Quantitative Macroeconomics w/ AI and ML
Lec. 3: Programming Basics for Economists

Zhigang Feng

Dec., 2025

Overview

Today's Goal

By the end of this lecture, you should be able to:

- Understand the **economist's tech stack**: OS, Python, IDEs, libraries, and AI assistants (Cursor).
- Read and write basic **Python** code:
  - data types, lists, dictionaries, loops, functions, classes.
- Follow a typical **data science workflow**:
  - download, clean, save, and visualize data (e.g., FRED GDP series).
- Get a **first flavor** of **PyTorch** and the ML workflow:
  - tiny neural net: predict next-quarter GDP growth from last 4 quarters.
  - ML theory is next lecture; today is about *how the code fits together*.

The Economist's Tech Stack

How the Pieces Fit Together

- **1. Operating System (OS)**
  - Windows, macOS, Linux.
  - Manages hardware (CPU, RAM, file system).
  - Python and all tools run *on top* of this.
- **2. Python**
  - High-level programming language for data analysis, simulations, and ML.
  - You install it (e.g., via **Anaconda** or `python.org`).
- **3. IDE / Notebook**
  - Where you edit and run code:
  - **Jupyter**, **VS Code/Cursor**, or **PyCharm**.

Jupyter, VS Code/Cursor, and PyCharm

Jupyter (Notebooks)

- Run code **cell-by-cell**.
- Great for:
  - exploration,
  - data cleaning,
  - plotting.
- Immediate feedback: results appear under each cell.
- Comes with Anaconda as **JupyterLab**.
- Recommended for:
  - homework,
  - small experiments,
  - demonstrations in this course.

VS Code/Cursor and PyCharm

- **VS Code/Cursor**:
  - general-purpose editor/IDE,
  - becomes a full Python IDE with the Python extension,
  - excellent for both **small** and **large** projects.
- **PyCharm**:
  - dedicated **Python IDE**,
  - more Python-specific tools out of the box (refactoring, inspections),
  - great for bigger research codebases.
- Both can handle complex projects. Choice is about personal taste and features.

Python, PyTorch, and AI: How They Relate

- **Python**:
  - General-purpose language.
  - You use it for **data cleaning, regression, simulations, plotting**.
- **PyTorch**:
  - A **Python library** specialized for:
    - tensors (multi-dimensional arrays),
    - automatic differentiation (autograd),
    - neural networks (deep learning).
- **AI / ML Models**:
  - Built **in PyTorch using Python**.
  - Examples:
    - time series forecasters,
    - macro-finance deep learning models,
    - text models for central bank communications.

Takeaway
Python is the language. PyTorch is the toolbox. AI / ML models are what you *build* with that toolbox.

AI Assistants in Your Stack (Cursor, Copilot, . . .)

- **AI Coding Tools**:
  - Cursor, GitHub Copilot, ChatGPT-style tools.
  - Think of them as **supercharged TAs**:
    - complete boilerplate code,
    - explain error messages,
    - suggest better structure,
    - translate math into code.
- Use them especially when:
  - installation / environment is broken,
  - you see a mysterious error,
  - you are implementing a new algorithm from a paper.

Using Cursor Effectively for This Class

- **Cursor** = VS Code-like editor with AI deeply integrated.
- Helpful modes:
  - **Inline suggestions**: completes loops, function definitions.
  - **Chat panel**: ask questions with your code context.
  - **Explain / Fix**: highlight code → ask "Explain this" or "Fix this error".

Example Prompts in Cursor

Debugging:

```
I'm trying to run this PyTorch training loop, but I get a shape mismatch error.
Here is the full code and error.  Explain what's wrong and show a fixed version.
```

Code comprehension:

```
Explain this class in simple terms as if I were an economics student.
What are the inputs, outputs, and what does forward() do?
```

Environment help:

```
My Python environment has both conda and pip.  How should I install pandas and torch
correctly on macOS?
```

Installing Python and Tools

Anaconda, Python, and Editors: Who Does What?

- **Anaconda**:
  - Distribution that installs:
    - Python interpreter,
    - many scientific packages (NumPy, pandas, etc.),
    - environment manager (`conda`),
    - **JupyterLab** as a notebook interface.
  - It does *not* replace an editor: you still need a place to write code.
- **Editors/IDEs**:
  - **JupyterLab** (browser) — comes with Anaconda.
  - **VS Code/Cursor** — external editor; can use Anaconda's Python.
  - **PyCharm** — external Python IDE; can also use Anaconda envs.

Mental model
Anaconda = **Python + packages + env manager**.

VS Code/Cursor/PyCharm = **where you write, run, and debug your code**.

Installing Python (Practical Version)

- **Easiest path (recommended): Anaconda**
  - Download from `https://www.anaconda.com/download`
  - Includes Python + many scientific packages (NumPy, pandas, etc.).
- **Alternative: python.org**
  - Download from `https://www.python.org/downloads/`
  - On Windows: check "`Add Python to PATH`".
- **Verify installation** in terminal / command prompt:

```
python --version      # or python3 --version
```

- If this fails: **ask AI (Cursor / ChatGPT) for help** using a clear prompt and include the full error.

Managing Packages

- Use pip (or conda if using Anaconda) to install libraries:

```
# Using pip
pip install pandas pandas-datareader matplotlib seaborn torch

# Using conda (Anaconda)
conda install pandas matplotlib seaborn
conda install -c pytorch pytorch
```

- You install each package **once per environment**, not every time you run the code.
- If installation errors appear, copy the error text and ask:
    - a classmate,
    - AI (with full error),
    - or the instructor.

Python Basics: Data Types & Structures

Atomic Data Types

Smallest units of data in Python:

- **Integers (`int`)**: whole numbers.

```
year = 2025
```

- **Floats (`float`)**: real numbers (decimals).

```
inflation_rate = 0.035
```

- **Strings (`str`)**: text.

```
country = "United States"
```

- **Booleans (`bool`)**: logical truth values.

```
is_recession = False
```

Working with Text (Strings)

Economists often clean country names, tickers, etc.

```
country_raw = "  united states  "
gdp = 23.5

# f-strings: modern way to format text
print(f"The GDP of {country_raw} is {gdp} trillion.")
# Output: "The GDP of   united states   is 23.5 trillion."

# Cleaning methods
country_clean = country_raw.strip().title()
print(country_clean)
# Output: "United States"
```

Lists: Ordered Collections

Lists behave like columns of numbers in Excel.

```
# Quarterly GDP growth (%)
gdp_growth = [1.2, 2.5, 3.1, -0.5]

# Indexing (starts at 0)
first = gdp_growth[0]       # 1.2

# Slicing
subset = gdp_growth[1:3]    # [2.5, 3.1]

# Modifying
gdp_growth[3] = 0.1         # replace -0.5 with 0.1
```

Warning: Reference vs Copy

Important: variable names are just *labels*, not boxes.

```
a = [1, 2, 3]
b = a             # b points to the SAME list
b[0] = 99

print(a)          # [99, 2, 3]
```

To make an independent copy:

```
c = a.copy()    # now c is separate
```

This idea (sharing vs copying) also appears with **NumPy arrays** and **PyTorch tensors**.

Dictionaries map **keys** to **values**, like labeled columns.

```
country_data = {
    "name": "Japan",
    "gdp": 4.2,              # trillion USD
    "population": 125,       # million
    "currency": "Yen"
}

print(country_data["gdp"])          # 4.2
country_data["income_per_cap"] = (
    country_data["gdp"] * 1e12 / (125e6)
)
```


Think of this as a tiny JSON object. Pandas DataFrames are like **tables of dictionaries**.

- **Tuples** `()`: like lists, but **immutable**.

```python
coords = (41.25, -95.93)   # (lat, lon)
```

  Useful for fixed data (e.g., parameters, coordinates, (year, quarter)).

- **Sets** `{}`: unordered collection of **unique** items.

```python
industry_codes = ["A", "B", "A", "C"]
unique_codes = set(industry_codes)   # {"A", "B", "C"}
```

  Great for removing duplicates or checking membership.

Python Basics: Logic, Functions, Modules

Operators

- **Arithmetic**: `+`, `-`, `*`, `/`
- Power: `**` (e.g., `2**3 = 8`)
- Modulo: `%` (remainder, e.g., `17%4 = 1`)
- **Comparison**: `==`, `!=`, `>`, `<`, `>=`, `<=`
- **Logical**: `and`, `or`, `not`

If / Elif / Else

```python
x = -1

if x > 0:
    print("Positive")
elif x < 0:
    print("Negative")
else:
    print("Zero")
```

For Loop

```python
gdp_growth = [1.2, 2.5, -0.5]

for g in gdp_growth:
    if g < 0:
        print("Recession quarter!")
```

While Loop

```python
count = 0
while count < 3:
    print(count)
    count += 1
```

List Comprehensions

Compact way to create lists. Like set-builder notation.

- Math: $S = \{x^2 \mid x \in \mathbb{N}, x > 0\}$
- Python:

```python
data = [1, 2, 3, 4, 5]

# Old way
squares = []
for x in data:
    squares.append(x**2)

# Pythonic way
squares = [x**2 for x in data]

# Filter + transform
gdp_growth = [1.2, 2.5, -0.5, 0.3]
positive_growth = [g for g in gdp_growth if g > 0]
```

Error Handling: try / except

Economic data is messy. Use `try/except` to avoid crashes.

```python
values = [100, 200, 0, 300]

for v in values:
    try:
        ratio = 10 / v
        print(f"Ratio: {ratio}")
    except ZeroDivisionError:
        print("Error: Division by zero, skipping...")
```

You can also catch `ValueError` when parsing strings from CSV files.

Functions: Reusable Blocks of Logic

```python
def cobb_douglas(k, l, alpha=0.3):
    """Output Y = K^alpha * L^(1-alpha)"""
    y = k**alpha * l**(1 - alpha)
    return y

# Usage
output = cobb_douglas(k=100, l=50)
print(output)
```

- **Inputs** are arguments (`k, l, alpha`).
- **Outputs** come from `return`.
- Docstring `"""..."""` explains the function.

Modules and Imports

- **Module**: a `.py` file containing Python code.
- **Package**: collection of modules (e.g., `pandas`, `torch`).

```
import math
print(math.sqrt(16))     # 4.0

import numpy as np
import pandas as pd
```

Custom module example:

```
# my_module.py
def square(x):
    return x * x

# main.py
import my_module
print(my_module.square(5))  # 25
```

Organization: Classes and Objects

Why Economists Need OOP

- **State management**:
  - An agent has wealth, age, employment status, expectations.
  - A class bundles these attributes together.
- **Behavior**:
  - Agent can consume, work, update beliefs, etc.
  - Methods define these behaviors.
- **PyTorch models are classes**:
  - A neural network is a class with:
    - attributes: layers (parameters),
    - methods: `forward` (how to compute outputs).

Anatomy of a Class

```python
class Agent:
    def __init__(self, wealth, age):
        # self = "this instance"
        self.wealth = wealth   # attribute
        self.age    = age      # attribute

    def consume(self, amount):
        # method (behavior)
        self.wealth -= amount

    def birthday(self):
        self.age += 1

# Instantiate (create objects)
a1 = Agent(wealth=1000, age=30)
a2 = Agent(wealth=500,  age=25)

# Call methods
a1.consume(100)
a1.birthday()
```

Without Classes vs With Classes

Without classes (state scattered):

```
# Separate variables
wealth_1 = 1000
age_1    = 30

wealth_2 = 500
age_2    = 25

def consume(wealth, amount):
    return wealth - amount

wealth_1 = consume(wealth_1, 100)
age_1    = age_1 + 1
```

With classes (state + behavior bundled):

```
class Agent:
    def __init__(self, wealth, age):
        self.wealth = wealth
        self.age    = age
```

Inheritance and Why PyTorch Uses It

```python
# Parent class
class GenericModel:
    def predict(self, x):
        print("Predicting...")

# Child class inherits from parent
class MyModel(GenericModel):
    pass

m = MyModel()
m.predict()  # "Predicting..."
```


- In PyTorch, your model will inherit from `nn.Module`:
  - registers parameters automatically,
  - integrates with optimizers and device management,
  - provides `model.parameters()`, `model.to(device)`, etc.

Data Science for Economists

The Data Workflow

Typical pipeline in an empirical macro project:

1. **Acquire** data (FRED, World Bank, CSVs).
2. **Clean**:
   - handle missing values, correct types, adjust dates.
3. **Persist**:
   - save clean data (CSV / pickle) so you do not redo steps 1–2.
4. **Analyze / Visualize**:
   - regressions, summary statistics, plots.
5. **Export**:
   - figures and tables for your LaTeX paper.

Acquire: FRED GDP Data

```python
import pandas_datareader.data as web
import datetime

start = datetime.datetime(1990, 1, 1)
end   = datetime.datetime(2025, 1, 1)

# 'GDP' is the FRED series ID for real GDP (quarterly)
gdp_data = web.DataReader('GDP', 'fred', start, end)

print(gdp_data.head())
```


- pandas_datareader sends a web request to the FRED API.
- FRED identifies each series by its **series ID** (here: 'GDP').
- The response (dates + values) is converted into a **pandas DataFrame**.
- Requires an internet connection. For heavy use, you can obtain an API key from FRED, but many simple calls work without.

Clean: Compute Growth and Filter


```
import pandas as pd
import numpy as np

# Compute quarterly growth rate
gdp_data['growth'] = gdp_data['GDP'].pct_change()

# Drop missing values from the first difference
clean = gdp_data.dropna()

# Example: post-2000
post_2000 = clean[clean.index >= '2000-01-01']

print(post_2000[['GDP', 'growth']].head())
```

Example: HP Filter (Trend and Cycle)

```
import numpy as np
from statsmodels.tsa.filters.hp_filter import hpfilter

# Log GDP
gdp_log = np.log(df['GDP'])

# HP filter with lambda=1600 for quarterly data
cycle, trend = hpfilter(gdp_log, lamb=1600)

df['gdp_trend'] = trend
df['gdp_cycle'] = cycle
```



```
import matplotlib.pyplot as plt

plt.figure(figsize=(10, 5))
plt.plot(df.index, gdp_log, label="log GDP", alpha=0.5)
plt.plot(df.index, df['gdp_trend'], label="HP trend", linewidth=2)
plt.title("HP Filter: log GDP and Trend")
plt.xlabel("Year")
plt.legend()
```

Persist: Saving Clean Data

```python
# Save to CSV for sharing or inspection
post_2000.to_csv("gdp_post2000.csv")

# Save to pickle (Python-specific, preserves types)
post_2000.to_pickle("gdp_post2000.pkl")

# Later: load from CSV
df = pd.read_csv("gdp_post2000.csv",
                 index_col='DATE', parse_dates=True)
```

Visualize: GDP Growth

```python
import matplotlib.pyplot as plt
import seaborn as sns

sns.set_theme(style="whitegrid")

x = df.index
y = df['growth'] * 100  # to percent

plt.figure(figsize=(10, 5))
plt.plot(x, y, label="Quarterly growth (%)")
plt.axhline(0, linestyle='--')
plt.title("US Real GDP Growth")
plt.xlabel("Year")
plt.ylabel("Growth (%)")
plt.legend()
plt.tight_layout()
```

Export: High-Quality Figures


```python
# Save for your paper
plt.savefig("gdp_growth.png", dpi=300,
            bbox_inches='tight')

# Vector graphics (good for LaTeX)
plt.savefig("gdp_growth.pdf", bbox_inches='tight')

plt.show()
```

PyTorch Basics

What is PyTorch?

- **Open-source deep learning library** in Python.
- Key components:
  - **Tensors**: multi-dimensional arrays (like NumPy arrays).
  - **Autograd**: automatic differentiation.
  - **nn.Module**: building neural networks.
  - **optim**: optimizers (SGD, Adam, etc.).
- Widely used for:
  - time series forecasting,
  - macro / finance ML,
  - text and image models.

PyTorch: Applications and Advantages

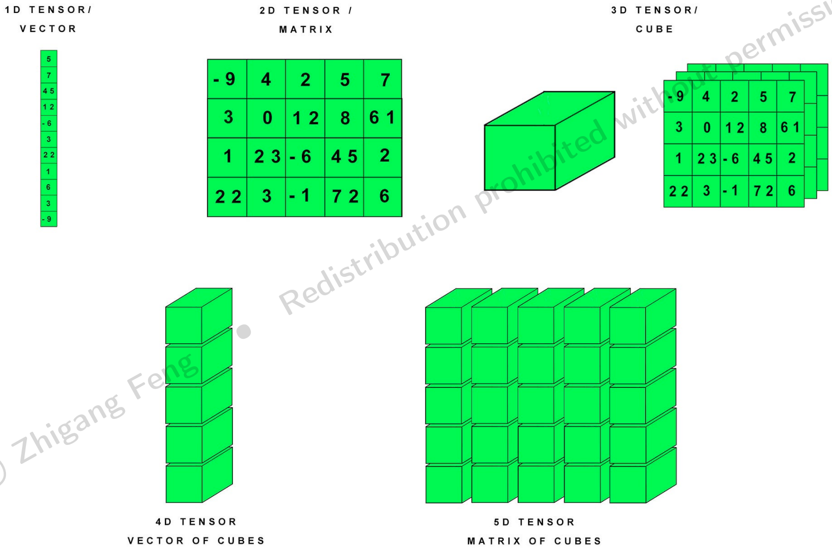
- **Applications**:
  - Computer vision: image classification, object detection.
  - NLP: text classification, language models, machine translation.
  - Reinforcement learning: training agents to interact with environments.
  - Economics: forecasting, structural models with neural nets, text analysis of policy statements.
- **Advantages**:
  - **Dynamic computation graph**:
    - Graph is built at runtime, easy to debug and modify.
  - **GPU acceleration**:
    - seamless move between CPU and GPU for speed.
  - **Active community**:
    - many tutorials, extensions, and research codebases.

Tensors: Like NumPy, but GPU-Ready

```python
import torch

# 1D tensor (vector)
t1 = torch.tensor([1, 2, 3, 4, 5])
print(t1)

# 2D tensor (matrix)
t2 = torch.tensor([[1, 2, 3],
                   [4, 5, 6]])
print(t2)

# Specify dtype
t_float = torch.tensor([1.0, 2.0, 3.0],
                       dtype=torch.float32)

# Zeros
t_zeros = torch.zeros((3, 4))
```


Tensors and NumPy arrays can be converted back and forth.

Tensors (Visual)



Tensors: Summary

- Multi-dimensional arrays with a uniform data type:
  - e.g., all `float32` or all `int64`.
- Similar to NumPy arrays:
  - support element-wise operations, reshaping, indexing, slicing.
  - can be moved between CPU and GPU memory.
- Designed for efficient numerical computation:
  - can handle large matrices and high-dimensional data.

Basic Tensor Operations

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

sum_result     = x + y   # [5, 7, 9]
product_result = x * y   # [4, 10, 18]

# Matrix-style operations
A = torch.tensor([[1.0, 2.0],
                  [3.0, 4.0]])
B = torch.tensor([[5.0, 6.0],
                  [7.0, 8.0]])

C = A @ B  # matrix multiplication
```

Reshaping: view() vs reshape()

```python
x = torch.arange(1, 10)     # [1, 2, ..., 9]

# Want 3x3
y = x.view(3, 3)            # requires x to be contiguous
z = x.reshape(3, 3)         # more flexible

print(y)
# tensor([[1, 2, 3],
#         [4, 5, 6],
#         [7, 8, 9]])
```

- `view`: returns a new tensor that shares memory, but only works if memory layout is contiguous (otherwise you may need `x = x.contiguous().view(...)`).
- `reshape`: tries to return a view; if that fails, it creates a copy.
- In most beginner code, `reshape` is a safe default.

Concatenation and Transpose

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

# Concatenate vertically (rows)
cat_rows = torch.cat((x, y), dim=0)

# Concatenate horizontally (cols)
cat_cols = torch.cat((x, y), dim=1)

# Transpose
xt = x.T   # or torch.transpose(x, 0, 1)
```

Autograd: Conceptual View

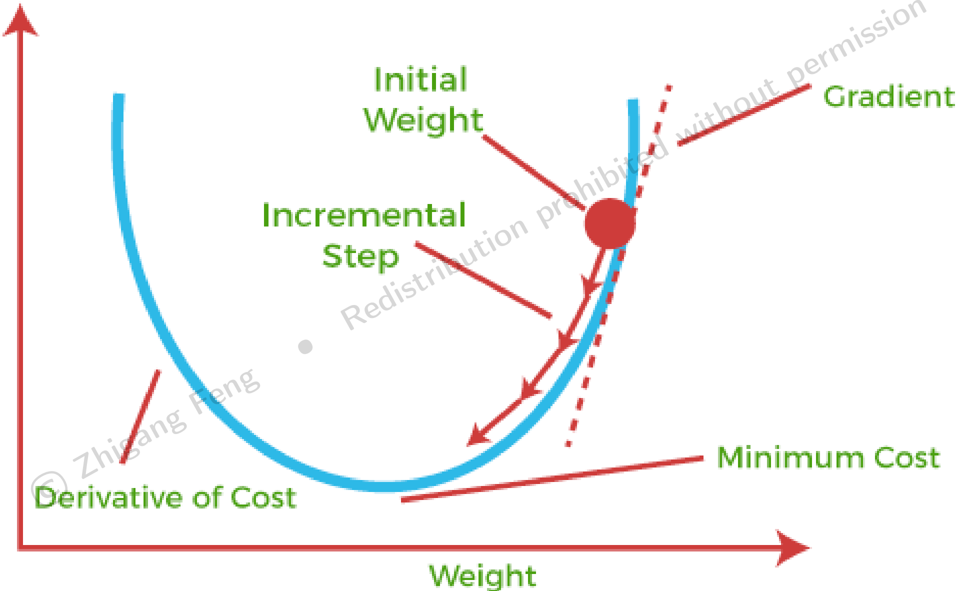
- PyTorch has an **automatic differentiation** engine: `autograd`.
- When you operate on tensors with `requires_grad=True`:
  - PyTorch builds a **computation graph** where:
    - nodes = tensors,
    - edges = operations.
- When you call `backward()` on a scalar output (typically a loss):
  - PyTorch traverses the graph backward,
  - applies the chain rule,
  - computes gradients for all tensors that require them.
- These gradients are stored in the `.grad` attribute of tensors (e.g., model parameters).

Autograd: Simple Example

```python
import torch

x = torch.tensor(2.0, requires_grad=True)
y = x * x                # y = x^2

y.backward()             # dy/dx at x=2
print(x.grad)            # tensor(4.)
```

- Here, y is the **loss** (a scalar).
- backward() computes $\frac{dy}{dx}$.
- In neural nets, parameters (weights, biases) play the role of x.

Neural Networks and Gradient Descent (Visual)



Autograd: Steps for Optimization

1. **Enable gradient tracking**: create tensors with `requires_grad=True` (PyTorch does this for parameters in `nn.Module`).
2. **Forward pass**: compute model output and loss.
3. **Backward pass**: call `loss.backward()` to compute gradients.
4. **Update parameters**: use optimizer (e.g., `optimizer.step()`).
5. **Zero gradients**: call `optimizer.zero_grad()` before the next iteration.

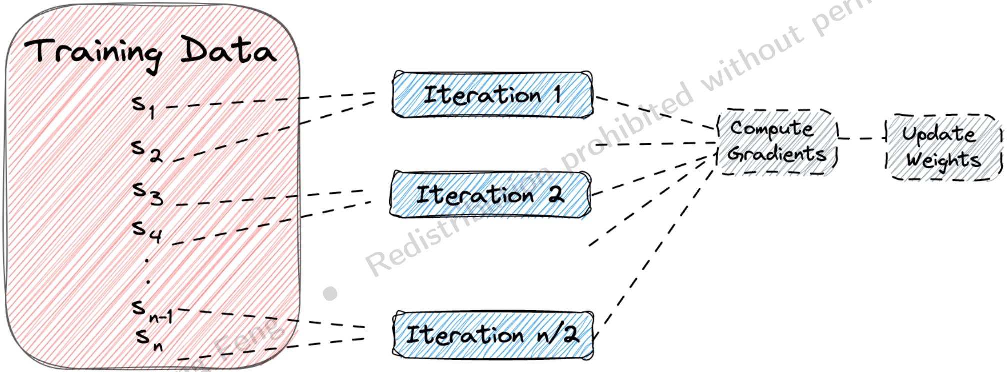
Optimizers and Training Loop Template

```python
import torch.nn as nn
import torch.optim as optim

model = nn.Linear(4, 1)          # simple linear model
optimizer = optim.SGD(model.parameters(), lr=0.01)
loss_fn = nn.MSELoss()

num_epochs = 10
for epoch in range(num_epochs):
    # x_batch, y_batch = ...
    optimizer.zero_grad()        # 1. clear old gradients
    preds = model(x_batch)       # 2. forward
    loss = loss_fn(preds, y_batch)  # 3. loss
    loss.backward()              # 4. backprop
    optimizer.step()             # 5. update params

    print(f"Epoch {epoch+1}, loss = {loss.item():.4f}")
```

Steps of a DNN Training Workflow (Reference)

1. Import libraries: `torch`, `torch.nn`, `torch.optim`, datasets.
2. Prepare data:
   - load or generate `x` and `y`,
   - convert to PyTorch tensors.
3. Build a dataset class (optional) and wrap with `DataLoader`.
4. Define the neural network as a class inheriting from `nn.Module`.
5. Choose loss function (e.g., MSE) and optimizer (e.g., Adam).
6. Train:
   - loop over epochs and batches,
   - forward $\rightarrow$ loss $\rightarrow$ backward $\rightarrow$ step.
7. Evaluate on validation/test data and visualize results.

Mini-batches and DataLoader (Visual)



DataLoader handles batching, shuffling, and iteration over the dataset.

DataLoader: Mini-batches (Concept)

- `torch.utils.data.DataLoader`:
  - splits data into mini-batches,
  - can shuffle each epoch,
  - can load data in parallel (via `num_workers`).
- Properties:
  - Each epoch: all data points used once (unless `drop_last=True`).
  - `batch_size`: trade-off between speed and stability.
  - `shuffle=True`: helpful for training neural nets.

DataLoader Example

```python
from torch.utils.data import Dataset, DataLoader
import torch

class CustomDataset(Dataset):
    def __init__(self, data, labels):
        self.data   = data
        self.labels = labels

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        return self.data[idx], self.labels[idx]

dataset = CustomDataset(torch.randn(1000, 10),
                        torch.randint(0, 2, (1000,)))
loader = DataLoader(dataset, batch_size=32,
                    shuffle=True)

for batch_x, batch_y in loader:
```

Monitoring Progress with tqdm

```
from tqdm import tqdm

for epoch in range(num_epochs):
    progress = tqdm(loader,
                    desc=f"Epoch {epoch+1}/{num_epochs}")
    for batch_x, batch_y in progress:
        optimizer.zero_grad()
        preds = model(batch_x)
        loss = loss_fn(preds, batch_y)
        loss.backward()
        optimizer.step()
        progress.set_postfix({"loss": loss.item()})
```

tqdm provides a progress bar and live loss updates.

GPU Acceleration (Optional for This Class)

```
device = torch.device("cuda" if torch.cuda.is_available()
                      else "cpu")

model = model.to(device)

for batch_x, batch_y in loader:
    batch_x = batch_x.to(device)
    batch_y = batch_y.to(device)
    # training steps...
```


- For this course, **CPU is fine** for small examples.
- GPU becomes important for:
  - large datasets,
  - big neural networks.

Reference: A Deeper MLP with Dropout

```
import torch.nn as nn

class EqumNN(nn.Module):
    def __init__(self, n1, n2, dropout_prob=0.0):
        super().__init__()
        self.fc1 = nn.Linear(n1, 64)
        self.dropout1 = nn.Dropout(p=dropout_prob)
        self.fc2 = nn.Linear(64, 32)
        self.dropout2 = nn.Dropout(p=dropout_prob)
        self.fc3 = nn.Linear(32, n2)

    def forward(self, x):
        x = nn.functional.relu(self.fc1(x))
        x = self.dropout1(x)
        x = nn.functional.relu(self.fc2(x))
        x = self.dropout2(x)
        x = self.fc3(x)
        return x
```

EqumNN: Mathematical Representation

Let $x \in \mathbb{R}^{n_1}$ be the input and $y_3 \in \mathbb{R}^{n_2}$ the output.

- First layer:

$$y_1 = \mathrm{ReLU}(W_1 x + b_1), \quad W_1 \in \mathbb{R}^{64\times n_1},\ b_1 \in \mathbb{R}^{64}.$$

- Second layer:

$$y_2 = \mathrm{ReLU}(W_2 y_1 + b_2), \quad W_2 \in \mathbb{R}^{32\times 64},\ b_2 \in \mathbb{R}^{32}.$$

- Third layer:

$$y_3 = W_3 y_2 + b_3, \quad W_3 \in \mathbb{R}^{n_2\times 32},\ b_3 \in \mathbb{R}^{n_2}.$$

- Dropout is used during training between layers as a regularization technique (randomly zeroes some components of y_1 and y_2).

In summary:

$$y_3 = W_3\,\mathrm{ReLU}(W_2\,\mathrm{ReLU}(W_1 x + b_1) + b_2) + b_3.$$

Mini Example: Predict GDP Growth with a Tiny MLP

Problem Setup

Goal: Use a small neural network to predict **next-quarter GDP growth** from the **last 4 quarters**.

- Input features at time t:

$$x_t = (\text{growth}_{t-4}, \text{growth}_{t-3}, \text{growth}_{t-2}, \text{growth}_{t-1})$$

- Target:

$$y_t = \text{growth}_t.$$

- Model:
  - small MLP with 1 hidden layer,
  - trained with MSE loss and Adam optimizer.
- This is **not** a sophisticated macro model — just a **simple demo** of the ML **workflow**.

Building Features from Cleaned Data

Assume we already have `df` with a `growth` column (from earlier).

```
import numpy as np
import torch
from torch.utils.data import TensorDataset, DataLoader

# 1. Get growth as NumPy array
g = df['growth'].values.astype(np.float32)

# 2. Build (X, y) where each X has last 4 quarters
X_list = []
y_list = []

for t in range(4, len(g)):
    X_list.append(g[t-4:t])  # last 4 quarters
    y_list.append(g[t])      # next quarter

X = np.stack(X_list)              # shape: [N, 4]
y = np.array(y_list)[:, None]  # shape: [N, 1]

# 3. Convert to tensors
```

Train / Test Split and DataLoader

```
N = X_tensor.shape[0]
train_size = int(0.8 * N)

X_train = X_tensor[:train_size]
y_train = y_tensor[:train_size]
X_test  = X_tensor[train_size:]
y_test  = y_tensor[train_size:]

train_ds = TensorDataset(X_train, y_train)
test_ds  = TensorDataset(X_test, y_test)

train_loader = DataLoader(train_ds, batch_size=32,
                          shuffle=True)
test_loader  = DataLoader(test_ds, batch_size=32,
                          shuffle=False)
```

Defining a Tiny MLP in PyTorch

```python
import torch.nn as nn
import torch.optim as optim

class GrowthMLP(nn.Module):
    def __init__(self):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(4, 16),
            nn.ReLU(),
            nn.Linear(16, 1)
        )

    def forward(self, x):
        return self.net(x)

model = GrowthMLP()
optimizer = optim.Adam(model.parameters(), lr=0.01)
loss_fn = nn.MSELoss()
```

Training Loop

```
num_epochs = 50

for epoch in range(num_epochs):
    model.train()
    epoch_loss = 0.0

    for batch_X, batch_y in train_loader:
        optimizer.zero_grad()
        preds = model(batch_X)
        loss = loss_fn(preds, batch_y)
        loss.backward()
        optimizer.step()

        epoch_loss += loss.item() * batch_X.size(0)

    epoch_loss /= len(train_loader.dataset)
    print(f"Epoch {epoch+1:03d} | Train MSE: {epoch_loss:.6f}")
```

Evaluating and Plotting Predictions

```
model.eval()
with torch.no_grad():
    y_pred_list = []
    y_true_list = []
    for batch_X, batch_y in test_loader:
        preds = model(batch_X)
        y_pred_list.append(preds.numpy())
        y_true_list.append(batch_y.numpy())

y_pred = np.vstack(y_pred_list).flatten()
y_true = np.vstack(y_true_list).flatten()

import matplotlib.pyplot as plt

plt.figure(figsize=(10, 4))
plt.plot(y_true, label="True growth")
plt.plot(y_pred, label="Predicted growth")
plt.title("Next-quarter GDP growth: true vs predicted")
plt.xlabel("Test observation index")
plt.ylabel("Growth")
plt.legend()
```

Exercise: AI-Assisted Dynamic Equilibrium Modeling

Exercise Overview

Goal: Build a reusable framework for solving and extending dynamic equilibrium models using AI tools.

- **Starting point**: Lab11A notebook — Krusell-Smith (1998) solved with Deep Equilibrium Nets (DEQN) in PyTorch.
- **Workflow** (5 parts):
  1. **Document the model** — formal math in LaTeX
  2. **Document the algorithm** — pseudo-code for DEQN
  3. **Structure the code** — refactor notebook into modules
  4. **Create a Claude Code project** — skills, agents, rules
  5. **Iterate with extensions** — add features, validate, repeat
- This is a **semester project**, not a homework. AI tools (Claude Code, Cursor) are expected to help you accomplish it.

The Krusell-Smith (1998) Model — Overview

- **Heterogeneous agents** with incomplete markets and aggregate uncertainty.
- Why it is a benchmark:
  - Simplest model combining **idiosyncratic risk** (employment shocks) with **aggregate risk** (TFP shocks).
  - Requires tracking the **wealth distribution** as a state variable.
  - Classic test case for computational methods in macro.
- The Lab11A notebook solves this model using **Deep Equilibrium Nets** (DEQN) in PyTorch.

Your task
Understand this model deeply enough to **document it formally** and then **extend it**.

The Model — Households

Households maximize lifetime utility:

$$\max\ \mathbb{E}_0 \sum_{t=0}^{\infty} \beta^t\, u(c_t), \quad u(c) = \frac{c^{1-\gamma}}{1-\gamma}$$

subject to:

- **Budget constraint**: $c_t + a_{t+1} = (1 + r_t)\, a_t + w_t\, e_t$
- **Borrowing constraint**: $a_{t+1} \geq \underline{a}$
- **Idiosyncratic productivity**: e_t follows a Markov chain with transition matrix Π_e

In the notebook: $\gamma = 1/\texttt{eis} = 2$, $\beta = 0.98$, $\underline{a} = 0$, and e takes 3 values (Rouwenhorst discretization).

Representative firm with Cobb-Douglas production:

$$Y_t = Z_t\, K_t^{\alpha}\, L^{1-\alpha}$$

Competitive prices:

$$r_t = \alpha\, Z_t \left(\frac{K_t}{L}\right)^{\alpha-1} - \delta, \quad w_t = (1-\alpha)\, Z_t \left(\frac{K_t}{L}\right)^{\alpha}$$

Aggregate TFP shock: Z_t follows a Markov chain (Tauchen, 5 states, $\rho = 0.9$, $\sigma = 0.007$).

Recursive Competitive Equilibrium: A policy function $a'(a, e, Z, \mu)$, a price system (r, w), and a law of motion for μ (the wealth distribution) such that:

1. Households optimize given prices.
2. Firms maximize profits.
3. Markets clear: $K_{t+1} = \int a'(a, e, Z_t, \mu_t)\, d\mu_t$.

Part 1 — Document the Model

Instructions: Write a standalone LaTeX file that formally describes the model.

- **Include**:
  - Model primitives (preferences, technology, shocks)
  - Household problem (Bellman equation or sequential formulation)
  - Firm problem and competitive prices
  - Equilibrium definition (markets, consistency)
- **Tip**: Use AI to translate from the notebook's code comments to formal math. Paste the notebook cells into Cursor and ask: *"Write the formal economic model that this code implements."*
- **Deliverable**: `model_documentation.tex`

Why document first?
Writing the math forces you to understand the model **before** touching the code. This prevents "coding by copy-paste" and makes extensions much easier.

Deep Equilibrium Nets (Azinovic, Gaegauf, Scheidegger, 2022):

- **Key idea**: A neural network approximates the policy function $\sigma(a, e, Z, K) \to$ savings rate.
- **Training**: Minimize Euler equation errors across simulated economies.
- **Three components**:
  1. **Initialization** — use Sequence-Space Jacobian (SSJ) to compute steady state and calibrate grids.
  2. **Policy network** — neural net maps individual + aggregate state to savings decisions.
  3. **Cloud simulation** — N parallel economies evolve jointly; sample agents for Euler errors.

Why DEQN?
It avoids the "curse of dimensionality" that plagues grid-based methods. Adding state variables (new shocks, assets) requires only changing the NN input dimension — no exponential grid explosion.

The DEQN Algorithm — Steps

1. **Calibrate & steady state**: Use SSJ to solve for K_{ss}, grids, and the stationary distribution μ_{ss}.
2. **Initialize cloud**: N copies of μ_{ss}, each with a random TFP draw.
3. **Forward pass**: Neural net maps $(a, e, Z, K) \to$ savings rate σ. Compute savings $a' = \underline{a} + \sigma \cdot (\text{coh} - \underline{a})$.
4. **Euler errors**: Sample random agents, compute:

$$\varepsilon = 1 - \frac{\beta \, \mathbb{E}\left[(1 + r')\, u'(c')\right]}{u'(c)}$$

5. **Backpropagate**: Loss $= \mathbb{E}[\varepsilon^2]$. Update NN weights via Adam.
6. **Transport distribution**: Move $\mu_t \to \mu_{t+1}$ using the policy function (linear interpolation on asset grid $+$ Markov transition for e).
7. **Repeat**: Update $K_{t+1} = \int a \, d\mu_{t+1}$, draw next Z, go to step 3.

Part 2 — Document the Algorithm

Instructions: Add an algorithm section to your `model_documentation.tex`.

- Write each step in **pseudo-code** (not Python — language-agnostic).
- Document:
  - **Inputs**: calibration parameters, grid sizes, NN architecture.
  - **Outputs**: trained policy network, simulated distribution.
  - **Convergence criterion**: loss $< \epsilon$ or max epochs.
- **Map each step to the notebook cell** that implements it:
  - Step 1 (SSJ) $\rightarrow$ Cells 4–5
  - Step 2 (Network) $\rightarrow$ Cell 7
  - Step 3 (Physics) $\rightarrow$ Cell 8
  - Steps 4–7 (Training loop) $\rightarrow$ Cell 10
- **Deliverable**: Updated `model_documentation.tex` with algorithm section.

Part 3 — Structure the Code

The notebook is **monolithic** — all logic in one file. Restructure into modules:

File	Responsibility
`config.json`	All parameters: β, α, δ, grid sizes, NN architecture, training hyperparameters
`model.py`	`KrusellSmithModel` class: prices, utility, distribution transport
`network.py`	`KS_Network` class: neural net architecture
`train.py`	Training loop: cloud simulation, Euler errors, optimization
`dashboard.py`	Live monitoring: loss plots, policy function visualization, K dynamics
`main.py`	Entry point: load config, initialize, train, save results

Tip: Paste notebook cells into Cursor and ask: *"Refactor this into a class in model.py with methods for get_prices, utility_prime, and transport_distribution."*

Code Structure — `config.json` Example

```json
{
  "model": {
    "beta": 0.98,  "gamma": 2.0,
    "alpha": 0.36, "delta": 0.025,
    "rho_e": 0.966, "sd_e": 0.5,
    "rho_tfp": 0.9, "sigma_tfp": 0.007
  },
  "grid": {
    "nA": 100, "nE": 3, "nTFP": 5,
    "amin": 0, "amax": 50
  },
  "network": {
    "n_hidden": 64, "n_layers": 3,
    "activation": "mish"
  },
  "training": {
    "n_epochs": 2000, "batch_size": 256,
    "n_agents_sample": 64,
    "lr": 1e-3, "scheduler_step": 500,
    "scheduler_gamma": 0.5
```

- **Separation of concerns**:
  - Economics logic (`model.py`) is separate from ML logic (`network.py`) and orchestration (`train.py`).
  - You can change the NN architecture without touching the economics, and vice versa.
- **Configuration-driven**:
  - Change parameters in `config.json` without modifying code.
  - Run experiments by swapping config files.
- **Dashboard for monitoring**:
  - Watch the loss curve, policy functions, and K dynamics evolve during training.
  - Catch problems early (divergence, flat loss, unreasonable policies).
- **Testability**: Each module can be tested independently. Does `model.get_prices(K, Z)` return the right values? Does the NN output have the right shape?

Part 4 — Create a Claude Code Project

Goal: Set up a `.claude/` directory so Claude Code can help you extend models systematically.

Based on the architecture from Lecture 3A, create:

- `CLAUDE.md` — project overview, folder structure, workflow rules, current state
- **Skills** (`.claude/skills/`) — slash commands for common operations
- **Agents** (`.claude/agents/`) — specialized reviewers
- **Rules** (`.claude/rules/`) — workflow constraints that are always enforced

Why this matters
Without a `CLAUDE.md`, the AI has no memory of your project structure. With one, it knows *exactly* where your model is, what your code does, and how to extend it safely.

Claude Code Project — Skills to Create

Skill	What it does
`/extend-model`	Add a feature to the model (new shock, preference, constraint). Updates `model_documentation.tex`.
`/extend-algorithm`	Update the DEQN algorithm for the extended model. Adjusts pseudo-code and identifies code changes needed.
`/extend-code`	Modify the Python codebase to implement the extension. Updates `model.py`, `network.py`, etc.
`/run-experiment`	Run training with a specific `config.json`. Save results and logs.
`/validate-model`	Check Euler errors, inspect policy function shapes, verify steady-state convergence.
`/document-extension`	Update `model_documentation.tex` after a code change is validated.

Each skill is a markdown file in `.claude/skills/` that tells Claude Code exactly what steps to follow.

Claude Code Project — Agents and Rules

Agents (specialized reviewers in `.claude/agents/`):

`model-reviewer`	Check economic logic: are the FOCs correct? Is the equilibrium definition consistent?
`code-reviewer`	Check Python/PyTorch: shapes, device placement, gradient flow, numerical stability.
`algorithm-reviewer`	Check numerical methods: convergence, interpolation accuracy, distribution transport.

Rules (always-on constraints in `.claude/rules/`):

`iterative-workflow.md`	Always extend minimally: one feature at a time.
`validation-protocol.md`	Always validate (Euler errors $< 10^{-3}$) before extending further.
`documentation-first.md`	Update `model_documentation.tex` before writing code.

Start: Base KS model — working and validated.

Extension 1: Add labor supply choice (minimal change)

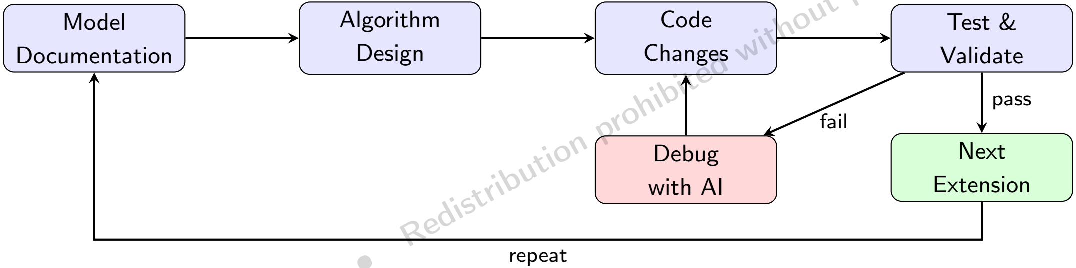
- **Model**: Add labor-leisure tradeoff to household problem.

$$u(c, \ell) = \frac{c^{1-\gamma}}{1-\gamma} - \psi \frac{\ell^{1+\phi}}{1+\phi}$$

- **Algorithm**: Neural net now outputs **two** decisions: (savings rate, labor supply).
- **Code**: Update `network.py` output dimension, add labor FOC to `model.py`.
- **Validate**: Does it converge? Are policy functions reasonable?

Extension 2: Add a second asset (bonds + capital).
Extension 3: Your research model.

The Iteration Cycle



This is the **"Research Architect"** workflow from Lecture 3A. Each cycle adds one feature, validates it, then moves on.

Deliverables

Submit the following:

1. `model_documentation.tex` — formal model description + algorithm pseudo-code.
2. **Structured Python codebase** — 6 files:
   - `config.json`, `model.py`, `network.py`,
   - `train.py`, `dashboard.py`, `main.py`
3. **Claude Code project** — `.claude/` directory with:
   - `CLAUDE.md` (project guide),
   - at least 3 skills,
   - at least 2 agents,
   - at least 2 rules.
4. **At least one model extension** — documented, coded, and validated.
5. **Brief report** (1–2 pages): What worked? What didn't? How did AI help?

Tips for Working with AI

- **Start small**: Get the base model working and validated first. Do not jump to extensions until you understand the base.
- **Be specific**: Instead of *"refactor this code,"* say *"Refactor this cell into a class with methods `get_prices`, `utility_prime`, and `transport_distribution`."*
- **Validate everything**: AI-generated code can have subtle bugs in economic logic. Always check:
  - Do Euler errors converge below 10^{-3}?
  - Are policy functions monotone in wealth?
  - Does aggregate K stay near the steady state?
- **Document as you go**: The `.tex` file IS your thinking. If you cannot write down the math, you do not understand the model well enough to code it.
- **Use the iterative cycle**: Never jump ahead. One extension at a time.

Wrap-Up

What You Should Take Away

- **Tech stack**: OS $\rightarrow$ Python $\rightarrow$ Jupyter/VS Code/Cursor/PyCharm $\rightarrow$ libraries $\rightarrow$ AI assistants.
- **Python basics**: data types, lists, dictionaries, loops, functions, classes (attributes, methods, `self`).
- **Data workflow**: acquire, clean, HP-filter, persist, visualize.
- **PyTorch flavor**:
  - tensors, autograd, models as classes, optimizers, DataLoader.
  - tiny MLP predicting GDP growth from past values.
- Next lecture: **AI and ML theory for economists**:
  - loss functions, overfitting, regularization, evaluation, etc.