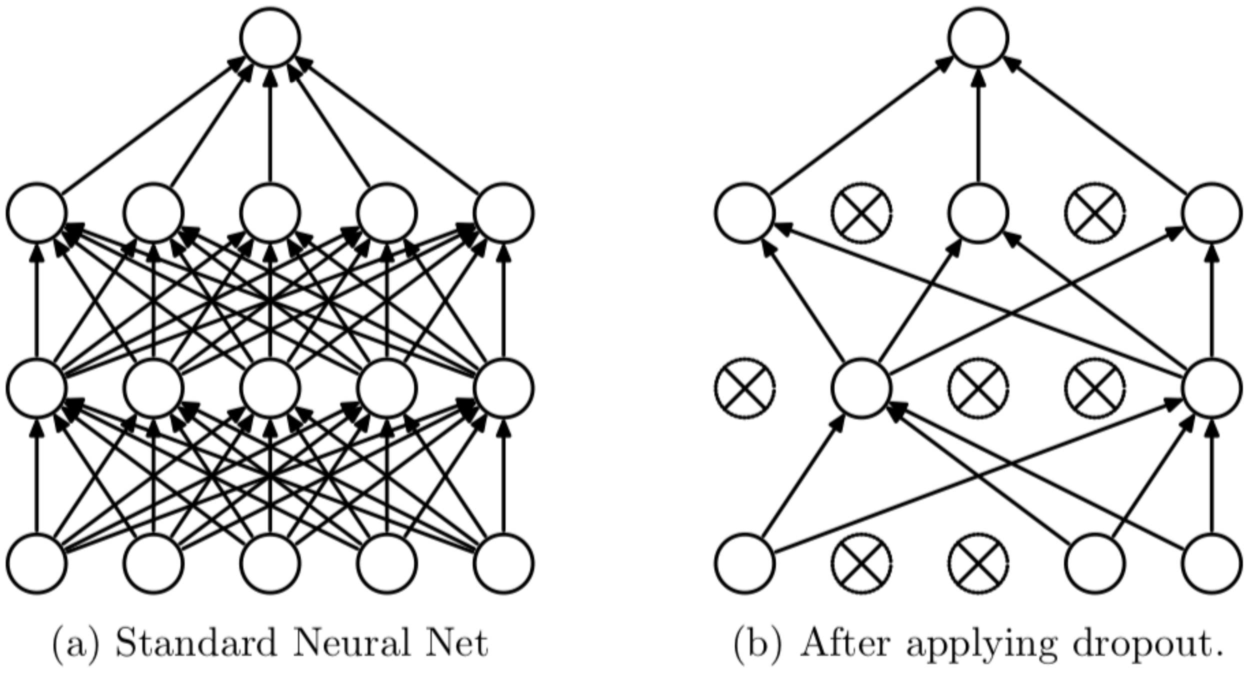
Understanding Dropout

Roan Gylberth
Jul 21, 2019 · 6 min read ★

One particular layer that is useful, yet mysterious when training neural networks is Dropout. Dropout is created as a regularization technique, that we can use to reduce the model capacity so that our model can achieve lower generalization error. The intuition is easy, we didn't use all neurons but only turn on some neuron in each training iteration with probability *p*. But how does dropout works, and is it the same as the implementation?



Dropout — Srivastava et al. (2014)

Regularization

When training neural networks, our models are prone to over-fitting, achieving very low error on training data but have a much higher error on the testing data. To counter this, we use regularization technique to lower the model capacity so that we can reduce the gap of training and testing error and our model can have similar performance in both training and testing phase.

If we recall how neural networks learn, it is learning by altering the weight parameters to minimize the loss function *L*. Hence, our optimization task is to minimize *L*.

$$\theta = \arg \min_{\theta} \; L(\theta) \tag{1}$$

Regularization works by adding some weighted regularization function in the loss so our model is better conditioned and more stable.

$$\theta = \arg \min_{\theta} \; L(\theta) + \lambda \mathcal{R}(\theta) \tag{2}$$

Where λ is the regularization parameters and R(θ) is the regularization function. A popular example of regularization technique is L2 Regularization or weight decay which use l2 norm of the weights as the regularization function, specifically

$$\mathcal{R}(\theta) = \frac{1}{2}||w||_2^2 \tag{3}$$

where *w* is the weights. We often opt to leave the biases from the equation because it could hurt our model badly and leads to under-fitting.

Model Ensemble

One particular method to reduce generalization error is by combining several different models, which we often call model ensemble or model averaging. This makes sense because, in one model, we can have errors in one part of the test data, while another model has errors in another part of the test data. Thus, by combining several models we can get a more robust result since the parts that are already correct in most models won't change and the error will be reduced. So, the averaged models will perform at least as well as any of its members.

For example, we train *k* models which each model have error *e_i.* Suppose that the errors have variance *v* and covariance *c*. Then the error from averaging all *k* models is

$$\bar{e} = \frac{}{k}\sum_i e_i \tag{4}$$

and the expected squared error of the average model error is

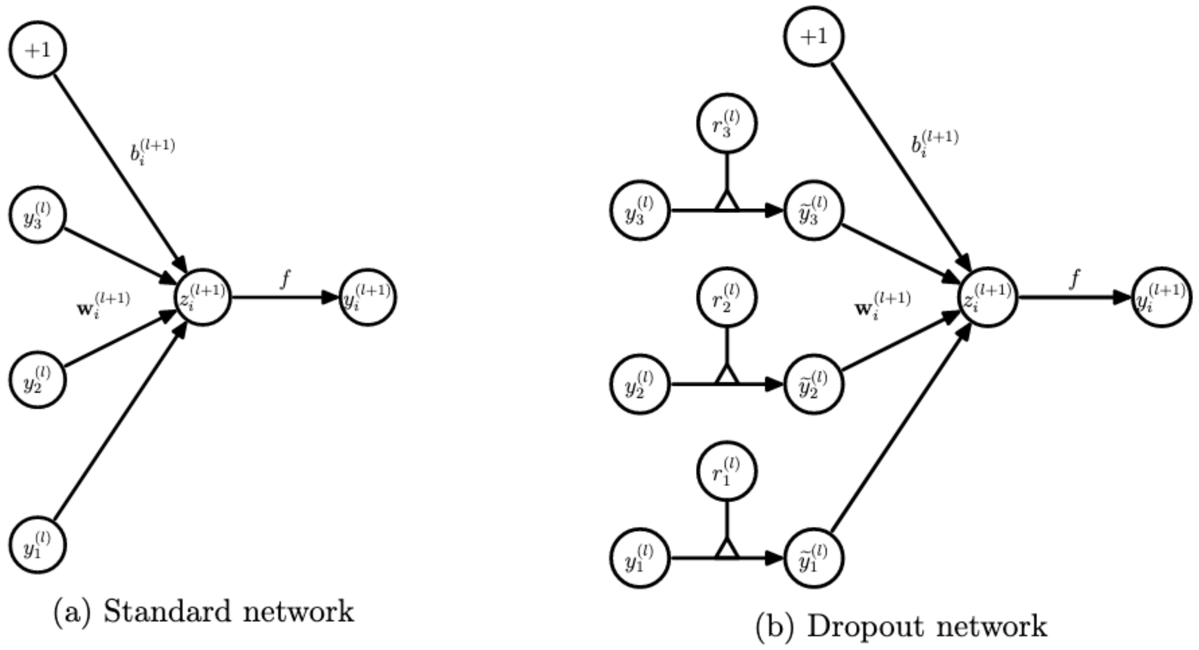
$$\begin{aligned}
\mathbb{E}\left[\bar{e}\right] &= \mathbb{E}\left[\left(\frac{1}{k}\sum_i e_i\right)^2\right] \\
&= \frac{1}{k^2}\mathbb{E}\left[\left(\sum_i e_i\right)^2\right] \\
&= \frac{1}{k^2}\mathbb{E}\left[\sum_i\left(e_i^2 + \sum_{i\neq j} e_i e_j\right)\right] \\
&= \frac{1}{k^2}\left(k\mathbb{E}\left[e_i^2\right] + \left(k^2 - k\right)\mathbb{E}\left[e_i e_j\right]\right) \\
&= \frac{1}{k}\mathbb{E}\left[e_i^2\right] + \frac{k-1}{k}\mathbb{E}\left[e_i e_j\right] \\
&= \frac{1}{k}v + \frac{k-1}{k}c
\end{aligned} \tag{5}$$

We can see that if we have perfectly correlated errors, then we have $v=c$ and the mean squared error of the models reduced to v. Also if we have perfectly uncorrelated errors, then we have $c=0$ and the mean squared error becomes (1/k)v. This shows that the expected squared error is inversely proportional to the ensemble size.

Unfortunately, training many models, especially deep neural networks are computationally expensive. The problems didn't just come at training phase, running inference on (exponentially) many models are computationally expensive too. This makes averaging many deep neural networks become unpractical, especially when the resources are limited. Srivastava et al. acknowledges this challenge and comes up with a way to get an approximation of that process.

There are two phases that we need to understand, i.e., training and testing phase.

Training Phase



Neurons at training phase — Srivastava et al. (2014)

The intuition for dropout training phase are quite simple. We turn off some neurons at training time to make the networks different at each training iteration. The way we turn off the neurons can be viewed in the figure above. We multiply each input *y_i* to a neuron *r_i* that is a two-points distribution that outputs either 0 or 1 with Bernoulli distribution. So, without dropout, the forward pass in training phase is

$$\begin{aligned} z_i^{(l+1)} &= \mathbf{w}_i^{(l+1)}\mathbf{y}^{(l)} + b_i^{(l+1)} \\ y_i^{(l+1)} &= f\left(z_i^{(l+1)}\right) \end{aligned} \tag{6}$$

where the input of the activation function *f* is just the sum of weights **w** multiplied by the input **y**. Now, with dropout, the forward pass in the training phase become

$$\begin{aligned} r_i^{(l)} &\sim \text{Bernoulli}(p) \\ \tilde{\mathbf{y}}^{(l)} &= \mathbf{r}^{(l)} * \mathbf{y}^{(l)} \end{aligned}$$

$$y_i^{(l+1)} = f\left(z_i^{(l+1)}\right)$$

where we can see in the second line, we add a neuron *r* which either keep the neuron by multiplying the input with 1 with probability *p* or shut down the neuron by multiplying the input with 0 with probability 1-*p*, then do the same forward pass as without dropout.

Testing Phase

For a neural networks with *n* neurons, the training phase can be seen as training an ensemble of 2^*n* possible different neural networks, since there are two possibilities of the state of each neuron. At the testing phase, we can use the same methods of averaging as model ensemble, that is we run the inference in all possible neural networks and average the results. However, doing this is not feasible and computationally expensive. Hence, Srivastava et al. propose a way to get the approximation of the average, that is by doing inference in the neural networks without dropout.

To get the approximation, we have to make sure that the output of the inference is the same the expected value of the inference while training. Suppose that the output of a neuron is *z* and the dropout probability *p*(*r*) is *p*, then the expected value of the neuron with dropout is

$$\begin{aligned}\mathbb{E}[rz] &= \sum_r p(r)rz \\ &= pz + (1-p)0 \\ &= pz\end{aligned} \tag{8}$$

hence, to get the same output in the testing as the expected output of the training, we can scale the weights of each neuron in testing phase by *p*. This method is called weight scaling inference.

Dropout Implementation

One example of the most straightforward dropout implementation is the one

```python
keep_prob = 0.5

def train_step(X):
    hidden_layer_1 = np.maximum(0, np.dot(W1, X) + b1)
    dropout_mask_1 = np.random.binomial(1, keep_prob, hidden_layer_1.shape)
    hidden_layer_1 *= dropout_mask_1
    hidden_layer_2 = np.maximum(0, np.dot(W2, hidden_layer_1) + b2)
    dropout_mask_2 = np.random.binomial(1, keep_prob, hidden_layer_2.shape)
    hidden_layer_2 *= dropout_mask_2
    out = np.dot(W3, hidden_layer_2) + b3

    # backward pass: compute gradients... (not shown)
    # perform parameter update... (not shown)

def predict(X):
    # ensembled forward pass
    hidden_layer_1 = np.maximum(0, np.dot(W1, X) + b1) * keep_prob # NOTE: scale the activations
    hidden_layer_2 = np.maximum(0, np.dot(W2, hidden_layer_1) + b2) * keep_prob # NOTE: scale th
    out = np.dot(W3, hidden_layer_2) + b3
```

dropout.py hosted with ♡ by GitHub view raw

This is the implementation of dropout in three layered DNN with ReLU as the activation function. See that we apply dropout before the input come to the hidden layer 2 and the output layer. Since that the Bernoulli distribution is a special case of Binomial distribution where n=1, we can use `numpy.random.binomial` to create the dropout mask.

Notice that in the prediction phase we multiply each layer with the `keep_prob`, this is the implementation of the weight scaling inference described above. However, running this for each layers can increase prediction time. So, there is another implementation that avoids this by scaling the weights by `1/keep_prob` in training time, which usually called the inverted dropout.

```python
keep_prob = 0.5

def train_step(X):
    hidden_layer_1 = np.maximum(0, np.dot(W1, X) + b1)
    dropout_mask_1 = np.random.binomial(1, keep_prob, hidden_layer_1.shape) / keep_prob
    hidden_layer_1 *= dropout_mask_1
    hidden_layer_2 = np.maximum(0, np.dot(W2, hidden_layer_1) + b2)
    dropout_mask_2 = np.random.binomial(1, keep_prob, hidden_layer_2.shape) / keep_prob

    # backward pass: compute gradients... (not shown)
    # perform parameter update... (not shown)

def predict(X):
    # ensembled forward pass
    hidden_layer_1 = np.maximum(0, np.dot(W1, X) + b1)
    hidden_layer_2 = np.maximum(0, np.dot(W2, hidden_layer_1) + b2)
    out = np.dot(W3, hidden_layer_2) + b3
```

inverted_dropout.py hosted with ♡ by GitHub view raw

Notice that instead of scaling the output by `keep_prob` in the prediction, we scale the weight by `1/keep_prob` in the training phase. In this way, the expected value of the outputs is already z, so there is no scaling needed in the prediction phase.

References:

1. N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, R. Salakhutdinov, “Dropout: A Simple Way to Prevent Neural Networks from Overfitting” (2014)
2. I. Goodfellow, Y. Bengio, A. Courville, “Deep Learning” (2015)
3. CS231N Lecture Notes, http://cs231n.github.io/neural-networks-2/
4. S. Wager, S. Wang, P. Liang, “Dropout Training as Adaptive Regularization” (2013)

Machine Learning Dropout Regularization Deep Learning Artificial Intelligence