Contents

Chapter 1

Introduction

Add a brief introduction to NumPyNet

1.1 Convolutional Neural Network

1.2 Super Resolution

Chapter 2

Layers

Layers are the main core of a Convolutional Neural Network: every one of them performs, on its respective input, a different operation. The concatenation of multiple Layer form a CNN, where for concatenation we mean that the output of a layer become the input of the next one.

In the NumPyNet framework a Layer is a python `class`, this allow the users to instantiate an object of the chosen type and call one of its methods.

Main Method:

- **forward** : this function is defined for every Layer and perform the so-called *forward pass*, that is the implementation of the transformation the Layer performs on the Input. It usually receives as argument just the output of the previous Layer
- **backward** : this function is defined for every Layer and perform the so-called *backward pass*, that is an implementation of the BackPropagation algorithm for the error. It computes the delta to be back-propagated during *training* and, eventually, the updates to trainable weights It usually receives as input only the global delta of the network, on which it performs a transformation, depending on the layer.
- **update** : this function is defined only for layers with trainable weights, updating them following a Gradient Descent with Momentum algorithm. It receives as parameters:
  - momentum :float, default = 0., scale factor of weight update
  - decay :float, default = 0., determines the decay of weights_update
  - lr :float, default = 1e-02, learning rate of the layer
  - lr_scale :float, default = 1., learning rate scale of the layer

2.1 Activation Layer

Activation functions (or transfer functions) are linear or non linear equations which process the output of a Neural Network neuron and bound it into a limit range of values (commonly $\in [0, 1]$ or $\in [-1, 1]$). The output of simple neuron[1] can be computed as dot product of the input and neuron weights; in this case the output values ranging from $-inf$ to $+inf$ and moreover it is just a simple linear function. Linear functions are very simple to trait but they are limited in their complexity and thus in their learning power. Neural Networks without activation functions are just simple linear regression model. Neural Networks are considered as *Universal Function Approximators* so the introduction of non-linearity allow them to model a wide range of functions and to learn more complex relations in the pattern data. From a biological point of view the activation functions model the on/off state of a neuron in the output decision process.

Many activation functions were proposed during the years and each one has its characteristics but not an appropriate field of application. The better activation function to use in a particular situation (to a particular problem) is still an open question. Each one has its pro and cons in some situations so each Neural Network libraries implements a wide range of them and it leaves the user to perform his own tests. In Tab. 2.1 we show the list of activation functions implemented in our library with mathematical formulation and its derivative. An important feature of any activation function, in fact, is that it should be differentiable since the main procedure of model optimization implies the backpropagation of the error gradients.

As can be shown in Tab. 2.1 it is easier to compute the activation function derivative as function of it. This is an (well known) important type of optimization in computational term since it reduces the number of operations and it allows to apply the backward gradient directly.

To better understand the effects of activation functions we can perform these functions on a simple test image and comment the results. This can be easy done using the example scripts inserted inside our library[2].

In Fig. **??** the effects of said functions are reported on a test image. For each function we show the output of the activation and its gradient. For visualization purposes the image values are rescaled $\in [-1, 1]$ before the input to the functions.

2.2 Average Pool Layer

Average pool layer explanation

[1] We assume for simplicity a fully connected Neural Network neuron.

[2] Aware of the author no other example implementations have been done. This makes the NumPyNet library a useful tool for neural network study.

Name	**Equation**	**Derivative**
Linear	$f(x) = x$	$f'(x) = 1$
Logistic	$f(x) = \frac{1}{1+\exp(-x)}$	$f'(x) = (1 - f(x)) * f(x)$
Loggy	$f(x) = \frac{2}{1+\exp(-x)} - 1$	$f'(x) = 2 * (1 - \frac{f(x)+1}{2}) * \frac{f(x)+1}{2}$
Relu	$f(x) = \max(0, x)$	$f'(x) = \begin{cases} 1 & \text{if} x > 0 \\ 0 & \text{if} x \leq 0 \end{cases}$
Elu	$f(x) = \max(\exp(x) - 1, x)$	$f'(x) = \begin{cases} 1 & \text{if} x \geq 0 \\ f(x) + 1 & \text{if} x < 0 \end{cases}$
Relie	$f(x) = \max(x * 1e - 2, x)$	$f'(x) = \begin{cases} 1 & \text{if} x > 0 \\ 1e - 2 & \text{if} x \leq 0 \end{cases}$
Ramp	$f(x) = \begin{cases} x^2 + 0.1 * x^2 & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases}$	$f'(x) = \begin{cases} f(x) + 1 & \text{if} x > 0 \\ f(x) & \text{if} x \leq 0 \end{cases}$
Tanh	$f(x) = \tanh(x)$	$f'(x) = 1 - f(x)^2$
Plse	$f(x) = \begin{cases} (x + 4) * 1e - 2 & \text{if } x < -4 \\ (x - 4) * 1e - 2 + 1 & \text{if } x > 4 \\ x * 0.125 + 5 & \text{if } -4 \leq x \leq 4 \end{cases}$	$f'(x) = \begin{cases} 1e - 2 & \text{if } x < 0 \text{ or } x > 1 \\ 0.125 & \text{if } 0 \leq x \leq 1 \end{cases}$
Leaky	$f(x) = \begin{cases} x * C & \text{if } x \leq 0 \\ x & \text{if } x > 0 \end{cases}$	$f'(x) = \begin{cases} 1 & \text{if} x > 0 \\ C & \text{if} x \leq 0 \end{cases}$
HardTan	$f(x) = \begin{cases} -1 & \text{if } x < -1 \\ +1 & \text{if } x > 1 \\ x & \text{if } -1 \leq x \leq 1 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{if } x < -1 \text{ or } x > 1 \\ 1 & \text{if } -1 \leq x \leq 1 \end{cases}$
LhTan	$f(x) = \begin{cases} x * 1e - 3 & \text{if } x < 0 \\ (x - 1) * 1e - 3 + 1 & \text{if } x > 1 \\ x & \text{if } 0 \leq x \leq 1 \end{cases}$	$f'(x) = \begin{cases} 1e - 3 & \text{if } x < 0 \text{ or } x > 1 \\ 1 & \text{if } 0 \leq x \leq 1 \end{cases}$
Selu	$f(x) = \begin{cases} 1.0507 * 1.6732 * (e^x - 1) & \text{if } x < 0 \\ x * 1.0507 & \text{if } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} x * 1e - 3 & \text{if } x0 \\ (x - 1) * 1e - 3 + 1 & \text{if } x > 1 \end{cases}$
SoftPlus	$f(x) = log(1 + e^x)$	$f'(x) = \frac{e^{f(x)}}{1+e^{f(x)}}$
SoftSign	$f(x) = \frac{x}{\lvert x\rvert+1}$	$f'(x) = \frac{1}{(\lvert f(x)\rvert+1)^2}$
Elliot	$f(x) = \frac{\frac{1}{2} * S * x}{1+\lvert x+S\rvert} + \frac{1}{2}$	$f'(x) = \frac{\frac{1}{2} * S}{(1+\lvert f(x)+S\rvert)^2}$
SymmElliot	$f(x) = \frac{S * x}{1+\lvert x * S\rvert}$	$f'(x) = \frac{S}{(1+\lvert f(x) * S\rvert)^2}$

Table 2.1: List of common activation functions with correspondig mathematical equation and derivative. The derivative is expressed as function of $f(x)$ to optimize their numerical evaluation.

2.3 Batch Normalization Layer

2.4 Convolutional Layer

Convolutional Neural Network (CNN) are particularly designed for image analysis. Convolution is the mathematical integration of two functions in which the second one is translated by a given value.

In signal processing this operation is also called *crossing correlation* and it is equivalent to the *autocorrelation* function computed in a given point. In image processing the first function is represented by the image I and the second one is a kernel k (or filter) which shift along the image. In this case we will have a 2D discrete version of the formula given by:

$$C = k * I$$

$$C[i, j] = \sum_{u=-N}^{N} \sum_{v=-M}^{M} k[u, v] \cdot I[i - u, j - v]$$

where $C[i, j]$ is the pixel value of the resulting image and N, M are kernel dimensions.

The use of CNN in modern image analysis applications can be traced back to multiple causes. First of all the image dimensions are increasingly bigger and thus the number of variables/features, i.e pixels, is often too big to manage with standard DNN[3]. Moreover if we consider detection problems, i.e the problem of detecting an set of features (or an object) inside a larger pattern, we want a system able to recognize the object regardless of where it appears into the input. In other words, we want that our model would be independent by simple translations.

Both the above problems can overcome by CNN models using a small kernel, i.e weight mask, which maps the full input. A CNN is able to successfully capture the spatial and temporal dependencies in an signal through the application of relevant filters.

[3] If we consider a simple image 224×224 with 3 color channels we obtain a set of $150'528$ features. A classical DNN layer with this input size should have 1024 nodes for a total of more than 150 million weights to tune.

Chapter 3

Network