DL Demystified 0 - Building a Strong Base - Linear Algebra

Dr. Adnan Masood (with ChatGPT)

January 28, 2025

[18]:
```
######################################################################
#                                                                    #
#  CS435 Generative AI: Security, Ethics and Governance              #
#                                                                    #
#  Instructor: Dr. Adnan Masood                                      #
#  Contact:    adnanmasood@gmail.com                                 #
#                                                                    #
#  Notebook is MIT Licensed                                          #
######################################################################
```

0.1 # Linear Algebra, Matrix Operations, and the Foundations of Deep Learning

Welcome, everyone! Today, we are going to explore the magical world of **Linear Algebra**—specifically focusing on **matrix operations**, **matrices multiplication (matmul)**, **eigenvectors**, and how all these connect to **neural networks**, **deep learning**, **Generative AI**, and **Large Language Models (LLMs)**.

We'll approach this in **five levels**, from a curious beginner to an advanced researcher. Along the way, we'll build intuition, see a brief history, do mock calculations, and implement working examples in **PyTorch**. By the end, you'll understand not just *how*, but *why* these concepts are so important to modern AI.

0.2 1. Building the Intuiton:

Imagine you have some rows of numbers and columns of numbers—this is a *matrix*. If you have two such matrices, you can *add* them by adding the matching numbers, or *multiply* them in a special way to get another set of rows and columns. It's kind of like mixing up ingredients in a recipe: you take certain parts of one and certain parts of another, and you get a brand-new result.

Eigenvectors (pronounced "eye-gen-vectors") are special directions inside a matrix that don't get turned around when you apply the matrix—like a magic arrow that points in a direction that never changes no matter what you do. These are super important in math and help us solve puzzles about how data moves, changes, or scales.

All this is important in computers because we need to handle pictures, words, and numbers. If you want a computer to learn to recognize cats in pictures (like in a phone app or in YouTube), it uses

these matrix ideas to figure out patterns. That's what deep learning is about—using math to teach computers to learn from examples!

Digging Deeper

Linear algebra is the study of lines, planes, and volumes in many dimensions, but in practical terms, we often represent those things with *matrices* (grids of numbers). Matrix multiplication is more than just multiplying numbers—it's a set of rules for combining rows of one matrix with columns of another, to produce a new grid of numbers.

An *eigenvector* is a vector that, when a matrix acts on it, just gets scaled by some factor (the *eigenvalue*) instead of changing direction. This concept is used in many algorithms, including finding patterns or compressing large data sets.

In **deep learning** (the technology behind LLMs, image recognition, etc.), we use these matrix operations to transform data step by step, so the computer can learn complex functions that can classify images or generate new text.

Matrices are at the heart of linear transformations. If A is an $m \times n$ matrix and x is an n-dimensional vector, then the product Ax is an m-dimensional vector. This operation is fundamental because it represents linear transformations from an n-dimensional space to an m-dimensional space.

For eigenvectors, formally, if A is a square matrix, an eigenvector v satisfies:

$$Av = \lambda v$$

for some scalar λ called the *eigenvalue*. If you think about it, applying the transformation A to v results in the same direction, but possibly a different magnitude.

In deep learning, we stack multiple linear transformations (plus non-linear activations). Weight matrices hold the parameters that get learned during training, and we combine them with biases and activation functions. The power of deep networks comes from layering many transformations so we can approximate very complex functions.

0.2.1 Level D: Graduate Explorers

We delve deeper into spectral decompositions, diagonalization, and the concept of covariance matrices. In machine learning, the decomposition of the covariance matrix into eigenvectors (Principal Component Analysis) helps reduce dimensionality. For a real symmetric matrix (A), the spectral theorem ensures an orthogonal eigenbasis.

Eigenvalues also play a critical role in the optimization of neural networks—understanding the Hessian matrix's eigenvalues can tell us about the curvature of the loss surface. Large eigenvalues might indicate a steep direction, which can lead to instability in gradient-based optimization.

In large language models (LLMs), these transformations become extremely high-dimensional. Techniques such as matrix factorization, attention mechanisms, and advanced optimizations rely on heavy usage of linear algebra.

Research in deep learning architecture design often revolves around advanced linear algebra concepts. For example, analyzing the expressivity of neural networks can involve looking at the rank of weight matrices in each layer, or studying the effect of regularization on the singular values.

Generative models like Variational Autoencoders (VAEs) or Diffusion Models rely on transformations that are linear plus carefully selected non-linear components, optimized via high-dimensional gradient-based methods.

Eigenvalues and eigenvectors are integral to stability and expressivity analyses, as well as to compressed representations (e.g., matrix/tensor decompositions). The entire field of model compression (pruning, factorization, low-rank approximation) is heavily grounded in linear algebra.

0.3 2. Intuition Behind the Technology

Why does linear algebra show up everywhere in deep learning? Because everything we do with data—whether images, text, or signals—eventually gets converted into numbers (vectors). A neural network is essentially a sequence of matrix multiplications plus some squishing functions (activations). If the matrix multiplications are well-chosen (trained with data), they can do everything from identifying spam emails to writing poetry.

0.4 3. Brief History: Invention and Underlying Tech

- **Linear Algebra** has roots going back to the study of systems of linear equations (e.g., solutions to multiple equations with multiple unknowns). Mathematicians like Gauss, Euler, and others formalized these ideas.
- **Matrix multiplication** was well-understood in the 19th century, with further expansions in the 20th century—leading to the concept of transformations in higher dimensions.
- **Eigenvalues and eigenvectors** were studied intensively by mathematicians like Hilbert and others in the context of linear transformations, diagonalization, and functional analysis.
- **Neural Networks** (1940s – 1960s) started with perceptrons and continued with multi-layer perceptrons in the 1980s and 1990s, culminating in "Deep Learning" breakthroughs around 2012 when large-scale matrix operations (on GPUs) and huge datasets became feasible.
- **Generative AI & LLMs** rose with the advent of the Transformer architecture (2017), which relies on linear algebra for multi-head attention, matrix factorization, and large-scale optimization.

Today, we see that powerful computing on large matrices/tensors is key to advanced AI.

0.5 4. Underlying Tech (How It Works)

Modern AI frameworks like PyTorch or TensorFlow are optimized for *tensor operations*, which are generalizations of matrices to higher dimensions. Under the hood, you're effectively doing a bunch of matrix multiplications and additions, all accelerated by GPUs.

0.5.1 Key Steps in Using Linear Algebra for Deep Learning:

1. **Represent** your data as vectors/matrices/tensors.
2. **Multiply** by weight matrices (learnable parameters) and **add** biases.
3. **Apply** a non-linear transformation (activation function).
4. **Stack** multiple layers (matrix + activation) to build *depth*.
5. **Calculate** a loss function.

6. **Use** gradient descent or similar algorithms to update weight matrices.

0.6 5. ELI5 Math: Building the Foundations

0.6.1 Matrix Multiplication

If you have an $m \times n$ matrix A and an $n \times p$ matrix B, the product $C = AB$ is an $m \times p$ matrix. The entry C_{ij} is computed by taking the i-th row of A and the j-th column of B, multiplying corresponding elements, and summing up:

$$C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}$$

0.6.2 Eigenvalues and Eigenvectors

For a square matrix A $(n \times n)$, an eigenvector $v \neq 0$ and an eigenvalue λ satisfy:

$$Av = \lambda v$$

This means v remains in the same *direction* after being transformed by A, but is scaled by λ.

0.6.3 Weights and Biases in a Simple Neural Network

- A **weight** matrix W transforms input vectors into new vectors.
- A **bias** vector b shifts the output.
- Example layer output: $\text{out} = Wx + b$

0.6.4 Connection to Deep Learning

Deep neural networks chain these transformations together, e.g.:

$$x^{(1)} = f(W^{(1)} x^{(0)} + b^{(1)})$$

$$x^{(2)} = f(W^{(2)} x^{(1)} + b^{(2)})$$

...

and so on, where f is an activation function (like ReLU).

0.7 6. Illustrative Example with PyTorch (ELI5 Style)

Let's do some basic matrix operations and find eigenvalues/eigenvectors using **PyTorch**. We'll keep it simple so everyone can see how it works.

[19]:
```
import torch
import math

# Let's create two matrices A and B
A = torch.tensor([[1.0, 2.0],
                  [3.0, 4.0]])
B = torch.tensor([[2.0, 0.0],
                  [1.0, 2.0]])

print('Matrix A:\n', A)
print('Matrix B:\n', B)

# Matrix multiplication
C = torch.matmul(A, B)
print('\nA x B = C:\n', C)
```

```
Matrix A:
 tensor([[1., 2.],
        [3., 4.]])
Matrix B:
 tensor([[2., 0.],
        [1., 2.]])

A x B = C:
 tensor([[ 4.,  4.],
        [10.,  8.]])
```

Next, let's compute the eigenvalues and eigenvectors of a matrix. We'll pick **A** for that example.

[20]:
```
# Eigen decomposition of A
eigenvalues, eigenvectors = torch.linalg.eig(A)

print('Eigenvalues of A:', eigenvalues)
print('Eigenvectors of A:\n', eigenvectors)

# Let's verify the first eigenpair by a manual matrix multiplication
v = eigenvectors[:,0]  # first eigenvector
lambda_v = eigenvalues[0]

# A v should be approximately lambda_v * v
# Av = torch.matmul(A, v) #TODO- whats wrong here?
#HINT- # may be Cast A to complex float to perform the multiplication with a␣
 ↪complex vector
Av = torch.matmul(A.to(torch.complex64), v)
lambda_v_times_v = lambda_v * v

print('\nCheck A * v vs lambda_v * v')
print('A * v =', Av)
```

```
print('lambda_v * v =', lambda_v_times_v)
```

```
Eigenvalues of A: tensor([-0.3723+0.j,  5.3723+0.j])
Eigenvectors of A:
 tensor([[-0.8246+0.j, -0.4160+0.j],
        [ 0.5658+0.j, -0.9094+0.j]])

Check A * v vs lambda_v * v
A * v = tensor([ 0.3070+0.j, -0.2106+0.j])
lambda_v * v = tensor([ 0.3070-0.j, -0.2106+0.j])
```

> **Note**: The eigenvalues might be complex if the matrix is not perfectly symmetric or if it's not diagonalizable in real numbers. In our case, matrix (A) is real and should yield real eigenvalues (though with floating point computations, you might see small imaginary parts).

0.8 7. Example Calculations & Terms

1. **Weight (W)**: Think of it as a scaling factor or a matrix that changes the shape of your data.
2. **Bias (b)**: A shift term added to the multiplication result.
3. **Forward Pass**: The process where you compute the output of a network by applying all transformations in sequence.
4. **Loss Function**: The measure of how far your predictions are from the truth.
5. **Backpropagation**: How the network adjusts weights and biases by calculating gradients of the loss.

0.8.1 Example of a single-layer neural network pass:

$$y = Wx + b$$

If x is a 3-dimensional input, W might be a 2×3 matrix, and b a 2-dimensional vector—thus the output y is 2-dimensional.

0.9 8. Step by Step Example: Building a Tiny Neural Network from Scratch

We'll create a toy example in PyTorch that: 1. Takes an input matrix (like our feature set), 2. Multiplies by a weight matrix, adds bias, 3. Uses a ReLU activation, 4. Computes a simple loss function, 5. Performs a gradient descent update.

0.9.1 Illustrative Problem

Let's say we have 2D input data (two features per data point) and we want to predict a single output (like a regression problem). We'll create random data and see how the network learns.

0.9.2 Real World Problem

This could correspond to something like predicting the price of a product based on two features: weight and height, or predicting a student's test score based on two features: hours studied and hours slept, etc. The concept is the same.

0.10 9. Implementation in PyTorch

We'll do a small demonstration. We'll create a random dataset, define a single-layer network, define a loss, and run a few steps of gradient descent.

[21]:
```
import torch
import torch.nn as nn
import torch.optim as optim

torch.manual_seed(42)

# Generate some random data
X = torch.randn(100, 2)  # 100 data points, each with 2 features
# Let's define a ground truth linear model: y = 3*x1 - 2*x2 + 1
true_W = torch.tensor([[3.0], [-2.0]])  # shape (2,1)
true_b = 1.0
y = X @ true_W + true_b  # shape (100, 1)

# We'll create a simple linear model
model = nn.Sequential(
    nn.Linear(2, 1),  # from 2 input features to 1 output
)

# Define a loss function and optimizer
criterion = nn.MSELoss()
optimizer = optim.SGD(model.parameters(), lr=0.01)

# Training loop (gradient descent)
for epoch in range(200):
    optimizer.zero_grad()
    predictions = model(X)
    loss = criterion(predictions, y)
    loss.backward()
    optimizer.step()

# Let's inspect the learned parameters
[W_learned, b_learned] = list(model.parameters())
print('Learned Weights:', W_learned)
print('Learned Bias:', b_learned)

print(f"Final Loss: {loss.item():.4f}")
```

```
Learned Weights: Parameter containing:
tensor([[ 2.9117, -1.8449]], requires_grad=True)
Learned Bias: Parameter containing:
tensor([0.8718], requires_grad=True)
Final Loss: 0.0310
```

Explanation: Over the training, the network adjusts its weights and bias to match

the pattern in the data ((y = 3x_1 - 2x_2 + 1)). Ideally, after enough iterations, we should get close to `[3, -2]` for the weights and `1` for the bias.

0.11 10. Additional Questions (Points to Ponder)

1. What if you had 3D input data or 100D input data? How do matrix dimensions change?
2. How does adding more layers (more matrix multiplications) help the network learn more complex patterns?
3. What is the role of activation functions in ensuring the network can learn non-linear relationships?
4. How would you extend this to classification problems (e.g., cat vs dog images)?
5. What happens if you remove biases from the network?
6. How do you interpret eigenvalues and eigenvectors in the context of data transformations?
7. Why do we need GPUs for large matrix multiplications?

0.12 11. Answers & Explanations

1. **Higher-dimensional input**: If you have more features, your weight matrix changes shape. For example, with 3D input and 2D output, the weight matrix is (2×3).
2. **Deeper networks**: Stacking more layers adds more transformations, giving the network the ability to learn more complex functions.
3. **Activation functions**: Without activations, multiple linear transformations would collapse into a single linear transformation. Activations create non-linearity, allowing the network to capture complex relationships.
4. **Classification**: You'd typically use a softmax layer at the end and a cross-entropy loss. The matrix dimension for the last layer equals the number of classes.
5. **No biases**: The model might be forced through the origin, possibly limiting its ability to fit data with offsets.
6. **Eigenvalues/eigenvectors**: In data transformations (e.g., PCA), the eigenvectors of the covariance matrix represent principal directions. In transformations, they indicate directions that don't change, only scale.
7. **GPUs**: They are highly optimized for parallel computations, such as large-scale matrix multiplications. Without GPUs, training large networks would be prohibitively slow.

0.13 12. An Illustrative Example

This snippet focuses on **matrix multiplication** and **eigen decomposition** in PyTorch.

[22]:
```
# TODO: Fill in the missing parts (look for the TODO comments)
import torch

# TODO 1: Create a 3x3 matrix named M with any numbers you like
M = torch.tensor([[1, 2, 3],
                  [4, 5, 6],
                  [7, 8, 9]], dtype=torch.float)

# TODO 2: Print M
```

```
print('Matrix M:\n', M)

# TODO 3: Create another 3x3 matrix N
N = torch.tensor([[9, 8, 7],
                  [6, 5, 4],
                  [3, 2, 1]], dtype=torch.float)

# Multiply M and N
MN = torch.matmul(M, N)
print('\nM x N = \n', MN)

# TODO 4: Compute the eigenvalues and eigenvectors of M
vals, vecs = torch.linalg.eig(M)
print('\nEigenvalues of M:', vals)
print('Eigenvectors of M:\n', vecs)

# TODO 5: Verify the first eigenpair. (We already did this above, replicate the␣
 ↪logic)
v_first = vecs[:,0]
lambda_first = vals[0]
Mv = torch.matmul(M.to(torch.complex64), v_first)
lambda_v = lambda_first * v_first
print('\nCheck M * v vs lambda_v * v')
print('M * v =', Mv)
print('lambda_v * v =', lambda_v)
```

```
Matrix M:
 tensor([[1., 2., 3.],
        [4., 5., 6.],
        [7., 8., 9.]])

M x N =
 tensor([[ 30.,  24.,  18.],
        [ 84.,  69.,  54.],
        [138., 114.,  90.]])

Eigenvalues of M: tensor([ 1.6117e+01+0.j, -1.1168e+00+0.j,  2.9486e-07+0.j])
Eigenvectors of M:
 tensor([[-0.2320+0.j, -0.7858+0.j,  0.4082+0.j],
        [-0.5253+0.j, -0.0868+0.j, -0.8165+0.j],
        [-0.8187+0.j,  0.6123+0.j,  0.4082+0.j]])

Check M * v vs lambda_v * v
M * v = tensor([ -3.7386+0.j,  -8.4665+0.j, -13.1944+0.j])
lambda_v * v = tensor([ -3.7386+0.j,  -8.4665+0.j, -13.1944+0.j])
```

This simple exercise helps reinforce the following concepts: - Creating matrices in PyTorch. - Performing matrix multiplication. - Calculating eigenvalues and eigenvectors. - Verifying the eigen

decomposition manually.

0.14 13. Glossary

- **Matrix**: A 2D array of numbers.
- **Matrix Multiplication (matmul)**: A specific rule for multiplying 2D arrays of compatible shapes.
- **Vector**: A 1D array of numbers.
- **Eigenvector**: A special vector that only gets scaled (not rotated) by a matrix transformation.
- **Eigenvalue**: The factor by which an eigenvector is scaled.
- **Weight Matrix**: The parameters in a neural network layer that multiply the input.
- **Bias**: An additive constant in each layer.
- **Activation Function**: A non-linear function applied after linear transformations (e.g., ReLU, Sigmoid).
- **Loss Function**: A measure of how well the model's predictions match the true labels.
- **Gradient Descent**: An optimization algorithm that adjusts parameters to reduce the loss.
- **PyTorch**: A popular deep learning framework that uses tensors for computation.

Conclusion

We've walked through linear algebra fundamentals, matrix operations, and how they apply to neural networks—from the ground up to advanced research levels. Understanding these core concepts is crucial to effec-

End of Notebook

[23]:
```python
import os, sys, platform, datetime, uuid, socket

def signoff(name="Anonymous"):
    colab_check = "Yes" if 'google.colab' in sys.modules else "No"
    mac_addr = ':'.join(format((uuid.getnode() >> i) & 0xff, '02x') for i in␣
 ↪reversed(range(0, 48, 8)))
    print("+++ Acknowledgement +++")
    print(f"Executed on: {datetime.datetime.now()}")
    print(f"In Google Colab: {colab_check}")
    print(f"System info: {platform.system()} {platform.release()}")
    print(f"Node name: {platform.node()}")
    print(f"MAC address: {mac_addr}")
    try:
        print(f"IP address: {socket.gethostbyname(socket.gethostname())}")
    except:
        print("IP address: Unknown")
    print(f"Signing off, {name}")

signoff("Ali Muhammad Asad")
```

```
+++ Acknowledgement +++
Executed on: 2025-01-27 15:10:16.225127
In Google Colab: No
System info: Linux 6.8.0-51-generic
Node name: alimuhammad-Inspiron-7559
MAC address: 20:47:47:74:94:05
IP address: 127.0.1.1
Signing off, Ali Muhammad Asad
```