HW8 Fully-Connected Neural Networks and "Dropout"

M113040105 劉東霖

壹.所使用到的 function:

一.rms_prop:

更新的公式如下圖所示:

RMSProp

$$v_t = \beta * v_{t-1} + (1-\beta)(\nabla w_t)^2$$

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{(v_t)} + \epsilon} \nabla w_t$$

如下圖所示，我先把會用到的東西從 config 字典裡面拿出來:

```
grad_squared=config['cache']
learning_rate=config['learning_rate']
decay_rate=config['decay_rate']
epsilon=config['epsilon']
```

再來算出目前為止 gradient 的總和，這裡算總和的方式為 Exponentially weighted averages，因為 gradient 有正有負，所以一開始先平方相加，要使用時再開根號即可。公式如下:

$$V_t = \beta * V_{t-1} + (1-\beta) * \theta_t$$

V_t as approximately average over $\approx \frac{1}{(1-\beta)}$ days' temperature

程式碼如下:

```
grad_squared=decay_rate*grad_squared+(1-decay_rate)*dw*dw
```

再來就是更新梯度的部分。跟 sgd momentum 的差別在於多除了一項 sqrt(grad_squared)+epsilon 來調整學習率，epsilon 是為了防止 overflow。程式碼如下:

```
next_w=w-(learning_rate*dw)/(torch.sqrt(grad_squared)+epsilon)
```

最後再把所有東西存回字典裡面，程式碼如下:

```
config['cache']=grad_squared
config['learning_rate']=learning_rate
config['decay_rate']=decay_rate
config['epsilon']=epsilon
```

二. adam:

更新的公式如下圖所示:

$$
\begin{aligned}
m_t &= \beta_1 * m_{t-1} + (1-\beta_1) * \nabla w_t \\
v_t &= \beta_2 * v_{t-1} + (1-\beta_2) * (\nabla w_t)^2 \\
\hat{m}_t &= \frac{m_t}{1-\beta_1^t} \qquad \hat{v}_t = \frac{v_t}{1-\beta_2^t} \\
w_{t+1} &= w_t - \frac{\eta}{\sqrt{\hat{v}_t + \epsilon}} * \hat{m}_t
\end{aligned}
$$

如下圖所示，我先把會用到的東西從 config 字典裡面拿出來，然後 t 執行前先+1:

```
learning_rate=config['learning_rate']
epsilon=config['epsilon']
b1=config['beta1']
b2=config['beta2']
m=config['m']
v=config['v']
t=config['t']
t+=1
```

再來算出目前為止 Mean Squared Gradient 和 Root Mean Squared Gradient 的總和。程式碼如下:

```
m=b1*m+(1-b1)*dw
v=b2*v+(1-b2)*dw**2
```

再來修正 Mean Squared Gradient 和 Root Mean Squared Gradient 的偏差，因 gradient 的總和可能會受到過大或過小的梯度估計值的影響，進而導致收斂速度變慢或不穩定。因此，修正偏差估計是 Adam 算法的一個重要步驟，有助於改進其性能。程式碼如下:

```
m_unbias=m/(1-b1**t)
v_unbias=v/(1-b2**t)
```

再來就是更新梯度的部分，程式碼如下:

```
next_w=w-learning_rate*m_unbias/(torch.sqrt(v_unbias)+epsilon)
```

最後再把所有東西存回字典裡面，程式碼如下:

```
config['m']=m
config['v']=v
config['t']=t
config['learning_rate']=learning_rate
config['epsilon']=epsilon
config['beta1']=b1
config['beta2']=b2
```

三.dropout forward and backward:

Dropout forward 在訓練的時候，會隨機生成 0 到 1 的數，假設 p=0.4，代表 40%會被 dropout，60%會保持原本的輸出。除以 p 的部分是讓 train 和 test 的值一樣。在測試的時候不需要 dropout，值直接 pass 過去。程式碼如下:

```
if mode == 'train':
    ###########################################################################
    # TODO: Implement training phase forward pass for inverted dropout.
    # Store the dropout mask in the mask variable.
    ###########################################################################
    # Replace "pass" statement with your code
    p=1-p
    mask=(torch.rand(*x.shape,device=x.device)<p)/p
    out=x*mask
    ###########################################################################
    #                             END OF YOUR CODE
    ###########################################################################
elif mode == 'test':
    ###########################################################################
    # TODO: Implement the test phase forward pass for inverted dropout.
    ###########################################################################
    # Replace "pass" statement with your code
    out=x
    ###########################################################################
    #                             END OF YOUR CODE
    ###########################################################################
```

Dropout backward 在訓練的時候，看哪裡有被 dropout 就不要 pass 過去，沒被 dropout 就還原原來的值。在測試的時候不需要 dropout，值直接 pass 過去。程式碼如下:

```
if mode == 'train':
    ####################
    # TODO: Implement
    ####################
    # Replace "pass"
    dx=dout*mask
    ####################
    #
    ####################
elif mode == 'test':
    dx = dout
```

四.Fully connected net:

這裡我只列出我在 loss 修改的部分。

在 forward 時，如果有使用到 dropout 的話，就用 dropout.forward 去看神經元有沒有被關掉，並用 cache 儲存結果，程式碼如下:

```
if self.use_dropout:
    out,cache=Dropout.forward(out,self.dropout_param)
    caches_drop.append(cache)
```

剛剛在 forward 時是先 forward 再看有沒有 dropout，在 backward 時順序就應該反過來，並用 cache.pop()傳遞中間參數。程式碼如下:

```
if self.use_dropout:
    dout=Dropout.backward(dout,caches_drop.pop())
```

完整的程式碼如下:

```python
out=X
caches=[]
caches_drop=[]
for i in range(self.num_layers-1):
    out,cache=Linear_ReLU.forward(out,self.params['W'+repr(i+1)],self.params['b'+repr(i+1)])
    caches.append(cache)
    if self.use_dropout:
        out,cache=Dropout.forward(out,self.dropout_param)
        caches_drop.append(cache)
##########################################################################
#                                    END OF YOUR CODE
##########################################################################
scores, cache = Linear.forward(out, self.params['W'+repr(self.num_layers)], self.params['b'+repr(self.num_layers)])
caches.append(cache)
```

```python
data_loss,dscore=softmax_loss(scores,y)
reg_loss=0.0
for i in range(self.num_layers):
    w=self.params['W'+repr(i+1)]
    reg_loss+=self.reg*torch.sum(w**2)*0.5
loss=data_loss+reg_loss
dout,dw,db=Linear.backward(dscore,caches.pop())
grads['W'+repr(self.num_layers)]=dw+self.reg*self.params['W'+repr(self.num_layers)]
grads['b'+repr(self.num_layers)]=db
for i in range(self.num_layers-2,-1,-1):
    if self.use_dropout:
        dout=Dropout.backward(dout,caches_drop.pop())
    dout,dw,db=Linear_ReLU.backward(dout,caches.pop())
    grads['W'+repr(i+1)]=dw+self.reg*self.params['W'+repr(i+1)]
    grads['b'+repr(i+1)]=db
```

貳.執行結果:

1.如下圖所示，用 rms prop 算出來的 w 和 cache 裡的參數跟實際上沒差很多:

```python
expected_next_w = torch.tensor([
    [-0.39223849, -0.34037513, -0.28849239, -0.23659121, -0.18467247],
    [-0.132737,   -0.08078555, -0.02881884,  0.02316247,  0.07515774],
    [ 0.12716641,  0.17918792,  0.23122175,  0.28326742,  0.33532447],
    [ 0.38739248,  0.43947102,  0.49155973,  0.54365823,  0.59576619]],
    dtype=torch.float64, device='cuda')
expected_cache = torch.tensor([
    [ 0.5976,      0.6126277,   0.6277108,   0.64284931,  0.65804321],
    [ 0.67329252,  0.68859723,  0.70395734,  0.71937285,  0.73484377],
    [ 0.75037008,  0.7659518,   0.78158892,  0.79728144,  0.81302936],
    [ 0.82883269,  0.84469141,  0.86060554,  0.87657507,  0.8926    ]],
    dtype=torch.float64, device='cuda')

print('next_w error: ', usefuns.grad.rel_error(expected_next_w, next_w))
print('cache error: ', usefuns.grad.rel_error(expected_cache, config['cache']))
```

```
next_w error:  4.064797880829826e-09
cache error:  1.8620321382570356e-09
```

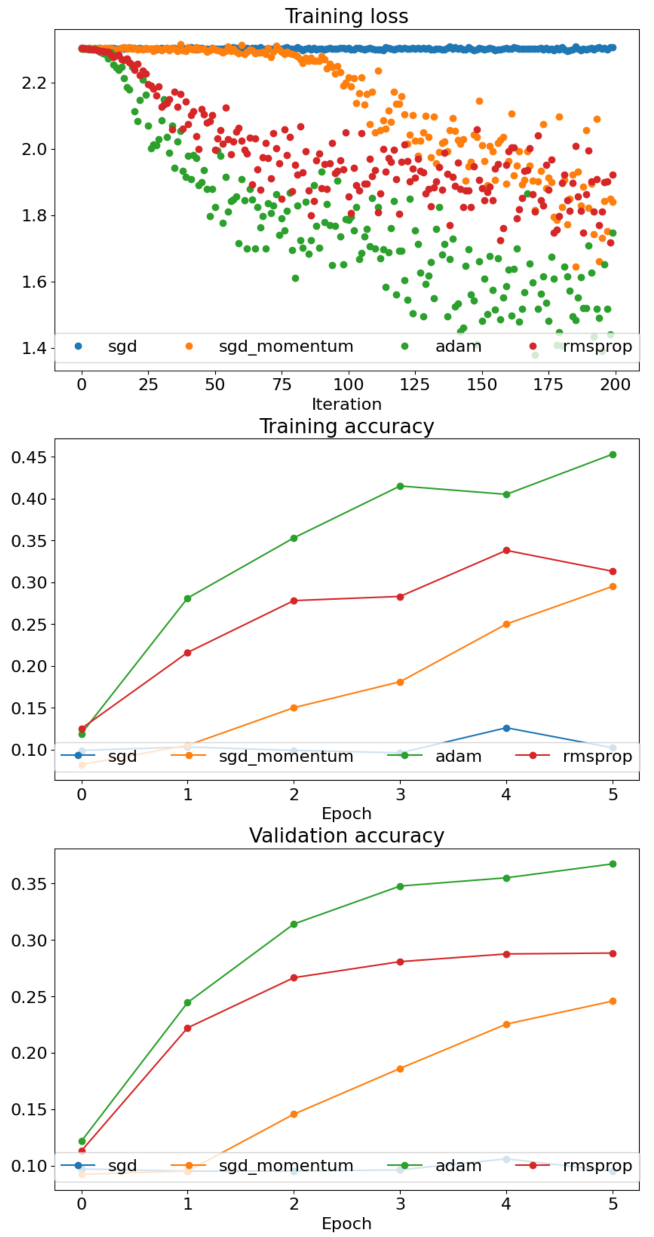
2. 如下圖所示，用 adam 算出來的 w 和 Mean Squared Gradient 和 Root Mean Squared Gradient 跟實際上沒差很多:

```
expected_next_w = torch.tensor([
    [-0.40094747, -0.34836187, -0.29577703, -0.24319299, -0.19060977],
    [-0.1380274,  -0.08544591, -0.03286534,  0.01971428,  0.0722929],
    [ 0.1248705,   0.17744702,  0.23002243,  0.28259667,  0.33516969],
    [ 0.38774145,  0.44031188,  0.49288093,  0.54544852,  0.59801459]],
      dtype=torch.float64, device='cuda')
expected_v = torch.tensor([
    [ 0.69966,     0.68908382,  0.67851319,  0.66794809,  0.65738853,],
    [ 0.64683452,  0.63628604,  0.6257431,   0.61520571,  0.60467385,],
    [ 0.59414753,  0.58362676,  0.57311152,  0.56260183,  0.55209767,],
    [ 0.54159906,  0.53110598,  0.52061845,  0.51013645,  0.49966,   ]],
      dtype=torch.float64, device='cuda')
expected_m = torch.tensor([
    [ 0.48,        0.49947368,  0.51894737,  0.53842105,  0.55789474],
    [ 0.57736842,  0.59684211,  0.61631579,  0.63578947,  0.65526316],
    [ 0.67473684,  0.69421053,  0.71368421,  0.73315789,  0.75263158],
    [ 0.77210526,  0.79157895,  0.81105263,  0.83052632,  0.85      ]],
      dtype=torch.float64, device='cuda')

# You should see relative errors around e-7 or less
print('next_w error: ', usefuns.grad.rel_error(expected_next_w, next_w))
print('v error: ', usefuns.grad.rel_error(expected_v, config['v']))
print('m error: ', usefuns.grad.rel_error(expected_m, config['m']))
```

```
next_w error:  3.756728297598868e-09
v error:  3.4048987160545265e-09
m error:  2.786377729853651e-09
```

3. 如下圖所示，比較 sgd 和 sgd_momentum 和 adam 和 rms prop 之間的準確率，發現 adam>rms prop>sgd momentum>sgd，loss 則是 adam<rms prop<sgd momentum<sgd 並把結果 plot 出來。

```
running with  adam
(Time 0.00 sec; Iteration 1 / 200) loss: 2.302573
(Epoch 0 / 5) train acc: 0.119000; val_acc: 0.121700
(Epoch 1 / 5) train acc: 0.281000; val_acc: 0.244400
(Epoch 2 / 5) train acc: 0.353000; val_acc: 0.313900
(Epoch 3 / 5) train acc: 0.415000; val_acc: 0.347500
(Epoch 4 / 5) train acc: 0.405000; val_acc: 0.354800
(Epoch 5 / 5) train acc: 0.453000; val_acc: 0.367100

running with  rmsprop
(Time 0.00 sec; Iteration 1 / 200) loss: 2.303421
(Epoch 0 / 5) train acc: 0.125000; val_acc: 0.113100
(Epoch 1 / 5) train acc: 0.216000; val_acc: 0.222000
(Epoch 2 / 5) train acc: 0.278000; val_acc: 0.266400
(Epoch 3 / 5) train acc: 0.283000; val_acc: 0.280600
(Epoch 4 / 5) train acc: 0.338000; val_acc: 0.287400
(Epoch 5 / 5) train acc: 0.313000; val_acc: 0.288200
```



4. 如下圖所示，輸出的平均值在訓練和測試期間大致相同。在訓練期間，設為零的輸出數應大致等於 p。而在測試期間，不應有任何輸出被設為零。

```
Running tests with p =  0.25
Mean of input:  9.997330335850453
Mean of train-time output:  10.008798328321282
Mean of test-time output:  9.997330335850453
Fraction of train-time output set to zero:  0.2490839958190918
Fraction of test-time output set to zero:  0.0

Running tests with p =  0.4
Mean of input:  9.997330335850453
Mean of train-time output:  9.977839345169954
Mean of test-time output:  9.997330335850453
Fraction of train-time output set to zero:  0.40116000175476074
Fraction of test-time output set to zero:  0.0

Running tests with p =  0.7
Mean of input:  9.997330335850453
Mean of train-time output:  10.006132149340187
Mean of test-time output:  9.997330335850453
Fraction of train-time output set to zero:  0.6999160051345825
Fraction of test-time output set to zero:  0.0
```

5. 如下圖所示，dropout backward 出來的梯度與實際上的誤差很小。

```
dropout_param = {'mode': 'train', 'p': 0.2, 'seed': 0}
out, cache = Dropout.forward(x, dropout_param)
dx = Dropout.backward(dout, cache)
dx_num = usefuns.grad.compute_numeric_gradient(lambda xx: Dropout.forward(xx, dropout_param)[0], x, dout

# Error should be around e-10 or less
print('dx relative error: ', usefuns.grad.rel_error(dx, dx_num))

dx relative error:  3.914942325636866e-09
```

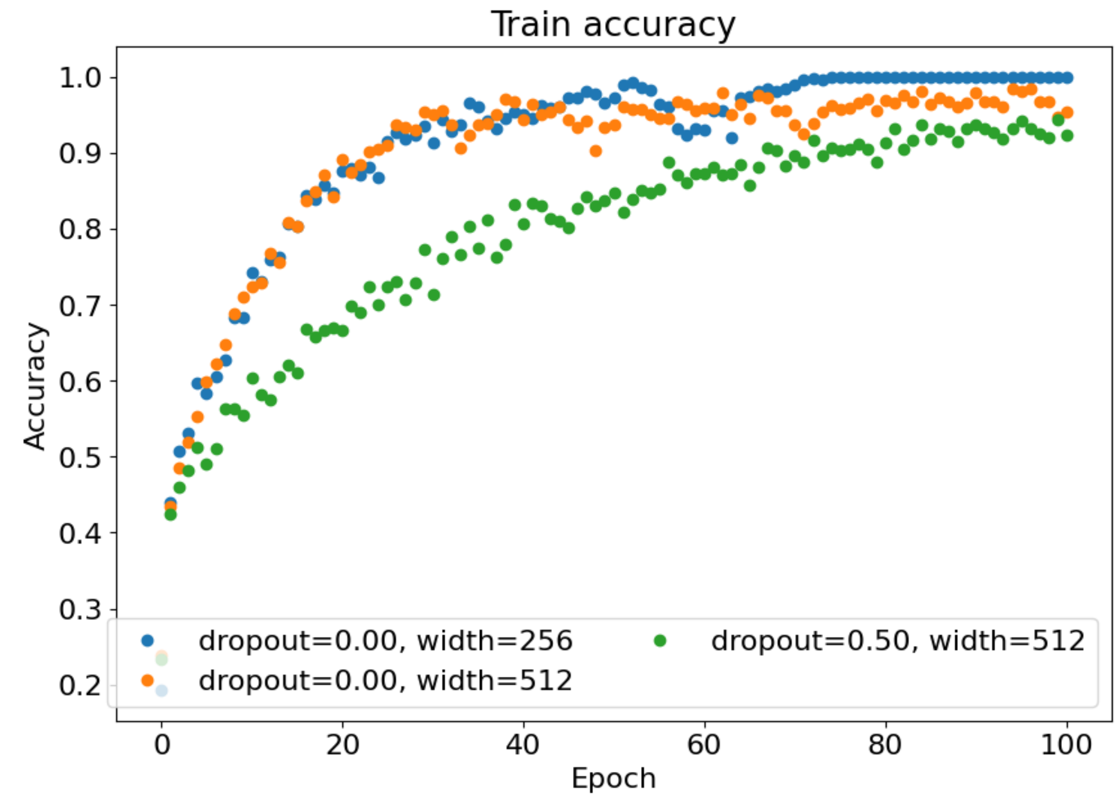
6. 如下圖所示，在 fully connect net 加上 dropout，發現跟實際計算出來的數值誤差很小。

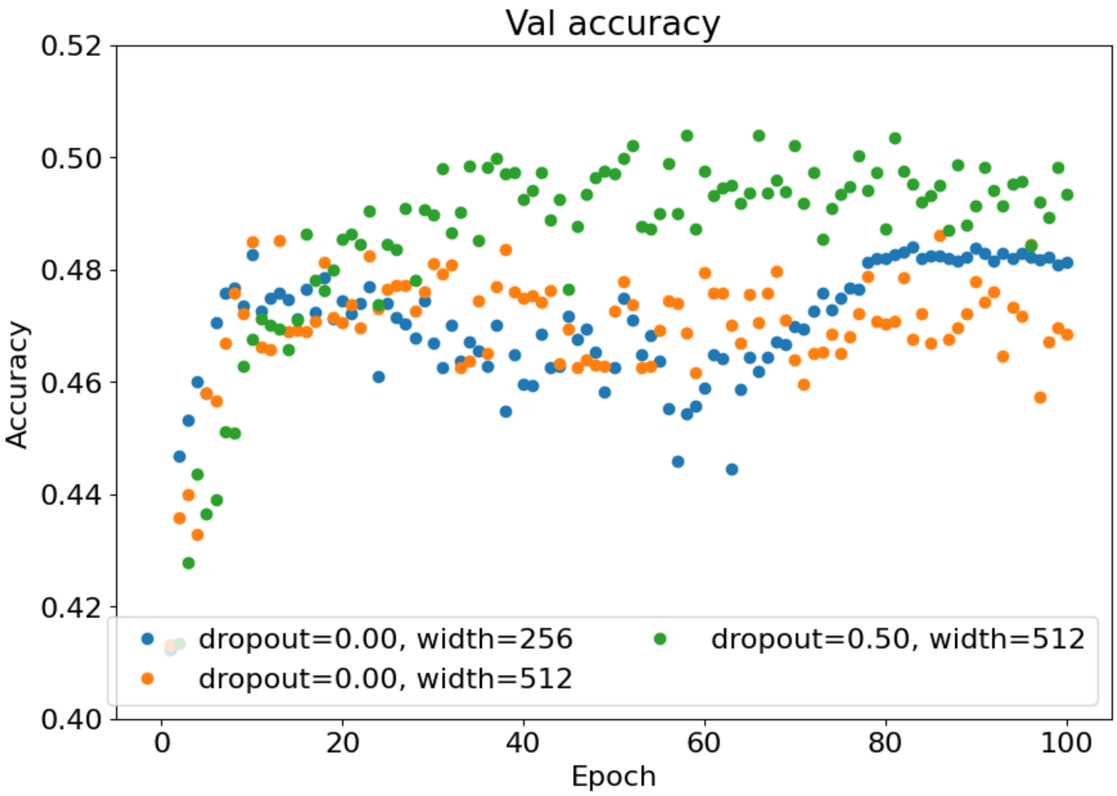
```
Running check with dropout =  0
Initial loss:  2.3053575717037686
W1 relative error: 6.06e-08
W2 relative error: 1.02e-07
W3 relative error: 5.89e-08
b1 relative error: 1.28e-07
b2 relative error: 2.05e-08
b3 relative error: 3.41e-09

Running check with dropout =  0.25
Initial loss:  2.3149601124989854
W1 relative error: 3.64e-08
W2 relative error: 4.60e-08
W3 relative error: 3.14e-08
b1 relative error: 6.83e-08
b2 relative error: 3.17e-08
b3 relative error: 3.63e-09

Running check with dropout =  0.5
Initial loss:  2.3021646399354614
W1 relative error: 7.21e-08
W2 relative error: 2.93e-08
W3 relative error: 2.56e-08
b1 relative error: 5.59e-08
b2 relative error: 2.64e-08
b3 relative error: 2.76e-09
```

6. 如下圖所示，比較不同的 dropout 和 hidden size，發現沒有用 dropout 時 train 的準確率高但 validation 的準確率低。用 dropout 時，train 的準確率雖然比沒有用 dropout 低，但 validation 的準確率卻比沒有用 dropout 高，可以降低 overfitting。

```
Training a model with dropout=0.00 and width=256
(Time 0.01 sec; Iteration 1 / 3900) loss: 2.304467
(Epoch 0 / 100) train acc: 0.193000; val_acc: 0.198200
(Epoch 10 / 100) train acc: 0.742000; val_acc: 0.482600
(Epoch 20 / 100) train acc: 0.876000; val_acc: 0.474400
(Epoch 30 / 100) train acc: 0.913000; val_acc: 0.467000
(Epoch 40 / 100) train acc: 0.951000; val_acc: 0.459700
(Epoch 50 / 100) train acc: 0.973000; val_acc: 0.462600
(Epoch 60 / 100) train acc: 0.930000; val_acc: 0.458900
(Epoch 70 / 100) train acc: 0.989000; val_acc: 0.469900
(Epoch 80 / 100) train acc: 1.000000; val_acc: 0.481900
(Epoch 90 / 100) train acc: 1.000000; val_acc: 0.483900
(Epoch 100 / 100) train acc: 1.000000; val_acc: 0.481300

Training a model with dropout=0.00 and width=512
(Time 0.00 sec; Iteration 1 / 3900) loss: 2.302387
(Epoch 0 / 100) train acc: 0.239000; val_acc: 0.220000
(Epoch 10 / 100) train acc: 0.723000; val_acc: 0.484900
(Epoch 20 / 100) train acc: 0.891000; val_acc: 0.470500
(Epoch 30 / 100) train acc: 0.951000; val_acc: 0.481100
(Epoch 40 / 100) train acc: 0.944000; val_acc: 0.475000
(Epoch 50 / 100) train acc: 0.937000; val_acc: 0.472700
(Epoch 60 / 100) train acc: 0.958000; val_acc: 0.479500
(Epoch 70 / 100) train acc: 0.937000; val_acc: 0.463900
(Epoch 80 / 100) train acc: 0.969000; val_acc: 0.470400
(Epoch 90 / 100) train acc: 0.979000; val_acc: 0.477900
(Epoch 100 / 100) train acc: 0.954000; val_acc: 0.468400
```

```
Training a model with dropout=0.50 and width=512
(Time 0.00 sec; Iteration 1 / 3900) loss: 2.306579
(Epoch 0 / 100) train acc: 0.233000; val_acc: 0.227000
(Epoch 10 / 100) train acc: 0.603000; val_acc: 0.467600
(Epoch 20 / 100) train acc: 0.666000; val_acc: 0.485400
(Epoch 30 / 100) train acc: 0.714000; val_acc: 0.489700
(Epoch 40 / 100) train acc: 0.807000; val_acc: 0.492400
(Epoch 50 / 100) train acc: 0.847000; val_acc: 0.497000
(Epoch 60 / 100) train acc: 0.873000; val_acc: 0.497500
(Epoch 70 / 100) train acc: 0.896000; val_acc: 0.502000
(Epoch 80 / 100) train acc: 0.913000; val_acc: 0.487300
(Epoch 90 / 100) train acc: 0.936000; val_acc: 0.491300
(Epoch 100 / 100) train acc: 0.923000; val_acc: 0.493500
```


Train accuracy
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
Accuracy
0
20
40
60
80
100
Epoch
dropout=0.00, width=256
dropout=0.00, width=512
dropout=0.50, width=512


Val accuracy
0.52
0.50
0.48
0.46
0.44
0.42
0.40
Accuracy
0
20
40
60
80
100
Epoch
dropout=0.00, width=256
dropout=0.00, width=512
dropout=0.50, width=512