Hands-On Machine Learning: summary

May 12, 2020

Chapter 10: Artificial Neural Networks

Artificial neural networks (ANNs): Machine learning model inspired by networks of biological neurons. They are at the very core of Deep Learning: versatile, powerful and scalable.

0.1 From Biological to Artificial Neurons

Reasons to believe this wave of interest in ANNs is more profound:

- Huge quantity of data available to train & ANNs outperform other ML technologies in large & complex problems
- Tremendous increase in computing power since the 90s + GPU & cloud platforms
- Training algorithms have improved (by small tweaks with huge impact)
- Theoretical limitations have been mostly rare in practice
- ANNs seem to have entered a virtuous cycle of funding & progress

0.1.1 Biological neurons

Individual biological neurons seem to behave in a simple way, but are organized in a network of billions, each connected to thousands of others. Highly complex computations can emerge from their combines efforts. It appears that biological neurons are organized in consecutive layers, especially in the cerebral cortex.

0.1.2 Logical computations with neurons

The first artificial neurons had only binary inputs & one binary output. Each neuron's output is active if more than a certain number of inputs are. A network like this can compute complex logical expressions.

0.1.3 The Perceptron

The *perceptron* is a simple ANN architecture composed by a single layer of threshold logic unit neurons. Their inputs and output are now real numbers; each input connection is associated with a weight. The TLU computes a weighted sum of its inputs $\left(z = \boldsymbol{x}^{\intercal}\boldsymbol{w}\right)$ and then applies a *step function*. Its output is thus $h_{\boldsymbol{w}}\left(\boldsymbol{x}\right) = \text{step}\left(z\right)$.

A single TLU can be used for simple linear binary classification. The perceptron contains a layer of TLUs. each *fully connected* to all inputs and also containing a *bias* term. Ir can thus perform multi-output binary classification: $h_{\boldsymbol{w},i} = \phi\left(\boldsymbol{X}\boldsymbol{W} + \boldsymbol{b}\right)$, where ϕ is the activation function (step for TLUs), $\boldsymbol{X}$ is the input matrix $(n_{\text{instances}} \times n_{\text{features}})$, $\boldsymbol{W}$ is the weights matrix $(n_{\text{features}} \times n_{\text{neurons}})$, and $\boldsymbol{b}$ is the bias vector (n_{neurons}).

The training is done in a way to reinforce connections that help reduce the error. The perceptron recieves one training instance at a time, and for each wrong output neuron, reinforces weights from inputs that would have contributed to the correct prediction:

$$w_{i,j}^{\text{next}} = w_{i,j} + \underbrace{\eta}_{\text{learning rate}} \underbrace{\left(y_j - \hat{y}_j\right)}_{\text{0 or 1}} x_i$$

If the training instances are *linearly separable*, the algorithm is proven to converge to a (non unique) solution. Contrary to logistic regression, these perceptrons do not output a class probability.

Limitations of perceptrons (such as the XOR problem) can be eliminated by stacking multiple layers

0.1.4 The Multilayer Perceptron and Backpropagation

A MLP is comprised of an input layer, an output layer and several *hidden* layers. Each layer except for the output) is fully connected (layers close to the input are called *lower*).

The training algorithm is known as **backpropagation**: gradient descent with an efficient technique for computing gradients automatically (*autodiff*). In a forward and a backward pass, the algorithm is able to compute the gradient with respect to every model parameter.

Algorithm breakdown

1. Handles one mini-batch at a time, and goes through full training set multiple times (each pass is called an *epoch*).
2. For each instance in the mini-batch, the output is computed in a *forward-pass* (intermediate results from all layers are preserved since they are needed for the backward pass).
3. The output error is measured using the loss function
4. Then it computes each output connection's contribution to the error. This is done analytically using the chain rule.
5. It now calculates the error contributions from the weights on the layer below (also with the chain rule); this is repeated working backwards to the input layer
6. Finally, a gradient step is performed to update all weights

It is important that all weights be initialized randomly, or else the training will fail (a *symmetry breaking* is required).

In order for the algorithm to work, the step function must be smoothed to have a well-defined non-zero derivative, thus allowing GD to make some progress at every step. A first replacement is the sigmoid function $\sigma\left(z\right) = \dfrac{1}{1 + \exp\left(-z\right)}$

Other choices:

- $\tanh(z) = 2\sigma(2z) - 1$; similar to the logistic, but with output in $[-1, 1]$. This tends to make each output mostly centered around 0 at the beginning of training, which helps speed up the convergence.
- ReLU $(z) = \max(0, z)$; derivative jumps at $z = 0$, which can make GD bounce; but works well and is very fast to compute. Its unbounded image helps reduce some issues during GD.

Nonlinearities introduced by activation functions are essential to the complexity of the model. A large enough DNN can theoretically approximate any continuous function.

0.1.5 Regression MLPs

One output neuron per output dimension (*e.g.:* 2 for locating the center of an object in an image; another 2 for a bounding box (height and width)). Usually no activation at the output, except for ReLU/softplus $(z) = \log[1 + \exp(z)]$ to restrict it to positive images; or σ/tanh to bound it.

Training loss is typically MSE, or MAE/Huber (a combination of both) if there are many training outliers. The number of hidden layers is usually $\sim 1 - 5$, and the neurons per layer are $\sim 10 - 100$.

0.1.6 Classification MLPs

For each binary classification, a single output neuron with logistic activation is used, which can be interpreted as the estimated probability of the positive class.

For single output multiclass classification, an output neuron per class is needed, and a softmax activation for the whole layer (which estimates each class' probability)

The training loss function is generally multiclass cross-entropy. The rest of the architecture is broadly the same as with regression.

0.2 Implementing MLPs with Keras

Keras is a high-level Deep learning API with several backends. TensorFlow comes bundled with its own implementation, `tf.keras`, which has many advantages (*e.g.* TF's Data API to load and preprocess data efficiently).

0.2.1 Building an image classifier using the `Sequential` API

We use Fashion MNIST (70k grayscale 28×28 images, 10 classes). Since we will be using GD, we scale the input features.

Creating the model `model = keras.models.Sequential()` is the simplest model, for a single stack of layers connected sequentially
`model.add(keras.layers.Flatten(input_shape = [28,28]))` converts the image to a 1D array. As is the first layer, we must pass the `input_shape` (for a single instance)
`model.add(keras.layers.Dense(#neurons, activation = 'relu'))`

Instead of adding layers by one, it is possible to pass them as a list to the `Sequential` constructor.

0.2.1.1 Inspection

`model.summary()` displays the layers (by name), along with their parameters. `model.layers` returns them as a list, but they can also be called by name as `model.get_layer('dense')`. Its parameters can be retrieved by