Dive into DL

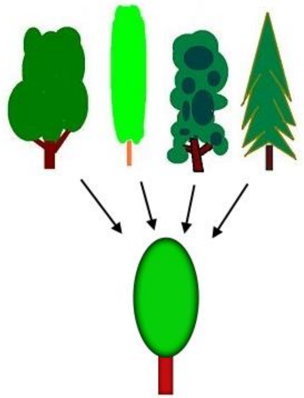
4.3 ~ 4.6

4.4 Model Selection, Underfitting, and Overfitting

- Our goal
  discover *patterns*
- *Overfitting*
  fitting our training data more closely than we fit the underlying distribution
- *Training Model*
  - *Regularization for avoid overfitting.*
  - altered the model structure or the hyperparameters

4.4.1. Training Error and Generalization Error

- training error
  error of our model as calculated on the training dataset
- generalization error(out-of-sample error)
  how accurately an algorithm is able to predict outcome values for previously unseen data.

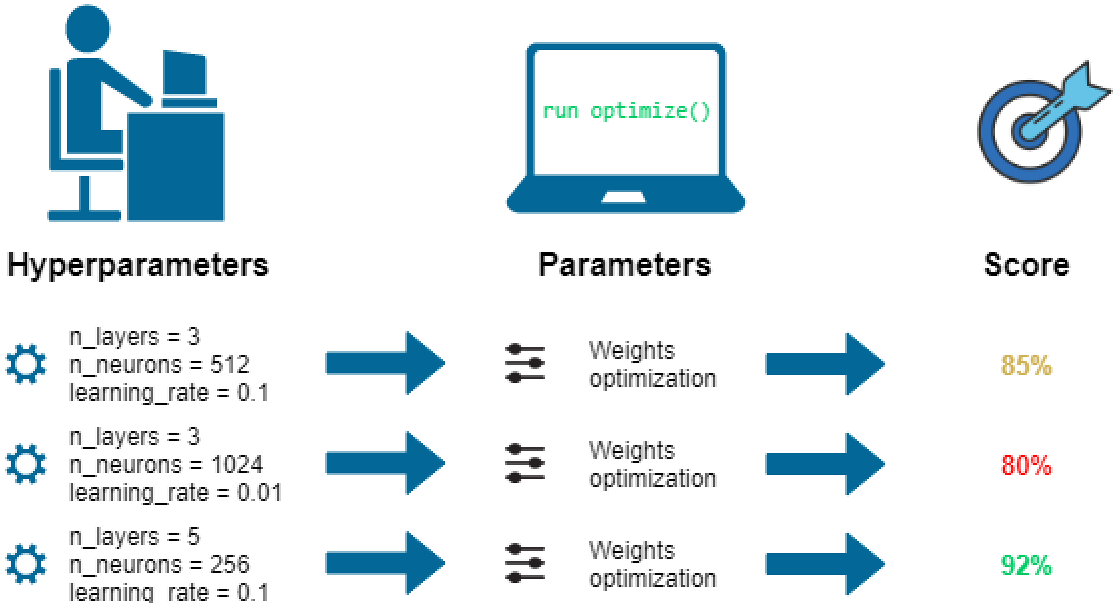


4.4.1.2. Model Complexity

- epochs, number of parameters, variable range ….
- Difficult to compare the complexity among different model classes (decision trees vs. neural networks).
- Model Complexity
  - explain arbitrary facts -> complex
  - explain the data -> truth

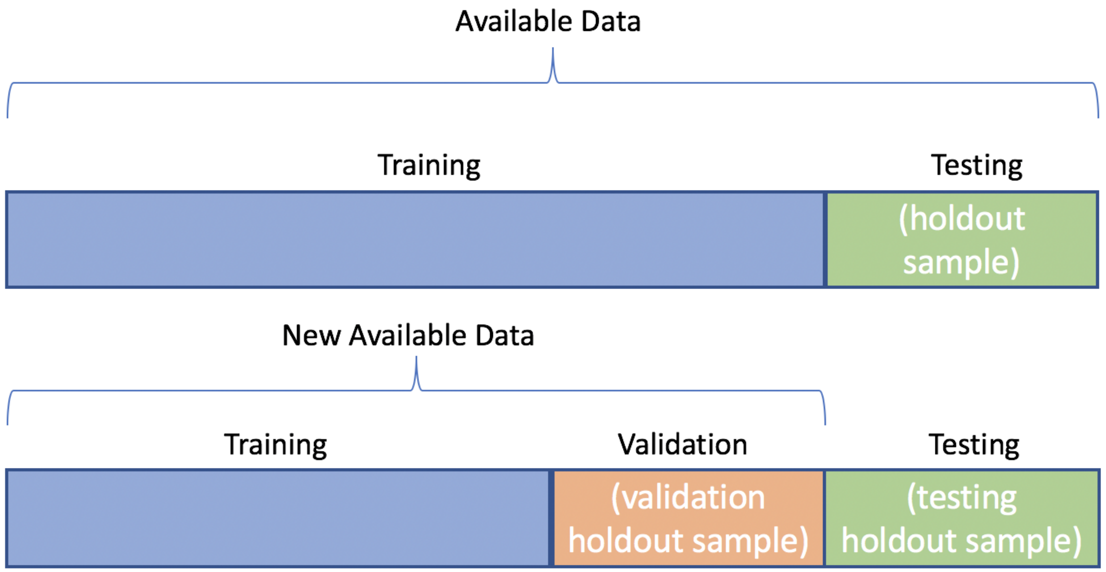
4.4.2. Model Selection

- Select our final model after evaluating several candidate models



4.4.2.1. Validation Dataset

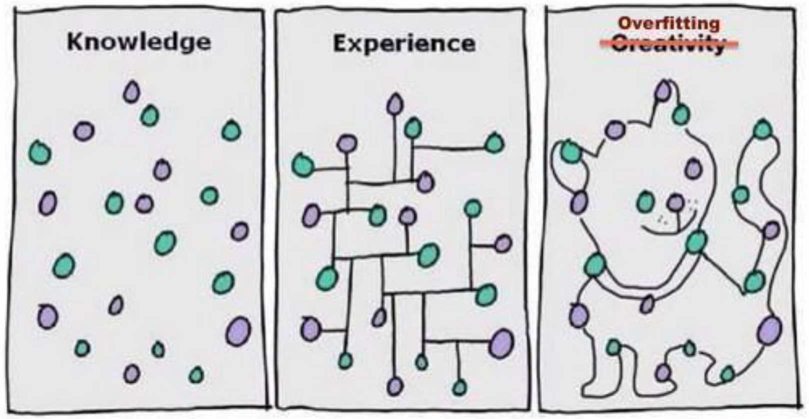
- In order to avoid overfit, the test set is used after the hyperparameter is selected.
- Dataset types
  - Train dataset
  - Validation dataset
  - Test dataset



4.4.2.2. K-Fold Cross-Validation

4.4.3. Underfitting or Overfitting?

- Overfitting
  - When the size of the dataset is small compared to the model.
  - Unable to reduce the training error
    -> model can be too simple.
  - Training error is lower than validation error
    -> can be *overfitting*
  - *Generalization gap* between our training and validation errors is small
    -> Model can be simplified



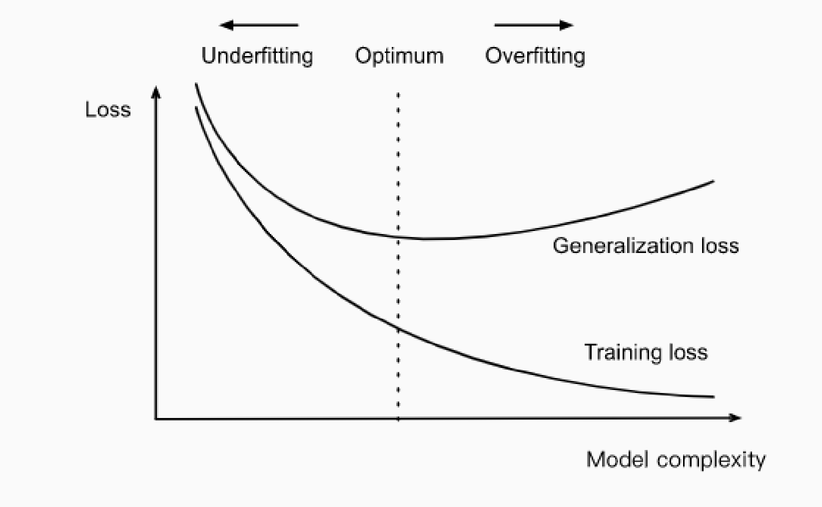
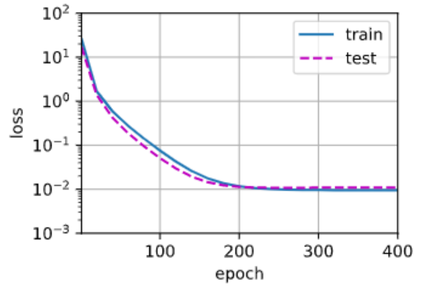


Fig. 4.4.1 Influence of model complexity on underfitting and overfitting

4.4 Training and Testing the Model

```
# Pick the first four dimensions, i.e., 1, x, x^2/2!, x^3/3! from the
# polynomial features
train(poly_features[:n_train, :4], poly_features[n_train:, :4],
      labels[:n_train], labels[n_train:])
```

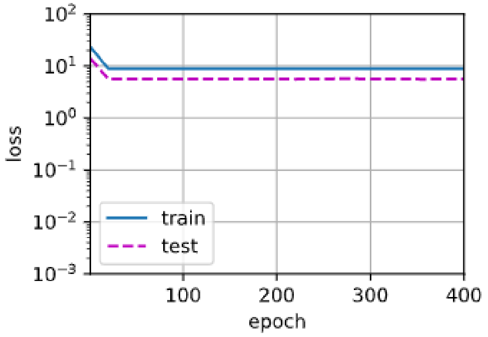
```
weight: [[ 4.98945    1.1688374 -3.381537   5.640259 ]]
```



Third-Order Polynomial Function Fitting (Normal)

```
# Pick the first two dimensions, i.e., 1, x, from the polynomial features
train(poly_features[:n_train, :2], poly_features[n_train:, :2],
      labels[:n_train], labels[n_train:])
```

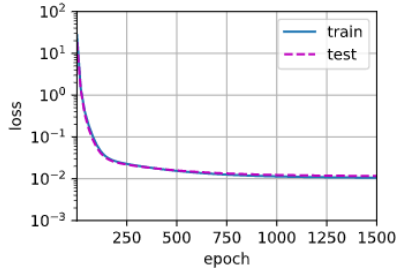
```
weight: [[2.489514  4.1598744]]
```



Linear Function Fitting (Underfitting)

```
[15] # Pick all the dimensions from the polynomial features
     train(poly_features[:n_train, :], poly_features[n_train:, :],
           labels[:n_train], labels[n_train:], num_epochs=1500)
```

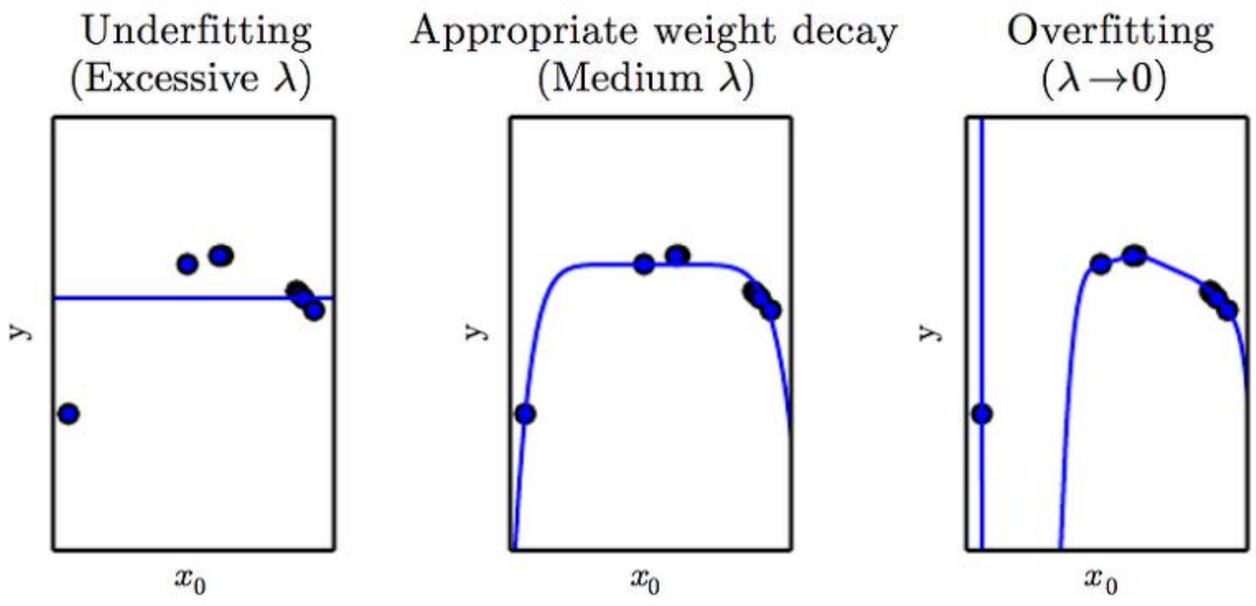
```
weight: [[ 4.97836351e+00  1.22782564e+00 -3.35342097e+00  5.31587553e+00
  -1.39680371e-01  1.02568424e+00  4.66782272e-01  3.72974835e-02
   2.16147974e-01 -4.92126256e-01 -6.52745739e-02  4.17911679e-01
   3.17682385e-01 -3.69303860e-04  1.01500556e-01 -1.05708912e-02
  -5.15405715e-01  4.23947692e-01  1.84150815e-01 -4.10240740e-01]]
```



Higher-Order Polynomial Function Fitting (Overfitting)

4.5. Weight Decay

- To avoid overfitting
  - collecting more training data
  - Adjusting function complexity(reducing the number of polynomial dimensions.)
- Weight decay (commonly called L2 regularization)
  - regularizing parametric machine learning models



4.5.1. Norms and Weight Decay

- Norm : function from a vector space

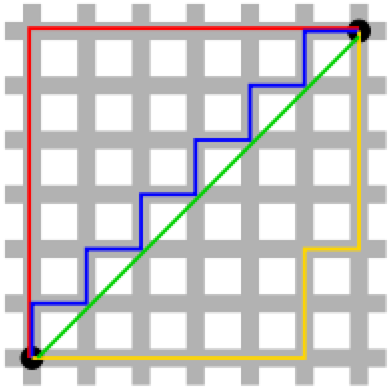
$$\|\mathbf{x}\|_p := \left(\sum_{i=1}^{n} |x_i|^p\right)^{1/p}.$$

- **L1 Norm**

$$d_1(\mathbf{p},\mathbf{q}) = \|\mathbf{p}-\mathbf{q}\|_1 = \sum_{i=1}^{n} |p_i - q_i|, \text{ where } (\mathbf{p},\mathbf{q}) \text{ are vectors } \mathbf{p} = (p_1, p_2, \dots, p_n) \text{ and } \mathbf{q} = (q_1, q_2, \dots, q_n)$$

- **L2 Norm**

$$\|\boldsymbol{x}\|_2 := \sqrt{x_1^2 + \cdots + x_n^2}.$$



4.5.1. **L2 Regularizatio (Ridge)**

L2 Loss Function

$$L = \sum_{i=1}^{n} (y_i - f(x_i))^2$$

L2 Regularization

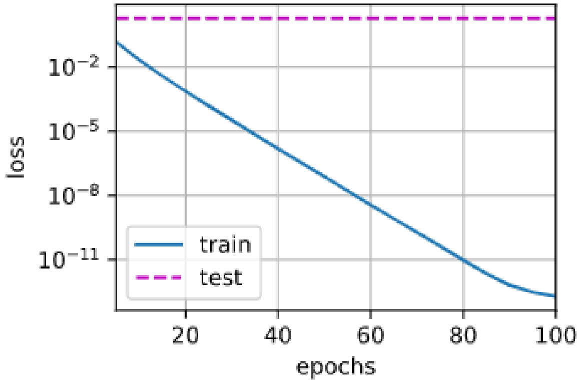
$$Cost = \frac{1}{n} \sum_{i=1}^{n} \{L(y_i, \hat{y_i}) + \frac{\lambda}{2} |w|^2\}$$

4.5.3.5. Using Weight Decay

```python
def train_concise(wd):
    net = tf.keras.models.Sequential()
    net.add(tf.keras.layers.Dense(
        1, kernel_regularizer=tf.keras.regularizers.l2(wd)))
    net.build(input_shape=(1, num_inputs))
    w, b = net.trainable_variables
    loss = tf.keras.losses.MeanSquaredError()
    num_epochs, lr = 100, 0.003
    trainer = tf.keras.optimizers.SGD(learning_rate=lr)
    animator = d2l.Animator(xlabel='epochs', ylabel='loss', yscale='log',
                            xlim=[5, num_epochs], legend=['train', 'test'])
    for epoch in range(num_epochs):
        for X, y in train_iter:
            with tf.GradientTape() as tape:
                # `tf.keras` requires retrieving and adding the losses from
                # layers manually for custom training loop.
                l = loss(net(X), y) + net.losses
            grads = tape.gradient(l, net.trainable_variables)
            trainer.apply_gradients(zip(grads, net.trainable_variables))
        if (epoch + 1) % 5 == 0:
            animator.add(epoch + 1, (d2l.evaluate_loss(net, train_iter, loss),
                                     d2l.evaluate_loss(net, test_iter, loss)))
    print('L2 norm of w:', tf.norm(net.get_weights()[0]).numpy())
```

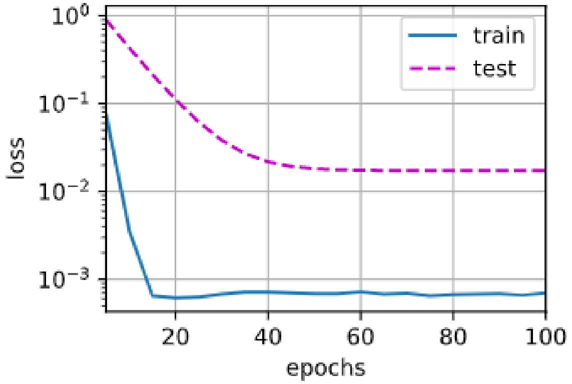
```python
train_concise(0)
```

L2 norm of w: 1.2907445



24]
```python
train_concise(3)
```

L2 norm of w: 0.028680595



4.6. Dropout

- spread out its weights among many features

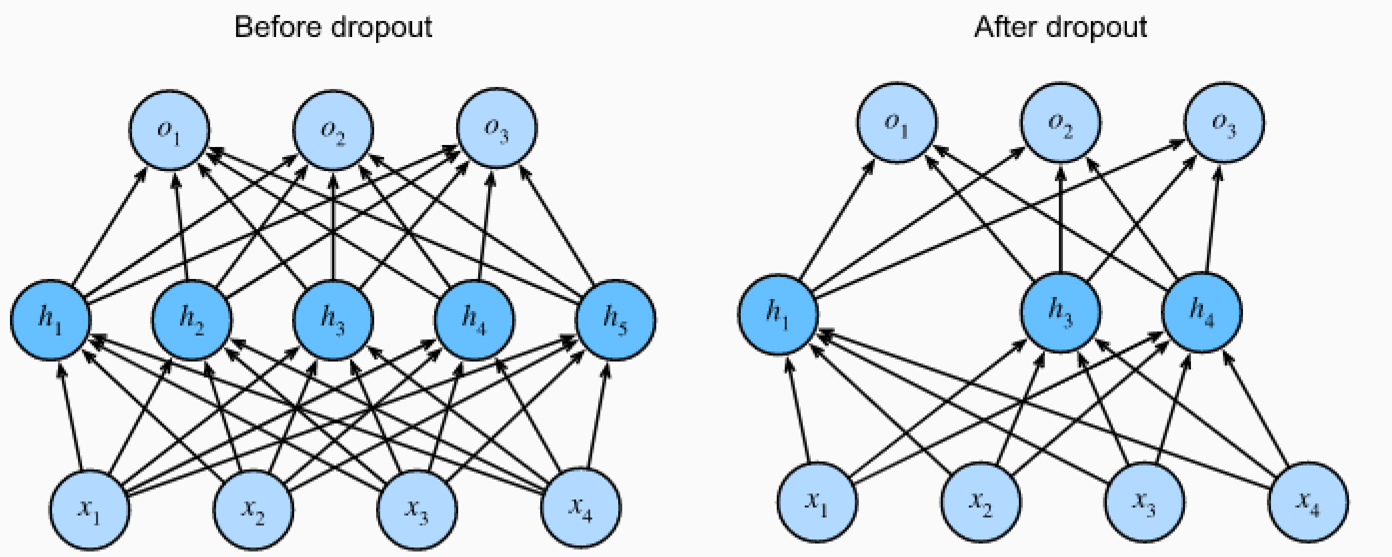


Fig. 4.6.1 MLP before and after dropout.

4.6.2. Robustness through Perturbations

neural network overfitting is characterized by a state in which each layer relies on a specific pattern of activations in the previous layer (=*co-adaptation)*
-> *dropout :* breaks up co-adaptation

At each training iteration, we added noise sampled from a distribution with mean zero

$\epsilon \sim \mathcal{N}(0, \sigma^2)$ to the input $\mathbf{x}$, yielding a perturbed point $\mathbf{x}' = \mathbf{x} + \epsilon$. In expectation, $E[\mathbf{x}'] = \mathbf{x}$.

with *dropout probability* p, each intermediate activation h is replaced by a random variable h' as follows:

$$h' = \begin{cases} 0 & \text{with probability } p \\ \frac{h}{1-p} & \text{otherwise} \end{cases}$$

By design, the expectation remains unchanged, i.e., $E[h'] = h$.

4.6 Implementation

```python
net = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(256, activation=tf.nn.relu),
    # Add a dropout layer after the first fully connected layer
    tf.keras.layers.Dropout(dropout1),
    tf.keras.layers.Dense(256, activation=tf.nn.relu),
    # Add a dropout layer after the second fully connected layer
    tf.keras.layers.Dropout(dropout2),
    tf.keras.layers.Dense(10),
])
```

```python
trainer = tf.keras.optimizers.SGD(learning_rate=lr)
d2l.train_ch3(net, train_iter, test_iter, loss, num_epochs, trainer)
```