**Chapter 5: Calculating Network Error with Loss**

To train a model, we tweak the weights and biases to improve the model’s accuracy and confidence. To do this, we calculate how many errors the model has.

The **loss function**, also referred to as the **cost function** algorithm that quantifies how wrong a model is. Loss is the measure of the loss function. And, we ideally want it to be 0.

You may wonder why we do not calculate the error of a model based on the argmax accuracy. Recall our earlier example of confidence: [0.22, 0.6, 0.18] vs [0.32, 0.36, 0.32]. If the correct class were indeed the middle one (index 1), the model accuracy would be identical between the two above. But are these two examples really​ as accurate as each other? They are not, because accuracy is simply applying an argmax to the output to find the index of the biggest value. The output of a neural network is actually confidence, and more confidence in the correct answer is better.

Loss function for:  
**Linear Regression**: Squared error or mean squared error

**Binary Logistic Regression**: Log Loss

**Classification**: Categorical Cross-Entropy Loss

**Cross-entropy** is a loss function that measures the difference between two probability distributions:

* The **true distribution** (ground truth labels).
* The **predicted distribution** (model outputs, typically after softmax or sigmoid).

Special cases of cross entropy:

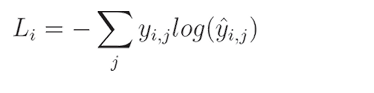
**Categorical cross-entropy** is a special case for **multi-class classification**,

**Log Loss** is another special case for two classes i.e **binary classification**.

**Categorical Cross-Entropy Loss**

As we are classifying the data, we’re using this loss function. Categorical cross-entropy is explicitly used to compare a “ground-truth” probability (y ​ or “targets​”) and some predicted distribution (y-hat ​ or “predictions​”).

The formula for calculating the categorical cross-entropy of y​ (actual/desired distribution) and y-hat​ (predicted distribution) is:



Where L​ i​ denotes sample loss value, i​ is the i-th sample in the set, j ​ is the label/output index, y denotes the target values, and y-hat​ denotes the predicted values.

The softmax activation function returns a probability distribution over all of the outputs. Cross-entropy compares two probability distributions.

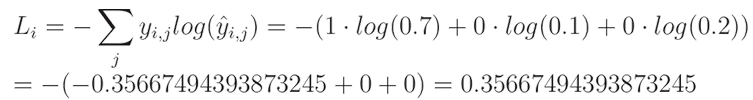
Let’s say:

softmax\_output = [0.7, 0.1, 0.2] ; which is the predicted probability distribution

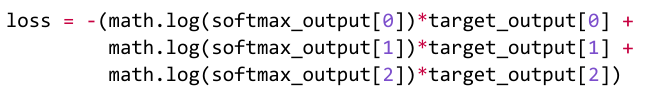
targeted probability distribution = [1, 0, 0]

*Arrays or vectors like this are called* ***one-hot****,​ meaning one of the values is “hot” (on), with a value of 1, and the rest are “cold” (off), with values of 0. When comparing the model’s results to a one-hot vector using cross-entropy, the other parts of the equation zero out, and the target probability’s log loss is multiplied by 1, making the cross-entropy calculation relatively simple. This is also a special case of the cross-entropy calculation, called* ***categorical cross-entropy****.*

So,



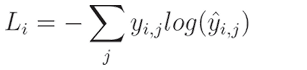
While programming,



As, values other than the target value in the target output is zero and the target value itself is 1 so the loss function can be simplified as



Hence, the formula



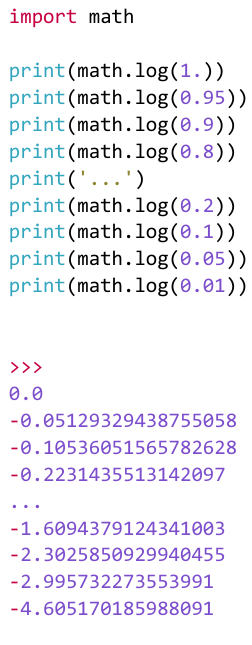
Is simplified to:



Where L​ i​ denotes sample loss value, i​ is the i-th sample in a set, k​ is the index of the target label (ground-true label), y​ denotes the target values and y-hat​ denotes the predicted values.

***What was once an involved formula reduces to the negative log of the target class’ confidence score***

the example confidence level might look like [0.22, 0.6, 0.18] or [0.32, 0.36, 0.32]. In both cases, the argmax​ of these vectors will return the second class as the prediction, but the model’s confidence about these predictions is high only for one of them. The Categorical Cross-Entropy Loss accounts for that and outputs a larger loss the lower the confidence is:

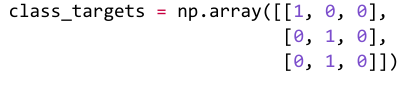


When the confidence level equals 1​, meaning the model is 100% “sure” about its prediction, the loss value for this sample equals 0​. The loss value raises with the confidence level, approaching 0.

**Targets can be either:**

* **one-hot encoded**, where all values, except for one, are zeros, and the correct label’s position is filled with 1.

Eg:



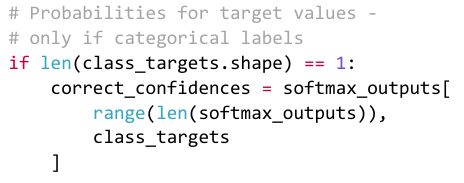
* **Sparse**, which means that the numbers they contain are the correct class numbers.

Eg:

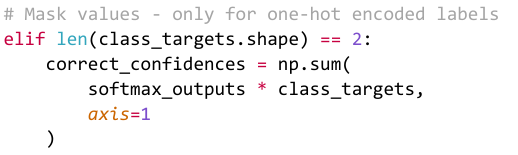


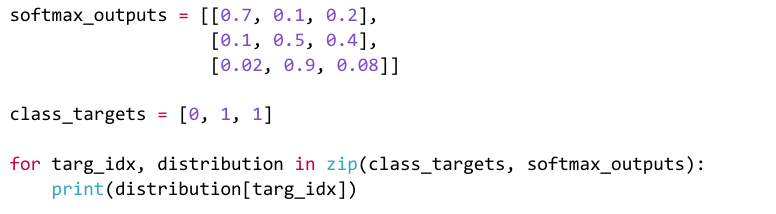
The check can be performed by counting the dimensions — if targets are single-dimensional (like a list), they are sparse, but if there are 2 dimensions (like a list of lists), then there is a set of one-hot encoded vectors.

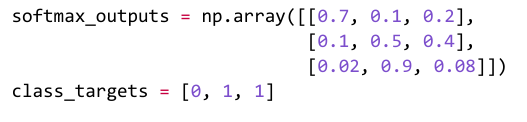
**If sparse target class:**

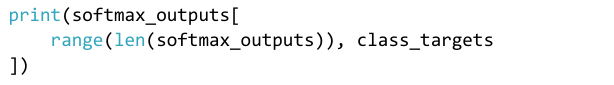


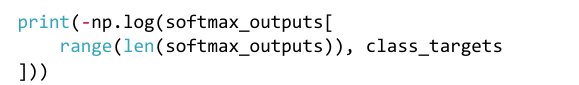
**If one hot encoded target class:**









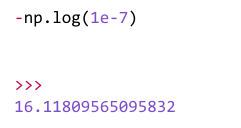


**If confidence score is 0**

Now when confidence score is calculated, there arises another problem, what if the model is absolutely wrong and has a confidence score of 0,

Then -np.log(0) yields negative infinity, which is a problem for further calculation ie to calculate average loss or the gradient later on.

**To fix it:**

1. **Adding a small value to the confidence score for example 1e-7**:  
   ****

This will insignificantly impact the result but yields additional two issues;

* When the model is fully correct ie confidence score is 1, loss becomes a negative value instead of being 0.

A close-up of numbers

AI-generated content may be incorrect.

* Shifting confidence towards 1​, even if by a very small value.

1. **Cip values from both sides by the same number, 1e-7​ in our case:**The lowest possible value will become 1e-7​ (like in the demonstration we just performed) but the highest possible value, instead of being 1+1e-7​, will become 1-1e-7​ (so slightly less than 1​).

This will prevent loss from being exactly 0​, making it a very small value instead, but won’t make it a negative value and won’t bias overall loss towards 1​.



Doing so, the lowest value of y\_pred cannot be less than 1e-7 and the highest cannot be more than 1 – 1e-7, if exists it gets clipped to these threshold values.

Eg.

arr = np.array([1, 5, 10, 15])

clipped = np.clip(arr, 3, 12)

print(clipped)

>>>

[ 3 5 10 12]

**Accuracy**, which describes how often the largest confidence is the correct class in terms of a fraction



predictions = [0, 0, 1]

class\_targets = [0, 1, 1]

predictions == class\_targets → [True, False, True] == 66.66%