Introduction
An Era of Deep Learning

Fig. 1. Applications of deep learning

(a) Recommendation
(b) Object Recognition
(c) Medical image classification
(d) Self-driving cars
(e) Playing Go games
(f) Medical image classification

Jaewoo Lee
ImageNet Challenge

ILSVRC top-5 error on ImageNet
What is deep learning?

(a) Traditional Learning Models

(b) Deep Learning Models
Review: classical models

- Logistic regression

\[ y = \phi(w^T x + w_0) = \phi\left(\sum_{i=1}^{p} w_i x_i + w_0\right), \]

where

- \( \mathbf{x} = (x_1, x_2, \ldots, x_p) \) feature vector
- \( \mathbf{w} = (w_1, w_2, \ldots, w_p) \) weight vector (or parameter vector)
- \( w_0 \) bias (or intercept) term
Fig. 3. Multiple layers of information processing
1 Dataset: an i.i.d. sample drawn from the population
   - \( D = \{ (x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n) \} \)

2 Task: predict \( Y \) given \( X \)
   - regression or classification?

3 Model: assumption on the (distribution) of data
   - linear, non-linear, parametric, non-parametric

4 Loss (a.k.a. score/cost): measure of how good a model is
   - \( \ell : Y \times Y \rightarrow \mathbb{R}_+ \)

5 Training: normally involve an optimization
   - (stochastic) gradient descent, Newton’s method

6 Evaluation: validate the learning outcome
MLP
Multilayer Perceptrons

- Multiple *layers* of units
- Connecting units in one layer to those in the subsequent layer
- Feed-forward neural network
- No cycle
- All input units are connected to all output units (fully connected layer).

Fig. 4. Simple MLP
MLP

Use machine learning algorithms as a black box!

- **Pre-activation**
  - \( z_1 = w_{11}x_1 + w_{12}x_2 \)
  - \( z_2 = w_{21}x_1 + w_{22}x_2 \)
  - \( z_3 = w_{31}x_1 + w_{32}x_2 \)

- **Matrix notation**

\[
\begin{pmatrix}
  z_1 \\
  z_2 \\
  z_3
\end{pmatrix} = 
\begin{pmatrix}
  w_{11} & w_{12} \\
  w_{21} & w_{22} \\
  w_{31} & w_{33}
\end{pmatrix} 
\begin{pmatrix}
  x_1 \\
  x_2
\end{pmatrix} + 
\begin{pmatrix}
  b_1 \\
  b_2 \\
  b_3
\end{pmatrix}
\]

- \( h = \phi(Wx + b) \)
MLP

• $h_i = \phi(w_i^T x + b)$
• Matrix notation

$z = Wx + b$

$h = (h_1, h_2, h_3) = \phi(z)$,

where $W \in \mathbb{R}^{m \times n}$.

▶ $W$: weight matrix
▶ $b$: bias term
▶ $z$: pre-activation
▶ $\phi$: activation function
What if we have two hidden layers?

- \( h_i = \phi(w_i^T x + b) \)
- Matrix notation
  \[
  h^{(1)} = \phi(W^{(1)} x + b^{(1)}) \\
  h^{(2)} = \phi(W^{(2)} h^{(1)} + b^{(2)})
  \]
- Hierarchy
  \[
  h^{(2)} = \phi(W^{(2)} \phi(W^{(1)} x + b^{(1)}) + b^{(2)})
  \]
Activation functions

- Determines what (or whether) to pass on from each neuron to the next layer
- Creates **Non-linearity** in the network

<table>
<thead>
<tr>
<th>Activation function</th>
<th>Equation</th>
<th>Example</th>
<th>1D Graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unit step (Heaviside)</td>
<td>$\phi(z) = \begin{cases} 0, &amp; z &lt; 0, \ 0.5, &amp; z = 0, \ 1, &amp; z &gt; 0, \end{cases}$</td>
<td>Perceptron variant</td>
<td><img src="image1" alt="Graph" /></td>
</tr>
<tr>
<td>Sign (Signum)</td>
<td>$\phi(z) = \begin{cases} -1, &amp; z &lt; 0, \ 0, &amp; z = 0, \ 1, &amp; z &gt; 0, \end{cases}$</td>
<td>Perceptron variant</td>
<td><img src="image2" alt="Graph" /></td>
</tr>
<tr>
<td>Linear</td>
<td>$\phi(z) = z$</td>
<td>Adaline, linear regression</td>
<td><img src="image3" alt="Graph" /></td>
</tr>
<tr>
<td>Piece-wise linear</td>
<td>$\phi(z) = \begin{cases} 1, &amp; z \geq \frac{1}{2}, \ z + \frac{1}{2}, &amp; -\frac{1}{2} &lt; z &lt; \frac{1}{2}, \ 0, &amp; z \leq -\frac{1}{2}, \end{cases}$</td>
<td>Support vector machine</td>
<td><img src="image4" alt="Graph" /></td>
</tr>
<tr>
<td>Logistic (sigmoid)</td>
<td>$\phi(z) = \frac{1}{1 + e^{-z}}$</td>
<td>Logistic regression, Multi-layer NN</td>
<td><img src="image5" alt="Graph" /></td>
</tr>
<tr>
<td>Hyperbolic tangent</td>
<td>$\phi(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$</td>
<td>Multi-layer NN</td>
<td><img src="image6" alt="Graph" /></td>
</tr>
</tbody>
</table>
Is deep learning difficult?

- Deep models:
  \[ y = \phi \left( W^{(2)} \phi \left( W^{(1)} x \right) \right) \]

- Classical Models:
  \[ y = \phi(Wx) \]

- Loss: \( L(W) = \frac{1}{2} \| \hat{Y}(W, X) - Y \|^2_F \)

- Simplistic case: \( \phi(x) = x. \)
  - \( W^{(1)} = w_1, W^{(2)} = w_2, X = Y = 1 \)

\[ L_{\text{deep}}(W) = (w_1 w_2 - 1)^2 \quad \text{VS} \quad L_{\text{lin}}(W) = (w_1 - 1)^2 + (w_2 - 1)^2 \]
The depth of NN creates non-convex loss surfaces.
Model

- What functions can be expressed by my model?
  - A NN can be viewed as a function (or a mapping).
  - $f_\theta : \mathbb{R}^2 \rightarrow \mathbb{R}^2$
  - $f_\theta : x \mapsto y$

- Model parameter
  - $\theta = \{W^{(1)}, b^{(1)}, W^{(2)}, b^{(2)}, W^{(3)}, b^{(3)}\}$
Pipeline

- We want to learn a map $f : \mathcal{X} \rightarrow \mathcal{Y}$.

$$X \xrightarrow{f_\theta} Y$$

- input $\{(x_i, y_i)\}$, $y_i = f$ evaluated at $x_i$
- output $\{\hat{y}_i = f_\theta(x_i)\}$
- parameter $\theta$: weight vectors

- Training: optimize $\theta$ to approximate the mapping $f$
Cost Functions

What to optimize?

- Define a distribution over a variable to predict \( p(y|x; \theta) \)
- Maximum Likelihood Principle

\[
L(\theta) = \mathbb{E}_{x,y \sim p_{\text{data}}} [\log p_{\text{model}}(y | x)]
\]

- Example: when \( p_{\text{model}}(y | x) = \mathcal{N}(y; f(x; \theta), I) \), MLE reduces to

\[
L(\theta) = \frac{1}{2} \mathbb{E}_{x,y \sim p_{\text{data}}} [\|y - f(x; \theta)\|^2].
\]
Forward Phase

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>0.5</td>
<td>1</td>
</tr>
<tr>
<td>0.2</td>
<td>0.7</td>
<td>0</td>
</tr>
<tr>
<td>0.9</td>
<td>0.1</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 1. Input data

- Assume $\phi(z) = \max(0, z)$ (ReLU activation function)

$$z^{(1)}_1 = 0.2 \cdot 0.1 + 0.6 \cdot 0.5 = 0.32$$
$$z^{(1)}_2 = 0.3 \cdot 0.1 + 0.5 \cdot 0.5 = 0.28$$
$$z^{(1)}_3 = 0.9 \cdot 0.1 + 0.7 \cdot 0.5 = 0.44$$
Forward Phase

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$\hat{y}$</th>
<th>$y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>0.5</td>
<td>0.8</td>
<td>1</td>
</tr>
<tr>
<td>0.2</td>
<td>0.7</td>
<td>0.3</td>
<td>0</td>
</tr>
<tr>
<td>0.9</td>
<td>0.1</td>
<td>0.1</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 2. After the forward step

- For each $x \in D$, compute the loss
- Average the losses

$$L(\theta) = \frac{1}{3} \left\{ (0.8 - 1)^2 + (0.3 - 0)^2 + (0.1 - 0)^2 \right\}$$
• Backpropagating *error correction* info.
  - How should we modify $\theta_l = \{w^{(l)}, b^{(l)}\}$?
  - It is given by the gradient: $\frac{\partial L(\theta)}{\partial \theta_l}$.
  - Let’s start with $\frac{\partial L(\theta)}{\partial \theta_3}$.

$$ \hat{y} = (w^{(3)})^T h^{(2)} = w_1^{(3)} h_1^{(2)} + w_2^{(3)} h_2^{(2)} + w_3^{(3)} h_3^{(2)} $$
Backward Phase: $\theta_3$

Recall that $L(\theta) = \frac{1}{2}(y - \hat{y})^2$.

\[ \hat{y} = (w^{(3)})^\top h^{(2)} = w_1^{(3)} h_1^{(2)} + w_2^{(3)} h_2^{(2)} + w_3^{(3)} h_3^{(2)} \]

- By the chain rule, we have

\[ \frac{\partial L}{\partial \theta_3} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial \theta_3} \]
Backward Phase: $\theta_2$

- How about $\frac{\partial L}{\partial \theta_2}$?
  - Again, by the chain rule, we have
    \[
    \frac{\partial L}{\partial \theta_3} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial \theta_3}
    \]
    \[
    \frac{\partial L}{\partial \theta_2} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial h^{(2)}} \frac{\partial h^{(2)}}{\partial z_2} \frac{\partial z_2}{\partial \theta_2}.
    \]

From the forward phase, we have
\[
\hat{y} = (w^{(3)})^\top h^{(2)}
\]
\[
h^{(2)} = \phi(z^{(2)})
\]
\[
z^{(2)} = W^{(2)} h^{(1)}.
\]
Backward Phase: $\theta_1$

- Now we compute $\frac{\partial L}{\partial \theta_1}$.
  - By the chain rule, we have
    
    $$
    \frac{\partial L}{\partial \theta_3} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial \theta_3}
    $$
    $$
    \frac{\partial L}{\partial \theta_2} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial h^{(2)}} \frac{\partial h^{(2)}}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial \theta_2}
    $$
    $$
    \frac{\partial L}{\partial \theta_1} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial z^{(1)}} \frac{\partial z^{(1)}}{\partial \theta_1}
    $$
    $$
    = \frac{\partial L}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial z^{(1)}} \frac{\partial z^{(1)}}{\partial \theta_1}
    $$
    
    $\hat{y}$
    $h_1^{(2)}$
    $h_2^{(2)}$
    $h_3^{(2)}$
    $h_1^{(1)}$
    $h_2^{(1)}$
    $h_3^{(1)}$
    $x_1$
    $x_2$
    $L(\theta)$
    $w_1^{(3)}$
    $w_2^{(3)}$
    $w_3^{(3)}$
CNN
MLPs are *universal* function approximators. They can learn a complex mapping $f : \mathcal{X} \rightarrow \mathcal{Y}$ between $X$ and $Y$.

Given input feature vector $x$, $f_\theta$ returns desired output.

- $f_\theta : x \mapsto \hat{y}$
- **fully connected**

Forward step: matrix multiplication

- $h^{(1)} = \phi(W_1^T x)$
- $h^{(2)} = \phi(W_2^T h^{(1)})$
MLP can be too complex!

Fig. 6. A fully connected network with 2 hidden layers

- How *many* parameters to learn?
  - If we have *n* units in each layer, what is the total number of parameters?
  - High complexity may lead to *overfitting* and local optima
  - Requires careful *initialization*
History

Hubel and Wiesel 1959

- measured neural responses in cat’s striate cortex
- suggested a hierarchy of feature detectors in the visual cortex
- higher level features responding to patterns of activations in lower level cells
Computer Vision

- Computer vision needs some robustness:
  - translation invariance

- need a network that recognizes the cat *regardless of its location*
- location might not be an important factor
MLP

\[ \hat{y} = f_{\theta}(x) \]

- How many parameters do we need for the first layer?
  - Parameters: \(280 \times 280 \times 1000 = 78.4M\)

- What if the input is slightly shifted?
A unit’s activation is dependent on the outputs of all nodes in the previous layer.
MLP

- Each unit in the hidden layer is affected by the *entire* inputs.
- How can we detect the same feature at different positions?
- expect low-level features to be *local*
- expect high-level features to be *coarser*
Let’s have a feature detector

An interesting pattern can be anywhere in the image.

- Define a kernel (or filter) $W$ that search for a specific pattern.
- The detector slides $W$ over the given image.
- Inspect $\langle X[\cdot, \cdot], W \rangle$ and determine if the region contains the pattern

Let’s have multiple of them!
Convolution

What is special about $\langle X[\cdot, \cdot], W \rangle$ operation?

- Suppose we have a 1D image $f = (f_1, f_2, \ldots, f_n)$ and a 1D kernel $g = (g_1, \ldots, g_m)$.
- $\langle X[\cdot, \cdot], W \rangle$ in 1D corresponds to

$$h[i] = (f * g)[i] = \sum_{j=1}^{m} g[j] \cdot f[i - j + m/2].$$

\[
\begin{array}{c}
\begin{array}{cccccc}
| & | & | & | & | & | \\
9 & 0 & 2 & 1 & 0 & 9 \\
\end{array} \\
\begin{array}{ccc}
| & | & | \\
1/3 & 1/3 & 1/3 \\
\end{array} \\
\begin{array}{cccccc}
| & | & | & | & | & | \\
\end{array}
\end{array}
\]
1D Convolution

\[ f = \begin{array}{cccccc}
0 & 9 & 0 & 2 & 1 & 0 & 9
\end{array} \]
\[ g = \begin{array}{c}
1/3 \quad 1/3 \quad 1/3
\end{array} \]

\[ h = \begin{array}{ccccccc}
\end{array} \]
### 1D Convolution

**$f = \begin{array}{c} 0 \\ \times \\ \times \\ \times \end{array} \begin{array}{cccccc} 9 & 0 & 2 & 1 & 0 & 9 \end{array}**

**$g = \begin{array}{cccc} 1/3 & 1/3 & 1/3 \end{array}**

**$h = \begin{array}{cccccccc} 3 & 3.66 & ? & ? & ? & ? \end{array}$**

Jaewoo Lee
1D Convolution

\[ f = \begin{bmatrix} 0 & 9 & 0 & 2 & 1 & 0 & 9 \end{bmatrix} \times \begin{bmatrix} g \end{bmatrix} = \begin{bmatrix} 1/3 & 1/3 & 1/3 \end{bmatrix} \]

\[ h = \begin{bmatrix} 3 & 3.66 & 1 & ? & ? & ? \end{bmatrix} \]

- This can be easily extended to 2D.
- But, is \( h \) useful?
Image Filters

(a) Edge detection

(b) Sharpen

(c) Blur

Fig. 7. Kernel (Image source: Wikipedia)
But how do we know which filter to apply?

\[ W = \begin{bmatrix}
  w_{11} & w_{12} & w_{13} \\
  w_{21} & w_{22} & w_{23} \\
  w_{31} & w_{32} & w_{33}
\end{bmatrix} \]

- Let our NN *learn* it from the data.
- \( W \) are *shared* across different locations.
Convolution

\( h_1 = w_1 x_1 + w_2 x_2 + w_3 x_3 \)
Convolution

\[ h_2 = w_1 x_2 + w_2 x_3 + w_3 x_4 \]
Convolution

\[ h_3 = w_1 x_3 + w_2 x_4 + w_3 x_5 \]
Convolution

\[ h_4 = w_1 x_4 + w_2 x_5 + w_3 x_6 \]
Convolution

\[ h_5 = w_1 x_7 + w_2 x_8 + w_3 x_9 \]
Convolution

Image

Convolved Feature

\[
\begin{array}{cccc}
1_{x1} & 1_{x0} & 1_{x1} & 0 \ 0 \\
0_{x0} & 1_{x1} & 1_{x0} & 1 \ 0 \\
0_{x1} & 0_{x0} & 1_{x1} & 1 \ 1 \\
0 \ 0 \ 1 \ 1 \ 1 \ 0 \\
0 \ 1 \ 1 \ 0 \ 0 \ 0
\end{array}
\]
Convolution

Image

Convolved Feature

1 1 1 1 0 0
0 1 1 1 1 0
0 0 1 1 1 1
0 0 1 1 1 0
0 0 1 1 0 0

4 3
Convolution

Image

Convolved Feature
Convolution

Image

Convolved Feature
Convolution

Image

Convolved Feature
Convolution

Image

Convolved Feature
Convolution

Image

\[
\begin{array}{cccc}
1 & 1 & 1 & 0 \\
0 & 1 & 1 & 1 \\
0 & 0 & 1 & 1 \\
0 & 0 & 1 & 0 \\
0 & 1 & 1 & 0 \\
\end{array}
\]

Convolved Feature

\[
\begin{array}{ccc}
4 & 3 & 4 \\
2 & 4 & 3 \\
2 & 3 & 4 \\
\end{array}
\]
What does a Convolution do?

Fig. 8. Image credit: I. Kokkinos
Output Size

In our example, we have

- a $5 \times 5$ image,
- a Kernel (or filter) of size $3 \times 3$, and
- Stride of 1.

This gives an output of size

$$\frac{N - K}{S} + 1.$$ 

Suppose we have

- $7 \times 7$ image
- $3 \times 3$ kernel
- stride of 3

What is the output size?
Zero Padding

- image size = $8 \times 8$
- kernel = $3 \times 3$ with stride of 1
- What is the size of output? \[ \frac{10 - 3}{1} + 1 = 8 \]
In General,

Fig. 9. Image source: Andrej Karpathy
Pooling Layer
Pooling Layer

The diagram illustrates a pooling layer operation on a 4x4 grid. The red region represents the pooling area, and the numbers outside the grid indicate the pooling parameters. Each pooling operation aggregates the values within the specified region to produce a single value.
Pooling Layer
Pooling Layer
Pooling Layer

- What does the pooling layer do?
  - result in a smaller representation (shrunk image)
  - target feature values becomes less sensitive to small change in the image
  - stride > 1 is called “down sampling”

- Max pooling
- Average pooling
- $L_p$ pooling ($p = 1 \Rightarrow$ average, $p = \infty \Rightarrow$ max pooling)
• The input to the next layer is *down sampled* image.
• Higher-layer filters see a larger region of the input than the equal-sized filters in the lower layers.
Fig. 11. Image source: Honglak Lee et al. 2011
What is PyTorch?

- Python implementation of deep learning tools
  - A collection of classes
  - https://pytorch.org
- Developed by Meta
1 Dataset: an *i.i.d.* sample drawn from the population
   - \( D = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\} \)

2 Task: predict \( Y \) given \( X \)
   - regression or classification?

3 Model: assumption on the (distribution) of data
   - linear, non-linear, parametric, non-parametric

4 Loss (a.k.a. score/cost): measure of how good a model is
   - \( \ell: \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}_+ \)

5 Training: normally involve an optimization
   - (stochastic) gradient descent, Newton’s method

6 Evaluation: validate the learning outcome
Building a Neural Network

- **Input:** MNIST dataset
  - 28 × 28 gray scale image
  - vectorize the image to \( x \in \mathbb{R}^{764} \)

- **Architecture**
  - 1st hidden layer: 512 units
  - 2nd hidden layer: 512 units
  - Output layer: 10 units (10 classes)
import torch.nn as nn

# Get cpu or gpu device for training.
device = "cuda" if torch.cuda.is_available() else "cpu"
print(f"Using {device} device")

# Define model
class NeuralNetwork(nn.Module):
    def __init__(self):
        super(NeuralNetwork, self).__init__()
        self.flatten = nn.Flatten()
        self.linear_relu_stack = nn.Sequential(
            nn.Linear(28*28, 512),
            nn.ReLU(),
            nn.Linear(512, 512),
            nn.ReLU(),
            nn.Linear(512, 10)
        )

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

model = NeuralNetwork().to(device)
print(model)
Building a Neural Network (2)

```python
import torch.nn.functional as F

# Define model

class NeuralNetwork(nn.Module):
    def __init__(self):
        super(NeuralNetwork, self).__init__()
        self.flatten = nn.Flatten()
        self.linear1 = nn.Linear(28*28, 512)
        self.linear2 = nn.Linear(512, 512)
        self.classifier = nn.Linear(512, 10)

    def forward(self, x):
        x = self.flatten(x)
        out = F.relu(self.linear1(x))
        out = F.relu(self.linear2(out))
        logits = self.classifier(out)
        return logits

model = NeuralNetwork().to(device)
print(model)
```
Loading Datasets

Defining Datasets

```python
import torch
from torch import nn
from torch.utils.data import DataLoader
from torchvision import datasets
from torchvision.transforms import ToTensor

# Download training data from open datasets.
training_data = datasets.FashionMNIST(
    root="data",  # path to the directory containing the dataset
    train=True,  # If True, downloads the dataset from the internet
    download=True,
    transform=ToTensor(),
)

# Download test data from open datasets.
test_data = datasets.FashionMNIST(
    root="data",
    train=False,
    download=True,
    transform=ToTensor(),
)
```
batch_size = 64

# Create data loaders.
train_dataloader = DataLoader(training_data, batch_size=batch_size)
test_dataloader = DataLoader(test_data, batch_size=batch_size)

for X, y in test_dataloader:
    print(f"Shape of X [N, C, H, W]: {X.shape}"")
    print(f"Shape of y: {y.shape} {y.dtype}"")
    break
Training NNs

```python
1. loss_fn = nn.CrossEntropyLoss()
2. optimizer = torch.optim.SGD(model.parameters(), lr=1e-3)
```

- `nn.MSELoss`
- `nn.NLLLoss`
- `nn.L1Loss`
- ...

See [here](#) for other available functions.
def train(dataloader, model, loss_fn, optimizer):
    size = len(dataloader.dataset)
    model.train()

    for batch, (X, y) in enumerate(dataloader):
        X, y = X.to(device), y.to(device)

        # Compute prediction error
        pred = model(X)
        loss = loss_fn(pred, y)

        # Backpropagation
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        if batch % 100 == 0:
            loss, current = loss.item(), batch * len(X)
            print(f"loss: {loss:>7f}  [{current:>5d}/{size:>5d}]"
def test(dataloader, model, loss_fn):
    size = len(dataloader.dataset)
    num_batches = len(dataloader)
    model.eval()
    test_loss, correct = 0, 0
    with torch.no_grad():
        for X, y in dataloader:
            X, y = X.to(device), y.to(device)
            pred = model(X)
            test_loss += loss_fn(pred, y).item()
            correct += (pred.argmax(1) == y).type(torch.float).sum().item()
    test_loss /= num_batches
    correct /= size
    print(f"Test Error: 
      Accuracy: {(100 * correct):>0.1f}%,
      Avg loss: {test_loss:>8f}
")
Training Loop

ePOCHS = 5

for t in range(epochs):
    print(f"Epoch {t+1}
           -------------------")
    train(train_dataloader, model, loss_fn, optimizer)
    test(test_dataloader, model, loss_fn)

print("Done!")