

Machine Learning

Textbook: Machine Learning Refined, by Watt, Borhani, and Katsaggelos, 2nd edition:
MLRefined.com

Previous Lecture

- ♦ A K -class linear classifier assigns one linear score $f_k(\mathbf{x})$ to each class.
- ♦ Prediction uses the winner-take-all rule $\hat{y} = \arg \max_k f_k(\mathbf{x})$.
- ♦ The boundary between classes j and k satisfies $f_j = f_k$ and is therefore a hyperplane for linear scores.
- ♦ The multiclass margin is the true-class score minus the largest competing score.
- ♦ A multiclass perceptron update raises the true-class score and lowers the strongest incorrect score.
- ♦ Independent sigmoids do not generally produce class probabilities that sum to one, motivating softmax.



§7.5 - Softmax and Categorical Cross-Entropy

Last time:

a multiclass classifier produced

K

scores

$f_1, \dots, f_K,$

and predicted

$\hat{y} = \arg \max_k f_k.$

But:

those scores were not probabilities.

Today:

turn them into a valid probability distribution

and derive the corresponding loss.

What Multiclass Probabilities Must Satisfy

For

K

mutually exclusive classes,

we want probabilities

$p_1, \dots, p_K$

such that

$p_k > 0$

for every class,

and

$$\sum_{k=1}^K p_k = 1.$$

We also want:

larger score

$\Rightarrow$

larger class probability.

The softmax transformation does all three.

The Softmax Function

Given scores

$$f_1, \dots, f_K,$$

define

$$p_k = \frac{e^{f_k}}{\sum_{j=1}^K e^{f_j}}.$$

This is the **softmax** transformation.

Each exponential is positive.

Therefore

$$p_k > 0.$$

Also:

$$\begin{aligned} & \sum_{k=1}^K p_k \\ &= \frac{\sum_k e^{f_k}}{\sum_j e^{f_j}} \\ &= 1. \end{aligned}$$

So softmax produces a valid categorical probability distribution.

A Three-Class Example

Suppose

$$\mathbf{f} = \begin{bmatrix} 2 \\ 1 \\ 0 \end{bmatrix}.$$

Then

$$e^{f_1} = e^2,$$

$$e^{f_2} = e,$$

$$e^{f_3} = 1.$$

Denominator:

$$e^2 + e + 1.$$

Therefore

$$\mathbf{p} = \frac{1}{e^2+e+1} \begin{bmatrix} e^2 \\ e \\ 1 \end{bmatrix}.$$

Numerically:

$$\mathbf{p} \approx \begin{bmatrix} 0.6652 \\ 0.2447 \\ 0.0900 \end{bmatrix}.$$

The probabilities sum to 1.

Softmax Preserves the Winning Class

Suppose

$$f_j > f_k.$$

Then

$$e^{f_j} > e^{f_k}.$$

The denominator is common to every class.

Therefore

$$p_j > p_k.$$

Thus

$$\arg \max_k p_k = \arg \max_k f_k.$$

Softmax changes the scale of the output,
but not the winner.

The hard decision regions from Lesson 1.19 remain unchanged.

Adding a Common Constant Changes Nothing

Replace every score by

$$f_k + c.$$

Then

$$\tilde{p}_k = \frac{e^{f_k+c}}{\sum_j e^{f_j+c}}.$$

Factor out

$$\begin{aligned} e^c: \\ \tilde{p}_k &= \frac{e^c e^{f_k}}{e^c \sum_j e^{f_j}} \\ &= p_k. \end{aligned}$$

Therefore softmax depends only on
relative score differences.

Exactly as the hard multiclass prediction rule did.

Why This Invariance Is Computationally Useful

Suppose one score is very large:

$$f_1 = 1000.$$

Directly computing

$$e^{1000}$$

can overflow numerically.

But adding or subtracting a common constant

does not change softmax.

So choose

$$c = -\max_j f_j.$$

Then the largest shifted score is

$$0.$$

All other shifted scores are nonpositive.

Thus

$$0 < e^{f_k - \max f} \leq 1.$$

This produces a stable computation.

Stable Softmax Formula

Let

$$M = \max_j f_j.$$

Then

$$p_k = \frac{e^{f_k - M}}{\sum_j e^{f_j - M}}.$$

This is mathematically identical

to ordinary softmax.

Numerically:

much safer.

This “subtract the maximum” idea

is standard in machine-learning software.

Probability Ratios

For any two classes

j and k :

$$\frac{p_j}{p_k}$$

$$\begin{aligned}
 &= \frac{e^{f_j} \sum_r e^{f_r}}{e^{f_k} \sum_r e^{f_r}} \\
 &= e^{f_j - f_k}.
 \end{aligned}$$

Therefore

$$\log\left(\frac{p_j}{p_k}\right) = f_j - f_k.$$

The difference of two class scores

is the log probability ratio between those classes.

This is the multiclass analogue of binary log-odds.

Decision Boundaries Reappear

Classes j and k

have equal probability when

$$p_j = p_k.$$

From the probability-ratio formula:

$$\begin{aligned}
 p_j &= p_k \\
 \Leftrightarrow \\
 f_j - f_k &= 0.
 \end{aligned}$$

Thus

$$p_j = p_k \quad \Leftrightarrow \quad f_j = f_k.$$

So softmax does not change

the pairwise boundaries derived in Lesson 1.19.

For linear scores:

they remain hyperplanes.

Binary Logistic Regression Is a Special Case

Take

$$K = 2.$$

Then

$$p_1 = \frac{e^{f_1}}{e^{f_1} + e^{f_2}}.$$

Divide numerator and denominator by

$$\begin{aligned}
 &e^{f_1}: \\
 p_1 &= \frac{1}{1 + e^{f_2 - f_1}}.
 \end{aligned}$$

Therefore

$$p_1 = \sigma(f_1 - f_2).$$

Likewise

$$p_2 = \sigma(f_2 - f_1).$$

Binary sigmoid classification

is exactly the two-class version of softmax.

One-Hot Encoding of the True Class

Suppose

$$K = 4$$

and the true class is

$$y = 3.$$

Represent the observed label by

$$\mathbf{y} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}.$$

Exactly one entry is 1.

All others are 0.

This is called

one-hot encoding.

For class y :

$$y_k = \begin{cases} 1, & k = y, \\ 0, & k \neq y. \end{cases}$$

Probability of the Observed Class

The model predicts probabilities

$$p_1, \dots, p_K.$$

For a one-hot label vector,

the probability assigned to the observed outcome is

$$P(Y = y \mid \mathbf{x}) = \prod_{k=1}^K p_k^{y_k}.$$

Only one exponent equals 1.

Therefore

$$\prod_k p_k^{y_k} = p_y.$$

The likelihood contribution of one observation

is simply the probability assigned to its true class.

Categorical Cross-Entropy

Take the negative logarithm:

$$-\log\left(\prod_{k=1}^K p_k^{y_k}\right).$$

Thus

$$\ell(\mathbf{y}, \mathbf{p}) = -\sum_{k=1}^K y_k \log p_k.$$

This is the **categorical cross-entropy**.

Because the label is one-hot:

$$\ell = -\log p_y.$$

The loss depends only on
the probability assigned to the observed class.

A Numerical Cross-Entropy Example

Using

$$\mathbf{p} = (0.6652, 0.2447, 0.0900)^T,$$

suppose the true class is
2.

Then

$$\mathbf{y} = (0, 1, 0)^T.$$

Cross-entropy:

$$\begin{aligned}\ell &= -\log(0.2447) \\ &\approx 1.408.\end{aligned}$$

If the true class were 1:

$$\begin{aligned}\ell &= -\log(0.6652) \\ &\approx 0.408.\end{aligned}$$

The same predicted distribution
receives different loss
depending on the observed class.

Confidently Wrong Multiclass Predictions

Suppose the true class is
3.

Compare two models.

Model A:

$$\mathbf{p} = (0.10, 0.20, 0.70).$$

Loss:

$$-\log(0.70) \approx 0.357.$$

Model B:

$$\mathbf{p} = (0.94, 0.05, 0.01).$$

Loss:

$$-\log(0.01) \approx 4.605.$$

Categorical cross-entropy strongly penalizes
confident probability assigned to the wrong class.

Write Cross-Entropy Directly in Terms of Scores

Softmax gives

$$p_y = \frac{e^{f_y}}{\sum_j e^{f_j}}.$$

Therefore

$$\begin{aligned} \ell &= -\log p_y \\ &= -\log\left(\frac{e^{f_y}}{\sum_j e^{f_j}}\right). \end{aligned}$$

Thus

$$\ell = -f_y + \log\left(\sum_{j=1}^K e^{f_j}\right).$$

This is the multiclass analogue of
the binary logistic-loss expression.

Differentiate the Log-Sum-Exp Term

Let

$$S = \sum_j e^{f_j}.$$

Then

$$\begin{aligned} &\frac{\partial}{\partial f_k} \log S \\ &= \frac{1}{S} \frac{\partial S}{\partial f_k} \\ &= \frac{e^{f_k}}{\sum_j e^{f_j}} \\ &= p_k. \end{aligned}$$

The derivative of log-sum-exp
is softmax.

This is the key simplification.

Gradient With Respect to Each Score

Recall

$$\ell = -f_y + \log \sum_j e^{f_j}.$$

For class k :

$$\begin{aligned} &\frac{\partial(-f_y)}{\partial f_k} \\ &= -y_k. \end{aligned}$$

The second term contributes

$$p_k.$$

Therefore

$$\frac{\partial \ell}{\partial f_k} = p_k - y_k.$$

Vector form:

$$\nabla_{\mathbf{f}} \ell = \mathbf{p} - \mathbf{y}.$$

Exactly the same “prediction minus label” structure we saw in binary cross-entropy.

A Numerical Gradient Example

Again use

$$\mathbf{f} = (2, 1, 0)^T,$$

so

$$\mathbf{p} \approx (0.6652, 0.2447, 0.0900)^T.$$

Suppose the true class is 2.

Then

$$\mathbf{y} = (0, 1, 0)^T.$$

Thus

$$\begin{aligned} \nabla_{\mathbf{f}} \ell &= \mathbf{p} - \mathbf{y} \\ &\approx \begin{bmatrix} 0.6652 \\ -0.7553 \\ 0.0900 \end{bmatrix}. \end{aligned}$$

Gradient descent:

lowers scores 1 and 3,

and raises the true-class score 2.

Gradient With Respect to the Weight Matrix

Recall

$$\mathbf{f} = W\tilde{\mathbf{X}}.$$

For class k :

$$f_k = \mathbf{w}_k^T \tilde{\mathbf{X}}.$$

So

$$\nabla_{\mathbf{w}_k} \ell = (p_k - y_k) \tilde{\mathbf{X}}.$$

Stacking all classes:

$$\nabla_W \ell = (\mathbf{p} - \mathbf{y}) \tilde{\mathbf{X}}^T.$$

This is an outer product.

Probability error by class
times
input features.

Batch Gradient

Suppose:

X contains augmented observations in its rows,

W contains class parameter vectors in its rows.

Then the score matrix is

$$F = XW^T.$$

Let
 P
be the matrix of softmax probabilities,

and
 Y
the one-hot label matrix.

For P observations:

$$\nabla_W g = \frac{1}{P} (P - Y)^T X.$$

This is the vectorized multiclass gradient.

No observation-by-observation calculus is needed once the matrix form is known.

Why the Gradient Components Sum to Zero

For one observation:

$$\begin{aligned} &\sum_k (p_k - y_k) \\ &= \sum_k p_k - \sum_k y_k. \end{aligned}$$

Both sums equal 1.

Therefore
 $\sum_k (p_k - y_k) = 0.$

So increasing the true-class score
is balanced by pressure to decrease competing scores.

This also reflects softmax's invariance
to adding a common constant to every score.

A Redundancy in the Score Parameterization

Softmax probabilities do not change

if the same quantity is added to every class score.

Therefore the absolute score level
is not identifiable from the probabilities.

Only score differences matter.

In practice this is handled naturally by
optimization conventions,
reference-class parameterizations,
or regularization.

For prediction:
there is no problem.

The probabilities and class rankings are uniquely determined
even though several score parameterizations may represent them.

Softmax in Python

A stable implementation:

```
def softmax(scores):  
    shifted = scores - np.max(scores)  
  
    exp_scores = np.exp(shifted)  
  
    return exp_scores / np.sum(exp_scores)
```

Example:

```
scores = np.array([2.0, 1.0, 0.0])  
  
p = softmax(scores)  
  
print(p)
```

Output:

```
[0.66524096 0.24472847 0.09003057]
```

The subtraction does not change the probabilities.

Categorical Cross-Entropy in Python

If the true class is given as an index:

```
def categorical_cross_entropy(scores, true_class):  
    p = softmax(scores)  
  
    return -np.log(p[true_class])
```

Here Python classes are indexed
 $0, \dots, K-1$.

For mathematical class 2:

```
loss = categorical_cross_entropy(  
    scores,  
    true_class=1
```

)

Output:

approximately
1.4076.

A More Stable Loss From Scores

We can avoid computing tiny probabilities explicitly.

For one observation:

$$\ell = -f_y + \log \sum_j e^{f_j}.$$

Subtract

$$M = \max_j f_j:$$

$$\ell = -(f_y - M) + \log \sum_j e^{f_j - M}.$$

Python:

```
def cross_entropy_from_scores(scores, true_class):  
    M = np.max(scores)  
    shifted = scores - M  
  
    log_sum_exp = np.log(  
        np.sum(np.exp(shifted))  
    )  
  
    return log_sum_exp - shifted[true_class]
```

Same loss.

Better numerical behavior.

Softmax for Many Observations

Suppose each row of

scores

contains one observation's class scores.

Then:

```
def softmax_rows(scores):  
    shifted = scores - np.max(  
        scores,  
        axis=1,  
        keepdims=True  
    )  
  
    exp_scores = np.exp(shifted)  
  
    return exp_scores / np.sum(  
        exp_scores,  
        axis=1,  
        keepdims=True  
    )
```

New Python idea:

`keepdims=True`

keeps the maximum or sum as a column.

That lets NumPy apply the operation
separately to every row.

One-Hot Labels in Python

Suppose

```
labels = np.array([0, 2, 1, 2])
K = 3
```

Create a one-hot matrix:

```
Y = np.eye(K)[labels]
```

Output:

```
[[1.  0.  0.]
 [0.  0.  1.]
 [0.  1.  0.]
 [0.  0.  1.]]
```

New Python idea:

```
np.eye(K)
```

creates the $K \times K$ identity matrix.

Indexing its rows by class labels
produces one-hot vectors.

Vectorized Gradient in Python

Suppose

X

has observations in rows,

and

W

has class parameters in rows.

```
scores = X @ W.T
```

```
P = softmax_rows(scores)
```

```
grad_W = (P - Y).T @ X / len(X)
```

This is exactly

$$\nabla_W g = \frac{1}{P} (P - Y)^T X.$$

Again:

the mathematical matrix formula
becomes executable code directly.

Softmax Is Not “Choosing Softly” During Prediction

Softmax produces probabilities.

The usual hard prediction is still

$$\hat{y} = \arg \max_k p_k = \arg \max_k f_k.$$

So why “soft” max?

The ordinary maximum returns only the winner.

Softmax gives a smooth distribution
that places more weight on larger scores
while retaining information about all classes.

This smoothness makes gradient-based training possible.

Cross-Entropy Versus Accuracy

Suppose the true class is
1.

Model A:
 $\mathbf{p} = (0.51, 0.49, 0).$

Model B:
 $\mathbf{p} = (0.99, 0.01, 0).$

Both predict class 1.

So accuracy treats them identically.

But cross-entropy gives
 $-\log(0.51) \approx 0.673$

versus
 $-\log(0.99) \approx 0.010.$

Cross-entropy trains the full probability distribution,
not merely the winning class.

The Multiclass Learning Pipeline

Input:
 $\tilde{\mathbf{x}}.$

Linear scores:
 $\mathbf{f} = W\tilde{\mathbf{x}}.$

Softmax probabilities:
$$p_k = \frac{e^{f_k}}{\sum_j e^{f_j}}.$$

Categorical cross-entropy:
$$\ell = -\sum_k y_k \log p_k.$$

Gradient:

$$\nabla_{\mathbf{f}} \ell = \mathbf{p} - \mathbf{y}.$$

Prediction:

$$\hat{y} = \arg \max_k p_k.$$

This is **multinomial logistic regression**
or **softmax regression**.

What did we learn?

- ◆ Softmax transforms K real-valued scores into positive class probabilities that sum to one.
- ◆ Softmax preserves the score ordering, so $\arg \max_k p_k = \arg \max_k f_k$.
- ◆ Adding the same constant to every score leaves the softmax probabilities unchanged and enables stable computation.
- ◆ Categorical cross-entropy is $-\sum_k y_k \log p_k$, which reduces to $-\log p_y$ for a one-hot label.
- ◆ Softmax plus categorical cross-entropy gives the simple score gradient $\nabla_{\mathbf{f}} \ell = \mathbf{p} - \mathbf{y}$.
- ◆ Binary logistic regression is the $K = 2$ special case of softmax regression.

Next

- We have now built binary and multiclass classifiers using full-batch gradients.
- Large datasets make computing every observation's gradient at every step expensive.
- Next we change the optimization procedure without changing the model.
- **1.21 - Stochastic and Mini-Batch Optimization**