Multi-Class Classification
Solution

• Traditional Method: 1-vs-other method
  • Too slow. If we have $n$-classes, we need to train $n$ models
  • Performance is not great, because the sample size is different for positive and negative classes

• Multiple Neurons
  • Use $n$ output neuron to correspond $n$ classes.
  • Easy, fast, and robust
  • Problem: how to model the probability? The values in the neural network can be negative or greater than 1.
Softmax: normalized exponential

\[ \sigma(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \]

Input: vector of reals

Output: probability distribution

softmax([1,2,7,3,2]):

Calculate \( e^x \): [2.72, 7.39, 1096.63, 20.09, 7.39]

Calculate sum(\( e^x \)): 2.72+7.39+1096.63+20.09+7.39 = 1134.22

Normalize: \( e^x / \text{sum}(e^x) \) = [0.002, 0.007, 0.967, 0.017, 0.007]

Result is a vector of reals.
A Simple Example

Here, we will go over a simple 2-layer neural network (no bias).
Mini-batch for Machine Learning

• We use a matrix to represent data.

• When s is very large $\rightarrow$ RAM overload
Mini-batch for Machine Learning

Training Data

\[ s_1, s_2, s_3, s_4, \ldots, s_{n-1}, s_n \]

Iteration 1

Iteration 2

Iteration n/2

Compute Gradients

Update Weights
Neural Network Easy Example

First pixel
- 3

Second pixel
- 2

Third pixel
- 4

Input
- $X_{in}$

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Bsz = 4

First Batch
Neural Network Easy Example

First pixel: 3
Second pixel: 2
Third pixel: 4

Input:

\[
\begin{array}{c}
X_{in} \\
| 3 | 2 | 4 | \\
|-------------------|---|---|---|
| . | . | . | \\
| . | . | . | \\
| . | . | . | \\
\end{array}
\]

1-st Layer (ReLU):

\[
\begin{array}{c|c}
| 1 | 0.5 | \\
| 0.1 | 1 | \\
| -2.3 | -0.5 | \\
\end{array}
\]

\[
\begin{align*}
A & \quad 1 \\
B & \quad 0.5 \\
\end{align*}
\]

Weights:

\[
w_1
\]

Activation Function: ReLU

Output:

\[
A \quad B
\]

Diagram:

- First pixel (3) connected to A with weights 1, 0.1, -2.3
- Second pixel (2) connected to A with weights 0.5
- Third pixel (4) connected to A with weights 1, -0.5
- A connected to B with weight 0.5
- B connected to A with weight 1
Neural Network Easy Example

<table>
<thead>
<tr>
<th>First pixel</th>
<th>Second pixel</th>
<th>Third pixel</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>

Input:

<table>
<thead>
<tr>
<th>3</th>
<th>2</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
</tbody>
</table>

1-st Layer (ReLU):

<table>
<thead>
<tr>
<th>1</th>
<th>0.1</th>
<th>0.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>-2.3</td>
<td>1</td>
<td>-0.5</td>
</tr>
</tbody>
</table>

Output Layer with Softmax:

<table>
<thead>
<tr>
<th>0.7</th>
<th>-2.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>-0.2</td>
</tr>
</tbody>
</table>

$w_1$ and $w_2$ are the weight matrices connecting the input layer to the 1-st layer and the 1-st layer to the output layer, respectively.
1-st Layer with ReLU

Output\textsubscript{A} = 3 \times 1 + 2 \times 0.1 + 4 \times -2.3 = -6.0

Output\textsubscript{A} = 0 \text{(after RELU)}
Neural Network Easy Example

First pixel

Second pixel

Third pixel

$A = 0$

$B = 3 \times 0.5 + 2 \times 1 + 4 \times -0.5 = 1.5$
Neural Network Easy Example

\[ \text{output}_C = \frac{e^{0.15}}{e^{0.15} + e^{-0.3}} \approx \frac{1.16}{1.16 + 0.74} = 0.61 \]

\[ \text{output}_D = \frac{e^{-0.3}}{e^{0.15} + e^{-0.3}} \approx \frac{0.74}{1.16 + 0.74} = 0.39 \]
Neural Network Easy Example

Input

3
2
4

1-st Layer (ReLU)

A
B

Output Layer with Softmax

0.1
0.7
0.1
-2.3
0.5
1
0.1
-0.5
-0.5
0.61
0.39

0.15
-0.3
-2.1
-0.2

1
1
0.7
-6.0
-1.5
0.61
0.39

C
D

o2
Neural Network Easy Example

First pixel

Second pixel

Third pixel

Input

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1-st Layer (ReLU)

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.1</td>
<td>0.5</td>
</tr>
<tr>
<td>0.1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>-2.3</td>
<td>-0.5</td>
<td></td>
</tr>
</tbody>
</table>

Output Layer with Softmax

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0.7</td>
<td>-2.1</td>
</tr>
<tr>
<td>0.1</td>
<td>-0.2</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Output

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0.61</td>
<td>0.39</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Neural Network Easy Example

First pixel
3

Second pixel
2

Third pixel
4

Input
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1-st Layer (ReLU)

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>0.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.1</td>
<td>-2.3</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>-0.5</td>
</tr>
</tbody>
</table>

Output Layer with Softmax

<table>
<thead>
<tr>
<th></th>
<th>0.7</th>
<th>-2.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.1</td>
<td>-0.2</td>
</tr>
</tbody>
</table>

Ground Truth

\[ \Delta \]
Neural Network Easy Example

\[ \Delta = 1 - 0.61 = 0.39 \]

\[ \Delta = 0 - 0.39 = -0.39 \]
Neural Network Easy Example

$$\Delta w_t = -\frac{\partial E}{\partial w_t} = o_1^T g'(o_2) \circ \Delta o_2$$

$$w_{t+1} = w_t + \alpha \Delta w_t$$

Assume $g'(.) = 1$

“$\circ$” represents elementwise multiplication for matrix
Neural Network Easy Example

\[
g'(o_2) \circ \Delta o_2
\]

\[
\begin{array}{cc}
0.39 & -0.39 \\
0.585 & -0.585
\end{array}
\]

\[
\Delta w_2 = o_1^T g'(o_2) \circ \Delta o_2
\]

\[
\begin{array}{cc}
0 & 0 \\
0.585 & -0.585
\end{array}
\]

\[
\Delta o_1 = g'(o_2) \circ \Delta o_2 w_2^T
\]

\[
\begin{array}{cc}
1.092 & 0.117 \\
0.585 & -0.585
\end{array}
\]

“\(\circ\)” represents elementwise multiplication for matrix

\[
\Delta w_t = -\frac{\partial E}{\partial w_t} = o_1^T g'(o_2) \circ \Delta o_2
\]

\[
w_{t+1} = w_t + \alpha \Delta w_t
\]

Assume \(g'(.) = 1\)
Backpropagation [Cont.]

\[
\Delta w_1 = o_0^T g'(o_1) \cdot \Delta o_1
\]

\[
\Delta o_0 = g'(o_1) \cdot \Delta o_1 w_1^T
\]

\[
\Delta w_2 = o_1^T g'(o_2) \cdot \Delta o_2
\]

\[
\Delta o_1 = g'(o_2) \cdot \Delta o_2 w_2^T
\]
Think: What will happen if we go forward again?

Input Layer

<table>
<thead>
<tr>
<th>3</th>
<th>2</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
</tbody>
</table>

1-st Layer (ReLU)

<table>
<thead>
<tr>
<th>1</th>
<th>0.1</th>
<th>-2.3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.5351</td>
<td>1.0234</td>
<td>-0.4532</td>
</tr>
</tbody>
</table>

Output Layer with Softmax

<table>
<thead>
<tr>
<th>0.7</th>
<th>-2.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1585</td>
<td>-0.2585</td>
</tr>
</tbody>
</table>
Think: What will happen if we go forward again?

The final output is closer to the actual label.
Tricks for Neural Network
Problem: Under and Overfitting

Underfitting: model not powerful enough, too much bias

Overfitting: model too powerful, fits to noise, doesn’t generalize well

Want the happy medium, how?
**Weight decay**: neural network regularization

We want the weights to be close to 0.

Let $L$ be the “loss” function; (e.g. $L = |y - g(in)|$, $L = (y - g(in))^2$, etc.)

$\lambda$ is a regularization parameter (for decay)
- Higher: more penalty for large weights, less powerful model
- Lower: less penalty, more overfitting

Before:
$$
\Delta w_t = -\frac{\partial}{\partial w_t} L(w_t)
$$
$$
w_{t+1} = w_t + \alpha \Delta w_t
$$

Now:
$$
w_{t+1} = w_t - \alpha [\frac{\partial}{\partial w_t} L(w_t) + \lambda w_t] = w_t - \alpha [-\Delta w_t + \lambda w_t]
$$
$$
= w_t - \alpha \frac{\partial}{\partial w_t} L(w_t) - \alpha \lambda w_t = w_t + \alpha \Delta w_t - \alpha \lambda w_t
$$

We use $\Delta w_t$ to represent the weight gradient for timepoint $t$ (the current step). Subtract a little bit of weight every iteration.
Momentum: speeding up SGD

If we keep moving in same direction we should move further every round

Before:
\[ \Delta w_t = -\frac{\partial}{\partial w} L(w_t) \]

Now:
\[ \Delta w_t = -\frac{\partial}{\partial w} L(w_t) + m\Delta w_{t-1} \]

\[ w_{t+1} = w_t + \alpha \Delta w_t \]

Side effect: smooth out updates if gradient is in different directions

\[ \Delta w_{t-1} \] represent the gradient calculated in the previous step.
NN updates with weight decay and momentum

\[ \Delta w'_t = -\frac{\partial}{\partial w_t} L(w_t) - \lambda w_t + m\Delta w'_{t-1} \]

\[ w_{t+1} = w_t + \alpha \Delta w'_t \]
Activations
Linear Activation

\[ g(x) = x \]
\[ g'(x) = 1 \]

- Only offers linear effects.
- For a 2-layer NN with linear activations for both layers.

\[ f(X) = g(g(Xw_1)w_2) = Xw_1w_2 = Xw \]

- Not so great, need Non-Linear activations to learn more complex data distribution.
Logistic Activation

\[ g(x) = \frac{1}{1 + e^{-x}} \]

\[ g'(x) = g(x)g(1 - x) \]

- Aka Sigmoid function (S-shape)
- Used in Logistic regression.
- The result is in range (0, 1),
- It can represent probability.
- A special case of logistic growth (population model).
ReLU Activation

\[ g(x) = \max(0, x) \]

\[ g'(x) = 1_{g(x) > 0} \]

- Rectified linear unit
- **Fast!** In backpropagation, 1 when positive, 0 otherwise.
- Optimizes important (positive) values and ignore the others.
- Analog to neurons
- Information loss is small (other neurons will carry information)
Visualization with ReLU

https://www.youtube.com/channel/UCYO_jab_esuFRV4b17AJtAw
The Dying ReLU Problem

\[ f'(x) = g(x) = 1, \quad x \geq 0 \]
\[ = 0, \quad x < 0 \]
LeakyReLU Activation

- No information loss (compared to ReLU)
- Solves “dying ReLU” problem (i.e. all neurons output 0)
- Similar to ReLU, pays less attention to less important neurons
- Not always better than ReLU
Homework 4
Neural Network
MNIST: Handwriting recognition

50,000 images of handwriting
Input: 28 x 28 x 1 (grayscale) $\rightarrow$ 784 pixel values
Numbers 0-9 $\rightarrow$ 10 classes

Train a linear softmax model
> 95% accuracy
Functions You need to Code

void activate_matrix(matrix m, ACTIVATION a)
void gradient_matrix(matrix m, ACTIVATION a, matrix d)
matrix forward_layer(layer *l, matrix in)
matrix backward_layer(layer *l, matrix delta)
void update_layer(layer *l, double rate, double momentum, double decay)

Run Experiments and Write a Report (hw4.pdf)

Play around with tryhw4.py file, and answer the questions.

Save your question to a PDF file and submit to Canvas for grading.
typedef enum {LINEAR, LOGISTIC, RELU, LRELU, SOFTMAX} ACTIVATION;

typedef struct {
    matrix in;       // Saved input to a layer
    matrix w;       // Current weights for a layer
    matrix dw;      // Current weight updates
    matrix v;       // Past weight updates (for use with momentum)
    matrix out;     // Saved output from the layer
    ACTIVATION activation; // Activation the layer uses
} layer;

typedef struct {
    layer *layers;
    int n;
} model;
Useful Matrix manipulation functions (matrix.c)

```c
matrix matrix_mult_matrix(matrix a, matrix b);
matrix transpose_matrix(matrix m);
matrix axpy_matrix(double a, matrix x, matrix y); // a * x + y
```
Forward Pass in Homework

**forward_model**

Input: model m, data X

**forward_layer**

Output

**forward_layer**

Input: layer l, data in

\[
X = \text{in} \times l \rightarrow w
\]

**activation_matrix**

(X, l \rightarrow \text{activation})

Output
Backward Pass in Homework

**backward_model**

Input: model m, matrix d

Output

**backward_layer**

Input: layer l, matrix delta

gradient_matrix

Calculate Δw

Calculate Δo

Δo
Weight Update in Homework

\[ \Delta w'_{t-1} = \Delta w - \lambda w + m \Delta w'_{t-1} \]

Input: layer \( l \), learning rate \( \alpha \), decay \( \lambda \), momentum \( m \)

Output \( w = w + \alpha \Delta w' \)

\[ \Delta w' = \Delta w - \lambda w + m \Delta w'_{t-1} \]

Input: model \( m \), learning rate \( \alpha \), decay \( \lambda \), momentum \( m \)

Output
void activate_matrix(matrix m, ACTIVATION a)

for(i = 0; i < m.rows; ++i){
    double sum = 0;
    for(j = 0; j < m.cols; ++j){
        double x = m.data[i][j];
        if(a == LOGISTIC){
            // TODO m.data[i][j] should equals 1 / (1 + exp(-x));
        } else if (a == RELU){
            // TODO m.data[i][j] should equals x if x > 0; otherwise, it should equal 0
        } else if (a == LRELU){
            // TODO m.data[i][j] should equals x if x > 0; otherwise, it should equal 0.1 * x.
        } else if (a == SOFTMAX){
            // TODO m.data[i][j] should equals exp(x) here, and we will normalize it later.
        }
        sum += m.data[i][j];
    }
    if (a == SOFTMAX) {
        // TODO: have to normalize by sum if we are using SOFTMAX
        // for all the possible j, we should normalize it as m.data[i][j] /= sum;
    }
}
TODO void gradient_matrix(matrix m, ACTIVATION a, matrix d)

int i, j;
for(i = 0; i < m.rows; ++i){
    for(j = 0; j < m.cols; ++j){
        double x = m.data[i][j];
        // TODO: multiply the correct element of d by the gradient
        // if a is SOFTMAX or a is LINEAR, we should do nothing (multiply by 1)
        // if a is LOGISTIC, d.data[i][j] should times x * (1.0 - x);
        // if a is RELU and x <= 0, d.data[i][j] should be zero
        // if a is LRELU and x <= 0, d.data[i][j] should multiple 0.1
    }
}

Calculate g’(m) * d, and store in-place to matrix d.
The matrix “m” is the output of a layer, and matrix “d” is the Δ of output.
matrix forward_layer(layer *l, matrix in)

Given the input data “in” and layer “l”, calculate the output data.

l->in = in; // Save the input for backpropagation

// TODO: multiply input by weights and apply activation function.
// Calculate out = in * l->w (note: matrix multiplication here)
// Then, apply activate_matrix function to out with l->activation

free_matrix(l->out); // free the old output
l->out = out; // Save the current output for gradient calculation
return out;
Given the layer “l” and delta, perform backward step:

1.4.1: Calculate the delta after considering the activation
1.4.2: Calculate Δw
1.4.3: Calculate and Return Δo (aka “dx”).

// delta is Δout
// TODO: modify it in place to be “g’(out) * delta” out with // gradient_matrix function.
// You can use gradient_matrix function with “l->out” and “l->activation” to “delta”

// TODO: then calculate dL/dw and save it in l->dw
free_matrix(l->dw);
// Calculate xt as the transpose matrix of “l->in”
// Calculate dw as xt times delta (matrix multiplication)
// free matrix xt to avoid memory leak
l->dw = dw;

// TODO: finally, calculate dL/dx and return it. (Similar to 1.4.2. Care memory leak)
// Calculate dx = delta * (l->w)^T,  where * is matrix multiplication and ^T is matrix transpose
return dx;
void update_layer(layer *l, double rate, double momentum, double decay)

// Calculate Δw_t = dL/dw_t - λw_t + mΔw_{t-1}
// save it to l->v
// Note that You can use axpy_matrix to perform the matrix summation/subtraction

// Update l->w
// l->w = rate * l->v + l->w

Note the multiplication and summation in this slides all mean matrix multiplication or matrix summation.
Functions You Need to Know before Experiments

For simplicity, we already filled the following functions for you. You should read and understand these functions (classifier.c) before running experiments.

```
layer make_layer(int input, int output, ACTIVATION activation)
matrix forward_model(model m, matrix X)
void backward_model(model m, matrix dL)
void update_model(model m, double rate, double momentum, double decay)
double accuracy_model(model m, data d)
double cross_entropy_loss(matrix y, matrix p)
void train_model(model m, data d, int batch, int iters, double rate, double momentum, double decay)
```
Get the Data

1. Download, Unzip, and Prepare the MNIST Dataset

   wget https://pjreddie.com/media/files/mnist_train.tar.gz
   tar xzf mnist_train.tar.gz
   tar xzf mnist_test.tar.gz
   find train -name \*.png > mnist.train
   find test -name \*.png > mnist.test

2. Download, Unzip, and Prepare the CIFAR-10 Dataset

   tar xzf cifar.tgz
   find cifar/train -name \*.png > cifar.train
   find cifar/test -name \*.png > cifar.test
Experiments (Write Your Answers to hw4.pdf)

1. Coding and Data prepare
2. MNIST Experiments
   1. Linear Softmax Model (1-layer)
      1. Run the basic model
      2. Tune the learning rate
      3. Tune the decay
   2. Neural Network (2-layer NNs and 3-layer NNs)
      1. Find the best activation
      2. Tune the learning rate
      3. Tune the decay
      4. Tune the decay for 3-layer Neural Network
3. Experiments for CIFAR-10
   1. Neural Network (3-layer NNs)
      1. Tune the learning rate and decay