

Modele1

November 15, 2025

```
[10]: import sys
import numpy as np
import matplotlib.pyplot as plt

np.random.seed(2)
```

```
[ ]:
```

```
[11]: import numpy as np

def extract_features(img):
    """
    img : (28,28,3)
    Retourne un vecteur de features ~ 3+3+18 dimensions.
    """

    features = []

    # Convert to float
    img = img.astype(np.float32)

    # Gray
    gray = img.mean(axis=2)

    # ----- RGB stats -----
    for c in range(3):
        features.append(img[:, :, c].mean())
        features.append(img[:, :, c].std())

    # ----- LBP -----
    def lbp(gray):
        H, W = gray.shape
        padded = np.pad(gray, ((1, 1), (1, 1)), mode="edge")

        lbp = np.zeros((H, W), dtype=np.uint8)

        offsets = [
            (-1, -1), (-1, 0), (-1, 1),
```

```

        ( 0, -1), (0, 1),
        ( 1, -1), ( 1, 0), (1, 1)
    ]

    for idx, (dy, dx) in enumerate(offsets):
        neigh = padded[1+dy:H+1+dy, 1+dx:W+1+dx]

        # cast pour éviter ton erreur
        bit = (neigh >= gray).astype(np.uint8)

        lbp |= (bit << idx)

    return lbp

lbp_img = lbp(gray)
hist_lbp = np.histogram(lbp_img, bins=16, range=(0, 256))[0]
features.extend(hist_lbp.tolist())

return np.array(features, dtype=np.float32)

```

```
[12]: class StandardScaler:
    def fit(self, X):
        self.mu = X.mean(axis=0)
        self.sigma = X.std(axis=0) + 1e-8
    def transform(self, X):
        return (X - self.mu) / self.sigma
    def fit_transform(self, X):
        self.fit(X)
        return self.transform(X)
```

```
[13]: class SoftmaxClassifier:
    def __init__(self, input_dim, num_classes, reg=0.0):
        self.W = 0.01 * np.random.randn(input_dim, num_classes).astype(np.
        float32)
        self.reg = reg # L2
        self.b = np.zeros(num_classes)

    def _softmax(self, scores):
        # scores: (N, K)
        scores = scores - scores.max(axis=1, keepdims=True) # stabilité num.
        exp_scores = np.exp(scores)
        return exp_scores / exp_scores.sum(axis=1, keepdims=True)
    def loss_and_grad(self, X, y, sample_weights=None):
        N = X.shape[0]

        scores = X @ self.W + self.b
```

```

probs = self._softmax(scores)

correct_logprobs = -np.log(probs[np.arange(N), y] + 1e-12)

if sample_weights is None:
    sample_weights = np.ones(N)

loss = np.sum(sample_weights * correct_logprobs) / N
loss += 0.5 * self.reg * np.sum(self.W * self.W)

dscores = probs
dscores[np.arange(N), y] -= 1

dscores *= sample_weights[:, None]
dscores /= N

dW = X.T @ dscores + self.reg * self.W
db = dscores.sum(axis=0)

return loss, dW, db

def fit(self, X, y, lr=1e-4, n_steps=1000, sample_weights=None, ↴
verbose=True):
    losses = []
    for step in range(n_steps):
        loss, dW, db = self.loss_and_grad(X, y, sample_weights)
        self.W -= lr * dW
        self.b -= lr * db
        losses.append(loss)

        if verbose and step % 100 == 0:
            print(f"Step {step}, loss = {loss:.4f}")

    return losses

def predict_proba(self, X):
    scores = X @ self.W
    probs = self._softmax(scores)
    return probs

def predict(self, X):
    probs = self.predict_proba(X)
    return probs.argmax(axis=1)

```

```
[14]: def accuracy(y_true, y_pred):
    y_true = np.asarray(y_true)
    y_pred = np.asarray(y_pred)
    return np.mean(y_true == y_pred)
```

```
[15]: def confusion_matrix_np(y_true, y_pred, num_classes=None):
    y_true = np.asarray(y_true).astype(int)
    y_pred = np.asarray(y_pred).astype(int)

    if num_classes is None:
        num_classes = max(y_true.max(), y_pred.max()) + 1

    cm = np.zeros((num_classes, num_classes), dtype=int)
    for t, p in zip(y_true, y_pred):
        cm[t, p] += 1
    return cm
```

```
[16]: def balanced_accuracy(y_true, y_pred):
    cm = confusion_matrix_np(y_true, y_pred)
    TP = np.diag(cm)
    real_pos = cm.sum(axis=1)
    recall = np.where(real_pos > 0, TP / real_pos, 0.0)
    return recall.mean()
```

```
[17]: def recall_per_class(cm):
    """
    cm : matrice de confusion (numpy array KxK)
    retourne un vecteur de recall par classe
    """
    TP = np.diag(cm)
    real_pos = cm.sum(axis=1) # total de vrais échantillons par classe

    # recall par classe (évite division par zéro)
    recall = np.where(real_pos > 0, TP / real_pos, 0.0)
    return recall
```

```
[ ]: import pickle
path_to_data = 'ift-3395-6390-kaggle-2-competition-fall-2025/train_data.pkl'

# --- Load training data ---
with open(path_to_data, "rb") as f:
    train_data = pickle.load(f)

X_imgs = train_data["images"]
y = train_data["labels"].reshape(-1)

# --- Feature extraction ---
```

```

X = np.array([extract_features(img) for img in X_imgs], dtype=np.float32)

# --- Normalize ---
scaler = StandardScaler()
X = scaler.fit_transform(X)

# --- Split ---
n_train = int(0.8 * len(X))
X_train, X_test = X[:n_train], X[n_train:]
y_train, y_test = y[:n_train], y[n_train:]

# --- Class weights ---
class_counts = np.bincount(y_train)
class_weights = 1.0 / class_counts
class_weights /= class_weights.sum()
sample_weights = class_weights[y_train]

# --- Train model ---
num_classes = len(np.unique(y))
model = SoftmaxClassifier(input_dim=X.shape[1], num_classes=num_classes, ↴
    reg=1e-3)

model.fit(X_train, y_train, lr=1e-3, n_steps=5000, ↴
    sample_weights=sample_weights)

# --- Evaluate ---
y_pred = model.predict(X_test)
acc = (y_pred == y_test).mean()
print("Test accuracy =", acc)

# --- Save model ---
pickle.dump((model, scaler), open("model_softmax.pkl", "wb"))

#print("Train accuracy:", acc)
print("Test accuracy:", acc)
#cm = confusion_matrix_np(y_true, y_pred)
#acc = accuracy(y_true, y_pred)
#bal_acc = balanced_accuracy(y_true, y_pred)

"""

rec = recall_per_class(cm)
print("Accuracy      :", acc)
print("Balanced acc   :", bal_acc)
print("Confusion matrix:\n", cm)
print("Recall par classe :", rec)
print("Recall moyen (macro):", rec.mean())
"""

```

```
Step 0, loss = 0.2204
Step 100, loss = 0.2202
Step 200, loss = 0.2200
Step 300, loss = 0.2197
Step 400, loss = 0.2195
Step 500, loss = 0.2193
Step 600, loss = 0.2190
Step 700, loss = 0.2188
Step 800, loss = 0.2186
Step 900, loss = 0.2184
Step 1000, loss = 0.2182
Step 1100, loss = 0.2180
Step 1200, loss = 0.2178
Step 1300, loss = 0.2176
Step 1400, loss = 0.2174
Step 1500, loss = 0.2172
Step 1600, loss = 0.2170
Step 1700, loss = 0.2168
Step 1800, loss = 0.2167
Step 1900, loss = 0.2165
Step 2000, loss = 0.2163
Step 2100, loss = 0.2161
Step 2200, loss = 0.2160
Step 2300, loss = 0.2158
Step 2400, loss = 0.2156
Step 2500, loss = 0.2155
Step 2600, loss = 0.2153
Step 2700, loss = 0.2152
Step 2800, loss = 0.2150
Step 2900, loss = 0.2149
Step 3000, loss = 0.2147
Step 3100, loss = 0.2146
Step 3200, loss = 0.2144
Step 3300, loss = 0.2143
Step 3400, loss = 0.2142
Step 3500, loss = 0.2140
Step 3600, loss = 0.2139
Step 3700, loss = 0.2138
Step 3800, loss = 0.2136
Step 3900, loss = 0.2135
Step 4000, loss = 0.2134
Step 4100, loss = 0.2132
Step 4200, loss = 0.2131
Step 4300, loss = 0.2130
Step 4400, loss = 0.2129
Step 4500, loss = 0.2128
Step 4600, loss = 0.2126
Step 4700, loss = 0.2125
```

```
Step 4800, loss = 0.2124
Step 4900, loss = 0.2123
Test accuracy = 0.2962962962962963
```

```
[ ]: '\nrec = recall_per_class(cm)\nprint("Accuracy      :", acc)\nprint("Balanced\nacc     :", bal_acc)\nprint("Confusion matrix:\n", cm)\nprint("Recall par classe\n:", rec)\nprint("Recall moyen (macro):", rec.mean())'
```

```
[ ]: with open("ift-3395-6390-kaggle-2-competition-fall-2025/test_data.pkl", "rb") as f:
    test_data = pickle.load(f)

X_test_imgs = test_data["images"]

X_test_feats = np.array([extract_features(img) for img in X_test_imgs], dtype=np.float32)

# normaliser avec les stats du train
X_test_norm =

# prédictions
y_pred = model.predict(X_test_norm).astype(int)
```