Optimization algorithms, understanding dropout and batch normalization

Data Science 2 AC 209b

Javier Zazo Pavlos Protopapas

February 15, 2019

Abstract

In this section, we will explain the different methods to compute moving averages according to the stationarity of the process, and relate these results to the optimization methods used in deep learning. We will also discuss several techniques that have been proposed in recent years to help tune the learning rate in a more systematic way. Additionally, we will motivate why dropout is regarded as a regularization technique and explain the underlying effects of its usage. We will consider some simple neural networks where a better understanding of dropout is available and infer this interpretation to other deep networks. Finally, we will describe "batch normalization" to enhance training speed in deep networks, and very briefly, we will present gradient checking as a technique to validate backpropagation of gradients.

1 Moving averages

Given a stationary process $x[n]$ and a sequence of observations x_1, x_2, x_3, …, x_n, …, we want to estimate the average of all observed values dynamically, i.e., obtain a new estimate whenever we receive a new observation. Simply, we could use a standard average after receiving n observations:

$$\overline{x}_{n+1} = \frac{1}{n}\left(x_1 + x_2 + \ldots + x_n\right). \tag{1}$$

However, this requires storing all the history values of the process, and also perform n sums on every instant. One can easily guess there is a better way of doing this:

$$\begin{aligned}\overline{x}_{n+1} &= \frac{1}{n}\sum_{i=1}^{n} x_i = \frac{1}{n}\left(x_n + \sum_{i=1}^{n-1} x_i\right) = \frac{1}{n}\left(x_n + (n-1)\frac{1}{n-1}\sum_{i=1}^{n-1} x_i\right) \\ &= \frac{1}{n}\left(x_n + (n-1)\overline{x}_n\right) = \overline{x}_n + \frac{1}{n}\left(x_n - \overline{x}_n\right). \end{aligned} \tag{2}$$

Essentially, we update the old estimate employing a dynamic step size $\alpha_n = 1/n$, and computation of a residual of the new observation and the old estimate:

$$\overline{x}_{n+1} = \overline{x}_n + \alpha_n\left(x_n - \overline{x}_n\right) \tag{3}$$

This procedure is called *simple moving average* (SMI).

However, because of the previous interpretation of step size and residual computation, we could easily generalize the previous scheme to other step sizes. In fact, from **stochastic approximation theory**, an estimator of the form (3) converges to the true mean value with probability 1, if the following conditions are met:

$$\sum_{i=1}^{\infty} \alpha_i = \infty \qquad \text{and} \qquad \sum_{i=1}^{\infty} \alpha_i^2 < \infty. \tag{4}$$

A step size $\alpha_n = 1/n$ satisfies these requirements.

How about a constant step size $\alpha_n = \alpha \in (0, 1]$, as you have probably seen in so many other procedures? Well, a constant step size does not fulfill the second requirement in (4), and therefore, does not converge to the true mean value. Nonetheless, it is specially useful for non-stationary processes, because if $0 < \alpha < 1$, it will give past observations less influence than current estimates, and may track more adequately the non-stationarity of the process. Note that $\alpha = 1$ simply forgets every past observed value, and $\alpha = 0$ discards any new observation.

When using a constant step size in (3), we get the following expression:

$$\begin{aligned}
\overline{x}_{n+1} &= \overline{x}_n + \alpha\left(x_n - \overline{x}_n\right) \\
&= \alpha x_n + (1-\alpha)\overline{x}_n \\
&= \alpha x_n + (1-\alpha)[\alpha x_{n-1} + (1-\alpha)\overline{x}_{n-1}] \\
&= \alpha x_n + (1-\alpha)\alpha x_{n-1} + (1-\alpha)^2\overline{x}_{n-1}] \\
&= \alpha x_n + (1-\alpha)\alpha x_{n-1} + (1-\alpha)^2\alpha x_{n-2} + \ldots + (1-\alpha)^{n-1}\alpha x_1 + (1-\alpha)^n\overline{x}_1 \\
&= \boxed{(1-\alpha)^n\overline{x}_1 + \sum_{i=1}^{n} \alpha(1-\alpha)^{n-i} x_i}\,.
\end{aligned} \tag{5}$$

Observations are weighted with $\alpha(1-\alpha)^{n-i}$. For small positive values of α, past observations are heavily penalized with $(1-\alpha)^{n-i}$: the further in the past, the less influence in the current estimate. For this reason, estimating the mean of process using a constant step size is referred as *exponentially weighted moving average* (EWMA).

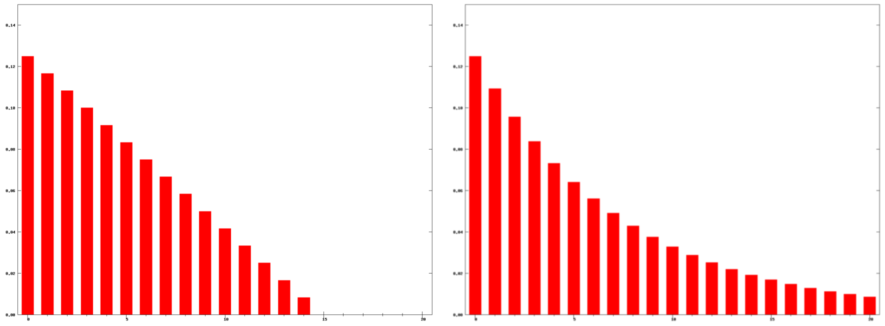
Note also in (5) how the initialization value $\overline{x}_1$ is forgotten as n grows large, but can have an important influence in early estimates. We will see later that Adam method incorporates a bias correction update to prevent estimates from being too far away from the early observed values. Figure 1 shows the dynamic weights for the SMI and EWMA procedures.

One way to visualize that the EWMA is indeed a weighted average, is by noticing that $(1-\alpha)^n + \sum_{i=1}^{n} \alpha(1-\alpha)^{n-i} = 1$. This means that every coefficient is weighted by some positive value, and all of them sum to 1. Alternatively, assume that we have infinite observations. First $(1-\alpha)^n\overline{x}_1 \to 0$, and since $1/\alpha = 1 + (1-\alpha) + (1-\alpha)^2 + (1-\alpha)^3 + \ldots$, we can rewrite (5) as

$$\lim_{n\to\infty} \overline{x}_n = \lim_{n\to\infty} \frac{x_n + (1-\alpha)x_{n-1} + (1-\alpha)^2 x_{n-2} + (1-\alpha)^3 x_{n-3} + \ldots}{1 + (1-\alpha) + (1-\alpha)^2 + (1-\alpha)^3 + \ldots}. \tag{6}$$

Previous expression resembles a weighted average (of convergent sums).

Figure 1: Effective weight values to estimate the mean of a given process.



As we will mention later when we present the optimization methods that exploit momentum, expression (3) is presented equivalently

$$\boxed{\overline{x}_{n+1} = \beta \overline{x}_n + (1-\beta) x_n}, \tag{7}$$

where $\beta = 1 - \alpha$. A high value of β gives a lot of weight to current estimate of the mean with respect to new observations. As an extreme case, $\beta = 1$ does not update the mean estimate, and $\beta = 0$ only considers the new observation. Essentially, β controls the amount of effective observations that are taken into account when computing the mean value. A rule of thumb discussed in some forums, is that $N = \frac{1+\beta}{1-\beta}$ observations accounts for about 86% of the total weights in a finite list of observations. This can give us the following intuition:

- $\beta = 0.9$ corresponds to around 20 observations.
- $\beta = 0.98$ gives $N \approx 100$ points, and corresponds to a wide window.
- $\beta = 0.5$ gives $N \approx 3$ points, and will be susceptible to outliers.

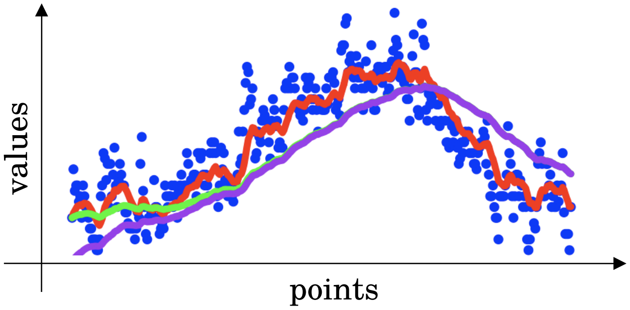
Consider Figure 2. Here the red line has a reasonable ability to track immediate changes of the process, likely $\beta \approx 0.9$. Alternatively, the violet line, is much more smooth and shows lag in tracking the average terms. Therefore, β has a higher value than with the red line case. Also it is worth noting that the violet curve presents a strong bias towards zero at the early stages. Green curve corrects this bias:

$$\overline{x}_n^{\text{corrected}} = \frac{\overline{x}_n}{1 - \beta^t}. \tag{8}$$

Equation (8) was proposed in the Adam paper [1], where the origin of the bias comes from the zero initialization:

$$\overline{x}_{n+1} = \beta^n \underbrace{\overline{x}_1}_{0} + (1-\beta) \sum_{i=1}^{n} \beta^{n-i} x_i \tag{9}$$

Figure 2: Exponential moving average example.



The correction is made by taken expectation of both sides of the EWMA update:

$$
\begin{aligned}
\mathbb{E}[\overline{x}_{n+1}] &= \mathbb{E}\left[(1-\beta)\sum_{i=1}^{n}\beta^{n-i}x_i\right] \\
&= \mathbb{E}[x_n](1-\beta)\sum_{i=1}^{n}\beta^{n-i}+\zeta \\
&= \mathbb{E}[x_n](1-\beta^n)+\zeta.
\end{aligned}
$$

If the process is stationary, $\zeta = 0$, or approximates zero if β is correctly tracking the unstationary process. Note however, that in practice bias correction is not entirely necessary, and many deep network frameworks like Keras or Caffe do not include it.

2 Optimization algorithms

The core understanding to the differences between optimization descent methods is how they make use of momentum techniques. We are not going to give details about the usage of stochastic gradient descent (SGD) with mini-batches or individual examples, as this has been explained in the main lecture. The goal in this section is to understand the differences between the stochastic optimization algorithms.

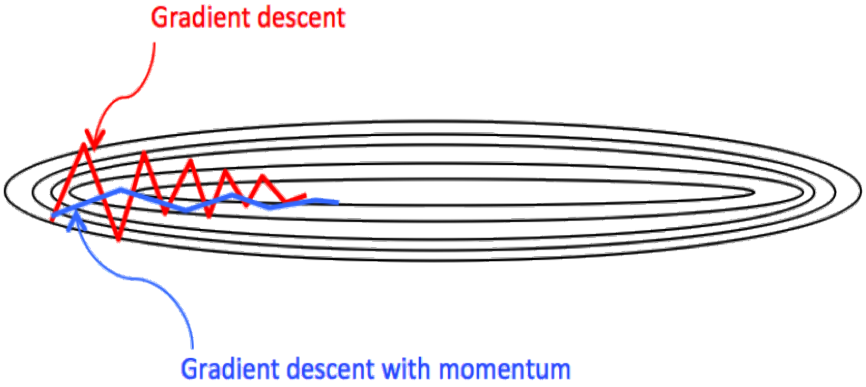
Stochastic gradient descent with momentum performs an exponential moving average over the components of the observed gradients. The aim is to reduce the variance of using noisy mini-batches with respect to the true gradient, and estimate a common direction of descent, as shown in Figure 3. Algorithm 1 provides the necessary steps for reference.

Algorithm 1 Stochastic gradient descent with momentum.

```
On iteration t for $W$ update:
    Compute $dW$ on current mini-batch.
    $v_{dW} = \beta v_{dW} + (1-\beta)dW$.
    $W = W - \alpha v_{dW}$.
```

RMSProp is presented in Algorithm 2. The idea is that high variance gradients will have larger values and the squared averages will be large. In these cases, tracking the second

Figure 3: Stochastic gradient descent with momentum



Algorithm 2 RMSProp.

```
On iteration t for $W$ update:
    Compute $dW$ on current mini-batch.
    $s_{dW} = \beta_2 s_{dW} + (1 - \beta_2) dW^2$.
    $W = W - \alpha \frac{dW}{\sqrt{s_{dW}} + \epsilon}$.
```

order gradient value allows to reduce the step size dynamically, and effectively use a larger step size than with SGD. Note that $\epsilon = 10^{-8}$ is necessary to control numerical stability.

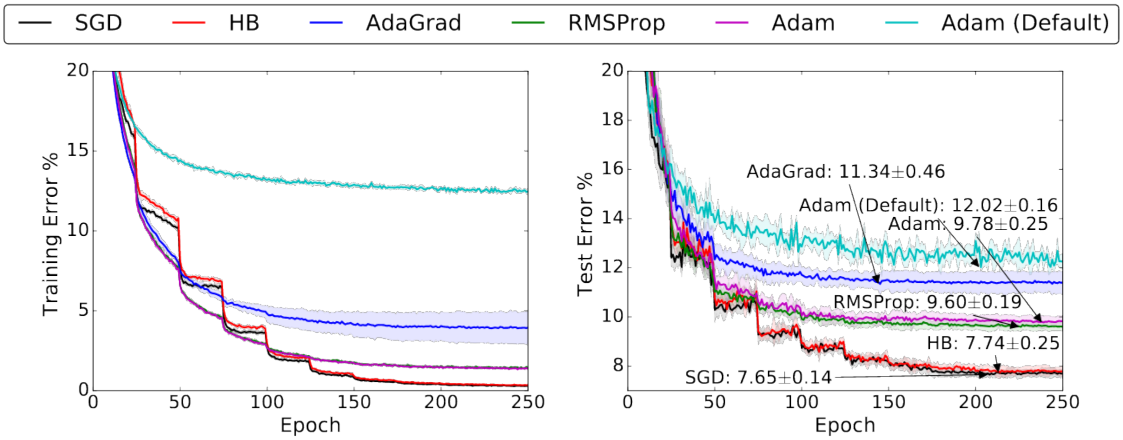
Adam is a combination of SGD with momentum and RMSprop, where the algorithm keeps track of the averaged gradient values, as well as the second order values (componentwise), see Algorithm 3. Additionally, it incorporates bias correction as we discussed in Section 1, but this is not completely necessary, and usually omitted in practice. The intention is that the algorithm searches for a direction of common descent from the noise introduced with mini-batches, as well as automatically adjust step size. Although it seems very complete, it also comes with problems in practice, which we will discuss next.

Algorithm 3 Adam.

```
On iteration t for $W$ update:
    Compute $dW$ on current mini-batch.
    $v_{dW} = \beta_1 v_{dW} + (1 - \beta_1) dW$.
    $s_{dW} = \beta_2 s_{dW} + (1 - \beta_2) dW^2$.
    $v^{\text{corrected}} = \frac{v_{dW}}{1 - \beta_1^t}$
    $s^{\text{corrected}} = \frac{s_{dW}}{1 - \beta_2^t}$
    $W = W - \alpha \frac{v^{\text{corrected}}}{\sqrt{s_{dW}} + \epsilon}$.
```

AMSGrad is a modification of Adam algorithm, where the authors of [2] noticed that Adam may fail to converge even in convex problems. Although their analysis is fabricated to some convoluted example, the authors claim that undesired effects occur in practice when using Adam. The intuition comes from the observation that some good and large mini-

Figure 4: The marginal value of adaptive gradient methods



(a) CIFAR-10 (Train)

(b) CIFAR-10 (Test)

batch gradients can be obfuscated by a large estimation of the second gradient moment, and not in situations with bad gradients, hindering convergence speed and quality. As a consequence, the authors propose to only use a monotonically increasing sequence of second order of gradients. The procedure is given in Algorithm 4. Note that this critic applies to RMSProp as well.

Algorithm 4 AMSGrad.

```
On iteration t for $W$ update:
    Compute $dW$ on current mini-batch.
    $v^{n+1}_{dW} = \beta_1 v^n_{dW} + (1-\beta_1) dW$.
    $s^{n+1}_{dW} = \beta_2 s^n_{dW} + (1-\beta_2) dW^2$.
    $\hat{s}^{n+1}_{dW} = \max(\hat{s}^n_{dW}, s^{n+1}_{dW})$
    $W = W - \alpha \frac{v^{\text{corrected}}}{\sqrt{\hat{s}^{n+1}_{dW}+\epsilon}}$.
```

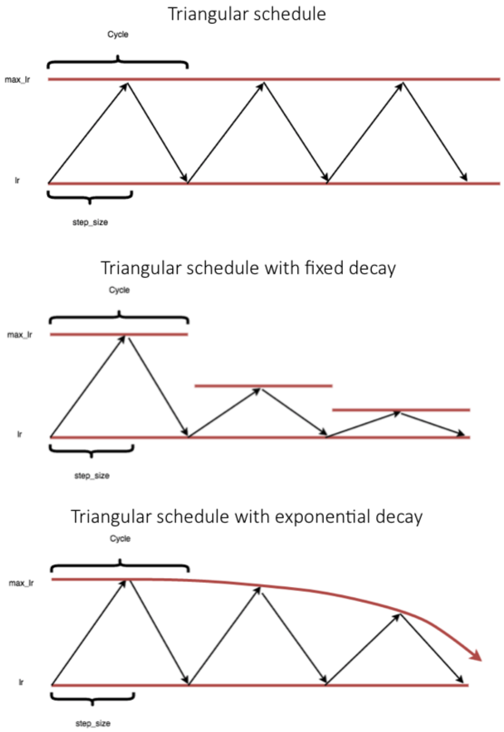
Finally, there has been novel analysis of adaptive optimization methods presented by [3], where the authors show that the best performing algorithms in terms of trainining and test errors are achieved by simple SGD and SGD with momentum. We refer to the original publication to learn about the different databases in which a comparison was tested, but show performances in Figure 4 for CIFAR-10.

3 Tuning the learning rate

Tuning the learning rate (LR) to train a neural network is a tedious task, and it can affect greatly the training/test errors, and convergence speed. Luckily, in recent years some successful ideas have been proposed that help analyze this problem in a reasonably systematic way.

In [4], the author proposes to use a cyclical update of the learning rate to improve the optimization process. The idea is that large learning rates speed up the convergence rate

Figure 5: Cyclical learning rates for neural networks.



when the estimate is far away from a local minima, but smaller step sizes help achieve better precision when close to a local extreme point. However, intermediate estimates can still converge easily to saddle points where the gradient is small, or even to narrow local optima (that do not generalize well), and a sudden increase in the learnign rate may escape this basins of atraction and boost performance. The sudden increase of the LR magnitude may have a short term negative effect, but still present a longer term beneficial effect. Puting this into practice, make the LR magnitude cyclical with periodic number of epochs.

Additionally, exponentially decreasing the magnitude of the LR can still be considered in such a scheme. These ideas are pictured in Figure 5 with intuitive examples.

However, the question of how to find a good initial estimate, or a range of good LR values, still persists. The same reference [4] proposes to do this learning with a single pass/epoch training over all mini-batches. We start with a small LR, and increase its magnitude exponentially with every new mini-batch, while computing the global loss on the validation set. The loss curve after the whole epoch should start high, decrease with a more pronounced slope, and then start to increase again after the LR becomes too high. Because this graph is not averaged, depends on individual mini-batches and not epochs, it will be noisy, but may still be able to convey some information about good LR magnitudes. Not only individual LR values, but also ranges of operation for the cyclic update scheme. Essentially, a sensible recommendation is not to use the global minimum of the curve, but some value more conservative that can still perform well in subsequent iterations. Figure 6 exemplifies this idea.

Figure 6: Estimating the learning rate.

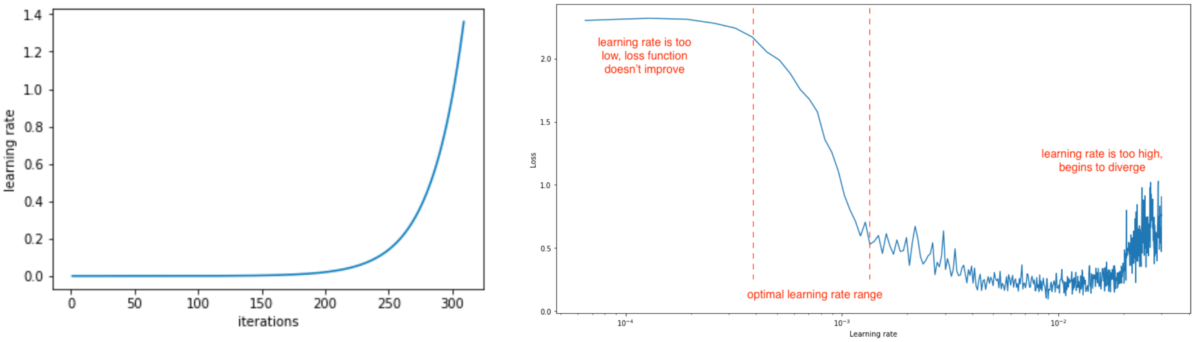
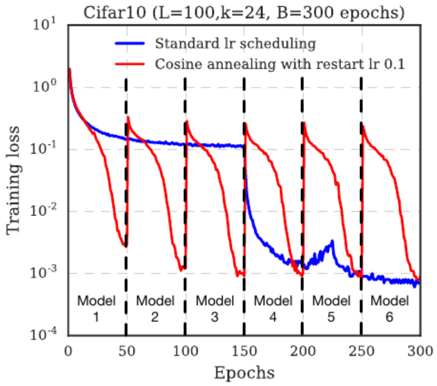


Figure 7: SGD with warm restarts



A very similar idea to the use of cyclic LR values is also proposed in [5], with the goal of actually escaping local minima, in the search of a better one. This entails to restart the LR every T_i epochs, and record the best estimates before every restart. The restarts are not from scratch, but from the last converged values before the LR is increased. The authors propose a cosine update rule with maximum and minimum LR values:

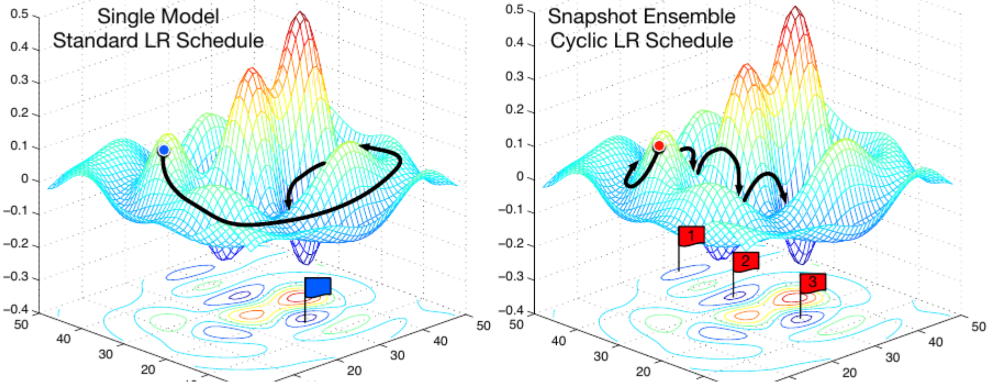
$$\alpha_t = \alpha^i_{\text{min}} + \frac{1}{2}(\alpha^i_{\text{max}} - \alpha^i_{\text{min}})(1 + \cos(\frac{T_c ur}{T_i}\pi)). \tag{10}$$

The authors also note that the cycle can be lengthened in time, in order to gain better estimates as time increases, as well as exponentially decreasing the minimum and maximum values of the cosine update rule. Figure 7 shows such a scheme.

Finally, [6] extends the SGDR idea to actually compute a series of trained neural networks, and use all of them in an ensemble. Ensemble networks are more robust and accurate than individual networks, as they may take a majority or average vote on the outputs of their particular task. Furthermore, they can also be regarded as a regularization technique because if every individual network overfits, their ensemble will probably generalize better (if the overfitting estimates are independent).

Figure 8 shows how a standard minimization procedure converges to a single solution, but using SGDR technique, several local solutions can be obtained with similar cost. Then, every estimated network is recorded and used for the task of interest. The authors compare this technique of snapshot ensembles, i) with standard ensembles using independent starting

Figure 8: Snapshot ensembles: train 1, get M for free.



points, and ii) with no cycled trained networks. All of these comparisons after the same number of running epochs. We refer you to their original publication to check on their results for different databases and network architectures, but this technique consistently outperforms other configurations.

4 Dropout

Dropout is a regularization technique for deep neural networks, first proposed in [7, 8]. 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 9.

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 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 10.

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 [8] 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

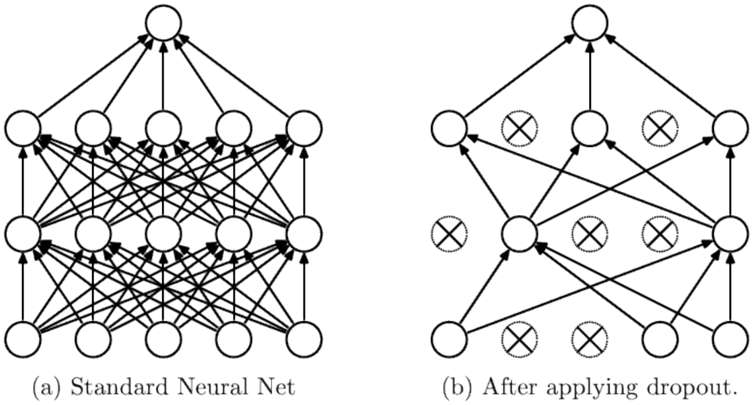


Figure 9: Dropout example, from [7].

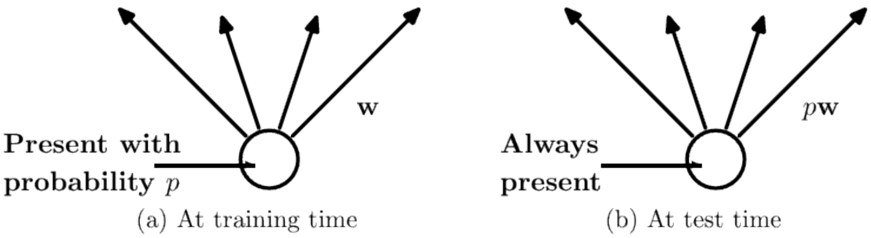


Figure 10: Dropout unit at training and testing times, from [7].

particular, for layer l,

$$\begin{aligned} z^{[l]} &= \frac{1}{p_l} W^{[l]} D^{[l]} a^{[l-1]} + b^{[l]} \\ a^{[l]} &= g(z^{[l]}), \end{aligned} \tag{11}$$

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.

4.1 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

$$z^{[l]} = W^{[l]} D^{[l]} z^{[l-1]}, \tag{12}$$

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

$$\mathbb{E}\{z^{[l]}\} = p_l W^{[l]} z^{[l-1]}, \tag{13}$$

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 $p_l W^{[l]}$.

4.1.1 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

$$E^{\text{ens}} = (y^{(i)} - p_l W^{[l]} x^{(i)})^2 \tag{14}$$

$$E^{\text{d}} = (y^{(i)} - W^{[l]} D^{[l]} x^{(i)})^2. \tag{15}$$

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 (14), and the expected gradient for (15), it can be shown as in [9] that

$$\mathbb{E}\{E^{\text{d}}\} = E^{\text{ens}} + \sum_{r=1}^{n_1} \frac{1}{2} \operatorname{var}(D^{[l]})(x_r^{(i)})^2 w_r^2 \tag{16}$$

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

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.

4.2 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

$$\min_{w \in \mathbb{R}^n} \quad \|y - Xw\|^2. \tag{17}$$

Problem (17) 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

$$\min_{w} \quad \mathbb{E}_{D \sim \text{Bernoulli}(p)} \left\{ \|y - D \odot Xw\|^2 \right\} \tag{18}$$

Operating over the previous expression (see [8]) we get

$$\min_{w} \quad \|y - pXw\|^2 + p(1-p)\|\Gamma w\|^2 \tag{19}$$

with $\Gamma = (\operatorname{diag}(X^T X))^{-1/2}$. 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

$$\min_{\tilde{w}} \quad \|y - X\tilde{w}\|^2 + \frac{1-p}{p}\|\Gamma\tilde{w}\|^2, \tag{20}$$

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.

4.3 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

$$\mathrm{NWGM} = \frac{1}{1+e^{-\sum_j p w_j x_j}} = \sigma(pz). \tag{21}$$

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 sub-networks, which already has a regularization effect.

4.3.1 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

$$\mathbb{E}\left\{\frac{\partial E^{\mathrm{d}}}{\partial w_i}\right\} \approx \frac{\partial E^{\mathrm{ens}}}{\partial w_i} + \lambda\sigma'(pz)x_i^2 \operatorname{var}(p) w_i. \tag{22}$$

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.

4.4 Dropout in Deep Neural Networks

A similar analysis can be extended to deeper networks, as well as using ReLu units [10]. Consider a network of sigmoidal units. The output of unit i in layer l:

$$a_i^{[l]} = \sigma\Big(\sum_j W_{ij}^{[l]} a^{[l-1]}\Big).$$

The normalized weighted geometric mean over all possible network configurations is

$$\text{NWGM}(a_i^{[l]}) = \frac{\Pi_N (a_i^{[l]})^{P(N)}}{\Pi_N (1 - a_i^{[l]})^{P(N)} + \Pi_N (a_i^{[l]})^{P(N)}},$$

and the averaging properties of dropout take the following form:

$$\mathbb{E}\{a_i^{[l]}\} = \sigma\Big(\mathbb{E}\Big\{\sum_j W_{ij}^{[l]} a_i^{[l-1]}\Big\}\Big).$$

This result states that the output of a logistic unit within a NN of the ensemble network corresponds to that of the normalized geometric mean. Furthermore, the dynamics of deep neural network can be approximated in a similar way as performed for a single logistic unit.

Overall, the take-home 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.

To conclude, the convergence properties of the dropout method can be understood via the stochastic gradient descent.

5 Batch normalization

Batch normalization is a novel method of adaptive reparametrization, motivated by the difficulty of training very deep models [11]. 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:

$$\hat{y} \leftarrow (w^{[1]} - \epsilon g^{[1]})(w^{[2]} - \epsilon g^{[2]}) \dots (w^{[L]} - \epsilon g^{[L]}) x. \tag{23}$$

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

$$\widetilde{X}^{\{i\}} = \frac{X^{\{i\}} - \mu}{\sigma + \epsilon} \tag{24}$$

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

$$\mu = \frac{1}{m}\sum_r x^{\{i\}(r)}, \text{ and } \sigma^2 = \frac{1}{m}(x^{\{i\}(r)} - \mu)^2. \tag{25}$$

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:

$$Z^{\{i\}[l]}_{\text{norm}} = \frac{Z^{\{i\}[l]} - \mu^{\{i\}[l]}}{\sigma^{\{i\}[l]} + \epsilon} \tag{26}$$

where

$$\mu^{\{i\}[l]} = \frac{1}{m}\sum_r z^{\{i\}[l](r)}, \text{ and } (\sigma^{\{i\}[l]})^2 = \frac{1}{m}\sum_r (z^{\{i\}[l](r)} - \mu^{\{i\}[l]})^2. \tag{27}$$

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 [11] 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:

$$\widetilde{Z}^{\{i\}[l]} = \gamma^{\{i\}[l]} Z^{\{i\}[l]}_{\text{norm}} + \beta^{\{i\}[l]}. \tag{28}$$

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 $Z^{\{i\}[l]}_{\text{norm}}$ 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:

$$\gamma_t = \alpha\gamma_t + (1-\alpha)\gamma^{\{i\}[l]}, \tag{29}$$

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.

Recap: Batch normalization is performed as follows: we start with the ouput at a certain layer $l-1$ for a given batch; we perform the forward pass through the layer weights $W^{[l]}$; we compute the mini-batch's mean and deviation (27), and normalize as in (26); we forward pass through the scaling with (29); finally, we apply the non-linear activation function (ReLu, etc.).

6 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:

$$f'(x) = \lim_{\epsilon \to 0} \frac{f(x+\epsilon) - f(x-\epsilon)}{2\epsilon} \approx \frac{f(x+\epsilon) - f(x-\epsilon)}{2\epsilon}. \tag{30}$$

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 $\theta_r^+ = (\theta_1, \dots, \theta_r + \epsilon, \dots, \theta_n)$ and $\theta_r^- = (\theta_1, \dots, \theta_r - \epsilon, \dots, \theta_n)$ If $f(\cdot)$ represents the loss function of the neural network, we have for a single the following partial derivative:

$$df(\theta_r) \approx \frac{f(\theta_r^+) - f(\theta_r^-)}{2\epsilon} \tag{31}$$

The whole procedure of gradient checking is described in Algorithm 5.

References

[1] Diederik P Kingma and Jimmy Ba, "Adam: A method for stochastic optimization," 2014, arXiv:1412.6980.

[2] Sashank J. Reddi, Satyen Kale, and Sanjiv Kumar, "On the convergence of adam and beyond," in *International Conference on Learning Representations*, 2018, `https://openreview.net/forum?id=ryQu7f-RZ`.

[3] Ashia C Wilson, Rebecca Roelofs, Mitchell Stern, Nati Srebro, and Benjamin Recht, "The marginal value of adaptive gradient methods in machine learning," in *Advances in Neural Information Processing Systems*, 2017, pp. 4148–4158.

Algorithm 5 Gradient checking.

```
Reshape input vector in a column vector $\theta$.
for each $r$ component do
    $\theta_{\text{old}} \leftarrow \theta_r$
    Calculate $f(\theta_r^+)$ and $f(\theta_r^-)$.
    Compute $d\theta_r^{\text{approx}}$.
    Restore $\theta_r \leftarrow \theta_{\text{old}}$
end for
Verify relative error is below some threshold:
    $\xi = \frac{\|d\theta^{\text{approx}} - d\theta\|}{\|d\theta^{\text{approx}}\| + \|d\theta\|}$    (32)
```

[4] Leslie N Smith, "Cyclical learning rates for training neural networks," 2017, arXiv:1506.01186.

[5] Ilya Loshchilov and Frank Hutter, "SGDR: Stochastic gradient descent with warm restarts," 2016, arXiv:1608.03983.

[6] Gao Huang, Yixuan Li, Geoff Pleiss, Zhuang Liu, John E Hopcroft, and Kilian Q Weinberger, "Snapshot ensembles: Train 1, get m for free," 2017, arXiv:1704.00109.

[7] 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.

[8] 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`.

[9] 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`.

[10] 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.

[11] 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`.