Supervised Machine Learning and Learning Theory

Lecture 14: The backpropagation algorithm

October 22, 2024

Warm-up questions

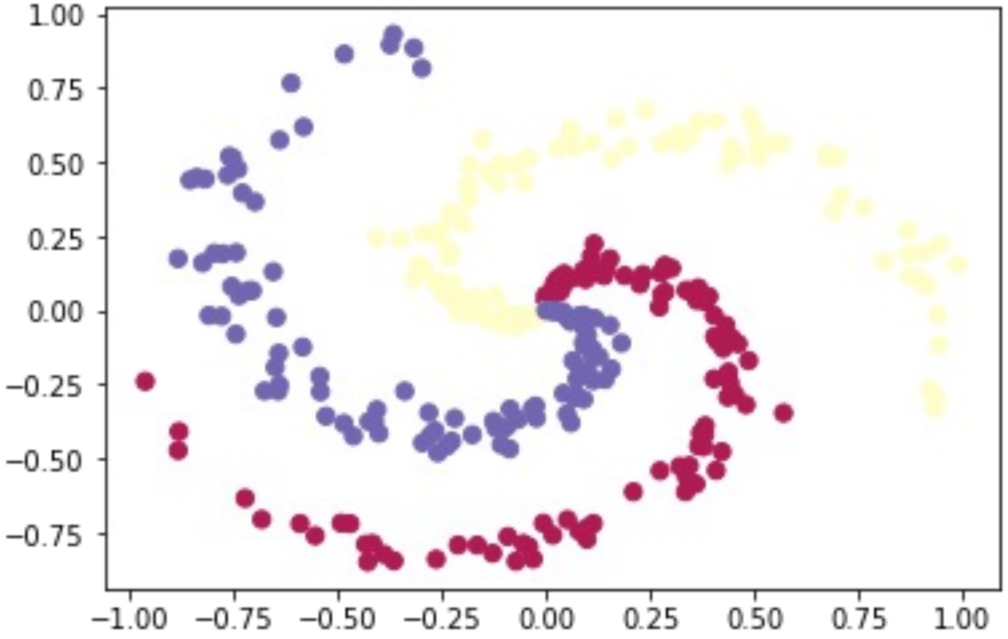
- What are the basic building blocks of neural network?
- Can you name two different neural network architectures?
- For an intermediate layer with width r (neurons), how many trainable parameters are associated with this layer?
- For a digit with label y and a softmax output vector from the neural network $u = [u_0, u_1, \dots, u_9]$, what is the cross-entropy loss of a neural network f for this input?

Lecture plan

- **Computing the gradient using numpy**

Review: Using neural networks for regression and classification

- Neural networks can be used to solve regression and classification problems
  - A toy data setting for training a neural network in PyTorch
  - We will use a linear classifier, then a nonlinear classifier, and compare their results



Review: Generating data

- Generate a two-dimensional dataset with nonlinear decision boundaries

generating some data

In [2]:
```
N = 100 # number of points per class
D = 2 # dimensionality
K = 3 # number of classes
X = np.zeros((N*K,D)) # data matrix (each row = single example)
y = np.zeros(N*K, dtype='uint8') # class labels

for j in range(K):
  ix = range(N*j,N*(j+1))
  r = np.linspace(0.0,1,N) # radius
  t = np.linspace(j*4,(j+1)*4,N) + np.random.randn(N)*0.2 # theta
  X[ix] = np.c_[r*np.sin(t), r*np.cos(t)]
  y[ix] = j

# lets visualize the data:
plt.scatter(X[:, 0], X[:, 1], c=y, s=40, cmap=plt.cm.Spectral)
plt.show()
```

Review: Initialization

- **Initialization:** Every entry of W (classifier parameters) is drawn from a standard Gaussian with mean zero and variance one
  - D: input dimension, K: number of classes

Initialize the parameters

In [3]:
```
# initialize parameters randomly
W = 0.01 * np.random.randn(D,K)
b = np.zeros((1,K))

step_size = 1e-0
reg = 1e-3
```

Review: Matrix multiplication

- X: dimension 300×2
- W: dimension 2×3
- b: dimension 1×3

Compute the output

In [4]:
```
# compute class scores for a linear classifier
scores = X @ W + b
```

Loss function

- **Training loss:** Averaged cross-entropy loss plus an ℓ_2 penalty

- **Averaged cross-entropy loss** (average over training dataset)
  - Given a prediction for every label $y \in \{1,2,\ldots,K\}$, let u be this vector
  - $\ell(u) = -\log \frac{\exp(u_y)}{\sum_{i=1}^{K} \exp(u_i)}$ (Fact: $\ell(u) \geq 0$)
- **ℓ_2 penalty:** Sum of squared values of W and b

- **Final loss function:**

$$\frac{1}{n}\sum_{i}^{n} \ell(f(x_i), y_i) + \frac{\alpha}{2}(\|W\|_F^2 + \|b\|^2)$$

Compute in numpy

```
num_examples = X.shape[0]
# get unnormalized probabilities
exp_scores = np.exp(scores)
# normalize them for each example
probs = exp_scores / np.sum(exp_scores, axis=1, keepdims=True)

correct_logprobs = -np.log(probs[range(num_examples),y])
```

```
reg_loss = 0.5*reg*np.sum(W*W)
loss = data_loss + reg_loss
```

Compute gradient in numpy

- Output for label k: u_k, cross-entropy loss $\ell(W, b)$
  - **Chain rule:** $\frac{\partial \ell(W,b)}{\partial W} = \sum_{i=1}^{n} \sum_k \frac{\partial \ell}{\partial u_k} \frac{\partial u_k}{\partial W}$
  - **Claims** (softmax probability for label k: p_k)

$$\frac{\partial \ell}{\partial u_k} = p_k - 1_{y=k}$$

$$\frac{\partial u}{\partial W} = X^\top$$

Compute the analytic gradient

In [8]:
```
dscores = probs
dscores[range(num_examples),y] -= 1
dscores /= num_examples

dW = X.T @ dscores
db = np.sum(dscores, axis=0, keepdims=True)
dW += reg*W # don't forget the regularization gradient
```

Gradient on bias adds up all the log probabilities

Explaining weight decay

- Gradient of ℓ_2 penalty

$$\frac{\alpha}{2}\nabla||W||_F^2 = \alpha W$$

- Weight decay with learning rate η

$$W - \eta \cdot \alpha W = (1 - \eta\alpha)W$$

Compute the analytic gradient

In [8]:
```
dscores = probs
dscores[range(num_examples),y] -= 1
dscores /= num_examples

dW = X.T @ dscores
db = np.sum(dscores, axis=0, keepdims=True)
dW += reg*W # don't forget the regularization gradient
```

Training loss

```
iteration 10: loss 0.9134056496088602
iteration 20: loss 0.8323889971607258
iteration 30: loss 0.7955967913635283
iteration 40: loss 0.7762634535759677
iteration 50: loss 0.7651042787584552
iteration 60: loss 0.7582423095449976
iteration 70: loss 0.7538293272190891
iteration 80: loss 0.7508959335854734
iteration 90: loss 0.7488963644108956
iteration 100: loss 0.7475063136555101
iteration 110: loss 0.7465247676838905
iteration 120: loss 0.7458228704214372
iteration 130: loss 0.7453157377782931
iteration 140: loss 0.7449461859000616
iteration 150: loss 0.7446749691022985
iteration 160: loss 0.744474730614621
iteration 170: loss 0.7443261494995304
iteration 180: loss 0.7442154278913563
iteration 190: loss 0.7441326186704039
iteration 200: loss 0.7440704927051738
```

This is quite high for three classes:

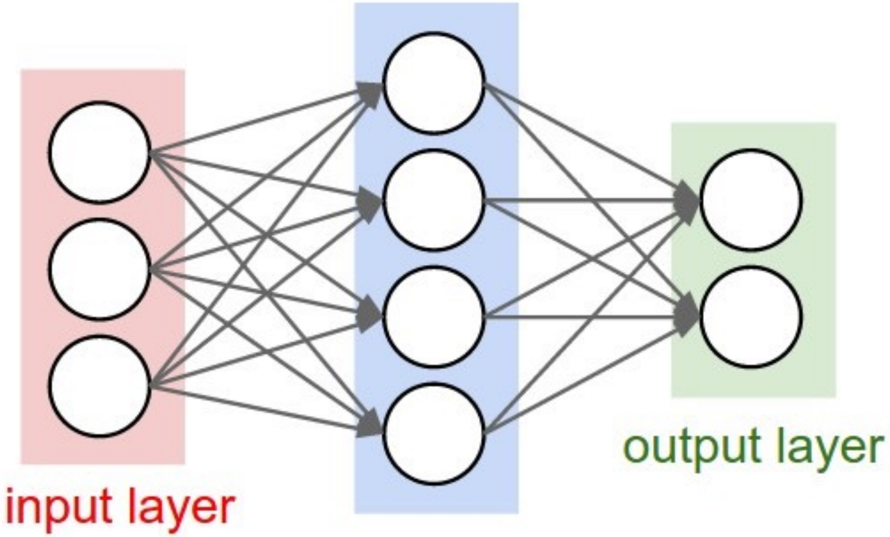
$$-\log\frac{1}{3} = 1.10$$

Use nonlinear classifiers

- First trainable layer: weight matrix $W_1 \in \mathbb{R}^{D \times h}$, bias $b_1 \in \mathbb{R}^h$
- Activation function
- A second trainable layer: weight matrix $W_2 \in \mathbb{R}^{h \times K}$, bias $b_2 \in \mathbb{R}^K$

Initialize the parameters

In [3]:
```
# initialize parameters randomly
h = 100 # size of hidden layer
W = 0.01 * np.random.randn(D,h)
b = np.zeros((1,h))
W2 = 0.01 * np.random.randn(h,K)
b2 = np.zeros((1,K))

step_size = 1e-0
reg = 1e-3
```



Forward pass in the two-layer neural network

- Rectified linear units (ReLU): $\sigma(z) = \max(z, 0)$

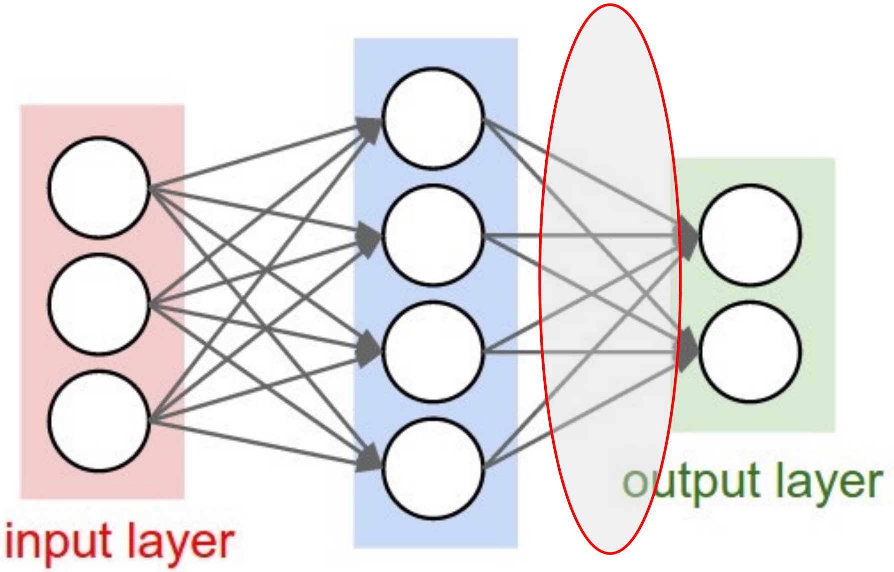
$$u = \sigma(XW_1 + 1 \cdot b_1)W_2 + 1 \cdot b_2$$

Compute the output

```
In [4]: # evaluate class scores with a 2-layer Neural Network
        hidden_layer = np.maximum(0, np.dot(X, W) + b) # note, ReLU activation
        scores = hidden_layer @ W2 + b2
```

Gradient of the second layer

- **Gradient of the second layer:** Similar to the linear case
  - Treat the hidden layer output as input

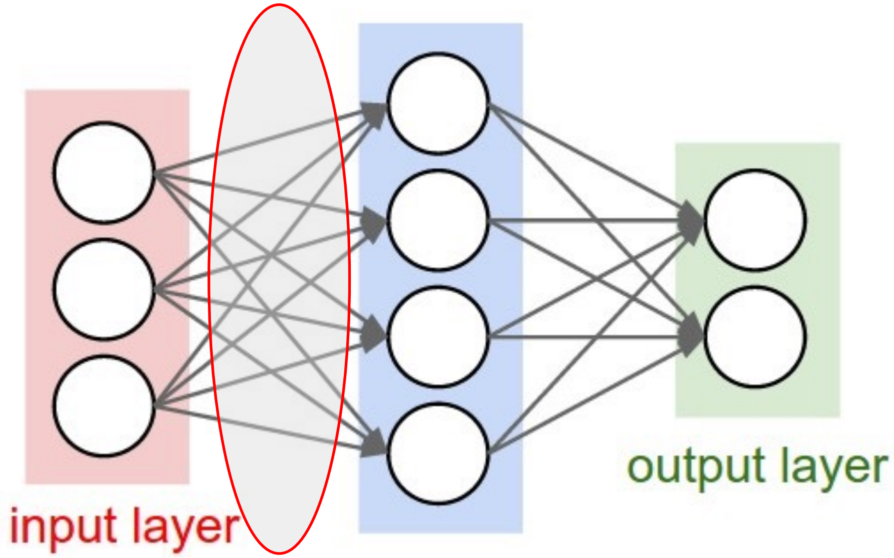


Compute the analytic gradient

In [6]:
```
# backpropate the gradient to the parameters
dscores = probs
dscores[range(num_examples),y] -= 1
dscores /= num_examples

# first backprop into parameters W2 and b2
dW2 = hidden_layer.T @ dscores
db2 = np.sum(dscores, axis=0, keepdims=True)

dhidden = dscores @ W2.T

# backprop the ReLU non-linearity
dhidden[hidden_layer <= 0] = 0

# finally into W,b
dW = X.T @ dhidden
db = np.sum(dhidden, axis=0, keepdims=True)
```

Gradient of the first layer

- **Gradient of the first layer: Use chain rule**



Compute the analytic gradient

In [6]:
```
# backpropate the gradient to the parameters
dscores = probs
dscores[range(num_examples),y] -= 1
dscores /= num_examples

# first backprop into parameters W2 and b2
dW2 = hidden_layer.T @ dscores
db2 = np.sum(dscores, axis=0, keepdims=True)

dhidden = dscores @ W2.T

# backprop the ReLU non-linearity
dhidden[hidden_layer <= 0] = 0

# finally into W,b
dW = X.T @ dhidden
db = np.sum(dhidden, axis=0, keepdims=True)
```

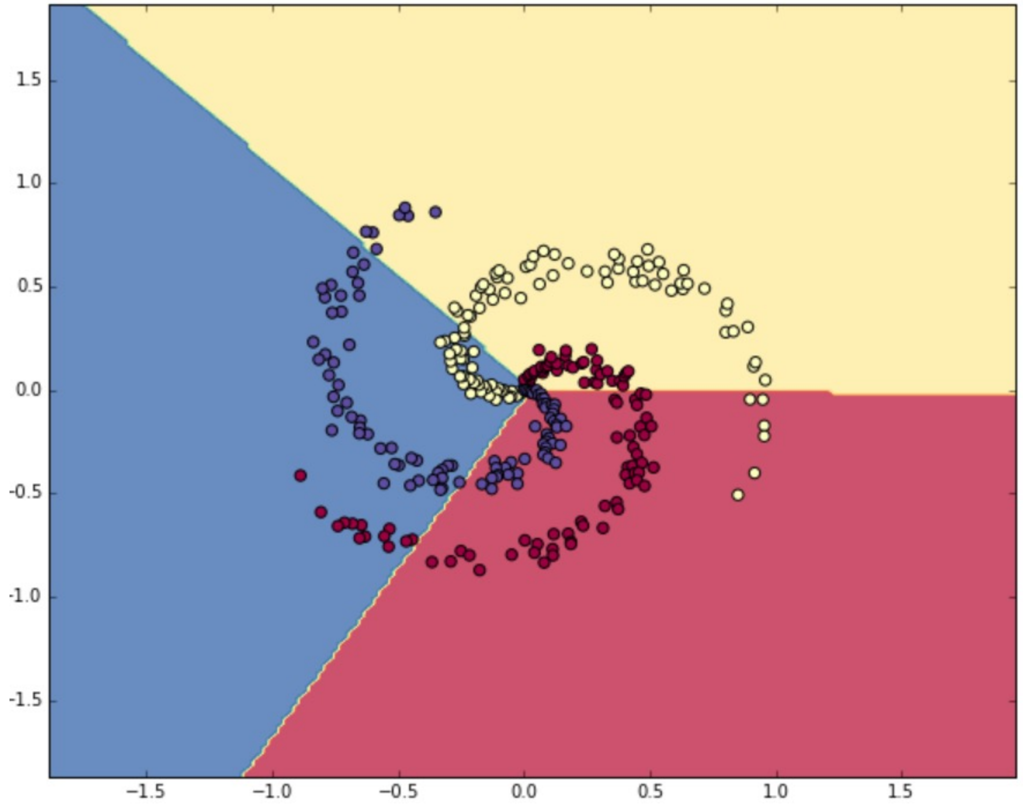
Gradient of the first layer

- Let $h = \sigma(XW_1 + 1 \cdot b_1)$
  - $\frac{\partial \ell}{\partial h} = \frac{\partial \ell}{\partial u} \cdot \frac{\partial u}{\partial h}$
  - Recall $u = \sigma(XW_1 + 1 \cdot b_1)W_2 + 1 \cdot b_2 = hW_2 + 1 \cdot b_2$
- Use chain rule to get $\frac{\partial \ell}{\partial W_1} = \frac{\partial \ell}{\partial h} \cdot \frac{\partial h}{\partial W_1}$
  - $\frac{\partial h}{\partial W_1}$: get the derivative of the activation, then the derivative h of W_1
  - $db_1 = \frac{\partial \ell}{\partial b_1}$ is similar

Compute the analytic gradient

```
In [6]: # backpropate the gradient to the parameters
        dscores = probs
        dscores[range(num_examples),y] -= 1
        dscores /= num_examples

        # first backprop into parameters W2 and b2
        dW2 = hidden_layer.T @ dscores
        db2 = np.sum(dscores, axis=0, keepdims=True)

        dhidden = dscores @ W2.T

        # backprop the ReLU non-linearity
        dhidden[hidden_layer <= 0] = 0

        # finally into W,b
        dW = X.T @ dhidden
        db = np.sum(dhidden, axis=0, keepdims=True)
```

Results




```
iteration 1000: loss 0.40454021503681153
iteration 2000: loss 0.26346369806692593
iteration 3000: loss 0.25607811374045586
iteration 4000: loss 0.25410664245334263
iteration 5000: loss 0.2526010149171124
iteration 6000: loss 0.25198089929407874
iteration 7000: loss 0.25155952434511186
iteration 8000: loss 0.2512825150552082
iteration 9000: loss 0.251104422840225
iteration 10000: loss 0.2509892383094693
```

Lecture plan

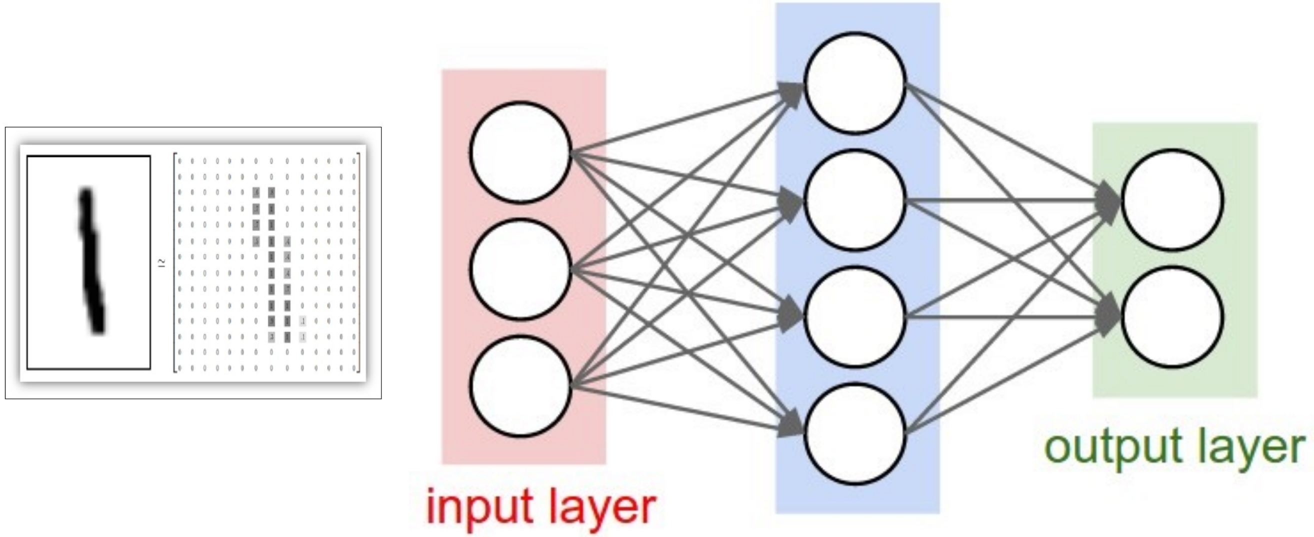
- **How backpropagation works**

Overview

- Backpropagation algorithm is the workhorse of modern deep networks
- Both PyTorch and TensorFlow implement the backpropagation

Training loss objective

- Train parameters W_1, b_1, W_2, b_2 to minimize the cross-entropy loss
- Minimize the cross-entropy loss as the training objective



Prediction over {0,**1**,2,3,4,5,6,7,8,9}

Softmax output [0.01, **0.9**, 0.01, 0.01, 0.01, 0.01, 0.01, 0.02, 0.01, 0.01]

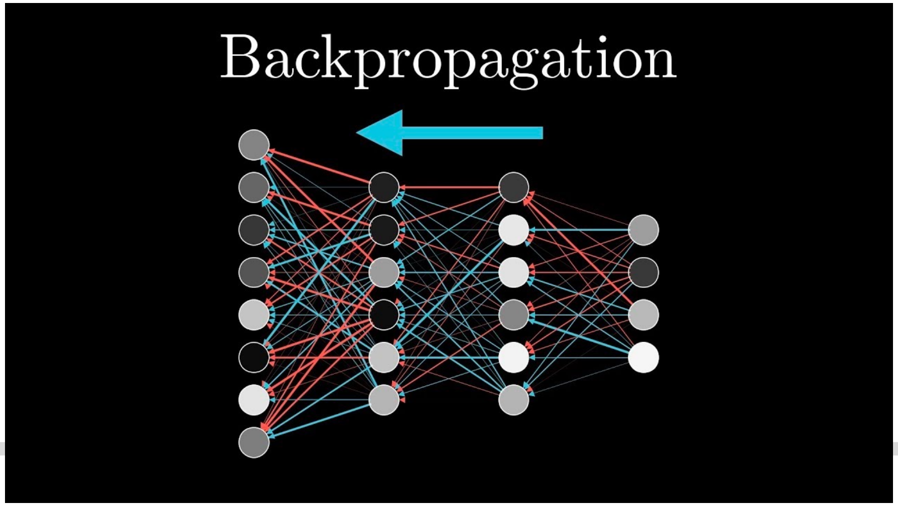
Loss value: $-\log\frac{0.9}{1} = 0.046$

The gradient of each layer

- Suppose x is a data point with label y: Let $\ell(x, y)$ be the loss of
- The output of the backpropagation algorithm will be
  - Gradient of ℓ with respect to W_1, b_1 (layer 1)
  - Gradient of ℓ with respect to W_2, b_2 (layer 2)
  - …
  - Gradient of ℓ with respect to W_L, b_L (layer L)

How backpropagation works

- Backpropagation consists of two steps
  - **Step 1: Use forward pass** to compute the input to every layer and the output of every layer
  - **Step 2: Use backward pass** to compute the gradient
  - In total, we need to run two passes over the entire neural network to conduct this computation!



Forward pass

- Input: $o_0 = x$
- For $i = 1, 2, \ldots, L$
  - Input to layer i: $z_i = o_{i-1}W_i + b_i$
  - Output of layer i: $o_i = \sigma_i(z_i)$
- Return o_L

- Important takeaway
  - Input to layer i: z_i
  - Output of layer i: o_i

The backward pass

- Setup
  - Loss function ℓ
  - i-th trainable layer: weight matrix $W_i \in \mathbb{R}^{d_{i-1} \times d_i}$, bias $b_i \in \mathbb{R}^{d_i}$
  - Activation function: $\sigma_i: \mathbb{R} \to \mathbb{R}$
- **Output:** $\frac{\partial \ell}{\partial W_i}$ and $\frac{\partial \ell}{\partial b_i}$ for all $i = 1, 2, \dots, L$

Simplified example: One dimension

- A two-layer linear network with mean squared loss

$$\ell(x, y) = (w_2 w_1 x - y)^2$$

- **Claims**

$$\frac{\partial \ell}{\partial w_2} = 2(w_2 w_1 x - y) w_1 x$$

$$\frac{\partial \ell}{\partial w_1} = 2(w_2 w_1 x - y) w_2 x$$

With nonlinear activation

- **Nonlinear activation**

$$\ell(x, y) = (w_2 \sigma_1(w_1 x) - y)^2$$

- **Claims**

$$\frac{\partial \ell}{\partial w_2} = 2(w_2 \sigma_1(w_1 x) - y)\sigma_1(w_1 x)$$

$$\frac{\partial \ell}{\partial w_1} = 2(w_2 \sigma(w_1 x) - y) w_2 \sigma_1'(w_1 x) x$$

- Compare with the previous example, we have an additional term which is $\sigma_1{}'(w_1 x)$

Multi-layer linear network

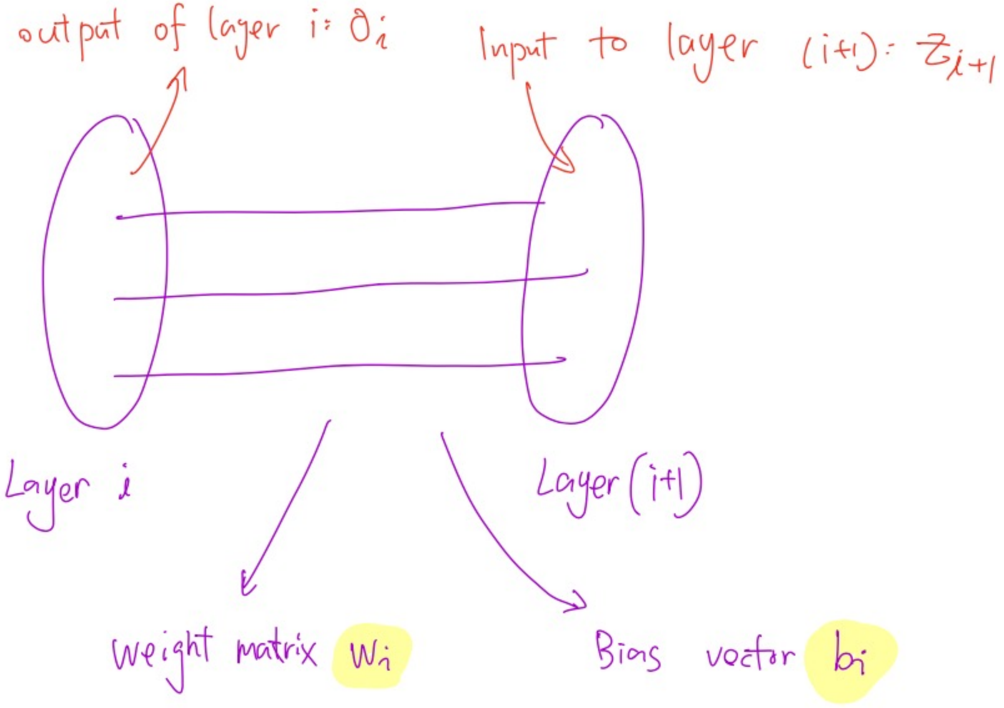
- A multi-layer linear network with squared loss

$$\ell(x, y) = (w_L w_{L-1} \dots w_1 x - y)^2$$

- **Claims**
  - $\frac{\partial \ell}{\partial w_L} = 2(w_L w_{L-1} \dots w_1 x - y) w_{L-1} \dots w_1 x$
  - $\frac{\partial \ell}{\partial w_{L-1}} = 2(w_L w_{L-1} \dots w_1 x - y) w_L w_{L-2} \dots w_1 x$
  - …
  - $\frac{\partial \ell}{\partial w_1} = 2(w_L w_{L-1} \dots w_1 x - y) w_L w_{L-1} \dots w_2 x$

Looking at an intermediate layer

- Illustration



Loss function : ℓ

Want : $\frac{\partial \ell}{\partial W_i}$ and $\frac{\partial \ell}{\partial b_i}$

Multi-layer linear network (simplified)

- Input to layer i: $z_i = w_{i-1} w_{i-2} \dots w_1 x$
- Output of layer i: $o_i = w_i w_{i-1} \dots w_1 x$ (assume bias at layer i is equal to zero)
- Loss ℓ: squared loss

$$\frac{\partial \ell}{\partial w_L} = 2(w_L w_{L-1} \dots w_1 x - y) w_{L-1} \dots w_1 x = 2(z_L w_L - y) z_L$$

  - $\ell(x, y) = (z_L w_L - y)^2$
  - Thus, $\frac{\partial \ell}{\partial w_L} = 2(z_L w_L - y) z_L$, and $\frac{\partial \ell}{\partial z_L} = 2(z_L w_L - y) w_L$
  - What about $\frac{\partial \ell}{\partial w_{L-1}}$?
  - Notice that $z_L = w_{L-1} o_{L-1}$: $\frac{\partial \ell}{\partial w_{L-1}} = \frac{\partial \ell}{\partial z_L} \cdot \frac{\partial z_L}{\partial w_{L-1}} = \frac{\partial \ell}{\partial z_L} \cdot o_{L-1}$

Multi-layer linear network (simplified)

- For any i

$$\frac{\partial \ell}{\partial w_i} = \frac{\partial \ell}{\partial z_{i+1}} \cdot \frac{\partial z_{i+1}}{\partial w_i} = \frac{\partial \ell}{\partial z_{i+1}} \cdot o_i$$

  - Input to layer $i+1$: $z_{i+1} = w_i o_i$; $\frac{\partial z_{i+1}}{\partial w_i} = o_i$

$$\frac{\partial \ell}{\partial z_{i+1}} = \frac{\partial \ell}{\partial z_i} \cdot \frac{\partial z_{i+1}}{\partial z_i} = \frac{\partial \ell}{\partial z_i} \cdot w_i$$

  - Notice that $z_{i+1} = w_i z_i$, recall activation is linear

  - Thus, $\frac{\partial z_{i+1}}{\partial z_i} = w_i$

With nonlinear activation

$$z_{i+1} = w_i o_i = w_i \sigma(z_i)$$

- In this case, we instead have $\frac{\partial z_{i+1}}{\partial z_i} = w_i \sigma'(z_i)$
- The rest of the calculation remains the same
- The other caveat is that in this example, we focused on one-dimensional input. For multi-dimensional input, the idea is the same, although the computation is hairier

Summary

- To wrap up, we have shown how to derive backward pass for a multi-layer, nonlinear neural network with mean squared loss
- For cross-entropy loss, the steps are the same except that the gradient of the loss is more complicated

- **Key idea:**
  - Store intermediate input, output at each layer
  - Use chain rule to backprop the gradient all the way from the output layer to the input layer

Summary: The backward pass

- Write $\frac{\partial \ell}{\partial w_i}$ and $\frac{\partial \ell}{\partial b_i}$ based on $\frac{\partial \ell}{\partial w_{i+1}}$ and $\frac{\partial \ell}{\partial b_{i+1}}$
  - Decompose the gradient at this layer back to the gradient of the previous layer

- Find the gradient at every layer by going backward from the final output layer
  - Find out $\frac{\partial \ell}{\partial w_L}$ and $\frac{\partial \ell}{\partial b_L}$
  - Find out $\frac{\partial \ell}{\partial w_{L-1}}$ and $\frac{\partial \ell}{\partial b_{L-1}}$
  - …
  - Find out $\frac{\partial \ell}{\partial w_1}$ and $\frac{\partial \ell}{\partial b_1}$

Announcements

- Project document guideline: https://docs.google.com/document/d/1EmhNv4yWqkrABGb_BMmwvphdPk9LI61mgHsmzO2T1Lk/edit?usp=sharing

- Khoury MS apprenticeship nominations: at most five nominations (first come first serve), two spots remaining