You can access these slides on the course Github:
https://github.com/natrask/ENM1050

ENGR 1050
Intro to Scientific Computation

Lecture 10 – Deep Neural Networks

Prof. Nat Trask
Mechanical Engineering & Applied Mechanics
University of Pennsylvania

Rest of Semester Planning

Lecture Wednesday Optional

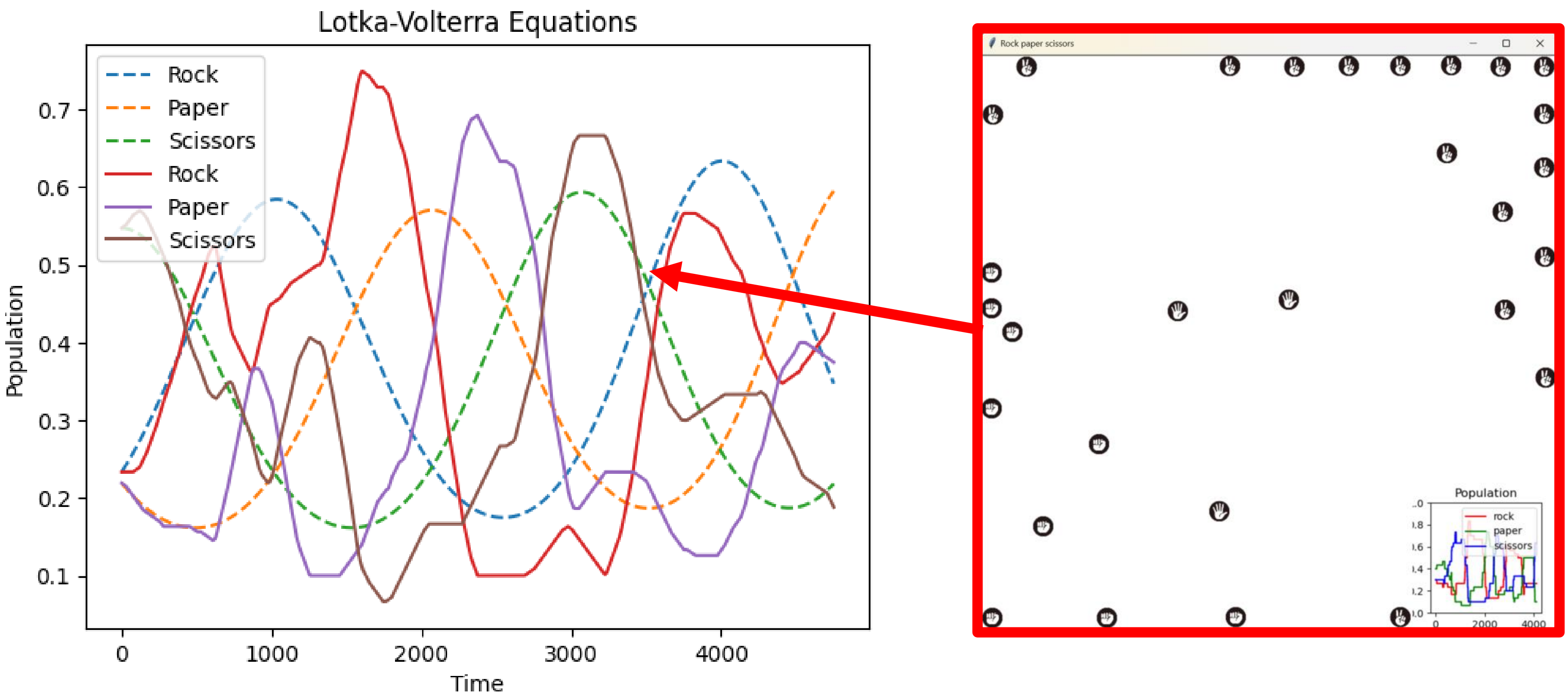
Poll for cancelling Wed lecture before thanksgiving

Wednesday homework

Come to my OH – everyone who has come has left with a completed code!

Today we will introduce neural networks, and then review the process for how to fit a pytorch model, and provide a step-by-step guide through the homework for those who haven't been able to get 1-1 time yet

Motivation: improving qualitative fit to data



Correct trends: we obtain repeated rock, paper, scissors pattern, but incorrect peaks

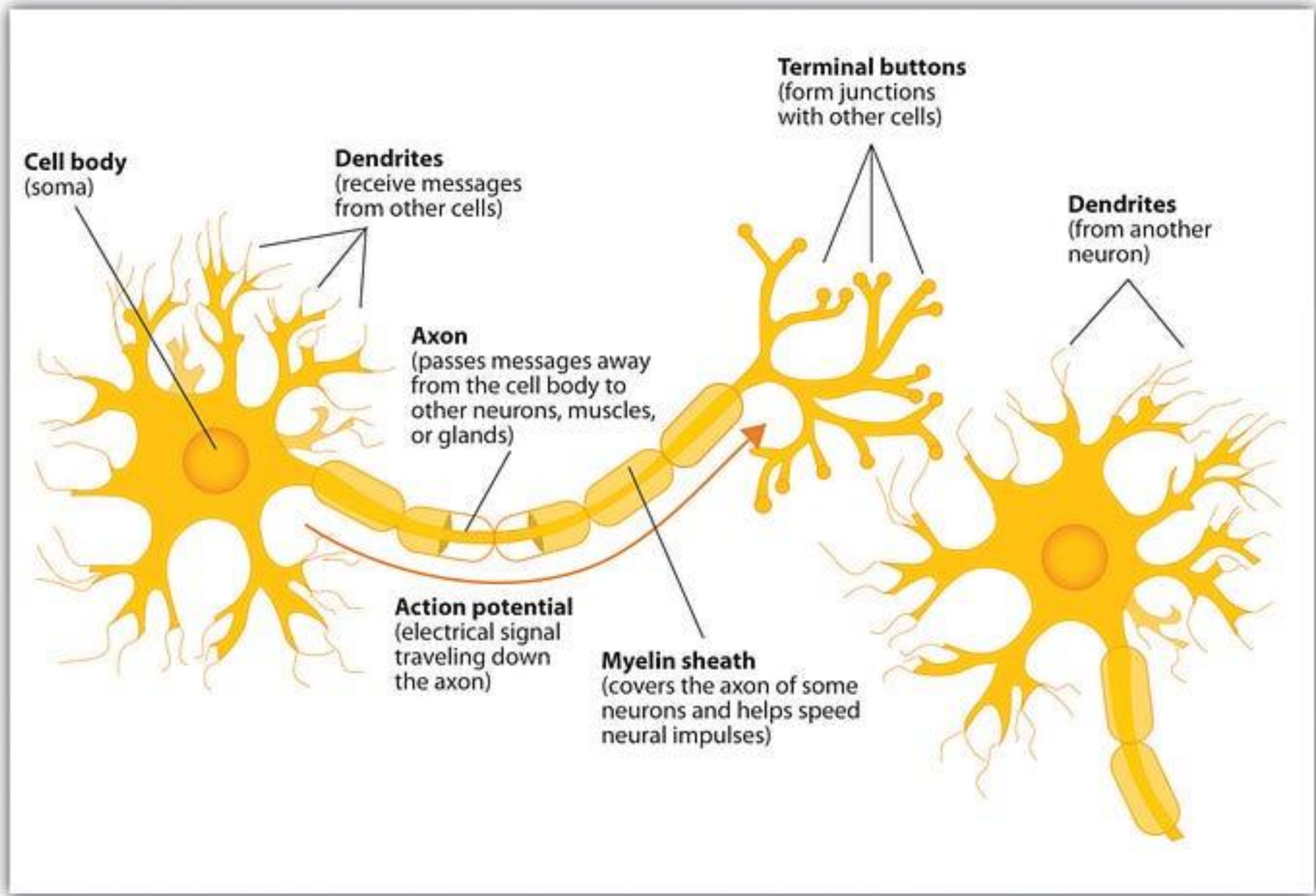
Limits of human cognition. Our model form was limited to some logic and guesses. Can we use machine learning to find better model form?

$$\dot{x} = \alpha\, x * y + \beta\, x * z$$

$$\dot{y} = \gamma\, y * x + \delta\, y * z$$

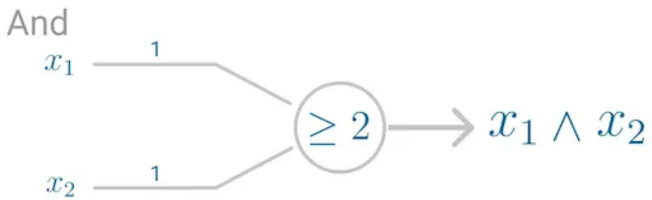
$$\dot{z} = \epsilon\, z * x + \zeta\, z * y$$

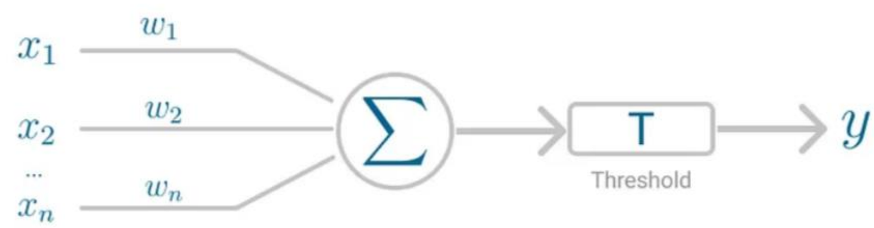
Introducing multilayer perceptrons (MLPs)

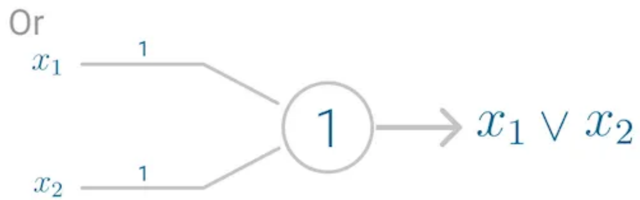


https://towardsdatascience.com/multilayer-perceptron-explained-with-a-real-life-example-and-python-code-sentiment-analysis-cb408ee93141

Introducing multilayer perceptrons (MLPs)





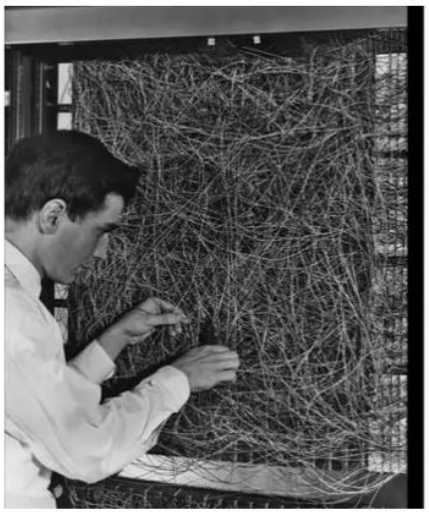


$$y = \begin{cases} 1, \text{if } \overbrace{\sum_i w_i x_i}^{\text{weighted sum}} - \underbrace{T}_{\text{threshold}} > 0 \\ 0, \text{otherwise} \end{cases}$$

In 1943, Warren McCulloch and Walter Pitts proposed the binary artificial neuron as a logical model of biological neural networks.

https://towardsdatascience.com/multilayer-perceptron-explained-with-a-real-life-example-and-python-code-sentiment-analysis-cb408ee93141

$$\mathcal{L}(\mathbf{w}) = -\sum_{i \in \mathcal{M}} \mathbf{w}^{\mathrm{T}} \mathbf{x}_i y_i$$



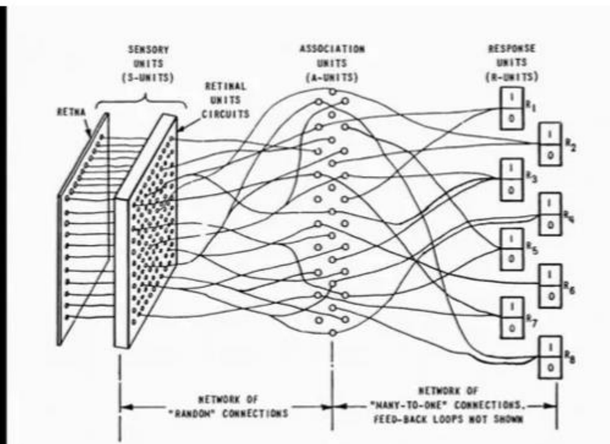
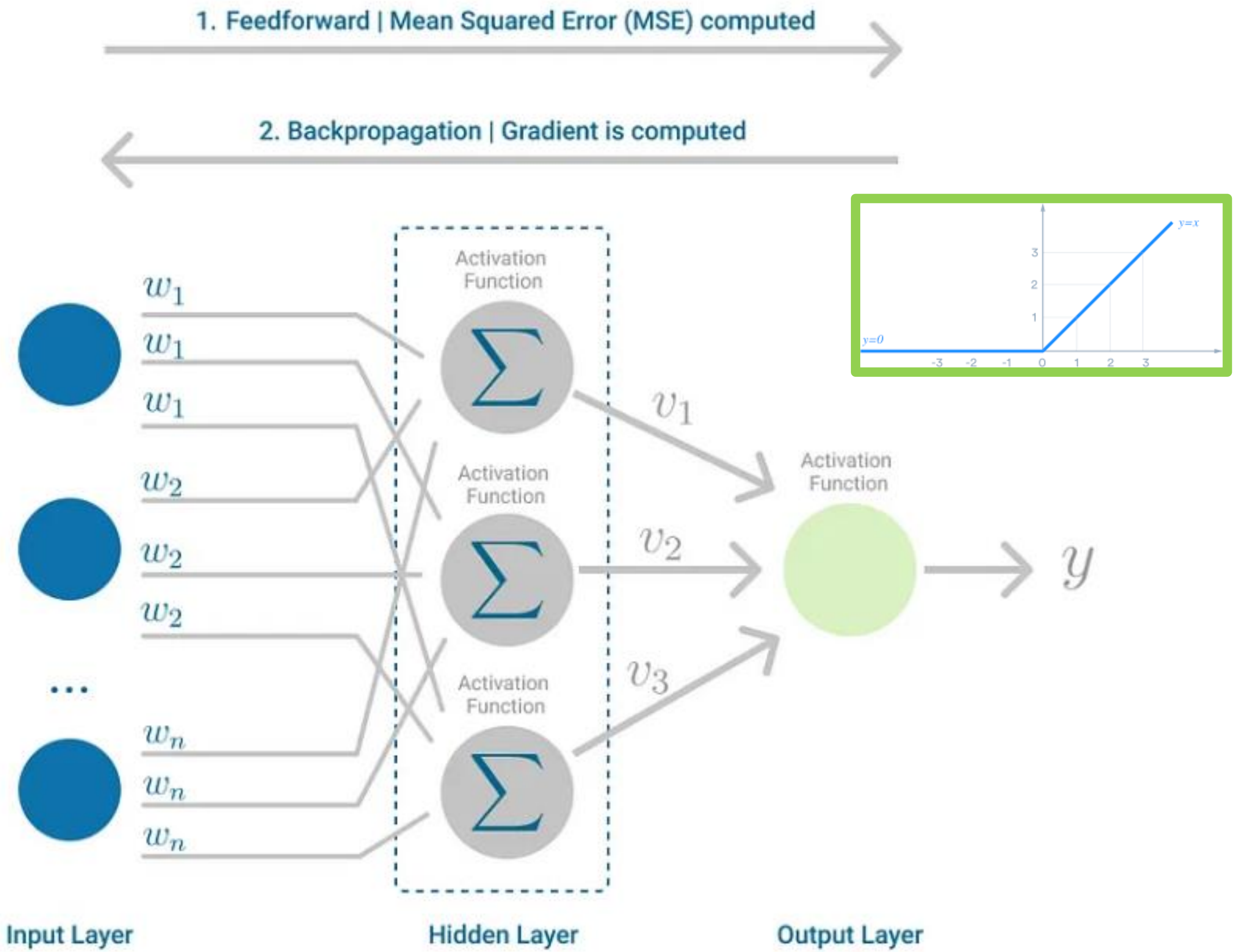


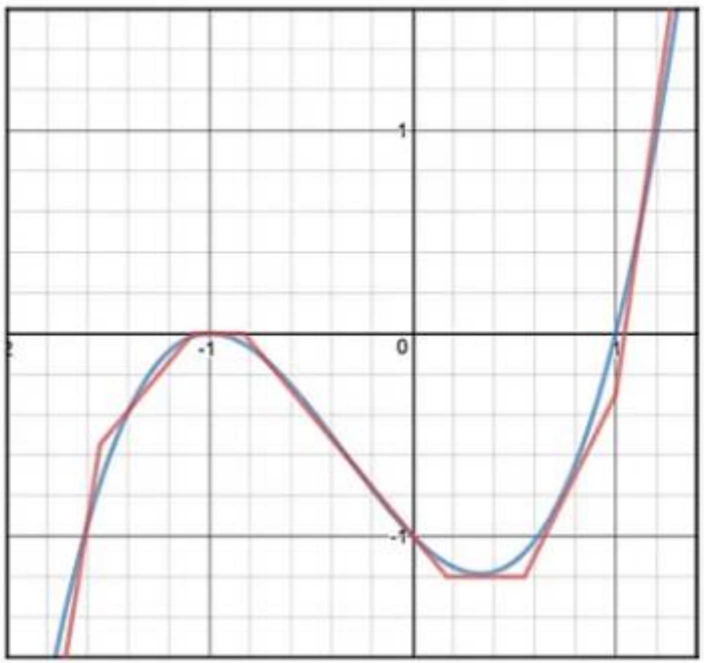
Figure I ORGANIZATION OF THE MARK I PERCEPTRON

The Mark I Perceptron was a machine that implemented the perceptron algorithm for image recognition.



https://towardsdatascience.com/multilayer-perceptron-explained-with-a-real-life-example-and-python-code-sentiment-analysis-cb408ee93141

Turns out to just be math - no actual neuroscience



$$n_1(x) = Relu(-5x - 7.7)$$
$$n_2(x) = Relu(-1.2x - 1.3)$$
$$n_3(x) = Relu(1.2x + 1)$$
$$n_4(x) = Relu(1.2x - .2)$$
$$n_5(x) = Relu(2x - 1.1)$$
$$n_6(x) = Relu(5x - 5)$$

$$Z(x) = -n_1(x) - n_2(x) - n_3(x) + n_4(x) + n_5(x) + n_6(x)$$

https://towardsdatascience.com/can-neural-networks-really-learn-any-function-65e106617fc6

How to understand the math (without doing math)

Theorem (Cybenko)

Let σ be any continuous discriminatory function. Then finite sums of the form

$$G(x) = \sum_{j=1}^{N} \alpha_j \sigma(w_j^T x + b_j), \text{ where } w_j \in \mathbb{R}^n,\ \alpha_j,\ b_j \in \mathbb{R}$$

are dense in $C(I_n)$.

In other words, given any $\varepsilon > 0$ and $f \in C(I_n)$, there is a sum $G(x)$ of the above form such that

$$|G(x) - f(x)| < \varepsilon, \quad \forall x \in I_n$$

In plain English: there are choices of parameters for neural networks can approximate any function of any type *to any accuracy*

Whether we can find those parameters with gradient descent is another story!

Some deep learning history (from MLP Wikipedia entry)

In 1943, Warren McCulloch and Walter Pitts proposed the binary artificial neuron as a logical model of biological neural networks.[11]

In 1958, Frank Rosenblatt proposed the multilayered perceptron model, consisting of an input layer, a hidden layer with randomized weights that did not learn, and an output layer with learnable connections.[12]

In 1962, Rosenblatt published many variants and experiments on perceptrons in his book *Principles of Neurodynamics*, including up to 2 trainable layers by "back-propagating errors".[13] However, it was not the backpropagation algorithm, and he did not have a general method for training multiple layers.

In 1965, Alexey Grigorevich Ivakhnenko and Valentin Lapa published Group Method of Data Handling. It was one of the first deep learning methods, used to train an eight-layer neural net in 1971.[14][15][16]

In 1967, Shun'ichi Amari reported [17] the first multilayered neural network trained by stochastic gradient descent, was able to classify non-linearily separable pattern classes. Amari's student Saito conducted the computer experiments, using a five-layered feedforward network with two learning layers.[16]

Backpropagation was independently developed multiple times in early 1970s. The earliest published instance was Seppo Linnainmaa's master thesis (1970).[18][19][16] Paul Werbos developed it independently in 1971,[20] but had difficulty publishing it until 1982.[21]

In 1986, David E. Rumelhart et al. popularized backpropagation.[22][23]

In 2003, interest in backpropagation networks returned due to the successes of deep learning being applied to language modelling by Yoshua Bengio with co-authors.[24]

Some deep learning history (from MLP Wikipedia entry)

In 1943, Warren McCulloch and Walter Pitts proposed the binary artificial neuron as a logical model of biological neural networks.[11]

In 1958, Frank Rosenblatt proposed the multilayered perceptron model, consisting of an input layer, a hidden layer with randomized weights that did not learn, and an output layer with learnable connections.[12]

In 1962, Rosenblatt published many variants and experiments on perceptrons in his book *Principles of Neurodynamics*, including up to 2 trainable layers by "back-propagating errors".[13] However, it was not the backpropagation algorithm, and he did not have a general method for training multiple layers.

In 1965, Alexey Grigorevich Ivakhnenko and Valentin Lapa published Group Method of Data Handling. It was one of the first deep learning methods, used to train an eight-layer neural net in 1971.[14][15][16]

In 1967, Shun'ichi Amari reported [17] the first multilayered neural network trained by stochastic gradient descent, was able to classify non-linearily separable pattern classes. Amari's student Saito conducted the computer experiments, using a five-layered feedforward network with two learning layers.[16]

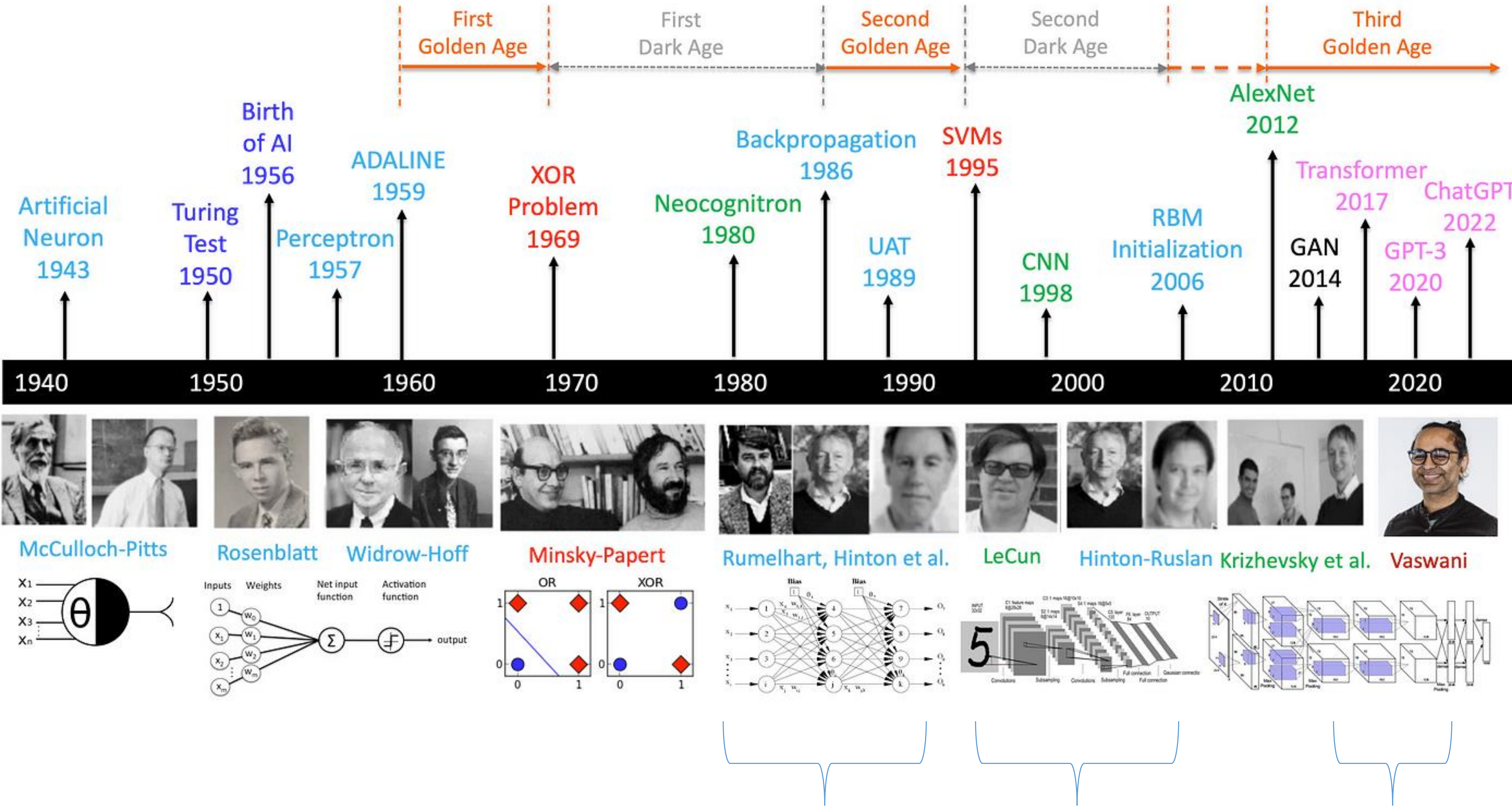
Backpropagation was independently developed multiple times in early 1970s. The earliest published instance was Seppo Linnainmaa's master thesis (1970).[18][19][16] Paul Werbos developed it independently in 1971,[20] but had difficulty publishing it until 1982.[21]

In 1986, David E. Rumelhart et al. popularized backpropagation.[22][23]

Automatic Differentiation!!!

In 2003, interest in backpropagation networks returned due to the successes of deep learning being applied to language modelling by Yoshua Bengio with co-authors.[24]

A Brief History of AI with Deep Learning



Today **Last Wed** **Modern LLMs**

https://medium.com/@lmpo/a-brief-history-of-ai-with-deep-learning-26f7948bc87b

A Review of Pytorch Pipeline

Steps to fit a model in PyTorch

If you are already comfortable, feel free to complete todays exercise instead

PyTorch Modeling Pipeline

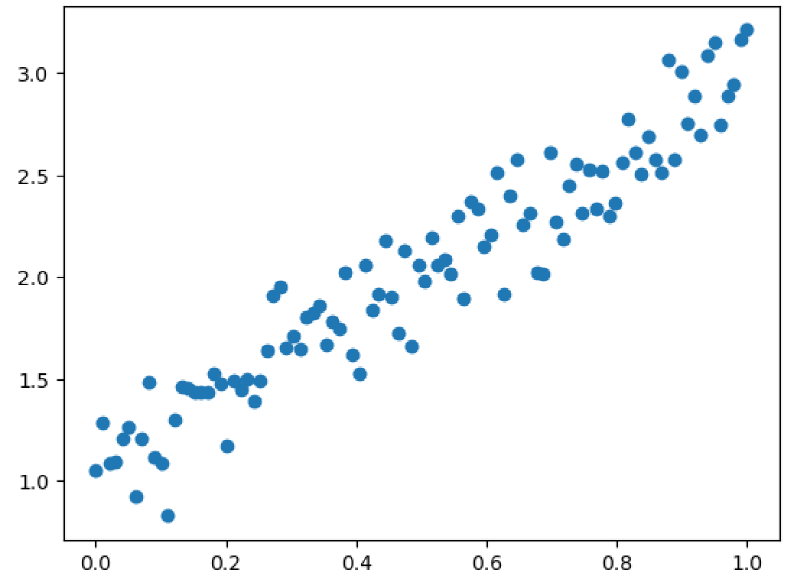
1. **Load data in**
2. **Build PyTorch class of hypothesized model form**
   A. Specify trainable coefficients in initializer
   B. Specify model form in forward()
3. **Set up optimizer**
4. **Train model**
5. **Visualize results**

PyTorch Modeling Pipeline

1. **Load data in**
2. **Build PyTorch class of hypothesized model form**
   A. Specify trainable coefficients in initializer
   B. Specify model form in forward()
3. **Set up optimizer**
4. **Train model**
5. **Visualize results**

Step 1: Load data in

```
import numpy as np
import matplotlib.pyplot as plt
def generate_noisylinear_data(N):
    x = np.linspace(0,1,N)
    y = 2*x + 1 + np.random.normal(0,0.2,N)
    # modify function here
    return  x,y

# Scatter plot data to get a sense of the dataset
[xdata,ydata] = generate_noisylinear_data(100)
plt.plot(xdata,ydata,'o')

# Put our data into a tensor
import torch
x_data = torch.tensor(xdata,dtype=torch.float32)
y_data = torch.tensor(ydata,dtype=torch.float32)

# Print size of tensors, reshape if necessary
print(x_data.shape)
print(y_data.shape)
```




```
torch.Size([100])
torch.Size([100])
```

Step 1: What if you're handed data in wrong order?

```python
import torch

# Example tensor with incorrect shape
tensor = torch.randn(2, 3)  # Example: 2 rows, 3 columns

# Transpose the tensor
transposed_tensor = tensor.transpose(0, 1)  # Swap dimensions 0 and 1

print("Original Tensor:")
print(tensor)
print("\nTransposed Tensor:")
transposed_tensor

# Example: if your data is in the form (number of samples, number of features),
# and you want to use a neural network that expects (number of features, number of samples),
# you can transpose it like this:
# data_transposed = data.transpose(0, 1)
```

```
Original Tensor:
tensor([[-0.2186, -0.6188, -0.0537],
        [ 0.9262,  1.2093,  0.7276]])

Transposed Tensor:
tensor([[-0.2186,  0.9262],
        [-0.6188,  1.2093],
        [-0.0537,  0.7276]])
```

Step 1: What if you're handed data in wrong shape?

```
import torch

# Example tensor with incorrect shape
tensor = torch.randn(10, 2, 3)  # Example tensor with shape (10, 2, 3)

# Desired shape
new_shape = (10, 6)

# Resize the tensor using view()
# Note: The total number of elements must remain the same.
resized_tensor = tensor.view(new_shape)

# Alternatively, you can use reshape() which can handle cases where the tensor needs to be copied
# resized_tensor = tensor.reshape(new_shape)

print("Original tensor shape:", tensor.shape)
print("Resized tensor shape:", resized_tensor.shape)
```

```
Original tensor shape: torch.Size([10, 2, 3])
Resized tensor shape: torch.Size([10, 6])
```

PyTorch Modeling Pipeline

1. Load data in
2. **Build PyTorch class of hypothesized model form**
   A. Specify trainable coefficients in initializer
   B. Specify model form in forward()
3. Set up optimizer
4. Train model
5. Visualize results

Step 2: Build PyTorch model class

Ex. 1:
Linear regression

```
# Define the model (in our case, its y = A*x + b)
class LinearFitLayer(nn.Module):
    def __init__(self):
        super(LinearFitLayer, self).__init__()
        # Random guess for parameters A and B
        self.A = nn.Parameter(torch.randn(1))
        self.B = nn.Parameter(torch.randn(1))

    def forward(self, x):
        return self.A * x + self.B
```

Ex. 2:
Neural network

```
class NeuralNetworkLayer(nn.Module):
    def __init__(self):
        super(NeuralNetworkLayer, self).__init__()
        # Define a simple MLP with one hidden layer
        self.Nneurons = 1000
        self.hidden = nn.Linear(1, self.Nneurons)  # 1 input feature, 10 hidden units
        self.output = nn.Linear(self.Nneurons, 1)  # 10 hidden units, 1 output feature


    def forward(self, x):
        # We use the ReLU activation function for the hidden layer
        x = torch.relu(self.hidden(x))
        x = self.output(x)
        return x
```

Step 2: Build PyTorch model class

Overload the nn.Module base class

Ex. 1:
Linear regression

```python
# Define the model (in our case, its y = A*x + b)
class LinearFitLayer(nn.Module):
    def __init__(self):
        super(LinearFitLayer, self).__init__()
        # Random guess for parameters A and B
        self.A = nn.Parameter(torch.randn(1))
        self.B = nn.Parameter(torch.randn(1))

    def forward(self, x):
        return self.A * x + self.B
```

Ex. 2:
Neural network

```python
class NeuralNetworkLayer(nn.Module):
    def __init__(self):
        super(NeuralNetworkLayer, self).__init__()
        # Define a simple MLP with one hidden layer
        self.Nneurons = 1000
        self.hidden = nn.Linear(1, self.Nneurons)  # 1 input feature, 10 hidden units
        self.output = nn.Linear(self.Nneurons, 1)  # 10 hidden units, 1 output feature


    def forward(self, x):
        # We use the ReLU activation function for the hidden layer
        x = torch.relu(self.hidden(x))
        x = self.output(x)
        return x
```

Step 2: Build PyTorch model class

Ex. 1:
Linear regression

```python
# Define the model (in our case, its y = A*x + b)
class LinearFitLayer(nn.Module):
    def __init__(self):
        super(LinearFitLayer, self).__init__()
        # Random guess for parameters A and B
        self.A = nn.Parameter(torch.randn(1))
        self.B = nn.Parameter(torch.randn(1))

    def forward(self, x):
        return self.A * x + self.B
```

Define and initialize model parameters

Ex. 2:
Neural network

```python
class NeuralNetworkLayer(nn.Module):
    def __init__(self):
        super(NeuralNetworkLayer, self).__init__()
        # Define a simple MLP with one hidden layer
        self.Nneurons = 1000
        self.hidden = nn.Linear(1, self.Nneurons)  # 1 input feature, 10 hidden units
        self.output = nn.Linear(self.Nneurons, 1)  # 10 hidden units, 1 output feature


    def forward(self, x):
        # We use the ReLU activation function for the hidden layer
        x = torch.relu(self.hidden(x))
        x = self.output(x)
        return x
```

Step 2: Build PyTorch model class

Ex. 1:
Linear regression

```
# Define the model (in our case, its y = A*x + b)
class LinearFitLayer(nn.Module):
    def __init__(self):
        super(LinearFitLayer, self).__init__()
        # Random guess for parameters A and B
        self.A = nn.Parameter(torch.randn(1))
        self.B = nn.Parameter(torch.randn(1))

    def forward(self, x):
        return self.A * x + self.B
```

Define how to evaluate model

Ex. 2:
Neural network

```
class NeuralNetworkLayer(nn.Module):
    def __init__(self):
        super(NeuralNetworkLayer, self).__init__()
        # Define a simple MLP with one hidden layer
        self.Nneurons = 1000
        self.hidden = nn.Linear(1, self.Nneurons)  # 1 input feature, 10 hidden units
        self.output = nn.Linear(self.Nneurons, 1)  # 10 hidden units, 1 output feature


    def forward(self, x):
        # We use the ReLU activation function for the hidden layer
        x = torch.relu(self.hidden(x))
        x = self.output(x)
        return x
```

PyTorch Modeling Pipeline

1. **Load data in**
2. **Build PyTorch class of hypothesized model form**
   A. Specify trainable coefficients in initializer
   B. Specify model form in forward()
3. **Set up optimizer**
4. **Train model**
5. **Visualize results**

Step 3: Set up optimizer

```
# Instantiate the custom layer
model = NeuralNetworkLayer()

# Define the loss function and optimizer
criterion = nn.MSELoss()
optimizer = optim.SGD(model.parameters(), lr=0.001)
```

Step 3: What are some other choices for loss?

For Regression:

1. `nn.MSELoss()`: This is the Mean Squared Error loss, commonly used for regression problems. It calculates the squared difference between the predicted and target values, and averages it over the batch.
   - **When to use:** When you want to penalize large errors more heavily than small errors. It's a good default choice for most regression tasks.
2. `nn.L1Loss()`: This is the Mean Absolute Error loss, which calculates the absolute difference between the predicted and target values.
   - **When to use:** When you want to be less sensitive to outliers in your data, as it penalizes all errors equally.
3. `nn.SmoothL1Loss()`: This is a combination of MSE and MAE. It uses MSE for small errors and MAE for large errors, providing a balance between the two.
   - **When to use:** When you want a loss function that's robust to outliers but also penalizes large errors more heavily than MAE. It's often used in object detection tasks.

```
# How to define other types of losses
criterion = nn.MSELoss()
criterion = nn.L1Loss()
criterion = nn.HuberLoss()
criterion = nn.SmoothL1Loss()
```

For more information:
https://pytorch.org/docs/stable/nn.html#loss-functions

Step 3: What are some other choices for optimizer?

1. `torch.optim.Adam`:
   - **Characteristics:** A popular adaptive learning rate optimizer that combines the benefits of `AdaGrad` and `RMSprop`. It computes individual learning rates for different parameters based on their past gradients.
   - **When to use:** Often a good default choice for many tasks, especially when dealing with complex models and large datasets.
2. `torch.optim.RMSprop`:
   - **Characteristics:** Another adaptive learning rate optimizer that divides the learning rate by a running average of the magnitudes of recent gradients. It helps to prevent oscillations and speeds up convergence.
   - **When to use:** Can be effective for recurrent neural networks (RNNs) and tasks where the gradients can vary significantly.
3. `torch.optim.Adagrad`:
   - **Characteristics:** An adaptive learning rate optimizer that scales the learning rate for each parameter based on the sum of its past squared gradients. It gives larger updates to infrequent parameters and smaller updates to frequent ones.
   - **When to use:** Useful for sparse data and tasks where the gradients are sparse.

```python
# How to specify different optimizers
optimizer = optim.SGD(model.parameters(), lr=0.001)
optimizer = optim.Adam(model.parameters(), lr=0.001)
optimizer = optim.RMSprop(model.parameters(), lr=0.001)
optimizer = optim.Adagrad(model.parameters(), lr=0.001)
optimizer = optim.Adadelta(model.parameters(), lr=0.001)
optimizer = optim.Adamax(model.parameters(), lr=0.001)
optimizer = optim.NAdam(model.parameters(), lr=0.001)
optimizer = optim.LBFGS(model.parameters(), lr=0.001)
```

For more information:
https://pytorch.org/docs/stable/optim.html

PyTorch Modeling Pipeline

1. **Load data in**
2. **Build PyTorch class of hypothesized model form**
   A. Specify trainable coefficients in initializer
   B. Specify model form in forward()
3. **Set up optimizer**
4. **Train model**
5. **Visualize results**

Step 4: Train model

```python
# Training loop
num_epochs = 5000
for epoch in range(num_epochs):
    # Forward pass
    outputs = model(x_data)
    loss = criterion(outputs, y_data)

    # Backward pass and optimization
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    if (epoch+1) % 1000 == 0:
        print(f'Epoch [{epoch+1}/{num_epochs}], Loss: {loss.item():.4f}')
```

No need to modify!
Same for all (basic) models

Step 4: How to save (partially) trained model?

```
# Define a function to save intermediate results
def save_intermediate_results(filename, model, optimizer, epoch, loss):
  data = {
      'model_state_dict': model.state_dict(),
      'optimizer_state_dict': optimizer.state_dict(),
      'epoch': epoch,
      'loss': loss,
  }
  with open(filename, 'wb') as f:
    pickle.dump(data, f)
```

```
# Example usage: save intermediate results every 1000 epochs
for epoch in range(num_epochs):
  # ... (your forward pass, backward pass, and optimization) ...

  if (epoch+1) % 1000 == 0:
    print(f'Epoch [{epoch+1}/{num_epochs}], Loss: {loss.item():.4f}')
    save_intermediate_results('intermediate_results.pkl', model, optimizer, epoch, loss)
```

Step 4: How to load (partially) trained model?

```python
# Define a function to load intermediate results
def load_intermediate_results(filename):
  with open(filename, 'rb') as f:
    data = pickle.load(f)
  return data['model_state_dict'], data['optimizer_state_dict'], data['epoch'], data['loss']
```

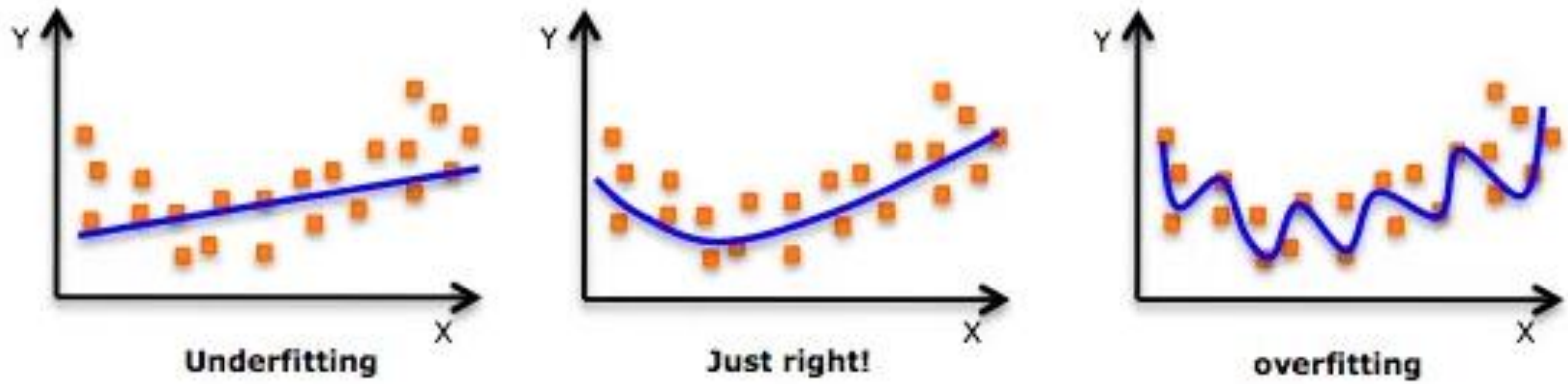
```python
# Example usage: load intermediate results
try:
  model_state_dict, optimizer_state_dict, loaded_epoch, loaded_loss = load_intermediate_results('intermediate_results.pkl')
  model.load_state_dict(model_state_dict)
  optimizer.load_state_dict(optimizer_state_dict)
  print(f'Loaded checkpoint from epoch {loaded_epoch} with loss {loaded_loss}')
  # Resume training from the loaded epoch
  start_epoch = loaded_epoch + 1
except FileNotFoundError:
  print('No checkpoint found, starting training from scratch.')
  start_epoch = 0
```

Add to the beginning of the training loop to load intermediate results, if they're present

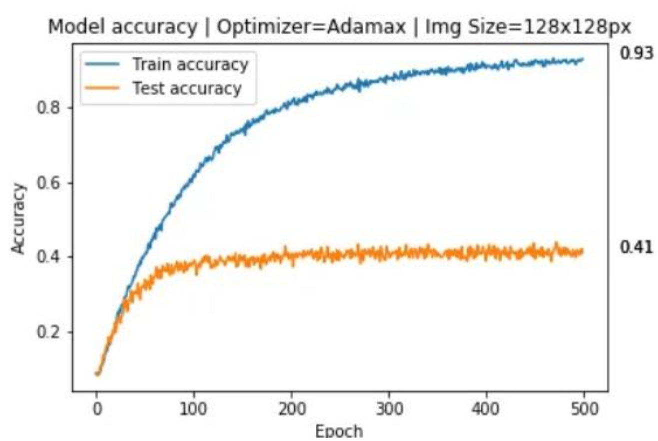
PyTorch Modeling Pipeline

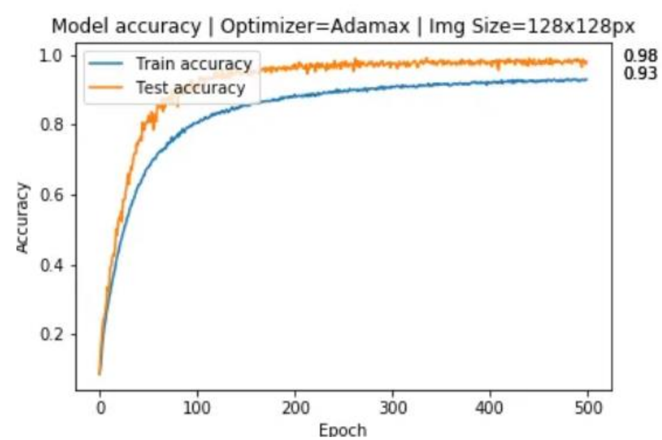
1. **Load data in**
2. **Build PyTorch class of hypothesized model form**
   A. Specify trainable coefficients in initializer
   B. Specify model form in forward()
3. **Set up optimizer**
4. **Train model**
5. **Visualize results**

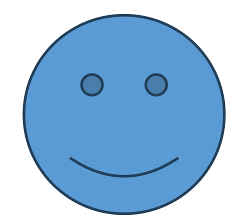
Step 5: Checking for overfitting



Step 1: Split data into training data and some held out test data (80/20 split)
Step 2: Check that accuracy is similar between two datasets







Rest of Class

I’ll step through the 5 steps using the HW for those who haven’t been able to get help at OH

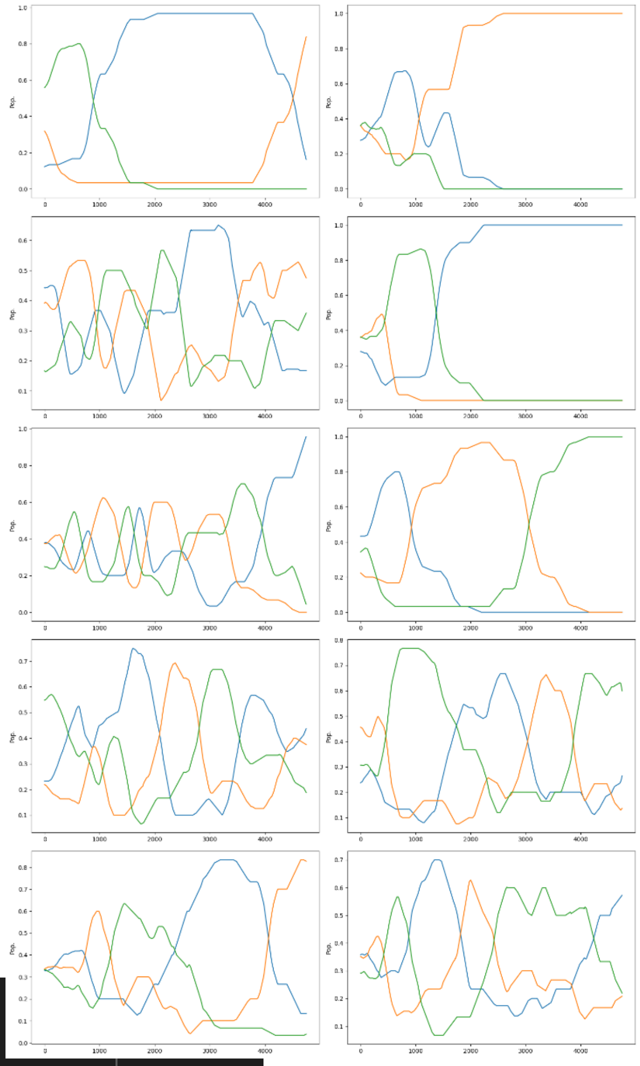
Others, feel free to complete ENM1050/Code examples/Lecture_16.ipynb to get started with MLPs

PyTorch Modeling Pipeline

1. **Load data in**
2. **Build PyTorch class of hypothesized model form**
   A. Specify trainable coefficients in initializer
   B. Specify model form in forward()
3. **Set up optimizer**
4. **Train model**
5. **Visualize results**

Step 1. Load data in

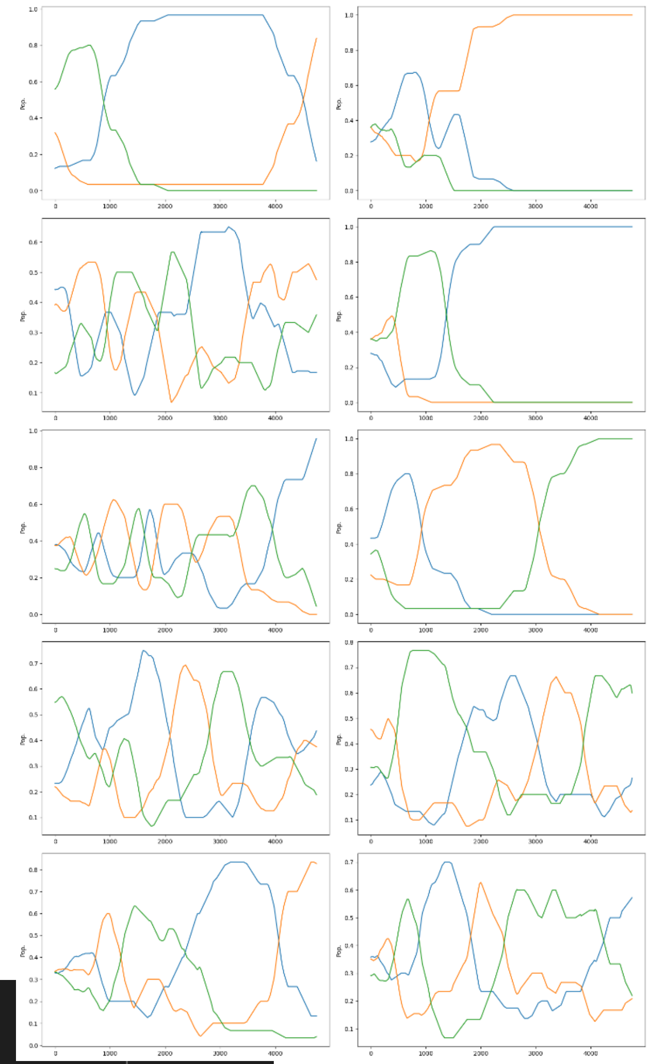
- Open ENM1050/Pytorch_Model_Fitting_Tutorial.ipynb
- Code plots smoothed data as **smoothed_sim**
- Save smoothed data into a list



```
# Plot data on each subplot
smoothedDataList = []
for idx, sim in enumerate(dataList[:10]):
    row = idx // 2
    col = idx % 2
    smoothed_sim = np.apply_along_axis(moving_average, 0, sim)  # Apply smoothing filter
    smoothedDataList.append(smoothed_sim)
    axs[row, col].plot(smoothed_sim)
    axs[row, col].set_ylabel('Pop.')
```

Step 1. Load data in

- Open ENM1050/Pytorch_Model_Fitting_Tutorial.ipynb
- Code plots smoothed data as **smoothed_sim**
- Save smoothed data into a list



```
# Plot data on each subplot
smoothedDataList = []
for idx, sim in enumerate(dataList[:10]):
    row = idx // 2
    col = idx % 2
    smoothed_sim = np.apply_along_axis(moving_average, 0, sim)  # Apply smoothing filter
    smoothedDataList.append(smoothed_sim)
    axs[row, col].plot(smoothed_sim)
    axs[row, col].set_ylabel('Pop.')
```

Step 1. Load data in

- Take data from list and put into a pytorch tensor

```
[19] # Calculate derivatives using a finite difference
     x_torch = torch.tensor(np.array(smoothedDataList))
     dt = 1.0
     dxdt_torch = torch.diff(x_torch,axis=1)/dt

     # Check that data is the same shape
     print(x_torch.shape,dxdt_torch.shape)

     torch.Size([10, 4751, 3]) torch.Size([10, 4750, 3])
```

- Because data is the wrong shape (why is it off by one?) we need to throw away a data point

```
[20] #throw away last point to make dxdt the same size as x
     x_torch = x_torch[:,:-1,:]

     # Check that data is the same shape
     print(x_torch.shape,dxdt_torch.shape)

     torch.Size([10, 4750, 3]) torch.Size([10, 4750, 3])
```

PyTorch Modeling Pipeline

1. Load data in
2. **Build PyTorch class of hypothesized model form**
   A. Specify trainable coefficients in initializer
   B. Specify model form in forward()
3. Set up optimizer
4. Train model
5. Visualize results

Step 2. Build pytorch model

- To get started, copy the example code we went over last Monday, lotkaFit.py

```
class PredatorPreyModel(nn.Module):
    def __init__(self):
        super(PredatorPreyModel, self).__init__()
        # Define whatever model parameters you want to add.
        # Don't forget to set 'requires_grad=True' for trainable parameters
        self.params = nn.parameter.Parameter(torch.tensor([1.0,1.0,1.0,1.0], requires_grad=True))
        # self.params = nn.parameter.Parameter(torch.tensor([1.0,0.1,-1.5,0.75], requires_grad=True))



    def forward(self, x):
        u = x.index_select(-1,torch.tensor(0))
        v = x.index_select(-1,torch.tensor(1))

        xdot =  self.params[0]*u - self.params[1]*u*v
        ydot = -self.params[2]*v + self.params[3]*u*v

        return torch.concatenate([xdot,ydot],dim=-1)
```

This gives us something close, but it is set up for a single predator/prey system (2 unknowns) – we need to modify for 3 (rock/paper/scissors)

Step 2. Build pytorch model

- To get started, copy the example code we went over last Monday, lotkaFit.py

```
class PredatorPreyModel(nn.Module):
    def __init__(self):
        super(PredatorPreyModel, self).__init__()
        # Define whatever model parameters you want to add.
        # Don't forget to set 'requires_grad=True' for trainable parameters
        self.params = nn.parameter.Parameter(torch.tensor([1.0,1.0,1.0,1.0], requires_grad=True))
        # self.params = nn.parameter.Parameter(torch.tensor([1.0,0.1,-1.5,0.75], requires_grad=True))



    def forward(self, x):
        u = x.index_select(-1,torch.tensor(0))
        v = x.index_select(-1,torch.tensor(1))

        xdot =  self.params[0]*u - self.params[1]*u*v
        ydot = -self.params[2]*v + self.params[3]*u*v

        return torch.concatenate([xdot,ydot],dim=-1)
```

This gives us something close, but it is set up for a single predator/prey system (2 unknowns) – we need to modify for 3 (rock/paper/scissors)

Step 2. Build pytorch model

```
[10] from torch import nn
     from torch import optim
     import torch

     class PredatorPreyModel(nn.Module):
         def __init__(self):
             super(PredatorPreyModel, self).__init__()
             # Define whatever model parameters you want to add.
             # Don't forget to set 'requires_grad=True' for trainable parameters
             self.params = nn.parameter.Parameter(torch.tensor([1.0,1.0,1.0,1.0,1.0,1.0], requires_grad=True))
             # self.params = nn.parameter.Parameter(torch.tensor([1.0,0.1,-1.5,0.75], requires_grad=True))



         def forward(self, x):
             u = x.index_select(-1,torch.tensor(0))
             v = x.index_select(-1,torch.tensor(1))
             w = x.index_select(-1,torch.tensor(2))

             xdot =  self.params[0]*u*v + self.params[1]*u*w
             ydot =  self.params[2]*v*u + self.params[3]*v*w
             zdot =  self.params[4]*w*u + self.params[5]*w*v

             return torch.concatenate([xdot,ydot,zdot],dim=-1)
```

Modify to 6 parameters

Pull out rock, paper <u>and</u> scissor vars

Change model form

Output all 3 vars

This gives us something close, but it is set up for a single predator/prey system (2 unknowns) – we need to modify for 3 (rock/paper/scissors)

PyTorch Modeling Pipeline

1. **Load data in**
2. **Build PyTorch class of hypothesized model form**
    A. Specify trainable coefficients in initializer
    B. Specify model form in forward()
3. **Set up optimizer**
4. **Train model**
5. **Visualize results**

Step 3: Set up optimizer

```
# Instantiate the custom layer
model = PredatorPreyModel()

# Define the loss function and optimizer
criterion = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=0.01)
```

Key strategy!
Use the loss to steer the model for the RHS of
dx/dt = f(x)
Toward the finite differenced data
f(x) = (x[i+1] – x[i])/dt

PyTorch Modeling Pipeline

1. **Load data in**
2. **Build PyTorch class of hypothesized model form**
   A. Specify trainable coefficients in initializer
   B. Specify model form in forward()
3. **Set up optimizer**
4. **Train model**
5. **Visualize results**

Step 4. Train model

```
num_epochs = 10000
for epoch in range(num_epochs):
    # Forward pass
    modeloutput = model(x_torch)
    loss = criterion(modeloutput, dxdt_torch)

    # Backward pass and optimization
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    if epoch % 1000 == 0:
        print('Epoch {}, Loss {}'.format(epoch, loss.item()), model.params.detach().numpy())
```

```
Epoch 0, Loss 0.02856928091471473 [0.99 0.99 0.99 0.99 0.99 0.99]
Epoch 1000, Loss 2.459292527010483e-07 [-0.00364322  0.00379141  0.0034043  -0.00342114 -0.00352881  0.00371535]
Epoch 2000, Loss 2.4592925253375095e-07 [-0.00364326  0.00379145  0.00340434 -0.00342118 -0.00352907  0.00371563]
Epoch 3000, Loss 2.4592925253369266e-07 [-0.00364327  0.00379145  0.00340434 -0.00342118 -0.00352907  0.00371563]
Epoch 4000, Loss 2.459292525336738e-07 [-0.00364327  0.00379145  0.00340434 -0.00342118 -0.00352907  0.00371563]
Epoch 5000, Loss 2.4592925253366895e-07 [-0.00364327  0.00379145  0.00340434 -0.00342118 -0.00352907  0.00371563]
Epoch 6000, Loss 2.4592925253366746e-07 [-0.00364327  0.00379145  0.00340434 -0.00342118 -0.00352907  0.00371563]
Epoch 7000, Loss 2.4592925253366704e-07 [-0.00364327  0.00379145  0.00340434 -0.00342118 -0.00352907  0.00371563]
Epoch 8000, Loss 2.459292525336671e-07 [-0.00364327  0.00379145  0.00340434 -0.00342118 -0.00352907  0.00371563]
Epoch 9000, Loss 2.45929252533667e-07 [-0.00364327  0.00379145  0.00340434 -0.00342118 -0.00352907  0.00371563]
```

As training progresses, take a look at how the parameters evolve

Do these make sense?

PyTorch Modeling Pipeline

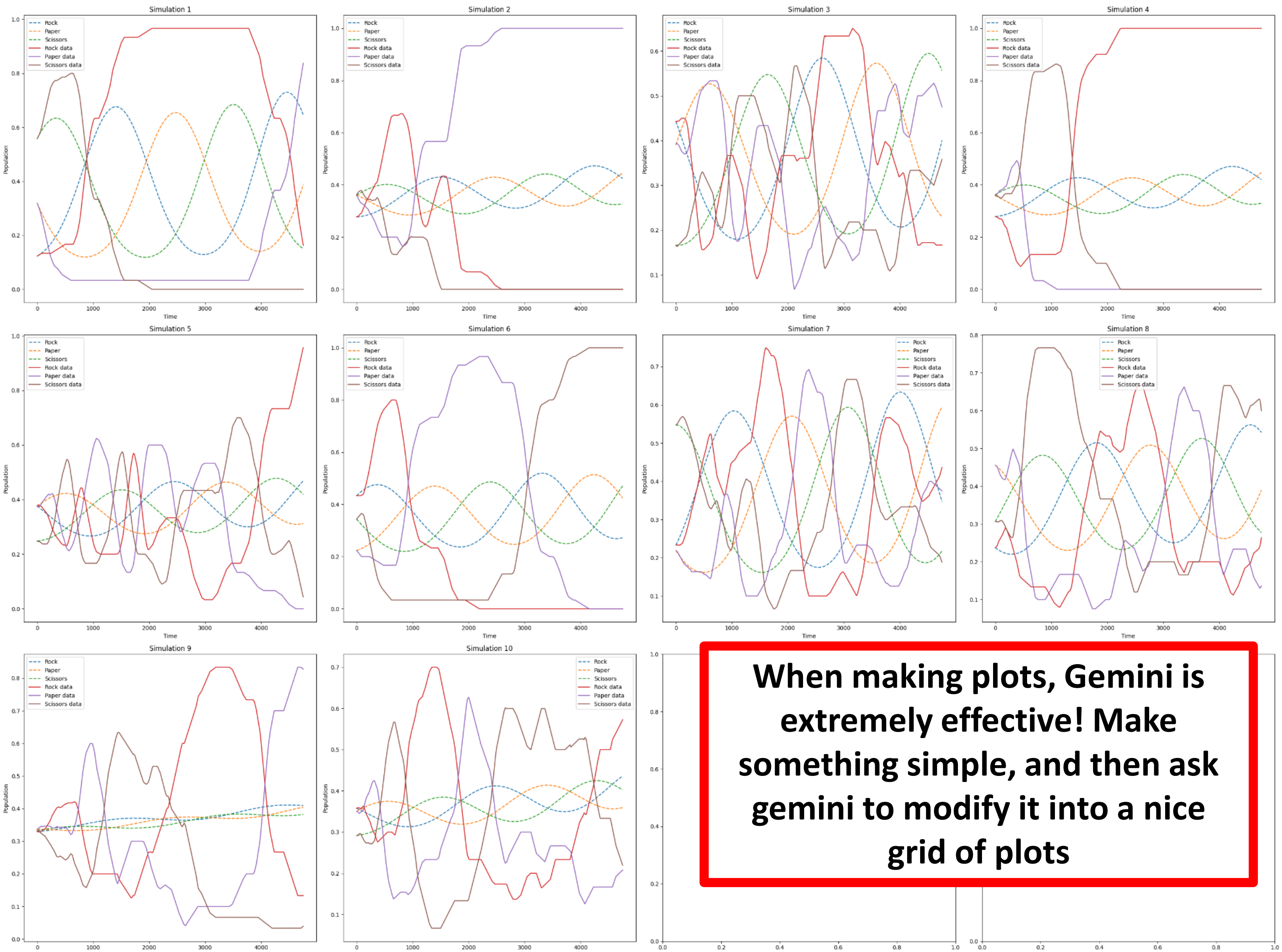
1. **Load data in**
2. **Build PyTorch class of hypothesized model form**
   A. Specify trainable coefficients in initializer
   B. Specify model form in forward()
3. **Set up optimizer**
4. **Train model**
5. **Visualize results**

Step 5. Visualize results

Take the example code from lotkaFitter.py to use odeint to solve the learned equation

```
from scipy.integrate import odeint
fitModelList = []
def rhs_func(y, t, model):
    with torch.no_grad():  # Disable gradient calculations for odeint
        return model(torch.tensor(y)).detach().numpy()  # Assuming model outputs a PyTorch tensor

# Get solution for the first solution in dataset (dataindex = 0)
dataIndex = 0

# Get initial conditions from dataset
x0 = smoothedDataList[dataIndex][0,0]
y0 = smoothedDataList[dataIndex][0,1]
z0 = smoothedDataList[dataIndex][0,2]
xinit = smoothedDataList[dataIndex][0,:]

# Time points for the solution
t = np.linspace(0, 4751,4751)

# Solve the ODE using odeint
solution = odeint(rhs_func, xinit, t, args=(model,))
```

Step 5. Visualize results



When making plots, Gemini is extremely effective! Make something simple, and then ask gemini to modify it into a nice grid of plots

Step 5. Visualize results

```python
# Create a 2x5 grid of plots
fig, axs = plt.subplots(3,4, figsize=(32,24))  # Changed to 2 rows, 5 columns

# Flatten the axs array for easier iteration
axs = axs.flatten()

# Plot data on each subplot
for dataIndex in range(10):
    # Initial conditions
    x0 = smoothedDataList[dataIndex][0,0]
    y0 = smoothedDataList[dataIndex][0,1]
    z0 = smoothedDataList[dataIndex][0,2]
    xinit = smoothedDataList[dataIndex][0,:]
    # Time points for the solution
    t = np.linspace(0, 4751,4751)

    # Solve the ODE using odeint
    solution = odeint(rhs_func, xinit, t, args=(model,))
    # solution = odeint(rhs_func, [x0,y0,z0], t, args=(model,))

    # Extract the results
    x_population = solution[:, 0]
    y_population = solution[:, 1]
    z_population = solution[:, 2]

    # Plot on the current subplot
    axs[dataIndex].plot(t, x_population, '--', label='Rock')
    axs[dataIndex].plot(t, y_population, '--', label='Paper')
    axs[dataIndex].plot(t, z_population, '--', label='Scissors')
    axs[dataIndex].plot(t, smoothedDataList[dataIndex][:, 0], label='Rock data')
    axs[dataIndex].plot(t, smoothedDataList[dataIndex][:, 1], label='Paper data')
    axs[dataIndex].plot(t, smoothedDataList[dataIndex][:, 2], label='Scissors data')

    axs[dataIndex].set_xlabel('Time')
    axs[dataIndex].set_ylabel('Population')
    axs[dataIndex].set_title(f'Simulation {dataIndex + 1}')  # Add title for each subplot
    axs[dataIndex].legend()

# Adjust layout
plt.tight_layout()

# Display the figure
plt.show()
```

When making plots, Gemini is extremely effective! Make something simple, and then ask gemini to modify it into a nice grid of plots

In-Class 11: MLPs and HW help

At this point, everyone in the class should be comfortable fitting a pytorch model to a dataset. If you aren't please ask for help!

- **Moving forward, come to OH early and often.**
- **Now that we're over the hump with using pytorch, we will do several exercises so that things start feeling downhill from here.**

No in-class submission today.