Assignment 1: 1-layer Mini Batch Gradient Descent

—

Project Report

—

DD2424

Fabian Assarsson

2018-04-05

Executive Summary

We implement and evaluate a 1-layered neural network with a softmax activation function and a cross-entropy loss. We employ the mini-batch gradient-descent algorithm to efficiently train the network through a gradient-based learning. By training, validating, and testing on the CIFAR-10-dataset, we evaluate different parameter schemes in terms of learning-rate, number of epochs, size of dataset and batch-size. We also test whether xavier-initialization improves the stability of the learning scheme, and change the size of training data as well as the total number of training epochs.

Contents

1 Introduction

1.1 Notation

We employ much the same notation as is seen in Deep Learning[2].

Numbers and Arrays

a – denotes a scalar, integer or real
$\boldsymbol{a}$ – denotes a vector
$\boldsymbol{A}$ – denotes a matrix
$\mathbf{A}$ – denotes a tensor
$\boldsymbol{I}$ – the identity matrix
$\text{diag}(\boldsymbol{a})$ – denotes a square, diagonal matrix with entries given by $\boldsymbol{a}$
a – a scalar random variable
$\mathbf{a}$ – a vector-valued random variable
$\mathbf{A}$ – a matrix-valued random variable

Indexing

a_i – Element i of vector $\boldsymbol{a}$, indexed from 1.
a_{-i} – All elements from $\boldsymbol{a}$, except i
$A_{i.j}$ – Element i,j of matrix $\boldsymbol{A}$
$\boldsymbol{A_{i,:}}$ – Row of matrix $\boldsymbol{A}$
$\boldsymbol{A_{:,i}}$ – Column of matrix $\boldsymbol{A}$
$A_{i,j,k}$ – Element i,j,k of tensor $\mathbf{A}$
$\mathbf{A}_{:,:,\mathbf{i}}$ – 2D slice of a 3D tensor
a_i – Element i of random vector $\mathbf{a}$

Linear Algebra Operations

$\boldsymbol{A}^\top$ – Transpose of matrix $\boldsymbol{A}$.
$\boldsymbol{A}^+$ – Moore-Penrose Pseudoinverse of $\boldsymbol{A}$
$\boldsymbol{A} \odot \boldsymbol{B}$ – Hadamard product between $\boldsymbol{A}$ and $\boldsymbol{B}$
$\det(\boldsymbol{A})$ – The determinant of $\boldsymbol{A}$

Calculus

$\frac{dy}{dx}$ – derivative of y w.r.t x
$\frac{\delta y}{\delta x}$ – partial derivative of y w.r.t x.
$\nabla_{\boldsymbol{x}} y$ – gradient of y w.r.t $\boldsymbol{x}$
$\nabla_{\boldsymbol{X}} y$ – matrix derivates of y w.r.t $\boldsymbol{X}$
$\nabla_{\mathbf{X}} y$ – tensor containing derivates of y w.r.t $\mathbf{X}$

Probability and Information Theory

a⊥b – The random variables a and b are independent
a⊥b|c – They are conditionally independent given c
P(a) – A probability distribution over a discrete variable a
p(a) – A probability distribution over a continuous variable a
a $\sim P$ – random variable a has distribution P
$\mathbb{E}_{x\sim P}[f(x)]$ or $\mathbb{E}f(x)$ – Expectation of $f(x)$ with respect to $P(x)$
Var($f(x)$) – Variance of $f(x)$ under $P(x)$
Cov($f(x)$,$g(x)$) – Covariance of $f(x)$ and $g(x)$ under $P(x)$
H(x) – Shannon Entropy of the random variable x
$D_{\mathrm{KL}}(P||Q)$ – Kullback-Leibler divergence of P and Q
$\mathcal{N}(\mathbf{x};\mu,\mathbf{\Sigma})$ – Gaussian distribution over $\mathbf{x}$ with mean μ and covariance $\mathbf{\Sigma}$

1.2 The one-layer neural network

In our first assignment, we have implemented a fairly straight forward and simple one-layer neural network. Given a set of training data $\boldsymbol{X} = \{\boldsymbol{x}_1, ..., \boldsymbol{x}_n \,|\, \boldsymbol{x}_i \in \mathbb{R}^d\}$ and a corresponding set of training labels $\boldsymbol{Y} = \{y_1, ..., y_n \,|\, y_i \in \{1, ..., K\}\}$, we can construct a dataset $\mathcal{D} = \{(\boldsymbol{x}_i, y_i)\}_{i=1}^n$. The goal of our assignment is to map the corresponding input-data to the correct output-label with as good accuracy[1] as possible. With our network, we feed our training data through an affine transformation, governed by parameters $\boldsymbol{W}, \boldsymbol{b}$, and then perform a non-linear transformation of the result. Ideally, as we have K classes to possibly map each input-example against, we would like that our non-linear transformation produce some kind of probability distribution over the classes, with some class having the distinctively highest probability. We then select the class with highest such "probability score" and output that as the network's guess. To achieve this, we need to choose a suitable non-linearity as well as a function that determines the "wrongness" of our suggestions, i.e. a cost function. With our chosen functions we will employ mini-batch gradient descent to update our parameters in the negative directions of the cost w.r.t to our parameters to make it as small as possible. It is standard in neural network literature to have a cost function that consists of a loss function and a regularization term, so we will have a short discussion on both.

1.2.1 The Softmax activation

Given that our dataset is "fixed" and that we do not preprocess it in any way, and given that our parameters are initialized as

$$\begin{aligned} \boldsymbol{W} &\sim \mathcal{N}(\mathbf{x};\mu,\mathbf{\Sigma}), \mu = \mathbf{0}, \mathbf{\Sigma} = \sigma^2\boldsymbol{I}, \sigma = 0.1 \\ \boldsymbol{b} &\sim \mathcal{N}(\mathbf{x};\mu,\mathbf{\Sigma}), \mu = \mathbf{0}, \mathbf{\Sigma} = \sigma^2\boldsymbol{I}, \sigma = 0.1 \end{aligned} \tag{1}$$

we can't be sure of the norm of the output from our affine transformation. In order to be a proper probability distribution, the values need to be normalized

[1] subject to the constraint of avoiding overfitting

and as some values can be negative it makes intuitive sense to tream them as they where log probabilities. To recover the actual probabilities, we therefore need to take the exponent of all outputed values and then normalize. It turns out that this reasoning leads us to the very definition of the softmax non-linearity function:

$$\text{softmax}(\boldsymbol{s}) = \frac{\exp(\boldsymbol{s})}{\mathbf{1}^\top \exp(\boldsymbol{s})} \tag{2}$$

1.2.2 Cross-Entropy Loss

As we have decided to interpret the output of our affine transformation as unnormalized log probabilities of class membership and then used the softmax activiation to normalize and recover, we now have a "proper" probability distribution $\hat{P}$ of class membership outputted by our network. If we use a one-hot encoding for each $y_i \in \{1, ..., K\} \rightarrow \boldsymbol{y}_i = \boldsymbol{e}_j$ when y_i takes on value j, we can view our truth-labels as belonging to a different distribution P. It makes sense to try to make these distributions as close to each other as possible, but as regular euclidean distances doesn't apply in probabilistic settings we need to introduce the *cross-entropy loss*. As cross-entropy between $\hat{P}$ and P can be expressed as:

$$H(P, \hat{P}) = H(P) + D_{\text{KL}}(P||\hat{P}) = \mathbb{E}_P[-log\hat{P}] \tag{3}$$

We that, not only do we do an analogous "distance minimization" (by minimizing the Kullback-Leibler divergence), but we also have a log-term that gets canceled out by our softmax for much easier and nice-looking gradients! The cross-entropy loss therefore makes sense in two ways; it is easy to interpret and calculate and it's intuitive to want to minimize the encoding cost between our "true distribution" given by our labels, and our guessed distribution given by the network. Minimizing the cross-entropy loss is also equivalent to performing a maximum likelihood estimation of our data. A reasonable assumption about our data is that it is independent and identically distributed, giving rise to a joint likelihood function $\mathcal{L}$ of the form:

$$\mathcal{L} = \prod_n \mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}}) \tag{4}$$

The likelihood is the probability which our model gives to our correct class. With one-hot encoding this reduces to the dot product between our ground truth one-hot label and our guessed distribution. If we want to maximize this function, it is the same as minimizing it's negative log (since the logarithm is monotonic):

$$-\log\mathcal{L} = -\sum_n \log\mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}}) \tag{5}$$

or, with one-hot encoding after some manipulation is equal to our cross-entropy loss:

$$-\log\mathcal{L} = -\sum_n \log(\boldsymbol{y}^\top \boldsymbol{p}) \tag{6}$$

1.2.3 Regularization

Regularization of this network, with an L_2-penalty on the weight parameters, will punish models that have large weight parameters and reward those with small, diffuse ones.

We have already stated that our cross entropy function is performing a maximum likelihood estimation ('MLE'). But, if we want to go "fully Bayesian", we would like to perform a maximum a posteriori estimation of our parameters given our fixed data. This results in us needing to choose a suitable prior distribution over our parameters, namely the conjugate prior to our likelihood. As the conjugate prior to a gaussian is another gaussian, we can change the likelihood maximization objective to a maximum a posteriori instead:

$$\text{MAP} = \prod_n \mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}})\, \mathcal{N}(\boldsymbol{W}; \boldsymbol{0}, \boldsymbol{\lambda}^{-1}) \tag{7}$$

Which, under the same argument as above, reduces to:

$$-\log \text{MAP} \propto -\sum_n \log(\boldsymbol{y}^\top \boldsymbol{p}) - \boldsymbol{\lambda}\boldsymbol{W}^2 \tag{8}$$

Consequently, introducing a L_2-regularization on the weight matrix $\boldsymbol{W}$ is the same as performing a MAP-estimate of our parameters when we assume a Gaussian prior on the distribution of our weights. It should be noted here that it doesn't imply that we 'create' our weight-matrix as a gaussian (which we do), but rather that our "guess" on the final form of the learned matrix, before any evidence, is that it is Gaussian.

1.3 Gradient calculations and verifications

The gradient calculations was performed correctly. To ensure this, they where checked against ComputeGradsNum as provided with the relative error method from the assignment. I picked the largest differing value and ensured that it was below 10^{-6}, an arbitrary threshold value. The actual result, tediously calculated on the full 10 000 example and 3072-dimensional data set was:

$$\begin{aligned} \epsilon_W &= 1.4593 * 10^{-8} \\ \epsilon_b &= 2.0347 * 10^{-8} \end{aligned} \tag{9}$$

1.4 Improvement schemes

1.4.1 More data

If possible, the addition of more data is a great way to increase generalization capabilities of the model and to reduce overfitting when training much.

1.4.2 Xavier Initialization

We employ the xavier-initialization as introduced by Glorot and Bengio[1]. It exploits the fact that our affine transformation scales the variance of our data with $n\text{Var}(\boldsymbol{W})$. We want this scaling to be 1, so as to preserve the variance across the layer and ensure that our forward activation don't become too small or large. It is trivial, under these assumptions (that are somewhat weak being based on i.d.d and linearity), to see that $n\text{Var}(\boldsymbol{W}) = \frac{1}{n}$. If we do the same analysis for the back-prop step, we end up with random Gaussian initialization with the following variance:

$$\text{Var}(\boldsymbol{W}) = \frac{2}{n_{in} + n_{out}} \tag{10}$$

1.4.3 Longer training time

The introduction of longer training time let's us converge for a longer time. To avoid overfitting we will employ *early stopping* when our validation error decreases slowly enough. As our training error - under the correct parameter regime - will continuously decrease as the model fits to the noise from observed examples, we need to "force quit" the learning earlier. As the model doesn't see the validation data at any point in time, it makes sense to stop when that error doesn't improve. My threshold was set at 10^{-4}.

2 Results

2.1 Without improvements

2.1.1 Parameters: $\lambda = 0$ epochs $= 40$ batches $= 100$ $\eta = 0.01$

Accuracy: 36.16 %

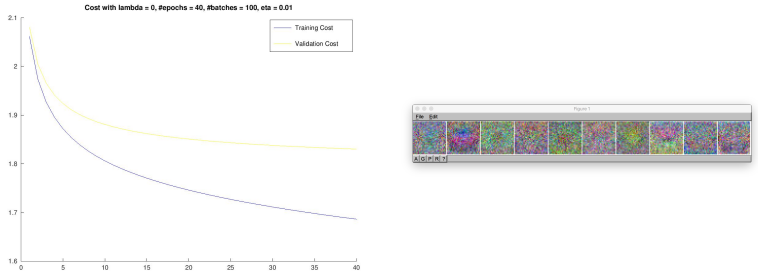

Figure 1: The training and validation loss at different epochs with final class template representation

2.1.2 Parameters: $\lambda = 0$ epochs $= 40$ batches $= 100$ $\eta = 0.1$

Accuracy: 18.22 %

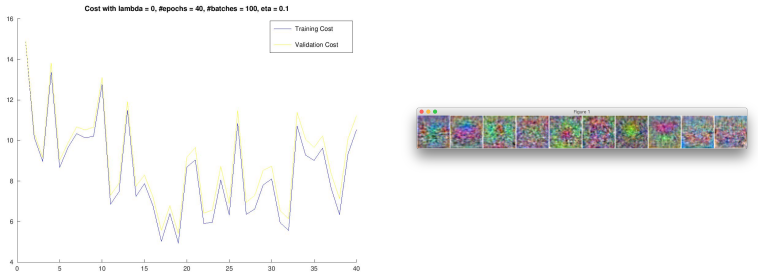

Figure 2: The training and validation loss at different epochs with final class template representation

2.1.3 Parameters: $\lambda = 0.1$ epochs $= 40$ batches $= 100$ $\eta = 0.01$

Accuracy: 33.33 %

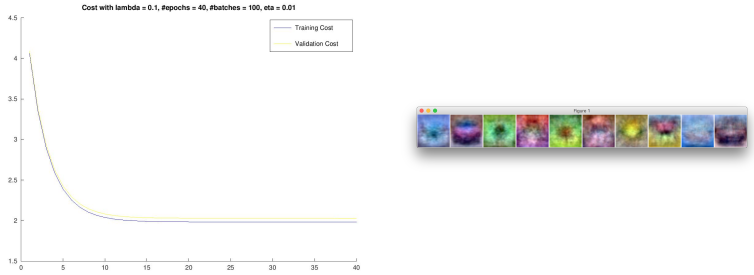

Figure 3: The training and validation loss at different epochs with final class template representation

2.1.4 Parameters: $\lambda = 1$ epochs $= 40$ batches $= 100$ $\eta = 0.01$

Accuracy: 21.90 %

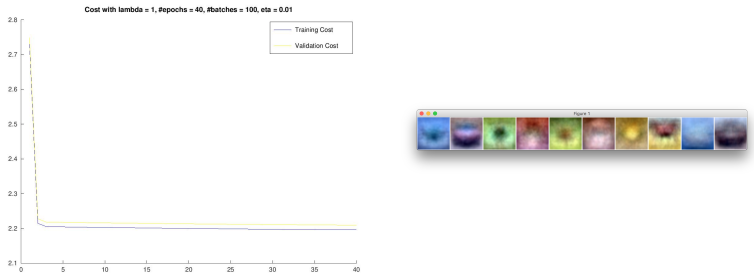

Figure 4: The training and validation loss at different epochs with final class template representation

2.2 With improvements

Accuracy: 40.65 %

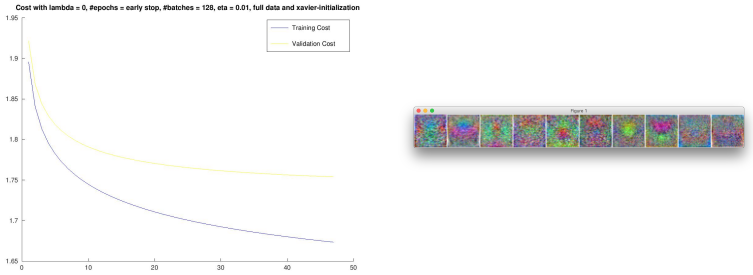

Figure 5: The training and validation loss at different epochs with final class template representation

3 Discussion

3.0.1 Effects of Regularization

The regularization diffuses the outputted probability vector. A high regularization will push towards a uniform distribution whereas no regularization/L_1-regularization will push for more "spikey distributions". See the introduction on regularization for more information.

3.0.2 The importance of the learning rate

As we only train with the use of a subgradient, the learning rate sets the "jump length" of our updates on the error surface. A large such rate will force us to make "long jumps" and perhaps overshooting optimal values and oscillating between suboptimal points. A low learning-rate, on the other hand, will be more likely to put is in a local optimum, and in a much longer time since more steps are needed to traverse the same distance.

References

[1] Xavier Glorot and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks". In: *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. 2010, pp. 249–256.

[2] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. `http://www.deeplearningbook.org`. MIT Press, 2016.