Assignment 06

20142921 SengHyun Lee

2019.11.06

Binary classification based on 3 layers neural network

import library & GPU Setting

In [1]:

```
import matplotlib.pyplot as plt
import numpy as np
import math

import torch
from torch.utils.data import Dataset, DataLoader
import torchvision.transforms as transforms
import torchvision
```

In [2]:

```
# global settings
# torch.set_default_dtype(torch.float64)
torch.set_default_tensor_type('torch.cuda.DoubleTensor')
torch.set_printoptions(precision=16)
torch.cuda.set_device(0)

# setting check
print("current device : %s" % (torch.cuda.current_device()))
print("device count : %s" % (torch.cuda.device_count()))
print("device name : %s" % (torch.cuda.get_device_name(0)))
print("CUDA available? : %s" % (torch.cuda.is_available()))
```

```
current device : 0
device count : 1
device name : GeForce RTX 2060
CUDA available? : True
```

In [3]:

```
def output_plot(g1, g2, title, color, label, legend):
    plt.title(title)
    plt.plot(np.arange(1, len(g1) + 1), g1, color=color[0], alpha=0.5, label=label[0])
    plt.plot(np.arange(1, len(g2) + 1), g2, color=color[1], alpha=0.5, label=label[1])
    plt.legend(loc=legend)
    plt.show()
```

```python
def output_frame_plot(tloss, vloss, tacc, vacc, title):
    print(" << %s >>" % title)
    print("-----------------------------------------------------")
    print("                |   %10s    |    %10s     |" % ('loss', 'accuracy'))
    print("-----------------------------------------------------")
    print("training        |   %.10f   |    %.10f    |" % (tloss, tacc))
    print("-----------------------------------------------------")
    print("validation      |   %.10f   |    %.10f    |" % (vloss, vacc))
    print("-----------------------------------------------------")
```

Declare the constants

In [4]:

```python
IMAGE_WIDTH = 100
IMAGE_HEIGHT = 100
IMAGE_CHANNEL = 1
DIMENSION = IMAGE_CHANNEL * IMAGE_HEIGHT * IMAGE_WIDTH
```

Load train & validation datasets (preprocess)

- batch size = 3
- number of epoch = 1

In [5]:

```python
def pre_process(batch_size=3, num_workers=1):
    transform = transforms.Compose([  # transforms.Resize((256,256)),
        transforms.Grayscale(),
        # the code transforms.Graysclae() is for changing the size [3,100,100] to [1, 100,␣
↪100] (notice : [channel, height, width] )
        transforms.ToTensor(), ])

    # train_data_path = 'relative path of training data set'
    # change the valuse of batch_size, num_workers for your program
    # if shuffle=True, the data reshuffled at every epoch
    train_data_path = './horse-or-human/train'
    trainset = torchvision.datasets.ImageFolder(root=train_data_path, transform=transform)
    trainloader = torch.utils.data.DataLoader(
        dataset=trainset,
        batch_size=batch_size,
        shuffle=False,
    )

    validation_data_path = './horse-or-human/validation'
    valset = torchvision.datasets.ImageFolder(root=validation_data_path,␣
↪transform=transform)
    valloader = torch.utils.data.DataLoader(
        dataset=valset,
        batch_size=batch_size,
        shuffle=False,
```

```python
    )

    train_data = torch.empty((DIMENSION, 0), dtype=torch.double)
    validation_data = torch.empty((DIMENSION, 0), dtype=torch.double)

    train_label = torch.empty((1, 0))
    validation_label = torch.empty((1, 0))

    for i, (inputs, labels) in enumerate(trainloader):
        # inputs is the image
        # labels is the class of the image

        # if you don't change the image size, it will be [batch_size, 1, 100, 100]
        # [batch_size, 1, height, width] => [ width * height * channel, batch_size ]
        # x = inputs.transpose((2, 3, 0, 1)).reshape((DIMENSION, len(labels)))
        x = np.array(inputs).transpose((2, 3, 0, 1)).reshape((DIMENSION, len(labels)))
        x = torch.from_numpy(x.astype(np.double))
        y = labels.reshape((1, len(labels))).type(torch.double)

        x = x.to(0)
        y = y.to(0)

        train_data = torch.cat((train_data, x), dim=1)
        train_label = torch.cat((train_label, y), dim=1)

    # load validation images of the batch size for every iteration
    for i, data in enumerate(valloader):
        # inputs is the image
        # labels is the class of the image
        inputs, labels = data

        # [batch_size, 1, height, width] => [ width * height * channel, batch_size ]
        x = np.array(inputs).transpose((2, 3, 0, 1)).reshape((DIMENSION, len(labels)))
        x = torch.from_numpy(x.astype(np.double))
        y = labels.reshape((1, len(labels))).type(torch.double)

        x = x.to(0)
        y = y.to(0)

        validation_data = torch.cat((validation_data, x), dim=1)
        validation_label = torch.cat((validation_label, y), dim=1)

    return train_data, validation_data, train_label, validation_label
```

Implements of 3 layers neural network

Architecture

First layer

- $Z^{[1]} = W^{[1]}X + b^{[1]}$: X denotes the input data
- $A^{[1]} = g^{[1]}(Z^{[1]})$: $g^{[1]}$ is the activation function at the first layer

Second layer

- $Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]}$
- $A^{[2]} = g^{[2]}(Z^{[2]})$: $g^{[2]}$ is the activation function at the second layer

Third layer

- $Z^{[3]} = W^{[3]}A^{[2]} + b^{[3]}$
- $A^{[3]} = g^{[3]}(Z^{[3]})$: $g^{[3]}$ is the activation function at the third (output) layer

Activation function

- $g^{[1]}, g^{[2]}$ and $g^{[3]} = \frac{1}{1+\exp^{-z}}$ (sigmoid)

Loss function

$\mathcal{L} = \frac{1}{n}\sum_{i=1}^{n} f_i + \frac{\lambda}{2}\left(\|W^{[1]}\|_F^2 + \|W^{[2]}\|_F^2 + \|W^{[3]}\|_F^2\right)$

- $f_i = -y_i \log \hat{y}_i - (1 - y_i)\log(1 - \hat{y}_i)$ (Cross Entropy)
- $\|W\|_F = \left(\sum_i \sum_j w_{ij}^2\right)^{\frac{1}{2}}$ (Frobenius Norm)

Parameters

- learning rate $= 0.015$
- tolerance $= 10^{-6}$
- initialization : $Var(w_i) = \frac{1}{n_{in}}$
- regularization : L_2^2 Regularization

Implemenations

In [6]:

```
def binary_classify(train_data, validation_data,
                    train_label, validation_label,
                    gn_act, gn_d_act, init, learning_rate=0.0002, regular_weight=0.01):

    num_of_layers = 3

    n = train_label.shape[1]

    n1, n2 = 100, 150
    learning_rate = learning_rate
    regular_weight = regular_weight

    epsilon = 10e-6

    # INITIALIZE u v z
    u, v, w = init(DIMENSION, n1, n2)

    # INITIALIZE bias
    b1 = torch.zeros((n1, 1))
    b2 = torch.zeros((n2, 1))
    b3 = torch.zeros((1, 1))
```

```
    train_losses = []
    test_losses = []
    train_accuracies = []
    test_accuracies = []

    def _nan_to_num(tensor):
        return tensor
        # return torch.from_numpy(np.nan_to_num(tensor.cpu().numpy()))

    # def safe_ln(x, minval=10e-20):
    #     return np.log(x.clip(min=minval))

    def sq_frobenius(mat):
        return (torch.sum(mat ** 2)).item()

    def cross_entropy(prob, ans):
        return -((_nan_to_num(ans * torch.log(prob))) +
                 (_nan_to_num((1 - ans) * torch.log(1-prob))))

    def loss(prob, ans):
        a = torch.sum(_nan_to_num(cross_entropy(prob, ans))).item() / ans.shape[1]
        b = (regular_weight / (2*ans.shape[1])) * (sq_frobenius(u) + sq_frobenius(v) +
↪sq_frobenius(w))
        return a + b

    def accuracy(prob, ans):
        arr = (prob > 0.5).long()
        arr = arr - ans.long()
        arr = (arr == 0).long()
        return torch.sum(arr).item() / ans.shape[1]

    def iterate():
        p_train_loss = 0
        nonlocal u, v, w, b1, b2, b3
        nonlocal train_losses, test_losses, train_accuracies, test_accuracies

        while True:

            # forward propagation #
            act = gn_act()
            next(act)
            z1 = torch.mm(u.T, train_data) + b1
            a1 = act.send(z1)

            z2 = torch.mm(v.T, a1) + b2
            a2 = act.send(z2)

            z3 = torch.mm(w.T, a2) + b3
            a3 = act.send(z3)

            act = gn_act()
```

```
            next(act)
            vz = torch.mm(u.T, validation_data) + b1
            vz = torch.mm(v.T, act.send(vz)) + b2
            vz = torch.mm(w.T, act.send(vz)) + b3
            ####

            # back propagation #
            d_act = gn_d_act()
            next(d_act)
            cw = (a3 - train_label)
            dw = torch.mm(cw, a2.T) / z3.shape[1]

            cv = torch.mm(w, cw) * d_act.send(z2)
            dv = torch.mm(cv, a1.T) / z3.shape[1]

            cu = torch.mm(v, cv) * d_act.send(z1)
            du = torch.mm(cu, train_data.T) / z3.shape[1]

            b3 = b3 - (learning_rate * (torch.sum(cw, dim=1, keepdim=True) / z3.shape[1]))
            b2 = b2 - (learning_rate * (torch.sum(cv, dim=1, keepdim=True) / z3.shape[1]))
            b1 = b1 - (learning_rate * (torch.sum(cu, dim=1, keepdim=True) / z3.shape[1]))

            # gradient descent #
            w = w - (learning_rate * dw).T - (learning_rate * (regular_weight * w)/n)
            v = v - (learning_rate * dv).T - (learning_rate * (regular_weight * v)/n)
            u = u - (learning_rate * du).T - (learning_rate * (regular_weight * u)/n)
            ####

            # get losses
            t_hat, v_hat = a3, act.send(vz)

            n_train_loss = loss(t_hat, train_label)
            n_test_loss = loss(v_hat, validation_label)

            # get accuracies
            n_train_acc = accuracy(t_hat, train_label)
            n_test_acc = accuracy(v_hat, validation_label)

            train_losses.append(n_train_loss)
            test_losses.append(n_test_loss)
            train_accuracies.append(n_train_acc)
            test_accuracies.append(n_test_acc)

            if abs(p_train_loss - n_train_loss) < epsilon:
                break
            else:
#                 print('tl: %s, vl: %s, ta: %s, va: %s' % (n_train_loss, n_test_loss, n_train_acc, n_test_acc))
                p_train_loss = n_train_loss
                continue
```

```
    iterate()

    return train_losses, test_losses, train_accuracies, test_accuracies
```

In [7]:

```
def learn(title, learning_rate=0.015, regular_weight=49.195):

    t_data, v_data, t_label, v_label = pre_process(batch_size=3)
    train_loss, test_loss, train_acc, test_acc = [], [], [], []

    # initialization functions
    def xaiver_initialize(n0, n1, n2):
        u = torch.randn((n0, n1)) * math.sqrt(1 / n0)
        v = torch.randn((n1, n2)) * math.sqrt(1 / n1)
        w = torch.randn((n2, 1)) * math.sqrt(1 / n2)
        return u, v, w

    def gen_xaiver_initialize(n0, n1, n2):
        u = torch.randn((n0, n1)) * math.sqrt(1 / (n0 + n1))
        v = torch.randn((n1, n2)) * math.sqrt(1 / (n1 + n2))
        w = torch.randn((n2, 1)) * math.sqrt(1 / (n2 + 1))
        return u, v, w

    # activation functions
    def sigmoid(z):
        return 1 / (1 + torch.exp(-z))

    def d_sigmoid(z):
        return sigmoid(z) * (1 - sigmoid(z))

    # sigmoid
    def case1(learning_rate, regular_weight):
        def act():
            z = yield
            z = yield sigmoid(z)
            z = yield sigmoid(z)
            z = yield sigmoid(z)

        def d_act():
            z = yield
            z = yield d_sigmoid(z)
            z = yield d_sigmoid(z)

        classify(
            gn=act,
            dgn=d_act,
            learning_rate=learning_rate,
            regular_weight=regular_weight,
            init=xaiver_initialize
```

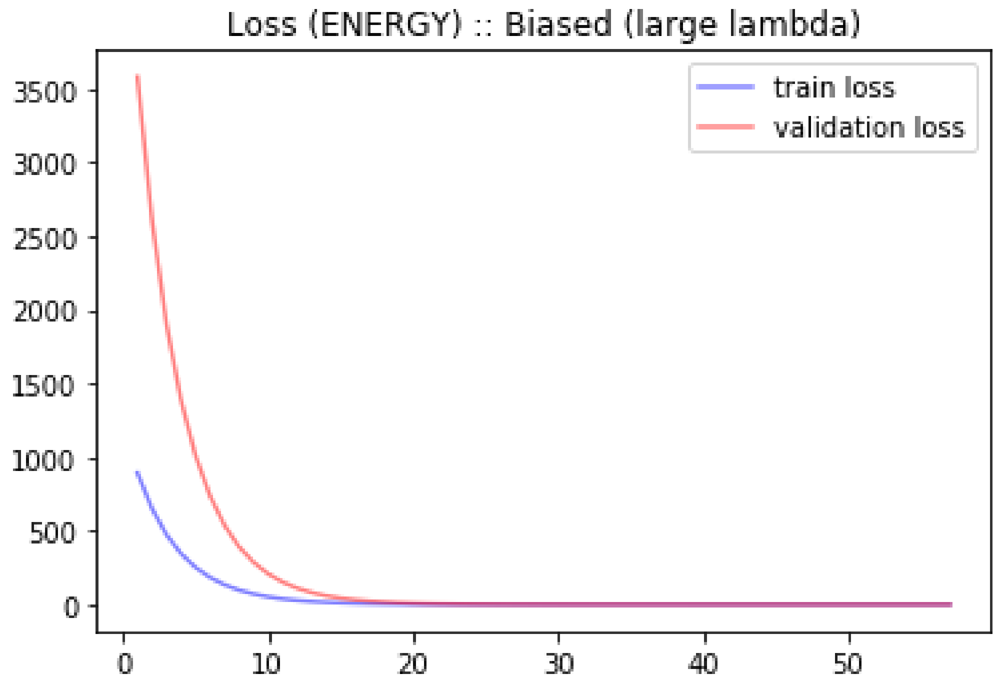
```
        )
        plot()

    def classify(gn, dgn, learning_rate, regular_weight, init):
        nonlocal train_loss, test_loss, train_acc, test_acc

        train_loss, test_loss, train_acc, test_acc = binary_classify(
            t_data, v_data,
            t_label, v_label,
            gn, dgn,
            learning_rate=learning_rate,
            regular_weight=regular_weight,
            init=init
        )

    def plot():
        output_plot(train_loss, test_loss,
                    title="Loss (ENERGY) :: " + title, color=('blue', 'red'),
                    label=('train loss', 'validation loss'), legend='upper right')

        output_plot(train_acc, test_acc,
                    title="Accuracy :: " + title, color=('blue', 'red'),
                    label=('train accuracy', 'validation accuracy'), legend='lower right')

        output_frame_plot(
            train_loss[-1], test_loss[-1],
            train_acc[-1], test_acc[-1],
            title=title
        )

    case1(learning_rate=learning_rate, regular_weight=regular_weight)
```

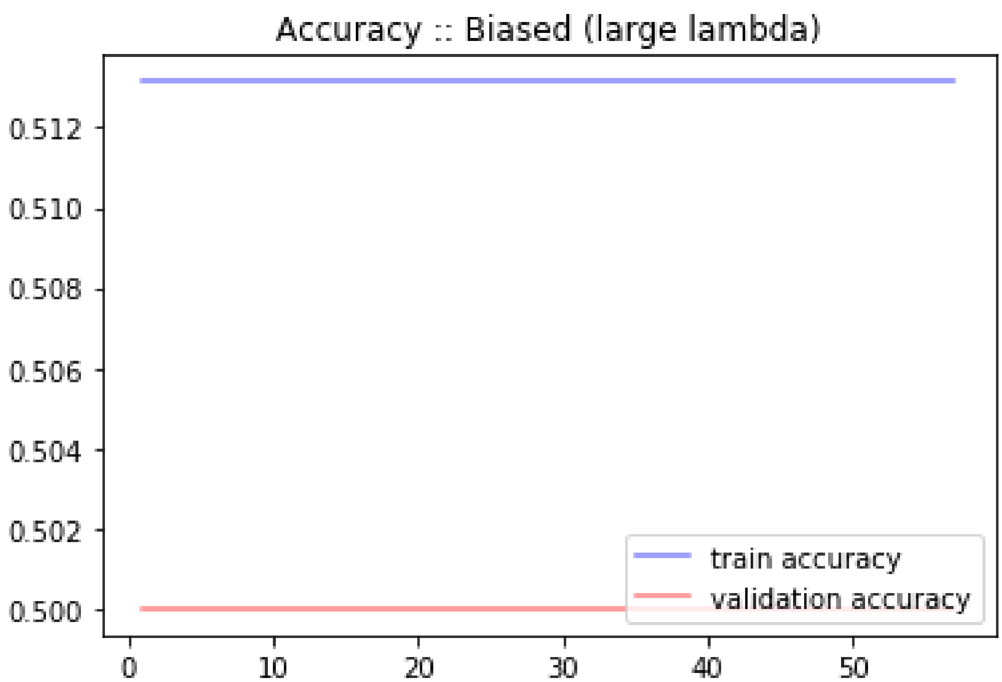
Result

1. Bias ($\lambda = 10000$)

In [13]:

```
learn(title="Biased (large lambda)", learning_rate=0.015, regular_weight=10000)
```





<< Biased (large lambda) >>

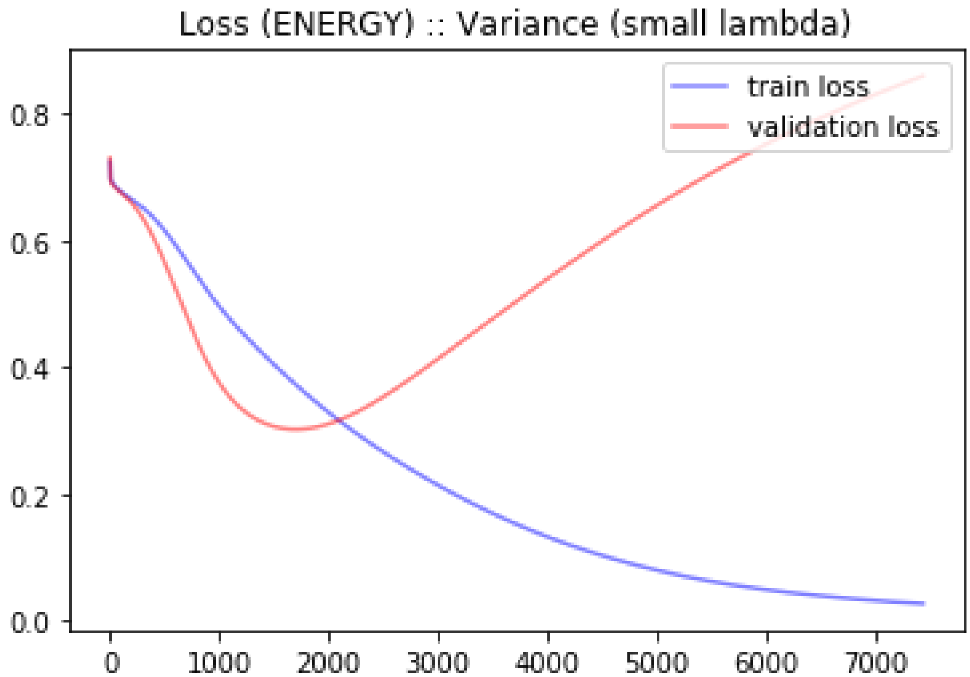
	loss	accuracy
training	0.6929842893	0.5131450828
validation	0.6936387257	0.5000000000

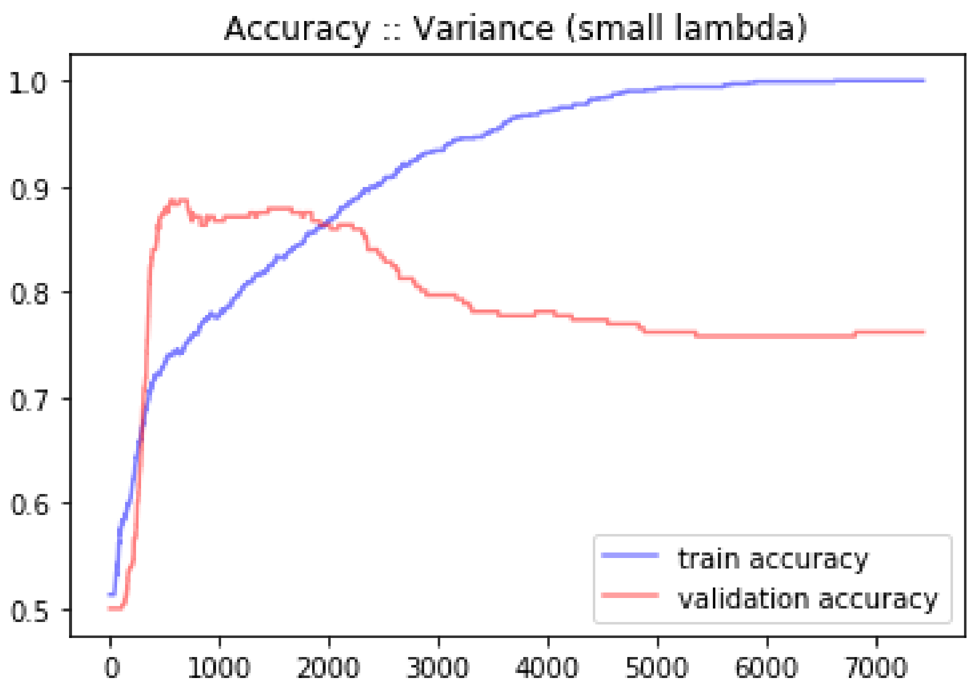
---

2. Variance $(\lambda = 0)$

In [11]:

```
learn(title="Varianced (small lambda)", learning_rate=0.015, regular_weight=0)
```



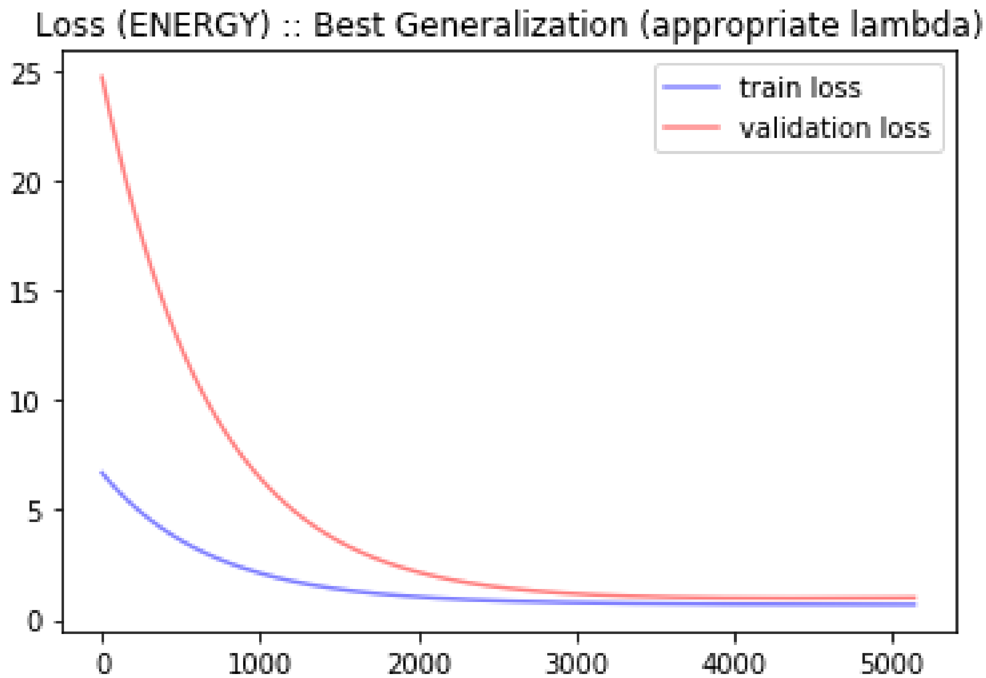


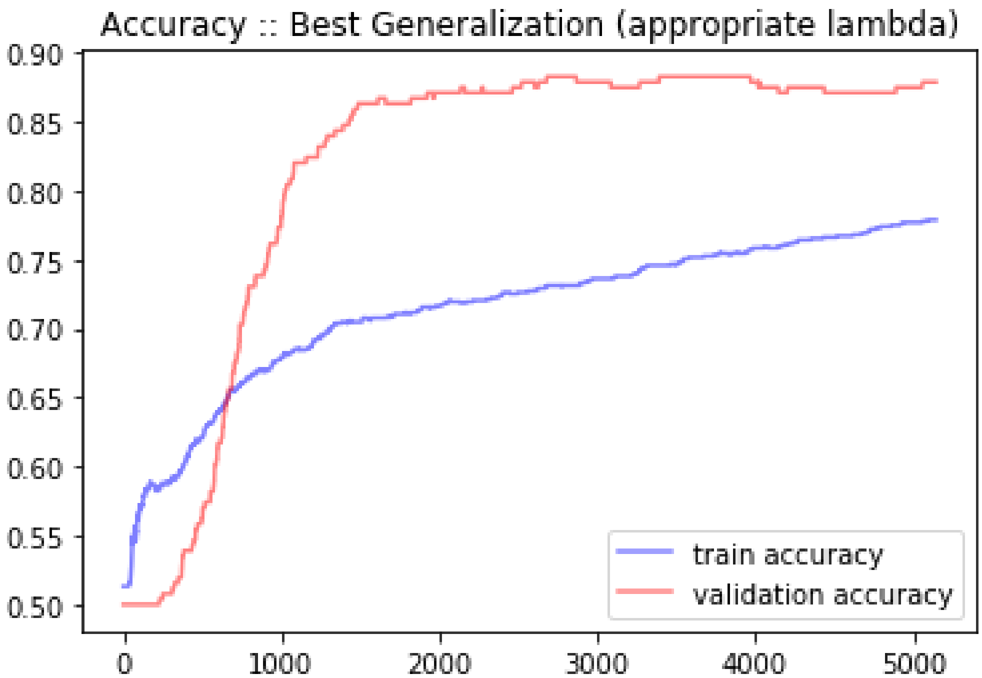
```
 << Variance (small lambda) >>
------------------------------------------------------
                |          loss     |       accuracy      |
------------------------------------------------------
training        |    0.0269909477    |     1.0000000000     |
------------------------------------------------------
validation      |    0.8595741527    |     0.7617187500     |
------------------------------------------------------
```

3. Best Generalization ($\lambda = 49.195$)

In [12]:

```
learn(title="Best Generalization (appropriate lambda)", learning_rate=0.015,␣
 ↪regular_weight=49.195)
```





<< Best Generalization (appropriate lambda) >>

	loss	accuracy
training	0.6791322161	0.7789678676
validation	0.9949059266	0.8789062500