```
# app_energy_prediction.py

import streamlit as st

import pandas as pd

import numpy as np

import plotly.express as px

import plotly.graph_objects as go

import matplotlib.pyplot as plt

from sklearn.preprocessing import MinMaxScaler

from sklearn.metrics import mean_absolute_percentage_error,
mean_squared_error

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import LSTM, Dense, Dropout

from tensorflow.keras.optimizers import Adam

from tensorflow.keras.callbacks import EarlyStopping, ModelCheckpoint,
ReduceLROnPlateau

import tensorflow as tf

import os


# --- PAGE CONFIGURATION ---

st.set_page_config(page_title="Energy Prediction Dashboard ⚡", layout="wide")

st.markdown(

    """

    <style>

    .stApp { background-color: #0f1a2b; color: #ffffff; }

    h1, h2, h3, h4, h5, h6 {color: #00d4ff;}
    .stButton>button {background-color:#0088cc; color:white;}
    .metric-container {background-color: #1e2a3b; padding: 10px; border-radius: 
5px;}
    </style>
    """, unsafe_allow_html=True
)

st.title("⚡ Deep LSTM Energy Prediction Dashboard")

# Custom RMSE function (identique à votre code)
def root_mean_squared_error(y_true, y_pred):
    return tf.sqrt(tf.reduce_mean(tf.square(y_pred - y_true)))

# --- Upload CSV ---
st.sidebar.header("📂 Upload CSV")
uploaded_file = st.sidebar.file_uploader("Choisir votre fichier 
historic_demand.csv", type="csv")

@st.cache_data
def load_data(file):
    df = pd.read_csv(file)

    # Vérifier les colonnes disponibles
    st.sidebar.info(f"Colonnes détectées: {list(df.columns)}")
    # Colonnes essentielles pour ce dataset
    required_cols = ["settlement_date", "england_wales_demand"]
    if not all(col in df.columns for col in required_cols):
        st.error(f"❌ Votre dataset doit contenir: {required_cols}")
        st.stop()

    # Nettoyer et préparer les données
    df = df.copy()

    # Convertir la date
    df["date_time"] = pd.to_datetime(df["settlement_date"], errors='coerce')
    df.dropna(subset=["date_time"], inplace=True)

    # Créer des features temporelles
    df["hour"] = (df["settlement_period"] - 1) // 2
    df["minute"] = ((df["settlement_period"] - 1) % 2) * 30
    df["datetime"] = df["date_time"] + pd.to_timedelta(df["hour"], unit='h') +
pd.to_timedelta(df["minute"], unit='m')

    # Features supplémentaires
    df["month"] = df["datetime"].dt.month
    df["day_of_week"] = df["datetime"].dt.dayofweek
    df["is_weekend"] = df["day_of_week"].isin([5, 6]).astype(int)

    # Renommer la colonne de demande
    df["energy_demand"] = df["england_wales_demand"]

    # Gérer les valeurs manquantes
    numeric_cols = df.select_dtypes(include=[np.number]).columns
    df[numeric_cols] = df[numeric_cols].fillna(method='ffill').fillna(method='bfill')

    df.set_index("datetime", inplace=True)
    return df

# --- DATA LOADING ---
if uploaded_file:
    if "data" not in st.session_state:
        with st.spinner("Chargement des données énergétiques..."):
            st.session_state["data"] = load_data(uploaded_file)

    data = st.session_state["data"]
    st.success(f"✅ Dataset chargé ! ({len(data)} enregistrements,
{data.index.min().strftime('%Y-%m-%d')} to {data.index.max().strftime('%Y-%m-
%d')})")

    # Aperçu des données
    st.subheader("📊 Aperçu des données")
    col1, col2, col3 = st.columns(3)
    with col1:
        st.metric("Demande moyenne", f"{data['energy_demand'].mean():.0f} MW")
    with col2:
        st.metric("Demande max", f"{data['energy_demand'].max():.0f} MW")
    with col3:
        st.metric("Demande min", f"{data['energy_demand'].min():.0f} MW")

    # --- Sidebar Configuration ---
    st.sidebar.header("⚙ Configuration")

    # Filtre temporel
    min_date, max_date = data.index.min(), data.index.max()
    date_range = st.sidebar.date_input(
        "Période à analyser",
        [min_date.date(), max_date.date()],
        min_value=min_date.date(),
        max_value=max_date.date()
    )

    if len(date_range) == 2:
        start_date, end_date = pd.to_datetime(date_range[0]),
pd.to_datetime(date_range[1])
        filtered_data = data[(data.index >= start_date) & (data.index <= end_date)]
    else:
        filtered_data = data.copy()

    # Filtre par jour de semaine
    days_map = 
{0:"Lundi",1:"Mardi",2:"Mercredi",3:"Jeudi",4:"Vendredi",5:"Samedi",6:"Dimanch
e"}
    selected_days = st.sidebar.multiselect("Jour(s) à analyser", 
list(days_map.values()), default=list(days_map.values()))
    day_indices = [k for k,v in days_map.items() if v in selected_days]
    filtered_data = filtered_data[filtered_data['day_of_week'].isin(day_indices)]

    # Sélection des features
    available_features = ['energy_demand', 'month', 'hour', 'day_of_week', 
'is_weekend', 'is_holiday']

    # Ajouter les colonnes disponibles
    optional_cols = ['embedded_wind_generation', 'embedded_solar_generation',
                'ifa_flow', 'ifa2_flow', 'britned_flow', 'moyle_flow',
                'temperature', 'humidity']

    for col in optional_cols:
        if col in filtered_data.columns:
            available_features.append(col)

    selected_features = st.sidebar.multiselect(
        "Features pour le modèle",
        available_features,
        default=['energy_demand', 'month', 'hour', 'day_of_week', 'is_holiday']
    )
    if 'energy_demand' not in selected_features:
        st.error("❌ 'energy_demand' doit être inclus")
        st.stop()

    # --- VISUALISATIONS DES DONNÉES HISTORIQUES ---
    st.header("📈 Analyse Historique de la Demande Énergétique")

    # Sélection du type de visualisation
    viz_type = st.selectbox(
        "Type de visualisation historique",
        ["Série temporelle", "Distribution horaire", "Analyse saisonnière",
"Corrélations"]
    )

    if viz_type == "Série temporelle":
        # Agrégation par période
        agg_period = st.selectbox("Période d'agrégation", ["Heure", "Jour", "Mois"])

        if agg_period == "Heure":
            ts_data = filtered_data['energy_demand'].resample('H').mean()
        elif agg_period == "Jour":
            ts_data = filtered_data['energy_demand'].resample('D').mean()
        else:  # Mois
            ts_data = filtered_data['energy_demand'].resample('M').mean()
        fig = px.line(ts_data, title=f"Demande énergétique - Agrégation
{agg_period.lower()}")
        fig.update_layout(xaxis_title="Date", yaxis_title="Demande (MW)",
template="plotly_dark")
        st.plotly_chart(fig, use_container_width=True)

    elif viz_type == "Distribution horaire":
        # Heatmap de la demande par heure et jour de semaine
        pivot_data = filtered_data.pivot_table(
            values='energy_demand',
            index='hour',
            columns='day_of_week',
            aggfunc='mean'
        )
        pivot_data.columns = [days_map[i] for i in pivot_data.columns]

        fig = px.imshow(
            pivot_data,
            title="Demande moyenne par heure et jour de semaine (MW)",
            aspect="auto",
            color_continuous_scale="Viridis"
        )
        fig.update_layout(template="plotly_dark")
        st.plotly_chart(fig, use_container_width=True)
    elif viz_type == "Analyse saisonnière":
        # Demande par mois
        monthly_data = filtered_data.groupby('month')['energy_demand'].mean()
        fig = px.bar(monthly_data, title="Demande moyenne par mois")
        fig.update_layout(xaxis_title="Mois", yaxis_title="Demande moyenne (MW)",
template="plotly_dark")
        st.plotly_chart(fig, use_container_width=True)

    else:  # Corrélations
        corr_data = filtered_data[selected_features].corr()
        fig = px.imshow(
            corr_data,
            title="Matrice de corrélation",
            aspect="auto",
            color_continuous_scale="RdBu",
            zmin=-1, zmax=1
        )
        fig.update_layout(template="plotly_dark")
        st.plotly_chart(fig, use_container_width=True)

    # --- DEEP LSTM PREDICTION ---
    st.header("� Prédiction Deep LSTM")

    # Paramètres du modèle
```

```
st.sidebar.header("⍰ Paramètres Deep LSTM")
seq_len = st.sidebar.slider("Longueur de séquence", 24, 168, 48)
test_size = st.sidebar.slider("Taille du test (%)", 10, 40, 20)

# Préparation des données pour le modèle Deep LSTM
def prepare_deep_lstm_data(data, features, sequence_length, test_size=0.2):
    """Prépare les données pour le modèle Deep LSTM"""
    df_model = data[features].copy()

    # Normalisation
    scaler = MinMaxScaler()
    scaled_data = scaler.fit_transform(df_model)

    # Création des séquences
    X, y = [], []
    for i in range(sequence_length, len(scaled_data)):
        X.append(scaled_data[i-sequence_length:i])
        y.append(scaled_data[i, 0])  # Première colonne = energy_demand

    X, y = np.array(X), np.array(y)

    # Split train/test
    split_idx = int(len(X) * (1 - test_size))
    X_train, X_test = X[:split_idx], X[split_idx:]
```

```
        y_train, y_test = y[:split_idx], y[split_idx:]

        return X_train, X_test, y_train, y_test, scaler, df_model

    if st.button("🚀 Entraîner le modèle Deep LSTM"):
        with st.spinner("Préparation des données et entraînement du modèle Deep
LSTM..."):

            # Préparation des données
            X_train_keras, X_test_keras, y_train_keras, y_test_keras, scaler, df_model
= prepare_deep_lstm_data(
                filtered_data, selected_features, seq_len, test_size/100
            )

            st.info(f"📊 Dimensions des données: X_train {X_train_keras.shape}, X_test
{X_test_keras.shape}")

            # --- VOTRE MODÈLE DEEP LSTM EXACT ---
            model = Sequential()
            model.add(LSTM(256, input_shape=(X_train_keras.shape[1],
X_train_keras.shape[2]), return_sequences=True))
            model.add(Dropout(0.5))
            model.add(LSTM(128, return_sequences=True))
            model.add(Dropout(0.5))
            model.add(LSTM(32))
            model.add(Dropout(0.5))
            model.add(Dense(1))

            model.compile(loss=root_mean_squared_error, optimizer="adam")

            # Callbacks
            early_stopping = EarlyStopping(monitor='val_loss', patience=8, verbose=0,
mode='min')

            os.makedirs("./models_data/deep_lstm", exist_ok=True)

            checkpoint_save = ModelCheckpoint(
                "./models_data/deep_lstm/checkpoint.weights.h5",
                save_weights_only=True,
                monitor='val_loss',
                mode='min',
                save_best_only=True
            )

            reduce_lr_loss = ReduceLROnPlateau(monitor='val_loss', factor=0.1,
patience=5, verbose=0, mode='min')

            # Entraînement
            history_deep_lstm = model.fit(
                X_train_keras,
                y_train_keras,
                epochs=100,
                batch_size=144,
                validation_data=(X_test_keras, y_test_keras),
                callbacks=[early_stopping, checkpoint_save, reduce_lr_loss],
                verbose=0
            )

            # --- COURBE DE LOSS ---
            st.subheader("📈 Évolution de la Loss")

            fig, ax = plt.subplots(figsize=(10, 6))
            ax.plot(history_deep_lstm.history["loss"], label="Training Loss",
linewidth=2)
            ax.plot(history_deep_lstm.history["val_loss"], label="Validation Loss",
linewidth=2)
            ax.set_xlabel("Epoch")
            ax.set_ylabel("Loss (RMSE)")
            ax.set_title("Loss Evolution - Deep LSTM")
            ax.legend()
            ax.grid(True, alpha=0.3)
            st.pyplot(fig)

            # --- PRÉDICTIONS ---
            pred_deep_lstm = model.predict(X_test_keras, verbose=0)
            # Conversion inverse (votre méthode)
            test_data_keras_s = X_test_keras[:, -1, :]
            results_deep_lstm = test_data_keras_s.copy()
            results_deep_lstm[:, -1] =
pred_deep_lstm.reshape(pred_deep_lstm.shape[0])
            results_deep_lstm = scaler.inverse_transform(results_deep_lstm)

            # DataFrame de résultats
            test_dates = filtered_data.index[seq_len:][-len(y_test_keras):]
            result_frame = pd.DataFrame({
                'actual': scaler.inverse_transform(np.column_stack([y_test_keras,
X_test_keras[:, -1, 1:]]))[:, 0],
                'pred_deep_lstm': results_deep_lstm[:, -1]
            }, index=test_dates)

            # --- MÉTRIQUES ---
            mape_deep_lstm =
mean_absolute_percentage_error(result_frame["actual"],
result_frame["pred_deep_lstm"])
            rmse_deep_lstm = np.sqrt(mean_squared_error(result_frame["actual"],
result_frame["pred_deep_lstm"]))

            st.success("✅ Modèle Deep LSTM entraîné avec succès !")

            col1, col2 = st.columns(2)
            col1.metric("MAPE", f"{mape_deep_lstm:.2f}%")
```

```
col2.metric("RMSE", f"{rmse_deep_lstm:.2f} MW")

# --- VISUALISATION COMPLÈTE ---
st.subheader("📊 Prédictions complètes")

fig_full = go.Figure()
fig_full.add_trace(go.Scatter(
    x=result_frame.index, y=result_frame['actual'],
    mode='lines', name='Demande Réelle',
    line=dict(color='#00d4ff', width=2)
))
fig_full.add_trace(go.Scatter(
    x=result_frame.index, y=result_frame['pred_deep_lstm'],
    mode='lines', name='Prédiction Deep LSTM',
    line=dict(color='#ff6b6b', width=1.5, dash='dash')
))
fig_full.update_layout(
    title="Comparaison complète - Prédictions Deep LSTM",
    xaxis_title="Date", yaxis_title="Demande (MW)",
    template="plotly_dark", height=500
)
st.plotly_chart(fig_full, use_container_width=True)

# --- ZOOM 2 SEMAINES ---
```

```
    st.subheader("🔍 Zoom sur 2 semaines")

    if len(result_frame) > 336:
        begin_date = result_frame.index[-336]
        end_date = result_frame.index[-1]
    else:
        begin_date = result_frame.index[0]
        end_date = result_frame.index[-1]

    period_data = result_frame[(result_frame.index > begin_date) &
(result_frame.index < end_date)]

    fig_zoom, ax_zoom = plt.subplots(figsize=(15, 5))
    ax_zoom.plot(period_data.index, period_data["actual"], "-o", label="Test
Set", markersize=3)
    ax_zoom.plot(period_data.index, period_data["pred_deep_lstm"], "-d",
label="Deep LSTM", markersize=2)
    ax_zoom.legend()
    ax_zoom.set_title("Prediction on Test Set - Two Weeks")
    ax_zoom.set_ylabel("Energy Demand (MW)")
    ax_zoom.set_xlabel("Date")
    ax_zoom.grid(True, alpha=0.3)
    plt.xticks(rotation=45)
    st.pyplot(fig_zoom)
```

```
            # --- EXPORT ---
            st.subheader("💾 Export des résultats")
            csv = result_frame.reset_index().rename(columns={'index': 'datetime'}).to_csv(index=False)
            st.download_button(
                "📥 Télécharger les prédictions CSV",
                csv,
                file_name="deep_lstm_predictions.csv",
                mime="text/csv"
            )

else:
    st.info("""
    📤 **Instructions:**
    1. **Uploader** votre fichier historic_demand.csv
    2. **Colonnes requises:** settlement_date, england_wales_demand
    3. **Explorez** les visualisations historiques
    4. **Entraînez** le modèle Deep LSTM optimisé
    """)

# Footer
st.markdown("---")
st.markdown("<div style='text-align: center; color: #666;'>Deep LSTM Energy Prediction Dashboard</div>", unsafe_allow_html=True)
```