1　再帰型ニューラルネットワーク

世の中には株価や売り上げ、気温変化、文章、音楽など時間によって変化する情報が多く存在し、このようなデータを時系列データと呼ぶ。機械学習において通常のニューラルネットワークでは時系列データを扱うのは困難である。時系列データを扱うためのニューラルネットワークとして再帰型ニューラルネットワーク（RNN=Recurrent neural network）が提案されている。

ＲＮＮは以下の図のように中間の隠れ層が繰り返し処理される（再帰される）構造を持っており、これによって時間的に変化する情報を扱うことが可能となっている。

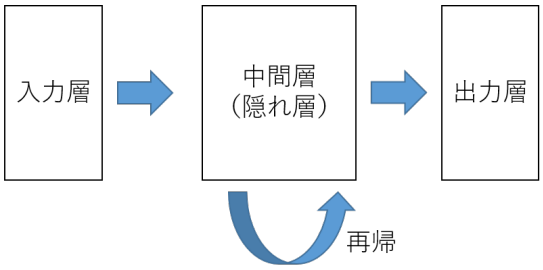


図　RNN の概要図

RNN では図のように時刻毎に順伝播および逆伝番が計算されるが、時間方向で中間層が連結されており、順伝播においては時刻 t における隠れ層の演算結果が時刻 t+1 の中間層に伝播され、時刻毎に出力が得られる。逆伝播においても時刻 t+1 のネットワークからの誤差が時刻 t に伝播し、勾配が求められることでパラメータが更新される。RNN は NN や CNN とは異なり、ある時刻のデータのみで勾配計算とパラメータ更新を行うのではなく、前後の時刻における演算結果を順伝播、逆伝播して前後の時刻の関係性を考慮して勾配計算とパラメータ更新を行うことで、時系列データの学習と推論を可能としている。しかしながら RNN は NN、CNN よりも勾配計算される回数が多くなるため、勾配消失の問題が発生しやすいという欠点を持っている点に注意が必要である。

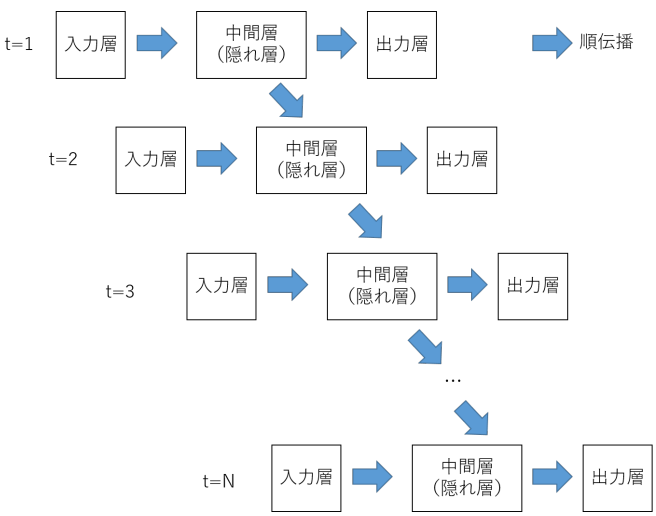


図　RNN における順伝播

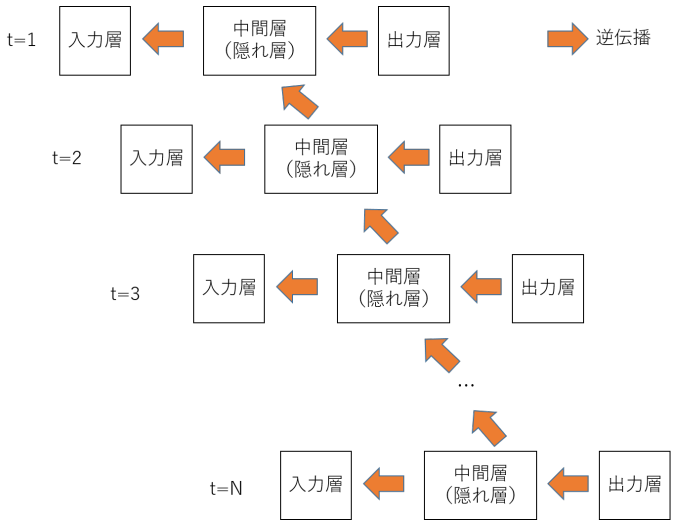


図　RNN における逆伝播

Pytorch では nn.Module に RNN の機能が実装されており、これを用いることで RNN によるネットワークを構築することができる。

```
import torch.nn as nn

class Net(nn.Module):
    def __init__(self):
        super().__init__()
        #RNN層の定義
        self.rnn = nn.RNN(input_size = 1, #入力データ数（1つの時系列データを扱うので1）
                          hidden_size = 64, #ニューロン数(ここでは例として64とする)
                          batch_first = True #入力形状をバッチサイズ,時刻数,入力数にする
                          )
        #全結合層の定義
        self.fc = nn.Linear(64,1) #(最終出力は1つの時系列データなので1)

    def forward(self,x):
        #戻り値は各時刻の出力y_rnnと隠れ層の演算結果h
        y_rnn, h = self.rnn(x, None)
        #-1を指定すると最後の時刻の結果を取得して結合できる
        y = self.fc(y_rnn[:, -1, :])

        return y

net = Net()
print(net)
```

ここでは具体例として以下の Sin 関数のような時系列データを学習させて、sin 関数の出力が得られるかＲＮＮを実装して理解することを図る。

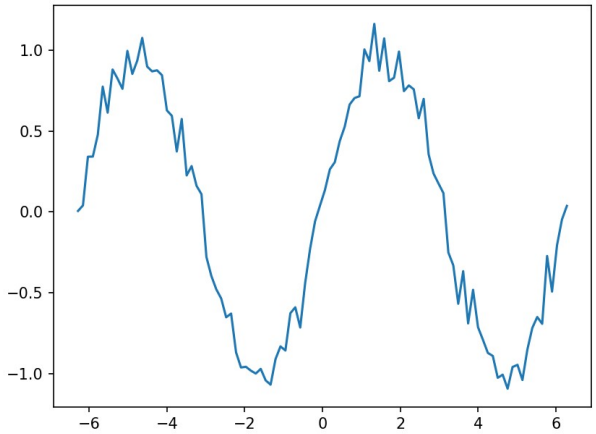
```
import torch
import math
import matplotlib.pyplot as plt

#-2πから2πまで100個の数値を作成する
x = torch.linspace(-2*math.pi, 2*math.pi, 100)

#ノイズを乗せてsinっぽいデータを作成する
y = torch.sin(x) + 0.1*torch.randn(len(x))

plt.plot(x,y)
plt.show()
```

このコードを実行すると以下のようなノイズが入った周期的な時系列データが得られる。



RNN ではいくつのデータの流れ（履歴）を学習データとして、その直後のデータを正解データとして学習することでデータの流れからその後のデータの予測を行うものである。
ここでは以下のような形でデータと正解データを用意して学習させることを考える。
各時刻のデータを 10 個ずつシーケンスとしてまとめて学習データとする。このとき次の時刻のデータが正解データとなる。この 10 個＋1 個のデータを 1 サンプルとして学習させる。

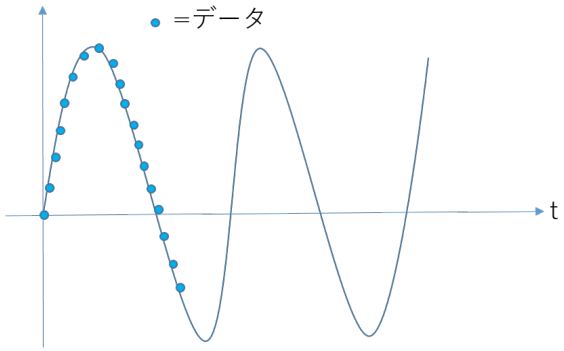


図　学習に用いる全データ

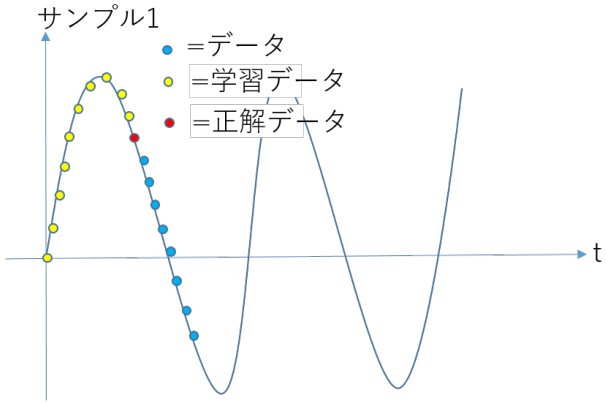


図　学習データ 1（サンプル 1）
t=0～9 が学習データ、t=10 が正解データ

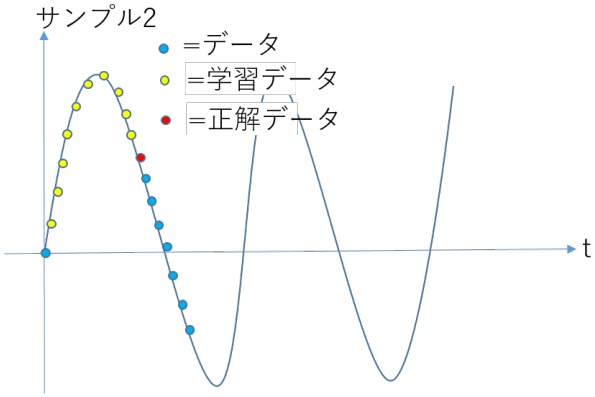


図　学習データ 2（サンプル 2）
t=1～10 が学習データ、t=11 が正解データ

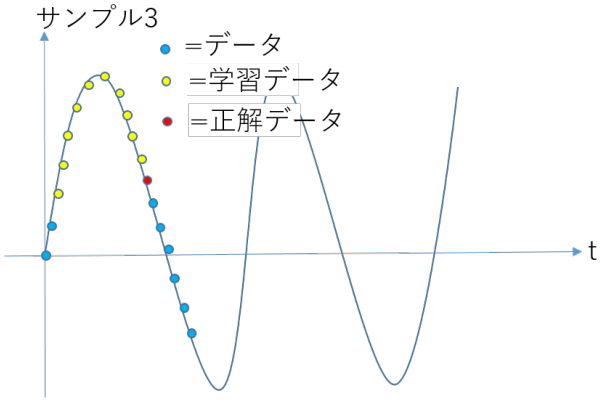


図　学習データ 3（サンプル 3）
t=2～11 が学習データ、t=12 が正解データ

上記のようなサンプルを残りのデータについても作成して用意する。

今回の例ではこの 10 個のデータの流れをまとめた複数のサンプルから、その次のデータがどのような形で現れるのかを学習することで少数のデータの流れからその後のデータの流れ（この例では sin 関数）を予測する。

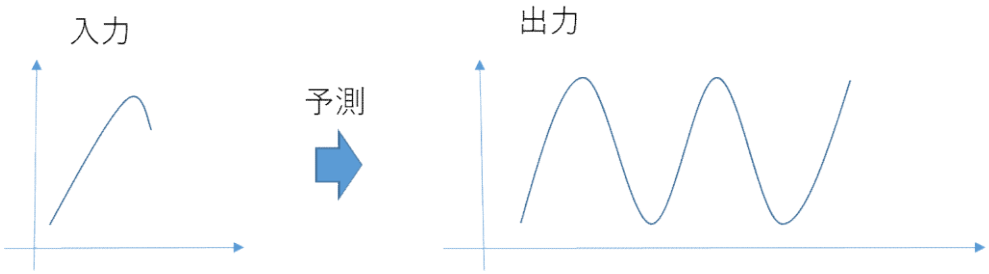


図　作成する学習モデルの概要

まず必要なモジュールをインポートしてノイズが入った sin 関数っぽいデータを生成する。

```
import torch
import math
import matplotlib.pyplot as plt

from torch.utils.data import TensorDataset, DataLoader
import torch.nn as nn
from torch import optim

#-2πから2πまで100個の数値を作成する
sinx = torch.linspace(-2*math.pi, 2*math.pi, 100)
print(len( sinx ))

#ノイズを乗せてsinっぽいデータを作成する
siny = torch.sin( sinx ) + 0.1*torch.randn(len( sinx ))
```

次に RNN の学習データを作成するためにシーケンス数とサンプル数を定義する。

先に述べたように今回は 10 個のデータをシーケンスとしてまとめてこれを 1 つのサンプルとする。

また学習データと正解データを格納する変数（tensor）を作成し、これを 0 で初期化しておく。

```
#シーケンス数 100個の時系列データをt=n+0からt=n+9(n=0~90)まで
#10個分まとめて入力とする
n_time = 10

#学習データはt=0,1,2,3,4,5,6,7,8,9のデータ → 正解はt=10のデータ
#次の学習データはt=1,2,3,4,5,6,7,8,9,10のデータ → 正解はt=11のデータ
#次の学習データはt=2,3,4,5,6,7,8,9,10,11のデータ → 正解はt=12のデータ
#...と上記の組(t=n+0からt=n+9のデータのまとまり)が90組あることになる
#nの値は0から90なので、全データからシーケンス数を引くと組数となる
#sample数（学習させるデータ数）
n_sample = len( sinx ) - n_time

#入力データを格納する変数の定義（sample数、シーケンス数、入力データ数）
input_data = torch.zeros((n_sample, n_time, 1))

#正解データを格納する変数の定義(sample数,正解は1つの値なので次元数は1)
correct_data = torch.zeros((n_sample,1))
```

生成した sin っぽいデータから先に述べたように 10 個ずつデータを t=0 から順番に取り出して入力データを作成し、このサンプルの次の時刻のデータを正解データとして 1 つ取り出す。この操作をサンプル数の総数分繰り返し、tensor の形に変換する。
最後に tensor を Dataloader を用いてミニバッチ学習できる形にする。

```
#サンプル数分の入力データと正解データをつくる
for i in range(n_sample):
    #t=n+0からt=n+9までの値をスライスを用いてyから取り出して入力データを作成
    #view(-1,1)を用いてデータの構造を調整(シーケンス数,入力データ数=1)
    input_data[i] = siny[i:i+n_time].view(-1,1)

    #正解データの作成、入力データの次の時刻のデータ
    correct_data[i] = siny[i+n_time:i+n_time+1]

#入力データと正解データのペアをTensorの形にする
dataset = TensorDataset(input_data, correct_data)

#データセットの作成
train_loader = DataLoader(dataset,batch_size=8,shuffle = True)
```

データの準備が出来たら RNN のネットワークを構築する。
先の例と同じものを流用して用いることとする。

```
#RNNの構築
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        #RNN層の定義
        self.rnn = nn.RNN(input_size = 1, #入力データ数 (1つの時系列データを扱うので1)
                          hidden_size = 64, #ニューロン数(ここでは例として64とする)
                          batch_first = True #入力形状をバッチサイズ,シーケンス数,入力数にする
                          )
        #全結合層の定義
        self.fc = nn.Linear(64,1) #(最終出力は1つの時系列データなので1)

    def forward(self,x):
        #戻り値は各時刻の出力y_rnnと隠れ層の演算結果h
        y_rnn, h = self.rnn(x, None)
        #-1を指定すると最後の時刻の結果を取得して結合できる
        y = self.fc(y_rnn[:, -1, :])

        return y

#RNNのインスタンス
net = Net()
```

RNN の構築が出来たら次に損失関数と最適化アルゴリズムを設定する。今回は回帰問題であるので、損失関数として 2 乗誤差関数を用い、最適化アルゴリズムは SGD を用いることとする。
併せてログ用の空のリストを定義しておく。

```
#損失関数の定義 回帰問題なので2乗誤差関数を設定
loss_func = nn.MSELoss()

#最適化アルゴリズム
optimizer = optim.SGD(net.parameters(), lr = 0.01)

#損失関数のログ保存用
record_loss_train = []
```

データ、ネットワーク、損失関数、最適化アルゴリズムの準備ができたので、学習を行う。
今回はエポック数は 100、バッチサイズは 8 とする。
エポック 10 回毎に損失関数の値と予測したグラフを画面に表示して確認できるようにしておく。

```
#学習　エポック数100
epochs = 100
for i in range(epochs):
    #学習モード
    net.train()
    loss_train = 0
    #ミニバッチ学習
    for j, (x,t) in enumerate(train_loader):
        y = net(x)
        loss = loss_func(y, t)
        loss_train += loss.item()
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
    loss_train /= j+1
    record_loss_train.append(loss_train)

    #学習過程の表示
    if i % 10 == 0 or i == epochs - 1:
        #評価モード
        net.eval()
        print("Epochs:",i, "Loss Train:",loss_train)
        predicted = list(input_data[0].view(-1))#初期値
        for i in range(n_sample):
            #直近の時系列データを取り出す
            x = torch.tensor(predicted[-n_time : ])
            #バッチサイズ,シーケンス数,入力数
            x = x.view(1, n_time, 1)
            #順伝播で予測して結果をリストに格納
            y = net(x)
            predicted.append(y[0].item())
        #学習データを表示
        plt.plot(range(len( sinx )),siny,label="Correct")
        #学習した後の予測データを表示
        plt.plot(range(len(predicted)),predicted,label="Predicted")
        plt.legend()
        plt.show()
```

実行すると以下のようにエポック 10 回毎の損失関数の値の履歴が表示される。

```
========== RESTART: C:/Users/owner/D
100
Epochs: 0 Loss Train: 0.45989555687022704
Epochs: 10 Loss Train: 0.046805075059334435
Epochs: 20 Loss Train: 0.033231744542717934
Epochs: 30 Loss Train: 0.023329909852085013
Epochs: 40 Loss Train: 0.021125627448782325
Epochs: 50 Loss Train: 0.022740245137053233
Epochs: 60 Loss Train: 0.019636538345366716
Epochs: 70 Loss Train: 0.019724449259229004
Epochs: 80 Loss Train: 0.019796583297041554
Epochs: 90 Loss Train: 0.01901110797189176
Epochs: 99 Loss Train: 0.018887387743840616
```

またエポック 10 回毎に予測したグラフが表示される。
エポック数が少ないうちは全く見当違いのグラフを生成しているが、学習が進むにつれて段々sin 関数に近い形状をとるような予測値が得られていることが見て取れる。

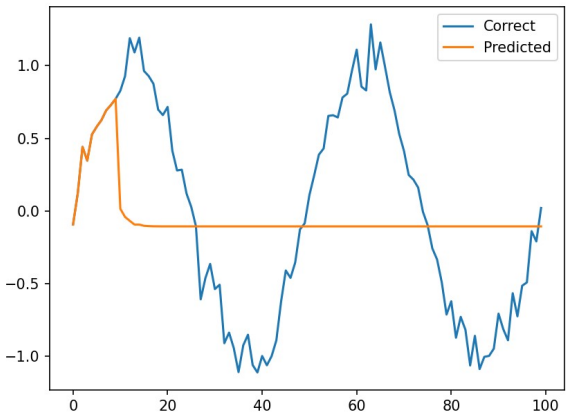


図　エポック 10

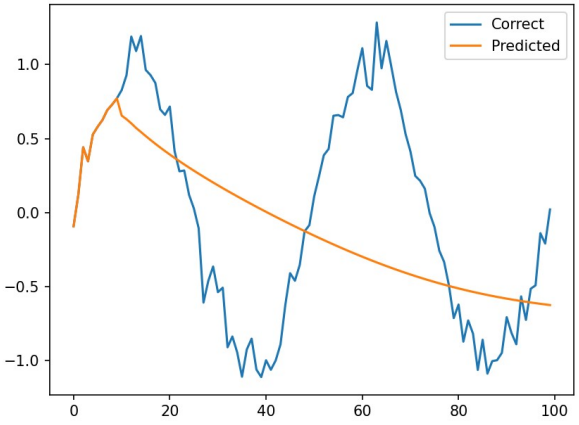


図　エポック 20

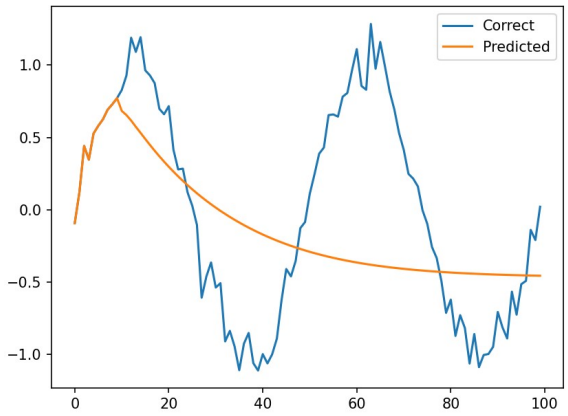


図　エポック 30

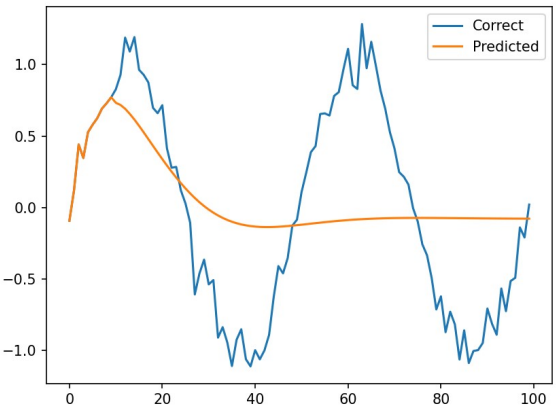


図　エポック 40

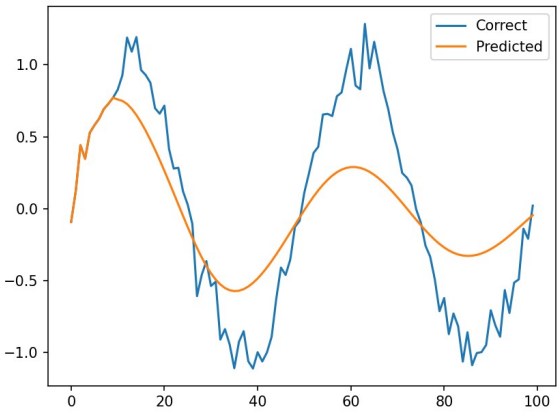


図　エポック 50

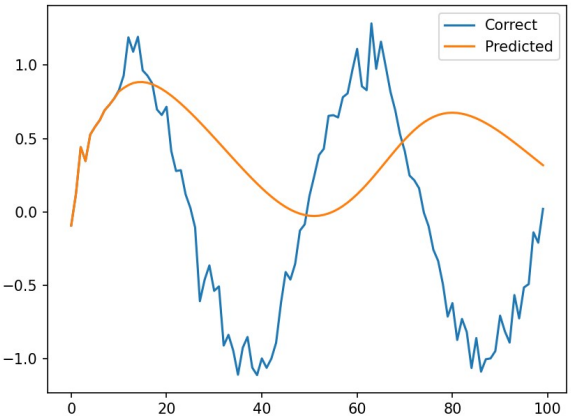


図　エポック 60

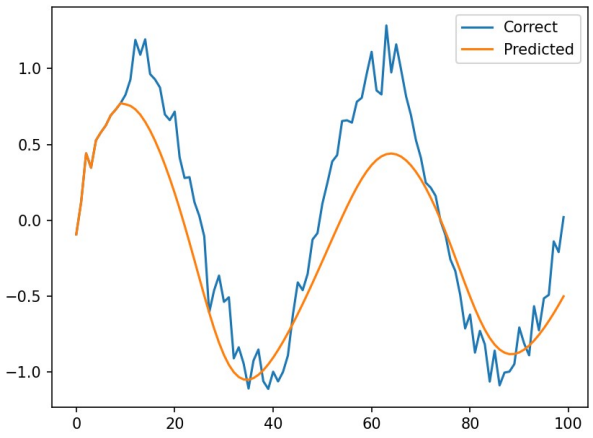


図　エポック 70

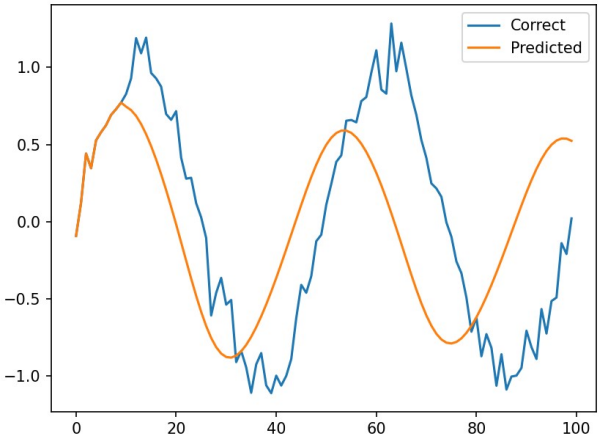


図　エポック 80

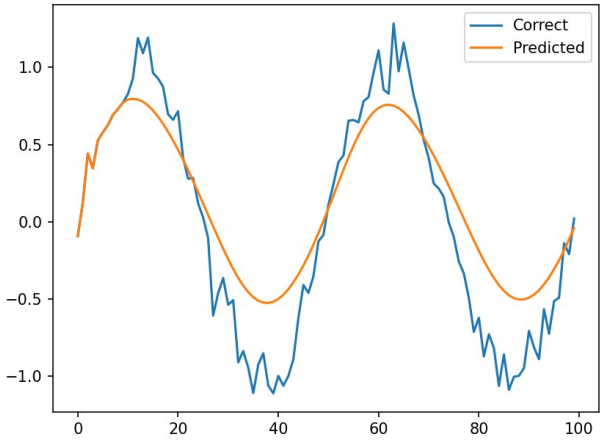


図　エポック 90

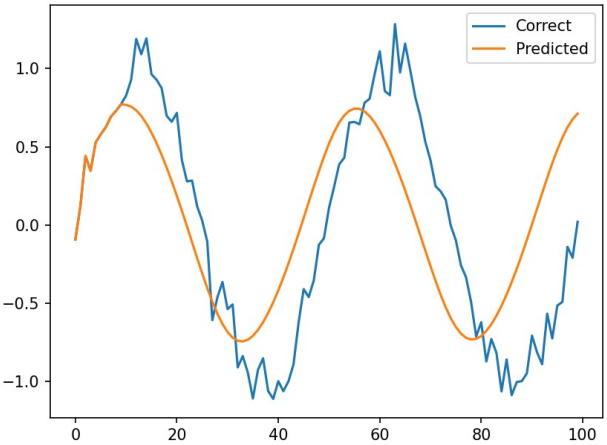


図　エポック 100

2　LSTM

RNN は NN や CNN とは異なり時系列データを扱うことができるが、先に述べたように勾配消失の問題が発生しやすいという欠点がある。また時系列データとして長期間のデータを扱うことが難しいという欠点もある。これを改良するネットワークとして LSTM (Long Short Term Memory) が提案されている。隠れ層にあたる部分に LTSM 層と呼ばれる層が設けられており、LSTM 層が繰り返し処理されることで時系列データの学習を可能としている。

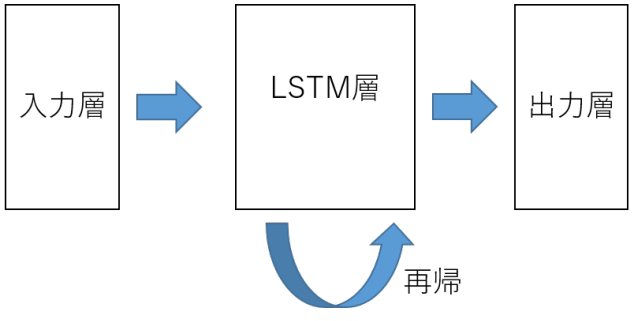


図　LSTM によるネットワーク概要図

LSTM 層は内部に「ゲート」、「記憶セル」と呼ばれる仕組みが備わっており、過去の情報を「忘れるか忘れないか」を判断しながら必要な情報のみを次に時刻に引き継ぐことで勾配消失や長期にわたる時系列データの処理が可能となっている。LSTM 層の内部は下図のように出力ゲート、忘却ゲート、入力ゲート、記憶セルを組み合わせた構造になっている。

この図における実線は現在のデータの流れを表しており、点線は 1 つ前の時刻のデータの流れを表している。x_t がこの時刻における層への入力データであり、h_t がこの時刻における LSTM 層の出力、h_{t-1} が 1 つ前の時刻における LSTM 層の出力である。

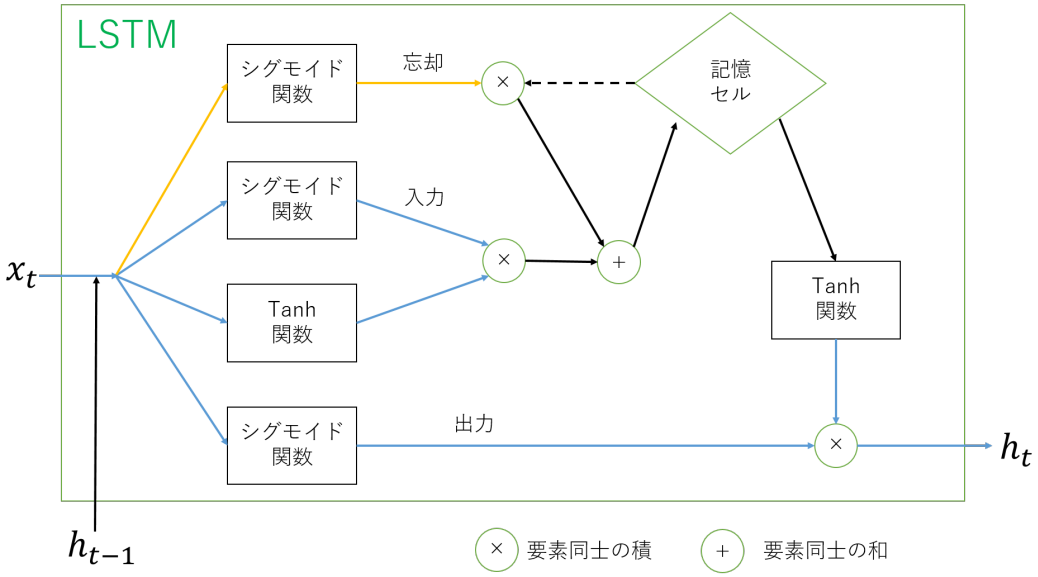


図　ＬＳTM層の内部構造

以下、各ゲートの仕組みを概説する。

出力ゲートは入力と前の時刻の出力 h_{t-1} にそれぞれ重みをかけたうえで合流させてシグモイド関数に入れる。出力ゲートを経たデータは記憶セルから来たデータと要素ごとの積が計算され、記憶セルに保持された過去のデータをどの程度層の出力に反映させるか調整する役割を持っている。

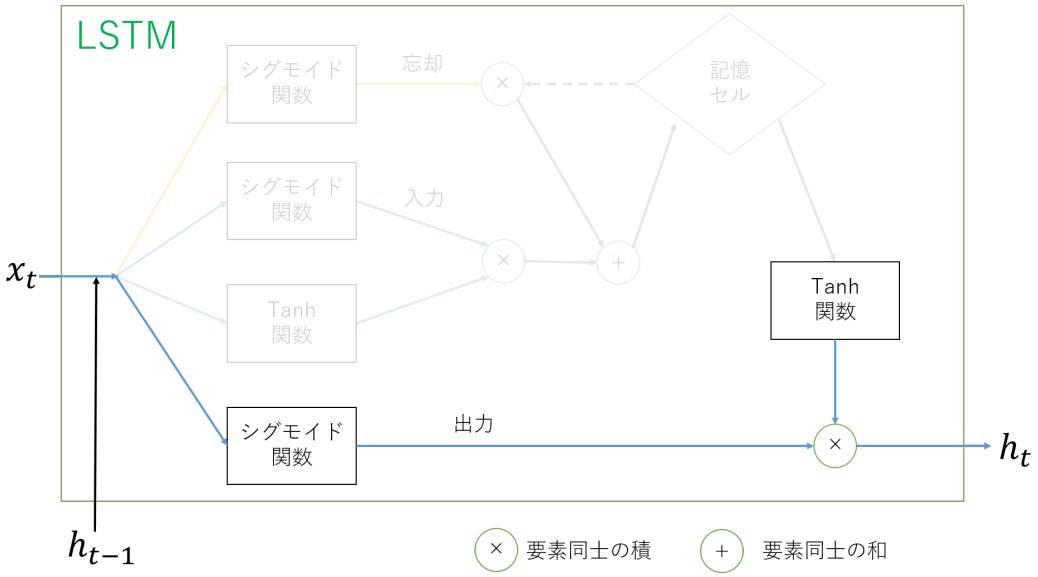


図　出力ゲートの概要

忘却ゲートは入力と前の時刻の出力 h_{t-1} にそれぞれ重みをかけたうえで合流させてシグモイド関数に入れる。忘却ゲートを経たデータは記憶セル保持されていた過去のデータと要素ごとに積がとられ、過去の記憶をどの程度残すのか調整する。シグモイド関数の出力は 0 から 1 の値となるため、これを記憶せるからのデータにかけることで 0 に近ければ忘れ、 1 に近い値なら記憶を保持することができる。

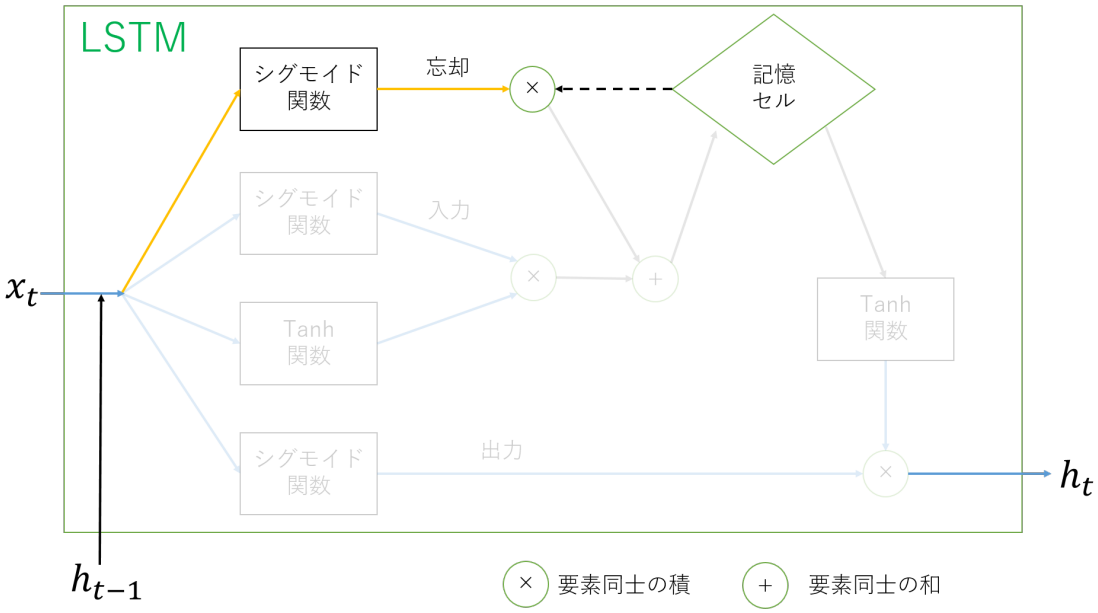


図　忘却ゲートの概要

入力ゲートは入力と前の時刻の出力 h_{t-1} にそれぞれ重みをかけたうえで合流させてシグモイド関数と Tanh 関数に入れ、それぞれの関数の出力に対して要素ごとに積を計算する。シグモイド関数の出力は 0 から 1 の値をとり、Tanh 関数を経たデータをどの程度記憶セルに保持させるかを調整する。

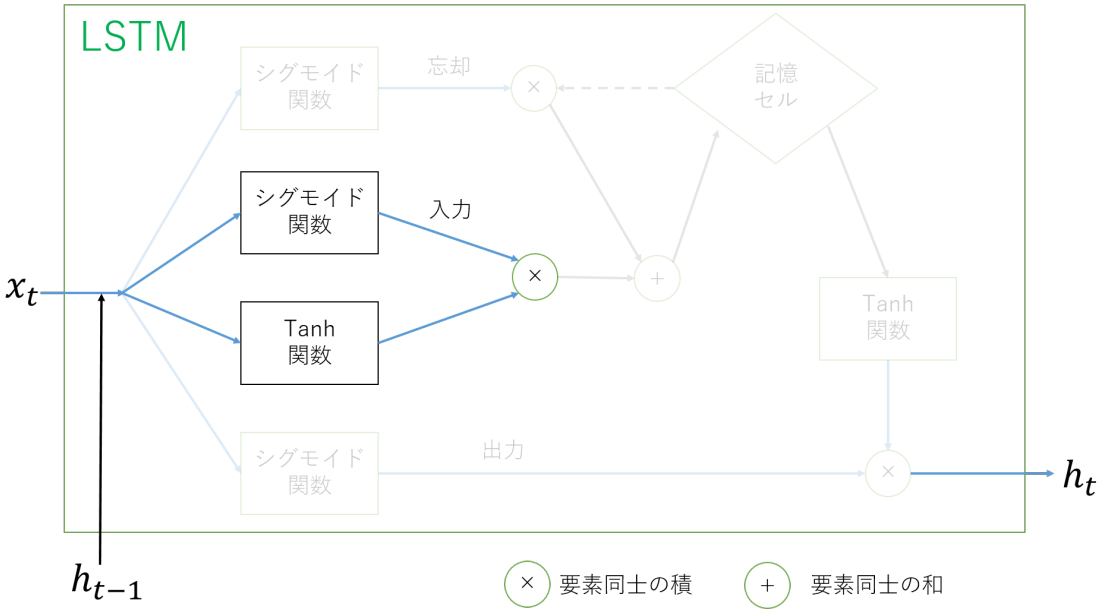


図　入力ゲートの概要

記憶セルは忘却ゲートからのデータと入力ゲートからのデータを要素ごとに足し合わせて、新たな記憶として保持する。これにより LSTM は長期的な記憶の保持が可能となり RNN では難しい長期的な時系列データの処理が可能となっている。記憶セルの情報は毎回、出力ゲートに Tanh 関数を通じて渡され、要素ごとに掛け合わされたものが LSTM 層の出力とされる。

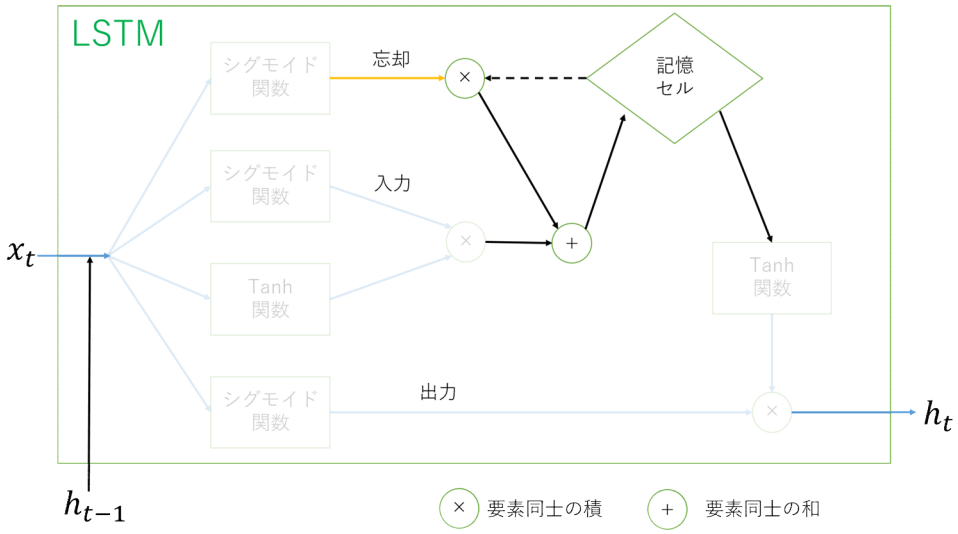


図　記憶セルの概要

Pytorch では nn.Module に LSTM の機能が実装されており、これを用いることで LSTM によるネットワークを構築することができる。

```
import torch.nn as nn

class Net(nn.Module):
    def __init__(self):
        super().__init__()
        #LTSM層の定義
        self.lstm = nn.LSTM(input_size = 28, #入力次元数（ここでは例として28）
                            hidden_size = 256, #ニューロン数（ここでは例として256）
                            batch_first = True #入力をバッチサイズ、シーケンス数、入力数とする
                            )
        #全結合層の定義
        self.fc = nn.Linear(256,28) #入力数256,出力数28 一例

    def forward(self,x):
        #戻り値は各時刻の出力y_lstmと層の出力hおよび記憶セルc
        y_lstm, (h, c) = self.lstm(x,None)
        #全結合層では最後の時刻のデータを-1で指定して入力
        y = self.fc(y_lstm[: , -1, :])
        return y

#インスタンス
net = Net()
print(net)
```

実行すると以下のように LSTM の層が構築できていることが分かる。

```
==================== RESTART: C:/Users/owner/Desktop/ccc.py
Net(
  (lstm): LSTM(28, 256, batch_first=True)
  (fc): Linear(in_features=256, out_features=28, bias=True)
)
```

先ほどの sin 関数を予測するコードにおいて、以下のように RNN の部分を変更してエポック数も 1000 回に変更して実行してみる。

ネットワークの変更

```
#RNNの構築
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        #RNN層の定義
        self.rnn = nn.LSTM(input_size = 1, #入力データ数（1つの時系列データを扱うので1）
                           hidden_size = 64, #ニューロン数(ここでは例として64とする)
                           batch_first = True #入力形状をバッチサイズ,シーケンス数,入力数にする
                           )
        #全結合層の定義
        self.fc = nn.Linear(64,1) #(最終出力は1つの時系列データなので1)

    def forward(self,x):
        #戻り値は各時刻の出力y_rnnと隠れ層の演算結果h
        y_rnn, (h,c)= self.rnn(x, None)
        #-1を指定すると最後の時刻の結果を取得して結合できる
        y = self.fc(y_rnn[:, -1, :])

        return y
```

学習回数および結果表示の間隔変更

```
#学習　エポック数100
epochs = 1000
for i in range(epochs):
    #学習モード
    net.train()
    loss_train = 0
    #ミニバッチ学習
    for j, (x,t) in enumerate(train_loader):
        y = net(x)
        loss = loss_func(y, t)
        loss_train += loss.item()
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
    loss_train /= j+1
    record_loss_train.append(loss_train)

    #学習過程の表示
    if i % 100 == 0 or i == epochs - 1:
        #評価モード
        net.eval()
        print("Epochs:",i, "Loss Train:",loss_train)
        predicted = list(input_data[0].view(-1))#初期値
        for i in range(n_sample):
            #直近の時系列データを取り出す
            x = torch.tensor(predicted[-n_time : ])
            #バッチサイズ,シーケンス数,入力数
            x = x.view(1, n_time, 1)
            #順伝播で予測して結果をリストに格納
            y = net(x)
            predicted.append(y[0].item())
        #学習データを表示
        plt.plot(range(len( sinx )),siny,label="Correct")
        #学習した後の予測データを表示
        plt.plot(range(len(predicted)),predicted,label="Predicted")
        plt.legend()
        plt.show()
```

RNN のときよりも時間はかかるが RNN の場合よりも一部、元のデータを表現できていることが分かる。

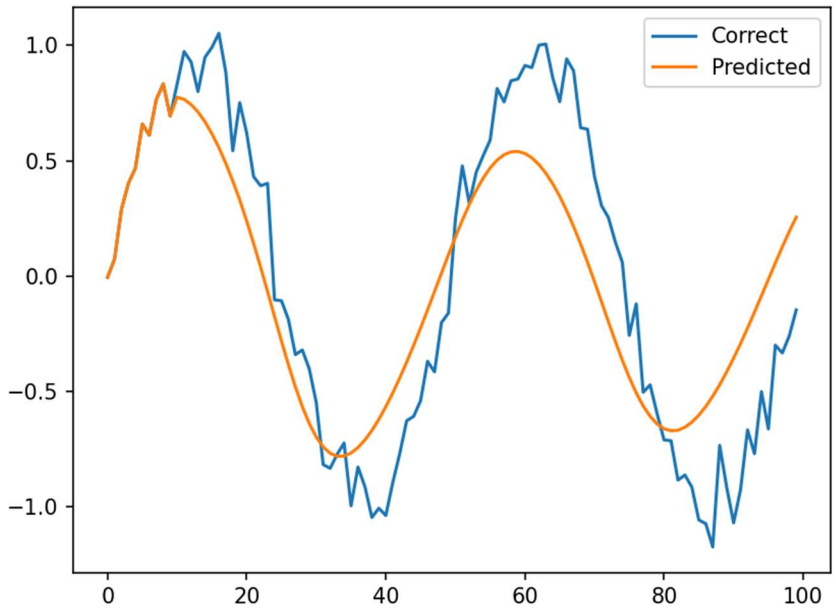


図　エポック 999

RNN は時系列データから未来のデータの出現パターンの予測（生成）が可能なネットワークであるため、これを応用すると文章の生成や画像の生成、音声の生成などが可能であり、昨今活用が増大している生成 AI のもととなっている技術である。