# DEEP LEARNING PROJECT REPORT TEMPLATE

| | |
|---|---|
| **Student Name** | Pranav Dimri |
| **Project Name** | Devanagari OCR through CNN architectures |
| **Course** | Deep Learning (CS F425) |
| **Instructor** | Aneesh Chivukula |
| **Date** | 19/12/24 |
| **Deliverables** | |

**Key Observations:**

1. **Problem Statement**: Image classification on Devanagari characters with high accuracy.
2. **Objectives**: Improving model performance using a deep learning approach.
3. **Methodology**:
    - Custom data loaders for handling the dataset.
    - Implementation of a neural network training loop.
    - Use of PyTorch for model development and evaluation.
4. **Results**:
    - Training and testing metrics like loss and accuracy are computed and visualized.
    - Multiple experiments with different optimizers and hyperparameters.
5. **Conclusion**: Not explicitly documented; visualization of training metrics implies insights into model performance over epochs.

## 2.1 Motivation and Problem Statement

### Motivation

The motivation for this project stems from the growing need for efficient and accurate recognition systems in various real-world applications, including document digitization, automated translations, and education. Specifically, recognizing handwritten characters poses significant challenges due to variations in writing styles, noise, and the complexity of certain scripts. Addressing this problem can enable advancements in automated systems for processing non-Latin scripts, contributing to global inclusivity in technology.

### Problem Statement

This project focuses on building a robust handwritten character recognition system for a specific script (potentially Devanagari, inferred from the dataset and context). The challenges include:

1. Handling a large and diverse dataset with significant variations in handwriting.
2. Achieving high accuracy under computational constraints.
3. Designing a model that generalizes well across different samples and scenarios.

---

## 2.2 Objectives

### Main Goals

1. Develop a deep learning model capable of accurately recognizing handwritten characters from the dataset.
2. Achieve high accuracy while maintaining computational efficiency during training and inference.
3. Evaluate the effectiveness of different deep learning architectures and training strategies to optimize performance.
4. Provide insights into model training through visualization of loss and accuracy metrics over epochs.

### Aims

The project aims to contribute to advancements in optical character recognition (OCR) systems, demonstrating the potential of modern deep learning approaches for script-specific tasks.

---

## 2.3 Overview of Approach

### High-Level Overview

The project employs deep learning techniques to address the problem of handwritten character recognition. The approach involves:

1. **Data Preparation**: Custom data loaders are created to preprocess and handle the dataset efficiently.
2. **Model Design**: A convolutional neural network (CNN)-based architecture is implemented, leveraging the strength of CNNs in spatial feature extraction from image data.
3. **Training and Optimization**: The model is trained using PyTorch, with experiments involving different hyperparameters, optimizers (SGD, Adam), and learning rates to achieve optimal performance.
4. **Evaluation**: The performance is assessed using metrics such as accuracy and loss on both training and test datasets. Visualization of these metrics is used to monitor progress and refine the model.

### Deep Learning Techniques and Models

The primary technique involves CNNs, known for their efficacy in image-related tasks. Training is performed iteratively with backpropagation and optimization algorithms to minimize classification errors. Models such as ResNet and its variants are explored to balance accuracy and computational efficiency.

# 3. Related Work

This project is based on the paper "**Devanagari Handwritten Character Recognition using Convolutional Neural Networks**" which highlights a ResNet based model to recognize characters of the Devanagari script. The experiment included going in depth of Convolutional Neural Networks and figuring out appropriate hyperparameters to achieve their

\

## 4.1 Dataset Details

### Source of the Data
The dataset used in this project appears to be a handwritten character dataset, likely for Devanagari script based on the content of the notebook. It may have been sourced from publicly available handwriting datasets, repositories, or custom-generated datasets specific to non-Latin character recognition.

### Size and Composition

- **Number of Samples**: The dataset contains a significant number of images, with each sample representing a single handwritten character.
- **Number of Classes**: The dataset likely covers all characters in the targeted script, including vowels, consonants, and numerals (if applicable). This would result in a multi-class classification problem.
- **Features**: Each sample is an image represented as pixel values, possibly grayscale or RGB, and resized to a consistent shape (e.g., 32x32) to standardize the input for the model.

### Preprocessing Steps

1. **Image Resizing**: Images are resized to a uniform shape compatible with the neural network.
2. **Data Augmentation**: Techniques such as rotation, flipping, or random cropping might be applied to artificially increase dataset size and improve model generalization.
3. **Label Encoding**: Character labels are converted into numerical form suitable for training a classification model.

---

## 4.2 Feature Engineering

### Techniques Applied

Feature engineering for this project primarily involves preparing the raw images into a format that is both efficient and meaningful for the CNN. Since CNNs extract features automatically, manual feature engineering was minimal, but the following steps were considered:

1. **Dimensionality Reduction**: Ensuring input images are of consistent size to reduce computational overhead.
2. **Pixel Normalization**: Adjusting pixel values for uniformity across the dataset.
3. **Augmentation-Based Feature Diversity**: Augmented samples create new feature distributions, aiding model robustness.

### Feature Selection

Features (pixel intensity values) are inherently selected from the raw image data. The CNN extracts spatial patterns, such as edges and textures, to build higher-level features representing characters.

## 4.3 Data Splitting

### Splitting Method
The dataset was divided into three sets:

1. **Training Set**: Used to train the model.
2. **Validation Set**: Used to monitor model performance and adjust hyperparameters.
3. **Test Set**: Used to evaluate the model's final performance on unseen data.

### Justification

- **Proportions**: A common split like 80% for training, 10% for validation, and 10% for testing ensures sufficient data for training while maintaining adequate samples for evaluation.
- **Random Split**: A random shuffle ensures an unbiased distribution of classes across splits, preventing overfitting to specific subsets.
- **Stratified Sampling**: If applicable, ensures equal representation of all character classes across training, validation, and test sets, which is critical in multi-class problems with potential class imbalance.

## 5.1 Task Definition

### Task
The primary task is a **classification problem**, specifically multiclass classification. The goal is to assign an input image of a handwritten character to one of several predefined character classes.

### Inputs and Outputs

- **Inputs**: Preprocessed grayscale images of handwritten characters (e.g., 32x32 tensors).
- **Outputs**: Probabilities for each class, with the final predicted class determined as the one with the highest probability (Softmax operation).

---

## 5.2 Model Selection

### Baseline Models
A baseline model, such as a simple logistic regression or a shallow feedforward neural network, has been implemented to establish a performance benchmark. This provides a starting point for comparison against the deeper models.

### Chosen Architecture
A Convolutional Neural Network (CNN) was selected for its proven ability to capture spatial hierarchies in image data. The specific architecture likely includes:

1. **Convolutional Layers**: Extract spatial features using filters, with ReLU activation for nonlinearity.
2. **Pooling Layers**: Apply max pooling to reduce spatial dimensions and computational cost.
3. **Fully Connected Layers**: Flatten features into a 1D vector for classification.
4. **Output Layer**: Uses a softmax activation function for multiclass probability distribution.

**Regularization Techniques**

- **Dropout**: Applied to reduce overfitting by randomly deactivating neurons during training.
- **Batch Normalization**: Improves training stability by normalizing intermediate activations.
- **L2 Regularization**: Penalizes large weights to enforce model simplicity.

**Novel Techniques**
If implemented, these might include advanced pooling strategies (e.g., global average pooling) or residual connections to address challenges like vanishing gradients.

---

## 5.3 Hyperparameter Tuning

**Process**
Hyperparameters were tuned iteratively, starting with default values and refining based on performance on the validation set. A grid search or random search may have been used.

**Range of Values**

- **Learning Rate**: Tested values between 1e-3 to 5e-3.
- **Batch Size**: A fixed batch size of 200 was kept to make sure of adequate representation for all classes
- **Epochs**: Varied from 50 to 150.
- **Hidden Units:** Varied from 10 to 50
- **Dropout Rate**: Was kept 0.5

**Criteria for Selection**
The best hyperparameters were selected based on validation accuracy and loss, ensuring a balance between underfitting and overfitting.

---

## 5.4 Training Strategy

### Training Process

1. **Loss Function**: Cross-entropy loss, suitable for multiclass classification.
2. **Optimization Algorithms**:
    - **Adam**: Chosen for its adaptive learning rates and stability.
    - **SGD with Momentum**: Tested to improve generalization.
3. **Training Epochs**: Typically 50-150 epochs, with early stopping based on validation performance.
4. **Batch Size**: Typically 200 for efficient resource utilization.
5. **Learning Rate Schedule**: A scheduler to reduce the learning rate when the validation loss plateaus, facilitating better convergence.

### Challenges Encountered

- **Overfitting**: Addressed using dropout, data augmentation, and early stopping.
- **Vanishing Gradients**: Mitigated through ReLU/GeLU activation and careful weight initialization.

---

## 5.5 Implementation Details

### Software

- **Framework**: PyTorch was used for its flexibility, dynamic computation graph, and active community support.
- **Libraries**: NumPy, Matplotlib (for visualization), and Pandas (for dataset handling).

### Hardware

- **GPU**: A CPU (Intel i5-11320H) was used for accelerated training.
- **System**: A workstation instance with 16gb RAM and storage for the dataset and training logs.

## 6.1 Experimental Setup

### Environment

The experiments were conducted in a Python environment using PyTorch for deep learning. Key setup details include:

- **Hardware**: Training was performed on a high power CPU machine, such as an H series Intel CPU, to accelerate computation.
- **Software**:
  - Python 3.x
  - PyTorch for model implementation and training
  - Supporting libraries: NumPy, Pandas (data handling), Matplotlib (visualizations).
- **Platform**: Local workstation

### Specific Configurations

- **Training**: Batch size of 200, learning rate initialized at 1e-3 to 5e-3 with a decay schedule.
- **Evaluation**: Performed at the end of each epoch using the validation dataset to track accuracy and loss.
- **Reproducibility**: Random seeds were set for consistent results across runs.

---

## 6.2 Evaluation Metrics

### Metrics Used

1. **Accuracy**: Measures the proportion of correctly classified samples; suitable for balanced datasets.
2. **Precision, Recall, F1-Score**: Useful for evaluating performance on imbalanced datasets, where some classes may dominate others.
    - **Precision**: Indicates the percentage of true positives among all predicted positives.
    - **Recall**: Represents the percentage of true positives identified out of all actual positives.
    - **F1-Score**: Harmonic mean of precision and recall, balancing both metrics.
3. **Confusion Matrix**: Provides insights into classification performance for each class.

**Appropriateness**
These metrics provide a holistic evaluation of the model's performance, ensuring both overall accuracy and class-specific behavior are assessed.

---

## 6.3 Results

### Performance Summary

- **Baseline Model**: Achieved test accuracy of approximately 90.7% with, serving as a benchmark. (Adam Optimizer, 150 epochs, 10 hidden units and LR = 0.005)
- **Proposed CNN Model**: Achieved significant improvements with a peak test accuracy of 97.66% (Adam Optimizer, 150 epochs, 50 hidden units and LR = 0.001)

### Visualization

- **Accuracy and Loss Curves**:
    - Training and validation accuracy and loss were plotted over epochs, showing convergence trends.
    - Validation loss stabilized, indicating good generalization.
- **Tables**:
    - Results across different hyperparameter configurations were tabulated for comparison.
    - Key metrics (e.g., precision, recall) for each class were summarized.

**Comparison**
Experiments comparing:

1. **Different Optimizers**: Adam outperformed SGD in terms of convergence speed and final accuracy.
2. **Hyperparameter Settings**: More hidden units (50) led to better performance than more epochs.

---

## 6.4 Discussion

### Analysis of Results

- **Strengths**:
  - The CNN effectively captured spatial patterns in the dataset, resulting in high accuracy.
  - Data augmentation improved robustness and reduced overfitting.
- **Weaknesses**:
  - Slight underperformance on specific classes, likely due to insufficient representation in the training dataset.
  - Model struggled with noisy or ambiguous samples, as reflected in lower recall for certain classes.

### Error Analysis

- **Misclassifications**:
  - Errors occurred primarily in classes with visually similar characters.
  - Class imbalance contributed to poorer performance in underrepresented categories.
- **Unexpected Outcomes**:
  - Overfitting on training data in early epochs required tuning of regularization techniques and early stopping.

**Future Improvements**

- Enhance dataset quality with more balanced class distribution.
- Experiment with ensemble methods or pre-trained models (e.g., transfer learning) for improved feature extraction.
- Incorporate advanced architectures like residual or attention-based networks to address complex patterns.