UNIVERSITY OF CALIFORNIA, BERKELEY

Lecture 19 - CS:189

Oscar Ortega

July 16, 2021

1 ESL: 11.5S

1.1 11.5.1: STARTING VALUES

- Note that if the weights are near zero, then the operative part of the Sigmoid is roughly linear and the linear network collapses into an approximately linear model. So, we tend to start with weights near 0, where individual units localize to directions and introduced non-linearity's where needed.
- Note that if we had starting weights of exactly 0, this would lead to perfect symmetry and the algorithm never moving, while starting instead with large weights often leads to poor solutions.

1.2 OVERFITTING

- We use an early stopping rule to avoid over-fitting, which has the effect of regularizing a model.
- **Weight Decay** is when we add a penalty to the cost function $R(\theta)+\lambda J(\theta)$, it is analogous to performing ridge regression on a modified cost function.

1.3 SCALING OF THE INPUTS

- We want to standardize all inputs to have mean 0 and standard deviation one, to ensure that all inputs are treated equally in the regularization process.
- Allows one to choose a meaningful range for the random starting weights.

1.4 Number of Hidden Units and Layers

- Generally speaking it is better to have too many hidden units than too few as extra weights can always be shrunk toward zero if appropriate regularization is used.

1.5 Multiple Minima

- The error function $R(\theta)$ is generally non-convex, and possesses many local minima, therefore our final solution will often be quite dependent on the choice of starting weights. One can try to mitigate this discrepancy by trying a number of random starting configurations and choosing the one with lowest penalized error. One can also average the results of predictions whose training started at different points. The non-linearity of the model makes it a poor idea to average the weights. One can also implement bagging which averages the predictions of networks training from randomly perturbed versions of the training data.

2 Heurisics for Faster Training

- Stochastic gradient descent; faster than batch on large, redundant data set.
- When running batch versus stochastic gradient descent one can notice that the descent for batch gradient descent is smoother when comparing epoch by epoch, but an epoch for gradient descent is a lot quicker. The stochastic gradient descent algorithm also finds minima a lot quicker, when there is redundant data.
- **Normalizing the data**
  - **center** each feature so mean is zero
  - then scale each features so variance is approximately 1
  - we want to do this so gradient descent has the chance to converge faster and to avoid in our loss function where the gradient is close to zero.
  - in general, when we have a convex-function that we are trying to minimize and the isocountours are very ovally, gradient descent ping-pongs in a pretty inefficient way. We use momentum to try and fix this.
- Centering the hidden units helps too:
  - We can do this by replacing sigmoids with $\tanh(\gamma) = \frac{e^{\gamma}-e^{-\gamma}}{e^{\gamma}+e^{-\gamma}} = 2s(2\gamma)-1$. Note, that if we use tanh units, we will want to update our code for the back-propagation.
  - We can also have different learning rates for each layer of weights.
  - Earlier layers have smaller gradients, need larger learning rates.

epoch: one epoch presents every training example once, Training usually takes many epochs, but if sample is huge it can take less than one

2.1 Emphasizing Schemes

- Present examples from rare classes more often, or with a bigger learning rate
- same for classified label

2.2 Second-order optimization

Often inexpensive to compute so not used in practice:

- Nonlinear conjugate gradient method: works well for small nets and the dataset is also small and we are performing regression instead of classification, this is also just a batch method and cannot be applied to stochastic gradient descent
- this is too slow with redundant data.
- Stochastic Levenberg Marquarat: approximating th diagonal of the hessian
- Acceleration schemes: RMSprop, Adam, AMS grad. Often adjust the learning rate in sophisticated ways.

3 Heuristics for Avoiding Bad Local Minima

- Stochastic Gradient Descent: "Brownian" motion motion gets you out of shallow local minima.
- Momentum: Gradient Descent changes "velocity" slowly. Carries us through shallow local minima to deeper ones.
- when we first get started: we first compute the gradient times the learning rate and modify the value of the weight of w to $w + \nabla w$, and them modify ∇w to $-\epsilon \nabla J(w) + \beta \nabla w$ in each iteration of the algorithm.
- This is good for both batch and stochastic. Choose hyperparameter $\beta < 1$.
- Tradeoff between β and ϵ: The larger β, the smaller ϵ needs to be. Should sum to about one. Some people suggest starting the β parameter to about 0.5 in initialization and then increasing the parameter to about 0.9 as one progresses through the epochs.

4 Heurisitcs to Avoid Over-fitting

- Ensembles of neural nets. Random initial weights + bagging
- l_2 regularization, aka **weight-decay**
- this corresponds to adding $\lambda \| w' \|_2^2$ to the cost/loss function, where w' is the vector of all weights except the bias terms.

- Effect: computing the partial $\epsilon \frac{\partial J}{\partial w_i}$ has extra term $-2\epsilon\lambda w_i$
- Weight decays by factor of $1 - 2\epsilon\lambda w_i$ if not reinforced training. Will pull things towards zero if not reinforced.
- Weight decay avoids over-fitting
- **dropout** emulates an ensemble in use network. We are going to just randomly disable subsets of the units, perform some training and repeat this process. Typically will disable between 10 percent of the input units, and 50 percent of the training units during each epoch of braining. Forces a very robust representation of the learning is pretty effective. Don't dropout the outputs.
- Fewer Hidden Units will help to reduce over-fitting as well.

5 Convulutional Neural Nets: (ConvNets; CNNs)

Caused the third big wave of machine learning!

Vision: inputs are large images 200 * 200 image = 40,000 pixels. If we connect them all to 40,000 hidden units which mean we have 1.6 billion connections. Too many edges and way too many parameters. Too many weights and too little data.
Convnet Ideas:

- Local Connectivity:
  - A Hidden Unit (in early layer) connects only to small patch of units in previous layer.
  - speeds up forward and backprop
- Shared Weights:
  - Groups of hidden units share same set of inputs.
  - those shared weights are called **masks**, also known as filters or kernels.
  - We learn several masks.
  - Masks * patches = hidden units in first hidden layer.
  - because of the shared weights, if one patch learns to detect edges, **every** patch has an edge detector
  - CNNs exploit repeated structure in , and audio.
  - **convolution**: the same linear transformation applied to different parts of image by shifting.
  - note how we now have fewer weights, we can also think of this as a sort of regularization.