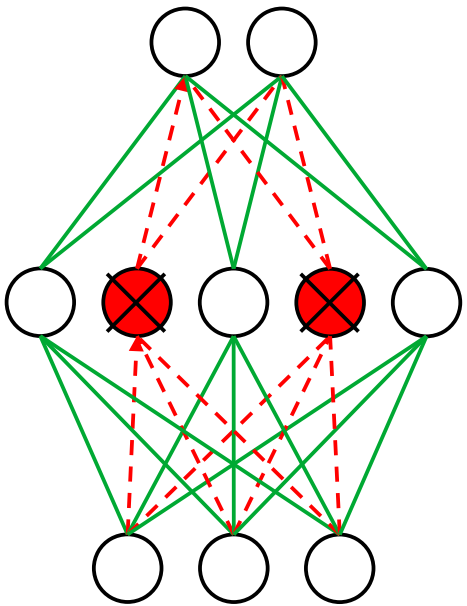


6. Dropout

Part 1: Overview and zero-out step

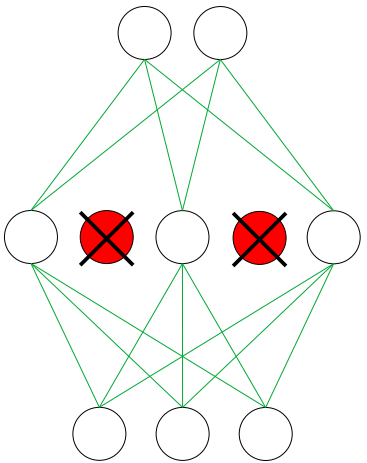
This video was produced in Korean and translated into English, and the audio was generated by AI (TTS).

www.youtube.com/@meanxai

MX-AI

[MXDL-6-01]
1. Dropout overview: Zero-out and Scaling
2. Ensemble effect
3. Zero-out & Error Backpropagation
4. Zero-out verification code

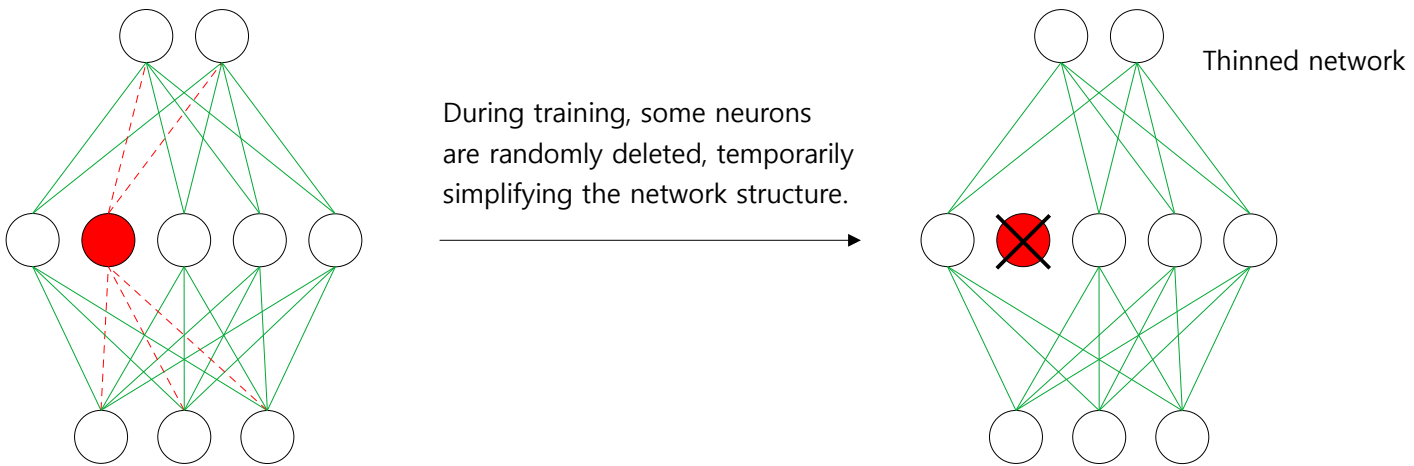
[MXDL-6-02]
5. Scale-down during prediction stage
6. Scale-up during training stage
7. Scale-up verification code
8. Optimal dropout rate
9. Dropout and Regularization
10. Implement a neural network model with Dropout using Keras



MX-AI

Dropout overview: Zero-out and Scale-up (or scale-down)

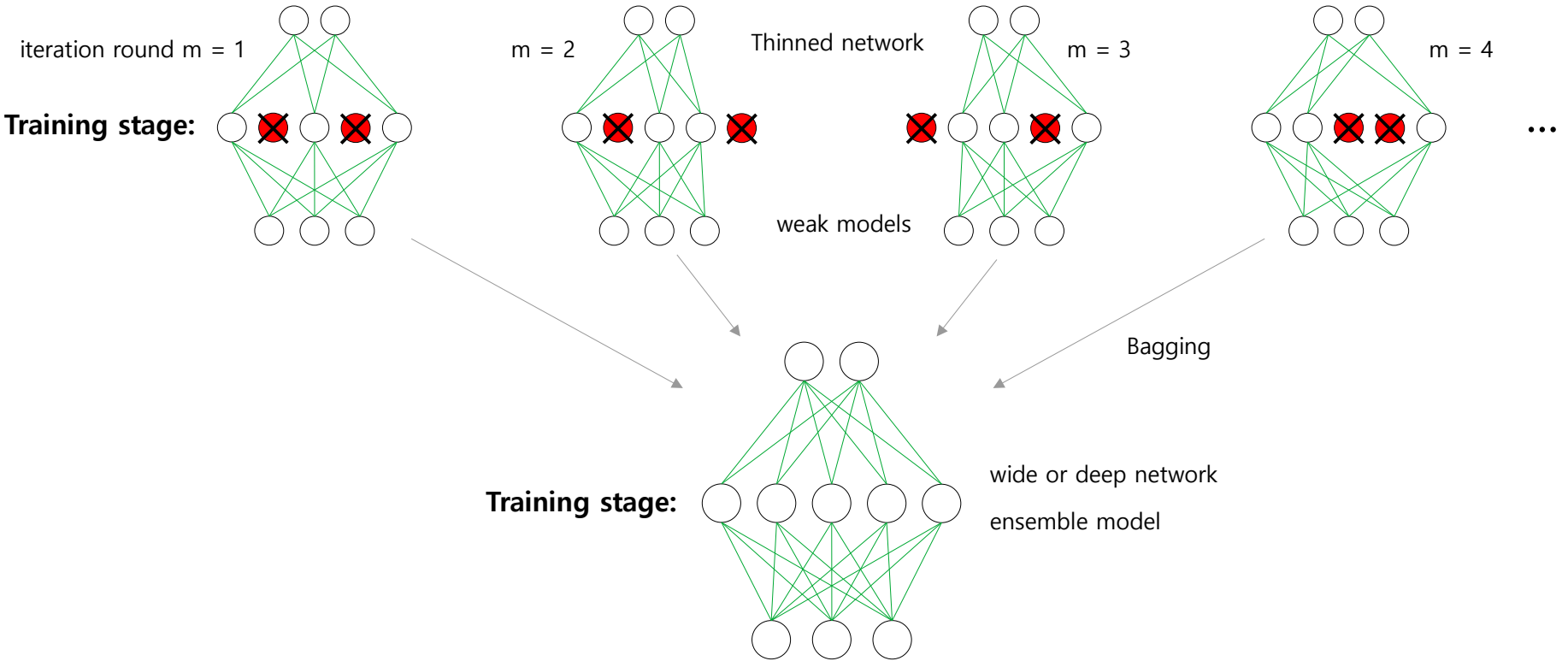
- Dropout is one of the ways to prevent overfitting. It works in two steps: zero-out and scaling.
- During the training process, some neurons in the hidden layer (or input layer) are randomly selected and the outputs of those neurons are set to 0. This is **zero-out** step. If the output of a neuron is 0, any weights connected to that neuron will not be updated during training.
- At each iteration, some neurons are selected and dropped, resulting in a thin network with those neurons excluded.
- In the prediction process, all neurons are used without dropout any neurons.
- Because the training and prediction processes work differently, it is necessary to scale the outputs of neurons. During the training process, we can make the outputs of neurons that were not selected larger, or we can make the weights smaller during the prediction process. The former is called the **scale-up** method, and the latter is called the **scale-down** method. Keras uses the scale-up method.



MX-AI

Dropout overview: Ensemble effect

- During the training process, it is similar to multiple small networks being trained independently, and during the prediction process, it is similar to the multiple small networks being combined into a single large network. This is similar to bagging in ensemble methods.
- This leads to ensemble effects, which reduce the variance of predictions and stabilize the model. This will help prevent overfitting.



MX-AI

■ Dropout paper

Journal of Machine Learning Research 15 (2014) 1929-1958 *Submitted 11/13; Published 6/14*

Dropout: A Simple Way to Prevent Neural Networks from Overfitting

Nitish Srivastava nitish@cs.toronto.edu
Geoffrey Hinton hinton@cs.toronto.edu
Alex Krizhevsky kriz@cs.toronto.edu
Ilya Sutskever ilya@cs.toronto.edu
Ruslan Salakhutdinov rsalakhu@cs.toronto.edu
Department of Computer Science, University of Toronto, 10 Kings College Road, Rm 3302, Toronto, Ontario, M5S 3G4, Canada.

Editor: Yoshua Bengio

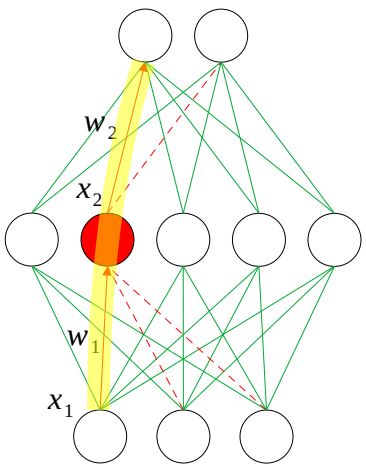
Abstract

Deep neural nets with a large number of parameters are very powerful machine learning systems. However, overfitting is a serious problem in such networks. Large networks are also slow to use, making it difficult to deal with overfitting by combining the predictions of many different large neural nets at test time. Dropout is a technique for addressing this problem. The key idea is to randomly drop units (along with their connections) from the neural network during training. This prevents units from co-adapting too much. **During training**, dropout samples from an exponential number of different "thinned" networks. At **test time**, it is easy to approximate the effect of averaging the predictions of all these thinned networks by simply using a single unthinned network that has smaller weights. This significantly reduces overfitting and gives major improvements over other regularization methods. We show that dropout improves the performance of neural networks on supervised learning tasks in vision, speech recognition, document classification and computational biology, obtaining state-of-the-art results on many benchmark data sets.

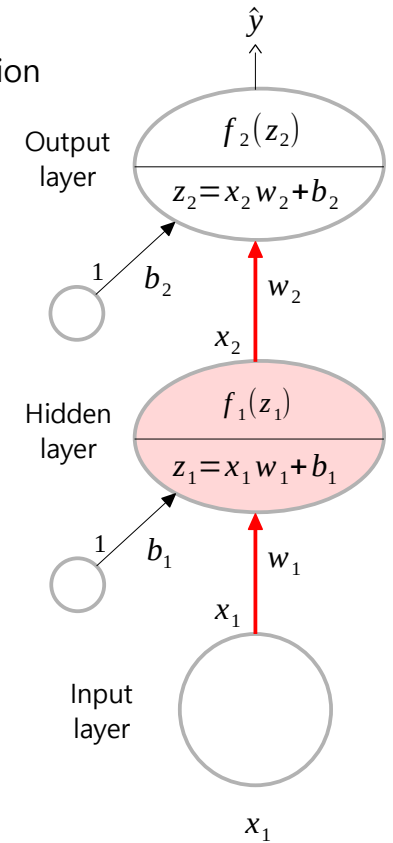
MX-AI

■ Zero-out & Error Backpropagation

- If you set the outputs of dropped neurons to 0, the weights connected to those neurons will not be updated.
- The reason is because of the backpropagation algorithm.



yellow path →



- If $x_2 = 0$, red w, w_1, w_2, etc. will not be updated.

■ Error Backpropagation

$L=\frac{1}{2}(\hat{y}-y)^2$: Loss function

$\hat{y}=f_2(z_2)=f_2(x_2w_2+b_2)$ $\quad x_2=f_1(z_1)=f_1(x_1w_1+b_1)$

(f_1, f_2 : activation function)

$w_2 \leftarrow w_2-\alpha\frac{\partial L}{\partial w_2}$: Gradient Descent

$w_2 \leftarrow w_2-\alpha\frac{\partial L}{\partial \hat{y}}\cdot\frac{\partial \hat{y}}{\partial z_2}\cdot\frac{\partial z_2}{\partial w_2}=w_2-\alpha\frac{\partial L}{\partial \hat{y}}\cdot\frac{\partial \hat{y}}{\partial z_2}\cdot x_2=w_2 \quad (if\ x_2=0)$

- w_2 is not updated.

$w_1 \leftarrow w_1-\alpha\frac{\partial L}{\partial w_1}$: Gradient Descent

$w_1 \leftarrow w_1-\alpha\frac{\partial L}{\partial \hat{y}}\cdot\frac{\partial \hat{y}}{\partial z_2}\cdot\frac{\partial z_2}{\partial x_2}\cdot\frac{\partial x_2}{\partial z_1}\cdot\frac{\partial z_1}{\partial w_1}=w_1 \quad (if\ x_2=0,\ \frac{\partial x_2}{\partial z_1}=0)$

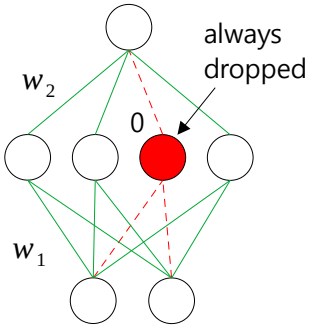
- w_1 is also not updated.
- For more information on error backpropagation, see the video [MXDL-3-01].

MX-AI

■ Zero-out verification code

```
# [MXDL-6-01] 1.zero_out.py
# If the output of a neuron is 0, we check that the weights
# connected to this neuron are not actually updated.
from tensorflow.keras.layers import Input, Dense, Multiply
from tensorflow.keras.models import Model
import numpy as np

x = np.array([[0, 0], [0, 1], [1, 0], [1, 1]], dtype=np.float32)
y = np.array([[0], [1], [1], [0]], dtype=np.float32)

# A mask that sets the output of the third neuron in the hidden
# layer to 0.
mask = np.array([[1., 1., 0., 1.]])

# Create a simple network
x_input = Input(batch_shape=(None, 2))
h1 = Dense(4, name='h1', activation='sigmoid')(x_input)
h2 = Multiply()([h1, mask])   # set the output of the third neuron
                              # to 0.
y_output = Dense(1, name='y', activation='sigmoid')(h2)

model = Model(x_input, y_output)
model.compile(loss='mse', optimizer='adam')
model_h1 = Model(x_input, h1)  # Model for checking the output h1.
model_h2 = Model(x_input, h2)  # Model for checking the output h2.

print('\n# weights of h1 before training: w1')
print(model.get_layer('h1').get_weights()[0])

print('\n# weights of y before training: w2')
print(model.get_layer('y').get_weights()[0])
```

```
# Training
print('\n----- train -----')
model.fit(x, y, epochs=100, verbose=0)

# Check the output h1.
print('\n# h1 output:\n')
print(model_h1.predict(x, verbose=0))

# Check the output h2.
# Check that the outputs of the third neuron are all 0.
print('\n# h2 output:\n')
print(model_h2.predict(x, verbose=0))

# After training, check the weights of the h1 layer.
# Compare with before training.
# Check if the 3rd column is the same as before training.
# Check if w's in the 3rd column are not updated.
print('\n# weights of h1 after training:')
print(model.get_layer('h1').get_weights()[0])

# After training, check the weights of the y layer.
# Compare with before training.
# Check if w's in the 3rd column are not updated.
print('\n# weights of y after training:')
print(model.get_layer('y').get_weights()[0])
```



MX-AI

Zero-out verification code

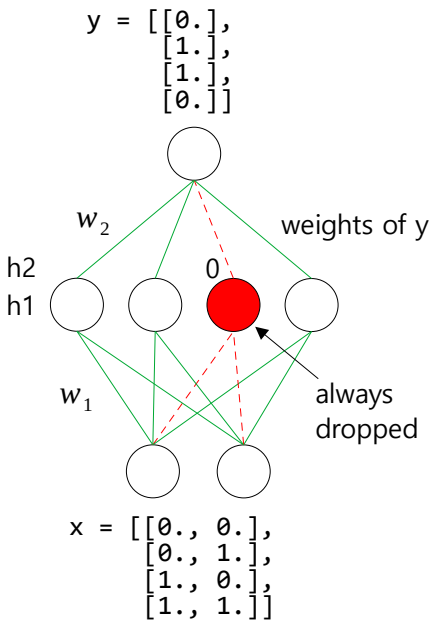
- If the output of a neuron is 0, we check that the weights connected to this neuron are not actually updated.

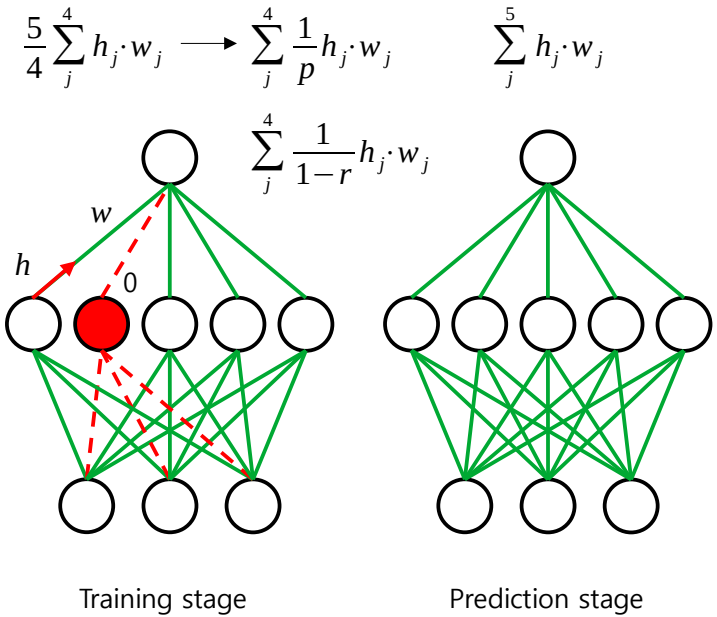
```
# weights of h1 before training: w1
[[-0.02797794  0.8613353  -0.7598376   0.1321249 ]
 [ 0.89024925  0.45562005  0.54993534  0.7180009 ]]

# weights of y before training: w2
[[-1.0356007 ]
 [-0.48115206]
 [-0.28035724]
 [ 0.36044633]]

----- train -----

# h1 output:

[[0.47742894 0.47743133 0.5        0.5239397 ]
 [0.6701091  0.5680519  0.63412064 0.71319747]
 [0.44813588 0.6638858  0.31868154 0.58102524]
 [0.643553   0.73979473 0.4477162  0.7580679 ]]

# h2 output: Force the 3rd column of h1 to be 0.

[[0.47742894 0.47743133 0.         0.5239397 ]
 [0.6701091  0.5680519  0.         0.71319747]
 [0.44813588 0.6638858  0.         0.58102524]
 [0.643553   0.73979473 0.         0.7580679 ]]
```

All values in the third column are 0.

```
# weights of h1 after training: w1
[[-0.1178598   0.770995   -0.7598376   0.2311513 ]
 [ 0.79902416  0.36424303  0.54993534  0.8151321 ]]

# weights of y after training: w2
[[-0.94482946]
 [-0.39023882]
 [-0.28035724]
 [ 0.4539777 ]]
```

Weights were not updated.





$$L_D = L_E + \frac{1}{2} p(1-p) \sum_{i=1}^{3} (w_i x_i)^2$$

6. Dropout

Part 2: Scale-down, Scale-up

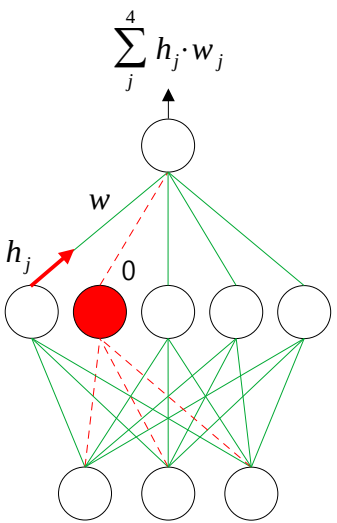
This video was produced in Korean and translated into English, and the audio was generated by AI (TTS).

www.youtube.com/@meanxai

Scale-down during prediction stage

- During training, some neurons in a layer are dropped out, and during prediction, all neurons are used, so the average output of the training and prediction processes do not match. So we need to adjust the average output.
- This can be done during training or during prediction. In the training stage, the outputs of that layer are scaled up, or in the prediction stage, the weights of that layer are scaled down.
- The example below shows how to scale down the weights during the prediction process.

Example: retention probability (p) = 4/5
dropout probability (1-p) = 1/5

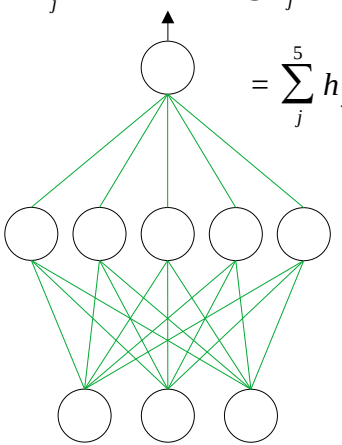


[Training stage]

correction

$$\sum_j^5 h_j \cdot w_j \rightarrow \frac{4}{5}\sum_j^5 h_j \cdot w_j$$

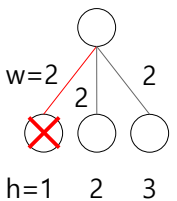
$$= \sum_j^5 h_j \cdot pw_j$$



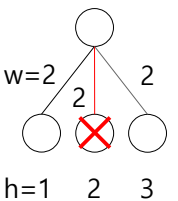
[Prediction stage]

At training time: The average output is 8.

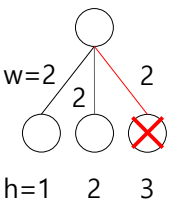
$\sum_j^2 h_j \cdot w_j \rightarrow$ 2*2+3*2=10



1*2+3*2=8

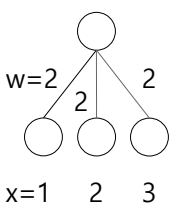


1*2+2*2=6



At test time: If you multiply the output by 2, you get 8.

$\sum_j^3 h_j \cdot w_j \rightarrow$ 2+4+6=12 → 12 * 2/3=8



Ensemble model
during prediction

MX-AI

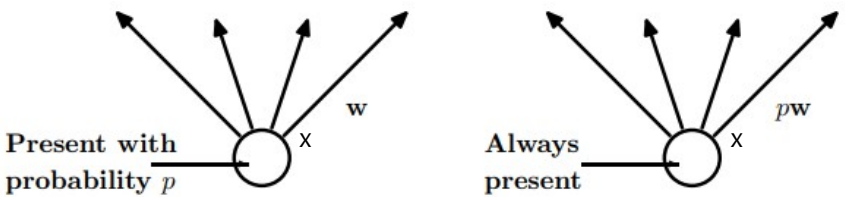
■ Scale-down during prediction stage

Reference:

[1] Geoffrey Hinton, Nitish Srivastava, et, al., 2012, Improving neural networks by preventing co-adaptation of feature detectors

[2] Nitish Srivastava, Geoffrey Hinton, et, al., 2014, Dropout: A Simple Way to Prevent Neural Networks from Overfitting

p : retention probability
x : output of the neuron
w : weights



(a) At training time (b) At test time

Figure 2: **Left**: A unit at training time that is present with probability p and is connected to units in the next layer with weights w. **Right**: At test time, the unit is always present and the weights are multiplied by p. The output at test time is same as the expected output at training time.

[2]

1. Introduction

...

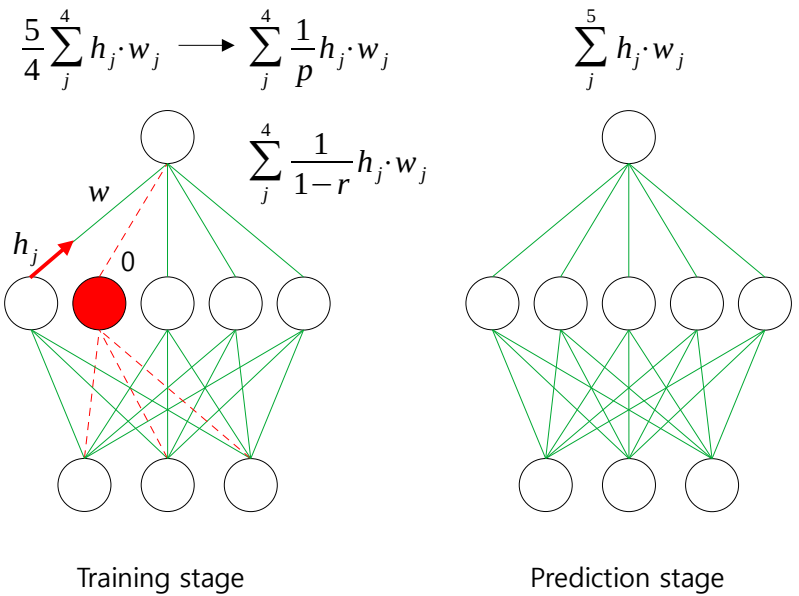
At test time, it is not feasible to explicitly average the predictions from exponentially many thinned models. However, a very simple approximate averaging method works well in practice. The idea is to use a single neural net at test time without dropout. The weights of this network are **scaled-down** versions of the trained weights. If a unit is retained with probability p during training, the outgoing weights of that unit are multiplied by p at test time as shown in Figure 2. This ensures that for any hidden unit the expected output (under the distribution used to drop units at training time) is the same as the actual output at test time.

MX-AI

■ Scale-up during training stage

- The example below shows how to scale up the outputs of the layer during the training process.
- Keras uses this method.

Example: dropout probability (1-p) = 1/5 = r
retention probability (p) = 4/5



Dropout layer

(keras Dropout document)

```
tf.keras.layers.Dropout(rate, noise_shape=None, seed=None, **kwargs)
```

The Dropout layer randomly sets input units to 0 with a frequency of rate at each step during training time, which helps prevent overfitting. Inputs not set to 0 are **scaled up** by 1/(1 - rate) such that the sum over all inputs is unchanged.

Note that the Dropout layer only applies when training is set to True such that no values are dropped during inference. When using model.fit, training will be appropriately set to True automatically, and in other contexts, you can set the kwarg explicitly to True when calling the layer.

(This is in contrast to setting trainable=False for a Dropout layer. trainable does not affect the layer's behavior, as Dropout does not have any variables/weights that can be frozen during training.)

```
Example:
x = Dense(3)(x_data)          # input layer
h1 = Dense(4)(x)              # hidden layer
h2 = Dropout(rate=0.5)(h1)    # dropout layer
y_hat = Dense(1)(h2)          # output layer
```

MX-AI

■ Scale-up verification code $dropout\ rate(r)=0.5,\ \frac{1}{1-0.5}=2$

```
# [MXDL-6-02] 2.scale_up.py
# Check the scale-up results of dropout in Keras.
from tensorflow.keras.layers import Input, Dense, Dropout
from tensorflow.keras.models import Model
from tensorflow.keras import initializers
from tensorflow.keras.optimizers import Adam
import numpy as np

x = np.array([[1, 1]], dtype=np.float32)   # input
y = np.array([[0.01]], dtype=np.float32)   # desired output

x_input = Input(batch_shape=(None, 2))
h1 = Dense(4, name='h1')(x_input)
h2 = Dropout(rate = 0.5, name='h2')(h1)
y_output = Dense(1, name='y')(h2)

model = Model(x_input, y_output)
model.compile(loss='mse', optimizer=Adam(learning_rate=0.1))
model.summary()

model_h1 = Model(x_input, h1)
model_h2 = Model(x_input, h2)

# Training
model.fit(x, y)

# before Dropout
h1_output = model_h1(x)
print('\nbefore Dropout (h1):\n', h1_output.numpy().round(4))

# after Dropout
h2_output = model_h2(x, training=True)
print('\nafter Dropout (h2):\n', h2_output.numpy().round(4))
```

```
First run:
before Dropout (h1):
 [[-0.013   0.8914 -0.5011  1.3449]]

after Dropout (h2):
 [[-0.      1.7829 -1.0023  0.    ]]
```

scale-up by 2

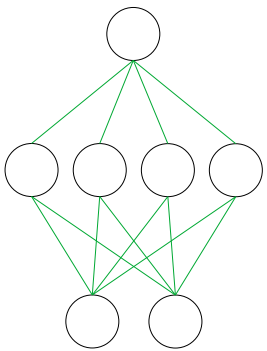
zero-out

```
Second run:
before Dropout (h1):
 [[ 1.1716  1.0362  0.2058 -0.383 ]]

after Dropout (h2):
 [[ 2.3432  0.      0.     -0.766 ]]
```

scale-up by 2

```
Third run:
before Dropout (h1):
 [[-0.2437 -0.589   0.0023  0.5428]]

after Dropout (h2):
 [[-0.4873 -0.      0.      0.    ]]
```

zero-out



The dropout rate of 0.5 is probabilistic, so it does not necessarily mean that 2 will be dropped out. However, the scale-up still applies by a factor of 2.

MX-AI

■ Optimal dropout rate

Reference:

[2] Nitish Srivastava, Geoffrey Hinton, et, al., 2014, Dropout: A Simple Way to Prevent Neural Networks from Overfitting

A.4 Dropout Rate

Dropout introduces an extra hyperparameter—the probability of retaining a unit p. This hyperparameter controls the intensity of dropout. p = 1, implies no dropout and low values of p mean more dropout. Typical values of p for hidden units are in the range 0.5 to 0.8. For input layers, the choice depends on the kind of input. For real-valued inputs (image patches or speech frames), a typical value is 0.8. For hidden layers, the choice of p is coupled with the choice of number of hidden units n. Smaller p requires big n which slows down the training and leads to underfitting. Large p may not produce enough dropout to prevent overfitting.

MX-AI

■ Dropout and Regularization

- [3] Pierre Baldi, Peter Sadowski, 2014, The dropout learning algorithm
- Applying dropout is equivalent to applying regularization to a network without dropout.
- The regularization constant is highest when p is 0.5.

δ : Bernoulli random variable [0, 1] – dropped or not

p : retention probability $\quad P(\delta_i=1)=p$

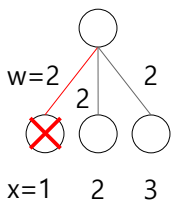
$E[\delta]=p, \quad Var[\delta]=p(1-p)$

[Example]

$x_i=[1,2,3] \quad w_i=[2,2,2]$

$\delta_i=[0,1,1] \quad p=\frac{2}{3}$

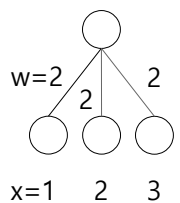
Dropout models during training



$y_D=\sum_{i=1}^{3}\delta_i w_i x_i$

= 0*2*1 + 1*2*2 + 1*2*3 = 10

Ensemble model during prediction



$y_E=\sum_{i=1}^{3} p\, w_i x_i$

= (2/3)*2*1 + (2/3)*2*2 + (2/3)*2*3 = 8

11. Dropout Dynamics

11.2. Dropout gradient and adaptive regularization: single linear unit

$$\begin{cases} L_D=\frac{1}{2}(y-y_D)^2 \\ L_D=\frac{1}{2}(y-\sum_{i=1}^{3}\delta_i w_i x_i)^2 \end{cases} \qquad \begin{aligned} \frac{\partial L_D}{\partial w_i} &= -y\,\delta_i x_i+(\sum_{i=1}^{3}\delta_i w_i x_i)\delta_i x_i \\ &= -y\,\delta_i x_i+w_i\delta_i^2 x_i^2+\sum_{j=1,j\neq i}^{3} w_j\delta_i\delta_j x_i x_j \end{aligned}$$

$$\begin{cases} L_E=\frac{1}{2}(y-y_E)^2 \\ L_E=\frac{1}{2}(y-\sum_{i=1}^{3} p\,w_i x_i)^2 \end{cases} \qquad \begin{aligned} \frac{\partial L_E}{\partial w_i} &= -y\,p\,x_i+(\sum_{i=1}^{3} p\,w_i x_i)\,p\,x_i \\ &= -y\,p\,x_i+w_i p^2 x_i^2+\sum_{j=1,j\neq i}^{3} w_j\,p\,p\,x_i x_j \end{aligned}$$

$$\begin{aligned} E[\frac{\partial L_D}{\partial w_i}] &= -y\,E[\delta_i]x_i+w_i E[\delta_i^2]x_i^2+\sum_{j=1,j\neq i}^{3} w_j E[\delta_i]E[\delta_j]x_i x_j \\ &= -y\,p\,x_i+w_i p^2 x_i^2+w_i Var[\delta_i]x_i^2+\sum_{j=1,j\neq i}^{3} w_j\,p\,p\,x_i x_j \\ &= \frac{\partial L_E}{\partial w_i}+w_i Var[\delta_i]x_i^2=\frac{\partial L_E}{\partial w_i}+w_i p(1-p)x_i^2 \end{aligned}$$

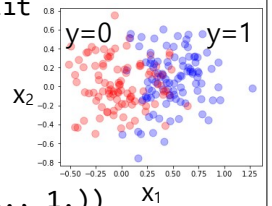
$$L_D = L_E + \frac{1}{2}p(1-p)\sum_{i=1}^{3}(w_i x_i)^2$$

This is the regularized loss for ensemble model.

MX-AI

- Implement a neural network model with dropout using Keras

```
# [MXDL-6-02] 3.dropout.py
import numpy as np
from tensorflow.keras.layers import Input, Dense, Dropout
from tensorflow.keras.models import Model
from tensorflow.keras import optimizers
from sklearn.datasets import make_blobs
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt

# Create a dataset for binary classification
x, y = make_blobs(n_samples=200, n_features=2,
                  centers=[[0., 0.], [0.5, 0.1]],
                  cluster_std=0.25, center_box=(-1., 1.))
```

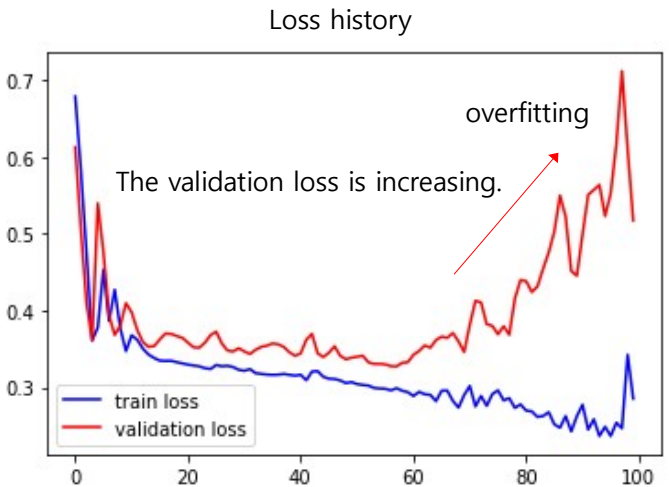



```
# Visually see the data
plt.figure(figsize=(5,4))
color = [['red', 'blue'][a] for a in y.reshape(-1,)]
plt.scatter(x[:, 0], x[:, 1], s=100, c=color, alpha=0.3)
plt.show()

# Split the dataset into the training and test data.
x_train, x_test, y_train, y_test = train_test_split(x, y)

# Create a model.
n_input = x.shape[1]  # number of input neurons
n_output = 1          # number of output neurons
n_hidden = 64         # number of hidden neurons
d_rate = 0.5          # dropout rate
adam = optimizers.Adam(learning_rate=0.01)

x_input = Input(batch_shape=(None, n_input))
h = Dense(n_hidden, activation = 'relu')(x_input)
h = Dropout(rate=d_rate)(h)

# Additional 3 hidden layers
# The data is simple, but we intentionally added many hidden
# layers to the model to demonstrate the effect of regularization.
for i in range(3):
    h = Dense(n_hidden, activation='relu')(h)
    h = Dropout(rate=d_rate)(h)

y_output = Dense(n_output, activation='sigmoid')(h)
model = Model(x_input, y_output)
model.compile(loss='binary_crossentropy', optimizer=adam)

# Training
h = model.fit(x_train, y_train,
              validation_data=(x_test, y_test),
              epochs=100, batch_size=100)

# Visually see the loss history
plt.plot(h.history['loss'], c='blue', label='train loss')
plt.plot(h.history['val_loss'], c='red', label='validation loss')
plt.legend()
plt.show()

# Check the accuracy of the test data.
y_pred = (model.predict(x_test) > 0.5) * 1
acc = (y_pred.reshape(-1,) == y_test).mean()
print("\nAccuracy of the test data = {:4f}".format(acc))
```

MX-AI

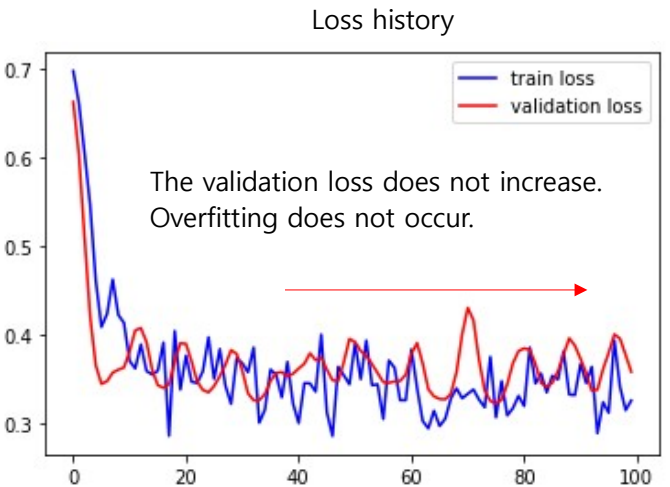
- Implement a neural network model with dropout using Keras

1. Without dropout



Accuracy of the test data = 0.78

2. With dropout



Accuracy of the test data = 0.86