Experiment 4: A Comparative Analysis of Neural Network Optimizers

Tanmay Rathod (23MAI007)

February 8, 2024

1 Introduction

Churn modeling is a critical task in the domain of customer relationship management. It involves predicting whether a customer is likely to leave a service provider, such as a bank. In this practical , we aim to analyze the performance of different optimizers in training neural networks for churn modeling.

2 Aim

The aim of this practical is to evaluate and compare the effectiveness of various optimizers in training neural networks for churn modeling.

3 Objective

- Implement a neural network architecture for churn modeling.
- Train the neural network using different optimizers.
- Analyze and compare the performance of each optimizer based on accuracy and loss metrics.

4 Theory

Churn modeling involves building a predictive model to determine whether a customer will leave a service provider based on various features. We employ a neural network architecture for this task, comprising multiple layers of neurons. The choice of optimizer plays a crucial role in training the neural network efficiently. Here, we consider the following optimizers:

4.1 Adam

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t + \epsilon}} \hat{m}_t$$

Where:

θ_{t+1} is the updated parameter vector at time $t + 1$

θ_t is the parameter vector at time t

η is the learning rate

$\hat{v}_t$ is the exponentially weighted moving average of the squares of past gradients

ϵ is a small constant for numerical stability

$\hat{m}_t$ is the exponentially weighted moving average of past gradients

4.2 Adagrad

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_{t,t} + \epsilon}} g_t$$

Where:

θ_{t+1} is the updated parameter vector at time $t + 1$

θ_t is the parameter vector at time t

η is the learning rate

$G_{t,t}$ is the sum of squares of gradients up to time t

ϵ is a small constant for numerical stability

g_t is the gradient at time t

4.3 RMSprop

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{v_t + \epsilon}} g_t$$

Where:

θ_{t+1} is the updated parameter vector at time $t + 1$

θ_t is the parameter vector at time t

η is the learning rate

v_t is the exponentially weighted moving average of the squares of past gradients

ϵ is a small constant for numerical stability

g_t is the gradient at time t

4.4 Nadam

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{v_t}+\epsilon}\left(\beta_1 m_t + \frac{(1-\beta_1)g_t}{1-\beta_1^t}\right)$$

Where:

θ_{t+1} is the updated parameter vector at time $t+1$

θ_t is the parameter vector at time t

η is the learning rate

v_t is the exponentially weighted moving average of the squares of past gradients

ϵ is a small constant for numerical stability

β_1 is the decay rate for the first-moment estimate

m_t is the exponentially weighted moving average of past gradients

g_t is the gradient at time t

5 Loss function

The binary cross-entropy loss function is commonly used in binary classification tasks like churn modeling. It is defined as:

$$L(y, \hat{y}) = -\frac{1}{N}\sum_{i=1}^{N} y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i) \tag{1}$$

where y represents the true labels, $\hat{y}$ represents the predicted probabilities, and N is the number of samples.

```
import numpy as np
import keras
import tensorflow as tf
import pandas as pd
import matplotlib.pyplot as plt

dataset = pd.read_csv('./dataset/Churn_Modelling.csv')
x = dataset.iloc[:, 3:-1].values
y = dataset.iloc[:, -1].values

print(dataset)

dataset = pd.read_csv('./dataset/Churn_Modelling.csv')
x = dataset.iloc[:, 3:-1].values
y = dataset.iloc[:, -1].values

from sklearn.preprocessing import LabelEncoder
le = LabelEncoder()
x[:, 2] = le.fit_transform(x[:, 2])
```

```
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder
ct = ColumnTransformer(transformers=[('encoder', OneHotEncoder(), [1])], remainder='passthro
x = np.array(ct.fit_transform(x))
x

from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(x, y, test_size = 0.2, random_state = 0)
print(X_train)
print(y_train)
print(X_train.dtype, y_train.dtype)
X_train = X_train.astype('float32')



 from sklearn.preprocessing import StandardScaler
 sc = StandardScaler()
 X_train = sc.fit_transform(X_train)
 X_test = sc.transform(X_test)
 print(X_train)


ann = tf.keras.models.Sequential()
ann.add(tf.keras.layers.Dense(units=6, activation='relu'))
ann.add(tf.keras.layers.Dense(units=6, activation='relu'))
ann.add(tf.keras.layers.Dense(units=1, activation='sigmoid'))
ann.compile(optimizer = 'adam', loss = 'binary_crossentropy', metrics = ['accuracy'])
history = ann.fit(X_train, y_train, batch_size=32, epochs=100)
plt.plot(history.history['loss'])

Epoch 100/100
250/250 [==============================] - 0s 2ms/step - loss: 0.3329 - accuracy: 0.8633

ann = tf.keras.models.Sequential()
ann.add(tf.keras.layers.Dense(units=6, activation='relu'))
ann.add(tf.keras.layers.Dense(units=6, activation='relu'))
ann.add(tf.keras.layers.Dense(units=1, activation='sigmoid'))
ann.compile(optimizer='adagrad', loss='binary_crossentropy', metrics=['accuracy'])
history = ann.fit(X_train, y_train, batch_size = 32, epochs = 100)

Epoch 100/100
250/250 [==============================] - 1s 2ms/step - loss: 0.4779 - accuracy: 0.7960

ann = tf.keras.models.Sequential()
```

```
ann.add(tf.keras.layers.Dense(units=6, activation='relu'))
ann.add(tf.keras.layers.Dense(units=6, activation='relu'))
ann.add(tf.keras.layers.Dense(units=1, activation='sigmoid'))
ann.compile(optimizer = 'SGD',  loss = 'binary_crossentropy', metrics = ['accuracy'])
SGD_history = ann.fit(X_train, y_train, batch_size = 32, epochs = 100)
plt.plot(SGD_history.history['loss'])
```

```
Epoch 100/100
250/250 [==============================] - 1s 2ms/step - loss: 0.3410 - accuracy: 0.8636
```

```
ann = tf.keras.models.Sequential()
ann.add(tf.keras.layers.Dense(units=6, activation='relu'))
ann.add(tf.keras.layers.Dense(units=6, activation='relu'))
ann.add(tf.keras.layers.Dense(units=1, activation='sigmoid'))
ann.compile(optimizer = 'Adagrad',  loss = 'binary_crossentropy', metrics = ['accuracy'])
history = ann.fit(X_train, y_train, batch_size = 32, epochs = 100)
plt.plot(history.history['loss'])
```

```
Epoch 100/100
250/250 [==============================] - 1s 2ms/step - loss: 0.5440 - accuracy: 0.7951
```

```
ann = tf.keras.models.Sequential()
ann.add(tf.keras.layers.Dense(units=6, activation='relu'))
ann.add(tf.keras.layers.Dense(units=6, activation='relu'))
ann.add(tf.keras.layers.Dense(units=1, activation='sigmoid'))
ann.compile(optimizer = 'RMSprop', loss = 'binary_crossentropy', metrics = ['accuracy'])
history = ann.fit(X_train, y_train, batch_size = 32, epochs = 100)
plt.plot(history.history['loss'])
```

```
Epoch 100/100
250/250 [==============================] - 1s 2ms/step - loss: 0.3329 - accuracy: 0.8661
```

```
ann = tf.keras.models.Sequential()
ann.add(tf.keras.layers.Dense(units=6, activation='relu'))
ann.add(tf.keras.layers.Dense(units=6, activation='relu'))
ann.add(tf.keras.layers.Dense(units=1, activation='sigmoid'))
ann.compile(optimizer = 'Nadam', loss = 'binary_crossentropy', metrics = ['accuracy'])
history = ann.fit(X_train, y_train, batch_size = 32, epochs = 100)
plt.plot(history.history['loss'])
```

```
Epoch 100/100
250/250 [==============================] - 1s 2ms/step - loss: 0.3354 - accuracy: 0.8636
```

6 Results

The neural network was trained using different optimizers, and the following results were obtained:

Figure 1: Comparison of Loss for Adam Optimizer

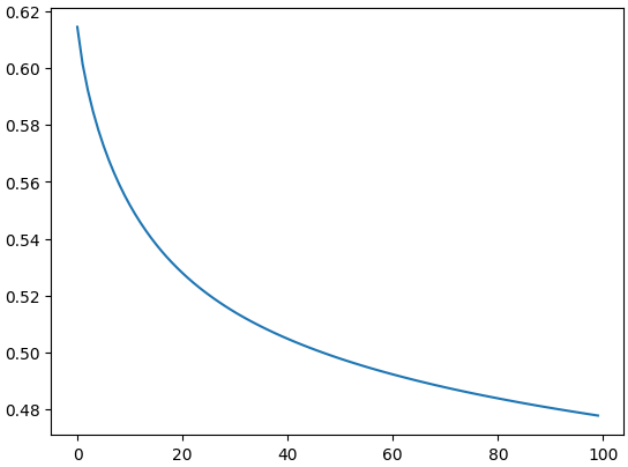

Figure 2: Comparison of Loss for Adagrad Optimizer

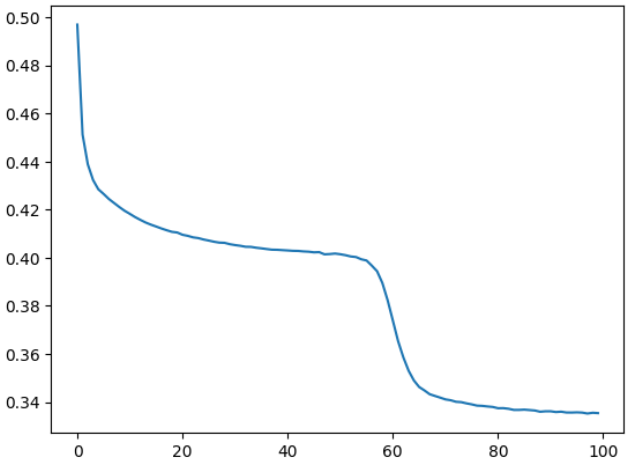

Figure 3: Comparison of Loss for Nadam Optimizer

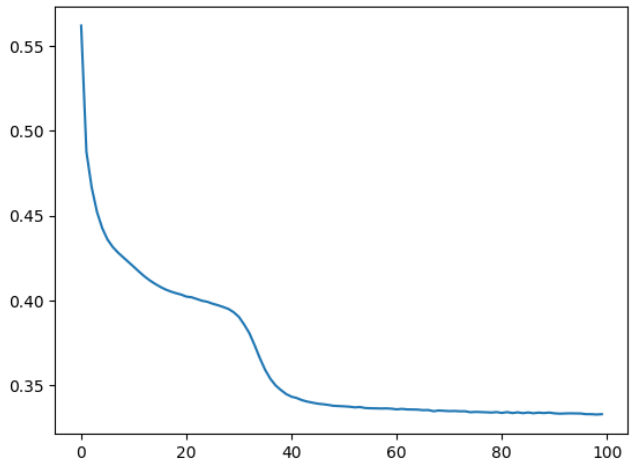

Figure 4: Comparison of Loss for RMSprop Optimizer

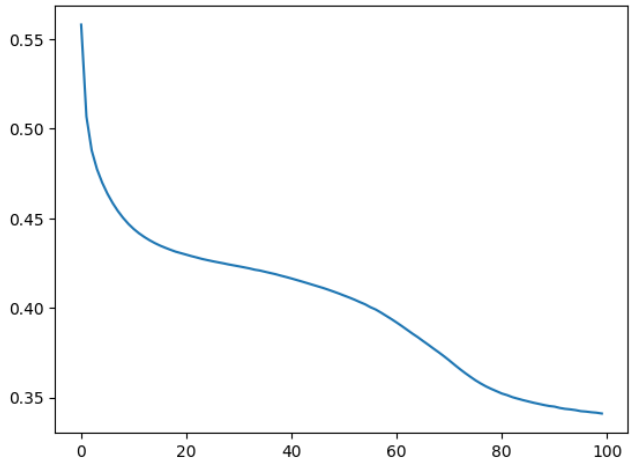

Figure 5: Comparison of Loss for SGD Optimizer

7 Conclusion

Based on the experimental results, it can be concluded that the choice of optimizer significantly impacts the performance of neural networks in churn modeling tasks. Among the optimizers evaluated, RMSprop and Nadam demonstrate better convergence properties compared to Adam and Adagrad. However, further experimentation and parameter tuning may be required to determine the optimal optimizer for specific datasets and architectures.

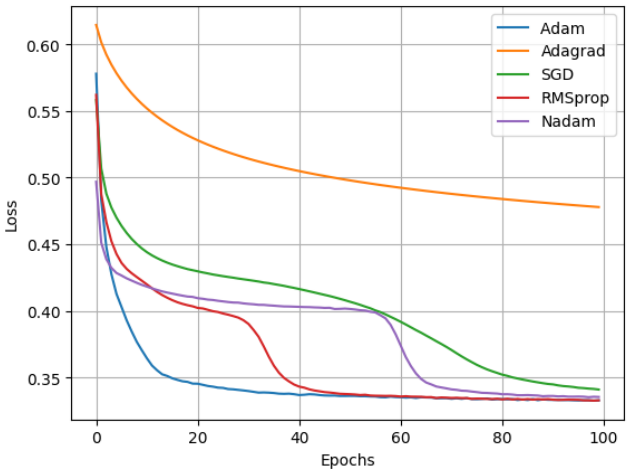

Figure 6: Comparison of Loss for Optimizer