Linear Classifiers

Table of Contents

[Parametric Approach 2](#_Toc138881652)

[Visual Viewpoint 5](#_Toc138881653)

[Geometric Viewpoint 6](#_Toc138881654)

[Loss Function 6](#_Toc138881655)

[Multiclass SVM Loss 7](#_Toc138881656)

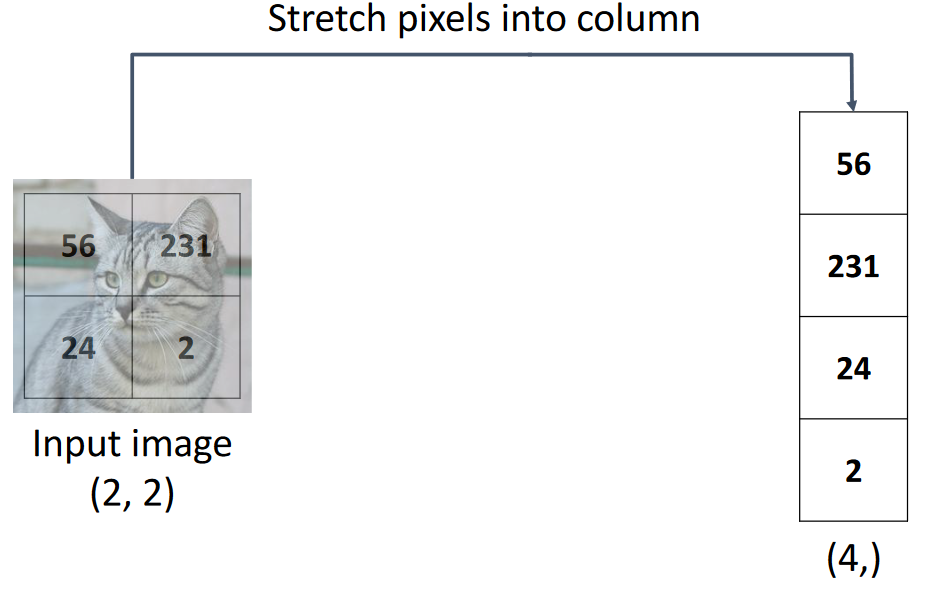
[Regularization 9](#_Toc138881657)

[Cross-Entropy Loss 10](#_Toc138881658)

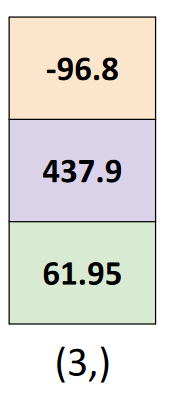
As an introduction to our study of neural networks, we will now be looking at **linear classifiers**. These are the building blocks of neural networks. For this topic, we will be referencing the CIFAR10 dataset.

## Parametric Approach

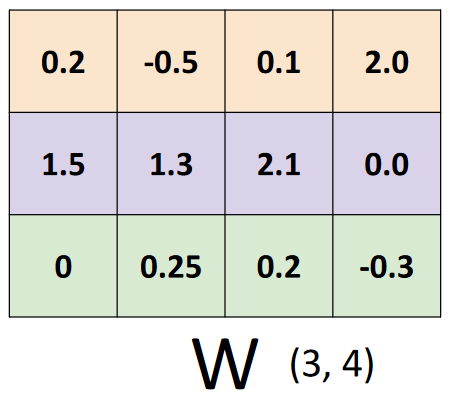
The input for the linear classifiers is an image of dimensions **32x32x3**. However, linear classifiers are incapable of taking **3D inputs**, so we need to first **reshape** this into a single vector of size **3072**. For simplicity, we are instead considering that the input is an 2x2x1 input image.



On the other end, we have 10 possible outputs for the CIFAR10 dataset, so the output vector of size **10**. Each unit of this vector represents the confidence of our classifier that the input belongs to one of the 10 classes. For our simplified version, we are considering just three classes, cat, dog and ship.

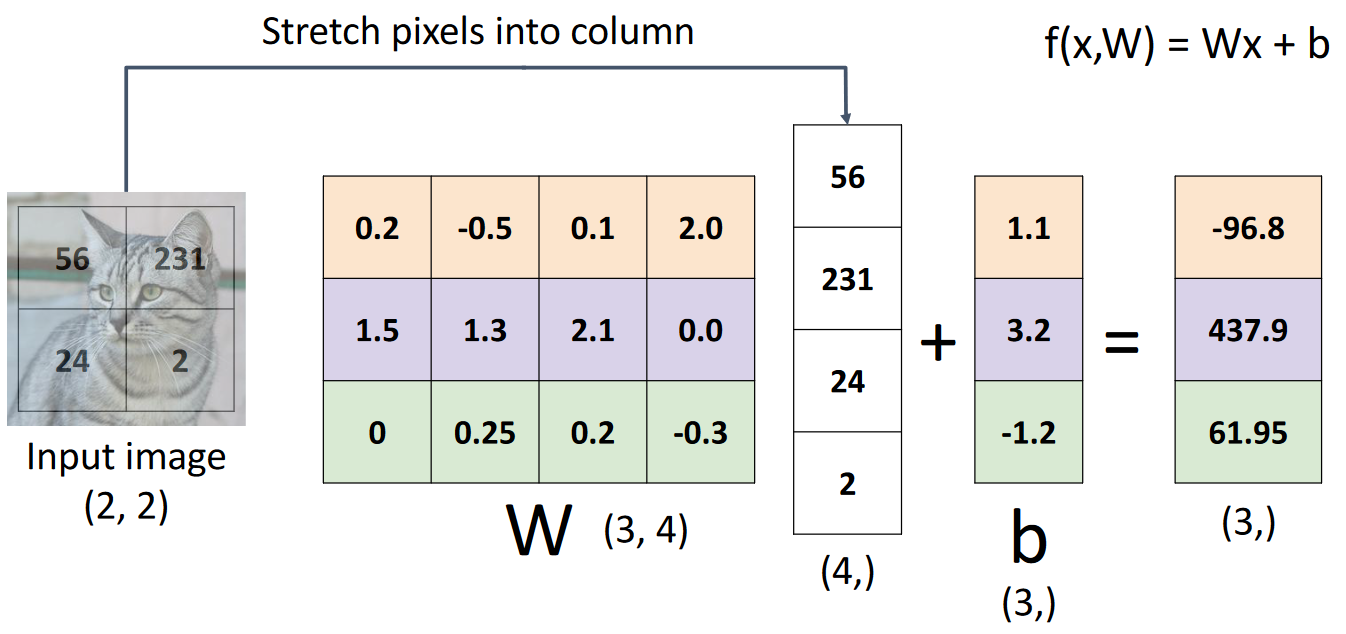


From matrix multiplication, we know that we can convert a 3072 unit vector to a 10 unit vector by multiplying the input with a matrix of size 10x3072. The values of this matrix are called **weights** or **parameters**. Again, for the simplified version, we have a 3x4 matrix.

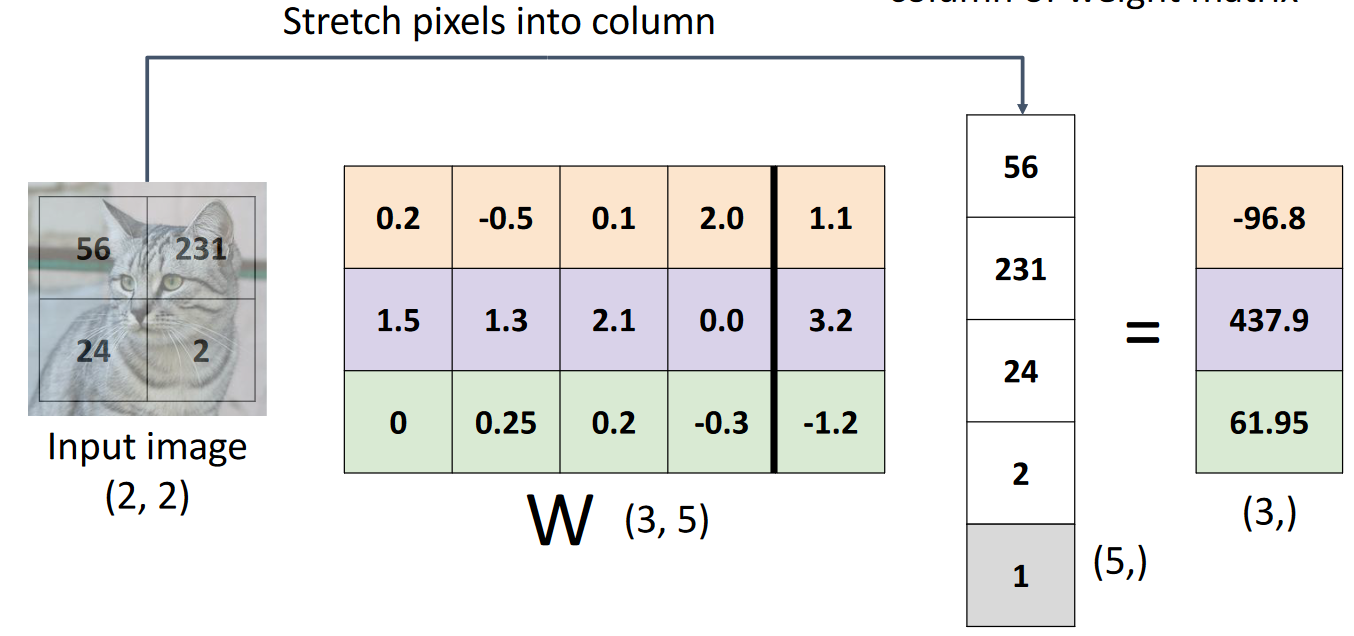


Each of the four pixels in the input image are a separate **feature**. All of the features are being used to determine the probability that one of the classes is correct. Not all of the features are equally important for some classification, so the weights exist to vary their **priority**.

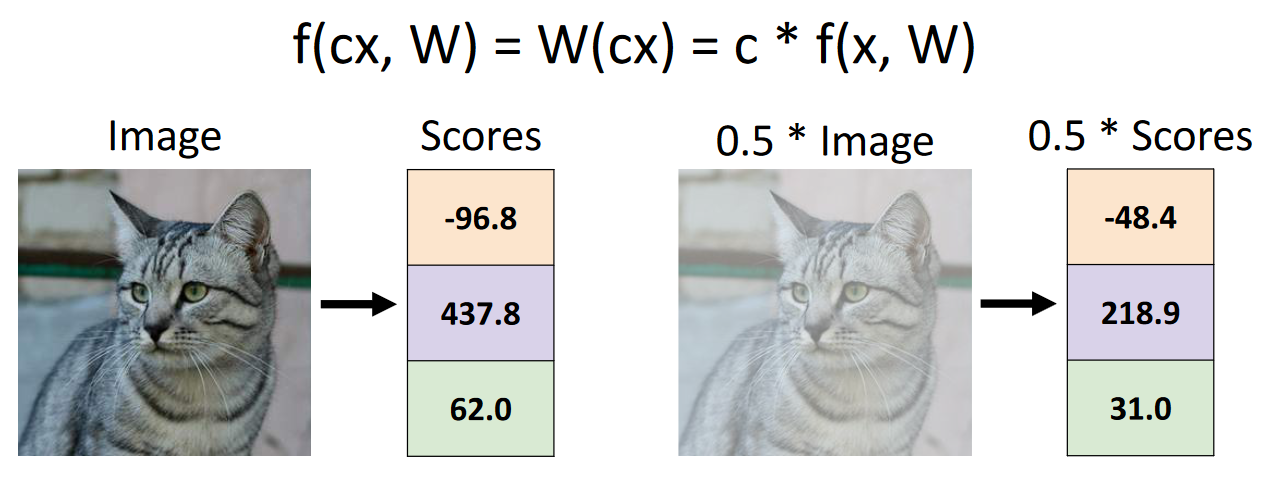
A simple linear classifier like this will result in a **decision boundary** that goes through the origin. To shift the line up and down, we add a **bias** (similar to ).



An alternative way of adding the bias that combines the matrix multiplication and addition is shown below.

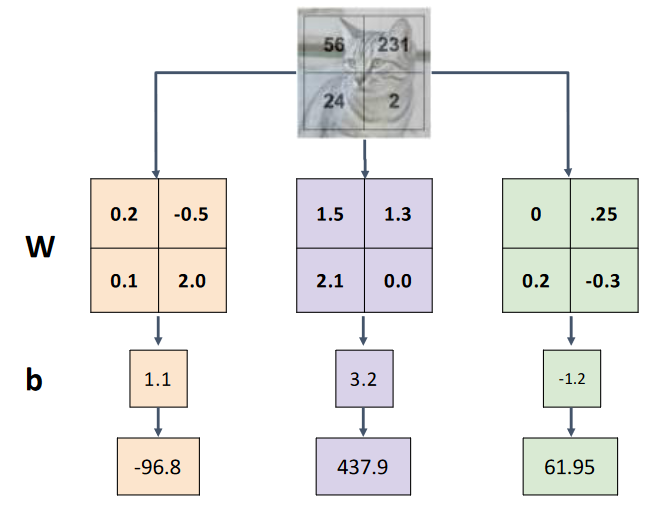


The predictions of a linear classifier are, well, linear. This means that introducing a multiplier to the equation above will not result in a change in the final outcome, even though the values will be different.



### Visual Viewpoint

The method of understanding linear classifiers we saw above is the **algebraic approach**. Another way of looking at the matrix multiplication is as though each row of the weight matrix is a **template** we are applying on top of the input image.



The neat part of visualizing things in this way is that we can technically convert the templates we find into images. For a trained classifier, the templates converted to RGB images would look like this:

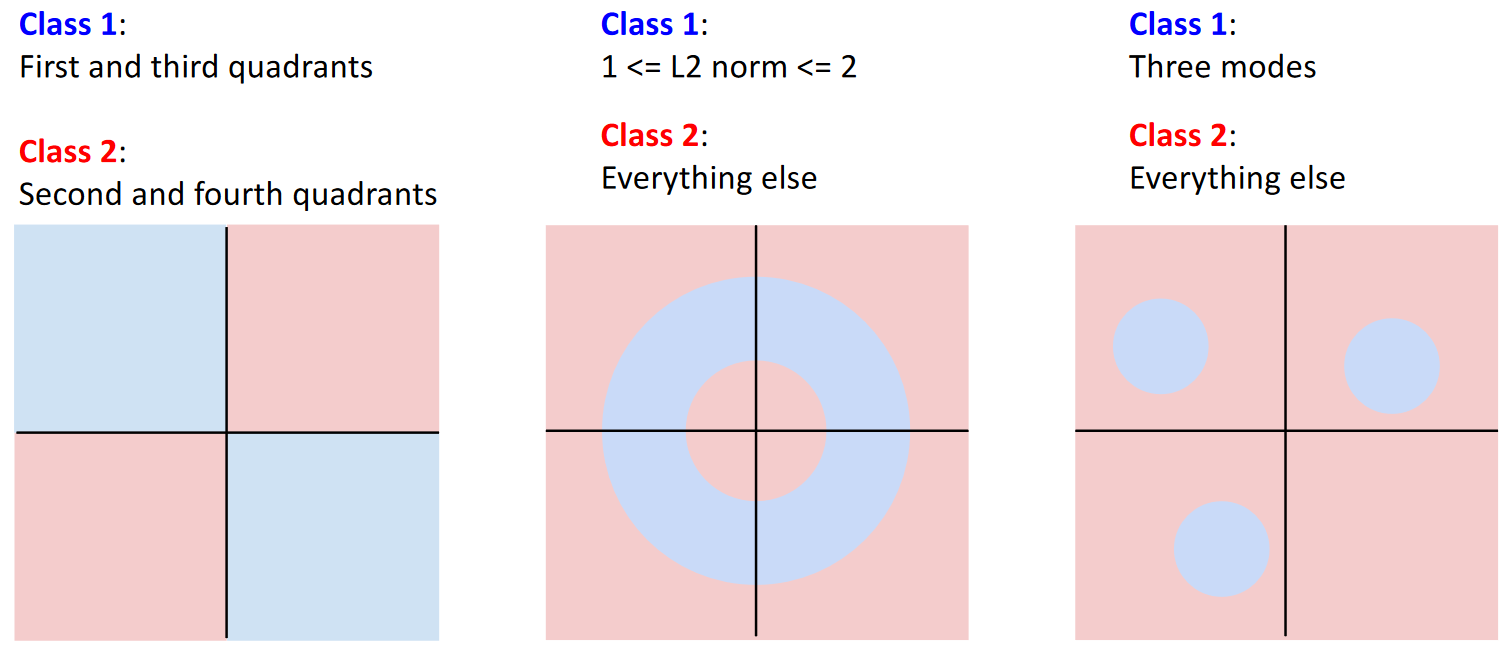


Notice how the templates are vague representations of what the actual images for each class are expected to look like.

This visualization also brings out a major issue of simple linear classifiers. A single template like this cannot capture all possible variations of data. For example, notice that the horse template has two heads! Most likely, 50% of the input data had the horse facing one way and 50% had it facing the other way. To be able to appropriately capture this information, we need more layers, which we will be looking into later.

### Geometric Viewpoint

Yet another way of visualizing the same information is the **geometric viewpoint**. This is just a decision boundary draw in a high-dimensional space. This viewpoint makes it easy to notice issues with drawing the decision boundaries. For example, we are unable to draw decision boundaries for non-linear divisions.



## Loss Function

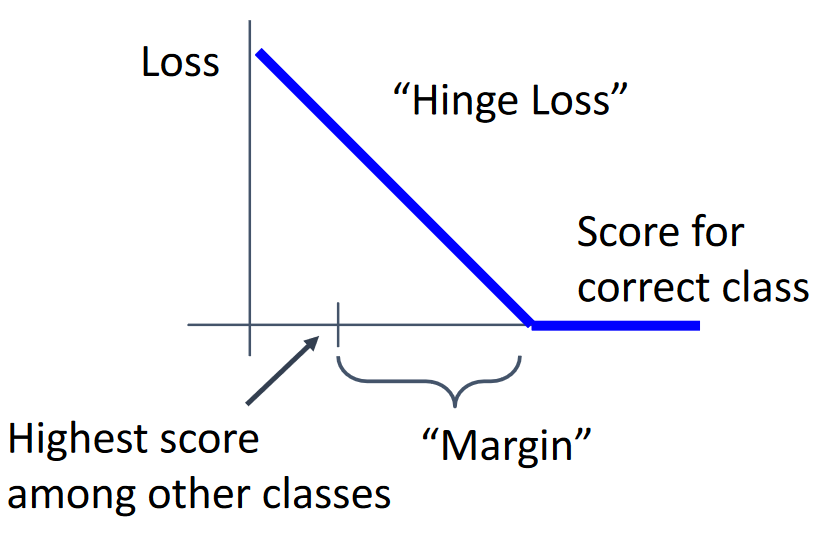
So far, we have only looked at the calculation process for a given set of weights. However, how do we actually **learn** the weights? This consists of two parts. First, we use a **loss function** to find a numerical representation of how good the current weights are. A low value indicates that the weights are good while a high value indicates the opposite. Second, we **optimize** the weight values to minimize the loss.

For a dataset of images where is the input image and is the ground truth label for the image, the loss is found as

Essentially, we are calculating the loss for each of the input images by comparing the predictions of our classifier with the true label and taking the average across the entire dataset.

### Multiclass SVM Loss

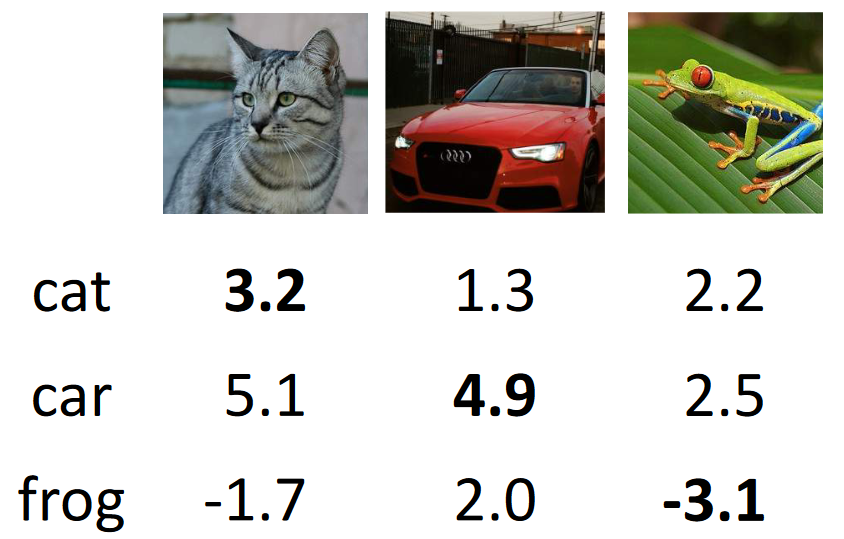
The **Multiclass SVM Loss** is a loss function that not only demands that the score for the correct class be higher than that of the other classes, but also that the score be higher by at least a certain amount, called the **margin**. If the margin is maintained, there is no loss, but if it is not, the loss increases linearly, with the highest values being for cases where the correct class has a lower score than other classes.



Mathematically, this is calculated as:

Here, is the margin.

Notice that we are skipping over the class that is being judged since that class will always give us a value of , which is redundant information.



For the example above, consider that . Thus, the scores are:

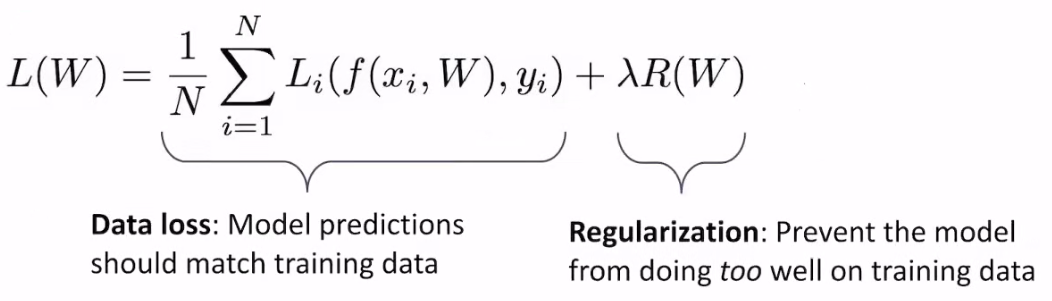
Thus, we got a positive value for the first and third classes, both because the correct class did not have the highest score. The value for the third class was much higher, since the correct class being actively predicted against. The second class was the only one that got a 0 value since the correct class was predicted and the margin was maintained.

One major issue with the multiclass SVM loss is that we cannot keep pushing the scores for the correct class and the other classes apart. There is a limit to how far apart they will be, as dictated by the margin. This does not necessarily cause issues, but it is not the theoretical ideal.

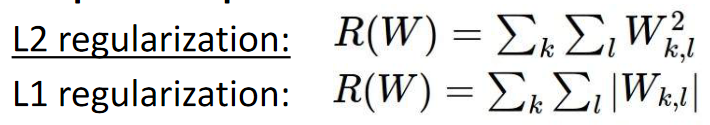
### Regularization

Suppose we found some value of for which we get 0 loss using SVM loss. However, multiples of will also give us 0 loss. This brings up the question of how to decide between these values. Most likely, we want simpler values of since those have a higher chance of generalizing well.

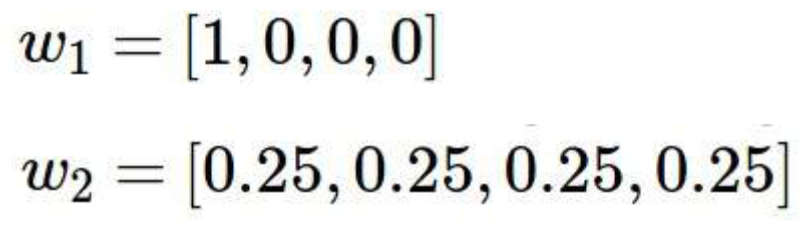
**Regularization** helps us do exactly this. The regularization term has nothing to do with the input. It only works with the weights. Here, is the **regularization strength**, which is a hyperparameter.



There are various regularization terms such as L2 regularization, L1 regularization, dropout, batch normalization, etc.



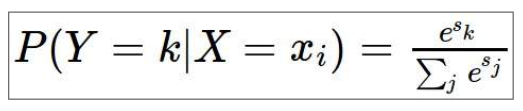
For the two weights below, regularization gives the same priority but regularization prefers . will give a lower value for which will result in a lower loss.



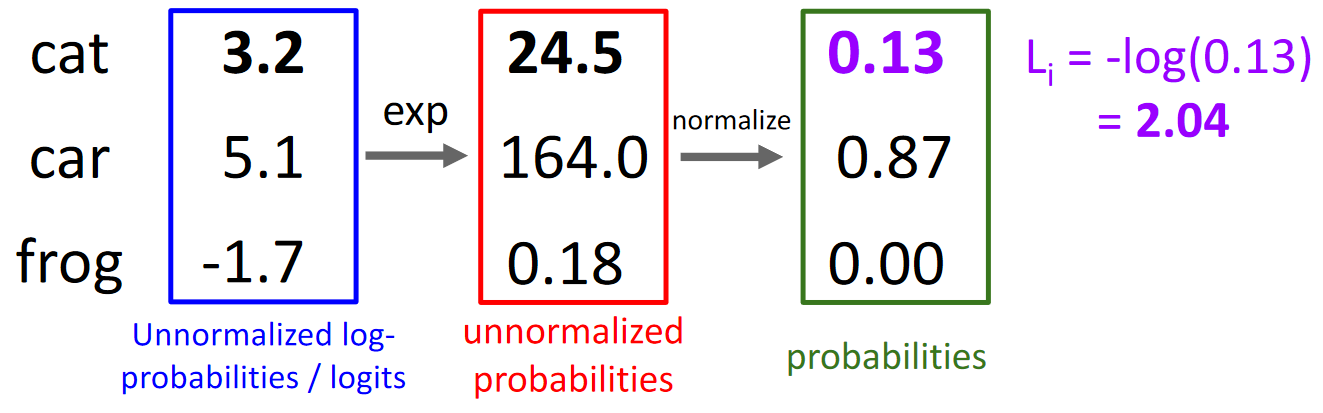
Another way of looking at this is that L2 regularization ‘spreads out’ the weights. By doing so, it does a better job than L1 regularization at choosing simpler weights. However, both have the same purpose, to choose simpler weights.

### Cross-Entropy Loss

**Cross-Entropy Loss** interprets the raw classifier scores as **probabilities**. It does this by converting the scores to probabilities using the **softmax function**.



The actual loss values are the **negative log** of the probability values.



The minimum possible loss is while the maximum possible loss is infinite.

Cross entropy loss solves the issue of SVM loss failing to push values beyond a certain threshold. The loss value will never be 0 unless the model manages to make perfect predictions.