Summary

sagisk

November 7, 2021

1 Chapter 3: Training Deep Neural Networks

1.1 Backpropagation:

Problem: With the increasing number of layers even for a small number of nodes a recursively nested function will result in a summation of forbidding number of recursively nested terms. For, example for a NN with 10-layers and only 2-nodes per layer there are 2^{10} recursively nested terms . A hidden layer often gets its input from multiple units, which results in multiple paths from a variable w to an output o. So, we have:

Input: w Hidden layers: $g_1(w), g_2(w), ..., g_k(w)$. Output layer: $f(g_1(w), g_2(w), ..., g_k(w))$.

So, $\frac{df(g_1(w),g_2(w),...,g_k(w))}{dw} = \sum_{i=1}^{k} \frac{df(g_1(w),g_2(w),...,g_k(w))}{dg_i(w)} \frac{g_i(w)}{dw}$

The generalization of this for any DAG is given as: Lemma (3.2.1): Pathwise Aggregation Lemma. For DAG let i-th node contians variable $y(i)$, $z(i,j) = \frac{dy(j)}{d(y_i)}$ - be the local derivative of the directed edge (i,j) and P be a set of non-null paths from variable w to o-the output of the graph. Then, $\frac{do}{dw} = \sum_{p \in P} \prod_{(i,j) \in p} z(i,j)$

Though the above lemma gives us a way to compute the derivative we again face the problem of going through all nodes of all paths and computing their product recursively to add up them. This is exponentially expensive operation.

Solution: The usage of an iterative approach i.e. dynamic programming and the chain rule of differential calculus. In this case each local gradient only worries about its specific input and output and one computes the gradient as a product of local gradients along the path which is the main reason that backpropagation is needed. Let $S(w,o) = \sum_{p \in P} \prod_{(i,j) \in p} z(i,j)$ and let $A(i)$ be a non-empty set of nodes that comes at the end points of outgoing edges from node i. Then for any indermediate node i (between w and o) we have $S(i,o) = \sum_{j \in A(i)} S(j,o) z(i,j)$. Hence, using dynamic updates we can start from $S(o,o) = 1$ and apply the compuation backwards from nodes incident to o (output).

So, in fact, we get that 'backpropagation'' is the same as path-aggregative form Lemma (3.2.1) just the compuations are applied in a particular order to minimize the complexity with the usage of dynamic programming.

In case of NN we have two choices as how to compute $z(i,j)$. 1) Post-activation where $z(i,j)$ is the product of the local derivative of the activation function at node j and the weight of the edge (i, j). 2) Pre-activation where $z(i,j)$ is the product of the local derivative of the activation function at node i and the weight of the edge (i, j).

1.2 Backpropagation with Post-Activation:

Forward phase: The input values cascade forward through the nn and all the intermediate hiddent and output variable are computed for the given input. Also the derivative of loss function with respect to the output (or outputs) node (nodes) is (are) computed i.e. $\frac{dL}{do}$. Backward phase: Going from backwards computes the gradient of the loss function w.r.t. various weights.

$\frac{dL}{dw(h_{r-1},h_r)} = \frac{dL}{do} [\sum_{[h_r,h_{r+1},...,h_k,o] \in P} \frac{do}{dh_k} \prod_{i=r}^{k-1} \frac{dh_{i+1}}{dh_i}] \frac{dh_r}{dw(h_{r-1},h_r)} = \frac{dL}{dh_r} \frac{dh_r}{dw(h_{r-1},h_r)} = \Delta(h_r,o) \frac{dh_r}{dw(h_{r-1},h_r)}$

Here, $\Delta(h_r,o) = \frac{dL}{dh_r}$ is very similar to the $S(i,o) = \frac{do}{dy_i}$.

1. Compute the $\Delta(o,o) = \frac{dL}{do}$.

2. Propagate backwards: $\Delta(h_r, o) = \frac{dL}{dh_r} = \sum_{h:h_r \to h} \frac{dL}{h} \frac{dh}{dh_r} = \sum_{h:h_r \to h} = \Delta(h, o) \frac{dh}{dh_r}$. Since, h is in later layer then h_r we have that $\Delta(h, o)$ has already been calculated. Also, $a_h = h_r w(h_r, h)$ then, $\frac{dh}{dh_r} = \frac{dh}{da_h} \frac{da_h}{dh_r} = \Phi'(a_h) w(h_r, h)$

Finally, putting all together: $\Delta(h_r, o) = \sum_{h:h_r \to h} \Phi'(a_h) w(h_r, h) \Delta(h, o)$

1.3 Backpropagation with Pre-Activation:

$\frac{dL}{dw(h_{r-1}, h_r)} = \frac{dL}{do} \Phi(a_o) [\sum_{[h_r, h_{r+1}, ..., h_k, o] \in P} \frac{da_o}{da_{h_k}} \prod_{i=r}^{k-1} \frac{da_{h_{i+1}}}{da_{h_i}}] h_{r-1} = \frac{dL}{da_{h_r}} h_{r-1} = \delta(h_r, o) h_{r-1}$

$\delta(h_r, o) = \frac{dL}{da_{h_r}} = \sum_{h:h_r \to h} \frac{dL}{da_h} \frac{da_h}{da_{h_r}} = \sum_{h:h_r \to h} \delta(h, o) \frac{da_h}{da_{h_r}}$

$\frac{da_h}{da_{h_r}} = \frac{da_h}{dh_r} \frac{dh_r}{da_{h_r}} = w(h_r, h) \Phi'(a_{h_r})$

1.4 Shared Weights:

For shared weights as in autoencoders, cnn, rnn, let the weight w be shared at T different nodes and the corresponding copies of the weights at these nodes are $w_1, ..., w_T$. Then $\frac{dL}{dw} = \sum_{i=1}^{T} \frac{dL}{dw_i} \frac{dw_i}{dw} = \sum_{i=1}^{T} \frac{dL}{dw_i}$. So, we pretend that these weights are independent, compute their derivatives, and add them!

Tuning Hyperparameters: Common hyperparameters in NN are learning rate, weight of regularization and so on. Tuning phase: A phase to tune the hyperparameters on a validation set. Grid search: A set of values for each hyperparameter is selected and all combinations of the selected values of hyperparameters are tested to get the optimal choice. However, is costly.

Randomly sample the hyperparameters from the grid range.

The models are not run till completion. Instead they run for a certain number of epochs and test the progress. Runs that are poor can be quickly killed.

Feature preprocessing: Mean-centering often is combined with standardization. If we want only non-negative values add the most negative entry of a feature to the corespondign feature values of each data point. Often is combined with min-max normalization. If we want the data to be in $(0, 1)$ let $\min_j, \max_j$ be the minimum and maximum of the j-th feature. Then, $x_{ij} = \frac{x_{ij} - \min_j}{\max_j - \min_j}$

Whitening: The idea is taht uncorrelated concepts in the data will have equal omportacne and nn can decide which of them to emphasize in the learning process. The axis-system is rotated to create a new set of de-correlated features, each of which is scaled to unit variance. Typically, principal component analysis is used to achieve this goal. Usually $k < d$ top eigenvectors are chosen to form a matrix P. Then we multiply our initial matrix DP to get whitening matrix U. In whitening, each column of U is scaled to unit variance by dividing it with its standard deviation

Initialization: One possible approach to initialize the weights is to generate random values from a Gaus- sian distribution with zero mean and a small standard deviation, such as 102. Typically, this will result in small random values that are both positive and negative. One problem with this initialization is that it is not sensitive to the number of inputs to a specific neurn. o balance this fact, each

weight is initialized to a value drawn from a Gaussian distribution with standard deviation 1/r, where r is the number of inputs to that neuron.

1.5 Vanishing and Exploding Gradient:

Set up: Assume we have $(m + 1)$ layers (including the input layer). The weights are $w_1, ..., w_m$. Sigmoid activation function $\Phi(.)$ is applied at each layer. Let x be input, $h_1, ..., h_{m-1}$ the hidden values in the various layer, o the output values. Then, by backpropogation: $\frac{dL}{dh_t} = \Phi'(h_{t+1}) w_{t+1} \frac{dL}{dh_{t+1}}$ The maximum value of sigomoid function f is 0.5 and hence the maximum value of its derivative is 0.25 since derivative is $f(1 - f)$. Vanishing gradient: Each weight update will cause $\frac{dL}{dh_t}$ to be 0.25 that of $\frac{dL}{dh_{t+1}}$. So, after about r layers this value will typically be less than 0.25^r. So, the earlier layers will get small updates compared to later layers. This problem is known as vanishing gradient problem. Exploding gradient: One fix of vanishing gradient is to make sure that we chose weights such that the product of the weight with the activation function's derivative will be ≥ 1. However, practically, it is almost impossible.

Partial Solution: The usage of ReLU activation function and the hard tanh. Since they have derivative of 1 or 0. Their problems are "Brain Damage". For example, if the input is nonnegative and all weights are initialized to negative values ReLU will not activate and will always gives 0 irrespective of its input.

1.6 Gradient-Descent Strategies:

Learning Rate Decay:

- Problem: A low learning rate used early will cause the to take too long to come close to an optimal solution. A large initial learning rate will let to come close to an optimal solution but then algorithm will oscillate a very long time or even diverge.
- Solution: Use decaying rates:
  1. Exponential decay: $a_t = a_0 e^{-kt}$
  2. Inverse Decay: $a_t = \frac{a_0}{1+kt}$

k-is hyperparameter.

Momentum-Based learning: $V = bV - a\frac{dL}{dW}$, b-is the friction hyperparameter. $W = W + V$

Nesterov Momentum: $V = bV - a\frac{dL(W+bV)}{dW}$ $W = W + V$ Using the value of the gradient a little further along the previous update can lead to faster convergence. In the previous analogy of the rolling marble, such an approach will start applying the "brakes" on the gradient-descent procedure when the marble starts reaching near the bottom of the bowl, because the lookahead will "warn" it about the reversal in gradient direction.

AdaGrad: Let A_i be the aggregate value for the i-th parameter. $A_i = A_i + (\frac{dL}{dw_i})^2$ $w_i = w_i + \frac{a}{\sqrt{A_i}}(\frac{dL}{dw_i})$ Scaling the derivative inversely with $\sqrt{A_i}$ is a kind of "signal-to-noise" normalization because A_i only measures the historical magnitude of the gradient rather than its sign; it encourages faster relative movements along gently sloping directions with consistent sign of the gradient.

RMSProp:

$A_i = \rho A_i + (1-\rho)(\frac{dL}{dw_i})^2$ $w_i = w_i + \frac{a}{\sqrt{A_i}}(\frac{dL}{dw_i})$ Does not keep all the history of the i-th parameter cause too old history can be no more useful. Instead uses exponential averaging to average values and not slow the progress by redundant history.

RMSProp with Nesterov Momentum.

$A_i = \rho A_i + (1-\rho)(\frac{dL(W+bV)}{dw_i})^2$ $v_i = bv_i - \frac{a}{\sqrt{A_i}}(\frac{dL(W+bV)}{dw_i})$, $b \in (0,1)$ $w_i = w_i + v_i$

AdaDelta: The same as RMSProp except it eliminates the need of hyperparameter a by making it a part of learning process. Here we will use δ_i instead of a and use the following update on it: $\delta_i = \rho\delta_i + (1-\rho)(\Delta w_i)^2$ where Δw_i is the increment in the value of w_i. $w_i = w_i - \sqrt{\frac{\delta_i}{A_i}}(\frac{dL}{dw_i})$

Adam: Keeps track of both expoentially smoothed value of the gradient and its square.

$A_i = \rho A_i + (1-\rho)(\frac{dL}{dw_i})^2, \rho \in (0,1)$ $F_i = \rho_f F_i + (1+\rho_f)(\frac{dL}{dw_i})$ $w_i = w_i - \frac{a_t}{\sqrt{A_i}}F_i$, where $a_t = a(\frac{\sqrt{1-\rho^T}}{1-\rho_f^T})$

Gradient clipping: If the partialt derivatives along different directions have drastically different magnitudes than one of this can be used: Value-based clipping: All partial derivatives that are less than the minimum are set to minimum threshold and the ones that are bigger than maximum threshold are set to maximum threshold. Norm-based clipping: Entire gradient vector is normalized by L_2 norm. However, it is particularly effective in avoiding the exploding gradient problem in recurrent neural networks.

References