Problem solving in ML with NNs

Problems in ML involve some input data with a set of variables; the task is to predict some property that depends on the values of these variables. In supervised learning, these output values are known for a collection of samples (training set).

- The connection between input values and the corresponding predictions is some (usually complex) mathematical function or algorithm that needs to be found
- With neural networks, the algorithm is represented by layers of elements (neurons) performing simple computations
- Neuron parameters (weights and biases) are first initialized randomly, and iterated towards more appropriate values
- NN model is used to compute predictions for training samples; these are compared with the known true labels with a loss function (quantifies the difference)
- Comparison results are used to adjust the neuron parameters so as to decrease loss value
- Training is terminated when the loss value is low enough; after this, the model is tested with a separate collection of labeled samples (test set) that have not been used for training

Problem categories in machine learning

The various different kinds of problems studied in machine learning can be broadly divided in two main categories depending on what the model is supposed to predict:

- **Regression**
  In regression problems, the model predicts the value of a continuous quantity, typically represented by a real (floating point) number.
  Examples: Predicting the prize of a house from environmental data, tomorrow's temperature from available meteorological data, computer program's running time from its specifications …

- **Classification**
  In classification problems, the model predicts which of the (two or several) possible classes the input data sample belongs to by outputting the discrete label referring to that class.
  Examples: Predicting whether a given e-mail is spam or not (binary classification), identifying hand-written digits (multiclass classification) …

Supervised learning with Keras

Machine learning projects with Tensorflow/Keras involve going through the following steps:

- Arrange your training/test samples and corresponding labels into **NumPy arrays** of suitable shape. Some preprocessing is often necessary at this stage.
- Select the architecture of your model: number of hidden layers and the number of neurons in each layer. Choose an activation function for each layer depending on the type of your problem and the desired form of the predicted output. **Build** the model.
- Choose an appropriate loss function (again, depending on the type of your problem) for monitoring the performance of the model. Pick an optimizer to handle the updating of the model weights to decrease the loss score during training. **Compile** your model.
- Pick a value for mini-batch size, and **train** your model for some number of epochs.
- **Evaluate** your model with test data to find out how well the model generalizes; i.e. how it performs with previously unseen data.

Training procedure

After selecting the architecture of the network, and initializing its parameters, the network is trained. First, the input data is divided into groups (*batches*) of fixed size, and the following steps are repeated:

- Feed a batch of input-data elements through the network, whereby the corresponding predictions (output values) are computed.

- The accuracy of the predictions is quantified by computing the value of a **loss function** (also: objective function, cost function), which measures the difference between the predicted output values and the known target values (loss score).

- The parameters of the network are subjected to small modifications, which tend to decrease the loss score – and, therefore, produce better predictions next time.

- Proceed to the next batch and repeat the same steps. Having done so for all the batches constitutes one *epoch* of training. Between epochs the input data can be reshuffled to obtain batches of varying content.

Loss function: regression problems

In regression problems, a widely used loss function is the *mean-squared error* (MSE): the average square of the difference between the predicted outputs and the target outputs.

$$L_{\mathrm{MSE}} = \frac{1}{N}\sum_{i=1}^{N}(y_i - t_i)^2$$

The summation is taken over all the N samples in the batch under consideration. Sometimes an additional factor of two appears in the denominator.

y_i : the predicted output for the *i*th input sample

t_i : the target output for the *i*th input sample

Loss function: binary classification

In binary classification problems, each input sample is associated with one of two available classes (1 and 2). The target values of the samples are all either $t = 1$ (class 1) or $t = 0$ (class 2). The final output layer of the network usually has a sigmoid activation function, which outputs a number y between 0 and 1; this is interpreted as the probability of the sample to belong to class 1 (and, therefore, the probability of belonging to class 2 is $1 - y$).

Likelihood function for the data set:

$$p(\boldsymbol{t}|\mathrm{model}) = \prod_{i=1}^{N} \{{y_i}^{t_i}(1 - y_i)^{1-t_i}\}$$

The likelihood function itself is not a good choice for a loss function, since it is a product of sample terms. This product can be converted to a sum by taking the logarithm; multiplying by -1 ensures that the function is always positive.

Cross entropy loss
(negative log likelihood)

$$L_{\mathrm{CE}} = -\sum_{i=1}^{N} \{t_i \log y_i + (1 - t_i) \log(1 - y_i)\}$$

Loss function: multiple classes

In multiclass classification with K classes, the target output is a vector of length K; all the elements of the vector are 0 except for the one corresponding to the true class, which has value 1. For example, with $K = 5$ classes a sample belonging to class 2 would have a target vector $\boldsymbol{t} = (0,\ 1,\ 0,\ 0,\ 0)$. The last layer in the network model has softmax activation, ensuring that the elements of the predicted output vector for each sample are in the range from 0 to 1, and have a total sum of 1.

Categorical cross entropy loss for multiclass classification

$$\boxed{L_{\mathrm{CE}} = -\sum_{i=1}^{N}\sum_{j=1}^{K}\{t_{ij}\log y_{ij}\}}$$

y_{ij} : predicted probability of belonging to class *j* for the *i*th input sample

t_{ij} : target probability (either 0 or 1) of class *j* for the *i*th input sample

Derivative of a function of a single variable

Consider a function $y = f(x)$, which transforms a real input number x into an output number y:

$$\begin{tikzcd} x \arrow[r] & f \arrow[r] & y \end{tikzcd}$$

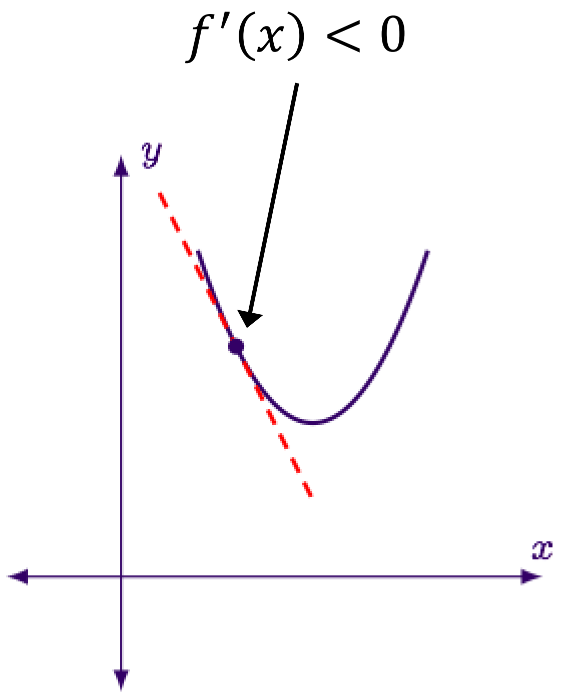
A small change Δx in the input value results in a small change Δy in the output value:

$$\begin{tikzcd} x + \Delta x \arrow[r] & f \arrow[r] & y + \Delta y \end{tikzcd}$$

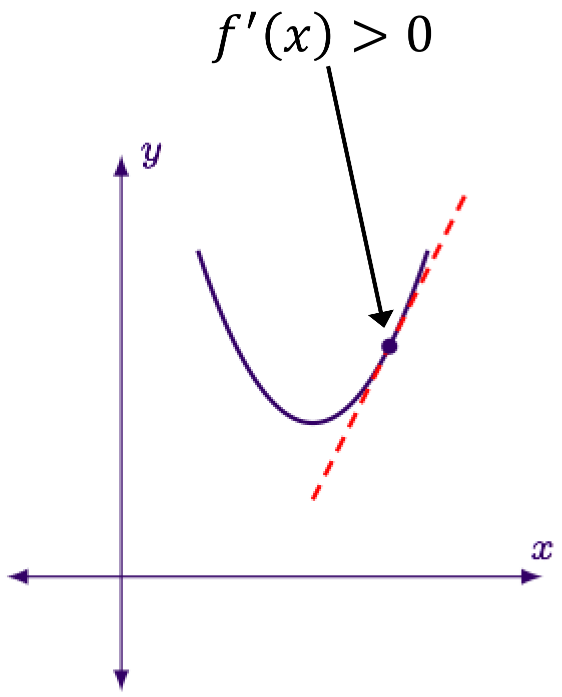
Derivative of the function $f(x)$ at x:

$$\frac{df}{dx} = f'(x) = \lim_{\Delta x \to 0} \frac{\Delta y}{\Delta x} = \lim_{\Delta x \to 0} \frac{f(x + \Delta x) - f(x)}{\Delta x}$$

Derivative: a graphical interpretation



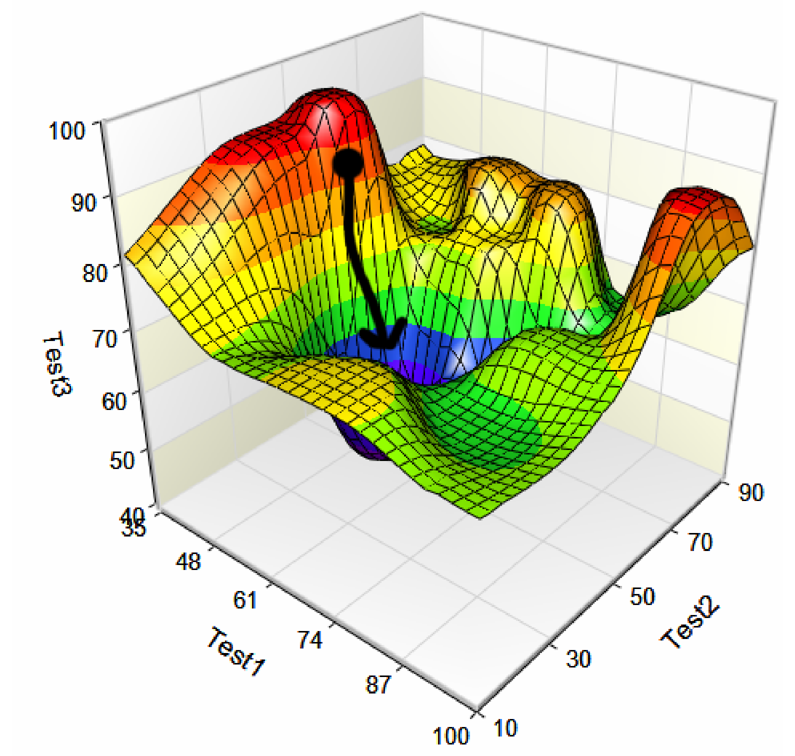
When the tangent line has a negative slope, the function is decreasing at that point.



When the tangent line has a positive slope, the function is increasing at that point.

Note: the function has a minimum at the point where its derivative vanishes. This point can be located by selecting a starting point and moving in the direction (positive or negative) indicated by the sign of the derivative (negative or positive, respectively).

Gradient descent



In the case of many-variable functions, there are derivatives with respect to every variable; the derivatives then define a vector in an n-dimensional space, and is called the gradient. At the minimum of a many-variable function, the gradient of the function vanishes. This point can be found by taking small steps in the n-dimensional landscape to the direction opposed to the local gradient vector. This method of locating the minimum of the function is called *gradient descent*.

$$\boldsymbol{x}^{(i)} \Rightarrow \boldsymbol{x}^{(i+1)} = \boldsymbol{x}^{(i)} - \varepsilon\, \boldsymbol{\nabla} f$$

It should be noted that this algorithm can end up at a *local* minimum of the function instead of the *global* minimum. The probability for this increases as the dimensionality of the space (number of variables) increases.

Gradient descent in machine learning

In machine learning, the function to be minimized is the chosen loss function of the neural network. The variables of this function are the parameters – all the weights and biases – of the model.

First, the parameters are initialized with some (typically small random) values. Then the gradient of the loss function is computed for a batch of data samples by finding the values of the partial derivatives with respect to the model parameters. The parameters are then updated by subtracting from them the corresponding component of the gradient, multiplied by a small number (the *learning rate*).

Weight update: $\boldsymbol{w}^{(i)} \Rightarrow \boldsymbol{w}^{(i+1)} = \boldsymbol{w}^{(i)} - \underset{\text{learning rate}}{\varepsilon} \, \boldsymbol{\nabla} L$

This procedure (with minor modifications depending on the choice of the optimizer) is repeated for a different batch of input values until the loss score is low enough, or until the predetermined maximum number of epochs is reached.

Training the model

For training, the loss score is computed for a certain number of samples each time, and the weight parameters of the model are updated according to the value of the gradient of the loss function.

- batch gradient descent: the loss is computed as an average over all the samples
- stochastic gradient descent: the loss is computed for a single sample at a time
- mini-batch stochastic gradient descent: the loss is computed with a subset of samples

Batch gradient descent is usually very expensive to use, and stochastic gradient descent with one sample only very noisy (wildly varying from one step to the next). An efficient compromise is a mini-batch of intermediate size.

In selecting the number of epochs, it should be realized that it is possible to train the model too long. At some point, the training produces overfitting, and starts to learn features specific to the training set only (after this, the model performs worse with unseen data).

Validating the model

In order to find out the performance of the model with unseen data, and to recognize overfitting, it is a good practice to select a part of the training set apart for validation. One rule-of-thumb possibility is to divide the entire set as follows: 60 % training, 20% validation, and 20% testing.

The validation samples are not used for the actual training of the model (for computing the gradients and performing weight updates). However, the loss value is computed for the validation set alongside with the training set, and the results can be used to monitor the performance of the model with unseen data – while the training is in progress.

The validation results can be used to see how long the model should be trained before it starts to overfit (at this point the validation loss begins to *increase*, even though the training loss keeps decreasing). All the training samples can then be trained from the beginning (including the validation samples) for the optimal number of epochs. After this, the model is ready for testing.