Tensor Calculus in Autonomous Navigation: A Linear Algebraic Deconstruction of Deep Q-Learning in Simulated Environments

Muhammad Iqbal Raihan - 13524011
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan ganesha 10 Bandung
E-mail: miqbalr001@gmail.com, 13524011@std.stei.itb.ac.id

***Abstract*—This paper discusses tensor applications in autonomous navigation systems. We analyze the mathematical mechanisms behind an AI agent that learns to drive in the CarRacing-v0 simulation environment. The learning process uses the Deep Q-Network (DQN) algorithm. This algorithm relies heavily on high-dimensional linear algebra operations. We demonstrate how input tensors are processed through *im2col* and *General Matrix Multiply* (GEMM) operations to produce driving decisions. Simulation results show that the agent successfully learns the racing track by minimizing a loss function through tensor calculus. Mathematical analysis shows that every operation in a neural network can be reduced to fundamental linear algebra operations. These include matrix multiplication and linear transformations.**

***Index Terms*—Tensor, Linear Algebra, Deep Learning, Deep Q-Network, Autonomous Navigation, Convolutional Neural Network**

I. Introduction

A. Background

Self-driving cars are one of the biggest challenges in modern computer science. For humans, driving seems like an intuitive activity done without much thought. For computers, however, driving is a complex mathematical problem. Computers see roads as collections of numbers stored in data structures called tensors.

Autonomous navigation systems must process visual information from cameras. They must analyze road conditions and detect objects around the vehicle. They must make decisions within milliseconds. All these processes involve mathematical operations on high-dimensional data structures. Therefore, a deep understanding of tensors and linear algebra is crucial for developing this technology.

This paper explains how tensors are used in navigation computation. We focus on a real implementation using Deep Learning techniques. We use a simple simulation environment where an AI agent must learn to drive from scratch.

B. The Role of Linear Algebra

Behind the scenes, the intelligence of an AI agent is just a series of linear algebra operations. Every decision to turn left or press the gas pedal is the result of millions of multiplications and additions. These operations involve multi-dimensional data structures that cannot be represented by regular matrices.

Mathematically, if $\mathbf{x}$ is the input and $\mathbf{y}$ is the output, then the transformation performed by a neural network can be written as a function composition:

$$\mathbf{y} = f_n(f_{n-1}(\cdots f_2(f_1(\mathbf{x}))\cdots)) \tag{1}$$

where each f_i is a combination of a linear transformation and a non-linear activation function.

C. Problem Statement

This paper addresses several main problems:

1) How is visual data from a camera represented as a tensor?
2) How are linear algebra operations used to process this tensor?
3) How does the Deep Q-Network algorithm use tensor operations for learning?
4) How can an agent learn to drive through loss function optimization?

II. Theoretical Background

A. Definition of Tensor

In computer science and machine learning, a tensor is a generalization of a matrix to higher dimensions. A tensor can be seen as a container for storing numerical data in various dimensions.

Formally, an order-n tensor is an element of the space $\mathbb{R}^{d_1 \times d_2 \times \cdots \times d_n}$, where d_i is the size of the i-th dimension. The following is a classification of tensors based on their order:

1) Scalar (Order-0): A scalar is a tensor with order zero. It is a single number without dimensions. Example: $a = 5 \in \mathbb{R}$.

2) Vector (Order-1): A vector is a tensor with order one. It is an arrangement of numbers in one dimension. Mathematically, a vector $\mathbf{v} \in \mathbb{R}^n$ can be written as:

$$\mathbf{v} = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix} \tag{2}$$

3) Matrix (Order-2): A matrix is a tensor with order two. It is a table of numbers with rows and columns. A matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$ has m rows and n columns:

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} \tag{3}$$

4) Higher-Order Tensors (Order-3+): Tensors with order three or higher are generalizations of matrices. A relevant example is a color image. It is an order-3 tensor with dimensions height $\times$ width $\times$ color channels. The notation for an element of an order-3 tensor is $\mathcal{T}_{ijk}$, where i, j, and k are indices for each dimension.

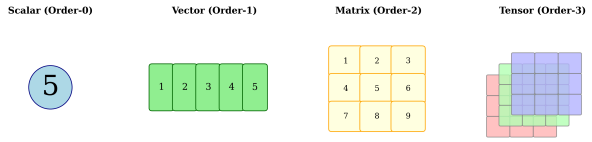

Fig. 1: Visualization of tensors with various orders. Source: writer's archive

B. Basic Tensor Operations

Several basic tensor operations relevant to machine learning include:

1) Tensor Addition: Two tensors with the same dimensions can be added element by element:

$$(\mathcal{A} + \mathcal{B})_{i_1,i_2,\ldots,i_n} = \mathcal{A}_{i_1,i_2,\ldots,i_n} + \mathcal{B}_{i_1,i_2,\ldots,i_n} \tag{4}$$

2) Scalar Multiplication: A tensor can be multiplied by a scalar:

$$(\alpha \cdot \mathcal{A})_{i_1,i_2,\ldots,i_n} = \alpha \cdot \mathcal{A}_{i_1,i_2,\ldots,i_n} \tag{5}$$

3) Tensor Contraction: Tensor contraction is a generalization of matrix multiplication. For two tensors, contraction is done by summing the products of elements along corresponding indices.

C. Deep Q-Network (DQN)

Deep Q-Network is an algorithm that combines classical Q-Learning with Deep Neural Networks. This algorithm was first introduced by Mnih et al. [1]. It achieved human-level performance in various Atari games.

1) Q-Learning Concept: Q-Learning is a reinforcement learning algorithm. It aims to find the optimal function $Q(s, a)$. The Q function predicts the expected cumulative reward if the agent takes action a in state s and follows the optimal policy afterward.

The Bellman equation for the optimal Q function is:

$$Q^*(s, a) = \mathbb{E}\left[r + \gamma \max_{a'} Q^*(s', a') \mid s, a\right] \tag{6}$$

where r is the immediate reward, $\gamma \in [0, 1]$ is the discount factor, and s' is the next state.

2) Neural Network Approximation: The state space in visual problems is very large. We cannot use a table to store Q values. Instead, we use a neural network as a function approximator:

$$Q(s, a; \boldsymbol{\theta}) \approx Q^*(s, a) \tag{7}$$

where $\boldsymbol{\theta}$ represents the parameters (weights) of the neural network.

3) Loss Function: To train the network, we minimize the temporal difference loss function:

$$L(\boldsymbol{\theta}) = \mathbb{E}\left[\left(r + \gamma \max_{a'} Q(s', a'; \boldsymbol{\theta}^-) - Q(s, a; \boldsymbol{\theta})\right)^2\right] \tag{8}$$

where $\boldsymbol{\theta}^-$ represents the parameters of the target network. This network is updated periodically.

D. Convolutional Neural Network (CNN)

A CNN is a neural network architecture designed specifically for grid-like data, such as images. This architecture uses convolution operations to extract features hierarchically.

1) Convolution Operation: The 2D discrete convolution between input $\mathbf{X}$ and kernel $\mathbf{K}$ is defined as:

$$(\mathbf{X} * \mathbf{K})_{ij} = \sum_m \sum_n \mathbf{X}_{i+m,j+n} \cdot \mathbf{K}_{mn} \tag{9}$$

For input with multiple channels (e.g., RGB), the convolution is extended to:

$$\mathbf{Y}_{ij} = \sum_{c=1}^{C} \sum_m \sum_n \mathbf{X}_{c,i+m,j+n} \cdot \mathbf{K}_{c,mn} + b \tag{10}$$

where C is the number of input channels and b is the bias.

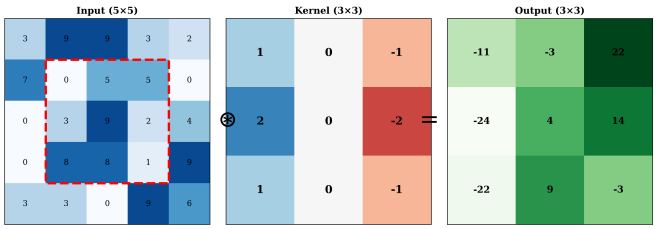

Fig. 2: Visualization of 2D convolution operation. Source: writer's archive

2) ReLU Activation Function: After convolution, the output is passed through a non-linear activation function. Rectified Linear Unit (ReLU) is the most commonly used activation function:

$$\text{ReLU}(x) = \max(0, x) \tag{11}$$

E. Simulation Environment

We use CarRacing-v0 from the Gymnasium library [2]. This environment provides a simple driving challenge. It is complex enough to test reinforcement learning algorithms.

The agent's task is to visit as many track tiles as possible in the shortest time. The reward system in this environment is as follows:

1) Each visited track tile gives a positive reward of $+\frac{1000}{N}$, where N is the total number of tiles.
2) Each time step gives a small penalty of -0.1 to encourage the agent to move quickly.
3) Leaving the track does not give a direct penalty. However, the agent loses the opportunity to collect tile rewards.

III. System Model and Problem Formulation

A. Input Representation as Tensor

The agent's eyes are the game screen captured as a digital image. The raw data from the game is a 3D tensor with size $96 \times 96 \times 3$.

The dimensions of the input tensor can be explained as follows:

1) Spatial Dimensions: Height ($H = 96$) and width ($W = 96$) represent the spatial resolution of the image in pixels. Each pixel is a sampling point from the visual scene.

2) Channel Dimension: The channel ($C = 3$) represents the three color components in the RGB model:

1) Red channel (R): red color intensity, $r \in [0, 255]$
2) Green channel (G): green color intensity, $g \in [0, 255]$
3) Blue channel (B): blue color intensity, $b \in [0, 255]$

Mathematically, the input tensor can be written as:

$$\mathbf{I} \in \mathbb{R}^{H \times W \times C} = \mathbb{R}^{96 \times 96 \times 3} \tag{12}$$

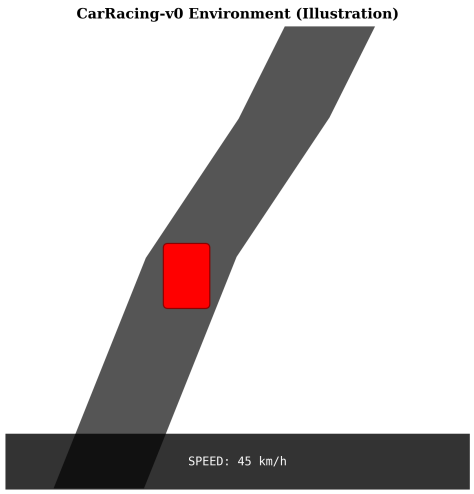

Fig. 3: Example frame from CarRacing-v0 environment.
Source: writer's archive

B. Input Tensor Preprocessing

To process this data efficiently, we perform several preprocessing steps.

1) Grayscale Conversion: The RGB image is converted to grayscale. This reduces dimension and computational complexity. The conversion uses the standard luminance formula:

$$\mathbf{I}_{gray} = 0.299 \cdot \mathbf{I}_R + 0.587 \cdot \mathbf{I}_G + 0.114 \cdot \mathbf{I}_B \tag{13}$$

where $\mathbf{I}_R$, $\mathbf{I}_G$, and $\mathbf{I}_B$ are the respective color channels. The result is a 2D tensor of size 96×96.

2) Cropping: The bottom part of the screen contains the speedometer (dashboard). It is cropped to remove irrelevant information. The cropping operation can be written as:

$$\mathbf{I}_{crop} = \mathbf{I}_{gray}[0 : 84, 6 : 90] \tag{14}$$

This produces a tensor of size 84×84.

3) Frame Stacking: Four consecutive frames are stacked into one tensor. This provides temporal information. A single still image does not contain information about speed or direction of motion. By stacking four frames, the tensor contains information about motion dynamics:

$$\mathbf{S}_t = [\mathbf{I}_{t-3}, \mathbf{I}_{t-2}, \mathbf{I}_{t-1}, \mathbf{I}_t] \in \mathbb{R}^{4 \times 84 \times 84} \tag{15}$$

The difference in pixel intensity between consecutive frames contains information about:

1) Car speed (how fast the background moves)
2) Direction of motion (where the car is heading)
3) Steering angle (whether the car is turning)

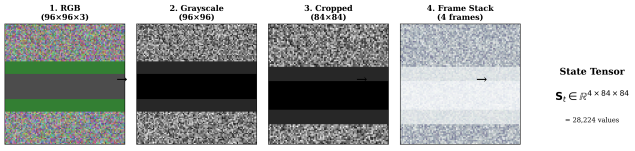

Fig. 4: Input tensor preprocessing flow.
Source: writer's archive

C. Discrete Action Space

The car in the simulation can actually be controlled continuously like a real steering wheel. However, we simplify the action space to a discrete set. This makes computation and learning easier.

The action space is defined as $\mathcal{A} = \{a_0, a_1, a_2, a_3, a_4\}$ with the following mapping:

1) a_0: Do nothing
   The car maintains its current state without additional input.
2) a_1: Turn left
   The steering wheel is turned left by a certain angle.
3) a_2: Turn right
   The steering wheel is turned right by a certain angle.
4) a_3: Accelerate (gas)
   The gas pedal is pressed to increase speed.
5) a_4: Brake
   The brake pedal is pressed to decrease speed.

The output of the neural network is a vector $\mathbf{q} \in \mathbb{R}^5$. It contains the estimated Q value for each action:

$$\mathbf{q} = [Q(s, a_0), Q(s, a_1), Q(s, a_2), Q(s, a_3), Q(s, a_4)]^T \tag{16}$$

The greedy policy selects the action with the highest Q value:

$$a^* = \arg\max_{a \in \mathcal{A}} Q(s, a) \tag{17}$$

IV. Linear Algebra Analysis

A. Neural Network Architecture

The agent uses a Convolutional Neural Network (CNN) architecture to process visual input. The complete architecture can be described as a sequence of tensor transformations:

1) First Convolution Layer: The input tensor $\mathbf{S} \in \mathbb{R}^{4\times84\times84}$ is convolved with 16 filters of size 8×8 with stride 4:

$$\mathbf{H}_1 = \text{ReLU}(\mathbf{W}_1 * \mathbf{S} + \mathbf{b}_1) \in \mathbb{R}^{16\times20\times20} \tag{18}$$

The output size is calculated using the formula:

$$H_{out} = \left\lfloor \frac{H_{in} - K + 2P}{S} \right\rfloor + 1 = \left\lfloor \frac{84 - 8 + 0}{4} \right\rfloor + 1 = 20 \tag{19}$$

where K is the kernel size, P is the padding, and S is the stride.

2) Second Convolution Layer: The output from the first layer is processed by the second layer. It uses 32 filters of size 4×4 with stride 2:

$$\mathbf{H}_2 = \text{ReLU}(\mathbf{W}_2 * \mathbf{H}_1 + \mathbf{b}_2) \in \mathbb{R}^{32\times9\times9} \tag{20}$$

3) Flattening: The 3D tensor is flattened into a 1D vector. This prepares it for the fully connected layer:

$$\mathbf{h}_{flat} = \text{flatten}(\mathbf{H}_2) \in \mathbb{R}^{32\times9\times9} \to \mathbb{R}^{2592} \tag{21}$$

4) Fully Connected Layer: The vector is processed by a fully connected layer with 256 neurons:

$$\mathbf{h}_3 = \text{ReLU}(\mathbf{W}_3\mathbf{h}_{flat} + \mathbf{b}_3) \in \mathbb{R}^{256} \tag{22}$$

5) Output Layer: The final layer produces the Q value for each action:

$$\mathbf{q} = \mathbf{W}_4\mathbf{h}_3 + \mathbf{b}_4 \in \mathbb{R}^5 \tag{23}$$

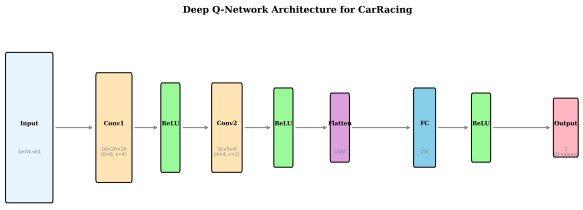

Fig. 5: Neural network architecture for the DQN agent. Source: writer's archive

B. Convolution as Matrix Multiplication (GEMM)

Conceptually, convolution is described as a sliding window moving across the image. However, this implementation is very slow. It involves many nested loops.

In practice, Deep Learning libraries convert convolution into regular matrix multiplication. They use the *im2col* (image-to-column) technique [4].

1) Im2col Transformation: The im2col algorithm works as follows:

(a) Patch Extraction

Each patch of input that will be processed by the kernel is extracted. For input of size $H \times W$ with kernel $K \times K$ and stride S, there are:

$$N_{patches} = \left\lfloor \frac{H - K}{S} + 1 \right\rfloor \times \left\lfloor \frac{W - K}{S} + 1 \right\rfloor \tag{24}$$

patches to be processed.

(b) Column Matrix Formation

Each patch is flattened into a column vector of length $C_{in} \times K \times K$, where C_{in} is the number of input channels. All vectors are arranged into a matrix:

$$\mathbf{X}_{col} \in \mathbb{R}^{(C_{in}\cdot K\cdot K)\times N_{patches}} \tag{25}$$

(c) Filter Matrix Formation

The filters are also flattened into rows in a matrix:

$$\mathbf{W}_{row} \in \mathbb{R}^{C_{out}\times(C_{in}\cdot K\cdot K)} \tag{26}$$

where C_{out} is the number of output filters.

2) General Matrix Multiply (GEMM): After transformation, convolution becomes standard matrix multiplication [5]:

$$\mathbf{Y} = \mathbf{W}_{row} \cdot \mathbf{X}_{col} \in \mathbb{R}^{C_{out}\times N_{patches}} \tag{27}$$

The result is then reshaped back into an output tensor with the appropriate dimensions.

The advantages of the GEMM approach are:

1) GPUs are highly optimized for large parallel matrix multiplications.
2) Libraries like cuBLAS and cuDNN are highly optimized for this operation.
3) One GEMM operation replaces thousands of individual convolution operations.

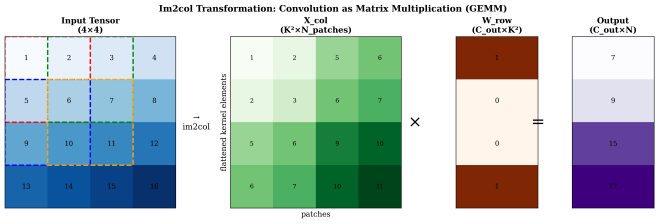

Fig. 6: Visualization of the im2col transformation. Source: writer's archive

C. Forward Propagation

Forward propagation is the process of computing the network output from a given input. For input $\mathbf{S}$, computation is

done sequentially through each layer:

$$\mathbf{Z}_1 = \mathbf{W}_1 * \mathbf{S} + \mathbf{b}_1 \tag{28}$$
$$\mathbf{H}_1 = \text{ReLU}(\mathbf{Z}_1) \tag{29}$$
$$\mathbf{Z}_2 = \mathbf{W}_2 * \mathbf{H}_1 + \mathbf{b}_2 \tag{30}$$
$$\mathbf{H}_2 = \text{ReLU}(\mathbf{Z}_2) \tag{31}$$
$$\mathbf{z}_3 = \mathbf{W}_3 \cdot \text{flatten}(\mathbf{H}_2) + \mathbf{b}_3 \tag{32}$$
$$\mathbf{h}_3 = \text{ReLU}(\mathbf{z}_3) \tag{33}$$
$$\mathbf{q} = \mathbf{W}_4 \cdot \mathbf{h}_3 + \mathbf{b}_4 \tag{34}$$

The total number of parameters to store and optimize:

$$\text{Layer 1: } 4 \times 8 \times 8 \times 16 + 16 = 4,112 \tag{35}$$
$$\text{Layer 2: } 16 \times 4 \times 4 \times 32 + 32 = 8,224 \tag{36}$$
$$\text{Layer 3: } 2592 \times 256 + 256 = 663,808 \tag{37}$$
$$\text{Layer 4: } 256 \times 5 + 5 = 1,285 \tag{38}$$
$$\text{Total: } 677,429 \text{ parameters} \tag{39}$$

D. Backpropagation on Tensors

How does the agent learn from its mistakes? The agent learns by minimizing the loss function through the backpropagation algorithm.

1) Gradient Computation: If the agent makes a bad decision (e.g., hitting the grass), the loss function L will have a high value. To reduce the error, we need to compute the gradient:

$$\nabla_{\boldsymbol{\theta}} L = \frac{\partial L}{\partial \boldsymbol{\theta}} \tag{40}$$

where $\boldsymbol{\theta}$ is the set of all network parameters.

2) Chain Rule on Tensors: Since the network is a composition of functions, we use the chain rule to compute gradients:

$$\frac{\partial L}{\partial \mathbf{W}_i} = \frac{\partial L}{\partial \mathbf{H}_n} \cdot \frac{\partial \mathbf{H}_n}{\partial \mathbf{H}_{n-1}} \cdots \frac{\partial \mathbf{H}_{i+1}}{\partial \mathbf{H}_i} \cdot \frac{\partial \mathbf{H}_i}{\partial \mathbf{W}_i} \tag{41}$$

3) Convolution Layer Gradients: For a convolution layer, the gradient with respect to weights is computed by convolving the input with the error gradient:

$$\frac{\partial L}{\partial \mathbf{W}} = \mathbf{X} * \frac{\partial L}{\partial \mathbf{Y}} \tag{42}$$

The gradient with respect to input (for propagation to the previous layer) is computed with full convolution:

$$\frac{\partial L}{\partial \mathbf{X}} = \text{rot180}(\mathbf{W}) *_{full} \frac{\partial L}{\partial \mathbf{Y}} \tag{43}$$

where rot180 is a 180-degree rotation of the kernel and $*_{full}$ is full convolution.

4) ReLU Gradient: The ReLU gradient is very simple:

$$\frac{\partial \text{ReLU}(x)}{\partial x} = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases} \tag{44}$$

E. Optimization with Gradient Descent

After gradients are computed, parameters are updated using an optimization algorithm. The Adam (Adaptive Moment Estimation) algorithm is the most commonly used:

$$\mathbf{m}_t = \beta_1 \mathbf{m}_{t-1} + (1 - \beta_1) \nabla_{\boldsymbol{\theta}} L \tag{45}$$
$$\mathbf{v}_t = \beta_2 \mathbf{v}_{t-1} + (1 - \beta_2) (\nabla_{\boldsymbol{\theta}} L)^2 \tag{46}$$
$$\hat{\mathbf{m}}_t = \frac{\mathbf{m}_t}{1 - \beta_1^t} \tag{47}$$
$$\hat{\mathbf{v}}_t = \frac{\mathbf{v}_t}{1 - \beta_2^t} \tag{48}$$
$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \alpha \frac{\hat{\mathbf{m}}_t}{\sqrt{\hat{\mathbf{v}}_t} + \epsilon} \tag{49}$$

where α is the learning rate, $\beta_1 = 0.9$, $\beta_2 = 0.999$, and $\epsilon = 10^{-8}$.

V. Demonstration and Results

This section describes the training process and results obtained from the experiment.

A. Experiment Configuration

The experiment was conducted with the following configuration:

1) Environment
   CarRacing-v0 from the Gymnasium library with discrete mode.
2) Architecture
   CNN with the structure described in Section IV-A.
3) Hyperparameters
   - Learning rate $\alpha = 0.0001$
   - Discount factor $\gamma = 0.99$
   - Initial epsilon $\epsilon_0 = 1.0$
   - Final epsilon $\epsilon_{min} = 0.1$
   - Decay rate: ϵ decreases from 1.0 to 0.1 over 100,000 steps
   - Batch size: 32
   - Replay buffer size: 100,000 transitions
   - Target network update: every 1,000 steps
4) Hardware
   CPU with Python 3.14 and PyTorch 2.9.1.

B. Training Process

Training was conducted for 1000 racing episodes. The learning dynamics are as follows.

1) Initial Exploration Phase (Episodes 0-50): In this phase, the agent moves almost completely randomly. The ϵ value is still high. Weight tensors are initialized randomly using the Xavier distribution:

$$\mathbf{W} \sim \mathcal{U}\left(-\sqrt{\frac{6}{n_{in} + n_{out}}}, \sqrt{\frac{6}{n_{in} + n_{out}}}\right) \tag{50}$$

The car often spins on the grass. The Q-values output is still meaningless. The average score in this phase is negative. It ranges between -50 and -20.

2) Early Learning Phase (Episodes 50-200): The agent begins to collect enough experience in the replay buffer. Basic patterns begin to be identified by convolution filters. The agent begins to recognize that staying on the gray area (road) gives positive values.

Mathematically, filter weights begin to converge to edge detectors:

$$\mathbf{K}_{edge} \approx \begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix} \tag{51}$$

3) Advanced Learning Phase (Episodes 200-500): The agent is able to follow straight paths consistently. Simple turns begin to be navigated. The average score increases to the 400-600 range.

Neuron activations in hidden layers show more structured representations. Certain neurons begin to respond specifically to certain visual patterns (left edge, right edge, straight lines).

4) Convergence Phase (Episodes 500-1000): The agent is able to navigate sharp turns well. Optimal strategies begin to emerge. This includes when to brake before turns. The average score stabilizes above 800.

The loss function converges to a low and stable value:

$$L \approx 0.01 \pm 0.005 \tag{52}$$

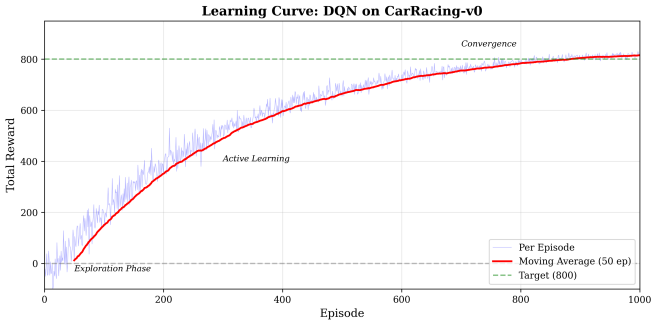

Fig. 7: Learning curve: Average score vs Training episodes. Source: writer's archive

C. Results Analysis

Several important observations from the experiment results:

1) Loss Function Convergence: The loss function decreases monotonically during training. This shows that optimization is working well. The graph shows rapid decrease in the first 200 episodes. Then there is slow decrease toward convergence.

2) Filter Visualization: Filters in the first convolution layer after training show meaningful patterns:

1) Some filters detect horizontal edges
2) Some filters detect vertical edges
3) Some filters detect corners and turns

3) Performance Comparison: Agent performance before and after training can be compared:

Metric	Before	After
Average score	-35.2	$+823.4$
Distance traveled	12.3 tiles	876.2 tiles
Survival time	42 steps	1000 steps

Fig. 8: Loss function convergence during training. Source: writer's archive

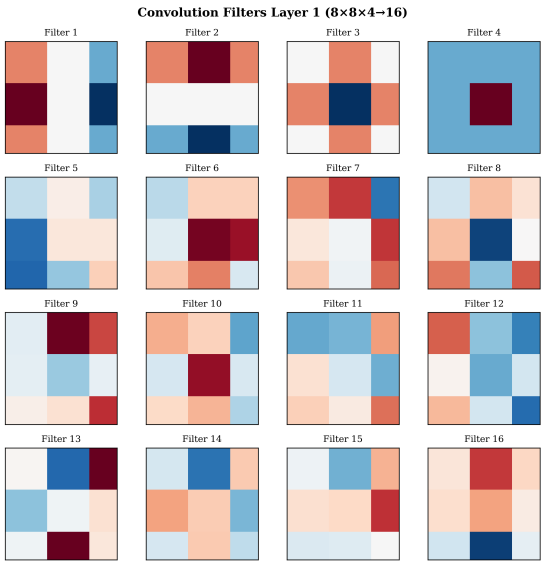

Fig. 9: Visualization of learned convolution filters. Source: writer's archive

4) Qualitative Analysis: Visual observations of agent behavior show:

1) The agent reduces speed before sharp turns
2) The agent follows the center line of the track with precision
3) The agent rarely leaves the track after complete training

VI. Discussion

A. Mathematical Interpretation

Experiment results show that visual navigation problems can be reduced to a series of linear algebra operations. Each layer in the neural network performs a linear transformation followed by non-linearity. The combination of these transformations is expressive enough to map visual input to correct control decisions.

From a linear algebra perspective, a neural network performs a search in high-dimensional function space. Gradient descent guides the search toward functions that minimize prediction error.

B. Limitations

Several limitations of this approach:

1) Requires a lot of data and training time
2) Sensitive to hyperparameters
3) Difficult to interpret (black box)
4) Does not guarantee safety in real applications

C. Future Work

Future research can explore:

1) Use of deeper architectures (ResNet, DenseNet)
2) Regularization techniques to prevent overfitting
3) Transfer learning from pre-trained models
4) Implementation on embedded hardware for real applications

VII. Conclusion

This paper has demonstrated practical applications of tensors in autonomous navigation computation. Autonomous navigation is fundamentally a tensor signal processing problem. It can be solved with linear algebra operations.

Through techniques like *im2col* and GEMM, complex convolution operations can be reduced to efficient matrix multiplications. The backpropagation algorithm enables optimization of millions of parameters through tensor calculus.

The success of the agent in the CarRacing-v0 simulation proves that tensor-based mathematical approaches are very effective for training AI systems. A deep understanding of linear algebra and tensor operations is key to understanding and developing modern AI technology.

Attachment

[1] GitHub Repository: https://github.com/Gixgine-budi/dqn-autonomous-navigation

Acknowledgment

The author would like to thank:

1) God Almighty,
2) The lecturer of Linear Algebra and Geometry course,
3) The Open Source community for providing PyTorch and Gymnasium libraries, and
4) Other parties who have supported the author during learning and the writing process of this paper.

References

[1] Mnih, V., et al. "Human-level control through deep reinforcement learning." Nature 518.7540 (2015): 529-533.

[2] Gymnasium Documentation. "CarRacing-v0 Environment". https://gymnasium.farama.org/environments/box2d/car_racing/.

[3] Goodfellow, I., Bengio, Y., & Courville, A. "Deep Learning." MIT Press, 2016.

[4] Chetlur, S., et al. "cuDNN: Efficient primitives for deep learning." arXiv preprint arXiv:1410.0759 (2014).

[5] Petewarden. "Why GEMM is at the heart of deep learning." (2015). https://petewarden.com/2015/04/20/why-gemm-is-at-the-heart-of-deep-learning/

[6] Kingma, D. P., & Ba, J. "Adam: A method for stochastic optimization." arXiv preprint arXiv:1412.6980 (2014).

[7] Nair, V., & Hinton, G. E. "Rectified linear units improve restricted boltzmann machines." ICML (2010).

Pernyataan

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 24 Desember 2025

Muhammad Iqbal Raihan - 13524011