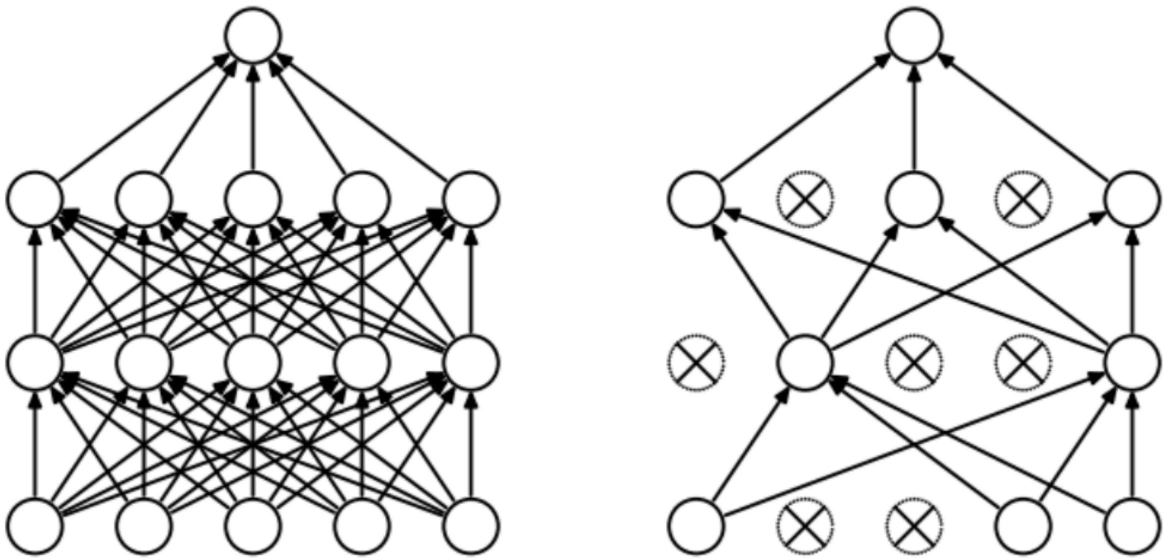
# Dropouts in ANN

`from keras.` `layers` `import` `Dropout`

`Dropout(0.3)`



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

- **Dropout** is a powerful regularization method that **randomly deactivates neurons** during training to prevent overfitting.
- It is especially useful when the model has a large number of parameters and could overfit the training data.

## Core Concept:

- During the training process, dropout randomly "drops" or deactivates a certain percentage of neurons. This means that these neurons are temporarily excluded from both the forward pass and the backward pass of the network.
- This process is repeated for each training batch, so different sets of neurons are deactivated each time.
- During the inference or testing phase, all neurons are active, but their outputs are typically scaled down to compensate for the fact that they were less active

during training.

## Dropout Rate:

- The "dropout rate" is a hyperparameter that determines the percentage of neurons to be dropped.
- Common values are between 0.2 and 0.5.

## Training vs. Inference:

- **Dropout is only applied during the training phase.**
- During inference, the entire network is used.
    - For testing we multiply weights by $(1 - p)$
    - $p$ = dropout rate

# Python code:

```
model = Sequential([
    Dense(128, activation='relu', input_dim=784),  # Input layer (28×28 flattened to 784)
    Dropout(0.3),  # Drop 30% of the neurons in the first hidden layer

    Dense(64, activation='relu'),
    Dropout(0.3),  # Drop 30% of the neurons in the second hidden layer

    Dense(10, activation='softmax')  # Output layer (10 classes for MNIST)
])
```

## ⚡ Pro Tips

1. **Combine with L2 Regularization**:

```
Dense(64, activation='relu', kernel_regularizer=l2(0.01))
```

2. **Adjust Rate Based on Layer Size**:
   - Higher dropout for **larger layers** (e.g., `0.5` for 512 neurons).
   - Lower dropout for **small layers** (e.g., `0.2` for 64 neurons).
3. **Monitor Training Curves**:
   - If **training loss >> validation loss**, increase dropout.
   - If **both losses are high**, reduce dropout.
4. Initially, **test this on the last layer**. If you see any results, you can try it on remaining hidden layers.
5. CNN → 0.4 to 0.5

   RNN → 0.2 to 0.3

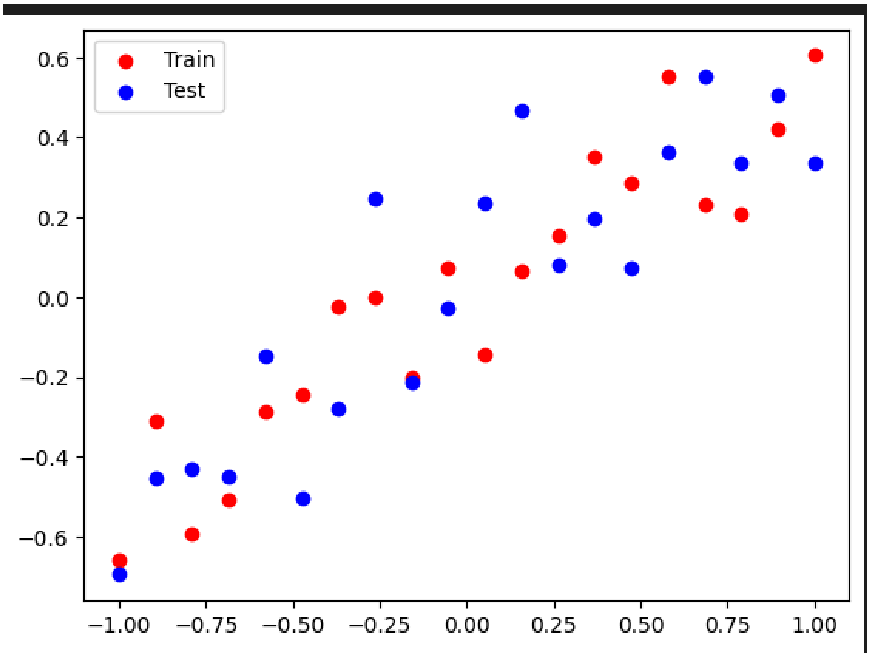
   ANN → 0.1 to 0.5

# Drawbacks

- Delay in convergence
  - Slow training
- Value of loss function changes
  - So, it becomes difficult to debug the gradients

# Regression Example

**Data:**




## Without Dropout:

```
model_1 = Sequential()
model_1.add(Dense(128, input_dim=1, activation="relu"))
model_1.add(Dense(128, activation="relu"))
model_1.add(Dense(1, activation="linear"))
adam = Adam(learning_rate=0.01)
model_1.compile(loss='mse', optimizer=adam, metrics=['mse'])
history = model_1.fit(X_train, y_train, epochs=500,
                validation_data = (X_test, y_test),
                verbose=False)
```

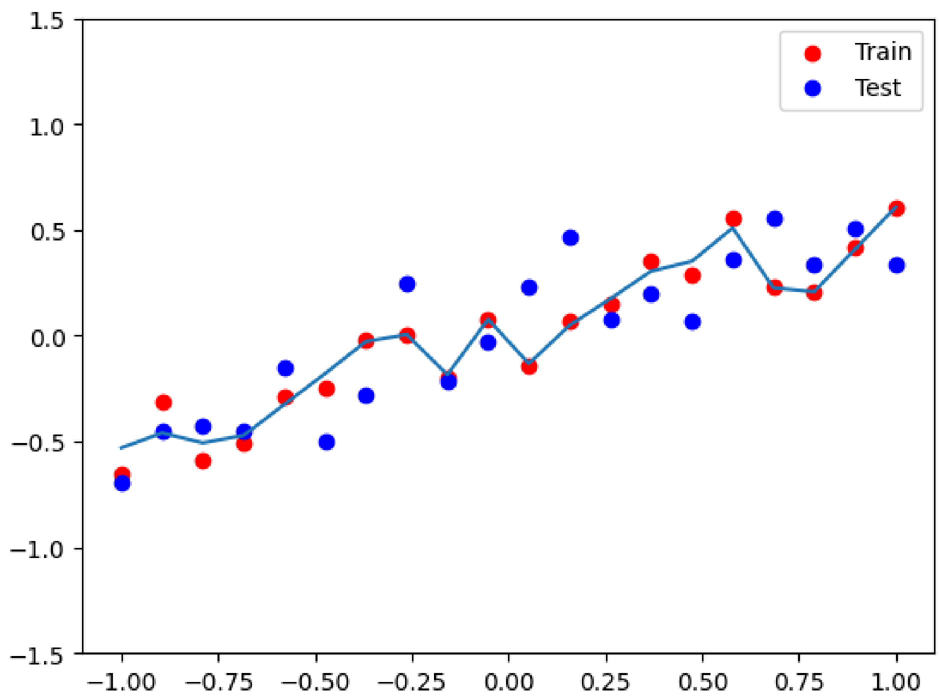
```
# evaluate the model
_, train_mse = model_1.evaluate(X_train, y_train, verbose=0)
_, test_mse = model_1.evaluate(X_test, y_test, verbose=0)
print('Train: {}, Test: {}'.format(train_mse, test_mse))
```

```
Train: 0.003131699515506625, Test: 0.046952348202466965
```

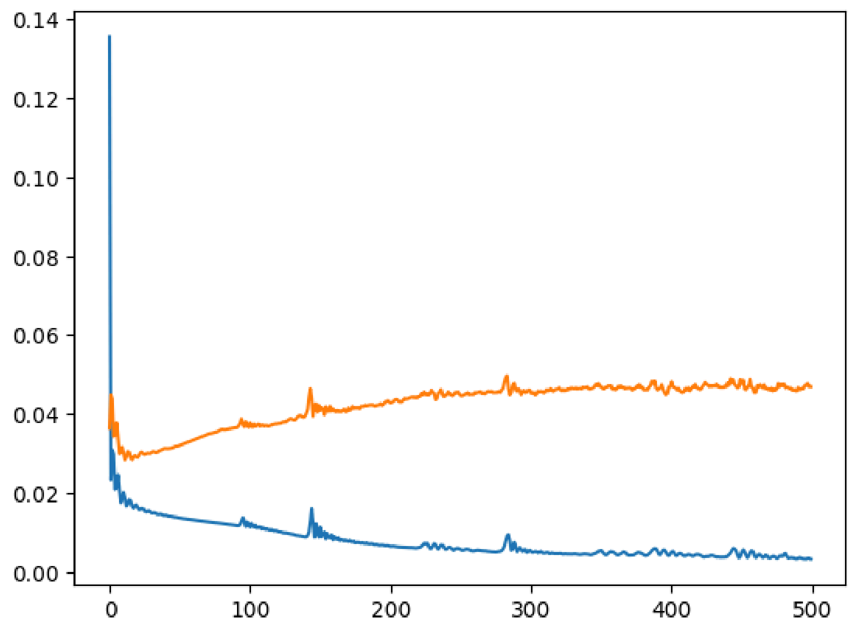
- More than **10x difference** between training & testing loss
    - Indicates overfitting

```
y_pred_1 = model_1.predict(X_test)

plt.figure()
plt.scatter(X_train, y_train, c='red', label='Train')
plt.scatter(X_test, y_test, c='blue', label='Test')
plt.plot(X_test, y_pred_1)
plt.legend()
plt.ylim((-1.5, 1.5))
plt.show()
```




```
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
```




## With Dropout:

```
model_2 = Sequential()
model_2.add(Dense(128, input_dim=1, activation="relu"))
model_2.add(Dropout(0.2))
model_2.add(Dense(128, activation="relu"))
model_2.add(Dropout(0.2))
model_2.add(Dense(1, activation="linear"))
#adam = Adam(learning_rate=0.01)
model_2.compile(loss='mse', optimizer='adam', metrics=['mse'])

drop_out_history = model_2.fit(X_train, y_train, epochs=500,
                        validation_data = (X_test, y_test),
                        verbose=False)
```

```
# evaluate the model
_, train_mse = model_2.evaluate(X_train, y_train, verbose=0)
```

```
_, test_mse = model_2.evaluate(X_test, y_test, verbose=0)
print('Train: {}, Test: {}'.format(train_mse, test_mse))
```

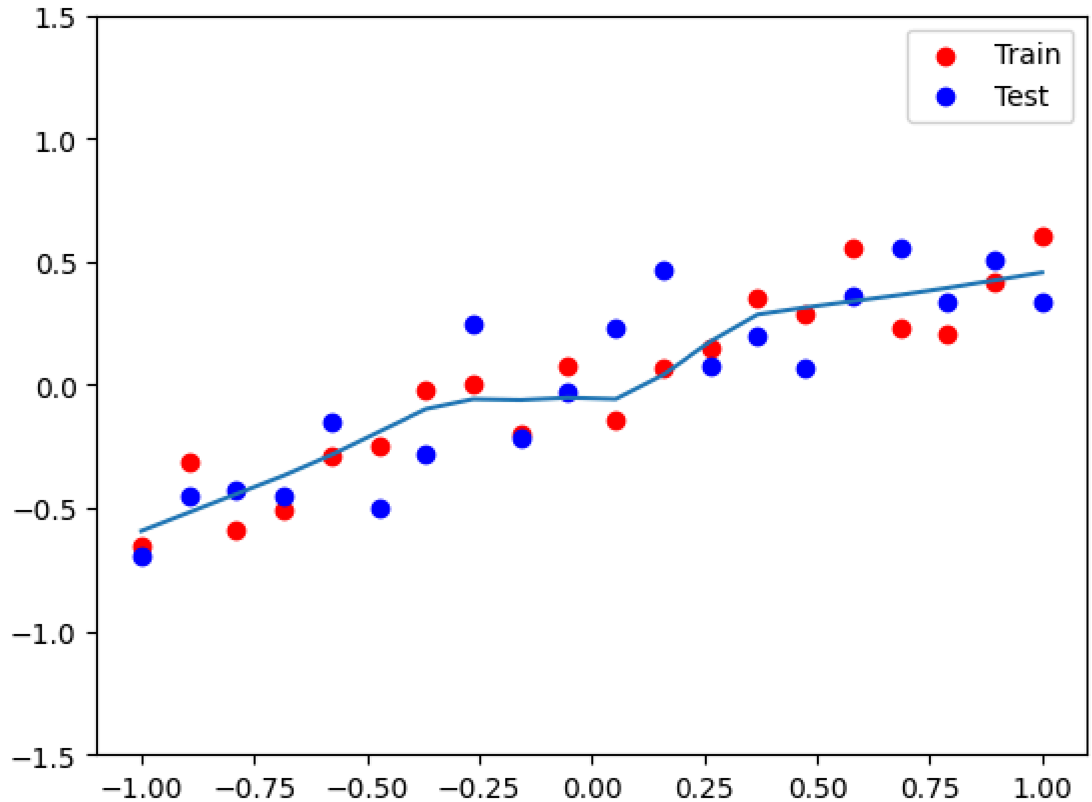
```
Train: 0.013462228700518608, Test: 0.03443724662065506
```

`model_2.evaluate(X_train, y_train, verbose=0)` :

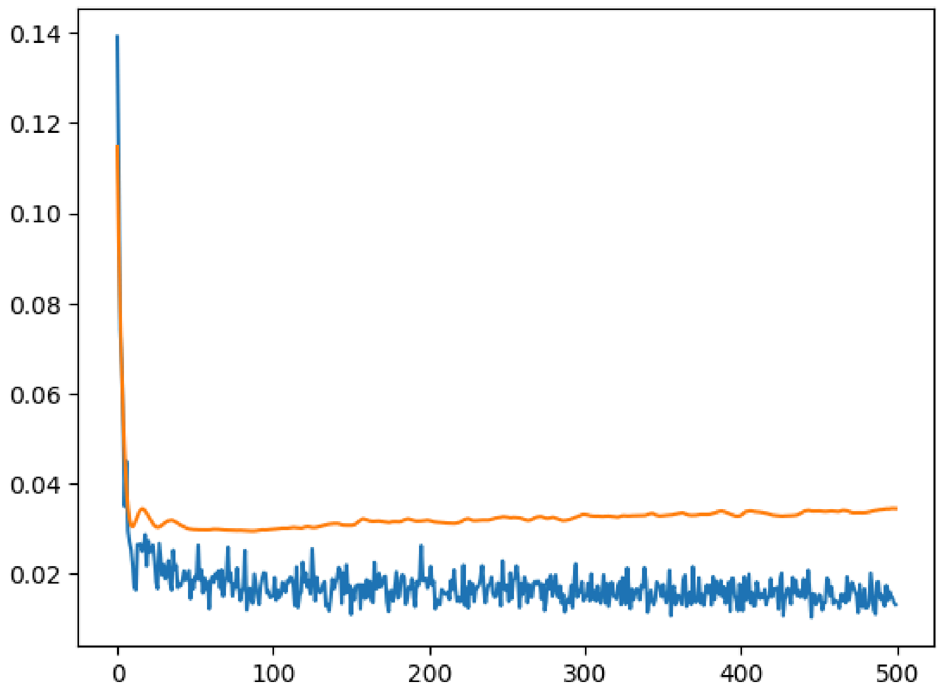
- This will return same value twice because both the **loss** and **metric** being returned by the `evaluate()` function are the same.

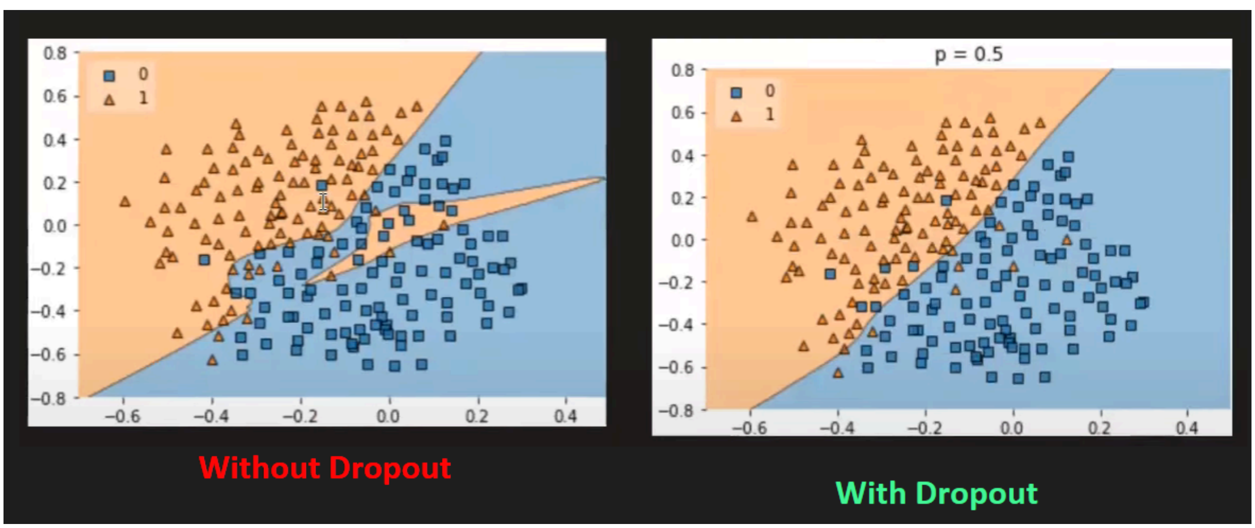
```
y_pred_2 = model_2.predict(X_test)

plt.figure()
plt.scatter(X_train, y_train, c='red', label='Train')
plt.scatter(X_test, y_test, c='blue', label='Test')
plt.plot(X_test, y_pred_2)
plt.legend()
plt.ylim((-1.5, 1.5))
plt.show()
```




```
plt.plot(drop_out_history.history['loss'])
plt.plot(drop_out_history.history['val_loss'])
```


0.14
0.12
0.10
0.08
0.06
0.04
0.02
0
100
200
300
400
500


0
1
p = 0.5
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
−0.8
−0.6
−0.4
−0.2
0.0
0.2
0.4
Without Dropout
With Dropout