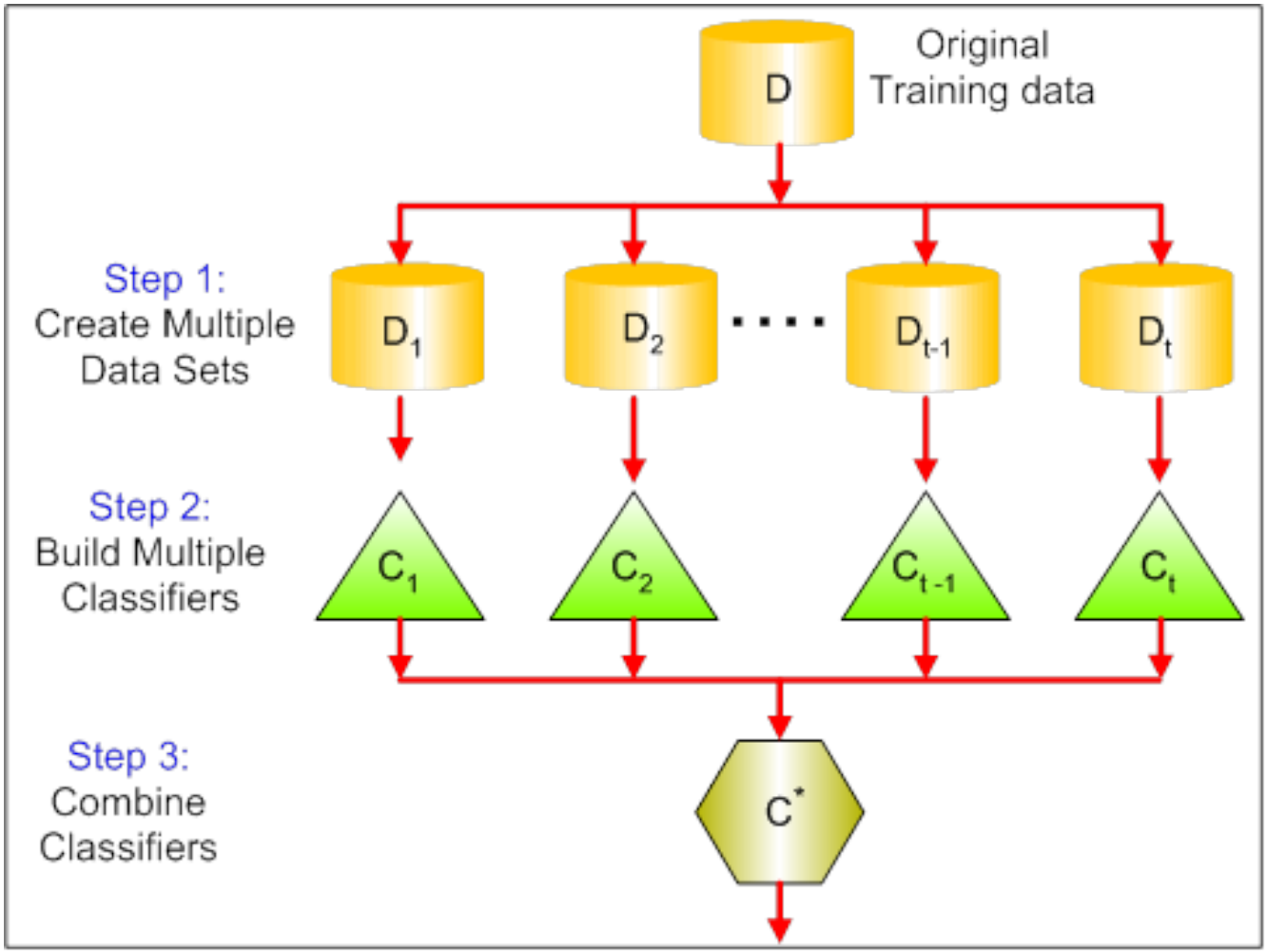
Bagging

Bagging


D
Original
Training data
Step 1:
Create Multiple
Data Sets
D1
D2
Dt-1
Dt
Step 2:
Build Multiple
Classifiers
C1
C2
Ct-1
Ct
Step 3:
Combine
Classifiers
C*

Bagging

1. From a given dataset, construct multiple training sets by sampling with replacement $(D_1, D_2, \ldots, D_t)$
2. Train i^{th} instance of the classifier C_i using training set D_i
3. Combine the output of different models to reduce generalization error
4. The models can correspond to different classifiers
5. It could be different instances of the same classifier trained with:
   - different hyperparameters
   - different features
   - different samples of the training data

Why Bagging Works

1. Average prediction $f_A(\mathbf{x}) = \sum_{i=1}^{t} f_i(\mathbf{x})$
2. Average prediction error is

$$E = \frac{1}{t} \sum_{i=1}^{t} \mathbb{E}[(y - f_i(\mathbf{x})^2]$$

3. Let E_A denotes the error of average predictor f_A. Thus,

$$E_A = \mathbb{E}[(y - f_A(\mathbf{x}))^2]$$

Why Bagging Works

Average prediction error can be re-written as follows.

$$E = \frac{1}{t}\sum_{i=1}^{t}\mathbb{E}[(y - f_i(\mathbf{x})^2] = \frac{1}{t}\sum_{i=1}^{t}\mathbb{E}[y^2 + f_i^2(\mathbf{x}) - 2yf_i(\mathbf{x})]$$

$$= \mathbb{E}[y^2] - 2\mathbb{E}[y\frac{1}{t}\sum_{i=1}^{t}f_i(\mathbf{x})] + \frac{1}{t}\sum_{i=1}^{t}\mathbb{E}[f_i^2(\mathbf{x})]$$

$$= \mathbb{E}[y^2] - 2\mathbb{E}[yf_A(\mathbf{x})] + \mathbb{E}[\frac{1}{t}\sum_{i=1}^{t}f_i^2(\mathbf{x})]$$

Using Jensen's inequality, we get the following.

$$\frac{1}{t}\sum_{i=1}^{t}f_i^2(\mathbf{x}) = \mathbb{E}_i[f_i^2(\mathbf{x})] \geq (\mathbb{E}_i[f_i(\mathbf{x})])^2 = \left(\frac{1}{t}\sum_{i=1}^{t}f_i(\mathbf{x})\right)^2$$

Why Bagging Works

Using $\frac{1}{t}\sum_{i=1}^{t} f_i^2(\mathbf{x}) \geq \left(\frac{1}{t}\sum_{i=1}^{t} f_i(\mathbf{x})\right)^2$, we get

$$
\begin{aligned}
E &\geq \mathbb{E}[y^2] - 2\mathbb{E}[y f_A(\mathbf{x})] + \mathbb{E}\left[\left(\frac{1}{t}\sum_{i=1}^{t} f_i(\mathbf{x})\right)^2\right], \text{ Jensen's inequality} \\
&= \mathbb{E}[y^2] - 2\mathbb{E}[y f_A(\mathbf{x})] + \mathbb{E}[f_A^2(\mathbf{x})] \\
&= \mathbb{E}[(y - f_A(\mathbf{x}))^2] = E_A
\end{aligned}
$$

Thus, f_A has lower prediction error.[1]

[1] Leo Breiman. 1996. Bagging predictors. Mach. Learn. 24, 2 (August 1996), 123-140.

How Bagging Works

- Consider for example a set of k regression models.
- Suppose that each model makes an error ϵ_i on each example, with the errors drawn from a zero-mean multivariate normal distribution with variances $\mathbb{E}[\epsilon_i^2] = V$ and covariances $\mathbb{E}[\epsilon_i \epsilon_j] = C$.
- Then the error made by the average prediction of all the ensemble models is $\frac{1}{k}\sum_{i=1}^{k} \epsilon_i$. The expected squared error of the ensemble predictor is

$$\text{mse} = \mathbb{E}\left[\left(\frac{1}{k}\sum_{i=1}^{k}\epsilon_i\right)^2\right] = \frac{1}{k^2}\mathbb{E}\left[\sum_{i=1}^{k}\epsilon_i^2 + \sum_{i=1}^{k}\sum_{j\neq i}\epsilon_i\epsilon_j\right]$$

$$= \frac{1}{k^2}\left[\sum_{i=1}^{k}\mathbb{E}[\epsilon_i^2] + \sum_{i=1}^{k}\sum_{j\neq i}\mathbb{E}[\epsilon_i\epsilon_j]\right]$$

$$= \frac{1}{k^2}[kV + k(k-1)C] = \frac{V}{k} + \frac{(k-1)C}{k}$$

Implications

- If the errors of the model are perfectly correlated then $V = C$ and mse $= V$ [bagging does not help: the mse of the ensemble is as bad as the individual models]
- If the errors of the model are independent or uncorrelated then $C = 0$ and the mse of the ensemble reduces to $\frac{V}{k}$. On average, the ensemble will perform at least as well as its individual members.
- Typically model averaging(bagging ensemble) always helps

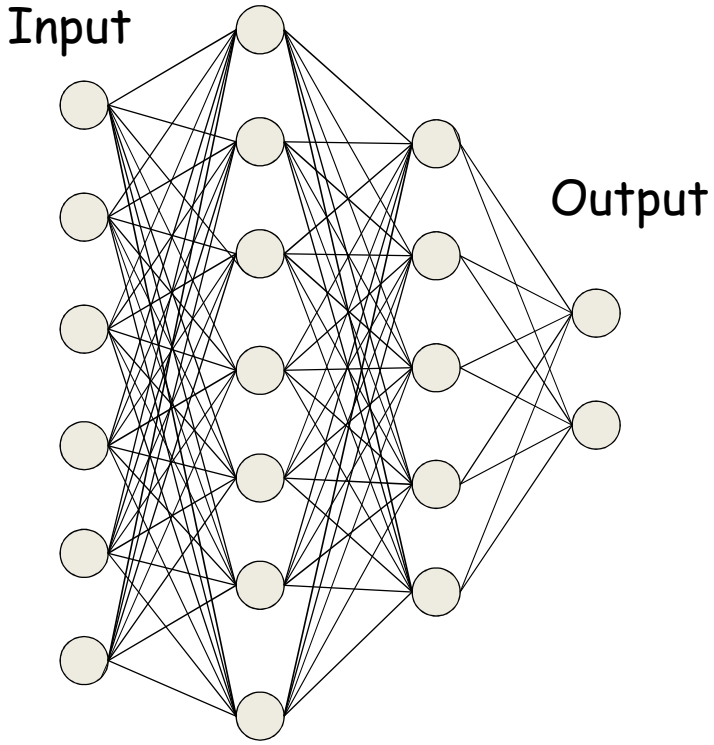
Challenges to Implement Bagging

1. Training several large neural networks for making an ensemble is prohibitively expensive
2. **Option 1**: Train several neural networks having different architectures(obviously expensive)
3. **Option 2**: Train multiple instances of the same network using different training samples (again expensive)
4. Even if we manage to train with option 1 or option 2, combining several models at test time is infeasible in real time applications

Dropout to do Bagging

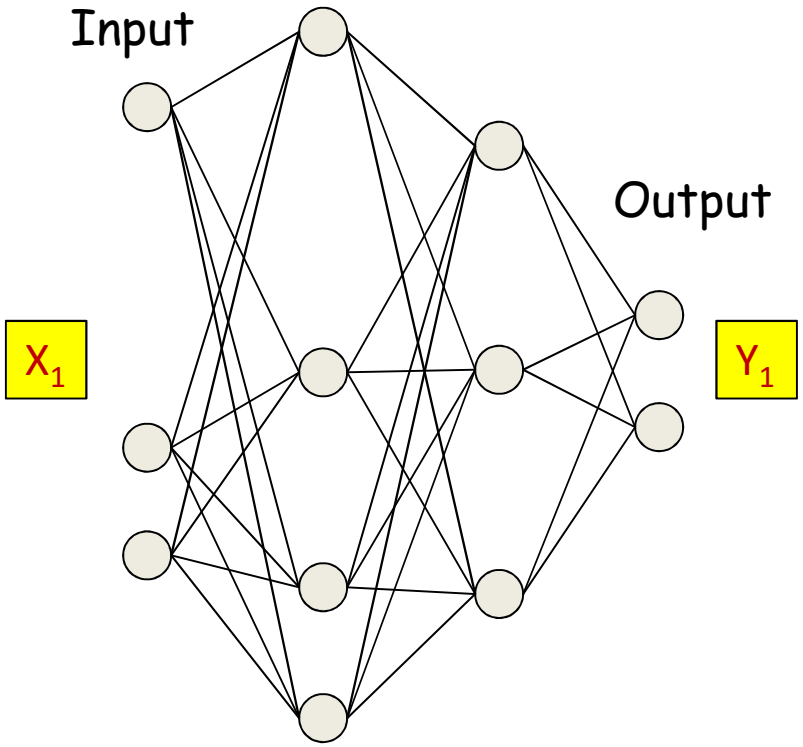
- Dropout is a technique which addresses both these issues.
- Effectively it allows training several neural networks without any significant computational overhead.
- Also gives an efficient approximate way of combining exponentially many different neural networks.

Dropout



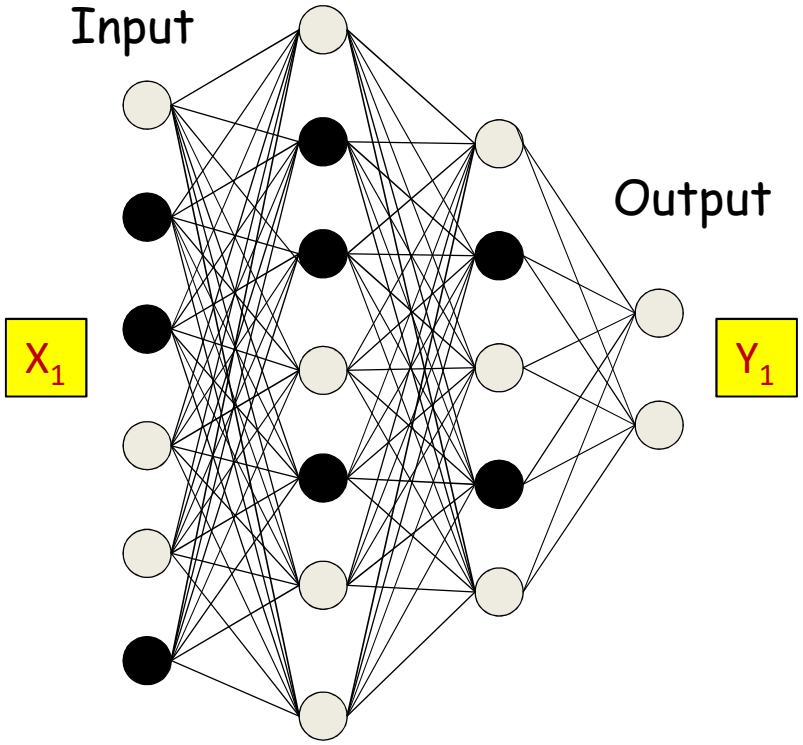
- **During training:** For each input, at each iteration, "turn off" each neuron with a probability 1-α

Dropout



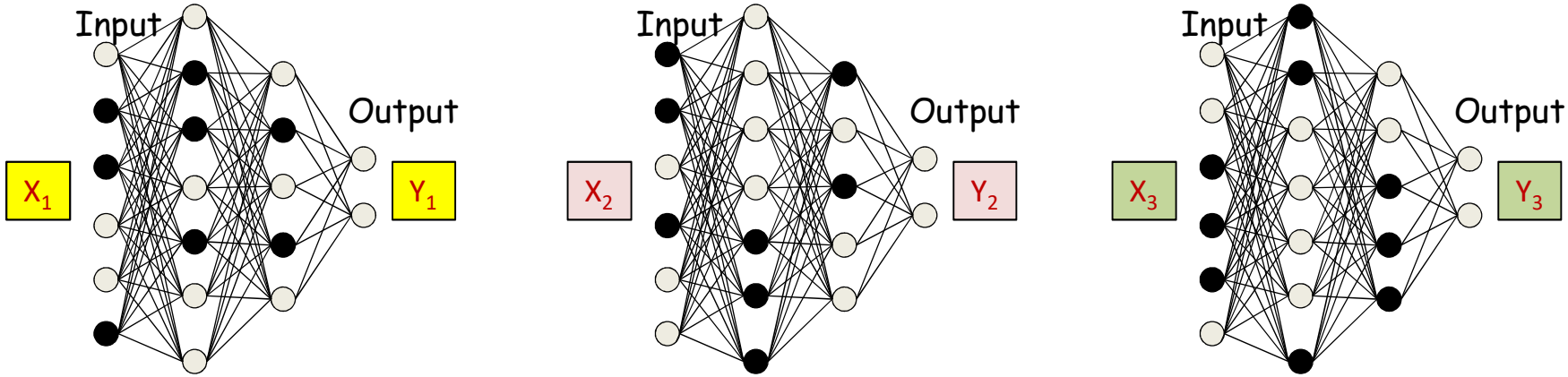
- **During training:** For each input, at each iteration, "turn off" each neuron with a probability 1-α
  - Also turn off inputs similarly

Dropout



- **During training:** For each input, at each iteration, "turn off" each neuron (including inputs) with a probability 1-α
  - In practice, set them to 0 according to the success of a Bernoulli random number generator with success probability 1-α

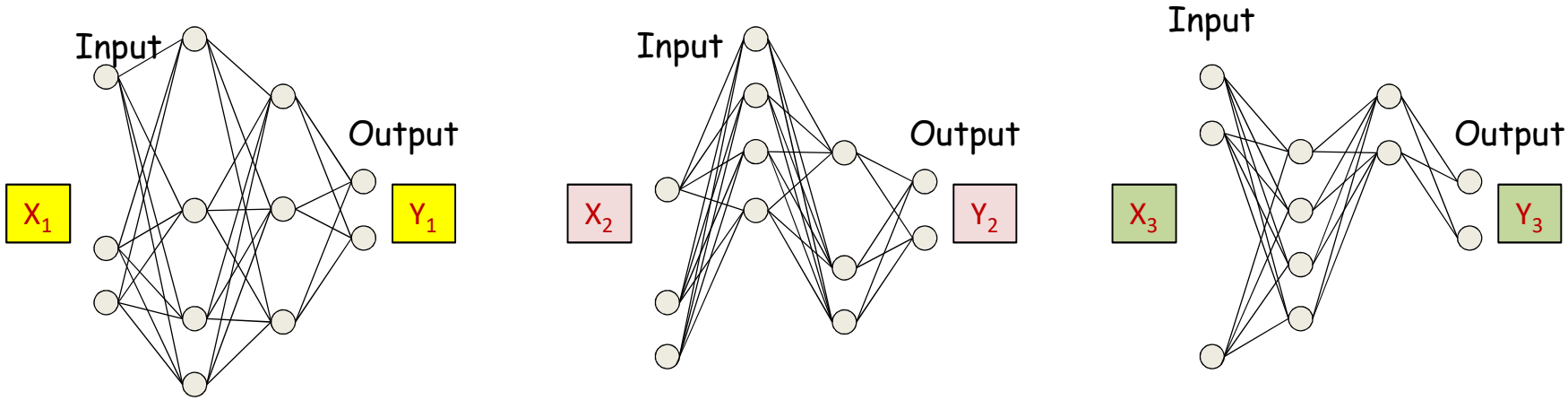
Dropout



The pattern of dropped nodes changes for *each input*
i.e. in every pass through the net

- **During training:** For each input, at each iteration, "turn off" each neuron (including inputs) with a probability 1-α
  - In practice, set them to 0 according to the success of a Bernoulli random number generator with success probability 1-α

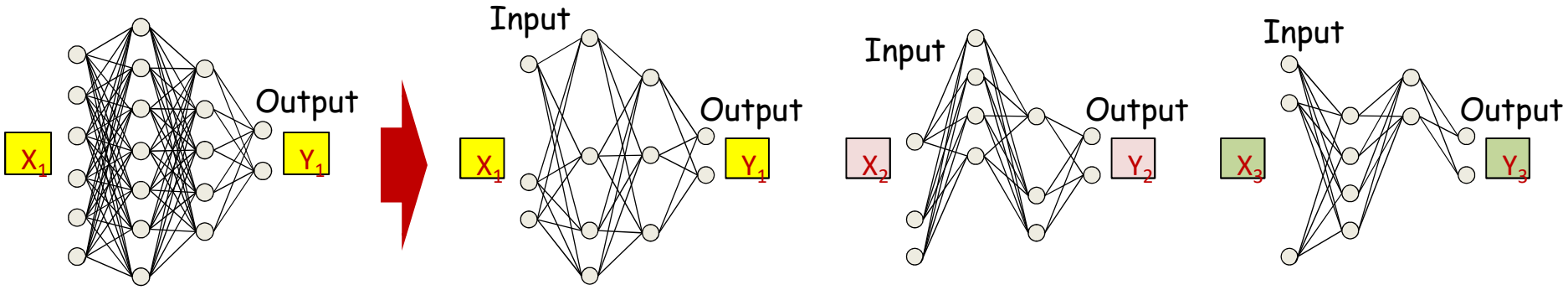
Dropout



The pattern of dropped nodes changes for *each input*
i.e. in every pass through the net

- **During training:** Backpropagation is effectively performed only over the remaining network
  - The effective network is different for different inputs
  - Gradients are obtained only for the weights and biases *from* "On" nodes *to* "On" nodes
    - For the remaining, the gradient is just 0

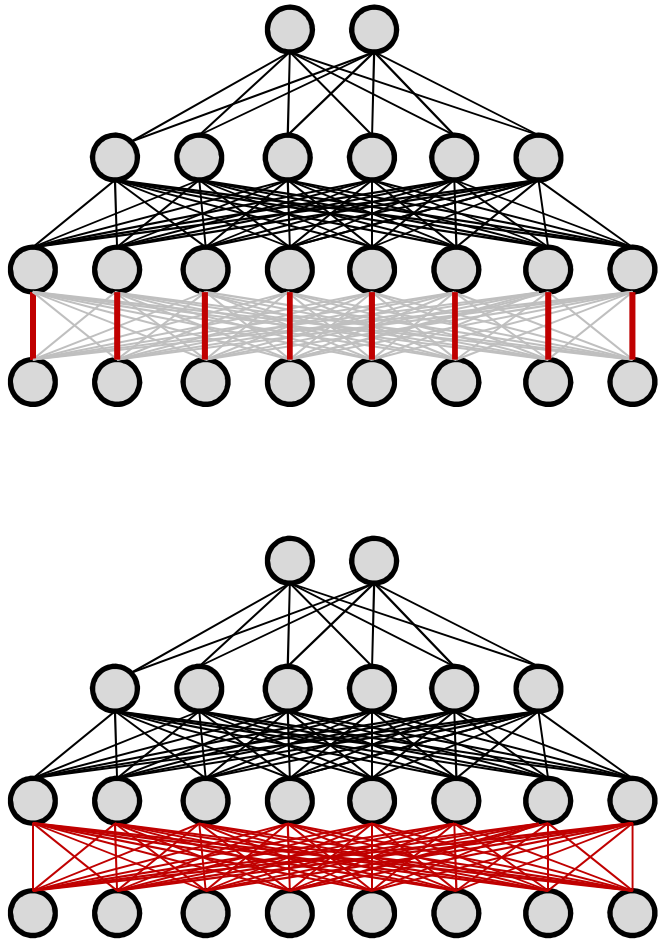
Statistical Interpretation



- For a network with a total of N neurons, there are 2^N possible sub-networks
  - Obtained by choosing different subsets of nodes
  - Dropout *samples* over all 2^N possible networks
  - Effectively learns a network that *averages* over all possible networks
    - Bagging

Dropout as a mechanism to increase pattern density

- Dropout forces the neurons to learn “rich” and redundant patterns
- E.g. without dropout, a non-compressive layer may just “clone” its input to its output
  - Transferring the task of learning to the rest of the network upstream
- Dropout forces the neurons to learn denser patterns
  - With redundancy



The forward pass

- Input: D dimensional vector $\mathbf{x} = [x_j,\ j = 1 \ldots D]$
- Set:
  - $D_0 = D,$ is the width of the 0th (input) layer
  - $y_j^{(0)} = x_j,\ j = 1 \ldots D; \qquad y_0^{(k=1\ldots N)} = x_0 = 1$
- For layer $k = 1 \ldots N$
  - For $j = 1 \ldots D_k$
    - $z_j^{(k)} = \sum_{i=0}^{N_k} w_{i,j}^{(k)} y_i^{(k-1)} + b_j^{(k)}$
    - $y_j^{(k)} = f_k\left(z_j^{(k)}\right)$
    - If $(k = dropout\ layer)$:
      - $mask(k,j) = Bernoulli(\alpha)$
      - If $mask(k,j) == 0$
        - » $y_j^{(k)} = 0$
- Output:
  - $Y = y_j^{(N)}, j = 1..D_N$

Backward Pass

- Output layer (N) :
  - $\frac{\partial Div}{\partial Y_i} = \frac{\partial Div(Y,d)}{\partial y_i^{(N)}}$
  - $\frac{\partial Div}{\partial z_i^{(k)}} = f_k'\left(z_i^{(k)}\right)\frac{\partial Div}{\partial y_i^{(k)}}$
- For layer $k = N-1$ *downto* 0
  - For $i = 1 \ldots D_k$
    - If (not dropout layer OR $mask(k,i)$)
      - $\frac{\partial Div}{\partial y_i^{(k)}} = \sum_j w_{ij}^{(k+1)} \frac{\partial Div}{\partial z_j^{(k+1)}} mask(k+1,j)$
      - $\frac{\partial Div}{\partial z_i^{(k)}} = f_k'\left(z_i^{(k)}\right)\frac{\partial Div}{\partial y_i^{(k)}}$
      - $\frac{\partial Div}{\partial w_{ij}^{(k+1)}} = y_i^{(k)} \frac{\partial Div}{\partial z_j^{(k+1)}} mask(k+1,j)$ for $j = 1 \ldots D_{k+1}$
    - Else
      - $\frac{\partial Di}{\partial z_i^{(k)}} = 0$

What each neuron computes

- Each neuron actually has the following activation:

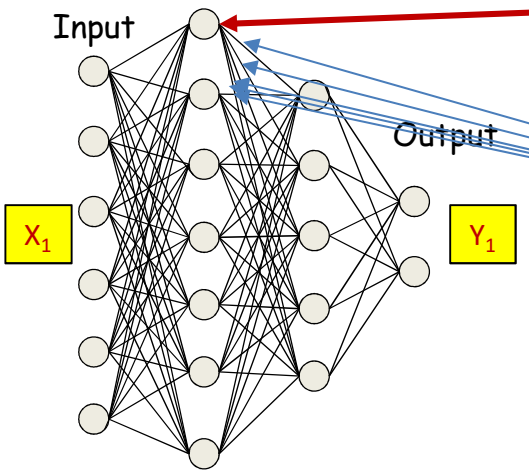
$$y_i^{(k)} = D\sigma\left(\sum_j w_{ji}^{(k)} y_j^{(k-1)} + b_i^{(k)}\right)$$

  - Where D is a Bernoulli variable that takes a value 1 with probability α
- D may be switched on or off for individual sub networks, but over the ensemble, the *expected output* of the neuron is

$$y_i^{(k)} = \alpha\sigma\left(\sum_j w_{ji}^{(k)} y_j^{(k-1)} + b_i^{(k)}\right)$$

- During *test* time, we will use the *expected* output of the neuron
  - Which corresponds to the bagged average output
  - Consists of simply scaling the output of each neuron by α

Dropout during test: implementation



apply α here (to the output of the neuron) OR..

Push the α to all outgoing weights

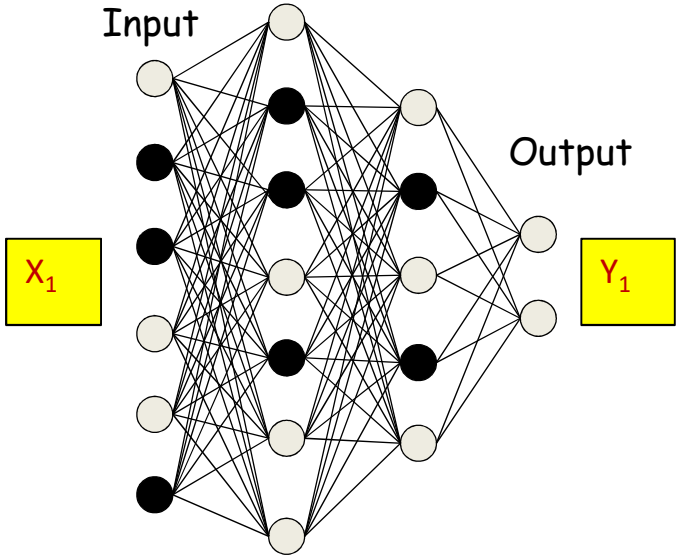
$$y_i^{(k)} = \alpha\sigma\left(z_i^{(k)}\right)$$

$$\begin{aligned} z_i^{(k)} &= \sum_j w_{ji}^{(k)} y_j^{(k-1)} + b_i^{(k)} \\ &= \sum_j w_{ji}^{(k)} \alpha\sigma\left(z_j^{(k-1)}\right) + b_i^{(k)} \\ &= \sum_j \left(\alpha w_{ji}^{(k)}\right) \sigma\left(z_j^{(k-1)}\right) + b_i^{(k)} \end{aligned}$$

$$W_{test} = \alpha W_{trained}$$

- Instead of multiplying every output by α, multiply all weights by α

Dropout : alternate implementation

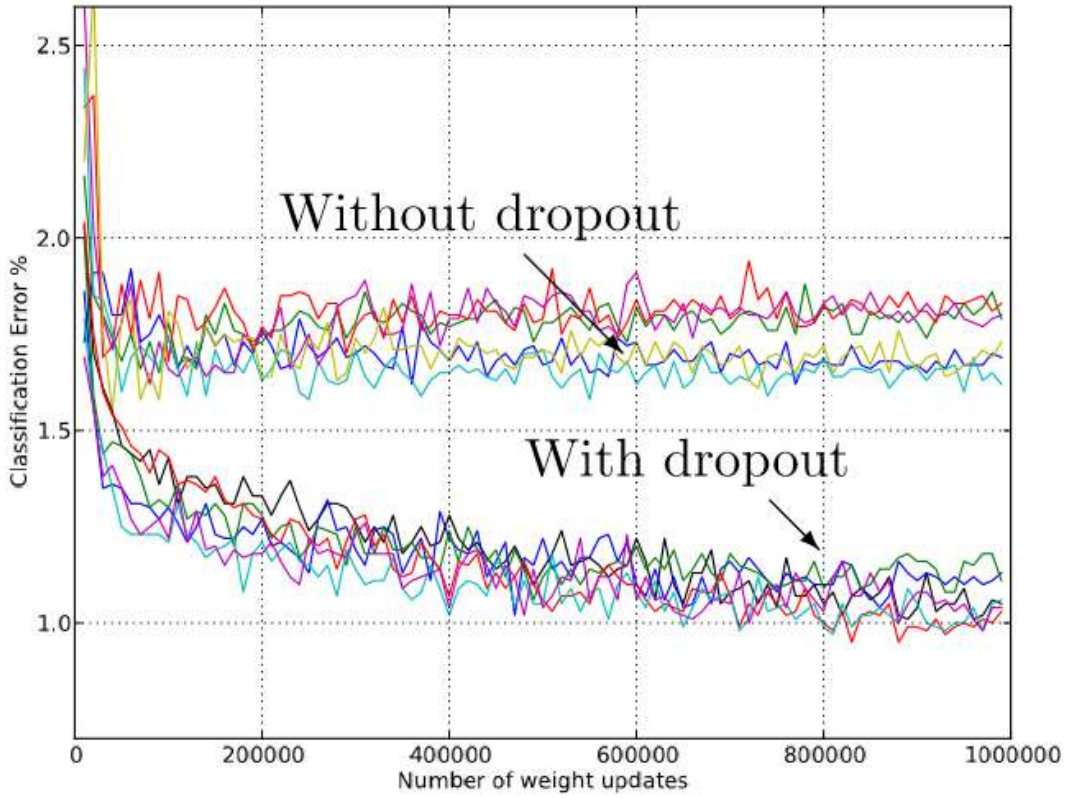


- Alternately, during *training,* replace the activation of all neurons in the network by $\alpha^{-1}\sigma(.)$
  - This does not affect the dropout procedure itself
  - We will use $\sigma(.)$ as the activation during testing, and not modify the weights

The forward pass (testing)

- Input: D dimensional vector $\mathbf{x} = [x_j,\ j = 1 \ldots D]$
- Set:
  - $D_0 = D$, is the width of the 0th (input) layer
  - $y_j^{(0)} = x_j,\ j = 1 \ldots D; \quad y_0^{(k=1\ldots N)} = x_0 = 1$
- For layer $k = 1 \ldots N$
  - For $j = 1 \ldots D_k$
    - $z_j^{(k)} = \sum_{i=0}^{N_k} w_{i,j}^{(k)} y_i^{(k-1)} + b_j^{(k)}$
    - $y_j^{(k)} = f_k\left(z_j^{(k)}\right)$
    - If $(k = dropout\ layer)$:
      - » $y_j^{(k)} = y_j^{(k)}/\alpha$
    - Else
      - » $y_j^{(k)} = 0$
- Output:
  - $Y = y_j^{(N)}, j = 1..D_N$

Dropout: Typical results



- From Srivastava et al., 2013. Test error for different architectures on MNIST with and without dropout
  - 2-4 hidden layers with 1024-2048 units