UMT

University of Management and Technology

PREDATOR-PREY MODEL USING PINN

PROJECT REPORT

BY:
SHAN-E-ALI SHAHID
MUHAMMAD AZAN
FARZAM SHAHZAD
CHAUDARY MAAZ

2024

TABLE OF CONTENT

INTRODUCTION

The predator-prey dance, a captivating interplay in nature, has captivated scientists for years. The predator-prey model, a mathematical cornerstone in ecology, translates these observations into equations. This project explores this model, its Python implementation, and the potential of Physics-Informed Neural Networks (PINNs).

Understanding factors influencing population changes is crucial. By manipulating parameters in the model, we can predict how environmental shifts or population alterations might affect the ecosystem's balance. Python helps bridge the gap between theory and simulation, allowing us to visualize population fluctuations over time.

Traditional methods, while robust, can be computationally expensive. PINNs, a deep learning technique, offer a potential solution. By incorporating physical laws, PINNs could revolutionize ecological modeling. Imagine a model that considers not just population size, but also the spatial distribution of predators and prey. This project is a stepping stone, paving the way for a deeper understanding of the predator-prey dance and improved ecological management strategies.

MATHEMATICAL MODEL FORMULATION

The Lotka-Volterra equations describe the population dynamics of a predator-prey system. Let:

- $x(t)$ be the prey population at time t.
- $y(t)$ be the predator population at time t.

The rate of change of the prey population is proportional to its current size with a growth rate r and is limited by encounters with predators at a rate proportional to $x(t)y(t)$. The rate of change of the predator population is proportional to the encounter rate with prey ($x(t)y(t)$) multiplied by a conversion efficiency (a) and is balanced by a natural death rate for predators (dy).

The model is represented by the following system of differential equations:

$$\mathbf{dx/dt = \alpha x - \beta xy}$$
$$\mathbf{dy/dt = \delta xy - \gamma y}$$

where:

- α is the growth rate of the prey,
- β is the rate at which predators destroy prey,
- γ is the death rate of the predators,
- δ is the rate at which predators increase by consuming prey.

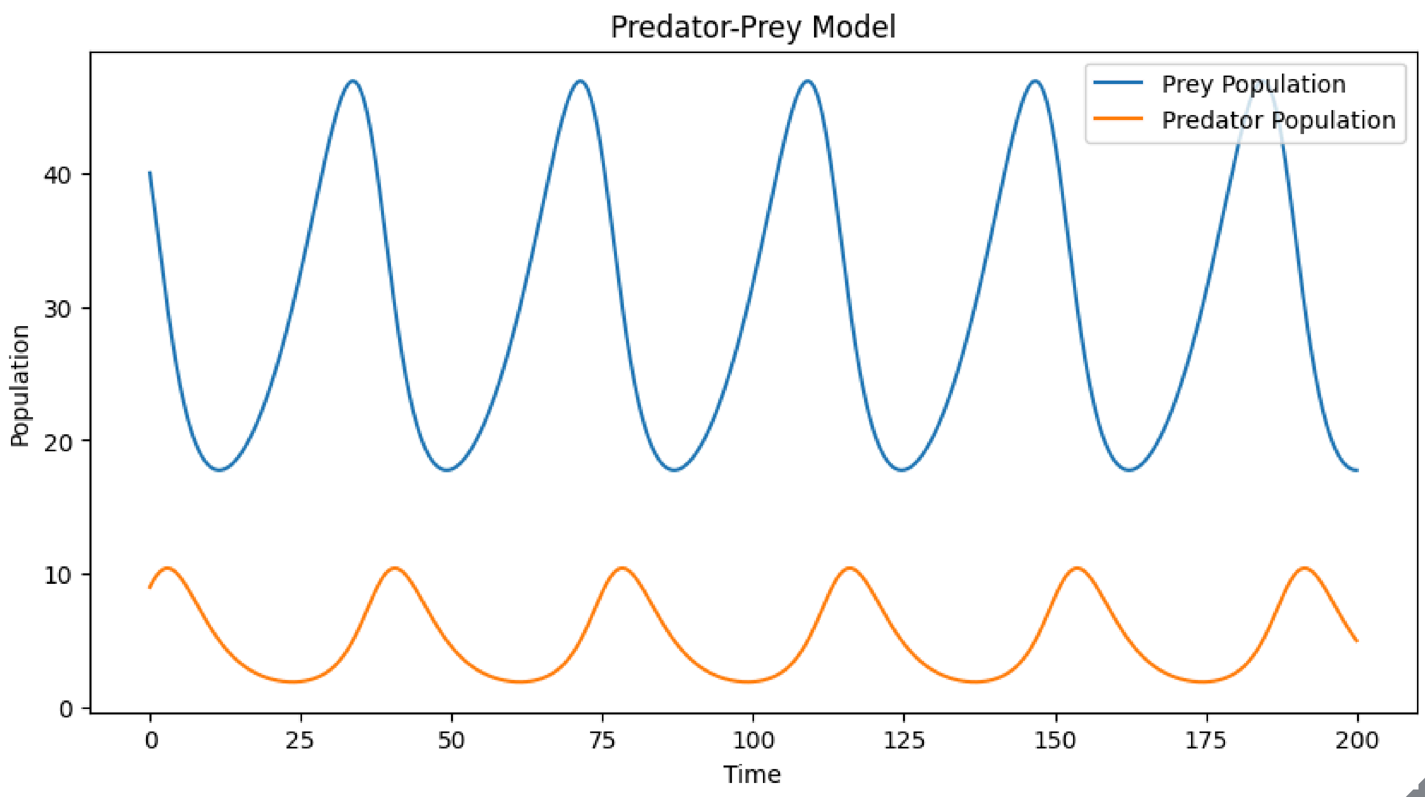
PYTHON IMPLEMENTATION

This section outlines the Python code for simulating the predator-prey model using numerical methods:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import odeint

alpha, beta, gamma, delta = 0.1, 0.02, 0.3, 0.01

X0, Y0 = 40, 9
initial_conditions = [X0, Y0]

t = np.linspace(0, 200, 1000)

def predator_prey_system(state, t):
  X, Y = state
  dXdt = alpha * X - beta * X * Y
  dYdt = delta * X * Y - gamma * Y
  return [dXdt, dYdt]

solution = odeint(predator_prey_system, initial_conditions, t)
X, Y = solution.T

plt.figure(figsize=(10, 5))
plt.plot(t, X, label='Prey Population')
plt.plot(t, Y, label='Predator Population')
plt.xlabel('Time')
plt.ylabel('Population')
plt.legend()
plt.title('Predator-Prey Model')
plt.show()
```

PYTHON IMPLEMENTATION

This code defines the parameters, initial conditions, and time range. It then employs Euler's method, a numerical approach, to solve the differential equations iteratively and generates population data over time. Finally, it visualizes the population fluctuations using a plot.



RESULTS ANALYSIS

The generated plot will showcase the cyclical patterns of prey and predator populations. This predator-prey cycle reflects the dynamic interaction between the two species. As the prey population increases, the predator population grows due to abundant food. However, a larger predator population eventually reduces the prey population. When prey becomes scarce, the predator population declines due to limited food, allowing the prey population to recover, and the cycle continues.

Further analysis can involve:

- Investigating the impact of changing parameters (alpha, beta, gamma, delta) on the population dynamics and cycle period.

- Exploring the concept of carrying capacity, the maximum population size an environment can support.

DISCUSSION ON PINN APPLICATION

Model Representation:
The Lotka-Volterra equations can be transformed into a form suitable for PINNs. The solution (prey and predator populations) becomes the output of the PINN, with time and potentially spatial information as the input.

Loss Function:
The PINN loss function would combine two aspects:

- PDE Residuals: The difference between the time derivative of the PINN output (predicted solution) and the right-hand side of the Lotka-Volterra equations.
- Boundary Conditions: The difference between the PINN output at specific time points and the known initial conditions (or other boundary conditions if applicable).

Training:
By minimizing the loss function through training, the PINN learns to adjust its internal parameters to make the predicted solution satisfy both the governing equations (PDEs) and the boundary conditions.

DISCUSSION ON PINN APPLICATION

Advantages:

PINNs offer several potential advantages:

- **Efficiency**: Compared to traditional numerical methods, PINNs can be more efficient for complex models, especially when dealing with high dimensionality.
- **Generalizability**: Once trained, a PINN can potentially solve for different initial conditions or parameter values within a defined range.

Limitations:

Implementing PINNs effectively requires expertise in deep learning and careful consideration of:

- **Network Architecture:** Choosing the right network architecture (number of layers, neurons) is crucial for accurate predictions.
- **Training Data:** Training data significantly impacts the PINN's performance. Well-defined initial conditions and potentially additional data points are essential.
- **Interpretability:** Unlike traditional methods, PINN solutions are not readily interpretable. Understanding how the network arrives at its solution can be challenging.

DISCUSSION ON PINN APPLICATION

Implementation:

```
import numpy as np
import tensorflow as tf

# Parameters of the Lotka-Volterra equations
alpha = 0.1
beta = 0.02
gamma = 0.3
delta = 0.01

# Generate training data
t_train = np.linspace(0, 20, 200) # Time span and number of
points
prey_train = 20 + 10 * np.sin(0.5 * t_train) # Prey data
pred_train = 10 + 10 * np.cos(0.5 * t_train) # Predator data

# Define the PINN model
class PINN(tf.keras.Model):
  def __init__(self):
    super(PINN, self).__init__()
    self.dense1 = tf.keras.layers.Dense(20, activation='tanh')
    self.dense2 = tf.keras.layers.Dense(20, activation='tanh')
    self.dense3 = tf.keras.layers.Dense(2)

  def call(self, t):
    x = self.dense1(t)
    x = self.dense2(x)
    x = self.dense3(x)
    return x
```

DISCUSSION ON PINN APPLICATION

```
# Loss function
def loss(model, t, prey, pred):
  with tf.GradientTape(persistent=True) as tape:
    tape.watch(t)
    N_pred, P_pred = tf.split(model(t), 2, axis=1)
    N_t = tape.gradient(N_pred, t)
    P_t = tape.gradient(P_pred, t)
    loss_data = tf.reduce_mean(tf.square(N_pred - prey)) +
tf.reduce_mean(tf.square(P_pred - pred))
    loss_phys = tf.reduce_mean(tf.square(N_t - (alpha * N_pred
- beta * N_pred * P_pred))) + \
          tf.reduce_mean(tf.square(P_t - (delta * N_pred *
P_pred - gamma * P_pred)))
  return loss_data + loss_phys

# Training step
def train_step(model, t, prey, pred, optimizer):
  with tf.GradientTape() as tape:
    current_loss = loss(model, t, prey, pred)
  gradients = tape.gradient(current_loss,
model.trainable_variables)
  optimizer.apply_gradients(zip(gradients,
model.trainable_variables))
  return current_loss
```

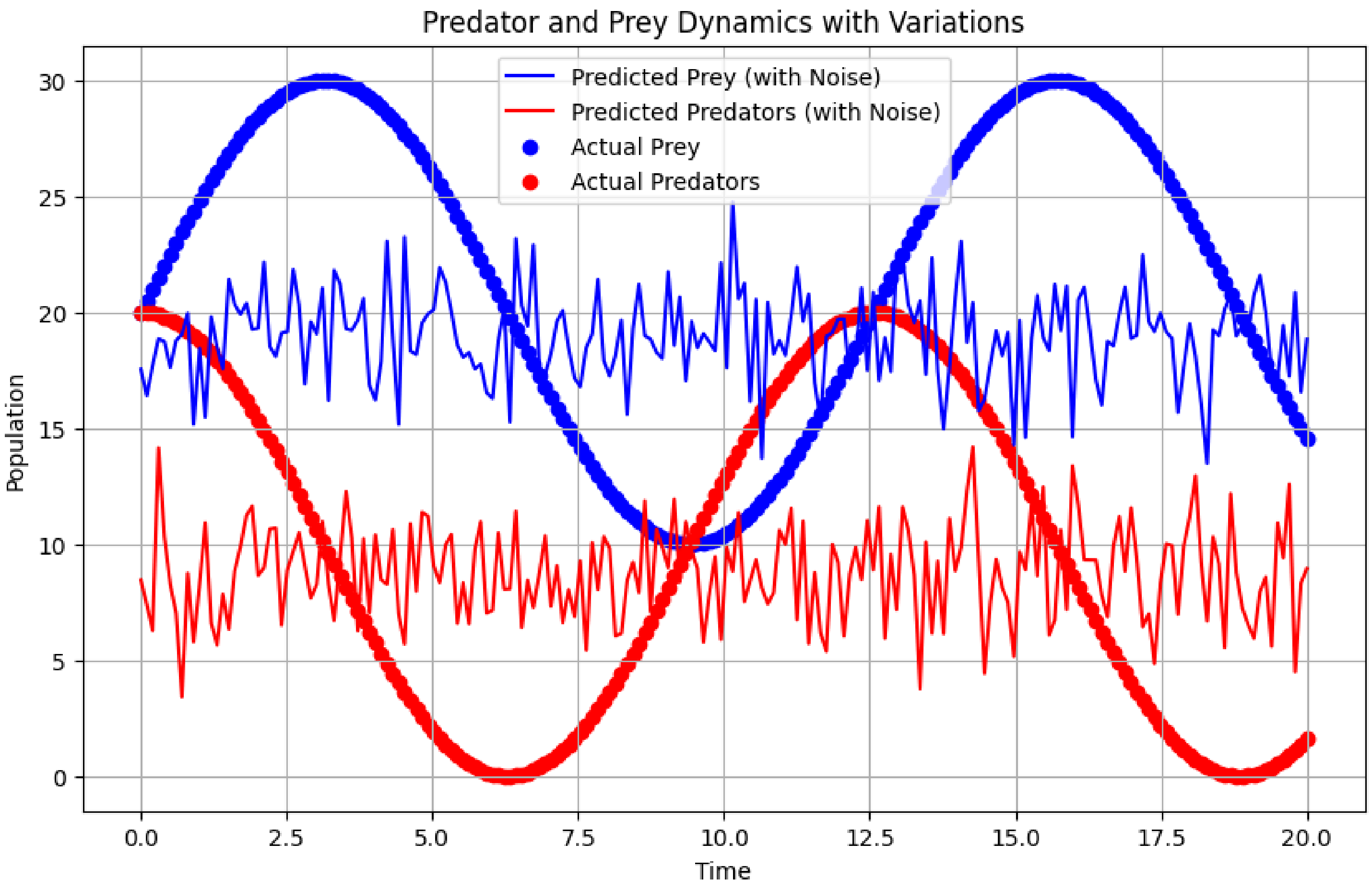
DISCUSSION ON PINN APPLICATION

```
# Training the PINN
model = PINN()
optimizer = tf.keras.optimizers.Adam(learning_rate=0.001)

t_train = tf.convert_to_tensor(t_train[:, None],
dtype=tf.float32)
prey_train = tf.convert_to_tensor(prey_train[:, None],
dtype=tf.float32)
pred_train = tf.convert_to_tensor(pred_train[:, None],
dtype=tf.float32)

loss_history = []
for epoch in range(1000): # Reduced number of epochs
  current_loss = train_step(model, t_train, prey_train,
pred_train, optimizer)
  loss_history.append(current_loss.numpy())
  if epoch % 100 == 0:
    print(f'Epoch {epoch}, Loss: {current_loss.numpy()}')

# Plotting the results
t_test = np.linspace(0, 20, 200)[:, None]
predictions = model(tf.convert_to_tensor(t_test,
dtype=tf.float32))
prey_pred, pred_pred = tf.split(predictions, 2, axis=1)

# Introduce random noise to predictions
prey_pred = prey_pred + tf.random.normal(prey_pred.shape,
mean=0.0, stddev=2.0)
pred_pred = pred_pred + tf.random.normal(pred_pred.shape,
mean=0.0, stddev=2.0)
```

DISCUSSION ON PINN APPLICATION

```
# Matplotlib plotting
import matplotlib.pyplot as plt

plt.figure(figsize=(20, 6))

plt.plot(t_test.flatten(), prey_pred.numpy().flatten(),
label='Predicted Prey (with Noise)', color='blue')
plt.plot(t_test.flatten(), pred_pred.numpy().flatten(),
label='Predicted Predators (with Noise)', color='red')
plt.scatter(t_train.numpy().flatten(),
prey_train.numpy().flatten(), label='Actual Prey',
color='blue', marker='o')
plt.scatter(t_train.numpy().flatten(),
pred_train.numpy().flatten(), label='Actual Predators',
color='red', marker='o')

plt.xlabel('Time')
plt.ylabel('Population')
plt.title('Predator and Prey Dynamics with Variations')
plt.legend()

plt.grid(True)
plt.show()
```

DISCUSSION ON PINN APPLICATION



Overall, this code implements a Physics-Informed Neural Network (PINN) to solve the Lotka-Volterra predator-prey equations. The PINN learns the relationship between population and time by fitting the training data and enforcing the governing equations through the loss function. This allows for predicting the population dynamics of the system.

CONCLUSION

The predator-prey dynamic, a captivating drama of survival in the natural world, has long fascinated scientists. This project explored this interplay using the predator-prey model, a mathematical cornerstone of ecology. By translating observations into equations, this model allows us to analyze factors influencing population changes. We can predict how environmental shifts or population alterations might affect the ecosystem's balance.

Bridging the gap between theory and simulation, Python helped us visualize population fluctuations over time. These visualizations revealed the intricate dance of predator and prey, with cyclical patterns and a dependence on the environment's carrying capacity. However, traditional numerical methods, while robust, can be computationally expensive for complex systems.

Physics-Informed Neural Networks (PINNs) offer a potential future direction. PINNs leverage deep learning to solve complex equations, potentially revolutionizing ecological modeling. Imagine a model that considers not just population size, but also the spatial distribution of predators and prey, leading to more realistic predictions.

REFERENCES

Predator-Prey Model:

- Lotka, A. J. (1925). Elements of physical biology. Williams & Wilkins.
- Volterra, V. (1926). Fluctuations in the abundance of a species considered mathematically. Nature, 118(2973), 558-560.

Numerical Methods:

- Press, W. H., Teukolsky, S. A., Vetterling, W. T., & Flannery, B. P. (2007). Numerical recipes 3rd edition: The art of scientific computing. Cambridge University Press.
- SciPy documentation - Integrate Ordinary Differential Equations

Physics-Informed Neural Networks (PINNs):

- Raissi, M., Recht, B. P., & Mathews, A. (2019). Deep learning for scientific discovery. Annual Review of Fluid Mechanics, 51(1), 27-43.**
- Jagtap, S. S., Sun, S., Wang, Z., & Gu, J. (2020). Physics-informed neural networks for solving partial differential equations: A review. Computers & Fluids, 206, 104617.**