Evaluating Optimization Methods for Predicting Football Outcomes with Feedforward Neural Networks

Julius Graf
julius.graf@berkeley.edu

Louis Sallé-Tourne
louis_salle-tourne@berkeley.edu

December 15, 2024

Abstract

This academic project, inspired by the "Challenge Data - Football: Can you guess the winner? by QRT" competition, aims to explore the behavior of neural network learning processes, with a particular focus on the optimization algorithms used during training. The project is part of the UC Berkeley Fall 2024 graduate course INDENG 240: Optimization Analytics, taught by Prof. Phillip Kerger. By analyzing a dataset containing various information about football teams and their match outcomes, we will train neural networks to predict the winners of football matches. This will allow us to apply and evaluate the gradient descent techniques learned in class.

Contents

1 Introduction **1**

2 Mathematical formalization **2**
2.1 Optimization problem . . . 2
2.2 Algorithms . . . 3
2.2.1 Gradient Descent . . . 3
2.2.2 Stochastic Gradient Descent . . . 3
2.2.3 Adam Optimization . . . 4

3 Results **5**
3.1 Parameters . . . 5
3.2 Results . . . 5
3.2.1 Performance Comparison . . . 5
3.2.2 Convergence Plots . . . 5

4 Conclusion **6**

1 Introduction

We aim to contribute to the 2024 "Challenge Data - Football: Can you guess the winner? by QRT"Football: Can you guess the winner? by QRT. Given that this project is part of the INDENG 240 course, our primary focus will be on the optimization aspects: defining the optimization problem, modeling through neural networks, and selecting appropriate optimization algorithms.

The competition organizers provide the training dataset for this challenge directly. Two tables give the input data:

- `train_home_team_statistics_df`: This table contains all information about the home team, with `game_id` as the key. It includes a variety of statistical data such as shots taken, passes made, and other performance metrics,

- `train_away_team_statistics_df`: This table has the same structure as the home team table but contains data for the away team, also keyed by `game_id`.

These tables can be joined using the `game_id` to create a comprehensive dataset for each match. The entire dataset comprises 12,203 rows, with 142 columns for each team (home and away). Given that we split the training data into training, testing, and validation data with an 80% ratio, the training set denoted $\mathcal{D}_m$ is of size $m = 7873$. The columns are predominantly numerical, except for the first two: `LEAGUE` and `TEAM_NAME`. Thus, the input size is $p = 280$ (dropping the league and the team name, one has 140 regressors for each of both teams). Examples of numerical columns include:

- `TEAM_SHOTS_TOTAL_season_sum`: Total shots taken by the team throughout the whole season,
- `TEAM_SHOTS_INSIDEBOX_season_sum`: Total shots taken inside the box by the team throughout the whole season,
- `TEAM_SHOTS_OFF_TARGET_season_sum`: Total off-target shots taken by the team throughout the whole season,
- `TEAM_SHOTS_ON_TARGET_season_sum`: Total on-target shots taken by the team throughout the whole season,
- `TEAM_SHOTS_OUTSIDEBOX_season_sum`: Total shots taken outside the box by the team throughout the whole season,
- `TEAM_PASSES_season_sum`: Total number of passes made by the team throughout the whole season,
- `TEAM_SUCCESSFUL_PASSES_season_sum`: Total number of successful passes made by the team throughout the whole season,

The label dataset, `Y_train`, contains the same number of rows (12,203) and includes the columns `HOME_WINS`, `DRAW`, and `AWAY_WINS`, with a value of 1 indicating the match result. Therefore, the whole dataset size is 12,203 Our task is a multiclass classification problem (thus, the number of classes is $c = 3$), where we aim to predict the outcome of each football match.

To achieve this, we will utilize the `PyTorch` library to build and train our neural network. This project will allow us to apply and evaluate the gradient descent techniques and other optimization algorithms learned in class. By focusing on the optimization aspects, we hope to gain insights into how different optimization strategies affect the performance of neural networks in predicting football match outcomes.

2 Mathematical formalization

2.1 Optimization problem

We consider a neural network $f_\theta \colon x \in \mathbb{R}^p \mapsto W_2\sigma(W_1x + b_1) + b_2 \in \mathbb{R}^c$, where the learnable parameters are $\theta = (W_1, W_2, b_1, b_2)$. These parameters belong to the space

$$\Theta = \mathcal{M}_{h,p}(\mathbb{R}) \times \mathcal{M}_{c,h}(\mathbb{R}) \times \mathbb{R}^h \times \mathbb{R}^c.$$

Here, $W_1 \in \mathcal{M}_{h,p}(\mathbb{R})$ and $W_2 \in \mathcal{M}_{c,h}(\mathbb{R})$ are weight matrices, while $b_1 \in \mathbb{R}^h$ and $b_2 \in \mathbb{R}^c$ are bias vectors. The activation function is set to $\sigma = \mathrm{ReLU}$. For practical purposes, the parameter space Θ can be assimilated into $\mathbb{R}^d$ with

$$d = h(1 + p + c) + c.$$

This transformation is achieved by flattening the matrices W_1 and W_2 using a column-wise vectorization function $\phi_{p,q} \colon \mathcal{M}_{p,q}(\mathbb{R}) \to \mathbb{R}^{pq}$, defined as

$$\phi_{p,q}(A) = \begin{bmatrix} A_1 \\ A_2 \\ \vdots \\ A_q \end{bmatrix},$$

where $A_i \in \mathbb{R}^p$ is the i-th column of $A \in \mathcal{M}_{p,q}(\mathbb{R})$. Thus, $W_1 \equiv \phi_{h,p}(W_1)$ and $W_2 \equiv \phi_{c,h}(W_2)$. We adopt the cross-entropy loss function, defined as follows:

$$\forall \mathcal{D} \subseteq \mathbb{R}^p \times \mathbb{R}^c,\ |\mathcal{D}| < \infty, \quad \forall \theta \in \mathbb{R}^d, \quad l(\theta, \mathcal{D}) = -\frac{1}{m} \sum_{(x,y)\in\mathcal{D}} \sum_{j=1}^{c} y_j \operatorname{softmax}(f_\theta(x))_j.$$

Here, $\mathcal{D}$ is a dataset of finite size, with $m = |\mathcal{D}|$. For a specific dataset

$$\mathcal{D}_m = \{(x_i, y_i) \in \mathbb{R}^p \times \mathbb{R}^c \mid 1 \leq i \leq m\},$$

we aim to solve the following optimization problem:

$$(\mathcal{P}): \quad \min_{\theta\in\Theta} l(\theta, \mathcal{D}_m) + \frac{\kappa}{2m}\|\theta\|_2^2.$$

Here, κ is a regularization weight that penalizes the magnitude of the parameters. The focus of our study is to compare various optimization methods for solving the problem $(\mathcal{P})$.

2.2 Algorithms

In this subsection, we outline the optimization methods used to solve the problem $(\mathcal{P})$. These include Gradient Descent (GD), Stochastic Gradient Descent (SGD), and the Adam optimizer. Each method is described below.

2.2.1 Gradient Descent

The classic Gradient Descent algorithm iteratively updates the parameters θ by following the negative gradient of the loss function.

Algorithm 1 Gradient Descent (GD)

1: Initialize θ_0, $n \leftarrow 0$, $\eta > 0$, $\kappa \geq 0$ and number of epochs T
2: **for** $n = 1, 2, \ldots, T$ **do**
3: $\quad \theta_n = \theta_{n-1}\left(1 - \frac{\eta\kappa}{m}\right) - \frac{\eta}{m}\sum_{i=1}^{m} \nabla_\theta l(\theta_{n-1}, \{x_i, y_i\})$
4: **end for**
5: **return** θ_n

However, since the gradient is computed in a single run over the whole dataset, convergence will be pretty slow and the main bottleneck is memory inefficiency. We therefore next present two more advanced algorithms, that are also used in practice, to compare their performance afterward.

2.2.2 Stochastic Gradient Descent

Stochastic gradient descent works by studying the Markov chain $\{X_n;\, n \in \mathbb{N}\}$ defined through its random mapping representation $X_{n+1} = f_{n+1}(X_n)$ where

$$f_{n+1}(\theta) = \theta\left(1 - \frac{\eta\kappa}{m}\right) - \frac{\eta}{b}\sum_{i\in B_{n+1}} \nabla_\theta l(\theta, \{x_i, y_i\}).$$

Here, b is the (fixed) size of the mini-batch B_{n+1}, a random subset of $[\![1, m]\!]$, dependent on $n+1$ as it is reshuffled at each iteration. The functions f_n for n in $\mathbb{N}^*$ are iid copies of a locally Lipschitz random mapping $f\colon \mathbb{R}^d \to \mathbb{R}^d$.[1]

[1] Yanlin Qu, Jose Blanchet, and Peter Glynn. *Deep Learning for Computing Convergence Rates of Markov Chains.* 2024. arXiv: `2405.20435 [cs.LG]`. URL: `https://arxiv.org/abs/2405.20435`.

Algorithm 2 Stochastic Gradient Descent (SGD)

Initialize X_0, $b \in \mathbb{N}^*$, $k \leftarrow 0$, $\eta > 0$, $\kappa \geq 0$ and number of epochs T
for $n = 1, 2, \ldots, T$ **do**
  Sample random permutation π according to $\pi \sim \mathcal{U}(\mathfrak{S}_m)$
  Divide $\mathcal{D}_m$ into $N = \lceil m/b \rceil$ mini-batches $B_j^{(n)} = \pi(\llbracket (j-1)b + 1, \min(jb, m) \rrbracket)$ for j in $\llbracket 1, N \rrbracket$
  for $j = 1, \ldots, N$ **do**
    $X_k \leftarrow X_k \left(1 - \frac{\eta\kappa}{m}\right) - \frac{\eta}{b} \sum_{i \in B_j^{(n)}} \nabla_\theta l(X_k, \{x_i, y_i\})$
    $k \leftarrow k + 1$
  end for
end for
return θ_k

Note that when one chooses $b = m$, the algorithm becomes deterministic and corresponds exactly to the gradient descent algorithm. In fact, one will only have one mini-batch $\pi(\llbracket 1, m \rrbracket) = \llbracket 1, m \rrbracket$ for all π in $\mathfrak{S}_m$. Now, as we've seen in class, momentum is a point that enhances convergence and renders the algorithm less prone to local minima. The following algorithm, Adam, goes beyond the idea of first-order momentum.

2.2.3 Adam Optimization

The Adam optimizer extends SGD by incorporating momentum and adaptive learning rates. It tracks the first- and second-order moments of the gradient using moving averages. Bias-corrected estimates $\hat{u}_n$ and $\hat{v}_n$ are computed to ensure stability. The update rule is:

$$\theta_k \leftarrow \theta_k - \eta\, \hat{u}_k \odot \left(\hat{v}_k + \epsilon^2\right)^{-1/2},$$

where $\eta > 0$ is the learning rate, $\epsilon > 0$ is a small constant for numerical stability, and $\kappa \geq 0$ is the regularization weight.

Algorithm 3 Adam

Initialize X_0, $u_0 \leftarrow 0$, $v_0 \leftarrow 0$, $k \leftarrow 0$, $\alpha > 0$, $\beta > 0$, $\epsilon > 0$, $\eta > 0$, $\kappa \geq 0$, $b \in \mathbb{N}^*$, and number of epochs T
for $n = 1, 2, \ldots, T$ **do**
  Sample random permutation π according to $\pi \sim \mathcal{U}(\mathfrak{S}_m)$
  Divide $\mathcal{D}_m$ into $N = \lceil m/b \rceil$ mini-batches $B_j^{(n)} = \pi(\llbracket (j-1)b + 1, \min(jb, m) \rrbracket)$ for j in $\llbracket 1, N \rrbracket$
  for $j = 1, \ldots, N$ **do**
    $g_k \leftarrow \frac{1}{b} \sum_{i \in B_j^{(n)}} \nabla_\theta l(X_k, \{x_i, y_i\}) + \frac{\kappa}{m} X_k$
    $u_k \leftarrow \alpha u_k + (1 - \alpha) g_k$
    $v_k \leftarrow \beta v_k + (1 - \beta) g_k \odot g_k$
    $\hat{u}_k \leftarrow u_k / \left(1 - \alpha^{k+1}\right)$
    $\hat{v}_k \leftarrow v_k / \left(1 - \beta^{k+1}\right)$
    $X_k \leftarrow X_k - \eta\, \hat{u}_k \odot \left(\hat{v}_k + \epsilon^2\right)^{-1/2}$
    $k \leftarrow k + 1$
  end for
end for
return θ_k

3 Results

3.1 Parameters

The practical values used in this study are summarized in Table 1. Parameters are categorized into Model Parameters, Training Settings, and Optimization Settings for clarity. Mathematical notations are provided where applicable, along with the corresponding values.

Category	Notation	Value
Model Parameters		
Input Dimension	p	280
Number of Classes	c	3
Hidden Layer Dimension	h	32
Parameter Dimension	d	9091
Activation Function	σ	ReLU
Training Settings		
Learning Rate	η	0.001
Batch Size	b	64
Number of Epochs	T	20
Loss Function	$l(\theta, \mathcal{D})$	Cross-Entropy Loss
Regularization weight	κ	0.7873
Early Stopping Patience	–	5 epochs
Size of Training Set	m	7873
Train-Test Splitting Factor	–	0.8
Optimization Settings		
Optimizer	–	Adam, SGD, GD (comparative study)
Learning Rate Scheduler	StepLR(τ, γ)	StepLR (step size: $\tau = 20$, decay factor: $\gamma = 0.5$)
Betas	(α, β)	(0.9, 0.999)
Numerical Stabilizer	ϵ	0.001

Table 1: Parameters, mathematical notation, and corresponding values used for training and evaluation. Parameters are grouped into model-specific settings, training configurations, and optimization techniques.

3.2 Results

3.2.1 Performance Comparison

The performance of the models trained using the SGD and Adam optimizers is summarized in the table below:

Optimizer	Final Validation Loss	Final Validation Accuracy (%)
SGD	1.0303	48.96
Adam	1.0690	48.84

Table 2: Performance comparison of SGD and Adam optimizers.

3.2.2 Convergence Plots

The figures below show the convergence of the training and validation losses and the validation accuracies for both optimizers.

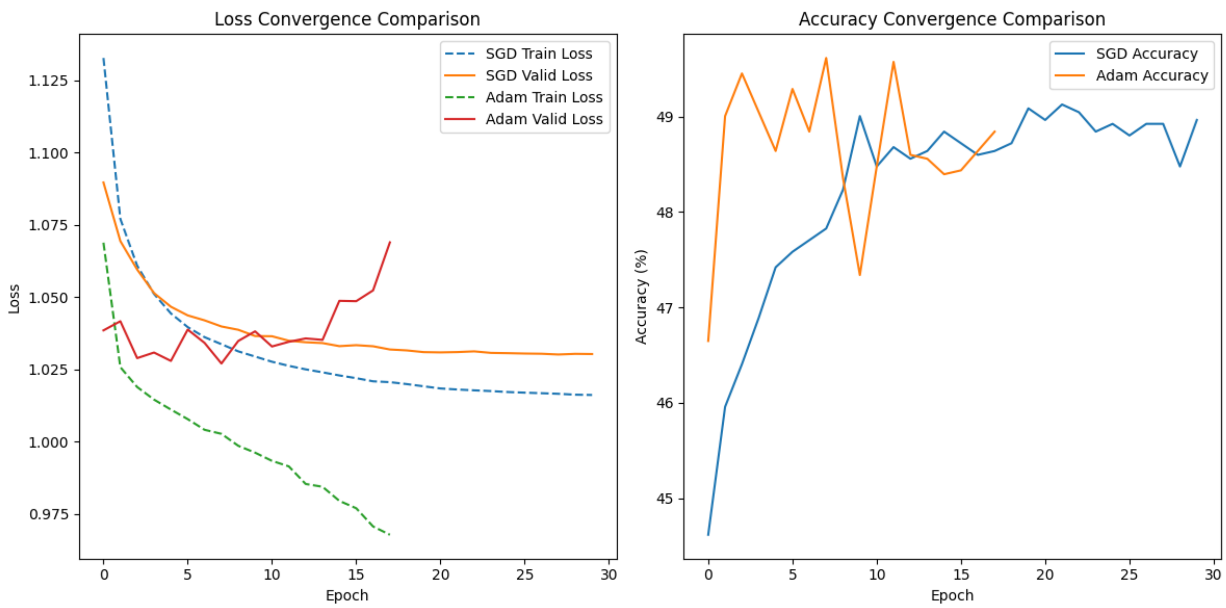

Figure 1: Left: Loss convergence for SGD and Adam optimizers. Right: Accuracy convergence for SGD and Adam optimizers.

4 Conclusion

This study evaluated the performance of feedforward neural networks trained with two optimization algorithms, SGD and Adam, to predict football match outcomes. Key results were compared against two benchmark models: (1) the baseline predicting that the HOME team wins, achieving 44% accuracy, and (2) a gradient boosting tree model trained solely on numerical features, yielding 47.5% accuracy on the training set.

The neural network models achieved validation accuracies of 48.96% with SGD and 48.84% with Adam, demonstrating marginal improvement over the benchmarks. Both optimizers displayed similar final performance but revealed notable differences in their behavior:

- SGD achieved slightly higher validation accuracy and lower final validation loss, indicating a more stable convergence to a solution that generalizes better. The simplicity of SGD's fixed learning rate may have contributed to more consistent updates during training, particularly for this problem's dataset and network structure.

- Adam, while competitive, showed a slightly higher final validation loss, which suggests a possible issue with overfitting or sensitivity to hyperparameter settings. Adam's adaptive learning rate mechanism can sometimes cause oscillations or premature convergence in scenarios where the learning rate becomes too aggressive or inconsistent. This could explain its slightly inferior performance compared to SGD in this study.

Despite these challenges, Adam exhibited faster initial convergence, as observed in the loss and accuracy plots. Its ability to adjust learning rates dynamically allowed it to make rapid progress early in training, which is a notable advantage over SGD. This aspect of Adam can be particularly beneficial in cases where computational efficiency or early stopping is critical.

In summary, while SGD proved marginally superior in this specific task due to its stability and generalization, Adam's faster convergence remains advantageous in certain contexts. The results underscore the importance of tailoring optimization strategies to the characteristics of the dataset and model. Future work could involve further hyperparameter tuning, regularization techniques, and hybrid training strategies to fully harness the strengths of both optimizers and achieve better overall performance.

Appendix: Python3 Code

```python
import pandas as pd
import numpy as np
from sklearn import model_selection
import warnings
import matplotlib.pyplot as plt
import torch
from torch.utils.data import DataLoader, TensorDataset
import torch.nn as nn
import torch.optim as optim

warnings.filterwarnings('ignore')

# Load and preprocess data
train_home_team_statistics_df = pd.read_csv('data/Train_Data/train_home_team_statistics_df.csv', index_col=0)
train_away_team_statistics_df = pd.read_csv('data/Train_Data/train_away_team_statistics_df.csv', index_col=0)
train_scores = pd.read_csv('data/Y_train_1rknArQ.csv', index_col=0)

train_home = train_home_team_statistics_df.iloc[:, 2:]
train_away = train_away_team_statistics_df.iloc[:, 2:]

train_home.columns = 'HOME_' + train_home.columns
train_away.columns = 'AWAY_' + train_away.columns

train_data = pd.concat([train_home, train_away], join='inner', axis=1)
train_scores = train_scores.loc[train_data.index]

train_data = train_data.replace({np.inf: np.nan, -np.inf: np.nan})
train_scores_indices = train_scores.idxmax(axis=1).map({'HOME_WINS': 0, 'DRAW': 1, 'AWAY_WINS': 2})

X_train, X_valid, y_train, y_valid = model_selection.train_test_split(
    train_data, train_scores_indices, train_size=0.8, random_state=42
)

# Prepare data for PyTorch
X_train_tensor = torch.from_numpy(X_train.fillna(0).to_numpy()).float()
y_train_tensor = torch.from_numpy(y_train.to_numpy()).long()
X_valid_tensor = torch.from_numpy(X_valid.fillna(0).to_numpy()).float()
y_valid_tensor = torch.from_numpy(y_valid.to_numpy()).long()

train_dataset = TensorDataset(X_train_tensor, y_train_tensor)
valid_dataset = TensorDataset(X_valid_tensor, y_valid_tensor)

train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
valid_loader = DataLoader(valid_dataset, batch_size=64, shuffle=False)

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Define the Neural Network
class NeuralNet(nn.Module):
    def __init__(self, input_size, hidden_size, num_classes):
        super(NeuralNet, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(hidden_size, num_classes)

    def forward(self, x):
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        return x

# Define evaluation function
def evaluate(model, criterion, loader):
```

```python
    model.eval()
    total_loss, correct, total = 0, 0, 0
    with torch.no_grad():
        for inputs, labels in loader:
            inputs, labels = inputs.to(device), labels.to(device)
            outputs = model(inputs)
            total_loss += criterion(outputs, labels).item()
            predicted = outputs.argmax(dim=1)
            correct += (predicted == labels).sum().item()
            total += labels.size(0)
    accuracy = 100 * correct / total
    return total_loss / len(loader), accuracy

# Early stopping implementation
class EarlyStopping:
    def __init__(self, patience, verbose=False):
        self.patience = patience
        self.verbose = verbose
        self.best_loss = float('inf')
        self.counter = 0

    def step(self, current_loss):
        if current_loss < self.best_loss:
            self.best_loss = current_loss
            self.counter = 0
            return True
        else:
            self.counter += 1
            if self.counter >= self.patience:
                if self.verbose:
                    print("Early stopping triggered.")
                return False
        return True

# Train the model
def train_model(optimizer_name, lr=0.001, weight_decay=1e-5, hidden_size=32, num_epochs=30,
    early_stopping_patience=10):
    input_size = X_train_tensor.shape[1]
    num_classes = len(np.unique(y_train))

    model = NeuralNet(input_size, hidden_size, num_classes).to(device)
    criterion = nn.CrossEntropyLoss()

    if optimizer_name == 'SGD':
        optimizer = optim.SGD(model.parameters(), lr=lr, weight_decay=weight_decay)
    elif optimizer_name == 'Adam':
        optimizer = optim.Adam(model.parameters(), lr=lr, weight_decay=weight_decay)
    else:
        raise ValueError("Unknown optimizer")

    scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=20, gamma=0.5)
    early_stopping = EarlyStopping(patience=early_stopping_patience, verbose=True)

    train_losses, valid_losses, accuracies = [], [], []
    learning_rates = []

    for epoch in range(num_epochs):
        model.train()
        train_loss = 0
        for inputs, labels in train_loader:
            inputs, labels = inputs.to(device), labels.to(device)

            outputs = model(inputs)
            loss = criterion(outputs, labels)

            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
```

```python

            train_loss += loss.item()

        train_loss /= len(train_loader)
        train_losses.append(train_loss)

        valid_loss, valid_accuracy = evaluate(model, criterion, valid_loader)
        valid_losses.append(valid_loss)
        accuracies.append(valid_accuracy)

        learning_rates.append(optimizer.param_groups[0]['lr'])

        print(f'Epoch [{epoch+1}/{num_epochs}], Loss: {valid_loss:.4f}, Accuracy: {valid_accuracy:.2f}%, Learning Rate: {learning_rates[-1]:.6f}')

        if not early_stopping.step(valid_loss):
            break

        scheduler.step()

    return train_losses, valid_losses, accuracies, valid_loss, valid_accuracy, learning_rates

# Plot metrics
def plot_metrics(train_losses, valid_losses, accuracies, title_suffix=""):
    plt.figure(figsize=(12, 6))
    plt.subplot(1, 2, 1)
    plt.plot(train_losses, label='Train Loss')
    plt.plot(valid_losses, label='Valid Loss')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.title(f'Loss Convergence {title_suffix}')
    plt.legend()

    plt.subplot(1, 2, 2)
    plt.plot(accuracies, label='Accuracy')
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy (%)')
    plt.title(f'Accuracy Convergence {title_suffix}')
    plt.legend()
    plt.tight_layout()
    plt.show()

# Plot learning rate progression
def plot_learning_rate(learning_rates, title_suffix=""):
    plt.figure(figsize=(6, 4))
    plt.plot(learning_rates, label='Learning Rate')
    plt.xlabel('Epoch')
    plt.ylabel('Learning Rate')
    plt.title(f'Learning Rate Progression {title_suffix}')
    plt.legend()
    plt.tight_layout()
    plt.show()

# Plot comparison
def plot_comparison(sgd_results, adam_results):
    plt.figure(figsize=(12, 6))

    plt.subplot(1, 2, 1)
    plt.plot(sgd_results[0], label='SGD Train Loss', linestyle='--')
    plt.plot(sgd_results[1], label='SGD Valid Loss')
    plt.plot(adam_results[0], label='Adam Train Loss', linestyle='--')
    plt.plot(adam_results[1], label='Adam Valid Loss')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.title('Loss Convergence Comparison')
    plt.legend()

```

```
    plt.subplot(1, 2, 2)
    plt.plot(sgd_results[2], label='SGD Accuracy')
    plt.plot(adam_results[2], label='Adam Accuracy')
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy (%)')
    plt.title('Accuracy Convergence Comparison')
    plt.legend()

    plt.tight_layout()
    plt.show()

# Train and evaluate models
sgd_results = train_model('SGD')
adam_results = train_model('Adam')

# Plot results
plot_comparison(sgd_results, adam_results)
plot_learning_rate(sgd_results[5], title_suffix="(SGD)")
plot_learning_rate(adam_results[5], title_suffix="(Adam)")

# Final performance
print(f"SGD Final Validation Loss: {sgd_results[3]:.4f}, Final Validation Accuracy: {
    sgd_results[4]:.2f}%")
print(f"Adam Final Validation Loss: {adam_results[3]:.4f}, Final Validation Accuracy: {
    adam_results[4]:.2f}%")
```

Listing 1: See the repository https://github.com/juliusgraf/challenge_qrt_optim.