Name: **Student ID**:

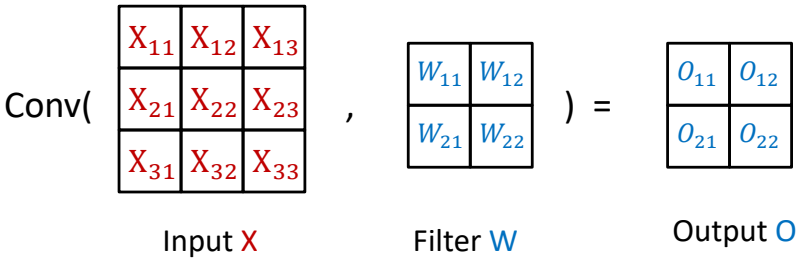
Quiz 8

Overfitting has always been the enemy of generalization. Dropout is very simple and yet very effective way to regularize networks by reducing coadaptation between the neurons.

Let $\mathbf{Y}$ be the intermediate activation of the neuron before dropout and $\mathbf{H}$ be the output of dropout.

1. Write the forward propagation of dropout in mathematical formate.
2. Write the backward propagation of dropout in mathematical formate.

3. Consider a simple convolutional layer example with input X being a 3×3 matrix and Filter W being a 2×2 matrix, as shown below:



Please show that the outputs are given by

$$
\begin{aligned}
O_{11} &= X_{11}W_{11} + X_{12}W_{12} + X_{21}W_{21} + X_{22}W_{22}, \\
O_{12} &= X_{12}W_{11} + X_{13}W_{12} + X_{22}W_{21} + X_{23}W_{22}, \\
O_{21} &= X_{21}W_{11} + X_{22}W_{12} + X_{31}W_{21} + X_{32}W_{22}, \\
O_{22} &= X_{22}W_{11} + X_{23}W_{12} + X_{32}W_{21} + X_{33}W_{22}.
\end{aligned}
$$

4. Consider the general case, where the input consists of N data points, each with C channels, height H and width W. We convolve each input with F different filters, where each filter spans all C channels and has height HH and width WW.

   Observing the rule in the above simple example, we can obtain that the outputs of the general case can be implement by

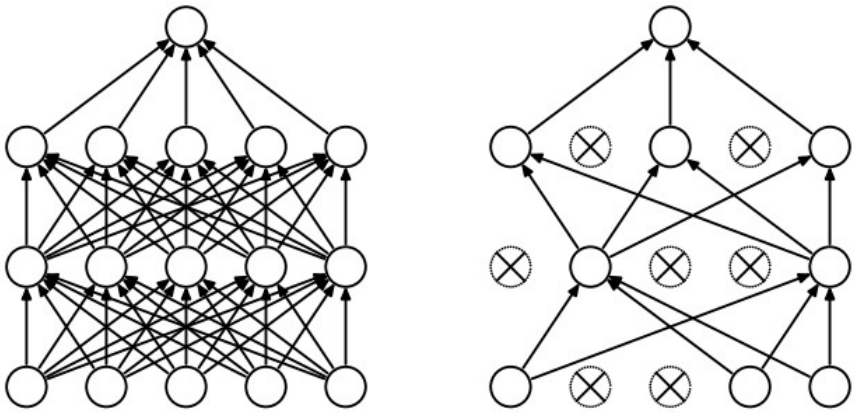
```
    N, C, H, W = x.shape
    F, C, HH, WW = w.shape
    Hout = 1 + (H + 2 * pad - HH) // stride
    Wout = 1 + (W + 2 * pad - WW) // stride
    x = torch.nn.functional.pad(x, (pad, pad, pad, pad))

    out = torch.zeros((N, F, Hout, Wout), dtype = x.dtype, device = x.device)
    for n in range(N):
      for f in range(F):
        for i in range(Hout):
          for j in range(Wout):
            out[n,f,i,j] = (x[n,:, i * stride : i * stride + HH, j * stride : j *
              stride + WW] * w[f]).sum() + b[f]
data.describe()
```

Dropout Layer – The unconventional regularization technique

Models with a large number of parameters can describe amazingly complex functions and phenomena. But this large number of parameters also means that the network has the freedom to describe the dataset it has been trained on without capturing any insights, failing to generalize to new data. This problem, known as **overfitting**, is a big problem in deep neural networks, which have millions of parameters. Several regularization techniques overcome this problem by imposing a constraint on the parameters and modifying the cost function. Dropout is a recent advancement in regularization (original paper), which unlike other techniques, works by modifying the network itself.

Dropout works by randomly and temporarily deleting neurons in the hidden layer during the training with probability p. We forward propagate input through this modified layer which has $n \times p$ active neurons and also backpropagate the result through the same. During testing/prediction, we feed the input to unmodified layer, but scale the layer output with p.



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

So how does dropout help in regularization?

Heuristically, using dropout is like we are training on different set of network. So the output produced is like using an averaging scheme on output of a large number of networks, which is often found to be powerful way of reducing overfitting.

Also, since a neuron cannot rely on the presence of other neurons, it is forced to learn features that are not dependent on the presence of other neurons. Thus network learns robust features, and are less susceptible to noise.

How and where to use dropout?

Since dropout does not constraints the parameter, applying L2 regularization or any other parameter based regularization should be used along with dropout. It is because while using dropout (or inverted

dropout, explained later) the effective learning rate is higher than the rate chosen, and so parameter based regularization can help simplify selecting proper learning rate.

Dropout is mostly used in fully connected layers and not with convolutional layer because convolutional layer have considerable resistance to overfitting due to shared weights of the filters, and so there is a less need for dropout.

Forward Propagation

For the forward pass, we know that each neuron has a probability of being turned off by probability p. It is possible to model the application of Dropout, during training phase, by transforming the input as:

$$\mathbf{H} = \mathbf{D} \odot \sigma(\mathbf{Z}),$$

where $\mathbf{H}$ is the output activation, $\mathbf{D}$ is a vector of Bernoulli variables and $\mathbf{Y} = \sigma(\mathbf{Z})$ is the intermediate activation of the neuron before dropout. A Bernoulli random variable is defined as

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

where D are the possible outcomes.

Thus when the output of the neuron is scaled to 0, it does not contribute any further during both forward and backward pass, which is essentially dropout.

During training phase, we trained the network with only a subset of the neurons. So during testing, we have to scale the output activations by factor of p, which means we have to modify the network during test phase. A simpler and commonly used alternative called **Inverted Dropout** scales the output activation during training phase by $\frac{1}{p}$ so that we can leave the network during testing phase untouched.

The implementation is fairly simple:

```
# p: Dropout parameter. We keep each neuron output with probability p.
#
#    NOTE 2: Keep in mind that p is the probability of **keep** a neuron
#    output; this might be contrary to some sources, where it is referred to
#    as the probability of dropping a neuron output.
mask = (np.random.rand(*x.shape) < p) / p
out = mask * x
```

Backward Propagation

The dropout layer has no learnable parameters, and doesn't change the volume size of the output. So the backward pass is fairly simple. We simply back propagate the gradients through the neurons that were not killed off during the forward pass, as changing the output of the killed neurons doesn't change the output, and thus their gradient is 0.

Mathematically, we need to find the input gradient $\partial L/\partial \mathbf{Y}$. Using chain rule,

$$\begin{aligned}
\frac{\partial L}{\partial \mathbf{Y}} &= \frac{\partial L}{\partial \mathbf{H}} \odot \frac{\partial \mathbf{H}}{\partial \mathbf{Y}} \\
&= \frac{\partial L}{\partial \mathbf{H}} \odot \frac{\partial \begin{cases} Y_{ij}, & \text{if } D_{ij} = 1 \\ 0, & \text{if } D_{ij} = 0 \end{cases}}{\partial \mathbf{Y}} \\
&= \frac{\partial L}{\partial \mathbf{H}} \odot \partial \begin{cases} 1, & \text{if } D_{ij} = 1 \\ 0, & \text{if } D_{ij} = 0 \end{cases} \\
&= \frac{\partial L}{\partial \mathbf{H}} \odot \mathbf{D}
\end{aligned}$$

Translating this to python, we have:

```
# mask is saved during forward pass
dx = dout * mask
```

Source Code

Here is the source code for Dropout layer with forward and backward API implemented.

```
def dropout_forward(x, dropout_param):
    """
    Performs the forward pass for (inverted) dropout.

    Inputs:
    - x: Input data, of any shape
    - dropout_param: A dictionary with the following keys:
      - p: Dropout parameter. We keep each neuron output with probability p.
      - mode: 'test' or 'train'. If the mode is train, then perform dropout;
        if the mode is test, then just return the input.
      - seed: Seed for the random number generator. Passing seed makes this
        function deterministic, which is needed for gradient checking but not
        in real networks.

    Outputs:
    - out: Array of the same shape as x.
    - cache: tuple (dropout_param, mask). In training mode, mask is the dropout
      mask that was used to multiply the input; in test mode, mask is None.

    NOTE: Please implement **inverted** dropout, not the vanilla version of dropout.
    See http://cs231n.github.io/neural-networks-2/#reg for more details.

    NOTE 2: Keep in mind that p is the probability of **keep** a neuron
    output; this might be contrary to some sources, where it is referred to
    as the probability of dropping a neuron output.
    """
    p, mode = dropout_param['p'], dropout_param['mode']
```

```
    if 'seed' in dropout_param:
        np.random.seed(dropout_param['seed'])

    mask = None
    out = None

    if mode == 'train':
        ###########################################################################
        # TODO: Implement training phase forward pass for inverted dropout.   #
        # Store the dropout mask in the mask variable.                            #
        ###########################################################################
        mask = (np.random.randn(*x.shape) < p) / p
        out = x * mask
        ###########################################################################
        #                             END OF YOUR CODE                            #
        ###########################################################################
    elif mode == 'test':
        ###########################################################################
        # TODO: Implement the test phase forward pass for inverted dropout.   #
        ###########################################################################
        out = x
        ###########################################################################
        #                              END OF YOUR CODE                           #
        ###########################################################################

    cache = (dropout_param, mask)
    out = out.astype(x.dtype, copy=False)

    return out, cache


def dropout_backward(dout, cache):
    """
    Perform the backward pass for (inverted) dropout.

    Inputs:
    - dout: Upstream derivatives, of any shape
    - cache: (dropout_param, mask) from dropout_forward.
    """
    dropout_param, mask = cache
    mode = dropout_param['mode']

    dx = None
    if mode == 'train':
        ###########################################################################
        # TODO: Implement training phase backward pass for inverted dropout   #
        ###########################################################################
        dx = dout * mask
        ###########################################################################
        #                            END OF YOUR CODE                             #
        ###########################################################################
    elif mode == 'test':
        dx = dout
    return dx
```