10 - Multilayer Perceptron

Ethan Graham

20th June 2023

Quick Reminder

Recall Logistic Regression

Logistic regression can handle a few outliers, but can't handle complex non-linear boundaries.

The question is now, how can we learn a function y s.t. $y(\mathbf{x}; \mathbf{w})$ is close to 1 for positive samples and close to -1 for negative samples?

- **Adaboost:** Use several hyperplanes
- **Forests:** use several hyperplanes
- **SVMs:** Map to a higher dimension
- **Neural Networks:** Map to a higher dimension and use lots of hyperplanes.

Reformulation of Logistic Regression

We can formulate logistic regression as a neuron. It takes $|\mathbf{x}|$ input size and 1 output size (class). The output value is $y(\mathbf{x}) = \sigma(\mathbf{w} \cdot \mathbf{x} + b)$

The idea with a multi-layer perceptron is that we do this many times, with all outputs connected to all the next inputs etc...

Example $\mathbf{h} = \sigma_1(\mathbf{W_1 x} + \mathbf{b_1})$, $\mathbf{y} = \sigma_2(\mathbf{W_2 h} + \mathbf{b_2})$

In this case here we have one hidden layer $\mathbf{h}$ and then one output layer $\mathbf{y}$

The output is a differentiable function of the weights. This is important.

We have several choices for the σ_i function (which can be different for every weight/layer). Commonly we chose:

- **Relu:** Simple and efficient gradient propagation, but if values fall into negative range then the neurons can become "dead" and stop the flow of gradient (halting optimization)
- **Sigmoid:** Clamps values between 0 and 1, but in extreme cases the gradient can be super small, which isn't optimal for optimization.

- **Softmax:** Is usually used at the output layer of the network, enabling the network to make multiclass predictions.

if the output layer of the network is of size 1, then the final matrix W is actually a vector w

PyTorch Neural Network Example

```
class MLP(nn.Module):
    def __init__(self, n1=10, nIn=2, nOut=1):
        self.l1 = nn.Linear(nIn, n1)
        self.l2 = nn.Linear(n1, nOut)

    def forward(self, x):
        h = sigm(self.l1(x))
        return sigm(self.l2(h))

    def loss(self, x, target):
        loss_fn = torch.nn.CrossEntropyLoss()
        output = self(x)
        return loss_fn(output_target)
```

There are better ways to do this without having to be explicit with everything though. Here `loss_fn` is equal to $-\sum t_n^k \, ln(p_n^k)$

Reminder Gradient Descent

The formula is defined as

$\mathbf{w}^{\tau+1} = \mathbf{w}^{\tau} + \eta \nabla f(\mathbf{w}^{\tau})$ Where η is the **learning** rate which much be carefully chosen

Stochastic Gradient Descent

$\mathbf{w}^{\tau+1} = \mathbf{w}^{\tau} + \eta \sum_{n \in B^{\tau}} \nabla E_n(\mathbf{w}^{\tau})$

Where B^{τ} represents a different randomly chosen set of indices for every iteration, known as a mini-batch.

Randomly choosing batches helps reduce the chances of falling into a local minimum (instead of a global one) and makes the computation possible on GPUs even when dealing with large datasets.

MLP with ReLU

Empirically, gradients tend to disappear when using sigmoid on values that aren't close to 0. ReLU tends to perform a lot better Each node defines a hyperplane, and the resulting function is piecewise linear affine and continuous. $\rightarrow$ piecewise : every *piece* of the function is continuous linear and affine.