Homework 3: Ensemble Methods and Recurrent Neural Networks

Large-Scale Data Analysis 2020

Shahriyar Mahdi Robbani (XQR418)

May 27, 2020

1 AdaBoost

Let $\{(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_n, y_n)\} \in (\mathbb{R}^d \times \{-1, 1\})^n$ be the training data, $t = 1, \ldots, T$ 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:

$$w_i^{(t+1)} = \begin{cases} w_i^{(t)} / (2\epsilon^{(t)}) & \text{if } h^{(t)}(\boldsymbol{x}_j) \neq y_j \\ w_i^{(t)} / (2(1 - \epsilon^{(t)})) & \text{otherwise} \end{cases} \tag{2}$$

And importance $\alpha^{(t)}$ of each weight defined in the lecture as:

$$\alpha^{(t)} = \frac{1}{2} \ln\left(\frac{1 - \varepsilon^{(t)}}{\varepsilon^{(t)}}\right) \tag{3}$$

The prediction $h^{(t)}(\boldsymbol{x}_j)$ and class y_j can only take the values 1 and -1. Therefore when $h^{(t)}(\boldsymbol{x}_j) = y_j$, we can rewrite equation (1) as:

$$w_i^{(t+1)} = w_i^{(t)} \frac{\exp\left(-\alpha^{(t)}\right)}{2\sqrt{\varepsilon^{(t)}\left(1 - \varepsilon^{(t)}\right)}} \tag{4}$$

Substituting equation (3) into equation (4) we get:

$$\begin{aligned} w_i^{(t+1)} &= w_i^{(t)} \frac{\exp\left(-\frac{1}{2}\ln\left(\frac{1-\varepsilon^{(t)}}{\varepsilon^{(t)}}\right)\right)}{2\sqrt{\varepsilon^{(t)}\left(1 - \varepsilon^{(t)}\right)}} \\ &= w_i^{(t)} \frac{\left(\frac{1-\varepsilon^{(t)}}{\varepsilon^{(t)}}\right)^{-\frac{1}{2}}}{2\sqrt{\varepsilon^{(t)}\left(1 - \varepsilon^{(t)}\right)}} \\ &= w_i^{(t)} \frac{\sqrt{1 - \varepsilon^{(t)}}}{\sqrt{\varepsilon^{(t)}}} \frac{1}{2\sqrt{\varepsilon^{(t)}}\sqrt{1 - \varepsilon^{(t)}}} \\ &= w_i^{(t)} / (2\epsilon^{(t)}) \end{aligned} \tag{5}$$

Similarly, when $h^{(t)}(\boldsymbol{x}_j) \neq y_j$, we can rewrite equation (1) as:

$$w_i^{(t+1)} = w_i^{(t)} \frac{\exp\left(\alpha^{(t)}\right)}{2\sqrt{\varepsilon^{(t)}\left(1-\varepsilon^{(t)}\right)}} \tag{6}$$

Substituting equation (3) into equation (6) we get:

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

We can see equations (5) and (7) are the two cases of equation (2), we have proved that equation (1) equals equation (2).

1.2 Step 2

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

$$w_i^{(t)} \frac{\exp\left(-\alpha^{(t)} h^{(t)}(\boldsymbol{x}_i) y_i\right)}{2\sqrt{\varepsilon^{(t)}\left(1-\varepsilon^{(t)}\right)}} = \frac{\exp\left(-f^{(t)}(\boldsymbol{x}_i) y_i\right)}{n\prod_{\tau=1}^{t} 2\sqrt{\varepsilon^{(\tau)}\left(1-\varepsilon^{(\tau)}\right)}} \ . \tag{8}$$

Since each weight depends on the previous weight, we can calculate the weights recursively. So at round t we have a weight:

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

Which is the same as:

$$w_i^{(t)} = w_i^{(t-2)} \frac{\exp\left(-\alpha^{(t-2)} h^{(t-2)}(\boldsymbol{x}_i) y_i\right)}{2\sqrt{\varepsilon^{(t-2)}\left(1-\varepsilon^{(t-2)}\right)}} \frac{\exp\left(-\alpha^{(t-1)} h^{(t-1)}(\boldsymbol{x}_i) y_i\right)}{2\sqrt{\varepsilon^{(t-1)}\left(1-\varepsilon^{(t-1)}\right)}} \tag{10}$$

Going all the way down to the base case, where $w_1 = \frac{1}{n}$, this becomes:

$$\begin{aligned} w_i^{(t)} &= \frac{1}{n} \frac{\prod_{\tau=1}^{t} \exp\left(-\alpha^{(\tau)} h^{(\tau)}(x_i)(y_i)\right)}{\prod_{\tau=1}^{t} 2\sqrt{\varepsilon^{(\tau)}\left(1-\varepsilon^{(\tau)}\right)}} \\ &= \frac{\exp\left(\sum_{\tau=1}^{t} -\alpha^{(\tau)} h^{(\tau)}(x_i)(y_i)\right)}{n\prod_{\tau=1}^{t} 2\sqrt{\varepsilon^{(\tau)}\left(1-\varepsilon^{(\tau)}\right)}} \end{aligned} \tag{11}$$

The decision function at round t is defined as:

$$f^{(t)} = \sum_{i=1}^{t} \alpha^{(i)} h^{(i)} \tag{12}$$

Substituting this into equation (11) we get:

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

1.3 Step 3

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

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

The weights are normalized, that is:

$$\sum_{i=1}^{n} w_i^{(t)} = 1$$

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

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

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

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

Substituting equation (15) into equation (13) we get:

$$w_i^{(t} = \frac{\exp\left(-f^{(t)}(\boldsymbol{x}_i)y_i\right)}{\sum_{l=1}^{n}\exp\left(-f^{(t)}(\boldsymbol{x}_l)y_l\right)} \quad . \tag{16}$$

2 Gradient Boosting

In gradient boosting, samples x_i are weighted at each round and have a larger weight if that sample was incorrectly predicted. This means different distributions of x_i are used in each round and this distribution depends on the error made on the previous round so the predicted $\hat{y}_i^{(t)}$ are not independent from each other.

3 Recurrent Neural Networks

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.

The following parameters were added:

```
# LSTM parameters
# Forget layer parameters
W_f = tf.Variable(np.random.rand(dims, num_neurons), dtype=tf.float32)
b_f = tf.Variable(np.zeros((1, num_neurons)), dtype=tf.float32)
U_f = tf.Variable(np.random.rand(num_neurons, num_neurons), dtype=tf.
    float32)
# Input layer parameters
W_i = tf.Variable(np.random.rand(dims, num_neurons), dtype=tf.float32)
b_i = tf.Variable(np.zeros((1, num_neurons)), dtype=tf.float32)
U_i = tf.Variable(np.random.rand(num_neurons, num_neurons), dtype=tf.
    float32)
# Modulation layer parameters
W_j = tf.Variable(np.random.rand(dims, num_neurons), dtype=tf.float32)
```

```
b_j = tf.Variable(np.zeros((1, num_neurons)), dtype=tf.float32)
U_j = tf.Variable(np.random.rand(num_neurons, num_neurons), dtype=tf.
    float32)
# Output layer parameters
W_o = tf.Variable(np.random.rand(dims, num_neurons), dtype=tf.float32)
b_o = tf.Variable(np.zeros((1, num_neurons)), dtype=tf.float32)
U_o = tf.Variable(np.random.rand(num_neurons, num_neurons), dtype=tf.
    float32)
# Fully connected layer parameters
W_y = tf.Variable(np.random.rand(num_neurons, dims), dtype=tf.float32)
b_y = tf.Variable(np.zeros((1, dims)), dtype=tf.float32)

# Initial hidden state and cell state
h_0 = tf.Variable(np.zeros((1, num_neurons), dtype=np.float32))
c_0 = tf.Variable(np.zeros((1, num_neurons), dtype=np.float32))

# backpropagation
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]
```

The `step` function was changed to include all the LSTM gates:

```
@tf.function
def step(x_t, h, c):
    # forget layer
    f = tf.nn.sigmoid(
        tf.matmul(x_t, W_f) + tf.matmul(h, U_f) + b_f
    )
    # input layer:
    i = tf.nn.sigmoid(
        tf.matmul(x_t, W_i) + tf.matmul(h, U_i) + b_i
    )
    # modulation layer:
    j = tf.nn.tanh(
        tf.matmul(x_t, W_j) + tf.matmul(h, U_j) + b_j
    )
    # output layer:
    o = tf.nn.sigmoid(
        tf.matmul(x_t, W_o) + tf.matmul(h, U_o) + b_o
    )
    # memory (cell) layer:
    c = tf.math.multiply(f,c) + tf.math.multiply(i,j)
    # hidden state
    h = tf.math.multiply(o, tf.nn.tanh(c))
    # fully connected
    y_hat = tf.matmul(h, W_y) + b_y
    return y_hat, h, c
```

The `iterate_series` function was changed to include the cell state as an output and input.

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

The initial cell state and the modified iterate series functions were added to the training loop:

```
# train for a set number of iterations
for iteration in range(n_iterations):
    # generates a long time series / normally loaded from dataset (e.g.
    stocks, weather)
    x, y = generateData(signal_length, predict_ahead, signal_repeats,
    batch_size, noise_strength)
    h = None
    predictions = None
    loss_list = []
    grads = None

    # do not feed complete series, but chunks of it (
    truncated_backprop_length)
    for i in range(0, total_series_length, truncated_backprop_length):

        with tf.GradientTape() as tape:
            if h is None:
                # initialize hidden state (h_0) -> new shape (batch_size,
    num_neurons)
                h = tf.repeat(h_0, batch_size, 0)
                c = tf.repeat(c_0, batch_size, 0) #added
            x_part = x[:, i: i + truncated_backprop_length]
            y_part = y[:, i: i + truncated_backprop_length]

            # get predictions for this part (forward pass)
            y_hat, h, c = iterate_series(x_part, h, c) # changed

            # calculate mean squared error
            loss = tf.reduce_mean((y_hat - y_part)**2)
        # backprop
        if grads is None:
            grads = tape.gradient(loss, trainable_vars)
        else:
            grads = grads + tape.gradient(loss, trainable_vars)

        loss_list.append(loss)
        # combine with previous predictions
        predictions = tf.concat([predictions, y_hat], 1) if predictions is
    not None else y_hat
```

The mean training loss for the model was 0.003. Figure 1 shows the learned predictions.

4 Recurrent Neural Networks in Keras

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

1. Add bidirectional sequence processing by utilizing `tf.keras.layers.Bidirectional`.
2. Stack 2 LSTM layers. What is the difference to bidirectional processing? (you may need to use the `return_sequences` parameter)
3. Add a 1-d convolution layer (`tf.keras.layers.Conv1D`) before the recurrent part. You will need to reshape the data, you can use the `tf.keras.layers.Reshape` layer for that.
4. Gradient clipping can be a helpful to train recurrent networks. Keras offers to clip gradients directly through the optimizer. Try this with clip values of 0.1, 1, and 10.

4.1 Bidirectional

```
def get_model_bidirectional(name="Bidirectional", shape=INPUT_SHAPE):
    inp = Input(shape)
    x = Bidirectional(LSTM(64))(inp)
    x = Dense(1)(x)
```

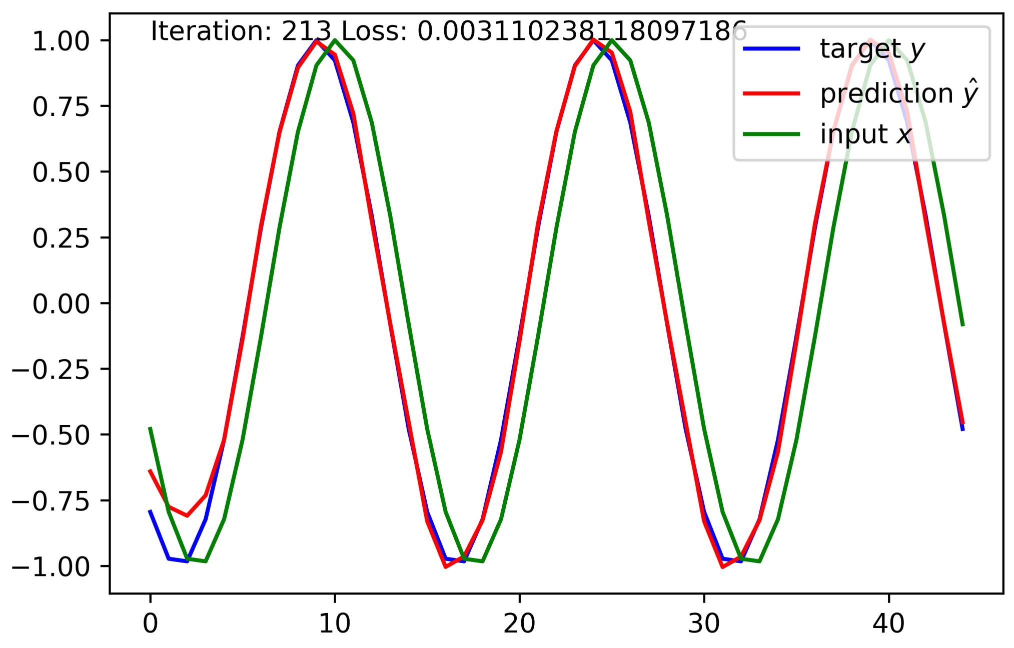

Figure 1: Learned Predictions of the LSTM model

```
    model = Model(inp, x, name=name)
    model.summary()
    model.compile(sgd, loss='mae')
    return model
```

```
Model: "Bidirectional"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
input_11 (InputLayer)        [(None, 19, 22)]          0         
_________________________________________________________________
bidirectional_1 (Bidirection (None, 128)               44544     
_________________________________________________________________
dense_10 (Dense)             (None, 1)                 129       
=================================================================
Total params: 44,673
Trainable params: 44,673
Non-trainable params: 0
_________________________________________________________________
Model: Bidirectional
Mean Error: 1.9972226952917502
```

4.2 Stacked LSTM

```
def get_model_stacked(name="stacked", shape=INPUT_SHAPE):
    inp = Input(shape)
    x = LSTM(64, return_sequences=True)(inp)
    x = LSTM(64, go_backwards=True)(x)
    x = Dense(1)(x)
    model = Model(inp, x, name=name)
    model.summary()
    model.compile(sgd, loss='mae')
    return model
```

```
Model: "stacked"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_12 (InputLayer)        [(None, 19, 22)]          0
_________________________________________________________________
lstm_11 (LSTM)               (None, 19, 64)            22272
_________________________________________________________________
lstm_12 (LSTM)               (None, 64)                33024
_________________________________________________________________
dense_11 (Dense)             (None, 1)                 65
=================================================================
Total params: 55,361
Trainable params: 55,361
Non-trainable params: 0
_________________________________________________________________
Model: stacked
Mean Error: 1.9909247734882982
```

The bidirectional layers outputs two concatenated LSTM layers, while the stacked layers output one single LSTM layer. In the stacked layer, the input of the backward layer is affected by the forward layer since it passes through the forward layer, but the two layers are independent in the bidirectional layer.

4.3 Convolutional

```
def get_model_convolutional(name="convolutional", shape=INPUT_SHAPE):
    inp = Input(shape)
    x = Conv1D(filters=32, kernel_size=3, strides=1)(inp)
    x = LSTM(64, return_sequences=False)(x)
    x = Dense(1)(x)

    model = Model(inp, x, name=name)
    model.summary()
    model.compile(sgd, loss='mae')
    return model
```

```
Model: "convolutional"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_13 (InputLayer)        [(None, 19, 22)]          0
_________________________________________________________________
conv1d (Conv1D)              (None, 17, 32)            2144
_________________________________________________________________
lstm_13 (LSTM)               (None, 64)                24832
_________________________________________________________________
dense_12 (Dense)             (None, 1)                 65
=================================================================
Total params: 27,041
Trainable params: 27,041
Non-trainable params: 0
_________________________________________________________________
Model: convolutional
Mean Error: 2.0492551517846094
```

A recurrent neural network uses every piece of the input data to make a prediction. A convolutional neural network extracts features from the input data a subset of the input data and uses the features to make a prediction. Not all input data may be necessary to make a good prediction therefore convolutional networks can be less computationally expensive.

4.4 Gradient Clipping

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

```
Model: RNN
Mean Error: 2.0829751688270535
```