دانشکده مهندسی کامپیوتر

گزارش درس مبانی هوش محاسباتی

عنوان گزارش:

تمرین سوم درس هوش محاسباتی (شبکه عصبی مصنوعی)

ارائه‌دهندگان:

فرزانه آقازاده **زینب جنتی** **فاطمه نجفی**

استاد درس:

دکتر حسین کارشناس

دستیاران درس:

رضا برزگر

علی شاه زمانی

آرمان خلیلی

نیم سال دوم ۱۴۰۳_۱۴۰۴

پیاده سازی بخش ۱

```
X_train, X_test,y_train,y_test = train_test_split(
    X,y,
    test_size=0.2,
    random_state=42,
)
print(X_train)
```

در این مرحله از کد، آماده‌سازی دیتاها انجام می‌شود و کل دیتاست به دو بخش (۸۰٪ آموزش و ۲۰٪ آزمون) به طوری که نسبت دسته‌بندی‌ها توی هر دو قسمت به طور یکسان تقسیم شود. به این کار می‌گیم نمونه‌گیری طبقه‌بندی‌شده.

```
def check_missing_values():
    missing = X.isnull().sum()
    missing = missing[missing > 0]
    if not missing.empty:
        print(missing)
    else:
        print("\n No Missing Data \n")

def detect_outliers(z_thresh=3.29):
    for col in X.columns:
        if pd.api.types.is_numeric_dtype(X[col]):
            z_scores = zscore(X[col])
            outliers = (abs(z_scores) > z_thresh)
            num_outliers = outliers.sum()
            print(f"col '{col}': {num_outliers} outlier found.")

def normal(X):
    return X.reshape(X.shape[0], -1).astype(np.float32) / 255.0

def plot_distributions():
    for col in X_train.columns:
        if pd.api.types.is_numeric_dtype(X_train[col]):
            plt.figure(figsize=(6,4))
            sns.histplot(X_train[col], kde=True)
            plt.title(f"Distribution of {col}")
            plt.show()
        else:
          print("\n",col, "is not numeric")
```

این سه تابع کمک می‌کنند که بفهمیم آیا داده‌ها ناقص هستند، آیا مقادیر خیلی پرت هستند و یا شکل توزیع داده‌ها به چه صورت هست. این سه تابع، سه مرحله‌ی مهم از تحلیل اکتشافی داده‌ها (EDA) را انجام می‌دهند.

```
def normal(X):
    return X.reshape(X.shape[0], -1).astype(np.float32) / 255.0
```

.

این تابع برای نرمال‌سازی داده‌های تصویر طراحی شده است.

۱. تغییر شکل داده به صورت (تعداد نمونه‌ها, سایر ابعاد)

۲. تبدیل نوع داده به float32

۳. تقسیم همه مقادیر بر ۲۵۵

In [18]:
```
y = np.where(data['label'] == 0, 1, 0)
print(y)
```

```
[0 0 0 ... 0 0 0]
```

In [24]:
```
print("not airplain:", np.sum(y_train == 0))
print("airplain:", np.sum(y_train == 1))
```

```
not airplain: 35973
airplain: 4027
```

این تابع برای تبدیل لیبل کلاس هواپیما به ۱ و تبدیل بقیه لیبل ها به ۰ استفاده شده است و با چاپ این مقادیر میبینیم که بهدرستی این کار صورت گرفته است.

In [27]:
```
def sigmoid(z):
    return 1 / (1 + np.exp(-z))
```

In [34]:
```
def binary_cross_entropy(y_true, y_pred):
    bce = -np.mean(y_true * np.log(y_pred) + (1 - y_true) * np.log(1 - y_pred))
    return bce
```

تابع Sigmoid

تابع فعال‌سازی غیرخطی که خروجی را بین ۰ و ۱ نگه می‌دارد

تابع هزینه(Binary Cross-Entropy)

برای مسائل طبقه‌بندی باینری مناسب است. خطای بین پیش‌بینی و مقدار واقعی را محاسبه می‌کند.

```
def forward(X, y, W, b):
    z = np.dot(X, W) + b
    return sigmoid(z)

def compute_loss(y_true, y_pred):
        return binary_cross_entropy(y, y_pred)

def backward(X, y, W, b, lr, loss, y_pred):
    dz = y_pred - y
    dw = np.dot(X.T,dz)/X.shape[0]
    db= np.mean(dz)

    W  -= lr * dw
    b -= lr * db

    return W, b
def params(X):
    W = np.random.normal(0, 1/np.sqrt(X.shape[1]), size=X.shape[1])
    b = 0
    return W,b
```

تابع forward

محاسبه خروجی شبکه از لایه ورودی به خروجی

شامل یک لایه ورودی و یک لایه خروجی

تابع backward

محاسبه گرادیان‌ها برای به‌روزرسانی وزن‌ها

از قاعده زنجیره‌ای برای محاسبه مشتقات استفاده می‌کند

پارامتر ها را بر اساس مشتق بدست آمده به روزرسانی میکند.

تابع params

پارامتر های اولیه شبکه عصبی، وزن های رندوم و بایاس را ایجاد میکند.

```python
def train(X, y, X_test, y_test, epochs=100, lr=0.1):
    W, b = params(X)

    history = {
        'train_loss': [],
        'test_loss': [],
        'accuracy': []
    }

    for epoch in range(epochs):
        # Forward pass (train)
        y_pred_train = forward(X, y, W, b)
        train_loss = binary_cross_entropy(y, y_pred_train)

        # Backward pass
        W, b = backward(X, y, W, b, lr, train_loss, y_pred_train)

        # Evaluation on test set
        #y_pred_test = forward(X_test, y_test, W, b)
        test_loss = 0# binary_cross_entropy(y_test, y_pred_test)
        accuracy = 0#np.mean((y_pred_test >= 0.5).astype(int) == y_test)

        # Store history
        history['train_loss'].append(train_loss)
        history['test_loss'].append(test_loss)
        history['accuracy'].append(accuracy)

        if epoch % 10 == 0:
            print(f"Epoch {epoch:3d} | Train Loss: {train_loss}")

    return W, b, history
```

تابع train

حلقه اصلی آموزش برای تعداد مشخصی دوره(epochs)

در هر دوره forward و backward pass انجام می‌شود.

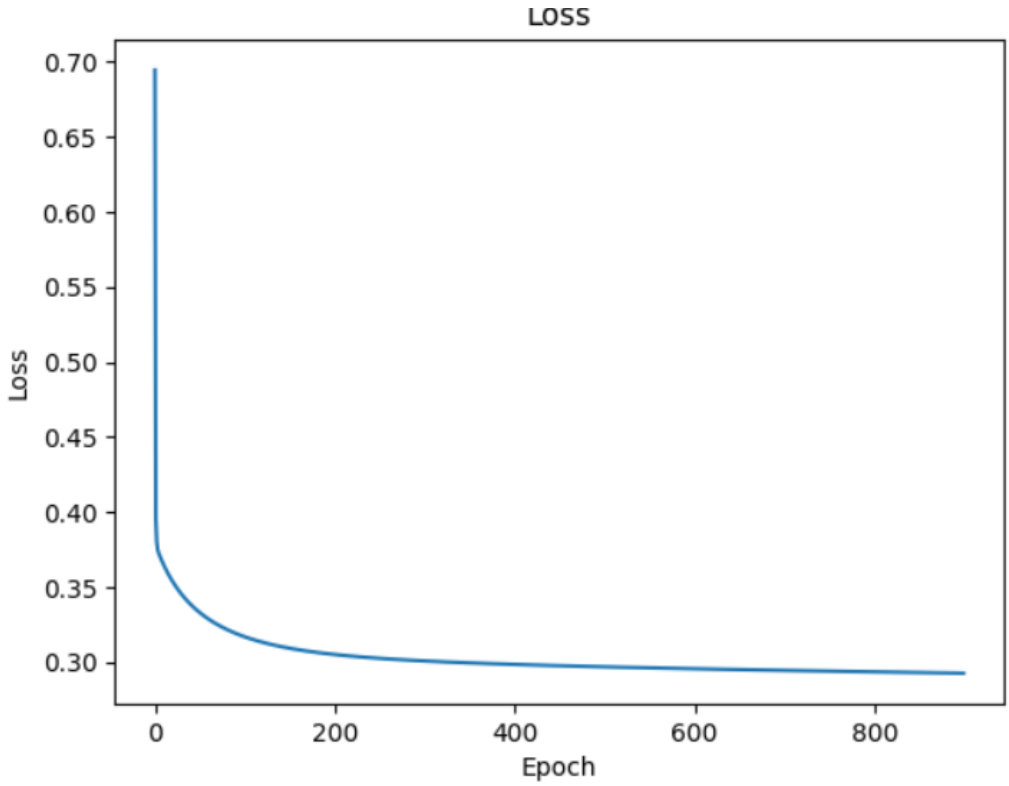
به روز رسانی پارامترها بر اساس نرخ یادگیری داده شده صورت میگیرد.

نتایج شبکه عصبی بدون لایه پنهان

```
W , b, h= train(X_train, y_train, X_test, y_test, 900, 0.01)
y_pred = predict(X_test, W, b)
```

```
Confusion Matrix:
TP: 135, FP: 124
FN: 838, TN: 8903
F1-score: 0.21915584083165113
```

این شبکه موفق شد بعد ۹۰۰ بار اجرا به نرخ تشخیص ۰.۲ برسد. یکی از دلایلی که یادگیری کند انجام شده نسبت تعداد کلاس های هواپیما به کل داده است که تنها ده درصد را تشکیل میدهد و این موضوع یادگیری را برای مدلدشوار تر میکند و به تعداد اجرای بیشتری نیاز پیدا میکند. البته با استفاده از تکنیک هایی شبیه ارسال داده های به صورت batch میتوان نتیجه ی بهتری دریافت کرد.



پیاده سازی بخش ۲

پیش پردازش داده ها نسبت به قبل تغییری نداشته است.

In [40]:

```
def sigmoid(z):
    return 1 / (1 + np.exp(-z))
```

In [52]:

```
def binary_cross_entropy(y_true, y_pred):
        bce = -np.mean(y_true * np.log(y_pred) + (1 - y_true) * np.log(1 - y_pred))
        return bce
```

تغییر در اضافه شدن یک لایه ی پنهان به شبکه عصبی است.

```
def forward(X, W1, b1, W2, b2):

    z1 = np.dot(X, W1) + b1
    A1 = sigmoid(z1)

    z2 = np.dot(A1, W2) + b2
    A2 = sigmoid(z2)
    A2 = A2.reshape(-1, 1)

    return z1, A1, z2, A2
```

تابع forward

محاسبه خروجی شبکه از لایه ورودی به خروجی

شامل یک لایه ورودی یک لایه پنهان و یک لایه خروجی(دو لایه)

با اضافه شدن لایه پنهان باید محاسبات برای هر دو لایه ی پنهان و خروجی محسابه شود. در این تابع از یک لایه پنهان و سپس از لایه خروجی ورودی ها را متقل کرده و نتیجه را بدست میاوریم.

```python
def compute_loss(y_true, y_pred):
        return binary_cross_entropy(y, y_pred)

def backward(X, y, Z1, A1, Z2, A2, W1, W2, lr):

    m = X.shape[0]
    dZ2 = A2 - y
    dW2 = np.dot(A1.T, dZ2) / m
    db2 = np.sum(dZ2, axis=0, keepdims=True) / m

    dA1 = np.dot(dZ2, W2.T)
    dZ1 = dA1 * A1 * (1 - A1)
    dW1 = np.dot(X.T, dZ1) / m
    db1 = np.sum(dZ1, axis=0, keepdims=True) / m

    return dW1, db1, dW2, db2
```

تابع backward

محاسبه گرادیان‌ها برای به‌روزرسانی وزن‌ها

از قاعده زنجیره‌ای برای محاسبه مشتقات استفاده می‌کند.

برای هر دو لایه این مشتقات محاسبه میشود و پارامتر ه ای مربوط به نرون های هر لایه محاسبه میشوند.

پارامتر ها را بر اساس مشتق بدست آمده به روزرسانی میکند.

```python
def update_parameters(W1, b1, W2, b2, dW1, db1, dW2, db2, lr):
    W1 -= lr * dW1
    b1 -= lr * db1
    W2 -= lr * dW2
    b2 -= lr * db2
    return W1, b1, W2, b2

    return W1, b1, W2, b2
def params(input_dim, hidden_dim = 64, output_dim=1):
    W1 = np.random.randn(input_dim, hidden_dim) * np.sqrt(2./input_dim)
    b1 = np.zeros((1, hidden_dim))
    W2 = np.random.randn(hidden_dim, 1) * np.sqrt(1./hidden_dim)
    b2 = np.zeros((1, 1))
    return W1, b1, W2, b2
```

در تابع update مقادیر بدست آمده در تابع backward باعث به روزرسانی پارامترهای هر دو لایه میشود.

در تابعparams نیاز داریم که مقایدر اولیه هر دو لایه را مشخص کنیم.

```python
def train(X, y, X_test, y_test, epochs=100, lr=0.1, hidden_dim=64):
    input_dim = X_train.shape[1]
    W1, b1, W2, b2 = params(input_dim, hidden_dim)

    history = {
        'train_loss': [],
        'test_loss': [],
        'accuracy': []
    }

    for epoch in range(epochs):
        # Forward pass (train)
        Z1, A1, Z2, A2 = forward(X_train, W1, b1, W2, b2)
        train_loss = binary_cross_entropy(y_train, A2)

        # Backward pass
        dW1, db1, dW2, db2 = backward(X_train, y_train, Z1, A1, Z2, A2, W1, W2, lr)
        W1, b1, W2, b2 = update_parameters(W1, b1, W2, b2, dW1, db1, dW2, db2, lr)
        # # Evaluation on test set
        # #y_pred_test = forward(X_test, y_test, W, b)
        # test_loss = 0# binary_cross_entropy(y_test, y_pred_test)
        # accuracy = 0#np.mean((y_pred_test >= 0.5).astype(int) == y_test)

        # Store history
        history['train_loss'].append(train_loss)

        if epoch % 10 == 0:
            print(f"Epoch {epoch:3d} | Train Loss: {train_loss}")

    return W1, b1, W2, b2, history
```

تابع train تقریبا مشابه مرحله قبل عمل میکند

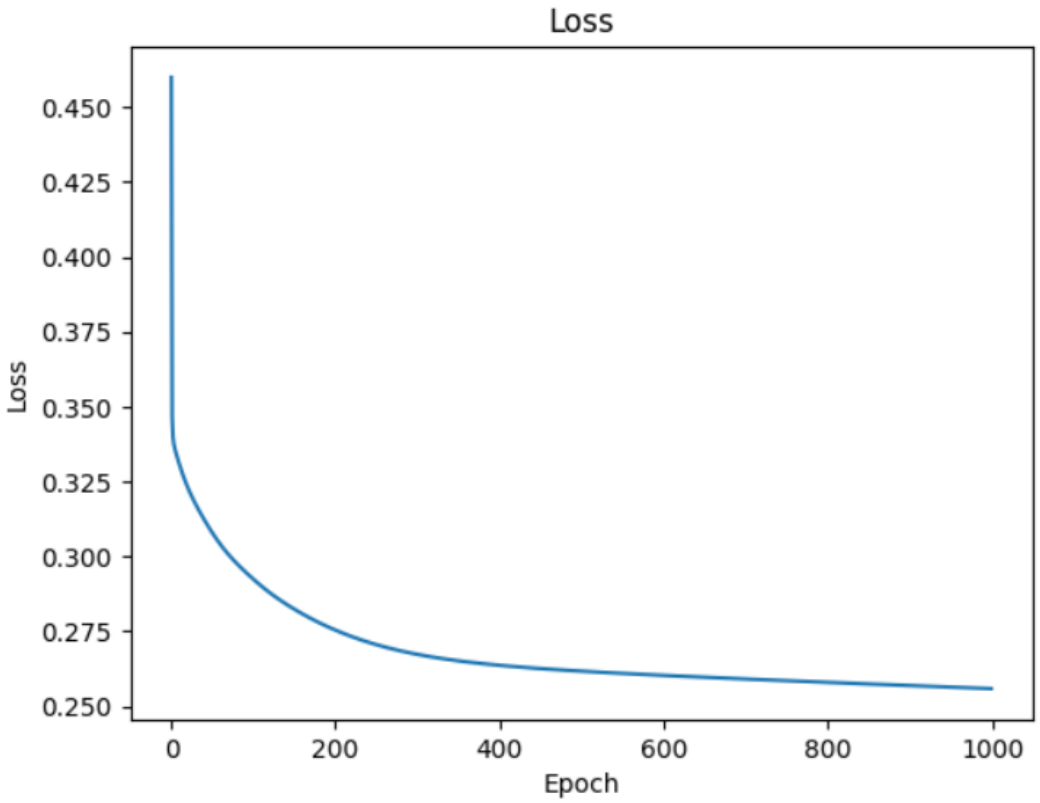
تنها پارامتر های برای لایه نهان به روز رسانی میشوند.

In [146]:
```
W1, b1, W2, b2, h= train(X_train, y_train, X_test, y_test, 1000, 0.1)
```

In [148]:
```
evaluate(y_test, y_pred)
```

```
Confusion Matrix:
TP: 130, FP: 51
FN: 843, TN: 8976
F1-score: 0.2253
```

مشکلات شبکه عصبی بخش ۱ در این شبکه هم وجود دارد و با نرخ یادگیری متفاوت و لاس اولیه متفاوت نتایج تقریبا شبیه هم شده است.



پیاده سازی بخش ۳

در توابع آماده سازی داده تنها مرحله ای که تفاوت کرده است تخصیص انکدر به هر ده کلاس موجود در ستون لیبل ها.

In [18]:
```
def one_hot_encode(labels, num_classes):
    one_hot = np.zeros((y.shape[0], num_classes))
    one_hot[np.arange(y.shape[0]), y] = 1
    return one_hot
```

برچسب‌های عددی به بردار one-hot تبدیل می‌شن مثلاً 3 → [0 0 0 1 0 0 0 0 0 0]

```
def binary_cross_entropy(y_true, y_pred):
    epsilon = 1e-12
    y_pred = np.clip(y_pred, epsilon, 1. - epsilon)
    if len(y_pred.shape) == 1:
        y_pred = y_pred.reshape(-1, 1)
    return -np.mean(np.sum(y_true * np.log(y_pred), axis=1))
```

تابع کراس انتروپی برای چند کلاسه پیادهسازی شده است.

```
def forward(X, W1, b1, W2, b2):

    z1 = np.dot(X, W1) + b1
    A1 = sigmoid(z1)

    z2 = np.dot(A1, W2) + b2
    A2 = softmax(z2)
```

تابع فعالسازی softmax در لایه ی خروجی جایگزین شده است.

```
def params(input_dim, hidden_dim = 64, output_dim=10):
    W1 = np.random.randn(input_dim, hidden_dim) * np.sqrt(2./input_dim)
    b1 = np.zeros((1, hidden_dim))

    W2 = np.random.randn(hidden_dim, output_dim) * np.sqrt(1./hidden_dim)
    b2 = np.zeros((1, output_dim))
```

لایه ی انتهایی که به ۱۰ خروجی نیاز دارد جایگزین شده است.

```
    for epoch in range(epochs):
        # Forward pass (train)
        Z1, A1, Z2, A2 = forward(X_train, W1, b1, W2, b2)
        train_loss = binary_cross_entropy(y_train, A2)

        # Backward pass
        dW1, db1, dW2, db2 = backward(X_train, y_train, Z1, A1, Z2, A2, W1, W2, lr)
        W1, b1, W2, b2 = update_parameters(W1, b1, W2, b2, dW1, db1, dW2, db2, lr)
        # # Evaluation on test set
        # #y_pred_test = forward(X_test, y_test, W, b)
        # test_loss = 0# binary_cross_entropy(y_test, y_pred_test)
        # accuracy = 0#np.mean((y_pred_test >= 0.5).astype(int) == y_test)

        # Store history
        history['train_loss'].append(train_loss)

        if epoch % 10 == 0:
            print(f"Epoch {epoch:3d} | Train Loss: {train_loss}")

    return W1, b1, W2, b2, history
```

همانندقبل آموزش داده شده است. و بر اساس معیار های قبلی و طبقه بندی اشتفاده شده است.

In [61]:

```
W1, b1, W2, b2, h= train(X_train, y_train, X_test, y_test, 2000, 0.1)
```

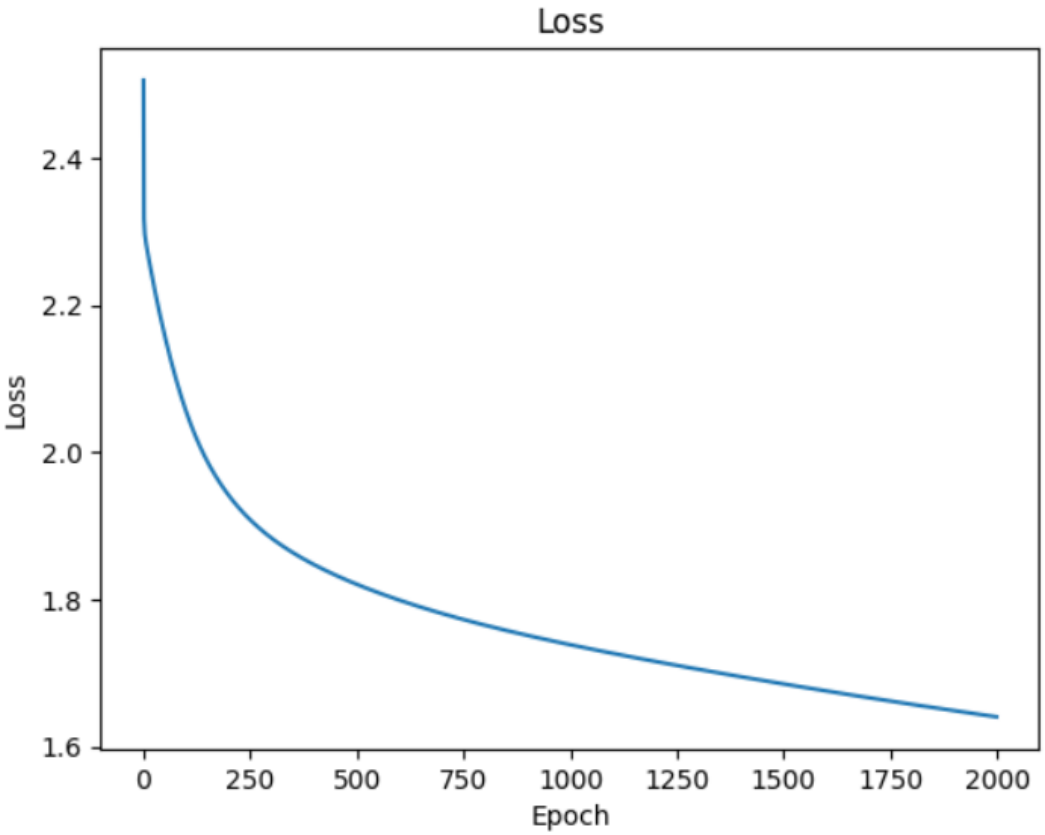
```
evaluate(y_test, y_pred)
```

```
              precision    recall  f1-score   support

           0       0.11      0.91      0.19       973
           1       0.73      0.25      0.37       979
           2       0.75      0.00      0.01      1030
           3       0.57      0.01      0.02      1023
           4       0.64      0.03      0.05       933
           5       0.74      0.06      0.11      1015
           6       0.59      0.11      0.19       996
           7       0.70      0.17      0.28       994
           8       0.74      0.29      0.42      1017
           9       0.68      0.21      0.32      1040

    accuracy                           0.20     10000
   macro avg       0.63      0.20      0.20     10000
weighted avg       0.63      0.20      0.20     10000

Confusion Matrix:
TP: 1339, FP: 564
FN: 8661, TN: 89436
F1-score: 0.2250
```



در این بخش برای دریافت نتایج مجبور به آموزش به تعدادتقریبا دو برابر مواردقبلی شدیم و دقت مدل هنوز نیاز به بهتر شدن دارد.

پیاده سازی بخش ۴

```
class NeuralNetwork:
    def __init__(self, layer_dims, activations,
initializer='he', optimizer='momentum', lr=0.01):
        self.params = {}
        self.activations = activations
        self.optimizer = optimizer
        self.lr=lr
        for l in range(1, len(layer_dims)):
            if initializer == 'he':
                self.params[f'W{l}'] =
he_initialization((layer_dims[l-1], layer_dims[l]))
            elif initializer == 'xavier':
```

```
                self.params[f'W{l}'] =
xavier_initialization((layer_dims[l-1], layer_dims[l]))
            self.params[f'b{l}'] = np.zeros((1, layer_dims[l]))

    def forward(self, X):
        caches = []
        A = X
        L = len(self.params) // 2

        for l in range(1, L):
            W = self.params[f'W{l}']
            b = self.params[f'b{l}']
            Z = np.dot(A, W) + b
            A = self._apply_activation(Z, self.activations[l-1])
            caches.append((Z, A))

        W = self.params[f'W{L}']
        b = self.params[f'b{L}']
        Z = np.dot(A, W) + b
        A = softmax(Z)
        caches.append((Z, A))

        return A, caches

    def _apply_activation(self, z, activation):
        if activation == 'relu':
            return relu(z)
        elif activation == 'sigmoid':
            return sigmoid(z)
        elif activation == 'tanh':
            return np.tanh(z)
    def _activation_derivative(self, z, activation_name):
            if activation_name == 'relu':
                return (z > 0).astype(float)
            elif activation_name == 'sigmoid':
                s = 1 / (1 + np.exp(-z))
                return s * (1 - s)
            elif activation_name == 'tanh':
                return 1 - np.tanh(z)**2
            else:
                raise ValueError(f"error.")

    def train(self, X, y, epochs=100, lr=0.01, batch_size=32,
val_data=None):
        history = {'train_loss': [], 'train_acc': [],
'val_loss': [], 'val_acc': []}
        self.lr=lr
        for epoch in range(epochs):
```

```
            for i in range(0, X.shape[0], batch_size):
                X_batch = X[i:i+batch_size]
                y_batch = y[i:i+batch_size]

                # Forward , backward
                A, caches = self.forward(X_batch)
                grads = self.backward(X_batch, y_batch, caches)

                self.update_params(grads, lr)


            train_loss, train_acc = self.evaluate(X, y)
            history['train_loss'].append(train_loss)
            history['train_acc'].append(train_acc)

            if val_data is not None:
                val_loss, val_acc = self.evaluate(val_data[0],
val_data[1])
                history['val_loss'].append(val_loss)
                history['val_acc'].append(val_acc)

                print(f"Epoch {epoch}: Train
Loss={train_loss:.4f}, Acc={train_acc:.4f} | Val
Loss={val_loss:.4f}, Acc={val_acc:.4f}")

        return history
    def backward(self, X_batch, y_batch, caches):

        grads = {}
        L = len(self.params) // 2
        m = X_batch.shape[0]


        dZ = caches[-1][1] - y_batch
        grads[f'dW{L}'] = np.dot(caches[-2][1].T, dZ) / m
        grads[f'db{L}'] = np.sum(dZ, axis=0, keepdims=True) / m


        for l in reversed(range(L-1)):
            dA = np.dot(dZ, self.params[f'W{l+2}'].T)
            dZ = dA * self._activation_derivative(caches[l][0],
self.activations[l])
            grads[f'dW{l+1}'] = np.dot(caches[l-1][1].T if l > 0
else X_batch.T, dZ) / m
            grads[f'db{l+1}'] = np.sum(dZ, axis=0,
keepdims=True) / m

        return grads
```

```
    def update_params(self, grads, lr=None, t=1):
        if lr is None:
            lr = self.lr

        beta1 = 0.9
        beta2 = 0.999
        eps = 1e-8

        if self.optimizer == "momentum" and not hasattr(self,
"velocities"):
            self.velocities = {}
            for key in self.params:
                self.velocities[key] =
np.zeros_like(self.params[key])

        if self.optimizer == "adam" and not hasattr(self, "m"):
            self.m, self.v = {}, {}
            for key in self.params:
                self.m[key] = np.zeros_like(self.params[key])
                self.v[key] = np.zeros_like(self.params[key])

        for key in self.params:
            grad_key = "d" + key
            if grad_key not in grads:
                continue

            if self.optimizer == "sgd":
                self.params[key] -= lr * grads[grad_key]

            elif self.optimizer == "momentum":
                self.velocities[key] = 0.9 *
self.velocities[key] - lr * grads[grad_key]
                self.params[key] += self.velocities[key]

            elif self.optimizer == "adam":
                self.m[key] = beta1 * self.m[key] + (1 - beta1)
* grads[grad_key]
                self.v[key] = beta2 * self.v[key] + (1 - beta2)
* (grads[grad_key] ** 2)

                m_hat = self.m[key] / (1 - beta1 ** t)
                v_hat = self.v[key] / (1 - beta2 ** t)

                self.params[key] -= lr * m_hat / (np.sqrt(v_hat)
+ eps)

    def evaluate(self, X, y):
        A, _ = self.forward(X)
```

```
        loss = self._compute_loss(y, A)

        predictions = np.argmax(A, axis=1)
        true_labels = np.argmax(y, axis=1)
        accuracy = np.mean(predictions == true_labels)

        return loss, accuracy

    def _compute_loss(self, y_true, y_pred):
        epsilon = 1e-12
        y_pred = np.clip(y_pred, epsilon, 1. - epsilon)
        return -np.mean(np.sum(y_true * np.log(y_pred), axis=1))
```

پیاده سازی مدولار تابع neural network

این تابع با دریافت ورودی تعداد لایه ها نوع فعال ساز و اپتیمایزر و نرخ یادگیری شروع به ساخت یک شبکه عصبی میکند.

بعد از تشخیص ورش های انتخابی کاربر با استفاده از توابع آن ها هر لایه از شبکه عصبی را با روش مخصوصش آموزش داده و در انتها یک تاریخچهاز مقادیر loss روی داده های آموزش و تست بر میگرداند.

تفاوت بین بهینه‌سازهای مختلف مثل SGD، Momentum و Adam در نحوه به‌روزرسانی وزن‌ها با استفاده از گرادیان‌ها است. این تفاوت‌ها روی سرعت همگرایی، پایداری آموزش و کیفیت نتایج نهایی تأثیر زیادی می‌گذارند.

۱ - SGD (Stochastic Gradient Descent) ساده‌ترین روش

W = W - lr * dW

```
    if self.optimizer == "sgd":
        self.params[key] -= lr * grads[grad_key]
```

فقط از گرادیان جاری برای به‌روزرسانی استفاده می‌کند.

نوسان زیاد دارد، مخصوصاً اگر سطح خطا پر از دره و پستی‌بلندی باشد.

ممکن است در دره‌ها یا نزدیک مینیمم محلی گیر کند.

به نرخ یادگیری (lr) خیلی حساس است.

- ساده و سریع
- حافظه کمی مصرف می‌کند
- نوسانات زیاد در جهت‌گیری

- همگرایی کند

۲- Momentum تکانه(بهبود بر پایه‌ی سرعت)

$$v = \beta * v - lr * dW$$

$$W = W + v$$

```
elif self.optimizer == "momentum":
    self.velocities[key] = beta1 * self.velocities[key] - lr * grads[grad_key]
    self.params[key] += self.velocities[key]
```

علاوه بر گرادیان جاری، میانگین وزنی گرادیان‌های قبلی را هم نگه می‌دارد.

در جهت‌های پایدار سرعت می‌گیرد، در جهت‌های متغیر، کاهش نوسان دارد.

کمک می‌کند از مینیمم‌های محلی یا سطوح صاف عبور کند.

- همگرایی سریع‌تر ازSGD
- نوسانات کمتر
- نیاز به پارامتر β
- نسبت به Adam کمی حساس‌تر به تنظیمات

۳ — Adam (Adaptive Moment Estimation) ترکیبی از Momentum وRMSProp

$$m = \beta1 * m + (1 - \beta1) * dW$$

$$v = \beta2 * v + (1 - \beta2) * dW^2$$

$$W = W - lr * \hat{m} / (\sqrt{\hat{v}} + \varepsilon)$$

```
elif self.optimizer == "adam":
    self.m[key] = beta1 * self.m[key] + (1 - beta1) * grads[grad_key]
    self.v[key] = beta2 * self.v[key] + (1 - beta2) * (grads[grad_key] ** 2)

    m_hat = self.m[key] / (1 - beta1 ** t)
    v_hat = self.v[key] / (1 - beta2 ** t)

    self.params[key] -= lr * m_hat / (np.sqrt(v_hat) + eps)
```

از دو متغیر استفاده می‌کند:

m: میانگین نمایی گرادیان‌ها مثل (Momentum)

v: میانگین نمایی مربع گرادیان‌ها مثل(RMSProp)

هر وزن، نرخ یادگیری خاص خودش را دارد(adaptive learning rate)

- بسیار سریع و پایدار
- مناسب برای داده‌های noisy یاsparse
- معمولاً نیاز به تنظیم زیاد ندارد
- نسبت به SGD گاهی در مینیمم‌های دقیق‌تر گیر نمی‌افتد مقداری(over-adaptive)

تابع forward ورودی X(ماتریس داده) را از تمام لایه‌ها عبور می‌دهد.

در هر لایه:

- Z = A_prev @ W + b
- A = activation(Z)

در خروجی آخر، از softmax برای پیش‌بینی چندکلاسه استفاده می‌کند.

لیست caches شامل مقادیر میانی Z و A برای استفاده در Backprop است.

backward : محاسبه گرادیان‌ها باBackpropagation

grads = self.backward(X_batch, y_batch, caches)

از خروجی شروع می‌کند (لایه آخر):

خطایy - A = dZ :softmax + cross-entropy

سپس به عقب می‌رود:

برای هر لایه dA, dZ, dW, db

همه‌ی گرادیان‌ها را در grads ذخیره می‌کند.

تابع update_params به‌روزرسانی وزن‌ها

def update_params(self, grads, lr=None, t=1):

- به‌روزرسانی وزن‌ها با یکی از روش‌های:

sgd

momentum

adam

نتایج:

Relu

```
Epoch 45: Train Loss=1.4971, Acc=0.4713 | Val Loss=1.8080, Acc=0.4086
Epoch 46: Train Loss=1.5032, Acc=0.4692 | Val Loss=1.8143, Acc=0.4038
Epoch 47: Train Loss=1.4785, Acc=0.4754 | Val Loss=1.8012, Acc=0.4102
Epoch 48: Train Loss=1.5166, Acc=0.4691 | Val Loss=1.8343, Acc=0.4011
Epoch 49: Train Loss=1.4881, Acc=0.4724 | Val Loss=1.8194, Acc=0.4026
```

Sigmoid

```
Epoch 43: Train Loss=1.1493, Acc=0.5952 | Val Loss=1.6138, Acc=0.4751
Epoch 44: Train Loss=1.1491, Acc=0.5964 | Val Loss=1.6237, Acc=0.4740
Epoch 45: Train Loss=1.1471, Acc=0.5976 | Val Loss=1.6318, Acc=0.4740
Epoch 46: Train Loss=1.1455, Acc=0.5982 | Val Loss=1.6400, Acc=0.4749
Epoch 47: Train Loss=1.1449, Acc=0.5991 | Val Loss=1.6492, Acc=0.4752
Epoch 48: Train Loss=1.1447, Acc=0.5994 | Val Loss=1.6586, Acc=0.4734
Epoch 49: Train Loss=1.1445, Acc=0.5996 | Val Loss=1.6680, Acc=0.4739
```

برای مقایسه این دو روش هرکدام رابرای ۵۰ اپیزود آموززش دادیم. به طور کلی میتوان دید که سیگموید با نرخ یادگیری یکسان و تعداد ایپاک برابر تنها تفاوت در نقطه ی ابتدایی توانسته بهتر عمل کند. در اینجا این دو روش با بهینه ساز آدام آموزش دیده اند.

تفاوت بین سیگموید (Sigmoid) و (ReLU) در رفتار ریاضی آن‌ها هنگام فعال‌سازی نورون‌ها است. این دو تابع در لایه‌های پنهان شبکه‌های عصبی استفاده می‌شوند و تأثیر زیادی بر یادگیری، همگرایی و عملکرد نهایی مدل دارند.

Sigmoid

```
elif activation_name == 'sigmoid':
    s = 1 / (1 + np.exp(-z))
    return s * (1 - s)
```

خروجی بین 0 و 1

پیوسته و مشتق‌پذیر

حالت «s-shaped» دارد

مزایا:

- برای مدل‌های قدیمی مثل logistic regression خوب است
- مناسب برای خروجی‌های احتمال مثلاً در binary classification

معایب:

- **مشکل: "Vanishing Gradient"**
  - در مقادیر بزرگ یا خیلی منفی، مشتق به نزدیک صفر می‌رسد ← یادگیری کند می‌شود
- **خروجی همیشه مثبت:**
  - باعث می‌شود میانگین خروجی‌ها از صفر دور شود ← باعث نوسانات در گرادیان‌ها می‌شود

ReLU (Rectified Linear Unit)

```
if activation_name == 'relu':
    return (z > 0).astype(float)
```

خروجی در بازه $[0, \infty)$

در ورودی‌های منفی، خروجی صفر است

مزایا:

- بسیار سریع در محاسبه
- مشتق ساده 1 برای $x > 0$، و 0 برای $x < 0$
- تا حد زیادی Vanishing Gradient ندارد
- باعث sparse شدن فعال‌سازی‌ها می‌شود ← یادگیری سریع‌تر

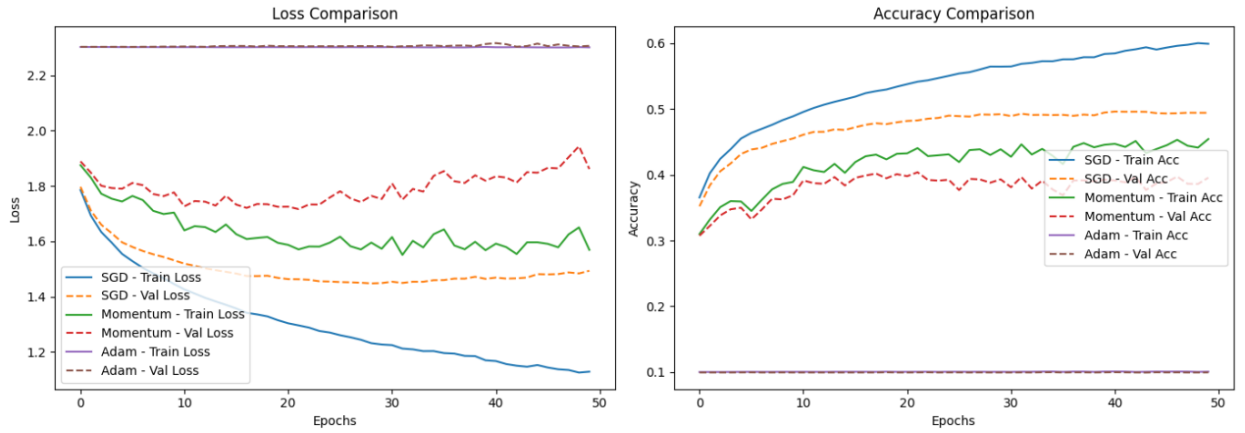
معایب:

- Dead ReLU Problem: اگر نورونی برای همیشه ورودی منفی بگیرد، گرادیانش صفر می‌شود و هیچ‌وقت فعال نمی‌شود.

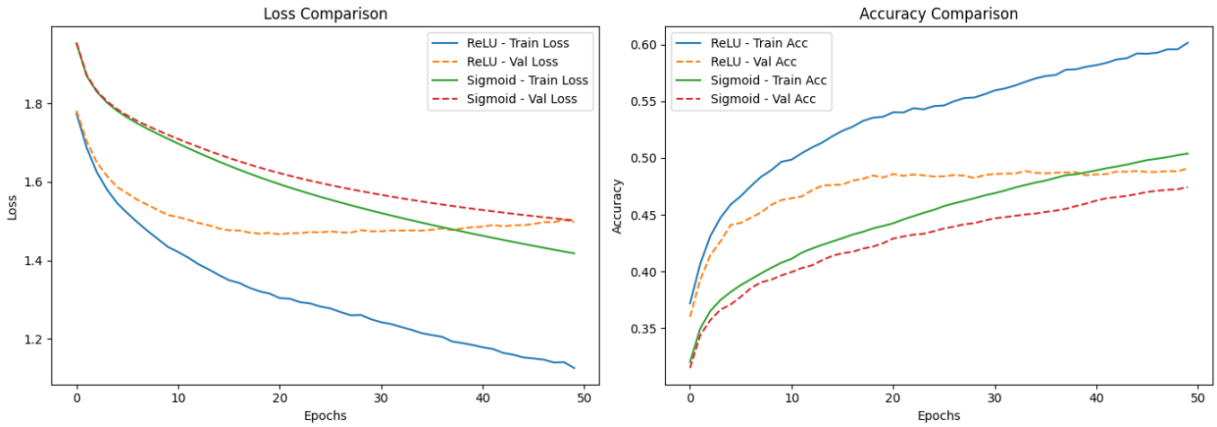
کدام بهتر است؟

- در شبکه‌های عصبی مدرن (به‌خصوص شبکه‌های عمیق)، ReLU تقریباً همیشه بهتر از Sigmoid است.
- Sigmoid فقط برای لایه‌ی خروجی binary classification یا کاربردهای خاص استفاده می‌شود.
- در این پروژه که شبکه‌ای با لایه‌های پنهان برای دسته‌بندی CIFAR-10 است، ReLU در لایه‌های پنهان بهتر عمل می‌کند، چون:
  - سریع‌تر یاد می‌گیرد
  - دچار مشکلات گرادیان نمی‌شود
  - عملکرد بهتری در دقت و loss دارد

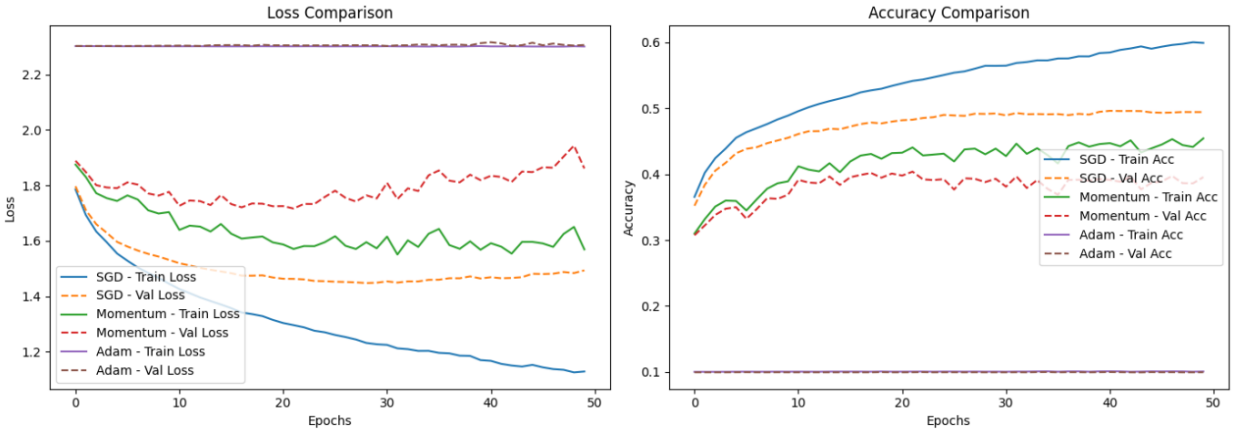
اما در بررسی ابتدایی وقتی که تعداد لایه ها را کم انتخاب کردیم (تنها یک لایه پنهان) و بهینه ساز adam را انتخاب کردیم پیشرفت خیلی کمی حاصل شد و سیگموید بهتر از relu عمل کرد.



اما با انتخاب بهینه ساز sgd کاملا مشهود است که در شبکه عصبی ما با فعالساز relu سریع تر و بهتر یاد میگیرد.



بهینه ساز های متفاوت



به طور کلی در این ۵۰ ایپاک برای هر کدام از روش های بهینه ساز گرادیان کاهشی استاندارد سریع تر و بهتر عمل کرده است. و ترکیب relu با sgd بهترین جواب را برای ما بدست آورده است.

momentum هم بعد از sgd عملکرد خوب ولی کندتری نشان داده. مشخص است که به نقاط بیشتری سر زده و برای بیرون آمدن از بهینه های محلی مناسب تر است.

اما به نظر میرسد Adam با ReLU دچار "گرادیان بسیار کوچک" شده است.

- Adam از m و v برای تصحیح گرادیان استفاده می‌کند.
- وقتی مقدار گرادیان‌ها خیلی کوچک باشد آپدیت وزن‌ها تقریباً صفر میشود ← آموزش متوقف میشود.

https://github.com/FZNjfi/Neural-network.git

لینک گیتهاب