Softmax Regression in Python: Multi-class Classification

Machine Learning From Scratch: Part 7

Suraj Verma Apr 26 · 9 min read ★

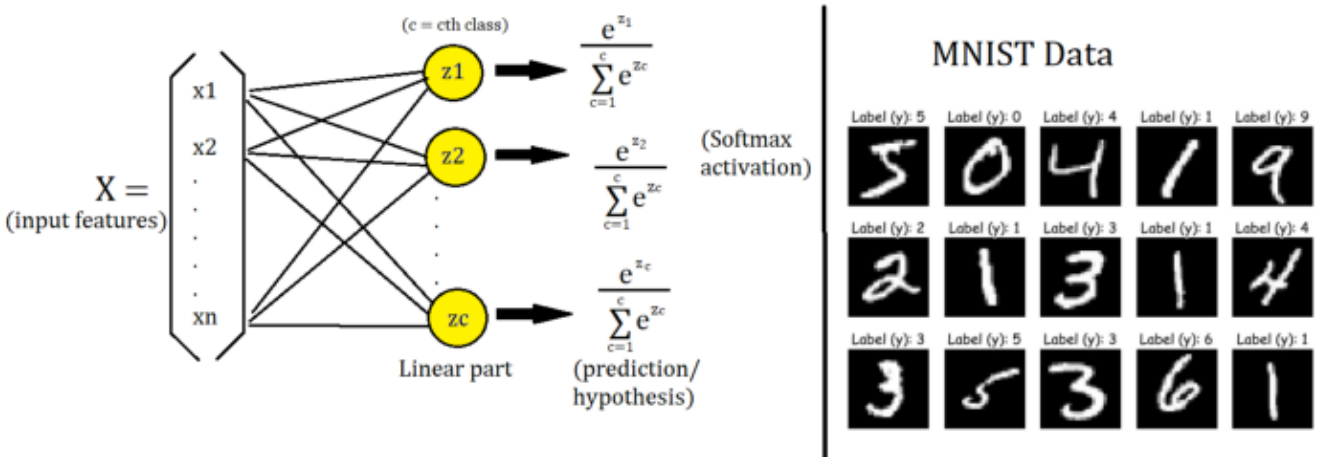


Image by Author

In this article, we are going to look at the Softmax Regression which is used for multi-class classification problems, and implement it on the MNIST hand-written digit recognition dataset.

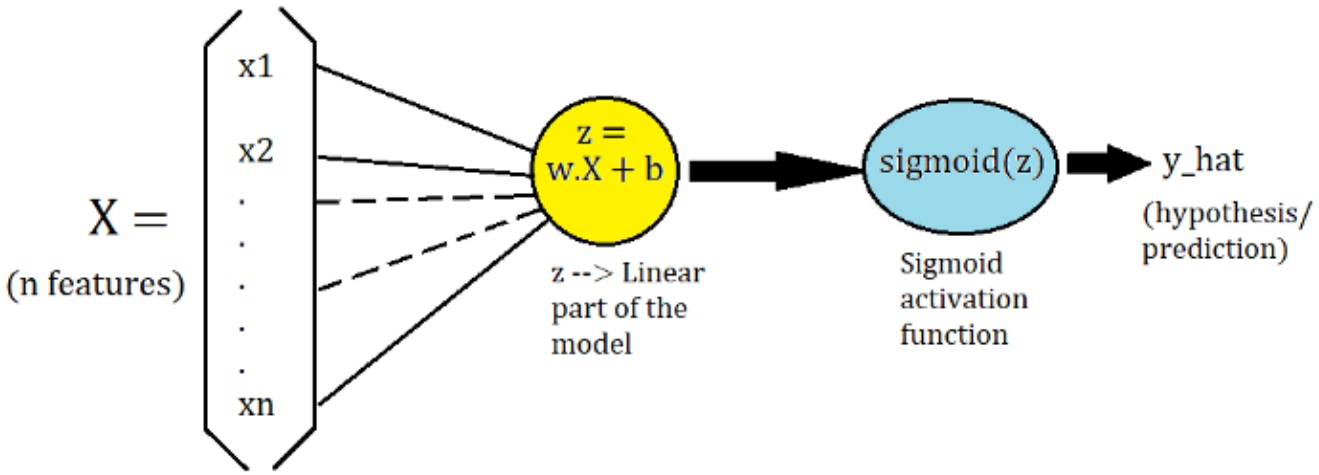
First, we will build on **Logistic Regression** to understand the **Softmax function**, then we will look at the **Cross-entropy loss**, **one-hot encoding**, and code it alongside.

Finally, we will code the **training** function(`fit`) and see our **accuracy**. We will also **plot** our predictions.

We will do it all using Python NumPy and Matplotlib. We will see how we get an ***accuracy of 91%*** in both training and test set.

Logistic Regression From Scratch in Python
Machine Learning From Scratch: Part 5
towardsdatascience.com

Logistic Regression Recap



Logistic Regression model; Image by Author

As we can see above, in the logistic regression model we take a vector `x` (which represents only a single example out of `m`) of size `n` (features) and take a dot product with the `weights` and add a `bias` . We will call it `z` (linear part) which is `w.X + b` . After that, we apply the activation function which is sigmoid for logistic regression to calculate `y_hat` .

$$y\_hat = sigmoid(z) = sigmoid(w.X + b)$$

> *Every edge going from* `X` *to the linear part represents a* `weight` *and every circle of linear part has a* `bias` .

Logistic Regression is used for binary classification which means there are 2 classes(**0** or **1**) and because of the sigmoid function we get an output(y_hat) between 0 and 1. We interpret this output(`y_hat`) of a logistic model as a probability of `y` being 1, then the probability of `y` being 0 becomes `(1-y_hat)` .

A question for you — What is the shape of `w` in the above image?

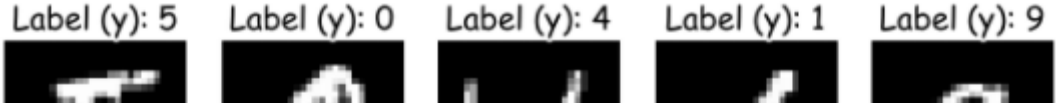
Softmax Regression

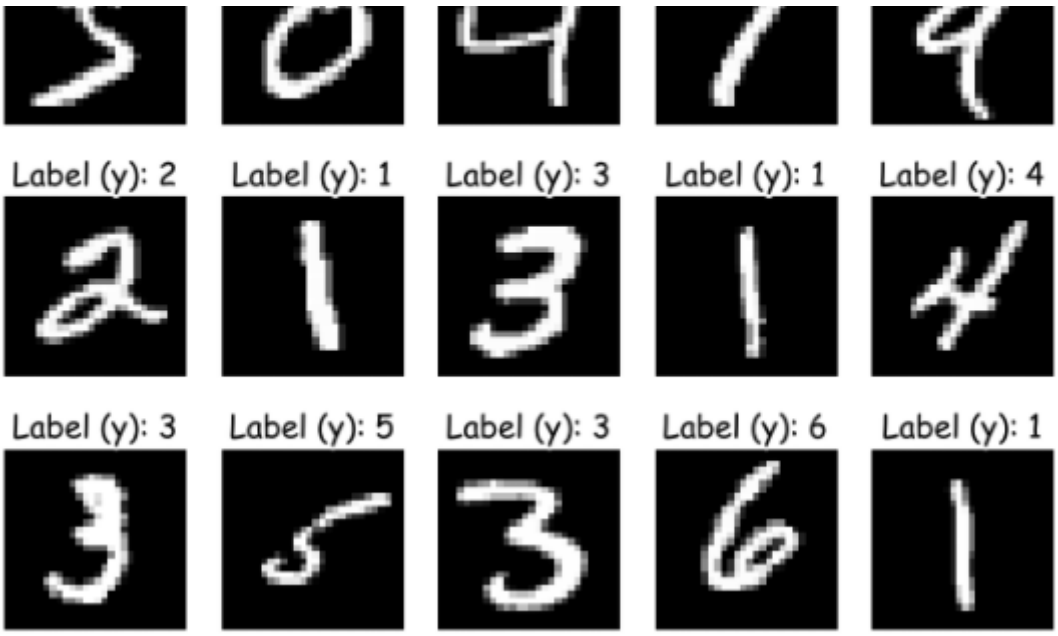
Now, we set a goal for us — **To identify which digit is in the image.**

We will use the MNIST hand-written dataset as a motivating example to understand Softmax Regression. It has **10 classes** each representing a digit from **0** to **9**. Let us look at the dataset first.

Loading MNIST dataset from `keras.datasets` and plotting. Also, splitting the training and test set. It has 60,000 examples in the training set and 10,000 examples in the test set.

```
#Loading
from keras.datasets import mnist

(train_X, train_y), (test_X, test_y) = mnist.load_data()

#Plotting

fig = plt.figure(figsize=(10,7))

for i in range(15):
    ax = fig.add_subplot(3, 5, i+1)
    ax.imshow(train_X[i], cmap=plt.get_cmap('gray'))
    ax.set_title('Label (y): {y}'.format(y=train_y[i]))
    plt.axis('off')
```





MNIST data; Image by Author

Our data has images of digits and labels representing which digit it is. Each image is **grayscale** and is of size **28x28 pixels**.

> *First question — **How do we take an image as an input?***

Every image can be represented as a 3-D matrix. If we have a color image of size **28x28**, we have **28 x 28 x 3** numbers to represent that image(**3** for the RGB(Red-Green-Blue) channel).

But we have grayscale images which means every image has only 1 channel. So, every image is just **28x28 pixels**. To input an image into the model we will flatten this **28x28** into a vector of length **784**(28*28) and these **784** numbers which just pixels represent the `n` features for every example(image) in the dataset.

```
X_train = train_X.reshape(60000,28*28)
X_test = test_X.reshape(10000,28*28)
```

> *X_train is a matrix with 60,000 rows and 784 columns.*
>
> *m = 60,000; n = 784*

One-hot Encoding

Now, our `y` is a vector that looks like this —

```
train_y
>> array([5, 0, 4, ..., 5, 6, 8], dtype=uint8)
```

It is a NumPy array with labels. We cannot use this for our model, we have to somehow modify it into zeros and ones which is what we call one-hot encoding. We want our `y` to look like this (a matrix of size 60,000 x 10)—

	0	1	2	3	4	5	6	7	8	9
1	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0
2	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
3	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0
4	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
5	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0
...	...	...	...	...	...	...	...	...	...	...
59996	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0
59997	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0
59998	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0
59999	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0
60000	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0

One-Hot Representation of y; Image by Author

The rows represent i'th example and i'th column tells us the label. For example, for the first example, there is a 1 where the column name is 5 and the rest are zeros. So, the label for the first example is 5 and similarly for others. For every example, there will be only one and only one column with a 1.0 and rest will be zeros.

Let's code a function to one-hot encode our labels —

> c = *Number of classes*

```
def one_hot(y, c):

    # y--> label/ground truth.
    # c--> Number of classes.

    # A zero matrix of size (m, c)
    y_hot = np.zeros((len(y), c))

    # Putting 1 for column where the label is,
    # Using multidimensional indexing.
    y_hot[np.arange(len(y)), y] = 1

    return y_hot
```

> *Reference — Multi-dimensional indexing in NumPy*

Softmax Function

While doing multi-class classification using Softmax Regression, we have a **constraint** that our model will predict only one class of `c` classes. For our data, it means that the model will predict only one of the digits(from 0 to 9) to be in the image.

We interpreted the output of the logistic model as a probability. Similarly, we want to interpret the output of a multi-class classification model as a **probability distribution**. So, we want our model to output a vector of size `c` with each value in the vector representing the probability of each class. In other words, the **c'th** value in the vector represents the probability of our prediction being the **c'th** class. *Since they are all probabilities, their sum will be equal to 1.*

To be in accordance with the above assumptions, we use a softmax function.

The softmax for the **c'th class** is defined as —

$$\frac{e^{z_c}}{\sum_{c=1}^{C} e^{z_c}}$$

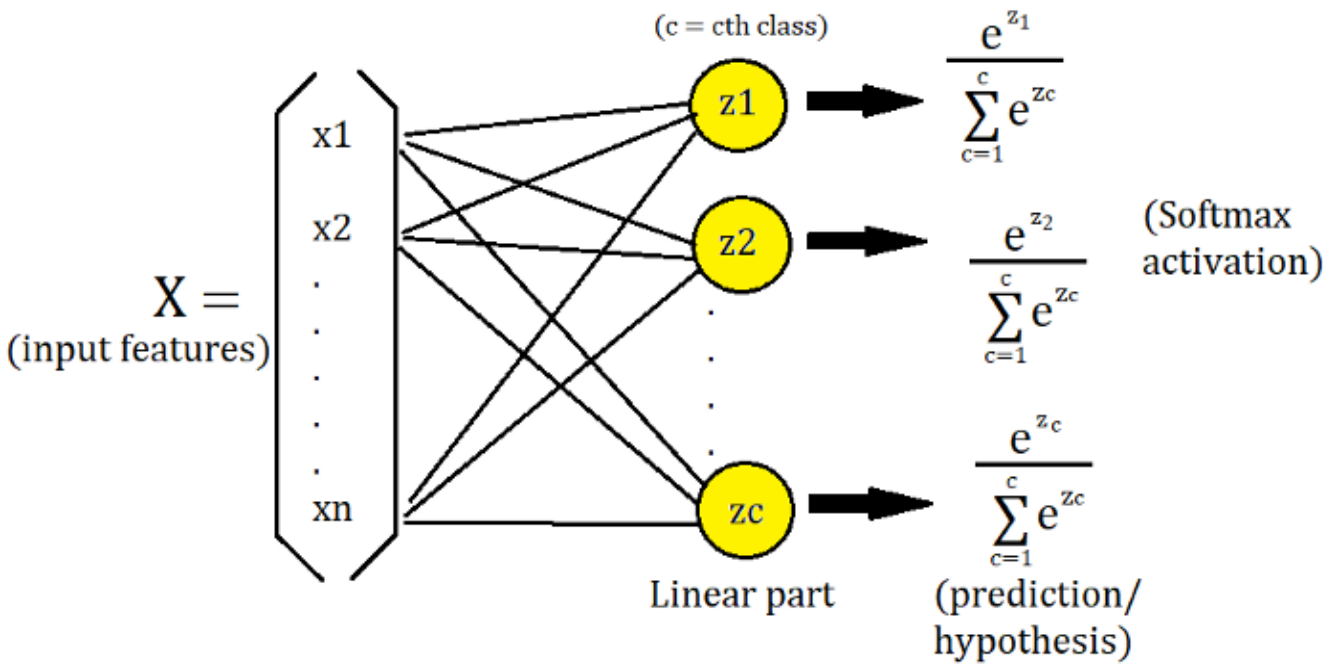
Softmax function; Image by Author

where, `z` is the linear part. For example, `z1 = w1.x + b1` and similarly for others.

$$y\_hat = softmax(w.X + b)$$

> *`c` (number of classes) = 10 for our data.*

Let's try to understand the Softmax function and Softmax Regression with the help of the below model diagram.



Softmax Regression Model; Image by Author

1. First, we have flattened our **28x28** image into a vector of length **784**, represented by `x` in the above image.
2. Second, we calculate the linear part for each class → `zc = wc.X + bc` where, `zc` is the linear part of the **c'th** class and `wc` is the set of weights of the **c'th** class. `bc` is the bias for the c'th class.
3. Third, we calculate the softmax of `zc` for each class using the above formula. Combining all the classes, we get a vector of size `c` and their sum is equal to 1. And whichever class has the highest value(probability) will be our prediction.

> ***Note that every class has a different set of weights and a different bias.***
>
> ***Also note that the above diagram shows the `x` as only one example.***

Every `zc` is a (1,1) matrix. But since we have `m` examples, we will represent `zc` (`z` for the c'th class) as a vector of size `m`. Now, to combine all the **`z`** **'s** classes, we will stack them side by side which will give us a matrix of size (`m`, `c`).

Similarly, for every class, the set of `weights` is a vector of size `n`. So, to combine all the classes, we will stack them side by side which will give us a matrix of size (`n`, `c`). Also, we have one bias for every class, the combined biases will be a vector of size `c`.

Combining all the classes for the softmax gives a vector of size `c`. And combining all the `m` examples give a matrix of size (`m`, `c`).

Shapes —

1. **X →(m,n)**
2. **y →(m,c) [one hot encoded]**
3. **w →(n,c)**
4. **b →vector of size c**

Let's write the code for the softmax function. See comments(#).

```
def softmax(z):

    # z--> linear part.

    # subtracting the max of z for numerical stability.
    exp = np.exp(z - np.max(z))

    # Calculating softmax for all examples.
    for i in range(len(z)):
        exp[i] /= np.sum(exp[i])

    return exp
```

> *Refrence — Numerically stable softmax*
>
> *Refrence — Derivative of Softmax*

Cross-Entropy Loss

For every parametric machine learning algorithm, we need a loss function, which we want to minimize (find the global minimum of) to determine the optimal parameters(`w` and `b`) which will help us make the best predictions.

For softmax regression, we use the cross-entropy(CE) loss —

$$L = -\sum_{c=1}^{c} y_c \log(\hat{y}_c)$$

CE loss; Image by Author

> ***Refrence for how to calculate derivative of loss.***
>
> *Refrence — Derivative of Cross Entropy Loss with Softmax*
>
> *Refrence — Derivative of Softmax loss function*

In code, the loss looks like this —

```
loss = -np.mean(np.log(y_hat[np.arange(len(y)), y]))
```

Again using multidimensional indexing — Multi-dimensional indexing in NumPy

> *Note that `y` is not one-hot encoded in the loss function.*

Training

1. Initialize parameters — `w` and `b` .
2. Find optimal `w` and `b` using Gradient Descent.
3. Use `softmax(w.X + b)` to predict.

```
def fit(X, y, lr, c, epochs):

    # X --> Input.
    # y --> true/target value.
    # lr --> Learning rate.
    # c --> Number of classes.
    # epochs --> Number of iterations.


    # m-> number of training examples
    # n-> number of features
    m, n = X.shape

    # Initializing weights and bias randomly.
    w = np.random.random((n, c))
    b = np.random.random(c)

    # Empty list to store losses.
    losses = []

    # Training loop.
    for epoch in range(epochs):

        # Calculating hypothesis/prediction.
        z = X@w + b
        y_hat = softmax(z)

        # One-hot encoding y.
        y_hot = one_hot(y, c)

        # Calculating the gradient of loss w.r.t w and b.
        w_grad = (1/m)*np.dot(X.T, (y_hat - y_hot))
        b_grad = (1/m)*np.sum(y_hat - y_hot)

        # Updating the parameters.
        w = w - lr*w_grad
        b = b - lr*b_grad

        # Calculating loss and appending it in the list.
        loss = -np.mean(np.log(y_hat[np.arange(len(y)), y]))
        losses.append(loss)

        # Printing out the loss at every 100th iteration.
        if epoch%100==0:
            print('Epoch {epoch}==> Loss = {loss}'
                  .format(epoch=epoch, loss=loss))

    return w, b, losses
```

Now let's train our MNIST data —

See comments(#).

```
# Flattening the image.
X_train = train_X.reshape(60000,28*28)

# Normalizing.
X_train = X_train/255

# Training
w, b, l = fit(X_train, train_y, lr=0.9, c=10, epochs=1000)
>> Epoch 0==> Loss = 4.765269749988195
Epoch 100==> Loss = 0.41732772667703605
Epoch 200==> Loss = 0.36146764856576774
Epoch 300==> Loss = 0.3371995534802398
Epoch 400==> Loss = 0.32295154931574305
Epoch 500==> Loss = 0.31331228168677683
Epoch 600==> Loss = 0.3062124554422963
Epoch 700==> Loss = 0.3006810669534496
Epoch 800==> Loss = 0.29619875396394774
Epoch 900==> Loss = 0.29246033264255616
```

> *We can see that the loss goes down after every iteration, which means we are doing good.*

Predicting

See comments(#).

```
def predict(X, w, b):

    # X --> Input.
    # w --> weights.
    # b --> bias.

    # Predicting
    z = X@w + b
    y_hat = softmax(z)

    # Returning the class with highest probability.
    return np.argmax(y_hat, axis=1)
```

Measuring Accuracy

```
def accuracy(y, y_hat):
    return np.sum(y==y_hat)/len(y)
```

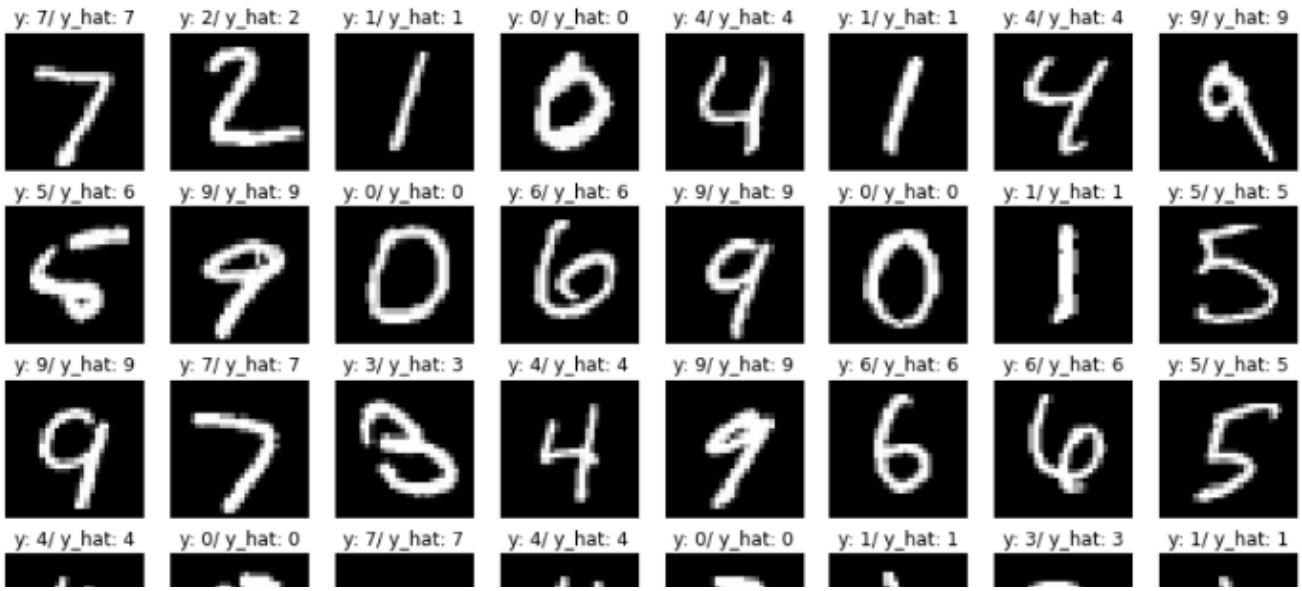
Let's calculate the accuracy of our model on the training and test set.

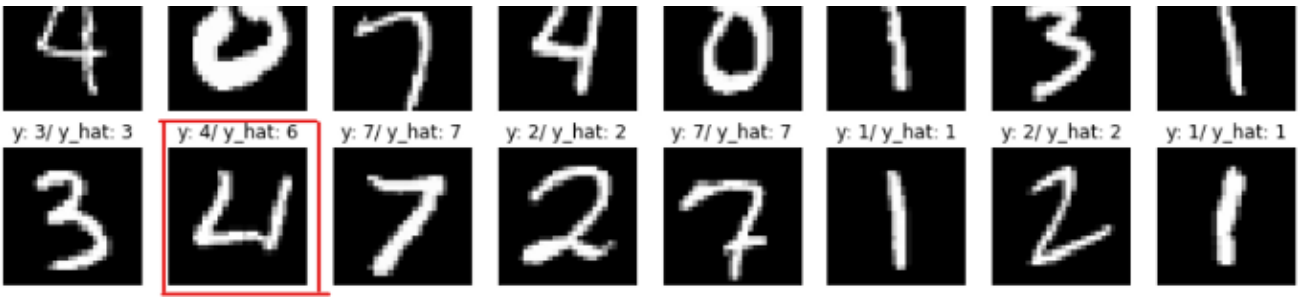
```
# Accuracy for training set.
train_preds = predict(X_train, w, b)
accuracy(train_y, train_preds)
>> 0.9187666

# Accuracy for test set.

# Flattening and normalizing.
X_test = test_X.reshape(10000,28*28)
X_test = X_test/255

test_preds = predict(X_test, w, b)
accuracy(test_y, test_preds)
>> 0.9173
```

> *Our model has an accuracy of 91.8% on the training set and an accuracy of 91.7% on the test set.*
>
> *Not Bad.*

Plotting Predictions

```
fig = plt.figure(figsize=(15,10))

for i in range(40):
    ax = fig.add_subplot(5, 8, i+1)
    ax.imshow(test_X[i], cmap=plt.get_cmap('gray'))

    ax.set_title('y: {y}/ y_hat: {y_hat}'
                 .format(y=test_y[i], y_hat=test_preds)
    plt.axis('off')
```





Predictions; Image by Author

The above figure shows the predictions by the model for 40 examples in the training set. We can see that some of them are wrongly predicted, like the one encircled in red in the above picture.

Thanks for reading. For questions, comments, concerns, talk to be in the response section. More ML from scratch is coming soon.

Check out the Machine Learning from scratch series —

- Part 1: **Linear Regression from scratch in Python**
- Part 2: **Locally Weighted Linear Regression in Python**
- Part 3: **Normal Equation Using Python: The Closed-Form Solution for Linear Regression**
- Part 4: **Polynomial Regression From Scratch in Python**
- Part 5: **Logistic Regression From Scratch in Python**
- Part 6: **Implementing the Perceptron Algorithm in Python**

Data Science Machine Learning Python Artificial Intelligence Neural Networks