Assignment 3: k-layer Mini Batch GD with Batch Normalization

–

Project Report

–

DD2424

Fabian Assarsson

2018-06-06

Executive Summary

We implement and evaluate a k-layered neural network with a hidden layer consisting of ReLu-activated nodes, a softmax output 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. We employ batch normalization to stabilize and speed up training. By training, validating, and testing on the CIFAR-10-dataset, we evaluate different parameter schemes in terms of learning-rate and regularization term. We also implement momentum to speed up training, as well as learning rate decay.

Contents

1 Introduction

1.1 Notation

We employ much the same notation as is seen in Deep Learning[3].

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 k-layer neural network

In our third assignment, we have implemented a k-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. We iterate this process k times. Firstly, we would like to have hidden non-linearities that generates - according to some criteria - "good results". This transformation may allow us disentangle the data in the distorted space, as well as incorporate other benefits before we try to find class representations. And secondly, as we have K classes to possibly map each input-example against, we would like that our output 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 Hidden layer with m_l nodes

One obvious question one might ask is why we want to introduce several layers, or depth. A one-layer neural network has two general assumed priors inherent to the model itself. First, it operates under a "smoothness" prior, roughly

[1]subject to the constraint of avoiding overfitting

implying that small shifts in input space result in small shifts in the output space. The second is that of distributed representation. Each node, width wise, can be assumed to be responsible for some large factor of variation in the data. The canonical example is that how a colour node, a model-type node and manufacturer node can represent a lot of types of vehicles with just three nodes. Increasing depth strenghtens a third prior; the prior of compositional hierarchy. Images, for example, can be thought of as edges that form motifs which assemble in to parts and forms,[5]. The accumulation of composite non-linear functions capture this assumption about the data in a powerful way. In the k-layer example, it is also trivial to see why: Widening the softmax layer will destroy the intuition behind producing a probability distribution corresponding to our K class labels. Introducing intermittent layer with variable width increases the number of trainable parameters and therefore model expressability. Additionally, as showcased in[6], depth is exponentially more efficient than width in terms of representational capacity.

1.2.2 The ReLu activation

When agreeing on the fact that deep network have several benefits over shallow and wide counterparts (i.e. shallow networks that have the same number of computed parameters), we need to choose a non-linearity in this transformation as well. While the softmax makes sense for the output layer in terms of the probabilistic and information theoretic interpretations discussed below, the intermediate non-linearity must be chosen according to some other criterias. It makes sense to assume that we would like learning to be fast and the model robust towards small perturbations in the input data. Historically tanh and sigmoid functions have been used to achieve this purpose, but as depth and model sizes have increased (and our demand on performance), the problem with saturating gradients have become evident. As activations grow very large via large weights, the gradient tends towards zero on both function tails which slows down the updates and makes the gradients vanish. As the Relu function is either zero or strictly linear, this doesn't happen[4]. Another benefit is that of the sparse representation a ReLu-layer entails with random initialization[2]. The sparsity is argued to increase information disentanglement (one change in an input dimension effects less of the parameter values by zeroed connections), variable sized representation and linear separability (sparse representation is trivially seen to be more easy to linearely separate).

1.2.3 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

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

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 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:

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

1.2.4 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 activation 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 $y_i \in \{1, ..., K\} \to \boldsymbol{y}_i = \boldsymbol{e}_j$ 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:

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

1.2.5 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 ComputeGradsNumSlow as provided with the relative error method from the assignment. I picked the largest differing value and ensured that it was around 10^{-3}, an arbitrary threshold value. The actual result, calculated on 5 examples and 10-dimensional data set was:

$$\begin{aligned}
\epsilon_{W_1} &= 3.2168 * 10^{-3} \\
\epsilon_{b_1} &= 3.8478 * 10^{-11} \\
\epsilon_{W_2} &= 1.3427 * 10^{-7} \\
\epsilon_{b_2} &= 3.5090 * 10^{-11} \\
\epsilon_{W_3} &= 3.1874 * 10^{-10} \\
\epsilon_{b_3} &= 1.5456 * 10^{-10}
\end{aligned} \tag{9}$$

1.4 Momentum

Introducing momentum puts a premium on gradient update components that are similar over time, thus reducing the oscillating tendencies that are commonplace in subgradient optimization techniques. In practice, this should improve learning speed as we are moving less (in an euclidean sense) in contradictive direction and more in common ones. Our results enforce this reasoning, as showcased below:

1.5 Parameter search

Two levels of parameter search was conducted. A rough search over the learning rate was followed by a conjoined "course to fine" search over combinations of learning rates and regularization terms, in the same manner as the earlier assignment.

1.5.1 learning rate

A wide search was performed with the suggested small regularization term.

- 23.89 % accuracy for λ 1e-06 and η 0.001
- 33.22 % accuracy for λ 1e-06 and η 0.01
- 35.56 % accuracy for λ 1e-06 and η 0.05
- 37.89 % accuracy for λ 1e-06 and η 0.1
- 34.44 % accuracy for λ 1e-06 and η 0.3
- 33.24 % accuracy for λ 1e-06 and η 0.5

We see clear trends that too small values give a slow learning rate and poor performance, but that we are allowed somewhat higher learning-rates due to our batch-normalization. 0.02-0.2 seems optimal.

1.5.2 learning rate and regularization term

For this test, an exhaustive and iterative search was performed with a fixed learning rate in our optimal range, paired with 20 regularization terms from a much larger span of $(10^{-7}, 10^{-1})$. The accuracy is strictly decreasing as a function of increasing learning rate, so a low rate of 10^{-7} was chosen as well.

2 Results

2.1 3 layers with and without Batch Normalization

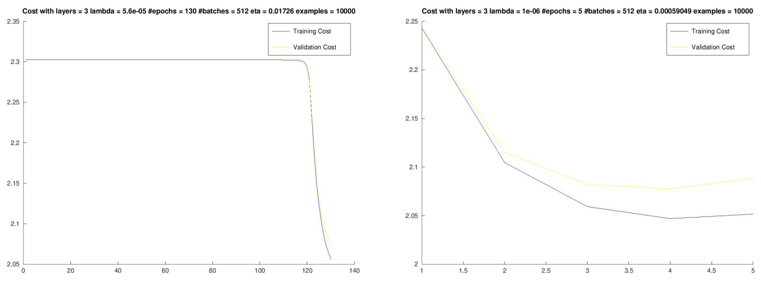

Figure 1: Left: Without Batch Normalization. Right: With Batch Normalization

The batch normalization makes an incredible difference in how long time it takes for the network to "latch on" to the learning. Earlier motivations related to the reduction of internal covariate shift might not be the answer to this observed improvement. In fact, it can be shown that the Lipschitz continuity of both the optimization function and it's gradients improves with batch normalization. (That is, the Lipschitz constant is smaller after applying batch normalization. This reduces the irregularities of both the optimization space and the gradient-space, which smoothens learning and makes it faster).

2.2 2 layers with and without batch normalization

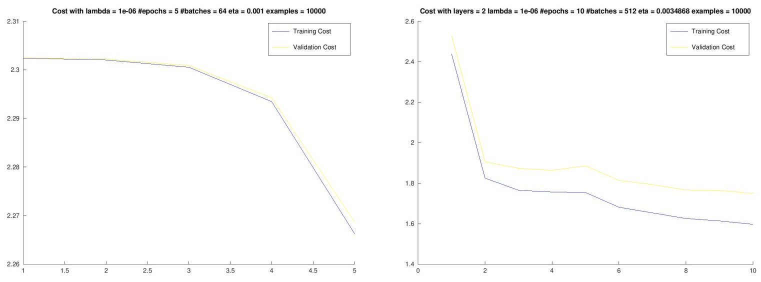

Figure 2: Left: Without Batch Normalization. Right: With Batch Normalization. LR: 0.001

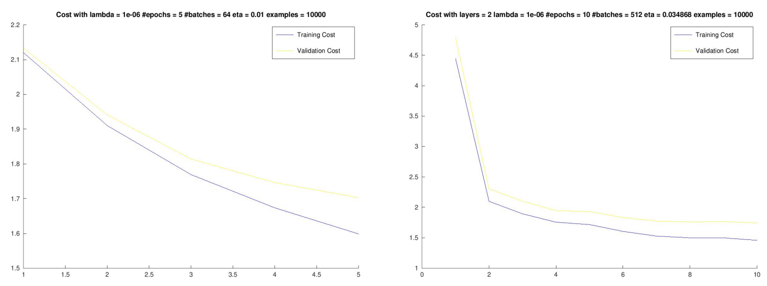

Figure 3: Left: Without Batch Normalization. Right: With Batch Normalization. LR: 0.01

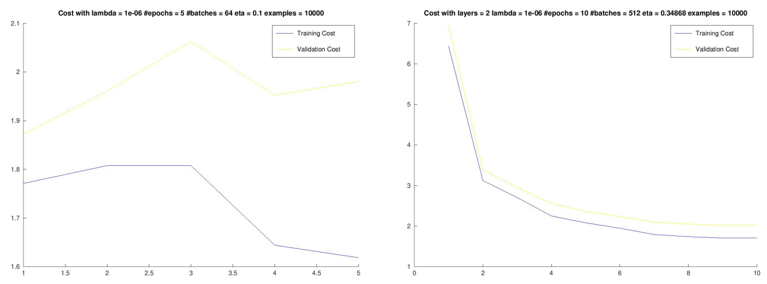

Figure 4: Left: Without Batch Normalization. Right: With Batch Normalization. LR: 0.1 and 1 for BN

2.3 Best results

The best results obtained was 41.74 %

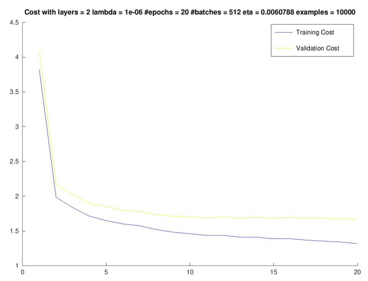

Figure 5: Training and validation error

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] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. “Deep sparse rectifier neural networks”. In: *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*. 2011, pp. 315–323.

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

[4] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. “Imagenet classification with deep convolutional neural networks”. In: *Advances in neural information processing systems*. 2012, pp. 1097–1105.

[5] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. “Deep learning”. In: *nature* 521.7553 (2015), p. 436.

[6] Matus Telgarsky. “Benefits of depth in neural networks”. In: *arXiv preprint arXiv:1602.04485* (2016).