Accuracy and Loss

Sigmoid Cross Entropy

While developing models we use many functions to check the model's accuracy and loss. For a model to be in good condition, loss calculation is a must since loss acts like a penalty. The lower the loss better will be the working of the model.

There are many kinds of loss functions. One such function is the Sigmoid cross entropy function of TensorFlow. The sigmoid function or logistic function is the function that generates an S-shaped curve. This function is used to predict probabilities therefore, the range of this function lies between 0 and 1.

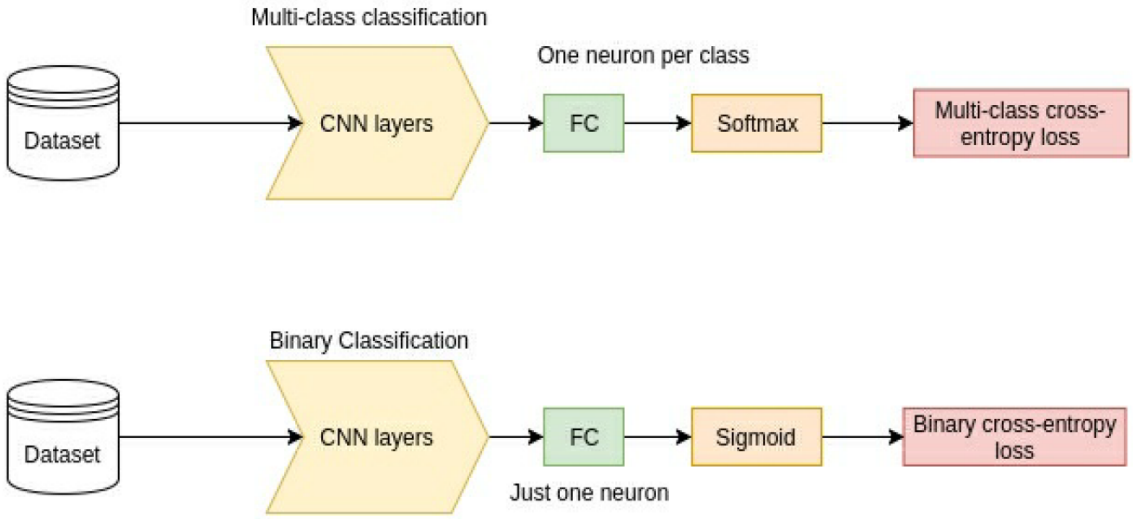
Cross Entropy loss is the difference between the actual and the expected outputs. This is also known as the log loss function and is one of the most valuable techniques in the field of Machine Learning.

CNN Architecture

The crucial part of an image classification model is its CNN layers. These layers will be responsible for extracting features from image data. The output of these CNN layers will be a feature vector, which like before, we can use as input for the classifier of our choice. For many CNN models, the classifier will be just a fully connected layer attached to the output of our CNN.

It is important to note that at its core, the CNN architecture used in classification or a regression problem such as localization would be the same.

The only real difference will be what happens after the CNN layers have done their feature extraction. For example, one difference could be the loss function used for different tasks, as it is shown below:



Different problems that CNNs can be used to solve. It will become apparent that lots of tasks involving images can be solved using a CNN to extract some meaningful feature vector from the input data, which will then be manipulated in some way and fed to different loss functions, depending on the task. For now, let's crack on and focus firstly on the task of image classification by looking at the loss functions commonly used for it.

Cross-entropy loss (log loss)

The simplest form of image classification is binary classification. This is where we have a classifier that has just one object to classify, for example, dog/no dog. In this case, a loss function we are likely to use is the binary cross-entropy loss.

The cross entropy function between true labels *p* and model predictions *q* is defined as:

$$H(p, q) = -\sum_i p_i \log(q_i)$$

With *i* being the index for each possible element of our labels and predictions.

However, as we are dealing with the binary case when we have only two possible outcomes, y=1 and y=0, then p$\in\{y, 1-y\}$ and q $\in\{\hat{y}, 1-\hat{y}\}$ can be simplified down and we get:

$$H(p, q) = -(y \log(\hat{y}) + (1-y) \log(1-\hat{y}))$$

This is equivalent

Iterating over m training examples, the cost function *L* to be minimized then becomes this:

$$L(\hat{y}, y) = -\frac{1}{m} \sum_{i=1}^{m} (y_i \log(\hat{y_i}) + (1-y_i) \log(1-\hat{y_i}))$$

This is intuitively correct, as when $y = 1$, we want to minimize $L(\hat{y}, y) = -\log \hat{y}$, which requires large $\hat{y}$ and when $y = 0$, we want to minimize $L(\hat{y}, y) = -\log(1-\hat{y})$, which requires a small $\hat{y}$.

In TensorFlow, the binary cross entropy loss can be found in the `tf.losses` module. It is useful to know that the name for the raw output $\hat{y}$ of our model is logits. Before we can pass this to the cross entropy loss we need to apply the **sigmoid** function to it so our output is scaled between 0 and 1. TensorFlow actually combines all these steps together into one operation, as shown in the code below. Also TensorFlow will take care of averaging the loss across the batch for us.

```
loss = tf.losses.sigmoid_cross_entropy(multi_class_labels=labels_in, logits=model_prediction)
```

p	q	log(q)	p.log(q)	-(1-p)*LOG(1-q)
1	0.8	-0.09691	0.09691	
0	0.2	-0.69897		0.096910013
0	0.4	-0.39794		0.22184875

Multi-class cross entropy loss

Multi-class cross entropy loss is used in multi-class classification, such as the MNIST digits classification problem from `Chapter 2`, *Deep Learning and Convolutional Neural Networks*. Like above we use the cross entropy function which after a few calculations we obtain the multi-class cross-entropy loss *L* for each training example being:

$$L(\hat{y}, y) = -\sum_{k}^{K} y^{(k)} \log \hat{y}^{(k)}$$

Here, $y^{(k)}$ is 0 or 1, indicating whether class label k is the correct classification for predicting $\hat{y}^{(k)}$. To use this loss, we first need to add a softmax activation to the output $\hat{y}$ of the final FC layer in our model. The combined cross-entropy with softmax looks like this:

$$L(\hat{y}, y) = -\sum_{k}^{K} y^{(k)} \log \frac{e^{\hat{y}^{(k)}}}{\sum_{j=1}^{K} e^{\hat{y}^{(j)}}}$$

It is useful to know that the name for the raw output $\hat{y}$ of our model is logits. Logits are what is passed to the softmax function. The softmax function is the multi-class version of the sigmoid function. Once it is passed through the softmax function, we can use our multi-class cross entropy loss. TensorFlow actually combines all these steps together into one operation, as shown:

```
loss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=model_logits, labels=labels_in))
```

We must use `tf.reduce_mean`, as we will get a loss value for each image in our batch. We use `tf.reduce_mean` to get the average loss for our batch.

The train/test dataset split

For the time being, be aware that we need to split our dataset into two sets: training and test. As mentioned in `Chapter 1`, *Setup and Introduction to TensorFlow*, this needs to be done because we need to somehow check whether the model is able to generalize out of its own training samples (whether it's able to correctly recognize images that it has never seen during training). If our model can't do this, it isn't of much use to us.

Here are a few other important points to remember:

- The training and testing data must come from the same distribution (so combine and shuffle all your data before splitting)
- The training set is often bigger than the test set (for instance, training: 70% of total, testing: 30% of total).

For the examples that we will deal with in these early chapters, these basics will be enough, but in subsequent chapters, we will see, in further detail, how to properly set up your dataset for bigger projects.

Datasets

In this section, we will discuss the most important and famous recent datasets used in image classification. This is necessary, because it is likely that any perusal into Computer Vision will overlap with them (including in this book!). Before the arrival of convolutional neural networks, the two main datasets used in image classification competitions by the research community were the Caltech and PASCAL datasets.

The Caltech dataset was established by California Institute of Technology and was released in two versions. Caltech-101 was published in 2003 with 101 categories of about 40 to 800 images per category, and Caltech-256 in 2007 with 256 object categories, containing a total of 30607 images. The images were collected from Google images and PicSearch, and their size was roughly 300x400 pixels.

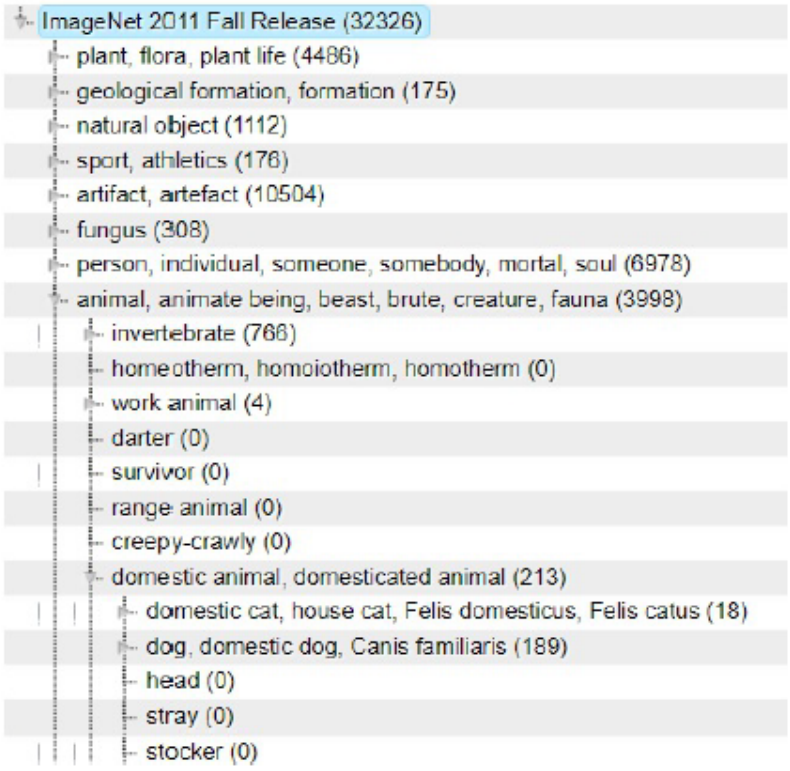
The Pascal **Visual Object Classes** (**VOC**) challenge was established in 2005. Organized every year till 2012, it provides a famous benchmark dataset of a wide range of natural images for *Image category Classification, Object detection, Segmentation, and Action Classification*. It is a diverse dataset that includes images from flickr of various sizes, pose, orientation, illumination, and occlusion. It has been developed in stages from the year 2005 (only four classes: bicycles, cars, motorbikes, and people, train/validation/test: 1578 images containing 2209 annotated objects) to year 2012 (twenty classes, The train/validation data has 11,530 images containing 27,450 ROI annotated objects and 6,929 segmentations).

The major change came with the PASCAL (VOC) 2007 challenge, when the number of classes increased from 4 to 20 and have been fixed since then. Evaluation metrics for the Classification task changed to average precision. The annotation for test data are only provided until the VOC 2007 challenge.

With the arrival of more sophisticated classification methods, the preceding datasets were not sufficient, and the ImageNet dataset along with the CIFAR dataset, described in the following sections, became the new standards in classification testing.

ImageNet

The ImageNet dataset was created in 2010 as a collaborative effort of Alex Berg (Columbia University), Jia Deng (Princeton University), and Fei-Fei Li (Stanford University) to run as a tester competition on large-scale visual recognition, in conjunction with the *PASCAL Visual Object Classes Challenge*, 2010. The dataset is a collection of images that represent the contents of WordNet. WordNet is a lexical database for the English language. It groups English words into sets of synonyms called **synsets** in a hierarchical structure. The following screenshot shows the WordNet structure of nouns. The number in the brackets is the number of synsets in the subtree.



The evolution of image classification algorithms, which almost solved the classification challenge on the existing datasets led to the need for a new dataset that would allow image classification at a large scale. This is closer to a real-world scenario where we would like the machine to describe the content of an arbitrary image simulating human capability. Compared to its predecessors, where the number of classes is in the 100s, ImageNet offers over 10 million high-quality images covering more than 10,000 classes. Many of these classes are related to each other, which makes the task of classification more challenging, for example, distinguishing many breeds of dogs. Since the dataset is enormously huge, it

was hard to annotate each image with all the classes that are present in it, so by convention, each image was labeled only with one class.

Since 2010, the annual ImageNet Large Scale Visual Recognition Challenge (ILSVRC) challenge focuses on image classification, single-object localization, and detection. The data for the object classification challenge consists of 1.2 million images (from 1000 categories/synsets), of training data, 50,000 images of validation data, and 100,000 images of testing data.

In the classification challenge the main metric used to evaluate an algorithm is the top-5 error rate. The algorithm is allowed to give five predicted classes and will not be penalized if at least one of the predicted classes matches the ground truth label.

Formally if we let i be the image and C_i be the ground truth label. Then, we have the predicted labels c_{ij}, $j \in [1,5]$, where at least one is equal to C_i to consider it a successful prediction. Consider that the error of a prediction is as follows:

$$d_{ij} = \begin{cases} 0 & c_{ij} = C_i \\ 1 & c_{ij} \neq C_i \end{cases}$$

Meaning the final error of an algorithm is then the proportion of test images on which a mistake was made, as shown:

$$error = \frac{1}{N} \sum_{i=1}^{N} \min_j d_{ij}$$

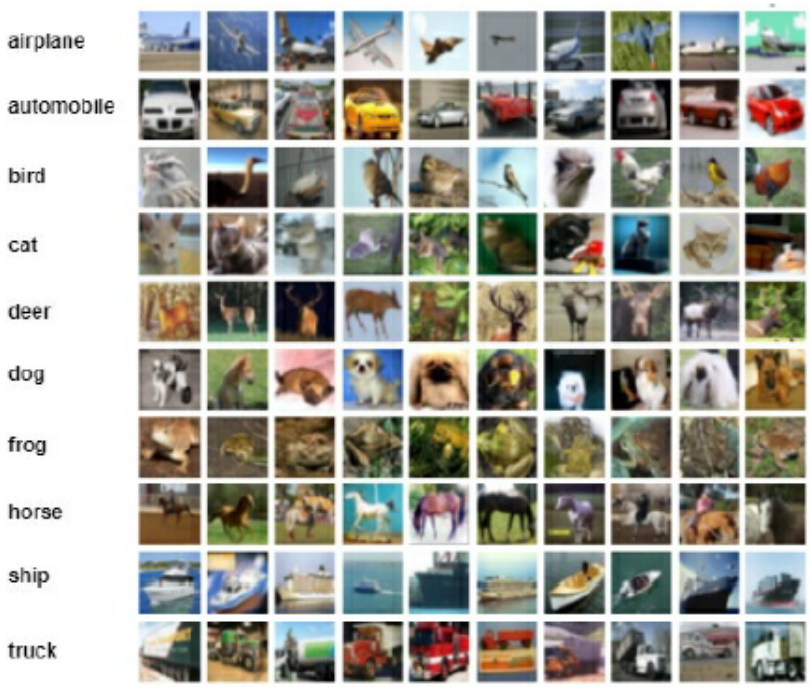
Imagenet was one of the big reasons why deep learning has taken off in recent years. Before deep learning became popular the top-5 error rate in ILSVRC was around 28% and not going down much at all. However, in 2012, the winner of the challenge, SuperVision, smashed the top-5 classification error down to 16.4%. The teams model, now known as AlexNet, was a deep convolutional neural network. This huge victory woke people up to the power of CNNs and it became the stepping stone for many modern CNN architectures.

In the following years CNN models continued to dominate and the top-5 error rate kept falling. The 2014 winner GoogLeNet reduced the error rate to 6.7% and this was halved again in 2015 to 3.57% by ResNet. Since then there has been smaller improvements with 2017's winner "WMW squeeze and excitation networks" produced a 2.25% error.

CIFAR

The CIFAR-10 and CIFAR-100 datasets are small (compared to modern standards) image datasets collected by Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. These datasets are widely used by the research community for image classification tasks. They are considered challenging, because the image quality is very low and the objects in the images are sometimes partially visible. At the same time, the datasets are convenient due to this small image size, so researchers can quickly produce results on them. CIFAR-100 increases the challenge since there are only a small number of images per class and the number of classes is fairly large. The CIFAR10 and CIFAR100 datasets contain 60,000 images each. Images in both datasets are 32x32x3 RGB color images.

In CIFAR-10, there are 10 classes and each class has 6,000 images. The dataset is divided into 50,000 training images and 10,000 test images. The following is the list of classes and some random images from each class from the CIFAR-10 dataset, so you can see what it looks like:



CIFAR-100 has 100 classes and 600 images per class. These 100 classes are divided into 20 superclasses. Each image has a **fine** label (the class that it belongs to) and a **coarse** label (the superclass it belongs to). The list of classes and Superclasses in CIFAR-100 is available at https://www.cs.toronto.edu/~kriz/cifar.html. Increasing the number of classes from coarse (20) to fine (100) can be helpful to maximize inter-class variability. This means that we want our model to consider two similar-looking objects in the image to belong to a different class. For example, a bed and a couch look similar but not exactly the same, and placing them in separate classes will ensure that they look different to the trained models.

The algorithm evaluation process for CIFAR is the same as in ImageNet. The best reported top-1 error for CIFAR-10 is 3.58%, and for CIFAR-100 is 17.31%, as reported by Saining Xie et al. in *Aggregated Residual Transformations for Deep Neural Networks*, where they presented the novel ResNeXt architecture. The Current state-of-the-art techniques for image classification using *Deep learning results on CIFAR-10 and CIFAR-100* can be found at http://rodrigob.github.io/are_we_there_yet/build/classification_datasets_results.html and https://github.com/RedditSota/state-of-the-art-result-for-machine-learning-problems.

Learning rate scheduling

In the last chapter, we briefly mentioned a problem that can occur by keeping a constant learning rate during training. As our model starts to learn, it is very likely that our initial learning rate will become too big for it to continue learning. The gradient descent updates will start overshooting or circling around our minimum; as a result, the loss function will not decrease in value. To solve this issue, we can, from time to time, decrease the value of the learning rate. This process is called learning rate scheduling, and there are several popular approaches.

The first method involves reducing the learning rate at fixed time steps during training, such as when training is 33% and 66% complete. Normally, you would decrease the learning rate by a factor of 10 when it reaches these set times.

The second approach involves reducing the learning rate according to an exponential or quadratic function of the time steps. An example of a function that would do this is as follows:

```
decayed_learning_rate = learning_rate * decay_rate ^ (global_step / decay_steps)
```

By using this approach, the learning rate is smoothly decreased over the time of training.

A final approach is to use our validation set and look at the current accuracy on the validation set. While the validation accuracy keeps increasing, we do nothing to our learning rate. Once the validation accuracy stops increasing, we decrease the learning rate by some factor. This process is repeated until training finishes.

All methods can produce good results, and it may be worth trying out all these different methods when you train to see which one works better for you. For this particular model, we will use the second approach of an exponentially decaying learning rate. We use the TensorFlow operation —`tf.train.exponential_decay`—to do this, which follows the formula previously shown. As input, it takes the current learning rate, the global step, the amount of steps before decaying and a decay rate.