IV. Preliminary Knowledge: Python and PyTorch Implementations

Young-geun Kim

Department of Statistics and Probability

STT 997 (SS 2025)

MICHIGAN STATE UNIVERSITY

Introduction

- This lecture aims to leverage Python and PyTorch implementations to (i) deepen understanding of preliminary knowledge about Statistics and Statistical Learning, and (ii) empirically validate key theoretical results.
- This lecture omits some basics of Python. For additional resources on the basics of Python, see `https://cs231n.github.io/python-numpy-tutorial/#scipy`.
- Most of the code drafts in this lecture were generated by ChatGPT 4, a generative AI.

Outline

1. Law of Large Numbers and Central Limit Theorem

2. Logistic Regression

3. Neural Network

Recapping Law of Large Numbers

- *(Weak) Law of Large Numbers:* Let $\bar{X}$ be the mean of n i.i.d. samples from a distribution with a finite first moment μ. Then, $\bar{X} \xrightarrow{p} \mu$. That is, for any $\epsilon > 0$, $\mathbb{P}(|\bar{X} - \mu| > \epsilon) \to 0$.
- *Experiment:* We will examine whether $\mathbb{P}(|\bar{X} - \mu| > \epsilon)$ converges to zero:
  1. Sample a dataset of size $n = 1000$, denoted as $(x_i)_{i=1}^{n}$, $M = 50$ times to draw a spaghetti plot having M trajectories of $\bar{X}$ (x-axis: n; y-axis: $\bar{X}$).
  2. Sample a dataset of size $n = 1000$, totaling $M = 1000$ times. Plot empirical estimates of $\mathbb{P}(|\bar{X} - \mu| > \epsilon = 0.1)$ as a function of n.
- For the distribution of X, we consider the Student's t distribution with a degree of freedom ν: $p_{\theta:=\nu}(x) \propto (1 + x^2/\nu)^{-(\nu+1)/2}$. Here, $\mu = 0$ for $\nu > 1$ and μ is undefined otherwise. Its variance is finite if and only if $\nu > 2$.

Python Code 1: Law of Large Numbers I

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import t

# Part 1: Spaghetti plot
np.random.seed(0)  # For reproducibility

# Parameters
n = 1000
M = 50
nu = 3  # degrees of freedom for t-distribution

# Plotting spaghetti plot
plt.figure(figsize=(10, 6))
for i in range(M):
    plt.plot(range(1, n+1), np.cumsum(t.rvs(df=nu, size=n)) / np.arange(1, n+1), alpha=0.5)
plt.title('Spaghetti plot of sample means for M=50 trials')
```

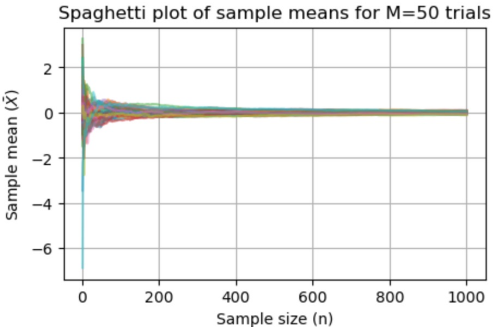
Python Code 1: Law of Large Numbers II

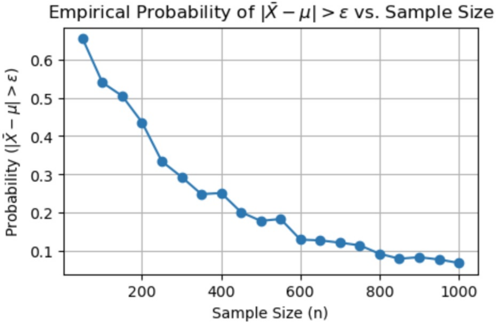
```
plt.xlabel('Sample size (n)')
plt.ylabel('Sample mean ($\\bar{X}$)')
plt.grid(True)
plt.show()

# Part 2: Empirical probability plot
N = 1000  # Max sample size
M = 1000  # Number of trials
epsilon = 0.1

# Computing probabilities
probabilities = []
sample_sizes = np.arange(50, N+1, 50)  # Step through sample sizes from 50 to N

for n in sample_sizes:
    count_within_epsilon = 0
    for _ in range(M):
```

Python Code 1: Law of Large Numbers III

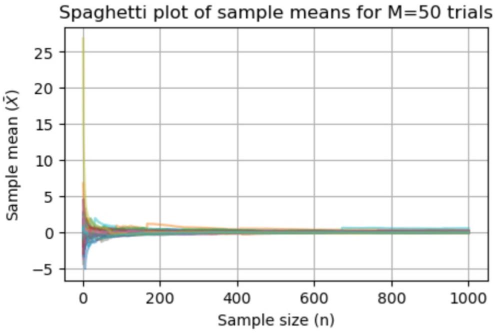
```
        sample_mean = t.rvs(df=nu, size=n).mean()
        if abs(sample_mean) > epsilon:
            count_within_epsilon += 1
    probabilities.append(count_within_epsilon / M)

# Plotting the probabilities
plt.figure(figsize=(10, 6))
plt.plot(sample_sizes, probabilities, marker='o')
plt.title('Empirical Probability of $|\\bar{X} - \\mu| > \\epsilon$ vs. Sample Size')
plt.xlabel('Sample Size (n)')
plt.ylabel('Probability ($|\\bar{X} - \\mu| > \\epsilon$)')
plt.grid(True)
plt.show()
```

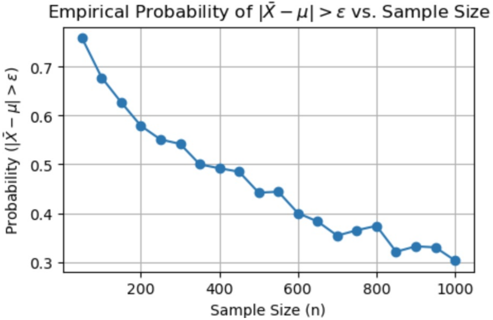
Python Code 1: Law of Large Numbers IV





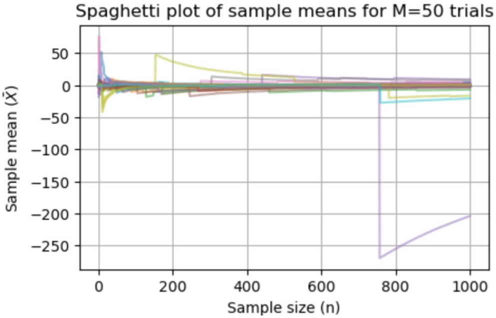
Advanced Discussion

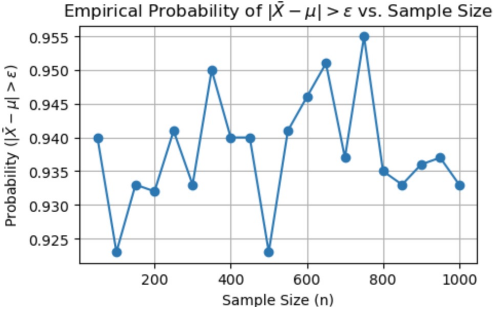




- Results when $\nu = 2$. In this case, $\mathbb{E}_X X = 0$ and $\mathrm{Var}_X\, X = \infty$.
- Given $n = 1000$, the empirical estimate of $\mathbb{P}(|\bar{X} - \mu| > \epsilon)$ increased a lot. We need larger sample sizes to empirically confirm the convergence of $\bar{X}$.

Advanced Discussion





- Results when $\nu = 1$. In this case, $\mathbb{E}_X X$ is undefined. This distribution is referred to as the *Cauchy distribution*. In the right figure, $\mu = 0$ was chosen for consistent visualization. **Q:** Conditions for the Law of Large Numbers do not hold; However, this does NOT guarantee that $\bar{X}$ diverges. Based on plots, should we conclude that $\bar{X}$ does not converge to zero (or any other point), or could this result be due to a small sample size issue?

Recapping Central Limit Theorem

- *Central Limit Theorem:* Let $\bar{X}$ be the mean of n i.i.d. samples from a distribution with a finite population mean μ and standard deviation σ. Then,

$$\sqrt{n}(\bar{X} - \mu)/\sigma \xrightarrow{d} Z \sim N(0, 1). \tag{1}$$

  That is, $\mathbb{P}(\sqrt{n}(\bar{X} - \mu)/\sigma \leq z) \to \mathbb{P}(Z \leq z)$ for any $z \in \mathbb{R}$.

- *Experiment:* We will examine whether $\mathbb{P}(\sqrt{n}(\bar{X} - \mu)/\sigma \leq z)$ converges to $\mathbb{P}(Z \leq z)$:
  1. Sample a dataset of size $n \in \{25, 50, 100, 200\}$, denoted as $(x_i)_{i=1}^n$, $M = 1000$ times to compute M estimates of $\sqrt{n}(\bar{X} - \mu)/\sigma$ for each n.
  2. Examine whether the empirical distribution functions of $\sqrt{n}(\bar{X} - \mu)/\sigma$ converge to the distribution function of a standard normal $N(0, 1)$.

- For the distribution of X, we consider a Gaussian mixture model: $p_\theta(x) = \pi_1 p(x; \mu_1, \sigma_1^2) + \pi_2 p(x; \mu_2, \sigma_2^2)$ where $\theta := (\pi_1, \pi_2, \mu_1, \sigma_1, \mu_2, \sigma_2)^T$. Here, $\mu = \pi_1\mu_1 + \pi_2\mu_2$ and $\sigma = \sqrt{(\pi_1\sigma_1^2 + \pi_2\sigma_2^2) + (\pi_1(\mu_1 - \mu)^2 + \pi_2(\mu_2 - \mu)^2)}$.

Python Code 2: Central Limit Theorem I

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import norm

# Parameters for Gaussian Mixture
mu1, sigma1 = 0, 1    # Mean and std dev of first normal distribution
mu2, sigma2 = 3, 1.5  # Mean and std dev of second normal distribution
pi1, pi2 = 0.5, 0.5    # Proportion of the first and second distribution

# Combined mean and variance
mu = pi1 * mu1 + pi2 * mu2
sigma = np.sqrt(pi1 * (sigma1**2) + pi2 * (sigma2**2)
                + pi1 * ((mu - mu1)**2) + pi2 * ((mu - mu2)**2))

# Sample sizes to test
sample_sizes = [25, 50, 100, 200]
M = 1000  # Number of experiments
```

Python Code 2: Central Limit Theorem II

```python
# Function to generate samples from a Gaussian Mixture
def generate_gaussian_mixture(n):
    n1 = np.random.binomial(n, pi1)
    n2 = n - n1
    samples1 = np.random.normal(mu1, sigma1, n1)
    samples2 = np.random.normal(mu2, sigma2, n2)
    return np.concatenate((samples1, samples2))

# Function to calculate empirical distribution function
def ecdf(data):
    n = len(data)
    x = np.sort(data)
    y = np.arange(1, n+1) / n
    return x, y

# Create figure for ECDF and CDF comparison
```

Python Code 2: Central Limit Theorem III

```
fig, ax = plt.subplots(figsize=(12, 8))

# Define line styles for different sample sizes
line_styles = ['-', '--', '-.', ':']

mise_values = []

for i, n in enumerate(sample_sizes):
    sample_means = []
    for _ in range(M):
        samples = generate_gaussian_mixture(n)
        sample_mean = np.mean(samples)
        sample_means.append(sample_mean)

    # Normalize sample means
    sample_means = np.array(sample_means)
    normalized_means = np.sqrt(n) * ((sample_means - mu) / sigma)
```

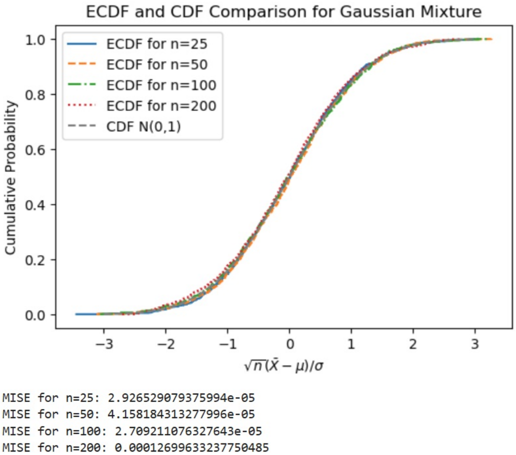
Python Code 2: Central Limit Theorem IV

```python
    # Calculate and plot ECDF
    x, y = ecdf(normalized_means)
    ax.step(x, y, where='post', label=f'ECDF for n={n}', linestyle=line_styles[i])

    # Calculate MISE
    x_grid = np.linspace(min(x), max(x), 500)
    ecdf_values = np.interp(x_grid, x, y)
    cdf_values = norm.cdf(x_grid)
    mise = np.mean((ecdf_values - cdf_values)**2)
    mise_values.append(mise)

# Plotting CDF of standard normal distribution
x_grid = np.linspace(-3, 3, 100)
ax.plot(x_grid, norm.cdf(x_grid), 'k--', label='CDF N(0,1)', color='gray')

ax.set_title('ECDF and CDF Comparison for Gaussian Mixture')
```

Python Code 2: Central Limit Theorem V

```
ax.set_xlabel(r'$\sqrt{n}(\bar{X}-\mu)/\sigma$')
ax.set_ylabel('Cumulative Probability')
ax.legend()

plt.show()

# Print MISE values for each sample size
for size, mise in zip(sample_sizes, mise_values):
    print(f'MISE for n={size}: {mise}')
```

Python Code 2: Central Limit Theorem VI



```
MISE for n=25: 2.926529079375994e-05
MISE for n=50: 4.158184313277996e-05
MISE for n=100: 2.709211076327643e-05
MISE for n=200: 0.00012699633237750485
```

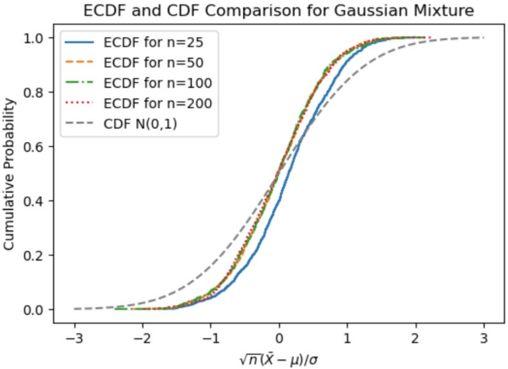
Q: Why mean integrated squared errors (MISEs) do not monotonically decrease?

Python Code 2’: Central Limit Theorem? I

- The following is the draft version of the function *generate_gaussian_mixture* that ChatGPT gave:

```
# Function to generate samples from a Gaussian Mixture
def generate_gaussian_mixture(n):
    n1 = int(n*pi1) # previous example: n1 = np.random.binomial(n, pi1)
    n2 = n - n1
    samples1 = np.random.normal(mu1, sigma1, n1)
    samples2 = np.random.normal(mu2, sigma2, n2)
    return np.concatenate((samples1, samples2))
```

Python Code 2': Central Limit Theorem? II



```
MISE for n=25: 0.007840964793214642
MISE for n=50: 0.005802817753966564
MISE for n=100: 0.004922393300011094
MISE for n=200: 0.0052024150821751965
```

Q: What happened?

Outline

1. Law of Large Numbers and Central Limit Theorem
2. Logistic Regression
3. Neural Network

Optimization

- The estimator $\hat{f}_n$ usually does not have a (computable) closed-from expression:

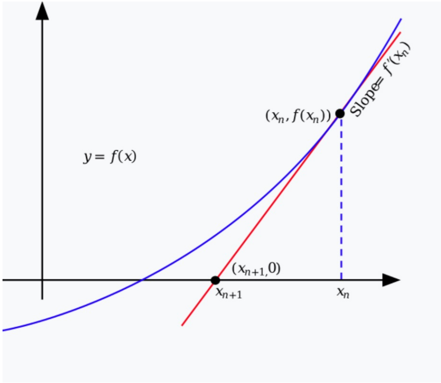
$$\hat{f}_n \in \arg\min_{f \in \mathcal{F}} n^{-1} \sum_{i=1}^{n} l(f(\vec{x}_i), y_i). \tag{2}$$

- Typical remedies are to apply iterative algorithms to solve the above optimization problem.
- In this subsection, we focus on simple logistic regression where we have an univariate X, $\mathcal{F} = \{f_\theta(x) = \sigma(\beta_0 + \beta_1 x) \mid \theta = (\beta_0, \beta_1)^T \in \mathbb{R}^2\}$, and we use the cross-entropy loss:

$$l(f_\theta(x), y) = -I(y = 1) \log \sigma(f_\theta(x)) - I(y = -1) \log(1 - \sigma(f_\theta(x)))$$

where $\sigma(x) = 1/(1 + \exp(-x))$ denotes the standard logistic function (or sigmoid function), I is the indicator function, and y takes values in $\{1, \text{-}1\}$ indicating the class labels.

Optimization: Newton-Raphson Method



- The *Newton-Raphson* method: $x^{(t+1)} = x^{(t)} - f(x^{(t)})/f'(x^{(t)})$ is used to find a solution to $f(x) = 0$.

Image source: `https://en.wikipedia.org/wiki/Newton%27s_method`.

Optimization: Newton-Raphson Method

- Let $R_n(f_\theta)$ be the empirical risk. In logistic regression, the minimizer $(\hat{f}_n =) f_{\hat{\theta}_n}$ uniquely exists and is the solution to:

$$\frac{\partial R_n(\theta)}{\partial \theta} = \begin{pmatrix} n^{-1} \sum_{i=1}^n \Big(\sigma(\beta_0 + \beta_1 x_i) - I(y_i = 1) \Big) \\ n^{-1} \sum_{i=1}^n x_i \Big(\sigma(\beta_0 + \beta_1 x_i) - I(y_i = 1) \Big) \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}. \tag{3}$$

  The $R_n(\theta)$ is the same as the negative log-likelihood, so $\hat{\theta}_n$ is a maximum likelihood estimator.[1]

- The Newton-Raphson method for the logistic regression can be expressed as:

$$\hat{\theta}_n^{(t+1)} = \hat{\theta}_n^{(t)} - \left(\partial^2 R_n(\theta) / \partial\theta \partial\theta^T \right)^{-1} \left(\partial R_n(\theta)/\partial\theta\right)\big|_{\theta = \hat{\theta}_n}. \tag{4}$$

[1] In this context, Equation (3) is a *score equation*.

Optimization: Newton-Raphson Method

- The Hessian can be expressed as:

$$\frac{\partial^2 R_n(\theta)}{\partial\theta\partial\theta^T} = \begin{pmatrix} n^{-1}\sum_{i=1}^n w_i(\theta) & n^{-1}\sum_{i=1}^n w_i(\theta)x_i \\ n^{-1}\sum_{i=1}^n w_i(\theta)x_i & n^{-1}\sum_{i=1}^n w_i(\theta)x_i^2 \end{pmatrix} \tag{5}$$

where $w_i(\theta) := \sigma(\beta_0 + \beta_1 x_i)\Big(1 - \sigma(\beta_0 + \beta_1 x_i)\Big)$.

Optimization: Newton-Raphson Method

- *Experiment:* We will examine whether $\hat{\theta}_n^{(t)}$ converges to the true θ^*:
  1. Sample a dataset of size $n \in \{100, 200, 400, 800\}$, denoted as $((x_i, y_i))_{i=1}^n$.
  2. Apply the Newton-Raphson method to find $\hat{\theta}_n^{(T=25)}$. The initial point $\hat{\theta}_n^{(0)}$ is randomly sampled from $U([-0.1, 0.1]^2)$.
  3. Repeat the previous steps $M = 500$ times to compute M estimates of $\hat{\theta}_n^{(T)}$ for each n.
  4. Plot $\|\hat{\theta}_n^{(T)} - \theta^*\|$ as a function of n and plot a scattar plot to visualize $\sqrt{n}(\hat{\theta}_n^{(T)} - \theta^*)$.
- For the distribution of (X, Y), we consider the following setting: $Y \sim 2\,\mathrm{Ber}(p) - 1$ and $X|Y = y \sim N(x; \mu_y, \sigma_\epsilon^2)$. Then, $\theta^* = ((\mu_{-1}^2 - \mu_1^2)/2\sigma_\epsilon^2, (\mu_1 - \mu_{-1})/\sigma_\epsilon^2)^T$.

Python Code 3: Newton-Raphson Method I

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import bernoulli, norm

def sigmoid(x):
    """Stable sigmoid function to prevent overflow."""
    return 1 / (1 + np.exp(-x))

def logistic_regression(x, y, theta_init, max_iter=25):
    """Performs logistic regression using the Newton-Raphson method."""
    n = len(y)
    theta = theta_init
    indicator = (y+1)/2  # Convert class labels from {1, -1} to {1, 0}

    for _ in range(max_iter):
        p = sigmoid(x @ theta)  # Current estimate of probability
        gradient = x.T @ (p - indicator) / n
```

Python Code 3: Newton-Raphson Method II

```
        W = np.diag(p * (1 - p))
        Hessian = x.T @ W @ x / n
        theta -= np.linalg.inv(Hessian) @ gradient
    return theta

def generate_data(n, mu_y, sigma_epsilon):
    """Generates binary classification data with logistic features."""
    y = 2 * bernoulli.rvs(0.5, size=n) - 1
    x = norm.rvs(loc=[mu_y[yy] for yy in y], scale=sigma_epsilon, size=n)
    return np.vstack([np.ones(n), x]).T, y  # Add intercept

# Setup parameters
sample_sizes = [100, 200, 400, 800]
M = 500
mu_y = {1: 1, -1: -2}
sigma_epsilon = 1.0
theta_star = np.array([(mu_y[-1]**2 - mu_y[1]**2) / (2 * sigma_epsilon**2),
```

Python Code 3: Newton-Raphson Method III

```
                       (mu_y[1] - mu_y[-1]) / sigma_epsilon])

norm_diffs = []
scaled_diffs = {n: [] for n in sample_sizes}

# Perform simulations
for n in sample_sizes:
    theta_estimates = []
    for _ in range(M):
        x, y = generate_data(n, mu_y, sigma_epsilon)
        theta_init = np.random.uniform(-0.1, 0.1, 2)
        theta_hat = logistic_regression(x, y, theta_init)
        norm_diff = np.linalg.norm(theta_hat - theta_star)
        theta_estimates.append(norm_diff)
        scaled_diff = np.sqrt(n) * (theta_hat - theta_star)
        scaled_diffs[n].append(scaled_diff)
    norm_diffs.append(np.mean(theta_estimates))
```

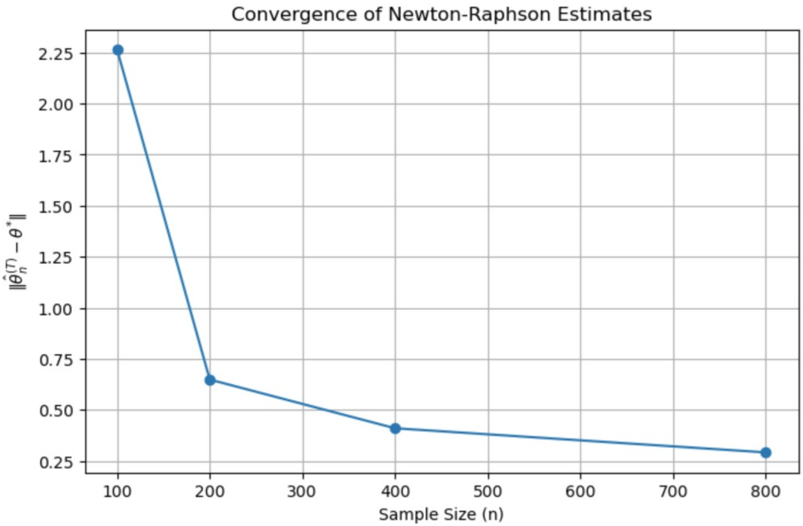
Python Code 3: Newton-Raphson Method IV

```
# Plot for norm differences
plt.figure(figsize=(8, 5))
plt.plot(sample_sizes, norm_diffs, marker='o')
plt.xlabel('Sample Size (n)')
plt.ylabel(r'$\|\hat{\theta}_{n}^{(T)} - \theta^{*}\|$')
plt.title('Convergence of Newton-Raphson Estimates')
plt.grid(True)
plt.show()

# Plot scaled differences as scatter plots
fig, axes = plt.subplots(1, len(sample_sizes), figsize=(20, 5), sharex=True, sharey=True)
for i, n in enumerate(sample_sizes):
    diffs = np.array(scaled_diffs[n])
    sigma_x = np.std(diffs[:, 0])
    sigma_y = np.std(diffs[:, 1])
    limit_x = 3 * sigma_x
```

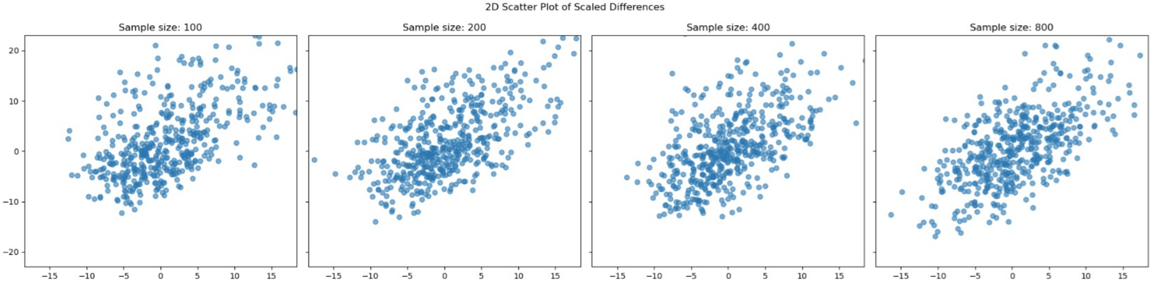
Python Code 3: Newton-Raphson Method V

```python
    limit_y = 3 * sigma_y

    axes[i].scatter(diffs[:, 0], diffs[:, 1], alpha=0.6)
    axes[i].set_title(f'Sample size: {n}')
    axes[i].set_xlim([-limit_x, limit_x])
    axes[i].set_ylim([-limit_y, limit_y])

plt.suptitle('2D Scatter Plot of Scaled Differences')
plt.tight_layout()
plt.show()
```

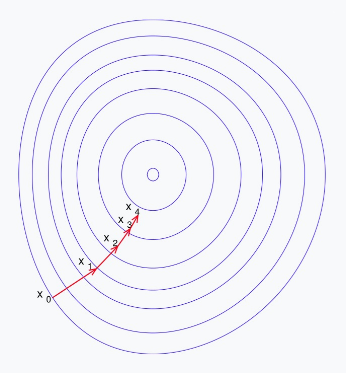
Results: Newton-Raphson Method



Results: Newton-Raphson Method



Optimization: Gradient Descent Method



- The gradient descent method: $x^{(t+1)} = x^{(t)} - \gamma_t f'(x^{(t)})$ is used to find a solution to $f(x) = 0$ where γ_t represents learning rates.

Image source: `https://en.wikipedia.org/wiki/Gradient_descent`.

Optimization: Gradient Descent Method

- The gradient descent method for the logistic regression can be expressed as:

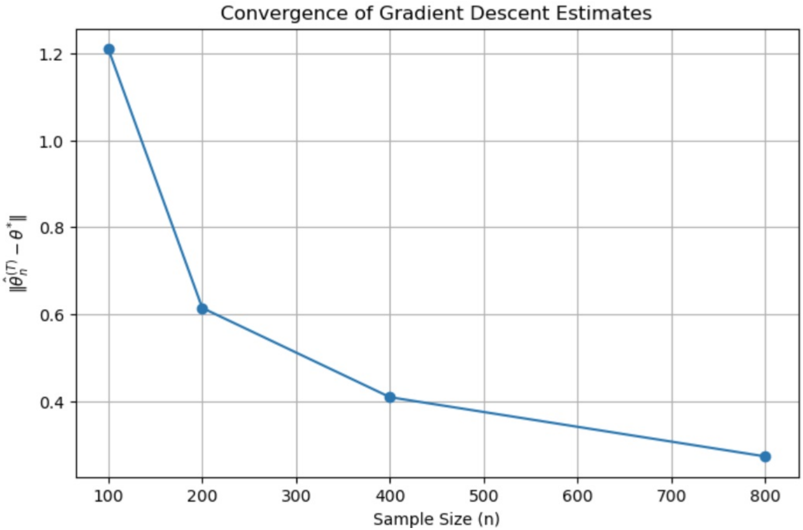
$$\hat{\theta}_n^{(t+1)} = \hat{\theta}_n^{(t)} - \gamma_t \Big(\partial R_n(\theta) / \partial \theta \Big) |_{\theta = \hat{\theta}_n}. \tag{6}$$

- *Experiment:* We use the experimental setting from the previous example for the Newton-Raphson method while updating the optimization part to the gradient descent method with $T = 1000$ and $\gamma_t = 50/(t + 50)$.

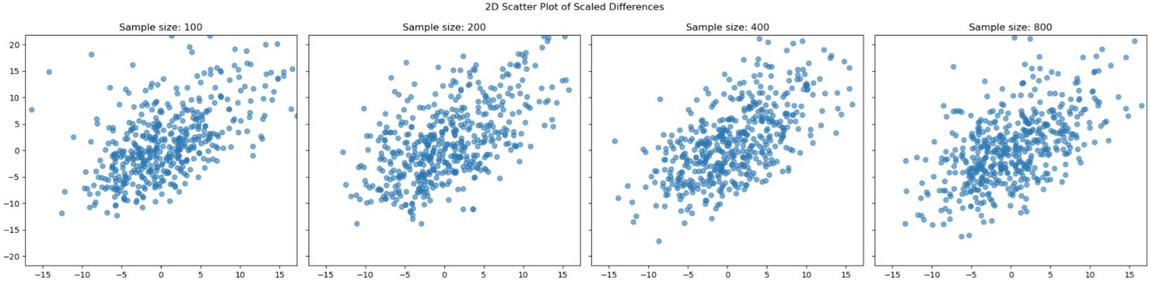
Python Code 4: Gradient Descent Method

```python
def gd_logistic_regression(x, y, theta_init, init_learning_rate=20.0, max_iter=1000):
    """Performs logistic regression using Gradient Descent."""
    n = len(y)
    theta = theta_init
    indicator = (y+1)/2  # Convert class labels from {1, -1} to {1, 0}

    for t in range(max_iter):
        learning_rate = init_learning_rate*(50/(t+50))
        p = sigmoid(x @ theta)
        gradient = x.T @ (p - indicator)/n
        theta -= learning_rate * gradient
    return theta
```

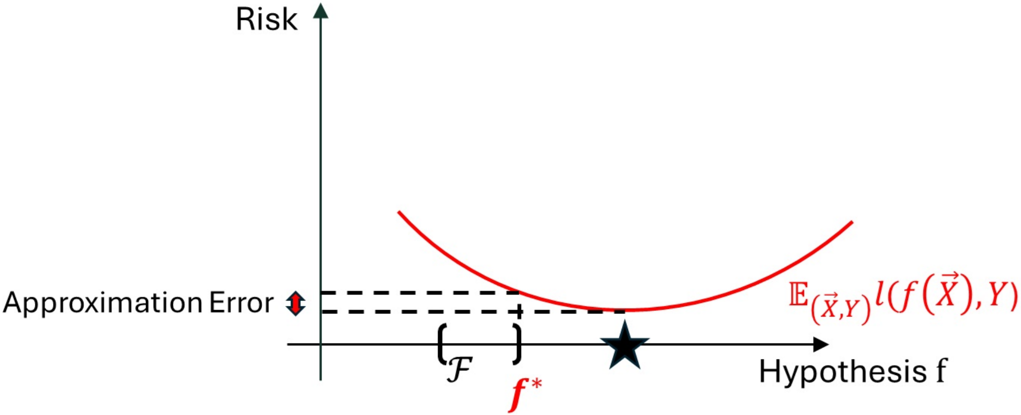
Results: Gradient Descent Method



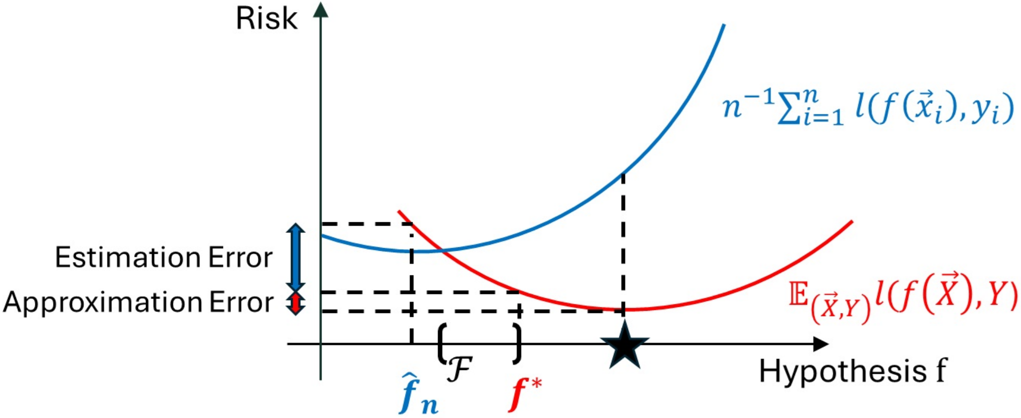
Results: Gradient Descent Method



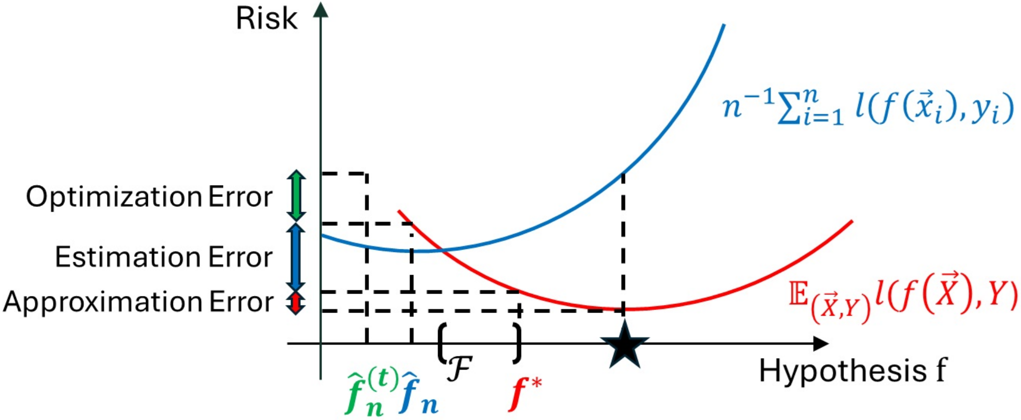
Recapping Approximation Error



Recapping Estimation Error


Risk
$n^{-1}\sum_{i=1}^{n} l(f(\vec{x}_i), y_i)$
Estimation Error
Approximation Error
$\mathbb{E}_{(\vec{X},Y)} l(f(\vec{X}), Y)$
$\mathcal{F}$
Hypothesis f
$\hat{f}_n$
f^*

Optimization Error



Outline

1. Law of Large Numbers and Central Limit Theorem

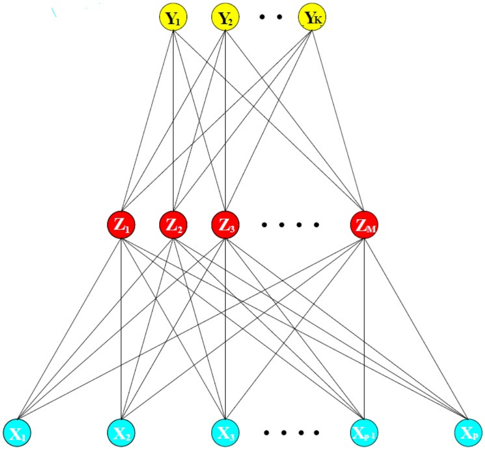
2. Logistic Regression

3. Neural Network

Introduction to PyTorch

- PyTorch is a popular open-source machine learning library for Python, known for its flexibility and ease of use in research.
- It provides two high-level features:
  1. Tensor computation (like NumPy) with strong GPU acceleration
  2. Automatic differentiation for building and training neural networks
- Further details about PyTorch will be provided in a later main lecture, along with implementations of several deep generative models. In this lecture, we will review two basic neural network-based classifiers on the MNIST dataset.

Recapping Neural Network



The figure is from Hastie (2009).

Recapping Neural Network

- Outputs from neural networks with one hidden layer can be expressed as:

$$f_\theta(\vec{x}) = \beta_0 + \sum_{h=1}^{H} \beta_h \sigma(\alpha_h + \vec{w}_h^T \vec{x}) \tag{7}$$

where $\theta := (\beta_0, \ldots, \beta_H, \alpha_1, \ldots, \alpha_H, \vec{w}_1, \ldots, \vec{w}_H)^T$ and $g_h(\cdot) = \beta_h \sigma(\cdot)$ where σ is the sigmoid function (can be replaced with other activation functions). Here, $\alpha_h + \vec{w}_h^T \vec{x}$ represents the h-th hidden node value, and $(\beta_1, \cdots, \beta_H)^T$ and β_0 are weight vector and bias term, respectively, for the output layer.

PyTorch Code 1: Dense Network with One Hidden Layer I

```
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision.transforms as transforms
import torchvision.datasets as datasets
import matplotlib.pyplot as plt
from torch.utils.data import DataLoader

# Define the neural network with one hidden layer
class FCNet(nn.Module):
    def __init__(self):
        super(FCNet, self).__init__()
        self.fc1 = nn.Linear(28*28, 512)  # Flatten 28x28 images to a 784 vector for each imag
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(512, 10)  # 10 output classes

    def forward(self, x):
```

PyTorch Code 1: Dense Network with One Hidden Layer II

```
        x = x.view(-1, 28*28)  # Flatten the images
        x = self.relu(self.fc1(x))
        x = self.fc2(x)
        return x

# Load the MNIST dataset
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])

train_dataset = datasets.MNIST(root='./data', train=True, download=True, transform=transform)
test_dataset = datasets.MNIST(root='./data', train=False, download=True, transform=transform)

train_loader = DataLoader(dataset=train_dataset, batch_size=64, shuffle=True)
test_loader = DataLoader(dataset=test_dataset, batch_size=1000, shuffle=False)
```

PyTorch Code 1: Dense Network with One Hidden Layer III

```python
# Initialize the network and optimizer
model = FCNet()
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.5)

# Training the network
def train(num_epochs, model, loaders):
    train_loader, test_loader = loaders
    for epoch in range(num_epochs):
        model.train()
        for data, target in train_loader:
            optimizer.zero_grad()
            output = model(data)
            loss = criterion(output, target)
            loss.backward()
            optimizer.step()
```

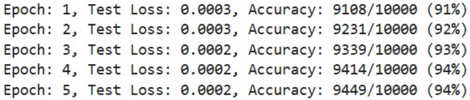
PyTorch Code 1: Dense Network with One Hidden Layer IV

```
        model.eval()
        test_loss = 0
        correct = 0
        with torch.no_grad():
            for data, target in test_loader:
                output = model(data)
                test_loss += criterion(output, target).item()
                pred = output.data.max(1, keepdim=True)[1]
                correct += pred.eq(target.data.view_as(pred)).sum()
        test_loss /= len(test_loader.dataset)
        print(f'Epoch: {epoch+1}, Test Loss: {test_loss:.4f},
        Accuracy: {correct}/{len(test_loader.dataset)} ({100. * correct / len(test_loader.data

# Visualization of results
def visualize(model, loader):
    model.eval()
    data, _ = next(iter(loader))
```

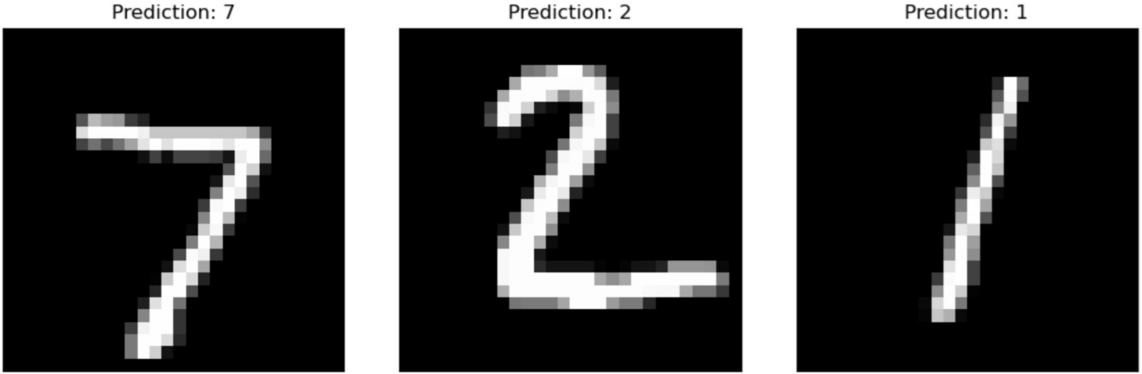
PyTorch Code 1: Dense Network with One Hidden Layer V

```
    with torch.no_grad():
        output = model(data)
    plt.figure(figsize=(10, 8))
    for i in range(6):
        plt.subplot(2, 3, i + 1)
        plt.tight_layout()
        plt.imshow(data[i][0], cmap='gray', interpolation='none')
        plt.title("Prediction: {}".format(output.data.max(1, keepdim=True)[1][i].item()))
        plt.xticks([])
        plt.yticks([])
    plt.show()

# Run training and visualization
train(5, model, (train_loader, test_loader))
visualize(model, test_loader)
```

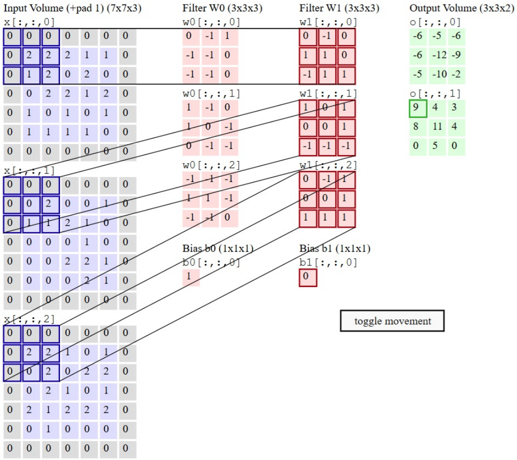
Result: Dense Neural Network



```
Epoch: 1, Test Loss: 0.0003, Accuracy: 9108/10000 (91%)
Epoch: 2, Test Loss: 0.0003, Accuracy: 9231/10000 (92%)
Epoch: 3, Test Loss: 0.0002, Accuracy: 9339/10000 (93%)
Epoch: 4, Test Loss: 0.0002, Accuracy: 9414/10000 (94%)
Epoch: 5, Test Loss: 0.0002, Accuracy: 9449/10000 (94%)
```



Recapping Convolutional Neural Network



Input Volume (+pad 1) (7x7x3)

x[:,:,0]

0	0	0	0	0	0	0
0	2	2	2	1	1	0
0	1	2	0	2	0	0
0	0	2	2	1	2	0
0	1	0	1	0	1	0
0	1	1	1	1	0	0
0	0	0	0	0	0	0

x[:,:,1]

0	0	0	0	0	0	0
0	0	2	0	0	1	0
0	1	1	2	1	0	0
0	0	0	0	1	0	0
0	0	0	2	2	1	0
0	0	0	0	2	1	0
0	0	0	0	0	0	0

x[:,:,2]

0	0	0	0	0	0	0
0	2	2	1	0	1	0
0	0	2	0	2	2	0
0	0	2	1	0	1	0
0	2	1	2	2	2	0
0	0	1	0	0	0	0
0	0	0	0	0	0	0

Filter W0 (3x3x3)

w0[:,:,0]

0	-1	1
-1	-1	0
-1	-1	0

w0[:,:,1]

1	-1	0
1	0	-1
0	-1	-1

w0[:,:,2]

-1	-1	-1
1	1	-1
-1	-1	0

Bias b0 (1x1x1)

b0[:,:,0]

1

Filter W1 (3x3x3)

w1[:,:,0]

0	-1	0
1	1	0
-1	1	1

w1[:,:,1]

1	0	1
0	0	1
-1	-1	-1

w1[:,:,2]

0	-1	1
0	0	1
1	1	1

Bias b1 (1x1x1)

b1[:,:,0]

0

Output Volume (3x3x2)

o[:,:,0]

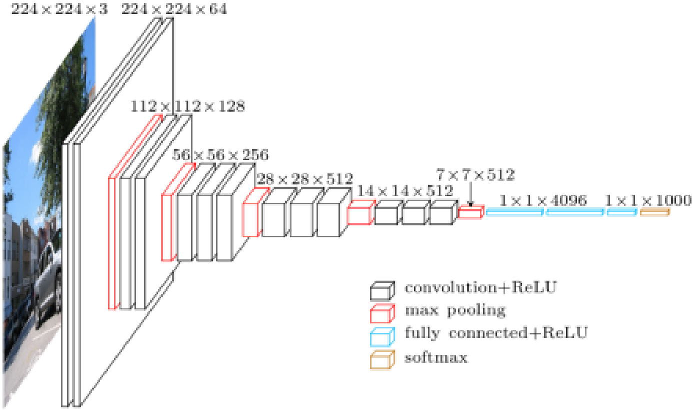
-6	-5	-6
-6	-12	-9
-5	-10	-2

o[:,:,1]

9	4	3
8	11	4
0	5	0

The figure is from `https://cs231n.github.io/convolutional-networks/`.

Recapping Convolutional Neural Network



The figure is from Simonyan (2014).

PyTorch Code 2: Convolutional Neural Network I

```python
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision.transforms as transforms
import torchvision.datasets as datasets
import matplotlib.pyplot as plt
from torch.utils.data import DataLoader

# Define the neural network architecture
class SimpleCNN(nn.Module):
    def __init__(self):
        super(SimpleCNN, self).__init__()
        self.conv1 = nn.Conv2d(1, 10, kernel_size=5)
        self.conv2 = nn.Conv2d(10, 20, kernel_size=5)
        self.pool = nn.MaxPool2d(2)
        self.fc1 = nn.Linear(320, 50)
        self.fc2 = nn.Linear(50, 10)
```

PyTorch Code 2: Convolutional Neural Network II

```python
    def forward(self, x):
        x = self.pool(torch.relu(self.conv1(x)))
        x = self.pool(torch.relu(self.conv2(x)))
        x = x.view(-1, 320)
        x = torch.relu(self.fc1(x))
        x = self.fc2(x)
        return x

# Load the MNIST dataset
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])

train_dataset = datasets.MNIST(root='./data', train=True, download=True, transform=transform)
test_dataset = datasets.MNIST(root='./data', train=False, download=True, transform=transform)
```

PyTorch Code 2: Convolutional Neural Network III

```python
train_loader = DataLoader(dataset=train_dataset, batch_size=64, shuffle=True)
test_loader = DataLoader(dataset=test_dataset, batch_size=1000, shuffle=False)

# Initialize the network and optimizer
model = SimpleCNN()
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.5)

# Train the network
def train(num_epochs, model, loaders):
    train_loader, test_loader = loaders
    for epoch in range(num_epochs):
        model.train()
        for data, target in train_loader:
            optimizer.zero_grad()
            output = model(data)
```

PyTorch Code 2: Convolutional Neural Network IV

```
    loss = criterion(output, target)
    loss.backward()
    optimizer.step()

model.eval()
test_loss = 0
correct = 0
with torch.no_grad():
    for data, target in test_loader:
        output = model(data)
        test_loss += criterion(output, target).item()
        pred = output.data.max(1, keepdim=True)[1]
        correct += pred.eq(target.data.view_as(pred)).sum()
test_loss /= len(test_loader.dataset)
print(f'Epoch: {epoch+1}, Test Loss: {test_loss:.4f},
Accuracy: {correct}/{len(test_loader.dataset)} ({100. * correct / len(test_loader.data
```

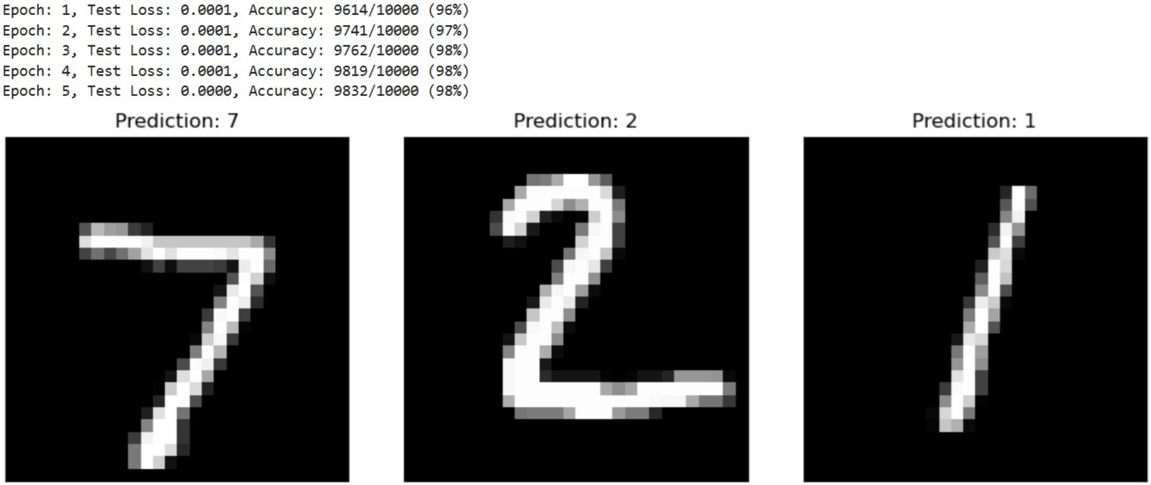
PyTorch Code 2: Convolutional Neural Network V

```python
# Visualizing some results
def visualize(model, loader):
    model.eval()
    data, _ = next(iter(loader))
    with torch.no_grad():
        output = model(data)
    plt.figure(figsize=(10, 8))
    for i in range(6):
        plt.subplot(2, 3, i + 1)
        plt.tight_layout()
        plt.imshow(data[i][0], cmap='gray', interpolation='none')
        plt.title("Prediction: {}".format(output.data.max(1, keepdim=True)[1][i].item()))
        plt.xticks([])
        plt.yticks([])
    plt.show()

# Run training and visualization
```

PyTorch Code 2: Convolutional Neural Network VI

```
train(5, model, (train_loader, test_loader))
visualize(model, test_loader)
```

Result: Convolutional Neural Network



```
Epoch: 1, Test Loss: 0.0001, Accuracy: 9614/10000 (96%)
Epoch: 2, Test Loss: 0.0001, Accuracy: 9741/10000 (97%)
Epoch: 3, Test Loss: 0.0001, Accuracy: 9762/10000 (98%)
Epoch: 4, Test Loss: 0.0001, Accuracy: 9819/10000 (98%)
Epoch: 5, Test Loss: 0.0000, Accuracy: 9832/10000 (98%)
```

References I

Hastie, T. (2009). The elements of statistical learning: data mining, inference, and prediction.

Simonyan, K. (2014). Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*.