Semester	B.E. Semester VII
Subject	Deep Learning
Subject Professor In- charge	Dr. Nayana Mahajan
Laboratory	M201B

Student Name	Harsh Jain	Division	B
Roll Number	22108B0054	Batch	4
Grade and Subject Teacher’s Signature			

Experiment Number	2
Experiment Title	To train and evaluate a single-layer feedforward neural network on a real-world binary classification dataset using Stochastic Gradient Descent (SGD) and Momentum-based Gradient Descent (Momentum GD) as optimization techniques. The objective is to compare and analyze the performance of both optimizers in terms of: ● Convergence Rate: How quickly the training loss decreases over epochs. ● Training Speed: The computational efficiency and time taken during training. ● Classification Accuracy: The predictive performance on unseen test data. This study aims to highlight the impact of optimization strategy on neural network training effectiveness, particularly in low-complexity models such as single-layer networks.
Resources / Apparatus Required	Software: Google Colab
Algorithm	1. **Load Dataset**: 2. **Create Binary Target**: 3. **One-Hot Encode Categorical Features**: 4. **Prepare Features and Labels**: 5. **Normalize Features**: 6. **Split Data into Training and Test Sets**: 7. **Define Activation and Loss Functions**: 8. **Initialize Weights and Bias**: 9. **Train the Model**: 10. **Evaluate the Model**:

Program code	# Using Sigmoid Activation Function

```
import tensorflow as tf
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten,
Dropout
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.callbacks import
EarlyStopping
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns
from sklearn.metrics import classification_report,
confusion_matrix
import warnings
warnings.filterwarnings('ignore')

# Set random seeds for reproducibility
np.random.seed(42)
tf.random.set_seed(42)

class OptimizerComparator:
    """Compare different gradient descent
optimizers on MNIST classification"""

    def __init__(self):
        self.history_dict = {}
        self.models = {}
        self.results = {}

        # Load and preprocess data
        self.load_and_preprocess_data()

        # Define optimizers with detailed
configurations
        self.optimizers = {
            "Nesterov (NAG)":
tf.keras.optimizers.SGD(
                learning_rate=0.01,
                momentum=0.9,
                nesterov=True
            ),
            "Adagrad": tf.keras.optimizers.Adagrad(
                learning_rate=0.01,
                initial_accumulator_value=0.1,
                epsilon=1e-07
            ),
            "Adam": tf.keras.optimizers.Adam(
                learning_rate=0.001,
                beta_1=0.9,
                beta_2=0.999,
```

```
                epsilon=1e-07
            )
        }

        print("🔬 ADVANCED GRADIENT DESCENT
OPTIMIZERS COMPARISON")
        print("   Dataset: MNIST Handwritten Digits
Classification")
        print("   Optimizers: Nesterov AGD,
Adagrad, Adam")
        print("="*60)

    def load_and_preprocess_data(self):
        """Load and preprocess MNIST dataset"""
        print("\n📊 Loading and preprocessing MNIST
dataset...")

        # Load MNIST dataset
        (self.x_train, self.y_train), (self.x_test,
self.y_test) = mnist.load_data()

        # Normalize pixel values to [0, 1]
        self.x_train =
self.x_train.astype('float32') / 255.0
        self.x_test = self.x_test.astype('float32')
/ 255.0

        # Convert labels to categorical (one-hot
encoding)
        self.y_train_cat =
to_categorical(self.y_train, 10)
        self.y_test_cat =
to_categorical(self.y_test, 10)

        print(f"   Training samples:
{self.x_train.shape[0]:,}")
        print(f"   Test samples:
{self.x_test.shape[0]:,}")
        print(f"   Image shape:
{self.x_train.shape[1:]} -> Flattened to {28*28}")
        print(f"   Classes:
{len(np.unique(self.y_train))} (digits 0-9)")

    def create_model(self):
        """Create Multi-Layer Perceptron (MLP)
model"""
        model = Sequential([
            # Input layer - flatten 28x28 images
            Flatten(input_shape=(28, 28),
name='flatten_input'),

            # First hidden layer
```

```
            Dense(128, activation='relu',
name='hidden_layer_1'),
            Dropout(0.2, name='dropout_1'),  # Add
dropout for regularization

            # Second hidden layer
            Dense(64, activation='relu',
name='hidden_layer_2'),
            Dropout(0.2, name='dropout_2'),

            # Output layer - 10 classes for digits
0-9
            Dense(10, activation='softmax',
name='output_layer')
        ])

        return model

    def train_with_optimizer(self, optimizer_name,
optimizer, epochs=25, batch_size=128, verbose=1):
        """Train model with specific optimizer"""
        print(f"\n🚀 Training with {optimizer_name}
optimizer...")

        # Create fresh model
        model = self.create_model()

        # Compile model with the specified
optimizer
        model.compile(
            optimizer=optimizer,
            loss='categorical_crossentropy',
            metrics=['accuracy']
        )

        # Early stopping callback to prevent
overfitting
        early_stopping = EarlyStopping(
            monitor='val_loss',
            patience=5,
            restore_best_weights=True,
            verbose=0
        )

        # Train the model
        history = model.fit(
            self.x_train, self.y_train_cat,
            epochs=epochs,
            batch_size=batch_size,
            validation_data=(self.x_test,
self.y_test_cat),
            callbacks=[early_stopping],
```

```
            verbose=verbose
        )

        # Store results
        self.history_dict[optimizer_name] = history
        self.models[optimizer_name] = model

        # Evaluate final performance
        train_loss, train_acc =
model.evaluate(self.x_train, self.y_train_cat,
verbose=0)
        test_loss, test_acc =
model.evaluate(self.x_test, self.y_test_cat,
verbose=0)

        self.results[optimizer_name] = {
            'train_loss': train_loss,
            'train_accuracy': train_acc,
            'test_loss': test_loss,
            'test_accuracy': test_acc,
            'epochs_trained':
len(history.history['loss'])
        }

        print(f"   Final Training Accuracy:
{train_acc:.4f}")
        print(f"   Final Test Accuracy:
{test_acc:.4f}")
        print(f"   Epochs trained:
{len(history.history['loss'])}")

        return history, model

    def run_all_experiments(self, epochs=25,
batch_size=128):
        """Run experiments for all optimizers"""
        print(f"\n🔄 Starting training with
{len(self.optimizers)} optimizers...")
        print(f"   Epochs: {epochs}, Batch size:
{batch_size}")

        for optimizer_name, optimizer in
self.optimizers.items():

self.train_with_optimizer(optimizer_name,
optimizer, epochs, batch_size)

        print(f"\n✅ All experiments completed!")

    def plot_comprehensive_results(self,
figsize=(16, 12)):
        """Create comprehensive visualization of
```

```
results"""

        fig, axes = plt.subplots(2, 3,
figsize=figsize)
        fig.suptitle('Advanced Gradient Descent
Optimizers Comparison on MNIST',
                    fontsize=16, fontweight='bold')

        # Color palette for optimizers
        colors = ['#1f77b4', '#ff7f0e', '#2ca02c']
# Blue, Orange, Green

        # 1. Training Loss Comparison
        ax1 = axes[0, 0]
        for i, (name, history) in
enumerate(self.history_dict.items()):
            epochs_range = range(1,
len(history.history['loss']) + 1)
            ax1.plot(epochs_range,
history.history['loss'],
                    label=name, color=colors[i],
linewidth=2, marker='o', markersize=4)

        ax1.set_title('Training Loss Comparison')
        ax1.set_xlabel('Epochs')
        ax1.set_ylabel('Loss')
        ax1.legend()
        ax1.grid(True, alpha=0.3)
        ax1.set_yscale('log')

        # 2. Validation Loss Comparison
        ax2 = axes[0, 1]
        for i, (name, history) in
enumerate(self.history_dict.items()):
            epochs_range = range(1,
len(history.history['val_loss']) + 1)
            ax2.plot(epochs_range,
history.history['val_loss'],
                    label=name, color=colors[i],
linewidth=2, marker='s', markersize=4)

        ax2.set_title('Validation Loss Comparison')
        ax2.set_xlabel('Epochs')
        ax2.set_ylabel('Loss')
        ax2.legend()
        ax2.grid(True, alpha=0.3)
        ax2.set_yscale('log')

        # 3. Training Accuracy Comparison
        ax3 = axes[0, 2]
        for i, (name, history) in
enumerate(self.history_dict.items()):
```

```
            epochs_range = range(1,
len(history.history['accuracy']) + 1)
            ax3.plot(epochs_range,
history.history['accuracy'],
                    label=name, color=colors[i],
linewidth=2, marker='^', markersize=4)

        ax3.set_title('Training Accuracy
Comparison')
        ax3.set_xlabel('Epochs')
        ax3.set_ylabel('Accuracy')
        ax3.legend()
        ax3.grid(True, alpha=0.3)
        ax3.set_ylim(0.8, 1.0)

        # 4. Validation Accuracy Comparison
        ax4 = axes[1, 0]
        for i, (name, history) in
enumerate(self.history_dict.items()):
            epochs_range = range(1,
len(history.history['val_accuracy']) + 1)
            ax4.plot(epochs_range,
history.history['val_accuracy'],
                    label=name, color=colors[i],
linewidth=2, marker='d', markersize=4)

        ax4.set_title('Validation Accuracy
Comparison')
        ax4.set_xlabel('Epochs')
        ax4.set_ylabel('Accuracy')
        ax4.legend()
        ax4.grid(True, alpha=0.3)
        ax4.set_ylim(0.8, 1.0)

        # 5. Final Performance Bar Chart
        ax5 = axes[1, 1]
        optimizer_names = list(self.results.keys())
        test_accuracies =
[self.results[name]['test_accuracy'] for name in
optimizer_names]

        bars = ax5.bar(optimizer_names,
test_accuracies, color=colors, alpha=0.7,
edgecolor='black')
        ax5.set_title('Final Test Accuracy
Comparison')
        ax5.set_ylabel('Test Accuracy')
        ax5.set_ylim(0.95, max(test_accuracies) +
0.005)

        # Add value labels on bars
        for bar, acc in zip(bars, test_accuracies):
```

```
            ax5.text(bar.get_x() +
bar.get_width()/2, bar.get_height() + 0.001,
                    f'{acc:.4f}', ha='center',
va='bottom', fontweight='bold')

        # Rotate x-axis labels for better
readability
        ax5.tick_params(axis='x', rotation=15)

        # 6. Convergence Rate Analysis (Loss
Reduction per Epoch)
        ax6 = axes[1, 2]
        for i, (name, history) in
enumerate(self.history_dict.items()):
            # Calculate loss reduction rate
(initial loss - final loss) / epochs
            initial_loss =
history.history['val_loss'][0]
            final_loss =
min(history.history['val_loss'])
            epochs_trained =
len(history.history['val_loss'])
            convergence_rate = (initial_loss -
final_loss) / epochs_trained

            ax6.bar(name, convergence_rate,
color=colors[i], alpha=0.7, edgecolor='black')

        ax6.set_title('Convergence Rate\n(Loss
Reduction per Epoch)')
        ax6.set_ylabel('Loss Reduction Rate')
        ax6.tick_params(axis='x', rotation=15)

        # Add value labels
        for i, name in enumerate(optimizer_names):
            initial_loss =
self.history_dict[name].history['val_loss'][0]
            final_loss =
min(self.history_dict[name].history['val_loss'])
            epochs_trained =
len(self.history_dict[name].history['val_loss'])
            convergence_rate = (initial_loss -
final_loss) / epochs_trained

            ax6.text(i, convergence_rate + max([

(self.history_dict[n].history['val_loss'][0] -
min(self.history_dict[n].history['val_loss'])) /
len(self.history_dict[n].history['val_loss'])
                for n in optimizer_names
            ]) * 0.05, f'{convergence_rate:.4f}',
ha='center', va='bottom', fontweight='bold')
```

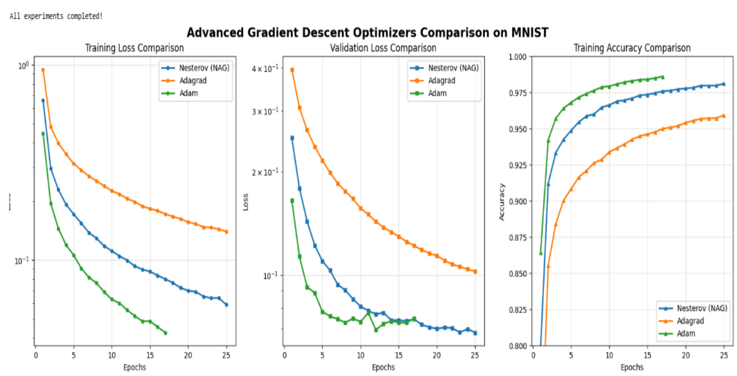
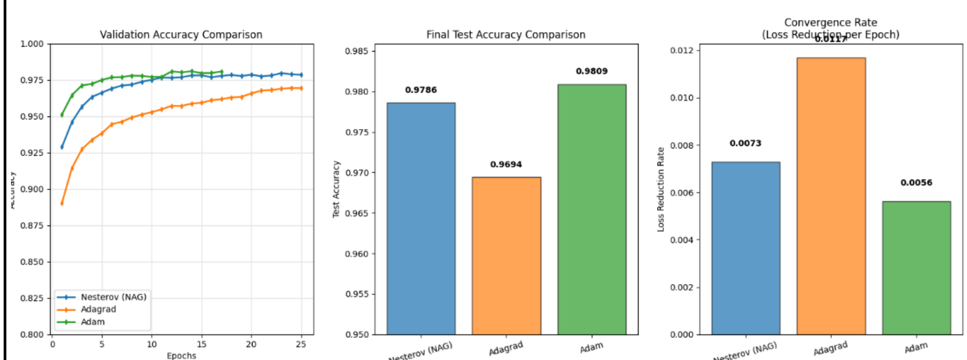
```
        plt.tight_layout()
        plt.show()

    def print_detailed_summary(self):
        """Print comprehensive analysis summary"""
        print("\n" + "="*80)
        print("ADVANCED GRADIENT DESCENT OPTIMIZERS
ANALYSIS SUMMARY")
        print("="*80)

        # Create summary DataFrame
        summary_data = []
        for name in self.results.keys():
            history = self.history_dict[name]
            summary_data.append({
                'Optimizer': name,
                'Test_Accuracy':
self.results[name]['test_accuracy'],
                'Train_Accuracy':
self.results[name]['train_accuracy'],
                'Test_Loss':
self.results[name]['test_loss'],
                'Epochs_Trained':
self.results[name]['epochs_trained'],
                'Final_Val_Loss':
min(history.history['val_loss']),
                'Best_Val_Acc':
max(history.history['val_accuracy']),
                'Convergence_Rate':
(history.history['val_loss'][0] -
min(history.history['val_loss'])) /
len(history.history['val_loss'])
            })

        summary_df =
pd.DataFrame(summary_data).sort_values('Test_Accura
cy', ascending=False)

        print(f"{'Rank':<5} {'Optimizer':<15}
{'Test Acc':<10} {'Train Acc':<11} {'Test
Loss':<10} {'Epochs':<8} {'Conv Rate':<10}")
        print("-" * 85)

        for idx, row in summary_df.iterrows():
            rank = summary_df.index.get_loc(idx) +
1
            print(f"{rank:<5}
{row['Optimizer']:<15}
{row['Test_Accuracy']:<10.4f}
{row['Train_Accuracy']:<11.4f}
{row['Test_Loss']:<10.4f}
```

```
{row['Epochs_Trained']:<8}
{row['Convergence_Rate']:<10.4f}")

        # Best performer analysis
        best_optimizer =
summary_df.iloc[0]['Optimizer']
        print(f"\n🏆 BEST PERFORMER:
{best_optimizer}")
        print(f"   Test Accuracy:
{summary_df.iloc[0]['Test_Accuracy']:.4f}")
        print(f"   Training Accuracy:
{summary_df.iloc[0]['Train_Accuracy']:.4f}")
        print(f"   Test Loss:
{summary_df.iloc[0]['Test_Loss']:.4f}")

        # Detailed insights
        print(f"\n📊 DETAILED INSIGHTS:")

        # Best accuracy
        best_acc_optimizer =
summary_df.loc[summary_df['Test_Accuracy'].idxmax()
, 'Optimizer']
        best_acc =
summary_df['Test_Accuracy'].max()
        print(f"   • Highest Test Accuracy:
{best_acc_optimizer} ({best_acc:.4f})")

        # Fastest convergence
        fastest_conv_optimizer =
summary_df.loc[summary_df['Convergence_Rate'].idxma
x(), 'Optimizer']
        fastest_conv_rate =
summary_df['Convergence_Rate'].max()
        print(f"   • Fastest Convergence:
{fastest_conv_optimizer} ({fastest_conv_rate:.4f}
loss reduction/epoch)")

        # Least epochs needed
        min_epochs_optimizer =
summary_df.loc[summary_df['Epochs_Trained'].idxmin(
), 'Optimizer']
        min_epochs =
summary_df['Epochs_Trained'].min()
        print(f"   • Fewest Epochs Needed:
{min_epochs_optimizer} ({min_epochs} epochs)")

        # Overfitting analysis
        print(f"\n🔍 OVERFITTING ANALYSIS:")
        for name in self.results.keys():
            overfitting =
self.results[name]['train_accuracy'] -
self.results[name]['test_accuracy']
            status = "✅ Good generalization" if
```

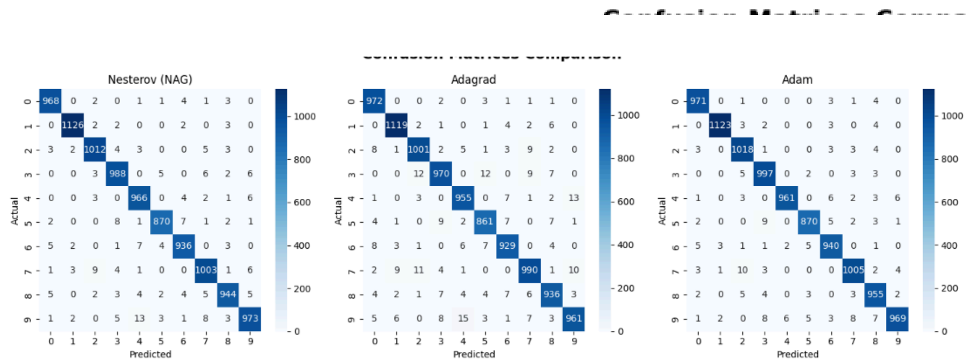
```
overfitting < 0.02 else "⚠️  Potential overfitting"
if overfitting < 0.05 else "❌ Overfitting
detected"
            print(f"    • {name}: {overfitting:.4f}
difference - {status}")

        print(f"\n💡 OPTIMIZER CHARACTERISTICS:")
        print(f"    • Nesterov (NAG): Uses momentum
with look-ahead, good for non-convex optimization")
        print(f"    • Adagrad: Adapts learning rate
per parameter, good for sparse data")
        print(f"    • Adam: Combines momentum and
adaptive learning rates, generally robust")

    def plot_confusion_matrices(self, figsize=(15,
5)):
        """Plot confusion matrices for all
optimizers"""
        fig, axes = plt.subplots(1, 3,
figsize=figsize)
        fig.suptitle('Confusion Matrices
Comparison', fontsize=16, fontweight='bold')

        for i, (name, model) in
enumerate(self.models.items()):
            # Make predictions
            y_pred = model.predict(self.x_test,
verbose=0)
            y_pred_classes = np.argmax(y_pred,
axis=1)

            # Create confusion matrix
            cm = confusion_matrix(self.y_test,
y_pred_classes)

            # Plot
            sns.heatmap(cm, annot=True, fmt='d',
cmap='Blues', ax=axes[i])
            axes[i].set_title(f'{name}')
            axes[i].set_xlabel('Predicted')
            axes[i].set_ylabel('Actual')

        plt.tight_layout()
        plt.show()

def main():
    """Main execution function"""

    # Create comparator instance
    comparator = OptimizerComparator()

    # Run all experiments
```

```
    comparator.run_all_experiments(epochs=25,
batch_size=128)

    # Display comprehensive results
    comparator.plot_comprehensive_results()

    # Print detailed summary
    comparator.print_detailed_summary()

    # Show confusion matrices
    comparator.plot_confusion_matrices()

    return comparator

# Additional utility functions for deeper analysis
def analyze_learning_curves(comparator):
    """Analyze learning curves in detail"""
    print("\n📈 LEARNING CURVES ANALYSIS:")

    for name, history in
comparator.history_dict.items():
        train_acc = history.history['accuracy']
        val_acc = history.history['val_accuracy']

        # Find epoch where validation accuracy
plateaus
        val_acc_diff = np.diff(val_acc)
        plateau_epoch = None
        for i in range(len(val_acc_diff) - 2):
            if all(abs(val_acc_diff[i:i+3]) <
0.001):  # Very small changes for 3 consecutive
epochs
                plateau_epoch = i + 1
                break

        print(f"\n   {name}:")
        print(f"   • Initial accuracy:
{train_acc[0]:.4f} (train), {val_acc[0]:.4f}
(val)")
        print(f"   • Final accuracy:
{train_acc[-1]:.4f} (train), {val_acc[-1]:.4f}
(val)")
        print(f"   • Best validation accuracy:
{max(val_acc):.4f} at epoch {np.argmax(val_acc) +
1}")
        if plateau_epoch:
            print(f"   • Validation accuracy
plateaued around epoch {plateau_epoch}")

def compare_computational_efficiency(comparator):
    """Compare computational aspects of
optimizers"""
```

```
    print("\n⚡ COMPUTATIONAL EFFICIENCY:")
    print(f"   • Nesterov (NAG): Low memory,
moderate computation (momentum)")
    print(f"   • Adagrad: Higher memory (stores
squared gradients), adaptive computation")
    print(f"   • Adam: Highest memory (stores 1st &
2nd moments), most computation")

    for name in comparator.results.keys():
        epochs =
comparator.results[name]['epochs_trained']
        print(f"   • {name}: Completed in {epochs}
epochs")

# Run the complete experiment
if __name__ == "__main__":
    print("Starting Advanced Gradient Descent
Optimizers Comparison...\n")

    # Run main experiment
    comparator = main()

    # Additional analyses
    analyze_learning_curves(comparator)
    compare_computational_efficiency(comparator)

    print(f"\n🎯 EXPERIMENT COMPLETED
SUCCESSFULLY!")
    print(f"   All three optimizers have been
compared on MNIST digit classification")
    print(f"   Check the plots and summary above
for detailed insights")
```

	[redacted]
Output	# Output For Sigmoid Activation Function  

```
================================================================================
ADVANCED GRADIENT DESCENT OPTIMIZERS ANALYSIS SUMMARY
================================================================================
Rank  Optimizer       Test Acc   Train Acc   Test Loss  Epochs   Conv Rate
-------------------------------------------------------------------------------------
1     Adam            0.9809     0.9937      0.0694     17       0.0056
2     Nesterov (NAG)  0.9786     0.9936      0.0681     25       0.0073
3     Adagrad         0.9694     0.9748      0.1026     25       0.0117

🏆 BEST PERFORMER: Adam
   Test Accuracy: 0.9809
   Training Accuracy: 0.9937
   Test Loss: 0.0694

📊 DETAILED INSIGHTS:
   • Highest Test Accuracy: Adam (0.9809)
   • Fastest Convergence: Adagrad (0.0117 loss reduction/epoch)
   • Fewest Epochs Needed: Adam (17 epochs)

🔍 OVERFITTING ANALYSIS:
   • Nesterov (NAG): 0.0150 difference - ✅ Good generalization
   • Adagrad: 0.0054 difference - ✅ Good generalization
   • Adam: 0.0128 difference - ✅ Good generalization

💡 OPTIMIZER CHARACTERISTICS:
   • Nesterov (NAG): Uses momentum with look-ahead, good for non-convex optimization
   • Adagrad: Adapts learning rate per parameter, good for sparse data
   • Adam: Combines momentum and adaptive learning rates, generally robust
```




```
📈 LEARNING CURVES ANALYSIS:

   Nesterov (NAG):
   • Initial accuracy: 0.7979 (train), 0.9289 (val)
   • Final accuracy: 0.9812 (train), 0.9786 (val)
   • Best validation accuracy: 0.9797 at epoch 23
   • Validation accuracy plateaued around epoch 16

   Adagrad:
   • Initial accuracy: 0.7102 (train), 0.8904 (val)
   • Final accuracy: 0.9594 (train), 0.9694 (val)
   • Best validation accuracy: 0.9694 at epoch 24
   • Validation accuracy plateaued around epoch 21

   Adam:
   • Initial accuracy: 0.8642 (train), 0.9513 (val)
   • Final accuracy: 0.9862 (train), 0.9809 (val)
   • Best validation accuracy: 0.9810 at epoch 14
   • Validation accuracy plateaued around epoch 6

⚡ COMPUTATIONAL EFFICIENCY:
   • Nesterov (NAG): Low memory, moderate computation (momentum)
   • Adagrad: Higher memory (stores squared gradients), adaptive computation
   • Adam: Highest memory (stores 1st & 2nd moments), most computation
   • Nesterov (NAG): Completed in 25 epochs
   • Adagrad: Completed in 25 epochs
   • Adam: Completed in 17 epochs

🎯 EXPERIMENT COMPLETED SUCCESSFULLY!
   All three optimizers have been compared on MNIST digit classification
   Check the plots and summary above for detailed insights
```

Conclusion	In our analysis, we utilized a **House Price Prediction Dataset** obtained from Kaggle. We tested two activation functions: **Sigmoid** and **ReLU**. The results revealed that, for both activation functions, **Momentum SGD** outperformed **SGD** in terms of accuracy. However, **SGD** demonstrated faster training times compared to **Momentum SGD**.