Lecture 2: Deep Learning Fundamentals (Part 1)
Announcements

- A0 will be released later today. Due next Wed Jan 15.
  - Setup assignment for later homeworks.
Last time: key ingredients of deep learning success

Algorithms

Compute

Data
Today: Deep learning fundamentals (part 1)

- Machine learning vs. deep learning framework

- Deep learning basics through a simple example
  - Defining a neural network architecture
  - Defining a loss function
  - Optimizing the loss function

- Model implementation using deep learning frameworks
Machine learning framework

Data-driven learning of a mapping from input to output
Machine learning framework

Data-driven learning of a mapping from input to output

Traditional machine learning approaches

Input → Feature extractor → Machine learning model → Output

(e.g., color and texture histograms) → (e.g., support vector machines and random forests) → (e.g., presence or not or disease)
Deep learning (a type of machine learning)
Deep learning (a type of machine learning)

Traditional machine learning

Input → Feature extractor → Machine learning model → Output

(e.g., color and texture histograms) (e.g., support vector machines and random forests) (e.g., presence or not or disease)
Deep learning (a type of machine learning)

Traditional machine learning

Input  \rightarrow  \text{Feature extractor}  \rightarrow  \text{Machine learning model}  \rightarrow  \text{Output}

(e.g., )  \rightarrow  (e.g., color and texture histograms) \rightarrow  (e.g., support vector machines and random forests) \rightarrow  (e.g., presence or not or disease)

Deep learning

Input  \rightarrow  \text{Deep Learning Model}  \rightarrow  \text{Output}

(e.g., )  \rightarrow  (e.g., convolutional and recurrent neural networks) \rightarrow  (e.g., presence or not or disease)
Deep learning (a type of machine learning)

Traditional machine learning

Input → Feature extractor → Machine learning model → Output

(e.g., input) → (e.g., color and texture histograms) → (e.g., support vector machines and random forests) → (e.g., presence or not or disease)

Deep learning

Input → Deep Learning Model → Output

(e.g., input) → (e.g., convolutional and recurrent neural networks) → (e.g., presence or not or disease)

Directly learns what are useful (and better!) features from the training data
How do deep learning models perform feature extraction?

Input (e.g., )

Output (e.g., presence or not or disease)
How do deep learning models perform feature extraction?

Hierarchical structure of neural networks allows compositional extraction of increasingly complex features.

Input (e.g.,)

Output (e.g., presence or not or disease)

Low-level features

Mid-level features

High-level features

Feature visualizations from Zeiler and Fergus 2013
How do deep learning models perform feature extraction?

Input (e.g.,)

Hierarchical structure of neural networks allows compositional extraction of increasingly complex features

Output (e.g., presence or not or disease)

Low-level features

Mid-level features

High-level features

Will see soon how neural networks learn what are useful features to extract ("feature learning")

Feature visualizations from Zeiler and Fergus 2013
Topics we will cover

- Preparing data for deep learning
- Neural network models
- Training neural networks
- Evaluating models

Input (e.g., chest X-ray)

Output (e.g., presence or not of disease)
First (today): deep learning basics through a simple example

Let us consider the task of **regression**: predicting a single real-valued output from input data

**Model input**: data vector $x = [x_1, x_2, \ldots, x_N]$  
**Model output**: prediction (single number) $\hat{y}$
First (today): deep learning basics through a simple example

Let us consider the task of **regression**: predicting a single real-valued output from input data

**Model input**: data vector $x = [x_1, x_2, \ldots, x_N]$  
**Model output**: prediction (single number) $\hat{y}$

Example: predicting hospital length-of-stay from clinical variables in the electronic health record

$x = [\text{age, weight, \ldots, temperature, oxygen saturation}]$  
$\hat{y} = \text{length-of-stay (days)}$

Example: predicting expression level of a target gene from the expression levels of $N$ landmark genes

$x \in \mathcal{R}^N = \text{expression levels of } N \text{ landmark genes}$  
$\hat{y} = \text{expression level of target gene}$
Defining a neural network architecture

Our first architecture: a single-layer, fully connected neural network
Defining a neural network architecture

Our first architecture: a single-layer, fully connected neural network

all inputs of a layer are connected to all outputs of a layer
Defining a neural network architecture

Our first architecture: a **single-layer, fully connected** neural network

For simplicity, use a 3-dimensional input \((N = 3)\)

Output:

\[
\hat{y} = w_1 x_1 + w_2 x_2 + w_3 x_3 + b
\]

\[
= w^T x + b
\]
Defining a neural network architecture

Our first architecture: a **single-layer, fully connected** neural network.

For simplicity, use a 3-dimensional input (N = 3).

Output: \[ \hat{y} = w_1 x_1 + w_2 x_2 + w_3 x_3 + b \]
\[ = w^T x + b \]

bias term (allows constant shift)
Defining a neural network architecture

Our first architecture: a **single-layer, fully connected** neural network

For simplicity, use a 3-dimensional input (N = 3)

Output:

\[
\hat{y} = w_1 x_1 + w_2 x_2 + w_3 x_3 + b
\]

\[
= w^T x + b
\]
Defining a neural network architecture

Our first architecture: a **single-layer, fully connected** neural network

For simplicity, use a 3-dimensional input ($N = 3$)

Output:

$$\hat{y} = w_1 x_1 + w_2 x_2 + w_3 x_3 + b$$

$$= w^T x + b$$

Neural network parameters:

$$W = \{[w_1, w_2, w_3], b\}$$
Defining a neural network architecture

Our first architecture: a **single-layer, fully connected** neural network

For simplicity, use a 3-dimensional input (N = 3)

- **Output:**
  \[ \hat{y} = w_1 x_1 + w_2 x_2 + w_3 x_3 + b \]
  \[ = w^T x + b \]

**Neural network parameters:**

- Layer inputs
- Layer output(s)
- Bias term (allows constant shift)
- Layer “weights”
- Layer bias

all inputs of a layer are connected to all outputs of a layer.
Defining a neural network architecture

Our first architecture: a **single-layer, fully connected** neural network.

For simplicity, use a 3-dimensional input ($N = 3$).

![Diagram of a single-layer, fully connected neural network]

**Output:**

$$\hat{y} = w_1 x_1 + w_2 x_2 + w_3 x_3 + b$$

$$= w^T x + b$$

**Neural network parameters:**

$$W = \{[w_1, w_2, w_3], b\}$$

Often refer to all parameters together as just “weights”. Bias is implicitly assumed.
Defining a neural network architecture

Our first architecture: a **single-layer, fully connected** neural network.

For simplicity, use a 3-dimensional input ($N = 3$).

Neural network parameters:

Our first architecture:

- **Single-layer**, **fully connected** neural network.
- All inputs of a layer are connected to all outputs of a layer.
- For simplicity, use a 3-dimensional input ($N = 3$).
- Output:
  
  \[ \hat{y} = w_1 x_1 + w_2 x_2 + w_3 x_3 + b \]
  
  \[ = w^T x + b \]

Neural network parameters:

\[ W = \{ [w_1, w_2, w_3], b \} \]

Caveats of our first (simple) neural network architecture:

- Single layer still “shallow”, not yet a “deep” neural network. Will see soon how to stack multiple layers.
- Also equivalent to a linear regression model! But useful base case for deep learning.

Bias term (allows constant shift)

Layer “weights”

Layer bias

Often refer to all parameters together as just “weights”. Bias is implicitly assumed.
Defining a loss function

Output: \[ \hat{y} = w_1 x_1 + w_2 x_2 + w_3 x_3 + b \]
\[ = w^T x + b \]

Neural network parameters:
\[ W = \{[w_1, w_2, w_3], b\} \]
Defining a loss function

Output: $\hat{y} = w_1 x_1 + w_2 x_2 + w_3 x_3 + b$

$= w^T x + b$

Neural network parameters:

$W = \{ [w_1, w_2, w_3], b \}$

Loss functions are quantitative measures of how satisfactory the model predictions are (i.e., how “good” the model parameters are).
Defining a loss function

Output: \( \hat{y} = w_1 x_1 + w_2 x_2 + w_3 x_3 + b \)
\[= w^T x + b \]

Neural network parameters:

\[ W = \{ [w_1, w_2, w_3], b \} \]

Loss functions are quantitative measures of how satisfactory the model predictions are (i.e., how “good” the model parameters are).

We will use the mean square error (MSE) loss which is standard for regression.
Defining a loss function

Output: \( \hat{y} = w_1 x_1 + w_2 x_2 + w_3 x_3 + b \)
\[ = w^T x + b \]

Neural network parameters:
\[ W = \{ [w_1, w_2, w_3], b \} \]

Loss functions are quantitative measures of how satisfactory the model predictions are (i.e., how “good” the model parameters are).

We will use the mean square error (MSE) loss which is standard for regression.

MSE loss for a single example \( x^i \), when the prediction is \( \hat{y}^i \) and the correct (ground truth) output is \( y^i \):

\[ L^i(W) = (\hat{y}^i - y^i)^2 \]
Defining a loss function

Output: \( \hat{y} = w_1 x_1 + w_2 x_2 + w_3 x_3 + b \)
\[ = w^T x + b \]

Neural network parameters:
\[ W = \{ [w_1, w_2, w_3], b \} \]

Loss functions are quantitative measures of how satisfactory the model predictions are (i.e., how “good” the model parameters are).

We will use the mean square error (MSE) loss which is standard for regression.

MSE loss for a single example \( x^i \), when the prediction is \( \hat{y}^i \) and the correct (ground truth) output is \( y^i \):
\[
L^i(W) = (\hat{y}^i - y^i)^2 \]
the loss is small when the prediction is close to the ground truth
Defining a loss function

Output: \( \hat{y} = w_1 x_1 + w_2 x_2 + w_3 x_3 + b \)
\[ = w^T x + b \]

Neural network parameters:
\[ W = \{ [w_1, w_2, w_3], b \} \]

Loss functions are quantitative measures of how satisfactory the model predictions are (i.e., how “good” the model parameters are).

We will use the mean square error (MSE) loss which is standard for regression.

MSE loss for a single example \( x^i \), when the prediction is \( \hat{y}^i \) and the correct (ground truth) output is \( y^i \):
\[ L^i(W) = (\hat{y}^i - y^i)^2 \]

\( i = \{1, \ldots, M\} \): 
\[ L = \frac{1}{M} \sum_i L^i(W) \]

the loss is small when the prediction is close to the ground truth
Optimizing the loss function: gradient descent

Goal: find the “best” values of the model parameters that minimize the loss function
Optimizing the loss function: gradient descent

Goal: find the “best” values of the model parameters that minimize the loss function

The approach we will take: gradient descent
Optimizing the loss function: gradient descent

Goal: find the “best” values of the model parameters that minimize the loss function

The approach we will take: gradient descent

“Loss landscape”: the value of the loss function at every value of the model parameters

Figure credit: https://easyai.tech/wp-content/uploads/2019/01/tiduxiajiang-1.png
Optimizing the loss function: gradient descent

Goal: find the “best” values of the model parameters that minimize the loss function

The approach we will take: gradient descent

“Loss landscape”: the value of the loss function at every value of the model parameters

Main idea: iteratively update the model parameters, to take steps in the local direction of steepest (negative) slope, i.e., the negative gradient

Figure credit: https://easyai.tech/wp-content/uploads/2019/01/tiduxiajiang-1.png
Goal: find the “best” values of the model parameters that minimize the loss function

The approach we will take: gradient descent

“Loss landscape”: the value of the loss function at every value of the model parameters

Main idea: iteratively update the model parameters, to take steps in the local direction of steepest (negative) slope, i.e., the negative gradient

We will be able to use gradient descent to iteratively optimize the complex loss function landscapes corresponding to deep neural networks!

Figure credit: https://easyai.tech/wp-content/uploads/2019/01/tiduxiajiang-1.png
The derivative of a function is a measure of local slope.

Ex: \( f(x) = x^2 \quad \frac{\partial f}{\partial x} = 2x \)
Review from calculus: derivatives and gradients

The **derivative** of a function is a measure of local slope.

Ex: \( f(x) = x^2 \) \[ \frac{\partial f}{\partial x} = 2x \]
Review from calculus: derivatives and gradients

The **derivative** of a function is a measure of local slope.

Ex: \( f(x) = x^2 \) \[ \frac{\partial f}{\partial x} = 2x \]
Review from calculus: derivatives and gradients

The **derivative** of a function is a measure of local slope.

Ex: \( f(x) = x^2 \quad \frac{\partial f}{\partial x} = 2x \)
Review from calculus: derivatives and gradients

The **derivative** of a function is a measure of local slope.

Ex: \( f(x) = x^2 \quad \frac{\partial f}{\partial x} = 2x \)

The **gradient** of a function of multiple variables is the vector of partial derivatives of the function with respect to each variable.

Ex: \( f(x_1, x_2) = 3x_1^2 + x_2^2 \quad \nabla f_x = [6x_1, 2x_2] \)
Review from calculus: derivatives and gradients

The **derivative** of a function is a measure of local slope.

Ex: \[ f(x) = x^2 \quad \frac{\partial f}{\partial x} = 2x \]

The **gradient** of a function of multiple variables is the vector of partial derivatives of the function with respect to each variable.

Ex: \[ f(x_1, x_2) = 3x_1^2 + x_2^2 \quad \nabla f_x = [6x_1, 2x_2] \]

The gradient evaluated at a particular point is the direction of steepest ascent of the function.

\[ \nabla f_x \bigg|_{x_1=1, x_2=1} = [6, 2] \]
Review from calculus: derivatives and gradients

The **derivative** of a function is a measure of local slope.

Ex: \( f(x) = x^2 \quad \frac{\partial f}{\partial x} = 2x \)

The **gradient** of a function of multiple variables is the vector of partial derivatives of the function with respect to each variable.

Ex: \( f(x_1, x_2) = 3x_1^2 + x_2^2 \quad \nabla f_x = [6x_1, 2x_2] \)

The gradient evaluated at a particular point is the direction of steepest ascent of the function.

\[
\nabla f_x \bigg|_{x_1=1, x_2=1} = [6, 2]
\]

The negative of the gradient is the direction of steepest descent -> direction we want to move in the loss function landscape!
Gradient descent algorithm

Let the gradient of the loss function with respect to the model parameters $w$ be:

$$\nabla L_W = \left[ \frac{\partial L}{\partial w_1}, \frac{\partial L}{\partial w_2}, \ldots, \frac{\partial L}{\partial w_K} \right]$$
Gradient descent algorithm

Let the gradient of the loss function with respect to the model parameters \( w \) be:

\[
\nabla L_W = \left[ \frac{\partial L}{\partial w_1}, \frac{\partial L}{\partial w_2}, \ldots, \frac{\partial L}{\partial w_K} \right]
\]

For ease of notation, rewrite parameter \( b \) as \( w_0 \) corresponding to \( x_0 = 1 \):

\[
\hat{y} = w_0 x_0 + w_1 x_1 + w_2 x_2 + w_3 x_3
\]

\[
W = \{[w_0, w_1, w_2, w_3]\}
**Gradient descent algorithm**

Let the gradient of the loss function with respect to the model parameters $w$ be:

$$\nabla L_W = \left[ \frac{\partial L}{\partial w_1}, \frac{\partial L}{\partial w_2}, ..., \frac{\partial L}{\partial w_K} \right]$$

Then we can minimize the loss function by iteratively updating the model parameters (“taking steps”) in the direction of the negative gradient, until convergence:

$$W := W - \alpha \nabla L_W$$

For ease of notation, rewrite parameter $b$ as $w_0$ corresponding to $x_0 = 1$:

$$\hat{y} = w_0 x_0 + w_1 x_1 + w_2 x_2 + w_3 x_3$$

$$W = \{[w_0, w_1, w_2, w_3]\}$$
Gradient descent algorithm

Let the gradient of the loss function with respect to the model parameters \( w \) be:

\[
\nabla L_W = \left[ \frac{\partial L}{\partial w_1}, \frac{\partial L}{\partial w_2}, \ldots, \frac{\partial L}{\partial w_K} \right]
\]

Then we can minimize the loss function by iteratively updating the model parameters ("taking steps") in the direction of the negative gradient, until convergence:

\[
W := W - \alpha \nabla L_W
\]

"step size" hyperparameter (design choice) indicating how big of a step in the negative gradient direction we want to take at each update. Too big -> may overshoot minima. Too small -> optimization takes too long.

For ease of notation, rewrite parameter \( b \) as \( w_0 \) corresponding to \( x_0 = 1 \):

\[
\hat{y} = w_0 x_0 + w_1 x_1 + w_2 x_2 + w_3 x_3
\]

\[
W = \{[w_0, w_1, w_2, w_3]\}
Gradient descent algorithm: in code

```python
# initialize vector of weight parameters to random values
weights = random_init(weights_dimension)

while True:
    # evaluate the gradient of the loss function with respect to the weights
    weights_grad = evaluate_gradient(loss_fcn, data, weights)
    # update the weights in the direction of the negative gradient
    weights = weights - step_size * weights_grad
```
Stochastic gradient descent (SGD)

Evaluating gradient involves iterating over all data examples, which can be slow!

In practice, usually use stochastic gradient descent: estimate gradient over a sample of data examples (usually as many as can fit on GPU at one time, e.g. 32, 64, 128)

```python
# initialize vector of weight parameters to random values
weights = random_init(weights_dimension)

while True:
    # sample a batch of data examples
data_batch = sample_data(data, 128)

    # evaluate the gradient of the loss function with respect to the weights
weights_grad = evaluate_gradient(loss_fcn, data_batch, weights)
# update the weights in the direction of the negative gradient
weights = weights - step_size * weights_grad
```
Optimizing the loss function: our example

Loss function:
Per-example: \( L^i(W) = (\hat{y}^i - y^i)^2 \)
Over M examples: \( L = \frac{1}{M} \sum_i L^i(W) \)

Output:  
\[
\hat{y} = w_1 x_1 + w_2 x_2 + w_3 x_3 + b \\
= w^T x + b
\]

Neural network parameters:
\[
W = \{ [w_1, w_2, w_3], b \} 
\]
Optimizing the loss function: our example

Loss function:
Per-example: \( L^i(W) = (\hat{y}^i - y^i)^2 \)
Over M examples: \( L = \frac{1}{M} \sum_{i} L^i(W) \)

Gradient of loss w.r.t. weights:
Partial derivative of loss w.r.t. kth weight:
\[
\frac{\partial L^i}{\partial w_k} = \frac{\partial L^i}{\partial \hat{y}^i} \frac{\partial \hat{y}^i}{\partial w_k} = 2(\hat{y}^i - y^i)x_k^i
\]

Output: \( \hat{y} = w_1 x_1 + w_2 x_2 + w_3 x_3 + b \)
\[ = w^T x + b \]

Neural network parameters:
\( W = \{[w_1, w_2, w_3], b\} \)
Optimizing the loss function: our example

Output: \( \hat{y} = w_1 x_1 + w_2 x_2 + w_3 x_3 + b \)

\[ = w^T x + b \]

Neural network parameters:
\[ W = \{ [w_1, w_2, w_3], b \} \]

Loss function:
Per-example: \( L^i(W) = (\hat{y}^i - y^i)^2 \)

Over M examples: \[ L = \frac{1}{M} \sum_i L^i(W) \]

Gradient of loss w.r.t. weights:
Partial derivative of loss w.r.t. kth weight:
\[ \frac{\partial L^i}{\partial w_k} = \frac{\partial L^i}{\partial \hat{y}^i} \frac{\partial \hat{y}^i}{\partial w_k} = 2(\hat{y}^i - y^i) x_k^i \]
Optimizing the loss function: our example

Loss function:
- Per-example: $L^i(W) = (\hat{y}^i - y^i)^2$
- Over $M$ examples: $L = \frac{1}{M} \sum_i L^i(W)$

Gradient of loss w.r.t. weights:
- Partial derivative of loss w.r.t. $k$th weight:
  $$\frac{\partial L^i}{\partial w_k} = \frac{\partial L^i}{\partial \hat{y}^i} \frac{\partial \hat{y}^i}{\partial w_k} = 2(\hat{y}^i - y^i)x_k^i$$

Output:
$$\hat{y} = w_1 x_1 + w_2 x_2 + w_3 x_3 + b = w^T x + b$$

Neural network parameters:
$$W = \{ [w_1, w_2, w_3], b \}$$
Optimizing the loss function: our example

\[ \hat{y} = w_1 x_1 + w_2 x_2 + w_3 x_3 + b \]

Output: \[ \hat{y} = w^T x + b \]

Neural network parameters:
\[ W = \{[w_1, w_2, w_3], b\} \]

Loss function:
Per-example: \[ L^i(W) = (\hat{y}^i - y^i)^2 \]
Over M examples: \[ L = \frac{1}{M} \sum_i L^i(W) \]

Gradient of loss w.r.t. weights:
Partial derivative of loss w.r.t. kth weight:
\[
\frac{\partial L^i}{\partial w_k} = \frac{\partial L^i}{\partial \hat{y}^i} \frac{\partial \hat{y}^i}{\partial w_k} = 2(\hat{y}^i - y^i)x_k^i
\]
\[
\frac{\partial L}{\partial w_k} = \frac{1}{M} \sum_i \frac{\partial L^i}{\partial w_k} = \frac{1}{M} \sum_i 2(\hat{y}^i - y^i)x_k^i
\]
Optimizing the loss function: our example

Loss function:
- Per-example: \( L^i(W) = (\hat{y}^i - y^i)^2 \)
- Over M examples: \( L = \frac{1}{M} \sum_i L^i(W) \)

Gradient of loss w.r.t. weights:
- Partial derivative of loss w.r.t. kth weight:
  \[
  \frac{\partial L^i}{\partial w_k} = \frac{\partial L^i}{\partial \hat{y}^i} \frac{\partial \hat{y}^i}{\partial w_k} = 2(\hat{y}^i - y^i)\ x_k^i
  \]
  \[
  \frac{\partial L}{\partial w_k} = \frac{1}{M} \sum_i \frac{\partial L^i}{\partial w_k} = \frac{1}{M} \sum_i 2(\hat{y}^i - y^i)\ x_k^i
  \]

Full gradient expression:
\[
\nabla L_W = \left[ \frac{\partial L}{\partial w_0}, \ldots, \frac{\partial L}{\partial w_3} \right] = \frac{1}{M} \sum_i 2(\hat{y}^i - y^i)\ x^i
\]

Output: \( \hat{y} = w_1 x_1 + w_2 x_2 + w_3 x_3 + b \)

Neural network parameters:
\( W = \{ [w_1, w_2, w_3], b \} \)
Optimizing the loss function: our example

```python
# initialize model parameters to be learned
W = np.random.rand(input_dim, output_dim)  # Initial model parameters randomly (dimensions [4,1] in our case)

# perform gradient descent
step_size = 1e-2
while(keep_training):
    # evaluate and accumulate the loss and the gradient of the loss function with respect to the weights, for each example
    d_W = 0
    for i in range(num_examples):
        x_i = X[np.newaxis,i,:]
        y_i = Y[i]
        y_hat_i = x_i.dot(W)
        loss += np.asscalar(y_hat_i - y_i)**2
        d_W += 2*(y_hat_i - y_i) * (x_i.T)
    # (1/M) in the loss and gradient expressions
    loss /= num_examples
    d_W /= num_examples
    # perform gradient update
    W = W - step_size * d_W
```
# initialize model parameters to be learned
W = np.random.rand(input_dim, output_dim)

# perform gradient descent
step_size = 1e-2
while(keep_training):
    # evaluate and accumulate the loss and the gradient of the loss function with respect
    # to the weights, for each example
    d_W = 0
    for i in range(num_examples):
        x_i = X[np.newaxis,i,:]
        y_i = Y[i]
        y_hat_i = x_i.dot(W)
        loss += np.asscalar(y_hat_i - y_i)**2
        d_W += 2*(y_hat_i - y_i) * (x_i.T)
    # (1/M) in the loss and gradient expressions
    loss /= num_examples
    d_W /= num_examples
    # perform gradient update
    W = W - step_size * d_W

Evaluate gradient
# initialize model parameters to be learned
W = np.random.rand(input_dim, output_dim)

# perform gradient descent
step_size = 1e-2
while(keep_training):
    # evaluate and accumulate the loss and the gradient of the loss function with respect to the weights, for each example
    d_W = 0
    for i in range(num_examples):
        x_i = X[np.newaxis,i,:]
        y_i = Y[i]
        y_hat_i = x_i.dot(W)
        loss += np.asscalar(y_hat_i - y_i)**2
        d_W += 2*(y_hat_i - y_i) * (x_i.T)
    # (1/M) in the loss and gradient expressions
    loss /= num_examples
    d_W /= num_examples
    # perform gradient update
    W := W - α∇L_W
    = W - α\frac{1}{M} \sum_i 2(\hat{y}^i - y^i)x^i
Optimizing the loss function: our example

```python
# initialize model parameters to be learned
W = np.random.randn(input_dim, output_dim)

# perform gradient descent
step_size = 1e-2
while(keep_training):
    # evaluate and accumulate the loss and the gradient of the loss function with respect
    # to the weights, for each example
    d_W = 0
    for i in range(num_examples):
        x_i = X[np.newaxis, i, :]
        y_i = Y[i]
        y_hat_i = x_i.dot(W)
        loss += np.asscalar(y_hat_i - y_i)**2
        d_W += 2*(y_hat_i - y_i) * (x_i.T)
    # (1/M) in the loss and gradient expressions
    loss /= num_examples
    d_W /= num_examples
    # perform gradient update
    W := W - α∇L_W
    = W - α\frac{1}{M} \sum_i 2(\hat{y}_i - y_i)x_i
```
In a more efficient vectorized form

```python
# initialize model parameters to be learned
W = np.random.rand(input_dim, output_dim)

# perform gradient descent
step_size = 1e-2

while(keep_training):
    # evaluate the loss and the gradient of the loss function with respect
    # to the weights, in vectorized form
    Y_curr = X.dot(W)
    loss = np.sum(np.square(Y_curr - Y)) / num_examples

    d_Y_curr = 2*(Y_curr - Y) / num_examples
    d_X = d_Y_curr.dot(W.T)
    d_W = X.T.dot(d_Y_curr)

    # perform gradient update
    W = W - step_size * d_W
```
In a more efficient vectorized form

```python
# initialize model parameters to be learned
W = np.random.rand(input_dim, output_dim)

# perform gradient descent
step_size = 1e-2

while(keep_training):
    # evaluate the loss and the gradient of the loss function with respect
    # to the weights, in vectorized form
    Y_curr = X.dot(W)
    loss = np.sum(np.square(Y_curr - Y)) / num_examples

    d_Y_curr = 2*(Y_curr - Y) / num_examples
    d_X = d_Y_curr.dot(W.T)
    d_W = X.T.dot(d_Y_curr)

    # perform gradient update
    W = W - step_size * d_W
```

X now has dimensions [num_examples, input_dim]. Compute loss for all examples at once (no more for loop).
Now: a two-layer fully-connected neural network

\[ W^1 = \begin{bmatrix} w_{11}^1 & w_{12}^1 & w_{13}^1 \\ w_{21}^1 & w_{22}^1 & w_{23}^1 \\ w_{31}^1 & w_{32}^1 & w_{33}^1 \end{bmatrix} \quad b^1 = \begin{bmatrix} b_1^1 \\ b_2^1 \\ b_3^1 \end{bmatrix} \]

\[ W^2 = \begin{bmatrix} w_{11}^2 & w_{12}^2 & w_{13}^2 \end{bmatrix} \quad b^2 = \begin{bmatrix} b_1^2 \end{bmatrix} \]

Output:

\[ h = \sigma(W^1 x + b^1) \]
\[ \hat{y} = W^2 h + b^2 \]
Now: a two-layer fully-connected neural network

Output: \( h = \sigma(W^1 x + b^1) \)
\( \hat{y} = W^2 h + b^2 \)

Full function expression:
\( \hat{y} = W^2(\sigma(W^1 x + b^1)) + b^2 \)

\[
W^1 = \begin{bmatrix}
    w_{11}^1 & w_{12}^1 & w_{13}^1 \\
    w_{21}^1 & w_{22}^1 & w_{23}^1 \\
    w_{31}^1 & w_{32}^1 & w_{33}^1
\end{bmatrix}
\]
\[
b^1 = \begin{bmatrix}
    b_1^1 \\
    b_2^1 \\
    b_3^1
\end{bmatrix}
\]

\[
W^2 = \begin{bmatrix}
    w_{11}^2 & w_{12}^2 & w_{13}^2
\end{bmatrix}
\]
\[
b^2 = \begin{bmatrix}
    b_1^2
\end{bmatrix}
\]
Now: a two-layer fully-connected neural network

\[
\begin{align*}
W^1 &= \begin{bmatrix}
w_{11}^1 & w_{12}^1 & w_{13}^1 \\
w_{21}^1 & w_{22}^1 & w_{23}^1 \\
w_{31}^1 & w_{32}^1 & w_{33}^1
\end{bmatrix} & b^1 &= \begin{bmatrix}
b_1^1 \\
b_2^1 \\
b_3^1
\end{bmatrix} \\
W^2 &= \begin{bmatrix}
w_{11}^2 & w_{12}^2 & w_{13}^2
\end{bmatrix} & b^2 &= \begin{bmatrix}
b_1^2
\end{bmatrix}
\end{align*}
\]

Output: 
\[
h = \sigma(W^1 x + b^1)
\]
\[
\hat{y} = W^2 h + b^2
\]

Full function expression:
\[
\hat{y} = W^2 (\sigma(W^1 x + b^1)) + b^2
\]
Now: a two-layer fully-connected neural network

Output: 
\[ h = \sigma(W^1 x + b^1) \]
\[ \hat{y} = W^2 h + b^2 \]

Full function expression:
\[ \hat{y} = W^2(\sigma(W^1 x + b^1)) + b^2 \]

Activation functions introduce non-linearity into the model -- allowing it to represent highly complex functions.

Sigmoid “activation function”

\[ \sigma(a) = \frac{1}{1 + e^{-a}} \]
Now: a two-layer fully-connected neural network

Output: $h = \sigma(W^1x + b^1)$
$\hat{y} = W^2h + b^2$

Full function expression:
$\hat{y} = W^2(\sigma(W^1x + b^1)) + b^2$

Activation functions introduce non-linearity into the model -- allowing it to represent highly complex functions.

A fully-connected neural network (also known as multi-layer perceptron) is a stack of [affine transformation + activation function] layers. There is no activation function at the last layer.
Now: a two-layer fully-connected neural network

Output: \[ \hat{y} = W^2(\sigma(W^1 x + b^1)) + b^2 \]

\[ W^1 = \begin{bmatrix} w_{11}^1 & w_{12}^1 & w_{13}^1 \\ w_{21}^1 & w_{22}^1 & w_{23}^1 \\ w_{31}^1 & w_{32}^1 & w_{33}^1 \end{bmatrix} \quad b^1 = \begin{bmatrix} b_1^1 \\ b_2^1 \\ b_3^1 \end{bmatrix} \]

\[ W^2 = \begin{bmatrix} w_{11}^2 & w_{12}^2 & w_{13}^2 \end{bmatrix} \quad b^2 = \begin{bmatrix} b_1^2 \end{bmatrix} \]
Now: a two-layer fully-connected neural network

Output: \( \hat{y} = W^2(\sigma(W^1 x + b^1)) + b^2 \)

Neural network parameters:
\[ W = \{W^1, b^1, W^2, b^2\} \]

\[
W^1 = \begin{bmatrix} w_{11}^1 & w_{12}^1 & w_{13}^1 \\ w_{21}^1 & w_{22}^1 & w_{23}^1 \\ w_{31}^1 & w_{32}^1 & w_{33}^1 \end{bmatrix} \quad b^1 = \begin{bmatrix} b_1^1 \\ b_2^1 \\ b_3^1 \end{bmatrix}
\]

\[
W^2 = \begin{bmatrix} w_{11}^2 & w_{12}^2 & w_{13}^2 \end{bmatrix} \quad b^2 = \begin{bmatrix} b_1^2 \end{bmatrix}
\]
Now: a two-layer fully-connected neural network

Output: \[ \hat{y} = W^2(\sigma(W^1 x + b^1)) + b^2 \]

Neural network parameters:
\[ W = \{W^1, b^1, W^2, b^2\} \]

Loss function (regression loss, same as before):
Per-example: \[ L^i(W) = (\hat{y}^i - y^i)^2 \]
Over M examples: \[ L = \frac{1}{M} \sum_i L^i(W) \]

\[
W^1 = \begin{bmatrix}
w_{11} & w_{12} & w_{13} \\
w_{21} & w_{22} & w_{23} \\
w_{31} & w_{32} & w_{33}
\end{bmatrix} \quad b^1 = \begin{bmatrix}
b_1^1 \\
b_2^1 \\
b_3^1
\end{bmatrix}
\]

\[
W^2 = \begin{bmatrix}
w_{11}^2 & w_{12}^2 & w_{13}^2 \\
w_{12}^2 & w_{12}^2 & w_{13}^2
\end{bmatrix} \quad b^2 = \begin{bmatrix}
b_1^2
\end{bmatrix}
\]
Now: a two-layer fully-connected neural network

Output: \( \hat{y} = W^2(\sigma(W^1 x + b^1)) + b^2 \)

Neural network parameters:
\[
W = \{W^1, b^1, W^2, b^2\}
\]

Loss function (regression loss, same as before):

Per-example: \( L^i(W) = (\hat{y}^i - y^i)^2 \)

Over M examples: \( L = \frac{1}{M} \sum_i L^i(W) \)

Gradient of loss w.r.t. weights:
Function more complex -> now much harder to derive the expressions! Instead… computational graphs and backpropagation.
Computing gradients with backpropagation
Computing gradients with backpropagation

Network output: \( \hat{y} = W^2(\sigma(W^1 x + b^1)) + b^2 \)

Think of computing loss function as staged computation of intermediate variables:
Computing gradients with backpropagation

Network output: \( \hat{y} = W^2(\sigma(W^1 x + b^1)) + b^2 \)

Think of computing loss function as staged computation of intermediate variables:

```
\begin{align*}
W^1 & \quad W^2 \\
\vphantom{W^1} x & \quad \vphantom{W^2} z \\
\vphantom{W^1} \vphantom{W^2} b^1 & \quad \vphantom{W^1} \vphantom{W^2} h \\
\vphantom{W^1} \vphantom{W^2} \vphantom{b^1} y & \quad \vphantom{W^1} \vphantom{W^2} \vphantom{b^1} \hat{y} \\
\end{align*}
```

"Forward pass":

\begin{align*}
z &= W^1 x + b^1 \\
h &= \sigma(z) \\
\hat{y} &= W^2 h + b^2 \\
L &= (\hat{y} - y)^2
\end{align*}
Computing gradients with backpropagation

Network output: \[ \hat{y} = W^2(\sigma(W^1 x + b^1)) + b^2 \]

Think of computing loss function as staged computation of intermediate variables:

```
W^1
x
z
W^2
h
\hat{y}
L
b^1
b^2
y
```

“Forward pass”: 
\[
\begin{align*}
  z &= W^1 x + b^1 \\
  h &= \sigma(z) \\
  \hat{y} &= W^2 h + b^2 \\
  L &= (\hat{y} - y)^2
\end{align*}
\]

Now, can use a repeated application of the chain rule, going backwards through the computational graph, to obtain the gradient of the loss with respect to each node of the computation graph.
Computing gradients with backpropagation

Network output: \( \hat{y} = W^2(\sigma(W^1 x + b^1)) + b^2 \)

Think of computing loss function as staged computation of intermediate variables:

```
Network output: \( \hat{y} = W^2(\sigma(W^1 x + b^1)) + b^2 \)
```

Now, can use a repeated application of the chain rule, going backwards through the computational graph, to obtain the gradient of the loss with respect to each node of the computation graph.

```
“Forward pass”:  
\begin{align*}
  z &= W^1 x + b^1 \\
  h &= \sigma(z) \\
  \hat{y} &= W^2 h + b^2 \\
  L &= (\hat{y} - y)^2
\end{align*}
```

```
“Backward pass”:  
\begin{align*}
  \frac{\partial L}{\partial \hat{y}} &= 2(\hat{y} - y) \\
  \frac{\partial L}{\partial W^2} &= \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial W^2} \\
  \frac{\partial L}{\partial H} &= \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial H} \\
  \frac{\partial L}{\partial Z} &= \frac{\partial L}{\partial H} \frac{\partial H}{\partial Z} \\
  \frac{\partial L}{\partial W^1} &= \frac{\partial L}{\partial Z} \frac{\partial Z}{\partial W^1} \\
\end{align*}
```

(not all gradients shown)
Computing gradients with backpropagation

Network output: \( \hat{y} = W^2(\sigma(W^1x + b^1)) + b^2 \)

Think of computing loss function as staged computation of intermediate variables:

Now, can use a repeated application of the chain rule, going backwards through the computational graph, to obtain the gradient of the loss with respect to each node of the computation graph.

“Forward pass”:
- \( z = W^1x + b^1 \)
- \( h = \sigma(z) \)
- \( \hat{y} = W^2h + b^2 \)
- \( L = (\hat{y} - y)^2 \)

“Backward pass”:
- \( \frac{\partial L}{\partial \hat{y}} = 2(\hat{y} - y) \) (not all gradients shown)

Plug in from earlier computations via chain rule:

\[
\begin{align*}
\frac{\partial L}{\partial W^2} &= \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial W^2} \\
\frac{\partial L}{\partial W^1} &= \frac{\partial L}{\partial H} \frac{\partial H}{\partial W^1} \\
\frac{\partial L}{\partial Z} &= \frac{\partial L}{\partial H} \frac{\partial H}{\partial Z} \\
\frac{\partial L}{\partial z} &= \frac{\partial L}{\partial H} \frac{\partial H}{\partial z}
\end{align*}
\]
Computing gradients with backpropagation

Network output: \( \hat{y} = W^2(\sigma(W^1 x + b^1)) + b^2 \)

Think of computing loss function as staged computation of intermediate variables:

Now, can use a repeated application of the chain rule, going backwards through the computational graph, to obtain the gradient of the loss with respect to each node of the computation graph.

“Forward pass”:
- \( z = W^1 x + b^1 \)
- \( h = \sigma(z) \)
- \( \hat{y} = W^2 h + b^2 \)
- \( L = (\hat{y} - y)^2 \)

“Backward pass”:
- \( \frac{\partial L}{\partial \hat{y}} = 2(\hat{y} - y) \)
- \( \frac{\partial L}{\partial W^2} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial W^2} \)
- \( \frac{\partial L}{\partial H} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial H} \)
- \( \frac{\partial L}{\partial Z} = \frac{\partial L}{\partial H} \frac{\partial H}{\partial Z} \)
- \( \frac{\partial L}{\partial W^1} = \frac{\partial L}{\partial Z} \frac{\partial Z}{\partial W^1} \)

Plug in from earlier computations via chain rule

Local gradients to derive

(not all gradients shown)
Computing gradients with backpropagation

**Key idea:** Don’t mathematically derive entire math expression for e.g. $dL / dW^1$. By writing it as nested applications of the chain rule, only have to derive simple “local” gradients representing relationships between connected nodes of the graph (e.g. $dH / dW^1$).
Computing gradients with backpropagation

**Key idea:** Don’t mathematically derive entire math expression for e.g. dL / dW¹. By writing it as nested applications of the chain rule, only have to derive simple “local” gradients representing relationships between connected nodes of the graph (e.g. dH / dW¹).

Can use more or less intermediate variables to control how difficult local gradients are to derive!
A diagram showing a function $f$ with inputs $x$ and $y$, and an output $z$. The slide credit is to CS231n.
The image illustrates a function $f$ with inputs $x$ and $y$, and output $z$. The function has a "local gradient" which can be represented as the partial derivatives $\frac{\partial z}{\partial x}$ and $\frac{\partial z}{\partial y}$.
The diagram illustrates the flow of gradients through a function $f$. The input $x$ and $y$ are applied to $f$, which outputs $z$. The partial derivatives are shown as follows:

- $\frac{\partial z}{\partial x}$
- $\frac{\partial z}{\partial y}$
- $\frac{\partial L}{\partial z}$

The terms "local gradient" and "upstream gradient" are highlighted.
\[
\frac{\partial L}{\partial x} = \frac{\partial L}{\partial z} \frac{\partial z}{\partial x}
\]

"local gradient"

"Downstream gradients"

"Upstream gradient"
The diagram illustrates the computation of gradients in a function $f$. The "local gradient" is shown with $\frac{\partial L}{\partial x} = \frac{\partial L}{\partial z} \frac{\partial z}{\partial x}$.

The "Downstream gradients" are denoted as $\frac{\partial L}{\partial y} = \frac{\partial L}{\partial z} \frac{\partial z}{\partial y}$.

The "Upstream gradient" is represented as $\frac{\partial L}{\partial z}$.

The diagram uses the notation $x$, $y$, and $z$ to illustrate the flow of gradients.

Slide credit: CS231n
"Downstream gradients"\[\frac{\partial L}{\partial y} = \frac{\partial L}{\partial z} \frac{\partial z}{\partial y}\]

"Upstream gradient"\[\frac{\partial L}{\partial z}\]

"local gradient"\[\frac{\partial z}{\partial x}\]

Slide credit: CS231n
Training our two-layer neural network in code, using backpropagation

```python
# initialize model parameters to be learned
W1 = np.random.rand(input_dim, hid_dim)
W2 = np.random.rand(hid_dim, output_dim)
b1 = np.random.rand(1, hid_dim)
b2 = np.random.rand(1, output_dim)

# perform gradient descent
step_size = 1e-2

while(keep_training):

    # forward pass, computing loss
    Z_curr = X.dot(W1) + b1
    H_curr = sigmoid_array(Z_curr)
    Y_curr = H_curr.dot(W2) + b2
    loss = np.sum(np.square(Y_curr - Y)) / num_examples

    # backward pass, computing gradients of loss with respect to each variable in the computation graph
    d_Y_curr = 2*(Y_curr - Y) / num_examples
    d_H_curr = d_Y_curr.dot(W2.T)
    d_W2 = H_curr.T.dot(d_Y_curr)
    d_b2 = d_Y_curr
    d_Z_curr = d_H_curr * sigmoid_array(Z_curr)*(1-sigmoid_array(Z_curr))
    d_X = d_Z_curr.dot(W1.T)
    d_W1 = X.T.dot(d_Z_curr)
    d_b1 = d_Y_curr

    # perform gradient update
    W1 = W1 - step_size * d_W1
    b1 = b1 - step_size * d_b1
    W2 = W2 - step_size * d_W2
    b2 = b2 - step_size * d_b2
```
Training our two-layer neural network in code, using backpropagation

```python
# initialize model parameters to be learned
W1 = np.random.rand(input_dim, hid_dim)
W2 = np.random.rand(hid_dim, output_dim)
b1 = np.random.rand(1, hid_dim)
b2 = np.random.rand(1, output_dim)

# perform gradient descent
step_size = 1e-2
while(keep_training):
    # forward pass, computing loss
    Z_curr = X.dot(W1) + b1
    H_curr = sigmoid_array(Z_curr)
    Y_curr = H_curr.dot(W2) + b2
    loss = np.sum(np.square(Y_curr - Y)) / num_examples

    # backward pass, computing gradients of loss with respect to each
    # variable in the computation graph
    d_Y_curr = 2*(Y_curr - Y) / num_examples
    d_H_curr = d_Y_curr.dot(W2.T)
    d_W2 = H_curr.T.dot(d_Y_curr)
    d_b2 = d_Y_curr
    d_Z_curr = d_H_curr * sigmoid_array(Z_curr)*(1-sigmoid_array(Z_curr))
    d_X = d_Z_curr.dot(W1.T)
    d_W1 = d_X.T.dot(d_Z_curr)
    d_b1 = d_Y_curr

    # perform gradient update
    W1 = W1 - step_size * d_W1
    b1 = b1 - step_size * d_b1
    W2 = W2 - step_size * d_W2
    b2 = b2 - step_size * d_b2
```
# Training our two-layer neural network in code, using backpropagation

```python
# initialize model parameters to be learned
W1 = np.random.randn(input_dim, hid_dim)
W2 = np.random.randn(hid_dim, output_dim)
b1 = np.random.rand(1, hid_dim)
b2 = np.random.rand(1, output_dim)

# perform gradient descent
step_size = 1e-2

while(keep_training):
    # forward pass, computing loss
    Z_curr = X.dot(W1) + b1
    H_curr = sigmoid_array(Z_curr)
    Y_curr = H_curr.dot(W2) + b2
    loss = np.sum(np.square(Y_curr - Y)) / num_examples

    # backward pass, computing gradients of loss with respect to each
    # variable in the computation graph
    d_Y_curr = 2*(Y_curr - Y) / num_examples
    d_H_curr = d_Y_curr.dot(W2.T)
    d_W2 = H_curr.T.dot(d_Y_curr)
    d_b2 = d_Y_curr
    d_Z_curr = d_H_curr * sigmoid_array(Z_curr)*(1-sigmoid_array(Z_curr))
    d_X = d_Z_curr.dot(W1.T)
    d_W1 = d_X.T.dot(d_Z_curr)
    d_b1 = d_Y_curr

    # perform gradient update
    W1 = W1 - step_size * d_W1
    b1 = b1 - step_size * d_b1
    W2 = W2 - step_size * d_W2
    b2 = b2 - step_size * d_b2
```
Training our two-layer neural network in code, using backpropagation

```python
# initialize model parameters to be learned
W1 = np.random.rand(input_dim, hid_dim)
W2 = np.random.rand(hid_dim, output_dim)
b1 = np.random.rand(1, hid_dim)
b2 = np.random.rand(1, output_dim)

# perform gradient descent
step_size = 1e-2

while(keep_training):

    # forward pass, computing loss
    Z_curr = X.dot(W1) + b1
    H_curr = sigmoid_array(Z_curr)
    Y_curr = H_curr.dot(W2) + b2
    loss = np.sum(np.square(Y_curr - Y)) / num_examples

    # backward pass, computing gradients of loss with respect to each
    # variable in the computation graph
    d_Y_curr = 2*(Y_curr - Y) / num_examples
    d_H_curr = d_Y_curr.dot(W2.T)
    d_W2 = H_curr.T.dot(d_Y_curr)
    d_b2 = d_Y_curr
    d_Z_curr = d_H_curr * sigmoid_array(Z_curr)*(1-sigmoid_array(Z_curr))
    d_X = d_Z_curr.dot(W1.T)
    d_W1 = X.T.dot(d_Z_curr)
    d_b1 = d_Y_curr

    # perform gradient update
    W1 = W1 - step_size * d_W1
    b1 = b1 - step_size * d_b1
    W2 = W2 - step_size * d_W2
    b2 = b2 - step_size * d_b2
```
Training our two-layer neural network in code, using backpropagation.

```python
# initialize model parameters to be learned
W1 = np.random.rand(input_dim, hid_dim)
W2 = np.random.rand(hid_dim, output_dim)
b1 = np.random.rand(1, hid_dim)
b2 = np.random.rand(1, output_dim)

# perform gradient descent
step_size = 1e-2

while(keep_training):

    # forward pass, computing loss
    Z_curr = X.dot(W1) + b1
    H_curr = sigmoid_array(Z_curr)
    Y_curr = H_curr.dot(W2) + b2
    loss = np.sum(np.square(Y_curr - Y)) / num_examples

    # backward pass, computing gradients of loss with respect to each
    # variable in the computation graph
    d_Y_curr = 2*(Y_curr - Y) / num_examples
    d_H_curr = d_Y_curr.dot(W2.T)
    d_W2 = H_curr.T.dot(d_Y_curr)
    d_b2 = d_Y_curr
    d_Z_curr = d_H_curr * sigmoid_array(Z_curr)*(1-sigmoid_array(Z_curr))
    d_X = d_Z_curr.dot(W1.T)
    d_W1 = d_X.T.dot(d_Z_curr)
    d_b1 = d_Y_curr

    # perform gradient update
    W1 = W1 - step_size * d_W1
    b1 = b1 - step_size * d_b1
    W2 = W2 - step_size * d_W2
    b2 = b2 - step_size * d_b2
```

Upstream gradient  
Downstream gradient  
Backward pass
Training our two-layer neural network in code, using backpropagation.

```python
# initialize model parameters to be learned
W1 = np.random.rand(input_dim, hid_dim)
W2 = np.random.rand(hid_dim, output_dim)
b1 = np.random.rand(1, hid_dim)
b2 = np.random.rand(1, output_dim)

# perform gradient descent
step_size = 1e-2
while(keep_training):
    # forward pass, computing loss
    Z_curr = X.dot(W1) + b1
    H_curr = np.tanh(Z_curr)
    Y_curr = H_curr.dot(W2) + b2
    loss = np.sum(np.square(Y_curr - Y)) / num_examples

    # backward pass, computing gradients of loss with respect to each variable in the computation graph
    d_Y_curr = 2*(Y_curr - Y) / num_examples
    d_H_curr = d_Y_curr.dot(W2.T)
    d_W2 = H_curr.T.dot(d_Y_curr)
    d_b2 = d_Y_curr
    d_Z_curr = d_H_curr * sigmoid_array(Z_curr)*(1-sigmoid_array(Z_curr))
    d_X = d_Z_curr.dot(W1.T)
    d_W1 = X.T.dot(d_Z_curr)
    d_b1 = d_Y_curr

    # perform gradient update
    W1 = W1 - step_size * d_W1
    b1 = b1 - step_size * d_b1
    W2 = W2 - step_size * d_W2
    b2 = b2 - step_size * d_b2
```

Note: sometimes we see transposes and flipped dot product order. These are required to make the dimensions of the vectorized computations work out, such that each element of the matrices and vectors are as you would expect. dW should have the same dimensions as W! Can try working out element expressions for yourself... but won’t focus on this in this class.
Deep learning software frameworks

- Makes our lives easier by providing implementations and higher-level abstractions of many components for deep learning, and running them on GPUs:
  - Dataset batching, model definition, gradient computation, optimization, etc.
Deep learning software frameworks

- Makes our lives easier by providing implementations and higher-level abstractions of many components for deep learning, and running them on GPUs:
  - Dataset batching, model definition, gradient computation, optimization, etc.

- Automatic differentiation: if we define nodes in a computational graph, will automatically implement backpropagation for us
  - Supports many common operations with local gradients already implemented
  - Can still define custom operations
Deep learning software frameworks

- Makes our lives easier by providing implementations and higher-level abstractions of many components for deep learning, and running them on GPUs:
  - Dataset batching, model definition, gradient computation, optimization, etc.

- Automatic differentiation: if we define nodes in a computational graph, will automatically implement backpropagation for us
  - Supports many common operations with local gradients already implemented
  - Can still define custom operations

- A number of popular options, e.g. Tensorflow and PyTorch. Recent stable versions (TF 2.0, PyTorch 1.3) work largely in a similar fashion (not necessarily true for earlier versions). We will use Tensorflow 2.0 in this class.
Training our two-layer neural network in code, in Tensorflow 2.0

```python
# Our (X,Y) training set converted to TF tensors
X_tf = tf.convert_to_tensor(X, np.float32)
Y_tf = tf.convert_to_tensor(Y, np.float32)

# Create a TF dataset with specified minibatch size
batch_size = 50
dataset = tf.data.Dataset.from_tensor_slices((X_tf, Y_tf))
dataset = dataset.batch(batch_size)

# initialize model parameters to be learned
W1 = tf.Variable(tf.random.uniform((input_dim, hid_dim)))
W2 = tf.Variable(tf.random.uniform((hid_dim, output_dim)))
b1 = tf.Variable(tf.random.uniform((1, hid_dim)))
b2 = tf.Variable(tf.random.uniform((1, output_dim)))

# perform gradient descent
epochs = 5000
optimizer = tf.optimizers.SGD(learning_rate=1e-2)
losses = []

for epoch in range(epochs):
    for batch in dataset:
        X_batch, Y_batch = batch
        with tf.GradientTape() as tape:

            # forward pass
            H_batch = tf.add(tf.matmul(X_batch, W1), b1)
            Z_batch = tf.math.sigmoid(H_batch)
            Out_batch = tf.add(tf.matmul(H_batch, W2), b2)
            loss = tf.losses.MSE(Y_batch, Out_batch)

            # backward pass and gradient update
            gradients = tape.gradient(loss, [W1, W2, b1, b2])
            optimizer.apply_gradients(zip(gradients, [W1, W2, b1, b2]))
```
Training our two-layer neural network in code, in Tensorflow 2.0

```python
# Our (X,Y) training set converted to TF tensors
X_tf = tf.convert_to_tensor(X, np.float32)
Y_tf = tf.convert_to_tensor(Y, np.float32)

# Create a TF dataset with specified minibatch size
batch_size = 50
dataset = tf.data.Dataset.from_tensor_slices((X_tf, Y_tf))
dataset = dataset.batch(batch_size)

# initialize model parameters to be learned
W1 = tf.Variable(tf.random.uniform((input_dim, hid_dim)))
W2 = tf.Variable(tf.random.uniform((hid_dim, output_dim)))
b1 = tf.Variable(tf.random.uniform((1, hid_dim)))
b2 = tf.Variable(tf.random.uniform((1, output_dim)))

# perform gradient descent
epochs = 5000
optimizer = tf.optimizers.SGD(learning_rate=1e-2)
losses = []

for epoch in range(epochs):
    for batch in dataset:
        X_batch, Y_batch = batch
        with tf.GradientTape() as tape:

            # forward pass
            H_batch = tf.add(tf.matmul(X_batch, W1), b1)
            H_batch = tf.math.sigmoid(H_batch)
            Out_batch = tf.add(tf.matmul(H_batch, W2), b2)
            loss = tf.losses.MSE(Y_batch, Out_batch)

            # backward pass and gradient update
            gradients = tape.gradient(loss, [W1, W2, b1, b2])
            optimizer.apply_gradients(zip(gradients, [W1, W2, b1, b2]))
```

Convert data to TF tensors, create a TF dataset
Training our two-layer neural network in code, in Tensorflow 2.0

```python
# Our (X,Y) training set converted to TF tensors
X_tf = tf.convert_to_tensor(X, np.float32)
Y_tf = tf.convert_to_tensor(Y, np.float32)

# Create a TF dataset with specified minibatch size
batch_size = 50
dataset = tf.data.Dataset.from_tensor_slices((X_tf, Y_tf))
dataset = dataset.batch(batch_size)

# Initialize model parameters to be learned
W1 = tf.Variable(tf.random.uniform((input_dim, hid_dim)))
W2 = tf.Variable(tf.random.uniform((hid_dim, output_dim)))
b1 = tf.Variable(tf.random.uniform((1, hid_dim)))
b2 = tf.Variable(tf.random.uniform((1, output_dim)))

# Perform gradient descent
epochs = 5000
optimizer = tf.optimizers.SGD(learning_rate=1e-2)
losses = []

for epoch in range(epochs):
    for batch in dataset:
        X_batch, Y_batch = batch
        with tf.GradientTape() as tape:

            # Forward pass
            H_batch = tf.add(tf.matmul(X_batch, W1), b1)
            H_batch = tf.math.sigmoid(H_batch)
            Out_batch = tf.add(tf.matmul(H_batch, W2), b2)
            loss = tf.losses.MSE(Y_batch, Out_batch)

            # Backward pass and gradient update
            gradients = tape.gradient(loss, [W1, W2, b1, b2])
            optimizer.apply_gradients(zip(gradients, [W1, W2, b1, b2]))
```

Initialize parameters to be learned as tf.Variable -> allows them to receive gradient updates during optimization
Training our two-layer neural network in code, in Tensorflow 2.0

```python
# Our (X,Y) training set converted to TF tensors
X_tf = tf.convert_to_tensor(X, np.float32)
Y_tf = tf.convert_to_tensor(Y, np.float32)

# Create a TF dataset with specified minibatch size
batch_size = 50
dataset = tf.data.Dataset.from_tensor_slices((X_tf, Y_tf))
dataset = dataset.batch(batch_size)

# Initialize model parameters to be learned
W1 = tf.Variable(tf.random.uniform((input_dim, hid_dim)))
W2 = tf.Variable(tf.random.uniform((hid_dim, output_dim)))
b1 = tf.Variable(tf.random.uniform((1, hid_dim)))
b2 = tf.Variable(tf.random.uniform((1, output_dim)))

# perform gradient descent
epochs = 5000
optimizer = tf.optimizers.SGD(learning_rate=1e-2)
losses = []

for epoch in range(epochs):
    for batch in dataset:
        X_batch, Y_batch = batch
        with tf.GradientTape() as tape:

            # forward pass
            _batch = tf.add(tf.matmul(X_batch, W1), b1)
            H_batch = tf.math.sigmoid(_batch)
            Out_batch = tf.add(tf.matmul(H_batch, W2), b2)
            loss = tf.losses.MSE(Y_batch, Out_batch)

            # backward pass and gradient update
            gradients = tape.gradient(loss, [W1, W2, b1, b2])
            optimizer.apply_gradients(zip(gradients, [W1, W2, b1, b2]))
```

Initialize a TF optimizer
Training our two-layer neural network in code, in Tensorflow 2.0

```python
# Our \((X, Y)\) training set converted to TF tensors
X_tf = tf.convert_to_tensor(X, np.float32)
Y_tf = tf.convert_to_tensor(Y, np.float32)

# Create a TF dataset with specified minibatch size
batch_size = 50
dataset = tf.data.Dataset.from_tensor_slices((X_tf, Y_tf))
dataset = dataset.batch(batch_size)

# initialize model parameters to be learned
W1 = tf.Variable(tf.random.uniform((input_dim, hid_dim)))
W2 = tf.Variable(tf.random.uniform((hid_dim, output_dim)))
b1 = tf.Variable(tf.random.uniform((1, hid_dim)))
b2 = tf.Variable(tf.random.uniform((1, output_dim)))

# perform gradient descent
epochs = 5000
optimizer = tf.optimizers.SGD(learning_rate=1e-2)
losses = []

for epoch in range(epochs):
    for batch in dataset:
        X_batch, Y_batch = batch
        with tf.GradientTape() as tape:
            # forward pass
            Z_batch = tf.add(tf.matmul(X_batch, W1), b1)
            H_batch = tf.math.sigmoid(Z_batch)
            Out_batch = tf.add(tf.matmul(H_batch, W2), b2)
            loss = tf.losses.MSE(Y_batch, Out_batch)

            # backward pass and gradient update
            gradients = tape.gradient(loss, [W1, W2, b1, b2])
            optimizer.apply_gradients(zip(gradients, [W1, W2, b1, b2]))
```

All operations defined under the gradient tape will be used to construct a computational graph
Training our two-layer neural network in code, in Tensorflow 2.0

```python
# Our \((X,Y)\) training set converted to TF tensors
X_tf = tf.convert_to_tensor(X, np.float32)
Y_tf = tf.convert_to_tensor(Y, np.float32)

# Create a TF dataset with specified minibatch size
batch_size = 50
dataset = tf.data.Dataset.from_tensor_slices((X_tf, Y_tf))
dataset = dataset.batch(batch_size)

# Initialize model parameters to be learned
W1 = tf.Variable(tf.random.uniform((input_dim, hid_dim)))
W2 = tf.Variable(tf.random.uniform((hid_dim, output_dim)))
b1 = tf.Variable(tf.random.uniform((1, hid_dim)))
b2 = tf.Variable(tf.random.uniform((1, output_dim)))

# Perform gradient descent
epochs = 5000
optimizer = tf.optimizers.SGD(learning_rate=1e-2)
losses = []

for epoch in range(epochs):
    for batch in dataset:
        X_batch, Y_batch = batch
        with tf.GradientTape() as tape:
            # Forward pass
            H_batch = tf.matmul(X_batch, W1) + b1
            H_batch = tf.math.sigmoid(H_batch)
            Out_batch = tf.matmul(H_batch, W2) + b2
            loss = tf.losses.MSE(Y_batch, Out_batch)

            # Backward pass and gradient update
            gradients = tape.gradient(loss, [W1, W2, b1, b2])
            optimizer.apply_gradients(zip(gradients, [W1, W2, b1, b2]))
```

The computational graph for our two-layer neural network
Training our two-layer neural network in code, in Tensorflow 2.0

```
# Our (X,Y) training set converted to TF tensors
X_tf = tf.convert_to_tensor(X, np.float32)
Y_tf = tf.convert_to_tensor(Y, np.float32)

# Create a TF dataset with specified minibatch size
batch_size = 50
dataset = tf.data.Dataset.from_tensor_slices((X_tf, Y_tf))
dataset = dataset.batch(batch_size)

# initialize model parameters to be learned
W1 = tf.Variable(tf.random.uniform((input_dim, hid_dim)))
W2 = tf.Variable(tf.random.uniform((hid_dim, output_dim)))
b1 = tf.Variable(tf.random.uniform((1, hid_dim)))
b2 = tf.Variable(tf.random.uniform((1, output_dim)))

# perform gradient descent
epochs = 5000
optimizer = tf.optimizers.SGD(learning_rate=1e-2)
losses = []

for epoch in range(epochs):
    for batch in dataset:
        X_batch, Y_batch = batch
        with tf.GradientTape() as tape:

            # forward pass
            H_batch = tf.add(tf.matmul(X_batch, W1), b1)
            H_batch = tf.math.sigmoid(H_batch)
            Out_batch = tf.add(tf.matmul(H_batch, W2), b2)
            loss = tf.losses.MSE(Y_batch, Out_batch)

            # backward pass and gradient update
            gradients = tape.gradient(loss, [W1, W2, b1, b2])
            optimizer.apply_gradients(zip(gradients, [W1, W2, b1, b2]))

        losses.append(loss.numpy())
```

Evaluate gradients using automatic differentiation and perform gradient update
Also high level libraries built on top of Tensorflow, that provide even easier-to-use APIs:

**In Tensorflow 2.0:**

```plaintext
for epoch in range(epochs):
    for batch in dataset:
        X_batch, Y_batch = batch
        with tf.GradientTape() as tape:

            # forward pass
            Z_batch = tf.add(tf.matmul(X_batch, W1), b1)
            H_batch = tf.math.sigmoid(Z_batch)
            Out_batch = tf.add(tf.matmul(H_batch, W2), b2)
            loss = tf.losses.MSE(Y_batch, Out_batch)

            # backward pass and gradient update
            gradients = tape.gradient(loss, [W1, W2, b1, b2])
            optimizer.apply_gradients(zip(gradients, [W1, W2, b1, b2]))
```

**In Keras:**

```plaintext
keras_model = tf.keras.models.Sequential([
    tf.keras.layers.Dense(units=3, activation='sigmoid', use_bias=True),
    tf.keras.layers.Dense(units=1, use_bias=True)
])
keras_model.compile(optimizer=tf.keras.optimizers.SGD(learning_rate=1e-2),
                    loss='mse')
k keras model.fit(dataset, epochs=1000)
```
Also high level libraries built on top of Tensorflow, that provide even easier-to-use APIs:

**In Tensorflow 2.0:**

```python
for epoch in range(epochs):
    for batch in dataset:
        X_batch, Y_batch = batch
        with tf.GradientTape() as tape:
            # forward pass
            Z_batch = tf.add(tf.matmul(X_batch, W1), b1)
            H_batch = tf.math.sigmoid(Z_batch)
            Out_batch = tf.add(tf.matmul(H_batch, W2), b2)
            loss = tf.losses.MSE(Y_batch, Out_batch)

            # backward pass and gradient update
            gradients = tape.gradient(loss, [W1, W2, b1, b2])
            optimizer.apply_gradients(zip(gradients, [W1, W2, b1, b2]))
```

**In Keras:**

```python
keras_model = tf.keras.models.Sequential([
    tf.keras.layers.Dense(units=3, activation='sigmoid', use_bias=True),
    tf.keras.layers.Dense(units=1, use_bias=True)
])
keras_model.compile(optimizer=tf.keras.optimizers.SGD(learning_rate=1e-2),
                    loss='mse')
keras_model.fit(dataset, epochs=1000)
```
Also high level libraries built on top of Tensorflow, that provide even easier-to-use APIs:

In Tensorflow 2.0:

```python
for epoch in range(epochs):
    for batch in dataset:
        X_batch, Y_batch = batch
        with tf.GradientTape() as tape:

            # forward pass
            Z_batch = tf.add(tf.matmul(X_batch, W1), b1)
            H_batch = tf.math.sigmoid(Z_batch)
            Out_batch = tf.add(tf.matmul(H_batch, W2), b2)
            loss = tf.losses.MSE(Y_batch, Out_batch)

            # backward pass and gradient update
            gradients = tape.gradient(loss, [W1, W2, b1, b2])
            optimizer.apply_gradients(zip(gradients, [W1, W2, b1, b2]))
```

In Keras:

```python
keras_model = tf.keras.models.Sequential([tf.keras.layers.Dense(units=3, activation='sigmoid', use_bias=True),
                                          tf.keras.layers.Dense(units=1, use_bias=True)])
keras_model.compile(optimizer=tf.keras.optimizers.SGD(learning_rate=1e-2),
                    loss='mse')
keras_model.fit(dataset, epochs=1000)
```

[Image: Diagram showing a fully-connected layer]
Also high level libraries built on top of Tensorflow, that provide even easier-to-use APIs:

In Tensorflow 2.0:

```python
for epoch in range(epochs):
    for batch in dataset:
        X_batch, Y_batch = batch
        with tf.GradientTape() as tape:
            # forward pass
            Z_batch = tf.add(tf.matmul(X_batch, W1), b1)
            H_batch = tf.math.sigmoid(Z_batch)
            Out_batch = tf.add(tf.matmul(H_batch, W2), b2)
            loss = tf.losses.MSE(Y_batch, Out_batch)

            # backward pass and gradient update
            gradients = tape.gradient(loss, [W1, W2, b1, b2])
            optimizer.apply_gradients(zip(gradients, [W1, W2, b1, b2]))
```

In Keras:

```python
keras_model = tf.keras.models.Sequential([
    tf.keras.layers.Dense(units=3, activation='sigmoid', use_bias=True),
    tf.keras.layers.Dense(units=1, use_bias=True)
])
keras_model.compile(optimizer=tf.keras.optimizers.SGD(learning_rate=1e-2),
                    loss='mse')
keras_model.fit(dataset, epochs=1000)
```

Activation function and bias configurations included!
Training more complex neural networks is a straightforward extension

```python
keras_model = tf.keras.models.Sequential([  
    tf.keras.layers.Dense(units=3, activation='sigmoid', use_bias=True),  
    tf.keras.layers.Dense(units=3, activation='sigmoid', use_bias=True),  
    tf.keras.layers.Dense(units=3, activation='sigmoid', use_bias=True),  
    tf.keras.layers.Dense(units=3, activation='sigmoid', use_bias=True),  
    tf.keras.layers.Dense(units=3, activation='sigmoid', use_bias=True),  
    tf.keras.layers.Dense(units=1, use_bias=True)
])
keras_model.compile(optimizer=tf.keras.optimizers.SGD(learning_rate=1e-2),  
                    loss='mse')
k keras_model.fit(dataset, epochs=1000)
```

Now a 6-layer network
Summary

- You now know how to define and train a neural network!

- Friday’s section (1:30-2:50pm McCullough 115) will provide a more in-depth tutorial on Tensorflow, and a hands-on exercise for training a model on MNIST -- highly encouraged!

- Next time: will go more in-depth on each of the components we talked about today:
  - Preparing data for deep learning
  - Different neural network architectures
  - Different loss functions
  - Optimization choices
  - Training models in practice
  - Etc.