Simulasi Model Hybrid Deep Learning pada Sistem Rekomendasi Cerdas

Tim Pencipta:

Kadek Cahya Dewi, S.T, M.Cs / 0009098408

Putu Adriani Prayustika, S.E., M.M / 0001048404

Putu Indah Ciptayani, S.Kom, M.Cs. / 0013048502

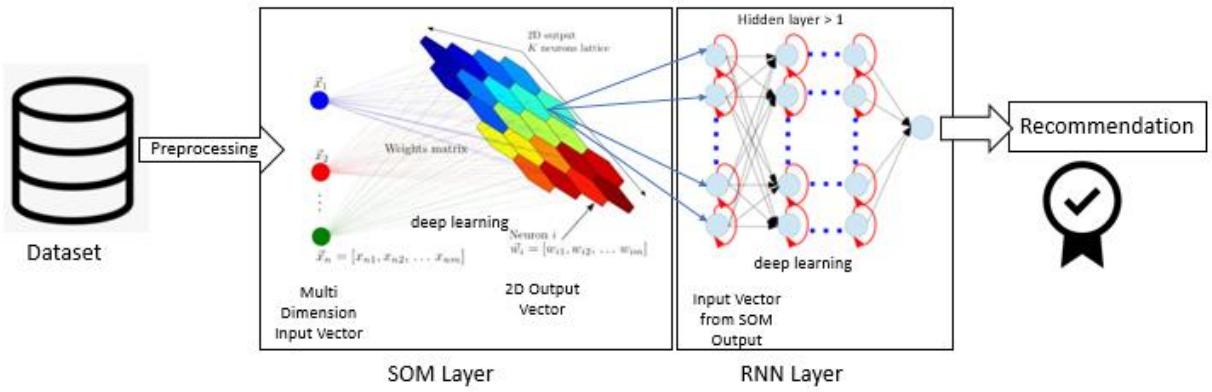
SEPTEMBER

2021

Deskripsi dan Spesifikasi Produk Simulasi Model Hybrid Deep Learning pada Sistem Rekomendasi Cerdas

A. Deskripsi Model

Trend perkembangan teknologi saat ini adalah mengarah ke sistem cerdas. Namun saat ini belum ada yang menggabungkan dua metode deep learning pada algoritma rekomendasi, sehingga penting untuk melakukan pemodelan sistem rekomendasi cerdas menggunakan *hybrid deep learning*. Produk ini merupakan simulasi dari model hybrid deep learning pada sistem rekomendasi cerdas. *Hybrid deep learning* yang dimaksud adalah gabungan dari metode Clustering Self Organizing Maps (SOM) dan metode *Recurrent Neural Network* (RNN). Gambar 1 merupakan gambar model hybrid deep learning pada sistem rekomendasi cerdas.



Gambar 1. Model hybrid deep learning pada sistem rekomendasi cerdas.

Berdasarkan hasil SLR dataset dari sebuah sistem rekomendasi cerdas dapat berbasis text, image, audio, video atau kombinasinya. Preprocessing dilakukan terlebih dahulu sebelum masuk ke layer hybrid deep learning. Hybrid deep learning terdiri dari dari dua layer yaitu layer Self Organing Map (SOM) dan layer Recurrent Neural Network (RNN). SOM layer akan memetakan multidimensional input vector kedalam 2D output vector. Kemudian SOM output vector tersebut akan menjadi input vector bagi RNN layer. RNN layer kemudian menghasilkan output sebagai rekomendasi sesuai dengan hasil deep learning pada RNN layer.

B. Print Out Program

Program simulasi dibuat menggunakan bahasa pemrograman Python. Beberapa modul pada python yang digunakan adalah modul numpy, pandas, matplotlib, tensorflow, keras, torch, sklearn.

```
#hybrid deep learning model modify SOM and RNN method
# ================================================================

#step 1 import libraries
import math
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import torch
import torch.nn as nn
from torch.autograd import Variable
from sklearn.model_selection import train_test_split
from torch.utils.data import DataLoader, TensorDataset

def closest_node(data, t, map, m_rows, m_cols):
  # (row,col) of map node closest to data[t]
  result = (0,0)
  small_dist = 1.0e20
  for i in range(m_rows):
    for j in range(m_cols):
      ed = euc_dist(map[i][j], data[t])
      if ed < small_dist:
        small_dist = ed
        result = (i, j)
  return result

def euc_dist(v1, v2):
  return np.linalg.norm(v1 - v2)

def manhattan_dist(r1, c1, r2, c2):
  return np.abs(r1-r2) + np.abs(c1-c2)

def most_common(lst, n):
  # lst is a list of values 0 . . n
  if len(lst) == 0: return 10
  counts = np.zeros(shape=n, dtype=np.int)
  for i in range(len(lst)):
    counts[lst[i]] += 1
  return np.argmax(counts)

def main():
  # Step 2 Prepare Dataset
  # load data

  train = pd.read_csv("SOM_RNN_train.csv", dtype = np.float32)
```

```
# split data into features(pixels) and labels(numbers from 0 to 9)
targets_numpy = train.label.values
features_numpy = train.loc[:,train.columns != "label"].values/255 # normalization

print("features numpy",features_numpy.shape)
print("target numpy",targets_numpy.shape)


# step 3 SOM Model
np.random.seed(1)
Dim = 784
Rows = 28; Cols = 28
RangeMax = Rows + Cols
LearnMax = 0.5 # learning rate
StepsMax = 5000 # epoch #5000 forbest precision
quantErr = 0.0 #qe

print("Constructing SOM")
# make SOM map
#map = np.random.random_sample(size=(Rows,Cols,Dim))
map = np.random.randint(999,size=(Rows,Cols,Dim))

# SOM training
for s in range(StepsMax):
  if s % (StepsMax/10) == 0: print("step = ", str(s))
  pct_left = 1.0 - ((s * 1.0) / StepsMax)
  curr_range = (int)(pct_left * RangeMax)
  curr_rate = pct_left * LearnMax

  t = np.random.randint(len(features_numpy))
  # find winner
  (bmu_row, bmu_col) = closest_node(features_numpy, t, map, Rows, Cols)
  # update map
  for i in range(Rows):
    for j in range(Cols):
      if manhattan_dist(bmu_row, bmu_col, i, j) < curr_range:
        map[i][j] = map[i][j] + curr_rate * \
(features_numpy[t] - map[i][j])
print("map shape:",map.shape)
print("SOM construction complete \n")
trans_map = map.transpose(2,0,1).reshape(784,784)
featuresRNN = pd.DataFrame.from_records(trans_map)
featuresRNN.to_csv(r'export_trans_features.csv', index = False, header=True)

print("mapping SOM label start")
```

```
# associate each data label with a map node
print("Associating each data label to one map node ")
mapping = np.empty(shape=(Rows,Cols), dtype=object)
for i in range(Rows):
  for j in range(Cols):
    mapping[i][j] = []

for t in range(len(features_numpy)):
  (m_row, m_col) = closest_node(features_numpy, t, map, Rows, Cols)
  mapping[m_row][m_col].append(math.floor(targets_numpy[t]))

label_map = np.zeros(shape=(Rows,Cols), dtype=np.int)
for i in range(Rows):
  for j in range(Cols):
    label_map[i][j] = most_common(mapping[i][j], 784)

print("label map shape:",label_map.shape)
trans_label = label_map.reshape([784,1])
targetsRNN = pd.DataFrame.from_records(trans_label)
targetsRNN.to_csv(r'export_trans_labels.csv', index = False, header=True)
print("SOM mapping complete \n")

#SOM end

# prepare for RNN create dataframe
print("RNN start.... \n")
features = pd.read_csv("export_trans_features.csv", dtype = np.float32)
targets = pd.read_csv("export_trans_labels.csv", dtype = np.float32)

# prepare data into features(pixels) and labels(numbers from 0 to 9)
targets_numpy = targets.to_numpy()
targets_numpy = targets_numpy.reshape([784])
features_numpy = features.loc[:,:].values/255 # normalization features.to_numpy()

print("features numpy",features_numpy.shape)
print("target numpy",targets_numpy.shape)


# train test split. Size of train data is 80% and size of test data is 20%.
features_train,        features_test,        targets_train,        targets_test        =
train_test_split(features_numpy,
                                                    targets_numpy,
                                                    test_size = 0.2,
                                                    random_state = 42)
```

```
  # create feature and targets tensor for train set. As you remember we need variable to accumulate gradients. Therefore first we create tensor, then we will create variable
  featuresTrain = torch.from_numpy(features_train).type(torch.LongTensor)
  targetsTrain = torch.from_numpy(targets_train).type(torch.LongTensor) # data type is long

  print("features train",featuresTrain.shape)
  print("target train",targetsTrain.shape)

  # create feature and targets tensor for test set.
  featuresTest = torch.from_numpy(features_test)
  targetsTest = torch.from_numpy(targets_test).type(torch.LongTensor) # data type is long

  print("features test",featuresTest.shape)
  print("target test",targetsTest.shape)

  # batch_size, epoch and iteration
  batch_size = 100
  n_iters = 8000
  num_epochs = n_iters / (len(features_train) / batch_size)
  num_epochs = int(num_epochs)

  # Pytorch train and test sets
  train = TensorDataset(featuresTrain,targetsTrain)
  test = TensorDataset(featuresTest,targetsTest)

  # data loader
  train_loader = DataLoader(train, batch_size = batch_size, shuffle = False)
  test_loader = DataLoader(test, batch_size = batch_size, shuffle = False)

  # Step 4 Create RNN
  # Create RNN Model
  class RNNModel(nn.Module):
    def __init__(self, input_dim, hidden_dim, layer_dim, output_dim):
        super(RNNModel, self).__init__()

        # Number of hidden dimensions
        self.hidden_dim = hidden_dim

        # Number of hidden layers
        self.layer_dim = layer_dim

        # RNN
```

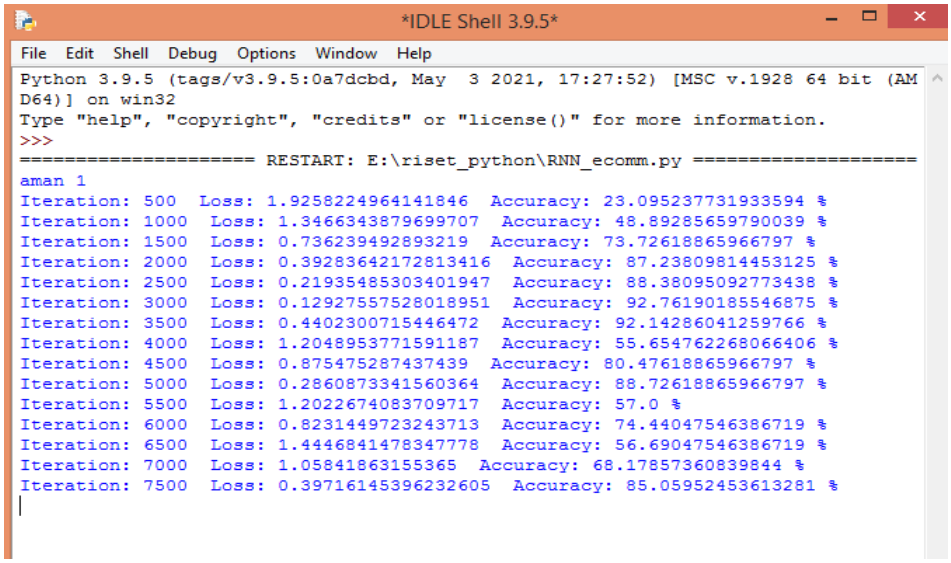
```
        self.rnn = nn.RNN(input_dim, hidden_dim, layer_dim, batch_first=True,
nonlinearity='relu')

        # Readout layer
        self.fc = nn.Linear(hidden_dim, output_dim)

    def forward(self, x):

        # Initialize hidden state with zeros
        h0 = Variable(torch.zeros(self.layer_dim, x.size(0), self.hidden_dim))

        # One time step
        out, hn = self.rnn(x, h0)
        out = self.fc(out[:, -1, :])
        return out

input_dim = 28    # input dimension
hidden_dim = 100  # hidden layer dimension
layer_dim = 1     # number of hidden layers
output_dim = 11   # output dimension

# Step 5 Instantiate model
model = RNNModel(input_dim, hidden_dim, layer_dim, output_dim)


# Step 6 Instantiate Loss
# Cross Entropy Loss
error = nn.CrossEntropyLoss()

# Step 7 Instantiate Optimizer
# SGD Optimizer
learning_rate = 0.05
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)

# Step 8 training

seq_dim = 28
loss_list = []
iteration_list = []
accuracy_list = []
count = 0
for epoch in range(num_epochs):
    for i, (images, labels) in enumerate(train_loader):
        train  = Variable(images.view(-1, seq_dim, input_dim))
        labels = Variable(labels)
```

```
# Clear gradients
optimizer.zero_grad()

# Forward propagation
outputs = model(train.float())

#print("output ke-",i,":",outputs)
#print("labels forward ke-",i,":",labels)

# Calculate softmax and cross entropy loss
loss = error(outputs, labels)

# Calculating gradients
loss.backward()

# Update parameters
optimizer.step()

count += 1

if count % 250 == 0:
    # Calculate Accuracy
    correct = 0
    total = 0
    # Iterate through test dataset
    # Step 9 Prediction
    for images, labels in test_loader:
        images = Variable(images.view(-1, seq_dim, input_dim))

        # Forward propagation
        outputs = model(images.float())

        # Get predictions from the maximum value
        predicted = torch.max(outputs.data, 1)[1]

        # Total number of labels
        total += labels.size(0)

        correct += (predicted == labels).sum()

    accuracy = 100 * correct / float(total)

    # store loss and iteration
    loss_list.append(loss.data)
```

```
            iteration_list.append(count)
            accuracy_list.append(accuracy)
            if count % 500 == 0:
                # Print Loss
                print('Iteration: {}    Loss: {}    Accuracy: {} %'.format(count, loss.item(),
accuracy))

  # visualization loss
  plt.plot(iteration_list,loss_list)
  plt.xlabel("Number of iteration")
  plt.ylabel("Loss")
  plt.title("SOM-RNN: Loss vs Number of iteration")
  plt.show()

  # visualization accuracy
  plt.plot(iteration_list,accuracy_list,color = "red")
  plt.xlabel("Number of iteration")
  plt.ylabel("Accuracy")
  plt.title("SOM-RNN: Accuracy vs Number of iteration")
  plt.savefig('graph.png')
  plt.show()

# =================================================================
if __name__=="__main__":
  main()
```

C. Uji Coba Program

Program diuji dengan dataset dari kaggle.com. Hasil uji coba berhasil meningkatkan performa dengan meningkatkan akurasi menjadi 100%. Berikut adalah hasil uji coba dari model RNN:



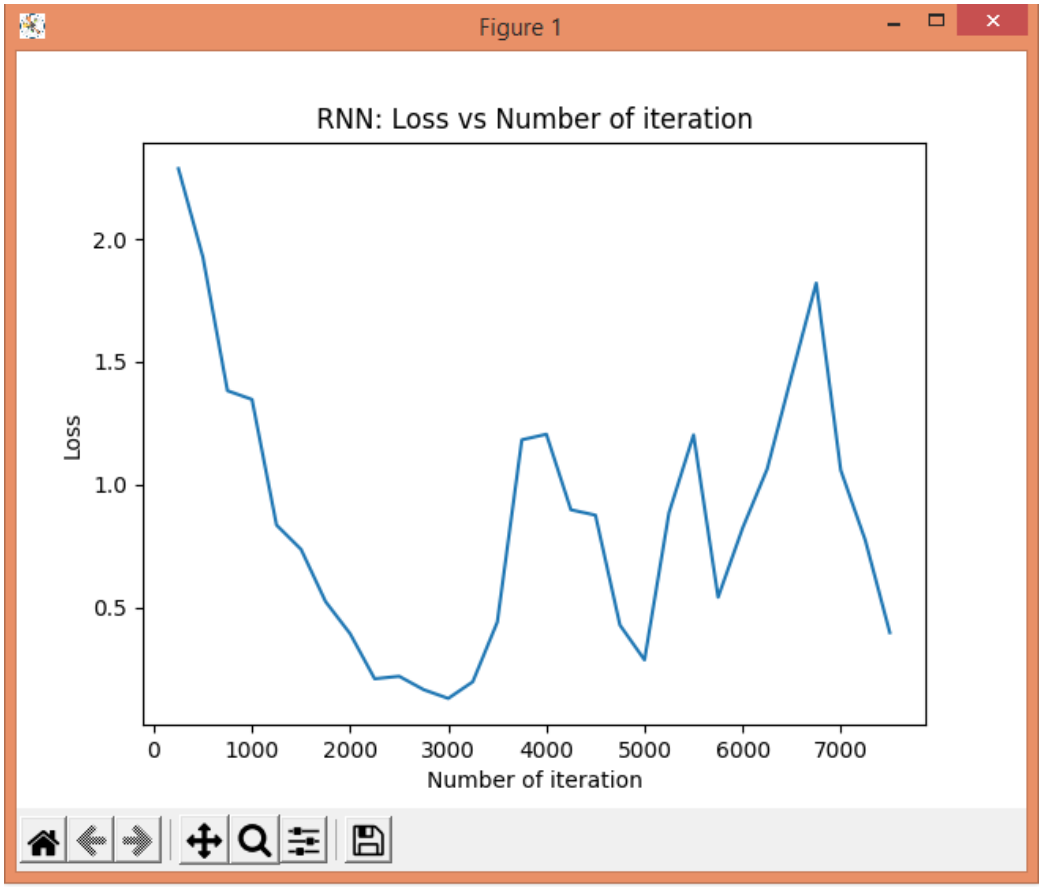


Figure 1
RNN: Loss vs Number of iteration
Loss
2.0
1.5
1.0
0.5
0
1000
2000
3000
4000
5000
6000
7000
Number of iteration

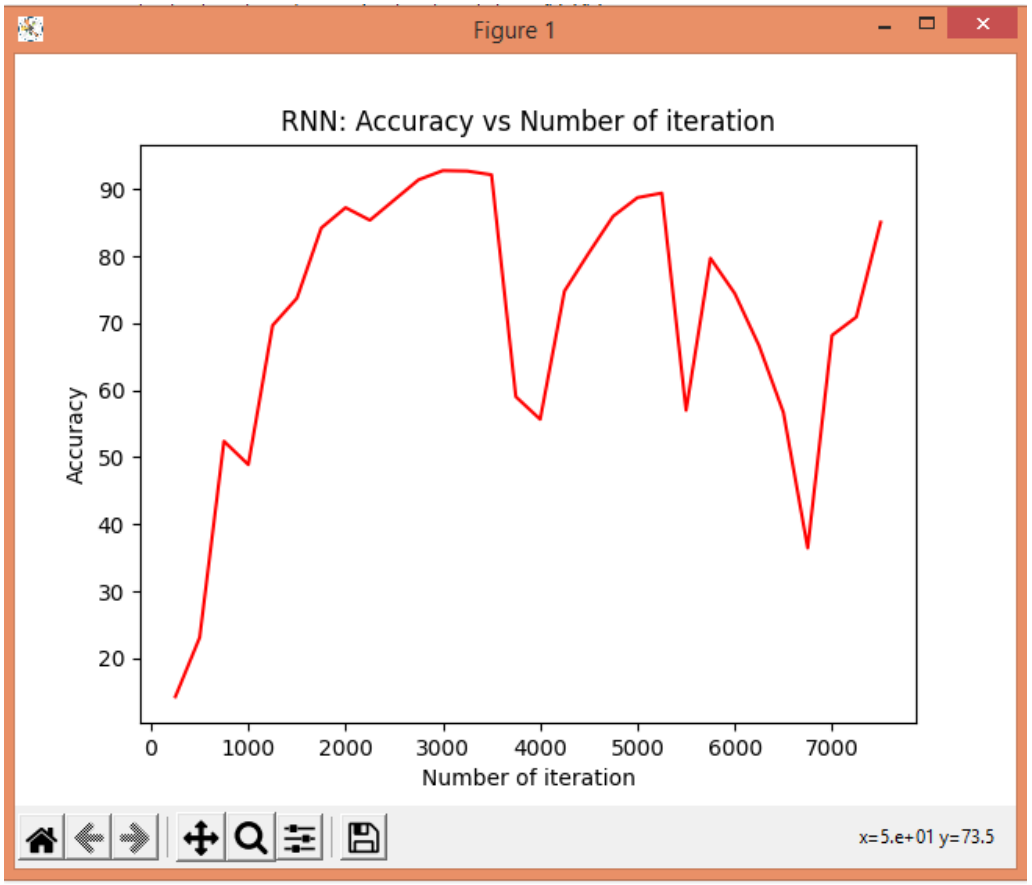
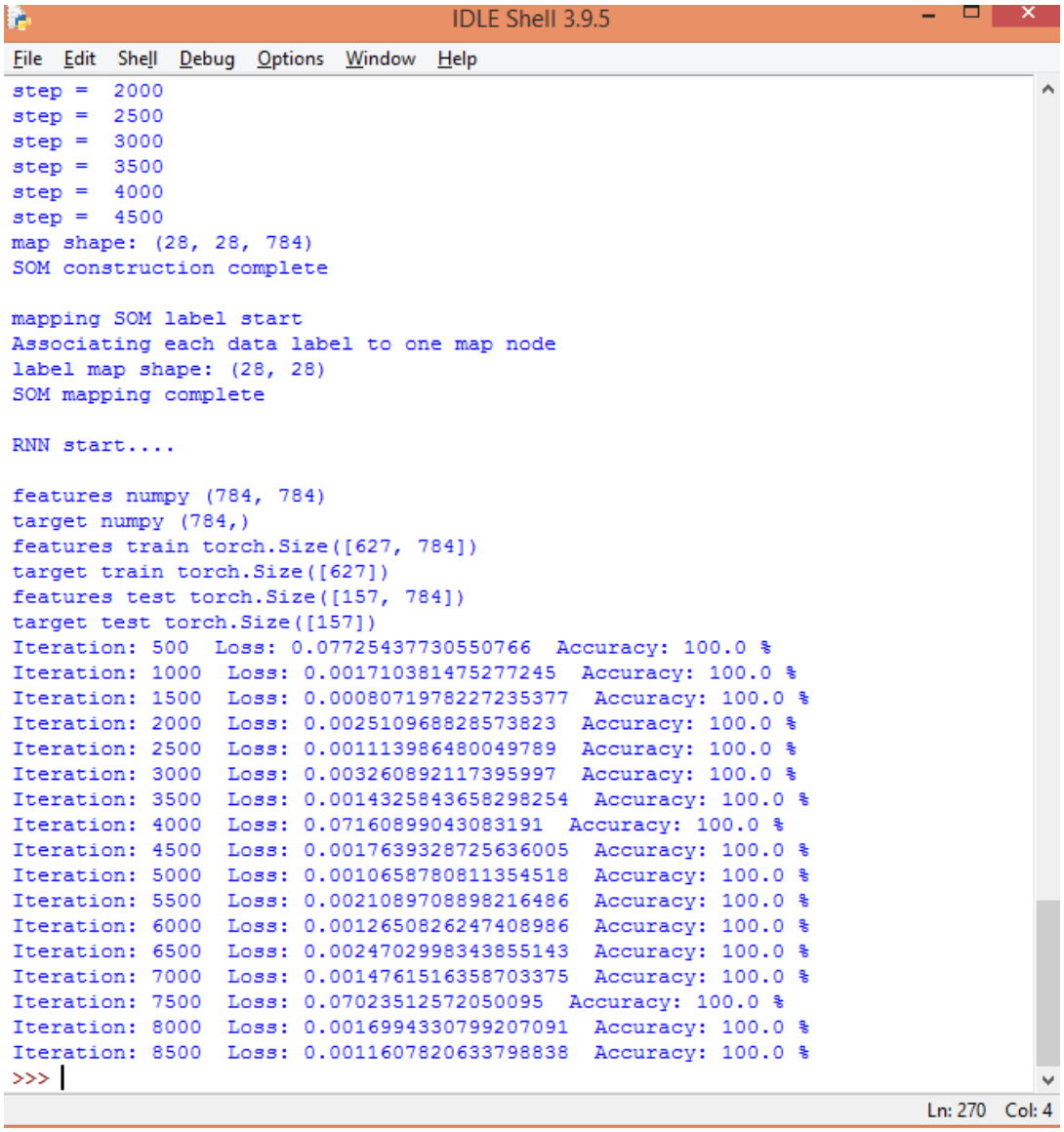


Figure 1
RNN: Accuracy vs Number of iteration
Accuracy
90
80
70
60
50
40
30
20
0
1000
2000
3000
4000
5000
6000
7000
Number of iteration
x=5.e+01 y=73.5

Berikut adalah hasil uji coba dari model SOM-RNN:



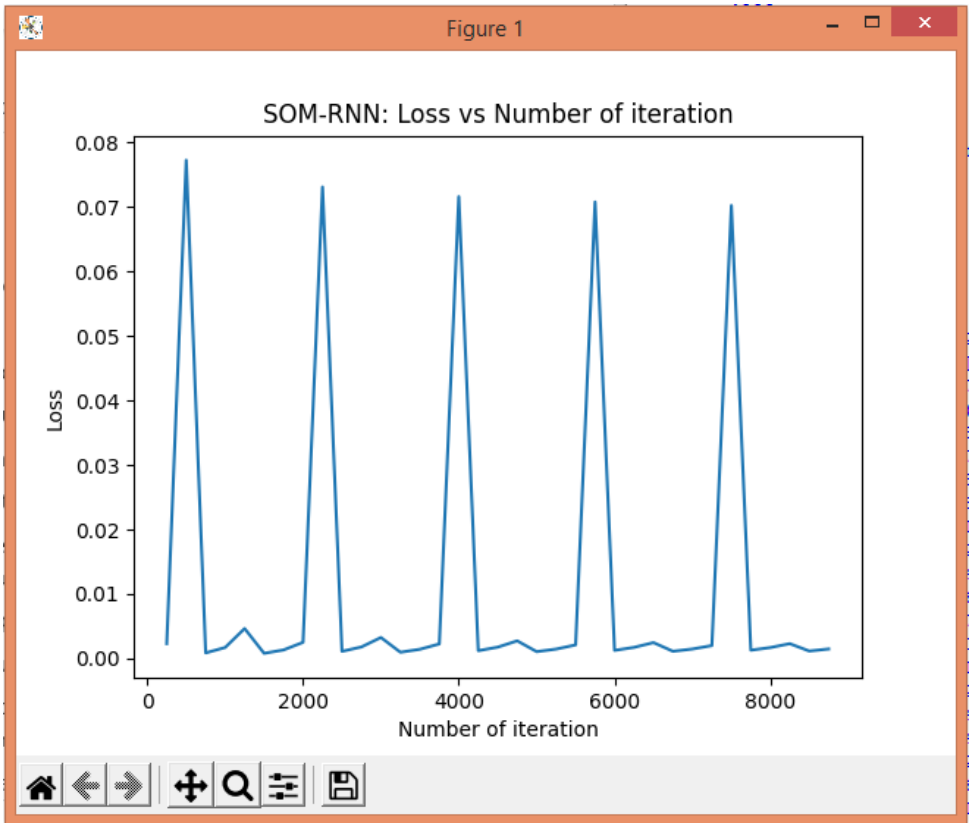


Figure 1
SOM-RNN: Loss vs Number of iteration
Loss
0.08
0.07
0.06
0.05
0.04
0.03
0.02
0.01
0.00
0
2000
4000
6000
8000
Number of iteration

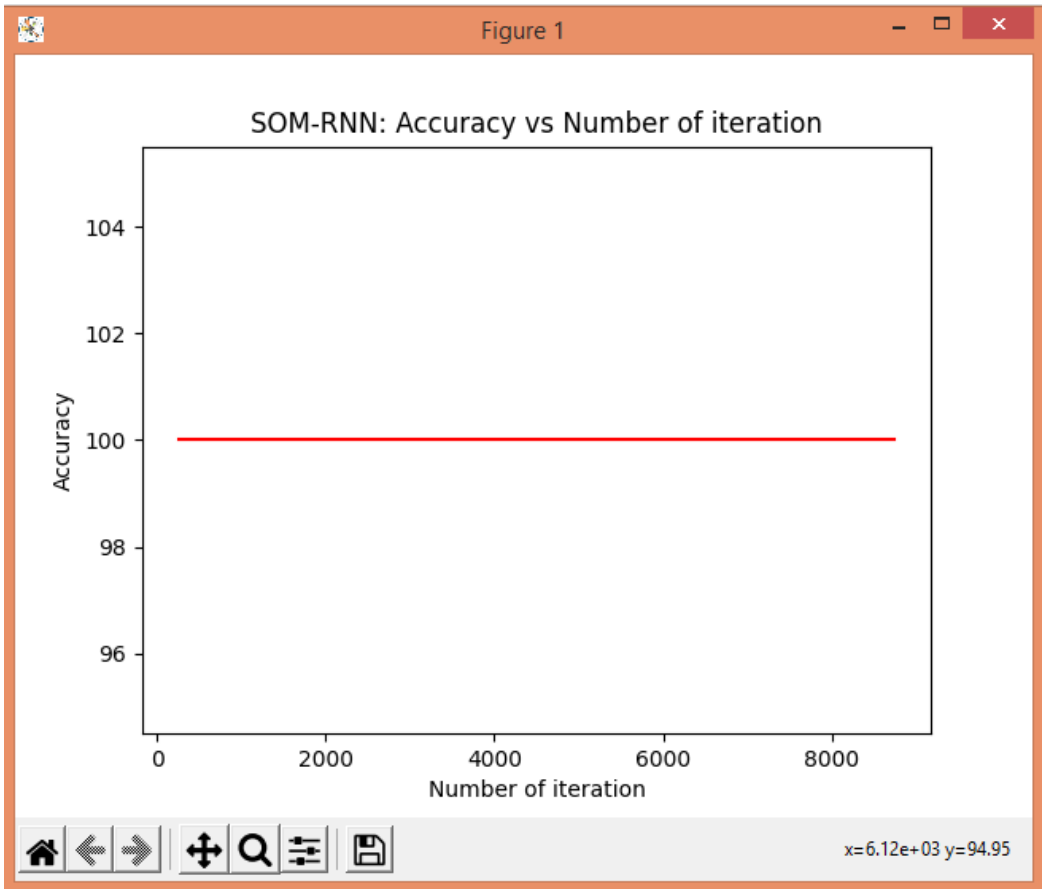


Figure 1
SOM-RNN: Accuracy vs Number of iteration
Accuracy
104
102
100
98
96
0
2000
4000
6000
8000
Number of iteration
x=6.12e+03 y=94.95