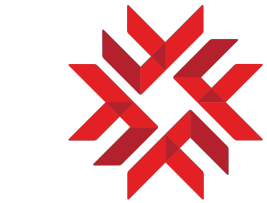
Vision Transformers (VIT)

Course:
Deep Learning with Tensorflow & Kersa 2



Developed by:
Mohammad Noorchenarboo

November 17, 2025

Current Section

Motivating Problem: Capturing Global Context

How do CNNs struggle to model long-range dependencies and dynamic relationships in images?

While CNNs excel at local feature extraction via convolutional kernels, they face key challenges:

1. **Limited Receptive Field**: Each neuron "sees" only a small neighborhood.
2. **Fixed, Static Weights**: Convolutional filters are learned once and applied uniformly.
3. **Information Bottlenecks**: Pooling layers reduce spatial resolution, potentially losing fine details.

Motivating Problem: Capturing Global Context

Receptive Field Formula

In a CNN, the receptive field refers to the region of the input image that a particular neuron in a given layer "sees" or is influenced by. For a stack of L convolutional layers with kernel size k and stride 1, the theoretical receptive field R_L is:

$$R_L = k + (k-1)(L-1).$$

E.g. with $k = 3$ and $L = 5$:

$$R_5 = 3 + 2 \times 4 = 11 \text{ pixels}$$

Tunnel Vision Analogy

A CNN's receptive field is like looking through a narrow tunnel–each step broadens the view slightly, but distant landmarks remain out of sight until many layers later.

Additional Weaknesses of CNNs

Beyond locality, CNNs exhibit:

Static Filter Pitfall

Filters learned during training remain fixed at inference–unable to adapt focus dynamically to different object relationships.

Pooling Caveat

Spatial pooling (max/average) compresses information:

$$\text{Downsample factor} = \frac{\text{input size}}{\text{output size}}$$

Excessive pooling may discard critical fine-grained details.

Summary of CNN Limitations

Limitation	Implication
Limited Receptive Field	Neurons can only integrate information from a small neighborhood; requires many layers to see global structure.
Static Filters	Cannot dynamically reweight features based on context; inflexible to varying object relationships.
Pooling Information Loss	Reduces spatial resolution, potentially discarding critical details for fine classification/localization.
Parameter Growth	Increasing receptive field via depth/dilation inflates parameter count and computational cost.

Current Section

What Does the Encoder Do?

Motivating Question: How does an encoder-only model like BERT "understand" text without generating anything?

Encoder – Definition and Role

An encoder ingests the entire input sequence at once and applies **bidirectional self-attention** to produce contextual embeddings $\mathbf{H} = [\mathbf{h}_1, \ldots, \mathbf{h}_n]$, where each $\mathbf{h}_i$ attends to all tokens in the sequence.

Encoder as a Group Discussion

Imagine a group of experts (tokens) all talking to each other simultaneously to form a shared understanding of a document; the encoder's bidirectional attention is like that discussion.

What Does the Encoder Do?

Real-World Example: Search Ranking

In semantic search, both query and document are encoded:

$$\mathbf{q} = \text{Encoder}(\text{query}), \quad \mathbf{d} = \text{Encoder}(\text{document})$$

and ranked by similarity, e.g. $\cos(\mathbf{q}, \mathbf{d})$.

Caveat

Because the encoder sees all tokens at once, it has no mechanism to predict "what comes next," and thus cannot generate new text.

What Does the Decoder Do?

Motivating Question: How does a decoder-only model like GPT generate fluent text based on a prompt?

Decoder – Definition and Role

A decoder generates text autoregressively using **causal self-attention**: at step t, it computes

$$P(y_t \mid y_{<t}, \mathbf{X})$$

by attending only to previous tokens $y_{<t}$ (and optional input $\mathbf{X}$).

Decoder as a Storyteller

Picture a storyteller who composes a tale one sentence at a time, each new sentence influenced by what has already been said.

What Does the Decoder Do?

Real-World Example: Text Generation

Given a prompt “Once upon a time,” the decoder produces the next token distribution and appends tokens one by one to form a story.

Caveat

Because attention is strictly left-to-right, the decoder cannot revise earlier tokens based on later context.

Encoder vs. Decoder: Task Alignment

Motivating Question: Which tasks align naturally with encoders versus decoders?

Encoder-Only Tasks

- Text classification (e.g. sentiment analysis)
- Semantic search and retrieval
- Token-level labeling (e.g. Part-of-Speech tagging)

Decoder-Only Tasks

- Free-form text generation (stories, code)
- Auto-completion and dialogue response
- Autoregressive translation (prompt-to-text)

Summary Table: Encoder vs. Decoder

Aspect	**Encoder** vs. **Decoder**
Core Mechanism	Encoder: Bidirectional self-attention; Decoder: Causal (left-to-right) self-attention
Main Function	Encoder: Contextualize input only; Decoder: Generate output sequentially
Text Generation Capability	Encoder: No; Decoder: Yes
Contextual Understanding	Encoder: Full bidirectional context; Decoder: Past context only
Typical Use Cases	Encoder: Classification, retrieval; Decoder: Storytelling, auto-complete, dialogue

Current Section

Why Split an Image into Patches?

Motivating Question: How can a Transformer–designed for sequences–process images effectively?

Patch Embedding – Definition

Divide an input image of size $H \times W \times C$ into non-overlapping patches of size $P \times P$, yielding

$$N = \frac{H}{P} \times \frac{W}{P}$$

patches. Each patch is flattened to a vector of dimension P^2C and projected to a D-dimensional embedding.

Why Split an Image into Patches?

Image as a Puzzle

Think of an image as a completed complicated puzzle. Splitting into patches is like separating it into individual puzzle pieces–each piece carries part of the full picture, and later all pieces can be reassembled (via attention) to recover the whole.

$$\mathbf{x}_p = \text{Flatten}(\text{patch}_p) \in \mathbb{R}^{P^2 C}, \quad \mathbf{z}_p = \mathbf{x}_p W_E + b_E \in \mathbb{R}^D$$

where $W_E \in \mathbb{R}^{P^2 C \times D}$ is the patch projection matrix.

Patch Size Trade-Offs

Motivating Question: What are the consequences of choosing larger versus smaller patch sizes?

Effects of Smaller Patches

- **Higher resolution:** captures fine-grained details.
- **More tokens:** N increases as P decreases.
- **Compute cost:** self-attention scales as $\mathcal{O}(N^2)$.

Effects of Larger Patches

- **Lower resolution:** may miss small patterns.
- **Fewer tokens:** reduces sequence length.
- **Reduced compute:** faster but less expressive.

Patch Size Trade-Offs

Caveat: Computational Bottleneck

Excessively small patches blow up N, leading to quadratic growth in attention cost and memory usage.

Risk: Information Loss

Excessively large patches merge distinct features, harming the model's ability to distinguish fine details.

Patch Size Like Map Resolution

Choosing patch size is like selecting map resolution: too coarse, you lose streets; too fine, you're overwhelmed by data points.

Summary Table: Patch Splitting

Aspect	**Small Patches vs. Large Patches**
Number of Tokens N	Small P: large N; Large P: small N
Detail Captured	Small P: high; Large P: low
Attention Cost	Small P: $\mathcal{O}(N^2)$ high; Large P: lower
Memory Usage	Small P: high; Large P: lower
Representation Fidelity	Small P: fine-grained; Large P: coarse

Current Section

Why Project Flattened Patches into Embeddings?

Motivating Question: Given a flattened patch vector of raw pixels, why do we need a learned linear projection before feeding it to the Transformer?

Linear Projection – Definition and Role

Each flattened patch $\mathbf{x}_p \in \mathbb{R}^{P^2 C}$ is mapped to a D-dimensional embedding:

$$\mathbf{z}_p = \mathbf{x}_p W_E + b_E, \quad W_E \in \mathbb{R}^{P^2 C \times D},\ b_E \in \mathbb{R}^D.$$

This aligns patch representations to the model's hidden dimension and enables parameter sharing across patches.

Why Project Flattened Patches into Embeddings?

Key Benefits

- **Dimensionality alignment:** raw pixel vectors often have dimension much larger or smaller than the Transformer's hidden size D.
- **Parameter efficiency:** a shared W_E reduces the number of parameters versus separate per-patch mappings.
- **Learnable features:** the projection can adapt to capture useful patch-level patterns.

Caveat

If D is too small, the projection may bottleneck and discard important patch information; if too large, it may overfit and increase computation.

Alternative Inputs: Pre-trained CNN Features

Motivating Question: Instead of raw pixel patches, can we use features extracted by a pre-trained CNN as input tokens?

CNN Features – Definition and Role

Use a backbone CNN to produce N feature vectors:

$$\mathbf{f}_i = \mathrm{CNNFeature}(\mathrm{patch}_i) \in \mathbb{R}^F,$$

then project via $\mathbf{z}_i = \mathbf{f}_i W_F + b_F,\ W_F \in \mathbb{R}^{F \times D}$.

CNN as a Microscope

A CNN acts like a microscope that pre-abstracts local patterns, so the Transformer starts from higher-level "observations" rather than raw pixels.

Alternative Inputs: Pre-trained CNN Features

Advantages of CNN Features

- **Stronger inductive biases:** convolution encodes locality and translation invariance.
- **Reduced model size:** Transformer can be smaller since features are already rich.
- **Faster convergence:** benefits from pre-training on large image datasets.

Risk and Limitation

Reliance on CNN features may limit the Transformer's ability to learn global dependencies from raw data and can introduce biases of the CNN architecture.

Summary Table: Projection Methods

Aspect	Raw Patch Projection vs. CNN Feature Input
Input Dimension	P^2C (raw pixels) vs. F (CNN feature size)
Mapping Matrix	$W_E \in \mathbb{R}^{P^2C \times D}$ vs. $W_F \in \mathbb{R}^{F \times D}$
Inductive Bias	Minimal (learned) vs. strong (convolution)
Parameter Efficiency	High (shared across patches)
Higher (smaller Transformer)	
Pre-training Requirement	None (end-to-end) vs. CNN backbone pre-trained
Flexibility	Learns from scratch
May inherit CNN biases	

Current Section

Why Do We Need Positional Encodings?

Motivating Question: Transformers are permutation-invariant. How does a Vision Transformer know the spatial arrangement of patches?

Positional Encoding – Definition

A positional encoding injects information about each patch's position into its embedding, so the model can distinguish "patch at (row i, col j)" from "patch at (row k, col ℓ)".

Coordinates on a Map

Just as a map labels each region with its grid coordinates so you know where to look, positional encodings label each patch embedding with its location.

Fixed (Sinusoidal) Positional Encoding

Motivating Question: How can we encode positions without learning extra parameters?

Sinusoidal Encoding

For embedding dimension D, position p and index i ($0 \leq i < D$):

$$\mathrm{PE}(p)_{2i} = \sin\left(\frac{p}{10000^{2i/D}}\right), \quad \mathrm{PE}(p)_{2i+1} = \cos\left(\frac{p}{10000^{2i/D}}\right).$$

Add elementwise: $\tilde{\mathbf{z}}_p = \mathbf{z}_p + \mathrm{PE}(p)$.

Orthogonal Phase Embedding

Using sine on even indices and cosine on odd indices gives two signals that are 90 degree out of phase, creating an orthogonal pair whose combined values form a unique and easily invertible code for each position.

Fixed (Sinusoidal) Positional Encoding

Numerical Example:

Let $D = 4$, position $p = 1$:

$$\mathrm{PE}(1)_0 = \sin\left(1/10000^{0/4}\right) = \sin(1) = 0.8415, \quad \mathrm{PE}(1)_1 = \cos(1) = 0.5403,$$

$$\mathrm{PE}(1)_2 = \sin\left(1/10000^{2/4}\right) = \sin(1/100) \approx 0.0099998, \quad \mathrm{PE}(1)_3 = \cos(1/100) \approx 0.9950.$$

Adding to a patch embedding $\mathbf{z}_1 = [z_0, z_1, z_2, z_3]$ yields

$$\tilde{\mathbf{z}}_1 = [z_0 + 0.8415,\ z_1 + 0.5403,\ z_2 + 0.0099998,\ z_3 + 0.9950].$$

Learnable Positional Encoding

Motivating Question: What if we let the model learn the best way to encode positions?

Learnable Encoding

Initialize a matrix $\mathbf{P} \in \mathbb{R}^{N \times D}$ of position embeddings. For patch index p ($0 \leq p < N$), add:

$$\tilde{\mathbf{z}}_p = \mathbf{z}_p + \mathbf{P}_p.$$

Both $\mathbf{P}$ and the rest of the model are learned end-to-end.

Custom Map Legend

Like a cartographer drawing custom symbols for each map location, the model learns unique embeddings for each patch index.

Learnable Positional Encoding

Numerical Example:

Let $N = 2$, $D = 4$, and suppose

$$\mathbf{P} = \begin{bmatrix} 0.1 & 0.2 & 0.3 & 0.4 \\ -0.1 & 0.0 & 0.1 & 0.2 \end{bmatrix}.$$

For patch 0 with $\mathbf{z}_0 = [1, 1, 1, 1]$,

$$\tilde{\mathbf{z}}_0 = [1.1, 1.2, 1.3, 1.4].$$

For patch 1 with $\mathbf{z}_1 = [2, 2, 2, 2]$,

$$\tilde{\mathbf{z}}_1 = [1.9, 2.0, 2.1, 2.2].$$

Summary Table: Positional Encoding

Aspect	**Fixed (Sinusoidal)**	**Learnable**
Parameterization	No extra parameters	$N \times D$ parameters
Injection Method	Analytical sine/cosine formulas	Direct lookup from learned matrix **P**
Key Advantage	Extrapolates to unseen positions	Learns task-specific spatial patterns
Main Limitation	May not adapt fully to data	Cannot generalize to sequence lengths beyond N
Computational Cost	$\mathcal{O}(D)$ operations per position	$\mathcal{O}(1)$ table lookup per position
Numerical Example	See slide on $p = 1, D = 4$	See slide on $N = 2, D = 4$

Current Section

What Is the [CLS] Token and Why Introduce It?

Motivating Question: How can a Vision Transformer produce a single, global summary of all patch embeddings for classification or other downstream tasks?

[CLS] Token – Definition and Role

A special learnable token `[CLS]` is prepended to the sequence of patch embeddings. After all Transformer layers, its final hidden state $\mathbf{h}_{[\mathrm{CLS}]}$ serves as a global representation of the entire input.

Chairperson of a Meeting

Imagine all patch embeddings as meeting participants reporting their local observations. The `[CLS]` token is the chairperson who listens to everyone and summarizes the discussion into a single report.

Input sequence:

$$[\mathbf{z}_{[\mathrm{CLS}]}, \mathbf{z}_1, \mathbf{z}_2, \ldots, \mathbf{z}_N].$$

Positional Encoding for the [CLS] Token

Motivating Question: Since `[CLS]` is not a spatial patch, how do we assign it a positional encoding?

Positional Embedding of `[CLS]`

We allocate an embedding $\mathrm{PE}_{[\mathrm{CLS}]}$ (learnable or fixed) for the class token's position (often index 0), and add it to its token embedding:

$$\tilde{\mathbf{z}}_{[\mathrm{CLS}]} = \mathbf{z}_{[\mathrm{CLS}]} + \mathrm{PE}_{[\mathrm{CLS}]}.$$

This allows the model to distinguish `[CLS]` from all patches.

Caveat: No Spatial Semantics

Unlike patches, `[CLS]` has no (row, col) position; its positional encoding is purely a learned identifier, not a location.

How [CLS] Aggregates Information

Motivating Question: How does `[CLS]` gather and summarize features from all patches?

Attention-Based Aggregation

In each self-attention layer, `[CLS]` attends to every patch token and is attended to by them. Over L layers, its hidden state evolves:

$$\mathbf{h}_{[\mathrm{CLS}]}^{(\ell)} = \mathrm{Attention}\big(\mathbf{h}_{[\mathrm{CLS}]}^{(\ell-1)}, \{\mathbf{h}_i^{(\ell-1)}\}\big).$$

After the final layer, $\mathbf{h}_{[\mathrm{CLS}]}^{(L)}$ encodes global context.

Information Funnel

Each attention layer is like pouring all patch reports through a funnel–the [CLS] state at the bottom collects and concentrates the information.

Summary Table: [CLS] Token Characteristics

Aspect	**[CLS] Token**
Purpose	Global summary vector for classification or regression
Initialization	Learnable token embedding $\mathbf{z}_{[\text{CLS}]}$
Positional Encoding	Separate learnable or fixed embedding at index 0
Aggregation Mechanism	Self-attention to/from all patch tokens across L layers
Output Use	Input to task-specific head (e.g. MLP for classification)
No Spatial Semantics	Carries no explicit (row, col) information

Current Section

Overview of the Transformer Block

Motivating Question: Given input representations $\mathbf{H}^{(\ell-1)}$, how does one block produce refined outputs $\mathbf{H}^{(\ell)}$?
Each block comprises four stages in sequence:

1. **Multi-Head Self-Attention** (contextual mixing)
2. **Add & LayerNorm**
3. **Feed-Forward Network (MLP)** (nonlinear transformation)
4. **Add & LayerNorm**

Workshop Workflow

A workshop: first everyone shares knowledge (attention), then you pause to organize notes (add&norm), then each person works on their own design (MLP), and finally you tidy up again before the next round.

1. Multi-Head Self-Attention

Motivating Question: Why should each token consult all others before updating itself?

Definition

Compute for input $\mathbf{H} \equiv \mathbf{H}^{(\ell-1)}$:

$$\mathbf{Q}_i = \mathbf{H}W_i^Q,\ \mathbf{K}_i = \mathbf{H}W_i^K,\ \mathbf{V}_i = \mathbf{H}W_i^V,$$

$$\text{head}_i = \text{softmax}\Big(\frac{\mathbf{Q}_i\mathbf{K}_i^T}{\sqrt{d_k}}\Big)\mathbf{V}_i, \quad \text{MHA}(\mathbf{H}) = [\text{head}_1, \ldots, \text{head}_h]\, W^O.$$

Roundtable Discussion

Each token is like a participant: it listens to everyone else's viewpoints (keys/values) but weighs them by relevance (queryâkey score) before forming its own updated opinion.

2. Add & LayerNorm after Attention

Motivating Question: How do we stabilize and preserve the original signal after mixing context?

Add & Norm

Let $\mathbf{A} = \mathrm{MHA}(\mathbf{H}^{(\ell-1)})$. Then

$$\mathbf{X} = \mathrm{LayerNorm}\big(\mathbf{H}^{(\ell-1)} + \mathbf{A}\big).$$

Checkpoint and Reset

After a spirited discussion, you merge the new insights with your prior notes (residual), then normalize formatting so everyone's notes remain on the same scale.

3. Position-wise Feed-Forward Network

Motivating Question: Why apply an independent nonlinear transform to each token?

Definition

For each row $\mathbf{x}_i$ of $\mathbf{X}$:

$$\mathrm{FFN}(\mathbf{x}_i) = \sigma(\mathbf{x}_i W_1 + b_1)\, W_2 + b_2.$$

Apply to all tokens in parallel to obtain $\mathbf{F}$.

Independent Research

Each participant takes the shared discussion notes and diverges into personal deep-dive reading, discovering higher-order patterns unique to their token.

4. Add & LayerNorm after MLP

Motivating Question: How do we integrate the nonlinear features while maintaining stability?

Add & Norm

Given $\mathbf{F} = [\mathrm{FFN}(\mathbf{x}_i)]$ and $\mathbf{X}$:

$$\mathbf{H}^{(\ell)} = \mathrm{LayerNorm}\big(\mathbf{X} + \mathbf{F}\big).$$

Final Notes Consolidation

After individual study, you merge the new insights back into your main notes and normalize formatting, ready for the next iteration.

Summary of Transformer Block

Step	Purpose	Formula
Multi-Head Self-Attention	Contextual mixing	$\mathrm{MHA}(\mathbf{H}) = [\mathrm{head}_i]\, W^O$
Add & LayerNorm_1	Stabilize + preserve input	$\mathbf{X} = \mathrm{LN}(\mathbf{H}^{(\ell-1)} + \mathrm{MHA}(\mathbf{H}^{(\ell-1)}))$
Feed-Forward Network	Nonlinear feature extraction	$\mathrm{FFN}(\mathbf{x}) = \sigma(\mathbf{x} W_1 + b_1) W_2 + b_2$
Add & LayerNorm_2	Integrate nonlinear features	$\mathbf{H}^{(\ell)} = \mathrm{LN}(\mathbf{X} + \mathrm{FFN}(\mathbf{X}))$

Current Section

What Comes After the Transformer Blocks?

Motivating Question: Once all L Transformer encoder blocks have refined our token embeddings, how do we convert them into task-specific outputs (e.g. class probabilities)?

Overview of Remaining Steps

After the final block, we have:

$$\mathbf{H}^{(L)} = [\mathbf{h}_{[\text{CLS}]}^{(L)}, \mathbf{h}_1^{(L)}, \ldots, \mathbf{h}_N^{(L)}].$$

We must 1. **Extract** a global summary (`[CLS]`). 2. **Optionally normalize** or dropout. 3. **Apply a task head** (e.g. MLP + softmax). 4. **Compute loss** and backpropagate.

Final Report Preparation

After rounds of discussion (blocks), the chairperson ([CLS]) prepares the final report, formats it (norm/dropout), and submits it to the verdict committee (task head).

1. Extract the [CLS] Representation

Motivating Question: Which embedding carries the global information we need for classification or regression?

Selecting `[CLS]`

We take the final hidden state of the class token:

$$\mathbf{u} = \mathbf{h}_{[\mathrm{CLS}]}^{(L)} \in \mathbb{R}^D.$$

This vector encodes information aggregated from all patches across all layers.

Caveat

If the task requires per-patch predictions (e.g. segmentation), use the individual $\mathbf{h}_i^{(L)}$ instead of `[CLS]`.

2. Apply Dropout and LayerNorm

Motivating Question: How can we regularize and stabilize the summary vector before the final head?

Dropout and Normalization

Often we apply:

$$\mathbf{u}' = \mathrm{LayerNorm}(\mathbf{u}), \qquad \mathbf{u}'' = \mathrm{Dropout}(\mathbf{u}').$$

This prevents overfitting and keeps magnitudes consistent.

Proofreading and Redaction

Before sending the report, you proofread (norm) and redact sensitive parts (dropout) to ensure consistency and privacy.

3. Task-Specific Head

Motivating Question: How do we map the summary vector to our desired output (e.g. class scores or a continuous value)?

Classification Head

For C classes, use an MLP + softmax:

$$\mathbf{o} = W_{\text{cls}}\mathbf{u}'' + \mathbf{b}_{\text{cls}}, \qquad \hat{\mathbf{y}} = \text{softmax}(\mathbf{o}) \in \mathbb{R}^C.$$

Regression Head

For scalar output (e.g. bounding box coordinate):

$$\hat{y} = W_{\text{reg}}\mathbf{u}'' + b_{\text{reg}} \in \mathbb{R}.$$

4. Loss Computation and Backpropagation

Motivating Question: How do we train the entire model end-to-end for our task?

Loss Functions

- **Classification:** Cross-entropy $\mathscr{L} = -\sum_{c=1}^{C} y_c \log \hat{y}_c$.
- **Regression:** Mean squared error $\mathscr{L} = \frac{1}{2}(\hat{y} - y)^2$.

Backpropagation

Gradients of $\mathscr{L}$ flow through the head, dropout/norm, Transformer blocks, and patch projections, updating all parameters.

Summary Table: From Blocks to Output

Step	Operation	Formula
Extract `[CLS]`	Global summary	$\mathbf{u} = \mathbf{h}_{[\text{CLS}]}^{(L)}$
Norm + Dropout	Regularization	$\mathbf{u}'' = \text{Dropout}(\text{LayerNorm}(\mathbf{u}))$
Task Head	Classification or regression	$\hat{\mathbf{y}} = \text{softmax}(W_{\text{cls}}\mathbf{u}'' + \mathbf{b})$
Loss	Supervised objective	$\mathscr{L} = -\sum y \log \hat{y}$ or $\frac{1}{2}(\hat{y} - y)^2$