UniTs - University of Trieste

Faculty of Data Science and Artificial Intelligence/Scientific Computing

Department of mathematics informatics and geosciences

Generative AI and NLP

Author:
Christian Faccio

March 29, 2025

github.com/christianfaccio ✉ christianfaccio@outlook.it

Contents

1
Git

GitHub Cheatsheet

These informations have been taken from GitHub Docs.

1.1 About Git

1.1.1 About version control and Git

Definition: ***Version control***

A version control system, or VCS, tracks the history of changes as people and teams collaborate on projects together. As developers make changes to the project, any earlier version of the project can be recovered at any time.

Developers can revew project history to find out:

- Which changes were made?
- Who made the changes?
- When were the changes made?
- Why were changes needed?

In a **distributed version control system (DVCSs)**, every developer has a full copy of the prokect and project history. They don't need a constant connection to a central repository. `Git` is the most popular distributed version control syste. It is commonly used for both open source and commercial software development, with significant benefits for individuals, teams and businesses.

Tip:

- **Git lens** is used to see the entire timeline of the project, including decisions, changes and progressions. From the moment they access the history of a project, the developer has all the context they need to understand it and start contributing.
- Developers work in every time zone. With a DVCS like Git, collaboration can happen any time while maintaining source code integrity. Using **branches**, developers can safely propose changes to production code.

1.1.2 Repositories

A repository, or Git project, encompasses the entire collection of files and folders associated with a project, along with each file's revision history. The file history appears as **snapshots** in time called commits. The commits can be organized into multiple lines of development called branches. Because Git is a DVCS, repositories are self-contained units and anyone who has a copy of the repository can access the entire codebase and its history. Using the command line or other ease-of-use interfaces, a Git repository also allows for: interaction with the history, cloning the repository, creating branches, committing, merging, comparing changes across versions of code, and more.

A repository, or Git project, encompasses the entire collection of files and folders associated with a project, along with each file's revision history. The file history appears as snapshots in time called commits. The commits can be organized into multiple lines of development called branches. Because Git is a DVCS, repositories are self-contained units and anyone who has a copy of the repository can access the entire codebase and its history. Using the command line or other ease-of-use interfaces, a Git repository also allows for: interaction with the history, cloning the repository, creating branches, committing, merging, comparing changes across versions of code, and more.

1.1.3 GitHub

GitHub hosts Git repositories and provides developers with tools to ship better code through command line features, issues (threaded discussions), pull requests, code review, or the use of a collection of free and for-purchase apps in the GitHub Marketplace. With collaboration layers like the GitHub flow, a community of 100 million developers, and an ecosystem with hundreds of integrations, GitHub changes the way software is built.

GitHub builds collaboration directly into the development process. Work is organized into repositories where developers can outline requirements or direction and set expectations for team members. Then, using the GitHub flow, developers simply create a branch to work on updates, commit changes to save them, open a pull request to propose and discuss changes, and merge pull requests once everyone is on the same page.

1.1.4 Command Line Interface

To use Git, developers use specific commands to copy, create, change, and combine code. These commands can be executed directly from the command line or by using an application like GitHub Desktop. Here are some common commands for using Git:

- `git init`: Initializes a new Git repository. Until you run this command inside a repository or directory, it's just a regular folder. Only after you input this does it accept further Git commands.
- `git config`: Short for ”configure,” this is most useful when you're setting up Git for the first time.
- `git help`: Forgot a command? Type this into the command line to bring up the 21 most common git commands.
- `git status`: We all get a little nervous the first time we commit our changes. This command shows you the status of changes as untracked, modified, or staged.
- `git add`: This does not add new files to your repository. Instead, it brings new files to Git's attention. After you add files, they're included in Git's ”snapshots” of the repository.
- `git commit`: Git's most important command. After you make any sort of change, you input this in order to take a ”snapshot” of the repository. Usually it goes `git commit -m "Message here"`.
- `git branch`: Working on multiple features at once? Use this command to list, create, or delete branches.
- `git checkout`: Literally allows you to ”check out” a repository that you are not currently inside. This is a navigational command that lets you move to the repository you want to check.
- `git merge`: When you're done working on a branch, you can merge your changes back to the master branch, which is visible to all collaborators.
- `git push`: If you're working on your local computer, and want your commits to be visible online on GitHub as well, you ”push” the changes up to GitHub with this command.

- `git pull`: If you’re working on your local computer and want your commits to be visible online on GitHub as well, you ”push” the changes up to GitHub with this command.
- `git clone`: If you’re starting fresh and want to clone an existing repository from GitHub to your local computer, you can use this command to copy the repository to your computer.
- `git remote`: When you clone a repository from GitHub, it automatically creates a connection or ”remote” that you can also push changes to.

Example: Contribute to an existing repository

```
# download a repository on GitHub to our machine
# Replace `owner/repo` with the owner and name of the repository to clone
git clone https://github.com/owner/repo.git

# change into the `repo` directory
cd repo

# create a new branch to store any new changes
git branch my-branch

# switch to that branch (line of development)
git checkout my-branch

# make changes, for example, edit `file1.md` and `file2.md` using the text editor

# stage the changed files
git add file1.md file2.md

# take a snapshot of the staging area (anything that's been added)
git commit -m "my snapshot"

# push changes to github
git push --set-upstream origin my-branch
```

Example: Start a new repository adn publish on GitHub

```
# create a new directory, and initialize it with git-specific functions
git init my-repo

# change into the `my-repo` directory
cd my-repo

# create the first file in the project
touch README.md

# git isn't aware of the file, stage it
git add README.md

# take a snapshot of the staging area
git commit -m "add README to initial commit"

# provide the path for the repository you created on github
```

```
git remote add origin https://github.com/YOUR-USERNAME/YOUR-REPOSITORY-NAME.git

# push changes to github
git push --set-upstream origin main
```

Example: Contribute to an existing branch on GitHub

```
# change into the `repo` directory
cd repo

# update all remote tracking branches, and the currently checked out branch
git pull

# change into the existing branch called `feature-a`
git checkout feature-a

# make changes, for example, edit `file1.md` using the text editor

# stage the changed file
git add file1.md

# take a snapshot of the staging area
git commit -m "edit file1"

# push changes to github
git push
```

1.1.5 Models for collaborative development

THere are two primary ways people collaborate on GitHub:

- **Fork and pull model**: A project owner creates a master repository, and contributors fork that repository to their accounts. They clone the repository to their local machine, make changes, commit them
- **Shared repository model**: In this model, all collaborators have push access to the same repository. This is more common for small teams and organizations.

With a shared repository, individuals and teams are explicitly designated as contributors with read, write, or administrator access. This simple permission structure, combined with features like protected branches, helps teams progress quickly when they adopt GitHub.

For an open source project, or for projects to which anyone can contribute, managing individual permissions can be challenging, but a fork and pull model allows anyone who can view the project to contribute. A fork is a copy of a project under a developer's personal account. Every developer has full control of their fork and is free to implement a fix or a new feature. Work completed in forks is either kept separate, or is surfaced back to the original project via a pull request. There, maintainers can review the suggested changes before they're merged.

1.2 Pushing commits to a remote repository

1.2.1 About `git push`

The `git push` command takes two arguments.

- A remote name, for example, `origin`
- A branch name, for example, `main`

For example, the command `git push origin main` pushes the commits in the local `main` branch to the remote repository named `origin`.

Example: ***Pushing changes to a remote repository***

```
# push changes to the remote repository
git push origin main
```

1.2.2 Renaming branches

To rename a branch, you'd use the same `git push` command, but you would add one more argument: the name of the new branch. For example:

Example: ***Renaming a branch***

```
# rename the local branch to new-name
git push origin main:new-name
```

This pushes the `main` branch to the remote repository, but the branch is renamed to `new-name`.

Tip:

- If you want to delete a branch, you can use the `--delete` flag with the `git push` command. For example, `git push origin --delete new-name` will delete the `new-name` branch from the remote repository.
- If you want to rename the branch you're currently on, you can use the `-u` flag to set the upstream branch. For example, `git push -u origin main:new-name` will push the `main` branch to the remote repository, renaming it to `new-name`, and set the upstream branch to `new-name`.

1.2.3 Dealing with ”non-fast-forward” errors

If your local copy of a repository is out of sync with, or ”behind”, the upstream repository you’re pushing to, you’ll get a message saying `non-fast-forware updates were rejected`. This means that you must retrieve, or ”fetch,” the upstream changes, before you are able to push your local changes.

Tip: ***Fetching***

Fetching means retrieving recent commits from a remote repository without merging them into your local branch. This lets you view and compare new changes before deciding how to incorporate them into your work.

To fetch the changes from the remote repository, you can use the `git fetch` command. This command retrieves the changes from the remote repository, but it doesn’t merge them into your local branch. After fetching the changes, you can merge them into your local branch using the `git merge` command.

1.2.4 Pushing tags

By default, and without additional parameters, `git push` sends all matching branches that have the same names as remote branches.
To push a single tag, you cna issue the same command as pushing a branch:

```
git push origin tag-name
```

To push all your tags, you can type the command:

```
git push origin --tags
```

1.2.5 Deleting a remote branch or tag

The syntax to delete a branch is a bit more arcane at first glance:

```
git push origin --delete branch-name
```

Note that there is a space before the colon. The command resembles the same steps you’d take to rename a branch. However, here, you’re telling Git to push **nothing** into `branch-name`, effectively deleting it. Because of this, `git push` deletes the branch on the remote repository.

1.2.6 Remotes and forks

You can **fork a repository** on GitHub.
When you clone a repository you own, you provide it with a remote URL that tells Git where to fetch and push updates. If you want to collaborate with the original repository, you’d add a new remote URL, typically called `upstream`, to your local Git clone:

```
git remote add upstream REMOTE_URL
```

Now you can fetch updates and branches from their fork:

```
git fetch upstream
# Grab the upstream remote's branches
> remote: Counting objects: 75, done.
> remote: Compressing objects: 100% (53/53), done.
> remote: Total 62 (delta 27), reused 44 (delta 9)
> Unpacking objects: 100% (62/62), done.
> From https://github.com/OCTOCAT/REPO
>  * [new branch]      main     -> upstream/main
```

When you're done making local changes, you can push your local branch to GitHub and initiate a pull request.

1.3 Getting changes from a remote repository

1.3.1 Options for getting changes

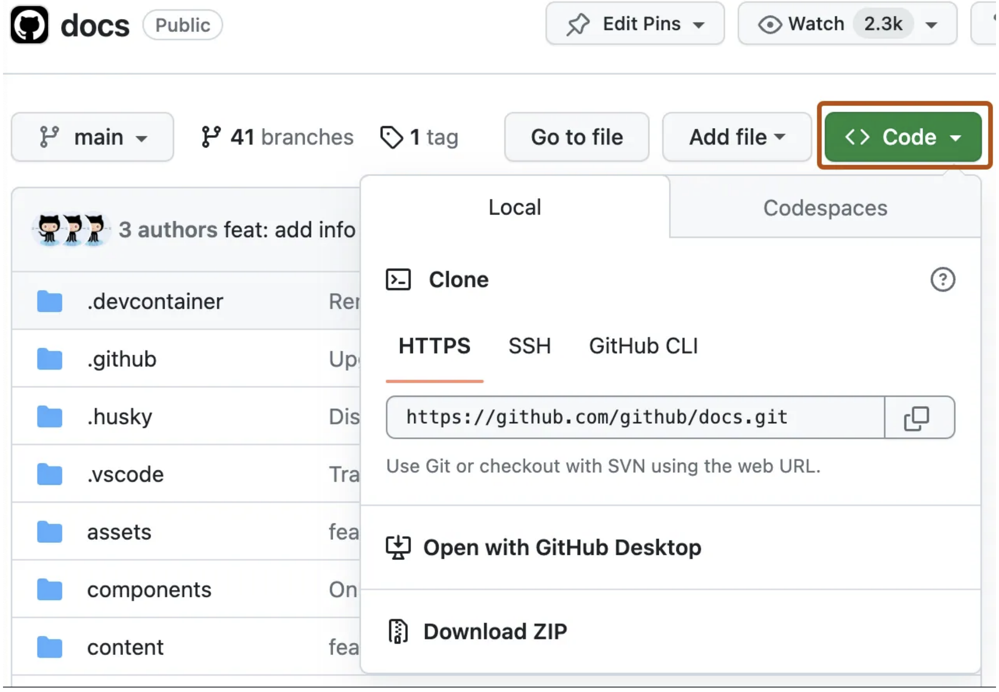
These commands are very useful when interacting with a remote repository. `clone` and `fetch` download remote code from a repository's remote URL to your local computer, `merge` is used to merge different people's work together with yours, and `pull` is a combination of `fetch` and `merge`.

1.3.2 Cloning a repository

To grab a complete copy of another user's repository, use `git clone` like this:

```
git clone https://github.com/USERNAME/REPOSITORY.git
# Clones a repository to your computer
```

You can choose from different URLs when cloning a repository. While logged in to GitHub, there URLs are available on the main page of the repository when you click `<> Code`.



```
https://github.com/github/docs.git
```

Use Git or checkout with SVN using the web URL.

When you run `git clone`, the following actions occur:

- A new folder called `repo` is made
- It is initialized as a Git repository
- A remote named `origin` is created, pointing to the URL you cloned from
- All of the repository's files and commits are downloaded there
- The default branch is checked out

For every branch `foo`, a corresponding remote-tracking branch `refs/remotes/origin/foo` is created in your local repository. You can usually abbreviate such remote-tracking branch names to `origin/foo`.

1.3.3 Fetching changes from a remote repository

Use `fit fetch` to retrieve new work done by other people. Fetching from a repository grabs all the new remote-tracking branches and tags without merging those changes into your own branches. If you already have a local repository with a remote URL set up for the desired project, you can grab all the new information by using `fit fetch *remotename*` in the terminal.

```
git fetch REMOTE-NAME
# Fetches updates made to a remote repository
```

1.3.4 Merging changes into your local branch

Merging combines your local changes with changes made by others.
Typically, you'd merge a remote-tracking branch (i.e., a branch fetched from a remote repository) with your local branch:

```
git merge REMOTE-NAME/BRANCH-NAME
# Merges updates made online with your local work
```

1.3.5 Pulling changes from a remote repository

`git pull` is a convenient shortcut for completing both `git fetch` and `git merge` in the same command:

```
git pull REMOTE-NAME BRANCH-NAME
# Grabs online updates and merges them with your local work
```

Because `pull` performs a merge on the retrieved changes, you should ensure that your local work is committed before running the `pull` command. If you run into a **merge conflict** you cannot resolve, or if you decide to quit the merge, you can use `git merge --abort` to take the branch back to where it was in before you pulled.

1.4 Dealing with non-fast-forward errors

Sometimes, Git can't make your change to a remote repository without losing commits. When this happens, your push is refused.

If another person has pushed to the same branch as you, Git won't be able to push your changes:

Example:

```
git push origin main
> To https://github.com/USERNAME/REPOSITORY.git
>  ! [rejected]        main -> main (non-fast-forward)
> error: failed to push some refs to 'https://github.com/USERNAME/REPOSITORY.git'
> To prevent you from losing history, non-fast-forward updates were rejected
> Merge the remote changes (e.g. 'git pull') before pushing again. See the
> 'Note about fast-forwards' section of 'git push --help' for details.
```

You can fix this by fetching and merging the changes made on the remote branch with the changes that you have made locally:

```
git fetch origin
# Fetches updates made to an online repository

git merge origin YOUR_BRANCH_NAME
# Merges updates made online with your local work
```

Or simply use `git pull` to perform both commands at once:

```
git pull origin YOUR_BRANCH_NAME
# Grabs online updates and merges them with your local work
```

1.5 Splitting a subfolder out into a new repository

You can turn a folder within a Git repository into a brand new repository.

Warning:

You need Git version 2.22.0 or later to follow these instructions, otherwise `git filter-repo` will not work.

If you create a new clone of the repository, you won't lose any of your Git history or changes when you split a folder into a separate repository. However, note that the new repository won't have the branches and tags of the original repository.

Steps:

1. Open Terminal.
2. Change the current working directory to the location where you want to create your new repository.
3. Clone the repository that contains the subfolder.

```
git clone https://github.com/USERNAME/REPOSITORY-NAME
```

4. Change the current working directory to your cloned repository.

```
cd REPOSITORY-NAME
```

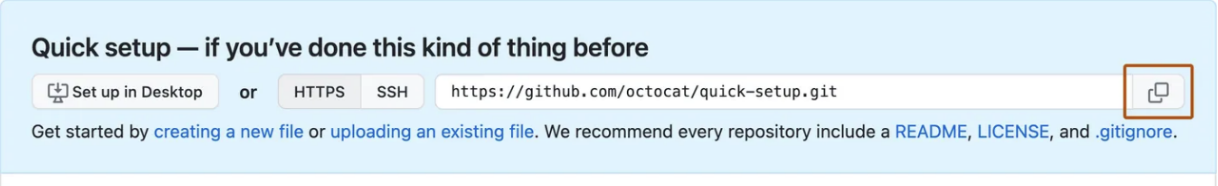
5. To filter out the subfolder from the rest of the files in the repository, install `git-filter-repo`, then run `git filter-repo` with the following arguments.
   - `FOLDER-NAME`: The folder within your project where you'd like to create a separate repository.

```
git filter-repo --path FOLDER-NAME/
# Filter the specified branch in your directory and remove empty commits
```

The repository should now only contain the files that were in your subfolder(s).
If you want one specific subfolder to be the new root folder of the new repository, you can use the following command:

```
git filter-repo --subdirectory-filter FOLDER-NAME/
# Filter the specified branch in your directory and remove empty commits
```

6. Create a repository on GitHub.
7. At the top of your new repository on GitHub's Quick Setup page, click □□ to copy the remote repository URL.



Quick setup — if you've done this kind of thing before

https://github.com/octocat/quick-setup.git

Get started by creating a new file or uploading an existing file. We recommend every repository include a README, LICENSE, and .gitignore.

Tip:

For information on the difference between HTTPS and SSH URLs, see About remote repositories.

8. Add a new remote name with the URL you copied for your repository. For example, `origin` or `upstream` are two common choices.

```
git remote add NEW-REMOTE-NAME URL
```

9. Verify that the remote URL was added with your new repository name.

```
git remote -v
# Verify new remote URL
> origin  https://github.com/USERNAME/NEW-REPOSITORY-NAME.git (fetch)
> origin  https://github.com/USERNAME/NEW-REPOSITORY-NAME.git (push
    )
```

10. Push your changes to the new repository on GitHub.

```
git push -u origin BRANCH-NAME
```

1.6 About Git subtree merges

Typically, a subtree merge is used to contain a repository within a repository. The ”subrepository” is stored in a folder of the main repository.

The best way to explain subtree merges is to show by example. We will:

- Make an empty repository called `test` that represents our project.
- Merge another repository into it as a subtree called `Spoon-Knife`.
- The `test` project will use that subproject as if it were part of the same repository.
- Fetch updates from `Spoon-Knife` into our `test` project.

1.6.1 Setting up the empty repository for a subtree merge

1. Open Terminal.
2. Create a new directory and navigate to it.

```
mkdir test
cd test
```

3. Initialize the directory as a Git repository.

```
git init
> Initialized empty Git repository in /Users/octocat/tmp/test/.git/
```

4. Create and commit a new file.

```
touch .gitignore
git add .gitignore
git commit -m "initial commit"
> [main (root-commit) 3146c2a] initial commit
>  0 files changed, 0 insertions(+), 0 deletions(-)
>  create mode 100644 .gitignore
```

1.6.2 Adding a new repository as a subtree

1. Add a new remote URL pointing to the separate project that we’re interested in.

```
git remote add -f spoon-knife https://github.com/octocat/Spoon-
    Knife.git
> Updating spoon-knife
> warning: no common commits
> remote: Counting objects: 1732, done.
> remote: Compressing objects: 100% (750/750), done.
> remote: Total 1732 (delta 1086), reused 1558 (delta 967)
> Receiving objects: 100% (1732/1732), 528.19 KiB | 621 KiB/s, done.
> Resolving deltas: 100% (1086/1086), done.
> From https://github.com/octocat/Spoon-Knife
>  * [new branch]      main     -> Spoon-Knife/main
```

2. Merge the `Spoon-Knife` project into the local Git project. This does not change any of your files locally, but it does prepare Git for the next step.
   If you're using Git 2.9 or above:

   ```
   git merge -s ours --no-commit --allow-unrelated-histories spoon-knife/main
   > Automatic merge went well; stopped before committing as requested
   ```

   If you're using Git 2.8 or below:

   ```
   git merge -s ours --no-commit spoon-knife/main
   > Automatic merge went well; stopped before committing as requested
   ```

3. Create a new directory called **spoon-knife**, and copy the Git history of the `Spoon-Knife` project into it.

   ```
   git read-tree --prefix=spoon-knife/ -u spoon-knife/main
   > fatal: refusing to merge unrelated histories
   ```

4. Commit the changes to keep them safe.

   ```
   git commit -m "Subtree merged in spoon-knife"
   > [main fe0ca25] Subtree merged in spoon-knife
   ```

Although we've only added one subproject, any number of subprojects can be incorporated into a Git repository.

Tip:

If you create a fresh clone of the repository in the future, the remotes you've added will not be created for you. You will have to add them again using the `git remote add` command.

1.6.3 Synchronizing with updates and changes

When a subproject is added, it is not automatically kept in sync with the upstream changes. You will need to update the subproject with the following command:

```
git pull -s subtree REMOTE-NAME BRANCH-NAME
```

For the example above, this would be:

```
git pull -s subtree spoon-knife main
```

1.7 About Git rebase

Typically, you would use `git rebase` to:

- Edit previous commit messages
- Combine multiple commits into one
- Delete or revert commits that are no longer necessary

> **⚠ Warning:**
>
> Because changing your commit history can make things difficult for everyone else using the repository, it's considered bad practice to rebase commits when you've already pushed to a repository. To learn how to safely rebase, see About pull request merges.

1.7.1 Rebasing commits against a branch

To rebase all the commits between another branch and the current branch state, you can enter the following command in your shell (either the command prompt for Windows, or the terminal for Mac and Linux):

```
git rebase --interactive OTHER-BRANCH-NAME
```

1.7.2 Rebasing commits against a point in time

To rebase the last few commits in your current branch, you can enter the following command in your shell:

```
git rebase --interactive HEAD~7
```

1.7.3 Commands available while rebasing

There are six commands available while rebasing:

- `pick`
  simply means that the commits is included. Rearranging the order of the `pick` commands changes the order of the commits when the rebase is underway. If you choose not to include a comit, you should delete the entire line.
- `reword`
  it is similar to `pick`, but after you use it, the rebase process will pause and give you a chance to alter the commit message. Any changes made by the commit are not affected.
- `edit`
  if you choose to `edit` a commit, you'll be given the chance to amend the commit, meaning that you can add or change the commit entirely. you can also make more commits before you continue the rebase. This allows you to split a large commit into smaller ones, or, remove erroneus changes made in a commit.
- `squash`
  it lets you combine two or more commits into a single one. A commit is squashed into the commit above it. Git gives you the chance to write a new commit message describing both changes.
- `fixup`
  similar to `squash`, but the commit to be merged has its message discarded. The commit is

simply merged into the commit above it, and the earlier commit's message is used to describe both changes.

- `exec`
  This lets you run arbitrary shell commands against a commit.

1.7.4 An example of using `git rebase`

No matter which command you use, Git will launch your default editor and open a file that details the commits in the range you've chosen. That file looks something like this:

Example:

```
pick 1fc6c95 Patch A
pick 6b2481b Patch B
pick dd1475d something I want to split
pick c619268 A fix for Patch B
pick fa39187 something to add to patch A
pick 4ca2acc i cant' typ goods
pick 7b36971 something to move before patch B

# Rebase 41a72e6..7b36971 onto 41a72e6
#
# Commands:
#  p, pick = use commit
#  r, reword = use commit, but edit the commit message
#  e, edit = use commit, but stop for amending
#  s, squash = use commit, but meld into previous commit
#  f, fixup = like "squash", but discard this commit's log message
#  x, exec = run command (the rest of the line) using shell
#
# If you remove a line here THAT COMMIT WILL BE LOST.
# However, if you remove everything, the rebase will be aborted.
#
```

Breaking this information from top to bottom, we see that:

- Seven commits are listed, which indicates that there were seven changes between out starting point and our current branch state.
- The commits you chose to rebase are sorted in the order of the oldest changes (at the top) to te newest changes (at the botom).
- Each line lists a command (by default, `pick`), the commit SHA, and the commit message. The entire `git rebase` procedure centers around your manipulation of these three columns. The changes you make are rebased onto your repository.
- After the commits, Git tells you the range of commits we're working with (`41a72e6..7b36971`).
- Finally, Git gives some help by telling you the commands that are available to you when rebasing commits.

1.8 Using Git rebase on the command line

Here, all of the `git rebase` commands available, exept for `exec`, are covered.

We'll start our rebase by entering `git rebase --interactive HEAD 7` on the terminal. Our favourite text editor will display the following lines:

```
pick 1fc6c95 Patch A
pick 6b2481b Patch B
pick dd1475d something I want to split
pick c619268 A fix for Patch B
pick fa39187 something to add to patch A
pick 4ca2acc i cant' typ goods
pick 7b36971 something to move before patch B
```

In this example, we're going to:

- Squash the fifth commit (`fa39187`) into the `"Patch A"` commit (`1fc6c95`), using `squash`.
- Move the last commit (`7b36971`) up before the `"Patch B"` commit (`6b2481b`), and keep it as `pick`.
- Merge the `"A fix for Patch B"` commit (`c619268`) into the `"Patch B"` commit (`6b2481b`), and disregard the commit message using `fixup`.
- Split the third commit (`dd1475d`) into two smaller commits, using `edit`.
- Fix the commit message of the misspelled commit (`4ca2acc`), using `reword`.

This sounds like a lot of work, but by taking it one step at a time, we can easily make those changes.

To start, we'll need to modify the commands in the file to look like this:

```
pick 1fc6c95 Patch A
squash fa39187 something to add to patch A
pick 7b36971 something to move before patch B
pick 6b2481b Patch B
fixup c619268 A fix for Patch B
edit dd1475d something I want to split
reword 4ca2acc i cant' typ goods
```

We've changed each line's command from `pick` to the command we're interested in.

Now, save and close the editor; this will start the interactive rebase.

Git skips the first rebase command, `pick 1fc6c95`, since it does not need to do anything. It goes to the next command, `squash fa39187`. Since this operation requires your input, Git opens your text editor once again. The file it opens up looks something like this:

```
# This is a combination of two commits.
# The first commit's message is:

Patch A

# This is the 2nd commit message:

something to add to patch A

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# Not currently on any branch.
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
# modified:   a
#
```

This file is Git's way of saying, ”Hey, here's what I'm about to do with this `squash`.”It lists the first commit's message (`"Patch A"`), and the second commit's message (`"something to add to patch A"`). If you're happy with these commit messages, you can save the file, and close the editor. Otherwise, you have the option of changing the commit message by simply changing the text.

When the editor is closed, the rebase continues:

```
ù
pick 1fc6c95 Patch A
squash fa39187 something to add to patch A
pick 7b36971 something to move before patch B
pick 6b2481b Patch B
fixup c619268 A fix for Patch B
edit dd1475d something I want to split
reword 4ca2acc i cant' typ goods
```

Git processes the two `pick` commands (for `pick 7b36971` and `pick 6b2481b`). It also processes the `fixup` command (`fixup c619268`), since it does not require any interaction. `fixup` merges the changes from `c619268` into the commit before it, `6b2481b`. Both changes will have the same commit message: `"Patch B"`.

Git gets to the `edit dd1475d` operation, stps, and prints the following message to the terminal:

```
You can amend the commit now, with

    git commit --amend

Once you are satisfied with your changes, run

    git rebase --continue
```

At this pount, you can edit eny of the files in your project to make any additional changes. For each change you make, you'll need to perform a new commit, and you can do that by entering the `git commit --amend` command. When you're finished making all your changes, you can run `git rebase --continue`.

Git then gets to the `reword 4ca2acc` command. It opens up your text editor one more time, and presents, the following information:

```
i cant' typ goods

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# Not currently on any branch.
# Changes to be committed:
```

```
#   (use "git reset HEAD^1 <file>..." to unstage)
#
# modified:   a
#
```

As before, Git is showing the commit message for you to edit. You can change the text (`"i can't typ goods"`), save the file, and close the editor. Git will finish the rebase and return you to the terminal.

1.8.1 Pushing rebased code to GitHub

Since you've altered Git history, the usual `git push origin` will not work. You'll need to modify the command by "force-pushing" your latest changes:

```
# Don't override changes
git push origin main --force-with-lease

# Override changes
git push origin main --force
```

⚠ **Warning:**

Force pushing has serious implications because it changes the historical sequence of commits for the branch. Use it with caution, especially if your repository is being accessed by multiple people.

1.9 Resolving merge conflicts after a Git rebase

When you perform a `git rebase` operation, you're typically moving commits around. Because of this, you might get into a situation where a merge conflict is introduced. That means that two of your commits modified the same line in the same file, and Git does not know which change to apply.

After you reorder and manipulate commits using `git rebase`, should a merge conflict occur, Git will tell you so whith the following message in the terminal:

```
error: could not apply fa39187... something to add to patch A

When you have resolved this problem, run "git rebase --continue".
If you prefer to skip this patch, run "git rebase --skip" instead.
To check out the original branch and stop rebasing, run "git rebase --abort".
Could not apply fa39187f3c3dfd2ab5faa38ac01cf3de7ce2e841... Change fake file
```

Here, Git is telling you which commit is causing the conflict (`fa39187`). You're given three choices:

- You can run `git rebase --abort` to completely undo the rebase. Git will return you to your branch's state as it was before `git rebase` was called.
- You can run `git rebase --skip` to completely skip the commit. That means that none of the changes introduced by the problematic commit will be included. It is very rare that you would

choose this option.

- You can fix the conflict.

To fix the conflict, you can follow the standard procedures for resolving merge conflicts from the command line. When you're finished, you'll need to call `git rebase --continue` in order for Git to continue processing the rest of the rebase.

1.10 Dealing with special characters in branch and tag names

Most repositories use simple branch names, such as `main` or `update-icons`. Tag names also usually follow a basic format, such as a version number like `v1.2.3`. Both branch names and tag names may also use the path separator (`/`) for structure, for example `area/item` or `level-1/level-2/level-3`. Other than some exceptions - such as not starting or ending a name with slash, or having consecutive slashes in the name - Git has very few restrictions on what characters may be used in branch and tag names.

1.10.1 Why you need to escape special characters

When using a CLI, you might have situations where a branch or tag name contains special characters that have a special meaning for your shell environment. To use these characters safely in a Git command, they must be quoted or escaped, otherwise the command may have unintended effects. For example, the `$` character is used by many shells to refer to a variable. Most shells would interpret a valid branch name like `hello-$USER` as equivalent to the word "hello", followed by a hyphen, followed by the current value of the `USER` variable, rather than the literal string `hello-$USER`. If a branch name includes the `$` character, then the shell must be stopped from expanding it as a variable reference. Similarly, if a branch name contains a semi-colon (`;`), most shells interpret it as a command separator, so it needs to be quoted or escaped.

1.10.2 How to escape special characters in branch and tag names

Most branch and tag names with special characters can be handled by including the name in single quotes, for example `h́ello-$USEŔ`.

- In the **Bash** shell, enclosing a string of characters in single quotes preserves the literal value of the characters within the single quotes.
- **Zsh** behaves similar to Bash, however this behavior is configurable using the `RC_QUOTES`
- **PowerShell** also treats characters literally when inside single quotes.

For these shells, the main exception is when the branch or tag name itself contains a single quote. In this case, you should consult the official documentation for your shell.

1.10.3 Naming branches and tags

If possible, create branch and tag names that don't contain special characters, as these would need to be escaped. A safe default set of characters to use for branch names and tag names is:

- The English alphabet(`a` to `z` and `A` to `Z`)
- Numbers (`0` to `9`)
- A limited set of punctuation characters:
  - period (`.`)
  - hyphen (`-`)
  - underscore (`_`)

  - forward slash (`/`)

To avoid confusion, you should start branch names with a letter.

1.10.4 Restrictions on names in GitHub

GitHub restricts a small number of branch and tags names from being pushed up. Those restrictions are:

- No names which look like Git object IDs (40 characters containing only 0-9 and A-F), to prevent confusion with actual Git object IDs.
- No names beginning with `refs/` , to prevent confusion with the full name of Git refs.

1.11 Troubleshooting the 2GB push limit

GitHub has a maximum 2GB limit for a single push. You might hit this limit when trying to upload very large repositories for the first time, importing large repositories from other platforms, or when trying to rewrite the history of large existing repositories.

If you hit this limit, you may see one of the following error messages:

- `fatal: the remote end hung up unexpectedly`
- `remote: fatal: pack exceeds maximum allowed size`

You can either split up your push into smaller parts, or delete the Git history and start from scratch. If you have made a single commit that's larger than 2 GB and you can't delete the Git history and start from scratch, then you will need to perform an interactive rebase to split the large commit into multiple smaller ones.

1.11.1 Splitting up a large push

You can avoid hitting the limit by breaking your push into smaller parts, each of which should be under 2 GB in size. If a branch is within this size limit, you can push it all at once. However, if a branch is larger than 2 GB, you'll need to split the push into even smaller portions and push only a few commits at a time.

1. If you haven't configured the remote yet, add the repository as a new remote.
2. To find suitable commits spread out along the history of the main branch in your local repository, run the following command:

   ```
   git log --oneline --reverse refs/heads/BRANCH-NAME | awk 'NR \% 1000 == 0'
   ```

   This command reveals every 1000th commit. You can increase or decrease the number to adjust the step size.
3. Push each of these commits one at a time to your GitHub hosted repository.

   ```
   git push REMOTE-NAME +<YOUR\_COMMIT\_SHA\_NUMBER>:refs/heads/BRANCH-NAME
   ```

   If you see the message `remote: fatal: pack exceeds maximum allowed size` , reduce the step size in step 2 and try again.
4. Go through the same process for every commit you identified in the history from step 2.
5. If this is the first time this repository is being pushed to GitHub, perform a final mirror push to ensure any remaining refs are pushed up.

```
git push REMOTE-NAME --mirror
```

If this is still too large, you'll need to push up other branches in stages using the same steps. Once you're familiar with the procedure, you can automate steps 2 to 4 to simplify the process. For example:

```
step_commits=\$(git log --oneline --reverse refs/heads/BRANCH-NAME | awk 'NR \% 1000 == 0')
echo "\$step\_commits" | while read commit message; do git push REMOTE-NAME +\$commit:refs/heads/BRANCH-NAME; done
```

1.11.2 Starting from scratch

If the repository does not have any history, or your initial commit was over 2 GB on its own and you don't mind resetting the Git history, you can also start from scratch.

1. On your local copy, delete the hidden `.git` folder to remove all the previous Git history and convert it back into a normal folder full of files.
2. Create a new empty folder.
3. Run `git init` and `git lfs install` on the new folder, and add the new empty GitHub repository as a remote.
4. If you already use Git Large File Storage and have all of the Git LFS tracking rules you intend to use already listed in the `.gitattributes` file in the old folder, that should be the first file you copy across to the new folder. You should ensure the tracking rules are in place before you add any other files, so that there's no chance things intended for Git LFS will be committed to regular Git storage.
   If you do not already use Git LFS, you can skip this step, or you can set up the tracking rules you intend to use in the `.gitattributes` file in the new folder before you copy any other files across.
5. Move batches of files that are smaller than 2 GB from the old folder to the new folder. After each batch is moved, create a commit and push it before moving the next batch. You can take a cautious approach and stick to around 2 GB. Alternatively, if you have a folder with files meant for Git LFS, you can ignore those files when considering the 2 GB limit per batch.

Once the old folder is empty, the GitHub repository should contain everything. If you are using Git LFS, all files meant for Git LFS should be pushed to Git LFS storage.

2 PyTorch

PyTorch (PT) is a Python (and C++) library for Machine Learning (ML) particularly suited for Neural Networks and their applications.

Its great selection of built-in modules, models, functions, CUDA capability, tensor arithmetic support and automatic differentiation functionality make it one of the most used scientific libraries for Deep Learning.

Note: for this series of labs, we advise to install Python >= 3.7

We advise to install PyTorch following the directions given in its home page. Just typing `pip install torch` mat not be the correct action as you have to take into account the compatibility with `cuda`. If you have `cuda` installed, you can find your version by typing `nvcc --version` in your terminal (Linux/iOS).

If you're using Windows, we first suggest to install Anaconda and then install PyTorch from the `anaconda prompt` software via `conda` (preferably) or `pip`.

If you're using Google Colab, all the libraries needed to follow this lecture should be pre-installed there.

Observation: ***For Colab users***

Google Colab is a handy tool that we suggest you use for this course—especially if your laptop does not support CUDA or has limited hardware capabilities. Anyway, note that we'll try to avoid GPU code as much as possible. Essentially, Colab renders available to you a virtual machine with a limited hardware capability and disk where you can execute your code inside a given time window. You can even ask for a GPU (if you use it too much you'll need to start waiting a lot before it's available though).

Some commands:

```
# Run shell commands using !
!ls
!pwd
!cd ..
!git clone https://github.com/...
```

You can also transfer files on Colab. Since your files reside on the virtual machine, there're two ways to operate file transfer on Colab:

1. You can upload files from your local machine to the virtual machine by clicking on the folder icon on the left side of the screen.
2. You can use the following code to transfer files from your Google Drive to the virtual machine:

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

Pytorch is a numerical library that makes it very convenient to train deep networks on GPU hardware.

It introduces a new programming vocabulary that takes a few steps beyond regular numerical python code. Although pytorch code can look simple and concrete, much of of the subtlety of what happens is invisible, so when working with pytorch code it helps to thoroughly understand the runtime model.

The berevity of the code is what makes pytorch code fun to write. But it also reflects why pytorch can be so fast even though the python interpreter is so slow. Although the main python logic slogs along sequentially in a single very slow CPU thread, just a few python instructions can load a huge amount of work into the GPU. That means the program can keep the GPU busy churning through massive numerical computations, for most part, without waiting for the python interpreter.

Is is worth understanding five core idioms that work together to make this possible. This tutorial has five sections, one for each topic:

1. **GPU Tensor arithmetic**: the notation for manipulating n-dimensional arrays of numbers on CPU or GPU.
2. **Autograd**: how to build a tensor computation graph and use it to get derivatives of any scalar with respect to any input.
3. **Optimization**: how to use the autograd derivatives to optimize the parameters of a neural network.
4. **Network modules**: ways to update tensor parameters to reduce any computed objective, using autograd gradients.
5. **Dataset and Dataloaders**: for efficient multithreaded prefetching of large streams of data.

2.1 GPU Tensor Arithmetic

The first big trick for doing math fast on a modern computer is to do giant array operations all at once. To faciliate this, PyTorch provides its own multidimensional array class, called **Tensor**. They are essentially the equivalent of NumPy **ndarray**s. The main difference is that PyTorch tensors can be used on a GPU to accelerate computing. Almost all the numpy operations are also available on torch tensors. But if something is missing, torch tensors can be directly converted to and from numpy using `x.numpy()` and `torch.from_numpy()`. So what is different and why did the pytorch authors bother to reimplement this whole library?

- PyTorch Tensors can live on their GPU or CPU (numpy is CPU only)
- PyTorch can automatically track tensor computations to enable **automatic differentiation**

Example: ***Tensors***

```python
import torch
import numpy as np

# Create a tensor
x = torch.tensor([1, 2, 3])

# Create a tensor from a numpy array
y = torch.from_numpy(np.array([1, 2, 3]))

# Check the type
print(x, y)
print(x.dtype, y.dtype)
```

Observation: ***Tensor data type***

The default data type for a tensor is `torch.float32`, thus it implicitly converts data to this type. You can change it by using the `dtype` argument.

PyTorch is not so different from numpy, although the PyTorch API has more convenience methods such as `x.clamp(0).pow(2)`. So code is often shorter in PyTorch.

- **Elementwise operations**: Most tensor operations are simple (embarassingly parallelizable) elementwise operations, where the same math is done on every element of the array `x+y`, `x*y`, `x.abs()`, ...
- **Copy semantics by default**: Almost every operation return a new copy of the tensor without overwriting the input tensor. The exceptions are functions that end in an underscore such as `x.mul_(2)` which doubles the contents of x in-place
- **Common reduction operations**: There are some common operations such as `max`, `min`, `mean`, ... that reduce the array by one or more dimension. In PyTorch, you can specify which dimension you want to reduce by passing the argument `dim=n`
- **Why does min return two things?** Note that `[data, indexes] = x.sort(dim=0)` and `[vals, indexes] = x.min(dim=0)` return the pair of both the answer and the index values, so you do not need to separately recompute `argsort` or `argmin` when you need to know where the min came from.
- **What about linear algebra?** It's there. `torch.mm(a,b)` is matrix multiplication, `torch.inverse(a)` inverts, `torch.eig(a)` gets eigenvalues, etc.

The other thing to know is that pytorch tends to be very fast, often much faster than numpy even on CPU, because its implementation is aggressively parallelized behind-the-scenes. Pytorch is willing to use multiple threads in situations where numpy just uses one.

As in NumPy, we can call the `.shape` attribute to get the shape of the tensor. Moreover, Tensors have also the `.size()` method which is analogous to `.shape`.

```python
print(x.shape)
print(x.size())
```

Notice how a Tensor shape is not a tuple.

Create a random tensor:

```python
x = torch.rand(2, 3)
print(x)
```

Get the total number of elements in a tensor:

```python
print(x.numel())
```

Calculate the size of the Tensor within the RAM:

```python
print(x.element_size() * x.numel())
```

2.1.1 Slicing a Tensor

Slicing a tensor is similar to slicing a NumPy array. You can use the `[start:stop:step]` syntax to slice a tensor.

```python
x = torch.tensor([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print(x[0, 1])
print(x[0, :])

# Slicing with step
print(x[0, ::2])
```

2.1.2 Linear Algebra

PyTorch provides a wide range of linear algebra operations. For instance, you can calculate the dot product of two tensors using the `torch.dot()` function.

```python
x = torch.tensor([1, 2, 3])
y = torch.tensor([4, 5, 6])
print(torch.dot(x, y))
```

You can also calculate the matrix product of two tensors using the `torch.mm()` function or the `@` operator.

```python
x = torch.tensor([[1, 2], [3, 4]])
y = torch.tensor([[5, 6], [7, 8]])

print(torch.mm(x, y))
print(x @ y)
```

As for ndarrays, Tensors' arithmetic operations support **broadcasting**. Roughly speaking, when two Tensors have different shapes and a binary operation is performed, the smaller tensor is broadcasted to the larger tensor's shape. Of course, this is not always possible, but as a rule of thumb, if some dimensions of a Tensor are one and the other dimensions are the same, broadcasting works.

? **Example**: ***Broadcasting***

```python
x = torch.tensor([[1, 2], [3, 4]])
y = torch.tensor([5, 6])

print(x + y)
```

2.1.3 Reshaping and Permuting a Tensor

Sometimes it may be necessary to reshape the tensors to apply some specific operations. Take the example of RGB images: they can be seen as $3 \times h \times w$ tensors, where h and w are the height and width of the image, respectively. To apply a convolutional layer, we need to reshape the tensor to $h \times w \times 3$.

Sometimes it may be necessary to "flatten" the three matrices into vectors, thus obtaining a $3 \times h \times w$ tensor.

```python
image = torch.load('image.pt')
print(image.shape)
```

This flattening may be achieved via the `.reshape()` method.

```python
image = image.reshape(3, -1)
print(image.shape)
```

We can also use the `.view()` method to reshape a tensor.

```python
image = image.view(3, -1)
print(image.shape)
```

Observation: ***Difference between*** `.view()` ***and*** `.reshape()`

The `.view()` method returns a new tensor with the same data as the original tensor, but with a different shape. The `.reshape()` method returns a new tensor with the same data as the original tensor, but with a different shape. The difference is that `.view()` may return a view of the original tensor, while `.reshape()` always returns a new tensor.

To permute the dimensions of a tensor, we can use the `.permute()` method.

```python
image = image.permute(1, 0)
print(image.shape)
```

2.1.4 Conversion from NumPy to PyTorch

PyTorch provides a function to convert a NumPy array to a PyTorch tensor: `torch.from_numpy()`.

```python
import numpy as np

x = np.array([1, 2, 3])
y = torch.from_numpy(x)
print(y)
```

2.1.5 Using GPUs

All `Torch.Tensor` methods support GPU computation via built-in CUDA wrappers. Just transfer the involved `Tensor` and let the magic happen.

```python
x = torch.tensor([1, 2, 3])
y = torch.tensor([4, 5, 6])

if torch.cuda.is_available():
    x = x.cuda()
    y = y.cuda()

    z = x + y
    print(z)
```

2.1.6 Subscripts and multiple dimensions

Pytorch code is full of multidimensional arrays. The key to reading this kind of code is stopping to think about the careful, sometimes tangled, use of multiple array subscripts.

Slicing is used to slice ranges of elements from a tensor. The syntax is the same as for numpy arrays. For example, `x[0, 1:3]` would return the second and third elements of the first row of x. The general syntax is `x[start:stop:step]`. **Unsqueezing to add a dimension and broadcasting** is used to add a new dimension to a tensor. For example, if x is a 3x4 tensor, `x.unsqueeze(0)` would return a 1x3x4 tensor. This is useful when you want to add a dimension to a tensor to make it easier to broadcast with another tensor. For example, if x is a 3x4 tensor and y is a 1x4 tensor, `x + y` would not work because the dimensions are not compatible. But `x + y.unsqueeze(0)` would work because the dimensions are compatible after unsqueezing y. The general syntax is `x.unsqueeze(n)`. While a single integer subscript like `x[0]` eliminates a single dimension, the special subscript `x[None]` does the reverse and adds an extra dimension of size one. An extra dimension of size one is more useful than you might imagine, because PyTorch can combine different-shaped arrays as long as the shape differences appear only on dimensions of size one by **broadcasting** the singleton dimensions. An example that uses broadcasting to calculate an outer product is illustrated later. Moreover, **Fancy indexing** is used to select arbitrary elements from a tensor.

Example:

```python
import torch
from matplotlib import pyplot as plt

# Make an array of normally distributed randoms.
m = torch.randn(2, 5).abs()
print(f'm is {m}, and m[1,2] is {m[1,2]}\n')
print(f'column zero, m[:,0] is {m[:,0]}')
print(f'row zero m[0,:] is {m[0,:]}\n')
dot_product = (m[0,:] * m[1,:]).sum()
print(f'The dot product of rows (m[0,:] * m[1,:]).sum() is {dot_product}\n')
outer_product = m[0,:][None,:] * m[1,:][:,None]
print(f'The outer product of rows m[0,:][None,:] * m[1,:][:,None] is:\n{outer_product}')

fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(2, 2, figsize=(5, 5), dpi=100)
def color_mat(ax, m, title):
    ax.set_title(title)
    ax.imshow(m, cmap='hot', vmax=1.5, interpolation='nearest')
    ax.get_xaxis().set_ticks(range(m.shape[1]))
    ax.get_yaxis().set_ticks(range(m.shape[0]))
color_mat(ax1, m, 'm[:,:]')
color_mat(ax2, m[0,:][None,:], 'm[0,:][None,:]')
color_mat(ax3, m[1,:][:,None], 'm[1,:][:,None]')
color_mat(ax4, outer_product, 'm[0,:][None,:] * m[1,:][:,None]')
fig.tight_layout()
fig.show()
```

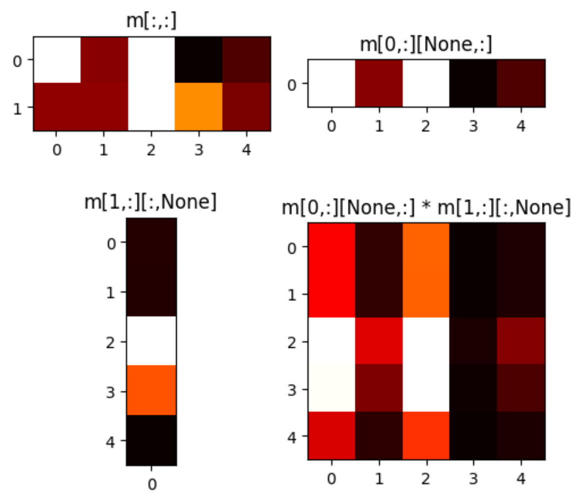


Figure 2.1: Outer product of two rows of a matrix

2.1.7 Devices and types

One of the big reasons to use pytorch instead of numpy is that pytorch can do computations on the GPU. But because moving data on and off of a GPU device is more expensive than keeping it within the device, pytorch treats a Tensor's computing device as pseudo-type that requires explicit declaration and explicit conversion. Here are some things to know about pytorch devices and types:

- **Single precision CPU default**: by default a torch tensor will be stored on the CPU and will store single-precision 32-bit `torch.float` values
- **Specifying data type**: to store a different data type such as integers, use the argument `dtype=torch.long` when you create the Tensor. As an example, `z = torch.zeros(10, dtype=torch.long)`.
- **Specifying GPU**: to store the tensor on the GPU, specify `device='cuda'` when you make it, for example `identity_matrix = torch.eye(5, device='cuda')`. Instead, `device='cpu'` indicates the default CPU storage.

  Even on a multi-GPU machine it is fine to pretend there is only one GPU. Setting the environment variable `CUDA_VISIBLE_DEVICES=3` before you start the program will set up the process to see GPU#3 as the only GPU. This is useful for debugging and for running multiple copies of the same program on the same machine.

  As an aside, in principle you could instead target one of many GPUs with `device='cuda:3'`, but if you want to use multiple GPUs for the same computation your best bet is to use a multiprocess utility class that manages data distribution between forked processes automatically, while each python process touches only one GPU.
- **Copying a tensor to a different device or type**: you cannot directly combine tensors that are on different devices (e.g. GPU vs CPU or different GPUs); this is similar to how most different data-type combinations are also prohibited. In both cases you will need to convert types and move devices explicitly to make tensors compatible before combining them. The `x.to(y.device)` or `x.to(y.type)` function can be used to do the conversion.

  There are also commonly-used convenience synonyms `x.cpu()`, `x.cuda()`, `x.float()`, `x.long()`, etc. for making a copy of x with the specified device or type. There is a bit of cost, so move data only when needed.

- **GPU rounding is nondeterministic**: computationally the GPU is not perfectly equivalent to the CPU. To speed parallelization, the GPU does not do associative operations such as summation in a deterministic sequential order. Since changing the order of summations can alter rounding behavior in fixed-precision arithmetic, GPU rounding can be different from CPU results an even nondeterministic. When the numerical algorithm is well-behaved, the difference should be small enough that you do not care, but you should know it is different.
- **Float is faster**: all commodity GPU hardware is fast at single-precision 32-bit floating point math, about 20x CPU speed. Be aware that only expensive cards are fast at 64-bit double-precision math. If you change `torch.float` in the below example to `torch.double` on an Nvidia Titan or consumer card without hardware double-precision support, you will slow down to just-slightly-faster-than-CPU speeds. Similarly 16-bit `torch.half` or `torch.bfloat16` or other cool options will only be faster on newer hardware, and with these data types you need to take care that reduced precision is not damaging your results.

  So `float` is the default and usually the best.

  Also note that some operations (like linear algebra) are floating-point only and cannot be done on integers.

  An example of some CPU versus GPU speed comparisons is below.

Example: *Speed comparison*

```python
import torch, time
from matplotlib import pyplot as plt

# Here is a demonstration of moving data between GPU and CPU.
# We multiply a batch of vectors through a big linear opeation 10 times
r = torch.randn(1024, 1024, dtype=torch.float)
x = torch.randn(32768, 1024, dtype=r.dtype)
iterations = 10

def time_iterated_mm(x, matrix):
    start = time.time()
    result = 0
    for i in range(iterations):
        result += torch.mm(matrix, x.to(matrix.device).t())
    torch.cuda.synchronize()
    elapsed = time.time() - start
    return elapsed, result.cpu()

cpu_time, cpu_result = time_iterated_mm(x.cpu(), r.cpu())
print(f'time using the CPU alone: {cpu_time:.3g} seconds')

mixed_time, mixed_result = time_iterated_mm(x.cpu(), r.cuda())
print(f'time using GPU, moving data from CPU: {mixed_time:.3g} seconds')

pinned_time, pinned_result = time_iterated_mm(x.cpu().pin_memory(), r.cuda())
print(f'time using GPU on pinned CPU memory: {pinned_time:.3g} seconds')

gpu_time, gpu_result = time_iterated_mm(x.cuda(), r.cuda())
print(f'time using the GPU alone: {gpu_time:.3g} seconds')

plt.figure(figsize=(4,2), dpi=150)
plt.ylabel('iterations per sec')
plt.bar(['cpu', 'mixed', 'pinned', 'gpu'],
        [iterations/cpu_time,
            iterations/mixed_time,
            iterations/pinned_time,
            iterations/gpu_time])
plt.show()

print(f'Your GPU is {cpu_time / gpu_time:.3g}x faster than CPU'
        f' but only {cpu_time / mixed_time:.3g}x if data is repeatedly copied from the CPU')
print(f'When copying from pinned memory, speedup is {cpu_time / pinned_time:.3g}x')
print(f'Numerical differences between GPU and CPU: {(cpu_result - gpu_result).norm() / cpu_result.norm()}')
```

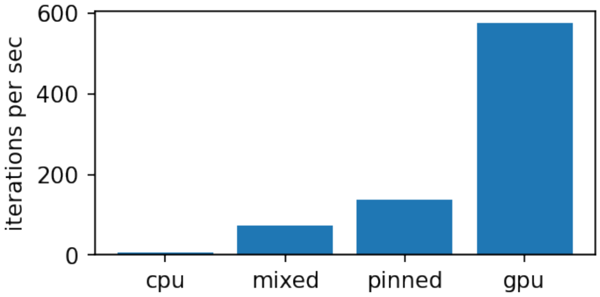


Figure 2.2: Speed comparison between CPU and GPU

2.1.8 Performance tips

GPU operations are async. When pytorch operates on GPU tensors, the python code does not wait for computations to complete. Sp GPU calculations get queued up, and they will be done as quickly as possible in the background while your python is free to work on other things like loading the next batch of training data.

Moving data to cpu waits for computations. You do not need to worry about the GPU asynchrony, because as soon as you actually ask to look at the data, e.g., when you move GPU data to CPU (or print it or save it), pytorch will block and wait for the GPU operations to finish computing what you need before proceeding. The call seen above to `torch.cuda.synchronize()` flushes the GPU queue without requesting the data, but you will not need to do this unless you are doing performance timing.

Pinned memory transfers are async and faster. Copying data from CPU to GPU can be sped up if the CPU data is put in pinned memory (i.e., at a fixed non-swappable block of RAM). Therefore when data loaders gather together lots of CPU data that is destined for the GPU, they should be configured to stream their results into pinned memory. See the performance comparison above.

2.1.9 PyTorch Tensor dimension-ordering conventions

Multidimensional data convension. As soon as you have more than one dimension, you need to decide how to order the axes. To reduce confusion, most data processing follows the same global convention. In particular, much image-related data in pytorch is four dimensional, and the dimensions are ordered like this: `data[batch_index, channel_index, y_position, x_position]`, that is:

- Dimension 0 is used to index separate images within a batch.
- Dimension 1 indexes channels within an image representation (e.g., 0,1,2 = R,G,B, or more dims for more channels).
- Dimension 2 (if present) indexes the row position (y-value, starting from the top)
- Dimension 3 (if present) indexes the column position (x-value, starting from the left)

There a way to remember this ordering: adjacent entries that vary only in the last dimensions are stored physically closer in RAM; since they are often combined with each other, this could help with locality, whereas the first (batch) dimension usually just groups separate independent data points which are not combined much, so they do not need to be physically close.

Stream-oriented data without grid geometry will drop the last dimensions, and 3d grid data will be 5-dimensional, adding a depth z before y. This same 4d-axis ordering convention is also seen in caffe and tensorflow.

Separate tensors can be put together into a single batch tensor using `torch.cat([a, b, c])` or `torch.stack([a, b, b])`.

Multidimensional linear operation convention. When storing matrix weights or a convolution weights, linear algebra conventions are followed:

- Dimension 0 (number of rows) matches the output channel dimension
- Dimension 1 (number of columns) matches the input channel dimension
- Dimension 2 (if present) is the convolutional kernel y-dimension
- Dimension 3 (if present) is the convolutional kernel x-dimension

Since this convention assumes channels are arranged in different rows whereas the data convention puts different batch items in different rows, some axis transposition is often needed before applying linear algebra to the data.

Permute and view reshape an array without moving memory. The `permute` and `view` methods are useful for rearranging, flattening and unflattening axes. `x.permute(1,0,2,3).view(x.shape[1],-1)`. They just alter the view of the block of numbers in memory without moving any of the numbers around, so they are fast.

Reshaping sometimes needs copying. Some sequences of axis permutations and flattenings cannot be done without copying the data into the new order in memory; the `x.contiguous()` method copies the data iinto the natural order given by the current view; also `x.reshape()` is similar to `view` but will makea copy if necessary so you do not need to think about it.

2.1.10 Einsum notation

Matrix multiplication can be generalized to tensors of arbitrary number of dimensions, but keeping tensor dimensions straight can be confusing. The solution to this is Einstein notation: assign letter variables to each axis of the input tensors, and then explicitly write down which axes end up in the output tensor. For example, an outer product might be written as `i, j → ij`, whereas matrix multiplication could be `ij, jk → ik`.

Einstein notation is a topic of active development and programming language design: here is a recent paper on the history and future of Einstein APIs.

In PyTorch, Einstein notation is available as `einsum`. Here is how ordinary matrix multiplication looks as einsum:

Example: ***Einsum***

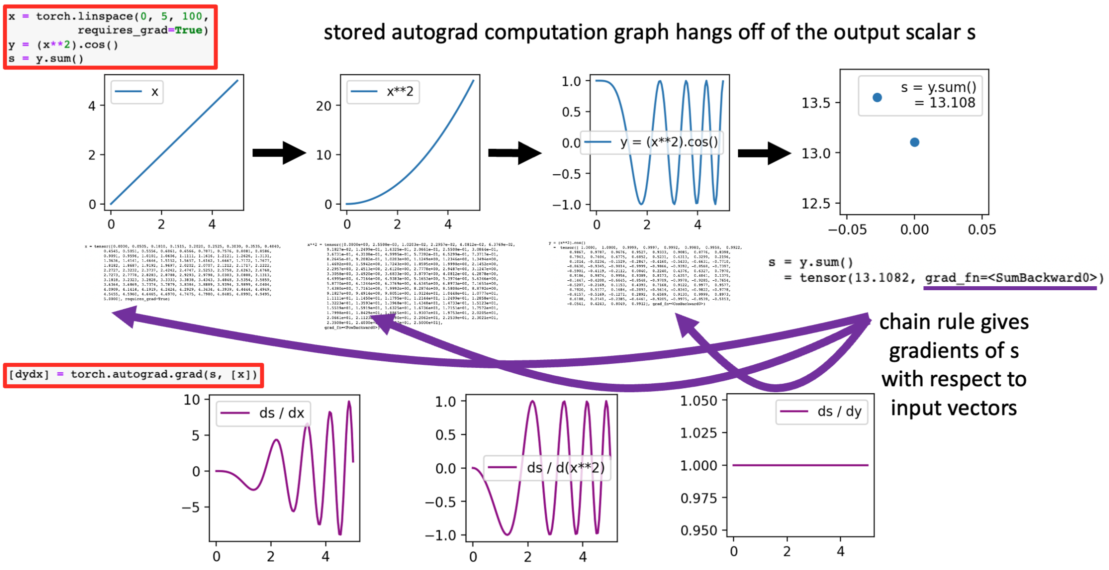
```python
A = torch.randn(2,5)
B = torch.randn(5,3)

# Uncomment to see ordinary matrix multiplication
# print(torch.mm(A, B))

# Ordinary matrix multiplication written as an einsum
print(torch.einsum('ij, jk -> ik', A, B))
```

2.2 Autograd

If you flag a torch Tensor with the attribute `x.requires_grad=True`, then PyTorch will automatically keep track the computational history of all tensors that are derived from `x`. This allows PyTorch to figure out derivatives of any scalar result with regard to changes in the components of x.



```
x = torch.linspace(0, 5, 100,
        requires_grad=True)
y = (x**2).cos()
s = y.sum()
```

```
s = y.sum()
  = tensor(13.1082, grad_fn=<SumBackward0>)
```

```
[dydx] = torch.autograd.grad(s, [x])
```

Figure 2.3: Computational graph

The function `torch.autograd.grad(output_scalar, [list of input_tensors])` will return a list of the derivatives of the output scalar with respect to each of the input tensors. It computes `d(output_scalar)/d(input_tensor)` for each input tensor component in the list. For it to work, the input tensors and output must be part of the same `requires_grad = True` computation.

In the example below,, `x` is explicitly marked `requires_grad = True`, so `y.sum()`, which is derived from x, automatically comes along with the computation history, and can be differentiated.

Example: ***Autograd***

```
import torch
from matplotlib import pyplot as plt

x = torch.linspace(0, 5, 100,
            requires_grad=True)
y = (x**2).cos()
s = y.sum()
[dydx] = torch.autograd.grad(s, [x])

plt.plot(x.detach(), y.detach(), label='y')
plt.plot(x.detach(), dydx, label='dy/dx')
plt.legend()
plt.show()
```

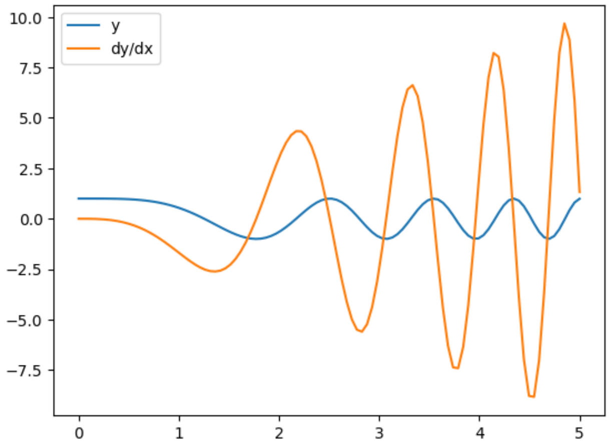


Figure 2.4: Derivative of $y = \cos(x^2)$

Observation: ***Example above***

Note that in the above example, because the components of the vector space are independent of each other, we happen to have `dy[j] / dx[i] == 0` when `j ≠ i`, so that `d(y.sum())/dx[i] = dy[i]/dx[i]`. That means computing a single gradient vector of the sum `s` is equivalent to computing elementwise derivatives `dy/dx`.

Every tensor that depends on `x` will be `requires_grad = True` and connected to the complete computation history. But if you were to convert a tensor to a regular python number, PyTorch would not be able to see the calculations and would not be able to compute gradients on it.

To avoid programming mistakes where some computation invisibly goes through a non-PyTorch number that cannot be tracked, PyTorch disables requires-grad tensors from being converted to untrackable numbers. You need to explicitly call `x.detach()` or `y.detach()` first, to explicitly say that you want an untracked reference, before plotting the data or using it as non-PyTorch numbers.

2.2.1 Backpropagation and In-Place gradients

In a typical neural network we will not just be getting gradients with regard to one input like `x` above, but with regard to a list of dozens or hundreds of tensor parameters that have all been marked with `requires_grad = True`. It can be inconvenient to keep track of which gradient outputs go with which original tensor input. But since the gradients have exactly the same shape as the inputs, it is natural to store computed gradients in-place on the tensors themselves.

To simplify this common operation, PyTorch provides the `.backward()` methods, which computes the gradients of y with respect to every tracked dependency, and stores the results in the field `x.grad` for every input vector `x` that was marked as `requires_grad = True`.

```
x = torch.linspace(0, 5, 100, requires_grad=True)
y = (x**2).cos()
y.sum().backward()   # populates the grad attribute below.
print(x.grad)

plt.plot(x.detach(), y.detach(), label='y')
plt.plot(x.detach(), x.grad, label='dy/dx')
plt.legend()
```

```python
plt.show()
```

2.2.2 Accumulating and zeroing gradients

If you find that your data batches are too large to get gradients of the whole thing, then it is usually possible to split the batches into smaller pieces and add the gradients. Because gradient accumulation is a common pattern, if you call `.backward()` when parameters `x.grad` already exists, it is not an error. The new gradient will be added to the old one.

That means that you need to set any previous value of `x.grad` to zero before running `backward()`, or else the new gradient will be added to the old one. Optimizers have a utility `optim.zero_grad()` to do this to all the optimized parameters at once.

2.2.3 Saving memory on inference

Normally, all the parameters of a neural network are set to `requires_grad = True` by default, so they are ready to be trained. But that means that whenever you run a network you will get output which is also requires-grad, and it will be attached to a long computation history that consumes a lot of precious GPU memory.

To avoid all this expense when you have no intention of training the network, you could go through all the network parameters to set `requires_grad = False`.

Another way to avoid the computation history is to enclose the entire computation within a `with torch.no_grad():` block. This will suppress all the autograd mechanics (which means `.backward()` will not work) and will save memory.

Note that this is different from the role of `net.eval()` which puts the network in inference mode computationally (batchnorm, dropout, and other operations behave differently in training and inference); `net.eval()` does not have any effect on `requires_grad`.

Moreover:

- Normally gradients with respect to intermediate values are not stored in `.grad`, but you can ask for intermediate gradients oìto be stored using `v.retain_grad()`.
- If you want higher-order derivatives, then you want PyTorch to build the computation graph when is computing the gradient itself, so this graph can be differentiated again. To do this, use the `create_graph = True` option on the `grad` or `backward` methods.
- Usually you only need to call `y.backward()` once per computation tree, and PyTorch will not let you call it again. To save memory, PyTorch will have deallocated the computation graph after you have computed a single gradient. But if you need more than one gradient, you can use `retain_graph = True`.

2.3 PyTorch Optimizers

Optimizers have a simple job: given gradients of an objective with respect to a set of input parameters, adjust the parameters reduce te objective. They o this by modifying each parameter by a small amount in the direction given by the gradient.

```python
#@title Run this cell to setup visualization...
# This cell defines plot_progress() which plots an optimization trace.

import matplotlib
```

```python
from matplotlib import pyplot as plt

def plot_progress(bowl, track, losses):
    # Draw the contours of the objective function, and x, and y
    fig, (ax1, ax2) = plt.subplots(1,2, figsize=(12, 5))
    for size in torch.linspace(0.1, 1.0, 10):
        angle = torch.linspace(0, 6.3, 100)
        circle = torch.stack([angle.sin(), angle.cos()])
        ellipse = torch.mm(torch.inverse(bowl), circle) * size
        ax1.plot(ellipse[0,:], ellipse[1,:], color='skyblue')
    track = torch.stack(track).t()
    ax1.set_title('progress of x')
    ax1.plot(track[0,:], track[1,:], marker='o')
    ax1.set_ylim(-1, 1)
    ax1.set_xlim(-1.6, 1.6)
    ax1.set_ylabel('x[1]')
    ax1.set_xlabel('x[0]')
    ax2.set_title('progress of y')
    ax2.xaxis.set_major_locator(matplotlib.ticker.MaxNLocator(integer=True))
    ax2.plot(range(len(losses)), losses, marker='o')
    ax2.set_ylabel('objective')
    ax2.set_xlabel('iteration')
    fig.show()

from IPython.display import HTML
HTML('''<script>function toggle_code(){$('.rendered.selected div.input').toggle().find('textarea').focus();} $(toggle_code())</script>
<a href="javascript:toggle_code()">Toggle</a> the code for plot_progress.''')
```

2.3.1 Gradient Descent just abstracts the gradient

You can apply gradient descent by hand easily by just using `loss.backward()` to compute the gradient of the loss with respect to every possible parameter `x`, and then apply `x -= learning_rate * x.grad` to nudge `x` in the gradient direction that makes the loss smaller. Here is an example of gradient descent on a simple bowl-shaped objective function.

Example: ***Gradient Descent***

```python
import torch

x_init = torch.randn(2)
x = x_init.clone()

bowl = torch.tensor([[ 0.4410, -1.0317], [-0.2844, -0.1035]])
track, losses = [], []

for iter in range(21):
    x.requires_grad = True
    loss = torch.mm(bowl, x[:,None]).norm()
    loss.backward()
    with torch.no_grad():
        x = x - 0.1 * x.grad
    track.append(x.detach().clone())
    losses.append(loss.detach())

plot_progress(bowl, track, losses)
```

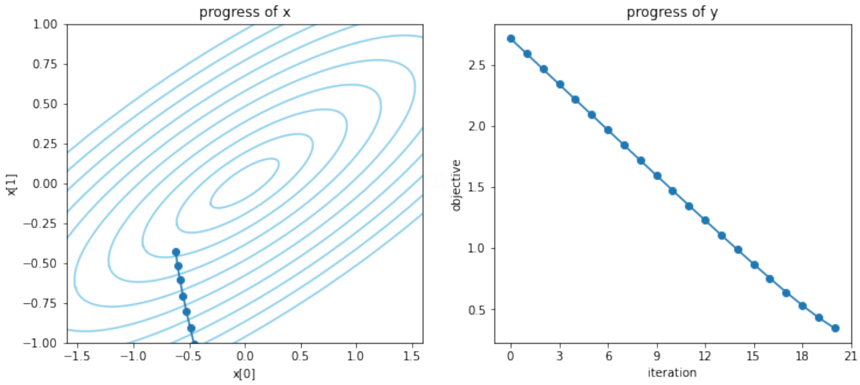


Figure 2.5: Gradient Descent

2.3.2 Built-in optimization algorithms

PyTorch includes several optimization algorithms.

The actual optimization algorithms employ a number of techniques to make the process faster and more robust as repeated steps are taken, by trying to adapt to the shape of the objective surface as it is explored. The simplest method is SGD-with-momentum, which is implemented in PyTorch as `pytorch.optim.SGD`.

To use SGD, you need to calculate your objective and fill in gradients on all the parameters before it can take a step.

1. Set your parameters (x in this case) to `x.requires_grad = True` so autograd tracks them
2. Create the optimizer and tell it about the parameters to adjust ([x] here)
3. In a loop, compute your objective, then call `loss.backward()` to fill in `x.grad` and then `optimizer.step()` to adjust x accordingly

Notice that we use `optimizer.zero_grad()` each time to set x.grad to zero before recomputing gradients; if we do not do this, then the new gradient will be added to the old one.

? Example: *SGD*

```python
import torch

x = x_init.clone()
x.requires_grad = True
optimizer = torch.optim.SGD([x], lr=0.1, momentum=0.5)

bowl = torch.tensor([[ 0.4410, -1.0317], [-0.2844, -0.1035]])
track, losses = [], []

for iter in range(21):
    loss = torch.mm(bowl, x[:,None]).norm()
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
    track.append(x.detach().clone())
    losses.append(loss.detach())

plot_progress(bowl, track, losses)
```

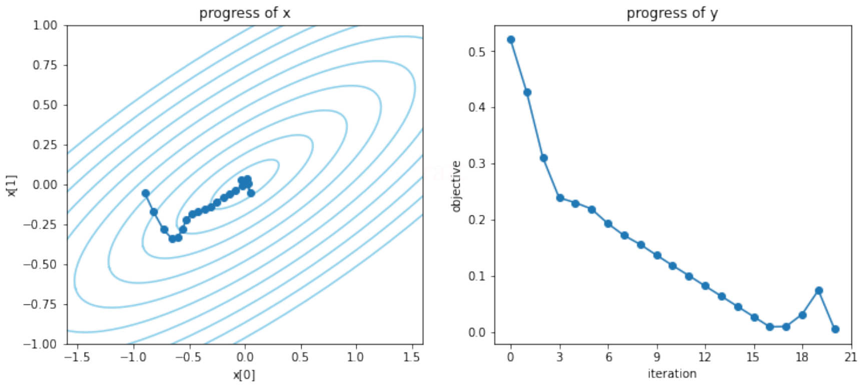


Figure 2.6: SGD

Other optimizers are similar. Adam is a popular adaptive method that does well without much tuning, and can be dropped in to replace plain SGD.

Some other fancy optimizers, such as LBFGS, need to be given an objective function that they can call repeatedly to probe gradients themselves.

Example: *Adam*

```python
# The code below uses Adam
x = x_init.clone()
x.requires_grad = True
optimizer = torch.optim.Adam([x], lr=0.1)

track, losses = [], []

for iter in range(21):
    loss = torch.mm(bowl, x[:,None]).norm()
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
    track.append(x.detach().clone())
    losses.append(loss.detach())

plot_progress(bowl, track, losses)
```

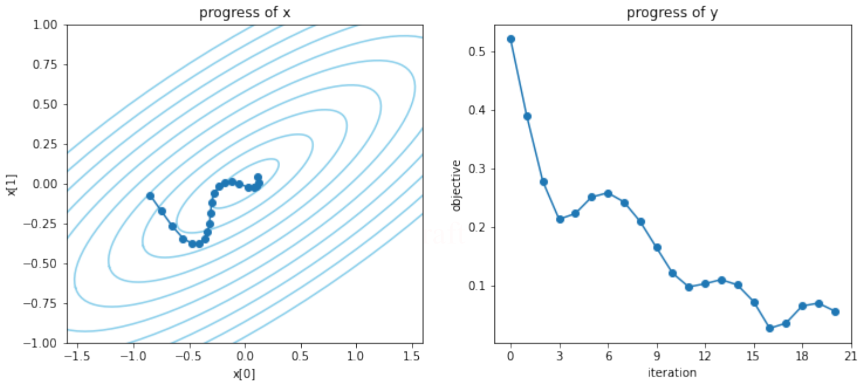


Figure 2.7: Adam

2.3.3 Other tricks

- **Learning rate schedules**: one of the simplest and most effective ways to improve training is to adjust the learning rate, decreasing it during training. There are many differerent strategies for scheduling learning rates, and pytorch comes with a set of `torch.optim.lr_scheduler` classes to make it easy to drop in a variety of methods.
- **Mutliple optimizers**: Sometimes you want to optimize more than one objective. The ordinary solution to this is to make an single overall objective as a (weighted) sum of all the objectives. However, sometimes you want to apply one objective to some parameters and a different objective to a different set of parameters. This occurs, for example in *adversarial training* such as in GANs, where two networks are learning to play against each other. In this case you can use multiple different optimizers, one for each opposing objective.

2.4 NN modules

PyTorch uses the `torch.nn.Module` class to represent a neural network. A `Module` is a callable function that can be:

- **Parametrized** by trainable `Parameter` tensors that the module can list out
- **Composed** out of children `Module`s that contribute parameters
- **Saved and Loaded** by listing named parameters and other attribute buffers

PyTorch comes with several built-in elementary network modules like a generic single-layer `Linear` network, or a generic `Sequential` composition of other networks, but of course you can write your own `Module` subclasses by just defining `Parameter` attributes and using them to implement a computation.

To see how every `Module` manages its own portion of responsibilities of all the network duties above, we first look at how to use the built-in `Linear` and `Sequential` modules.

Using `torch.nn.Linear as NN`

The linear layer is not just a good starting example: it is the fundamental workhorse of all neural networks, so as simple as it is, it is worth examining carefully.

`torch.nn.Linear` implements the function $y = Ax + b$, which takes m-dimensional input `x` and produces n-dimensional output `y`, by multiplying by the $n \times n$ matrix `A` (whose specific values are called the **weights**) and adding n-dimensional vector `b` (whose values are called the **bias**). We can make a Linear network with 3D input and 2D output just like the following:

```python
import torch
net = torch.nn.Linear(3, 2)
print(net)
```

Like any Module, our little network can be run as a function. As expected, when we give it 3D vectors as input, we get a 2D vector as output.

```python
    net(torch.tensor([[1.0, 0.0, 0.0]]))
    # tensor([[ 0.0000, -0.0734]], grad_fn=<AddmmBackward>)
```

Observation: ***Linear network***

Notice the double nesting in the vector data above. This is needed because out `Linear` network is slightly different from a plain matrix-vector multiplication. By convention, PyTorch Modules are set up to process data in batches, so to give it a single 3D vector, instead of passing just a vector, we have passed it a singleton batch containing one vector.

```
x_batch = torch.tensor([
    [1.0, 0.  , 0. ],
    [0.  , 1.0, 0. ],
    [0.  , 0.  , 1.0],
    [0.  , 0.  , 0. ],
])
#tensor([[-0.4530,  0.6047],
#        [-0.3436, -0.0084],
#        [-0.0957,  0.2989],
#        [-0.1118,  0.0888]], grad_fn=<AddmmBackward0>)
```

By default, the weights and biases of the `Linear` network are initialized randomly. You can see the values of the weights and biases by looking at the `.weight` and `.bias` attributes of the network.

```
print(net.weight)
print(net.bias)
```

Above you can see that bothe the weight and the bias are trainable parameters, because they both have the `Parameter` type. The tensors also both marked as `requires_grad = True`, which means that they are marked to participate in autograd and optimization for training.

These are the only two trainable parameters of the network. To check this, we can list all the parameters by name, with `net.named_parameters()`.

```
for name, param in net.named_parameters():
print(f'{name} = {param}\n')

#weight = Parameter containing:
#tensor([[-0.3413, -0.2319,  0.0160],
#        [ 0.5159, -0.0972,  0.2101]], requires_grad=True)

#bias = Parameter containing:
#tensor([-0.1118,  0.0888], requires_grad=True)
```

A module can also be saved by saving its state dictionary, which includes all the parameters and buffers. `net.state_dict()` is similar to `net.named_parameters()` but it returns a detached reference to the data (i.e., texttrequires_grad = False) so the data can be saved directly. Also, for more complicated modules, `state_dict()` may include other non-trainable attributes that are needed to save the network's state.

```
for k, v in net.state_dict().items():
    print(f'{k}: {v.type()}{tuple(v.shape)}')

import os
os.makedirs('checkpoints', exist_ok=True)
torch.save(net.state_dict(), 'checkpoints/linear.pth')

# weight: torch.FloatTensor(2, 3)
# bias: torch.FloatTensor(2,)
```

PyTorch also comes with convenient `torch.save` and `torch.load` functions for saving state dicts to files.

```python
net.load_state_dict(torch.load('checkpoints/linear.pth'))

# <All keys matched successfully>
```

2.4.1 Training Example: Optimizing a Linear Layer

To train a network we need to come up with a score for how close we are to the goal. This scalar number is called the **objective** or the **loss**.

As an example, suppose we would like this network to always output `[1, 1]` regardless of input. Then a reasonable loss would be the mean suqared distance to `[1, 1]`, computed like this:

```python
y_batch = net(x_batch)
loss = ((y_batch - torch.tensor([[1.0, 1.0]])) ** 2).sum(1).mean()
print(f'loss is {loss}')
```

We can use autograd get gradients to see how small changes in every parameter would impact the loss.

```python
loss.backward()
print(net.weight.grad)
print(net.bias.grad)
# tensor([[ 0.0000,  0.0000, -0.0000],
#         [ 0.0000,  0.0000, -0.0000]])
# tensor([0.0000, 0.0000])
# Note that the gradients are all zero, because the output is already at the target value.
# If we change the target to [0, 0], then the gradients are non-zero.
loss = ((y_batch - torch.tensor([[0.0, 0.0]])) ** 2).sum(1).mean()
loss.backward()
print(net.weight.grad)
# tensor([[ 0.0000,  0.0000, -0.0000],
#         [ 0.0000,  0.0000, -0.0000]])
print(net.bias.grad)
# tensor([-0.0000, -0.0000])
```

Simple gradient descent can be done directly. To improve our layer, we can use simple gradient descent with a learning rate of 0.01. That is, we can adjust each parameter by subtracting 0.01 times the gradient. If we do this repeatedly, we should get closer to our objective.

Any time we directly update the network parameters, we need to temporarily disable the autograd machinery using `with torch.no_grad()`.

```python
net = torch.nn.Linear(3, 2)
log = []
for _ in range(10000):
    y_batch = net(x_batch)
    loss = ((y_batch - torch.tensor([[1.0, 1.0]])) ** 2).sum(1).mean()
    log.append(loss.item())
    net.zero_grad()
    loss.backward()
    with torch.no_grad():
        for p in net.parameters():
            p[...] -= 0.01 * p.grad
```

```
print(f'weight is {net.weight}\n')
print(f'bias is {net.bias}\n')

%matplotlib inline
import matplotlib.pyplot as plt
plt.ylabel('loss')
plt.xlabel('iteration')
plt.plot(log)
plt.show()

#weight is Parameter containing:
#tensor([[7.5908e-06, 7.5909e-06, 7.5911e-06],
#        [1.4001e-05, 1.4001e-05, 1.4001e-05]], requires_grad=True)
#
#bias is Parameter containing:
#tensor([1.0000, 1.0000], requires_grad=True)
```

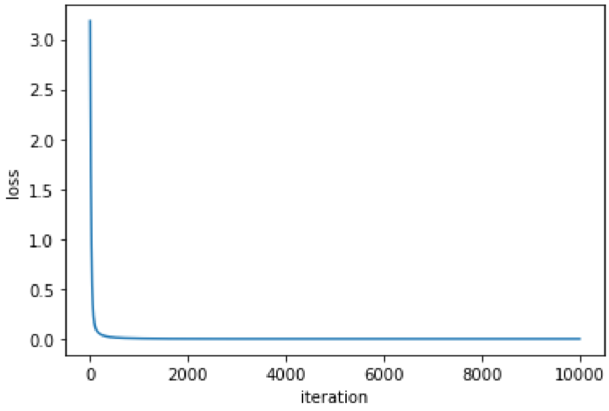


Figure 2.8: Loss over iterations

2.4.2 Using `torch.nn.Sequential` to compose networks

Unlike `Linear`, most networks are made by composing many smaller networks. The simplest way to do this is chain networks together end-to-end, connecting each outpu to the next input. For example, we can simply compose `Linear` layers.

Defining Multilayer Perceptron. Of course, to get something more interesting than another linear function, we need to do something nonlinear between the linear steps. If we add a nonlinearity between each step (for example, if we clamp negative numbers to zero - an operation called **ReLU**), then we can get a **Multilayer Perceptron**, which is known to be a universal function approximator, i.e., a family of piecewise-linear functions that can approximate any function.

Here is how we can express the network as a nested set of Sequentials:

```
from collections import OrderedDict
from torch.nn import Linear, ReLU, Sequential

mlp = torch.nn.Sequential(OrderedDict([
('layer1', Sequential(Linear(2, 20), ReLU())),
('layer2', Sequential(Linear(20, 20), ReLU())),
('layer3', Sequential(Linear(20, 2)))
]))
```

```

print(mlp)

#Sequential(
#  (layer1): Sequential(
#    (0): Linear(in_features=2, out_features=20, bias=True)
#    (1): ReLU()
#  )
#  (layer2): Sequential(
#    (0): Linear(in_features=20, out_features=20, bias=True)
#    (1): ReLU()
#  )
#  (layer3): Sequential(
#    (0): Linear(in_features=20, out_features=2, bias=True)
#  )
#)
```

In the above, we have nested two levels of Sequentials. In the outermost level, we have defined and named three layers.

Then each layer is itself a Sequential that executes a parametrized `Linear` operation followed by a `ReLU` nonlinear clamping operation. We have not bothered to name each of the innermost steps, so the Sequential just automatically numbers them.

Every submodule has a fully qualified name. We can het a full recursive list of submodules by listing `net.named_modules()`.

```
for n, c in mlp.named_modules():
print(f'{n or "The whole network"} is a {type(c).__name__}')

#The whole network is a Sequential
#layer1 is a Sequential
#layer1.0 is a Linear
#layer1.1 is a ReLU
#layer2 is a Sequential
#layer2.0 is a Linear
#layer2.1 is a ReLU
#layer3 is a Sequential
#layer3.0 is a Linear
```

A module's parameters include all its child module parameters. We can see this by listing all the parameters by name.

```
for name, param in mlp.named_parameters():
print(f'{name} has shape {tuple(param.shape)}')

#layer1.0.weight has shape (20, 2)
#layer1.0.bias has shape (20,)
#layer2.0.weight has shape (20, 20)
#layer2.0.bias has shape (20,)
#layer3.0.weight has shape (2, 20)
#layer3.0.bias has shape (2,)
```

There are now six parameters: a weight and a bias for each of the three `Linear` layers.

Training a Classifier. This slightly more complicated network can now represent a more general class of functions. As an example, we can use this architecture to learn to compute a classifier function.

Suppose we want to classify points on a plane as either above a sine-wave (class 1) or below a sine-wave (class 0). Here is the ordinary training loop to train our MLP to do it, using the Adam optimizer:

```
from torch.nn.functional import cross_entropy

def classify_target(x, y):
    return (y > (x * 3).sin()).long()

mlp.cuda()
optimizer = Adam(mlp.parameters(), lr=0.01)
for iteration in range(1024):
    in_batch = torch.randn(10000, 2, device='cuda')
    target_batch = classify_target(in_batch[:,0], in_batch[:,1])
    out_batch = mlp(in_batch)
    loss = cross_entropy(out_batch, target_batch)
    if iteration > 0:
        mlp.zero_grad()
        loss.backward()
        optimizer.step()
    if iteration == 2 ** iteration.bit_length() - 1:
        pred_batch = out_batch.max(1)[1]
        accuracy = (pred_batch == target_batch).float().sum() / len(
            in_batch)
        print(f'Iteration {iteration} accuracy: {accuracy}')
        visualize_net(mlp, classify_target)
```

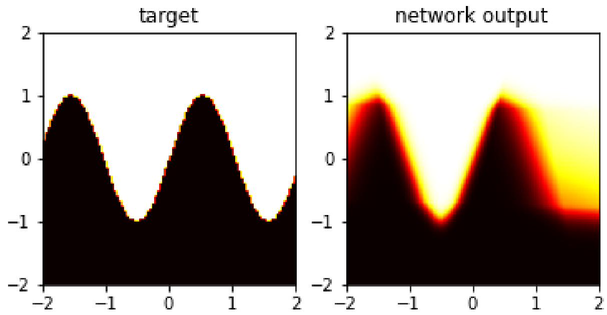


Figure 2.9: Classifier accuracy

2.4.3 Defining `forward` to create custom networks

Sometimes you will want to hook up network components in a more complicated way than just a sequential operation of predefined components.

For example, ResNet is built on the observation that learning can work much better if, instead of learning an arbitrary linear operation, we learn perturbations of the identity. I.e., have a layer learn to compute a small residual instead of the whole total answer.

To apply the residual trick in our little three-layer network, we cannot just use an overall `Sequential`: instead we will define the operation by writing our own `forward` function. It looks like this:

```python
class MyNetwork(torch.nn.Module):
def __init__(self):
    super().__init__()
    self.layer1 = Sequential(Linear(2, 20), ReLU())
    self.residual_layer2 = Sequential(Linear(20, 20), ReLU())
    self.layer3 = Linear(20, 2)

def forward(self, x):
    x = self.layer1(x)
    x = x + self.residual_layer2(x)
    x = self.layer3(x)
    return x

res_mlp = MyNetwork()
print(res_mlp)

# Exercise left to you: try training res_mlp just like we trained mlp above.


#MyNetwork(
#  (layer1): Sequential(
#    (0): Linear(in_features=2, out_features=20, bias=True)
#    (1): ReLU()
#  )
#  (residual_layer2): Sequential(
#    (0): Linear(in_features=20, out_features=20, bias=True)
#    (1): ReLU()
#  )
#  (layer3): Linear(in_features=20, out_features=2, bias=True)
#)
```

Training the network above can be done exactly the same way as training our previous `Sequential` mlp. Try copying+pasting the training code and adapting it to this new network.

2.4.4 Other `Module` tricks

- `torch.nn.Parameter` wraps trainable parameters. In the `__init__()` method, you can define more tensors as parameters to be optimized, by wrapping parameter tensors with a `torch.nn.Parameter` before setting the attribute. See the pytorch Linear source code to see an example.
- `module.training` allows special behavior during training. Some modules behave differently at training time than at inference time. For example a Dropout layer will drop half the channels and amplify the other half randomly during training, but at inference, for better performance, it will include all the channels. To support this sort of trick, there is a `module.train()` method to put a module (recursively) into training mode and a `module.eval()` to put it into inference mode. The `module.training` boolean tells which mode is current.
- **buffers** can be learned without the optimizer. Not every attribute of a module needs to be trainable by the optimizer. Some attributes can be learned in a different way, for example by observing and averaging statistics observed during training time. The most famous example of this is the Batchnorm layer, which observes mean and variance during training, and accumulates statistics to enforces zero mean and unit variance.
- **Predefined model architecture** are available; for example

`torchvision.models.resnet18(num_classes=100)` will create a ResNet-18 classifier model, configured to do a 100-way classification of images.

2.5 Datasets and Dataloaders in PyTorch

```
# To save time, start this download first, before reading through the examples.
import torch, torchvision, os
if not os.path.isfile('datasets/miniplaces/train/yard/00001000.jpg'):
    torchvision.datasets.utils.download_and_extract_archive(
        'http://dissect.csail.mit.edu/datasets/miniplaces.zip',
        'datasets', md5='bfabeb497c7eca01c74cd8441a9ac108')
```

Data sets can be thought of as big arrays of data. If the data set is small enough (e.g. MNIST, which has 60,000 28x28 grayscale images), a dataset can be literally represented as an array - or more precisely, as a single PyTorch tensor. With one number per pixel, MNIST takes about 200MB of RAM, which fits comfortably into a modern computer.

But larger-scale datasets like ImageNet of Places365 have more than a million higher-resolution full-color images. In these cases, an ordinary python array or PyTorch tensor would require more than a TB of RAM, which is impractical on most computers.

Instead, we need to load the data from dist (or SSD). Unfortunately, the latency of loading from disk is very slow compared to RAM, so we need to do the loading cleverly if we want to load the data quickly.

To solve the problem, PyTorch provides two classes:

- `torch.utils.data.Dataset` - This very simple base class represents an array where the actual data may be slow to fetch, typically because the data is in disk files that require some loading, decoding, or other preprocessing. PyTorch provides a variety of different `Dataset` subclasses. As an example, there is a handy one called `ImageFolder` that treats a directory tree of image files as an array of classified images.
- `torch.utils.data.DataLoader` - This fancy class wraps a `Dataset` as a stream of data batches. Behind the scenes it uses a few techniques to feed the data faster. You do not need to subclass `DataLoader` - its purpose is to make a `Dataset` speedy.

2.5.1 Looking at an image dataset using ImageFolder

The most common `Dataset` used in computer vision is `ImageFolder`, which loads a set of image files from a directory tree. It treats every subdirectory of images as a classification category. To demonstrate it, we will use it to load images from the miniplaces dataset loaded above.

Directory layout. Notice that `datasets/miniplaces/val` contains a set of 100 directories with names like `golf_course`. Each of these directories contain 100 images, each stored as a jpeg file: 10000 images in total.

```
ls datasets/miniplaces/val/golf_course
```

Constructing an ImageFolder. Making an ImageFolder at the root directory of the dataset creates an object that behaves like an array: it has a length and each entry contains a tuple with an image and a number. The image is stored as a `PIL` object, which is a standard in python object for images and the number denotes the classification class - with one class for each folder, numbered in alphabetical order.

```python
val_set = torchvision.datasets.ImageFolder('datasets/miniplaces/val')
print('Length is', len(val_set))
item = val_set[5100]
print('5100th item is a pair', item)

# Display the PIL image and the class name directly.
display(item[0])
print('Class name is', val_set.classes[item[1]])
```

Transforming the PIL image into a PyTorch tensor. A PIL image is not convenient for training: we would prefer our dataset to return PyTorch tensors. So we can tell `ImageFolder` to do this by specifying the `transform` function on construction. PyTorch comes with a standard transform function `torchvision.transforms.ToTensor()` which converts an image to a PyTorch tensor. Now when indexing into the dataset, we will get a PyTorch tensor instead of a PIL image.

```python
val_set =  torchvision.datasets.ImageFolder(
    'datasets/miniplaces/val',
    transform=torchvision.transforms.ToTensor())
print(val_set[1000])

# There is an inverse transform that can be used to convert it back to a PIL image,
# handy if we want to see it.
as_image = torchvision.transforms.ToPILImage()
display(as_image(val_set[1000][0]))
```

2.5.2 Fast Dataset Access using DataLoader

When we use a dataset, we will usually run through the whole dataset in batches. We could do this ourselves, as in line 6-8 below, by just fetching the images one at a time and grouping them.

But a faster way to iterate the dataset is to wrap our `val_set` object in a `torch.utils.data.DataLoader` object, as shown on line 14-18 below. The `val_loader` we get can magically pull data out of the Dataset much faster than doing it in the simple way; the `DataLoader` class does this by using several threads to load and prefetch the data.

The speedup will depend on the system and the number of threads you use (the number of threads to use is specified using `num_workers`). In practice using `DataLoader` will typically be 5-20 times faster than direct `Dataset` access.

```python
1  import time
2  
3  print('Going over the data set as an array.')
4  start = time.time()
5  summed_image_dataset = 0
6  batch_size = 100
7  for i in range(0, len(val_set), batch_size):
8      image_batch = torch.stack([val_set[i+j][0] for j in range(batch_size)])
9      summed_image_dataset += image_batch.sum(0)
10 end = time.time()
11 print(f'Took {end - start} seconds')
12 
13 print('Going over the same dataset using a dataloader.')
```

```python
start = time.time()
val_loader = torch.utils.data.DataLoader(
    val_set, batch_size=batch_size, num_workers=10)
summed_image_loader = 0
for image_batch, label_batch in val_loader:
    summed_image_loader += image_batch.sum(0)
end = time.time()
print(f'Took {end - start} seconds')

print('Numerical difference is exactly', (summed_image_loader -
    summed_image_dataset).norm().item())
```

2.5.3 Using a DataLoader for Training

We can put everything together by using the data from a data loader to train a classifier.

The following is a simplistic example of training an image classifier. It uses the Adam optimizer and the ResNet-18 neural network architecture, and trains for a couple minutes, just passing once over the training set.

```python
from tqdm import tqdm

# Create a Dataset of miniplaces training images.
train_set = torchvision.datasets.ImageFolder(
    'datasets/miniplaces/train',
    torchvision.transforms.ToTensor())

# Wrap the Dataset in a high-speed DataLoader with batch_size 100.
train_loader = torch.utils.data.DataLoader(
    train_set, batch_size=100, num_workers=10,
    shuffle=True,
    pin_memory=True)

# Create an untrained neural network using the ResNet 18 architecture.
model = torchvision.models.resnet18(num_classes=100).cuda()

# Set up the model for training using the Adam optimizer.
model.train()
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)

# To train, optimize an objective on batches of training data.
# Here we look at every training image once.
for batch in tqdm(train_loader):
    images, labels = [d.cuda() for d in batch]
    optimizer.zero_grad()
    scores = model(images.cuda())
    loss = torch.nn.functional.cross_entropy(scores, labels)
    loss.backward()
    optimizer.step()
```

2.5.4 Checking Accuracy with a Held-Out Dataset

To check if network has learned anything useful, we can check whether the model can make good predictions on unseen images. The easy way to do this is to create a second `ImageFolder` dataset (and `DataLoader`) with a second set of images that was **not** used for training.

While the achieved accuracy after a couple minutes of training is not perfect, it is already much better than random.

```python
# Create a validation dataset and data loader.
val_set = torchvision.datasets.ImageFolder(
    'datasets/miniplaces/val',
    torchvision.transforms.ToTensor())
val_loader = torch.utils.data.DataLoader(
    val_set, batch_size=100, num_workers=10,
    pin_memory=True)

# This function runs over the validation images and counts accurate predictions.
def accuracy():
    model.eval()
    correct = 0
    for iter, batch in enumerate(val_loader):
        images, labels = [d.cuda() for d in batch]
        with torch.no_grad():
            scores = model(images.cuda())
        correct += (scores.max(1)[1] == labels).float().sum()
    return correct.item() / len(val_set)

print(f'Accuracy on unseen images {accuracy() * 100}% (random guesses would be 1%)')
```

2.5.5 Improving Training using Data Augmentation

One of the main ways to stretch a dataset to make it more effective for training is to randomly adjust the images. For example if we randomly adjust the crop, color, or orientation of the image while loading, using the same image file multiple times will produce different training examples for the network. This is an easy way to increase the amount of training diversity in the dataset without requiring more actual images.

To do data augmentation in a PyTorch `Dataset`, you can specify more operations on `transform=` besides `ToTensor()`.

In particular, there is a `Compose` transform that makes it easy to chain a series of data transformations; and `torchvision.transforms` includes a number of useful image transforms such as random resized crops and image flips.

Here is an example:

```python
# Create a Dataset of miniplaces training images.
train_set = torchvision.datasets.ImageFolder(
    'datasets/miniplaces/train',
    torchvision.transforms.Compose([
        torchvision.transforms.RandomCrop(112),
        torchvision.transforms.RandomHorizontalFlip(),
        torchvision.transforms.ToTensor(),
    ]))
train_loader = torch.utils.data.DataLoader(
    train_set, batch_size=100, num_workers=10,
    shuffle=True,
    pin_memory=True)

```

```python
# Now let's train for one more epoch, and test the accuracy
model.train()
for batch in tqdm(train_loader):
    images, labels = [d.cuda() for d in batch]
    optimizer.zero_grad()
    scores = model(images.cuda())
    loss = torch.nn.functional.cross_entropy(scores, labels)
    loss.backward()
    optimizer.step()
print(f'Accuracy on unseen images {accuracy() * 100}% (random guesses
    would be 1%)')
```