ViT Mideval Report

Rahul Ranjan
Roll No: 240832

Neural Network Fundamentals

Forward Propagation

We learnt to implement the linear and non-linear transformations:

$$Z_1 = W_1 \cdot X + b_1 \quad \text{(Linear Step)} \tag{1}$$
$$A_1 = \tanh(Z_1) \quad \text{(Activation)} \tag{2}$$
$$Z_2 = W_2 \cdot A_1 + b_2 \quad \text{(Linear Step)} \tag{3}$$
$$A_2 = \text{softmax}(Z_2) \quad \text{(Output Probabilities)} \tag{4}$$

Softmax is an activation function that converts a vector of raw numbers (logits) into a probability distribution. It forces all output values to be between 0 and 1, and ensures they sum up to 1.

$$\text{Softmax}(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}} \tag{5}$$

The Cost Function (Categorical Cross-Entropy)

For multi-class classification, Mean Squared Error (MSE) is suboptimal because it doesn't punish "confident but wrong" predictions heavily enough. We used **Cross-Entropy Loss**:

$$L = -\frac{1}{m} \sum y_{true} \cdot \log(y_{pred}) \tag{6}$$

This function grows exponentially as the predicted probability for the correct class approaches zero, forcing the model to learn quickly.

Backpropagation (The Calculus)

- **Output Error:** $dZ_2 = A_2 - Y$ (Prediction - Truth)
- **Gradients for Layer 2:**

$$dW_2 = \frac{1}{m} dZ_2 \cdot A_1^T \tag{7}$$
$$db_2 = \text{mean}(dZ_2) \tag{8}$$

- **Backpropagating through Activation:** To find the error at the hidden layer (dZ_1), we multiply by the derivative of the Tanh function: $(1 - A_1^2)$.

$$dZ_1 = W_2^T \cdot dZ_2 \times (1 - A_1^2) \tag{9}$$

- **Gradients for Layer 1:**

$$dW_1 = \frac{1}{m} dZ_1 \cdot X^T \tag{10}$$

Training Stability & Weight Initialization

- **Symmetry Breaking:** If all weights start at 0, every neuron calculates the same output and the same gradient. The network essentially acts as a single neuron, never learning complex features.

Evaluation Metrics for Imbalanced Data

- **Confusion Matrix:** A table of evaluate performance of classification model revealing TP, TN , FP and FN.
- **Precision:** TP / (TP + FP)
- **Recall :** TP/ (TP + FN)
- **F1 Score :** 2 Precision.Recall / (Precision + Recall)

Regularization Techniques

To prevent **Overfitting** (High Variance):

- **L1 vs. L2 Regularization:**
  - **L1 (Lasso):** Drives weights to exactly zero. Useful for feature selection/compression.
  - **L2 (Ridge):** Penalizes large weights. Used for general "Weight Decay."
- **Dropout:**
- *Concept:* Randomly zeroing out neurons during training forces the network to be redundant.
- *Scaling:* If we keep 50% of neurons, we must multiply the activations by 2 (or 1/0.5) to maintain the expected sum magnitude.

Advanced Optimizers

- **Momentum:** Adds a "velocity" term to the update, helping the optimizer plow through local minima and small bumps.
- **RMSProp:** Solves the problem of "Exploding Gradients" by dividing the learning rate by the moving average of squared gradients (normalizing the step size).
- **Adam:** It efficiantly updates weights by combining both Momentum and RMSProp.

Convolutional Neural Networks (CNNs)

Convolution & Pooling

- **Convolution:** It applies filters to the input image to extract features. A filter is swept across the image during convolutional process.
- **Pooling (Max/Average):** Pooling downsamples the image, reducing computation and making the model robust to small shifts in the input.

Batch Normalization

We implemented a custom `BatchNormalizedLayer` to solve **Internal Covariate Shift**.

- **Mechanism:** For every mini-batch, the layer calculates μ and σ^2 and normalizes inputs to $\mathcal{N}(0, 1)$.
- **The Learnable Scale/Shift:** Why we need parameters γ and β: they allow the network to "undo" the normalization if a different distribution (e.g., non-zero mean) is better for the next layer.
- **Bias term (b)** in the preceding Dense/Conv layer is unnecessary when using Batch Norm, as the normalization step (subtracting mean) would cancel it out anyway.

Recurrent Neural Networks (RNNs)

The Sequence Paradigm

Unlike CNNs/MLPs which treat inputs as independent snapshots (IID), RNNs maintain a **Hidden State** (h_t) that acts as a memory of previous inputs.

$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t) \tag{11}$$

The Vanishing Gradient Problem

When Backpropagating Through Time (BPTT), gradients are multiplied by the weight matrix W repeatedly ($W \cdot W \cdot W \dots$) (mathematical flaw in standard RNNs).

- If weights are small, gradients vanish (network forgets early inputs).
- If weights are large, gradients explode (network becomes unstable).
- **Solution Implemented: Gradient Clipping** (`np.clip`) caps gradients between -5 and 5, preventing explosion.

Modern RNN: LSTM & GRU

To solve the "forgetting" problem, we use Gated Architectures.

- **LSTM (Long Short-Term Memory):**They introduce a **memory cell** that can maintain information over long durations, controlled by mechanisms called "gates". **The Three Gates of LSTM:**
  - **Forget Gate:** Decides what information from the past is no longer relevant and should be thrown away. It looks at the previous state and current input to output a value between 0 (forget completely) and 1 (keep completely).
  - **Input Gate:** Decides what *new* information is important enough to store in the long-term memory. It uses a sigmoid function to decide *what* to update and a tanh function to create a candidate of *new values* to add.
  - **Output Gate:** Decides what part of the internal memory should be output as the "hidden state" (short-term memory) for the current time step. This output influences the prediction for the immediate moment.

GRU (Gated Recurrent Unit): The GRU is a simplified version of the LSTM.

- **Simpler Structure:** It combines the forget and input gates into a single **Update Gate** and merges the cell state and hidden state.
- **Two Main Gates:**
  1. **Update Gate:** Controls how much past information to keep.
  2. **Reset Gate:** Controls how much past information to ignore.
- **Benefit:** GRUs often achieve similar performance to LSTMs but are faster to train because they have fewer parameters.

Advanced Architectures

- **Deep RNNs:** Formed by stacking multiple layers of RNNs, LSTMs, or GRUs on top of each other.
  - *Lower layers* capture simple patterns (local features).
  - *Higher layers* capture abstract concepts (long-term features).
- **Bidirectional RNNs (BiRNN):** Instead of just reading left-to-right, this model processes the sequence in **both directions** (forward and backward) simultaneously.
  - *Use Case:* Essential for NLP tasks like filling in a missing word, where the answer depends on both the words *before* and *after* the blank.+1
  - *Limitation:* Cannot be used for real-time prediction (like live speech) because you need the future context.
- **Encoder-Decoder Architecture:** Used for mapping one sequence to another of a different length, such as **Language Translation**.
  - **Encoder:** Reads the input sentence and compresses it into a "context".
  - **Decoder:** Generates the translated output step-by-step from that context.