Vision Transformer
Mid-eval Report

Azaad Katiyar (240249)

January 10, 2026

1 Introduction

In this project, we aim to understand and implement Vision Transformers (ViTs) for image classification. The work starts with basic machine learning and deep learning concepts and then moves toward attention-based models used in modern computer vision.
Images are first handled using traditional neural networks and convolutional neural networks (CNNs). Later, transformer concepts such as self-attention are applied to images by dividing them into patches. The project uses the CIFAR-10 dataset to train and compare CNN and ViT models.

2 WEEK-0

2.1 Python Foundations

Before diving into the foundations of ML and deep learning, we had a class in which we revised basic python, matplotlib, pandas and numpy. In this lecture, we brushed upon topics like loops, functions, lists, dictionaries, classes, creating plots using matplotlib plotting package, manipulating data, dataframes, datatypes like arrays, indexing, broadcasting, and many such topics.

2.2 Assignment 1: Foundations of Python, NumPy, Pandas and Matplotlib

The objectives of this assignment were to demonstrate proficiency in programming in Python by creating a custom data handling class, Array manipulation, indexing, and broadcasting using the NumPy library, Dataframe handling, splitting, and visualization using Pandas and Matplotlib. The assignment was structured into three distinct parts, each targeting a core library.

3 WEEK-1

3.1 Activations and Loss

To introduce non linearity into our model, we use **Activation functions** which are applied to the output of a neuron in the model. Without them, our model will only be able to learn linear patterns.
Some common examples of Activation functions:
1. Sigmoid: $\mathbf{f(z)} = \frac{\mathbf{1}}{\mathbf{1+e^{-z}}}$ It squeezes any real value into a range between 0 and 1.
2. Tanh: $\mathbf{f(z)} = \frac{\mathbf{e^{z}-e^{-z}}}{\mathbf{e^{z}+e^{-z}}}$ Slightly different from sigmoid, it squeezes any real value between -1 and 1.
3. ReLU: $\mathbf{f(z)} = \mathbf{max(0, z)}$ Most commonly used in deep learning, helps prevent vanishing gradients.
4. Softmax: $\mathbf{softmax(x_i)} = \frac{\mathbf{e^{x_i}}}{\mathbf{\sum e^{x_j}}}$ Turns a vector of numbers into a probability distribution, sums to 1.

A **Loss function** is used to quantify the difference between actual value and predicted value.
Regression Losses:
1. Mean absolute error (MAE): $\mathbf{L} = \frac{\mathbf{1}}{\mathbf{n}} \sum \mathbf{|y - \hat{y}|}$ Uniform gradient; Gradient DNE at 0.
2. Mean squared error (MSE): $\mathbf{L} = \frac{\mathbf{1}}{\mathbf{n}} \sum \mathbf{|y - \hat{y}|^2}$ Fast gradient; Gradient exists everywhere.

Classification Losses:
1. Binary Cross Entropy (BCE): $\mathbf{L} = -\mathbf{[y log(p) + (1 - y) log(1 - p)]}$ For Binary classification
2. Categorical Cross Entropy (CCE): $\mathbf{L} = -\sum \mathbf{y_i log(p_i)}$ For multi-class classification

3.2 Linear Regression, Logistic Regression and Gradient Descent

Regression simply means modeling a relationship.
Linear Regression: It models a straight-line relationship between input x and output y. It assumes that output can

be expressed as $y = mx + c$. We try to find the best possible m and c which give minimum error.
Logistic Regression: It is used when the output is a class or probability instead of a continuous number. The first step is to find $z = mx + c$, same as linear regression but now we pass this z into the sigmoid function which squeezes it between 0 and 1.
Gradient Descent: This is an optimization method used to reduce the loss function by gradually changing the model's parameters.It first begins with some initialized value given by us, evaluates the loss, and then adjusts the parameters such that loss decreases most rapidly.
Gradient descent update rule: $\mathbf{m_{new}} = \mathbf{m_{old}} - \eta \frac{\delta \mathbf{L}}{\delta \mathbf{m}}$
We usually stop gradient descent when either change is error becomes very small or parameter update become negligible or max number of iterations is reached.

3.3 K-Means Clustering

K-Means Clustering is an unsupervised machine learning algorithm that partitions a dataset into K distinct and non overlapping groups called clusters based on their similarities.
Algorithm:
1. K random points of data set are chosen to be centroids.
2. Distances between all data points and the K chosen centroids are calculated.
3. Based on the distances, each point is allotted to the nearest cluster.
4. New cluster centroids are updated by finding a mean of all the points of the cluster.
5. This process is repeated until the calculated new centroid remains the same.

3.4 Some Theoretical Concepts

3.4.1 Evaluation Metrics

Model evaluation metrics are critical for assessing the performance of machine learning models, particularly in classification tasks. Some of the most commonly used metrics are:
1. $Accuracy = \frac{TP+TN}{TP+TN+FP+FN}$
Percentage of correctly classified samples out of the total number of samples.
2. $Precision = \frac{TP}{TP+FP}$
Ratio of truly positive cases to the total predicted positive cases.
3. $Recall = \frac{TP}{TP+FN}$
Ratio of truly positive cases identified by the model correctly to the total positive cases present.
4. F1 Score $= 2 * \frac{Precision*Recall}{Precision+Recall}$
Harmonic mean of the precision and recall.

3.4.2 Regularization

Overfitting is a problem that we encounter when a model learns the training data too well, in a way it memorizes the data, rather than finding patterns in it, and then performs poorly on new data. In regularization, we add a penalty term to the cost function which forces the algorithm to reduce the value of some of the parameters to nearly 0, nullifying the effect of those related neurons.
L1 & L2 regularization: In L1, we add the sum of the absolute values of weights to the cost function. This pushes some weights exactly to 0 value.
In L2, we add the sum of the squared weights to the cost function. This reduces the value of some weights to near 0 but not exactly 0.
$Cost_{L1} = \frac{1}{2m}\sum_{i=1}^{m}(\text{Loss}) + \lambda \sum_{j=1}^{n} |w_j|$; $Cost_{L2} = \frac{1}{2m}\sum_{i=1}^{m}(\text{Loss}) + \lambda \sum_{j=1}^{n} |w_j|^2$
Dropout Layer: Dropout is used specifically during the training phase. During each training iteration, the layer randomly "drops out" (sets to zero) a percentage of neurons. This prevents neurons from becoming overly dependent on each other (co-adaptation). If a neuron can't rely on its neighbor being there, it must learn more robust, independent features.

3.4.3 Gradient Descent Methods

1. Batch Gradient Descent
Uses entire training dataset to compute gradients, stable and accurate updates, slow and memory-intensive for large datasets.
2. Stochastic Gradient Descent (SGD)
Updates parameters using one training example at a time, faster and more noisy, helps escape local minima, fluctuates a lot.
3. Mini-Batch Gradient Descent
Uses small batches of data points, is the best of both worlds, stable and efficient, most commonly used.

3.4.4 Assignment 2: Neural Network for Digit Recognition from Scratch

The main objective of this assignment was to build, train, and evaluate a two-layer neural network from scratch to perform digit classification on the MNIST dataset, using only NumPy for the core computations. TensorFlow/Keras was used for loading the MNIST dataset.
Data Preprocessing and Network Implementation: The MNIST dataset was preprocessed by downsampling to 1,000 images per digit, resulting in 10,000 training images and 9,786 test images. Each 28×28 image was flattened into a 784×1 vector, pixel values were normalized to the [0,1] range, and labels were one-hot encoded. A neural network was implemented from scratch using modular functions. ReLU and softmax activation functions (with derivatives) were defined, parameters were initialized with small random weights and zero biases, and forward propagation computed outputs and cross-entropy loss. Backpropagation calculated gradients, and gradient descent updated parameters.
Results and Performance: Trained for 2,000 iterations with a learning rate of 0.2, the model showed a smooth decrease in cost. It achieved 98.16% training accuracy and 93.71% test accuracy, validating the correctness and effectiveness of the implementation.

4 WEEK-2

4.1 Convolutional Neural Networks (CNN)

4.1.1 Why not multi-layer perceptron ?

Processing a 1000 x 1000 image with a fully connected layer requires 1 trillion parameters which is impossible to train. CNNs solve this problem through: 1. Translation Invariance: Same filters work at every position (this reduces parameters to 1 million) 2. Locality: Only nearby pixels matter (further reduces to Δ^2).

4.1.2 What is Convolution Operation ?

Convolution is a mathematical operation where a small filter (kernel) slides across an image, computing element-wise multiplications and summing results. $H_{i,j} = \sum_{k,l} W_{i,j,k,l}\, X_{k,l} + b_{i,j}$ forms the foundation of convolutional neural networks. It is actually called cross-correlation operator. In CNNs, convolution is used to extract features. Kernels slide over the image to detect patterns such as edges and textures.

4.1.3 Terminologies in CNN

1. Feature Map: Spatial output of a convolutional layer.
2. Receptive field: Region of input affecting a neuron's activation.
3. Padding: Without padding, every convolution shrinks the image, it is used to preserve spatial size and preserve information at the borders.
4. Stride: Step size with which the convolution kernel moves across the input.
5. Pooling: Reduces the spatial size of feature maps while keeping the most important information.
Max Pooling → takes the maximum value
Average Pooling → takes the average value.

4.1.4 Core CNN Operations

Conv2D: Applies learned filters to extract features from images
MaxPooling2D: Reduces spatial dimensions using max operation (2x2 pooling reduces image to 1/4 size)
Flatten: Converts 3D tensor (height x width x channels) to 1D vector
Dense: Fully connected layer for classification tasks
ReLU: Non-linear activation function (max(0, x)) for learning complex patterns

4.2 Batch Normalization

Batch Normalization normalizes activations to have zero mean and unit variance.
Formula: $\mathbf{BN}(\mathbf{x}) = \gamma * \frac{\mathbf{x-mean}}{\sqrt{(\mathbf{variance}+\epsilon)}} + \beta$, where γ is the learnable scale and β is shift.
This improves training stability and allows higher learning rates, faster training, acts as regularizer and reduces overfitting.

4.3 Assignment 3: Core Theoretical Concepts in Machine learning

The objective of this assignment was to demonstrate a theoretical understanding of fundamental machine learning concepts by providing detailed written answers to some questions.
Evaluation Metrics: The definition and calculation of key classification metrics were reviewed. For the provided example, the results came out to be: Accuracy (90%), Precision (50%), Recall (80%), and F1-score (61.5%).
Overfitting and Underfitting: The critical trade-offs between model complexity and the amount of training data.
Vanishing and Exploding Gradients: The causes of these common deep learning training problems were detailed, along with prevention strategies like using ReLU activations and proper weight initialization.

Dropout Layers: The concept of dropout was explained as a regularization method where a random fraction of neurons are temporarily deactivated during each training step to prevent memorization of data.
Gradient Descent Variants: The key differences between Batch, Stochastic, and Mini-Batch Gradient Descent were studied, focusing on the trade-offs between computational efficiency and stability.
Optimizers: The role of advanced optimization algorithms like Momentum, RMSprop, and Adam was explained as a means to accelerate and stabilize the training process.

5 WEEK-3

5.1 Recurrent Neural Networks (RNN)

5.1.1 Importance of Sequences

Until now, all the models we have covered assume data points to be independent. But in sequences, words in text depend on previous words for example, stock prices depend on historical prices.
For this purpose, we use RNNs which maintain a hidden state that carries information from past to future.

5.1.2 Auto-Regressive Models

An auto-regressive model tries to predict the next value using past values: $\mathrm{P}(x_t|x_{t-1}, ..., x_1)$
But the problem in it is that this can be memory intensive, so we simplify using T^{th} order Markov assumption (only last T observations matter): $\mathrm{P}(x_t|x_{t-1}, ..., x_{t-T})$

5.1.3 Latent Autoregressive Models

Instead of keeping all past data, the Latent Autoregressive model keeps a hidden state which is a small summary of everything the model has seen so far. It gets updated at every step.

$$h_t = g(h_{t-1}, x_{t-1}) \text{ and } \hat{x_t} = P(x_t, h_t)$$

5.1.4 Language Models & Tokenization

Tokenization: Since computers cannot understand raw text directly, so we first break text into small pieces which we call tokens. Whenever we encounter a token which is not previously seen, it is represented by a ¡unk¿ token signifying unknown value.
Language Model: A language model estimates the joint probability of entire sequence: $\mathrm{P}(x_1, x_2, ..., x_n)$
The model predicts how likely the next word is, given the previous words and doing this for all words gives the probability of the entire sentence.
Perplexity(Quality metric) tells us how confused the model is while predicting words.
$Perplexity = \exp(-\frac{1}{n} * \sum logP(x_t|x_{t-1}, ..., x_1))$
Lower perplexity means model is confident which gives good predictions.

5.1.5 RNN Architecture & Equations

Recurrent neural networks (RNNs) are neural networks with hidden states. Hidden state is the memory of the RNN which stores information from previous time steps and gets updated every time a new input comes in.
New memory = function(current input + old memory) To update memory, the RNN multiplies the input and previous memory with learned weights, adds them together, and then passes the result through tanh, keeping values between -1 and 1.
RNN Formula:

$$\text{Hidden State:} \quad H_t = \tanh\left(X_t W_{xh} + H_{t-1} W_{hh} + b_h\right)$$
$$\text{Output:} \quad O_t = H_t W_{hq} + b_q$$

where X_t = input at time t, h_{t-1} = previous hidden state, W = weight matrices.
Input Representation (One Hot Encoding): Before giving words to the RNN, we convert them into numbers. Each word is represented as a vector. Vector length = size of vocabulary. Only one position is 1, all others are 0.

5.1.6 Backpropagation Through Time (BPTT)

Unroll RNN across T time steps, each time step looks like a new layer, so an RNN over T time steps becomes a deep network with T layers
Chain rule: $\frac{dL}{dW} = \frac{1}{T}\sum(\frac{dL}{dO_t} * \frac{dO_t}{dh_t} * \frac{dh_t}{dW})$ for all t
We can compute the full gradients through chain rule, but this is very slow and gradients can blow up, so we just calculate the sum upto T steps i.e truncate the time steps.

5.1.7 Gradient Clipping

During training, when we send the error backwards, gradients get multiplied again and again which can make gradients either too large - exploding gradients or too small - vanishing gradients.
Gradient Clipping: Instead of changing the learning rate, we can control the gradient size. The idea is not to let gradient become too large.
Compute the gradient vector g and measure its size (norm)
If gradient size is less than θ then do nothing but if gradient size is greater than θ then shrink it to size θ.

$$\mathbf{g} \leftarrow \min\left(1,\ \frac{\theta}{\|\mathbf{g}\|}\right)\mathbf{g}$$

5.1.8 Assignment 4: Implementing Batch Normalization in a CNN

The objectives for this assignment were to understand the theory and working behind Batch Normalization, to implement a baseline CNN for image classification on the CIFAR10 dataset, and to enhance the CNN by integrating Batch Normalization layers and quantitatively comparing the performance of the baseline model against the improved model. Two models were built and trained to facilitate a direct comparison.
CNN Models and Results: A baseline CNN was built with two convolutional blocks. The first block used two Conv2D layers with 32 filters, ReLU activations, MaxPooling, and Dropout (0.25). The second block mirrored this structure with 64 filters. The classifier consisted of Flatten, a Dense layer with 512 units and ReLU, Dropout (0.5), and a Softmax output for 10 classes. An enhanced CNN used the same architecture but added Batch Normalization after each convolutional and dense linear layer, before ReLU.
Comparative Analysis: Trained for 25 epochs on CIFAR-10, the baseline achieved 75.77% test accuracy, while the Batch Normalization model reached 79.32%. Batch Normalization improved convergence, smoothed loss curves, and increased accuracy by 3.55%.

6 WEEK-4

In this week, we covered modern RNNs like LSTM, GRU, Bidirectional RNN.

6.1 Long-Short Term Memory (LSTM)

Problems faced by RNNs which are solved by LSTMs:
1) Forgetting information quickly ; 2) Struggling to learn long-term dependencies ; 3) Vanishing gradients

In LSTM, 2 states are maintained - hidden state (short-term memory) and cell state (long-term memory) controlled by gates. LSTM uses gates to control information flow. Each gate outputs values between 0 and 1.
Forget Gate: It decides how much past information should be retained in the memory cell.
$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] + b_f\right)$ (0=forget, 1=keep)
Input Gate: Decides what new information should be stored.
$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] + b_i\right)$
$\tilde{c}_t = \tanh\left(W_c \cdot [h_{t-1}, x_t] + b_c\right)$
Cell Update: $c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t$ combines old and new information
Output Gate: Decides what part of memory becomes the output
$o_t = \sigma\left(W_o \cdot [h_{t-1}, x_t] + b_o\right)$
$h_t = o_t \odot \tanh(c_t)$

6.2 Gated Recurrent Unit (GRU)

GRU is a simpler version of LSTM. LSTM has many gates and two memories. GRU makes things simpler and faster by using fewer gates and only one memory (hidden state). Instead of three gates (like LSTM), GRU uses only two:
Reset Gate: Decides how much of the past information is useful. If reset gate is small → ignore old information ; if it is large → keep past information.
Update Gate: Decides how much old memory to keep and how much new information to add. It acts like a mixer between old and new memory.

$$\begin{aligned}
R_t &= \sigma\left(X_t W_{xr} + H_{t-1} W_{hr} + b_r\right)\\
Z_t &= \sigma\left(X_t W_{xz} + H_{t-1} W_{hz} + b_z\right)\\
\tilde{H}_t &= \tanh\left(X_t W_{xh} + (R_t \odot H_{t-1}) W_{hh} + b_h\right)\\
H_t &= Z_t \odot H_{t-1} + (1 - Z_t) \odot \tilde{H}_t
\end{aligned}$$

6.3 Bi-directional RNN

A Bidirectional RNN reads a sequence in both directions:
Forward → left to right (which means past → future)
Backward → right to left (means future → past)
How it works ?
At each position in the sequence, one RNN processes information from the past and another RNN processes information from the future. Their outputs are combined which gives a richer understanding of the word.
Why this is useful ?
Where normal RNNs only know the past, Bidirectional RNNs know both past context and future context. So predictions are more accurate.
Limitation of Bidirectional RNN ? They need the entire sentence before processing and cannot be used when data arrives step by step. So they are not suitable for real-time tasks like: Real-time translation

6.4 Encoder-Decoder (Seq2Seq) Architecture

In many tasks, input and output lengths are different where a normal RNN struggles, like english to french translation. Seq2Seq uses two models:
Encoder — This understands the input. It reads the entire input sequence, one step at a time and converts it into a single fixed-size summary, called the context vector.
Decoder — This generates the output. It takes the context vector and generates the output sequence word by word.

6.5 Assignment 5: Character Level RNN for Text Generation

The objective of this assignment was to implement a character level RNN language model from its core components to generate new dinosaur names based on patterns learned from a provided text file "dinos.txt".
RNN Implementation and Training: Data preparation involved reading (dinos.txt) file, building a 27-character vocabulary (a–z and newline), and creating character–index mapping dictionaries for text processing. Two core components were implemented: gradient clipping to restrict gradients within [5, 5] and prevent exploding gradients, and a sampling function to generate text by iteratively sampling characters from the softmax output.
Training was performed using stochastic gradient descent for 35,000 iterations, processing one name per step. Sampled outputs were generated every 2,000 iterations to track learning. Initially, outputs were random strings, but over time the model produced increasingly structured and realistic dinosaur-like names, demonstrating successful learning of sequential patterns and validating the RNN's generative capability.