PyTorch Tutorial

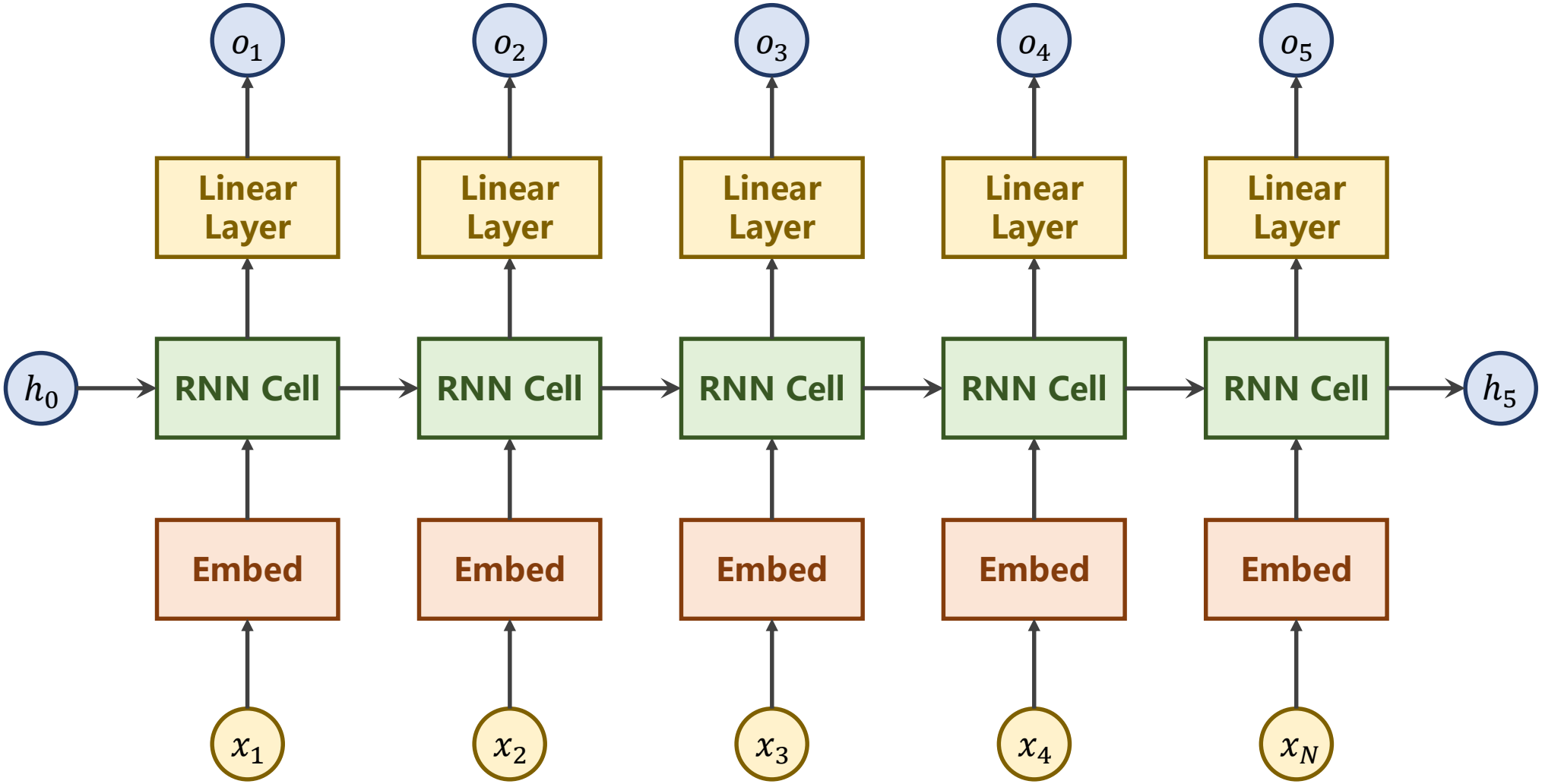
13. RNN Classifier

RNN Classifier – Name Classification

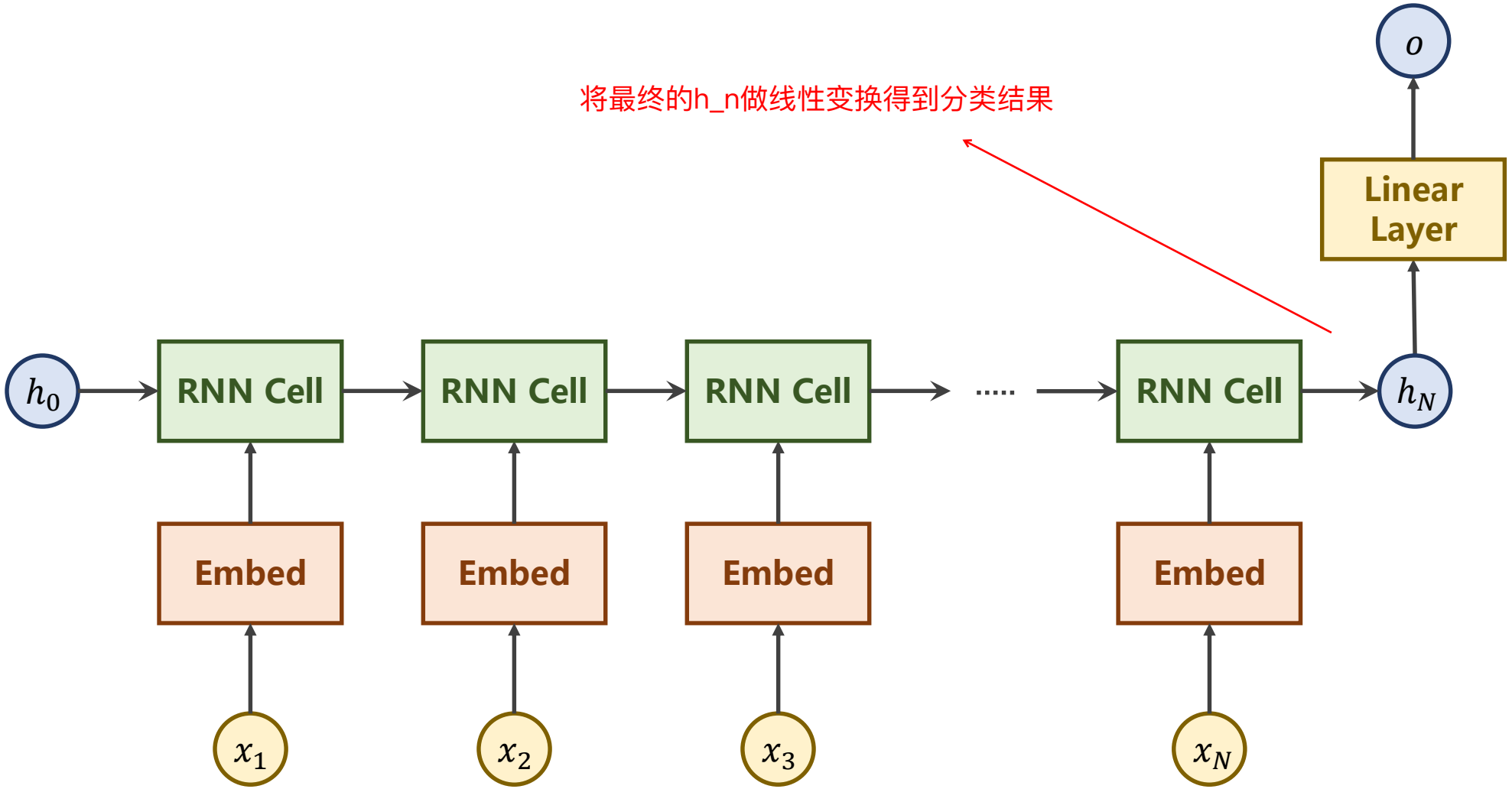
Name	Country
Maclean	English
Vajnichy	Russian
Nasikovsky	Russian
Usami	Japanese
Fionin	Russian
Sharkey	English
Balagul	Russian
Pakhrin	Russian
Tansho	Japanese

We shall train on a few thousand surnames from 18 languages of origin, and predict which language a name is from based on the spelling.

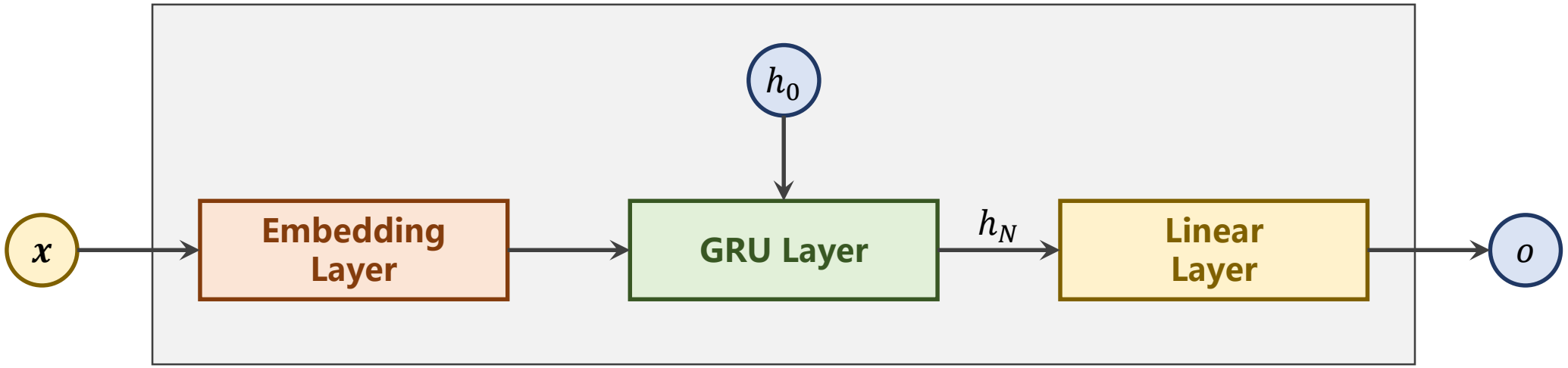
Revision


o_1
o_2
o_3
o_4
o_5
Linear Layer
Linear Layer
Linear Layer
Linear Layer
Linear Layer
h_0
RNN Cell
RNN Cell
RNN Cell
RNN Cell
RNN Cell
h_5
Embed
Embed
Embed
Embed
Embed
x_1
x_2
x_3
x_4
x_N

Our Model

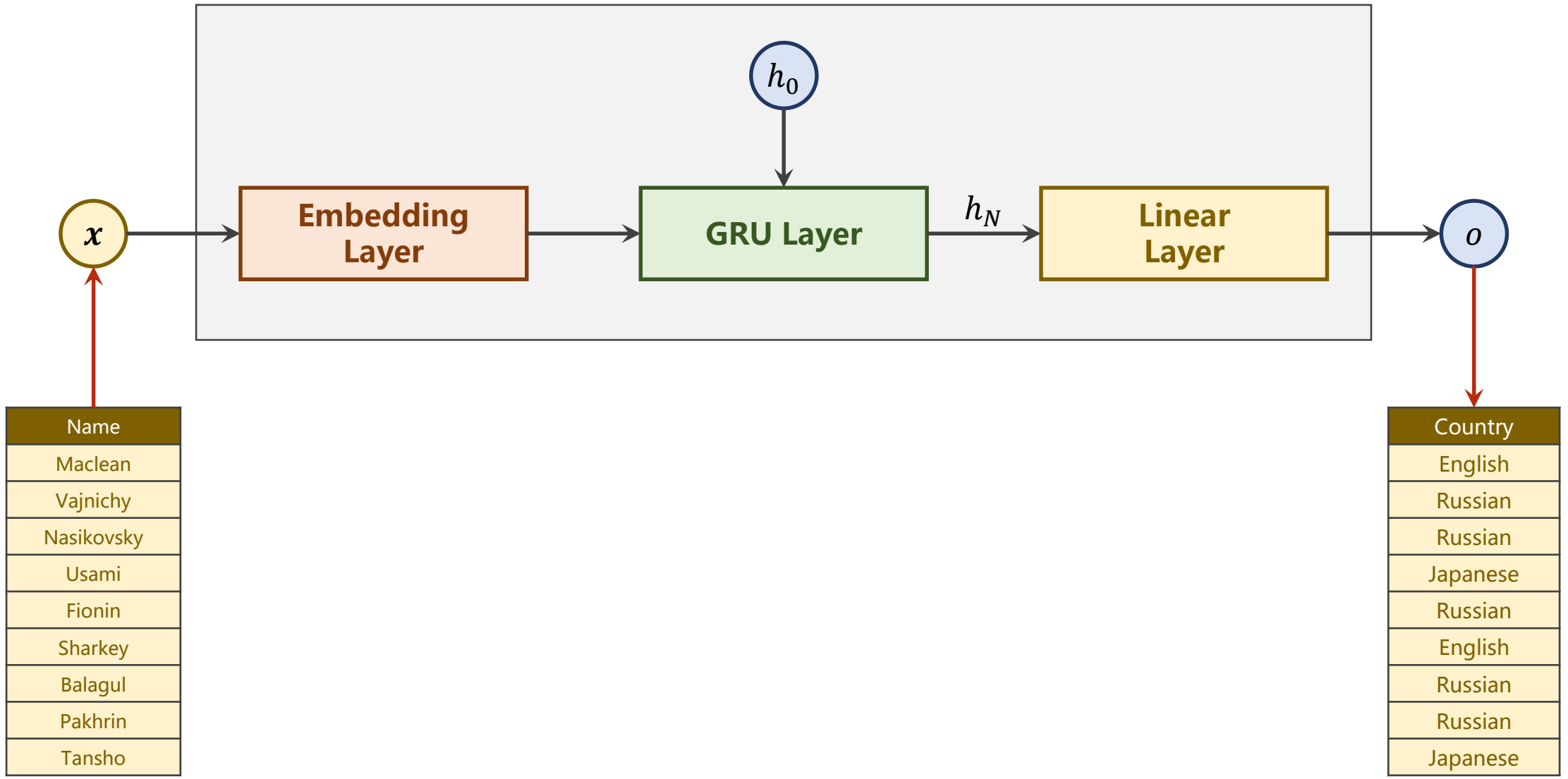

将最终的h_n做线性变换得到分类结果
o
Linear Layer
h_0
RNN Cell
RNN Cell
RNN Cell
.....
RNN Cell
h_N
Embed
Embed
Embed
Embed
x_1
x_2
x_3
x_N

Our Model



Name	Country
Maclean	English
Vajnichy	Russian
Nasikovsky	Russian
Usami	Japanese
Fionin	Russian
Sharkey	English
Balagul	Russian
Pakhrin	Russian
Tansho	Japanese

Our Model


h_0
x
Embedding Layer
GRU Layer
h_N
Linear Layer
o
Name
Maclean
Vajnichy
Nasikovsky
Usami
Fionin
Sharkey
Balagul
Pakhrin
Tansho
Country
English
Russian
Russian
Japanese
Russian
English
Russian
Russian
Japanese

Implementation – Main Cycle

```
if __name__ == '__main__':
    classifier = RNNClassifier(N_CHARS, HIDDEN_SIZE, N_COUNTRY, N_LAYER)
    if USE_GPU:
        device = torch.device("cuda:0")
        classifier.to(device)

    criterion = torch.nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(classifier.parameters(), lr=0.001)

    start = time.time()
    print("Training for %d epochs..." % N_EPOCHS)
    acc_list = []
    for epoch in range(1, N_EPOCHS + 1):
        # Train cycle
        trainModel()
        acc = testModel()
        acc_list.append(acc)
```

Instantiate the classifier model.

Implementation – Main Cycle

```
if __name__ == '__main__':
    classifier = RNNClassifier(N_CHARS, HIDDEN_SIZE, N_COUNTRY, N_LAYER)
    if USE_GPU:
        device = torch.device("cuda:0")
        classifier.to(device)

    criterion = torch.nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(classifier.parameters(), lr=0.001)

    start = time.time()
    print("Training for %d epochs..." % N_EPOCHS)
    acc_list = []
    for epoch in range(1, N_EPOCHS + 1):
        # Train cycle
        trainModel()
        acc = testModel()
        acc_list.append(acc)
```

Whether use GPU for training model.

Implementation – Main Cycle

```
if __name__ == '__main__':
    classifier = RNNClassifier(N_CHARS, HIDDEN_SIZE, N_COUNTRY, N_LAYER)
    if USE_GPU:
        device = torch.device("cuda:0")
        classifier.to(device)

    criterion = torch.nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(classifier.parameters(), lr=0.001)

    start = time.time()
    print("Training for %d epochs..." % N_EPOCHS)
    acc_list = []
    for epoch in range(1, N_EPOCHS + 1):
        # Train cycle
        trainModel()
        acc = testModel()
        acc_list.append(acc)
```

Using cross entropy loss as loss function.

Using Adam optimizer.

Implementation – Main Cycle

```
if __name__ == '__main__':
    classifier = RNNClassifier(N_CHARS, HIDDEN_SIZE, N_COUNTRY, N_LAYER)
    if USE_GPU:
        device = torch.device("cuda:0")
        classifier.to(device)

    criterion = torch.nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(classifier.parameters(), lr=0.001)

    start = time.time()
    print("Training for %d epochs..." % N_EPOCHS)
    acc_list = []
    for epoch in range(1, N_EPOCHS + 1):
        # Train cycle
        trainModel()
        acc = testModel()
        acc_list.append(acc)
```

For printing elapsed time.

```
def time_since(since):
    s = time.time() - since
    m = math.floor(s / 60)
    s -= m * 60
    return '%dm %ds' % (m, s)
```

Implementation – Main Cycle

```
if __name__ == '__main__':
    classifier = RNNClassifier(N_CHARS, HIDDEN_SIZE, N_COUNTRY, N_LAYER)
    if USE_GPU:
        device = torch.device("cuda:0")
        classifier.to(device)

    criterion = torch.nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(classifier.parameters(), lr=0.001)

    start = time.time()
    print("Training for %d epochs..." % N_EPOCHS)
    acc_list = []
    for epoch in range(1, N_EPOCHS + 1):
        # Train cycle
        trainModel()
        acc = testModel()
        acc_list.append(acc)
```

In every epoch, training and testing the model once.

Implementation – Main Cycle

```
if __name__ == '__main__':
    classifier = RNNClassifier(N_CHARS, HIDDEN_SIZE, N_COUNTRY, N_LAYER)
    if USE_GPU:
        device = torch.device("cuda:0")
        classifier.to(device)

    criterion = torch.nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(classifier.parameters(), lr=0.001)

    start = time.time()
    print("Training for %d epochs..." % N_EPOCHS)
    acc_list = []
    for epoch in range(1, N_EPOCHS + 1):
        # Train cycle
        trainModel()
        acc = testModel()
        acc_list.append(acc)
```

Recording the accuracy of testing.

```
import matplotlib.pyplot as plt
import numpy as np

epoch = np.arange(1, len(acc_list) + 1, 1)
acc_list = np.array(acc_list)
plt.plot(epoch, acc_list)
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.grid()
plt.show()
```

Implementation – Preparing Data

Name	Characters	ASCII
Maclean	['M', 'a', 'c', 'l', 'e', 'a', 'n']	[77 97 99 108 101 97 110]
Vajnichy	['V', 'a', 'j', 'n', 'i', 'c', 'h', 'y']	[86 97 106 110 105 99 104 121]
Nasikovsky	['N', 'a', 's', 'i', 'k', 'o', 'v', 's', 'k', 'y']	[78 97 115 105 107 111 118 115 107 121]
Usami	['U', 's', 'a', 'm', 'i']	[85 115 97 109 105]
Fionin	['F', 'i', 'o', 'n', 'i', 'n']	[70 105 111 110 105 110]
Sharkey	['S', 'h', 'a', 'r', 'k', 'e', 'y']	[83 104 97 114 107 101 121]
Balagul	['B', 'a', 'l', 'a', 'g', 'u', 'l']	[66 97 108 97 103 117 108]
Pakhrin	['P', 'a', 'k', 'h', 'r', 'i', 'n']	[80 97 107 104 114 105 110]
Tansho	['T', 'a', 'n', 's', 'h', 'o']	[84 97 110 115 104 111]

Implementation – Preparing Data

ASCII
[77 97 99 108 101 97 110]
[86 97 106 110 105 99 104 121]
[78 97 115 105 107 111 118 115 107 121]
[85 115 97 109 105]
[70 105 111 110 105 110]
[83 104 97 114 107 101 121]
[66 97 108 97 103 117 108]
[80 97 107 104 114 105 110]
[84 97 110 115 104 111]

After padding
[77 97 99 108 101 97 110   0   0   0]
[86 97 106 110 105 99 104 121   0   0]
[78 97 115 105 107 111 118 115 107 121]
[85 115 97 109 105   0   0   0   0   0]
[70 105 111 110 105 110   0   0   0   0]
[83 104 97 114 107 101 121   0   0   0]
[66 97 108 97 103 117 108   0   0   0]
[80 97 107 104 114 105 110   0   0   0]
[84 97 110 115 104 111   0   0   0   0]

Implementation – Preparing Data

Country	Index	Country	Index
Arabic	0	Chinese	1
Czech	2	Dutch	3
English	4	French	5
German	6	Greek	7
Irish	8	Italian	9
Japanese	10	Korean	11
Polish	12	Portuguese	13
Russian	14	Scottish	15
Spanish	16	Vietnamese	17

Implementation – Preparing Data

Reading data from .gz file with package ***gzip*** and ***csv***.

```
import gzip
import csv
```

```
class NameDataset(Dataset):
    def __init__(self, is_train_set=True):
        filename = 'data/names_train.csv.gz' if is_train_set else 'data/names_test.csv.gz'
        with gzip.open(filename, 'rt') as f:
            reader = csv.reader(f)
            rows = list(reader)
        self.names = [row[0] for row in rows]
        self.len = len(self.names)
        self.countries = [row[1] for row in rows]
        self.country_list = list(sorted(set(self.countries)))
        self.country_dict = self.getCountryDict()
        self.country_num = len(self.country_list)

    def __getitem__(self, index):
        return self.names[index], self.country_dict[self.countries[index]]

    def __len__(self):
        return self.len
```

Implementation – Preparing Data

Save names and countries in ***list***.

```
class NameDataset(Dataset):
    def __init__(self, is_train_set=True):
        filename = 'data/names_train.csv.gz' if is_train_set else 'data/names_test.csv.gz'
        with gzip.open(filename, 'rt') as f:
            reader = csv.reader(f)
            rows = list(reader)
        self.names = [row[0] for row in rows]
        self.len = len(self.names)
        self.countries = [row[1] for row in rows]
        self.country_list = list(sorted(set(self.countries)))
        self.country_dict = self.getCountryDict()
        self.country_num = len(self.country_list)

    def __getitem__(self, index):
        return self.names[index], self.country_dict[self.countries[index]]

    def __len__(self):
        return self.len
```

Name
Maclean
Vajnichy
Nasikovsky
Usami
Fionin
Sharkey
Balagul
Pakhrin
Tansho

Country
English
Russian
Russian
Japanese
Russian
English
Russian
Russian
Japanese

Implementation – Preparing Data

```
class NameDataset(Dataset):
    def __init__(self, is_train_set=True):
        filename = 'data/names_train.csv.gz' if is_train_set else 'data/names_test.csv.gz'
        with gzip.open(filename, 'rt') as f:
            reader = csv.reader(f)
            rows = list(reader)
        self.names = [row[0] for row in rows]
        self.len = len(self.names)
        self.countries = [row[1] for row in rows]
        self.country_list = list(sorted(set(self.countries)))
        self.country_dict = self.getCountryDict()
        self.country_num = len(self.country_list)

    def __getitem__(self, index):
        return self.names[index], self.country_dict[self.countries[index]]

    def __len__(self):
        return self.len
```

Country	Index	Country	Index
Arabic	0	Chinese	1
Czech	2	Dutch	3
English	4	French	5
German	6	Greek	7
Irish	8	Italian	9
Japanese	10	Korean	11
Polish	12	Portuguese	13
Russian	14	Scottish	15
Spanish	16	Vietnamese	17

Save countries and its index in ***list*** and ***dictionary***.

Implementation – Preparing Data

```
class NameDataset(Dataset):
    def __init__(self, is_tra
        filename = 'data/names
        with gzip.open(filenar
            reader = csv.reade
            rows = list(reader
        self.names = [row[0] f
        self.len = len(self.na
        self.countries = [row[
        self.country_list = list(sorted(set(self.countries)))
        self.country_dict = self.getCountryDict()
        self.country_num = len(self.country_list)

    def __getitem__(self, index):
        return self.names[index], self.country_dict[self.countries[index]]

    def __len__(self):
        return self.len
```

train . gz'

Name	Country
Maclean	English
Vajnichy	Russian
Nasikovsky	Russian
Usami	Japanese
Fionin	Russian
Sharkey	English
Balagul	Russian
Pakhrin	Russian
Tansho	Japanese

Country	Index	Country	Index
Arabic	0	Chinese	1
Czech	2	Dutch	3
English	4	French	5
German	6	Greek	7
Irish	8	Italian	9
Japanese	10	Korean	11
Polish	12	Portuguese	13
Russian	14	Scottish	15
Spanish	16	Vietnamese	17

Save countries and its index in ***list*** and ***dictionary***.

Implementation – Preparing Data

```
class NameDataset(Dataset):
    def __init__(self, is_tra
        filename = 'data/names
        with gzip.open(filenar
            reader = csv.reade
            rows = list(reader
        self.names = [row[0] f
        self.len = len(self.na
        self.countries = [row[
        self.country_list = list(sorted(set(self.countries)))
        self.country_dict = self.getCountryDict()
        self.country_num = len(self.country_list)

    def __getitem__(self, index):
        return self.names[index], self.country_dict[self.countries[index]]

    def __len__(self):
        return self.len
```

train . gz'

Name	Country
Maclean	English
Vajnichy	Russian
Nasikovsky	**Russian**
Usami	Japanese
Fionin	Russian
Sharkey	English
Balagul	Russian
Pakhrin	Russian
Tansho	Japanese

Country	Index	Country	Index
Arabic	0	Chinese	1
Czech	2	Dutch	3
English	4	French	5
German	6	Greek	7
Irish	8	Italian	9
Japanese	10	Korean	11
Polish	12	Portuguese	13
Russian	**14**	Scottish	15
Spanish	16	Vietnamese	17

Save countries and its index in ***list*** and ***dictionary***.

Implementation – Preparing Data

```
class NameDataset(Dataset):
    def __init__(self, is_train_set=True):
        filename = 'data/names_train.csv.gz' if is_train_set else 'data/names_test.csv.gz'
        with gzip.open(filename, 'rt') as f:
            reader = csv.reader(f)
            rows = list(reader)
        self.names = [row[0] for row in rows]
        self.len = len(self.names)
        self.countries = [row[1] for row in rows]
        self.country_list = list(sorted(set(self.countries)))
        self.country_dict = self.getCountryDict()
        self.country_num = len(self.country_list)

    def __getitem__(self, index):
        return self.names[index], self.country_dict[self.countries[index]]

    def __len__(self):
        return self.len
```

Return length of dataset.

Implementation – Preparing Data

Country	Index	Country	Index
Arabic	0	Chinese	1
Czech	2	Dutch	3
English	4	French	5
German	6	Greek	7
Irish	8	Italian	9
Japanese	10	Korean	11
Polish			
Russian			
Spanish			

Convert ***list*** into ***dictionary***.

```
class NameDataset(Dataset):

    ......

    def getCountryDict(self):
        country_dict = dict()
        for idx, country_name in enumerate(self.country_list, 0):
            country_dict[country_name] = idx
        return country_dict

    def idx2country(self, index):
        return self.country_list[index]

    def getCountriesNum(self):
        return self.country_num
```

Implementation – Preparing Data

Country	Index	Country	Index
Arabic	0	Chinese	1
Czech	2	Dutch	3
English	4	French	5
German	6	Greek	7
Irish	8	Italian	9
Japanese	10	Korean	11
Polish	12	Portuguese	13
Russian	14	Scottish	15
Spanish	16	Vietnamese	17

Return country name giving index.

```
class NameDataset(Dataset):

    ......

    def getCountryDict(self):
        country_dict = dict()
        for idx, country_name in enumerate(self.country_list, 0):
            country_dict[country_name] = idx
        return country_dict

    def idx2country(self, index):
        return self.country_list[index]

    def getCountriesNum(self):
        return self.country_num
```

Implementation – Preparing Data

Country	Index	Country	Index
Arabic	0	Chinese	1
Czech	2	Dutch	3
English	4	French	5
German	6	Greek	7
Irish	8	Italian	9
Japanese	10	Korean	11
Polish	12	Portuguese	13
Russian	14	Scottish	15
Spanish	16	Vietnamese	17

```
class NameDataset(Dataset):

    ......

    def getCountryDict(self):
        country_dict = dict()
        for idx, country_name in enumerate(self.country_list, 0):
            country_dict[country_name] = idx
        return country_dict

    def idx2country(self, index):
        return self.country_list[index]

    def getCountriesNum(self):
        return self.country_num
```

Return the number of countries.

Implementation – Preparing Data

```
trainset = NameDataset(is_train_set=True)
trainloader = DataLoader(trainset, batch_size=BATCH_SIZE, shuffle=True)
testset = NameDataset(is_train_set=False)
testloader = DataLoader(testset, batch_size=BATCH_SIZE, shuffle=False)

N_COUNTRY = trainset.getCountriesNum()
```

Prepare Dataset and DataLoader

Implementation – Preparing Data

```
trainset = NameDataset(is_train_set=True)
trainloader = DataLoader(trainset, batch_size=BATCH_SIZE, shuffle=True)
testset = NameDataset(is_train_set=False)
testloader = DataLoader(testset, batch_size=BATCH_SIZE, shuffle=False)

N_COUNTRY = trainset.getCountriesNum()
```

N_COUNTRY is the output size of our model.

Implementation – Preparing Data

```
# Parameters
HIDDEN_SIZE = 100
BATCH_SIZE = 256
N_LAYER = 2
N_EPOCHS = 100
N_CHARS = 128
USE_GPU = False
```

```
trainset = NameDataset(is_train_set=True)
trainloader = DataLoader(trainset, batch_size=BATCH_SIZE,
testset = NameDataset(is_train_set=False)
testloader = DataLoader(testset, batch_size=BATCH_SIZE, shuffle=False)

N_COUNTRY = trainset.getCountriesNum()
```

N_COUNTRY is the output size of our model.

Implementation – Model Design

```
class RNNClassifier(torch.nn.Module):
    def __init__(self, input_size, hidden_size, output_size, n_layers=1, bidirectional=True):
        super(RNNClassifier, self).__init__()
        self.hidden_size = hidden_size
        self.n_layers = n_layers
        self.n_directions = 2 if bidirectional else 1

        self.embedding = torch.nn.Embedding(input_size, hidden_size)
        self.gru = torch.nn.GRU(hidden_size, hidden_size, n_layers,
                                bidirectional=bidirectional)
        self.fc = torch.nn.Linear(hidden_size * self.n_directions, output_size)

    def _init_hidden(self, batch_size):
        hidden = torch.zeros(self.n_layers * self.n_directions,
                             batch_size, self.hidden_size)
        return create_tensor(hidden)
```

Implementation – Model Design

```
class RNNClassifier(torch.nn.Module):
    def __init__(self, input_size, hidden_size, output_size, n_layers=1, bidirectional=True):
        super(RNNClassifier, self).__init__()
        self.hidden_size = hidden_size
        self.n_layers = n_layers
        self.n_directions = 2 if bidirectional else 1

        self.embedding = torch.nn.Embedding(input_size, hidden_size)
        self.gru = torch.nn.GRU(hidden_size, hidden_size, n_layers,
                                bidirectional=bidirectional)
        self.fc = torch.nn.Linear(hidden_size * self.n_directions, output_size)

    def _init_hidden(self, batch_size):
        hidden = torch.zeros(self.n_layers * self.n_directions,
                             batch_size, self.hidden_size)
        return create_tensor(hidden)
```

Parameters of GRU layer.

Implementation – Model Design

```
class RNNClassifier(torch.nn.Module):
    def __init__(self, input_size, hidden_siz
        super(RNNClassifier, self).__init__()
        self.hidden_size = hidden_size
        self.n_layers = n_layers
        self.n_directions = 2 if bidirectiona

        self.embedding = torch.nn.Embedding(input_size, hidden_size)
        self.gru = torch.nn.GRU(hidden_size, hidden_size, n_layers,
                                bidirectional=bidirectional)
        self.fc = torch.nn.Linear(hidden_size * self.n_directions, output_size)

    def _init_hidden(self, batch_size):
        hidden = torch.zeros(self.n_layers * self.n_directions,
                             batch_size, self.hidden_size)
        return create_tensor(hidden)
```

The input of Embedding Layer with shape:

$$(seqLen, batchSize)$$

The output of Embedding Layer with shape:

$$(seqLen, batchSize, hiddenSize)$$

Implementation – Model Design

```
class RNNClassifier(torch.nn.Module):
    def __init__(self, input_size, hid
        super(RNNClassifier, self).__i
        self.hidden_size = hidden_size
        self.n_layers = n_layers
        self.n_directions = 2 if bidir

        self.embedding = torch.nn.Embe
        self.gru = torch.nn.GRU(hidde
                                bidirectional=bidirectional)
        self.fc = torch.nn.Linear(hidden_size * self.n_directions, output_size)

    def _init_hidden(self, batch_size):
        hidden = torch.zeros(self.n_layers * self.n_directions,
                             batch_size, self.hidden_size)
        return create_tensor(hidden)
```

The **inputs** of GRU Layer with shape:

$input: (seqLen, batchSize, hiddenSize)$

$hidden: (nLayers * nDirections, batchSize, hiddenSize)$

The **outputs** of GRU Layer with shape:

$output: (seqLen, batchSize, hiddenSize * nDirections)$

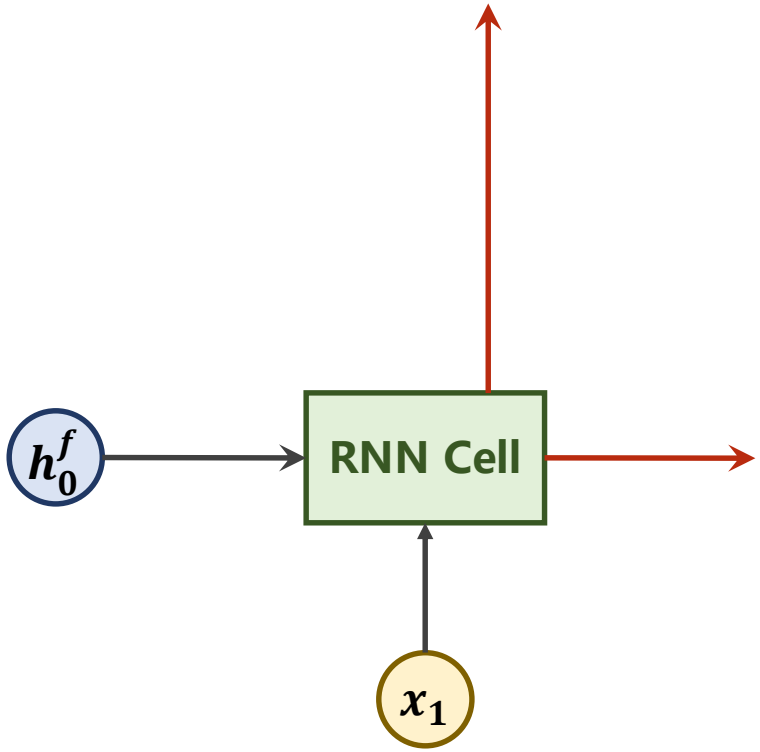
$hidden: (nLayers * nDirections, batchSize, hiddenSize)$

Implementation – Model Design

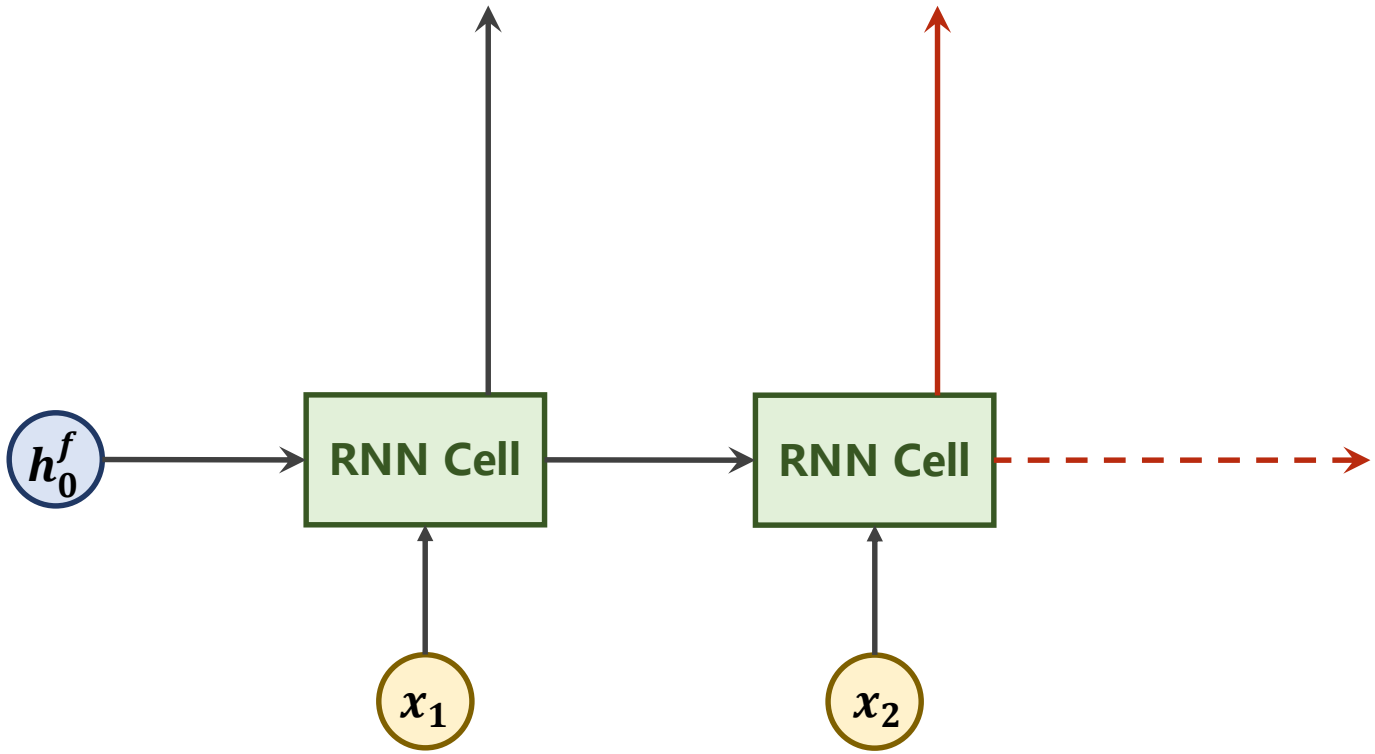
```
class RNNClassifier(torch.nn.Module):
    def __init__(self, input_size, hidden_size, output_size, n_layers=1, bidirectional=True):
        super(RNNClassifier, self).__init__()
        self.hidden_size = hidden_size
        self.n_layers = n_layers
        self.n_directions = 2 if bidirectional else 1

        self.embedding = torch.nn.Embedding(input_size, hidden_size)
        self.gru = torch.nn.GRU(hidden_size, hidden_size, n_layers,
                                bidirectional=bidirectional)
        self.fc = torch.nn.Linear(hidden_size * self.n_directions, output_size)

    def _init_hidden(self, batch_size):
        hidden = torch.zeros(self.n_layers * self.n_directions,
                             batch_size, self.hidden_size)
        return create_tensor(hidden)
```

What is the Bi-Direction RNN/LSTM/GRU?

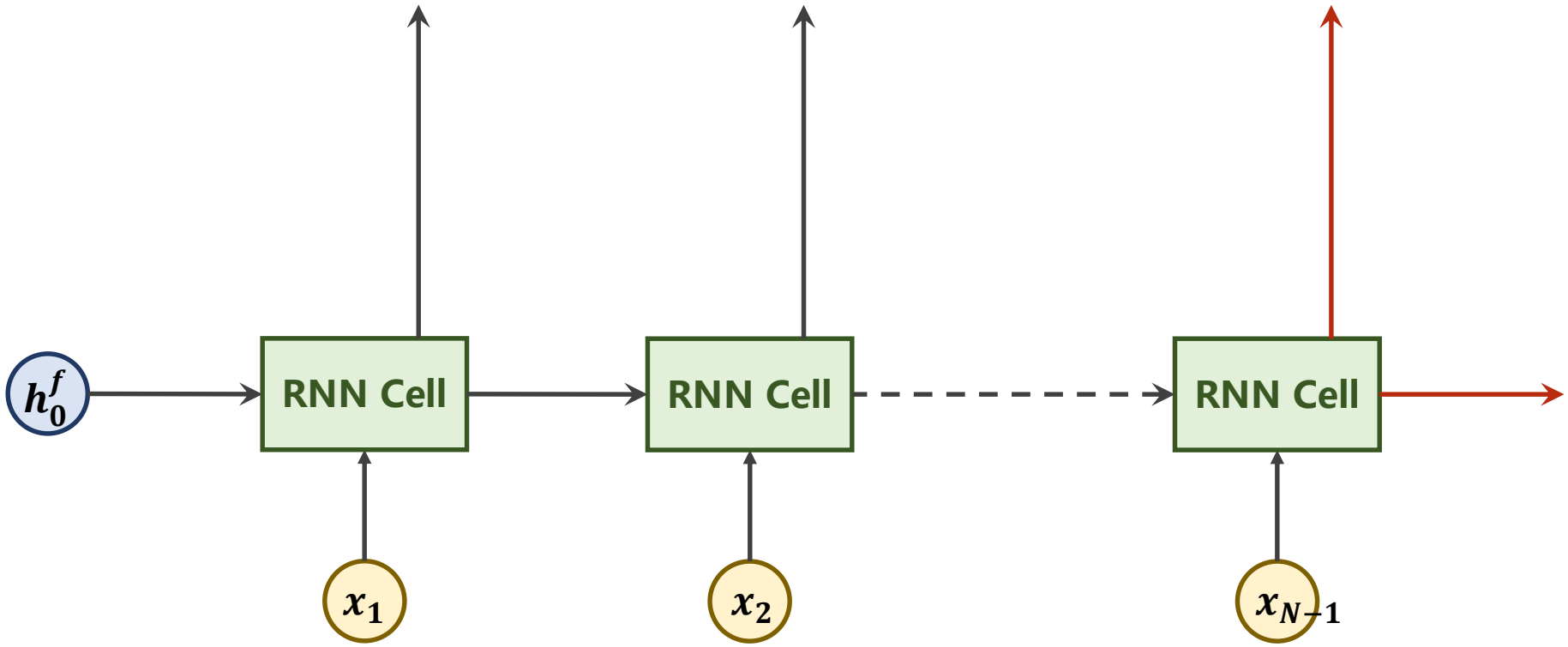
Implementation – Bi-direction RNN/LSTM/GRU



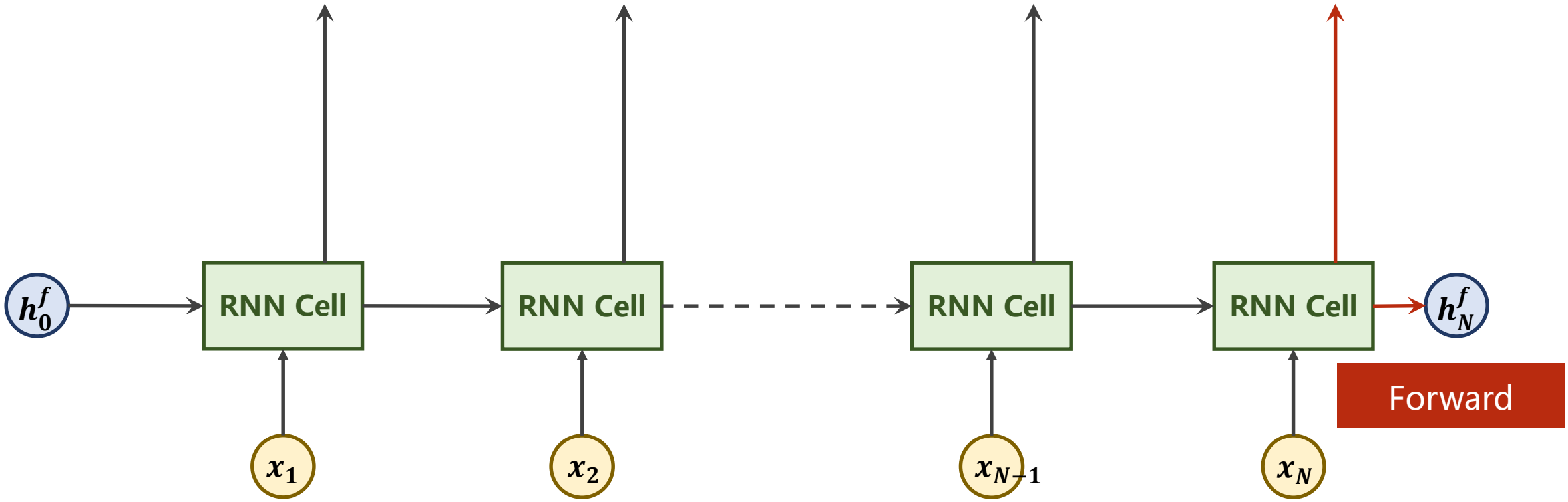
Implementation – Bi-direction RNN/LSTM/GRU



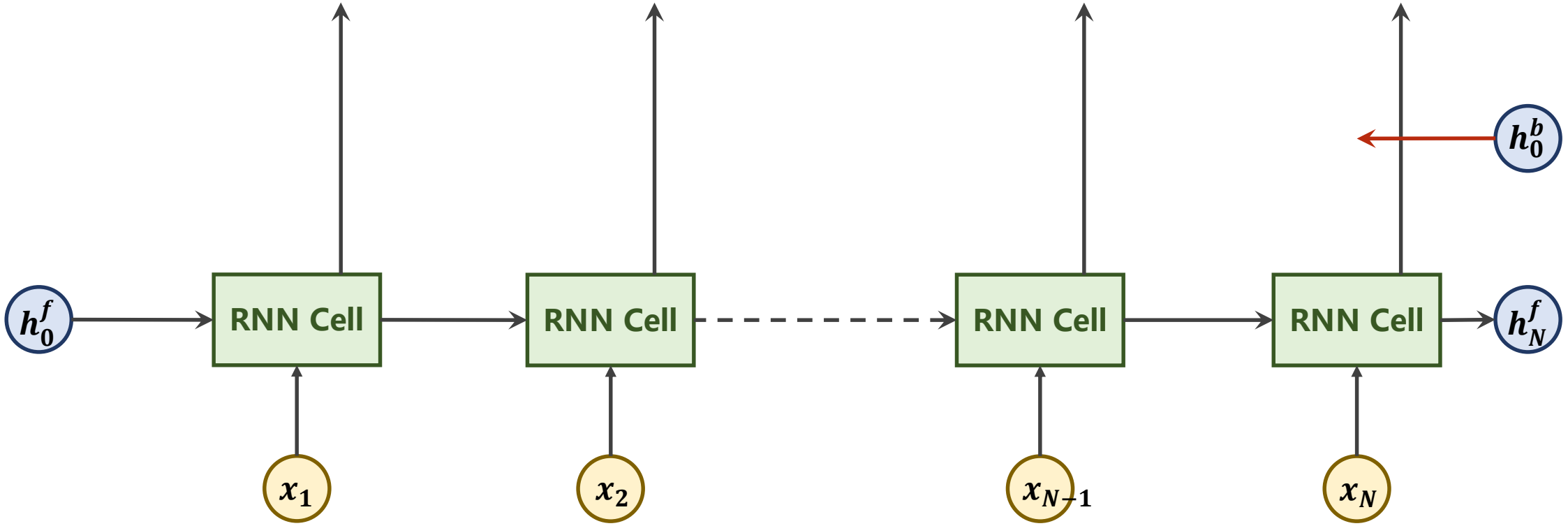
Implementation – Bi-direction RNN/LSTM/GRU



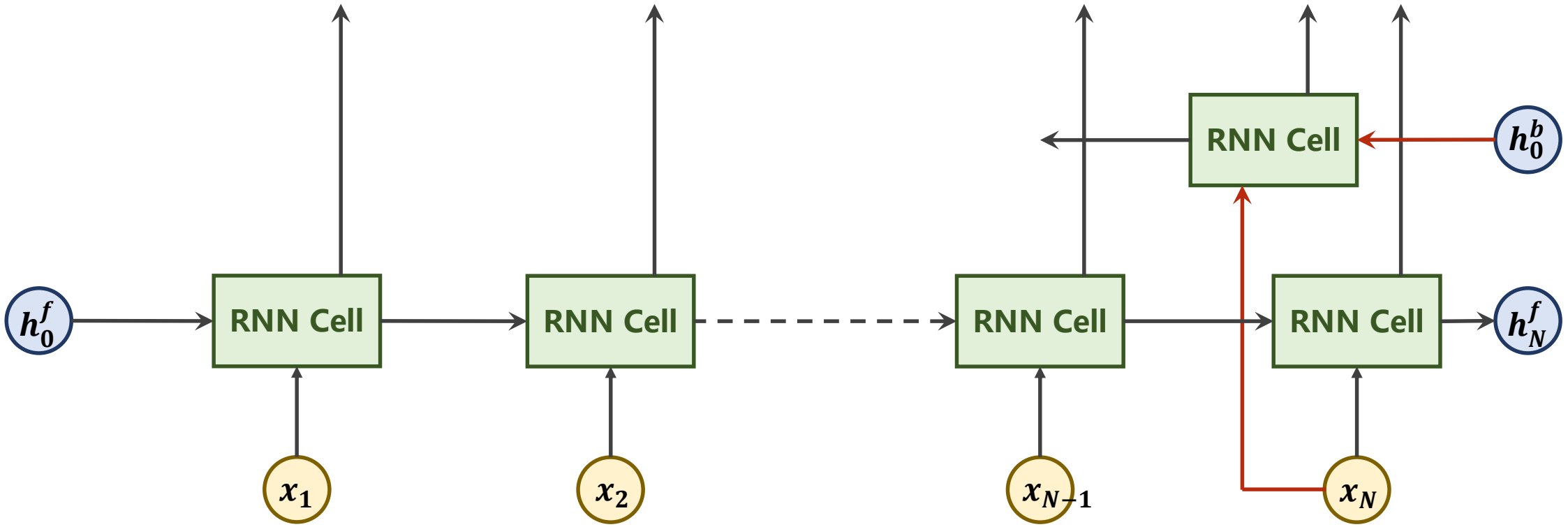
Implementation – Bi-direction RNN/LSTM/GRU



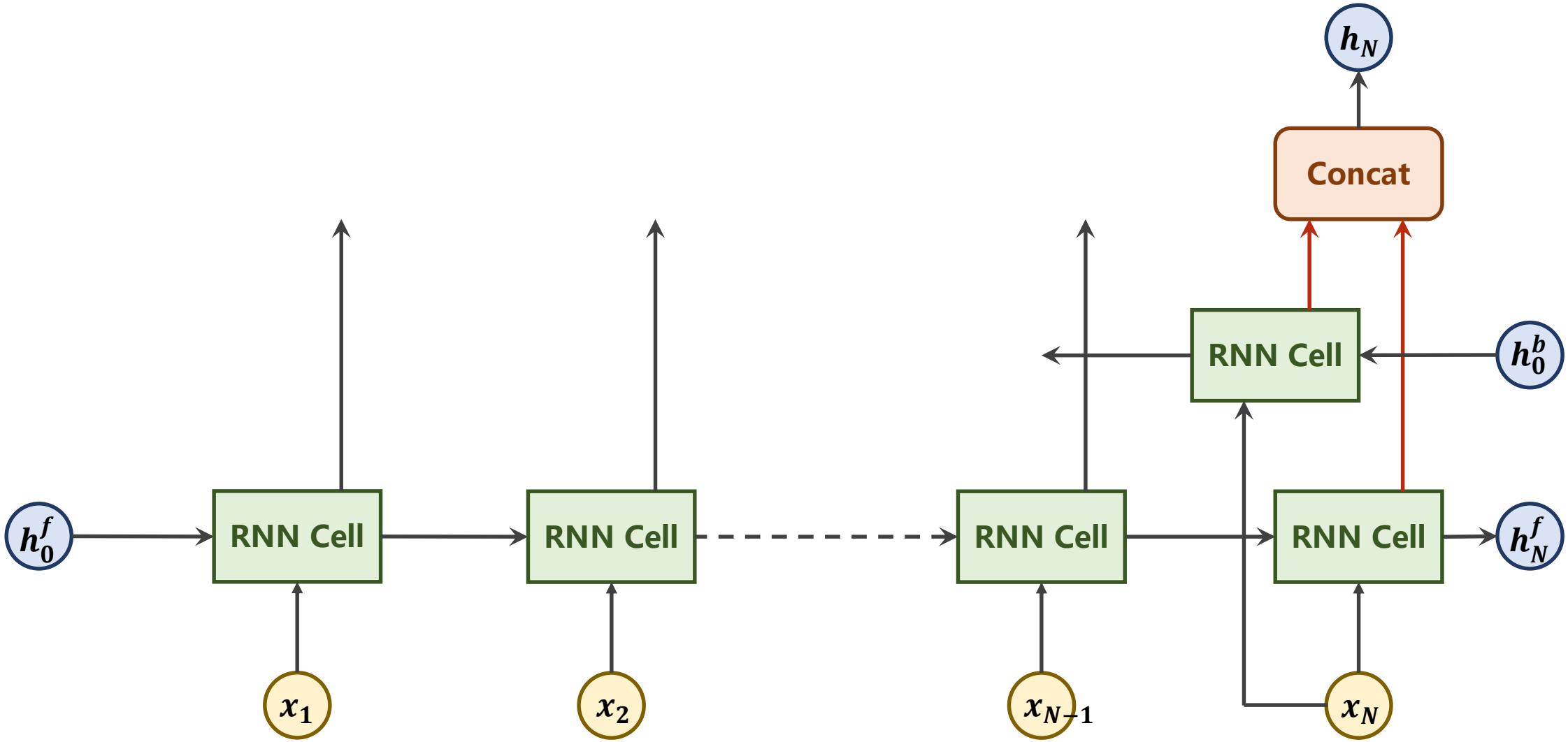
Implementation – Bi-direction RNN/LSTM/GRU



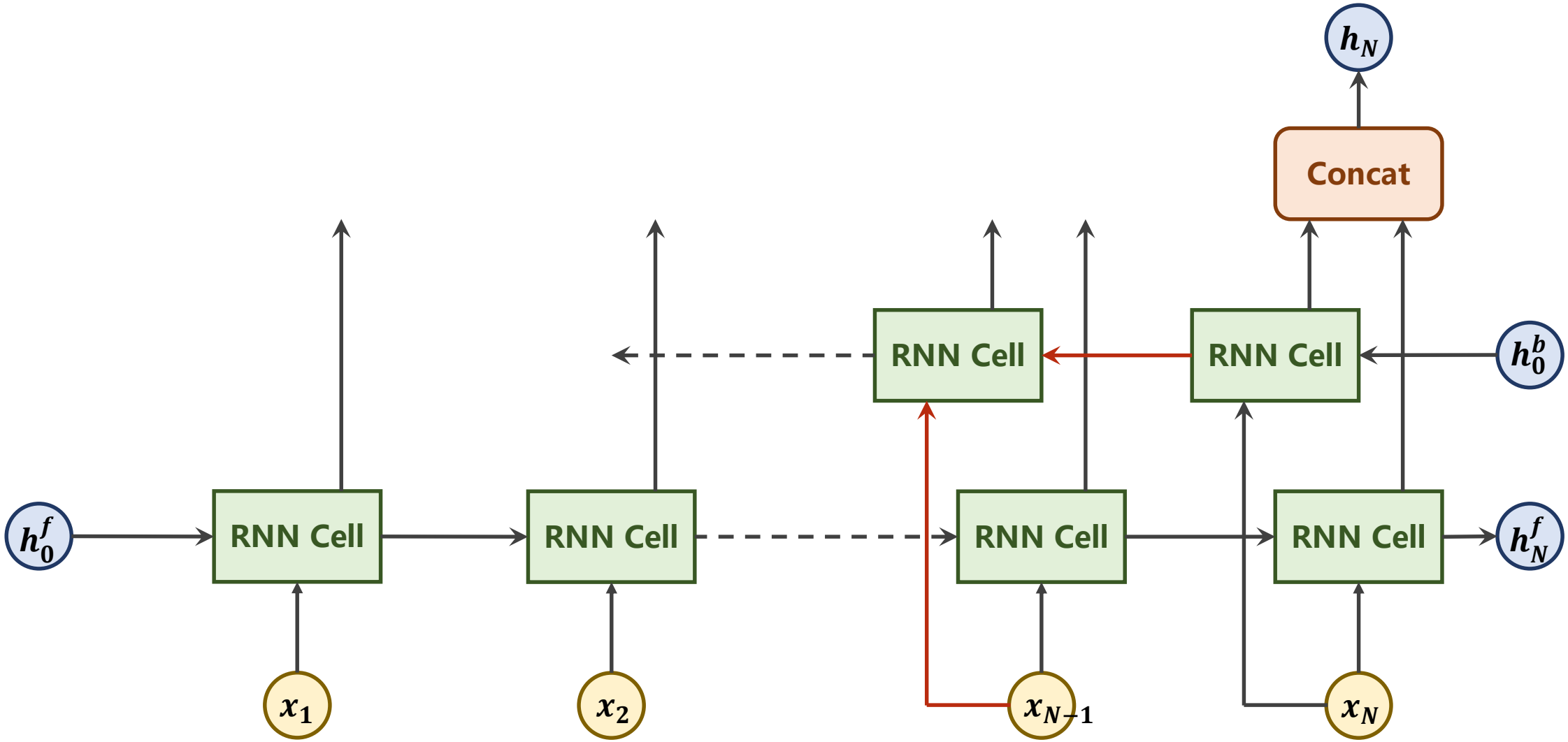
Implementation – Bi-direction RNN/LSTM/GRU



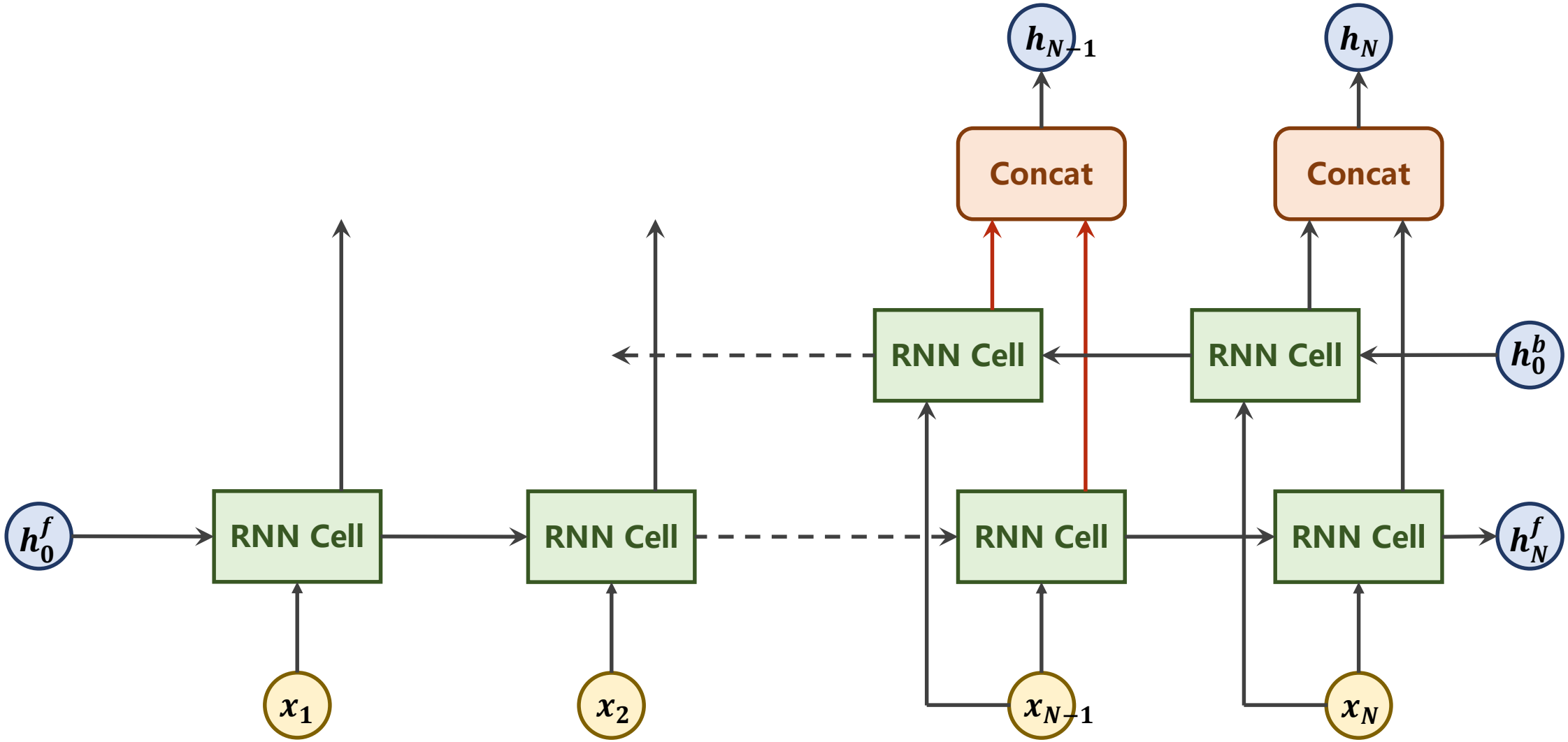
Implementation – Bi-direction RNN/LSTM/GRU



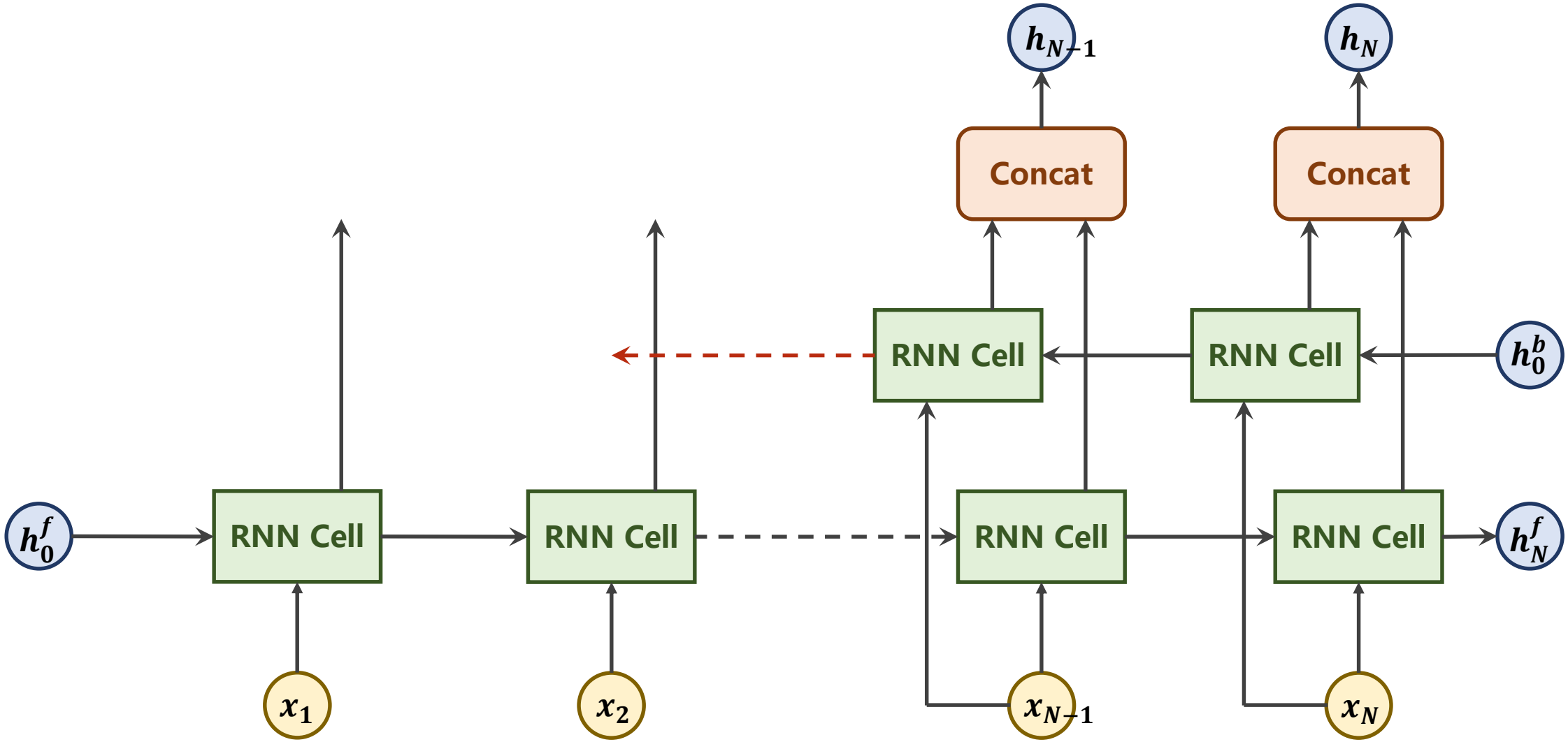
Implementation – Bi-direction RNN/LSTM/GRU



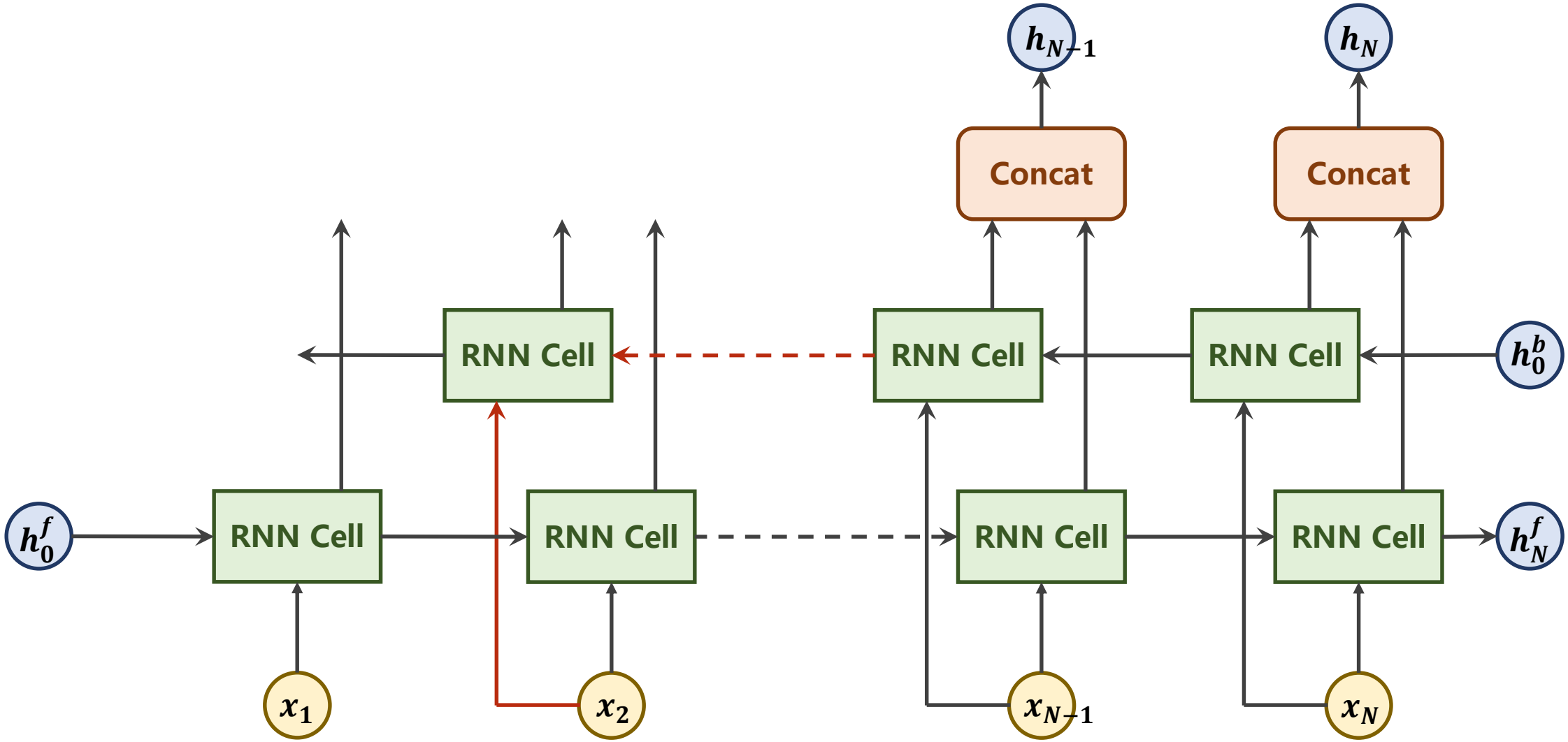
Implementation – Bi-direction RNN/LSTM/GRU



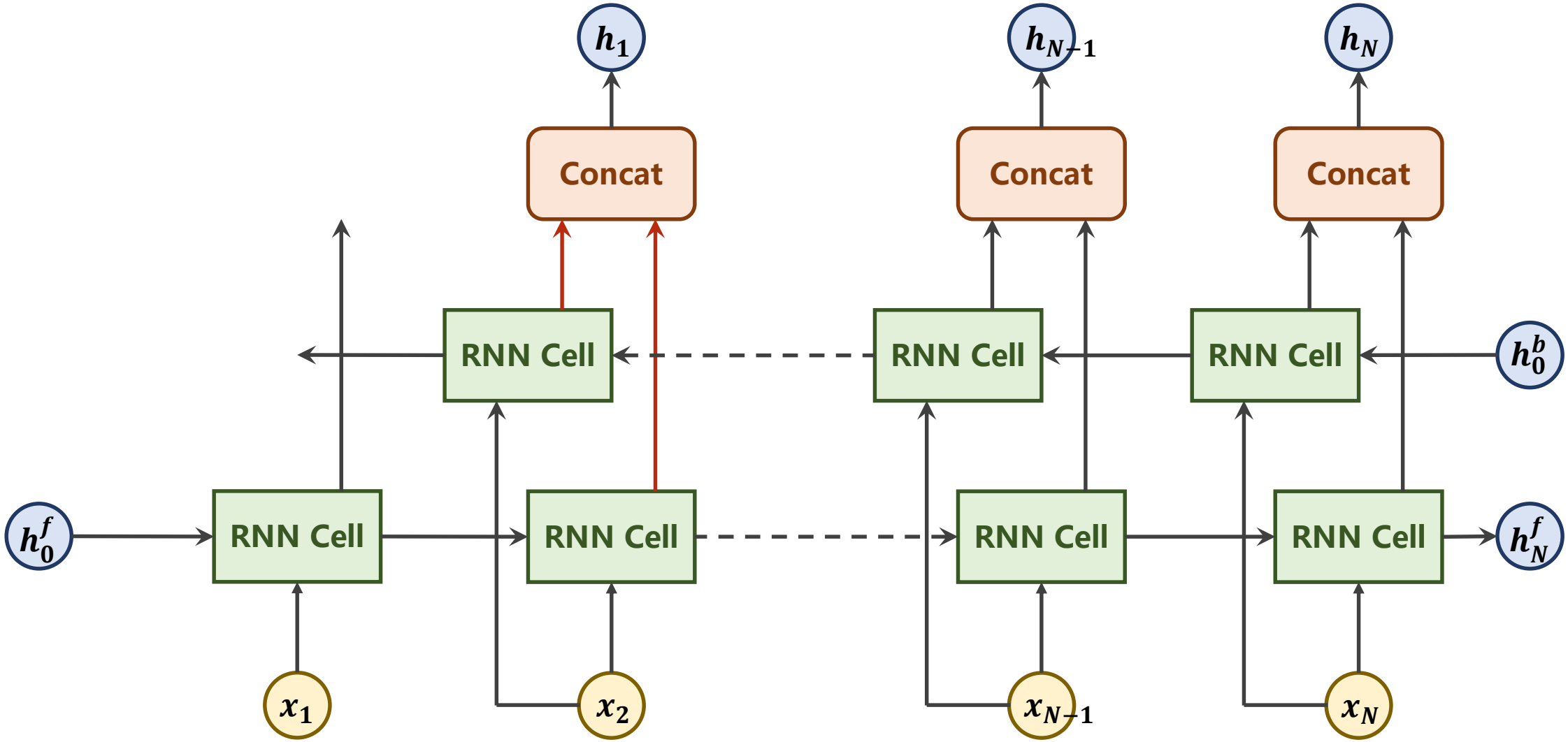
Implementation – Bi-direction RNN/LSTM/GRU



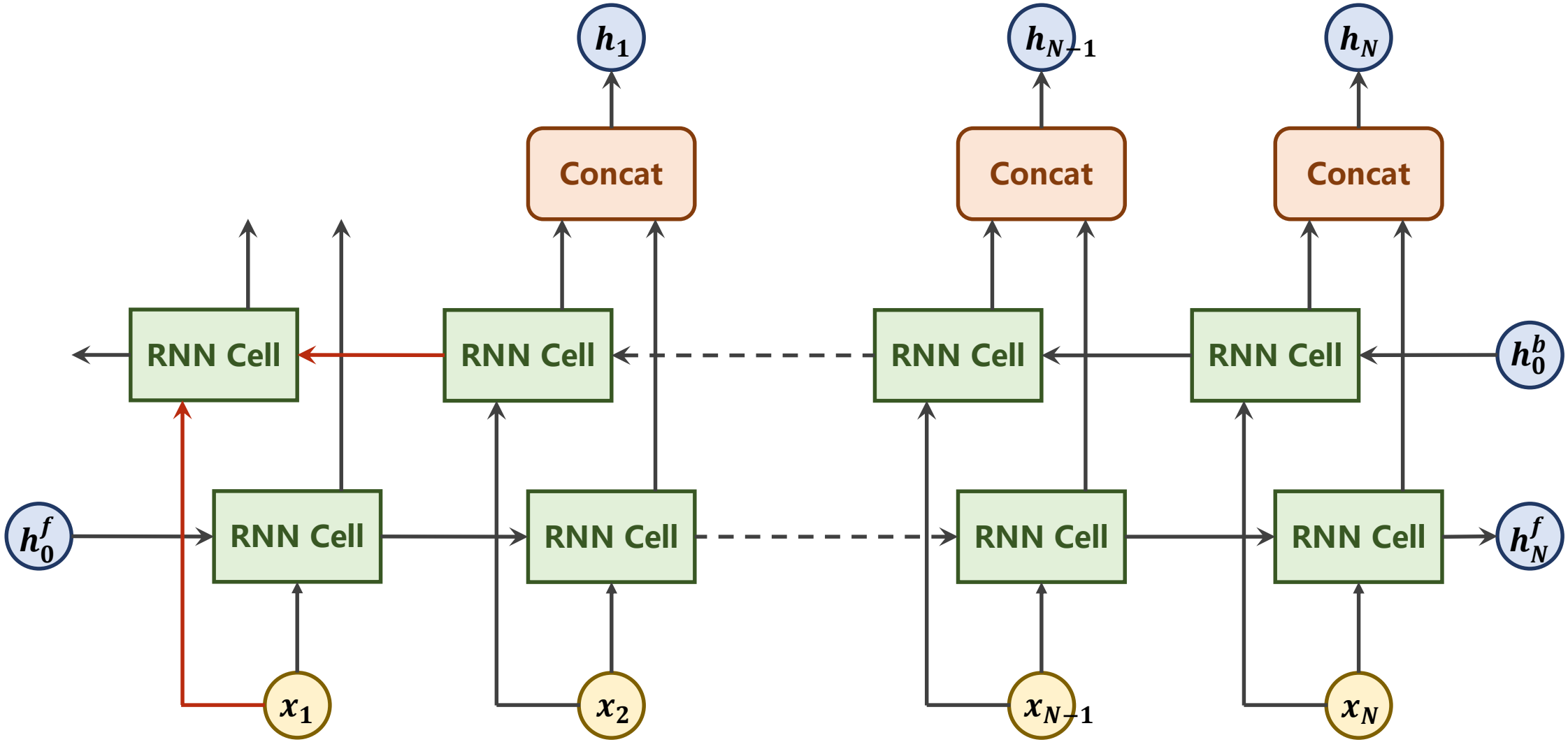
Implementation – Bi-direction RNN/LSTM/GRU



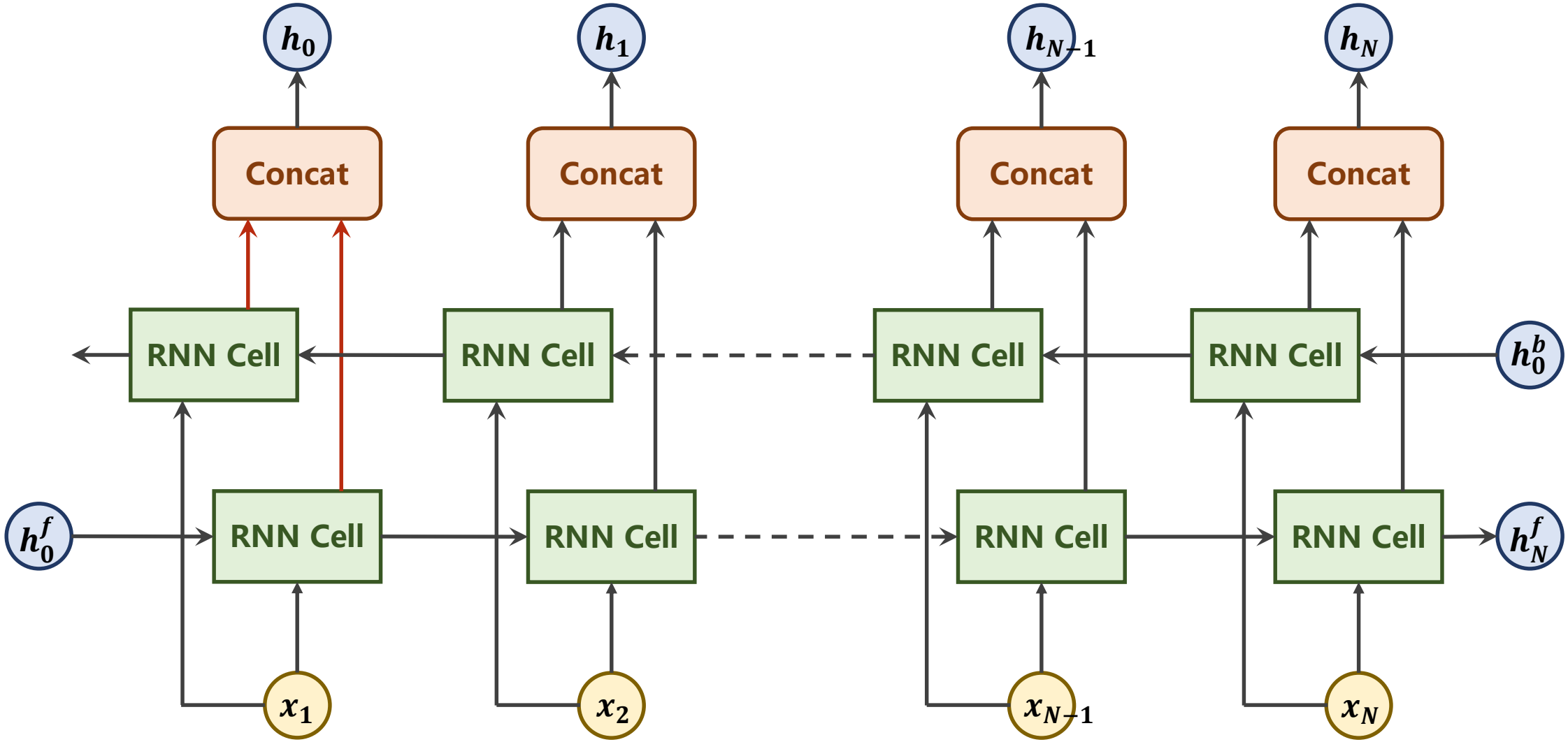
Implementation – Bi-direction RNN/LSTM/GRU



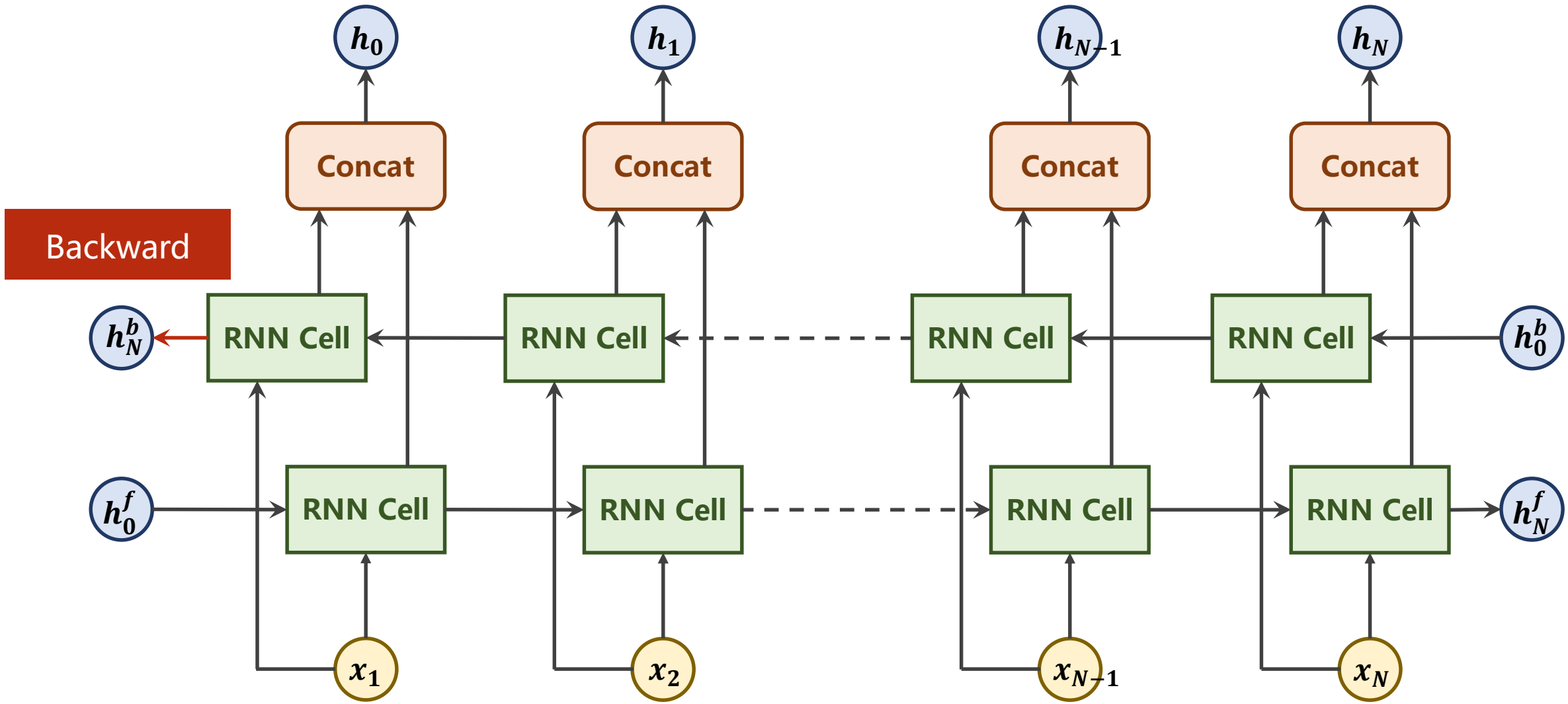
Implementation – Bi-direction RNN/LSTM/GRU



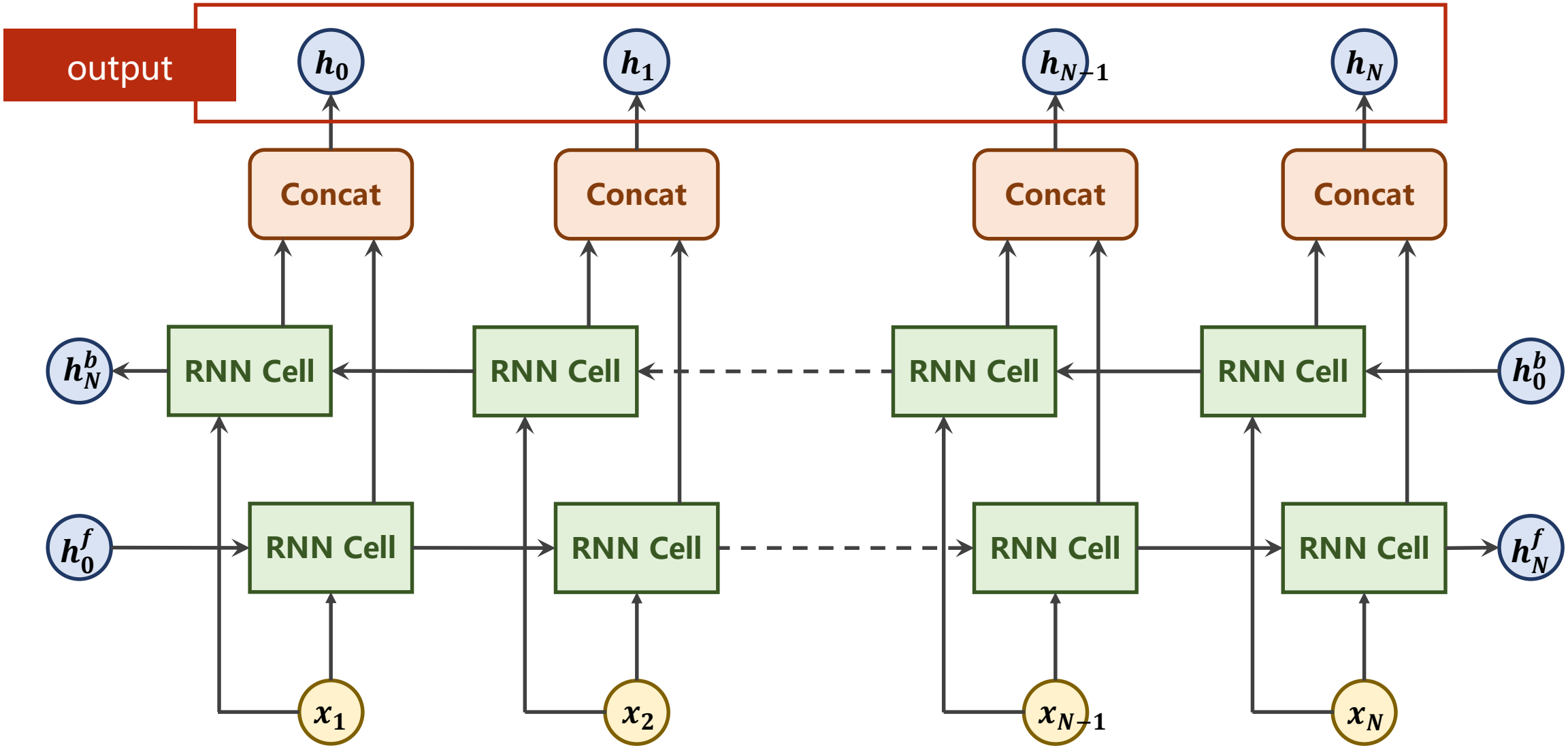
Implementation – Bi-direction RNN/LSTM/GRU



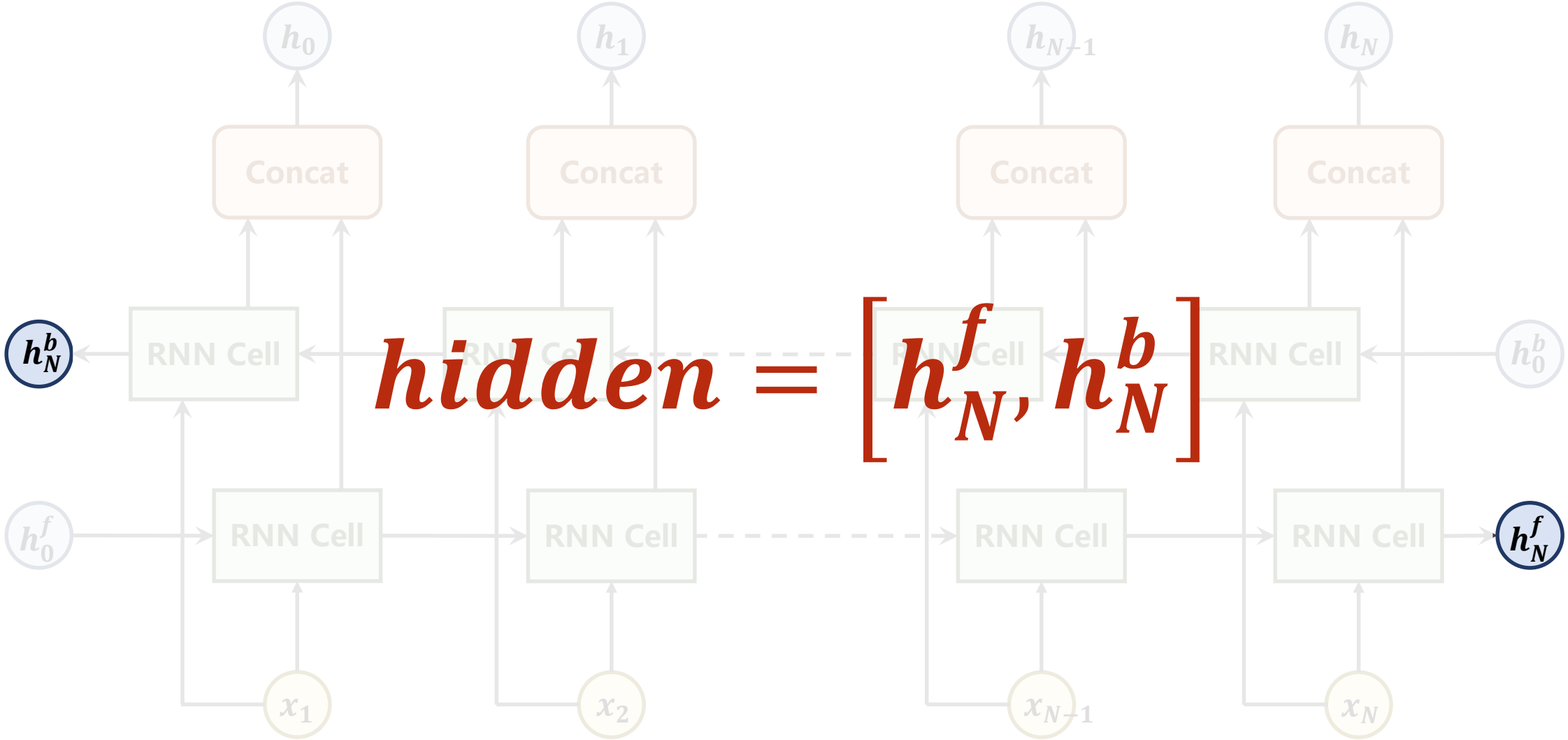
Implementation – Bi-direction RNN/LSTM/GRU



Implementation – Bi-direction RNN/LSTM/GRU



Implementation – Bi-direction RNN/LSTM/GRU



Implementation – Model Design

```
class RNNClassifier(torch.nn.Module):
    def __init__(self, input_size, hid
        super(RNNClassifier, self).__i
        self.hidden_size = hidden_size
        self.n_layers = n_layers
        self.n_directions = 2 if bidir

        self.embedding = torch.nn.Embe
        self.gru = torch.nn.GRU(hidde
                                bidirectional=bidirectional)
        self.fc = torch.nn.Linear(hidden_size * self.n_directions, output_size)

    def _init_hidden(self, batch_size):
        hidden = torch.zeros(self.n_layers * self.n_directions,
                             batch_size, self.hidden_size)
        return create_tensor(hidden)
```

The **inputs** of GRU Layer with shape:

$input: (seqLen, batchSize, hiddenSize)$

$hidden: (nLayers * nDirections, batchSize, hiddenSize)$

The **outputs** of GRU Layer with shape:

$output: (seqLen, batchSize, hiddenSize * nDirections)$

$hidden: (nLayers * nDirections, batchSize, hiddenSize)$

Implementation – Model Design

```
class RNNClassifier(torch.nn.Module):
    def __init__(self, input_size, hid
        super(RNNClassifier, self).__i
        self.hidden_size = hidden_size
        self.n_layers = n_layers
        self.n_directions = 2 if bidir

        self.embedding = torch.nn.Embe
        self.gru = torch.nn.GRU(hidden_
                                bidirectional=bidirectional)
        self.fc = torch.nn.Linear(hidden_size * self.n_directions, output_size)

    def _init_hidden(self, batch_size):
        hidden = torch.zeros(self.n_layers * self.n_directions,
                             batch_size, self.hidden_size)
        return create_tensor(hidden)
```

The **inputs** of GRU Layer with shape:

$input: (seqLen, batchSize, hiddenSize)$

$hidden: (nLayers * nDirections, batchSize, hiddenSize)$

The **outputs** of GRU Layer with shape:

$output: (seqLen, batchSize, hiddenSize * nDirections)$

$hidden: (nLayers * \mathbf{nDirections}, batchSize, \mathbf{hiddenSize})$

Implementation – Model Design

```
class RNNClassifier(torch.nn.Module):
    def forward(self, input, seq_lengths):
        # input shape : B x S -> S x B
        input = input.t()
        batch_size = input.size(1)

        hidden = self._init_hidden(batch_size)
        embedding = self.embedding(input)

        # pack them up
        gru_input = pack_padded_sequence(embedding, seq_lengths)

        output, hidden = self.gru(gru_input, hidden)
        if self.n_directions == 2:
            hidden_cat = torch.cat([hidden[-1], hidden[-2]], dim=1)
        else:
            hidden_cat = hidden[-1]
        fc_output = self.fc(hidden_cat)
        return fc_output
```

Implementation – Model Design

```
class RNNClassifier(torch.nn.Module):
    def forward(self, input, seq_lengths):
        # input shape : B x S -> S x B
        input = input.t()
        batch_size = input.size(1)

        hidden = self._init_hidden(batch_size)
        embedding = self.embedding(input)

        # pack them up
        gru_input = pack_padded_sequence(embedding, seq_lengths)

        output, hidden = self.gru(gru_input, hidden)
        if self.n_directions == 2:
            hidden_cat = torch.cat([hidden[-1], hidden[-2]], dim=1)
        else:
            hidden_cat = hidden[-1]
        fc_output = self.fc(hidden_cat)
        return fc_output
```

Implementation – Model Design

```
class RNNClassifier(torch.nn.Module):
    def forward(self, input, seq_lengths):
        # input shape : B x S -> S x B
        input = input.t()
        batch_size = input.size(1)
```

After padding
[77 97 99 108 101 97 110 0 0 0]
[86 97 106 110 105 99 104 121 0 0]
[78 97 115 105 107 111 118 115 107 121]
[85 115 97 109 105 0 0 0 0 0]
[70 105 111 110 105 110 0 0 0 0]
[83 104 97 114 107 101 121 0 0 0]
[66 97 108 97 103 117 108 0 0 0]
[80 97 107 104 114 105 110 0 0 0]
[84 97 110 115 104 111 0 0 0 0]

$(batchSize, seqLen)$

After transpose								
77	86	78	85	70	83	66	80	84
97	97	97	115	105	104	97	97	97
99	106	115	97	111	97	108	107	110
108	110	105	109	110	114	97	104	115
101	105	107	105	105	107	103	114	104
97	99	111	0	110	101	117	105	111
110	104	118	0	0	121	108	110	0
0	101	115	0	0	0	0	0	0
0	0	107	0	0	0	0	0	0
0	0	121	0	0	0	0	0	0

$(seqLen, batchSize)$

Implementation – Model Design

```
class RNNClassifier(torch.nn.Module):
    def forward(self, input, seq_lengths):
        # input shape : B x S -> S x B
        input = input.t()
        batch_size = input.size(1)

        hidden = self._init_hidden(batch_size)
        embedding = self.embedding(input)

        # pack them up
        gru_input = pack_padded_sequence(embedding, seq_lengths)

        output, hidden = self.gru(gru_input, hidden)
        if self.n_directions == 2:
            hidden_cat = torch.cat([hidden[-1], hidden[-2]], dim=1)
        else:
            hidden_cat = hidden[-1]
        fc_output = self.fc(hidden_cat)
        return fc_output
```

Save batch-size for make initial ***hidden***.

Implementation – Model Design

```
class RNNClassifier(torch.nn.Module):
    def forward(self, input, seq_lengths):
        # input shape : B x S -
        input = input.t()
        batch_size = input.size

        hidden = self._init_hidden(batch_size)
        embedding = self.embedding(input)

        # pack them up
        gru_input = pack_padded_sequence(embedding, seq_lengths)

        output, hidden = self.gru(gru_input, hidden)
        if self.n_directions == 2:
            hidden_cat = torch.cat([hidden[-1], hidden[-2]], dim=1)
        else:
            hidden_cat = hidden[-1]
        fc_output = self.fc(hidden_cat)
        return fc_output
```

Initial hidden with shape:

$(nLayer * nDirections, batchSize, hiddenSize)$

Implementation – Model Design

```
class RNNClassifier(torch.nn.Module):
    def forward(self, input, seq_lengths):
        # input shape : B x S -> S x B
        input = input.t()
        batch_size = input.size

        hidden = self._init_hidden(batch_size)
        embedding = self.embedding(input)

        # pack them up
        gru_input = pack_padded_sequence(embedding, seq_lengths)

        output, hidden = self.gru(gru_input, hidden)
        if self.n_directions == 2:
            hidden_cat = torch.cat([hidden[-1], hidden[-2]], dim=1)
        else:
            hidden_cat = hidden[-1]
        fc_output = self.fc(hidden_cat)
        return fc_output
```

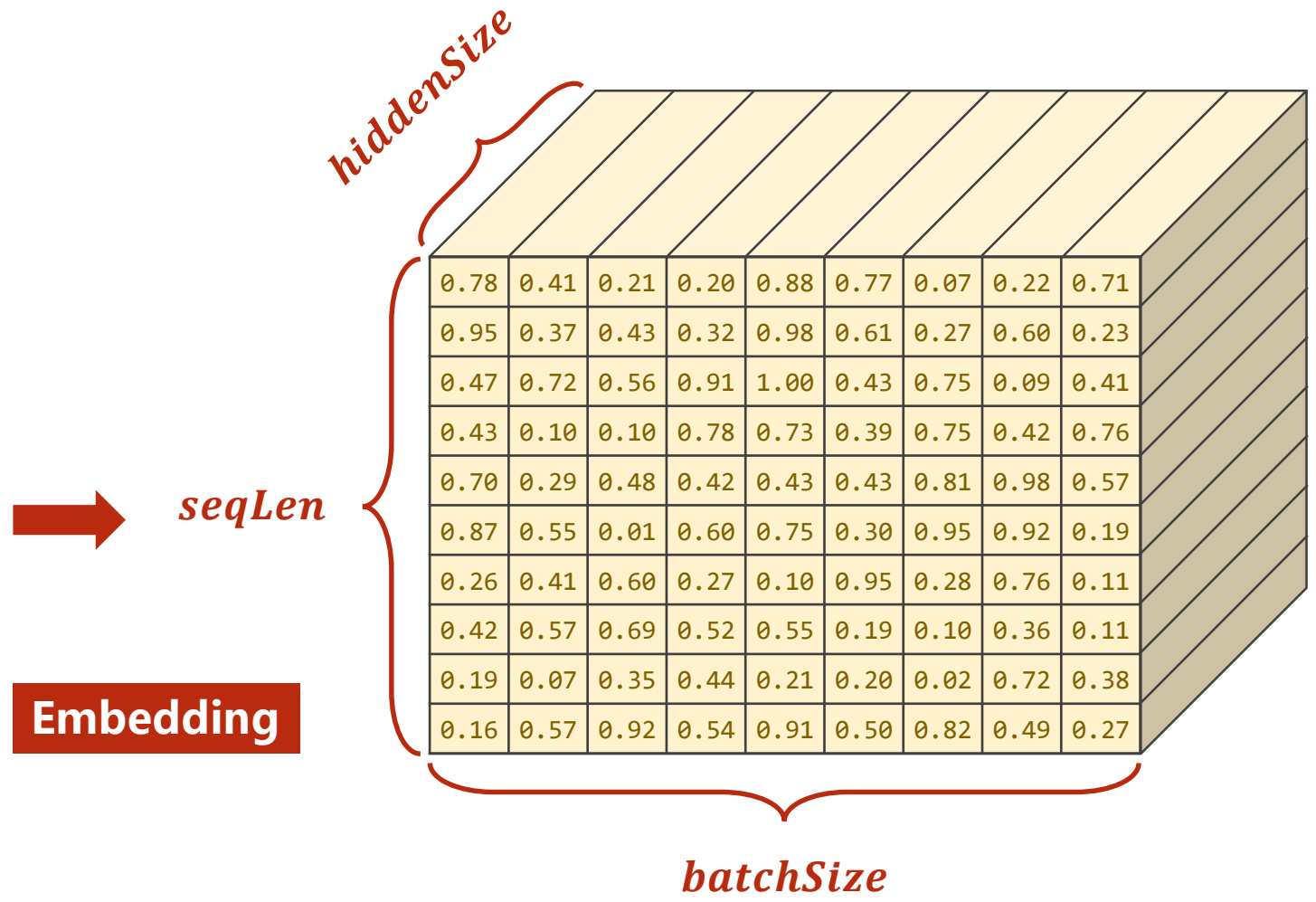
Result of embedding with shape:

$$(seqLen, batchSize, hiddenSize)$$

Implementation – Model Design

77	86	78	85	70	83	66	80	84
97	97	97	115	105	104	97	97	97
99	106	115	97	111	97	108	107	110
108	110	105	109	110	114	97	104	115
101	105	107	105	105	107	103	114	104
97	99	111	0	110	101	117	105	111
110	104	118	0	0	121	108	110	0
0	101	115	0	0	0	0	0	0
0	0	107	0	0	0	0	0	0
0	0	121	0	0	0	0	0	0

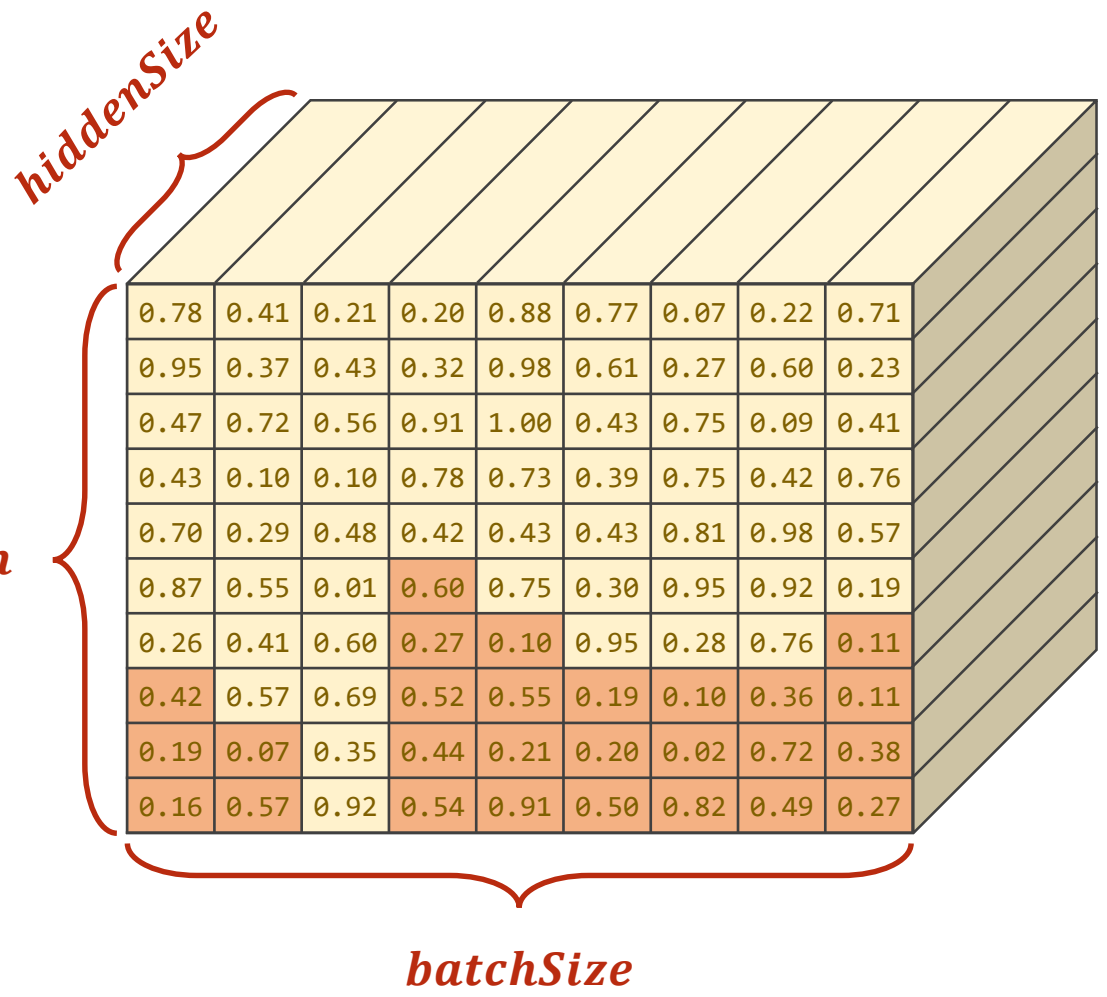
Embedding



Implementation – Model Design

77	86	78	85	70	83	66	80	84
97	97	97	115	105	104	97	97	97
99	106	115	97	111	97	108	107	110
108	110	105	109	110	114	97	104	115
101	105	107	105	105	107	103	114	104
97	99	111	0	110	101	117	105	111
110	104	118	0	0	121	108	110	0
0	101	115	0	0	0	0	0	0
0	0	107	0	0	0	0	0	0
0	0	121	0	0	0	0	0	0

Embedding



Implementation – Model Design

```
class RNNClassifier(torch.
    def forward(self, inpu
        # input shape : B
        input = input.t()
        batch_size = input.

        hidden = self._init_hidden(batch_size)
        embedding = self.embedding(input)

        # pack them up
        gru_input = pack_padded_sequence(embedding, seq_lengths)

        output, hidden = self.gru(gru_input, hidden)
        if self.n_directions == 2:
            hidden_cat = torch.cat([hidden[-1], hidden[-2]], dim=1)
        else:
            hidden_cat = hidden[-1]
        fc_output = self.fc(hidden_cat)
        return fc_output
```

The first parameter with shape:

$$(seqLen, batchSize, hiddenSize)$$

The second parameter is a tensor, which is a list of sequence length of each batch element.

Implementation – Model Design

```
class RNNClassifier(torch.nn.Module):
    def forward(self, input, seq_lengths):
        # input shape : B x S -> S x B
        input = input.t()
        batch_size = input.size(1)

        hidden = self._init_hid
        embedding = self.embedd

        # pack them up
        gru_input = pack_padded_sequence(embedding, seq_lengths)

        output, hidden = self.gru(gru_input, hidden)
        if self.n_directions == 2:
            hidden_cat = torch.cat([hidden[-1], hidden[-2]], dim=1)
        else:
            hidden_cat = hidden[-1]
        fc_output = self.fc(hidden_cat)
        return fc_output
```

Result of embedding with shape:

$$(seqLen, batchSize, hiddenSize)$$

Implementation – Model Design

```
class RNNClassifier(torch.
    def forward(self, inpu
        # input shape : B
        input = input.t()
        batch_size = input.

        hidden = self._init_hidden(batch_size)
        embedding = self.embedding(input)

        # pack them up
        gru_input = pack_padded_sequence(embedding, seq_lengths)

        output, hidden = self.gru(gru_input, hidden)
        if self.n_directions == 2:
            hidden_cat = torch.cat([hidden[-1], hidden[-2]], dim=1)
        else:
            hidden_cat = hidden[-1]
        fc_output = self.fc(hidden_cat)
        return fc_output
```

The first parameter with shape:

$$(seqLen, batchSize, hiddenSize)$$

The second parameter is a tensor, which is a list of sequence length of each batch element.

It returns a **PackedSquence** object.

Implementation – Model Design

torch.nn.utils.rnn.PackedSequence(*cls*, **args*) [source]

Holds the data and list of `batch_sizes` of a packed sequence.

All RNN modules accept packed sequences as inputs.

Note

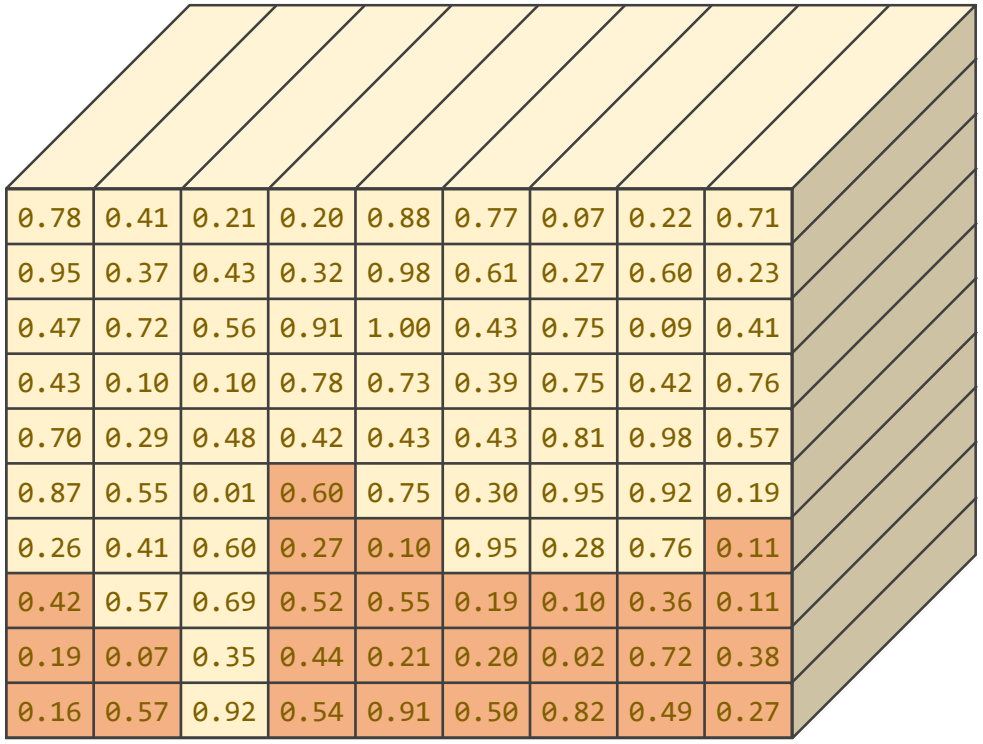
Instances of this class should never be created manually. They are meant to be instantiated by functions like `pack_padded_sequence()`.

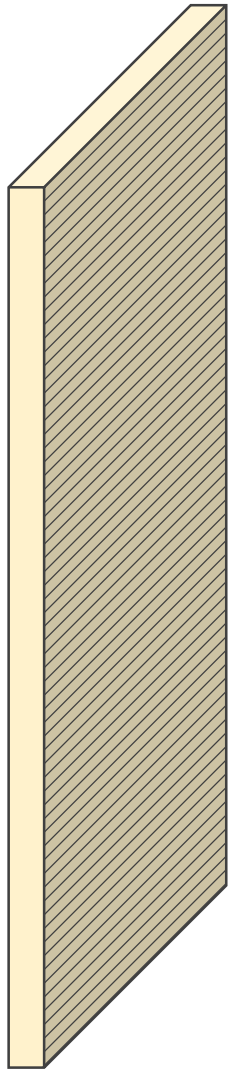
Batch sizes represent the number elements at each sequence step in the batch, not the varying sequence lengths passed to `pack_padded_sequence()`. For instance, given data `abc` and *x* the `PackedSequence` would contain data `axbc` with `batch_sizes=[2,1,1]`.

Variables:

- **data** (*Tensor*) – Tensor containing packed sequence
- **batch_sizes** (*Tensor*) – Tensor of integers holding information about the batch size at each sequence step

Implementation – Model Design





It cannot work!

7	8	10	5	6	7	7	7	6

Must be sorted by descendent

Implementation – Model Design

77	86	78	85	70	83	66	80	84
97	97	97	115	105	104	97	97	97
99	106	115	97	111	97	108	107	110
108	110	105	109	110	114	97	104	115
101	105	107	105	105	107	103	114	104
97	99	111	0	110	101	117	105	111
110	104	118	0	0	121	108	110	0
0	101	115	0	0	0	0	0	0
0	0	107	0	0	0	0	0	0
0	0	121	0	0	0	0	0	0

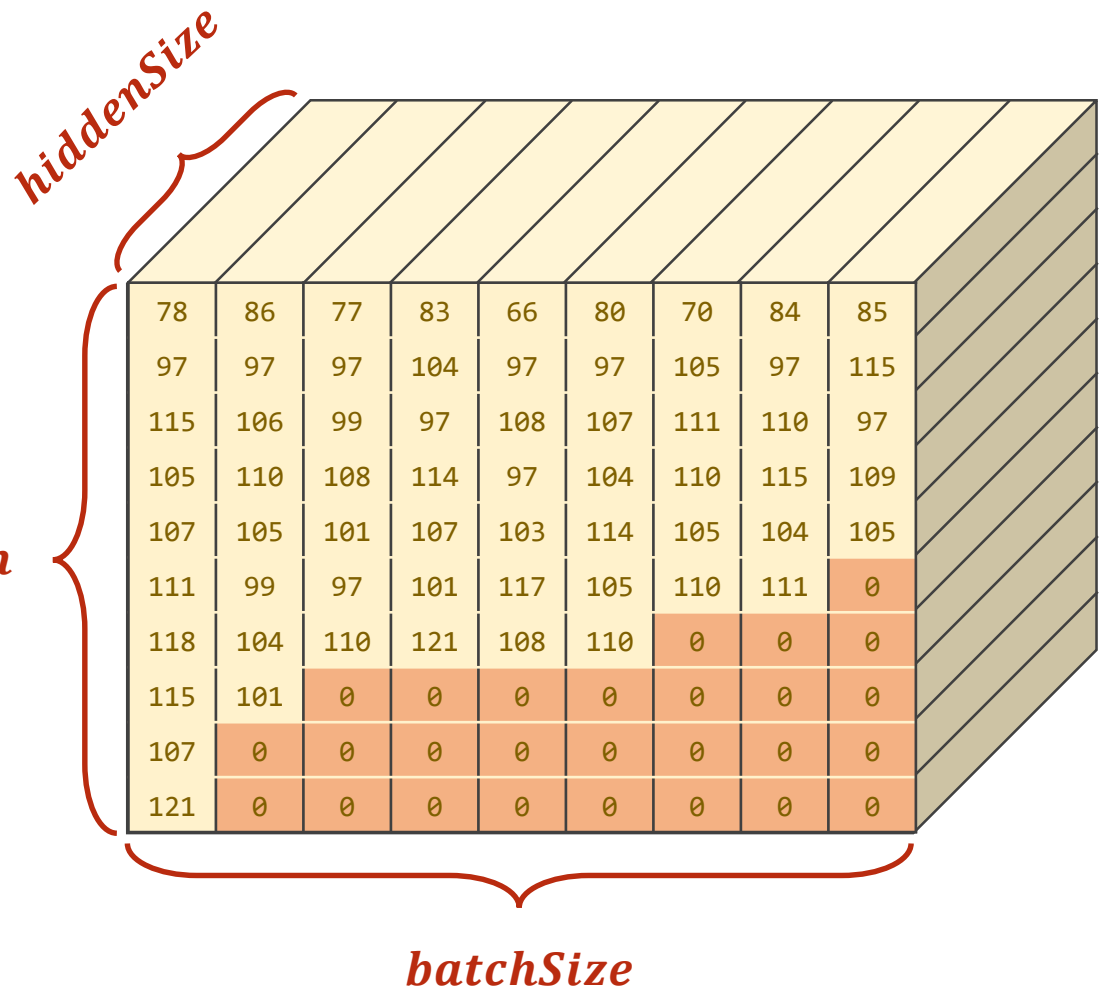
78	86	77	83	66	80	70	84	85
97	97	97	104	97	97	105	97	115
115	106	99	97	108	107	111	110	97
105	110	108	114	97	104	110	115	109
107	105	101	107	103	114	105	104	105
111	99	97	101	117	105	110	111	0
118	104	110	121	108	110	0	0	0
115	101	0	0	0	0	0	0	0
107	0	0	0	0	0	0	0	0
121	0	0	0	0	0	0	0	0

We have to sort the batch element by length of sequence.

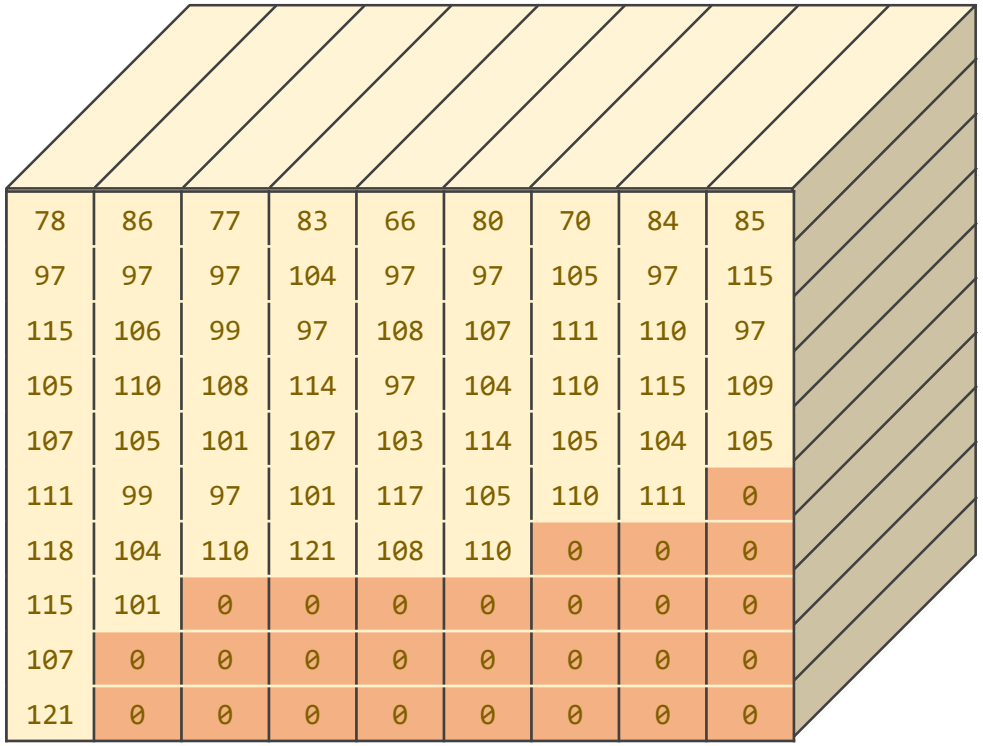
Implementation – Model Design

78	86	77	83	66	80	70	84	85
97	97	97	104	97	97	105	97	115
115	106	99	97	108	107	111	110	97
105	110	108	114	97	104	110	115	109
107	105	101	107	103	114	105	104	105
111	99	97	101	117	105	110	111	0
118	104	110	121	108	110	0	0	0
115	101	0	0	0	0	0	0	0
107	0	0	0	0	0	0	0	0
121	0	0	0	0	0	0	0	0

Embedding



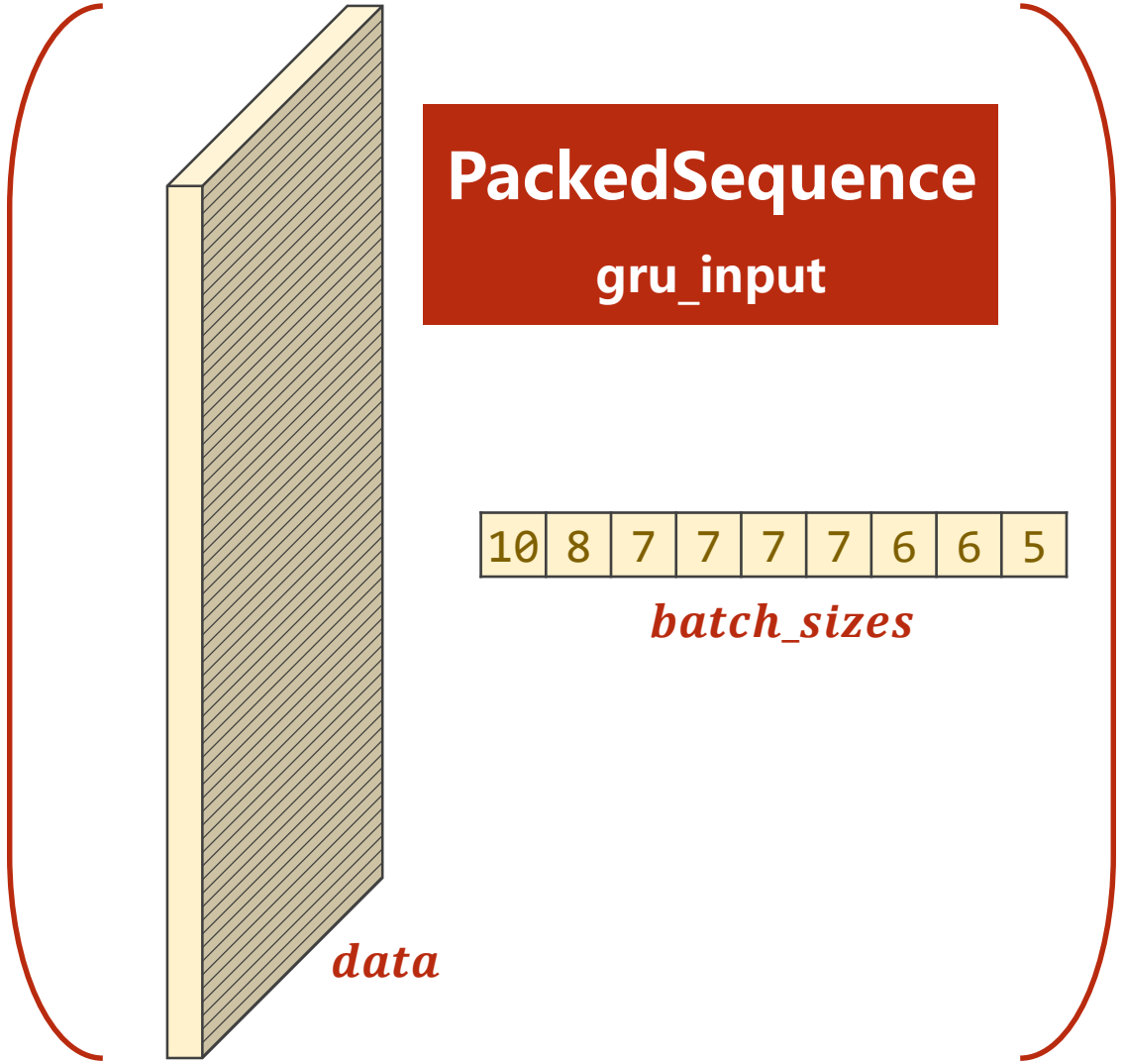
Implementation – Model Design



embedding

10	8	7	7	7	7	6	6	5

seq_length



10	8	7	7	7	7	6	6	5

batch_sizes

Implementation – Model Design

```
class RNNClassifier(torch.
    def forward(self, inpu
        # input shape : B
        input = input.t()
        batch_size = input.

        hidden = self._init_hidden(batch_size)
        embedding = self.e

        # pack them up
        gru_input = pack_padded_sequence(embedding, seq_lengths)

        output, hidden = self.gru(gru_input, hidden)
        if self.n_directions == 2:
            hidden_cat = torch.cat([hidden[-1], hidden[-2]], dim=1)
        else:
            hidden_cat = hidden[-1]
        fc_output = self.fc(hidden_cat)
        return fc_output
```

The first parameter with shape:

$$(seqLen, batchSize, hiddenSize)$$

The second parameter is a tensor, which is a list of sequence length of each batch element.

It returns a **PackedSquence** object.

Implementation – Model Design

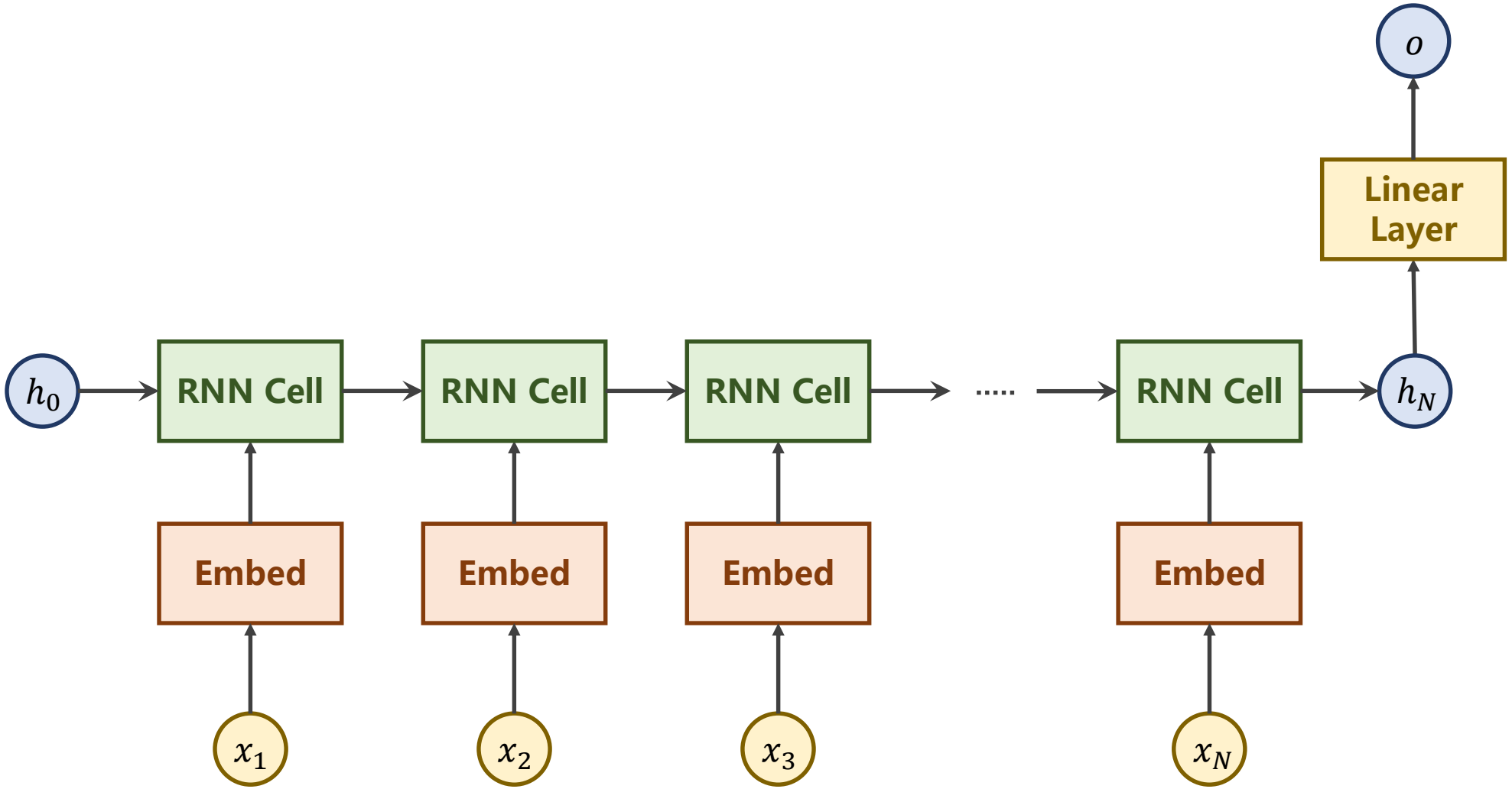
```
class RNNClassi
    def forward
        # input
        input =
        batch_s

        hidden = self._init_hidden(batch_size)
        embedding = self.embedding(input)

        # pack them up
        gru_input = pack_padded_sequence(embedding, seq_lengths)

        output, hidden = self.gru(gru_input, hidden)
        if self.n_directions == 2:
            hidden_cat = torch.cat([hidden[-1], hidden[-2]], dim=1)
        else:
            hidden_cat = hidden[-1]
        fc_output = self.fc(hidden_cat)
        return fc_output
```

The output is a PackedSequence object, actually it is a tuple.

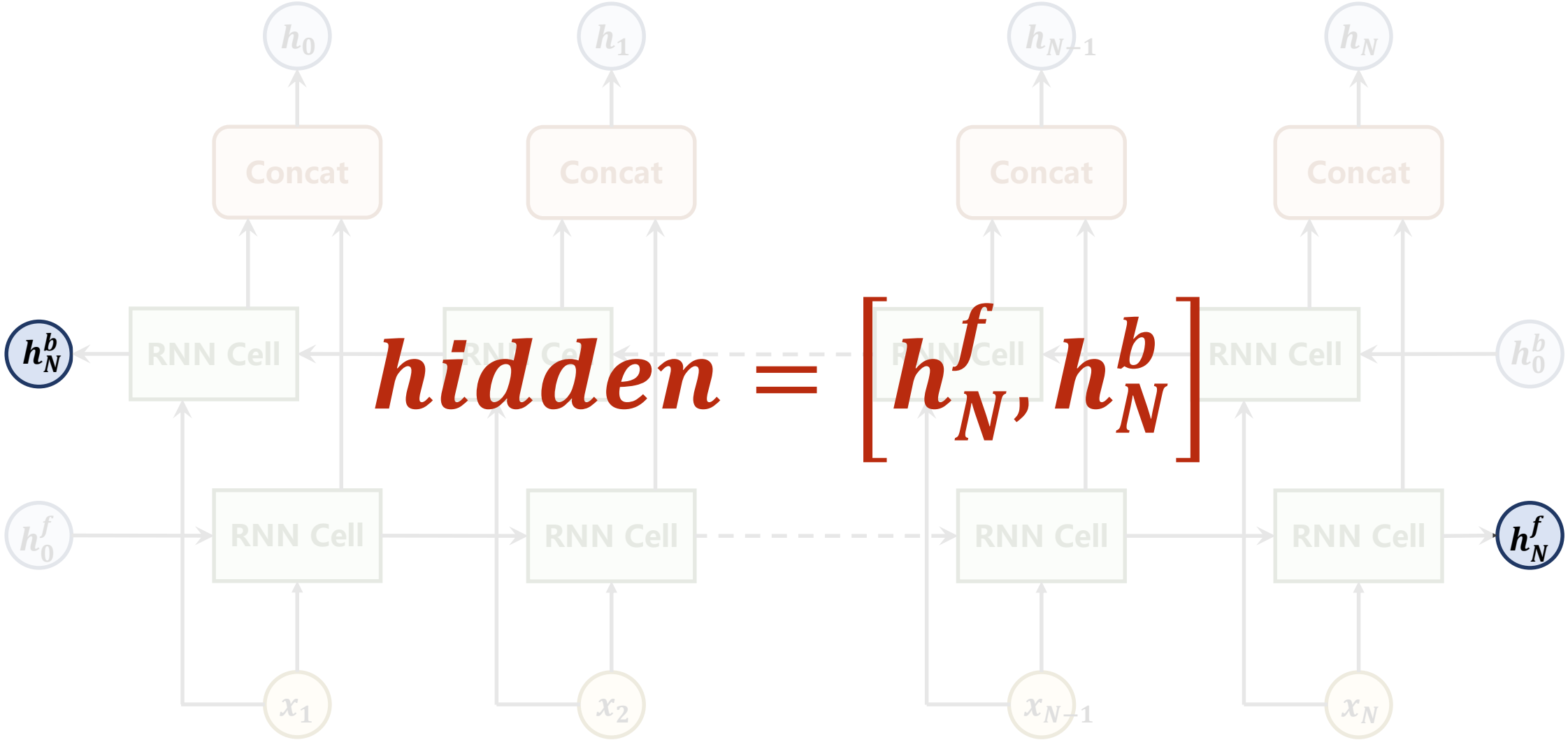
the shape of ***hidden***, which we concerned, with shape:

$$(nLayers * nDirection, batchSize, hiddenSize)$$

Implementation – Model Design



Implementation – Model Design



Implementation – Model Design

```
class RNNClassifier(torch.nn.Module):
    def forward(self, input, seq_lengths):
        # input shape : B x S -> S x B
        input = input.t()
        batch_size = input.size(1)

        hidden = self._init_hidden(batch_size)
        embedding = self.embedding(input)

        # pack
        gru_inp

        output, hidden = self.gru(gru_input, hidden)
        if self.n_directions == 2:
            hidden_cat = torch.cat([hidden[-1], hidden[-2]], dim=1)
        else:
            hidden_cat = hidden[-1]
        fc_output = self.fc(hidden_cat)
        return fc_output
```

If we use bidirectional GRU, the forward hidden and backward hidden should be concatenate.

Implementation – Model Design

```
class RNNClassifier(torch.nn.Module):
    def forward(self, input, seq_lengths):
        # input shape : B x S -> S x B
        input = input.t()
        batch_size = input.size(1)

        hidden = self._init_hidden(batch_size)
        embedding = self.embedding(input)

        # pack them up
        gru_input = pack_padded_sequence(embedding, seq_lengths)

        output, hidden = self.gru(gru_input, hidden)
        if self.n_directions == 2:
            hidden_cat = torch.cat([hidden[-1], hidden[-2]], dim=1)
        else:
            hidden_cat = hidden[-1]
        fc_output = self.fc(hidden_cat)
        return fc_output
```

Use linear classifier.

Implementation – Model Design

```
class RNNClassifier(torch.nn.Module):
    def forward(self, input, seq_lengths):
        # input shape : B x S -> S x B
        input = input.t()
        batch_size = input.size(1)

        hidden = self._init_hidden(batch_size)
        embedding = self.embedding(input)

        # pack them up
        gru_input = pack_padded_sequence(embedding, seq_lengths)

        output, hidden = self.gru(gru_input, hidden)
        if self.n_directions == 2:
            hidden_cat = torch.cat([hidden[-1], hidden[-2]], dim=1)
        else:
            hidden_cat = hidden[-1]
        fc_output = self.fc(hidden_cat)
        return fc_output
```

Implementation – Convert name to tensor

Name
Maclean
Vajnichy
Nasikovsky
Usami
Fionin
Sharkey
Balagul
Pakhrin
Tansho

78	86	77	83	66	80	70	84	85
97	97	97	104	97	97	105	97	115
115	106	99	97	108	107	111	110	97
105	110	108	114	97	104	110	115	109
107	105	101	107	103	114	105	104	105
111	99	97	101	117	105	110	111	0
118	104	110	121	108	110	0	0	0
115	101	0	0	0	0	0	0	0
107	0	0	0	0	0	0	0	0
121	0	0	0	0	0	0	0	0

10	8	7	7	7	7	6	6	5

Implementation – Convert name to tensor

Name	Characters	ASCII
Maclean	['M', 'a', 'c', 'l', 'e', 'a', 'n']	[77 97 99 108 101 97 110]
Vajnichy	['V', 'a', 'j', 'n', 'i', 'c', 'h', 'y']	[86 97 106 110 105 99 104 121]
Nasikovsky	['N', 'a', 's', 'i', 'k', 'o', 'v', 's', 'k', 'y']	[78 97 115 105 107 111 118 115 107 121]
Usami	['U', 's', 'a', 'm', 'i']	[85 115 97 109 105]
Fionin	['F', 'i', 'o', 'n', 'i', 'n']	[70 105 111 110 105 110]
Sharkey	['S', 'h', 'a', 'r', 'k', 'e', 'y']	[83 104 97 114 107 101 121]
Balagul	['B', 'a', 'l', 'a', 'g', 'u', 'l']	[66 97 108 97 103 117 108]
Pakhrin	['P', 'a', 'k', 'h', 'r', 'i', 'n']	[80 97 107 104 114 105 110]
Tansho	['T', 'a', 'n', 's', 'h', 'o']	[84 97 110 115 104 111]

Implementation – Convert name to tensor

ASCII
[77 97 99 108 101 97 110]
[86 97 106 110 105 99 104 121]
[78 97 115 105 107 111 118 115 107 121]
[85 115 97 109 105]
[70 105 111 110 105 110]
[83 104 97 114 107 101 121]
[66 97 108 97 103 117 108]
[80 97 107 104 114 105 110]
[84 97 110 115 104 111]

After padding
[77 97 99 108 101 97 110   0   0   0]
[86 97 106 110 105 99 104 121   0   0]
[78 97 115 105 107 111 118 115 107 121]
[85 115 97 109 105   0   0   0   0   0]
[70 105 111 110 105 110   0   0   0   0]
[83 104 97 114 107 101 121   0   0   0]
[66 97 108 97 103 117 108   0   0   0]
[80 97 107 104 114 105 110   0   0   0]
[84 97 110 115 104 111   0   0   0   0]

Implementation – Convert name to tensor

After padding
[77 97 99 108 101 97 110 0 0 0]
[86 97 106 110 105 99 104 121 0 0]
[78 97 115 105 107 111 118 115 107 121]
[85 115 97 109 105 0 0 0 0 0]
[70 105 111 110 105 110 0 0 0 0]
[83 104 97 114 107 101 121 0 0 0]
[66 97 108 97 103 117 108 0 0 0]
[80 97 107 104 114 105 110 0 0 0]
[84 97 110 115 104 111 0 0 0 0]

$(batchSize, seqLen)$

After transpose								
77	86	78	85	70	83	66	80	84
97	97	97	115	105	104	97	97	97
99	106	115	97	111	97	108	107	110
108	110	105	109	110	114	97	104	115
101	105	107	105	105	107	103	114	104
97	99	111	0	110	101	117	105	111
110	104	118	0	0	121	108	110	0
0	101	115	0	0	0	0	0	0
0	0	107	0	0	0	0	0	0
0	0	121	0	0	0	0	0	0

$(seqLen, batchSize)$

Implementation – Convert name to tensor

77	86	78	85	70	83	66	80	84
97	97	97	115	105	104	97	97	97
99	106	115	97	111	97	108	107	110
108	110	105	109	110	114	97	104	115
101	105	107	105	105	107	103	114	104
97	99	111	0	110	101	117	105	111
110	104	118	0	0	121	108	110	0
0	101	115	0	0	0	0	0	0
0	0	107	0	0	0	0	0	0
0	0	121	0	0	0	0	0	0

78	86	77	83	66	80	70	84	85
97	97	97	104	97	97	105	97	115
115	106	99	97	108	107	111	110	97
105	110	108	114	97	104	110	115	109
107	105	101	107	103	114	105	104	105
111	99	97	101	117	105	110	111	0
118	104	110	121	108	110	0	0	0
115	101	0	0	0	0	0	0	0
107	0	0	0	0	0	0	0	0
121	0	0	0	0	0	0	0	0

We have to sort the batch element by length of sequence.

Implementation – Convert name to tensor

```
def make_tensors(names, countries):
    sequences_and_lengths = [name2list(name) for name in names]
    name_sequences = [sl[0] for s
    seq_lengths = torch.LongTenso
    countries = countries.long()

    # make tensor of name, BatchSize x SeqLen
    seq_tensor = torch.zeros(len(name_sequences), seq_lengths.max()).long()
    for idx, (seq, seq_len) in enumerate(zip(name_sequences, seq_lengths), 0):
        seq_tensor[idx, :seq_len] = torch.LongTensor(seq)

    # sort by length to use pack_padded_sequence
    seq_lengths, perm_idx = seq_lengths.sort(dim=0, descending=True)
    seq_tensor = seq_tensor[perm_idx]
    countries = countries[perm_idx]

    return create_tensor(seq_tensor), \
           create_tensor(seq_lengths),\
           create_tensor(countries)
```

```
def name2list(name):
    arr = [ord(c) for c in name]
    return arr, len(arr)
```

Implementation – Convert name to tensor

```
def make_tensors(names, countries):
    sequences_and_lengths = [name2list(name) for name in names]
    name_sequences = [sl[0] for sl in sequences_and_lengths]
    seq_lengths = torch.LongTensor([sl[1] for sl in sequences_and_lengths])
    countries = countries.long()

    # make tensor of name, BatchSize
    seq_tensor = torch.zeros(len(nam
    for idx, (seq, seq_len) in enume
        seq_tensor[idx, :seq_len] =

    # sort by length to use pack_pad
    seq_lengths, perm_idx = seq_leng
    seq_tensor = seq_tensor[perm_idx
    countries = countries[perm_idx]

    return create_tensor(seq_tensor)
           create_tensor(seq_lengths
           create_tensor(countries)
```

ASCII
[77 97 99 108 101 97 110]
[86 97 106 110 105 99 104 121]
[78 97 115 105 107 111 118 115 107 121]
[85 115 97 109 105]
[70 105 111 110 105 110]
[83 104 97 114 107 101 121]
[66 97 108 97 103 117 108]
[80 97 107 104 114 105 110]
[84 97 110 115 104 111]

Implementation – Convert name to tensor

```
def make_tensors(names, countries):
    sequences_and_lengths = [name2list(name) for name in names]
    name_sequences = [sl[0] for sl in sequences_and_lengths]
    seq_lengths = torch.LongTensor([sl[1] for sl in sequences_and_lengths])
    countries = countries.long()

    # make tensor of name, BatchSize x SeqLen
    seq_tensor = torch.zeros(len(name_sequences), seq_lengths.max()).long()
    for idx, (seq, seq_len) in enumerate(zip(name_sequences, seq_lengths), 0):
        seq_tensor[idx, :seq_len] = torch.LongTensor(seq)

    # sort by length to use pack_padded_sequence
    seq_lengths, perm_idx = seq_lengths.sort(dim=0, descending=True)
    seq_tensor = seq_tensor[perm_idx]
    countries = countries[perm_idx]

    return create_tensor(seq_tensor), \
           create_tensor(seq_lengths),\
           create_tensor(countries)
```

Implementation – Convert name to tensor

```
def make_tensors(names, countries):
    sequences_and_lengths = [name2list(name) for name in names]
    name_sequences = [sl[0] for sl in sequences_and_lengths]
    seq_lengths = torch.LongTensor([sl[1] for sl in sequences_and_lengths])
    countries = countries.long()

    # make tensor of name, BatchSize x SeqLen
    seq_tensor = torch.zeros(len(name_sequences), seq_lengths.max()).long()
    for idx, (seq, seq_len) in enumerate(zip(name_sequences, seq_lengths), 0):
        seq_tensor[idx, :seq_len] = torch.LongTensor(seq)

    # sort by length to use pack_padded_sequence
    seq_lengths, perm_idx = seq_lengths.sort(dim=0, des
    seq_tensor = seq_tensor[perm_idx]
    countries = countries[perm_idx]

    return create_tensor(seq_tensor), \
           create_tensor(seq_lengths),\
           create_tensor(countries)
```

After padding

```
[ 77  97  99 108 101  97 110   0   0   0]
[ 86  97 106 110 105  99 104 121   0   0]
[ 78  97 115 105 107 111 118 115 107 121]
[ 85 115  97 109 105   0   0   0   0   0]
[ 70 105 111 110 105 110   0   0   0   0]
[ 83 104  97 114 107 101 121   0   0   0]
[ 66  97 108  97 103 117 108   0   0   0]
[ 80  97 107 104 114 105 110   0   0   0]
[ 84  97 110 115 104 111   0   0   0   0]
```

Implementation – Convert name to tensor

```
def make_tensors(names, countries):
    sequences_and_lengths = [name2list(name) for name in names]
    name_sequences = [sl[0] for sl in sequences_and_lengths]
    seq_lengths = torch.LongTensor([sl[1] for sl in sequences_and_lengths])
    countries = countries.long()

    # make tensor of name, BatchSize x SeqLen
    seq_tensor = torch.zeros(len(name_sequences), seq_lengths.max()).long()
    for idx, (seq, seq_len) in enumerate(zip(name_sequences, seq_lengths), 0):
        seq_tensor[idx, :seq_len] = torch.LongTensor(seq)

    # sort by length to use pack_padded_sequence
    seq_lengths, perm_idx = seq_lengths.sort(dim=0, descending=True)
    seq_tensor = seq_tensor[perm_idx]
    countries = countries[perm_idx]

    return create_tensor(seq_tensor), \
           create_tensor(seq_lengths),\
           create_tensor(countries)
```

Implementation – Convert name to tensor

After padding
[77 97 99 108 101 97 110   0   0   0]
[86 97 106 110 105 99 104 121   0   0]
[78 97 115 105 107 111 118 115 107 121]
[85 115 97 109 105   0   0   0   0   0]
[70 105 111 110 105 110   0   0   0   0]
[83 104 97 114 107 101 121   0   0   0]
[66 97 108 97 103 117 108   0   0   0]
[80 97 107 104 114 105 110   0   0   0]
[84 97 110 115 104 111   0   0   0   0]

After padding
[78 97 115 105 107 111 118 115 107 121]
[86 97 106 110 105 99 104 121   0   0]
[83 104 97 114 107 101 121   0   0   0]
[66 97 108 97 103 117 108   0   0   0]
[80 97 107 104 114 105 110   0   0   0]
[77 97 99 108 101 97 110   0   0   0]
[70 105 111 110 105 110   0   0   0   0]
[84 97 110 115 104 111   0   0   0   0]
[85 115 97 109 105   0   0   0   0   0]

```
    # sort by length to use pack_padded_sequence
    seq_lengths, perm_idx = seq_lengths.sort(dim=0, descending=True)
    seq_tensor = seq_tensor[perm_idx]
    countries = countries[perm_idx]

    return create_tensor(seq_tensor), \
           create_tensor(seq_lengths),\
           create_tensor(countries)
```

Implementation – Convert name to tensor

```
def make_tensors(names, countries):
    sequences_and_lengths = [name2list(name) for name in names]
    name_sequences = [sl[0] for sl in sequences_and_lengths]
    seq_lengths = torch.LongTensor([sl[1] for sl in sequences_and_lengths])
    countries = countries.long()

    # make tensor of name, BatchSize x SeqLen
    seq_tensor = torch.zeros(len(name_sequences), seq_lengths.max()).long()
    for idx, (seq, seq_len) in enumerate(zip(name_sequences, seq_lengths), 0):
        seq_tensor[idx, :seq_len] = torch.LongTensor(seq)

    # sort by length to use pack_padded_sequence
    seq_lengths, perm_idx = seq_lengths.sort(dim=0, descending=True)
    seq_tensor = seq_tensor[perm_idx]
    countries = countries[perm_idx]

    return create_tensor(seq_tensor), \
           create_tensor(seq_lengths),\
           create_tensor(countries)
```

```
def create_tensor(tensor):
    if USE_GPU:
        device = torch.device("cuda:0")
        tensor = tensor.to(device)
    return tensor
```

Implementation – One Epoch Training

```
def trainModel():
    total_loss = 0
    for i, (names, countries) in enumerate(trainloader, 1):
        inputs, seq_lengths, target = make_tensors(names, countries)
        output = classifier(inputs, seq_lengths)
        loss = criterion(output, target)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        total_loss += loss.it
        if i % 10 == 0:
            print(f'[{time_s
            print(f'[{i * le
            print(f'loss={to
    return total_loss
```

1. forward – compute output of model
2. forward – compute loss
3. zero grad
4. backward
5. update

Implementation – One Epoch Training

```
def trainModel():
    total_loss = 0
    for i, (names, countries) in enumerate(trainloader, 1):
        inputs, seq_lengths, target = make_tensors(names, countries)
        output = classifier(inputs, seq_lengths)
        loss = criterion(output, target)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        total_loss += loss.item()
        if i % 10 == 0:
            print(f'[{time_since(start)}] Epoch {epoch} ', end='')
            print(f'[{i * len(inputs)}/{len(trainset)}] ', end='')
            print(f'loss={total_loss / (i * len(inputs))}')
    return total_loss
```

Implementation – Testing

```
def testModel():
    correct = 0
    total = len(testset)
    print("evaluating trained model ...")
    with torch.no_grad():
        for i, (names, countries) in enumerate(testloader, 1):
            inputs, seq_lengths, target = make_tensors(names, countries)
            output = classifier(inputs, seq_lengths)
            pred = output.max(dim=1, keepdim=True)[1]
            correct += pred.eq(target.view_as(pred)).sum().item()

        percent = '%.2f' % (100 *
        print(f'Test set: Accura

    return correct / total
```

Tell PyTorch not to compute gradient of computational graph.

Saving time and memory!

Implementation – Testing

```
def testModel():
    correct = 0
    total = len(testset)
    print("evaluating trained model ...")
    with torch.no_grad():
        for i, (names, countries) in enumerate(testloader, 1):
            inputs, seq_lengths, target = make_tensors(names, countries)
            output = classifier(inputs, seq_lengths)
            pred = output.max(dim=
            correct += pred.eq(tar

        percent = '%.2f' % (100 * correct / total)
        print(f'Test set: Accuracy {correct}/{total} {percent}%')

    return correct / total
```

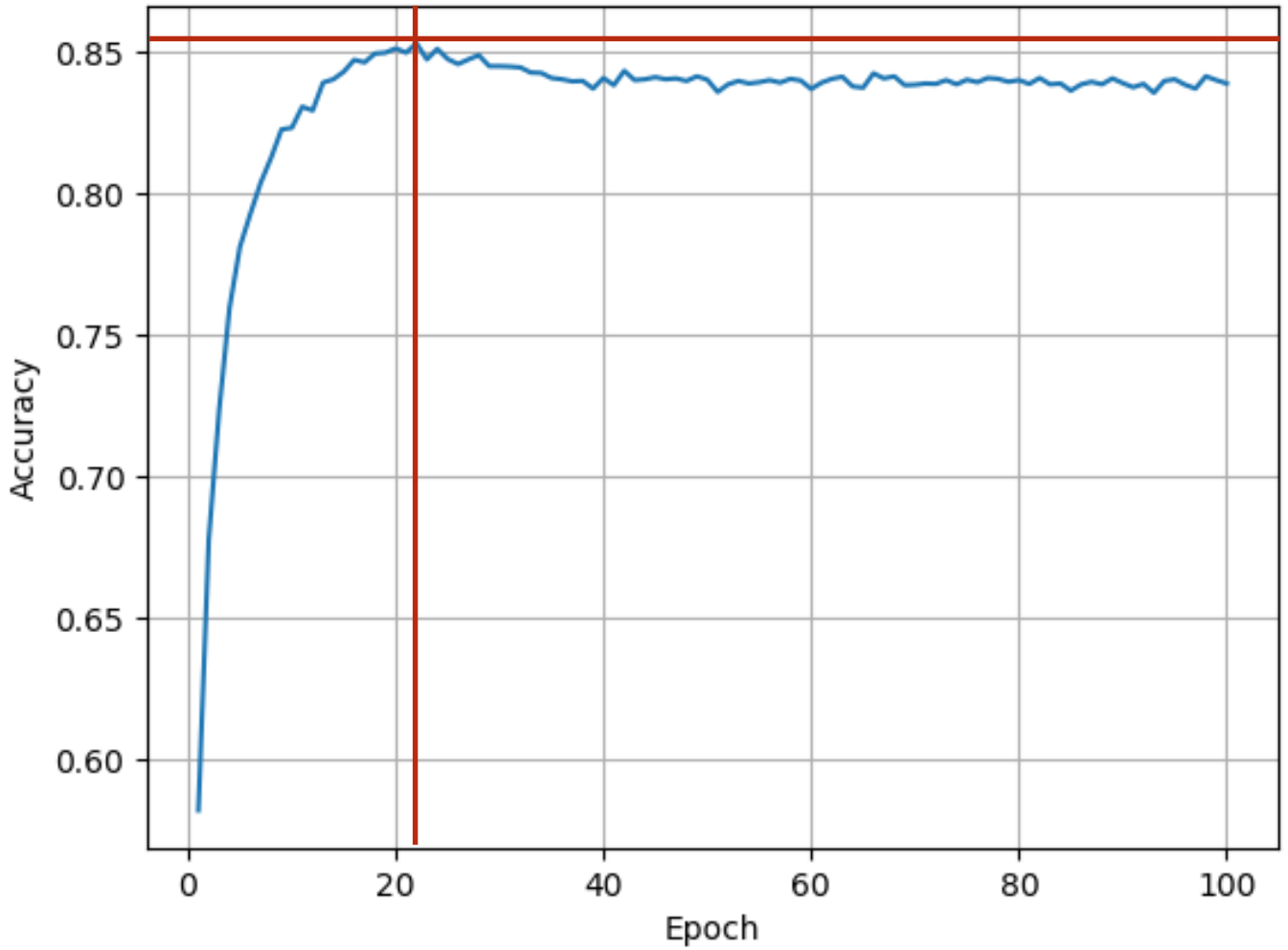
Compute the output of the model.

Implementation – Testing

```
def testModel():
    correct = 0
    total = len(testset)
    print("evaluating trained model ...")
    with torch.no_grad():
        for i, (names, countries) in enumerate(testloader, 1):
            inputs, seq_lengths, target = make_tensors(names, countries)
            output = classifier(inputs, seq_lengths)
            pred = output.max(dim=1, keepdim=True)[1]
            correct += pred.eq(target.view_as(pred)).sum().item()

        percent = '%.2f' % (100 *
        print(f'Test set: Accuracy {correct}/{total} {percent}%')

    return correct / total
```

Compute number of predicted correctly.

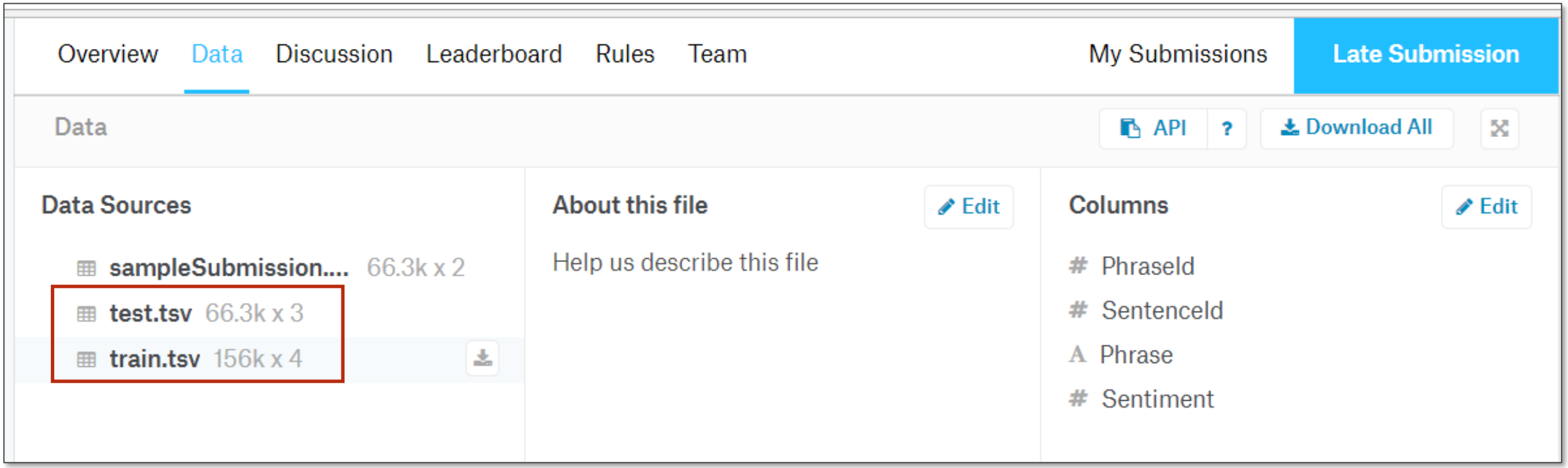
Implementation – Result



Exercise 13-1 Sentiment Analysis on Movie Reviews

- The Rotten Tomatoes movie review dataset is a corpus of movie reviews used for sentiment analysis.
  - dataset: https://www.kaggle.com/c/sentiment-analysis-on-movie-reviews/data
  - The dataset is comprised of tab-separated files with phrases from the Rotten Tomatoes dataset.
- The sentiment labels are:
  - 0 - negative
  - 1 - somewhat negative
  - 2 - neutral
  - 3 - somewhat positive
  - 4 - positive

Exercise 13-1 Sentiment Analysis on Movie Reviews



Exercise 13-1 Sentiment Analysis on Movie Reviews

PhraseId	SentenceId	Phrase	Sentiment
1	1	A series of escapades demonstrating the adage that what is good for the goose is also good for the gander , some of which occasionally amuses but none of which amounts to much of a story .	1
2	1	A series of escapades demonstrating the adage that what is good for the goose	2
3	1	A series	2
4	1	A	2
5	1	series	2
6	1	of escapades demonstrating the adage that what is good for the goose	2
7	1	of	2
8	1	escapades demonstrating the adage that what is good for the goose	2
9	1	escapades	2
10	1	demonstrating the adage that what is good for the goose	2

PyTorch Tutorial

13. RNN Classifier