```python
import numpy as np
import tensorflow as tf
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import train_test_split
from tensorflow.keras.callbacks import EarlyStopping
import matplotlib.pyplot as plt

class XenopoulosQuantumDialectics:
    """
    Υλοποίηση των τύπων N[Fi(Gj)] και N[E1(G1)] για κβαντικές εφαρμογές.
    Βασίζεται στο έργο του Επαμεινώνδα Ξενόπουλου (1920-1994).
    """

    def __init__(self, qubit_type='superconducting'):
        self.scaler = MinMaxScaler()
        self.qubit_type = qubit_type
        self.model = self._build_model()

    def _build_model(self):
        """Κατασκευή νευρωνικού δικτύου με διαλεκτική αρχιτεκτονική"""
        model = tf.keras.Sequential([
            tf.keras.layers.Dense(128, activation='relu', input_shape=(4,)),
            tf.keras.layers.Dropout(0.3),
            tf.keras.layers.Dense(64, activation='relu'),
            tf.keras.layers.Dense(1, activation='linear')
        ])

        model.compile(
            optimizer=tf.keras.optimizers.Adam(learning_rate=0.001),
```

```
        loss='mse',
        metrics=['mae']
    )
    return model

def _N_Fi_Gj(self, Fi, Gj):
    """Διαλεκτικός τύπος αλληλεπίδρασης συστήματος-περιβάλλοντος"""
    if self.qubit_type == 'photonic':
        return Fi * (1 - Gj**2) + 0.05 * np.exp(-3 * Gj)
    elif self.qubit_type == 'topological':
        return Fi * (1 - np.sqrt(Gj)) + 0.01 * np.log(1 + Fi)
    else:  # superconducting
        return Fi * (1 - Gj) + 0.1 * np.exp(-5 * Gj)

def load_data(self):
    """Φόρτωμα δεδομένων ανά τύπο qubit"""
    if self.qubit_type == 'photonic':
        return {
            "PhotonLifetime": [100, 150, 200],
            "PhotonLoss": [0.2, 0.15, 0.1],
            "ErrorRate": [1.2, 0.7, 0.3]
        }
    elif self.qubit_type == 'topological':
        return {
            "CoherenceTime": [10000, 15000, 20000],
            "TopologicalNoise": [0.2, 0.1, 0.05],
            "ErrorRate": [0.01, 0.008, 0.005]
        }
    else:  # superconducting
        return {
            "CoherenceTime": [50, 90, 150],
```

```
            "Noise": [8.0, 4.0, 1.0],
            "ErrorRate": [1.5, 0.8, 0.2]
        }

    def prepare_data(self):
        """Προετοιμασία και κανονικοποίηση δεδομένων"""
        raw_data = self.load_data()
        scaled_data = self.scaler.fit_transform(np.array(list(raw_data.values())).T)

        Fi = scaled_data[:, 0]
        Gj = scaled_data[:, 1]
        N = self._N_Fi_Gj(Fi, Gj)

        X = np.column_stack((Fi, Gj, N, scaled_data[:, 2]))  # Συμπεριλαμβάνει επιπλέον μεταβλητές
        y = scaled_data[:, 2]

        return X, y

    def train(self, epochs=500, validation_split=0.2):
        """Εκπαίδευση μοντέλου με early stopping"""
        X, y = self.prepare_data()
        early_stop = EarlyStopping(monitor='val_loss', patience=20, restore_best_weights=True)

        history = self.model.fit(
            X, y,
            epochs=epochs,
            validation_split=validation_split,
            callbacks=[early_stop],
            verbose=0
        )
```

```
        return history

    def predict_error_rate(self, future_conditions):
        """Πρόβλεψη σφαλμάτων για νέες συνθήκες"""
        scaled_input = self.scaler.transform([future_conditions])
        Fi = scaled_input[0,0]
        Gj = scaled_input[0,1]
        N = self._N_Fi_Gj(Fi, Gj)

        prediction = self.model.predict([[Fi, Gj, N, scaled_input[0,2]]])
        return self.scaler.inverse_transform(
            np.column_stack((scaled_input, prediction))
        )[0][-1]

    def visualize_results(self, historical_data, predictions):
        """Οπτικοποίηση ιστορικών και προβλεπόμενων τιμών"""
        plt.figure(figsize=(12,6))

        if self.qubit_type == 'photonic':
            x_axis = historical_data["PhotonLifetime"]
            x_label = "Χρόνος Ζωής Φωτονίου (ns)"
        else:
            x_axis = historical_data["CoherenceTime"]
            x_label = "Χρόνος Συνόχης (μs)"

        plt.plot(x_axis, historical_data["ErrorRate"], 'o-', label="Πραγματικά Δεδομένα")
        plt.plot(predictions['future_condition'], predictions['error_rate'], 's--',
label="Πρόβλεψη")

        plt.title(f"Απόδοση {self.qubit_type.capitalize()} Qubits")
        plt.xlabel(x_label)
```

```
        plt.ylabel("Ποσοστό Σφάλματος (%)")
        plt.legend()
        plt.grid(True)
        plt.show()

# Παράδειγμα Χρήσης για Υπεραγώγιμα Qubits
if __name__ == "__main__":
    simulator = XenopoulosQuantumDialectics(qubit_type='superconducting')
    history = simulator.train()

    # Πρόβλεψη για νέες συνθήκες (CoherenceTime=180μs, Noise=0.5dB)
    prediction = simulator.predict_error_rate([180, 0.5, 0.15])  # Τρίτη παράμετρος ανάλογα με τύπο qubit
    print(f"Προβλεπόμενο Σφάλμα: {prediction:.2f}%")

    # Οπτικοποίηση
    simulator.visualize_results(
        simulator.load_data(),
        {'future_condition': 180, 'error_rate': prediction}
    )
```

Βασικά Σημεία Κώδικα:

1. **Καθολική Κλάση**: Ολοκληρωμένη κλάση `XenopoulosQuantumDialectics` που καλύπτει όλες τις λειτουργίες
2. **Υποστήριξη Πολλαπλών Τύπων Qubits**:
   - Υπεραγώγιμα (superconducting)
   - Φωτονικά (photonic)
   - Τοπολογικά (topological)
3. **Βελτιστοποιημένες Συναρτήσεις**:

- Διαφορετικοί τύποι N[Fi(Gj)] ανά τύπο qubit
- Αυτόματη κανονικοποίηση δεδομένων

4. **Προηγμένες Τεχνικές**:
   - Early stopping
   - Dropout layers για αποφυγή overfitting
   - Δυναμική οπτικοποίηση