Regularization methods and gradient check

Data Science 2 CS 209b

Javier Zazo Pavlos Protopapas

March 17, 2018

Abstract

In this section, we will motivate the reason why dropout is regarded as a regularization technique. We will consider some simple neural networks where a better understanding of dropout is available and infer this interpretation to other deep networks. In addition to this, we will describe "batch normalization" to enhance training speed in deep networks. Finally, we will present gradient checking as a technique to validate backpropagation gradients. This method is important to verify manual implementations of the optimization procedure, or novel functions not present in the framework's library.

1 Dropout

Dropout is a regularization technique for deep neural networks, first proposed in [1, 2]. It can be used in situations where overfitting or high variance is expected or known to occur, and can be used in combination or as an alternative to L2 penalization. The method is employed at training time, by eliminating the output of some units randomly (setting the output to zero). The frequency for a unit in a given layer to be switched off is governed by the dropout probability, and this probability can vary for different hidden layers, or input. This idea is depicted in Figure 1.

With the employment of dropout at training time, each training example can be viewed as providing gradients from different, randomly sampled architectures. The result of these operations can be regarded as an ensemble of

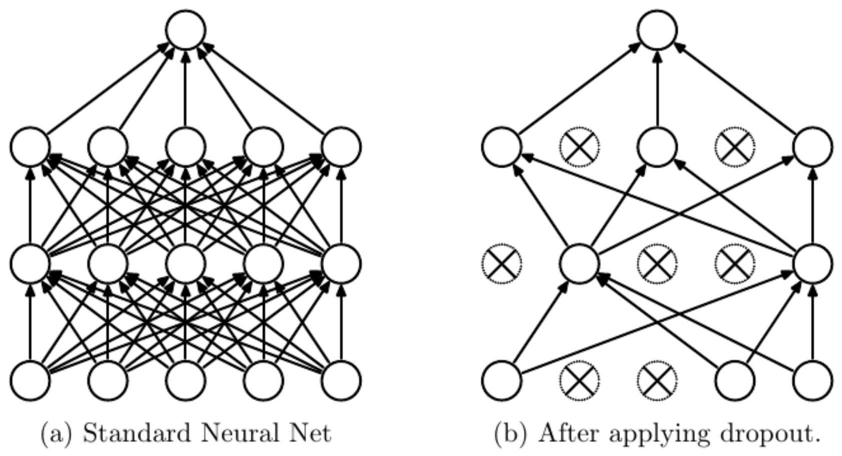

Figure 1: Dropout example, from [1].

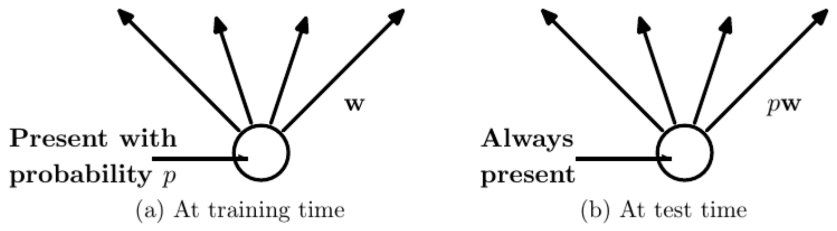

Figure 2: Dropout unit at training and testing times, from [1].

different neural networks with better generalization capabilities compared to the original architecture. At test and prediction times, all units are present, and the disconnection of units is no longer performed, see Figure 2.

The main motivation behind the algorithm is to prevent the co-adaptation of feature detectors for a set of neurons, and avoid overfitting. It works by enforcing the neurons to develop an individual role on their own given a population behavior, rather than relying two much on a specific set of neurons that are always present. In [2] it is argued that this method reduces the chances of complex co-adaptations that reduce the chance of individual improvements for units of the network.

Dropout was first proposed eliminating units at training time, and re-weighting network parameters at testing time (and prediction times) to correct the offset for eliminating units. Current frameworks normally implement *inverted dropout*, where weighting is performed at training and no additional correction is applied at testing or when predicting values. In particular, for layer l,

$$\begin{aligned} z^{[l]} &= \frac{1}{p_l} W^{[l]} D^{[l]} a^{[l-1]} + b^{[l]} \\ a^{[l]} &= g(z^{[l]}), \end{aligned} \tag{1}$$

where $z^{[l]}$ corresponds to the linear combination of the outcomes of previous layers $a^{[l-1]}$, $W^{[l]}$ correspond to the network's optimization weights and $b^{[l]}$ is an offset. Diagonal matrix $D^{[l]}$ corresponds to a realization of Bernoulli variables with probability of keeping the output p_l on every diagonal element, and $g(\cdot)$ refers to the activation function of the neural unit (ReL, tanh, etc.), element-wise. Feedforward and backpropagation would carry this weight parameter $\frac{1}{p_l}$ at training time, but should be omitted at evaluation time. This form of *inverted dropout* is more common in most implementations.

1.1 Dropout in linear networks

Consider a linear neural network for which all activation units correspond to the identity function. For a single training example and layer l, we get

$$z^{[l]} = W^{[l]} D^{[l]} z^{[l-1]}, \tag{2}$$

when using normal dropout.

For a fixed input vector $z^{[l-1]}$ and weights $W^{[l]}$, the expectation of the activity of all units taken over all possible realizations of the dropout variables, is given by

$$\mathbb{E}\{z^{[l]}\} = p_l W^{[l]} z^{[l-1]}, \tag{3}$$

where p_l corresponds to the probability of keeping a unit on layer l. The ensemble average can therefore be easily computed using feedforward propagation in the original network, replacing weights $W^{[l]}$ with $p_l W^{[l]}$.

1.1.1 Dynamics of a single linear unit

Consider the error terms of a single linear unit E^{ens} and E^{d} for the averaged ensemble network and dropout network, respectively. For a single training example pair $(x^{(i)}, y^{(i)})$ we get

$$E^{\text{ens}} = (y^{(i)} - p_l W^{[l]} x^{(i)})^2 \tag{4}$$

$$E^{\text{d}} = (y^{(i)} - W^{[l]} D^{[l]} x^{(i)})^2. \tag{5}$$

Errors can be summed over all examples, but for simplicity we consider a single example. We assume there is a single layer and refer to it with l.

Taking the gradient over the error function (4), and the expected gradient for (5), it can be shown as in [3] that

$$\mathbb{E}\{E^{\mathrm{d}}\} = E^{\mathrm{ens}} + \sum_{r=1}^{n_1} \frac{1}{2} \operatorname{var}(D^{[l]})(x_r^{(i)})^2 w_r^2 \tag{6}$$

which corresponds to a ridge regression over variable $w = [w_1, \ldots, w_{n_1}]^T$.

Thus, remarkably, the expectation of the gradient with dropout is the gradient of the regularized ensemble error. This justifies the regularization effect of dropout for linear units.

1.2 Dropout in linear regression

Let's consider applying dropout to the classical problem of linear regression. We have a set of features given by $X \in \mathbb{R}^{m\times n}$ where m is the total number of data points and n the dimension of our linear model; and $y \in \mathbb{R}^n$ observations as a column vector. The problem consists of minimizing

$$\min_{w\in\mathbb{R}^n} \quad \|y - Xw\|^2. \tag{7}$$

Problem (7) corresponds to a network in which each row of X is a training example. Using dropout every component of the training examples is kept with probability p, and the input can be transformed into $D \odot X$, where '$\odot$' corresponds to the Hadamard or element-wise product; D is a random matrix formed of Bernoulli variables with probability p. We obtain

$$\min_{w} \quad \mathbb{E}_{D\sim\mathrm{Bernoulli}(p)}\big\{\|y - D \odot Xw\|^2\big\} \tag{8}$$

Operating over the previous expression (see [2]) we get

$$\min_{w} \quad \|y - pXw\|^2 + p(1-p)\|\Gamma w\|^2 \tag{9}$$

with $\Gamma = (\operatorname{diag}(X^TX))^{-1/2}$. This last expression corresponds to a ridge regression with a particular form of Γ that depends on the data X. If some component i of Γ is big, then the objective penalizes such component in w more than others.

If we make the change of variable $\tilde{w} = pw$, we get

$$\min_{\tilde{w}} \quad \|y - X\tilde{w}\|^2 + \frac{1-p}{p}\|\Gamma\tilde{w}\|^2, \tag{10}$$

where the penalization term is explicit with $\lambda = \frac{1-p}{p}$. If we decrease p to have a greater dropout rate, we are increasing the regularization effect. This result explicitly relates dropout to a particular form of ridge regression.

1.3 Dropout in logistic regression

Consider a single logistic unit with n inputs such that $\sigma(z) = 1/(1 + e^{-z})$ and $z = x^T w$. Variable w corresponds to the network weights. The total number of possible network configurations connecting or disconnecting inputs is $m = 2^n$, so we can get $O_1, \ldots, O_m$ possible outcomes depending on these connections. Each sampled subnetwork has an associated probability $P_1, \ldots, P_m$, which corresponds to a specific output.

We define the following values: the mean over possible outcomes $E = \sum_i P_i O_i$; the weighted geometric mean $G = \prod_i O_i^{P_i}$; and the weighted geometric mean of the complements $G' = \prod_i (1-O_i)^{P_i}$. Finally we also define the normalized weighted geometric mean $\mathrm{NWGM} = G/(G + G')$. With simple operations, we can show that

$$\mathrm{NWGM} = \frac{1}{1 + e^{-\sum_j p w_j x_j}} = \sigma(pz). \tag{11}$$

In the previous result p corresponds to the probability of keeping the input component when using dropout.

The implication is that using dropout over a single logistic unit corresponds to a weighted geometric average over all possible sampled subnetworks, which already has a regularization effect.

1.3.1 Dynamics of a single logistic unit

The result from a single linear unit generalizes to a sigmoidal unit as well. In this case the expected gradient of the dropout network approximates

$$\mathbb{E}\left\{\frac{\partial E^{\mathrm{d}}}{\partial w_i}\right\} \approx \frac{\partial E^{\mathrm{ens}}}{\partial w_i} + \lambda \sigma'(pz) x_i^2 \operatorname{var}(p) w_i. \tag{12}$$

Thus, the expectation of the dropout gradient corresponds approximately to the gradient of the ensemble network plus a ridge regularization term with proper adaptive coffecients. A similar analysis can be extended to deeper networks, as well as using ReLu units [4].

Overall, the take-out message is that the expected dropout gradient corresponds to an approximated ensemble network, regularized by an adaptive weight decay with a propensity for self-consistent variance minimization. Finally, the convergence properties of the dropout method can be understood via the stochastic gradient descent.

2 Batch normalization

Batch normalization is a novel method of adaptive reparametrization, motivated by the difficulty of training very deep models [5]. When we use gradient descent, parameters from all layers are updated at the same time. Then, the composition of many functions can have unexpected results because all of these functions have been changed simultaneously. This makes learning with gradient descent challenging, because the learning rate can easily explode the gradients or not affect them at all.

The previous idea is justified as follows. Consider a linear network with a single neuron per layer and single input $x \in \mathbb{R}$. The output has the following form $\hat{y} = w^{[1]}w^{[2]} \dots w^{[L]}x$, which is nonlinear in the w parameters. If we update $w \leftarrow w - \epsilon g$, where $g = \nabla_w \hat{y}$, we get the following updated output:

$$\hat{y} \leftarrow (w^{[1]} - \epsilon g^{[1]})(w^{[2]} - \epsilon g^{[2]}) \dots (w^{[L]} - \epsilon g^{[L]})x. \tag{13}$$

The previous expression contains many high order components (up to order L), that can influence greatly the value of $\hat{y}$. This makes the learning rate very hard to adjust. Batch normalization provides a novel reparametrization that significantly reduces the problem of coordinating updates across many layers.

The method can be applied to any hidden layer, and is inspired by the normalization step normally applied to an input. Consider that the input to the network is a minibatch $X^{\{i\}} = (x^{\{i\}(1)}, \dots, x^{\{i\}(m)})$, where m refers to the size of the minibatch, $\{i\}$ to the minibatch index, and $Y^{\{i\}} = (y^{\{i\}(1)}, \dots, y^{\{i\}(m)})$ indicates labels. The input can be normalized simply with

$$\widetilde{X}^{\{i\}} = \frac{X^{\{i\}} - \mu}{\sigma + \epsilon} \tag{14}$$

where $\epsilon = 10^{-8}$ is frequently used,

$$\mu = \frac{1}{m}\sum_r x^{\{i\}(r)}, \text{ and } \sigma^2 = \frac{1}{m}(x^{\{i\}(r)} - \mu)^2. \tag{15}$$

This technique normalizes the input to have zero mean and unit variance.

Batch normalization extends this concept to other hidden layers. Assume that $Z^{\{i\}[l]} = W^{[l]}A^{\{i\}[l-1]} + b^{[l]}$ is the linear combination performed at layer l before the activation function, such as ReLu, or tanh. Weights $W^{[l]}$ and $b^{[l]}$ correspond to the linear coefficients and offset, respectively, and $A^{\{i\}[l-1]}$ the output of the previous layer. One way to implement batch normalization is to normalize the linear outcomes:

$$Z_{\text{norm}}^{\{i\}[l]} = \frac{Z^{\{i\}[l]} - \mu^{\{i\}[l]}}{\sigma^{\{i\}[l]} + \epsilon} \tag{16}$$

where

$$\mu^{\{i\}[l]} = \frac{1}{m}\sum_r z^{\{i\}[l](r)}, \text{ and } (\sigma^{\{i\}[l]})^2 = \frac{1}{m}\sum_r (z^{\{i\}[l](r)} - \mu^{\{i\}[l]})^2. \tag{17}$$

We remark that the normalization parameters depend on the minibatch and change with different examples. Another option is to normalize $A^{\{i\}[l-1]}$ before the linear transformation, but [5] recommends the former option.

Once that the linear outputs are normalized, the outcome is rescaled with new parameters $\gamma^{\{i\}[l]}$ and $\beta^{\{i\}[l]}$ that need to be learned in the training process:

$$\widetilde{Z}^{\{i\}[l]} = \gamma^{\{i\}[l]} Z_{\text{norm}}^{\{i\}[l]} + \beta^{\{i\}[l]}. \tag{18}$$

Although this reparametrization may seem repetitive, it actually helps to decompose the learning process of the weights between layers. The scheme has the same expressive capabilities as before (simply by setting $\beta^{\{i\}[l]} = \mu^{\{i\}[l]}$ and $\gamma^{\{i\}[l]} = \sigma^{\{i\}[l]}$), but the new parametrization has better learning dynamics.

These dynamics are justified because the weights from one layer do not affect the statistics (first and second order) of the next layer. The mean and deviation are always normalized, and an increase of the values in the previous layer does not produce an immediate increase in the next layer. The values of every layer are updated independently because of this normalization step.

Another aspect is that the normalization of $Z_{\text{norm}}^{\{i\}[l]}$ make the offsets $b^{[l]}$ obsolete, so they should be removed from the optimization process. Note also that weights $W^{[l]}$ are shared for different minibatches, but $\gamma^{\{i\}[l]}$ and $\beta^{\{i\}[l]}$ depend greatly on the statistics of the minibatch and the number of elements. For this reason, at testing time, a weighted average of these terms is computed during training for a robust evaluation.

This weighted moving average consists of the following update:

$$\gamma_t = \alpha\gamma_t + (1-\alpha)\gamma^{\{i\}[l]}, \tag{19}$$

where $\gamma^{\{i\}[l]}$ is the outcome of a minibatch training process, and γ_t is updated every time a minibatch is processed and used for testing time. A similar procedure is employed to track β_t. All frameworks that implement batch normalization incorporate a way to track these values for testing time.

The update rate α is set such that low values (~ 0.5) establish a very non-smoothed moving average, whereas high values (~ 0.99) result into a very smooth outcome and incorporate many values into the averaged estimate. The estimates in the first part of the averaging process are biased, so early estimates would need to be corrected if used for testing purposes: $\frac{\gamma_t}{1-\alpha^t}$. In general, this step can be overlooked if the number of epochs and minibatches become high enough.

3 Gradient checking

Gradient checking is a useful technique to debug code of manual implementations of neural networks. I briefly want to discuss this technique because it is easy to understand and use if needed. It is not intended for training of networks because it is slow, but it can help to identify errors in a backpropagation implementation.

We use the definition of the derivative of a function to motivate its approximation:

$$f'(x) = \lim_{\epsilon \to 0} \frac{f(x+\epsilon) - f(x-\epsilon)}{2\epsilon} \approx \frac{f(x+\epsilon) - f(x-\epsilon)}{2\epsilon}. \tag{20}$$

The previous expression approximates the derivative for small ϵ. The approximation error is of the order $O(\epsilon^2)$. In the multivariate case, the ϵ term affects a single component. We define $\theta_r^+ = (\theta_1, \ldots, \theta_r + \epsilon, \ldots, \theta_n)$ and $\theta_r^- = (\theta_1, \ldots, \theta_r - \epsilon, \ldots, \theta_n)$ If $f(\cdot)$ represents the loss function of the neural network, we have for a single the following partial derivative:

$$df(\theta_r) \approx \frac{f(\theta_r^+) - f(\theta_r^-)}{2\epsilon} \tag{21}$$

The whole procedure of gradient checking is described in Algorithm 1.

Algorithm 1 Gradient checking.

```
1: Reshape input vector in a column vector $\theta$.
2: for each $r$ component do
3:     $\theta_{\text{old}} \leftarrow \theta_r$
4:     Calculate $f(\theta_r^+)$ and $f(\theta_r^-)$.
5:     Compute $d\theta_r^{\text{approx}}$.
6:     Restore $\theta_r \leftarrow \theta_{\text{old}}$
7: end for
8: Verify relative error is below some threshold:
```

$$\xi = \frac{\|d\theta^{\text{approx}} - d\theta\|}{\|d\theta^{\text{approx}}\| + \|d\theta\|} \tag{22}$$

References

[1] Geoffrey E Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R Salakhutdinov, "Improving neural networks by preventing co-adaptation of feature detectors," 2012, arXiv:1207.0580.

[2] Nitish Srivastava, Geoffrey E Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting.," *Journal of machine learning research*, vol. 15, no. 1, pp. 1929–1958, 2014, `http://jmlr.org/papers/volume15/srivastava14a/srivastava14a.pdf`.

[3] Pierre Baldi and Peter J Sadowski, "Understanding dropout," in *Advances in Neural Information Processing Systems*, 2013, pp. 2814–2822, `http://papers.nips.cc/paper/4878-understanding-dropout.pdf`.

[4] Pierre Baldi and Peter Sadowski, "The dropout learning algorithm," *Artificial Intelligence*, vol. 210, pp. 78 – 122, 2014, doi:10.1016/j.artint.2014.02.004.

[5] Sergey Ioffe and Christian Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in

Proceedings of the 32nd International Conference on Machine Learning, Francis Bach and David Blei, Eds., Lille, France, 07–09 Jul 2015, vol. 37 of *Proceedings of Machine Learning Research*, pp. 448–456, PMLR, `http://proceedings.mlr.press/v37/ioffe15.html`.