 2017 BY VIKAS GUPTA (HTTPS://WWW.LEARNOPENCV.COM/AUTHOR/VIKAS/)

This post is part of the series on Deep Learning for Beginners, which consists of the following tutorials :

1. Neural Networks : A 30,000 Feet View for Beginners (/neural-networks-a-30000-feet-view-for-beginners/)
2. Installation of Deep Learning frameworks (Tensorflow and Keras with CUDA support) (/installing-deep-learning-frameworks-on-ubuntu-with-cuda-support/)
3. Introduction to Keras (/deep-learning-using-keras-the-basics/)
4. Understanding Feedforward Neural Networks (/understanding-feedforward-neural-networks/)
5. Image Classification using Feedforward Neural Networks
6. Image Recognition using Convolutional Neural Network (/image-classification-using-convolutional-neural-networks-in-keras/)
7. Understanding Activation Functions (/understanding-activation-functions-in-deep-learning/)
8. Understanding AutoEncoders using Tensorflow (/understanding-autoencoders-using-tensorflow-python/)
9. Image Classification using pre-trained models in Keras (/keras-tutorial-using-pre-trained-imagenet-models/)
10. Transfer Learning using pre-trained models in Keras (/keras-tutorial-transfer-learning-using-pre-trained-models/)
11. Fine-tuning pre-trained models in Keras (/keras-tutorial-fine-tuning-using-pre-trained-models)
12. More to come . . .

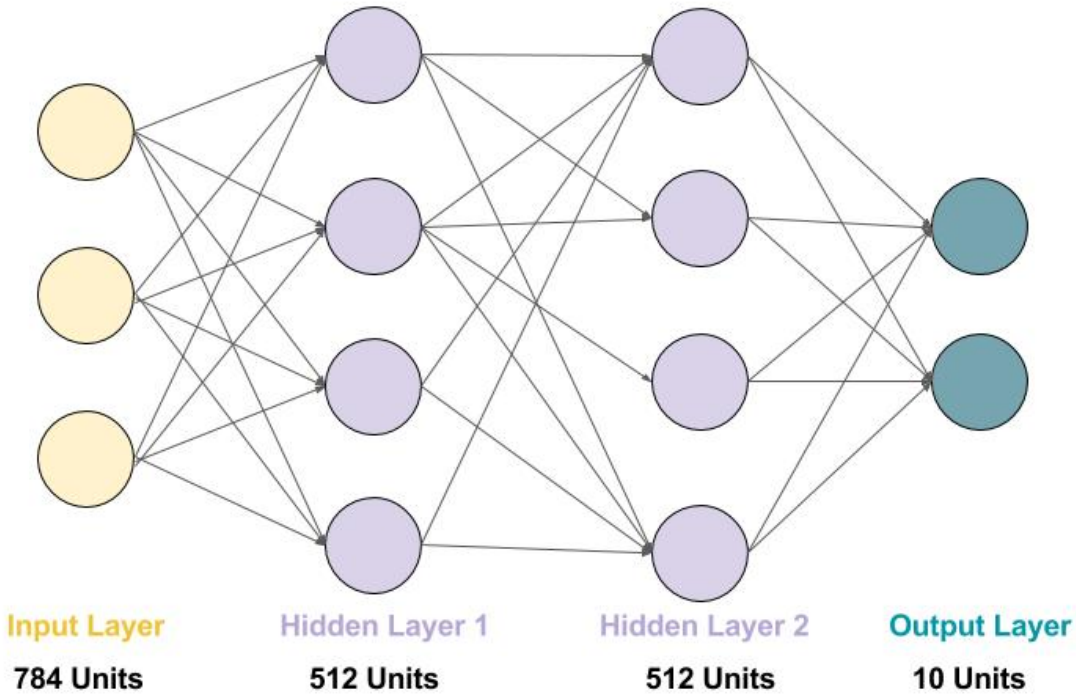
In this article, we will learn how to implement a Feedforward Neural Network in Keras. We will use handwritten digit classification as an example to illustrate the effectiveness of a feedforward network. We will also see how to spot and overcome **Overfitting** during training.

MNIST (https://en.wikipedia.org/wiki/MNIST_database) is a commonly used handwritten digit dataset consisting of 60,000 images in the training set and 10,000 images in the test set. So, each digit has 6000 images in the training set. The digits are size-normalized and centered in a fixed-size (28×28) image. The task is to train a machine learning algorithm to recognize a new sample from the test set correctly.

1. The Network

For a quick understanding of Feedforward Neural Network, you can have a look at our **previous article**

(/understanding-feedforward-neural-networks/). We will use raw pixel values as input to the network. The images are matrices of size 28×28. So, we reshape the image matrix to an array of size 784 (28*28) and feed this array to the network. We will use a network with 2 hidden layers having 512 neurons each. The output layer will have 10 layers for the 10 digits. A schematic diagram is shown below.



(/wp-content/uploads/2017/10/mlp-mnist-schematic.jpg)

Check out **this post** (/deep-learning-using-keras-the-basics/) if you don't have Keras installed yet! Also, download the code from the link below to follow along with the post.

Download Code

To easily follow along this tutorial, please download code by clicking on the button below. It's FREE!

DOWNLOAD CODE
(HTTPS://BIGVISIONLLC.LEADPAGES.NET/LEADBOX/143948B73F72A2%3A173C9390C346DC/5649050225344512/)

Let us dive into the code!

2. Load the Data

Keras comes with the MNIST data loader. It has a function mnist.load_data() which downloads the data from

its servers if it is not present on your computer. The data loaded using this function is divided into training and test sets. This is done by the following :

```
from keras.datasets import mnist
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
```

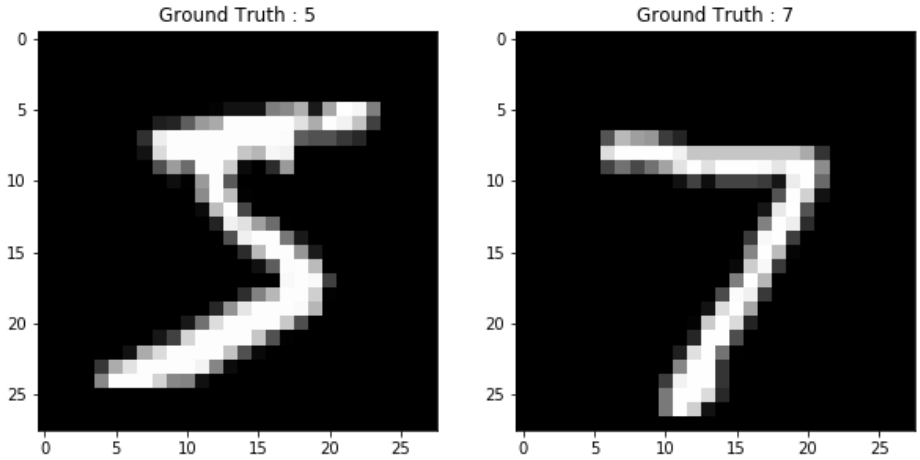
3. Checkout the Data

Let's see how the data looks like. The data consists of handwritten numbers ranging from 0 to 9, along with their ground truth. It has 60,000 train samples and 10,000 test samples. Each sample is a 28×28 grayscale image.

```
from keras.utils import to_categorical

print('Training data shape : ', train_images.shape, train_labels.shape)

print('Testing data shape : ', test_images.shape, test_labels.shape)

# Find the unique numbers from the train labels
classes = np.unique(train_labels)
nClasses = len(classes)
print('Total number of outputs : ', nClasses)
print('Output classes : ', classes)

plt.figure(figsize=[10,5])

# Display the first image in training data
plt.subplot(121)
plt.imshow(train_images[0,:,:], cmap='gray')
plt.title("Ground Truth : {}".format(train_labels[0]))

# Display the first image in testing data
plt.subplot(122)
plt.imshow(test_images[0,:,:], cmap='gray')
plt.title("Ground Truth : {}".format(test_labels[0]))
```

Output:

```
Training data shape :  (60000, 28, 28) (60000,)
Testing data shape :  (10000, 28, 28) (10000,)
Total number of outputs :  10
Output classes :  [0 1 2 3 4 5 6 7 8 9]

<matplotlib.text.Text at 0x7f64179992d0>
```



(/wp-content/uploads/2017/10/sample-data-mnist.png)

4. Process the data

The images are grayscale and the pixel values range from 0 to 255. We will apply the following preprocessing to the data before feeding it to the network.

1. Convert each image matrix (28×28) to an array (28*28 = 784 dimenstional) which will be fed to the network as a single feature.

```
# Change from matrix to array of dimension 28x28 to array of dimention 784
dimData = np.prod(train_images.shape[1:])
train_data = train_images.reshape(train_images.shape[0], dimData)
test_data = test_images.reshape(test_images.shape[0], dimData)
```

2. Convert the data to float and scale the values between 0 to 1.

```
# Change to float datatype
train_data = train_data.astype('float32')
test_data = test_data.astype('float32')

# Scale the data to lie between 0 to 1
train_data /= 255
test_data /= 255
```

3. Convert the labels from integer to categorical (one-hot) encoding since that is the format required by Keras to perform multiclass classification. One-hot encoding is a type of boolean representation of integer data. It converts the integer to an array of all zeros except a 1 at the index of the integer. For example, using a one-hot encoding for 10 classes, the integer 5 will be encoded as 0000010000

```
# Change the labels from integer to categorical data
train_labels_one_hot = to_categorical(train_labels)
test_labels_one_hot = to_categorical(test_labels)

# Display the change for category label using one-hot encoding
print('Original label 0 : ', train_labels[0])
print('After conversion to categorical ( one-hot ) : ', train_labels_one_hot[0])
```

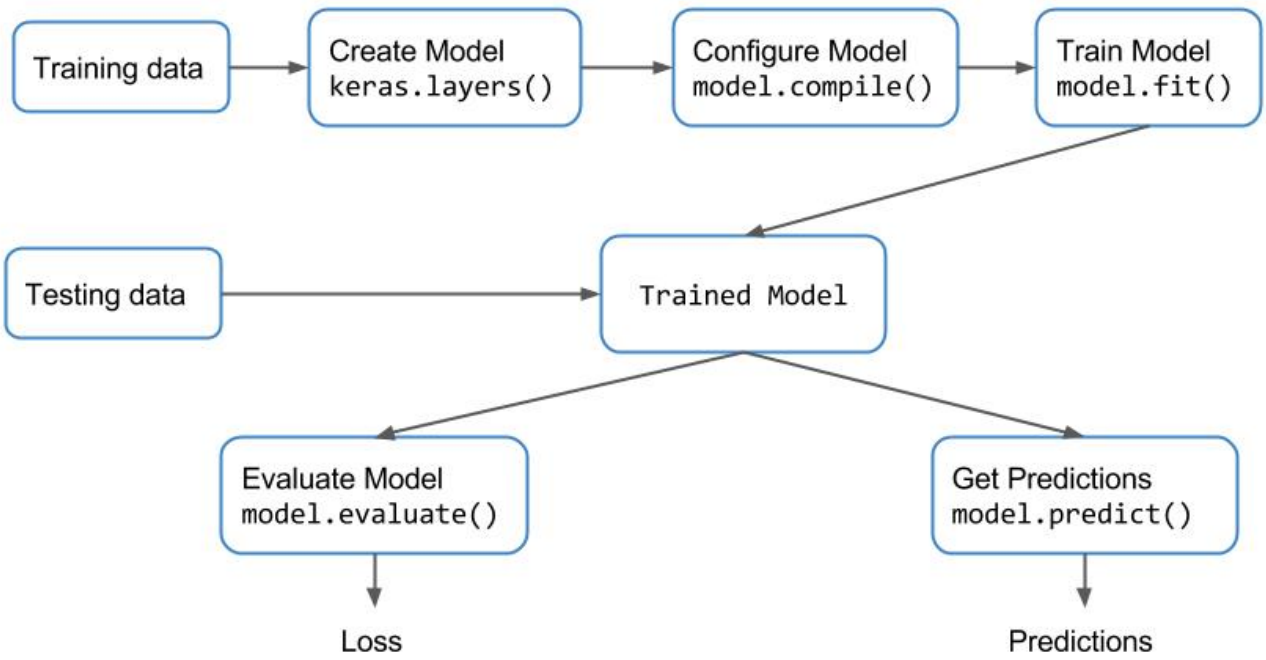
Output:

Original label 0 : 5

After conversion to categorical (one-hot) : [0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]

5. Keras Workflow for training the network

We have described the Keras Workflow in our **previous post** (/deep-learning-using-keras-the-basics/). The block diagram is given here for reference. Basically, once you have the training and test data, you can follow these steps to train a neural network in Keras.



(/wp-content/uploads/2017/09/keras-workflow.jpg)

5.1. Create the Network

We had mentioned that we will be using a network with 2 hidden layers and an output layer with 10 units. The number of units in the hidden layers is kept to be 512. The input to the network is the 784-dimensional array converted from the 28×28 image.

We will use the Sequential model for building the network. In the Sequential model, we can just stack up layers by adding the desired layer one by one. We use the Dense layer, also called fully connected layer since we are building a feedforward network in which all the neurons from one layer are connected to the

neurons in the previous layer. Apart from the Dense layer, we add the ReLU activation function which is required to introduce non-linearity to the model. This will help the network learn non-linear decision boundaries. The last layer is a softmax layer as it is a multiclass classification problem. For binary classification, we can use sigmoid.

```
from keras.models import Sequential
from keras.layers import Dense

model = Sequential()
model.add(Dense(512, activation='relu', input_shape=(dimData,)))
model.add(Dense(512, activation='relu'))
model.add(Dense(nClasses, activation='softmax'))
```

5.2. Configure the Network

In this step, we configure the optimizer to be rmsprop. We also specify the loss type which is categorical cross entropy which is used for multiclass classification. We also specify the metrics (accuracy in this case) which we want to track during the training process. You can also try using any other optimizer such as adam or SGD.

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

5.3. Train the Model

The network is ready to get trained. This is done using the fit() function in Keras. We specify the number of epochs as 20. This means that the whole dataset will be fed to the network 20 times. We will be using the test data for validation.

```
history = model.fit(train_data, train_labels_one_hot, batch_size=256, epochs=20, verbose=1,
                    validation_data=(test_data, test_labels_one_hot))
```

5.4. Evaluate the trained model

We check the performance on the whole test data using the evaluate() method.

```
[test_loss, test_acc] = model.evaluate(test_data, test_labels_one_hot)
print("Evaluation result on Test Data : Loss = {}, accuracy = {}".format(test_loss, test_acc))
```

Output:
Evaluation result on Test Data : Loss = 0.135059975359, accuracy = 0.9807

The results look good. However, we would want to have another look at the results.

6. Check for Overfitting

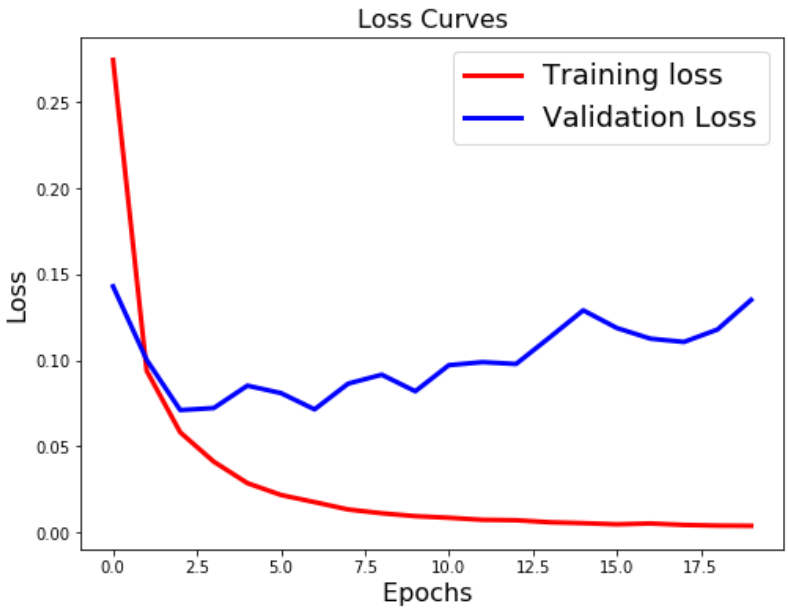
The fit() function returns a history object which has a dictionary of all the metrics which were required to be

tracked during training. We can use the data in the history object to plot the loss and accuracy curves to check how the training process went.
You can use the history.history.keys() function to check what metrics are present in the history. It should look like the following

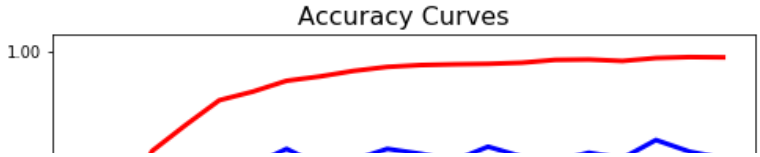
['acc', 'loss', 'val_acc', 'val_loss']

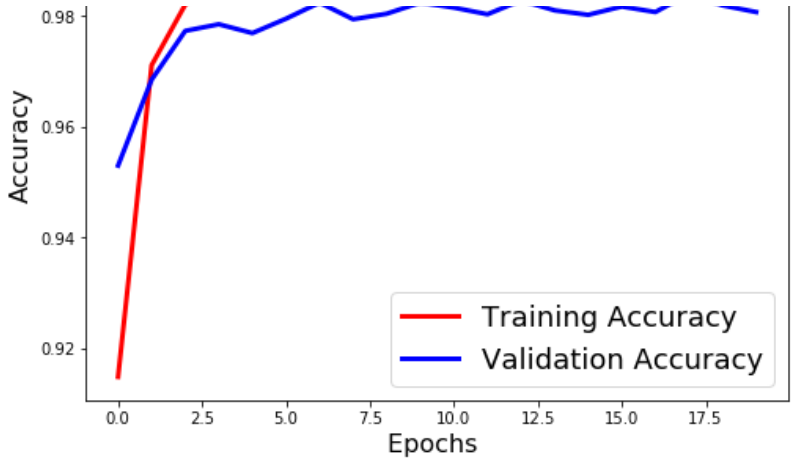
Let us plot the loss and accuracy curves.

```
#Plot the Loss Curves
plt.figure(figsize=[8,6])
plt.plot(history.history['loss'],'r',linewidth=3.0)
plt.plot(history.history['val_loss'],'b',linewidth=3.0)
plt.legend(['Training loss', 'Validation Loss'],fontsize=18)
plt.xlabel('Epochs ',fontsize=16)
plt.ylabel('Loss',fontsize=16)
plt.title('Loss Curves',fontsize=16)

#Plot the Accuracy Curves
plt.figure(figsize=[8,6])
plt.plot(history.history['acc'],'r',linewidth=3.0)
plt.plot(history.history['val_acc'],'b',linewidth=3.0)
plt.legend(['Training Accuracy', 'Validation Accuracy'],fontsize=18)
plt.xlabel('Epochs ',fontsize=16)
plt.ylabel('Accuracy',fontsize=16)
plt.title('Accuracy Curves',fontsize=16)
```



(/wp-content/uploads/2017/10/loss-curve-without-reg.png)





(/wp-content/uploads/2017/10/acc-curve-without-reg.png)

Although the accuracy obtained above is very good, if you see the loss and accuracy curves in the above figures, you'll notice that the validation loss initially decrease, but then it starts increasing gradually. Also, there is a substantial difference between the training and test accuracy. This is a **clear sign of Overfitting** which means that the network has memorized the training data very well, but is not guaranteed to work on unseen data. Thus, the difference in the training and test accuracy.

7. Add Regularization to the model

Overfitting occurs mainly because the network parameters are getting too biased towards the training data. We can add a dropout layer to overcome this problem to a certain extent. In case of dropout, a fraction of neurons is randomly turned off during the training process, reducing the dependency on the training set by some amount.

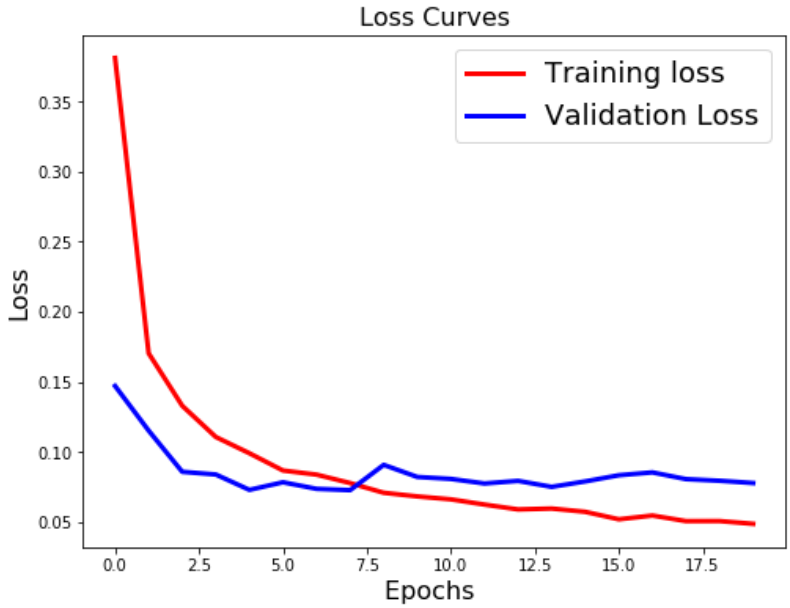
```
from keras.layers import Dropout

model_reg = Sequential()
model_reg.add(Dense(512, activation='relu', input_shape=(dimData,)))
model_reg.add(Dropout(0.5))
model_reg.add(Dense(512, activation='relu'))
model_reg.add(Dropout(0.5))
model_reg.add(Dense(nClasses, activation='softmax'))
```

8. Check performance after regularization

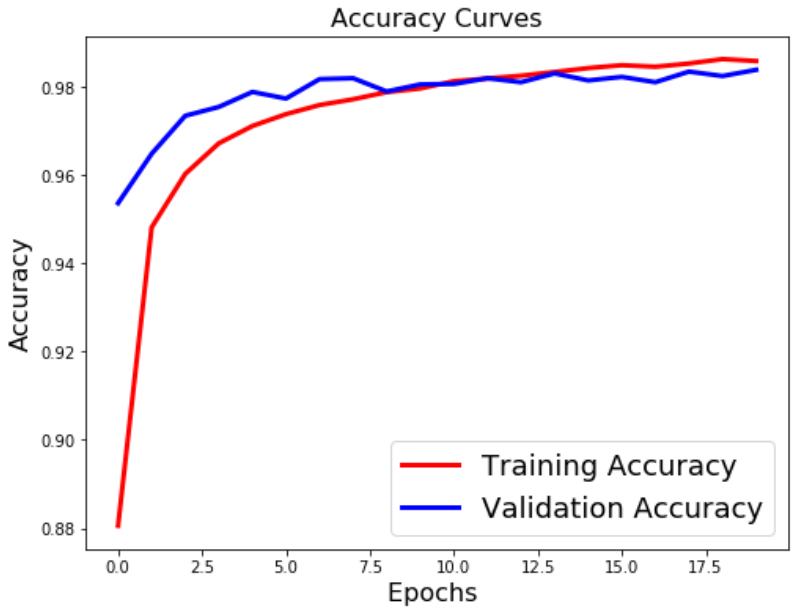
We will train the network again in the same way we did earlier and check the loss and accuracy curves.

```
model_reg.compile(optimizer='rmsprop', loss='categorical_crossentropy', metrics=['accuracy'])
history_reg = model_reg.fit(train_data, train_labels_one_hot, batch_size=256, epochs=20, verbose=1,
                            validation_data=(test_data, test_labels_one_hot))

#Plot the Loss Curves
plt.figure(figsize=[8,6])
plt.plot(history_reg.history['loss'],'r',linewidth=3.0)
```

```
plt.plot(history_reg.history['val_loss'],'b',linewidth=3.0)
plt.legend(['Training loss', 'Validation Loss'],fontsize=18)
plt.xlabel('Epochs ',fontsize=16)
plt.ylabel('Loss',fontsize=16)
plt.title('Loss Curves',fontsize=16)

#Plot the Accuracy Curves
plt.figure(figsize=[8,6])
plt.plot(history_reg.history['acc'],'r',linewidth=3.0)
plt.plot(history_reg.history['val_acc'],'b',linewidth=3.0)
plt.legend(['Training Accuracy', 'Validation Accuracy'],fontsize=18)
plt.xlabel('Epochs ',fontsize=16)
plt.ylabel('Accuracy',fontsize=16)
plt.title('Accuracy Curves',fontsize=16)
```



(/wp-content/uploads/2017/10/loss-curve-with-reg.png)



(/wp-content/uploads/2017/10/acc-curve-with-reg.png)

From the above loss and accuracy curves, we can observe that

- The validation loss is not increasing
- The difference between the train and validation accuracy is not very high

Thus, we can say that the model has better generalization capability as the performance does not decrease drastically in case of unseen data also.

9. Inference on a single image

We have seen that the first image in the test set is the number 7. Let us see what the model predicts.

9.1. Getting the predicted class

During the inference stage, it might be sufficient to know the class of the input data. It can be done as follows.

```
# Predict the most likely class
model_reg.predict_classes(test_data[[0],:])
```

Output:
array([7])

9.2. Getting the probabilities

In the above method there is no score which tells us about the confidence with which the model does the prediction. In some cases, for example when there are many classes, we may want the probabilities of the different classes which indicates how confident the model is about the occurence of a particular class. We can take the decision based on these scores.

```
# Predict the probabilities for each class
model_reg.predict(test_data[[0],:])
```

Output:
array([[1.46786899e-23, 1.73912635e-15, 3.05286026e-12,
3.48179753e-12, 2.16374247e-22, 3.82367185e-19,
2.31083363e-30, 1.00000000e+00, 2.78843536e-18,
1.55856298e-14]], dtype=float32)

This gives the probability score for each class. We can see that the score for the 8th index is almost 1 which indicates that the predicted class is 7 with a confidence score of 1.

10. Exercise

We had used 2 hidden layers and relu activation. Try to change the number of hidden layer and the activation to tanh or sigmoid and see what happens. Also change the dropout ratio and check the performance.

Although the performance is pretty impressive with this model, we will see how to improve it further using a Convolutional Neural Network in the next post. Stay tuned!

Subscribe & Download Code

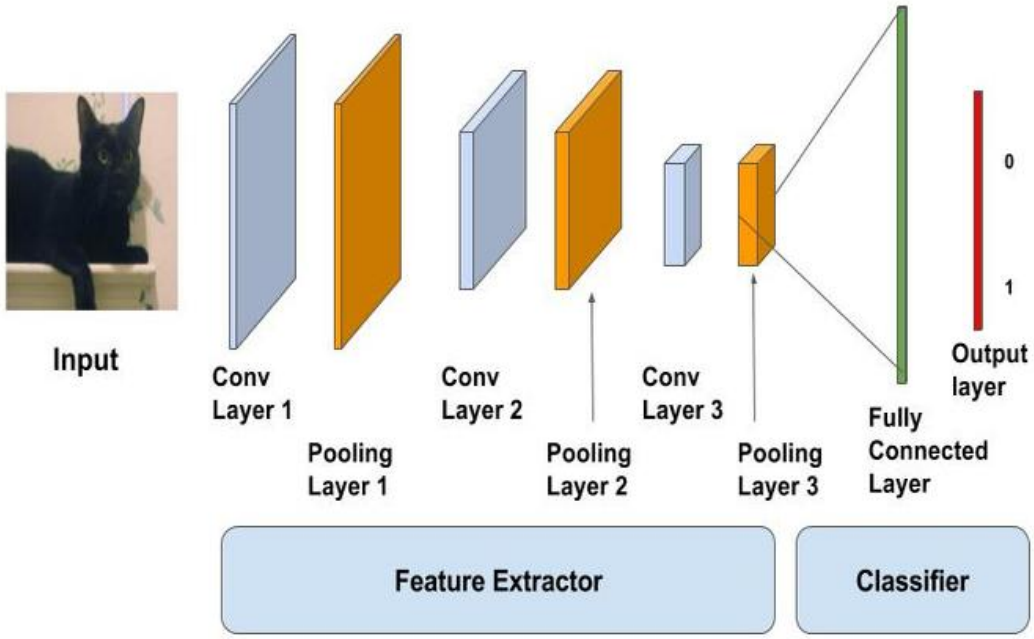
If you liked this article and would like to download code and example images used in this post, please subscribe (https://bigvisionllc.leadpages.net/leadbox/143948b73f72a2%3A173c9390c346dc/5649050225344512/) to our newsletter. You will also receive a free Computer Vision Resource (https://bigvisionllc.leadpages.net/leadbox/143948b73f72a2%3A173c9390c346dc/5649050225344512/) Guide. In our newsletter, we share OpenCV tutorials and examples written in C++/Python, and Computer Vision and Machine Learning algorithms and news.

Subscribe Now
(https://bigvisionllc.leadpages.net/leadbox/143948b73f72a2%3A173c9390c346dc/5649050225344512/)

Learn OpenCV

Image Classification using Convolutional Neural Networks in Keras

NOVEMBER 29, 2017 BY VIKAS GUPTA (HTTPS://WWW.LEARNOPENCV.COM/AUTHOR/VIKAS/)



(https://www.learnopencv.com/wp-content/uploads/2017/11/cnn-schema1.jpg)

In this tutorial, we will learn the basics of Convolutional Neural Networks (CNNs) and how to use them for an Image Classification task. We will also see how data augmentation helps in improving the performance of the network. We discussed Feedforward Neural Networks (/understanding-feedforward-neural-networks/), Activation Functions (/understanding-activation-functions-in-deep-learning/), and Basics of Keras (/deep-learning-using-keras-the-basics/) in the previous tutorials. We will use the MNIST and CIFAR10 datasets for illustrating various concepts.

This post is part of the series on Deep Learning for Beginners, which consists of the following tutorials :

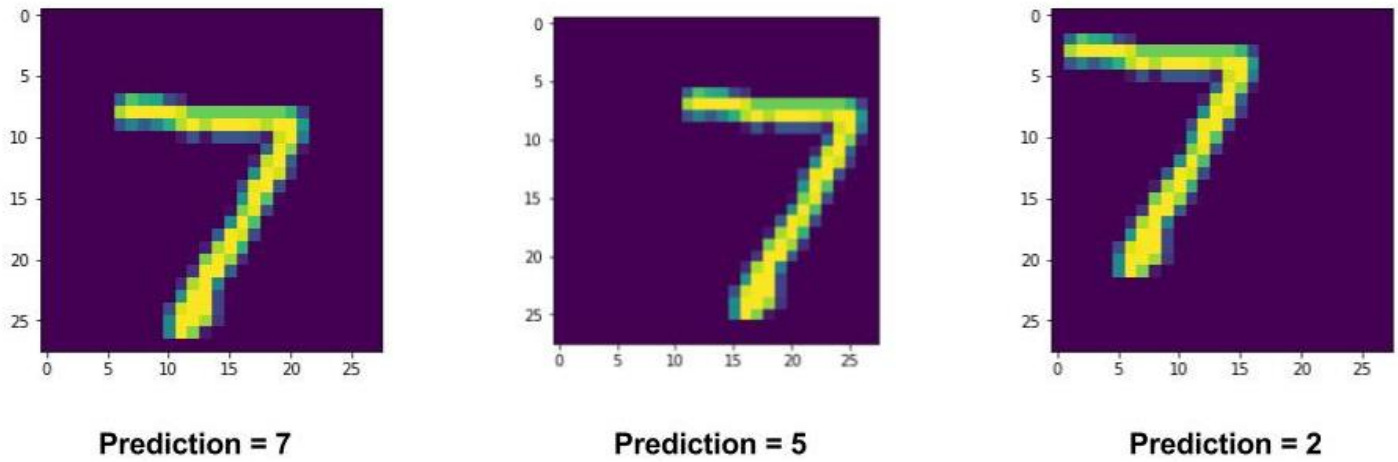
1. Neural Networks : A 30,000 Feet View for Beginners (/neural-networks-a-30000-feet-view-for-beginners/)
2. Installation of Deep Learning frameworks (Tensorflow and Keras with CUDA support) (/installing-deep-learning-frameworks-on-ubuntu-with-cuda-support/)
3. Introduction to Keras (/deep-learning-using-keras-the-basics/)
4. Understanding Feedforward Neural Networks (/understanding-feedforward-neural-networks/)
5. Image Classification using Feedforward Neural Networks (/image-classification-using-feedforward-

neural-network-in-keras/)
6. Image Recognition using Convolutional Neural Network
7. Understanding Activation Functions (/understanding-activation-functions-in-deep-learning/)
8. Understanding AutoEncoders using Tensorflow (/understanding-autoencoders-using-tensorflow-python/)
9. Image Classification using pre-trained models in Keras (/keras-tutorial-using-pre-trained-imagenet-models/)
10. Transfer Learning using pre-trained models in Keras (/keras-tutorial-transfer-learning-using-pre-trained-models/)
11. Fine-tuning pre-trained models in Keras (/keras-tutorial-fine-tuning-using-pre-trained-models)
12. More to come . . .

1. Motivation

In our previous article on Image Classification (/image-classification-using-feedforward-neural-network-in-keras/), we used a Multilayer Perceptron on the MNIST digits dataset. The performance was pretty good as we achieved 98.3% accuracy on test data. But there was a problem with that approach. In our training dataset, all images are centered. If the images in the test set are off-center, then the MLP approach fails miserably. We want the network to be **Translation-Invariant**.

Given below is an example of the number 7 being pushed to the top-left and bottom-right. The classifier predicts it correctly for the centered image but fails in the other two cases. To make it work for these images, either we have to train separate MLPs for different locations or we have to make sure that we have all these variations in the training set as well, which I would say is difficult, if not impossible.



(/wp-content/uploads/2017/11/failure-mlp-mnist.jpg)

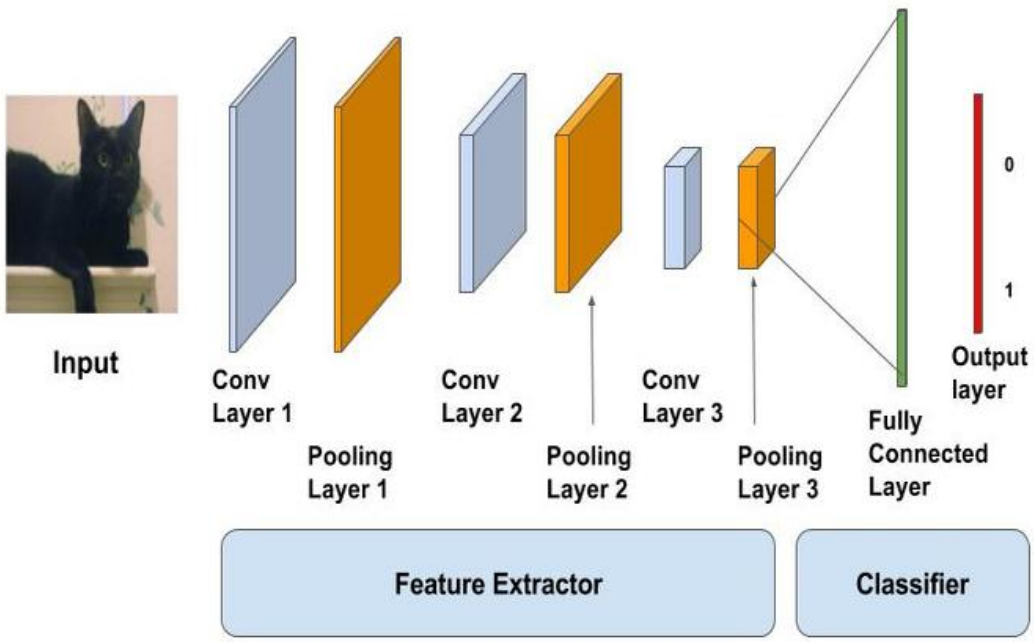
The Fully connected network tries to learn global features or patterns. It acts as a good classifier.

Another major problem with a fully connected classifier is that the number of parameters increases very fast since each node in layer L is connected to a node in layer L-1. So it is not feasible to design very deep networks using an MLP structure alone.

Both the above problems are solved to a great extent by using Convolutional Neural Networks which we will see in the next section. We will first describe the concepts involved in a Convolutional Neural Network in brief and then see an implementation of CNN in Keras so that you get a hands-on experience.

2. Convolutional Neural Network

Convolutional Neural Networks are a form of Feedforward Neural Networks (/understanding-feedforward-neural-networks/). Given below is a schema of a typical CNN. The first part consists of Convolutional and max-pooling layers which act as the feature extractor. The second part consists of the fully connected layer which performs non-linear transformations of the extracted features and acts as the classifier.



(https://www.learnopencv.com/wp-content/uploads/2017/11/cnn-schema1.jpg)

In the above diagram, the input is fed to the network of stacked Conv, Pool and Dense layers. The output can be a softmax layer indicating whether there is a cat or something else. You can also have a sigmoid layer to give you a probability of the image being a cat. Let us see the two layers in detail.

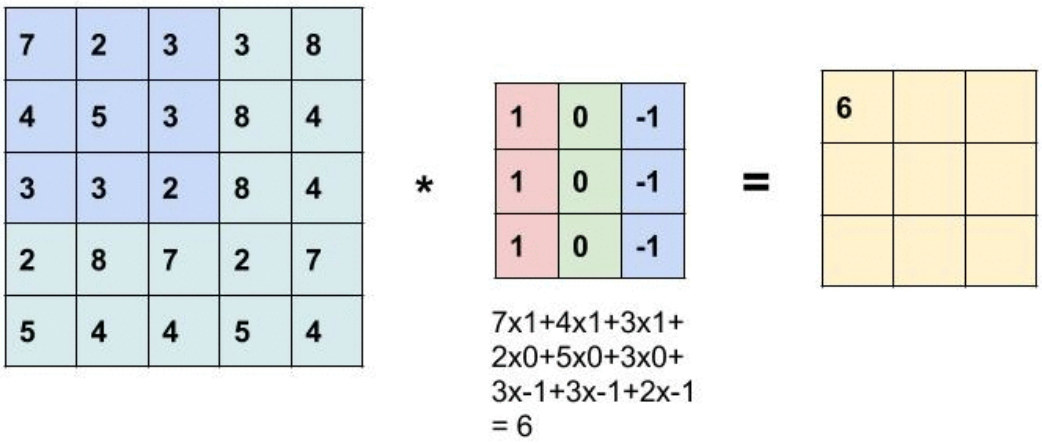
2.1. Convolutional Layer

The convolutional layer can be thought of as the eyes of the CNN. The neurons in this layer look for specific

features. If they find the features they are looking for, they produce a high activation.

Convolution can be thought of as a weighted sum between two signals (in terms of signal processing jargon) or functions (in terms of mathematics). In image processing, to calculate convolution at a particular location (x, y), we extract k x k sized chunk from the image centered at location (x, y). We then multiply the values in this chunk element-by-element with the convolution filter (also sized k x k) and then add them all to obtain a single output. That's it! Note that k is termed as the kernel size.

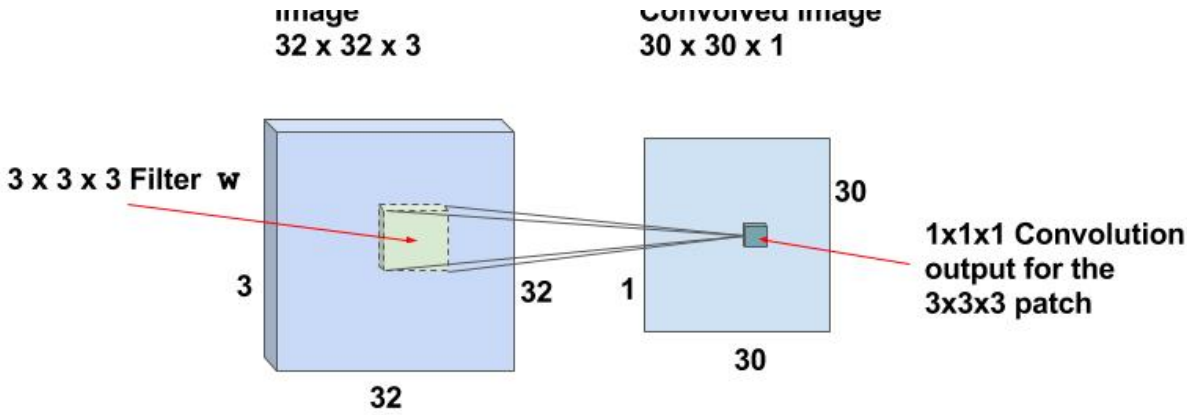
An example of convolution operation on a matrix of size 5×5 with a kernel of size 3×3 is shown below :



(/wp-content/uploads/2017/11/convolution-example-matrix.gif)
The convolution kernel is slid over the entire matrix to obtain an activation map.

Let's look at a concrete example and understand the terms. Suppose, the input image is of size 32x32x3. This is nothing but a 3D array of depth 3. Any convolution filter we define at this layer must have a depth equal to the depth of the input. So we can choose convolution filters of depth 3 (e.g. 3x3x3 or 5x5x3 or 7x7x3 etc.). Let's pick a convolution filter of size 3x3x3. So, referring to the above example, here the convolutional kernel will be a cube instead of a square.

Image Convolved Image



(/wp-content/uploads/2017/11/convolution-demo-diagram.jpg)

If we can perform the convolution operation by sliding the 3x3x3 filter over the entire 32x32x3 sized image, we will obtain an output image of size 30x30x1. This is because the convolution operation is not defined for a strip 2 pixels wide around the image. We have to ensure the filter is always inside the image. So 1 pixel is stripped away from left, right, top and bottom of the image.

> *The same filters are slid over the entire image to find the relevant features. This makes the CNNs Translation Invariant.*

2.1.1. Activation Maps

For a 32x32x3 input image and filter size of 3x3x3, we have 30x30x1 locations and there is a neuron corresponding to each location. Then 30x30x1 outputs or activations of all neurons are called the activation maps. The activation map of one layer serves as the input to the next layer.

2.1.2. Shared weights and biases

In our example, there are 30×30 = 900 neurons because there are that many locations where the 3x3x3 filter can be applied. Unlike traditional neural nets where weights and biases of neurons are independent of each other, in case of CNNs the neurons corresponding to one filter in a layer share the same weights and biases.

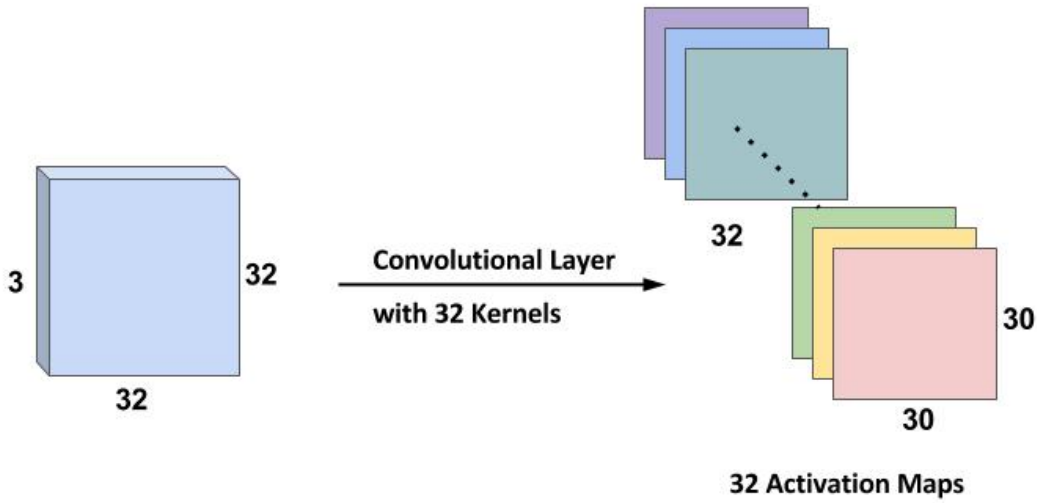
2.1.3. Stride

In the above case, we slid the window by 1 pixel at a time. We can also slide the window by more than 1 pixel. This number is called the stride.

2.1.4. Multiple Filters

Typically, we use more than 1 filter in one convolution layer. If we use 32 filters we will have an activation map of size 30x30x32. Please refer to Figure below for a graphical view.
Note that all neurons associated with the same filter share the same weights and biases. So the number of weights while using 32 filters is simply 3x3x3x32 = 288 and the number of biases is 32.

The 32 Activation maps obtained from applying the convoltional Kernels is shown below.



(/wp-content/uploads/2017/11/activation-maps-32-kernel.jpg)

2.1.5. Zero padding

As you can see, after each convolution, the output reduces in size (as in this case we are going from 32×32 to 30×30). For convenience, it's a standard practice to pad zeros to the boundary of the input layer such that the output is the same size as input layer. So, in this example, if we add a padding of size 1 on both sides of

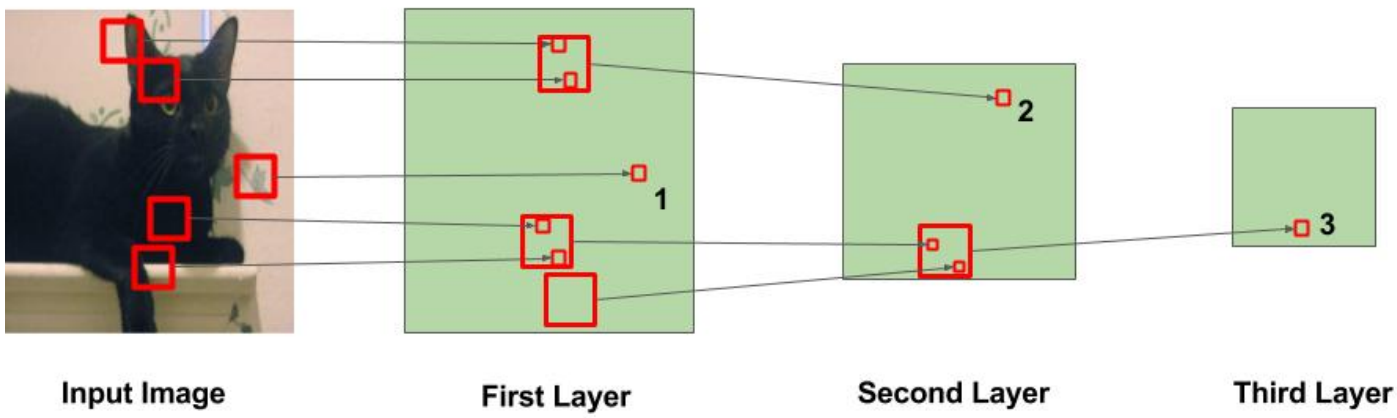
the input layer, the size of the output layer will be 32x32x32 which makes implementation simpler as well. Let's say you have an input of size NxN, a filter of size F and you are using stride S and a zero padding of size P is added to the input image. Then, the output will be of size MxM where,

$$M = \frac{N-F+2P}{S} + 1$$

We can calculate the padding required so that the input and the output dimensions are the same by setting in the above equation and solving for P.

2.2. CNNs learn Hierarchical features

Let's discuss how CNNs learn hierarchical features.



(/wp-content/uploads/2017/11/cnn-hierarchical-features.jpg)

In the above figure, the big squares indicate the region over which the convolution operation is performed and the small squares indicate the output of the operation which is just a number. The following observations are to be noted :

- In the first layer, the square marked 1 is obtained from the area in the image where the leaves are painted.
- In the second layer, the square marked 2 is obtained from the bigger square in Layer 1. The numbers in

this square are obtained from multiple regions from the input image. Specifically, the whole area around the left ear of the cat is responsible for the value at the square marked 2.

- Similarly, in the third layer, this cascading effect results in the square marked 3 being obtained from a large region around the leg area.

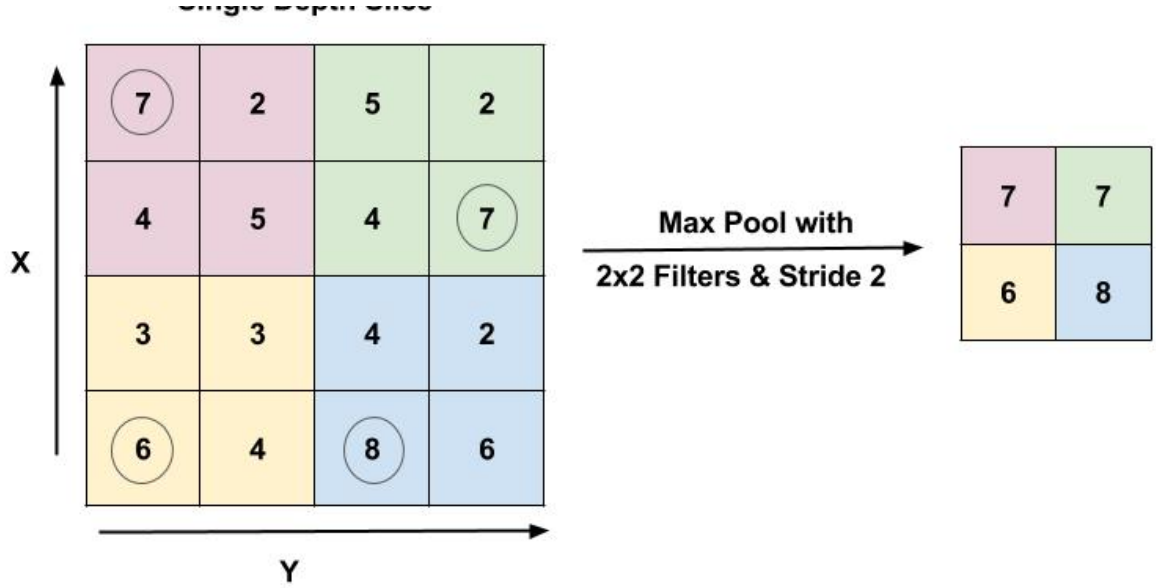
We can say from the above that the initial layers are looking at smaller regions of the image and thus can only learn simple features like edges / corners etc. As we go deeper into the network, the neurons get information from larger parts of the image and from various other neurons. Thus, the neurons at the later layers can learn more complicated features like eyes / legs and what not!

2.3. Max Pooling Layer

Pooling layer is mostly used immediately after the convolutional layer to reduce the spatial size (only width and height, not depth). This reduces the number of parameters, hence computation is reduced. Using fewer parameters avoids overfitting.

Note: Overfitting is the condition when a trained model works very well on training data, but does not work very well on test data.

The most common form of pooling is Max pooling where we take a filter of size p and apply the maximum operation over the sized part of the image.



(/wp-content/uploads/2017/11/max-pooling-demo.jpg)
Figure : Max pool layer with filter size 2×2 and stride 2 is shown. The output is the max value in a 2×2 region shown using encircled digits.

The most common pooling operation is done with the filter of size 2×2 with a stride of 2. It essentially reduces the size of input by half.

Now let's take a break from the theoretical discussion and jump into the implementation of a CNN.

3. Implementing CNNs in Keras

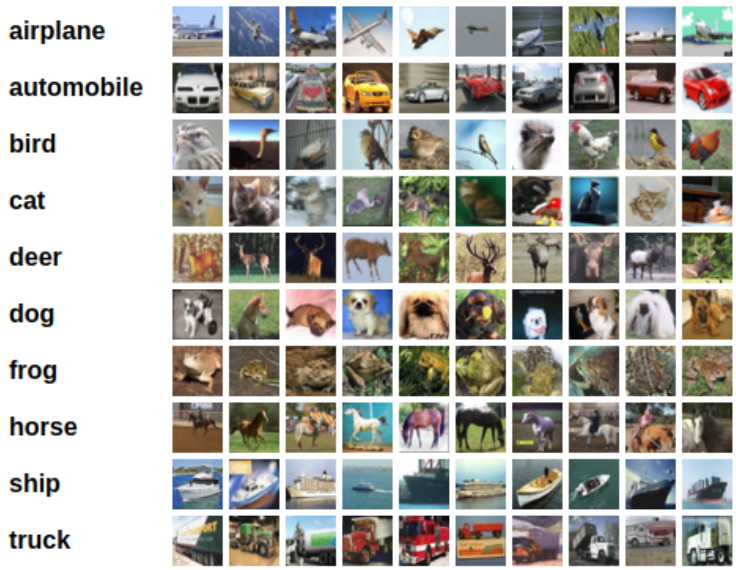
Download Code

To easily follow along this tutorial, please download code by clicking on the button below. It's FREE!

DOWNLOAD CODE
(HTTPS://BIGVISIONLLC.LEADPAGES.NET/LEADBOX/143948B73F72A2%3A173C9390C346DC/5649050225344512/)

3.1. The Dataset – CIFAR10

The CIFAR10 dataset comes bundled with Keras. It has 50,000 training images and 10,000 test images. There are 10 classes like airplanes, automobiles, birds, cats, deer, dog, frog, horse, ship and truck. The images are of size 32×32. Given below are a few examples.



(/wp-content/uploads/2017/11/cifar10-display-images.png)
Image Credit : Alex Krizhevsky

3.2. The Network

For implementing a CNN, we will stack up Convolutional Layers, followed by Max Pooling layers. We will also include Dropout to avoid overfitting. Finally, we will add a fully connected (Dense) layer followed by a softmax layer. Given below is the model structure.

```
1  from keras.models import Sequential
2  from keras.layers import Dense, Conv2D, MaxPooling2D, Dropout, Flatten
3
4  def createModel():
5      model = Sequential()
6      model.add(Conv2D(32, (3, 3), padding='same', activation='relu', input_shape=input_shape))
7      model.add(Conv2D(32, (3, 3), activation='relu'))
8      model.add(MaxPooling2D(pool_size=(2, 2)))
9      model.add(Dropout(0.25))
10
11     model.add(Conv2D(64, (3, 3), padding='same', activation='relu'))
12     model.add(Conv2D(64, (3, 3), activation='relu'))
13     model.add(MaxPooling2D(pool_size=(2, 2)))
14     model.add(Dropout(0.25))
15
16     model.add(Conv2D(64, (3, 3), padding='same', activation='relu'))
17     model.add(Conv2D(64, (3, 3), activation='relu'))
18     model.add(MaxPooling2D(pool_size=(2, 2)))
19     model.add(Dropout(0.25))
20
21     model.add(Flatten())
22     model.add(Dense(512, activation='relu'))
23     model.add(Dropout(0.5))
24     model.add(Dense(nClasses, activation='softmax'))
25
26     return model
```

In the above code, we use 6 convolutional layers and 1 fully-connected layer. Line 6 and 7 adds convolutional layers with 32 filters / kernels with a window size of 3×3. Similarly, in line 10, we add a conv layer with 64 filters. In line 8, we add a max pooling layer with window size 2×2. In line 9, we add a dropout

layer with a dropout ratio of 0.25. In the final lines, we add the dense layer which performs the classification among 10 classes using a softmax layer.

If we check the model summary we can see the shapes of each layer.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 32, 32, 32)        896
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 30, 30, 32)        9248
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 15, 15, 32)        0
_________________________________________________________________
dropout_1 (Dropout)          (None, 15, 15, 32)        0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 15, 15, 64)        18496
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 13, 13, 64)        36928
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 6, 6, 64)          0
_________________________________________________________________
dropout_2 (Dropout)          (None, 6, 6, 64)          0
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 6, 6, 64)          36928
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 4, 4, 64)          36928
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 2, 2, 64)          0
_________________________________________________________________
dropout_3 (Dropout)          (None, 2, 2, 64)          0
_________________________________________________________________
flatten_1 (Flatten)          (None, 256)               0
_________________________________________________________________
dense_1 (Dense)              (None, 512)               131584
_________________________________________________________________
dropout_4 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_2 (Dense)              (None, 10)                5130
=================================================================
Total params: 276,138
Trainable params: 276,138
Non-trainable params: 0
```

(/wp-content/uploads/2017/11/keras-cnn-cifar-model-summary.png)

It shows that since we have used padding in the first layer, the output shape is same as the input (32×32). But the second conv layer shrinks by 2 pixels in both dimensions. Also, the output size after pooling layer decreases by half since we have used a stride of 2 and a window size of 2×2. The final droupout layer has an output of 2x2x64. This has to be converted to a single array. This is done by the flatten layer which converts the 3D array into a 1D array of size 2x2x64 = 256. The final layer has 10 nodes since there are 10 classes.

3.3. Training the network

For training the network, we will follow the simple workflow of create -> compile -> fit described here (/deep-

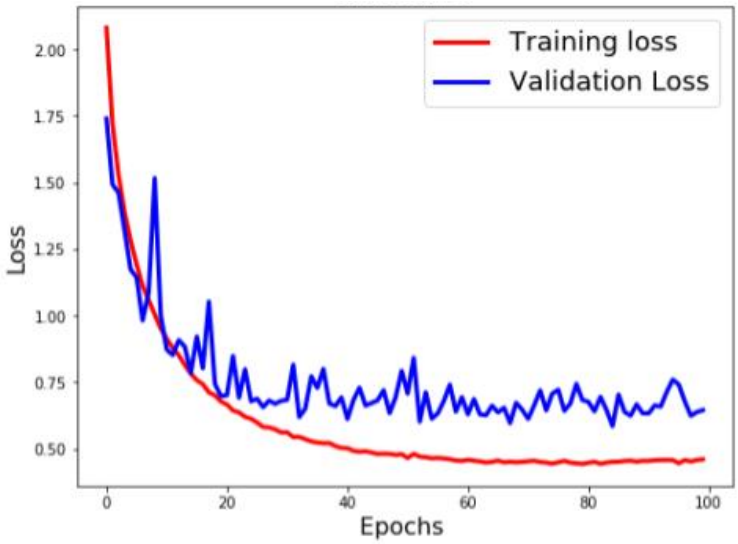
learning-using-keras-the-basics/). Since it is a 10 class classification problem, we will use a categorical cross entropy loss and use RMSProp optimizer to train the network. We will run it for some number of epochs. Here we run it for 100 epochs.

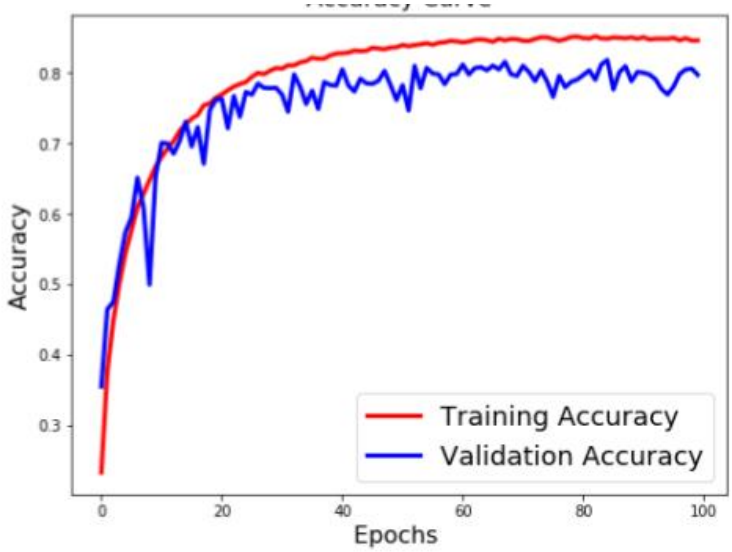
```
model1 = createModel()
batch_size = 256
epochs = 100
model1.compile(optimizer='rmsprop', loss='categorical_crossentropy', metrics=['accuracy'])

history = model1.fit(train_data, train_labels_one_hot, batch_size=batch_size, epochs=epochs, verbose=1,
                     validation_data=(test_data, test_labels_one_hot))

model1.evaluate(test_data, test_labels_one_hot)
```

3.4. Loss & Accuracy Curves

Given below are the loss and accuracy curves.

```
# Loss Curves
plt.figure(figsize=[8,6])
plt.plot(history.history['loss'],'r',linewidth=3.0)
plt.plot(history.history['val_loss'],'b',linewidth=3.0)
plt.legend(['Training loss', 'Validation Loss'],fontsize=18)
plt.xlabel('Epochs ',fontsize=16)
plt.ylabel('Loss',fontsize=16)
plt.title('Loss Curves',fontsize=16)

# Accuracy Curves
plt.figure(figsize=[8,6])
plt.plot(history.history['acc'],'r',linewidth=3.0)
plt.plot(history.history['val_acc'],'b',linewidth=3.0)
plt.legend(['Training Accuracy', 'Validation Accuracy'],fontsize=18)
plt.xlabel('Epochs ',fontsize=16)
plt.ylabel('Accuracy',fontsize=16)
plt.title('Accuracy Curves',fontsize=16)
```





(/wp-content/uploads/2017/11/cnn-keras-curves-without-aug.jpg)

From the above curves, we can see that there is a considerable difference between the training and validation loss. This indicates that the network has tried to memorize the training data and thus, is able to get better accuracy on it. This is a sign of Overfitting. But we have already used Dropout in the network, then why is it still overfitting. Let us see if we can further reduce overfitting using something else.

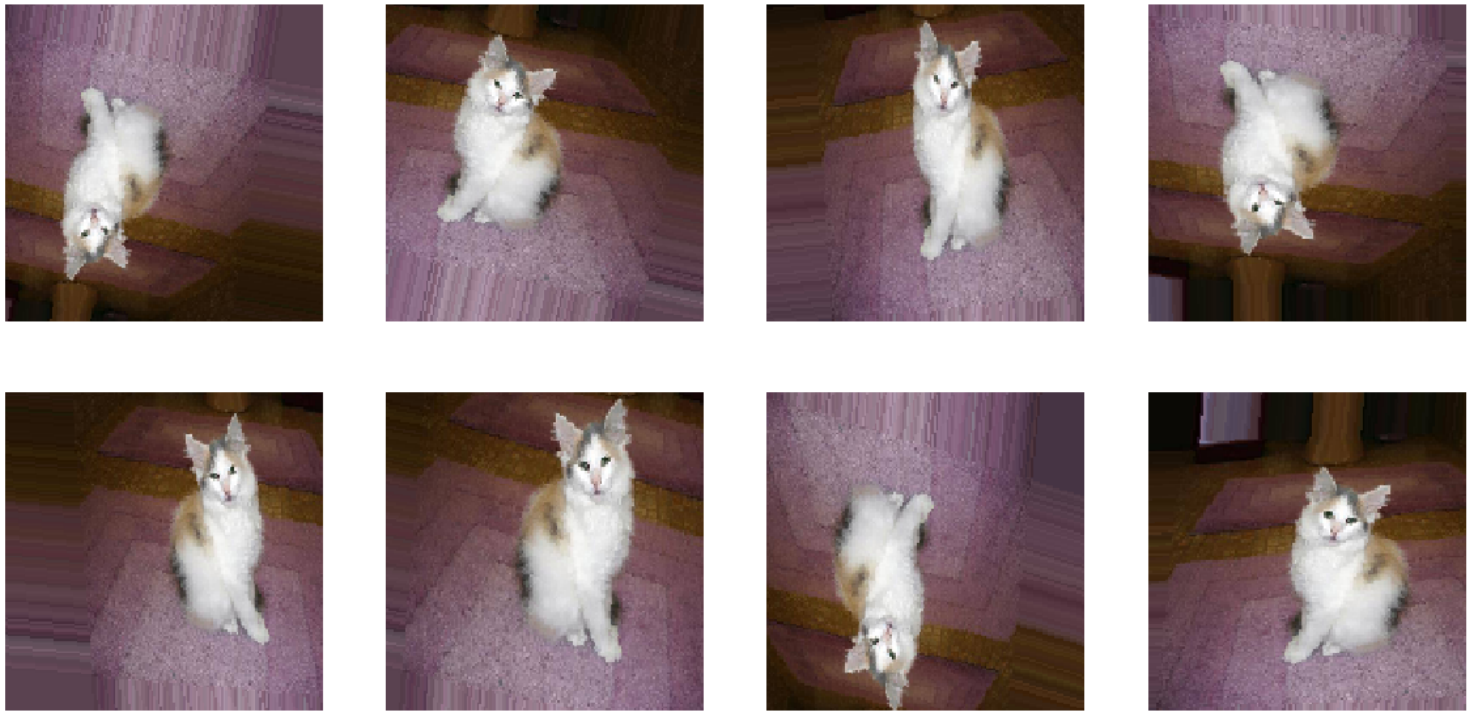
4. Using Data Augmentation

One of the major reasons for overfitting is that you don't have enough data to train your network. Apart from regularization, another very effective way to counter Overfitting is Data Augmentation. It is the process of artificially creating more images from the images you already have by changing the size, orientation etc of the image. It can be a tedious task but fortunately, this can be done in Keras using the ImageDataGenerator instance.

```
from keras.preprocessing.image import ImageDataGenerator

ImageDataGenerator(
    rotation_range=10.,
    width_shift_range=0.1,
    height_shift_range=0.1,
    shear_range=0.,
    zoom_range=.1.,
    horizontal_flip=True,
    vertical_flip=True)
```

In the above code, we have provided some of the operations that can be done using the ImageDataGenerator for data augmentation. This includes rotation of the image, shifting the image left/right/top/bottom by some amount, flip the image horizontally or vertically, shear or zoom the image etc.

For the complete list, check the documentation (https://keras.io/preprocessing/image/). Some generated images are shown below.



(/wp-content/uploads/2017/11/data-aug.png)

4.1. Training with Data Augmentation

Similar to the previous section, we will create the model, but use data augmentation while training. We will use ImageDataGenerator for creating a generator which will feed the network.

```
from keras.preprocessing.image import ImageDataGenerator

model2 = createModel()

model2.compile(optimizer='rmsprop', loss='categorical_crossentropy', metrics=['accuracy'])

batch_size = 256
```

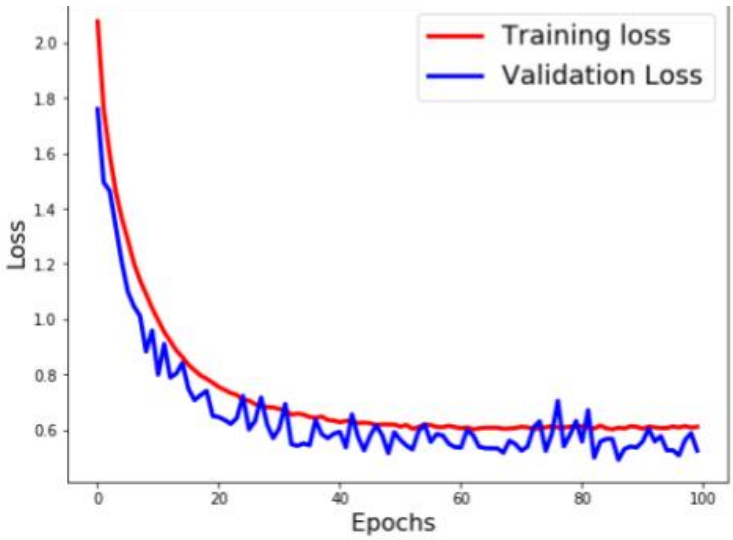
```
batch_size = 256
epochs = 100
datagen = ImageDataGenerator(
#         zoom_range=0.2, # randomly zoom into images
#         rotation_range=10,  # randomly rotate images in the range (degrees, 0 to 180)
        width_shift_range=0.1,  # randomly shift images horizontally (fraction of total width)
        height_shift_range=0.1,  # randomly shift images vertically (fraction of total height)
        horizontal_flip=True,  # randomly flip images
        vertical_flip=False)  # randomly flip images

# Fit the model on the batches generated by datagen.flow().
history2 = model2.fit_generator(datagen.flow(train_data, train_labels_one_hot, batch_size=batch_size),
                                steps_per_epoch=int(np.ceil(train_data.shape[0] / float(batch_size))),
                                epochs=epochs,
                                validation_data=(test_data, test_labels_one_hot),
                                workers=4)

model2.evaluate(test_data, test_labels_one_hot)
```

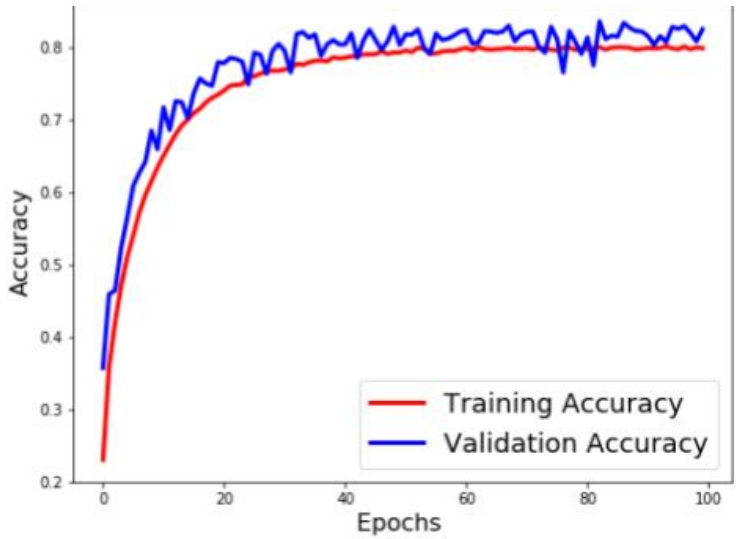
In the above code,

1. We first create the model and configure it.
2. Then we create an ImageDataGenerator object and configure it using parameters for horizontal flip, and image translation.
3. The datagen.flow() function generates batches of data, after performing the data transformations / augmentation specified during the instantiation of the data generator.
4. The fit_generator function will train the model using the data obtained in batches from the datagen.flow function.

4.2. Loss & Accuracy Curves

```
# Loss Curves
plt.figure(figsize=[8,6])
plt.plot(history2.history['loss'],'r',linewidth=3.0)
plt.plot(history2.history['val_loss'],'b',linewidth=3.0)
plt.legend(['Training loss', 'Validation Loss'],fontsize=18)
plt.xlabel('Epochs ',fontsize=16)
plt.ylabel('Loss',fontsize=16)
plt.title('Loss Curves',fontsize=16)

# Accuracy Curves
plt.figure(figsize=[8,6])
plt.plot(history2.history['acc'],'r',linewidth=3.0)
plt.plot(history2.history['val_acc'],'b',linewidth=3.0)
plt.legend(['Training Accuracy', 'Validation Accuracy'],fontsize=18)
plt.xlabel('Epochs ',fontsize=16)
plt.ylabel('Accuracy',fontsize=16)
plt.title('Accuracy Curves',fontsize=16)
```





(/wp-content/uploads/2017/11/cnn-keras-curves-with-aug.jpg)

The test accuracy is greater than training accuracy. This means that the model has generalized very well. This comes from the fact that the model has been trained on much worse data (for example – flipped images), so it is finding the normal test data easier to classify.

5. What next?

It looks like there were a lot of parameters to chose from and then training took a long time. We would not want to get tied down with these two problems when we are working on simple problems. Many researchers working in this field very generously open-source their trained models which have been trained on millions of images and for hundreds of hours on many GPUs. We can leverage their models and try to use their trained models as the starting point rather than starting from scratch. We will learn how to do Transfer Learning and Fine-tuning in our next post.

References

https://github.com/fchollet/keras/blob/master/examples
(https://github.com/fchollet/keras/blob/master/examples)

Subscribe & Download Code

If you liked this article and would like to download code and example images used in this post, please

subscribe (https://bigvisionllc.leadpages.net/leadbox/143948b73f72a2%3A173c9390c346dc/5649050225344512/) to our newsletter. You will also receive a free Computer Vision Resource (https://bigvisionllc.leadpages.net/leadbox/143948b73f72a2%3A173c9390c346dc/5649050225344512/) Guide. In our newsletter, we share OpenCV tutorials and examples written in C++/Python, and Computer Vision and Machine Learning algorithms and news.

Subscribe Now (https://bigvisionllc.leadpages.net/leadbox/143948b73f72a2%3A173c9390c346dc/5649050225344512/)

Learn OpenCV

Understanding Activation Functions in Deep Learning

OCTOBER 30, 2017 BY ADITYA SHARMA (HTTPS://WWW.LEARNOPENCV.COM/AUTHOR/ADITYASHARMA/)

This post is part of the series on Deep Learning for Beginners, which consists of the following tutorials :

1. Neural Networks : A 30,000 Feet View for Beginners (/neural-networks-a-30000-feet-view-for-beginners/)
2. Installation of Deep Learning frameworks (Tensorflow and Keras with CUDA support) (/installing-deep-learning-frameworks-on-ubuntu-with-cuda-support/)
3. Introduction to Keras (/deep-learning-using-keras-the-basics/)
4. Understanding Feedforward Neural Networks (/understanding-feedforward-neural-networks/)
5. Image Classification using Feedforward Neural Networks (/image-classification-using-feedforward-neural-network-in-keras/)
6. Image Recognition using Convolutional Neural Network (/image-classification-using-convolutional-neural-networks-in-keras/)
7. Understanding Activation Functions
8. Understanding AutoEncoders using Tensorflow (/understanding-autoencoders-using-tensorflow-python/)
9. Image Classification using pre-trained models in Keras (/keras-tutorial-using-pre-trained-imagenet-models/)
10. Transfer Learning using pre-trained models in Keras (/keras-tutorial-transfer-learning-using-pre-trained-models/)
11. Fine-tuning pre-trained models in Keras (/keras-tutorial-fine-tuning-using-pre-trained-models)
12. More to come . . .

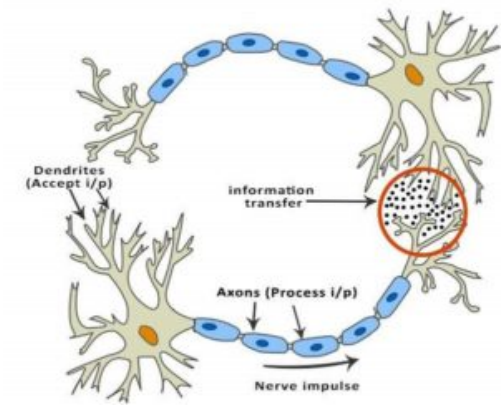
In this post, we will learn about different kinds of activation functions; we will also see which activation function is better than the other. This post assumes that you have a basic idea of Artificial Neural Networks (ANN), but in case you don't, I recommend you first read the post on understanding feedforward neural networks (/understanding-feedforward-neural-networks/).

1. What is an Activation Function?

Biological neural networks inspired the development of artificial neural networks. However, ANNs are not even an approximate representation of how the brain works. It is still useful to understand the relevance of an activation function in a biological neural network before we know as to why we use it in

an artificial neural network.

A typical neuron has a physical structure that consists of a cell body, an axon that sends messages to other neurons, and dendrites that receives signals or information from other neurons.



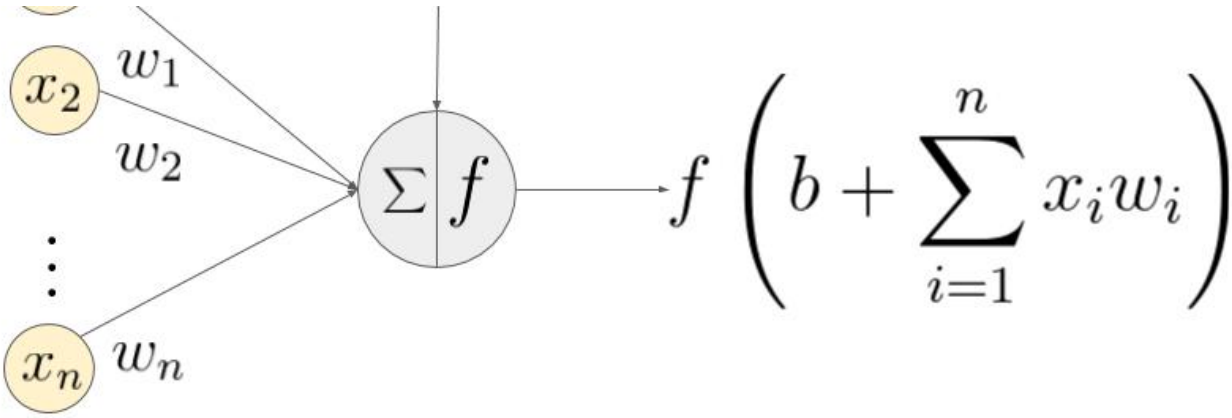
(/wp-content/uploads/2017/10/biological-neural-network.jpg)

Biological Neural Network (Image Credit (https://www.tutorialspoint.com/artificial_intelligence/artificial_intelligence_neural_networks

In the above picture, the red circle indicates the region where the two neurons communicate. The neuron receives signals from other neurons through the dendrites. The weight (strength) associated with a dendrite, called synaptic weights, gets multiplied by the incoming signal. The signals from the dendrites are accumulated in the cell body, and if the strength of the resulting signal is above a certain threshold, the neuron passes the message to the axon. Otherwise, the signal is killed by the neuron and is not propagated further.

The activation function takes the decision of whether or not to pass the signal. In this case, it is a simple step function with a single parameter – the threshold. Now, when we learn something new (or unlearn something), the threshold and the synaptic weights of some neurons change. This creates new connections among neurons making the brain learn new things.

Let us understand the same concept again but this time using an artificial neuron.



An example of a neuron showing the input (x_1 - x_n), their corresponding weights (w_1 - w_n), a bias (b) and the activation function f applied to the weighted sum of the inputs.

(/wp-content/uploads/2017/10/neuron-diagram.jpg)

In the above figure, $(x_1, \cdots, x_n)$ is the signal vector that gets multiplied with the weights $(w_1, w_2, \cdots, w_n)$. This is followed by accumulation (i.e. summation + addition of bias b). Finally, an activation function f is applied to this sum.

Note that the weights $(w_1, w_2, \cdots, w_n)$ and the bias b transform the input signal linearly. The activation, on the other hand, transforms the signal non-linearly and it is this non-linearity that allows us to learn arbitrarily complex transformations between the input and the output.

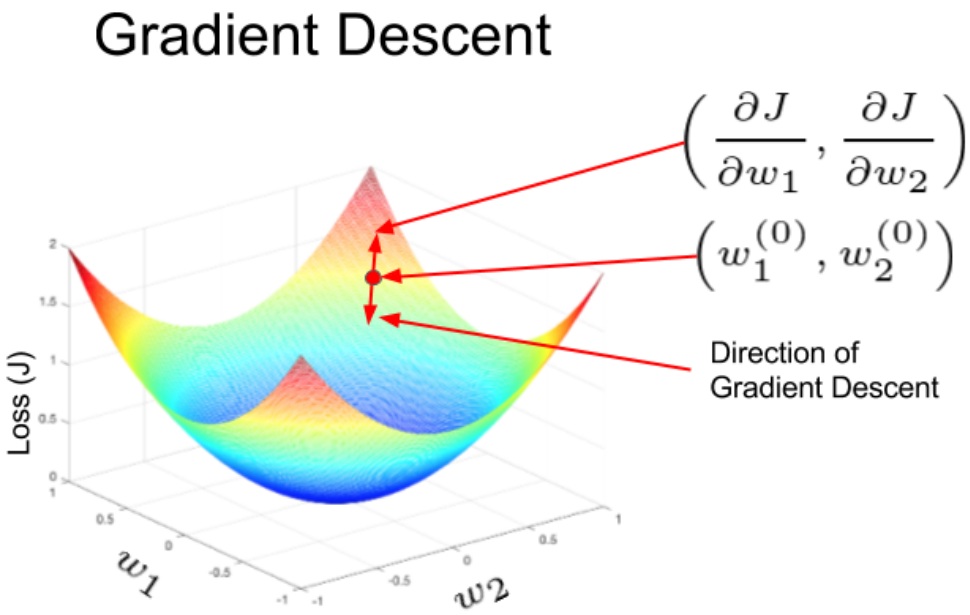
Over the years, various functions have been used, and it is still an active area of research to find a proper activation function that makes the neural network learn better and faster.

2. How does the network learn?

It is essential to get a basic idea of how the neural network learns. Let's say that the desired output of the network is y. The network produces an output y'. The difference between the predicted output and the desired output $(y - y)$ is converted to a metric known as the loss function (J). The loss is high when the neural network makes a lot of mistakes, and it is low when it makes fewer mistakes. The goal of the training process is to find the weights and bias that minimise the loss function over the training set.

In the figure below, the loss function is shaped like a bowl. At any point in the training process, the partial derivatives of the loss function w.r.t to the weights is nothing but the slope of the bowl at that location. One can see that by moving in the direction predicted by the partial derivatives, we can reach

the bottom of the bowl and therefore minimize the loss function. This idea of using the partial derivatives of a function to iteratively find its local minimum is called the **gradient descent**.



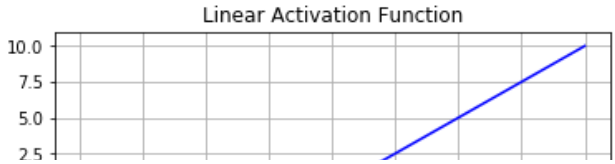
(/wp-content/uploads/2017/10/gradient-descent-2d-diagram.png)

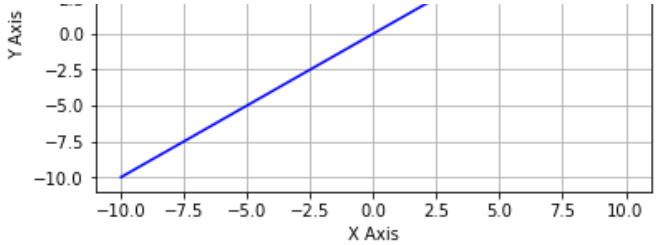
In Artificial neural networks the weights are updated using a method called **Backpropagation**. The partial derivatives of the loss function w.r.t the weights are used to update the weights. In a sense, the error is backpropagated in the network using derivatives. This is done in an iterative manner and after many iterations, the loss reaches a minimum value, and the derivative of the loss becomes zero.

We plan to cover backpropagation in a separate blog post. The main thing to note here is the presence of derivatives in the training process.

3. Types of Activation Functions

- Linear Activation Function: It is a simple linear function of the form $f(x) = x$Basically, the input passes to the output without any modification.

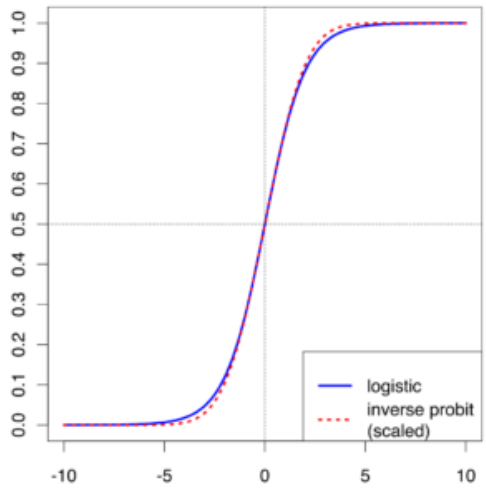




(/wp-content/uploads/2017/10/linear-activation-function-1.png)

Figure : Linear Activation Function

- Non-Linear Activation Functions: These functions are used to separate the data that is not linearly separable and are the most used activation functions.A non-linear equation governs the mapping from inputs to outputs. Few examples of different types of non-linear activation functions are sigmoid, tanh, relu, lrelu, prelu, swish, etc. We will be discussing all these activation functions in detail.



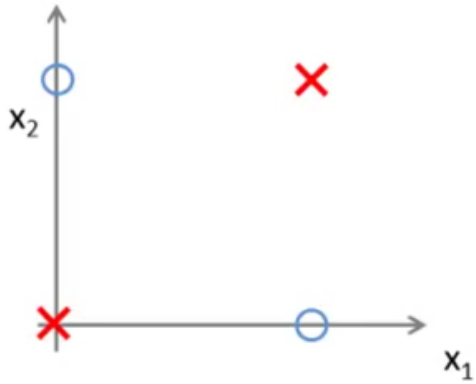
(/wp-content/uploads/2017/10/non-linear-activation-function.png)

Figure: Non-Linear Activation Function

4. Why do we need a non-linear activation function in an artificial neural network?

Neural networks are used to implement complex functions, and non-linear activation functions enable them to approximate arbitrarily complex functions. Without the non-linearity introduced by the activation function, multiple layers of a neural network are equivalent to a single layer neural network.

Let's see a simple example to understand why without non-linearity it is impossible to approximate even simple functions like XOR and XNOR gate. In the figure below, we graphically show an XOR gate. There are two classes in our dataset represented by a cross and a circle. When the two features, $X1$ and $X2$ are the same, the class label is a red cross, otherwise, it is a blue circle. The two red crosses have an output of 0 for input value (0,0) and (1,1) and the two blue rings have an output of 1 for input value (0,1) and (1,0).



(/wp-content/uploads/2017/10/xor.png)

Figure: Graphical Representation of XOR gate

From the above picture, we can see that the data points are not linearly separable. In other words, we can not draw a straight line to separate the blue circles and the red crosses from each other. Hence, we will need a non-linear decision boundary to separate them.

The activation function is also crucial for squashing the output of the neural network to be within certain bounds. The output of a neuron $\sum_i^n w_i x_i + b$ can take on very large values. This output, when fed to the next layer neuron without modification, can be transformed to even larger numbers thus making the process computationally intractable. One of the tasks of the activation function is to map the output of a neuron to something that is bounded (e.g., between 0 and 1).

With this background, we are ready to understand different types of activation functions.

5. Types of Non-Linear Activation Functions

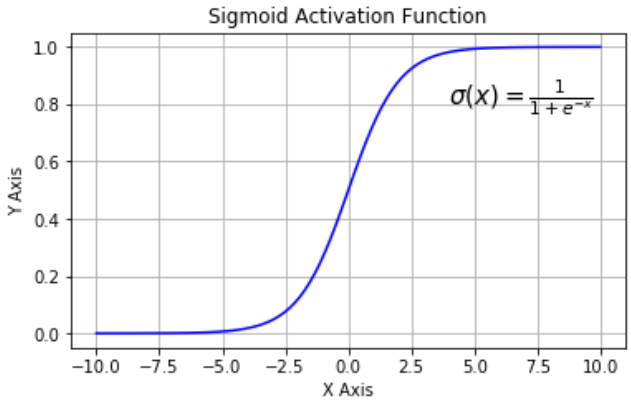
5.1. Sigmoid

It is also known as Logistic Activation Function. It takes a real-valued number and squashes it into a

range between 0 and 1. It is also used in the output layer where our end goal is to predict probability. It converts large negative numbers to 0 and large positive numbers to 1. Mathematically it is represented as

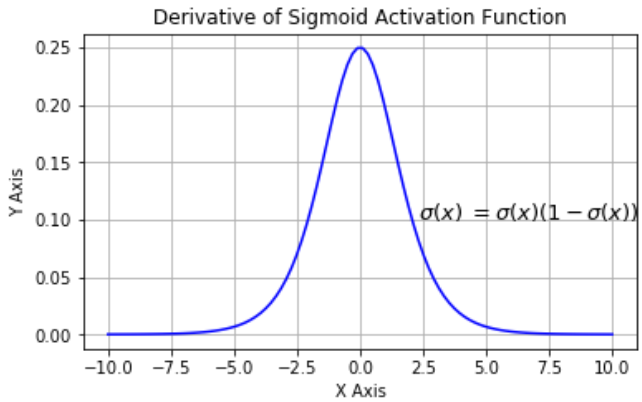
$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

The figure below shows the sigmoid function and its derivative graphically



(/wp-content/uploads/2017/10/sigmoid-activation-function.png)

Figure: Sigmoid Activation Function



(/wp-content/uploads/2017/10/sigmoid-derivative.png)

Figure: Sigmoid Derivative

The three major drawbacks of sigmoid are:

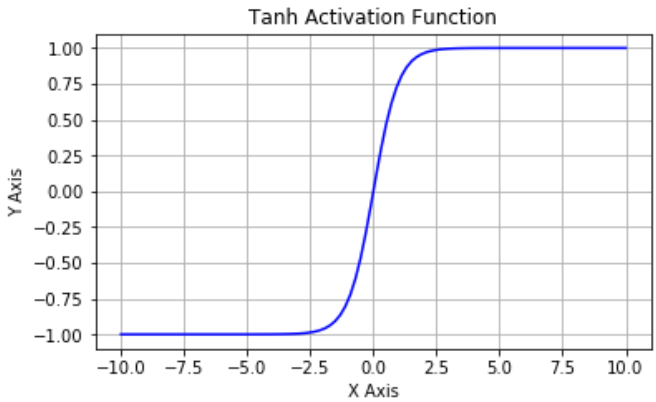
1. **Vanishing gradients**: Notice, the sigmoid function is flat near 0 and 1. In other words, the gradient

of the sigmoid is 0 near 0 and 1. During backpropagation through the network with sigmoid activation, the gradients in neurons whose output is near 0 or 1 are nearly 0. These neurons are called saturated neurons. Thus, the weights in these neurons do not update. Not only that, the weights of neurons connected to such neurons are also slowly updated. This problem is also known as vanishing gradient. So, imagine if there was a large network comprising of sigmoid neurons in which many of them are in a saturated regime, then the network will not be able to backpropagate.

2. **Not zero centered**: Sigmoid outputs are not zero-centered.
3. **Computationally expensive**: The exp() function is computationally expensive compared with the other non-linear activation functions.

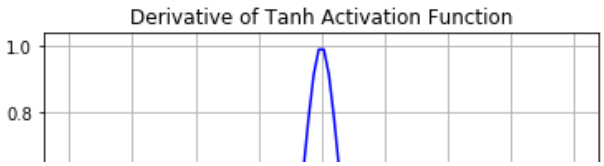
The next non-linear activation function that I am going to discuss addresses the zero-centered problem in sigmoid.

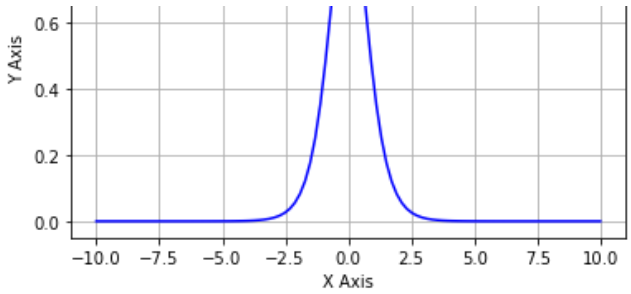
5.2. Tanh



(/wp-content/uploads/2017/10/tanh-1.png)

Figure: Tanh Activation Function





(/wp-content/uploads/2017/10/tanh-derivative.png)

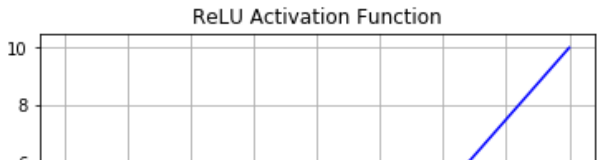
Figure: Tanh Derivative

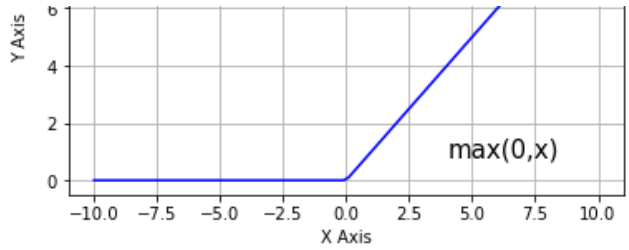
It is also known as the hyperbolic tangent activation function. Similar to sigmoid, tanh also takes a real-valued number but squashes it into a range between -1 and 1. Unlike sigmoid, tanh outputs are zero-centered since the scope is between -1 and 1. You can think of a tanh function as two sigmoids put together. In practice, tanh is preferable over sigmoid. The negative inputs considered as strongly negative, zero input values mapped near zero, and the positive inputs regarded as positive. The only drawback of tanh is:

1. The tanh function also suffers from the vanishing gradient problem and therefore kills gradients when saturated.

To address the vanishing gradient problem, let us discuss another non-linear activation function known as the rectified linear unit (ReLU) which is a lot better than the previous two activation functions and is most widely used these days.

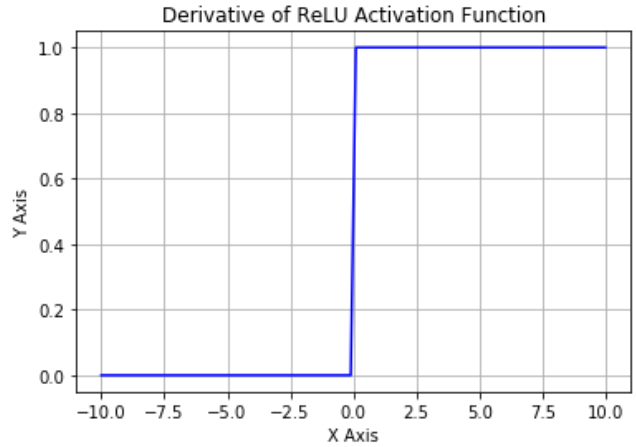
5.3. Rectified Linear Unit (ReLU)





(/wp-content/uploads/2017/10/relu-activation-function-1.png)

Figure: ReLU Activation Function



(/wp-content/uploads/2017/10/relu-derivative.png)

Figure: ReLU Derivative

ReLU is half-rectified from the bottom as you can see from the figure above. Mathematically, it is given by this simple expression

$$f(x) = \max(0, x)$$

This means that when the input x < 0 the output is 0 and if x > 0 the output is x. This activation makes the network converge much faster. It does not saturate which means it is resistant to the vanishing gradient problem at least in the positive region (when x > 0), so the neurons do not backpropagate all zeros at least in half of their regions. ReLU is computationally very efficient because it is implemented using simple thresholding. But there are few drawbacks of ReLU neuron :

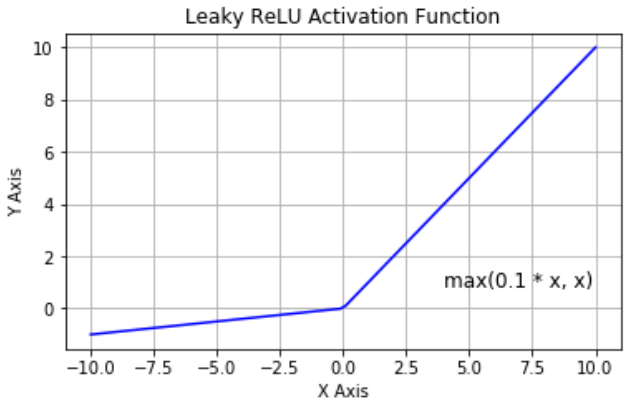
1. Not zero-centered: The outputs are not zero centered similar to the sigmoid activation function.

2. The other issue with ReLU is that if x < 0 during the forward pass, the neuron remains inactive and it kills the gradient during the backward pass. Thus weights do not get updated, and the network does not learn. When x = 0 the slope is undefined at that point, but this problem is taken care of

during implementation by picking either the left or the right gradient.

To address the vanishing gradient issue in ReLU activation function when x < 0 we have something called Leaky ReLU which was an attempt to fix the dead ReLU problem. Let's understand leaky ReLU in detail.

5.4. Leaky ReLU



(/wp-content/uploads/2017/10/leaky-relu-activation.png)

Figure : Leaky ReLU activation function

This was an attempt to mitigate the dying ReLU problem. The function computes

$$f(x) = max(0.1x, x)$$

The concept of leaky ReLU is when x < 0, it will have a small positive slope of 0.1. This function somewhat eliminates the dying ReLU problem, but the results achieved with it are not consistent. Though it has all the characteristics of a ReLU activation function, i.e., computationally efficient, converges much faster, does not saturate in positive region.

The idea of leaky ReLU can be extended even further. Instead of multiplying x with a constant term we can multiply it with a hyperparameter which seems to work better the leaky ReLU. This extension to leaky ReLU is known as Parametric ReLU.

5.5. Parametric ReLU

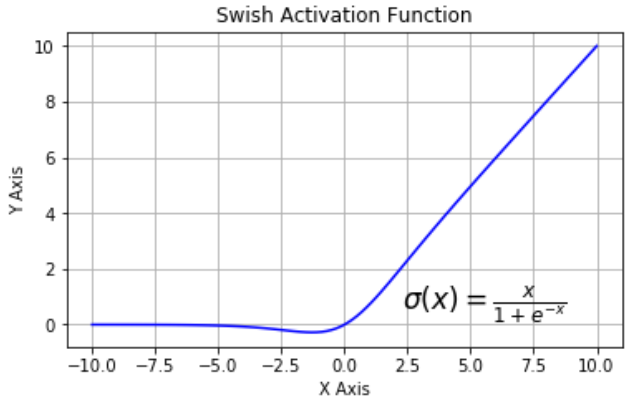
The PReLU function is given by

$$f(x) = \max(\alpha x, x)$$

Where α is a hyperparameter. The idea here was to introduce an arbitrary hyperparameter α, and this α can be learned since you can backpropagate into it. This gives the neurons the ability to choose what slope is best in the negative region, and with this ability, they can become a ReLU or a leaky ReLU.

In summary, it is better to use ReLU, but you can experiment with Leaky ReLU or Parametric ReLU to see if they give better results for your problem

5.6. SWISH



(/wp-content/uploads/2017/10/swish.png)

Figure: SWISH Activation Function

Also known as a self-gated activation function, has recently been released by researchers at Google. Mathematically it is represented as

$$\sigma(x) = \frac{x}{1+e^{-x}}$$

According to the paper (https://arxiv.org/abs/1710.05941v1), the SWISH activation function performs better than ReLU

From the above figure, we can observe that in the negative region of the x-axis the shape of the tail is different from the ReLU activation function and because of this the output from the Swish activation function may decrease even when the input value increases. Most activation functions are monotonic,

i.e., their value never decreases as the input increases. Swish has one-sided boundedness property at zero, it is smooth and is non-monotonic. It will be interesting to see how well it performs by changing just one line of code.

Subscribe & Download Code

If you liked this article and would like to download code (C++ and Python) and example images used in other posts of this blog, please subscribe (https://bigvisionllc.leadpages.net/leadbox/143948b73f72a2%3A173c9390c346dc/5649050225344512/) to our newsletter. You will also receive a free Computer Vision Resource (https://bigvisionllc.leadpages.net/leadbox/143948b73f72a2%3A173c9390c346dc/5649050225344512/) Guide. In our newsletter, we share OpenCV tutorials and examples written in C++/Python, and Computer Vision and Machine Learning algorithms and news.

Subscribe Now (https://bigvisionllc.leadpages.net/leadbox/143948b73f72a2%3A173c9390c346dc/5649050225344512/)

Learn OpenCV

Understanding Autoencoders using Tensorflow (Python)

NOVEMBER 15, 2017 BY ADITYA SHARMA (HTTPS://WWW.LEARNOPENCV.COM/AUTHOR/ADITYASHARMA/)

This post is part of the series on Deep Learning for Beginners, which consists of the following tutorials :

1. Neural Networks : A 30,000 Feet View for Beginners (/neural-networks-a-30000-feet-view-for-beginners/)
2. Installation of Deep Learning frameworks (Tensorflow and Keras with CUDA support) (/installing-deep-learning-frameworks-on-ubuntu-with-cuda-support/)
3. Introduction to Keras (/deep-learning-using-keras-the-basics/)
4. Understanding Feedforward Neural Networks (/understanding-feedforward-neural-networks/)
5. Image Classification using Feedforward Neural Networks (/image-classification-using-feedforward-neural-network-in-keras/)
6. Image Recognition using Convolutional Neural Network (/image-classification-using-convolutional-neural-networks-in-keras/)
7. Understanding Activation Functions (/understanding-activation-functions-in-deep-learning/)
8. Understanding AutoEncoders using Tensorflow
9. Image Classification using pre-trained models in Keras (/keras-tutorial-using-pre-trained-imagenet-models/)
10. Transfer Learning using pre-trained models in Keras (/keras-tutorial-transfer-learning-using-pre-trained-models/)
11. Fine-tuning pre-trained models in Keras (/keras-tutorial-fine-tuning-using-pre-trained-models)
12. More to come . . .

In this article, we will learn about autoencoders in deep learning. We will show a practical implementation of using a Denoising Autoencoder on the MNIST (https://en.wikipedia.org/wiki/MNIST_database) handwritten digits dataset as an example. In addition, we are sharing an implementation of the idea in Tensorflow.

1. What is an autoencoder?

An autoencoder is an unsupervised machine learning algorithm that takes an image as input and reconstructs it using fewer number of bits. That may sound like image compression, but the biggest difference between an autoencoder and a general purpose image compression algorithms is that in case of

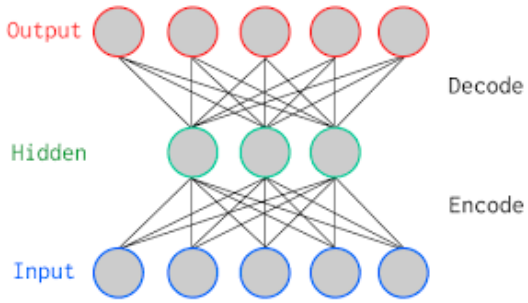
autoencoders, the compression is achieved by learning on a training set of data. While reasonable compression is achieved when an image is similar to the training set used, autoencoders are poor general-purpose image compressors; JPEG compression will do vastly better.

Autoencoders are similar in spirit to dimensionality reduction techniques like principal component analysis (https://en.wikipedia.org/wiki/Principal_component_analysis). They create a space where the essential parts of the data are preserved, while non-essential (or noisy) parts are removed.

There are two parts to an autoencoder

1. **Encoder**: This is the part of the network that compresses the input into a fewer number of bits. The space represented by these fewer number of bits is called the “latent-space” and the point of maximum compression is called the bottleneck. These compressed bits that represent the original input are together called an “encoding” of the input.
2. **Decoder**: This is the part of the network that reconstructs the input image using the encoding of the image.

Let’s look at an example to understand the concept better.



(/wp-content/uploads/2017/11/AutoEncoder.png)

Figure: 2-layer Autoencoder

In the above picture, we show a vanilla autoencoder — a 2-layer autoencoder with one hidden layer. The input and output layers have the same number of neurons. We feed five real values into the autoencoder which is compressed by the encoder into three real values at the bottleneck (middle layer). Using these three real values, the decoder tries to reconstruct the five real values which we had fed as an input to the network.

In practice, there are a far larger number of hidden layers in between the input and the output.

There are various kinds of autoencoders like sparse autoencoder, variational autoencoder, and denoising autoencoder. In this post, we will learn about a denoising autoencoder.

2. Denoising Autoencoder

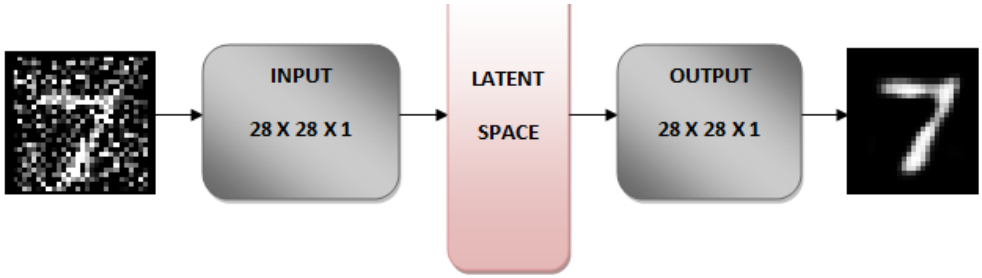


Figure: Denoising Autoencoder

The idea behind a denoising autoencoder is to learn a representation (latent space) that is robust to noise. We add noise to an image and then feed this noisy image as an input to our network. The encoder part of the autoencoder transforms the image into a different space that preserves the handwritten digits but removes the noise. As we will see later, the original image is 28 x 28 x 1 image, and the transformed image is 7 x 7 x 32. You can think of the 7 x 7 x 32 image as a 7 x 7 image with 32 color channels.

The decoder part of the network then reconstructs the original image from this 7 x 7 x 32 image and voila the noise is gone!

How does this magic happen?

During training, we define a loss (cost function) to minimize the difference between the reconstructed image and the original noise-free image. In other words, we learn a 7 x 7 x 32 space that is noise free.

Download Code

To easily follow along this tutorial, please download the iPython notebook code by clicking on the button below. It's FREE!

DOWNLOAD CODE
(HTTPS://BIGVISIONLLC.LEADPAGES.NET/LEADBOX/143948B73F72A2%3A173C9390C346DC/5649050225344512/)

3. Implementation of Denoising Autoencoder

This implementation is inspired by this excellent post Building Autoencoders in Keras (https://blog.keras.io/building-autoencoders-in-keras.html).

3.1 The Network

The images are matrices of size 28 x 28. We reshape the image to be of size 28 x 28 x 1, convert the resized

image matrix to an array, rescale it between 0 and 1, and feed this as an input to the network. The encoder transforms the 28 x 28 x 1 image to a 7 x 7 x 32 image. You can think of this 7 x 7 x 32 image as a point in a 1568 (because 7 x 7 x 32 = 1568) dimensional space. This 1568 dimensional space is called the bottleneck or the latent space. The architecture is graphically shown below.

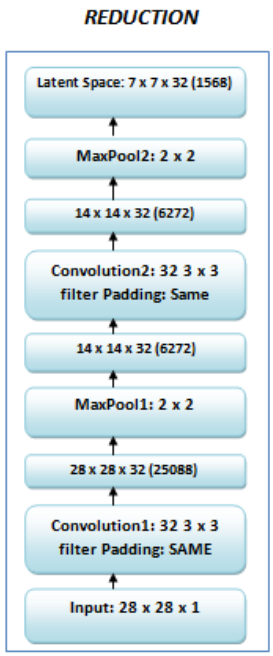
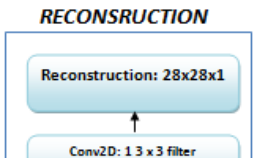


Figure: Architecture of Encoder Model

The decoder does the exact opposite of an encoder; it transforms this 1568 dimensional vector back to a 28 x 28 x 1 image. We call this output image a “reconstruction” of the original image. The structure of the decoder is shown below.



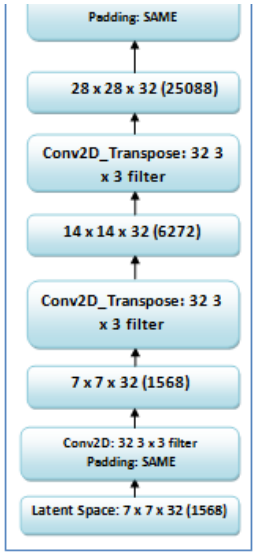


Figure: Architecture of Decoder Model

Let's dive into the implementation of an autoencoder using tensorflow.

3.2 Encoder

The encoder has two convolutional layers and two max pooling layers. Both Convolution layer-1 and Convolution layer-2 have 32-3 x 3 filters. There are two max-pooling layers each of size 2 x 2.

```
#Encoder
with tf.name_scope('en-convolutions'):
 conv1 = tf.layers.conv2d(inputs_,filters=32,kernel_size=(3,3),strides=(1,1),padding='SAME',use_bias=True,activation=lr
# Now 28x28x32

with tf.name_scope('en-pooling'):
 maxpool1 = tf.layers.max_pooling2d(conv1,pool_size=(2,2),strides=(2,2),name='pool1')
# Now 14x14x32

with tf.name_scope('en-convolutions'):
 conv2 = tf.layers.conv2d(maxpool1,filters=32,kernel_size=(3,3),strides=(1,1),padding='SAME',use_bias=True,activation=l
# Now 14x14x32

with tf.name_scope('encoding'):
 encoded = tf.layers.max_pooling2d(conv2,pool_size=(2,2),strides=(2,2),name='encoding')
# Now 7x7x32.

#latent space
```

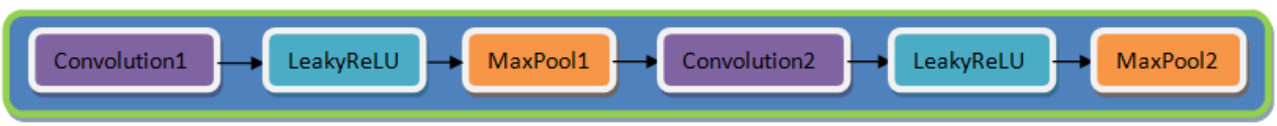


Figure: Encoder Block Diagram

3.3 Decoder

The decoder has two Conv2d_transpose layers, two Convolution layers, and one Sigmoid activation function.

Conv2d_transpose is for upsampling which is opposite to the role of a convolution layer. The Conv2d_transpose layer upsamples the compressed image by two times each time we use it.

```
#Decoder
with tf.name_scope('decoder'):
 conv3 = tf.layers.conv2d(encoded,filters=32,kernel_size=(3,3),strides=(1,1),name='conv3',padding='SAME',use_bias=True,
#Now 7x7x32

 upsample1 = tf.layers.conv2d_transpose(conv3,filters=32,kernel_size=3,padding='same',strides=2,name='upsample1')
# Now 14x14x32

 upsample2 = tf.layers.conv2d_transpose(upsample1,filters=32,kernel_size=3,padding='same',strides=2,name='upsample2')
# Now 28x28x32

 logits = tf.layers.conv2d(upsample2,filters=1,kernel_size=(3,3),strides=(1,1),name='logits',padding='SAME',use_bias=Tr
#Now 28x28x1

# Pass logits through sigmoid to get denoisy image
 decoded = tf.sigmoid(logits,name='recon')
```

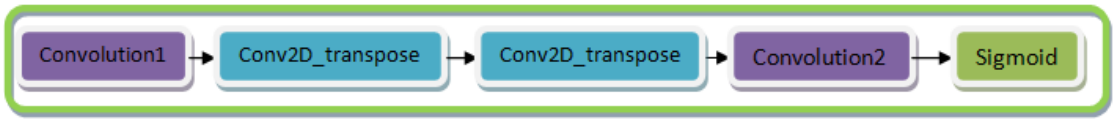


Figure: Decoder Block Diagram

Finally, we calculate the loss of the output using cross-entropy (https://en.wikipedia.org/wiki/Cross_entropy) loss function and use Adam optimizer (https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/) to optimize our loss function.

3.4 Why do we use a leaky ReLU and not a ReLU as an activation function?

We want gradients to flow while we backpropagate through the network. We stack many layers in a system in which there are some neurons whose value drop to zero or become negative. Using a ReLU as an activation function clips the negative values to zero and in the backward pass, the gradients do not flow through those neurons where the values become zero. Because of this the weights do not get updated, and the network stops learning for those values. So using ReLU is not always a good idea. However, we encourage you to change the activation function to ReLU and see the difference.

```
def lrelu(x,alpha=0.1):
  return tf.maximum(alpha*x,x)
```

Therefore, we use a leaky ReLU which instead of clipping the negative values to zero, cuts them to a specific amount based on a hyperparameter alpha. This ensures that the network learns something even when the pixel value is below zero.

3.5 Load the data

Once the architecture has been defined, we load the training and validation data.

As shown below, Tensorflow allows us to easily load the MNIST data. The training and testing data loaded is stored in variables train_X and test_X respectively. Since its an unsupervised task we do not care about the labels.

```
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)
train_X = mnist.train.images
test_X = mnist.test.images
```

3.6 Data Analysis

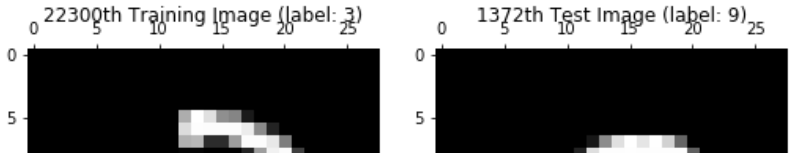
Before training a neural network, it is always a good idea to do a sanity check on the data.

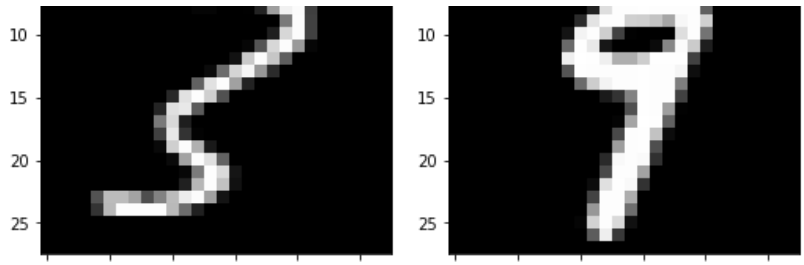
Let's see how the data looks like. The data consists of handwritten numbers ranging from 0 to 9, along with their ground truth labels. It has 55,000 train samples and 10,000 test samples. Each sample is a 28×28 grayscale image.

```
print('Training data shape' :train_X.shape)
print('Testing data shape' :test_X.shape)

nsample = 1
rand_train_idx = np.random.randint(mnist.train.images.shape[0], size=nsample)

for i in rand_train_idx:
  curr_img = np.reshape(mnist.train.images[i, :], (28,28))
  curr_lbl = np.argmax(mnist.train.labels[i, :])
  plt.matshow(curr_img, cmap=plt.get_cmap('gray'))
  plt.title(""+str(i)+"th Training Image "+ "(label: " + str(curr_lbl) + ")")
  plt.show()

rand_test_idx = np.random.randint(mnist.test.images.shape[0], size=nsample)

for i in rand_test_idx:
  curr_img = np.reshape(mnist.test.images[i, :], (28,28))
  curr_lbl = np.argmax(mnist.test.labels[i, :])
  plt.matshow(curr_img, cmap=plt.get_cmap('gray'))
  plt.title(""+str(i)+"th Test Image "+ "(label: " + str(curr_lbl) + ")")
  plt.show()
```

Output:

```
(Training data shape :&amp;nbsp; (55000, 784))
(Testing data shape :&amp;nbsp; (10000, 784))
```





3.7 Preprocessing the data

The images are grayscale and the pixel values range from 0 to 255. We apply following preprocessing to the data before feeding it to the network.

1. Convert each 784-dimensional vector into a matrix of size 28 x 28 x 1 which is fed into the network.

```
batch_train_x = mnist.train.next_batch(batch_size)
batch_test_x = mnist.test.next_batch(batch_size)
imgs_train= batch_train_x[0].reshape((-1, 28, 28, 1))
imgs_test = batch_test_x[0].reshape((-1, 28, 28, 1))

```

1. Add noise to both train and test images which we then feed into the network. Noise factor is a hyperparamter and can be tuned accordingly.

```
noise_factor = 0.5
x_train_noisy = imgs_train + noise_factor * np.random.normal(loc=0.0, scale=1.0, size=imgs_train.shape)
x_test_noisy = imgs_test + noise_factor * np.random.normal(loc=0.0, scale=1.0, size=imgs_test.shape)
x_train_noisy = np.clip(x_train_noisy, 0., 1.)
x_test_noisy = np.clip(x_test_noisy, 0., 1.)
```

3.8 Training the model

The network is ready to get trained. We specify the number of epochs as 25 with batch size of 64. This means that the whole dataset will be fed to the network 25 times. We will be using the test data for validation.

```
batch_cost, _ = sess.run([cost, opt], feed_dict={inputs_: x_train_noisy,targets_: imgs,learning_rate:lr})
```

3.9 Evaluate the model

We check the performance of the model on our test set by checking the cost (loss).

```
batch_cost_test = sess.run(cost, feed_dict={inputs_: x_test_noisy,targets_: imgs_test})
```

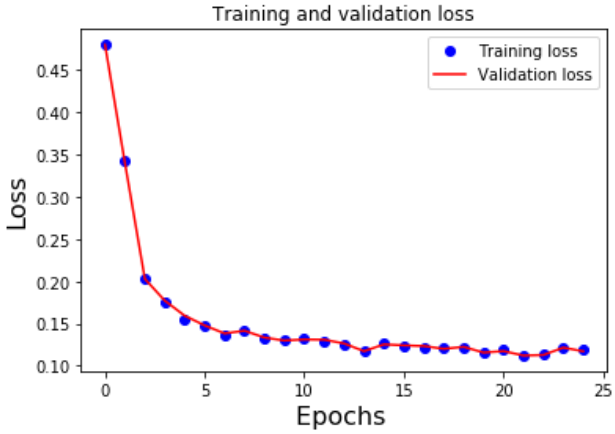
Output

```
('Epoch: 25/25...', 'Training loss: 0.1196', 'Validation loss: 0.1171')
```

After 25 epochs we can see our training loss and validation loss is quite low which means our network did a pretty good job. Let's now see the loss plot between training and validation data.

3.10 Training Vs. Validation Loss Plot

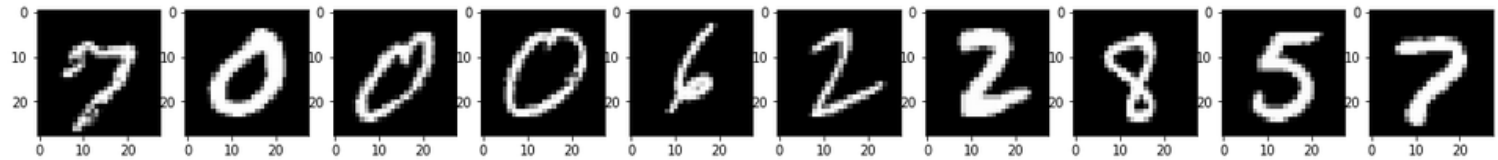
```
loss.append(batch_cost)
valid_loss.append(batch_cost_test)
plt.plot(range(e+1), loss, 'bo', label='Training loss')
plt.plot(range(e+1), valid_loss, 'r', label='Validation loss')
plt.title('Training and validation loss')
plt.xlabel('Epochs ',fontsize=16)
plt.ylabel('Loss',fontsize=16)
plt.legend()
plt.figure()
plt.show()
```



From the above loss plot, we can observe that the validation loss and training loss are both steadily decreasing in the first ten epochs. This training loss and the validation loss are also very close to each other. This means that our model has generalized well to unseen test data.

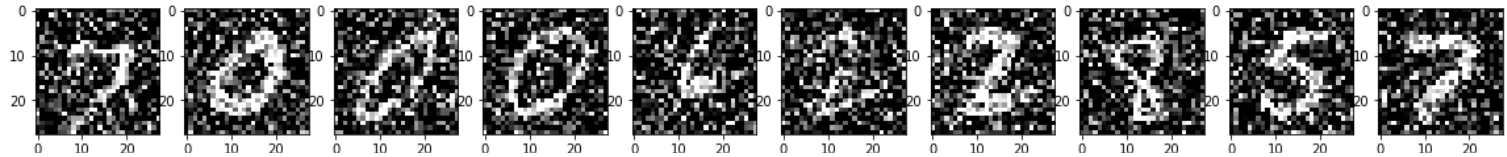
We can further validate our results by observing the original, noisy and reconstruction of test images.

3.11 Results



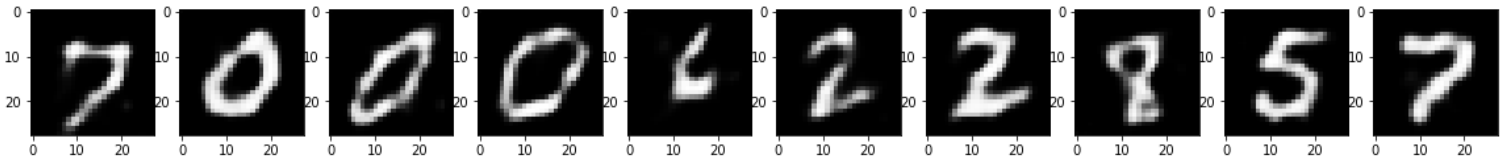
(/wp-content/uploads/2017/11/original-mnist-2.png)

Figure: Original Images



(/wp-content/uploads/2017/11/noisy-mnist-1.png)

Figure: Images with Noise



(/wp-content/uploads/2017/11/reconstruction-mnist-1.png)

Figure: Reconstruction of Noisy Images

From the above figures, we can observe that our model did a good job in denoising the noisy images that we had fed into our model.

Subscribe & Download Code

If you liked this article and would like to download code (iPython notebook), please subscribe (https://bigvisionllc.leadpages.net/leadbox/143948b73f72a2%3A173c9390c346dc/5649050225344512/) to our newsletter. You will also receive a free Computer Vision Resource (https://bigvisionllc.leadpages.net/leadbox/143948b73f72a2%3A173c9390c346dc/5649050225344512/) Guide. In our newsletter, we share OpenCV tutorials and examples written in C++/Python, and Computer Vision and Machine Learning algorithms and news.

Subscribe Now (https://bigvisionllc.leadpages.net/leadbox/143948b73f72a2%3A173c9390c346dc/5649050225344512/)

Learn OpenCV

Keras Tutorial : Using pre-trained Imagenet models

DECEMBER 26, 2017 BY VIKAS GUPTA (HTTPS://WWW.LEARNOPENCV.COM/AUTHOR/VIKAS/)

(https://www.learnopencv.com/wp-content/uploads/2017/12/keras-classification-results-gif.gif)

This post is part of the series on Deep Learning for Beginners, which consists of the following tutorials :

1. Neural Networks : A 30,000 Feet View for Beginners (/neural-networks-a-30000-feet-view-for-beginners/)
2. Installation of Deep Learning frameworks (Tensorflow and Keras with CUDA support) (/installing-deep-learning-frameworks-on-ubuntu-with-cuda-support/)
3. Introduction to Keras (/deep-learning-using-keras-the-basics/)
4. Understanding Feedforward Neural Networks (/understanding-feedforward-neural-networks/)

5. Image Classification using Feedforward Neural Networks (/image-classification-using-feedforward-neural-network-in-keras/)
6. Image Recognition using Convolutional Neural Network (/image-classification-using-convolutional-

neural-networks-in-keras/)
7. Understanding Activation Functions (/understanding-activation-functions-in-deep-learning/)
8. Understanding AutoEncoders using Tensorflow (/understanding-autoencoders-using-tensorflow-python/)
9. Image Classification using pre-trained models in Keras
10. Transfer Learning using pre-trained models in Keras (/keras-tutorial-transfer-learning-using-pre-trained-models/)
11. Fine-tuning pre-trained models in Keras (/keras-tutorial-fine-tuning-using-pre-trained-models)
12. More to come . . .

In this post we will learn how to use pre-trained models trained on large datasets like ILSVRC, and also learn how to use them for a different task than it was trained on. We will be covering the following topics in the next three posts :

1. Image classification using different pre-trained models (this post)
2. Training a classifier for a different task, using the features extracted using the above-mentioned models – This is also referred to **Transfer Learning.**
3. Training a classifier for a different task, by modifying the weights of the above models – This is called **Fine-tuning**.

What is ImageNet

ImageNet (http://www.image-net.org/) is a project which aims to provide a large image database for research purposes. It contains more than 14 million images which belong to more than 20,000 classes (or synsets). They also provide bounding box annotations for around 1 million images, which can be used in Object Localization tasks. It should be noted that they only provide urls of images and you need to download those images.

What is ILSVRC

ImageNet Large Scale Visual Recognition Challenge (ILSVRC (http://image-net.org/challenges/LSVRC/2017/index)) is an annual competition organized by the ImageNet team since 2010, where research teams evaluate their computer vision algorithms various visual recognition tasks such as Object Classification and Object Localization. The training data is a subset of ImageNet with 1.2 million images belonging to 1000 classes. Deep Learning came to limelight in 2012 when Alex Krizhevsky and his team won the competition by a margin of a whooping 11%. ILSVRC and Imagenet are sometimes used interchangeably.

Why use pre-trained models?

Allow me a little digression.

Imagine two people, Mr. Couch Potato and Mr. Athlete. They sign up for soccer training at the same time. Neither of them has ever played soccer and the skills like dribbling, passing, kicking etc. are new to both of them.

Mr. Couch Potato does not move much, and Mr. Athlete does. That is the core difference between the two even before the training has even started. As you can imagine, the skills Mr. Athlete has developed as an athlete (e.g. stamina, speed and even sporting instincts) are going to be very useful for learning soccer even though Mr. Athlete has never trained for soccer.

Mr. Athlete benefits from his pre-training.

The same holds true for using pre-trained models in Neural Networks. A pre-trained model is trained on a different task than the task at hand but provides a very useful starting point because the features learned while training on the old task are useful for the new task.

We have seen earlier that we can create and train small convolutional networks (CNNs) to classify digits (using MNIST) or different objects (using CIFAR10). These small networks fall short when there are many classes and the objects vary in size / shape / appearance etc, as the model lacks the complexity which is required to model such large variations in data.

Even though it is possible to model any function using just a single hidden layer theoretically, but the number of neurons required to do so would be very large, making the network difficult to train. Thus, we use deep networks with many hidden layers which try to learn different features at different layers as we saw in the previous post on CNNs.

Deep networks have a large number of unknown parameters (in millions). The task of training a network is to find the optimum parameters using the training data. From linear algebra, we know that in order to solve an equation with three unknown parameters, we need three equations (data). And, if we know only two equations, we can get exact values of maximum 2 parameters and only an approximate value for the 3rd unknown parameter.

Similarly, for finding all the unknown parameters accurately, we would need a lot of data (in millions). If we have very few data, we will get only approximate values for most of the parameters, which we don’t want. Moral of the story is

> *For Deep Networks – More data -> Better learning.*

The problem is that it is difficult to get such huge labeled datasets for training the network.

Another problem, related to deep networks is that even if you get the data, it takes a large amount of time to train the network (hundreds of hours). Thus, it takes a lot of time, money and effort to train a deep network successfully.

Fortunately, we can leverage the models already trained on very large amounts of data for difficult tasks with thousands of classes. Many Research groups share the models they have trained for competitions like ILSVRC. The models have been trained on millions of images and for hundreds of hours on powerful GPUs. Most often we use these models as a starting point for our training process, instead of training our own model from scratch.

Enough of background, let's see how to use pre-trained models for image classification in Keras.

Pre-trained models present in Keras

The winners of ILSVRC have been very generous in releasing their models to the open-source community. There are many models such as AlexNet, VGGNet, Inception, ResNet, Xception and many more which we can choose from, for our own task. Apart from the ILSVRC winners, many research groups also share their models which they have trained for similar tasks, e.g, MobileNet, SqueezeNet etc.

These networks are trained for classifying images into one of 1000 categories or classes.

Keras comes bundled with many models. A trained model has two parts – Model Architecture and Model Weights. The weights are large files and thus they are not bundled with Keras. However, the weights file is automatically downloaded (one-time) if you specify that you want to load the weights trained on ImageNet data. It has the following models (as of Keras version 2.1.2):

- VGG16,
- InceptionV3,
- ResNet,
- MobileNet,

- Xception,
- InceptionResNetV2

Loading a Model in Keras

We can load the models in Keras using the following code

```
import keras
import numpy as np
from keras.applications import vgg16, inception_v3, resnet50, mobilenet

#Load the VGG model
vgg_model = vgg16.VGG16(weights='imagenet')

#Load the Inception_V3 model
inception_model = inception_v3.InceptionV3(weights='imagenet')

#Load the ResNet50 model
resnet_model = resnet50.ResNet50(weights='imagenet')

#Load the MobileNet model
mobilenet_model = mobilenet.MobileNet(weights='imagenet')
```

In the above code, we first import the python module containing the respective models. Then we load the model architecture and the imagenet weights for the networks. If you don't want to initialize the network with imagenet weights, replace 'imagenet' with None.

Loading and pre-processing an image

We can load the image using any library such as OpenCV, PIL, skimage etc. Keras also provides an image module which provides functions to import images and perform some basic pre-processing required before feeding it to the network for prediction. We will use the keras functions for loading and pre-processing the image. Specificallly, we perform the following steps on an input image:

1. Load the image. This is done using the load_img() function. Keras uses the PIL format for loading images. Thus, the image is in width x height x channels format.
2. Convert the image from PIL format to Numpy format (height x width x channels) using image_to_array() function.
3. The networks accept a 4-dimensional Tensor as an input of the form (batchsize, height, width, channels). This is done using the expand_dims() function in Numpy.

```
from keras.preprocessing.image import load_img
from keras.preprocessing.image import img_to_array
from keras.applications.imagenet_utils import decode_predictions
import matplotlib.pyplot as plt
%matplotlib inline

filename = 'images/cat.jpg'
```

```
filename = 'images/cat.jpg'
# load an image in PIL format
original = load_img(filename, target_size=(224, 224))
print('PIL image size',original.size)
plt.imshow(original)
plt.show()

# convert the PIL image to a numpy array
# IN PIL - image is in (width, height, channel)
# In Numpy - image is in (height, width, channel)
numpy_image = img_to_array(original)
plt.imshow(np.uint8(numpy_image))
plt.show()
print('numpy array size',numpy_image.shape)

# Convert the image / images into batch format
# expand_dims will add an extra dimension to the data at a particular axis
# We want the input matrix to the network to be of the form (batchsize, height, width, channels)
# Thus we add the extra dimension to the axis 0.
image_batch = np.expand_dims(numpy_image, axis=0)
print('image batch size', image_batch.shape)
plt.imshow(np.uint8(image_batch[0]))
```

Output

('PIL image size', (224, 224))
('numpy array size', (224, 224, 3))
('image batch size', (1, 224, 224, 3))

(https://www.learnopencv.com/wp-content/uploads/2017/12/cat.jpg)

Predicting the Object Class

Once we have the image in the right format, we can feed it to the network and get the predictions. The image we got in the previous step should be normalized by subtracting the mean of the ImageNet data. This is because the network was trained on the images after this pre-processing. We follow the following steps to

get the classification results.

1. Preprocess the input by subtracting the mean value from each channel of the images in the batch. Mean is an array of three elements obtained by the average of R, G, B pixels of all images obtained from ImageNet. The values for Imagenet are : [103.939, 116.779, 123.68]. This is done using the preprocess_input() function.
2. Get the classification result, which is a Tensor of dimension (batchsize x 1000). This is done by model.predict() function.
3. Convert the result to human-readable labels – The vector obtained above has too many values to make any sense. Keras provides a function decode_predictions() which takes the classification results, sorts it according to the confidence of prediction and gets the class name (instead of a class-number). We can also specify how many results we want, using the top argument in the function. The output shows the *class ID, class name and the confidence of prediction*.

```
# prepare the image for the VGG model
processed_image = vgg16.preprocess_input(image_batch.copy())

# get the predicted probabilities for each class
predictions = vgg_model.predict(processed_image)
# print predictions

# convert the probabilities to class labels
# We will get top 5 predictions which is the default
label = decode_predictions(predictions)
print label
```

Output

[[(u'n02123597', u'Siamese_cat', 0.30934173),
(u'n01877812', u'wallaby', 0.080341272),
(u'n02326432', u'hare', 0.075098492),
(u'n02325366', u'wood_rabbit', 0.050530687),
(u'n03223299', u'doormat', 0.048173629)]]

Comparison of Results from various Models

Let us see what the different models say for a few images.

Giving a cat image as input, and running it on the 4 models, we get the following output.

(https://www.learnopencv.com/wp-content/uploads/2017/12/cat_output.jpg)

Giving an Dog as an input, this is the output.


ResNet50: Siamese_cat, 0.14


VGG16: Rhodesian_ridgeback, 0.43
MobileNet: American_Staffordshire_terrier, 0.74
Inception: Staffordshire_bullterrier, 0.69

(https://www.learnopencv.com/wp-content/uploads/2017/12/dog_output.jpg)

With an orange,

(https://www.learnopencv.com/wp-content/uploads/2017/12/orange_output.jpg)

For a tomato, we get

(https://www.learnopencv.com/wp-content/uploads/2017/12/tomato_output.jpg)

For a watermelon we get

(https://www.learnopencv.com/wp-content/uploads/2017/12/watermelon_output.jpg)

Well, it looks like the ILSVRC does not recognize tomatoes and watermelons. We will see how to train a classifier using these same models with our own data to recognize any other set of objects which are not present in the ILSVRC dataset. This would be the topic of our next two posts. Stay tuned!

Subscribe & Download Code

If you liked this article and would like to download code and example images used in this post, please subscribe (https://bigvisionllc.leadpages.net/leadbox/143948b73f72a2%3A173c9390c346dc/5649050225344512/) to our newsletter. You will also receive a free Computer Vision Resource (https://bigvisionllc.leadpages.net/leadbox/143948b73f72a2%3A173c9390c346dc/5649050225344512/) Guide. In our newsletter, we share OpenCV tutorials and examples written in C++/Python, and Computer Vision and Machine Learning algorithms and news.

Subscribe Now
(https://bigvisionllc.leadpages.net/leadbox/143948b73f72a2%3A173c9390c346dc/5649050225344512/)

Learn OpenCV

Keras Tutorial : Transfer Learning using pre-trained models

JANUARY 3, 2018 BY VIKAS GUPTA (HTTPS://WWW.LEARNOPENCV.COM/AUTHOR/VIKAS/)

This post is part of the series on Deep Learning for Beginners, which consists of the following tutorials :

1. Neural Networks : A 30,000 Feet View for Beginners (/neural-networks-a-30000-feet-view-for-beginners/)
2. Installation of Deep Learning frameworks (Tensorflow and Keras with CUDA support) (/installing-deep-learning-frameworks-on-ubuntu-with-cuda-support/)
3. Introduction to Keras (/deep-learning-using-keras-the-basics/)
4. Understanding Feedforward Neural Networks (/understanding-feedforward-neural-networks/)
5. Image Classification using Feedforward Neural Networks (/image-classification-using-feedforward-neural-network-in-keras/)
6. Image Recognition using Convolutional Neural Network (/image-classification-using-convolutional-neural-networks-in-keras/)
7. Understanding Activation Functions (/understanding-activation-functions-in-deep-learning/)
8. Understanding AutoEncoders using Tensorflow (/understanding-autoencoders-using-tensorflow-python/)
9. Image Classification using pre-trained models in Keras (/keras-tutorial-using-pre-trained-imagenet-models/)
10. Transfer Learning using pre-trained models in Keras
11. Fine-tuning pre-trained models in Keras (/keras-tutorial-fine-tuning-using-pre-trained-models)
12. More to come . . .

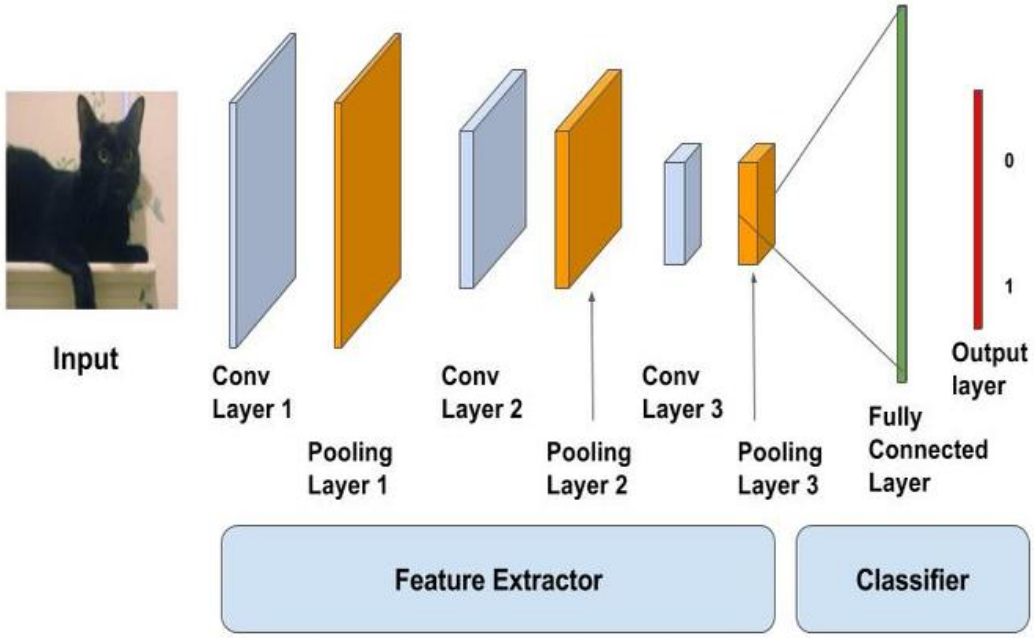
In our previous tutorial (/keras-tutorial-using-pre-trained-imagenet-models/), we learned how to use models which were trained for Image Classification on the ILSVRC data. In this tutorial, we will discuss how to use those models as a Feature Extractor and train a new model for a different classification task.

Suppose you want to make a household robot which can cook food. The first step would be to identify different vegetables. We will try to build a model which identifies Tomato, Watermelon, and Pumpkin for this tutorial. In the previous tutorial, we saw the pre-trained models were not able to identify them because these categories were not learned by the models.

Transfer Learning vs Fine-tuning

The pre-trained models are trained on very large scale image classification problems. The convolutional

layers act as feature extractor and the fully connected layers act as Classifiers.

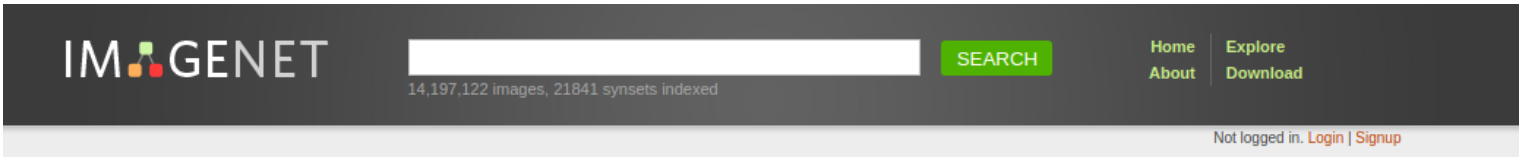


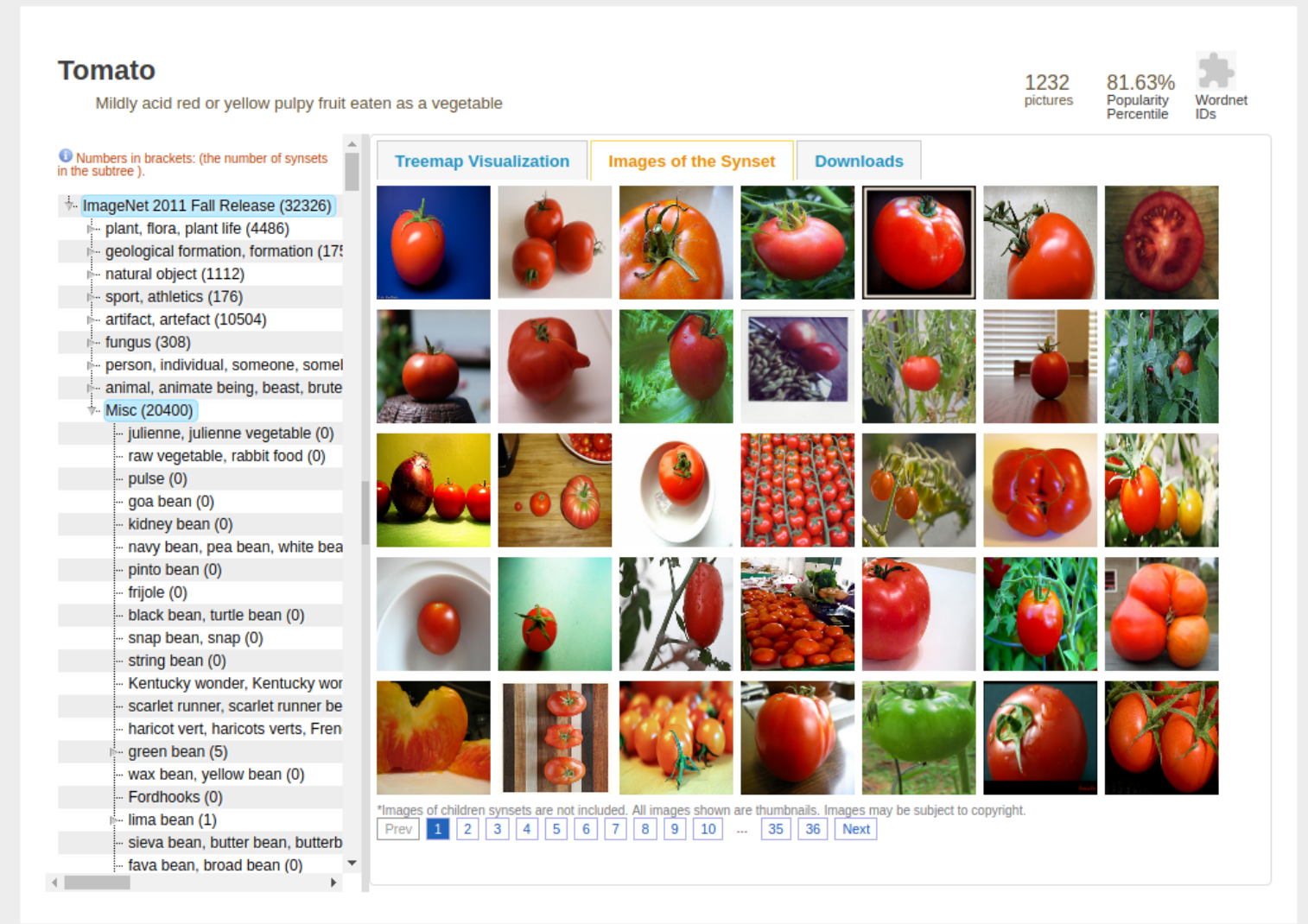
(/wp-content/uploads/2017/11/cnn-schema1.jpg)

Since these models are very large and have seen a huge number of images, they tend to learn very good, discriminative features. We can either use the convolutional layers merely as a feature extractor or we can tweak the already trained convolutional layers to suit our problem at hand. The former approach is known as **Transfer Learning** and the latter as **Fine-tuning**.

As a rule of thumb, when we have a small training set and our problem is similar to the task for which the pre-trained models were trained, we can use transfer learning. If we have enough data, we can try and tweak the convolutional layers so that they learn more robust features relevant to our problem. You can get a detailed overview of Fine-tuning and transfer learning **here** (http://cs231n.github.io/transfer-learning/). We will discuss Transfer Learning in Keras in this post.

ImageNet Jargon





(/wp-content/uploads/2018/01/imagenet-tomato.png)

ImageNet is based upon WordNet which groups words into sets of synonyms (synsets). Each synset is assigned a “wnid” (Wordnet ID). Note that in a general category, there can be many subcategories and each of them will belong to a different synset. For example Working Dog (sysnet = n02103406), Guide Dog (sysnet = n02109150), and Police Dog (synset = n02106854) are three different synsets.

The wnid’s of the 3 object classes we are considering are given below

- n07734017 -> Tomato
- n07735510 -> Pumpkin
- n07756951 -> WaterMelon

Download and prepare Data

For downloading Imagenet images by wnid, there is a nice code repository written by Tzuta Lin which is

available on **Github** (https://github.com/tzutalin/ImageNet_Utils). You can use this to download images of a specific “wnid”. You can visit the github page and follow the instructions to download the images for any of the wnid’s.

However, If you are just starting out and do not want to download full size images, you can use another python library available through pip – imagenetscraper (https://pypi.python.org/pypi/imagenetscraper). It is easy to use and also provides resizing options. Installation and usage instructions are provided below. Note that it works with python3 only.


```
# Install imagenetscraper
pip3 install imagenetscraper

# Download the images for the three wnids and keep them in separate folders.
imagenetscraper n07756951 watermelon
imagenetscraper n07734017 tomato
imagenetscraper n07735510 pumpkin
```

I found that the data is very noisy, i.e. there is a lot of clutter, the objects are occluded etc. So, I shortlisted around 250 images for each class. We need to create two directories namely “train” and “validation” so that we can use the Keras functions for loading images in batches.

Load the pre-trained model

```
from keras.applications import VGG16

vgg_conv = VGG16(weights='imagenet',
                 include_top=False,
                 input_shape=(224, 224, 3))
```

In the above code, we load the VGG Model along with the ImageNet weights similar to our previous tutorial. There is, however, one change – include_top=False. We have not loaded the last two fully connected layers which act as the classifier. We are just loading the convolutional layers. It should be noted that the last layer has a shape of 7 x 7 x 512.

Extract Features

The data is divided into 80:20 ratio and kept in separate train and validation folders. Each folder should contain 3 folders belonging to he respective classes. You can change the directory according to your system.

```
train_dir = '/clean_dataset/train'
```

```
train_dir = './clean-dataset/train'
validation_dir = './clean-dataset/validation'

nTrain = 600
nVal = 150
```

We will use the ImageDataGenerator class to load the images and flow_from_directory function to generate batches of images and labels.

```
datagen = ImageDataGenerator(rescale=1./255)
batch_size = 20

train_features = np.zeros(shape=(nTrain, 7, 7, 512))
train_labels = np.zeros(shape=(nTrain,3))

train_generator = datagen.flow_from_directory(
    train_dir,
    target_size=(224, 224),
    batch_size=batch_size,
    class_mode='categorical',
    shuffle=shuffle)
```

Then we use model.predict() function to pass the image through the network which gives us a 7 x 7 x 512 dimensional Tensor. We reshape the Tensor into a vector. Similarly, we find the validation_features.

```
i = 0
for inputs_batch, labels_batch in train_generator:
    features_batch = vgg_conv.predict(inputs_batch)
    train_features[i * batch_size : (i + 1) * batch_size] = features_batch
    train_labels[i * batch_size : (i + 1) * batch_size] = labels_batch
    i += 1
    if i * batch_size >= nImages:
        break

train_features = np.reshape(train_features, (nTrain, 7 * 7 * 512))
```

Create your own model

We will create a simple feedforward network with a softmax output layer having 3 classes.

```
from keras import models
from keras import layers
from keras import optimizers

model = models.Sequential()
model.add(layers.Dense(256, activation='relu', input_dim=7 * 7 * 512))
model.add(layers.Dropout(0.5))
model.add(layers.Dense(3, activation='softmax'))
```

Train the model

Training a network in Keras is as simple as calling model.fit() function as we have seen in our earlier tutorials.

```
model.compile(optimizer=optimizers.RMSprop(lr=2e-4),
              loss='categorical_crossentropy',
              metrics=['acc'])

history = model.fit(train_features,
                    train_labels,
                    epochs=20,
```

```
                    epochs=20,
                    batch_size=batch_size,
                    validation_data=(validation_features,validation_labels))
```

Check Performance

We would like to visualize which images were wrongly classified.

```
fnames = validation_generator.filenames

ground_truth = validation_generator.classes

label2index = validation_generator.class_indices

# Getting the mapping from class index to class label
idx2label = dict((v,k) for k,v in label2index.iteritems())

predictions = model.predict_classes(validation_features)
prob = model.predict(validation_features)

errors = np.where(predictions != ground_truth)[0]
print("No of errors = {}/{}".format(len(errors),nVal))
```

Let us see which images were predicted wrongly

```
for i in range(len(errors)):
    pred_class = np.argmax(prob[errors[i]])
    pred_label = idx2label[pred_class]

    print('Original label:{}, Prediction :{}, confidence : {:.3f}'.format(
        fnames[errors[i]].split('/')[0],
        pred_label,
        prob[errors[i]][pred_class]))

    original = load_img('{}/{}'.format(validation_dir,fnames[errors[i]]))
    plt.imshow(original)
    plt.show()
```

(/wp-content/uploads/2018/01/result-transfer-learning-image.jpg)

We will try to improve on the limitations of transfer learning by using another approach called Fine-tuning in our next post.

References

https://blog.keras.io/building-powerful-image-classification-models-using-very-little-data.html
Deep Learning with Python Github Repository (https://github.com/fchollet/deep-learning-with-python-notebooks)

Subscribe & Download Code

If you liked this article and would like to download code and example images used in this post, please subscribe (https://bigvisionllc.leadpages.net/leadbox/143948b73f72a2%3A173c9390c346dc/5649050225344512/) to our newsletter. You will also receive a free Computer Vision Resource (https://bigvisionllc.leadpages.net/leadbox/143948b73f72a2%3A173c9390c346dc/5649050225344512/) Guide. In our newsletter, we share OpenCV tutorials and examples written in C++/Python, and Computer Vision and Machine Learning algorithms and news.

Subscribe Now
(https://bigvisionllc.leadpages.net/leadbox/143948b73f72a2%3A173c9390c346dc/5649050225344512/)

Learn OpenCV

Keras Tutorial : Fine-tuning using pre-trained models

FEBRUARY 6, 2018 BY VIKAS GUPTA (HTTPS://WWW.LEARNOPENCV.COM/AUTHOR/VIKAS/)

(/wp-content/uploads/2018/01/keras-ft-result.jpg)

This post is part of the series on Deep Learning for Beginners, which consists of the following tutorials :

1. Neural Networks : A 30,000 Feet View for Beginners (/neural-networks-a-30000-feet-view-for-beginners/)
2. Installation of Deep Learning frameworks (Tensorflow and Keras with CUDA support) (/installing-deep-learning-frameworks-on-ubuntu-with-cuda-support/)
3. Introduction to Keras (/deep-learning-using-keras-the-basics/)
4. Understanding Feedforward Neural Networks (/understanding-feedforward-neural-networks/)
5. Image Classification using Feedforward Neural Networks (/image-classification-using-feedforward-neural-network-in-keras/)
6. Image Recognition using Convolutional Neural Network (/image-classification-using-convolutional-neural-networks-in-keras/)
7. Understanding Activation Functions (/understanding-activation-functions-in-deep-learning/)
8. Understanding AutoEncoders using Tensorflow (/understanding-autoencoders-using-tensorflow-python/)
9. Image Classification using pre-trained models in Keras (/keras-tutorial-using-pre-trained-imagenet-models/)
10. Transfer Learning using pre-trained models in Keras (/keras-tutorial-transfer-learning-using-pre-trained-models/)
11. Fine-tuning pre-trained models in Keras
12. More to come . . .

In the previous two posts, we learned how to use pre-trained models and how to extract features from them for training a model for a different task. In this tutorial, we will learn how to fine-tune a pre-trained model for a different task than it was originally trained for.

We will try to improve on the problem of classifying pumpkin, watermelon, and tomato discussed in the previous post (/keras-tutorial-transfer-learning-using-pre-trained-models/). We will be using the same data for this tutorial.

What is Fine-tuning of a network

We have already explained (/keras-tutorial-using-pre-trained-imagenet-models/#why-pretrained-models) the importance of using pre-trained networks in our previous article. Just to recap, when we train a network from scratch, we encounter the following two limitations :

- Huge data required – Since the network has millions of parameters, to get an optimal set of parameters, we need to have a lot of data.
- Huge computing power required – Even if we have a lot of data, training generally requires multiple iterations and it takes a toll on the computing resources.

The task of fine-tuning a network is to tweak the parameters of an already trained network so that it adapts to the new task at hand. As explained here (/image-classification-using-convolutional-neural-networks-in-keras/#cnn-hierachical), the initial layers learn very general features and as we go higher up the network, the layers tend to learn patterns more specific to the task it is being trained on. Thus, for fine-tuning, we want to keep the initial layers intact (or freeze them) and retrain the later layers for our task.

Thus, fine-tuning avoids both the limitations discussed above.

- The amount of data required for training is not much because of two reasons. First, we are not training the entire network. Second, the part that is being trained is not trained from scratch.
- Since the parameters that need to be updated is less, the amount of time needed will also be less.

Fine-tuning in Keras

Let us directly dive into the code without much ado. We will be using the same data which we used in the previous post. You can choose to use a larger dataset if you have a GPU as the training will take much longer if you do it on a CPU for a large dataset. We will use the VGG model for fine-tuning.

Download Code

To easily follow along this tutorial, please download code by clicking on the button below. It's FREE!

DOWNLOAD CODE
(HTTPS://BIGVISIONLLC.LEADPAGES.NET/LEADBOX/143948B73F72A2%3A173C9390C346DC/5649050225344512/)

Load the pre-trained model

First, we will load a VGG model without the top layer (which consists of fully connected layers).

```
from keras.applications import VGG16
#Load the VGG model
vgg_conv = VGG16(weights='imagenet', include_top=False, input_shape=(image_size, image_size, 3))
```

Freeze the required layers

In Keras, each layer has a parameter called “trainable”. For freezing the weights of a particular layer, we should set this parameter to False, indicating that this layer should not be trained. That’s it! We go over each layer and select which layers we want to train.

```
# Freeze the layers except the last 4 layers
for layer in vgg_conv.layers[:-4]:
    layer.trainable = False

# Check the trainable status of the individual layers
for layer in vgg_conv.layers:
    print(layer, layer.trainable)
```

```
<keras.engine.topology.InputLayer object at 0x7fd4586361d0> False
<keras.layers.convolutional.Conv2D object at 0x7fd458636c88> False
<keras.layers.convolutional.Conv2D object at 0x7fd458636048> False
<keras.layers.pooling.MaxPooling2D object at 0x7fd4583f57b8> False
<keras.layers.convolutional.Conv2D object at 0x7fd4573ebf98> False
<keras.layers.convolutional.Conv2D object at 0x7fd456ff9be0> False
<keras.layers.pooling.MaxPooling2D object at 0x7fd45717ca90> False
<keras.layers.convolutional.Conv2D object at 0x7fd4571b0080> False
<keras.layers.convolutional.Conv2D object at 0x7fd4571b0860> False
<keras.layers.convolutional.Conv2D object at 0x7fd4571b46a0> False
<keras.layers.pooling.MaxPooling2D object at 0x7fd45735c978> False
<keras.layers.convolutional.Conv2D object at 0x7fd45734d5f8> False
<keras.layers.convolutional.Conv2D object at 0x7fd45734d470> False
<keras.layers.convolutional.Conv2D object at 0x7fd457332e80> False
<keras.layers.pooling.MaxPooling2D object at 0x7fd4571d1cf8> False
<keras.layers.convolutional.Conv2D object at 0x7fd4571fc1d0> True
<keras.layers.convolutional.Conv2D object at 0x7fd4571fcf98> True
<keras.layers.convolutional.Conv2D object at 0x7fd4571fa7f0> True
<keras.layers.pooling.MaxPooling2D object at 0x7fd457276048> True
```

(/wp-content/uploads/2018/01/keras-ft-trainable-layers.png)

Create a new model

Now that we have set the trainable parameters of our base network, we would like to add a classifier on top of the convolutional base. We will simply add a fully connected layer followed by a softmax layer with 3 outputs. This is done as given below.

```
from keras import models
from keras import layers
from keras import optimizers

# Create the model
model = models.Sequential()

# Add the vgg convolutional base model
model.add(vgg_conv)

# Add new layers
model.add(layers.Flatten())
model.add(layers.Dense(1024, activation='relu'))
model.add(layers.Dropout(0.5))
model.add(layers.Dense(3, activation='softmax'))

# Show a summary of the model. Check the number of trainable parameters
model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
vgg16 (Model)                (None, 7, 7, 512)         14714688
_________________________________________________________________
flatten_2 (Flatten)          (None, 25088)             0
_________________________________________________________________
dense_3 (Dense)              (None, 1024)              25691136
_________________________________________________________________
dropout_2 (Dropout)          (None, 1024)              0
_________________________________________________________________
dense_4 (Dense)              (None, 3)                 3075
=================================================================
Total params: 40,408,899
Trainable params: 32,773,635
Non-trainable params: 7,635,264
_________________________________________________________________
```

(https://www.learnopencv.com/wp-content/uploads/2018/01/keras-ft-model-summary.png)

Setup the data generators

We have already separated the data into train and validation and kept it in the “train” and “validation” folders. We can use ImageDataGenerator available in Keras to read images in batches directly from these folders and optionally perform data augmentation. We will use two different data generators for train and validation folders.

```
train_datagen = ImageDataGenerator(
      rescale=1./255,
      rotation_range=20,
      width_shift_range=0.2,
      height_shift_range=0.2,
      horizontal_flip=True,
      fill_mode='nearest')
```

```
        fill_mode= 'nearest' )

validation_datagen = ImageDataGenerator(rescale=1./255)

# Change the batchsize according to your system RAM
train_batchsize = 100
val_batchsize = 10

train_generator = train_datagen.flow_from_directory(
        train_dir,
        target_size=(image_size, image_size),
        batch_size=train_batchsize,
        class_mode='categorical')

validation_generator = validation_datagen.flow_from_directory(
        validation_dir,
        target_size=(image_size, image_size),
        batch_size=val_batchsize,
        class_mode='categorical',
        shuffle=False)
```

Train the model

Till now, we have created the model and set up the data for training. So, we should proceed with the training and check out the performance. We will have to specify the optimizer and the learning rate and start training using the model.fit() function. After the training is over, we will save the model.

```
# Compile the model
model.compile(loss='categorical_crossentropy',
              optimizer=optimizers.RMSprop(lr=1e-4),
              metrics=['acc'])
# Train the model
history = model.fit_generator(
      train_generator,
      steps_per_epoch=train_generator.samples/train_generator.batch_size ,
      epochs=30,
      validation_data=validation_generator,
      validation_steps=validation_generator.samples/validation_generator.batch_size,
      verbose=1)

# Save the model
model.save('small_last4.h5')
```

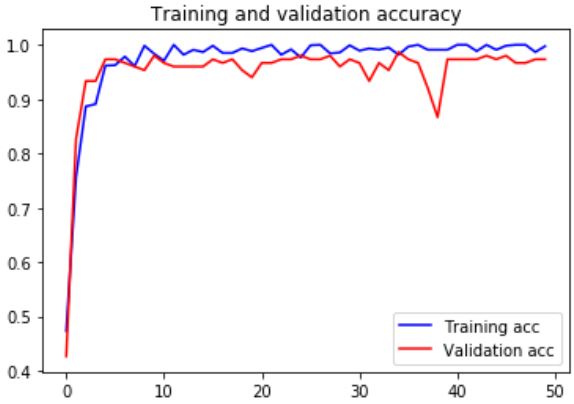
Check Performance

We obtained an accuracy of 90% with the transfer learning approach discussed in our previous blog. Here we are getting a much better accuracy of 98%.

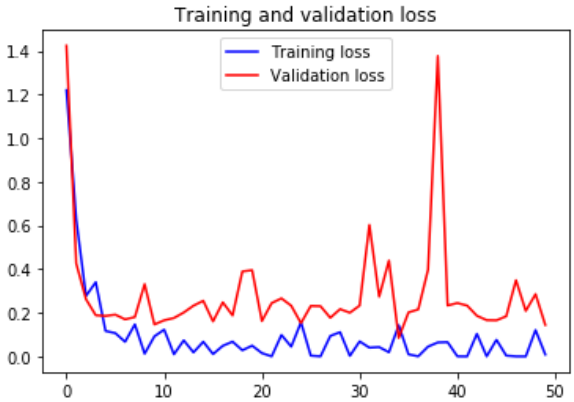
Let us see the loss and accuracy curves.

```
acc = history.history['acc']
val_acc = history.history['val_acc']
loss = history.history['loss']
val_loss = history.history['val_loss']

epochs = range(len(acc))

```

```
plt.plot(epochs, acc, 'b', label='Training acc')
plt.plot(epochs, val_acc, 'r', label='Validation acc')
plt.title('Training and validation accuracy')
plt.legend()

plt.figure()

plt.plot(epochs, loss, 'b', label='Training loss')
plt.plot(epochs, val_loss, 'r', label='Validation loss')
plt.title('Training and validation loss')
plt.legend()

plt.show()
```



(/wp-content/uploads/2018/01/keras-ft-accuracy-curve.png)



(/wp-content/uploads/2018/01/keras-ft-loss-curve.png)

Also, let us visually see the errors that we got.

```
# Create a generator for prediction
validation_generator = validation_datagen.flow_from_directory(
        validation_dir,
        target_size=(image_size, image_size),
        batch_size=val_batchsize,
        class_mode='categorical',
        shuffle=False)
```

```
        shuffle=False)

# Get the filenames from the generator
fnames = validation_generator.filenames

# Get the ground truth from generator
ground_truth = validation_generator.classes

# Get the label to class mapping from the generator
label2index = validation_generator.class_indices

# Getting the mapping from class index to class label
idx2label = dict((v,k) for k,v in label2index.items())

# Get the predictions from the model using the generator
predictions = model.predict_generator(validation_generator, steps=validation_generator.samples/validation_generator.bat
predicted_classes = np.argmax(predictions,axis=1)

errors = np.where(predicted_classes != ground_truth)[0]
print("No of errors = {}/{}".format(len(errors),validation_generator.samples))

# Show the errors
for i in range(len(errors)):
    pred_class = np.argmax(predictions[errors[i]])
    pred_label = idx2label[pred_class]

    title = 'Original label:{}, Prediction :{}, confidence : {:.3f}'.format(
        fnames[errors[i]].split('/')[0],
        pred_label,
        predictions[errors[i]][pred_class])

    original = load_img('{}/{}'.format(validation_dir,fnames[errors[i]]))
    plt.figure(figsize=[7,7])
    plt.axis('off')
    plt.title(title)
    plt.imshow(original)
    plt.show()
```

(/wp-content/uploads/2018/01/keras-ft-error1.png)

(/wp-content/uploads/2018/01/keras-ft-error2.png)

(/wp-content/uploads/2018/01/keras-ft-error3.png)

Experiments

We have done 3 experiments to see the effect of fine-tuning and data augmentation. We kept the validation set same as the previous post i.e. 50 images per class.

1. Freezing all layers and learning a classifier on top of it – similar to transfer learning. The number of errors was 15 out of 150 images which is similar to what we got in the previous post.
2. Training the last 3 convolutional layers – We got 9 errors out of 150.

3. Training the last 3 convolutional layers with data augmentation – The number of errors reduced to 3 out of 150.

I hope you find this useful. Try doing your own experiments and post your findings in the comments section.

References

Keras Blog (https://blog.keras.io/building-powerful-image-classification-models-using-very-little-data.html)
Deep Learning with Python Github Repository (https://github.com/fchollet/deep-learning-with-python-notebooks)

Subscribe & Download Code

If you liked this article and would like to download code and example images used in this post, please subscribe (https://bigvisionllc.leadpages.net/leadbox/143948b73f72a2%3A173c9390c346dc/5649050225344512/) to our newsletter. You will also receive a free Computer Vision Resource (https://bigvisionllc.leadpages.net/leadbox/143948b73f72a2%3A173c9390c346dc/5649050225344512/) Guide. In our newsletter, we share OpenCV tutorials and examples written in C++/Python, and Computer Vision and Machine Learning algorithms and news.

Subscribe Now
(https://bigvisionllc.leadpages.net/leadbox/143948b73f72a2%3A173c9390c346dc/5649050225344512/)