Contents

In order to use this file, activate org-babel for ipython and press C-c C-c to execute code blocks.

First we import number and set its random seed to a fixed number for reproducibility.

```
import numpy as np
# For reproducibility
np.random.seed(123)

from keras import backend as K
import os

def set_keras_backend(backend):

    if K.backend() != backend:
        os.environ['KERAS_BACKEND'] = backend
        import importlib
        importlib.reload(K)
        assert K.backend() == backend

set_keras_backend("theano")
```

Plotting the first image in the training data so that we have an idea of what we're looking at.

```
%matplotlib inline
# Visualize data
from matplotlib import pyplot as plt
# plt.imshow(X_train[0])
```

```
net = NeuralNet(input_node_size = 784,
                output_node_size = 10,
                hidden_layers_node_size = [512])
```

```
# net.train(X_train, Y_train, epochs=6)
```

1 NeuralNet

```
class NeuralNet(object):

    def __init__(self,
                 input_node_size = None,              # Number of nodes in input layer
                 output_node_size = None,             # Number of nodes in output laye
                 input_shape = None,
                 hidden_layers_node_size = []         # Number of nodes in each hidden
                ):
                    from keras.models import Sequential
                    self.model = Sequential()
                    from keras.layers import Dense, Dropout, Activation, Flatten, LSTM
                    # First layer requires input dimension ie input_shape
                    self.model.add(
                                   LSTM(units=64,
                                         input_dim=input_node_size
                                         )
                                   )
                    self.model.add(Activation('relu'))
                    # Add layers to model for all hidden layers
                    for node_size in hidden_layers_node_size:
                        self.model.add(
                                       Dense(units=node_size)
                                      )
                        self.model.add(Activation('relu'))
                        self.model.add(Dropout(0.3))
                    #          from keras import regularizers
                    #          self.model.add(Dense(64,
                    #                          input_dim=64,
                    #                          kernel_regularizer=regularizers.l2(0.01)
                    #                          activity_regularizer=regularizers.l1(0.0
                    #                  )
                    # Last layer requires activation to be softmax
                    self.model.add(
                                   Dense(units=output_node_size,
                                         activation='softmax'
                                         )
                                  )
                    # Compile model
```

```
                self.model.compile(loss='categorical_crossentropy',
                                   optimizer='adam',
                                   metrics=['accuracy'])
                #model.fit(x_train, y_train, epochs=5, batch_size=32)
    def train(self, train_x, train_y, epochs):
        self.model.fit(train_x, train_y, epochs, batch_size = 32)
    def run(self, X, Y, steps):
        metrics = []
        metrics = self.model.evaluate(X, Y, batch_size = 32, steps = steps)
        return metrics
    def label(self, X, steps):
        predictions = self.model.predict(X, batch_size = 32, steps = steps)
        return predictions
```

1.1 init

The Sequential model is a linear stack of layers. We pass in a list of layer instances to it to make a Neural Net.

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

Let's import the core layers from Keras which are almost always used.

```
from keras.layers import Dense, Dropout, Activation, Flatten, LSTM
```

The model should know what input shape it should expect. For this reason, we sepcifiy an input size for the first layer.

```
# First layer requires input dimension ie input_shape
self.model.add(
               LSTM(units=64,
                     input_dim=input_node_size
                     )
               )
self.model.add(Activation('relu'))

# Add layers to model for all hidden layers
for node_size in hidden_layers_node_size:
    self.model.add(
                   Dense(units=node_size)
                  )
    self.model.add(Activation('relu'))
    self.model.add(Dropout(0.3))
```

Adding a regularizer does not improve the model

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

# Last layer requires activation to be softmax
self.model.add(
               Dense(units=output_node_size,
                     activation='softmax'
                     )
              )

# Compile model
self.model.compile(loss='categorical_crossentropy',
                   optimizer='adam',
                   metrics=['accuracy'])
#model.fit(x_train, y_train, epochs=5, batch_size=32)
```

1.2 train

fit the model with training datasets

inputs: $train_x$ - training data $train_y$ - training labels epochs - number of iterations over the entirity of both the x and y data desired

returns: Nothing

```
def train(self, train_x, train_y, epochs):
    self.model.fit(train_x, train_y, epochs, batch_size = 32)
```

1.3 run

evaluates the model with test data

inputs: X - test data Y - test labels steps - number of iterations over the entire dataset before evaluation is completed

returns: metrics - the test losses as well as the metric defined in init, which in this case is accuracy

```
def run(self, X, Y, steps):
    metrics = []
    metrics = self.model.evaluate(X, Y, batch_size = 32, steps = steps)
    return metrics
```

1.4 label

predicts the labels of the data given

Inputs: X - unlabeled test data steps - number of iterations over the entire dataset before evaluation is completed

returns: predictions - a numpy array of predictions

```
def label(self, X, steps):
    predictions = self.model.predict(X, batch_size = 32, steps = steps)
    return predictions
```