Auto-encoding Variational Bayes with Extensions

Shraman Ray Chaudhuri
Department of EECS, MIT
Cambridge, MA 02139
`shraman@mit.edu`

Abstract

We investigate in full detail the Auto-encoding Variational Bayes paradigm originally proposed by Kingma et. al. and scrutinize both its limitations and effectiveness. We also propose and implement vectors for improvement, detailing both theoretical advantages and empirical results, including Amari's natural gradient extension to stochastic optimization and variational Gaussian dropout. Finally, we provide a ground-up implementation of an efficient and modifiable Variational Auto-Encoder that summarizes our findings.

1 Introduction

Recent trends in high-performance neural computation frameworks have shown that a Bayesian interpretation of neural net regression [14] can open doors to a wide variety of applications [8]. One interesting use of neural computation is performing variational inference for intractable posteriors while simultaneously performing maximum a posteriori (MAP) estimates over the parameters of the true posterior. The realization of these algorithms often includes an analogy to *encoders* (for the variational approximation) and *decoders* (for the MAP estimate).

Variational inference has proven to be highly scalable [5], adaptable [9], and intuitive [2] as an optimization problem. Therefore, it isn't too surprising that a coveted goal is to eliminate as many restrictions on the approximating model (e.g. conjugacy) while still being able to perform efficient, accurate VI. The auto-encoding variational Bayes (AEVB) framework partially accomplishes these goals.

The rest of this paper is organized as follows: Section 2 roughly outlines the AEVB framework, Sections 3 & 4 discuss key optimizations explored, Section 5 details the ground-up implementation of the project, Section 6 describes a few replicable experiments to assess the performance of the AEVB model, and the Appendices include all original figures and derivations (to economize on space).

2 Variational Autoencoding Framework [1]

The notation used throughout this paper is as follows: x represents observed data, z are latent variables, θ are hyperparameters of true distributions p, and ϕ are variational parameters of the approximating distribution q. To briefly review the problem of variational inference, we aim to maximize the lower bound on the log marginal likelihood $\log p_\theta(z)$:

$$\begin{aligned}\log p_\theta(x) &\geq \mathcal{L}(\theta, \phi; x) \\ &= E_{q\phi}[p_\theta(x, z)] + E_{q\phi}[q_\phi(z)]\end{aligned}$$

To maximize the right-hand side, we need to compute the gradient $\nabla_\phi E_{q\phi}[p_\theta(x, z)] + \nabla_\phi E_{q\phi}[q_\phi(z)]$, for which it is often difficult to derive a closed-form. We could indeed limit our approximating

6.882, Bayesian Inference and Modelling (Prof. Tamara Broderick), MIT.

distribution q to certain conjugate families, but AEVB attempts to eliminate that restriction to allow for a richer set of approximate models.

The other option for computing gradients on these expectations is *Monte-Carlo Estimation* (MCE). The straightforward Monte-Carlo estimator for gradients by applying the chain-rule is $\nabla_\phi E_{q\phi}[f(z)] \approx \frac{1}{L}\sum_{i=1}^{L} f(z)\nabla_q \log_{q\phi}(z^{(i)})$, which exhibits high variance. Although this doesn't lead to any theoretical disadvantages, the advent of large datasets require gradient-descent mechanisms to work with minibatches (i.e. stochastic gradient descent) which perform very poorly when the gradient estimator exhibits high variance [1]

A straightforward solution is to reparameterize z so that our Monte-Carlo sampler does not need to succumb to high variance. For example, if our approximating distribution is Gaussian with diagonal covariance, i.e. $q_\phi(z) = \mathcal{N}(\mu, \sigma^2)$, we can reparameterize z by introducing an auxiliary variable ϵ such that $z = \mu + \sigma \cdot \epsilon$, and $\epsilon \sim \mathcal{N}(0, I)$. The advantage brought by this simple trick is two-fold: we are outsourcing the uncertainty on z to an auxiliary variable (not dependent on ϕ) so we can compute the gradient of an expectation rather than an rely on MCE of individual gradients, and more importantly, we've greatly reduced the variance of our MCE by sampling ϵ instead of z.

The final estimator derived in [1] (through straightforward algebraic manipulation) is: $-KL(q_\phi(z)||p_\theta(z)) + \frac{1}{L}\sum_{i=1}^{L}\log p_\theta(x|z^{(i)})$ where z is parameterized as above. [2] This is the objective function we feed into our encoder-decoder pair of networks to maximize via stochastic gradient descent (SGD). The authors of [1] highlight a key intution for this form of the ELBO in particular—the first term prevents diverging too far from the prior on z (preventing overfitting as a *regularizer*) and the second term is a *reconstruction error* that drives both variational inference in the encoder and maximum likelihood in the decoder.

The physical representation of the AEVB framework and intuitions thereof are in Appendix C.

3 Optimization I: Natural Gradient

A key lemma in regular convex optimization is that the gradient points in the direction of steepest ascent of the objective function, with respect to the parameters we optimize, and almost all flavors of SGD make this assumption. Although generally valid for most regression models, this assumption breaks down once we consider relative entropy as a metric for local "distance." Amari showed a few applications where exploiting the contour of the relative entropy function space could lead to faster learning, since we can derive gradients that *actually* point in the direction of steepest ascent. This led to a secondary goal of this project: to incorporate natural gradients in a novel setting—the AEVB framework—which is unprecedented for this special class of algorithms (as far as I know). This section adapts the insights from [6] and [7] particularly for AEVB.

3.1 The Natural Gradient for AEVB

Recall that Lagrange Multipliers can be used to optimize a function over contours in the input space produced by simple equality constraints (a constructive review of Lagrange Multipliers is provided in Appendix A). Let's adapt this technique to the problem of finding an optimal direction $\Delta\theta$ to descend in our relative entropy manifold. [3] Formally, we are trying to determine $\arg\min_{\Delta\theta} f(\theta + \Delta\theta)$ s.t. $KL(p_\theta||p_{\theta+\Delta\theta}) = c$ where f is any arbitrary loss function w.r.t. θ (which parameterizes a probability distribution). The constraint can be intepreted as making progress at a constant rate c in the KL-divergence manifold while moving through the parameter space.

We make a few Taylor approximations of our two functions $f(\theta + \Delta\theta)$ and $KL(p_\theta||p_{\theta+\Delta\theta})$. For the former, we use a first-order approximation $f(\theta) + \nabla f(\theta)\Delta\theta$. For the latter, we use a second-order approximation $KL(p_\theta||p_{\theta+\theta}) + \nabla KL(p_\theta||p_\theta)\Delta\theta + \frac{1}{2}\Delta\theta^T\nabla^2 KL(p_\theta||p_\theta)\Delta\theta$, where it is straightforward to check that the first two terms are (individually) zero [7]. Our approximation of the KL-divergence is then $-\frac{1}{2}\Delta\theta^T E[\nabla^2 \log p_\Theta(z)]\Delta\theta \overset{\text{def}}{=} \frac{1}{2}\Delta\theta^T\mathbf{F}\Delta\theta$, where $\mathbf{F}$ is also known as the Fisher matrix (i.e. the Hessian of the KL divergence).

[1]We now have *two* sources of variance—one from randomly sampling the minibatches and another for the Monte-Carlo estimate of the gradients.

[2]This new KL-divergence term is with respect to the *prior* $p_\theta(z)$, not the posterior

[3]In particular, a Riemannian manifold that behaves locally Euclidean

The form for our optimization problem directly calls for Lagrange multipliers. The Lagrangian of our problem is $\mathcal{L}(\Delta\theta, \lambda) = f(\theta) + \nabla f(\theta + \Delta\theta) + \lambda\frac{1}{2}\Delta\theta^T\mathbf{F}\Delta\theta$, and solving with respect to $\Delta\theta$ (see Appendix B for derivations), we have $\Delta\theta^* = -\frac{1}{\lambda^*}\mathbf{F^{-1}}\nabla f(\theta)$. [4] Therefore, the optimal direction (natural gradient) is simply $\nabla_N f(\theta) = \mathbf{F^{-1}}\nabla f(\theta)$. By pre-multiplying our computed gradients in the AEVB framework by the inverse Fisher matrix, we can potentially converge to a better ELBO estimate. Computation of the Fisher matrix for various activations and neural network units are provided in [7], which were used religiously to implement a working prototype.

3.2 Implementation Challenges

A considerable caveat is that the above derivations based on Taylor approximations are accurate only as $\Delta\theta \to 0$, so one immediate challenge is ascertaining fast convergence while placing a limit on our step size $\Delta\theta$ (too high, and the natural gradient would diverge since our approximations fall apart).

A key challenge was simply computing the Fisher information matrix $\mathbf{F}$. In general, computing Hessians via naive pairwise operations lead to quadratic runtime $O(N^2)$ where N is the dimensionality of our parameter space. Contrast this with the relatively straightforward computation of the gradient in $O(N)$. It is difficult to interface such a computation with existing high-performance implementations of gradient descent (let alone parallelize it) and this comes with good reason since the Fisher information matrix is often used as a standalone device rather than an augmentation to stochastic gradient descent. Therefore, recursive methods such as those presented in [11] are too slow, and the implementation in Section 5 opts for a cruder first-order approximation.

The final challenge is maintaining this Fisher information matrix across training iterations so that the Hessian information doesn't need to be recomputed at every timestep. A lot of quasi-Newton gradient descent techniques and implementations thereof are able to update an approximate Hessian so that only a fraction of the cells need to be recomputed. This is, however, outside the scope of this project.

4 Optimization II: Dropout and Reparameterization

A followup paper [4] was written by the original authors of [1] introducing yet another reparameterization trick that decreases the minibatch variance with respect to the auxiliary random variable ϵ while keeping the space complexity as low as possible. The problem with the estimator in Section 2 is that its variance does not decrease inverse-linearly with the size of the minibatch M. To eliminate the covariance between estimators, we would require M different weight matrices to produce M different Monte-Carlo estimates of z, one for each datapoint. However, this would lead to an enormous blow-up in memory, even with efficient computation graphs (see Section 5).

Fortunately, [4] provides an ingenious solution where we can reparameterize so that we instead of sampling the weights, we add an auxiliary noise variable to each of the *outputs* of the encoder network. As in Section 2, the distribution from which this auxiliary noise variable is sampled from can simply be $\eta_{m,j} \sim \mathcal{N}(0, I)$ where m represents the index in the minibatch and j a specific coordinate of z.

On top of this, we can leverage the "dropout" technique introduced recently as free-lunch for regularization in neural networks (e.g. as in [3]). The concept of dropout is partially turning off the activations for some nodes by multiplying each weight by a coefficient sampled from $\mathcal{N}(0, I)$. It turns out that the above reparameterization trick is essentially performing this dropout technique under the hood—where the $\eta_{m,j}$ values at the output produce the same effect on $q_\phi(z)$ as if we were sampling from their marginal.

Ironically, implementing this dropout technique was extremely straightforward (adding a few nodes to the vanilla AEVB's computation graph) and led to better improvements than the natural gradient modification above (see Section 6/Appendix D).

5 Implementation

You can find the full GitHub repository at this URL:

[4] λ^* is the optimal λ that satisfies $\nabla_\lambda\mathcal{L}(\theta, \lambda) = 0$; however, recall that we were simply looking for the direction of the optimum and so the scalar $-\frac{1}{\lambda^*}$ is superfluous since we must tune the step-size separately to achieve *practical* convergence rates.

`https://github.com/shraman-rc/VINNy`

The repository has been refactored many times to be highly self-contained, easily modifiable, and equipped with a suite of qualitative and quantitative tests. An updated `README` is coming soon, but check out `vae.py` and `tests.py` for the vanilla Variational Autoencoder framework and test suites respectively. Basic installation of packages with `pip install` should suffice in running the code.

The entire repository is implemented using highly performant scientific Python, facilitated via libraries like TensorFlow [12] and NumPy. The main computation workhorse comes from the *computation graph* interface provided by TensorFlow to easily construct inter-variable relationships and objective functions (which can indeed incorporate randomness—e.g. for sampling the auxiliary random variable ϵ). Some TF-related optimizations incorporated include manually controlling gradient magnitudes (to prevent exploding near the first stages of training, while also counteracting early annealing of the weights), pruning expensive operations (such as `sqrt` ops and `log/exp`'s) and/or computing them ad-hoc, and using built-in differentiation operators to perform the Fisher matrix computation as explained in Section 3.

Although TensorFlow is quite powerful, there were several roadblocks and non-trivial debug situations that were difficult to overcome given that it is still a nascent library. The foremost problem was that the original paper [1] outlined a model for the VAE that was *highly* underspecified. The activation and optimization engines were surprisingly outdated and led to performance issues with the neural network architecture they proposed. Some examples of incompatibilities included the depth of the MLP (multi-layer perceptron), the acceptable learning rate given batch size and complexity of the computation graph, and exploding gradients using state-of-the-art optimizers (e.g ADAM). These were very implementation-specific and therefore hard to diagnose.

Other less obvious problems included the family of approximating distributions to use for the variational inference (for my particular case, I had to use a Gaussian variational distribution $q_\phi(z)$ but a *Bernoulli* likelihood distribution $p_\theta(x|z)$ to model *images*). Finally, overfitting was a large issue which was partially ameliorated by scaling up the coefficient for the KL-divergence term (with respect to the *prior*—not the posterior) so that the variational approximation wouldn't diverge too far from $p_\theta(z)$.

6 Experiments

The initial tests measured the reconstruction quality (while varying the dimensionality of the latent space $\mathbf{z}$) via predictive likelihood on a held-out test set of approximately 5000 images. It turns out that the predictive likelihood measure did not change significantly after a 10-dimensional latent space, which makes sense since there are only 10 digits. See Appendix D for visualization of a 2D latent space as well as visual assessment of reconstruction performance.

After the main optimizations outlined in Sections 3 and 4, there were slight improvements to the convergence time as well as the evidence lower bound calculation. The natural gradient method was not as effective as Variational Dropout and simple reparameterization techniques, and keeping track of the Fisher information matrix turned out to be very computationally disadvantageous (close to 70x decrease in runtime). As stated, informative plots can be found in Appendix D.

7 Conclusion

It was incredibly rewarding to build an end-to-end implementation of AEVB and progressively see the optimizations come to life. To summarize some key takeaways: (1) be wary of "black-box" methods especially those which incorporate samplers as high variance is often tricky to avoid, (2) statistical optimizations (e.g. the natural gradient) may be outperformed by numerical ones (e.g. ADAM) which may in-turn be outperformed by other statistical ones (e.g. Variational Dropout), and (3) hybrid perspectives, such as Bayesian approach to neural net regression, can often lead to some of the most simple, elegant solutions (e.g. reparameterization).

References

[1] Kingma, D.P. & Welling, M. (2014) Auto-encoding Variational Bayes. *Advances in Neural Information Processing Systems*.

[2] Blei, D. (2011) Variational Inference (Lecture Notes). *Princeton CoS597C*.

[3] Ioffe, S. & Szegedy C. (2015) Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. *Journal of Machine Learning Research*.

[4] Kingma, D.P. & Salimans, T. & Welling, M. (2015) Variational Dropout and the Local Reparameterization Trick. *Advances in Neural Information Processing Systems*.

[5] Broderick, T. & Boyd, N. & Wibisono, A. & Wilson, A. C. & Jordan, M. (2013) Streaming Variational Bayes. *Advances in Neural Information Processing Systems*.

[6] Hoffman, M. & Blei, D. & Wang, C. & Paisley, J. (2013) Stochastic Variational Inference. *Journal of Machine Learning Research*.

[7] Pascanu, R. & Bengio, Y. (2014) Revisiting the Natural Gradient for Deep Networks *International Conference on Learning Representations*.

[8] Gregor, K. & Danihelka, I. & Graves, A. & Rezende, D. J. & Wierstra, D. (2015) DRAW: A Recurrent Neural Network for Image Generation *International Conference on Machine Learning*.

[9] Salimans, T. & Kingma, D.P. & Welling, M. (2015) MCMC and Variational Inference: Bridging the Gap *Journal of Machine Learning Research*.

[10] Blei, D. & Ng, A. & Jordan, M. (2002) Latent Dirichlet Allocation *Advances in Neural Information Processing Systems*.

[11] Cavanaugh, J. E. & Shumway, R. H. (2011) On Computing the Expected Fisher Information Matrix for State-Space Model Parameters *Elsevier Letters in Statistics and Probability*.

[12] Monga, R. & Vinyals, O. & Dean, J. & et. al. (2015) TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems *Google Research*.

[13] LeCun, Y. & Cortes, C. & Burges, C. J. C. (1998) The MNIST Database of Handwritten Digits. *NYU, Google Labs, Microsoft Research*

[14] Bishop, C. (2006) Pattern Recognition and Machine Learning, Chapter 3.3-3.5, 4.2-4.6 *Advanced Computing Machinery, Springer Information & Statistics Series*

Appendix A: Lagrange Multipliers

Say we want to minimize a function $f(x)$ with simple constraint $h(x) = c$ (which can be rewritten as $g(x) = h(x) - c = 0$). We fold the constraint into the optimization function itself by making an extra observation—if $f(x)$ is to be optimized over the contour $g(x) = 0$ then the optimum must lie at a point where the contours are parallel—and since the gradients are orthogononal to the contour lines, the gradients must also be parallel (or antiparallel). We then have that $\nabla_x f(x) = \lambda \nabla_x g(x)$ (where λ can indeed be negative). If we define a loss function $\mathcal{L}$ s.t. $\nabla_x \mathcal{L}(x) = \nabla_x f(x) + \lambda \nabla_x g(x)$, then by optimizing $\mathcal{L}$, we achieve our parallel constraint $\nabla_x f(x) = -\lambda \nabla_x g(x)$ as desired.

To incorporate the final constraint $g(x) = 0$, we can optimize the same function $\mathcal{L}(\lambda) = f(x) + \lambda g(x)$ with respect to λ so that $\nabla_\lambda \mathcal{L}(\lambda) = 0$ gives us our desired $g(x) = 0$ condition. Then, by finding the optimum of $\mathcal{L}(x, \lambda) = f(x) + \lambda g(x)$, we have a necessary (but not sufficient) constraint to solving our original constrained optimization problem.

Appendix B: Solving Lagrangian in Section 3

$$\begin{aligned}
\nabla_{\Delta\theta} \mathcal{L}(\Delta\theta, \lambda) &= \nabla_{\Delta\theta} f(\theta) + \nabla_{\Delta\theta}(\nabla f(\theta)\Delta\theta) + \nabla_{\Delta\theta}(\lambda \frac{1}{2} \Delta\theta^T \mathbf{F} \Delta\theta) \\
&= \nabla \mathcal{L}(\theta) + \lambda \mathbf{F} \Delta\theta \\
&= 0 \\
&\implies \Delta\theta^* = -\frac{1}{\lambda^*} \mathbf{F^{-1}} \nabla f(\theta)
\end{aligned}$$

Appendix C: Figures for Section 2

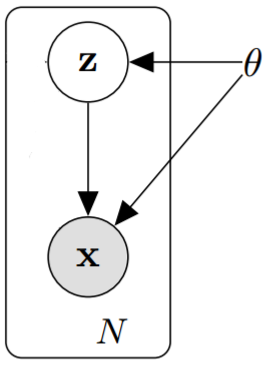

Figure 1: The "coding" model under consideration. We have 1 latent per data point, where the latent distribution is parameterized by hyperparameters θ. This model, however simplistic, definitely appear frequently in the real world (e.g. performing MAP estimates on latent variable z—which can be interpreted as a "code"—is equivalent to find a probabilistic compression/decompression scheme for data x)

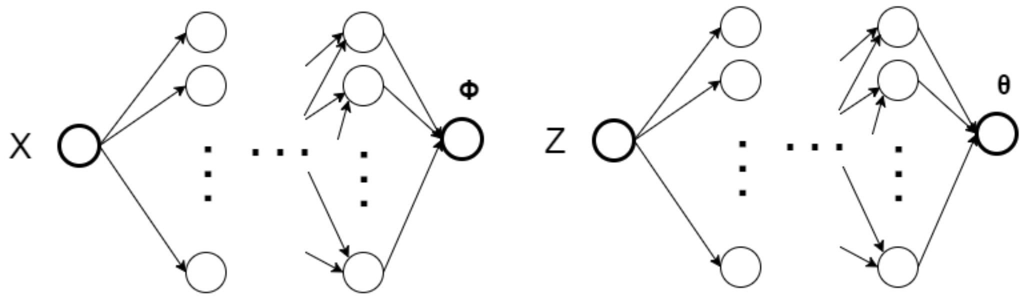

Figure 2: The architectures for the encoder and decoder. The hidden layers were composed of multi-layer perceptrons (implemented with full matrix multiplication with weight matrices and ReLU activation). In the case of a Gaussian variational approximation, ϕ would be μ, σ. Note that ϕ implicitly contains the parameters for the encoder (weights, biases) and θ for the decoder. Finally, the z that is pushed through the decoder is computed via sampling $\epsilon \sim p(\epsilon)$ and plugging into $\mu + \sigma \cdot \epsilon$

Appendix D: Figures for Section 6

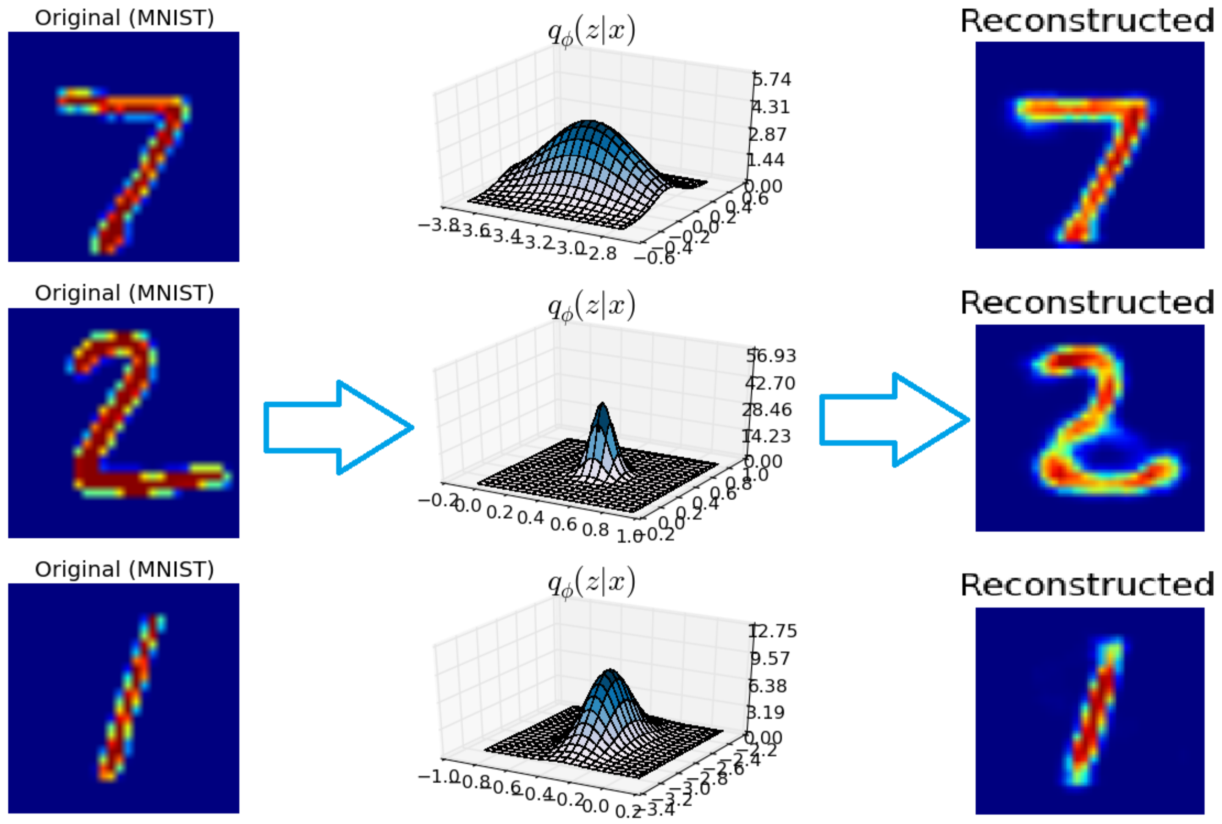

Figure 3: Assessment of Variational Auto-encoder Performance. The left column shows the original MNIST images, the middle column shows the pdf's of each (Gaussian) approximation $q_\phi(z|x^{(i)})$ parameterized by the $\phi = (\mu, \log \sigma^{\mathbf{2}})$ outputs of the encoder, and the right column shows a sample from the likelihood distribution $p_\theta(x|z)$ which is parameterized by the output of the decoder. I only chose a latent space of dimension 2 for easy visualization, but I will soon integrate PCA for higher dimensional latent spaces in the GitHub repo.

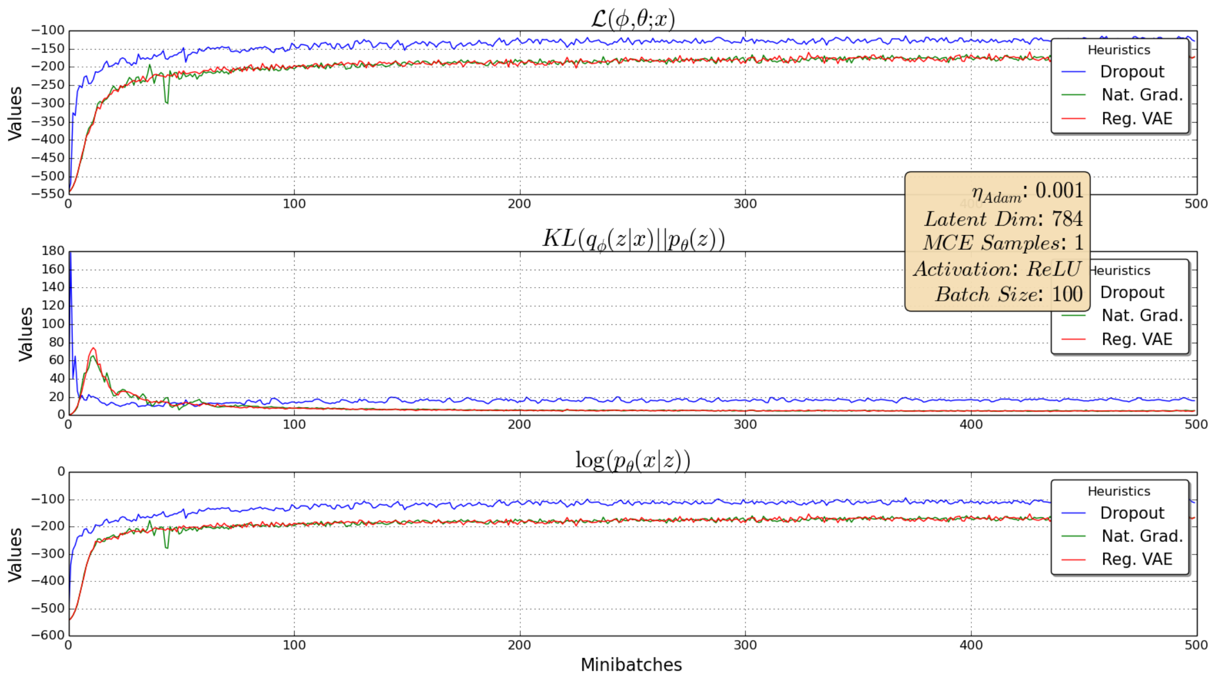

Figure 4: Assessment of Optimization Performance. The three different plots, top to bottom, represent the ELBO estimate (cost function), the KL-divergence with respect to the prior (regularization term), and the predictive likelihood (negative reconstruction error). All 3 runs were performed with the same set of optimal parameters shown in the beige box. The red line shows the performance of the first working prototype of AEVB. The green line shows performance after applying the inverse Fisher matrix to the gradients. The blue line represents Variational Dropout—i.e. applying Gaussian coefficients to encoder activations. We can clearly see a performance increase in both convergence rate and final ELBO estimate when applying the dropout technique, but there is not much difference when using the natural gradient.

Appendix E: Work in Progress

Although the original goals of this project were met earlier than expected (to correctly implement and debug a Variational Autoencoder framework, and assess its performance on a real-world dataset), there are several directions I believe will improve the architecture in some way, shape, or form. Some improvements and analyses thereof were discussed in sections 3 and 4, and others (works in progress) are discussed herewith.

7.1 Optimization III: Streaming VB

To expand the implementation to a streaming setting, I turned to [5] since the factorization and i.i.d. assumptions aligned well with the variational approximations of interest for VAEs. Furthermore, the second reparameterization trick can counteract minibatch variance so that doing posterior updates exactly as outlined in [5], except with distributed encoders, would efficient and accurate. I've implemented a meta-framework in Go (a distributed systems language) to test this optimization and the codebase has been noticeably refactored to support logical connections between Variational Autoencoders in the TensorFlow computation graph and the Go meta-framework. The Go branch has been omitted from the repo.