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

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

where $\{(\boldsymbol{x}_i, y_i)\}_{i=1}^N$ 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

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

Prove that any global minimizer to (2) must have balanced weight matrices. I.e. $\|\boldsymbol{U}\|_F = \|\boldsymbol{V}\|_F$. 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

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

Prove that by contrast, any global minimizer to (3) will typically *not* have balanced weight norms. I.e. will not satisfy $\|\boldsymbol{U}\|_F = \|\boldsymbol{V}\|_F$. Specifically, let $(\boldsymbol{U}^*, \boldsymbol{V}^*)$, be a global minimizer and let $\|\boldsymbol{U}^*\|_F \|\boldsymbol{V}^*\|_F = c$. Assume $c > 0$. Show that $\boldsymbol{U}^*$, $\boldsymbol{V}^*$ 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 $(\boldsymbol{U}^*, \boldsymbol{V}^*)$ be a global minimizer of (1) and assume $\boldsymbol{U}^* \neq \boldsymbol{0}$. Let $\alpha > 0$ and note by positive homogeneity,

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

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

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

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

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

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

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

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 regularization[1] is included. In particular, the regularizer in (c) also guarantees bounded minimizers.

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

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

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

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

After some manipulation, it follows that $\|\boldsymbol{U}^*\|_F = \|\boldsymbol{V}^*\|_F$ 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.)

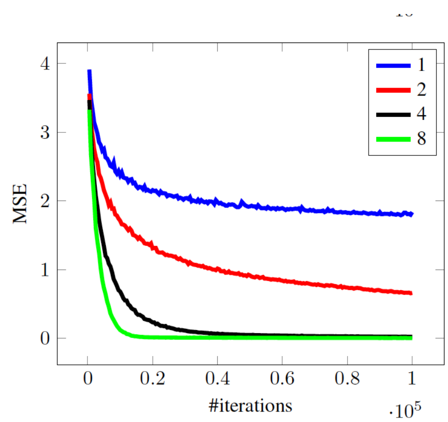


Figure 1: Effect of over-parameterization on single hidden layer network training from [1]. The data are synthetically generated from a network with fixed "planted" weights, and hidden layer size r_0. Mean squared error (MSE) is shown as a function of iteration for varying "over-parameterization levels". (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 $(\boldsymbol{x}, \boldsymbol{y})$ are drawn from a single hidden layer ReLU network with planted weights. Specifically, generate random Gaussian weight matrices $\boldsymbol{U}_0 \in \mathbb{R}^{n\times r_0}$, $\boldsymbol{V}_0 \in \mathbb{R}^{D\times r_0}$ with $(\boldsymbol{U}_0)_{ij} \sim \mathcal{N}(0, 1/r_0)$, $(\boldsymbol{V}_0)_{ij} \sim \mathcal{N}(\boldsymbol{0}, 1/D)$. Generate samples using these fixed "planted" weights as follows

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

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

[1] `https://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 $\|\boldsymbol{y}_i\|_2^2$ 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)

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

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, $\eta \in \{10^{-4}, 10^{-3}, \ldots, 0.1\}$. 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 $\|\mathbf{y}_i\|_2^2$ 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.

    From: https://github.com/pytorch/examples/blob/master/imagenet/main.py
    """
    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
```

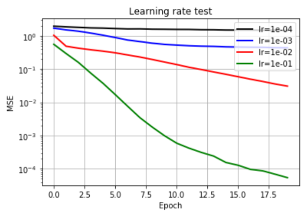
(c) Learning rate test

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=1.158s obj=1.975e+00, lr=1.000e-04
Epoch 1, time=1.123s obj=1.890e+00, lr=1.000e-04
Epoch 2, time=1.130s obj=1.820e+00, lr=1.000e-04
Epoch 3, time=1.126s obj=1.756e+00, lr=1.000e-04
Epoch 4, time=1.116s obj=1.729e+00, lr=1.000e-04
Epoch 5, time=1.115s obj=1.678e+00, lr=1.000e-04
Epoch 6, time=1.120s obj=1.633e+00, lr=1.000e-04
Epoch 7, time=1.120s obj=1.639e+00, lr=1.000e-04
Epoch 8, time=1.118s obj=1.603e+00, lr=1.000e-04
Epoch 9, time=1.125s obj=1.595e+00, lr=1.000e-04
Epoch 10, time=1.122s obj=1.580e+00, lr=1.000e-04
Epoch 11, time=1.137s obj=1.577e+00, lr=1.000e-04
Epoch 12, time=1.186s obj=1.550e+00, lr=1.000e-04
Epoch 13, time=1.161s obj=1.544e+00, lr=1.000e-04
Epoch 14, time=1.121s obj=1.518e+00, lr=1.000e-04
Epoch 15, time=1.127s obj=1.512e+00, lr=1.000e-04
Epoch 16, time=1.131s obj=1.510e+00, lr=1.000e-04
Epoch 17, time=1.129s obj=1.494e+00, lr=1.000e-04
Epoch 18, time=1.128s obj=1.490e+00, lr=1.000e-04
Epoch 19, time=1.126s obj=1.460e+00, lr=1.000e-04

Running trial r=120, init_lr=1.00e-03, bs=10

Epoch 0, time=1.125s obj=1.723e+00, lr=1.000e-03
Epoch 1, time=1.125s obj=1.531e+00, lr=1.000e-03
Epoch 2, time=1.143s obj=1.395e+00, lr=1.000e-03
Epoch 3, time=1.146s obj=1.227e+00, lr=1.000e-03
Epoch 4, time=1.130s obj=1.058e+00, lr=1.000e-03
Epoch 5, time=1.130s obj=8.998e-01, lr=1.000e-03
Epoch 6, time=1.147s obj=7.584e-01, lr=1.000e-03
Epoch 7, time=1.143s obj=6.797e-01, lr=1.000e-03
Epoch 8, time=1.143s obj=6.086e-01, lr=1.000e-03
Epoch 9, time=1.172s obj=5.604e-01, lr=1.000e-03
Epoch 10, time=1.176s obj=5.315e-01, lr=1.000e-03
Epoch 11, time=1.181s obj=5.096e-01, lr=1.000e-03
Epoch 12, time=1.158s obj=4.954e-01, lr=1.000e-03
Epoch 13, time=1.170s obj=4.873e-01, lr=1.000e-03
Epoch 14, time=1.157s obj=4.719e-01, lr=1.000e-03
Epoch 15, time=1.141s obj=4.654e-01, lr=1.000e-03
Epoch 16, time=1.150s obj=4.605e-01, lr=1.000e-03
Epoch 17, time=1.196s obj=4.575e-01, lr=1.000e-03
Epoch 18, time=1.173s obj=4.450e-01, lr=1.000e-03
Epoch 19, time=1.158s obj=4.422e-01, lr=1.000e-03

Running trial r=120, init_lr=1.00e-02, bs=10

Epoch 0, time=1.142s obj=1.051e+00, lr=1.000e-02
Epoch 1, time=1.168s obj=4.923e-01, lr=1.000e-02
Epoch 2, time=1.154s obj=4.297e-01, lr=1.000e-02
Epoch 3, time=1.143s obj=3.856e-01, lr=1.000e-02
Epoch 4, time=1.149s obj=3.499e-01, lr=1.000e-02
Epoch 5, time=1.131s obj=3.125e-01, lr=1.000e-02
Epoch 6, time=1.128s obj=2.689e-01, lr=1.000e-02
Epoch 7, time=1.128s obj=2.332e-01, lr=1.000e-02
Epoch 8, time=1.137s obj=1.967e-01, lr=1.000e-02
Epoch 9, time=1.123s obj=1.646e-01, lr=1.000e-02
Epoch 10, time=1.127s obj=1.381e-01, lr=1.000e-02
Epoch 11, time=1.142s obj=1.144e-01, lr=1.000e-02
Epoch 12, time=1.131s obj=9.766e-02, lr=1.000e-02
Epoch 13, time=1.175s obj=8.271e-02, lr=1.000e-02
Epoch 14, time=1.197s obj=7.031e-02, lr=1.000e-02
Epoch 15, time=1.160s obj=5.935e-02, lr=1.000e-02
Epoch 16, time=1.148s obj=5.011e-02, lr=1.000e-02
Epoch 17, time=1.172s obj=4.274e-02, lr=1.000e-02
Epoch 18, time=1.183s obj=3.591e-02, lr=1.000e-02
Epoch 19, time=1.173s obj=3.101e-02, lr=1.000e-02

Running trial r=120, init_lr=1.00e-01, bs=10

Epoch 0, time=1.166s obj=5.633e-01, lr=1.000e-01
Epoch 1, time=1.166s obj=2.968e-01, lr=1.000e-01
Epoch 2, time=1.145s obj=1.606e-01, lr=1.000e-01
Epoch 3, time=1.135s obj=7.652e-02, lr=1.000e-01
Epoch 4, time=1.140s obj=3.775e-02, lr=1.000e-01
Epoch 5, time=1.138s obj=1.719e-02, lr=1.000e-01
Epoch 6, time=1.133s obj=7.702e-03, lr=1.000e-01
Epoch 7, time=1.142s obj=3.484e-03, lr=1.000e-01
Epoch 8, time=1.136s obj=1.817e-03, lr=1.000e-01
Epoch 9, time=1.156s obj=9.954e-04, lr=1.000e-01
Epoch 10, time=1.153s obj=5.913e-04, lr=1.000e-01
Epoch 11, time=1.142s obj=4.194e-04, lr=1.000e-01
Epoch 12, time=1.146s obj=3.124e-04, lr=1.000e-01
Epoch 13, time=1.217s obj=2.419e-04, lr=1.000e-01
Epoch 14, time=1.143s obj=1.554e-04, lr=1.000e-01
Epoch 15, time=1.128s obj=1.280e-04, lr=1.000e-01
Epoch 16, time=1.134s obj=9.622e-05, lr=1.000e-01
Epoch 17, time=1.128s obj=8.687e-05, lr=1.000e-01
Epoch 18, time=1.127s obj=6.897e-05, lr=1.000e-01
Epoch 19, time=1.135s 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()
```



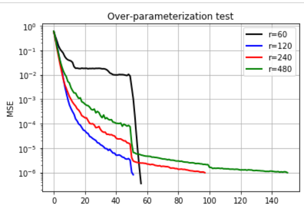
The best initial learning rate clearly is $\eta = 0.1$.

(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([60, 120, 240, 480], [0.1], [10], epochs=200)

         Epoch 41, time=1.115s obj=5.096e-06, lr=1.000e-01
         Epoch 42, time=1.108s obj=5.262e-06, lr=1.000e-01
         Epoch 43, time=1.111s 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=1.111s 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
```

```
In [15]: plt.clf()
         lr, bs = 0.1, 10
         for r, color in zip([60, 120, 240, 480], ['k', 'b', 'r', 'g']):
             plt.plot(obj_dict_r_test[(r, lr, bs)], color+'-', lw=2.0, label='r={:d}'.format(r))
         plt.xlabel('Epoch')
         plt.ylabel('MSE')
         plt.title('Over-parameterization test')
         plt.legend(loc='upper right')
         ax = plt.gca()
         ax.set_yscale('log')
         ax.grid()
         plt.show()
```



We observe a pattern quite different from Livni et al. All 4 networks converge to near zero error ($< 10^{-6}$) within 150 epochs. The fastest to converge is not $r = 480$ but $r = 120$, which importantly is the size we tuned the learning rate on.

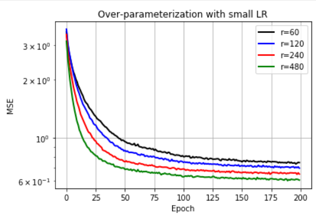
(e) Trying to reproduce pattern from Livni et al.

In (d), we initialized the network weights to be smaller than those of the planted network, and used a large learning rate tuned on $r = 120$. Now we use a much smaller learning rate ($\eta = .001$), a larger batch size ($m = 100$), and an "accurate" initialization scale.

```
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=7.472e-01, lr=1.250e-04
Epoch 188, time=0.697s obj=7.417e-01, lr=1.250e-04
Epoch 189, time=0.709s obj=7.507e-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.