COMPSCI 389
Introduction to Machine Learning

Automatic Differentiation for ML

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

Optimization Perspective (Review)

- Recall: Our goal is to find model parameters w that minimize the loss function L:

$$\text{argmin}_w \, L(w, D)$$

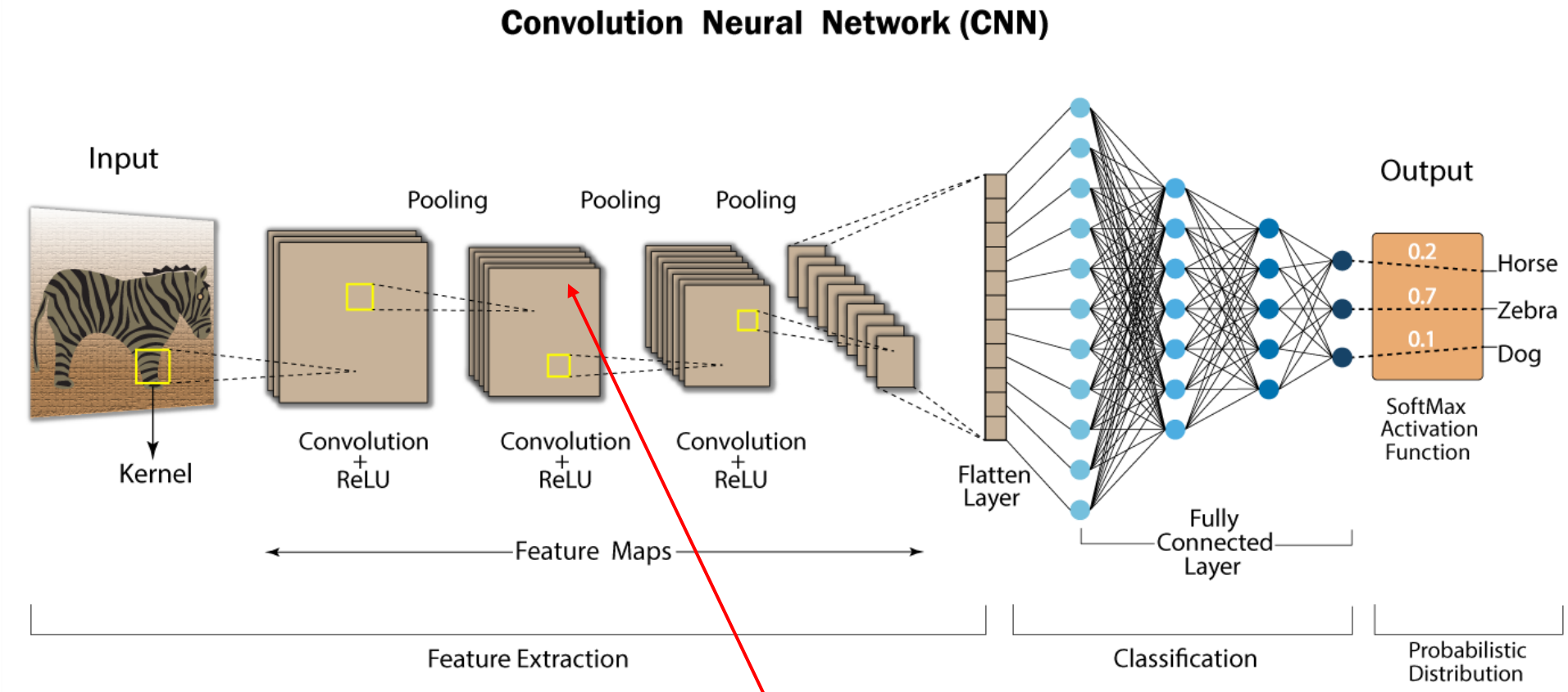
- We can do this using gradient descent.
  - Vector notation:

$$w \leftarrow w - \alpha \frac{\partial L(w, D)}{\partial w}$$

  - Scalar notation:

$$\forall j, \qquad w_j - \alpha \frac{\partial L(w, D)}{\partial w_j}$$

Review



To train the model, we need the derivative of the loss function with respect to each weight.
How can we compute the derivative with respect to this weight in the model?

Answer: Automatic Differentiation!

Automatic Differentiation for ML (Overview)

- First, select an implement a parametric model, f_w.
- Second, select and implement a loss function, L.
- Third, implement gradient descent using automatic differentiation to compute $\frac{\partial L(w,D)}{\partial w}$.

The remainder of this presentation covers:

`20 Automatic Differentiation for ML.ipynb`

Gradient descent on the sample MSE for a linear parametric model (no basis) on GPA data.

- Define the model:

```
def linear_model(X, weights):
    return np.dot(X, weights)
```

- Define the loss function:

```
def loss_function(weights, X, y):
    predictions = linear_model(X, weights)
    return np.mean((predictions - y)**2)
```

- Create a new function that is the gradient of the loss function:

```
grad_loss = grad(loss_function) # Defaults to grad(loss_function, 0)
```

```python
def linear_model(X, weights):
    return np.dot(X, weights)
```

```python
def loss_function(weights, X, y):
    predictions = linear_model(X, weights)
    return np.mean((predictions - y)**2)
```

```python
grad_loss = grad(loss_function) # Defaults to grad(loss_function, 0)
```

- Initialize the weight vector

```python
num_weights = X_train.shape[1]
#weights = np.zeros((num_weights, 1))
weights = np.random.randn(num_weights, 1)
```

Any initialization is valid. Often weights are initialized to zero or random values (here samples from a standard normal distribution)

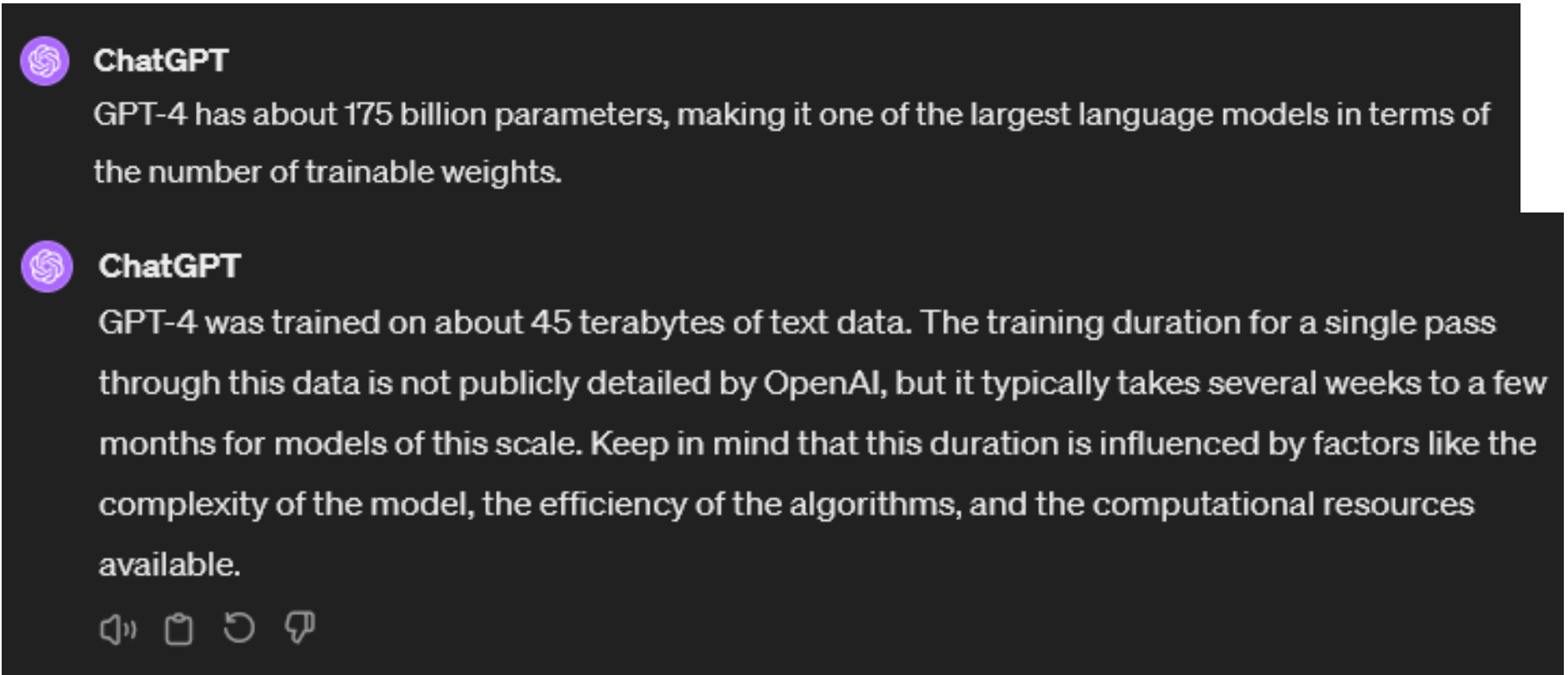
```python
def linear_model(X, weights):
    return np.dot(X, weights)
def loss_function(weights, X, y):
    predictions = linear_model(X, weights)
    return np.mean((predictions - y)**2)
grad_loss = grad(loss_function)
num_iterations = 50
learning_rate = 0.05

# Training loop
for iteration in range(num_iterations):
    weights -= learning_rate * grad_loss(weights, X_train, y_train)

    # Print loss every 10 iterations
    if iteration % 10 == 0:
        current_loss = loss_function(weights, X_train, y_train)
        print(f"Iteration {iteration}, Loss: {current_loss}")

# Evaluate on test data
test_loss = loss_function(weights, X_test, y_test)
print(f"Test MSE: {test_loss}")
```

```
Iteration 0, Loss: 15.686453929819447
Iteration 10, Loss: 3.593373473015224
Iteration 20, Loss: 1.611696934073126
Iteration 30, Loss: 1.044454252510009
Iteration 40, Loss: 0.8214814808118163
Test MSE: 0.7355874502987186
```

This is the power of automatic differentiation: We only had to implement our loss function and parametric model. We did **not** implement or manually compute their derivatives!

Mini-Batch Gradient Descent

- When the data set is large, computing the sample MSE (or gradient of the sample MSE) for the entire training set can take a *very* long time.



Mini-Batch Gradient Descent

- **Idea**: Split the training data into **mini-batches**.
- Each mini-batch is a collection of several rows (training points).
- Each iteration of gradient descent can use a different mini-batch of the training data.
- The process of running gradient descent on all mini-batches one time is called an **epoch**.
  - Each epoch corresponds to one pass over the entire data set, performing one gradient update for each mini-batch.
- Training typically involves running several epochs.
- Different splits of the data into mini-batches are typically used for each epoch.
  - This can be achieved by shuffling the data between epochs.
- We typically define the size of each mini-batch, not the number of mini-batches.

- 43,303 points total
- 34,642 points for training
- Mini-batch size = 100
  - This is a hyperparameter
  - $\left\lceil \frac{34{,}862}{100} \right\rceil = 349$ mini-batches

```
num_epochs = 50
learning_rate = 0.05
minibatch_size = 100
```

- 43,303 points total
- 34,642 points for training
- Mini-batch size = 100
  - This is a hyperparameter
  - $\left\lceil \frac{34{,}862}{100} \right\rceil = 349$ mini-batches
- $i \in \{0, 100, 200, \ldots, 34700, 34800\}$
  - The starting index of each mini-batch

```
num_epochs = 50
learning_rate = 0.05
minibatch_size = 100

for epoch in range(num_epochs):
    # Shuffle the training data
    X_train_shuffled, y_train_shuffled = shuffle(X_train, y_train)

    # Loop over mini-batches
    for i in range(0, X_train.shape[0], minibatch_size):
```

Shuffle the training data to get different mini-batches each epoch.

- 43,303 points total
- 34,642 points for training
- Mini-batch size = 100
  - This is a hyperparameter
  - $\left\lceil \frac{34{,}862}{100} \right\rceil = 349$ mini-batches
- $i \in \{0, 100, 200, \ldots, 34700, 34800\}$
  - The starting index of each mini-batch
- Most batches run from i to $i + 100$
  - The last batch runs from i to the last point, which may be before $i + 100$.

```
num_epochs = 50
learning_rate = 0.05
minibatch_size = 100

for epoch in range(num_epochs):
    # Shuffle the training data
    X_train_shuffled, y_train_shuffled = shuffle(X_train, y_train)

    # Loop over mini-batches
    for i in range(0, X_train.shape[0], minibatch_size):
        end = min(i + minibatch_size, X_train_shuffled.shape[0]) #
```

- 43,303 points total
- 34,642 points for training
- Mini-batch size = 100
  - This is a hyperparameter
  - $\left\lceil \frac{34{,}862}{100} \right\rceil = 349$ mini-batches
- $i \in \{0, 100, 200, \ldots, 34700, 34800\}$
  - The starting index of each mini-batch
- Most batches run from i to $i + 100$
  - The last batch runs from i to the last point, which may be before $i + 100$.
- The inputs and outputs for the current batch.

```
num_epochs = 50
learning_rate = 0.05
minibatch_size = 100

for epoch in range(num_epochs):
    # Shuffle the training data
    X_train_shuffled, y_train_shuffled = shuffle(X_train, y_train)

    # Loop over mini-batches
    for i in range(0, X_train.shape[0], minibatch_size):
        end = min(i + minibatch_size, X_train_shuffled.shape[0]) #
        X_batch = X_train_shuffled[i:end]
        y_batch = y_train_shuffled[i:end]
```

- 43,303 points total
- 34,642 points for training
- Mini-batch size = 100
  - This is a hyperparameter
  - $\left\lceil \frac{34{,}862}{100} \right\rceil = 349$ mini-batches
- $i \in \{0, 100, 200, \ldots, 34700, 34800\}$
  - The starting index of each mini-batch
- Most batches run from i to $i + 100$
  - The last batch runs from i to the last point, which may be before $i + 100$.
- The inputs and outputs for the current batch.
- Perform a gradient update using only the data from the current batch.

```
num_epochs = 50
learning_rate = 0.05
minibatch_size = 100

for epoch in range(num_epochs):
    # Shuffle the training data
    X_train_shuffled, y_train_shuffled = shuffle(X_train, y_train)

    # Loop over mini-batches
    for i in range(0, X_train.shape[0], minibatch_size):
        end = min(i + minibatch_size, X_train_shuffled.shape[0]) #
        X_batch = X_train_shuffled[i:end]
        y_batch = y_train_shuffled[i:end]

        gradients = grad_loss(weights, X_batch, y_batch)
        weights -= learning_rate * gradients
```

- 43,303 points total
- 34,642 points for training
- Mini-batch size = 100
  - This is a hyperparameter
  - $\left\lceil \frac{34{,}862}{100} \right\rceil = 349$ mini-batches
- $i \in \{0, 100, 200, \ldots, 34700, 34800\}$
  - The starting index of each mini-batch
- Most batches run from i to $i + 100$
  - The last batch runs from i to the last point, which may be before $i + 100$.
- The inputs and outputs for the current batch.
- Perform a gradient update using only the data from the current batch.
- Print the loss every 10 epochs

```
num_epochs = 50
learning_rate = 0.05
minibatch_size = 100

for epoch in range(num_epochs):
    # Shuffle the training data
    X_train_shuffled, y_train_shuffled = shuffle(X_train, y_train)

    # Loop over mini-batches
    for i in range(0, X_train.shape[0], minibatch_size):
        end = min(i + minibatch_size, X_train_shuffled.shape[0]) #
        X_batch = X_train_shuffled[i:end]
        y_batch = y_train_shuffled[i:end]

        gradients = grad_loss(weights, X_batch, y_batch)
        weights -= learning_rate * gradients

    # Print loss every 10 epochs
    if epoch % 10 == 0:
        current_loss = loss_function(weights, X_train, y_train)
        print(f"Epoch {epoch}, Loss: {current_loss}")
```

- 43,303 points total
- 34,642 points for training
- Mini-batch size = 100
  - This is a hyperparameter
  - $\left\lceil \frac{34{,}862}{100} \right\rceil = 349$ mini-batches
- $i \in \{0, 100, 200, \ldots, 34700, 34800\}$
  - The starting index of each mini-batch
- Most batches run from i to $i + 100$
  - The last batch runs from i to the last point, which may be before $i + 100$.
- The inputs and outputs for the current batch.
- Perform a gradient update using only the data from the current batch.
- Print the loss every 10 epochs
- Compute the loss on the test set

```
num_epochs = 50
learning_rate = 0.05
minibatch_size = 100

for epoch in range(num_epochs):
    # Shuffle the training data
    X_train_shuffled, y_train_shuffled = shuffle(X_train, y_train)

    # Loop over mini-batches
    for i in range(0, X_train.shape[0], minibatch_size):
        end = min(i + minibatch_size, X_train_shuffled.shape[0]) #
        X_batch = X_train_shuffled[i:end]
        y_batch = y_train_shuffled[i:end]

        gradients = grad_loss(weights, X_batch, y_batch)
        weights -= learning_rate * gradients

    # Print loss every 10 epochs
    if epoch % 10 == 0:
        current_loss = loss_function(weights, X_train, y_train)
        print(f"Epoch {epoch}, Loss: {current_loss}")

# Evaluate on test data
test_loss = loss_function(weights, X_test, y_test)
print(f"Test MSE: {test_loss}")
```

Mini-Batch Gradient Descent (Results)

Without Mini-Batches

```
Iteration 0, Loss: 15.686453929819447
Iteration 10, Loss: 3.593373473015224
Iteration 20, Loss: 1.611696934073126
Iteration 30, Loss: 1.044454252510009
Iteration 40, Loss: 0.8214814808118163
Test MSE: 0.7355874502987186
```

Using Mini-Batches

```
Epoch 0, Loss: 0.5832311803176145
Epoch 10, Loss: 0.584201610532094
Epoch 20, Loss: 0.5832333795591383
Epoch 30, Loss: 0.5836298546523091
Epoch 40, Loss: 0.5840496234285119
Test MSE: 0.589257892731320 6
```

- Notice that the sample MSE reached lower values in fewer epochs when using mini-batches!
- Mini-batch gradient descent tends to:
  - A) Speed up gradient computations because it uses less data per update, allowing for more frequent updates.
  - B) Speed up the optimization process, achieving lower losses in fewer epochs
- The complete explanation for why mini-batches can lead to quicker and more efficient learning is complex and may be covered in COMPSCI 682, Neural Networks.

End