Batch Normalization and Dropout

Eric Auld

November 17, 2018

Abstract

The stochastic regularization techniques *dropout* and *batch normalization* are examined with an eye toward separating their optimization and regularization effects. Experiments are presented which suggest that they accomplish regularization in different ways. The methods are examined from the Bayesian perpective, via Gaussian process inference, and possibilities for future work are outlined based on the the results. Code for all experiments is available on `www.github.com/ericauld`.

1 Introduction

Dropout and batch normalization are two techniques for optimizing deep neural networks which have appeared in the last ten years. Dropout was explicitly introduced as a regularizer, that is, to stop the network from overfitting. Batch normalization, on the other hand, was introduced to aid in the training objective. However, regularizing effects of it have been discovered, which are not well-understood. This report is to review the methods, and begin an investigation into separating the regularizing from the optimizing effects of batch normalization, and, more generally, to gain insight into how batch normalization and dropout accomplish their regularization.

We begin with a review of the methods, and their variants, and how they relate to our questions.

2 Batch normalization

2.1 Stochastic gradient descent in the context of neural networks

In the common case where our objective function is an average of many terms, such as

$$\mathcal{L}(\theta) \overset{(\text{def})}{=} \frac{1}{n}\sum_{i=1}^{n} \ell(x_i, y_i, \theta)$$

the approach of stochastic gradient descent is to select one of these terms at a time and take the gradient step with respect to this term:

$$\theta_{\text{new}} = \theta_{\text{old}} - \eta g_i$$

where g_i is the gradient of the ith term, and $\eta > 0$ is a learning rate.

If we sample $g_i \in \{g_1, \ldots, g_n\}$ uniformly at random, we can treat g_i as a random variable with mean $\frac{1}{n}\sum g_i \overset{(\text{def})}{=} g$ and variance

$$\text{Var}(g_{\text{sgd}}) = \frac{1}{n}\left[\left(\frac{1}{n}\sum_{i=1}^{n} g_i g_i^T\right) - gg^T\right]$$

In the context of neural networks, with their highly nonlinear objective functions, and quickly evolving architectures, proofs of convergence of SGD have been scarce; however, there has been some progress. For instance, in a June 2018 preprint [24], Xiao Zhang et al. have produced a linear convergence rate of a single-layer ReLU network trained with SGD to a given maximum error with high probability and under hypotheses on the minimum size of the underlying training set.

2.2 Minibatch training

Stochastic gradient descent as such is uncommon in neural network training; minibatch training is much preferred. Minibatch training instead selects a subset $g_{i_1}, \ldots, g_{i_b}$ of the gradients, for $b < N$, and takes the average gradient step with respect to these.

$$\theta_{\text{new}} = \theta_{\text{old}} - \frac{\eta}{b}\sum_{j=1}^{b} g_{i_j}$$

Various sampling methods for the multiset $i_1, \ldots, i_b$ (for the indices are not necessarily distinct) are possible. If we sample the indices from a uniform distribution over $\{1, \ldots, n\}$, with replacement, then our minibatch gradient $\frac{1}{b}\sum_{j=1}^{b} g_{i_j}$ is a random variable with mean g, and variance [15]

$$\text{Var}(g_{\text{mb},1}) = \frac{1}{b}\left[\left(\frac{1}{N}\sum_{i=1}^{N} g_i g_i^T\right) - gg^T\right].$$

If we sample uniformly without replacement, the resulting random variable is still unbiased, and its variance differs by $O(1/N)$, which we demonstrate in appendix B.

$$\text{Var}(g_{\text{mb},2}) = \frac{1}{b}\left[\left(\frac{1}{N}\sum_{i=1}^{N} g_i g_i^T\right) - gg^T\right] + O(1/N). \tag{1}$$

Minibatch training is well-suited for parallel computing, because the propagation of the minibatch elements to the next BN layer can be done in parallel, and indeed, these days minibatch sizes of between 10 and $10,000$ elements are common because of their harmony with parallel computing architecture [9].

2.3 Intro to batch normalization

Batch normalization was introduced in 2015 by Ioffe and Szegedy [12]. The method adds another step to selected layers of the network, often to all of the hidden layers. Specifically, a typical layer of a feedforward neural network is an affine map $\mathbb{R}^m \to \mathbb{R}^n$, followed by a nonlinear function $\mathbb{R} \to \mathbb{R}$, applied in parallel to all the entries of the vector in $\mathbb{R}^n$. Typical nonlinearities are the ReLU $x \overset{\phi}{\mapsto} \max(0, x)$ or the sigmoid $x \overset{\sigma}{\mapsto} \frac{1}{1+e^{-x}}$. The additional step proposed by batch normalization can be placed either between the affine and activation steps, or after both of them (see figure 1).

The first step of BN is a normalization step, in which an affine transformation is applied to each unit u_k so that the mean and variance of the set $u_k(x_{i_1}), \ldots, u_k(x_{i_j}) \in \mathbb{R}$ are zero and one, respectively. Then, as if to undo the previous step, in the "scale/shift" step, an affine transformation $x \mapsto \gamma x + b$ is applied to each unit, setting the mean and variance to a new value. Crucially γ and β are trainable parameters, and this entire operation can be backpropagated through, which makes BN a practical optimization tool for neural networks.

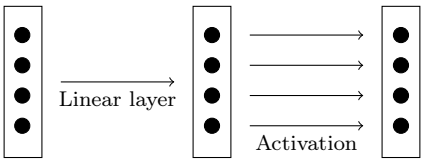
One of the remarkable things about batch normalization is that it when it is used, the value of the network given some weights and an input $\mathcal{N}(\theta, x_i)$ is no longer well-defined (at least during training). This is because in the batch norm layers, we subtract μ and divide by σ, and these quantities depend on the other elements in the minibatch. Of course, this doesn't work at test time, because there is no minibatch. What then should we choose for the normalizing μ and σ for each hidden unit? Batch normalization solves this problem by keeping track of the μs and σs it encounters for each hidden unit–more precisely, it stores a running average of them. Then at test time, this running average is used.

Anytime there is a discontinuity between the behavior of the network at training time and at test time, this is a possible source of test error. To smooth over the difference between the μ, σ used at training time and at test time, Batch REnormalization [11] was introduced, by one of the original authors of the batch norm paper, Ioffe. This method scales by some mixture of the current minibatch σ and μ, with their running averages so far observed.

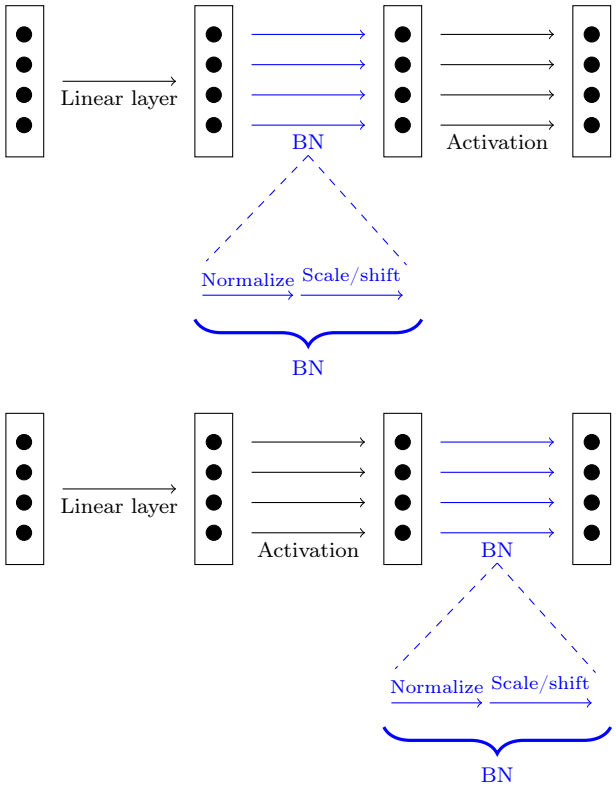
2.4 "Internal Covariate Shift"

Much of the benefit of batch normalization comes from its allowing for a higher learning rate. The original explanation for this was that it prevented "internal covariate shift" [12]. Namely, the weights on layer ℓ take their gradient step in the best direction given that the other weights are held constant; but of course the other layers do the same thing, and therefore it is hypothesized that the inputs that layer ℓ sees at time $t+1$ may be in a different range than the ones they've been training for at times $t, t-1, \ldots$. Therefore layer ℓ may not be well-trained to deal with its new inputs from layer $\ell - 1$.

Batch normalization purports to solve this by ensuring that if the values of the γs and βs on each layer do not change too much, the mean and the variance of the input to layer ℓ will certainly not change much, although the



(a) Typical layer in a network without batch normalization



(b) Two possible arrangements for a layer incoporating batch normalization

Figure 1: A normal layer in a deep neural network, and two possible architectures for a layer incorporating batch normalization. The batch normalization is done in parallel over the units, indicated by the parallel arrows. Batch normalization, crucially, involves normalizing the values that appear at a *particular* unit, but for *different* elements of the minibatch. (We normalize e.g. $u_1(x_1), u_1(x_2), \ldots, u_1(x_b)$, not e.g. $u_1(x_1), u_2(x_1), \ldots, u_n(x_1)$.) After this first normalization step, the scale/shift step applies an affine transformation $x \mapsto ax + b$, to each unit, with $a, b \in \mathbb{R}$ trainable parameters. After the minibatch is fixed, every part of the network is deterministic, but, without knowledge of the minibatch, the value of the network at x is not well-defined.

particular values of the inputs may, and according to the explanation, this allows the network to reach a better training error with less training time.

2.5 Combination of BN with other methods and architectures

We discuss several examples of how BN interacts with other optimization and regularization methods. We delay our discussion of the interaction of BN with dropout to a later section.

2.5.1 Batch normalization as less expensive whitening

One might first ask: why do we focus only on individual units, and not the entire layer? Our first inclination might be to whiten the entire layer u of hidden units by applying $u \mapsto \hat{\Sigma}^{-1/2}(u - \hat{\mu}_u)$, where $\hat{\Sigma}$ and $\hat{\mu}_u$ are the sample mean and covariance of the vector u calculated over the minibatch, leaving us with $\hat{u}$ with zero-centered, uncorrelated distributions. The first step of batch normalization can be seen as an approximation to this process that simply ignores the possible correlations between entries of u. This is for computational considerations. Typically, the size of the layer is already as large as is computationally feasible, to increase the expressive power of the network. Therefore an operation like inverting the $n \times n$ matrix Σ is likely to be too expensive.

2.5.2 Batch normalization and ridge regression

It is somewhat illogical to apply BN at each layer and also penalize the L^2 norm of the weights, because under BN, the norm of the weights as such makes no difference. More precisely, if we write the output of a network involving BN at each layer by $\mathcal{N}(x, \theta, \gamma, \beta)$, then scaling of the weights θ does not change the value of the network [13]

$$\mathcal{N}(x, \theta, \gamma, \beta) = \mathcal{N}(x, \alpha\theta, \gamma, \beta)$$

(Recall for this calculation that biases usually do not enter into BN networks, since they are absorbed into β.)

2.5.3 Batch normalization and RNNs

Batch normalization has been applied to recurrent neural networks (RNNs) with some success [1]. However, here batch normalization runs into the problem that the values of the network at a layer, used to do normalization, are not well-defined, because the same layer is reached multiple times. We might consider storing different statistics for each time the layer is reached, but the problem with this is that often test examples are longer than any single training example, e.g. in topic modeling, and so at test time we'd be reaching layers with no statistics stored at all. This is often solved by applying the *layer normalization* [1] method, in which the axis of normalization is changed: instead of fixing

the unit and varying the minibatch member, we fix the minibatch member and normalize over the units in a given layer. We keep statistics γ and β as before, and do the procedure largely the same at test time.

2.6 Precedent for stochastic techniques in network regularization

The regularization effect of noise in neural networks is a standard concept in the field. The 1995 article [4] by Christopher Bishop established that noising the inputs of a shallow neural network can be viewed as a second-order approximation to ridge regression. Precisely, suppose we define our objective function as the expectation over training data x and labels t

$$\mathcal{L}(\theta) \overset{(\text{def})}{=} \frac{1}{2}\mathbb{E}_{x,t}\|\mathcal{N}(\theta, x) - t\|^2.$$

Then, applying noise to the inputs x, and taking the expectation of the noise over the possible instantiations of the noise (a theme to which we will return), we obtain a new objective function, and take the perspective that noisy training is actually performing Monte Carlo approximation to this loss function, when it takes particular values of the noise.

$$\mathcal{L}_{\text{noised}}(\theta) \overset{(\text{def})}{=} \frac{1}{2}\mathbb{E}_{\epsilon}\mathbb{E}_{x,t}\|\mathcal{N}(\theta, x + \epsilon) - t\|^2,$$

where $\epsilon \sim \mathcal{N}(0, \eta^2)$ is some additive noise. Then by Taylor expanding y in ϵ at $\epsilon = 0$, rearranging the terms cleverly, using that the expectation of the noise is zero, we can obtain

$$\mathcal{L}_{\text{noised}}(\theta) = \mathcal{L}(\theta) + \eta^2 R.$$

R is a regularization term involving first and second derivatives of the network function $\mathcal{N}$ with respect to the inputs (not the weights). In fact, we can further show that omitting the second derivatives makes an $O(\eta^2)$ difference, and since we are only approximating up to η^2 anyway, we can omit these computationally expensive terms.

We will see that a similar calculation can be made for dropout, in section 4.2.

3 Investigating normalization techniques in optimization

The batch normalization technique, as we've seen, is "normalization-plus": it has a step to normalize the units, but then another step to reset the mean and variance. So to simplify the investigation, what can be accomplished with only normalization? We survey the use of normalization in machine learning, and along the way, we perform experiments regarding simpler techniques that may ultimately be brought to bear on batch normalization.

Feature normalization AKA *feature scaling* techniques are common in many machine learning algorithms, including k-means, SVMs, and principal component analysis. One general motivation may be thought of in terms of weighting the loss penalty appropriately. In the context of a neural network, suppose our network has a distribution of outputs $y \in \mathbb{R}^n$ over various inputs $x \in \mathbb{R}^m$. It is natural that some of our features may take place at different scales. However, these differences in scale can distort the loss function. Suppose our loss function is the expectation of the squared error over the training data (x, t):

$$\mathcal{L}(\theta) \stackrel{(\text{def})}{=} \frac{1}{2}\mathbb{E}_{x,t}\|\mathcal{N}(\theta, x + \epsilon) - t\|^2.$$

Suppose that at the beginning of training, the network misses its target in components i and j both with an average error of 50%. But suppose feature i occurs at a scale which is higher by a factor of 10. Then we will be weighting the error in feature i 10 times more in our loss function, and our gradient steps will devote themselves 10 times more to fixing the error in component i. Since we have no reason a priori to prefer this, we can normalize our penalty by the scale of the data at each component, so that this does not bias the optimization. Cf. the loss penalty for dropout in section 4.2.

3.1 Experiments with normalizing the nodes in polynomial regression

In polynomial regression, we attempt to fit a polynomial to some training data. For instance, if we have n training points (x, y) and a polynomial of degree n, we need the vector c of coefficients of the polynomial to solve

$$Vc = y,$$

where V is the $n \times n$ Vandermonde matrix with nodes the x_i.

There is a general normalization procedure that can be applied to this problem: instead of using training examples (x_i, y_i), we can, if we find it more convenient, use training examples $\nu(x_i), \nu(y_i)$, where ν is some function we know how to invert, then train the model on these examples, and at test time, use the function

$$x \stackrel{\nu}{\mapsto} \hat{x} \stackrel{\text{trained model}}{\longmapsto} \hat{y} \stackrel{\nu^{-1}}{\longmapsto} y.$$

If ν were an affine transformation $x \mapsto ax + b$, then as we train the model, we are in effect replacing V by VM, where M is an upper triangular matrix:

$$\underbrace{\begin{pmatrix} 1 & x_1 & \dots & x_1^{n-1} \\ 1 & x_2 & \dots & x_2^{n-1} \\ 1 & x_3 & \dots & x_3^{n-1} \\ \vdots & & & \\ 1 & x_n & \dots & x_n^{n-1} \end{pmatrix}}_{V} \underbrace{\begin{pmatrix} 1 & b & b^2 & \cdots & \\ & a & 2ab & \cdots & \\ & & a^2 & \cdots & \\ & & & \ddots & \\ & & & & a^{n-1} \end{pmatrix}}_{M}$$

What kind of ν might we be interested in, and why? If we consider how to solve $Vc = y$ for c (or more generally, find the best c we can given restricted time and computation), such a solution may involve solving for the inverse of V, or V^TV, in the normal equations. Or it may involve using an optimization technique such as SGD on the mean square error, in which case V^TV is the objective of the Hessian function. In either case, we care about the condition number of V^TV, or its square root, the condition number of V.

Which nodes are best for this condition number? In the simplest case of fitting a linear function through two points, some arithmetic shows that the minimum condition number of

$$\begin{bmatrix} 1 & x_1 \\ 1 & x_2 \end{bmatrix}$$

is achieved at $(x_1, x_2) = (1, -1)$ or $(-1, 1)$, which is exactly what we'd get from a linear transformation making the mean zero and the variance 1. We show this calculation in appendix A. As the number of points grows, solving for an optimum configuration analytically becomes difficult. We conduct numerical experiments on the best affine normalization for this problem.

More precisely, we investigate the effect of using affine normalization where we first normalize the nodes $x_1, \ldots, x_n$, and then directly set their mean and variance μ and σ. Displayed in figure 2 are the found best values of $\log \sigma$ and μ for these nodes of polynomial regression, with various distributions for the randomly-selected starting nodes.

3.2 Analytic bound for condition number of Vandermonde matrix

Polynomial regression is not our main interest, nor the condition number of Vandermonde matrices. However, there is an analytic bound for the condition number of the Vandermonde matrix that may be instructive. Recall that the condition number of an invertible matrix V can be expressed as $\|V\|\|V^{-1}\|$. It will be convenient to allow ourselves to work with other norms besides the operator 2-norm. In particular, the rows of V are easy to understand, and the columns of V^{-1} are easy to understand, so we'd like to work with $\|V\|_\infty$ and $\|V^{-1}\|_1$. Because

$$\frac{1}{\sqrt{n}}\|A\|_\infty \leq \|A\|_2 \leq \sqrt{m}\|A\|_\infty \quad \text{and} \quad \frac{1}{n}\|A\|_1 \leq \|A\|_2 \leq \sqrt{n}\|A\|_1$$

for any $m \times n$ matrix A, we have a crude bound

$$\kappa_2(V) \leq n\|V\|_\infty\|V^{-1}\|_1$$

for an $n \times n$ Vandermonde matrix V.

The infinity norm of a matrix is the largest absolute row-sum, the infinity norm of the Vandermonde matrix is just $1 + |x_*| + |x_*|^2 + \ldots + |x_*|^{n-1} = \frac{|x_*|^n - 1}{|x_*| - 1}$, where x_* is the node with maximum modulus. The inverse matrix to a square

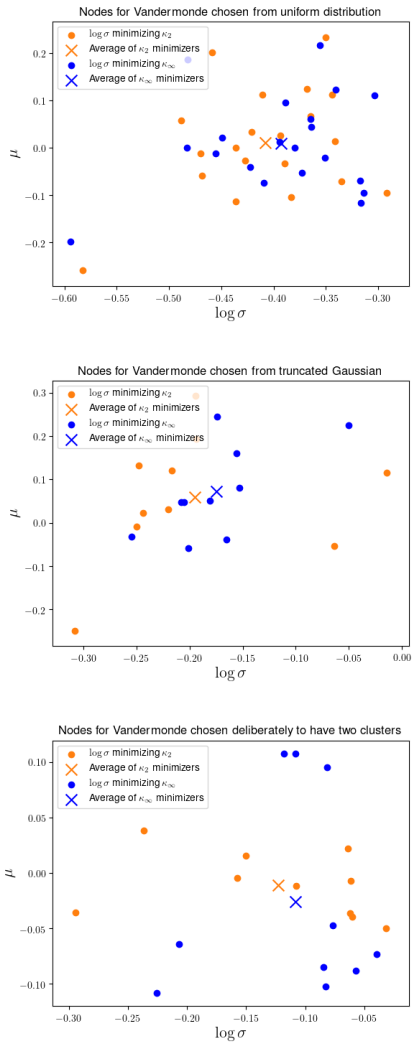


Figure 2: Results of searching for an optimal mean and variance for nodes for polynomial regression. Displayed are the values for $\log \sigma$ and μ that minimize $\kappa_2(V)$. In the first figure, points are chosen uniformly on an interval $[-100, 100]$, then normalized, and the optimal σ and μ are sought. In the second figure, a truncated Gaussian distribution on the same interval is used, with $-100 = -3\sigma$ and $100 = 3\sigma$. In the third example, we deliberately choose the points to be clustered into two groups, hypothesizing that a nonlinear normalization technique may be superior here.

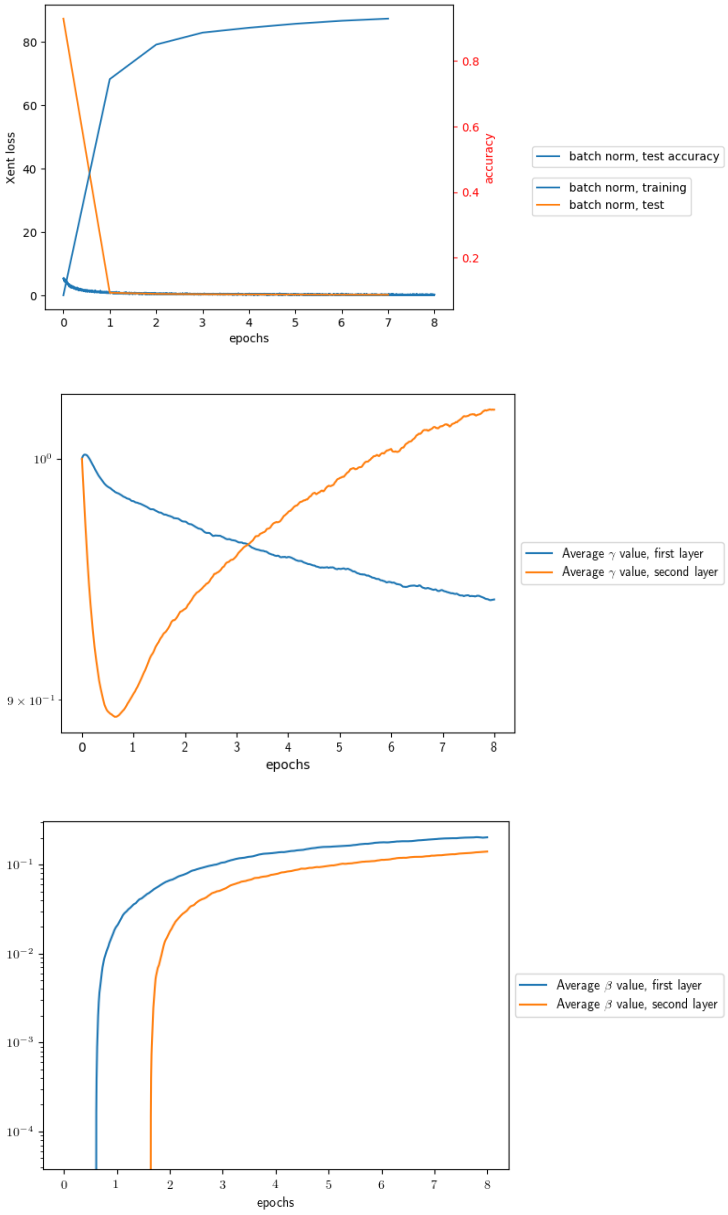


Figure 3: Here we train a two-layer ReLU network with batch normalization, and 20 and 25 hidden units on the first and second layers, using the Adam optimizer. We display the average γ and β values on each layer as the network trains. Compare to the values found in figure 2.

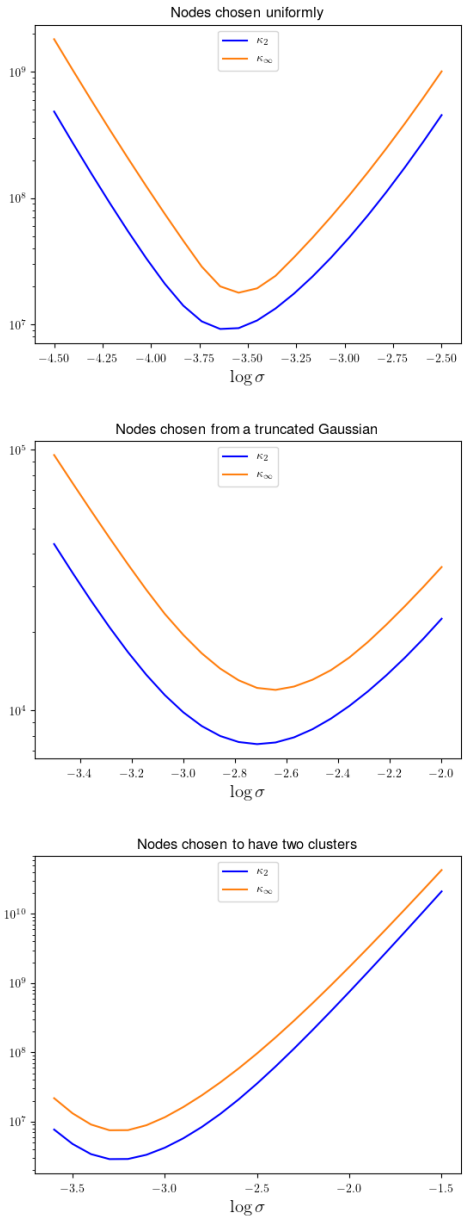


Figure 4: Average of 500 runs. The values of the condition numbers themselves have enormous variance, perhaps because $\|V^{-1}\|$ is so sensitive to changes in the smallest gap between two points.

Vandermonde matrix with distinct nodes $x_1, \ldots, x_n$ is the matrix whose ith column has entries the coefficients of the ith Lagrange basis polynomial

$$p_i(x) = \prod_{j \neq i} \frac{x - x_j}{x_i - x_j}$$

The sum of the absolute value of the coefficients of the numerator is bounded by $\prod_{j \neq i} 1 + |x_j|$ [8, p. 118], and so $\|V^{-1}\|_1$ is less than

$$\max_i \prod_{j \neq i} \frac{1 + |x_j|}{|x_j - x_i|}.$$

and therefore we can say that

$$\kappa_2(V) \leq \left(\frac{m^N - 1}{m - 1} \right) \cdot \max_{1 \leq i \leq n} \prod_{j \neq i} \left(\frac{1 + |x_j|}{|x_i - x_j|} \right).$$

The first factor depends only on the maximum modulus of the nodes. The second factor depends on the modulus of the nodes and also on the spacing of the points. In particular, when nodes are clustered together, it damages the conditioning more if these points have high modulus. Therefore, there may be a conflict between spreading out the nodes and lowering their moduli. For instance, if we have clusters of points that are close within the clusters, but the clusters are not close to each other, affine normalization $x \mapsto ax + b$ might not be a good idea for any value of a and b: choosing a too high will leave the points with high modulus, while choosing a too low will make the gaps between the points too small.

It is possible in such situations that we might want to use a nonlinear normalization function, such as $x \mapsto k\sigma(x)$ for some k, or perhaps $x \mapsto k\sigma(k'x)$ for some k, where σ is some sigmoid, centered about zero. This is an area for further investigation.

4 Dropout

4.1 Intro to dropout

Dropout was introduced in 2014 by Srivastava et al [20]. The method aims to prevent overfitting by deliberately reducing the model's capacity during the training phase, at random. Each time a labeled training input is selected to be used for a training step, a "mask" is sampled along with it, to be placed over the network. That is, a subset of the input units and hidden units are dropped out of the network, and the weights related to these units are frozen for this training step, since they are temporarily irrelevant to the loss. These units to be dropped out are chosen independently with a probability that is a hyperparameter. At test time, no mask is applied.

The removal of the mask at test time is an abrupt shift. The network is used to training when only say 80% of its units are present, and now they are all present. There are two possibilities to accommodate for this: either the weights are scaled down at test time by a factor of p, where p is the probability of dropout, or, in so-called "inverted dropout", the reverse is done: the weights are scaled *up* during training time. We may express inverted dropout simply by saying that each hidden and input unit is multiplied by a random variable M which is $\frac{1}{1-p}$ with probability $1-p$ and 0 with probability p, so M has expectation 1. Common values for the dropout probabilities are .8 for the input units and .5 for the hidden units, see e.g. [3].

Often regularization techniques alter the loss function in some way, and one possible perspective on dropout is that instead of using the usual loss function $\mathcal{L}(\mathcal{D}, \theta)$ (whatever that may be), it is now using the loss function

$$\mathbb{E}_{\mathcal{M}} \mathcal{L}_{\mathcal{M}}(\mathcal{D}, \theta) \tag{2}$$

where $\mathcal{L}_{\mathcal{M}}$ is the loss function for the network with mask $\mathcal{M}$ on it. Here $\mathcal{M}$ is the random variable of masks, with values in $\{0,1\}^k$, where k is the number of hidden units plus the dimension of the input. The idea is that each time a mask is a selected and a training step is taken, we are taking a Monte Carlo approximation of $\mathbb{E}_{\mathcal{M}} \mathcal{L}_{\mathcal{M}}(\mathcal{D}, \theta)$ by selecting a particular value of $\mathcal{M}$ and training on that.

4.2 Dropout as L^2 regularization

It was shown in the original dropout paper [20, p. 1950], that if we apply dropout to simple linear regression (so the only things available to dropout are the entries of the inputs), using the squared error penalty, then the implied dropout objective 2, which in this case can be written as $\mathbb{E}_{\mathcal{M}}\left(\frac{1}{2}\|(\mathcal{M} \odot X)\theta - t\|^2\right)$, where $\odot$ is entrywise multiplication, X is the feature matrix, and t is the vector of labels, satisfies

$$\mathbb{E}_{\mathcal{M}}\left(\frac{1}{2}\|(\mathcal{M} \odot X)\theta - t\|^2\right) = \frac{1}{2}\|pX\theta - y\|^2 + \frac{1}{2}p(1-p)\|\Gamma\theta\|^2$$

where p is the probability of a unit remaining in the network, and Γ is a diagonal matrix whose ith entry represents the scale of the ith entry of the inputs. Therefore dropout is doing L^2 regularization after doing feature scaling.

4.3 Dropout without dropping out

The central limit theorem tells us that if we have a large-enough independent sum of numbers with multiplicative Bernoulli noise $\mathcal{M}_i \sim B(p)$, and $q \overset{(\text{def})}{=} 1-p$,

$$\sum_{i=1}^{N} \mathcal{M}_i a_i,$$

then the sum

$$\frac{(M_1 a_1 + \cdots + M_n a_n) - \mathbb{E}(M_1 a_1 + \cdots + M_n a_n)}{\sqrt{a_1^2 pq + \cdots + a_n^2 pq}}$$

will converge in distribution to a standard normal $\mathcal{N}(0,1)$, provided that no individual a_i is too much bigger than the others, i.e. provided

$$\max_i a_i^2 = o(\sum_i a_i^2)$$

Therefore the "fast dropout" introduced by Wang and Manning [23] proposes to approximate the output of a dropout layer as the expectation $\mathbb{E}(M_1 a_1 + \cdots + M_n a_n)$ plus some Gaussian noise of variance $pq \sum a_i^2$, and either sampling this noise during forward propagation, or else integrating it out, in a Bayesian fashion.

4.4 Dropout as model averaging

One way to conceive of this implied loss function (2) of dropout training is that we are training 2^N different models, corresponding to the N possible nodes to be dropped out, and that these models are sharing parameters.

Pierre Baldi and Peter Sadowski [2] have worked toward making this intuition rigorous. In the implied objective function 2, they show that the expectation can be moved over a layer with softmax activation, at the cost of approximating the expectation (i.e. weighted additive mean) by a weighted geometric mean (WGM) [2, p. 85], namelykl;

$$\sigma \circ L(\mathbb{E}x_{\mathcal{M}}) = \sigma(\mathbb{E}_{\mathcal{M}} L(x_{\mathcal{M}})) = \mathrm{WGM}_{\mathcal{M}}(\sigma \circ L(x_{\mathcal{M}})) \tag{3}$$

$$\approx \mathbb{E}_{\mathcal{M}}((\sigma \circ L)(x_{\mathcal{M}}))$$

where $\sigma : \mathbb{R}^n \to \mathbb{R}^n$ is the function with ith component $\sigma(y)_i \overset{(\mathrm{def})}{=} \frac{e^{y_i}}{\sum_j e^{y_j}}$, and L is linear. They also discuss the extension of this approximation to the now-more-common case of the ReLU nonlinearity.

4.5 Future work: extending the model averaging perspective

It is worth noting, and a place for further work, that equation 3 above does not depend on the peculiarities of dropout: it is a property of the interaction of the weighted geometric mean and the softmax function (or as a special case, the logistic sigmoid). Namely, if we have independent random variables $X_1, \ldots, X_n$, and we look at the value of (say) the first entry after applying a softmax layer

$$\sigma_1(X) \overset{(\mathrm{def})}{=} \frac{\exp(\lambda X_1)}{\sum_{j=1}^n \exp(\lambda X_j)}$$

and then take the normalized geometric mean of these values, under possible realizations of the X_i

$$\frac{\prod_x \sigma_1(x)^{\mathbb{P}(X=x)}}{\sum_{i=1}^n \left(\prod_x \sigma_i(x)^{\mathbb{P}(X=x)}\right)}$$

then we can repeat the calculations leading to 3 and get

$$\frac{\prod_x \sigma_1(x)^{\mathbb{P}(X=x)}}{\sum_{i=1}^n \left(\prod_x \sigma_i(x)^{\mathbb{P}(X=x)}\right)} = \sigma_1(EX_1, \ldots, EX_n)$$

This has nothing to do with the peculiarity of X_1 being the output of a linear dropout layer.

One possibility for this type of analysis are where X_1 is the value of a given input x_1 after a batch normalization layer. In this case the randomness comes from which $b-1$ of the other $n-1$ elements are selected as fellow elements of the minibatch. Although in this case the $X_1, \ldots, X_n$ may not be independent, depending on the batch selection method, as we have seen in equation 1, if the data size is much larger than the batch size, the assumption of independence may not be so far off.

4.6 Dropout as (adaptive) regularization

Following on the theme of interpreting dropout as a modification to the loss function, Stefan Wager et al. have pursued an interpretation of dropout as "adaptive regularization", that is, regularization which attempts to learn from current and past training examples. Specifically, they show that if an output unit is a generalized linear model as a function of the input, as is the case in linear or logistic regression, then the noise from dropout can be approximated by a penalty which applies L^2 regularization after normalizing by an approximation to the Fisher information matrix [22, p. 5]. This is similar to the way that the AdaGrad algorithm [6] applies L^2 regularization after normalizing by an entrywise sum of squares of previous steps, and in fact, this normalizer approximates the Fisher information also [22, p. 7]. This has the effect of giving more emphasis to rare but highly discriminative features of the data set, which plain SGD often misses.

5 Investigating regularizing effects of BN and dropout

5.1 Coadapatation as a source or symptom of overfitting

It has been informally described that dropout helps overfitting by preventing "coadaptation" between the weights [20], or in other words, it helps overfitting by preventing the units from relying on each other to correct their mistakes. But "coadaptation" remains a nebulous concept. In the following set of experiments, we consider coadaptation as the expected decrease in performance when dropout is applied *at test time*. The more that a network is able to maintain its accuracy when nodes are dropped out, the less "coadapted" it is, under this somewhat circular definition.

The motivating questions for such experiments are: Is "coadaptation" an identifiable type of overfitting? Are there multiple "kinds" of overfitting, which various regularizers fight in different ways? Can we determine whether various regularizers fight coadaptation? If we find that a regularizer does not fight coadaptation, can it be made to do so in a way that improves its performance?

As a first step toward these questions, we investigate how "vulnerable" various networks are to dropout. Figure 4 shows how much various networks are affected by dropping out units *during test time*. We look at a basic fully-connected network with one hidden unit and twenty hidden units, as well as a network trained with batch normalization on this hidden layer, and one trained with dropout. We might suspect that the network trained with dropout is the least vulnerable to dropout at test time, and indeed that is what we observe.

An interesting result is that the network trained with batch normalization was actually *more* vulnerable to dropout at test time than the basic network was, as if batch normalization learns another way not to overfit that is incompatible with the dropout way. Cf. section 5.2, which deals with the compatibility of dropout with batch normalization.

	Accuracy after training	Avg accuracy when tested with dropout	Accuracy reduction
Basic network	82.0%	64.8%	21.0%
BN network	89.8%	60.0%	33.2%
Dropout network	83.1%	72.0%	13.4%

Figure 5: Measuring "coadaptation" in trained networks by observing their test performance when dropped out. The networks were fully connected, training on MNIST with one hidden layer of twenty units. In each case, random dropout masks were applied at test time, and the average accuracy over twenty sweeps through the test data was recorded.

5.2 BN and Dropout together?

When asking whether batch normalization and dropout regularize and optimize in similar ways, an interesting question is whether they are complementary in practice.

Initially dropout and BN were used together, but got in each other's way, for a simple reason. To illustrate, let us consider "inverted dropout", where the masks are Bernoulli random variables multiplied by $\frac{1}{p}$. But while this mask preserves the mean, the formula $\operatorname{Var}(XY) = \operatorname{Var} Y\,\mathbb{E}(X^2) + \operatorname{Var} X\,\mu_Y^2$ when $X \perp Y$ shows that inverted dropout adds an extra term of $\mathbb{E}(X^2)\operatorname{Var}(\frac{1}{p}B(p)) = (\sigma^2_{x,\text{data}} + \mu^2_{x,\text{data}}) \cdot \frac{1-p}{p}$ at training time, which is removed at test time. So this extra variance is taught to the BN layer, and then at test time, BN has adjusted to the wrong variance.[1]

[1]The above description is somewhat oversimplified: the typical layer would consist of first

More recent techniques, such as in [10], have corrected this discrepancy by recalculating the moving average values for μ and σ that are used at test time: after training, simply pass some of the training set through the trained network until we've learned new average values for the μ and σ, and use those at test time.

6 Considering stochastic regularizers in networks with infinitely wide hidden layers

Works in the mid-1990s by Neal [19] and MacKay [18] investigated the possibility that one could replace supervised training of neural networks by performing inference with Gaussian processes, noting that as we let hidden layers become infinitely wide, a neural network with independent Gaussian priors over its weights is well-approximated by a Gaussian process. In this section, we investigate how BN and dropout interact with this framework.

6.1 Brief intro to Gaussian processes and Bayesian neural nets

Suppose we have a deep neural network N with the following architecture:

$$x^0 \xrightarrow{\text{Aff}} z^0 \xrightarrow{\text{NL}} x^1 \xrightarrow{\text{Aff}} z^1 \xrightarrow{\text{NL}} \cdots \xrightarrow{NL} x^{N-1} \xrightarrow{\text{Aff}} z^N$$

which alternates between affine maps and entry-wise nonlinearities. Suppose we have a continuous distribution of inputs $x^0 \in \mathbb{R}^{d(x^0)}$. Instead of optimizing some loss function via gradient descent or the like, we proceed in a Bayesian manner, with a prior distribution over our weights and biases, and use training data to update these priors. Namely, we give each weight an independent Gaussian prior with mean zero and variance $\sigma_w^2/\sqrt{n_\ell}$, where n_ℓ is the width of the layer of the unit (which we take to be large). Suppose we use ReLU activation units after fully connected layers.

It is shown in [14, p. 12] that

Theorem 1. *Each hidden unit of this network is a Gaussian process on the space of inputs with mean zero. The covariance function is the same for units on the same layer, and so can be denoted unambiguously by $K^\ell(x, x')$. It can be*

dropout then an affine transformation, then batch normalization, then an activation unit. The calculations in [16] show that if the layer is very wide and the affine transformation is very connected, unlike say a convolutional neural network, then this can help to fix the discrepancy between the variances at test time and at training time. This agrees with what has been observed in practice.

computed recursively by the formula

$$K^0(x,x') = \mathbb{E}((z_j^0 \mid x)(z_j^0 \mid x')) = \sigma_b^2 + \sigma_w^2 \frac{\langle x, x' \rangle}{d(x^0)}$$

$$K^\ell(x,x') = \sigma_b^2 + (2\pi)^{-1}\sigma_w^2 \sqrt{K^{\ell-1}(x,x)K^{\ell-1}(x',x')} \left(\sin\theta_{x,x'}^{\ell-1} + (\pi - \theta_{x,x'}^{\ell-1})\cos\theta_{x,x'}^{\ell-1}\right)$$

$$\theta_{x,x'}^{\ell-1} \overset{(def)}{=} \cos^{-1}\left(\frac{K^{\ell-1}(x,x')}{\sqrt{K^{\ell-1}(x,x)K^{\ell-1}(x',x')}}\right)$$

We provide a large part of the proof in appendix C, because we want to alter the derivation to apply to dropout and batch normalization.

6.2 Dropout and batch normalization in networks with infinitely wide hidden layers

If we let the width of the hidden layers approach infinity, and apply (inverted) dropout to those hidden layers, our intuition is that not much changes, because there are still infinitely many remaining units, but perhaps the variance has increased. This is borne out by the following calculation. For the first layer,

$$\begin{aligned}
\mathrm{Var}(z_1^0) &= E\Big[(W_{11}^0 M_1^0 x_1 + \cdots + W_{1,d(x^0)}^0 M_{d(x^0)}^0 x_{d(x^0)} + b_1^0) \\
&\qquad \cdot (W_{11}^0 M_1^0 x_1' + \cdots + W_{1,d(x^0)}^0 M_{d(x^0)}^0 x_{d(x^0)}' + b_1^0)\Big] \\
&= \sigma_w^2 \mathbb{E}[(M_1^0)^2]\frac{\langle x, x' \rangle}{d(x^0)} + \sigma_b^2 \\
&= p^{-1}\sigma_w^2 + \sigma_b^2
\end{aligned}$$

And we can calculate the covariance function recursively as

$$\begin{aligned}
\mathrm{Cov}((z_1^\ell \mid x)(z_1^\ell \mid x')) &= \mathbb{E}\Big[(z_1^\ell \mid x)(z_1^\ell \mid x')\Big] \\
&= \mathbb{E}\Big[(W_{11}^\ell M_1^\ell x_1 + \cdots + W_{1,d(x^\ell)}^\ell M_{d(x^\ell)}^\ell x_{d(x^\ell)} + b_1^\ell) \\
&\qquad\qquad \cdot (W_{11}^\ell M_1^\ell x_1' + \cdots + W_{1,d(x^\ell)}^0 M_{d(x^\ell)}^\ell x_{d(x^\ell)}' + b_1^\ell)\Big] \\
&= \frac{\sigma_w^2}{d(x^\ell)} \cdot d(x^\ell) \cdot \mathbb{E}[(M_1^\ell)^2]\mathbb{E}\Big[(x_1^\ell \mid x)(x_1^\ell \mid x')\Big] + \sigma_b^2 \\
&= p^{-1}\sigma_w^2 \cdot \mathbb{E}\Big[(\phi(z_1^{\ell-1} \mid x)\phi(z_1^{\ell-1} \mid x'))\Big] + \sigma_b^2
\end{aligned}$$

suppressing the layer superscript for x^ℓ and x'^ℓ. This gives us the same structure as before, with an additional factor of p^{-1} on (one term of) the covariance function of each dropout layer.

The situation with batch normalization is more delicate, because of the dependence on the minibatch. There are a couple of possible approaches: it's possible that one might view this as a stochastic process on the b-fold Cartesian product of the input space, with points corresponding to minibatch selections. (Or perhaps we'd use a subset of this space, depending on the selection method.) Another possibility is to look at it as a stochastic process on the input space, *conditional* on the other members of the minibatch. Under this approach, the result after the normalization step would be

$$\hat{z}_j \mid x_i = \langle \hat{\Sigma} W_j^0, W_j^0 \rangle^{-1/2} \langle x_i - \hat{\mu}, W_j^0 \rangle$$

where $\hat{\mu}$ and $\hat{\Sigma}$ are the sample mean and variance of the minibatch. The calculation of the covariance and mean functions get more complicated with the factor of $\langle \hat{\Sigma} W_j^0, W_j^0 \rangle$, and simplifying these is a direction for future work.

7 Further work

There is a growing body of literature on stochastic regularization techniques performing variational inference, which we would like to bring to bear on this research. [7] establishes this for dropout, and [5] for SGD, whose noise we have remarked is a simpler version of batch normalization's noise. [15] has made similar investigations for minibatch training.

The paper [7] also aims to estimate model uncertainty using dropout, and papers [21] and [17], released this year, do the same line of investigation for batch normalization, using the Bayesian approach we review in section **??**. These seem like promising methods for addressing the questions we started with.

8 Conclusion

Batch normalization certainly acts as both an optimization tool and a regularizer; however, it is clear that there is more than one type of overfitting, in a manner of speaking, since BN does worse under dropout. Both the stochastic aspect and the normalizing aspect of batch normalization have histories in regularization.

Dropout has a rich variety of interpretations; as an average of models, as additive noise on the units, and as adaptive regularization. Batch normalization is still less understood. If we can succeed in a key principle or principles that make these methods work, there are possibilities that they can eventually be generalized or improved. In particular, generalizing batch normalization may have a lot of potential. It deserves a place in a broad class of techniques which we call "division of labor" techniques, where one or more attributes of the data are isolated to be directly selected. There is no reason in principle why such techniques could not be extended to other attributes we're interested in, and which are relevant to optimization or regularization. Examples of these could be statistics encoding second-order information about the network, for example.

A Nodes for polynomial regression which minimize $\kappa_2(V)$ in the linear case

Suppose we want to minimize the condition number of

$$\begin{bmatrix} 1 & x_1 \\ 1 & x_2 \end{bmatrix}.$$

We look at minimizing $\frac{\sigma_1^2}{\sigma_2^2}$, where σ_i are the singular values of V, or equivalently, σ_i^2 are the eigenvalues of V^TV. The characteristic polynomial of V^TV is $x^2 - [2 + (x_1^2 + x_2^2]x + (x_1 - x_2)^2$, with two positive roots

$$\begin{aligned} \sigma_1^2, \sigma_2^2 &= 1 + \frac{x_1^2 + x_2^2}{2} \pm \sqrt{\left(1 + \frac{x_1^2 + x_2^2}{2}\right)^2 - (x_1 - x_2)^2} \\ &\overset{\text{(def)}}{=} a \pm \sqrt{a^2 - b^2} \end{aligned}$$

and

$$\kappa_2(V)^2 = \frac{\sigma_1^2}{\sigma_2^2} = \frac{1 + \sqrt{1 - (b/a)^2}}{1 - \sqrt{1 - (b/a)^2}}$$

So to minimize this quantity, we need only minimize b/a, and using the KKT conditions gets us that $\{x_1, x_2\} = \{1, -1\}$.

B Computation of minibatch gradient

If we sample our minibatch without replacement, then the indices $i_j \in \{1, \ldots, N\}$ are no longer independent. The variance of each individual $\nabla f(x_{i_j})$ is still $\frac{1}{N}\left(\sum g_i g_i^T\right) - gg^T$, but now we have to account for the covariance of distinct elements:

$$\begin{aligned} &\operatorname{Var}\left(\frac{1}{b}\left(\nabla f(x_{i_1}) + \cdots + \nabla f(x_{i_b})\right)\right) \\ &= \frac{1}{b^2} \cdot b \cdot \left[\left(\frac{1}{N}\sum_{i=1}^{N} g_i g_i^T\right) - gg^T\right] + \frac{1}{b^2}\binom{b}{2} \cdot 2 \cdot \operatorname{Cov}(g_{i_1}, g_{i_2}) \end{aligned}$$

The $\binom{b}{2}$ accounts for the various distinct indices $i_j, i_k \in \{i_1, \ldots, i_b\}$ we could choose. The factor of 2 corresponds to the fact that we need to account for both $\operatorname{Cov}(g_{i_j}, g_{i_k})$ and $\operatorname{Cov}(g_{i_k}, g_{i_j})$. And we are justified in looking only at $\operatorname{Cov}(g_{i_1}, g_{i_2})$, because the others must be the same, by symmetry.

To find $\operatorname{Cov}(g_{i_1}, g_{i_2})$, we have to examine expressions like $\mathbb{E}(g_{i_1} g_{i_2}^T) - gg^T$. It is convenient to write out $\left(\sum_{i=1}^n g_i\right)\left(\sum_{j=1}^n g_j\right)^T$ as $D + O$, the sum of the n diagonal terms and the $n(n-1)$ off diagonal terms. Then $\mathbb{E}(g_{i_1} g_{i_2}^T) - gg^T$

becomes

$$\begin{aligned}\mathrm{Cov}(g_{i_1}, g_{i_2}) &= \frac{1}{N}\frac{1}{N-1}O - \frac{1}{N^2}(D+O) \\ &= \frac{1}{N^2}\frac{1}{N-1}O - \frac{1}{N^2}D\end{aligned}$$

Since O has $N(N-1)$ terms, and D has N terms, we see that this should shrink as N increases[2], as the g_{i_j} become closer to independent. This is true regardless of the batch size b. We can write this more simply as

$$\begin{aligned}\mathrm{Var}\left(\frac{1}{b}\left(\nabla f(x_{i_1}) + \cdots + \nabla f(x_{i_b})\right)\right) = \frac{1}{b}\left[\left(\frac{1}{N}\sum_{i=1}^{N} g_i g_i^T\right) - gg^T\right] \\ + \frac{b-1}{b}\left(\frac{1}{N}\left(\mathbb{E}(g_i g_i^T) + \mathbb{E}_{i\neq j} g_i g_j^T\right)\right).\end{aligned}$$

C Derivation of the covariance function for GP

Remark 1. *The dimensions of z^ℓ and x^ℓ aren't equal in general, but the dimensions of $z^{\ell-1}$ and x^ℓ are equal.*

Lemma 1 ([14], p. 4)**.** *For each layer x^ℓ, the components of the random variable $(x^\ell \mid x^0) \in \mathbb{R}^{d(x^\ell)}$ are independent for each layer ℓ. Also, the components of $(z^\ell \mid x^0) \in \mathbb{R}^{d(z^\ell)}$ are independent for each ℓ.*

Proof. This is a simple induction argument. We can see that $(z^0 \mid x^0)$ has independent entries, because $z^0 = Ax^0 + b$ where $A \in \mathbb{R}^{d(x^0)\times d^{z^0}}$ and $b \in \mathbb{R}^{d(z^0)}$ have independent Gaussian entries. Then $(x^1 \mid x^0)$ does as well, because its components are entry-wise functions of z^0, and so on. □

Now suppose that we take the dimensions of the hidden layers $d(z^k),\ 0 \leq k < N$, to be large (and so along with them, $d(x^{k+1})$ are large, for $0 < k+1 < N$). Then each entry z_i^ℓ, for $1 < \ell \leq N-1$ and $1 \leq i \leq d(x^\ell)$, we have that z_i^ℓ is a sum of many independent random variables

$$z_i^j = \langle W_i^j, x^j \rangle + b$$

and so can be approximated by a Gaussian, provided that none of the terms are too much bigger than the others. (See condition 4.3.) To ensure that this variance does not explode, we scale the variance of the entries of W_j^i for each i and j by the square root of the width of the preceding layer, $(d(x^j))^{-1/2}$.

Therefore, the individual hidden units $z_i^\ell \mid x$ are well-approximated by Gaussian distributions.

[2]Provided the g_i are bounded.

Lemma 2. *Given inputs $x, x' \in \mathbb{R}^{d(x^0)}$, and given any entry z_i^ℓ, for $1 < \ell \leq N-1$ and $1 \leq i \leq d(x^\ell)$, the random vector $\begin{bmatrix} z_i^\ell \mid x \\ z_i^\ell \mid x' \end{bmatrix}$ is well-approximated by a multivariate Gaussian.*

Proof. This is merely an appeal to the multivariate central limit theorem, and the argument is the same as before: we argue that $x_i^\ell \perp x_j^\ell \mid x$ for $i \neq j$, and similarly given x', and then

$$\begin{bmatrix} z_i^\ell \mid x \\ z_i^\ell \mid x' \end{bmatrix} = W_{i1}^\ell \begin{bmatrix} x_1^\ell \mid x \\ x_1^\ell \mid x' \end{bmatrix} + \cdots + W_{i,d(x^\ell)}^\ell \begin{bmatrix} x_{d(x^\ell)}^\ell \mid x \\ x_{d(x^\ell)}^\ell \mid x' \end{bmatrix} + b^\ell$$

a sum of many independent terms. □

Since the same proof can be extended to any number of inputs, we have met the conditions for z_j^i to be a Gaussian process, indexed by the space of inputs x.

It remains to find the mean and covariance functions for this Gaussian process. The mean of all the hidden units is zero, because of the zero-centered independent priors over the weights. For the covariance function, first note that the symmetry shows that the distributions of z_i^ℓ are the same for all $i \in \{1, \ldots, d(z^\ell)\}$, and therefore can be written unambiguously as $K^\ell(x, x')$ for $0 \leq \ell \leq N-1$. To find K^0,

$$\begin{aligned} K^0(x, x') &= \mathbb{E}((z_1^0 \mid x)(z_1^0 \mid x')) \\ &= \mathbb{E}\Big[(W_{11}^0 x_1 + \cdots + W_{1,d(x^0)}^0 x_{d(x^0)} + b_1^0)(W_{11}^0 x_1' + \cdots + W_{1,d(x^0)}^0 x_{d(x^0)}' + b_1^0)\Big] \\ &= \sigma_w^2 \frac{\langle x, x' \rangle}{d(x^0)} + \sigma_b^2, \end{aligned}$$

since distinct weights are independent, and the weights are independent of the bias.[14]

Given a covariance function $K^{\ell-1}(x, x') \overset{\text{(def)}}{=} \mathrm{Cov}((z_1^{\ell-1} \mid x)(z_1^{\ell-1} \mid x'))$, the covariance $\mathrm{Cov}((z_1^\ell \mid x)(z_1^\ell \mid x'))$ is [14]

$$\begin{aligned} \mathrm{Cov}((z_1^\ell \mid x)(z_1^\ell \mid x')) &= \mathbb{E}\Big[(z_1^\ell \mid x)(z_1^\ell \mid x')\Big] \\ &= \mathbb{E}\Big[(W_{11}^\ell x_1 + \cdots + W_{1,d(x^\ell)}^\ell x_{d(x^\ell)} + b_1^\ell) \\ &\qquad \cdot (W_{11}^\ell x_1' + \cdots + W_{1,d(x^\ell)}^0 x_{d(x^\ell)}' + b_1^\ell)\Big] \\ &= \frac{\sigma_w^2}{d(x^\ell)} \cdot d(x^\ell) \cdot \mathbb{E}\Big[(x_1^\ell \mid x)(x_1^\ell \mid x')\Big] + \sigma_b^2 \\ &= \sigma_w^2 \cdot \mathbb{E}\Big[(\phi(z_1^{\ell-1} \mid x)\phi(z_1^{\ell-1} \mid x'))\Big] + \sigma_b^2 \end{aligned}$$

We have suppressed the layer superscript for x^ℓ and x'^ℓ for readability.

Let us pause to compute the value of $\mathbb{E}\Big[(\phi(z_1^{\ell-1} \mid x)\phi(z_1^{\ell-1} \mid x'))\Big]$.

$$\mathbb{E}\Big[(\phi(z_1^{\ell-1} \mid x)\phi(z_1^{\ell-1} \mid x'))\Big] = \int_{x_1,x_2>0} x_1 x_2 \, dX$$

where $X \in \mathbb{R}^2$ is a random variable with distribution $\sim \mathcal{N}(0, \Sigma^{\ell-1})$, where

$$\Sigma^{\ell-1} \overset{\text{(def)}}{=} \begin{bmatrix} K^{\ell-1}(x,x) & K^{\ell-1}(x,x') \\ K^{\ell-1}(x',x) & K^{\ell-1}(x',x') \end{bmatrix}$$

The region of integration $\{(x_1, x_2) \in \mathbb{R}^2 : x_1, x_2 > 0$ is due to the nonlinearity function.

Remark 2. *In case $x = x'$, we have that this integral is just $\int_0^\infty x^2 dX$, for $X \sim N(0, K^{\ell-1}(x,x))$, or just half of* $\mathrm{Var}(X)$*, giving simply* $\mathbb{E}\Big[(\phi(z_1^{\ell-1} \mid x)\phi(z_1^{\ell-1} \mid x))\Big] = \frac{K^{\ell-1}(x,x)}{2}$.

Evaluating the integral for the general case $x \neq x'$ gives

Theorem 2. *[14, p. 12] The covariance function for z_i^ℓ is the same for all $i \in \{1, \ldots, d(z^\ell)\}$, and therefore can be written unambiguously as $K^\ell(x,x')$ for $0 \leq \ell \leq N-1$. It can be given recursively by*

$$K^0(x,x') = \mathbb{E}((z_j^0 \mid x)(z_j^0 \mid x')) = \sigma_b^2 + \sigma_w^2 \frac{\langle x, x' \rangle}{d(x^0)}$$

$$K^\ell(x,x') = \sigma_b^2 + (2\pi)^{-1}\sigma_w^2 \sqrt{K^{\ell-1}(x,x)K^{\ell-1}(x',x')} \left(\sin\theta_{x,x'}^{\ell-1} + (\pi - \theta_{x,x'}^{\ell-1})\cos\theta_{x,x'}^{\ell-1}\right)$$

$$\theta_{x,x'}^{\ell-1} \overset{(def)}{=} \cos^{-1}\left(\frac{K^{\ell-1}(x,x')}{\sqrt{K^{\ell-1}(x,x)K^{\ell-1}(x',x')}}\right)$$

References

[1] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. "Layer normalization". In: *arXiv preprint arXiv:1607.06450* (2016).

[2] Pierre Baldi and Peter Sadowski. "The dropout learning algorithm". In: *Artificial intelligence* 210 (2014), pp. 78–122.

[3] Pierre Baldi and Peter J Sadowski. "Understanding dropout". In: *Advances in neural information processing systems*. 2013, pp. 2814–2822.

[4] Chris M Bishop. "Training with noise is equivalent to Tikhonov regularization". In: *Neural computation* 7.1 (1995), pp. 108–116.

[5] Pratik Chaudhari and Stefano Soatto. "Stochastic gradient descent performs variational inference, converges to limit cycles for deep networks". In: *2018 Information Theory and Applications Workshop (ITA)*. IEEE. 2018, pp. 1–10.

[6] John Duchi, Elad Hazan, and Yoram Singer. "Adaptive subgradient methods for online learning and stochastic optimization". In: *Journal of Machine Learning Research* 12.Jul (2011), pp. 2121–2159.

[7] Yarin Gal. "Uncertainty in deep learning". In: *University of Cambridge* (2016).

[8] Walter Gautschi. "On inverses of Vandermonde and confluent Vandermonde matrices". In: *Numerische Mathematik* 4.1 (1962), pp. 117–123.

[9] Ian Goodfellow et al. *Deep learning*. Vol. 1. MIT press Cambridge, 2016.

[10] Dan Hendrycks and Kevin Gimpel. "Adjusting for Dropout Variance in Batch Normalization and Weight Initialization". In: *arXiv preprint arXiv:1607.02488* (2016).

[11] Sergey Ioffe. "Batch renormalization: Towards reducing minibatch dependence in batch-normalized models". In: *Advances in Neural Information Processing Systems*. 2017, pp. 1945–1953.

[12] Sergey Ioffe and Christian Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". In: (Feb. 2015). eprint: 1502.03167. URL: https://arxiv.org/pdf/1502.03167.

[13] Twan van Laarhoven. "L2 regularization versus batch and weight normalization". In: *arXiv preprint arXiv:1706.05350* (2017).

[14] Jaehoon Lee et al. "Deep neural networks as gaussian processes". In: *arXiv preprint arXiv:1711.00165* (2017).

[15] Chris Junchi Li et al. "Batch size matters: A diffusion approximation framework on nonconvex stochastic gradient descent". In: *stat* 1050 (2017), p. 22.

[16] Xiang Li et al. "Understanding the disharmony between dropout and batch normalization by variance shift". In: *arXiv preprint arXiv:1801.05134* (2018).

[17] Ping Luo et al. "Understanding Regularization in Batch Normalization". In: *arXiv preprint arXiv:1809.00846* (2018).

[18] David JC MacKay. "Gaussian processes-a replacement for supervised neural networks?" In: (1997).

[19] Radford M Neal. *Bayesian learning for neural networks*. Vol. 118. Springer Science & Business Media, 2012.

[20] Nitish Srivastava et al. "Dropout: a simple way to prevent neural networks from overfitting". In: *The Journal of Machine Learning Research* 15.1 (2014), pp. 1929–1958.

[21] Mattias Teye, Hossein Azizpour, and Kevin Smith. "Bayesian Uncertainty Estimation for Batch Normalized Deep Networks". In: *arXiv preprint arXiv:1802.06455* (2018).

[22] Stefan Wager, Sida Wang, and Percy S Liang. "Dropout training as adaptive regularization". In: *Advances in neural information processing systems*. 2013, pp. 351–359.

[23] Sida Wang and Christopher Manning. "Fast dropout training". In: *international conference on machine learning*. 2013, pp. 118–126.

[24] Xiao Zhang et al. "Learning One-hidden-layer ReLU Networks via Gradient Descent". In: *arXiv preprint arXiv:1806.07808* (2018).