Deep Learning

Pietro Marcatti

First Semester 2022/2023

1 History of Deep Learning

1.1 Perceptron

The perceptron algorithm was invented in 1958. The perceptron became the first model for binary classification. It has one weight w_i per input x_i. If the result is larger than a threshold it returns 1 otherwise 0 or -1 (non linearity ?). To train a perceptron we repeat the following steps:

- Initialize weigths randomly
- Take one sample x_i and predict y_i
- For erroneous predictions update weights
  - If prediction $y = 0$ and ground truth $y_i = 1$, increase the weights
  - If prediction $y = 1$ and ground truth $y_i = 0$, decrease the weights
- Repeat until no errors are made

However the perceptron can't solve a simple, although non-linear, problem such as the XOR. To improve on the perceptron model you must add new layers but there was a stagnation on the neural networks research. The stagnation was caused by a lack of motivation from the community due to the discouraging results of the first perceptron models. Still during the AI winter a couple important findings were published such as back-propagation and recurrent neural networks.

In 2009 the ImageNet dataset was published. It colleted images for each of the 100k terms in WordNet (16M images in total). Terms were organized hierearchally, es: Vehicle →Ambulance. The ImageNet challange was instituted: 1 million images, 1000 classes, top-5 and top-1 error measured. To build ImageNet they started colleting candidate images from the internet. They then classified the candidates with Amazon Mechanical Turk service.
A more recent important achievement was the one obtained by AlphaGo a deep learning model, based on reinforced learning, that in 2016 defeated the best Go player.

Deep learning is the first class of learning algorithms that is scalable: performance just keeps getting better as you feed them more data. Instead when working on a small amount of data the performance of a traditional learning model (logistic regression, SVM, decision tree etc) is better.
The three key factors for deep learning scaling are:

- Data
- Computation/hardware
- Algorithms

2 Logistic Regression

Let's start with a simple two feature model:

- x_1 number of lectures you attend
- x_2 hours spent on the laboratory activities

With logistic regression we want to learn a probabilistic function:

$$\hat{y} = P(y = 1|x)$$

In particular the goal is to find the parameters w and b of the following function (hypothesis).

$$H_{w,b}(x) = g = (w^T \cdot x + b) = \frac{1}{1 + e^{-(w^T \cdot x + b)}}$$

where $g(z)$ is the sigmoid function so that:

$$\begin{cases} H_{w,b}(x) \geq 0.5 & \text{if } y = 1 \\ H_{w,b}(x) < 0.5 & \text{if } y = 0 \end{cases}$$

To get our discrete classification we map the output of the hypthesis function as follow:

$$\begin{cases} H_{w,b}(x) \geq 0.5 & \rightarrow "1" \\ H_w, b(x) < 0.5 & \rightarrow "0" \end{cases}$$

The decision boundary is $H_{w,b}(x) = 0.5 \rightarrow w^T \cdot x + b = 0 \rightarrow -3 + x_1 + 2x_2$ supposing we have $b = 3$ and $w = [1, 2]$. The hypotesis function is > 0.5 when the argument is > 0, that is becuause of the shape and output of the sigmoid.

2.1 Cost Function

To find w and b so that:

$$\begin{cases} H_{w,b}(x) >= 0.5 & \text{if } y = 1 \\ H_w, b(x) < 0.5 & \text{if } y = 0 \end{cases}$$

the logistic classifirer defines the following cost function:

$$J(w,b) = \frac{1}{m} \cdot \sum_{i=1}^{m} Cost(h_{w,b}(x^i), y^i) \tag{1}$$

$$Cost(h_{w,b}(x^i), y^i) = -y^i \cdot ln(h_{w,b}(x^i)) - (1 - y^i) \cdot ln(1 - h_{w,b}(x^i)) \tag{2}$$

This cost function or loss function is convex and is derivable respect to w and b. In general we call the function to learn the **hypotesis** but in deep learning it's also called **model**, while the **cost function** in deep learning is also called **loss function**.

Gradient Descent

Gradient descent is a general algorithm for minimizing derivable functions and we are applying it to linear regression.
The gradient descent algorithm repeats until convergence the following update on weigths:

$$\Theta_j = \Theta_j - \alpha \frac{\partial}{\partial \Theta_j} J(\vec{\Theta_0})$$

It is always necessary to evaluate every new parameter before updating any of them as to not calculate the latters with the new values for the first ones.
The hyperparameter α is called the learning rate and it represents the size of the steps that we are taking down the direction of the gradient. For sufficiently small α the cost function should decrease on every iteration. This being said if α is too small it may take a big number of iteration to reach convergence. Finally, a value that is too big for α might cause the cost function to not decrease on every step because we might jump over the "dip", hence it might not converge. It is then a good idea to vary the parameter to experiment with the cost function and its convergence.

Application of gradient descent on logistic regression

Once we've established our decision function, in this example the logistic regression, we can procede to apply the gradient descent. To do that we first need to calculate the partial derivatives of the Loss function with respect to every parameter ($\vec{w}$ and $\vec{b}$) so that we can apply the changes. In this example with two inputs:

$$
\begin{aligned}
w_1 &:= w_1 - \alpha \frac{\partial J(\vec{w}, \vec{b})}{\partial w_1} \\
w_2 &:= w_2 - \alpha \frac{\partial J(\vec{w}, \vec{b})}{\partial w_2} \\
b &:= b - \alpha \frac{\partial J(\vec{w}, \vec{b})}{\partial b}
\end{aligned}
$$

Neural Networks

Neural Networks expand on the simple ideas of perceptron computing units by adding more layers on neurons betweeen the input and the output. As the shape of the neural network changes so does the way we represent it:

- $W^{[j]}$: its the matrix of weights controlling the function mapping from layer j-1 to layer j
- $b^{[j]}$: the value of the bias for the j-th layer
- $a^{[j]}$: activation vector for the layer j
- $g^{[j]}$: activation function for the layer j

Activation Functions

In deep neural networks you also have to specify what activation function you want to apply to the activation vectors of each hidden layer. In the hidden layers we rarely see the sigmoid function because it is said to kill the gradient. This refers to the fact that the sigmoid, for particularly large or small inputs, saturates at 1 or 0 respectively. The hyperbolic tangent also suffers from this vanishing gradient problem but it generally performs better than the sigmoid. The sigmoid function is still widely used in the output or final layer to give a probabilistic meaning to our output as it's betweeen [0,1].
What are other types of activation function that we can be used in the hidden layers?

ReLU or Rectified Linear Unit : It's particularly easy to train and achieves great performance. Its derivative is particularly easy to calculate, given that you consider the point of non-derivability in 0.

Leaky ReLU : Leaky ReLU was created to solve the dying ReLU problem which affects neuron with ReLU activation functions. If their parameter is stuck at 0 no matter the input their output will be 0. If this happens no gradient flows and there cannot be any learning. To avoid this it assigns a slightly sloped straight line for the negative values.

It is fundamental to use non-linear activation functions otherwise the output of the neural network will be a simple linear combination of the inputs. Let's suppose our activation function is the identity function: $g^{[1]}(z) = g^{[2]}(z) = z$

$$a^{[2]} = z^{[2]} = W^{[2]}a^{[2]}+b^{[2]} = W^{[2]}(W^{[1]}x+b^{[1]})+b^{[2]} = (W^{[2]}W^{[1]})x+(W^{[2]}b^{[1]}+b^{[2]})$$

We can define $W' = W^{[2]}W^{[1]}$ and $b' = W^{[2]}b^{[1]} + b^{[2]}$ then we can rewrite $a^{[2]} = W'x + b'$. Linear activation function is then useless unless in the last layer when performing a regression task.

Back propagation

Back propagation is the concept and algorithm that allows to update the weights of the neural network to learn. It relies on two simple steps:

Forward Step: in this step is activated on one example and the error of each neuron of the output layer is computed.

Backwards Step: in this step the network error is used for updating the weights. Starting at the output layer, the error is propagated backwards through the network layer by layer. This is done recursively computing the local gradient of each neuron.

Definition 2.1 (Jacobian Matrix) *The derivative of a vector function (a vector whose components are functions) with respect to an input vector x is called a Jacobian Matrix and is represented as:*

$$\frac{\partial y}{\partial x} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_n} \end{bmatrix}$$

It is particularly important to set a reasonable initial value for the weights. If we used 0 on all weights, all $z^{[1]}$ would be 0, hence $a_1^{[1]} = \ldots = a_n^{[1]}$ because we would always get the same mapping by the activation function. The derivaties used to obtain the gradient would be equal and so the weights and bias would all be changed by the same amount remaining all equal. We refer to this phenomenon as the symmetry problem and to counter it we can simply initialize the weights to a small random value. The reason for such a small value has to do with the previously mentioned vanishing problem. If your network starts working with big numbers it will get to the sigmoid in the output layer and be saturated on the 1 or 0 with very little gradient to move.

Hyperparameters

We've already referred to some of our variables as hyperparameters. We can identify an hyperparameter by considering if it is a variable that can be changed to impact the functioning and the performance of our model without it being a parameter such as the weights and the bias. Examples of hyperparameters are:

- Learning rate
- Number of interaction (or epochs)
- Number of hidden layers
- Number of hidden units in each hidden layers
- Activation functions

The process of tuning these values to achieve the best result is for us, right now, all empirical.

Improving our Neural Network

In the process of training our neural network we often divide all the data we have available in two or three sets. These are the training and test set, plus optionally, the validation set. The training set contains the data we use to perform the learning of our model. If available we then measure the performance on the validation set in real world mock-up scenario. Eventually, when our model is stable we test it on the test set to get a real measure of it's performance.
It is fundamental that validation and test have the same distribution to ensure meaningful learning.
We divide our training data set into smaller batches of usually around 16-32-64 samples. We compute forward and backwards on every single sample. Once we complete all the batches we completed an epoch. We can then start again but now, on the first batch, our neural network will have weights that will have already changed thanks to all other batches.

Overfit vs. Underfit

Regularization

Logistic Regression: Minimization problem with regularization: $\min_{w,b} J(w,b)$

$$J(w,b) = \frac{1}{m}\left[\sum_{i=1}^{m} Cost(h_{w,b}(x^{(i)}), y^{(i)}) + \frac{\lambda}{2}\sum_{j=1}^{n} w_i^2\right]$$

λ is called the regularization parameter, usually b is ignored in the regularization process. By setting a big regularization parameter we are saying that our minimization algorithm should focus on reducing the weights. The goal is to have the weights all in the same order of magnitude.

Doesn't regularization kill the importance of some features over others?

Regularization with Neural Networks:

$$J(W^{[1]}, b^{[1]}W^{[2]}, b^{[2]}) = \frac{1}{m}\left[\sum_{i=1}^{m}\mathbb{L}(\hat{y}, y^{(i)}) + \frac{\lambda}{2}\sum_{i=1}^{L}\left\|W^{[l]}\right\|_F^2\right]$$

Where λ is the regularization parameter, l is the layer.

$$\left\|W^{[i]}\right\|_F^2 = \sum_{i=1}^{n[l-1]}\sum_{j=1}^{n^{[l]}}(W_{i,j}^{[l]})^2$$

Where $W^{[l]} \in \mathbb{R}^{n^{[l]} \times n^{[l-1]}}$
Regularization helps preventing overfitting because by using a big value for λ we minimize weigths close to 0 and some of them are basically dead (almost 0)

Too many dead nodes means underfitting? Would it ever be useful to have a small λ

Normalization

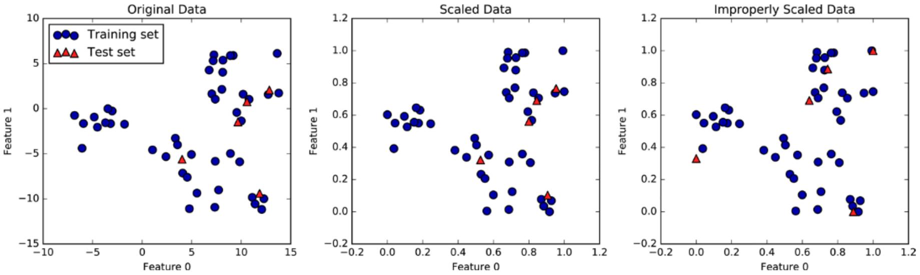
Mean normalization transforms dta to have a mean of zero and a standard deviation of 1. Such normalization is the preferred normalization for Neural Networks. Given a feature x_i the mapped feature is $x^i = \frac{x_i - \mu_i}{\sigma_i}$ where μ_i, σ_i are the mean and standard deviation computed on the training set. It is important to normalize each feature separately as its data come from different distributions and the feature number is exactly what the index x_i refers to.
The MinMaxNormalization shifts the data such that all features are exactly between 0 and 1.

$$x_i^{'} = \frac{x_i - \min x}{\max x - \min x}$$

It's very important to apply the same transformation to the trainin set and the test set for the supervised model to work on the test set. The figure illustrates what would happen if we were to use the minimum and range of the test set instead. The correct procedure is to normalize the training set and then once we have a stable model we can normalize also the test set with the same min and max values computed for the training set.

NLP and Word Embedding

Natural language processing is a field of deep learning in which you focus on recognizing elements of written text speech particles. Other fields of use are the summarization of text, question answering, automatic email response and forwarding, chatbots ecc ecc
Word embedding is a field of deep learning. It aims to represent a word as



feature vectors, and consequently as numbers. The simplest way is to assign each word of a vocabulary a number but this way you could sort words which doesn't map as a semantic characterstic for words. A different way is to use a one-hot vector for each words. These vectors are the size of the vocabulary and only for the word we want to represent we put 0 in the corresponding position, otherwise 0. This representation has some pros: it's simple and it doesnt imply ordering. The cons: huge vectors and no embedded meaning.