assignment_1

January 18, 2026

1 Assignment 1: Mathematical Foundations of Deep Learning

Objective: To implement and understand the linear algebra, calculus, and probability concepts that form the mathematical foundation of deep learning, including vectors, matrices, gradients, entropy, and KL divergence.

1.1 Part 1: NumPy Implementations

1.1.1 Import Required Libraries

[ ]:
```
import numpy as np
import torch

print("NumPy version:", np.__version__)
print("PyTorch version:", torch.__version__)
```

```
NumPy version: 2.4.1
PyTorch version: 2.9.1
```

1.1.2 1(i) Vector Operations with NumPy

Create two vectors and compute: - Dot product - L2 norm

[2]:
```
# Create two vectors
v1 = np.array([1, 2, 3, 4])
v2 = np.array([5, 6, 7, 8])

print("Vector v1:", v1)
print("Vector v2:", v2)

# Dot product
dot_product = np.dot(v1, v2)
print("\nDot product (v1 · v2):", dot_product)

# L2 norm (Euclidean norm)
l2_norm_v1 = np.linalg.norm(v1)
l2_norm_v2 = np.linalg.norm(v2)
print("\nL2 norm of v1:", l2_norm_v1)
```

```
print("L2 norm of v2:", l2_norm_v2)
```

```
Vector v1: [1 2 3 4]
Vector v2: [5 6 7 8]

Dot product (v1 · v2): 70

L2 norm of v1: 5.477225575051661
L2 norm of v2: 13.19090595827292
```

1.1.3 1(ii) Matrix Operations with NumPy

Create two matrices and compute: - Matrix multiplication - Eigenvalues

[3]:
```
# Create two matrices
A = np.array([[1, 2],
              [3, 4]])
B = np.array([[5, 6],
              [7, 8]])

print("Matrix A:")
print(A)
print("\nMatrix B:")
print(B)

# Matrix multiplication
matrix_product = np.matmul(A, B)  # or A @ B
print("\nMatrix multiplication (A × B):")
print(matrix_product)

# Eigenvalues (for square matrix A)
eigenvalues_A, eigenvectors_A = np.linalg.eig(A)
print("\nEigenvalues of A:", eigenvalues_A)
print("Eigenvectors of A:")
print(eigenvectors_A)

# Eigenvalues for matrix B
eigenvalues_B, eigenvectors_B = np.linalg.eig(B)
print("\nEigenvalues of B:", eigenvalues_B)
```

```
Matrix A:
[[1 2]
 [3 4]]

Matrix B:
[[5 6]
 [7 8]]

Matrix multiplication (A × B):
```

```
[[19 22]
 [43 50]]

Eigenvalues of A: [-0.37228132  5.37228132]
Eigenvectors of A:
[[-0.82456484 -0.41597356]
 [ 0.56576746 -0.90937671]]

Eigenvalues of B: [-0.15206735 13.15206735]
```

1.1.4 1(iii) Partial Derivatives and Gradient Computation

For the function $f(x, y) = x^2 + 3y^2 + 2xy$

Compute: - Partial derivative with respect to x: $\frac{\partial f}{\partial x} = 2x + 2y$ - Partial derivative with respect to y: $\frac{\partial f}{\partial y} = 6y + 2x$ - Gradient vector: $\nabla f = \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}\right)$

[4]:
```
# Define the function f(x, y) = x^2 + 3y^2 + 2xy
def f(x, y):
    return x**2 + 3*y**2 + 2*x*y

# Partial derivatives (analytically derived)
def partial_f_x(x, y):
    """Partial derivative of f with respect to x: ∂f/∂x = 2x + 2y"""
    return 2*x + 2*y

def partial_f_y(x, y):
    """Partial derivative of f with respect to y: ∂f/∂y = 6y + 2x"""
    return 6*y + 2*x

# Gradient vector
def gradient_f(x, y):
    """Gradient vector: ∇f = (∂f/∂x, ∂f/∂y)"""
    return np.array([partial_f_x(x, y), partial_f_y(x, y)])

# Example: Compute at point (x=1, y=2)
x, y = 1, 2

print(f"Function f(x,y) = x² + 3y² + 2xy")
print(f"\nAt point (x={x}, y={y}):")
print(f"f({x}, {y}) = {f(x, y)}")
print(f"\nPartial derivative ∂f/∂x = 2x + 2y = {partial_f_x(x, y)}")
print(f"Partial derivative ∂f/∂y = 6y + 2x = {partial_f_y(x, y)}")
print(f"\nGradient vector ∇f = {gradient_f(x, y)}")
```

```
Function f(x,y) = x² + 3y² + 2xy

At point (x=1, y=2):
f(1, 2) = 17
```

```
Partial derivative  f/ x = 2x + 2y = 6
Partial derivative  f/ y = 6y + 2x = 14

Gradient vector  f = [ 6 14]
```

1.1.5 1(iv) Entropy Calculation

Given a discrete probability distribution, compute Shannon Entropy:

$$H(P) = -\sum_i P(x_i) \log_2 P(x_i)$$

[5]:
```
# Define a discrete probability distribution
P = np.array([0.25, 0.25, 0.25, 0.25])  # Uniform distribution
print("Probability distribution P:", P)
print("Sum of probabilities:", np.sum(P))

# Entropy function: H(P) = -Σ P(x) * log2(P(x))
def entropy(p):
    """Compute Shannon entropy of a probability distribution"""
    # Filter out zero probabilities to avoid log(0)
    p = p[p > 0]
    return -np.sum(p * np.log2(p))

entropy_P = entropy(P)
print(f"\nEntropy H(P) = {entropy_P:.4f} bits")

# Example with a non-uniform distribution
Q = np.array([0.5, 0.25, 0.125, 0.125])
print(f"\nProbability distribution Q: {Q}")
entropy_Q = entropy(Q)
print(f"Entropy H(Q) = {entropy_Q:.4f} bits")

# Note: Uniform distribution has maximum entropy
print(f"\nNote: Uniform distribution has higher entropy ({entropy_P:.4f}) than␣
 ↪non-uniform ({entropy_Q:.4f})")
```

```
Probability distribution P: [0.25 0.25 0.25 0.25]
Sum of probabilities: 1.0

Entropy H(P) = 2.0000 bits

Probability distribution Q: [0.5   0.25  0.125 0.125]
Entropy H(Q) = 1.7500 bits

Note: Uniform distribution has higher entropy (2.0000) than non-uniform (1.7500)
```

1.1.6 1(v) Kullback-Leibler (KL) Divergence

Given two probability distributions P and Q, compute KL divergence:

$$D_{KL}(P\|Q) = \sum_i P(x_i) \log \frac{P(x_i)}{Q(x_i)}$$

KL divergence measures how one probability distribution diverges from a reference distribution.

[6]:
```
# Define two probability distributions
P = np.array([0.4, 0.3, 0.2, 0.1])
Q = np.array([0.25, 0.25, 0.25, 0.25])

print("Distribution P:", P)
print("Distribution Q:", Q)

# KL Divergence: D_KL(P || Q) = Σ P(x) * log(P(x) / Q(x))
def kl_divergence(p, q):
    """Compute KL divergence D_KL(P || Q)"""
    # Filter out zero probabilities in P
    mask = p > 0
    p = p[mask]
    q = q[mask]
    return np.sum(p * np.log(p / q))

kl_div_PQ = kl_divergence(P, Q)
print(f"\nKL Divergence D_KL(P || Q) = {kl_div_PQ:.4f}")

# KL divergence is not symmetric
kl_div_QP = kl_divergence(Q, P)
print(f"KL Divergence D_KL(Q || P) = {kl_div_QP:.4f}")

print(f"\nNote: KL divergence is NOT symmetric: D_KL(P||Q)  D_KL(Q||P)")
```

```
Distribution P: [0.4 0.3 0.2 0.1]
Distribution Q: [0.25 0.25 0.25 0.25]

KL Divergence D_KL(P || Q) = 0.1064
KL Divergence D_KL(Q || P) = 0.1218

Note: KL divergence is NOT symmetric: D_KL(P||Q)  D_KL(Q||P)
```

1.2 Part 2: PyTorch Implementations

Understanding and implementing tensor operations used in deep learning frameworks.

1.2.1 2(i) Creating Tensors with PyTorch

Create tensors of different dimensions: - Scalar (0D tensor) - Vector (1D tensor) - Matrix (2D tensor)

[7]:
```
# Scalar (0D tensor)
scalar = torch.tensor(5.0)
print("Scalar (0D tensor):", scalar)
print("Shape:", scalar.shape)
print("Dimension:", scalar.ndim)

# Vector (1D tensor)
vector = torch.tensor([1.0, 2.0, 3.0, 4.0])
print("\nVector (1D tensor):", vector)
print("Shape:", vector.shape)
print("Dimension:", vector.ndim)

# Matrix (2D tensor)
matrix = torch.tensor([[1.0, 2.0, 3.0],
                       [4.0, 5.0, 6.0]])
print("\nMatrix (2D tensor):")
print(matrix)
print("Shape:", matrix.shape)
print("Dimension:", matrix.ndim)

# 3D Tensor (for completeness)
tensor_3d = torch.tensor([[[1, 2], [3, 4]],
                          [[5, 6], [7, 8]]])
print("\n3D Tensor:")
print(tensor_3d)
print("Shape:", tensor_3d.shape)
print("Dimension:", tensor_3d.ndim)
```

```
Scalar (0D tensor): tensor(5.)
Shape: torch.Size([])
Dimension: 0

Vector (1D tensor): tensor([1., 2., 3., 4.])
Shape: torch.Size([4])
Dimension: 1

Matrix (2D tensor):
tensor([[1., 2., 3.],
        [4., 5., 6.]])
Shape: torch.Size([2, 3])
Dimension: 2

3D Tensor:
tensor([[[1, 2],
```

```
         [3, 4]],

        [[5, 6],
         [7, 8]]])
Shape: torch.Size([2, 2, 2])
Dimension: 3
```

1.2.2 2(ii) Basic Tensor Operations in PyTorch

Perform: - Addition - Matrix multiplication - Dot product - L2 norm

[8]:
```
# Create tensors for operations
t1 = torch.tensor([1.0, 2.0, 3.0, 4.0])
t2 = torch.tensor([5.0, 6.0, 7.0, 8.0])

print("Tensor t1:", t1)
print("Tensor t2:", t2)

# Addition
addition = t1 + t2  # or torch.add(t1, t2)
print("\n1. Addition (t1 + t2):", addition)

# Create matrices for matrix multiplication
A = torch.tensor([[1.0, 2.0],
                  [3.0, 4.0]])
B = torch.tensor([[5.0, 6.0],
                  [7.0, 8.0]])

print("\nMatrix A:")
print(A)
print("\nMatrix B:")
print(B)

# Matrix multiplication
mat_mul = torch.matmul(A, B)  # or A @ B
print("\n2. Matrix multiplication (A @ B):")
print(mat_mul)

# Dot product
dot_product = torch.dot(t1, t2)
print("\n3. Dot product (t1 · t2):", dot_product)

# L2 norm
l2_norm_t1 = torch.norm(t1)
l2_norm_t2 = torch.norm(t2)
print("\n4. L2 norm of t1:", l2_norm_t1)
print("   L2 norm of t2:", l2_norm_t2)
```

```
Tensor t1: tensor([1., 2., 3., 4.])
Tensor t2: tensor([5., 6., 7., 8.])

1. Addition (t1 + t2): tensor([ 6.,  8., 10., 12.])

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

Matrix B:
tensor([[5., 6.],
        [7., 8.]])

2. Matrix multiplication (A @ B):
tensor([[19., 22.],
        [43., 50.]])

3. Dot product (t1 · t2): tensor(70.)

4. L2 norm of t1: tensor(5.4772)
   L2 norm of t2: tensor(13.1909)
```

1.2.3 2(iii) Gradient Computation with Autograd

Use PyTorch's autograd to compute gradients of a scalar-valued function automatically.

For the same function: $f(x, y) = x^2 + 3y^2 + 2xy$

[9]:
```python
# Create tensors with requires_grad=True to track gradients
x = torch.tensor(1.0, requires_grad=True)
y = torch.tensor(2.0, requires_grad=True)

print(f"x = {x.item()}, requires_grad = {x.requires_grad}")
print(f"y = {y.item()}, requires_grad = {y.requires_grad}")

# Define the function f(x, y) = x^2 + 3y^2 + 2xy
f = x**2 + 3*y**2 + 2*x*y

print(f"\nFunction f(x, y) = x² + 3y² + 2xy")
print(f"f({x.item()}, {y.item()}) = {f.item()}")

# Compute gradients using backward()
f.backward()

# Gradients are stored in .grad attribute
print(f"\nGradients computed by autograd:")
print(f" f/ x = 2x + 2y = {x.grad.item()}")
print(f" f/ y = 6y + 2x = {y.grad.item()}")
```

```
# Verify with analytical solution
print(f"\nVerification (analytical):")
print(f" f/ x at (1, 2) = 2(1) + 2(2) = {2*1 + 2*2}")
print(f" f/ y at (1, 2) = 6(2) + 2(1) = {6*2 + 2*1}")
```

```
x = 1.0, requires_grad = True
y = 2.0, requires_grad = True

Function f(x, y) = x² + 3y² + 2xy
f(1.0, 2.0) = 17.0

Gradients computed by autograd:
 f/ x = 2x + 2y = 6.0
 f/ y = 6y + 2x = 14.0

Verification (analytical):
 f/ x at (1, 2) = 2(1) + 2(2) = 6
 f/ y at (1, 2) = 6(2) + 2(1) = 14
```

1.2.4 2(iv) Gradient Computation Modes Comparison

Demonstrate the difference between: - `requires_grad=True` vs `requires_grad=False` - `torch.no_grad()` context - `tensor.detach()`

[10]:
```
# 1. requires_grad=True vs requires_grad=False
print("=" * 50)
print("1. requires_grad=True vs requires_grad=False")
print("=" * 50)

# With requires_grad=True - gradient tracking enabled
x_grad = torch.tensor(2.0, requires_grad=True)
y_grad = x_grad ** 2
print(f"\nWith requires_grad=True:")
print(f"x = {x_grad.item()}, x.requires_grad = {x_grad.requires_grad}")
print(f"y = x² = {y_grad.item()}, y.requires_grad = {y_grad.requires_grad}")
print(f"y.grad_fn = {y_grad.grad_fn}")

# With requires_grad=False - no gradient tracking
x_no_grad = torch.tensor(2.0, requires_grad=False)
y_no_grad = x_no_grad ** 2
print(f"\nWith requires_grad=False:")
print(f"x = {x_no_grad.item()}, x.requires_grad = {x_no_grad.requires_grad}")
print(f"y = x² = {y_no_grad.item()}, y.requires_grad = {y_no_grad.
 ↪requires_grad}")
print(f"y.grad_fn = {y_no_grad.grad_fn}")

# 2. torch.no_grad() context
```

```
print("\n" + "=" * 50)
print("2. torch.no_grad() context")
print("=" * 50)

x = torch.tensor(3.0, requires_grad=True)
print(f"\nBefore no_grad: x.requires_grad = {x.requires_grad}")

# Inside torch.no_grad(), gradient computation is disabled
with torch.no_grad():
    y = x ** 2
    print(f"Inside torch.no_grad():")
    print(f"  y = x² = {y.item()}")
    print(f"  y.requires_grad = {y.requires_grad}")
    print(f"  y.grad_fn = {y.grad_fn}")

# Outside torch.no_grad(), gradient computation resumes
y_outside = x ** 2
print(f"\nOutside torch.no_grad():")
print(f"  y = x² = {y_outside.item()}")
print(f"  y.requires_grad = {y_outside.requires_grad}")
print(f"  y.grad_fn = {y_outside.grad_fn}")

# 3. tensor.detach()
print("\n" + "=" * 50)
print("3. tensor.detach()")
print("=" * 50)

x = torch.tensor(4.0, requires_grad=True)
y = x ** 2
print(f"\nOriginal tensor y:")
print(f"  y = {y.item()}, y.requires_grad = {y.requires_grad}")

# detach() creates a new tensor that shares data but doesn't require gradients
y_detached = y.detach()
print(f"\nDetached tensor y_detached = y.detach():")
print(f"  y_detached = {y_detached.item()}, y_detached.requires_grad =␣
 ↪{y_detached.requires_grad}")
print(f"  y_detached.grad_fn = {y_detached.grad_fn}")

# Summary
print("\n" + "=" * 50)
print("Summary:")
print("=" * 50)
print("""
- requires_grad=True: Enables gradient tracking for the tensor
- requires_grad=False: Disables gradient tracking (default for most tensors)
```

```
- torch.no_grad(): Context manager that temporarily disables gradient␣
 ↪computation
                   (useful during inference to save memory and computation)
- tensor.detach(): Creates a new tensor detached from the computation graph
                   (shares data but won't track gradients)
""")
```

```
==================================================
1. requires_grad=True vs requires_grad=False
==================================================

With requires_grad=True:
x = 2.0, x.requires_grad = True
y = x² = 4.0, y.requires_grad = True
y.grad_fn = <PowBackward0 object at 0x127de71c0>

With requires_grad=False:
x = 2.0, x.requires_grad = False
y = x² = 4.0, y.requires_grad = False
y.grad_fn = None

==================================================
2. torch.no_grad() context
==================================================

Before no_grad: x.requires_grad = True
Inside torch.no_grad():
  y = x² = 9.0
  y.requires_grad = False
  y.grad_fn = None

Outside torch.no_grad():
  y = x² = 9.0
  y.requires_grad = True
  y.grad_fn = <PowBackward0 object at 0x127de71c0>

==================================================
3. tensor.detach()
==================================================

Original tensor y:
  y = 16.0, y.requires_grad = True

Detached tensor y_detached = y.detach():
  y_detached = 16.0, y_detached.requires_grad = False
  y_detached.grad_fn = None

==================================================
```

```
Summary:
==================================================

- requires_grad=True: Enables gradient tracking for the tensor
- requires_grad=False: Disables gradient tracking (default for most tensors)
- torch.no_grad(): Context manager that temporarily disables gradient
computation
                    (useful during inference to save memory and computation)
- tensor.detach(): Creates a new tensor detached from the computation graph
                    (shares data but won't track gradients)
```

1.3 Conclusion

This assignment covered the fundamental mathematical concepts used in deep learning:

Part 1 (NumPy): - Vector operations: dot product and L2 norm - Matrix operations: multiplication and eigenvalues - Calculus: partial derivatives and gradient computation for f(x,y) = x^2 + $3y^2$ + 2xy - Probability: Shannon entropy and KL divergence

Part 2 (PyTorch): - Tensor creation with different dimensions - Basic tensor operations (addition, matrix multiplication, dot product, L2 norm) - Automatic differentiation using autograd - Understanding gradient computation modes (requires_grad, no_grad, detach)