Andrew Ng Course Note 7

Shu Wang

May 31 2023

1 Implementation

Example code:

```
import tensorflow as tf
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense
 model = Sequential([
 Dense(units = 25,activation = 'sigmoid')
 Dense(units = 15,activation = 'sigmoid')
 Dense(units = 1,activation = 'sigmoid')
 ])
```

Then compile the model:

```
from tensorflow.keras.losses import
BinaryCrossentropy
 model.compile(loss = BinaryCrossentropy())


 model.fit(X,Y,epochs = 100)
```

In logistic regression:

First step: specify how to compute output given input x and parameters w, b (define model)

```
z = np.dot(w,x)+b
f_x = 1/(1_np.exp(-z))


loss = -y*np.log(f_x) - (1-y)* np.log(1-f_x)
```

Second step: specify cost and loss

In mathematical representation: $L(f_{\vec{w},b}(\vec{x}), y)$

$J(\vec{w}, b) = \frac{1}{m} \sum_{i=1}^{m} L(f_{\vec{w},b}(x^{(i)}), y^{(i)})$

Third step:

Train on data to minimize $J(\overrightarrow{w}, b)$

Similarly, in neural network:

Step 1:

```
import tensorflow as tf
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense
 model = Sequential([
 Dense(units = 25,activation = 'sigmoid')
 Dense(units = 15,activation = 'sigmoid')
 Dense(units = 1,activation = 'sigmoid')
 ])
```

Step 2: Compile the model:

```
from tensorflow.keras.losses import
BinaryCrossentropy
 model.compile(loss = BinaryCrossentropy())
```

Step 3:

```
 model.fit(X,Y,epochs = 100)
```

1.1 Cost and loss function

In binary classification:

$L(f_{\overrightarrow{w},b}(\overrightarrow{x}^{(i)}, y^{(i)}) = -y^{(i)}log(f_{\overrightarrow{w},b}(\overrightarrow{x}^{(i)}, y^{(i)}) - (1 - y^{(i)})log(1 - f_{\overrightarrow{w},b}(\overrightarrow{x}^{(i)}))$

Also called binary cross entropy.

In regression:

Mean Squared Error

Minimize the cost function using gradient descent.

$w_j^{[l]} = w_j^{[l]} - \alpha * \frac{\partial}{\partial w_j} J(\overrightarrow{w}, b)$

$b_j^{[l]} = b_j^{[l]} - \alpha * \frac{\partial}{\partial b} J(\overrightarrow{w}, b)$

Number of epochs: define the number of iterations.

1.2 Activation function

Different activation functions:

ReLu(Rectified Linear Unit) function:

$g(z) = max(0, z)$

Linear activation function:

$g(z) = z$

Choosing the activation function for the output layer usually depends on what is the label y you are trying to predict.

In hidden layers: choosing ReLu.

ReLu is faster to compute, ReLu goes flat only in one part of the graph, while the sigmoid function on both sides of the graph. If using gradient descent, it will be slow in flatten areas.

```
import tensorflow as tf
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense
 model = Sequential([
 Dense(units = 25,activation = 'relu')
 Dense(units = 15,activation = 'relu')
 Dense(units = 1,activation = 'sigmoid')
 ])
```

If we use linear activation function, then the problem would be no different than linear regression.

A simple example:

if we have one feature with activation function $g(z) = z$,then:

In first hidden layer:

$a^{[1]} = w_1^{[1]}x + b_1^{[1]}$

$a^{[2]} = w_1^{[2]}x + b_1^{[2]} = w_1^{[2]} * (w_1^{[1]}x + b_1^{[1]}) + b_1^{[2]}$

$\overrightarrow{a}^{[2]} = (\overrightarrow{w_1}^{[2]} * \overrightarrow{w_1}^{[1]})x + w_1^{[2]}b_1^{[1]} + b_1^{[2]}$

$\overrightarrow{a}^{[2]}$ can be written as a new linear function.

This doesn't let the neural network learn anything new.

2 Multiclass Classification

target y can take on more than two possible values.

Logistic Regression:

$z = \overrightarrow{w} \cdot \overrightarrow{x} + b$

$a = g(z) = \frac{1}{1+e^{-z}} = P(y = 1|\overrightarrow{x})$

Softmax Regression:

$z_1 = \overrightarrow{w_1} \cdot \overrightarrow{x} + b$

$z_2 = \overrightarrow{w_2} \cdot \overrightarrow{x} + b$

$z_3 = \overrightarrow{w_3} \cdot \overrightarrow{x} + b$

$z_4 = \overrightarrow{w_4} \cdot \overrightarrow{x} + b$

$a_1 = \frac{e^{z_1}}{e^{z_1}+e^{z_2}+e^{z_3}+e^{z_4}} = P(y=1|\overrightarrow{x})$

$a_2 = \frac{e^{z_2}}{e^{z_1}+e^{z_2}+e^{z_3}+e^{z_4}} = P(y=2|\overrightarrow{x})$

$a_3 = \frac{e^{z_3}}{e^{z_1}+e^{z_2}+e^{z_3}+e^{z_4}} = P(y=3|\overrightarrow{x})$

$a_4 = \frac{e^{z_4}}{e^{z_1}+e^{z_2}+e^{z_3}+e^{z_4}} = P(y=4|\overrightarrow{x})$

General Case:

$z_j = \overrightarrow{w_j} \cdot \overrightarrow{x} + b$

$a_j = \frac{e^{z_j}}{\sum_{k=1}^{N} e^{z_k}} = P(y=j|\overrightarrow{x})$

Logistic regression cost:

$a_1 = g(z) = \frac{1}{1+e^{-z}} = P(y=1|\overrightarrow{x})$

$a_2 = 1 - a_1$

$L(f_{\overrightarrow{w},b}(\overrightarrow{x}^{(i)}, y^{(i)}) = -y^{(i)} log(f_{\overrightarrow{w},b}(\overrightarrow{x}^{(i)}, y^{(i)}) - (1-y^{(i)}) log(1 - f_{\overrightarrow{w},b}(\overrightarrow{x}^{(i)}))$

Rewrite:

$loss = -y log(a_1) - (1-y) log(1-a_1)$

Softmax regresison:

$\text{loss}(\text{a}_1, a_2, ...a_n, y) =$

$$\begin{cases} \text{-log}(\text{a}_1) & \text{if y}^{(i)} = 1 \\ \text{-log}(\text{a}_2) & \text{if y}^{(i)} = 2 \\ ...\text{-log}(\text{a}_n) & \text{if y}^{(i)} = N \end{cases}$$

(1)

If a_j is close to 1, which means the model predicts this label with high probability, then the crossentropy loss would be close to 0.

Note that you are using only one value in this cross-entropy based on y.

Sparse categorical: y only take on one of these values.

Numerical round off errors:

Option 1:

$x = \frac{2}{10000}$

Option 2:

$x = (1 + \frac{1}{10000}) + (1 - \frac{1}{10000})$

More numerically accurate implementation of logistic loss:

Insisting on computing as an intermediate value.

```
model.compile(loss = BinaryCrossEntropy(from_logits =True))
```

Usage: set the output layer to just use a linear activation function and it puts both the activation function as well as the cross entropy loss into the specification of this loss function here.

Softmax regression:

$(a_1, ...a_10) = g(z_1, z_2, ...z_10)$

$$\text{L}(\overrightarrow{a}, y) = \begin{cases} \text{-log}(\text{a}_1) & \text{if y}^{(i)} = 1 \\ \text{-log}(\text{a}_2) & \text{if y}^{(i)} = 2 \\ ...\text{-log}(\text{a}_n) & \text{if y}^{(i)} = N \end{cases} \tag{2}$$

More accurate:

plug in equation of a_1 into z_1

Multi-label classification:

Several different labels as output.

Target output y is a vector.

Adam algorithm:

make α bigger.

Adjust learning rate automatically.

Adaptive Moment Estimation.

$w_1 = w_1 - \alpha_1 * \frac{\partial}{\partial w_1} J(\overrightarrow{w}, b)$

$w_2 = w_2 - \alpha_2 * \frac{\partial}{\partial w_2} J(\overrightarrow{w}, b)$

$b = b - \alpha_{11} * \frac{\partial}{\partial b} J(\overrightarrow{w}, b)$

If a parameter w_j seems to be moving in roughly the same direction, then increase the learning rate for that parameter and vise versa.

```
model.compile(optimizer = tf.keras.optimizers.Adam(learning_rate = 1e-3),
loss = BinaryCrossEntropy(from_logits =True))
```

Convolutional Layer:

Rather than looking at all pixels, the neuron only look at the pixels in a limited region.

Faster computation;

Need less training data(less prone to overfitting)

1D input

Each layer looks at different window in a time series data.

3 Backward Propagation

Cost function $J(w) = w^2$

Say $w = 3$, $J(w) = 9$

If we increase w by a tiny amount $\epsilon = 0.001$, How does $J(w)$ change?

$w = 3 + 0.001, J(w) = w^2 = 9.006001$

$J(w)$ roughly goes up by 6 times as w.

$\frac{\partial}{\partial w} J(w) = 6.$

If $w \uparrow \epsilon$ causes $J(w) \uparrow k\epsilon$ then

$\frac{\partial}{\partial w} J(w) = k$

If this derivative is small, then gradient descent will make a small update to w_j.

Use sympy:

```
import sympy


J,w = sympy.symbols("J","w")


J = w** 2


dJ_dw = sympy.diff(J,w)


dJ_dw


dJ_dw.subs([w,2])


#gives you derivative of J w.r.t w at w = 2
```

$a = wx + b$, linear activation $a = g(z) = z$

$J(w, b) = \frac{1}{2}(a - y)^2$

$w = 2, b = 8, x = -2, y = 2$

$w \longrightarrow c = wx \longrightarrow a = c + b \longrightarrow d = a - y$

$\longrightarrow J = \frac{1}{2}d^2$

This is the forward prop step.

$\frac{\partial J}{\partial d} = 2$

$\frac{\partial J}{\partial a} = 2$

$\frac{\partial J}{\partial a} = \frac{\partial d}{\partial a} \times \frac{\partial J}{\partial d}$

$\frac{\partial J}{\partial c} = \frac{\partial a}{\partial c} \times \frac{\partial J}{\partial a}$

$\frac{\partial J}{\partial b} = \frac{\partial a}{\partial b} \times \frac{\partial J}{\partial b}$