Using a convolutional neural network to classify the Street View House Number dataset

Andrea Ferretti
andrea.ferretti1@studenti.unimi.it

Neural networks

Neural networks are a family of predictors characterized by the combination of simple computational units, called neurons. A neuron typically performs the following computation: $g(\boldsymbol{x}) = \sigma(\boldsymbol{\omega}^T\boldsymbol{x})$, where the elements of the vector $\boldsymbol{\omega}$ are the parameters of the neuron, σ is a non-linear function called activation function, and $\boldsymbol{x}$ is the vector $x_0, \dots, x_n$ where $x_0 = 1$ and x_i for $i \in \{i, \dots, n\}$ the output computed by another neuron. In the supervised learning setting, neurons are combined in a graph-like structure resulting in a computational network able to learn, by adjusting the parameters $\boldsymbol{\omega}$ of each neuron, the underlying mapping between data points and their corresponding labels.

One of the most basic architecture a neural network can assume is called feedforward network. In this type of network neurons are organized into successive layers: one input layer, one or more hidden layers, and one output layer. The computation flows from the input layer towards the output layer and each neuron passes its output to all the neurons in the next layer. The i-th neuron in the input layer simply outputs the value of the i-th dimension of the data point. The neurons in the other layers compute a non-linear function of the weighted sum of the outputs of the neurons in the previous layer (plus a bias), every neuron has therefore his own vector of weights and a bias term. The funciton computed by the neurons in the hidden layers is called the activation function and usually is some kind of sigmoid function such as the logistic function:

$$S(x) = \frac{1}{1 + e^{-x}} \tag{1}$$

The neurons in the output layer compute a function that varies depending on the type of problem: for a regressor the identity function can be used to obtain a real valued output for each neuron, for a classifier, instead, the softmax function is typically used. The softmax function $softmax : \mathbb{R}^m \to \mathbb{R}^m$ is defined as:

$$softmax(\boldsymbol{v})_i = \frac{e^{\boldsymbol{v}_i}}{\sum_{t=0}^{m} e^{\boldsymbol{v}_t}}, \text{ for } i = 0, \ldots, m-1 \text{ and } \boldsymbol{v} \in \mathbb{R}^m \tag{2}$$

By having a network with as many output neurons as the number of possible classes to be predicted a probability distribution over those classes is obtained using softmax. It is to note that, in order for this function to be computed, every output neuron needs to know the values of the other output neurons (the various $\boldsymbol{v}_t$) before the softmax is applied. Figure 1 gives an example of a feedforward neural network.

Consider the pair $(\boldsymbol{x}, y)$ consisting of a data point and its corresponding label, let $\hat{\boldsymbol{y}}$ be the output of the network, in order to assest the goodness of the prediction a non-negative loss function $\ell(y, \hat{\boldsymbol{y}})$ is used so that the greater its value the worse is the prediciton. In the case of a classification problem $y \in Y$, where Y is the ordered set of possible symbolic labels with $|Y| = m$, therefore, by having a network with m neurons in the output layer and using the $softmax$ function, the network prediction $\hat{\boldsymbol{y}}$ represents a probability distribution over Y given the data point $\boldsymbol{x}$. For conveniency of notation let y_i^* indicate the first element of Y for $i = 0$, the second element of Y for $i = 1$ and so on for all the elements of Y. This means that the i-th element of $\hat{\boldsymbol{y}}$ gives the probability that the label for the data point $\boldsymbol{x}$ will be y_i^* in the network prediction. In this case, as loss function, the categorical cross-entropy loss can be used and it is

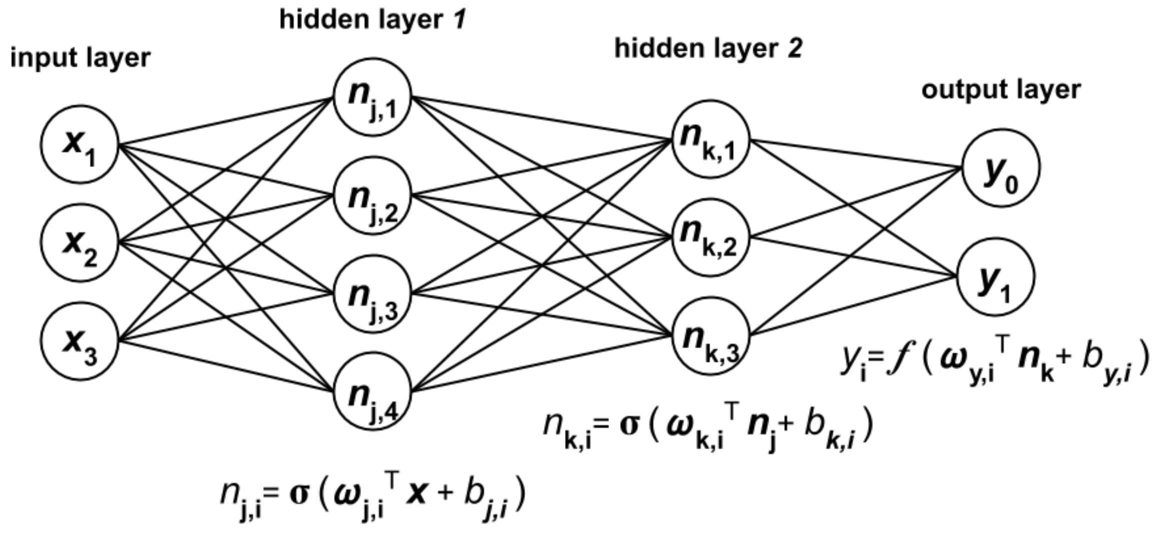

Figure 1: A feedforward network where on each neuron there is a label representing the value computed by it and passed to the nodes of the next layer through the arcs connecting them. A neuron assigns a weight ω to each incoming arch and performs the shown linear combination. The sigmoid funciton of equation (1) can be used as the activation function σ. The function computed by the neurons in the output layer could be $softmax$ (2) where $\boldsymbol{v}_i = \boldsymbol{\omega}_{y,i}^T \boldsymbol{n}_k + b_{y,i}$. Here the bias b is explicited in the computation rather than adding another dimension to the various vectors $\boldsymbol{\omega}$, $\boldsymbol{n}$, and $\boldsymbol{x}$ so that $\omega_{r,i,0} = b_{r,i}$ for $r \in \{j, k, y\}$ and $x_0 = n_{p,0} = 1$ for $p \in \{j, k\}$.

defined as:

$$\ell(y, \hat{\boldsymbol{y}}) = -\sum_{i=0}^{m-1} P(y = y_i^*) \log(\hat{y}_i) \tag{3}$$

where $P(y = y_i^*)$ is the probability that the true label for the data point $\boldsymbol{x}$ is y_i^*. Since , in practice, to a single data point in the training set corresponds only one label with probability 1, equation (3) can be simplified to:

$$\ell(y, \hat{\boldsymbol{y}}) = -\sum_{i=0}^{m-1} \mathbb{I}(y = y_i^*) \log(\hat{y}_i) = -log(\hat{y}_i), \text{ where } i \text{ verifies } y_i^* = y$$

where $\mathbb{I}(E) \in \{0, 1\}$ is the indicator function of an event E, meaning that $\mathbb{I}(E) = 1$ if and only if E occours and $\mathbb{I}(E) = 0$ otherwise.

Given the loss function ℓ the network can be trained on the training data to minimize ℓ

with respect to the parameters of each neuron. This is typically done with the mini-batched stochastic gradient descent algorithm, where, at each timestep, a fixed number of examples creating a subset B_t of the training set is randomly selected and every parameter is iteratively updated following the rule:

$$\omega_{l,n,i} \leftarrow \omega_{l,n,i} - \eta_t \frac{1}{|B_t|} \sum_{b \in B_t} \frac{\partial \ell(y_b, \hat{\boldsymbol{y}}_b)}{\partial \omega_{l,n,i}}$$

where, as depicted in Figure 1, $\omega_{l,n,i}$ represents the i-th parameter of the n-th neuron in the l-th layer, and $\hat{\boldsymbol{y}}_b$ is the output computed by the network for the example (x_b, y_b) of the training set; η_t is the learning rate that determines the update size. To calculate efficiently the partial derivatives with respect to all the parameters ω the backpropagation algorithm is used. The algorithm uses the chain rule to compute the derivative of a composite function:

$$\frac{df(g(x))}{dx} = \frac{df(g)}{dg} \frac{dg(x)}{dx}$$

When using a sigomid as the activation function of the neurons, the networks can experience the so called vanishing gradient problem, in which the gradients of the sigmoids of several neurons become so small that the several multiplications of the chain rule cause, especially in deep networks, the update of the weight depending on those gradients to be almost non existant. This happens because the sigmoid function has small gradients for high absolute value inputs; one possible solutoin is to use the rectifier $f(x) = max(0, x)$ as the activation function, this allows the gradient to always be equal to 1 when the output of the neuron is non-negative.

For input data that presents a grid-like structure a specific kind of network architecture called convolutional neural network (abbreviated ConvNet) has been developed. Similarly to feedforward networks it retains the forward direction of the computation, but it adds different types of layers such as convolutional layers and pooling layers. A convolutional layer is composed of several filters (also called kernels), each one performs a spatial convolution applying itself over the input dimensions; the output volume produced by this layer is called a feature map. A filter is composed by a mutlidimensional vector where each element is a real number called weight, and a single bias. Applying a filter over a portion of the input volume consist in multiplying each element of the input with the corresponding weight, then all the products are added together, along with the bias term, and the sum is used as input to the non-linear activation function producing the final single-valued result. The filter is then slid along the dimenions of the input and the operations are repeated to form one channel of the feature map; stacking together the results of convolutions of multiple filters the complete feature map is obtained. Considering as an example the case of creating a convolutional layer for a RGB image, the input (the image) can be seen as a 2 dimensional volume with 3 channels. The filter will therefore be 3 dimensional and slide along the 2 dimensions of the input, and the result of convoluting the filter will be a 2 dimensional output. The size of the first 2 dimensions of the filter, who both affect how big the portion of the input captured by one applicaton is, can be chosen freely, but the third dimension needs to be the equal to the number of channels of the input, otherwise applying the filter would not be possible. In order for the sizes of the output dimensions to be equal to the input ones the input can be padded with zero valued elements. Adding more equally dimensioned filters to the layer increases the number of channels the output volume will have. The parameters of a convolutional layer are therefore given by the weights and biases of each filter. During training the network can learn which parameters allow filters to extract the most relevant features from the input.

The pooling layer function is to reduce the spatial size of a feature map, practically downsampling the convolutional layer output in order to decrease the total number of parameters and the computational burden of the network. It does so using a particular kind of filter, that returns the maximum value among the ones it's being applied on, sliding it across the dimensions of each input channel indipendently. The output produced has the same number of channels as the input but the size of the other dimensions is reduces. This way only the most dominant features detected by the convolutional layer are retained. Going back to the RGB image example, assuming that a convolutional layer used 10 filters and produced a feature map where the first and second dimensions are equal to 30, a pooling layer, whose filter is of size 2 on both dimension and slides across with a stride of 2 elements, outputs a volume of size 15 on both dimension and 10 channels.

A typical ConvNet combines a variable number of blocks composed of one or more convolutional layer followed by a pooling layer (it is to note that if the sizes of the feature map produced by the convolutional layer are already small, the pooling layer may be avoided). The last output volume of this chain is then flattened into a one dimensional vector, losing its grid-like structure, and connected to a feedforward network that handles the classification part. This allows to have the same loss function as a feedforward network and to similarly train the weights via stochastic gradient descent. Figure 2 shows one of the possible structures of a ConvNet.

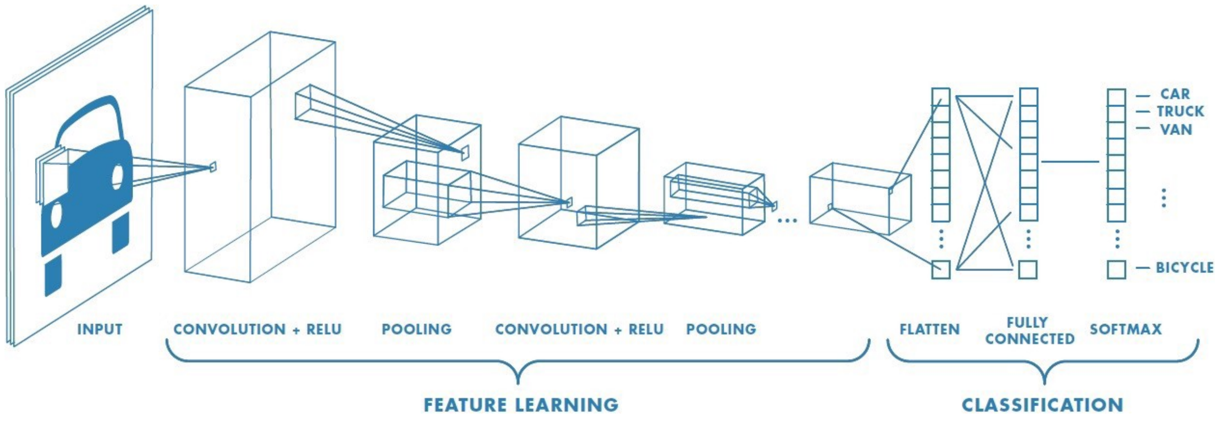

Figure 2: A visual representation of a convolutional neural network applied to image classification. The main components are: the convolutional layers producing the feature maps, the pooling layers used to downsample the convolutional layers output, and a standard fully connected feedforward network to handle classification

Regularization consists in setting costraints to the model in order to avoid overfitting of the training set. Several techniques have been developed: L2 regularization adds to the loss function a fraction of the L2 norm of the weights, penalizing models where some of the weights have a value much greater that the rest; this is done to increase the generalization capabilities by incouraging the model to use all the input values when making a prediciton, instead of having few input dimensions heavily influencing the prediction. Dropout consists in randomly (with a fixed probability) dropping neurons, along with their connections, from the network during training, preventing neurons from co-adapting too much. At test time all neurons are active and their outputs are scaled to match the output during training in expectation. Lastly batch normalization is seen as a normalization technique that normalizes the output of each neuron with respect to the values produced by that neuron for the batch that the network it's iterating over during training, the normalized value is then linearly transormed to give the data a new distribution. The parameters that determine this transofrmation are learnt throught training.

The Street View House Number dataset

Model training

Results