Chapter 4

Rachel Hu

April 29, 2019

```
In [2]: import d2l
        import mxnet as mx
        from mxnet import autograd, gluon, init, nd
        from mxnet.gluon import loss as gloss, data as gdata, nn

        import time

        import numpy as np

        from IPython.core.interactiveshell import InteractiveShell
        InteractiveShell.ast_node_interactivity = "all"
```

4.1.4. Exercises

1. **Compute the derivative of the tanh and the pReLU activation function.**

   a. The derivative of the Tanh function is:

   $$\frac{d}{dx}\tanh(x) = 1 - \tanh^2(x).$$

   b. The derivative of the pReLU function is:

   $$\text{pReLU}(x) = \max(0, x) + \alpha \min(0, x)$$

   $$\frac{d}{dx}\text{pReLU}(x) = \begin{cases} 1 & \text{if } x > 0 \\ undefined & \text{if } x = 0 \\ \alpha & \text{if } x < 0 \end{cases}$$

2. **Show that a multilayer perceptron using only ReLU (or pReLU) constructs a continuous piecewise linear function.**

   By definiton, a `continous piecewise linear function` is a real-valued function defined on the real numbers, whose graph is composed of continous straight-line sections.

   $$\text{pReLU}(x) = \begin{cases} x & \text{if } x >= 0 \\ \alpha x & \text{if } x < 0 \end{cases}$$

3. **Show that $tanh(x) + 1 = 2sigmoid(2x)$.**

$$LHS = \tanh(x) + 1 = \frac{1 - \exp(-2x)}{1 + \exp(-2x)} + 1 = \frac{2}{1 + \exp(-2x)}.$$

$$RHS = 2\text{sigmoid}(2x) = \frac{2}{1 + \exp(-2x)}.$$

4. **Assume we have a multilayer perceptron without nonlinearities between the layers. In particular, assume that we have d input dimensions, d output dimensions and that one of the layers had only d/2 dimensions. Show that this network is less expressive (powerful) than a single layer perceptron.**

   A multilayer perceptron without nonlinearities is equal to one layer perceptron.

$$\hat{\mathbf{y}} = \mathbf{W_2}(\mathbf{W_1}X + b_1) + b_2 = \mathbf{W_2}\mathbf{W_1}X + (\mathbf{W_2}b_1 + b_2) := \mathbf{W_3}X + b_3$$

   Hence, if any of layer of d/2 dimension, then the rank of W_3 will be at most d/2, which can not express the final output of dimension d. However, a single layer percptron with sofmax regression will add nonlinearities to the model, which will learn and represent almost any arbitrary complex function which maps inputs to outputs.

5. **Assume that we have a nonlinearity that applies to one minibatch at a time. What kinds of problems do you expect this to cause?**

   Minibatch may not be as representative as whole batch. As a result, parameters learned from the minibatch may get weird gradients and get harder to converge.

4.2.7. Exercises¶

1. **Change the value of the hyper-parameter `num_hiddens` in order to see how this hyperparameter influences your results.**

   The upper bound on the number of hidden neurons that won't result in over-fitting is:

$$N_h = \frac{N_s}{(\alpha * (N_i + N_o))}$$

   where α = an arbitrary scaling factor usually 2-10; N_i = number of input neurons; N_o = number of output neurons; N_s = number of samples in training data set.

   Below the upper bound, the larger the num_hiddens, the better your results might be.

2. **Try adding a new hidden layer to see how it affects the results.**

   In general, adding a new hidden layer to a shallow networks should improve the accuracy. Since wide and shallow networks are very good at memorization, but not so good at generalization. Multiple layers are much better at generalizing because they learn all the intermediate features between the raw data and the high-level classification.

3. **How does changing the learning rate change the result.**

   If a learning rate is too high, it may overshoot the minimum and fail to converge in the end. If it is too low, then gradient descent can be slow.

4.3.2. Exercises¶

1. **Try adding a few more hidden layers to see how the result changes.**

2. **Try out different activation functions. Which ones work best?**

   Sigmoid and Tanh both suffers from vanishing gradient problems.

   ReLU rectifies the problem but it could result to "Dead Neuron" (since partial of its weights never get updated). Leaky ReLu to fix the problem of dying neurons. Also, ReLU can be only use within the hidden layer of NN.

   Softmax can be use in the output layers of classification model.

3. **Try out different initializations of the weights.**

   Zero initialization. All weights will be the same in the end, since the derivative with respect to loss function is the same.

   Random initialization. Initializing weights randomly, following normal distribution. May suffer from vanishing gradients and exploding gradients.

   Xavier initialization. The initializer fills the weights with random numbers in the range of [−c,c], where $c = \sqrt{\frac{3.}{0.5*(n_{in}+n_{out})}}$. n_{in} is the number of neurons feeding into weights, and n_{out} is the number of neurons the result is fed to.

4.4.6. Exercises

1. **Can you solve the polynomial regression problem exactly? Hint - use linear algebra.**

   Given the polynomial regression samples $\{(X_i, Y_i)\}_{i=1}^n$, we have $Y = \beta_0 + \beta_1 X + \beta_2 X^2 + \cdots + \beta_p X^p + \varepsilon = \mathbf{X}\boldsymbol{\beta} + \varepsilon$

   where

$$\mathbf{X} = \begin{pmatrix} 1 & X_1 & \cdots \\ X_1^p & & \\ \vdots & \vdots & \ddots \\ \vdots & & \\ 1 & X_n & \cdots \\ X_n^p & & \end{pmatrix}_{n\times(p+1)}$$

   Hence, $\hat{\boldsymbol{\beta}} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}\mathbf{y}$, where $\mathbf{y} = (Y_1, \ldots, Y_n)^T$.

   Note $(p+1) \times (p+1) = \text{rank}(\mathbf{X}^T\mathbf{X}) = \text{rank}(\mathbf{X}) = n$. Then, if $p = n-1$, $\mathbf{X}^T\mathbf{X}$ has dimension $n\times n$ and its rank is n, so no problem, is invertible. But if $p = n$, the dimension of $\mathbf{X}^T\mathbf{X}$ is $(n+1)\times(n+1)$ and the rank remains n, so in that case (and also if $p > n$) is not invertible (linear dependence arises).

2. **Model selection for polynomials**

   a. **Plot the training error vs. model complexity (degree of the polynomial). What do you observe?**

   b. **Plot the test error in this case.**

c. **Generate the same graph as a function of the amount of data?**

See details in *4.4.4. Polynomial Regression*.

3. **What happens if you drop the normalization of the polynomial features x_i by $1/i!$. Can you fix this in some other way?**

   There might be very large values of gradients and losses, due to very large values for exponents i.

4. **What degree of polynomial do you need to reduce the training error to 0?**

   As explained in Q1, if $p = n - 1$, $\mathbf{X}^T\mathbf{X}$ has dimension $n \times n$ and its rank is n. Then $\hat{\boldsymbol{\beta}} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}\mathbf{y}$, is the exact answer and hence the training error is 0.

5. **Can you ever expect to see 0 generalization error?**

   Yes. Sometimes if we accidentally have training set including all testing set's features and labels, (i.e. testing set items are all duplicated to training set). Then we may see a 0 generalization error.

4.5.6. Exercises

1. **Experiment with the value of `[U+1D706]` in the estimation problem in this page. Plot training and test accuracy as a function of `[U+1D706]` . What do you observe?**

```
In [ ]: num_epochs, lr, batch_size = 10, 0.1, 256
        loss = gloss.SoftmaxCrossEntropyLoss()
        train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
        drop_prob1, drop_prob2 = 0.2, 0.5


        def fit_and_plot_lambda(wd_list, num_epochs, lr=0.1, wd=5):
            '''
            wd_list : a list of number which represents weight_decay value
            '''
            train_ls, test_ls = [], []
            for wd in wd_list:
                net = nn.Sequential()
                net.add(nn.Dense(1))
                net.initialize(init.Normal(sigma=0.1))
                trainer_w = gluon.Trainer(net.collect_params('.*weight'), 'sgd',
                                          {'learning_rate': lr, 'wd': wd})
                trainer_b = gluon.Trainer(net.collect_params('.*bias'), 'sgd',
                                          {'learning_rate': lr})

                for _ in range(num_epochs):
                    for X, y in train_iter:
                        with autograd.record():
                            l = loss(net(X), y)
                        l.backward()
```

$$\|A\|_{\mathrm{F}} = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n}|a_{ij}|^2} = \sqrt{\operatorname{trace}(A^{\mathsf{T}}A)}$$

title

```
                # Call the step function on each of the two Trainer instances to
                # update the weight and bias separately
                trainer_w.step(batch_size)
                trainer_b.step(batch_size)
            train_ls.append(loss(net(train_features),train_labels).mean().asscalar())
            test_ls.append(loss(net(test_features),test_labels).mean().asscalar())
        d2l.semilogy(wd_list, train_ls, 'weight_decay', 'loss',
                     wd_list, test_ls, ['train', 'test'])

    fit_and_plot_lambda(wd_list=range(10), num_epochs=100)
```

2. **Use a validation set to find the optimal value of [U+1D706] . Is it really the optimal value? Does this matter?**

   Given different hyperparameters, network architecture and dataset, the optimal weight decay will vary. Hence there is no global optimal value.

3. **What would the update equations look like if instead of [U+2016]w[U+2016]^2 we used $\sum_i |w_i|$ as our penalty of choice (this is called ℓ1 regularization).**

   For L2, the loss function and corresponding stochastic gradient descent updates is :

$$l(\mathbf{w}, b) + \frac{\lambda}{2}\|\boldsymbol{w}\|^2$$

$$w \leftarrow \left(1 - \frac{\eta\lambda}{|\mathcal{B}|}\right)\mathbf{w} - \frac{\eta}{|\mathcal{B}|}\sum_{i\in\mathcal{B}}\mathbf{x}^{(i)}\left(\mathbf{w}^{\top}\mathbf{x}^{(i)} + b - y^{(i)}\right),$$

   For L1, the loss function and corresponding stochastic gradient descent updates is :

$$l(\mathbf{w}, b) + \frac{\lambda}{2}\sum_i |w_i|$$

$$w \leftarrow \left(1 - \frac{\eta\lambda}{2|\mathcal{B}||\mathbf{w}|}\right)\mathbf{w} - \frac{\eta}{|\mathcal{B}|}\sum_{i\in\mathcal{B}}\mathbf{x}^{(i)}\left(\mathbf{w}^{\top}\mathbf{x}^{(i)} + b - y^{(i)}\right),$$

4. **We know that [U+2016]w[U+2016]$^2 = \mathbf{w}^{\top}\mathbf{w}$. Can you find a similar equation for matrices (mathematicians call this the Frobenius norm)?**

5. **Review the relationship between training error and generalization error. In addition to weight decay, increased training, and the use of a model of suitable complexity, what other ways can you think of to deal with overfitting?**

   Add more training examples; Increase dropout; Decrease features, etc.

6. **In Bayesian statistics we use the product of prior and likelihood to arrive at a posterior via $p(w|x) \propto p(x|w)p(w)$. How can you identify $p(w)$ with regularization?**

4.6.7. Exercises

1. **Try out what happens if you change the dropout probabilities for layers 1 and 2. In particular, what happens if you switch the ones for both layers?**

In [ ]:
```
## TODO: Run on GPU

    ## network 1
    net467_1 = nn.Sequential()
    net467_1.add(nn.Dense(256, activation="relu"),
            nn.Dropout(drop_prob1),
            nn.Dense(256, activation="relu"),
            nn.Dropout(drop_prob2),
            nn.Dense(10))
    net467_1.initialize(init.Normal(sigma=0.01))
    trainer = gluon.Trainer(net467_1.collect_params(), 'sgd', {'learning_rate': lr})
    d2l.train_ch3(net467_1, train_iter, test_iter, loss, num_epochs, batch_size, None,
                  None, trainer)
```

In [ ]:
```
## TODO: Run on GPU

    ## network 2, switch dropout rate
    net467_2 = nn.Sequential()
    net467_2.add(nn.Dense(256, activation="relu"),
            nn.Dropout(drop_prob2),
            nn.Dense(256, activation="relu"),
            nn.Dropout(drop_prob1),
            nn.Dense(10))
    net467_2.initialize(init.Normal(sigma=0.01))
    trainer = gluon.Trainer(net467_2.collect_params(), 'sgd', {'learning_rate': lr})
    d2l.train_ch3(net467_2, train_iter, test_iter, loss, num_epochs, batch_size, None,
                  None, trainer)
```

2. **Increase the number of epochs and compare the results obtained when using dropout with those when not using it.**

In [ ]:
```
## TODO: Run on GPU

    ## network 3
    num_epochs = 50
    net467_3 = nn.Sequential()
    net467_3.add(nn.Dense(256, activation="relu"),
            nn.Dropout(drop_prob2),
            nn.Dense(256, activation="relu"),
            nn.Dropout(drop_prob1),
            nn.Dense(10))
    net467_3.initialize(init.Normal(sigma=0.01))
    trainer = gluon.Trainer(net467_3.collect_params(), 'sgd', {'learning_rate': lr})
    d2l.train_ch3(net467_3, train_iter, test_iter, loss, num_epochs, batch_size, None,
                  None, trainer)
```

3. **Compute the variance of the the activation random variables after applying dropout.**

   Dropout replaces an activation h with a random variable h' with expected value h and with variance given by the dropout probability p .

$$h' = \begin{cases} 0 & \text{with probability } p \\ \frac{h}{1-p} & \text{otherwise} \end{cases}$$

4. **Why should you typically not using dropout?**

   Dropout can help with regularization, but at a risk of lossing improtant information. Especially applying Dropout in the first layer will lead to significant inforamtion loss.

5. **If changes are made to the model to make it more complex, such as adding hidden layer units, will the effect of using dropout to cope with overfitting be more obvious?**

   For an overfitted model, adding a hidden layer with dropout may not help. Especially in the situation that the effective neurons in this layer is larger than the number of neurons in the later layers, since this is equal to adding an extra hidden layer.

6. **Using the model in this section as an example, compare the effects of using dropout and weight decay. What if dropout and weight decay are used at the same time?**

In [ ]:
```
## TODO: Run on GPU
```

```
## random simulate dataset
n_train, n_test, num_inputs = 20, 100, 200
true_w, true_b = nd.ones((num_inputs, 1)) * 0.01, 0.05

features = nd.random.normal(shape=(n_train + n_test, num_inputs))
labels = nd.dot(features, true_w) + true_b
labels += nd.random.normal(scale=0.01, shape=labels.shape)
train_features, test_features = features[:n_train, :], features[n_train:, :]
train_labels, test_labels = labels[:n_train], labels[n_train:]
train_iter = gdata.DataLoader(gdata.ArrayDataset(train_features, train_labels), batch_si
```

```
def fit_and_plot_gluon_467_6(dropout, wd, num_epochs=50, lr=0.01, batch_size=256):
    net = nn.Sequential()
    net.add(nn.Dense(256),
            nn.Dropout(drop_prob),
            nn.Dense(10))
    net.initialize(init.Normal(sigma=0.01))
    loss = gloss.L2Loss()

    trainer_w = gluon.Trainer(net.collect_params('.*weight'), 'sgd',
                              {'learning_rate': lr, 'wd': wd})
    # The bias parameter has not decayed. Bias names generally end with "bias"
```

```
    trainer_b = gluon.Trainer(net.collect_params('.*bias'), 'sgd',
                              {'learning_rate': lr})
    train_ls, test_ls = [], []
    for _ in range(num_epochs):
        for X, y in train_iter:
            with autograd.record():
                l = loss(net(X), y)
            l.backward()
            # Call the step function on each of the two Trainer instances to
            # update the weight and bias separately
            trainer_w.step(batch_size)
            trainer_b.step(batch_size)
        train_ls.append(loss(net(train_features),
                             train_labels).mean().asscalar())
        test_ls.append(loss(net(test_features),
                            test_labels).mean().asscalar())
    d2l.semilogy(range(1, num_epochs + 1), train_ls, 'epochs', 'loss',
                 range(1, num_epochs + 1), test_ls, ['train', 'test'])
#     print('L2 norm of w:', net[0].weight.data().norm().asscalar())

fit_and_plot_gluon_467_6(dropout=0.5, wd=0)
fit_and_plot_gluon_467_6(dropout=0.5, wd=3)
fit_and_plot_gluon_467_6(dropout=0, wd=3)
```

7. **What happens if we apply dropout to the individual weights of the weight matrix rather than the activations?**

   The regularization effect will be the same. If we turn partial of the weights to be zero, these neurons won't learn any signal from the inputs, which will have the similar functionality as dropout on activation.

8. **Replace the dropout activation with a random variable that takes on values of $[0, \gamma/2, \gamma]$. Can you design something that works better than the binary dropout function? Why might you want to use it? Why not?**

   Define the following dropout activation function:

$$h' = \begin{cases} 0 & \text{with probability } p_1 \\ \gamma/2 * h & \text{with probability } (1-p_1)p_2 \\ \gamma * h & \text{with probability } (1-p_1)(1-p_2) \end{cases}$$

   The has the expectation remained unchanged, we need to have

$$0 + (\gamma/2)h(1-p_1)p_2 + \gamma h(1-p_1)(1-p_2) = h$$

   Thus,

$$\gamma = \frac{2}{(2-p_2)(1-p_1)}$$

   For example, if we let $p_1 = 0.2$, $p_2 = 0.75$, then $\gamma = 2$ by the above formula,

$$h' = \begin{cases} 0 & \text{with probability 0.2} \\ h & \text{with probability 0.6} \\ 2h & \text{with probability 0.2} \end{cases}$$

4.7.6. Exercises

1. **Assume that the inputs X are matrices. What is the dimensionality of the gradients?**

   Notice the dimensionality of each layer's graditents is equal the dimensionality of each layer's weighs. i.e.

   $$dim(\frac{\partial J}{\partial \mathbf{W}^{(1)}}) = dim(\mathbf{W}^{(1)})$$

   If the inputs $X \in \mathbb{R}^{d \times c}$ are matrices at each row, the weight matrix $\mathbf{W}^{(1)}$ would have a dimension equal to $h \times (d \times c)$, where h is the hidden layer dimension.

2. **Add a bias to the hidden layer of the model described in this chapter.**

   a. **Draw the corresponding compute graph.**

   b. **Derive the forward and backward propagation equations.**

   c. Given a hidden layer with a bias:

   $$\mathbf{z} = \mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}\mathbf{h} = \phi(\mathbf{z})\mathbf{o} = \mathbf{W}^{(2)}\mathbf{h} + \mathbf{b}^{(2)}L = l(\mathbf{o}, y)s = \frac{\lambda}{2}\left(\|\mathbf{W}^{(1)}\|_F^2 + \|\mathbf{W}^{(2)}\|_F^2\right)J = L + s$$

   hence the backward propagation will be

   $$\frac{\partial J}{\partial \mathbf{W}^{(2)}} = \text{prod}\left(\frac{\partial J}{\partial \mathbf{o}}, \frac{\partial \mathbf{o}}{\partial \mathbf{W}^{(2)}}\right) + \text{prod}\left(\frac{\partial J}{\partial s}, \frac{\partial s}{\partial \mathbf{W}^{(2)}}\right) = \frac{\partial J}{\partial \mathbf{o}}\mathbf{h}^\top + \lambda \mathbf{W}^{(2)}$$

   $$\frac{\partial J}{\partial \mathbf{b}^{(2)}} = \text{prod}\left(\frac{\partial J}{\partial \mathbf{o}}, \frac{\partial \mathbf{o}}{\partial \mathbf{b}^{(2)}}\right) + \text{prod}\left(\frac{\partial J}{\partial s}, \frac{\partial s}{\partial \mathbf{b}^{(2)}}\right) = \frac{\partial J}{\partial \mathbf{o}} \times 1 + 0 = \frac{\partial J}{\partial \mathbf{o}}$$

   $$\frac{\partial J}{\partial \mathbf{W}^{(1)}} = \frac{\partial J}{\partial \mathbf{o}}\frac{\partial \mathbf{o}}{\partial \mathbf{h}}\frac{\partial \mathbf{h}}{\partial \mathbf{z}}\frac{\partial \mathbf{z}}{\partial \mathbf{W}^{(1)}} + \text{prod}\left(\frac{\partial J}{\partial s}, \frac{\partial s}{\partial \mathbf{W}^{(1)}}\right) = \frac{\partial J}{\partial \mathbf{o}}{\mathbf{W}^{(2)}}^\top \odot \phi'(\mathbf{z})\,\mathbf{x}^\top + \lambda \mathbf{W}^{(1)}$$

   $$\frac{\partial J}{\partial \mathbf{b}^{(1)}} = \frac{\partial J}{\partial \mathbf{o}}\frac{\partial \mathbf{o}}{\partial \mathbf{h}}\frac{\partial \mathbf{h}}{\partial \mathbf{z}}\frac{\partial \mathbf{z}}{\partial \mathbf{b}^{(1)}} + \text{prod}\left(\frac{\partial J}{\partial s}, \frac{\partial s}{\partial \mathbf{b}^{(1)}}\right) = \frac{\partial J}{\partial \mathbf{o}}{\mathbf{W}^{(2)}}^\top \odot \phi'(\mathbf{z})$$

3. **Compute the memory footprint for training and inference in model described in the current chapter.**

   Training: need memory for

   $$\frac{\partial J}{\partial \mathbf{o}}, \mathbf{W}^{(2)}, \mathbf{W}^{(1)}, \phi'(\mathbf{z}), \mathbf{x},$$

   Inference: only need memory for

   $$\mathbf{W}^{(1)}, \mathbf{W}^{(2)}, \mathbf{b}^{(1)}, \mathbf{b}^{(2)}$$

4. **Assume that you want to compute second derivatives. What happens to the compute graph? Is this a good idea?**

5. **Assume that the compute graph is too large for your GPU.**

   a.**Can you partition it over more than one GPU?**

   ```
   By default, MXNet uses data parallelism to partition the workload over multiple devices. A
   ```

   b.**What are the advantages and disadvantages over training on a smaller minibatch?**

   Advantages: 1. More robust convergence, avoiding local minima (as its model update frequency is higher than batch gradient descent.) 2. Computationally efficient process than stochastic gradient descent. 3. Memory efficient than batch gradient descent.

   Disadvantages: 1. Tune minibatch hyperparameter

4.8.4. Exercises

1. **Can you design other cases of n,fjbhbf?**

2. **Can we initialize all weight parameters in linear regression or in softmax regression to the same value?**

   When all weights initialized to the same value, the weights will update with exact same gradient all the time. Only when initializing the model to small random values breaks the symmetry and allows different weights to learn independently of each other.

3. **Look up analytic bounds on the eigenvalues of the product of two matrices. What does this tell you about ensuring that gradients are well conditioned?**

   To compute the overall error gradient,

$$\begin{aligned}\frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_k} &= \frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_{t-1}} \frac{\partial \mathbf{h}_{t-1}}{\partial \mathbf{h}_{t-2}} \cdots \frac{\partial \mathbf{h}_{k+1}}{\partial \mathbf{h}_k} \\ &= \prod_{i=k+1}^{t} \frac{\partial \mathbf{h}_i}{\partial \mathbf{h}_{i-1}} \\ &= \prod_{i=k+1}^{t} \mathbf{W}^\top \operatorname{diag}\left[f'\left(\mathbf{h}_{i-1}\right)\right]\end{aligned}$$

At each time step t,

$$\left\| \frac{\partial \mathbf{h}_i}{\partial \mathbf{h}_{i-1}} \right\| \leq \left\| \mathbf{W}^\top \right\| \left\| \operatorname{diag}\left[f'\left(\mathbf{h}_{i-1}\right)\right] \right\| \leq \gamma_{\mathbf{W}} \gamma_{\mathbf{h}}$$

where define $\gamma_{\mathbf{W}}$ as the largest eigenvalue associated with [U+2016]W^T[U+2016], as its upper bound, and $\gamma_{\mathbf{h}}$ as the largest eigenvalue associated with $\|\operatorname{diag}\left[f'\left(\mathbf{h}_{i-1}\right)\right]\|$. Notice that for *tanh* we have $\gamma_{\mathbf{h}} = 1$ while for *sigmoid* we have $\gamma_{\mathbf{h}} = 1/4$. Hence, after t steps,

$$\left\| \frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_k} \right\| = \left\| \prod_{i=k+1}^{t} \frac{\partial \mathbf{h}_i}{\partial \mathbf{h}_{i-1}} \right\| \leq \left(\gamma_{\mathbf{W}} \gamma_{\mathbf{h}}\right)^{t-k}$$

As a result, $\gamma_{\mathbf{W}} \gamma_{\mathbf{h}} < 1$ the gradient tends to vanish while for $\gamma_{\mathbf{W}} \gamma_{\mathbf{h}} > 1$ the gradient tends to explode.

4. **If we know that some terms diverge, can we fix this after the fact? Look at the paper on LARS by You, Gitman and Ginsburg, 2017 for inspiration.**

4.9.5. Exercises

(a) **What could happen when we change the behavior of a search engine? What might the users do? What about the advertisers?**

After the user input the query and get result sites from the search engine, the following scenario may happen:

i. The user click the site but quickly return - This may indicate a negative relevancy signal between search query and results.

ii. The user click the site and stay for a while - This may indicate a positive relevancy signal between search query and results.

iii. The user click the site and stay for either long or short, but return to refine the query in the search box - This may indicate the user themselves may not have been as specific as they needed to be.

The advertisers can active measure the user behavior through CTR(Click Through Rate)/CTP(Click Through Probobility), and users' actions after clicking. Based on the above behavior, advertisers can active adjust their advertising strategy.

(b) **Implement a covariate shift detector. Hint - build a classifier.**

(c) **Implement a covariate shift corrector.**

(d) **What could go wrong if training and test set are very different? What would happen to the sample weights?**

The model may has a low training loss and validation loss, but a high testing loss. And no matter how we ture the model, the accuracy may still pretty low.