CS8395 Assignment2 Kun Peng

1 Error Composition

1. **Population Error for Hypothesis $\hat{f}$**: The population error for a hypothesis $\hat{f}$ is the expected loss over the distribution of all possible data points. Mathematically, it's defined as $\mathcal{R}(\hat{f}) = \mathbb{E}[\ell(\hat{f}(X), Y)]$, where ℓ is the loss function, **X** is the input, and **Y** is the true output.
2. **Decomposition of Population Error**:
   - The population error can be decomposed as follows:
     $\mathcal{R}(\hat{f}) - \mathcal{R}(f^*) = (\mathcal{R}(\hat{f}) - \hat{\mathcal{R}}(\hat{f})) + (\hat{\mathcal{R}}(\hat{f}) - \hat{\mathcal{R}}(f^*)) + (\hat{\mathcal{R}}(f^*) - \mathcal{R}(f^*))$,
     where f^* is the best hypothesis in the class $\mathcal{F}$.
   - The first term is the statistical error, the second term is the optimization error, and the third term is the approximation error.
   - The proof would involve showing that these errors represent respectively: variance due to finite sample size, the gap due to the optimization algorithm, and the bias due to the limited capacity of the hypothesis class.
3. **Meaning and Influence on Errors**:
   - Optimization error increases if the optimization algorithm fails to find the best solution within $\mathcal{F}_\delta$.
   - Approximation error increases if $\mathcal{F}_\delta$ is too simple to capture the underlying data structure.
   - Statistical error decreases with more data and typically increases with higher model complexity.
4. **Impact of Gradient Descent (GD) on Error Terms**:
   - GD primarily affects the optimization error as it's the method used to minimize the empirical risk$\hat{\mathcal{R}}$.
5. **Data Augmentation's Effect on Error Terms**:
   - Data augmentation primarily aims to reduce the statistical error by effectively increasing the size and diversity of the training dataset, which can help the model generalize better to unseen data.

2 Group Equivariance and Group Invariance

1. **G-Invariance**: A function $h : X \to X$ is said to be G-Invariant with respect to a group G if the application of any group element $g \in G$ to the domain Ω does not change the outcome of h. Formally, for all $g \in G$, $h(\rho(g)x) = h(x)$.
2. **G-Equivariance**: A function $f : X \to X$ is G-Equivariant with respect to a group G if the application of a group element $g \in G$ to the domain is commutative with the function f. That is, for all $g \in G$, $f(\rho(g)x) = \rho(g)f(x)$.
3. **Compositions**:
   - **h ∘ f**: If h is G-Invariant and f is G-Equivariant, then the composition $h(f(x))$ for any $g \in G$ would be $h(\rho(g)f(x)) = h(f(\rho(g)x))$, since h is G-Invariant and does not change with the application of $\rho(g)$. Therefore, $h \circ f$ is G-Invariant.

- **f ∘ h**: For the composition $f(h(x))$ and for any $g \in G$, we have $f(\rho(g)h(x))$. Since h is G-Invariant, $\rho(g)h(x) = h(x)$, and thus $f(\rho(g)h(x)) = f(h(x))$. However, whether $f(h(x))$ equals $\rho(g)f(h(x))$ depends on the specific forms of f and h. Generally, without more information, we cannot conclude that $f \circ h$ is G-Invariant or G-Equivariant.

3 Deep Learning Blue Print

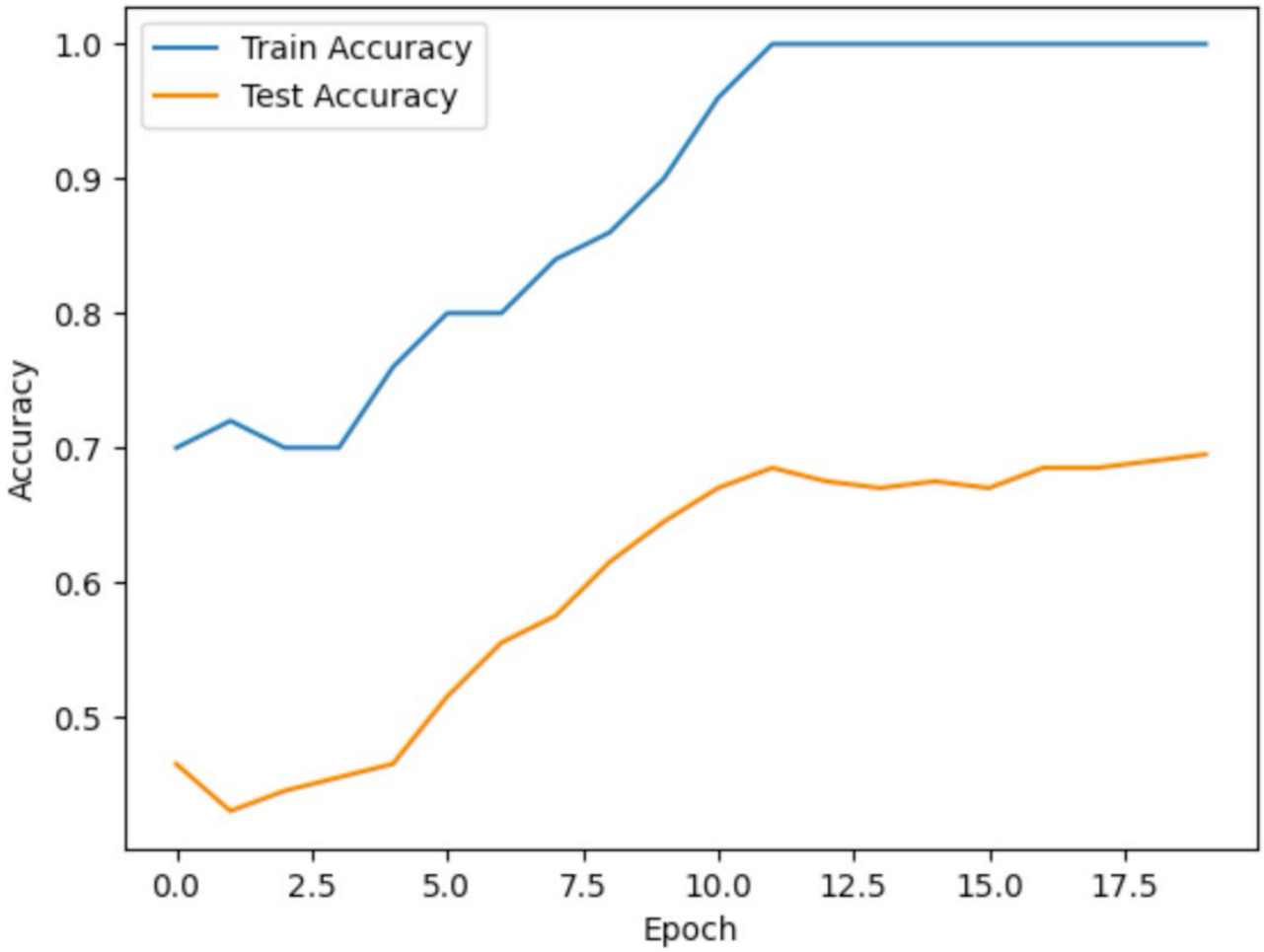
Part 1

1. **Model Architecture**: Given the signal dimensionality (d=1000), an MLP with one or more hidden layers can be used. The choice of the number of layers and the number of units per layer can impact the model's capacity. For achieving 100% training accuracy, a model with sufficient capacity is required. However, over-parameterizing can lead to overfitting.
2. **Training Procedure**: Use the Adam optimizer as specified. The model will be trained for multiple epochs, and training and test accuracies will be tracked and plotted.
3. **Accuracy Calculation & Plotting**: After each epoch, we'll calculate and plot the training and test accuracies.

```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset
import matplotlib.pyplot as plt
import numpy as np
import pickle
# Load the dataset
with open('hw2_p3.pkl', 'rb') as file:
    data = pickle.load(file)
    # Assuming the data dictionary contains 'train' and 'test' as keys
    print(data[0])
    print(data[1])
    X_train = data[0]
    y_train = data[1]
    X_test = data[2]
    y_test = data[3]

class SimpleMLP(nn.Module):
    def __init__(self, input_dim, output_dim):
        super(SimpleMLP, self).__init__()
        self.fc1 = nn.Linear(input_dim, 512)
        self.fc2 = nn.Linear(512, 128)
        self.fc3 = nn.Linear(128, output_dim)
        self.relu = nn.ReLU()

    def forward(self, x):
        x = self.relu(self.fc1(x))
        x = self.relu(self.fc2(x))
```

```
        x = self.fc3(x)
        return x


model = SimpleMLP(input_dim=1000, output_dim=2)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters())


def train_model(model, criterion, optimizer, X_train, y_train, X_test, y_test, epochs=20):
    train_acc, test_acc = [], []
    for epoch in range(epochs):
        # Training phase
        model.train()
        optimizer.zero_grad()
        outputs = model(X_train)
        loss = criterion(outputs, y_train)
        loss.backward()
        optimizer.step()

        # Accuracy computation
        with torch.no_grad():
            model.eval()
            train_correct = (torch.argmax(model(X_train), dim=1) == y_train).sum().item()
            test_correct = (torch.argmax(model(X_test), dim=1) == y_test).sum().item()
            train_acc.append(train_correct / len(y_train))
            test_acc.append(test_correct / len(y_test))

    return train_acc, test_acc


# Assuming train and test datasets are prepared
train_loader = DataLoader(TensorDataset(X_train, y_train), batch_size=64, shuffle=True)
test_loader = DataLoader(TensorDataset(X_test, y_test), batch_size=64)

# Training the model
train_accuracy, test_accuracy = train_model(model, criterion, optimizer, X_train, y_train,
X_test, y_test)

# Plotting
plt.plot(train_accuracy, label='Train Accuracy')
plt.plot(test_accuracy, label='Test Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
```

here is the plot:



Model Parameters Calculation

For a model with a single hidden layer of 100 units:

- First layer parameters: $1000 \times 100 + 100 (weights + biases)$
- Second layer parameters: $100 \times 2 + 2 (weights + biases)$

Total parameters = $1000 \times 100 + 100 + 100 \times 2 + 2 = 100302$.

Part 2

1. **Augmentation Strategy**: Before feeding each training sample to the model, flip a coin. With probability 0.5, apply a random circular shift to the sample. This is intended to increase the robustness of the model to shifts in the input data.
2. **Impact on Generalization**: This approach encourages the model to learn shift-invariant features, which could improve generalization to unseen test data by effectively increasing the variety of training samples.
3. **Implementation Note**: During training, modify the data loading or preprocessing step to include the

random circular shift with a 50% chance.

Here is the plot of the

What Data Augmentation Enforces

Data augmentation via random circular shifts forces the network to learn features that are invariant to the position of the Gaussians in the input signals. This not only increases the effective size of the training dataset but also encourages the network to focus on the presence and characteristics of the Gaussians, rather than their exact positions, potentially improving the model's ability to generalize from the training data to unseen test data.

Part 3

Network Design

1. **Convolutional Layer**: The first layer is a 1D convolutional layer with 16 kernels of size 25, circular padding, and padding size equal to half the kernel size $ksize//2$ This layer is translation equivariant.
2. **Nonlinearity**: After the convolutional layer, apply a ReLU nonlinearity.
3. **Global Average Pooling (GAP)**: This layer will take the average over all dimensions, making the feature map translation invariant. It reduces each feature map to a single number by taking the average of all values.
4. **Linear Classifier**: The output of the GAP layer is then fed into a linear classifier. Assuming the global average pooling outputs a 16-dimensional vector (since we have 16 filters), the input features to the linear classifier will be 16, and the output size will depend on the number of classes in your classification problem. For simplicity, let's assume it's a binary classification problem, so the output size will be 1.

PyTorch Implementation

Here is a simplified implementation in PyTorch:

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

class OneDConvNet(nn.Module):
    def __init__(self, num_classes=2, ksize=25):
        super(OneDConvNet, self).__init__()
        self.conv1 = nn.Conv1d(1, 16, kernel_size=ksize, padding=ksize//2,
padding_mode='circular')
        self.global_avg_pool = nn.AdaptiveAvgPool1d(1)
        self.fc = nn.Linear(16, num_classes)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = self.global_avg_pool(x)
```

```
        x = torch.flatten(x, 1) # Flatten the output for the linear layer
        x = self.fc(x)
        return x

# Assuming binary classification
model = OneDConvNet(num_classes=2, ksize=25)

# Use Adam optimizer
optimizer = optim.Adam(model.parameters())
```

Parameter Count

- **Convolutional Layer**: `(ksize * input_channels * output_channels) + output_channels` = `(25 * 1 * 16) + 16` = 416
- **Linear Classifier**: `(input_features * output_features) + output_features` = `(16 * 2) + 2` = 34

Total parameters = 416 (Conv1d) + 34 (Linear) = 450 parameters.

Generalization to Test Data

The generalization capability can be inferred from the test set performance. Techniques like cross-validation, regularization (e.g., dropout), and monitoring the loss and accuracy curves for both training and test sets during training can provide insights into how well the model is generalizing.

Without actual training and evaluation on a dataset, it's not possible to definitively say whether this network generalizes well to test data. The performance would depend on factors like the complexity of the dataset, the representativeness of the training data, and hyperparameter tuning.