Intro to deep learning

Dr. Janoś Gabler, University of Bonn

Lecture 9: Training neural networks in Pytorch

Topics

- Git and GitHub
- Recap: Feed-forward neural networks from scratch
- Defining Models in Pytorch
- Writing Training Loops in Pytorch
- Optimization algorithms

Git and GitHub

What is Git

- Distributed version control system
  - Go back in time
  - Have multiple versions of code
- Used by almost every programmer in the world
- Code is in a git-repository (the parent folder of your project)
- Used from the terminal
- Takes a while to learn, but we only need the absolute basics

What is GitHub

- Webpage where you can upload git repositories
- Collaboration tools
  - Pull requests
  - Review features
  - Automated tests
- You need to sign up for an account

What is GitHub Classroom

- Helps me to collect your assignments
- Helps you by creating a repository for you

Create a GitHub Account and accept invitation

- Go to https://github.com/
- Create an account if you don't have one
- Choose a name that is easy to type, memorize and pronounce
- Accept the invitation

GitHub Classroom

GitHub Education

dl-intro

Accept the assignment —

final-project

Once you accept this assignment, you will be granted access to the `final-project-janosg` repository in the iame-uni-bonn organization on GitHub.

Accept this assignment

GitHub Classroom

GitHub Education

You accepted the assignment, **final-project** . We're configuring your repository now. This may take a few minutes to complete. Refresh this page to see updates.

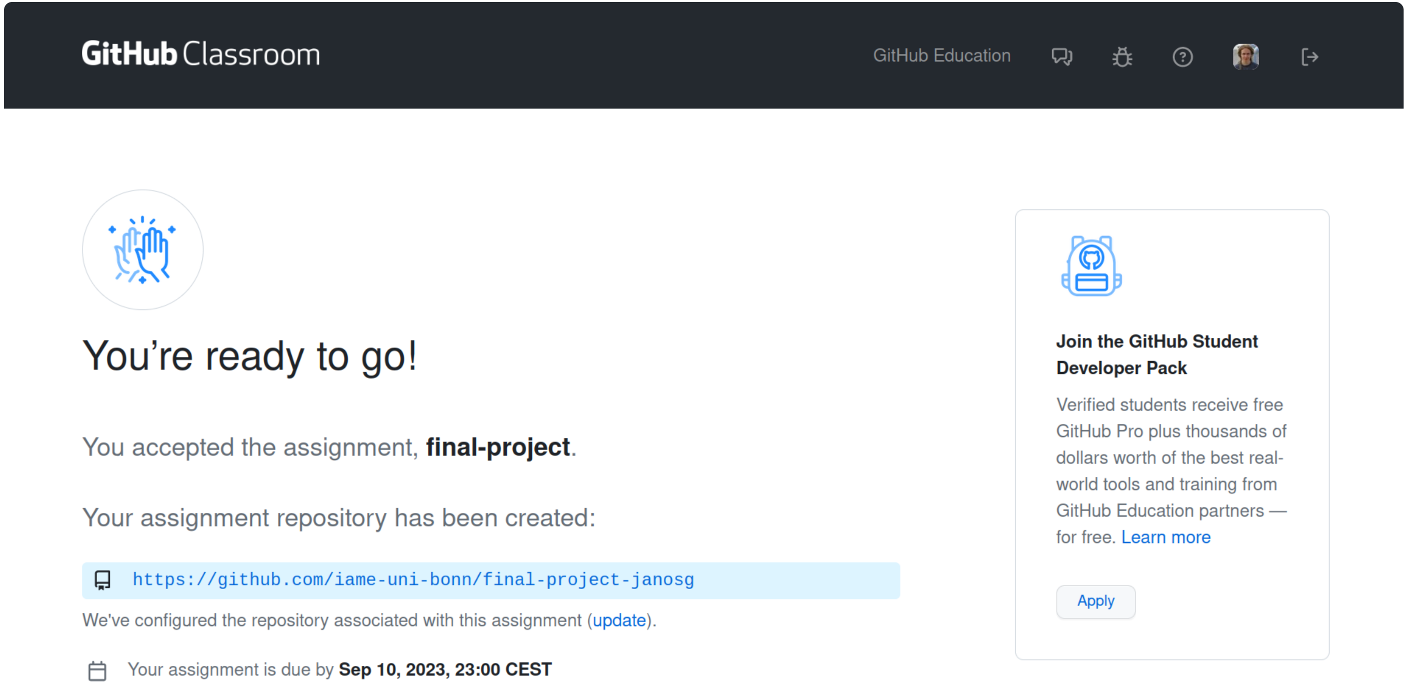
Your assignment is due by **Sep 10, 2023, 23:00 CEST**

Note: You may receive an email invitation to join iame-uni-bonn on your behalf. No further action is necessary.

Join the GitHub Student Developer Pack

Verified students receive free GitHub Pro plus thousands of dollars worth of the best real-world tools and training from GitHub Education partners — for free. Learn more

Apply


GitHub Classroom
GitHub Education
You're ready to go!
You accepted the assignment, final-project.
Your assignment repository has been created:
https://github.com/iame-uni-bonn/final-project-janosg
We've configured the repository associated with this assignment (update).
Your assignment is due by Sep 10, 2023, 23:00 CEST
Join the GitHub Student Developer Pack
Verified students receive free GitHub Pro plus thousands of dollars worth of the best real-world tools and training from GitHub Education partners — for free. Learn more
Apply

Search or jump to...

Pull requests Issues Codespaces Marketplace Explore

iame-uni-bonn / **final-project-janosg** Private

Edit Pins · Unwatch 1 · Fork 0 · Star 0

Code · Issues · Pull requests · Actions · Projects · Wiki · Security · Insights · Settings

main · **1** branch · **0** tags

Go to file · Add file · Code

github-classroom[bot] Add assignment deadline url · 7b29418 1 minute ago · **1** commit

README.md	Add assignment deadline url	1 minute ago

README.md

Review the assignment due date

About

final-project-janosg created by GitHub Classroom

- Readme
- Activity
- **0** stars
- **1** watching
- **0** forks

Releases

No releases published

Create a new release

Packages

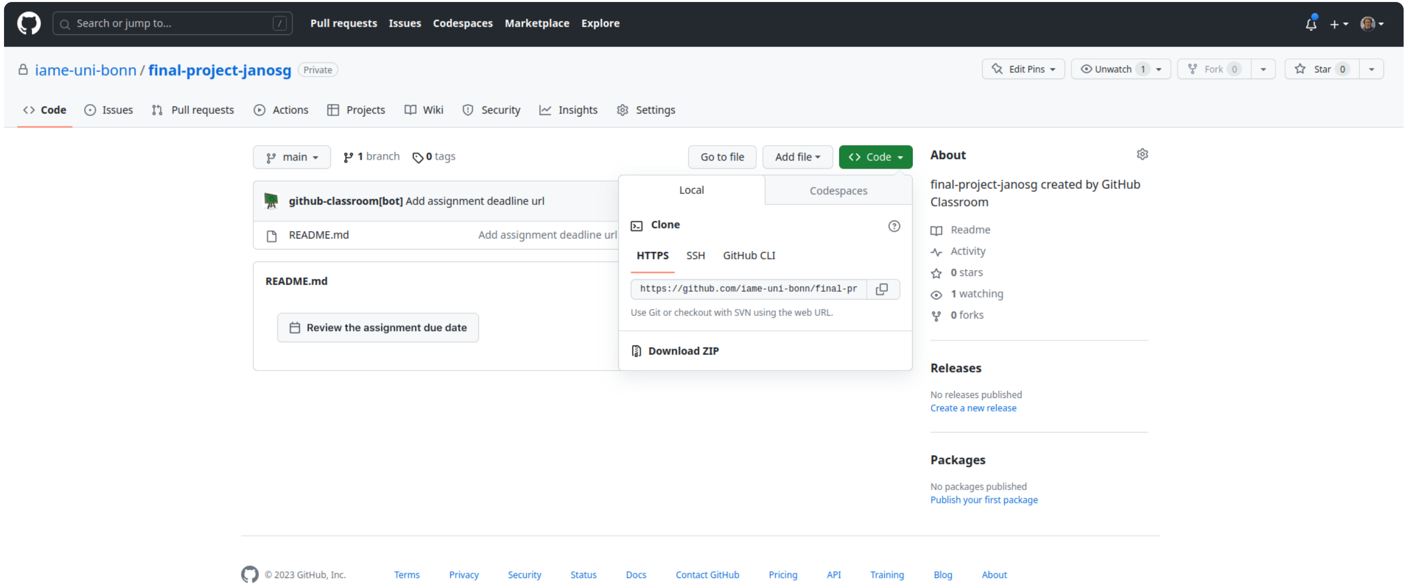
No packages published

Publish your first package

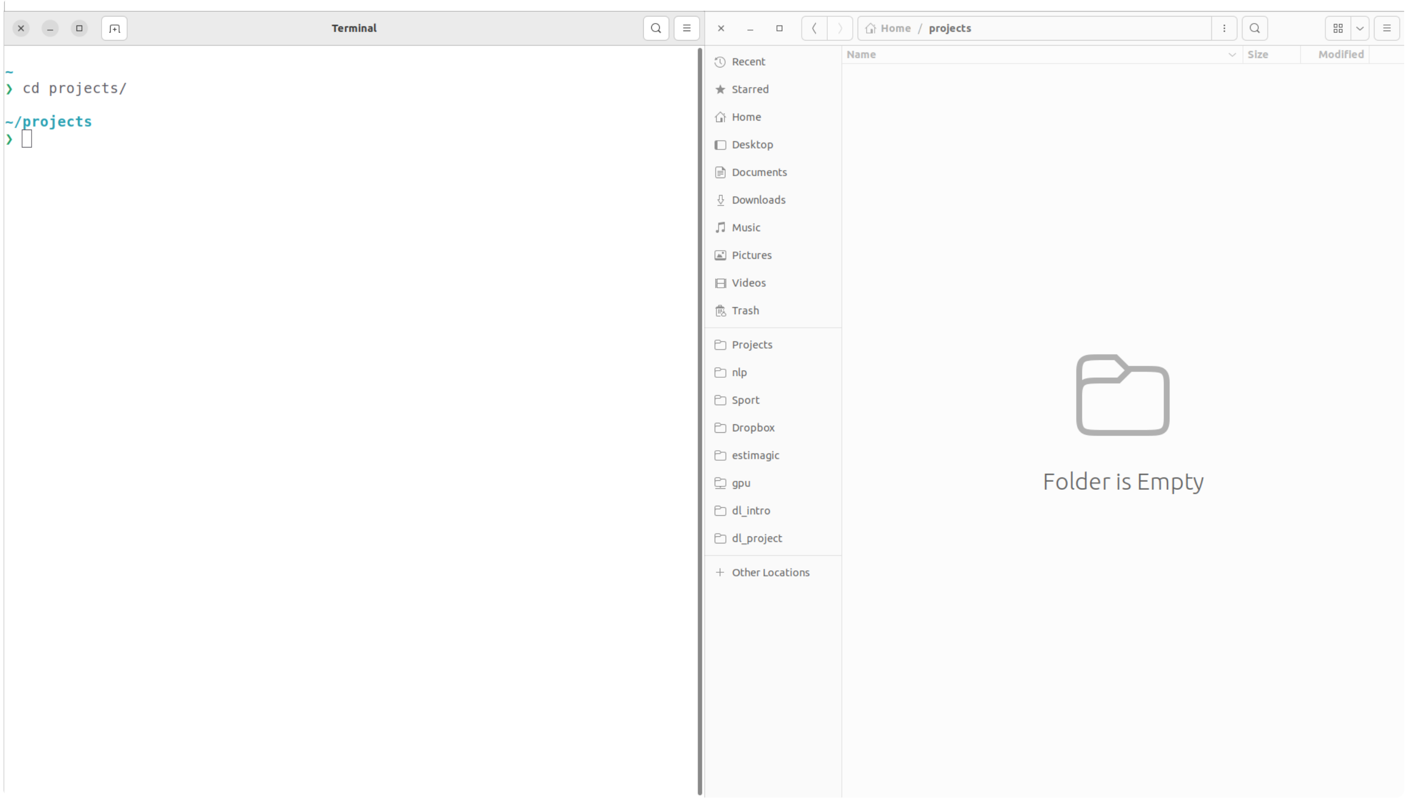
Clone the repository to your computer

- Cloning means downloading the repository to your computer
- You do it from a terminal
  - Open the terminal
  - Navigate to a folder to which you want to download the repo
  - Use git-clone

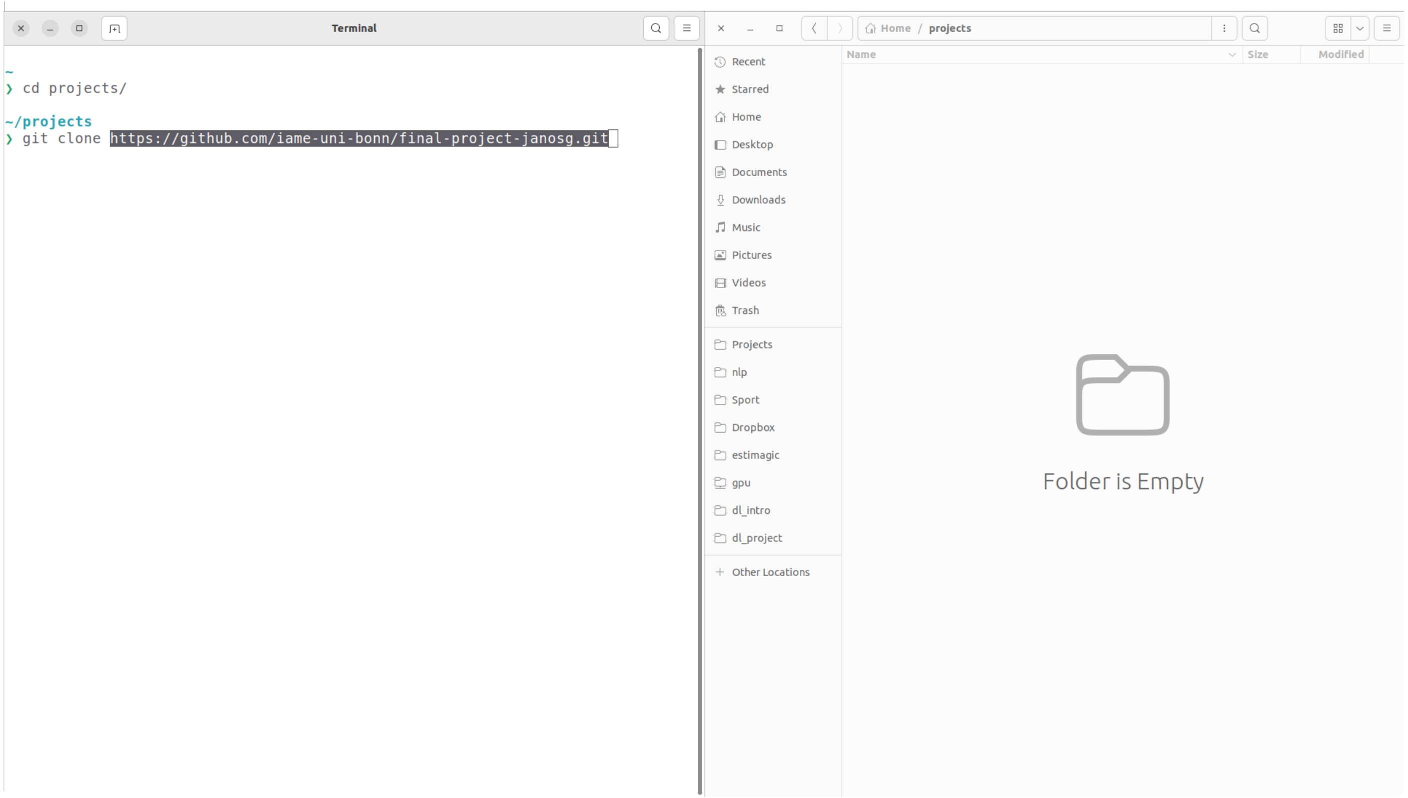
Get the clone link online



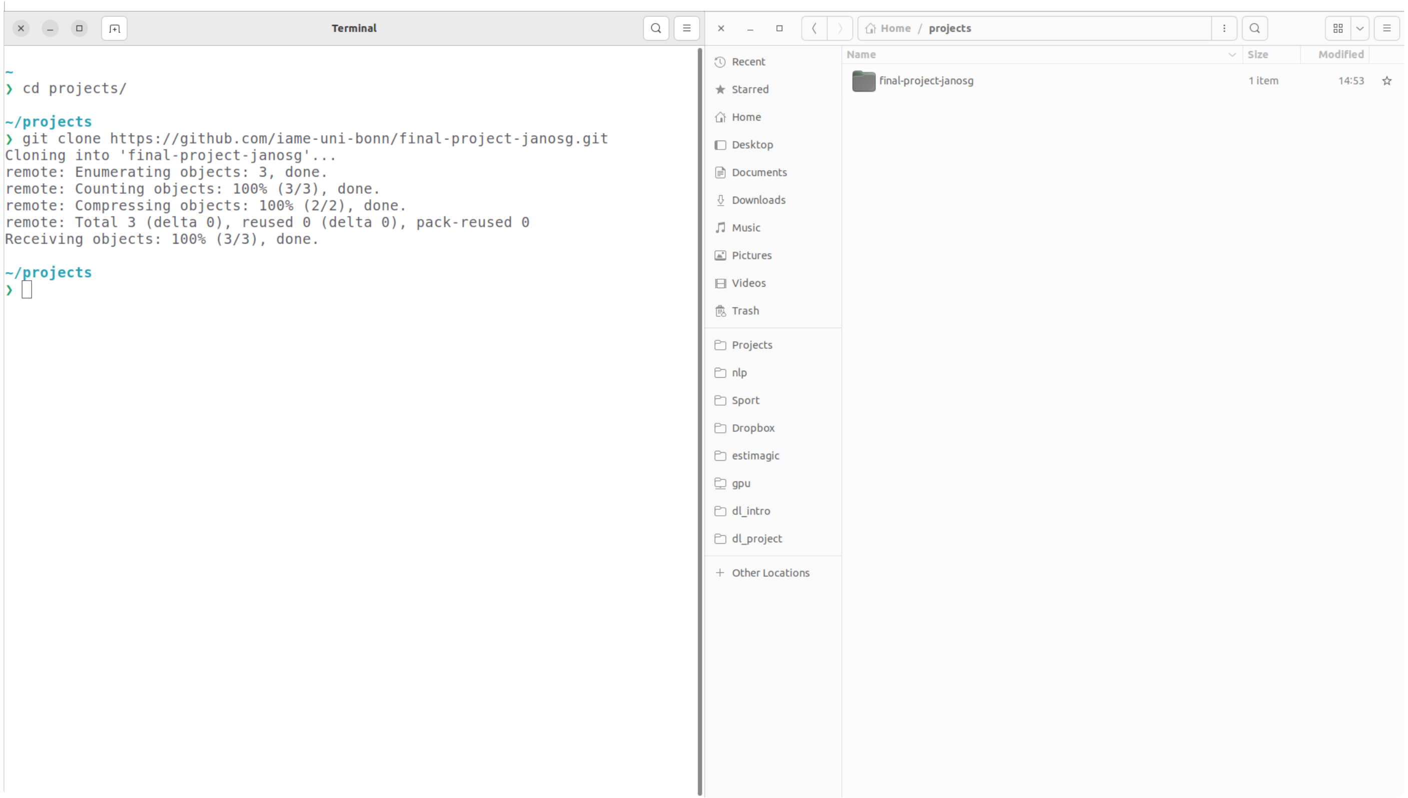
Open a terminal and navigate to folder



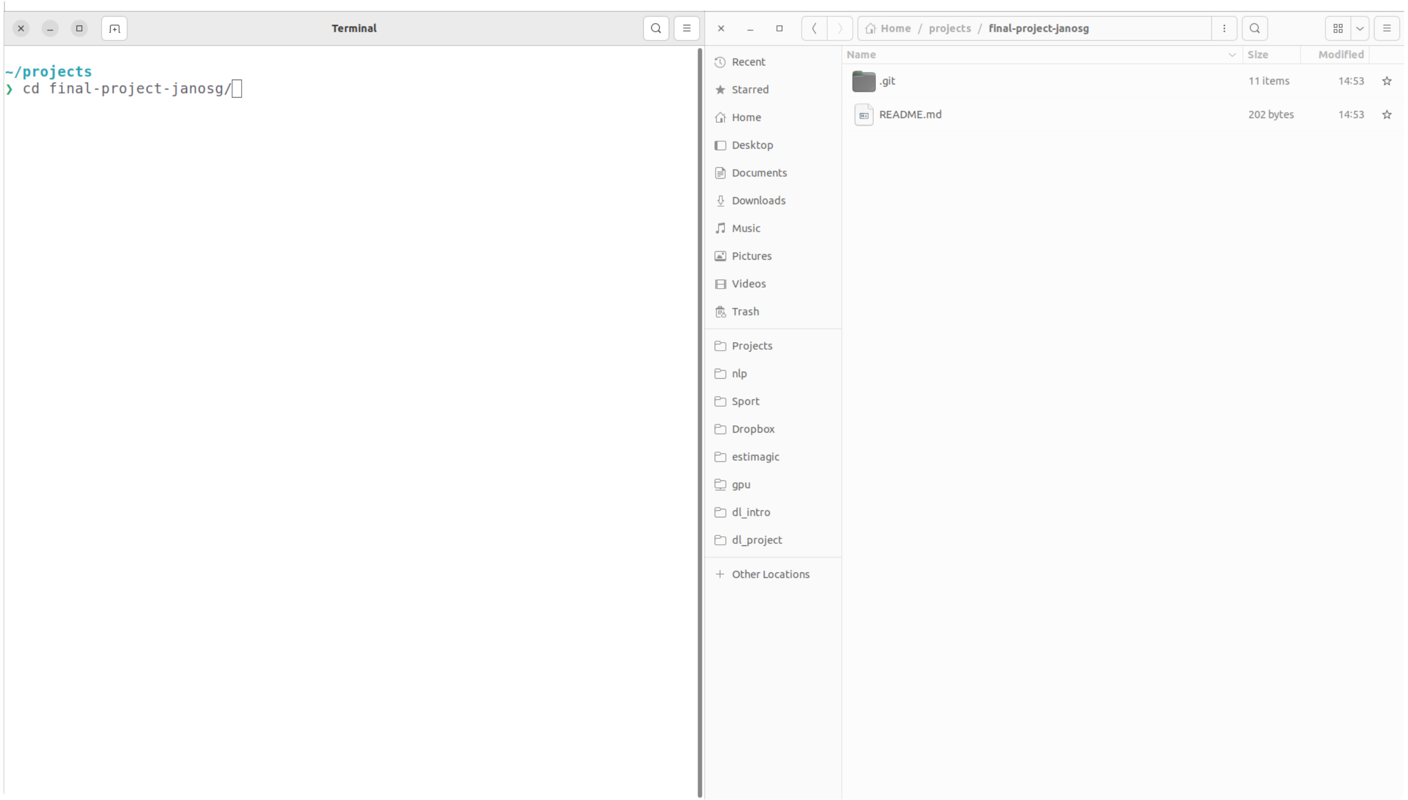
Type clone command



Hit Enter to download repo



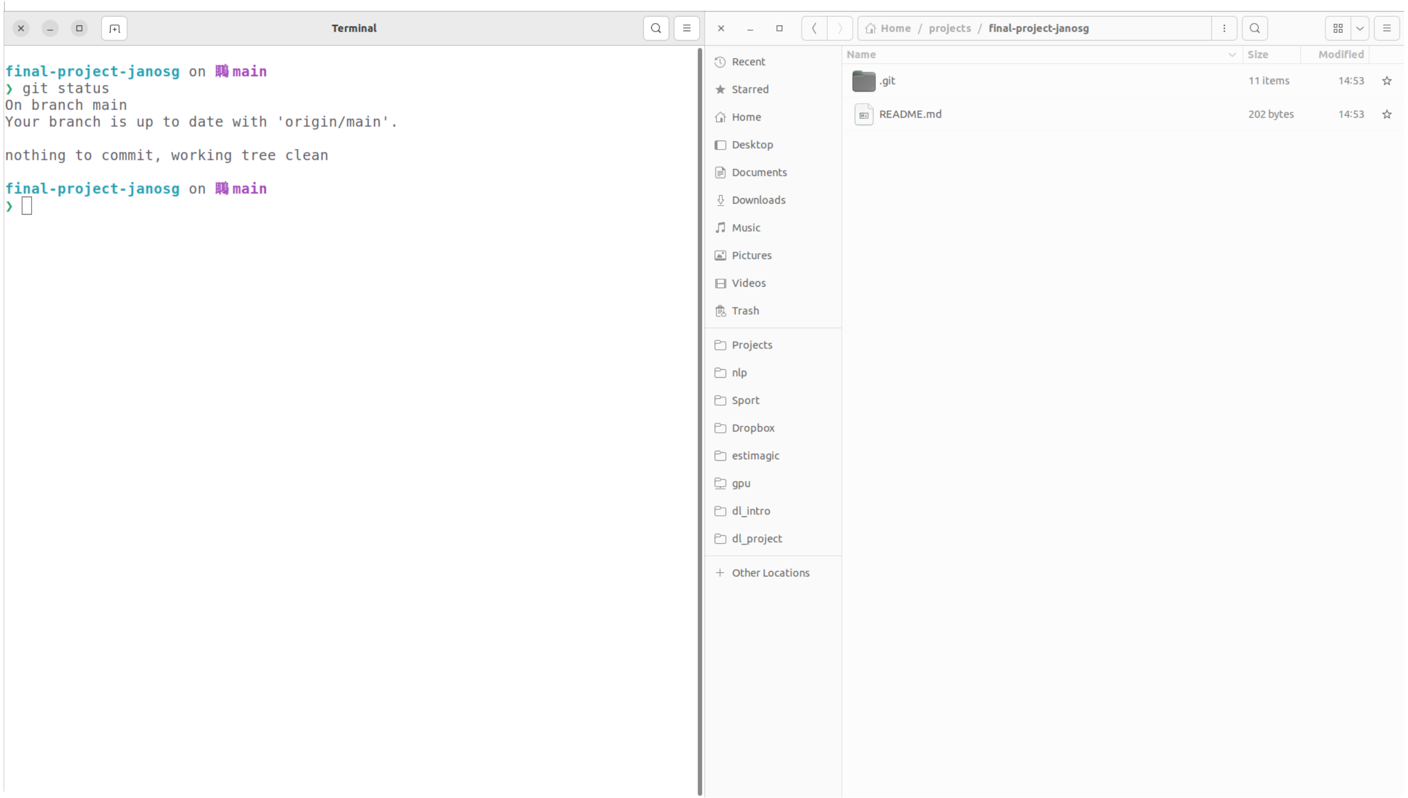
`cd` into the repo



`git status`

- Executing `git status` inside a git repository gives you information
  - Which files were added?
  - Which files were modified
  - Are there un-pushed changes?
- It should be every other git command you type!

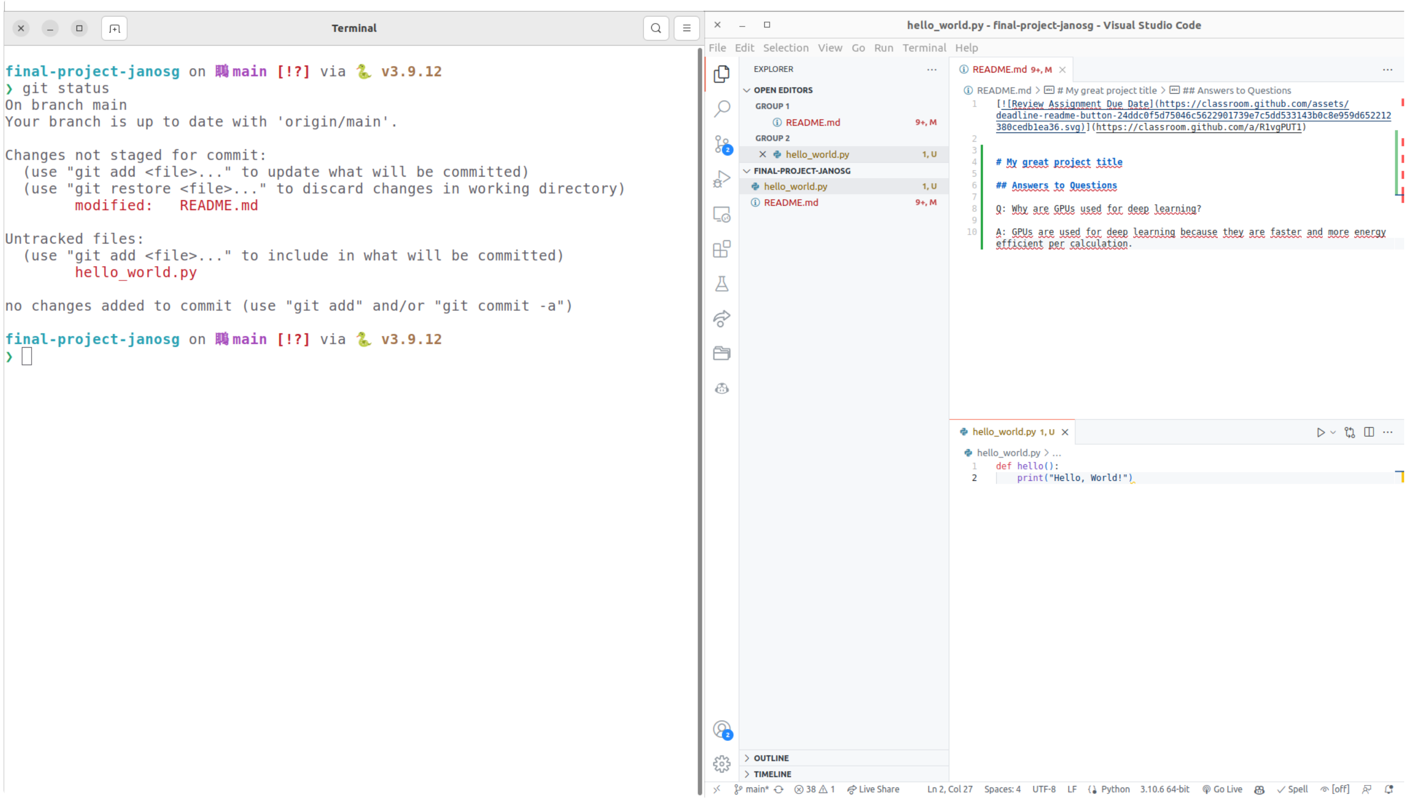
Status before changes



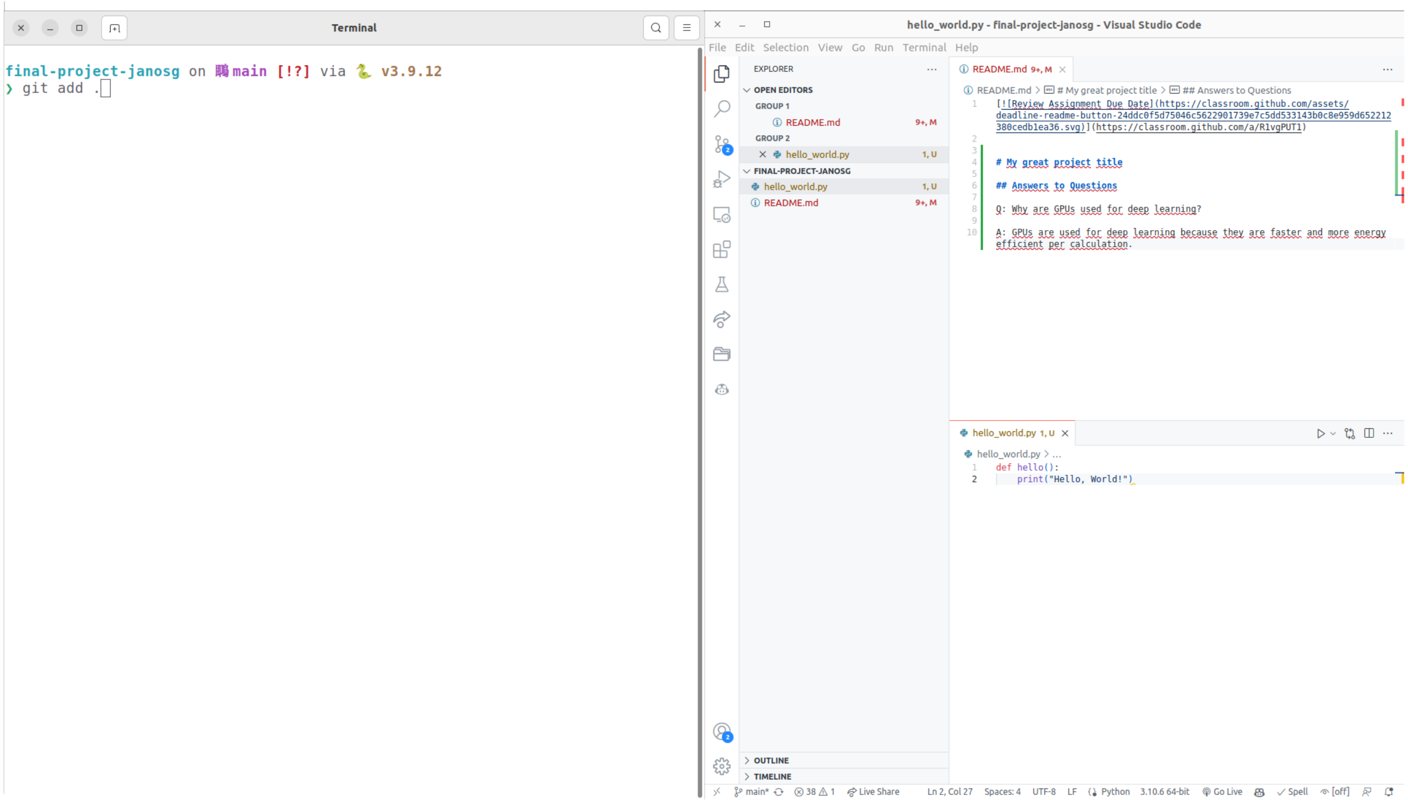
Make some changes

- You can now make your changes
  - Add files
  - Modify existing files
- The changes will not be synchronized to github automatically
- To share them, you need to commit and push

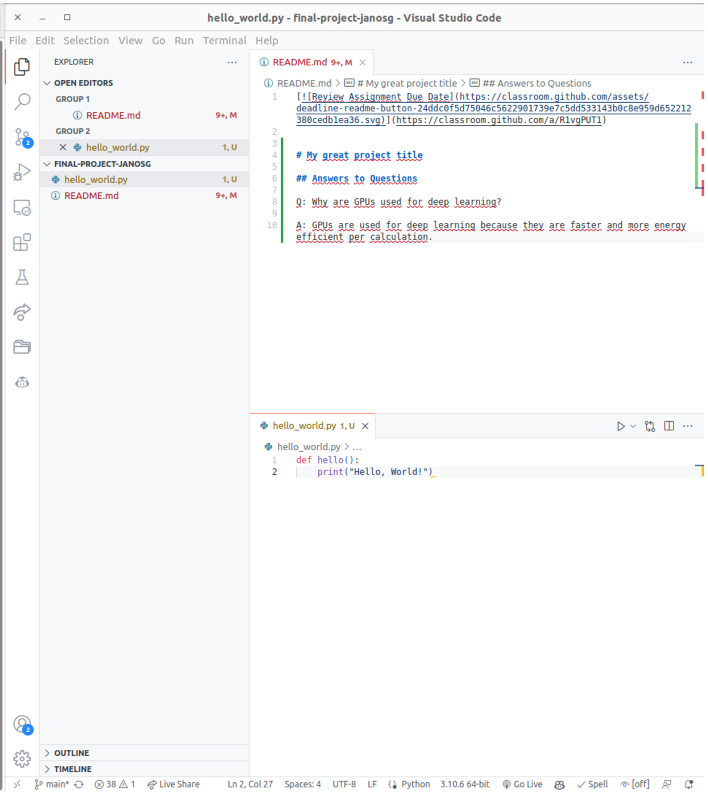
Status after changing and creating files



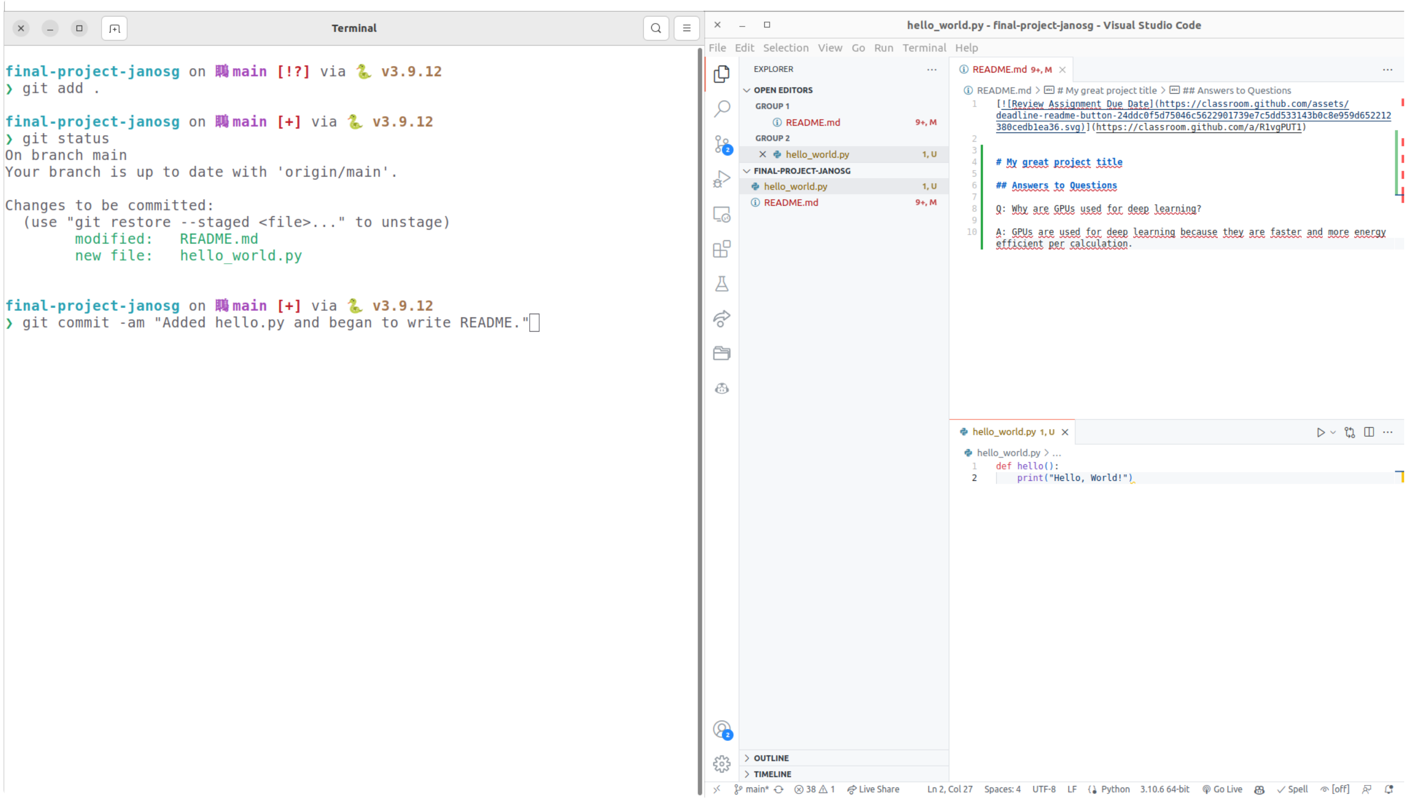
Use `git add .`


Terminal
final-project-janosg on main [!?] via v3.9.12
❯ git add .
hello_world.py - final-project-janosg - Visual Studio Code
File Edit Selection View Go Run Terminal Help
EXPLORER
OPEN EDITORS
GROUP 1
README.md 9+, M
GROUP 2
hello_world.py 1, U
FINAL-PROJECT-JANOSG
hello_world.py 1, U
README.md 9+, M
My great project title
Answers to Questions
Q: Why are GPUs used for deep learning?
A: GPUs are used for deep learning because they are faster and more energy efficient per calculation.
def hello():
print("Hello, World!")
OUTLINE
TIMELINE

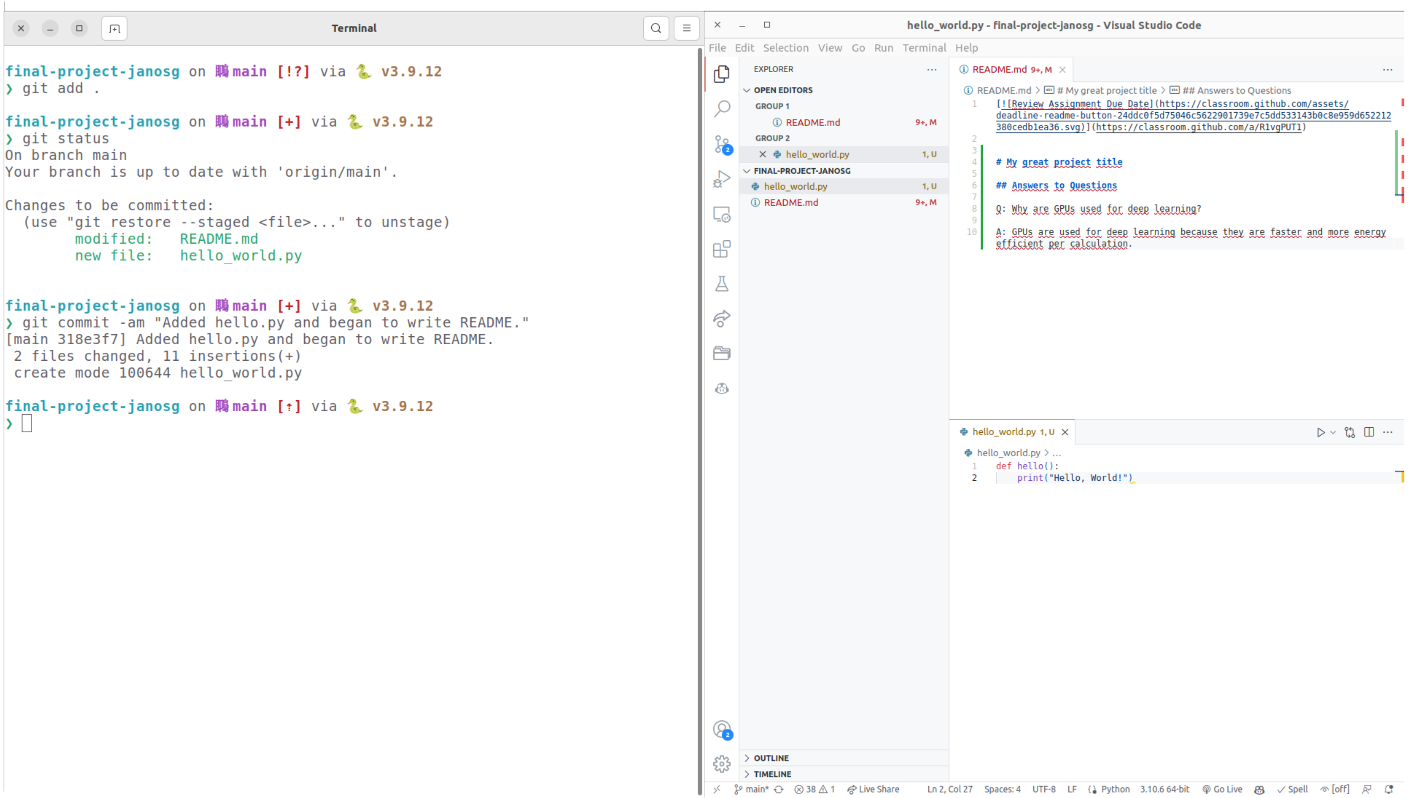
Status after `git add .`



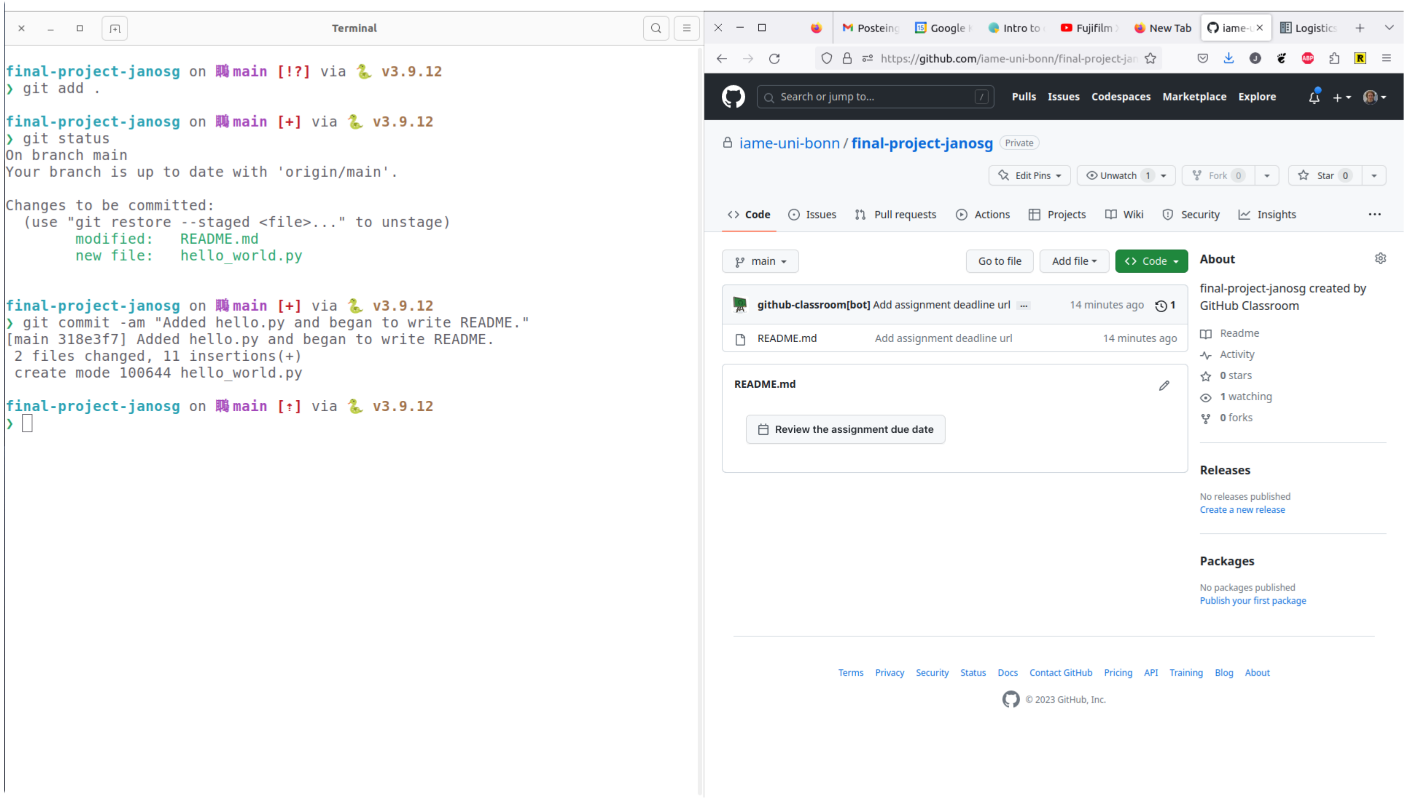
Type commit command with message



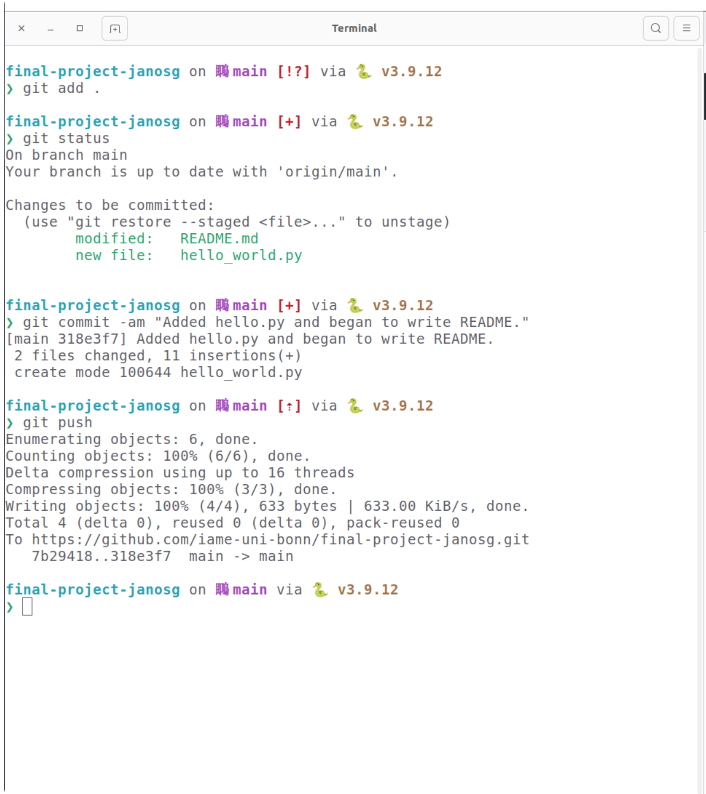
Execute commit

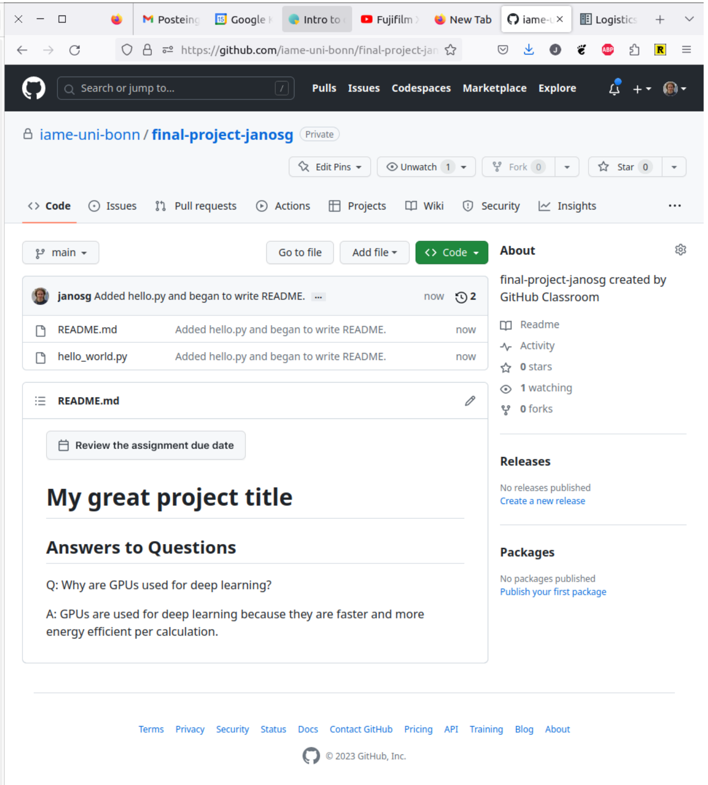


Commit does not publish changes



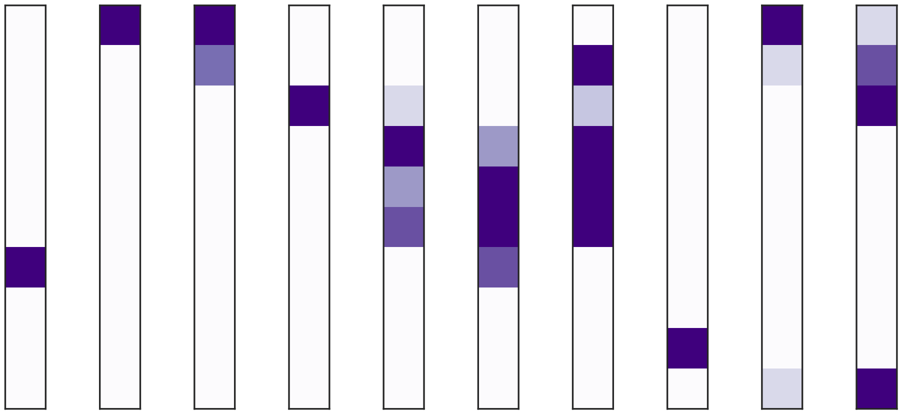
Git push


Terminal
final-project-janosg on main [!?] via v3.9.12
❯ git add .
final-project-janosg on main [+] via v3.9.12
❯ git status
On branch main
Your branch is up to date with 'origin/main'.
Changes to be committed:
(use "git restore --staged <file>..." to unstage)
modified: README.md
new file: hello_world.py
final-project-janosg on main [+] via v3.9.12
❯ git commit -am "Added hello.py and began to write README."
[main 318e3f7] Added hello.py and began to write README.
2 files changed, 11 insertions(+)
create mode 100644 hello_world.py
final-project-janosg on main [⇡] via v3.9.12
❯ git push
Enumerating objects: 6, done.
Counting objects: 100% (6/6), done.
Delta compression using up to 16 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (4/4), 633 bytes | 633.00 KiB/s, done.
Total 4 (delta 0), reused 0 (delta 0), pack-reused 0
To https://github.com/iame-uni-bonn/final-project-janosg.git
7b29418..318e3f7 main -> main
final-project-janosg on main via v3.9.12
❯


https://github.com/iame-uni-bonn/final-project-jan
Search or jump to...
Pulls
Issues
Codespaces
Marketplace
Explore
iame-uni-bonn / final-project-janosg
Private
Edit Pins
Unwatch 1
Fork 0
Star 0
Code
Issues
Pull requests
Actions
Projects
Wiki
Security
Insights
main
Go to file
Add file
Code
janosg Added hello.py and began to write README.
now
2
README.md
Added hello.py and began to write README.
now
hello_world.py
Added hello.py and began to write README.
now
README.md
Review the assignment due date
My great project title
Answers to Questions
Q: Why are GPUs used for deep learning?
A: GPUs are used for deep learning because they are faster and more energy efficient per calculation.
About
final-project-janosg created by GitHub Classroom
Readme
Activity
0 stars
1 watching
0 forks
Releases
No releases published
Create a new release
Packages
No packages published
Publish your first package
Terms
Privacy
Security
Status
Docs
Contact GitHub
Pricing
API
Training
Blog
About
© 2023 GitHub, Inc.

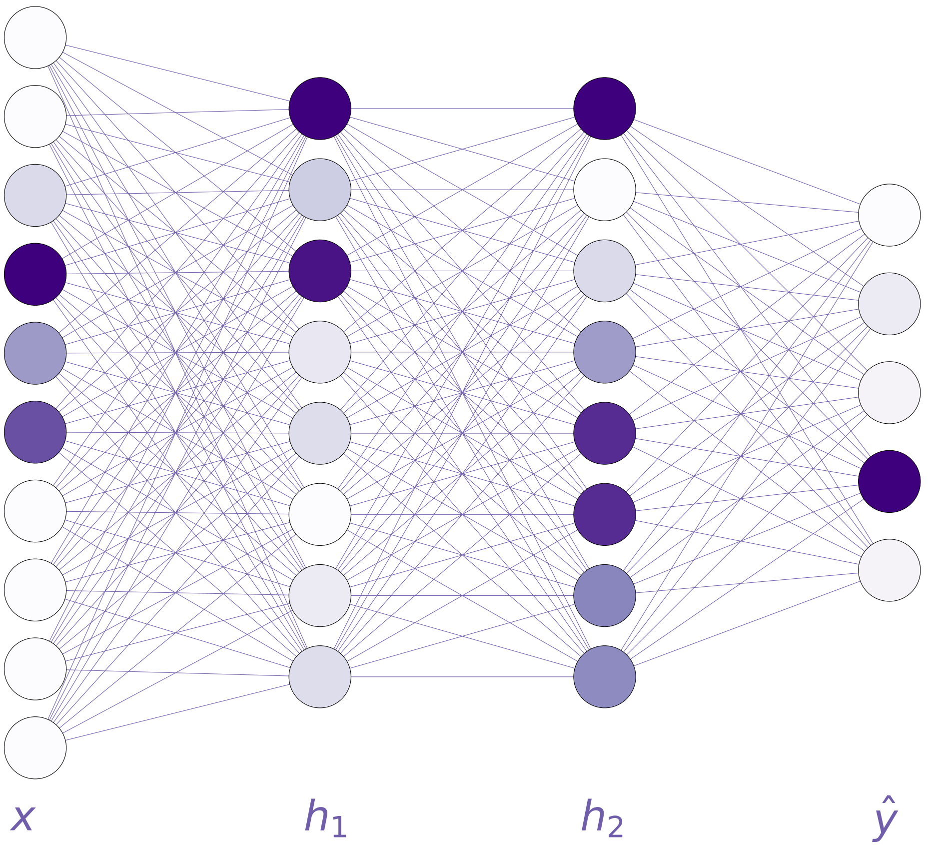
Recap: Feed-forward neural networks

Line length recognition



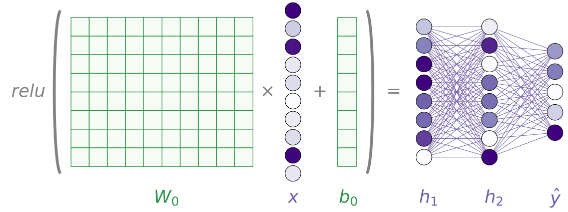
- Images of 10 x 1 Pixels with lines of length 1 to 5
- Task: Estimate the length of the line
- Can see correct result from flattened image

The Multi Layer Perceptron

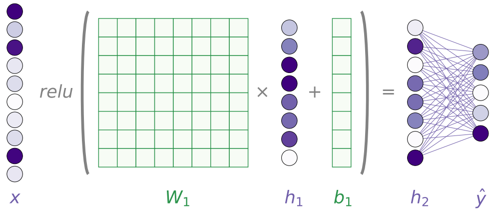


Written out matrix multiplications

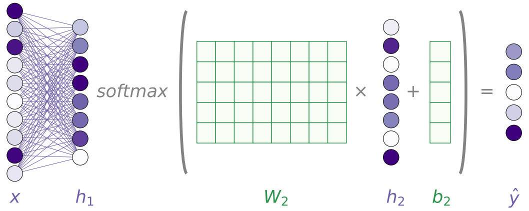
$$h_1 = relu(W_0 x + b_0)$$



$$h_2 = relu(W_1 h_1 + b_1)$$



$$\hat{y} = softmax(W_2 h_2 + b_2)$$



Why do we need nonlinearities

- Without nonlinearities, our model would be:

$$W_2(W_1(W_0x + b_0) + b_1) + b_2$$

- This could be simplified to:

$$Wx + b$$

- Thus we would end up with one linear model!

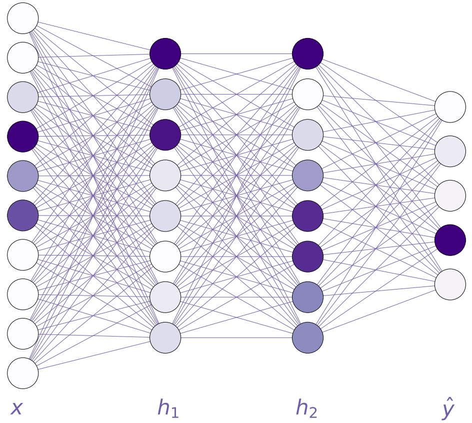
Trainable parameters

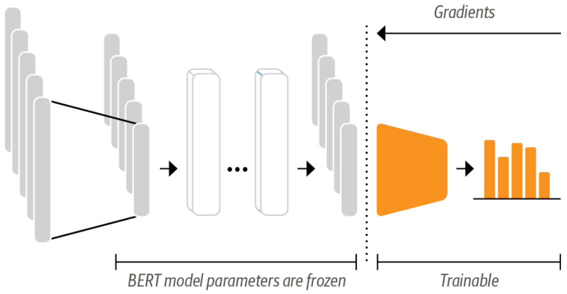
- Weights have shapes:
  - $n^{hidden} \times n^{in}, n^{hidden} \times n^{hidden}, n^{out} \times n^{hidden}$
- Biases have shapes:
  - $n^{hidden}, n^{hidden}, n^{out}$
- 205 for line lengths example
- 13 002 for digit recognition
- ~60 Million in the model we fine-tuned
- 175 Billion in GPT-3

Why are neural networks so porwerful?

- Networks get their power from training!
- Done with some form of gradient descent
- Last week, we did it from scratch
- This week, we will look at simpler ways

How does it relate to feature extraction

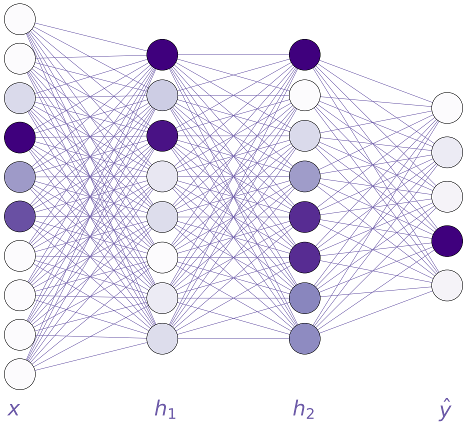


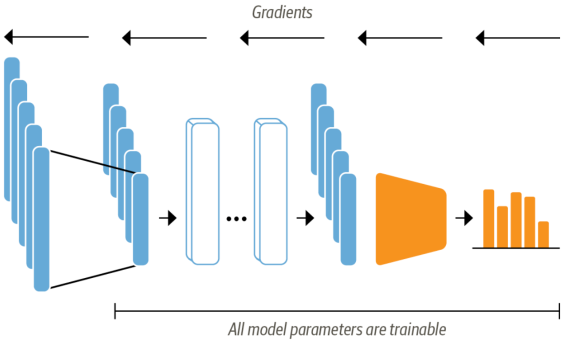


Source: Natural Language Processing with transformers, Fig 2-4

- Parameters are not initialized randomly but pre-trained
- Only W_2 and b_2 are specialized to classification task

How does it relate to fine-tuning

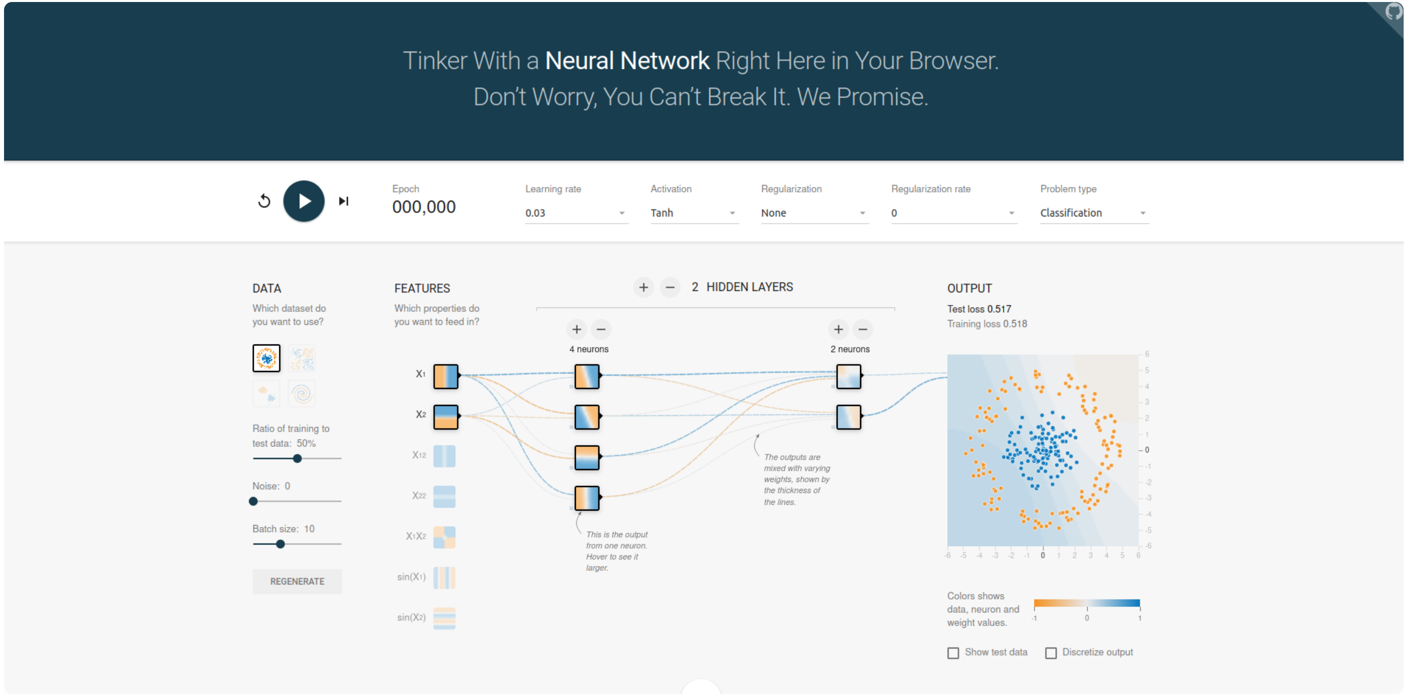




Source: Natural Language Processing with transformers, Fig 2-6

- Parameters are not initialized randomly but pre-trained
- All parameters are specialized to classification task

Tensorflow Playground



Use pytorch properly

What did we do last week?

- Implement entire training process from scratch
  - The model
  - The optimizer
  - The loss function
  - The training loop
- Why did we do this?
  - Need to know the mechanics of a neural network
  - It will be easier to understand the built-in pytorch functions after you did the same steps from scratch

What was annoying last week?

- Had to initialize weights and biases with correct shape
- Several operations on each parameter tensor
  - Set requires_grad to True
  - (Put on GPU)
  - Update with gradients and zero_ gradients
- Had to know the mechanics of a lot of functions
  - relu, softmax, nll_loss, gradient descent
- Had to think about numerical stability
- Our code was slow

Goal for this lecture: Simpler training

```
# training hyperparameters
n_epochs = 3
batch_size = 64
learning_rate = 0.01

# initialization
model = NeuralNetwork(n_in, n_hidden, n_out).to(device)
loss_func = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
train_dataloader = DataLoader(
    training_data, batch_size=batch_size, shuffle=True, drop_last=True,
)
test_dataloader = DataLoader(
    test_data, batch_size=batch_size, drop_last=True,
)

# training loop
for t in range(n_epochs):
    print(f"Epoch {t+1}\n-------------------------------")
    train_loop(train_dataloader, model, loss_func, optimizer)
    test_loop(test_dataloader, model)
print("Done!")
```

Approach

- Good news
  - After today you can put pytorch on your CV
  - Some of the snippets you write can be re-used in other projects
- Bad news
  - You will have to learn quite a few new concepts
- Approach
  - I show you the final training loop at the beginning
  - You practice all the building blocks in exercises
  - (You don’t have to write the final loop yourself today)

Steps

1. Defining a model
2. Define DataLoaders (should be called batch loaders)
3. Loss functions
4. Optimizers
5. Inner training loop
6. Inner test loop

Defining models

Purpose of the model

The pytorch model replaces two steps we did manually

- Initializing the parameters (and defining their shapes)
- Defining the calculations done in the model

Models are classes

- Defining a model = defining a class
- Class specifies:
  - Number, type and shape of layers (similar to defining the parameter matrices)
  - Forward method (similar to our model function)
- Class inherits other methods
  - Initialize all parameters
  - Put all parameters on a device
  - ...

Classes in Python

```
class Circle:
    def __init__(self, x, y, radius):
        self.x = x
        self.y = y
        self.radius = radius

    def area(self):
        return np.pi * self.radius ** 2

    def diameter(self):
        return 2 * self.radius

    def __repr__(self):
        return f"Circle at x={self.x}, y={self.y} with rad
```

```
circle = Circle(0, 0, 1)
circle
```

```
Circle at x=0, y=0 with radius 1
```

```
circle.area()
```

```
3.141592653589793
```

- Defined with `class` keyword
- Class = bundle of methods and data
- Methods are like functions but often have no arguments beyond the class attributes
- `__dunder__` methods are special
- Class != instance
  - Class: Blueprint
  - Instance: Has concrete values for attributes

The `__init__` method

```python
class Circle:
    def __init__(self, x, y, radius):
        print("I was called")
        self.x = x
        self.y = y
        self.radius = radius
```

```python
circle = Circle(0, 0, 1)
```

```
print("I was called")
```

- `__init__` sets up the model instance
- arguments of `__init__` become arguments of class
- Can execute arbitrary code here
- First argument is always `self`
- See `self` as a flexible data container where you can store and retrieve stuff and to which you have access in all methods

Methods

```python
class Circle:
    def __init__(self, x, y, radius):
        self.x = x
        self.y = y
        self.radius = radius

    def area(self):
        return np.pi * self.radius ** 2
```

```python
circle = Circle(0, 0, 1)
circle.area()
```

```
3.141592653589793
```

- Methods are defined like functions
- First argument is always `self`
- Can have additional arguments
- When calling the method, `self` is passed automatically
- Inside methods you can do everything you can do inside functions

Anatomy of a Pytorch Model

```
class NeuralNetwork(nn.Module):
    def __init__(self, n_in, n_hidden, n_out):
        super().__init__()
        ...

    def forward(self, x):
        ...
        return logits
```

- Pytorch models are subclasses of `nn.Module`
- Have two mandatory methods:
  - `__init__` which calls the init method of it's superclass and defines shapes
  - `__forward__` Which does what our model function did

Writing the `__init__` method

```python
class NeuralNetwork(nn.Module):
    def __init__(self, n_in, n_hidden, n_out):
        super().__init__()
        self.flatten = nn.Flatten()
        self.all_layers = nn.Sequential(
            nn.Linear(n_in, n_hidden),
            nn.ReLU(),
            nn.Linear(n_hidden, n_out),
        )

    def forward(self, x):
        x = self.flatten(x)
        logits = self.all_layers(x)
        return logits
```

- Similar model as last week but only one hidden layer

- Always call `super().__init__()`
- Assign functions we need later to `self`
  - `flatten`: Go from (28x28) to (784)
  - `all_layers`: Model calculations
- Use built-in pytorch functions
  - `nn.Sequential` chains functions
  - `Linear` and `ReLU`
- Each layer knows how many parameters it needs and registers them with the class

Writing the forward function

```
class NeuralNetwork(nn.Module):
    def __init__(self, n_in, n_hidden, n_out):
        super().__init__()
        self.flatten = nn.Flatten()
        self.all_layers = nn.Sequential(
            nn.Linear(n_in, n_hidden),
            nn.ReLU(),
            nn.Linear(n_hidden, n_out),
        )

    def forward(self, x):
        x = self.flatten(x)
        logits = self.all_layers(x)
        return logits
```

- Forward does the actual calculation
- Mainly calls functions that were assigned to self before
- Should return logits, i.e. not take softmax yet

Using the model

```
n_in = 28 * 28
n_hidden = 16
n_out = 10
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

model = NeuralNetwork(n_in, n_hidden, n_out).to(device)
model
```

```
NeuralNetwork(
  (flatten): Flatten(start_dim=1, end_dim=-1)
  (all_layers): Sequential(
    (0): Linear(in_features=784, out_features=16, bias=True)
    (1): ReLU()
    (2): Linear(in_features=16, out_features=10, bias=True)
  )
)
```

- The model class worked for general shapes, the instance has specific ones
- Model has a printable and human readable text summary
- Can put all tensors belonging to the model on the GPU with one method

Looking at model parameters

```
list(model.parameters())
```

```
[Parameter containing:
 tensor([[ 0.0057, ..., -0.0129],
         ...,
         [ 0.0142, ..., -0.0051]],
        requires_grad=True),
 Parameter containing:
 tensor([-0.0002, ..., -0.0232],
        requires_grad=True),
 ...
 ]
```

- Each layer knew which parameters it needs
- Registered them automatically with the model
- They get initialized randomly behind the scenes
- Optimizers will work on this parameter list
- You don’t ever have to look at them

What else is in `torch.nn`?

- Many different nonlinearities
- Many different layers
  - Convolutional layers
  - Recurrent layers
  - Transformers
- Loss functions
- See the documentation for details

Task 1

DataLoaders

Purpose of the DataLoaders

The DataLoader serves two purposes

- Make it easy to loop over batches of the data
- (Make it possible to load data in parallel)

How we did it last time

```
batch_indices = torch.randperm(len(data)).reshape(-1, batch_size)
for idxs in batch_indices:
    batch = data[idxs]
```

- Did not work if dataset size was not a multiple of batch size
- Hard to read if you don’t know the trick

DataLoaders

```
batch_size = 64
train_dataloader = DataLoader(
    training_data,
    batch_size=batch_size,
    shuffle=True,
    drop_last=True,
)

for i, (X, y) in enumerate(train_dataloader):
    logits = model(X)
    ...
```

```
len(train_dataloader)
```

```
937
```

```
len(train_dataloader.dataset)
```

```
60000
```

- the dataloader is something you can iterate over
- mechanics of shuffling and batching are abstracted away
- Enable or disable shuffling
- `` `drop_last` `` is how we handle dataset length that are not multiples of batch size

Task 2

Loss functions

What we did last week

```
def nll_loss(probs, labels):
    likelihoods = probs[torch.arange(len(probs)), labels] + 1e-50
    loglikes = torch.log(likelihoods)
    return -loglikes.mean()
```

- Needed to bother with the mechanics of indexing
- Very crude way of ensuring numerical stability

Pre-implemented loss function

Docs > torch.nn

Loss Functions

`nn.L1Loss`	Creates a criterion that measures the mean absolute error (MAE) between each element in the input x and target y.
`nn.MSELoss`	Creates a criterion that measures the mean squared error (squared L2 norm) between each element in the input x and target y.
`nn.CrossEntropyLoss`	This criterion computes the cross entropy loss between input logits and target.
`nn.CTCLoss`	The Connectionist Temporal Classification loss.
`nn.NLLLoss`	The negative log likelihood loss.
`nn.PoissonNLLLoss`	Negative log likelihood loss with Poisson distribution of target.
`nn.GaussianNLLLoss`	Gaussian negative log likelihood loss.
`nn.KLDivLoss`	The Kullback-Leibler divergence loss.
`nn.BCELoss`	Creates a criterion that measures the Binary Cross Entropy between the target and the input probabilities:
`nn.BCEWithLogitsLoss`	This loss combines a *Sigmoid* layer and the *BCELoss* in one single class.
`nn.MarginRankingLoss`	Creates a criterion that measures the loss given inputs $x1$, $x2$, two 1D mini-batch or 0D *Tensors*, and a label 1D mini-batch or 0D *Tensor* y (containing 1 or -1).
`nn.HingeEmbeddingLoss`	Measures the loss given an input tensor x and a labels tensor y (containing 1 or -1).

- Many loss functions are pre-implemented
- Numerically stable implementations
- We will use `` `CrossEntropyLoss` ``
- See the documentation for details

```
loss_func = nn.CrossEntropyLoss()
loss_func(logits, y)
```

```
tensor(2.3018, grad_fn=<NllLossBackward0>)
```

Task 3

Optimizers

What we did last week

```
for i in range(3):
    # SGD updates for each parameter
    weights[i].data = weights[i].data - learning_rate * weights[i].grad.data
    biases[i].data = biases[i].data - learning_rate * biases[i].grad.data
    # Zero the gradients for the next iteration
    weights[i].grad.data.zero_()
    biases[i].grad.data.zero_()
```

- Had to know the gradient descent update equation
- Had to do things per-parameter
- Had to know when to use `` `tensor` `` and `` `tensor.data` ``

Optimizers in pytorch

```
learning_rate = 0.1
optimizer = torch.optim.SGD(model.parameters(), lr=learnin
```

In the training loop:

```
...
loss.backward()
optimizer.step()
optimizer.zero_grad()
```

- Optimizer instance knows which parameter tensors are there
- Automatically applies relevant steps to all of them
- Many optimizers available

The optimization problem

- $\theta \in \mathcal{R}^d$ is a vector of parameters
- $Z \in \mathcal{R}^{n \times m}$ is a matrix containing a batch of data (x and y)
- $\ell(\theta, Z)$ is a scalar loss function
- $j(\theta, Z)$ is the gradient of ℓ w.r.t. θ
- Goal: $min_\theta \ell(\theta)$
- In words: find parameters of the neural net that minimize the loss function

SGD

- Tuning parameters:
  - η: learning rate, typically 1e-3
- Update equation
  - $\theta_{k+1} = \theta_k - \eta \cdot j(\theta_k, Z)$
- Problems
  - Get’s stuck in local optima (gradient = 0)
  - Learning rate is the same for all parameters
  - Learning rate is hart to pick

SGD + Momentum

- Tuning parameters:
  - η: learning rate, typically 1e-3
  - γ: momentum parameter, typically 0.9
- Update equations:
  - $\nu_k = \gamma \nu_{k-1} + \eta \cdot j(\theta_k, Z)$
  - $\theta_{k+1} = \theta_k - \nu_k$
- Advantage
  - Can pass over a local flat spot
  - Less oscillation around the main direction of progress

Adam (Adaptive Moment Estimation)

- Tuning parameters
  - η: learning rate, typically 1e-3
  - β_1: momentum in gradients, typically 0.9
  - β_2: momentum in squared gradients, typically 0.99
  - ϵ: clipping value, typically 1e-8
- Update equations
  - $m_k = \frac{1}{1-\beta_1} \cdot [\beta_1 m_{k-1}) + (1-\beta_1) j(\theta_k, Z)]$
  - $v_k = \frac{1}{1-\beta_2} \cdot [\beta_2 v_{k-1} + (1-\beta_2) j(\theta_k, Z)^2]$
  - $\theta_{k+1} = \theta_k - \frac{\eta}{\sqrt{v_k}+\epsilon} m_k$

Adam

- Adam behaves like a heavy ball with friction, which thus prefers flat minima in the error surface
- Widely used in practice (e.g. for GPT)
- Often together with a learning rate schedule, where η decays over time

Task 4

Inner training loop

Remember the goal

```python
# training hyperparameters
n_epochs = 3
batch_size = 64
learning_rate = 0.01

# initialization
model = NeuralNetwork(n_in, n_hidden, n_out).to(device)
loss_func = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
train_dataloader = DataLoader(
    training_data, batch_size=batch_size, shuffle=True, drop_last=True,
)
test_dataloader = DataLoader(
    test_data, batch_size=batch_size, drop_last=True,
)

# training loop
for t in range(n_epochs):
    print(f"Epoch {t+1}\n-------------------------------")
    train_loop(train_dataloader, model, loss_func, optimizer)
    test_loop(test_dataloader, model)
print("Done!")
```

The inner `train_loop

```
def train_loop(dataloader, model, loss_fn, optimizer):
    model.train()

    # loop over the dataloader
        # evaluate model
        # evaluate loss
        # backpropagate
        # call optimizer step
        # zero the gradients
```

- `model.train()` Puts the model in training mode
- Best practice: Do it for any model, even if it does not make a difference
- Some models have layers that behave differently during training and inference

Task 5

Remember the goal

```
# training hyperparameters
n_epochs = 3
batch_size = 64
learning_rate = 0.01

# initialization
model = NeuralNetwork(n_in, n_hidden, n_out).to(device)
loss_func = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
train_dataloader = DataLoader(
    training_data, batch_size=batch_size, shuffle=True, drop_last=True,
)
test_dataloader = DataLoader(
    test_data, batch_size=batch_size, drop_last=True,
)

# training loop
for t in range(n_epochs):
    print(f"Epoch {t+1}\n-------------------------------")
    train_loop(train_dataloader, model, loss_func, optimizer)
    test_loop(test_dataloader, model)
print("Done!")
```

Inner test loop

```
def test_loop(dataloader, model):
    model.eval()

    correct = 0
    # use torch.no_grad()
        # loop over data
            # calculate predictions
            # add te number of correct predictions to corr
    # calculate accuracy
    # print accuracy
```

- Want to have similar monitory as last time
- Print accuracy on the test data after each epoch

Task 6

Task 7

Summary

- We solved the same problem as last week but this time used pytask properly
- Not less code, but it scales to larger model
- Using the ingredients you learned today, you can build large neural networks and train them on CPU or GPU
- There are many different optimizers
- If in doubt, use Adam