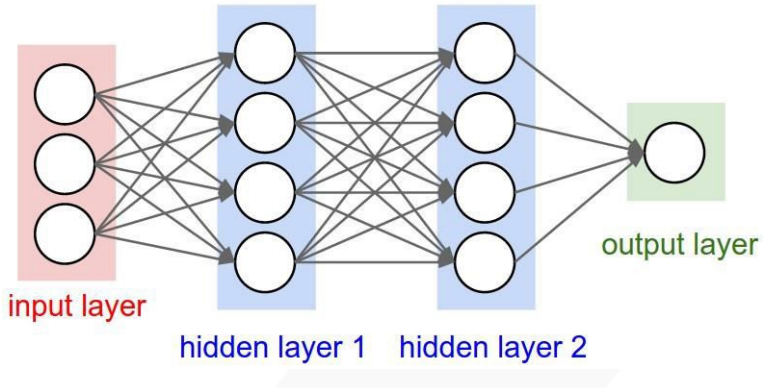
ANALYTIXLABS

Introduction to Keras

- ANN – Deep Learning – Keras
- ANN Sample Architectures
- Keras Overview
- Keras Steps
- Keras in depth
  - Data Representation
  - Keras Models and Layers
  - Activations, Losses and Optimizers
  - Learning rate scheduler
  - Metrics and Performance evaluation strategies
  - Regularizers
  - Saving and loading
  - Model visualization
  - Callbacks
- Keras Cheat Sheet, Examples and Models
- Keras and Visualization

ANN – Deep Learning - Keras

- Generally speaking, **neural networks** are nonlinear machine learning models.
  - They can be used for **supervised** or **unsupervised** learning.
- **Deep learning** refers to training neural nets with multiple layers.
  - They are more powerful but only if you have lots of data to train them on.
- **Keras** is used to create neural network models



Keras Overview

Keras:

- Neural Network library written in python
- Design to be simple and straightforward
- Built on top of different deep learning libraries such as Tensorflow, Theano and CNTK
- Keras has builtin GPU support with CUDA – CUDA is framework for using the GPU on Nvidia video cards for mathematical (tensor) operations

Why Keras?

- Simple
- Highly modular
- Deep enough to build models

Keras: Backend

- TensorFlow, Theano, CNTK, (Microsoft)
- Can deploy in production via TensorFlow Serving

When to use Keras?

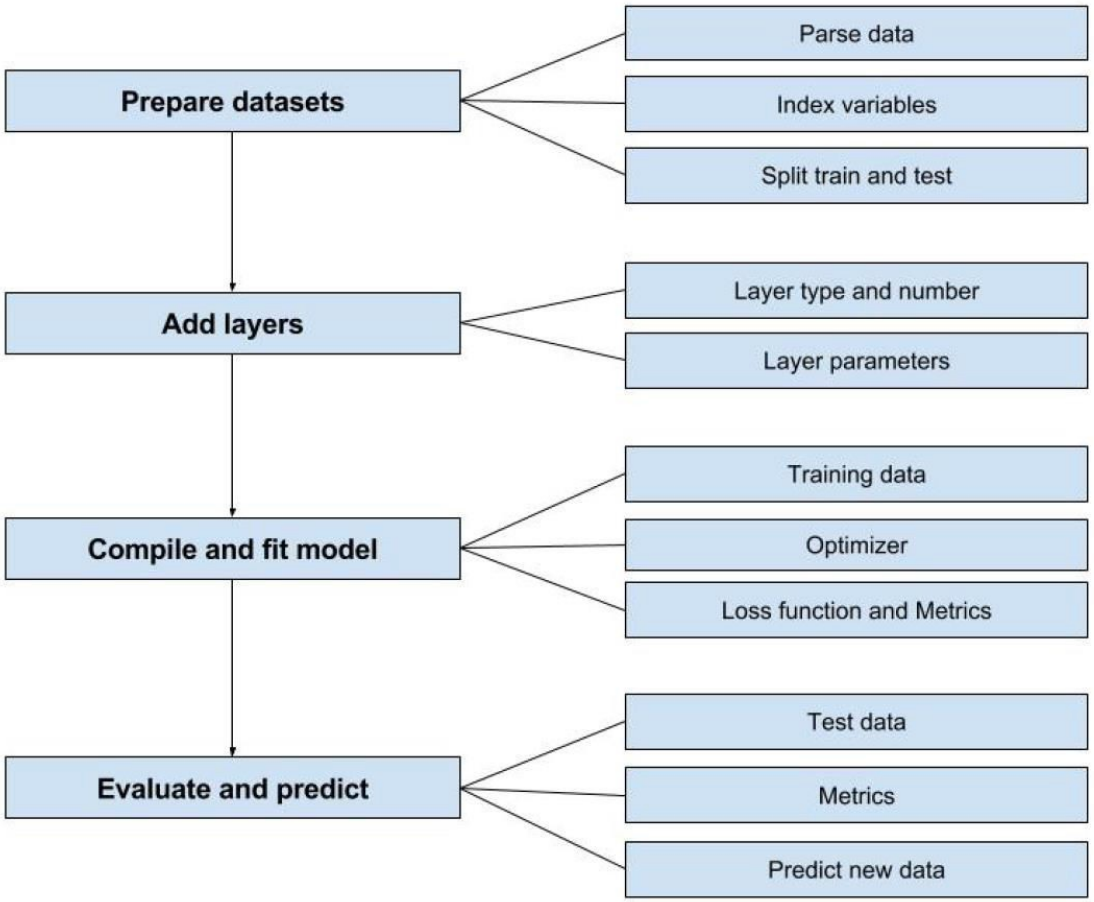
If you're a beginner and interested in quickly implementing your ideas

- Python + **Keras**: Super fast implementation, good extensibility

If you want to do fundamental research in Deep Learning

- Python + **Tensorflow** or PyTorch: Excellent extensibility

Keras Steps


Prepare datasets
Parse data
Index variables
Split train and test
Add layers
Layer type and number
Layer parameters
Compile and fit model
Training data
Optimizer
Loss function and Metrics
Evaluate and predict
Test data
Metrics
Predict new data

Keras steps - Basics

1. Sequential model (a linear stack of layers)

```
from keras.models import Sequential
model = Sequential()
```

```
from keras.layers import Dense, Activation

model.add(Dense(units=64, input_dim=100))
model.add(Activation('relu'))
model.add(Dense(units=10))
model.add(Activation('softmax'))
```

2. Compile Model

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

Also, you can further configure your optimizer

```
model.compile(loss=keras.losses.categorical_crossentropy,
optimizer=keras.optimizers.SGD(lr=0.01, momentum=0.9))
```

Keras steps - Basics

3. Training

```
model.fit(x_train, y_train, epochs=5, batch_size=32)
```

You can feed data batches manualy

```
model.train_on_batch(x_batch, y_batch)
```

4. Evaluation

```
loss_and_metrics = model.evaluate(x_test, y_test, batch_size=128)
```

5. Prediction

```
classes = model.predict(x_test, batch_size=128)
```

Keras in depth – Data Representation

- Discrete Variables
  - Numeric variables that have a countable number of values between any two values.
  - E.g., the number of children in a family - 3.
- Continuous Variables
  - Numeric variables that have an infinite number of values between any two values. E.g., temperature - 25.9
- Categorical variables
  - A finite number of categories or distinct groups. Categorical data might not have a logical order. E.g., gender - male/female

Keras in depth – Data Representation (Utile)

Although you can use any other method for feature preprocessing, keras has a couple of utilities to help, such as:

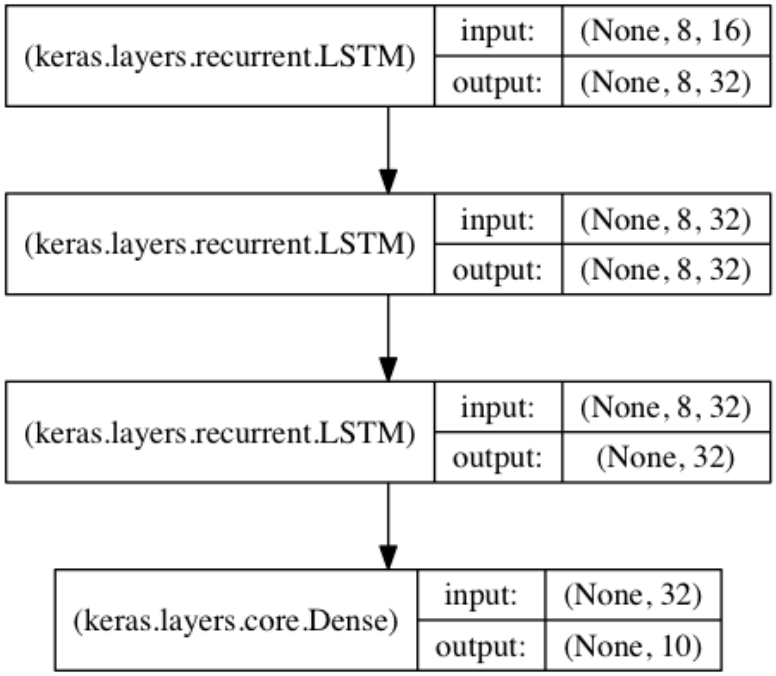
- To_categorical (to one-hot encode data)
- Text preprocessing utilities, such as tokenizing
- LabelEncoder " using Skikit Learn"

```
from sklearn.preprocessing import LabelEncoder
from keras.utils import np_utils

encoder = LabelEncoder()
encoder.fit(Y)
encoded_Y = encoder.transform(Y)
# convert integers to dummy variables (i.e. one hot encoded)
dummy_y = np_utils.to_categorical(encoded_Y)
```

Keras Models - Sequential Models

1. Sequential Models : stack of layers

```
data_dim = 16
timesteps = 8
model = Sequential()
model.add(LSTM(32, return_sequences=True,
input_shape=(timesteps, data_dim)))
model.add(LSTM(32, return_sequences=True))
model.add(LSTM(32))
model.add(Dense(10, activation='softmax'))
model.compile(loss='categorical_crossentropy
', optimizer='rmsprop',
metrics=['accuracy'])
```



Keras Models - Model Class API

- The most important part of Keras are models
- Model = layers, loss, and an optimizer
- These are the objects that you add Layers to, call compile() and fit() on
- Models can be saved and check pointed for later use

Keras Models - Model Class API

2. Model Class API

- Optimized over all outputs Graph model
- allows for two or more independent networks to diverge or merge
- Allows for multiple separate inputs or outputs
- Different merging layers (sum or concatenate)

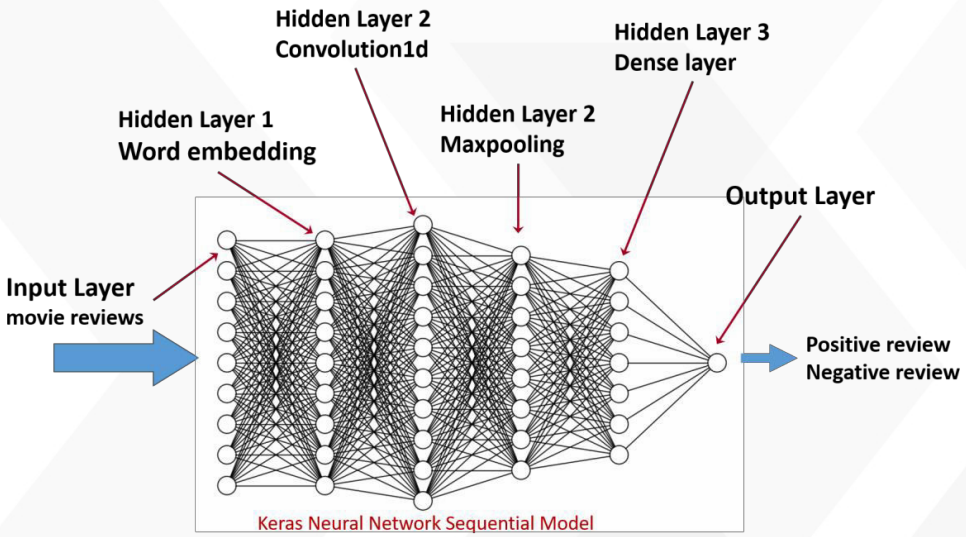
```
from keras.models import Model
from keras.layers import Input, Dense
a = Input(shape=(32,))
b = Dense(32)(a)
model = Model(inputs=a, outputs=b)
```

Keras Layers

Layers are used to define what your architecture looks like

Examples of layers are:

- Dense layers (this is the normal, fully-connected layer)
- Convolutional layers (applies convolution operations on the previous layer)
- Pooling layers (used after convolutional layers)
- Dropout layers (these are used for regularization, to avoid overfitting)



Keras Layers

Keras has a number of pre-built layers. Notable examples include:

Regular dense, MLP type

```
keras.layers.core.Dense(units, activation=None, use_bias=True, kernel_initializer='glorot_uniform',
bias_initializer='zeros', kernel_regularizer=None, bias_regularizer=None,
activity_regularizer=None, kernel_constraint=None, bias_constraint=None)
```

Recurrent layers, LSTM, GRU, etc

```
keras.layers.recurrent.LSTM(units, activation='tanh', recurrent_activation='hard_sigmoid',
use_bias=True, kernel_initializer='glorot_uniform', recurrent_initializer='orthogonal',
bias_initializer='zeros', unit_forget_bias=True, kernel_regularizer=None, recurrent_regularizer=None,
bias_regularizer=None, activity_regularizer=None, kernel_constraint=None, recurrent_constraint=None,
bias_constraint=None, dropout=0.0, recurrent_dropout=0.0)
```

Keras Layers

2D Convolutional layers

```
keras.layers.convolutional.Conv2D(filters, kernel_size, strides=(1, 1), padding='valid',
data_format=None, dilation_rate=(1, 1), activation=None, use_bias=True,
kernel_initializer='glorot_uniform', bias_initializer='zeros', kernel_regularizer=None,
bias_regularizer=None, activity_regularizer=None, kernel_constraint=None,
bias_constraint=None)
```

Autoencoders can be built with any other type of layer

```
from keras.layers import Dense, Activation
model.add(Dense(units=32, input_dim=512))
model.add(Activation('relu'))
model.add(Dense(units=512))
model.add(Activation('sigmoid'))
```

Keras Layers

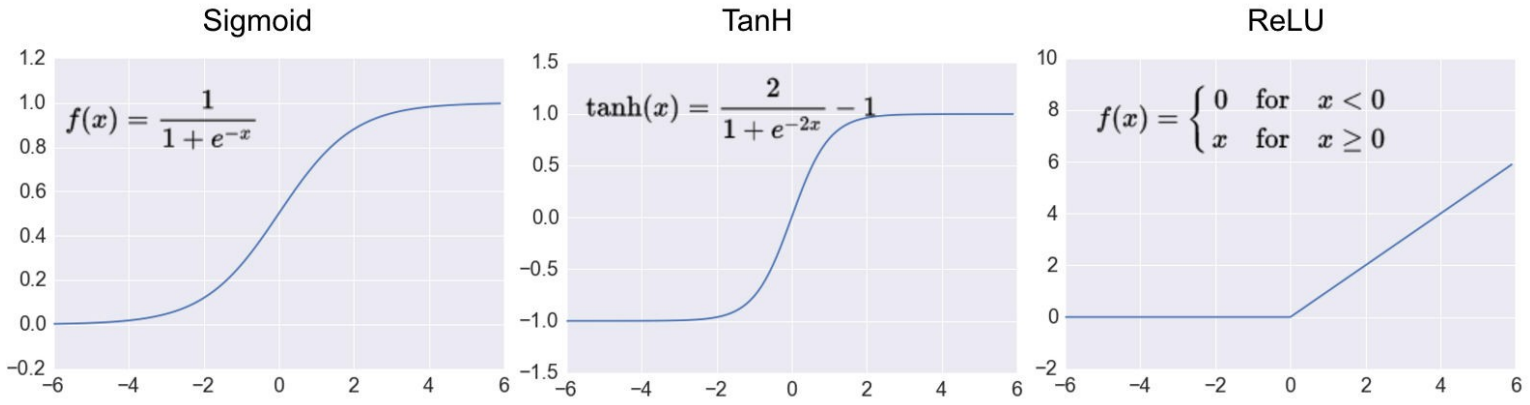
Other types of layer include:

- Noise
- Pooling
- Normalization
- Embedding
- And many more...

Activations

All your favorite activations are available:

1. Sigmoid, tanh, ReLu, softplus, hard sigmoid, linear
2. Advanced activations implemented as a layer (after desired neural layer)
3. Advanced activations: LeakyReLu, PReLu, ELU, Parametric Softplus, Thresholded linear and Thresholded Relu



Losses (Objectives)

- Loss functions are used to compare network's predicted output with the real output, in each pass of the backpropagations algorithm
- Loss functions are used to tell the model how the weights should be updated
- Common loss functions are
  - Mean Squared error
  - Cross Entropy
  - etc

Losses (Objectives)

```
model.compile(loss='mean_squared_error', optimizer='sgd')
```

binary_crossentropy
categorical_crossentropy
mean_squared_error
sparse_categorical_crossentropy
kullback_leibler_divergence
poisson
cosine_proximity

mean_absolute_error
mean_absolute_percentage_error
mean_squared_logarithmic_error
squared_hinge
hinge
categorical_hinge
logcosh

In case you want to convert your categorical data:

```
from keras.utils.np_utils import to_categorical
categorical_labels = to_categorical(int_labels, num_classes=None)
```

Optimizers

Optimizers are strategies used to update the network's weights in the backpropagation algorithm
The most simple optimizer is SGD, but there are many other you can choose

An optimizer is one of the two arguments required for compiling a Keras model:

```
from keras import optimizers
model = Sequential()
model.add(Dense(64, kernel_initializer='uniform', input_shape=(10,)))
model.add(Activation('tanh'))

sgd = optimizers.SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True)
model.compile(loss='mean_squared_error', optimizer=sgd)
```

or you can call it by its name. In the latter case, the default parameters for the optimizer will be used.

```
# pass optimizer by name: default parameters will be used
model.compile(loss='mean_squared_error', optimizer='sgd')
```

Optimizers - More

1. **SGD :**Stochastic gradient descent optimizer.
2. **RMSprop:** RMSProp optimizer is usually a good choice for recurrent neural networks.
   Adagrad
3. **Nadam :** Much like Adam is essentially RMSprop with momentum, Nadam is Adam RMSprop with Nesterov momentum
4. **Adadelta**
5. **Adam**

```
keras.optimizers.SGD(lr=0.01, momentum=0.0, decay=0.0, nesterov=False)
keras.optimizers.RMSprop(lr=0.001, rho=0.9, epsilon=1e-08, decay=0.0)
keras.optimizers.Adagrad(lr=0.01, epsilon=1e-08, decay=0.0)
keras.optimizers.Adadelta(lr=1.0, rho=0.95, epsilon=1e-08, decay=0.0)
keras.optimizers.Adam(lr=0.001, beta_1=0.9, beta_2=0.999, epsilon=1e-08,
decay=0.0) keras.optimizers.Nadam(lr=0.002, beta_1=0.9, beta_2=0.999, epsilon=1e-
08, schedule_decay=0.004)
```

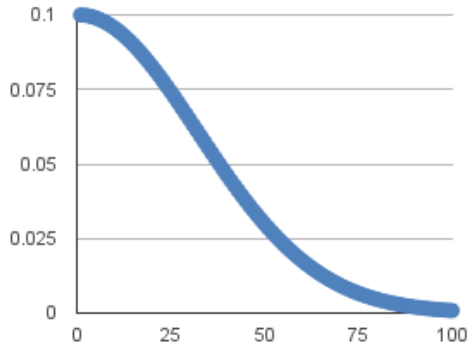
You can also use a wrapper class for native TensorFlow optimizers **TFOptimizer:**.

```
keras.optimizers.TFOptimizer(optimizer)
```

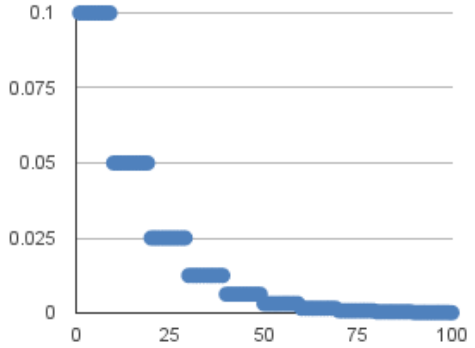
Most optimizers can be tuned using hyper parameters, such as learning rate to use, whether or not to use momentum

Learning Rate Scheduler

In Keras you have two types of learning rate schedule:



a time-based learning rate schedule.



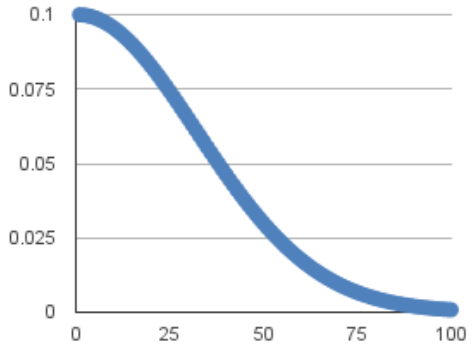
a drop-based learning rate schedule.

Time Based Learning Rate

LearningRate = LearningRate * 1/(1 + decay * epoch)

if we use the initial learning rate value of 0.1 and the decay of 0.001, the first 5 epochs will adapt the learning rate as follows:

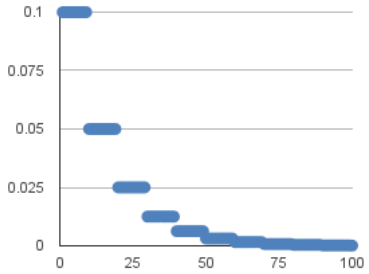
Epoch	Learning Rate
1	0.1
2	0.0999000999
3	0.0997006985
4	0.09940249103
5	0.09900646517



Drop Based Learning Rate

LearningRateScheduler: a function that takes an epoch index as input (integer, indexed from 0) and returns a new learning rate as output (float).

```
# learning rate schedule
def step_decay(epoch):
	initial_lrate = 0.1
	drop = 0.5
	epochs_drop = 10.0
	lrate = initial_lrate * math.pow(drop, math.floor((1+epoch)/epochs_drop))
	return lrate

lrate = LearningRateScheduler(step_decay)
callbacks_list = [lrate]
# Fit the model
model.fit(X, Y, validation_split=0.33, epochs=50, batch_size=28, callbacks=callbacks_list, verbose=2)
```



Tips for using Learning Rate

Increase the initial learning rate.

- Because the learning rate will very likely decrease, start with a larger value to decrease from. A larger learning rate will result in a lot larger changes to the weights, at least in the beginning, allowing you to benefit from the fine tuning later.

Use a large momentum.

- Using a larger momentum value will help the optimization algorithm to continue to make updates in the right direction when your learning rate shrinks to small values.

Experiment with different schedules.

- It will not be clear which learning rate schedule to use so try a few with different configuration options and see what works best on your problem. Also try schedules that change exponentially and even schedules that respond to the accuracy of your model on the training or test datasets.

Metrics

1. Accuracy
   - **binary_accuracy**
   - **categorical_accuracy**
   - **sparse_categorical_accuracy**
   - **top_k_categorical_accuracy**
   - **sparse_top_k_categorical_accuracy**
2. Precision
3. Recall
4. F1Score

```
from keras import metrics
model.compile(loss='categorical_crossentropy',
optimizer='adadelta',
metrics=['accuracy', 'f1score', 'precision', 'recall'])
```

```
1280/5640 [=====>.....] - ETA: 20s - loss: 1.5566 - fmeasure: 0.8134 - precision: 0.8421 - recall: 0.7867
```

Metrics - Custom

Custom metrics can be passed at the compilation step. The function would need to take `(y_true, y_pred)` as arguments and return a single tensor value.

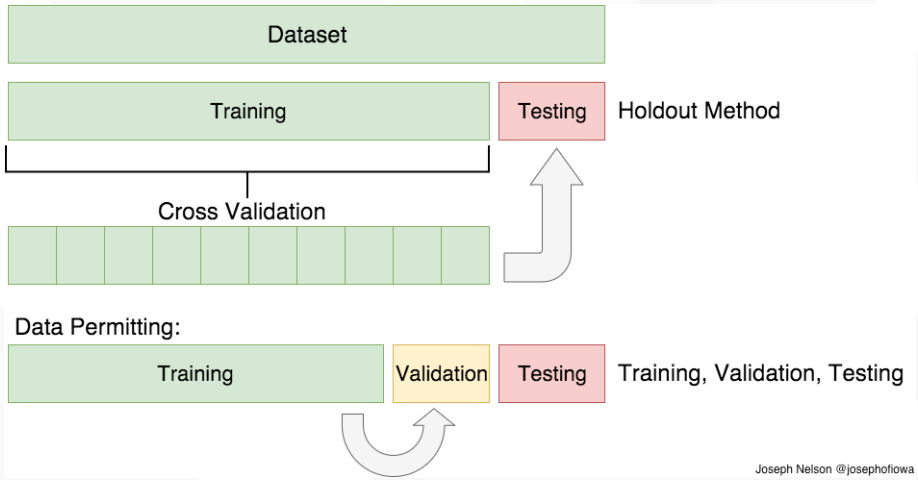
```
import keras.backend as K
def mean_pred(y_true, y_pred):
    return K.mean(y_pred)

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

Performance evaluation strategies

The large amount of data and the complexity of the models require very long training times.

Keras provides a three convenient ways of evaluating your deep learning algorithms :

- Use an automatic verification dataset.
- Use a manual verification dataset.
- Use a manual k-Fold Cross Validation.



Automatic Verification Dataset

Keras can separate a portion of your training data into a validation dataset and evaluate the performance of your model on that validation dataset each epoch

```
model.fit(X, Y, validation_split=0.33, epochs=150, batch_size=10)
```

```
...
Epoch 148/150
514/514 [==============================] - 0s - loss: 0.5219 - acc: 0.7354 - val_loss: 0.5414 - val_acc: 0.7520
Epoch 149/150
514/514 [==============================] - 0s - loss: 0.5089 - acc: 0.7432 - val_loss: 0.5417 - val_acc: 0.7520
Epoch 150/150
514/514 [==============================] - 0s - loss: 0.5148 - acc: 0.7490 - val_loss: 0.5549 - val_acc: 0.7520
```

Manual Verification Dataset

Keras also allows you to manually specify the dataset to use for validation during training.

```
# MLP with manual validation set
from sklearn.model_selection import train_test_split
import numpy
# fix random seed for reproducibility
seed = 7
numpy.random.seed(seed)
X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size=0.33, random_state=seed)

# Fit the model
model.fit(X_train, y_train, validation_data=(X_test,y_test), epochs=150, batch_size=10)
```

Manual k-Fold Cross Validation

```
from sklearn.model_selection import StratifiedKFold
kfold = StratifiedKFold(n_splits=10, shuffle=True, random_state=seed)
cvscores = []
for train, test in kfold.split(X, Y):
  # create model
        model = Sequential()
        model.add(Dense(12, input_dim=8, activation='relu'))
        model.add(Dense(8, activation='relu'))
        model.add(Dense(1, activation='sigmoid'))
        model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
        # Fit the model
        model.fit(X[train], Y[train], epochs=150, batch_size=10, verbose=0)
        # evaluate the model
        scores = model.evaluate(X[test], Y[test], verbose=0)
        print("%s: %.2f%%" % (model.metrics_names[1], scores[1]*100))
        cvscores.append(scores[1] * 100)
print("%.2f%% (+/- %.2f%%)" % (numpy.mean(cvscores), numpy.std(cvscores)))
```

```
acc: 77.92%
acc: 68.83%
acc: 72.73%
acc: 64.94%
acc: 77.92%
acc: 35.06%
acc: 74.03%
acc: 68.83%
acc: 34.21%
acc: 72.37%
64.68% (+/- 15.50%)
```

Regularizers

Nearly everything in Keras can be **regularized** to avoid overfitting

In addition to the Dropout layer, there are all sorts of other regularizers available, such as:

- Weight regularizers
- Bias regularizers
- Activity regularizers

```
from keras import regularizers
model.add(Dense(64, input_dim=64,
kernel_regularizer=regularizers.l2(0.01),
activity_regularizer=regularizers.l1(0.01)))
```

Architecture/Weight Saving and Loading

Saving/loading whole models (architecture + weights + optimizer state)

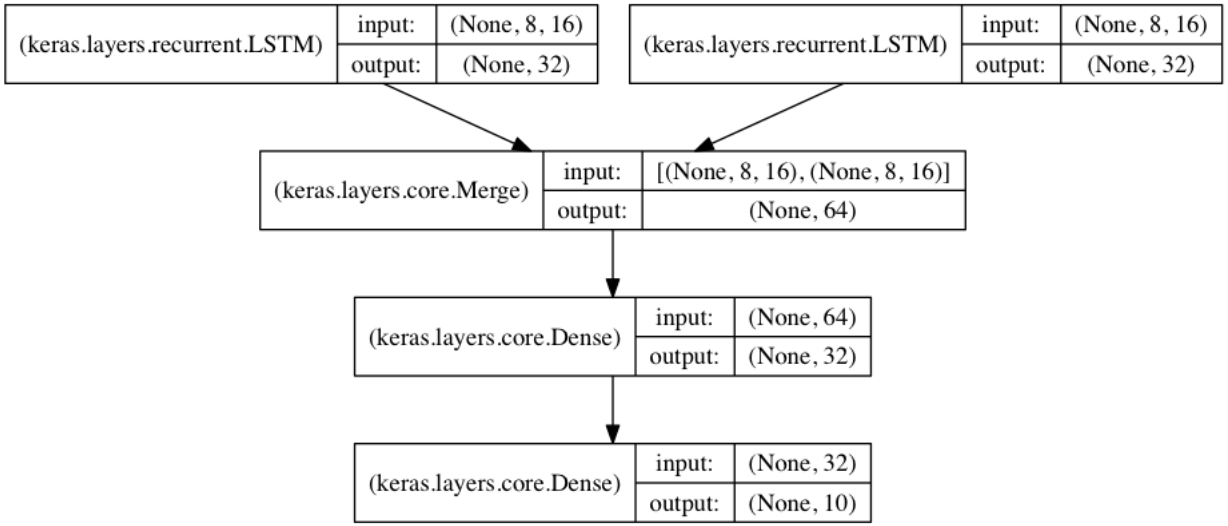
```
from keras.models import load_model
model.save('my_model.h5') # creates a HDF5 file 'my_model.h5'
del model # deletes the existing model
# returns a compiled model
# identical to the previous one
model = load_model('my_model.h5')
```

You can also save/load only a model's architecture or weights only.

Model Visualization

In case you want an image of your model :

```
from keras.utils import plot_model
plot_model(model, to_file='model.png')
```



Callbacks

Allow for function call during training

- Callbacks can be called at different points of training (batch or epoch)
- Existing callbacks: Early Stopping, weight saving after epoch
- Easy to build and implement, called in training function, fit()

Callbacks - Examples

TerminateOnNaN : Callback that terminates training when a NaN loss is encountered.

EarlyStopping: Stop training when a monitored quantity has stopped improving.

ModelCheckpoint : Save the model after every epoch.

```
keras.callbacks.ModelCheckpoint(filepath, monitor='val_loss', verbose=0,
save_best_only=False, save_weights_only=False, mode='auto', period=1)
```

ReduceLROnPlateau: Reduce learning rate when a metric has stopped improving.

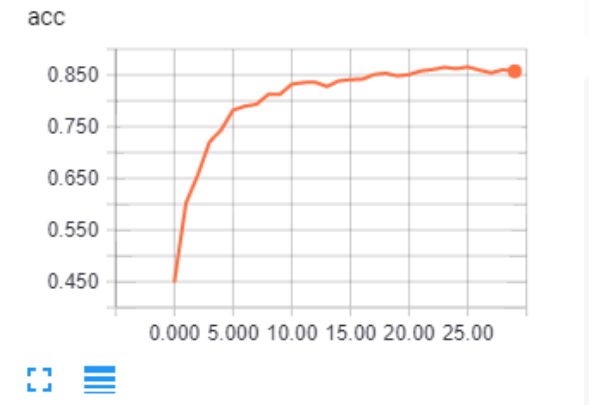
```
keras.callbacks.ReduceLROnPlateau(monitor='val_loss', factor=0.1,
patience=10, verbose=0, mode='auto', epsilon=0.0001, cooldown=0, min_lr=0)
```

Also, You can create a custom callback by extending the base class `keras.callbacks.Callback`

Keras + TensorBoard: Visualizing Learning

- TensorBoard is a visualization tool provided with TensorFlow.
- This callback writes a log for TensorBoard, which allows you to visualize dynamic graphs of your training and test metrics, as well as activation histograms for the different layers in your model.

```
tbCallBack= keras.callbacks.TensorBoard(log_dir='./logs',
histogram_freq=0, batch_size=32, write_graph=True,
write_grads=False, write_images=False, embeddings_freq=0,
embeddings_layer_names=None, embeddings_metadata=None)
```




```
# Use your terminal
tensorboard --logdir path_to_current_dir/logs
```

Keras + Scikit

You can integrate Keras models into a **Scikit-learn** Pipeline

- There are special wrapper functions available on Keras to help you implement the methods that are expected by a scikit-learn classifier, such as fit(), predict(), predict_proba(),etc.
- You can also use things like scikit-learn's grid_search, to do model selection on Keras models, to decide what are the best hyperparameters for a given task.

Keras: CPU/GPU

If your computer has a good graphics card, it can be used to speed up model training

All models up to now were trained using the GPU, and force keras to use the CPU instead

Model Architecture - Cheat Sheet

Multilayer Perceptron (MLP)

Binary Classification

```
>>> from keras.layers import Dense
>>> model.add(Dense(12,
                    input_dim=8,
                    kernel_initializer='uniform',
                    activation='relu'))
>>> model.add(Dense(8,kernel_initializer='uniform',activation='relu'))
>>> model.add(Dense(1,kernel_initializer='uniform',activation='sigmoid'))
```

Multi-Class Classification

```
>>> from keras.layers import Dropout
>>> model.add(Dense(512,activation='relu',input_shape=(784,)))
>>> model.add(Dropout(0.2))
>>> model.add(Dense(512,activation='relu'))
>>> model.add(Dropout(0.2))
>>> model.add(Dense(10,activation='softmax'))
```

Regression

```
>>> model.add(Dense(64,activation='relu',input_dim=train_data.shape[1]))
>>> model.add(Dense(1))
```

Convolutional Neural Network (CNN)

```
>>> from keras.layers import Activation,Conv2D,MaxPooling2D,Flatten
>>> model2.add(Conv2D(32,(3,3),padding='same',input_shape=x_train.shape[1:]))
>>> model2.add(Activation('relu'))
>>> model2.add(Conv2D(32,(3,3)))
>>> model2.add(Activation('relu'))
>>> model2.add(MaxPooling2D(pool_size=(2,2)))
>>> model2.add(Dropout(0.25))
>>> model2.add(Conv2D(64,(3,3), padding='same'))
>>> model2.add(Activation('relu'))
>>> model2.add(Conv2D(64,(3, 3)))
>>> model2.add(Activation('relu'))
>>> model2.add(MaxPooling2D(pool_size=(2,2)))
>>> model2.add(Dropout(0.25))
>>> model2.add(Flatten())
>>> model2.add(Dense(512))
>>> model2.add(Activation('relu'))
>>> model2.add(Dropout(0.5))
>>> model2.add(Dense(num_classes))
>>> model2.add(Activation('softmax'))
```

Recurrent Neural Network (RNN)

```
>>> from keras.klayers import Embedding,LSTM
>>> model3.add(Embedding(20000,128))
>>> model3.add(LSTM(128,dropout=0.2,recurrent_dropout=0.2))
>>> model3.add(Dense(1,activation='sigmoid'))
```

Compile Model - Cheat Sheet

Compile Model

MLP: Binary Classification

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

MLP: Multi-Class Classification

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

MLP: Regression

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

Recurrent Neural Network

```
>>> model3.compile(loss='binary_crossentropy',
                   optimizer='adam',
                   metrics=['accuracy'])
```

Training, Evaluating and Prediction – Cheat Sheet

Model Training

```
>>> model3.fit(x_train4,
               y_train4,
               batch_size=32,
               epochs=15,
               verbose=1,
               validation_data=(x_test4,y_test4))
```

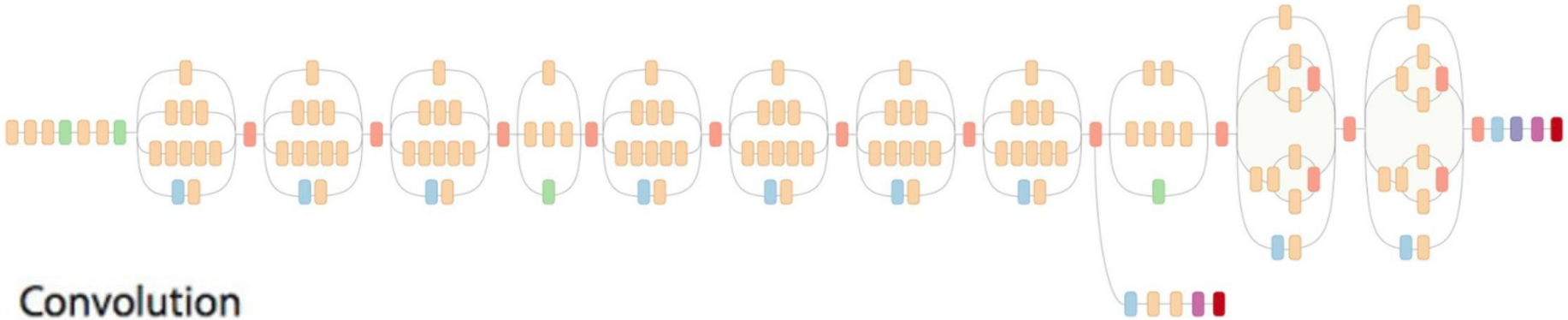
Evaluate Your Model's Performance

```
>>> score = model3.evaluate(x_test,
                            y_test,
                            batch_size=32)
```

Prediction

```
>>> model3.predict(x_test4, batch_size=32)
>>> model3.predict_classes(x_test4,batch_size=32)
```

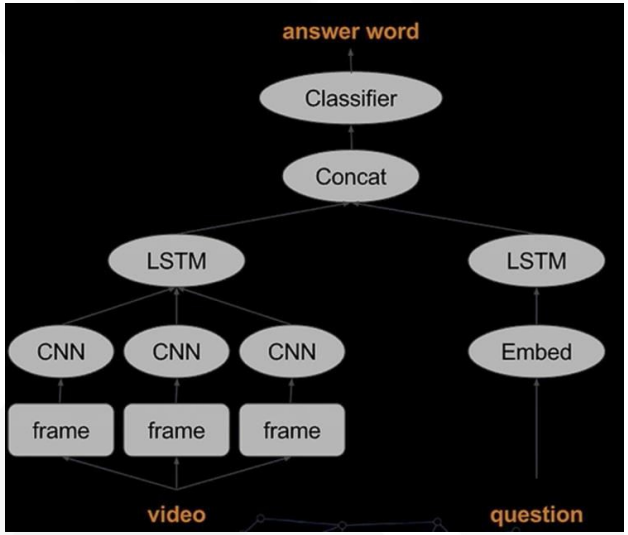
Inception (v3)



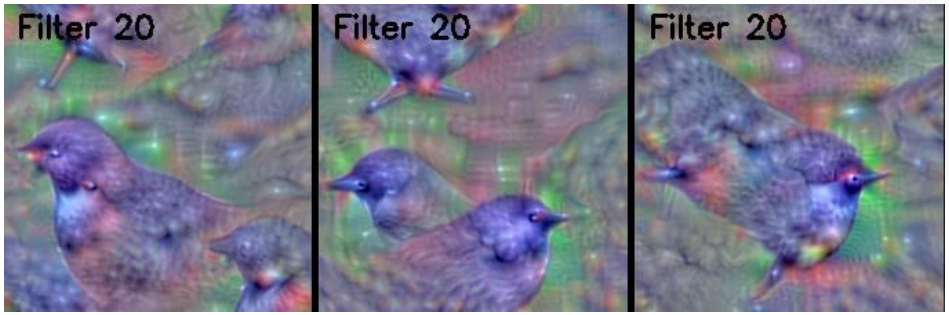
Convolution
AvgPool
MaxPool
Concat
Dropout
Fully connected
Softmax

Model Code

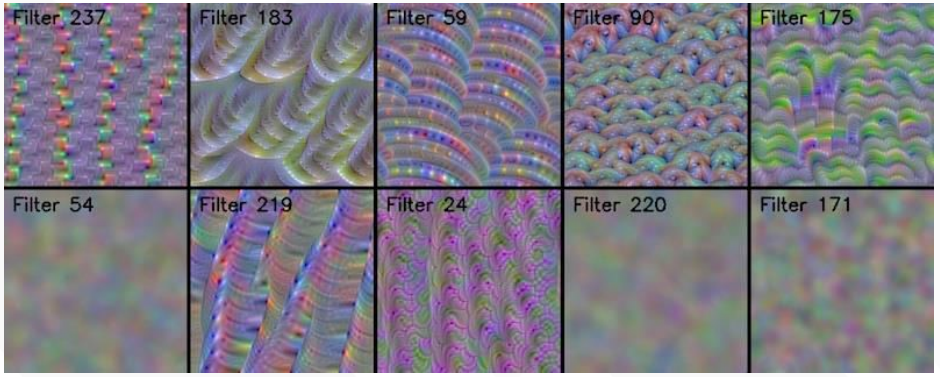
```
video = tf.keras.layers.Input(shape=(None, 150, 150, 3))
cnn = tf.keras.applications.InceptionV3(weights='imagenet',
                                        include_top=False,
                                        pool='avg')

cnn.trainable = False
encoded_frames = tf.keras.layers.TimeDistributed(cnn)(video)
encoded_vid = tf.layers.LSTM(256)(encoded_frames)

question = tf.keras.layers.Input(shape=(100), dtype='int32')
x = tf.keras.layers.Embedding(10000, 256, mask_zero=True)(question)
encoded_q = tf.keras.layers.LSTM(128)(x)

x = tf.keras.layers.concat([encoded_vid, encoded_q])
x = tf.keras.layers.Dense(128, activation=tf.nn.relu)(x)
outputs = tf.keras.layers.Dense(1000)(x)

model = tf.keras.models.Model([video, question], outputs)
model.compile(optimizer=tf.AdamOptimizer(),
              loss=tf.softmax_crossentropy_with_logits)
```



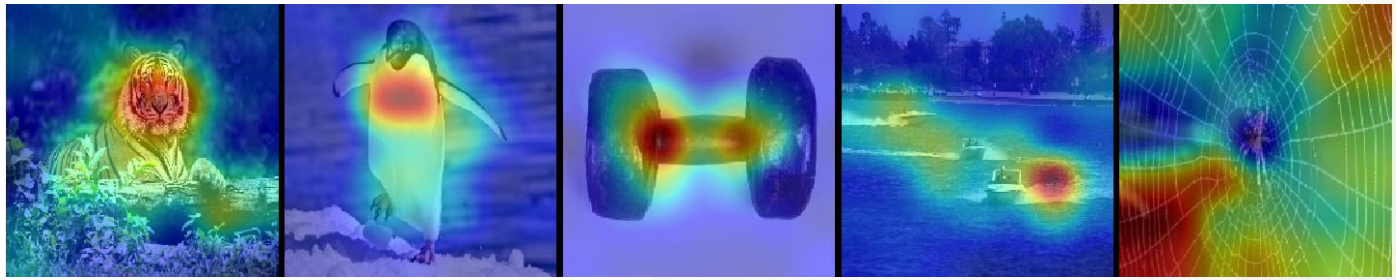
Visualization Plugin: Toolbox for Keras



Dense layer visualization: How can we assess whether a network is over/under fitting or generalizing well?

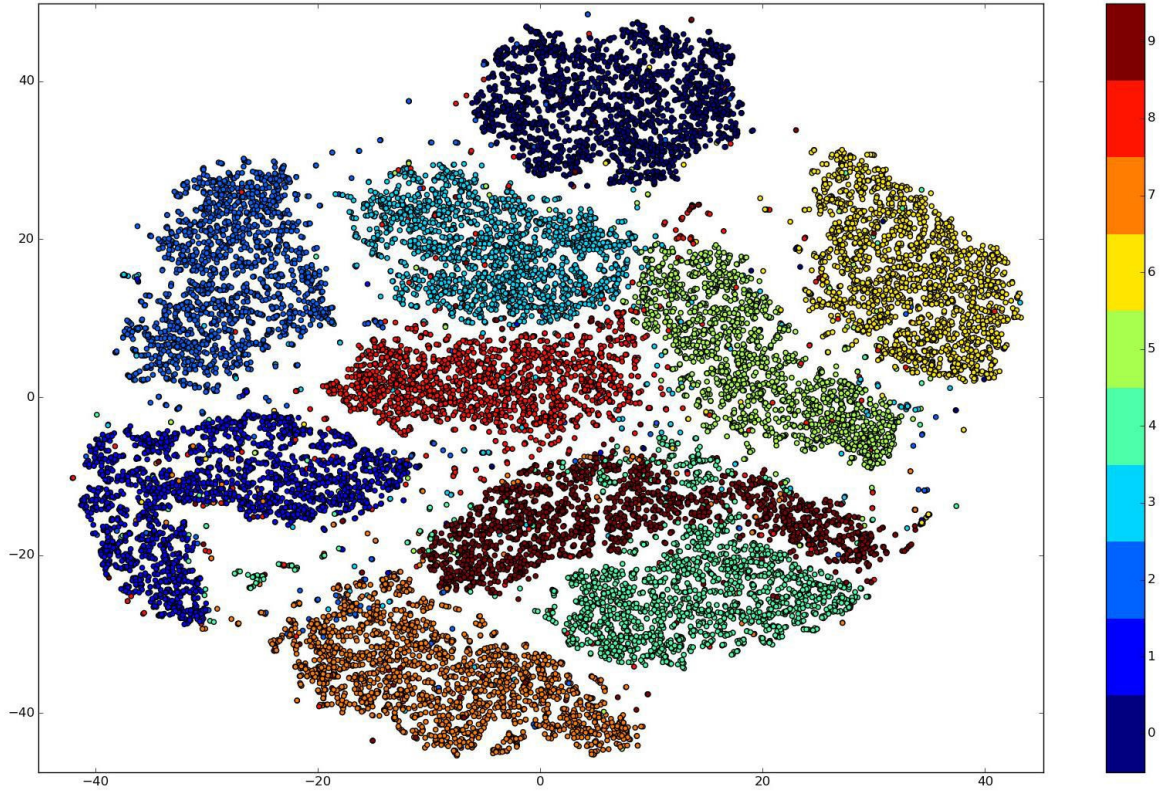


Conv filter visualization



Attention Maps :How can we assess whether a network is attending to correct parts of the image in order to generate a decision?

More Visualization tools: T-SNE



t-Distributed Stochastic Neighbor Embedding (t-SNE) is a (prize-winning) technique for dimensionality reduction that is particularly well suited for the visualization of high-dimensional datasets.

Contact Us

Visit us on: http://www.analytixlabs.in/

For more information, please contact us: http://www.analytixlabs.co.in/contact-us/

Or email: info@analytixlabs.co.in

Call us we would love to speak with you: (+91) 88021-73069

Join us on:

Twitter - http://twitter.com/#!/AnalytixLabs

Facebook - http://www.facebook.com/analytixlabs

LinkedIn - http://www.linkedin.com/in/analytixlabs

Blog - http://www.analytixlabs.co.in/category/blog/