Training Day-71 Report:

Cost Function and **Gradient Descent**:

Cost Function

A **Cost Function** measures the error or difference between the predicted values and actual values in a model. It quantifies the model's performance and helps in optimizing it. The objective is to minimize the cost function during the training phase to improve the model's accuracy.

Types of Cost Functions:

1. **Mean Squared Error (MSE):**

MSE=1n∑i=1n(yi−y^i)2MSE = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2

Where:

- yiy_i: Actual output
- y^i\hat{y}_i: Predicted output
- nn: Number of data points

2. **Log Loss (Cross-Entropy Loss):** Commonly used for classification tasks.

LogLoss=−1n∑i=1n[yilog⁡(y^i)+(1−yi)log⁡(1−y^i)]Log Loss = -\frac{1}{n} \sum_{i=1}^{n} \left[y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i) \right]

3. **Hinge Loss:** Used for SVM models in classification.

Gradient Descent

Gradient Descent is an optimization algorithm used to minimize the cost function. It updates the model's parameters iteratively by moving in the direction of the negative gradient of the cost function.

Algorithm Steps:

1. **Initialize** the parameters (e.g., weights and biases) randomly or with zeros.
2. Compute the **gradient** of the cost function with respect to each parameter.
3. Update the parameters using the formula:

θ=θ−α∂J(θ)∂θ\theta = \theta - \alpha \frac{\partial J(\theta)}{\partial \theta}

Where:

- θ\theta: Model parameter

    - α\alpha: Learning rate
    - ∂J(θ)∂θ\frac{\partial J(\theta)}{\partial \theta}: Gradient of the cost function
4. Repeat until the cost function converges to a minimum.

Variants of Gradient Descent:

1. **Batch Gradient Descent:** Uses the entire dataset for each iteration.
2. **Stochastic Gradient Descent (SGD):** Updates parameters using a single data point at each iteration.
3. **Mini-Batch Gradient Descent:** Combines aspects of batch and SGD by using small batches of data.

This explanation is concise and suitable for understanding the concepts of cost function and gradient descent in the context of machine learning【6†source】【7†source】. Let me know if you need more details or practical examples!

Training Day-72 Report:

Linear Algebra Basics and the Vanilla Implementation of Neural Networks:

Linear Algebra Basics

Linear algebra is fundamental in machine learning and neural networks as it provides the mathematical framework for operations on data.

Key Concepts

1. Scalars: A single number (e.g., x=5x = 5).
2. Vectors: A 1D array of numbers (e.g., v=[v1,v2,…,vn]\mathbf{v} = [v_1, v_2, \ldots, v_n]).
3. Matrices: A 2D array of numbers (e.g., A=[a11a12a21a22]\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}).
4. Tensors: A generalization of vectors and matrices to higher dimensions.

Common Operations

1. Addition: Adding matrices or vectors element-wise.

A+B=[a11+b11a12+b12a21+b21a22+b22]\mathbf{A} + \mathbf{B} = \begin{bmatrix} a_{11} + b_{11} & a_{12} + b_{12} \\ a_{21} + b_{21} & a_{22} + b_{22} \end{bmatrix}

2. Dot Product: For vectors u\mathbf{u} and v\mathbf{v}:

u·v=∑iuivi\mathbf{u} \cdot \mathbf{v} = \sum_{i} u_i v_i

3. Matrix Multiplication: The dot product of rows of the first matrix with columns of the second matrix.

C=A·B\mathbf{C} = \mathbf{A} \cdot \mathbf{B}

4. Transpose: Flipping a matrix over its diagonal.

AT=[a11a21a12a22]\mathbf{A}^T = \begin{bmatrix} a_{11} & a_{21} \\ a_{12} & a_{22} \end{bmatrix}

5. Inverse: If A·A−1=I\mathbf{A} \cdot \mathbf{A}^{-1} = \mathbf{I}, then A−1\mathbf{A}^{-1} is the inverse of A\mathbf{A}.

Applications in ML

- Representing datasets (matrices where rows are samples and columns are features).
- Performing transformations (rotation, scaling).
- Solving systems of linear equations.

Vanilla Implementation of Neural Networks

A "vanilla" neural network refers to a simple, fully connected feedforward neural network.

Components

1. Input Layer: The data features.
2. Hidden Layers: Layers between input and output, containing neurons that perform intermediate computations.
3. Output Layer: Provides the final prediction or classification.

Forward Propagation

1. Input to Hidden Layer:

z1=W1·x+b1\mathbf{z}_1 = \mathbf{W}_1 \cdot \mathbf{x} + \mathbf{b}_1

- W1\mathbf{W}_1: Weight matrix for the hidden layer.
- b1\mathbf{b}_1: Bias vector for the hidden layer.
- x\mathbf{x}: Input vector.
- z1\mathbf{z}_1: Linear transformation result.

2. Activation Function:

a1=f(z1)\mathbf{a}_1 = f(\mathbf{z}_1)

Common activations:

- Sigmoid: f(x)=11+e−xf(x) = \frac{1}{1 + e^{-x}}
- ReLU: f(x)=max(0,x)f(x) = \max(0, x)

3. Hidden to Output Layer:

y=f(W2·a1+b2)\mathbf{y} = f(\mathbf{W}_2 \cdot \mathbf{a}_1 + \mathbf{b}_2)

- W2\mathbf{W}_2: Weight matrix for the output layer.
- b2\mathbf{b}_2: Bias vector for the output layer.

Backpropagation

1. Compute the loss using a cost function (e.g., Mean Squared Error, Cross-Entropy Loss).
2. Calculate gradients using the chain rule.
3. Update weights and biases using gradient descent: W=W−α·∂J∂W\mathbf{W} = \mathbf{W} - \alpha \cdot \frac{\partial J}{\partial \mathbf{W}}
   - α\alpha: Learning rate.
   - JJ: Cost function.

Python Implementation

```
import numpy as np

# Activation function (Sigmoid)
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

# Derivative of sigmoid
def sigmoid_derivative(x):
    return x * (1 - x)

# Input dataset
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y = np.array([[0], [1], [1], [0]])

# Initialize weights and biases
np.random.seed(0)
weights1 = np.random.rand(2, 2)
weights2 = np.random.rand(2, 1)
bias1 = np.random.rand(1, 2)
bias2 = np.random.rand(1, 1)

# Training process
for epoch in range(10000):
    # Forward propagation
    z1 = np.dot(X, weights1) + bias1
    a1 = sigmoid(z1)
    z2 = np.dot(a1, weights2) + bias2
    output = sigmoid(z2)
    # Backpropagation
    loss = y - output
    d_output = loss * sigmoid_derivative(output)
    d_hidden = d_output.dot(weights2.T) * sigmoid_derivative(a1)

    # Update weights and biases
    weights2 += a1.T.dot(d_output)
    weights1 += X.T.dot(d_hidden)
    bias2 += np.sum(d_output, axis=0, keepdims=True)
    bias1 += np.sum(d_hidden, axis=0, keepdims=True)

print("Trained Output:", output)
```

This provides a foundation for understanding and implementing basic neural networks. Let me know if you want to delve deeper into any aspect!

Training Day-73 Report:

Networks in Python and TensorFlow Basics under the context of Deep Learning:

Networks in Python

In deep learning, a network refers to an artificial neural network (ANN), which consists of layers of interconnected nodes (neurons) for feature extraction and prediction.

Building a Neural Network in Python (from scratch)

Below is an example of a simple feedforward neural network:

```
import numpy as np


# Sigmoid activation function
def sigmoid(x):
    return 1 / (1 + np.exp(-x))


# Derivative of sigmoid
def sigmoid_derivative(x):
    return x * (1 - x)


# Input data (XOR problem)
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y = np.array([[0], [1], [1], [0]])


# Initialize weights and biases
np.random.seed(42)
weights_input_hidden = np.random.rand(2, 2)  # 2 input nodes, 2 hidden nodes
weights_hidden_output = np.random.rand(2, 1) # 2 hidden nodes, 1 output node
bias_hidden = np.random.rand(1, 2)
bias_output = np.random.rand(1, 1)
```

```
# Training parameters
epochs = 10000
learning_rate = 0.1

for epoch in range(epochs):
    # Forward pass
    hidden_layer_input = np.dot(X, weights_input_hidden) + bias_hidden
    hidden_layer_output = sigmoid(hidden_layer_input)
    final_input = np.dot(hidden_layer_output, weights_hidden_output) + bias_output
    final_output = sigmoid(final_input)

    # Backward pass (calculate gradients)
    error = y - final_output
    d_output = error * sigmoid_derivative(final_output)
    d_hidden_layer = d_output.dot(weights_hidden_output.T) *
sigmoid_derivative(hidden_layer_output)

    # Update weights and biases
    weights_hidden_output += hidden_layer_output.T.dot(d_output) * learning_rate
    weights_input_hidden += X.T.dot(d_hidden_layer) * learning_rate
    bias_output += np.sum(d_output, axis=0) * learning_rate
    bias_hidden += np.sum(d_hidden_layer, axis=0) * learning_rate

print("Output after training:")
print(final_output)
```

This example demonstrates the structure and operations of a neural network.

TensorFlow Basics

TensorFlow is an open-source library widely used for building and training machine learning and deep learning models.

Core Components

1. Tensors: Multidimensional arrays (like NumPy arrays) that flow through the network.
2. Graphs: Computational graphs represent operations and the flow of tensors.
3. Operations (Ops): Nodes in the computational graph.

Basic Workflow in TensorFlow

1. Import TensorFlow:
2. import tensorflow as tf
3. Define a Neural Network: Example: A simple feedforward neural network for classification.
4. import tensorflow as tf
5. from tensorflow.keras.models import Sequential
6. from tensorflow.keras.layers import Dense
7.
8. # Create a sequential model
9. model = Sequential([
10.   Dense(32, activation='relu', input_shape=(2,)), # Input layer with 32 neurons
11.   Dense(16, activation='relu'), # Hidden layer with 16 neurons
12.   Dense(1, activation='sigmoid') # Output layer with sigmoid for binary classification
13.])
14.
15. # Compile the model
16. model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
17.
18. # Dummy dataset (XOR problem)
19. X = [[0, 0], [0, 1], [1, 0], [1, 1]]
20. y = [0, 1, 1, 0]
21.
22. # Train the model
23. model.fit(X, y, epochs=100, verbose=0)
24.
25. # Evaluate the model

26. print("Model evaluation:", model.evaluate(X, y))
27. Key TensorFlow APIs:
    - tf.keras: High-level API for building models.
    - tf.data: Tools for creating efficient input pipelines.
    - tf.function: For converting Python functions into TensorFlow graphs.
28. Building Custom Models: Use the TensorFlow low-level API to define your model architecture and gradients manually.

```
29. class CustomModel(tf.keras.Model):
30.     def __init__(self):
31.         super(CustomModel, self).__init__()
32.         self.hidden_layer = tf.keras.layers.Dense(32, activation='relu')
33.         self.output_layer = tf.keras.layers.Dense(1, activation='sigmoid')
34.
35.     def call(self, inputs):
36.         x = self.hidden_layer(inputs)
37.         return self.output_layer(x)
38.
39. # Instantiate and compile the custom model
40. model = CustomModel()
41. model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
```

Training Day-74 Report:

Simple Neural Network with TensorFlow, **Word Embedding**, and understanding **CBOW** and **Skip-gram** models.

1. Simple Neural Network with TensorFlow

Goal:

Build a neural network using TensorFlow to solve a binary classification problem (e.g., XOR).

Implementation:

```
import tensorflow as tf

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense


# XOR dataset

X = [[0, 0], [0, 1], [1, 0], [1, 1]]

y = [0, 1, 1, 0]


# Define a sequential model

model = Sequential([

    Dense(8, activation='relu', input_shape=(2,)),  # Hidden layer with 8 neurons

    Dense(1, activation='sigmoid')                  # Output layer for binary classification

])


# Compile the model

model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])


# Train the model

model.fit(X, y, epochs=100, verbose=1)
```

```
# Evaluate the model
print("Evaluation Results:", model.evaluate(X, y))

# Predictions
print("Predictions:", model.predict(X))
```

This simple NN demonstrates how to use TensorFlow to solve small problems with minimal code.

2. Word Embedding

Word embeddings convert words into dense vectors in a continuous vector space, capturing semantic meanings.

Using TensorFlow/Keras:

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, Flatten, Dense

# Sample vocabulary and corresponding tokenized sentences
vocab_size = 50
embedding_dim = 8
max_length = 10

# Example tokenized input
sentences = [[1, 2, 3, 4], [3, 4, 1, 2]]
padded_sentences = tf.keras.preprocessing.sequence.pad_sequences(sentences, maxlen=max_length)

# Define the embedding layer model
model = Sequential([
    Embedding(input_dim=vocab_size, output_dim=embedding_dim, input_length=max_length),
    Flatten(),
    Dense(1, activation='sigmoid')
])
```

```
# Compile and summarize the model
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
model.summary()

# Dummy output labels
labels = [1, 0]

# Train the model
model.fit(padded_sentences, labels, epochs=10)
```

This creates word embeddings for a vocabulary and uses them in a classification task.

3. CBOW (Continuous Bag of Words) & Skip-gram

CBOW and Skip-gram are two approaches used in Word2Vec for learning word embeddings.

CBOW:

- Predicts a target word from its surrounding context words.
- Suitable for frequent words in a corpus.

Skip-gram:

- Predicts context words from a target word.
- Suitable for infrequent words.

TensorFlow Implementation of Skip-gram:

```
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Embedding, Dot, Flatten

# Sample dataset
word_pairs = [(1, 2), (1, 3), (2, 3), (2, 4)]  # (target, context)
vocab_size = 5  # Vocabulary size
embedding_dim = 8
```

```
# Prepare inputs and labels
targets, contexts = zip(*word_pairs)
targets = np.array(targets)
contexts = np.array(contexts)

# Define the model
input_target = Input((1,))
input_context = Input((1,))

embedding = Embedding(vocab_size, embedding_dim, input_length=1)

target_embedding = embedding(input_target)
context_embedding = embedding(input_context)

dot_product = Dot(axes=-1)([target_embedding, context_embedding])
output = Flatten()(dot_product)

model = Model(inputs=[input_target, input_context], outputs=output)
model.compile(optimizer='adam', loss='mse')

# Train the model
model.fit([targets, contexts], np.ones(len(word_pairs)), epochs=100)
```

This implementation focuses on Skip-gram, where the network learns word embeddings by maximizing the similarity between target and context words.

Summary:

- **Simple NN**: Built a basic neural network using TensorFlow for XOR classification.
- **Word Embedding**: Showed how to use TensorFlow's Embedding layer for converting words into dense vectors.

- **CBOW & Skip-gram**: Explained concepts and provided a Skip-gram implementation in TensorFlow.

Training Day-75 Report:

Word Relations, Convolutional Neural Networks (CNNs), and MaxPooling:

1. Word Relations

Definition:

Word relations refer to the semantic connections between words, such as synonyms, antonyms, or analogies. Word embeddings, like Word2Vec, GloVe, or FastText, are often used to model these relations.

Example:

Using word embeddings, the relationship "king - man + woman = queen" can be captured mathematically.

Implementation of Word Relations with Word2Vec:

```
from gensim.models import Word2Vec


# Sample sentences

sentences = [["king", "queen", "man", "woman", "prince", "princess"]]


# Train Word2Vec model

model = Word2Vec(sentences, vector_size=50, window=2, min_count=1, workers=4)


# Find similar words

print("Most similar to 'king':", model.wv.most_similar("king"))


# Word analogy

print("Result of 'king - man + woman':", model.wv.most_similar(positive=['king', 'woman'],
negative=['man']))
```

Word relations allow tasks like analogy solving, clustering, and understanding the context of words in NLP.

2. Convolutional Neural Networks (CNNs)

Overview:

CNNs are specialized for processing grid-like data such as images. They apply convolution operations to extract features like edges, textures, and shapes.

Components of a CNN:

1. Convolution Layer: Extracts features from the input using filters (kernels).
2. Activation Function: Often ReLU, to introduce non-linearity.
3. Pooling Layer: Reduces the spatial dimensions, retaining essential features (e.g., MaxPooling).
4. Fully Connected Layer: For classification or regression tasks.

CNN Implementation for Image Classification:

```
import tensorflow as tf

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense


# Build a CNN model

model = Sequential([

  Conv2D(32, (3, 3), activation='relu', input_shape=(64, 64, 3)),  # Convolution layer

  MaxPooling2D(pool_size=(2, 2)),                        # MaxPooling layer

  Conv2D(64, (3, 3), activation='relu'),                 # Convolution layer

  MaxPooling2D(pool_size=(2, 2)),                        # MaxPooling layer

  Flatten(),                                   # Flatten the output

  Dense(128, activation='relu'),                       # Fully connected layer

  Dense(10, activation='softmax')                        # Output layer

])


# Compile the model

model.compile(optimizer='adam', loss='sparse_categorical_crossentropy',
metrics=['accuracy'])
```

Summary

```
model.summary()
```

3. MaxPooling

Purpose:

MaxPooling is a down-sampling technique that reduces the spatial dimensions of feature maps while retaining the most prominent features.

Example of MaxPooling:

For a 2×22 \times 2 pooling window:

[1324]⇒Max: 4\begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix} \quad \Rightarrow \quad \text{Max: } 4

MaxPooling in TensorFlow:

```
import numpy as np

import tensorflow as tf

from tensorflow.keras.layers import MaxPooling2D


# Dummy input feature map

feature_map = np.array([[[[1], [3]], [[2], [4]]]], dtype=np.float32)  # Shape: (1, 2, 2, 1)


# Apply MaxPooling

max_pool = MaxPooling2D(pool_size=(2, 2))

result = max_pool(feature_map)


print("Input Feature Map:", feature_map)

print("After MaxPooling:", result.numpy())
```

MaxPooling Benefits:

1. Reduces computation by lowering spatial dimensions.
2. Mitigates overfitting by retaining dominant features.
3. Introduces slight invariance to translations in input data.

Summary:

- Word Relations: Leveraging embeddings for analogies and semantic connections.
- CNNs: Extract features hierarchically for image tasks.
- MaxPooling: Simplifies data by reducing dimensions while retaining key features.