UVA CS 6316: Machine Learning : 2019 Fall
Course Project: Deep2Reproduce @
https://github.com/qiyanjun/deep2reproduce/tree/master/2019Fall

On the Optimization of Deep Networks:

Implicit Acceleration by Overparameterization

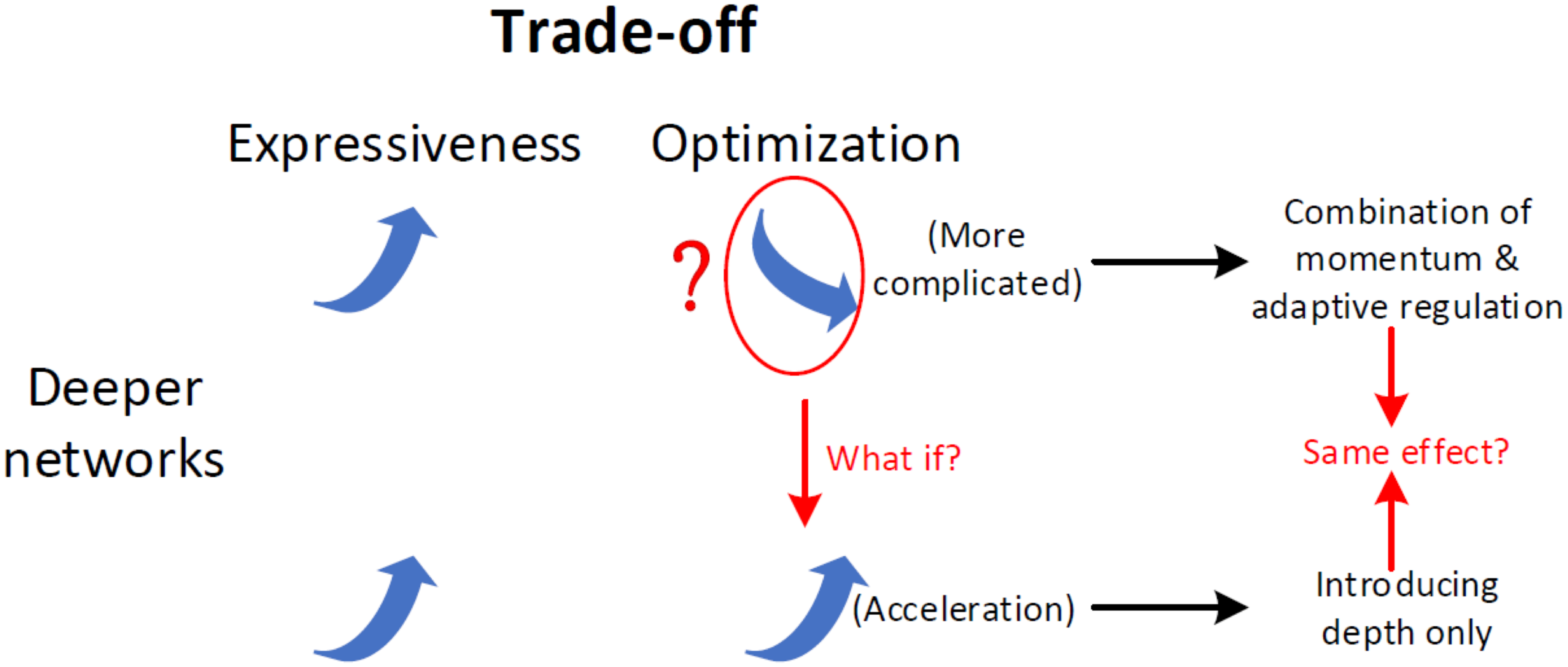
Reproduced By: Fuxiao Liu, Yinzhu Jin, Dexuan Zhang, Tianyang Luo

Outline

Background and Motivation

- Conventional wisdom in deep learning states that increasing depth improves expressiveness but complicates optimization.
- *Momentum, adaptive regularization and AdaGrad.*
- This paper conveys a rather counterintuitive message: **Increasing depth can *accelerate* optimization.**

An Intuitive Figure Showing WHY Claim



Outline

- Background and Motivation
- Claim/Target Task
- Related Work and Proposed Solution
- Implementation Toolbox and Data Summary
- Experiments and Analysis
- Conclusion and Future Work

Claim/Target Task

- If increasing depth leads to faster training on a given dataset, **how can one tell whether the improvement arose from a true acceleration phenomenon, or simply due to better representational power** (the shallower network was unable to attain the same training loss).
- *Linear Neural Network.*
- Adding layers does not alter expressiveness; it manifests itself only in the replacement of a matrix parameter by a product of matrices – an *overparameterization*.

Outline

Related Work

- Theoretical study of optimization in deep learning is a highly active area of research. Works along this line typically analyze critical points (local minima, saddles) in the landscape of the training objective, either for linear networks.
- Other works characterize other aspects of objective landscapes, for example Safran & Shamir (2016) showed that under certain conditions a monotonically descending path from initialization to global optimum exists.
- For linear networks. Like ours, these works analyze gradient descent through its corresponding differential equations. Fukumizu (1998) focuses on linear regression with l2 loss, but does not consider the effect of varying depth.
- Use of preconditioners to speed up optimization is also a well-known technique, including classic Newton’s method.In terms of combining momentum and adaptive precondition- ing, Adam (Kingma & Ba, 2014) is a popular approach,

Proposed Solution - theoretical derivation

- An N-layer linear neural network, can be seen as a function:

$$\Phi(x) = W_N W_{N-1} \ldots W_1 x$$

Denote: $W_e = W_N W_{N-1} \ldots W_1$ which is a k×d vector

- At each time of gradient descent:

$$W_j^{(t+1)} \longleftarrow (1 - \eta\lambda) W_j^{(t)} - \eta \frac{\partial L}{\partial W_j} (W_1^{(t)}, \ldots, W_N^{(t)})$$

η: learning rate; λ: weight decay coefficient

- The gradient of each parameter can be seen as a function of t, given learning rate is very small
- Assume the following stands, which is approximately true if parameters are initialized closed to zero:

$$W_{j+1}^T(t_0)W_{j+1}(t_0) = W_j(t_0)W_j^T(t_0)$$

- Then we can deduct that this equation also stands for $\forall t \geq t_0$.
- This leads to the expression of W_e along t, returning back to the discrete situation, we can get update rule for W_e:

$$W_e^{(t+1)} \longleftarrow (1-\eta\lambda N)W_e^{(t)} - \eta\sum_{j=1}^{N}\left[W_e^{(t)}\left(W_e^{(t)}\right)^T\right]^{(j-1)/N}\frac{dL}{dW}(W_e^{(t)})\left[\left(W_e^{(t)}\right)^T W_e^{(t)}\right]^{(N-j)/N}$$

- To make it more interpretable, we can convert parameter matrix into a vector in column-first order, then the equation is:

$$vec(W_e^{(t+1)}) \longleftarrow (1 - \eta\lambda N) vec(W_e^{(t)}) - \eta P_{W_e^{(t)}} \frac{dL}{dW}(W_e^{(t)})$$

$P_{W_e^{(t)}}$ Actually equals to $\sum_{j=1}^{N} [W_e^\top W_e]^{\frac{N-j}{N}} \odot [W_e W_e^\top]^{\frac{j-1}{N}}$ Kronecker product

whose eigenvectors are: $vec(u_r v_{r'}^T), r = 1 \dots k, r' = 1 \dots d$ left/right singular vector of We

with corresponding eigenvalues:

$$\sum_{j=1}^{N} \sigma_r^{2(N-j)/N} \sigma_{r'}^{2(j-2)/N}$$

singular values of We

$$vec(W_e^{(t+1)}) \longleftarrow (1 - \eta\lambda N)vec(W_e^{(t)}) - \eta P_{W_e^{(t)}} \frac{dL}{dW}(W_e^{(t)}),$$

The transformation applied to the gradient can be seen as a **preconditioning**, which favors directions that correspond to singular vectors whose presence in W_e is stronger.

Since parameters are initialized near zero, the location in parameter space can also be regarded as **the overall movement made by the algorithm**.

Thus, overparameterization promotes movement along directions already taken by the optimization, and therefore can be seen as a form of acceleration

Single-output Case

$$\begin{aligned} W_e^{(t+1)} \mapsfrom\ & (1-\eta\lambda N)\cdot W_e^{(t)} \\ & -\eta\|W_e^{(t)}\|_2^{2-\frac{2}{N}}\cdot\Big(\tfrac{dL^1}{dW}(W_e^{(t)})+ \\ & \qquad (N-1)\cdot Pr_{W_e^{(t)}}\big\{\tfrac{dL^1}{dW}(W_e^{(t)})\big\}\Big) \end{aligned}$$

Two parts:

- Adaptive learning rate
- Amplify gradient on direction of We

Overparameterization effect cannot be attained via regularization

How to prove?

Brief summary:

- $F(W) := \|W\|_2^{2-\frac{2}{N}} \cdot \left(\frac{dL^1}{dW}(W) + (N-1) \cdot Pr_W\left\{\frac{dL^1}{dW}(W)\right\}\right)$ is not gradient field of any function
- Assume there exists such a function
- We can find a closed curve, s.t. Linear Integral of F(W) over this curve does not vanish.
- This contradicts with the gradient theorem, which says for a continuously differentiable function h, and a piecewise smooth curve, it stands:

$$\int_{\Gamma} \frac{dh}{dW} = h(\gamma_e) - h(\gamma_s)$$

Remember that the whole theoretical derivation relies on two approximation:

1. $W_{j+1}^T(t_0)W_{j+1}(t_0) = W_j(t_0)W_j^T(t_0)$ is true for initialization

1. Learning rate is very small so that the update rule can be seen as a continuous function.

To fully justify the effects of overparameterization, experiments are needed.

Outline

- Background and Motivation
- Claim/Target Task
- Related Work and Proposed Solution
- Implementation Toolbox and Data Summary ⇦
- Experiments and Analysis
- Conclusion and Future Work

Implementation Toolbox

TensorFlow

K Keras

PyTorch

scikit learn

NumPy

Data Summary

Gas Sensor Array Drift at Different Concentrations Dataset

- Obtained from UCI Machine Learning Repository
- Only ethanol data used
- Scalar regression task with 2565 examples comprising 128 features
- Perform scaling before the experiments (min-max scaling, Z-score normalization)

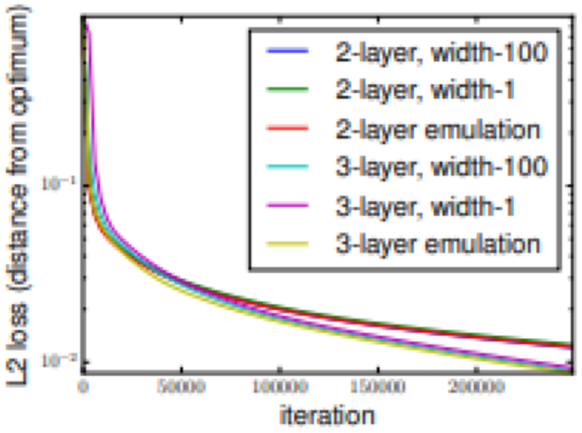
MNIST dataset

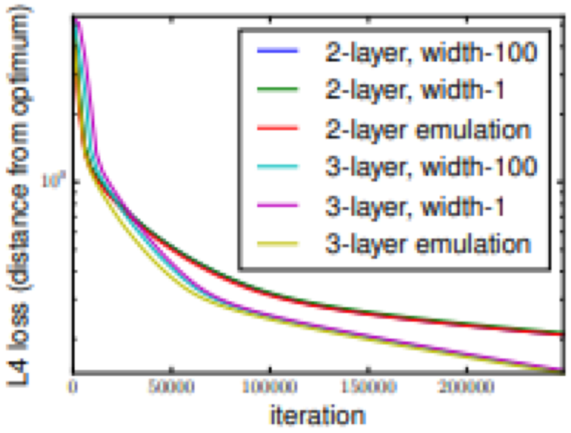
- Embedded in a built-in tutorial in TensorFlow

Outline

Experimental Results

Experiment 1



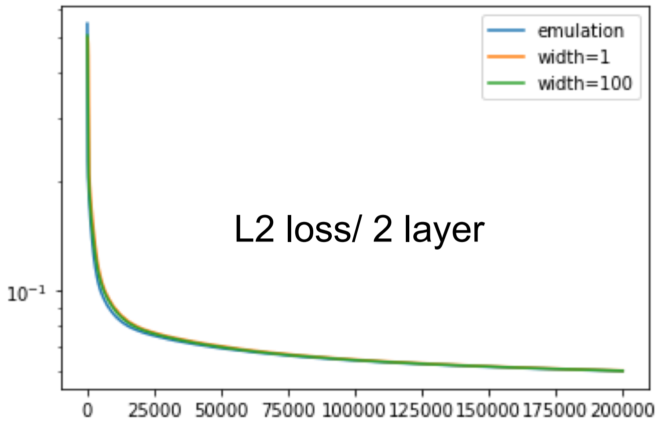


This experiment shows that the theoretical update rule can also empirically explain the update of deep linear neural network.

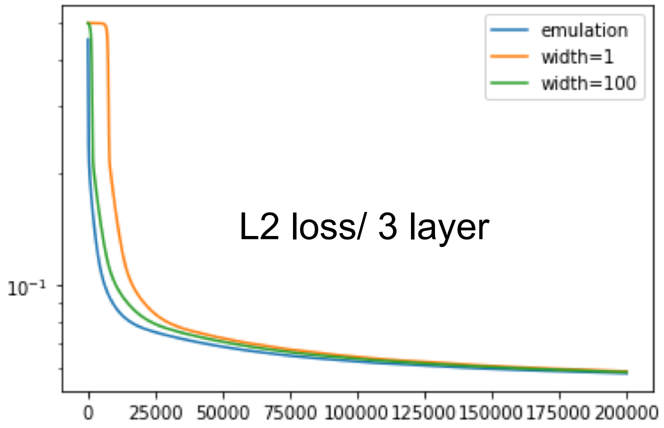
We can also see that the width of network is not related to the converging speed.

Experimental Results

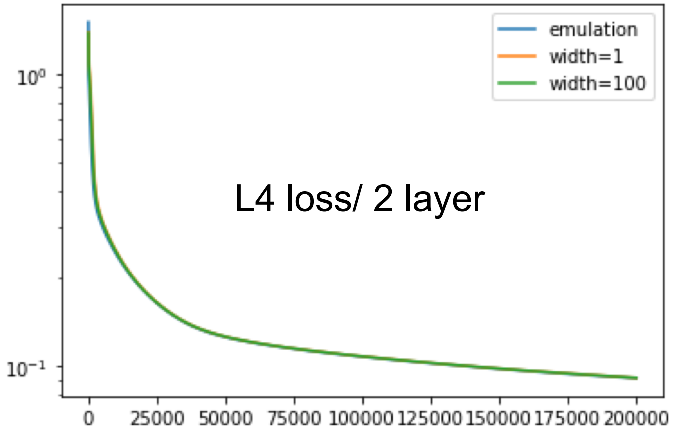
Experiment 1



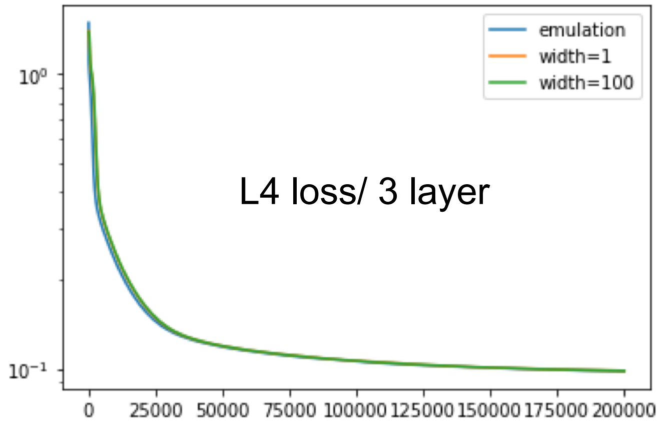
L2 loss/ 2 layer



L2 loss/ 3 layer



L4 loss/ 2 layer

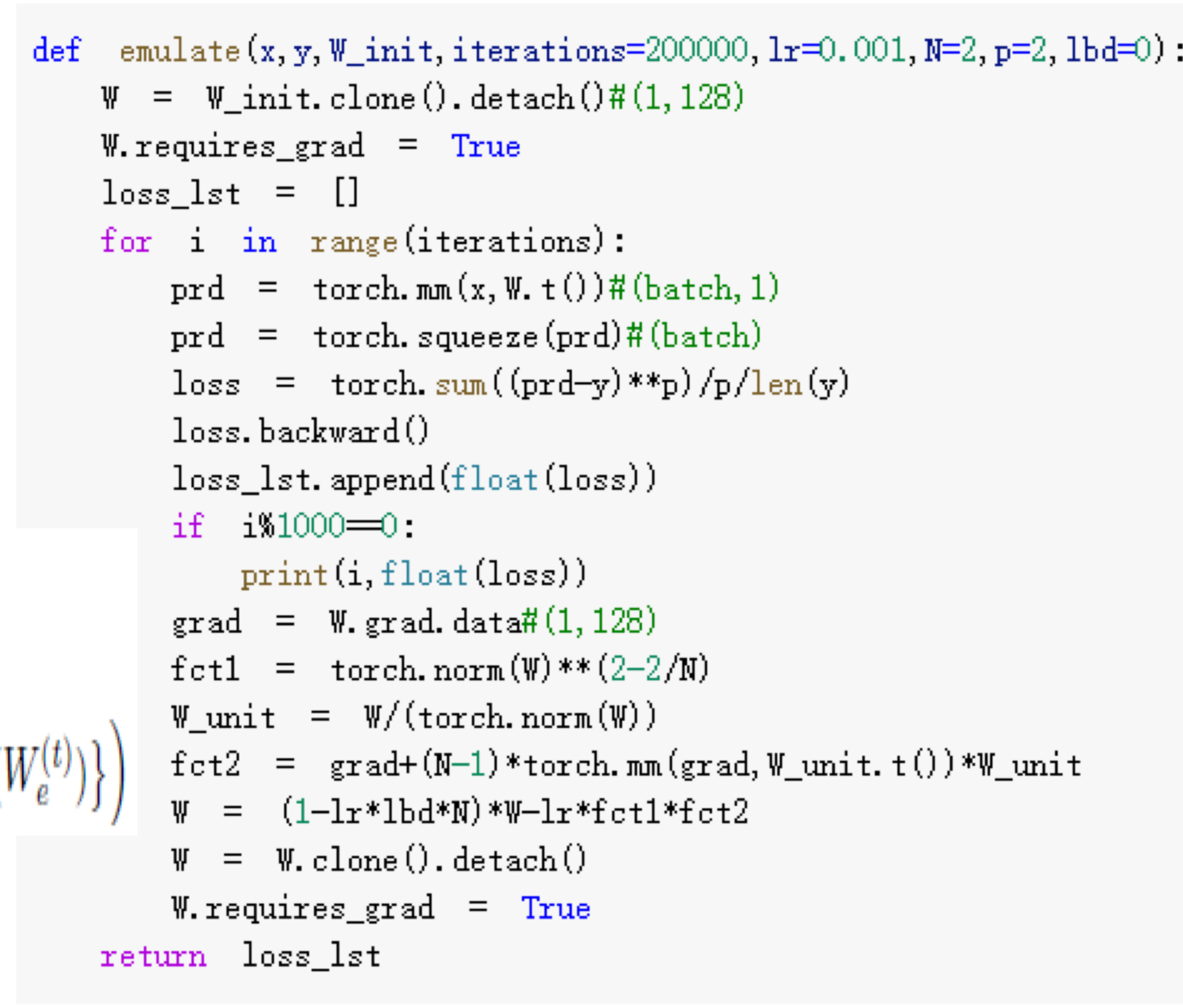


L4 loss/ 3 layer

Experiment 1 Code

emulate function implements the theoretical update rule of linear neural network.

$$W_e^{(t+1)} \leftarrow (1-\eta\lambda N)\cdot W_e^{(t)} - \eta\|W_e^{(t)}\|_2^{2-\frac{2}{N}} \cdot \left(\frac{dL^1}{dW}(W_e^{(t)}) + (N-1)\cdot Pr_{W_e^{(t)}}\left\{\frac{dL^1}{dW}(W_e^{(t)})\right\}\right)$$



```
def  emulate(x,y,W_init,iterations=200000,lr=0.001,N=2,p=2,lbd=0):
    W  =  W_init.clone().detach()#(1,128)
    W.requires_grad  =  True
    loss_lst  =  []
    for  i  in  range(iterations):
        prd  =  torch.mm(x,W.t())#(batch,1)
        prd  =  torch.squeeze(prd)#(batch)
        loss  =  torch.sum((prd-y)**p)/p/len(y)
        loss.backward()
        loss_lst.append(float(loss))
        if  i%1000==0:
            print(i,float(loss))
        grad  =  W.grad.data#(1,128)
        fct1  =  torch.norm(W)**(2-2/N)
        W_unit  =  W/(torch.norm(W))
        fct2  =  grad+(N-1)*torch.mm(grad,W_unit.t())*W_unit
        W  =  (1-lr*lbd*N)*W-lr*fct1*fct2
        W  =  W.clone().detach()
        W.requires_grad  =  True
    return  loss_lst
```

Experiment 1 Code

Step 1. Calculate loss

```
loss = torch.sum((prd-y)**p)/p/len(y)
```

Step 2. Get gradient

```
loss.backward()
```

```
grad = W.grad.data
```

Step 3. Calculate adaptive learning rate scalar

```
fct1 = torch.norm(W)**(2-2/N)
```

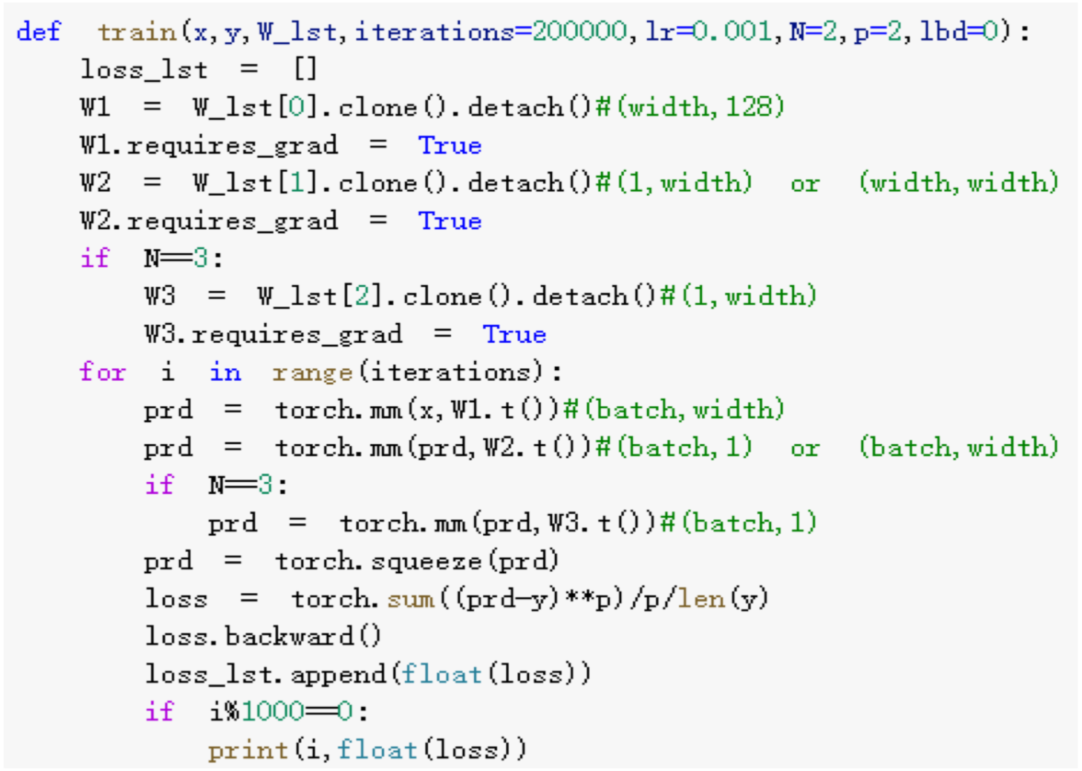
Step 4. Calculate the projection of gradient on the direction of W

```
W_unit = W/(torch.norm(W))
fct2 = grad+(N-1)*torch.mm(grad,W_unit.t())*W_unit
```

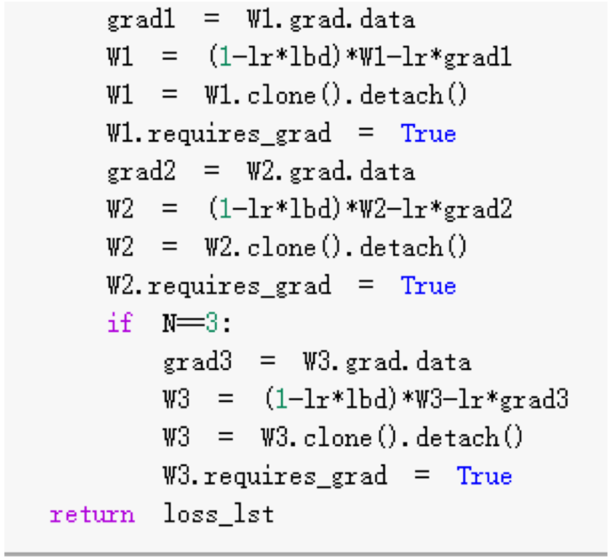
Step 5. Update W

```
W = (1-lr*lbd*N)*W-lr*fct1*fct2
```

Experiment 1 Code



```
def  train(x,y,W_lst,iterations=200000,lr=0.001,N=2,p=2,lbd=0):
    loss_lst  =  []
    W1  =  W_lst[0].clone().detach()#(width,128)
    W1.requires_grad  =  True
    W2  =  W_lst[1].clone().detach()#(1,width)  or  (width,width)
    W2.requires_grad  =  True
    if  N==3:
        W3  =  W_lst[2].clone().detach()#(1,width)
        W3.requires_grad  =  True
    for  i  in  range(iterations):
        prd  =  torch.mm(x,W1.t())#(batch,width)
        prd  =  torch.mm(prd,W2.t())#(batch,1)  or  (batch,width)
        if  N==3:
            prd  =  torch.mm(prd,W3.t())#(batch,1)
        prd  =  torch.squeeze(prd)
        loss  =  torch.sum((prd-y)**p)/p/len(y)
        loss.backward()
        loss_lst.append(float(loss))
        if  i%1000==0:
            print(i,float(loss))
```

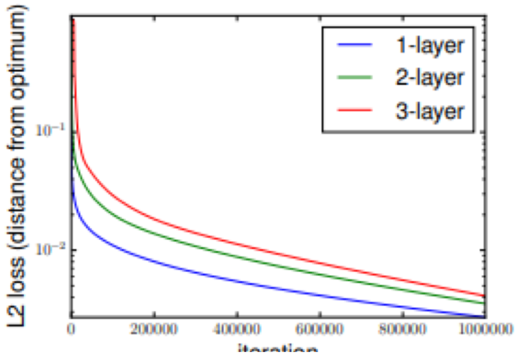


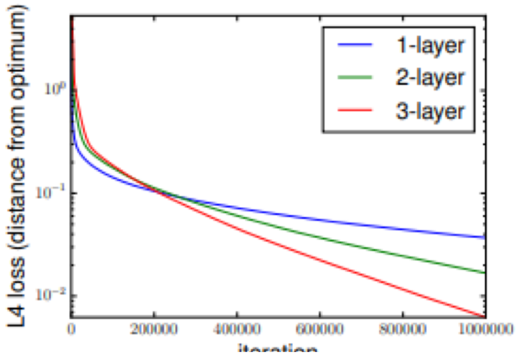
```
        grad1  =  W1.grad.data
        W1  =  (1-lr*lbd)*W1-lr*grad1
        W1  =  W1.clone().detach()
        W1.requires_grad  =  True
        grad2  =  W2.grad.data
        W2  =  (1-lr*lbd)*W2-lr*grad2
        W2  =  W2.clone().detach()
        W2.requires_grad  =  True
        if  N==3:
            grad3  =  W3.grad.data
            W3  =  (1-lr*lbd)*W3-lr*grad3
            W3  =  W3.clone().detach()
            W3.requires_grad  =  True
    return  loss_lst
```

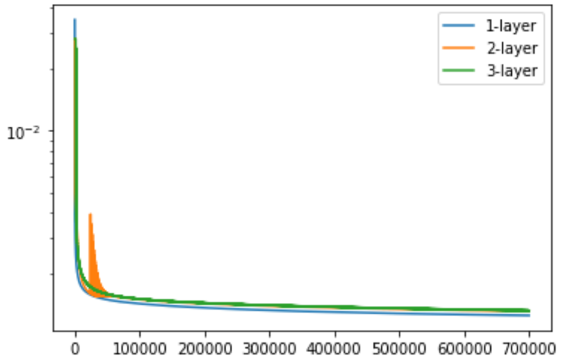
This function simply trains a linear neural network using gradient descent.

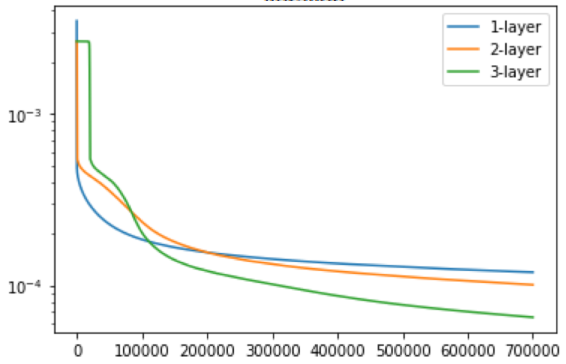
Experimental Results

Experiment 2









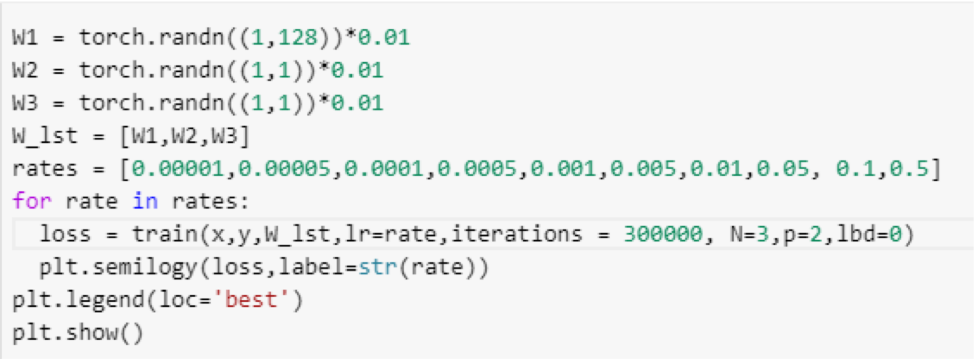
Figures show the convergence of the gradient descent of optimization of single layer against depth-2 and depth-3.

With L2 loss, deeper networks show slower convergence rate. However, with L4 loss, consist of quantitative analysis, depth indicates optimization.

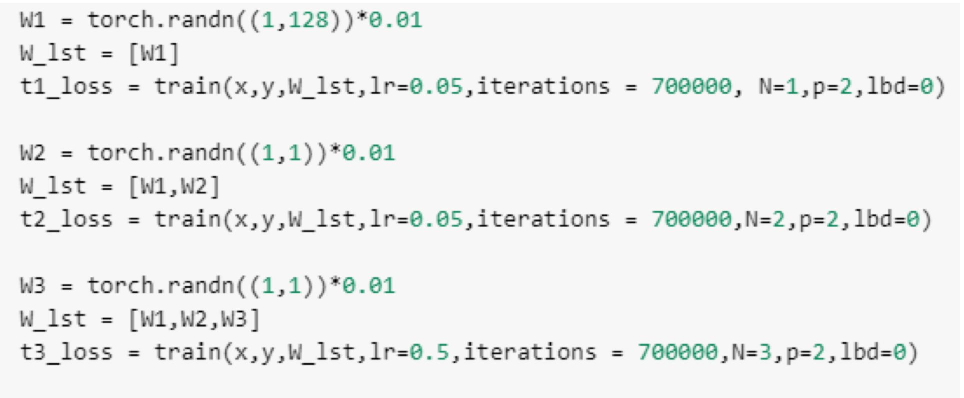
Gradient descent optimization of single layer model vs. linear networks of depth 2 and 3

Experimental Results

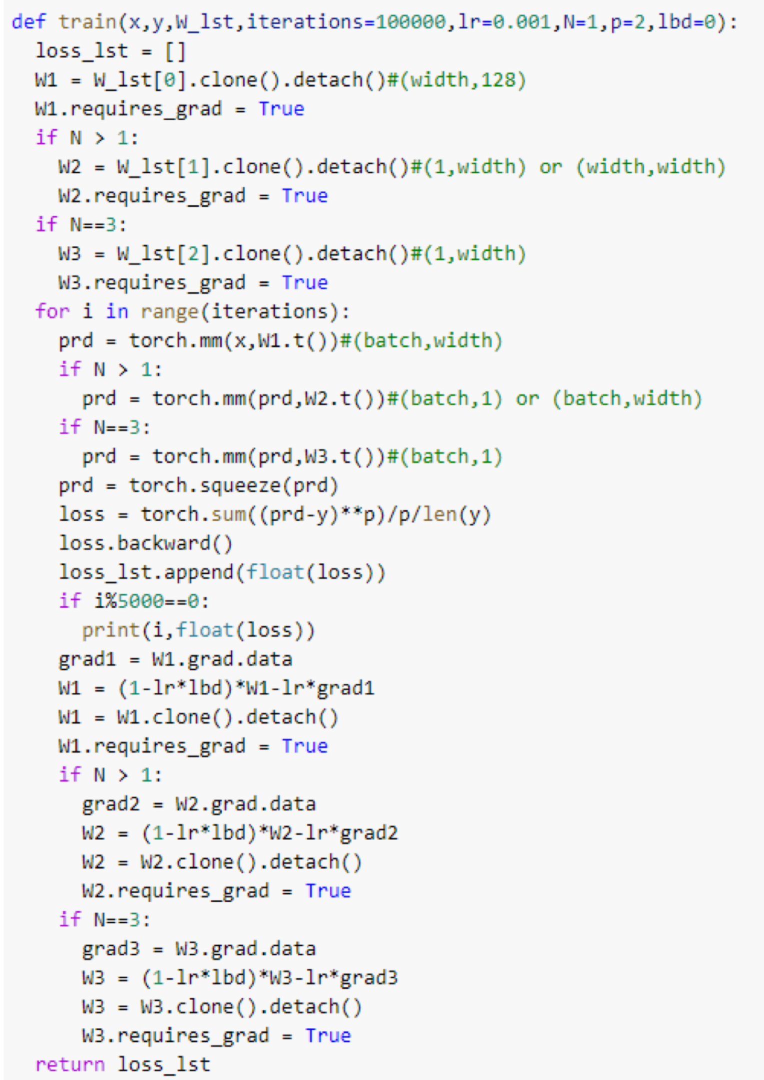
Experiment 2



```python
W1 = torch.randn((1,128))*0.01
W2 = torch.randn((1,1))*0.01
W3 = torch.randn((1,1))*0.01
W_lst = [W1,W2,W3]
rates = [0.00001,0.00005,0.0001,0.0005,0.001,0.005,0.01,0.05, 0.1,0.5]
for rate in rates:
  loss = train(x,y,W_lst,lr=rate,iterations = 300000, N=3,p=2,lbd=0)
  plt.semilogy(loss,label=str(rate))
plt.legend(loc='best')
plt.show()
```

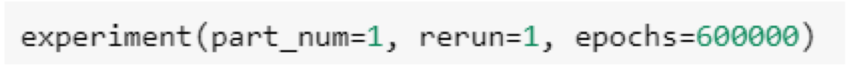


```python
W1 = torch.randn((1,128))*0.01
W_lst = [W1]
t1_loss = train(x,y,W_lst,lr=0.05,iterations = 700000, N=1,p=2,lbd=0)

W2 = torch.randn((1,1))*0.01
W_lst = [W1,W2]
t2_loss = train(x,y,W_lst,lr=0.05,iterations = 700000,N=2,p=2,lbd=0)

W3 = torch.randn((1,1))*0.01
W_lst = [W1,W2,W3]
t3_loss = train(x,y,W_lst,lr=0.5,iterations = 700000,N=3,p=2,lbd=0)
```

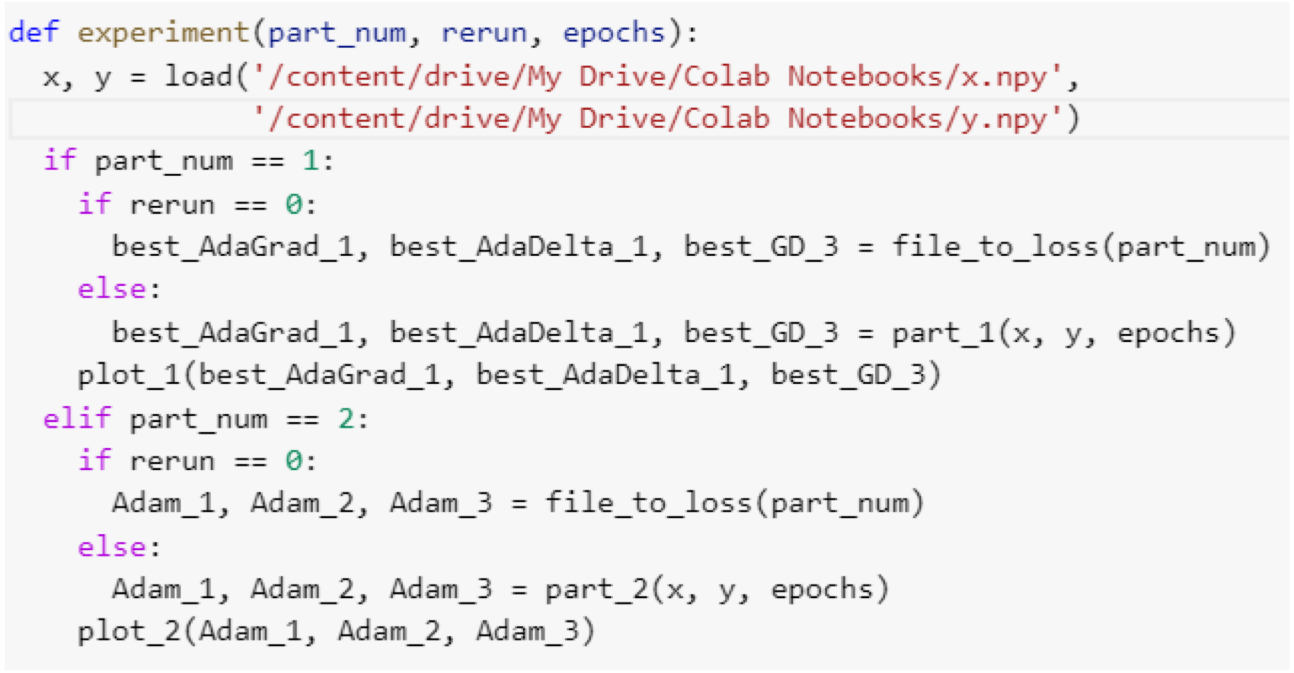


```python
def train(x,y,W_lst,iterations=100000,lr=0.001,N=1,p=2,lbd=0):
  loss_lst = []
  W1 = W_lst[0].clone().detach()#(width,128)
  W1.requires_grad = True
  if N > 1:
    W2 = W_lst[1].clone().detach()#(1,width) or (width,width)
    W2.requires_grad = True
  if N==3:
    W3 = W_lst[2].clone().detach()#(1,width)
    W3.requires_grad = True
  for i in range(iterations):
    prd = torch.mm(x,W1.t())#(batch,width)
    if N > 1:
      prd = torch.mm(prd,W2.t())#(batch,1) or (batch,width)
    if N==3:
      prd = torch.mm(prd,W3.t())#(batch,1)
    prd = torch.squeeze(prd)
    loss = torch.sum((prd-y)**p)/p/len(y)
    loss.backward()
    loss_lst.append(float(loss))
    if i%5000==0:
      print(i,float(loss))
    grad1 = W1.grad.data
    W1 = (1-lr*lbd)*W1-lr*grad1
    W1 = W1.clone().detach()
    W1.requires_grad = True
    if N > 1:
      grad2 = W2.grad.data
      W2 = (1-lr*lbd)*W2-lr*grad2
      W2 = W2.clone().detach()
      W2.requires_grad = True
    if N==3:
      grad3 = W3.grad.data
      W3 = (1-lr*lbd)*W3-lr*grad3
      W3 = W3.clone().detach()
      W3.requires_grad = True
  return loss_lst
```

Experiment 3 (Part 1)



```
experiment(part_num=1, rerun=1, epochs=600000)
```



```
def experiment(part_num, rerun, epochs):
  x, y = load('/content/drive/My Drive/Colab Notebooks/x.npy',
              '/content/drive/My Drive/Colab Notebooks/y.npy')
  if part_num == 1:
    if rerun == 0:
      best_AdaGrad_1, best_AdaDelta_1, best_GD_3 = file_to_loss(part_num)
    else:
      best_AdaGrad_1, best_AdaDelta_1, best_GD_3 = part_1(x, y, epochs)
    plot_1(best_AdaGrad_1, best_AdaDelta_1, best_GD_3)
  elif part_num == 2:
    if rerun == 0:
      Adam_1, Adam_2, Adam_3 = file_to_loss(part_num)
    else:
      Adam_1, Adam_2, Adam_3 = part_2(x, y, epochs)
    plot_2(Adam_1, Adam_2, Adam_3)
```

Setting

- Linear neural network:
  AdaGrad
  AdaDelta
  Overparameterization

- Epoch number:
  600,000 epochs

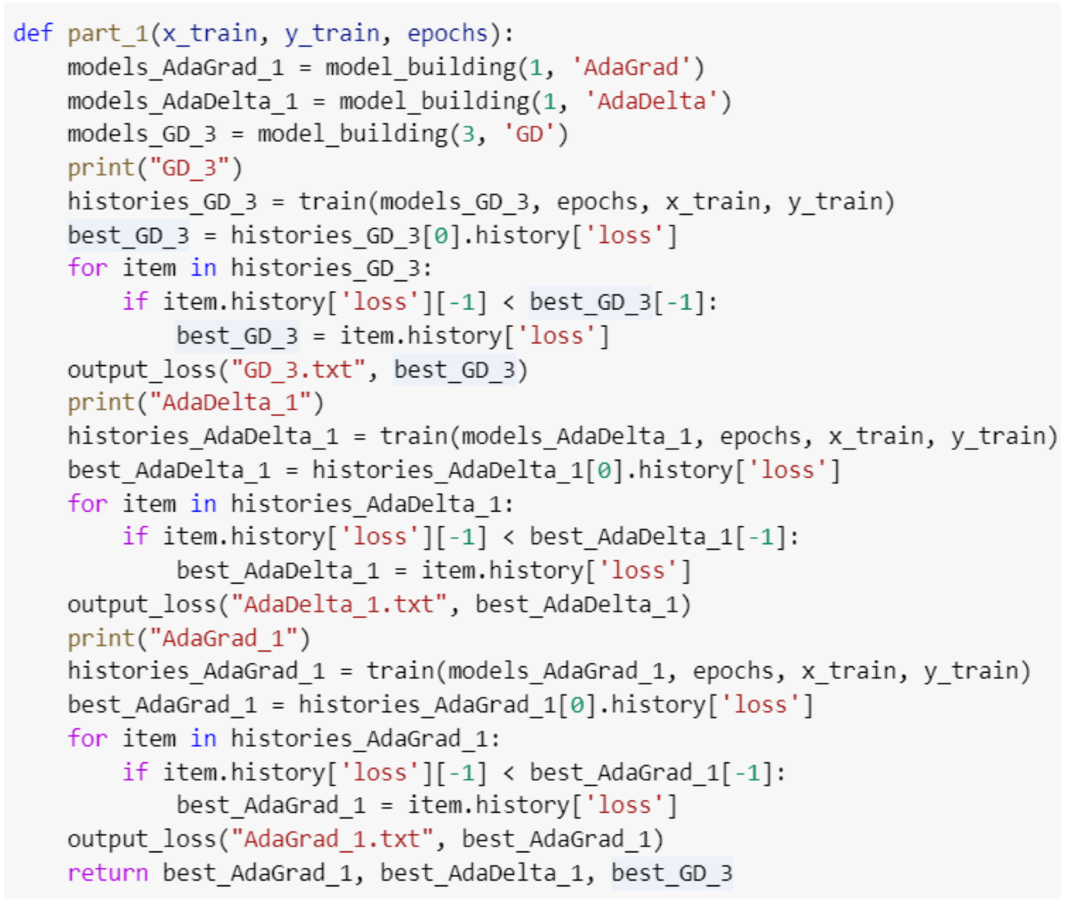
Experiment 3 (Part 1)

Setting

- Linear neural network
  AdaGrad
  AdaDelta
  Overparameterization

- Epoch number:
  600,000 epochs

- Choose the best model according to the loss in the last epoch

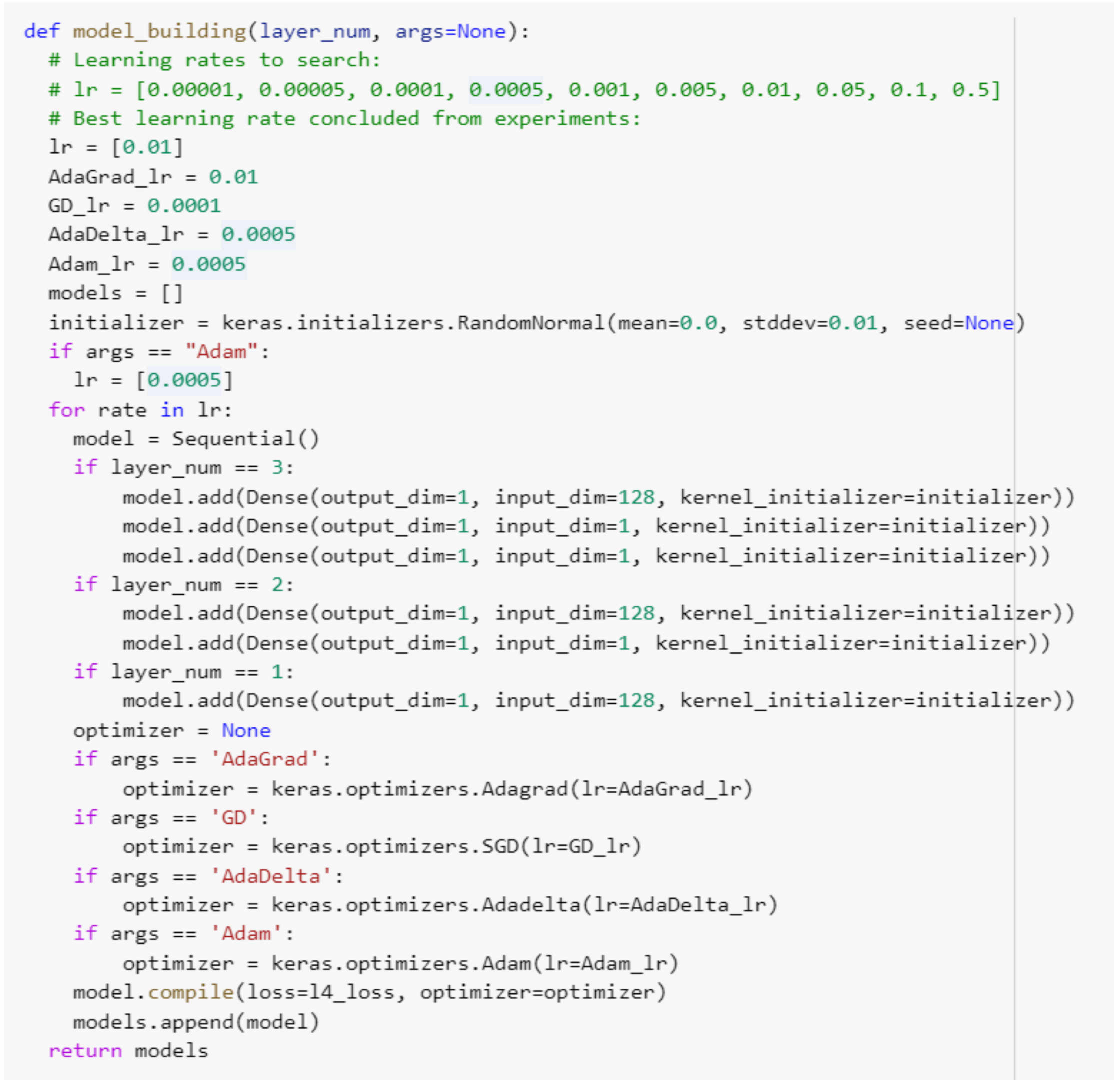


```python
def part_1(x_train, y_train, epochs):
    models_AdaGrad_1 = model_building(1, 'AdaGrad')
    models_AdaDelta_1 = model_building(1, 'AdaDelta')
    models_GD_3 = model_building(3, 'GD')
    print("GD_3")
    histories_GD_3 = train(models_GD_3, epochs, x_train, y_train)
    best_GD_3 = histories_GD_3[0].history['loss']
    for item in histories_GD_3:
        if item.history['loss'][-1] < best_GD_3[-1]:
            best_GD_3 = item.history['loss']
    output_loss("GD_3.txt", best_GD_3)
    print("AdaDelta_1")
    histories_AdaDelta_1 = train(models_AdaDelta_1, epochs, x_train, y_train)
    best_AdaDelta_1 = histories_AdaDelta_1[0].history['loss']
    for item in histories_AdaDelta_1:
        if item.history['loss'][-1] < best_AdaDelta_1[-1]:
            best_AdaDelta_1 = item.history['loss']
    output_loss("AdaDelta_1.txt", best_AdaDelta_1)
    print("AdaGrad_1")
    histories_AdaGrad_1 = train(models_AdaGrad_1, epochs, x_train, y_train)
    best_AdaGrad_1 = histories_AdaGrad_1[0].history['loss']
    for item in histories_AdaGrad_1:
        if item.history['loss'][-1] < best_AdaGrad_1[-1]:
            best_AdaGrad_1 = item.history['loss']
    output_loss("AdaGrad_1.txt", best_AdaGrad_1)
    return best_AdaGrad_1, best_AdaDelta_1, best_GD_3
```

Experiment 3 (Part 1)

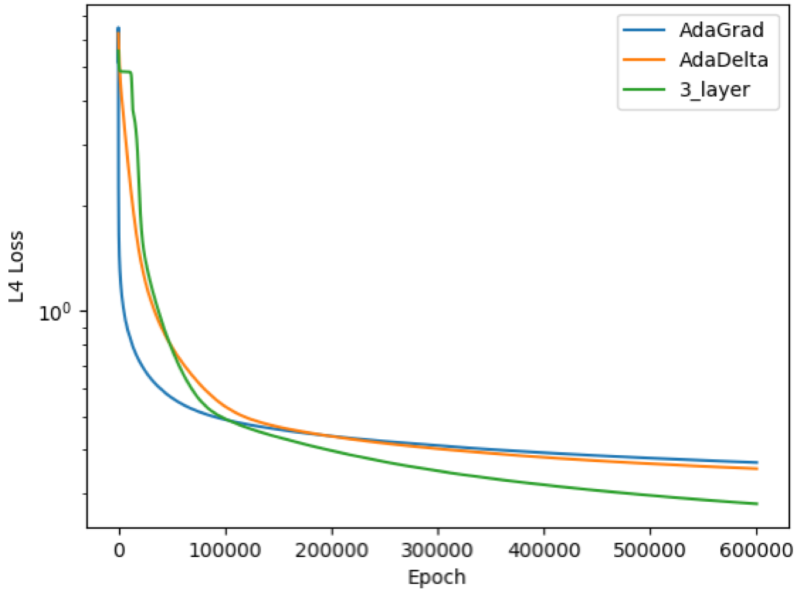
Setting

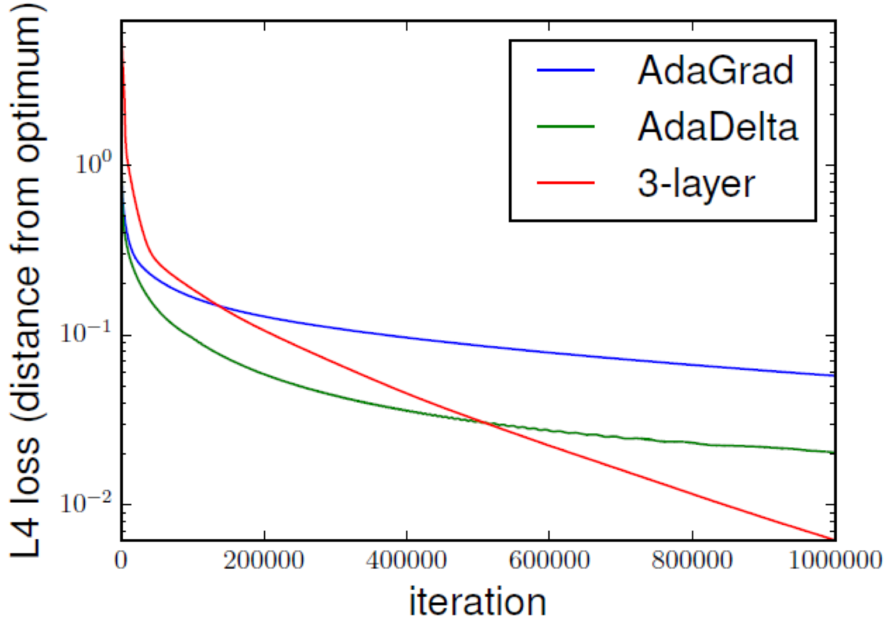
- Linear neural network
  AdaGrad: 128 * 1
  AdaDelta: 128 * 1
  Overparameterization: 128 * 1, 1 * 1, 1 * 1
- Learning rates for each model to search: [0.00001, 0.00005, 0.0001, 0.0005, 0.001, 0.005, 0.01, 0.05, 0.1, 0.5]
- Choose the best model according to the loss in the last epoch
- Epoch number: 600,000 epochs



```
def model_building(layer_num, args=None):
  # Learning rates to search:
  # lr = [0.00001, 0.00005, 0.0001, 0.0005, 0.001, 0.005, 0.01, 0.05, 0.1, 0.5]
  # Best learning rate concluded from experiments:
  lr = [0.01]
  AdaGrad_lr = 0.01
  GD_lr = 0.0001
  AdaDelta_lr = 0.0005
  Adam_lr = 0.0005
  models = []
  initializer = keras.initializers.RandomNormal(mean=0.0, stddev=0.01, seed=None)
  if args == "Adam":
    lr = [0.0005]
  for rate in lr:
    model = Sequential()
    if layer_num == 3:
        model.add(Dense(output_dim=1, input_dim=128, kernel_initializer=initializer))
        model.add(Dense(output_dim=1, input_dim=1, kernel_initializer=initializer))
        model.add(Dense(output_dim=1, input_dim=1, kernel_initializer=initializer))
    if layer_num == 2:
        model.add(Dense(output_dim=1, input_dim=128, kernel_initializer=initializer))
        model.add(Dense(output_dim=1, input_dim=1, kernel_initializer=initializer))
    if layer_num == 1:
        model.add(Dense(output_dim=1, input_dim=128, kernel_initializer=initializer))
    optimizer = None
    if args == 'AdaGrad':
        optimizer = keras.optimizers.Adagrad(lr=AdaGrad_lr)
    if args == 'GD':
        optimizer = keras.optimizers.SGD(lr=GD_lr)
    if args == 'AdaDelta':
        optimizer = keras.optimizers.Adadelta(lr=AdaDelta_lr)
    if args == 'Adam':
        optimizer = keras.optimizers.Adam(lr=Adam_lr)
    model.compile(loss=l4_loss, optimizer=optimizer)
    models.append(model)
  return models
```

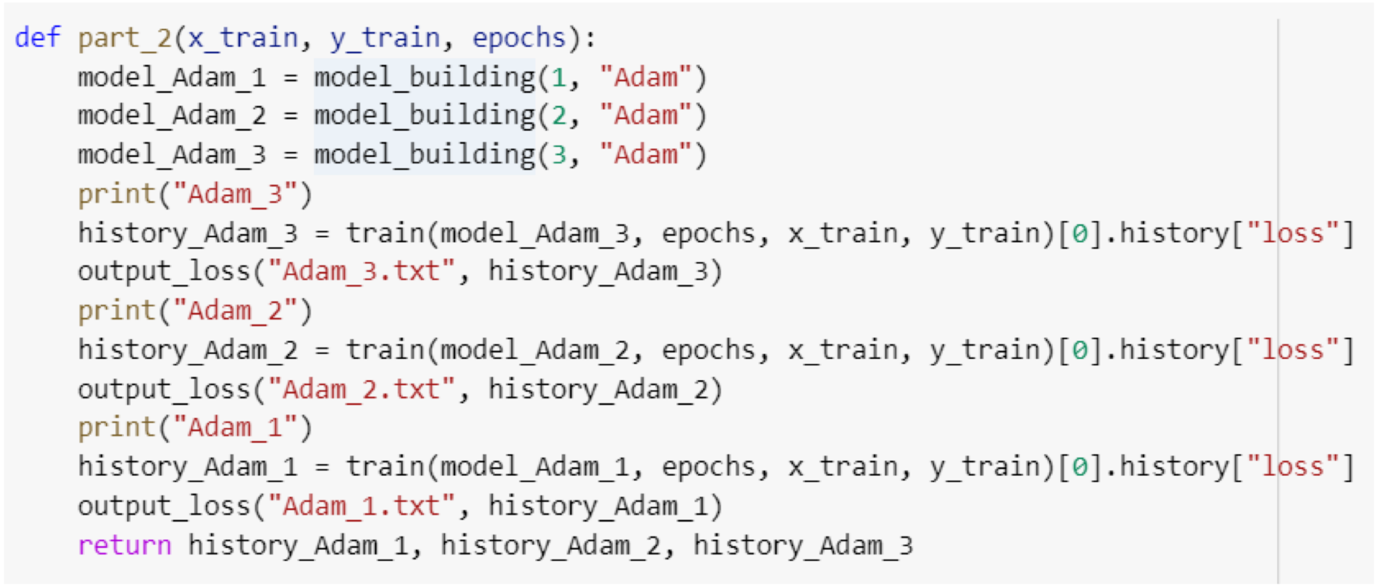
Experiment 3 (Part 1): Result





In this paper's setting, overparameterization is a more effective optimization strategy than some carefully designed algorithms tailored for convex problems.

Experiment 3 (Part 2)



```
def part_2(x_train, y_train, epochs):
    model_Adam_1 = model_building(1, "Adam")
    model_Adam_2 = model_building(2, "Adam")
    model_Adam_3 = model_building(3, "Adam")
    print("Adam_3")
    history_Adam_3 = train(model_Adam_3, epochs, x_train, y_train)[0].history["loss"]
    output_loss("Adam_3.txt", history_Adam_3)
    print("Adam_2")
    history_Adam_2 = train(model_Adam_2, epochs, x_train, y_train)[0].history["loss"]
    output_loss("Adam_2.txt", history_Adam_2)
    print("Adam_1")
    history_Adam_1 = train(model_Adam_1, epochs, x_train, y_train)[0].history["loss"]
    output_loss("Adam_1.txt", history_Adam_1)
    return history_Adam_1, history_Adam_2, history_Adam_3
```

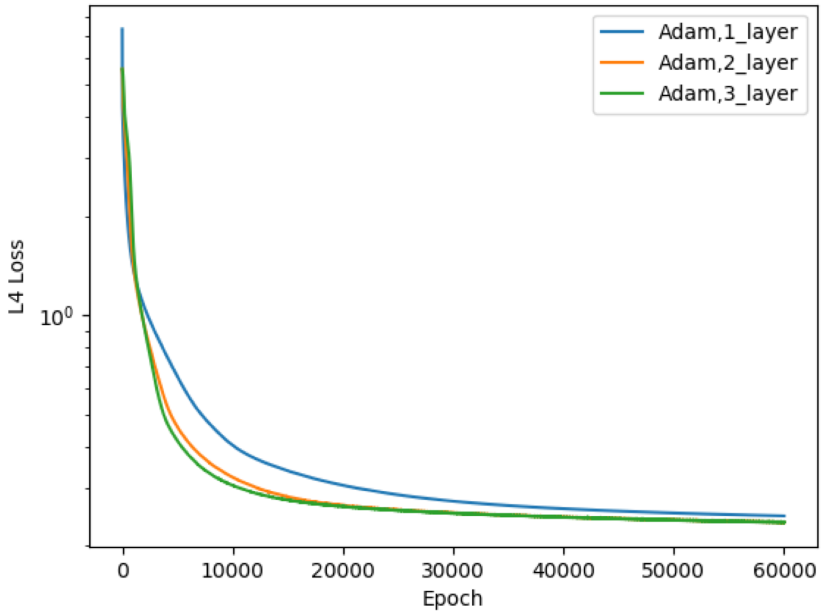
Setting

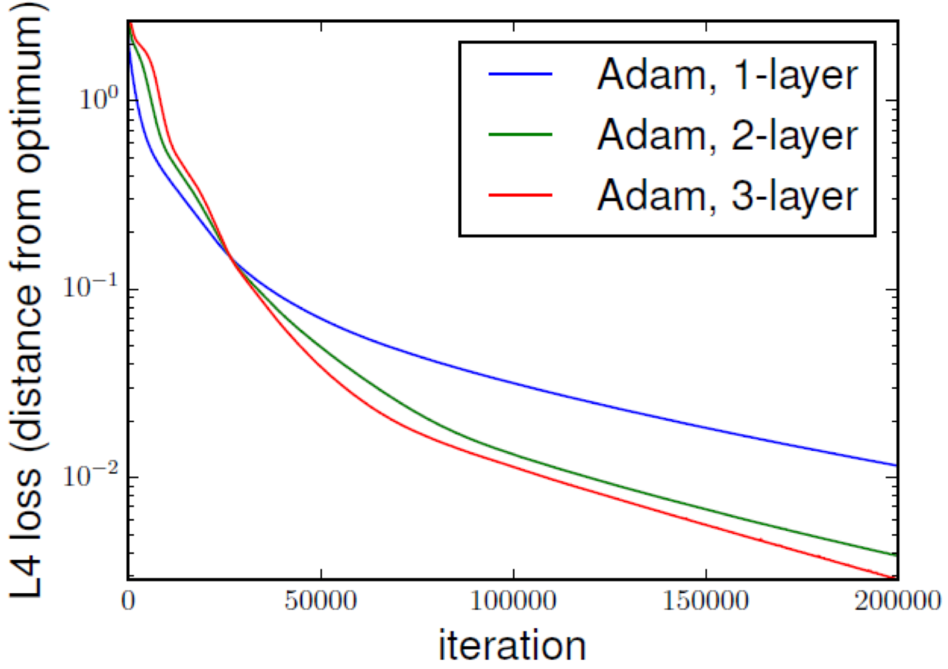
- Linear neural network with the optimizer Adam:
  1 layer: 128 * 1
  2 layers: 128 * 1, 1 * 1
  3 layers: 128 * 1, 1 * 1, 1 * 1

- Fixed learning rate: 0.0005
  (Learning rate is 0.001 in the paper.)

- Epoch number: 60,000 epochs

Experiment 3 (Part 2): Result





- When introducing overparameterization simultaneously with Adam, further acceleration is attained.
- This suggests that at least in some cases, not only plain gradient descent benefits from depth, but also more elaborate algorithms commonly employed in state of the art applications.

Experiment 4 (MNIST Convolutional Network)

- This is an example implicit acceleration of overparameterization on a nonlinear model by replacing hidden layers with depth-2 linear networks
- Two minor changes:

Hidden dense layer: 3136×512 weight matrix replaced by multiplication of 3136×512 and 512×512 matrices.

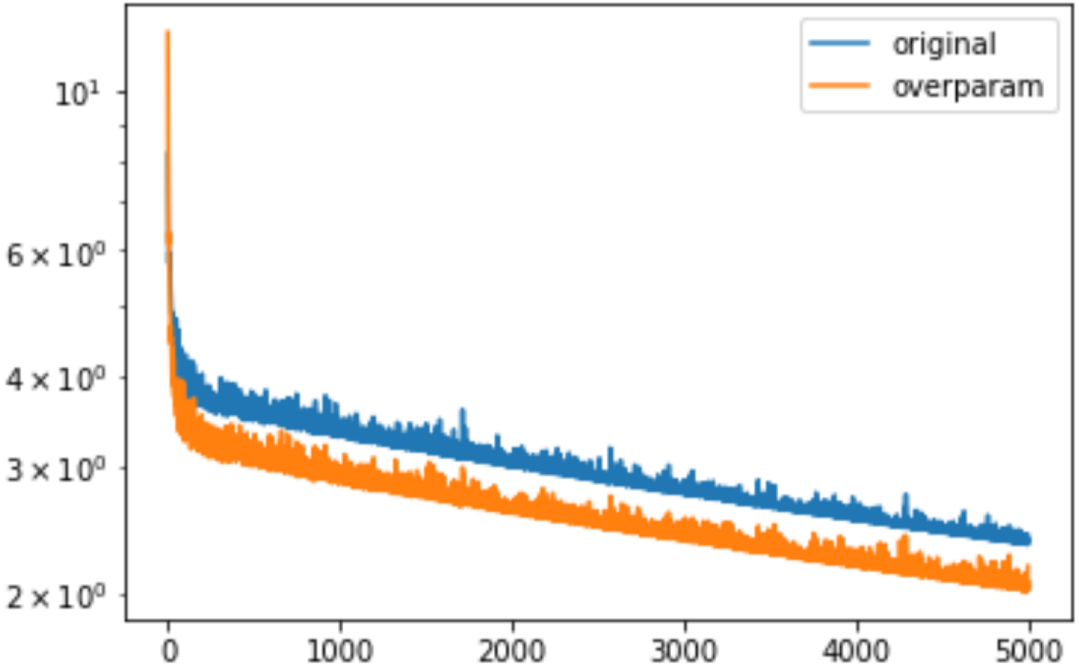
Output layer: 512×10 weight matrix replaced by multiplication of 512×10 and 10×10 matrices

Code

```
fc1_weights = tf.Variable(  # fully connected, depth 512.
    tf.truncated_normal([IMAGE_SIZE // 4 * IMAGE_SIZE // 4 * 64, 512],
                        stddev=0.1,
                        seed=SEED,
                        dtype=data_type()))
fc1_weights_op = tf.Variable(  # fully connected, depth 512.
    tf.truncated_normal([512, 512],
                        stddev=0.1,
                        seed=SEED,
                        dtype=data_type()))
fc1_biases = tf.Variable(tf.constant(0.1, shape=[512], dtype=data_type()))
fc2_weights = tf.Variable(tf.truncated_normal([512, NUM_LABELS],
                                              stddev=0.1,
                                              seed=SEED,
                                              dtype=data_type()))
fc2_weights_op = tf.Variable(tf.truncated_normal([NUM_LABELS, NUM_LABELS],
                                              stddev=0.1,
                                              seed=SEED,
                                              dtype=data_type()))
fc2_biases = tf.Variable(tf.constant(
    0.1, shape=[NUM_LABELS], dtype=data_type()))
```

```
# fully connected layers.
pool_shape = pool.get_shape().as_list()
reshape = tf.reshape(
    pool,
    [pool_shape[0], pool_shape[1] * pool_shape[2] * pool_shape[3]])
# Fully connected layer. Note that the '+' operation automatically
# broadcasts the biases.
hidden = tf.nn.relu(tf.matmul(tf.matmul(reshape, fc1_weights), fc1_weights_op) + fc1_biases)
# Add a 50% dropout during training only. Dropout also scales
# activations such that no rescaling is needed at evaluation time.
if train:
  hidden = tf.nn.dropout(hidden, 0.5, seed=SEED)
return tf.matmul(tf.matmul(hidden, fc2_weights), fc2_weights_op) + fc2_biases
```

Result



- As reported above for linear networks, it is likely that for non-linear networks the effect of depth on optimization is mixed – some settings accelerate by it, while others do not.
- Comprehensive characterization of the cases in which depth accelerates optimization warrants much further study.

Experimental Analysis

- Experiments are consistent with the predicted results from theoretical derivation.
- Experiments are conducted with Linear Neural Network to rule out the factor of expressiveness.
- Experimental results indicates that overparameterization by depth can induce a faster model training based on gradient descent over a convex problem.
- With sanity test, the experiment with convolutional network indicates that overparameterization could also be useful in some non-idealized deep learning settings.

Outline

Conclusion and Future Work

Overparameterization by depth can accelerate optimization.

- Linear neural networks
  - A preconditioning scheme: a combination between certain forms of adaptive learning rate and momentum.
  - Depends on depth, instead of width: minimal computational price.
- Non-linear neural networks
  - Challenging to theoretically analyze
  - Empirically, replacing an internal weight matrix by a product of two accelerates optimization, with expressiveness unchanged

Analysis is based on gradient descent over classic convex problems.

How about other explicit acceleration methods? Momentum? Adagrad?

Can we quantify the effect?

Job Split

Fuxiao Liu: Experiment 4, slides making, presentation

Yinzhu Jin: Experiment 1, theoretical derivation, slides making, presentation

Dexuan Zhang: Experiment 2, slides making, presentation

Tianyang Luo: Experiment 3, slides making, presentation

Reference

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J.,Devin, M., Ghemawat, S., Irving, G., Isard, M., et al. Tensorflow: A system for large-scale machine learning. In OSDI, volume 16, pp. 265–283, 2016.

Arora, R., Basu, A., Mianjy, P., and Mukherjee, A. Understanding deep neural networks with rectified linear units. International Conference on Learning Representations (ICLR), 2018.

Baldi, P. and Hornik, K. Neural networks and principal component analysis: Learning from examples without local minima. Neural networks, 2(1):53–58, 1989.

Boyce, W. E., DiPrima, R. C., and Haines, C. W. Elementary differential equations and boundary value problems, volume 9. Wiley New York, 1969.

Buck, R. C. Advanced calculus. Waveland Press, 2003. Choromanska, A., Henaff, M., Mathieu, M., Arous, G. B., and LeCun, Y. The loss surfaces of multilayer networks. In Artificial Intelligence and Statistics, pp. 192–204, 2015.

Cohen, N., Sharir, O., Levine, Y., Tamari, R., Yakira, D., and Shashua, A. Analysis and design of convolutional networks via hierarchical tensor decompositions. arXiv preprint arXiv:1705.02302, 2017.

Daniely, A. Depth separation for neural networks. arXiv preprint arXiv:1702.08489, 2017.

Duchi, J., Hazan, E., and Singer, Y. Adaptive subgradient methods for online learning and stochastic optimization. Journal of Machine Learning Research, 12(Jul):2121–2159, 2011.

Eldan, R. and Shamir, O. The power of depth for feedforward neural networks. arXiv preprint arXiv:1512.03965, 2015.

Fukumizu, K. Effect of batch learning in multilayer neural networks. Gen, 1(04):1E–03, 1998.

Goh, G. Why momentum really works. Distill, 2017. doi: 10. 23915/distill.00006. URL http://distill.pub/2017/Momentum.

Goodfellow, I., Bengio, Y., Courville, A., and Bengio, Y. Deep learning, volume 1. MIT press Cambridge, 2016.

Goodfellow, I. J., Vinyals, O., and Saxe, A. M. Qualitatively characterizing neural network optimization problems. arXiv preprint arXiv:1412.6544, 2014.

Haeffele, B. D. and Vidal, R. Global Optimality in Tensor Factorization, Deep Learning, and Beyond. CoRR abs/1202.2745, cs.NA, 2015.

Hardt, M. and Ma, T. Identity matters in deep learning. arXiv preprint arXiv:1611.04231, 2016.

Hazan, E., Agarwal, A., and Kale, S. Logarithmic regret algorithms for online convex optimization. Mach. Learn., 69(2-3):169–192, December 2007. ISSN 0885-6125.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. arXiv preprint arXiv:1512.03385, 2015.

Helmke, U. and Moore, J. B. Optimization and dynamical systems. Springer Science & Business Media, 2012.

Ioffe, S. and Szegedy, C. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In International conference on machine learning, pp. 448–456, 2015.

Jones, E., Oliphant, T., Peterson, P., et al. SciPy: Open source scientific tools for Python, 2001–. URL http://www.scipy.org/. [Online; accessed ¡today¿].

Kawaguchi, K. Deep learning without poor local minima. In Advances in Neural Information Processing Systems, pp. 586– 594, 2016.

Kingma, D. and Ba, J. Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980, 2014.

Lee, H., Ge, R., Risteski, A., Ma, T., and Arora, S. On the ability of neural nets to express distributions. arXiv preprint arXiv:1702.07028, 2017.

Nesterov, Y. A method of solving a convex programming problem with convergence rate o (1/k2). In Soviet Mathematics Doklady, volume 27, pp. 372–376, 1983.

Nocedal, J. Updating quasi-newton matrices with limited storage. Mathematics of Computation, 35(151):773–782, 1980.

Raghu, M., Poole, B., Kleinberg, J., Ganguli, S., and Sohl-Dickstein, J. On the expressive power of deep neural networks. arXiv preprint arXiv:1606.05336, 2016.

Rodriguez-Lujan, I., Fonollosa, J., Vergara, A., Homer, M., and Huerta, R. On the calibration of sensor arrays for pattern recognition using the minimal number of experiments. Chemometrics and Intelligent Laboratory Systems, 130:123–134, 2014.

Safran, I. and Shamir, O. On the quality of the initial basin in overspecified neural networks. In International Conference on Machine Learning, pp. 774–782, 2016.

Safran, I. and Shamir, O. Spurious local minima are common in two-layer relu neural networks. arXiv preprint arXiv:1712.08968, 2017.

Saxe, A. M., McClelland, J. L., and Ganguli, S. Exact solutions to the nonlinear dynamics of learning in deep linear neural networks. arXiv preprint arXiv:1312.6120, 2013.

Soudry, D. and Carmon, Y. No bad local minima: Data independent training error guarantees for multilayer neural networks. arXiv preprint arXiv:1605.08361, 2016.

Srivastava, N., Hinton, G. E., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. Dropout: a simple way to prevent neural networks from overfitting. Journal of Machine Learning Research, 15(1):1929–1958, 2014.

Su, W., Boyd, S., and Candes, E. A differential equation for modeling nesterovs accelerated gradient method: Theory and insights. In Advances in Neural Information Processing Systems, pp. 2510–2518, 2014.

Tieleman, T. and Hinton, G. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. COURSERA: Neural networks for machine learning, 4(2):26–31, 2012.

Vergara, A., Vembu, S., Ayhan, T., Ryan, M. A., Homer, M. L., and Huerta, R. Chemical gas sensor drift compensation using classifier ensembles. Sensors and Actuators B: Chemical, 166: 320–329, 2012.

Wibisono, A., Wilson, A. C., and Jordan, M. I. A variational perspective on accelerated methods in optimization. Proceedings of the National Academy of Sciences, 113(47):E7351–E7358, 2016.

Zeiler, M. D. Adadelta: an adaptive learning rate method. arXiv preprint arXiv:1212.5701, 2012.

Thank you!