Pseudo Code for GARCH CNN

This is pseudo code on how to get GARCH parameters, train a model with option prices with the historical asset prices, and then calibrate the parameters.

Step 1: Data Collection (Option Prices)

1. Historical Asset Prices: This is required to estimate the initial GARCH parameters.
2. Historical Option Prices: These are used to train the ANN and ultimately calibrate the GARCH parameters.

Step 2: Initial GARCH Model Estimation

Estimate the initial parameters of the GARCH model using historical asset price data:

```
import pandas as pd
from arch import arch_model

# Load historical asset prices
asset_prices = pd.read_csv('asset_prices.csv')
returns = 100 * asset_prices['Close'].pct_change().dropna()

# Estimate initial GARCH parameters
garch_model = arch_model(returns, vol='Garch', p=1, q=1)
garch_fit = garch_model.fit()
initial_params = garch_fit.params
```

Step 3: Create Feature Set for ANN

Using the initial GARCH model, simulate the volatility and prepare the dataset for ANN training.

```
import numpy as np

# Load historical option prices
option_prices = pd.read_csv('option_prices.csv')

# Simulate volatility using initial GARCH parameters
simulated_volatility = garch_fit.conditional_volatility

# Create input features
features = pd.DataFrame({
    'volatility': simulated_volatility[-len(option_prices):],  # Ensure the lengths
match
    'strike': option_prices['Strike'],
    'maturity': option_prices['Maturity']
})

# Target variable is the observed option prices
targets = option_prices['ObservedPrice']
```

Step 4: Design and Train ANN

Design an ANN to learn the relationship between the simulated volatilities (from the GARCH model) and the observed option prices.

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

# Define the ANN architecture
model = Sequential()
model.add(Dense(64, input_dim=3, activation='relu'))
```

```
model.add(Dense(64, activation='relu'))
model.add(Dense(1))

# Compile the model
model.compile(optimizer='adam', loss='mean_squared_error')

# Train the model
model.fit(features, targets, epochs=100, batch_size=32, validation_split=0.2)`
```

Step 5: Optimize GARCH Parameters

Use the trained ANN to optimize the GARCH parameters by minimizing the error between the ANN-predicted option prices and the observed option prices.

```
from scipy.optimize import minimize

# Define a function to update GARCH parameters and recalculate simulated volatility
def update_garch_parameters(params):
    garch_model = arch_model(returns, vol='Garch', p=1, q=1)
    garch_fit = garch_model.fit(update_freq=0, starting_values=params)
    return garch_fit.conditional_volatility

# Objective function for optimization
def objective(params):
    simulated_volatility = update_garch_parameters(params)
    features['volatility'] = simulated_volatility[-len(option_prices):]
    predicted_prices = model.predict(features)
    return np.mean((predicted_prices - targets) ** 2)

# Use scipy.optimize to minimize the objective function
result = minimize(objective, initial_params)
optimized_params = result.x
```

Step 6: Validation

Validate the calibrated model by comparing its predictions to actual market data.

```
# Assuming you have a validation set of option prices
validation_option_prices = pd.read_csv('validation_option_prices.csv')

# Simulate volatility using the optimized GARCH parameters
optimized_volatility = update_garch_parameters(optimized_params)

# Create validation features
validation_features = pd.DataFrame({
    'volatility': optimized_volatility[-len(validation_option_prices):],
    'strike': validation_option_prices['Strike'],
    'maturity': validation_option_prices['Maturity']
})

# Predict option prices using the trained ANN
validation_predictions = model.predict(validation_features)

# Compare predicted prices to actual prices
validation_targets = validation_option_prices['ObservedPrice']
validation_error = np.mean((validation_predictions - validation_targets) ** 2)

print(f'Validation Error: {validation_error}')
```

Generating Simulated Data (European Options)

```
import numpy as np
import pandas as pd
from arch import arch_model

# Set random seed for reproducibility
np.random.seed(42)

# Define GARCH model parameters
garch_params = {
    'omega': 0.1,
    'alpha': 0.05,
    'beta': 0.9
}

# Generate GARCH(1,1) process
n = 1000  # Number of data points
garch_model = arch_model(None, vol='Garch', p=1, q=1, mean='Zero', dist='Normal')
simulated_data = garch_model.simulate([garch_params['omega'], garch_params['alpha'],
garch_params['beta']], n)

# Extract simulated returns and conditional volatility
simulated_returns = simulated_data['data']
simulated_volatility = simulated_data['volatility']

from scipy.stats import norm

# Black-Scholes option pricing formula
def black_scholes_price(S, K, T, r, sigma, option_type='call'):
    d1 = (np.log(S / K) + (r + 0.5 * sigma ** 2) * T) / (sigma * np.sqrt(T))
    d2 = d1 - sigma * np.sqrt(T)

    if option_type == 'call':
        price = S * norm.cdf(d1) - K * np.exp(-r * T) * norm.cdf(d2)
    elif option_type == 'put':
        price = K * np.exp(-r * T) * norm.cdf(-d2) - S * norm.cdf(-d1)

    return price

# Generate option prices
strike_prices = np.linspace(80, 120, 10)
maturities = np.linspace(0.1, 1, 10)
risk_free_rate = 0.01

option_prices = []
for i in range(n):
    for K in strike_prices:
        for T in maturities:
            S = 100  # Assume the underlying asset price is 100
            sigma = simulated_volatility[i]
            price = black_scholes_price(S, K, T, risk_free_rate, sigma)
            option_prices.append((S, K, T, sigma, price))

# Convert to DataFrame
option_prices_df = pd.DataFrame(option_prices, columns=['S', 'K', 'T', 'sigma',
'price'])
```

```
# Prepare features and target
features = option_prices_df[['sigma', 'K', 'T']]
target = option_prices_df['price']

# Split into training and validation sets
from sklearn.model_selection import train_test_split

features_train, features_val, target_train, target_val = train_test_split(features,
target, test_size=0.2, random_state=42)

...
```