Travaux Pratiques — Cours d'Apprentissage Profond

Aurélien Plyer & Kevin Helvig

9 octobre 2025

Table des matières

1 Session 3 — RNN, Seq2Seq et Modèles d'État Sélectifs

Cette session se déroule en trois TP successifs, chacun a pour but de générer un notebook Jupyter. Le fil rouge consiste à explorer les réseaux de neuronnes récurrent. A commencer par implémenter et comparer un RNN simple et un LSTM. Puis à déployer ces cellules dans le cadre d'une structure de prédiction/traduciton de séquence en mode encodeur / décodeur (Seq2Seq). Au final une exploration des SSM est proposé comme sujet d'ouverture.

1.1 TP 3.1 — RNN et LSTM sur séries temporelles

Objectif : implémenter **manuellement** un RNN tanh puis un LSTM, les entraîner sur un jeu synthétique multi-sinusoïdal et sur une série réelle de températures.

1. Générer les fenêtres glissantes. Après avoir créé la série synthétique (sinusoïdes + bruit), complétez la fonction ci-dessous située dans le notebook :

```
def build_windowed_tensors(features: np.ndarray, targets: np.
    ndarray, seq_len: int):
    '''Retourne X:(N, seq_len, d_in) et y:(N, d_out) en utilisant
        des fenêtres glissantes.
    features: (T, d_in)  -- ex. colonnes [signal]
    targets:  (T, d_out) -- ex. colonnes [signal décalé]'''
    # TODO
    return X_tensor, y_tensor
```

Pour générer la série synthétique multi-fréquences utilisée dans le notebook, vous pouvez partir du bloc suivant (à placer dans une cellule dédiée) :

```
import numpy as np

def generate_synthetic_series(n_samples: int = 8000, seed: int =
    123):
    '''Crée un signal multi-fréquences + bruit et retourne (
        features, targets).'''
    rng = np.random.default_rng(seed)
    t = np.arange(n_samples, dtype=np.float32)
    base = (0.8 * np.sin(2 * np.pi * 0.01 * t)
            + 0.4 * np.sin(2 * np.pi * 0.04 * t + np.pi / 4))
    trend = 0.0015 * t
    noise = 0.05 * rng.standard_normal(n_samples)
```

```
    series = (base + trend + noise).astype(np.float32)
    features = series[:-1][:, None]  # (T-1, 1)
    targets = series[1:][:, None]    # (T-1, 1)
    return features, targets

features, targets = generate_synthetic_series()
print(features.shape, targets.shape)
```

Attendus.

— Chaque fenêtre doit utiliser les `seq_len` pas immédiatement *avant* la cible (pas de fuite d'information).

— Vérifiez avec `assert X.shape[0] == y.shape[0]` et que `X[0, -1]` correspond à la dernière observation juste avant `y[0]`.

— Tracez une fenêtre représentative (cf. figure fournie dans le notebook) pour confirmer le décalage.

2. Implémenter le RNN "à la main". Reprenez les équations :

$$h_t = \tanh(W_{xh}x_t + W_{hh}h_{t-1} + b_h),$$
$$\hat{y}_t = W_{hy}h_t + b_y.$$

Complétez la classe `ManualRNNRegressor` :

```
class ManualRNNRegressor(nn.Module):
    def __init__(self, input_dim: int, hidden_dim: int, output_dim:
        int):
        super().__init__()
        self.hidden_dim = hidden_dim
        self.input_linear = nn.Linear(input_dim, hidden_dim)
        self.hidden_linear = nn.Linear(hidden_dim, hidden_dim)
        self.output_linear = nn.Linear(hidden_dim, output_dim)

    def forward(self, x):
        # x: (batch, seq_len, input_dim)
        # TODO: initialiser h0, dérouler la boucle temporelle,
           renvoyer la dernière prédiction
        return y_hat
```

Tests rapides.

— Vérifiez la stabilité avec un batch jouet : `torch.zeros(2, 5, input_dim)`.

— Confirmez que la MSE décroît en lançant `train_model` pendant 50 itérations (val $<$ train).

3. LSTM : portes d'entrée, d'oubli, et de sortie. Rappelez-vous les équations :

$$\begin{bmatrix} f_t \\ i_t \\ g_t \\ o_t \end{bmatrix} = \begin{bmatrix} \sigma \\ \sigma \\ \tanh \\ \sigma \end{bmatrix} \left(W \begin{bmatrix} x_t \\ h_{t-1} \end{bmatrix} + b \right),$$

$$c_t = f_t \odot c_{t-1} + i_t \odot g_t,$$

$$h_t = o_t \odot \tanh(c_t).$$

Complétez `ManualLSTMRegressor` sans utiliser `nn.LSTM`. Exigez que `self.gates` produise un tenseur de taille `(batch, 4 * hidden_dim)`.

À vérifier.

— Comme pour le RNN, testez sur une entrée nulle et assurez-vous que la sortie reste finie.

— Comparez la MSE finale RNN vs LSTM : le LSTM doit converger plus rapidement.

— Générez les prédictions déroulées libres (fonction `get_predictions`) et tracez les séquences prédites vs cibles.

4. Série réelle : températures de Delhi.

a) Implémentez `add_seasonal_features` : ajoutez `day_sin`, `day_cos` suivant $\sin(2\pi d/365)$.

b) Reprenez `build_windowed_tensors` avec `seq_len = 60` pour former train/val/test.

c) Entraînez successivement le RNN et le LSTM et comparez les MSE.

Pour charger les données climat (fichiers `DailyDelhiClimateTrain.csv` / `DailyDelhiClimateTest.csv` dans `Session3/data/`), insérez par exemple :

```
import pandas as pd
from pathlib import Path

DATA_DIR = Path("Session3/data")
train_path = DATA_DIR / "DailyDelhiClimateTrain.csv"
test_path = DATA_DIR / "DailyDelhiClimateTest.csv"

def load_and_prepare_climate(train_file: Path, test_file: Path):
    df_train = pd.read_csv(train_file, parse_dates=["date"],
       index_col="date")
    df_test = pd.read_csv(test_file, parse_dates=["date"],
       index_col="date")
    df_full = pd.concat([df_train, df_test])
    df_full = df_full.sort_index().interpolate(method="time")
    return df_full
```

```
climate_df = load_and_prepare_climate(train_path, test_path)
print(climate_df.head())
```

Le notebook sépare ensuite train/val/test en respectant l'ordre temporel.

Tracez absolument les courbes prédictions vs vérités en degrés Celsius. Un LSTM correctement entraîné doit reproduire les tendances saisonnières et lisser le bruit.

Livrables pour le notebook.

— Captures des courbes de pertes train/val.
— Graphiques synthétiques (jeu sinus) et réels (températures) avec analyse écrite.
— Tableau récapitulatif MSE RNN vs LSTM sur les deux jeux.

1.2 TP 3.2 — Seq2Seq, attention et schedule sampling

Objectif : construire une architecture encodeur–décodeur GRU, y ajouter une attention bilinéaire puis un schedule sampling, et transférer sur un problème réel (climat).

1. Jeu synthétique multi-fréquences. Reprenez le générateur fourni. Vous devez ensuite créer le dataset PyTorch :

```
def to_tensor_dataset(x_enc, x_dec, y_dec):
    '''Empile (enc_inputs, dec_inputs, dec_targets) dans un
       TensorDataset.'''
    # TODO: torch.from_numpy(...).float()
    return TensorDataset(enc_tensor, dec_tensor, tgt_tensor)
```

Pour générer les échantillons synthétiques multi-fréquences (encodeur/décodeur), utilisez le gabarit complet :

```
import numpy as np

def generate_seq2seq_data(n_samples: int = 6000, enc_len: int = 50,
    dec_len: int = 30, seed: int = 123):
    rng = np.random.default_rng(seed)
    t = np.arange(n_samples, dtype=np.float32)
    base = 0.8 * np.sin(2 * np.pi * 0.01 * t)
    harmonic = 0.4 * np.sin(2 * np.pi * 0.04 * t + np.pi / 4)
    drift = 0.003 * t
    signal = base + harmonic + drift
    noise = 0.05 * rng.standard_normal(n_samples)
    series = (signal + noise).astype(np.float32)
```

```
    enc_inputs, dec_inputs, dec_targets = [], [], []
    total_len = enc_len + dec_len
    for start in range(n_samples - total_len - 1):
        window = series[start:start + total_len + 1]
        enc_inputs.append(window[:enc_len][:, None])
        dec_inputs.append(window[enc_len:enc_len + dec_len][:, None
            ])
        dec_targets.append(window[enc_len + 1:enc_len + dec_len +
            1][:, None])

    x_enc = np.stack(enc_inputs, axis=0)
    x_dec = np.stack(dec_inputs, axis=0)
    y_dec = np.stack(dec_targets, axis=0)
    return x_enc, x_dec, y_dec

x_enc, x_dec, y_dec = generate_seq2seq_data()
print(x_enc.shape, x_dec.shape, y_dec.shape)
```

Vérifiez que `len(dataset)== x_enc.shape[0]` et que `dec_tensor[:, 0]` correspond bien à la dernière observation de l'encodeur (amorçage du décodeur).

2. Encodeur et décodeur de base. Complétez `EncoderGRU` et `DecoderGRU` afin de réutiliser les états cachés. Pensez à :

- retourner *toute* la séquence d'états encodeur (pour l'attention future) ;
- initialiser le décodeur avec l'état final de l'encodeur ;
- gérer le teacher forcing via `teacher_forcing_ratio` dans `run_epoch_seq2seq`.

3. Entraînement avec teacher forcing constant. Utilisez la routine fournie :

```
train_losses, val_losses = train_seq2seq(
    model, train_loader, val_loader,
    criterion=nn.MSELoss(),
    optimizer=torch.optim.Adam(model.parameters(), lr=1e-3),
    teacher_forcing_ratio=0.7,
    epochs=40,
)
```

Contrôle. Tracez la prédiction déroulée libre de 50 pas et vérifiez que la dérive reste bornée (sinusoïdes correctement suivies).

4. Attention bilinéaire. Introduisez la matrice W_a et implémentez :

$$\alpha_{t,s} = \frac{\exp(h_t^\top W_a h_s^{\text{enc}})}{\sum_{s'} \exp(h_t^\top W_a h_{s'}^{\text{enc}})}, \quad c_t = \sum_s \alpha_{t,s} h_s^{\text{enc}}.$$

```
class BilinearAttention(nn.Module):
    def __init__(self, hidden_dim):
        super().__init__()
        self.W = nn.Parameter(torch.randn(hidden_dim, hidden_dim) *
            0.1)

    def forward(self, h_dec, h_enc):
        '''h_dec: (B, L_dec, H), h_enc: (B, L_enc, H) -> context: (
           B, L_dec, H)'''
        # TODO: scores = h_dec @ W @ h_enc^T, softmax sur L_enc,
           pondérer h_enc
        return context, attn_weights
```

Tests.

— Assurez-vous que `attn_weights.sum(dim=-1)` vaut 1 (tolérance 10^{-5}).

— Visualisez une carte d'attention (heatmap) pour vérifier que les poids suivent le motif sinusoïdal.

5. Schedule sampling. Faites décroître le ratio de teacher forcing p_t linéairement de 0.9 à 0.2 sur 50 époques :

$$p_t = p_{\max} - (p_{\max} - p_{\min}) \times \frac{\text{epoch}}{\text{epochs} - 1}.$$

Tracez simultanément les courbes train/val et la trajectoire de p_t.

6. Données réelles : humidité (Delhi).

a) Implémentez `preprocess_climate` (interpolation temporelle + normalisation `StandardScaler`).

b) Construisez les tenseurs via `build_real_seq2seq` (fenêtres encodeur 28 jours, décodeur 14 jours).

c) Utilisez l'encodeur avec attention et un `DecoderAttentionExog` qui concatène variables exogènes (pression, température, etc.).

Attendus.

— Courbes de prédiction sur 14 jours montrant la dynamique d'humidité.

— Table de métriques (MSE / MAE) teacher forcing constant vs schedule sampling.

— Carte d'attention moyenne illustrant la zone temporelle la plus informative.

1.3 TP 3.3 — Modèles d'État (SSM) et variantes sélectives

Objectif : relier la discrétisation continue $Ax + Bu$ à un noyau de convolution causal, entraîner un SSM diagonal sur un problème de filtrage, puis introduire une variante sélective inspirée de Mamba pour l'adding task.

1. Discrétisation exacte (C → D). Complétez :

```
def c2d_diagonal(a_diag: torch.Tensor, B: torch.Tensor, dt: float):
    '''Retourne A_bar (N,) et B_bar (N, d_in) pour A = diag(a).'''
    # TODO: exp(dt * a), gérer le cas a ~ 0
    return A_bar, B_bar
```

Formule à implémenter :

$$\bar{A}_i = e^{\Delta t a_i}, \qquad \bar{B}_{i,:} = \frac{e^{\Delta t a_i} - 1}{a_i} B_{i,:}, \quad \text{et } \bar{B}_{i,:} = \Delta t \, B_{i,:} \text{ si } |a_i| < 10^{-6}.$$

Test. Comparez à l'approximation d'Euler : la norme relative doit être $< 5\%$ pour $\Delta t \leq 0.1$.

2. Noyau causal et convolution.

```
def ssm_kernel_from_diagonal(C, A_bar, B_bar, L):
    '''Construit k:(d_out, d_in, L) avec k_t = C A_bar^t B_bar.'''
    # TODO: boucle sur t, accumuler dans une liste, torch.stack
       (..., dim=-1)
    return kernel
```

Vérifiez que `causal_conv1d_from_kernel` et `forward_scan` d'une même couche donnent la même sortie :

```
with torch.no_grad():
    y_conv, kernel = layer.forward_conv(u)
    y_scan = layer.forward_scan(u)
    assert torch.allclose(y_conv, y_scan, atol=1e-4)
```

3. Filtrage passe-bas. Avant l'entraînement, vous pouvez reprendre intégralement la classe génératrice fournie dans le notebook :

```
import numpy as np
import torch
from torch.utils.data import Dataset

class Filtering1DDataset(Dataset):
```

```
    '''Signal basse fréquence + hautes fréquences + bruit (filtrage
        causal).'''
    def __init__(self, n_samples=800, seq_len=256, n_low=2, n_high
       =2, noise_std=0.2, seed=0):
        super().__init__()
        rng = np.random.RandomState(seed)
        t = np.arange(seq_len)[None, :]
        xs, ys = [], []
        for _ in range(n_samples):
            low_freqs = rng.uniform(0.5, 2.0, size=(n_low, 1))
            low_phs = rng.uniform(0, 2 * np.pi, size=(n_low, 1))
            low = np.sin(2 * np.pi * low_freqs * t / seq_len +
               low_phs).sum(axis=0)

            high_freqs = rng.uniform(6.0, 20.0, size=(n_high, 1))
            high_phs = rng.uniform(0, 2 * np.pi, size=(n_high, 1))
            high = 0.5 * np.sin(2 * np.pi * high_freqs * t /
               seq_len + high_phs).sum(axis=0)

            noise = noise_std * rng.randn(1, seq_len)
            u = (low + high + noise).astype(np.float32)
            y = low.astype(np.float32)[:, None]
            xs.append(u.T)
            ys.append(y)
        self.x = np.stack(xs, axis=0)
        self.y = np.stack(ys, axis=0)

    def __len__(self):
        return self.x.shape[0]

    def __getitem__(self, idx):
        return torch.from_numpy(self.x[idx]), torch.from_numpy(self
           .y[idx])

train_set = Filtering1DDataset(n_samples=800, seq_len=256, seed=0)
val_set = Filtering1DDataset(n_samples=200, seq_len=256, seed=1)
print(train_set[0][0].shape, train_set[0][1].shape)
```

Entraînez ensuite `SSMFilterNet` sur `Filtering1DDataset`. Cible : retrouver la composante basse fréquence. Reproduisez les figures cibles :

Analysez :

— la décroissance temporelle du noyau (stabilité) ;

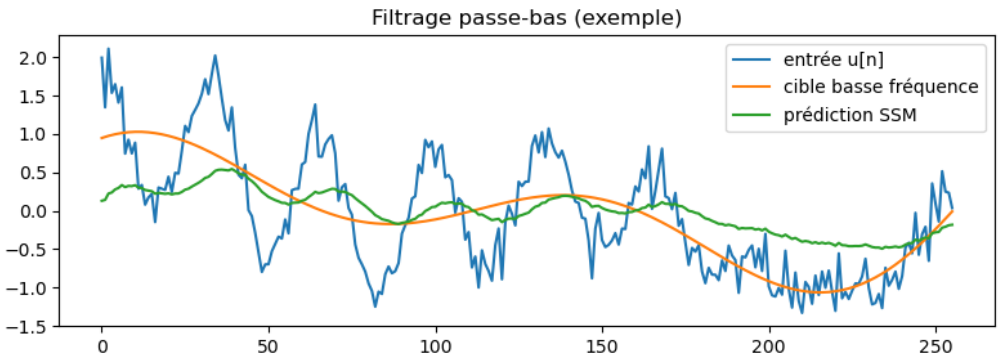

FIGURE 1 – Signal bruité vs cible basse fréquence.

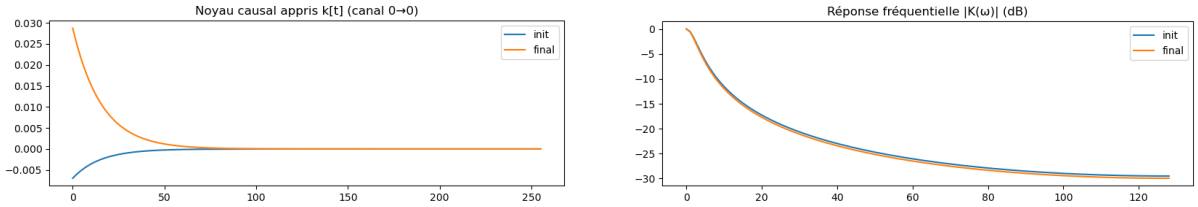

FIGURE 2 – Noyau causal appris et réponse fréquentielle attendue (filtre passe-bas).

— l'atténuation des hautes fréquences (-20 dB ou plus) ;
— la cohérence conv vs scan (figure ci-dessous).

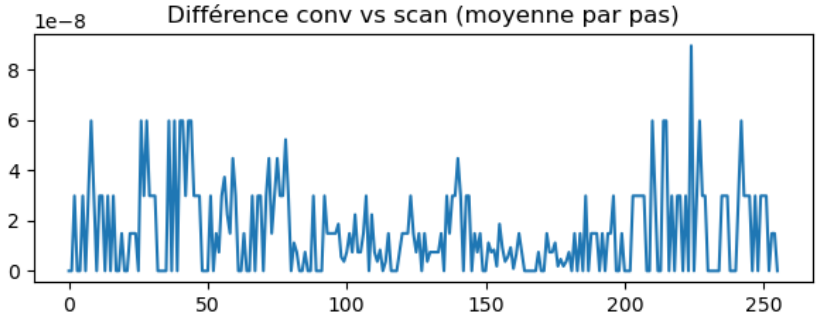

FIGURE 3 – Différence moyenne $|y_{\text{conv}} - y_{\text{scan}}|$ à contrôler.

4. SSM sélectif et adding task. Le modèle `SelectiveSSMLayer` applique :

$$\Delta_t = \text{softplus}(W_\Delta u_t + b), \quad x_{t+1} = e^{\Delta_t A} x_t + \left(\frac{e^{\Delta_t A} - 1}{A} \right) B u_t, \quad y_t = \sigma(W_g u_t) \odot (C x_t) + D u_t.$$

Vous pouvez utiliser la définition suivante (copiée du notebook) pour la tâche de somme :

```
class AddingTaskDataset(Dataset):
```

```
    '''Deux positions aléatoires marquées dont on doit sommer les
       valeurs.'''
    def __init__(self, n_samples=800, seq_len=256, seed=0):
        super().__init__()
        rng = np.random.RandomState(seed)
        xs, ys = [], []
        for _ in range(n_samples):
            a = rng.rand(seq_len, 1).astype(np.float32)
            m = np.zeros((seq_len, 1), dtype=np.float32)
            idxs = rng.choice(seq_len, size=2, replace=False)
            m[idxs, 0] = 1.0
            y = np.array([a[idxs, 0].sum()], dtype=np.float32)
            xs.append(np.concatenate([a, m], axis=1))
            ys.append(y)
        self.x = np.stack(xs, axis=0)
        self.y = np.stack(ys, axis=0)

    def __len__(self):
        return self.x.shape[0]

    def __getitem__(self, idx):
        return torch.from_numpy(self.x[idx]), torch.from_numpy(self
           .y[idx])

train_set = AddingTaskDataset(n_samples=800, seq_len=256, seed=2)
val_set = AddingTaskDataset(n_samples=200, seq_len=256, seed=3)
print(train_set[0][0].shape, train_set[0][1].shape)
```

Complétez la boucle temporelle (déjà amorcée dans le notebook) et vérifiez :

- `deltas.mean()` reste < 1.0 ;
- les Δ_t augmentent autour des indices où le masque vaut 1.

Reproduisez les figures cibles pour valider votre implémentation :

5. Lecture guidée.

- Comparez la sortie filtrée et la cible : la courbe prédite doit suivre la basse fréquence.
- Inspectez le noyau causal et sa FFT : confirmez le caractère passe-bas.
- Pour l'adding task, concentrez-vous sur la précision du scalaire final et la dynamique des Δ_t.

Livrables pour le notebook.

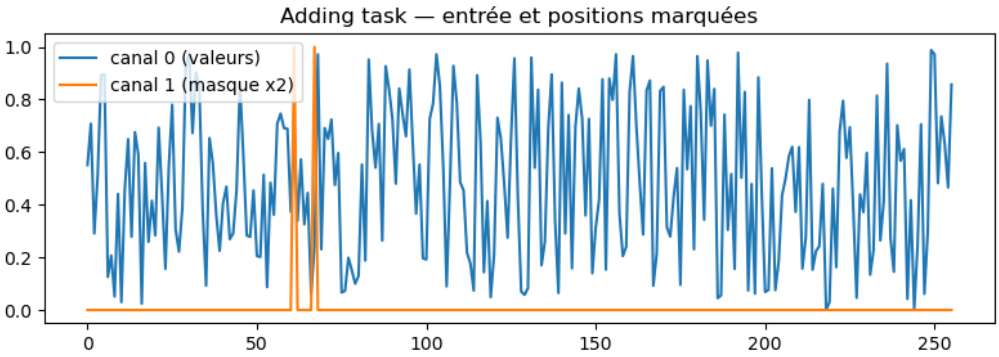

FIGURE 4 – Adding task : canal valeurs, masque binaire et prédiction finale (scalaire).

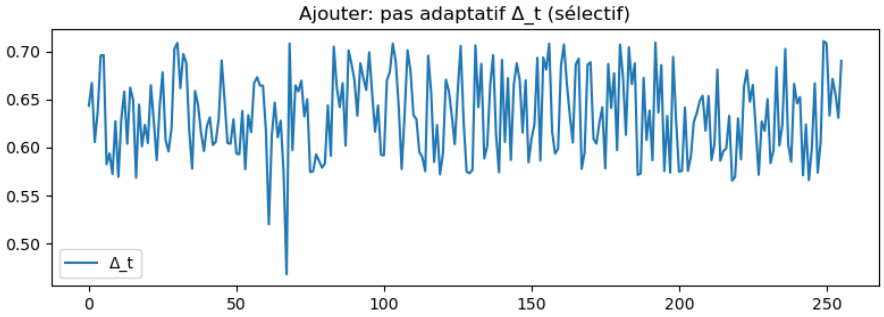

FIGURE 5 – Pas adaptatifs Δ_t attendus : pics alignés avec les marqueurs.

— Implémentation des fonctions `c2d_diagonal` et `ssm_kernel_from_diagonal` documentées.
— Graphiques reproduisant les cibles ci-dessus avec commentaires.
— Tableau de performances : MSE filtrage (train/val/test) et MSE/MAE adding task.

Conseil final. Traitez chaque notebook comme un mini-projet autonome : verrouillez une étape avant de passer à la suivante. Utilisez systématiquement des tests rapides (asserts, courbes) pour valider vos implémentations avant de lancer des entraînements longs.