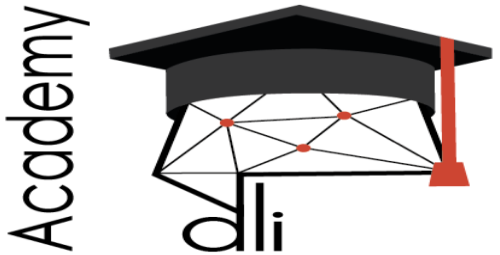


Anatomy of a Deep Learning Application in Python/Keras

Francesco Pugliese, PhD

Italian National Institute of Statistics, Division "Information and Application Architecture", Directorate for methodology and statistical design

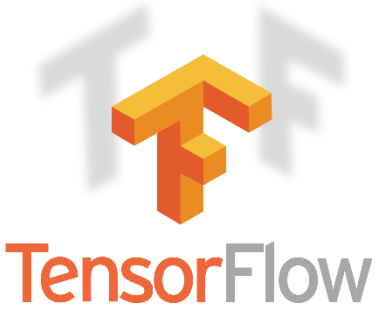
Email **Francesco Pugliese** :

f.pugliese@deeplearningitalia.com

francesco.pugliese@istat.it

Frameworks FOR DEEP LEARNING

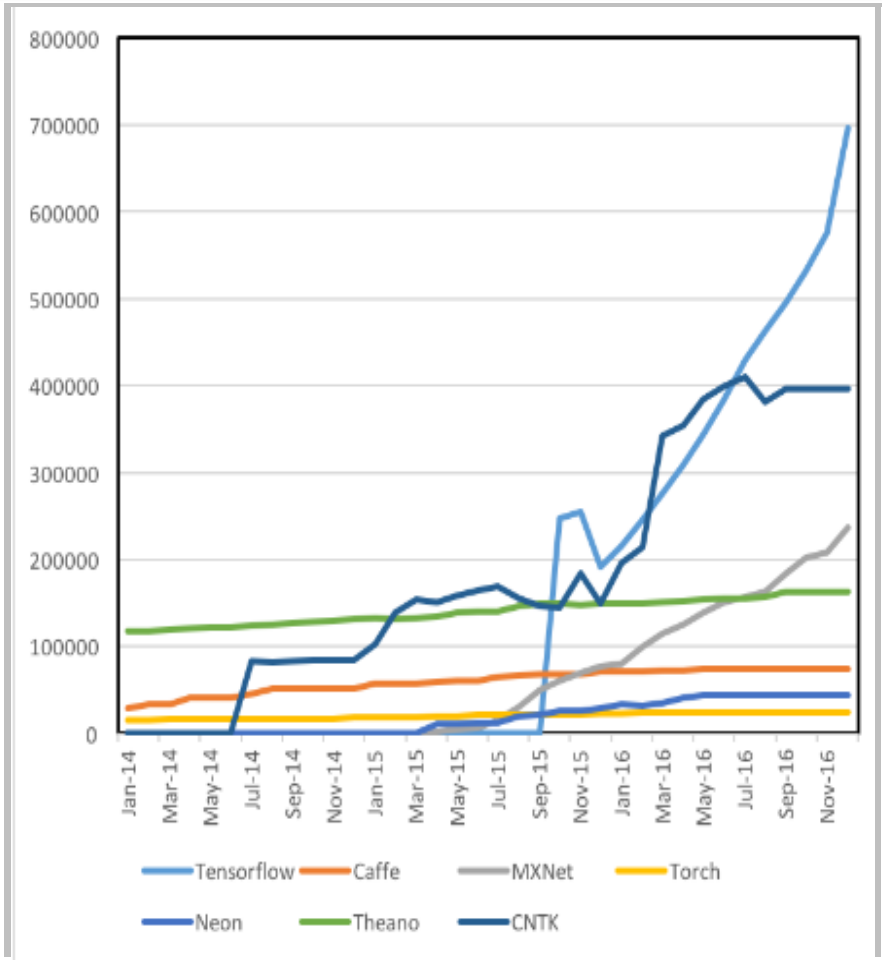
Keras is an higher-level interface for Theano (which works as backend). Keras displays a more intuitive set of abstractions that make it easy to configure neural networks regardless of the backend scientific computing library.

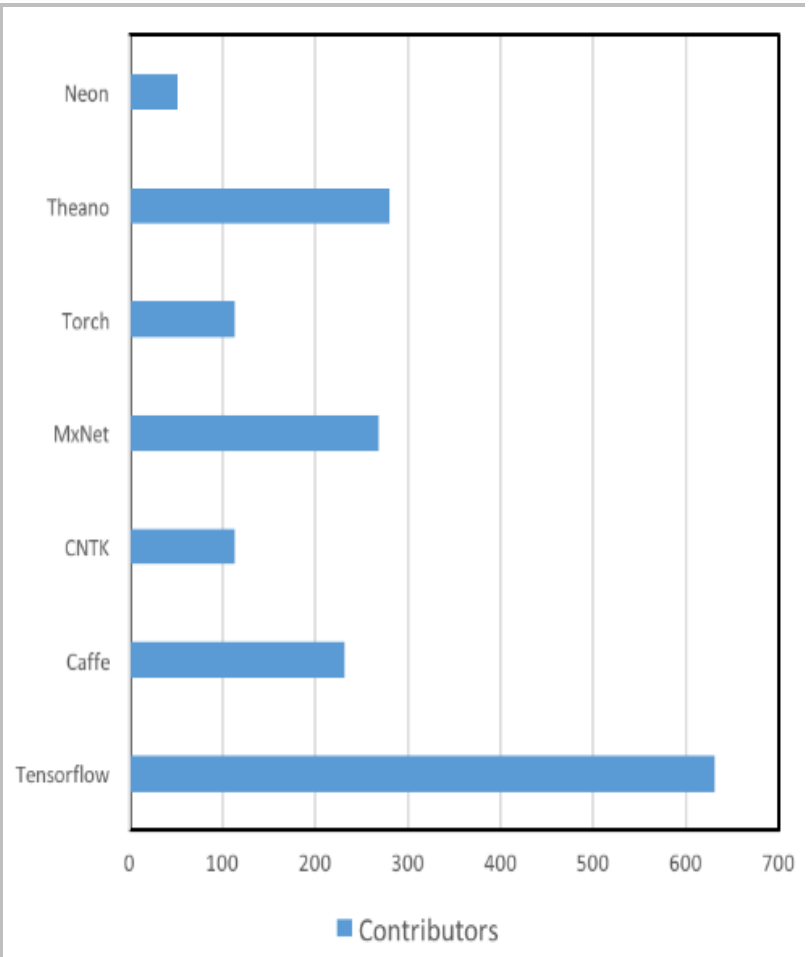


TensorFlow is an open-source software library for dataflow programming across a range of tasks. It is a symbolic math library, and also used for machine learning applications such as neural networks. It is used for both research and production at Google.

PyTorch is an open-source machine learning library for Python, derived from Torch, used for applications such as natural language processing. It is primarily developed by **Facebook's** artificial-intelligence research group, and **Uber's** "Pyro" software for probabilistic programming is built on it.

Comparison of GitHub Contributors for Deep Learning Frameworks

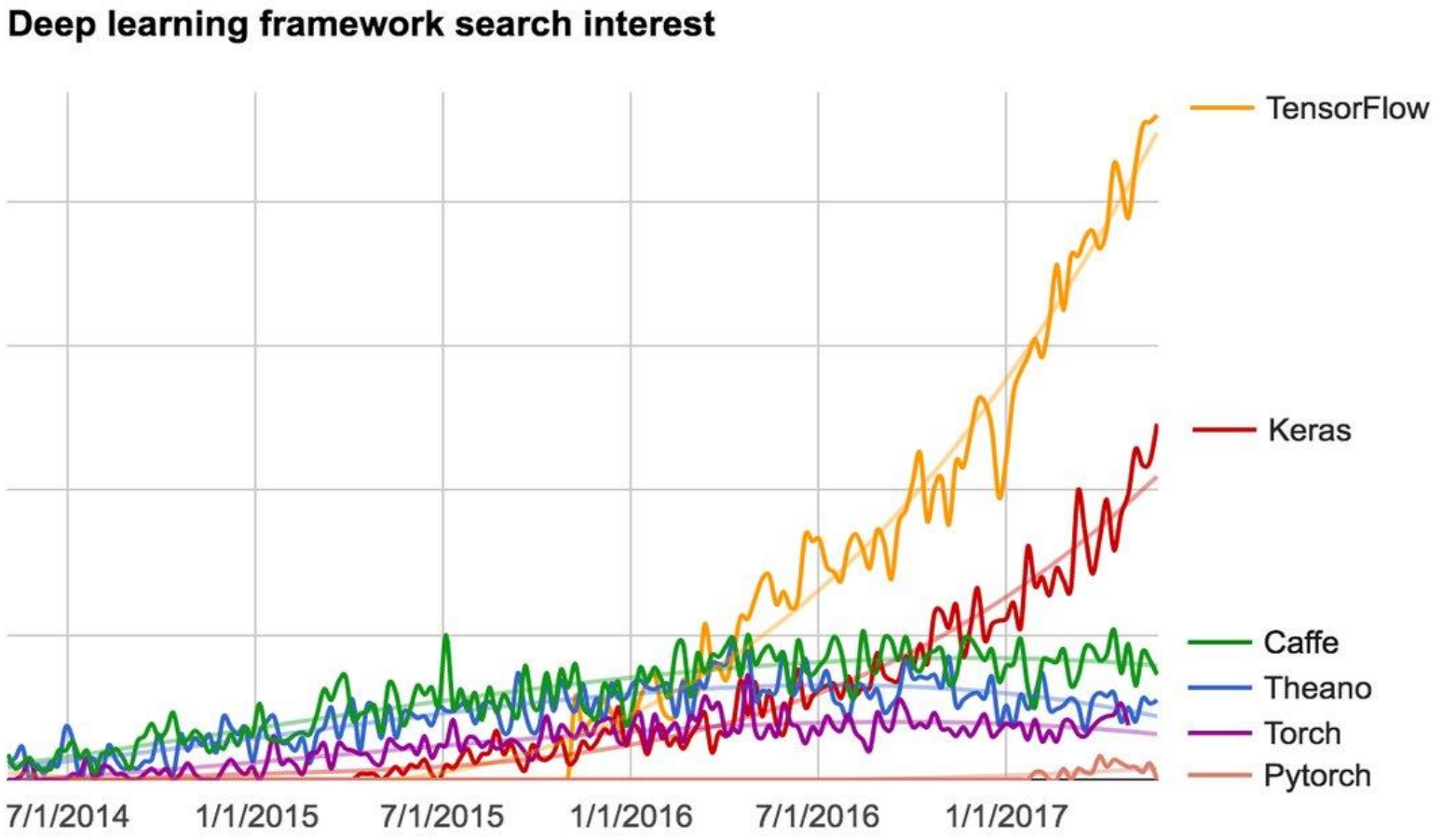




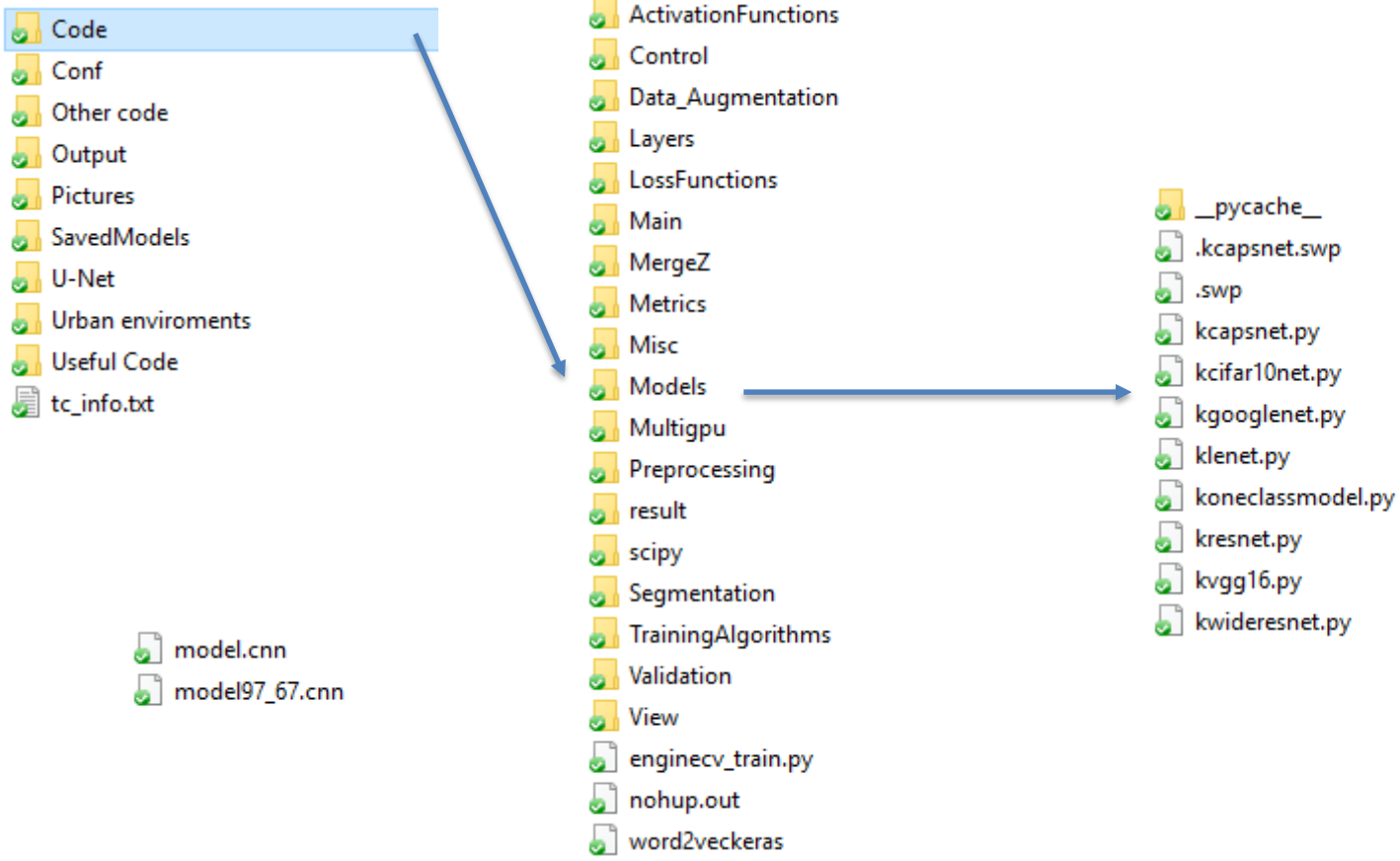
Frameworks FOR DEEP LEARNING

	Languages	Tutorials and training materials	CNN modeling capability	RNN modeling capability	Architecture: easy-to-use and modular front end	Speed	Multiple GPU support	Keras compatible
Theano	Python, C++	++	++	++	+	++	+	+
Tensor-Flow	Python	+++	+++	++	+++	++	++	+
Torch	Lua, Python (new)	+	+++	++	++	+++	++	
Caffe	C++	+	++		+	+	+	
MXNet	R, Python, Julia, Scala	++	++	+	++	++	+++	
Neon	Python	+	++	+	+	++	+	
CNTK	C++	+	+	+++	+	++	+	

Evolution of Keras over years



Back-End Structure



Keras Library

Keras: The Python Deep Learning library

You have just found Keras.

Keras is a high-level neural networks API, written in Python and capable of running on top of TensorFlow, CNTK, or Theano. It was developed with a focus on enabling fast experimentation. *Being able to go from idea to result with the least possible delay is key to doing good research.*

Use Keras if you need a deep learning library that:

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

Read the documentation at Keras.io.

Keras is compatible with: **Python 2.7-3.6.**

Getting started: 30 seconds to Keras Library

The core data structure of Keras is a **model**, a way to organize layers. The simplest type of model is the `Sequential` model, a linear stack of layers. For more complex architectures, you should use the Keras functional API, which allows to build arbitrary graphs of layers.

Here is the `Sequential` model:

```
from keras.models import Sequential

model = Sequential()
```

Stacking layers is as easy as `.add()`:

```
from keras.layers import Dense

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

Getting started: 30 seconds to Keras Library

Once your model looks good, configure its learning process with `.compile()`:

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

If you need to, you can further configure your optimizer. A core principle of Keras is to make things reasonably simple, while allowing the user to be fully in control when they need to (the ultimate control being the easy extensibility of the source code).

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

You can now iterate on your training data in batches:

```
# x_train and y_train are Numpy arrays --just like in the Scikit-Learn API.
model.fit(x_train, y_train, epochs=5, batch_size=32)
```

Getting started: 30 seconds to Keras Library

Alternatively, you can feed batches to your model manually:

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

Evaluate your performance in one line:

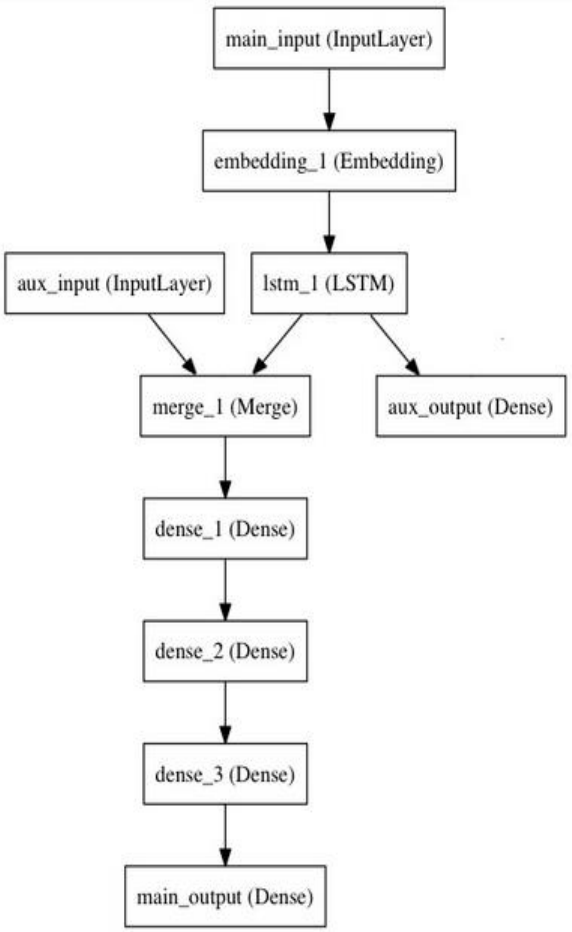
```
loss_and_metrics = model.evaluate(x_test, y_test, batch_size=128)
```

Or generate predictions on new data:

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

Building a question answering system, an image classification model, a Neural Turing Machine, or any other model is just as fast. The ideas behind deep learning are simple, so why should their implementation be painful?

Keras functional model



First example: a densely-connected network

The `Sequential` model is probably a better choice to implement such a network, but it helps to start with something really simple.

- A layer instance is callable (on a tensor), and it returns a tensor
- Input tensor(s) and output tensor(s) can then be used to define a `Model`
- Such a model can be trained just like Keras `Sequential` models.

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

# This returns a tensor
inputs = Input(shape=(784,))

# a layer instance is callable on a tensor, and returns a tensor
x = Dense(64, activation='relu')(inputs)
x = Dense(64, activation='relu')(x)
predictions = Dense(10, activation='softmax')(x)

# This creates a model that includes
# the Input layer and three Dense layers
model = Model(inputs=inputs, outputs=predictions)
model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
model.fit(data, labels)  # starts training
```

Keras functional model

Model class API

In the functional API, given some input tensor(s) and output tensor(s), you can instantiate a `Model` via:

```
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)
```

This model will include all layers required in the computation of `b` given `a`.

In the case of multi-input or multi-output models, you can use lists as well:

```
model = Model(inputs=[a1, a2], outputs=[b1, b2, b3])
```

For a detailed introduction of what `Model` can do, read this guide to the Keras functional API.

Keras Sequential model

The `Sequential` model is a linear stack of layers.

You can create a `Sequential` model by passing a list of layer instances to the constructor:

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

model = Sequential([
    Dense(32, input_shape=(784,)),
    Activation('relu'),
    Dense(10),
    Activation('softmax'),
])
```

You can also simply add layers via the `.add()` method:

```
model = Sequential()
model.add(Dense(32, input_dim=784))
model.add(Activation('relu'))
```

Keras Sequential model

Compilation

Before training a model, you need to configure the learning process, which is done via the `compile` method. It receives three arguments:

- An optimizer. This could be the string identifier of an existing optimizer (such as `rmsprop` or `adagrad`), or an instance of the `Optimizer` class. See: optimizers.
- A loss function. This is the objective that the model will try to minimize. It can be the string identifier of an existing loss function (such as `categorical_crossentropy` or `mse`), or it can be an objective function. See: losses.
- A list of metrics. For any classification problem you will want to set this to `metrics=['accuracy']`. A metric could be the string identifier of an existing metric or a custom metric function.

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

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

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

# For custom metrics
import keras.backend as K

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

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

Keras Sequential model

Training

Keras models are trained on Numpy arrays of input data and labels. For training a model, you will typically use the `fit` function. Read its documentation here.

```
# For a single-input model with 2 classes (binary classification):

model = Sequential()
model.add(Dense(32, activation='relu', input_dim=100))
model.add(Dense(1, activation='sigmoid'))
model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['accuracy'])

# Generate dummy data
import numpy as np
data = np.random.random((1000, 100))
labels = np.random.randint(2, size=(1000, 1))

# Train the model, iterating on the data in batches of 32 samples
model.fit(data, labels, epochs=10, batch_size=32)
```

```
# For a single-input model with 10 classes (categorical classification):

model = Sequential()
model.add(Dense(32, activation='relu', input_dim=100))
model.add(Dense(10, activation='softmax'))
model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# Generate dummy data
import numpy as np
data = np.random.random((1000, 100))
labels = np.random.randint(10, size=(1000, 1))

# Convert labels to categorical one-hot encoding
one_hot_labels = keras.utils.to_categorical(labels, num_classes=10)

# Train the model, iterating on the data in batches of 32 samples
model.fit(data, one_hot_labels, epochs=10, batch_size=32)
```

Keras Sequential model

Multilayer Perceptron (MLP) for multi-class softmax classification:

```
import keras
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation
from keras.optimizers import SGD

# Generate dummy data
import numpy as np
x_train = np.random.random((1000, 20))
y_train = keras.utils.to_categorical(np.random.randint(10, size=(1000, 1)), num_classes=10)
x_test = np.random.random((100, 20))
y_test = keras.utils.to_categorical(np.random.randint(10, size=(100, 1)), num_classes=10)

model = Sequential()
# Dense(64) is a fully-connected layer with 64 hidden units.
# in the first layer, you must specify the expected input data shape:
# here, 20-dimensional vectors.
model.add(Dense(64, activation='relu', input_dim=20))
model.add(Dropout(0.5))
model.add(Dense(64, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(10, activation='softmax'))

sgd = SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True)
model.compile(loss='categorical_crossentropy',
              optimizer=sgd,
              metrics=['accuracy'])

model.fit(x_train, y_train,
          epochs=20,
          batch_size=128)
score = model.evaluate(x_test, y_test, batch_size=128)
```

Keras Sequential model

MLP for binary classification:

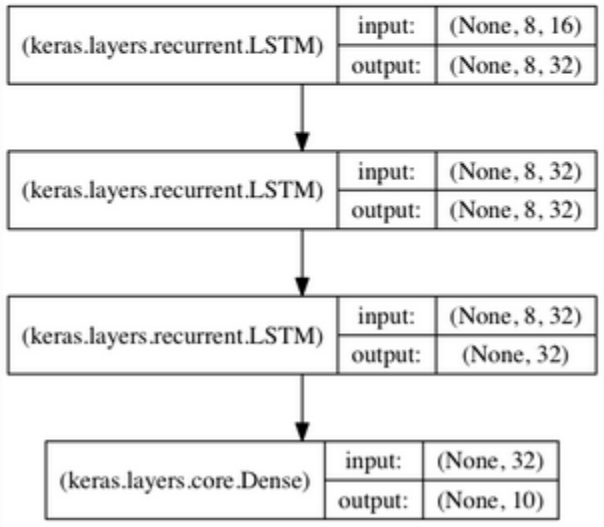
```
import numpy as np
from keras.models import Sequential
from keras.layers import Dense, Dropout

# Generate dummy data
x_train = np.random.random((1000, 20))
y_train = np.random.randint(2, size=(1000, 1))
x_test = np.random.random((100, 20))
y_test = np.random.randint(2, size=(100, 1))

model = Sequential()
model.add(Dense(64, input_dim=20, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(64, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(1, activation='sigmoid'))

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

model.fit(x_train, y_train,
          epochs=20,
          batch_size=128)
score = model.evaluate(x_test, y_test, batch_size=128)
```

Keras Sequential model

Stacked LSTM for sequence classification

In this model, we stack 3 LSTM layers on top of each other, making the model capable of learning higher-level temporal representations.

The first two LSTMs return their full output sequences, but the last one only returns the last step in its output sequence, thus dropping the temporal dimension (i.e. converting the input sequence into a single vector).



Keras Sequential model

Same stacked LSTM model, rendered "stateful"

A stateful recurrent model is one for which the internal states (memories) obtained after processing a batch of samples are reused as initial states for the samples of the next batch. This allows to process longer sequences while keeping computational complexity manageable.

You can read more about stateful RNNs in the FAQ.

```
from keras.models import Sequential
from keras.layers import LSTM, Dense
import numpy as np

data_dim = 16
timesteps = 8
num_classes = 10
batch_size = 32

# Expected input batch shape: (batch_size, timesteps, data_dim)
# Note that we have to provide the full batch_input_shape since the network is stateful.
# the sample of index i in batch k is the follow-up for the sample i in batch k-1.
model = Sequential()
model.add(LSTM(32, return_sequences=True, stateful=True,
               batch_input_shape=(batch_size, timesteps, data_dim)))
model.add(LSTM(32, return_sequences=True, stateful=True))
model.add(LSTM(32, stateful=True))
model.add(Dense(10, activation='softmax'))

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

# Generate dummy training data
x_train = np.random.random((batch_size * 10, timesteps, data_dim))
y_train = np.random.random((batch_size * 10, num_classes))

# Generate dummy validation data
x_val = np.random.random((batch_size * 3, timesteps, data_dim))
y_val = np.random.random((batch_size * 3, num_classes))

model.fit(x_train, y_train,
          batch_size=batch_size, epochs=5, shuffle=False,
          validation_data=(x_val, y_val))
```

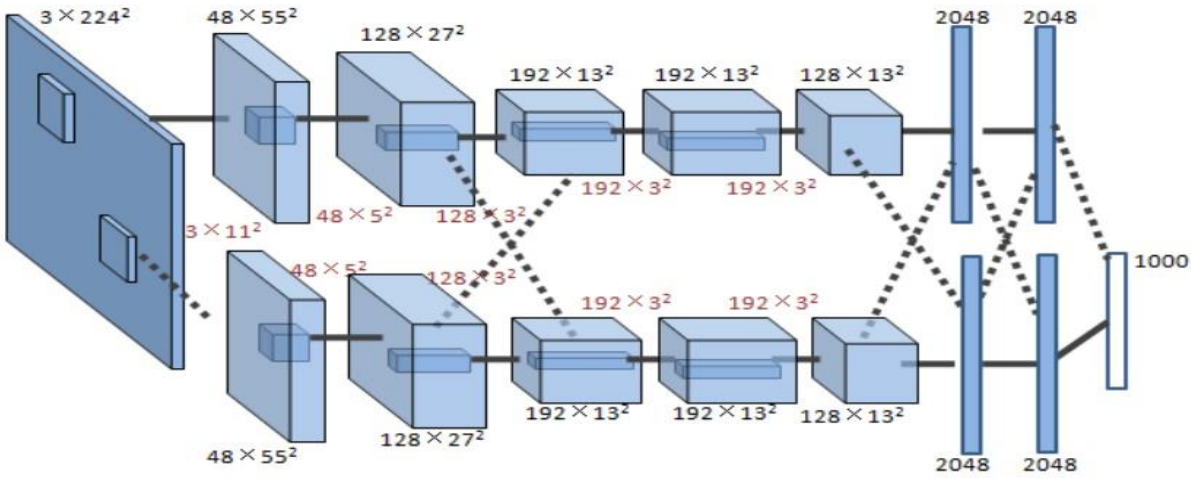
AlexNet - University of Toronto

Critical Feautures (Krizhevsky, A. et al, 2012)

- **8 trainable layers:** 5 convolutional layers and 3 fully connected layers.
- **Max pooling layers** after 1st, 2nd and 5th layer.
- **Rectified Linear Units (ReLUs) (**Nair, V., & Hinton, G. E. 2010).
- **Local Response Normalization.**
- **60 millions parameters, 650 thousands neurons**.
- **Regularizations**: Dropout (prob 0.5 in the first 2 fc layers, Data Augmentation (translactions, horizontal reflections, PCA on RGB).
- **Trained** on 2 **GTX 580** 3 GB GPUs.

Results:

- **1 CNNs: 40.7%** Top-1 Error, **18.2%** Top-5 Error
- **5 CNNs: 38.1%** Top-1 Error,**16.4%** Top-5 Error
- **SIFT+FVs: 26.2%** Top-5 Error (Sánchez, J., et al., 2013).



AlexNet in Keras

```
def get_alexnet(input_shape,nb_classes,mean_flag):
        # code adapted from https://github.com/heuritech/convnets-keras

        inputs = Input(shape=input_shape)

        if mean_flag:
                mean_subtraction = Lambda(mean_subtract, name='mean_subtraction')(inputs)
                conv_1 = Convolution2D(96, 11, 11,subsample=(4,4),activation='relu',
                                       name='conv_1', init='he_normal')(mean_subtraction)
        else:
                conv_1 = Convolution2D(96, 11, 11,subsample=(4,4),activation='relu',
                                       name='conv_1', init='he_normal')(inputs)

        conv_2 = MaxPooling2D((3, 3), strides=(2,2))(conv_1)
        conv_2 = crosschannelnormalization(name="convpool_1")(conv_2)
        conv_2 = ZeroPadding2D((2,2))(conv_2)
        conv_2 = merge([
            Convolution2D(128,5,5,activation="relu",init='he_normal', name='conv_2_'+str(i+1))(
                splittensor(ratio_split=2,id_split=i)(conv_2)
            ) for i in range(2)], mode='concat',concat_axis=1,name="conv_2")

        conv_3 = MaxPooling2D((3, 3), strides=(2, 2))(conv_2)
        conv_3 = crosschannelnormalization()(conv_3)
        conv_3 = ZeroPadding2D((1,1))(conv_3)
        conv_3 = Convolution2D(384,3,3,activation='relu',name='conv_3',init='he_normal')(conv_3)
```

AlexNet in Keras

```
conv_4 = ZeroPadding2D((1,1))(conv_3)
conv_4 = merge([
    Convolution2D(192,3,3,activation="relu", init='he_normal', name='conv_4_'+str(i+1))(
        splittensor(ratio_split=2,id_split=i)(conv_4)
    ) for i in range(2)], mode='concat',concat_axis=1,name="conv_4")

conv_5 = ZeroPadding2D((1,1))(conv_4)
conv_5 = merge([
    Convolution2D(128,3,3,activation="relu",init='he_normal', name='conv_5_'+str(i+1))(
        splittensor(ratio_split=2,id_split=i)(conv_5)
    ) for i in range(2)], mode='concat',concat_axis=1,name="conv_5")

dense_1 = MaxPooling2D((3, 3), strides=(2,2),name="convpool_5")(conv_5)

dense_1 = Flatten(name="flatten")(dense_1)
dense_1 = Dense(4096, activation='relu',name='dense_1',init='he_normal')(dense_1)
dense_2 = Dropout(0.5)(dense_1)
dense_2 = Dense(4096, activation='relu',name='dense_2',init='he_normal')(dense_2)
dense_3 = Dropout(0.5)(dense_2)
dense_3 = Dense(nb_classes,name='dense_3_new',init='he_normal')(dense_3)

prediction = Activation("softmax",name="softmax")(dense_3)

alexnet = Model(input=inputs, output=prediction)

return alexnet
```

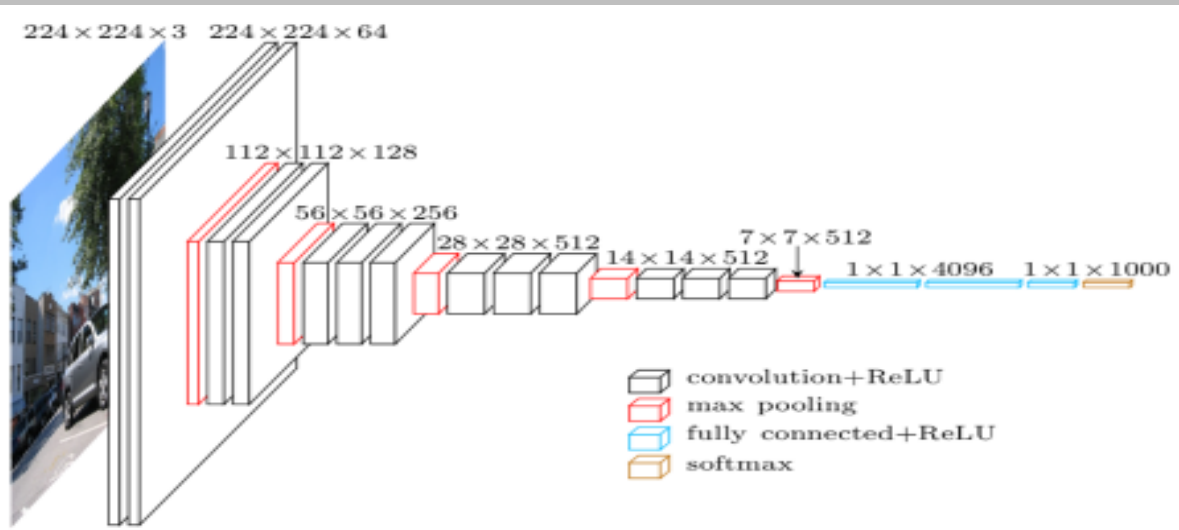
VGG-Net – University of Oxford

Critical Feautures (Simonyan, K., & Zisserman, A., 2014):

- **Kernels with small receptive fields:** 3x3 which is the smallest size to capture the notion of left/right up/down, center. It is easy to see that a stack of two 3×3 conv. layers (without spatial pooling in between) has an effective receptive field of 5×5, and so on.
- Small size **Receptive Field** is a way to increase the nonlinearity of the decision function fields of the conv. layers.
- **Increasing depth architectures: VGG-16** (2xConv3-64, 2xConv3-128, 3xConv3-256, 6xConv3-512, 3xFC), **VGG-19** (same as VGG-16 but with 8xConv3-512).
- **Upside:** less complex topology, outperforms GoogleNet on single-network classification accuracy
- **Downside: 138 millions parameters for VGG-16 !**

Results:

- **Multi ConvNet model :** (D/[256;512]/256,384,512), (E/[256;512]/256,384,512), multi-crop & dense eval: **23.7%** Top-1 Error, **6.8%** Top-5 Error.



VggNet in Keras

```
class VGG_16:

    @staticmethod
    def build(width, height, depth, classes, mul_factor, summary, weightsPath=None):

        model = Sequential()
        model.add(ZeroPadding2D((1,1),input_shape=(depth, height, width)))
        model.add(Convolution2D(64, 3, 3, activation='relu'))
        model.add(ZeroPadding2D((1,1)))
        model.add(Convolution2D(64, 3, 3, activation='relu'))
        model.add(MaxPooling2D((2,2), strides=(2,2)))

        model.add(ZeroPadding2D((1,1)))
        model.add(Convolution2D(128, 3, 3, activation='relu'))
        model.add(ZeroPadding2D((1,1)))
        model.add(Convolution2D(128, 3, 3, activation='relu'))
        model.add(MaxPooling2D((2,2), strides=(2,2)))

        model.add(ZeroPadding2D((1,1)))
        model.add(Convolution2D(256, 3, 3, activation='relu'))
        model.add(ZeroPadding2D((1,1)))
        model.add(Convolution2D(256, 3, 3, activation='relu'))
        model.add(ZeroPadding2D((1,1)))
        model.add(Convolution2D(256, 3, 3, activation='relu'))
        model.add(MaxPooling2D((2,2), strides=(2,2)))
```

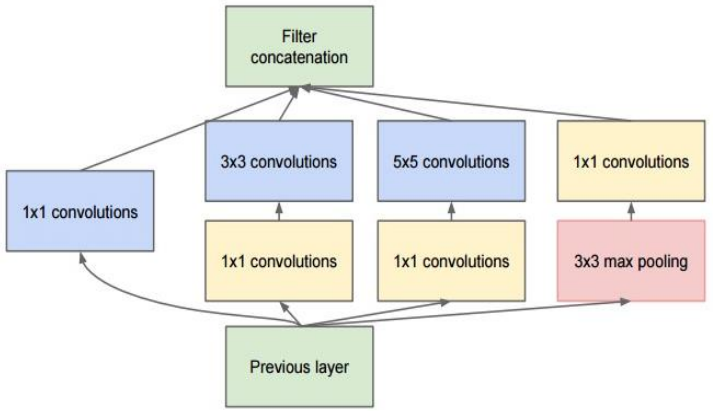
VggNet in Keras

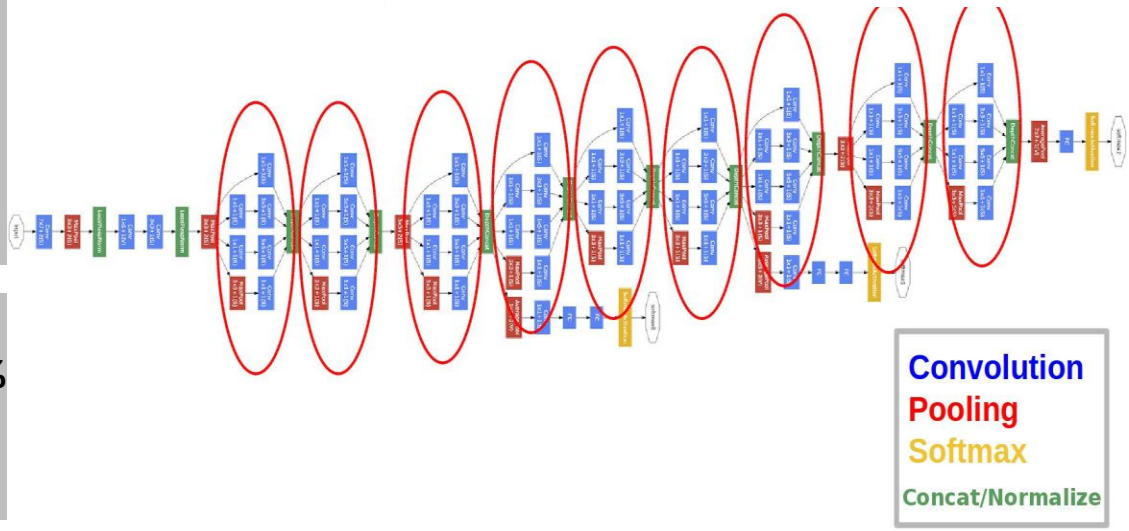
```
model.add(ZeroPadding2D((1,1)))
model.add(Convolution2D(512, 3, 3, activation='relu'))
model.add(ZeroPadding2D((1,1)))
model.add(Convolution2D(512, 3, 3, activation='relu'))
model.add(ZeroPadding2D((1,1)))
model.add(Convolution2D(512, 3, 3, activation='relu'))
model.add(MaxPooling2D((2,2), strides=(2,2)))

model.add(ZeroPadding2D((1,1)))
model.add(Convolution2D(512, 3, 3, activation='relu'))
model.add(ZeroPadding2D((1,1)))
model.add(Convolution2D(512, 3, 3, activation='relu'))
model.add(ZeroPadding2D((1,1)))
model.add(Convolution2D(512, 3, 3, activation='relu'))
model.add(MaxPooling2D((2,2), strides=(2,2)))

model.add(Flatten())
model.add(Dense(4096, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(4096, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(classes, activation='softmax'))

if summary==True:
    model.summary()

if weightsPath:
    model.load_weights(weightsPath)

return model
```

GoogleNet – Google

Critical Feautures (Szegedy, C., et al., 2015):

- **Computationally Effective Deep architecture:** 22 layer
- **Why the name inception, you ask?** Because the module represents a *network within a network*. If you don't get the reference, go watch Christopher Nolan's **"INCEPTION"**, computer scientists are hilarious.
- Inception: it isbasically the parallel combination of 1×1 3×3, and 5×5 convolutional filters.
- **Bottleneck layer:** The great insight of the inception module is the use of 1×1 conv-olutional blocks (NiN) to reduce the number of features before the expensive parallel blocks.
- **Upside: 4 millions parameters!**
- **Downside: Not scalable**!





Results:

- **7 Models Ensemble : 6.67%** Top-5 Error.

GoogleNet in Keras

```
class GoogleNet:

    @staticmethod
    def build(width, height, depth, classes, mul_factor, weightsPath=None):
        input = Input(shape=(depth, height, width))                                    # Set Input Shape
        conv1_7x7_s2 = Convolution2D(64,7,7,subsample=(2,2),border_mode='same',activation='relu',name='conv1/7x7_s2',W_regularizer=l2(0.0002))(input)
        conv1_zero_pad = ZeroPadding2D(padding=(1, 1))(conv1_7x7_s2)
        pool1_helper = PoolHelper()(conv1_zero_pad)
        pool1_3x3_s2 = MaxPooling2D(pool_size=(3,3),strides=(2,2),border_mode='valid',name='pool1/3x3_s2')(pool1_helper)
        pool1_norm1 = LRN(name='pool1/norm1')(pool1_3x3_s2)
        conv2_3x3_reduce = Convolution2D(64,1,1,border_mode='same',activation='relu',name='conv2/3x3_reduce',W_regularizer=l2(0.0002))(pool1_norm1)
        conv2_3x3 = Convolution2D(192,3,3,border_mode='same',activation='relu',name='conv2/3x3',W_regularizer=l2(0.0002))(conv2_3x3_reduce)
        conv2_norm2 = LRN(name='conv2/norm2')(conv2_3x3)
        conv2_zero_pad = ZeroPadding2D(padding=(1, 1))(conv2_norm2)
        pool2_helper = PoolHelper()(conv2_zero_pad)
        pool2_3x3_s2 = MaxPooling2D(pool_size=(3,3),strides=(2,2),border_mode='valid',name='pool2/3x3_s2')(pool2_helper)

        # First Inception Module
        inception_3a_1x1 = Convolution2D(64,1,1,border_mode='same',activation='relu',name='inception_3a/1x1',W_regularizer=l2(0.0002))(pool2_3x3_s2)
        inception_3a_3x3_reduce = Convolution2D(96,1,1,border_mode='same',activation='relu',name='inception_3a/3x3_reduce',W_regularizer=l2(0.0002))(pool2_3x3_s2)
        inception_3a_3x3 = Convolution2D(128,3,3,border_mode='same',activation='relu',name='inception_3a/3x3',W_regularizer=l2(0.0002))(inception_3a_3x3_reduce)
        inception_3a_5x5_reduce = Convolution2D(16,1,1,border_mode='same',activation='relu',name='inception_3a/5x5_reduce',W_regularizer=l2(0.0002))(pool2_3x3_s2)
        inception_3a_5x5 = Convolution2D(32,5,5,border_mode='same',activation='relu',name='inception_3a/5x5',W_regularizer=l2(0.0002))(inception_3a_5x5_reduce)
        inception_3a_pool = MaxPooling2D(pool_size=(3,3),strides=(1,1),border_mode='same',name='inception_3a/pool')(pool2_3x3_s2)
        inception_3a_pool_proj = Convolution2D(32,1,1,border_mode='same',activation='relu',name='inception_3a/pool_proj',W_regularizer=l2(0.0002))(inception_3a_pool)
        inception_3a_output = merge([inception_3a_1x1,inception_3a_3x3,inception_3a_5x5,inception_3a_pool_proj],mode='concat',concat_axis=1,name='inception_3a/output')
```

GoogleNet in Keras

```
# Second Inception Module
inception_3b_1x1 = Convolution2D(128,1,1,border_mode='same',activation='relu',name='inception_3b/1x1',W_regularizer=l2(0.0002))(inception_3a_output)
inception_3b_3x3_reduce = Convolution2D(128,1,1,border_mode='same',activation='relu',name='inception_3b/3x3_reduce',W_regularizer=l2(0.0002))(inception_3a_output)
inception_3b_3x3 = Convolution2D(192,3,3,border_mode='same',activation='relu',name='inception_3b/3x3',W_regularizer=l2(0.0002))(inception_3b_3x3_reduce)
inception_3b_5x5_reduce = Convolution2D(32,1,1,border_mode='same',activation='relu',name='inception_3b/5x5_reduce',W_regularizer=l2(0.0002))(inception_3a_output)
inception_3b_5x5 = Convolution2D(96,5,5,border_mode='same',activation='relu',name='inception_3b/5x5',W_regularizer=l2(0.0002))(inception_3b_5x5_reduce)
inception_3b_pool = MaxPooling2D(pool_size=(3,3),strides=(1,1),border_mode='same',name='inception_3b/pool')(inception_3a_output)
inception_3b_pool_proj = Convolution2D(64,1,1,border_mode='same',activation='relu',name='inception_3b/pool_proj',W_regularizer=l2(0.0002))(inception_3b_pool)
inception_3b_output = merge([inception_3b_1x1,inception_3b_3x3,inception_3b_5x5,inception_3b_pool_proj],mode='concat',concat_axis=1,name='inception_3b/output')
inception_3b_output_zero_pad = ZeroPadding2D(padding=(1, 1))(inception_3b_output)
pool3_helper = PoolHelper()(inception_3b_output_zero_pad)
pool3_3x3_s2 = MaxPooling2D(pool_size=(3,3),strides=(2,2),border_mode='valid',name='pool3/3x3_s2')(pool3_helper)

# Third Inception Module
inception_4a_1x1 = Convolution2D(192,1,1,border_mode='same',activation='relu',name='inception_4a/1x1',W_regularizer=l2(0.0002))(pool3_3x3_s2)
inception_4a_3x3_reduce = Convolution2D(96,1,1,border_mode='same',activation='relu',name='inception_4a/3x3_reduce',W_regularizer=l2(0.0002))(pool3_3x3_s2)
inception_4a_3x3 = Convolution2D(208,3,3,border_mode='same',activation='relu',name='inception_4a/3x3',W_regularizer=l2(0.0002))(inception_4a_3x3_reduce)
inception_4a_5x5_reduce = Convolution2D(16,1,1,border_mode='same',activation='relu',name='inception_4a/5x5_reduce',W_regularizer=l2(0.0002))(pool3_3x3_s2)
inception_4a_5x5 = Convolution2D(48,5,5,border_mode='same',activation='relu',name='inception_4a/5x5',W_regularizer=l2(0.0002))(inception_4a_5x5_reduce)
inception_4a_pool = MaxPooling2D(pool_size=(3,3),strides=(1,1),border_mode='same',name='inception_4a/pool')(pool3_3x3_s2)
inception_4a_pool_proj = Convolution2D(64,1,1,border_mode='same',activation='relu',name='inception_4a/pool_proj',W_regularizer=l2(0.0002))(inception_4a_pool)
inception_4a_output = merge([inception_4a_1x1,inception_4a_3x3,inception_4a_5x5,inception_4a_pool_proj],mode='concat',concat_axis=1,name='inception_4a/output')
loss1_ave_pool = AveragePooling2D(pool_size=(5,5),strides=(3,3),name='loss1/ave_pool')(inception_4a_output)
loss1_conv = Convolution2D(128,1,1,border_mode='same',activation='relu',name='loss1/conv',W_regularizer=l2(0.0002))(loss1_ave_pool)
loss1_flat = Flatten()(loss1_conv)
loss1_fc = Dense(1024,activation='relu',name='loss1/fc',W_regularizer=l2(0.0002))(loss1_flat)
loss1_drop_fc = Dropout(0.7)(loss1_fc)
loss1_classifier = Dense(1000,name='loss1/classifier',W_regularizer=l2(0.0002))(loss1_drop_fc)
loss1_classifier_act = Activation('softmax')(loss1_classifier)
```

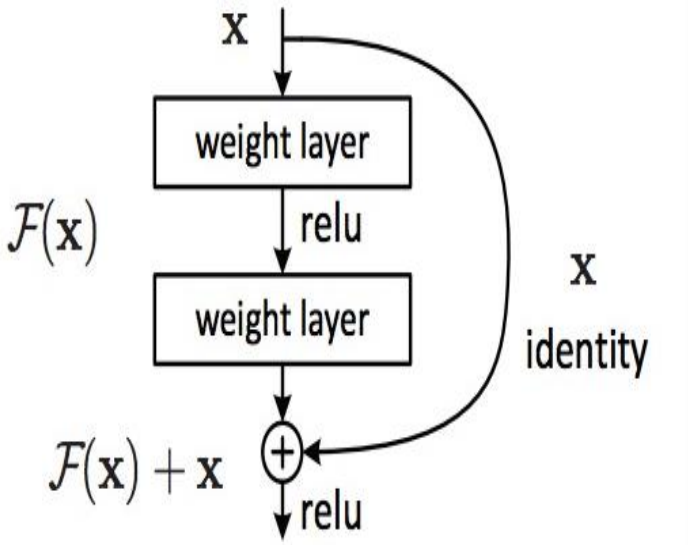
GoogleNet in Keras

```
# Fourth Inception Module
inception_4b_1x1 = Convolution2D(160,1,1,border_mode='same',activation='relu',name='inception_4b/1x1',W_regularizer=l2(0.0002))(inception_4a_output)
inception_4b_3x3_reduce = Convolution2D(112,1,1,border_mode='same',activation='relu',name='inception_4b/3x3_reduce',W_regularizer=l2(0.0002))(inception_4a_output)
inception_4b_3x3 = Convolution2D(224,3,3,border_mode='same',activation='relu',name='inception_4b/3x3',W_regularizer=l2(0.0002))(inception_4b_3x3_reduce)
inception_4b_5x5_reduce = Convolution2D(24,1,1,border_mode='same',activation='relu',name='inception_4b/5x5_reduce',W_regularizer=l2(0.0002))(inception_4a_output)
inception_4b_5x5 = Convolution2D(64,5,5,border_mode='same',activation='relu',name='inception_4b/5x5',W_regularizer=l2(0.0002))(inception_4b_5x5_reduce)
inception_4b_pool = MaxPooling2D(pool_size=(3,3),strides=(1,1),border_mode='same',name='inception_4b/pool')(inception_4a_output)
inception_4b_pool_proj = Convolution2D(64,1,1,border_mode='same',activation='relu',name='inception_4b/pool_proj',W_regularizer=l2(0.0002))(inception_4b_pool)
inception_4b_output = merge([inception_4b_1x1,inception_4b_3x3,inception_4b_5x5,inception_4b_pool_proj],mode='concat',concat_axis=1,name='inception_4b_output')


model = Model(input=input, output=[loss1_classifier])


model.summary()


#if a weights path is supplied (indicating that the model was pre-trained), then load the weights
if weightsPath is not None:
    model.load_wights(weightsPath)

return model
```

Residual Networks – Microsoft

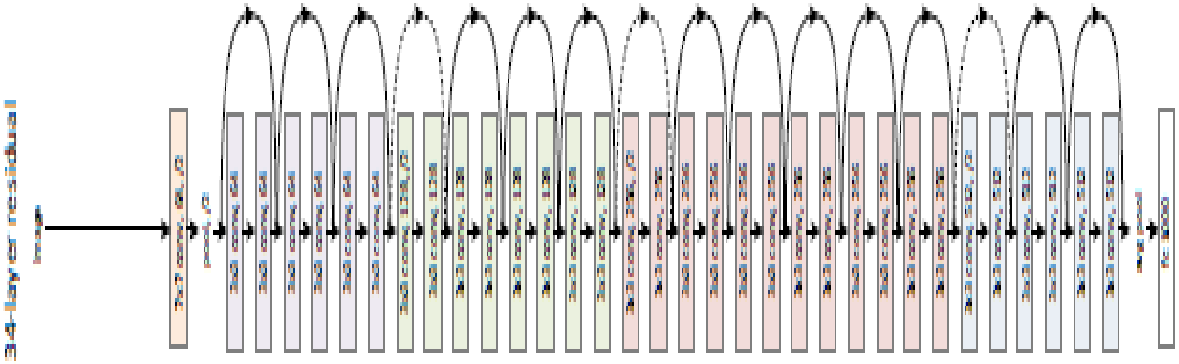
Critical Feautures (He, K., et al., 2016). :

- **Degradation Problem:** Stacking more and more layers **IS NOT** better. With the network depth increasing, accuracy gets saturated and then degrades rapidly! It's an issue of "solvers".
- **Solves the "Degradation problem":** by fitting a residual mapping which is easier to optimize.
- **Shortcut connections:**
- **Very deep architecture:** up to 1202 layers with WideResnet with only **19.4 million parameters**!
- **Upside: Increasing accuracy with more depth**
- **Downside: They don't consider other architectures breakthroughs.**



Results:

- **ResNet : 3.57%** Top-5 Error.
- **CNNs** show superhuman abilities at Image Recognition! **5%** Human estimated Top-5 error (Johnson, R. C., 2015).



ResNet in Keras

```
class WideResNet:

    @staticmethod
    def build(width, height, depth, classes, summary, weightsPath=None):
        n = 8  # depth = 6*n + 4
        k = 4  # widen factor

        img_input = Input(shape=(depth, height, width))

        # one conv at the beginning (spatial size: 32x32)
        x = ZeroPadding2D((1, 1))(img_input)
        x = Conv2D(16, (3, 3))(x)

        # Stage 1 (spatial size: 32x32)
        x = bottleneck(x, n, 16, 16 * k, dropout=0.3, subsample=(1, 1))
        # Stage 2 (spatial size: 16x16)
        x = bottleneck(x, n, 16 * k, 32 * k, dropout=0.3, subsample=(2, 2))
        # Stage 3 (spatial size: 8x8)
        x = bottleneck(x, n, 32 * k, 64 * k, dropout=0.3, subsample=(2, 2))

        x = BatchNormalization(axis=1)(x)
        x = Activation('relu')(x)
        x = AveragePooling2D((8, 8), strides=(1, 1))(x)
        x = Flatten()(x)
        preds = Dense(classes, activation='softmax')(x)

        model = Model(inputs=img_input, outputs=preds)

        if summary==True:
            model.summary()

        #model = to_multi_gpu(model, 2)

        #if a weights path is supplied (indicating that the model was pre-trained), then load the weights
        if weightsPath is not None:
            model.load_wights(weightsPath)

        return model
```

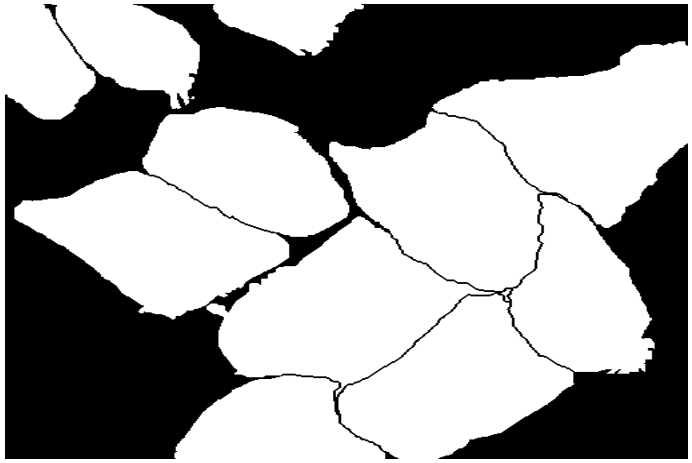
ResNet in Keras

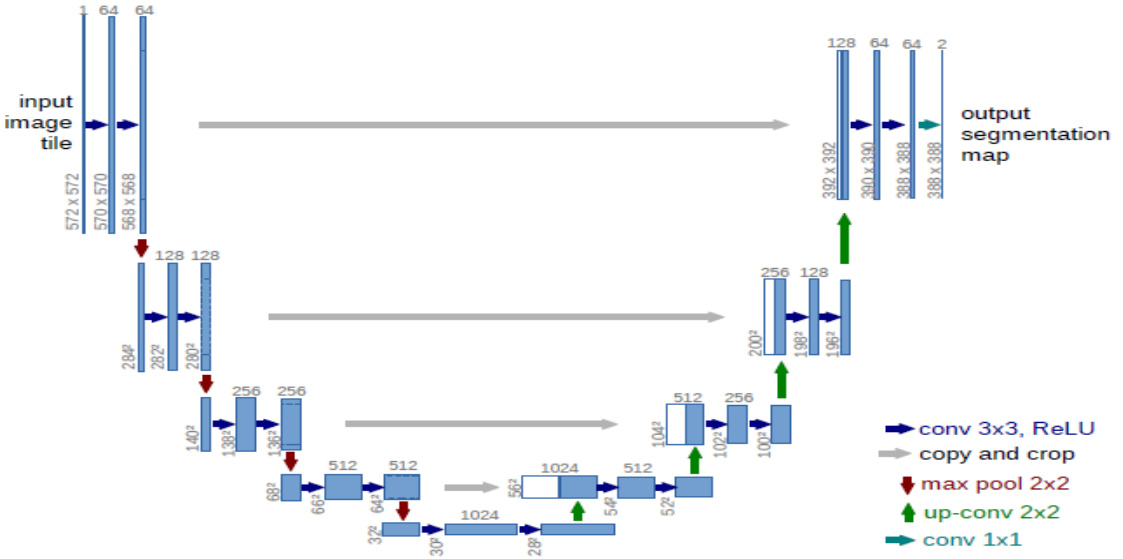
```
def bottleneck(incoming, count, nb_in_filters, nb_out_filters, dropout=None, subsample=(2, 2)):
    outgoing = wide_basic(incoming, nb_in_filters, nb_out_filters, dropout, subsample)
    for i in range(1, count):
        outgoing = wide_basic(outgoing, nb_out_filters, nb_out_filters, dropout, subsample=(1, 1))

    return outgoing


def wide_basic(incoming, nb_in_filters, nb_out_filters, dropout=None, subsample=(2, 2)):
    nb_bottleneck_filter = nb_out_filters

    if nb_in_filters == nb_out_filters:
        # conv3x3
        y = BatchNormalization(axis=1)(incoming)
        y = Activation('relu')(y)
        y = ZeroPadding2D((1, 1))(y)
        y = Conv2D(nb_bottleneck_filter, (3, 3), strides=subsample, kernel_initializer='he_normal', padding='valid')(y)

        # conv3x3
        y = BatchNormalization(axis=1)(y)
        y = Activation('relu')(y)
        if dropout is not None:
            y = Dropout(dropout)(y)
        y = ZeroPadding2D((1, 1))(y)
        y = Conv2D(nb_bottleneck_filter, (3, 3), strides=(1, 1), kernel_initializer='he_normal', padding='valid')(y)

        return add([incoming, y])

    else:  # Residual Units for increasing dimensions
        # common BN, ReLU
        shortcut = BatchNormalization(axis=1)(incoming)
        shortcut = Activation('relu')(shortcut)

        # conv3x3
        y = ZeroPadding2D((1, 1))(shortcut)
        y = Conv2D(nb_bottleneck_filter, (3, 3), strides=subsample, kernel_initializer='he_normal', padding='valid')(y)

        # conv3x3
        y = BatchNormalization(axis=1)(y)
        y = Activation('relu')(y)
        if dropout is not None:
            y = Dropout(dropout)(y)
        y = ZeroPadding2D((1, 1))(y)
        y = Conv2D(nb_out_filters, (3, 3), strides=(1, 1), kernel_initializer='he_normal', padding='valid')(y)

        # shortcut
        shortcut = Conv2D(nb_out_filters, (1, 1), strides=subsample, kernel_initializer='he_normal', padding='same')(shortcut)

    return add([shortcut, y])
```

U-NET (Fully connected CNN)

Critical Feautures (Ronneberger, O., et al., 2015):

- **U-NET** can be trained end-to-end from very **few images** and outperforms the prior best methods.
- It consists of a **contracting path (left side)** to capture context and an symmetric **expansive path (right side**) enabling precise localization.
- **Upsampling part (repeating rows and cols)** has a large number of feature channels which allow the network to propagate context information to higher resolution layers.
- **Spatial Dropout:** feature maps dropout.
- **Upside: Small training set.**
- **Downside: Risk of overfitting.**





UNET in Keras

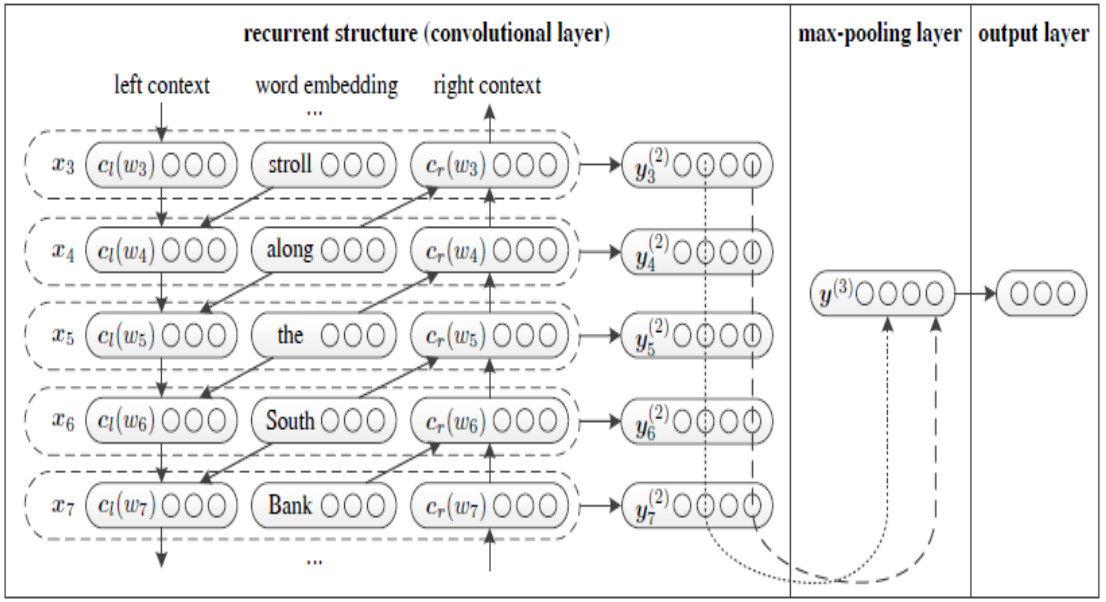
```
def get_unet():
    inputs = Input((1,img_rows, img_cols))
    conv1 = Convolution2D(32, 3, 3, activation='relu', border_mode='same')(inputs)
    conv1 = Convolution2D(32, 3, 3, activation='relu', border_mode='same')(conv1)
    pool1 = MaxPooling2D(pool_size=(2, 2))(conv1)

    conv2 = Convolution2D(64, 3, 3, activation='relu', border_mode='same')(pool1)
    conv2 = Convolution2D(64, 3, 3, activation='relu', border_mode='same')(conv2)
    pool2 = MaxPooling2D(pool_size=(2, 2))(conv2)

    conv3 = Convolution2D(128, 3, 3, activation='relu', border_mode='same')(pool2)
    conv3 = Convolution2D(128, 3, 3, activation='relu', border_mode='same')(conv3)
    pool3 = MaxPooling2D(pool_size=(2, 2))(conv3)

    conv4 = Convolution2D(256, 3, 3, activation='relu', border_mode='same')(pool3)
    conv4 = Convolution2D(256, 3, 3, activation='relu', border_mode='same')(conv4)
    pool4 = MaxPooling2D(pool_size=(2, 2))(conv4)

    conv5 = Convolution2D(512, 3, 3, activation='relu', border_mode='same')(pool4)
    conv5 = Convolution2D(512, 3, 3, activation='relu', border_mode='same')(conv5)

    up6 = merge([UpSampling2D(size=(2, 2))(conv5), conv4], mode='concat', concat_axis=1)
    conv6 = Convolution2D(256, 3, 3, activation='relu', border_mode='same')(up6)
    conv6 = Convolution2D(256, 3, 3, activation='relu', border_mode='same')(conv6)

    up7 = merge([UpSampling2D(size=(2, 2))(conv6), conv3], mode='concat', concat_axis=1)
    conv7 = Convolution2D(128, 3, 3, activation='relu', border_mode='same')(up7)
    conv7 = Convolution2D(128, 3, 3, activation='relu', border_mode='same')(conv7)

    up8 = merge([UpSampling2D(size=(2, 2))(conv7), conv2], mode='concat', concat_axis=1)
    conv8 = Convolution2D(64, 3, 3, activation='relu', border_mode='same')(up8)
    conv8 = Convolution2D(64, 3, 3, activation='relu', border_mode='same')(conv8)

    up9 = merge([UpSampling2D(size=(2, 2))(conv8), conv1], mode='concat', concat_axis=1)
    conv9 = Convolution2D(32, 3, 3, activation='relu', border_mode='same')(up9)
    conv9 = Convolution2D(32, 3, 3, activation='relu', border_mode='same')(conv9)

    conv10 = Convolution2D(1, 1, 1, activation='sigmoid')(conv9)

    model = Model(input=inputs, output=conv10)
```

Recurrent Convolutional Neural Networks (RCNN) (*Lai, S., et al. 2015*)

- They adopt a a recurrent structure to **capture contextual information** as far as possible when learning word representations, which may introduce considerably **less noise compared** to traditional window-based neural networks.

- The **bi-directional recurrent structure** of RCNNs.

- **RCNNs** exhibit a time complexity of O(n)



RCNN Equations

- RCNNs exhibit a **time complexity of O(n)** , which is linearly correlated with the length of the text length.

- **7 equations** defining all the Neural Network topology

- **Input length** can be variable

$$c_l(w_i) = f(W^{(l)} c_l(w_{i-1}) + W^{(sl)} e(w_{i-1})) \quad (1)$$

$$c_r(w_i) = f(W^{(r)} c_r(w_{i+1}) + W^{(sr)} e(w_{i+1})) \quad (2)$$

$$x_i = [c_l(w_i); e(w_i); c_r(w_i)] \quad (3)$$

$$y_i^{(2)} = \tanh\left(W^{(2)} x_i + b^{(2)}\right) \quad (4)$$

$$y^{(3)} = \max_{i=1}^{n} y_i^{(2)} \quad (5)$$

$$y^{(4)} = W^{(4)} y^{(3)} + b^{(4)} \quad (6)$$

$$p_i = \frac{\exp\left(y_i^{(4)}\right)}{\sum_{k=1}^{n} \exp\left(y_k^{(4)}\right)} \quad (7)$$

RCNN in Keras

```
class SentimentModelRecConvNet:
    @staticmethod

    def build(input_length, vector_dim):
        hidden_dim_RNN = 200
        hidden_dim_Dense = 100

        embedding = Input(shape=(input_length, vector_dim))

        left_context = LSTM(hidden_dim_RNN, return_sequences = True)(embedding)                                  # Equation 1
        # left_contex: batch_size x tweet_length x hidden_state_dim
        right_context = LSTM(hidden_dim_RNN, return_sequences = True, go_backwards = True)(embedding)            # Equation 2
        # right_cntext: come left_contex
        together = concatenate([left_context, embedding, right_context], axis = 2)                               # Equation 3
        semantic = TimeDistributed(Dense(hidden_dim_Dense, activation = "tanh"))(together)                       # Equation 4
        pool_rnn = Lambda(lambda x: backend.max(x, axis = 1), output_shape = (hidden_dim_Dense, ))(semantic)      # Equation 5
        pool_rnn_args = Lambda(lambda x: backend.argmax(x, axis=1), output_shape = (hidden_dim_Dense, ))(semantic)

        output = Dense(1, input_dim = hidden_dim_Dense, activation = "sigmoid")(pool_rnn)                        # Equations 6, 7

        deepnetwork = Model(inputs=embedding, outputs=output)
        deepnetwork_keywords = Model(inputs=embedding, outputs=pool_rnn_args)

        return [deepnetwork, deepnetwork_keywords]
```

NLP: A Textual Classifier with Keras – 20NewsGroup Dataset

GloVe word embeddings

We will be using GloVe embeddings, which you can read about here. GloVe stands for "Global Vectors for Word Representation". It's a somewhat popular embedding technique based on factorizing a matrix of word co-occurence statistics.

Specifically, we will use the 100-dimensional GloVe embeddings of 400k words computed on a 2014 dump of English Wikipedia. You can download them here (warning: following this link will start a 822MB download).

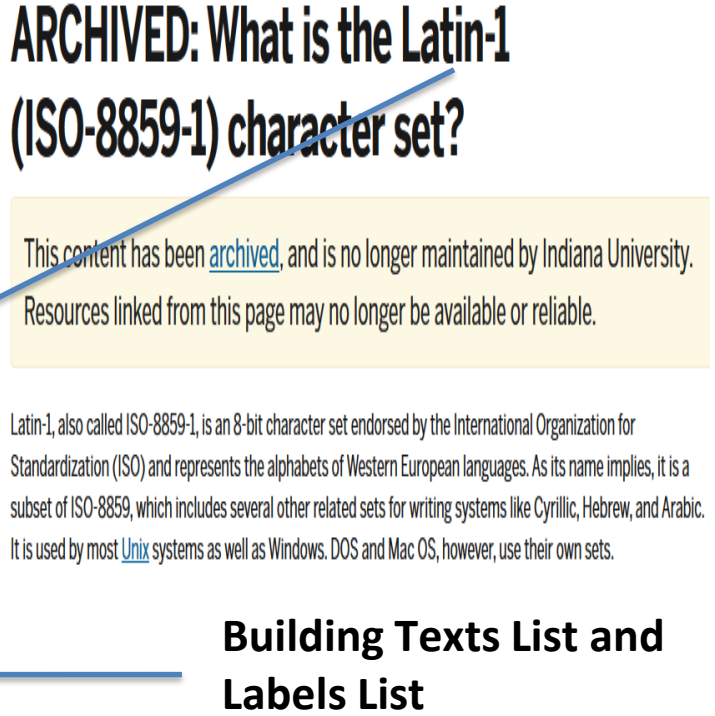
20 Newsgroup dataset

The task we will try to solve will be to classify posts coming from 20 different newsgroup, into their original 20 categories --the infamous "20 Newsgroup dataset". You can read about the dataset and download the raw text data here.

comp.graphics comp.os.ms-windows.misc comp.sys.ibm.pc.hardware comp.sys.mac.hardware comp.windows.x	rec.autos rec.motorcycles rec.sport.baseball rec.sport.hockey	sci.crypt sci.electronics sci.med sci.space
misc.forsale	talk.politics.misc talk.politics.guns talk.politics.mideast	talk.religion.misc alt.atheism soc.religion.christian

NLP Preprocessing: Dataset Loading

```
texts = []  # list of text samples
labels_index = {}  # dictionary mapping label name to numeric id
labels = []  # list of label ids
for name in sorted(os.listdir(TEXT_DATA_DIR)):
    path = os.path.join(TEXT_DATA_DIR, name)
    if os.path.isdir(path):
        label_id = len(labels_index)
        labels_index[name] = label_id
        for fname in sorted(os.listdir(path)):
            if fname.isdigit():
                fpath = os.path.join(path, fname)
                if sys.version_info < (3,):
                    f = open(fpath)
                else:
                    f = open(fpath, encoding='latin-1')
                t = f.read()
                i = t.find('\n\n')  # skip header
                if 0 < i:
                    t = t[i:]
                texts.append(t)
                f.close()
                labels.append(label_id)

print('Found %s texts.' % len(texts))
```



ARCHIVED: What is the Latin-1 (ISO-8859-1) character set?

This content has been archived, and is no longer maintained by Indiana University. Resources linked from this page may no longer be available or reliable.

Latin-1, also called ISO-8859-1, is an 8-bit character set endorsed by the International Organization for Standardization (ISO) and represents the alphabets of Western European languages. As its name implies, it is a subset of ISO-8859, which includes several other related sets for writing systems like Cyrillic, Hebrew, and Arabic. It is used by most Unix systems as well as Windows. DOS and Mac OS, however, use their own sets.

Building Texts List and Labels List

NLP Preprocessing: Dataset Loading

```
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences

tokenizer = Tokenizer(nb_words=MAX_NB_WORDS)
tokenizer.fit_on_texts(texts)
sequences = tokenizer.texts_to_sequences(texts)
```

Stop-words Cleaning and tokenization

```
word_index = tokenizer.word_index
print('Found %s unique tokens.' % len(word_index))

data = pad_sequences(sequences, maxlen=MAX_SEQUENCE_LENGTH)
```

Building of the Word Index

Sequence 1D 0-Padding

```
labels = to_categorical(np.asarray(labels))
print('Shape of data tensor:', data.shape)
print('Shape of label tensor:', labels.shape)

# split the data into a training set and a validation set
indices = np.arange(data.shape[0])
np.random.shuffle(indices)
data = data[indices]
labels = labels[indices]
nb_validation_samples = int(VALIDATION_SPLIT * data.shape[0])

x_train = data[:-nb_validation_samples]
y_train = labels[:-nb_validation_samples]
x_val = data[-nb_validation_samples:]
y_val = labels[-nb_validation_samples:]
```

To categorical variables conversion

Shuffle of Data and Labels

Data and Labels Splitting

NLP Preprocessing: Embedding Layer Setting-Up

```
embeddings_index = {}
f = open(os.path.join(GLOVE_DIR, 'glove.6B.100d.txt'))
for line in f:
    values = line.split()
    word = values[0]
    coefs = np.asarray(values[1:], dtype='float32')
    embeddings_index[word] = coefs
f.close()

print('Found %s word vectors.' % len(embeddings_index))
```

Embedding vectors loading and check

```
embedding_matrix = np.zeros((len(word_index) + 1, EMBEDDING_DIM))
for word, i in word_index.items():
    embedding_vector = embeddings_index.get(word)
    if embedding_vector is not None:
        # words not found in embedding index will be all-zeros.
        embedding_matrix[i] = embedding_vector
```

Embedding matrix arrangement

```
from keras.layers import Embedding

embedding_layer = Embedding(len(word_index) + 1,
                            EMBEDDING_DIM,
                            weights=[embedding_matrix],
                            input_length=MAX_SEQUENCE_LENGTH,
                            trainable=False)
```

Keras Embedding Layer primitives

NLP: Cov1D Model

Training a 1D convnet

Finally we can then build a small 1D convnet to solve our classification problem:

```
sequence_input = Input(shape=(MAX_SEQUENCE_LENGTH,), dtype='int32')
embedded_sequences = embedding_layer(sequence_input)
x = Conv1D(128, 5, activation='relu')(embedded_sequences)
x = MaxPooling1D(5)(x)
x = Conv1D(128, 5, activation='relu')(x)
x = MaxPooling1D(5)(x)
x = Conv1D(128, 5, activation='relu')(x)
x = MaxPooling1D(35)(x)  # global max pooling
x = Flatten()(x)
x = Dense(128, activation='relu')(x)
preds = Dense(len(labels_index), activation='softmax')(x)

model = Model(sequence_input, preds)
model.compile(loss='categorical_crossentropy',
              optimizer='rmsprop',
              metrics=['acc'])

# happy learning!
model.fit(x_train, y_train, validation_data=(x_val, y_val),
          epochs=2, batch_size=128)
```

This model reaches **95% classification accuracy** on the validation set after only 2 epochs. You could probably get to an even higher accuracy by training longer with some regularization mechanism (such as dropout) or by fine-tuning the `Embedding` layer.

We can also test how well we would have performed by not using pre-trained word embeddings, but instead initializing our `Embedding` layer from scratch and learning its weights during training. We just need to replace our `Embedding` layer with the following:

```
embedding_layer = Embedding(len(word_index) + 1,
                            EMBEDDING_DIM,
                            input_length=MAX_SEQUENCE_LENGTH)
```

After 2 epochs, this approach only gets us to **90% validation accuracy**, less than what the previous model could reach in just one epoch. Our pre-trained embeddings were definitely buying us something. In general, using pre-trained embeddings is relevant for natural processing tasks were little training data is available (functionally the embeddings act as an injection of outside information which might prove useful for your model).

REFERENCES

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems* (pp. 1097-1105).

Simonyan, K., & Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*.

Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., ... & Rabinovich, A. (2015). Going deeper with convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 1-9).

He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 770-778).

AKNOWLEDGEMENTS

THANK YOU

FOR YOUR ATTENTION

Francesco Pugliese