Regularization

Sudeshna Sarkar

10 Feb 2017

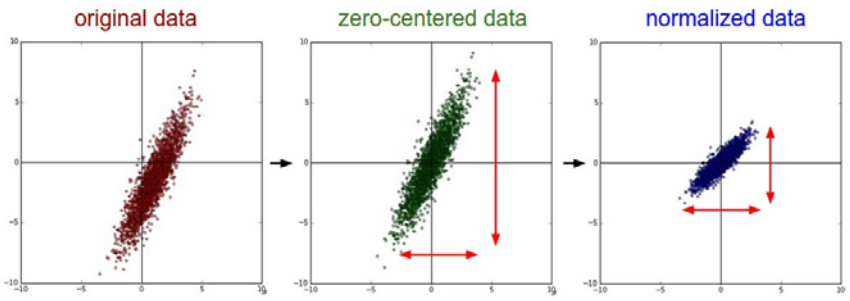
Sources

- Chapter 7 of Deep Learning
  www.deeplearningbook.org
- cs231n
  - http://cs231n.github.io/
- Neural Networks Part 2: Setting up the Data and the Losspreprocessing, weight initialization, batch normalization, regularization (L2/dropout), loss functions
- Neural Networks Part 3: Learning and Evaluationgradient checks, sanity checks, babysitting the learning process, momentum (+nesterov), second-order methods, Adagrad/RMSprop, hyperparameter optimization, model ensembles
- *Sebastian Ruder (2016). An overview of gradient descent optimisation algorithms. arXiv preprint arXiv:1609.04747.*

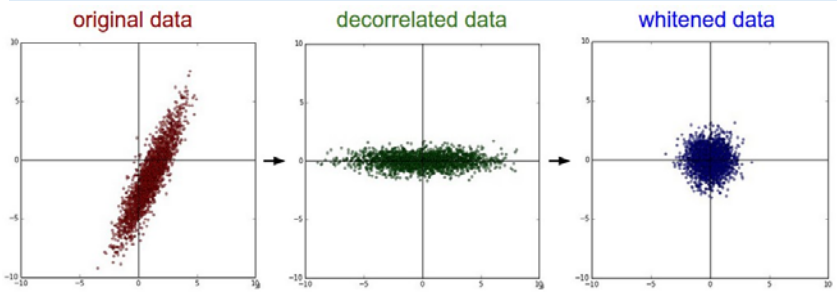
http://sebastianruder.com/optimizing-gradient-descent/

Data Preprocessing

- **Mean subtraction** is the most common form of preprocessing. It involves subtracting the mean across every individual *feature* in the data, and has the geometric interpretation of centering the cloud of data around the origin along every dimension
- **Normalization** refers to normalizing the data dimensions so that they are of approximately the same scale.
  - One is to divide each dimension by its standard deviation, once it has been zero-centered
  - Another form of this preprocessing normalizes each dimension so that the min and max along the dimension is -1 and 1 respectively.



- **PCA and Whitening** is another form of preprocessing. In this process, the data is first centered as described above. Then, we can compute the covariance matrix that tells us about the correlation structure in the data:
- We can compute the SVD factorization of the data covariance matrix:
- We can use this to reduce the dimensionality of the data by only using the top few eigenvectors, and discarding the dimensions along which the data has no variance. This is also sometimes refereed to as Principal Component Analysis (PCA) dimensionality reduction:



Weight Initialization

- Small random numbers.
- Calibrating the variances with 1/sqrt(n)
  - normalize the variance of each neuron's output to 1 by scaling its weight vector by the square root of its fan-in
- Sparse initialization:
  - set all weight matrices to zero
  - every neuron is randomly connected (with weights sampled from a small gaussian as above) to a fixed number of neurons below it.
- In practice, the current recommendation is to use ReLU units and use the w = np.random.randn(n) * sqrt(2.0/n),

- *Sebastian Ruder (2016). An overview of gradient descent optimisation algorithms. arXiv preprint arXiv:1609.04747.*
- *http://sebastianruder.com/optimizing-gradient-descent/*

definition

- "Regularization is any modification we make to a learning algorithm that is intended to reduce its generalization error but not its training error."
- There are many regularization strategies.
  - Put extra constraints on a machine learning model, such as adding restrictions on the parameter values.
  - Add extra terms in the objective function that can bethought of as corresponding to a soft constraint on the parameter values.
  - Sometimes these constraints and penalties are designed to encode specific kinds of prior knowledge.
  - Other times, these constraints and penalties are designed to express a generic preference for a simpler model class in order to promote generalization.
  - Sometimes penalties and constraints are necessary to make an underdetermined problem determined

Regularization in Deep Learning

- Mostly based on regularizing estimators.
- Regularization of an estimator works by trading increased bias for reduced variance

Parameter Norm Penalties

- limiting the capacity of models by adding a parameter norm penalty Ω(θ) to the objective function J.

$J(\theta; X, y) = J(\theta; X, y) + \alpha\Omega(\theta)$

where $\alpha \in [0, \infty)$ is a hyperparameter that weights the relative contribution of the norm penalty term, Ω, relative to the standard objective function J

L^2 Parameter Regularization

- commonly known as weight decay
- $\Omega(\theta) = \frac{1}{2}\|w\|_2^2$
- also known as ridge regression or Tikhonov regularization
- $J(w; X, y) = \frac{\alpha}{2} w^T w + J(w; X, y)$
- $\nabla_w \tilde{J}(w; X, y) = \alpha w + \nabla_w J(w; X, y)$
- $w \leftarrow w - \varepsilon(\alpha w + \nabla_w J(w; X, y))$ OR
- $w \leftarrow (1 - \varepsilon\alpha)w - \varepsilon \nabla_w J(w; X, y)$

L^2 Parameter Regularization

- $w \leftarrow (1 - \varepsilon \alpha)w - \varepsilon \nabla_w J(w; X, y)$
- the addition of the weight decay term has modified the learning rule to multiplicatively shrink the weight vector by a constant factor on each step, just before performing the usual gradient update

L^1 Regularization

$\Omega(\theta) = \|w\|_1 = \sum_i |w_i|$

$J(w; X, y) = \alpha\|w\|_1 + J(w; X, y)$

the corresponding gradient :

$\nabla_w \tilde{J}(w; X, y) = \alpha \, \text{sign}(w) + \nabla_w J(w; X, y)$

In comparison to L2 regularization, L1 regularization results in a solution that is more sparse. Sparsity in this context refers to the fact that some parameters have an optimal value of zero

Norm Penalties

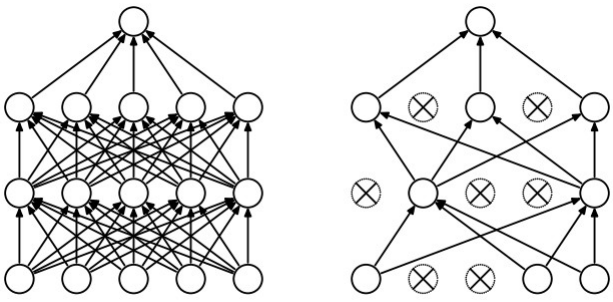
- L1: Encourages sparsity, equivalent to MAP Bayesian estimation with Laplace prior
- Squared L2: Encourages small weights, equivalent to MAP Bayesian estimation with Gaussian prior

Max norm constraints

- enforce an absolute upper bound on the magnitude of the weight vector
- and use projected gradient descent to enforce the constraint.
- corresponds to performing the parameter update as normal, and then enforcing the constraint by clamping the weight vector $\|\vec{w}\|_2 < c$

Dropout

- an extremely effective, simple and recently introduced regularization technique by Srivastava et al.
  - Dropout: A Simple Way to Prevent Neural Networks from Overfitting (pdf)
- that complements the other methods (L1, L2, maxnorm).
- While training, dropout is implemented by only keeping a neuron active with some probability p (a hyperparameter), or setting it to zero otherwise.



(a) Standard Neural Net (b) After applying dropout.

- During training, Dropout can be interpreted as sampling a Neural Network within the full Neural Network, and only updating the parameters of the sampled network based on the input data.
- During testing there is no dropout applied, with the interpretation of evaluating an averaged prediction across the exponentially-sized ensemble of all sub-networks
- http://cs231n.github.io/neural-networks-2/#datapre

Theme of noise in forward pass.

- Dropout falls into a more general category of methods that introduce stochastic behavior in the forward pass of the network.
- During testing, the noise is marginalized over analytically (multiplying by p), or numerically (e.g. via sampling, by performing several forward passes with different random decisions and then averaging over them).

In practice

- It is most common to use a single, global L2 regularization strength that is cross-validated.
- It is also common to combine this with dropout applied after all layers. The value of p=0.5 is a reasonable default, but this can be tuned on validation data.

Loss functions: Classification

- we assume a dataset of examples and a single correct label (out of a fixed set) for each example.
- SVM Loss / Hinge Loss

$$L_i = \sum_{j \neq y_i} \max\left(0, f_j - f_{y_i} + 1\right)$$

- Softmax classifier uses the cross-entropy loss

$$L_i = -log\left(\frac{e^{f_{y_i}}}{\sum_j e^{f_j}}\right)$$

Loss functions: Classification

Softmax classifier uses the cross-entropy loss

$$L_i = -log\left(\frac{e^{f_{y_i}}}{\sum_j e^{f_j}}\right)$$

- Problem: Large number of classes
  When the set of labels is very large (e.g. words in English dictionary), it may be helpful to use Hierarchical Softmax
- The hierarchical softmax decomposes labels into a tree. Each label is then represented as a path along the tree, and a Softmax classifier is trained at every node of the tree to disambiguate between the left and right branch.

Attribute classification

- y_i is a binary vector
- For example, images on Instagram can be thought of as labeled with a certain subset of hashtags from a large set of all hashtags, and an image may contain multiple.
- build a binary classifier for every single attribute independently. May take the form:

$$L_i = \sum_j \max\left(0, 1 - y_{ij} f_j\right)$$

- the sum is over all categories j, and yij is either +1 or -1 depending on whether the i-th example is labeled with the j-th attribute, and the score vector fj will be positive when the class is predicted to be present and negative otherwiseloss is accumulated if a positive example has score less than +1, or when a negative example has score greater than -1.

- An alternative: train a logistic regression classifier for every attribute independently. A binary logistic regression classifier has only two classes (0,1), and calculates the probability of class 1 as:

$$P(y = 1|x; w, b) = \frac{1}{1 + e^{-(w^T x + b)}} = \sigma(w^T x + b)$$
$$P(y = 0|x; w, b) = 1 - P(y = 1|x; w, b)$$

An example is thus classified as positive (y=1) if $\sigma(w^T x + b) > 0.5$

or equivalently if $w^T x + b > 0$

- The loss function then maximizes the log likelihood of this probability.
- This simplifies to:

$$L_i = \sum_j y_{ij} \log\left(\sigma(f_j)\right) + (1 - y_{ij}) \log\left(1 - \sigma(f_j)\right)$$

The labels y_{ij} are assumed to be either 1 (positive) or 0 (negative)

Gradient on f:

$$\frac{\partial L_i}{\partial f_j} = y_{ij} - \sigma(f_j)$$

Regression

- it is common to compute the loss between the predicted quantity and the true answer and then measure the L2 squared norm, or L1 norm of the difference. The L2 norm squared would compute the loss for a single example of the form:

Structured prediction

- The structured loss refers to a case where the labels can be arbitrary structures such as graphs, trees, or other complex objects.
- The basic idea behind the structured SVM loss is to demand a margin between the correct structure y_i and the highest-scoring incorrect structure.

Gradient Descent optimization algorithms

- Gradient descent is the most common way to optimize neural networks.
- Minimize objective function $J(\theta)$
  - by updating the parameters in the opposite direction of the gradient of the objective function $\nabla_\theta J(\theta)$ w.r.t. to the parameters.
- The learning rate η determines the size of the steps we take to reach a (local) minimum.
- we follow the direction of the slope of the surface created by the objective function downhill until we reach a valley.

1. Batch gradient descent

- computes the gradient of the cost function w.r.t. to the parameters θ for the entire training dataset:

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta)$$

- As we need to calculate the gradients for the whole dataset to perform just one update, batch gradient descent can be very slow.
- guaranteed to converge to the global minimum for convex error surfaces and to a local minimum for non-convex surfaces.

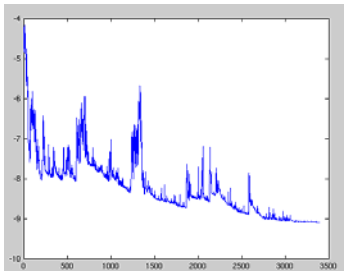
Stochastic gradient descent

- performs a parameter update for each training example x(i) and label y(i):

$\Theta = \theta - \eta \cdot \nabla\theta J(\theta; x(i); y(i))$

- usually much faster and can also be used to learn online.
- SGD performs frequent updates with a high variance that cause the objective function to fluctuate heavily

- when we slowly decrease the learning rate, SGD shows the same convergence behaviour as batch gradient descent.



Mini-batch gradient descent

- performs an update for every mini-batch of nn training examples:
- $\theta = \theta - \eta \cdot \nabla\theta J(\theta; x(i:i+n); y(i:i+n))$
- Common mini-batch sizes range between 50 and 256

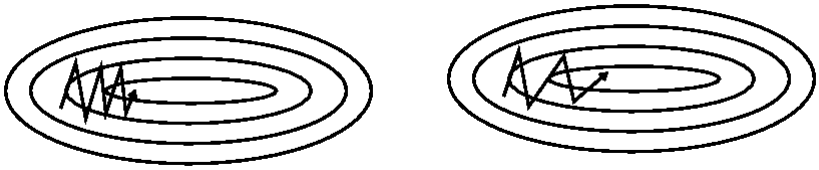
Challenges

- Choosing a learning rate
- Learning rate schedules [11] try to adjust the learning rate during training by e.g. annealing, i.e. reducing the learning rate according to a pre-defined schedule or when the change in objective between epochs falls below a threshold.
- Additionally, the same learning rate applies to all parameter updates. If our data is sparse and our features have very different frequencies, we might not want to update all of them to the same extent, but perform a larger update for rarely occurring features.

Gradient descent optimization algorithms

Momentum

- SGD has trouble navigating ravines, i.e. areas where the surface curves much more steeply in one dimension than in another
- SGD oscillates across the slopes of the ravine while only making hesitant progress along the bottom towards the local optimum



- Momentum is a method that helps accelerate SGD in the relevant direction and dampens oscillations
- by adding a fraction γ of the update vector of the past time step to the current update vector

- $v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta)$
- $\theta = \theta - vt$
- The momentum term γ is usually set to 0.9 or a similar value.
- when using momentum, we push a ball down a hill. The ball accumulates momentum as it rolls downhill, becoming faster and faster on the way (until it reaches its terminal velocity if there is air resistance, i.e. $\gamma<1$).

Nesterov accelerated gradient

- We'd like to have a smarter ball, a ball that has a notion of where it is going so that it knows to slow down before the hill slopes up again.
- Computing $\theta - \gamma v_{t-1}$ thus gives us an approximation of the next position of the parameters
- We can now effectively look ahead by calculating the gradient not w.r.t. to our current parameters θ but w.r.t. the approximate future position of our parameters:
- $v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta - \gamma v_{t-1})$
- $\theta = \theta - v_t$

- γ set to around 0.9.
- Momentum first computes the current gradient (small blue vector) and then takes a big jump in the direction of the updated accumulated gradient (big blue vector)
- NAG first makes a big jump in the direction of the previous accumulated gradient (brown vector), measures the gradient and then makes a correction (red vector), which results in the complete NAG update (green vector).
- This anticipatory update prevents us from going too fast and results in increased responsiveness.

This much TILL MIDTERM

Adagrad

- we would also like to adapt our updates to each individual parameter to perform larger or smaller updates depending on their importance.
- Adagrad adapts the learning rate to the parameters,
  - performing larger updates for infrequent and
  - smaller updates for frequent parameters.
  - For this reason, it is well-suited for dealing with sparse data
- Dean et al. found that Adagrad greatly improved the robustness of SGD and used it for training large-scale neural nets at Google, -- learned to recognize cats in Youtube videos.
- Pennington et al. used Adagrad to train GloVe word embeddings, as infrequent words require much larger updates than frequent ones.

Adagrad

- Adagrad uses a different learning rate for every parameter θi at every time step t,
- $g_{t,i}$: the gradient of the objective function w.r.t. to the parameter θi at time step t:
- $g_{t,i}=\nabla_{\theta}J(\theta_i)$
- SGD update: $\theta_{t+1,i} = \theta_{t,i} - \eta g_{t.i}$
- Adagrad:

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} g_{t.i}$$

$G_t \in \mathbb{R}^{d \times d}$ is a diagonal matrix where each diagonal element i,i is the sum of the squares of the gradients w.r.t. θi up to time step t

Adagrad's main weakness: accumulation of the squared gradients in the denominator: causes the learning rate to shrink and eventually become infinitesimally small

Adadelta

- Adadelta restricts the window of accumulated past gradients to some fixed size w.
- the sum of gradients is recursively defined as a decaying average of all past squared gradients.
- $E[g^2]t=\gamma E[g^2]_{t-1}+(1-\gamma)g2t$

Adam

- Adaptive Moment Estimation (Adam) computes adaptive learning rates for each parameter.
- In addition to storing an exponentially decaying average of past squared gradients vt like Adadelta, Adam also keeps an exponentially decaying average of past gradients mt, similar to momentum:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$
$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) {g_t}^2$$

m_t and v_t are estimates of the first moment (the mean) and the second moment (the uncentered variance) of the gradients respectively.

- mt and vt are initialized as vectors of 0's, they are biased towards zero, especially during the initial time steps, and especially when the decay rates are small (i.e. β1 and β2 are close to 1).
- They counteract these biases by computing bias-corrected first and second moment estimates:

$$\widehat{m_t} = \frac{m_t}{1 - {\beta_1}^t}$$

$$\widehat{v_t} = \frac{v_t}{1 - {\beta_2}^t}$$

They then use these to update the parameters -- the Adam update rule

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\widehat{v_t}} + \epsilon} \widehat{m_t}$$

propose default values of 0.9 for β1, 0.999 for β2, and 10−8 for ϵ.