

Homework 4 - Berkeley STAT 157

Your name: Andrew Tan


SID 26673509

teammates (individual) (Please add your name, SID and teammates to ease Ryan and Rachel to grade.)

Handout 2/12/2019, due 2/19/2019 by 4pm in Git by committing to your repository.

In this homework, we will build a model based real house sale data from a [Kaggle competition](https://www.kaggle.com/c/house-prices-advanced-regression-techniques) (<https://www.kaggle.com/c/house-prices-advanced-regression-techniques>). This notebook contains codes to download the dataset, build and train a baseline model, and save the results in the submission format. Your jobs are

1. Developing a better model to reduce the prediction error. You can find some hints on the last section.
2. Submitting your results into Kaggle and take a screenshot of your score. Then replace the following image URL with your screenshot.

34	Andrew Tan		0.10984	8	now
Your Best Entry ↑					
Your submission scored 0.11314, which is not an improvement of your best score. Keep trying!					

We have two suggestions for this homework:

1. Start as earlier as possible. Though we will cover this notebook on Thursday's lecture, tuning hyper-parameters takes time, and Kaggle limits #submissions per day.
2. Work with your project teammates. It's a good opportunity to get familiar with each other.

Your scores will depend your positions on Kaggle's Leaderboard. We will award the top-3 teams/individuals 500 AWS credits.

Accessing and Reading Data Sets

The competition data is separated into training and test sets. Each record includes the property values of the house and attributes such as street type, year of construction, roof type, basement condition. The data includes multiple datatypes, including integers (year of construction), discrete labels (roof type), floating point numbers, etc.; Some data is missing and is thus labeled 'na'. The price of each house, namely the label, is only included in the training data set (it's a competition after all). The 'Data' tab on the competition tab has links to download the data.

We will read and process the data using pandas, an [efficient data analysis toolkit](http://pandas.pydata.org/pandas-docs/stable/) (<http://pandas.pydata.org/pandas-docs/stable/>). Make sure you have pandas installed for the experiments in this section.

My Approach

1. I regress on log-price directly, minimizing the same function that the Kaggle contest is being scored on.
2. I experiment with various ways of replacing missing data, including mean, median, and conditional values.
3. I performed extensive cross-validation and hyperparameter tuning, specifically for layers, nodes per layer, dropout values, weight decay, learning rate, and batch size.
4. I improved the model by adding additional layers, using ReLU activation function, and dropout.
5. I experimented with various optimization methods, but ultimately decided on using the default Adam optimizer.
6. For my final solution, I used an ensemble of different models and took the average of my top models. I experimented with various combinations and ensembles using Excel.

```
In [1]: # If pandas is not installed, please uncomment the following line:  
# !pip install pandas  
  
%matplotlib inline  
import d2l  
from mxnet import autograd, gluon, init, nd  
from mxnet.gluon import data as gdata, loss as gloss, nn, utils  
import numpy as np  
import pandas as pd
```

We downloaded the data into the current directory. To load the two CSV (Comma Separated Values) files containing training and test data respectively we use Pandas.

```
In [2]: utils.download('https://github.com/d2l-ai/d2l-en/raw/master/data/kaggle_house  
_pred_train.csv')  
utils.download('https://github.com/d2l-ai/d2l-en/raw/master/data/kaggle_house  
_pred_test.csv')  
train_data = pd.read_csv('kaggle_house_pred_train.csv')  
test_data = pd.read_csv('kaggle_house_pred_test.csv')
```

The training data set includes 1,460 examples, 80 features, and 1 label., the test data contains 1,459 examples and 80 features.

```
In [3]: print(train_data.shape)  
print(test_data.shape)  
  
(1460, 81)  
(1459, 80)
```

Let's take a look at the first 4 and last 2 features as well as the label (SalePrice) from the first 4 examples:

```
In [4]: train_data.iloc[0:4, [0, 1, 2, 3, -3, -2, -1]]
```

Out[4]:

	Id	MSSubClass	MSZoning	LotFrontage	SaleType	SaleCondition	SalePrice
0	1	60	RL	65.0	WD	Normal	208500
1	2	20	RL	80.0	WD	Normal	181500
2	3	60	RL	68.0	WD	Normal	223500
3	4	70	RL	60.0	WD	Abnorml	140000

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 doesn't carry any information for prediction purposes. Hence we remove it from the dataset before feeding the data into the network.

```
In [5]: all_features = pd.concat((train_data.iloc[:, 1:-1], test_data.iloc[:, 1:]))
```

Data Preprocessing

As stated above, we have a wide variety of datatypes. Before we feed it into a deep network we need to perform some amount of processing. Let's start with the numerical features. We begin by replacing missing values with the mean. This is a reasonable strategy if features are missing at random. To adjust them to a common scale we rescale them to zero mean and unit variance. This is accomplished as follows:

$$x \leftarrow \frac{x - \mu}{\sigma}$$

To check that this transforms x to data with zero mean and unit variance simply calculate

$\mathbf{E}[(x - \mu)/\sigma] = (\mu - \mu)/\sigma = 0$. To check the variance we use $\mathbf{E}[(x - \mu)^2] = \sigma^2$ and thus the transformed variable has unit variance. The reason for 'normalizing' the data is that it brings all features to the same order of magnitude. After all, we do not know *a priori* which features are likely to be relevant. Hence it makes sense to treat them equally.

```
In [6]: numeric_features = all_features.dtypes[all_features.dtypes != 'object'].index
all_features[numeric_features] = all_features[numeric_features].apply(
    lambda x: (x - x.mean()) / (x.std()))
# after standardizing the data all means vanish, hence we can set missing values to 0
# all_features = all_features.fillna(0)
```

Next we deal with discrete values. This includes variables such as 'MSZoning'. We replace them by a one-hot encoding in the same manner as how we transformed multiclass classification data into a vector of 0 and 1. For instance, 'MSZoning' assumes the values 'RL' and 'RM'. They map into vectors (1, 0) and (0, 1) respectively. Pandas does this automatically for us.

```
In [7]: # Dummy_na=True refers to a missing value being a legal eigenvalue, and creates an indicative feature for it.
all_features = pd.get_dummies(all_features, dummy_na=True)
all_features = all_features.fillna(all_features.mean()) # fill values using mean
# all_features = all_features.fillna(all_features.median()) # fill values using median
all_features.shape
```

```
Out[7]: (2919, 331)
```

You can see that this conversion increases the number of features from 79 to 331. Finally, via the `values` attribute we can extract the NumPy format from the Pandas dataframe and convert it into MXNet's native representation - NDAarray for training.

```
In [8]: n_train = train_data.shape[0]
train_features = nd.array(all_features[:n_train].values)
test_features = nd.array(all_features[n_train:].values)
train_labels = nd.array(train_data.SalePrice.values).reshape((-1, 1))
train_labels = train_labels.log() # train on log prices
```

Training

To get started we train a linear model with squared loss. This will obviously not lead to a competition winning submission but it provides a sanity check to see whether there's meaningful information in the data. It also amounts to a minimum baseline of how well we should expect any 'fancy' model to work.

```
In [22]: loss = gloss.L2Loss()

def get_net():
    net = gluon.nn.Sequential()
    net.add(gluon.nn.Dense(128, activation='relu'))
    net.add(gluon.nn.Dropout(0.1))
    # net.add(gluon.nn.Dense(64, activation='relu'))
    # net.add(gluon.nn.Dropout(0.2))
    net.add(gluon.nn.Dense(1))
    net.initialize(init.Xavier())
    return net
```

House prices, like shares, are relative. That is, we probably care more about the relative error $\frac{y-\hat{y}}{y}$ than about the absolute error. For instance, getting a house price wrong by USD 100,000 is terrible in Rural Ohio, where the value of the house is USD 125,000. On the other hand, if we err by this amount in Los Altos Hills, California, we can be proud of the accuracy of our model (the median house price there exceeds 4 million).

One way to address this problem is to measure the discrepancy in the logarithm of the price estimates. In fact, this is also the error that is being used to measure the quality in this competition. After all, a small value δ of $\log y - \log \hat{y}$ translates into $e^{-\delta} \leq \frac{\hat{y}}{y} \leq e^{\delta}$. This leads to the following loss function:

$$L = \sqrt{\frac{1}{n} \sum_{i=1}^n (\log y_i - \log \hat{y}_i)^2}$$

```
In [10]: def log_rmse(net, features, labels):
    # To further stabilize the value when the logarithm is taken, set the value less than 1 as 1.
    clipped_preds = nd.clip(net(features), 1, float('inf'))
    rmse = nd.sqrt(2 * loss(clipped_preds.log(), labels.log()).mean())
    return rmse.asscalar()
```

Unlike in the previous sections, the following training functions use the Adam optimization algorithm. Compared to the previously used mini-batch stochastic gradient descent, the Adam optimization algorithm is relatively less sensitive to learning rates. This will be covered in further detail later on when we discuss the details on [Optimization Algorithms](#) ([./chapter_optimization/index.md](#)) in a separate chapter.

```
In [11]: def train(net, train_features, train_labels, test_features, test_labels,
    num_epochs, learning_rate, weight_decay, batch_size):
    train_ls, test_ls = [], []
    train_iter = gdata.DataLoader(gdata.ArrayDataset(
        train_features, train_labels), batch_size, shuffle=True)
    # The Adam optimization algorithm is used here.
    trainer = gluon.Trainer(net.collect_params(), 'adam', {
        'learning_rate': learning_rate, 'wd': weight_decay})
    for epoch in range(num_epochs):
        for X, y in train_iter:
            with autograd.record():
                l = loss(net(X), y)
            l.backward()
            trainer.step(batch_size)
        train_ls.append(log_rmse(net, train_features, train_labels))
        if test_labels is not None:
            test_ls.append(log_rmse(net, test_features, test_labels))
    return train_ls, test_ls
```

k-Fold Cross-Validation

The k-fold cross-validation was introduced in the section where we discussed how to deal with [“Model Selection, Underfitting and Overfitting” \(underfit-overfit.md\)](#). We will put this to good use to select the model design and to adjust the hyperparameters. We first need a function that returns the i-th fold of the data in a k-fold cross-validation procedure. It proceeds by slicing out the i-th segment as validation data and returning the rest as training data. Note - this is not the most efficient way of handling data and we would use something much smarter if the amount of data was considerably larger. But this would obscure the function of the code considerably and we thus omit it.

```
In [12]: def get_k_fold_data(k, i, X, y):
    assert k > 1
    fold_size = X.shape[0] // k
    X_train, y_train = None, None
    for j in range(k):
        idx = slice(j * fold_size, (j + 1) * fold_size)
        X_part, y_part = X[idx, :], y[idx]
        if j == i:
            X_valid, y_valid = X_part, y_part
        elif X_train is None:
            X_train, y_train = X_part, y_part
        else:
            X_train = nd.concat(X_train, X_part, dim=0)
            y_train = nd.concat(y_train, y_part, dim=0)
    return X_train, y_train, X_valid, y_valid
```

The training and verification error averages are returned when we train k times in the k-fold cross-validation.

```
In [13]: def k_fold(k, X_train, y_train, num_epochs,
    learning_rate, weight_decay, batch_size):
    train_l_sum, valid_l_sum = 0, 0
    for i in range(k):
        data = get_k_fold_data(k, i, X_train, y_train)
        net = get_net()
        train_ls, valid_ls = train(net, *data, num_epochs, learning_rate,
                                   weight_decay, batch_size)
        train_l_sum += train_ls[-1]
        valid_l_sum += valid_ls[-1]
        if i == 0:
            d2l.semilogy(range(1, num_epochs + 1), train_ls, 'epochs', 'rmse'
            ,
                        range(1, num_epochs + 1), valid_ls,
                        ['train', 'valid'])
        print('fold %d, train rmse: %f, valid rmse: %f' % (
            i, train_ls[-1], valid_ls[-1]))
    return train_l_sum / k, valid_l_sum / k
```

Model Selection

We pick a rather un-tuned set of hyperparameters and leave it up to the reader to improve the model considerably. Finding a good choice can take quite some time, depending on how many things one wants to optimize over. Within reason the k-fold crossvalidation approach is resilient against multiple testing. However, if we were to try out an unreasonably large number of options it might fail since we might just get lucky on the validation split with a particular set of hyperparameters.

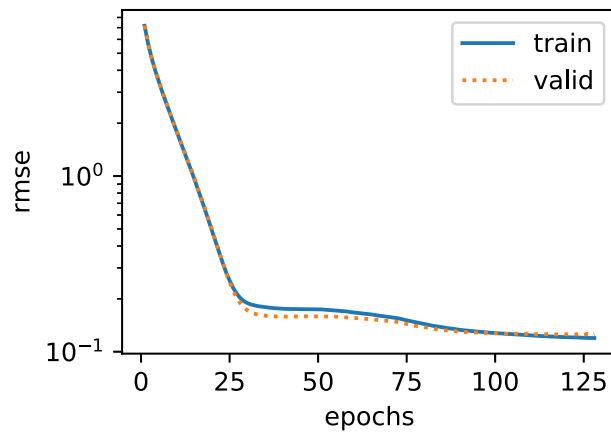
```

In [14]: k, num_epochs, lr, weight_decay, batch_size = 5, 128, 0.1, 130, 128
         lrs = [0.01, 0.05, 0.1, 0.15]
         weight_decays = [100, 110, 130, 140, 150]
         batch_sizes = [64, 128]

         best_l = float('inf')
         best_param = None
         tune_params = {}
         for lr in lrs:
             for weight_decay in weight_decays:
                 for batch_size in batch_sizes:
                     print("lr:", lr)
                     print("weight_decay:", weight_decay)
                     print("batch_size:", batch_size)
                     train_l, valid_l = k_fold(k, train_features, train_labels, num_epochs, lr,
                                             weight_decay, batch_size)
                     print('%d-fold validation: avg train rmse: %f, avg valid rmse: %f'
                           ,
                           % (k, train_l, valid_l))
                     if valid_l < best_l:
                         best_l = valid_l
                         best_param = (lr, weight_decay, batch_size)
                     tune_params[(lr, weight_decay, batch_size)] = valid_l
         print()

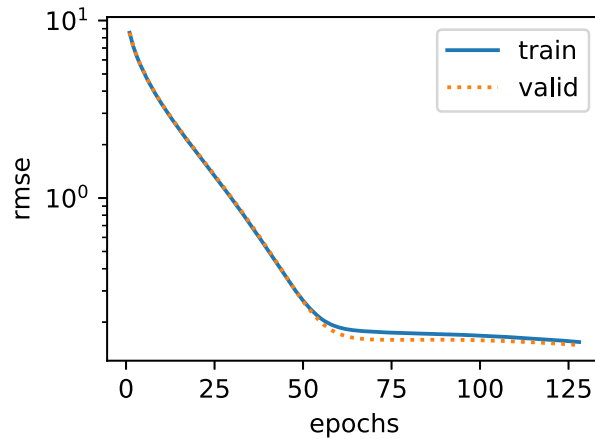
```


lr: 0.01
weight_decay: 100
batch_size: 64



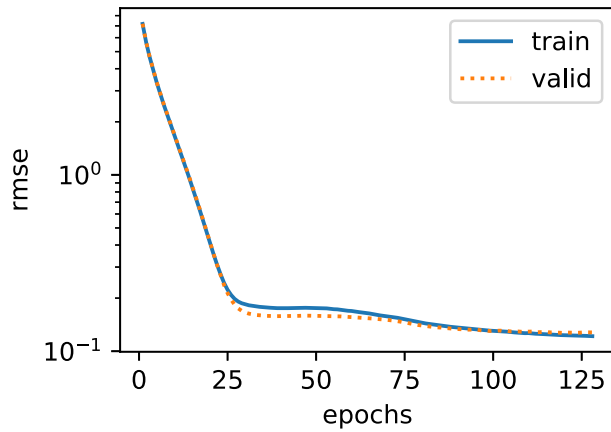
fold 0, train rmse: 0.119630, valid rmse: 0.126455
fold 1, train rmse: 0.112200, valid rmse: 0.136586
fold 2, train rmse: 0.117556, valid rmse: 0.134694
fold 3, train rmse: 0.120140, valid rmse: 0.111584
fold 4, train rmse: 0.111916, valid rmse: 0.154247
5-fold validation: avg train rmse: 0.116289, avg valid rmse: 0.132713

lr: 0.01
weight_decay: 100
batch_size: 128



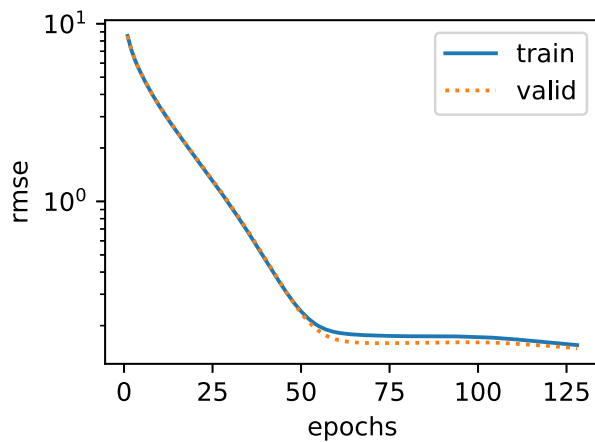
fold 0, train rmse: 0.154507, valid rmse: 0.148501
fold 1, train rmse: 0.153863, valid rmse: 0.183585
fold 2, train rmse: 0.154373, valid rmse: 0.158083
fold 3, train rmse: 0.154365, valid rmse: 0.146100
fold 4, train rmse: 0.153550, valid rmse: 0.178723
5-fold validation: avg train rmse: 0.154132, avg valid rmse: 0.162998

lr: 0.01
weight_decay: 110
batch_size: 64



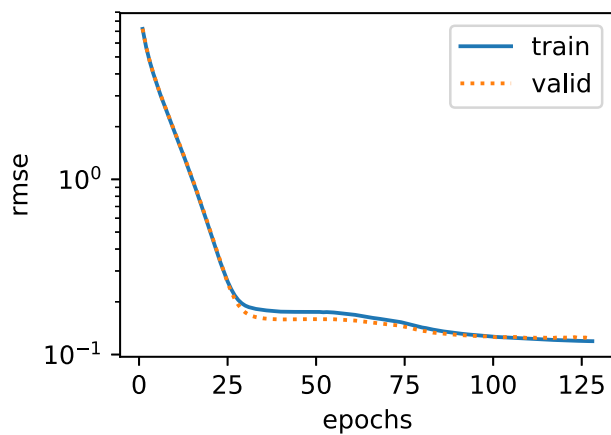
```
fold 0, train rmse: 0.121673, valid rmse: 0.127793
fold 1, train rmse: 0.113840, valid rmse: 0.137033
fold 2, train rmse: 0.116794, valid rmse: 0.134182
fold 3, train rmse: 0.119781, valid rmse: 0.111671
fold 4, train rmse: 0.110855, valid rmse: 0.153374
5-fold validation: avg train rmse: 0.116589, avg valid rmse: 0.132811
```

```
lr: 0.01
weight_decay: 110
batch_size: 128
```



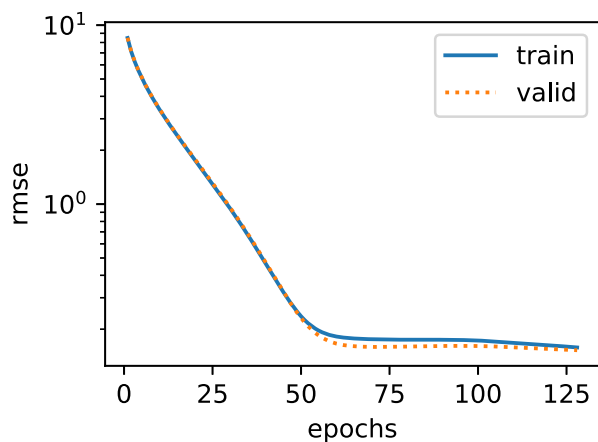
```
fold 0, train rmse: 0.155375, valid rmse: 0.149142
fold 1, train rmse: 0.150363, valid rmse: 0.175684
fold 2, train rmse: 0.148078, valid rmse: 0.152465
fold 3, train rmse: 0.155834, valid rmse: 0.147139
fold 4, train rmse: 0.148884, valid rmse: 0.176753
5-fold validation: avg train rmse: 0.151707, avg valid rmse: 0.160236
```

```
lr: 0.01
weight_decay: 130
batch_size: 64
```



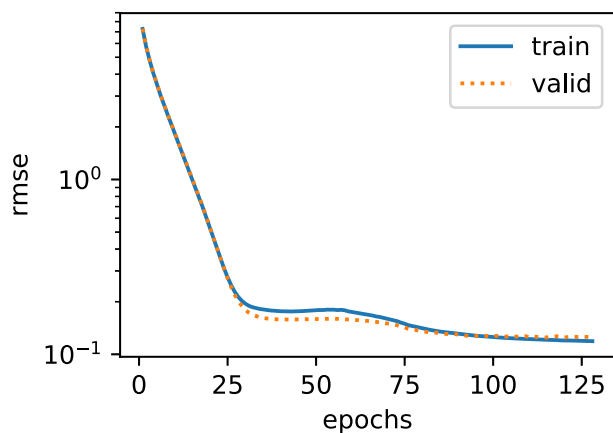
```
fold 0, train rmse: 0.119198, valid rmse: 0.124712
fold 1, train rmse: 0.115081, valid rmse: 0.137463
fold 2, train rmse: 0.116954, valid rmse: 0.134121
fold 3, train rmse: 0.121701, valid rmse: 0.113371
fold 4, train rmse: 0.111315, valid rmse: 0.153559
5-fold validation: avg train rmse: 0.116850, avg valid rmse: 0.132645
```

```
lr: 0.01
weight_decay: 130
batch_size: 128
```



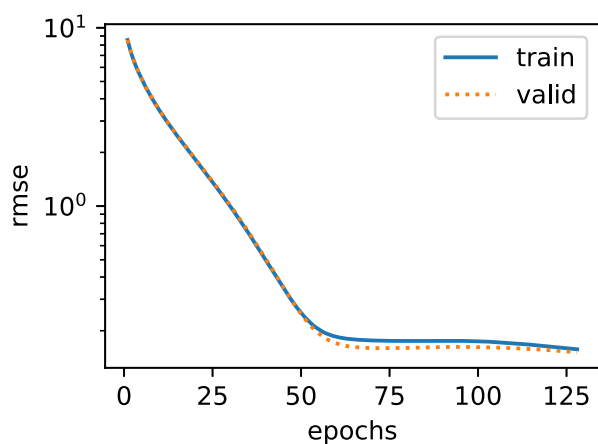
```
fold 0, train rmse: 0.158180, valid rmse: 0.152626
fold 1, train rmse: 0.148528, valid rmse: 0.176354
fold 2, train rmse: 0.154432, valid rmse: 0.158630
fold 3, train rmse: 0.153272, valid rmse: 0.144827
fold 4, train rmse: 0.146598, valid rmse: 0.175688
5-fold validation: avg train rmse: 0.152202, avg valid rmse: 0.161625
```

```
lr: 0.01
weight_decay: 140
batch_size: 64
```



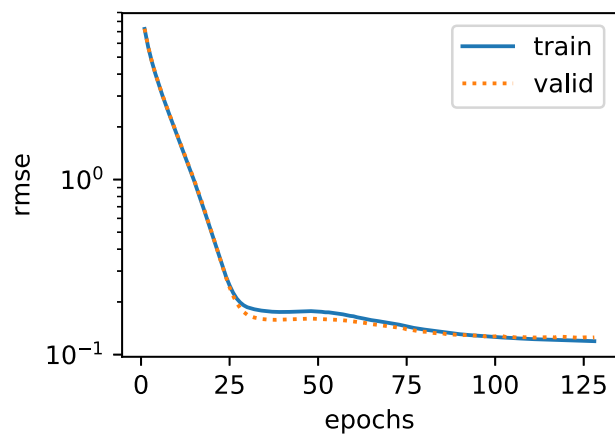
```
fold 0, train rmse: 0.118903, valid rmse: 0.125251
fold 1, train rmse: 0.113367, valid rmse: 0.137002
fold 2, train rmse: 0.116673, valid rmse: 0.133613
fold 3, train rmse: 0.121238, valid rmse: 0.112199
fold 4, train rmse: 0.110166, valid rmse: 0.152904
5-fold validation: avg train rmse: 0.116069, avg valid rmse: 0.132194
```

```
lr: 0.01
weight_decay: 140
batch_size: 128
```



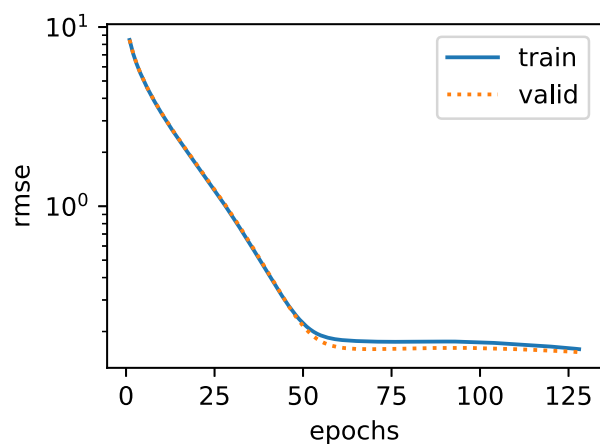
```
fold 0, train rmse: 0.157235, valid rmse: 0.151390
fold 1, train rmse: 0.146509, valid rmse: 0.169532
fold 2, train rmse: 0.152710, valid rmse: 0.156833
fold 3, train rmse: 0.155666, valid rmse: 0.146596
fold 4, train rmse: 0.148061, valid rmse: 0.175985
5-fold validation: avg train rmse: 0.152036, avg valid rmse: 0.160067
```

```
lr: 0.01
weight_decay: 150
batch_size: 64
```



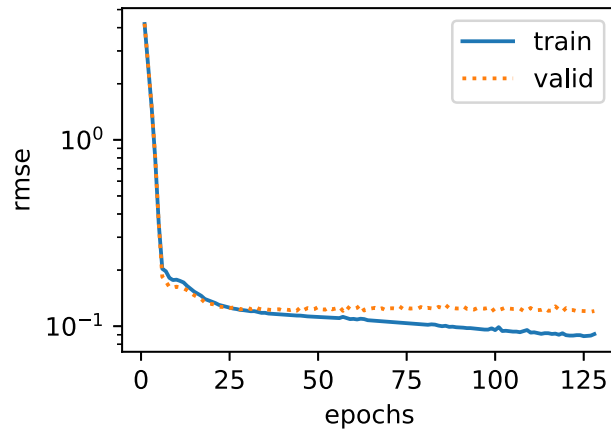
```
fold 0, train rmse: 0.119336, valid rmse: 0.125439
fold 1, train rmse: 0.112908, valid rmse: 0.135831
fold 2, train rmse: 0.116282, valid rmse: 0.134488
fold 3, train rmse: 0.121419, valid rmse: 0.112685
fold 4, train rmse: 0.109803, valid rmse: 0.151325
5-fold validation: avg train rmse: 0.115950, avg valid rmse: 0.131954
```

```
lr: 0.01
weight_decay: 150
batch_size: 128
```



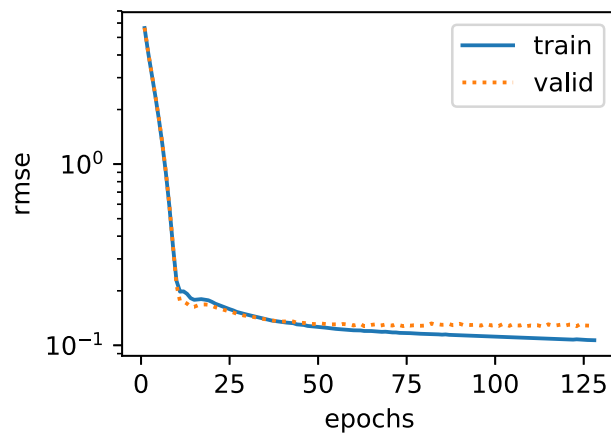
```
fold 0, train rmse: 0.159437, valid rmse: 0.153268
fold 1, train rmse: 0.143460, valid rmse: 0.165463
fold 2, train rmse: 0.149597, valid rmse: 0.154328
fold 3, train rmse: 0.157907, valid rmse: 0.147521
fold 4, train rmse: 0.147358, valid rmse: 0.175688
5-fold validation: avg train rmse: 0.151552, avg valid rmse: 0.159253
```

```
lr: 0.05
weight_decay: 100
batch_size: 64
```



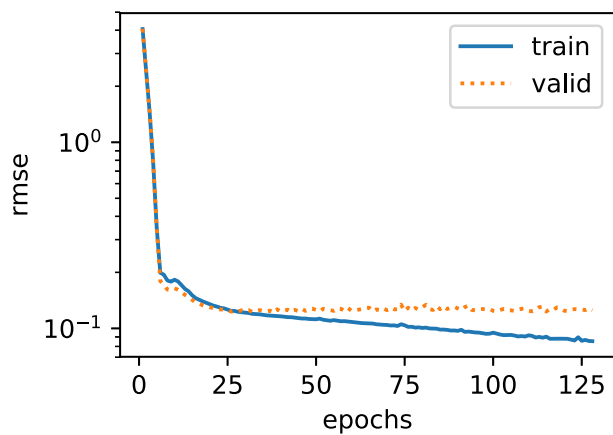
```
fold 0, train rmse: 0.090717, valid rmse: 0.120592
fold 1, train rmse: 0.088622, valid rmse: 0.140749
fold 2, train rmse: 0.086249, valid rmse: 0.133975
fold 3, train rmse: 0.091032, valid rmse: 0.117847
fold 4, train rmse: 0.088676, valid rmse: 0.146949
5-fold validation: avg train rmse: 0.089059, avg valid rmse: 0.132023
```

```
lr: 0.05
weight_decay: 100
batch_size: 128
```



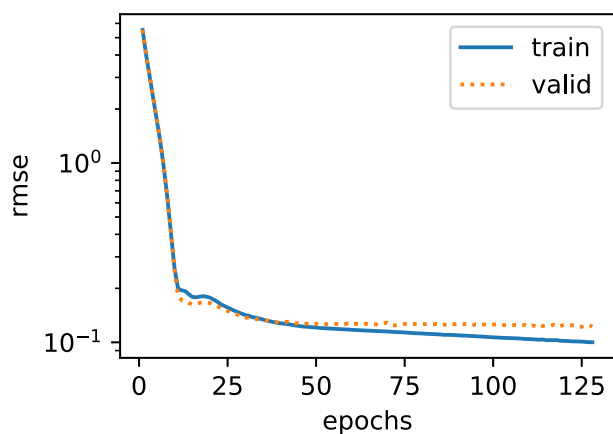
```
fold 0, train rmse: 0.106679, valid rmse: 0.127668
fold 1, train rmse: 0.091684, valid rmse: 0.139895
fold 2, train rmse: 0.097258, valid rmse: 0.133504
fold 3, train rmse: 0.099099, valid rmse: 0.112729
fold 4, train rmse: 0.096852, valid rmse: 0.149107
5-fold validation: avg train rmse: 0.098314, avg valid rmse: 0.132581
```

```
lr: 0.05
weight_decay: 110
batch_size: 64
```



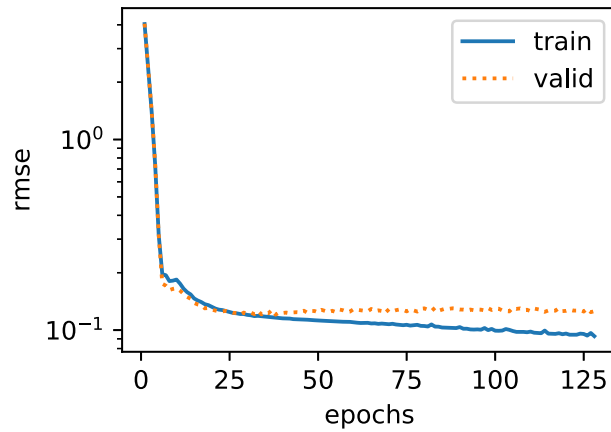
```
fold 0, train rmse: 0.085381, valid rmse: 0.125231
fold 1, train rmse: 0.083641, valid rmse: 0.138965
fold 2, train rmse: 0.081031, valid rmse: 0.132274
fold 3, train rmse: 0.088396, valid rmse: 0.113623
fold 4, train rmse: 0.082456, valid rmse: 0.147538
5-fold validation: avg train rmse: 0.084181, avg valid rmse: 0.131526
```

```
lr: 0.05
weight_decay: 110
batch_size: 128
```



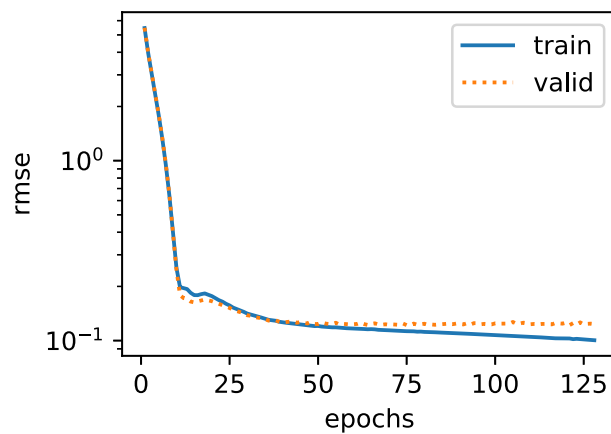
```
fold 0, train rmse: 0.100159, valid rmse: 0.122470
fold 1, train rmse: 0.094628, valid rmse: 0.139497
fold 2, train rmse: 0.095876, valid rmse: 0.133081
fold 3, train rmse: 0.100654, valid rmse: 0.111302
fold 4, train rmse: 0.095748, valid rmse: 0.146701
5-fold validation: avg train rmse: 0.097413, avg valid rmse: 0.130610
```

```
lr: 0.05
weight_decay: 130
batch_size: 64
```



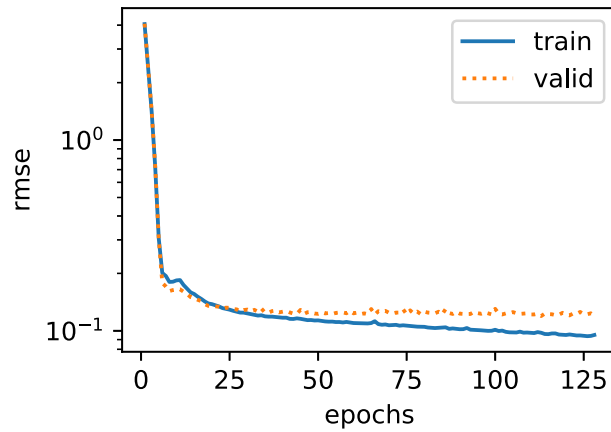
```
fold 0, train rmse: 0.092909, valid rmse: 0.125677
fold 1, train rmse: 0.084986, valid rmse: 0.138872
fold 2, train rmse: 0.082047, valid rmse: 0.134631
fold 3, train rmse: 0.095933, valid rmse: 0.114205
fold 4, train rmse: 0.090778, valid rmse: 0.148330
5-fold validation: avg train rmse: 0.089331, avg valid rmse: 0.132343
```

```
lr: 0.05
weight_decay: 130
batch_size: 128
```



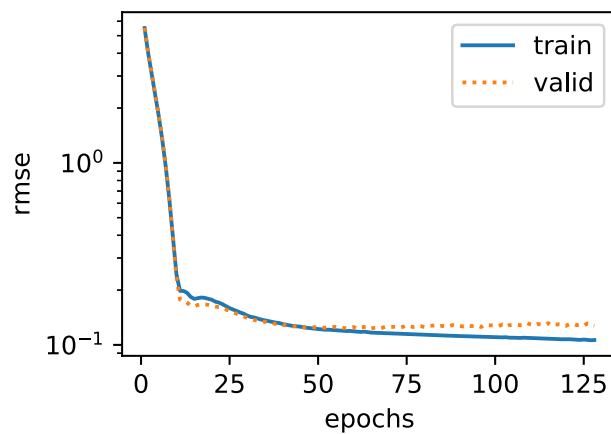
```
fold 0, train rmse: 0.100465, valid rmse: 0.123697
fold 1, train rmse: 0.094785, valid rmse: 0.138865
fold 2, train rmse: 0.093436, valid rmse: 0.131483
fold 3, train rmse: 0.108944, valid rmse: 0.114298
fold 4, train rmse: 0.101078, valid rmse: 0.148609
5-fold validation: avg train rmse: 0.099741, avg valid rmse: 0.131390
```

```
lr: 0.05
weight_decay: 140
batch_size: 64
```

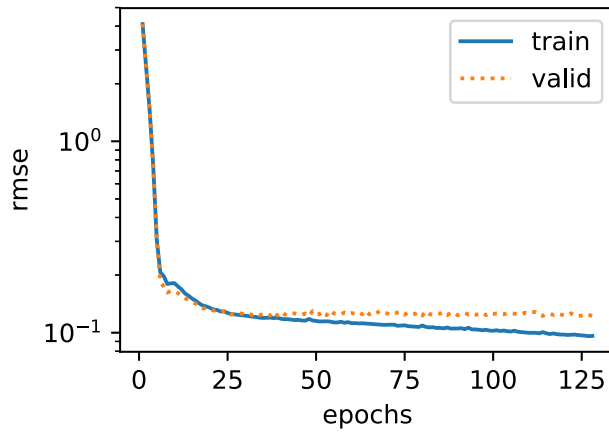
```
fold 0, train rmse: 0.095132, valid rmse: 0.119201
fold 1, train rmse: 0.086145, valid rmse: 0.139159
fold 2, train rmse: 0.086920, valid rmse: 0.132647
fold 3, train rmse: 0.095225, valid rmse: 0.112917
fold 4, train rmse: 0.091708, valid rmse: 0.148881
5-fold validation: avg train rmse: 0.091026, avg valid rmse: 0.130561
```

```
lr: 0.05
weight_decay: 140
batch_size: 128
```



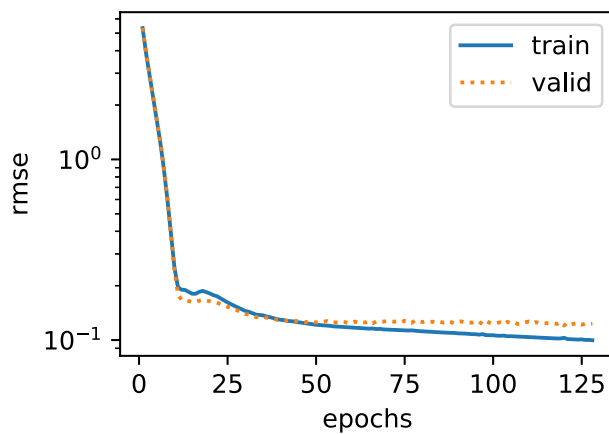
```
fold 0, train rmse: 0.106300, valid rmse: 0.127452
fold 1, train rmse: 0.095237, valid rmse: 0.139649
fold 2, train rmse: 0.091830, valid rmse: 0.132491
fold 3, train rmse: 0.106670, valid rmse: 0.114767
fold 4, train rmse: 0.093852, valid rmse: 0.147822
5-fold validation: avg train rmse: 0.098778, avg valid rmse: 0.132436
```

```
lr: 0.05
weight_decay: 150
batch_size: 64
```



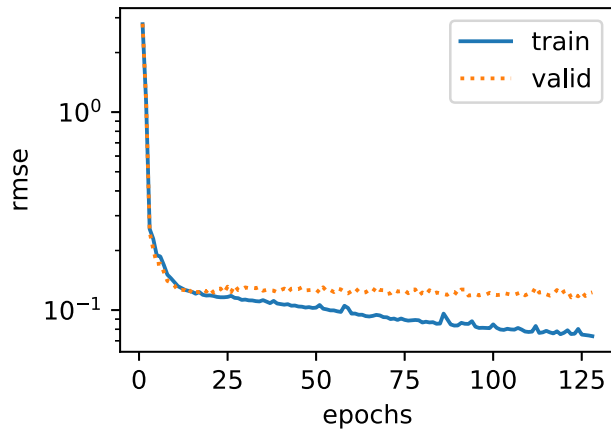
```
fold 0, train rmse: 0.096184, valid rmse: 0.122476
fold 1, train rmse: 0.090567, valid rmse: 0.140013
fold 2, train rmse: 0.092162, valid rmse: 0.134231
fold 3, train rmse: 0.098313, valid rmse: 0.113240
fold 4, train rmse: 0.091832, valid rmse: 0.149673
5-fold validation: avg train rmse: 0.093812, avg valid rmse: 0.131927
```

```
lr: 0.05
weight_decay: 150
batch_size: 128
```



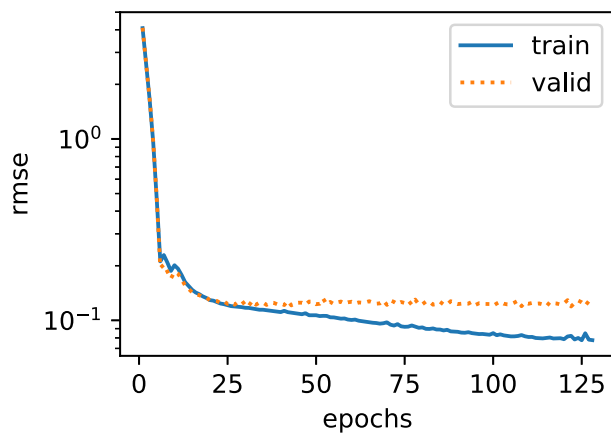
```
fold 0, train rmse: 0.099789, valid rmse: 0.123447
fold 1, train rmse: 0.094740, valid rmse: 0.136853
fold 2, train rmse: 0.091058, valid rmse: 0.132635
fold 3, train rmse: 0.100239, valid rmse: 0.109920
fold 4, train rmse: 0.097253, valid rmse: 0.147757
5-fold validation: avg train rmse: 0.096616, avg valid rmse: 0.130122
```

```
lr: 0.1
weight_decay: 100
batch_size: 64
```



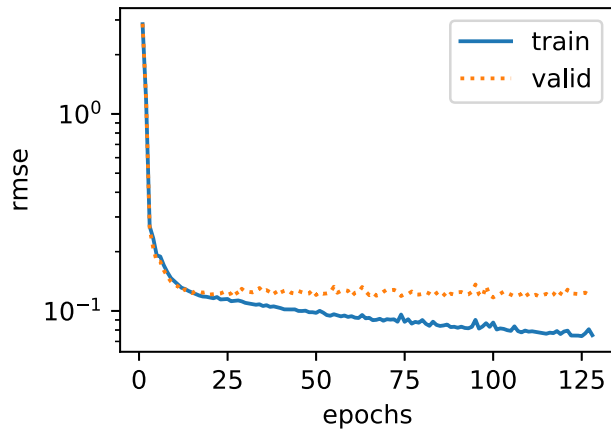
```
fold 0, train rmse: 0.073836, valid rmse: 0.122930
fold 1, train rmse: 0.069128, valid rmse: 0.139204
fold 2, train rmse: 0.067852, valid rmse: 0.134629
fold 3, train rmse: 0.083154, valid rmse: 0.118111
fold 4, train rmse: 0.074476, valid rmse: 0.145543
5-fold validation: avg train rmse: 0.073689, avg valid rmse: 0.132084
```

```
lr: 0.1
weight_decay: 100
batch_size: 128
```



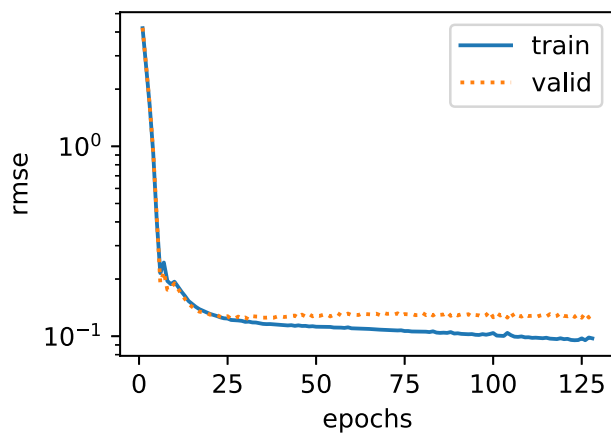
```
fold 0, train rmse: 0.077749, valid rmse: 0.123808
fold 1, train rmse: 0.081385, valid rmse: 0.138812
fold 2, train rmse: 0.066654, valid rmse: 0.135262
fold 3, train rmse: 0.086223, valid rmse: 0.113832
fold 4, train rmse: 0.077616, valid rmse: 0.143499
5-fold validation: avg train rmse: 0.077925, avg valid rmse: 0.131043
```

```
lr: 0.1
weight_decay: 110
batch_size: 64
```



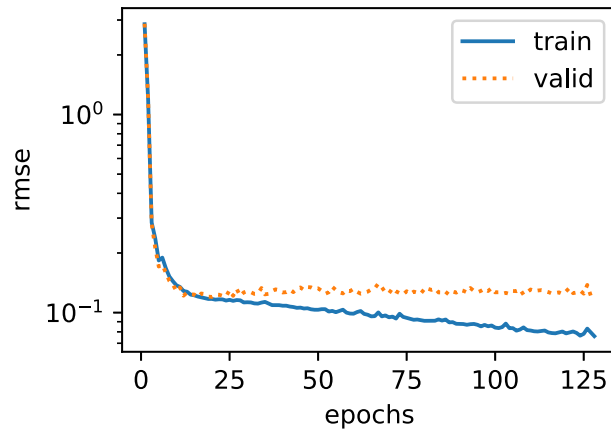
```
fold 0, train rmse: 0.075285, valid rmse: 0.120280
fold 1, train rmse: 0.068225, valid rmse: 0.140062
fold 2, train rmse: 0.071543, valid rmse: 0.132744
fold 3, train rmse: 0.069003, valid rmse: 0.110128
fold 4, train rmse: 0.076816, valid rmse: 0.151200
5-fold validation: avg train rmse: 0.072174, avg valid rmse: 0.130883
```

```
lr: 0.1
weight_decay: 110
batch_size: 128
```



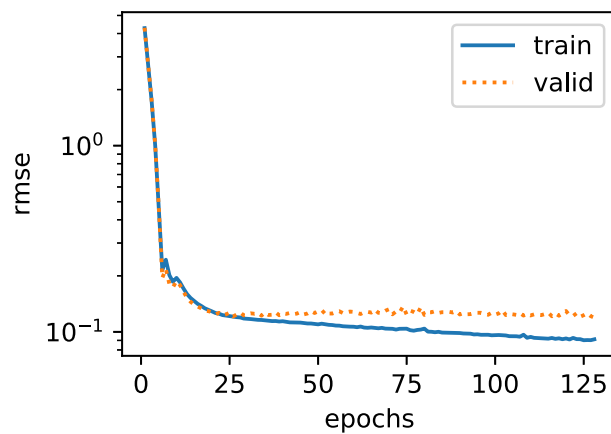
```
fold 0, train rmse: 0.097440, valid rmse: 0.135733
fold 1, train rmse: 0.078113, valid rmse: 0.138311
fold 2, train rmse: 0.076876, valid rmse: 0.132418
fold 3, train rmse: 0.079935, valid rmse: 0.112970
fold 4, train rmse: 0.079423, valid rmse: 0.148173
5-fold validation: avg train rmse: 0.082357, avg valid rmse: 0.133521
```

```
lr: 0.1
weight_decay: 130
batch_size: 64
```



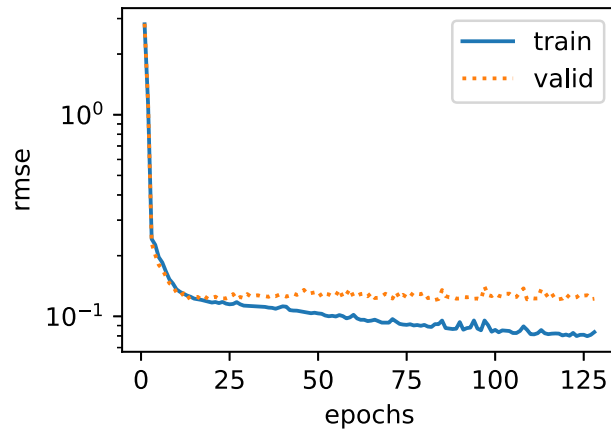
```
fold 0, train rmse: 0.076042, valid rmse: 0.122433
fold 1, train rmse: 0.075714, valid rmse: 0.142262
fold 2, train rmse: 0.073579, valid rmse: 0.140125
fold 3, train rmse: 0.074631, valid rmse: 0.112639
fold 4, train rmse: 0.077712, valid rmse: 0.151046
5-fold validation: avg train rmse: 0.075536, avg valid rmse: 0.133701
```

```
lr: 0.1
weight_decay: 130
batch_size: 128
```



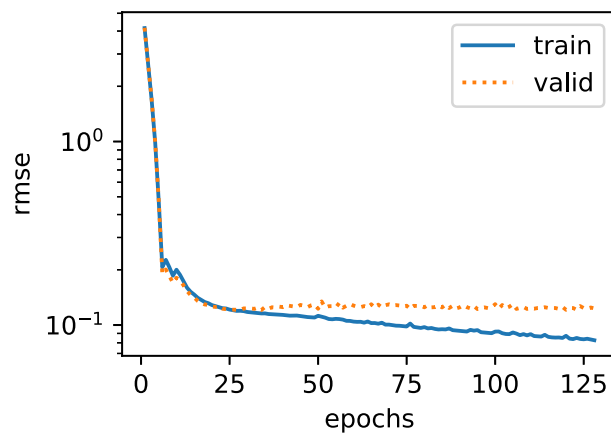
```
fold 0, train rmse: 0.091240, valid rmse: 0.119259
fold 1, train rmse: 0.083084, valid rmse: 0.139361
fold 2, train rmse: 0.079938, valid rmse: 0.132155
fold 3, train rmse: 0.085229, valid rmse: 0.114842
fold 4, train rmse: 0.083883, valid rmse: 0.148209
5-fold validation: avg train rmse: 0.084675, avg valid rmse: 0.130765
```

```
lr: 0.1
weight_decay: 140
batch_size: 64
```



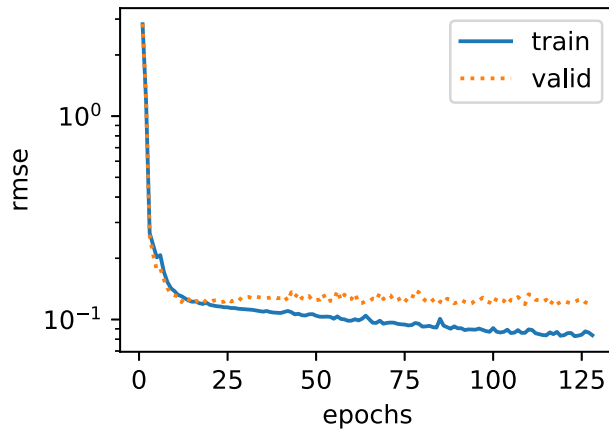
```
fold 0, train rmse: 0.083529, valid rmse: 0.122302
fold 1, train rmse: 0.079188, valid rmse: 0.147563
fold 2, train rmse: 0.067070, valid rmse: 0.133723
fold 3, train rmse: 0.077185, valid rmse: 0.117427
fold 4, train rmse: 0.078664, valid rmse: 0.149049
5-fold validation: avg train rmse: 0.077127, avg valid rmse: 0.134013
```

```
lr: 0.1
weight_decay: 140
batch_size: 128
```



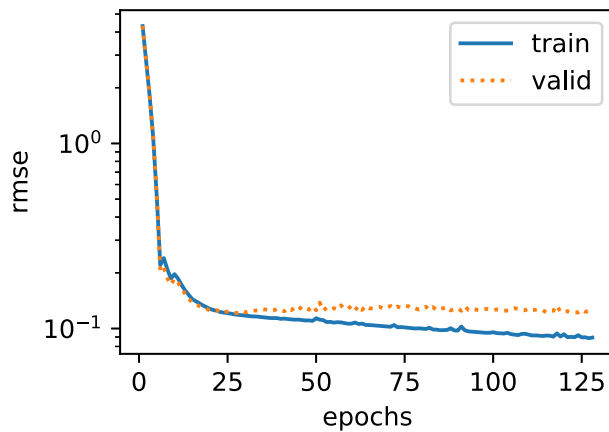
```
fold 0, train rmse: 0.082621, valid rmse: 0.123157
fold 1, train rmse: 0.077876, valid rmse: 0.139419
fold 2, train rmse: 0.078759, valid rmse: 0.133782
fold 3, train rmse: 0.083098, valid rmse: 0.112841
fold 4, train rmse: 0.084295, valid rmse: 0.144303
5-fold validation: avg train rmse: 0.081330, avg valid rmse: 0.130700
```

```
lr: 0.1
weight_decay: 150
batch_size: 64
```



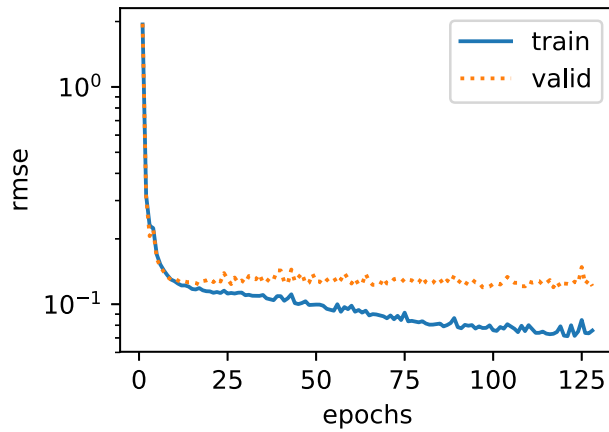
```
fold 0, train rmse: 0.083689, valid rmse: 0.124236
fold 1, train rmse: 0.077945, valid rmse: 0.140175
fold 2, train rmse: 0.082227, valid rmse: 0.139307
fold 3, train rmse: 0.090447, valid rmse: 0.117796
fold 4, train rmse: 0.087911, valid rmse: 0.149888
5-fold validation: avg train rmse: 0.084444, avg valid rmse: 0.134280
```

```
lr: 0.1
weight_decay: 150
batch_size: 128
```



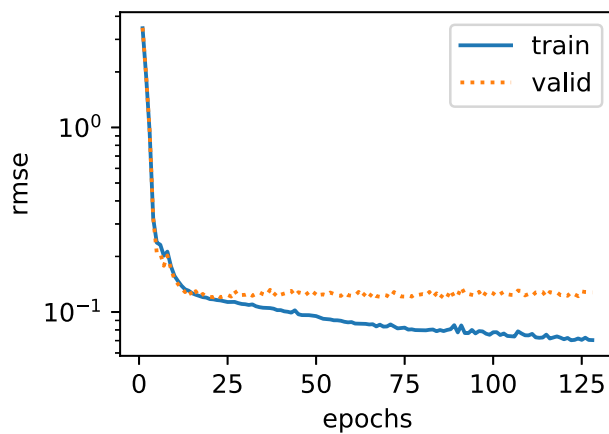
```
fold 0, train rmse: 0.089305, valid rmse: 0.126913
fold 1, train rmse: 0.084334, valid rmse: 0.139761
fold 2, train rmse: 0.080849, valid rmse: 0.132153
fold 3, train rmse: 0.090807, valid rmse: 0.114121
fold 4, train rmse: 0.087467, valid rmse: 0.146962
5-fold validation: avg train rmse: 0.086552, avg valid rmse: 0.131982
```

```
lr: 0.15
weight_decay: 100
batch_size: 64
```



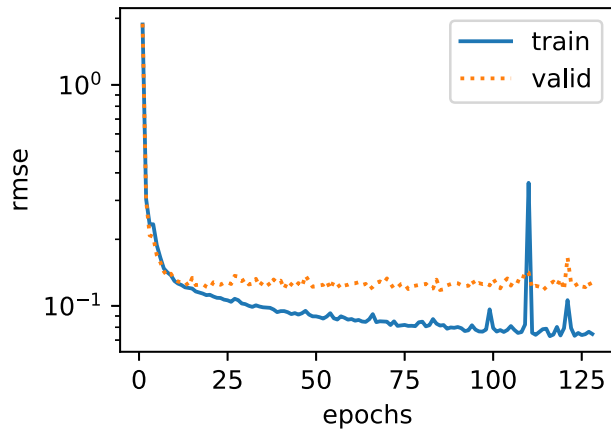
```
fold 0, train rmse: 0.075661, valid rmse: 0.121361
fold 1, train rmse: 0.064298, valid rmse: 0.142135
fold 2, train rmse: 0.060189, valid rmse: 0.136716
fold 3, train rmse: 0.066702, valid rmse: 0.108326
fold 4, train rmse: 0.074941, valid rmse: 0.152237
5-fold validation: avg train rmse: 0.068358, avg valid rmse: 0.132155
```

```
lr: 0.15
weight_decay: 100
batch_size: 128
```



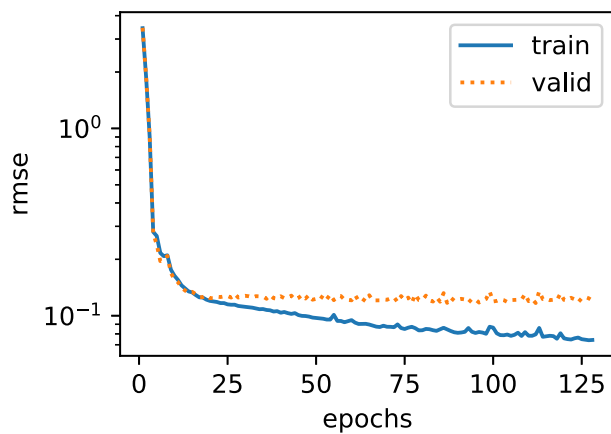
```
fold 0, train rmse: 0.070450, valid rmse: 0.127665
fold 1, train rmse: 0.061878, valid rmse: 0.142515
fold 2, train rmse: 0.066786, valid rmse: 0.136102
fold 3, train rmse: 0.084309, valid rmse: 0.107735
fold 4, train rmse: 0.082212, valid rmse: 0.144613
5-fold validation: avg train rmse: 0.073127, avg valid rmse: 0.131726
```

```
lr: 0.15
weight_decay: 110
batch_size: 64
```

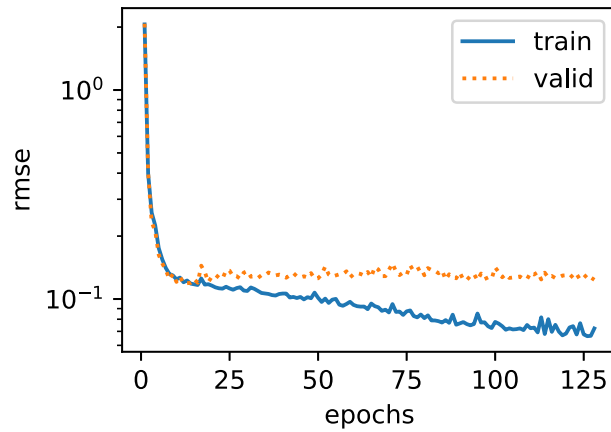
```
fold 0, train rmse: 0.074594, valid rmse: 0.119684
fold 1, train rmse: 0.070368, valid rmse: 0.154641
fold 2, train rmse: 0.087307, valid rmse: 0.137383
fold 3, train rmse: 0.069551, valid rmse: 0.121209
fold 4, train rmse: 0.067991, valid rmse: 0.147530
5-fold validation: avg train rmse: 0.073962, avg valid rmse: 0.136089
```

```
lr: 0.15
weight_decay: 110
batch_size: 128
```



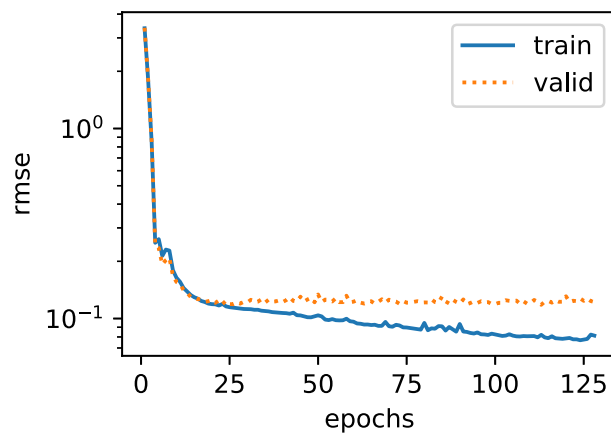
```
fold 0, train rmse: 0.074220, valid rmse: 0.120438
fold 1, train rmse: 0.066520, valid rmse: 0.141513
fold 2, train rmse: 0.065145, valid rmse: 0.133355
fold 3, train rmse: 0.074806, valid rmse: 0.114040
fold 4, train rmse: 0.071199, valid rmse: 0.145623
5-fold validation: avg train rmse: 0.070378, avg valid rmse: 0.130994
```

```
lr: 0.15
weight_decay: 130
batch_size: 64
```



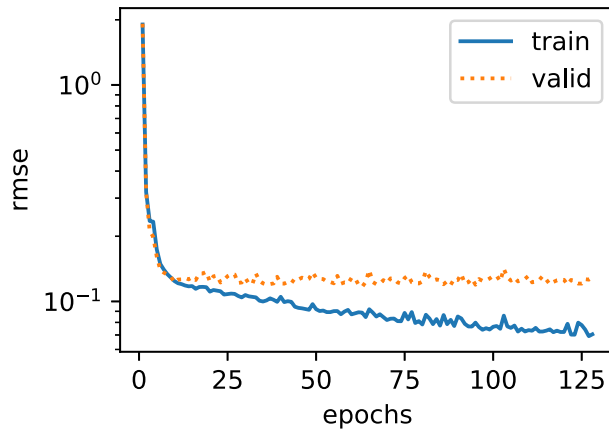
```
fold 0, train rmse: 0.072291, valid rmse: 0.123543
fold 1, train rmse: 0.064537, valid rmse: 0.142448
fold 2, train rmse: 0.084917, valid rmse: 0.138463
fold 3, train rmse: 0.071781, valid rmse: 0.114414
fold 4, train rmse: 0.074010, valid rmse: 0.151441
5-fold validation: avg train rmse: 0.073507, avg valid rmse: 0.134062
```

```
lr: 0.15
weight_decay: 130
batch_size: 128
```



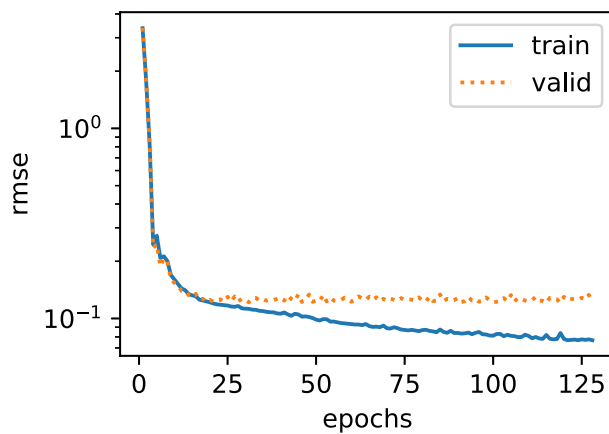
```
fold 0, train rmse: 0.081302, valid rmse: 0.122098
fold 1, train rmse: 0.073121, valid rmse: 0.139801
fold 2, train rmse: 0.067407, valid rmse: 0.132014
fold 3, train rmse: 0.077654, valid rmse: 0.116369
fold 4, train rmse: 0.078769, valid rmse: 0.148510
5-fold validation: avg train rmse: 0.075651, avg valid rmse: 0.131758
```

```
lr: 0.15
weight_decay: 140
batch_size: 64
```



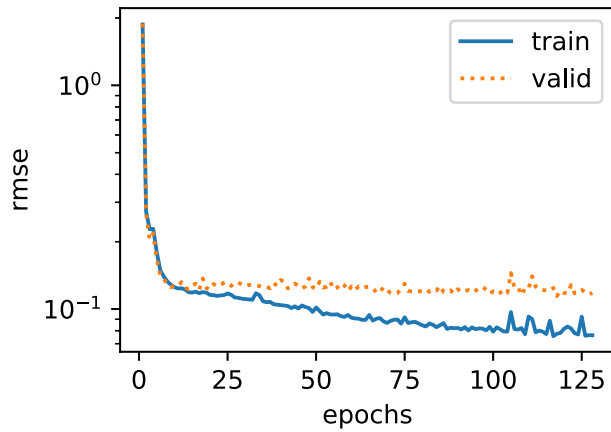
```
fold 0, train rmse: 0.070489, valid rmse: 0.129083
fold 1, train rmse: 0.070166, valid rmse: 0.142385
fold 2, train rmse: 0.070229, valid rmse: 0.132940
fold 3, train rmse: 0.079784, valid rmse: 0.125817
fold 4, train rmse: 0.068947, valid rmse: 0.143710
5-fold validation: avg train rmse: 0.071923, avg valid rmse: 0.134787
```

```
lr: 0.15
weight_decay: 140
batch_size: 128
```



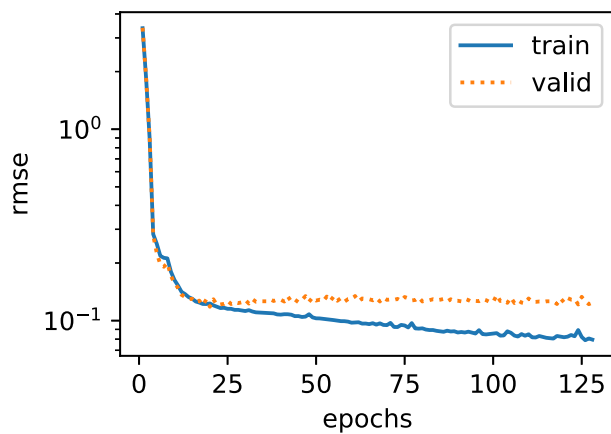
```
fold 0, train rmse: 0.076840, valid rmse: 0.123117
fold 1, train rmse: 0.072046, valid rmse: 0.142914
fold 2, train rmse: 0.070099, valid rmse: 0.141223
fold 3, train rmse: 0.086458, valid rmse: 0.114258
fold 4, train rmse: 0.080255, valid rmse: 0.145325
5-fold validation: avg train rmse: 0.077140, avg valid rmse: 0.133367
```

```
lr: 0.15
weight_decay: 150
batch_size: 64
```



```
fold 0, train rmse: 0.076346, valid rmse: 0.118105
fold 1, train rmse: 0.065219, valid rmse: 0.144467
fold 2, train rmse: 0.066819, valid rmse: 0.132092
fold 3, train rmse: 0.065503, valid rmse: 0.116742
fold 4, train rmse: 0.071639, valid rmse: 0.153782
5-fold validation: avg train rmse: 0.069105, avg valid rmse: 0.133037
```

```
lr: 0.15
weight_decay: 150
batch_size: 128
```

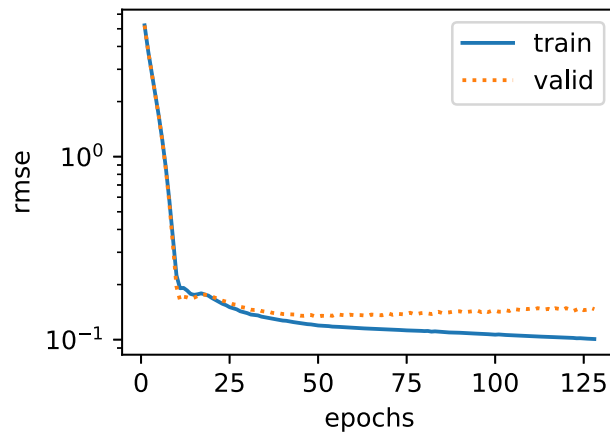


```
fold 0, train rmse: 0.079596, valid rmse: 0.124050
fold 1, train rmse: 0.070639, valid rmse: 0.143771
fold 2, train rmse: 0.069002, valid rmse: 0.135553
fold 3, train rmse: 0.082669, valid rmse: 0.115354
fold 4, train rmse: 0.078404, valid rmse: 0.149595
5-fold validation: avg train rmse: 0.076062, avg valid rmse: 0.133665
```

```
In [15]: print(best_l)
         print(best_param)

0.13012246936559677
(0.05, 150, 128)
```

```
In [23]: k, num_epochs, lr, weight_decay, batch_size = 10, 128, 0.05, 150, 128
train_l, valid_l = k_fold(k, train_features, train_labels, num_epochs, lr,
                           weight_decay, batch_size)
print('%d-fold validation: avg train rmse: %f, avg valid rmse: %f'
      % (k, train_l, valid_l))
```



```
fold 0, train rmse: 0.100783, valid rmse: 0.146684
fold 1, train rmse: 0.103865, valid rmse: 0.099752
fold 2, train rmse: 0.102268, valid rmse: 0.110464
fold 3, train rmse: 0.096494, valid rmse: 0.168207
fold 4, train rmse: 0.096850, valid rmse: 0.152871
fold 5, train rmse: 0.099278, valid rmse: 0.109438
fold 6, train rmse: 0.102910, valid rmse: 0.122301
fold 7, train rmse: 0.103905, valid rmse: 0.103917
fold 8, train rmse: 0.098059, valid rmse: 0.164531
fold 9, train rmse: 0.100922, valid rmse: 0.130204
10-fold validation: avg train rmse: 0.100533, avg valid rmse: 0.130837
```

You will notice that sometimes the number of training errors for a set of hyper-parameters can be very low, while the number of errors for the K -fold cross validation may be higher. This is most likely a consequence of overfitting. Therefore, when we reduce the amount of training errors, we need to check whether the amount of errors in the k -fold cross-validation have also been reduced accordingly.

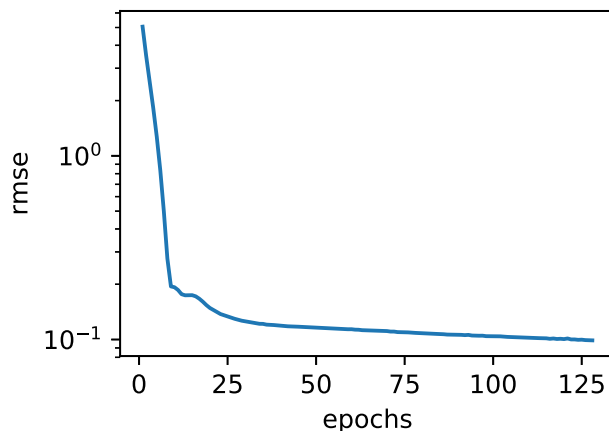
Predict and Submit

Now that we know what a good choice of hyperparameters should be, we might as well use all the data to train on it (rather than just $1 - 1/k$ of the data that is used in the crossvalidation slices). The model that we obtain in this way can then be applied to the test set. Saving the estimates in a CSV file will simplify uploading the results to Kaggle.

```
In [20]: def train_and_pred(train_features, test_feature, train_labels, test_data,
                           num_epochs, lr, weight_decay, batch_size):
    net = get_net()
    train_ls, _ = train(net, train_features, train_labels, None, None,
                        num_epochs, lr, weight_decay, batch_size)
    d2l.semilogy(range(1, num_epochs + 1), train_ls, 'epochs', 'rmse')
    print('train rmse %f' % train_ls[-1])
    # apply the network to the test set
    preds = net(test_features)
    preds = preds.exp()
    preds = preds.asnumpy()
    # reformat it for export to Kaggle
    test_data['SalePrice'] = pd.Series(preds.reshape(1, -1)[0])
    submission = pd.concat([test_data['Id'], test_data['SalePrice']], axis=1)
    submission.to_csv('submission.csv', index=False)
```

Let's invoke the model. A good sanity check is to see whether the predictions on the test set resemble those of the k-fold crossvalidation process. If they do, it's time to upload them to Kaggle.

```
In [21]: train_and_pred(train_features, test_features, train_labels, test_data,
                        num_epochs, lr, weight_decay, batch_size)
```



```
train rmse 0.098946
```

A file, `submission.csv` will be generated by the code above (CSV is one of the file formats accepted by Kaggle). Next, we can submit our predictions on Kaggle and compare them to the actual house price (label) on the testing data set, checking for errors. The steps are quite simple:

- Log in to the Kaggle website and visit the House Price Prediction Competition page.
- Click the “Submit Predictions” or “Late Submission” button on the right.
- Click the “Upload Submission File” button in the dashed box at the bottom of the page and select the prediction file you wish to upload.
- Click the “Make Submission” button at the bottom of the page to view your results.

Hints

1. Can you improve your model by minimizing the log-price directly? What happens if you try to predict the log price rather than the price?
2. Is it always a good idea to replace missing values by their mean? Hint - can you construct a situation where the values are not missing at random?
3. Find a better representation to deal with missing values. Hint - What happens if you add an indicator variable?
4. Improve the score on Kaggle by tuning the hyperparameters through k-fold crossvalidation.
5. Improve the score by improving the model (layers, regularization, dropout).
6. What happens if we do not standardize the continuous numerical features like we have done in this section?

Note for converting this notebook into PDF. If you use 'File -> Download as -> PDF', you may get the error that svg cannot be converted because inkscape is not installed and cannot find PNG images. The easiest way is printing this notebook as a PDF in your browser. Or, you can install inkscape to convert SVG (On macOS, you may `brew cask install xquartz inkscape`, on Ubuntu, you may `sudo apt-get install inkscape`) and change the image URL to local filenames.