Exponential Ornstein Uhlenbeck model: dataset and learning

Benjamin COHEN - Julien MASSIP

Reference model

$$dX_t = X_t e^{Y_t} dW_t^1$$
$$dY_t = \alpha(m - Y_t)dt + \beta dW_t^2$$
$$d\langle W^1, W^2\rangle_t = \rho dt$$

Payoff : $(X_T - K)_+$

How to create a neural network to price calls in this model?

Summary

I/ GPU generation of reference prices

a- Algorithmic structure

b- Implementation details

II/ Learning by neural networks

I/ a) Algorithmic structure

To make a neural network, we need a lot of data.

In addition, data with a wide parameter panel.

Classic method: Monte Carlo + Euler scheme → Long calculation times.

Solution: Use GPU parallel computing.

Each GPU thread will do a Monte Carlo with a unique combination of parameters.

The data with parameters and prices are then stored in .csv

I/ a) Algorithmic structure

List of parameters to distribute:

Strike: 4 values.

m: 10 values.

alpha: 10 values.

bare (alpha and beta binder): 10 values.

rho: 10 values.

Y0: 10 values.

4 x 10 x 10 x 10 x 10 x 10 = 625 x 640 total threads.

625 blocks each with 640 threads.

I/ a) Algorithmic structure

How to distribute parameters to different threads?

Modulo possible, but expensive.

We must keep a “proximity” between the threads to optimize.

Idea to implement:

thread id: 0 1 2 3 4 5 6 ……………..…………………… N-2 N-1 (initial identifier)

id strike: 0 0 0 0 0 0 0 ……. 0 1 1 1 1 ……3…… 3 3 (identifier linked to strikes)

pack of size N / 4

nv id: 0 1 2 3 4 5 6 …… N/4-1 0 1 2……0…. N/4-2 N/4-1

(initial id by packs)

I/ a) Algorithmic structure

Implementation read parameter in GPU:

```
int pidx, same;
int idx = blockDim.x * blockIdx.x + threadIdx.x;   (id thread)
same = idx;
// StrR
pidx = idx*4/(640*625);   division entière ⇒ (id strike)
StrR = Strd[pidx];
// mR
same -= (pidx*640*625/4);   modulo implicite ⇒ (nv id)
pidx = same*4*10/(640*625);
mR = md[pidx];
```

ATTENTION: The same order must be used to read the parameters on the CPU!

I/ b) Implementation details

The Y0 parameter must be the last loaded so as not to distort the values:

```
same -= (pidx*640*625/(4*10*10*10*10));
pidx = same*4*10*10*10*10*10/(640*625);

for (int i = 0; i < Ntraj; i++) {    // Monte Carlo loop
    // Initialize Y
    float YR = Y0d[pidx];
```

Square roots are expensive, so it is preferable to think in square root time steps, even if it means calculating less expensive squares:

```
float dt = sqrtf(1.0f/(64.0f*12.0f));

int N = T/(dt*dt);  // nb step for Euler

// Euler schema
X = X + X*expf(YR)*dt*G.x;
YR = YR + alphaR*(mR - YR)*dt*dt + betaR*dt*B;
```

II/ Learning by neural network

Generate datasets with the previous step at different maturities and strikes then vary the other parameters.

Implementation under Pytorch and use of the GPU.

We limit ourselves to prices lower than 100, which already gives us 5M elements.

Goal: Find the prices previously calculated in Monte Carlo only from the parameters.

II/ a) Pre-processing and initialization

```
# Prepare the data
X = df[['alpha', 'beta', 'm', 'rho', 'Y0', 'Maturity', 'Strike']].values
y = df['price'].values

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

```
# Define RMSE loss function
def rmse_loss(y_true, y_pred):
    return torch.sqrt(torch.mean((y_true - y_pred)**2))

# Define batch size
batch_size = 32768

# Standardize the data
scaler_X = StandardScaler()
X_train_scaled = scaler_X.fit_transform(X_train)
X_test_scaled = scaler_X.transform(X_test)

scaler_y = StandardScaler()
y_train_scaled = scaler_y.fit_transform(y_train.reshape(-1, 1)).flatten()  # Scale the target variable
y_test_scaled = scaler_y.transform(y_test.reshape(-1, 1)).flatten()  # Scale the target variable

# Convert data to PyTorch tensors
X_train_tensor = torch.tensor(X_train_scaled, dtype=torch.float32)
X_test_tensor = torch.tensor(X_test_scaled, dtype=torch.float32)

y_train_tensor = torch.tensor(y_train_scaled, dtype=torch.float32).view(-1, 1)
y_test_tensor = torch.tensor(y_test_scaled, dtype=torch.float32).view(-1, 1)

# Create DataLoader for training and testing sets
train_dataset = TensorDataset(X_train_tensor, y_train_tensor)
train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)

test_dataset = TensorDataset(X_test_tensor, y_test_tensor)
test_loader = DataLoader(test_dataset, batch_size=batch_size)
```

II/ b) Neural Network Structure

```
# Define the neural network architecture
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(7, 256)  # Increase neurons
        self.bn1 = nn.BatchNorm1d(256)  # BatchNorm layer
        self.fc2 = nn.Linear(256, 512)  # Additional layer
        self.bn2 = nn.BatchNorm1d(512)  # BatchNorm layer
        self.fc3 = nn.Linear(512, 1024)   # Additional layer
        self.bn3 = nn.BatchNorm1d(1024)  # BatchNorm layer
        self.fc4 = nn.Linear(1024, 2048)   # Additional layer
        self.bn4 = nn.BatchNorm1d(2048)  # BatchNorm layer
        self.fc5 = nn.Linear(2048, 1024)   # Additional layer
        self.bn5 = nn.BatchNorm1d(1024)  # BatchNorm layer
        self.fc6 = nn.Linear(1024, 512)   # Additional layer
        self.bn6 = nn.BatchNorm1d(512)  # BatchNorm layer
        self.fc7 = nn.Linear(512, 256)   # Additional layer
        self.bn7 = nn.BatchNorm1d(256)  # BatchNorm layer
        self.fc8 = nn.Linear(256, 1)
        self.relu = nn.ReLU()
        self.dropout1 = nn.Dropout(0.1)
        self.dropout2 = nn.Dropout(0.4)
        self.dropout3 = nn.Dropout(0.6)

    def forward(self, x):
        x = self.relu(self.bn1(self.fc1(x)))
        x = self.dropout1(self.relu(self.bn2(self.fc2(x))))
        x = self.dropout2(self.relu(self.bn3(self.fc3(x))))
        x = self.dropout3(self.relu(self.bn4(self.fc4(x))))
        x = self.dropout3(self.relu(self.bn5(self.fc5(x))))
        x = self.dropout2(self.relu(self.bn6(self.fc6(x))))
        x = self.dropout1(self.relu(self.bn7(self.fc7(x))))
        x = self.fc8(x)
        return x
```

II/ c) Optimizer, LR Scheduler, Early Stopping, Warmup

```
# Instantiate the model and move it to GPU
model = Net().cuda()

# Define the loss function and optimizer
criterion = rmse_loss
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)

# Define the learning rate scheduler
scheduler = ReduceLROnPlateau(optimizer, 'min', patience=4, factor=0.1, verbose=True)

# Define early stopping parameters
patience = 10
best_loss = float('inf')
counter = 0

# Warm-up phase
warmup_epochs = 2
for epoch in range(warmup_epochs):
    model.train()
    for inputs, labels in train_loader:
        inputs, labels = inputs.cuda(), labels.cuda()
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
```

II/ d) Training

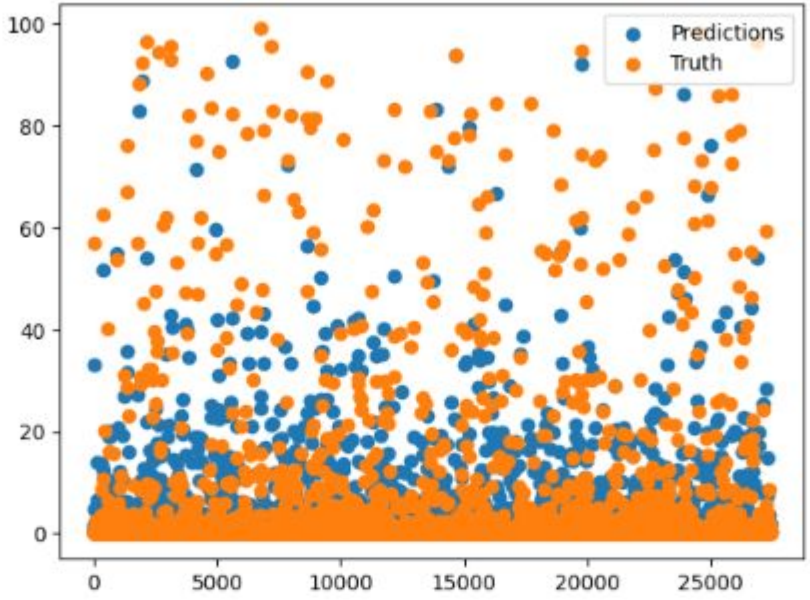
```
# Train the neural network
num_epochs = 50
for epoch in range(num_epochs):
    model.train()  # Set model to training mode
    running_loss = 0.0
    for inputs, labels in train_loader:
        inputs, labels = inputs.cuda(), labels.cuda()
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        running_loss += loss.item() * inputs.size(0)
    epoch_loss = running_loss / len(train_dataset)

    # Validation
    model.eval()  # Set model to evaluation mode
    with torch.no_grad():
        running_val_loss = 0.0
        for inputs, labels in test_loader:
            inputs, labels = inputs.cuda(), labels.cuda()
            outputs = model(inputs)
            val_loss = criterion(outputs, labels)
            running_val_loss += val_loss.item() * inputs.size(0)
        epoch_val_loss = running_val_loss / len(test_dataset)

    scheduler.step(epoch_val_loss)

    if epoch_val_loss < best_loss:
        best_loss = epoch_val_loss
        counter = 0
        # Save the best model
        torch.save(model.state_dict(), 'best_model.pth')
    else:
        counter += 1
        if counter >= patience:
            print("Early stopping triggered.")
            break

    if (epoch+1) % 1 == 0:
        print(f'Epoch [{epoch+1}/{num_epochs}], Train Loss: {epoch_loss:.4f}, Val Loss: {epoch_val_loss:.4f}')

# Load the best model
model.load_state_dict(torch.load('best_model.pth'))
```

II/ e) Evaluation

```python
# Evaluate the best model on the test set
model.eval()
with torch.no_grad():
    running_loss = 0.0
    for inputs, labels in test_loader:
        inputs, labels = inputs.cuda(), labels.cuda()
        outputs = model(inputs)
        # Unscale predictions
        unscaled_outputs = scaler_y.inverse_transform(outputs.cpu().numpy()).flatten()
        unscaled_labels = scaler_y.inverse_transform(labels.cpu().numpy()).flatten()
        # Calculate RMSE loss on unscaled predictions and labels
        unscaled_loss = criterion(torch.tensor(unscaled_outputs), torch.tensor(unscaled_labels))
        running_loss += unscaled_loss.item() * inputs.size(0)
    unscaled_test_loss = running_loss / len(test_dataset)
    print(f'Unscaled Test Loss: {unscaled_test_loss:.4f}')
```

II/ f) Results

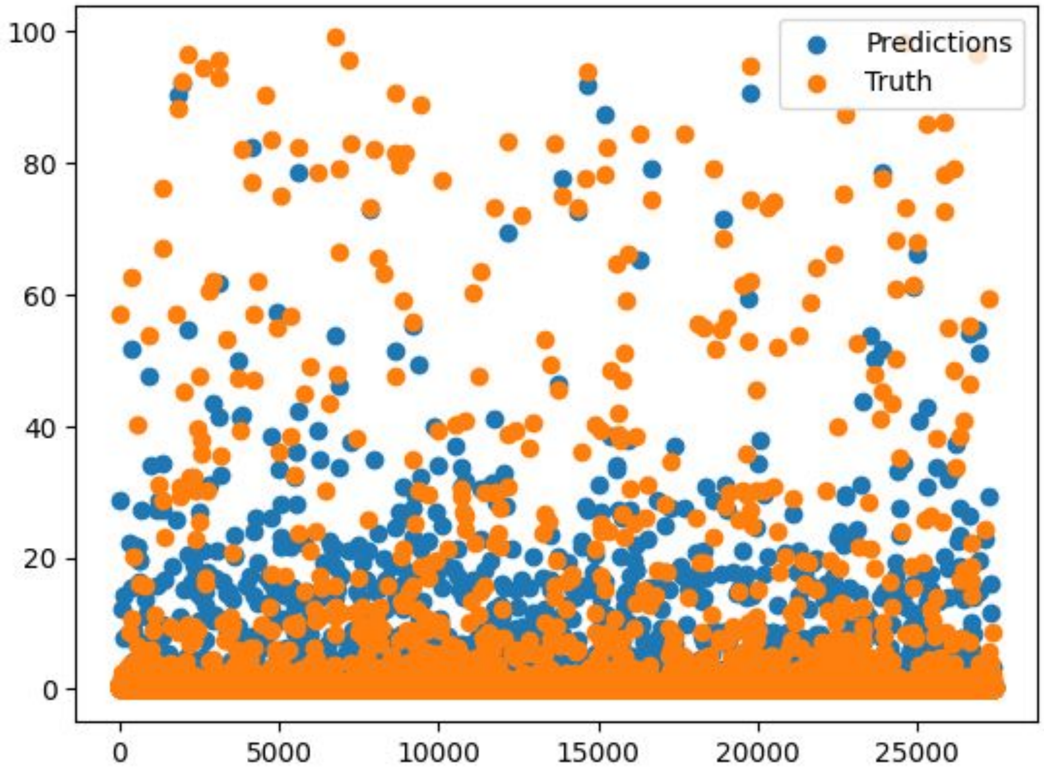
```
Epoch [1/50], Train Loss: 0.9029, Val Loss: 0.9034
Epoch [2/50], Train Loss: 0.8974, Val Loss: 0.8932
Epoch [3/50], Train Loss: 0.8950, Val Loss: 0.8877
Epoch [4/50], Train Loss: 0.8904, Val Loss: 0.8779
Epoch [5/50], Train Loss: 0.8862, Val Loss: 0.8734
Epoch [6/50], Train Loss: 0.8819, Val Loss: 0.8653
Epoch [7/50], Train Loss: 0.8762, Val Loss: 0.8625
Epoch [8/50], Train Loss: 0.8715, Val Loss: 0.8545
Epoch [9/50], Train Loss: 0.8661, Val Loss: 0.8467
Epoch [10/50], Train Loss: 0.8596, Val Loss: 0.8472
Epoch [11/50], Train Loss: 0.8546, Val Loss: 0.8428
Epoch [12/50], Train Loss: 0.8501, Val Loss: 0.8335
Epoch [13/50], Train Loss: 0.8470, Val Loss: 0.8233
Epoch [14/50], Train Loss: 0.8385, Val Loss: 0.8304
Epoch [15/50], Train Loss: 0.8347, Val Loss: 0.8115
Epoch [16/50], Train Loss: 0.8309, Val Loss: 0.8087
Epoch [17/50], Train Loss: 0.8242, Val Loss: 0.8012
Epoch [18/50], Train Loss: 0.8224, Val Loss: 0.7961
Epoch [19/50], Train Loss: 0.8140, Val Loss: 0.7951
Epoch [20/50], Train Loss: 0.8119, Val Loss: 0.7996
Epoch [21/50], Train Loss: 0.8080, Val Loss: 0.7832
Epoch [22/50], Train Loss: 0.8046, Val Loss: 0.7767
Epoch [23/50], Train Loss: 0.8005, Val Loss: 0.7708
Epoch [24/50], Train Loss: 0.7962, Val Loss: 0.7696
Epoch [25/50], Train Loss: 0.7952, Val Loss: 0.7632
Epoch [26/50], Train Loss: 0.7903, Val Loss: 0.7643
Epoch [27/50], Train Loss: 0.7879, Val Loss: 0.7570
Epoch [28/50], Train Loss: 0.7866, Val Loss: 0.7559
Epoch [29/50], Train Loss: 0.7830, Val Loss: 0.7597
Epoch [30/50], Train Loss: 0.7805, Val Loss: 0.7540
Epoch [31/50], Train Loss: 0.7759, Val Loss: 0.7516
Epoch [32/50], Train Loss: 0.7790, Val Loss: 0.7483
Epoch [33/50], Train Loss: 0.7730, Val Loss: 0.7511
Epoch [34/50], Train Loss: 0.7723, Val Loss: 0.7449
Epoch [35/50], Train Loss: 0.7685, Val Loss: 0.7374
Epoch [36/50], Train Loss: 0.7696, Val Loss: 0.7494
Epoch [37/50], Train Loss: 0.7661, Val Loss: 0.7371
Epoch [38/50], Train Loss: 0.7653, Val Loss: 0.7346
Epoch [39/50], Train Loss: 0.7647, Val Loss: 0.7361
Epoch [40/50], Train Loss: 0.7632, Val Loss: 0.7325
Epoch [41/50], Train Loss: 0.7610, Val Loss: 0.7326
Epoch [42/50], Train Loss: 0.7590, Val Loss: 0.7350
Epoch [43/50], Train Loss: 0.7571, Val Loss: 0.7319
Epoch [44/50], Train Loss: 0.7578, Val Loss: 0.7258
Epoch [45/50], Train Loss: 0.7563, Val Loss: 0.7275
Epoch [46/50], Train Loss: 0.7541, Val Loss: 0.7230
Epoch [47/50], Train Loss: 0.7536, Val Loss: 0.7299
Epoch [48/50], Train Loss: 0.7544, Val Loss: 0.7271
Epoch [49/50], Train Loss: 0.7513, Val Loss: 0.7243
Epoch [50/50], Train Loss: 0.7502, Val Loss: 0.7216
Unscaled Test Loss: 3.6760
```



II/ f) Results

```
Epoch [1/150], Train Loss: 0.9022, Val Loss: 0.8946
Epoch [2/150], Train Loss: 0.8994, Val Loss: 0.8991
Epoch [3/150], Train Loss: 0.8956, Val Loss: 0.8879
Epoch [4/150], Train Loss: 0.8916, Val Loss: 0.8824
Epoch [5/150], Train Loss: 0.8875, Val Loss: 0.8769
Epoch [6/150], Train Loss: 0.8838, Val Loss: 0.8871
Epoch [7/150], Train Loss: 0.8783, Val Loss: 0.8767
Epoch [8/150], Train Loss: 0.8753, Val Loss: 0.8631
Epoch [9/150], Train Loss: 0.8681, Val Loss: 0.8512
Epoch [10/150], Train Loss: 0.8635, Val Loss: 0.8441
Epoch [11/150], Train Loss: 0.8576, Val Loss: 0.8364
Epoch [12/150], Train Loss: 0.8530, Val Loss: 0.8352
Epoch [13/150], Train Loss: 0.8467, Val Loss: 0.8277
Epoch [14/150], Train Loss: 0.8426, Val Loss: 0.8144
Epoch [15/150], Train Loss: 0.8345, Val Loss: 0.8063
Epoch [16/150], Train Loss: 0.8309, Val Loss: 0.8053
Epoch [17/150], Train Loss: 0.8238, Val Loss: 0.7975
Epoch [18/150], Train Loss: 0.8234, Val Loss: 0.7907
Epoch [19/150], Train Loss: 0.8170, Val Loss: 0.7964
Epoch [20/150], Train Loss: 0.8127, Val Loss: 0.7846
Epoch [21/150], Train Loss: 0.8084, Val Loss: 0.7838
Epoch [22/150], Train Loss: 0.8033, Val Loss: 0.7812
Epoch [23/150], Train Loss: 0.8005, Val Loss: 0.7689
Epoch [24/150], Train Loss: 0.7968, Val Loss: 0.7693
Epoch [25/150], Train Loss: 0.7949, Val Loss: 0.7588
Epoch [26/150], Train Loss: 0.7890, Val Loss: 0.7601
Epoch [27/150], Train Loss: 0.7880, Val Loss: 0.7627
Epoch [28/150], Train Loss: 0.7850, Val Loss: 0.7540
Epoch [29/150], Train Loss: 0.7822, Val Loss: 0.7537
Epoch [30/150], Train Loss: 0.7797, Val Loss: 0.7527
Epoch [31/150], Train Loss: 0.7776, Val Loss: 0.7464
Epoch [32/150], Train Loss: 0.7749, Val Loss: 0.7488
Epoch [33/150], Train Loss: 0.7726, Val Loss: 0.7387
Epoch [34/150], Train Loss: 0.7707, Val Loss: 0.7464
Epoch [35/150], Train Loss: 0.7693, Val Loss: 0.7478
Epoch [36/150], Train Loss: 0.7708, Val Loss: 0.7415
Epoch [37/150], Train Loss: 0.7685, Val Loss: 0.7489
Epoch [38/150], Train Loss: 0.7651, Val Loss: 0.7368
Epoch [39/150], Train Loss: 0.7631, Val Loss: 0.7395
Epoch [40/150], Train Loss: 0.7618, Val Loss: 0.7299
Epoch [41/150], Train Loss: 0.7598, Val Loss: 0.7291
Epoch [42/150], Train Loss: 0.7576, Val Loss: 0.7399
Epoch [43/150], Train Loss: 0.7556, Val Loss: 0.7284
Epoch [44/150], Train Loss: 0.7568, Val Loss: 0.730
Epoch [45/150], Train Loss: 0.7558, Val Loss: 0.728
Epoch [46/150], Train Loss: 0.7535, Val Loss: 0.728
Epoch [47/150], Train Loss: 0.7527, Val Loss: 0.727
Epoch [48/150], Train Loss: 0.7515, Val Loss: 0.725
Epoch [49/150], Train Loss: 0.7507, Val Loss: 0.728
Epoch [50/150], Train Loss: 0.7512, Val Loss: 0.725
Epoch [51/150], Train Loss: 0.7497, Val Loss: 0.722
Epoch [52/150], Train Loss: 0.7479, Val Loss: 0.724
Epoch [53/150], Train Loss: 0.7495, Val Loss: 0.723
Epoch [54/150], Train Loss: 0.7483, Val Loss: 0.723
Epoch [55/150], Train Loss: 0.7468, Val Loss: 0.725
Epoch [56/150], Train Loss: 0.7455, Val Loss: 0.721
Epoch [57/150], Train Loss: 0.7437, Val Loss: 0.720
Epoch [58/150], Train Loss: 0.7428, Val Loss: 0.720
Epoch [59/150], Train Loss: 0.7432, Val Loss: 0.718
Epoch [60/150], Train Loss: 0.7408, Val Loss: 0.718
Epoch [61/150], Train Loss: 0.7440, Val Loss: 0.718
Epoch [62/150], Train Loss: 0.7416, Val Loss: 0.724
Epoch [63/150], Train Loss: 0.7395, Val Loss: 0.719
Epoch [64/150], Train Loss: 0.7397, Val Loss: 0.715
Epoch [65/150], Train Loss: 0.7386, Val Loss: 0.725
Epoch [66/150], Train Loss: 0.7381, Val Loss: 0.718
Epoch [67/150], Train Loss: 0.7380, Val Loss: 0.715
Epoch [68/150], Train Loss: 0.7352, Val Loss: 0.715
Epoch [69/150], Train Loss: 0.7372, Val Loss: 0.712
Epoch [70/150], Train Loss: 0.7366, Val Loss: 0.714
Epoch [71/150], Train Loss: 0.7349, Val Loss: 0.716
Epoch [72/150], Train Loss: 0.7342, Val Loss: 0.718
Epoch [73/150], Train Loss: 0.7331, Val Loss: 0.713
Epoch [74/150], Train Loss: 0.7343, Val Loss: 0.711
Epoch [75/150], Train Loss: 0.7333, Val Loss: 0.712
Epoch [76/150], Train Loss: 0.7340, Val Loss: 0.714
Epoch [77/150], Train Loss: 0.7329, Val Loss: 0.710
Epoch [78/150], Train Loss: 0.7320, Val Loss: 0.709
Epoch [79/150], Train Loss: 0.7317, Val Loss: 0.709
Epoch [80/150], Train Loss: 0.7309, Val Loss: 0.711
Epoch [81/150], Train Loss: 0.7311, Val Loss: 0.713
Epoch [82/150], Train Loss: 0.7290, Val Loss: 0.713
Epoch [83/150], Train Loss: 0.7314, Val Loss: 0.7095
Epoch [84/150], Train Loss: 0.7283, Val Loss: 0.7112
Epoch [85/150], Train Loss: 0.7289, Val Loss: 0.7064
Epoch [86/150], Train Loss: 0.7276, Val Loss: 0.7093
Epoch [87/150], Train Loss: 0.7264, Val Loss: 0.7086
Epoch [88/150], Train Loss: 0.7286, Val Loss: 0.7068
Epoch [89/150], Train Loss: 0.7277, Val Loss: 0.7069
Epoch [90/150], Train Loss: 0.7266, Val Loss: 0.7076
Epoch [91/150], Train Loss: 0.7245, Val Loss: 0.7079
Epoch [92/150], Train Loss: 0.7262, Val Loss: 0.7088
Epoch [93/150], Train Loss: 0.7265, Val Loss: 0.7062
Epoch [94/150], Train Loss: 0.7231, Val Loss: 0.7101
Epoch [95/150], Train Loss: 0.7243, Val Loss: 0.7043
Epoch [96/150], Train Loss: 0.7250, Val Loss: 0.7122
Epoch [97/150], Train Loss: 0.7244, Val Loss: 0.7044
Epoch [98/150], Train Loss: 0.7238, Val Loss: 0.7068
Epoch [99/150], Train Loss: 0.7224, Val Loss: 0.7054
Epoch [100/150], Train Loss: 0.7239, Val Loss: 0.7056
Epoch [101/150], Train Loss: 0.7241, Val Loss: 0.7056
Epoch [102/150], Train Loss: 0.7254, Val Loss: 0.7069
Epoch [103/150], Train Loss: 0.7208, Val Loss: 0.7035
Epoch [104/150], Train Loss: 0.7221, Val Loss: 0.7072
Epoch [105/150], Train Loss: 0.7221, Val Loss: 0.7044
Epoch [106/150], Train Loss: 0.7203, Val Loss: 0.7069
Epoch [107/150], Train Loss: 0.7219, Val Loss: 0.7025
Epoch [108/150], Train Loss: 0.7189, Val Loss: 0.7039
Epoch [109/150], Train Loss: 0.7194, Val Loss: 0.7017
Epoch [110/150], Train Loss: 0.7208, Val Loss: 0.7072
Epoch [111/150], Train Loss: 0.7214, Val Loss: 0.7035
Epoch [112/150], Train Loss: 0.7182, Val Loss: 0.7009
Epoch [113/150], Train Loss: 0.7174, Val Loss: 0.7014
Epoch [114/150], Train Loss: 0.7202, Val Loss: 0.6993
Epoch [115/150], Train Loss: 0.7182, Val Loss: 0.7019
Epoch [116/150], Train Loss: 0.7187, Val Loss: 0.6998
Epoch [117/150], Train Loss: 0.7178, Val Loss: 0.7021
Epoch [118/150], Train Loss: 0.7185, Val Loss: 0.7024
Epoch [119/150], Train Loss: 0.7178, Val Loss: 0.7000
Epoch [120/150], Train Loss: 0.7188, Val Loss: 0.6997
Epoch [121/150], Train Loss: 0.7154, Val Loss: 0.7023
Epoch [122/150], Train Loss: 0.7159, Val Loss: 0.7015
Epoch [123/150], Train Loss: 0.7181, Val Loss: 0.7013
Epoch [124/150], Train Loss: 0.7055, Val Loss: 0.6957
Epoch [125/150], Train Loss: 0.7022, Val Loss: 0.6948
Epoch [126/150], Train Loss: 0.7013, Val Loss: 0.6945
Epoch [127/150], Train Loss: 0.7002, Val Loss: 0.6938
Epoch [128/150], Train Loss: 0.7001, Val Loss: 0.6935
Epoch [129/150], Train Loss: 0.6997, Val Loss: 0.6935
Epoch [130/150], Train Loss: 0.7000, Val Loss: 0.6932
Epoch [131/150], Train Loss: 0.6985, Val Loss: 0.6937
Epoch [132/150], Train Loss: 0.6986, Val Loss: 0.6927
Epoch [133/150], Train Loss: 0.6981, Val Loss: 0.6929
Epoch [134/150], Train Loss: 0.6978, Val Loss: 0.6927
Epoch [135/150], Train Loss: 0.6979, Val Loss: 0.6926
Epoch [136/150], Train Loss: 0.6974, Val Loss: 0.6924
Epoch [137/150], Train Loss: 0.6972, Val Loss: 0.6927
Epoch [138/150], Train Loss: 0.6966, Val Loss: 0.6925
Epoch [139/150], Train Loss: 0.6970, Val Loss: 0.6919
Epoch [140/150], Train Loss: 0.6970, Val Loss: 0.6919
Epoch [141/150], Train Loss: 0.6965, Val Loss: 0.6917
Epoch [142/150], Train Loss: 0.6966, Val Loss: 0.6919
Epoch [143/150], Train Loss: 0.6965, Val Loss: 0.6914
Epoch [144/150], Train Loss: 0.6966, Val Loss: 0.6914
Epoch [145/150], Train Loss: 0.6957, Val Loss: 0.6916
Epoch [146/150], Train Loss: 0.6958, Val Loss: 0.6910
Epoch [147/150], Train Loss: 0.6949, Val Loss: 0.6912
Epoch [148/150], Train Loss: 0.6953, Val Loss: 0.6913
Epoch [149/150], Train Loss: 0.6957, Val Loss: 0.6912
Epoch [150/150], Train Loss: 0.6950, Val Loss: 0.6911
Unscaled Test Loss: 3.5204
```

II/ f) Results



100
80
60
40
20
0
0
5000
10000
15000
20000
25000
Predictions
Truth