Homework 2

Yuanteng Chen

1. Why Learning Rates Cannot be Too Big.

(a) For what values of learning rate $\eta > 0$ is the recurrence (3) stable?

Solution:

$$W_{t+1} = (1-2\eta\sigma^2)W_t + 2\eta\sigma y$$

$$2\eta\sigma y = 2\eta\sigma^2 \cdot \frac{y}{\sigma}$$

$$\therefore \quad W_{t+1} = (1-2\eta\sigma^2)(W_t - \frac{y}{\sigma}) + \frac{y}{\sigma}$$

$$W_1 = (1-2\eta\sigma^2)(W_0 - \frac{y}{\sigma}) + \frac{y}{\sigma}$$

$$W_2 = (1-2\eta\sigma^2)(W_1 - \frac{y}{\sigma}) + \frac{y}{\sigma}$$

$$= (1-2\eta\sigma^2)[(1-2\eta\sigma^2)(W_0 - \frac{y}{\sigma}) + \frac{y}{\sigma} - \frac{y}{\sigma}] + \frac{y}{\sigma}$$

$$= (1-2\eta\sigma^2)^2(W_0 - \frac{y}{\sigma}) + \frac{y}{\sigma}$$

$$\vdots$$

$$W_{t+1} = (1-2\eta\sigma^2)^{t+1}(W_0 - \frac{y}{\sigma}) + \frac{y}{\sigma}$$

$|1-2\eta\sigma^2| < 1 \iff$ recurrence is stable

$$\therefore \quad 0 < \eta < \frac{1}{\sigma^2}$$

(b). get within a factor $(1-\varepsilon)$ of w^*

$$\Downarrow$$

$$|W_t - w^*| < \varepsilon |w^*|$$

$\because$ $W_{t+1} = (1-2\eta\sigma^2)^{t+1}(W_0 - \frac{y}{\sigma}) + \frac{y}{\sigma}$

$W^* = \frac{y}{\sigma}$

$|W_t - \frac{y}{\sigma}| < \varepsilon |\frac{y}{\sigma}|$

$W_t - \frac{y}{\sigma} = (1-2\eta\sigma^2)^t (W_0 - \frac{y}{\sigma})$

$\because$ initial condition $W_0 = 0$

$\therefore$ $|(1-2\eta\sigma^2)^t(-\frac{y}{\sigma})| < \varepsilon |\frac{y}{\sigma}|$

$\therefore$ $(1-2\eta\sigma^2)^t < \varepsilon$

$\log(1-2\eta\sigma^2)^t < \log\varepsilon$

$\because$ $\log(1-2\eta\sigma^2) < 0$

$\therefore$ $t > \frac{\log\varepsilon}{\log(1-2\eta\sigma^2)}$

(c) $$\begin{bmatrix} \sigma_L & 0 \\ 0 & \sigma_s \end{bmatrix} \begin{bmatrix} W[1] \\ W[2] \end{bmatrix} = \begin{bmatrix} y[1] \\ y[2] \end{bmatrix}$$

$\sigma_L \gg \sigma_s$

initial condition $w=0$, a single learning rate η

Solution: assume $\Sigma = \begin{bmatrix} \sigma_L & 0 \\ 0 & \sigma_s \end{bmatrix}$

same as (d):

$W_{t+1} = (E - 2\eta\Sigma^2)W_t + 2\eta\Sigma y$

$\Sigma^2 = \begin{bmatrix} \sigma_L^2, & 0 \\ 0, & \sigma_s^2 \end{bmatrix}$ $E - 2\eta\Sigma^2 = \begin{bmatrix} 1-2\eta\sigma_L^2 & 0 \\ 0 & 1-2\eta\sigma_s^2 \end{bmatrix}$

$\therefore \begin{cases} |1-2\eta\sigma_l^2| < 1 \\ |1-2\eta\sigma_s^2| < 1 \end{cases}$ $\because \sigma_l >> \sigma_s$ $\therefore \eta < \frac{1}{\sigma_l^2} < \frac{1}{\sigma_s^2}$

(d) depending on η, σ_l, σ_s, which of the two dimensions is converging faster and which one is converging slower?

Solution: in (c) $W_{t+1} = (1-2\eta\sigma^2)^{t+1}(W_0 - \frac{y}{\sigma}) + \frac{y}{\sigma}$

$$\therefore \begin{cases} W[1]_{t+1} = (1-2\eta\sigma_l^2)^{t+1}(W_0 - \frac{y[1]}{\sigma_l}) + \frac{y[1]}{\sigma_l} \\ W[2]_{t+1} = (1-2\eta\sigma_s^2)^{t+1}(W_0 - \frac{y[2]}{\sigma_s}) + \frac{y[2]}{\sigma_s} \end{cases}$$

if $|1-2\eta\sigma_l^2| < |1-2\eta\sigma_s^2| < 1$
then $W[1]$ is converging faster,
else $W[2]$ is converging faster.

(e) when $|1-2\eta\sigma_l^2| = |1-2\eta\sigma_s^2|$, we get the fastest overall convergence to the solution.

$\because \sigma_l >> \sigma_s$ $\therefore$ $2 = 2\eta(\sigma_l^2 + \sigma_s^2)$

$$\eta = \frac{1}{\sigma_l^2 + \sigma_s^2}$$

(f): the speed of converging of σ_i $(i \neq l, s)$

is between σ_l and σ_s,

so they will not influence the choice of possible learning rates.

(g)
I have no idea ......

2. Accelerating Gradient Descent with Momentum.

$$L(w) = \|y - Xw\|_2^2$$

$$w_{t+1} = w_t - \eta z_{t+1}$$

$$z_{t+1} = (1-\beta) z_t + \beta g_t$$

the gradient descent update:

$$w_{t+1} = (I - 2\eta (X^T X)) w_t + 2\eta X^T y$$

$$w^* = (X^T X)^{-1} X^T y$$

(18) $w_{t+1} = w_t - \eta z_{t+1}$

$$z_{t+1} = (1-\beta) z_t + \beta (2X^T X w_t - 2X^T y)$$

$$x_t = V^T (w_t - w^*)$$

$$a_t = V^T z_t$$

(a) ① $w_{t+1} = w_t - \eta z_{t+1}$

$$V^T w_{t+1} = V^T w_t - \eta V^T z_{t+1}$$

$$V^T w_{t+1}[i] = V^T w_t[i] - \eta V^T z_{t+1}[i]$$

$\because\ x_t = v^T(w_t - w^*)$

$\therefore\ v^T w_{t+1} = x_{t+1} + v^T \cdot w^*$

$v^T w_t = x_t + v^T \cdot w^*$

$\therefore\ x_{t+1} + v^T \cdot w^* = x_t + v^T \cdot w^* - \eta v^T z_{t+1}$

$x_{t+1} = x_i - \eta v^T z_{t+1}$

$\because\ a_t = v^T z_t$

$\therefore\ x_{t+1} = x_t - \eta a_{t+1}$

$x_{t+1}[i] = x_t[i] - \eta a_{t+1}[i]$

② $z_{t+1} = (1-\beta) z_t + \beta(2X^T X w_t - 2X^T y)$

$v^T z_{t+1} = (1-\beta) v^T z_t + \beta v^T (2X^T X w_t - 2X^T y)$

$a_{t+1} = (1-\beta) a_t + \beta v^T (2V\Sigma U^T U \Sigma v^T w_t - 2X^T y)$

$a_{t+1} = (1-\beta) a_t + \beta v^T (2V\Sigma^2 V^T w_t - 2X^T y)$

$a_{t+1} = (1-\beta) a_t + \beta(2v^T V \Sigma^2 v^T w_t - 2v^T X^T y)$

$a_{t+1} = (1-\beta) a_t + \beta(2\Sigma^2 v^T w_t - 2v^T X^T y)$

$\because\ x_t = v^T(w_t - w^*)$

$V x_t = w_t - w^*$ $\quad \therefore w_t = V x_t + w^*$

$\therefore\ a_{t+1} = (1-\beta) a_t + \beta(2\Sigma^2 v^T (V x_t + w^*) - 2v^T X^T y)$

$a_{t+1} = (1-\beta) a_t + \beta(2\Sigma^2 x_t + 2\Sigma^2 v^T w^* - 2v^T X^T y)$

$$\because w^* = (X^TX)^{-1}X^Ty$$
$$= (V\Sigma^2V^T)^{-1}X^Ty$$
$$= V\Sigma^{-2}V^TX^Ty$$

plug w^* into :

$$a_{t+1} = (1-\beta)a_t + \beta(2\Sigma^2x_t + 2\Sigma^2V^TV\Sigma^{-2}V^TX^Ty - 2V^TX^Ty)$$

$$a_{t+1} = (1-\beta)a_t + \beta(2\Sigma^2x_t + 2V^TX^Ty - 2V^TX^Ty)$$
$$a_{t+1} = (1-\beta)a_t + 2\beta\Sigma^2x_t$$
$$\Downarrow$$
$$a_{t+1}[i] = (1-\beta)a_t[i] + 2\beta\sigma_i^2x_t[i]$$

$$\therefore \begin{cases} w_{t+1} = w_t - \eta z_{t+1} \\ z_{t+1} = (1-\beta)z_t + \beta(2X^TXw_t - 2X^Ty) \end{cases}$$
$$\Downarrow$$
$$\begin{cases} a_{t+1}[i] = (1-\beta)a_t[i] + 2\beta\sigma_i^2x_t[i] \\ x_{t+1}[i] = x_t[i] - \eta a_{t+1}[i] \end{cases}$$

(b) $$\begin{bmatrix} a_{t+1}[i] \\ x_{t+1}[i] \end{bmatrix} = R_i \begin{bmatrix} a_t[i] \\ x_t[i] \end{bmatrix}$$ Derive R_i

$$\because \begin{cases} a_{t+1}[i] = (1-\beta)a_t[i] + 2\beta\sigma_i^2x_t[i] \\ x_{t+1}[i] = x_t[i] - \eta a_{t+1}[i] \end{cases}$$

$$x_{t+1}[i] = x_t[i] - \eta a_{t+1}[i]$$

$$= x_t[i] - \eta(1-\beta)a_t[i] - 2\eta\beta\sigma_i^2 x_t[i]$$

$$= (1-2\eta\beta\sigma_i^2)x_t[i] - \eta(1-\beta)a_t[i]$$

$$\therefore \begin{cases} a_{t+1}[i] = (1-\beta)a_t[i] + 2\beta\sigma_i^2 x_t[i] \\ x_{t+1}[i] = (1-2\eta\beta\sigma_i^2)x_t[i] - \eta(1-\beta)a_t[i] \end{cases}$$

$$\therefore R_i = \begin{bmatrix} 1-\beta, & 2\beta\sigma_i^2 \\ \eta(\beta-1), & 1-2\eta\beta\sigma_i^2 \end{bmatrix}$$

c( )

$$R_i \vec{x} = \lambda \vec{x}$$

$$(R_i - \lambda E)\vec{x} = 0$$

$$|R_i - \lambda E| = 0$$

$$\begin{vmatrix} 1-\beta-\lambda & 2\beta\sigma_i^2 \\ \eta(\beta-1) & 1-2\eta\beta\sigma_i^2-\lambda \end{vmatrix} = 0$$

$$1-\beta-\lambda-2\eta\beta\sigma_i^2+2\eta\beta^2\sigma_i^2+2\eta\beta\sigma_i^2\lambda-\lambda+\beta\lambda+\lambda^2$$
$$+2\eta\sigma_i^2\beta(1-\beta) = 0$$

$$\lambda^2-(2-\beta-2\eta\beta\sigma_i^2)\lambda+(1-\beta) = 0$$

$$\Delta = (2-\beta-2\eta\beta\sigma_i^2)^2-4(1-\beta)$$

$$\begin{cases} \Delta \geq 0 \iff \text{real eigenvalues} \\ \Delta < 0 \iff \text{complex eigenvalues.} \end{cases}$$

(d). when λ is repeated: $\Delta = 0$

$$(2-\beta-2\eta\beta\sigma_i^2)^2 = 4(1-\beta)$$

$$2-\beta-2\eta\beta\sigma_i^2 = \pm 2\sqrt{1-\beta}$$

$$\eta = \frac{2-\beta \mp 2\sqrt{1-\beta}}{2\beta\sigma_i^2}$$

$$\therefore \text{highest } \eta = \frac{2-\beta+2\sqrt{1-\beta}}{2\beta\sigma_i^2}$$

(e) when λ is real

$$\eta > \frac{2-\beta+2\sqrt{1-\beta}}{2\beta\sigma_i^2} \quad \text{or} \quad \eta < \frac{2-\beta-2\sqrt{1-\beta}}{2\beta\sigma_i^2}$$

(f) when λ is complex:

$$\frac{2-\beta+2\sqrt{1-\beta}}{2\beta\sigma_i^2} < \eta < \frac{2-\beta-2\sqrt{1-\beta}}{2\beta\sigma_i^2}$$

(g) optimal rate $= \dfrac{(\sigma_{max}/\sigma_{min})^2-1}{(\sigma_{max}/\sigma_{min})^2+1}$

$\sigma_{max}^2 = 5 \quad \sigma_{min}^2 = 0.05$

$\therefore$ rate $= \frac{100-1}{100+1} = \frac{99}{101}$

using ordinary gradient descent:

$$\left(\frac{99}{101}\right)^{T_1} \leq 99.5\%$$

using this learning rate with momentum:

in (c):

we got $\lambda_1 \cdot \lambda_2 = \sqrt{1-\beta} = \sqrt{0.9} < 1$

$\therefore$ the higher one of λ_1, λ_2 is $\geq \sqrt{0.9}$

$\therefore$ the convergence rate $r \geq \sqrt{0.9}$

$$(r)^{T_2} \leq 99.5\%$$

$$\log (r)^{T_2} \leq \log 99.5\%$$

$$T_2 \log r \leq \log 99.5\%.$$

$$T_2 \geq \frac{\log 99.5\%}{\log r}$$

$$T_1 > T_2 \quad \left(\sqrt{0.9} < \frac{99}{101}\right)$$

3- Regularization and Instance Noise

$\tilde{X}_i = X_i + N_i \quad N_i \sim N(0, \sigma^2 I_n)$

$$\tilde{X} = \begin{bmatrix} \tilde{x}_1^T \\ \tilde{x}_2^T \\ \cdots \\ \tilde{x}_m^T \end{bmatrix} \quad \tilde{X_i} \in R^n \text{ and } y = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix} \in R^m$$

$$\underset{w}{\operatorname{argmin}} E[\|\tilde{X}w - y\|^2]$$

(a) $E[\|\tilde{X}W - y\|^2]$

$= E[\sum_{i=1}^{m} (\tilde{X}_i^T w - y_i)^2]$

$= \sum_{i=1}^{m} E[((X_i + N_i)^T W - y_i)^2]$

$= \sum_{i=1}^{m} E[(X_i^T w + N_i^T w - y_i)^2]$

$= \sum_{i=1}^{m} E[((X_i^T w - y_i) + N_i^T w)^2]$

$= \sum_{i=1}^{m} E[(X_i^T w - y_i)^2 - 2(N_i^T w)(X_i^T w - y_i) + (N_i^T w)^2]$

$= \sum_{i=1}^{m} E[(X_i^T w - y_i)^2] - 2E((N_i^T w)(X_i^T w - y_i))$
$+ E(N_i^T W)^2$

$= \sum_{i=1}^{m} (X_i^T w - y_i)^2 - 2E[(N_i^T w)(X_i^T W - y_i)] + E(W^T N_i N_i^T W)$

$\because N_i \in (0, \sigma^2 I_n)$

$\therefore 2E[(N_i^T w)(X_i^T w - y_i)] = 0$

$E[N_i N_i^T] = \sigma^2 I_n$

$\therefore = \sum_{i=1}^{m} (X_i^T w - y_i)^2 + w^T \sigma^2 I_n W$

$= (Xw - y)^2 + \sigma^2 \|w\|^2 \cdot m$

$\Downarrow$

is equivalent to a regularized least squares problem:

$$\underset{w}{\arg\min} \frac{1}{m} \|Xw - y\|^2 + \lambda \|w\|^2$$

(b). $\tilde{X_i} = X + N_t$, $N_t \in (0, \sigma^2)$

$L(w) = \frac{1}{2} (\tilde{X} w - y)^2 \quad w_0 = 0$

$\frac{\partial L}{\partial w} = (\tilde{X} w - y) \tilde{X}$

$= (w(X+N_t) - y)(X+N_t)$

$= w(X+N_t)^2 - y(X+N_t)$

$= w(X^2 + 2XN_t + N_t^2) - y(X+N_t)$

$\therefore W_{t+1} = W_t - \eta \frac{\partial L}{\partial w}$

$= W_t - \eta [W_t (X^2 + 2XN_t + N_t^2) - y(X+N_t)]$

$= W_t (1 - \eta (X^2 + 2XN_t + N_t^2)) - \eta y (X+N_t)$

as N_i is i.i.d. $E(N_i) = 0 \quad E(N_i^2) = \sigma^2$

$E(W_{t+1}) = E(W_t) \cdot E(1 - \eta (X^2 + 2XN_t + N_t^2))$
$\qquad - E(\eta y (X+N_t))$

$E(W_{t+1}) = E(W_t) \cdot E(1 - \eta (X^2 + \sigma^2)) - E(\eta y X)$

$= E(W_t)(1 - \eta (X^2 + \sigma^2)) - \eta y X$

(c). For what values of learning rate η do we expect the expectation of the learned weight to converge using gradient descent ?

Solution:

gradient descent to converge

$\Leftrightarrow \quad -1 < 1-\eta(x^2+\sigma^2) < 1$

$$0 < \eta < \frac{2}{x^2+\sigma^2}$$

(d) what would we expect $E(W_t)$ to converge as $t \to \infty$? How does this differ from the situation without noise?

Solution:

when $E(W_t)$ to converge :

$$\frac{\partial L}{\partial w} = w(x^2+2xN_t+N_t^2)-y(x-N_t)$$

$$E\left(\frac{\partial L}{\partial w}\right) = w(x^2+\sigma^2)-yx = 0$$

$$w = \frac{yx}{x^2+\sigma^2}$$

$$w = \frac{y}{x}\,\frac{1}{1+\frac{\sigma^2}{x^2}}$$

without noise : $w = \frac{y}{x}$

there is a scalar value $\frac{1}{1+\frac{\sigma^2}{x^2}}$ when noise is added to x .

7. (a) CSDN, ChatGPT

(b)

(c) 16 hours.

Setup Environment

If you are working on this assignment using Google Colab, please execute the codes below.

Alternatively, you can also do this assignment using a local anaconda environment (or a Python virtualenv). Please clone the GitHub repo by running `git clone https://github.com/Berkeley-CS182/cs182hw2.git` and refer to `README.md` for further details.

In [1]:

```
#@title Mount your Google Drive

import os
from google.colab import drive
drive.mount('/content/gdrive')
```

```
Mounted at /content/gdrive
```

In [2]:

```
#@title Set up mount symlink

DRIVE_PATH = '/content/gdrive/My\ Drive/cs182hw2_sp23'
DRIVE_PYTHON_PATH = DRIVE_PATH.replace('\\', '')
if not os.path.exists(DRIVE_PYTHON_PATH):
  %mkdir $DRIVE_PATH

## the space in `My Drive` causes some issues,
## make a symlink to avoid this
SYM_PATH = '/content/cs182hw2'
if not os.path.exists(SYM_PATH):
  !ln -s $DRIVE_PATH $SYM_PATH
```

In [ ]:

```
#@title Install dependencies

!pip install numpy==1.21.6 imageio==2.9.0 matplotlib==3.2.2
```

```
Collecting numpy==1.21.6
  Downloading numpy-1.21.6-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_x86_6
4.whl (15.9 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━
━━━━ 15.9/15.9 MB 43.7 MB/s eta 0:00:00
Collecting imageio==2.9.0
  Downloading imageio-2.9.0-py3-none-any.whl (3.3 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━
━━━━ 3.3/3.3 MB 63.7 MB/s eta 0:00:00
Collecting matplotlib==3.2.2
  Downloading matplotlib-3.2.2.tar.gz (40.3 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━
━━━━ 40.3/40.3 MB 21.5 MB/s eta 0:00:00
  Preparing metadata (setup.py) ... done
Requirement already satisfied: pillow in /usr/local/lib/python3.10/dist-packages
(from imageio==2.9.0) (9.4.0)
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.10/dist-pac
kages (from matplotlib==3.2.2) (0.11.0)
Requirement already satisfied: kiwisolver>=1.0.1 in /usr/local/lib/python3.10/dis
t-packages (from matplotlib==3.2.2) (1.4.5)
Requirement already satisfied: pyparsing!=2.0.4,!=2.1.2,!=2.1.6,>=2.0.1 in /usr/l
ocal/lib/python3.10/dist-packages (from matplotlib==3.2.2) (3.1.1)
Requirement already satisfied: python-dateutil>=2.1 in /usr/local/lib/python3.10/
dist-packages (from matplotlib==3.2.2) (2.8.2)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.10/dist-package
s (from python-dateutil>=2.1->matplotlib==3.2.2) (1.16.0)
Building wheels for collected packages: matplotlib
  Building wheel for matplotlib (setup.py) ... done
  Created wheel for matplotlib: filename=matplotlib-3.2.2-cp310-cp310-linux_x86_6
4.whl size=11974320 sha256=03bc393952de7912d0eaf4f70c296f5cbcb93db3df14f57f7fff49
f8923b7249
  Stored in directory: /root/.cache/pip/wheels/bb/81/f3/48b8bd245846ae69fcb2281c8
4e848bfea1f5260a870c148ae
Successfully built matplotlib
Installing collected packages: numpy, matplotlib, imageio
  Attempting uninstall: numpy
    Found existing installation: numpy 1.23.5
    Uninstalling numpy-1.23.5:
      Successfully uninstalled numpy-1.23.5
  Attempting uninstall: matplotlib
    Found existing installation: matplotlib 3.7.1
    Uninstalling matplotlib-3.7.1:
      Successfully uninstalled matplotlib-3.7.1
  Attempting uninstall: imageio
    Found existing installation: imageio 2.31.3
    Uninstalling imageio-2.31.3:
      Successfully uninstalled imageio-2.31.3
ERROR: pip's dependency resolver does not currently take into account all the pac
kages that are installed. This behaviour is the source of the following dependenc
y conflicts.
jax 0.4.14 requires numpy>=1.22, but you have numpy 1.21.6 which is incompatible.
jaxlib 0.4.14+cuda11.cudnn86 requires numpy>=1.22, but you have numpy 1.21.6 whic
h is incompatible.
mizani 0.9.3 requires matplotlib>=3.5.0, but you have matplotlib 3.2.2 which is i
ncompatible.
plotnine 0.12.3 requires matplotlib>=3.6.0, but you have matplotlib 3.2.2 which i
s incompatible.
plotnine 0.12.3 requires numpy>=1.23.0, but you have numpy 1.21.6 which is incomp
atible.
tensorflow 2.13.0 requires numpy<=1.24.3,>=1.22, but you have numpy 1.21.6 which
```

```
is incompatible.
Successfully installed imageio-2.9.0 matplotlib-3.2.2 numpy-1.21.6
```

In [3]:

```
#@title Clone homework repo

%cd $SYM_PATH
if not os.path.exists("cs182hw2"):
  !git clone https://github.com/Berkeley-CS182/cs182hw2.git
%cd cs182hw2
```

```
/content/gdrive/My Drive/cs182hw2_sp23
/content/gdrive/My Drive/cs182hw2_sp23/cs182hw2
```

In [4]:

```
#@title Download datasets

%cd deeplearning/datasets/
!bash ./get_datasets.sh
%cd ../..
```

```
/content/gdrive/My Drive/cs182hw2_sp23/cs182hw2/deeplearning/datasets
--2023-09-11 03:46:26--  http://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
(http://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz)
Resolving www.cs.toronto.edu (www.cs.toronto.edu)... 128.100.3.30
Connecting to www.cs.toronto.edu (www.cs.toronto.edu)|128.100.3.30|:80... connect
ed.
HTTP request sent, awaiting response... 200 OK
Length: 170498071 (163M) [application/x-gzip]
Saving to: ‘cifar-10-python.tar.gz’

cifar-10-python.tar 100%[===================>] 162.60M  51.5MB/s    in 3.2s

2023-09-11 03:46:30 (51.5 MB/s) - ‘cifar-10-python.tar.gz’ saved [170498071/170
498071]

cifar-10-batches-py/
cifar-10-batches-py/data_batch_4
cifar-10-batches-py/readme.html
cifar-10-batches-py/test_batch
cifar-10-batches-py/data_batch_3
cifar-10-batches-py/batches.meta
cifar-10-batches-py/data_batch_2
cifar-10-batches-py/data_batch_5
cifar-10-batches-py/data_batch_1
/content/gdrive/My Drive/cs182hw2_sp23/cs182hw2
```

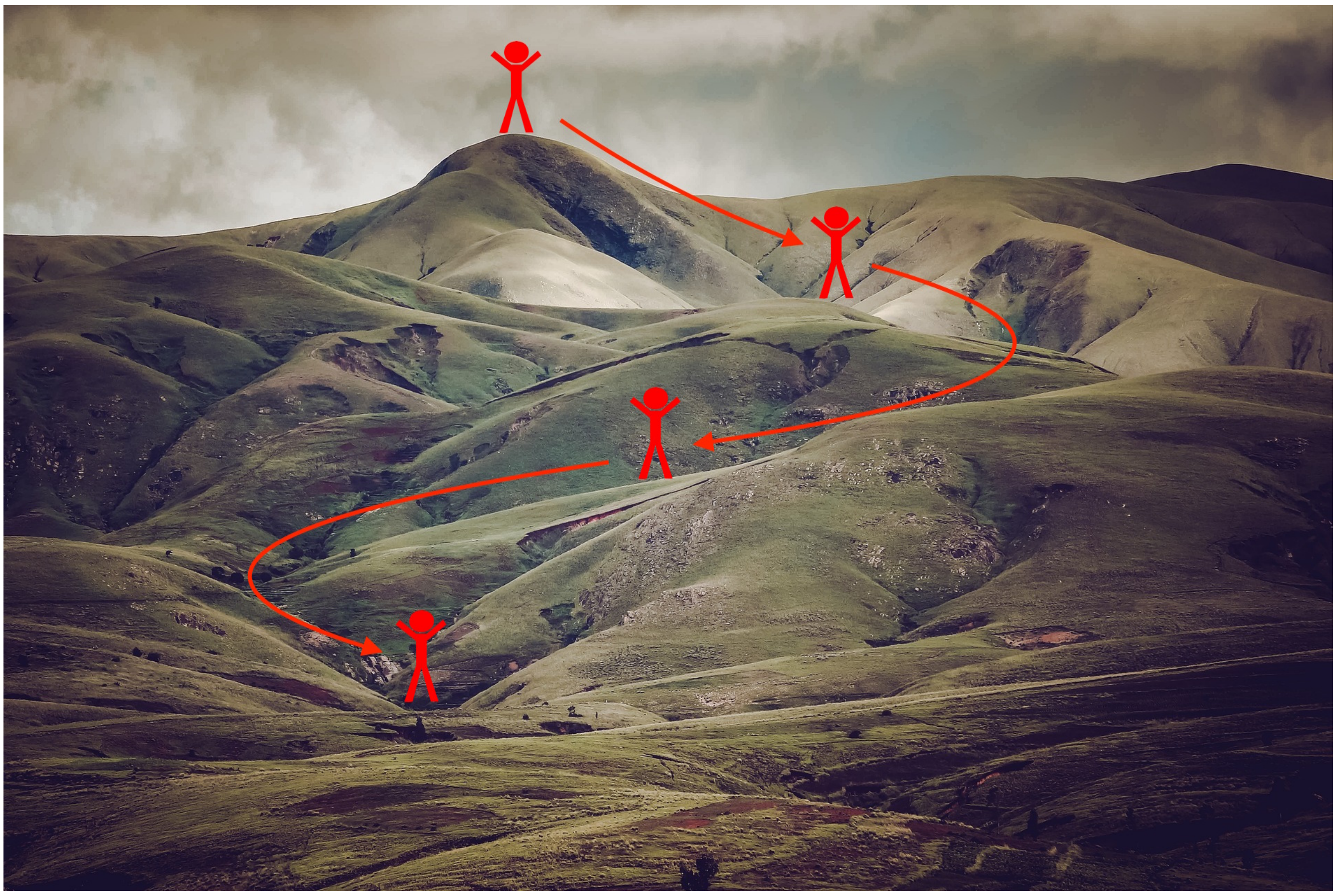
In [5]:

```
#@title Configure Jupyter Notebook

import matplotlib
%matplotlib inline
%load_ext autoreload
%autoreload 2
```

Optimization Methods and Initizalization

Until now, you've always used Gradient Descent to update the parameters and minimize the cost. In this notebook, you will learn more advanced optimization methods that can speed up learning and perhaps even get you to a better final value for the cost function. Having a good optimization algorithm can be the difference between waiting days vs. just a few hours to get a good result.

Gradient descent goes "downhill" on a cost function J. Think of it as trying to do this:



<u>Figure 1</u> : Minimizing the cost is like finding the lowest point in a hilly landscape
At each step of the training, you update your parameters following a certain direction to try to get to the lowest possible point.

In [6]:

```
# As usual, a bit of setup

import json
import time
import numpy as np
import matplotlib.pyplot as plt
from deeplearning.classifiers.fc_net import *
from deeplearning.data_utils import get_CIFAR10_data
from deeplearning.gradient_check import eval_numerical_gradient, eval_numerical_gradient_array
from deeplearning.solver import Solver
import random
import torch
seed = 7
torch.manual_seed(seed)
random.seed(seed)
np.random.seed(seed)

plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

# for auto-reloading external modules
# see http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-ipython

def rel_error(x, y):
    """ returns relative error """
    return np.max(np.abs(x - y) / (np.maximum(1e-8, np.abs(x) + np.abs(y))))
```

In [7]:

```
# Load the (preprocessed) CIFAR10 data.

data = get_CIFAR10_data()
for k, v in data.items():
    print('%s: ' % k, v.shape)
```

```
deeplearning/datasets/cifar-10-batches-py/data_batch_1
deeplearning/datasets/cifar-10-batches-py/data_batch_2
deeplearning/datasets/cifar-10-batches-py/data_batch_3
deeplearning/datasets/cifar-10-batches-py/data_batch_4
deeplearning/datasets/cifar-10-batches-py/data_batch_5
deeplearning/datasets/cifar-10-batches-py/test_batch
X_train:  (49000, 3, 32, 32)
y_train:  (49000,)
X_val:  (1000, 3, 32, 32)
y_val:  (1000,)
X_test:  (1000, 3, 32, 32)
y_test:  (1000,)
```

1 - Stochastic Gradient Descent

A simple optimization method in machine learning is gradient descent (GD). When you take gradient steps with respect to all m examples on each step, it is also called Batch Gradient Descent.

A variant of this is Stochastic Gradient Descent (SGD), which is equivalent to mini-batch gradient descent where each mini-batch has just 1 example. The update rule that you have just implemented does not change. What changes is that you would be computing gradients on just one training example at a time, rather than on the whole training set. The code examples below illustrate the difference between stochastic gradient descent and (batch) gradient descent.

In Stochastic Gradient Descent, you use only 1 training example before updating the gradients. When the training set is large, SGD can be faster. But the parameters will "oscillate" toward the minimum rather than converge smoothly. Here is an illustration of this:

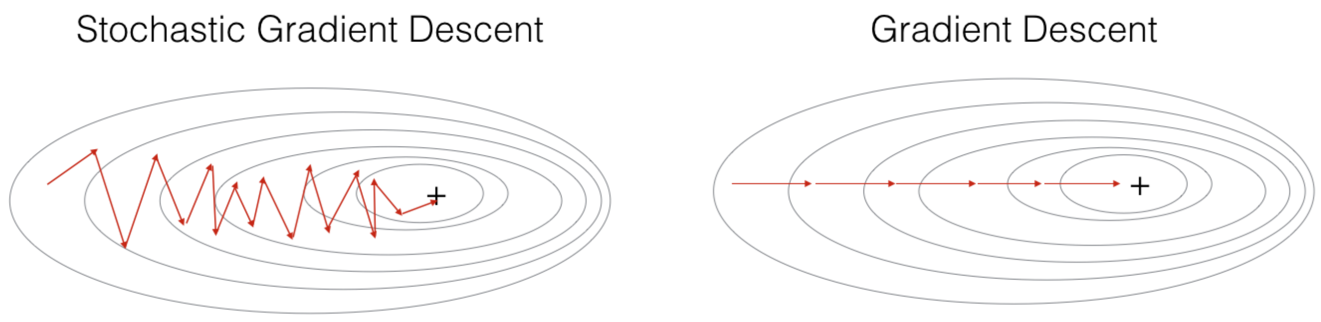
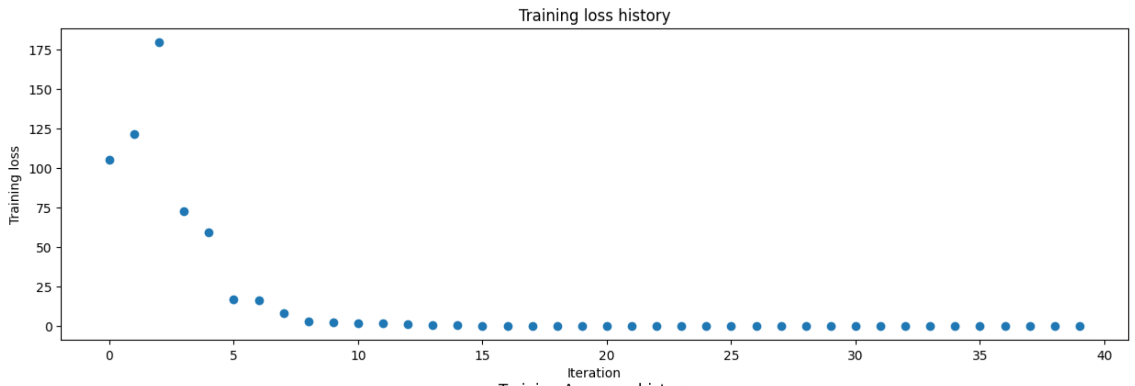


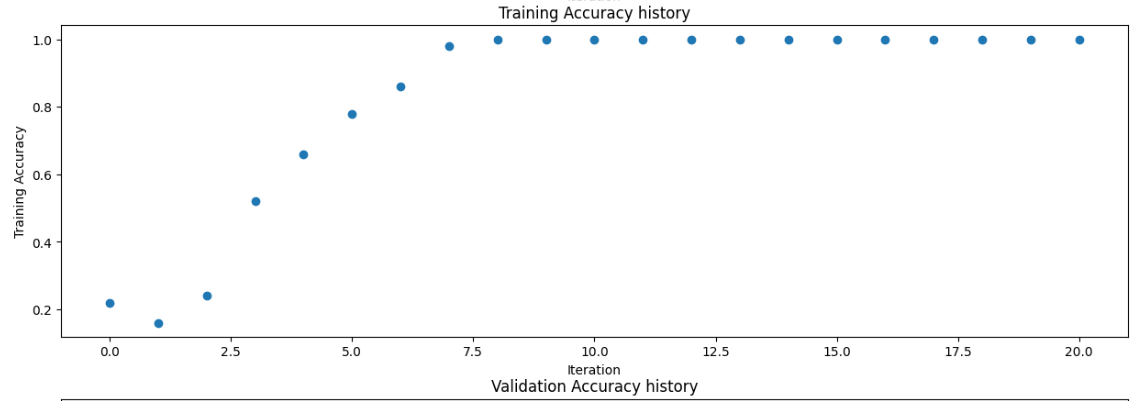
Figure 1 : **SGD vs GD**
"+" denotes a minimum of the cost. SGD leads to many oscillations to reach convergence. But each step is a lot faster to compute for SGD than for GD, as it uses only one training example (vs. the whole batch for GD).

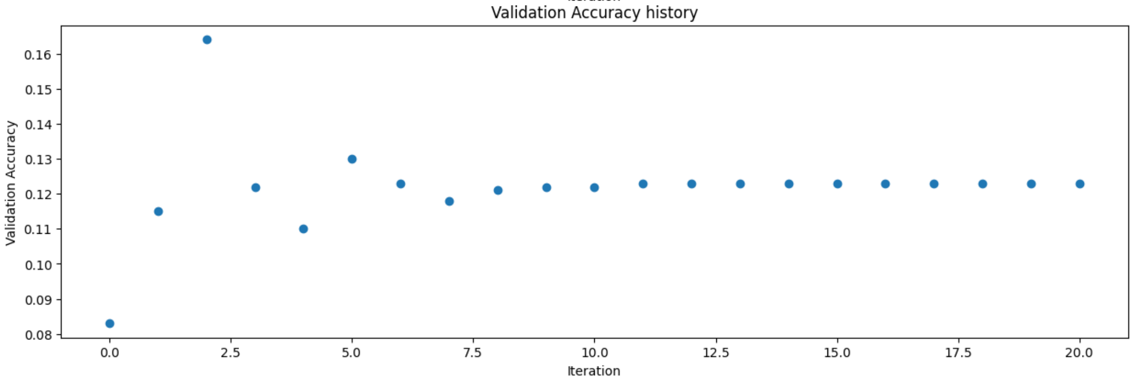
In [10]:

```
## Use a five-layer Net to overfit 50 training examples.


num_train = 50
small_data = {
  'X_train': data['X_train'][:num_train],
  'y_train': data['y_train'][:num_train],
  'X_val': data['X_val'],
  'y_val': data['y_val'],
}

weight_scale = 1e-1
learning_rate = 1e-3
model = FullyConnectedNet([100, 100, 100, 100],
                weight_scale=weight_scale, dtype=np.float64)

solver = Solver(model, small_data,
                print_every=10, num_epochs=20, batch_size=25,
                update_rule='sgd',
                optim_config={
                  'learning_rate': learning_rate,
                }
         )
solver.train()

plt.subplot(3, 1, 1)
plt.plot(solver.loss_history, 'o')
plt.title('Training loss history')
plt.xlabel('Iteration')
plt.ylabel('Training loss')

plt.subplot(3, 1, 2)
plt.plot(solver.train_acc_history, 'o')
plt.title('Training Accuracy history')
plt.xlabel('Iteration')
plt.ylabel('Training Accuracy')

plt.subplot(3, 1, 3)
plt.plot(solver.val_acc_history, 'o')
plt.title('Validation Accuracy history')
plt.xlabel('Iteration')
plt.ylabel('Validation Accuracy')
plt.gcf().set_size_inches(15, 15)

plt.show()
```

```
(Iteration 1 / 40) loss: 105.354393
(Epoch 0 / 20) train acc: 0.220000; val_acc: 0.083000
(Epoch 1 / 20) train acc: 0.160000; val_acc: 0.115000
(Epoch 2 / 20) train acc: 0.240000; val_acc: 0.164000
(Epoch 3 / 20) train acc: 0.520000; val_acc: 0.122000
(Epoch 4 / 20) train acc: 0.660000; val_acc: 0.110000
(Epoch 5 / 20) train acc: 0.780000; val_acc: 0.130000
(Iteration 11 / 40) loss: 1.659190
(Epoch 6 / 20) train acc: 0.860000; val_acc: 0.123000
(Epoch 7 / 20) train acc: 0.980000; val_acc: 0.118000
(Epoch 8 / 20) train acc: 1.000000; val_acc: 0.121000
(Epoch 9 / 20) train acc: 1.000000; val_acc: 0.122000
(Epoch 10 / 20) train acc: 1.000000; val_acc: 0.122000
(Iteration 21 / 40) loss: 0.000355
(Epoch 11 / 20) train acc: 1.000000; val_acc: 0.123000
(Epoch 12 / 20) train acc: 1.000000; val_acc: 0.123000
(Epoch 13 / 20) train acc: 1.000000; val_acc: 0.123000
(Epoch 14 / 20) train acc: 1.000000; val_acc: 0.123000
(Epoch 15 / 20) train acc: 1.000000; val_acc: 0.123000
(Iteration 31 / 40) loss: 0.000367
(Epoch 16 / 20) train acc: 1.000000; val_acc: 0.123000
(Epoch 17 / 20) train acc: 1.000000; val_acc: 0.123000
(Epoch 18 / 20) train acc: 1.000000; val_acc: 0.123000
(Epoch 19 / 20) train acc: 1.000000; val_acc: 0.123000
(Epoch 20 / 20) train acc: 1.000000; val_acc: 0.123000
```



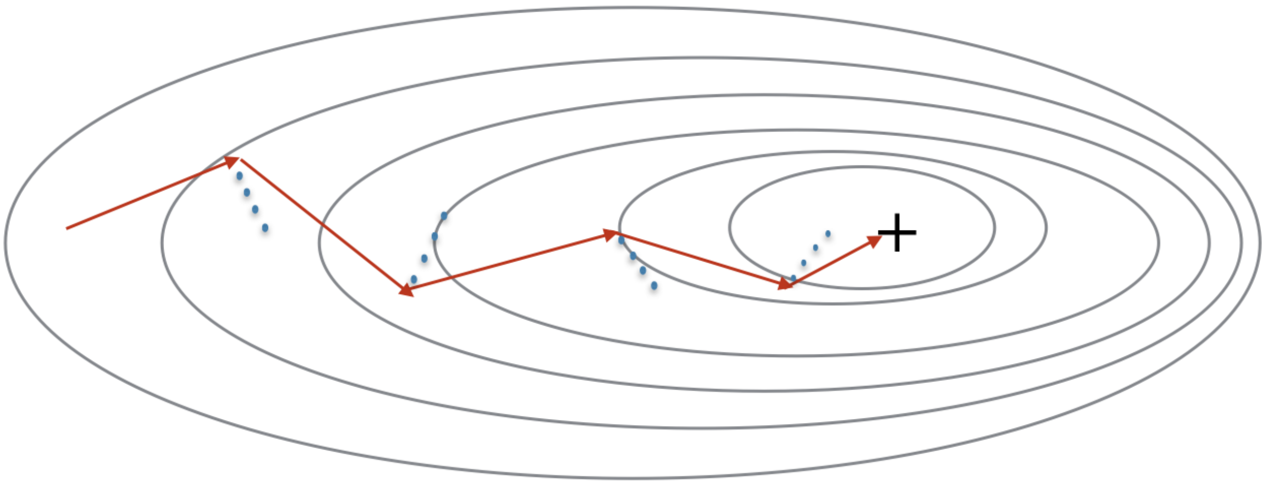




2 - Momentum

Because mini-batch gradient descent makes a parameter update after seeing just a subset of examples, the direction of the update has some variance, and so the path taken by mini-batch gradient descent will "oscillate" toward convergence. Using momentum can reduce these oscillations.

Momentum takes into account the past gradients to smooth out the update. We will store the 'direction' of the previous gradients in the variable v. Formally, this will be the exponentially weighted average of the gradient on previous steps. You can also think of v as the "velocity" of a ball rolling downhill, building up speed (and momentum) according to the direction of the gradient/slope of the hill.



<u>Figure 3</u>: The red arrows shows the direction taken by one step of mini-batch gradient descent with momentum. The blue points show the direction of the gradient (with respect to the current mini-batch) on each step. Rather than just following the gradient, we let the gradient influence v (velocity) and then take a step in the direction of v.

The momentum update rule for a weight matrix w is:

$$\begin{cases} v_{dw}^{t} = m * v_{dw}^{(t-1)} + dw \\ w = w - \alpha v_{dw}^{t} \end{cases} \tag{3}$$

where m is the momentum and α is the learning rate. Note that the iterator `t` starts at 1.

Stochastic gradient descent with momentum is a widely used update rule that tends to make deep networks converge faster than vanilla stochstic gradient descent, it can be viewed conceptually a larger "effective batch size" versus vanilla stochastic gradient descent.

Open the file `deeplearning/optim.py` and read the documentation at the top of the file to make sure you understand the API. **Implement the SGD+momentum update rule** in the function `sgd_momentum` and run the following to check your implementation. You should see errors less than 1e-7.

In [14]:

```
from deeplearning.optim import sgd_momentum

N, D = 4, 5
w = np.linspace(-0.4, 0.6, num=N*D).reshape(N, D)
dw = np.linspace(-0.6, 0.4, num=N*D).reshape(N, D)
v = np.linspace(0.6, 0.9, num=N*D).reshape(N, D)

config = {'learning_rate': 1e-3, 'velocity': v}
next_w, _ = sgd_momentum(w, dw, config=config)

expected_next_w = np.asarray([
 [-0.39994, -0.347375263, -0.294810526, -0.242245789, -0.189681053],
 [-0.137116316, -0.084551579, -0.031986842, 0.020577895, 0.073142632],
 [0.125707368, 0.178272105, 0.230836842, 0.283401579, 0.335966316],
 [0.388531053, 0.441095789, 0.493660526, 0.546225263, 0.59879]])
expected_velocity = np.asarray([
 [-0.06, 0.006842105, 0.073684211, 0.140526316, 0.207368421],
 [0.274210526, 0.341052632, 0.407894737, 0.474736842, 0.541578947],
 [0.608421053, 0.675263158, 0.742105263, 0.808947368, 0.875789474],
 [0.942631579, 1.009473684, 1.076315789, 1.143157895, 1.21]
])

print ('next_w error: ', rel_error(next_w, expected_next_w))
print ('velocity error: ', rel_error(expected_velocity, config['velocity']))
```

```
next_w error:  6.39419001716215753e-09
velocity error:  1.923077600561035e-08
```

Once you have done so, run the following to train a six-layer network with both SGD and SGD+momentum. You should see the SGD+momentum update rule converge a bit faster.

In [15]:

```
num_train = 4000
small_data = {
  'X_train': data['X_train'][:num_train],
  'y_train': data['y_train'][:num_train],
  'X_val': data['X_val'],
  'y_val': data['y_val'],
}

solvers = {}

for update_rule in ['sgd', 'sgd_momentum']:
    print ('running with ', update_rule)
    model = FullyConnectedNet([100, 100, 100, 100, 100], weight_scale=5e-2)

    solver = Solver(model, small_data,
                  num_epochs=5, batch_size=100,
                  update_rule=update_rule,
                  optim_config={
                    'learning_rate': 1e-2,
                  },
                  verbose=True)
    solvers[update_rule] = solver
    solver.train()
    os.makedirs("submission_logs", exist_ok=True)
    solver.record_histories_as_npz("submission_logs/optimizer_experiment_{}".format(update_rule))
    print

plt.subplot(3, 1, 1)
plt.title('Training loss')
plt.xlabel('Iteration')

plt.subplot(3, 1, 2)
plt.title('Training accuracy')
plt.xlabel('Epoch')

plt.subplot(3, 1, 3)
plt.title('Validation accuracy')
plt.xlabel('Epoch')

for update_rule, solver in solvers.items():
    plt.subplot(3, 1, 1)
    plt.plot(solver.loss_history, 'o', label=update_rule)

    plt.subplot(3, 1, 2)
    plt.plot(solver.train_acc_history, '-o', label=update_rule)

    plt.subplot(3, 1, 3)
    plt.plot(solver.val_acc_history, '-o', label=update_rule)

for i in [1, 2, 3]:
    plt.subplot(3, 1, i)
    plt.legend(loc='upper center', ncol=4)
plt.gcf().set_size_inches(15, 15)
plt.show()
```

```
running with  sgd
(Iteration 1 / 200) loss: 2.490171
(Epoch 0 / 5) train acc: 0.119000; val_acc: 0.105000
(Iteration 11 / 200) loss: 2.238576
(Iteration 21 / 200) loss: 2.216133
(Iteration 31 / 200) loss: 2.054264
(Epoch 1 / 5) train acc: 0.289000; val_acc: 0.268000
(Iteration 41 / 200) loss: 2.010914
(Iteration 51 / 200) loss: 1.942512
(Iteration 61 / 200) loss: 1.905087
(Iteration 71 / 200) loss: 1.908923
(Epoch 2 / 5) train acc: 0.347000; val_acc: 0.301000
(Iteration 81 / 200) loss: 1.872406
(Iteration 91 / 200) loss: 1.805315
(Iteration 101 / 200) loss: 2.003714
(Iteration 111 / 200) loss: 1.725869
(Epoch 3 / 5) train acc: 0.391000; val_acc: 0.330000
(Iteration 121 / 200) loss: 1.782641
(Iteration 131 / 200) loss: 1.722920
```

A further step: as we discussed above, we can see how SGD+Momentum is conceptually giving you a larger "effective batch size" by increase the batch size used in the SGD above. In this way, SGD+Momentum can significantly speed up training.

Tune the batch size for plain SGD so that the training accuracy is similar to that of SGD with momentum. The average accuracy difference between them should be less than 0.04 . The accuracy is averaged over three different random seeds for better stability.

In [16]:

```
###############################################################################
# TODO: Tune the batch size for the SGD below until you observe              #
# similar end of iteration training performance.                             #
# It means rel_error(train_acc) < 0.04                                       #
###############################################################################
batch_sizes = {
  'sgd_momentum': 100,
  'sgd': 400,  # tune the batch size of SGD (must be multiples of 100)
}

num_train = 6000
small_data = {
  'X_train': data['X_train'][:num_train],
  'y_train': data['y_train'][:num_train],
  'X_val': data['X_val'],
  'y_val': data['y_val'],
}

solvers = {}
total_acc = {}

labels = {
  'sgd_momentum': 'sgd_momentum',
  'sgd': 'sgd_large_bsz',
}

for update_rule in ['sgd', 'sgd_momentum']:
    print ('running with', update_rule, ' ; seed =', seed)
    # set the epochs so that we have the same number of steps for both rules
    training_epochs = 5 * int(batch_sizes[update_rule]/100)
    solvers[update_rule] = {}
    total_acc[update_rule] = 0

    for seed in [100, 200, 300]:
        torch.manual_seed(seed)
        np.random.seed(seed)
        model = FullyConnectedNet([100, 100, 100, 100, 100], weight_scale=5e-2)

        solver = Solver(
            model, small_data,
            num_epochs=training_epochs,
            batch_size=batch_sizes[update_rule],
            update_rule=update_rule,
            optim_config={
                'learning_rate': 1e-2,  # please do not change the learning rate
            },
            verbose=True,
            log_acc_iteration=True)

        solvers[update_rule][seed] = solver
        solver.train()
        solver.record_histories_as_npz(
            "submission_logs/sgd_momentum_compare_{}_{}"
            .format(update_rule, seed)
        )

        total_acc[update_rule] += solvers[update_rule][seed].train_acc_history[-1]

print('Average Training Acc for sgd:', total_acc['sgd'] / 3)
print('Average Training Acc for sgd_momentum:', total_acc['sgd_momentum'] / 3)
print('Train Acc Difference: ',
       rel_error(total_acc['sgd'] / 3,
                 total_acc['sgd_momentum'] / 3))

def plot_solver_seeds(solver_s, x_field, y_field, seeds, label):
    a = np.array([getattr(solver_s[seed], y_field) for seed in seeds])
    if x_field is None:
        plt_x = np.arange(a.shape[1]) + 1
    else:
        plt_x = getattr(solver_s[seeds[0]], x_field)
    plt.plot(plt_x, a.mean(axis=0), label=label)
    plt.fill_between(plt_x, a.min(axis=0), a.max(axis=0), alpha=0.4)

plt.subplot(3, 1, 1)
plt.title('Training loss')
plt.xlabel('Iteration')

plt.subplot(3, 1, 2)
plt.title('Training accuracy')
plt.xlabel('Iteration')

plt.subplot(3, 1, 3)
plt.title('Validation accuracy')
plt.xlabel('Iteration')

for update_rule, solver_s in solvers.items():
    plt.subplot(3, 1, 1)
    # plt.plot(solver.loss_history, 'o', label=labels[update_rule])
    plot_solver_seeds(solver_s, None, 'loss_history',
                      [100, 200, 300], labels[update_rule])

    plt.subplot(3, 1, 2)
    # plt.plot(solver.log_acc_iteration_history, solver.train_acc_history, '-o', label=labels[upd
    plot_solver_seeds(solver_s, 'log_acc_iteration_history', 'train_acc_history',
                      [100, 200, 300], labels[update_rule])

    plt.subplot(3, 1, 3)
    # plt.plot(solver.log_acc_iteration_history, solver.val_acc_history, '-o', label=labels[updat
    plot_solver_seeds(solver_s, 'log_acc_iteration_history', 'val_acc_history',
                      [100, 200, 300], labels[update_rule])

for i in [1, 2, 3]:
    plt.subplot(3, 1, i)
    plt.legend(loc='upper center', ncol=4)
plt.gcf().set_size_inches(15, 15)
plt.show()
```

```
running with sgd  ; seed = 7
(Iteration 1 / 300) loss: 2.612065
(Epoch 0 / 20) train acc: 0.102000; val_acc: 0.093000
(Iteration 11 / 300) loss: 2.224131
(Epoch 1 / 20) train acc: 0.208000; val_acc: 0.199000
(Iteration 21 / 300) loss: 2.112900
(Epoch 2 / 20) train acc: 0.246000; val_acc: 0.245000
(Iteration 31 / 300) loss: 2.035767
(Iteration 41 / 300) loss: 1.978076
(Epoch 3 / 20) train acc: 0.280000; val_acc: 0.273000
(Iteration 51 / 300) loss: 2.001033
(Epoch 4 / 20) train acc: 0.320000; val_acc: 0.286000
(Iteration 61 / 300) loss: 1.938605
(Iteration 71 / 300) loss: 1.951672
(Epoch 5 / 20) train acc: 0.324000; val_acc: 0.295000
(Iteration 81 / 300) loss: 1.848817
(Epoch 6 / 20) train acc: 0.365000; val_acc: 0.306000
(Iteration 91 / 300) loss: 1.899495
(Iteration 101 / 300) loss: 1.875119
```

3 - Adam

Adam is one of the most effective optimization algorithms for training neural networks. It combines ideas from RMSProp and Momentum.

How does Adam work?

1. It calculates an exponentially weighted average of past gradients, and stores it in variables v (before bias correction) and $m^{corrected}$ (with bias correction).
2. It calculates an exponentially weighted average of the squares of the past gradients, and stores it in variables s (before bias correction) and $v^{corrected}$ (with bias correction).
3. It updates parameters in a direction based on combining information from "1" and "2".

$$\begin{cases} m_{dw} = \beta_1 m_{dw} + (1-\beta_1)\frac{\partial \mathcal{J}}{\partial W} \\ m^{corrected}_{dw} = \frac{m_{dw}}{1-(\beta_1)^t} \\ v_{dw} = \beta_2 v_{dw} + (1-\beta_2)(\frac{\partial \mathcal{J}}{\partial W})^2 \\ v^{corrected}_{dw} = \frac{v_{dw}}{1-(\beta_2)^t} \\ w = w - \alpha \frac{m^{corrected}_{dw}}{\sqrt{v^{corrected}_{dw}}+\varepsilon} \end{cases}$$

where:

- t counts the number of steps taken of Adam
- β_1 and β_2 are hyperparameters that control the two exponentially weighted averages.
- α is the learning rate
- ε is a very small number to avoid dividing by zero

RMSProp [1] and Adam [2] are update rules that set per-parameter learning rates by using a running average of the second moments of gradients.

In the file `deeplearning/optim.py`, **implement the RMSProp update rule** in the `rmsprop` function (optional, the solution is provided at the bottom of optim.py) and **implement the Adam update rule** in the `adam` function, and check your implementations using the tests below.

[1] Tijmen Tieleman and Geoffrey Hinton. "Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude." COURSERA: Neural Networks for Machine Learning 4 (2012).

[2] Diederik Kingma and Jimmy Ba, "Adam: A Method for Stochastic Optimization", ICLR 2015.

In [22]:

```
# Test RMSProp implementation; you should see errors less than 1e-7.
from deeplearning.optim import rmsprop

N, D = 4, 5
w = np.linspace(-0.4, 0.6, num=N*D).reshape(N, D)
dw = np.linspace(-0.6, 0.4, num=N*D).reshape(N, D)
cache = np.linspace(0.6, 0.9, num=N*D).reshape(N, D)

config = {'learning_rate': 1e-2, 'cache': cache}
next_w, _ = rmsprop(w, dw, config=config)

expected_next_w = np.asarray([
  [-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]])
expected_cache = np.asarray([
  [ 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    ]])

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

```
next_w error:  9.524687511038133e-08
cache error:  2.6477955807156126e-09
```

In [23]:

```
# Test Adam implementation; you should see errors around 1e-7 or less.
from deeplearning.optim import adam

N, D = 4, 5
w = np.linspace(-0.4, 0.6, num=N*D).reshape(N, D)
dw = np.linspace(-0.6, 0.4, num=N*D).reshape(N, D)
m = np.linspace(0.6, 0.9, num=N*D).reshape(N, D)
v = np.linspace(0.7, 0.5, num=N*D).reshape(N, D)

config = {'learning_rate': 1e-2, 'm': m, 'v': v, 't': 5}
next_w, _ = adam(w, dw, config=config)

expected_next_w = np.asarray([
  [-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]])
expected_v = np.asarray([
  [ 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,    ]])
expected_m = np.asarray([
  [ 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      ]])
expected_t = 6

print ('next_w error: ', rel_error(expected_next_w, next_w))
print ('v error: ', rel_error(expected_v, config['v']))
print ('m error: ', rel_error(expected_m, config['m']))
print ('t error: ', rel_error(expected_t, config['t']))
```

```
next_w error:  1.1395691798535431e-07
v error:  4.208314038113071e-09
m error:  4.214963193114416e-09
t error:  0.0
```

Once you have debugged your RMSProp and Adam implementations, run the following to train a pair of deep networks using these new update rules. As a sanity check, you should see that RMSProp and Adam typically obtain at least 45% training accuracy within 5 epochs.

In [24]:

```
num_train = 4000
small_data = {
  'X_train': data['X_train'][:num_train],
  'y_train': data['y_train'][:num_train],
  'X_val': data['X_val'],
  'y_val': data['y_val'],
}

learning_rates = {'rmsprop': 1e-4, 'adam': 1e-3, 'sgd': 1e-2, 'sgd_momentum': 1e-2}
for update_rule in ['sgd', 'sgd_momentum', 'adam', 'rmsprop']:
    print ('running with ', update_rule)

    torch.manual_seed(0)
    np.random.seed(0)

    model = FullyConnectedNet([100, 100, 100, 100, 100], weight_scale=5e-2)

    solver = Solver(model, small_data,
                  num_epochs=5, batch_size=100,
                  update_rule=update_rule,
                  optim_config={
                    'learning_rate': learning_rates[update_rule]
                  },
                  verbose=True,)
    solvers[update_rule] = solver
    solver.train()
    solver.record_histories_as_npz("submission_logs/optimizer_experiment_{}".format(update_rule))
    print

plt.subplot(3, 1, 1)
plt.title('Training loss')
plt.xlabel('Iteration')

plt.subplot(3, 1, 2)
plt.title('Training accuracy')
plt.xlabel('Epoch')

plt.subplot(3, 1, 3)
plt.title('Validation accuracy')
plt.xlabel('Epoch')

for update_rule, solver in solvers.items():
    plt.subplot(3, 1, 1)
    plt.plot(solver.loss_history, label=update_rule)

    plt.subplot(3, 1, 2)
    plt.plot(solver.train_acc_history, '-o', label=update_rule)

    plt.subplot(3, 1, 3)
    plt.plot(solver.val_acc_history, '-o', label=update_rule)

for i in [1, 2, 3]:
    plt.subplot(3, 1, i)
    plt.legend(loc='upper center', ncol=4)
plt.gcf().set_size_inches(15, 15)
plt.show()
```

```
running with  sgd
(Iteration 1 / 200) loss: 2.920601
(Epoch 0 / 5) train acc: 0.095000; val_acc: 0.106000
(Iteration 11 / 200) loss: 2.259420
(Iteration 21 / 200) loss: 2.259472
(Iteration 31 / 200) loss: 2.111666
(Epoch 1 / 5) train acc: 0.265000; val_acc: 0.249000
(Iteration 41 / 200) loss: 2.049079
(Iteration 51 / 200) loss: 2.008793
(Iteration 61 / 200) loss: 1.969929
(Iteration 71 / 200) loss: 2.015482
(Epoch 2 / 5) train acc: 0.342000; val_acc: 0.323000
(Iteration 81 / 200) loss: 1.877204
(Iteration 91 / 200) loss: 1.746421
(Iteration 101 / 200) loss: 1.820583
(Iteration 111 / 200) loss: 1.814485
(Epoch 3 / 5) train acc: 0.343000; val_acc: 0.315000
(Iteration 121 / 200) loss: 1.839074
(Iteration 131 / 200) loss: 1.712924
```

Initialization

Training your neural network requires specifying an initial value of the weights. A well chosen initialization method will help learning.

A well chosen initialization can:

- Speed up the convergence of gradient descent
- Increase the odds of gradient descent converging to a lower training (and generalization) error

We will use three different initilization methods to illustrate this concept.

- Zero Initialization:

  This initializes the weights to 0.
- Random Initialization:

  This initializes the weights drawn from a distribution with *manually* specified scales. In this homework, **we use normal distribution with the `weight_scale` argument in `fc_net.py` as its std.**
- He/Xavier/Glorot Initialization:

  This is a special case for random initialization, where the scaling factor is set so that the std of each parameter is `gain / sqrt(fan_mode)` . `gain` is determined by the activation function. For example, linear activation has `gain = 1` and ReLU activation has `gain = sqrt(2)` . There are three types of fan mode:
  - Fan in: `fan_mode = in_dim` , i.e., the width of the preceding layer, preserving the magnitude in forward pass. **This is what you need to implement below** and also the default in PyTorch.
  - Fan out: `fan_mode = out_dim` , i.e., the width of the succeeding layer, preserving the magnitude in backpropagation.
  - Average: `fan_mode = (in_dim + out_dim) / 2` .

  When the std is determined, another choice is between normal distribution or uniform distribution. In this homework **we use normal distribution for initialization.**

In [26]:

```
################################################################################
# TODO:
# 1. implement three initialization schemes in
#    deeplearning/classifiers/fc_net.py
# 2. record the mean of l2 norm of the gradients
#    in the deeplearning/solver.py
################################################################################

learning_rates = {'sgd': 1e-3}
update_rule = 'sgd'
solvers = dict()

num_train = 4000
small_data = {
  'X_train': data['X_train'][:num_train],
  'y_train': data['y_train'][:num_train],
  'X_val': data['X_val'],
  'y_val': data['y_val'],
}

for initialization in ['he', 'random', 'zero']:
    print ('running with ', update_rule)

    model = FullyConnectedNet([50]*10, initialization=initialization)
    weight_stds = [float(model.params["W" + str(i)].std()) for i in range(1, 12)]
    print("initialization scheme:", initialization)
    if initialization == "he":
        # It is fine if the rel_error is less than 0.03 due to randomness
        print("Layer 1, rel_error", rel_error(0.02551551815399, weight_stds[0]))
        print("Layer 2, rel_error", rel_error(0.2, weight_stds[1]))
    elif initialization == "random":
        # It is fine if the rel_error is less than 0.03 due to randomness
        print("Layer 1, rel_error", rel_error(0.01, weight_stds[0]))
        print("Layer 2, rel_error", rel_error(0.01, weight_stds[1]))
    with open("submission_logs/w_stds_{}.json".format(initialization), "w", encoding="utf-8") as
        json.dump(weight_stds, f)

    solver = Solver(model, small_data,
                    num_epochs=5, batch_size=100,
                    update_rule=update_rule,
                    optim_config={
                      'learning_rate': learning_rates[update_rule]
                    },
                    verbose=True)
    solvers[initialization] = solver
    solver.train()
    solver.record_histories_as_npz("submission_logs/initialization_experiment_{}".format(initiali
    print

plt.subplot(4, 1, 1)
plt.title('Training loss')
plt.xlabel('Iteration')


plt.subplot(4, 1, 2)
plt.title('Training accuracy')
plt.xlabel('Epoch')

plt.subplot(4, 1, 3)
plt.title('Validation accuracy')
plt.xlabel('Epoch')

plt.subplot(4, 1, 4)
plt.title('Mean of the Gradient Norm')
plt.xlabel('Iteration')

for initialization, solver in solvers.items():
    plt.subplot(4, 1, 1)
    plt.plot(solver.loss_history, label=initialization)

    plt.subplot(4, 1, 2)
    plt.plot(solver.train_acc_history, '-o', label=initialization)

    plt.subplot(4, 1, 3)
    plt.plot(solver.val_acc_history, '-o', label=initialization)

    plt.subplot(4, 1, 4)
    plt.plot(solver.log_grad_norm_history, label=initialization)

for i in [1, 2, 3, 4]:
    plt.subplot(4, 1, i)
    plt.legend(loc='upper center', ncol=4)
plt.gcf().set_size_inches(15, 20)

plt.show()
```

```
running with  sgd
initialization scheme: he
Layer 1, rel_error 0.0009423196634826634
Layer 2, rel_error 0.0043755401773241205
(Iteration 1 / 200) loss: inf
(Epoch 0 / 5) train acc: 0.135000; val_acc: 0.127000
(Iteration 11 / 200) loss: 7.885903
(Iteration 21 / 200) loss: 4.763057
(Iteration 31 / 200) loss: 3.685940
(Epoch 1 / 5) train acc: 0.156000; val_acc: 0.159000
(Iteration 41 / 200) loss: 3.786861
(Iteration 51 / 200) loss: 3.344775
(Iteration 61 / 200) loss: 3.113828
(Iteration 71 / 200) loss: 3.200791
(Epoch 2 / 5) train acc: 0.181000; val_acc: 0.159000
(Iteration 81 / 200) loss: 2.806365
(Iteration 91 / 200) loss: 2.329191
(Iteration 101 / 200) loss: 2.435604
(Iteration 111 / 200) loss: 2.340992
```

Question:

What you observe in the mean of gradient norm plot above in the above plots? Try to give an explanation. **Write your answer on the written assignment.**

Train a good model!

Train the best fully-connected model that you can on CIFAR-10, storing your best model in the `best_model` variable and the solver used in the `best_solver` variable. We require you to get at least 45% accuracy *on the validation set* using a fully-connected net.

If you are careful it should be possible to get accuracies above 55%, but we don't require it for this part and won't assign extra credit for doing so. Later in the assignment we will ask you to train the best convolutional network that you can on CIFAR-10, and we would prefer that you spend your effort working on convolutional nets rather than fully-connected nets.

In [28]:

```
best_model = None
best_solver = None


width = 200  # please don't change this
n_layers = 10  # please don't change this

################################################################################
# TODO: Train the best FullyConnectedNet that you can on CIFAR-10.             #
# Store your best model in the best_model variable                             #
# and the solver used to train it in the best_solver variable                  #
# Please use the He Initialization and adam.                                   #
# You could tune the following variables only below,                           #
# it shoud achieve above 45% accuracy on the validation set.                   #
################################################################################
lr = 5e-3
num_epochs = 10
batch_size = 128
lr_decay = 0.9
update_rule = 'adam'
################################################################################
#                              END OF YOUR CODE                                #
################################################################################

np.random.seed(2023)  # please don't change this for reproducibility
torch.manual_seed(2023)  # please don't change this for reproducibility
model = FullyConnectedNet([width] * n_layers,
                          initialization='he'
                          )
solver = Solver(model,
                data,
                num_epochs=num_epochs,
                batch_size=batch_size,
                update_rule=update_rule,
                optim_config={
                  'learning_rate': lr
                },
                lr_decay=lr_decay,
                verbose=True)
solver.train()
best_model = model
best_solver = solver
```

```
(Iteration 1 / 3820) loss: inf
(Epoch 0 / 10) train acc: 0.087000; val_acc: 0.119000
(Iteration 11 / 3820) loss: inf
(Iteration 21 / 3820) loss: 3.000119
(Iteration 31 / 3820) loss: 2.611009
(Iteration 41 / 3820) loss: 2.285165
(Iteration 51 / 3820) loss: 2.196919
(Iteration 61 / 3820) loss: 2.189476
(Iteration 71 / 3820) loss: 2.088932
(Iteration 81 / 3820) loss: 2.216598
(Iteration 91 / 3820) loss: 2.023130
(Iteration 101 / 3820) loss: 2.065231
(Iteration 111 / 3820) loss: 2.049746
(Iteration 121 / 3820) loss: 2.072670
(Iteration 131 / 3820) loss: 2.181953
(Iteration 141 / 3820) loss: 2.147633
(Iteration 151 / 3820) loss: 2.123560
(Iteration 161 / 3820) loss: 1.991900
(Iteration 171 / 3820) loss: 2.039955
```

Test your model

Run your best model on the validation and test sets and record the training logs of the best solver. You should achieve above 45% accuracy on the validation set.

In [29]:

```
y_test_pred = np.argmax(best_model.loss(data['X_test']), axis=1)
y_val_pred = np.argmax(best_model.loss(data['X_val']), axis=1)
val_acc = (y_val_pred == data['y_val']).mean()
test_acc = (y_test_pred == data['y_test']).mean()
print ('Validation set accuracy: ', val_acc)
print ('Test set accuracy: ', test_acc)
best_solver.record_histories_as_npz('submission_logs/best_fc_model.npz')
import json
with open("submission_logs/results.json", "w", encoding="utf-8") as f:
    json.dump(dict(
        val_acc = val_acc,
        test_acc = test_acc,
        lr = lr,
        num_epochs = num_epochs,
        batch_size = batch_size,
        lr_decay = lr_decay,
        update_rule = update_rule
    ), f)
```

```
Validation set accuracy:  0.486
Test set accuracy:  0.493
```

Collect your submissions

On Colab, after running the following cell, you can download your submissions from the `Files` tab, which can be opened by clicking the file icon on the left hand side of the screen.

In [30]:

```
!rm -f cs182hw2_submission.zip
!zip -r cs182hw2_submission.zip . -x "*.git*" "*deeplearning/datasets*" "*.ipynb_checkpoints*" "
```

```
  adding: deeplearning/ (stored 0%)
  adding: deeplearning/__init__.py (stored 0%)
  adding: deeplearning/classifiers/ (stored 0%)
  adding: deeplearning/classifiers/__init__.py (stored 0%)
  adding: deeplearning/classifiers/fc_net.py (deflated 80%)
  adding: deeplearning/data_utils.py (deflated 68%)
  adding: deeplearning/gradient_check.py (deflated 68%)
  adding: deeplearning/layer_utils.py (deflated 57%)
  adding: deeplearning/layers.py (deflated 78%)
  adding: deeplearning/optim.py (deflated 75%)
  adding: deeplearning/solver.py (deflated 70%)
  adding: deeplearning/vis_utils.py (deflated 65%)
  adding: hw2_optimizer_init.ipynb (deflated 76%)
  adding: submission_logs/ (stored 0%)
  adding: submission_logs/optimizer_experiment_sgd.npz (deflated 46%)
  adding: submission_logs/optimizer_experiment_sgd_momentum.npz (deflated 46%)
  adding: submission_logs/sgd_momentum_compare_sgd_100.npz (deflated 46%)
  adding: submission_logs/sgd_momentum_compare_sgd_200.npz (deflated 46%)
  adding: submission_logs/sgd_momentum_compare_sgd_300.npz (deflated 46%)
  adding: submission_logs/sgd_momentum_compare_sgd_momentum_100.npz (deflated 4
6%)
  adding: submission_logs/sgd_momentum_compare_sgd_momentum_200.npz (deflated 4
6%)
  adding: submission_logs/sgd_momentum_compare_sgd_momentum_300.npz (deflated 4
6%)
  adding: submission_logs/optimizer_experiment_adam.npz (deflated 46%)
  adding: submission_logs/optimizer_experiment_rmsprop.npz (deflated 46%)
  adding: submission_logs/w_stds_he.json (deflated 48%)
  adding: submission_logs/initialization_experiment_he.npz (deflated 47%)
  adding: submission_logs/w_stds_random.json (deflated 50%)
  adding: submission_logs/initialization_experiment_random.npz (deflated 63%)
  adding: submission_logs/w_stds_zero.json (deflated 80%)
  adding: submission_logs/initialization_experiment_zero.npz (deflated 63%)
  adding: submission_logs/best_fc_model.npz (deflated 53%)
  adding: submission_logs/results.json (deflated 20%)
```