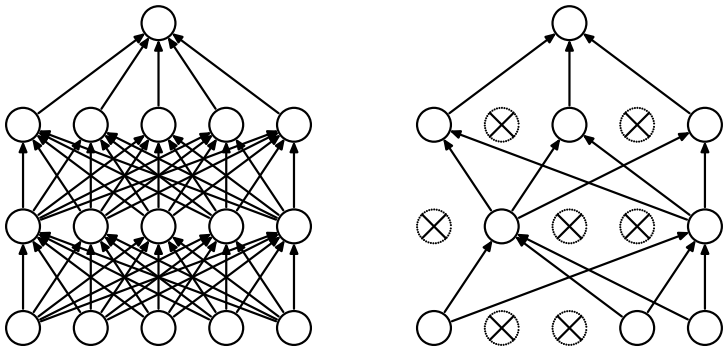
Deep learning

6.3. Dropout

François Fleuret
https://fleuret.org/dlc/

A first "deep" regularization technique is **dropout** (Srivastava et al., 2014). It consists of removing units at random during the forward pass on each sample, and putting them all back during test.



(a) Standard Neural Net (b) After applying dropout.

Figure 1: Dropout Neural Net Model. **Left**: A standard neural net with 2 hidden layers. **Right**: An example of a thinned net produced by applying dropout to the network on the left. Crossed units have been dropped.

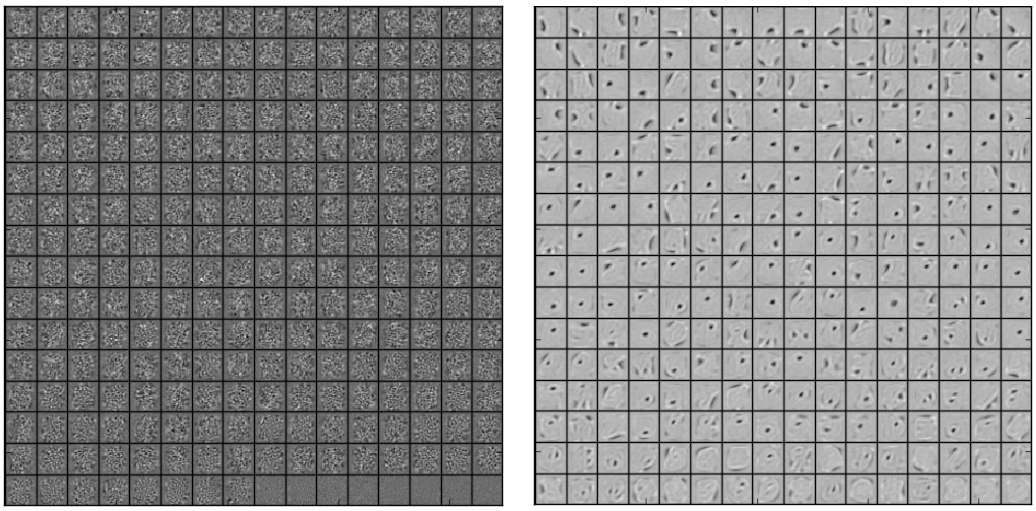
(Srivastava et al., 2014)

Notes

A key idea in deep learning is to engineer architectures to make them easier to train.
So far, we saw that we can choose the architecture (number of layers, units, filters, filter sizes, etc.), the activation function(s), and the parameter initialization.
We can go one step further by adding mechanisms specifically designed to facilitate the training, such as "dropout". As pictured on the right it removes at random some of the units during the forward pass. The units to remove are selected at random independently for every sample. The backward pass is done consistently, i.e. through the kept units alone.

This method increases independence between units, and distributes the representation. It generally improves performance.

> "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. It must perform well in a wide variety of different contexts provided by the other hidden units."
>
> (Srivastava et al., 2014)



(a) Without dropout

(b) Dropout with $p = 0.5$.

Figure 7: Features learned on MNIST with one hidden layer autoencoders having 256 rectified linear units.

(Srivastava et al., 2014)

A network with dropout can be interpreted as an ensemble of 2^N models with heavy weight sharing (Goodfellow et al., 2013).

One has to decide on which units/layers to use dropout, and with what probability p units are dropped.

During training, for each sample, as many Bernoulli variables as units are sampled independently to select units to remove.

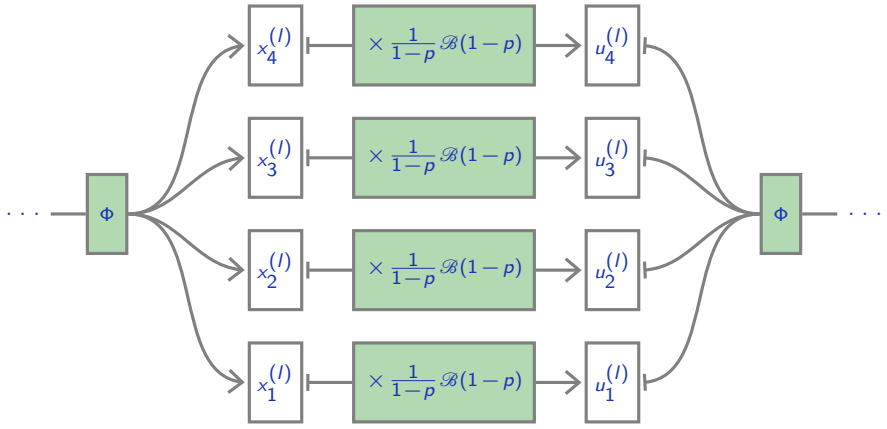
Let X be a unit activation, and D be an independent Boolean random variable of probability $1 - p$. We have

$$\mathbb{E}(D\,X) = \mathbb{E}(D)\,\mathbb{E}(X) = (1-p)\mathbb{E}(X)$$

To keep the means of the inputs to layers unchanged, the initial version of dropout was multiplying activations by $1 - p$ during test.

The standard variant in use is the "inverted dropout". It multiplies activations by $\frac{1}{1-p}$ during train and keeps the network untouched during test.

Dropout is not implemented by actually switching off units, but equivalently as a module that drops activations at random on each sample.



Dropout is implemented in PyTorch as `nn.Dropout`, which is a `torch.Module`.

In the forward pass, it samples a Boolean variable for each component of the tensor it gets as input, and zeroes entries accordingly.

Default probability to drop is $p = 0.5$, but other values can be specified.

```
>>> x = torch.full((3, 5), 1.0).requires_grad_()
>>> x
tensor([[ 1.,  1.,  1.,  1.,  1.],
        [ 1.,  1.,  1.,  1.,  1.],
        [ 1.,  1.,  1.,  1.,  1.]])
>>> dropout = nn.Dropout(p = 0.75)
>>> y = dropout(x)
>>> y
tensor([[ 0.,  0.,  4.,  0.,  4.],
        [ 0.,  4.,  4.,  4.,  0.],
        [ 0.,  0.,  4.,  0.,  0.]])
>>> l = y.norm(2, 1).sum()
>>> l.backward()
>>> x.grad
tensor([[ 0.0000,  0.0000,  2.8284,  0.0000,  2.8284],
        [ 0.0000,  2.3094,  2.3094,  2.3094,  0.0000],
        [ 0.0000,  0.0000,  4.0000,  0.0000,  0.0000]])
```

If we have a network

```
model = nn.Sequential(nn.Linear(10, 100), nn.ReLU(),
                      nn.Linear(100, 50), nn.ReLU(),
                      nn.Linear(50, 2));
```

we can simply add dropout layers

```
model = nn.Sequential(nn.Linear(10, 100), nn.ReLU(),
                      nn.Dropout(),
                      nn.Linear(100, 50), nn.ReLU(),
                      nn.Dropout(),
                      nn.Linear(50, 2));
```

A model using dropout has to be set in "train" or "test" mode.

The method `nn.Module.train(mode)` recursively sets the flag `training` to all sub-modules.

```
>>> dropout = nn.Dropout()
>>> model = nn.Sequential(nn.Linear(3, 10), dropout, nn.Linear(10, 3))
>>> dropout.training
True
>>> model.train(False)
Sequential (
  (0): Linear (3 -> 10)
  (1): Dropout (p = 0.5)
  (2): Linear (10 -> 3)
)
>>> dropout.training
False
```


Notes

A network containing a dropout layer behaves differently in train and in test.
In some specific situation, dropout can be used in test as a way to randomize the output, for instance to estimate prediction confidence.

As pointed out by Tompson et al. (2015), units in a 2d activation map are generally locally correlated, and dropout has virtually no effect. They proposed SpatialDropout, which drops channels instead of individual units.

```
>>> dropout2d = nn.Dropout2d()
>>> x = torch.full((2, 3, 2, 4), 1.)
>>> dropout2d(x)
tensor([[[[ 2.,  2.,  2.,  2.],
          [ 2.,  2.,  2.,  2.]],

         [[ 0.,  0.,  0.,  0.],
          [ 0.,  0.,  0.,  0.]],

         [[ 2.,  2.,  2.,  2.],
          [ 2.,  2.,  2.,  2.]]],


        [[[ 2.,  2.,  2.,  2.],
          [ 2.,  2.,  2.,  2.]],

         [[ 0.,  0.,  0.,  0.],
          [ 0.,  0.,  0.,  0.]],

         [[ 0.,  0.,  0.,  0.],
          [ 0.,  0.,  0.,  0.]]]])
```

Another variant is dropconnect, which drops connections instead of units.

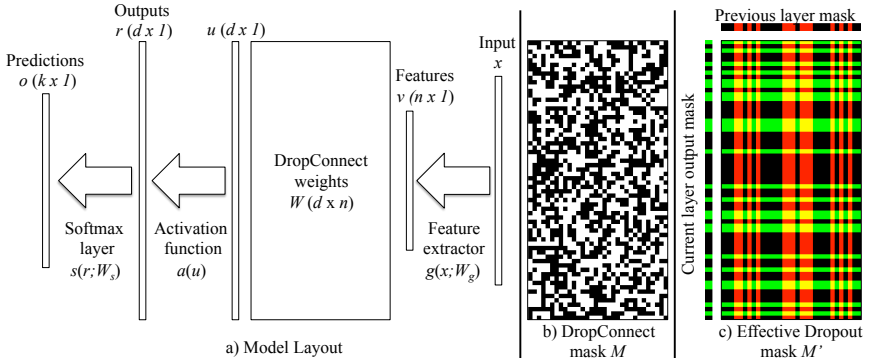


Figure 1. (a): An example model layout for a single DropConnect layer. After running feature extractor $g()$ on input x, a random instantiation of the mask M (e.g. (b)), masks out the weight matrix W. The masked weights are multiplied with this feature vector to produce u which is the input to an activation function a and a softmax layer s. For comparison, (c) shows an effective weight mask for elements that Dropout uses when applied to the previous layer's output (red columns) and this layer's output (green rows). Note the lack of structure in (b) compared to (c).

(Wan et al., 2013)

It cannot be implemented as a separate layer and is computationally intensive.

Notes

Dropconnect is computationally and memory expensive because each sample has its own weight matrix. It requires to engineer the layer themselves, and cannot be implemented as easily as a dropout module.

References

I. J. Goodfellow, D. Warde-Farley, M. Mirza, A. Courville, and Y. Bengio. **Maxout networks**. In International Conference on Machine Learning (ICML), 2013.

N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. **Dropout: A simple way to prevent neural networks from overfitting**. Journal of Machine Learning Research (JMLR), 15:1929–1958, 2014.

J. Tompson, R. Goroshin, A. Jain, Y. LeCun, and C. Bregler. **Efficient object localization using convolutional networks**. In Conference on Computer Vision and Pattern Recognition (CVPR), 2015.

L. Wan, M. D. Zeiler, S. Zhang, Y. LeCun, and R. Fergus. **Regularization of neural network using dropconnect**. In International Conference on Machine Learning (ICML), 2013.