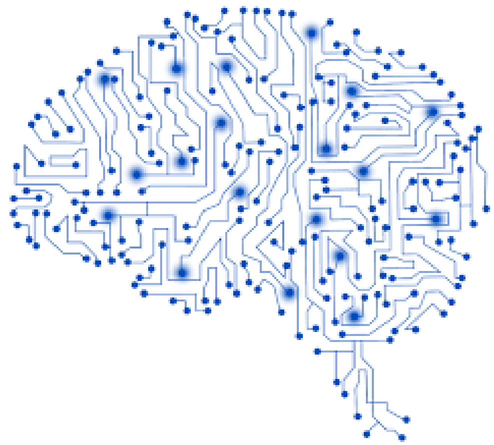
University of Minho
School of Engineering

ISLab



LSTM networks with TensorFlow™

Connective Systems and Classifiers

Perfil ML:FA @ MiEI/4º ano - 2º Semestre

Bruno Fernandes, Victor Alves

12/03/2020

Contents

LSTM Networks

MLPs are a good starting point for modeling sequence prediction problems … But we now have better options!

The Long Short-Term Memory, or LSTM, network is a type of Recurrent Neural Network, or RNN, specially designed for sequence problems. The promise of RNNs, and LSTMs in particular, is that the temporal dependence and contextual information in the input data can be learned!

Q.: When were LSTMs introduced?
A.: In 1997, in the work of Hochreiter & Schmidhuber, but many more contributed to the modern LSTM. Check the original paper here!

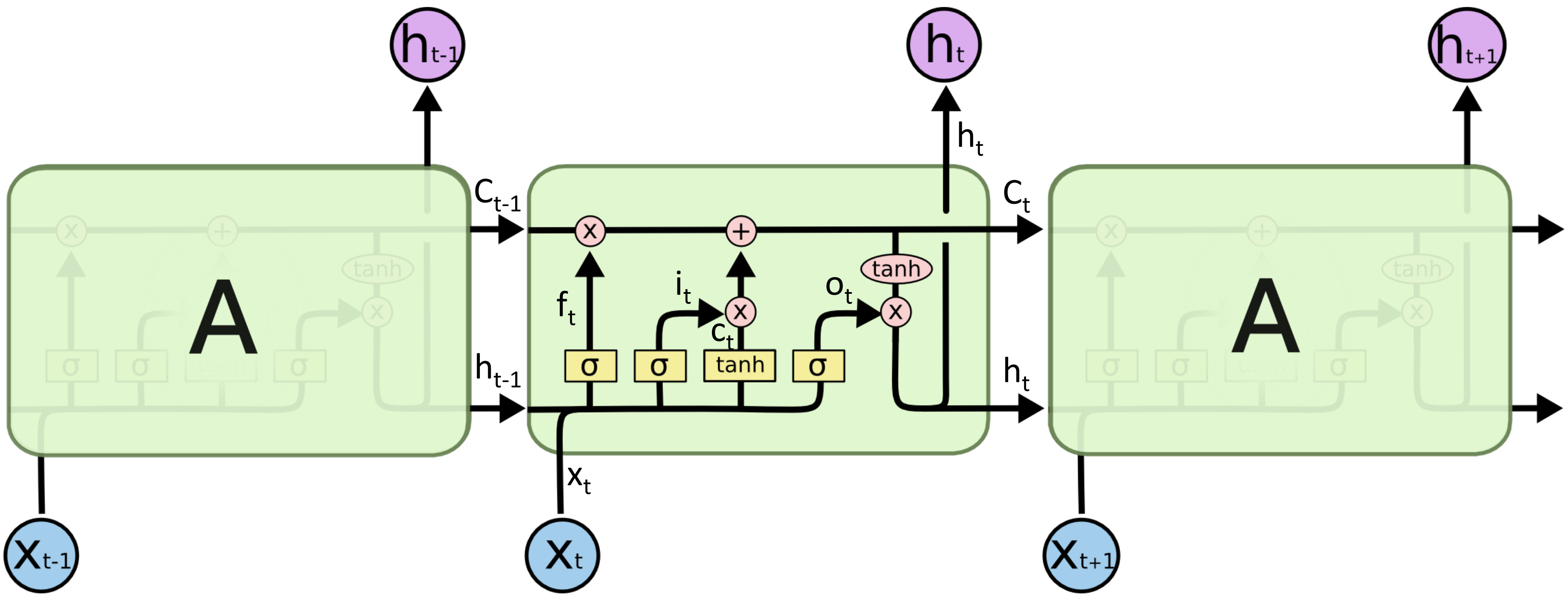
LSTM Networks

- The computational unit of the LSTM network is called the memory cell (memory block, neuron or just cell are also used)!
- LSTM neurons are comprised of weights and gates
- The key to the LSTM neurons (memory cells) are the gates. These too are weighted functions that further govern the information flow (internal state) in the cell.

LSTM Networks

- A neuron has three gates:
  - Forget Gate that decides what information to discard from the internal state
  - Input Gate that decides which values from the input to add to the internal state
  - Output Gate that decides what to output based on the input and internal state
- The forget and input gates are used to update the internal state of the neuron
- The output gate is a final limiter on what the cell actually outputs.
- It is these gates and the consistent data flow that keep each cell stable (avoiding the vanishing and exploding gradients problem)

A memory cell



(https://colah.github.io/posts/2015-08-Understanding-LSTMs/)

Application of LSTMs

Automatic Image Caption Generation

(Show and Tell: A Neural Image Caption Generator, 2014)

Application of LSTMs

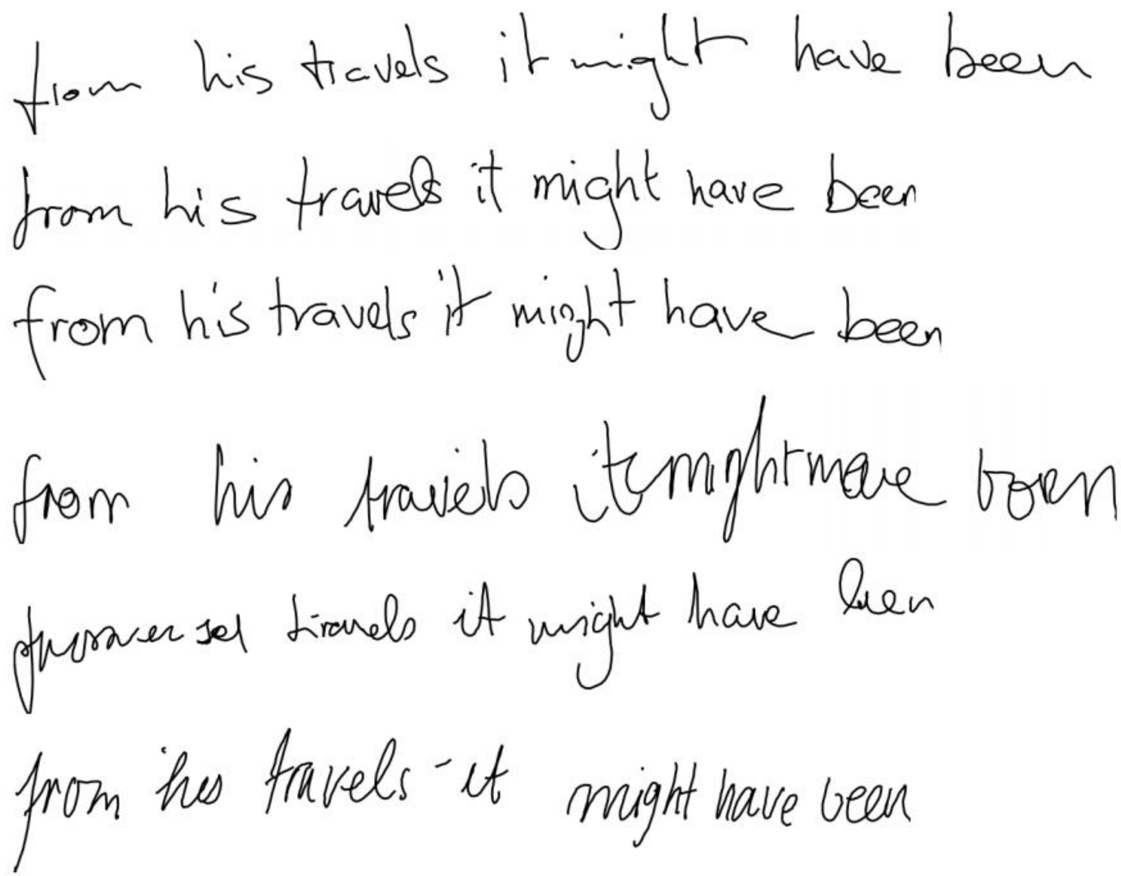
Automatic Translation of Text

Type	Sentence
Our model	Ulrich UNK , membre du conseil d' administration du constructeur automobile Audi , affirme qu' il s' agit d' une pratique courante depuis des années pour que les téléphones portables puissent être collectés avant les réunions du conseil d' administration afin qu' ils ne soient pas utilisés comme appareils d' écoute à distance .
Truth	Ulrich Hackenberg , membre du conseil d' administration du constructeur automobile Audi , déclare que la collecte des téléphones portables avant les réunions du conseil , afin qu' ils ne puissent pas être utilisés comme appareils d' écoute à distance , est une pratique courante depuis des années .
Our model	" Les téléphones cellulaires , qui sont vraiment une question , non seulement parce qu' ils pourraient potentiellement causer des interférences avec les appareils de navigation , mais nous savons , selon la FCC , qu' ils pourraient interférer avec les tours de téléphone cellulaire lorsqu' ils sont dans l' air " , dit UNK .
Truth	" Les téléphones portables sont véritablement un problème , non seulement parce qu' ils pourraient éventuellement créer des interférences avec les instruments de navigation , mais parce que nous savons , d' après la FCC , qu' ils pourraient perturber les antennes-relais de téléphonie mobile s' ils sont utilisés à bord " , a déclaré Rosenker .
Our model	Avec la crémation , il y a un " sentiment de violence contre le corps d' un être cher " , qui sera " réduit à une pile de cendres " en très peu de temps au lieu d' un processus de décomposition " qui accompagnera les étapes du deuil " .
Truth	Il y a , avec la crémation , " une violence faite au corps aimé " , qui va être " réduit à un tas de cendres " en très peu de temps , et non après un processus de décomposition , qui " accompagnerait les phases du deuil " .

(Sequence to Sequence Learning with Neural Networks, 2014)

Application of LSTMs

Automatic Handwriting Generation



(Generating Sequences With Recurrent Neural Networks, 2014)

Sequence Prediction

- Sequence prediction is different to other types of supervised learning problems

- The sequence imposes an explicit order on the observations that must be preserved when training models and making predictions

- Order is important and it must be respected in the formulation of prediction problems that use the sequence data as input!

Sequence Prediction

[1, 2, 3, 4, 5, 6] → Sequence Prediction Model → [6]

[1, 2, 3, 4, 5, 6] → Sequence Prediction Model → [‘good’]

[[1, 2, 3], [4, 5, 6]] → Sequence Prediction Model → [7, 8, 9]

[1, 2, 3, 4, 5, 6] → Sequence Prediction Model → [7, 8, 9, 10, 11]

Sequence Prediction

Single-step vs Multi-step

- Single-step consists in forecasting just the next value of the sequence (next-element prediction)
  - Given the observed temperature over the last 7 days forecast the temperature at timestep 8
- Multi-step consists in forecasting multiple timesteps beyond the input sequences (sequence-to-sequence prediction)
  - Given the observed temperature over the last 7 days forecast the temperature at timestep 8, 9 and 10
  - Accurate straightforward multi-step forecasting remains a challenge to our days

Sequence Prediction

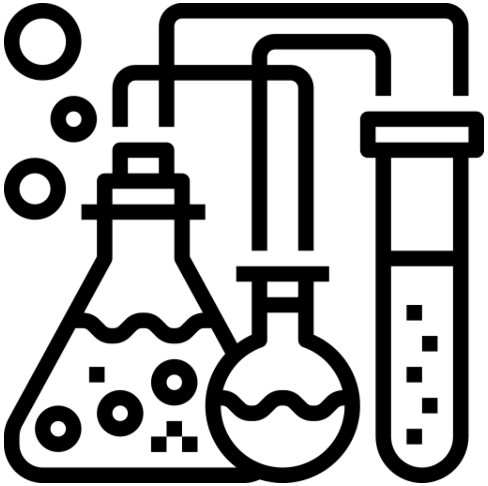
Univariate vs Multivariate

- Univariate, or single input, is where you use a single feature as input over multiple input timesteps
    - Given the observed pollution in previous timesteps predict the expected pollution value

- Multivariate, or multiple input, is where you use multiple features as input over multiple input timesteps
    - Given the observed pollution and weather in previous timesteps predict the expected pollution value

Preparing Data for LSTMs

1. Preparing Numeric Data
2. Preparing Categorical Data
3. Preparing Sequences with Varied Lengths
4. Preparing Sequence Prediction as Supervised Learning



Preparing Numeric Data

When a network is fit on unscaled data, it is possible for large inputs to slow down the learning and convergence of the network! It may prevent the network from learning our problem!

There are two types of scaling that we may want to consider (use scikit-learn):

- Normalization
  - rescaling data so that all values fall within the range of 0 and 1
- Standardization
  - rescaling the distribution of values so that the mean of observed values is 0 and the standard deviation is 1
  - assumes observations fit a Gaussian distribution with a well behaved mean and standard deviation - which may not always be the case

Normalization

Normalization can be achieved using the scikit-learn object MinMaxScaler

```
from pandas import Series
from sklearn.preprocessing import MinMaxScaler

#define series
series = Series([0.0, 10.0, 20.0, 30.0, 40.0, 50.0, 60.0, 70.0, 80.0,
90.0, 100.0])

#prepare data for normalization
values = series.values.reshape((len(series.values), 1))

#fit and transform or fit_transform
scaler = MinMaxScaler(feature_range=(0, 1))
normalized = scaler.fit_transform(values)
print(normalized)

#inverse transform is also a thing
inversed = scaler.inverse_transform(normalized)
print(inversed)
```

»

```
[[0. ]
 [0.1]
 [0.2]
 ...
 [0.9]
 [1. ]]

[[  0.]
 [ 10.]
 [ 20.]
...
 [ 90.]
 [100.]]
```

Standardization

Standardization can be achieved using the scikit-learn object StandardScaler

```
from pandas import Series
from sklearn.preprocessing import StandardScaler

#define series
series = Series([4.1, 1.0, 2.6, 7.9, 5.5, 9.0, 8.8, 3.0, 6.3])

#prepare data for standardization
values = series.values.reshape((len(series.values), 1))

#fit and transform or fit_transform
scaler = StandardScaler()
standardized = scaler.fit_transform(values)
print(standardized)

#inverse transform is still a thing
inversed = scaler.inverse_transform(standardized)
print(inversed)
```

»

```
[[-0.46286604]
 [-1.60569456]
 [-1.01584758]
 ...
 [-0.86838584]
 [ 0.34817357]]

[[4.1]
 [1. ]
 [2.6]
 ...
 [3. ]
 [6.3]]
```

Preparing Categorical Data

Categorical data, often called nominal data, are variables that contain label values rather than numeric values

There are two types of treatment we may want to consider:

- Label (or integer) Encoding
  - integer values have a natural ordered relationship between each other and ML algorithms may be able to understand and harness this relationship
- One Hot Encoding
  - categorical variables where no such ordinal relationship exists

Preparing Categorical Data

It can be achieved using the scikit-learn objects LabelEncoder and OneHotEncoder

```
from numpy import array, argmax
from sklearn.preprocessing import LabelEncoder, OneHotEncoder

#define array
values = array(['cold', 'warm', 'cold', 'hot', 'warm', 'warm', 'hot'])

#label encoding
label_encoder = LabelEncoder()
label_encoded = label_encoder.fit_transform(values)
print(label_encoded)

#one hot encoding
onehot_encoder = OneHotEncoder(sparse=False, categories='auto')
label_encoded = label_encoded.reshape(len(label_encoded), 1)
onehot = onehot_encoder.fit_transform(label_encoded)
print(onehot)

#inverse transform keeps being a thing
inverted = label_encoder.inverse_transform([argmax(onehot[0, :])])
print(inverted)
```

```
[0 2 0 1 2 2 1]

[[1. 0. 0.]
 [0. 0. 1.]
 [1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]
 [0. 0. 1.]
 [0. 1. 0.]]

['cold']
```

Sequence with Varied Lengths

Deep learning libraries assume a vectorized representation of data. In the case of variable length sequence prediction problems, sequences should have the same length

There are some types of treatment we may want to consider:

- Padding (adding a number, usually 0, to the beginning or the end of a sequence)
  - Pre-Sequence Padding
  - Post-Sequence Padding
- Truncation (trim a sequence to a desired length from its beginning or end)
  - Pre-Sequence Truncation
  - Post-Sequence Truncation

Padding Sequences

Use tf.keras pad_sequences object

```
from tensorflow.keras.preprocessing.sequence import pad_sequences

#define sequences
sequences = [
[1, 2, 3, 4],
[1, 2, 3],
[1]
]

#pre-sequence padding
pre_padded = pad_sequences(sequences)
print(pre_padded)

#post-sequence padding
post_padded = pad_sequences(sequences, padding='post')
print(post_padded)
```

```
[[1 2 3 4]
 [0 1 2 3]
 [0 0 0 1]]

[[1 2 3 4]
 [1 2 3 0]
 [1 0 0 0]]
```

Truncating Sequences

Use tf.keras pad_sequences object and define the desired length on the maxlen argument

```
from tensorflow.keras.preprocessing.sequence import pad_sequences

#define sequences
seq = [
[1, 2, 3, 4],
[1, 2, 3],
[1]
]

#pre-sequence truncation
pre_truncated = pad_sequences(seq, maxlen=2)
print(pre_truncated)

#post-sequence truncation
post_truncated = pad_sequences(seq, maxlen=2, truncating='post')
print(post_truncated)
```

```
[[3 4]
 [2 3]
 [0 1]]

[[1 2]
 [1 2]
 [0 1]]
```

Sequence Prediction
as Supervised Learning

Sequence prediction problems must be re-framed as Supervised Learning problems!

Data must be transformed from a sequence to pairs of input/output. As we already know, a supervised learning problem allows models to learn how to predict output patterns from input data.

Pandas shift() function may help transform time series data into a supervised learning problem! Given a DataFrame, the shift() function can be used to create copies of columns that are pushed forward (rows of NaN values added to the front) or pulled back (rows of NaN values added to the end)

1
2
3
4
5
6
7
8
9

X, y
1, 2
2, 3
3, 4
4, 5
5, 6
6, 7
7, 8
8, 9

Sequence Prediction
as Supervised Learning


```
from pandas import DataFrame


#define a mock timeseries as a column and as a sequence of 10 nrs
df = DataFrame()
df['t'] = [x for x in range(10)]
print(df)
```

»

```
   t
0  0
1  1
2  2
 ...
9  9
```

Sequence Prediction

as Supervised Learning


```
from pandas import DataFrame


#define a mock timeseries as a column and as a sequence of 10 nrs
df = DataFrame()
df['t'] = [x for x in range(10)]
print(df)


#shift all obs down by 1 timestep by inserting a new row at the top
#the new row has no data so it will be represented as NaN
df['t-1'] = df['t'].shift(1)
print(df)
```

```
   t
0  0
1  1
2  2
 ...
9  9
-----------------
   t  t-1
0  0  NaN
1  1  0.0
2  2  1.0
  ...
9  9  8.0
```

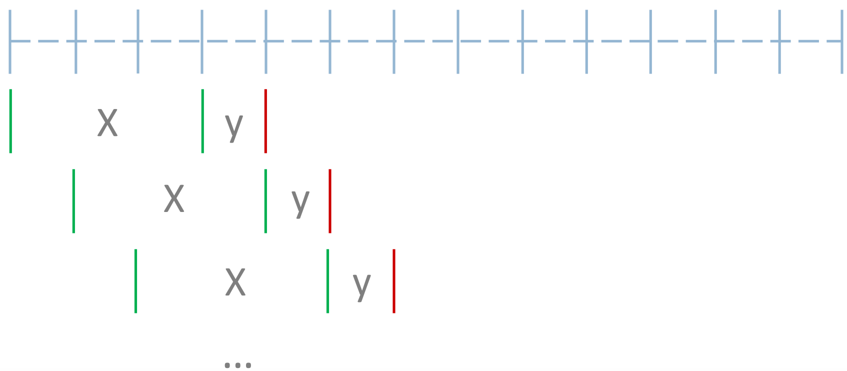
Sequence Prediction

as Supervised Learning


```
from pandas import DataFrame


#define a mock timeseries as a column and as a sequence of 10 nrs
df = DataFrame()
df['t'] = [x for x in range(10)]
print(df)


#shift all obs down by 1 timestep by inserting a new row at the top
#the new row has no data so it will be represented as NaN
df['t-1'] = df['t'].shift(1)
print(df)


#shift backward using a negative int value
#this pulls the observations up by inserting a new row at the end
df['t+1'] = df['t'].shift(-1)
print(df)
```

```
   t
0  0
1  1
2  2
 ...
9  9
-----------------
   t  t-1
0  0  NaN
1  1  0.0
2  2  1.0
  ...
9  9  8.0
-----------------
   t  t-1  t+1
0  0  NaN  1.0
1  1  0.0  2.0
2  2  1.0  3.0
...
9  9  8.0  NaN
```

Sequence Prediction
as Supervised Learning



Algorithm 2: From an unsupervised to a supervised problem.

Input: dataset, timesteps, multi_steps

Output: input data, label data

X, y = list(), list()

while $i \in range(len(dataset)$ - $(timesteps + multi\_steps))$ **do**

  input_index = i + timesteps

  label_index = input_index + multi_steps

  X.append(dataset[i:input_index, :])

  y.append(dataset[input_index:label_index, speed_diff])

end

return X, y

Sequence Prediction

Missing Timesteps

Missing timesteps may be problematic for RNNs! There are several ways of handling those:

- Masking with default values
- Interpolation
- Model-based computation

Sequence Prediction

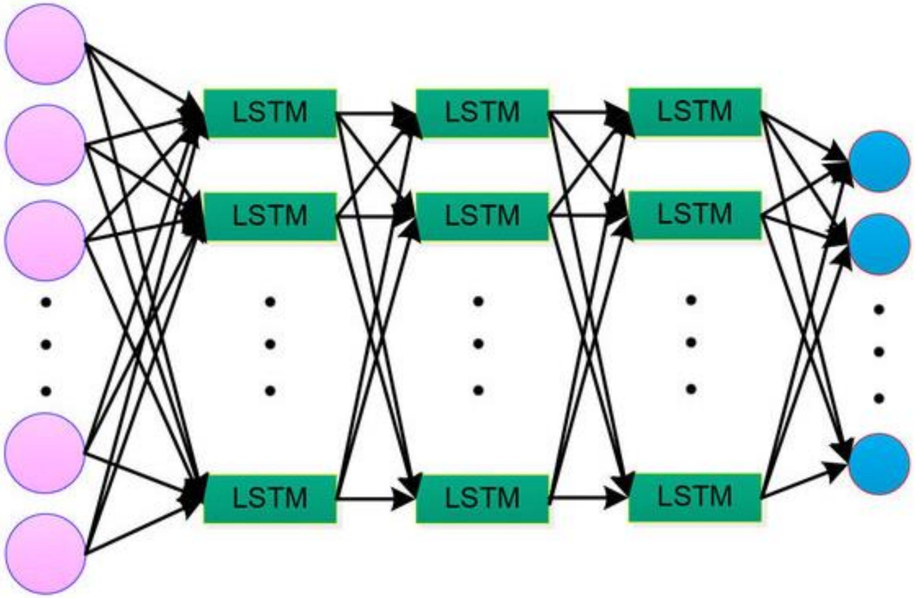
Missing Timesteps

Algorithm 1: Computation of missing timesteps.

```
dataset.resample('20min')
dataset['temperature', 'precipitation'].interpolate(method ='linear',
 limit_direction ='forward', limit='6h')
initialize consequent_missing_obs
foreach i, row ∈ enumerate(dataset, 0) do
    if row['speed_diff'] is NaN and consequent_missing_obs <10h then
        increment consequent_missing_obs
        if i - (3 × one_week) >0 then
            value_1week_before = dataset[i - one_week]
            value_2weeks_before = dataset[i - one_week × 2]
            value_3weeks_before = dataset[i - one_week × 3]
            row['speed_diff'] = mean(value_1week_before,
             value_2weeks_before, value_3weeks_before)
        else
            value_1week_after = dataset[i + one_week]
            value_2weeks_after = dataset[i + one_week × 2]
            value_3weeks_after = dataset[i + one_week × 3]
            row['speed_diff'] = mean(value_1week_after, value_2weeks_after,
             value_3weeks_after)
        end
    else if row['speed_diff'] is not NaN then
        reset consequent_missing_obs
end
```

Developing LSTM networks

1. The Shape of Inputs
2. Shuffling Samples (Not)
3. The Importance of Batches in LSTMs
4. LSTM's State Management
5. The echo prediction problem



The Shape of Inputs

The first step is to define the model! It is in the first LSTM layer where we define the shape of the inputs the model is expecting! Input must be a 3D vector comprising:

- Samples
  - One sequence is a sample! A batch is comprised of one, or more, samples
- Timesteps
  - Past observations of a feature. One timestep is one point of observation in the sample.
- Features
  - These are columns in your data. One feature is one observation at a timestep.

Use numpy reshape() function!

```
from numpy import array

data = array([0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0])
print(data.shape)
data = data.reshape((1, 10, 1))
print(data.shape)
```

```
(10,)

(1, 10, 1)
```

The Shape of Inputs

The first LSTM layer in the network must define the shape of the inputs it is expecting! We need to define a tuple containing the **number of timesteps** and the **number of features**!

For example, if we had 10 timesteps and 1 feature (univariate sequence) per sample, the model would be defined as:

```
#an LSTM layer with 5 memory cells/neurons
model = Sequential()
model.add(LSTM(5, input_shape=(10,1)))
...
```

The **number of samples does not have to be specified**!

Shuffling Samples (Not)

Samples within an epoch are usually shuffled. This is a good practice when working with MLP networks. However, if we are trying to preserve state across samples, then the order of samples in the training set must be preserved!

```
#an LSTM layer with 5 memory cells/neurons
model = Sequential()
model.add(LSTM(5, input_shape=(10,1)))
...
history = model.fit(X, y, shuffle=False, batch_size=10, epochs=100)
```

Once fit, an history object is returned, providing a summary of the performance of the model during training, recorded at each epoch. These metrics can be plotted and analyzed in regard to overfitting or underfitting (we will get back to this later!)

The Importance of Batches

Mini-batch gradient descent with a batch size of 32 is a common configuration for LSTMs.

We know that each LSTM memory unit maintains internal state that is accumulated! By default, the internal state of all memory units in the network is reset after each batch!! This is what is called a **Stateless LSTM** (default behaviour)!

This means that the configuration of the batch size imposes a relation between:

- The efficiency of learning (how many samples are processed before an update)
- The speed of learning (how often weights are updated)
- The influence of the internal state (how often internal state is reset)

LSTMs State Management

However, tf.Keras provides flexibility so that we may control when to reset the internal state! This is what is called a **Stateful LSTM**!

```
...
#a batch size of 10 samples, with 5 timesteps and 1 feature
model.add(LSTM(2, stateful=True, batch_input_shape=(10, 5, 1)))
...
```

When stateful LSTM layers are used, we must also define the batch size as part of the input shape. The batch_input_shape argument requires a 3-dimensional tuple defined as batch size, time steps and features!

With stateful LSTM layers, the internal state is shared across batches (instead of being reset)!

LSTMs State Management

With stateful LSTM we have fine grained control over when to reset the internal state. Hence, if we want to reset it we must call the reset_states() function.

```
#reset the internal state at the end of each single epoch
for i in range(100):
    model.fit(X, y, epochs=1, batch_input_shape=(10, 5, 1))
    model.reset_states()
```

The same batch size used in the definition of the stateful LSTM must also be used when making predictions

```
...
predictions = model.predict(X, batch_size=10)
...
```

LSTMs State Management

The internal state is also accumulated when evaluating a network and when making predictions!

Therefore, if using a stateful LSTM we must reset the state after evaluating the network or after making predictions! <- Important!!

When sequences in different batches are related to each other, we should use stateful mode! Otherwise, when a sequence represents a complete sequence, we should go with stateless mode!

Some Last Tips

- LSTM input must be 3D. The meaning of the 3 input dimensions are:
  - Samples
  - Timesteps
  - Features
- The LSTM input layer is defined by the input shape argument on the first hidden layer
- The input shape argument takes a tuple of two values that define:
  - Timesteps
  - Features
- The number of samples is assumed to be 1 or more
- The reshape() function on NumPy arrays can be used to reshape your 1D or 2D data to be 3D

Vanilla LSTMs

The simplest LSTM configuration is the Vanilla LSTM

- We will name it Vanilla to differentiate it from deep LSTMs

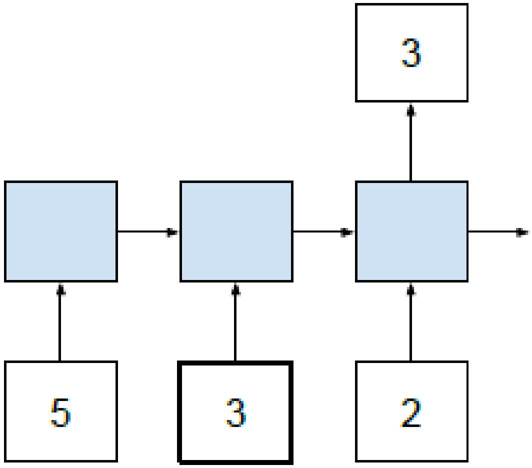
The Echo Sequence Prediction Problem

The Echo Sequence Prediction Problem

A nice problem for demonstrating the memory capability of LSTMs!

The task is that, given a sequence of random integers as input, to output the value of a random integer at a specific timestep that is not specified to the model!

For example - given an input sequence of random integers [5, 3, 2] with the chosen timestep being the second value, then the expected output is 3 (technically, it is a sequence classification problem).



The Echo Sequence Prediction Problem

Lets first generate random sequences and then one-hot encode such sequences!

```
#generate a sequence of random numbers [0, n_features=10) with sequence length=5
def generate_sequence(length, nr_features):
    return [randint(0, nr_features-1) for _ in range(length)]

#one hot encoded sequence
def one_hot_encode(sequence, nr_features):
    encoded = list()
    for value in sequence:
        one_hot_encoded = np.zeros(nr_features)
        one_hot_encoded[value] = 1
        encoded.append(one_hot_encoded)
    return array(encoded)

#decode a one hot encoded sequence
def one_hot_decode(encoded_seq):
    return [argmax(value) for value in encoded_seq]
```

The Echo Sequence Prediction Problem

Finally, reshape the input sequence to be 3D and create a function to create samples for us that are already encoded and that returns the X and the y (sequence as supervised)

```
#function to generate one sequence sample of random numbers [0, n_features=10)
#length of sequence (5) is the timesteps; the one hot encoded sequence (10) are the feature
#returns both the X (input) and the y (label)
def generate_sample(length, nr_features, out_index):
    #generate the sequence
    sequence = generate_sequence(length, nr_features)
    #one hot encode it
    encoded = one_hot_encode(sequence, nr_features)
    #reshape it to be 3D (1 sample, length timesteps, nr_features features)
    X = encoded.reshape((1, length, nr_features))
    #select the output (y) by getting the encoded value at the specified out_index
    #this must remain consistent for all generated examples for a model, so that it can learn
    y = encoded[out_index].reshape(1, nr_features)
    return X, y
```

The Echo Sequence Prediction Problem

Time to define and compile a Stateless Model! Lets use sequences with 5 timesteps, each one having a value between 0 and 9.

```
#control variables
timesteps = 5
features = 10
out_index = 2
epochs = 500
#define and compile the model
model = tf.keras.Sequential()
model.add(tf.keras.layers.LSTM(16, input_shape=(timesteps, features)))
model.add(tf.keras.layers.Dense(features, activation = 'softmax' ))
model.compile(
        loss= tf.keras.losses.categorical_crossentropy,
        optimizer= tf.keras.optimizers.Adam(learning_rate=0.001),
        metrics=['accuracy'])

#printing the model's summary
print(model.summary())
```

```
_________________________________________________
Layer (type)        Output Shape          Param#
=================================================
lstm (LSTM)         (None, 16)            1728
_________________________________________________
dense (Dense)       (None, 10)            170
=================================================
Total params: 1,898
Trainable params: 1,898
Non-trainable params: 0
_________________________________________________
None
```

The Echo Sequence Prediction Problem

Now we fit the Model. Accuracy will be either 0 or 1 because we are making sequence classification prediction on one sample and reporting the result. We then evaluate it!

```
#fit the model
for i in range(epochs):
    X, y = generate_sample(timesteps, features, out_index)
    history = model.fit(X, y, shuffle=False, epochs=1, verbose=0)
    print('Epoch: %d; Loss: %.2f; Accuracy: %.2f' %(i,
history.history['loss'][0], history.history['accuracy'][0]))

#evaluate the model by simply making predictions on randomly
#generated sequences and counting the number of correct predictions
correct = 0
for i in range(100):
    X, y = generate_sample(timesteps, features, out_index)
    yhat = model.predict(X)
    if one_hot_decode(yhat) == one_hot_decode(y):
        correct += 1
print('Accuracy: %.2f' %((correct/100)*100.0))
```

```
...
Epoch: 4994; Loss: 0.20;
Accuracy: 1.00
Epoch: 4995; Loss: 1.84;
Accuracy: 0.00
Epoch: 4996; Loss: 0.37;
Accuracy: 1.00
Epoch: 4997; Loss: 1.15;
Accuracy: 1.00
Epoch: 4998; Loss: 0.49;
Accuracy: 1.00
...

Accuracy: 57.00
```

The Echo Sequence Prediction Problem

Now we make predictions on new randomly generated sequences (well, pretty much the same we did when evaluating)!

```
#prediction on new sequence data
X, y = generate_sample(timesteps, features, out_index)
yhat = model.predict(X)

#print values
print('Sequence: %s' % [one_hot_decode(x) for x in X])
print('Expected: %s' % one_hot_decode(y))
print('Predicted: %s' % one_hot_decode(yhat))
```

```
Sequence: [[5, 8, 7, 7, 5]]
Expected: [7]
Predicted: [7]
```

Deep LSTMs

Algorithm 3: Function for LSTM model's conception and compilation.

```
Input: timesteps, multisteps, features, h_layers=2, h_neurons=64,
activation='relu', dropout_rate=0.5, deep_dense=False
Output: Sequential LSTM Model
model = Sequential()
while i ∈ range(h_layers) do
    if i == 0 then
        if i+1 == h_layers then
            model.add(CuDNNLSTM(h_neurons, return_sequences=False,
             input_shape=(timesteps, features)))
        else
            model.add(CuDNNLSTM(int(h_neurons/2),
             return_sequences=True, input_shape=(timesteps, features)))
            model.add(Dropout(dropout_rate))
        end
    else if i+1 == h_layers then
        model.add(CuDNNLSTM(h_neurons × 2, return_sequences=False))
    else
        model.add(CuDNNLSTM(h_neurons, return_sequences=True))
        model.add(Dropout(dropout_rate))
    end
end
model.add(Dense(h_neurons, activation=activation))
model.add(Dropout(dropout_rate))
if deep_dense then
    model.add(Dense(int(h_neurons/2), activation=activation))
    model.add(Dropout(dropout_rate))
model.add(Dense(multisteps))
model.compile(loss=rmse, optimizer= Adam(), metrics = [mae, rmse])
return model
```

Glossary

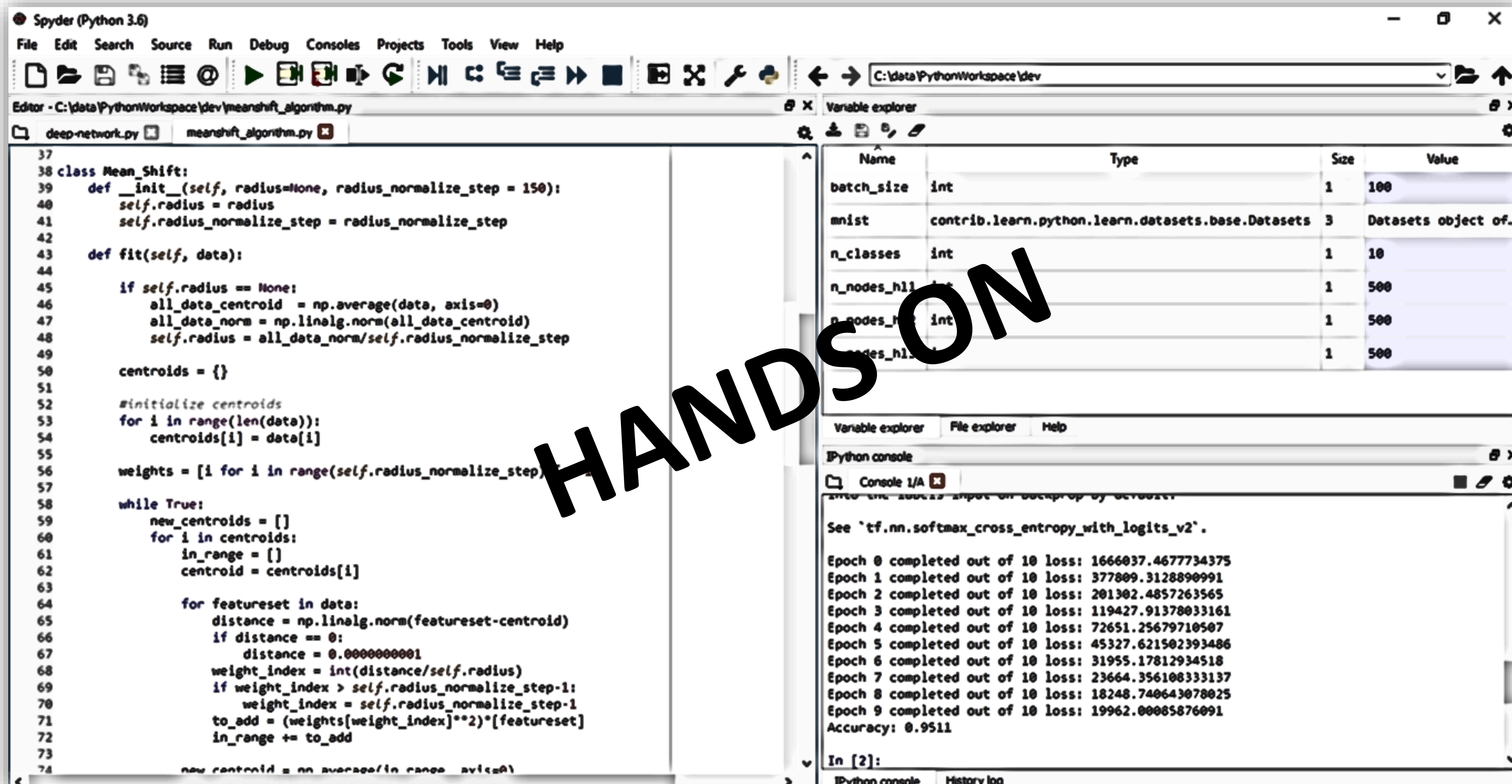
- Gates: used by RNNs such as LSTMs or GRUs, to decide how to update/handle the internal state of a memory cell
- LSTMs: Long Short-Term Memory networks, a type of RNNs
- Input Shape for LSTMs: 3D vector of (samples, timesteps, features)
- Single-step vs Multi-step: forecasting just the next value of the sequence (ex.: the next hour) vs forecasting multiple timesteps beyond the input sequences (ex.: the next twelve hours)
- Stateless vs Stateful LSTMs: internal state of all memory units in the network is reset after each batch automatically (default behavior) vs controlling when to reset the internal state
- Univariate vs Multivariate: using a single feature as input (ex.: speed_diff) vs using multiple features as input (ex.: speed_diff, time_diff and temperature)

Resources

- Official Documentation
  - https://www.tensorflow.org/api_docs/python/tf/keras/layers/LSTM
  - https://www.tensorflow.org/guide/keras/rnn
  - …

- Papers, Books, online courses, tutorials…
  - Understanding LSTM Networks https://colah.github.io/posts/2015-08-Understanding-LSTMs
  - The Unreasonable Effectiveness of Recurrent Neural Networks http://karpathy.github.io/2015/05/21/rnn-effectiveness
  - Learning to Forget: Continual Prediction with LSTM, 1999 https://pdfs.semanticscholar.org/e10f/98b86797ebf6c8caea6f54cacbc5a50e8b34.pdf
  - Hochreiter, S. & Schmidhuber, J., “Long Short-Term Memory”, Neural Computation 9(8), pp. 1735-1780, 1997. http://www.bioinf.jku.at/publications/older/2604.pdf
  - (Book) Long Short-Term Memory Networks With Python by Jason Brownlee

Hands On


HANDS ON
Spyder (Python 3.6)
File Edit Search Source Run Debug Consoles Projects Tools View Help
Editor - C:\data\PythonWorkspace\dev\meanshift_algorithm.py
deep-network.py
meanshift_algorithm.py
class Mean_Shift:
def __init__(self, radius=None, radius_normalize_step = 150):
self.radius = radius
self.radius_normalize_step = radius_normalize_step
def fit(self, data):
if self.radius == None:
all_data_centroid = np.average(data, axis=0)
all_data_norm = np.linalg.norm(all_data_centroid)
self.radius = all_data_norm/self.radius_normalize_step
centroids = {}
#initialize centroids
for i in range(len(data)):
centroids[i] = data[i]
while True:
new_centroids = []
for i in centroids:
in_range = []
centroid = centroids[i]
for featureset in data:
distance = np.linalg.norm(featureset-centroid)
if distance == 0:
distance = 0.0000000001
weight_index = int(distance/self.radius)
if weight_index > self.radius_normalize_step-1:
weight_index = self.radius_normalize_step-1
to_add = (weights[weight_index]**2)*[featureset]
in_range += to_add
Variable explorer
Name Type Size Value
batch_size int 1 100
mnist contrib.learn.python.learn.datasets.base.Datasets 3 Datasets object of…
n_classes int 1 10
500
500
500
Variable explorer File explorer Help
IPython console
Console 1/A
See `tf.nn.softmax_cross_entropy_with_logits_v2`.
Epoch 0 completed out of 10 loss: 1666037.4677734375
Epoch 1 completed out of 10 loss: 377809.3128890991
Epoch 2 completed out of 10 loss: 201302.4857263565
Epoch 3 completed out of 10 loss: 119427.91378033161
Epoch 4 completed out of 10 loss: 72651.25679710507
Epoch 5 completed out of 10 loss: 45327.621502393486
Epoch 6 completed out of 10 loss: 31955.17812934518
Epoch 7 completed out of 10 loss: 23664.356108333137
Epoch 8 completed out of 10 loss: 18248.740643078025
Epoch 9 completed out of 10 loss: 19962.00085876091
Accuracy: 0.9511
In [2]:
IPython console History log