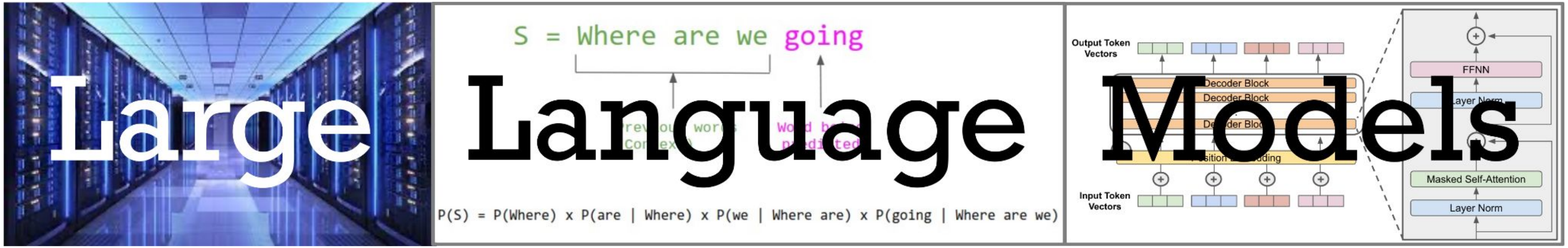


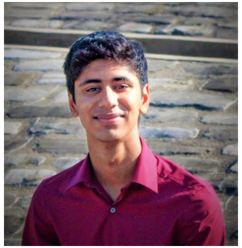
CIS 7000 - Fall 2024

From Pytorch to Hugging Face: How to run your own LLM

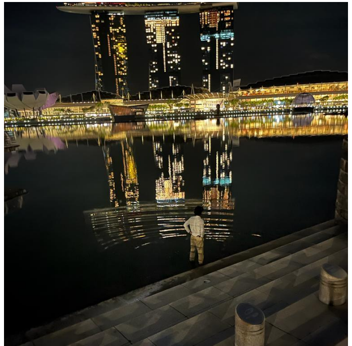
Amish Sethi and Matthew Kuo

A Bit About Amish

- Hi! My name is Amish, and I am your head TA



- I was born in Dallas, Texas and grew up in Pittsburgh, PA
- I am a Junior in SEAS, majoring in CIS and getting an accelerated masters in CIS as well
- I've been doing research with Professor Naik for about a year now, focusing on how to chain LLM optimizations



- My hobbies include reading, chess, traveling, and going out with friends

A Bit About Matthew

- Junior in CIS doing an MSE in CIS
- Born in Cali and moved to Taiwan for middle/high school
- Research with Mayur about building a foundation model
- Hobbies include Valorant, poker, and running

Announcements

- HW0 due yesterday
- HW1 Part 1 released yesterday and due September 15th
- Wednesday lecture will cover Transformer architecture

Today’s Agenda

- PyTorch
    - Tensors
    - Example Neural Network
- Hugging Face

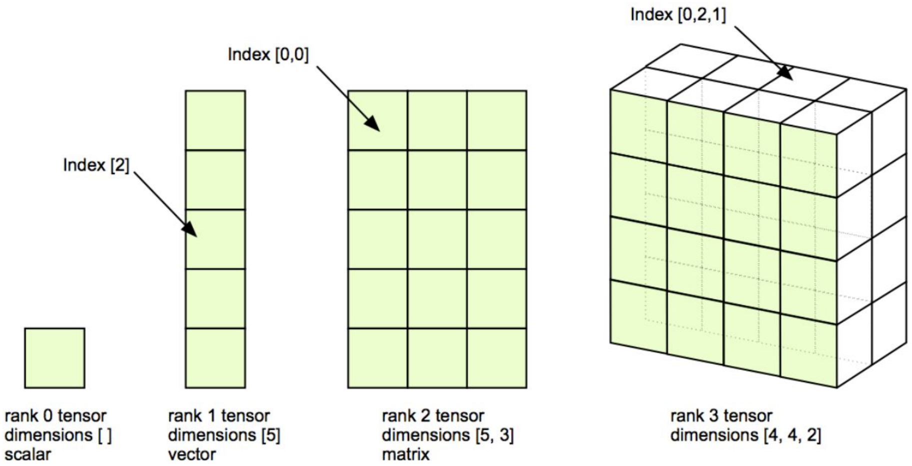
What is PyTorch?

- A **Machine Learning** Framework in Python
- Two main features:
  - N-dimensional Tensor computation (like NumPy) on GPUs
  - Automatic differentiation for training deep neural networks
- Widely used in the machine learning community

Tensors

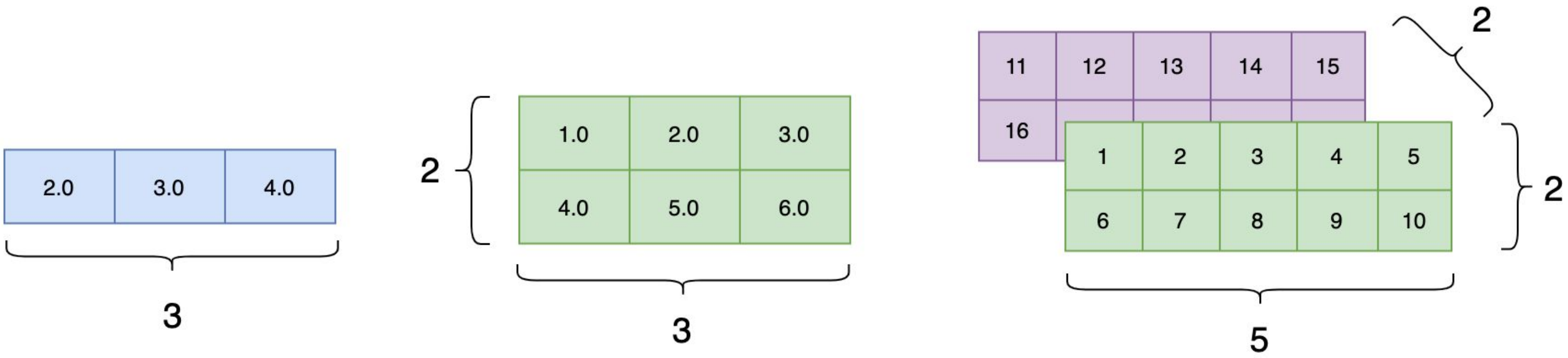
Tensors

- High-dimensional matrices (arrays)



Tensors

Shape of Tensors



1-D Tensor, shape[3]

2-D Tensor, shape[2, 3]

3-D Tensor, shape[2, 2, 5]

Creating Tensors

- Directory transform from python list

  `x = torch.tensor([[1,-1], [-1,1]])`

- Tensor of constant zeros & ones

  `x = torch.zeros(2, 2)`

  `y = torch.ones(2,3)`

  shape

```
tensor([[1., -1.],
        [-1., 1.]])
```

```
tensor([[0., 0.],
        [0., 0.]])
```

```
tensor([[[1., 1., 1., 1., 1.],
         [1., 1., 1., 1., 1.]]])
```

Common Operations

- Addition/Subtraction

`z = x + y`

- Power

`y = x.pow(2)`

- Summation

`y = x.sum()`

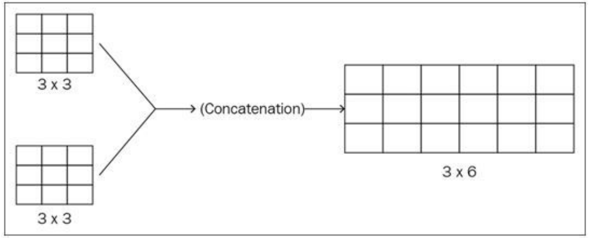
- Mean

`y = x.mean()`

More Common Operations

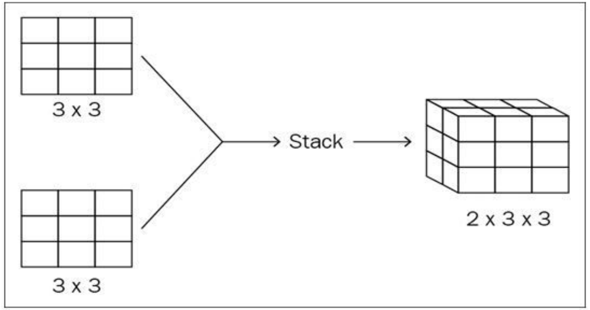
- Concatenate multiple tensors

  `z = torch.cat((x, y), dim=0)`



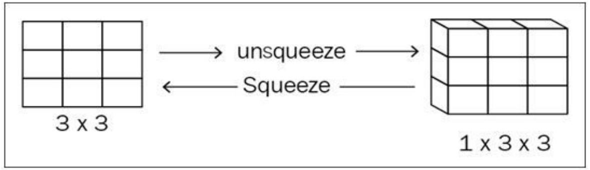
- Stacking multiple tensors

  `z = torch.stack((x, y), dim=0)`



- Squeeze/Unsqueeze

  `y = torch.squeeze(x)`



Transpose

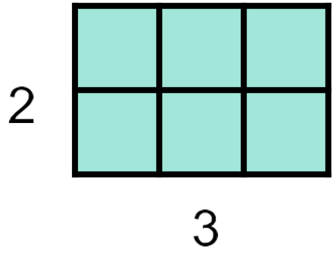
- Transpose the two specified dimensions

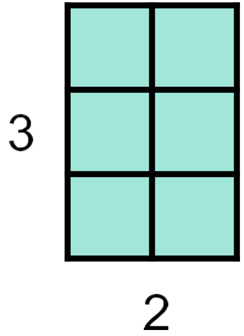
`x = torch.zeros([2,3])`

x.shape ⇒ (2,3)

`y = x.transpose(0, 1)`

x.shape ⇒ (3,2)





Data Types

- **Note**: Using different data types for model and data will cause errors

Data Type	dtype
16-bit floating point	torch.float16
16-bit brain floating point	torch.bfloat16
32-bit floating point	torch.float32
8-bit signed integer	torch.int8

Device of Tensors

- By default, tensors are on the **CPU**
- However, you can change this by using the **.to()** operation
- Changing to _CPU_

```
x = x.to('cpu')
```

- Changing to GPU

```
x = x.to('cuda')
```

Gradient Calculation

```
x = torch.tensor([[1.0, 0.0], [-1.0, 1.0]], requires_grad=True)
```

```
z = x.pow(2).sum()
```

```
z.backward()
```

`x.grad` ⇒ outputs [[2.0, 0.0], [-2.0, 2.0]]

1

$$x = \begin{bmatrix} 1 & 0 \\ -1 & 1 \end{bmatrix}$$

2

$$z = \sum_i \sum_j x_{i,j}^2$$

3

$$\frac{\partial z}{\partial x_{i,j}} = 2x_{i,j}$$

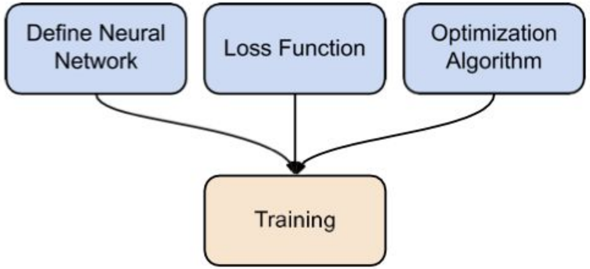
4

$$\frac{\partial z}{\partial x} = \begin{bmatrix} 2 & 0 \\ -2 & 2 \end{bmatrix}$$

Example Neural Network

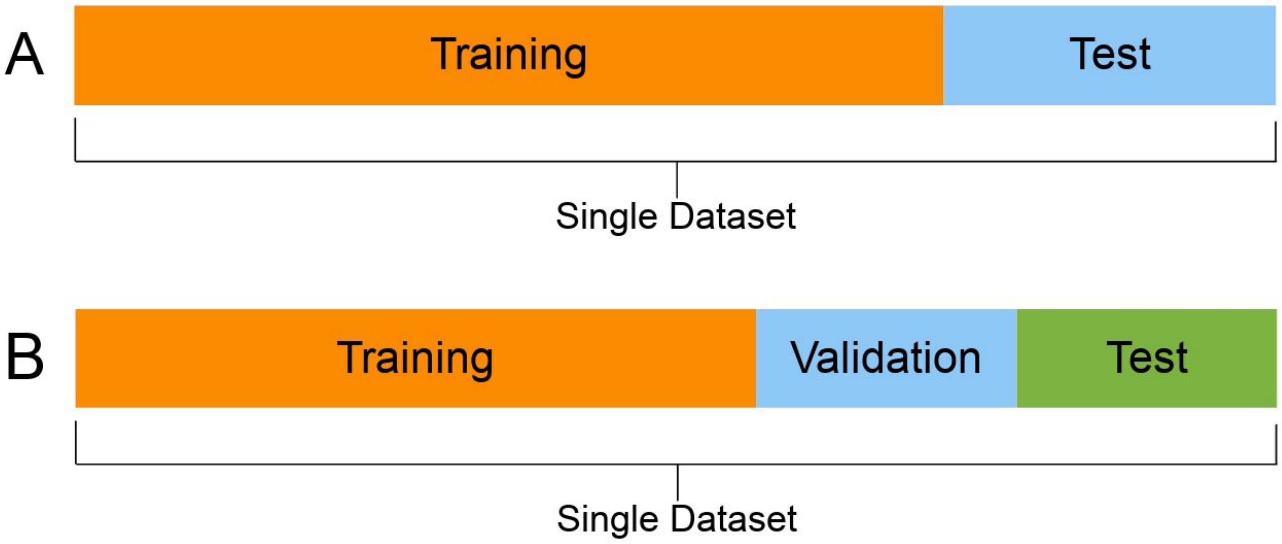
Training Neural Networks

- Main operations during training:
  - Defining the Neural Network (your model)
  - Calculating the loss
  - Optimizing the weights



Training and Testing Neural Networks

- Split the dataset into **training**, **validation**, and **testing**
  - The ratio can be anything but most of the time it is a 7:2:1 split



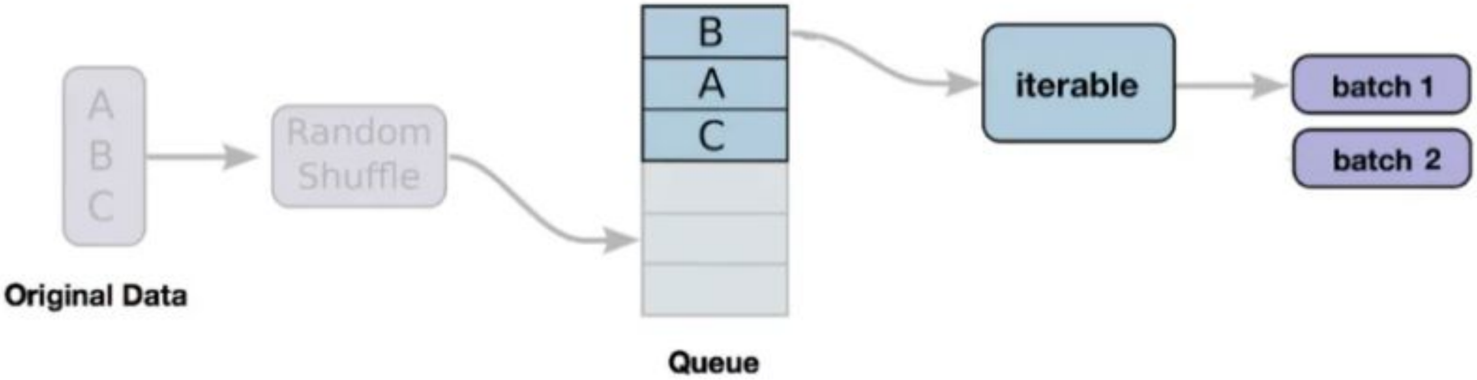
Creating a Dataset

```python
class SimpleDataset(Dataset):
    ## Reading the data (including labels) and preprocessing them
    def __init__(self, features, labels):
        self.features = torch.tensor(features, dtype=torch.float32)
        self.labels = torch.tensor(labels, dtype=torch.long)

    ## Returns the length of the dataset
    def __len__(self):
        return len(self.features)

    ## Returns one sample at a time
    def __getitem__(self, idx):
        feature = self.features[idx]
        label = self.labels[idx]
        return feature, label
```

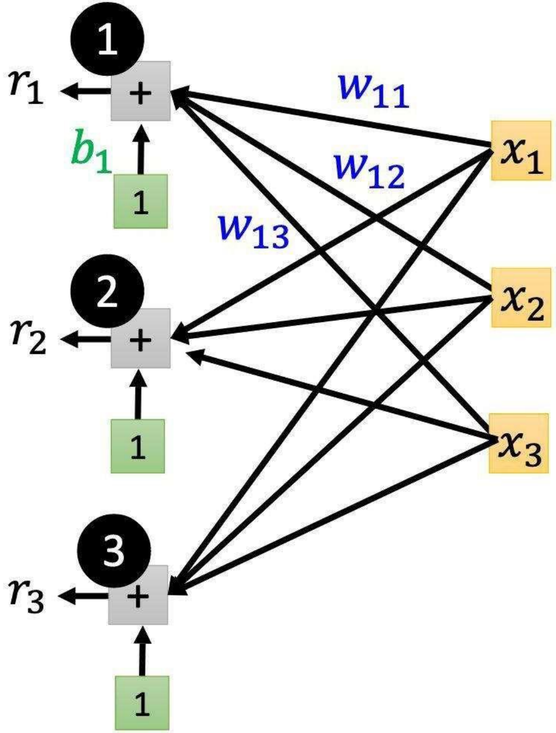
Dataloader

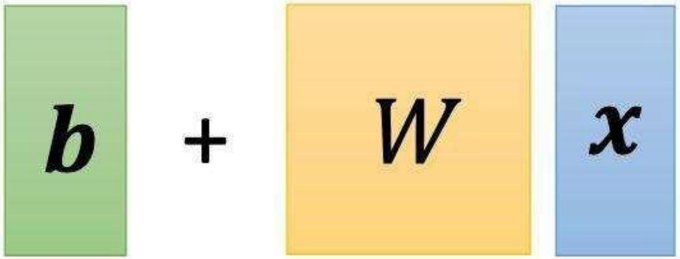
```
dataloader = Dataloader(dataset, batch_size=6, shuffle=True)
```

Training: True
Testing: False



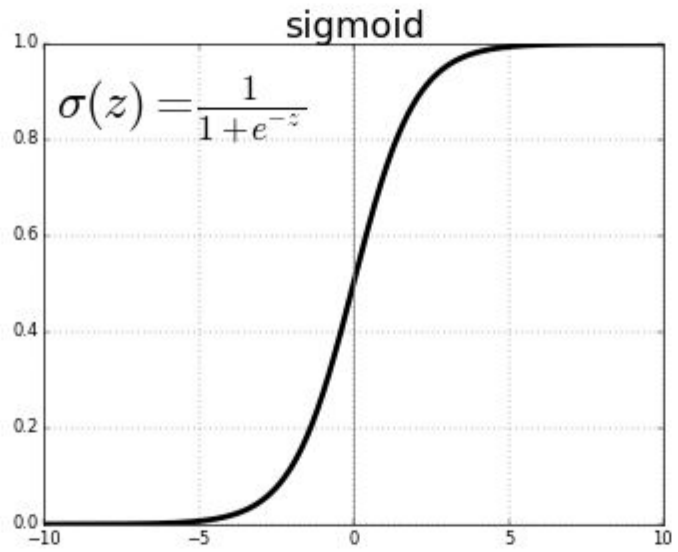
Neural Networks



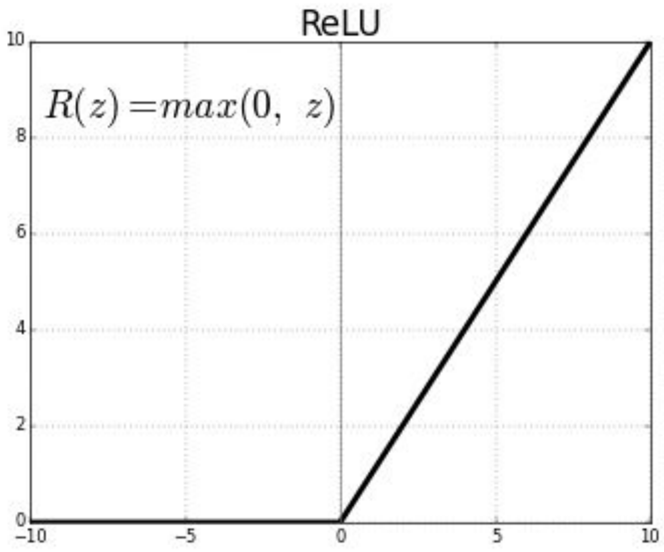


Non-Linear Activation Functions

nn.Sigmoid



nn.ReLU



Building Your Own Neural Network

```python
class SimpleNN(nn.Module):
    ## Initialize the models and define the layers
    def __init__(self):
        super(SimpleNN, self).__init__()
        self.fc1 = nn.Linear(2, 10)
        self.fc2 = nn.Linear(10, 2)

    ## Compute the output of the NN
    def forward(self, x):
        x = torch.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

Loss Functions

`nn.MSELoss`

- Mean Squared Error
- Mostly for **regression tasks**

`nn.CrossEntropyLoss`

- Cross Entropy
- Mostly for **classification tasks**

Mean

Error

Squared

$$\mathrm{MSE} = \boxed{\frac{1}{n}\sum_{i=1}^{n}}\left(\boxed{Y_i - \hat{Y}_i}\right)^{\boxed{2}}$$

$$H = -\sum p(x) \log p(x)$$

Optimizers

- Gradient-based algorithms that adjusts the network parameters to reduce the errors
- Ex. Stochastic Gradient Descent (SGD)

  `torch.optim.SGD(model.parameters(), lr)`

- For every batch of data:
  - Call `optimizer.zero_grad()` to reset the gradient
  - Call `loss.backward()` to run the backward pass
  - Call `optimizer.step()` to adjust the parameters

Hugging Face

What is Hugging Face

- Hugging Face is a leading platform for natural language processing (NLP) and AI.
- It provides open-source tools, libraries, and pre-trained models for NLP, machine learning, and AI applications.
- Popular for the *Transformers* library, which enables easy access to state-of-the-art models like BERT, GPT, and T5.

Hugging Face

Datasets in Hugging Face

- Hugging Face provides access to a vast collection of datasets for NLP tasks through the datasets library.
- Easily load and explore datasets for tasks like text classification, sentiment analysis, translation, and more.
- Supports custom datasets, allowing users to prepare data for model training and evaluation.
- Key features:
  - Access datasets via load_dataset() function.
  - Datasets are optimized for both speed and scalability.
  - Includes built-in dataset versioning and caching

Tokenizers

- Tokenizers convert raw text into a format that models can understand.
- Hugging Face provides an efficient and customizable tokenizers library to handle tokenization.
- Key features:
  - Supports different tokenization techniques like Byte-Pair Encoding (BPE), WordPiece, and SentencePiece.
  - Tokenization happens quickly with parallelization support.
  - Handles special tokens like [CLS], [SEP], and padding/truncation automatically.
  - Easily load pre-trained tokenizers with AutoTokenizer.

```
from transformers import AutoTokenizer
tokenizer = AutoTokenizer.from_pretrained('bert-base-uncased')
tokens = tokenizer("Hello, Hugging Face!")
```

Loading Pre-trained models

- Hugging Face makes it easy to load and use pre-trained models for various tasks like text classification, translation, and text generation.
- Transformers library provides access to state-of-the-art models like BERT, GPT, T5, and more.
- Steps to load a model:
  - Use AutoModel or task-specific classes like AutoModelForSequenceClassification.
  - Download and load pre-trained models with one line of code.
  - Fine-tune models for specific tasks or use them for inference directly.

```python
from transformers import AutoModelForCausalLM, AutoTokenizer

tokenizer = AutoTokenizer.from_pretrained("gpt2")
model = AutoModelForCausalLM.from_pretrained("gpt2")

inputs = tokenizer("Hello, Hugging Face!", return_tensors="pt")
outputs = model.generate(inputs["input_ids"], max_length=50)
print(tokenizer.decode(outputs[0], skip_special_tokens=True))
```

Trainer

- Hugging Face makes it easy to fine-tune pre-trained models on your custom datasets.
- Use Trainer class to handle training loops, evaluation, and optimization automatically.
- Define training arguments and train with the Trainer class.

```python
# Define training arguments
training_args = TrainingArguments(
    output_dir="./results",
    evaluation_strategy="epoch",
    per_device_train_batch_size=16,
    num_train_epochs=3,
    logging_dir="./logs",
)

# Initialize Trainer
trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=train_dataset,
    eval_dataset=eval_dataset,
)

# Train the model
trainer.train()
```