Deep Learning HW2 due 12/5/2019

0712238 林彥彤 Maxwill Lin

Answer the following questions for MNIST and CIFAR10 datasets.

1. Design of architecture
2. Learning curve, accuracy, distribution of weights and biases
3. Examples and show hidden layers
4. add L2 regularization to see effect on 2.
5. describe the preprocessing of CIFAR10

And add some discussion.

1. Architecture design, Experiments
    a. Experiments smaller filter size on MNIST
    b. selu on CIFAR10
    c. avg pooling on CIFAR10
2. MNIST vs CIFAR10 accuracy curve
3. Data augmentation by transformation
4. Negative weights in CIFAR10
5. Inference on why regularized model performed poorly on CIFAR10
6. Reason to rewrite MNIST L2 to PyTorch

Implementations:
Tensorflow 1.0(MNIST no L2), PyTorch(MNIST + L2, others)
Save/Load models and records
Visualization of hidden layers + records
Mismatch finding/class finding functions

Task 1: MNIST dataset

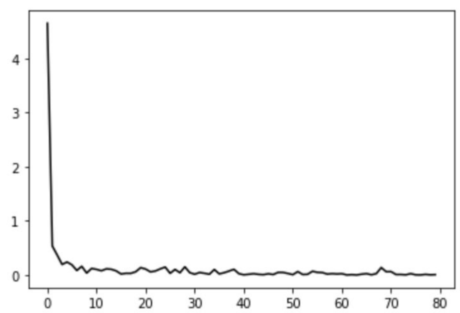
1.Architecture(PyTorch code, follows the structure of Tensorflow1)

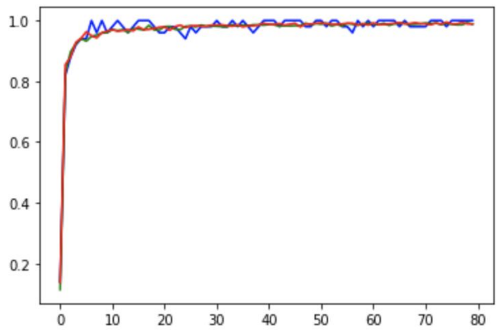
```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        #in, out, kernel, padding(both sides=>*2)
        self.conv1 = nn.Conv2d(1, 32, 5, padding=2) # 1 28 28 -> 32 28 28 -> 32 14 14
        self.conv2 = nn.Conv2d(32, 64, 5, padding=2)# 32 14 14 -> 64 14 14 -> 64 7 7
        self.pool = nn.MaxPool2d(2, 2, padding=0)
        self.fc1 = nn.Linear(64*7*7, 1024)          #64*7*7 -> 1024
        self.fc2 = nn.Linear(1024, 10)              #1024   -> 10
        self.dpout = nn.Dropout(p=0.5)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = self.pool(x)
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 64*7*7)
        x = F.relu(self.fc1(x))
        x = self.dpout(x)
        x = self.fc2(x)
        #print(x.shape) is important
        return x
```

```
Tainable Valuables:
conv1.weight     800
conv1.bias       32
conv2.weight     51200
conv2.bias       64
fc1.weight       3211264
fc1.bias         1024
fc2.weight       10240
fc2.bias         10

Total    3274634
```

(NO L2 regularized, on Tensorflow 1)

2.Learning curve, accuracy curve

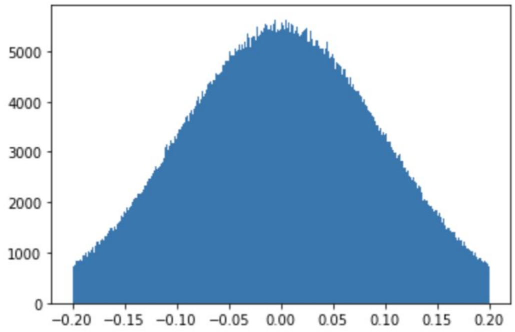


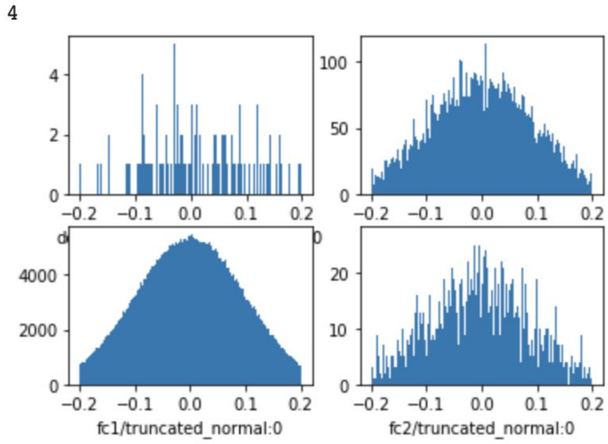


(accuracy curves: red = test, blue = train, during training test on random 200 samples)

total = 8000(iteration)*500(batch size), accuracy on test set = 0.9896000027656555 (train in tf)

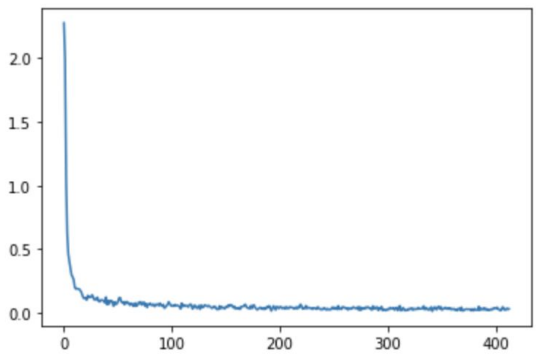
2.distribution of weights and biases

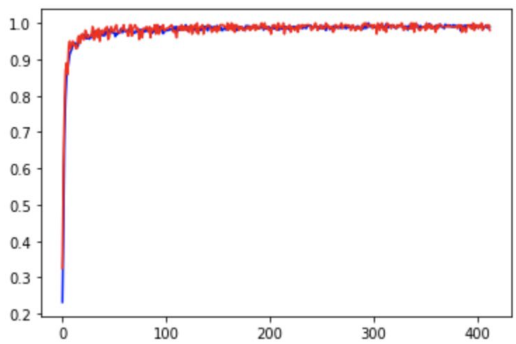




(L2 regularized, rewrite in pytorch)

2.Learning curve, accuracy curve



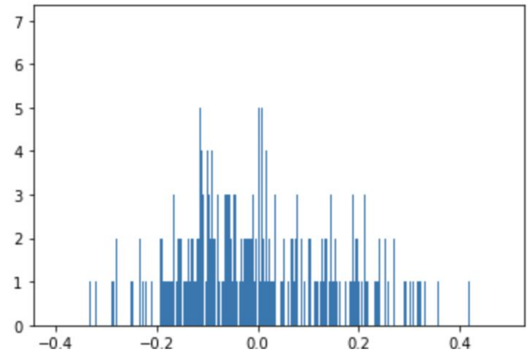


(accuracy curves: red = test, blue = train, during training test on random 200 samples)

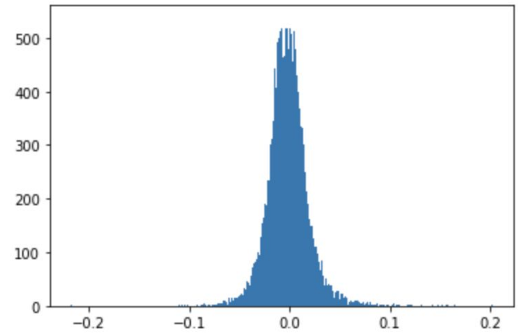
total = 7(epoch)*60000(dataset size, batch size=5), accuracy on test set = 0.99265 (train in pytorch)

2.distribution of weights and biases

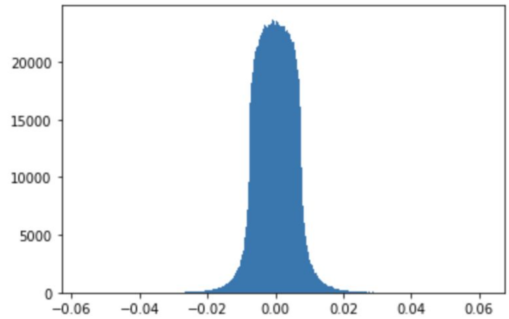
conv1.weight 800



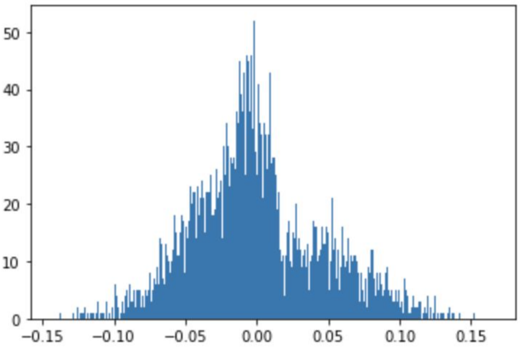
conv2.weight 51200

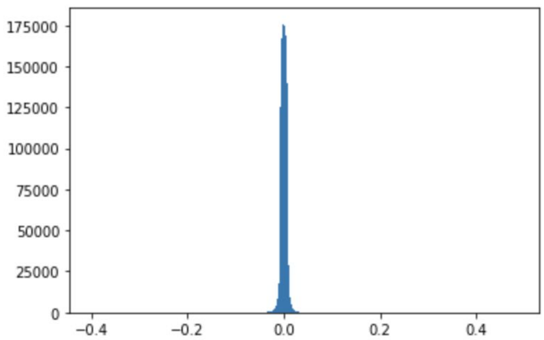


fc1.weight 3211264



fc2.weight 10240





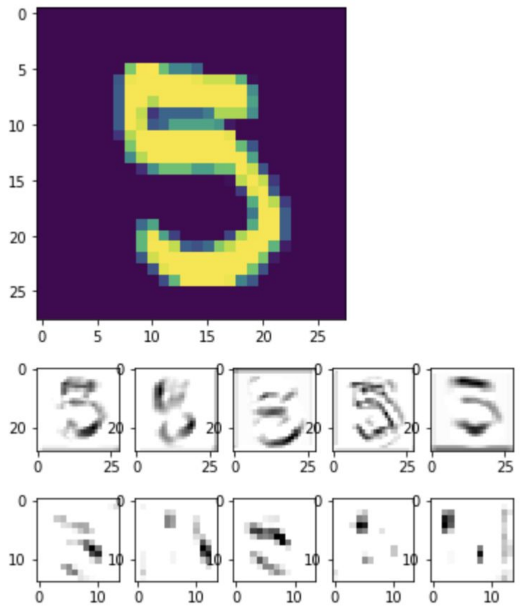
we can see that applying L2 regularization with lambda = 1e-3 made the distribution toward 0 by a significant amount and cause overall sparsity. (tf.contrib.apply_l2 and tf.layers.l2 + collection + tf.add_n all fail to work(is calculated forward but not backward), rewrite in PyTorch instead.)

The accuracy curves of train and test data are both about the same due to the lower complexity of the task.

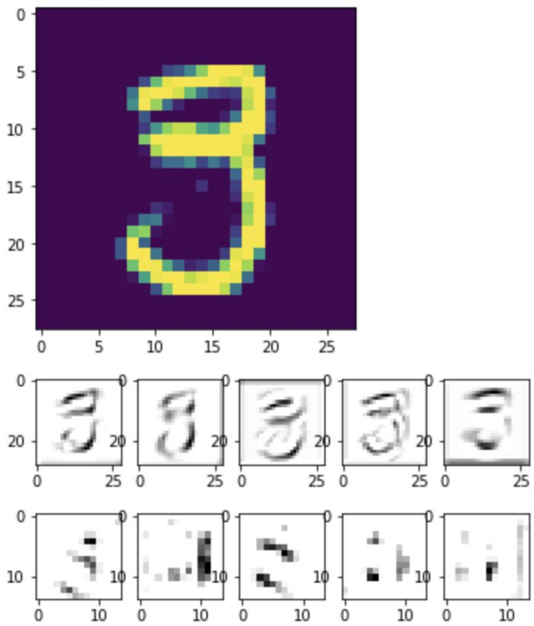
3. Examples

pair 1(5/3)

```
find mismatch @ 31
label: 5
prediction: 3
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
tensor([[1.2061e-06, 5.9835e-06, 4.4336e-08,
         1.1896e-06, 1.0466e-05, 1.2395e-04,
       grad_fn=<SoftmaxBackward>)
```

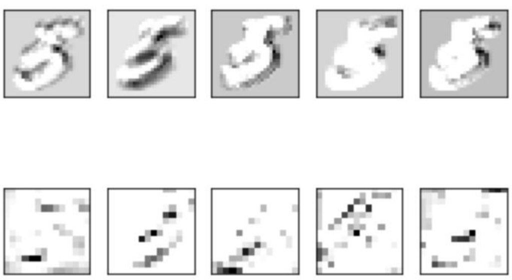


```
find sample of 3 @ 5
```

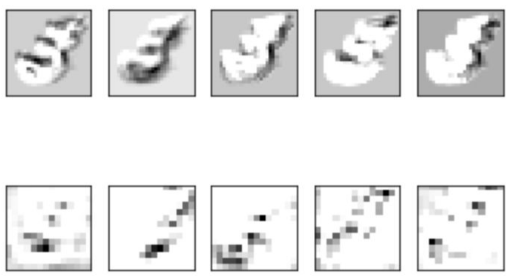


pair 2(5/3)

```
label :  5
prediction :  3
prediction vector:  [[1.5221794e-07 8.99904
  1.4286059e-01 1.5580963e-06 4.4749604e-06
```

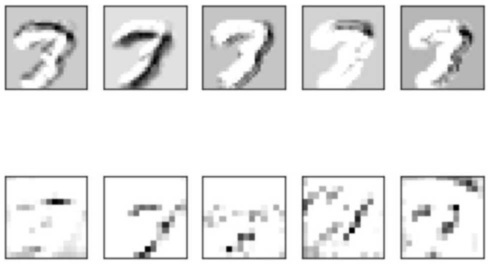


```
label :  3
prediction :  3
prediction vector:  [[1.04192395e-06 1.3698
  1.69246312e-06 9.59719357e-04 1.38573605e
  1.60067371e-04 6.44983811e-05]]
```

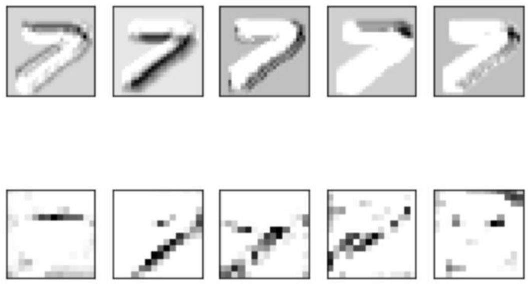


pair 3(3/7)

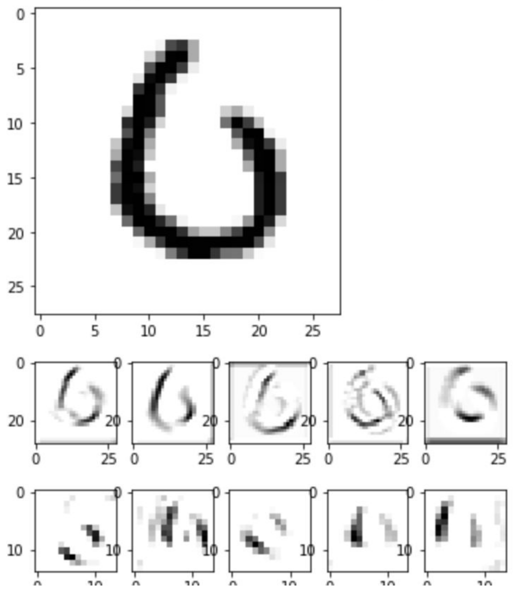
```
label :  3
prediction :  7
prediction vector:  [[4.46038285e-07 3.1167
  1.40096015e-08 1.81879168e-05 3.26417582e
  2.48183795e-07 1.32240623e-03]]
```



```
find example of correct prediction of label
label :  7
prediction :  7
prediction vector:  [[1.5863737e-08 8.6987
  2.1697337e-09 1.7434316e-11 9.9954081e-0
```

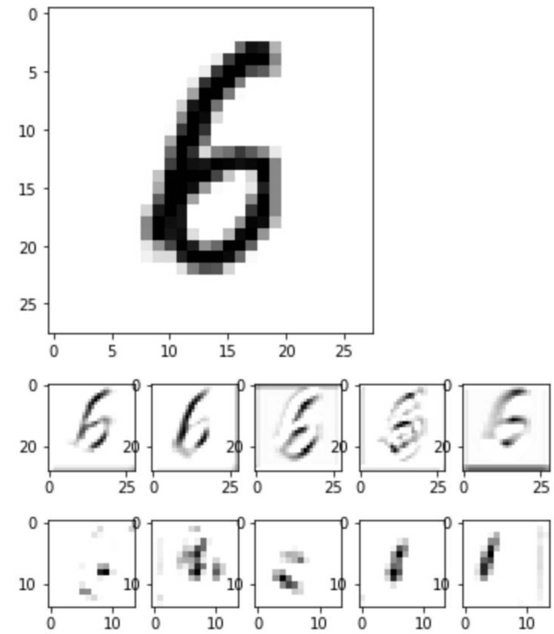


```
find mismatch @ 180
label: 0
prediction: 6
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
tensor([[4.0176e-01, 4.0673e-07, 1.0058e-06,
         5.9818e-01, 1.8804e-07, 1.1407e-06,
       grad_fn=&lt;SoftmaxBackward&gt;)
```



pair 4(6/0)

```
find sample of 6 @ 8
```



For all samples, we see that in hidden layers, our model does catch some important features(connectedness/segment/boundary)

pair 1: the model is misled by the soft connection in the right side.

pair 2: the model struggles to determine 5 from 3 as the connection part is not clear.

pair 3: the model fails to capture the small curve in the middle.

part 4: the hidden activation captures useful information and this picture is even controversial to human.

Through the visualization of hidden layers of the model.

We can see it tries to capture the boundary int the first convolution layer and continues to transform the data to abstract concepts in the second convolution layer, finally reaching a 99%up accuracy on MNIST dataset.

Task 2: CIFAR-10 dataset

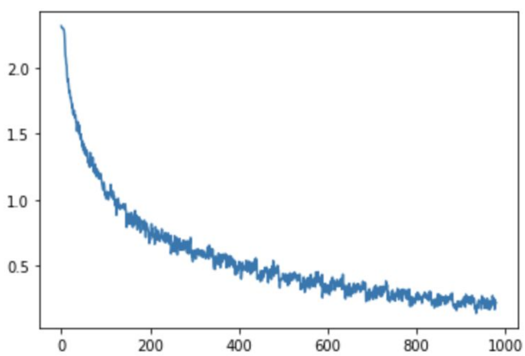
1.Architecture(PyTorch code)

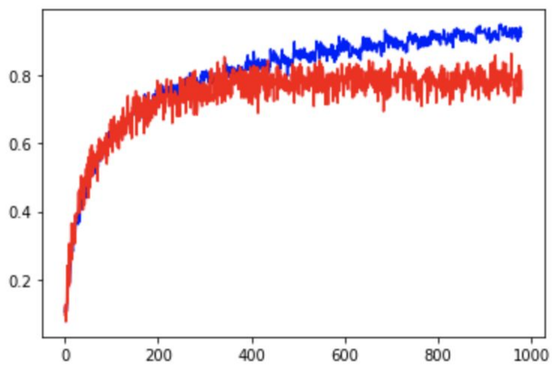
```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        #in, out, kernel
        self.conv1 = nn.Conv2d(3,   64,  3) #3 32 32, 64 30 30
        self.conv2 = nn.Conv2d(64,  128, 3) #64 15 15, 128 15 15
        self.conv3 = nn.Conv2d(128, 256, 3) #128 7 7, 256 4 4
        self.pool = nn.MaxPool2d(2, 2)
        self.fc1 = nn.Linear(256*2*2, 128) #256 2 2, 128
        self.fc2 = nn.Linear(128, 256)     #128 256
        self.fc3 = nn.Linear(256, 10)      #256 10

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = self.pool(F.relu(self.conv3(x)))
        x = x.view(-1, 256*2*2)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

```
Tainable Valuables:
conv1.weight      1728
conv1.bias        64
conv2.weight      73728
conv2.bias        128
conv3.weight      294912
conv3.bias        256
fc1.weight        131072
fc1.bias          128
fc2.weight        32768
fc2.bias          256
fc3.weight        2560
fc3.bias          10

Total     537610
```

(NO L2 regularized)

2.Learning curve, accuracy curve



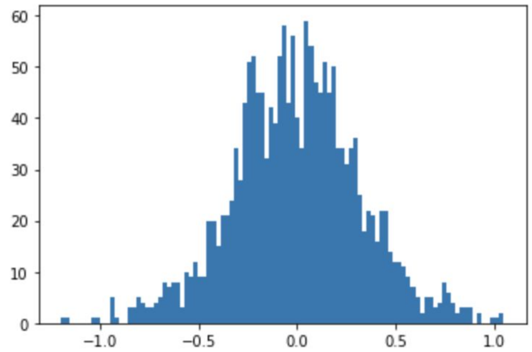



```
calc_accuracy(model, test_all_loader, device)
```

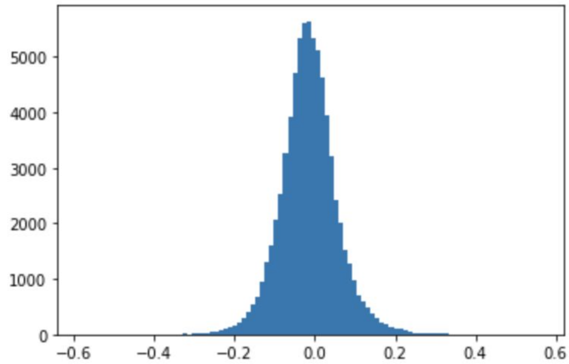
```
0.7901
```

2.Weight and biases

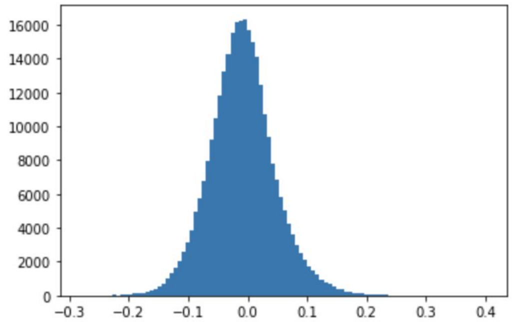
conv1.weight 1728



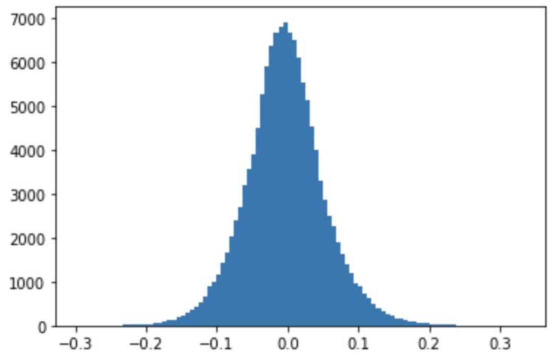
conv2.weight 73728



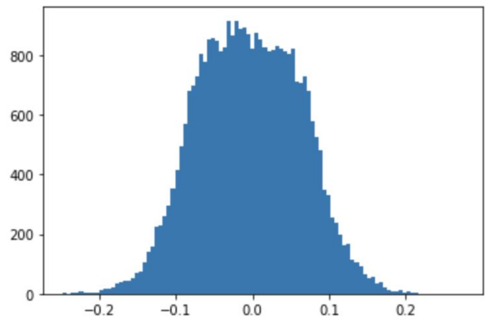
conv3.weight 294912



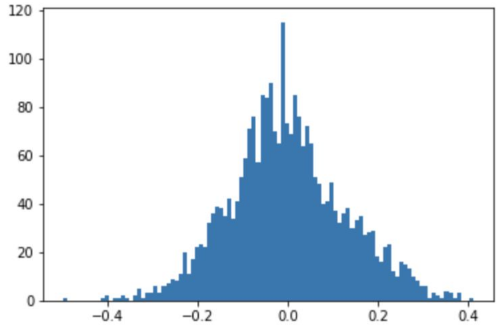
fc1.weight 131072

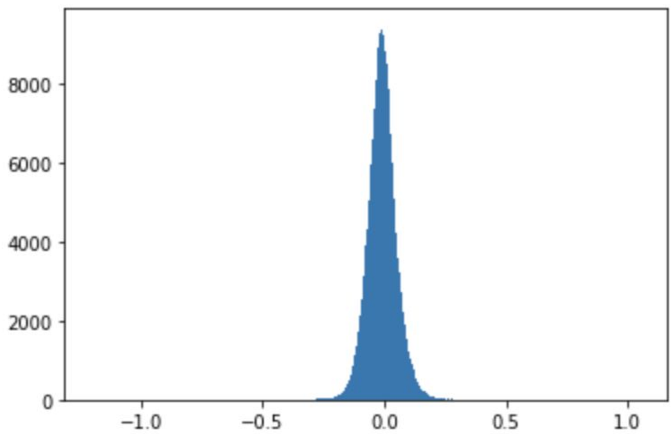


fc2.weight 32768



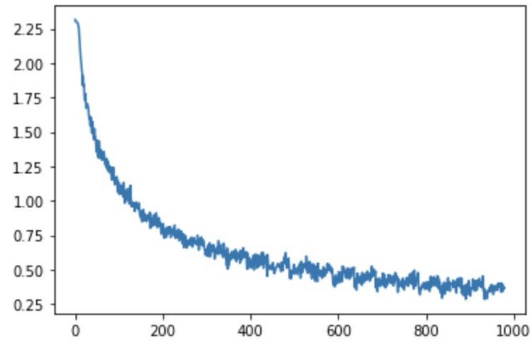
fc3.weight 2560

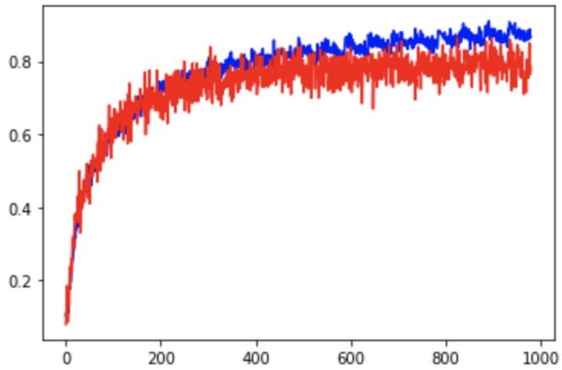




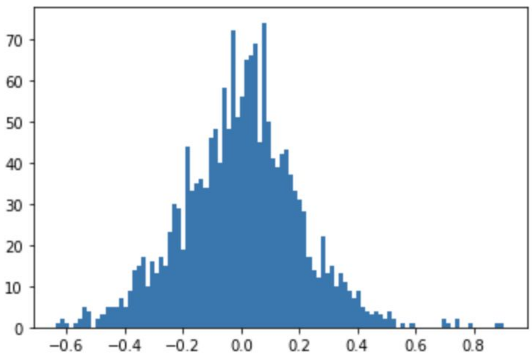
```
calc_accuracy(model, test_all_loader, device)
```

(L2 Regularized, lambda = 1e-3) 0.7808

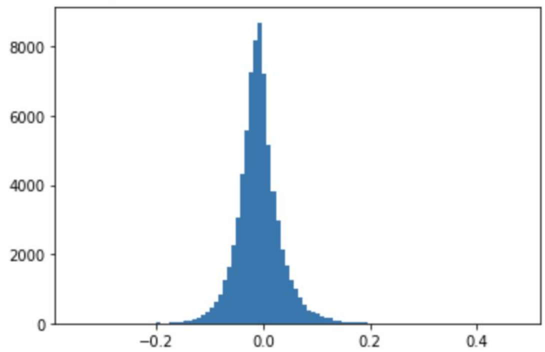




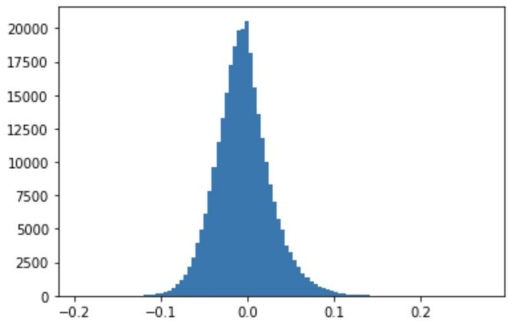
conv1.weight 1728



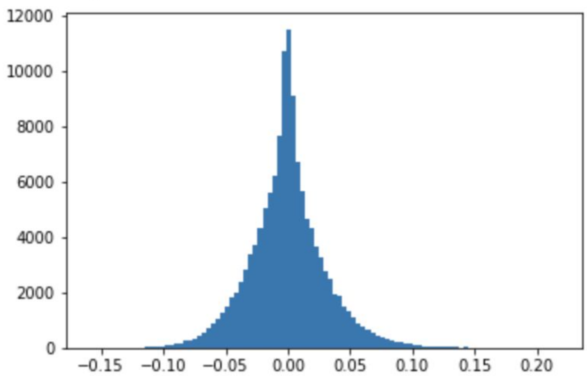
conv2.weight 73728

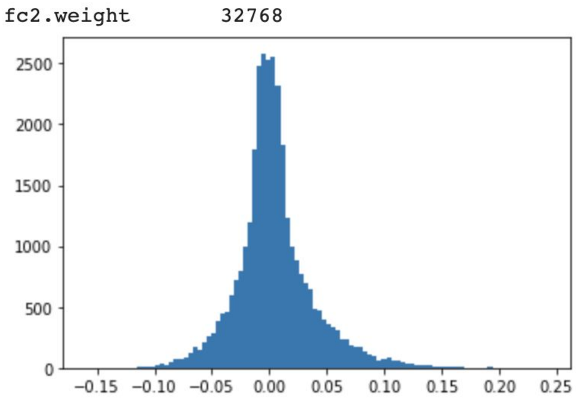


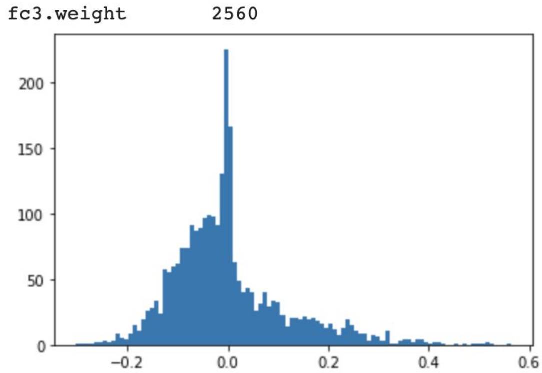
conv3.weight 294912

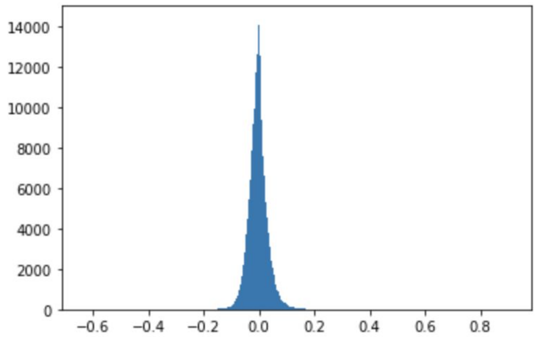


fc1.weight 131072





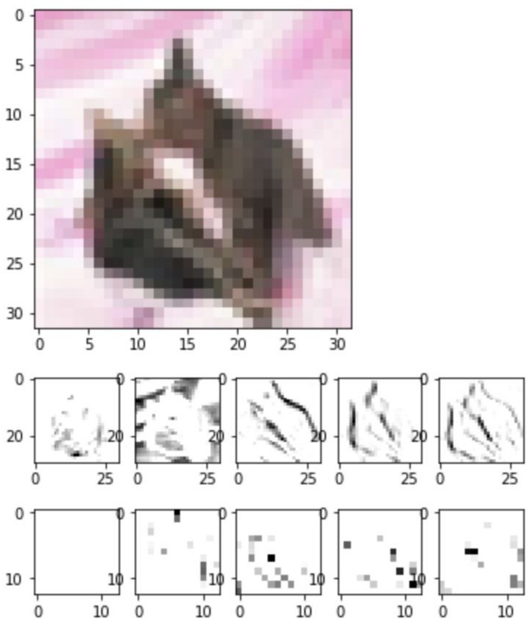


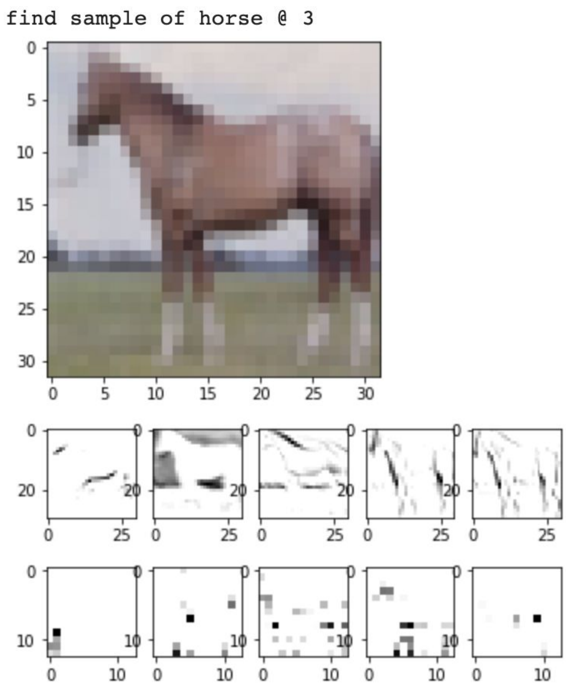


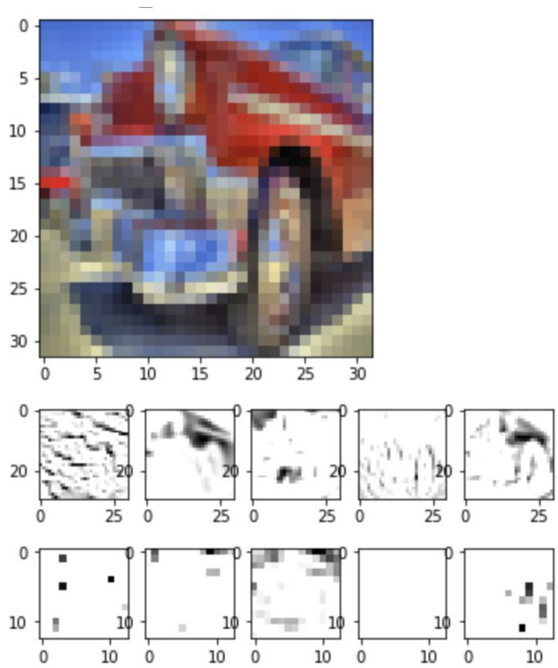
observation:

similarly, regularization did the job that making distribution closer to 0, also the difference in train and test accuracy is closer. But notice that the conv1 and fc3 layer still have a lot of bigger negative weight; This lead to the trial of self normalizing activation function(selu) in the discussion part.

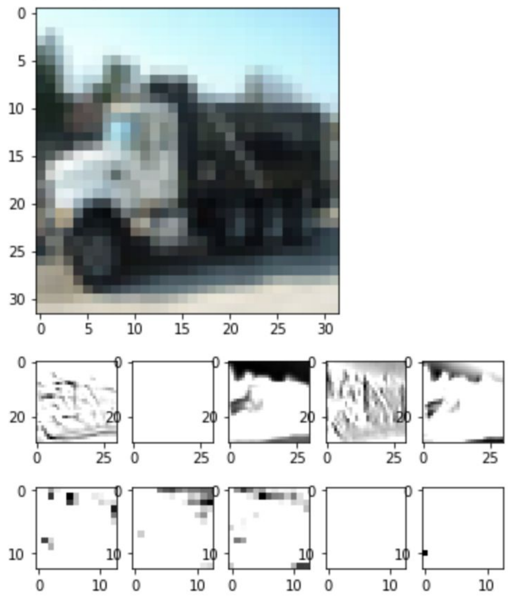
3.Example



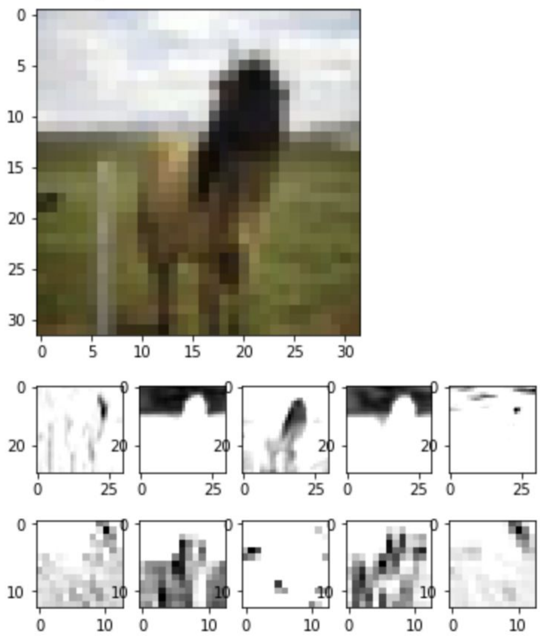




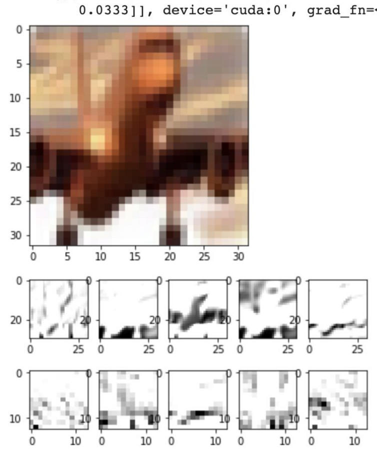
find sample of truck @ 5



find sample of horse @ 2



```
find mismatch @ 0
label: plane
prediction: horse
('plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse',
tensor([[0.0487, 0.0062, 0.2394, 0.0794, 0.1664, 0.0811, 0.0725
         0.0333]], device='cuda:0', grad_fn=<SoftmaxBackward>)
```



The model is able to construct the boundaries most of the time, and the conv2 activation is now hard to understand by human beings, but it's maybe because that the # of features of conv2 is 128 so that the samples printed out can be less related to the current classes. However, it is still confusing to me that in the first two pairs, the features are express in pointwise, while in the last pair(plane/horse) it(the same filters) tends to express in strides and blurred pictures instead.

By the training accuracy and testing, accuracy is close and the learning curve is stabilizing, an inference can be made that the model is not powerful enough to fit the CIFAR10 dataset.

But due to the limited time, and there are actually a lot of deep network structures on published papers perform well on CIFAR10. It seems not meaningful to train one copy in our homework.

5. Preprocessing of CIFAR10

```
transform_train = transforms.Compose(
    [
     transforms.RandomHorizontalFlip(),
     transforms.RandomGrayscale(),
     transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

transform = transforms.Compose(
    [
     transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])
```

preprocessing + data augmentation by Pytorch

for the large noise and variety of CIFAR10 data, I do the augmentations so that the model can learn that pictures of different horizontal directions(head to the left or to the right)/colors(cause by the lights and shadows) should be classified as the same labels to the original data.

Discussion

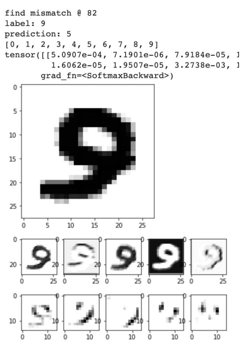
1. Architecture and Model Experiments
   a. smaller filter size on MNIST (3/7 case)
   b. smaller filter size + double filter # MNIST(balance param #)
   c. selu on CIFAR10 (weights tend to be negative)
   d. avg pooling on CIFAR10(inspired by ALEX net)
2. MNIST vs CIFAR10 accuracy curve
3. Data augmentation by transformation
4. Negative weights in CIFAR10
5. Inference on why regularized model performed poorly on CIFAR10
6. Reason to rewrite MNIST L2 to PyTorch

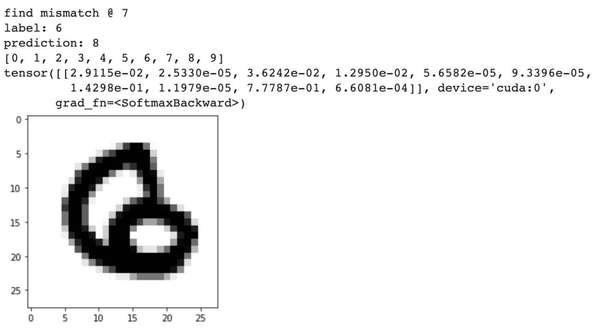
1a. 1b.

```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        #in, out, kernel, padding(both sides=>*2)
        self.conv1 = nn.Conv2d(1, 32, 3, padding=1)
        self.conv2 = nn.Conv2d(32, 64, 3, padding=1)
        self.pool = nn.MaxPool2d(2, 2, padding=0)
        self.fc1 = nn.Linear(64*7*7, 1024)
        self.fc2 = nn.Linear(1024, 10)
        self.dpout = nn.Dropout(p=0.5)
```

```
Tainable Valuables:
conv1.weight      288
conv1.bias        32
conv2.weight      18432
conv2.bias        64
fc1.weight        3211264
fc1.bias          1024
fc2.weight        10240
fc2.bias          10

Total     3241354
```

```
Tainable Valuables:
conv1.weight      576
conv1.bias        64
conv2.weight      73728
conv2.bias        128
fc1.weight        6422528
fc1.bias          1024
fc2.weight        10240
fc2.bias          10

Total     6508298
```





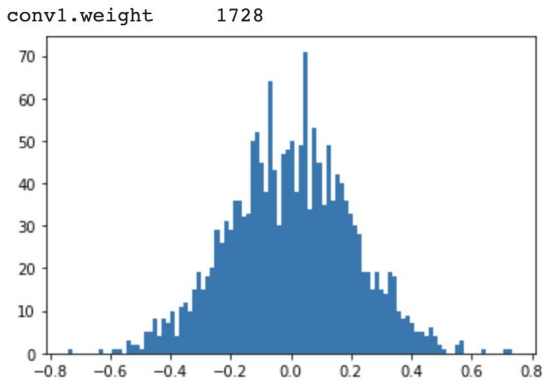
Inspire by the mismatched pair 3/7, want to see if smaller kernel size can emphasize the subtle difference.

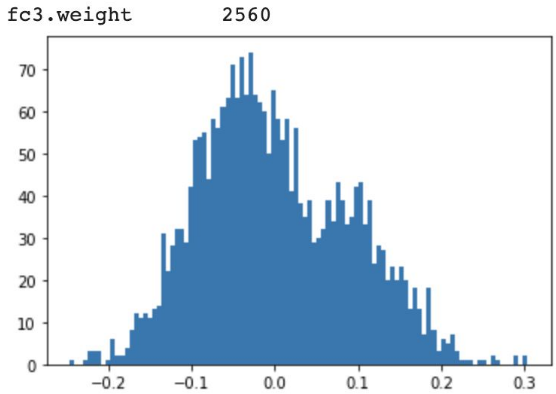
The accuracy of this 3-size-filter model(with L2) is still 99.02/98.95(double params) compared to 99.286 of the 5-size-filter model(with L2)
but it seems that there are much more simple mismatches.
I think its caused by small kernel size fail to capture features such as longer segment, etc.
Also, doubling params seems not to work, 32/64 filters are enough for the structure of the network.

1c. selu on CIFAR10(other params fixed, lambda = 1e-3)

```
def forward(self, x):
    x = self.pool(F.selu(self.conv1(x)))
    x = self.pool(F.selu(self.conv2(x)))
    x = self.pool(F.selu(self.conv3(x)))
    x = x.view(-1, 256*2*2)
    x = F.relu(self.fc1(x))
    x = F.relu(self.fc2(x))
    x = self.fc3(x)
    return x
```

This doesn't improve the performance as I thought.
Applying selu, conv1 remains untouched(not passing selu yet), f3 weight is indeed centralized.
Accuracy dropped from 0.78 to 0.76.
Maybe normalizing is not such a good idea in this model or this task.
My inference is that bigger negative weight and z-out is actually good as a way to "strongly" suggest that a picture is "less likely" to be a class.

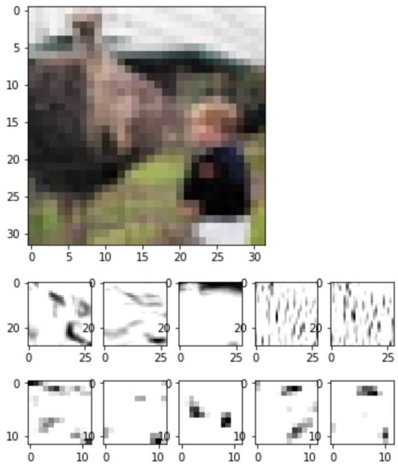


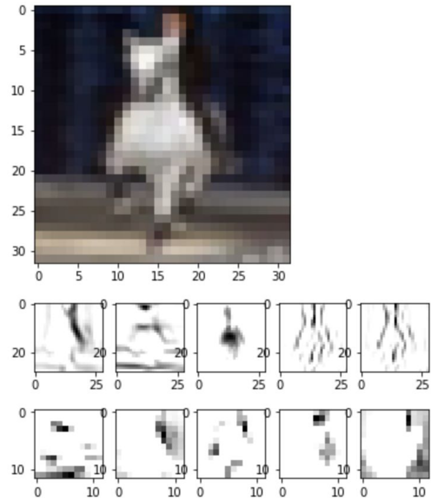


1d. average pooling on CIFAR(inspired by ALEX net)

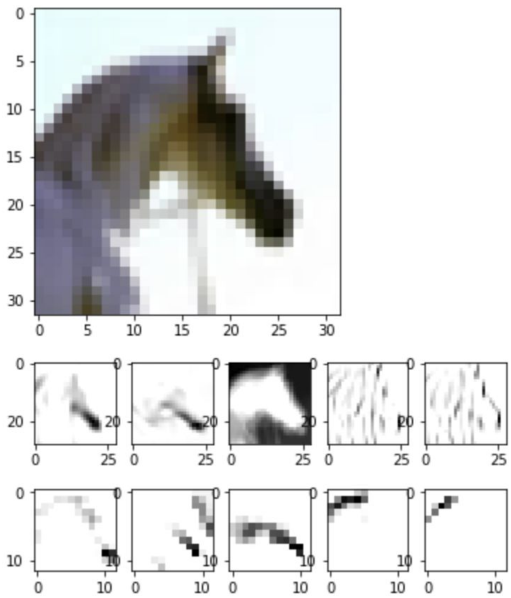
```
def __init__(self):
    super(CNN, self).__init__()
    #in, out, kernel
    self.conv1 = nn.Conv2d(3,   64,  5, 1)
    self.conv2 = nn.Conv2d(64,  128, 3) #6
    self.conv3 = nn.Conv2d(128, 256, 3) #1
    self.pool = nn.AvgPool2d(2, 2)
    self.fc1 = nn.Linear(256*2*2, 128) #25
    self.fc2 = nn.Linear(128, 256)     #12
    self.fc3 = nn.Linear(256, 10)      #25
```

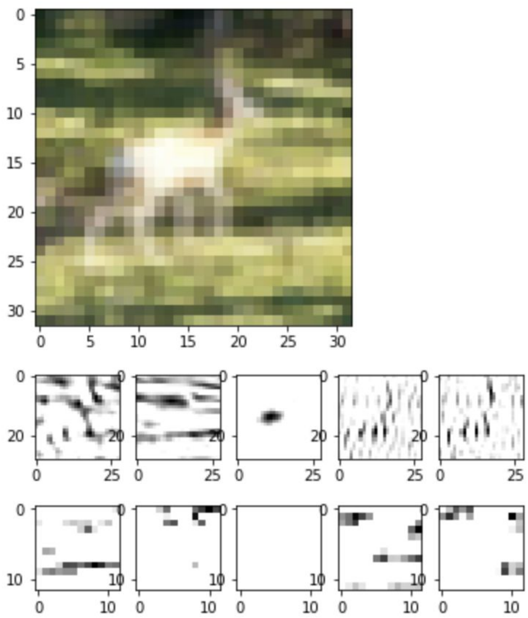
find sample of horse @ 13



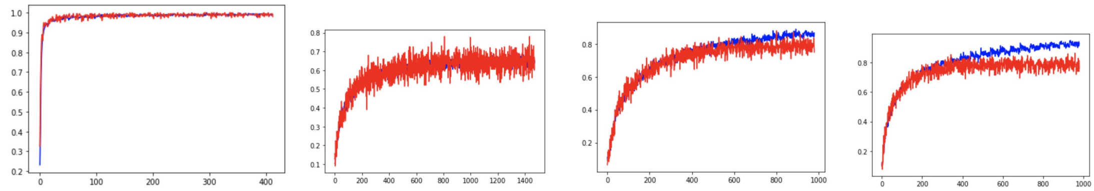


accuracy doesn’t change much(from 0.78 to 0.7964).
but the conv layers seem much meaningful in my opinion.
seems the network really has to go deeper to get better performance.





2.MNIST vs CIFAR10 accuracy curve:



from left to tight, MNIST, CIFAR reg lambda=0.01, CIFAR L2 reg lambda=1e-3, CIPAR no reg
This clearly demonstrates:
the low complexity of MNIST
the effect of the L2 regularization as a good way to close the gap between training and testing data set.
L2 regularization do so at the cost of the power of the model to fit data
(test accuracy: no reg-0.79 reg-1e-3-0.78 reg-001-.6277)

3. Data augmentation by transformation

As discussed formerly, is a good way to augment data and increase the model's generalization ability.

4. Negative weights in CIFAR10

As discussed formerly, I think that it is required for my model to perform well on the task by rejecting possibilities for an input being of a class when some feature is observed.

5. Inference on why regularized model performed poorly on CIFAR10

the lack of power of model + regularization further limits the power.

6. Reason to rewrite MNIST L2 to PyTorch

I wanted to learn both TF1.0 and Pytorch at the same time, so both Libraries are used in my homework, also, I implemented save/load model/record structure because I was doing works on Google Colab.

Actually, there was a version written in TF 2.0 as well. These works took me a lot of time.

But in the TF1.0 version of MNIST task, I was bugged over a seemly completely correct code, the total loss is huge, but the optimizer fails to optimize take consideration of L2 losses. So finally, I gave out and turn to changing my pytorch code for MNIST task with the same architecture as TF1.0 tp finish the homework.

pictures of codes:

```
def weight_variable(shape, lamb=0.001):
    initial = tf.truncated_normal(shape, stddev=0.1)
    tf.add_to_collection('Ws', initial)
    if lamb > 0.0:
        print('L2 regularization activated')
        L2loss = tf.multiply(tf.nn.l2_loss(initial), lamb, name='L2_loss')
        tf.add_to_collection('losses', L2loss)
    return tf.Variable(initial)
def bias_variable(shape):
  initial = tf.constant(0.1, shape = shape)
  tf.add_to_collection('bs', initial)
  return tf.Variable(initial)
```

```
#maxpool1 + conv2 + maxploo2
with tf.name_scope("conv2"):
  h_pool1 = max_pool_2x2(h_dconv)
  W_conv2 = weight_variable([5, 5, 32, 64])
  b_conv2 = bias_variable([64])
  h_conv2 = tf.nn.relu(conv2d(h_pool1, W_conv2) + b_conv2)
  h_pool2 = max_pool_2x2(h_conv2)
```

```
L2 regularization activated
```

```
#fully connected layre 1
with tf.name_scope("fc1"):
  W_fc1 = weight_variable([7 * 7 * 64, 1024])
  b_fc1 = bias_variable([1024])

  h_pool2_flat = tf.reshape(h_pool2, [-1, 7*7*64])
  h_fc1 = tf.nn.relu(tf.matmul(h_pool2_flat, W_fc1) + b_fc1)
```

```
L2 regularization activated
```

```
#define loss and evaluations:
with tf.name_scope("Losses"):
  cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(prediction), reduction_indices=[1]))
  tf.add_to_collection('losses', cross_entropy)
  total_loss = tf.add_n(tf.get_collection('losses'))
with tf.name_scope("trainer"):
  train_step = tf.train.AdamOptimizer(1e-4).minimize(cross_entropy)
  train_step_L2 = tf.train.AdamOptimizer(1e-4).minimize(total_loss)######## L2 ##################
with tf.name_scope("statistics"):
  correct_prediction = tf.equal(tf.argmax(prediction,1), tf.argmax(y_,1))
  accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
```

```
with trange(n_iteration) as tqdmrange:
  for i in tqdmrange:
    tqdmrange.set_description('interation {}'.format(i))
    batch = mnist.train.next_batch(batch_size, shuffle=True) #get next training batch
    if i%record_frequency == 0:
      test_batch = mnist.test.next_batch(test_size, shuffle=True)
      valid_batch = mnist.validation.next_batch(test_size, shuffle=True)
      #calculate and append training history
      #becareful to use batch size to validate istead of the whole dataset
      train_CE = cross_entropy.eval(feed_dict={ x:batch[0], y_: batch[1], keep_prob: 1.0})
      train_accuracy = accuracy.eval(feed_dict={ x:batch[0], y_: batch[1], keep_prob: 1.0})
      test_accuracy = accuracy.eval(feed_dict={ x: test_batch[0], y_: test_batch[1], keep_prob : 1.0})
      valid_accuracy = accuracy.eval(feed_dict={ x: valid_batch[0], y_: valid_batch[1], keep_prob : 1.0})
      rec["loss"].append(train_CE)
      rec["train"].append(train_accuracy)
      rec["test"].append(test_accuracy)
      rec["valid"].append(valid_accuracy)
      tqdmrange.set_postfix(loss=train_CE,  train_accuracy=train_accuracy, test_accuracy=test_accuracy)
    train_step_L2.run(feed_dict={x: batch[0], y_: batch[1], keep_prob: 0.5}) #L2 reg
```