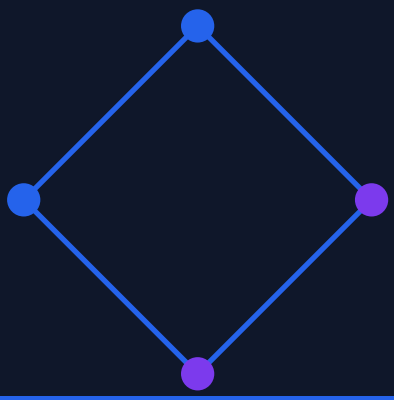
PMAI LAB

Mathematics • Python • Artificial Intelligence



LINEAR ALGEBRA FOR MACHINE LEARNING

Lecture 09

Neural Networks, Embeddings, Attention, and Gradient-Based Learning

Lecture focus Using matrices to represent neural-network layers, learned embeddings, attention, and the forward and backward computations used to train modern machine-learning models.

Dr. Hira Benish

Associate Professor of Mathematics

Student Learning Handout | Includes NumPy practice and review exercises

1 Lecture Overview

Modern neural networks repeatedly apply matrix multiplication, vector addition, nonlinear activation, and optimization. Embedding models convert discrete items into vectors, while attention uses matrix products to compare and combine information across a sequence.

This lecture connects these widely used machine-learning operations with linear transformations, inner products, norms, projections, and gradients developed earlier in the course.

Learning Outcomes

After completing this lecture, students should be able to:

- express a dense neural-network layer as an affine transformation;
- track matrix dimensions through a batched forward pass;
- explain why nonlinear activations are required in deep networks;
- calculate a small neural-network output by hand;
- interpret embeddings as learned coordinates in a vector space;
- compare embeddings using dot product and cosine similarity;
- construct query, key, and value matrices for attention;
- derive gradients of a linear layer using matrix calculus; and
- implement a forward pass, backpropagation step, and attention in NumPy.

1.1 Linear-Algebra Map of the Lecture

Machine-learning object	Linear-algebra operation
Dense layer	Matrix multiplication plus a bias vector
Embedding	Row selection from a learned matrix
Similarity	Dot product, cosine, or distance
Attention	Matrix of pairwise dot products and weighted sums
Backpropagation	Chain rule with transposed matrix products
Parameter update	Movement opposite the gradient vector

2 Dense Layers as Affine Transformations

Let a batch contain n observations with d_{in} input features. A dense layer with d_{out} units computes

$$Z = XW + \mathbf{1}\mathbf{b}^T,$$

where

$$X \in \mathbb{R}^{n \times d_{\text{in}}}, \quad W \in \mathbb{R}^{d_{\text{in}} \times d_{\text{out}}}, \quad \mathbf{b} \in \mathbb{R}^{d_{\text{out}}}, \quad Z \in \mathbb{R}^{n \times d_{\text{out}}}.$$

Every row of X is one observation. Every column of W contains the weights for one output unit. The bias is added to every row by broadcasting.

2.1 One Observation

For a row vector $\mathbf{x}^T$,

$$\mathbf{z}^T = \mathbf{x}^T W + \mathbf{b}^T.$$

The jth output is

$$z_j = \sum_{i=1}^{d_{\text{in}}} x_i w_{ij} + b_j.$$

Thus, each neuron forms a weighted linear combination followed by a translation.

2.2 Linear versus Affine

The map $X \mapsto XW$ is linear. Adding a nonzero bias makes the layer affine because the origin is no longer mapped to the origin. Neural-network terminology often calls both cases “linear layers,” but the distinction is mathematically important.

Shape Discipline

Before multiplying matrices, write their dimensions. The inner dimensions must agree, and the remaining dimensions determine the output:

$$(n \times d_{\text{in}})(d_{\text{in}} \times d_{\text{out}}) = n \times d_{\text{out}}.$$

Most implementation errors in neural networks are dimension or broadcasting errors.

Machine-Learning Connection

A dense layer learns useful directions in the input feature space. Its parameter count is $d_{\text{in}} d_{\text{out}} + d_{\text{out}}$, including weights and biases.

3 Deep Networks and Nonlinear Activations

A two-layer network may be written as

$$H = \phi\big(XW_1 + \mathbf{1}\mathbf{b}_1^T\big), \qquad \widehat{Y} = HW_2 + \mathbf{1}\mathbf{b}_2^T,$$

where ϕ acts element by element.

Common activations include

$$\mathrm{ReLU}(z) = \max(0, z), \qquad \sigma(z) = \frac{1}{1+e^{-z}}, \qquad \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}.$$

3.1 Why Nonlinearity Is Essential

Without an activation, two matrix layers collapse into one:

$$(XW_1 + \mathbf{1}\mathbf{b}_1^T)W_2 + \mathbf{1}\mathbf{b}_2^T = X(W_1W_2) + \mathbf{1}(\mathbf{b}_1^TW_2 + \mathbf{b}_2^T).$$

The result is still only an affine transformation, regardless of the number of layers. Nonlinear activations allow the network to represent curved and piecewise decision boundaries.

3.2 Shape Tracking in a Two-Layer Network

Quantity	Shape	Meaning
X	$n \times d$	Input batch
W_1	$d \times h$	Input-to-hidden weights
H	$n \times h$	Hidden representation
W_2	$h \times c$	Hidden-to-output weights
$\widehat{Y}$	$n \times c$	Predictions or output scores

The hidden width h is a model-design choice. Increasing it gives more representational capacity but also increases memory, computation, and the risk of overfitting.

Machine-Learning Connection

In practice, a batch is processed as one matrix operation. Vectorization uses optimized linear-algebra libraries and is much faster than looping over observations one at a time.

4 Worked Neural-Network Forward Pass

Consider one input observation and a network with two hidden units:

$$\mathbf{x}^T = \begin{bmatrix} 1 & 2 \end{bmatrix}, \qquad W_1 = \begin{bmatrix} 1 & -1 \\ 0.5 & 2 \end{bmatrix}, \qquad \mathbf{b}_1^T = \begin{bmatrix} 0 & 1 \end{bmatrix}.$$

4.1 Hidden Pre-Activation

$$\mathbf{z}_1^T = \mathbf{x}^T W_1 + \mathbf{b}_1^T = \begin{bmatrix} 1 & 2 \end{bmatrix} \begin{bmatrix} 1 & -1 \\ 0.5 & 2 \end{bmatrix} + \begin{bmatrix} 0 & 1 \end{bmatrix} = \begin{bmatrix} 2 & 4 \end{bmatrix}.$$

Apply ReLU:

$$\mathbf{h}^T = \mathrm{ReLU}(\mathbf{z}_1^T) = \begin{bmatrix} 2 & 4 \end{bmatrix}.$$

4.2 Output Layer

Let

$$W_2 = \begin{bmatrix} 1 \\ -0.5 \end{bmatrix}, \qquad b_2 = 0.5.$$

Then

$$\widehat{y} = \mathbf{h}^T W_2 + b_2 = \begin{bmatrix} 2 & 4 \end{bmatrix} \begin{bmatrix} 1 \\ -0.5 \end{bmatrix} + 0.5 = 0.5.$$

If the target is $y = 1.5$, the half-squared error is

$$L = \frac{1}{2}(\widehat{y} - y)^2 = \frac{1}{2}(0.5 - 1.5)^2 = 0.5.$$

Interpretation

The first layer changes the coordinates and creates two learned features. ReLU retains their positive parts. The second layer combines the learned features into a prediction. Training changes W_1, $\mathbf{b}_1$, W_2, and b_2 to reduce the loss.

Keep row-vector and column-vector conventions consistent. Equivalent formulas may look transposed in different textbooks or software libraries.

5 Embeddings and Vector Similarity

An embedding assigns each discrete object a learnable vector. For a vocabulary of V tokens and embedding dimension d, store

$$E \in \mathbb{R}^{V \times d}.$$

The ith row $E_{i,:}$ is the embedding of token i. If $\mathbf{e}_i$ is the ith one-hot vector, then

$$\mathbf{e}_i^T E = E_{i,:}.$$

An embedding lookup is therefore equivalent to multiplying a one-hot vector by E, but direct row selection is more efficient.

5.1 Comparing Embeddings

For embeddings $\mathbf{u}$ and $\mathbf{v}$,

$$\text{dot similarity} = \mathbf{u}^T\mathbf{v},$$

$$\text{cosine similarity} = \frac{\mathbf{u}^T\mathbf{v}}{\|\mathbf{u}\|_2\|\mathbf{v}\|_2}.$$

The dot product depends on direction and magnitude. Cosine similarity compares direction after normalizing length. The appropriate measure depends on how the model was trained.

5.2 Small Example

Let

$$\mathbf{u} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}, \qquad \mathbf{v} = \begin{bmatrix} 2 \\ 1 \end{bmatrix}.$$

Then

$$\mathbf{u}^T\mathbf{v} = 4, \qquad \cos(\mathbf{u}, \mathbf{v}) = \frac{4}{\sqrt{5}\sqrt{5}} = 0.8.$$

Machine-Learning Connection

Embeddings represent words, products, users, images, graph nodes, and other objects in continuous vector spaces. Nearby vectors often indicate similar behavior or context, but meaning is learned from the training objective and data.

Connection to Matrix Factorization

Learned embeddings are closely related to low-rank factorization. A large interaction or co-occurrence matrix is represented through smaller matrices whose rows provide compact latent coordinates.

6 Attention as Matrix Computation

Let $X \in \mathbb{R}^{n \times d_{\text{model}}}$ contain n token embeddings. Attention first produces queries, keys, and values:

$$Q = XW_Q, \qquad K = XW_K, \qquad V = XW_V.$$

With $Q, K \in \mathbb{R}^{n \times d_k}$ and $V \in \mathbb{R}^{n \times d_v}$, scaled dot-product attention is

$$S = \frac{QK^T}{\sqrt{d_k}}, \qquad A = \text{softmax}_{\text{row}}(S), \qquad Y = AV.$$

6.1 What Each Matrix Does

Matrix	Shape	Interpretation
QK^T	$n \times n$	Pairwise query-key compatibility scores
A	$n \times n$	Nonnegative row weights that sum to 1
V	$n \times d_v$	Information available for aggregation
Y	$n \times d_v$	Context-dependent weighted combinations

6.2 Why Divide by $\sqrt{d_k}$?

When vector dimension grows, dot products can become large in magnitude. Scaling keeps the scores in a more stable range before softmax and helps prevent extremely peaked probabilities and weak gradients.

6.3 Masks and Multi-Head Attention

A mask sets selected scores to a very negative value before softmax so that their weights become approximately zero. Multi-head attention repeats the calculation with different learned projection matrices, then combines the resulting representations.

Stable Softmax

For a score row **s**, compute

$$\text{softmax}(\mathbf{s})_i = \frac{e^{s_i - m}}{\sum_j e^{s_j - m}}, \qquad m = \max_j s_j.$$

Subtracting the maximum does not change the probabilities and prevents numerical overflow.

7 Backpropagation and Matrix Gradients

For a linear layer

$$Z = XW + \mathbf{1}\mathbf{b}^T,$$

suppose the upstream gradient is

$$G = \frac{\partial L}{\partial Z} \in \mathbb{R}^{n \times d_{\text{out}}}.$$

Then

$$\frac{\partial L}{\partial W} = X^T G, \qquad \frac{\partial L}{\partial \mathbf{b}} = G^T \mathbf{1}, \qquad \frac{\partial L}{\partial X} = G W^T.$$

These formulas show why transposes appear naturally in backpropagation. Each gradient has the same shape as the variable it updates.

7.1 Passing through an Activation

If $H = \phi(Z)$ and $G_H = \partial L / \partial H$, then

$$G_Z = G_H \odot \phi'(Z),$$

where $\odot$ is elementwise multiplication. For ReLU,

$$\phi'(z) = \begin{cases} 1, & z > 0, \\ 0, & z < 0. \end{cases}$$

7.2 Connection with Least Squares

For linear predictions $\widehat{\mathbf{y}} = X\mathbf{w}$ and

$$L(\mathbf{w}) = \frac{1}{2n} \|X\mathbf{w} - \mathbf{y}\|_2^2,$$

the gradient is

$$\nabla_{\mathbf{w}} L = \frac{1}{n} X^T (X\mathbf{w} - \mathbf{y}).$$

Gradient descent updates

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta \nabla_{\mathbf{w}} L,$$

where $\eta > 0$ is the learning rate.

Machine-Learning Connection

Closed-form least squares solves a linear model directly. Gradient-based learning scales to large datasets and nonlinear networks, where no comparable closed-form solution exists.

A gradient check compares an analytic derivative with a finite-difference approximation. It is useful for debugging small implementations, although automatic differentiation is used in modern training libraries.

8 Python and NumPy Practice

8.1 Forward Pass and One Gradient Step

```python
import numpy as np

X = np.array([[1., 2.], [2., -1.]])
y = np.array([[1.5], [0.0]])
W1 = np.array([[1., -1.], [0.5, 2.]])
b1 = np.array([0., 1.])
W2 = np.array([[1.], [-0.5]])
b2 = np.array([0.5])

Z1 = X @ W1 + b1
H = np.maximum(Z1, 0.0)
y_hat = H @ W2 + b2
loss = np.mean((y_hat - y) ** 2)

dY = 2 * (y_hat - y) / len(X)
dW2, db2 = H.T @ dY, dY.sum(axis=0)
dZ1 = (dY @ W2.T) * (Z1 > 0)
dW1, db1 = X.T @ dZ1, dZ1.sum(axis=0)

learning_rate = 0.05
W1 -= learning_rate * dW1
b1 -= learning_rate * db1
W2 -= learning_rate * dW2
b2 -= learning_rate * db2
print("Loss before update:", loss)
```

8.2 Embedding Lookup and Cosine Similarity

```python
E = np.array([[1., 0.], [1., 2.], [2., 1.]])
u, v = E[1], E[2]
cosine = (u @ v) / (np.linalg.norm(u) * np.linalg.norm(v))
print("Selected embedding:", u)
print("Cosine similarity:", cosine)
```

8.3 Scaled Dot-Product Attention

```python
def attention(Q, K, V):
    scores = Q @ K.T / np.sqrt(Q.shape[1])
    scores -= scores.max(axis=1, keepdims=True)
    weights = np.exp(scores)
    weights /= weights.sum(axis=1, keepdims=True)
    return weights @ V, weights

Q = K = np.eye(2)
V = np.array([[1., 0.], [0., 2.]])
context, weights = attention(Q, K, V)
print("Attention weights:\n", weights)
print("Context vectors:\n", context)
```

9 Review and Independent Practice

Practice Tasks

1. For $X \in \mathbb{R}^{32\times128}$ and $W \in \mathbb{R}^{128\times64}$, state the output shape and number of trainable parameters, including bias.
2. Explain why several dense layers without activations are equivalent to one affine layer.
3. Recalculate the worked forward pass after changing b_2 from 0.5 to -0.5.
4. Construct three embeddings and compare their dot-product and cosine similarities.
5. For $Q, K \in \mathbb{R}^{10\times16}$ and $V \in \mathbb{R}^{10\times8}$, determine the shapes of QK^T, the attention weights, and the output.
6. Derive $\partial L/\partial W$ and $\partial L/\partial X$ for $Z = XW$ using differentials.
7. Modify the NumPy network to run 100 updates. Record the loss and explain whether it decreases.

9.1 Key Takeaways

- Dense layers are batched affine transformations followed by activations.
- Nonlinear activations prevent a deep network from collapsing into one affine map.
- Embeddings provide learned continuous coordinates for discrete objects.
- Attention uses pairwise dot products and weighted matrix combinations.
- Backpropagation applies the chain rule through transposed matrix operations.
- Gradient descent moves parameters opposite the loss gradient.
- Shape checking and numerical stability are essential implementation skills.

9.2 Key Terms

Affine layer	Activation function	Hidden representation
Embedding matrix	Cosine similarity	Query, key, value
Attention score	Softmax	Attention mask
Backpropagation	Matrix gradient	Learning rate

Next lecture: Integrated machine-learning case studies, numerical reliability, and course review.