COMPSCI 389
Introduction to Machine Learning

PyTorch and Overfitting

Prof. Philip S. Thomas (pthomas@cs.umass.edu)

Note: This presentation covers (and provides additional context/information regarding)
`21 Introduction to PyTorch.ipynb`

Autograd

- Can be slow because it executes Python code.
- Is designed for differentiating arbitrary code
  - It does not have extra functionality for machine learning

Deep Learning Libraries

- There are many deep learning libraries that extend autograd to:
  - Leverage low-level compiled code for faster runtimes.
  - Enable forward and backwards passes on the GPU rather than CPU (more on this later).
  - Have built-in implementations of
    - Common loss functions
    - Common activation functions
    - Common network layers
      - Fully connected feed-forward
      - Convolutional layers
      - Pooling layers
      - Etc.

Deep Learning Libraries

- PyTorch
  - The most commonly used today.
  - What we will use in class.
- Tensorflow
  - Produced and maintained by Google
  - Integrates nicely with Google's cloud computing platforms
  - Steeper learning curve and more verbose syntax
- Keras, Caffe, MXNet, etc.
  - Many less popular alternatives

PyTorch

You can install PyTorch with:

```
pip install torch torchvision
```

We will use the following imports:

```
# New to this topic:
import torch
import torch.nn as nn           # For defining our neural network model
import torch.optim as optim     # For training the model using data
from torch.utils.data import TensorDataset, DataLoader  # For making mini-batches
```

Defining a Neural Network Architecture
Defining a Parametric Model

- Extend the `nn.Module` base class
  - The base class provides functionality for tracking trainable parameters (and their gradients), moving parameters to the GPU, saving and loading models, etc.
- Implement two functions:
  - `__init__(self)`: Define the different layers (number of units, number of inputs) and different activation functions that will be used.
  - `forward(self, x)`: Perform a forward pass on input x.
- You do *not* need to implement any gradients or the backwards pass!
  - PyTorch uses reverse mode automatic differentiation to automatically compute gradients.

Note: This model is bigger than needed for the GPA prediction problem. This allows us to more easily compare runtimes later, and to show a phenomenon called “overfitting”.

```python
class FullyConnectedNetwork(nn.Module):
    def __init__(self):
        # First call the nn.Module constructor to initialize other parts of the model. Always do this first.
        super(FullyConnectedNetwork, self).__init__()

        # Define layers. The lines below create the layers (memory is allocated for the weights here).
        self.fc1 = nn.Linear(9, 1024)  # First hidden layer with 1024 neurons and 9 inputs.
        self.fc2 = nn.Linear(1024, 512) # Second hidden layer with 512 neurons and 1024 inputs.
        self.fc3 = nn.Linear(512, 128)  # Third hidden layer with 128 neurons and 512 inputs.
        self.fc4 = nn.Linear(128, 1)   # Output layer with 1 neuron and 128 inputs.

        # Define activation function.
        self.relu = nn.ReLU()

    def forward(self, x):
        # Pass data through the network
        x = self.relu(self.fc1(x))
        x = self.relu(self.fc2(x))
        x = self.relu(self.fc3(x))
        x = self.fc4(x)                 # No activation after the output layer
        return x
```

`nn.Linear` represents a linear parametric model with no basis. That is, each unit is a perceptron without an activation function.

We can now create an instance of this model:

```
# Create an instance of the network
net = FullyConnectedNetwork()

# The network structure is printed as a sanity check
print(net)
```

```
FullyConnectedNetwork(
  (fc1): Linear(in_features=9, out_features=1024, bias=True)
  (fc2): Linear(in_features=1024, out_features=512, bias=True)
  (fc3): Linear(in_features=512, out_features=128, bias=True)
  (fc4): Linear(in_features=128, out_features=1, bias=True)
  (relu): ReLU()
)
```

`bias=True` indicates that each perceptron includes an extra feature that is always equal to 1 (and hence one extra weight beyond the number of outputs from the previous layer). This is what we discussed previously when we talked about appending a 1 to the columns of a data set to implement the "y-intercept" in linear regression. For perceptrons and neural networks, this extra weight is called the **bias**.

Next, let's load the GPA data, split it into training and testing, and standardize it.

```
df = pd.read_csv("https://people.cs.umass.edu/~pthomas/courses/COMPSCI_389/GPA.csv", delimiter=',') # Read GPA.
#df = pd.read_csv("data/GPA.csv", delimiter=',')

# We already loaded X and y, but do it again as a reminder
X = df.iloc[:, :-1]
y = df.iloc[:, -1]

# 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, shuffle=True)

# Standardize the features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train) # This sets the min/max values from the training data (without looking
X_test = scaler.transform(X_test)       # This uses the min/max scaling values chosen during training! (transfo
```

✓ 0.1s Python

PyTorch has its own objects for storing data, called PyTorch tensors. These are simply multidimensional arrays. Let's convert our data to these tensor objects. Note that the `tensor` constructor is not compatible with `pandas.Series` objects, so we call `y_train.values` and `y_test.values` to convert these to `numpy.ndarray` objects.

```python
# Convert data to PyTorch tensors
X_train_tensor = torch.tensor(X_train,        dtype=torch.float32)
y_train_tensor = torch.tensor(y_train.values, dtype=torch.float32).view(-1,1)
X_test_tensor  = torch.tensor(X_test,         dtype=torch.float32)
y_test_tensor  = torch.tensor(y_test.values,  dtype=torch.float32).view(-1,1)
```

✓ 0.0s Python

Loss Function

- PyTorch has many built-in loss functions, including MSE:

```
loss_function = nn.MSELoss()
```

Optimizer

- PyTorch has many built-in loss optimizers, including gradient descent (SGD), and Adam (SGD with a specific adaptive step size method).
  - Several optimizers are discussed in the Jupyter notebook.
  - Adam is the most common, and what we will use.

```
optimizer = optim.Adam(net.parameters())
```

`net.parameters()` contains the weights, and after backwards passes will also contain the gradient information. The optimizer uses this gradient information to update the weights.

```
epochs = 100                          # The number of epochs to run
for epoch in range(epochs):
```

```
    # Forward pass
    y_pred = net(X_train_tensor)
```

Perform a forward pass of the parametric model for each training point.

```
epochs = 100                          # The number of epochs to run
for epoch in range(epochs):
```

```
    # Forward pass
    y_pred = net(X_train_tensor)

    # Compute loss
    loss = loss_function(y_pred, y_train_tensor)
```

Perform a forward pass of the parametric model for each training point.

Perform a forward pass of the loss function.

```
epochs = 100                          # The number of epochs to run
for epoch in range(epochs):
```

```
    # Forward pass
    y_pred = net(X_train_tensor)

    # Compute loss
    loss = loss_function(y_pred, y_train_tensor)

    # Backward pass and optimize
    loss.backward()
```

Perform a backwards pass, computing the gradient of loss w.r.t. each model parameter (each weight)

```
epochs = 100                          # The number of epochs to run
for epoch in range(epochs):
```

```
    # Forward pass
    y_pred = net(X_train_tensor)

    # Compute loss
    loss = loss_function(y_pred, y_train_tensor)

    # Backward pass and optimize
    loss.backward()
    optimizer.step()
```

Update the weights using gradient descent with adaptive step size.

```
epochs = 100                          # The number of epochs to run
for epoch in range(epochs):
```

```
    # Forward pass
    y_pred = net(X_train_tensor)

    # Compute loss
    loss = loss_function(y_pred, y_train_tensor)

    # Backward pass and optimize
    loss.backward()
    optimizer.step()

    # Print statistics
    if epoch % 10 == 0:
        print(f'Epoch [{epoch}/{epochs}], Loss: {loss.item():.4f}')
```

Print the loss on the training set.

A note about `backward()`

- Each model parameter has a `.grad` attribute storing the gradient of the loss w.r.t. that parameter.
- Sometimes many gradients are computed before one step
  - This isn't something we have discussed
- To accommodate this, `loss.backward()` adds the derivative of the loss w.r.t. the model parameter to whatever is already stored in `.grad`
- So, after each gradient update, we need to set `.grad` back to zero

```python
epochs = 100                          # The number of epochs to run
for epoch in range(epochs):
    # Zero the gradients
    optimizer.zero_grad()

    # Forward pass
    y_pred = net(X_train_tensor)

    # Compute loss
    loss = loss_function(y_pred, y_train_tensor)

    # Backward pass and optimize
    loss.backward()
    optimizer.step()

    # Print statistics
    if epoch % 10 == 0:
        print(f'Epoch [{epoch}/{epochs}], Loss: {loss.item():.4f}')
```

Set all the parameter gradients to zero.

Results

- **Runtime**: 36.9 seconds on my desktop.
  - This is training a much more complicated model than the simple linear model we trained using autograd.
- Next, evaluate on the test set.
  - Note: When evaluating the model we do not need to store intermediate values during a forwards pass.
  - Torch.no_grad() tells PyTorch not to store extra information during a forwards pass.

```
Epoch [0/100], Loss: 8.3346
Epoch [10/100], Loss: 1.3681
Epoch [20/100], Loss: 0.9073
Epoch [30/100], Loss: 0.7348
Epoch [40/100], Loss: 0.7225
Epoch [50/100], Loss: 0.6655
Epoch [60/100], Loss: 0.6407
Epoch [70/100], Loss: 0.6143
Epoch [80/100], Loss: 0.5949
Epoch [90/100], Loss: 0.5819
```

```python
# Evaluate the model with test data
with torch.no_grad():
    y_pred_test = net(X_test_tensor)
    test_loss = loss_function(y_pred_test, y_test_tensor)
    print(f'Test Loss: {test_loss.item():.4f}')
```

```
Test Loss: 0.5778
```

Runtime

- 32.7 seconds is a significant training time for such a small data set.
- My work desktop has an Intel i9-9900k with 16 cores (CPU).
- It also has an RTX 2070 GPU
  - This has 2304 cores! (An RTX 4090 has 18,432 CUDA cores and 512 special “Tensor” cores)
- These GPU cores are limited in comparison to CPU cores.
  - No branch prediction
  - Limited cache
  - Shorter pipeline (typically)
    - Slower clock (1.605 GHz vs 5 MHz)
  - Not designed for parallel processing (many processes running at once)
- Designed to perform many simple operations like dot products efficiently and in parallel
  - These operations are useful for displaying graphics (e.g., applying simple functions to each pixel on the screen between every frame, changing things like lighting)
  - They are also useful for ML! Running an ANN means computing a *lot* of dot products (and some non-linearities).

PyTorch and CUDA

- The Jupyter Notebook associated with this lecture includes instructions for installing NVIDIA CUDA, which is necessary to use PyTorch to train models on the GPU.
- Note that PyTorch is compatible with CUDA 12.1, not the latest CUDA 12.4.
- Notice that you need to custom-install the version of PyTorch that is compatible with the version of CUDA that you installed.
- Some ML libraries and tools are only compatible with specific versions of CUDA, and so you may need to use different versions of CUDA and PyTorch for different applications.

Training on the GPU

- Check if CUDA (GPU support) is available.

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

- Move the network to the GPU.

```
net.to(device)
```

- Move the training data to the GPU.

```
X_train_tensor = X_train_tensor.to(device)
```

```
y_train_tensor = y_train_tensor.to(device)
```

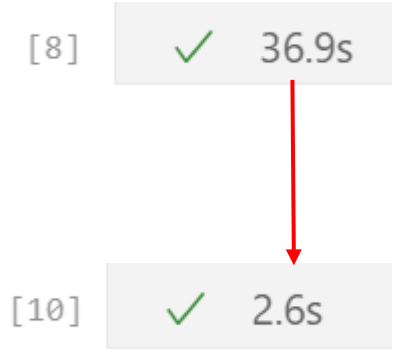
- When we are done training the model, we can then move it back to the CPU:

```
net.to('cpu')
```

```
net = FullyConnectedNetwork()                    # Create a new network to train from scratch
optimizer = optim.Adam(net.parameters())         # Create the optimizer for this network
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")   # Check if CUDA (GPU) available
display(device)                                  # Confirm that the GPU is being used

net.to(device)                                   # Move the network to GPU if available
X_train_tensor = X_train_tensor.to(device)       # Also move the tensors to the chosen device
y_train_tensor = y_train_tensor.to(device)

epochs = 100                                     # Number of epochs
for epoch in range(epochs):
    optimizer.zero_grad()                        # Zero the gradients
    y_pred = net(X_train_tensor)                 # Forward pass
    loss = loss_function(y_pred, y_train_tensor)    # Compute the loss for printing/plotting
    loss.backward()                              # Backwards pass
    optimizer.step()                             # Update the weights using the optimizer
    if epoch % 10 == 0:                          # Print statistics
        print(f'Epoch [{epoch}/{epochs}], Loss: {loss.item():.4f}')

net.to('cpu')                                    # Move the model back to the CPU
```

Move the model back to the CPU if you will run it or manipulate it on the CPU (e.g., saving the model/weights to a file). Leave on the GPU if you will only run it on the GPU.

```
device(type='cuda')

Epoch [0/100], Loss: 8.4130
Epoch [10/100], Loss: 1.5344
Epoch [20/100], Loss: 0.8190
Epoch [30/100], Loss: 0.7992
Epoch [40/100], Loss: 0.7053
Epoch [50/100], Loss: 0.6705
Epoch [60/100], Loss: 0.6278
Epoch [70/100], Loss: 0.6009
Epoch [80/100], Loss: 0.5837
Epoch [90/100], Loss: 0.5744
```



Mini-Batches (GPU)

- Typically, mini-batches are computed on the CPU.
- They are then passed to the GPU to perform a gradient update.
- PyTorch's `DataLoader` object facilitates passing data efficiently (using multiple CPU cores) between the CPU and GPU.
- Often GPUs have limited memory, so entire data sets may not fit on the GPU.
  - RTX 2070: **8 GB memory**, 2304 CUDA cores, ~$350
  - RTX 4090: **24 GB memory**, 16384 CUDA cores, ~$1,500
  - A100: **80 GB memory**, 6912 CUDA cores, ~$8,000
    - Custom made for deep learning with large models.
- Mini-batches can be created using the PyTorch `DataLoader`.
  - `DataLoader` works with PyTorch's own data set object: `TensorDataset`.

Convert the training data into a TensorDataset, so that it is compatible with DataLoader.

Note: `batch_size = 100`

```
net = FullyConnectedNetwork()
optimizer = optim.Adam(net.parameters())
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
net.to(device)

# Create a TensorDataset and DataLoader
train_dataset = TensorDataset(X_train_tensor, y_train_tensor)
train_loader = DataLoader(dataset=train_dataset, batch_size=100, shuffle=True)

epochs = 100
for epoch in range(epochs):
    total_loss = 0.0    # To sum the loss over all batches
    num_batches = 0     # A lazy way to get the number of batches: count them
    for X_batch, y_batch in train_loader:                          # Iterate over mini-batches
        X_batch, y_batch = X_batch.to(device), y_batch.to(device)   # Move batches to GPU
        optimizer.zero_grad()
        y_pred = net(X_batch)
        loss = loss_function(y_pred, y_batch)
        total_loss += loss.item()
        num_batches += 1
        loss.backward()
        optimizer.step()

    # Calculate the average loss over all mini-batches in this epoch
    average_loss = total_loss / num_batches

    if epoch % 10 == 0: # Print statistics
        print(f'Epoch [{epoch}/{epochs}], Average Loss: {average_loss:.4f}')
net.to('cpu')   # Move the model back to CPU if needed
```

Update tracking of losses to compute the average loss per mini-batch each epoch

```
net = FullyConnectedNetwork()
optimizer = optim.Adam(net.parameters())
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
net.to(device)

# Create a TensorDataset and DataLoader
train_dataset = TensorDataset(X_train_tensor, y_train_tensor)
train_loader = DataLoader(dataset=train_dataset, batch_size=100, shuffle=True)

epochs = 100
for epoch in range(epochs):
    total_loss = 0.0    # To sum the loss over all batches
    num_batches = 0     # A lazy way to get the number of batches: count them
    for X_batch, y_batch in train_loader:                          # Iterate over mini-batches
        X_batch, y_batch = X_batch.to(device), y_batch.to(device)   # Move batches to GPU
        optimizer.zero_grad()
        y_pred = net(X_batch)
        loss = loss_function(y_pred, y_batch)
        total_loss += loss.item()
        num_batches += 1
        loss.backward()
        optimizer.step()

    # Calculate the average loss over all mini-batches in this epoch
    average_loss = total_loss / num_batches

    if epoch % 10 == 0: # Print statistics
        print(f'Epoch [{epoch}/{epochs}], Average Loss: {average_loss:.4f}')
net.to('cpu')   # Move the model back to CPU if needed
```

Loop over batches using the `DataLoader`, sending batches to the GPU.

```
net = FullyConnectedNetwork()
optimizer = optim.Adam(net.parameters())
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
net.to(device)

# Create a TensorDataset and DataLoader
train_dataset = TensorDataset(X_train_tensor, y_train_tensor)
train_loader = DataLoader(dataset=train_dataset, batch_size=100, shuffle=True)

epochs = 100
for epoch in range(epochs):
    total_loss = 0.0    # To sum the loss over all batches
    num_batches = 0     # A lazy way to get the number of batches: count them
    for X_batch, y_batch in train_loader:                          # Iterate over mini-batches
        X_batch, y_batch = X_batch.to(device), y_batch.to(device)   # Move batches to GPU
        optimizer.zero_grad()
        y_pred = net(X_batch)
        loss = loss_function(y_pred, y_batch)
        total_loss += loss.item()
        num_batches += 1
        loss.backward()
        optimizer.step()

    # Calculate the average loss over all mini-batches in this epoch
    average_loss = total_loss / num_batches

    if epoch % 10 == 0: # Print statistics
        print(f'Epoch [{epoch}/{epochs}], Average Loss: {average_loss:.4f}')
net.to('cpu')   # Move the model back to CPU if needed
```

```
net = FullyConnectedNetwork()
optimizer = optim.Adam(net.parameters())
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
net.to(device)

# Create a TensorDataset and DataLoader
train_dataset = TensorDataset(X_train_tensor, y_train_tensor)
train_loader = DataLoader(dataset=train_dataset, batch_size=100, shuffle=True)

epochs = 100
for epoch in range(epochs):
    total_loss = 0.0    # To sum the loss over all batches
    num_batches = 0     # A lazy way to get the number of batches: count them
    for X_batch, y_batch in train_loader:                          # Iterate over mini-batches
        X_batch, y_batch = X_batch.to(device), y_batch.to(device)   # Move batches to GPU
        optimizer.zero_grad()
        y_pred = net(X_batch)
        loss = loss_function(y_pred, y_batch)
        total_loss += loss.item()
        num_batches += 1
        loss.backward()
        optimizer.step()

    # Calculate the average loss over all mini-batches in this epoch
    average_loss = total_loss / num_batches

    if epoch % 10 == 0: # Print statistics
        print(f'Epoch [{epoch}/{epochs}], Average Loss: {average_loss:.4f}')
net.to('cpu')   # Move the model back to CPU if needed
```

We do **not** need to “remove” the batches from GPU memory. This is done automatically.

What do we expect to happen?

- Recall, 100 epochs before:

```
Epoch [0/100], Loss: 8.4130
Epoch [10/100], Loss: 1.5344
Epoch [20/100], Loss: 0.8190
Epoch [30/100], Loss: 0.7992
Epoch [40/100], Loss: 0.7053
Epoch [50/100], Loss: 0.6705
Epoch [60/100], Loss: 0.6278
Epoch [70/100], Loss: 0.6009
Epoch [80/100], Loss: 0.5837
Epoch [90/100], Loss: 0.5744
```

- Now we run 100 epochs using mini-batches of size 100

```
Epoch [0/100], Average Loss: 0.7102
Epoch [10/100], Average Loss: 0.5707
Epoch [20/100], Average Loss: 0.5621
Epoch [30/100], Average Loss: 0.5527
Epoch [40/100], Average Loss: 0.5417
Epoch [50/100], Average Loss: 0.5260
Epoch [60/100], Average Loss: 0.5058
Epoch [70/100], Average Loss: 0.4798
Epoch [80/100], Average Loss: 0.4524
Epoch [90/100], Average Loss: 0.4282
```

We have never seen sample MSEs so low!

Evaluate on the test set

- Remember to use `torch.no_grad()` for faster evaluation

```
# Evaluate the model with test data (optional)
with torch.no_grad():
    y_pred_test = net(X_test_tensor)
    test_loss = loss_function(y_pred_test, y_test_tensor)
    print(f'Test Loss: {test_loss.item():.4f}')
```

- Note:
  - The `net` model was moved back to the CPU.
  - The testing data is on the CPU.
  - We could have left the `net` model on the GPU and moved the testing data to the GPU.

```
Epoch [0/100], Average Loss: 0.7102
Epoch [10/100], Average Loss: 0.5707
Epoch [20/100], Average Loss: 0.5621
Epoch [30/100], Average Loss: 0.5527
Epoch [40/100], Average Loss: 0.5417
Epoch [50/100], Average Loss: 0.5260
Epoch [60/100], Average Loss: 0.5058
Epoch [70/100], Average Loss: 0.4798
Epoch [80/100], Average Loss: 0.4524
Epoch [90/100], Average Loss: 0.4282
```

```
Test Loss: 0.7296
```

No method has achieved a test loss so… **high**!

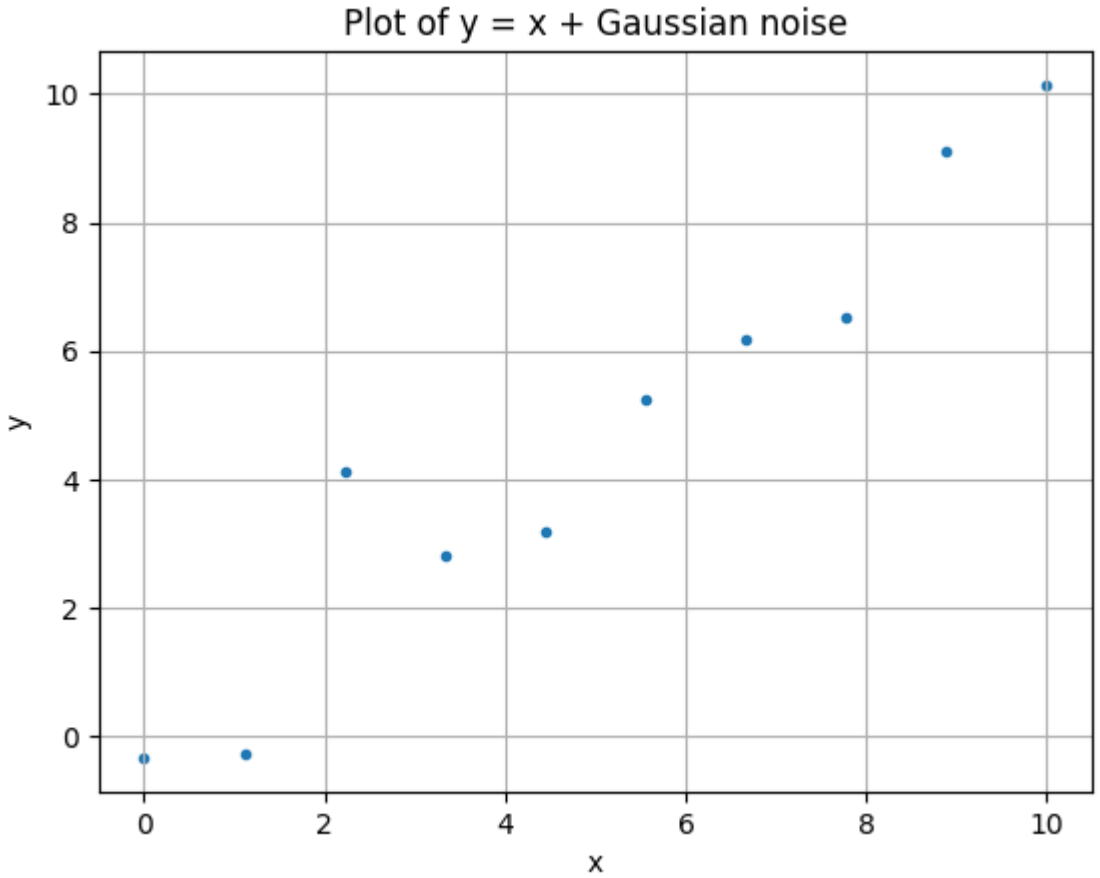
Question: Why is the test loss so much higher than the training loss?

Answer: The network has **overfit** to the training data.

Overfitting

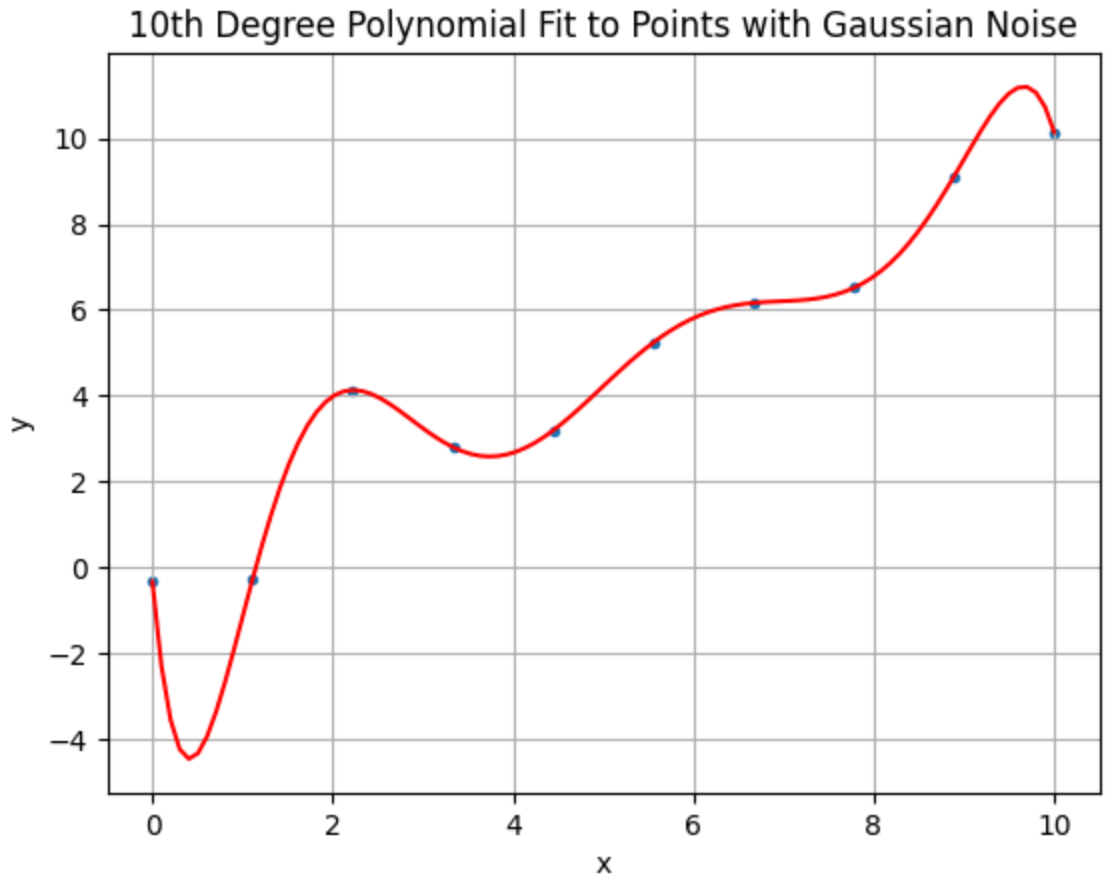
- Recall that the *training* error for *nearest neighbor* (NN) was zero, but the testing error was large.
  - NN essentially "memorized" the training data, and gave good predictions for the training data.
  - The model did not **generalize** to new inputs: it had high errors for points not in the training data.
- When this happens using parametric models, it is called **overfitting**.

10 points from $y = x + N(0,1)$



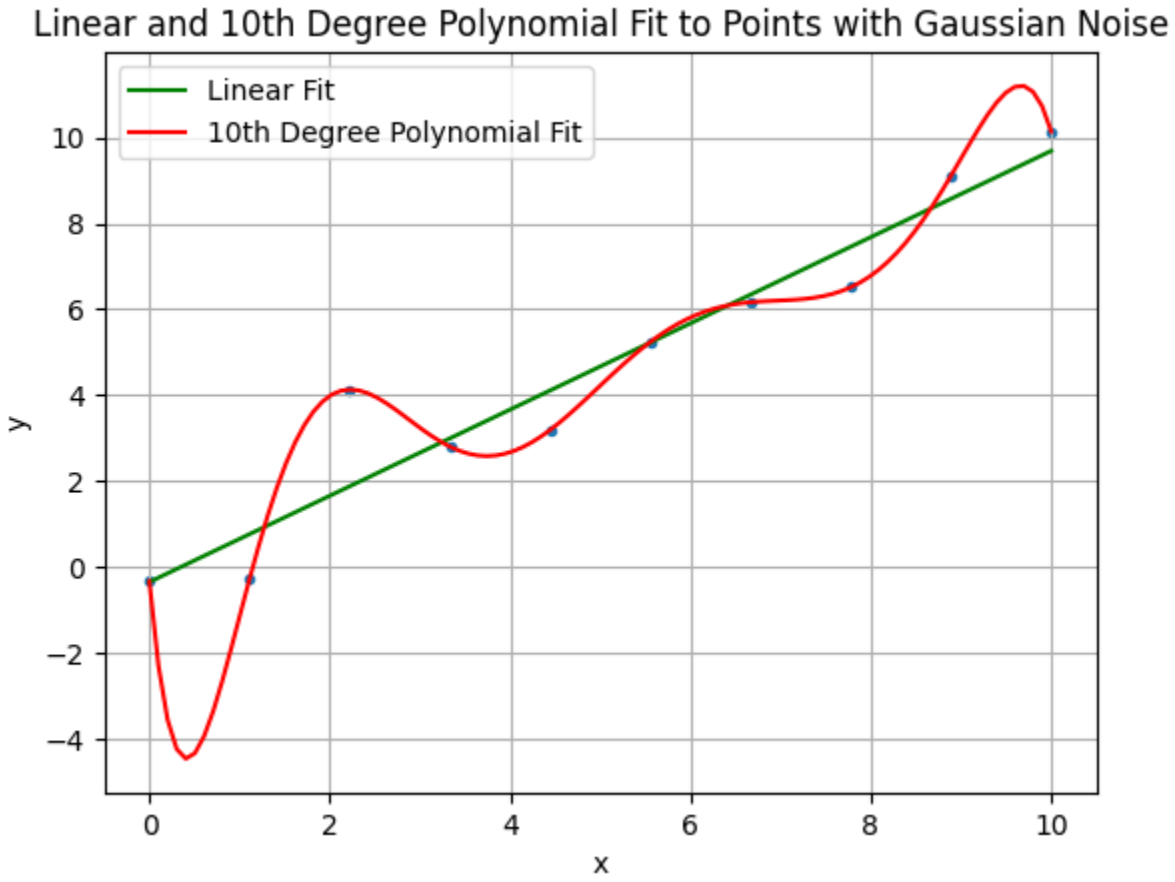
Least Squares fit, 10^{th} Degree Polynomial

- Linear parametric model using 10^{th} degree polynomial basis.
- The model perfectly predicts every training point!
- The model will have significant error for new points.



Least Squares fit, 10^{th} Degree Polynomial

- Linear parametric model using 10^{th} degree polynomial basis.
- The model perfectly predicts every training point!
- The model will have significant error for new points.
- A linear fit (with no basis) would provide better predictions for new points!

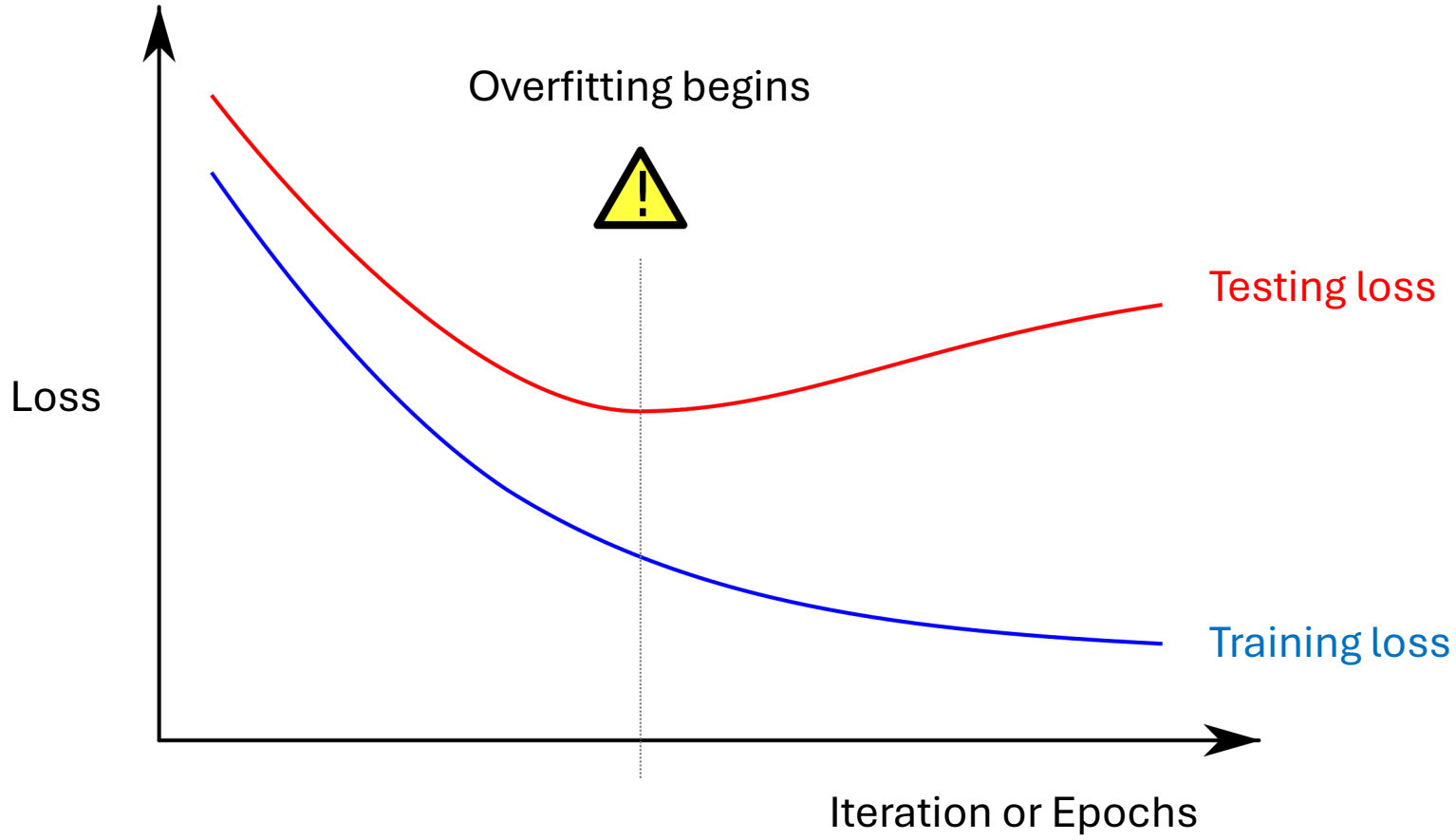


Overfitting

- When training parametric models with gradient descent, initially the loss decreases on the training **and** testing sets.
  - The model is learning general trends in the data that generalize to new points.
- Eventually, the model starts to learn specific trends in the training data that do **not** generalize to new points.
  - This typically results in lower loss on the training data, but higher loss (or no change in loss) for the testing data.

Plotting Training vs Testing Loss (General Case)

Idea: Stop training when the testing loss starts increasing.

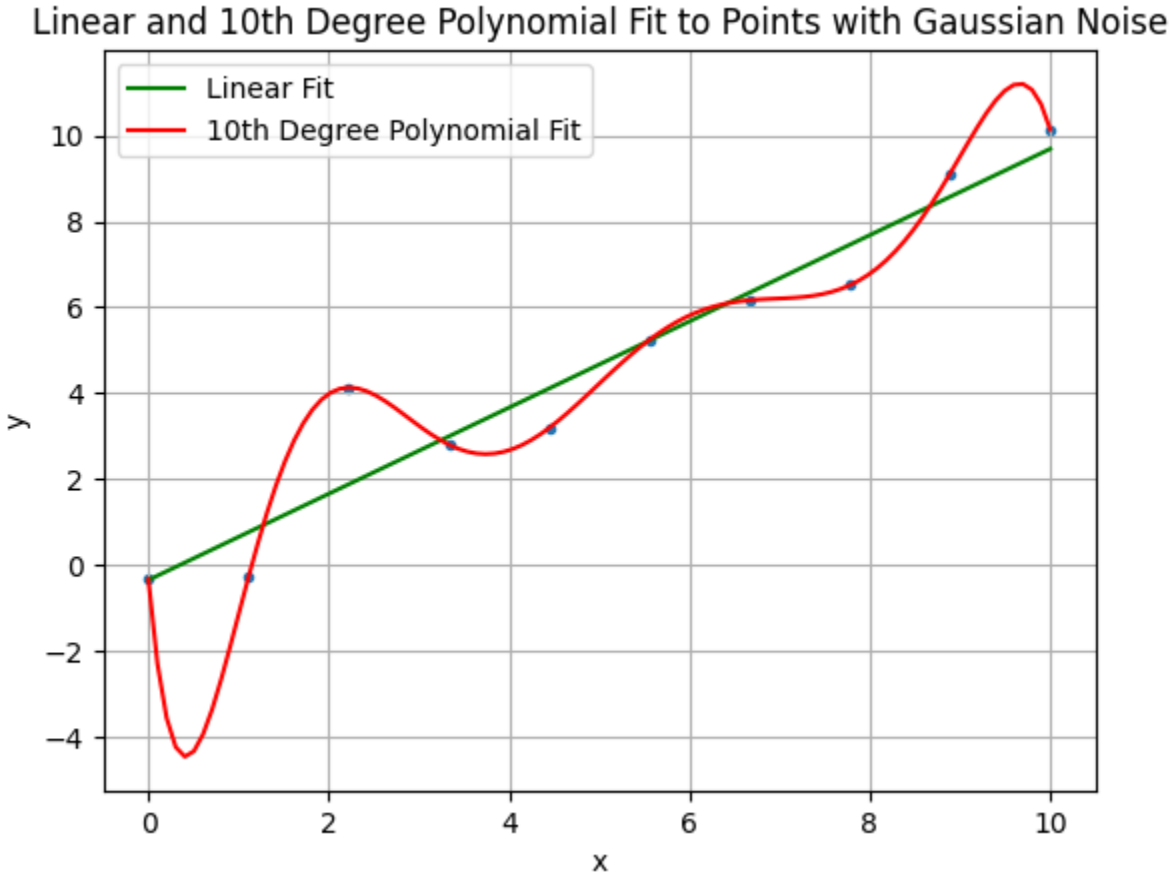


Plotting Training vs Testing Loss (GPA Data)

- With a *relatively* simple problem, overfitting begins within the first few epochs!

Overfitting and Model Complexity/Capacity

- Notice that we can't overfit this data using a line!
- The **model complexity** or **model capacity** refers to a parametric model's ability to represent general functions.
  - Models with higher complexity/capacity can represent more functions.
  - Models with higher complexity/capacity are more prone to over-fitting.



Avoiding Over-Fitting (Overview of Strategies)

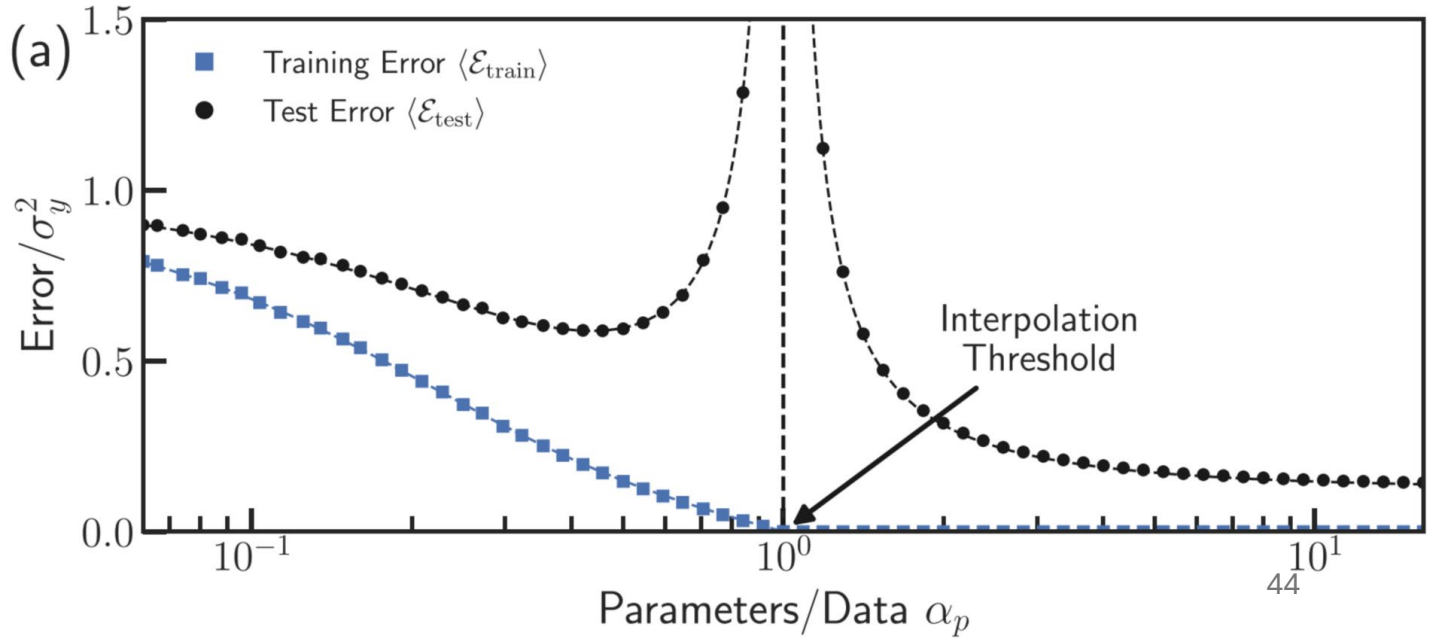
1. Early stopping: Stop training when testing error increases.
   - Typically split data into training, validation, and testing
   - Stop training when the error on the *validation* set begins to increase
   - This ensures that the training process never looks at the testing data
2. Include a "regularization" term in the loss function
   - Complete details are beyond the scope of this course.
   - Regularization terms increase the loss the farther the weight vector is from zero: $L_{\text{new}}(w, D) = L(w, D) + \lambda \|w\|$
     - Often using the L1 norm, $\|w\| = \sum_j |w_j|$ or the L2 norm $\|w\| = \sqrt{\sum_j w_j^2}$.
3. Other strategies (e.g., dropout)
4. Use a large network!

$\|\cdot\|$ denotes a **norm** (a notion of "length")

“Use a large network”: Double Descent

- Large networks seem like they should be particularly prone to overfitting.
- When trained sufficiently on large amounts of data, empirical evidence suggests that deep (large) networks tend not to over-fit!

This phenomenon, called *double descent*, is an active research topic!



End