Tutorial Neural Networks 2

Recommended Reading: Bishop chapters 5
(and optionally 9 for GMMs if desired)

1 the vanishing gradient

Problem 1: Why does the gradient vanish?

1 the vanishing gradient

Problem 1: Why does the gradient vanish?

Suppose we have a neural network $y = \boldsymbol{w}^{(2)}\phi(\boldsymbol{w}^{(1)}\boldsymbol{x})$ with one output, one layer of hidden units. From Gaussian MLE we define the loss $E = \frac{1}{2}\sum_n (y(\boldsymbol{x}_n) - z_n)^2$ over a learning data set $(\boldsymbol{x}_n, z_n)$.

Let's first look at the gradient w.r.t. the upper weights, those into the output neuron. For the weight from hidden neuron number i:

$$\begin{aligned}
\frac{\partial E}{\partial w_i^{(2)}} &= \frac{\partial \frac{1}{2}\sum_n \left(\boldsymbol{w}^{(2)}\phi(\boldsymbol{w}^{(1)}\boldsymbol{x}) - z_n\right)^2}{\partial w_i^{(2)}} && \text{(chain rule)} \\
&= (y(\boldsymbol{x}_n) - z_n)\frac{\partial \boldsymbol{w}^{(2)}\phi(\boldsymbol{w}^{(1)}\boldsymbol{x})}{\partial w_i^{(2)}} \\
&= (y(\boldsymbol{x}_n) - z_n)\,\phi_i(\boldsymbol{w}^{(1)}\boldsymbol{x})
\end{aligned}$$

So what about the weight from the input neuron i to hidden neuron j?

$$\begin{aligned}
\frac{\partial E}{\partial w_{ij}^{(1)}} &= \frac{\partial \frac{1}{2}\sum_n \left(\boldsymbol{w}^{(2)}\phi(\boldsymbol{w}^{(1)}\boldsymbol{x}) - z_n\right)^2}{\partial w_{ij}^{(1)}} && \text{(chain rule)} \\
&= (y(\boldsymbol{x}_n) - z_n)\frac{\partial \boldsymbol{w}^{(2)}\phi(\boldsymbol{w}^{(1)}\boldsymbol{x})}{\partial w_{ij}^{(1)}} \\
&= (y(\boldsymbol{x}_n) - z_n)\,\underbrace{w_j^{(2)}}_{\text{small}}\,\underbrace{\phi_j'(\boldsymbol{w}^{(1)}\boldsymbol{x})}_{\text{small}}\,x_i
\end{aligned}$$

Now let's analyse this result.

The residual, $(y(\boldsymbol{x}_n) - z_n)$, typically is small. But as it get smaller and smaller, the gradients gets smaller as it should—small error, small gradient.

The last term, x_i: well, if the input it small, the gradient can be small, as it really has no influence.

But the middle term: esp. the ϕ', for a sigmoid hidden unit—when it is saturated, its derivative is nearly 0. So it's a small term.

Then the $w_j^{(2)}$, a weight is usually a small value.

The above term means that the gradient is a small value.

In a lower layer, the output residual $(y(\boldsymbol{x}_n) - z_n)$ is repeatedly multiplied by the chain of weights leading into it, i.e. the residual at some neuron in layer k of K is $\prod_{l=k}^{K} w_{\cdot\cdot}^{(k)}(y(\boldsymbol{x}_n) - z_n)$. With all weights less than 1, this product can get rather small.

2 Regularisation

Problem 2: How can we prevent overfitting?

2 Regularisation

Problem 2: How can we prevent overfitting?

A normal way of regularisation is $L1$ or $L2$ regularisation, in which the weight sizes are added to the loss function. We change the loss function as follows:

$$E = \lambda_0 \sum_n \big(y(\boldsymbol{x}) - z_n\big)^2 + \lambda_2 \sum_i \sum_j \sum_k (w_{ij}^{(k)})^2$$

This L2 regularisation has the intuitive interpretation of heavily penalising peaky weight vectors and preferring diffuse weight vectors. This is a simple form of Tikhonov regularisation, also known as ridge regression.

In effect, the network to use all of its inputs a little rather that some of its inputs a lot. Also, during gradient descent parameter update, the L2 regularisation means that every weight is decayed linearly: $w+ = -\lambda w$ towards zero.

We can also do L1:

$$E = \lambda_0 \sum_n \big(y(\boldsymbol{x}) - z_n\big)^2 + \lambda_1 \sum_i \sum_j \sum_k |w_{ij}^{(k)}|$$

which is a soft version of LASSO. The L1 regularisation leads the weight vectors to become sparse during optimisation (i.e., very close to exactly zero). In other words, neurons with L1 regularisation end up using only a sparse subset of their most important inputs and become nearly invariant to the "noisy" inputs. In comparison, final weight vectors from L2 regularisation are usually diffuse, small numbers.

In practice, if you are not concerned with explicit feature selection, L2 regularisation can be expected to give superior performance over L1.

A stabler approach is doing a combination of the two:

$$E = \lambda_0 \sum_n \left(y(\boldsymbol{x}) - z_n\right)^2 + \lambda_1 \sum_i \sum_j \sum_k |w_{ij}^{(k)}| + \lambda_2 \sum_i \sum_j \sum_k (w_{ij}^{(k)})^2$$

so you can get a sparse model *and* get weight grouping.

But remember: dropout seems to surpass the above methods in effectivity. An intuitive reason may be the following: L1 or L2 regularisation pull the weights to 0. Dropout pulls the weights to what other weights think is right.

Problem 3: A convolutional network is, in fact, a neural network where weights are shared. After all, mapping a filter at various "places" of the image can be seen as multiplying many neurons, each having the same input weights.

Consider a neural network in which multiple weights are constrained to have the same value. Discuss how the standard backpropagation algorithm must be modified in order to ensure that such constraints are satisfied when evaluating the derivatives of an error function with respect to the adjustable parameters in the network.

Problem 3: A convolutional network is, in fact, a neural network where weights are shared. After all, mapping a filter at various "places" of the image can be seen as multiplying many neurons, each having the same input weights.

Consider a neural network in which multiple weights are constrained to have the same value. Discuss how the standard backpropagation algorithm must be modified in order to ensure that such constraints are satisfied when evaluating the derivatives of an error function with respect to the adjustable parameters in the network.

The modifications only affect derivatives with respect to weights in the convolutional layer. The units within a feature map (indexed m) have different inputs, but all share a common weight vector, w(m). Thus, errors ?(m) from all units within a feature map will contribute to the derivatives of the corresponding weight vector. In this situation, (5.50) becomes

$$\frac{\partial E}{\partial w_i^{(m)}} = \sum_j \frac{\partial E}{\partial a_j^{(m)}} \frac{\partial a_j^{(m)}}{\partial w_i^{(m)}} = \sum_j \delta_j^{(m)} y_{ij}^{(m)}$$

Here $a_j^{(m)}$ denotes the activation of the jth unit in the mth feature map, whereas $w_i^{(m)}$ denotes the ith element of the corresponding feature vector and, finally, $y_{ij}^{(m)}$ denotes the ith input for the jth unit in the mth feature map; the latter may be an actual input or the output of a preceding layer.

Note that $\delta_j^{(m)} = \partial E_n / \partial a_j^{(m)}$ will typically be computed recursively from the δs of the units in the following layer. If there are layer(s) preceding the convolutional layer, the standard backward propagation equations will apply; the weights in the convolutional layer can be treated as if they were independent parameters, for the purpose of computing the δs for the preceding layer?s units.

Problem 4: ⋆ This exercise is taken mostly from [1], chapter 5.5.7. The solutions are in the handwritten notes.

One way to reduce the effective complexity of a network with a large number of weights is to constrain weights within certain groups to be equal (*weight sharing*). However, this is only applicable to particular problems in which the form of the constraints can be specified in advance. Here we consider a form of *soft weight sharing* [2] in which the hard constraint of equal weights is replaced by a form of regularisation in which groups of weights are encouraged to have similar values. The division of weights into groups, the mean weight value for each group, and the spread of values within the groups are all determined as part of the learning process.

In general, we consider the distribution over the weights to be a *mixture of Gaussians*, i.e., for a given weight w_i:

$$p(w_i) = \sum_{j=1}^{N} \pi_j \mathcal{N}(w_i|\mu_j, \sigma_j^2)$$

(π_j are called the *mixing coefficients*) and weights are i.i.d:

$$p(\boldsymbol{w}) = \prod_i p(w_i)$$

Taking the negative logarithm then leads to a regularisation function of the form

$$\Omega(\boldsymbol{w}) = -\sum_i \ln \left(\sum_{j=1}^{M} \pi_j \mathcal{N}(w_i|\mu_j, \sigma_j^2) \right)$$

The total error function is thus given by

$$\widetilde{E}(\boldsymbol{w}) = E(\boldsymbol{w}) + \lambda \Omega(\boldsymbol{w})$$

where $E(\boldsymbol{w})$ is the usual standard error function (sum-of-squares, cross-entropy, ...) and λ is the regularisation coefficient. This error is minimised both with respect to the weights w_i and with respect to the parameters $\{\pi_j, \mu_j, \sigma_j\}$ of the mixture model. In order to evaluate the derivatives of the error function with respect to these parameters, it is convenient to regard the $\{\pi_j\}_j$ as prior probabilities and to introduce the corresponding posterior probabilities which are in the form (and you should prove that)

$$\gamma_j(w) = \frac{\pi_j \mathcal{N}(w|\mu_j, \sigma_j^2)}{\sum_k \pi_k \mathcal{N}(w|\mu_k, \sigma_k^2)} \tag{1}$$

Now, prove the following:

$$\frac{\partial \widetilde{E}}{\partial w_i} = \frac{\partial E}{\partial w_i} + \lambda \sum_j \gamma_j(w_i) \frac{(w_i - \mu_j)}{\sigma_j^2} \tag{2}$$

$$\frac{\partial \widetilde{E}}{\partial \mu_j} = \lambda \sum_i \gamma_j(w_i) \frac{(\mu_j - w_i)}{\sigma_j^2} \tag{3}$$

$$\frac{\partial \widetilde{E}}{\partial \sigma_j} = \lambda \sum_i \gamma_j(w_i) \left(\frac{1}{\sigma_j} - \frac{(w_i - \mu_j)^2}{\sigma_j^3} \right) \tag{4}$$

Note that in a practical implementation, new variables η_j defined by

$$\sigma_j^2 = \exp(\eta_j)$$

would be introduced, to ensure that σ_j remains positive, and thus the minimisation must be performed with respect to the η_j. Similarly, the mixing coefficients π_j are expressed in terms of a set of auxiliary variables $\{\phi\}_j$ using the softmax function given by

$$\pi_j = \frac{\exp(\phi_j)}{\sum_k \exp(\phi_k)}$$

The derivatives thus take the form (prove this too)

$$\frac{\partial \widetilde{E}}{\partial \phi_j} = \sum_i (\pi_j - \gamma_j(w_i)) \tag{5}$$

References

[1] C. M. Bishop: Pattern Recognition and Machine Learning. Springer, 2006.

[2] S. Nowlan and G Hinton: Simplifying neural networks by soft weight sharing. Neural Computation 4(4), 473-493, 1992.