Machine Learning for Software Analysis (MLSA)

University of Florence -- IMT School for Advanced Studies Lucca

Fabio Pinelli

fabio.pinelli@imtlucca.it (mailto:fabio.pinelli@imtlucca.it)
IMT School for Advanced Studies Lucca

2025/2026
October, 21 2025

Outline

- Introduction to RNN
- Gated Recurrent Unit (GRU)
- Long Short-Term Memory (LSTM)

Sequences

- Last time we applied Deep Learning Networks to images and tabular data
- Another type of data comes from *an ordered sequence of data points sharing a label*
- This particular structure has a single dimension
- This structure can be exploited by Recurrent Neural Networks and many variants as well as 1D Convolutional Neural Networks.
- Example of sequence data:
  - Text (NLP)
  - Time-series
    - Trajectories
    - Audio
    - Sensors data
    - etc.

Recurrent Neural Networks (RNN)

- RNNs take advantage of the underlying structure of the data:
  - The order of the data points
- RNN can be used for classification tasks, predictions, regression, etc.

How they work schematically?

1. Data points are sequentially presented to the newtork
2. The data points modify the internal (so called **hidden**) state
3. The final hidden state is a *representation* of the full sequence

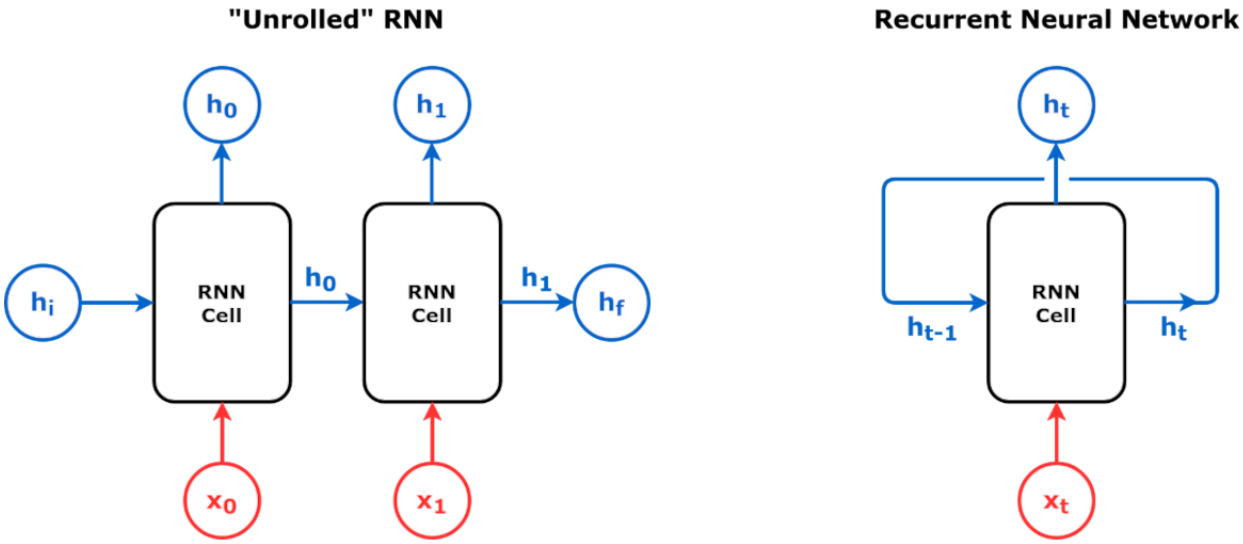
--> RNNs are all about producing a **hidden state** that best represents a **sequence**

Hidden state

The hidden state is a *simply* a **VECTOR** and its size is up to us. This means that we need to specify the number of the **Hidden dimension** (as for the CNN we need to specify the size of the kernel)

With these dimensions we specify the size of the vector that represents the Hidden State

RNN Architecture



RNN Architecture (2)

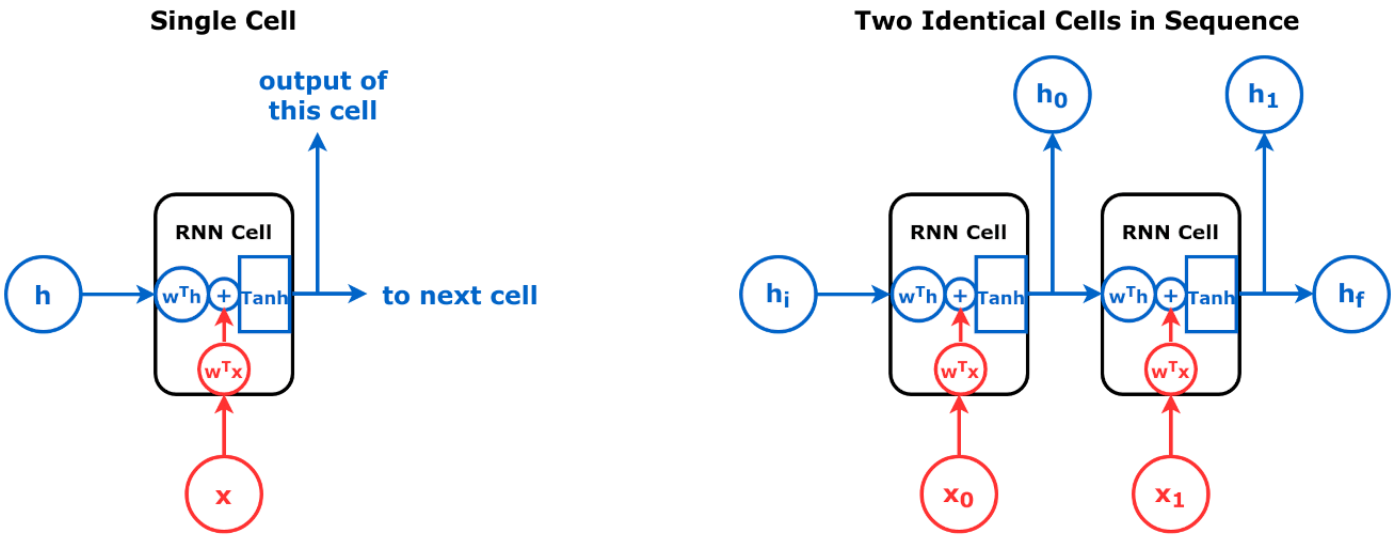
1. There is a initial Hidden State (h_i): it represents the state for the empty sequence;
2. A RNN cell takes two inputs: The *Hidden State* and a *Data Point* of the Sequence (x_0 and x_1)
    - The hidden state represents the state of the sequence *so far*
    - The two data points are two points of our input sequences
3. Two inputs are used to PRODUCE a new Hidden State (i.e., h_0 for the first data point)
4. The new hidden state is both the output of the current step and one of the input of the next step (see where h_0 is on the graph on the left)
5. If there is yet another data point in the sequence,
    - **THEN** we go back to step #2
    - **ELSE** the last hidden state (i.e., h_1) is the final hidden state of the whole RNN

RNN Architecture (3)

In reality the *unrolled* representation is misleading:

- There is only one cell
    - It learns a particular set of weights and biases
    - These will transform the inputs exactly the same way in every step of the sequence

RNN Architecture (4)



On the left figure we have a Single RNN Cell

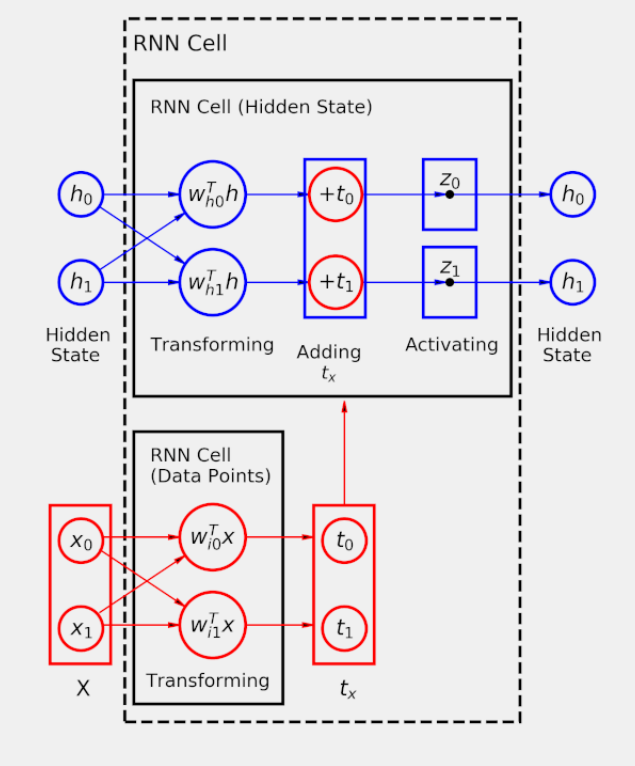
1. A linear layer to transform the hidden state (blue)
2. A linear layer to transform the data point (red)
3. An activation function applied to the SUM of both transformed inputs

Linear Layers transform the hidden state and the input.

The updated hidden state is both the output of this particular cell and the input of the "*next*" cell.

But there is not another **CELL** is the same cell over and over again (see right figure above). What does it mean?

- In the second step of the sequence:
  - The updated hidden state will run through the very same linear layer as the first hidden state
  - The second data point goes through the very same linear layer as the first data point



If we consider **2 Hidden Dimensions** the RNN looks like the figure above

- Two neurons are transforming the hidden state (blue)
- N RED Neurons transforming the data points are the same number of the hidden dimensions, the results of these two transformation need to be added together
- The data points can have a different dimension (in our example we have the input size equal to two but just for demonstration purposes)

Some example

In [ ]:
```
import numpy as np

import torch
import torch.optim as optim
import torch.nn as nn
import torch.nn.functional as F

from torch.utils.data import DataLoader, Dataset, random_split, TensorDataset
from torch.nn.utils import rnn as rnn_utils
```

In [ ]:
```
import numpy as np

def generate_sequences(n=128, variable_len=False, seed=13):
    basic_corners = np.array([[-1, -1], [-1, 1], [1, 1], [1, -1]])
    np.random.seed(seed)
    bases = np.random.randint(4, size=n)
    if variable_len:
        lengths = np.random.randint(3, size=n) + 2
    else:
        lengths = [4] * n
    directions = np.random.randint(2, size=n)
    points = [basic_corners[[(b + i) % 4 for i in range(4)]][slice(None, None, d*2-1)][:l] + np.random.randn
(l, 2) * 0.1 for b, d, l in zip(bases, directions, lengths)]
    return points, directions
```

In [ ]:
```
points, directions = generate_sequences(n=128, seed=13)
```

In [ ]:
```
points[0]
```

Out[ ]:
```
array([[ 1.03487506,  0.96613817],
       [ 0.80546093, -0.91690943],
       [-0.82507582, -0.94988627],
       [-0.86696831,  0.93424827]])
```

In [ ]:
```
n_features = 2
hidden_dim = 2

torch.manual_seed(19)
rnn_cell = nn.RNNCell(input_size=n_features, hidden_size=hidden_dim)
rnn_state = rnn_cell.state_dict()
rnn_state
```

Out[ ]:
```
OrderedDict([('weight_ih',
              tensor([[ 0.6627, -0.4245],
                      [ 0.5373,  0.2294]])),
             ('weight_hh',
              tensor([[-0.4015, -0.5385],
                      [-0.1956, -0.6835]])),
             ('bias_ih', tensor([0.4954, 0.6533])),
             ('bias_hh', tensor([-0.3565, -0.2904]))])
```

`weight_ih, bias_ih`

`i` stands for input, they represent the tensors of the RED neurons

`weight_hh, bias_hh`

`h` stands for hidden, they represent the tensors of the BLUE neurons

In [ ]:
```
linear_input = nn.Linear(n_features, hidden_dim)
linear_hidden = nn.Linear(hidden_dim, hidden_dim)

with torch.no_grad():
    linear_input.weight = nn.Parameter(rnn_state['weight_ih'])
    linear_input.bias = nn.Parameter(rnn_state['bias_ih'])
    linear_hidden.weight = nn.Parameter(rnn_state['weight_hh'])
    linear_hidden.bias = nn.Parameter(rnn_state['bias_hh'])
```

In [ ]:
```
'''
The initial hidden state representing the empty sequence.
It is initialized with all zeros (torch.zeros)
'''
initial_hidden = torch.zeros(1, hidden_dim)
initial_hidden
```

Out[ ]:
```
tensor([[0., 0.]])
```

In [ ]:
```
'''
Following the RNN architecture, we apply the hh linear layer
to  transform the hidden state
--> BLUE NEURONS
'''

th = linear_hidden(initial_hidden)
th
```

Out[ ]:
```
tensor([[-0.3565, -0.2904]], grad_fn=<AddmmBackward0>)
```

In [ ]:
```
X = torch.as_tensor(points[0]).float()
X
```

Out[ ]:
```
tensor([[ 1.0349,  0.9661],
        [ 0.8055, -0.9169],
        [-0.8251, -0.9499],
        [-0.8670,  0.9342]])
```

In [ ]:
```
tx = linear_input(X[0:1])
tx
```

Out[ ]:
```
tensor([[0.7712, 1.4310]], grad_fn=<AddmmBackward0>)
```

In [ ]:
```
adding = th + tx
adding
```

Out[ ]:
```
tensor([[0.4146, 1.1405]], grad_fn=<AddBackward0>)
```

In [ ]:
```
torch.tanh(adding)
```

Out[ ]:
```
tensor([[0.3924, 0.8146]], grad_fn=<TanhBackward0>)
```

In [ ]:
```
rnn_cell(X[0:1])
```

Out[ ]:
```
tensor([[0.3924, 0.8146]], grad_fn=<TanhBackward0>)
```

We need to remember that the RNN has two inputs:

- The data points
- The hidden state

We need to loop over the data points and provide the updated hidden state at each step:

In [ ]:
```
hidden = torch.zeros(1, hidden_dim)
for i in range(X.shape[0]):
    out = rnn_cell(X[i:i+1], hidden)
    print(out)
    hidden = out
```

```
tensor([[0.3924, 0.8146]], grad_fn=<TanhBackward0>)
tensor([[ 0.4347, -0.0481]], grad_fn=<TanhBackward0>)
tensor([[-0.1521, -0.3367]], grad_fn=<TanhBackward0>)
tensor([[-0.5297,  0.3551]], grad_fn=<TanhBackward0>)
```

The `tensor([[-0.5297,  0.3551]])` is the last hidden state and it is the representation of the full sequence

Looping over the data points in a sequence can require a lot of work and time...

Instead of a RNN cell we can use **RNN layer** that

- Takes care of the hidden state no matter how long the input sequence is
- We need to take care about the **shapes** of the **INPUTS** and **OUTPUTS**

- `input_size` : is the number of features in each data point of the sequence
- `hidden_size` : is the number of hidden dimensions

In [ ]:
```
n_features = 2
hidden_dim = 2

torch.manual_seed(19)
rnn = nn.RNN(input_size=n_features, hidden_size=hidden_dim)
rnn.state_dict()
```

Out[ ]:
```
OrderedDict([('weight_ih_l0',
              tensor([[ 0.6627, -0.4245],
                      [ 0.5373,  0.2294]])),
             ('weight_hh_l0',
              tensor([[-0.4015, -0.5385],
                      [-0.1956, -0.6835]])),
             ('bias_ih_l0', tensor([0.4954, 0.6533])),
             ('bias_hh_l0', tensor([-0.3565, -0.2904]))])
```

Another input parameter:

- `batch_first` : if `True` **ONLY** the input and output tensors are provided as (batch, seq, feature)
- ONLY --> the hidden state will never be batch first

Shapes

INPUTS

- The input tensor containing the sequence you want to run through the RNN:
  - The **Default** shape is **sequence-first** (sequence length, batch size, number of features) (**L,N,F**)
  - Choosing `batch_first = True` (**N,L,F**), this is what we get from a data loader
- The initial hidden state
  - tensor with shape (**1,N,H**)

OUTPUTS

- The output tensor contains the hidden states corresponding to the outpus of its RNN cells for all steps in the sequence:
  - Output sensor shape (**L,N,H**)
  - With `batch_first` it permutes the first dimension obtaining (**N,L,H**)
- The final hidden state follows the same rules as the initial hidden state

In [ ]:
```
'''
Let's create a batch with 3 sequences with 4 data points, each data point
is described by 2 coordinates
The shape is (3,4,2) -> batch_first (N,L,F)
Similarly to what we get from a data loader
'''

batch = torch.as_tensor(np.array(points[:3])).float()
batch.shape
```

Out[ ]:
```
torch.Size([3, 4, 2])
```

In [ ]:
```
permuted_batch = batch.permute(1, 0, 2)
permuted_batch.shape
```

Out[ ]:
```
torch.Size([4, 3, 2])
```

In [ ]:
```
torch.manual_seed(19)
rnn = nn.RNN(input_size=n_features, hidden_size=hidden_dim)
out, final_hidden = rnn(permuted_batch)
out.shape, final_hidden.shape
```

Out[ ]:
```
(torch.Size([4, 3, 2]), torch.Size([1, 3, 2]))
```

The last element of the output IS the final hidden state, let's check

In [ ]:
```
(out[-1] == final_hidden).all()
```

Out[ ]:
```
tensor(True)
```

In [ ]:
```
out[-1]
```

Out[ ]:
```
tensor([[-0.5297,  0.3551],
        [ 0.3142, -0.1232],
        [-0.2095,  0.4354]], grad_fn=<SelectBackward0>)
```

In [ ]:
```
print(out[-1])
print(final_hidden)
```

```
tensor([[-0.5297,  0.3551],
        [ 0.3142, -0.1232],
        [-0.2095,  0.4354]], grad_fn=<SelectBackward0>)
tensor([[[-0.5297,  0.3551],
         [ 0.3142, -0.1232],
         [-0.2095,  0.4354]]], grad_fn=<StackBackward0>)
```

In [ ]:
```
batch_hidden = final_hidden.permute(1, 0, 2)
batch.shape, batch_hidden.shape
```

Out[ ]:
```
(torch.Size([3, 4, 2]), torch.Size([3, 1, 2]))
```

In [ ]:
```
torch.manual_seed(19)
rnn_batch_first = nn.RNN(input_size=n_features, hidden_size=hidden_dim, batch_first=True)
out, final_hidden = rnn_batch_first(batch)
out.shape, final_hidden.shape
```

Out[ ]:
```
(torch.Size([3, 4, 2]), torch.Size([1, 3, 2]))
```

Two distinct shapes as result:

- batch_first (N,L,H) for the output
- sequence first (1,N,H) for the final hidden state

This might lead to confusion... but... Most of the time we need to handle the input, and data loaders typically provide batch_first shapes, therefore we will use `batch_first = True` .

The model

In [ ]:
```
test_points, test_directions = generate_sequences(seed=19)
train_data = TensorDataset(torch.as_tensor(points).float(),
                           torch.as_tensor(directions).view(-1, 1).float())
test_data = TensorDataset(torch.as_tensor(test_points).float(),
                          torch.as_tensor(test_directions).view(-1, 1).float())

train_loader = DataLoader(train_data, batch_size=16, shuffle=True)
test_loader = DataLoader(test_data, batch_size=16)
```

```
/tmp/ipython-input-1248862510.py:2: UserWarning: Creating a tensor from a list of numpy.ndarrays is extremely slow. Please consider converting the list to a single numpy.ndarray with numpy.array() before converting to a tensor. (Triggered internally at /pytorch/torch/csrc/utils/tensor_new.cpp:253.)
  train_data = TensorDataset(torch.as_tensor(points).float(),
```

In [ ]:
```
test_directions
```

Out[ ]:
```
array([1, 0, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 0, 1, 0, 0, 1, 0, 1, 1, 0, 1,
       1, 1, 1, 0, 0, 1, 1, 0, 1, 1, 1, 0, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1,
       1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1, 1, 0,
       0, 0, 0, 1, 0, 1, 1, 0, 1, 0, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1,
       1, 0, 1, 1, 0, 0, 1, 0, 1, 0, 1, 1, 0, 1, 0, 1, 1, 1, 0, 0, 1, 0,
       1, 1, 1, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 1, 1])
```

In [ ]:
```
class SquareModel(nn.Module):
    def __init__(self, n_features, hidden_dim, n_outputs):
        super(SquareModel, self).__init__()
        self.hidden_dim = hidden_dim
        self.n_features = n_features
        self.n_outputs = n_outputs
        self.hidden = None
        # Simple RNN
        self.basic_rnn = nn.RNN(self.n_features, self.hidden_dim, batch_first=True)
        # Classifier to produce as many logits as outputs
        self.classifier = nn.Linear(self.hidden_dim, self.n_outputs)

    def forward(self, X):
        # X is batch first (N, L, F)
        # output is (N, L, H)
        # final hidden state is (1, N, H)
        batch_first_output, self.hidden = self.basic_rnn(X)

        # only last item in sequence (N, 1, H)
        last_output = batch_first_output[:, -1]
        # classifier will output (N, 1, n_outputs)
        out = self.classifier(last_output)

        # final output is (N, n_outputs)
        return out.view(-1, self.n_outputs)
```

We want to understand if the squares are drawn in a **clock-wise** or **anticlock-wise** direction

1. We use a simple RNN to obtain the final hidden state
2. The final hidden state represents the full sequence
3. Use it to train a classifier layer

In [ ]:
```
class SquareModel(nn.Module):
    def __init__(self, n_features, hidden_dim, n_outputs):
        super(SquareModel, self).__init__()
        self.hidden_dim = hidden_dim
        self.n_features = n_features
        self.n_outputs = n_outputs
        self.hidden = None
        # Simple RNN
        self.basic_rnn = nn.RNN(self.n_features, self.hidden_dim, batch_first=True)
        # Classifier to produce as many logits as outputs
        self.classifier = nn.Linear(self.hidden_dim, self.n_outputs)

    def forward(self, X):
        # X is batch first (N, L, F)
        # output is (N, L, H)
        # final hidden state is (1, N, H)
        batch_first_output, self.hidden = self.basic_rnn(X)

        # only last item in sequence (N, 1, H)
        last_output = batch_first_output[:, -1]
        # classifier will output (N, 1, n_outputs)
        out = self.classifier(last_output)

        # final output is (N, n_outputs)
        return out.view(-1, self.n_outputs)
```

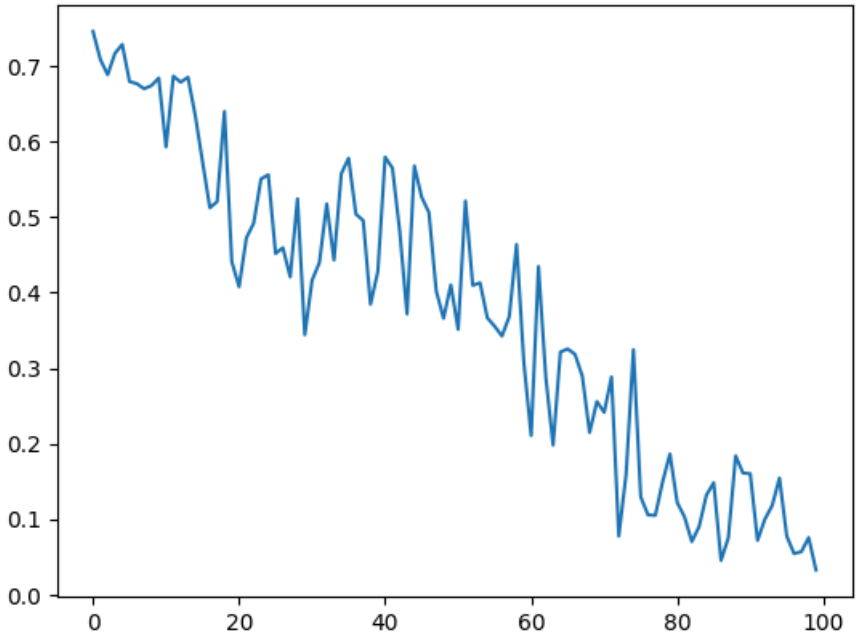
In [ ]:
```
torch.manual_seed(21)
model = SquareModel(n_features=2, hidden_dim=2, n_outputs=1)
loss = nn.BCEWithLogitsLoss()
optimizer = optim.Adam(model.parameters(), lr=0.01)
```

In [ ]:
```
epochs = 100
losses_RNN = []
for i in range(epochs):
    for seq, labels in train_loader:
        optimizer.zero_grad()
        model.hidden = (torch.zeros(1, 1, model.hidden_dim),
                        torch.zeros(1, 1, model.hidden_dim))

        y_pred = model(seq)

        single_loss = loss(y_pred, labels)
        single_loss.backward()
        optimizer.step()
    losses_RNN.append(single_loss.item())
    if i%25 == 1:
        print(f'epoch: {i:3} loss: {single_loss.item():10.8f}')

print(f'epoch: {i:3} loss: {single_loss.item():10.10f}')
```

```
epoch:   1 loss: 0.70908552
epoch:  26 loss: 0.45952681
epoch:  51 loss: 0.52138358
epoch:  76 loss: 0.10545339
epoch:  99 loss: 0.0324295834
```

In [ ]:
```
import matplotlib.pyplot as plt
plt.plot(losses_RNN)
```

Out[ ]:
```
[<matplotlib.lines.Line2D at 0x7e76d151f170>]
```



In [ ]:
```
def correct(model, x, y, threshold=.5):
      model.eval()
      yhat = model(x)
      y
      model.train()

      # We get the size of the batch and the number of classes
      # (only 1, if it is binary)
      n_samples, n_dims = yhat.shape
      if n_dims > 1:
          # In a multiclass classification, the biggest logit
          # always wins, so we don't bother getting probabilities

          # This is PyTorch's version of argmax,
          # but it returns a tuple: (max value, index of max value)
          _, predicted = torch.max(yhat, 1)
      else:
          n_dims += 1
          # In binary classification, we NEED to check if the
          # last layer is a sigmoid (and then it produces probs)
          if isinstance(model, nn.Sequential) and \
              isinstance(model[-1], nn.Sigmoid):
              predicted = (yhat > threshold).long()
          # or something else (logits), which we need to convert
          # using a sigmoid
          else:
              predicted = (torch.sigmoid(yhat) > threshold).long()

      # How many samples got classified correctly for each class
      result = []
      for c in range(n_dims):
          n_class = (y == c).sum().item()
          n_correct = (predicted[y == c] == c).sum().item()
          result.append((n_correct, n_class))
      return torch.tensor(result)
```

In [ ]:
```
final_results = []
for x, y in test_loader:
  res = correct(model, x,y)
  final_results.append(res)
```

In [ ]:
```
final_results
```

Out[ ]:
```
[tensor([[ 6,  6],
         [10, 10]]),
 tensor([[ 5,  5],
         [11, 11]]),
 tensor([[ 4,  4],
         [12, 12]]),
 tensor([[6, 7],
         [9, 9]]),
 tensor([[8, 8],
         [8, 8]]),
 tensor([[8, 8],
         [8, 8]]),
 tensor([[ 5,  6],
         [10, 10]]),
 tensor([[8, 9],
         [7, 7]])]
```

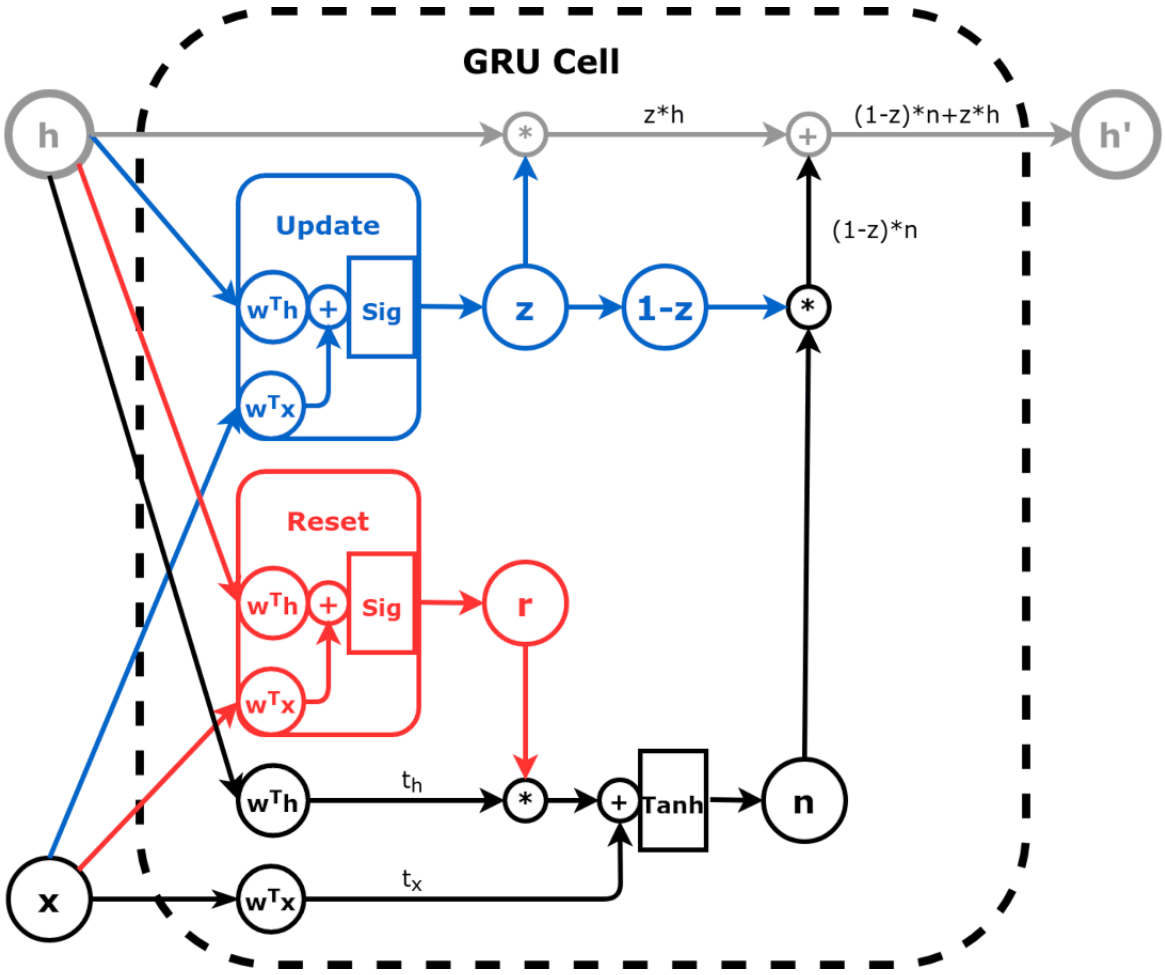
Gated Recurrent Units (GRUs)

- What if the previous hidden state contains more information than the newly computed one?
- What if the data point adds more information than the previous hidden state had?

The answer is:

- Gated Recurrent Units
  - Weighted average between **new** and **old** hidden states (z parameter to control the weights)
  - Scale the hidden state (t_h) by a factor r before adding to t_n, and this will be multiplicated by $(1 - z)$
  - r: *reset* gate, values between 0 and 1
  - z *update* gate, values between 0 and 1
  - Every gate produces a vector of values with a size corresponding to the number of hidden dimensions
  - Since gates produce vectors, operations involving them are element-wise multiplications.

$$
\begin{aligned}
\text{RNN: } h' &= \tanh(t_h + t_x) \\
\text{GRU: } h' &= \underbrace{\underbrace{\tanh(r * t_{hn} + t_{xn})}_{n} * (1 - z) + h * z}_{\text{weighted average of } n \text{ and } h}
\end{aligned}
$$



The GRUs try to solve the same problmes resolved by RNN, but in a better way.

Here we have 4 Linear Layers in this rapresentation of the GRU (one layer for each $w^t x$ or w^t h)

Code

In [ ]:
```
n_features = 2
hidden_dim = 2

torch.manual_seed(17)
gru_cell = nn.GRUCell(input_size=n_features, hidden_size=hidden_dim)
gru_state = gru_cell.state_dict()
gru_state
```

Out[ ]:
```
OrderedDict([('weight_ih',
              tensor([[-0.0930,  0.0497],
                      [ 0.4670, -0.5319],
                      [-0.6656,  0.0699],
                      [-0.1662,  0.0654],
                      [-0.0449, -0.6828],
                      [-0.6769, -0.1889]])),
             ('weight_hh',
              tensor([[-0.4167, -0.4352],
                      [-0.2060, -0.3989],
                      [-0.7070, -0.5083],
                      [ 0.1418,  0.0930],
                      [-0.5729, -0.5700],
                      [-0.1818, -0.6691]])),
             ('bias_ih',
              tensor([-0.4316,  0.4019,  0.1222, -0.4647, -0.5578,  0.4493])),
             ('bias_hh',
              tensor([-0.6800,  0.4422, -0.3559, -0.0279,  0.6553,  0.2918]))])
```

In [ ]:
```
Wx, bx = gru_state['weight_ih'], gru_state['bias_ih']
Wh, bh = gru_state['weight_hh'], gru_state['bias_hh']

print(Wx.shape, Wh.shape)
print(bx.shape, bh.shape)
```

```
torch.Size([6, 2]) torch.Size([6, 2])
torch.Size([6]) torch.Size([6])
```

In [ ]:
```
Wxr, Wxz, Wxn = Wx.split(hidden_dim, dim=0)
bxr, bxz, bxn = bx.split(hidden_dim, dim=0)

Whr, Whz, Whn = Wh.split(hidden_dim, dim=0)
bhr, bhz, bhn = bh.split(hidden_dim, dim=0)

Wxr, bxr
```

Out[ ]:
```
(tensor([[-0.0930,  0.0497],
         [ 0.4670, -0.5319]]),
 tensor([-0.4316,  0.4019]))
```

In [ ]:
```
def linear_layers(Wx, bx, Wh, bh):
    hidden_dim, n_features = Wx.size()
    lin_input = nn.Linear(n_features, hidden_dim)
    lin_input.load_state_dict({'weight': Wx, 'bias': bx})
    lin_hidden = nn.Linear(hidden_dim, hidden_dim)
    lin_hidden.load_state_dict({'weight': Wh, 'bias': bh})
    return lin_hidden, lin_input

r_hidden, r_input = linear_layers(Wxr, bxr, Whr, bhr) # reset gate - red
z_hidden, z_input = linear_layers(Wxz, bxz, Whz, bhz) # update gate - blue
n_hidden, n_input = linear_layers(Wxn, bxn, Whn, bhn) # candidate state - black
```

In [ ]:
```
def reset_gate(h, x):
    thr = r_hidden(h)
    txr = r_input(x)
    r = torch.sigmoid(thr + txr)
    return r  # red

def update_gate(h, x):
    thz = z_hidden(h)
    txz = z_input(x)
    z = torch.sigmoid(thz + txz)
    return z  # blue

def candidate_n(h, x, r):
    thn = n_hidden(h)
    txn = n_input(x)
    n = torch.tanh(r * thn + txn)
    return n  # black
```

In [ ]:
```
initial_hidden = torch.zeros(1, hidden_dim)
X = torch.as_tensor(points[0]).float()
first_corner = X[0:1]
```

In [ ]:
```
r = reset_gate(initial_hidden, first_corner)
r
```

Out[ ]:
```
tensor([[0.2387, 0.6928]], grad_fn=<SigmoidBackward0>)
```

In [ ]:
```
n = candidate_n(initial_hidden, first_corner, r)
n
```

Out[ ]:
```
tensor([[-0.8032, -0.2275]], grad_fn=<TanhBackward0>)
```

In [ ]:
```
z = update_gate(initial_hidden, first_corner)
z
```

Out[ ]:
```
tensor([[0.2984, 0.3540]], grad_fn=<SigmoidBackward0>)
```

In [ ]:
```
h_prime = n*(1-z) + initial_hidden*z
h_prime
```

Out[ ]:
```
tensor([[-0.5635, -0.1470]], grad_fn=<AddBackward0>)
```

In [ ]:
```
gru_cell(first_corner)
```

Out[ ]:
```
tensor([[-0.5635, -0.1470]], grad_fn=<AddBackward0>)
```

In [ ]:
```
'''
The only change in the code is RNN in GRU
'''


class SquareModelGRU(nn.Module):
    def __init__(self, n_features, hidden_dim, n_outputs):
        super(SquareModelGRU, self).__init__()
        self.hidden_dim = hidden_dim
        self.n_features = n_features
        self.n_outputs = n_outputs
        self.hidden = None
        # Simple GRU
        self.basic_rnn = nn.GRU(self.n_features, self.hidden_dim, batch_first=True) # only difference from RNN
to GRU
        # Classifier to produce as many logits as outputs
        self.classifier = nn.Linear(self.hidden_dim, self.n_outputs)

    def forward(self, X):
        # X is batch first (N, L, F)
        # output is (N, L, H)
        # final hidden state is (1, N, H)
        batch_first_output, self.hidden = self.basic_rnn(X)

        # only last item in sequence (N, 1, H)
        last_output = batch_first_output[:, -1]
        # classifier will output (N, 1, n_outputs)
        out = self.classifier(last_output)

        # final output is (N, n_outputs)
        return out.view(-1, self.n_outputs)
```

In [ ]:
```
torch.manual_seed(21)
modelGRU = SquareModelGRU(n_features=2, hidden_dim=2, n_outputs=1)
loss = nn.BCEWithLogitsLoss()
optimizer = optim.Adam(modelGRU.parameters(), lr=0.01)
```

In [ ]:
```
epochs = 100
losses_GRU = []
for i in range(epochs):
    for seq, labels in train_loader:
        optimizer.zero_grad()
        y_pred = modelGRU(seq)

        single_loss = loss(y_pred, labels)
        single_loss.backward()
        optimizer.step()
    losses_GRU.append(single_loss.item())
    if i%25 == 1:
        print(f'epoch: {i:3} loss: {single_loss.item():10.8f}')

print(f'epoch: {i:3} loss: {single_loss.item():10.10f}')
```

```
epoch:   1 loss: 0.69928366
epoch:  26 loss: 0.04697567
epoch:  51 loss: 0.03366457
epoch:  76 loss: 0.00736067
epoch:  99 loss: 0.0052382285
```

In [ ]:
```
final_results = []
for x, y in test_loader:
  res = correct(modelGRU, x,y)
  final_results.append(res)
```

In [ ]:
```
final_results
```

Out[ ]:
```
[tensor([[ 6,  6],
         [10, 10]]),
 tensor([[ 5,  5],
         [11, 11]]),
 tensor([[ 4,  4],
         [12, 12]]),
 tensor([[7, 7],
         [9, 9]]),
 tensor([[8, 8],
         [8, 8]]),
 tensor([[8, 8],
         [8, 8]]),
 tensor([[ 6,  6],
         [10, 10]]),
 tensor([[9, 9],
         [7, 7]])]
```

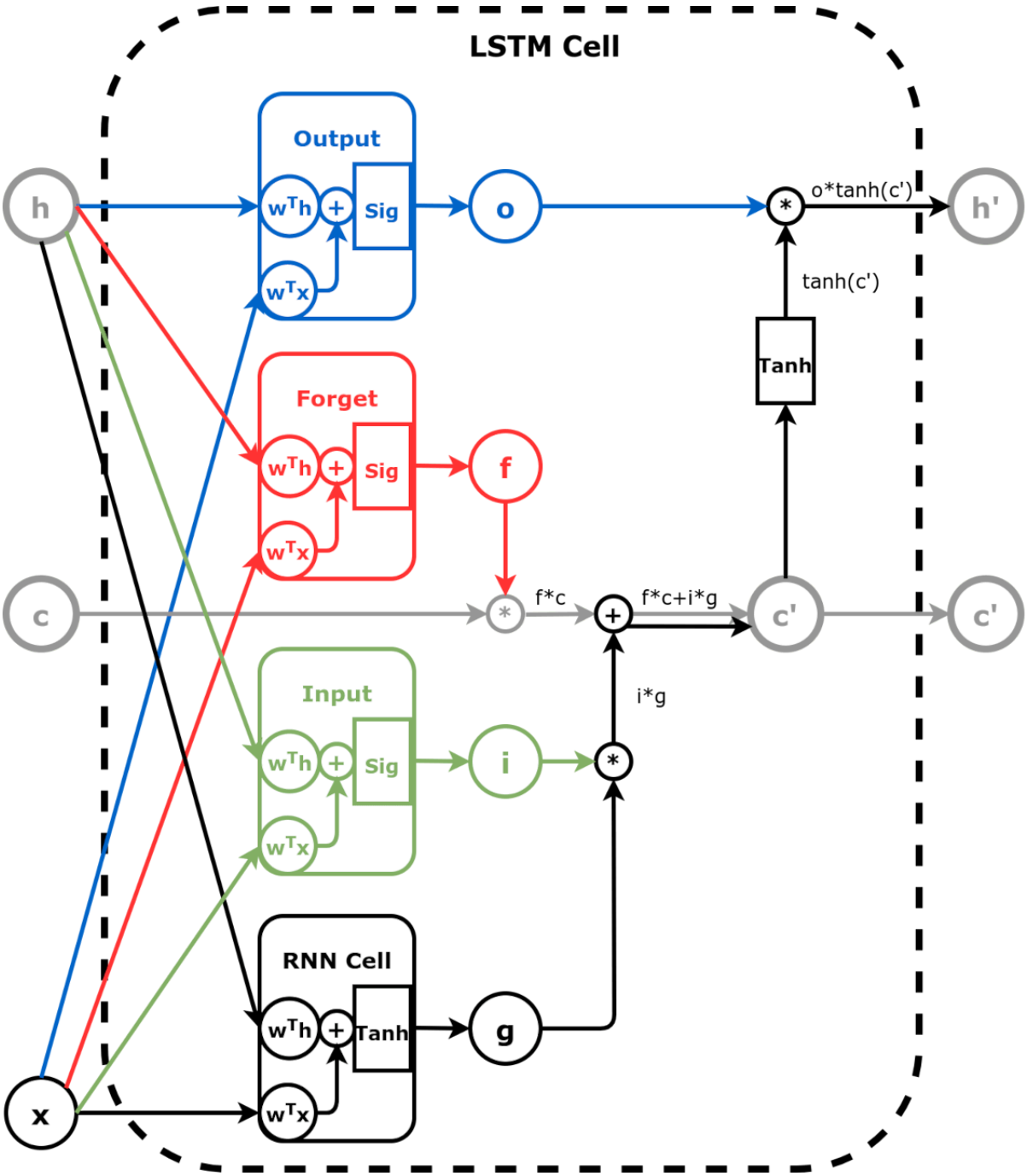
Long Short-Term Memory (LSTM)

LSTM uses two states instead of one:

- The hidden state (h) (bounded [1,-1] by $tanh$)
- The cell state (c) (unbounded)

In particular, we have:

- the candidate hidden state (g)
- the new cell state (c') computed as the weighted sum between the old cell state (c) and the new candidate hidden state (g)
- the weighted sum is "controlled" by two gates:
  - input gate
  - forget gate
- the new hidden state (h') is the bounded "version", (i.e., $tanh$) of the new cell state (c')
- the output gate "control" the new hidden state



- RED forget gate
- BLUE output gate
- GREEN input gate
- BLACK Candidate hidden state
- GREY old cell state

In [ ]:
```
'''
Let's try to replicate how the LSTM works
and then check the model
'''

n_features = 2
hidden_dim = 2

torch.manual_seed(17)
lstm_cell = nn.LSTMCell(input_size=n_features, hidden_size=hidden_dim)
lstm_state = lstm_cell.state_dict()
lstm_state
```

Out[ ]:
```
OrderedDict([('weight_ih',
              tensor([[-0.0930,  0.0497],
                      [ 0.4670, -0.5319],
                      [-0.6656,  0.0699],
                      [-0.1662,  0.0654],
                      [-0.0449, -0.6828],
                      [-0.6769, -0.1889],
                      [-0.4167, -0.4352],
                      [-0.2060, -0.3989]])),
             ('weight_hh',
              tensor([[-0.7070, -0.5083],
                      [ 0.1418,  0.0930],
                      [-0.5729, -0.5700],
                      [-0.1818, -0.6691],
                      [-0.4316,  0.4019],
                      [ 0.1222, -0.4647],
                      [-0.5578,  0.4493],
                      [-0.6800,  0.4422]])),
             ('bias_ih',
              tensor([-0.3559, -0.0279,  0.6553,  0.2918,  0.4007,  0.3262, -0.0778, -0.3002])),
             ('bias_hh',
              tensor([-0.3991, -0.3200,  0.3483, -0.2604, -0.1582,  0.5558,  0.5761, -0.3919]))])
```

In [ ]:
```
Wx, bx = lstm_state['weight_ih'], lstm_state['bias_ih']
Wh, bh = lstm_state['weight_hh'], lstm_state['bias_hh']

Wx.shape, bx.shape
```

Out[ ]:
```
(torch.Size([8, 2]), torch.Size([8]))
```

In [ ]:
```
Wx, bx = lstm_state['weight_ih'], lstm_state['bias_ih']
Wh, bh = lstm_state['weight_hh'], lstm_state['bias_hh']

# Split weights and biases for data points
Wxi, Wxf, Wxg, Wxo = Wx.split(hidden_dim, dim=0)
bxi, bxf, bxg, bxo = bx.split(hidden_dim, dim=0)
# Split weights and biases for hidden state
Whi, Whf, Whg, Who = Wh.split(hidden_dim, dim=0)
bhi, bhf, bhg, bho = bh.split(hidden_dim, dim=0)

# Creates linear layers for the components
i_hidden, i_input = linear_layers(Wxi, bxi, Whi, bhi) # input gate - green
f_hidden, f_input = linear_layers(Wxf, bxf, Whf, bhf) # forget gate - red
o_hidden, o_input = linear_layers(Wxo, bxo, Who, bho) # output gate - blue
```

In [ ]:
```
g_cell = nn.RNNCell(n_features, hidden_dim) # black
g_cell.load_state_dict({'weight_ih': Wxg, 'bias_ih': bxg,
                        'weight_hh': Whg, 'bias_hh': bhg})
```

Out[ ]:
```
<All keys matched successfully>
```

In [ ]:
```
def forget_gate(h, x):
    thf = f_hidden(h)
    txf = f_input(x)
    f = torch.sigmoid(thf + txf)
    return f  # red

def output_gate(h, x):
    tho = o_hidden(h)
    txo = o_input(x)
    o = torch.sigmoid(tho + txo)
    return o  # blue

def input_gate(h, x):
    thi = i_hidden(h)
    txi = i_input(x)
    i = torch.sigmoid(thi + txi)
    return i  # green
```

In [ ]:
```
initial_hidden = torch.zeros(1, hidden_dim)
initial_cell = torch.zeros(1, hidden_dim)

X = torch.as_tensor(points[0]).float()
first_corner = X[0:1]
```

In [ ]:
```
g = g_cell(first_corner)
i = input_gate(initial_hidden, first_corner)
gated_input = g * i
gated_input
```

Out[ ]:
```
tensor([[-0.1340, -0.0004]], grad_fn=<MulBackward0>)
```

In [ ]:
```
f = forget_gate(initial_hidden, first_corner)
gated_cell = initial_cell * f
gated_cell
```

Out[ ]:
```
tensor([[0., 0.]], grad_fn=<MulBackward0>)
```

In [ ]:
```
c_prime = gated_cell + gated_input
c_prime
```

Out[ ]:
```
tensor([[-0.1340, -0.0004]], grad_fn=<AddBackward0>)
```

In [ ]:
```
o = output_gate(initial_hidden, first_corner)
h_prime = o * torch.tanh(c_prime)
h_prime
```

Out[ ]:
```
tensor([[-5.4936e-02, -8.3816e-05]], grad_fn=<MulBackward0>)
```

In [ ]:
```
(h_prime, c_prime)
```

Out[ ]:
```
(tensor([[-5.4936e-02, -8.3816e-05]], grad_fn=<MulBackward0>),
 tensor([[-0.1340, -0.0004]], grad_fn=<AddBackward0>))
```

In [ ]:
```
lstm_cell(first_corner)
```

Out[ ]:
```
(tensor([[-5.4936e-02, -8.3816e-05]], grad_fn=<MulBackward0>),
 tensor([[-0.1340, -0.0004]], grad_fn=<AddBackward0>))
```

LSTM Layer

It takes care of the hidden and cell states.

LSTMs return two states: **hidden** and **cell** with the same shape

In [ ]:
```
'''
The main differences:
RNN-GRU --> LSTM
Two ouputs self.cell
'''


class SquareModelLSTM(nn.Module):
    def __init__(self, n_features, hidden_dim, n_outputs):
        super(SquareModelLSTM, self).__init__()
        self.hidden_dim = hidden_dim
        self.n_features = n_features
        self.n_outputs = n_outputs
        self.hidden = None
        self.cell = None
        # Simple LSTM
        self.basic_rnn = nn.LSTM(self.n_features, self.hidden_dim, batch_first=True)
        # Classifier to produce as many logits as outputs
        self.classifier = nn.Linear(self.hidden_dim, self.n_outputs)

    def forward(self, X):
        # X is batch first (N, L, F)
        # output is (N, L, H)
        # final hidden state is (1, N, H)
        # final cell state is (1, N, H)
        batch_first_output, (self.hidden, self.cell) = self.basic_rnn(X)

        # only last item in sequence (N, 1, H)
        last_output = batch_first_output[:, -1]
        # classifier will output (N, 1, n_outputs)
        out = self.classifier(last_output)

        # final output is (N, n_outputs)
        return out.view(-1, self.n_outputs)
```

In [ ]:
```
torch.manual_seed(21)
modelLSTM = SquareModelLSTM(n_features=2, hidden_dim=2, n_outputs=1)
loss = nn.BCEWithLogitsLoss()
optimizer = optim.Adam(modelLSTM.parameters(), lr=0.01)
```

In [ ]:
```
epochs = 100
losses_LSTM = []
for i in range(epochs):
    for seq, labels in train_loader:
        optimizer.zero_grad()
        y_pred = modelLSTM(seq)

        single_loss = loss(y_pred, labels)
        single_loss.backward()
        optimizer.step()
    losses_LSTM.append(single_loss.item())
    if i%25 == 1:
        print(f'epoch: {i:3} loss: {single_loss.item():10.8f}')

print(f'epoch: {i:3} loss: {single_loss.item():10.10f}')
```

```
epoch:   1 loss: 0.68456638
epoch:  26 loss: 0.05043306
epoch:  51 loss: 0.01348161
epoch:  76 loss: 0.00767727
epoch:  99 loss: 0.0061499123
```

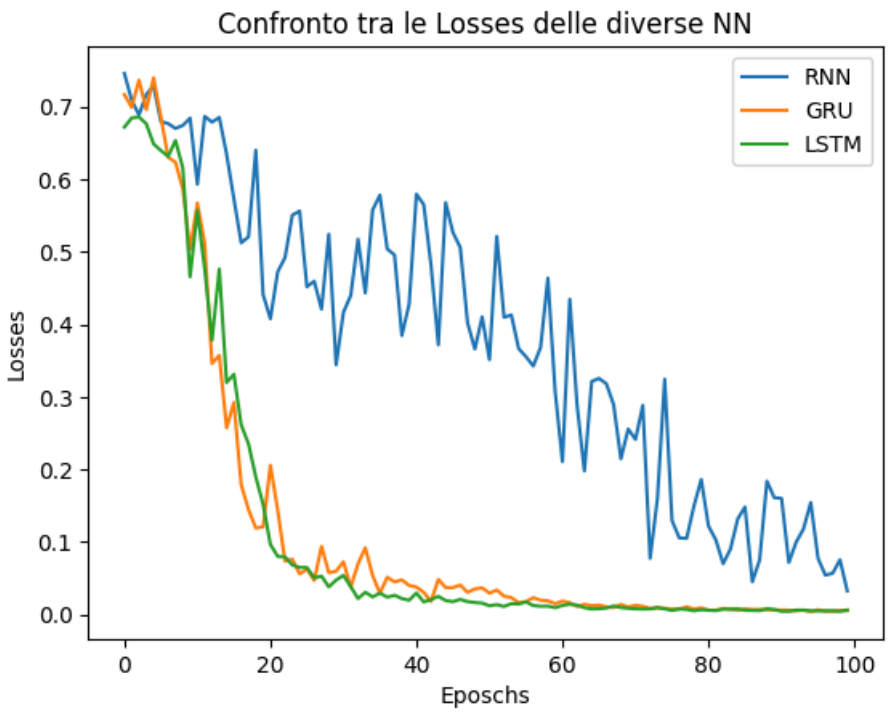
In [ ]:
```
final_results = []
for x, y in test_loader:
  res = correct(modelLSTM, x,y)
  final_results.append(res)
```

In [ ]:
```
final_results
```

Out[ ]:
```
[tensor([[ 6,  6],
         [10, 10]]),
 tensor([[ 5,  5],
         [11, 11]]),
 tensor([[ 4,  4],
         [12, 12]]),
 tensor([[7, 7],
         [9, 9]]),
 tensor([[8, 8],
         [8, 8]]),
 tensor([[8, 8],
         [8, 8]]),
 tensor([[ 6,  6],
         [10, 10]]),
 tensor([[9, 9],
         [7, 7]])]
```

In [ ]:
```
import matplotlib.pyplot as plt

x = np.arange(0, 100, 1)

# Plot di più liste sullo stesso grafico
plt.plot(x, losses_RNN, label='RNN')
plt.plot(x, losses_GRU, label='GRU')
plt.plot(x, losses_LSTM, label='LSTM')

# Aggiungo etichette e titolo
plt.xlabel('Eposchs')
plt.ylabel('Losses')
plt.title('Confronto tra le Losses delle diverse NN')
plt.legend()   # Mostra la legenda
```

Out[ ]:
```
<matplotlib.legend.Legend at 0x7e76b73d1d90>
```



In [ ]:

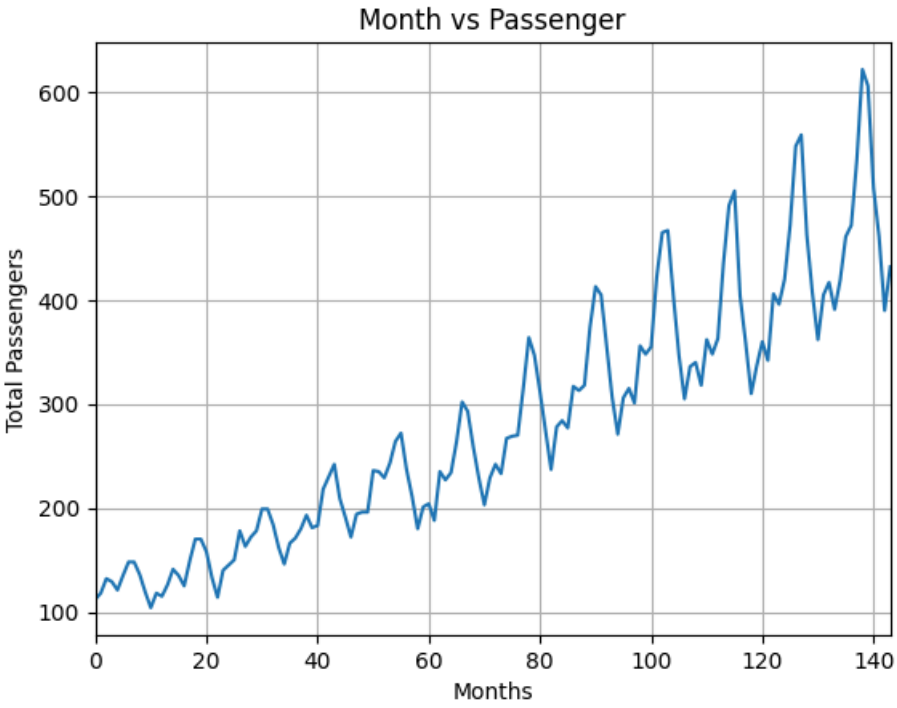
```
import seaborn as sns
import matplotlib.pyplot as plt
import torch
flight_data = sns.load_dataset("flights")
flight_data.head(50)
```

Out[ ]:

	year	month	passengers
0	1949	Jan	112
1	1949	Feb	118
2	1949	Mar	132
3	1949	Apr	129
4	1949	May	121
5	1949	Jun	135
6	1949	Jul	148
7	1949	Aug	148
8	1949	Sep	136
9	1949	Oct	119
10	1949	Nov	104
11	1949	Dec	118
12	1950	Jan	115
13	1950	Feb	126
14	1950	Mar	141
15	1950	Apr	135
16	1950	May	125
17	1950	Jun	149
18	1950	Jul	170
19	1950	Aug	170
20	1950	Sep	158
21	1950	Oct	133
22	1950	Nov	114
23	1950	Dec	140
24	1951	Jan	145
25	1951	Feb	150
26	1951	Mar	178
27	1951	Apr	163
28	1951	May	172
29	1951	Jun	178
30	1951	Jul	199
31	1951	Aug	199
32	1951	Sep	184
33	1951	Oct	162
34	1951	Nov	146
35	1951	Dec	166
36	1952	Jan	171
37	1952	Feb	180
38	1952	Mar	193
39	1952	Apr	181
40	1952	May	183
41	1952	Jun	218
42	1952	Jul	230
43	1952	Aug	242
44	1952	Sep	209
45	1952	Oct	191
46	1952	Nov	172
47	1952	Dec	194
48	1953	Jan	196
49	1953	Feb	196

In [ ]:
```
plt.title('Month vs Passenger')
plt.ylabel('Total Passengers')
plt.xlabel('Months')
plt.grid(True)
plt.autoscale(axis='x',tight=True)
plt.plot(flight_data['passengers'])
```

Out[ ]:
```
[<matplotlib.lines.Line2D at 0x7e76b73d06b0>]
```



In [ ]:
```
all_data = flight_data['passengers'].values.astype(float)

test_data_size = 12

train_data = all_data[:-test_data_size]
test_data = all_data[-test_data_size:]
```

In [ ]:
```
from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler(feature_range=(-1, 1))
train_data_normalized = scaler.fit_transform(train_data .reshape(-1, 1))
```

In [ ]:
```
train_data_normalized = torch.FloatTensor(train_data_normalized).view(-1)
```

In [ ]:
```
train_window = 12
```

In [ ]:
```
'''
The function will accept the raw input data and will return a list of tuples.
In each tuple, the first element will contain list of 12 items corresponding
to the number of passengers traveling in 12 months, the second tuple element
will contain one item i.e. the number of passengers in the 12+1st month
'''

def create_inout_sequences(input_data, tw):
    inout_seq = []
    L = len(input_data)
    for i in range(L-tw):
        train_seq = input_data[i:i+tw]
        train_label = input_data[i+tw:i+tw+1]
        inout_seq.append((train_seq ,train_label))
    return inout_seq
```

In [ ]:
```
train_inout_seq = create_inout_sequences(train_data_normalized, train_window)
```

In [ ]:
```
class ts_LSTM(nn.Module):
    def __init__(self, n_features=1, hidden_dim=100, n_outputs=1):
        super(ts_LSTM, self).__init__()
        self.hidden_dim = hidden_dim
        self.n_features = n_features
        self.n_outputs = n_outputs
        self.hidden = None
        self.cell = None
        # Simple LSTM
        self.basic_rnn = nn.LSTM(self.n_features, self.hidden_dim,batch_first=True)
        # Classifier to produce as many logits as outputs
        self.classifier = nn.Linear(self.hidden_dim, self.n_outputs)

    def forward(self, X):
        # X is batch first (N, L, F)
        # output is (N, L, H)
        # final hidden state is (1, N, H)
        # final cell state is (1, N, H)
        batch_first_output, (self.hidden, self.cell) = self.basic_rnn(X)

        # only last item in sequence (N, 1, H)
        last_output = batch_first_output[:, -1]
        # classifier will output (N, 1, n_outputs)
        out = self.classifier(last_output)

        # final output is (N, n_outputs)
        return out.view(-1, self.n_outputs)
```

In [ ]:
```
torch.manual_seed(21)
model = ts_LSTM()
loss_function = nn.MSELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
```

In [ ]:
```
epochs = 50

for i in range(epochs):
    for seq, labels in train_inout_seq:
        optimizer.zero_grad()
        # model.hidden_cell = (torch.zeros(1, 1, model.hidden_layer_size),
        #                      torch.zeros(1, 1, model.hidden_layer_size))
        # print(seq.shape)
        seq = seq.reshape(1,-1,1)
        labels = labels.reshape(1,1)
        y_pred = model(seq)
        labels = labels.reshape(1,1)
        single_loss = loss_function(y_pred, labels)
        single_loss.backward()
        optimizer.step()

    if i%25 == 1:
        print(f'epoch: {i:3} loss: {single_loss.item():10.8f}')

print(f'epoch: {i:3} loss: {single_loss.item():10.10f}')
```

```
epoch:   1 loss: 0.02986891
epoch:  26 loss: 0.00621752
epoch:  49 loss: 0.0003344537
```

In [ ]:
```
fut_pred = 12

test_inputs = train_data_normalized[-train_window:].tolist()

model.eval()

for i in range(fut_pred):
    seq = torch.FloatTensor(test_inputs[-train_window:])
    with torch.no_grad():
        seq = seq.reshape(1,-1,1)
        test_inputs.append(model(seq).item())
test_inputs[fut_pred:]
```

Out[ ]:
```
[0.43335869908332825,
 0.5117570757865906,
 0.637189507484436,
 0.7038470506668091,
 0.7698838710784912,
 0.8075113892555237,
 0.8465009927749634,
 0.8652442693710327,
 0.8778355121612549,
 0.8871387243270874,
 0.9525226354598999,
 1.1090991497039795]
```

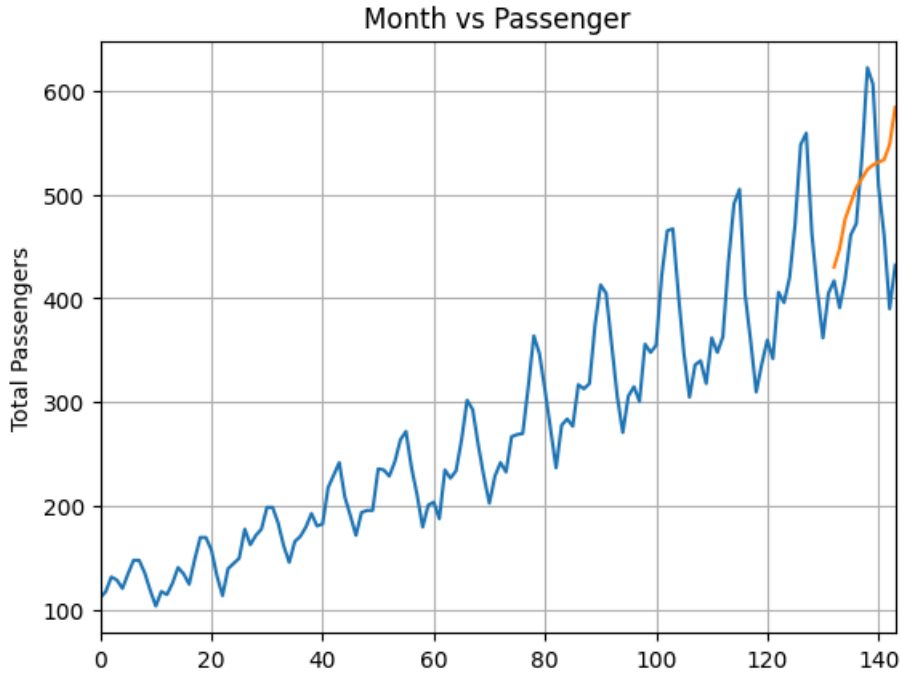
In [ ]:
```
actual_predictions_LSTM = scaler.inverse_transform(np.array(test_inputs[train_window:] ).reshape(-1, 1))
print(actual_predictions_LSTM)
```

```
[[430.08910404]
 [447.92473474]
 [476.46061295]
 [491.62520403]
 [506.64858067]
 [515.20884106]
 [524.07897586]
 [528.34307128]
 [531.20757902]
 [533.32405978]
 [548.19889957]
 [583.82005656]]
```

In [ ]:
```
x = np.arange(132, 144, 1)
print(x)
```

```
[132 133 134 135 136 137 138 139 140 141 142 143]
```

In [ ]:
```
plt.title('Month vs Passenger')
plt.ylabel('Total Passengers')
plt.grid(True)
plt.autoscale(axis='x', tight=True)
plt.plot(flight_data['passengers'])
plt.plot(x,actual_predictions_LSTM)
plt.show()
```



In [ ]:
```
'''
The only change in the code is RNN in GRU
'''

class mb_GRU(nn.Module):
    def __init__(self, n_features = 1, hidden_dim = 100, n_outputs = 1):
        super(mb_GRU, self).__init__()
        self.hidden_dim = hidden_dim
        self.n_features = n_features
        self.n_outputs = n_outputs
        self.hidden = None
        # Simple GRU
        self.basic_rnn = nn.GRU(self.n_features, self.hidden_dim, batch_first=True) # only difference from RNN
to GRU
        # Classifier to produce as many logits as outputs
        self.classifier = nn.Linear(self.hidden_dim, self.n_outputs)

    def forward(self, X):
        # X is batch first (N, L, F)
        # output is (N, L, H)
        # final hidden state is (1, N, H)
        batch_first_output, self.hidden = self.basic_rnn(X)

        # only last item in sequence (N, 1, H)
        last_output = batch_first_output[:, -1]
        # classifier will output (N, 1, n_outputs)
        out = self.classifier(last_output)

        # final output is (N, n_outputs)
        return out.view(-1, self.n_outputs)
```

In [ ]:
```
torch.manual_seed(21)
model = mb_GRU()
loss_function = nn.MSELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
```

In [ ]:
```
epochs = 50

for i in range(epochs):
    for seq, labels in train_inout_seq:
        optimizer.zero_grad()
        # model.hidden_cell = (torch.zeros(1, 1, model.hidden_layer_size),
        #                      torch.zeros(1, 1, model.hidden_layer_size))
        # print(seq.shape)
        seq = seq.reshape(1,-1,1)
        labels = labels.reshape(1,1)
        y_pred = model(seq)
        labels = labels.reshape(1,1)
        single_loss = loss_function(y_pred, labels)
        single_loss.backward()
        optimizer.step()

    if i%25 == 1:
        print(f'epoch: {i:3} loss: {single_loss.item():10.8f}')

print(f'epoch: {i:3} loss: {single_loss.item():10.10f}')
```

```
epoch:   1 loss: 0.00164406
epoch:  26 loss: 0.00150507
epoch:  49 loss: 0.0001447605
```

In [ ]:
```
fut_pred = 12

test_inputs = train_data_normalized[-train_window:].tolist()

model.eval()

for i in range(fut_pred):
    seq = torch.FloatTensor(test_inputs[-train_window:])
    with torch.no_grad():
        seq = seq.reshape(1,-1,1)
        test_inputs.append(model(seq).item())
test_inputs[fut_pred:]
```

Out[ ]:
```
[0.4311012029647827,
 0.519609808921814,
 0.6530308127403259,
 0.8561264276504517,
 1.2833694219589233,
 2.016995429992676,
 2.695582866668701,
 2.7145426273345947,
 1.9474432468414307,
 1.6038364171981812,
 1.6192395687103271,
 1.8188109397888184]
```

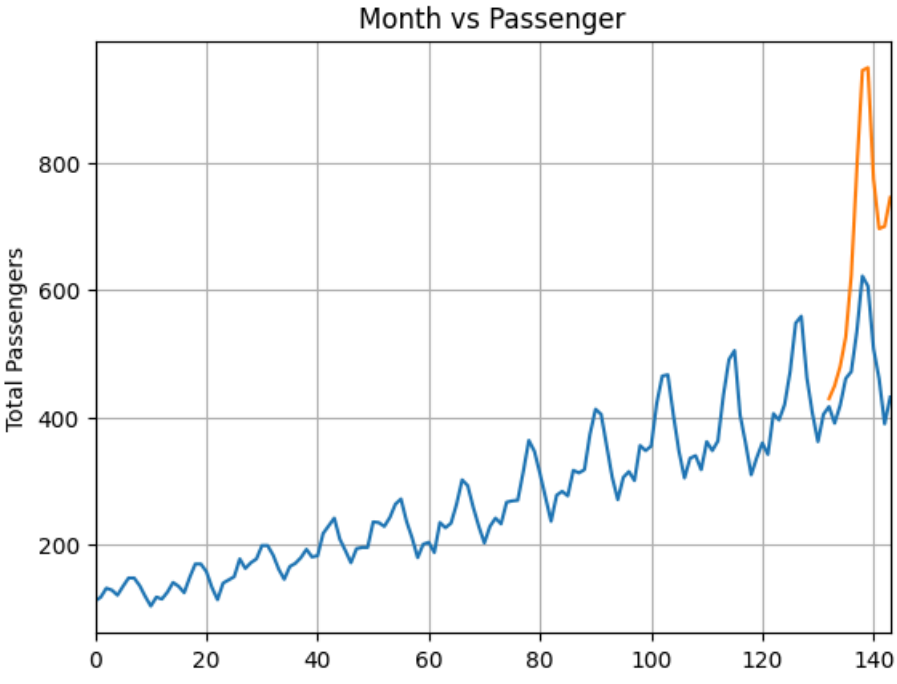
In [ ]:
```
actual_predictions_GRU = scaler.inverse_transform(np.array(test_inputs[train_window:] ).reshape(-1, 1))
print(actual_predictions_GRU)
```

```
[[429.57552367]
 [449.71123153]
 [480.0645099 ]
 [526.26876229]
 [623.4665435 ]
 [790.36646032]
 [944.74510217]
 [949.05844772]
 [774.54333866]
 [696.37278491]
 [699.87700188]
 [745.2794888 ]]
```

In [ ]:
```
x = np.arange(132, 144, 1)
print(x)
```

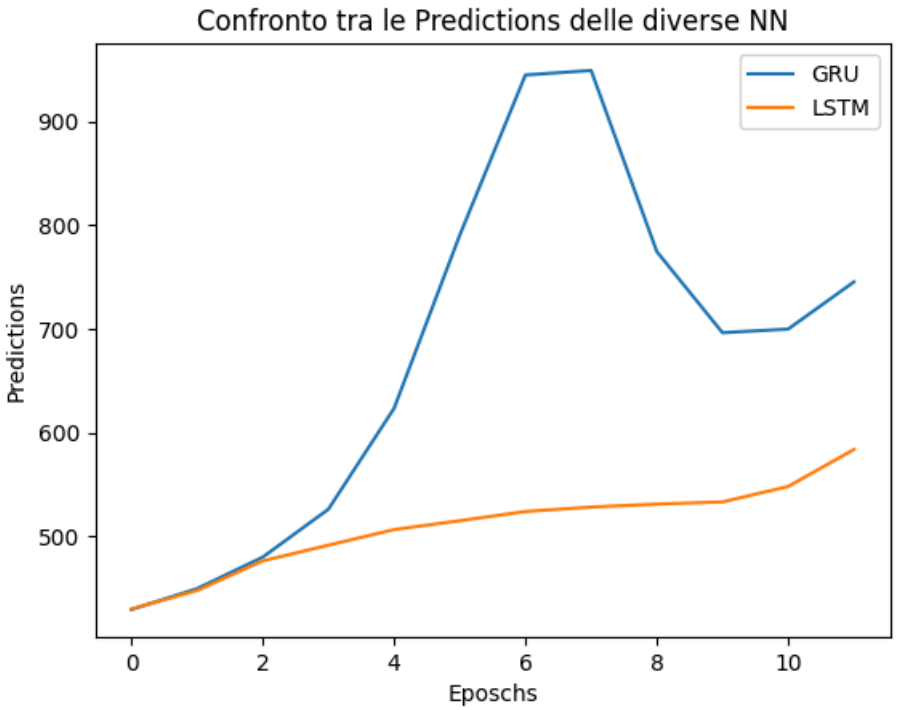
```
[132 133 134 135 136 137 138 139 140 141 142 143]
```

In [ ]:
```
plt.title('Month vs Passenger')
plt.ylabel('Total Passengers')
plt.grid(True)
plt.autoscale(axis='x', tight=True)
plt.plot(flight_data['passengers'])
plt.plot(x,actual_predictions_GRU)
plt.show()
```



In [ ]:
```
import matplotlib.pyplot as plt

x = np.arange(0, 12, 1)

# Plot di più liste sullo stesso grafico
plt.plot(x, actual_predictions_GRU, label='GRU')
plt.plot(x, actual_predictions_LSTM, label='LSTM')

# Aggiungo etichette e titolo
plt.xlabel('Eposchs')
plt.ylabel('Predictions')
plt.title('Confronto tra le Predictions delle diverse NN')
plt.legend()   # Mostra la legenda
```

Out[ ]:
```
<matplotlib.legend.Legend at 0x7e76b5386180>
```



In [ ]:
```
class ts_GRU(nn.Module):
    def __init__(self, n_features=1, hidden_dim=100, n_outputs=1):
        super(ts_GRU, self).__init__()
        self.hidden_dim = hidden_dim
        self.n_features = n_features
        self.n_outputs = n_outputs
        self.hidden = None
        # Simple GRU
        self.basic_rnn = nn.GRU(self.n_features, self.hidden_dim,batch_first=True)
        # Classifier to produce as many logits as outputs
        self.classifier = nn.Linear(self.hidden_dim, self.n_outputs)

    def forward(self, X):
        # X is batch first (N, L, F)
        # output is (N, L, H)
        # final hidden state is (1, N, H)
        # final cell state is (1, N, H)
        batch_first_output, self.hidden = self.basic_rnn(X)

        # only last item in sequence (N, 1, H)
        last_output = batch_first_output[:, -1]
        # classifier will output (N, 1, n_outputs)
        out = self.classifier(last_output)

        # final output is (N, n_outputs)
        return out.view(-1, self.n_outputs)
```

In [ ]:
```
torch.manual_seed(21)
model = ts_GRU()
loss_function = nn.MSELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
```

In [ ]:

```
epochs = 300

for i in range(epochs):
    for seq, labels in train_inout_seq:
        optimizer.zero_grad()
        # model.hidden_cell = (torch.zeros(1, 1, model.hidden_layer_size),
        #                      torch.zeros(1, 1, model.hidden_layer_size))
        # print(seq.shape)
        seq = seq.reshape(1,-1,1)
        labels = labels.reshape(1,1)
        y_pred = model(seq)
        labels = labels.reshape(1,1)
        single_loss = loss_function(y_pred, labels)
        single_loss.backward()
        optimizer.step()

    if i%25 == 1:
        print(f'epoch: {i:3} loss: {single_loss.item():10.8f}')

print(f'epoch: {i:3} loss: {single_loss.item():10.10f}')
```

```
epoch:   1 loss: 0.00164406
epoch:  26 loss: 0.00150507
epoch:  51 loss: 0.00000282
epoch:  76 loss: 0.00324978
```

```
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
/tmp/ipython-input-2900987898.py in <cell line: 0>()
     12         labels = labels.reshape(1,1)
     13         single_loss = loss_function(y_pred, labels)
---> 14         single_loss.backward()
     15         optimizer.step()
     16

/usr/local/lib/python3.12/dist-packages/torch/_tensor.py in backward(self, gradient, retain_graph, create_graph, inputs)
    645                 inputs=inputs,
    646             )
--> 647         torch.autograd.backward(
    648             self, gradient, retain_graph, create_graph, inputs=inputs
    649         )

/usr/local/lib/python3.12/dist-packages/torch/autograd/__init__.py in backward(tensors, grad_tensors, retain_graph, create_graph, grad_variables, inputs)
    352     # some Python versions print out the first line of a multi-line function
    353     # calls in the traceback and some print out the last line
--> 354     _engine_run_backward(
    355         tensors,
    356         grad_tensors_,

/usr/local/lib/python3.12/dist-packages/torch/autograd/graph.py in _engine_run_backward(t_outputs, *args, **kwargs)
    827         unregister_hooks = _register_logging_hooks_on_whole_graph(t_outputs)
    828     try:
--> 829         return Variable._execution_engine.run_backward(  # Calls into the C++ engine to run the backward pass
    830             t_outputs, *args, **kwargs
    831         )  # Calls into the C++ engine to run the backward pass

KeyboardInterrupt:
```

In [ ]:
```
fut_pred = 12

test_inputs = train_data_normalized[-train_window:].tolist()

model.eval()

for i in range(fut_pred):
    seq = torch.FloatTensor(test_inputs[-train_window:])
    with torch.no_grad():
        seq = seq.reshape(1,-1,1)
        test_inputs.append(model(seq).item())
test_inputs[fut_pred:]
```

Out[ ]:
```
[0.44217008352279663,
 0.5318998694419861,
 0.7742842435836792,
 0.9958425760269165,
 1.2512704133987427,
 1.7267005443572998,
 2.2256407737731934,
 2.341217517852783,
 2.8071672916412354,
 2.3699090480804443,
 2.4356255531311035,
 0.9082947373390198]
```

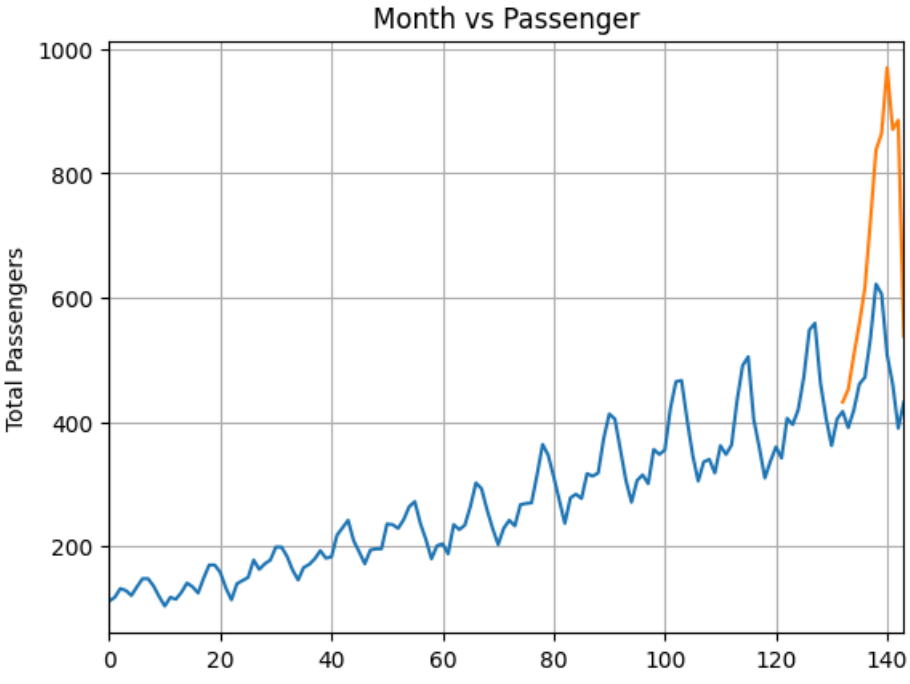
In [ ]:
```
actual_predictions = scaler.inverse_transform(np.array(test_inputs[train_window:] ).reshape(-1, 1))
print(actual_predictions)
```

```
[[432.093694  ]
 [452.5072203 ]
 [507.64966542]
 [558.05418605]
 [616.16401905]
 [724.32437384]
 [837.83327603]
 [864.12698531]
 [970.13055885]
 [870.65430844]
 [885.60481334]
 [538.13705274]]
```

In [ ]:
```
x = np.arange(132, 144, 1)
print(x)
```

```
[132 133 134 135 136 137 138 139 140 141 142 143]
```

In [ ]:
```
plt.title('Month vs Passenger')
plt.ylabel('Total Passengers')
plt.grid(True)
plt.autoscale(axis='x', tight=True)
plt.plot(flight_data['passengers'])
plt.plot(x,actual_predictions)
plt.show()
```



In [ ]:

In [ ]:

In [ ]:
```
'''
input_size: Corresponds to the number of features in the input.
Though our sequence length is 12, for each month we have only
1 value i.e. total number of passengers, therefore the input size will be 1.
'''

'''
hidden_layer_size: Specifies the number of hidden layers along
with the number of neurons in each layer.
We will have one layer of 100 neurons.
'''

'''
output_size: The number of items in the output,
since we want to predict the number of passengers
for 1 month in the future, the output size will be 1.
'''

'''
Inside the forward method, the input_seq is passed as a parameter,
which is first passed through the lstm layer. The output of the lstm layer is
the hidden and cell states at current time step, along with the output.
The output from the lstm layer is passed to the linear layer.
The predicted number of passengers is stored in the last item of the
predictions list, which is returned to the calling function.
'''


class LSTM(nn.Module):
    def __init__(self, input_size=1, hidden_layer_size=250, output_size=1):
        super().__init__()
        self.hidden_layer_size = hidden_layer_size

        self.lstm = nn.LSTM(input_size, hidden_layer_size)

        self.linear = nn.Linear(hidden_layer_size, output_size)

        self.hidden_cell = (torch.zeros(1,1,self.hidden_layer_size),
                            torch.zeros(1,1,self.hidden_layer_size))

    def forward(self, input_seq,to_print=True):
        if to_print:
          print(input_seq.shape)

        lstm_out, self.hidden_cell = self.lstm(input_seq.view(len(input_seq) ,1, -1), self.hidden_cell)
        predictions = self.linear(lstm_out.view(len(input_seq), -1))
        return predictions[-1]
```