Vision Transformer (ViT) - EEA Winter Project

Mayank Agarwal (240635)

Introduction

The project began with the basics of Python and its data manipulation libraries, which are heavily used in machine learning tasks and data processing. Important topics, such as Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs), were covered in detail to build the intuition necessary to understand the discovery and necessity of the Vision Transformer architecture.

1 Tools and Programming Basics & Assignment 1

This stage focused on the fundamentals of Python and data processing libraries essential for Machine Learning (ML) tasks. This stage is crucial, as ML workflows rely heavily on efficient data manipulation, visualization, and numerical computations. Without these optimized libraries, custom data processing implementations would be required, consuming significant time and lacking the efficiency of pre-implemented solutions.

1.1 Python Basics

Python is the leading language in ML workflows due to its simplicity and a mature ecosystem surrounding data science and specialized libraries like PyTorch and TensorFlow. Key programming concepts—including variables, data types, control structures, branching, looping, functions, array indexing, and I/O—were covered in detail to establish a foundation for future assignments.

1.2 Data Visualization with Matplotlib

Matplotlib was introduced for creating plots and charts, which are essential for analyzing ML model performance (e.g., loss curves, histograms). Core concepts include figure and axis objects, plotting functions (e.g., `plot()`, `scatter()`, `bar()`), customization (labels, titles, legends), and subplots for multi-view visualizations.

1.3 Data Manipulation with Pandas

Pandas was covered for handling structured data. Key topics included DataFrames and Series, reading/writing CSV files, indexing/slicing, handling missing values, grouping/aggregating data, and merging datasets. Concepts such as `apply()` for row-wise operations and pivot tables for summarization were emphasized. This library is critical for preprocessing in ML, including feature engineering and splitting data into training and test sets.

1.4 Numerical Computing with NumPy

NumPy forms the backbone for efficient array operations in ML. Important concepts include `ndarray` creation, broadcasting (automatic dimension alignment for operations), vectorization (avoiding loops for speed), linear algebra functions (e.g., dot product, matrix inversion), and random number generation. For example, array reshaping and slicing are key for preparing inputs for neural networks. NumPy leverages low-level access benefits, as it is implemented in C, allowing for highly optimized matrix operations.

2 Neural Networks Basics & Assignment 2

This week's resources focused on building the basics of Neural Networks. The mathematics behind neural networks were covered in detail to build a strong foundation.

This assignment utilized Week 1 concepts by implementing a Neural Network purely in NumPy. It taught basic low-level concepts to allow the building of a neural network from scratch, including layers, architecture, forward pass, backward pass, and gradient descent.

2.1 Model Architecture

- **Input (n_x):** Flattened MNIST data (784 features).
- **Hidden (n_h):** Dense layer with ReLU/Tanh activation for non-linearity.

- **Output (n_y):** Dense layer (10 units) with Softmax for class probabilities.

2.2 Mathematical Formulation

Forward Propagation: Computes predictions via linear transformations and non-linear activations.

$$Z^{[1]} = W^{[1]}X + b^{[1]} \to A^{[1]} = g(Z^{[1]}) \tag{1}$$

$$Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]} \to A^{[2]} = \text{softmax}(Z^{[2]}) \tag{2}$$

Cost Function: Uses Categorical Cross-Entropy to penalize confident incorrect predictions:

$$J = -\frac{1}{m}\sum Y \log(A^{[2]}) \tag{3}$$

Backward Propagation (Chain Rule): Derives gradients to minimize loss.

- Output Error: $dZ^{[2]} = A^{[2]} - Y$
- Layer 2 Gradients: $dW^{[2]} = \frac{1}{m}dZ^{[2]}(A^{[1]})^T, \quad db^{[2]} = \frac{1}{m}\sum dZ^{[2]}$
- Hidden Error: $dZ^{[1]} = (W^{[2]})^T dZ^{[2]} * g'(Z^{[1]})$
- Layer 1 Gradients: $dW^{[1]} = \frac{1}{m}dZ^{[1]}X^T, \quad db^{[1]} = \frac{1}{m}\sum dZ^{[1]}$

Gradient Descent: Updates parameters iteratively using learning rate α:

$$W := W - \alpha dW, \quad b := b - \alpha db \tag{4}$$

2.3 Metrics

We trained the model with a hidden layer of 1024 neurons, a learning rate of 0.1, and 1000 epochs. The model trained in approximately 5 minutes on a dataset of 8000 digits using a Google Colab GPU instance. The final test accuracy achieved on 2000 test images was **91.35%**. Below is the Loss vs. Epochs curve.

3 Convolutional Neural Networks and Modern Techniques

Building on programming basics, Week 2 dived into CNNs, a cornerstone of computer vision in machine learning. CNNs utilize the inherent structure of images and employ the concept of local patterns to make the model more efficient (fewer parameters) while increasing accuracy. This approach mirrors how humans perceive local structures and larger patterns composed of these smaller structures.

3.1 Fundamentals of CNNs

CNNs motivate the shift from fully connected layers to convolutions by emphasizing invariance (e.g., translation, rotation) and parameter efficiency. Core algorithms include:

- **Cross-Correlation Operation:** A kernel (filter) slides over the input image, computing dot products to produce feature maps. This detects local patterns like edges or textures.
- **Convolutional Layers:** Stacks of kernels learn hierarchical features—early layers capture edges, while later ones detect complex shapes. Padding and stride control output dimensions, preventing information loss at borders.
- **Multiple Channels:** Handles RGB images by applying kernels per channel, with 1×1 convolutions for dimensionality reduction.
- **Pooling Layers:** Max or average pooling reduces spatial size, enhancing translation invariance and computational efficiency. Stride and padding are again key parameters.

3.2 Modern CNN Enhancements: Batch Normalization

Batch Normalization (BatchNorm) is a pivotal technique for stabilizing deep network training. It normalizes activations within mini-batches using mean and variance, then scales/shifts with learnable parameters. This accelerates convergence, acts as a regularizer (reducing overfitting), and allows for higher learning rates.

In CNNs, BatchNorm is applied after convolutional layers but before activations. Implementation involves computing batch statistics during training and running averages for inference. It essentially converts each feature to a normal distribution (approximately). This allows subsequent layers to learn independently of the mean and variance of the inputs, as layers receive normalized inputs. This provides stable convergence and mitigates the diminishing gradient issues common in deep Neural Networks.

4 Recurrent Neural Networks

Week 3 introduced RNNs for sequential data, where order matters (e.g., text, time series). RNNs extend feedforward networks with recurrent connections to capture temporal dependencies, making them powerful for NLP and forecasting.

4.1 Core RNN Concepts

RNNs handle variable-length sequences using unrolling: the same parameters are shared across time steps, creating a chain-like structure. Key ideas include hidden states (memory of past inputs) and autoregressive modeling (outputs depend on prior elements).

Subsections covered:

- **Sequence Handling:** Tokenization, vocabulary building, encoding, etc.
- **RNN Architecture:** With hidden states vs. without; character-level models for text generation.
- **Backpropagation Through Time (BPTT):** Unrolls the network for gradient computation.

RNNs revolutionized sequence tasks in the 2010s but face stability issues, leading to modern variants. Stability issues arise from RNNs ”forgetting” old facts/knowledge. As BPTT can span several hundred timesteps, gradients vanish for older timesteps. This implies that information captured at earlier timesteps is not retained well, causing the model to yield incorrect results on tasks requiring long-range context.

LSTMs, GRUs, and similar architectures were introduced to allow older information to pass without vanishing, preserving long-range context. This concept is similar to how skip connections work. The concept of ”gates” was covered in detail while teaching these architectures.

5 Modern Recurrent Neural Networks

Week 4 advanced RNNs with gating mechanisms to mitigate gradient problems, culminating in mid-term evaluation readiness. These models are fleshed out as they enable practical applications like machine translation.

5.1 LSTM and GRU

- **LSTM (Long Short-Term Memory):** Uses memory cells with input, forget, and output gates to control information flow. This preserves long-range dependencies, avoiding vanishing gradients.
- **GRU (Gated Recurrent Unit):** Simpler than LSTM, with reset and update gates for candidate hidden states. It is computationally lighter while maintaining performance.

Both include implementations, deep stacking, and bidirectional variants (using past/future contexts, e.g., for sequence labeling).

6 Assignment 3: Theoretical Foundations of Deep Learning

This assignment serves as a theoretical checkpoint, moving beyond implementation to test the conceptual understanding of Deep Learning mechanics. Unlike previous coding assignments, which aimed at building the basics of python, the data science libraries, and working of Neural networks, this module focuses on the mathematical justification behind optimization algorithms, model evaluation strategies, and techniques to ensure training stability and generalization.

Concepts covered included optimizer types, gradient clipping, regularization techniques, dropout, evaluation metrics, and gradient instabilities along with mitigation strategies.

6.1 Advanced Optimizers

- **Momentum:** Accelerates SGD by accumulating past gradients (”velocity”) to navigate ravines.
- **RMSProp:** Adapts learning rates by dividing by the moving average of squared gradients.
- **Adam:** Combines Momentum and RMSProp, adapting learning rates for each parameter individually.

6.2 Evaluation Metrics

Moving beyond simple accuracy, we evaluated classification performance comprehensively using a **Confusion Matrix**.

Key Formulas:

- **Precision:** Accuracy of positive predictions.

$$\text{Precision} = \frac{TP}{TP + FP}$$

- **Recall:** Ability to find all positive instances.

$$\text{Recall} = \frac{TP}{TP + FN}$$

- **F1 Score:** Harmonic mean, crucial for imbalanced datasets.

$$F1 = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

7 Assignment 4: CNNs & Batch Normalization (CIFAR-10)

This assignment focuses on training a Deep CNN on the CIFAR-10 dataset. The primary objective is to implement and analyze Batch Normalization (BN), a technique essential for stabilizing training and improving convergence.

This is the model architecture we used to solve this assignment:

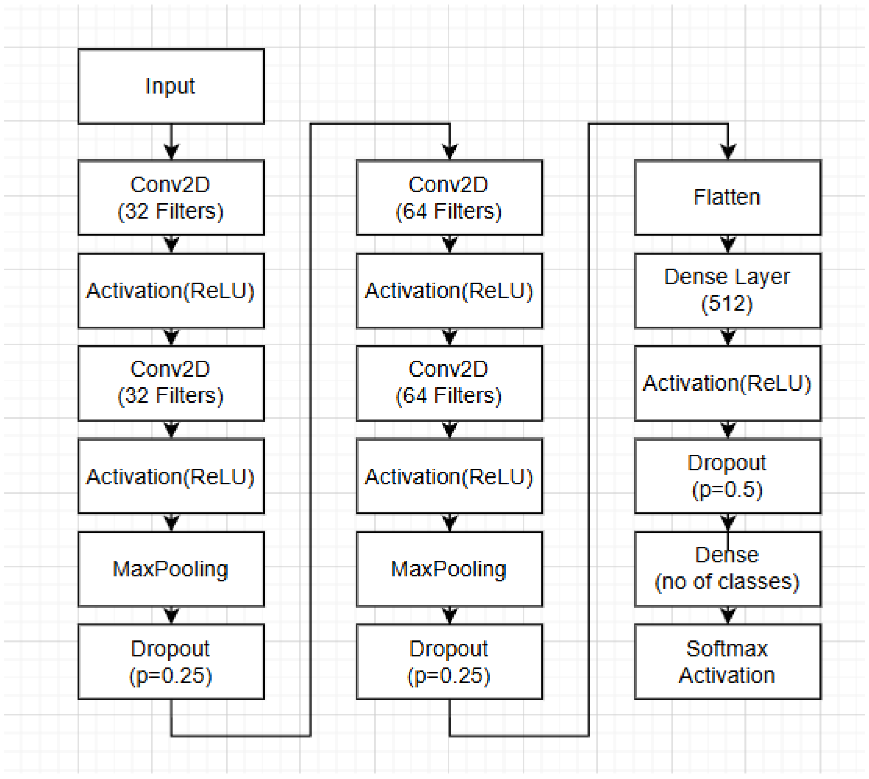

Figure 1: Model Architecture with Batch Normalization

Batch normalization helped improve the convergence of the model, as accuracy increased in the **improved model (80%) compared to the original model (75%).**

8 Assignment 5: Character-Level Language Modeling (RNN)

This assignment focuses on sequence generation using a Recurrent Neural Network (RNN) from scratch. The specific task is to build a Character-Level Language Model trained on a dataset of dinosaur names. The model learns the probability distribution of the next character given a sequence of previous characters, allowing it to synthesize new, unique dinosaur names.

The process involves computing the output probability distribution using softmax on the output logits. We randomly sample the next character index based on this distribution to add diversity to the generated names. The sampled character is then fed as the input for the next time step. This repeats until a newline character or max length is reached.

By the end of training, the model transitions from generating random strings to producing coherent, phonetically plausible dinosaur names (e.g., ”Macallosaurus”), demonstrating the RNN's ability to learn sequential patterns and structure.

Conclusion

The materials cover a logical progression: Python tools (Week 0), CNNs for spatial data (Week 2), basic RNNs (Week 3), and advanced sequential models (Week 4) by mid-term. Future extensions will include transformer architecture, and building on these concepts to make the final Vision Transformer Architecture.