Classification of images using a data driven approach: K-nearest neighbor and SVM

LECTURE 1, 06/12/2024 (1 H)

Abstract. In this first lecture, we will start by discussing how to perform the task of assigning a label (from a given ensemble of labels) to an image with a computer. We will introduce a linear classifier (fully connected) based on a score function that maps images to labels and a loss function that can quantify how good is the agreement between the assigned labels and the image truth. Then we will understand how the network is learning with a process called optimization, that includes various processes behind it: stochastic gradient descent, backpropagation etc.

Keywords: Data driven approach, k-nearest neighbor, classification and optimization tasks with stochastic gradient descent, backpropagation, neural network architecture, activation functions, spatial arrangement, layer sizing patterns, hyperparameters

Contents:

1. *Introduction to the problem of classifying images*
2. *Data-driven approach concept: an example with the k-nearest classifier*
3. *The simplest score function: a linear classifier (Support Vector Machine - SVM)*
4. *The loss function*
5. *The optimization process*
6. *Backpropagation and computational graphs*

Credits for the content in this lecture:

Almost all the material from this lecture is a collection of concepts, explanations, images and visualizations from other more technical computer engineering lectures, done with the aim of extracting and offering to the students of the master program of climate sciences only the minimal necessary information to achieve an beginner level of understanding of machine learning methods applied to computer vision. We extracted learning material from many currently available resources present on the web that we want to fully acknowledge here. All what presented here can be found in a more extended, technical and detailed way in the original sources listed here:

- Stanford course on convolutional neural network for computer vision
- CIFAR10 dataset of images
- Sebastian Raschka's course page
- Deep learning tutorial from Adrian Rosebrock
- Article of Kemal Erdem on Medium on t-SNE representations
- article from Martin Wattenber, Fernanda Viegas and Ian Johnson
- article on Multilayer perceptron by Carolina Bento

We deeply thank all the people that contributed to share this knowledge and make it available online, so that other people in the world can benefit from it.

1.1 Introduction to the problem of classifying images

The general task of classifying images consists of assigning a label to a given image starting from a set of possible labels. Our brain does this task almost daily on many different images, and it is nearly an automatic operation where we do not have to think. If you look at the two images below, you don't have to think to identify a cat and a dog, even if they have two similar sunglasses and they mainly lay in a similar position. You also were not bothered by the color of the background to identify the animals, nor confused by the color of their fur. (Figure 1).

Figure 1.1: Two images of a cat and a dog.

A computer, instead, sees an image more or less like this (Figure 2). To be precise, this is how it will see a black-and-white print: a matrix of numbers ranging from 0 to 255, typically. For color images, it would have to consider three matrices, like the one in black and white, one for each of the R, G, and B dimensions.

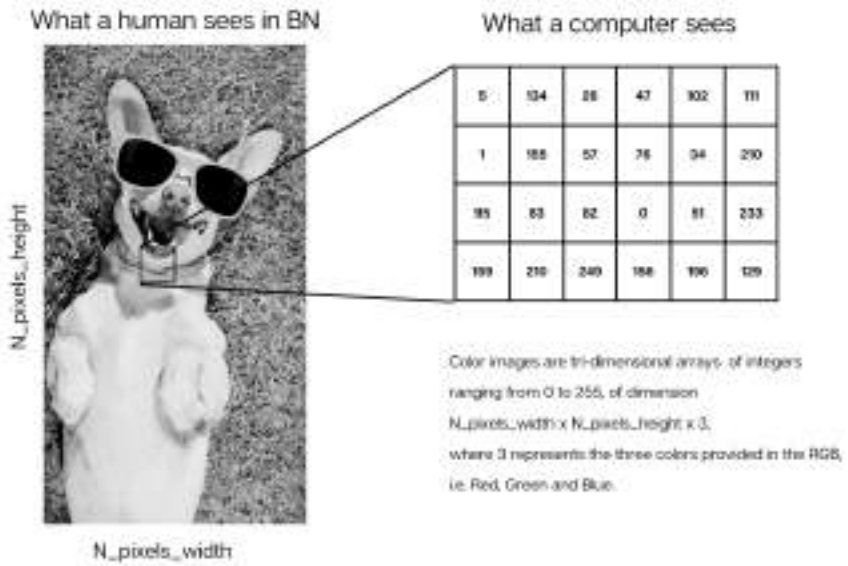


Figure 1.2: Representation of a black and white image as seen by a computer. Reconstructed based on figure 1 in https://cs231n.github.io/classification/

For our brain, the transformations of the image shown in Figure 3 are intuitive and we still recognize them as belonging to the label class dog; for the computer, the transformation the dog images undergo represented in Figure 3 challenge its ability to still recognize the images as belonging to the class dog. A classification algorithm must be **invariant** to all these transformations, that means that the classification result should not change if the image undergoes one of these transformations.

Figure 1.3: Examples of deformations and alterations of images that can represent a challenge for a computer to be learned. With rotations we can change the orientation of the objects, or with a scale transformation we can also alter the size of an object. We can apply deformations on parts of an object, or we can disturb the vision with clutter on the background. Also occlusions, consisting of showing only parts of the objects, are possible, and finally we can introduce a variability in the objects belonging to the same class. Image reinterpreted based on content in https://cs231n.github.io/classification/

1.2 Data-driven approach concept: an example with the k-nearest classifier

Now, we want to write an algorithm that enables our computer to classify images. **A classifier is a system that takes as input a vector of feature values and outputs a discrete class value for it**. How would you do it? This is quite a different task than reordering elements of an array or sorting words based on their starting letter. There's not a straightforward way to identify a strategy for classifying images.

One way to go is to learn from data, i.e., collect many samples of images of each of the selected classes. Then, by looking at the pictures of each class, we try to learn the visual characteristics we can identify. When one takes this approach, we say they use a **data-driven approach**. For this approach to work, we need to collect data in each of the classes we want the classifier to be able to classify the input.

The sequence of steps that a classifier needs to take are:

- **INPUT**: prepare an input of N images with N labels, each in one of the D classes of the dataset. This dataset is called the training dataset.

- **LEARNING**: In this phase, the classifier uses the training set to learn for each image, which is its label.
- **EVALUATION**: In this phase, the classifier is evaluated by testing it on a dataset it has never seen before (testing dataset). We can assess if the predicted label equals the image's label for each image in the test dataset.

Example of a classifier: the k-nearest neighbor

One example of an image classifier is the k-nearest neighbor. How does the nearest neighbor algorithm work? It is an algorithm based on proximity; it stores the labels of the training examples during the training, and since there's no additional processing other than merely memorizing the labels, the K-nn is also called the lazy algorithm. Once the training is over, the KNN algorithm identifies the k-nearest neighbors of a given image by deriving the label for that image from the labels of the k nearest images.

The CIFAR10 dataset contains 60000 color images in 10 classes, with 6000 images per class. The figure 4 shows 10 random images for each class.

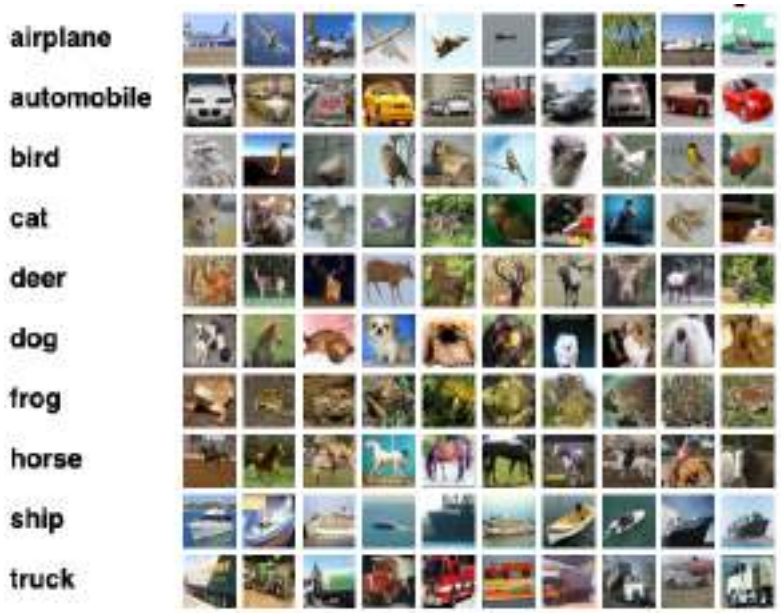


Figure 1.4: Example of 10 random images for the 10 classes from the CIFAR10 dataset, available at https://www.cs.toronto.edu/%7Ekriz/cifar.html

The classifier must prescribe a way to measure the distance to establish the K-nearest samples. Reasonable choices for calculating the distance between two images can be the L1 or the L2 distances (also sometimes called Manhattan and Euclidean distances, respectively).

$$d_{L_1}(I_1, I_2) = \sum_i |I_1^i - I_2^i|$$

$$d_{L_2}(I_1, I_2) = \sqrt{(\sum_i (I_1^i - I_2^i)^2}$$

The distances are calculated pixel-wise. For L1 an example is shown in Figure 5,

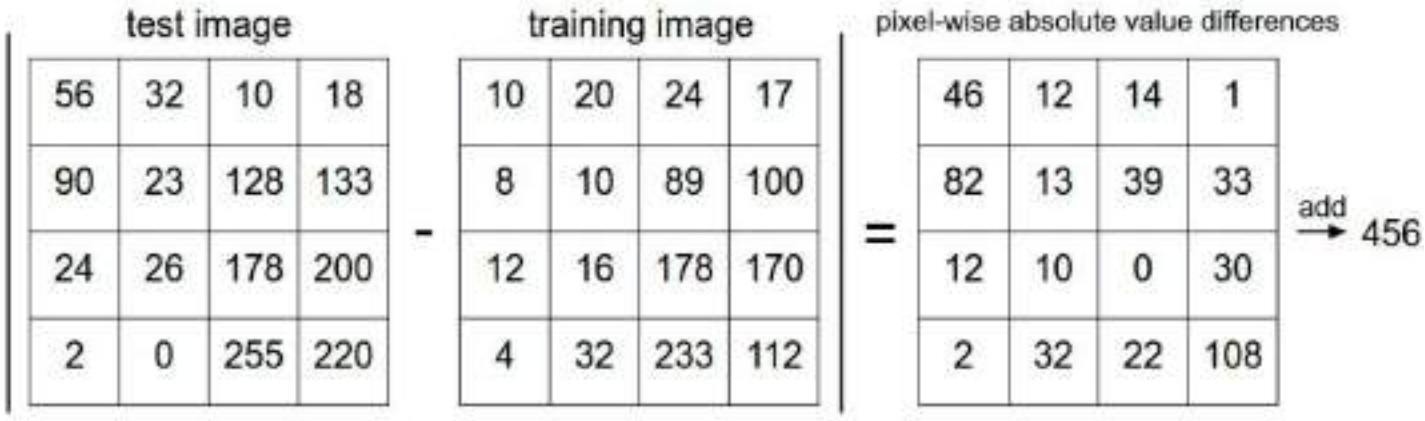


Figure 1.5: graphical representation of the calculation of the L1 pixelwise distance between an test and an training image. Image reinterpreted based on content in https://cs231n.github.io/classification/

while for L2, the difference is computed pixel-wise as before, but then all terms are squared and added under the square root.

Once the ensemble of the closest k images is derived, the predicted label is the one that recurs more frequently in the K-selected label›s. Figure 5 shows an example of the KNN algorithm (from Raschka's notes):

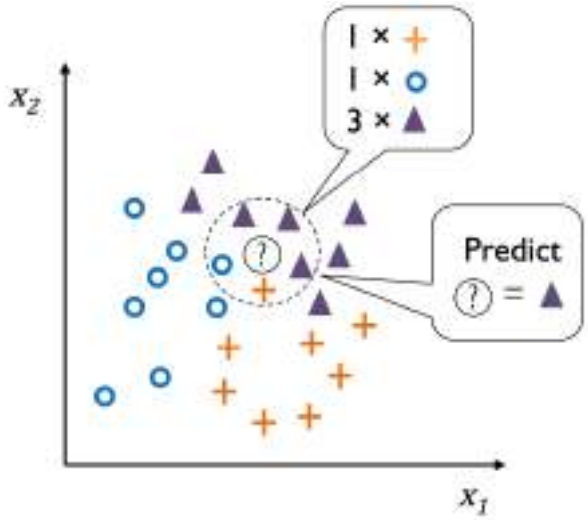


Figure 1.6: Example of K-nn algorithm with k=5 and 3 classes. In the figure, we can see that the predicted label of the grey triangle is the one that among the k-nearest neighbors, occurs more frequently, 3 times compared to 1 time for each of the other classes (Image from Sebastian Raschka's course page stat.wisc.edu/%7Esraschka/teaching/stat479-fs2018/).

The parameter k for the K nearest neighbor classifier and the distance metric used are "*hyperparameters*." The **hyperparameters are the ensemble of all the parameters (and choices) that we need to set for setting up the classifier, for which we don't know a proper value apriori.** To understand the best value/choice for the hyperparameters, we need to reserve a small part of the training dataset, called the **validation dataset**, to tweak hyperparameters. For this purpose, using a dataset that is not the test dataset is essential. The risk is to tune the parameters to work perfectly on the training data but then create, in this way, a classifier that cannot generalize on a dataset that is not its training dataset. This problem is called **overfitting**. For the example of the CIFAR10, we can set a

validation dataset of 1000 images, while the training data is 49000. The dataset configuration will thus look like this

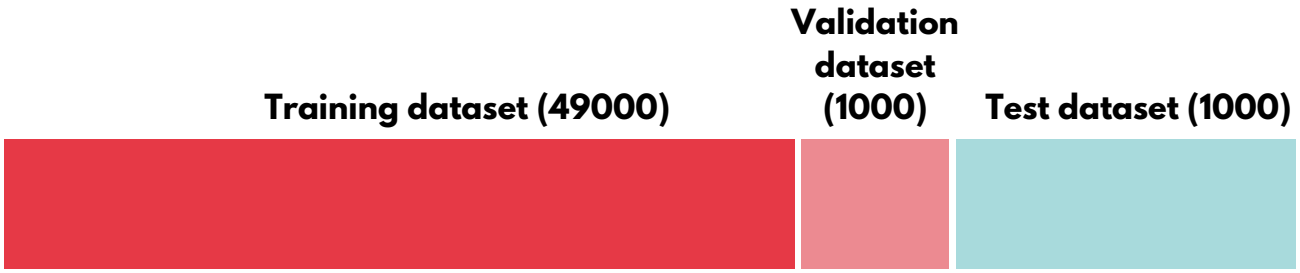
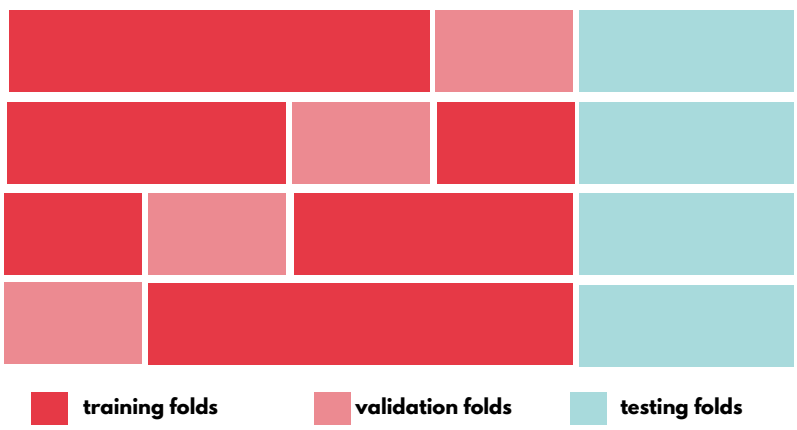


Figure 1.7: Example dataset splitting in training, validation and test dataset for the CIFAR10 dataset.

Cross-validation: another way to avoid overfitting is to split the data into folds, and then try each fold as validation dataset, and then average the results.



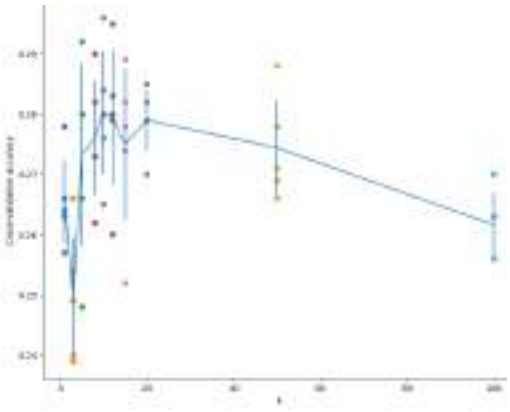


Figure 1.8: Example dataset splitting in training, validation and test dataset for tuning of the k-value hyperparameter for the knn algorithm for the CIFAR10 dataset. The figures are obtained with the knn algorithm provided in the exercises of the course https://cs231n.github.io/classification/

When we run the cross validation for determining the right value for k using 5 folds, for each fold we obtain one dot associated to the selected value of k we are testing. By plotting the accuracy for each of the k values tested, we can identify which is the k-value that works best for the data we are working with. In the figure, a value of k around 10 seems to best perform with the data.

NOTES ON KNN: In practice, K-Nearest neighbor is never used for image classification because it is very slow at test time and also because there is no information contained in the distance metrics of the pixels. Its typical applications range from text mining, to agriculture, finance, medical and facial recognition, since it is a simple algorithm based on value of K and the distance function and it is efficient on small datasets.

1.3 The simplest score function: a linear classifier (Support Vector Machine - SVM)

Let's define here a score function that maps the pixels of an image into a score assigned to each class. Let's define a training dataset of N example images xi (i=1...N) associated with a label yi belonging to the K labels of the N images (For every i an xi belonging to {x1,.... xN} has a label yi

belonging to {y1,....yN}). The dimension of the images is D = 32 x 32 x 3 = 3072 pixels, and K, as for CIFAR10, is 10 (ten classes: dog, tier, etc..) . Imagine that our training dataset is made of N=50000 images, and define now a function f that maps the 3072 pixels of one image into one of the K labels.

Formally, the function will be:

$$f(x_i, W, b) = Wx_i + b$$

where in this equation all pixels of xi are flattened in the shape [Dx1], and W, the matrix of weights of size [KxD] and the vector b of size [Kx1] are the parameters of the function f. Often, the parameters in W are called **weights**, and b is the **bias vector** influencing the output scores. We can illustrate the score function with the following diagram:

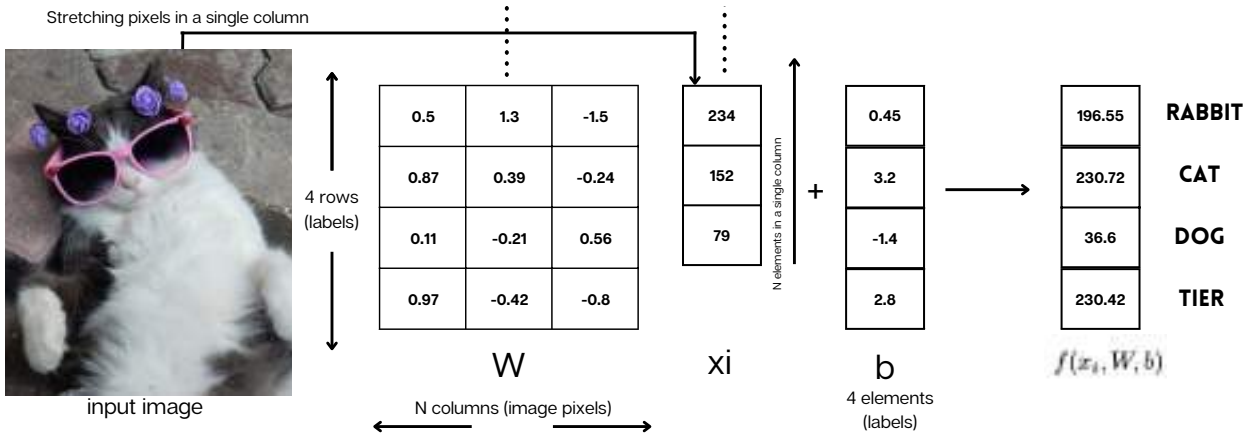


Figure 1.9: Representation of a linear classifier. Figure inspired by Karpathy's example in Stanford University's CS231n course.

How does the linear classifier work? For each class, it calculates a score which is the result of the sum of the products of the weights of each row by the normalized pixel values. During the training, we optimize the weights for the task of recognizing the specific label assigned to their column. An example of the weights (one column of the W matrix) of a trained linear classifier for the CIFAR10 is given in the figure 1.10. It is interesting to notice that for some labels we can recognize features that we expect for them, like for the horse, we see two horse necks, and for ship and plane, there's a dominance of blue pixels, due to the sea and the sky, respectively.

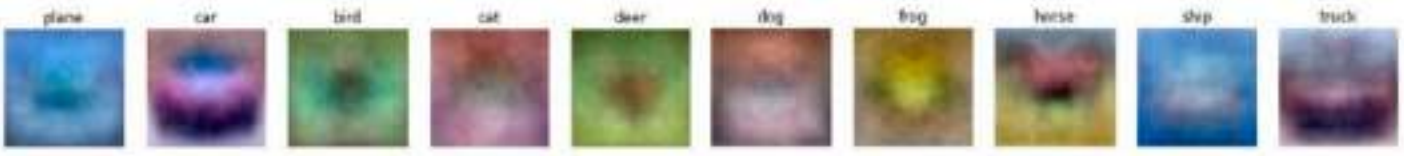


Figure 1.10: Example of weights optimized for the identification of the CIFAR10 classes. Figure from the course https://cs231n.github.io/classification/

NOTE: A frequent way to simplify the parameters W and b is to combine them into a single matrix that holds both and expanding the vector xi of an additional dimension holding the constant value 1, the default bias dimension. The new score function would then look like this:

$$f(x_i, W) = Wx_i$$

where now xi is [3073x1] and the new W matrix is [10x3073].

NOTE: In the diagram of Figure 1.10, we used the original image pixel values as input, therefore for the image xi you can see values ranging from 0 to 255. It is a good practice in machine learning to preprocess the input features (in our case, the image pixels) the input data, by centering them around their mean and by normalizing the values between -1 and 1. We will come back to this point, generally named regularization of the input data.

1.4 The loss function

How do we get the best weights in the W matrix for the classification? Before, we stated that the loss function establishes how good the classifier is. What is the form of the loss function? If we have N examples of a images xi and their corresponding labels yi, the loss function over this dataset of N elements is given by the sum of the losses Li over each of the single examples plus a regularization term R(W):

$$L = \frac{1}{N}\sum_{i=1}^{N} L_i(f(x_i, W), y_i) + \lambda R(W)$$

where f(xi, W) is the result of the scoring function (the predicted label) and yi is the so-called “ground truth”, i.e. the label associated with the xi image of the dataset. In general, excluding when we are overfitting, the smaller is the loss and the better is the classifier doing the job of classifying images. If the loss is large instead, we will need more work to increase the accuracy of the classification, i.e. we will need to better tune the weights through the optimization process. The **regularization term** R(W) is a term that helps to control the capacity of our model to generalize to unseen data. The parameter lambda is another hyperparameter, called the **learning rate**, that is essential for the learning task.

Regularization: a couple of details more. To expand on the regularization term, it is important to say that it helps the model to get a correct classification on data points on which it was not trained on. If we remove the regularization term from the loss function, the classifiers will tend to overfit the training data, and will lose its ability to generalize to test data or other datasets. However, also having too much regularization can become a problem, because it might cause underfitting, i.e. that the model does not perform well on the training data, and thus has difficulties in modelling the relations between the input images and the output labels. Figure 1.11 shows examples of underfitting and overfitting for a given set of data points.

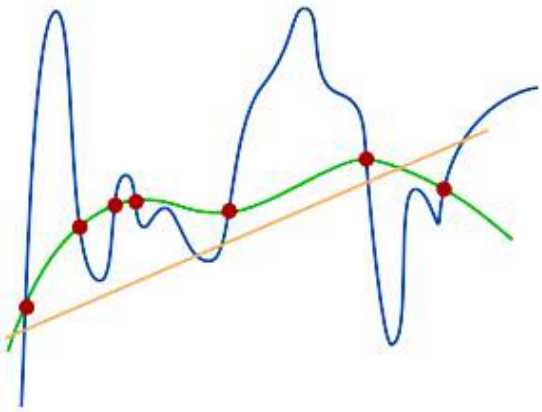


Figure 1.11: Example of underfitting model for the red data points in orange, an overfitting model is shown in blue and generalizing function in green. Figure from Deep learning tutorial from Adrian Rosebrock

There are multiple forms for the loss function. Here, we will introduce:

- the Multiclass SVM loss (Hinge loss)
- Softmax classifier (cross entropy loss)

Multiclass SVM loss (Hinge loss)

Let's introduce the form of the Hinge loss function, also called Multiclass SVM loss because it is a generalization to more categories than just 2. Indicating the result of the score function s = f(xi, W), we can write the Hinge loss for the sample {xi, yi} as:

$$L_i = \sum_{i \neq j} \begin{cases} 0 & \text{if } s_{y_i} \geq s_j + 1, \\ s_j - s_{y_i} + 1 & \text{otherwise} \end{cases}$$

s_{y_i}: score of the correct category

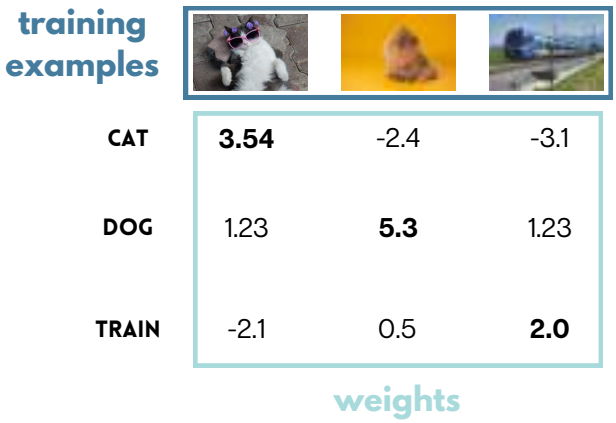
s_j: score of the incorrect category

The hinge loss function sums on all the incorrect classes j (j different than i), excluding the correct category; it compares the score of the correct category and the score of the incorrect category. If the score for the correct category is greater than the score for the incorrect category by some safety margin (that is set to 1), then we get a loss of zero. Otherwise, the loss it will return the value of the difference. Summing up over all the losses of the image (see definition of L given above) will give the final loss for the single example in the dataset (remember to include the regularization term).

The SVM loss can be re-written in a more compact form as:

$$L_i = \sum_{i \neq j} max(0, s_j - s_{y_i} + 1)$$

Example:



L(im_1) = max(0,1.23-3.54+1) + max(0,-2.1-3.54+1) = 0

L(im_2) = max(0,-2.4-5.3+1) + max(0,0.5-5.3+1) = 0

L(im_3) = max(0,-3.1-2+1) + max(0, 1.23-2+1) = 0.23

Loss over the full dataset:
L = L(im_1) + L(im_2) + L(im_3) = 0.23

Figure 1.12: Example of loss function calculation for SVM loss.

Softmax classifiers (cross-entropy loss)

Like in the previous case, we start from a function that, given the input image, assigns a score for each of the output labels. In the softmax classifier, we interpret the scores as unnormalized log probabilities for each of the labels. Given that the probability of a given label to be k given the input image xi is

$$P(Y = K|X = x_i) = \frac{e^{s_k}}{\sum_j e^{s_j}}$$

Since the loss function should minimize the negative log likelihood of the correct class, Li is:

$$L_i = -logP(Y = y_i|X = x_i) = -log(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}})$$

and therefore, we can write the **cross-entropy loss** as:

$$L_i = -log(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}})$$

Note that what is called the softmax function is the argument of the log, i.e. the exponentiation and normalization of the correct score with the sum of all the scores

Exercise: can you calculate the loss function using the training example presented before, if the chosen loss function is the cross-entropy loss function?

General approach: putting all the pieces together

A more general approach to the task of image classification can be described using three main components, that are a score function, which it maps the raw data into the class scores, a loss function, that quantifies the agreement between the predicted scores and the ground truth labels, and the regularization function, that determines the learning rate.

This approach has three main steps:

1) REPRESENTATION: We find a way to represent the classifier in a formal language the machine understands, i.e. we need to define the form our score function and train it.

2) EVALUATION. The score function assigns the score to each of the labels and the loss function establishes how good the classifier is and distinguishes the good from the bad ones.

3) OPTIMIZATION: This is a method that operates on the loss function and on the regularization function to search among the classifiers in the language of the highest-scoring one.

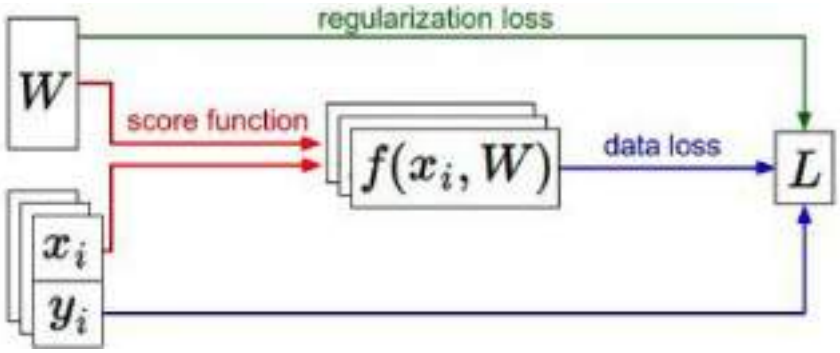


Figure 1.12: Example of the full approach representation. Figure from by Karpathy's example in Stanford University's CS231n course.

Before moving on, we want to introduce a way to represent the linear classifiers and all what we have been talking until now, which is the computational graph. Here below, we represent a linear classifier with the loss function we just described.

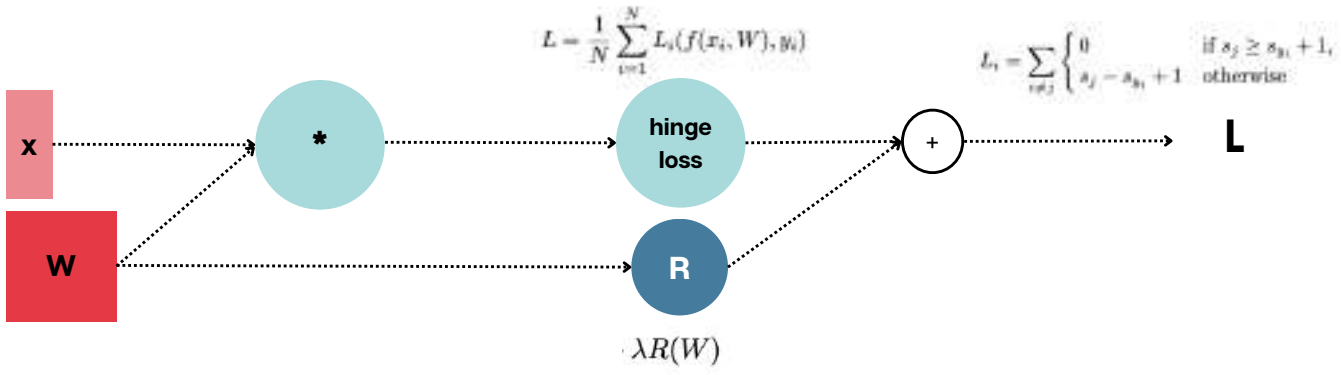


Figure 1.13: Example of computational graph applied to the linear classifier and to the SVM loss function.

1.5 The optimization process

The remaining question is "How do the get the best weights W?". The optimization process is the one that will do this job by finding the weights that minimize the loss function. Typically, the loss function is a multi-dimensional function and the minimization of such a complex is not easy. One can try different approaches:

- **Random search**: you can try different random weights and see which one of them works best. Typically, the random approach gives very bad results, but it might be useful for iterative refinement, i.e., since finding the best set of weights W is very difficult, one can start with a random set and then proceed to iteratively refine them, improving them at each step.
- **Random local search**: you can start with a random search and then apply perturbations to that, updating only if the loss of the perturbed state is lower than the original one. In this case the accuracy is slightly better than before, but still far from being acceptable.
- **Following the gradient**: for multidimensional function, the gradient identifies the direction of the steepest change. Thus, it can be identified as the direction where to go for updating the weights. In one dimension, this is just the derivative of the function in that point, while in multiple dimensions we need to consider the vector gradient, whose components are the partial derivatives with respect to each dimension. The slope in any direction is given by the dot product of the specified direction with the gradient. The steepest descent is the one with the most negative gradient.

Computing the gradient. There are two ways to compute the gradient: the easy but approximate way is to do it numerically using the finite differences, otherwise you one can use the calculus, i.e. derive the analytical expression of the derivative of the loss function with respect to the weights. This second method is exact but it is more prone to errors.

Numerical calculation. For the numerical calculation, one can implement in a code the formulation of the gradient:

$$\frac{\partial f}{\partial x} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

where x is a vector the h is an array of increments, and f is an array of partial derivatives. The code iterates over all dimensions, adds a small change h on that dimension and calculates the partial derivative of the loss function on that direction to see the change in the function. This formula can calculate the gradient at any point and for any function and the slope of the loss function along each dimension, given by the gradient, can be used to calculate an update in the weights. While this gradient is simple to compute, is it an approximate estimation of the steepest gradient and it is very computationally demanding.

Calculus. In this way, we can derive the exact expression of the gradient of L with respect to the weights and it is fast, but the calculus is prone to errors

Gradient descent and stochastic gradient descent (SGD). Usually, the way to proceed is to calculate the gradient analytically and then check with numerical gradient.

$$\nabla_W L(W) = \frac{1}{N}\sum_{i=1}^{N} \nabla_W L_i(f(x_i, W), y_i) + \lambda \nabla_W R(W)$$

The procedure of iteratively calculating the gradient and then performing a parameter update is called gradient descent. This is one of the most used ways of optimizing the loss function in neural networks. However, calculating the gradient for the entire loss function is expensive when N is large, so one way to approximate this step is to approximate the sum over N to a sum over a subset of the full total of the examples, i.e. on a minibatch of 32/64/128 elements of the sum. When we have millions of examples like it happens for convnets, a typical batch can contain 256 examples. The batch is then used to perform an update. When the minibatch comprises only one example, we talk about **stochastic gradient descent (SGD**).

1.6 Backpropagation and computational graphs

As we previously stated, we want to calculate the gradient of the loss function with respect to its weights:

$$\nabla_W L(W)$$

and we said that for doing this, we need to derive the analytic gradient of the loss function with respect to the weights. Backpropagation is a way to calculate gradients of given functions, by applying recursively the chain rule. The chain rule, in calculus, is the rule that allows to express the derivative of the composition h of two derivable functions f and g in terms of the derivatives of the two functions:

$$h(x) = f(g(x))$$

so:

$$h'(x) = f'(g(x))g'(x)$$

For neural networks, the function h will be the loss function and the input x will be the training data, dependent on the weights. Backpropagation is as essential component of machine learning and it is the method that practically allows the algorithm to learn.

Backpropagation consists of two main parts:

- Forward pass: the input goes through the network and provides a prediction.

- Backward pass: from the calculation of the gradient of the loss function at the final layer, recursively by applying the chain rule weights get updated in the network.

To show how practically backpropagation works, we build an easy example, with the function:

$$f(x) = (x + y)z$$

where we can define q = x + y, and it will become:

$$f(q, z) = qz$$
$$q = x + y$$

Based on these definitions, we will have:

$$\frac{\partial f}{\partial q} = z \qquad \frac{\partial q}{\partial x} = 1$$
$$\frac{\partial f}{\partial z} = q = x + y \qquad \frac{\partial q}{\partial y} = 1$$

and if we construct the same graphical representation as done for the linear classifier, where the forward pass computing values from the input to the output is in green and the backward pass is in red, we will get:

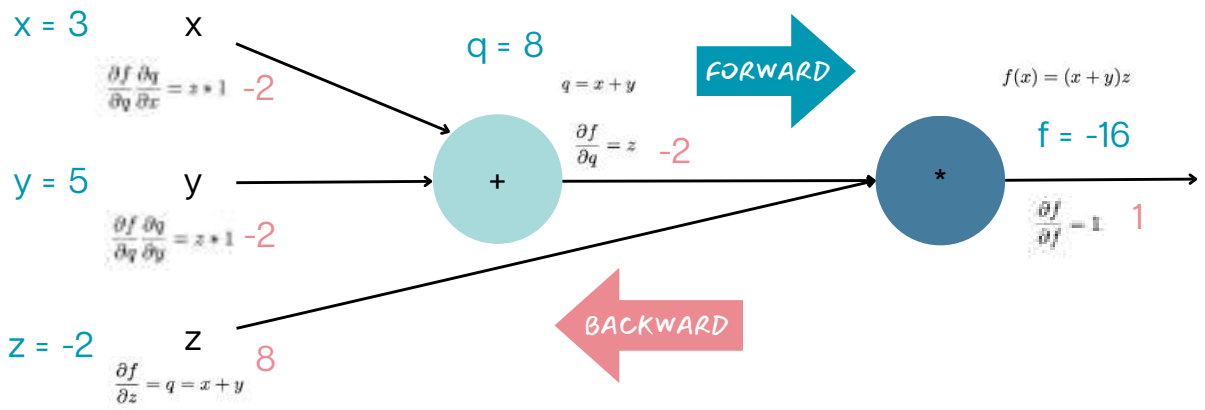


Figure 1.14: Representation of forward and backpropagation for a very simple function f(x, y, z). Figure reconstructed from the lecture series of the Stanford University's CS231n course.

It is important to note that the derivative of each variable contains information about the behavior of the whole function on that particular value, i.e. it is a local information. Every gate in the diagram gets an input and, based on that, it can provide the output and the gradient of its output with respect to its inputs and this operation do not depend on the rest of the circuit in which the gate is located. After the forward pass, during backpropagation, because of the chain rule, the gate multiplies the gradient of the loss function with respect to its output for the local gradients of its inputs, and passes it backwards to its inputs. This is better represented in figure 1.15:

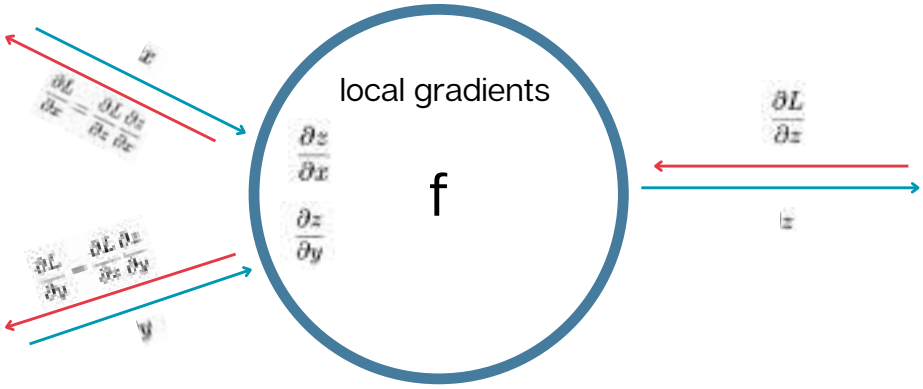


Figure 1.15: Representation of the behavior of a gate, Figure reconstructed from the lecture series of the Stanford University's CS231n course.

We can also analyze some patterns in backward propagation, that depend on the operations we do, for example:

- *Addition gate*: it distributes the gradient equally on both the input branches. For example, in figure 1.14 you can notice that the backpropagated gradient value of -2 in q is propagated unaltered to both the input x and y. It does so, independently of the values of the input gates.
- *Max gate*: a gate that in the forward pass transfers only the highest value of the inputs, in the backward propagation acts as a switcher, that means that it distributes the gradient unaltered to only one of its inputs, specifically the one that had the highest value during the forward pass.
- *Multiplication gate*: it is the hardest to interpret, we can just say that its local gradients are the the input values switched, see for example the backward pass in figure 1.14 at q and z: it is exactly 8 (equal to q) in z, and -2 (equal to z) in q.

Vectorized backpropagation. All what we presented for single variables can be extended in the same way to matrix and vector operations. If you are interested in digging more into this, take a look at the work of Erik Learned-Miller or to this paper from Atilim Gunes Baydin.