Pandas will be used to read in and process the data.
We can download and cache the Kaggle housing dataset for our convenience.

In [ ]:
```
%matplotlib inline
import pandas as pd
import torch
from torch import nn
from d2l import torch as d2l
```

In [3]:
```
class KaggleHouse(d2l.DataModule):
    def __init__(self, batch_size, train=None, val=None):
        super().__init__()
        self.save_hyperparameters()
        if self.train is None:
            self.raw_train = pd.read_csv(d2l.download(
                d2l.DATA_URL + 'kaggle_house_pred_train.csv', self.roo
                sha1_hash='585e9cc93e70b39160e7921475f9bcd7d31219ce'))
            self.raw_val = pd.read_csv(d2l.download(
                d2l.DATA_URL + 'kaggle_house_pred_test.csv', self.root
                sha1_hash='fa19780a7b011d9b009e8bff8e99922a8ee2eb90'))
```

The validation dataset has 1459 examples and 80 features,
while the training dataset has 1460 examples, 80 features, and 1 label.

In [4]:
```
data = KaggleHouse(batch_size=64)
print(data.raw_train.shape)
print(data.raw_val.shape)
```

```
Downloading ../data/kaggle_house_pred_train.csv from http://d2l-data.s3-accelerate.amazonaws.com/kaggle_house_pred_train.csv... (http://d2l-data.s3-accelerate.amazonaws.com/kaggle_house_pred_train.csv...)
Downloading ../data/kaggle_house_pred_test.csv from http://d2l-data.s3-accelerate.amazonaws.com/kaggle_house_pred_test.csv... (http://d2l-data.s3-accelerate.amazonaws.com/kaggle_house_pred_test.csv...)
(1460, 81)
(1459, 80)
```

Data Preprocessing

In [5]:
```
print(data.raw_train.iloc[:4, [0, 1, 2, 3, -3, -2, -1]])
```

```
   Id  MSSubClass MSZoning  LotFrontage SaleType SaleCondition  SaleP
rice
0   1          60       RL         65.0       WD        Normal     20
8500
1   2          20       RL         80.0       WD        Normal     18
1500
2   3          60       RL         68.0       WD        Normal     22
3500
3   4          70       RL         60.0       WD       Abnorml     14
0000
```

We can see that in each example, the first feature is the ID. This helps the model identify each training example. While this is convenient, it does not carry any information for prediction purposes. Hence, we will remove it from the dataset before feeding the data into the model. Besides, given a wide variety of data types, we will need to preprocess the data before we can start modeling.

In [6]:
```
@d2l.add_to_class(KaggleHouse)
def preprocess(self):
    # Remove the ID and label columns
    label = 'SalePrice'
    features = pd.concat(
        (self.raw_train.drop(columns=['Id', label]),
         self.raw_val.drop(columns=['Id'])))
    # Standardize numerical columns
    numeric_features = features.dtypes[features.dtypes != 'object'].in
    features[numeric_features] = features[numeric_features].apply(
        lambda x: (x - x.mean()) / (x.std()))
    # Replace NAN numerical features by 0
    features[numeric_features] = features[numeric_features].fillna(0)
    # Replace discrete features by one-hot encoding.
    features = pd.get_dummies(features, dummy_na=True)
    # Save preprocessed features
    self.train = features[:self.raw_train.shape[0]].copy()
    self.train[label] = self.raw_train[label]
    self.val = features[self.raw_train.shape[0]:].copy()
```

As you can see, this conversion results in an increase of 79 features to 331 features (excluding ID and label columns).

In [7]:
```
data.preprocess()
data.train.shape
```

Out[7]: (1460, 332)

Error Measure

We'll train a linear model with squared loss first. Unsurprisingly, our linear model will not result in a submission that wins the competition, but it does serve as a sanity check to determine whether the data contains any useful information. There may be a good likelihood that we have a data processing fault if we are unable to perform better than random guessing in this situation. If everything goes as planned, the linear model will act as a baseline, providing us an idea of how closely the simple model approaches the best reported models and how much improvement we can reasonably anticipate from more complex models.

In [8]:
```
@d2l.add_to_class(KaggleHouse)
def get_dataloader(self, train):
    label = 'SalePrice'
    data = self.train if train else self.val
    if label not in data: return
    get_tensor = lambda x: torch.tensor(x.values, dtype=torch.float32)
    # Logarithm of prices
    tensors = (get_tensor(data.drop(columns=[label])),  # X
               torch.log(get_tensor(data[label])).reshape((-1, 1)))  #
    return self.get_tensorloader(tensors, train)
```

K-Fold Cross-Validation

In [9]:

```
def k_fold_data(data, k):
    rets = []
    fold_size = data.train.shape[0] // k
    for j in range(k):
        idx = range(j * fold_size, (j+1) * fold_size)
        rets.append(KaggleHouse(data.batch_size, data.train.drop(index
                                data.train.loc[idx]))
    return rets
```

```
The average validation error is returned when we train
K times in the
K-fold cross-validation.
```

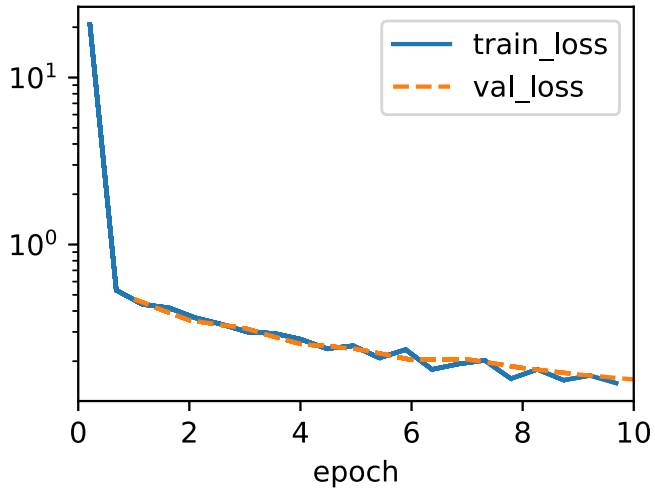
```
Model Selection
```

```
We choose a collection of hyperparameters that is not tuned, and we
leave it to the reader to refine the model. Depending on how many
factors one optimises over, finding a decent option can take some
time. K-fold cross-validation is typically fairly resistant to
multiple testing when used with a large enough dataset and the
typical types of hyperparameters. But if we test an excessive number
of possibilities, we can strike it lucky and discover that our
validation performance is no longer indicative of the real fault.
```

In [10]:
```
def k_fold(trainer, data, k, lr):
    val_loss, models = [], []
    for i, data_fold in enumerate(k_fold_data(data, k)):
        model = d2l.LinearRegression(lr)
        model.board.yscale='log'
        if i != 0: model.board.display = False
        trainer.fit(model, data_fold)
        val_loss.append(float(model.board.data['val_loss'][-1].y))
        models.append(model)
    print(f'average validation log mse = {sum(val_loss)/len(val_loss)}')
    return models
```

In [11]:
```
trainer = d2l.Trainer(max_epochs=10)
models = k_fold(trainer, data, k=5, lr=0.01)
```

```
average validation log mse = 0.17507080286741256
```



The amount of training mistakes for a given set of hyperparameters can occasionally be quite low, despite the fact that the number of errors on K-fold cross-validation is significantly higher. This suggests that we are fitting too closely. You should keep an eye on both figures while you workout. A stronger model may be supported by our data if there is less overfitting. Massive overfitting may imply that regularisation approaches can benefit us.

In [ ]:
```
Now that we know what a good choice of hyperparameters should be, we m
K models.
```

In [12]:
```
preds = [model(torch.tensor(data.val.values, dtype=torch.float32))
         for model in models]
# Taking exponentiation of predictions in the logarithm scale
ensemble_preds = torch.exp(torch.cat(preds, 1)).mean(1)
submission = pd.DataFrame({'Id':data.raw_val.Id,
                           'SalePrice':ensemble_preds.detach().numpy()})
submission.to_csv('submission.csv', index=False)
```

Summary

Real data frequently consists of a variety of different data kinds and requires preprocessing. Real-valued data should be rescaled by default to have a zero mean and unit variance. As is using the mean of the missing data in their place. Additionally, we can treat category features as one-hot vectors by converting them into indicator features. We can assess the difference in the prediction's logarithm when we tend to care more about the relative error than the absolute error. K-fold cross-validation can be used to choose the model and modify the hyperparameters.