Deep Learning: Assignment One

Aditi Nair (asn264) and Akash Shah (ass502)

March 7, 2017

1 Backprop

1. **Nonlinear Activation Functions**

SIGMOID:

$$\frac{\delta E}{\delta x_{in}} = \frac{\delta E}{\delta x_{out}} \cdot \frac{exp(x_{in})}{(1 + exp(x_{in}))^2}$$

TANH:

$$\frac{\delta E}{\delta x_{in}} = \frac{\delta E}{\delta x_{out}} \cdot \frac{4exp(4x_{in})}{(exp(2x_{in}) + 1)^2}$$

RELU:

We write:

$$\frac{\delta E}{\delta x_{in}} = \frac{\delta E}{\delta x_{out}} \cdot \frac{\delta x_{out}}{\delta x_{in}}$$

For the RELU unit, we notice that:

$$\frac{\delta x_{out}}{\delta x_{in}} = \begin{cases} 1 & \text{if } x_{in} > 0 \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

When we write $x_{in} = x_{in}^+ - x_{in}^-$, we can rewrite this as:

$$\frac{\delta x_{out}}{\delta x_{in}} = sign(x_{in}^+)$$

Therefore:

$$\frac{\delta E}{\delta x_{in}} = \frac{\delta E}{\delta x_{out}} \cdot sign(x_{in}^+)$$

2. **Softmax**

If $i \neq j$, then:

$$\frac{\delta (x_{out})_i}{\delta (x_{in})_j} = \frac{\beta e^{\beta((x_{in})_i + (x_{in})_j)}}{\left(\sum_k e^{-\beta (x_{in})_k}\right)^2}$$

If $i = j$, then:

$$\frac{\delta (x_{out})_i}{\delta (x_{in})_j} = \frac{\delta (x_{out})_i}{\delta (x_{in})_i} = \frac{-\beta e^{-\beta (x_{in})_i}\left[\left(\sum_k e^{-\beta (x_{in})_k}\right) + e^{-\beta (x_{in})_i}\right]}{\left(\sum_k e^{-\beta (x_{in})_k}\right)^2}$$

2 Techniques

2.1 Optimization

Gradient Descent Step by Momentum Method:

$$v_{t+1} = \mu v_t - \epsilon \nabla f(\theta_t)$$

$$\theta_{t+1} = \theta_t + v_{t+1}$$

$f(\theta)$ is an object function we are trying to minimize. $\epsilon > 0$ is the learning rate, $\mu \in [0, 1]$ is the momentum coefficient and v_t is the velocity vector.

Gradient Descent Step by Nesterov's Accelerated Gradient (NAG) Method:

$$v_{t+1} = \mu v_t - \epsilon \nabla f(\theta_t + \mu v_t)$$

$$\theta_{t+1} = \theta_t + v_{t+1}$$

Comparison of Methods:

In both methods, the previous velocity is decayed by μ, and then we apply a "correction" to it based on the gradient of f. The momentum method computes the gradient update based on the value of ∇f at θ_t whereas the NAG method computes the correction based on the value at $\theta_t + \mu v_t$, which looks like θ_{t+1} without the gradient-based correction. The NAG method is helpful in situations where μv_t is a suboptimal update direction for θ_{t+1}, since in this case $\nabla f(\theta_t + \mu v_t)$ will point more strongly to θ_t. In comparison, the momentum method would have to take the step incorporating μv_t and then correct the step later. Compounded over time, this allows NAG to prevent divergence.

2.2 Reducing Overfitting

1. Ideally we could have an ensemble of independently-trained weak or shallow neural networks whose results we could combine to get a more robust model. Since this is infeasible due to the computational burden of training neural networks and the limited availability of labelled data, we approximate having a larger number of less complex neural nets by applying dropout, in which we choose to randomly and independently drop units from a network with some probability. When we encounter a new training point or batch, we essentially train it on a new "thinned" network corresponding to the currently non-dropped network units. Each "thinned" network gets trained relatively few times, effectively creating a weak learner. At test time, we do not apply dropout, so we essentially combine the results of each "thinned" network. This is like an ensemble method, in that we create many weak learners and then combine their output at test time.

2. Let x_i be a unit in the hidden layer of a neural network. By applying dropout with a "keep" probability of p, the expected value of x_i at train time can be expressed as:

   $$\mathbb{E}[x_i] = p \cdot x_i + (1 - p) \cdot 0 = p x_i$$

   However, at test time, if we do not apply dropout and do not scale the values propagated from the unit x_i, the expected value will be:

   $$\mathbb{E}[x_i] = 1 \cdot x_i = x_i$$

   Now, on average, the values propagated from this unit will be very different from those encountered during training and the final probability scores of our model at test time will be distorted by this change. This effect will be exacerbated by the fact that dropout is generally applied to all or many of the units in the hidden layer. Therefore, we need to scale the outgoing weights of x_i by p at test time so that the expected value at test time equals the expected value at train time:

   $$\mathbb{E}[x_i] = p \cdot x_i = p x_i$$

3. Rather than obtain more labeled training data, we can augment the current dataset by including transformed versions of the original dataset including:

- Rotationed, translated and zoomed-in or zoomed-out versions of the original images.
- Stretched or squeezed versions of the original images.
- Elastic distortions of the original image, which is a type of image distortion that mimics natural oscillations in handwriting.

2.3 Initialization

1. We need to be careful about how we initialize weights in a neural network. If the weights of a matrix/vector are too small, then the signal will "shrink" as it passes through these units. If they are too big, the signal may explode and learning will diverge. For this reason, we rely on initialization methods to start off the weights at a safe range.

   In Xavier initialization, we choose w_{ij} in a weight matrix W from uniform (or Gaussian) distribution with $\mu = 0$ and $\sigma^2 = \frac{1}{N}$ where N is the number of input neurons to the corresponding unit. Generally, this initialization ensures that the variance of the input gradient and output gradient are the same in networks using non-linearities like the sigmoid and hyperbolic tangent functions.

   However, since the authors of the Xavier method derived the parameters for the distribution of W_{ij} based on properties of the sigmoid and hyperbolic tangent function, this is not necessarily well-suited for networks using RELU units, whose gradients generally behave very differently. Based on properties of the RELU unit, He et al. recommend choosing w_{ij} from a Gaussian distribution with $\mu = 0$ and $\sigma^2 = \frac{2}{N}$.

2. The VGG team first trained a small, shallow network. Next, they trained a deeper network where the first four and last three layers of the network were initialized with the trained weights from the first shallow model. These weights were allowed to change during the training procedure, and other weights in the deeper model were randomly initialized.

   Coate et al. used the following procedure to extract features from unlabelled data: first, they applied some basic preprocessing (like normalization) to the unlabelled data. Next they extracted "patches" from the images and developed vector representations of these patches using unsupervised methods like sparse auto-encoders, RBMs, K-Means, etc. Finally, they extracted the same-sized patches from the labelled images and applied the same unsupervised methods to the patches to get vector representations of the labelled patches. Assuming the unsupervised methods return feature vectors in $\mathbb{R}^k$, the resulting feature vectors were treated as a new image representations with k channels. After performing pooling on the quadrants of each channel, the authors were able to develop a final feature vector of length $4k$.

————————-DISCUSS OUR METHOD HERE————————-