ml_tutorials

Ather Abbas

Dec 02, 2022

CONTENTS:

CHAPTER

ONE

TUTORIALS

1.1 understanding Dense layer in Keras

This notebook describes dense layer or fully connected layer using tensorflow.

```
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Dense
```

```
def reset_seed(seed=313):
    tf.keras.backend.clear_session()
    tf.random.set_seed(seed)
    np.random.seed(seed)

np.set_printoptions(linewidth=100, suppress=True)

print(tf.__version__)
```

```
2.7.0
```

```
print(np.__version__)
```

```
1.21.6
```

set some global parameters

```
input_features = 2
batch_size = 10
dense_units = 5
```

define input to model

```
in_np = np.random.randint(0, 100, size=(batch_size,input_features))
print(in_np)
```

```
[[ 6  2]
 [ 8 60]
 [97 75]
```

(continues on next page)

(continued from previous page)

```
 [39 14]
 [ 4 80]
 [72 56]
 [69 54]
 [28  3]
 [65 53]
 [75 20]]
```

build a model consisting of single dense layer

```
reset_seed()


ins = Input(input_features, name='my_input')
out = Dense(dense_units, use_bias=False, name='my_output')(ins)
model = Model(inputs=ins, outputs=out)
```

```
out_np = model.predict(in_np)

print(out_np)
```

```
[[ -0.36387503   4.717352    -0.6198218   -2.749978     1.7211887 ]
 [-19.816442     9.008448   -15.283258    38.292225   -12.221191  ]
 [-20.268711    78.28703    -20.779022   -13.23278     17.0232    ]
 [ -2.702361    30.710205    -4.2809954  -17.143017    10.934539  ]
 [-26.766891     6.875136   -20.249329    55.738144   -18.769993  ]
 [-15.15605     58.125607   -15.506789    -9.580841    12.552248  ]
 [-14.636938    55.719513   -14.944724    -8.937692    11.944843  ]
 [  0.43734807  21.713993    -1.2955264  -17.468222     9.635738  ]
 [-14.506746    52.59042    -14.6155615   -6.8604274   10.712989  ]
 [ -2.862571    58.72981     -6.4870024  -38.033936    22.780798  ]]
```

```
print(out_np.shape)
```

```
(10, 5)
```

We can get all layers of model as list

```
print(model.layers)
```

```
[<keras.engine.input_layer.InputLayer object at 0x7f84a754aed0>, <keras.layers.core.
↪dense.Dense object at 0x7f8457e6de90>]
```

or a specific layer by its name

```
dense_layer = model.get_layer('my_output')
```

input to dense layer must be of the shape

```
print(dense_layer.input_shape)
```

```
(None, 2)
```

output from dense layer will be of the shape

```
print(dense_layer.output_shape)
```

```
(None, 5)
```

dense layer ususally has two variables i.e. weight/kernel and bias. As we did not use bias thus no bias is shown

```
print(dense_layer.weights)
```

```
[<tf.Variable 'my_output/kernel:0' shape=(2, 5) dtype=float32, numpy=
array([[ 0.0517453 ,  0.77041924, -0.0192523 , -0.7022766 ,  0.37126076],
       [-0.3371734 ,  0.04741824, -0.252154  ,  0.7318406 , -0.25318795]],␣
↪dtype=float32)>]
```

The shape of the dense weights is of the form *(input_size, units) dense_layer.weights* returns a list, the first variable of which kernel/weights. We can convert a numpy version of weights

```
dense_w = dense_layer.weights[0].numpy()
print(dense_w.shape)
```

```
(2, 5)
```

```
print(dense_w)
```

```
[[ 0.0517453   0.77041924 -0.0192523  -0.7022766   0.37126076]
 [-0.3371734   0.04741824 -0.252154    0.7318406  -0.25318795]]
```

The output from our model consisting of a single dense layer is simply the matrix multiplication between input and weight matrix as can be verified from below.

```
np.matmul(in_np, dense_w)
```

```
array([[ -0.36387503,   4.71735191,  -0.61982179,  -2.7499783 ,   1.72118866],
       [-19.81644177,   9.00844812, -15.28325796,  38.29222393, -12.22119117],
       [-20.26871151,  78.28703403, -20.77902257, -13.2327832 ,  17.02319735],
       [ -2.70236111,  30.71020567,  -4.28099561, -17.14301836,  10.93453836],
       [-26.766891  ,   6.8751359 , -20.24932861,  55.73814249, -18.76999331],
       [-15.15604925,  58.12560654, -15.50678921,  -9.58084011,  12.55224943],
       [-14.63693833,  55.71951234, -14.94472432,  -8.93769157,  11.94484305],
       [  0.43734807,  21.71399343,  -1.29552639, -17.46822262,   9.63573748],
       [-14.50674611,  52.59041715, -14.61556113,  -6.86042583,  10.71298796],
       [ -2.86257088,  58.72980773,  -6.48700237, -38.03393185,  22.78079808]])
```

compare above output from the model's output which was obtained earlier.

1.1.1 Using Bias

By default the *Dense* layer in tensorflow uses bias as well.

```
reset_seed()
tf.keras.backend.clear_session()

ins = Input(input_features, name='my_input')
out = Dense(5, use_bias=True,  name='my_output')(ins)
model = Model(inputs=ins, outputs=out)
```

```
out_np = model.predict(in_np)
print(out_np.shape)
print(out_np)
```

```
(10, 5)
[[ -0.36387503   4.717352    -0.6198218   -2.749978     1.7211887 ]
 [-19.816442     9.008448   -15.283258    38.292225   -12.221191  ]
 [-20.268711    78.28703    -20.779022   -13.23278     17.0232    ]
 [ -2.702361    30.710205    -4.2809954  -17.143017    10.934539  ]
 [-26.766891     6.875136   -20.249329    55.738144   -18.769993  ]
 [-15.15605     58.125607   -15.506789    -9.580841    12.552248  ]
 [-14.636938    55.719513   -14.944724    -8.937692    11.944843  ]
 [  0.43734807  21.713993    -1.2955264  -17.468222     9.635738  ]
 [-14.506746    52.59042    -14.6155615   -6.8604274   10.712989  ]
 [ -2.862571    58.72981     -6.4870024  -38.033936    22.780798  ]]
```

```
dense_layer = model.get_layer('my_output')
print(dense_layer.weights)
```

```
[<tf.Variable 'my_output/kernel:0' shape=(2, 5) dtype=float32, numpy=
array([[ 0.0517453 ,  0.77041924, -0.0192523 , -0.7022766 ,  0.37126076],
       [-0.3371734 ,  0.04741824, -0.252154  ,  0.7318406 , -0.25318795]],␣
↪dtype=float32)>, <tf.Variable 'my_output/bias:0' shape=(5,) dtype=float32,␣
↪numpy=array([0., 0., 0., 0., 0.], dtype=float32)>]
```

The bias vector above was all zeros thus had no effect on model output as the equation for dense layer becomes $$ y = Ax + b$$ We can initialize bias vector with ones and see the output

```
reset_seed()

ins = Input(input_features, name='my_input')
out = Dense(dense_units, use_bias=True, bias_initializer='ones', name='my_output')(ins)
model = Model(inputs=ins, outputs=out)
```

```
out_np = model.predict(in_np)
print(out_np.shape)
print(out_np)
```

```
(10, 5)
[[  0.63612497   5.717352     0.3801782   -1.7499781    2.7211885 ]
 [-18.816442    10.008448   -14.283258    39.292225   -11.221191  ]
```

(continues on next page)

(continued from previous page)

```
 [-19.268711    79.28703    -19.779022   -12.23278     18.0232    ]
 [ -1.7023611   31.710205    -3.2809954  -16.143017    11.934539  ]
 [-25.766891     7.875136   -19.249329    56.738144   -17.769993  ]
 [-14.15605     59.125607   -14.506789    -8.580841    13.552248  ]
 [-13.636938    56.719513   -13.944724    -7.9376917   12.944843  ]
 [  1.4373481   22.713993    -0.2955264  -16.468222    10.635738  ]
 [-13.506746    53.59042    -13.6155615   -5.8604274   11.712989  ]
 [ -1.862571    59.72981     -5.4870024  -37.033936    23.780798  ]]
```

```
dense_layer = model.get_layer('my_output')
print(dense_layer.weights)
```

```
[<tf.Variable 'my_output/kernel:0' shape=(2, 5) dtype=float32, numpy=
array([[ 0.0517453 ,  0.77041924, -0.0192523 , -0.7022766 ,  0.37126076],
       [-0.3371734 ,  0.04741824, -0.252154  ,  0.7318406 , -0.25318795]],␣
↪dtype=float32)>, <tf.Variable 'my_output/bias:0' shape=(5,) dtype=float32,␣
↪numpy=array([1., 1., 1., 1., 1.], dtype=float32)>]
```

We can verify that the model's output is obtained following the equation we wrote above.

```
dense_layer = model.get_layer('my_output')
dense_w = dense_layer.weights[0].numpy()
np.matmul(in_np, dense_w) + np.ones(dense_units)
```

```
array([[  0.63612497,   5.71735191,   0.38017821,  -1.7499783 ,   2.72118866],
       [-18.81644177,  10.00844812, -14.28325796,  39.29222393, -11.22119117],
       [-19.26871151,  79.28703403, -19.77902257, -12.2327832 ,  18.02319735],
       [ -1.70236111,  31.71020567,  -3.28099561, -16.14301836,  11.93453836],
       [-25.766891  ,   7.8751359 , -19.24932861,  56.73814249, -17.76999331],
       [-14.15604925,  59.12560654, -14.50678921,  -8.58084011,  13.55224943],
       [-13.63693833,  56.71951234, -13.94472432,  -7.93769157,  12.94484305],
       [  1.43734807,  22.71399343,  -0.29552639, -16.46822262,  10.63573748],
       [-13.50674611,  53.59041715, -13.61556113,  -5.86042583,  11.71298796],
       [ -1.86257088,  59.72980773,  -5.48700237, -37.03393185,  23.78079808]])
```

1.1.2 using *activation* function

We can add non-linearity to the output of dense layer by making use of *activation* keyword argument. A common *activation* function is *relu* which makes all the values below 0 as zero. In this case the equation of dense layer will become $$ y = alpha (Ax + b) $$ Where $alpha$ is the non-linearity applied.

```
reset_seed()

ins = Input(input_features, name='my_input')
out = Dense(dense_units, use_bias=True, bias_initializer='ones',
            activation='relu', name='my_output')(ins)
model = Model(inputs=ins, outputs=out)

out_np = model.predict(in_np)
print(out_np.shape)
print(out_np)
```

```
(10, 5)
[[ 0.63612497  5.717352    0.3801782   0.          2.7211885 ]
 [ 0.         10.008448    0.         39.292225    0.        ]
 [ 0.         79.28703     0.          0.         18.0232    ]
 [ 0.         31.710205    0.          0.         11.934539  ]
 [ 0.          7.875136    0.         56.738144    0.        ]
 [ 0.         59.125607    0.          0.         13.552248  ]
 [ 0.         56.719513    0.          0.         12.944843  ]
 [ 1.4373481  22.713993    0.          0.         10.635738  ]
 [ 0.         53.59042     0.          0.         11.712989  ]
 [ 0.         59.72981     0.          0.         23.780798  ]]
```

We can again verify that the above output from dense layer follows the equation that we wrote above.

```
def relu(X):
   return np.maximum(0,X)


dense_layer = model.get_layer('my_output')
dense_w = dense_layer.weights[0].numpy()
relu(np.matmul(in_np, dense_w) + np.ones(dense_units))
```

```
array([[ 0.63612497,  5.71735191,  0.38017821,  0.        ,  2.72118866],
       [ 0.        , 10.00844812,  0.        , 39.29222393,  0.        ],
       [ 0.        , 79.28703403,  0.        ,  0.        , 18.02319735],
       [ 0.        , 31.71020567,  0.        ,  0.        , 11.93453836],
       [ 0.        ,  7.8751359 ,  0.        , 56.73814249,  0.        ],
       [ 0.        , 59.12560654,  0.        ,  0.        , 13.55224943],
       [ 0.        , 56.71951234,  0.        ,  0.        , 12.94484305],
       [ 1.43734807, 22.71399343,  0.        ,  0.        , 10.63573748],
       [ 0.        , 53.59041715,  0.        ,  0.        , 11.71298796],
       [ 0.        , 59.72980773,  0.        ,  0.        , 23.78079808]])
```

1.1.3 customizing weights

we can set the weights and bias of dense layer to values of our choice. This is useful for example when we want to initialize the weights/bias with the values that we already have.

```
custom_dense_weights = np.array([[1, 2, 3 , 4,  5],
                                 [6, 7, 8 , 9 , 10]], dtype=np.float32)
custom_bias = np.array([0., 0., 0., 0., 0.])

reset_seed()

ins = Input(input_features, name='my_input')

dense_lyr = Dense(dense_units, use_bias=True, bias_initializer='ones', name='my_output')
out = dense_lyr(ins)

model = Model(inputs=ins, outputs=out)

dense_lyr.set_weights([custom_dense_weights, custom_bias])
```

The method *set_weights* must be called after initializing *Model* class. The input to *set_weights* is a list containing both weight matrix and bias vector respectively.

```
out_np = model.predict(in_np)
print(out_np.shape)
print(out_np)
```

```
WARNING:tensorflow:5 out of the last 5 calls to <function Model.make_predict_function.
↪<locals>.predict_function at 0x7f8454c58d40> triggered tf.function retracing. Tracing␣
↪is expensive and the excessive number of tracings could be due to (1) creating @tf.
↪function repeatedly in a loop, (2) passing tensors with different shapes, (3) passing␣
↪Python objects instead of tensors. For (1), please define your @tf.function outside of␣
↪the loop. For (2), @tf.function has experimental_relax_shapes=True option that relaxes␣
↪argument shapes that can avoid unnecessary retracing. For (3), please refer to https://
↪www.tensorflow.org/guide/function#controlling_retracing and https://www.tensorflow.org/
↪api_docs/python/tf/function for  more details.
(10, 5)
[[  18.   26.   34.   42.   50.]
 [ 368.  436.  504.  572.  640.]
 [ 547.  719.  891. 1063. 1235.]
 [ 123.  176.  229.  282.  335.]
 [ 484.  568.  652.  736.  820.]
 [ 408.  536.  664.  792.  920.]
 [ 393.  516.  639.  762.  885.]
 [  46.   77.  108.  139.  170.]
 [ 383.  501.  619.  737.  855.]
 [ 195.  290.  385.  480.  575.]]
```

```
dense_layer = model.get_layer('my_output')
dense_w = dense_layer.weights[0].numpy()
print(dense_w)
```

```
[[ 1.  2.  3.  4.  5.]
 [ 6.  7.  8.  9. 10.]]
```

Verify that the output from dense is just matrix multiplication.

```
np.matmul(in_np, custom_dense_weights) + np.zeros(dense_units)
```

```
array([[  18.,   26.,   34.,   42.,   50.],
       [ 368.,  436.,  504.,  572.,  640.],
       [ 547.,  719.,  891., 1063., 1235.],
       [ 123.,  176.,  229.,  282.,  335.],
       [ 484.,  568.,  652.,  736.,  820.],
       [ 408.,  536.,  664.,  792.,  920.],
       [ 393.,  516.,  639.,  762.,  885.],
       [  46.,   77.,  108.,  139.,  170.],
       [ 383.,  501.,  619.,  737.,  855.],
       [ 195.,  290.,  385.,  480.,  575.]])
```

1.1.4 Reducing Dimensions

Dense layer can be used to reduce last dimension of incoming input. In following the size is reduced from *(10, 20, 30)* ==> *(10, 20, 1)*

```
input_shape = 20, 30
in_np = np.random.randint(0, 100, size=(batch_size,*input_shape))

reset_seed()


ins = Input(input_shape, name='my_input')
out = Dense(1, use_bias=False, name='my_output')(ins)
model = Model(inputs=ins, outputs=out)
out_np = model.predict(in_np)
print('input shape: {}\n output shape: {}'.format(in_np.shape, out_np.shape))
```

```
WARNING:tensorflow:6 out of the last 6 calls to <function Model.make_predict_function.
↪<locals>.predict_function at 0x7f8454c76320> triggered tf.function retracing. Tracing␣
↪is expensive and the excessive number of tracings could be due to (1) creating @tf.
↪function repeatedly in a loop, (2) passing tensors with different shapes, (3) passing␣
↪Python objects instead of tensors. For (1), please define your @tf.function outside of␣
↪the loop. For (2), @tf.function has experimental_relax_shapes=True option that relaxes␣
↪argument shapes that can avoid unnecessary retracing. For (3), please refer to https://
↪www.tensorflow.org/guide/function#controlling_retracing and https://www.tensorflow.org/
↪api_docs/python/tf/function for  more details.
input shape: (10, 20, 30)
 output shape: (10, 20, 1)
```

Total running time of the script: (0 minutes 2.117 seconds)

1.2 Implementing LSTM in numpy from scratch

The purpose of this notebook is to illustrate how to build an LSTM from scratch in numpy.

```
import numpy as np
np.__version__
```

```
'1.21.6'
```

```
import tensorflow as tf
tf.__version__
```

```
'2.7.0'
```

```
from tensorflow.keras.layers import Input
from tensorflow.keras.layers import LSTM as KLSTM
from tensorflow.keras.models import Model
from tensorflow.keras.initializers import Orthogonal, GlorotUniform, Zeros
```

(continues on next page)

(continued from previous page)

```
assert tf.__version__ > "2.1", "results are not reproducible with Tensorflow below 2"
```

experiment setup

```
num_inputs = 3  # number of input features
lstm_units = 32
lookback_steps = 5  # also known as time_steps or sequence length
num_samples = 10  # length of x,y
```

in order to make the results comparable between tensorflow ansd numpy we use same weights for tensorflow inmplementation and numpy implementation

```
k_init = GlorotUniform(seed=313)
k_vals = k_init(shape=(num_inputs, lstm_units*4))

rec_init = Orthogonal(seed=313)
rec_vals = rec_init(shape=(lstm_units, lstm_units*4))

b_init = Zeros()
b_vals = b_init(lstm_units*4)

weights = [k_vals, rec_vals, b_vals]
```

1.2.1 Keras version of LSTM

```
# check the results of forward pass of original LSTM of Keras

inp = Input(shape=(lookback_steps, num_inputs))
lstm_lyr = KLSTM(lstm_units)
out = lstm_lyr(inp)

lstm_lyr.set_weights(weights)

model = Model(inputs=inp, outputs=out)

xx = np.random.random((num_samples, lookback_steps, num_inputs))

lstm_out_tf = model.predict(x=xx)
```

1.2.2 numpy version of LSTM

```
class LSTMNP(object):
    """vanilla LSTM in pure numpy
    Only forward loop"""
    def __init__(
            self,
            units:int,
            return_sequences:bool = False,
```

(continues on next page)

(continued from previous page)

```
            return_states:bool = False,
            time_major:bool = False
    ):
        self.units = units
        self.return_sequences = return_sequences
        self.return_states = return_states
        self.time_major = time_major

        self.kernel = k_vals.numpy()
        self.rec_kernel = rec_vals.numpy()
        self.bias = b_vals.numpy()


    def __call__(self, inputs, initial_state=None):
        # if not time_major original inputs have shape (batch_size, lookback_steps, num_
↪inputs)
        # otherwise inputs will have shape (lookback_steps, batch_size, num_inputs)

        if not self.time_major:
            inputs = np.moveaxis(inputs, [0, 1], [1, 0])

        # inputs have shape (lookback_steps, batch_size, num_inputs)
        lookback_steps, bs, ins = inputs.shape

        if initial_state is None:
            h_state = np.zeros((bs, self.units))
            c_state = np.zeros((bs, self.units))
        else:
            assert len(initial_state) == 2
            h_state, c_state = initial_state

        h_states = []
        c_states = []

        for step in range(lookback_steps):

            h_state, c_state = self.cell(inputs[step], h_state, c_state)

            h_states.append(h_state)
            c_states.append(c_state)

        h_states = np.stack(h_states)
        c_states = np.stack(c_states)

        if not self.time_major:
            h_states = np.moveaxis(h_states, [0, 1], [1, 0])
            c_states = np.moveaxis(c_states, [0, 1], [1, 0])

        o = h_states[:, -1]
        if self.return_sequences:
            o = h_states
```

(continues on next page)

(continued from previous page)

```
        if self.return_states:
            return o, c_states
        return o

    def cell(self, xt, ht, ct):
        """implements logic of LSTM"""

        # input gate
        k_i = self.kernel[:, :self.units]
        rk_i = self.rec_kernel[:, :self.units]
        b_i = self.bias[:self.units]
        i_t = self.sigmoid(np.dot(xt, k_i) + np.dot(ht, rk_i) + b_i)

        # forget gate
        k_f = self.kernel[:, self.units:self.units * 2]
        rk_f = self.rec_kernel[:, self.units:self.units * 2]
        b_f = self.bias[self.units:self.units * 2]
        ft = self.sigmoid(np.dot(xt, k_f) + np.dot(ht, rk_f) + b_f)

        # candidate cell state
        k_c = self.kernel[:, self.units * 2:self.units * 3]
        rk_c = self.rec_kernel[:, self.units * 2:self.units * 3]
        b_c = self.bias[self.units * 2:self.units * 3]
        c_t = self.tanh(np.dot(xt, k_c) + np.dot(ht, rk_c) + b_c)

        # cell state
        ct = ft * ct + i_t * c_t

        # output gate
        k_o = self.kernel[:, self.units * 3:]
        rk_o = self.rec_kernel[:, self.units * 3:]
        b_o = self.bias[self.units * 3:]
        ot = self.sigmoid(np.dot(xt, k_o) + np.dot(ht, rk_o) + b_o)

        # hidden state
        ht = ot * self.tanh(ct)

        return ht, ct

    @staticmethod
    def tanh(x):
        return np.tanh(x)

    @staticmethod
    def sigmoid(x):
        return 1. / (1 + np.exp(-x))
```

```
nplstm = LSTMNP(lstm_units)
lstm_out_np = nplstm(xx)
```

we can make sure that the results of numpy implementation and implementation of Tensorflow are exactly same

```
print(np.allclose(lstm_out_tf, lstm_out_np))
```

```
True
```

Total running time of the script: (0 minutes 0.523 seconds)

1.3 understanding Input/output of LSTM

The purpose of this notebook to determine the input and output shapes of LSTM in keras/tensorflow. It also shows how the output changes when we use different options such as `return_sequences` and `return_state` arguments in LSTM/RNN layers of tensorflow/keras.

```
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Model
from tensorflow.keras.layers import MaxPooling1D, Flatten, Conv1D
from tensorflow.keras.layers import Input, LSTM, Reshape, TimeDistributed

# to suppress scientific notation while printing arrays
np.set_printoptions(suppress=True)

def reset_graph(seed=313):
    tf.compat.v1.reset_default_graph()
    tf.compat.v1.set_random_seed(seed)
    np.random.seed(seed)

tf.__version__
```

```
'2.7.0'
```

```
seq_len = 9
in_features = 3
batch_size = 2
units = 5

# define input data
data = np.random.normal(0,1, size=(batch_size, seq_len, in_features))
print('input shape is', data.shape)
```

```
input shape is (2, 9, 3)
```

```
reset_graph()
```

1.3.1 Input to LSTM

```
# The input to LSTM is 3D where each dimension is expected to have following meaning
# (batch_size, sequence_length, num_inputs)
# the batch_size determines the number of samples, sequence_legth determines the length
# of historical/temporal data used by LSTM and num_inputs is the number of input features

# define model
inputs1 = Input(shape=(seq_len, in_features))
lstm1 = LSTM(units)(inputs1)
model = Model(inputs=inputs1, outputs=lstm1)
model.inputs
```

```
[<KerasTensor: shape=(None, 9, 3) dtype=float32 (created by layer 'input_2')>]
```

1.3.2 Output from LSTM

```
# In Keras, the output from LSTM is 2D and each dimension has following meaning
# (batch_size, units)
# the units here represents the number of units/neuron of LSTM layer.

# check output
output = model.predict(data)
print('output shape is ', output.shape)
print(output)
```

```
output shape is  (2, 5)
[[-0.04311746 -0.04708175  0.11244525  0.09445497  0.08160033]
 [ 0.22174549  0.23136306 -0.1471001   0.04506844 -0.0963508 ]]
```

1.3.3 Return Sequence

If we use `return_sequences=True`, we can get hidden state which is also output, at each time step instead of just one final output.

```
reset_graph()

print('input shape is', data.shape)

# define model
inputs1 = Input(shape=(seq_len, in_features))
lstm1 = LSTM(units, return_sequences=True)(inputs1)
model = Model(inputs=inputs1, outputs=lstm1)

# check output
output = model.predict(data)
print('output shape is ', output.shape)
print(output)
```

```
input shape is (2, 9, 3)
output shape is  (2, 9, 5)
[[[ 0.23949696  0.23758332  0.0201166  -0.07562752  0.14458913]
  [ 0.20123877  0.19533847  0.04180209 -0.12905313  0.20505369]
  [ 0.06623977  0.09107485  0.02961113 -0.06149743  0.07921001]
  [ 0.103291    0.14202026 -0.10353918 -0.13593747 -0.01541394]
  [ 0.11871371  0.11363701  0.01490535 -0.01338429  0.09110813]
  [ 0.18314067  0.17522626  0.04663869 -0.05388878  0.18176244]
  [ 0.31485227  0.24940978  0.0693886  -0.03106552  0.25046384]
  [ 0.17771643  0.09009738  0.16493434  0.06166327  0.21880664]
  [-0.04311746 -0.04708175  0.11244525  0.09445497  0.08160033]]

 [[ 0.0236822   0.057854    0.05342087 -0.10365748  0.14504817]
  [-0.03983979  0.04184275  0.13498983  0.14183497  0.11871135]
  [-0.08096419  0.02722256  0.16430669  0.19353093  0.18122804]
  [-0.10457274 -0.09090691  0.05876469  0.26642254 -0.02051181]
  [ 0.07231079  0.07811436  0.06489968  0.07280337  0.08751098]
  [-0.02732764  0.00174761  0.04222624 -0.02587408  0.02410888]
  [ 0.02454332  0.01909897 -0.09221498 -0.07524213 -0.09897806]
  [ 0.22740148  0.31498346 -0.19642149 -0.16686526 -0.2563934 ]
  [ 0.22174549  0.23136306 -0.1471001   0.04506844 -0.0963508 ]]]
```

1.3.4 Return States

If we use `return_state=True`, it will give final hidden state/output plus the cell state as well

```
reset_graph()

# define model
inputs1 = Input(shape=(seq_len, in_features))
lstm1, state_h, state_c = LSTM(units, return_state=True)(inputs1)
model = Model(inputs=inputs1, outputs=[lstm1, state_h, state_c])

# check output
_h, h, c = model.predict(data)
print('_h: shape {} values \n {}\n'.format(_h.shape, _h))
print('h: shape {} values \n {}\n'.format(h.shape, h))
print('c: shape {} values \n {}'.format(c.shape, c))
```

```
_h: shape (2, 5) values
 [[-0.04311746 -0.04708175  0.11244525  0.09445497  0.08160033]
 [ 0.22174549  0.23136306 -0.1471001   0.04506844 -0.0963508 ]]

h: shape (2, 5) values
 [[-0.04311746 -0.04708175  0.11244525  0.09445497  0.08160033]
 [ 0.22174549  0.23136306 -0.1471001   0.04506844 -0.0963508 ]]

c: shape (2, 5) values
 [[-0.0884207  -0.10446949  0.1710459   0.17895043  0.24443825]
 [ 0.3913621   0.40256596 -0.38461903  0.08493438 -0.22778362]]
```

using both at same time We can use both `return_sequences` and `return_states` at same time as well.

```
reset_graph()

# define model
inputs1 = Input(shape=(seq_len, in_features))
lstm1, state_h, state_c = LSTM(units, return_state=True, return_sequences=True)(inputs1)
model = Model(inputs=inputs1, outputs=[lstm1, state_h, state_c])

# check output
_h, h, c = model.predict(data)
print('_h: shape {} values \n {}\n'.format(_h.shape, _h))
print('h: shape {} values \n {}\n'.format(h.shape, h))
print('c: shape {} values \n {}'.format(c.shape, c))
```

```
_h: shape (2, 9, 5) values
 [[[ 0.23949696  0.23758332  0.0201166  -0.07562752  0.14458913]
  [ 0.20123877  0.19533847  0.04180209 -0.12905313  0.20505369]
  [ 0.06623977  0.09107485  0.02961113 -0.06149743  0.07921001]
  [ 0.103291    0.14202026 -0.10353918 -0.13593747 -0.01541394]
  [ 0.11871371  0.11363701  0.01490535 -0.01338429  0.09110813]
  [ 0.18314067  0.17522626  0.04663869 -0.05388878  0.18176244]
  [ 0.31485227  0.24940978  0.0693886  -0.03106552  0.25046384]
  [ 0.17771643  0.09009738  0.16493434  0.06166327  0.21880664]
  [-0.04311746 -0.04708175  0.11244525  0.09445497  0.08160033]]

 [[ 0.0236822   0.057854    0.05342087 -0.10365748  0.14504817]
  [-0.03983979  0.04184275  0.13498983  0.14183497  0.11871135]
  [-0.08096419  0.02722256  0.16430669  0.19353093  0.18122804]
  [-0.10457274 -0.09090691  0.05876469  0.26642254 -0.02051181]
  [ 0.07231079  0.07811436  0.06489968  0.07280337  0.08751098]
  [-0.02732764  0.00174761  0.04222624 -0.02587408  0.02410888]
  [ 0.02454332  0.01909897 -0.09221498 -0.07524213 -0.09897806]
  [ 0.22740148  0.31498346 -0.19642149 -0.16686526 -0.2563934 ]
  [ 0.22174549  0.23136306 -0.1471001   0.04506844 -0.0963508 ]]]

h: shape (2, 5) values
 [[-0.04311746 -0.04708175  0.11244525  0.09445497  0.08160033]
 [ 0.22174549  0.23136306 -0.1471001   0.04506844 -0.0963508 ]]

c: shape (2, 5) values
 [[-0.0884207  -0.10446949  0.1710459   0.17895043  0.24443825]
 [ 0.3913621   0.40256596 -0.38461903  0.08493438 -0.22778362]]
```

1.3.5 time major

By `time_major` we mean that the last dimention i.e. 3rd dimension represents time and the second last represents input features. Thus the 3D input to lstm will become (batch_size, num_inputs, sequence_length)

```
reset_graph()

# define model
inputs1 = Input(shape=(in_features, seq_len))
```

(continues on next page)

(continued from previous page)

```
lstm1 = LSTM(units, time_major=True)(inputs1)
model = Model(inputs=inputs1, outputs=[lstm1])
model.inputs
```

```
[<KerasTensor: shape=(None, 3, 9) dtype=float32 (created by layer 'input_6')>]
```

```
# we will have to shift the dimensions of numpy array to make it time_major
# check output
time_major_data = np.moveaxis(data, [1,2], [2,1])
time_major_data.shape
```

```
(2, 3, 9)
```

```
h = model.predict(time_major_data)
print('h: shape {} values \n {}\n'.format(h.shape, h))
```

```
h: shape (3, 5) values
 [[ 0.0856159   0.06631077 -0.43855685  0.1004677  -0.40924817]
 [ 0.02948599  0.02146549  0.01565967 -0.10389965  0.27761555]
 [ 0.09459803  0.14054263  0.1562092  -0.11277693 -0.12558709]]
```

1.3.6 CNN -> LSTM

We can append LSTM with any other layer. The only requirement is that the output from that layer should match the input requirement of LSTM i.e. the output from the layer that we want to add before LSTM should be 3D of shape (batch_size, num_inputs, seq_length)

```
reset_graph()

# define model
inputs = Input(shape=(seq_len, in_features))
cnn = Conv1D(filters=2, kernel_size=2, padding="same")(inputs)
max_pool = MaxPooling1D(padding="same")(cnn)
max_pool
```

```
<KerasTensor: shape=(None, 5, 2) dtype=float32 (created by layer 'max_pooling1d')>
```

as the shape of `max_pool` tensor matches the input requirement of LSTM we can combine it with LSTM

```
h = LSTM(units)(max_pool)
model = Model(inputs=inputs, outputs=h)
model.summary()
```

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_7 (InputLayer)        [(None, 9, 3)]            0
```

(continues on next page)

(continued from previous page)

```
 conv1d (Conv1D)             (None, 9, 2)              14

 max_pooling1d (MaxPooling1D  (None, 5, 2)             0
 )

 lstm (LSTM)                 (None, 5)                 160

=================================================================
Total params: 174
Trainable params: 174
Non-trainable params: 0
_________________________________________________________________
```

However, this is not how CNN is comined with LSTM at its start. The purpose is usually to break the sequence length into small sub-sequences and then apply the **same** CNN on those sub-sequences. We can achieve this as following

```
sub_sequences = 3

reset_graph()
# define model
inputs = Input(shape=(seq_len, in_features))
time_steps = seq_len // sub_sequences
reshape = Reshape(target_shape=(sub_sequences, time_steps, in_features))(inputs)
cnn = TimeDistributed(Conv1D(filters=2, kernel_size=2, padding="same"))(reshape)
max_pool = TimeDistributed(MaxPooling1D(padding="same"))(cnn)
flatten = TimeDistributed(Flatten())(max_pool)
flatten
```

```
<KerasTensor: shape=(None, 3, 4) dtype=float32 (created by layer 'time_distributed_2')>
```

the shape of `flatten` tensor again matches the input requirements of LSTM so we can again attach LSTM after it.

```
h = LSTM(units)(flatten)
model = Model(inputs=inputs, outputs=h)
model.summary()
```

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_8 (InputLayer)        [(None, 9, 3)]            0

 reshape (Reshape)           (None, 3, 3, 3)           0

 time_distributed (TimeDistr  (None, 3, 3, 2)          14
 ibuted)

 time_distributed_1 (TimeDis  (None, 3, 2, 2)          0
 tributed)

 time_distributed_2 (TimeDis  (None, 3, 4)             0
 tributed)
```

(continues on next page)

(continued from previous page)

```
 lstm (LSTM)                 (None, 5)                 200

=================================================================
Total params: 214
Trainable params: 214
Non-trainable params: 0
_________________________________________________________________
```

1.3.7 LSTM -> 1D CNN

We can put 1d cnn at the end of LSTM to further extract some features from LSTM output.

```
reset_graph()

print('input shape is', data.shape)

# define model
inputs = Input(shape=(seq_len, in_features))
lstm_layer = LSTM(units, return_sequences=True)
lstm_outputs = lstm_layer(inputs)
print('lstm output: ', lstm_outputs.shape)

conv1 = Conv1D(filters=64, kernel_size=2, activation='relu', input_shape=(seq_len,␣
↪units))(lstm_outputs)
print('conv output: ', conv1.shape)

max1d1 = MaxPooling1D(pool_size=2)(conv1)
print('max pool output: ', max1d1.shape)

flat1 = Flatten()(max1d1)
print('flatten output: ', flat1.shape)

model = Model(inputs=inputs, outputs=flat1)

# check output
output = model.predict(data)
print('output shape: ', output.shape)
```

```
input shape is (2, 9, 3)
lstm output:  (None, 9, 5)
conv output:  (None, 8, 64)
max pool output:  (None, 4, 64)
flatten output:  (None, 256)
output shape:  (2, 256)
```

The output from LSTM/RNN looks roughly as below. $$ h_t = tanh(b + Wh_{t-1} + UX_t) $$

weights of our input against every neuron in LSTM

```
print('kernel U: ', lstm_layer.get_weights()[0].shape)
```

```
kernel U:  (3, 20)
```

weights of our hidden state a.k.a the output of LSTM in the previous timestep (t-1) against every neuron in LSTM

```
print('recurrent kernel, W: ', lstm_layer.get_weights()[1].shape)
```

```
recurrent kernel, W:  (5, 20)
```

```
print('bias: ', lstm_layer.get_weights()[2].shape)
```

```
bias:  (20,)
```

This post is inspired from Jason Brownlee's [page](https://machinelearningmastery.com/return-sequences-and-return-states-for-lstms-in-keras/)

Total running time of the script: (0 minutes 3.516 seconds)

1.4 ANN in numpy

In this tutorial, we will build and train a multi layer perceptron from scratch in numpy. We will implement the forward pass, backward pass, weight update so that we can get the idea what actually happens under the hood when we train neural networks for a particular data. The main purpose is to understand the forward and backward propagation and its implementation.

```
import math
import numpy as np
```

1.4.1 data preparation

The purpose of data preparation step in a supervised learning task is to divide our data into input/output pairs. The number of input/output paris should be equal. We can call one pair of input and its corresponding output as `example`. We feed many such examples to the neural network to make it learn the relationship between inputs and outputs.

```
from ai4water.datasets import busan_beach
data = busan_beach()
print(data.shape)
# 1446, 14
```

1.4.2 splitting

The length of the data is above 1400. However, not the target column consists of many missing values. This will reduce the number of examples that we will finally have at our disposal. Furthermore, we will divide the total number of examples into training and validation sets. We will use 70% of the examples for training and 30% for the validation. The splitting is performed randomly. The value of seed 2809 is for reproducibility purpose.

```
from ai4water.preprocessing import DataSet
dataset = DataSet(data, val_fraction=0.0, seed=2809)
X_train, y_train = dataset.training_data()
X_val, y_val = dataset.test_data()
```

batch generation

```
def batch_generator(X, Y, size=32):

    for ii in range(0, len(X), size):
        X_batch, y_batch = X[ii:ii + size], Y[ii:ii + size]

        yield X_batch, y_batch
```

The purpose of `batch_generator` function is to divided out data (x,y) into batches. The size of each batch is determined by `size` argument.

hyperparameters Next we define hyperparameters of out feed forward neural network or multi-layer perceptron. The hyeparameters are those parameters which determine how the parameter are going to be estimated. These `parameters` here mean weights and biases whose values are calibrated/optimized due the training process.

```
lr = 0.01
epochs = 1000
l1_neurons = 10
l2_neurons = 5
l3_neurons = 1
```

`lr` is the learning rate. It determines the jump in the values of weights and biases which we make at each parameter update step. The parameter update step can be performed either after feeding the whole data to the neural network or feeding a single example to the network or a batch of examples to the network. In this example, we will update the parameters after each batch. The hyperparameter `epochs` determine, how many times we want to show our whole data to the neural network. The values of neurons determine the size of learnable parameters i.e., weights and biases in each layer. The larger the neurons, the bigger is the size of weights and biases matrices, the larger is the learning capacity of the network, the higher is the computation cost. The number of neurons in the last layer must match the number of target/output variables. In our case we have just one target variable.

1.4.3 weights and biases

Next, we initialize our weights and biases with random numbers. We will have three layers, 2 hidden and 1 output. Each of these layers will have two learnable parameters i.e. weights and biases. The size of the weights and biases in a layer depends upon the size of inputs that it is receiving and a user defined parameter i.e. `neurons` which is also calle `units`.

```
from numpy.random import default_rng

rng_l1 = default_rng(313)
rng_l2 = default_rng(313)
rng_l3 = default_rng(313)
w1 = rng_l1.standard_normal((dataset.num_ins, l1_neurons))
b1 = rng_l1.standard_normal((1, l1_neurons))
w2 = rng_l2.standard_normal((w1.shape[1], l2_neurons))
b2 = rng_l2.standard_normal((1, l2_neurons))
w3 = rng_l3.standard_normal((w2.shape[1], l3_neurons))
b3 = rng_l3.standard_normal((1, l3_neurons))
```

The `default_rng` is used to generate random numbers with reproducibility.

1.4.4 Forward pass

The forward pass consists of set of calculations which are performed on the input until we get the output. In other words, it is the modification of input through the application of fully connected layers.

```
for e in range(epochs):

    for batch_x, batch_y in batch_generator(X_train, y_train):
```

Each epoch can consist of multiple batches. The actual number of batches in an epoch depends upon the number of examples and batch size. If we have 100 examples in our dataset and the batch size is 25, then we will have 4 batches. In this case the inner loop will have 4 iterations.

1.4.5 hidden layer

$$l1_{out} = sigmoid(inputs * w1 + b1)$$

```
def sigmoid(inputs):
    return 1.0 / (1.0 + np.exp(-1.0 * inputs))

l1_out = np.dot(batch_x, w1) + b1
sig1_out = sigmoid(l1_out)
```

1.4.6 second hidden layer

The mathematics of second hidden layer is very much similar to first hidden layer. However, here we use the outputs from the first layer as inputs.

$$l2_{out} = sigmoid(l1_out * w2 + b2)$$

```
l2_out = np.dot(sig1_out, w2) + b2
sig2_out = sigmoid(l2_out)
```

1.4.7 output layer

The output layer is also similar to other hidden layers, except that we are not applying any activation function here. Here we are performing a regression task. Had it been a classification problem, we would have been interested in using a relevant activation function here.

$$l3_{out} = l2_out * w3 + b3$$

```
l3_out = np.dot(sig2_out, w3) + b3
```

1.4.8 loss calculation

This step evaluate the performance of our model with current state of parameters. It provides us a scalar value whose value would like to reduce or minimize as a result of training process. The choice of loss function depends upon the task and objective. Here we are using mean squared error between true and predicted values as our loss function.

```
def mse(true, prediction):
    return np.sum(np.power(prediction - true, 2)) / prediction.shape[0]

loss = mse(batch_y, out)
```

The function `mse` calculate mean squared error between any two arrays. The first array is supposed to be the true values and second array will be the prediction obtained from the forward pass.

Now we can write the how forward pass as below.

```
for e in range(epochs):

    epoch_losses = []

    for batch_x, batch_y in batch_generator(X_train, y_train):

        # FORWARD PASS
        l1_out = np.dot(batch_x, w1) + b1
        sig1_out = sigmoid(l1_out)

        l2_out = np.dot(sig1_out, w2) + b2
        sig2_out = sigmoid(l2_out)

        l3_out = np.dot(sig2_out, w3) + b3

        # LOSS CALCULATION
        loss = mse(batch_y, l3_out)
        epoch_losses.append(loss)
```

We are saving the loss obtained at each mini-batch step in a list `epoch_losses`. This will be used later for plotting purpose.

1.4.9 backward pass

We are interested in finding out how much loss is contributed by the each of the parameter in our neural network. So that we can tune/change the parameter accordingly. We have three layers and each layer has two kinds of parameters i.e., weights and biases. Therefore, we would like to find out how much loss is contributed by weights and biases in these three layers. This can be achieved by finding out the partial derivative of the loss with respect to partial derivative of the specific parameter i.e., weights and biases.

$$h < sub > theta; < /sub > (x) =$$
$$theta; < sub > o < /sub > x + theta; < sub > 1 < /sub > x$$

1.4.10 loss gradient

```
def mse_backward(true, prediction):
    return 2.0 * (prediction - true) / prediction.shape[0]

d_loss = mse_backward(batch_y, l3_out)  # -> (batch_size, num_outs)
```

1.4.11 third layer gradients

The third layer consisted of only two operations 1) dot product of inputs with `w3` and addition of *b3* bias in the result. Now during the backpropagation, we calculate three kinds of gradients 1) gradient of bias `b3`, 2) gradient of weights `w3` and 3) gradient of inputs to 3rd layer

```
# bias third layer
d_b3 = np.sum(d_loss, axis=0, keepdims=True)  # -> (1, 1)

# weight third layer
input_grad_w3 = np.dot(d_loss, w3.T)  # -> (batch_size, l2_neurons)
d_w3 = np.dot(sig2_out.T, d_loss)  # -> (l2_neurons, num_outs)
```

1.4.12 second layer gradients

```
def sigmoid_backward(inputs, out_grad):
    sig_out = sigmoid(inputs)
    d_inputs = sig_out * (1.0 - sig_out) * out_grad
    return d_inputs

# sigmoid second layer
# ((batch_size, batch_size), (batch_size, 14)) -> (batch_size, l2_neurons)
d_l2_out = sigmoid_backward(l2_out, input_grad_w3)

# bias second layer
d_b2 = np.sum(d_l2_out, axis=0, keepdims=True)  # -> (1, l2_neurons)

# weight second layer
input_grad_w2 = np.dot(d_l2_out, w2.T)  # -> (batch_size, l1_neurons)
d_w2 = np.dot(sig1_out.T, d_l2_out)  # -> (l1_neurons, l2_neurons)
```

1.4.13 first layer gradients

```
# ((batch_size, l1_neurons), (batch_size, l1_neurons)) -> (batch_size, l1_neurons)
d_sig1_out = sigmoid_backward(l1_out, input_grad_w2)

# bias first layer
d_b1 = np.sum(d_sig1_out, axis=0, keepdims=True)  # -> (1, l1_neurons)

# weight first layer
input_grads_w1 = np.dot(d_sig1_out, w1.T)  # -> (batch_size, num_ins)
```

(continues on next page)

(continued from previous page)

```
# derivative of (the propagated loss) w.r.t weights
d_w1 = np.dot(batch_x.T, d_sig1_out)  # -> (num_ins, l1_neurons)
```

1.4.14 parameter update

Now that we have calculated the gradients, we now know how much change needs to be carried out in each parameter (weights and biases). It is time to update weights and biases. This step is neural network libraries is carried out by the **optimizer``.

```
for e in range(epochs):

    epoch_losses = []

    for batch_x, batch_y in batch_generator(X_train, y_train):
        # FORWARD PASS
        ...

        # BACKWARD PASS
        ...

        # OPTIMIZER STEP
        w3 -= lr * d_w3
        b3 -= lr * d_b3

        w2 -= lr * d_w2
        b2 -= lr * d_b2

        w1 -= lr * d_w1
        b1 -= lr * d_b1
```

We can note that the parameter `lr` is kind of check on the change in parameters. Larger the value of `lr`, the larger will be the change and vice versa.

1.4.15 model evaluation

Once we have updated the parameters of the model i.e. we now have a new model, we would like to see how this new model performs. We do this by performing the forward pass with the updated parameters. However, this check is performed not on the training data but on a different data which is usually called validation data. We pass the inputs of the validation data through our network, calculate prediction and compare this prediction with true values to calculate a performance metric of our interest. Usually we would like to see the performance of our model on more than one performance metrics of different nature. The choice of performance metric is highly subjective to our task.

```
for e in range(epochs):

    epoch_losses = []

    for batch_x, batch_y in batch_generator(X_train, y_train):

        ...
    # Evaluation on validation data
```

(continues on next page)

(continued from previous page)

```
    l1_out_ = sigmoid(np.dot(x, w1) + b1)
    l2_out_ = sigmoid(np.dot(l1_out_, w2) + b2)
    predicted = np.dot(l2_out_, w3) + b3

    val_loss = mse(y_val, predicted)
```

1.4.16 loss curve

We will get the value of `loss` and `val_loss` after each mini-batch. We would be interesting in plotting these losses during the model training. We usually take the average of losses and val_losses during all the mini-batches in an epoch. Thus, after `n` epochs, we will have an array of length n for loss and val_loss. Plotting these arrays together provides us important information about the training behaviour of our neural network.

```
 from easy_mpl import plot

 train_losses = np.full(epochs, fill_value=np.nan)
 val_losses = np.full(epochs, fill_value=np.nan)

 for e in range(epochs):

     epoch_losses = []

         for batch_x, batch_y in batch_generator(X_train, y_train):
             ...
     # Evaluation on validation data
     l1_out_ = sigmoid(np.dot(x, w1) + b1)
     l2_out_ = sigmoid(np.dot(l1_out_, w2) + b2)
     predicted = np.dot(l2_out_, w3) + b3

     val_loss = mse(y_val, predicted)

     train_losses[e] = np.nanmean(epoch_losses)
     val_losses[e] = val_loss

plot(train_losses, label="Training", show=False)
plot(val_losses, label="Validation", grid=True)
```

metric for the prediction.

```
from SeqMetrics import RegressionMetrics

def eval_model(x, y, metric_name):
    # Evaluation on validation data
    l1_out_ = sigmoid(np.dot(x, w1) + b1)
    l2_out_ = sigmoid(np.dot(l1_out_, w2) + b2)
    prediction = np.dot(l2_out_, w3) + b3
    metrics = RegressionMetrics(y, prediction)
    return getattr(metrics, metric_name)()
```

```
from easy_mpl import plot

```

(continues on next page)

(continued from previous page)

```
train_losses = np.full(epochs, fill_value=np.nan)
val_losses = np.full(epochs, fill_value=np.nan)

for e in range(epochs):

    epoch_losses = []

    for batch_x, batch_y in batch_generator(X_train, y_train):
     ...
    # Evaluation on validation data
    val_loss = eval_model(batch_x, batch_y, 'mse')

    train_losses[e] = np.nanmean(epoch_losses)
    val_losses[e] = val_loss

plot(train_losses, label="Training", show=False)
plot(val_losses, label="Validation", grid=True)
```

Now we can also evaluate model for any other performance metric

```
print(eval_model(batch_x, batch_y, 'nse'))
```

1.4.17 early stopping

How long should we train our neural network i.e. what should be the value of epochs? The answer to this question can only be given by looking at the loss and validation loss curves. We should keep training our neural network as long as the performance of our network is improving on validation data i.e. as long as val_loss is decreasing. What if we have set the value of epochs to 5000 and the validation loss stops decreasing after 50th epoch? Should we wait for complete 5000 epochs to complete? We usually set a criteria to stop/break the training loop early depending upon the performance of our model on validation data. This is called early stopping. We following code, we break the training loop if the validation loss does not decrease for 50 consecutive epochs. The value of 50 is arbitrary here.

```
patience = 50
best_epoch = 0
epochs_since_best_epoch = 0

for e in range(epochs):

    epoch_losses = []

    for batch_x, batch_y in batch_generator(X_train, y_train):
        ...

    # calculation of val_loss
        ...

    if val_loss <= np.nanmin(val_losses):
        epochs_since_best_epoch = 0
        print(f"{e} {round(np.nanmean(epoch_losses).item(), 4)} {round(val_loss, 4)}")
    else:
        epochs_since_best_epoch += 1
```

(continues on next page)

(continued from previous page)

```
    if epochs_since_best_epoch > patience:
        print(f"""Early Stopping at {e} because val loss did not improved since
         {e - epochs_since_best_epoch}""")
        break
```

1.4.18 weight initialization

```
def glorot_initializer(
    shape:tuple,
    scale:float = 1.0,
    seed:int = 313
)->np.ndarray:
    ng_l1 = default_rng(seed)
    scale /= max(1., (shape[0] + shape[1]) / 2.)
    limit = math.sqrt(3.0 * scale)
    return ng_l1.uniform(-limit, limit, size=shape)

w1 = glorot_initializer((dataset.num_ins, l1_neurons))
w2 = glorot_initializer((w1.shape[1], l2_neurons))
w3 = glorot_initializer((w2.shape[1], l3_neurons))
```

1.4.19 Complete code

The complete python code which we have seen about is given as one peace below!

```
import numpy as np
import math
import matplotlib.pyplot as plt
from easy_mpl import plot, imshow
from numpy.random import default_rng
from SeqMetrics import RegressionMetrics
from ai4water.datasets import busan_beach
from ai4water.preprocessing import DataSet
from sklearn.preprocessing import StandardScaler

data = busan_beach()
#data = data.drop(data.index[317])
#data['tetx_coppml'] = np.log(data['tetx_coppml'])
print(data.shape)

s = StandardScaler()
data = s.fit_transform(data)

dataset = DataSet(data, val_fraction=0.0, seed=2809)
X_train, y_train = dataset.training_data()
X_val, y_val = dataset.test_data()

def batch_generator(X, Y, size=32):
```

(continues on next page)

(continued from previous page)

```
    N = X.shape[0]

    for ii in range(0, N, size):
        X_batch, y_batch = X[ii:ii + size], Y[ii:ii + size]

        yield X_batch, y_batch


def sigmoid(inputs):
    return 1.0 / (1.0 + np.exp(-1.0 * inputs))

def sigmoid_backward(inputs, out_grad):
    sig_out = sigmoid(inputs)
    d_inputs = sig_out * (1.0 - sig_out) * out_grad
    return d_inputs

def relu(inputs):
    return np.maximum(0, inputs)

def relu_backward(inputs, out_grad):
    d_inputs = np.array(out_grad, copy = True)
    d_inputs[inputs <= 0] = 0
    return d_inputs

def mse(true, prediction):
    return np.sum(np.power(prediction - true, 2)) / prediction.shape[0]

def mse_backward(true, prediction):
    return 2.0 * (prediction - true) / prediction.shape[0]

def eval_model(x, y, metric_name)->float:
    # Evaluation on validation data
    l1_out_ = sigmoid(np.dot(x, w1) + b1)
    l2_out_ = sigmoid(np.dot(l1_out_, w2) + b2)
    prediction = np.dot(l2_out_, w3) + b3
    metrics = RegressionMetrics(y, prediction)
    return getattr(metrics, metric_name)()


# hyperparameters
batch_size = 32
lr = 0.001
epochs = 1000
l1_neurons = 10
l2_neurons = 5
l3_neurons = 1

# parameters (weights and biases)

def glorot_initializer(
        shape:tuple,
        scale:float = 1.0,
```

(continues on next page)

(continued from previous page)

```
        seed:int = 313
)->np.ndarray:
    ng_l1 = default_rng(seed)
    scale /= max(1., (shape[0] + shape[1]) / 2.)
    limit = math.sqrt(3.0 * scale)
    return ng_l1.uniform(-limit, limit, size=shape)


rng_l1 = default_rng(313)
rng_l2 = default_rng(313)
rng_l3 = default_rng(313)
w1 = glorot_initializer((dataset.num_ins, l1_neurons))
b1 = np.zeros((1, l1_neurons))
w2 = glorot_initializer((w1.shape[1], l2_neurons))
b2 = np.zeros((1, l2_neurons))
w3 = glorot_initializer((w2.shape[1], l3_neurons))
b3 = np.zeros((1, l3_neurons))

train_losses = np.full(epochs, fill_value=np.nan)
val_losses = np.full(epochs, fill_value=np.nan)

tolerance = 1e-5
patience = 50
best_epoch = 0
epochs_since_best_epoch = 0

for e in range(epochs):

    epoch_losses = []

    for batch_x, batch_y in batch_generator(X_train, y_train, size=batch_size):

        # FORWARD PASS
        l1_out = np.dot(batch_x, w1) + b1
        sig1_out = sigmoid(l1_out)

        l2_out = np.dot(sig1_out, w2) + b2
        sig2_out = sigmoid(l2_out)

        l3_out = np.dot(sig2_out, w3) + b3

        # LOSS CALCULATION
        loss = mse(batch_y, l3_out)
        epoch_losses.append(loss)

        # BACKWARD PASS
        d_loss = mse_backward(batch_y, l3_out)  # -> (batch_size, num_outs)

        # bias third layer
        d_b3 = np.sum(d_loss, axis=0, keepdims=True)  # -> (1, 1)

        # weight third layer
```

(continues on next page)

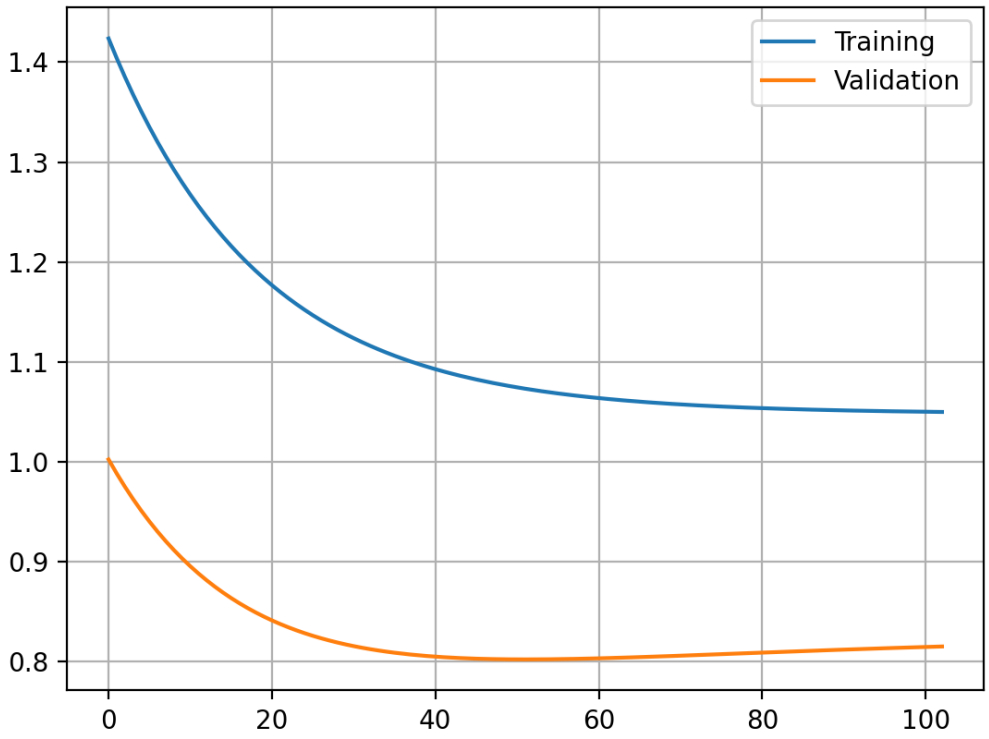
(continued from previous page)

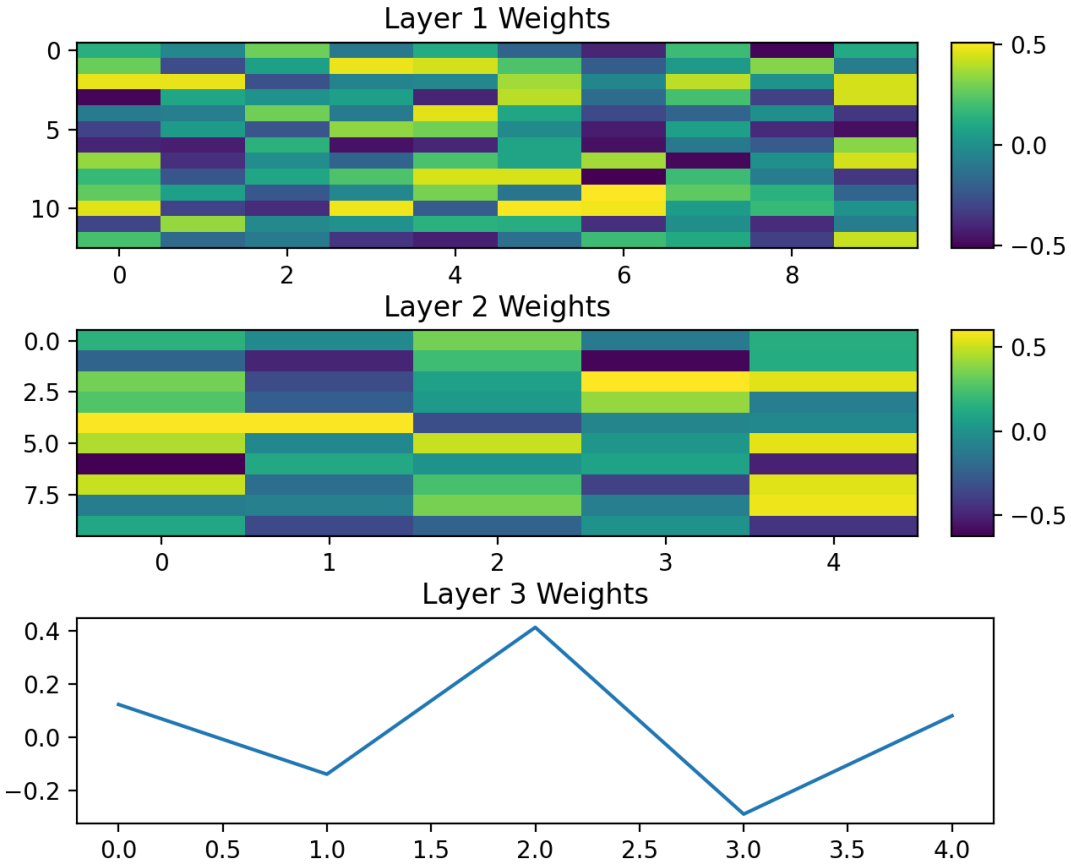
```
        input_grad_w3 = np.dot(d_loss, w3.T)  # -> (batch_size, l2_neurons)
        d_w3 = np.dot(sig2_out.T, d_loss)  # -> (l2_neurons, num_outs)

        # sigmoid second layer
        # ((batch_size, batch_size), (batch_size, 14)) -> (batch_size, l2_neurons)
        d_l2_out = sigmoid_backward(l2_out, input_grad_w3)

        # bias second layer
        d_b2 = np.sum(d_l2_out, axis=0, keepdims=True)  # -> (1, l2_neurons)

        # weight second layer
        input_grad_w2 = np.dot(d_l2_out, w2.T)  # -> (batch_size, l1_neurons)
        d_w2 = np.dot(sig1_out.T, d_l2_out)  # -> (l1_neurons, l2_neurons)

        # sigmoid first layer
        # ((batch_size, l1_neurons), (batch_size, l1_neurons)) -> (batch_size, l1_
↪neurons)
        d_sig1_out = sigmoid_backward(l1_out, input_grad_w2)

        # bias first layer
        d_b1 = np.sum(d_sig1_out, axis=0, keepdims=True)  # -> (1, l1_neurons)

        # weight first layer
        input_grads_w1 = np.dot(d_sig1_out, w1.T)  # -> (batch_size, num_ins)
        # derivate of (the propagated loss) w.r.t weights
        d_w1 = np.dot(batch_x.T, d_sig1_out)  # -> (num_ins, l1_neurons)

        # OPTIMIZER STEP
        w3 -= lr * d_w3
        b3 -= lr * d_b3

        w2 -= lr * d_w2
        b2 -= lr * d_b2

        w1 -= lr * d_w1
        b1 -= lr * d_b1

    # Evaluation on validation data
    val_loss = eval_model(X_val, y_val, "mse")

    train_losses[e] = np.nanmean(epoch_losses)
    val_losses[e] = val_loss

    if val_loss <= np.nanmin(val_losses):
        epochs_since_best_epoch = 0
        print(f"{e} {round(np.nanmean(epoch_losses).item(), 4)} {round(val_loss, 4)}")
    else:
        epochs_since_best_epoch += 1

    if epochs_since_best_epoch > patience:
        print(f"""Early Stopping at {e} because val loss did not improved since
         {e-epochs_since_best_epoch}""")
```

(continues on next page)

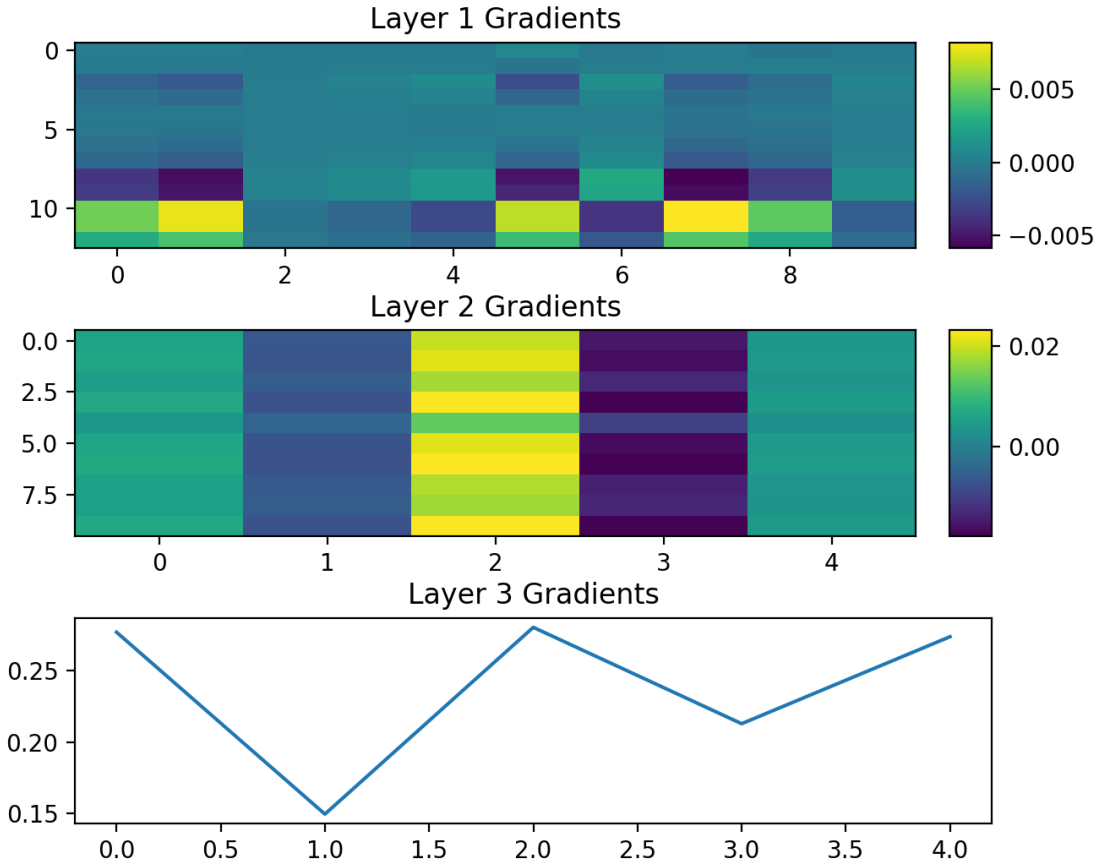
(continued from previous page)

```
        break


plot(train_losses, label="Training", show=False)
plot(val_losses, label="Validation", grid=True)

print(eval_model(X_val, y_val, "r2"))

_, (ax1, ax2, ax3) = plt.subplots(3, figsize=(7, 6), gridspec_kw={"hspace": 0.4})
imshow(w1, aspect="auto", colorbar=True, title="Layer 1 Weights", ax=ax1, show=False)
imshow(w2, aspect="auto", colorbar=True, title="Layer 2 Weights", ax=ax2, show=False)
plot(w3, ax=ax3, show=False, title="Layer 3 Weights")
plt.show()

_, (ax1, ax2, ax3) = plt.subplots(3, figsize=(7, 6), gridspec_kw={"hspace": 0.4})
imshow(d_w1, aspect="auto", colorbar=True, title="Layer 1 Gradients", ax=ax1, show=False)
imshow(d_w2, aspect="auto", colorbar=True, title="Layer 2 Gradients", ax=ax2, show=False)
plot(d_w3, ax=ax3, show=False, title="Layer 3 Gradients")
plt.show()
```




Layer 1 Weights
0
5
10
0
2
4
6
8
0.5
0.0
−0.5
Layer 2 Weights
0.0
2.5
5.0
7.5
0
1
2
3
4
0.5
0.0
−0.5
Layer 3 Weights
0.4
0.2
0.0
−0.2
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0

•



•

```
/home/docs/checkouts/readthedocs.org/user_builds/ml-tutorials/envs/latest/lib/python3.7/
↪site-packages/sklearn/experimental/enable_hist_gradient_boosting.py:17: UserWarning:␣
↪Since version 1.0, it is not needed to import enable_hist_gradient_boosting anymore.␣
↪HistGradientBoostingClassifier and HistGradientBoostingRegressor are now stable and␣
↪can be normally imported from sklearn.ensemble.
  "Since version 1.0, "
(1446, 14)

********** Removing Examples with nan in labels  **********

***** Training *****
input_x shape:  (152, 13)
target shape:  (152, 1)

********** Removing Examples with nan in labels  **********

***** Test *****
input_x shape:  (66, 13)
target shape:  (66, 1)
0 1.4237 1.0023
```

(continues on next page)

(continued from previous page)

```
1 1.404 0.9882
2 1.3853 0.975
3 1.3676 0.9627
4 1.3509 0.9511
5 1.335 0.9402
6 1.32 0.93
7 1.3058 0.9205
8 1.2923 0.9116
9 1.2796 0.9032
10 1.2675 0.8955
11 1.2561 0.8882
12 1.2452 0.8814
13 1.235 0.875
14 1.2252 0.8691
15 1.216 0.8636
16 1.2073 0.8584
17 1.199 0.8536
18 1.1912 0.8492
19 1.1837 0.845
20 1.1767 0.8412
21 1.17 0.8376
22 1.1637 0.8343
23 1.1577 0.8312
24 1.1521 0.8284
25 1.1467 0.8258
26 1.1416 0.8233
27 1.1367 0.8211
28 1.1322 0.819
29 1.1278 0.8172
30 1.1237 0.8154
31 1.1198 0.8139
32 1.1161 0.8124
33 1.1126 0.8111
34 1.1093 0.8099
35 1.1061 0.8088
36 1.1031 0.8079
37 1.1003 0.807
38 1.0976 0.8062
39 1.0951 0.8055
40 1.0927 0.8049
41 1.0904 0.8044
42 1.0882 0.8039
43 1.0862 0.8035
44 1.0842 0.8032
45 1.0824 0.8029
46 1.0806 0.8026
47 1.0789 0.8025
48 1.0774 0.8023
49 1.0759 0.8022
50 1.0745 0.8022
51 1.0731 0.8022
Early Stopping at 102 because val loss did not improved since
```

(continues on next page)

(continued from previous page)

```
         51
0.10823983440230535
```

1.4.20 comparison with tensorflow

```
from ai4water import Model
model = Model(
    model = {"layers": {
     "Input": {"shape": (dataset.num_ins,)},
     "Dense": l1_neurons,
     "Dense_2": l2_neurons,
     "Dense_3": l3_neurons
    }},
    lr=lr,
    batch_size=batch_size,
    epochs=epochs,
    optimizer="SGD"
)
h = model.fit(X_train, y=y_train, validation_data=(X_val, y_val))
```

Total running time of the script: (0 minutes 2.097 seconds)

1.5 understanding Dense layer in Keras

```
# simple `Conv1D`
```

```
import tensorflow as tf
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Dense, TimeDistributed, Conv1D, LSTM,␣
↪MaxPool1D
import numpy as np

def reset_seed(seed=313):
    tf.keras.backend.clear_session()
    tf.random.set_seed(seed)
    np.random.seed(seed)

np.set_printoptions(linewidth=150)

print(tf.__version__, np.__version__)
```

```
2.7.0 1.21.6
```

```
input_features = 3
lookback = 6
batch_size=2
input_shape = lookback,input_features
```

(continues on next page)

(continued from previous page)

```
ins = Input(shape=input_shape, name='my_input')
outs = Conv1D(filters=8, kernel_size=3,
              strides=1, padding='same', kernel_initializer='ones',
              name='my_conv1d')(ins)
model = Model(inputs=ins, outputs=outs)

input_array = np.arange(36).reshape((batch_size, *input_shape))
conv1d_weights = model.get_layer('my_conv1d').weights[0].numpy()
output_array = model.predict(input_array)
```

```
print(input_array.shape)

print(input_array)
```

```
(2, 6, 3)
[[[ 0  1  2]
  [ 3  4  5]
  [ 6  7  8]
  [ 9 10 11]
  [12 13 14]
  [15 16 17]]

 [[18 19 20]
  [21 22 23]
  [24 25 26]
  [27 28 29]
  [30 31 32]
  [33 34 35]]]
```

```
print(conv1d_weights.shape)

print(conv1d_weights)
```

```
(3, 3, 8)
[[[1. 1. 1. 1. 1. 1. 1. 1.]
  [1. 1. 1. 1. 1. 1. 1. 1.]
  [1. 1. 1. 1. 1. 1. 1. 1.]]

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

 [[1. 1. 1. 1. 1. 1. 1. 1.]
  [1. 1. 1. 1. 1. 1. 1. 1.]
  [1. 1. 1. 1. 1. 1. 1. 1.]]]
```

```
print(output_array)

print(output_array.shape)
```

```
[[[ 15.  15.  15.  15.  15.  15.  15.  15.]
  [ 36.  36.  36.  36.  36.  36.  36.  36.]
  [ 63.  63.  63.  63.  63.  63.  63.  63.]
  [ 90.  90.  90.  90.  90.  90.  90.  90.]
  [117. 117. 117. 117. 117. 117. 117. 117.]
  [ 87.  87.  87.  87.  87.  87.  87.  87.]]

 [[123. 123. 123. 123. 123. 123. 123. 123.]
  [198. 198. 198. 198. 198. 198. 198. 198.]
  [225. 225. 225. 225. 225. 225. 225. 225.]
  [252. 252. 252. 252. 252. 252. 252. 252.]
  [279. 279. 279. 279. 279. 279. 279. 279.]
  [195. 195. 195. 195. 195. 195. 195. 195.]]]
(2, 6, 8)
```

```
# multiple inputs multiple layers
```

```
input_features = 3
lookback = 3
batch_size=2
input_shape = lookback,input_features
ins1 = Input(shape=input_shape, name='my_input1')
ins2 = Input(shape=input_shape, name='my_input2')
outs1 = Conv1D(filters=8, kernel_size=3,
               strides=1, padding='same', kernel_initializer='ones',
               name='my_conv1d1')(ins1)
outs2 = Conv1D(filters=8, kernel_size=3,
               strides=1, padding='same', kernel_initializer='ones',
               name='my_conv1d2')(ins2)
model = Model(inputs=[ins1, ins2], outputs=[outs1, outs2])

sub_seq = 2
input_shape = sub_seq, 3, input_features
input_array = np.arange(36).reshape((batch_size, *input_shape))
input_array1 = input_array[:, 0, :, :]
input_array2 = input_array[:, 1, :, :]

conv1d1_weights = model.get_layer('my_conv1d1').weights[0].numpy()
conv1d2_weights = model.get_layer('my_conv1d2').weights[0].numpy()
output_array = model.predict([input_array1, input_array2])
```

```
print(input_array1, '\n\n', input_array2)
```

```
[[[ 0  1  2]
  [ 3  4  5]
  [ 6  7  8]]

 [[18 19 20]
  [21 22 23]
  [24 25 26]]]

 [[[ 9 10 11]
```

(continues on next page)

(continued from previous page)

```
  [12 13 14]
  [15 16 17]]

 [[27 28 29]
  [30 31 32]
  [33 34 35]]]
```

```
print(conv1d2_weights)

print(conv1d2_weights)
```

```
[[[1. 1. 1. 1. 1. 1. 1. 1.]
  [1. 1. 1. 1. 1. 1. 1. 1.]
  [1. 1. 1. 1. 1. 1. 1. 1.]]

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

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

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

 [[1. 1. 1. 1. 1. 1. 1. 1.]
  [1. 1. 1. 1. 1. 1. 1. 1.]
  [1. 1. 1. 1. 1. 1. 1. 1.]]]
```

```
print(output_array)
```

```
[array([[[ 15.,  15.,  15.,  15.,  15.,  15.,  15.,  15.],
        [ 36.,  36.,  36.,  36.,  36.,  36.,  36.,  36.],
        [ 33.,  33.,  33.,  33.,  33.,  33.,  33.,  33.]],

       [[123., 123., 123., 123., 123., 123., 123., 123.],
        [198., 198., 198., 198., 198., 198., 198., 198.],
        [141., 141., 141., 141., 141., 141., 141., 141.]]], dtype=float32), array([[[ 69.
↪,  69.,  69.,  69.,  69.,  69.,  69.,  69.],
        [117., 117., 117., 117., 117., 117., 117., 117.],
        [ 87.,  87.,  87.,  87.,  87.,  87.,  87.,  87.]],

       [[177., 177., 177., 177., 177., 177., 177., 177.],
        [279., 279., 279., 279., 279., 279., 279., 279.],
        [195., 195., 195., 195., 195., 195., 195., 195.]]], dtype=float32)]
```

```
# multiple inputs shared layer
```

```
input_features = 3
lookback = 6
sub_seq = 2
input_shape = sub_seq, 3, input_features
batch_size = 2

ins = Input(shape=input_shape, name='my_input')
conv = Conv1D(filters=8, kernel_size=3,
              strides=1, padding='same',
              kernel_initializer='ones', name='my_conv1d')

conv1_out = conv(ins[:, 0, :, :])
conv2_out = conv(ins[:, 1, :, :])
model = Model(inputs=ins, outputs=[conv1_out, conv2_out])

input_array = np.arange(36).reshape((batch_size, *input_shape))
conv1d_weights = model.get_layer('my_conv1d').weights[0].numpy()
output_array = model.predict(input_array)
```

```
print(input_array)
```

```
[[[[ 0  1  2]
   [ 3  4  5]
   [ 6  7  8]]

  [[ 9 10 11]
   [12 13 14]
   [15 16 17]]]


 [[[18 19 20]
   [21 22 23]
   [24 25 26]]

  [[27 28 29]
   [30 31 32]
   [33 34 35]]]]
```

```
print(conv1d_weights)
```

```
[[[1. 1. 1. 1. 1. 1. 1. 1.]
  [1. 1. 1. 1. 1. 1. 1. 1.]
  [1. 1. 1. 1. 1. 1. 1. 1.]]

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

 [[1. 1. 1. 1. 1. 1. 1. 1.]
```

(continues on next page)

(continued from previous page)

```
  [1. 1. 1. 1. 1. 1. 1. 1.]
  [1. 1. 1. 1. 1. 1. 1. 1.]]]
```

```
print(output_array[0])
```

```
[[[ 15.  15.  15.  15.  15.  15.  15.  15.]
  [ 36.  36.  36.  36.  36.  36.  36.  36.]
  [ 33.  33.  33.  33.  33.  33.  33.  33.]]

 [[123. 123. 123. 123. 123. 123. 123. 123.]
  [198. 198. 198. 198. 198. 198. 198. 198.]
  [141. 141. 141. 141. 141. 141. 141. 141.]]]
```

```
print(output_array[1])
```

```
[[[ 69.  69.  69.  69.  69.  69.  69.  69.]
  [117. 117. 117. 117. 117. 117. 117. 117.]
  [ 87.  87.  87.  87.  87.  87.  87.  87.]]

 [[177. 177. 177. 177. 177. 177. 177. 177.]
  [279. 279. 279. 279. 279. 279. 279. 279.]
  [195. 195. 195. 195. 195. 195. 195. 195.]]]
```

```
# `TimeDistributed Conv1D`
```

```
input_features = 3
lookback = 6
sub_seq = 2
input_shape = sub_seq, 3, input_features
batch_size = 2

ins = Input(shape=input_shape, name='my_input')
outs = TimeDistributed(Conv1D(filters=8, kernel_size=3,
              strides=1, padding='same', kernel_initializer='ones',
              name='my_conv1d'))(ins)
model = Model(inputs=ins, outputs=outs)

input_array = np.arange(36).reshape((batch_size, *input_shape))
output_array = model.predict(input_array)
```

```
print(input_array)
```

```
[[[[ 0  1  2]
   [ 3  4  5]
   [ 6  7  8]]

  [[ 9 10 11]
   [12 13 14]
   [15 16 17]]]
```

(continues on next page)

(continued from previous page)

```
 [[[18 19 20]
   [21 22 23]
   [24 25 26]]

  [[27 28 29]
   [30 31 32]
   [33 34 35]]]]
```

```
print(output_array)
```

```
[[[[ 15.  15.  15.  15.  15.  15.  15.  15.]
   [ 36.  36.  36.  36.  36.  36.  36.  36.]
   [ 33.  33.  33.  33.  33.  33.  33.  33.]]

  [[ 69.  69.  69.  69.  69.  69.  69.  69.]
   [117. 117. 117. 117. 117. 117. 117. 117.]
   [ 87.  87.  87.  87.  87.  87.  87.  87.]]]


 [[[123. 123. 123. 123. 123. 123. 123. 123.]
   [198. 198. 198. 198. 198. 198. 198. 198.]
   [141. 141. 141. 141. 141. 141. 141. 141.]]

  [[177. 177. 177. 177. 177. 177. 177. 177.]
   [279. 279. 279. 279. 279. 279. 279. 279.]
   [195. 195. 195. 195. 195. 195. 195. 195.]]]]
```

```
# So `TimeDistributed` Just applies same `Conv1D` to each sub-sequence/incoming input.

# `TimeDistributed` `LSTM`
```

```
tf.random.set_seed(313)

input_features = 3
sub_seq = 2
input_shape = sub_seq, 3, input_features
batch_size = 2

ins = Input(shape=input_shape, name='my_input')
outs = TimeDistributed(LSTM(units=8, name='my_lstm'))(ins)
model = Model(inputs=ins, outputs=outs)

input_array = np.arange(36).reshape((batch_size, *input_shape))
output_array = model.predict(input_array)
```

```
print(input_array)
```

```
[[[[ 0  1  2]
   [ 3  4  5]
```

(continues on next page)

(continued from previous page)

```
   [ 6  7  8]]

  [[ 9 10 11]
   [12 13 14]
   [15 16 17]]]


 [[[18 19 20]
   [21 22 23]
   [24 25 26]]

  [[27 28 29]
   [30 31 32]
   [33 34 35]]]]
```

```
print(output_array[:, 0, :])
```

```
[[-0.16062409  0.22913463  0.10585532  0.60561293 -0.00365596 -0.14262524  0.0095918   0.
↪00026523]
 [-0.00133879  0.01697312  0.01358414  0.24525642 -0.00000001 -0.00133852  0.00000001  0.
↪         ]]
```

```
print(output_array[:, 1, :])
```

```
[[-0.01696269  0.06433662  0.06806362  0.58035445 -0.00000462 -0.0139938   0.00001501  0.
↪00000001]
 [-0.00010769  0.00426498  0.00253441  0.08013909 -0.         -0.00013428  0.          0.
↪         ]]
```

```
# manual weight sharing of `LSTM`
```

```
tf.random.set_seed(313)

input_features = 3
sub_seq = 2
input_shape = sub_seq, 3, input_features
batch_size = 2

ins = Input(shape=input_shape, name='my_input')
lstm = LSTM(units=8, name='my_lstm')

lstm1_out = lstm(ins[:, 0, :, :])
lstm2_out = lstm(ins[:, 1, :, :])
model = Model(inputs=ins, outputs=[lstm1_out, lstm2_out])

input_array = np.arange(36).reshape((batch_size, *input_shape))
output_array = model.predict(input_array)
```

```
print(input_array)
```

```
[[[[ 0  1  2]
   [ 3  4  5]
   [ 6  7  8]]

  [[ 9 10 11]
   [12 13 14]
   [15 16 17]]]


 [[[18 19 20]
   [21 22 23]
   [24 25 26]]

  [[27 28 29]
   [30 31 32]
   [33 34 35]]]]
```

```
print(output_array[0])
```

```
[[-0.16062409  0.22913463  0.10585532  0.60561293 -0.00365596 -0.14262524  0.0095918   0.
↪00026523]
 [-0.00133879  0.01697312  0.01358414  0.24525642 -0.00000001 -0.00133852  0.00000001  0.
↪         ]]
```

```
print(output_array[1])
```

```
[[-0.01696269  0.06433662  0.06806362  0.58035445 -0.00000462 -0.0139938   0.00001501  0.
↪00000001]
 [-0.00010769  0.00426498  0.00253441  0.08013909 -0.         -0.00013428  0.          0.
↪         ]]
```

```
# Curious case of `Dense`
```

```
tf.random.set_seed(313)

input_features = 3
lookback = 6
batch_size=2
input_shape = lookback,input_features

input_shape = lookback, input_features
ins = Input(input_shape, name='my_input')
out = Dense(units=5,  name='my_output')(ins)
model = Model(inputs=ins, outputs=out)

input_array = np.arange(36).reshape(batch_size, *input_shape)
output_array = model.predict(input_array)
```

```
print(input_array.shape)
```

```
(2, 6, 3)
```

```
print(input_array)
```

```
[[[ 0  1  2]
  [ 3  4  5]
  [ 6  7  8]
  [ 9 10 11]
  [12 13 14]
  [15 16 17]]

 [[18 19 20]
  [21 22 23]
  [24 25 26]
  [27 28 29]
  [30 31 32]
  [33 34 35]]]
```

```
print(output_array.shape)
```

```
(2, 6, 5)
```

```
print(output_array)
```

```
[[[  0.27596885   0.70854443  -0.53744566  -0.70061463  -0.20640421]
  [  0.36203665   3.999878    -1.7514435   -2.695434     0.17058378]
  [  0.44810438   7.291211    -2.9654412   -4.6902537    0.5475718 ]
  [  0.53417236  10.582545    -4.179439    -6.685073     0.92456   ]
  [  0.6202401   13.8738785   -5.393437    -8.679893     1.3015478 ]
  [  0.7063078   17.165213    -6.607435   -10.674712     1.6785362 ]]

 [[  0.7923758   20.456545    -7.821433   -12.669531     2.0555243 ]
  [  0.8784433   23.747879    -9.035431   -14.664351     2.4325123 ]
  [  0.9645113   27.039211   -10.249429   -16.65917      2.8094997 ]
  [  1.0505793   30.330545   -11.463427   -18.65399      3.1864882 ]
  [  1.1366472   33.62188    -12.6774235  -20.64881      3.5634766 ]
  [  1.2227142   36.91321    -13.891422   -22.64363      3.9404643 ]]]
```

```
tf.random.set_seed(313)

input_features = 3
lookback = 6
sub_seq = 2
input_shape = sub_seq, 3, input_features
batch_size = 2

ins = Input(input_shape, name='my_input')
out = TimeDistributed(Dense(units=5, name='my_output'))(ins)
model = Model(inputs=ins, outputs=out)
```

(continues on next page)

(continued from previous page)

```
input_array = np.arange(36).reshape(batch_size, *input_shape)
output_array = model.predict(input_array)
```

```
print(input_array.shape)
```

```
(2, 2, 3, 3)
```

```
print(input_array)
```

```
[[[[ 0  1  2]
   [ 3  4  5]
   [ 6  7  8]]

  [[ 9 10 11]
   [12 13 14]
   [15 16 17]]]


 [[[18 19 20]
   [21 22 23]
   [24 25 26]]

  [[27 28 29]
   [30 31 32]
   [33 34 35]]]]
```

```
print(output_array.shape)
```

```
(2, 2, 3, 5)
```

```
print(output_array)
```

```
[[[[  0.27596885   0.70854443  -0.53744566  -0.70061463  -0.20640421]
   [  0.36203665   3.999878    -1.7514435   -2.695434     0.17058378]
   [  0.44810438   7.291211    -2.9654412   -4.6902537    0.5475718 ]]

  [[  0.53417236  10.582545    -4.179439    -6.685073     0.92456   ]
   [  0.6202401   13.8738785   -5.393437    -8.679893     1.3015478 ]
   [  0.7063078   17.165213    -6.607435   -10.674712     1.6785362 ]]]


 [[[  0.7923758   20.456545    -7.821433   -12.669531     2.0555243 ]
   [  0.8784433   23.747879    -9.035431   -14.664351     2.4325123 ]
   [  0.9645113   27.039211   -10.249429   -16.65917      2.8094997 ]]

  [[  1.0505793   30.330545   -11.463427   -18.65399      3.1864882 ]
   [  1.1366472   33.62188    -12.6774235  -20.64881      3.5634766 ]
   [  1.2227142   36.91321    -13.891422   -22.64363      3.9404643 ]]]]
```

```
# so far looks very similar to `TimeDistributed(Conv1D)` or `TimeDistributed(LSTM)`.
```

```
tf.random.set_seed(313)

input_features = 3
lookback = 6
input_shape = lookback, input_features
batch_size = 2

ins = Input(input_shape, name='my_input')
out = TimeDistributed(Dense(5, use_bias=False, name='my_output'))(ins)
model = Model(inputs=ins, outputs=out)

input_array = np.arange(36).reshape(batch_size, *input_shape)
output_array = model.predict(input_array)
```

```
print(input_array.shape)

print(input_array)
```

```
(2, 6, 3)
[[[ 0  1  2]
  [ 3  4  5]
  [ 6  7  8]
  [ 9 10 11]
  [12 13 14]
  [15 16 17]]

 [[18 19 20]
  [21 22 23]
  [24 25 26]
  [27 28 29]
  [30 31 32]
  [33 34 35]]]
```

```
print(output_array.shape)

print(output_array)
```

```
(2, 6, 5)
[[[  0.27596885   0.70854443  -0.53744566  -0.70061463  -0.20640421]
  [  0.36203665   3.999878    -1.7514435   -2.695434     0.17058378]
  [  0.44810438   7.291211    -2.9654412   -4.6902537    0.5475718 ]
  [  0.53417236  10.582545    -4.179439    -6.685073     0.92456   ]
  [  0.6202401   13.8738785   -5.393437    -8.679893     1.3015478 ]
  [  0.7063078   17.165213    -6.607435   -10.674712     1.6785362 ]]

 [[  0.7923758   20.456545    -7.821433   -12.669531     2.0555243 ]
  [  0.8784433   23.747879    -9.035431   -14.664351     2.4325123 ]
  [  0.9645113   27.039211   -10.249429   -16.65917      2.8094997 ]
  [  1.0505793   30.330545   -11.463427   -18.65399      3.1864882 ]
```

(continues on next page)

(continued from previous page)

```
  [  1.1366472   33.62188    -12.6774235  -20.64881      3.5634766 ]
  [  1.2227142   36.91321    -13.891422   -22.64363      3.9404643 ]]]
```

```
# So whether we we `TimeDistributed(Dense)` or `Dense`, they are actually equivalent.

# What if we try same with `Conv1D` or `LSTM` i.e. wrapping these layers in
# `TimeDistributed` without modifying/dividing input into sub-sequences?
```

```
input_features = 3
lookback = 6
input_shape = lookback, input_features

# uncomment following lines

# ins = Input(shape=(lookback, input_features), name='my_input')
# outs = TimeDistributed(Conv1D(filters=8, kernel_size=3,
#               strides=1, padding='valid', kernel_initializer='ones',
#               name='my_conv1d'))(ins)
# model = Model(inputs=ins, outputs=outs)

input_array = np.arange(36).reshape((batch_size, *input_shape))
```

The above error message can be slightly confusing or atleast can be resolved in a wrong manner as we do in following case;

```
input_features = 3
lookback = 6
input_shape = lookback, input_features
ins = Input(shape=(batch_size, lookback, input_features), name='my_input')
outs = TimeDistributed(Conv1D(filters=8, kernel_size=3,
              strides=1, padding='valid', kernel_initializer='ones',
              name='my_conv1d'))(ins)
model = Model(inputs=ins, outputs=outs)

input_array = np.arange(36).reshape((batch_size, *input_shape))
print(input_array.shape)
```

```
(2, 6, 3)
```

```
# So we are able to compile the model, although it is wrong.
```

```
# uncomment following 2 lines
# output_array = model.predict(input_array)
# print(output_array.shape)
```

```
# This error message is exactly related to `TimeDistributed` layer. The `TimeDistributed`
# layer here expects input having 4 dimensions, 1st being batch size, second being the
# sub-sequences, 3rd being the time-steps or whatever and 4rth being number of input
# features here.
```

(continues on next page)

(continued from previous page)

```
# Anyhow, the conclusion is, we can't just wrap layers in `TimeDistributed` except
# for `Dense` layer. Hence, using `TimeDistributed(Dense)` does not make any
# sense (to me until version 2.3.0).
```

```
# More than just weight sharing

# `TimeDistributed` layer is meant to provide more functionality than just weight
# sharing. We see, pooling layers or flatten layers wrapped into `TimeDistributed`
# layer even though pooling layers or flattening layers don't have any weights.
# This is because if we have applied `TimeDistributed(Conv1D)`, this will sprout
# output for each sub-sequence. We would naturally like to apply pooling and
# consequently flattening layers to each output fo the sub-sequences.
```

```
input_features = 3
sub_seq = 2
input_shape = sub_seq, 3, input_features
batch_size = 2

ins = Input(shape=input_shape, name='my_input')
conv_outs = TimeDistributed(Conv1D(filters=8, kernel_size=3,
               strides=1,
               padding='same',
               kernel_initializer='ones',
               name='my_conv1d'))(ins)
outs = TimeDistributed(MaxPool1D(pool_size=2))(conv_outs)
model = Model(inputs=ins, outputs=[outs, conv_outs])

input_array = np.arange(36).reshape((batch_size, *input_shape))
output_array, conv_output = model.predict(input_array)
```

```
print(input_array.shape)
```

```
(2, 2, 3, 3)
```

```
print(input_array)
```

```
[[[[ 0  1  2]
   [ 3  4  5]
   [ 6  7  8]]

  [[ 9 10 11]
   [12 13 14]
   [15 16 17]]]


 [[[18 19 20]
   [21 22 23]
   [24 25 26]]

  [[27 28 29]
```

(continues on next page)

(continued from previous page)

```
   [30 31 32]
   [33 34 35]]]]
```

```
print(conv_output.shape)
```

```
(2, 2, 3, 8)
```

```
print(conv_output)
```

```
[[[[ 15.  15.  15.  15.  15.  15.  15.  15.]
   [ 36.  36.  36.  36.  36.  36.  36.  36.]
   [ 33.  33.  33.  33.  33.  33.  33.  33.]]

  [[ 69.  69.  69.  69.  69.  69.  69.  69.]
   [117. 117. 117. 117. 117. 117. 117. 117.]
   [ 87.  87.  87.  87.  87.  87.  87.  87.]]]


 [[[123. 123. 123. 123. 123. 123. 123. 123.]
   [198. 198. 198. 198. 198. 198. 198. 198.]
   [141. 141. 141. 141. 141. 141. 141. 141.]]

  [[177. 177. 177. 177. 177. 177. 177. 177.]
   [279. 279. 279. 279. 279. 279. 279. 279.]
   [195. 195. 195. 195. 195. 195. 195. 195.]]]]
```

```
print(output_array.shape)
```

```
(2, 2, 1, 8)
```

```
print(output_array)
```

```
[[[[ 36.  36.  36.  36.  36.  36.  36.  36.]]

  [[117. 117. 117. 117. 117. 117. 117. 117.]]]


 [[[198. 198. 198. 198. 198. 198. 198. 198.]]

  [[279. 279. 279. 279. 279. 279. 279. 279.]]]]
```

```
input_features = 3
sub_seq = 2
input_shape = sub_seq, 3, input_features
batch_size = 2

ins = Input(shape=input_shape, name='my_input')
conv_outs = TimeDistributed(Conv1D(filters=8, kernel_size=3,
              strides=1, padding='same',
```

(continues on next page)

(continued from previous page)

```
              kernel_initializer='ones',
              name='my_conv1d'))(ins)
outs = TimeDistributed(MaxPool1D(pool_size=2, padding='same'))(conv_outs)
model = Model(inputs=ins, outputs=outs)

input_array = np.arange(36).reshape((batch_size, *input_shape))
output_array = model.predict(input_array)
```

```
print(input_array.shape)
```

```
(2, 2, 3, 3)
```

```
print(input_array)
```

```
[[[[ 0  1  2]
   [ 3  4  5]
   [ 6  7  8]]

  [[ 9 10 11]
   [12 13 14]
   [15 16 17]]]


 [[[18 19 20]
   [21 22 23]
   [24 25 26]]

  [[27 28 29]
   [30 31 32]
   [33 34 35]]]]
```

```
print(output_array.shape)
```

```
(2, 2, 2, 8)
```

```
print(output_array)

##############################################
```

```
[[[[ 36.  36.  36.  36.  36.  36.  36.  36.]
   [ 33.  33.  33.  33.  33.  33.  33.  33.]]

  [[117. 117. 117. 117. 117. 117. 117. 117.]
   [ 87.  87.  87.  87.  87.  87.  87.  87.]]]


 [[[198. 198. 198. 198. 198. 198. 198. 198.]
   [141. 141. 141. 141. 141. 141. 141. 141.]]

  [[279. 279. 279. 279. 279. 279. 279. 279.]
```

(continues on next page)

(continued from previous page)

```
   [195. 195. 195. 195. 195. 195. 195. 195.]]]]
```

Total running time of the script: (0 minutes 2.265 seconds)

1.6 Implementing LSTM in tensorflow from scratch

The purpose of this notebook is to illustrate how to build an LSTM from scratch in Tensorflow. Although the Tensorflow has implementation of LSTM in Keras. But since it comes with a lot of implementation options, reading the code of Tensorflow for LSTM can be confusing at the start. Therefore here is vanilla implementation of LSTM in Tensorflow. It has been shown that the results of this vanilla LSTM are full reproducible with Keras'LSTM. This shows that the simple implementation of LSTM in Tensorflow just has four equations and a for loop through time.

```
import os
import random

import numpy as np
np.__version__
```

```
'1.21.6'
```

```
import tensorflow as tf
tf.__version__
```

```
'2.7.0'
```

```
def seed_all(seed):
    """reset seed for reproducibility"""

    np.random.seed(seed)
    random.seed(seed)
    os.environ['PYTHONHASHSEED'] = str(seed)

    if int(tf.__version__.split('.')[0]) == 1:
        tf.compat.v1.random.set_random_seed(seed)
    elif int(tf.__version__.split('.')[0]) > 1:
        tf.random.set_seed(seed)


from tensorflow.keras.layers import Layer, Input, Dense
from tensorflow.keras.layers import LSTM as KLSTM
from tensorflow.keras.models import Model
from tensorflow.python.ops import array_ops
from tensorflow.python.keras import backend as K
```

```
assert tf.__version__ > "2.1", "results are not reproducible with Tensorflow below 2"
```

```
num_inputs = 3  # number of input features
lstm_units = 32
lookback_steps = 5  # also known as time_steps or sequence length
```

(continues on next page)

(continued from previous page)

```
num_samples = 10  # length of x,y


class SimpleLSTM(Layer):
    """A simplified implementation of LSTM layer with keras
    """

    def __init__(self, units, **kwargs):

        super(SimpleLSTM, self).__init__(**kwargs)

        self.activation = tf.nn.tanh
        self.rec_activation = tf.nn.sigmoid
        self.units = units

    def call(self, inputs):

        initial_state = tf.zeros((10, self.units))  # todo

        last_output, outputs, states = K.rnn(
            self.cell,
            inputs,
            [initial_state, initial_state]
        )

        return last_output

    def cell(self, inputs, states):

        h_tm1 = states[0]  # previous memory state
        c_tm1 = states[1]  # previous carry state

        k_i, k_f, k_c, k_o = array_ops.split(self.kernel, num_or_size_splits=4, axis=1)

        x_i = K.dot(inputs, k_i)
        x_f = K.dot(inputs, k_f)
        x_c = K.dot(inputs, k_c)
        x_o = K.dot(inputs, k_o)

        i = self.rec_activation(x_i + K.dot(h_tm1, self.rec_kernel[:, :self.units]))

        f = self.rec_activation(x_f + K.dot(h_tm1, self.rec_kernel[:, self.units:self.
↪units * 2]))

        c = f * c_tm1 + i * self.activation(x_c + K.dot(h_tm1, self.rec_kernel[:, self.
↪units * 2:self.units * 3]))

        o = self.rec_activation(x_o + K.dot(h_tm1, self.rec_kernel[:, self.units * 3:]))

        h = o * self.activation(c)

        return h, [h, c]
```

(continues on next page)

(continued from previous page)

```
    def build(self, input_shape):

        input_dim = input_shape[-1]

        self.kernel = self.add_weight(
            shape=(input_dim, self.units * 4),
            name='kernel',
            initializer="glorot_uniform")

        self.rec_kernel = self.add_weight(
            shape=(self.units, self.units * 4),
            name='recurrent_kernel',
            initializer="orthogonal")

        self.built = True

        return
```

```
inputs_tf = tf.range(150, dtype=tf.float32)
inputs_tf = tf.reshape(inputs_tf, (num_samples, lookback_steps, num_inputs))

seed_all(313)
lstm = SimpleLSTM(lstm_units)
h1 = lstm(inputs_tf)
h1_sum = tf.reduce_sum(h1)
print(K.eval(h1_sum))
```

```
25.964832
```

Now check the results of original lstm of Keras

```
seed_all(313)
lstm = KLSTM(lstm_units,
             recurrent_activation="sigmoid",
             unit_forget_bias=False,
             use_bias=False,
            )
h2 = lstm(inputs_tf)
h2_sum = tf.reduce_sum(h2)
print(K.eval(h2_sum))
```

```
25.964832
```

1.6.1 with bias

```
class LSTMWithBias(Layer):
    """A simplified implementation of LSTM layer with keras
    """

    def __init__(self, units, use_bias=True, **kwargs):

        super(LSTMWithBias, self).__init__(**kwargs)

        self.activation = tf.nn.tanh
        self.rec_activation = tf.nn.sigmoid
        self.units = units
        self.use_bias = use_bias

    def call(self, inputs):

        initial_state = tf.zeros((10, self.units))  # todo

        last_output, outputs, states = K.rnn(
            self.cell,
            inputs,
            [initial_state, initial_state]
        )

        return last_output

    def cell(self, inputs, states):

        h_tm1 = states[0]  # previous memory state
        c_tm1 = states[1]  # previous carry state

        k_i, k_f, k_c, k_o = array_ops.split(self.kernel, num_or_size_splits=4, axis=1)

        x_i = K.dot(inputs, k_i)
        x_f = K.dot(inputs, k_f)
        x_c = K.dot(inputs, k_c)
        x_o = K.dot(inputs, k_o)

        if self.use_bias:
            b_i, b_f, b_c, b_o = array_ops.split(
                self.bias, num_or_size_splits=4, axis=0)
            x_i = K.bias_add(x_i, b_i)
            x_f = K.bias_add(x_f, b_f)
            x_c = K.bias_add(x_c, b_c)
            x_o = K.bias_add(x_o, b_o)

        i = self.rec_activation(x_i + K.dot(h_tm1, self.rec_kernel[:, :self.units]))

        f = self.rec_activation(x_f + K.dot(h_tm1, self.rec_kernel[:, self.units:self.
↪units * 2]))

        c = f * c_tm1 + i * self.activation(x_c + K.dot(h_tm1, self.rec_kernel[:, self.
```

(continues on next page)

(continued from previous page)

```
↪units * 2:self.units * 3]))

        o = self.rec_activation(x_o + K.dot(h_tm1, self.rec_kernel[:, self.units * 3:]))

        h = o * self.activation(c)

        return h, [h, c]

    def build(self, input_shape):

        input_dim = input_shape[-1]

        self.bias = self.add_weight(
            shape=(self.units * 4,),
            name='bias',
            initializer="zeros")

        self.kernel = self.add_weight(
            shape=(input_dim, self.units * 4),
            name='kernel',
            initializer="glorot_uniform")

        self.rec_kernel = self.add_weight(
            shape=(self.units, self.units * 4),
            name='recurrent_kernel',
            initializer="orthogonal")

        self.built = True
        return
```

```
seed_all(313)

seed_all(313)
lstm = LSTMWithBias(lstm_units)
h1 = lstm(inputs_tf)
h1_sum = tf.reduce_sum(h1)
print(K.eval(h1_sum))
```

```
25.964832
```

```
seed_all(313)
lstm = KLSTM(lstm_units,
             recurrent_activation="sigmoid",
             unit_forget_bias=False)
h2 = lstm(inputs_tf)
h2_sum = tf.reduce_sum(h2)
print(K.eval(h2_sum))
```

```
25.964832
```

1.6.2 implementing temporal loop

```
# so far we had been using k.rnn() function to implement the temporal (for) loop
# of LSTM. Let's see what is inside it!

class LSTM(Layer):
    """A simplified implementation of LSTM layer with keras
    """

    def __init__(self, units, use_bias=True, **kwargs):

        super(LSTM, self).__init__(**kwargs)

        self.activation = tf.nn.tanh
        self.rec_activation = tf.nn.sigmoid
        self.units = units
        self.use_bias = use_bias

    def call(self, inputs, **kwargs):

        initial_state = tf.zeros((10, self.units))  # todo

        inputs = tf.transpose(inputs, [1, 0, 2])
        lookback, _, _ = inputs.shape
        state = [initial_state, initial_state]

        outputs, states = [], []
        for time_step in range(lookback):

            _out, state = self.cell(inputs[time_step], state)

            outputs.append(_out)
            states.append(state)

        outputs = tf.stack(outputs)
        states = tf.stack(states)

        outputs = tf.transpose(outputs, [1, 0, 2])

        last_output = outputs[:, -1]

        return last_output

    def cell(self, inputs, states):

        h_tm1 = states[0]  # previous memory state
        c_tm1 = states[1]  # previous carry state

        k_i, k_f, k_c, k_o = array_ops.split(self.kernel, num_or_size_splits=4, axis=1)

        x_i = K.dot(inputs, k_i)
        x_f = K.dot(inputs, k_f)
        x_c = K.dot(inputs, k_c)
```

(continues on next page)

(continued from previous page)

```
        x_o = K.dot(inputs, k_o)

        if self.use_bias:
            b_i, b_f, b_c, b_o = array_ops.split(
                self.bias, num_or_size_splits=4, axis=0)
            x_i = K.bias_add(x_i, b_i)
            x_f = K.bias_add(x_f, b_f)
            x_c = K.bias_add(x_c, b_c)
            x_o = K.bias_add(x_o, b_o)

        i = self.rec_activation(x_i + K.dot(h_tm1, self.rec_kernel[:, :self.units]))

        f = self.rec_activation(x_f + K.dot(h_tm1, self.rec_kernel[:, self.units:self.
↪units * 2]))

        c = f * c_tm1 + i * self.activation(x_c + K.dot(h_tm1, self.rec_kernel[:, self.
↪units * 2:self.units * 3]))

        o = self.rec_activation(x_o + K.dot(h_tm1, self.rec_kernel[:, self.units * 3:]))

        h = o * self.activation(c)

        return h, [h, c]

    def build(self, input_shape):

        input_dim = input_shape[-1]

        self.bias = self.add_weight(
            shape=(self.units * 4,),
            name='bias',
            initializer="zeros")

        self.kernel = self.add_weight(
            shape=(input_dim, self.units * 4),
            name='kernel',
            initializer="glorot_uniform")

        self.rec_kernel = self.add_weight(
            shape=(self.units, self.units * 4),
            name='recurrent_kernel',
            initializer="orthogonal")

        self.built = True
        return
```

```
seed_all(313)
lstm = LSTM(lstm_units)
h1 = lstm(inputs_tf)
h1_sum = tf.reduce_sum(h1)
print(K.eval(h1_sum))
```

```
25.964832
```

```
seed_all(313)
lstm = KLSTM(lstm_units,
             recurrent_activation="sigmoid",
             unit_forget_bias=False)
h2 = lstm(inputs_tf)
h2_sum = tf.reduce_sum(h2)
print(K.eval(h2_sum))
```

```
25.964832
```

1.6.3 adding some more options

```
class LSTM(Layer):
    """A simplified implementation of LSTM layer with keras
    """

    def __init__(
            self,
            units,
            use_bias=True,
            kernel_initializer='glorot_uniform',
            recurrent_initializer='orthogonal',
            bias_initializer='zeros',
            return_state=False,
            return_sequences=False,
            time_major=False,
            ** kwargs
    ):

        super(LSTM, self).__init__(**kwargs)

        self.activation = tf.nn.tanh
        self.rec_activation = tf.nn.sigmoid
        self.units = units
        self.use_bias = use_bias
        self.kernel_initializer = kernel_initializer
        self.recurrent_initializer = recurrent_initializer
        self.bias_initializer = bias_initializer
        self.return_state = return_state
        self.return_sequences = return_sequences
        self.time_major=time_major

    def call(self, inputs, **kwargs):

        initial_state = tf.zeros((10, self.units))  # todo

        if not self.time_major:
            inputs = tf.transpose(inputs, [1, 0, 2])
```

(continues on next page)

(continued from previous page)

```
        lookback, _, _ = inputs.shape

        state = [initial_state, initial_state]

        outputs, states = [], []
        for time_step in range(lookback):
            _out, state = self.cell(inputs[time_step], state)

            outputs.append(_out)
            states.append(state)

        outputs = tf.stack(outputs)
        h_s = tf.stack([states[i][0] for i in range(lookback)])
        c_s = tf.stack([states[i][1] for i in range(lookback)])


        if not self.time_major:
            outputs = tf.transpose(outputs, [1, 0, 2])
            h_s = tf.transpose(h_s, [1, 0, 2])
            c_s = tf.transpose(c_s, [1, 0, 2])
            states = [h_s, c_s]
            last_output = outputs[:, -1]
        else:
            states = [h_s, c_s]
            last_output = outputs[-1]

        h = last_output

        if self.return_sequences:
            h = outputs

        if self.return_state:
            return h, states

        return h

    def cell(self, inputs, states):

        h_tm1 = states[0]  # previous memory state
        c_tm1 = states[1]  # previous carry state

        k_i, k_f, k_c, k_o = array_ops.split(self.kernel, num_or_size_splits=4, axis=1)

        x_i = K.dot(inputs, k_i)
        x_f = K.dot(inputs, k_f)
        x_c = K.dot(inputs, k_c)
        x_o = K.dot(inputs, k_o)

        if self.use_bias:
            b_i, b_f, b_c, b_o = array_ops.split(
                self.bias, num_or_size_splits=4, axis=0)
```

(continues on next page)

(continued from previous page)

```
            x_i = K.bias_add(x_i, b_i)
            x_f = K.bias_add(x_f, b_f)
            x_c = K.bias_add(x_c, b_c)
            x_o = K.bias_add(x_o, b_o)

        i = self.rec_activation(x_i + K.dot(h_tm1, self.rec_kernel[:, :self.units]))

        f = self.rec_activation(x_f + K.dot(h_tm1, self.rec_kernel[:, self.units:self.
↪units * 2]))

        c = f * c_tm1 + i * self.activation(x_c + K.dot(h_tm1, self.rec_kernel[:, self.
↪units * 2:self.units * 3]))

        o = self.rec_activation(x_o + K.dot(h_tm1, self.rec_kernel[:, self.units * 3:]))

        h = o * self.activation(c)

        return h, [h, c]

    def build(self, input_shape):

        input_dim = input_shape[-1]

        self.bias = self.add_weight(
            shape=(self.units * 4,),
            name='bias',
            initializer=self.bias_initializer)

        self.kernel = self.add_weight(
            shape=(input_dim, self.units * 4),
            name='kernel',
            initializer=self.kernel_initializer)

        self.rec_kernel = self.add_weight(
            shape=(self.units, self.units * 4),
            name='recurrent_kernel',
            initializer=self.recurrent_initializer)

        self.built = True
        return
```

```
seed_all(313)

lstm = LSTM(lstm_units, return_sequences=True)
h1 = lstm(inputs_tf)
h1_sum = tf.reduce_sum(h1)
print(K.eval(h1_sum))
```

```
115.15204
```

```
seed_all(313)
```

(continues on next page)

(continued from previous page)

```
lstm = KLSTM(lstm_units,
             recurrent_activation="sigmoid",
             unit_forget_bias=False,
             return_sequences=True
            )
h2 = lstm(inputs_tf)
h2_sum = tf.reduce_sum(h2)
print(K.eval(h2_sum))
```

```
115.15204
```

1.6.4 builing Model and training

```
# It is possible to use our vanilla LSTM as a layer in Keras Model.
```

```
seed_all(313)
inp = Input(batch_shape=(10, lookback_steps, num_inputs))
lstm = LSTM(8)(inp)
out = Dense(1)(lstm)
model = Model(inputs=inp, outputs=out)
model.compile(loss='mse')

xx = np.random.random((100, lookback_steps, num_inputs))
y = np.random.random((100, 1))
h = model.fit(x=xx, y=y, batch_size=10, epochs=10)
```

```
Epoch 1/10

 1/10 [==>............................] - ETA: 9s - loss: 0.2666
10/10 [==============================] - 1s 2ms/step - loss: 0.2250
Epoch 2/10

 1/10 [==>............................] - ETA: 0s - loss: 0.1784
10/10 [==============================] - 0s 2ms/step - loss: 0.1670
Epoch 3/10

 1/10 [==>............................] - ETA: 0s - loss: 0.1832
10/10 [==============================] - 0s 2ms/step - loss: 0.1329
Epoch 4/10

 1/10 [==>............................] - ETA: 0s - loss: 0.0520
10/10 [==============================] - 0s 2ms/step - loss: 0.1106
Epoch 5/10

 1/10 [==>............................] - ETA: 0s - loss: 0.1114
10/10 [==============================] - 0s 2ms/step - loss: 0.0976
Epoch 6/10

 1/10 [==>............................] - ETA: 0s - loss: 0.0948
```

(continues on next page)

(continued from previous page)

```
10/10 [==============================] - 0s 2ms/step - loss: 0.0920
Epoch 7/10

 1/10 [==>...........................] - ETA: 0s - loss: 0.1118
10/10 [==============================] - 0s 2ms/step - loss: 0.0903
Epoch 8/10

 1/10 [==>...........................] - ETA: 0s - loss: 0.0413
10/10 [==============================] - 0s 2ms/step - loss: 0.0900
Epoch 9/10

 1/10 [==>...........................] - ETA: 0s - loss: 0.1029
10/10 [==============================] - 0s 2ms/step - loss: 0.0893
Epoch 10/10

 1/10 [==>...........................] - ETA: 0s - loss: 0.0631
10/10 [==============================] - 0s 2ms/step - loss: 0.0886
```

```
print(np.sum(h.history['loss']))
```

```
1.183198243379593
```

```
# now compare the results by using original Keras LSTM i.e. KLSTM

seed_all(313)
inp = Input(batch_shape=(10, lookback_steps, num_inputs))
lstm = KLSTM(8,
             recurrent_activation="sigmoid",
             unit_forget_bias=False
             )(inp)
out = Dense(1)(lstm)
model = Model(inputs=inp, outputs=out)
model.compile(loss='mse')

xx = np.random.random((100, lookback_steps, num_inputs))
y = np.random.random((100, 1))
h = model.fit(x=xx, y=y, batch_size=10, epochs=10)
```

```
Epoch 1/10

 1/10 [==>...........................] - ETA: 10s - loss: 0.2666
10/10 [==============================] - 1s 2ms/step - loss: 0.2250
Epoch 2/10

 1/10 [==>...........................] - ETA: 0s - loss: 0.1784
10/10 [==============================] - 0s 2ms/step - loss: 0.1670
Epoch 3/10

 1/10 [==>...........................] - ETA: 0s - loss: 0.1832
10/10 [==============================] - 0s 2ms/step - loss: 0.1329
Epoch 4/10
```

(continues on next page)

(continued from previous page)

```
 1/10 [==>...........................] - ETA: 0s - loss: 0.0520
10/10 [==============================] - 0s 2ms/step - loss: 0.1106
Epoch 5/10

 1/10 [==>...........................] - ETA: 0s - loss: 0.1114
10/10 [==============================] - 0s 2ms/step - loss: 0.0976
Epoch 6/10

 1/10 [==>...........................] - ETA: 0s - loss: 0.0948
10/10 [==============================] - 0s 2ms/step - loss: 0.0920
Epoch 7/10

 1/10 [==>...........................] - ETA: 0s - loss: 0.1118
10/10 [==============================] - 0s 2ms/step - loss: 0.0903
Epoch 8/10

 1/10 [==>...........................] - ETA: 0s - loss: 0.0413
10/10 [==============================] - 0s 2ms/step - loss: 0.0900
Epoch 9/10

 1/10 [==>...........................] - ETA: 0s - loss: 0.1029
10/10 [==============================] - 0s 2ms/step - loss: 0.0893
Epoch 10/10

 1/10 [==>...........................] - ETA: 0s - loss: 0.0631
10/10 [==============================] - 0s 2ms/step - loss: 0.0886
```

```
print(np.sum(h.history['loss']))
```

```
1.183198221027851
```

Total running time of the script: (0 minutes 3.490 seconds)

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search