Politechnika Wrocławska
Wydział Informatyki i Telekomunikacji

Sieci neuronowe

Raport ćw. 7

Autor

Szymon Sawczuk

nr albumu: **260287**

kierunek: **Informatyka Stosowana**

15 grudnia 2023

Spis treści

1 Ćwiczenie 7 - Sieci rekurencyjne

Celem ćwiczenia jest przebadanie różnych typów sieci rekurencyjnych na bazie zbioru recenzji IMDB (*IMDB movie review sentiment classification dataset*, n.d.). Zbiór składa się z wcześniej przetworzonych recenzji w formie wartości liczbowych oraz przypisanych do tych recenzji wartości 0, 1 (określają czy recenzja była pozytywna czy negatywna).

1.1 Załadowanie i analiza danych

Po załadowaniu z pomocą keras danych zbioru IMDB, w celach szybszych obliczeń zmniejszam owy zbiór do 40% losowo wybranych danych oraz buduje za pomoca biblioteki pandas DateFrame określający nasz zbiór treningowy oraz testowy.

```
data = keras.datasets.imdb.load_data(
    path="imdb.npz",
    num_words=None,
    skip_top=0,
    maxlen=None,
    seed=113,
    start_char=1,
    oov_char=2,
    index_from=3
)

train_data = pd.DataFrame(data[0][0], columns=["review"])
train_data["label"] = data[0][1]
test_data = pd.DataFrame(data[1][0], columns=["review"])
test_data["label"] = data[1][1]

# Get 40% of data
train_data = train_data.sample(frac=0.4)
test_data = test_data.sample(frac=0.4)
```

Wynikiem są zbiory:

```
Train
                                              review   label
0  [1, 4, 487, 7, 4, 11834, 5, 4, 2594, 120, 4511...   0
1  [1, 13, 482, 48, 25, 26, 83, 4, 920, 924, 5, 1...   0
2  [1, 13, 219, 12, 13, 1041, 19, 90, 1245, 21, 1...   0
3  [1, 14, 468, 8, 30, 6, 52, 20, 13, 197, 12, 62...   0
4  [1, 13, 633, 358, 6151, 11835, 402, 102, 41, 1...   0

Test
                                              review   label
0  [1, 17, 210, 1820, 9, 389, 17, 1016, 1983, 130...   1
1  [1, 15641, 154, 108, 15, 25, 197, 71, 856, 218...   0
2  [1, 1400, 377, 54, 4314, 6108, 16, 530, 51, 6,...   0
3  [1, 1276, 13, 2904, 4, 7711, 11, 35, 576, 2514...   1
4  [1, 14, 9, 1018, 801, 18, 4, 920, 924, 1308, 3...   0
```

Zbudowane zbiory wyglądają w dużym stopniu na zbalansowane (różnica między klasami jest mało znacząca):

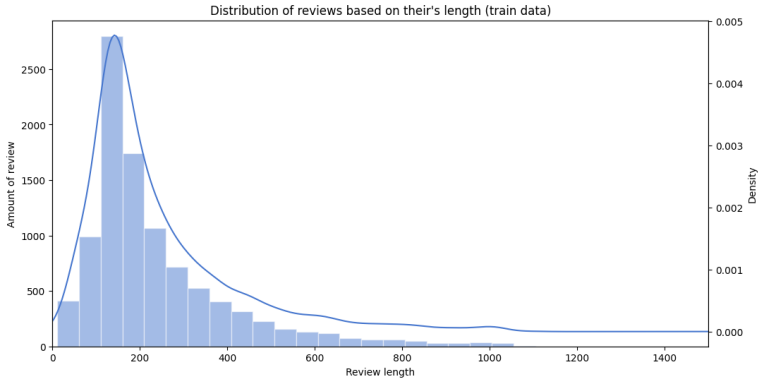
```
print(train_data["label"].value_counts()) # Get balance of train labels
print(test_data["label"].value_counts()) # Get balance of test labels
```

✓ 0.0s

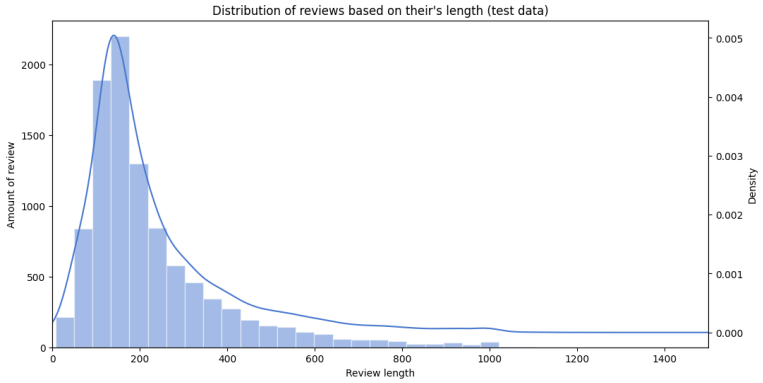
```
label
1    5003
0    4997
Name: count, dtype: int64
label
1    5017
0    4983
Name: count, dtype: int64
```

Rysunek 1: Zbalansowanie zbiorów

Warto także do eksperymentów zbadać dystrybucję długości recenzji w zbiorze treningowym i testowym:

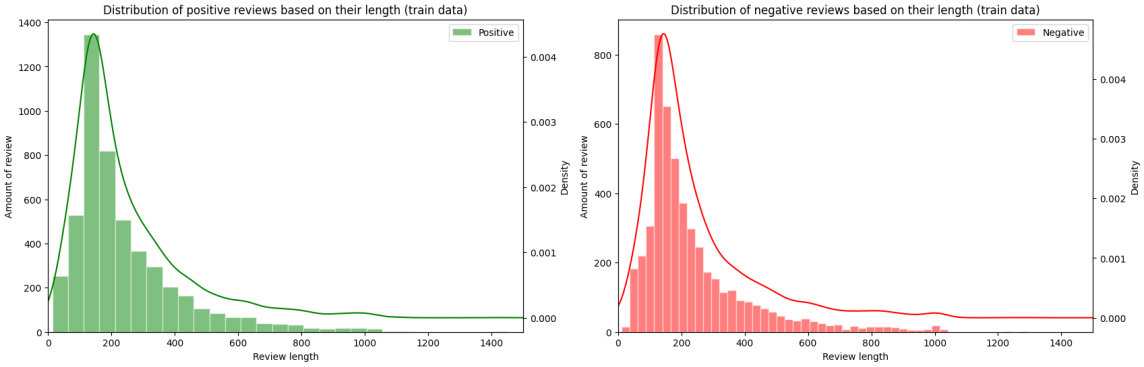


Rysunek 2: Dystrybucja zbioru treningowego względem długości recenzji

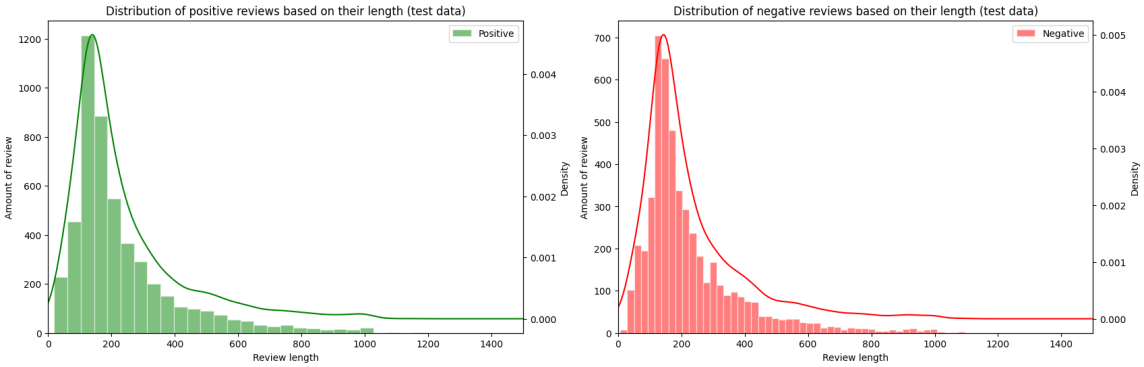


Rysunek 3: Dystrybucja zbioru testowego względem długości recenzji

Można zobaczyć, że największa ilość recenzji w zbiorach treningowych i testowych znajduje się w okolicach 100-180 słów. Podobne obserwację można zauważyć także badając dystrybucje poszczególnych klas recenzji:



Rysunek 4: Dystrybucja zbioru treningowego względem długości recenzji w rozdzieleniu na klasy



Rysunek 5: Dystrybucja zbioru testowego względem długości recenzji w rozdzieleniu na klasy

1.2 Sieć rekurencyjna

Sieć ta składa się z warstwy Embedding (*EMBEDDING*, n.d.), której celem jest przekształcenie danych wejściowych na wektory. Następnie tworzona jest na podstawie zmiennej network_type warstwa rekurencyjna sieci RNN (*RNN*, n.d.), albo LSTM (*LSTM*, n.d.). Obydwa typy są stworzone do przetwarzania sekwencji danych, natomiast RNN ma problemy z dłuższym zapamiętywaniem owych sekwencji i może pojawić się przez to problem zanikającego gradientu. Sieć LSTM natomiast została stworzona, aby zaradzić temu problemowi. Jest zbudowana w taki sposób, aby móc zapamiętywać długoterminowe zależności.

W warstwie wyjściowej sieci posiadamy jeden neuron, który po transformacji liniowej stosuje funkcję sigmoid'u.

W metodzie forward, RNN inicjalizuje stan ukryty, natomiast LSTM stan ukryty, a także komórkę pamięci. Następnie dane przechodzą po kolejnych warstwach sieci.

```
class RecursiveNetwork(torch.nn.Module):
    def __init__(self, hidden_layers: int, hidden_layers_size: int, input_size: int,
        output_size: int, network_type: str, vocabulary_size: int, batch_size: int):
        super(RecursiveNetwork, self).__init__()

        self.network_type = network_type
        self.hidden_layers_size = hidden_layers_size
        self.hidden_layers = hidden_layers
        self.batch_size = batch_size
```

```
        self.embedding = torch.nn.Embedding(vocabulary_size, input_size)

        match network_type:
            case "RNN":
                self.recursive_layer = torch.nn.RNN(input_size, hidden_layers_size,
                    hidden_layers, batch_first=True)
            case "LSTM":
                self.recursive_layer = torch.nn.LSTM(input_size, hidden_layers_size,
                    hidden_layers, batch_first=True)
            case _:
                raise ValueError("Unknown network type")
        self.output_layer = torch.nn.Linear(hidden_layers_size, output_size)
        torch.nn.init.normal_(self.output_layer.weight)
        torch.nn.init.normal_(self.output_layer.bias)
        self.sigmoid = torch.nn.Sigmoid()

    def forward(self, inputs):
       match self.network_type:
            case "RNN":
                hidden = torch.zeros(self.hidden_layers, self.batch_size,
                    self.hidden_layers_size)
            case "LSTM":
                hidden = torch.zeros(self.hidden_layers, self.batch_size,
                    self.hidden_layers_size), torch.zeros(self.hidden_layers,
                    self.batch_size, self.hidden_layers_size)
            case _:
                raise ValueError("Unknown network type")

       x = self.embedding(inputs)
       output, hidden = self.recursive_layer(x, hidden)
       output = output[:, -1, :]
       output = self.output_layer(output)
       output = self.sigmoid(output)

       return output, hidden
```

Do trenowania modelu utworzyłem funkcję podobną do tej z poprzedniego zadania.Miary jakości modelu obliczane są za pomocą klas o prefiksie Binaryclass z biblioteki torchmetrics.

```
def train_model(model: torch.nn.Module, train_set: torch.Tensor, test_set: torch.Tensor,
    batch_size: int, optimizer, loss, max_iter: int, learning_rate: float, verb=False):

    optimizer = optimizer(model.parameters(), lr = learning_rate)

    losses = []
    losses_test = []

    accuracy = []
    precision = []
    f_score = []
    recalls = []

    if batch_size > len(train_set):
        batch_size = len(train_set)

    data_loader_train = torch.utils.data.DataLoader(train_set, batch_size=batch_size,
        shuffle=True)
    data_loader_test = torch.utils.data.DataLoader(test_set, batch_size=batch_size,
        shuffle=True)
```

```
metric_accuracy = BinaryAccuracy()
metric_precision = BinaryPrecision()
metric_f_score = BinaryF1Score()
metric_recall = BinaryRecall()

for index in range(max_iter): #learn for max_iter
    curr_train_loss = 0
    curr_test_loss = 0
    metric_accuracy.reset()
    metric_precision.reset()
    metric_f_score.reset()
    metric_recall.reset()

    # for each batch perform learning
    for x_train_batch, y_train_batch in data_loader_train:
        optimizer.zero_grad()
        y_pred, train_hidden_pred = model(x_train_batch)
        loss_value = loss(y_pred.squeeze().float(), y_train_batch.float())
        loss_value.backward()
        optimizer.step()
        curr_train_loss += loss_value.item()

    losses.append(curr_train_loss / len(data_loader_train))

    with torch.no_grad():
        model.eval()
        for x_test_batch, y_test_batch in data_loader_test:
            test_pred, test_hidden_pred = model(x_test_batch)
            loss_test_value = loss(test_pred.squeeze().float(), y_test_batch.float())
            curr_test_loss += loss_test_value.item()

            #calculate scores for each batch of iteration
            metric_accuracy.update(test_pred.squeeze(), y_test_batch)
            metric_precision.update(test_pred.squeeze(), y_test_batch)
            metric_f_score.update(test_pred.squeeze(), y_test_batch)
            metric_recall.update(test_pred.squeeze(), y_test_batch)

        losses_test.append(curr_test_loss / len(data_loader_test))
        accuracy.append(metric_accuracy.compute())
        precision.append(metric_precision.compute())
        f_score.append(metric_f_score.compute())
        recalls.append(metric_recall.compute())

    if verb and index % 10 == 0:
        print("------------------------------------------------ Interation " + str(index))
        print("Train loss on " + str(index) + " iteration: ", losses[index])
        print("Test loss on " + str(index) + " iteration: ", losses_test[index])
        print("Accuracy on " + str(index) + " iteration: ", accuracy[index])
        print("Precision on " + str(index) + " iteration: ", precision[index])
        print("Recall on " + str(index) + " iteration: ", recalls[index])
        print("Fscore on " + str(index) + " iteration: ", f_score[index])
        print("------------------------------------------------\n")

print("Result of learning process for " + str(max_iter) + " iterations")
print("-----------------------------------------------------\n")
print("Train loss: ", losses[-1])
print("Test loss: ", losses_test[-1])
print("-----------------------------------------------\n")
print("Scores")
print("Accuracy: ", accuracy[-1])
```

```
    print("Precision: ", precision[-1])
    print("F_score: ", f_score[-1])
    print("Recall: ", recalls[-1])
    print("")
    return losses, losses_test, accuracy, precision, f_score, recalls
```

1.3 Eksperymenty na różnych parametrach modelu

Funkcja do wykonywania eksperymentów:

```
def find_max_item(data: torch.utils.data.TensorDataset) -> int:
    return max(review.max().item() for review, _ in data)

def get_vocabulary_size(train_data: torch.utils.data.TensorDataset, test_data:
    torch.utils.data.TensorDataset) -> int:
    return max(find_max_item(train_data), find_max_item(test_data)) + 1

import math
# options [0] -> network_type, [1] -> train_data, [2] -> test_data, [3] -> hidden_layers,
    [4] -> hidden_layers_size, [5] -> pad_size
def run_models(options: list, options_title: list):
    fig, axs = plt.subplots(math.ceil(len(options) / 2), 2, figsize=(20, 20))
    if math.ceil(len(options) / 2) == 1:
        axs = [axs] # Convert the single Axes object to a list

    metrics = pd.DataFrame(columns=["Accuracy", "Precision", "F_score", "Recall"])

    for option_index, option in enumerate(options):

        train_set =
            torch.utils.data.TensorDataset(torch.tensor(keras.utils.pad_sequences(option[1]["review"],
            maxlen=option[5])), torch.tensor(option[1]["label"]))
        test_set =
            torch.utils.data.TensorDataset(torch.tensor(keras.utils.pad_sequences(option[2]["review"],
            maxlen=option[5])), torch.tensor(option[2]["label"]))

        vocabulary_size = get_vocabulary_size(train_set, test_set)
        input_size = option[5]

        print(options_title[option_index])
        model = RecursiveNetwork(option[3], option[4], input_size, 1, option[0],
            vocabulary_size, batch_size)
        result = train_model(model, train_set, test_set, batch_size, optimizer, loss,
            max_iter, learning_rate, verb=verbose)

        plot_learning(result[0], result[1], options_title[option_index],
            axs[int(option_index / 2)][option_index % 2])
        new_row = [result[2][-1].item(), result[3][-1].item(), result[4][-1].item(),
            result[5][-1].item()]
        metrics.loc[len(metrics)] = new_row


    print("\nResults: ")
    metrics = metrics.set_axis(options_title, axis='index')
    metrics = metrics.transpose()
    display(metrics)
```

Parametry, które są wspólne dla wszystkich poniższych testów to:

- Optimizer - Adam
- Funkcja kosztu - Binarna entropia krzyżowa
- Liczba iteracji - 10
- Learning rate - 0.001
- 2 warstwy ukryte warstwy rekurencyjnej

- Liczba wyjść - 1
- Rozmiar batch'a - 500

W każdym eksperymencie porównywane zostają sieci RNN oraz LSTM.

1.4 Wyniki eksperymentów

1.4.1 Typ sieci oraz wymiar sieci rekurencyjnej

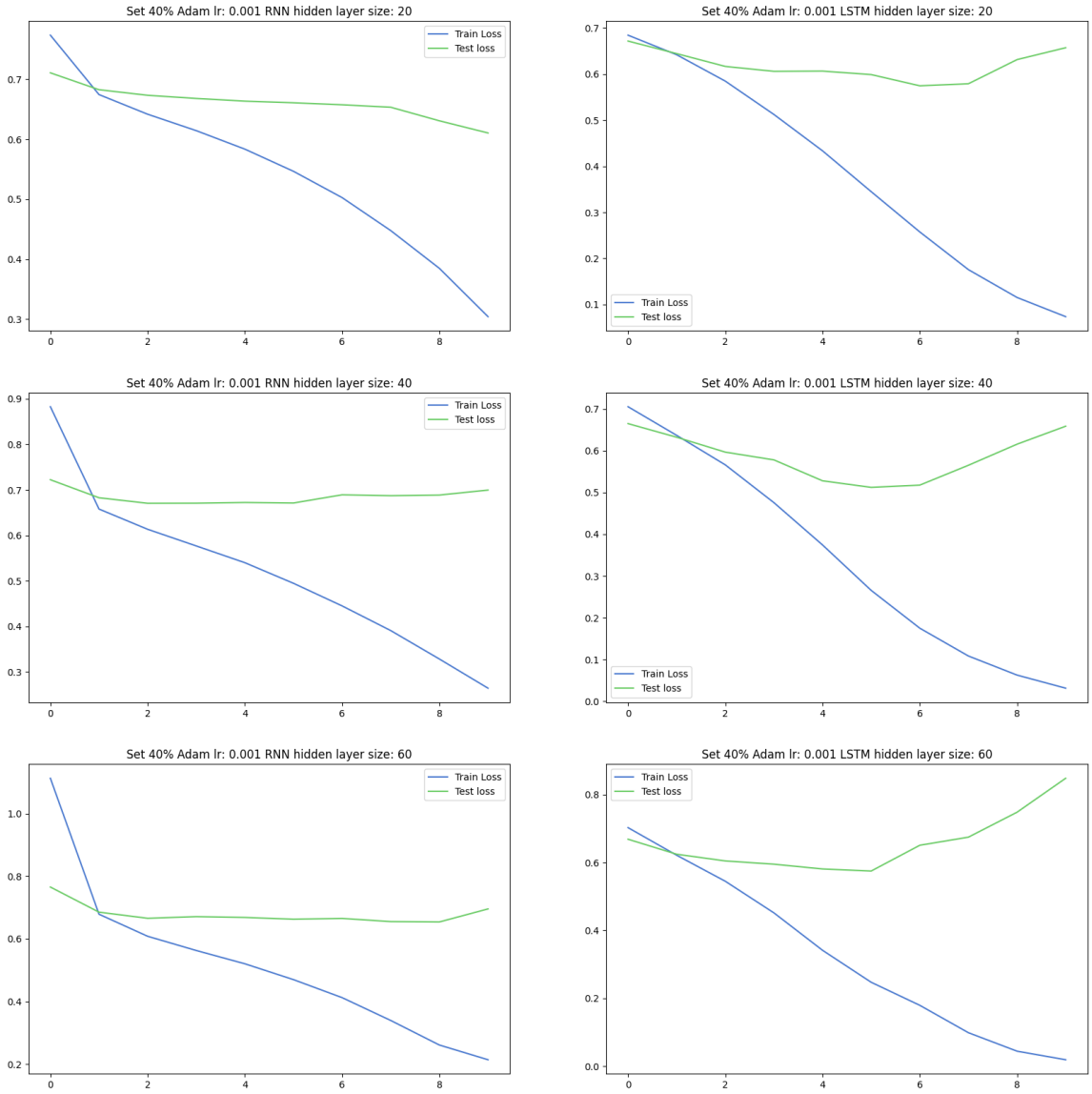
Dla rozmiaru przycięcia tekstów 250 uzyskujemy:

	20	**40**	**60**
Dokładność	0.7193	0.6831	0.7310
Precyzja	0.7240	0.7130	0.7523
F score	0.7179	0.6613	0.7206
Recall	0.7118	0.6166	0.6914

Tabela 1: Wyniki dla modelu RNN

	20	**40**	**60**
Dokładność	0.7790	0.7980	0.7629
Precyzja	0.7886	0.7919	0.7657
F score	0.7763	0.8010	0.7629
Recall	0.7644	0.8102	0.7600

Tabela 2: Wyniki dla modelu LSTM



Rysunek 6: Typ sieci oraz wymiar sieci rekurencyjnej

Wnioski:

- Sieć LSTM radzi sobie lepiej od RNN, wynika to z lepszym rozpoznawaniem dłuższych zależności w sekwencji danych
- Możemy zauważyć dla LSTM polepszenie się wyników przy zwiększeniu wymiaru sieci dla wartości 40 neuronów, natomiast gorsze uogólnianie nowych danych testowych dla 60 neuronów
- Dla RNN, jak i LSTM przyspieszyliśmy uczenie się sieci wraz z zwiększeniem wymiaru sieci, dla RNN objawia się to lepszymi wynikami dla 60 neuronów, niż dla 20 i 40 neuronów

1.4.2 Typ sieci oraz wpływ przycinania sekwencji do niepełnej długości

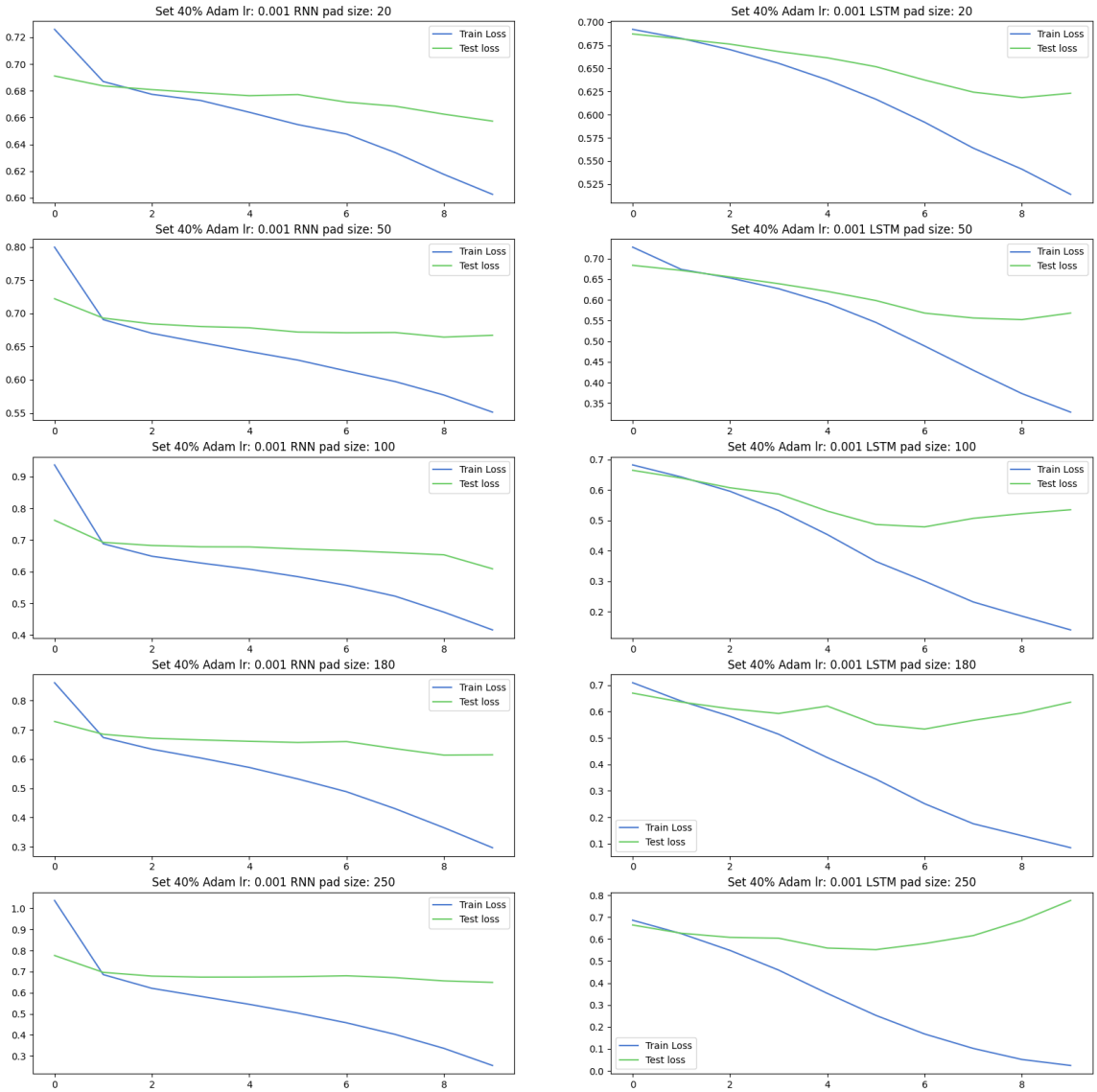
Dla wymiaru wartwy rekurencyjnej 40 uzyskujemy:

	20	**50**	**100**	**180**	**250**
Accuracy	0.6113	0.6166	0.6953	0.7246	0.7188
Precision	0.6233	0.6069	0.6829	0.6952	0.7394
F-score	0.5951	0.6367	0.7071	0.7453	0.7078
Recall	0.5693	0.6695	0.7331	0.8033	0.6787

Tabela 3: Wyniki dla modelu RNN

	20	**50**	**100**	**180**	**250**
Accuracy	0.6658	0.7372	0.7968	0.7784	0.7821
Precision	0.6524	0.7183	0.7833	0.7725	0.8000
F-score	0.6821	0.7494	0.8024	0.7818	0.7764
Recall	0.7146	0.7833	0.8226	0.7913	0.7542

Tabela 4: Wyniki dla modelu LSTM



Rysunek 7: Typ sieci oraz wpływ przycinania sekwencji do niepełnej długości

Wnioski:

- Podobnie jak w poprzednim badaniu, LSTM osiąga lepsze wyniki od RNN
- Zwiększając sekwencje danych dajemy sieci większy kontekst za pomocą, którego może się uczyć, dzięki czemu przyspieszyliśmy proces uczenia i owe sieci osiągają lepsze wyniki miar jakości
- Sieci osiągają najlepsze wyniki, kiedy przycinamy sekwencje w przedziale takim samym, gdzie dystrybucja długości danych w zbiorze osiąga największe wartości (owa dystrybucja pokazana jest na rysunku 2)

Badania sieci wskazują na lepsze działanie sieci LSTM względem RNN. Sieć LSTM osiągała wyniki miar w okolicach 80%. Natomiast sieć RNN w okolicach 74%.

Literatura

EMBEDDING. (n.d.). Retrieved from https://pytorch.org/docs/stable/generated/torch.nn.Embedding.html

IMDB movie review sentiment classification dataset. (n.d.). Retrieved from https://keras.io/api/datasets/imdb/

LSTM. (n.d.). Retrieved from https://pytorch.org/docs/stable/generated/torch.nn.LSTM.html

RNN. (n.d.). Retrieved from https://pytorch.org/docs/stable/generated/torch.nn.RNN.html