ML Report

B05502087王竑睿 B06902104吳由由 B06902116高為勳

Data preprocessing:

* Feature selection:

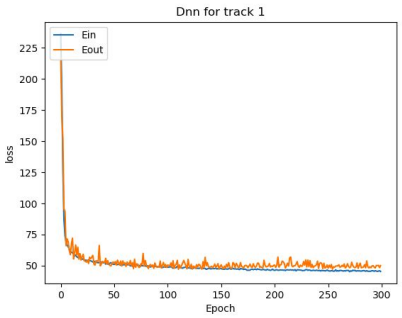
  Use sklearn SelectKBest, f_regression find the best 5000 feature for each y_i and use the joint collection of features for the rest of the process.

* Normalize the dataset by sklearn.preprocessing.StandardScaler()
* Random shuffle training dataset
* Use 44500 data for training and 3000 data for testing

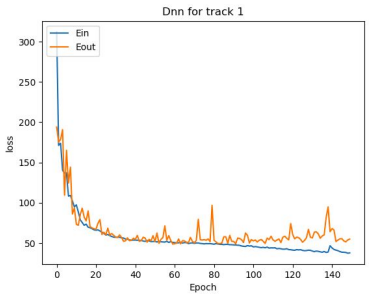
DNN Architecture:

```
=================================================================
Input layer
BatchNormalization
Dropout(0.01)
Dense(1024) activation function is 'relu'.
BatchNormalization
Dropout(0.01)
Dense(1024) activation function is 'relu'.
BatchNormalization
Output layer
=================================================================
```

For track1



Hidden layer:
1000-1000
Ein /Eout:
47.0215/51.4411
epoch=40

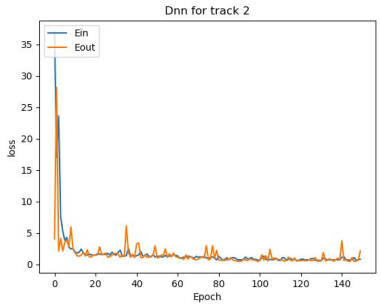


Hidden layer:
50-50
Ein/Eout
37.8391/55.0103
epoch=50

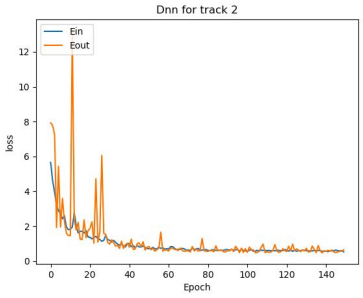
Conclusion:

複雜的模型很容易讓track 1 overfit. 雖然Ein可以表現得很好，但是Eout反而都降不下來
簡單的模型Ein會卡住降不下來，但相對Eout的表現有比較好

For track 2



Hidden layer:
1000-1000
Ein /Eout:
0.74/0.69
epoch=20



Hidden layer:
50-50
Ein/Eout
0.52/0.64
epoch=60

Conclusion:

在實驗中，track2在網路複雜時的overfit情況沒有像track 1嚴重，而且複雜的網路訓練出的結果比較穩定，不像簡單的網路有很嚴重的上下震盪，收斂的epoch也比較少。在track 2中用複雜一點的架構可以達到比較穩定且不錯的結果。

Early stop:

在我們的觀察中發現validation loss 的起伏很嚴重， 所以在我們的validation loss 小於自己定義的threshold時就讓訓練停下來

CNN Architecture:

Data preprocessing:

Training data x 原本是1*10000的feature, 將他們reshape成100*100 的2D array, 給conv2D做訓練

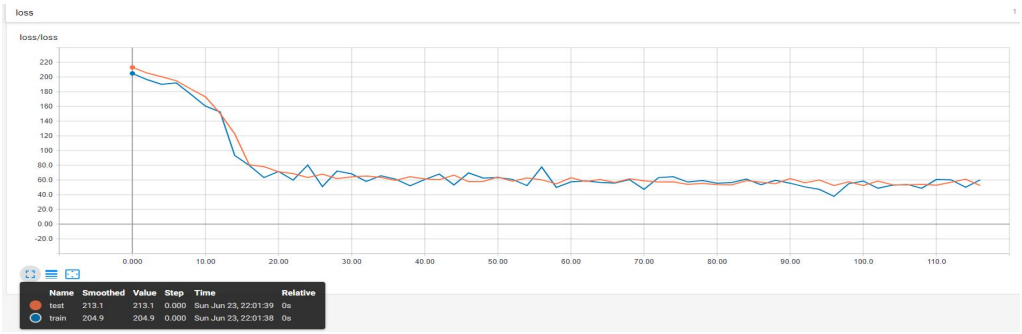
```
==========================================================================
Input layer
Conv2D(32,(3,3),input_shape=(100,100,1),activation='relu')
MaxPooling2D(pool_size=(2,2))
Conv2D(32,(3,3),activation='relu')
MaxPooling2D(pool_size=(2,2))
Flatten()
BatchNormalization()
Dense(units=128,activation='relu')
BatchNormalization()
Dense(units=64,activation='relu')
BatchNormalization()
model.add(Dense(units=3)) ## Output layer
==========================================================================
```

達到收斂所需要的epoch

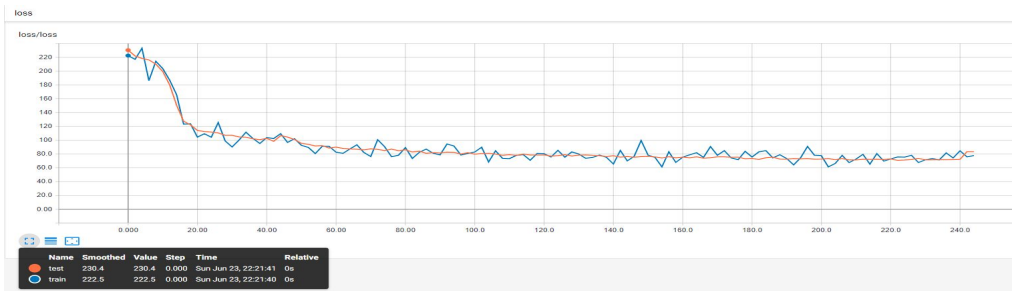
For large learning rate(like 0.0287): 150 to 200 (epoches)

For small learning rate(like 0.0087): 400 to 500 (epoches)

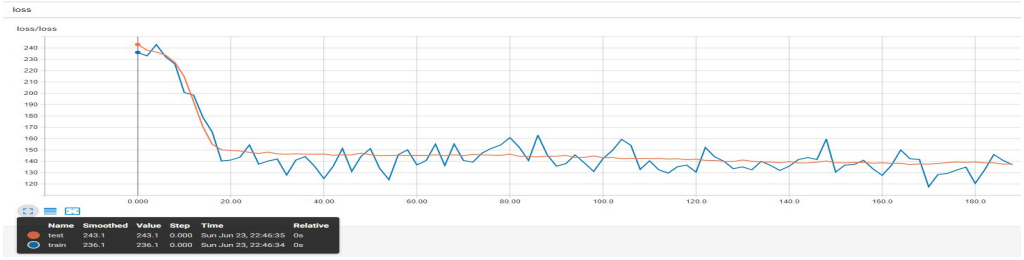
不同dopout 參數下跑出來的



dropout=0.87:
Ein: 47.096
Eout: 51.76229, epoch: 148



dropout=0.1:
Ein: 69.987
Eout:70.888,epoch = 249



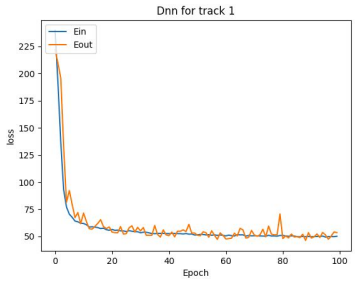
dropout=0.01:
Ein:135.195
Eout: 137.442,epoch = 189

Feature selection:

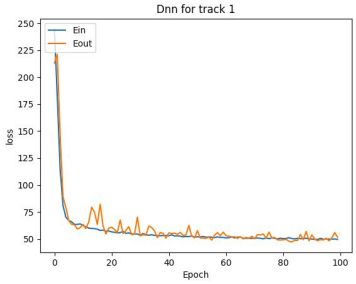
Approach:

利用sklearn 的SelectKBest, f_regression在10000維feature中找出分別對mesh size, penetration rate, alpha最重要的前kbest個column. 將這些column聯集起來作為我們select出來的feature來建traing data

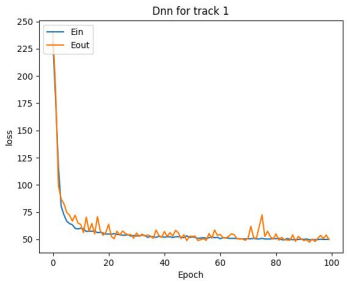
Kbest=9000,epoch=40
Ein / Eout= 50.1254/53.4927



Kbest=5000,epoch=42
Ein / Eout= 49.8425 / 48.2080



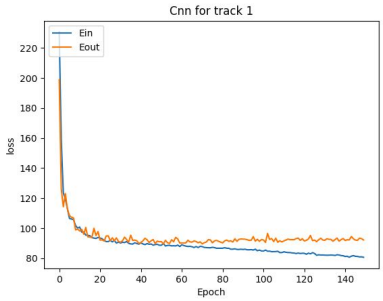
Kbest=1000, epoch=25
Ein / Eout= 50.1230 / 49.7543



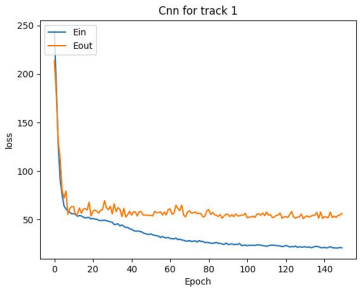
Conclusion:

我們發現當kbest=9000時雖然保留下來的feature很多，但Ein Eout的表現並不特別好，有可能是因為總資料量不夠多，若有更多資料才有可能好好運用這麼多feature. 當選擇kbest=5000時，Ein/ Eout 達到了最佳值，成功的讓我們的model Eout值可以掉到47以下。比較讓人驚訝的發現是就算我們選擇的kbest=1000, 相當於在總feature中只取了不到1/3的量來訓練，仍然達到了很不錯的Ein/Eout 結果，甚至比kbest=9000的狀況下還好，我們可以合理推測10000維feature之中，可能有很多東西對這個training來說是很不必要的。Feature selection是我們這次突破的其中一個很重要的方法

Batch Normalization:



Without batch normalization
Ein /Eout:
80.6184/92.1484
epoch=25

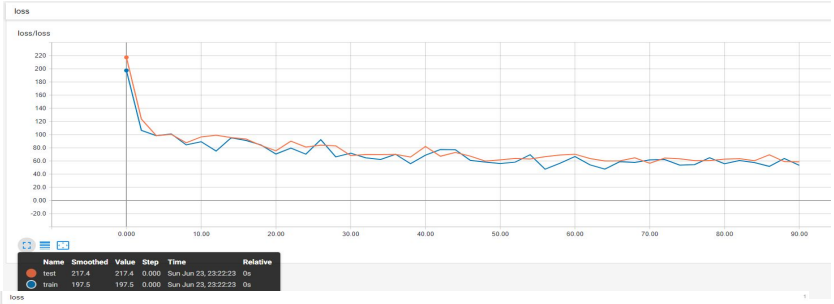


With batch normalization
Ein/Eout
20.9939/56.4534
epoch=15

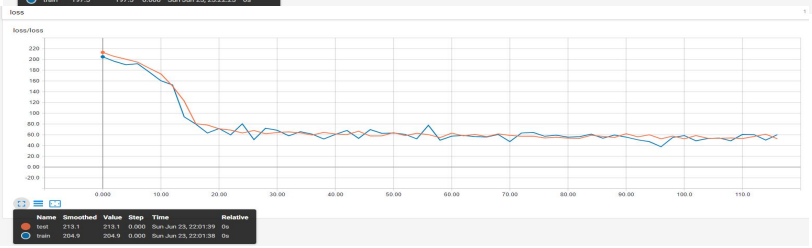
Conclusion:

加上batch normalization後達到收斂所需的epoch比較小，且Ein Eout都明顯比沒做batch normalization下降，是個可以大幅增加正確率與效率的方法

不同learning rate下達到的結果

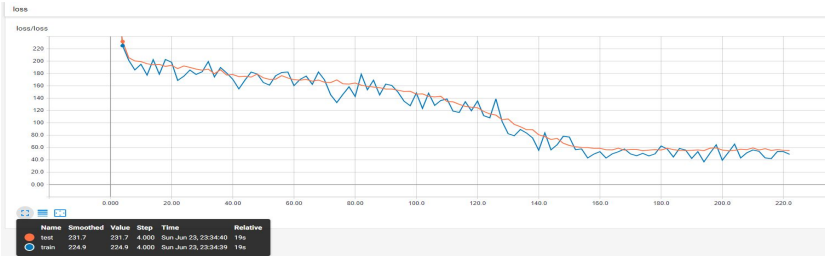


lr=0.1:
ein/eout = 54.307/56.702,
epoch = 92



lr=0.01:

ein/eout = 47.096/51.76229,
epoch = 14



lr=0.001:
ein/eout = 40.787/54.156,
epoch = 224

Conclusion:

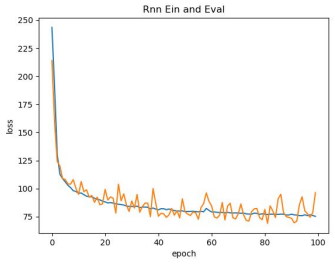
當learing rate逐漸減小時，因為每一步都走得比較小，所以學習時比較容易能夠讓ein下降，但也有可能因此陷入局部最佳解，使得eout因此而變大。除此之外，learning rate越小時，訓練的時間也會相對的變長許多，如上面的圖的結果便可得知learning對訓練效率的影響有多大，故選擇良好的learning rate會是在調整參數時一個很重要的一步。

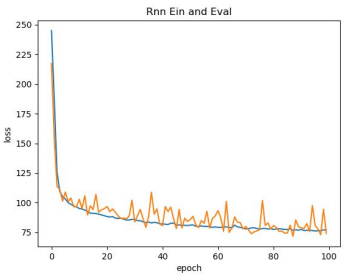
Rnn LSTM Architecture

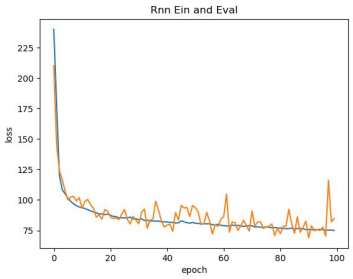
```
==========================================================================================
model=Sequential()
model.add(BatchNormalization(input_shape=(5000,1)))
model.add(Dropout(0.02))
model.add(LSTM(units=100,return_sequences=True,input_shape=(5000,1),activation="relu"))
model.add(BatchNormalization())
model.add(Dropout(0.02))
model.add(LSTM(units=100,return_sequences=True,activation="relu"))
model.add(BatchNormalization())
model.add(Dropout(0.02))
model.add(LSTM(units=50,activation="relu"))
model.add(BatchNormalization())
model.add(Dropout(0.02))
model.add(Dense(128, activation="relu"))
model.add(BatchNormalization())
model.add(Dropout(0.02))
model.add(Dense(64, activation="relu"))
model.add(BatchNormalization())
model.add(Dropout(0.02))
model.add(Dense(3))
==========================================================================================
```

Track1 :

1. **1 LSTM layer / 2 LSTM layers / 4 LSTM layers**

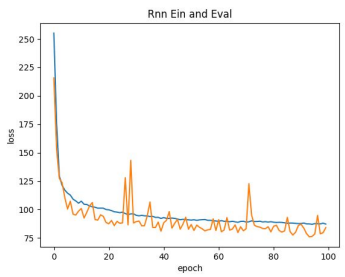




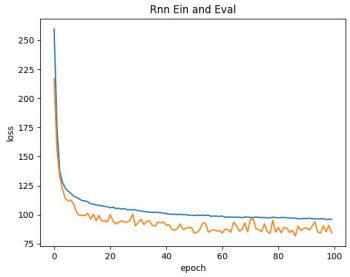


Ein=75.2043 Eout=96.3975 Ein=75.0032 Eout=74.9277 Ein=76.9793 Eout=74.0888

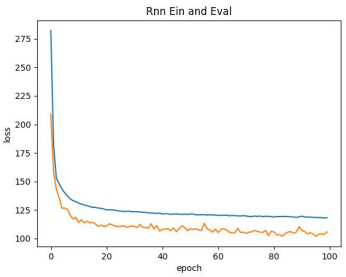
2. **Dropout with 0.1 / 0.2 / 0.5**



Ein=87.2894 Eout=76.0657



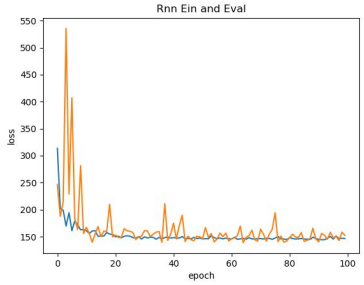
Ein=95.9767 Eout=84.2670



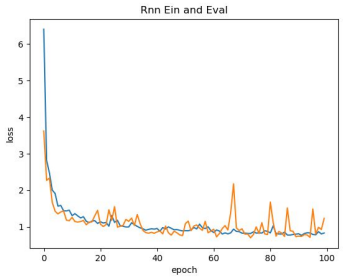
Ein=118.2302 Eout=105.7420

Track2 :

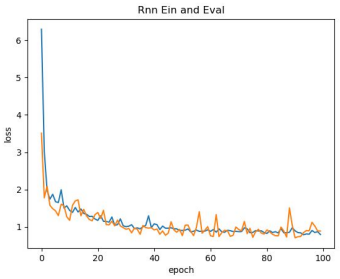
1. **1 LSTM layer / 2 LSTM layers / 3 LSTM layers**



Ein=154.3734 Eout=161.2673

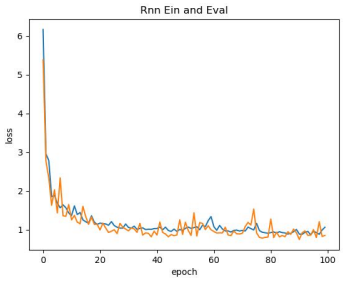


Ein=0.9674 Eout =1.5302

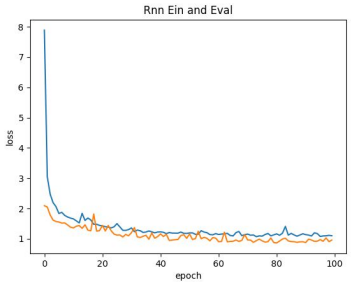


Ein=0.9534 Eout=1.0528

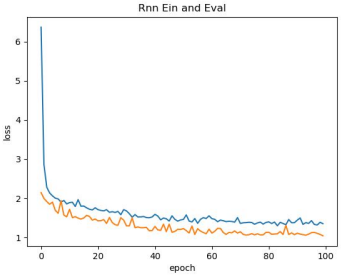
2. **Dropout with 0.1 / 0.2 / 0.5**



Ein=0.8302 Eout=0.9656

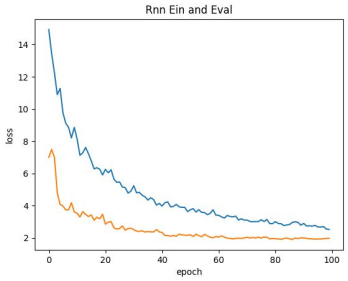


Ein=1.1150 Eout=0.9074

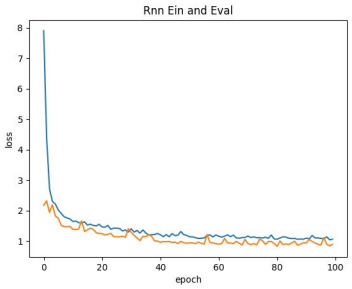


Ein=1.3534 Eval=1.0436

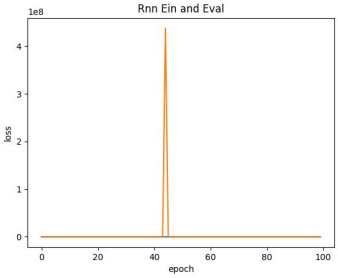
3. **learning rate 0.00005 / 0.005 / 0.5**



Ein=2.5244 Eout=1.9759



Ein=1.0660 Eout=0.8958



Ein=323.1586 Eout=437607890.3216

Conclusion

1. Layers :

由圖可知從一層LSTM layer增加到兩層時Ein和Eout有非常明顯地驟降。可能是因為本次data的時序性feature在兩層LSTM時才能夠extract出來。

2. Dropout :

由圖可知 Dropout 可以對於overfitting起到相當的克制，因為隨機停掉某些神經元的作用可以防止模型記憶訓練資料。但當Dropout的比率設定過高時，則會導致Ein和Eval都偏高。這很可能是因為Drop掉過多神經元時會導致資訊捨棄過多，使得訓練的效果較差。

3. Learning Rate :

由圖可知，learning rate 過小會使得更新的速率不夠快，致使在同樣的epoch數量下，最終結果underestimate。但當learning rate過大時，則可能會導致函數直接跳過某些local minimum而進到error相當大的區域，產生如上右圖過大的Eout。

Comparison

1.efficiency

DNN: 和model 的architecture有關，層數越多神經元越多需要的時間越多, learning rate 越小花的時間越多

CNN: 和window的大小，stride的步長，filter的大小與model的複雜度有關, learning rate 越小花的時間越多

RNN:使用unroll時犧牲space efficiency換取time efficiency。不使用unroll時則反之。

2.scalability

DNN>CNN=RNN

(i) DNN 本身是fully connected的，能更新的weight較多，自由度較大 所以通常對data都能夠起到一定的fit能力

(ii)CNN 使用一個windows捕捉相鄰資料之間的關係，如同本次資料後5000維相鄰100維是不同的實驗計算出的velocity coefficient，因此將10000x1 的 data reshape成100x100之後，windows能夠捕捉到相鄰實驗之間的關係。

(iii)RNN 本身能夠捕捉 data 的時序關係。透過觀察資料中的時序關係將data reshape成不同的(sample, step, feature)，能夠捕捉到data中以含有step個元素的time sequence的特性。但是在本次實驗中資料之間時序關係不明顯，RNN的表現就相對不好。

3.interpretability

RNN=CNN>DNN

RNN 可以找出feature間的時間關係，對前5000筆訓練資料而言，使用RNN可以捕捉其中的關係

CNN 我們用100*100的window的方式找出附近關聯，對後5000維50*100的實驗相當於是每個實驗的前後幾筆資料一起做training，應該可以找出些有用的feature與關聯

DNN 設計出的架構和實際上預測出來的東西不太具有物理關係，所以interpretability較低

4.ability:

RNN雖然能做到DNN、CNN做不到的對時間的反應，但它需要訓練的次數多且容易產生梯度消失問題，最重要的是，它不具有如DNN、CNN的特徵學習能力(從上面的實驗表現可得知)，因此RNN是專門解決時間序列問題的網路；而CNN則是專門解決影像問題，對一塊塊區域做特徵提取的網絡，以區域性特徵建構出整體特徵。

Final recommendation:

For track 1:

用全層連接的簡單淺層DNN來為這次的dataset 做regression是最好的approach, 讓network的每層間會更新出最合適的weight，能得到最好的結果 (Hidden layer: 50-50 in our case)

For track 2:

用全層連接的複雜DNN來預測track 2是最好的approach, track 2在複雜模型比較少overfit的情況，複雜模型比較能得出穩定的結果(Hidden layer: 1024-1024-12 in our case)

DNN Pros:

Dnn 網路中我們只需要設計架構把資料給進去後，就算不了解資料或神經連接的意義，只要神經元數夠多、資料夠多，就會有不錯的結果。在實驗中我們發現model的正確率和training data的資料量成正相關，資料越多時表現越好。所以Dnn model 在只有資料跟結果不知道其中意義的task中是個很不錯的model,因為架構建起來很簡單，又可正確預測出接近的結果，所以是一種很受歡迎的regression方式。

DNN Cons:

當Dnn的神經元數太少或層數不夠多時，Ein Eout都會降不下來。 當Dnn 的神經元的數目太多時，model會overfit得很嚴重。Dnn疊的太深時可能會梯度消失，以至於weight無法正確更新。神經元多資料量又足夠大時，training時間要花很久，還需要很多的運算資源。 找不到系統性調Dnn參數的方式，感覺很像在碰運氣，表現的好或不好都不知道理由，有時候花一個下午調參數比不過隔天的隨手一按。

Workload balance

B06902104吳由由: 實驗CNN與DNN，調整各種network architecture與參數對結果的影響, Early stop實做
B06902116高為勳: 實驗DNN，調整參數，找尋降低Eout的方法(例如feature selection及batch normalization實作)
B05502087王竑睿: 實驗RNN與DNN，tune model以及研究防止overfitting的方法(Dropout)