

Solution to Homework 5

Zhuo Chen and Shoeb Mohammed

April 18, 2016

1 Decision trees, entropy and information gain

1.1

$$\begin{aligned} H\left(\frac{p}{p+n}\right) &= \frac{p}{p+n} \log \frac{p+n}{p} + \frac{n}{p+n} \log \frac{p+n}{n} \\ &\leq \log \left(\frac{p}{p+n} \cdot \frac{p+n}{p} + \frac{n}{p+n} \cdot \frac{p+n}{n} \right) \quad \text{since } \log(\cdot) \text{ is concave function} \\ &= \log 2 \\ &= 1 \end{aligned} \tag{1}$$

When $p = n$, the formula gives $H(S) = \frac{1}{2} \log 2 + \frac{1}{2} \log 2 = 1$.

1.2

- Misclassification rate for A = $\frac{1}{4}$
- Misclassification rate for B = $\frac{1}{4}$
- Entropy gain model A = $\frac{3}{4} \log 3 - 1 \sim 0.1887$
- Entropy gain model B = $\frac{3}{2} - \frac{3}{4} \log 3 \sim 0.3113$.
This means entropy after split is lower for model B.
- Gini index model A = $\frac{3}{8}$
- Gini index model B = $\frac{1}{9}$

1.3

Yes, it is possible. For an example, consider a dataset with 700 examples of class C_1 and 100 of class C_2 . If a feature splits it into two leaves (200, 200) and (200, 200); then misclassification rate is bigger.

2 Bagging

2.1

Proof. Simplifying, we get $\epsilon_{bag}(x) = \frac{1}{L} \sum_{l=1}^L \epsilon_l(x)$. Thus,

$$\begin{aligned} E_{bag} &= E_X [\epsilon_{bag}(x)^2] \\ &= \frac{1}{L^2} E_X \left[\left(\sum_{l=1}^L \epsilon_l(x) \right)^2 \right] \\ &= \frac{1}{L^2} E_X \left[\sum_{l=1}^L \epsilon_l^2(x) + \sum_{\substack{1 \leq i, j \leq L \\ i \neq j}} \epsilon_i(x) \epsilon_j(x) \right] \\ &= \frac{1}{L^2} E_X \left[\sum_{l=1}^L \epsilon_l^2(x) \right] + \frac{1}{L^2} E_X \left[\sum_{\substack{1 \leq i, j \leq L \\ i \neq j}} \epsilon_i(x) \epsilon_j(x) \right] \\ &= \frac{1}{L^2} E_X \left[\sum_{l=1}^L \epsilon_l^2(x) \right] + \frac{1}{L^2} \sum_{\substack{1 \leq i, j \leq L \\ i \neq j}} E_X [\epsilon_i(x) \epsilon_j(x)] \\ &= \frac{1}{L^2} E_X \left[\sum_{l=1}^L \epsilon_l^2(x) \right] \quad \text{since } E_X [\epsilon_i(x) \epsilon_j(x)] = 0 \text{ for } i \neq j \\ &= \frac{1}{L^2} \sum_{l=1}^L E_X [\epsilon_l^2(x)] \\ &= \frac{1}{L} E_{avg} \end{aligned} \tag{2}$$

□

2.2

Proof.

$$\begin{aligned} E_{bag} &= E_X [\epsilon_{bag}(x)^2] \\ &= E_X \left[\left(\sum_{l=1}^L \frac{\epsilon_l(x)}{L} \right)^2 \right] \\ &\leq E_X \left[\sum_{l=1}^L \frac{\epsilon_l^2(x)}{L} \right] \text{ using Jensen's ineq. for } \lambda_i = \frac{1}{L}; \text{ and for random variables } 0 \leq U \leq V \implies E_p[U] \leq E_p[V] \\ &= \frac{1}{L} \sum_{l=1}^L E_X [\epsilon_l^2(x)] \\ &= E_{avg} \end{aligned} \tag{3}$$

□

3 Fully connected neural networks and convolutional neural networks

3.1 Fully connected feed-forward neural networks

3.1.1 Affine layer: forward

Testing affine_forward function:

difference (should be around 1e-9): 9.76985004799e-10

3.1.2 Affine layer: backward

Testing affine_backward function:

dx error (should be around 1e-10): 4.81624004347e-08

dtheta error (should be around 1e-10): 5.01568076009e-11

dtheta_0 error (should be around 1e-10): 1.41109427148e-11

3.1.3 ReLU layer: forward

Testing relu_forward function:

difference (should be around 1e-8): 4.99999979802e-08

3.1.4 ReLU layer: backward

Testing relu_backward function:

dx error: (should be around 1e-12): 3.27563500104e-12

Sandwich layers

Testing affine_relu_forward:

dx error: 1.94653644744e-10

dtheta error: 3.01172010229e-10

dtheta_0 error: 3.275617042e-12

Testing svm_loss:

loss: (should be around 9): 9.00075514465

dx error (should be around 1e-9): 1.40215660067e-09

Loss layers: softmax and SVM

Testing softmax_loss:

loss (should be around 2.3): 2.3026611292

dx error (should be around 1e-8): 8.78168746044e-09

3.1.5 Two layer network

Testing initialization ...
Testing test-time forward pass ...
Testing training loss (no regularization)
26.5948426952
Running numeric gradient check with reg = 0.0
theta1 relative error: 1.22e-08
theta1_0 relative error: 6.55e-09
theta2 relative error: 3.57e-10
theta2_0 relative error: 2.53e-10
Running numeric gradient check with reg = 0.7
theta1 relative error: 2.53e-07
theta1_0 relative error: 1.56e-08
theta2 relative error: 1.37e-07
theta2_0 relative error: 9.09e-10

3.1.6 Overfitting a two layer network

```
sgd_solver=solver.Solver(model,data,update_rule='sgd',  
optim_config={  
    'learning_rate': 1e-3,  
},  
lr_decay=0.95,  
num_epochs=10, batch_size=100,  
print_every=100)
```

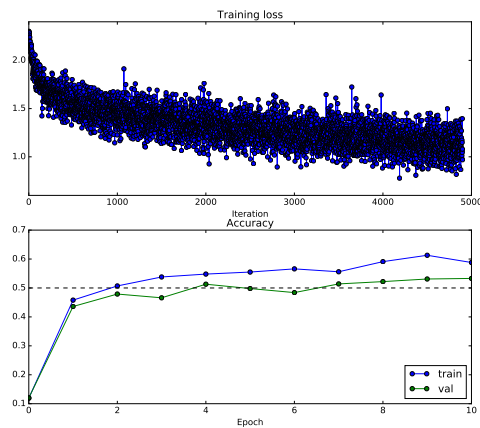


Figure 1: overfitting a two layer network

3.1.7 Multilayer network

```
Running check with reg = 0
Initial loss: 2.30057347606
theta1 relative error: 6.88e-07
theta1_0 relative error: 2.57e-08
theta2 relative error: 1.26e-06
theta2_0 relative error: 3.01e-09
theta3 relative error: 7.15e-08
theta3_0 relative error: 1.15e-10
Running check with reg = 3.14
Initial loss: 7.24807573499
theta1 relative error: 3.26e-08
theta1_0 relative error: 2.89e-08
theta2 relative error: 6.90e-06
theta2_0 relative error: 1.19e-08
theta3 relative error: 2.38e-08
theta3_0 relative error: 3.83e-10
```

3.1.8 Overfitting a three layer network

```
weight_scale = 1e-2
learning_rate = 1e-2
:
:
(Iteration 31 / 40) loss: 0.080666
(Epoch 16 / 20) train acc: 0.960000; val_acc: 0.189000
(Epoch 17 / 20) train acc: 1.000000; val_acc: 0.165000
(Epoch 18 / 20) train acc: 1.000000; val_acc: 0.181000
(Epoch 19 / 20) train acc: 1.000000; val_acc: 0.181000
(Epoch 20 / 20) train acc: 1.000000; val_acc: 0.177000
```



Figure 2: overfitting a three layer network

3.1.9 Overfitting a five layer network

On our laptop, with the particular parameters, didn't notice a significant slowdown. `learning_rate = 1e-2`

`weight_scale = 5e-2`

`:`

(Epoch 10 / 20) train acc: 0.980000; val_acc: 0.131000

(Iteration 21 / 40) loss: 0.180435

(Epoch 11 / 20) train acc: 1.000000; val_acc: 0.128000

(Epoch 12 / 20) train acc: 1.000000; val_acc: 0.130000

(Epoch 13 / 20) train acc: 0.980000; val_acc: 0.121000

(Epoch 14 / 20) train acc: 1.000000; val_acc: 0.135000

(Epoch 15 / 20) train acc: 1.000000; val_acc: 0.141000

(Iteration 31 / 40) loss: 0.037009

(Epoch 16 / 20) train acc: 1.000000; val_acc: 0.142000

(Epoch 17 / 20) train acc: 1.000000; val_acc: 0.140000

(Epoch 18 / 20) train acc: 1.000000; val_acc: 0.141000

(Epoch 19 / 20) train acc: 1.000000; val_acc: 0.138000

(Epoch 20 / 20) train acc: 1.000000; val_acc: 0.139000

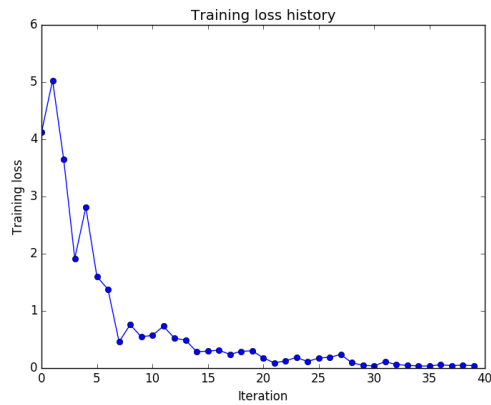


Figure 3: overfitting a five layer network

3.1.10 SGD+Momentum

next_theta error: 8.88234703351e-09

velocity error: 4.26928774328e-09

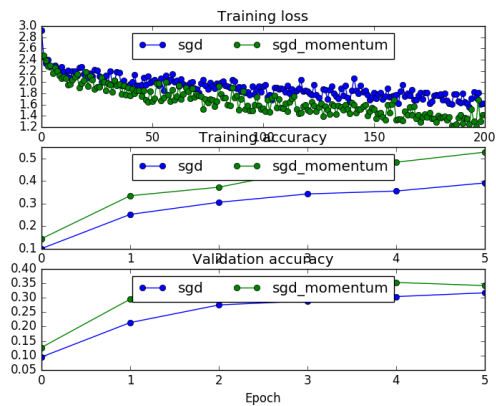


Figure 4: SGD+momentum converges faster

3.1.11 RMSProp

Test rmsprop

next_theta error: 9.52468751104e-08

cache error: 2.64779558072e-09

Test adam

next_theta error: 1.13988746733e-07

v error: 4.20831403811e-09
m error: 4.21496319311e-09

running with adam

⋮

(Epoch 4 / 5) train acc: 0.532000; val_acc: 0.378000
(Iteration 161 / 200) loss: 1.255240
(Iteration 171 / 200) loss: 1.213894
(Iteration 181 / 200) loss: 1.206805
(Iteration 191 / 200) loss: 1.078037
(Epoch 5 / 5) train acc: 0.602000; val_acc: 0.389000

running with rmsprop

⋮

(Epoch 4 / 5) train acc: 0.495000; val_acc: 0.338000
(Iteration 161 / 200) loss: 1.452923
(Iteration 171 / 200) loss: 1.495491
(Iteration 181 / 200) loss: 1.300043
(Iteration 191 / 200) loss: 1.495532
(Epoch 5 / 5) train acc: 0.522000; val_acc: 0.352000

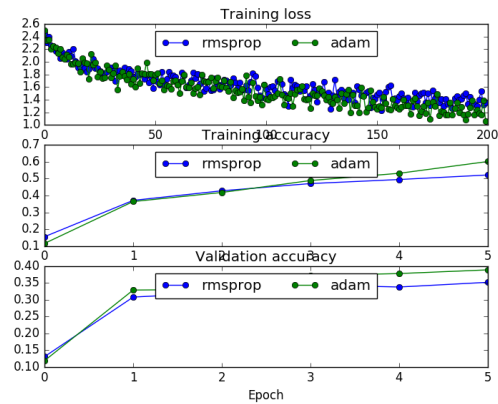


Figure 5: RMSProp and adam

3.1.12 Training a fully connected network for the CIFAR-10 dataset

```
num_epochs = 15
batch_size = 100
update_rule = adam
learning_rate = 5e-4
⋮
⋮
```

(Iteration 7291 / 7350) loss: 1.517843
(Iteration 7301 / 7350) loss: 1.358143
(Iteration 7311 / 7350) loss: 1.385554
(Iteration 7321 / 7350) loss: 1.652732
(Iteration 7331 / 7350) loss: 1.717880
(Iteration 7341 / 7350) loss: 1.258574
(Epoch 15 / 15) train acc: 0.587000; val_acc: 0.520000

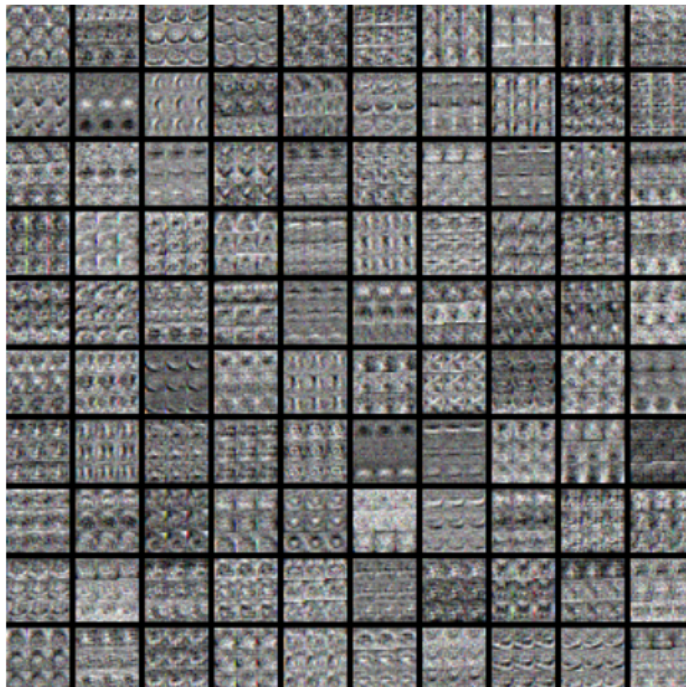


Figure 6: first level weights: Training a fully connected network for the CIFAR-10 dataset

3.2 Convolutional neural networks

3.2.1 Convolution: naive forward pass

Testing conv_forward_naive
difference: 2.21214765759e-08

3.2.2 Convolution: naive backward pass

Testing conv_backward_naive function
dx error: 3.8095645582e-09
dtheta error: 2.06384371241e-09
dtheta0 error: 1.75061765296e-11

3.2.3 Max pooling: naive forward pass

Testing max_pool_forward_naive function:
difference: 4.16666651573e-08

3.2.4 Max pooling: naive forward pass

Testing max_pool_backward_naive function:
dx error: 1.89289002555e-11

Fast layers

Testing conv_forward_fast:
Naive: 0.186149s
Fast: 0.047950s
Speedup: 3.882143x
Difference: 1.08780209978e-12

Testing conv_backward_fast:
Naive: 0.051805s
Fast: 0.018498s
Speedup: 2.800583x
dx difference: 2.22406937467e-12
dtheta difference: 1.01018341604e-13
dtheta0 difference: 1.82671908726e-15

Testing pool_forward_fast:
Naive: 0.022964s
fast: 0.005683s
speedup: 4.040863x
difference: 0.0

Testing pool_backward_fast:
Naive: 0.920883s
speedup: 38.093981x
dx difference: 0.0

Convolutional sandwich layers

Testing conv_relu_pool
dx error: 1.61462396734e-07
dtheta error: 3.47841604703e-10
dtheta0 error: 5.0140326449e-11

Testing conv_relu:
dx error: 6.41173584371e-09
dtheta error: 1.14601136903e-09
dtheta0 error: 1.85216131307e-10

3.2.5 Three layer convolutional neural network

Testing the CNN: loss computation

Initial loss (no regularization): 2.30258719511
Initial loss (with regularization): 2.50906848795

Testing the CNN: gradient check

theta1 max relative error: 9.317318e-03
theta1_0 max relative error: 1.084071e-05
theta2 max relative error: 2.227715e-02
theta2_0 max relative error: 9.350040e-08
theta3 max relative error: 5.010592e-05
theta3_0 max relative error: 1.805620e-09

Testing the CNN: overfit small data

⋮
(Iteration 18 / 20) loss: 0.617444
(Epoch 9 / 10) train acc: 0.830000; val_acc: 0.212000
(Iteration 19 / 20) loss: 0.580641
(Iteration 20 / 20) loss: 0.491107
(Epoch 10 / 10) train acc: 0.880000; val_acc: 0.221000

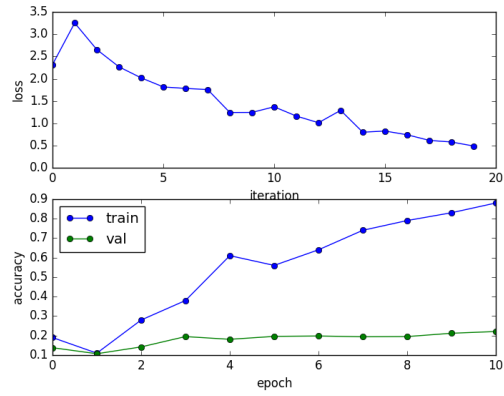


Figure 7: testing CNN: overfit small data

3.2.6 Train the CNN on the CIFAR-10 data

Training a three-layer convolutional network with hidden dimension of 500 for one epoch.

(Epoch 0 / 1) train acc: 0.083000; val_acc: 0.119000

(Epoch 1 / 1) train acc: 0.476000; val_acc: 0.481000



Figure 8: visualizing first layer convolutional filters. Epochs=2, $0.4 < \text{val_accuracy} < 0.5$

To improve validation accuracy $\geq 50\%$; we arrived at these parameters

- num_epochs=5
- batch_size=50
- update_rule='adam',
- 'learning_rate': 1e-3

⋮

(Epoch 0 / 5) train acc: 0.128000; val_acc: 0.097000
 (Epoch 1 / 5) train acc: 0.527000; val_acc: 0.527000
 (Epoch 2 / 5) train acc: 0.572000; val_acc: 0.529000
 (Epoch 3 / 5) train acc: 0.594000; val_acc: 0.552000
 (Epoch 4 / 5) train acc: 0.629000; val_acc: 0.574000
 (Epoch 5 / 5) train acc: 0.611000; val_acc: 0.576000



Figure 9: visualizing first layer convolutional filters. $\text{val_accuracy} > 0.5$