INTRODUCTION TO KERAS

High-level approach to Deep Learning
What is Keras?

Keras is a high-level neural networks API, capable of running on top of Tensorflow, Theano, and CNTK.

It enables fast experimentation through a high level, user-friendly, modular and extensible API.

Keras can also be run on both CPU and GPU.
Install Keras

Activate your conda env.
Install tensorflow (or tensorflow-gpu).

Then run:

conda install keras

or (highly recommended):

conda install keras-gpu
**TOY PROBLEM DATASETS**

Keras offers 7 datasets to test your networks such as mnist (as tensorflow does).

An example is:

```python
from keras.datasets import mnist

(x_train, y_train), (x_test, y_test) = mnist.load_data()
```
Prepare MNIST data

The required command are similar to the ones for tensorflow:

```python
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255
x_test /= 255
x_train = X_train.reshape(X_train.shape[0], 28, 28, 1)
x_test = X_test.reshape(X_test.shape[0], 28, 28, 1)
```
ENCODING: DEFINITION (1/2)

Encoding is the process of converting data into a format required for a number of information processing needs, including:

- Program compiling and execution
- Data transmission, storage and compression/decompression
- Application data processing, such as file conversion
Encoding can have two meanings:

- In **computer technology**, encoding is the process of applying a specific code, such as letters, symbols and numbers, to data for conversion into an equivalent cipher.
- In **electronics**, encoding refers to analog to digital conversion.
One-hot encoding

One hot encoding is a process by which categorical variables are converted into a form that could be provided to ML algorithms to do a better job in prediction.

This technique is performed in two steps.
Step 1: Integer Encoding

This is called a label encoding or an integer encoding and is easily reversible.

It consists of the assignment of each unique category value to an integer value.

E.g. “red” is 1, “green” is 2, and “blue” is 3.

NB: The integer values have a natural ordered relationship between each other and machine learning algorithms may be able to understand and harness this relationship.
**Step 2: One Hot Encoding (Why?)**

For categorical variables where no such ordinal relationship exists, the integer encoding is not enough.

In fact, using this encoding and allowing the model to assume a natural ordering between categories may result in poor performance or unexpected results (predictions halfway between categories).
STEP 2: ONE HOT ENCODING

In this case, a one-hot encoding can be applied to the integer representation. This is where the integer encoded variable is removed and a new binary variable is added for each unique integer value.

In the “color” variable example, there are 3 categories and therefore 3 binary variables are needed. A “1” value is placed in the binary variable for the color and “0” values for the other colors.
Encoding summary

Categorical data must often be encoded when working with machine learning algorithms. In particular, we notice that:

- That categorical data is defined as variables with a finite set of label values.

- That most machine learning algorithms require numerical input and output variables.

- That an integer and one hot encoding is used to convert categorical data to integer data.
We will also transform our labels into a one-hot encoding using the `to_categorical` method from Keras.

```python
from keras.utils import to_categorical

y_train = to_categorical(y_train, 10)
y_test = to_categorical(y_test, 10)
```
TWO POSSIBLE API(s) TO CREATE KERAS MODELS

Keras offers two APIs to create your networks:

- Sequential API
  (Easier but more constrained)

- Functional API
  (close to tensorflow)
The easiest way of creating a model in Keras is by using the sequential API, which lets you stack one layer after the other. The problem with the sequential API is that it doesn’t allow models to have multiple inputs or outputs, which are needed for some problems.
from keras.models import Sequential
from keras.layers import Conv2D, MaxPool2D, Dense, Flatten, Dropout

model = Sequential()
model.add(Conv2D(filters=32, kernel_size=(5, 5), activation='relu', input_shape=x_train.shape[1:]))
model.add(Conv2D(filters=32, kernel_size=(5, 5), activation='relu'))
model.add(MaxPool2D(pool_size=(2, 2)))
model.add(Dropout(rate=0.25))
model.add(Conv2D(filters=64, kernel_size=(3, 3), activation='relu'))
model.add(Conv2D(filters=64, kernel_size=(3, 3), activation='relu'))
model.add(MaxPool2D(pool_size=(2, 2)))
model.add(Dropout(rate=0.25))
model.add(Flatten())
model.add(Dense(256, activation='relu'))
model.add(Dropout(rate=0.5))
model.add(Dense(10, activation='softmax'))
from keras.models import Sequential
from keras.layers import Conv2D, MaxPool2D, Dense, Flatten, Dropout

model = Sequential([          
    Conv2D(filters=32, kernel_size=(5, 5), activation='relu', input_shape=x_train.shape[1:]),
    Conv2D(filters=32, kernel_size=(5, 5), activation='relu'),
    MaxPool2D(pool_size=(2, 2)),
    Dropout(rate=0.25),
    Conv2D(filters=64, kernel_size=(3, 3), activation='relu'),
    Conv2D(filters=64, kernel_size=(3, 3), activation='relu'),
    MaxPool2D(pool_size=(2, 2)),
    Dropout(rate=0.25),
    Flatten(),
    Dense(256, activation='relu'),
    Dropout(rate=0.5),
    Dense(10, activation='softmax')
])
**Functional API**

Alternatively, the functional API allows you to create the same models but offers you more flexibility at the cost of simplicity and readability.

It can be used with multiple input and output layers as well as shared layers, which enables you to build really complex network structures.
from keras.models import Model
from keras.layers import Conv2D, MaxPool2D, Dense, Flatten, Dropout, Input

inputs = Input(shape=x_train.shape[1:])

x = Conv2D(filters=32, kernel_size=(5, 5), activation='relu')(inputs)
x = Conv2D(filters=32, kernel_size=(5, 5), activation='relu')(x)
x = MaxPool2D(pool_size=(2, 2))(x)
x = Dropout(rate=0.25)(x)
x = Conv2D(filters=64, kernel_size=(3, 3), activation='relu')(x)
x = Conv2D(filters=64, kernel_size=(3, 3), activation='relu')(x)
x = MaxPool2D(pool_size=(2, 2))(x)
x = Dropout(rate=0.25)(x)
x = Flatten()(x)
x = Dense(256, activation='relu')(x)
x = Dropout(rate=0.5)(x)
predictions = Dense(10, activation='softmax')(x)

model = Model(inputs=inputs, outputs=predictions)
model.compile(
    loss='categorical_crossentropy',
    optimizer='adam',
    metrics=['accuracy']
)
from keras.preprocessing.image import ImageDataGenerator

datagen = ImageDataGenerator(
    rotation_range=10,
    zoom_range=0.1,
    width_shift_range=0.1,
    height_shift_range=0.1
)
**Fit a Model**

epochs = 3

batch_size = 32

history = model.fit_generator(
    datagen.flow(x_train, y_train, batch_size=batch_size),
    epochs=epochs, validation_data=(x_test, y_test),
    steps_per_epoch=x_train.shape[0] // batch_size
)
import matplotlib.pyplot as plt
plt.plot(history.history['acc'], label='training accuracy')
plt.plot(history.history['val_acc'], label='testing accuracy')
plt.title('Accuracy')
plt.xlabel('epochs')
plt.ylabel('accuracy')
plt.legend()
plt.plot(history.history['loss'], label='training loss')
plt.plot(history.history['val_loss'], label='testing loss')
plt.title('Loss')
plt.xlabel('epochs')
plt.ylabel('loss')
plt.legend()
Why do we still prefer TensorFlow?

- Control over the dataflow!!!
- Greater compatibility