Assignment 3 Part1 Summary

- 实现了任意层数的全连接神经网络的自动微分框架,并且实现了常见的一阶优化器

Regularization

$$L(W) = \underbrace{\frac{1}{N}\sum_{i=1}^{N} L_i(f(x_i, W), y_i)}_{\text{Data loss}} + \underbrace{\lambda R(W)}_{\text{Regularization}}$$

λ is a hyperparameter giving regularization strength

Data loss: Model predictions should match training data

Regularization: Prevent the model from doing *too* well on training data

- L1
- L2
- Dropout
- Batch normalization
- So on.

AutoGrad Framework for fc layers

整体架构:

(L-1){linear-ReLu} {Linear-Softmax}

L-1层 Linear-ReLu 最后加上一层 Linear - softmax.

Linear Layer

$$Y = XW + b \tag{1}$$

- Forward:计算出输出,并且返回当前的 X, W, b 以供之后计算梯度使用.
- Backward:将upstream Grad 与local Grad 相乘形成downstream Grad with respect to W,X and b 传播给上一层. 求导见之前笔记.

ReLu

$$Y = max(X, 0) \tag{2}$$

- Forward: 计算Y,并且返回X以供计算梯度使用.
- Backward: 计算梯度,根据X取值正负决定梯度是否向前传播.

Linear-ReLu

- 将两层合并成一个模块,减少代码量.

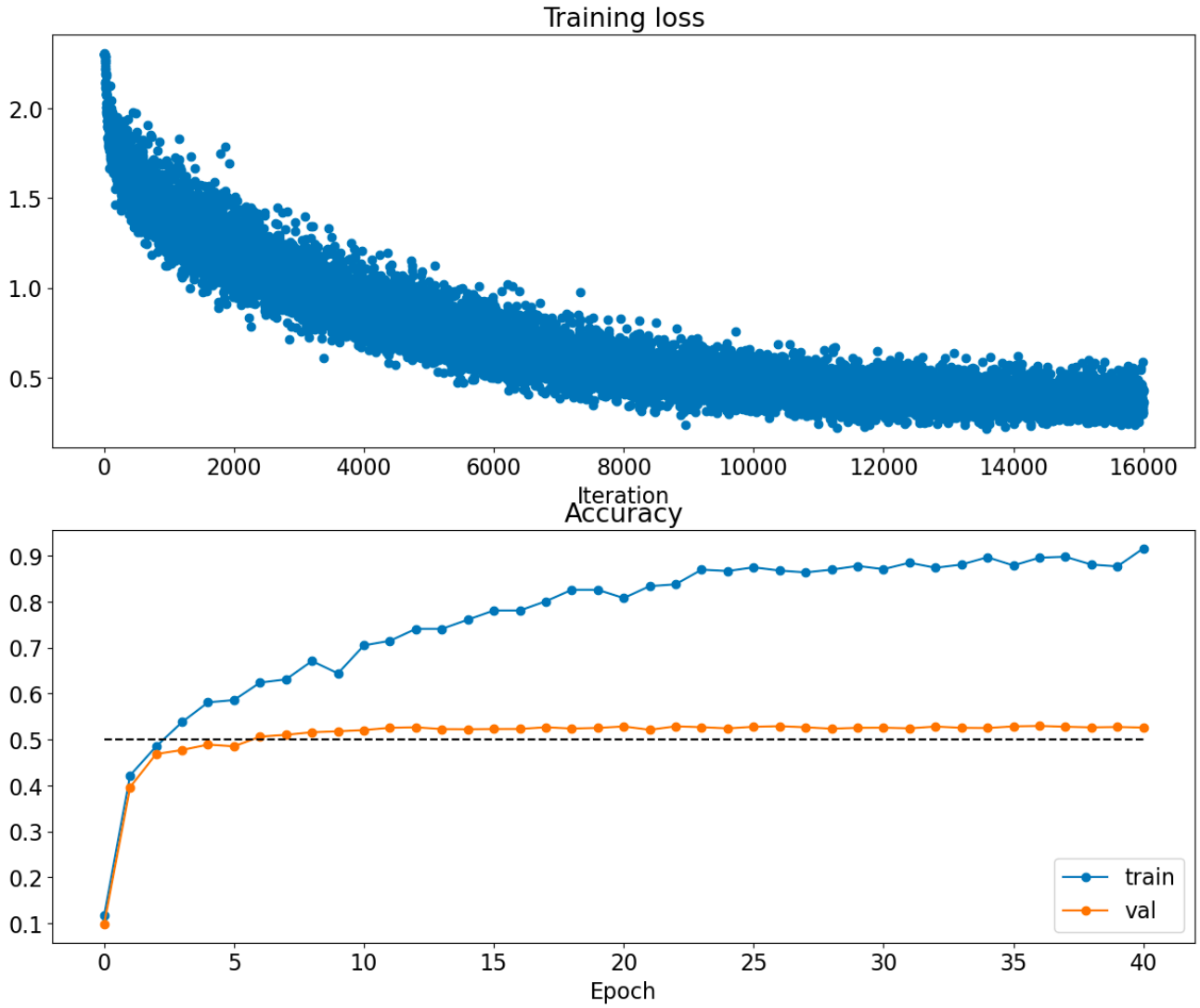
Softmax

- 计算cross- entropy,并且返回对输入的梯度,求导见之前笔记.

实验:

```
Testing initialization ...
Testing test-time forward pass ...
Testing training loss (no regularization)
Running numeric gradient check with reg =  0.0
W1 relative error: 2.94e-07
W2 relative error: 1.65e-09
b1 relative error: 1.01e-06
b2 relative error: 4.63e-09
Running numeric gradient check with reg =  0.7
W1 relative error: 2.70e-08
W2 relative error: 9.86e-09
b1 relative error: 2.28e-06
b2 relative error: 2.90e-08
```

计算出的解析梯度以及数值梯度的误差.



三层网络,使用SGD在cifar-10上得到如下结果

Optimizer

SGD

Stochastic Gradient Descent (SGD)

$$L(W) = \frac{1}{N}\sum_{i=1}^{N} L_i(x_i, y_i, W) + \lambda R(W)$$

$$\nabla_W L(W) = \frac{1}{N}\sum_{i=1}^{N} \nabla_W L_i(x_i, y_i, W) + \lambda \nabla_W R(W)$$

Full sum expensive when N is large!

Approximate sum using a **minibatch** of examples 32 / 64 / 128 common

```
# Stochastic gradient descent
w = initialize_weights()
for t in range(num_steps):
  minibatch = sample_data(data, batch_size)
  dw = compute_gradient(loss_fn, minibatch, w)
  w -= learning_rate * dw
```

Hyperparameters:
- Weight initialization
- Number of steps
- Learning rate
- Batch size
- Data sampling

Stochastic Gradient Descent (SGD)

$$L(W) = \mathbb{E}_{(x,y)\sim p_{data}}[L(x, y, W)] + \lambda R(W)$$
$$\approx \frac{1}{N}\sum_{i=1}^{N} L(x_i, y_i, W) + \lambda R(W)$$

Think of loss as an expectation over the full **data distribution** p_{data}

Approximate expectation via sampling

$$\nabla_W L(W) = \nabla_W \mathbb{E}_{(x,y)\sim p_{data}}[L(x, y, W)] + \lambda \nabla_W R(W)$$
$$\approx \sum_{i=1}^{N} \nabla_w L_W(x_i, y_i, W) + \nabla_w R(W)$$

对整个Batch梯度的蒙特卡洛估计

$$x_{t+1} = x_t - \alpha \nabla f(x_t)$$

```
for t in range(num_steps):
  dw = compute_gradient(w)
  w -= learning_rate * dw
```

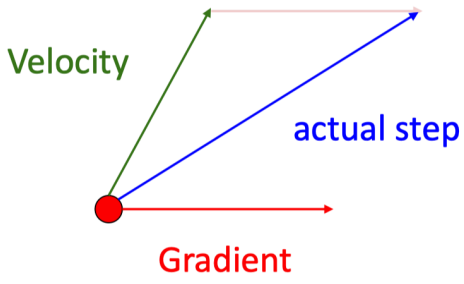
SGD问题:

- 损失函数有较高的条件数,在不同的方向上的学习步长可能会震荡或者几乎不变
- 局部极小值或者鞍点处梯度为0,几乎不训练.
- minibatch can be noisy.

SGD + Momentum

将优化过程看作是小球从山顶滚下山,在下山的过程中会积累速度.使得学习率发生改变. **对梯度在时间上积分**

Momentum update:



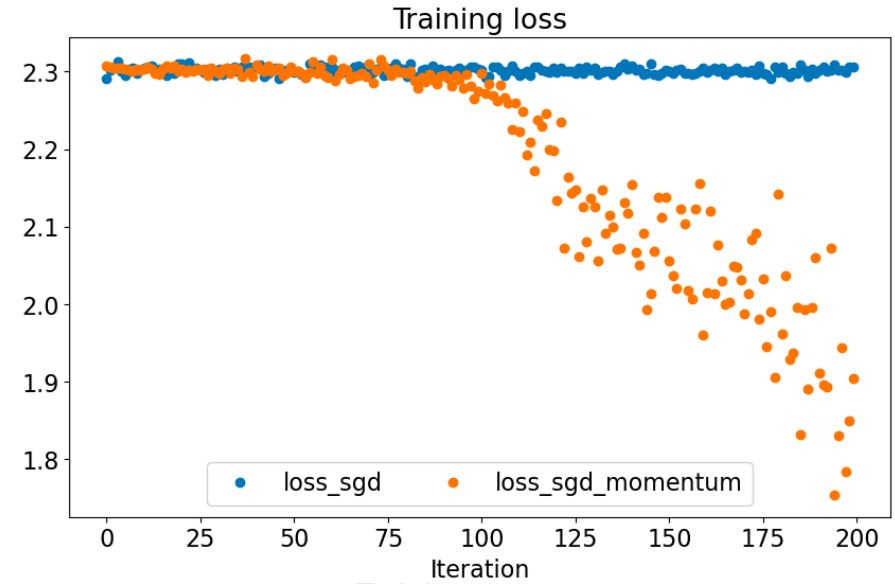
Combine gradient at current point with velocity to get step used to update weights

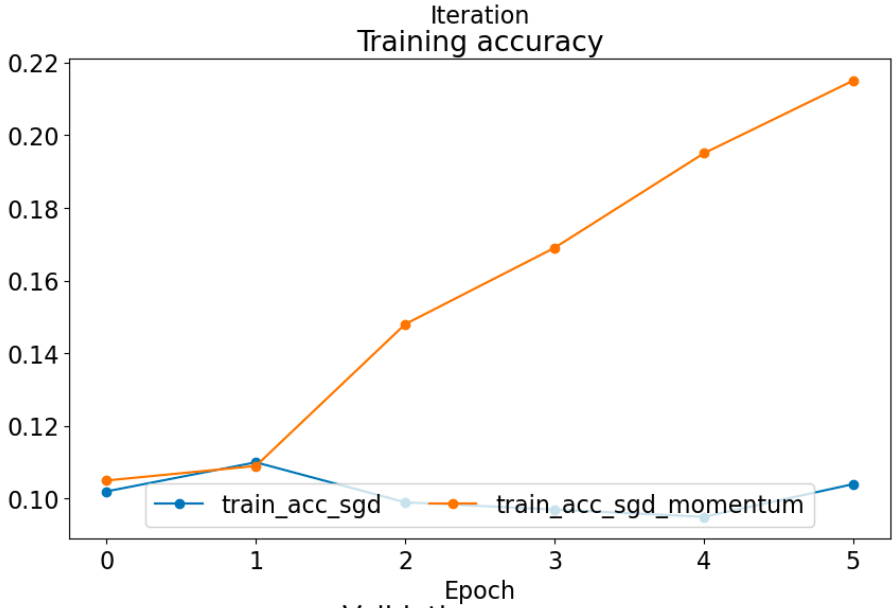
SGD+Momentum

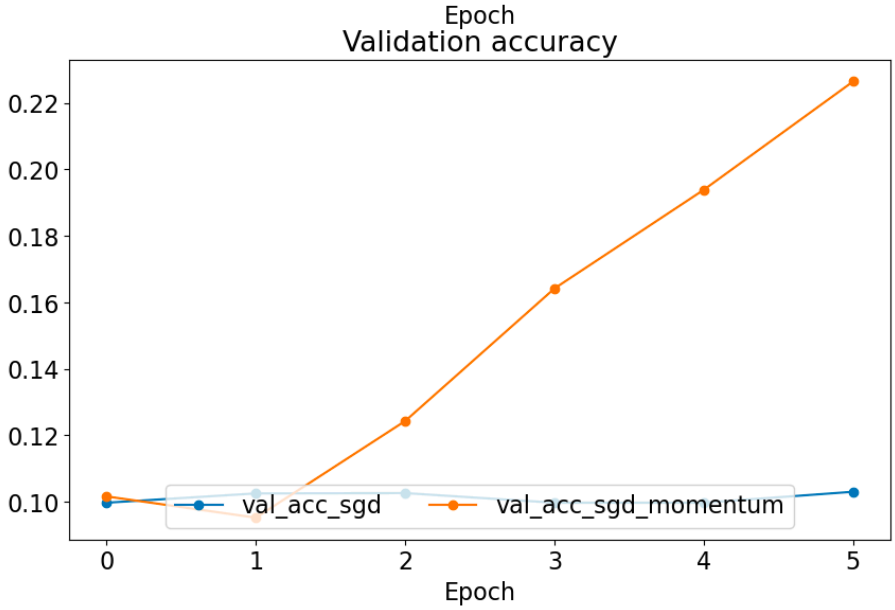
$$v_{t+1} = \rho v_t + \nabla f(x_t)$$
$$x_{t+1} = x_t - \alpha v_{t+1}$$

```
v = 0
for t in range(num_steps):
  dw = compute_gradient(w)
  v = rho * v + dw
  w -= learning_rate * v
```

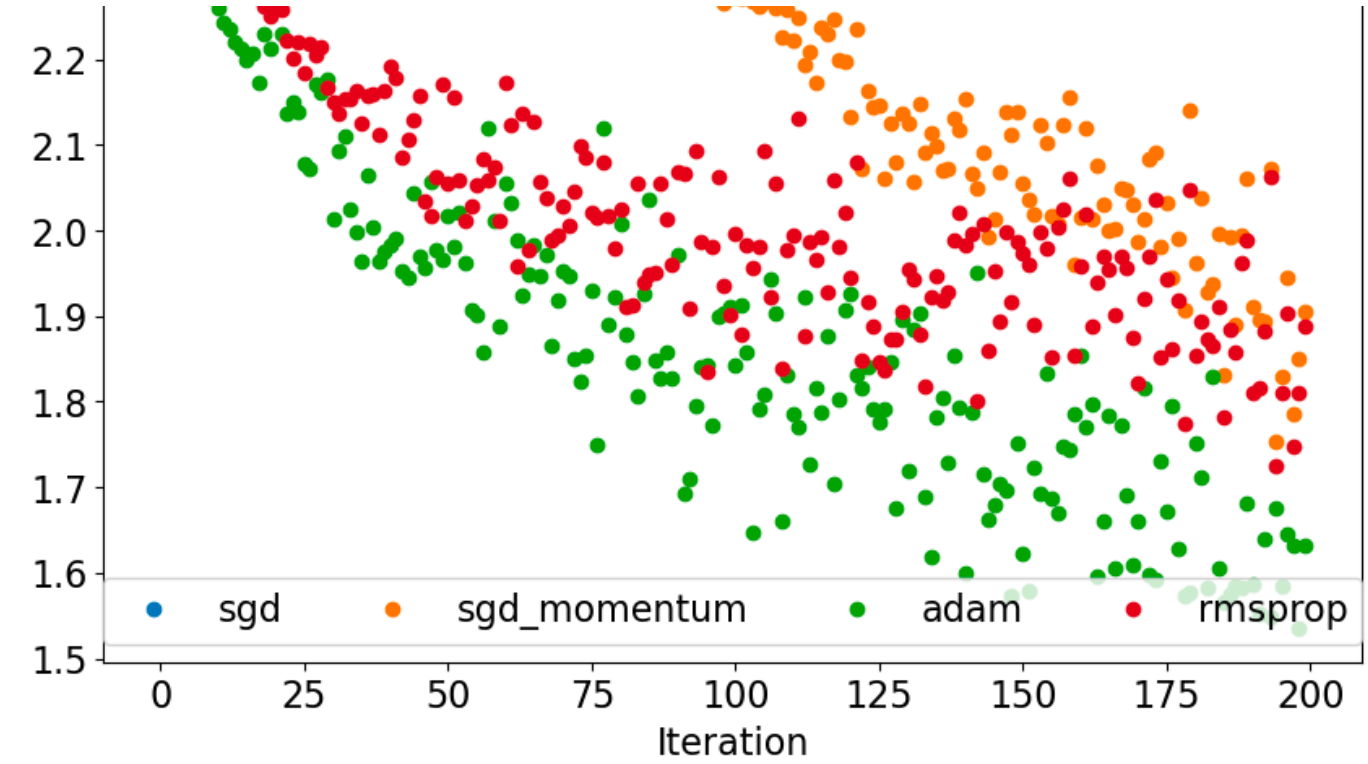
- Build up “velocity” as a running mean of gradients
- Rho gives “friction”; typically rho=0.9 or 0.99



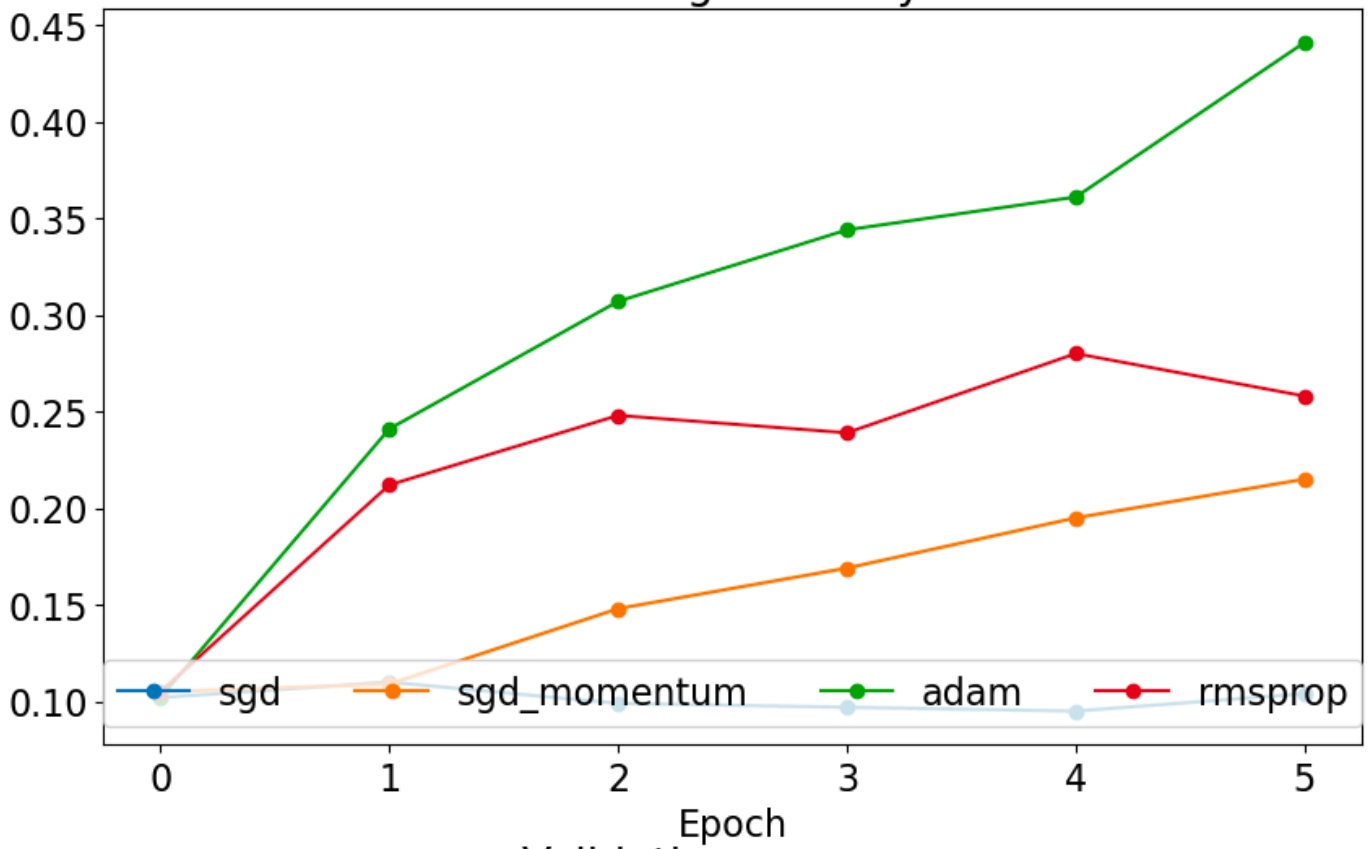




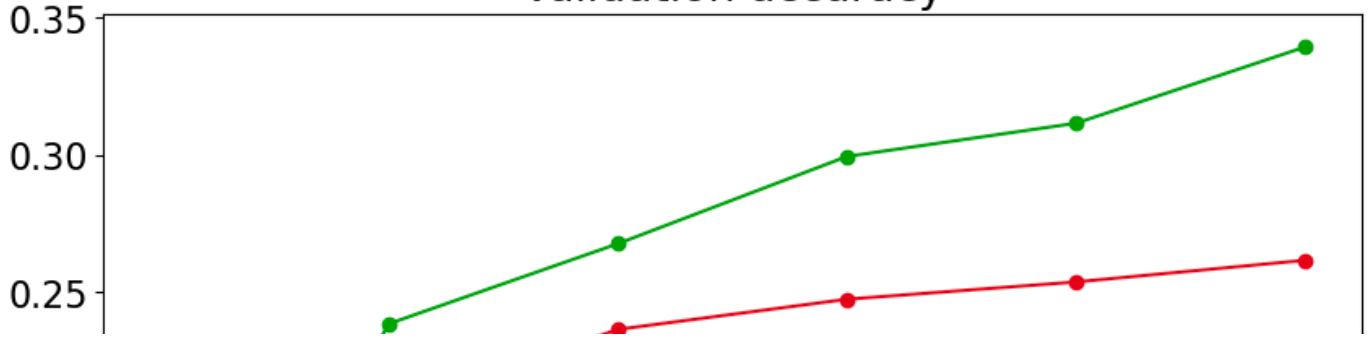
在CIFAR-10数据集上的训练结果.可以看到比SGD取得了很大进步.

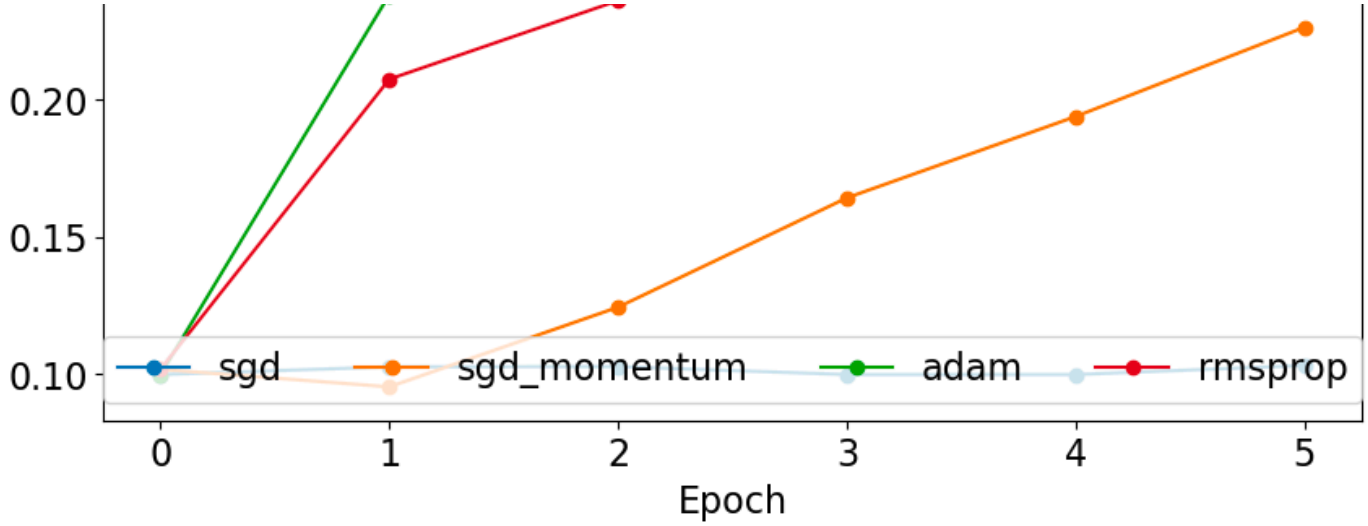


Training accuracy

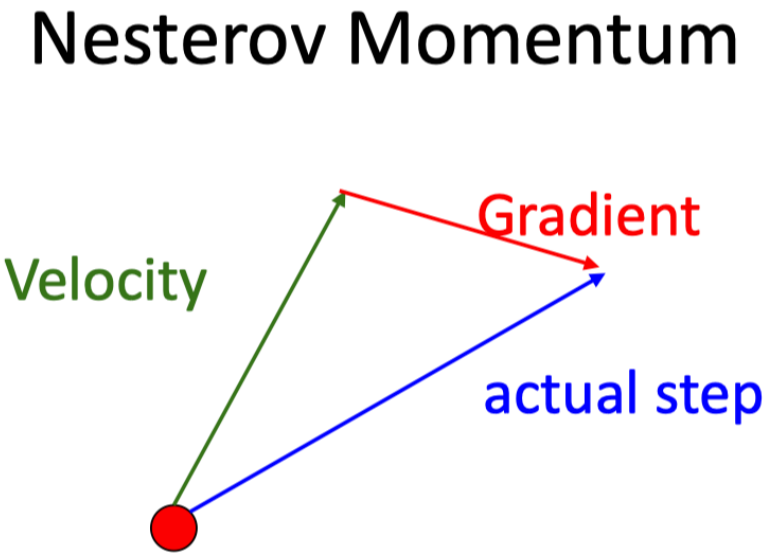


Validation accuracy





Nestrov Momentum



"Look ahead" to the point where updating using velocity would take us; compute gradient there and mix it with velocity to get actual update direction

$$v_{t+1} = \rho v_t - \alpha \nabla f(x_t + \rho v_t)$$

$$x_{t+1} = x_t + v_{t+1}$$

AdaGrad

```
grad_squared = 0
for t in range(num_steps):
  dw = compute_gradient(w)
  grad_squared += dw * dw
  w -= learning_rate * dw / (grad_squared.sqrt() + 1e-7)
```

Added element-wise scaling of the gradient based on the historical sum of squares in each dimension

"Per-parameter learning rates"
or "adaptive learning rates"

- 自适应学习率
- 当dw很大时,通过处以平方和,适当减少学习率
- 反之适当增大.
- 问题:grad_squared 一直上升 造成问题

RMSProp: Leaky Adagrad

```
grad_squared = 0
for t in range(num_steps):
  dw = compute_gradient(w)
  grad_squared += dw * dw
  w -= learning_rate * dw / (grad_squared.sqrt() + 1e-7)
```

AdaGrad

```
grad_squared = 0
for t in range(num_steps):
  dw = compute_gradient(w)
  grad_squared = decay_rate * grad_squared + (1 - decay_rate) * dw * dw
  w -= learning_rate * dw / (grad_squared.sqrt() + 1e-7)
```

RMSProp

Adam: RMSProp + Momentum

Adam (almost): RMSProp + Momentum

```
moment1 = 0
moment2 = 0
for t in range(1, num_steps + 1):  # Start at t = 1
  dw = compute_gradient(w)
  moment1 = beta1 * moment1 + (1 - beta1) * dw
  moment2 = beta2 * moment2 + (1 - beta2) * dw * dw
  w -= learning_rate * moment1 / (moment2.sqrt() + 1e-7)
```

Adam

Momentum

AdaGrad / RMSProp

```
grad_squared = 0
for t in range(num_steps):
  dw = compute_gradient(w)
  grad_squared = decay_rate * grad_squared + (1 - decay_rate) * dw * dw
  w -= learning_rate * dw / (grad_squared.sqrt() + 1e-7)
```

RMSProp

问题: t=1时,beta2 = 0.999时, 使得moment2很小, 从而使得步长很大.

- 引入bias correction

```
moment1 = 0
moment2 = 0
for t in range(1, num_steps + 1):  # Start at t = 1
  dw = compute_gradient(w)
  moment1 = beta1 * moment1 + (1 - beta1) * dw
  moment2 = beta2 * moment2 + (1 - beta2) * dw * dw
  moment1_unbias = moment1 / (1 - beta1 ** t)
  moment2_unbias = moment2 / (1 - beta2 ** t)
  w -= learning_rate * moment1_unbias / (moment2_unbias.sqrt() + 1e-7)
```

Momentum

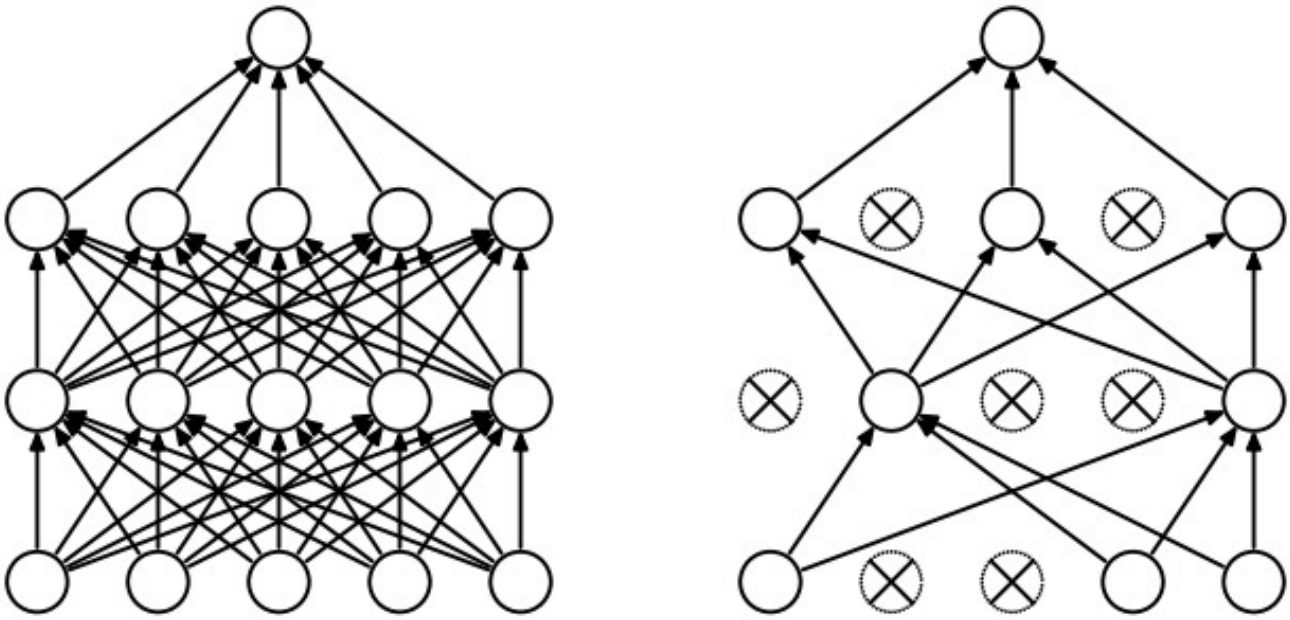
AdaGrad / RMSProp

Bias correction

Bias correction for the fact that first and second moment estimates start at zero

drop out

按照概率p,随机静默某一节点,类似大脑的代偿功能.



(a) Standard Neural Net (b) After applying dropout.

```
""" Vanilla Dropout: Not recommended implementation (see notes below) """

p = 0.5 # probability of keeping a unit active. higher = less dropout

def train_step(X):
  """ X contains the data """

  # forward pass for example 3-layer neural network
  H1 = np.maximum(0, np.dot(W1, X) + b1)
  U1 = np.random.rand(*H1.shape) < p # first dropout mask
```

```
  H1 *= U1 # drop!
  H2 = np.maximum(0, np.dot(W2, H1) + b2)
  U2 = np.random.rand(*H2.shape) < p # second dropout mask
  H2 *= U2 # drop!
  out = np.dot(W3, H2) + b3

  # backward pass: compute gradients... (not shown)
  # perform parameter update... (not shown)

def predict(X):
  # ensembled forward pass
  H1 = np.maximum(0, np.dot(W1, X) + b1) * p # NOTE: scale the activations
  H2 = np.maximum(0, np.dot(W2, H1) + b2) * p # NOTE: scale the activations
  out = np.dot(W3, H2) + b3
```

In the code above, inside the `train_step` function we have performed dropout twice: on the first hidden layer and on the second hidden layer. It is also possible to perform dropout right on the input layer, in which case we would also create a binary mask for the input `X`. The backward pass remains unchanged, but of course has to take into account the generated masks `U1,U2`.

Crucially, note that in the `predict` function we are not dropping anymore, but we are performing a scaling of both hidden layer outputs by $\mathbf{p}$. This is important because at test time all neurons see all their inputs, so we want the outputs of neurons at test time to be identical to their expected outputs at training time. For example, in case of $\mathbf{p = 0.5}$, the neurons must halve their outputs at test time to have the same output as they had during training time (in expectation). To see this, consider an output of a neuron $\mathbf{x}$ (before dropout). With dropout, the expected output from this neuron will become $\mathbf{px + (1 - p)0}$, because the neuron's output will be set to zero with probability $\mathbf{1 - p}$. At test time, when we keep the neuron always active, we must adjust $\mathbf{x \rightarrow px}$ to keep the same expected output. It can also be shown that performing this attenuation at test time can be related to the process of iterating over all the possible binary masks (and therefore all the exponentially many sub-networks) and computing their ensemble prediction.

The undesirable property of the scheme presented above is that we must scale the activations by $\mathbf{p}$ at test time. Since test-time performance is so critical, it is always preferable to use **inverted dropout**, which performs the scaling at train time, leaving the forward pass at test time untouched. Additionally, this has the appealing property that the prediction code can remain untouched when you decide to tweak where you apply dropout, or if at all. Inverted dropout looks as follows:

```
"""
Inverted Dropout: Recommended implementation example.
We drop and scale at train time and don't do anything at test time.
"""
```

```
p = 0.5 # probability of keeping a unit active. higher = less dropout

def train_step(X):
  # forward pass for example 3-layer neural network
  H1 = np.maximum(0, np.dot(W1, X) + b1)
  U1 = (np.random.rand(*H1.shape) < p) / p # first dropout mask. Notice /p!
  H1 *= U1 # drop!
  H2 = np.maximum(0, np.dot(W2, H1) + b2)
  U2 = (np.random.rand(*H2.shape) < p) / p # second dropout mask. Notice /p!
  H2 *= U2 # drop!
  out = np.dot(W3, H2) + b3

  # backward pass: compute gradients... (not shown)
  # perform parameter update... (not shown)

def predict(X):
  # ensembled forward pass
  H1 = np.maximum(0, np.dot(W1, X) + b1) # no scaling necessary
  H2 = np.maximum(0, np.dot(W2, H1) + b2)
  out = np.dot(W3, H2) + b3
```