Modele12

November 17, 2025

[1713]:
```
import sys
import numpy as np
import matplotlib.pyplot as plt

np.random.seed(0)
```

[1714]:
```
def extract_features_full(img):
    """
    Version COMPLÈTE: 337 features optimisées
    Pour rétinopathie diabétique sur images 28x28x3
    """
    features = []

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

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


    for c in range(3):
        channel = img[:, :, c]
        # Percentiles groupés pour performance
        percs = np.percentile(channel, [10, 25, 50, 75, 90])
        features.extend([
            channel.mean(),
            channel.std(),
            percs[0],  # p10
            percs[1],  # p25
            percs[2],  # p50 (médiane)
            percs[3],  # p75
            percs[4],  # p90
        ])

    gray_percs = np.percentile(gray, [10, 25, 50, 75, 90])
    features.extend([
        gray.mean(),
        gray.std(),
```

```
    gray.min(),
    gray.max(),
    gray_percs[0],  # p10
    gray_percs[1],  # p25
    gray_percs[2],
    gray_percs[3],  # p75
    gray_percs[4],  # p90
])

H, W = gray.shape
h2, w2 = H//2, W//2

quadrants = [
    gray[:h2, :w2],    # Top-left
    gray[:h2, w2:],    # Top-right
    gray[h2:, :w2],    # Bottom-left
    gray[h2:, w2:]     # Bottom-right
]

for quad in quadrants:
    features.extend([
        quad.mean(),
        quad.std(),
        quad.min(),
        quad.max()
    ])

features.append(gray.max() - gray.min())

features.append(gray.std() / (gray.mean() + 1e-8))

hist_entr, _ = np.histogram(gray.ravel(), bins=64, range=(0, 256))
hist_entr = hist_entr / (hist_entr.sum() + 1e-8)
entropy = -np.sum(hist_entr * np.log2(hist_entr + 1e-8))
features.append(entropy)

skew = ((gray - gray.mean())**3).mean() / (gray.std()**3 + 1e-8)
features.append(skew)


means = img.mean(axis=(0, 1))
stds = img.std(axis=(0, 1))

features.extend([
    means[0] / (means[1] + 1e-8),  # R/G ratio
    means[0] / (means[2] + 1e-8),  # R/B ratio
    means[1] / (means[2] + 1e-8),  # G/B ratio
```

```
        stds[0] / (stds[1] + 1e-8),     # R_std/G_std
        stds[0] / (stds[2] + 1e-8),     # R_std/B_std
        stds[1] / (stds[2] + 1e-8),     # G_std/B_std
    ])

    H, W = gray.shape
    padded = np.pad(gray, 1, mode="edge")
    lbp_img = 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]
        lbp_img |= ((neigh >= gray).astype(np.uint8) << idx)

    hist_lbp, _ = np.histogram(lbp_img, bins=256, range=(0, 256))
    features.extend(hist_lbp.tolist())

    hist_intensity, _ = np.histogram(gray, bins=16, range=(0, 256))
    features.extend(hist_intensity.tolist())

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

[1715]:
```
import numpy as np

def extract_features(img):
    """
    Version légère: ~100 features au lieu de 337
    Garde les plus importantes
    """
    features = []
    img = img.astype(np.float32)
    gray = img.mean(axis=2)

    # 1. RGB stats (18)
    for c in range(3):
        channel = img[:, :, c]
        features.append(channel.mean())
        features.append(channel.std())
        features.append(np.percentile(channel, 10))
        features.append(np.percentile(channel, 50))
        features.append(np.percentile(channel, 90))
        features.append(channel.max())

    # 2. Grayscale (6)
    features.append(gray.mean())
    features.append(gray.std())
```

```
    features.append(np.percentile(gray, 25))
    features.append(np.percentile(gray, 75))
    features.append(gray.min())
    features.append(gray.max())

    # 3. Edges (3)
    gx = np.abs(gray[:, 1:] - gray[:, :-1]).mean()
    gy = np.abs(gray[1:, :] - gray[:-1, :]).mean()
    features.extend([gx, gy, np.sqrt(gx**2 + gy**2)])

    # 4. Quadrants (8)
    H, W = gray.shape
    h2, w2 = H//2, W//2
    for quad in [gray[:h2, :w2], gray[:h2, w2:], gray[h2:, :w2], gray[h2:, w2:
↪]]:
        features.append(quad.mean())
        features.append(quad.std())

    # 5. Color ratios (3)
    R, G, B = img[:,:,0], img[:,:,1], img[:,:,2]
    features.append(R.mean() / (G.mean() + 1e-8))
    features.append(R.mean() / (B.mean() + 1e-8))
    features.append(G.mean() / (B.mean() + 1e-8))

    # 6. LBP (64 bins au lieu de 256)
    def lbp(gray):
        H, W = gray.shape
        padded = np.pad(gray, ((1, 1), (1, 1)), mode="edge")
        lbp_img = 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]
            bit = (neigh >= gray).astype(np.uint8)
            lbp_img |= (bit << idx)
        return lbp_img

    lbp_img = lbp(gray)
    hist_lbp = np.histogram(lbp_img, bins=64, range=(0, 256))[0]
    features.extend(hist_lbp.tolist())


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

[1716]:
```
import numpy as np
import itertools

def polynomial_expansion_degree3(X):
```

```
    N, d = X.shape
    features = []

    # Degree 1
    features.append(X)

    # Degree 2
    for i, j in itertools.combinations_with_replacement(range(d), 2):
        features.append((X[:, i] * X[:, j]).reshape(N, 1))

    # Degree 3
    for i, j, k in itertools.combinations_with_replacement(range(d), 3):
        features.append((X[:, i] * X[:, j] * X[:, k]).reshape(N, 1))

    return np.hstack(features)
```

[1717]:

```
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)
```

[1718]:

```
class SoftmaxClassifier:
    def __init__(self, input_dim, num_classes, reg=0.0, seed=None):

        if seed is not None:
            np.random.seed(seed)

        self.W = 0.01 * np.random.randn(input_dim, num_classes).astype(np.
↪float32)
        self.reg = reg  # L2
        self.b = np.zeros(num_classes)
        self.seed = seed

    def _softmax(self, scores):
        # scores: (N, K)
        scores = scores - scores.max(axis=1, keepdims=True)
        exp_scores = np.exp(scores)
        return exp_scores / exp_scores.sum(axis=1, keepdims=True)

    def loss(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**2)

        return loss, probs

    def grad(self, X, y, probs, sample_weights=None):
        N = X.shape[0]

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

        dscores = probs.copy()
        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 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, probs = self.loss(X, y, sample_weights)
            dW, db = self.grad(X, y, probs, 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
```

```python
    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)
```

[1719]:
```python
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)
```

[1720]:
```python
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
```

[1721]:
```python
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()
```

[1722]:
```python
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
```

[1723]:
```
import numpy as np

def random_flip(img):
    return np.fliplr(img)

def random_brightness(img):
    factor = 0.5 + np.random.rand()  # between 0.5 and 1.5
    return np.clip(img * factor, 0, 255).astype(np.uint8)

def random_noise(img):
    noise = np.random.normal(0, 10, img.shape)
    return np.clip(img + noise, 0, 255).astype(np.uint8)

def random_shift(img, max_shift=2):
    """Shift image by a few pixels using NumPy roll."""
    dx = np.random.randint(-max_shift, max_shift+1)
    dy = np.random.randint(-max_shift, max_shift+1)
    shifted = np.roll(img, dx, axis=0)
    shifted = np.roll(shifted, dy, axis=1)
    return shifted

def augment_image(img):
    """Apply one random augmentation with pure NumPy."""
    ops = [random_flip, random_brightness, random_noise, random_shift]
    op = np.random.choice(ops)
    return op(img)
```

[1724]:
```
import numpy as np

def rbf_kernel(x, X, sigma=1.0):
    # ||x - X||^2 = sum( (x_i - X_i)^2 )
    dists = np.sum((X - x)**2, axis=1)
    return np.exp(-dists / (2 * sigma**2))



class KernelPerceptron:
    def __init__(self, kernel_fn, n_classes, sigma=1.0):
        self.kernel_fn = kernel_fn
        self.n_classes = n_classes
        self.sigma = sigma


    def fit(self, X, y, max_epochs=10):
        """
        Entraînement multiclasse One-vs-Rest
        """
        N = X.shape[0]
```

```
        self.X = X
        self.classes = np.unique(y)
        self.train_y = y
        #np.random.seed(0)

        # On crée   pour chaque classe
        self.alpha = np.zeros((self.n_classes, N))

        # Precompute Gram matrix
        print("Computing Gram matrix...")
        self.K = np.zeros((N, N))
        for i in range(N):
            self.K[i] = self.kernel_fn(X[i], X, self.sigma)

        for c in self.classes:
            print(f"Training classifier for class {c}...")
            y_bin = np.where(y == c, 1, -1)

            count = 0
            i = 0
            n_iter = 0
            n_iter_max = max_epochs * N

            while count < N and n_iter < n_iter_max:
                pred = np.sign(np.sum(self.alpha[c] * y_bin * self.K[i]))

                if pred != y_bin[i]:
                    self.alpha[c][i] += 1
                    count = 0
                else:
                    count += 1

                i = (i + 1) % N
                n_iter += 1

        print("Training done.")

    def predict(self, X_test):
        """
        Retourne la classe prédite pour chaque x_test
        """
        outputs = []
        for x in X_test:
            # compute kernel K(x, X_train)
            k = self.kernel_fn(x, self.X, self.sigma)

            # score pour chaque classe
```

```
            scores = []
            for c in self.classes:
                y_bin = np.where(self.train_y == c, 1, -1)
                score = np.sum(self.alpha[c] * y_bin * k)
                scores.append(score)

            # classe la plus probable
            outputs.append(np.argmax(scores))

        return np.array(outputs)
```

[1725]:
```
def rbf_kernel_svm(X,Y, sigma):
    dists = np.sum(X**2, axis=1)[:,None] + np.sum(Y**2, axis=1)[None, :] - 2 *␣
↪X @ Y.T
    return np.exp(-dists / (2*sigma**2))

class SVM:
    def __init__(self, sigma=1.0, max_iter=5, sample_weights=None):
        self.sigma = sigma
        self.max_iter = max_iter
        self.sample_weights = sample_weights

    def fit(self, X, y):
        self.X = X
        self.y = y
        self.classes = np.unique(y)
        self.K = rbf_kernel_svm(X,X,self.sigma)

        N = X.shape[0]

        self.alpha = {}

        for c in self.classes:
            print(f"train class {c} vs rest")
            y_bin = np.where(y == c, 1, -1)
            alpha_c = np.zeros(N)

            for it in range(self.max_iter):
                for i in range(N):
                    f = np.sum(alpha_c * y_bin * self.K[:, i])
                    if y_bin[i] * f <= 0:
                        alpha_c[i] += self.sample_weights[i]

            self.alpha[c] = alpha_c

    def decision_function(self, X_test):
        K_test = rbf_kernel_svm(X_test, self.X, self.sigma)
```

```python
        scores = []
        for c in self.classes:
            y_bin = np.where(self.y == c,1,-1)
            s = K_test @ (self.alpha[c]*y_bin)
            scores.append(s)

        return np.vstack(scores).T

    def predict(self, X_test):
        scores = self.decision_function(X_test)
        return self.classes[np.argmax(scores, axis=1)]
```

[1726]:

```python
def PCA_(X, k=30):

    X_centered = X - X.mean(axis=0)
    C = np.cov(X_centered, rowvar=False)
    eigvals, eigvecs = np.linalg.eigh(C)
    idx = np.argsort(eigvals)[::-1]
    eigvecs = eigvecs[idx][:k]   # k = 30 par ex.
    X_reduced = X_centered @ eigvecs.T
    return X_reduced
```

[1727]:

```python
import tensorflow as tf
print(tf.__version__)
```

```
2.20.0
```

[1728]:

```python
import pickle
import numpy as np
#from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
import tensorflow as tf

# --- Load training data ---
path_to_data = 'ift-3395-6390-kaggle-2-competition-fall-2025/train_data.pkl'
with open(path_to_data, "rb") as f:
    train_data = pickle.load(f)

X_imgs = train_data["images"].astype(np.float32)
y = train_data["labels"].reshape(-1)
```

[1729]:

```python
def fft2_image(img):
    F = np.fft.fft2(img.mean(axis=2))       # Passe en grayscale
    F = np.fft.fftshift(F)
    mag = np.log1p(np.abs(F))
    return mag
```

[1730]:
```
def radial_profile(img):
    """ img: 2D array """
    h, w = img.shape
    cy, cx = h//2, w//2

    y, x = np.ogrid[:h, :w]
    r = np.sqrt((x-cx)**2 + (y-cy)**2).astype(int)

    radial_mean = np.bincount(r.ravel(), img.ravel()) / np.bincount(r.ravel())
    return radial_mean
```

[1731]:
```
def extract_simple_stats(img):
    gray = img.mean(axis=2)
    return np.array([gray.mean(), gray.std(), gray.min(), gray.max()], dtype=np.
↪float32)
```

[1732]:
```
def simple_augment(images):
    # flip horizontal + ajout bruit très léger
    flips = images[:, :, ::-1, :]   # inversion horizontale
    noise = images + 0.01*np.random.randn(*images.shape)
    return np.concatenate([images, flips, noise], axis=0)

#X_imgs = simple_augment(X_imgs)
#y = np.concatenate([y, y, y])      # labels doublés aussi
```

[1733]:
```
def fft_features_batch(images):
    # Convert to grayscale BEFORE FFT
    gray = images.mean(axis=3)    # (N, 28, 28)

    F = np.fft.fft2(gray, axes=(1,2))
    F_shift = np.fft.fftshift(F, axes=(1,2))
    mag = np.abs(F_shift)         # (N, 28, 28)

    return mag


fft_mag = fft_features_batch(X_imgs)
X_fft = np.array([radial_profile(img) for img in fft_mag], dtype=np.float32)
X_stats = np.array([extract_simple_stats(img) for img in X_imgs], dtype=np.
↪float32)
X = np.hstack([X_fft, X_stats])
```

[1734]:
```
scaler = StandardScaler()
X = scaler.fit_transform(X)
```

[1735]:
```
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, stratify=y, random_state=0
```

```
)
```

[1736]:
```
class_counts = np.bincount(y_train)
class_weights = (1.0 / class_counts)
class_weights /= class_weights.sum()
sample_weights = class_weights[y_train]
```

[1737]:
```
"""
model = SVM(sigma=, max_iter=15, sample_weights=sample_weights)
model.fit(X_train, y_train)
"""
```

[1737]:
```
'\nmodel = SVM(sigma=, max_iter=15,
sample_weights=sample_weights)\nmodel.fit(X_train, y_train)\n'
```

[1738]:
```
model = KernelPerceptron(kernel_fn=rbf_kernel, n_classes=5, sigma=5)
model.fit(X_train, y_train, max_epochs=5)
```

```
Computing Gram matrix…
Training classifier for class 0…
Training classifier for class 1…
Training classifier for class 2…
Training classifier for class 3…
Training classifier for class 4…
Training done.
Training done.
```

[1739]:
```
"""
model = SoftmaxClassifier(input_dim=X.shape[1], num_classes=num_classes, reg=0.
↪05, seed=0)
model.fit(X_train, y_train, lr=2, n_steps=5000, sample_weights=sample_weights)
"""
```

[1739]:
```
'\nmodel = SoftmaxClassifier(input_dim=X.shape[1], num_classes=num_classes,
reg=0.05, seed=0)\nmodel.fit(X_train, y_train, lr=2, n_steps=5000,
sample_weights=sample_weights)\n'
```

[1740]:
```
y_pred = model.predict(X_test)
acc = (y_pred == y_test).mean()
print("Test accuracy =", acc)

cm = confusion_matrix_np(y_test, y_pred)
bal_acc = balanced_accuracy(y_test, y_pred)
rec = recall_per_class(cm)

print("Balanced acc :", bal_acc)
print("Recall par classe :", rec)
print("Recall moyen :", rec.mean())
```

```
Test accuracy = 0.4166666666666667
Balanced acc : 0.3048129307621387
Recall par classe : [0.68041237 0.23076923 0.04878049 0.33333333 0.23076923]
Recall moyen : 0.3048129307621387
```

[1741]:
```python
pickle.dump((model, scaler), open("model_fft.pkl", "wb"))
```

[1742]:
```python
import pickle
import numpy as np
import pandas as pd

# --------------------------
# 1. Charger le modèle entraîné
# --------------------------
model, scaler = pickle.load(open("model_softmax.pkl", "rb"))

# --------------------------
# 2. Charger le test_data.pkl
# --------------------------
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"]


# Apply to test set
X_test_feats = np.array([extract_features(img) for img in X_test_imgs],␣
↪dtype=np.float32)

# --------------------------
# 4. Normaliser avec les stats du train
# --------------------------
X_test_norm = scaler.transform(X_test_feats)

# --------------------------
# 5. Prédire
# --------------------------
y_pred = model.predict(X_test_norm).astype(int)

# --------------------------
# 6. Générer le CSV Kaggle
# --------------------------
df = pd.DataFrame({
    "ID": np.arange(1, len(y_pred)+1),
    "Label": y_pred
})
```

```python
df.to_csv("ift3395_YamirPoldoSilvaV9.csv", index=False)

print("Fichier 'submission.csv' généré !")
```

```
Fichier 'submission.csv' généré !
```