Logistic Regression

Lecturer: Ngo Huy Bien
Software Engineering Department
Faculty of Information Technology
University of Science
Ho Chi Minh City, Vietnam
nhbien@fit.hcmus.edu.vn

Objectives

- To *write* a machine learning program recognizing Iris-Virginica using Logistic Regression
- To *write* a machine learning program recognizing a handwritten digit using Softmax Regression
- To explain machine learning *terminologies* via examples

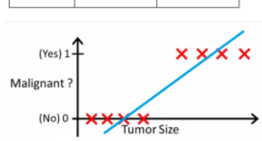
Contents

References

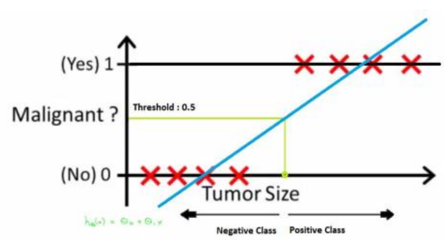
1. Aurelien Geron (2017). Hands on Machine Learning with Scikit Learn and TensorFlow. O'Reilly Media.
2. Umberto Michelucci (2018). Applied Deep Learning: A Case-Based Approach to Understanding Deep Neural Networks. Apress.

Classification Problems

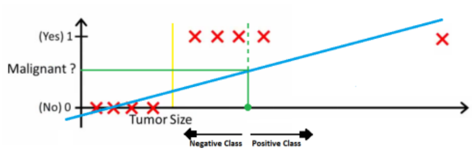
	Two Class Classification	
$y \in \{0, 1\}$	**1 or Positive Class**	**0 or Negative Class**
Email	Spam	Not Spam
Tumor	Malignant	Benign
Transaction	Fraudulent	Not Fraudulent



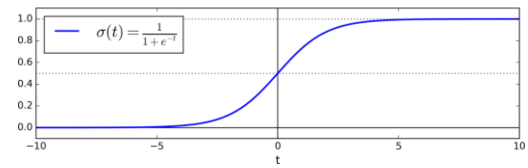
Using Linear Regression

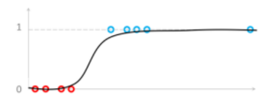


Why Not Use Linear Regression?



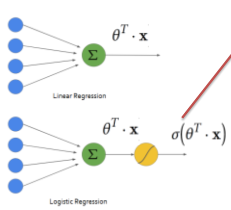
Logistic Function [1]





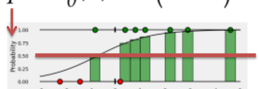
Logistic Regression Model

- Just like a Linear Regression model, a Logistic Regression model computes *a weighted sum* of the input features (plus a bias term), but instead of outputting the result directly like the Linear Regression model does, it *outputs* the logistic of this result.



$$\sigma(t) = \frac{1}{1 + \exp(-t)}$$

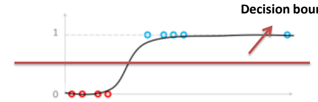
$$\hat{p} = h_\theta(\mathbf{x}) = \sigma(\theta^T \cdot \mathbf{x})$$

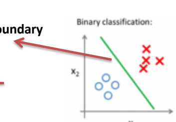


Binary Classifier

- Logistic Regression (also called Logit Regression) is commonly used to *estimate* the probability that an instance belongs to a particular class (e.g., what is the probability that this email is spam?).
- If the *estimated probability* is greater than 50%, then the model predicts that the instance belongs to that class (called the positive class, labeled "1"), or else it predicts that it does not (i.e., it belongs to the negative class, labeled "0").
- This makes it a *binary classifier*.

Decision boundary

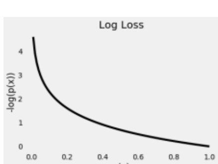




Cost Function (I)

- Cost function of a *single training instance*

$$c(\theta) = \begin{cases} -\log(\hat{p}) & \text{if } y = 1, \\ -\log(1-\hat{p}) & \text{if } y = 0. \end{cases} \quad \text{where} \quad \hat{p} = h_\theta(\mathbf{x}) = \sigma(\theta^T \cdot \mathbf{x})$$



Since we're trying to compute a *loss*, we need to penalize bad predictions.

y = 1, phat = 1 ⇒ cost ↓
y = 1, phat = 0 ⇒ cost ↑
y = 0, phat = 0 ⇒ cost ↓
y = 0, phat = 1 ⇒ cost ↑

Cost Function (II)

- Cost function of a *single training instance*

$$c(\theta) = \begin{cases} -\log(\hat{p}) & \text{if } y = 1, \\ -\log(1-\hat{p}) & \text{if } y = 0. \end{cases} \quad \text{where} \quad \hat{p} = h_\theta(\mathbf{x}) = \sigma(\theta^T \cdot \mathbf{x})$$

- Logistic Regression *cost function* (log loss)

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}log\left(\hat{p}^{(i)}\right) + \left(1 - y^{(i)}\right)log\left(1 - \hat{p}^{(i)}\right)\right]$$

- Logistic cost function *partial derivatives*

$$\frac{\partial}{\partial \theta_j} J(\theta) = \frac{1}{m}\sum_{i=1}^{m}\left(\sigma\left(\theta^T \cdot \mathbf{x}^{(i)}\right) - y^{(i)}\right)x_j^{(i)}$$

The Iris Dataset

- A famous dataset that contains the sepal and petal length and width of 150 iris flowers of *three different species*: Iris-Setosa, Iris-Versicolor, and Iris-Virginica.

Loading Data

```
from sklearn import datasets
import numpy as np
from sklearn.linear_model import LogisticRegression
import matplotlib.pyplot as plt

iris = datasets.load_iris()
print ('Dataset keys:', list(iris.keys()))
```

```
Dataset keys: ['data', 'target',
'target_names', 'DESCR', 'feature_names',
'filename']
```

Visualizing The Data

```
print (list(iris.keys()))
print (iris["target"])
print (iris["target_names"])
print (iris["feature_names"])
print (iris["data"][0:5, :])
```

```
['data', 'target', 'target_names', 'DESCR', 'feature_names', 'filename']
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2]
['setosa' 'versicolor' 'virginica']
['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal width
(cm)']
[[5.1 3.5 1.4 0.2]
 [4.9 3.  1.4 0.2]
 [4.7 3.2 1.3 0.2]
 [4.6 3.1 1.5 0.2]
 [5.  3.6 1.4 0.2]]
```

Preparing Training Data

```
X = iris["data"]
y = (iris["target"] == 2).astype(np.int) # 1 if Iris-
Virginica, else 0
print ('X shape:', X.shape)
print ('y shape:', y.shape)
print ('X[0]:', X[0])
print ('y:', y)
# print ('y[0]:', y[0])
```

```
X shape: (150, 4)
y shape: (150,)
X[0]: [5.1 3.5 1.4 0.2]
y: [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1]
```

Training A **LogisticRegression** Model

```
log_reg = LogisticRegression(multi_class ='ovr', solver = 'liblinear')
# multi_class : str, {‘ovr’, ‘multinomial’, ‘auto’}, default: ‘ovr’
# If the option chosen is ‘ovr’, then a binary problem is fit for each
label.
# solver : str, {‘newton-cg’, ‘lbfgs’, ‘liblinear’, ‘sag’, ‘saga’},
default: ‘liblinear’.
# Algorithm to use in the optimization problem.
# https://scikit-
learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegressi
on.html
log_reg.fit(X, y)
```

```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
          intercept_scaling=1, max_iter=100, multi_class='ovr',
          n_jobs=None, penalty='l2', random_state=None, solver='liblinear',
          tol=0.0001, verbose=0, warm_start=False)
```

Evaluating The Model

```
from sklearn.model_selection import
cross_val_predict
from sklearn.metrics import precision_score,
recall_score
y_train_pred_log_reg =
cross_val_predict(log_reg, X, y, cv=3)
print ('Precision score:', precision_score(y,
y_train_pred_log_reg))
print ('Recall score:', recall_score(y,
y_train_pred_log_reg))
```

```
Precision score: 0.9230769230769231
Recall score: 0.96
```

Making Prediction

```
print('Is X[3] Iris-Virginica?',
log_reg.predict(X[3].reshape(-1, 4)))
print('Is X[145] Iris-Virginica?',
log_reg.predict(X[145].reshape(-1, 4)))
X_new = [[4.9, 3.2, 1.3, 0.2]]
print('Is X_new Iris-Virginica?',
log_reg.predict(X_new))
```

```
Is X[3] Iris-Virginica? [0]
Is X[145] Iris-Virginica? [1]
Is X_new Iris-Virginica? [0]
```

Logistic Regression from Scratch [2]

The Single-Variable Chain Rule

$f : X \to Y \quad g : Y \to Z \xrightarrow{\text{Def.}} g \circ f : X \to Z \quad (g \circ f)(x) = g(f(x))$

$$\left.\begin{array}{ll} f : \mathbb{R} \to \mathbb{R} & g : \mathbb{R} \to \mathbb{R} \\ f(x) = x + 1 & g(x) = x^2 \end{array}\right\} \xrightarrow{\text{Def.}} (g \circ f)(x) = g(f(x)) = g(x+1) = (x+1)^2$$

The single-variable chain rule:

$$(f \circ g)'(x_0) = f'(g(x_0))g'(x_0)$$

$$\left.\begin{array}{l} g(x) = x + 1 \\ w(x) = x^2 \\ v(x) = sin(x) \end{array}\right\} \Rightarrow \begin{aligned} \frac{df(x)}{dx} = \frac{dv(w(g(x)))}{dx} &= v'(w(g(x)))w(g(x))'(x) \\ &= cos(w(g(x)))2(x+1) \\ &= 2cos[(x+1)^2](x+1) \end{aligned}$$

https://eli.thegreenplace.net/2016/the-chain-rule-of-calculus/

Calculating Derivatives

Model: $\hat{y}^{[i]} = \sigma\left(z^{[i]}\right) = \sigma\left(\boldsymbol{w}^T \boldsymbol{x}^{[i]} + b\right) = \sigma\left(w_1 x_1^{[i]} + w_2 x_2^{[i]} + \ldots + w_{n_x} x_{n_x}^{[i]} + b\right)$

Cost: $\mathcal{L}\left(\hat{y}^{[i]}, y^{[i]}\right) = -\left(y^{[i]} \log \hat{y}^{[i]} + \left(1 - y^{[i]}\right) \log\left(1 - \hat{y}^{[i]}\right)\right)$

Derivatives:

$$\frac{\partial \mathcal{L}\left(\hat{y}^{[i]}, y^{[i]}\right)}{\partial \hat{y}^{[i]}} = -\frac{y^{[i]}}{\hat{y}^{[i]}} + \frac{1 - y^{[i]}}{1 - \hat{y}^{[i]}}$$

$$\frac{d\hat{y}^{[i]}}{dz^{[i]}} = \hat{y}^{[i]}\left(1 - \hat{y}^{[i]}\right)$$

$$\frac{\partial z^{[i]}}{\partial w_j} = x_j^{[i]}$$

$$\frac{\partial z^{[i]}}{\partial b} = 1$$

Applying chain rules:

$$\frac{\partial \mathcal{L}\left(\hat{y}^{[i]}, y^{[i]}\right)}{\partial w_j} = \frac{\partial \mathcal{L}\left(\hat{y}^{[i]}, y^{[i]}\right)}{\partial \hat{y}^{[i]}} \frac{d\hat{y}^{[i]}}{dz^{[i]}} \frac{\partial z^{[i]}}{\partial w_j}$$

$$\frac{\partial \mathcal{L}\left(\hat{y}^{[i]}, y^{[i]}\right)}{\partial b} = \frac{\partial \mathcal{L}\left(\hat{y}^{[i]}, y^{[i]}\right)}{\partial \hat{y}^{[i]}} \frac{d\hat{y}^{[i]}}{dz^{[i]}} \frac{\partial z^{[i]}}{\partial b}$$

Stochastic Gradient Descent

$$w_{j,[n+1]} = w_{j,[n]} - \gamma \frac{\partial \mathcal{L}\left(\hat{y}^{[i]}, y^{[i]}\right)}{\partial w_j} = w_{j,[n]} - \gamma \left(1 - \hat{y}^{[i]}\right) x_j^{[i]}$$

$$b_{[n+1]} = b_{[n]} - \gamma \frac{\partial \mathcal{L}\left(\hat{y}^{[i]}, y^{[i]}\right)}{\partial b} = b_{j,[n]} - \gamma \left(1 - \hat{y}^{[i]}\right)$$

Vectorized Representation

Model: $Z = W^T X + B \quad \hat{Y} = \sigma(Z)$

Cost: $J(\boldsymbol{w}, b) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}\left(\hat{y}^{[i]}, y^{[i]}\right) \quad \mathcal{L}\left(\hat{y}^{[i]}, y^{[i]}\right) = -\left(y^{[i]} \log \hat{y}^{[i]} + \left(1 - y^{[i]}\right) \log\left(1 - \hat{y}^{[i]}\right)\right)$

Derivative:

$$\frac{\partial J(\boldsymbol{w}, b)}{\partial w_j} = \frac{1}{m}\sum_{i=1}^{m} \frac{\partial \mathcal{L}\left(\hat{y}^{[i]}, y^{[i]}\right)}{\partial w_j} = \frac{1}{m}\sum_{i=1}^{m} \frac{\partial \mathcal{L}\left(\hat{y}^{[i]}, y^{[i]}\right)}{\partial \hat{y}^{[i]}} \frac{d\hat{y}^{[i]}}{dz^{[i]}} \frac{\partial z^{[i]}}{\partial w_j} = \frac{1}{m}\sum_{i=1}^{m} \left(1 - \hat{y}^{[i]}\right) x_j^{[i]}$$

Vectorized representation: $\nabla_{\boldsymbol{w}} J(\boldsymbol{w}, b) = \frac{1}{m} X \left(\hat{Y} - Y\right)^T$

Derivative:

$$\frac{\partial J(\boldsymbol{w}, b)}{\partial b} = \frac{1}{m}\sum_{i=1}^{m} \frac{\partial \mathcal{L}\left(\hat{y}^{[i]}, y^{[i]}\right)}{\partial b} = \frac{1}{m}\sum_{i=1}^{m} \frac{\partial \mathcal{L}\left(\hat{y}^{[i]}, y^{[i]}\right)}{\partial \hat{y}^{[i]}} \frac{d\hat{y}^{[i]}}{dz^{[i]}} \frac{\partial z^{[i]}}{\partial b} = \frac{1}{m}\sum_{i=1}^{m} \left(1 - \hat{y}^{[i]}\right)$$

Vectorized representation: $\frac{\partial J(\boldsymbol{w}, b)}{\partial b} = \frac{1}{m}\sum_{i=1}^{m} \left(\hat{Y}_i - Y_i\right)$

Batch Gradient Descent

$$\boldsymbol{w}_{[n+1]} = \boldsymbol{w}_{[n]} - \gamma \frac{1}{m} X \left(\hat{Y} - Y\right)^T$$

$$b_{[n+1]} = b_{[n]} - \gamma \frac{1}{m} \sum_{i=1}^{m} \left(\hat{Y}_i - Y_i\right)$$

Computing sigmoid Activation

```
def sigmoid(z):
    s = 1.0 / (1.0 + np.exp(-z))
    return s

print ('sigmoid(0):', sigmoid(0))
print ('sigmoid(100):', sigmoid(100))
print ('sigmoid(-100):', sigmoid(-100))
```

```
sigmoid(0): 0.5
sigmoid(100): 1.0
sigmoid(-100): 3.7200759760208356e-44
```

Initializing W and b

```
def initialize(dim):
    W = np.zeros((dim, 1))
    b = 0
    return W, b

W, b = initialize(4)
print ('W:', W)
print ('b:', b)
```

```
W: [[0.]
 [0.]
 [0.]
 [0.]]
b: 0
```

Calculating Derivatives and Cost

```
def calculate_derivatives_and_cost(W, b, X, y):
    n, m = X.shape
    Z = np.dot(W.T, X) + b
    yhat = sigmoid(Z)

    cost = -1.0/m*np.sum(y*np.log(yhat)+(1.0-y)*np.log(1.0-yhat))

    dW = 1.0/m*np.dot(X, (yhat-y).T)
    db = 1.0/m*np.sum(yhat-y)
    derivatives = {"dW": dW, "db":db}

    return derivatives, cost

derivatives, cost = calculate_derivatives_and_cost(W, b, X.T, y)
# print ('W:', W)
# print ('b:', b)
print ('dW:', derivatives["dW"])
print ('db:', derivatives["db"])
print ('cost:', cost)
```

```
dW: [[ 0.72566667]
 [ 0.53733333]
 [ 0.02833333]
 [-0.07566667]]
db: 0.16666666666666669
cost: 0.6931471805599454
```

Training the Model

```
def optimize(W, b, X, y, num_iterations, learning_rate, print_cost =
False):
    costs = []
    for i in range(num_iterations):
        derivatives, cost = calculate_derivatives_and_cost(W, b, X, y)
        dW = derivatives ["dW"]
        db = derivatives ["db"]
        W = W - learning_rate*dW
        b = b - learning_rate*db
        if 0 == i % 1000:
            costs.append(cost)
        if print_cost and 0 == i % 1000:
            print ("Cost (iteration %i) = %f" %(i, cost))
    derivatives = {"dW": dW, "db": db}
    params = {"W": W, "b": b}
    return params, derivatives, costs
print ('W:', W)
print ('b:', b)
params, derivatives, costs = optimize(W, b, X.T, y, 5000, 0.1, True)
print ('W:', params['W'])
print ('b:', params['b'])
```

Training the Model: Output

```
W: [[0.]
 [0.]
 [0.]
 [0.]]
b: 0
Cost (iteration 0) = 0.693147
Cost (iteration 1000) = 0.126957
Cost (iteration 2000) = 0.099479
Cost (iteration 3000) = 0.087918
Cost (iteration 4000) = 0.081294
W: [[-3.37535802]
 [-3.32169488]
 [ 4.96017562]
 [ 5.53844134]]
b: -2.915667794269605
```

Making Predictions

```
def predict (W, b, X_new):
    n, m = X_new.shape
    y_prediction = np.zeros((1, m))
    W = W.reshape(n, 1)
    Z = np.dot(W.T, X_new) + b
    A = sigmoid(Z)
    for i in range(A.shape[1]):
        if (A[:,i] > 0.5):
            y_prediction[:, i] = 1
        elif (A[:,i] <= 0.5):
            y_prediction[:, i] = 0
    return y_prediction

final_W = params['W']
final_b = params['b']
print('Is X[3] Iris-Virginica?', predict(final_W, final_b, X[3].reshape(-1, 4).T))
print('Is X[145] Iris-Virginica?', predict(final_W,final_b, X[145].reshape(-1, 4).T))
X_new_2 = np.array([[4.9, 3.2, 1.3, 0.2]])
print('Is X_new_2 Iris-Virginica?', predict(final_W, final_b, X_new_2.reshape(-1, 4).T))
```

```
Is X[3] Iris-Virginica? [[0.]]
Is X[145] Iris-Virginica? [[1.]]
Is X_new_2 Iris-Virginica? [[0.]]
```

Evaluating the Model

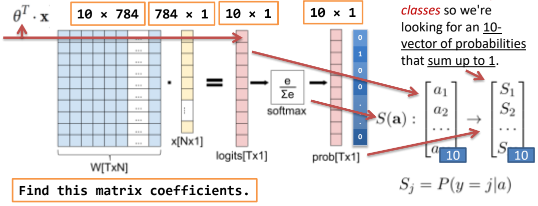
```
def fit(X_train, y_train, num_iterations = 50000, learning_rate = 0.1,
print_cost = True):
    n, m = X_train.shape
    W, b = initialize(n)
    params, derivatives, costs = optimize(W, b, X_train, y_train,
num_iterations, learning_rate, print_cost)
    W = params["W"]
    b = params["b"]
    y_prediction_train = predict(W, b, X_train)
    train_accuracy = 100.0 - np.mean(np.abs(y_prediction_train -
y_train)*100.0)
    model_info = {"costs": costs, "W": W, "b": b,
                  "learning_rate": learning_rate, "num_iterations":
num_iterations,
                  "t_prediction_train": y_prediction_train}
    print ("Train accuracy: ", train_accuracy)
    return model_info

model_info = fit(X.T, y)
```

```
Cost (iteration 0) = 0.693147
Cost (iteration 10000) = 0.066390
Cost (iteration 20000) = 0.058997
Cost (iteration 30000) = 0.055479
Cost (iteration 40000) = 0.053210
Train accuracy:  98.0
```

Recognizing A Handwritten Digit

- We have an *input* ***x*** with 784 features, and 10 possible *output classes*.
- The *weight matrix W (9x784)* is used to transform ***x*** into a vector with 10 elements (called "logits"), and the ***softmax*** *function* is used to "collapse" the logits into a vector of probabilities denoting the probability of ***x*** belonging to each one of the 10 output classes.

We have *10 output classes* so we're looking for an 10-vector of probabilities that sum up to 1.



The **softmax** Function

https://eli.thegreenplace.net/2016/the-softmax-function-and-its-derivative/

- The *softmax function* takes an N-dimensional vector of arbitrary real values and produces another N-dimensional vector with real values in the range (0, 1) that add up to 1.0.
- It maps

$$S(\mathbf{a}) : \mathbb{R}^N \rightarrow \mathbb{R}^N$$

$$S(\mathbf{a}) : \begin{bmatrix} a_1 \\ a_2 \\ \cdots \\ a_N \end{bmatrix} \rightarrow \begin{bmatrix} S_1 \\ S_2 \\ \cdots \\ S_N \end{bmatrix}$$

- And the actual per-element formula is:

$$S_j = \frac{e^{a_j}}{\sum_{k=1}^{N} e^{a_k}} \qquad \forall j \in 1..N$$

Computing The **softmax** Function

```
def softmax(a):
    """Compute the softmax of vector a."""
    exps = np.exp(a)
    return exps / np.sum(exps)

a_1 = np.array([1, 2, 3])
print ('softmax([1, 2, 3]) =', softmax(a_1))
a_2 = np.array([2, 2, 3])
print ('softmax(2, 2, 3) =', softmax(a_2))
```

```
softmax([1, 2, 3]) = [0.09003057 0.24472847 0.66524096]
softmax(2, 2, 3) = [0.21194156 0.21194156 0.57611688]
```

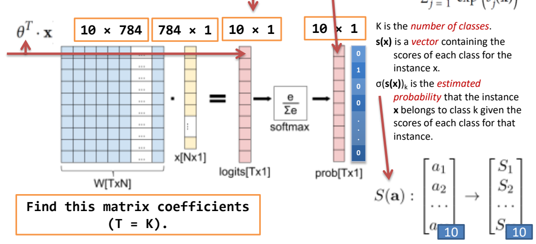
Softmax Regression

- Softmax score for class k

$$s_k(\mathbf{x}) = \theta_k^{\,T} \cdot \mathbf{x}$$

Softmax function:

$$\hat{p}_k = \sigma(\mathbf{s}(\mathbf{x}))_k = \frac{\exp\left(s_k(\mathbf{x})\right)}{\sum_{j=1}^{K} \exp\left(s_j(\mathbf{x})\right)}$$



K is the *number of classes*.
s(x) is a *vector* containing the scores of each class for the instance x.
σ(s(x))$_k$ is the *estimated probability* that the instance **x** belongs to class k given the scores of each class for that instance.

Find this matrix coefficients (T = K).

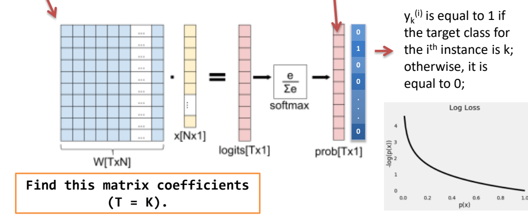
Cost Function

- Cross entropy *cost function*

$$J(\Theta) = -\frac{1}{m} \sum_{i=1}^{m} \sum_{k=1}^{K} y_k^{(i)} \log\left(\hat{p}_k^{(i)}\right)$$

Softmax function:

$$\hat{p}_k = \sigma(\mathbf{s}(\mathbf{x}))_k = \frac{\exp\left(s_k(\mathbf{x})\right)}{\sum_{j=1}^{K} \exp\left(s_j(\mathbf{x})\right)}$$



$y_k^{(i)}$ is equal to 1 if the target class for the i^{th} instance is k; otherwise, it is equal to 0;

Find this matrix coefficients (T = K).

Gradient Vector

- Cross entropy *gradient vector* for class k

$$\nabla_{\theta_k} J(\Theta) = \frac{1}{m} \sum_{i=1}^{m} \left(\hat{p}_k^{(i)} - y_k^{(i)}\right) \mathbf{x}^{(i)}$$

Vector Parameters Vector

where

- $y_k^{(i)}$ is equal to 1 if the target class for the i^{th} instance is k; otherwise, it is equal to 0;
- and

$$\hat{p}_k = \sigma(\mathbf{s}(\mathbf{x}))_k = \frac{\exp\left(s_k(\mathbf{x})\right)}{\sum_{j=1}^{K} \exp\left(s_j(\mathbf{x})\right)}$$

Downloading The MNIST Dataset

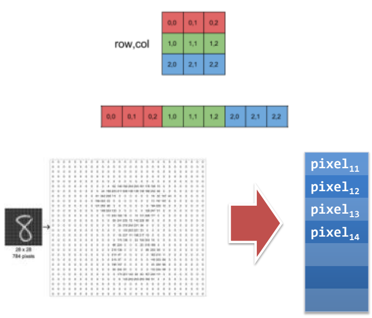
```
import numpy as np
from sklearn.datasets import fetch_openml

def sort_by_target(mnist):
    reorder_train = np.array(sorted([(target, i) for i, target in
enumerate(mnist.target[:60000])]))[:, 1]
    reorder_test = np.array(sorted([(target, i) for i, target in
enumerate(mnist.target[60000:])]))[:, 1]
    mnist.data[:60000] = mnist.data[reorder_train]
    mnist.target[:60000] = mnist.target[reorder_train]
    mnist.data[60000:] = mnist.data[reorder_test + 60000]
    mnist.target[60000:] = mnist.target[reorder_test + 60000]

try:
    mnist = fetch_openml('mnist_784', version=1, cache=True)
    mnist.target = mnist.target.astype(np.int8) # fetch_openml() returns targets
as strings
    sort_by_target(mnist) # fetch_openml() returns an unsorted dataset
except ImportError:
    from sklearn.datasets import fetch_mldata
    mnist = fetch_mldata('MNIST original')
```

Loading Data

```
X, y = mnist["data"], mnist["target"]
print('X.shape:', X.shape)
print('y.shape:', y.shape)
```

```
X.shape: (70000, 784)
y.shape: (70000,)
```

Reshaping Matrix To Vector



Visualizing The Data

```
%matplotlib inline
import matplotlib
import matplotlib.pyplot as plt
print(y[36000])
some_digit = X[36000]
some_digit_image = some_digit.reshape(28, 28)
plt.imshow(some_digit_image, cmap =
matplotlib.cm.binary,
interpolation="nearest")
plt.axis("off")
plt.show()
```

Preparing Training Data

- The MNIST dataset is actually already split into a *training set* (the first 60,000 images) and a *test set* (the last 10,000 images):

```
X_train, X_test, y_train, y_test = X[:60000], X[60000:], y[:60000],
y[60000:]

# Shuffle the training set;
# this will guarantee that all cross-validation folds will be similar
import numpy as np
shuffle_index = np.random.permutation(60000)
X_train, y_train = X_train[shuffle_index], y_train[shuffle_index]
print ('X_train.shape:', X_train.shape)
print ('y_train.shape:', y_train.shape)
print ('X_test.shape:', X_test.shape)
print ('y_test.shape:', y_test.shape)
```

```
X_train.shape: (60000, 784)
y_train.shape: (60000,)
X_test.shape: (10000, 784)
y_test.shape: (10000,)
```

Normalizing Image Inputs

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X_train_scaled =
scaler.fit_transform(X_train.astype(np.float64))
X_test_scaled = scaler.fit_transform(X_test.astype(np.float64))
print ('X_train_scaled.shape:', X_train_scaled.shape)
print ('X_test_scaled.shape:', X_test_scaled.shape)
```

```
X_train_scaled.shape: (60000, 784)
X_test_scaled.shape: (10000, 784)
```

Training A **Softmax Regression** Model

```
from sklearn.linear_model import LogisticRegression
softmax_reg = LogisticRegression(multi_class="multinomial",
solver="lbfgs", C=10)
softmax_reg.fit(X_train_scaled, y_train)
```

```
LogisticRegression(C=10, class_weight=None, dual=False,
fit_intercept=True,
          intercept_scaling=1, max_iter=100, multi_class='multinomial',
          n_jobs=None, penalty='l2', random_state=None, solver='lbfgs',
          tol=0.0001, verbose=0, warm_start=False)
```

Making Prediction

```
softmax_reg.predict([X_test_scaled[4000]])
#The classifier guesses that this image
represents a 3 (True).
```

```
array([3], dtype=int8)
```

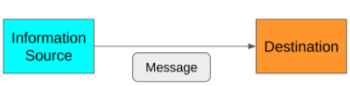
Verification

```
%matplotlib inline
import matplotlib
import matplotlib.pyplot as plt
print(y_test[4000])
X_test_4000 = X_test[4000]
X_test_4000_image = X_test_4000.reshape(28, 28)
plt.imshow(X_test_4000_image, cmap =
matplotlib.cm.binary,
interpolation="nearest")
plt.axis("off")
plt.show()
```


```
y_test[4000]: 3
```

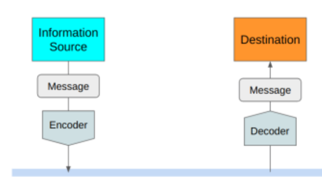
Who Invented Entropy and Why?

- https://towardsdatascience.com/demystifying-entropy-f2c3221e2550
- In 1948, Claude Shannon introduced the concept of *information entropy* in his paper “A Mathematical Theory of Communication”.
- Shannon was looking for a way to *efficiently* send messages without losing any information.
- Shannon measured the efficiency in terms of *average message length*.



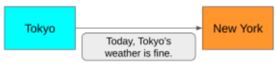
Information Entropy

- Shannon defined the *entropy* as the smallest possible average size of lossless **encoding** of the messages sent from the source to the destination.

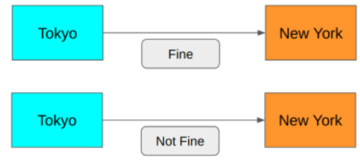


Efficient Encoding Example (I)

- Suppose you want to send a message from Tokyo to New York regarding Tokyo's weather today.
- Is this efficient?

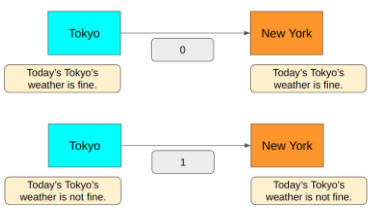


Efficient Encoding Example (II)



- It is much shorter. Can we do better?

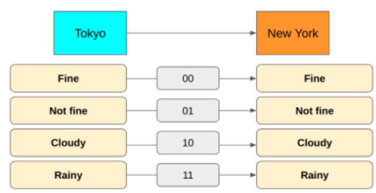
Efficient Encoding Example (III)



- But if we also want to talk about "Cloudy" or "Rainy", the 1-bit encoding won't be able to cover all cases. What can we do?

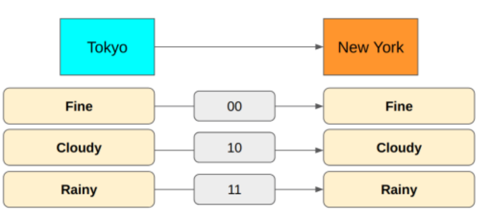
Efficient Encoding Example (IV)

- How about using 2 bits instead?
- Is this efficient?



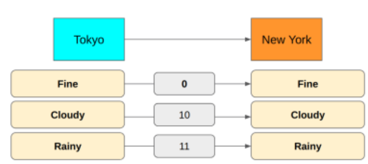
Efficient Encoding Example (V)

- *Semantically*, "Cloudy" and "Rainy" are both "Not fine" as well.
- Is this efficient?



Efficient Encoding Example (VI)

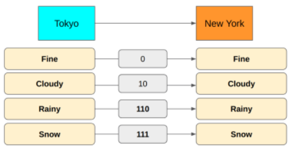
- "Fine" is "00" but we don't need the second zero.
- The first bit indicates "Fine or not".
- The second bit indicates "Cloudy or Rainy".



- But if we also want to talk about "Snow", what can we do?

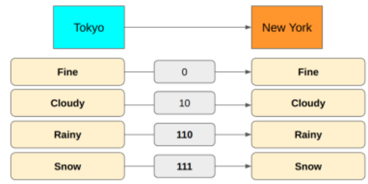
Efficient Encoding Example (VII)

- Does this encoding work?



- The encoding now uses the first bit for “Fine or not”, the second bit for “Cloudy or others” and the third bit for “Rain or snow”.
- “010110111” means “Fine, Cloudy, Rainy, Snow”.
- “110010111” means “Rainy, Fine, Cloudy, Snow”.

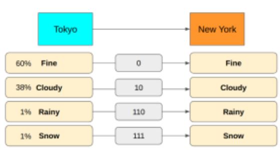
Are We Done?



- Have we achieved the *smallest* possible average encoding size using the above lossless encoding?
- How do we *calculate* the average encoding size anyway?

Average Bits Per Message

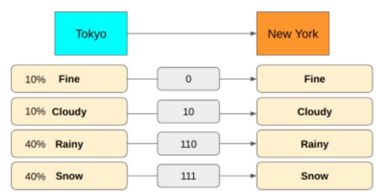
- Let’s suppose Tokyo is sending the weather messages to New York *many times* (say, every hour) using the lossless encoding we’ve discussed so far.



- Then, we can calculate the *average number of bits* used to send messages from Tokyo to New York.

`(0.6 x 1 bit)+(0.38 x 2 bits)+(0.01 x 3 bits)+(0.01 x 3 bits)=1.42 bits`

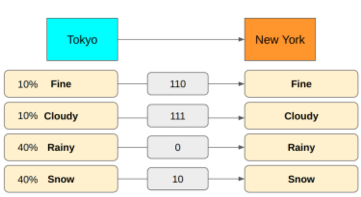
Another Example



`(0.1 x 1 bit)+(0.1 x 2 bits)+(0.4 x 3 bits)+(0.4 x 3 bits)=2.7 bits`

- Can we *reduce* the average encoding size by changing the encoding?

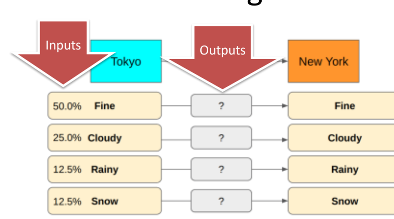
Smaller Average Encoding Size



`(0.1 x 3 bit)+(0.1 x 3 bits)+(0.4 x 1 bits)+(0.4 x 2 bits)=1.8 bits`

- How do we know if our encoding is the best one that achieves the *smallest* average encoding size?

The Smallest Average Encoding Size



- One way is to use many *trials and errors* to find the right encoding size.
- Is there any easy way to *calculate* the smallest possible average size?

The Minimum Number Of Bits (I)

- Suppose we have *8 message types*, each of which happens with *equal probability* (⅛ = 12.5%).
- What is the *minimum number of bits* we need to encode them without any ambiguity?

12.5%	A	?
12.5%	B	?
12.5%	C	?
12.5%	D	?
12.5%	E	?
12.5%	F	?
12.5%	G	?
12.5%	H	?

The Minimum Number Of Bits (II)

- When we need *N different values* expressed in bits, we need $\log_2 N$ bits and we don't need more than this.
- For example, $\log_2 8 = \log_2 2^3 = 3$ bits.

12.5%	A	000
12.5%	B	001
12.5%	C	010
12.5%	D	011
12.5%	E	100
12.5%	F	101
12.5%	G	110
12.5%	H	111

- In other words, if a message type happens 1 out of N times, the formula # Bits $= \log_2 N$ gives the minimum size required.
- As P=1/N is the probability of the message, the same thing can be expressed as:
  # Bits $= \log_2 N = \log_2(1/P) =$
  $= \log_2(P^{-1}) = -\log_2(P) = -\log_2(12.5/100)$

The Minimum *Average* Number Of Bits

- Calculate the minimum average encoding size (in bits) which is *entropy*:

$$Entropy = -\sum_i P(i)\log_2 P(i)$$

Inputs: Tokyo	Outputs	New York
50.0% Fine	?	Fine
25.0% Cloudy	?	Cloudy
12.5% Rainy	?	Rainy
12.5% Snow	?	Snow

P(Fine) = 0.5
$-\log_2 P(Fine) = -\log_2 0.5 = -\log_2 1/2 = \log_2 2 = 1$ bit

P(Cloudy) = 0.25
$-\log_2 P(Cloudy) = -\log_2 0.25 = -\log_2 1/4 = \log_2 4 = 2$ bits

P(Rainy) = 0.125
$-\log_2 P(Rainy) = -\log_2 0.125 = -\log_2 1/8 = \log_2 8 = 3$ bits

P(Snow) = 0.125
$-\log_2 P(Snow) = -\log_2 0.125 = -\log_2 1/8 = \log_2 8 = 3$ bits

```
Entropy = (0.5x1bit)+(0.25x2bits)+(0.125x3bits)+(0.125x3bits) = 1.75bits
```

Terminologies

- There are a lot of analogies used for entropy: *disorder*, *uncertainty*, *surprise*, *unpredictability* and *amount of information*.
- If entropy is high (encoding size is big on average), it means we have *many message types with small probabilities*. Hence, every time a new message arrives, you'd expect a different type than previous messages. You may see it as a disorder or uncertainty or unpredictability.
- When a message types with much *smaller probability* than other message types happens, it appears as a surprise because on average you'd expect other more frequently sent message types.
- A rare message type has *more information* than more frequent message types because it eliminates a lot of other probabilities and tells us more specific information. In the weather scenario, by sending "Rainy" which happens 12.5% of the times, we are reducing the uncertainty by 87.5% of the probability distribution ("Fine, Cloudy, Snow") provided we had no information before.

Entropy

- The entropy of a *discrete variable*:

$$Entropy = -\sum_i P(i)\log P(i)$$

- For *continuous variables*:

$$Entropy = -\int P(x)\log P(x)dx$$

where *x* is a continuous variable, and *P(x)* is the probability density function.

- For *both cases*: $Entropy = \mathbb{E}_{x\sim P}[-\log P(x)]$

or

$$H(P) = \mathbb{E}_{x\sim P}[-\log P(x)]$$

x~P means that we calculate the expectation with the probability distribution P.

Estimating Entropy

- The *entropy* tells us the theoretical minimum average encoding size for events that follow a particular probability distribution.

$$H(P) = \mathbb{E}_{x \sim P}[-\log P(x)]$$

- As long as we know the *probability distribution* of anything, we can calculate the entropy for it.
- If we do not know the probability distribution, we *cannot* calculate the entropy. So, we would need to estimate the probability distribution (or estimate the entropy).
- How to *estimate* the entropy? Using the *estimated probability distribution Q*, the estimated entropy would be:

$$EstimatedEntropy = \mathbb{E}_{x \sim Q}[-\log Q(x)]$$

Estimating Entropy Issues

$$EstimatedEntropy = \mathbb{E}_{x \sim Q}[-\log Q(x)]$$

- We are using the estimated probability distribution Q to calculate *the expectation*.
- We are estimating the *minimum encoding size* as -logQ based on the estimated probability distribution Q.
- As the estimated probability distribution Q affects both the expectation and encoding size estimation, the estimated entropy can be *very wrong*.

Cross-Entropy

- If we have the *real distribution P* after observing the weather for some period, we can calculate the realized average encoding size using the probability distribution P and the *actual encoding size* (based on Q) used during the weather reporting.
- It is called the *cross-entropy* between P and Q, which we can compare with the entropy.

$$CrossEntropy = \mathbb{E}_{x \sim P}[-\log Q(x)] \quad H(P,Q) = \mathbb{E}_{x \sim P}[-\log Q(x)]$$
$$Entropy = \mathbb{E}_{x \sim P}[-\log P(x)]$$

- We are comparing apple to apple as we use the same true distribution for both expectation calculations.
- We are comparing the *theoretical* minimum encoding and the *actual* encoding used in the weather reporting.

Cross-Entropy ≥ Entropy

$$CrossEntropy = \mathbb{E}_{x \sim P}[-\log Q(x)] \quad H(P,Q) = \mathbb{E}_{x \sim P}[-\log Q(x)]$$
$$Entropy = \mathbb{E}_{x \sim P}[-\log P(x)]$$

- For the *expectation*, we should use the true probability P as that tells the distribution of events.
- For the *encoding size*, we should use Q as that is used to encode messages.
- Since the *entropy* is the theoretical minimum average size, the *cross-entropy* is higher than or equal to the entropy but not less than that.
- If our estimate is perfect, Q = P and, hence, H(P, Q)=H(P). Otherwise, H(P, Q) > H(P).

Cross-Entropy as a Loss Function (I)

Animal	Label
Dog	[1 0 0 0 0]
Fox	[0 1 0 0 0]
Horse	[0 0 1 0 0]
Eagle	[0 0 0 1 0]
Squirrel	[0 0 0 0 1]

- We can treat *one hot encoding* as a probability distribution for each image.

$$\begin{aligned} P_1(dog) &= 1 \\ P_1(fox) &= 0 \\ P_1(horse) &= 0 \\ P_1(eagle) &= 0 \\ P_1(squirrel) &= 0 \end{aligned}$$

Cross-Entropy as a Loss Function (II)

Animal	Label
Dog	[1 0 0 0 0]
Fox	[0 1 0 0 0]
Horse	[0 0 1 0 0]
Eagle	[0 0 0 1 0]
Squirrel	[0 0 0 0 1]

$$\begin{aligned} P_2(dog) &= 0 \\ P_2(fox) &= 1 \\ P_2(horse) &= 0 \\ P_2(eagle) &= 0 \\ P_2(squirrel) &= 0 \end{aligned}$$

Cross-Entropy as a Loss Function (III)

Animal	Label
Dog	[1 0 0 0 0]
Fox	[0 1 0 0 0]
Horse	[0 0 1 0 0]
Eagle	[0 0 0 1 0]
Squirrel	[0 0 0 0 1]

$$H(P_1) = 0$$
$$H(P_2) = 0$$
$$H(P_3) = 0$$
$$H(P_4) = 0$$
$$H(P_5) = 0$$

- As such, the entropy of each image is all zero.
- In other words, one-hot encoded labels tell us what animal each image has with 100% certainty.

Cross-Entropy as a Loss Function (IV)

- Let's say we have a machine learning model that classifies those images.
- When we have *not adequately trained* the model, it may classify the first image (dog) as follows:

$$Q_1 = [0.4 \quad 0.3 \quad 0.05 \quad 0.05 \quad 0.2]$$
$$P_1 = [1 \quad 0 \quad 0 \quad 0 \quad 0]$$

- How well was the model's prediction? We can calculate the cross-entropy as follows:

$$\begin{aligned} H(P_1, Q_1) &= -\sum_i P_1(i) \log Q_1(i) \\ &= -(1 \log 0.4 + 0 \log 0.3 + 0 \log 0.05 + 0 \log 0.05 + 0 \log 0.2) \\ &= -\log 0.4 \\ &\approx 0.916 \end{aligned}$$

- This is *higher than the zero entropy* of the label but we do not have an intuitive sense of what this value means.

Cross-Entropy as a Loss Function (V)

- After the model is *well trained*, it may produce the following prediction for the first image.

$$Q_1 = [0.98 \quad 0.01 \quad 0 \quad 0 \quad 0.01]$$
$$P_1 = [1 \quad 0 \quad 0 \quad 0 \quad 0]$$

$$\begin{aligned} H(P_1, Q_1) &= -\sum_i P_1(i) \log Q_1(i) \\ &= -(1 \log 0.98 + 0 \log 0.01 + 0 \log 0 + 0 \log 0 + 0 \log 0.01) \\ &= -\log 0.98 \\ &\approx 0.02 \end{aligned}$$

- The cross-entropy *goes down* as the prediction gets more and more accurate. It becomes zero if the prediction is perfect. As such, the cross-entropy can be a *loss function* to train a classification model.

Binary Cross-Entropy

- For the *binary classifications*, the cross-entropy formula contains only two probabilities.
- For example, there are only *dogs or cats* in images.

$$\begin{aligned} H(P, Q) &= -\sum_{i=(cat,dog)} P(i) \log Q(i) \\ &= -P(cat) \log Q(cat) - P(dog) \log Q(dog) \end{aligned}$$

$$P(dog) = (1 - P(cat))$$

$$H(P, Q) = -P(cat) \log Q(cat) - (1 - P(cat)) \log(1 - Q(cat))$$

$$P = P(cat)$$
$$\hat{P} = Q(cat)$$
$$BinaryCrossEntropy = -P \log \hat{P} - (1 - P) \log(1 - \hat{P})$$

Jacobian Matrix Review

- https://explained.ai/matrix-calculus/index.html

$$f(x, y) = 3x^2y \qquad \nabla f(x, y) = [\frac{\partial f(x,y)}{\partial x}, \frac{\partial f(x,y)}{\partial y}] = [6yx, 3x^2]$$

$$g(x, y) = 2x + y^8 \quad \nabla g(x, y) = [2, 8y^7]$$

- If we have two functions, we can also *organize* their gradients into a matrix by stacking the gradients.
- When we do so, we get the *Jacobian matrix* (or just the Jacobian) where the gradients are rows:

$$J = \begin{bmatrix} \nabla f(x, y) \\ \nabla g(x, y) \end{bmatrix} = \begin{bmatrix} \frac{\partial f(x,y)}{\partial x} & \frac{\partial f(x,y)}{\partial y} \\ \frac{\partial g(x,y)}{\partial x} & \frac{\partial g(x,y)}{\partial y} \end{bmatrix} = \begin{bmatrix} 6yx & 3x^2 \\ 2 & 8y^7 \end{bmatrix}$$

Generalization of the Jacobian

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \qquad \begin{array}{rcl} y_1 & = & f_1(\mathbf{x}) \\ y_2 & = & f_2(\mathbf{x}) \\ & \vdots & \\ y_m & = & f_m(\mathbf{x}) \end{array} \qquad \begin{array}{l} y_1 = f_1(\mathbf{x}) = 3x_1^2 x_2 \\ y_2 = f_2(\mathbf{x}) = 2x_1 + x_2^8 \end{array}$$

$$\boxed{\mathbf{y} = \mathbf{f}(\mathbf{x})}$$

$$\frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \begin{bmatrix} \nabla f_1(\mathbf{x}) \\ \nabla f_2(\mathbf{x}) \\ \cdots \\ \nabla f_m(\mathbf{x}) \end{bmatrix} = \begin{bmatrix} \frac{\partial}{\partial \mathbf{x}} f_1(\mathbf{x}) \\ \frac{\partial}{\partial \mathbf{x}} f_2(\mathbf{x}) \\ \cdots \\ \frac{\partial}{\partial \mathbf{x}} f_m(\mathbf{x}) \end{bmatrix} = \begin{bmatrix} \frac{\partial}{\partial x_1} f_1(\mathbf{x}) & \frac{\partial}{\partial x_2} f_1(\mathbf{x}) & \cdots & \frac{\partial}{\partial x_n} f_1(\mathbf{x}) \\ \frac{\partial}{\partial x_1} f_2(\mathbf{x}) & \frac{\partial}{\partial x_2} f_2(\mathbf{x}) & \cdots & \frac{\partial}{\partial x_n} f_2(\mathbf{x}) \\ & & \cdots & \\ \frac{\partial}{\partial x_1} f_m(\mathbf{x}) & \frac{\partial}{\partial x_2} f_m(\mathbf{x}) & \cdots & \frac{\partial}{\partial x_n} f_m(\mathbf{x}) \end{bmatrix}$$

Possible Jacobian Shapes

	scalar x	vector $\mathbf{x}$
scalar f	$\frac{\partial f}{\partial x}$	$\frac{\partial f}{\partial \mathbf{x}}$
vector $\mathbf{f}$	$\frac{\partial \mathbf{f}}{\partial x}$	$\frac{\partial \mathbf{f}}{\partial \mathbf{x}}$

The Jacobian of **softmax** (I)

- Softmax is fundamentally a *vector function*.
- It takes a vector as *input* and produces a vector as *output*; in other words, it has multiple inputs and multiple outputs.
- Therefore, we cannot just ask for "the derivative of softmax"; We should instead specify:
  - Which component (*output element*) of softmax we're seeking to find the derivative of.
  - Since softmax has multiple inputs, with respect to which *input element* the partial derivative is computed.

$$S(\mathbf{a}) : \begin{bmatrix} a_1 \\ a_2 \\ \cdots \\ a_N \end{bmatrix} \rightarrow \begin{bmatrix} S_1 \\ S_2 \\ \cdots \\ S_N \end{bmatrix}$$

What we're looking for is the *partial derivatives*: $\frac{\partial S_i}{\partial a_j} = D_j S_i$

The Jacobian of **softmax** (II)

- Since **softmax** is a function $\mathbb{R}^N \rightarrow \mathbb{R}^N$, the most general "derivative" we compute for it is the *Jacobian matrix*:

$$S(\mathbf{a}) : \begin{bmatrix} a_1 \\ a_2 \\ \cdots \\ a_N \end{bmatrix} \rightarrow \begin{bmatrix} S_1 \\ S_2 \\ \cdots \\ S_N \end{bmatrix} \qquad \frac{\partial S_i}{\partial a_j} = D_j S_i \qquad DS = \begin{bmatrix} D_1 S_1 & \cdots & D_N S_1 \\ \vdots & \ddots & \vdots \\ D_1 S_N & \cdots & D_N S_N \end{bmatrix}$$

$$S_j = \frac{e^{a_j}}{\sum_{k=1}^{N} e^{a_k}} \quad \forall j \in 1..N \qquad D_j S_i = \frac{\partial S_i}{\partial a_j} = \frac{\partial \frac{e^{a_i}}{\sum_{k=1}^{N} e^{a_k}}}{\partial a_j}$$

? Calculate DS with N = 10 using $f'(x) = \frac{g'(x)h(x) - h'(x)g(x)}{[h(x)]^2}$

The Jacobian of **softmax** (III)

- i = j case

$$\frac{\partial \frac{e^{a_i}}{\sum_{k=1}^{N} e^{a_k}}}{\partial a_j} = \frac{e^{a_i}\Sigma - e^{a_j}e^{a_i}}{\Sigma^2} \longrightarrow \frac{\partial \frac{e^{a_i}}{\sum_{k=1}^{N} e^{a_k}}}{\partial a_j} = \frac{e^{a_i}\Sigma - e^{a_j}e^{a_i}}{\Sigma^2} = \frac{e^{a_i}}{\Sigma}\frac{\Sigma - e^{a_j}}{\Sigma} = S_i(1 - S_j)$$

- i ≠ j case

$$\frac{\partial \frac{e^{a_i}}{\sum_{k=1}^{N} e^{a_k}}}{\partial a_j} = \frac{0 - e^{a_j}e^{a_i}}{\Sigma^2} = -\frac{e^{a_j}}{\Sigma}\frac{e^{a_i}}{\Sigma} = -S_j S_i$$

$$\longrightarrow D_j S_i = \begin{cases} S_i(1 - S_j) & i = j \\ -S_j S_i & i \neq j \end{cases}$$

The Jacobian of **softmax** (IV)

$$DS = \begin{bmatrix} D_1 S_1 & \cdots & D_N S_1 \\ \vdots & \ddots & \vdots \\ D_1 S_N & \cdots & D_N S_N \end{bmatrix} \qquad S(\mathbf{a}) : \begin{bmatrix} a_1 \\ a_2 \\ \cdots \\ a_N \end{bmatrix} \rightarrow \begin{bmatrix} S_1 \\ S_2 \\ \cdots \\ S_N \end{bmatrix} \qquad \frac{\partial S_i}{\partial a_j} = D_j S_i$$

$$D_j S_i = \begin{cases} S_i(1 - S_j) & i = j \\ -S_j S_i & i \neq j \end{cases}$$

Using the Kronecker delta function:

$$\delta_{ij} = \begin{cases} 1 & i = j \\ 0 & i \neq j \end{cases} \longrightarrow D_j S_i = S_i(\delta_{ij} - S_j)$$

? D_1S_1 = ?, D_2S_1 = ?, D_3S_1 = ?, D_2S_2 = ?, D_2S_3 = ?, D_2S_{10} = ?

Multivariate Chain Rule

https://eli.thegreenplace.net/2016/the-chain-rule-of-calculus/

$f : X \to Y \quad g : Y \to Z$ **Def.** $g \circ f : X \to Z \quad (g \circ f)(x) = g(f(x))$

$f : \mathbb{R}^n \to \mathbb{R}^m$

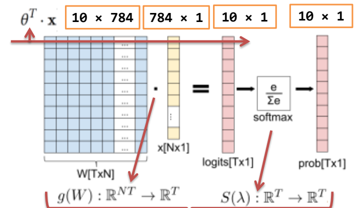
- We'll consider the outputs of *f* to be numbered from 1 to *m* as $f_1, f_2 \dots f_m$

The derivative of *f* at *a* is the *Jacobian matrix*:

$$Df(a) = \begin{bmatrix} D_1 f_1(a) & \cdots & D_n f_1(a) \\ \vdots & & \vdots \\ D_1 f_m(a) & \cdots & D_n f_m(a) \end{bmatrix}$$

$$\left.\begin{matrix} g : \mathbb{R}^n \to \mathbb{R}^m \\ f : \mathbb{R}^m \to \mathbb{R}^p \\ a \in \mathbb{R}^n \end{matrix}\right\} \xrightarrow{\text{Def.}} D(f \circ g)(a) = Df(g(a)) \cdot Dg(a)$$

The Jacobian of `softmax` Layer



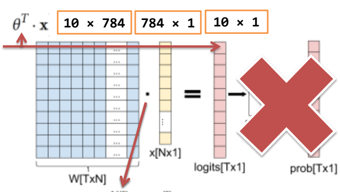
$$S(\mathbf{a}) : \begin{bmatrix} a_1 \\ a_2 \\ \cdots \\ a_T \end{bmatrix} \to \begin{bmatrix} S_1 \\ S_2 \\ \cdots \\ S_T \end{bmatrix}$$

g(W)

$$P(W) = S(g(W)) = (S \circ g)(W)$$

$$DP(W) = D(S \circ g)(W) = DS(g(W)) \cdot Dg(W)$$

Computing *Dg(W)* (I)



$$S(\mathbf{a}) : \begin{bmatrix} a_1 \\ a_2 \\ \cdots \\ a_T \end{bmatrix} \to \begin{bmatrix} S_1 \\ S_2 \\ \cdots \\ S_T \end{bmatrix}$$

g(W), $g_1 = a_1$
$D_1g_1 = dg_1/dW_{11}$
$D_2g_1 = dg_1/dW_{12}$

$g(W) : \mathbb{R}^{NT} \to \mathbb{R}^T$

$$Dg = \begin{bmatrix} D_1 g_1 & \cdots & D_{NT} g_1 \\ \vdots & \ddots & \vdots \\ D_1 g_T & \cdots & D_{NT} g_T \end{bmatrix}$$

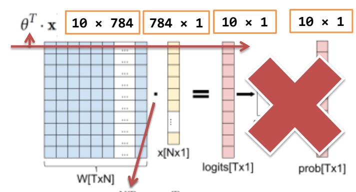
T rows and NT columns

$$g_1 = W_{11}x_1 + W_{12}x_2 + \cdots + W_{1N}x_N$$

$$D_1 g_1 = x_1 \quad D_2 g_1 = x_2 \quad \cdots$$

$$D_N g_1 = x_N \quad D_{N+1} g_1 = 0 \quad \cdots \quad D_{NT} g_1 = 0$$

Computing *Dg(W)* (II)



$$S(\mathbf{a}) : \begin{bmatrix} a_1 \\ a_2 \\ \cdots \\ a_T \end{bmatrix} \to \begin{bmatrix} S_1 \\ S_2 \\ \cdots \\ S_T \end{bmatrix}$$

g(W), $g_1 = a_1$
$D_1g_1 = dg_1/dW_{11}$
$D_2g_1 = dg_1/dW_{12}$
$D_1g_1 \dots D_{NT}g_1$

$g(W) : \mathbb{R}^{NT} \to \mathbb{R}^T$

$$Dg = \begin{bmatrix} D_1 g_1 & \cdots & D_{NT} g_1 \\ \vdots & \ddots & \vdots \\ D_1 g_T & \cdots & D_{NT} g_T \end{bmatrix} \to Dg = \begin{bmatrix} x_1 & x_2 & \cdots & x_N & \cdots & 0 & 0 & \cdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 & \cdots & x_1 & x_2 & \cdots & x_N \end{bmatrix}$$

T rows and NT columns

$$D_{ij} g_t = \frac{\partial (W_{t1}x_1 + W_{t2}x_2 + \cdots + W_{tN}x_N)}{\partial W_{ij}} = \begin{cases} x_j & i = t \\ 0 & i \neq t \end{cases}$$

Row t, Column (i – 1)N + j in matrix *Dg*

The Jacobian of `softmax` Layer

$$DP(W) = D(S \circ g)(W) = DS(g(W)) \cdot Dg(W)$$

D_kS_t (k=1…T; t=1…T)

$$DS = \begin{bmatrix} D_1 S_1 & \cdots & D_N S_1 \\ \vdots & \ddots & \vdots \\ D_1 S_N & \cdots & D_N S_N \end{bmatrix}$$

(N = T) T rows and T columns

$D_{ij}g_k$ (k=1…T)

$$Dg = \begin{bmatrix} x_1 & x_2 & \cdots & x_N & \cdots & 0 & 0 & \cdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 & \cdots & x_1 & x_2 & \cdots & x_N \end{bmatrix}$$

T rows and NT columns

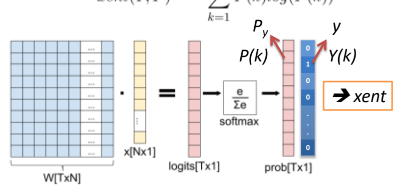
$$D_{ij} P_t = \sum_{k=1}^{T} D_k S_t \cdot D_{ij} g_k$$

The only *k* for which $D_{ij}g_k$ is nonzero is when i = k; then it's equal to x_j. Therefore:

$$D_{ij} P_t = D_i S_t x_j = S_t(\delta_{ti} - S_i) x_j$$

The Jacobian of Cross-Entropy Loss (I)

$$xent(Y, P) = -\sum_{k=1}^{T} Y(k) log(P(k))$$



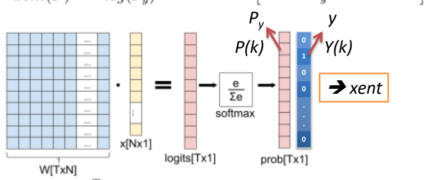
$$xent(P) = -log(P_y)$$

$$Dxent = \begin{bmatrix} D_1 xent & D_2 xent & \cdots & D_T xent \end{bmatrix}$$

$$Dxent = \begin{bmatrix} 0 & 0 & D_y xent & \cdots & 0 \end{bmatrix}$$

The Jacobian of Cross-Entropy Loss (II)

$$xent(P) = -log(P_y) \qquad Dxent = \begin{bmatrix} 0 & 0 & D_y xent & \cdots & 0 \end{bmatrix}$$



$$D_{ij} xent(W) = \sum_{k=1}^{T} D_k xent(P) \cdot D_{ij} P_k(W)$$

$$D_{ij} xent(W) = D_y xent(P) \cdot D_{ij} P_y(W)$$

By our definition, $P_y = S_y$

$$= -\frac{1}{P_y} \cdot S_y(\delta_{yi} - S_i)x_j \qquad \boxed{= (S_i - \delta_{yi})x_j}$$

Why So Complicated Proof?

The *advantage* of this approach is that it works exactly the same for more complex compositions of functions, where the "closed form" of the derivative for each element is much harder to compute otherwise.

Thank You for Your Time