Neural Networks Foundations

Dr. Mahmoud N Mahmoud
mnmahmoud@ncat.edu

North Carolina A & T State University

November 15, 2022

Outline

Linear Regression

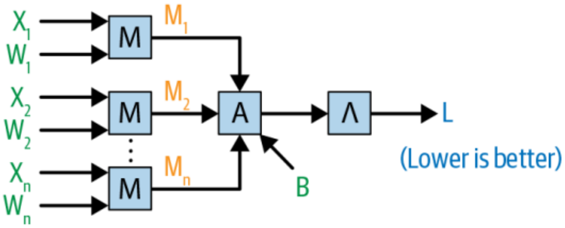
Linear Regression

Linear regression attempts to model the relationship between two variables by fitting a linear equation to observed data

$$y_i = \beta_0 + \beta_1 \times x_1 + ... + \beta_n \times x_k + \epsilon$$

- β_0 term to adjust the “baseline” value of the prediction.
- ϵ because in the error in the prediction.

Linear Regression: A Diagram



- Λ is a comparsion operator between the true output and the predicted output.
- L is the called loss.

Training Linear Regression

- let's handle the simpler scenario in which we don't have an intercept term
- We have observation vector $xi = [x_{i1}, x_{i2}, x_{i3} \dots x_{ik}]$
- Another vector of parameters that we'll call $W = [w_1, w_2, w_{w3} \dots w_k]^T$
- Our prediction would then simply be

$$p_i = x_i \times W = w_1 \times x_{i1} + w_2 \times x_{i2} + ... + w_k \times x_{ik}$$

Batch Prediction

$$p_{batch} = X_{batch} \times W = \begin{bmatrix} x_{11} & x_{12} & x_{13} & \dots & x_{1k} \\ x_{21} & x_{22} & x_{23} & \dots & x_{2k} \\ x_{31} & x_{32} & x_{33} & \dots & x_{3k} \end{bmatrix} \times \begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ \vdots \end{bmatrix} = \begin{bmatrix} x_{11} \times w_1 + x_{12} \times w_2 + x_{13}\ w_3 + \cdots \\ x_{21} \times w_1 + x_{22} \times w_2 + x_{23}\ w_3 + \cdots \\ x_{31} \times w_1 + x_{32} \times w_2 + x_{33}\ w_3 + \cdots \end{bmatrix}$$

- Generating predictions for a batch of observations in a linear regression can be done with a matrix multiplication

"Training" this model

1. At a high level, models take in batch of data, combine them with parameters in some way, and produce predictions.

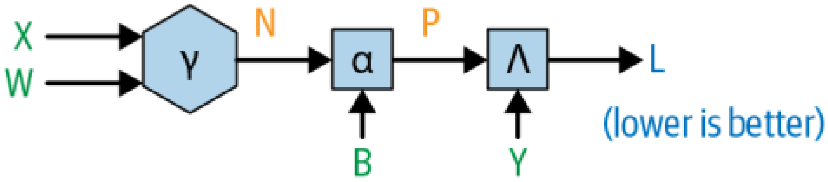
$$p_{batch} = \begin{bmatrix} p_1 \\ p_2 \\ p_3 \end{bmatrix}$$

2. Compute model penalty

$$MSE\left(p_{batch}, y_{batch}\right) = MSE\left(\begin{bmatrix} p_1 \\ p_2 \\ p_3 \end{bmatrix}, \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix}\right) = \frac{(y_1 - p_1)^2 + (y_2 - p_2)^2 + (y_3 - p_3)^2}{3}$$

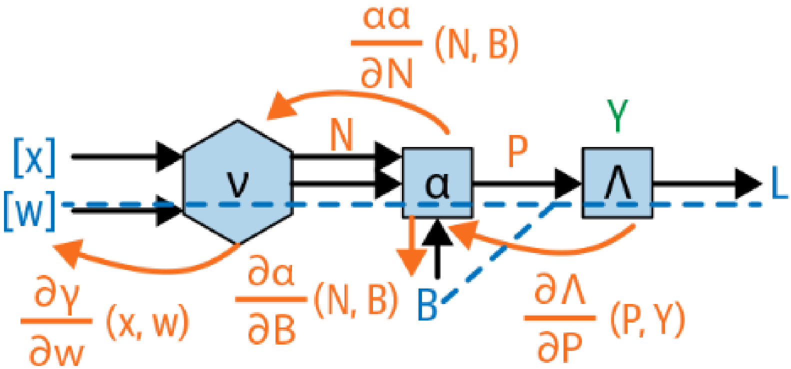
3. Compute the gradient of the error with respect to each element of W
4. Update W to reduce the error

Diagram and Math



- $N = X \times W$
- $P = N + B$
- $L = (Y - P)^2$

Calculating the Gradients: A Diagram



- We will get the gradient of L with respect to the weight and the bias.

Calculating the Gradients: Math

Gradient with respect to the weight

$$\frac{\partial \Lambda}{\partial P}(P, Y) \times \frac{\partial \alpha}{\partial N}(N, B) \times \frac{\partial \nu}{\partial W}(X, W)$$

Calculating the Gradients: Math

Gradient with respect to the weight

$$\frac{\partial \Lambda}{\partial P}(P, Y) \times \frac{\partial \alpha}{\partial N}(N, B) \times \frac{\partial \nu}{\partial W}(X, W)$$

- $\frac{\partial \Lambda}{\partial P}(P, Y) = -1 \times (2 \times (Y - P))$

Calculating the Gradients: Math

Gradient with respect to the weight

$$\frac{\partial \Lambda}{\partial P}(P,Y) \times \frac{\partial \alpha}{\partial N}(N,B) \times \frac{\partial \nu}{\partial W}(X,W)$$

- $\frac{\partial \Lambda}{\partial P}(P,Y) = -1 \times (2 \times (Y - P))$
- $\frac{\partial \alpha}{\partial N}$ = vector of ones

Calculating the Gradients: Math

Gradient with respect to the weight

$$\frac{\partial \Lambda}{\partial P}(P,Y) \times \frac{\partial \alpha}{\partial N}(N,B) \times \frac{\partial \nu}{\partial W}(X,W)$$

- $\frac{\partial \Lambda}{\partial P}(P,Y) = -1 \times (2 \times (Y - P))$
- $\frac{\partial \alpha}{\partial N}$ = vector of ones
- $\frac{\partial \gamma}{\partial W}$ = X^T

Calculating the Gradients: Math

Gradient with respect to the weight

$$\frac{\partial \Lambda}{\partial P}(P, Y) \times \frac{\partial \alpha}{\partial N}(N, B) \times \frac{\partial \nu}{\partial W}(X, W)$$

- $\frac{\partial \Lambda}{\partial P}(P, Y) = -1 \times (2 \times (Y - P))$
- $\frac{\partial \alpha}{\partial N}$ = vector of ones
- $\frac{\partial \gamma}{\partial W}$ = X^T

Calculating the Gradients: Math

Gradient with respect to the weight

$$\frac{\partial \Lambda}{\partial P}(P, Y) \times \frac{\partial \alpha}{\partial N}(N, B) \times \frac{\partial \nu}{\partial W}(X, W)$$

- $\frac{\partial \Lambda}{\partial P}(P, Y) = -1 \times (2 \times (Y - P))$
- $\frac{\partial \alpha}{\partial N}$ = vector of ones
- $\frac{\partial \gamma}{\partial W} = X^T$

Calculate $\frac{dL}{dB}$?

Outline

Linear Regression: Forward Pass

```
def forward_linear_regression(X_batch: ndarray,
                              y_batch: ndarray,
                              weights: Dict[str, ndarray])
                              -> Tuple[float, Dict[str, ndarray]]:
    '''
    Forward pass for the step-by-step linear regression.
    '''
    # assert batch sizes of X and y are equal
    assert X_batch.shape[0] == y_batch.shape[0]

    # assert that matrix multiplication can work
    assert X_batch.shape[1] == weights['W'].shape[0]

    # assert that B is simply a 1x1 ndarray
    assert weights['B'].shape[0] == weights['B'].shape[1] == 1

    # compute the operations on the forward pass
    N = np.dot(X_batch, weights['W'])

    P = N + weights['B']

    loss = np.mean(np.power(y_batch - P, 2))

    # save the information computed on the forward pass
    forward_info: Dict[str, ndarray] = {}
    forward_info['X'] = X_batch
    forward_info['N'] = N
    forward_info['P'] = P
    forward_info['y'] = y_batch

    return loss, forward_info
```

Linear Regression: Backward Pass

```python
def loss_gradients(forward_info: Dict[str, ndarray],
                   weights: Dict[str, ndarray]) -> Dict[str, ndarray]:
    '''
    Compute dLdW and dLdB for the step-by-step linear regression model.
    '''
    batch_size = forward_info['X'].shape[0]

    dLdP = -2 * (forward_info['y'] - forward_info['P'])

    dPdN = np.ones_like(forward_info['N'])

    dPdB = np.ones_like(weights['B'])

    dLdN = dLdP * dPdN

    dNdW = np.transpose(forward_info['X'], (1, 0))

    # need to use matrix multiplication here,
    # with dNdW on the left (see note at the end of last chapter)
    dLdW = np.dot(dNdW, dLdN)

    # need to sum along dimension representing the batch size
    # (see note near the end of this chapter)
    dLdB = (dLdP * dPdB).sum(axis=0)

    loss_gradients: Dict[str, ndarray] = {}
    loss_gradients['W'] = dLdW
    loss_gradients['B'] = dLdB

    return loss_gradients
```

Using These Gradients to Train the Model

Now we'll simply run the following procedure over and over again:

1. Select a batch of data.

Using These Gradients to Train the Model

Now we'll simply run the following procedure over and over again:

1. Select a batch of data.
2. Run the forward pass of the model.

Using These Gradients to Train the Model

Now we'll simply run the following procedure over and over again:

1. Select a batch of data.
2. Run the forward pass of the model.
3. Run the backward pass of the model using the info computed on the forward pass.

Using These Gradients to Train the Model

Now we'll simply run the following procedure over and over again:

1. Select a batch of data.
2. Run the forward pass of the model.
3. Run the backward pass of the model using the info computed on the forward pass.
4. Use the gradients computed on the backward pass to update the weights.

$$w_i = w_i - \text{learning rate} * \frac{dL}{dw_i}$$

Outline

Introduction

- As we saw that linear regression is only able to learn linear input/output relationship.
- How can we extend this chain of reasoning to design a more complex model that can learn nonlinear relationships?

Introduction

- As we saw that linear regression is only able to learn linear input/output relationship.
- How can we extend this chain of reasoning to design a more complex model that can learn nonlinear relationships?
- The central idea is that we'll first do many linear regressions, then feed the results through a nonlinear function, and finally do one last linear regression that ultimately makes the predictions.

Step 1: A Bunch of Linear Regressions

- if our data X had dimensions `[batch_size, num_features]`,

Step 1: A Bunch of Linear Regressions

- if our data X had dimensions `[batch_size, num_features]`,
- this output is, for each observation in the batch, simply a `weighted sum` of the original features.

Step 1: A Bunch of Linear Regressions

- if our data X had dimensions `[batch_size, num_features]`,
- this output is, for each observation in the batch, simply a `weighted sum` of the original features.
- To do multiple linear regressions at once, multiply our input by a weight matrix with dimensions `[num_features, num_outputs]` , resulting in an output of dimensions `[batch_size, num_outputs]`

Step 1: A Bunch of Linear Regressions

- if our data X had dimensions `[batch_size, num_features]`,
- this output is, for each observation in the batch, simply a `weighted sum` of the original features.
- To do multiple linear regressions at once, multiply our input by a weight matrix with dimensions `[num_features, num_outputs]` , resulting in an output of dimensions `[batch_size, num_outputs]`
- Now, for each observation, we have `num_outputs` different weighted sums of the original features.

A Nonlinear Function

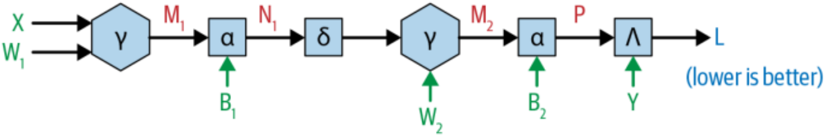
- We will feed each of these weighted sums through a nonlinear sigmoid function
- Why sigmoid? Why not square?
  - Preservation of information.
  - The function is nonlinear.
  - Has the nice property that its derivative can be expressed in terms of the function itself:

$$\frac{\partial \sigma}{\partial u}(x) = \sigma(x) \times (1 - \sigma(x))$$

Step 3: Another Linear Regression

- The output from each linear regression is weighted and fed again to another linear regression.
- The cascading of linear regressions enable learning complex input/output relations.

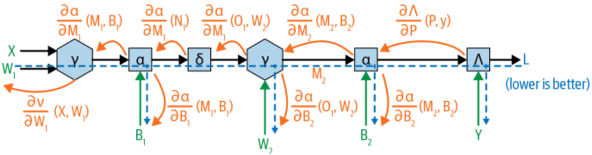
Simplified Diagram



Computational graph for a simple neural network.

Another Diagram (Most Popular)

Neural Networks: The Backward Pass



Derivative	Code
$\frac{\partial \Lambda}{\partial P}(P, y)$	`dLdP = -(forward_info[y] - forward_info[P])`
$\frac{\partial \alpha}{\partial M_2}(M_2, B_2)$	`np.ones_like(forward_info[M2])`
$\frac{\partial \alpha}{\partial B_2}(M_2, B_2)$	`np.ones_like(weights[B2])`
$\frac{\partial \nu}{\partial W_2}(O_1, W_2)$	`dM2dW2 = np.transpose(forward_info[O1], (1, 0))`
$\frac{\partial \nu}{\partial O_1}(O_1, W_2)$	`dM2dO1 = np.transpose(weights[W2], (1, 0))`
$\frac{\partial \sigma}{\partial u}(N_1)$	`dO1dN1 = sigmoid(forward_info[N1] × (1 - sigmoid(forward_info[N1])`
$\frac{\partial \alpha}{\partial M_1}(M_1, B_1)$	`dN1dM1 = np.ones_like(forward_info[M1])`
$\frac{\partial \alpha}{\partial B_1}(M_1, B_1)$	`dN1dB1 = np.ones_like(weights[B1])`
$\frac{\partial \nu}{\partial W_1}(X, W_1)$	`dM1dW1 = np.transpose(forward_info[X], (1, 0))`

Forward Pass: Code

```
def forward_loss(X: ndarray,
                 y: ndarray,
                 weights: Dict[str, ndarray]
                 ) -> Tuple[Dict[str, ndarray], float]:
    '''
    Compute the forward pass and the loss for the step-by-step
    neural network model.
    '''
    M1 = np.dot(X, weights['W1'])

    N1 = M1 + weights['B1']

    O1 = sigmoid(N1)

    M2 = np.dot(O1, weights['W2'])

    P = M2 + weights['B2']

    loss = np.mean(np.power(y - P, 2))

    forward_info: Dict[str, ndarray] = {}
    forward_info['X'] = X
    forward_info['M1'] = M1
    forward_info['N1'] = N1
    forward_info['O1'] = O1
    forward_info['M2'] = M2
    forward_info['P'] = P
    forward_info['y'] = y

    return forward_info, loss
```

Forward Pass: Backward Pass

```python
def loss_gradients(forward_info: Dict[str, ndarray],
                   weights: Dict[str, ndarray]) -> Dict[str, ndarray]:
    '''
    Compute the partial derivatives of the loss with respect to each of the parameters in the neural network.
    '''
    dLdP = -(forward_info['y'] - forward_info['P'])

    dPdM2 = np.ones_like(forward_info['M2'])

    dLdM2 = dLdP * dPdM2

    dPdB2 = np.ones_like(weights['B2'])

    dLdB2 = (dLdP * dPdB2).sum(axis=0)

    dM2dW2 = np.transpose(forward_info['O1'], (1, 0))

    dLdW2 = np.dot(dM2dW2, dLdP)

    dM2dO1 = np.transpose(weights['W2'], (1, 0))

    dLdO1 = np.dot(dLdM2, dM2dO1)

    dO1dN1 = sigmoid(forward_info['N1']) * (1- sigmoid(forward_info['N1']))

    dLdN1 = dLdO1 * dO1dN1

    dN1dB1 = np.ones_like(weights['B1'])

    dN1dM1 = np.ones_like(forward_info['M1'])

    dLdB1 = (dLdN1 * dN1dB1).sum(axis=0)

    dLdM1 = dLdN1 * dN1dM1

    dM1dW1 = np.transpose(forward_info['X'], (1, 0))

    dLdW1 = np.dot(dM1dW1, dLdM1)

    loss_gradients: Dict[str, ndarray] = {}
    loss_gradients['W2'] = dLdW2
    loss_gradients['B2'] = dLdB2.sum(axis=0)
    loss_gradients['W1'] = dLdW1
    loss_gradients['B1'] = dLdB1.sum(axis=0)

    return loss_gradients
```

Questions