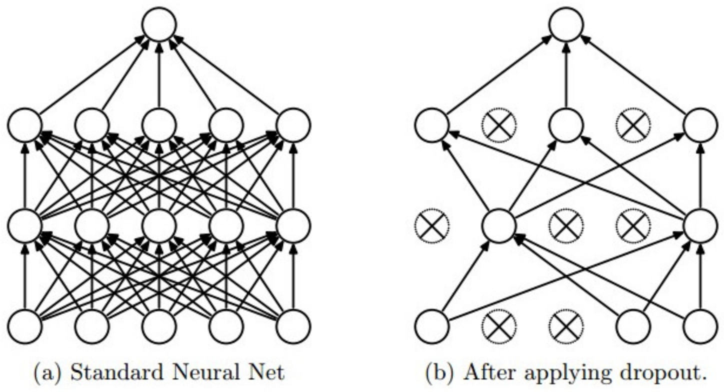
Analysis of Dropout

Jan 10, 2017
8 minute read

Overfitting is a problem in Deep Neural Networks (DNN): the model learns to classify **only** the training set, adapting itself to the training examples instead of learning decision boundaries capable of classifying generic instances. Many solutions to the overfitting problem have been presented during these years; one of them have overwhelmed the others due to its simplicity and its empirical good results: Dropout.

Dropout



Visual representation of Dropout, right from the paper. On the left there's the network before applying Dropout, on the right the same network with Dropout applied.
The network on the left it's the same network used at test time, once the parameters have been learned.

The idea behind Dropout is to train an ensemble of DNNs and average the results of the whole ensemble instead of train a single DNN.

The DNNs are built **dropping out** neurons with p probability, therefore keeping the others on with probability $q = 1 - p$. When a neuron is dropped out, its output is set to zero, no matter what the input or the associated learned parameter is.

The dropped neurons **do not contribute** to the training phase in both the forward and backward phases of back-propagation: for this reason every time a single neuron is dropped out it's like the training phase is done on a new network.

Quoting the authors:

> *In a standard neural network, the derivative received by each parameter tells it how it should change so the final loss function is reduced, given what all other units are doing. Therefore, units may change in a way that they fix up the mistakes of the other units. This may lead to complex co-adaptations. This in turn leads to overfitting*

> *because these co-adaptations do not generalize to unseen data. We hypothesize that for each hidden unit, Dropout prevents co-adaptation by making the presence of other hidden units unreliable. Therefore, a hidden unit cannot rely on other specific units to correct its mistakes.*

In short: Dropout works well in practice because it prevents the co-adaption of neurons during the training phase.

Now that we got an intuitive idea behind Dropout, let's analyze it in depth.

How Dropout works

As said before, Dropout turns off neurons with probability p and therefore let the others turned on with probability $q = 1 - p$.

Every single neuron has the same probability of being turned off. This means that:

Given

- $h(x) = xW + b$ a linear projection of a d_i-dimensional input x in a d_h-dimensional output space.
- $a(h)$ an activation function

it's possible to model the application of Dropout, in the training phase only, to the given projection as a modified activation function:

$$f(h) = D \odot a(h)$$

Where $D = (X_1, \cdots, X_{d_h})$ is a d_h-dimensional vector of Bernoulli variables X_i.

A Bernoulli random variable has the following probability mass distribution:

$$f(k;p) = \begin{cases} p & \text{if} \quad k = 1 \\ 1 - p & \text{if} \quad k = 0 \end{cases}$$

Where k are the possible outcomes.

It's evident that this random variable perfectly models the Dropout application on a single neuron. In fact, the neuron is turned off with probability $p = P(k = 1)$ and kept on otherwise.

It can be useful to see the application of Dropout on a generic i-th neuron:

$$O_i = X_i a(\sum_{k=1}^{d_i} w_k x_k + b) = \begin{cases} a(\sum_{k=1}^{d_i} w_k x_k + b) & \text{if} \quad X_i = 1 \\ 0 & \text{if} \quad X_i = 0 \end{cases}$$

where $P(X_i = 0) = p$.

Since during train phase a neuron is kept on with probability q, during the testing phase we have to emulate the behavior of the ensemble of networks used in the training phase.

To do this, the authors suggest scaling the activation function by a factor of q during the test phase in order to use the expected output produced in the training phase as the single output required in the test phase. Thus:

Train phase: $O_i = X_i a(\sum_{k=1}^{d_i} w_k x_k + b)$

Test phase: $O_i = q a(\sum_{k=1}^{d_i} w_k x_k + b)$

Inverted Dropout

Inverted Dropout

A slightly different approach is to use **Inverted Dropout**. This approach consists in the scaling of the activations during the training phase, leaving the test phase untouched.

The scale factor is the inverse of the keep probability: $\frac{1}{1-p} = \frac{1}{q}$, thus:

Train phase: $O_i = \frac{1}{q} X_i a(\sum_{k=1}^{d_i} w_k x_k + b)$

Test phase: $O_i = a(\sum_{k=1}^{d_i} w_k x_k + b)$

Inverted Dropout is how Dropout is implemented in practice in the various deep learning frameworks because it helps to define the model once and just change a parameter (the keep/drop probability) to run train and test on the same model.

Direct Dropout, instead, force you to modify the network during the test phase because if you don't multiply by q the output the neuron will produce values that are higher respect to the one expected by the successive neurons (thus the following neurons can saturate or explode): that's why Inverted Dropout is the more common implementation.

Dropout of a set of neurons

It can be easily noticed that a layer h with n neurons, in a single train step, can be seen as an ensemble of n Bernoulli experiments, each one with a probability of success equals to p.

Thus, the output of the layer h have a number of dropped neurons equals to:

$$Y = \sum_{i=1}^{d_h} X_i$$

Since every neuron is now modeled as a Bernoulli random variable and all these random variables are independent and identically distributed, the total number of dropped neuron is a random variable too, called **Binomial**:

$$Y \sim Bi(d_h, p)$$

Where the probability of getting exactly k success in n trials is given by the probability mass distribution:

$$f(k; n, p) = \binom{n}{k} p^k (1-p)^{n-k}$$

This formula can be easily explained:

- $p^k (1-p)^{n-k}$ is the probability of getting a single sequence of k successes on n trials and therefore $n - k$ failures.
- $\binom{n}{k}$ is the binomial coefficient used to calculate the number of possible sequence of success.

We can now use this distribution to analyze the probability of dropping a specified number of neurons.

When using Dropout, we define a fixed Dropout probability p for a chosen layer and we expect that a proportional number of neurons are dropped from it.

For example, if the layer we apply Dropout to has $n = 1024$ neurons and $p = 0.5$, we expect that 512 get dropped. Let's verify this statement:

$$Y = \sum_{i=1}^{1024} X_i \sim Bi(1024, 0.5)$$

$$P(Y = 512) = \binom{1024}{512} 0.5^{512} (1 - 0.5)^{1024-512} \approx 0.025$$

$$P(Y = 512) = \binom{}{512} 0.5 \quad (1 - 0.5) \quad \approx 0.025$$

Thus, the probability of dropping out exactly $np = 512$ neurons is of only 0.025!

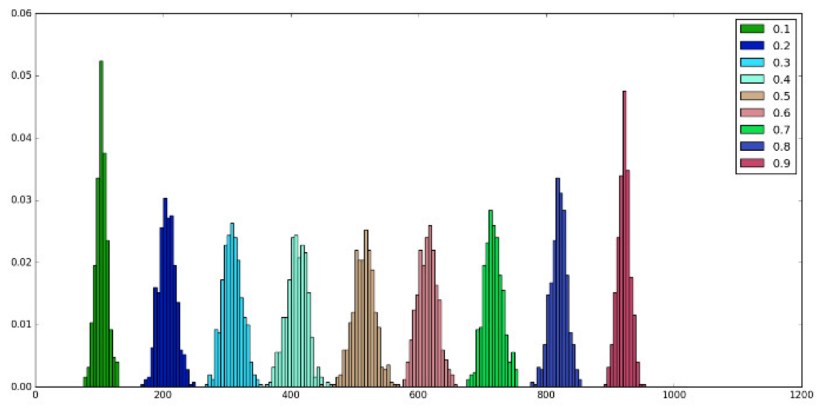
A python 3 script can help us to visualize how neurons are dropped for different values of p and a fixed value of n. The code is commented.

```python
import matplotlib.pyplot as plt
from scipy.stats import binom
import numpy as np

# number of neurons
n = 1024
# number of tests (input examples)
size = 500
# histogram bin width, for data visualization
binwidth = 5

for p in range(1, 10):
    # per layer probability
    prob = p / 10
    # generate size values from a bi(n, prob)
    rnd_values = binom.rvs(n, prob, size=size)
    # draw histogram of rnd values
    plt.hist(
        rnd_values,
        bins=[x for x in range(0, n, binwidth)],
        # normalize = extract the probabilities
        normed=1,
        # pick a random color
        color=np.random.rand(3, 1),
        # label the histogram with its probability
        label=str(prob))

plt.legend(loc='upper right')
plt.show()
```



The binomial distribution is very peaked around

np

As we can see from the image above no matter what the p value is, the number of neurons dropped on average is proportional to np, in fact:

$$E[Bi(n, p)] = np$$

Moreover, we can notice that the distribution of values is almost symmetric around $p = 0.5$ and the probability of dropping np neurons increase as the distance from $p = 0.5$ increase.

The scaling factor has been added by the authors to compensate the activation values, because they expect that during the training phase only a percentage of $1 - p$ neurons have been kept. During the training phase, instead, the 100% of neurons are kept on, thus the value should be scaled down accordingly.

Dropout & other regularizers

Dropout is often used with L2 normalization and other parameter constraint techniques (such as *Max Norm* [1]), this is not a case. Normalizations help to keep model parameters value low, in this way a parameter can't grow too much.

In brief, the L2 normalization (for example) is an additional term to the loss, where $\lambda \in [0, 1]$ is an hyper-parameter called regularization strength, $F(W; x)$ is the model and $\mathcal{E}$ is the error function between the real y and the predicted $\hat{y}$ value.

$$\mathcal{L}(y, \hat{y}) = \mathcal{E}(y, F(W; x)) + \frac{\lambda}{2} W^2$$

It's easy to understand that this additional term, when doing back-propagation via gradient descent, reduces the update amount. If η is the learning rate, the update amount of the parameter $w \in W$ is

$$w \leftarrow w - \eta \left(\frac{\partial F(W; x)}{\partial w} + \lambda w \right)$$

Dropout alone, instead, does not have any way to prevent parameter values from becoming too large during this update phase. Moreover, the inverted implementation leads the update steps to become bigger, as showed below.

Inverted Dropout and other regularizers

Since Dropout does not prevent parameters from growing and overwhelming each other, applying L2 regularization (or any other regularization technique that constraints the parameter values) can help.

Making explicit the scaling factor, the previous equation becomes:

$$w \leftarrow w - \eta \left(\frac{1}{q} \frac{\partial F(W; x)}{\partial w} + \lambda w \right)$$

It can be easily seen that when using Inverted Dropout, the learning rate is scaled by a factor of q. Since q has values in $]0, 1]$ the ratio between η and q can vary between:

$$r(q) = \frac{\eta}{q} \in [\eta = \lim_{q \to 1} r(q), +\infty = \lim_{q \to 0} r(q)]$$

For this reason, from now on we'll call q **boosting factor** because it boosts the learning rate. Moreover, we'll call $r(q)$ the **effective learning rate**.

The effective learning rate, thus, is higher respect to the learning rate chosen: for this reason normalizations that constrain the parameter values can help to simplify the learning rate selection process.

Summary

1. Dropout exists in two versions: direct (not commonly implemented) and inverted
2. Dropout on a single neuron can be modeled using a Bernoulli random variable
3. Dropout on a set of neurons can be modeled using a Binomial random variable
4. Even if the probability of dropping exactly np neurons is low, np neurons are dropped on average on a layer of n neurons.
5. Inverted Dropout boost the learning rate
6. Inverted Dropout should be using together with other normalization techniques that constrain the parameter values in order to simplify the learning rate selection procedure
7. Dropout helps to prevent overfitting in deep neural networks.

1. Max Norm impose a constraint to the parameters size. Chosen a value for the hyper-parameter c it impose the constraint $|w| \leq c$. ↩