Homework 3: Ensemble Methods and Recurrent Neural Networks

Large-Scale Data Analysis 2020

Stefano Pellegrini (mlq211)

May 31, 2020

1 AdaBoost (30 pts.)

Let $\{(\boldsymbol{x}_1, y_1), \dots, (\boldsymbol{x}_n, y_n)\} \in (\mathbb{R}^d \times \{-1, 1\})^n$ be the training data, $t = 1, \dots, T$ denote the boosting rounds. At round t, $w_i^{(t)}$ is the weight of training pattern i, $h^{(t)}$ is the base classifier from round t, $\alpha^{(t)}$ is the importance of this classifier, $\varepsilon^{(t)} = \sum_{j=1}^{n} w_j \mathbb{I}(h^{(t)}(\boldsymbol{x}_j) \neq y_j)$, and the aggregated classifier is $f^{(t)} = \sum_{i=1}^{t} \alpha^{(i)} h^{(i)}$.

1.1 Step 1

The goal is to prove for each i at any round t

$$w_i^{(t+1)} = w_i^{(t)} \frac{\exp\left(-\alpha^{(t)} h^{(t)}(\boldsymbol{x}_i) y_i\right)}{2\sqrt{\varepsilon^{(t)}\left(1 - \varepsilon^{(t)}\right)}} \quad . \tag{1}$$

We start from the update of the weights as defined in the lecture:

$$w_i^{(t+1)} = \begin{cases} w_i^{(t)} / \left(2\varepsilon^{(t)}\right) & \text{if } h^{(t)}(\boldsymbol{x}_i) \neq y_i \\ w_i^{(t)} / \left(2\left(1 - \varepsilon^{(t)}\right)\right) & \text{otherwise} \end{cases} \tag{2}$$

If we ensure that $\varepsilon^{(t)} < 0.5$, equation (2) guarantees that, at each round t, the weights of miss-classified training samples increase, and the weights of the correctly classified ones decrease. This is also true if we obtain $w_i^{(t+1)}$ by multiplying $w_i^{(t)}$ for $\exp(-\alpha^{(t)} h^{(t)}(\boldsymbol{x}_i) y_i)$, since $\exp\left(-\alpha^{(t)} h^{(t)}(\boldsymbol{x}_i) y_i\right) < 1$ if $h^{(t)}(\boldsymbol{x}_i) = y_i$ and $\exp\left(-\alpha^{(t)} h^{(t)}(\boldsymbol{x}_i) y_i\right) > 1$ if $h^{(t)}(\boldsymbol{x}_i) \neq y_i$. Also, as explained in the lecture, we want the weights, at any round t, to be normalized such that $\sum_{i=1}^{n} w_i^{(t)} = 1$, therefore, we divide the updated weights by $\sum_{i=1}^{n} w_i^{(t)} \exp(-\alpha^{(t)} h^{(t)}(\boldsymbol{x}_i) y_i)$ to obtain:

$$w_i^{(t+1)} = w_i^{(t)} \frac{\exp\left(-\alpha^{(t)} h^{(t)}(\boldsymbol{x}_i) y_i\right)}{\sum_{i=1}^{n} w_i^{(t)} \exp(-\alpha^{(t)} h^{(t)}(\boldsymbol{x}_i) y_i)} \tag{3}$$

As stated in [1], we can rewrite the denominator of the previous equation to get:

$$w_i^{(t+1)} = w_i^{(t)} \frac{\exp\left(-\alpha^{(t)} h^{(t)}(\boldsymbol{x}_i) y_i\right)}{\sum_{i: y_i = h^{(t)}(x_i)} w_i^{(t)} \exp(-\alpha^{(t)}) + \sum_{i: y_i \neq h_t(x_i)} w_i^{(t)} \exp(\alpha^{(t)})} \tag{4}$$

Substituting $\left(1 - \varepsilon^{(t)}\right)$ and $\varepsilon^{(t)}$ in equation (4), we obtain:

$$w_i^{(t+1)} = w_i^{(t)} \frac{\exp\left(-\alpha^{(t)} h^{(t)}(\boldsymbol{x}_i) y_i\right)}{(1 - \varepsilon^{(t)}) \exp(-\alpha^{(t)}) + \varepsilon \exp(\alpha^{(t)})} \tag{5}$$

Given that the importance of a classifier $\alpha^{(t)} = \frac{1}{2}\ln\left(\frac{1-\varepsilon^{(t)}}{\varepsilon^{(t)}}\right)$, and substituting it in the denominator of equation (5) we get:

$$
\begin{aligned}
w_i^{(t+1)} &= w_i^{(t)} \frac{\exp\left(-\alpha^{(t)} h^{(t)}(\boldsymbol{x}_i) y_i\right)}{(1-\varepsilon^{(t)})\exp(-\frac{1}{2}\ln(\frac{1-\varepsilon}{\varepsilon})) + \varepsilon\exp(\frac{1}{2}\ln(\frac{1-\varepsilon}{\varepsilon}))} \\
&= w_i^{(t)} \frac{\exp\left(-\alpha^{(t)} h^{(t)}\left(\boldsymbol{x}_i\right) y_i\right)}{2\sqrt{\varepsilon^{(t)}\left(1-\varepsilon^{(t)}\right)}}
\end{aligned} \tag{6}
$$

Therefore, as we can see, we obtained equation (1) starting from equation (2).

Furthermore, we can show that, for wrongly classified data points $h^{(t)}(\boldsymbol{x}_i)y_i = -1$, therefore, substituting -1 to $h^{(t)}(\boldsymbol{x}_i)y_i$ in equation (1), it follows:

$$
\begin{aligned}
w^{(t+1)} &= w_i^{(t)} \frac{\exp(\alpha)}{2\sqrt{\varepsilon^{(t)}\left(1-\varepsilon^{(t)}\right)}} \\
&= w_i^{(t)} \frac{\exp(\frac{1}{2}\ln(\frac{1-\varepsilon^{(t)}}{\varepsilon^{(t)}}))}{2\sqrt{\varepsilon^{(t)}\left(1-\varepsilon^{(t)}\right)}} = \frac{w_i^{(t)}}{2\varepsilon^{(t)}}
\end{aligned} \tag{7}
$$

Also, for correctly classified data points $h^{(t)}(\boldsymbol{x}_i)y_i = 1$, consequently, substituting 1 to $h^{(t)}(\boldsymbol{x}_i)y_i$ in equation (1), we obtain:

$$
\begin{aligned}
w^{(t+1)} &= w_i^{(t)} \frac{\exp(-\alpha)}{2\sqrt{\varepsilon^{(t)}\left(1-\varepsilon^{(t)}\right)}} \\
&= w_i^{(t)} \frac{\exp(-\frac{1}{2}\ln(\frac{1-\varepsilon^{(t)}}{\varepsilon^{(t)}}))}{2\sqrt{\varepsilon^{(t)}\left(1-\varepsilon^{(t)}\right)}} \\
&= \frac{w_i^{(t)}}{\exp(\frac{1}{2}\ln(\frac{1-\varepsilon^{(t)}}{\varepsilon^{(t)}}))2\sqrt{\varepsilon^{(t)}\left(1-\varepsilon^{(t)}\right)}} = \frac{w_i^{(t)}}{2(1-\varepsilon^{(t)})}
\end{aligned} \tag{8}
$$

As we can see, equation (7) and (9) show that equation (2) can be rewritten as equation (1).

1.2 Step 2

The goal is to prove for each i at any round t

$$
w_i^{(t)} \frac{\exp\left(-\alpha^{(t)} h^{(t)}(\boldsymbol{x}_i) y_i\right)}{2\sqrt{\varepsilon^{(t)}\left(1-\varepsilon^{(t)}\right)}} = \frac{\exp\left(-f^{(t)}(\boldsymbol{x}_i) y_i\right)}{n\prod_{\tau=1}^{t} 2\sqrt{\varepsilon^{(\tau)}\left(1-\varepsilon^{(\tau)}\right)}} \ . \tag{9}
$$

Since the weights on the data points at round $t+1$, depend on the weights and on the performance of the classifier at round t, they can be recursively computed from round $t+1$ to $t-1$ as:

$$
\begin{aligned}
w_i^{(t+1)} &= w_i^{(t)} \frac{\exp\left(-\alpha^{(t)} h^{(t)}(\boldsymbol{x_i}) y_i\right)}{2\sqrt{\varepsilon^{(t)}\left(1-\varepsilon^{(t)}\right)}} \\
&= w_i^{(t-1)} \frac{\exp(-\alpha^{(t-1)} h^{(t-1)}(\boldsymbol{x_i}) y_i)}{2\sqrt{\varepsilon^{(t-1)}\left(1-\varepsilon^{(t-1)}\right)}} \frac{\exp(-\alpha^{(t)} h^{(t)}(\boldsymbol{x_i}) y_i)}{2\sqrt{\varepsilon^{(t)}\left(1-\varepsilon^{(t)}\right)}}
\end{aligned} \tag{10}
$$

Rearranging the previous equation, we get:

$$w_i^{(t+1)} = w_i^{(t-1)} \frac{\exp((-\alpha^{(t-1)}h^{(t-1)}(\boldsymbol{x_i})y_i) + (-\alpha^{(t)}h^{(t)}(\boldsymbol{x_i})y_i))}{2\sqrt{\varepsilon^{(t-1)}\left(1-\varepsilon^{(t-1)}\right.}2\sqrt{\varepsilon^{(t)}\left(1-\varepsilon^{(t)}\right))}} \tag{11}$$

If we fully iterate to round 1, since the weights are equally initialized for all examples as $w^{(1)} = \left(\frac{1}{n}, \dots, \frac{1}{n}\right) \in \mathbb{R}^n$, it follows that:

$$w_i^{(t+1)} = \frac{1}{n} \frac{\exp(\sum_{i=1}^{t} -\alpha^{(i)}h^{(i)}(\boldsymbol{x_i})y_i)}{\prod_{\tau=1}^{t} 2\sqrt{\varepsilon^{(\tau)}\left(1-\varepsilon^{(\tau)}\right)}} \tag{12}$$

Since, at any round t, the decision function $f(t) = \sum_{t=1}^{t} \alpha^{(t)}h^{(t)}$, we can write the previous equation as:

$$w_i^{(t+1)} = \frac{\exp\left(-f^{(t)}\left(\boldsymbol{x}_i\right)y_i\right)}{n\prod_{\tau=1}^{t} 2\sqrt{\varepsilon^{(\tau)}\left(1-\varepsilon^{(\tau)}\right)}} \tag{13}$$

1.3 Step 3

The goal is to prove for each i at any round t

$$\frac{\exp\left(-f^{(t)}(\boldsymbol{x}_i)y_i\right)}{n\prod_{\tau=1}^{t} 2\sqrt{\varepsilon^{(\tau)}\left(1-\varepsilon^{(\tau)}\right)}} = \frac{\exp\left(-f^{(t)}(\boldsymbol{x}_i)y_i\right)}{\sum_{i=1}^{n}\exp\left(-f^{(t)}(\boldsymbol{x}_i)y_i\right)} \quad . \tag{14}$$

The weights are normalized, that is

$$\begin{aligned} \sum_{i=1}^{n} w_i^{(t+1)} &= 1 \\ \sum_{i=1}^{n} \frac{\exp\left(-f^{(t)}\left(\boldsymbol{x}_i\right)y_i\right)}{n\prod_{\tau=1}^{t} 2\sqrt{\varepsilon^{(\tau)}\left(1-\varepsilon^{(\tau)}\right)}} &= 1 \end{aligned} \tag{15}$$

We can simply move the denominator to the right side of the equation to get:

$$\sum_{i=1}^{n} \exp\left(-f^{(t)}\left(\boldsymbol{x}_i\right)y_i\right) = n\prod_{\tau=1}^{t} 2\sqrt{\varepsilon^{(\tau)}\left(1-\varepsilon^{(\tau)}\right)} \tag{16}$$

Finally, we just need to substitute equation (16) in equation (13) to obtain:

$$w_i^{(t+1)} = \frac{\exp\left(-f^{(t)}\left(\boldsymbol{x}_i\right)y_i\right)}{n\prod_{\tau=1}^{t} 2\sqrt{\varepsilon^{(\tau)}\left(1-\varepsilon^{(\tau)}\right)}} = \frac{\exp\left(-f^{(t)}\left(\boldsymbol{x}_i\right)y_i\right)}{\sum_{i=1}^{n}\exp\left(-f^{(t)}\left(\boldsymbol{x}_i\right)y_i\right)} \tag{17}$$

2 Gradient Boosting (10 pts.)

The assumption stated in question 2 is in general not true, it is an approximation used to simplify the optimization problem $\operatorname{argmin}_{\hat{y}_1,\dots,\hat{y}_n} \sum_{i=1}^{n} \mathcal{L}\left(y_i, \hat{y}_i\right)$, which under this assumption can be solved as $\operatorname{argmin}_{\hat{y}_i} \mathcal{L}\left(y_i, \hat{y}_i\right)$, for each $i = 1, \dots, n$.

As explained in the lectures, in gradient boosting we start with a base classifier, and we want to minimize the loss of the model by adding weak sequential classifiers incrementally, such that, at each round, the new classifier focus on reducing the residuals made by the previous one. So, since we incrementally add new sequential classifiers, that are trained to correct the mistakes made by the previous ones, and since the prediction of a boosting tree model, at round t, is $\hat{y}_i^{(t)} = \sum_{k=1}^{t} f_k\left(\boldsymbol{x}_i\right)$, the assumption that the predictions of the model $\hat{y}_i^{(t)}$, for some round t and $i = 1, \dots, n$ are independent, is in general not true.

3 Recurrent Neural Networks (30 pts.)

Replace the simple recurrent layer in `LSDA2020_RNN1.ipynb` notebook with a LSTM layer. Follow the equations from `LSDA2020_RNN2.ipynb` to implement the LSTM. You will need to add more variables and extend the step function. Also remember that the LSTM not only transfers the hidden state h_t to the next time step $t+1$, but also the cell state c_t.

I did the following modification to the `LSDA2020_RNN2.ipynb` notebook:

- Added the parameters initialization for all variables used in the LSTM:

```
# Initialize forget gate parameters
W_f = tf.Variable(np.random.rand(dims, num_neurons),
                  dtype=tf.float32)
U_f = tf.Variable(np.random.rand(num_neurons, num_neurons),
                  dtype=tf.float32)
b_f = tf.Variable(np.zeros((1, num_neurons)),
                  dtype=tf.float32)
# Initialize input gate parameters
W_i = tf.Variable(np.random.rand(dims, num_neurons),
                  dtype=tf.float32)
U_i = tf.Variable(np.random.rand(num_neurons, num_neurons),
                     dtype=tf.float32)
b_i = tf.Variable(np.zeros((1, num_neurons)),
                  dtype=tf.float32)
# Initialize modulation gate parameters
W_j = tf.Variable(np.random.rand(dims, num_neurons),
                  dtype=tf.float32)
U_j = tf.Variable(np.random.rand(num_neurons, num_neurons),
                     dtype=tf.float32)
b_j = tf.Variable(np.zeros((1, num_neurons)),
                  dtype=tf.float32)
# Initialize output gate parameters
W_o = tf.Variable(np.random.rand(dims, num_neurons),
                  dtype=tf.float32)
U_o = tf.Variable(np.random.rand(num_neurons, num_neurons),
                          dtype=tf.float32)
b_o = tf.Variable(np.zeros((1, num_neurons)),
                  dtype=tf.float32)
# Initialize hidden state
h_0 = tf.Variable(np.zeros((1, num_neurons),
                  dtype=np.float32))
# Initialize cell state
c_0 = tf.Variable(np.zeros((1, num_neurons),
                  dtype=np.float32))
```

- Changed the parameters that are updated during the gradient step:

```
# Define which parameters get updated during the gradient step
trainable_vars = [W_f, U_f, b_f, W_i, U_i, b_i,
                  W_j, U_j, b_j, W_o, U_o, b_o,
                  W_y, b_y]
if learn_h0:
    trainable_vars.append(h_0)
if learn_c0:
    trainable_vars.append(c_0)
```

- Extended step and iterate_series functions by adding the LSTM operations to the first one, and the cell state to the second one:

```
@tf.function
def step(x_t, c, h):
    ## LSTM layer
    # Forget gate
    f = tf.nn.tanh(
        tf.matmul(x_t, W_f) + tf.matmul(h, U_f) + b_f)
    # Input gate
    i = tf.nn.tanh(
        tf.matmul(x_t, W_i) + tf.matmul(h, U_i) + b_i)
    # Modulation gate
    j = tf.nn.tanh(
        tf.matmul(x_t, W_j) + tf.matmul(h, U_j) + b_j)
    # Output gate
    o = tf.nn.tanh(
        tf.matmul(x_t, W_o) + tf.matmul(h, U_o) + b_o)
    # Cell state
    c = f * c + i * j
    # Hidden state
    h = o * tf.nn.tanh(c)

    ## Dense layer
    y_hat = tf.matmul(h, W_y) + b_y
    return y_hat, c, h

@tf.function
def iterate_series(x, c, h):
    y_hat = []
    # iterate over time axis (1)
    for t in range(x.shape[1]):
        # give previous hidden state and input from the current time
    step
        y_hat_t, c, h = step(x[:, t], c, h)
        y_hat.append(y_hat_t)
    y_hat = tf.stack(y_hat, 1)
    return y_hat, c, h
```

- Passed on the hidden state h_t and the cell state c_t in the training loop and the iterate series function (iterate_series is shown in the previous point) (the "..." indicates that there are omitted lines of code):

```
...
# record any gradient "action" in the following section
        with tf.GradientTape() as tape:
            if h is None:
                # initialize cell state (c_0) and hidden state (h_0)
                c = tf.repeat(c_0, batch_size, 0)
                h = tf.repeat(h_0, batch_size, 0)
            ...
            # get predictions for this part (forward pass)
            y_hat, c, h = iterate_series(x_part, c, h)
            ...
```

- Initialize the cell state c_t and hidden state h_t to zero, as requested:

```
# learn a better initial hidden or cell state representation
learn_h0 = False
learn_c0 = False
```

The training loss after 250 training iterations is 0.006231256. Figure 1 shows the input x, a sinus signal $\sin(x_t)$, the target y, the signal 1 step ahead $\sin(x_{t+1})$, and the LSTM target prediction. We can see that the model prediction is reasonably good, except from the first time steps, which is due to the fact that we did not learn an hidden and cell states initialization better than zero. Please find the code attached in the file `HW3_exercise3.ipynb`.

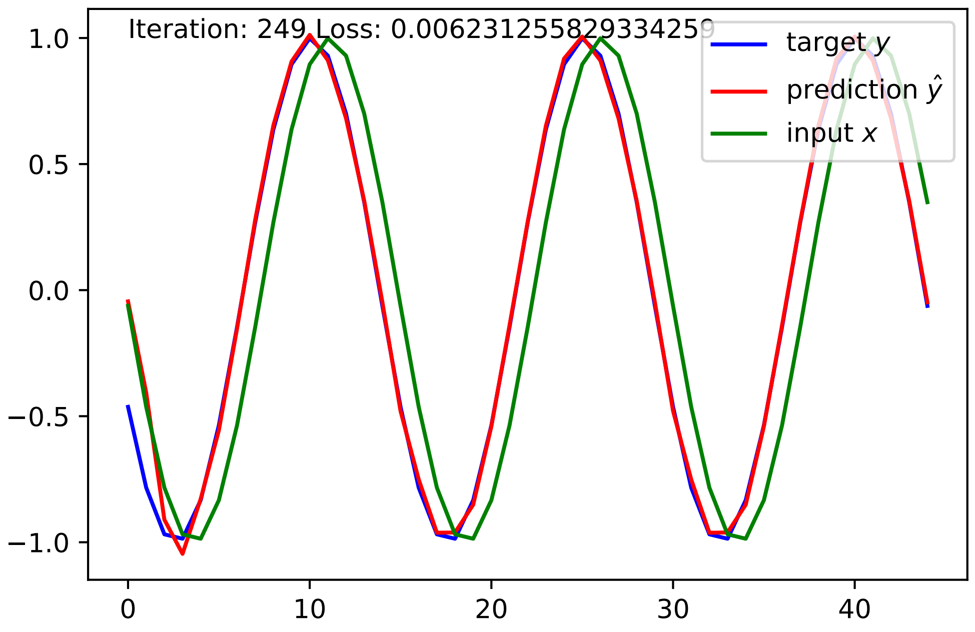

Figure 1: Figure of the LSTM learned prediction. The green line is the input x, sinus signal $\sin(x_t)$. The blue line is the target y, the signal 1 step ahead $\sin(x_{t+1})$. The red line is the LSTM prediction $\hat{y}$ of the signal 1 step ahead.

4 Recurrent Neural Networks in Keras (30 pts.)

Add different components to the RNN from `LSDA2020_RNN2.ipynb` and report the results on the validation set (the changed parts in the code).

In this exercise, I started from an LSTM model, and in each bullet I added different component to it. Also, for each model, I reported the mean absolute error on the validation data and a model overview. The base line model, used as reference to test the different components, is defined as following:

```
# LSTM model
def get_model_rnn(shape):
    inp = Input(shape)
    x = LSTM(64, return_sequences=False)(inp)
    x = Dense(1)(x)

    model = Model(inp, x)
    return model
```

We can see the overview of the base line LSTM model:

```

Model: "LSTM"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_3 (InputLayer)         [(None, 19, 22)]          0
_________________________________________________________________
bidirectional (Bidirectional (None, 128)               44544
_________________________________________________________________
dense_3 (Dense)              (None, 1)                 129
=================================================================
Total params: 44,673
Trainable params: 44,673
Non-trainable params: 0
_________________________________________________________________
```

The base line model had a mean validation error of 1.970323.
In all models defined in this exercise, I used a stockastic gradient descent (SGD) optimizer with a learning rate of 0.01, 5 epochs and batch size of 32. The only change to the optimizer is performed in exercise 4.4, where I added gradient clipping.

1. Add bidirectional sequence processing by utilizing `tf.keras.layers.Bidirectional`.

   In this model, I replaced the LSTM layer in the base line model, with a Bidirectional-LSTM, as shown in the following code:

   ```
   # Bidirectional LSTM
   def get_model_rnn_bidi(shape):
       inp = Input(shape)
       x = Bidirectional(LSTM(64, return_sequences=False))(inp)
       x = Dense(1)(x)
       model = Model(inp, x)
       return model
   ```

   The model overview is the following:

   ```
   Model: "Bidirectional-LSTM"
   _________________________________________________________________
   Layer (type)                 Output Shape              Param #
   =================================================================
   input_4 (InputLayer)         [(None, 19, 22)]          0
   _________________________________________________________________
   bidirectional (Bidirectional (None, 128)               44544
   _________________________________________________________________
   dense_4 (Dense)              (None, 1)                 129
   =================================================================
   Total params: 44,673
   Trainable params: 44,673
   Non-trainable params: 0
   _________________________________________________________________
   ```

   The mean validation error of the bidirectional LSTM was 1.955470.

2. Stack 2 LSTM layers. What is the difference to bidirectional processing? (you may need to use the `return_sequences` parameter)

   I modified the base line LSTM model by stacking 2 LSTM layers as shown:

   ```
   # Stack LSTM
   def get_model_rnn_stack(shape):
       inp = Input(shape)
       x = LSTM(64, return_sequences=True)(inp)
       x = LSTM(64, return_sequences=False)(x)
       x = Dense(1)(x)

       model = Model(inp, x)
       return model
   ```

   The model overview is the following:

   ```
   Model: "Stack-LSTM"
   _________________________________________________________________
   Layer (type)                 Output Shape              Param #
   =================================================================
   input_7 (InputLayer)         [(None, 19, 22)]          0
   _________________________________________________________________
   ```

```
lstm_6 (LSTM)                (None, 19, 64)            22272
_________________________________________________________________
lstm_7 (LSTM)                (None, 64)                33024
_________________________________________________________________
dense_7 (Dense)              (None, 1)                 65
=================================================================
Total params: 55,361
Trainable params: 55,361
Non-trainable params: 0
```

The mean validation error of the model with two stacked LSTM layers was 2.025149.

Differences between bidirectional and 2 stacked LSTM layers processing:
In a *stacked LSTM* sequence processing, the sequence is read from left to right by each layer, and the layer above provides a sequence output rather than a single value output to the LSTM layer below. While, a *bidirectional LSTM*, instead of reading the time series just left to right, it uses one LSTM layer that read the time series forward, and an other one backward, and then the output of both layers get concatenated, summed up, or selected and passed to the output layer. By reading the input sequence in both directions, the output layer can get information from past (backwards) and future (forward) states simultaneously, but this method can only be applied in problems where all timesteps of the input sequence are available.

3. Add a 1-d convolution layer (`tf.keras.layers.Conv1D`) before the recurrent part.

I added a 1-d convolution layer, to the base line LSTM model, as shown:

```
# Conv1d LSTM
def get_model_rnn_conv(shape):
    inp = Input(shape)
    x = Conv1D(32, kernel_size=3, strides=1)(inp)
    x = LSTM(64, return_sequences=False)(x)
    x = Dense(1)(x)

    model = Model(inp, x)
    return model
```

The model overview is the following:

```
Model: "Conv1D-LSTM"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_5 (InputLayer)         [(None, 19, 22)]          0
_________________________________________________________________
conv1d (Conv1D)              (None, 17, 32)            2144
_________________________________________________________________
lstm_4 (LSTM)                (None, 64)                24832
_________________________________________________________________
dense_5 (Dense)              (None, 1)                 65
=================================================================
Total params: 27,041
Trainable params: 27,041
Non-trainable params: 0
_________________________________________________________________
```

The mean validation error of the LSTM with 1-d convolution layer is 2.032951.

Differences between convolutional layer and recurrent model time series processing:

A *1-d convolutional layer* can be used for time series processing, it takes as input a n by j sequence, where n is the time series length and j is the number of features of the input, and it uses k number of kernels of given size to perform feature extraction.

Each kernel has its own weights, and it process the time series from beginning to end performing convolution (by sliding the filter over the input and summing the result of a matrix multiplication with the addition of a bias), and output a ”filtered” sequence (feature map) of reduced length (if `padding="same"` not specified) and k features, where k is the number of kernels used. Then, a non linear activation function can be applied to the output, before it is passed to the next layer.

A *recurrent model* can be used to recurrently process an input signal, represented as time steps of the series $(\ldots \boldsymbol{x}_{t-1}, \boldsymbol{x}_t, \boldsymbol{x}_{t+1} \ldots)$, or of the finite series $(\boldsymbol{x}_0, \ldots, \boldsymbol{x}_T)$, which can be modeled as different sequence tasks (one to one, one to many, many to one, many to many async, and many to many sync) depending on the nature of the problem and on how we model the data (fixed or not fixed size input or output). In a RNN, at each time step t, the output o of an hidden state h, is defined as

$$\mathbf{o} \leftarrow \mathbf{h}_t = \sigma\left(W \cdot \mathbf{x}_t + U \cdot \mathbf{h}_{t-1} + \mathbf{b}\right) \tag{18}$$

, so the input x at time t is multiplied for the weight matrix W, to which is added the output of the previous hidden state h_{t-1} (which explain the recurrency, since the hidden state reference itself), multiplied for its weight matrix U, finally the bias is added and a non linear activation function is applied to the result.

4. Gradient clipping can be helpful to train recurrent networks. Keras offers to clip gradients directly through the optimizer. Try this with clip values of 1.

   I added gradient clipping, to the base line LSTM model, as shown in the following code:

```
sgd_clipped = SGD(lr=0.01, momentum=0.9, nesterov=True, clipvalue = 1)
```

   The mean validation error of the LSTM model with gradient clipping was 2.121872.

Table 1. shows the performance of the different tested models. Please find the code attached in the file `HW3_exercise4.ipynb`.

Table 1: Mean absolute validation error of the tested models.

	LSTM	Bidirectional LSTM	Stack LSTM	1-D Conv LSTM	G. Clipping LSTM
MAE	1.970323	1.955470	2.025149	2.032951	2.121872

References

[1] M. B. W. B. Kai Arras, Cyrill Stachniss. Robotics 2, adaboost for people and place detection. `http://ais.informatik.uni-freiburg.de/teaching/ws09/robotics2/pdfs/rob2-10-adaboost.pdf`, 2012.