دانشگاه اصفهان
دانشکده مهندسی کامپیوتر
گروه مهندسی هوش مصنوعی

مبانی هوش محاسباتی

سند پروژه سوم

اعضای گروه:

ابوالفضل رنجبر

میعاد کیمیاگری

امیر طاها نجف

استاد درس:

دکتر حسین کارشناس

بهار ۱۴۰۳

همه کد ها در این لینک قابل دسترس هستند:

https://colab.research.google.com/drive/1sCjgKEGqsNtVMaupgo_snbtLRVKvCkUs?usp=sharing

۱- رگرسیون لجستیک – آیا تصویر یک هواپیما است؟

بارگذاری و نرمال‌سازی دیتاست CIFAR-10 و بازبرچسب‌گذاری داده‌ها

در این بخش ابتدا داده‌های مجموعه CIFAR-10 بارگذاری و برچسب‌ها به صورت باینری (هواپیما/غیرهواپیما) تبدیل می‌شوند. سپس تصاویر به صورت آرایه‌های یک‌بعدی (تخت) و مقادیر پیکسل‌ها بین ۰ تا ۱ نرمال‌سازی می‌شوند تا برای مدل‌های یادگیری ماشین مناسب باشند. این پیش‌پردازش باعث می‌شود مدل رگرسیون لجستیک بتواند به راحتی روی داده‌ها آموزش ببیند و نتایج دقیق‌تری ارائه دهد.

```
# CIFAR
(x_train, y_train), (x_test, y_test) = cifar10.load_data()
y_train, y_test = y_train.flatten(), y_test.flatten()
# preprocess
y_train_bin = (y_train == 0).astype(int)
y_test_bin = (y_test == 0).astype(int)
# normalize and flatten
x_train_flat = x_train.reshape(x_train.shape[0], -1) / 255.0
x_test_flat = x_test.reshape(x_test.shape[0], -1) / 255.0
```

پیاده‌سازی توابع sigmoid و binary_cross_entropy_loss

در این بخش دو تابع کلیدی برای پیاده‌سازی رگرسیون لجستیک تعریف شده است. تابع اول، سیگموید(sigmoid) ، مقدار ورودی را به بازه بین صفر تا یک نگاشت می‌کند و برای محاسبه احتمال تعلق هر نمونه به کلاس مثبت استفاده می‌شود. تابع دوم، binary_cross_entropy، به عنوان تابع هزینه عمل می‌کند و اختلاف بین برچسب‌های واقعی و پیش‌بینی مدل را با استفاده از فرمول آنتروپی متقاطع باینری محاسبه می‌کند. همچنین برای جلوگیری از مشکلات عددی، مقادیر پیش‌بینی شده با استفاده از تابع clip محدود می‌شوند تا از محاسبه لگاریتم صفر جلوگیری شود. این دو تابع نقش اساسی در آموزش و ارزیابی مدل رگرسیون لجستیک دارند.

```
def sigmoid(z):
    return 1 / (1 + np.exp(-z))
def binary_cross_entropy(y_true, y_pred):
    y_pred = np.clip(y_pred, 1e-10, 1 - 1e-10)  # for numerical stability
    return -np.mean(y_true * np.log(y_pred) + (1 - y_true) * np.log(1 - y_pred))
```

پیاده‌سازی آموزش مدل با استفاده از گرادیان کاهشی (Gradient Descent)

در این بخش، حلقه آموزش مدل رگرسیون لجستیک پیاده‌سازی شده است. در هر تکرار(epoch) ، ابتدا خروجی مدل با استفاده از ضرب نقطه‌ای وزن‌ها و ویژگی‌های ورودی به همراه بایاس محاسبه می‌شود و سپس با تابع سیگموید به احتمال تعلق به کلاس مثبت تبدیل می‌گردد. مقدار خطا (loss) با استفاده از تابع آنتروپی متقاطع باینری محاسبه می‌شود. سپس گرادیان‌ها نسبت به وزن‌ها و بایاس محاسبه شده و پارامترهای مدل با استفاده از گرادیان نزولی و نرخ یادگیری مشخص به‌روزرسانی می‌شوند. این فرآیند تا رسیدن به تعداد epoch تعیین‌شده ادامه می‌یابد و مقدار خطا در هر مرحله نمایش داده می‌شود تا روند یادگیری مدل قابل مشاهده باشد.

```
lr = 0.02
epochs = 500
progress = trange(epochs, desc="Training")
for epoch in progress:
    z = np.dot(x_train_flat, W) + b
    y_pred = sigmoid(z)
    loss = binary_cross_entropy(y_train_bin, y_pred)
    # grads
    dz = y_pred - y_train_bin
    dW = np.dot(x_train_flat.T, dz) / len(x_train_flat)
    db = np.mean(dz)
    # update params
    W -= lr * dW
    b -= lr * db
    progress.set_postfix({"loss": f"{loss:.4f}"})
```

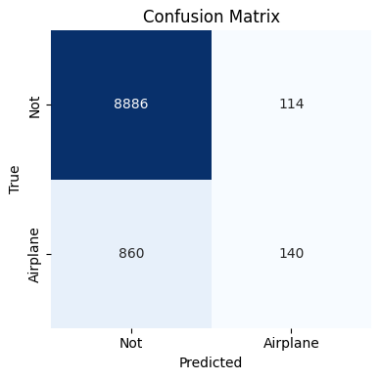
ارزیابی مدل با مجموعه‌داده آزمون با استفاده از confusion matrix و F1-Score

در این بخش، عملکرد مدل رگرسیون لجستیک آموزش‌دیده روی داده‌های تست ارزیابی می‌شود. ابتدا خروجی مدل برای داده‌های تست محاسبه شده و با استفاده از تابع سیگموید به احتمال تعلق به کلاس مثبت تبدیل می‌شود. سپس با آستانه ۰.۵، پیش‌بینی نهایی کلاس‌ها انجام می‌گیرد. مقدار خطا (loss) مدل روی داده‌های تست محاسبه و چاپ می‌شود. همچنین با استفاده از ماتریس آشفتگی(Confusion Matrix) ، دقت (Accuracy) و امتیاز F1 مدل ارزیابی می‌شود تا میزان صحت و کیفیت پیش‌بینی‌ها مشخص گردد. در ادامه، ماتریس آشفتگی به صورت تصویری نمایش داده می‌شود و برای بررسی کیفی، ۲۵ نمونه از تصاویر تست به همراه برچسب واقعی و پیش‌بینی‌شده نمایش داده می‌شوند. این ارزیابی‌ها به درک بهتر نقاط قوت و ضعف مدل کمک می‌کند.

خروجی:

```
Test Loss: 2.2427
Confusion Matrix:
[[8886  114]
 [ 860  140]]

F1 Score: 0.22328548644338117
Accuracy: 0.9026
```



۲- شبکه دودویی با یک لایه پنهان

۲-۱- کد Forward pass

از آن جا که مدل دارای یک لایه پنهان با ۶۴ نورون است، پس در واقع سه لایه داریم. لایه اول که همان لایه ورودی است عکسی که به صورت آرایه ای یک بعدی با سایز ۳۰۷۲ ($3 \times 32 \times 32$) در آمده را می‌گیرد. پس هر نورون لایه پنهان ۳۰۷۲ وزن به همراه یک بایاس دارد. از آنجا که مسئله رگرسیون لجستیک است در لایه خروجی فقط یک نورون داریم که به همه نورون‌های لایه پنهان متصل است پس ۶۴ وزن به همراه یک بایاس دارد. برای انجام محاسبات روی GPU از Tensorflow استفاده کردیم. (توجه کنید که پیاده سازی به صورت scratch است و صرفا برای محاسبات از کتابخانه استفاده شده است.)

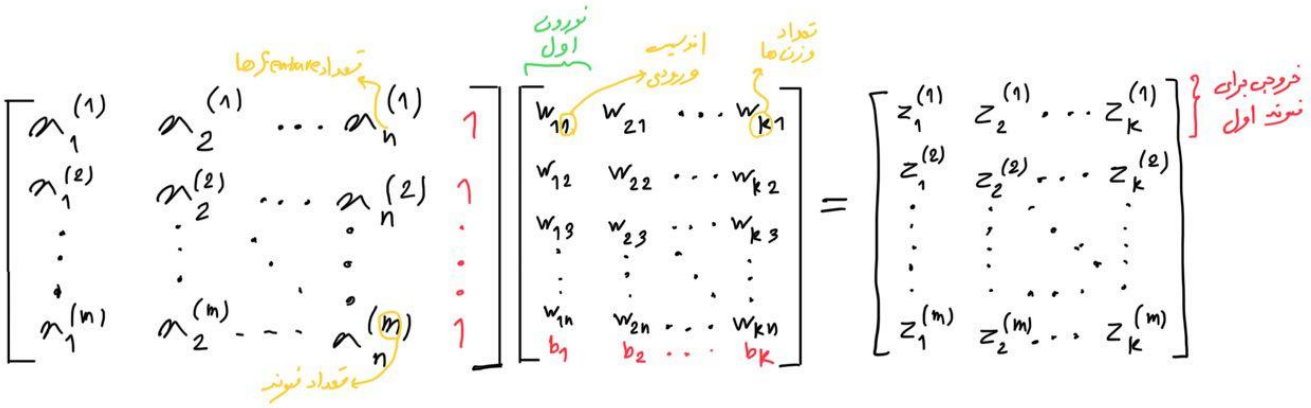
برای forward pass، متودی به نام fp تعریف کردیم که عکس‌هاس flatten شده را به عنوان آرگمان x ورودی می‌گیرد. از آنجا که محاسبات را به صورت vectorize انجام می‌دهیم، سایز x برابر است با (batch_size, 3072)

حالا باید رابطه اعمال پارامترها برای یک لایه را پیدا کنیم. می‌دانیم که برای یک نورون که خروجی آن z است از رابطه زیر استفاده می‌شود:

$$z = w_1 x_1 + w_2 x_2 + \cdots + \text{bias}$$

$$a = \text{activation\_function}(z)$$

برای این که مرحله اول را با یک ضرب ماتریسی، برای تمام نورون‌های یک لایه انجام بدهیم، کافی است ماتریس W را transpose کنیم تا هر ستون آن پارامترهای یک نورون را نشان بدهد و سپس آن را از سمت راست در ماتریس X که هر سطر آن یک نمونه است ضرب کنیم. یعنی داریم:



پس برای پیاده سازی کافی است ضرب ماتریسی بالا را برای لایه پنهان انجام دهیم، مقدار بردار بایاس را به هر سطر از ماتریس نتیجه که خروجی‌های نرون‌های مختلف برای یک ورودی است اضافه کنیم و در نهایت تابع فعال سازی را اعمال کنیم. این کار را برای لایه خروجی هم انجام می‌دهیم. یعنی داریم:

```
    def fp(self, x):
        z1 = tf.matmul(x, self.W_l1) + self.b_l1
        a1 = self.sigmoid(z1)

        z2 = tf.matmul(a1, self.W_l2) + self.b_l2
        a2 = self.sigmoid(z2)

        return a1, a2
```

a1 همان خروجی لایه اول و a2 خروجی لایه دوم است.

۲-۲- حساب کردن خطا

از binary cross entropy استفاده می‌کنیم. متود loss، پیش‌بینی ها یعنی y_hat و برچسب‌ها یعنی y را ورودی می‌گیرد. پس داریم:

$$\text{loss}(y, \hat{y}) = -\frac{1}{m}\sum[y\log(\hat{y}+\varepsilon) + (1-y)\log(1-\hat{y}+\varepsilon)]$$

اپسیلون برای جلوگیری از log(0) اضافه شده است.

برای پیاده سازی داریم:

```
    @staticmethod
    def loss(y, y_hat):
        eps = 1e-10
        return -tf.reduce_mean(y * tf.math.log(y_hat + eps) \
                               +(1 - y) * tf.math.log(1 - y_hat + eps))
```

۲-۳- حساب کردن گرادیان‌ها

مرحله اول: پیدا کردن گرادیان محلی لایه خروجی:

$$\hat{y} = \sigma(z^{[2]})$$

$$\delta^{[2]} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z^{[2]}}$$

مرحله ۲: پیدا کردن مشتقات نسبت به‌ورن‌ها و بایاس لایه دو:

$$\frac{\partial L}{\partial w^{[2]}} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z^{[2]}} \frac{\partial z^{[2]}}{\partial w^{[2]}} = \delta^{[2]} a^{[1]}$$

$$\frac{\partial L}{\partial b^{[2]}} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z^{[2]}} \frac{\partial z^{[2]}}{\partial b^{[2]}} = \delta^{[2]}$$

مرحله سوم: پیدا کردن گرادیان‌های محلی لایه پنهان:

$$\delta^{[1]} = \delta^{[2]} \frac{\partial z^{[2]}}{\partial a^{[1]}} \frac{\partial a^{[1]}}{\partial z^{[1]}}$$

مرحله چهارم: پیدا کردن مشتقات نسبت به وزن‌ها و بایاس لایه یک:

$$\frac{\partial L}{\partial w^{[1]}} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z^{[2]}} \frac{\partial z^{[2]}}{\partial a^{[1]}} \frac{\partial a^{[1]}}{\partial z^{[1]}} \frac{\partial z^{[1]}}{\partial w^{[1]}} = \delta^{[1]} x$$

$$\frac{\partial L}{\partial b^{[1]}} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z^{[2]}} \frac{\partial z^{[2]}}{\partial a^{[1]}} \frac{\partial a^{[1]}}{\partial z^{[1]}} \frac{\partial z^{[1]}}{\partial b^{[1]}} = \delta^{[1]}$$

برای تابع فعال سازی سیگموید داریم:

$$\sigma(z) = \frac{1}{1+e^{-z}} \Rightarrow \sigma'(z) = \frac{d}{dz}\left(\frac{1}{1+e^{-z}}\right) = \frac{e^{-z}}{(1+e^{-z})^2} = \sigma(z)\big(1-\sigma(z)\big)$$

با ترکیب گرادیان سیگموید و باینری کراس انتروپی داریم:

$$\frac{\partial L}{\partial z^{[2]}} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z^{[2]}} = \left(-\frac{y}{\hat{y}} + \frac{1-y}{1-\hat{y}}\right) \cdot \hat{y}(1-\hat{y}) = \hat{y} - y$$

پس برای پیاده سازی داریم:

```
    @staticmethod
    def loss_backward(y, y_hat):
        return y_hat - y
    @staticmethod
    def sigmoid_backward(x):
        return x * (1 - x)
```

```
    def bp(self, x, y, y_hat, a1):
        m = tf.cast(tf.shape(x)[0], tf.float32)

        # L2
        delta2 = self.loss_backward(y, y_hat)
        dw2 = tf.matmul(tf.transpose(a1), delta2) / m
        db2 = tf.reduce_sum(delta2, axis=0, keepdims=True) / m

        # L1
        delta1 = tf.matmul(delta2, tf.transpose(self.W_l2)) *
self.sigmoid_backward(a1)
        dw1 = tf.matmul(tf.transpose(x), delta1) / m
        db1 = tf.reduce_sum(delta1, axis=0, keepdims=True) / m

        return {'dw1': dw1, 'db1': db1, 'dw2': dw2, 'db2': db2}
```

۲-۴- آموزش مدل

تابع fit مسئول آموزش شبکه است. این تابع ابتدا داده‌های ورودی و برچسب‌ها را به تنسور تبدیل کرده و شکل y را به صورت ستونی تنظیم می‌کند. سپس وزن‌ها و بایاس‌های دو لایه (لایه پنهان و لایه خروجی) با استفاده از مقداردهی اولیه Xavier مقداردهی می‌شوند. در طول هر دوره (epoch)، اگر پارامتر shuffle فعال باشد، داده‌ها به‌صورت تصادفی بازچینی می‌شوند. سپس داده‌ها به مینی‌بچ‌هایی با اندازه مشخص تقسیم می‌شوند و برای هر مینی‌بچ، عملیات پیش‌رو (forward pass) و پس‌انتشار خطا (backpropagation) انجام شده و وزن‌ها به‌روزرسانی می‌گردند. در انتهای هر مینی‌بچ، مقدار تابع هزینه (loss) محاسبه و ثبت می‌شود و میانگین خطا در انتهای هر epoch گزارش می‌شود. این روند تا اتمام تعداد epoch تعیین‌شده ادامه می‌یابد.

```
    def fit(self, X, y, epochs=20, lr=0.1, batch_size=64, shuffle=True):

        for epoch in range(epochs):
            if shuffle:
                idx = tf.random.shuffle(idx)

            losses = []
            with trange(0, n_samples, batch_size, desc=f"Epoch
{epoch+1}/{epochs}") as t:
                for start in t:
                    end = start + batch_size
                    batch_idx = idx[start:end]
                    xb = tf.gather(X, batch_idx)
                    yb = tf.gather(y, batch_idx)

                    a1, a2 = self.fp(xb)
                    grads = self.bp(xb, yb, a2, a1)

                    self.W_l2 -= lr * grads['dw2']
                    self.b_l2 -= lr * grads['db2']
                    self.W_l1 -= lr * grads['dw1']
                    self.b_l1 -= lr * grads['db1']

                    loss = self.loss(yb, a2)
                    losses.append(loss.numpy())

                    t.set_postfix(loss=f"{loss.numpy():.4f}")
```

۲-۵- ارزیابی مدل

روی دیتاست آزمون مقادیر زیر به دست آمد:

```
Test Loss: 0.4564
Confusion Matrix:
[[8826  174]
 [ 610  390]]

F1 Score: 0.4987
Accuracy: 0.9216
```

پس دقت روی دیتاست تست برابر با ۹۲ درصد شد.

۳- طبقه‌بندی‌کننده چند کلاسه

در این بخش، داده‌های مجموعه CIFAR-10 بارگذاری می‌شوند و برچسب‌های کلاس به صورت آرایه یک‌بعدی تبدیل می‌گردند. سپس با استفاده از تابع `to_categorical`، برچسب‌ها به صورت one-hot (مناسب برای مسائل چندکلاسه) در می‌آیند تا برای آموزش شبکه‌های عصبی قابل استفاده باشند. در ادامه، تصاویر که به صورت آرایه‌های سه‌بعدی (تعداد نمونه، ارتفاع، عرض، کانال رنگی) هستند، به آرایه‌های دوبعدی (تعداد نمونه، تعداد کل پیکسل‌ها) تخت (flatten) می‌شوند و مقادیر پیکسل‌ها با تقسیم بر ۲۵۵ بین ۰ تا ۱ نرمال‌سازی می‌گردند. این پیش‌پردازش‌ها باعث می‌شود داده‌ها برای مدل‌های یادگیری ماشین و شبکه‌های عصبی آماده شوند و فرآیند آموزش به شکل بهینه‌تری انجام گیرد.

```
(x_train, y_train), (x_test, y_test) = cifar10.load_data()
y_train, y_test = y_train.flatten(), y_test.flatten()
# one hot
y_train_one_hot = to_categorical(y_train, 10)
y_test_one_hot = to_categorical(y_test, 10)
# normalize and flatten images
x_train_flat = x_train.reshape(x_train.shape[0], -1) / 255.0
x_test_flat = x_test.reshape(x_test.shape[0], -1) / 255.0
```

سه تابع اصلی برای شبکه عصبی چندکلاسه تعریف شده است:

cross_entropy_loss

این تابع برای محاسبه خطای آنتروپی متقاطع در مسائل چندکلاسه استفاده می‌شود. ابتدا مقادیر پیش‌بینی شده (y_pred) برای جلوگیری از مشکلات عددی بین یک بازه کوچک محدود (clip) می‌شوند. سپس با استفاده از فرمول آنتروپی متقاطع، میانگین خطا برای کل نمونه‌ها محاسبه می‌شود تا میزان اختلاف بین خروجی مدل و برچسب‌های واقعی به دست آید.

```
    def cross_entropy_loss(self, y_true, y_pred):
        eps = 1e-15
        y_pred = tf.clip_by_value(y_pred, eps, 1 - eps)
```

```
        return -tf.reduce_mean(tf.reduce_sum(y_true * tf.math.log(y_pred), axis=1))
```

forward

این تابع مرحله پیش‌رو (forward pass) شبکه را انجام می‌دهد. ابتدا ورودی X با وزن‌ها و بایاس لایه اول ترکیب شده و از تابع سیگموید عبور داده می‌شود تا خروجی لایه مخفی (a1) به دست آید. سپس خروجی لایه مخفی وارد لایه دوم شده و پس از اعمال تابعsoftmax ، خروجی نهایی شبکه (a2) تولید می‌شود که احتمال تعلق هر نمونه به هر کلاس را نشان می‌دهد.

```
    def forward(self, X):
        self.z1 = tf.matmul(X, self.W1) + self.b1
        self.a1 = self.sigmoid(self.z1)
        self.z2 = tf.matmul(self.a1, self.W2) + self.b2
        self.a2 = self.softmax(self.z2)
        return self.a2
```

backpropagation

این تابع مرحله پس‌انتشار خطا (backpropagation) را پیاده‌سازی می‌کند. ابتدا خطای خروجی (delta_output) با کم کردن مقدار پیش‌بینی شده از مقدار واقعی به دست می‌آید. سپس خطای لایه مخفی (delta_hidden) با استفاده از گرادیان زنجیره‌ای و مشتق سیگموید محاسبه می‌شود. گرادیان وزن‌ها و بایاس‌ها برای هر دو لایه محاسبه شده و پارامترها با استفاده از گرادیان نزولی و نرخ یادگیری به‌روزرسانی می‌شوند تا مدل بهینه‌تر شود.

```
    def backpropagation(self, X, y, y_pred, learning_rate):
        batch_size = tf.cast(tf.shape(X)[0], tf.float32)
        delta_output = y_pred - y
        delta_hidden = tf.matmul(delta_output, tf.transpose(self.W2)) * self.sigmoid_derivative(self.a1)
        dW2 = tf.matmul(tf.transpose(self.a1), delta_output) / batch_size
        db2 = tf.reduce_mean(delta_output, axis=0, keepdims=True)
        dW1 = tf.matmul(tf.transpose(X), delta_hidden) / batch_size
        db1 = tf.reduce_mean(delta_hidden, axis=0, keepdims=True)
        self.W2.assign_sub(learning_rate * dW2)
        self.b2.assign_sub(learning_rate * db2)
        self.W1.assign_sub(learning_rate * dW1)
        self.b1.assign_sub(learning_rate * db1)
```

این سه تابع هسته اصلی یادگیری شبکه عصبی چندکلاسه را تشکیل می‌دهند و باعث می‌شوند مدل بتواند با داده‌های آموزشی تطبیق پیدا کند و عملکرد خود را بهبود دهد.

تابع آموزش

تابع train مسئول آموزش شبکه عصبی چندکلاسه است. ابتدا داده‌های ورودی و برچسب‌ها به فرمت TensorFlow تبدیل می‌شوند و تعداد نمونه‌ها تعیین می‌شود. سپس حلقه آموزش به تعداد epoch مشخص اجرا می‌شود و در هر epoch داده‌ها به صورت تصادفی مخلوط می‌شوند. داده‌ها به دسته‌های کوچک (batch) تقسیم شده و برای هر دسته، پیش‌بینی مدل انجام و سپس با استفاده از تابع backpropagation وزن‌ها و بایاس‌ها به‌روزرسانی می‌شوند.

در پایان هرepoch ، کل داده‌ها یک بار از مدل عبور داده می‌شوند تا مقدار خطای آنتروپی متقاطع (cross-entropy loss) محاسبه و در لیست تاریخچه خطا ذخیره شود. این روند تا پایان همهepoch ها ادامه پیدا می‌کند و در نهایت تاریخچه خطا برگردانده می‌شود.

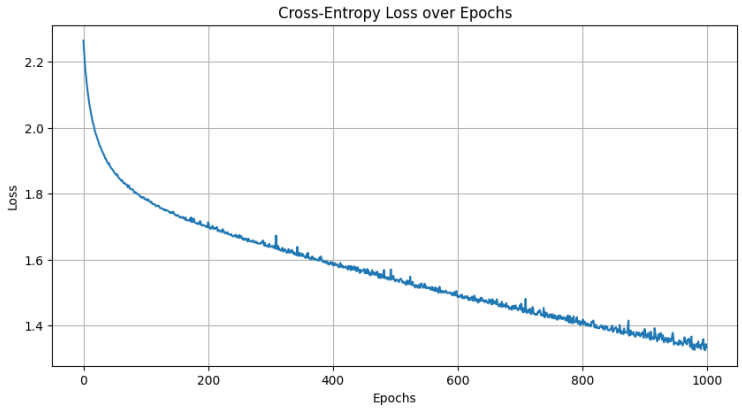
توابع predict و predict_classes نیز برای پیش‌بینی خروجی مدل روی داده‌های جدید استفاده می‌شوند؛ predict خروجی softmax مدل را برمی‌گرداند و predict_classes کلاس نهایی هر نمونه را با گرفتن بیشترین مقدار خروجی softmaxمشخص می‌کند.

```
def train(self, X, y, epochs=20, batch_size=128, learning_rate=0.01):
    X = tf.convert_to_tensor(X, dtype=tf.float32)
    y = tf.convert_to_tensor(y, dtype=tf.float32)
    n_samples = X.shape[0]
    loss_history = []
    progress = trange(epochs, desc="Training")
    for epoch in progress:
        indices = tf.random.shuffle(tf.range(n_samples))
        X_shuffled = tf.gather(X, indices)
        y_shuffled = tf.gather(y, indices)
        for i in range(0, n_samples, batch_size):
            X_batch = X_shuffled[i:i+batch_size]
            y_batch = y_shuffled[i:i+batch_size]
            y_pred = self.forward(X_batch)
            self.backpropagation(X_batch, y_batch, y_pred, learning_rate)
        y_pred = self.forward(X)
        loss = self.cross_entropy_loss(y, y_pred).numpy()
        loss_history.append(loss)
        progress.set_postfix({"loss": f"{loss:.4f}"})
    return loss_history
def predict(self, X):
    X = tf.convert_to_tensor(X, dtype=tf.float32)
    return self.forward(X)
def predict_classes(self, X):
    return tf.argmax(self.predict(X), axis=1).numpy()
```

آموزش و ارزیابی مدل

در این بخش، ابتدا ابعاد ورودی، تعداد نورون‌های لایه مخفی و تعداد کلاس‌ها برای شبکه عصبی تعیین می‌شود. سپس یک نمونه از کلاس MultiClassNN ساخته می‌شود. برای کاهش زمان آموزش، فقط ۱۰۰۰۰ نمونه اول داده‌های آموزش و برچسب‌های one-hotبه مدل داده می‌شود و مدل به مدت ۱۰۰۰ دوره با batch سایز ۲۵۶ و نرخ یادگیری ۰.۰۱ آموزش می‌بینند. پس از پایان آموزش، تاریخچه خطا (loss) در هر epoch ذخیره شده و با استفاده از matplotlib روند کاهش خطا در طول آموزش رسم می‌شود تا بتوان پیشرفت مدل را مشاهده کرد.

```
input_size = x_train_flat.shape[1]  # 3072 for CIFAR-10
hidden_size = 128
output_size = 10  # 10 classes
model = MultiClassNN(input_size, hidden_size, output_size)
subset_size = 10000
loss_history = model.train(
    x_train_flat[:subset_size],
    y_train_one_hot[:subset_size],
    epochs=1000,
    batch_size=256,
    learning_rate=0.01
)
```




```
Classification Report:
              precision    recall  f1-score   support

           0       0.51      0.49      0.50      1000
           1       0.53      0.49      0.51      1000
           2       0.33      0.32      0.33      1000
           3       0.35      0.20      0.25      1000
           4       0.34      0.39      0.36      1000
           5       0.39      0.29      0.33      1000
           6       0.44      0.52      0.47      1000
           7       0.43      0.54      0.48      1000
           8       0.53      0.60      0.56      1000
           9       0.44      0.48      0.46      1000

    accuracy                           0.43     10000
   macro avg       0.43      0.43      0.43     10000
weighted avg       0.43      0.43      0.43     10000
```

۴- چالش پیاده‌سازی نیمه پیشرفته

۴-۱- پیاده سازی کلاس ماژولر

کلاس Layer

این کلاس یک لایه‌ی پایه برای یک شبکه عصبی تعریف می‌کند که قابلیت گسترش توسط کلاس‌های فرزند را دارد. سازنده‌ی کلاس تعداد نورون‌ها را به عنوان ورودی می‌پذیرد و در صورت مشخص بودن، بایاس اولیه را مقداردهی می‌کند. سه متد انتزاعی build، forward و backward تعریف شده‌اند که به ترتیب برای ساخت وزن‌ها بر اساس ورودی، اجرای مرحله‌ی پیش‌رو (forward pass)، و اجرای مرحله‌ی پس‌رو (backward pass) استفاده می‌شوند و باید در کلاس‌های فرزند پیاده‌سازی شوند. همچنین، متد __call__ بررسی می‌کند که آیا لایه تاکنون ساخته شده یا نه؛ اگر نه، ابتدا آن را می‌سازد و سپس عملیات forward را انجام می‌دهد.

```
class Layer(abc.ABC):
    def __init__(self, n_neurons):
        self.n_neurons = n_neurons
        self.W: tf.Variable | None = None
        if n_neurons:
            self.b = tf.Variable(tf.zeros((1, n_neurons), tf.float32))
        else:
            self.b = None
        self._built: bool = False

    @abc.abstractmethod
    def build(self, input_shape): pass
    @abc.abstractmethod
    def forward(self, x: tf.Tensor, training: bool) -> tf.Tensor: pass
    @abc.abstractmethod
    def backward(self, prev_grads: tf.Tensor) -> tf.Tensor: pass
    def __call__(self, x: tf.Tensor, training: bool) -> tf.Tensor:
        if not self._built:
            self.build(x.shape)
            self._built = True

        return self.forward(x, training)
```

کلاس Dense

در این کلاس Dense که از کلاس پایه‌ی Layer ارث‌بری می‌کند، یک لایه‌ی کامل (Fully Connected) شبکه عصبی پیاده‌سازی شده است. در متد build، وزن‌ها W با استفاده از مقداردهی اولیه He (مناسب برای توابع فعال‌سازی ReLU)

مقداردهی می‌شوند، به طوری که انحراف معیار برابر با sqrt(2 / input_dim) است. در متد forward، ضرب ماتریسی بین ورودی x و وزن‌ها انجام شده و بایاس b به آن افزوده می‌شود تا خروجی لایه تولید شود. متد backward گرادیان‌های لازم برای به‌روزرسانی پارامترها را محاسبه می‌کند: dW گرادیان نسبت به وزن‌ها، db گرادیان نسبت به بایاس، و خروجی این متد گرادیان نسبت به ورودی لایه است که برای لایه‌ی قبلی در بک‌پراپگیشن استفاده می‌شود.

```
class Dense(Layer):
    def __init__(self, n_neurons):  super().__init__(n_neurons)
    def build(self, input_shape):
        in_dim = input_shape[-1]
        std = tf.sqrt(2.0 / in_dim)
        self.W = tf.Variable(tf.random.normal((in_dim, self.n_neurons),
                             stddev=std))

    def forward(self, x: tf.Tensor, training=False) -> tf.Tensor:
        self.x = x
        return tf.matmul(x, self.W) + self.b

    def backward(self, prev_grads: tf.Tensor) -> tf.Tensor:
        self.dW = tf.matmul(tf.transpose(self.x), prev_grads)
       # prev_grads -> (batch, out_size), x-> (batch, in_dim)
        self.db = tf.reduce_sum(prev_grads, axis=0, keepdims=True)
       # sum over the batch
# w -> (in_dim, out_dim(n_neurons)), prev_grads -> (batch, out_dim)
        return tf.matmul(prev_grads, tf.transpose(self.W))
```

کلاس Activation

کلاس Activation یک کلاس انتزاعی برای لایه‌های فعال‌سازی در شبکه‌های عصبی است که از کلاس پایه Layer ارث‌بری می‌کند. چون این نوع لایه‌ها پارامتر قابل آموزش (مانند وزن یا بایاس) ندارند، در سازنده n_neurons برابر با None قرار داده شده و متد build نیز عملی ندارد. دو متد انتزاعی _func و _func_backward به‌ترتیب برای محاسبه تابع فعال‌سازی و مشتق آن تعریف شده‌اند که باید در کلاس‌های فرزند (مثل ReLU یا Sigmoid) پیاده‌سازی شوند. در متد forward مقدار ورودی ذخیره شده و سپس تابع فعال‌سازی بر آن اعمال می‌شود. متد backward مشتق تابع فعال‌سازی را محاسبه کرده و آن را در گرادیان‌های مرحله‌ی بعدی ضرب می‌کند تا گرادیان نسبت به ورودی لایه محاسبه شود.

```
class Activation(Layer):
    def build(self, input_shape):

    @abc.abstractmethod
```

```
    def _func(self, x):pass

    @abc.abstractmethod
    def _func_backward(self, x): pass

    def forward(self, x: tf.Tensor, training=False):
        self.x = x
        return self._func(x)

    def backward(self, prev_grads: tf.Tensor):
        return prev_grads * self._func_backward(self.x)
```

فعال‌ساز ReLU

کلاس ReLU یک پیاده‌سازی مشخص از کلاس انتزاعی Activation است که تابع فعال‌سازی ReLU (Rectified Linear Unit) را پیاده می‌کند. در متد _func، مقدار ورودی با صفر مقایسه شده و بزرگ‌تر از صفر حفظ می‌شود، در حالی که مقادیر منفی به صفر تبدیل می‌شوند. در متد _func_backward، مشتق تابع ReLU محاسبه می‌شود که برای مقادیر مثبت برابر با ۱ و برای مقادیر صفر یا منفی برابر با ۰ است. این مشتق در فرآیند backpropagation برای انتقال گرادیان‌ها استفاده می‌شود.

```
class ReLU(Activation):
    def __init__(self):
        super().__init__()

    def _func(self, x):
        return tf.maximum(x, 0.0)

    def _func_backward(self, x):
        return tf.cast(x > 0, tf.float32)
```

فعال‌ساز Softmax

کلاس Softmax یک پیاده‌سازی از تابع فعال‌سازی Softmax است که برای لایه‌های خروجی در مسائل دسته‌بندی (classification) استفاده می‌شود. در متد _func، ابتدا برای پایداری عددی، بیشینه مقدار هر نمونه از ورودی کم می‌شود، سپس نمایی گرفته شده و نرمال‌سازی بر اساس مجموع نمایی‌ها انجام می‌شود تا خروجی به‌صورت احتمال بین صفر و یک باشد و مجموع آن در هر نمونه برابر ۱ شود. در متد _func_backward، به‌طور پیش‌فرض مقدار tf.ones_like(x) بازگردانده می‌شود؛ این فرض زمانی معتبر است که از تابع Softmax در ترکیب با تابع هزینه‌ی Cross-Entropy استفاده شود، چرا که مشتق ترکیبی این دو تابع ساده می‌شود و نیازی به مشتق کامل Softmax نیست. این ساده‌سازی بهینه‌سازی و پیاده‌سازی را راحت‌تر می‌کند.

```
class Softmax(Activation):
```

```
  def __init__(self):
    super().__init__()

  def _func(self, x):
    exp_x = tf.exp(x - tf.reduce_max(x, axis=1, keepdims=True))
    return exp_x / tf.reduce_sum(exp_x, axis=1, keepdims=True)

  def _func_backward(self, x):
    # assuming this is used with cross entropy
    return tf.ones_like(x)
```

فعال‌ساز Tanh

کلاس فعال‌سازی تانژانت هیپربولیک (tanh) است که از کلاس Activation ارث‌بری می‌کند. در متد _func، تابع tanh بر ورودی اعمال می‌شود که مقادیر را به بازه‌ی بین -۱ تا ۱ نگاشت می‌کند. در متد _func_backward، مشتق تابع tanh محاسبه می‌شود که برابر است با ۱ - tanh(x)^2 و در مرحله‌ی backpropagation برای محاسبه گرادیان نسبت به ورودی استفاده می‌شود. تابع tanh به‌ویژه در مدل‌هایی که به داده‌هایی با میانگین صفر حساس هستند کاربرد دارد، زیرا خروجی آن حول صفر متقارن است.

```
class ActivationTanh(Activation):
    def _func(self, x):
        return tf.math.tanh(x)

    def _func_backward(self, x):
        return 1.0 - tf.math.tanh(x)**2
```

کلاس CategoricalCrossEntropy

کلاس CategoricalCrossEntropy یک تابع هزینه (Loss Function) برای مسائل دسته‌بندی چندکلاسه است که از کلاس پایه‌ی Loss ارث‌بری می‌کند. در متد__call__ ، ابتدا برای پایداری عددی، بیشینه هر نمونه از logits کم می‌شود، سپس لگاریتم احتمال‌های softmax (log-softmax) محاسبه می‌گردد. حاصل ضرب این لگاریتم‌ها با مقادیر y_true که به‌صورت one-hotاست (انجام شده و سپس میانگین منفی آن‌ها به‌عنوان مقدار نهایی loss بازگردانده می‌شود.

در متد backward، توزیع softmax از روی logits بازسازی می‌شود و گرادیان نسبت به ورودی تابع softmax (که همان logits است) با استفاده از رابطه‌ی $\partial L/\partial z = p - y$ محاسبه می‌شود؛ این گرادیان بر اندازه‌ی batch تقسیم می‌شود تا میانگین گرادیان‌ها به‌دست آید. این ساده‌سازی به‌دلیل خاصیت ترکیب Softmax با Cross-Entropy است که مشتق را بسیار ساده و محاسبه‌پذیر می‌کند.

```
class CategoricalCrossEntropy(Loss):
    def __call__(self, y_true: tf.Tensor, logits: tf.Tensor) -> tf.Tensor:
        # y_true -> [batch, n_classes(one_hot)]
        # logits -> [batch, z(out_dim)]

        # shift for stability
        z = logits - tf.reduce_max(logits, axis=1, keepdims=True)
        # log probs
        logp = z - tf.math.log(tf.reduce_sum(tf.exp(z), axis=1,
keepdims=True))    # log(e^z_i/sigma(e^z_i)) = z_i - log(sigma(e^z_i))

        return -tf.reduce_mean(tf.reduce_sum(y_true * logp, axis=1))

    def backward(self, y_true: tf.Tensor, logits: tf.Tensor) -> tf.Tensor:
        # recreate softmax
        z = logits - tf.reduce_max(logits, axis=1, keepdims=True)
        exp_z = tf.exp(z)
        p = exp_z / tf.reduce_sum(exp_z, axis=1, keepdims=True)

        # gradient w.r.t. logits
        batch = tf.cast(tf.shape(y_true)[0], tf.float32)
        return (p - y_true) / batch
```

بهینه‌ساز SGD

کلاس SGD (Stochastic Gradient Descent) یک پیاده‌سازی ساده از بهینه‌ساز مبتنی بر گرادیان نزولی است که از کلاس انتزاعی Optimizer ارث‌بری می‌کند. در سازنده، نرخ یادگیری (lr) مشخص می‌شود که تعیین‌کننده اندازه گام به‌روزرسانی پارامترها است.

در متد apply، هر پارامتر p با استفاده از گرادیان متناظر g به‌روزرسانی می‌شود، به‌طوری‌که از مقدار فعلی آن، lr * g کم می‌شود (p.assign_sub(...)). این عملیات منجر به حرکت پارامترها در جهت عکس گرادیان شده و به تدریج تابع هزینه را کاهش می‌دهد. این ساده‌ترین شکل یادگیری گرادیانی است و پایه بسیاری از بهینه‌سازهای پیشرفته‌تر محسوب می‌شود.

```
class SGD(Optimizer):
  def __init__(self, lr: float=0.1):
    self.lr = lr

  def apply(self, params, grads):
    for p, g in zip(params, grads):
      p.assign_sub(self.lr * g)
```

بهینه‌ساز Momentum

تفاوت مومنتوم یا SGD این است که با استفاده از Exponentially Weighted Moving Average، میانگینی از گرادیان‌های قبلی را ذخیره می‌کند، با میانگین این مرحله جمع می‌کند و سپس آپدیت را طبق فرمول زیر انجام می‌دهد. مومنتوم نسبت به SGD ساده همگرایی سریع‌تری دارد، به‌ویژه در مسیرهای با نوسانات زیاد. در واقع، مومنتوم نوعی حافظه به الگوریتم اضافه می‌کند؛ به این صورت که اگر گرادیان‌ها در چند گام متوالی تقریبا در یک جهت باشند، سرعت به‌روزرسانی در آن جهت افزایش می‌یابد. این ویژگی به مدل کمک می‌کند که سریع‌تر در مسیر درست حرکت کند و از نوسانات زیاد در جهات اشتباه جلوگیری کند.

$$(v_t = \mu v_{t-1} + \eta g_t, \quad \theta_t = \theta_{t-1} - v_t)$$

در پیاده‌سازی زیر برای هر تنسور از پارامترها مومنتم جدا را در یک دیکشنری ذخیره می‌کنیم. در ابتدای کار مقادیر این مومنتم‌ها برابر صفر است. پس از آپدیت پارامترها در هر مرحله، بردار مومنتوم را هم در دیکشنری به‌روزرسانی می‌کنیم. هایپرپارامتر momentum هم مقدار تاثیر مونتوم در آپدیت وزن‌های هر مرحله را مشخص می‌کند.

```
class Momentum(Optimizer):
    def __init__(self, lr: float = 0.01, momentum: float = 0.9):
        self.lr = lr
        self.momentum = momentum
        self.velocities: dict[int, tf.Tensor] = {}

    def apply(self, params: list[tf.Variable], grads: list[tf.Tensor]):
        for p, g in zip(params, grads):
            key = p.ref()
            v = self.velocities.get(key, tf.zeros_like(p))

            # v_t = momentum·v_{t-1} + lr·g_t
            v = self.momentum * v + self.lr * g
            p.assign_sub(v)
            self.velocities[key] = v
```

بهینه‌ساز Adam

قبل از درک الگوریتمAdam، بهتر است با Adagrad آشنا شویم.

ایده‌ی Adagrad این است که برخی پارامترها ممکن است گرادیان‌های بسیار بزرگ‌تری نسبت به بقیه داشته باشند، بنابراین منطقی نیست که همه‌ی آن‌ها با یک نرخ یادگیری (learning rate) به‌روزرسانی شوند. برای حل این مشکل، Adagrad مجموع توان دوم گرادیان‌ها را در طول زمان نگه می‌دارد. پارامترهایی که گرادیان‌های بزرگی دارند، در نتیجه نرخ یادگیری‌شان به‌تدریج کاهش می‌یابد. اما ایراد Adagrad این است که نرخ یادگیری برای برخی پارامترها خیلی سریع به صفر نزدیک می‌شود و الگوریتم در به‌روزرسانی آن‌ها دچار مشکل می‌شود. به همین دلیل، الگوریتم RMSProp معرفی شد. این روش به‌جای جمع‌کردن ساده، روی توان دوم گرادیان‌ها یک میانگین نمایی (Exponential Moving Average) می‌گیرد تا یادگیری پایدارتر و کنترل‌شده‌تری داشته باشد.

الگوریتم Adam در واقع ترکیبی از RMSProp و Momentum است. از یک طرف مانند RMSProp روی توان دوم گرادیان‌ها میانگین نمایی می‌گیرد، و از طرف دیگر مانند الگوریتم Momentum، یک میانگین نمایی از خود گرادیان‌ها (نه فقط توان آن‌ها) نیز محاسبه می‌کند. علاوه بر این، برای جبران مقدار اولیه‌ی صفر در شروع آموزش، از Bias Correction استفاده می‌کند تا در مراحل ابتدایی، میانگین‌ها دچار خطا نباشند. البته این اصلاح بایاس با افزایش تعداد گام‌ها به‌مرور بی‌اثر می‌شود.

فرمول‌های به‌روزرسانی الگوریتم Adam به صورت زیر هستند:

برای مومنتوم داریم:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$$

برای adaptive learning rate داریم:

$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$$

برای Bias Correction داریم:

$$\widehat{m_t} = \frac{m_t}{1-\beta_1^t}$$

$$\widehat{v_t} = \frac{v_t}{1-\beta_2^t}$$

و در نهایت برای به‌روزرسانی پارامترها داریم:

$$\theta_t = \theta_{t-1} - \eta \cdot \frac{\widehat{m_t}}{\sqrt{\widehat{v_t}} + \epsilon}$$

کلاس Model

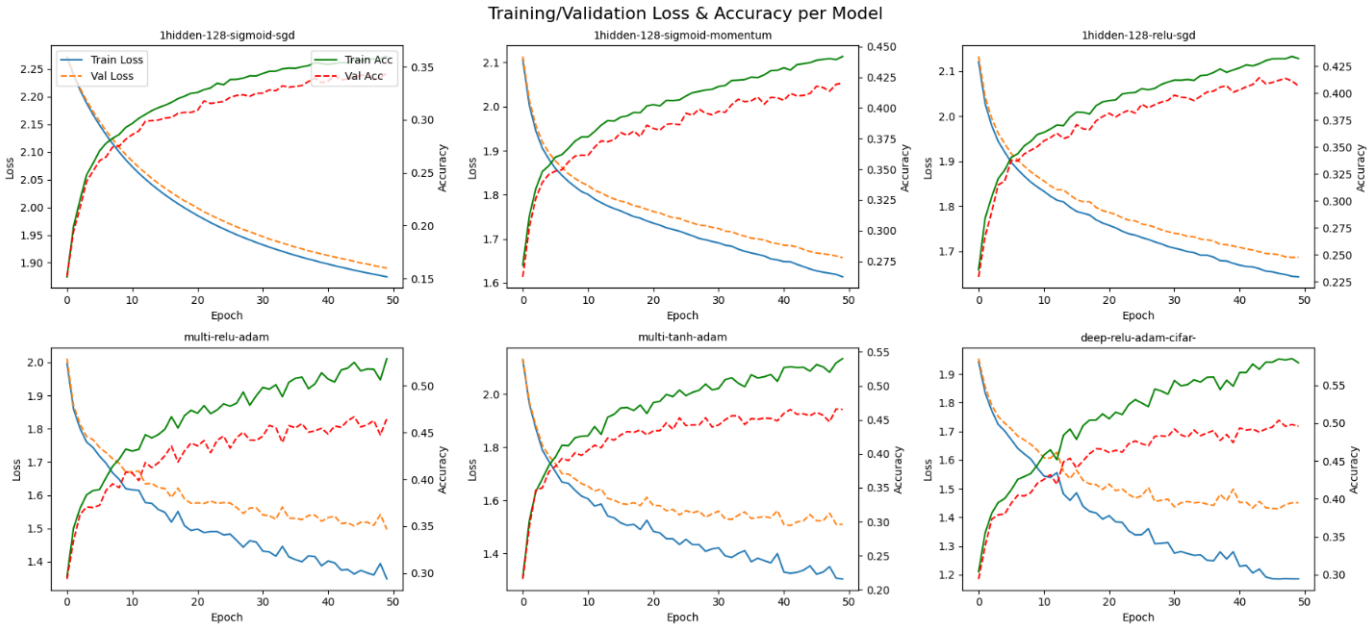
کلاس Model همه کلاس‌های قبلی را به هم متصل می‌کند. تمرکز اصلی آن روی محاسبه و به‌روزرسانی گرادیان‌هاست. در مرحله‌ی forward، داده‌ها از میان لایه‌ها عبور می‌کنند و خروجی نهایی با تابع خطا مقایسه می‌شود؛ سپس در مرحله‌ی backward،

گرادیان تابع خطا نسبت به خروجی محاسبه شده و به‌صورت معکوس از آخرین لایه تا لایه‌ی اول منتقل می‌شود. هر لایه گرادیان خود را محاسبه می‌کند و در نهایت با جمع‌آوری وزن‌ها و گرادیان‌های مربوطه، بهینه‌ساز این گرادیان‌ها را برای به‌روزرسانی پارامترهای مدل به‌کار می‌گیرد. این چرخه در هر epoch تکرار شده و روند یادگیری مدل از طریق به‌روزرسانی تدریجی وزن‌ها با استفاده از گرادیان‌ها انجام می‌شود. به علت زیاد بودن کد این بخش، از آوردن آن در این سند خودداری کردیم.

۴-۲- نمودارهای دقت و loss

در این بخش، چندین ساختار مختلف شبکه عصبی چندلایه (MLP) برای طبقه‌بندی تصاویر CIFAR-10 از ابتدا پیاده‌سازی و مقایسه شده‌اند. هر ساختار با تعداد لایه‌ها، نوع تابع فعال‌سازی) مانندSigmoid ، ReLU، (Tanhو بهینه‌سازSGD) ، Momentum، Adam متفاوت تعریف شده است تا تاثیر این عوامل بر عملکرد مدل بررسی شود.

برای هر پیکربندی، ابتدا مدل با لایه‌ها و بهینه‌ساز مشخص ساخته می‌شود و سپس با داده‌های آموزش به مدت ۱۰۰ دوره و batch سایز ۱۰۲۴ آموزش می‌بینند. پس از آموزش، مدل روی داده‌های تست ارزیابی شده و نتایج آن (مانند دقت و خطا) ذخیره می‌شود. در پایان، با رسم نمودار تاریخچه آموزش و اعتبارسنجی، روند تغییرات خطا و دقت برای همه مدل‌ها به صورت بصری نمایش داده می‌شود تا مقایسه ساختارهای مختلف ساده‌تر گردد.



عملکرد هر یک از مدل‌های شبکه عصبی چندلایه را روی داده‌های تست CIFAR-10 به طور کامل ارزیابی می‌کنیم. ابتدا برای هر مدل، معیارهایی مانند مقدار خطا(Loss) ، دقت(Accuracy) ، میانگین F1 (Macro-F1) و ماتریس آشفتگی محاسبه و چاپ می‌شود. سپس با استفاده از تابعclassification_report ، دقت، یادآوری و F1 هر کلاس به صورت جداگانه نمایش داده می‌شود. این گزارش‌ها به شما کمک می‌کند تا نقاط قوت و ضعف هر مدل را به طور دقیق بررسی و مقایسه کنید.

```
============  1hidden-128-sigmoid-sgd  ============
Loss        : 1.8096
Accuracy    : 0.3730
Macro-F1    : 0.3667
Confusion-matrix:
[[491  50  43  27  11  23  29  52 212  62]
 [ 62 410  21  41  17  50  65  34 126 174]
 [142  49 199  74 137  99 159  64  48  29]
 [ 57  78  85 218  45 233 114  78  38  54]
 [ 69  43 121  57 271  92 200  92  31  24]
 [ 48  49  91 132  61 360 112  80  50  17]
 [ 17  59  80  98  93 101 458  34  19  41]
 [ 63  55  72  56 110  77  70 356  43  98]
 [170  94  10  26   4  55   2  19 504 116]
 [ 65 171  13  36  12  20  43  45 132 463]]

Classification report:
              precision    recall  f1-score   support

           0     0.4147    0.4910    0.4496      1000
           1     0.3875    0.4100    0.3984      1000
           2     0.2707    0.1990    0.2294      1000
           3     0.2850    0.2180    0.2470      1000
           4     0.3561    0.2710    0.3078      1000
           5     0.3243    0.3600    0.3412      1000
           6     0.3658    0.4580    0.4067      1000
           7     0.4169    0.3560    0.3840      1000
           8     0.4190    0.5040    0.4576      1000
           9     0.4295    0.4630    0.4456      1000

    accuracy                         0.3730     10000
   macro avg     0.3669    0.3730    0.3667     10000
weighted avg     0.3669    0.3730    0.3667     10000


============  1hidden-128-sigmoid-momentum  ============
Loss        : 1.5283
Accuracy    : 0.4676
Macro-F1    : 0.4664
Confusion-matrix:
[[549  29  77  19  26  17  25  46 147  65]
 [ 49 526  22  32  30  33  21  45  75 167]
 [ 92  29 361  89 121  77 104  85  28  14]
 [ 27  26 102 318  68 185 104  65  36  69]
 [ 58  13 172  57 380  61 117  95  29  18]
 [ 31  17  96 190  88 356  73  86  31  32]
 [ 13  16 104 102 126  68 497  33  18  23]
 [ 46  26  60  62  92  78  31 525  12  68]
 [125  47  17  28  19  29   9  19 621  86]
 [ 63 136  12  34  20  27  39  55  71 543]]

Classification report:
              precision    recall  f1-score   support

           0     0.5214    0.5490    0.5348      1000
           1     0.6081    0.5260    0.5641      1000
           2     0.3529    0.3610    0.3569      1000
           3     0.3416    0.3180    0.3294      1000
```

```
           4     0.3918    0.3800    0.3858      1000
           5     0.3824    0.3560    0.3687      1000
           6     0.4873    0.4970    0.4921      1000
           7     0.4981    0.5250    0.5112      1000
           8     0.5815    0.6210    0.6006      1000
           9     0.5005    0.5430    0.5209      1000

    accuracy                         0.4676     10000
   macro avg     0.4665    0.4676    0.4664     10000
weighted avg     0.4665    0.4676    0.4664     10000


============  1hidden-128-relu-sgd  ============
Loss        : 1.5767
Accuracy    : 0.4492
Macro-F1    : 0.4469
Confusion-matrix:
[[431  35  65  38  32  24  24  35 247  69]
 [ 33 531  19  42  17  39  32  34  96 157]
 [ 81  30 294 108 135  97 136  52  47  20]
 [ 21  34  75 331  59 224 104  45  45  62]
 [ 49  23 144  60 370  82 128  80  44  20]
 [ 18  27  85 198  70 379  80  67  53  23]
 [  6  19  69 116 101  91 523  25  24  26]
 [ 26  37  54  85 102  88  42 452  33  81]
 [ 73  65  12  32  21  34   9  16 668  70]
 [ 43 169   8  40  18  27  39  40 103 513]]

Classification report:
              precision    recall  f1-score   support

           0     0.5519    0.4310    0.4840      1000
           1     0.5474    0.5310    0.5391      1000
           2     0.3564    0.2940    0.3222      1000
           3     0.3152    0.3310    0.3229      1000
           4     0.4000    0.3700    0.3844      1000
           5     0.3493    0.3790    0.3635      1000
           6     0.4682    0.5230    0.4941      1000
           7     0.5343    0.4520    0.4897      1000
           8     0.4912    0.6680    0.5661      1000
           9     0.4928    0.5130    0.5027      1000

    accuracy                         0.4492     10000
   macro avg     0.4507    0.4492    0.4469     10000
weighted avg     0.4507    0.4492    0.4469     10000


============  multi-relu-adam  ============
Loss        : 1.4541
Accuracy    : 0.4922
Macro-F1    : 0.4892
Confusion-matrix:
[[567  41  60  31  60   8  19  59 118  37]
 [ 50 651  18  21  24  12  16  36  59 113]
 [ 76  25 298  78 197  67 105 115  20  19]
 [ 30  17  80 312 109 166 136  95  23  32]
 [ 49  16  96  51 508  29 104 125  15   7]
 [ 20  14  94 182 112 374  73  95  21  15]
 [  6  15  52 100 177  41 541  41  18   9]
 [ 37  22  44  54 108  56  24 610  14  31]
 [152  77  20  30  37  29   9  17 589  40]
 [ 57 223  16  37  19  17  23  74  62 472]]

Classification report:
              precision    recall  f1-score   support

           0     0.5431    0.5670    0.5548      1000
           1     0.5913    0.6510    0.6197      1000
           2     0.3830    0.2980    0.3352      1000
```

```
           3     0.3482    0.3120    0.3291      1000
           4     0.3760    0.5080    0.4322      1000
           5     0.4681    0.3740    0.4158      1000
           6     0.5152    0.5410    0.5278      1000
           7     0.4815    0.6100    0.5382      1000
           8     0.6273    0.5890    0.6075      1000
           9     0.6090    0.4720    0.5318      1000

    accuracy                         0.4922     10000
   macro avg     0.4943    0.4922    0.4892     10000
weighted avg     0.4943    0.4922    0.4892     10000


============  multi-tanh-adam  ============
Loss        : 1.5094
Accuracy    : 0.4896
Macro-F1    : 0.4824
Confusion-matrix:
[[617  28  60  15  17  12  55  23 142  31]
 [ 64 566  21  25  13  12  37  19 106 137]
 [ 95  18 394  54  96  48 199  47  33  16]
 [ 45  30 107 293  48 129 238  37  40  33]
 [ 76  15 166  58 351  24 223  46  30  11]
 [ 38  18 111 214  65 278 174  50  33  19]
 [  8  13  69  44  60  22 746  13  14  11]
 [ 70  14  85  80  81  59  76 482  19  34]
 [119  58  20  22  12   7  29  12 682  39]
 [ 77 175  23  35  12  18  46  31  96 487]]

Classification report:
              precision    recall  f1-score   support

           0     0.5103    0.6170    0.5586      1000
           1     0.6053    0.5660    0.5850      1000
           2     0.3731    0.3940    0.3833      1000
           3     0.3488    0.2930    0.3185      1000
           4     0.4649    0.3510    0.4000      1000
           5     0.4565    0.2780    0.3456      1000
           6     0.4092    0.7460    0.5285      1000
           7     0.6342    0.4820    0.5477      1000
           8     0.5707    0.6820    0.6214      1000
           9     0.5954    0.4870    0.5358      1000

    accuracy                         0.4896     10000
   macro avg     0.4968    0.4896    0.4824     10000
weighted avg     0.4968    0.4896    0.4824     10000


============  deep-relu-adam-cifar-  ============
Loss        : 1.4571
Accuracy    : 0.5085
Macro-F1    : 0.5081
Confusion-matrix:
[[529  49  44  34  66  37  16  42 115  68]
 [ 39 623  17  29  14  27  15  29  41 166]
 [ 61  22 293 105 192  97 101  94  14  21]
 [ 16  25  59 353  83 245  97  74  12  36]
 [ 34  12  79  60 500  63 105 113  16  18]
 [ 13  11  47 212  93 449  72  74  12  17]
 [  4  18  47 109 137  77 568  17  11  12]
 [ 42  11  35  67  87  88  23 606   9  32]
 [ 93  87  15  42  39  28  12  22 589  73]
 [ 38 182  10  43  15  24  19  60  34 575]]

Classification report:
              precision    recall  f1-score   support

           0     0.6087    0.5290    0.5661      1000
           1     0.5990    0.6230    0.6108      1000
```

```
              2     0.4536    0.2930    0.3560      1000
              3     0.3349    0.3530    0.3437      1000
              4     0.4078    0.5000    0.4492      1000
              5     0.3956    0.4490    0.4206      1000
              6     0.5525    0.5680    0.5602      1000
              7     0.5358    0.6060    0.5687      1000
              8     0.6905    0.5890    0.6357      1000
              9     0.5648    0.5750    0.5699      1000

       accuracy                         0.5085     10000
      macro avg     0.5143    0.5085    0.5081     10000
   weighted avg     0.5143    0.5085    0.5081     10000
```

۵- پیاده‌سازی CNN با Tensorflow

مدل پایه CNN

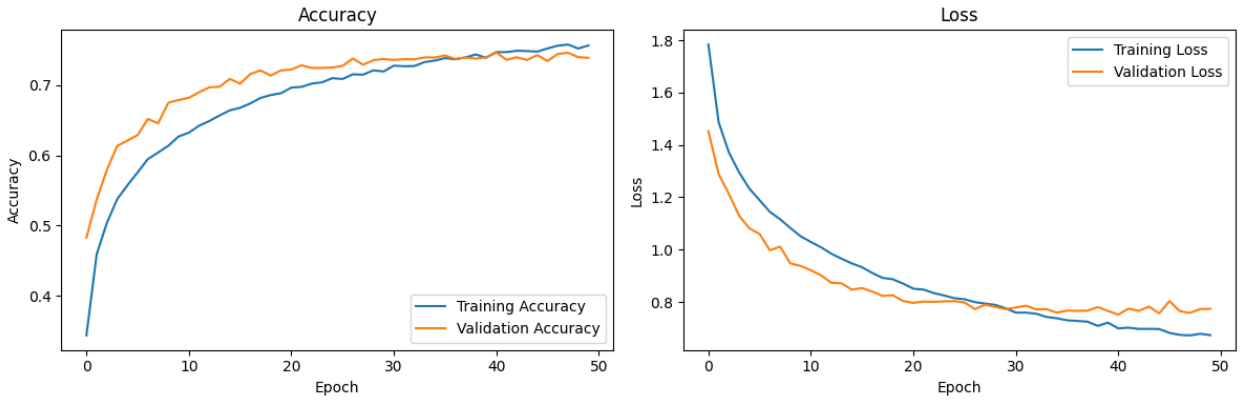
در این بخش، یک مدل شبکه عصبی کانولوشنی (CNN) ساده با استفاده از کتابخانه Keras برای دسته‌بندی تصاویر مجموعه داده CIFAR-10ساخته شده است. مدل شامل دو لایه کانولوشن است که به ترتیب ۳۲ و ۶۴ فیلتر با اندازه ۳×۳ دارند و هرکدام پس از خود یک لایه ماکس‌پولینگ و دراپ‌اوت برای کاهش بیش‌برازش قرار گرفته است. پس از تخت‌سازی(Flatten) ، یک لایه تمام‌متصل با ۱۲۸ نورون و تابع فعال‌سازی ReLU و یک لایه دراپ‌اوت دیگر قرار دارد. در نهایت، لایه خروجی با ۱۰ نورون و تابع فعال‌سازی Softmax برای دسته‌بندی ۱۰ کلاس مختلف استفاده شده است. مدل با استفاده از بهینه‌ساز Adam و تابع هزینه categorical_crossentropyکامپایل شده و آماده آموزش است. این ساختار باعث می‌شود مدل بتواند ویژگی‌های مکانی تصاویر را به خوبی استخراج و دسته‌بندی کند.

Layer (type)	Output Shape	Param #
conv2d_4 (Conv2D)	(None, 32, 32, 32)	896
max_pooling2d_4 (MaxPooling2D)	(None, 16, 16, 32)	0
dropout_4 (Dropout)	(None, 16, 16, 32)	0
conv2d_5 (Conv2D)	(None, 16, 16, 64)	18,496
max_pooling2d_5 (MaxPooling2D)	(None, 8, 8, 64)	0
dropout_5 (Dropout)	(None, 8, 8, 64)	0
flatten_2 (Flatten)	(None, 4096)	0
dense_4 (Dense)	(None, 128)	524,416
dropout_6 (Dropout)	(None, 128)	0
dense_5 (Dense)	(None, 10)	1,290

آموزش و ارزیابی

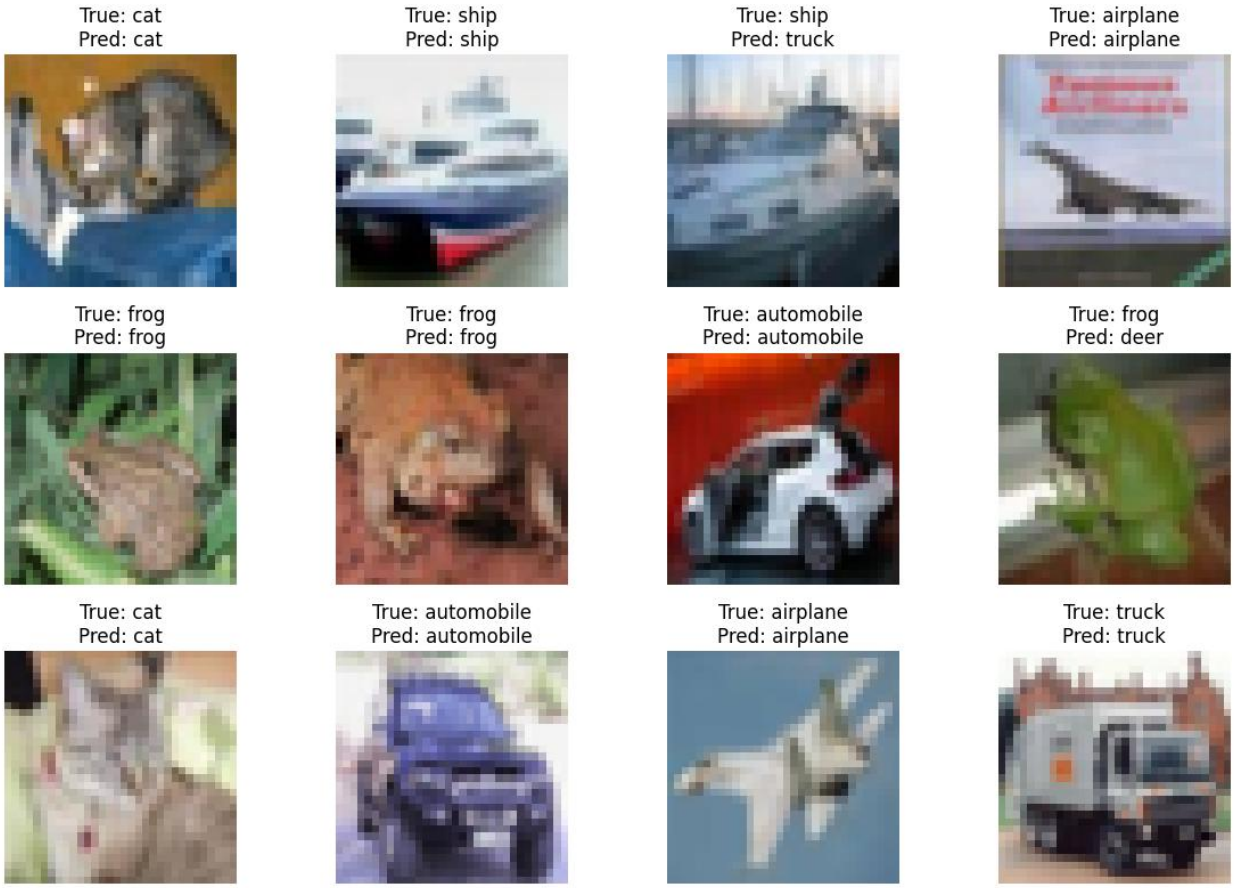
در این بخش، مدل شبکه عصبی کانولوشنی (CNN) آموزش داده می‌شود. برای جلوگیری از بیش‌برازش، از callback مربوط به EarlyStoppingاستفاده شده است تا اگر مقدار خطای اعتبارسنجی (val_loss) به مدت ۱۰ دوره متوالی بهبود نیابد، آموزش متوقف شود و بهترین وزن‌های مدل بازیابی گردد. مدل با استفاده از داده‌های آموزش و اعتبارسنجی (٪۲۰ داده‌ها برای اعتبارسنجی) و به مدت حداکثر ۵۰ دوره آموزش می‌بیند. پس از اتمام آموزش، روند دقت (accuracy) و خطا (loss) برای داده‌های آموزش و اعتبارسنجی در هر epoch رسم می‌شود تا بتوان عملکرد مدل و وجود احتمالی بیش‌برازش یا کم‌برازش را به صورت بصری بررسی کرد. این نمودارها به تحلیل بهتر فرآیند یادگیری مدل کمک می‌کنند.

```python
# Train the model
early_stopping = EarlyStopping(monitor='val_loss', patience=10, restore_best_weights=True)
# Train the model
history = model.fit(
    x_train, y_train,
    batch_size=64,
    epochs=50,
    validation_split=0.2,
    verbose=1,
    callbacks=[early_stopping] # Add callback here
)
```



در این بخش، عملکرد نهایی مدل شبکه عصبی کانولوشنی (CNN) آموزش‌دیده روی داده‌های تست ارزیابی می‌شود. ابتدا مدل با داده‌های تست ارزیابی شده و مقادیر دقت (accuracy) و خطا (loss) روی داده‌های دیده‌نشده چاپ می‌شود تا میزان موفقیت مدل سنجیده شود. سپس مدل روی کل داده‌های تست پیش‌بینی انجام می‌دهد و با استفاده از تابع argmax، کلاس پیش‌بینی‌شده برای هر تصویر استخراج می‌شود. برای بررسی کیفی عملکرد مدل، ۱۲ نمونه از تصاویر تست به همراه برچسب واقعی و برچسب پیش‌بینی‌شده نمایش داده می‌شوند. این کار به درک بهتر نقاط قوت و ضعف مدل در تشخیص کلاس‌های مختلف کمک می‌کند و می‌توان موارد درست یا اشتباه مدل را به صورت بصری مشاهده کرد.

```
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.7414 - loss: 0.7681
Test accuracy: 0.7443
Test loss: 0.7710
313/313 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step
```



مدل Regularized CNN

در این بخش، یک مدل شبکه عصبی کانولوشنی (CNN) با استفاده از تکنیک‌های منظم‌سازی (Regularization) ساخته شده است تا از بیش‌برازش مدل جلوگیری شود و عملکرد بهتری روی داده‌های تست داشته باشد. در این مدل، علاوه بر لایه‌های کانولوشن و تمام‌متصل، از چند روش منظم‌سازی استفاده شده است:

Regularization (l2) : با استفاده از پارامتر kernel_regularizer=l2(0.001) روی لایه‌های کانولوشن وDense ، جریمه‌ای به وزن‌ها اضافه می‌شود تا مدل از یادگیری وزن‌های بزرگ و بیش‌برازش جلوگیری کند.

Batch Normalization : بعد از هر لایه کانولوشن وDense ، لایه BatchNormalization قرار داده شده تا سرعت و پایداری آموزش افزایش یابد و مدل نسبت به تغییرات مقدار ورودی حساسیت کمتری داشته باشد.

Dropout : برای کاهش بیش‌برازش، بعد از هر بلاک کانولوشن و لایهDense ، لایه Dropout قرار داده شده است تا برخی نورون‌ها به صورت تصادفی غیرفعال شوند.

Activation : از تابع فعال‌سازی ReLU برای لایه‌های مخفی و Softmax برای لایه خروجی استفاده شده است.

در نهایت، مدل با بهینه‌ساز Adam و تابع هزینه categorical_crossentropy کامپایل شده و آماده آموزش است. این ساختار باعث افزایش تعمیم‌پذیری مدل و بهبود عملکرد روی داده‌های جدید می‌شود.

Layer (type)	Output Shape	Param #
conv2d_8 (Conv2D)	(None, 32, 32, 32)	896
batch_normalization_3 (BatchNormalization)	(None, 32, 32, 32)	128
activation_3 (Activation)	(None, 32, 32, 32)	0
max_pooling2d_8 (MaxPooling2D)	(None, 16, 16, 32)	0
dropout_10 (Dropout)	(None, 16, 16, 32)	0
conv2d_9 (Conv2D)	(None, 16, 16, 64)	18,496
batch_normalization_4 (BatchNormalization)	(None, 16, 16, 64)	256
activation_4 (Activation)	(None, 16, 16, 64)	0
max_pooling2d_9 (MaxPooling2D)	(None, 8, 8, 64)	0
dropout_11 (Dropout)	(None, 8, 8, 64)	0
flatten_4 (Flatten)	(None, 4096)	0
dense_8 (Dense)	(None, 128)	524,416
batch_normalization_5 (BatchNormalization)	(None, 128)	512
activation_5 (Activation)	(None, 128)	0
dropout_12 (Dropout)	(None, 128)	0
dense_9 (Dense)	(None, 10)	1,290

آموزش و ارزیابی

در این بخش، مدل Regularized CNN با استفاده از داده‌های آموزش و اعتبارسنجی آموزش داده می‌شود و برای جلوگیری از بیش‌برازش، دو تکنیک EarlyStopping و ModelCheckpoint به کار رفته است. EarlyStopping آموزش را زمانی متوقف می‌کند که مقدار val_loss به مدت ۱۰ دوره متوالی بهبود نیابد و بهترین وزن‌های مدل را بازیابی می‌کند تا از overfitting جلوگیری شود. ModelCheckpoint نیز بهترین مدل را بر اساس کمترین مقدار val_loss ذخیره می‌کند تا در پایان آموزش بتوان بهترین مدل را بارگذاری کرد. مدل با ۸۰٪ داده‌ها آموزش می‌بیند و ۲۰٪ برای اعتبارسنجی استفاده می‌شود و حداکثر ۵۰ دوره آموزش ادامه دارد. پس از اتمام آموزش، مدل روی داده‌های تست ارزیابی می‌شود و دقت نهایی آن نمایش داده می‌شود. این روش‌ها باعث افزایش تعمیم‌پذیری مدل و جلوگیری از بیش‌برازش می‌شوند.

```
Epoch 19: early stopping
Restoring model weights from the end of the best epoch: 9.
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.7544 - loss: 1.1741
Regularized CNN Test accuracy: 0.7522
```

پس از ارزیابی مدل Regularized CNN روی داده‌های تست، ماتریس آشفتگی (Confusion Matrix) محاسبه و رسم می‌شود. ابتدا مدل روی داده‌های تست پیش‌بینی انجام می‌دهد و کلاس پیش‌بینی‌شده برای هر نمونه با استفاده از تابع argmax استخراج می‌شود. سپس با مقایسه برچسب‌های واقعی و پیش‌بینی‌شده، ماتریس آشفتگی ساخته می‌شود که نشان می‌دهد مدل در تشخیص هر کلاس چه تعداد نمونه را به درستی یا اشتباه طبقه‌بندی کرده است. این ماتریس با استفاده از کتابخانه seaborn به صورت تصویری نمایش داده می‌شود تا بتوان عملکرد مدل را برای هر کلاس به طور جداگانه بررسی کرد و نقاط قوت و ضعف مدل در تشخیص کلاس‌های مختلف را بهتر تحلیل نمود.

مدل CNN عمیق‌تر

در این بخش، یک مدل شبکه عصبی کانولوشنی عمیق (Deep CNN) برای طبقه‌بندی تصاویر مجموعه داده CIFAR-10 ساخته شده است. این مدل از سه بلاک کانولوشنی متوالی تشکیل شده که هر بلاک شامل چندین لایه کانولوشن با تعداد فیلترهای افزایشی (۳۲، ۶۴ و ۱۲۸)، لایه BatchNormalization برای افزایش پایداری و سرعت آموزش، تابع فعال‌سازی ReLU و لایه Dropoutبرای کاهش بیش‌برازش است. پس از هر دو بلاک اول، لایه MaxPooling2D قرار گرفته تا ابعاد ویژگی‌ها کاهش یابد و ویژگی‌های مهم‌تر استخراج شوند. بعد از بلاک‌های کانولوشنی، ویژگی‌ها تخت (Flatten) شده و وارد لایه تمام‌متصل (Dense)با ۲۵۶ نورون می‌شوند که پس از آن نیزBatchNormalization ، ReLUو Dropout قرار دارد. در نهایت، لایه خروجی با ۱۰ نورون و تابع فعال‌سازی Softmax برای پیش‌بینی کلاس تصاویر استفاده شده است. این ساختار عمیق باعث می‌شود مدل بتواند ویژگی‌های پیچیده‌تر و سطوح بالاتری از تصاویر را یاد بگیرد و عملکرد بهتری در طبقه‌بندی داده‌ها داشته باشد. مدل با بهینه‌ساز Adam و تابع هزینه categorical_crossentropy کامپایل شده و آماده آموزش است.

Layer (type)	Output Shape	Param #
conv2d_10 (Conv2D)	(None, 32, 32, 32)	896
batch_normalization_6 (BatchNormalization)	(None, 32, 32, 32)	128
activation_6 (Activation)	(None, 32, 32, 32)	0
conv2d_11 (Conv2D)	(None, 32, 32, 32)	9,248
batch_normalization_7 (BatchNormalization)	(None, 32, 32, 32)	128
activation_7 (Activation)	(None, 32, 32, 32)	0
max_pooling2d_10 (MaxPooling2D)	(None, 16, 16, 32)	0
dropout_13 (Dropout)	(None, 16, 16, 32)	0
conv2d_12 (Conv2D)	(None, 16, 16, 64)	18,496
batch_normalization_8 (BatchNormalization)	(None, 16, 16, 64)	256
activation_8 (Activation)	(None, 16, 16, 64)	0
conv2d_13 (Conv2D)	(None, 16, 16, 64)	36,928
batch_normalization_9 (BatchNormalization)	(None, 16, 16, 64)	256
activation_9 (Activation)	(None, 16, 16, 64)	0
max_pooling2d_11 (MaxPooling2D)	(None, 8, 8, 64)	0
dropout_14 (Dropout)	(None, 8, 8, 64)	0
conv2d_14 (Conv2D)	(None, 8, 8, 128)	73,856
batch_normalization_10 (BatchNormalization)	(None, 8, 8, 128)	512

activation_10 (Activation)	(None, 8, 8, 128)	0
max_pooling2d_12 (MaxPooling2D)	(None, 4, 4, 128)	0
dropout_15 (Dropout)	(None, 4, 4, 128)	0
flatten_5 (Flatten)	(None, 2048)	0
dense_10 (Dense)	(None, 256)	524,544
batch_normalization_11 (BatchNormalization)	(None, 256)	1,024
activation_11 (Activation)	(None, 256)	0
dropout_16 (Dropout)	(None, 256)	0
dense_11 (Dense)	(None, 10)	2,570

آموزش و ارزیابی

در این بخش، مدل شبکه عصبی کانولوشنی عمیق (Deep CNN) آموزش داده می‌شود. مدل با استفاده از ۸۰٪ داده‌ها برای آموزش و ۲۰٪ برای اعتبارسنجی، به مدت ۳۰ دوره (epoch) و با اندازه دسته ۶۴ آموزش می‌بینند. پس از اتمام آموزش، مدل روی داده‌های تست ارزیابی می‌شود و دقت نهایی آن چاپ می‌گردد. این فرآیند به بررسی عملکرد مدل عمیق در طبقه‌بندی تصاویر مجموعه داده CIFAR-10 کمک می‌کند و نشان می‌دهد که افزایش عمق شبکه چگونه می‌تواند باعث بهبود استخراج ویژگی‌ها و افزایش دقت مدل شود.

```
625/625 ━━━━━━━━━━━━━━━━━━━━ 5s 8ms/step - accuracy: 0.8887 - loss: 0.3172 - val_accuracy: 0.8285 -
val_loss: 0.5378
Epoch 30/30
625/625 ━━━━━━━━━━━━━━━━━━━━ 10s 8ms/step - accuracy: 0.8840 - loss: 0.3272 - val_accuracy: 0.8444 -
val_loss: 0.4923
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.8368 - loss: 0.5084
Deep CNN Test accuracy: 0.8348
```

مقایسه مدل‌های توسعه داده‌شده

در این بخش، عملکرد سه مدل مختلف شبکه عصبی کانولوشنی Base CNN، Regularized CNN و Deep CNN روی داده‌های تست با یکدیگر مقایسه می‌شود. ابتدا دقت (Accuracy) و خطا (Loss) هر مدل چاپ می‌شود تا بتوان عملکرد عددی آن‌ها را مشاهده کرد. سپس با استفاده از نمودارهای میله‌ای، مقادیر دقت و خطا برای هر مدل به صورت بصری نمایش داده می‌شود. این مقایسه به شما کمک می‌کند تا تاثیر منظم‌سازی (Regularization) و افزایش عمق شبکه (Deep CNN) را بر بهبود دقت و کاهش خطا بررسی کنید و بهترین ساختار را برای مسئله طبقه‌بندی تصاویر انتخاب نمایید.

```
============================================================
MODEL COMPARISON RESULTS
============================================================
Base CNN        - Accuracy: 0.7443, Loss: 0.7710
Regularized CNN - Accuracy: 0.7522, Loss: 1.1774
Deep CNN        - Accuracy: 0.8348, Loss: 0.5138
```