

Homework 2 Solution: Mathematics of Deep Learning

(EN 580.745)

Instructor: René Vidal, Biomedical Engineering, Johns Hopkins University

1. Balanced weight matrices and positive homogeneity. Consider the single hidden layer neural network training problem

$$\underset{\mathbf{U}, \mathbf{V}}{\text{minimize}} \sum_{i=1}^N \mathcal{L}(y_i, \mathbf{U}\psi(\mathbf{V}^\top \mathbf{x}_i)), \quad (1)$$

where $\{(\mathbf{x}_i, y_i)\}_{i=1}^N$ is a fixed training dataset, \mathcal{L} is an arbitrary convex and non-negative loss function (e.g. logistic loss), and ψ denotes the pointwise ReLU activation $\psi(z) = \max(z, 0)$.

- (a) **(5 points)** Prove that if a global minimizer to (1) exists, then the set of minimizers is unbounded.
- (b) **(5 points)** Now suppose we add the following regularization

$$\underset{\mathbf{U}, \mathbf{V}}{\text{minimize}} \sum_{i=1}^N \mathcal{L}(y_i, \mathbf{U}\psi(\mathbf{V}^\top \mathbf{x}_i)) + \frac{\lambda}{2}(\|\mathbf{U}\|_F^2 + \|\mathbf{V}\|_F^2) \quad (2)$$

Prove that any global minimizer to (2) must have balanced weight matrices. I.e. $\|\mathbf{U}\|_F = \|\mathbf{V}\|_F$. Show that as a consequence, the set of minimizers to (2) is bounded.

- (c) **(5 points)** Finally, suppose we instead use a regularizer with *unbalanced* degrees of homogeneity

$$\underset{\mathbf{U}, \mathbf{V}}{\text{minimize}} \sum_{i=1}^N \mathcal{L}(y_i, \mathbf{U}\psi(\mathbf{V}^\top \mathbf{x}_i)) + \frac{\lambda}{2}(\|\mathbf{U}\|_F + \|\mathbf{V}\|_F^2) \quad (3)$$

Prove that by contrast, any global minimizer to (3) will typically *not* have balanced weight norms. I.e. will not satisfy $\|\mathbf{U}\|_F = \|\mathbf{V}\|_F$. Specifically, let $(\mathbf{U}^*, \mathbf{V}^*)$, be a global minimizer and let $\|\mathbf{U}^*\|_F \|\mathbf{V}^*\|_F = c$. Assume $c > 0$. Show that $\mathbf{U}^*, \mathbf{V}^*$ will be balanced if and only if $c = 1/4$.

- (d) **(5 points)** Give some intuition in 1-3 sentences why a bounded set of minimizers, or balanced weight matrices might be beneficial for optimization (take your pick).

Solution.

- (a) Let $(\mathbf{U}^*, \mathbf{V}^*)$ be a global minimizer of (1) and assume $\mathbf{U}^* \neq \mathbf{0}$. Let $\alpha > 0$ and note by positive homogeneity,

$$(\alpha \mathbf{U}^*)\psi((1/\alpha)(\mathbf{V}^*)^\top \mathbf{x}) = \mathbf{U}^*\psi((\mathbf{V}^*)^\top \mathbf{x}) \quad (4)$$

for all \mathbf{x} . Thus, $(\alpha \mathbf{U}^*, (1/\alpha)\mathbf{V}^*)$ must also be a global minimizer. Taking α arbitrarily large shows the set of minimizers must be unbounded. If $\mathbf{U}^* = \mathbf{0}$, then similarly $(\mathbf{0}, \alpha \mathbf{V})$ is a minimizer for all $\alpha > 0$ and any \mathbf{V} .

- (b) Fix some arbitrary factors (\mathbf{U}, \mathbf{V}) . First, if \mathbf{U} or \mathbf{V} are zero, then it's clear that (\mathbf{U}, \mathbf{V}) is optimal only when $\mathbf{U} = \mathbf{V} = 0$. So assume $\mathbf{U} \neq \mathbf{0} \neq \mathbf{V}$. Consider the scale optimization problem

$$\underset{\alpha > 0}{\text{minimize}} \sum_{i=1}^N \mathcal{L}(y_i, (\alpha \mathbf{U})\psi(((1/\alpha)\mathbf{V})^\top \mathbf{x}_i)) + \frac{\lambda}{2}(\|\alpha \mathbf{U}\|_F^2 + \|(1/\alpha)\mathbf{V}\|_F^2). \quad (5)$$

By the same scale invariance principle used in (a), we can ignore the loss and instead solve

$$\underset{\alpha > 0}{\text{minimize}} \frac{1}{2}(\alpha^2 \|\mathbf{U}\|_F^2 + (1/\alpha)^2 \|\mathbf{V}\|_F^2). \quad (6)$$

Note that (6) is strongly convex over $(0, \infty]$ and tends to $+\infty$ as $\alpha \rightarrow 0$. Thus, the minimum is achieved when the derivative equals zero, i.e. when $\alpha^4 = \frac{\|\mathbf{V}\|_F^2}{\|\mathbf{U}\|_F^2}$. It follows that (\mathbf{U}, \mathbf{V}) is optimal only when $\alpha = 1$, which in turn implies that $\|\mathbf{U}\|_F = \|\mathbf{V}\|_F$.

It is also true that the set of minimizers must be bounded when this regularization is included. This does not rely on the optimal weight matrices being balanced, however. Instead, it is clear that the set of minimizers will be bounded as long as the loss is non-negative and some coercive regularization¹ is included. In particular, the regularizer in (c) also guarantees bounded minimizers.

- (c) As in (b), fix nonzero (\mathbf{U}, \mathbf{V}) and consider the scale optimization problem, adapted to this new regularizer

$$\underset{\alpha > 0}{\text{minimize}} \quad \frac{1}{2}(\alpha \|\mathbf{U}\|_F + (1/\alpha)^2 \|\mathbf{V}\|_F^2). \quad (7)$$

Now the optimal α must satisfy $\alpha^3 = \frac{2\|\mathbf{V}\|_F^2}{\|\mathbf{U}\|_F}$. For the optimal $\mathbf{U}^* \neq \mathbf{0} \neq \mathbf{V}^*$, we must have $\alpha = 1$, hence

$$\begin{aligned} \|\mathbf{U}^*\|_F &= 2\|\mathbf{V}^*\|_F^2 \Rightarrow (c/2)^{\frac{1}{3}} = \|\mathbf{V}^*\|_F \\ &\Rightarrow \|\mathbf{U}^*\|_F = 2(c/2)^{\frac{2}{3}}. \end{aligned} \quad (8)$$

After some manipulation, it follows that $\|\mathbf{U}^*\|_F = \|\mathbf{V}^*\|_F$ if and only if $c = 1/4$.

- (d) Iterative optimization methods such as gradient descent take longer to converge to minimizers that have very large norm (hence likely to be very far from initialization). Balanced weight matrices result in “better conditioned” gradient steps, making it easier, for example, to choose an appropriate step size.

□

2. Benefit of over-parameterization on optimization. In this exercise you will attempt to replicate an experiment from [1] illustrating the benefit of over-parameterization for optimization (Figure 1). To save effort and keep the code short, you should implement the experiment using PyTorch, Tensorflow, or some other deep learning framework. (I.e. it would be better not to implement everything from scratch in MATLAB.)

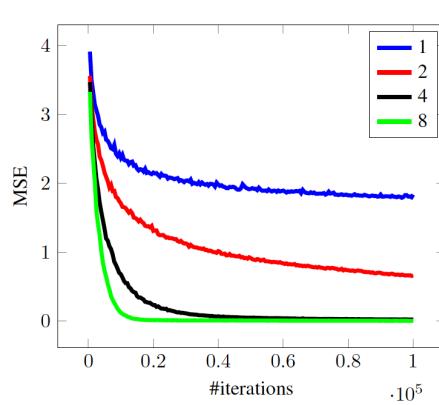


Figure 1: Effect of over-parameterization on single hidden layer network training from [1]. The data are synthetically generated from a network with fixed “planted” weights, and hidden layer size r_0 . Mean squared error (MSE) is shown as a function of iteration for varying “over-parameterization levels”. (I.e. a value of 2 means the trained network had a hidden layer of size $2r_0$.)

- (a) (**5 points**) Implement a synthetic dataset where samples (\mathbf{x}, \mathbf{y}) are drawn from a single hidden layer ReLU network with planted weights. Specifically, generate random Gaussian weight matrices $\mathbf{U}_0 \in \mathbb{R}^{n \times r_0}$, $\mathbf{V}_0 \in \mathbb{R}^{D \times r_0}$ with $(\mathbf{U}_0)_{ij} \sim \mathcal{N}(0, 1/r_0)$, $(\mathbf{V}_0)_{ij} \sim \mathcal{N}(\mathbf{0}, 1/D)$. Generate samples using these fixed “planted” weights as follows

$$\mathbb{R}^D \ni \mathbf{x} \sim \mathcal{N}(\mathbf{0}, \mathbf{I}) \quad \mathbb{R}^n \ni \mathbf{y} = \mathbf{U}_0 \psi(\mathbf{V}_0^\top \mathbf{x}), \quad (9)$$

where ψ again denotes the ReLU activation. The specifics of the implementation will vary depending on which framework you use. But the key requirement is that the dataset has no fixed size. Every mini-batch drawn from

¹https://en.wikipedia.org/wiki/Coercive_function

the dataset should contain fresh samples. To show that your code works, implement a short test with $D = 150$, $r_0 = 60$, and $n = 10$ that generates 10 mini-batches of size 50 and prints the average $\|\mathbf{y}_i\|_2^2$ value for each.

- (b) **(5 points)** Implement SGD to optimize the following least-squares objective for single hidden layer networks using the planted network dataset from part (a)

$$\underset{\mathbf{U}, \mathbf{V}}{\text{minimize}} \quad f(\mathbf{U}, \mathbf{V}) \triangleq \mathbb{E}_{(\mathbf{x}, \mathbf{y})} \frac{1}{2} \|\mathbf{y} - \mathbf{U}\psi(\mathbf{V}^\top \mathbf{x})\|_2^2. \quad (10)$$

Your algorithm should have the following parameters: an initial learning rate $\eta > 0$, and a fixed batch size $m > 0$. You should implement a learning rate schedule that decreases η by 0.5 every 50 epochs, where each “epoch” contains 10K samples. Log the average mini-batch objective value on each epoch.

- (c) **(10 points)** Set $D = 150$, $n = 10$, $r_0 = 60$. Set the batch size $m = 10$ and hidden layer size $r = 2r_0 = 120$. Consider 4 choices for the initial learning rate η on a log scale, $\eta \in \{10^{-4}, 10^{-3}, \dots, 0.1\}$. Run the optimization for 20 epochs. Plot the average objective value as a function of epoch for each choice of η . Which value yields convergence to the lowest objective value?
- (d) **(10 points)** Using the same problem setting and the optimal η from part (c), train networks of varying sizes $r/r_0 \in \{1, 2, 4, 8\}$ for 200 epochs. Again, plot the average objective value as a function of epoch, but now for each hidden layer size. How does your result compare to that of Livni & co-authors (Figure 1)?
- (e) **(bonus)** If your result is different from Livni et al., experiment with the various problem settings (e.g. scaling of weights, η , batch size, choice of optimization algorithm) to see if you can replicate their observed pattern. Conversely, if you did replicate their result, see if you can get fast convergence to near zero error for all choices of r under a different problem setting.

References

- [1] R. Livni, S. Shalev-Shwartz, and O. Shamir. On the computational efficiency of training neural networks. In *Advances in neural information processing systems*, pages 855–863, 2014. 2

```
In [1]: import time
from itertools import product

import numpy as np
import torch
from matplotlib import pyplot as plt

from torch import nn
from torch.nn import functional as F
from torch.utils.data import Dataset, DataLoader

torch.set_num_threads(1)
```

(a) Model and dataset

First, we implement a single hidden layer ReLU network model with D inputs, n outputs, and r hidden units. Note that to initialize, the first layer weights are sampled from $\mathcal{N}(0, 1/D)$ while the second layer weights are sampled from $\mathcal{N}(0, 1/r)$.

```
In [2]: class SinglelayerNet(nn.Module):
    """Single hidden layer network."""
    def __init__(self, D, n, r, frozen=False, init_scale=1.0):
        super(SinglelayerNet, self).__init__()

        self.D, self.n, self.r = D, n, r
        self.frozen = frozen
        self.init_scale = np.sqrt(init_scale)

        self.fc1 = nn.Linear(D, r, bias=False)
        self.fc2 = nn.Linear(r, n, bias=False)

        if frozen:
            for p in self.parameters():
                p.requires_grad = False

        self.reset_parameters()

    def reset_parameters(self):
        self.fc1.weight.data.normal_(std=self.init_scale / np.sqrt(self.D))
        self.fc2.weight.data.normal_(std=self.init_scale / np.sqrt(self.r))

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

Next we define a dataset for drawing fresh samples from a fixed network with frozen weights.

```
In [3]: class SynthPlantedDataset(Dataset):
    def __init__(self, N, D, n, r0, seed=2019):
        self.N, self.D, self.n, self.r0 = N, D, n, r0
        self.seed = seed

        if seed is not None:
            torch.manual_seed(seed)

        self.planted_net = SingleLayerNet(D, n, r0, frozen=True)

    def __len__(self):
        return self.N

    def __getitem__(self, _):
        # draw a fresh sample on every call to getitem
        x = torch.randn(self.D)
        y = self.planted_net(x.view(1, -1)).view(-1)
        return x, y
```

We generate a particular dataset instance with $N = 10000$, $D = 100$, $n = 10$, and $r_0 = 60$. We further construct a "data loader" that draws mini-batches of size $m = 50$. To test the dataset, we sample 10 mini-batches and print the average $\|y_i\|_2^2$ value from each.

```
In [4]: synth_ds = SynthPlantedDataset(10000, 150, 10, 60)
synth_loader = DataLoader(synth_ds, batch_size=50, shuffle=False)

In [5]: def test_synth_ds(synth_loader):
    synth_iter = iter(synth_loader)

    for ii in range(10):
        x, y = next(synth_iter)
        print('batch {}, ||y_i||^2={:.3f}'.format(ii, y.pow(2).sum(dim=1).mean()))

In [6]: test_synth_ds(synth_loader)
batch 0, ||y_i||^2=4.016
batch 1, ||y_i||^2=3.829
batch 2, ||y_i||^2=4.111
batch 3, ||y_i||^2=4.074
batch 4, ||y_i||^2=4.192
batch 5, ||y_i||^2=3.957
batch 6, ||y_i||^2=4.392
batch 7, ||y_i||^2=3.782
batch 8, ||y_i||^2=3.943
batch 9, ||y_i||^2=3.767
```

(b) SGD training

Next we implement SGD to train a network to regress the samples drawn from this dataset. i.e., to mirror the mapping of this "planted" network.

First, we copy in some useful training utilities from elsewhere.

```

In [7]: class AverageMeter(object):
    """Computes and stores the average and current value.
    """
    def __init__(self):
        self.reset()

    def reset(self):
        self.val = 0
        self.avg = 0
        self.sum = 0
        self.count = 0

    def update(self, val, n=1):
        self.val = val
        self.sum += val * n
        self.count += n
        self.avg = self.sum / self.count

In [8]: def get_learning_rate(optimizer):
    return np.median([param_group['lr']
                     for param_group in optimizer.param_groups])

Next, we define the mean-squared-error (MSE) objective function.

In [9]: def objfun(y, yhat):
    return (y - yhat).pow(2).sum(dim=1).mul(0.5).mean()

Our SGD training function initializes a new network and trains using SGD for a fixed number of epochs. The average objective value is recorded from each epoch. Moreover, the learning rate is decreased by 1/2 every 50 epochs.

In [10]: def train_single_layer_net(synth_ds, r, init_lr, batch_size=100, epochs=100, init_scale=0.1, seed=1904):
    if seed is not None:
        torch.manual_seed(seed)

    synth_loader = DataLoader(synth_ds, batch_size=batch_size, shuffle=False)
    # note that a smaller scale is often used to initialize the model weights vs the dataset weights
    model = SingleLayerNet(synth_ds.D, synth_ds.n, r, init_scale=init_scale)
    optimizer = torch.optim.SGD(model.parameters(), init_lr, momentum=0)
    scheduler = torch.optim.lr_scheduler.StepLR(optimizer, 50, gamma=0.5)

    obj = np.ones(epochs)*np.nan
    err = None
    for epoch in range(epochs):
        tic = time.time()
        epoch_obj = AverageMeter()
        try:
            for (x, y) in synth_loader:
                optimizer.zero_grad()
                yhat = model(x)
                batch_obj = objfun(y, yhat)
                if torch.isnan(batch_obj.data):
                    raise RuntimeError('Divergence!')

                batch_obj.backward()
                optimizer.step()
                epoch_obj.update(batch_obj.item(), x.size(0))
        except RuntimeError as e:
            err = e
            print(e)
        obj[epoch] = epoch_obj.avg
        print('Epoch {}, time={:.3f}s obj={:.3e}, lr={:.3e}'.format(
            epoch, time.time() - tic, epoch_obj.avg, get_learning_rate(optimizer)))
    scheduler.step()

    if err is not None or epoch_obj.avg <= 1e-6:
        break
    return obj

In [11]: def run_experiment(rs, init_lrs, batch_sizes, epochs=500, init_scale=0.1):
    obj_dict = dict()
    for r, init_lr, bs in product(rs, init_lrs, batch_sizes):
        print('\nRunning trial r={}, init_lr={:.2e}, bs={}\n'.format(r, init_lr, bs))
        obj_dict[(r, init_lr, bs)] = train_single_layer_net(synth_ds, r, init_lr,
                                                             batch_size=bs, epochs=epochs,
                                                             init_scale=init_scale)

    return obj_dict

```

(c) Learning rate test

Now for the first test, we search for a good initial learning rate for an intermediate size net $r = 120$.

```
In [12]: obj_dict_lr_test = run_experiment([120], [1e-4, 1e-3, 1e-2, 1e-1], [10], epochs=20)
```

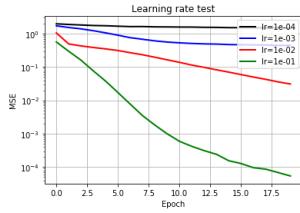
```
Running trial r=120, init_lr=1.00e-04, bs=10
Epoch 0, time=1.1598 obj=1.975e+00, lr=1.000e-04
Epoch 1, time=1.1238 obj=1.590e+00, lr=1.000e-04
Epoch 2, time=1.1308 obj=1.820e+00, lr=1.000e-04
Epoch 3, time=1.1264 obj=1.756e+00, lr=1.000e-04
Epoch 4, time=1.1168 obj=1.729e+00, lr=1.000e-04
Epoch 5, time=1.1158 obj=1.678e+00, lr=1.000e-04
Epoch 6, time=1.1208 obj=1.633e+00, lr=1.000e-04
Epoch 7, time=1.1208 obj=1.639e+00, lr=1.000e-04
Epoch 8, time=1.1188 obj=1.603e+00, lr=1.000e-04
Epoch 9, time=1.1258 obj=1.595e+00, lr=1.000e-04
Epoch 10, time=1.1228 obj=1.580e+00, lr=1.000e-04
Epoch 11, time=1.1378 obj=1.557e+00, lr=1.000e-04
Epoch 12, time=1.1868 obj=1.550e+00, lr=1.000e-04
Epoch 13, time=1.1618 obj=1.544e+00, lr=1.000e-04
Epoch 14, time=1.1218 obj=1.539e+00, lr=1.000e-04
Epoch 15, time=1.1278 obj=1.512e+00, lr=1.000e-04
Epoch 16, time=1.1318 obj=1.510e+00, lr=1.000e-04
Epoch 17, time=1.1298 obj=1.494e+00, lr=1.000e-04
Epoch 18, time=1.1288 obj=1.490e+00, lr=1.000e-04
Epoch 19, time=1.1268 obj=1.460e+00, lr=1.000e-04

Running trial r=120, init_lr=1.00e-03, bs=10
Epoch 0, time=1.1258 obj=1.723e+00, lr=1.000e-03
Epoch 1, time=1.1258 obj=1.531e+00, lr=1.000e-03
Epoch 2, time=1.1438 obj=1.395e+00, lr=1.000e-03
Epoch 3, time=1.1468 obj=1.227e+00, lr=1.000e-03
Epoch 4, time=1.1388 obj=1.190e+00, lr=1.000e-03
Epoch 5, time=1.1308 obj=0.998e+01, lr=1.000e-03
Epoch 6, time=1.1478 obj=7.588e+00, lr=1.000e-03
Epoch 7, time=1.1438 obj=6.797e-01, lr=1.000e-03
Epoch 8, time=1.1438 obj=6.086e-01, lr=1.000e-03
Epoch 9, time=1.1728 obj=5.604e-01, lr=1.000e-03
Epoch 10, time=1.1768 obj=5.315e-01, lr=1.000e-03
Epoch 11, time=1.1818 obj=5.096e-01, lr=1.000e-03
Epoch 12, time=1.1588 obj=4.954e-01, lr=1.000e-03
Epoch 13, time=1.1708 obj=4.873e-01, lr=1.000e-03
Epoch 14, time=1.1578 obj=4.719e-01, lr=1.000e-03
Epoch 15, time=1.1418 obj=4.654e-01, lr=1.000e-03
Epoch 16, time=1.1508 obj=4.605e-01, lr=1.000e-03
Epoch 17, time=1.1968 obj=4.575e-01, lr=1.000e-03
Epoch 18, time=1.1738 obj=4.450e-01, lr=1.000e-03
Epoch 19, time=1.1588 obj=4.422e-01, lr=1.000e-03

Running trial r=120, init_lr=1.00e-02, bs=10
Epoch 0, time=1.1428 obj=1.051e+00, lr=1.000e-02
Epoch 1, time=1.1688 obj=4.923e-01, lr=1.000e-02
Epoch 2, time=1.1548 obj=4.297e-01, lr=1.000e-02
Epoch 3, time=1.1438 obj=3.856e-01, lr=1.000e-02
Epoch 4, time=1.1498 obj=3.499e-01, lr=1.000e-02
Epoch 5, time=1.1318 obj=3.125e-01, lr=1.000e-02
Epoch 6, time=1.1288 obj=2.689e-01, lr=1.000e-02
Epoch 7, time=1.1378 obj=2.127e-01, lr=1.000e-02
Epoch 8, time=1.1378 obj=1.967e-01, lr=1.000e-02
Epoch 9, time=1.1238 obj=1.646e-01, lr=1.000e-02
Epoch 10, time=1.1278 obj=1.381e-01, lr=1.000e-02
Epoch 11, time=1.1428 obj=1.144e-01, lr=1.000e-02
Epoch 12, time=1.1318 obj=9.766e-02, lr=1.000e-02
Epoch 13, time=1.1758 obj=8.271e-02, lr=1.000e-02
Epoch 14, time=1.1978 obj=7.031e-02, lr=1.000e-02
Epoch 15, time=1.1608 obj=5.935e-02, lr=1.000e-02
Epoch 16, time=1.1488 obj=5.011e-02, lr=1.000e-02
Epoch 17, time=1.1728 obj=4.274e-02, lr=1.000e-02
Epoch 18, time=1.1838 obj=3.591e-02, lr=1.000e-02
Epoch 19, time=1.1738 obj=3.101e-02, lr=1.000e-02

Running trial r=120, init_lr=1.00e-01, bs=10
Epoch 0, time=1.1668 obj=5.633e-01, lr=1.000e-01
Epoch 1, time=1.1668 obj=2.968e-01, lr=1.000e-01
Epoch 2, time=1.1458 obj=1.606e-01, lr=1.000e-01
Epoch 3, time=1.1358 obj=7.652e-02, lr=1.000e-01
Epoch 4, time=1.1408 obj=3.775e-02, lr=1.000e-01
Epoch 5, time=1.1388 obj=1.719e-02, lr=1.000e-01
Epoch 6, time=1.1338 obj=7.702e-03, lr=1.000e-01
Epoch 7, time=1.1428 obj=3.484e-03, lr=1.000e-01
Epoch 8, time=1.1368 obj=1.817e-03, lr=1.000e-01
Epoch 9, time=1.1568 obj=9.954e-04, lr=1.000e-01
Epoch 10, time=1.1438 obj=4.193e-04, lr=1.000e-01
Epoch 11, time=1.1448 obj=4.193e-04, lr=1.000e-01
Epoch 12, time=1.1468 obj=3.124e-04, lr=1.000e-01
Epoch 13, time=1.2178 obj=2.419e-04, lr=1.000e-01
Epoch 14, time=1.1438 obj=1.554e-04, lr=1.000e-01
Epoch 15, time=1.1288 obj=1.280e-04, lr=1.000e-01
Epoch 16, time=1.1348 obj=9.622e-05, lr=1.000e-01
Epoch 17, time=1.1288 obj=8.687e-05, lr=1.000e-01
Epoch 18, time=1.1278 obj=6.897e-05, lr=1.000e-01
Epoch 19, time=1.1358 obj=5.427e-05, lr=1.000e-01
```

```
In [13]: plt.clf()
r, bs = 120, 10
for lr, color in zip([1e-4, 1e-3, 0.01, 0.1], ['k', 'b', 'r', 'g']):
    plt.plot(obj_dict_lr_test[(r, lr, bs)], color+'-', lw=2.0, label='lr={:.0e}'.format(lr))
plt.xlabel('Epoch')
plt.ylabel('MSE')
plt.title('Learning rate test')
plt.legend(loc='upper right')
ax = plt.gca()
ax.set_yscale('log')
ax.grid()
plt.show()
```



The best initial learning rate clearly is $\eta = 0.1$.

(d) Over-parameterization test

Now we test the effect of over-parameterization with this fixed learning rate.

```
In [14]: obj_dict_r_test = run_experiment([160, 120, 240, 480], [0.1], [10], epochs=200)

Epoch 41, time=1.115s obj=5.096e-06, lr=1.000e-01
Epoch 42, time=1.109s obj=5.262e-06, lr=1.000e-01
Epoch 43, time=1.115s obj=4.847e-06, lr=1.000e-01
Epoch 44, time=1.118s obj=4.277e-06, lr=1.000e-01
Epoch 45, time=1.115s obj=4.133e-06, lr=1.000e-01
Epoch 46, time=1.118s obj=3.622e-06, lr=1.000e-01
Epoch 47, time=1.115s obj=3.515e-06, lr=1.000e-01
Epoch 48, time=1.115s obj=3.709e-06, lr=1.000e-01
Epoch 49, time=1.108s obj=3.218e-06, lr=1.000e-01
Epoch 50, time=1.146s obj=1.028e-06, lr=5.000e-02
Epoch 51, time=1.121s obj=8.096e-07, lr=5.000e-02

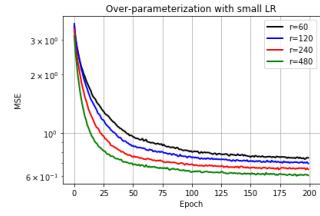
Running trial r=240, init_lr=1.00e-01, bs=10

Epoch 0, time=1.167s obj=5.647e-01, lr=1.000e-01
Epoch 1, time=1.154s obj=2.776e-01, lr=1.000e-01
Epoch 2, time=1.151s obj=1.203e-01, lr=1.000e-01
Epoch 3, time=1.146s obj=5.818e-02, lr=1.000e-01
Epoch 4, time=1.160s obj=2.698e-02, lr=1.000e-01
Epoch 5, time=1.158s obj=1.362e-02, lr=1.000e-01
Epoch 6, time=1.155s obj=6.810e-03, lr=1.000e-01
Epoch 7, time=1.152s obj=3.405e-03, lr=1.000e-01
Epoch 8, time=1.150s obj=1.702e-03, lr=1.000e-01
Epoch 9, time=1.148s obj=8.510e-04, lr=1.000e-01
Epoch 10, time=1.146s obj=4.255e-04, lr=1.000e-01
Epoch 11, time=1.144s obj=2.127e-04, lr=1.000e-01
Epoch 12, time=1.142s obj=1.063e-04, lr=1.000e-01
Epoch 13, time=1.140s obj=5.317e-05, lr=1.000e-01
Epoch 14, time=1.138s obj=2.659e-05, lr=1.000e-01
Epoch 15, time=1.136s obj=1.335e-05, lr=1.000e-01
Epoch 16, time=1.134s obj=6.675e-06, lr=1.000e-01
Epoch 17, time=1.132s obj=3.337e-06, lr=1.000e-01
Epoch 18, time=1.130s obj=1.670e-06, lr=1.000e-01
Epoch 19, time=1.128s obj=8.350e-07, lr=1.000e-01
Epoch 20, time=1.126s obj=4.175e-07, lr=1.000e-01
Epoch 21, time=1.124s obj=2.087e-07, lr=1.000e-01
Epoch 22, time=1.122s obj=1.043e-07, lr=1.000e-01
Epoch 23, time=1.120s obj=5.217e-08, lr=1.000e-01
Epoch 24, time=1.118s obj=2.608e-08, lr=1.000e-01
Epoch 25, time=1.116s obj=1.304e-08, lr=1.000e-01
Epoch 26, time=1.114s obj=6.520e-09, lr=1.000e-01
Epoch 27, time=1.112s obj=3.260e-09, lr=1.000e-01
Epoch 28, time=1.110s obj=1.630e-09, lr=1.000e-01
Epoch 29, time=1.108s obj=8.150e-10, lr=1.000e-01
Epoch 30, time=1.106s obj=4.075e-10, lr=1.000e-01
Epoch 31, time=1.104s obj=2.035e-10, lr=1.000e-01
Epoch 32, time=1.102s obj=1.017e-10, lr=1.000e-01
Epoch 33, time=1.100s obj=5.085e-11, lr=1.000e-01
Epoch 34, time=1.098s obj=2.542e-11, lr=1.000e-01
Epoch 35, time=1.096s obj=1.271e-11, lr=1.000e-01
Epoch 36, time=1.094s obj=6.355e-12, lr=1.000e-01
Epoch 37, time=1.092s obj=3.177e-12, lr=1.000e-01
Epoch 38, time=1.090s obj=1.589e-12, lr=1.000e-01
Epoch 39, time=1.088s obj=7.945e-13, lr=1.000e-01
Epoch 40, time=1.086s obj=3.972e-13, lr=1.000e-01
Epoch 41, time=1.084s obj=1.986e-13, lr=1.000e-01
Epoch 42, time=1.082s obj=9.930e-14, lr=1.000e-01
Epoch 43, time=1.080s obj=4.965e-14, lr=1.000e-01
Epoch 44, time=1.078s obj=2.482e-14, lr=1.000e-01
Epoch 45, time=1.076s obj=1.241e-14, lr=1.000e-01
Epoch 46, time=1.074s obj=6.205e-15, lr=1.000e-01
Epoch 47, time=1.072s obj=3.102e-15, lr=1.000e-01
Epoch 48, time=1.070s obj=1.551e-15, lr=1.000e-01
Epoch 49, time=1.068s obj=7.755e-16, lr=1.000e-01
Epoch 50, time=1.066s obj=3.877e-16, lr=1.000e-01
Epoch 51, time=1.064s obj=1.939e-16, lr=1.000e-01
Epoch 52, time=1.062s obj=9.695e-17, lr=1.000e-01
Epoch 53, time=1.060s obj=4.847e-17, lr=1.000e-01
Epoch 54, time=1.058s obj=2.423e-17, lr=1.000e-01
Epoch 55, time=1.056s obj=1.211e-17, lr=1.000e-01
Epoch 56, time=1.054s obj=6.056e-18, lr=1.000e-01
Epoch 57, time=1.052s obj=3.028e-18, lr=1.000e-01
Epoch 58, time=1.050s obj=1.514e-18, lr=1.000e-01
Epoch 59, time=1.048s obj=7.570e-19, lr=1.000e-01
Epoch 60, time=1.046s obj=3.785e-19, lr=1.000e-01
Epoch 61, time=1.044s obj=1.892e-19, lr=1.000e-01
Epoch 62, time=1.042s obj=9.461e-20, lr=1.000e-01
Epoch 63, time=1.040s obj=4.730e-20, lr=1.000e-01
Epoch 64, time=1.038s obj=2.365e-20, lr=1.000e-01
Epoch 65, time=1.036s obj=1.182e-20, lr=1.000e-01
Epoch 66, time=1.034s obj=5.910e-21, lr=1.000e-01
Epoch 67, time=1.032s obj=2.955e-21, lr=1.000e-01
Epoch 68, time=1.030s obj=1.477e-21, lr=1.000e-01
Epoch 69, time=1.028s obj=7.385e-22, lr=1.000e-01
Epoch 70, time=1.026s obj=3.692e-22, lr=1.000e-01
Epoch 71, time=1.024s obj=1.846e-22, lr=1.000e-01
Epoch 72, time=1.022s obj=9.230e-23, lr=1.000e-01
Epoch 73, time=1.020s obj=4.615e-23, lr=1.000e-01
Epoch 74, time=1.018s obj=2.297e-23, lr=1.000e-01
Epoch 75, time=1.016s obj=1.148e-23, lr=1.000e-01
Epoch 76, time=1.014s obj=5.740e-24, lr=1.000e-01
Epoch 77, time=1.012s obj=2.870e-24, lr=1.000e-01
Epoch 78, time=1.010s obj=1.435e-24, lr=1.000e-01
Epoch 79, time=1.008s obj=7.175e-25, lr=1.000e-01
Epoch 80, time=1.006s obj=3.587e-25, lr=1.000e-01
Epoch 81, time=1.004s obj=1.793e-25, lr=1.000e-01
Epoch 82, time=1.002s obj=8.963e-26, lr=1.000e-01
Epoch 83, time=1.000s obj=4.481e-26, lr=1.000e-01
Epoch 84, time=9.98s obj=2.240e-26, lr=1.000e-01
Epoch 85, time=9.96s obj=1.120e-26, lr=1.000e-01
Epoch 86, time=9.94s obj=5.600e-27, lr=1.000e-01
Epoch 87, time=9.92s obj=2.800e-27, lr=1.000e-01
Epoch 88, time=9.90s obj=1.400e-27, lr=1.000e-01
Epoch 89, time=9.88s obj=6.000e-28, lr=1.000e-01
Epoch 90, time=9.86s obj=2.900e-28, lr=1.000e-01
Epoch 91, time=9.84s obj=1.400e-28, lr=1.000e-01
Epoch 92, time=9.82s obj=6.000e-29, lr=1.000e-01
Epoch 93, time=9.80s obj=2.900e-29, lr=1.000e-01
Epoch 94, time=9.78s obj=1.400e-29, lr=1.000e-01
Epoch 95, time=9.76s obj=6.000e-30, lr=1.000e-01
Epoch 96, time=9.74s obj=2.900e-30, lr=1.000e-01
Epoch 97, time=9.72s obj=1.400e-30, lr=1.000e-01
Epoch 98, time=9.70s obj=6.000e-31, lr=1.000e-01
Epoch 99, time=9.68s obj=2.900e-31, lr=1.000e-01
Epoch 100, time=9.66s obj=1.400e-31, lr=1.000e-01
Epoch 101, time=9.64s obj=6.000e-32, lr=1.000e-01
Epoch 102, time=9.62s obj=2.900e-32, lr=1.000e-01
Epoch 103, time=9.60s obj=1.400e-32, lr=1.000e-01
Epoch 104, time=9.58s obj=6.000e-33, lr=1.000e-01
Epoch 105, time=9.56s obj=2.900e-33, lr=1.000e-01
Epoch 106, time=9.54s obj=1.400e-33, lr=1.000e-01
Epoch 107, time=9.52s obj=6.000e-34, lr=1.000e-01
Epoch 108, time=9.50s obj=2.900e-34, lr=1.000e-01
Epoch 109, time=9.48s obj=1.400e-34, lr=1.000e-01
Epoch 110, time=9.46s obj=6.000e-35, lr=1.000e-01
Epoch 111, time=9.44s obj=2.900e-35, lr=1.000e-01
Epoch 112, time=9.42s obj=1.400e-35, lr=1.000e-01
Epoch 113, time=9.40s obj=6.000e-36, lr=1.000e-01
Epoch 114, time=9.38s obj=2.900e-36, lr=1.000e-01
Epoch 115, time=9.36s obj=1.400e-36, lr=1.000e-01
Epoch 116, time=9.34s obj=6.000e-37, lr=1.000e-01
Epoch 117, time=9.32s obj=2.900e-37, lr=1.000e-01
Epoch 118, time=9.30s obj=1.400e-37, lr=1.000e-01
Epoch 119, time=9.28s obj=6.000e-38, lr=1.000e-01
Epoch 120, time=9.26s obj=2.900e-38, lr=1.000e-01
Epoch 121, time=9.24s obj=1.400e-38, lr=1.000e-01
Epoch 122, time=9.22s obj=6.000e-39, lr=1.000e-01
Epoch 123, time=9.20s obj=2.900e-39, lr=1.000e-01
Epoch 124, time=9.18s obj=1.400e-39, lr=1.000e-01
Epoch 125, time=9.16s obj=6.000e-40, lr=1.000e-01
Epoch 126, time=9.14s obj=2.900e-40, lr=1.000e-01
Epoch 127, time=9.12s obj=1.400e-40, lr=1.000e-01
Epoch 128, time=9.10s obj=6.000e-41, lr=1.000e-01
Epoch 129, time=9.08s obj=2.900e-41, lr=1.000e-01
Epoch 130, time=9.06s obj=1.400e-41, lr=1.000e-01
Epoch 131, time=9.04s obj=6.000e-42, lr=1.000e-01
Epoch 132, time=9.02s obj=2.900e-42, lr=1.000e-01
Epoch 133, time=9.00s obj=1.400e-42, lr=1.000e-01
Epoch 134, time=8.98s obj=6.000e-43, lr=1.000e-01
Epoch 135, time=8.96s obj=2.900e-43, lr=1.000e-01
Epoch 136, time=8.94s obj=1.400e-43, lr=1.000e-01
Epoch 137, time=8.92s obj=6.000e-44, lr=1.000e-01
Epoch 138, time=8.90s obj=2.900e-44, lr=1.000e-01
Epoch 139, time=8.88s obj=1.400e-44, lr=1.000e-01
Epoch 140, time=8.86s obj=6.000e-45, lr=1.000e-01
Epoch 141, time=8.84s obj=2.900e-45, lr=1.000e-01
Epoch 142, time=8.82s obj=1.400e-45, lr=1.000e-01
Epoch 143, time=8.80s obj=6.000e-46, lr=1.000e-01
Epoch 144, time=8.78s obj=2.900e-46, lr=1.000e-01
Epoch 145, time=8.76s obj=1.400e-46, lr=1.000e-01
Epoch 146, time=8.74s obj=6.000e-47, lr=1.000e-01
Epoch 147, time=8.72s obj=2.900e-47, lr=1.000e-01
Epoch 148, time=8.70s obj=1.400e-47, lr=1.000e-01
Epoch 149, time=8.68s obj=6.000e-48, lr=1.000e-01
Epoch 150, time=8.66s obj=2.900e-48, lr=1.000e-01
Epoch 151, time=8.64s obj=1.400e-48, lr=1.000e-01
Epoch 152, time=8.62s obj=6.000e-49, lr=1.000e-01
Epoch 153, time=8.60s obj=2.900e-49, lr=1.000e-01
Epoch 154, time=8.58s obj=1.400e-49, lr=1.000e-01
Epoch 155, time=8.56s obj=6.000e-50, lr=1.000e-01
Epoch 156, time=8.54s obj=2.900e-50, lr=1.000e-01
Epoch 157, time=8.52s obj=1.400e-50, lr=1.000e-01
Epoch 158, time=8.50s obj=6.000e-51, lr=1.000e-01
Epoch 159, time=8.48s obj=2.900e-51, lr=1.000e-01
Epoch 160, time=8.46s obj=1.400e-51, lr=1.000e-01
Epoch 161, time=8.44s obj=6.000e-52, lr=1.000e-01
Epoch 162, time=8.42s obj=2.900e-52, lr=1.000e-01
Epoch 163, time=8.40s obj=1.400e-52, lr=1.000e-01
Epoch 164, time=8.38s obj=6.000e-53, lr=1.000e-01
Epoch 165, time=8.36s obj=2.900e-53, lr=1.000e-01
Epoch 166, time=8.34s obj=1.400e-53, lr=1.000e-01
Epoch 167, time=8.32s obj=6.000e-54, lr=1.000e-01
Epoch 168, time=8.30s obj=2.900e-54, lr=1.000e-01
Epoch 169, time=8.28s obj=1.400e-54, lr=1.000e-01
Epoch 170, time=8.26s obj=6.000e-55, lr=1.000e-01
Epoch 171, time=8.24s obj=2.900e-55, lr=1.000e-01
Epoch 172, time=8.22s obj=1.400e-55, lr=1.000e-01
Epoch 173, time=8.20s obj=6.000e-56, lr=1.000e-01
Epoch 174, time=8.18s obj=2.900e-56, lr=1.000e-01
Epoch 175, time=8.16s obj=1.400e-56, lr=1.000e-01
Epoch 176, time=8.14s obj=6.000e-57, lr=1.000e-01
Epoch 177, time=8.12s obj=2.900e-57, lr=1.000e-01
Epoch 178, time=8.10s obj=1.400e-57, lr=1.000e-01
Epoch 179, time=8.08s obj=6.000e-58, lr=1.000e-01
Epoch 180, time=8.06s obj=2.900e-58, lr=1.000e-01
Epoch 181, time=8.04s obj=1.400e-58, lr=1.000e-01
Epoch 182, time=8.02s obj=6.000e-59, lr=1.000e-01
Epoch 183, time=8.00s obj=2.900e-59, lr=1.000e-01
Epoch 184, time=7.98s obj=1.400e-59, lr=1.000e-01
Epoch 185, time=7.96s obj=6.000e-60, lr=1.000e-01
Epoch 186, time=7.94s obj=2.900e-60, lr=1.000e-01
Epoch 187, time=7.92s obj=1.400e-60, lr=1.000e-01
Epoch 188, time=7.90s obj=6.000e-61, lr=1.000e-01
Epoch 189, time=7.88s obj=2.900e-61, lr=1.000e-01
Epoch 190, time=7.86s obj=1.400e-61, lr=1.000e-01
Epoch 191, time=7.84s obj=6.000e-62, lr=1.000e-01
Epoch 192, time=7.82s obj=2.900e-62, lr=1.000e-01
Epoch 193, time=7.80s obj=1.400e-62, lr=1.000e-01
Epoch 194, time=7.78s obj=6.000e-63, lr=1.000e-01
Epoch 195, time=7.76s obj=2.900e-63, lr=1.000e-01
Epoch 196, time=7.74s obj=1.400e-63, lr=1.000e-01
Epoch 197, time=7.72s obj=6.000e-64, lr=1.000e-01
Epoch 198, time=7.70s obj=2.900e-64, lr=1.000e-01
Epoch 199, time=7.68s obj=1.400e-64, lr=1.000e-01
Epoch 200, time=7.66s obj=6.000e-65, lr=1.000e-01
Epoch 201, time=7.64s obj=2.900e-65, lr=1.000e-01
Epoch 202, time=7.62s obj=1.400e-65, lr=1.000e-01
Epoch 203, time=7.60s obj=6.000e-66, lr=1.000e-01
Epoch 204, time=7.58s obj=2.900e-66, lr=1.000e-01
Epoch 205, time=7.56s obj=1.400e-66, lr=1.000e-01
Epoch 206, time=7.54s obj=6.000e-67, lr=1.000e-01
Epoch 207, time=7.52s obj=2.900e-67, lr=1.000e-01
Epoch 208, time=7.50s obj=1.400e-67, lr=1.000e-01
Epoch 209, time=7.48s obj=6.000e-68, lr=1.000e-01
Epoch 210, time=7.46s obj=2.900e-68, lr=1.000e-01
Epoch 211, time=7.44s obj=1.400e-68, lr=1.000e-01
Epoch 212, time=7.42s obj=6.000e-69, lr=1.000e-01
Epoch 213, time=7.40s obj=2.900e-69, lr=1.000e-01
Epoch 214, time=7.38s obj=1.400e-69, lr=1.000e-01
Epoch 215, time=7.36s obj=6.000e-70, lr=1.000e-01
Epoch 216, time=7.34s obj=2.900e-70, lr=1.000e-01
Epoch 217, time=7.32s obj=1.400e-70, lr=1.000e-01
Epoch 218, time=7.30s obj=6.000e-71, lr=1.000e-01
Epoch 219, time=7.28s obj=2.900e-71, lr=1.000e-01
Epoch 220, time=7.26s obj=1.400e-71, lr=1.000e-01
Epoch 221, time=7.24s obj=6.000e-72, lr=1.000e-01
Epoch 222, time=7.22s obj=2.900e-72, lr=1.000e-01
Epoch 223, time=7.20s obj=1.400e-72, lr=1.000e-01
Epoch 224, time=7.18s obj=6.000e-73, lr=1.000e-01
Epoch 225, time=7.16s obj=2.900e-73, lr=1.000e-01
Epoch 226, time=7.14s obj=1.400e-73, lr=1.000e-01
Epoch 227, time=7.12s obj=6.000e-74, lr=1.000e-01
Epoch 228, time=7.10s obj=2.900e-74, lr=1.000e-01
Epoch 229, time=7.08s obj=1.400e-74, lr=1.000e-01
Epoch 230, time=7.06s obj=6.000e-75, lr=1.000e-01
Epoch 231, time=7.04s obj=2.900e-75, lr=1.000e-01
Epoch 232, time=7.02s obj=1.400e-75, lr=1.000e-01
Epoch 233, time=7.00s obj=6.000e-76, lr=1.000e-01
Epoch 234, time=6.98s obj=2.900e-76, lr=1.000e-01
Epoch 235, time=6.96s obj=1.400e-76, lr=1.000e-01
Epoch 236, time=6.94s obj=6.000e-77, lr=1.000e-01
Epoch 237, time=6.92s obj=2.900e-77, lr=1.000e-01
Epoch 238, time=6.90s obj=1.400e-77, lr=1.000e-01
Epoch 239, time=6.88s obj=6.000e-78, lr=1.000e-01
Epoch 240, time=6.86s obj=2.900e-78, lr=1.000e-01
Epoch 241, time=6.84s obj=1.400e-78, lr=1.000e-01
Epoch 242, time=6.82s obj=6.000e-79, lr=1.000e-01
Epoch 243, time=6.80s obj=2.900e-79, lr=1.000e-01
Epoch 244, time=6.78s obj=1.400e-79, lr=1.000e-01
Epoch 245, time=6.76s obj=6.000e-80, lr=1.000e-01
Epoch 246, time=6.74s obj=2.900e-80, lr=1.000e-01
Epoch 247, time=6.72s obj=1.400e-80, lr=1.000e-01
Epoch 248, time=6.70s obj=6.000e-81, lr=1.000e-01
Epoch 249, time=6.68s obj=2.900e-81, lr=1.000e-01
Epoch 250, time=6.66s obj=1.400e-81, lr=1.000e-01
Epoch 251, time=6.64s obj=6.000e-82, lr=1.000e-01
Epoch 252, time=6.62s obj=2.900e-82, lr=1.000e-01
Epoch 253, time=6.60s obj=1.400e-82, lr=1.000e-01
Epoch 254, time=6.58s obj=6.000e-83, lr=1.000e-01
Epoch 255, time=6.56s obj=2.900e-83, lr=1.000e-01
Epoch 256, time=6.54s obj=1.400e-83, lr=1.000e-01
Epoch 257, time=6.52s obj=6.000e-84, lr=1.000e-01
Epoch 258, time=6.50s obj=2.900e-84, lr=1.000e-01
Epoch 259, time=6.48s obj=1.400e-84, lr=1.000e-01
Epoch 260, time=6.46s obj=6.000e-85, lr=1.000e-01
Epoch 261, time=6.44s obj=2.900e-85, lr=1.000e-01
Epoch 262, time=6.42s obj=1.400e-85, lr=1.000e-01
Epoch 263, time=6.40s obj=6.000e-86, lr=1.000e-01
Epoch 264, time=6.38s obj=2.900e-86, lr=1.000e-01
Epoch 265, time=6.36s obj=1.400e-86, lr=1.000e-01
Epoch 266, time=6.34s obj=6.000e-87, lr=1.000e-01
Epoch 267, time=6.32s obj=2.900e-87, lr=1.000e-01
Epoch 268, time=6.30s obj=1.400e-87, lr=1.000e-01
Epoch 269, time=6.28s obj=6.000e-88, lr=1.000e-01
Epoch 270, time=6.26s obj=2.900e-88, lr=1.000e-01
Epoch 271, time=6.24s obj=1.400e-88, lr=1.000e-01
Epoch 272, time=6.22s obj=6.000e-89, lr=1.000e-01
Epoch 273, time=6.20s obj=2.900e-89, lr=1.000e-01
Epoch 274, time=6.18s obj=1.400e-89, lr=1.000e-01
Epoch 275, time=6.16s obj=6.000e-90, lr=1.000e-01
Epoch 276, time=6.14s obj=2.900e-90, lr=1.000e-01
Epoch 277, time=6.12s obj=1.400e-90, lr=1.000e-01
Epoch 278, time=6.10s obj=6.000e-91, lr=1.000e-01
Epoch 279, time=6.08s obj=2.900e-91,
```

```
In [16]: obj_dict_r_test_small_eta = run_experiment([60, 120, 240, 480], [0.001], [100], epochs=200, init_scale=1.0)
Epoch 184, time=0.688s obj=7.478e-01, lr=1.250e-04
Epoch 185, time=0.698s obj=7.468e-01, lr=1.250e-04
Epoch 186, time=0.770s obj=7.453e-01, lr=1.250e-04
Epoch 187, time=0.687s obj=7.452e-01, lr=1.250e-04
Epoch 188, time=0.697s obj=7.449e-01, lr=1.250e-04
Epoch 189, time=0.707s obj=7.449e-01, lr=1.250e-04
Epoch 190, time=0.731s obj=7.517e-01, lr=1.250e-04
Epoch 191, time=0.707s obj=7.429e-01, lr=1.250e-04
Epoch 192, time=0.696s obj=7.498e-01, lr=1.250e-04
Epoch 193, time=0.700s obj=7.420e-01, lr=1.250e-04
Epoch 194, time=0.693s obj=7.418e-01, lr=1.250e-04
Epoch 195, time=0.692s obj=7.449e-01, lr=1.250e-04
Epoch 196, time=0.689s obj=7.346e-01, lr=1.250e-04
Epoch 197, time=0.687s obj=7.409e-01, lr=1.250e-04
Epoch 198, time=0.690s obj=7.485e-01, lr=1.250e-04
Epoch 199, time=0.692s obj=7.466e-01, lr=1.250e-04
```

Running trial r=120, init_lr=1.00e-03, bs=100

```
In [17]: plt.clf()
lr, bs = 0.001, 100
for r, color in zip([60, 120, 240, 480], ['k', 'b', 'r', 'g']):
    plt.plot(obj_dict_r_test_small_eta[(r, lr, bs)], color='-' ,lw=2.0, label='r={:d}'.format(r))
plt.xlabel('Epoch')
plt.ylabel('MSE')
plt.title('Over-parameterization with small LR')
plt.legend(loc='upper right')
ax = plt.gca()
ax.set_yscale('log')
ax.grid()
plt.show()
```



We observe a pattern that more closely resembles Livni et al.