Vectorization

Yu Wu, Ishan Patil

October 13, 2017

Exercises to be covered

We will implement some examples of image classification algorithms using a subset of the MNIST dataset

- logistic regression for just 0's and 1's
- softmax regression for all digits
- kNN for all digits

Key Takeaways

- Rule 0: Use built-in functions whenever possible
- Rule 1: Avoid using for loops (at least try really really hard)

Using built-in functions

- Most vector/ matrix operations have built-in function in numpy or Matlab (e.g dot product, matrix multiplication, log/exp of every element)
- Other functions could be implemented using combinations of these built-in functions

Two implementations of the sigmoid function

Version without using numpy functions:

```
def h1(theta, x):
    sum = 0.0
    for i in range(len(x)):
        sum -= theta[i] * x[i]
    return 1 / (1 + math.exp(sum))
```

Version with numpy functions:

```
def h2(theta, x):
    return 1 / (1 + np.exp(np.dot(theta, x)))
```

Logistic Regression

while not converged **do**

$\theta_j := \theta_j - \alpha \sum_{i=1}^{m}(h_\theta(x^i) - y^i)x_j^i$ for all $j = 1, 2, \cdots, n$

end while

n is the number of features (784), m is the number of training samples

First implementation of Gradient Descent Step

for each sample x_i **do**
  calculate $h_\theta(x^i) - y^i$
end for
for each index j **do**
  sum $= 0$
  for each sample x^i **do**
    sum $+=$ $(h_\theta(x^i) - y^j)x^i_j$
  end for
  θ_j -= α * sum
end for

Better implementation

Remember our update rule: $\theta_j := \theta_j - \alpha \sum_{i=1}^{m}(h_\theta(x^i) - y^i)x_j^i$
If we can simultaneously get all $h_\theta(x_1), h_\theta(x_2), \cdots, h_\theta(x_m)$ as a

$m \times 1$ vector h, then

$$X = \begin{bmatrix} \begin{bmatrix} x_1^1 \\ x_1^2 \\ \vdots \\ x_1^m \end{bmatrix} & \begin{bmatrix} x_2^1 \\ x_2^2 \\ \vdots \\ x_2^m \end{bmatrix} & \cdots & \begin{bmatrix} x_n^1 \\ x_n^2 \\ \vdots \\ x_n^m \end{bmatrix} \end{bmatrix} = \begin{bmatrix} x_1 & x_2 & \cdots & x_n \end{bmatrix}$$

$$h - y = \begin{bmatrix} h_\theta(x_1) - y^1 \\ h_\theta(x_2) - y^2 \\ \vdots \\ h_\theta(x_m) - y^m \end{bmatrix} = z, \sum_{i=1}^{m}(h_\theta(x_i) - y^i)x_j^i = \sum_{i=1}^{m} z_i x_j^i = z \cdot x_j$$

How do we get h?

- `np.exp()` could perform exponential operation on a vector element-wise!

$$X = \begin{bmatrix} (x^1)^T \\ (x^2)^T \\ \vdots \\ (x^m)^T \end{bmatrix}, X\theta = \begin{bmatrix} (x^1)^T\theta \\ (x^2)^T\theta \\ \vdots \\ (x^m)^T\theta \end{bmatrix} = \begin{bmatrix} \theta^T x^1 \\ \theta^T x^2 \\ \vdots \\ \theta^T x^m \end{bmatrix}$$

$$1 + 1/\text{np.exp}(-X\theta) = \begin{bmatrix} \frac{1}{1+\exp(-\theta^T x^1)} \\ \vdots \\ \frac{1}{1+\exp(-\theta^T x^m)} \end{bmatrix}$$

Improved version of Gradient descent step

Vectorized sigmoid function:

```python
def h_vec(theta, X):
    return 1 / (1 + np.exp(-np.matmul(X, theta)))
```

new gradient descent step:

---

calculate $z = h - y$
for each index j **do**
  θ_j -= α * $\texttt{np.dot}(z, x_j)$
end for

We can do better!

We can calculate all the update amount at once!

$$\Delta\theta_1 = \alpha z^T x_1, \Delta\theta_2 = \alpha z^T x_2, \cdots$$

So

$$\Delta\theta = [\Delta\theta_1, \Delta\theta_2, \cdots, \Delta\theta_n] = \alpha z^T [x_1, x_2, \cdots, x_n] = \alpha z^T X$$

More vectorized version

new gradient descent step:

$$\theta \mathrel{-}= \alpha(z^T X)^T$$

Python implementation:

```
def GD (theta, X_train, y_train, alpha):
        theta -= alpha * np.squeeze(np.matmul(
                np.reshape(h_all(theta, X_train) - y_train, [1, -1]), X_train))
```

Softmax regression

θ is no longer a vector, it is a $n \times c$ matrix, where c is the number of class (=10)

$$\theta = \begin{bmatrix} \vec{\theta_1} & \vec{\theta_2} & \cdots & \vec{\theta_c} \end{bmatrix}, \vec{\theta_k} \in \mathbb{R}^n, k = 1, 2, \cdots, c$$

y is also a matrix of the labels encoded using one-hot encoding:

$$y^i = 3 \rightarrow y^i = [0, 0, 0, 1, 0, 0, 0, 0, 0, 0]$$

$h_\theta(x^i)$ is now the softmax function:

$$h_\theta(x^i) = \begin{bmatrix} \frac{\exp(\vec{\theta_1}^T x^i)}{\sum_{k=1}^{c} \exp(\vec{\theta_k}^T x^i)}, & \frac{\exp(\vec{\theta_2}^T x^i)}{\sum_{k=1}^{c} \exp(\vec{\theta_k}^T x^i)}, & \cdots & \frac{\exp(\vec{\theta_c}^T x^i)}{\sum_{k=1}^{c} \exp(\vec{\theta_k}^T x^i)} \end{bmatrix}$$

Implementing the softmax function, part 1

In practice, $\vec{\theta_c}^T x^i$ could be pretty big, so $\exp(\vec{\theta_c}^T x^i)$ could cause overflow issues. One way to go around this problem is to subtract a constant a_i from each dot product, and the softmax function will still remain the same:

$$\frac{\exp(\vec{\theta_k}^T x^i - a_i)}{\sum_{k=1}^{c} \exp(\vec{\theta_k}^T x^i - a_i)} = \frac{\exp(-a_i) \cdot \exp(\vec{\theta_k}^T x^i)}{\exp(-a_i) \cdot \sum_{k=1}^{c} \exp(\vec{\theta_k}^T x^i)}$$

Often we set $a_i = \max_k \{\vec{\theta_k}^T x^i\}$. So the softmax function we will implement is essentially

$$h_\theta(x^i) = \left[\frac{\exp(\vec{\theta_1}^T x^i - \max_k\{\vec{\theta_k}^T x^i\})}{\sum_{k=1}^{c} \exp(\vec{\theta_k}^T x^i - \max_k\{\vec{\theta_k}^T x^i\})}, \quad \frac{\exp(\vec{\theta_2}^T x^i - \max_k\{\vec{\theta_k}^T x^i\})}{\sum_{k=1}^{c} \exp(\vec{\theta_k}^T x^i - \max_k\{\vec{\theta_k}^T x^i\})}, \quad \cdots \right]$$

Implementing the softmax function, part 2

Pseudo code:

---

for every sample x^i **do**
  temp $= [\theta_1^T x^i, \theta_2^T x^i, \cdots]$
  $a_i = \max_k \{\theta_k^T x^i\}$
  temp1 $=$ exp(temp - a_i)
  $h_\theta(x^i) =$ temp1 / sum(temp1)
end for

Can we compute h for all samples at once?

we can compute all $\vec{\theta_k}^T x^i$ again with matrix multiplication:

$$X = \begin{bmatrix} (x^1)^T \\ (x^2)^T \\ \dots \\ (x^m)^T \end{bmatrix}, \theta = \begin{bmatrix} \vec{\theta_1} & \vec{\theta_2} & \cdots & \vec{\theta_c} \end{bmatrix}, X\theta = \begin{bmatrix} \vec{\theta_1}^T x^1 & \vec{\theta_2}^T x_1 & \cdots \\ \vec{\theta_1}^T x^2 & \vec{\theta_2}^T x_2 & \cdots \\ \vdots & \vdots & \end{bmatrix}$$

However, we need to subtract a different constant a_i for each row.
How do we deal with that?

Tiling and broadcasting

We could get vector $a = [a_1, a_2, \cdots, a_m]^T$ by taking the maximum of every row using `np.amax(`$X\theta$`, axis=1)` we could get out desired result by tiling a c times so we have a compatible matrix:

$$A = \underbrace{\begin{bmatrix} a & a & \cdots & a \end{bmatrix}}_{c \text{ times}}, \begin{bmatrix} \vec{\theta_1}^T x^1 - a_1 & \vec{\theta_2}^T x_1 - a_1 & \cdots \\ \vec{\theta_1}^T x^2 - a_2 & \vec{\theta_2}^T x_2 - a_2 & \cdots \\ \vdots & \vdots & \end{bmatrix} = X\theta - A$$

Tiling in Matlab could be done using the `rempat` function, but in numpy this is done automatically if the dimensions match correctly. This automatic tiling behaviour is called broadcasting.

Putting everything together

The last piece of puzzle we need to solve is to compute the row sums of `np.exp`$(X\theta - A)$ and divide each row with the corresponding sum. This could again be done using `np.sum` with the attribute `axis=1` and tiling/broadcasting.
Putting everything together, the pseudo-code is

```
temp = Xθ
a = np.amax(temp,axis=1)
get A by tiling a
temp1 = np.exp(Xθ − A)
get row_sums by tiling np.sum(temp1, axis=1)
return h = temp1 / row_sums
```

Gradient descent step, first version

Our softmax function returns a matrix h with dimension $m \times c$. So $h - y$ is again a matrix h with dimension $m \times c$. From our exercise with logistic regression we know how to update an entire vector. Applying that here gives us:

---

for every label k **do**
  θ_k -= $\alpha(((h - y)_k)^T X)^T$
end for

Gradient descent step, second version

The algorithm in the previous page is the same as

$$\theta \mathrel{-}= \alpha((h - y)^T X)^T$$

K Nearest Neighbor Algorithm

X_train ($M \times D$) Y_train ($M \times 1$)
X_test ($N \times D$) Y_test ($N \times 1$)

- At training time, just remember our training data (X_train, Y_train)
- At test time, assign the class/label most common among its K closest neighbors by taking their majority vote.
- Naive algorithm, but degree of vectorization in code can affect performance significantly.

Broad Idea

- Compute **Dist** ($N \times M$) where **Dist[i,j]** is the euclidean distance between i^{th} test example and j^{th} training example.
- Compute **DistSorted** by sorting the elements in each row of Dist and assigning to each row, the indices (into X_train) of the sorted elements.
- Compute **KClosest** by grabbing only the first K columns of DistSorted.
- Compute **KClosestLabels** by getting the output labels corresponding to each of the training example indices in **KClosest**.
- For each row of **KClosestLabels** (each test example), assign the output label with highest frequency among the K labels in that row.

Computation of Dist

Naive way - Using 2 for loops

---

for each i in 1:N **do**
  for each j in 1:M **do**
    $Dist[i,j] = \sqrt{\sum_{k=1}^{D} (X_{test}[i,k] - X_{train}[j,k])^2}$
  end for
end for

N is the test examples, M is the number of training samples, D is the number of features.

Computation of Dist (Cont.)

Somewhat better - Using 1 for loop

for each i in 1:N **do**

$X_{testR} = repeat\ X_{test}[i,:]\ vertically\ M\ times$

$Dist[i,:] = \sqrt{\sum_{k=1}^{D} (X_{testR}[:,k] - X_{train}[:,k])^2}$

end for

N is the test examples, M is the number of training samples, D is the number of features.

Computation of Dist (Cont.)

Fully Vectorized Implementation

$$
\begin{aligned}
&X_{testSqr} = \textstyle\sum_{k=1}^{D} (X_{test}[:, k])^2 \\
&X_{trainSqr} = \textstyle\sum_{k=1}^{D} (X_{train}[:, k])^2 \\
&X_{testSqrR} = repeat\ X_{testSqr}\ horizontally\ M\ times \\
&X_{trainSqrR} = repeat\ X_{trainSqr}\ vertically\ N\ times \\
&X_{cross} = X_{test} \times X_{train}^{T} \\
&Dist = \sqrt(X_{testSqrR} + X_{trainSqrR} - 2 * X_{cross})
\end{aligned}
$$

N is the test examples, M is the number of training samples, D is the number of features.

Main Takeaway

- This method of computing distances between each vector (row/column) of two matrices is a thing that comes up quite often, not just in kNN algorithm.
- RBF kernel computation for SVM (element-wise operation on each of the values of **Dist**) is another example.
- Readily available functions to do this -
  - **MATLAB** - `pdist2`
  - **Python** - `scipy.spatial.distance.pdist`
- Main idea of tiling and broadcasting is what we want to emphasize more.