Experiment No. 3
Apply Nesterov Accelerated Gradient Algorithm on a feed forward neural network for Iris Flower classification
Date of Performance:
Date of Submission:

Aim: Apply Nesterov Accelerated Gradient Algorithm on a feed forward neural network for Iris Flower classification.

Objective: Ability to perform optimization technique on a feed forward neural network.

Theory:

Gradient Descent is an iterative optimization process that searches for an objective function's optimum value (Minimum/Maximum). It is one of the most used methods for changing a model's parameters in order to reduce a cost function in machine learning projects.

The primary goal of gradient descent is to identify the model parameters that provide the maximum accuracy on both training and test datasets. In gradient descent, the gradient is a vector pointing in the general direction of the function's steepest rise at a particular point. The algorithm might gradually drop towards lower values of the function by moving in the opposite direction of the gradient, until reaching the minimum of the function.

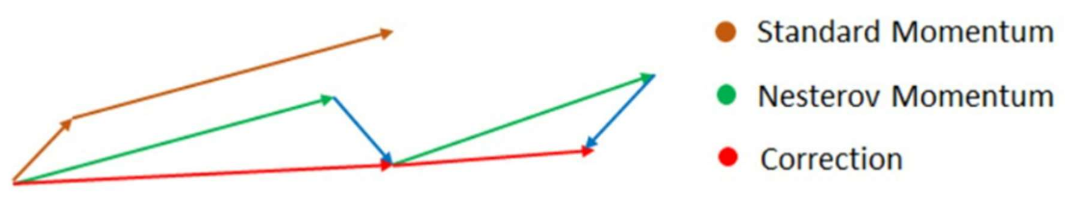
Types of Gradient Descent:

Typically, there are three types of Gradient Descent:

- Batch Gradient Descent
- Stochastic Gradient Descent
- Mini-batch Gradient Descent
- Nesterov Accelerated Gradient Algorithm

Nesterov Accelerated Gradient Algorithm:

Nesterov Accelerated Gradient, also known as Nesterov momentum or Nesterov's accelerated gradient descent, is an optimization technique that improves upon the standard momentum method. It was introduced by Yurii Nesterov in 1983 and has gained significant attention in recent years due to its superior convergence properties.



$$W_{t+1} = W_t - V_t$$

$$here, V_t = \beta * V_{t-1} + \eta \Delta W_t$$

The key idea behind NAG is to take into account the momentum term in the calculation of the gradient, by considering the future position of the parameter. Unlike standard momentum, which updates the parameters based on the current position, NAG incorporates an estimation of the future position by looking ahead. By doing so, NAG achieves faster convergence and better handling of oscillations in the loss landscape.

Nesterov Accelerated Gradient vs Standard Momentum

The main difference between Nesterov Accelerated Gradient and standard momentum is the order in which the gradient is calculated. In standard momentum, the gradient is calculated at the current location and then a big jump is taken in the direction of the updated accumulated gradient. In contrast, Nesterov momentum first makes a big jump in the direction of the previous accumulated gradient and then measures the gradient where it ends up and makes a correction. The intuition behind this is that it is better to correct a mistake after you have made it.

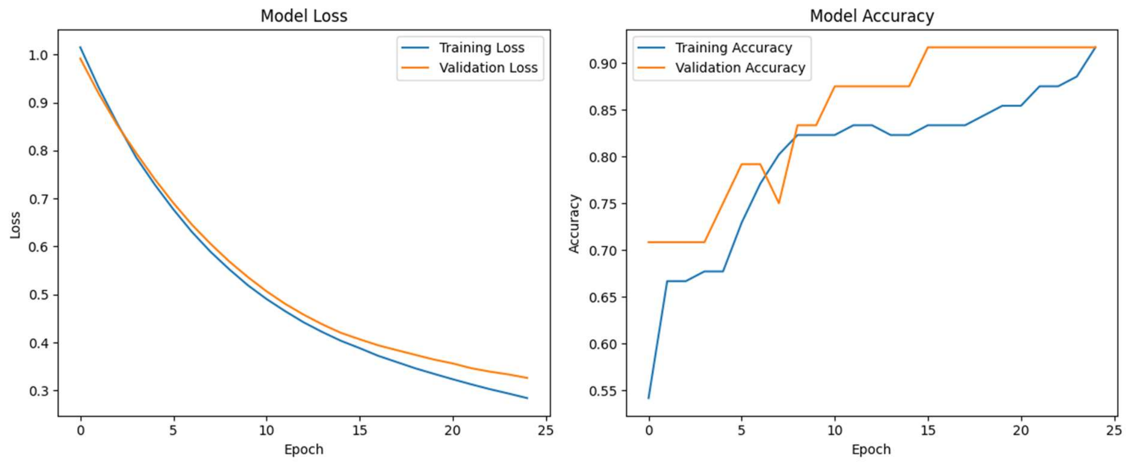
Nesterov Accelerated Gradient has been shown to converge faster than other optimization algorithms, especially when the cost function has a lot of shallow areas. In addition, it has been shown to be more robust to noise and can handle large-scale problems efficiently. However, it may not always be the best choice for all types of problems, and it's important to experiment with different optimization algorithms to find the one that works best for your specific problem.

Nesterov Accelerated Gradient is a momentum-based optimization algorithm that uses a look-ahead approach to calculate the gradient. It's a modification of the standard SGD algorithm and has been shown to be more efficient and robust in certain situations. Its use is becoming more and more prevalent in the field of machine learning, and it can be a powerful tool for improving the performance of your neural network models.

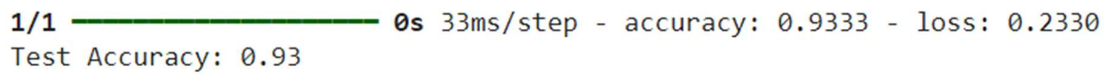
Implementation:

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import OneHotEncoder
import numpy as np
import matplotlib.pyplot as plt

# Load Iris dataset
iris = load_iris()
X = iris.data
y = iris.target

# Standardize features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# One-hot encode labels
encoder = OneHotEncoder(sparse=False)
y_encoded = encoder.fit_transform(y.reshape(-1, 1))

# Split dataset
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y_encoded, test_size=0.2, random_state=42)
import tensorflow as tf
```

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

# Define the model
model = Sequential([
    Dense(64, activation='relu', input_shape=(X_train.shape[1],)),
    Dense(64, activation='relu'),
    Dense(3, activation='softmax')  # Output layer with 3 classes
])

# Compile the model
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
history = model.fit(X_train, y_train, epochs=25, batch_size=32, validation_split=0.2)
# Extract loss and accuracy from the history object
history_dict = history.history

# Plot training & validation loss values
plt.figure(figsize=(12, 5))

plt.subplot(1, 2, 1)
plt.plot(history_dict['loss'])
plt.plot(history_dict['val_loss'])
plt.title('Model Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend(['Training Loss', 'Validation Loss'])

# Plot training & validation accuracy values
plt.subplot(1, 2, 2)
plt.plot(history_dict['accuracy'])
plt.plot(history_dict['val_accuracy'])
plt.title('Model Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend(['Training Accuracy', 'Validation Accuracy'])

plt.tight_layout()
plt.show()
```



```
test_loss, test_accuracy = model.evaluate(X_test, y_test)
print(f"Test Accuracy: {test_accuracy:.2f}")
```



```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 33ms/step - accuracy: 0.9333 - loss: 0.2330
Test Accuracy: 0.93
```

Conclusion:

Calculate and comment on the accuracy and structure of the network.

The neural network achieved a test accuracy of approximately `93.3%`, demonstrating its effectiveness in classifying the Iris species based on the provided features. The model's architecture, consisting of two hidden layers with 64 neurons each and ReLU activation functions, is well-suited for capturing complex patterns in the data. The validation accuracy and loss curves suggest that the model is well-generalized, avoiding both overfitting and underfitting. The choice of categorical crossentropy as the loss function and the Adam optimizer contributed to the model's efficient learning and overall strong performance.