Neural Networks

Ankur Mishra

2/26/18

Contents

1 Optimization

1.1 Two strategies

1. Random Search
   Randomly look through weights and record the W that returns the lowest loss. In a nutshell this is guess and check, which pretty much sucks, but slightly better than baseline ($10\% < 15\%$).

2. Gradient Descent
   Computing the slope accross every single direction; derivative. In multiple dimensions, the gradient is a vector of partial derivatives. It can be done numerically and analytically. In general, always use analytic gradients, but check if it is right with numeric gradients; also known as a gradient check.

   (a) Numerically Gradient
   Computing it can be thought as taking really small steps and finding the slope (Difference In Losses/Distance Between Points). Evaluating numerically is approximate and very slow, so don't do it. They are just easy to write.

   (b) Analytic Gradient
   Taking Derivatives. These are exact and very fast, but tricky to implement.

1.2 Mini-Batch Gradient Descent

Using small sections of training set to compute gradient. This is faster and better for the overall network and also creates noise which is better for optimization. Full-Batch will just give a straight line.
Common Sizes: $32, 64, 128$. Usually start with high learning rate and decays over time/epochs.

2 Neural Networks

2.1 Backpropagation

Way of computing the influence of every value on a computational graph by recursively using multivariable chain rule through the graph.

Chain Rule: $dL/dx = dL/dz * dz/dx$

You can break your backprop into other functions and find their derivatives. An example of this is breaking $\frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$ into the sigmoid function: $\frac{1}{1+e^{-x}}$

- plus (+) gate distrubutes gradients equally
- max gate routes gradient to max
- multiply (*) switches inputs and multiplies each by global gradient, equal inputs are picked arbitrarily

If two gradients combine when backpropagating –> add their gradients Linear Score Function: $f = W * x$
Two Layer Neural Network: $f = W_2 * max(0, W_1 * x)$
or Three Layers N-Network $f = W_3 * max(W_2 * max(0, W_1 * x))$
Bigger Networks are more powerful.

2.2 Activation Functions

2.2.1 Sigmoid Function

Equation: $o(x) = \frac{1}{1+e^{-x}}$
Historically the most popular since the implementation has a saturating firing rate. It squashes a number between 0 to 1.
Issues:

1. Saturated neurons kill gradients
   Only flows in the active region. If values are relatively high or low, gradients will only come out to be 0. Also drastically different inputs may come out to same outputs.
2. Outputs aren't 0 centered
   It doesn't converge as nicely. Integral of region comes out to 0.
3. Compute Time is Longer for Exponential Functions

2.2.2 tanh(x) Function

Is zero centered, range is between [-1, 1], and is 0 centered. Still has issue of killing gradients like sigmoid.

2.3 Rectified Linear Unit (ReLU)

Equation: $f(x) = max(0, x)$
Current standard for activiation functions, as it remedies most of tanh(x) and sigmoids problems, as it does not saturate (in + region) and is much more efficient. Still has issues:

1. Not Zero Centered
2. Kills Gradients which x < 0
3. Some initialization results in dead ReLUs If a neuron is not in activation region, it will die and never update. To fix people initialize slightly positive biases like .01; though not always effective.

2.3.1 Leaky ReLU / Parametric ReLU

Leaky ReLU equation $f(x) = max(.01x, x)$. Parametric ReLU equation: $f(x) = max(\alpha x, x)$, where α is a parameter that can be learned according to the network. These two maintain all the perks of ReLU and do not die, but still aren't amazing. Also aren't zero centered.

2.3.2 Exponential Linear Units

Exponential ReLU has all the benefits of ReLU, don't die, and closer to zero mean outputs, but compution for exponential takes time.

$$\text{Equation: } f(x) = \begin{cases} x, & \text{if } x > 0 \\ \alpha(exp(x) - 1), & \text{if } x \leq 0 \end{cases} \tag{1}$$

2.4 Image Pre-processing

Commonly pre-processing of images is done by mean centering. This either means to to subtract the mean value of each pixel by a [32,32,3] array, or to find the per-channel mean, which is subtract the mean from each pixel's RGB channels.

2.5 Weight Initialization

Setting weights to 0 will return 0 throughout network. Even .01 returns near zero values over the last few layers of a network in both forward and backward pass, which is known as vanishing gradient. Setting weight to 1, will supersaturate network, as all neurons come out as -1 or 1. The solution is Xavier initialization.

2.5.1 Xavier Initialization

W = np.random.randn(fan_{in}, fan_{out}) / np.sqrt(fan_{in}) for tanh(x). This breaks when using ReLU, so use W = np.random.randn(fan_{in}, fan_{out}) / np.sqrt(fan_{in} / 2) for ReLU.

2.6 Batch Normalization

This is to normalize data where you apply this equation to each layer:

$$\hat{x}^{(k)} = \frac{x^{(k)} - E[x^{(k)}]}{\sqrt{(Var(x^{(k)}))}}$$

Which is a vanilla differentiable function. What it does is it computes the mean of every feature and then divides by it.

$$y^{(k)} = \gamma^{(k)}\hat{x}^{(k)} + \beta^{(k)}$$

After this the function scaled by γ and then is shifted by β, which changes the range if the network wants to. Through learning the network can either learn to take it out or take advantage of it.
The general process of this is first the mini-batch mean is computed, then its variance. Using these two things, the values are normalized and finally are scaled and shifted.

2.6.1 Perks of Using It:

(a) Improves Gradient Flow

(b) Allows for High Learning Rates

(c) Reduces dependence on strong weight initialization

(d) Acts like regularization and slightly reduces need for dropout

2.7 Debugging Training

(a) Check if loss is reasonable
First disable regularization, and check loss. Then increase regularation, and check loss. If you are doing it right, the loss will also go up.
Then check if you can overfit your data with a small portion of your data-set with no regularation, by getting a loss/cost goes to 0 and accuracy is 100%.

(b) Your Learning Rate Should be Between 1e-3 to 1e-5

2.8 Parameter Updates

2.8.1 Simple Gradient Descent

Get Batch, Calculate Loss with Forward Pass, then Calculate Gradient with Backward Pass, and then perform parameter update. This however is the slowest way of training.

2.8.2 Momentum

A way of fixing this is to use momentum that uses v, which equals $\mu * v -$ (learning rate)dx. A physical representation of this is to think of it like a ball traveling down a U-shaped ramp. The learning rate is the ball's acceleration, and $\mu v - text(learningrate)dx$ is the ball's total velocity, which slowing down as the loss should be improving over time.
In general this usually overshoots and then converges to the expected loss. The velocity starts at 0.

2.8.3 Nesterov Accelerated Gradient

2.8.4 AdaGrad

2.8.5 RMSProp

2.8.6 Adam

Good for default cases.

2.8.7 Hessian and L-BFGS

2.9 Regularization: Drop-out