# MTH 602 Scientific Machine Learning

Homework 7

12/9/2025

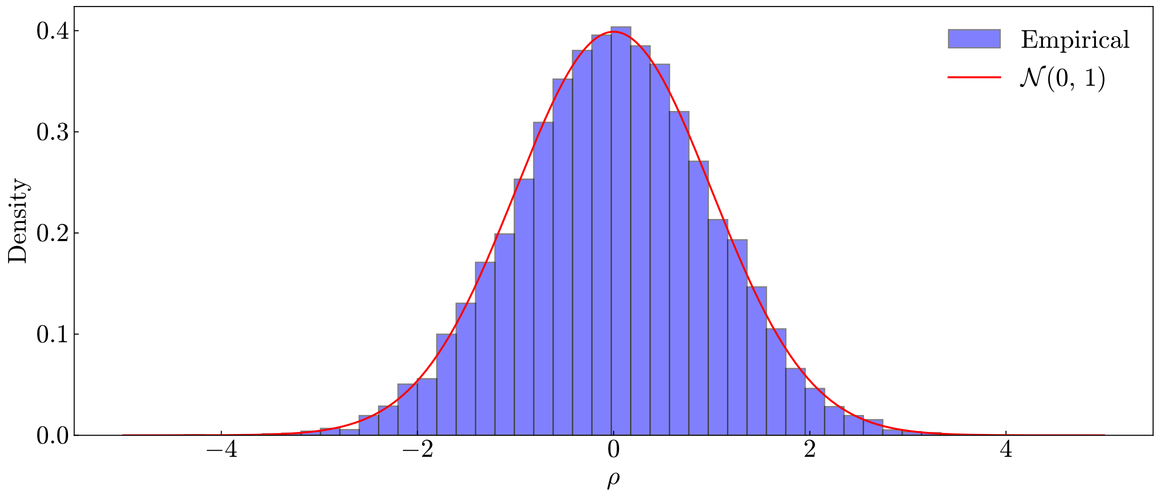
S. M. Mahfuzul Hasan

02181922

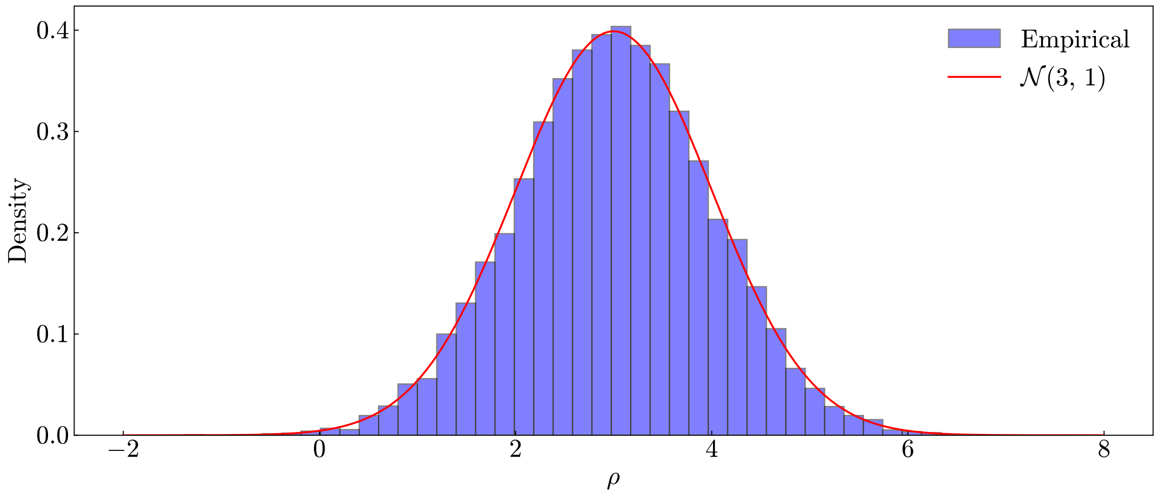
## I. CNNS FOR SIGNAL DETECTION: CAN A NONLINEAR CLASSIFIER BEAT THE MATCHED FILTER?

### B. Matched-filter recap and extended study

**1.** For each amplitude $A \in \{0, 3\}$, 20000 independent realizations of $\mathbf{y}$ are generated and corresponding $\rho$ values are computed.



(a)



(b)

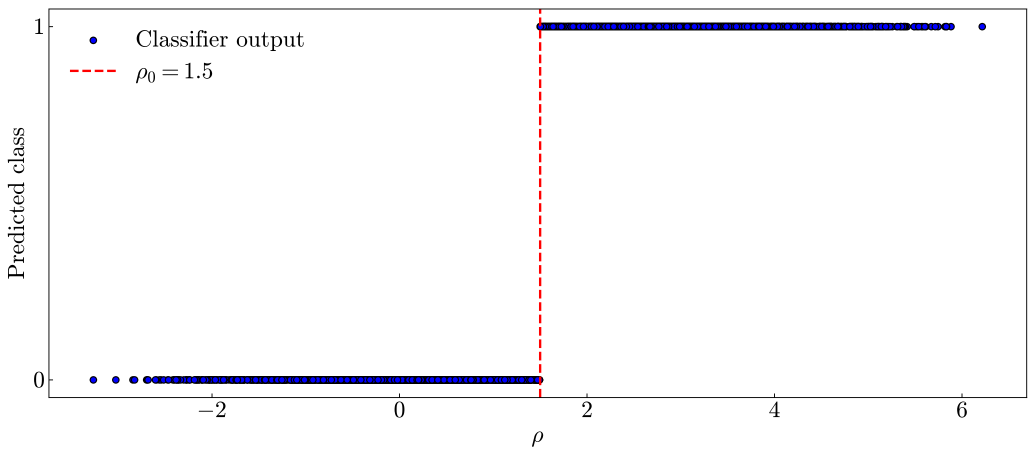
Figure 1: Empirical histogram of $\rho$ and theoretical Gaussian PDF for (a) $A = 0$ and (b) $A = 3$.

The empirical distributions of $\rho$ and theoretical Gaussian PDFs in fig. 1 for both the amplitude cases show very good agreement.

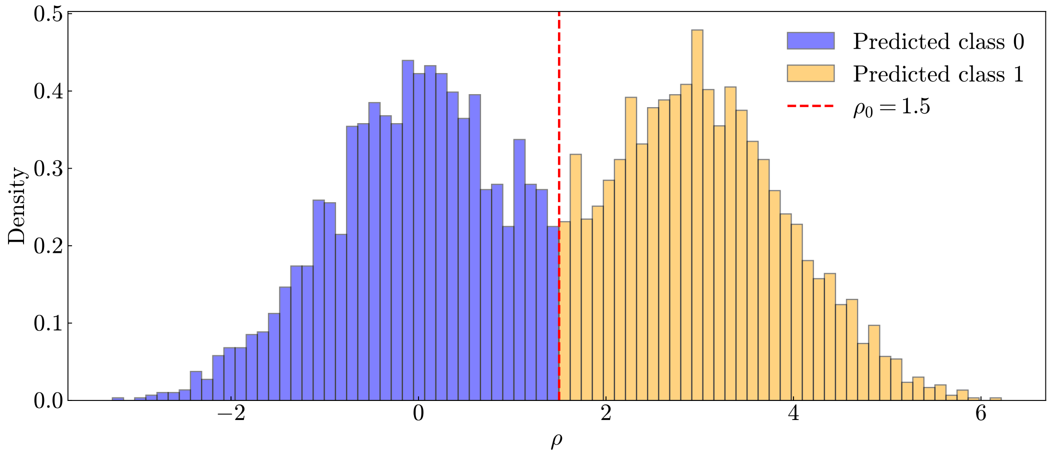
**2.** Please refer to listing 1 for `filter_classifier`.

To verify that the function is working properly, first, it is checked whether there is any mismatch between the prediction of class between theory and `filter_classifier`.

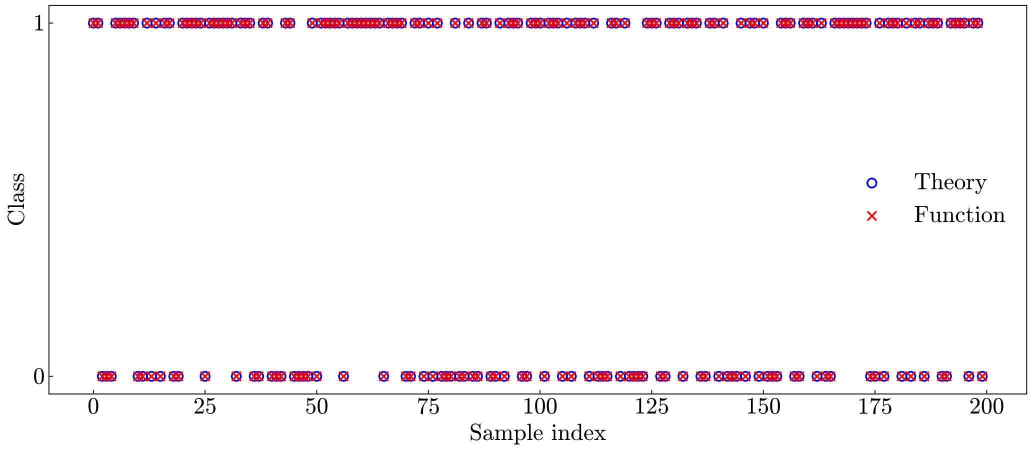
$$\text{The number of mismatched prediction} = 0$$



(a)



(b)



(c)

Figure 2: (a) Scatter plot of predicted class against $\rho$, (b) PDF of predicted class based on $\rho_0$, and (c) comparison between theory and `filter_classifier` for the prediction of class for the first 200 samples.

Then, the predicted class and their PDF are plotted against the detection statistics in fig. 2a and 2b respectively. Both show clear separation of "signal" and "no-signal" based on the threshold value $\rho_0$. Along with that, the comparison between the throretical estimation of classes and estimation of classes by `filter_classifier` shows complete agreement in fig. 2c.

## C. Building a labeled dataset for a CNN

**1.** Please refer to listing 1 for the construction of a labeled dataset of time-series

$$\left\{\left(\mathbf{y}^{(k)}, c^{(k)}\right)\right\}_{k=1}^{M}, \qquad c^{(k)} \in \{0, 1\}$$

The number of samples is set at 20000 (found to be sufficiently optimal), and the splitting renders

Number of no-signal examples: 9933 (49.665%)

Number of signal examples: 10067 (50.335%)

which shows almost 50% class balance.

**2.** The dataset is split into 80% training and 20% testing data. The class balance in the training and testing data:

Training: 50.337% signal

Testing: 50.325% signal

## D. Design and training of a 1D CNN classifier

**1.** Parameters in the 1D CNN:

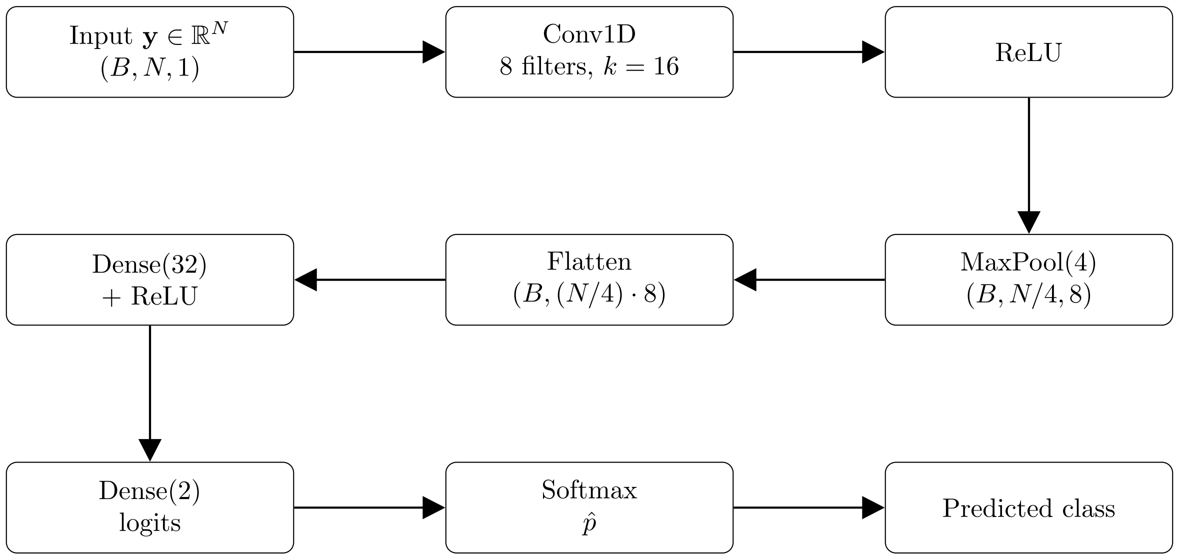


Figure 3: 1D CNN architecture for signal detection.

`Conv1D` layer $(1000 \to 1000)$: $16 \times 1 \times 8 + 8 = 136$

Dense layer $(1000/4 \cdot 8 = 2000 \to 32)$: $2000 \times 32 + 32 = 64032$

Dense output layer $(32 \to 2)$: $32 \times 2 + 2 = 66$

Total trainable parameters = 136 + 64032 + 66 = 64234

Total trainable parameters reported from listing 1 and 2 = 64234

**2.** The dataset is normalized by using the mean and standard deviation of training dataset.

$$\tilde{y}_i = \frac{y_i - \mu_{train}}{\sigma_{train}}$$

where, $i = 1, 2, ..., N$.

Only training data statistics are used so that

- validation and test data do not influence the model by leaking information.
- optimization remains stabilized throughout the training.

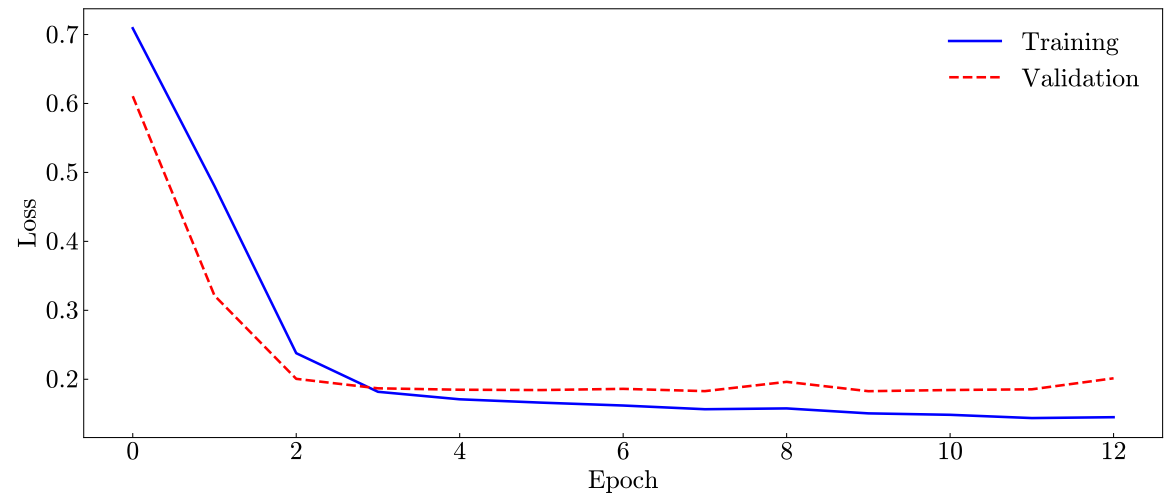
`Conv1D` layer expects input of the shape $(B, N, C)$. So, $\mathbf{y} \in \mathbb{R}^N$ is reshaped to (B,N,1) (where, $B$ is batch size) to add a channel dimension and make it compatible with convolution.

**3.** Please refer to listing 1 for the splitting and training of the data.

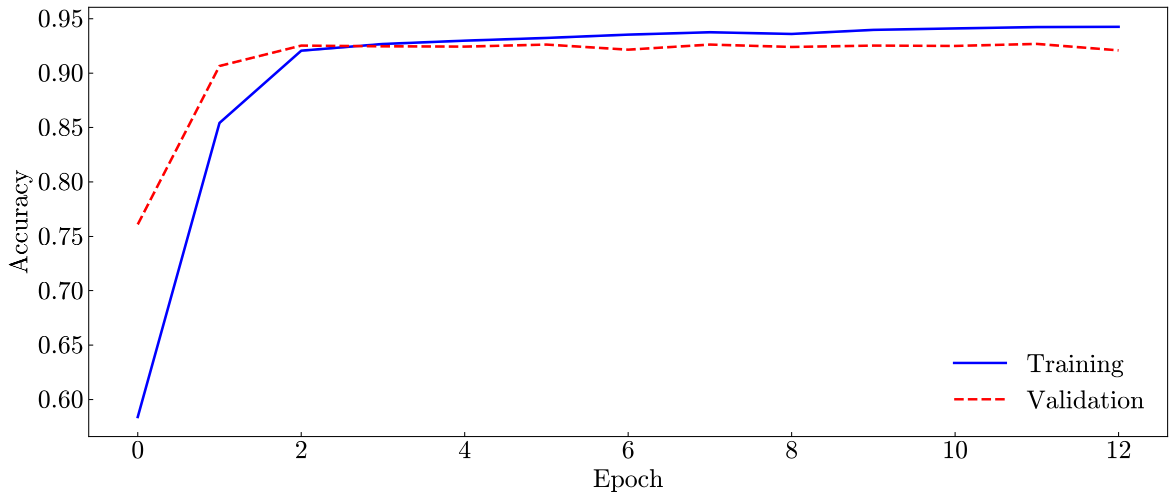
The set-up for training:

- Loss function: `softmax_cross_entropy_with_integer_labels()`
- Optimizer: `adamw()` (learning rate = $10^{-3}$, weight decay = $5 \times 10^{-4}$)
- Batch size: 256
- Maximum number of epochs: 20
- Maximum number of patience: 3

**4.** Training stopped at 13th epoch as the early stopping criterion was met.



(a)



(b)

Figure 4: (a) Training and validation loss, and (b) training and validation accuracy of the model.

It is apparent from fig. 4 that the model started to overfit very mildly when the early stopping criterion (current validation loss $<$ best validation loss $-10^{-6}$) was activated, and training was stopped. The validation loss plateaus quickly within 3 epochs, and does not really change afterwards. The accuracy of the validation set also points to that. Overall, the model generalizes well with unprevailing tendency of overfitting.

**5.** Test accuracy (threshold $\hat{\rho} = 0.5$): 0.9307500720024109 (final test loss: 0.1804021592993191)

## E. Quantitative comparison: CNN vs matched filter

**1.** Please refer to listing 1.

**2.** AUC (matched filter) = 0.9840, AUC (CNN) = 0.9803.

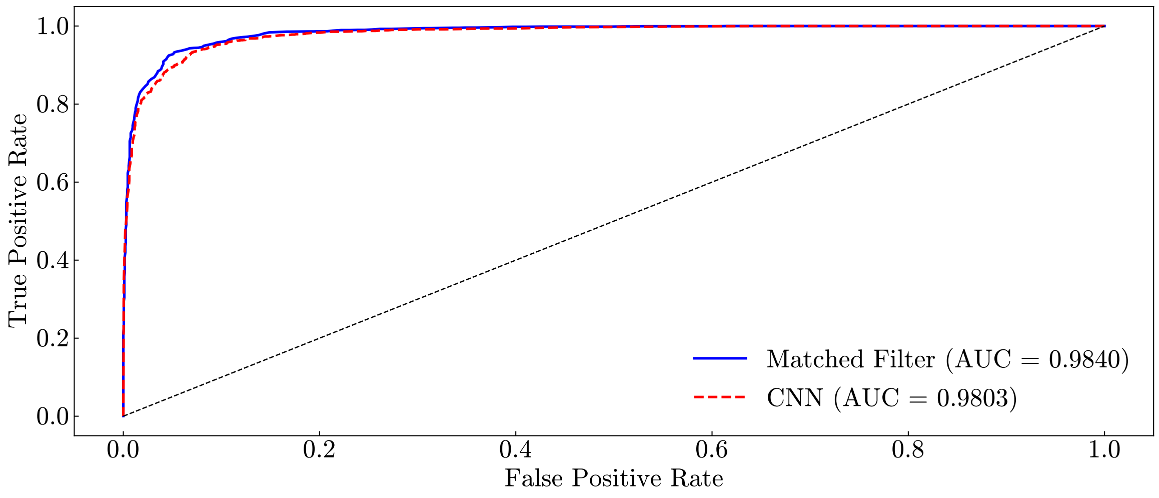


Figure 5: ROC curve for matched filter and CNN.

**3.** Discussions:

- CNN's ROC curve is very close to the ROC curve of matched filter. At very low FPR, matched filter has a higher peak compared to CNN, which is expected since matched filter is theoretically optimal. That is the only reason for AUC of CNN becoming 0.0037 less than matched filter's AUC. Overall, CNN impose almost the same level of performance as matched filter.
- At a few extremely low FPR values, CNN seems to have higher TPR than matched filter, but that is almost insignificant because it is more likely because of finite sampling noise rather than true outperformance. This is not surprising that matched filter proves to be superior as already mentioned that it is the optimal linear detector.

## F. Going Further

- **Interpreting the learned convolutional filters**

  **1.** From fig. 6, it is clear that none of the filters has any pattern that mimics template $\hat{s}$ in terms of phase, period, and even amplitude. They look noisy and irregular, and does not look like smooth sinusoid. Filter 1 has the best correlation with the template, but the value of correlation is 0.3696 which is low enough to conclude that it did not rediscover a sinusoidal matched filter shape. The bottomline is that CNN seems to learn something different than matched filter despite having a performance that is very close to it.

  **2.** CNN's learned filters are substantially different, since none of the kernels matches with template $\hat{s}$. So, CNN did not reproduce the optimal linear filter in Conv1. It seems

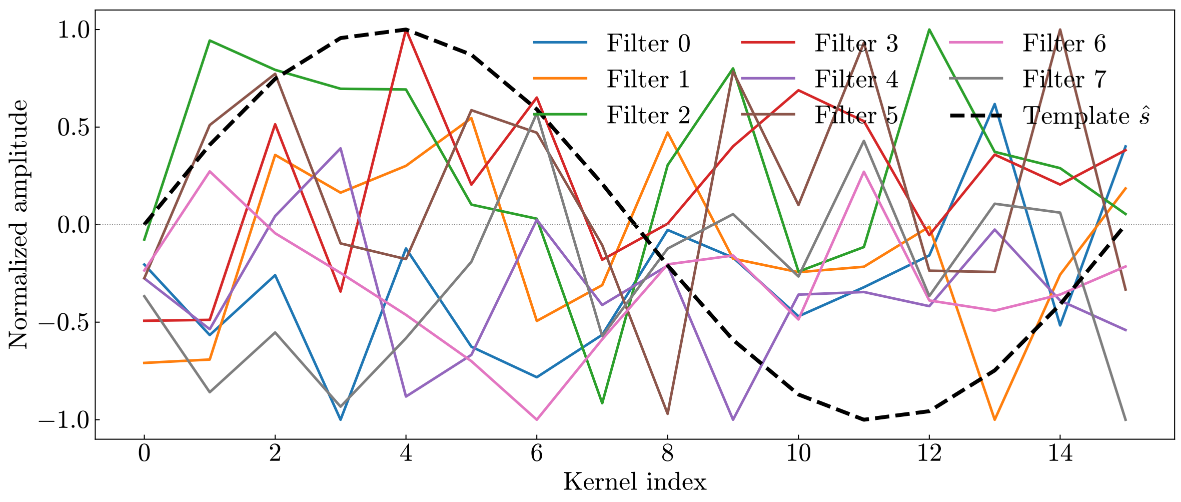
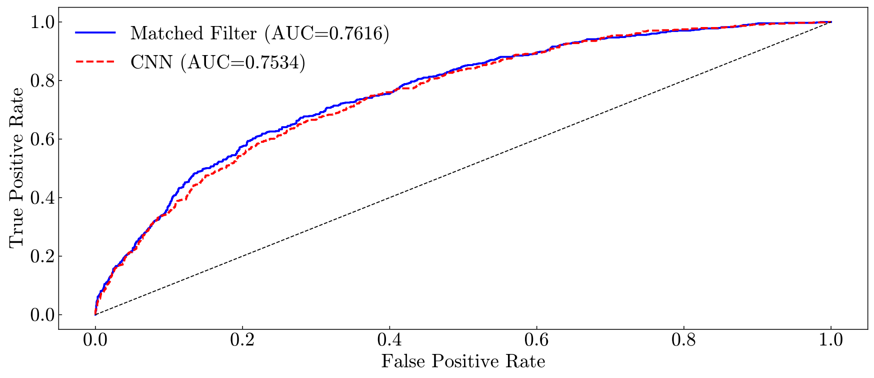


Figure 6: Learned 1D kernels and template $\hat{s}$ as a function of time index.

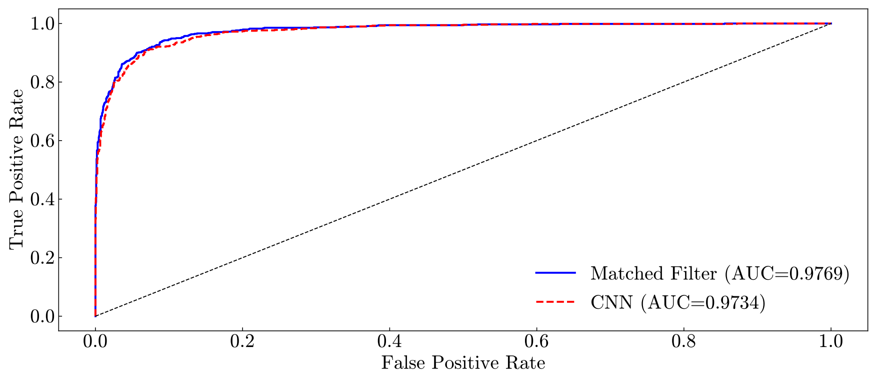
to pick on different features, likely constructing its classification rules through the non-linear combination of local features and dense layers.

- **Different signal strengths**

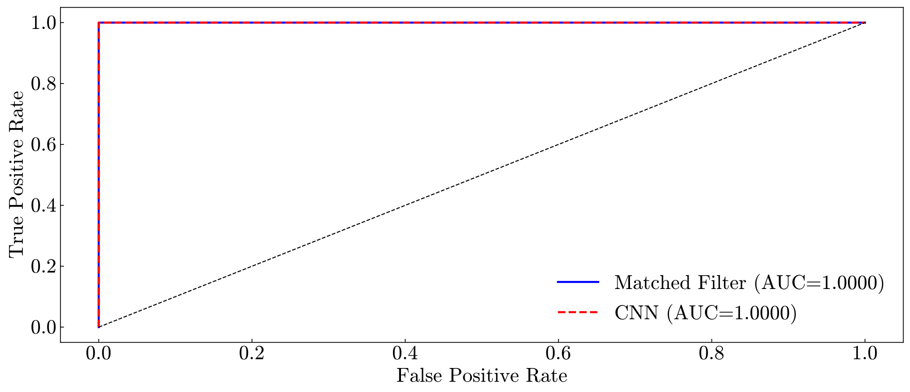
  **1.** Please refer to listing 1.



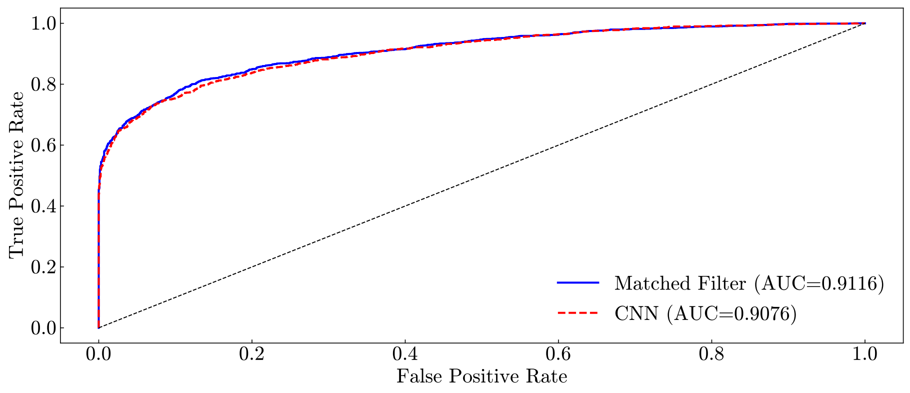
(a)



(b)



(c)



(d)

Figure 7: (a) ROC curves of matched filter and CNN for (a) $A = 1$, (b) $A = 3$, (c) $A = 10$, and (d) entire dataset.

**2.** From fig. 7, it is clear that matched filter remains superior as a class detector over CNN across all the amplitudes. However, the performance of CNN is very close to matched filter. One thing to note is that CNN can locally perform slightly better than matched filter as evident from fig. 7a and 7d clearly. Listing 2 shows a detailed statistics of the comparison between matched filter and CNN for all the amplitudes considered here.

```python
import numpy as np
import jax
jax.config.update("jax_enable_x64", True)
import jax.numpy as jnp
import jax.random as random
import optax
from flax import linen as nn
from flax.core.frozen_dict import freeze, unfreeze
#from flax import nnx
from sklearn.model_selection import train_test_split
from sklearn.metrics import roc_curve, auc
import matplotlib.pyplot as plt
import matplotlib as mpl
from scipy.stats import norm

# parameters for plotting
plt.rcParams['font.family'] = 'serif'
plt.rcParams['font.serif'] = 'cmr10'
plt.rcParams['mathtext.fontset'] = 'cm'
plt.rcParams['font.size'] = 20
mpl.rcParams['axes.unicode_minus'] = False
plt.rcParams['axes.formatter.use_mathtext'] = True

def generate_data(A, N=1000, seed=None):
    """Generate a noisy time series on {t_i, y_i}_{i=1}^N on [0, 2pi].."""
    rng = np.random.default_rng(seed)
    t = np.linspace(0.0, 2*np.pi, N)
    dt = (2*np.pi) / (N - 1)
    s_hat = np.sqrt(dt / np.pi) * np.sin(t)    # ||s_hat||_2 = 1
    s = A * s_hat
    n = rng.normal(loc=0.0, scale=1.0, size=N)
    y = s + n
    return t, y, s, n, s_hat, dt

def rho(y, s_hat):
    """detection statistics"""
    return np.dot(y, s_hat)

def sample_rhos(A, M, N=1000):
    """draw M realizations of y at amplitude A and return rhos."""
    rhos = np.zeros(M)
    # s_hat is independent of A, so just grab it once
    _, _, _, _, s_hat, _ = generate_data(A, N=N, seed=0)
    for k in range(M):
        _, y, _, _, _, _ = generate_data(A, N=N, seed=k)
        rhos[k] = rho(y, s_hat)
    return rhos, s_hat

# ===== B. Matched-filter recap and extended study =====
print("\n===== B. Matched-filter recap and extended study =====\n")
print("\n1. Generation of independent realizations of y:\n")
M = 20000
rhos_A0, s_hat = sample_rhos(A=0, M=M)
rhos_A3, _     = sample_rhos(A=3, M=M)

# plot for A=0
fig, ax = plt.subplots(figsize=(15, 6))
ax.hist(rhos_A0, bins=40, density=True, alpha=0.5, color='blue', edgecolor='black'
    , label='Empirical')
xs = np.linspace(-5, 5, 500)
ax.plot(xs, norm.pdf(xs, loc=0, scale=1), 'r-', label=rf'$\mathcal{{N}}\,(0,\,1)$'
    )
plt.xlabel(r"$\rho$")
plt.ylabel("Density")
plt.legend(loc="upper right", frameon=False)
plt.tick_params(axis="both", which="both", direction="in")
plt.savefig("a0_pdf.pdf", dpi=1080)
plt.show()

# plot for A=3
fig, ax = plt.subplots(figsize=(15, 6))
ax.hist(rhos_A3, bins=40, density=True, alpha=0.5, color='blue', edgecolor='black'
    , label='Empirical')
xs = np.linspace(-2, 8, 500)
ax.plot(xs, norm.pdf(xs, loc=3, scale=1), 'r-', label=rf'$\mathcal{{N}}\,(3,\,1)$'
    )
plt.xlabel(r"$\rho$")
plt.ylabel("Density")
plt.legend(loc="upper right", frameon=False)
plt.tick_params(axis="both", which="both", direction="in")
plt.savefig("a3_pdf.pdf", dpi=1080)
plt.show()

print("\n2. Creating filter_classifier and its verification:\n")
def filter_classifier(y, s_hat, rho0):
    rho_val = np.dot(y, s_hat)
    return int(rho_val >= rho0)

# get s_hat (does not depend on A)
t, _, _, _, s_hat, _ = generate_data(A=0, seed=0)

rho0 = 1.5 # threshold

# build dataset to predict class
def build_dataset(M=5000, N=1000, seed=0):
    rng = np.random.default_rng(seed)
    X = []
    labels = []

    for _ in range(M):
        c = rng.integers(0, 2)      # 0 or 1
        A = 0 if c == 0 else 3
        _, y, _, _, _, _ = generate_data(A, N=N, seed=rng.integers(1e9))
        X.append(y)
        labels.append(c)

    return np.array(X, dtype=np.float64), np.array(labels, dtype=np.int32)

X, labels = build_dataset(M=5000, N=1000, seed=0)

rhos = X @ s_hat
preds_theory = (rhos >= rho0).astype(int)

# compute classifier outputs
preds_func = np.array([filter_classifier(y, s_hat, rho0) for y in X])

print("All equal?", np.all(preds_theory == preds_func))
```

```python
print("Mismatches:", np.sum(preds_theory != preds_func))

# plot for predicted class
fig, ax = plt.subplots(figsize=(15, 6))
ax.scatter(rhos, preds_func, s=30, color='blue', edgecolors='black', label='
    Classifier output')
ax.axvline(rho0, color='red', linestyle='--', linewidth=2, label=rf'$\rho_0 = {
    rho0}$')
plt.xlabel(r"$\rho$")
plt.ylabel("Predicted class")
plt.yticks([0,1])
plt.legend(frameon=False)
plt.tick_params(axis="both", which="both", direction="in")
plt.savefig("pred_class.pdf", dpi=1080)
plt.show()

# plot for PDF of predicted class
fig, ax = plt.subplots(figsize=(15, 6))
ax.hist(rhos[preds_func == 0], bins=40, density=True, alpha=0.5, edgecolor='black'
    , color='blue', label='Predicted class 0')
ax.hist(rhos[preds_func == 1], bins=40, density=True, alpha=0.5, edgecolor='black'
    , color='orange', label='Predicted class 1')
ax.axvline(rho0, color='red', linestyle='--', lw=2, label=rf'$\rho_0 = {rho0}$')
plt.xlabel(r"$\rho$")
plt.ylabel("Density")
plt.legend(frameon=False)
plt.tick_params(axis="both", which="both", direction="in")
plt.savefig("pred_class_pdf.pdf", dpi=1080)
plt.show()

# plot comparing theoretical prediction vs. function predictions
fig, ax = plt.subplots(figsize=(15, 6))
ax.scatter(np.arange(200), preds_theory[:200], facecolors='none', edgecolors='blue
    ', s=60, linewidths=1.5, label='Theory')
ax.scatter(np.arange(200), preds_func[:200], c='red', marker='x', s=60, linewidths
    =1.5, label='Function')
plt.xlabel("Sample index")
plt.yticks([0, 1])
plt.ylabel("Class")
plt.legend(frameon=False)
plt.tick_params(axis="both", which="both", direction="in")
plt.savefig("pred_class_comp.pdf", dpi=1080)
plt.show()

# ===== C. Building a labeled dataset for a CNN =====
print("\n===== C. Building a labeled dataset for a CNN =====\n")
def build_dataset(M, N, seed=0):
    rng = np.random.default_rng(seed)
    X = []
    labels = []

    for k in range(M):
        c = rng.integers(0, 2)        # 0 = no signal, 1 = signal
        A = 0 if c == 0 else 3
        _, y, _, _, _, _ = generate_data(A, N=N, seed=rng.integers(1e9))
        X.append(y)
        labels.append(c)

    X = np.array(X, dtype=np.float64)
    labels = np.array(labels, dtype=np.int32)
    return X, labels

print("\n1. Construction of a label dataset of time series:\n")
N = 1000
M = 20000
X, c = build_dataset(M=M, N=N, seed=0)

print(f"Class balance: {np.sum(c==0)} no-signal, {np.sum(c==1)} signal")
print(f"Percentage signal: {100*np.mean(c):.3f}%")

print("\n2. Split of dataset:\n")
# 80-20 split
X_train, X_test, c_train, c_test = train_test_split(
    X, c,
    test_size=0.20,
    stratify=c,
    random_state=0
)

print("Train:", X_train.shape, c_train.shape)
print("Test :", X_test.shape,  c_test.shape)

print(f"Train class balance: {100*np.mean(c_train):.3f}% signal")
print(f"Test class balance: {100*np.mean(c_test):.3f}% signal")

# convert to JAX array
X_train = jnp.array(X_train, dtype=jnp.float64)
X_test  = jnp.array(X_test,  dtype=jnp.float64)
c_train = jnp.array(c_train)
c_test  = jnp.array(c_test)

# ===== Design and training of a 1D CNN classifier =====
print("\n===== D. Design and training of a 1D CNN classifier =====\n")

# 1D CNN using flax linen
class CNN1D(nn.Module):
    N: int

    @nn.compact
    def __call__(self, x):
        # x is (B, N, 1)

        # convolutional layer
        x = nn.Conv(features=8, kernel_size=(16,), padding="SAME")(x)

        # relu
        x = nn.relu(x)

        # max pooling
        x = nn.max_pool(x, window_shape=(4,), strides=(4,))

        # flattern
        x = x.reshape((x.shape[0], -1))  # (B, 2000)

        # one dense layer
        x = nn.Dense(32)(x)
        x = nn.relu(x)

        # final output layer
        logits = nn.Dense(2)(x)

        return logits

print("\n1. 1D CNN architecture parameter counts:\n")
# functions for trainable parameter count
def count_params(params):
    return sum(p.size for p in jax.tree_util.tree_leaves(params))

def detailed_param_count(params):
    result = {}

    def traverse(tree, parent=""):
        for key, val in tree.items():
            name = f"{parent}/{key}" if parent else key
            if isinstance(val, dict):
                traverse(val, name)
            else:
                result.setdefault(parent, 0)
                result[parent] += val.size

    traverse(unfreeze(params))
    return result

# initialize model and parameters
rng = jax.random.PRNGKey(0)
dummy = jnp.zeros((1, 1000, 1))   # NOTE: must be (B, N, 1)

model = CNN1D(N=1000)
params = model.init(rng, dummy)

# parameter counts
total = count_params(params)
print("Total parameters:", total)

details = detailed_param_count(params)
for layer, n in details.items():
    print(f"{layer:20s} : {n}")

print("\n2. Normalize and reshape the data:\n")
# normalize the data
X_train_np = np.array(X_train)
X_test_np  = np.array(X_test)

mean = X_train_np.mean()
std  = X_train_np.std()

X_train_np = (X_train_np - mean) / std
X_test_np  = (X_test_np  - mean) / std

# reshape for Conv1D
X_train_np = X_train_np[:, :, None]
X_test_np  = X_test_np[:, :, None]

print("\n3. Split for validation and training:\n")
# 80-20 train-validation split
X_tr_np, X_val_np, c_tr_np, c_val_np = train_test_split(
        X_train_np, np.array(c_train), test_size=0.2,
        stratify=np.array(c_train), random_state=1)

# convert to JAX arrays
X_tr  = jnp.array(X_tr_np)
X_val = jnp.array(X_val_np)
X_te  = jnp.array(X_test_np)

c_tr  = jnp.array(c_tr_np)
c_val = jnp.array(c_val_np)
c_te  = jnp.array(c_test)

print("\nTraining the 1D CNN:\n")

def loss_fn(params, batch_x, batch_y):
    logits = model.apply(params, batch_x)
    loss = optax.softmax_cross_entropy_with_integer_labels(logits, batch_y).mean()
    return loss, logits

def accuracy(logits, labels):
    preds = jnp.argmax(logits, axis=-1)
    return jnp.mean(preds == labels)

optimizer = optax.adamw(learning_rate=1e-3, weight_decay=5e-4)

@jax.jit
def train_step(params, opt_state, xb, yb):
    (loss, logits), grads = jax.value_and_grad(loss_fn, has_aux=True)(
        params, xb, yb
    )
    updates, opt_state = optimizer.update(grads, opt_state, params)
    params = optax.apply_updates(params, updates)
    acc = accuracy(logits, yb)
    return params, opt_state, loss, acc


@jax.jit
def eval_step(params, xb, yb):
    loss, logits = loss_fn(params, xb, yb)
    acc = accuracy(logits, yb)
    return loss, acc

# initialize the model and optimizer
model = CNN1D(N=N)
rng = jax.random.PRNGKey(23)
params = model.init(rng, X_tr[:8])   # dummy batch for shapes
opt_state = optimizer.init(params)

# training params
batch_size = 256   # "None" for full batch training
epochs = 20

train_losses = []
train_accs = []
val_losses = []
val_accs = []

patience = 3
best_val_loss = jnp.inf
patience_counter = 0
```

```python
params_best = None

# training loop
for ep in range(epochs):

    # reproducible shuffle
    rng, sub = jax.random.split(rng)
    perm = jax.random.permutation(sub, X_tr.shape[0])
    X_shuf = X_tr[perm]
    c_shuf = c_tr[perm]

    # batching mode
    if batch_size is None:
        """full-batch"""
        xb = X_shuf
        yb = c_shuf

        params, opt_state, loss, acc = train_step(params, opt_state, xb, yb)

        ep_loss = float(loss)
        ep_acc  = float(acc)

    else:
        """mini-batch"""
        num_batches = X_shuf.shape[0] // batch_size

        ep_loss = 0.0
        ep_acc  = 0.0

        for i in range(num_batches):
            xb = X_shuf[i*batch_size:(i+1)*batch_size]
            yb = c_shuf[i*batch_size:(i+1)*batch_size]

            params, opt_state, loss, acc = train_step(
                params, opt_state, xb, yb
            )
            ep_loss += float(loss)
            ep_acc  += float(acc)

        ep_loss /= num_batches
        ep_acc  /= num_batches

    # validation
    val_loss, val_acc = eval_step(params, X_val, c_val)

    train_losses.append(ep_loss)
    train_accs.append(ep_acc)
    val_losses.append(float(val_loss))
    val_accs.append(float(val_acc))

    print(f"Epoch {ep+1:2d} | "
          f"train_loss={ep_loss:.4f}, train_acc={ep_acc:.4f} | "
          f"val_loss={float(val_loss):.4f}, val_acc={float(val_acc):.4f}")

    # early stopping check
    if float(val_loss) < float(best_val_loss) - 1e-6:
        best_val_loss = float(val_loss)
        params_best = params
        patience_counter = 0
    else:
        patience_counter += 1

    if patience_counter >= patience:
        print(f"\nEarly stopping triggered at epoch {ep+1}")
        params = params_best
        break

# final test loss and accuracy
test_loss, test_acc = eval_step(params, X_te, c_te)
print("\nTest Loss:", float(test_loss))
print("Test Accuracy:", float(test_acc))

print("\n4. Plot training and validation loss:\n")
fig, ax = plt.subplots(figsize=(15, 6))
ax.plot(train_losses, "b-", lw=2, label="Training")
ax.plot(val_losses, "r--", lw=2, label="Validation")
plt.xlabel("Epoch")
plt.ylabel("Loss")
ax.legend(frameon=False)
ax.tick_params(axis="both", which="both", direction="in")
plt.savefig("cnn_loss.pdf", dpi=1080)
plt.show()

fig, ax = plt.subplots(figsize=(15, 6))
ax.plot(train_accs, "b-", lw=2, label="Training")
ax.plot(val_accs, "r--", lw=2, label="Validation")
plt.xlabel("Epoch")
plt.ylabel("Accuracy")
plt.legend(frameon=False)
plt.tick_params(axis="both", which="both", direction="in")
plt.savefig("cnn_acc.pdf", dpi=1080)
plt.show()

print("\n5. CNN's performance on the held out test set:\n")
test_logits = model.apply(params, X_te)
test_probs  = jax.nn.softmax(test_logits, axis=-1)[:, 1]
test_preds  = (test_probs >= 0.5).astype(int)

test_acc = jnp.mean(test_preds == c_te)

print("\nTest accuracy:", float(test_acc))

# ===== E. Quantitative comparison: CNN vs matched filter =====
print("\n===== E. Quantitative comparison: CNN vs matched filter =====\n")

print("\n1. Construct ROC:\n")

# matched-filter ROC
X_te_flat = np.array(X_te[..., 0])

rho_vals = X_te_flat @ s_hat

rho_range = np.linspace(rho_vals.min(), rho_vals.max(), 400)
tpr_mf = []
fpr_mf = []

for rho0 in rho_range:
    preds = (rho_vals >= rho0).astype(int)

    tp = np.sum((preds == 1) & (c_te == 1))
    fp = np.sum((preds == 1) & (c_te == 0))
    fn = np.sum((preds == 0) & (c_te == 1))
    tn = np.sum((preds == 0) & (c_te == 0))

    tpr_mf.append(tp / (tp + fn))
    fpr_mf.append(fp / (fp + tn))

auc_mf = auc(fpr_mf, tpr_mf)
"""
fpr_mf, tpr_mf, mf_thresholds = roc_curve(c_te, rho_vals)
auc_mf = auc(fpr_mf, tpr_mf)
"""
# CNN ROC
"""
def predict_cnn_proba(params, X):
    logits = model.apply(params, X)
    probs = jax.nn.softmax(logits, axis=-1)
    return np.array(probs[:, 1])   # probability of class "1" (signal)

cnn_probs = predict_cnn_proba(params, X_te)

fpr_cnn, tpr_cnn, cnn_thresholds = roc_curve(c_te, cnn_probs)
auc_cnn = auc(fpr_cnn, tpr_cnn)
"""
logits_te = model.apply(params, X_te)
probs = jax.nn.softmax(logits_te, axis=-1)
p_hat = np.array(probs[:, 1])

p_range = np.linspace(0, 1, 400)
tpr_cnn, fpr_cnn = [], []

for p0 in p_range:
    preds = (p_hat >= p0).astype(int)

    tp = np.sum((preds == 1) & (c_te == 1))
    fp = np.sum((preds == 1) & (c_te == 0))
    tn = np.sum((preds == 0) & (c_te == 0))
    fn = np.sum((preds == 0) & (c_te == 1))

    tpr_cnn.append(tp / (tp + fn))
    fpr_cnn.append(fp / (fp + tn))

print("\n2. Plot ROC:\n")

fig, ax = plt.subplots(figsize=(15, 6))
ax.plot(fpr_mf, tpr_mf, 'b-', lw=2, label=f"Matched Filter (AUC = {auc_mf:.4f})")
ax.plot(fpr_cnn, tpr_cnn, 'r--', lw=2, label=f"CNN (AUC = {auc_cnn:.4f})")
ax.plot([0, 1], [0, 1], 'k--', lw=1)
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.legend(frameon=False)
plt.tick_params(axis="both", which="both", direction="in")
plt.savefig("roc.pdf", dpi=1080)
plt.show()

# ===== F. Going further =====
print("\n===== F. Going further =====\n")
```

```python

print("\nInterpreting the learned convolutional filters:\n")

p = params['params']

print("\nAvailable layers:", list(p.keys()))

# first convolutional layer
k1 = np.array(p['Conv_0']['kernel'])
print(f"Conv_0 kernel shape: {k1.shape}\n")
K1 = k1.shape[0] # kernel size
n_filters_1 = k1.shape[2] # number of learned filters

# extract each filter (input channel = 1)
kernels_conv1 = [k1[:, 0, j] for j in range(n_filters_1)]

# construct the known matched-filter template at same resolution
t_kernel = np.linspace(0, 2*np.pi, K1)
dt_kernel = 2*np.pi / (K1 - 1)
s_hat_kernel = np.sqrt(dt_kernel / np.pi) * np.sin(t_kernel)

# normalize for plotting
s_hat_norm = s_hat_kernel / np.max(np.abs(s_hat_kernel))

# plot for learned kernel and $\hat{s}$
fig, ax = plt.subplots(figsize=(15, 6))
for j, k_j in enumerate(kernels_conv1):
    k_norm = k_j / (np.max(np.abs(k_j)) + 1e-12)
    ax.plot(k_norm, lw=2, label=f"Filter {j}")
ax.plot(s_hat_norm, 'k--', lw=3, label="Template $\hat{s}$")
ax.axhline(0, color='gray', linestyle=':', linewidth=0.7)
plt.xlabel("Kernel index")
plt.ylabel("Normalized amplitude")
plt.legend(ncol=3, frameon=False)
plt.tick_params(axis="both", which="both", direction="in")
plt.savefig("comp_kernel.pdf", dpi=1080)
plt.show()

# correlation between learned filters and $\hat{s}$
s_norm = s_hat_kernel / np.linalg.norm(s_hat_kernel)

correlations = []
for j, k_j in enumerate(kernels_conv1):
    k_norm = k_j / (np.linalg.norm(k_j) + 1e-12)
    corr = np.dot(k_norm, s_norm)
    correlations.append(corr)
    print(f"Filter {j}: correlation = {corr:+.4f}")

best_idx = np.argmax(np.abs(correlations))
best_corr = correlations[best_idx]

print(f"\nBest matching filter: Filter {best_idx}")
print(f"Absolute correlation: {abs(best_corr):.4f}")

print("\nDifferent signal strengths:\n")

# function for multi-amplitude dataset
def build_multiamp_dataset(M=20000, N=1000, seed=0):
    rng = np.random.default_rng(seed)
    X = []
    labels = []
    amplitudes = []

    for k in range(M):
        has_signal = rng.integers(0, 2)  # 50-50 split

        if has_signal == 0:
            A = 0
            label = 0  # no signal
        else:
            A = rng.choice([1, 3, 10])  # random signal strength
            label = 1  # signal present (A >= 1)

        _, y, _, _, _, _ = generate_data(A, N=N, seed=rng.integers(int(1e9)))
        X.append(y)
        labels.append(label)
        amplitudes.append(A)

    return np.array(X), np.array(labels), np.array(amplitudes)

X_multi, c_multi, A_multi = build_multiamp_dataset(M=20000, N=1000, seed=42)

# check distribution
print(f"\nDataset distribution:")
print(f"A=0: {np.sum(A_multi == 0)} ({100*np.mean(A_multi == 0):.1f}%)")
print(f"A=1: {np.sum(A_multi == 1)} ({100*np.mean(A_multi == 1):.1f}%)")
print(f"A=3: {np.sum(A_multi == 3)} ({100*np.mean(A_multi == 3):.1f}%)")
print(f"A=10: {np.sum(A_multi == 10)} ({100*np.mean(A_multi == 10):.1f}%)")

# 80-20 split
X_train_ma, X_test_ma, c_train_ma, c_test_ma, A_train_ma, A_test_ma =
    train_test_split(
    X_multi, c_multi, A_multi,
    test_size=0.2,
    stratify=c_multi,
    random_state=0
)

# normalize test data
X_test_ma_norm = (X_test_ma - mean) / std
X_test_ma_jax = jnp.array(X_test_ma_norm, dtype=jnp.float64)[:, :, None]

# CNN predictions
cnn_logits_ma = model.apply(params, X_test_ma_jax)
cnn_probs_ma = np.array(jax.nn.softmax(cnn_logits_ma, axis=-1)[:, 1])

# matched filter predictions
rhos_test_ma = X_test_ma @ s_hat

# performance of CNN
fpr_cnn_all, tpr_cnn_all, _ = roc_curve(c_test_ma, cnn_probs_ma)
auc_cnn_all = auc(fpr_cnn_all, tpr_cnn_all)

# performance of matched filter
fpr_mf_all, tpr_mf_all, _ = roc_curve(c_test_ma, rhos_test_ma)
auc_mf_all = auc(fpr_mf_all, tpr_mf_all)

print(f"\nCNN AUC: {auc_cnn_all:.4f}")
```

```python
print(f"Matched Filter AUC: {auc_mf_all:.4f}")

fig, ax = plt.subplots(figsize=(15, 6))
ax.plot(fpr_mf_all, tpr_mf_all, 'b-', lw=2, label=f'Matched Filter (AUC={
    auc_mf_all:.4f})')
ax.plot(fpr_cnn_all, tpr_cnn_all, 'r--', lw=2, label=f'CNN (AUC={auc_cnn_all:.4f})
    ')
ax.plot([0, 1], [0, 1], 'k--', lw=1)
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.legend(frameon=False)
plt.tick_params(axis="both", which="both", direction="in")
plt.savefig("comp_allamp.pdf", dpi=1080)
plt.show()

# per-amplitude analysis
for A_val in [1, 3, 10]:
    # include both signal at this amplitude and all no-signal examples
    mask = (A_test_ma == A_val) | (A_test_ma == 0)

    print(f"\nFor A={A_val}:")
    print(f"Total samples: {mask.sum()}")
    print(f"A=0 (no signal): {np.sum(A_test_ma[mask] == 0)}")
    print(f"A={A_val} (signal): {np.sum(A_test_ma[mask] == A_val)}")

    # CNN ROC and AUC
    fpr_cnn, tpr_cnn, _ = roc_curve(c_test_ma[mask], cnn_probs_ma[mask])
    auc_cnn = auc(fpr_cnn, tpr_cnn)

    # matched filter ROC and AUC
    fpr_mf, tpr_mf, _ = roc_curve(c_test_ma[mask], rhos_test_ma[mask])
    auc_mf = auc(fpr_mf, tpr_mf)

    print(f"CNN AUC: {auc_cnn:.4f}")
    print(f"Matched Filter AUC: {auc_mf:.4f}")
    print(f"Difference: {abs(auc_cnn - auc_mf):.4f}")

    fig, ax = plt.subplots(figsize=(15, 6))
    ax.plot(fpr_mf, tpr_mf, 'b-', lw=2, label=f'Matched Filter (AUC={auc_mf:.4f})'
        )
    ax.plot(fpr_cnn, tpr_cnn, 'r--', lw=2, label=f'CNN (AUC={auc_cnn:.4f})')
    ax.plot([0, 1], [0, 1], 'k--', lw=1)
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.legend(frameon=False)
    plt.tick_params(axis="both", which="both", direction="in")
    plt.savefig(f"comp_amp{A_val}.pdf", dpi=1080)
    plt.show()
```

Listing 1: cnn.py

```
===== B. Matched-filter recap and extended study =====

1. Generation of independent realizations of y:

2. Creating filter_classifier and its verification:

All equal? True
Mismatches: 0

===== C. Building a labeled dataset for a CNN =====

1. Construction of a label dataset of time series:

Class balance: 9933 no-signal, 10067 signal
Percentage signal: 50.335%

2. Split of dataset:

Train: (16000, 1000) (16000,)
Test : (4000, 1000) (4000,)
Train class balance: 50.337% signal
Test class balance: 50.325% signal

===== D. Design and training of a 1D CNN classifier =====

1. 1D CNN architecture parameter counts:

Total parameters: 64234
params/Conv_0        : 136
params/Dense_0       : 64032
params/Dense_1       : 66

2. Normalize and reshape the data:

3. Split for validation and training:

Training the 1D CNN:

Epoch  1 | train_loss=0.7091, train_acc=0.5839 | val_loss=0.6106, val_acc=0.7609
Epoch  2 | train_loss=0.4802, train_acc=0.8542 | val_loss=0.3214, val_acc=0.9066
Epoch  3 | train_loss=0.2378, train_acc=0.9206 | val_loss=0.2006, val_acc=0.9253
Epoch  4 | train_loss=0.1818, train_acc=0.9268 | val_loss=0.1869, val_acc=0.9247
Epoch  5 | train_loss=0.1709, train_acc=0.9298 | val_loss=0.1848, val_acc=0.9244
Epoch  6 | train_loss=0.1661, train_acc=0.9323 | val_loss=0.1843, val_acc=0.9262
Epoch  7 | train_loss=0.1619, train_acc=0.9354 | val_loss=0.1861, val_acc=0.9216
Epoch  8 | train_loss=0.1565, train_acc=0.9376 | val_loss=0.1828, val_acc=0.9262
Epoch  9 | train_loss=0.1577, train_acc=0.9360 | val_loss=0.1962, val_acc=0.9241
Epoch 10 | train_loss=0.1505, train_acc=0.9398 | val_loss=0.1827, val_acc=0.9253
Epoch 11 | train_loss=0.1484, train_acc=0.9411 | val_loss=0.1844, val_acc=0.9250
Epoch 12 | train_loss=0.1437, train_acc=0.9423 | val_loss=0.1854, val_acc=0.9269
Epoch 13 | train_loss=0.1450, train_acc=0.9426 | val_loss=0.2014, val_acc=0.9209

Early stopping triggered at epoch 13

Test Loss: 0.18040215929931913
Test Accuracy: 0.9307500720024109

4. Plot training and validation loss:
```

```

5. CNN's performance on the held out test set:

Test accuracy: 0.9307500720024109

===== E. Quantitative comparison: CNN vs matched filter =====

1. Construct ROC:

2. Plot ROC:

===== F. Going further =====

Interpreting the learned convolutional filters:

Available layers: ['Conv_0', 'Dense_0', 'Dense_1']
Conv_0 kernel shape: (16, 1, 8)

Filter 0: correlation = -0.3457
Filter 1: correlation = +0.3696
Filter 2: correlation = +0.1077
Filter 3: correlation = -0.0548
Filter 4: correlation = +0.0973
Filter 5: correlation = +0.0004
Filter 6: correlation = -0.1656
Filter 7: correlation = -0.3674

Best matching filter: Filter 1
Absolute correlation: 0.3696

Different signal strengths:

Dataset distribution:
A=0: 9955 (49.8%)
A=1: 3399 (17.0%)
A=3: 3319 (16.6%)
A=10: 3327 (16.6%)

CNN AUC: 0.9076
Matched Filter AUC: 0.9116

For A=1:
Total samples: 2670
A=0 (no signal): 1991
A=1 (signal): 679
CNN AUC: 0.7534
Matched Filter AUC: 0.7616
Difference: 0.0082

For A=3:
Total samples: 2672
A=0 (no signal): 1991
A=3 (signal): 681
CNN AUC: 0.9734
Matched Filter AUC: 0.9769
Difference: 0.0035

For A=10:
Total samples: 2640
```

```
A=0 (no signal): 1991
A=10 (signal): 649
CNN AUC: 1.0000
Matched Filter AUC: 1.0000
Difference: 0.0000
```

Listing 2: Output terminal for `cnn.py`

## II. NEURAL HAMILTONIAN FOR A 1D HARMONIC OSCILLATOR

### B. Running the starter code & dataset generation

**1.**

$$k = 1, A = 1$$

(a) $q_{num}(t)$ and $q_{exact}(t)$ are plotted in fig. 8 along with $p_{num}(t)$ and $p_{exact}(t)$.

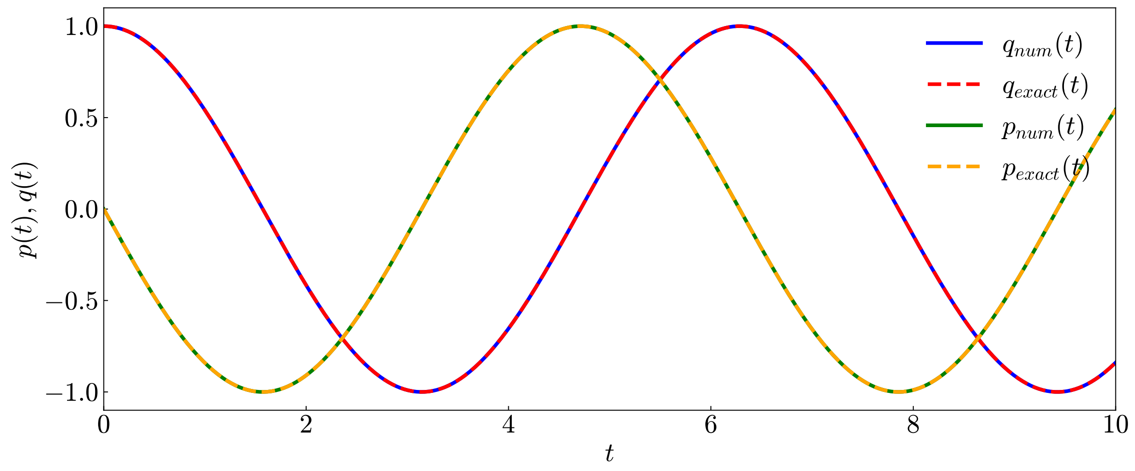


Figure 8: $q_{num}(t)$ and $q_{exact}(t)$.

They indeed look indistinguishable.

(b) Error $e(t) = q_{num}(t) - q_{exact}(t)$ is plotted in fig. 9.

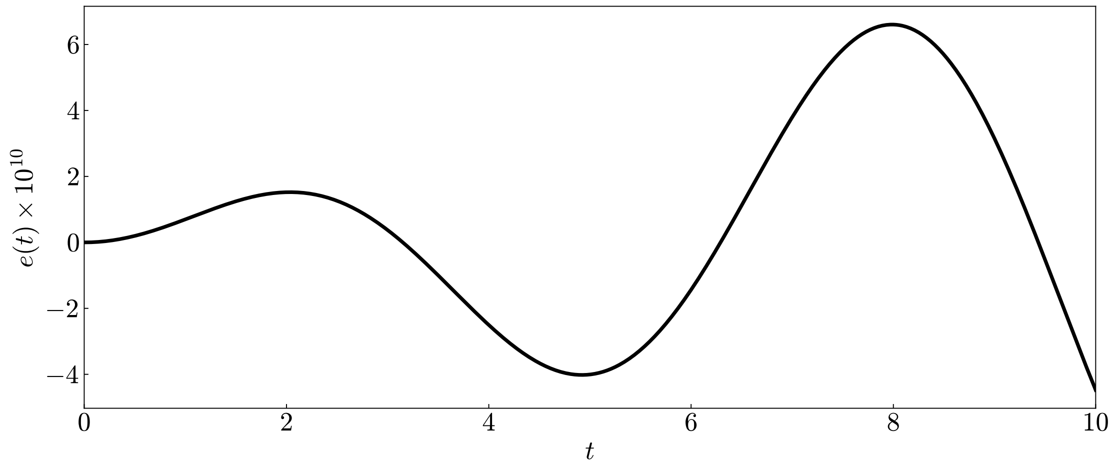


Figure 9: Error $e(t) = q_{num}(t) - q_{exact}(t)$.

The maximum absolute error as computed and reported in listing 3 and 4 respectively is $6.604\times10^{-10}$. The relative error with respect to $A = 1$ is also the same, which is very small. The difference is of course due to analytical and numerical way of solving the problem. However, the error is very tiny and insignificant.

(c) So, the lower bound of error for the best possible learned model is $6.604\times10^{-10}$.

**2.**

$$k = 1, A \in \{0.5, 1.0, 1.5, 2.0\}$$

(a) Please refer to listing 3.

(b) Please refer to listing 3. $(\dot{q}_n, \dot{p}_n)$ is obtained using analytic expressions:

$$\dot{q}_n = p_{exact}(t_n), \;\; \dot{p}_n = -kq_{exact}(t_n)$$

(c) Please refer to listing 3. Data points distribution in the 80-20 split:

$$\text{Total: } 4004, \;\; \text{Training: } 3203, \;\; \text{Validation: } 801$$

## C. Learning dynamics with a neural Hamiltonian

**(a)** Please refer to listing 3.

NN architecture:

- One hidden layer with 32 neurons.
- Activation function: `tanh`.

**(b)** Please refer to listing 3.

**(c)** Please refer to listing 3.

Hyperparameters:

- Optimizer: `lbfgs` (default settings).
- Maximum number of epochs: 500.
- Batch size: full batch.

**(d)** Fig. 10 shows that the model does not overfit and validation loss keeps decreasing smoothly with decreasing training loss. So, the model generalizes well.

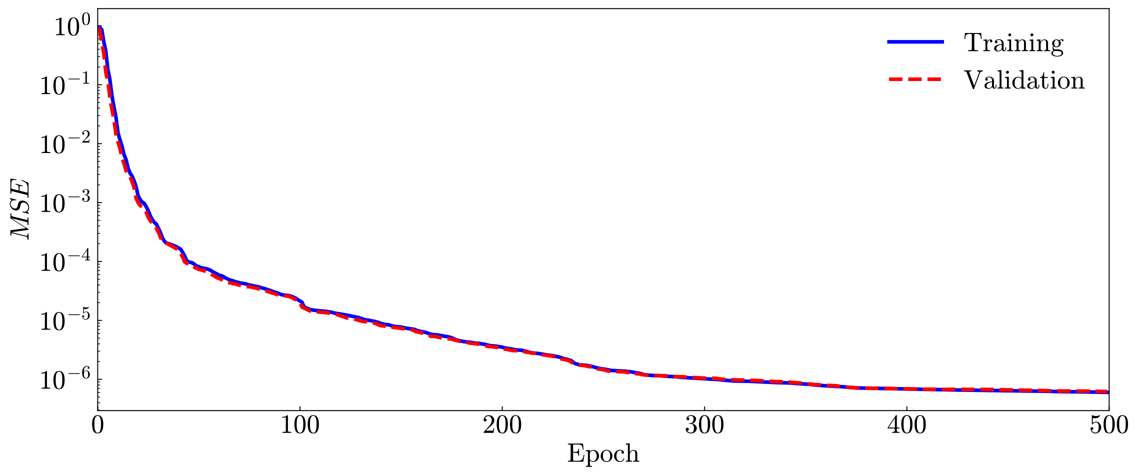
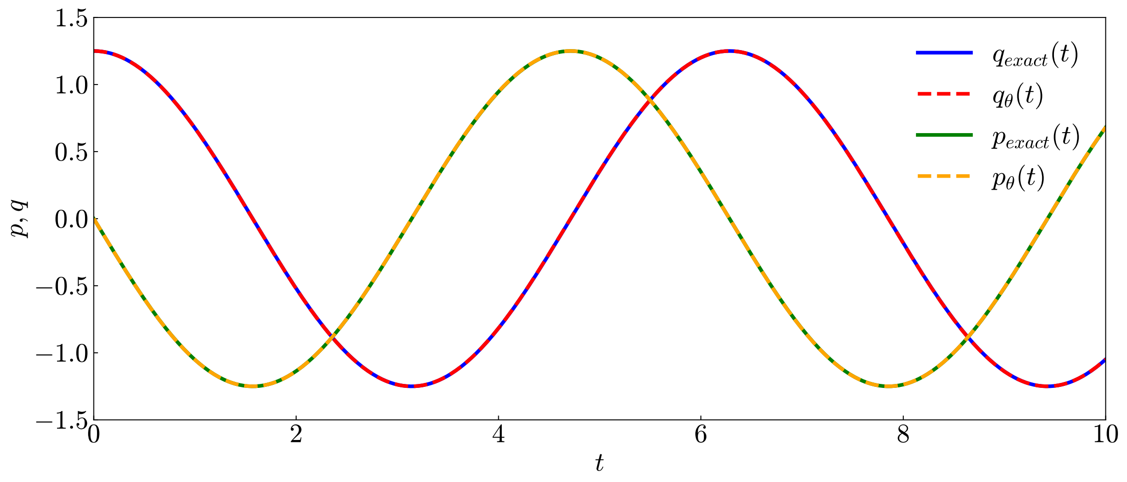


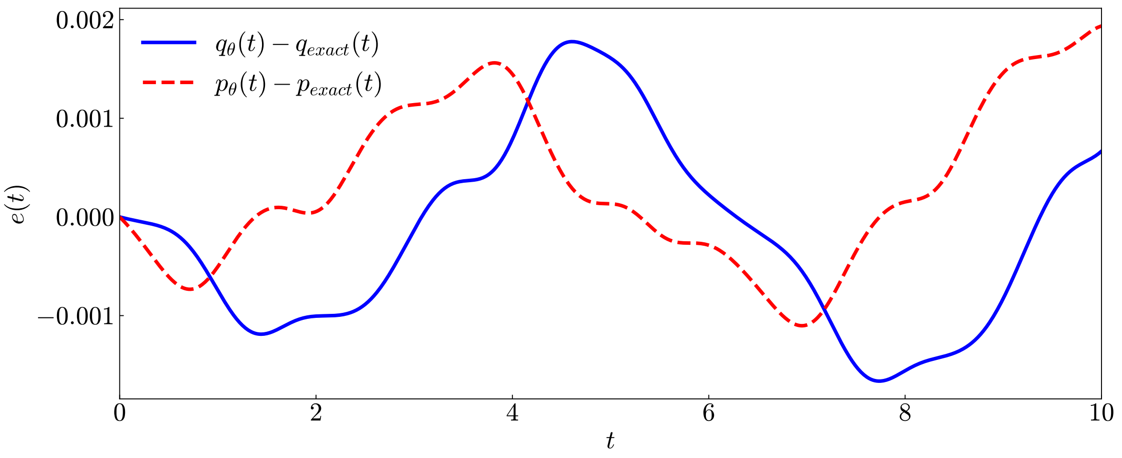
Figure 10: Training and validation losses vs. epoch.

## D. Using your new model

- $A_{test}$ is set at 1.25 for testing the model.



(a)



(b)

Figure 11: (a) Trajectories $(q_\theta, p_\theta)$ and $(q_{exact}, p_{exact})$, and (b) error between the trajectories.

Fig. 11 clearly shows that the model learns and performs really well for predicting the $H_\theta(q, p)$. The error bound between the model and exact solution is $\sim 10^{-3}$. So, the model generalizes really well, which was already evident from the training and validation losses in fig. 10.

- Clearly, the model predicts $H_\theta(q,p)$ well beyond $t = 10$ as can be seen from fig. 12. It has been run 10 times longer than it was trained for, still the error is $\sim 10^{-2}$ which further shows the fantastic generalization of the model.

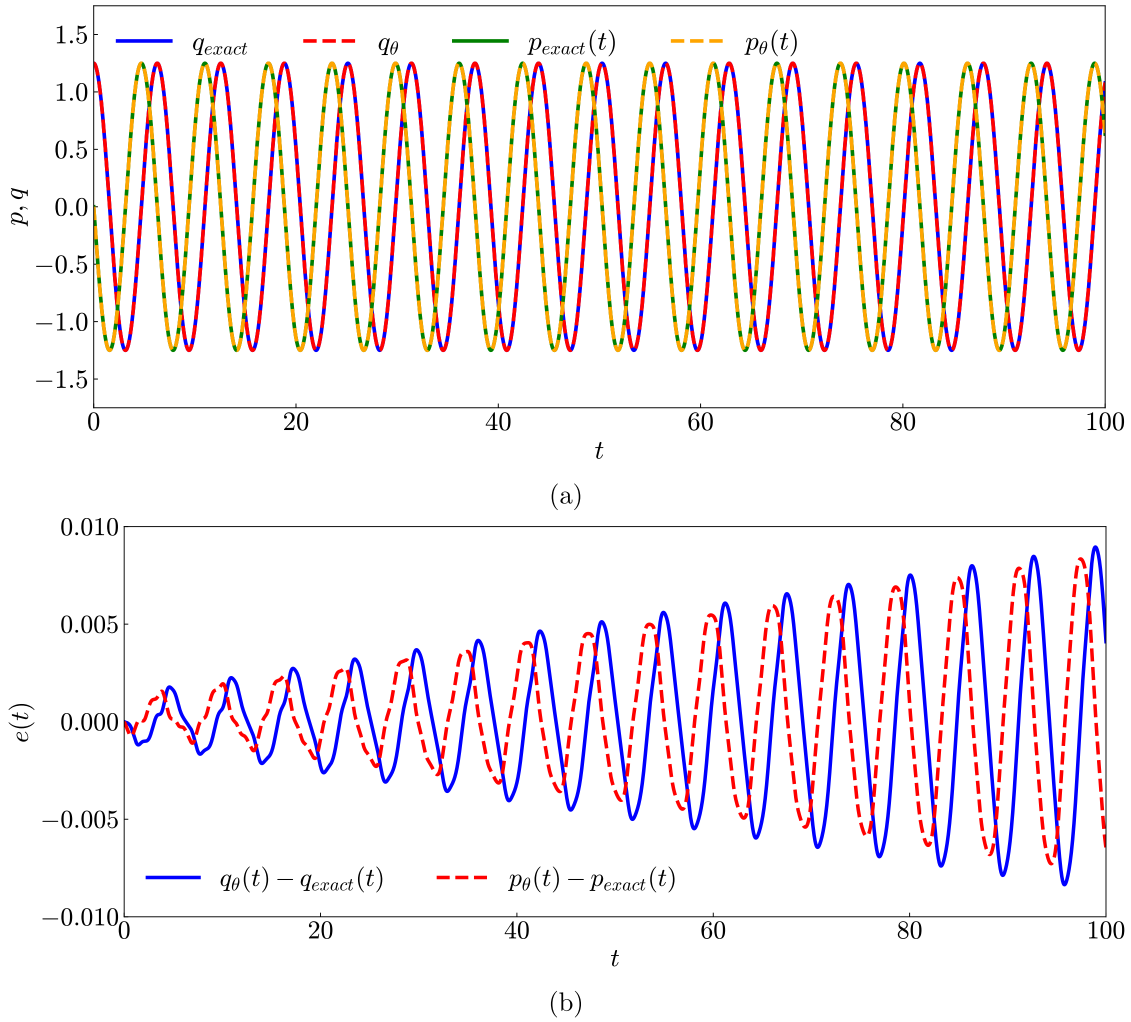


Figure 12: (a) Trajectories $(q_\theta, p_\theta)$ and $(q_{exact}, p_{exact})$, and (b) error between the trajectories extrapolated till $t = 100$.

## E. Going Further

**1.** The Hamiltonian should be conserved over time, which means

$$\frac{dH}{dt} = 0$$

This can be checked using chain rule,

$$\frac{dH}{dt} = \frac{\partial H}{\partial q}\frac{dq}{dt} + \frac{\partial H}{\partial p}\frac{dp}{dt} = -\dot{p}\dot{q} + \dot{q}\dot{p} = 0$$

So, the conservation of energy, i.e., Hamiltonian can be enforced as a physics to the model.

**2.** The loss function then can be written as,

$$L_{\text{total},\theta} = \frac{1}{N_{train}} \sum_{n=1}^{N_{train}} \|\dot{q}_\theta(q_n, p_n) - \dot{q}_n\|^2 + \|\dot{p}_\theta(q_n, p_n) - \dot{p}_n\|^2$$
$$+ \lambda \frac{1}{N_{train} - 1} \sum_{n=1}^{N_{train}-1} \|H_\theta(q_{n+1}, p_{n+1}) - H_\theta(q_n, p_n)\|^2$$

where, $\lambda (= 10^{-5})$ is the penalization hyperparameter.

**3.** The model is re-run with the same hyperparameters as the data-only case.

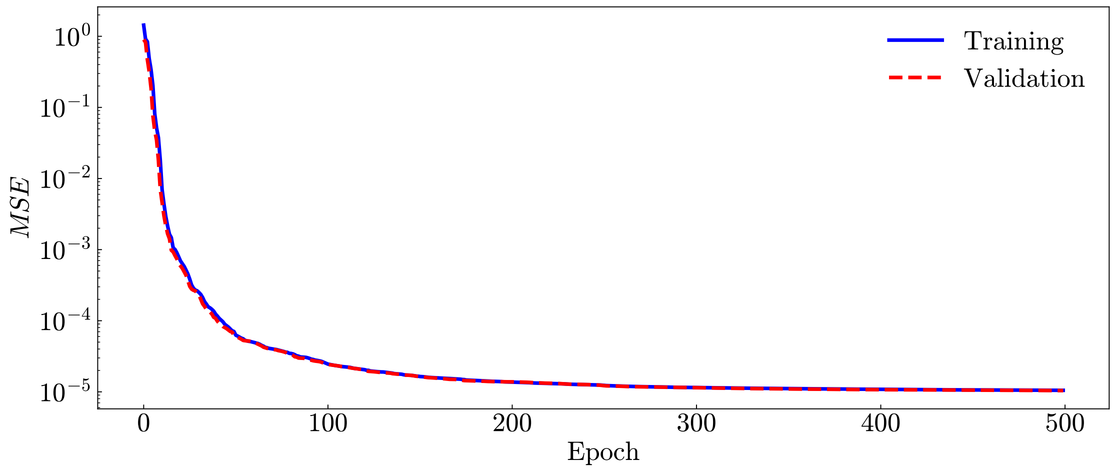
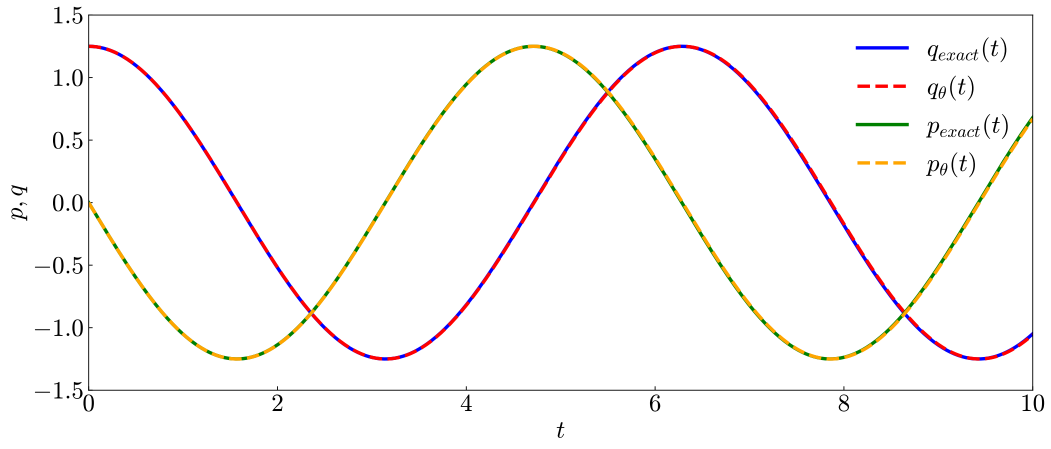


Figure 13: Training and validation losses w/ energy-penalty vs. epoch.

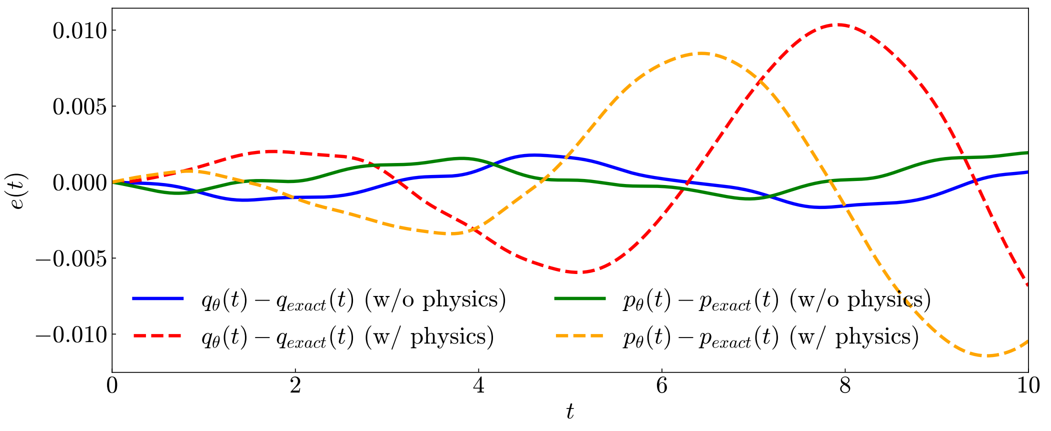
Fig. 13 shows that training and validation losses with physics incorporation are higher compared to fig. 10. Though the generalization is still good, but the loss is higher.

Fig. 14 and 15 show that the error in trajectories is higher with energy-penalty than without any penalty. The model can not perform as well as its data-only training when physics is



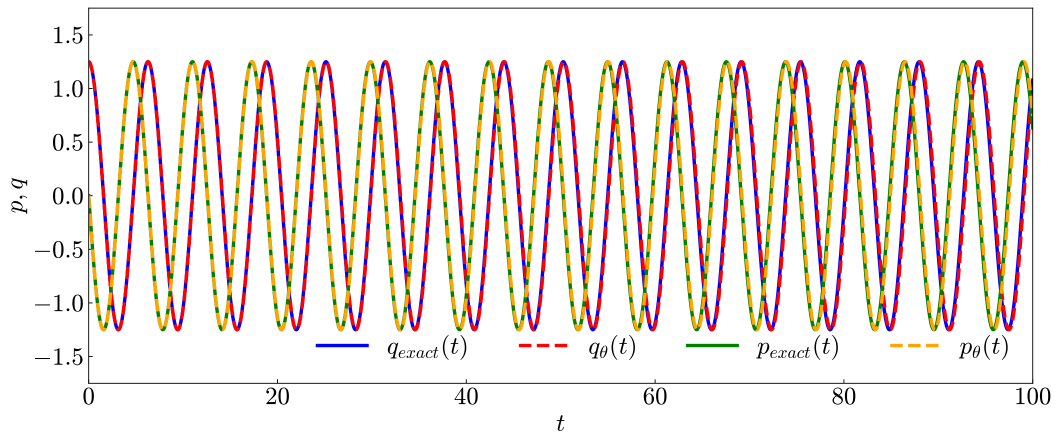
(a)

incorporated in the loss function. However, the overall generalization is still quite good.

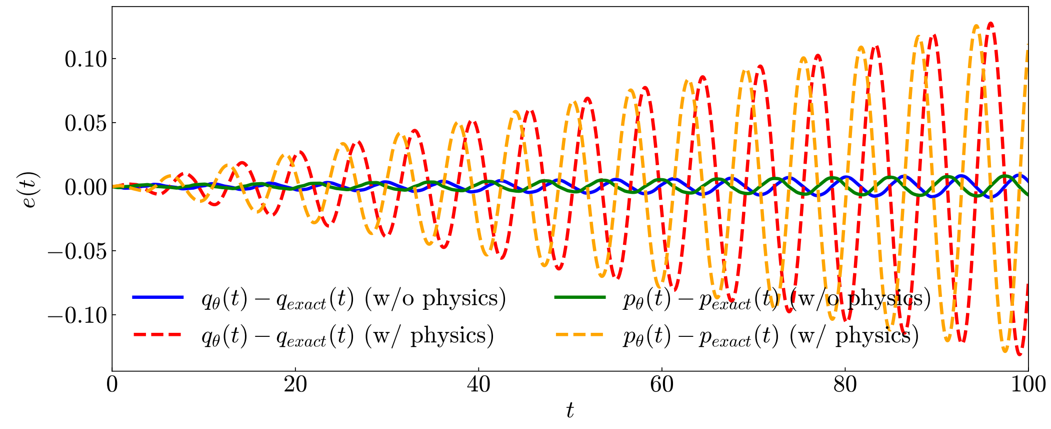


(b)

Figure 14: (a) Trajectories $(q_\theta, p_\theta)$ and $(q_{exact}, p_{exact})$, and (b) error between the trajectories w/ energy-penalty.



(a)



(b)

Figure 15: (a) Trajectories $(q_\theta, p_\theta)$ and $(q_{exact}, p_{exact})$, and (b) error between the trajectories extrapolated till $t = 100$ w/ energy-penalty.

```python
import numpy as np
import matplotlib.pyplot as plt
import matplotlib as mpl
import jax
jax.config.update("jax_enable_x64", True)
import optax
import jax.random as random
from flax import linen as nn
import jax.numpy as jnp
from typing import Sequence

# ===== B. Running the starter code & dataset generation =====
print("\n===== B. Running the starter code & dataset generation =====\n")

def sho_analytic(t, A=1.0, k=1.0):
    """ Analytic solution for the 1D harmonic oscillator.

    Initial conditions are
    q(0) = A
    p(0) = 0

    Parameters are
    mass m = 1
    spring constant k > 0
    """

    omega = np.sqrt(k)
    q     = A * np.cos(omega*t)
    p     = -A*omega*np.sin(omega*t)
    return q, p

def sho_rhs(q, p, k=1.0):
    """Right-hand side of the

    dq/dt = p
    dp/dt = -k q
    """

    dqdt = p
    dpdt = -k * q
    return dqdt, dpdt

def rk4_step(q, p, dt, k=1.0):
    """Single RK4 step taking us from time t to t+dt"""
    k1_q, k1_p = sho_rhs(q, p, k)
    k2_q, k2_p = sho_rhs(q + 0.5*dt*k1_q, p + 0.5*dt*k1_p, k)
    k3_q, k3_p = sho_rhs(q + 0.5*dt*k2_q, p + 0.5*dt*k2_p, k)
    k4_q, k4_p = sho_rhs(q + dt*k3_q, p + dt*k3_p, k)

    q_next = q + (dt/6.0) * (k1_q + 2*k2_q + 2*k3_q + k4_q)
    p_next = p + (dt/6.0) * (k1_p + 2*k2_p + 2*k3_p + k4_p)
    return q_next, p_next

def sho_numerical(A=1.0, k=1.0, T=10.0, dt=0.01):
    """ Numerical solution of the harmonic oscillator. """

    t = np.arange(0.0, T + dt, dt)
    q = np.zeros_like(t)
    p = np.zeros_like(t)

    # initial conditions
    q[0] = A
    p[0] = 0.0

    for n in range(len(t) - 1):
        q[n+1], p[n+1] = rk4_step(q[n], p[n], dt, k)

    return t, q, p

print("\n1. Compute exact and numerical p, q:\n")

times, q_num, p_num = sho_numerical(A=1.0, k=1.0, T=10.0, dt=0.01)
q_exact, p_exact    = sho_analytic(times, A=1.0, k=1.0)

# parameters for plotting
plt.rcParams['font.family'] = 'serif'
plt.rcParams['font.serif'] = 'cmr10'
plt.rcParams['mathtext.fontset'] = 'cm'
plt.rcParams['font.size'] = 22
mpl.rcParams['axes.unicode_minus'] = False
plt.rcParams['axes.formatter.use_mathtext'] = True

fig, ax = plt.subplots(figsize=(15, 6))
ax.plot(times,q_num, 'b-', lw=3, label=r'$q_{num}(t)$')
ax.plot(times,q_exact, 'r--', lw=3, label=r'$q_{exact}(t)$')
ax.plot(times,p_num, 'g-', lw=3, label=r'$p_{num}(t)$')
ax.plot(times,p_exact, '--', color="orange", lw=3, label=r'$p_{exact}(t)$')
plt.xlabel(r"$t$")
plt.ylabel(r"$p(t), q(t)$")
plt.xlim(0,10)
plt.legend(frameon=False)
plt.tick_params(axis="both", which="both", direction="in")
plt.savefig("pq_exact_num.pdf", dpi=1080)
plt.show()

# plot error
fig, ax = plt.subplots(figsize=(15, 6))
ax.plot(times, (q_num - q_exact)*1e10, "k-", lw=3)
plt.xlabel(r'$t$')
plt.ylabel(r'$e(t)\times10^{10}$')
plt.xlim(0,10)
plt.tick_params(axis="both", which="both", direction="in")
plt.savefig("q_exact_num_err.pdf", dpi=1080)
plt.show()

max_err = np.max(np.abs(q_num - q_exact))
print(f"Maximum absolute error = {max_err:.3e}")

print("\n2. Data generation:\n")

k = 1.0
A_list = [0.5, 1.0, 1.5, 2.0]
T = 10.0
dt = 0.01

times = np.arange(0.0, T + dt, dt)  # shape (1001,)
all_q = []
```

```python
all_p = []

for A in A_list:
    q_A, p_A = sho_analytic(times, A=A, k=k)
    all_q.append(q_A)
    all_p.append(p_A)

all_q = np.stack(all_q, axis=0)  # shape (m, N_t)
all_p = np.stack(all_p, axis=0)  # shape (m, N_t)

# flatten trajectories over all amplitudes and times
q_flat = all_q.reshape(-1)  # shape (4004,)
p_flat = all_p.reshape(-1)  # shape (4004,)

# use analytic expression (qdot = p_exact,  pdot = -k * q_exact)
qdot_flat = p_flat.copy()
pdot_flat = -k * q_flat

# inputs X = (q, p), outputs y = (qdot, pdot)
X = np.stack([q_flat,   p_flat],   axis=1)  # shape (4004, 2)
Y = np.stack([qdot_flat, pdot_flat], axis=1)  # shape (4004, 2)

print("X shape:", X.shape)
print("Y shape:", Y.shape)

rng = np.random.default_rng(seed=0)

N = X.shape[0]
perm = rng.permutation(N)

# 80-20 split
train_frac = 0.8
N_train = int(train_frac * N)
train_idx = perm[:N_train]
val_idx   = perm[N_train:]

X_train, Y_train = X[train_idx], Y[train_idx]
X_val,   Y_val   = X[val_idx],   Y[val_idx]

print("Train size:", X_train.shape[0])
print("Val size  :", X_val.shape[0])

X_train_jax = jnp.asarray(X_train)
Y_train_jax = jnp.asarray(Y_train)
X_val_jax   = jnp.asarray(X_val)
Y_val_jax   = jnp.asarray(Y_val)

# ===== C. Learning dynamics with a neural Hamiltonian =====
print("\n===== C. Learning dynamics with a neural Hamiltonian =====\n")

# NN architecture
class HamiltonianNN(nn.Module):
    hidden: Sequence[int] = (32,)

    @nn.compact
    def __call__(self, x):
        x = x.astype(jnp.float64)
        for h in self.hidden:
            x = nn.tanh(nn.Dense(h, dtype=jnp.float64)(x))
            #x = nn.relu(nn.Dense(h, dtype=jnp.float64)(x))
        return nn.Dense(1, dtype=jnp.float64)(x)

# Hamiltonian vector fields' prediction
def hamiltonian_vector_field(params, model, qp):

    def H_single(z):
        return model.apply(params, z[None,:]).sum()

    grads = jax.vmap(jax.grad(H_single))(qp)
    dH_dq = grads[:,0:1]
    dH_dp = grads[:,1:2]

    return jnp.concatenate([dH_dp, -dH_dq], axis=1)

# loss function
def loss_fn(params, model, X, Y):
    Y_pred = hamiltonian_vector_field(params, model, X)
    return jnp.mean((Y_pred - Y)**2)

# lbfgs as optimizer
optimizer = optax.lbfgs()

model = HamiltonianNN()
key = random.PRNGKey(0)
params = model.init(key, X_train_jax[:1])
opt_state = optimizer.init(params)

def lbfgs_step(params, opt_state, X, Y):

    # compute loss and gradient
    loss, grad = jax.value_and_grad(loss_fn)(params, model, X, Y)

    # update call
    updates, opt_state = optimizer.update(
        grad,
        opt_state,
        params=params,
        value=loss,
        grad=grad,
        value_fn=lambda p: loss_fn(p, model, X, Y)
    )

    # apply update
    params = optax.apply_updates(params, updates)
    return params, opt_state, loss

# params
num_epochs = 500
train_losses = []
val_losses = []

# training loop
for epoch in range(1, num_epochs+1):

    params, opt_state, train_loss = lbfgs_step(
        params, opt_state, X_train_jax, Y_train_jax
    )

```

```python
    val_loss = loss_fn(params, model, X_val_jax, Y_val_jax)

    train_losses.append(train_loss)
    val_losses.append(val_loss)

    if epoch == 1 or epoch % 50 == 0:
        print(f"Epoch {epoch}: train_loss={train_loss:.6e}, val_loss={val_loss:.6e
            }")

# plot training and validation losses
fig, ax = plt.subplots(figsize=(15, 6))
ax.semilogy(train_losses, "b-", lw=3, label="Training")
ax.semilogy(val_losses, "r--", lw=3, label="Validation")
plt.xlabel(r"Epoch")
plt.ylabel(r"$MSE$")
plt.xlim(0, 500)
plt.legend(frameon=False)
plt.tick_params(axis="both", which="both", direction="in")
plt.savefig("hamilton_loss.pdf", dpi=1080)
plt.show()

# ===== D. Using your new model =====
print("\n===== D. Using your new model =====\n")

A_test = 1.25

def learned_rhs(params, model, q, p):
    qp = jnp.array([[q, p]])
    qdot, pdot = hamiltonian_vector_field(params, model, qp)[0]
    return float(qdot), float(pdot)

def rk4_step_learned(q, p, dt, params, model):
    k1_q, k1_p = learned_rhs(params, model, q, p)
    k2_q, k2_p = learned_rhs(params, model, q + 0.5*dt*k1_q, p + 0.5*dt*k1_p)
    k3_q, k3_p = learned_rhs(params, model, q + 0.5*dt*k2_q, p + 0.5*dt*k2_p)
    k4_q, k4_p = learned_rhs(params, model, q + dt*k3_q, p + dt*k3_p)
    q_next = q + (dt/6.0)*(k1_q + 2*k2_q + 2*k3_q + k4_q)
    p_next = p + (dt/6.0)*(k1_p + 2*k2_p + 2*k3_p + k4_p)
    return q_next, p_next

def integrate_learned(A_test, T=10.0, dt=0.01):
    t = np.arange(0.0, T+dt, dt)
    q = np.zeros_like(t); p = np.zeros_like(t)
    q[0] = A_test; p[0] = 0.0
    for n in range(len(t)-1):
        q[n+1], p[n+1] = rk4_step_learned(q[n], p[n], dt, params, model)
    return t, q, p

# integrate and compare to analytic solution
t, q_learn, p_learn = integrate_learned(A_test, T=10.0, dt=0.01)
q_exact, p_exact    = sho_analytic(t, A=A_test, k=1.0)

# plot trajectories
fig, ax = plt.subplots(figsize=(15, 6))
ax.plot(t, q_exact, 'b-', lw=3, label=r'$q_{exact}(t)$')
ax.plot(t, q_learn, 'r--', lw=3, label=r'$q_\theta(t)$')
ax.plot(t, p_exact, 'g-', lw=3, label=r'$p_{exact}(t)$')
ax.plot(t, p_learn, '--', color="orange", lw=3, label=r'$p_\theta(t)$')
plt.xlabel(r'$t$')
plt.ylabel(r'$p, q$')
plt.xlim(0,10)
plt.ylim(-1.5,1.5)
plt.legend(frameon=False)
plt.tick_params(axis="both", which="both", direction="in")
plt.savefig("learned_pq.pdf", dpi=1080)
plt.show()

fig, ax = plt.subplots(figsize=(15, 6))
ax.plot(t, q_learn - q_exact, 'b-', lw=3, label=r'$q_\theta(t) - q_{exact}(t)$')
ax.plot(t, p_learn - p_exact, 'r--', lw=3, label=r'$p_\theta(t) - p_{exact}(t)$')
plt.xlabel(r'$t$')
plt.ylabel(r'$e(t)$')
plt.xlim(0,10)
plt.legend(frameon=False)
plt.tick_params(axis="both", which="both", direction="in")
plt.savefig("learned_pq_err.pdf", dpi=1080)
plt.show()

# extrapolate
T_long = 100.0
t_long, q_learn_long, p_learn_long = integrate_learned(A_test, T=T_long, dt=0.01)
q_exact_long, p_exact_long = sho_analytic(t_long, A=A_test, k=1.0)

# plot trajectories over long period
fig, ax = plt.subplots(figsize=(15, 6))
ax.plot(t_long, q_exact_long, 'b-', lw=3, label=r'$q_{exact}$')
ax.plot(t_long, q_learn_long, 'r--', lw=3, label=r'$q_\theta$')
ax.plot(t_long, p_exact_long, 'g-', lw=3, label=r'$p_{exact}(t)$')
ax.plot(t_long, p_learn_long, '--', color="orange", lw=3, label=r'$p_\theta(t)$')
plt.xlabel(r'$t$')
plt.ylabel(r'$p, q$')
plt.xlim(t_long.min(), t_long.max())
plt.ylim(-1.75,1.75)
plt.legend(loc="upper left", ncol=4, frameon=False)
plt.tick_params(axis="both", which="both", direction="in")
plt.savefig("learned_pq_ex.pdf", dpi=1080)
plt.show()

# plot error vs time
fig, ax = plt.subplots(figsize=(15, 6))
ax.plot(t_long, q_learn_long - q_exact_long, "b-", lw=3, label=r'$q_\theta(t) - q_
    {exact}(t)$')
ax.plot(t_long, p_learn_long - p_exact_long, "r--", lw=3, label=r'$p_\theta(t) -
    p_{exact}(t)$')
plt.xlabel(r'$t$');
plt.ylabel(r'$e(t)$')
plt.xlim(t_long.min(), t_long.max())
plt.ylim(-0.01,0.01)
plt.legend(loc="lower left", ncol=2, frameon=False)
plt.tick_params(axis="both", which="both", direction="in")
plt.savefig("learned_pq_ex_err.pdf", dpi=1080)
plt.show()

# ===== E. Going further =====
# conservation-based physics loss
def loss_fn_phys(params, model, X_batch, Y_batch, lam=1e-5):
    Y_pred = hamiltonian_vector_field(params, model, X_batch)
    data_loss = jnp.mean((Y_pred - Y_batch)**2)

    # energy conservation loss: H(q_{t+1},p_{t+1}) - H(q_t,p_t) should be 0
    X_t   = X_batch[:-1]
    X_tp1 = X_batch[1:]

    H_t   = model.apply(params, X_t)
    H_tp1 = model.apply(params, X_tp1)

    energy_conservation = jnp.mean((H_tp1 - H_t)**2)

    return data_loss + lam * energy_conservation

def train_step_phys(params, opt_state, Xb, Yb):
    loss, grads = jax.value_and_grad(loss_fn_phys)(params, model, Xb, Yb)

    updates, opt_state = optimizer.update(
        grads,
        opt_state,
        params=params,
        value=loss,
        grad=grads,
        value_fn=lambda p: loss_fn_phys(p, model, Xb, Yb)
    )
    params = optax.apply_updates(params, updates)
    return params, opt_state, loss


# retrain model
num_epochs = 500
train_losses = []
val_losses = []

params = model.init(random.PRNGKey(99), X_train_jax[:1])
opt_state = optimizer.init(params)

for epoch in range(1, num_epochs+1):
    params, opt_state, train_loss = train_step_phys(
        params, opt_state, X_train_jax, Y_train_jax
    )
    val_loss = loss_fn_phys(params, model, X_val_jax, Y_val_jax)

    train_losses.append(train_loss)
    val_losses.append(val_loss)

    if epoch == 1 or epoch % 50 == 0:
        print(f"Epoch {epoch}: train={train_loss:.3e}, val={val_loss:.3e}")

# plot physics-informed training performance
fig, ax = plt.subplots(figsize=(15, 6))
ax.semilogy(train_losses, "b-", lw=3, label="Training")
ax.semilogy(val_losses, "r--", lw=3, label="Validation")
plt.xlabel(r"Epoch")
plt.ylabel(r"$MSE$")
plt.legend(frameon=False)
plt.tick_params(axis="both", which="both", direction="in")
plt.savefig("hamilton_loss_phys.pdf", dpi=1080)
plt.show()

# evaluate for new amplitude
```

```python
A_test = 1.25

def learned_rhs(params, model, q, p):
    qp = jnp.array([[q, p]])
    qdot, pdot = hamiltonian_vector_field(params, model, qp)[0]
    return float(qdot), float(pdot)

def rk4_step_learned(q, p, dt):
    k1_q, k1_p = learned_rhs(params, model, q, p)
    k2_q, k2_p = learned_rhs(params, model, q + 0.5*dt*k1_q, p + 0.5*dt*k1_p)
    k3_q, k3_p = learned_rhs(params, model, q + 0.5*dt*k2_q, p + 0.5*dt*k2_p)
    k4_q, k4_p = learned_rhs(params, model, q + dt*k3_q, p + dt*k3_p)
    return (
        q + (dt/6)*(k1_q + 2*k2_q + 2*k3_q + k4_q),
        p + (dt/6)*(k1_p + 2*k2_p + 2*k3_p + k4_p)
    )

def integrate_learned(A, T, dt=0.01):
    t = np.arange(0,T+dt,dt)
    q = np.zeros_like(t); p = np.zeros_like(t)
    q[0] = A; p[0] = 0
    for n in range(len(t)-1):
        q[n+1], p[n+1] = rk4_step_learned(q[n], p[n], dt)
    return t, q, p

# for T = 10
t10, q_learn10, p_learn10 = integrate_learned(A_test, T=10)
q_exact10, p_exact10 = sho_analytic(t10, A=A_test)

# plot predicted trajectories
fig, ax = plt.subplots(figsize=(15, 6))
ax.plot(t10, q_exact10, 'b-', lw=3, label=r'$q_{exact}(t)$')
ax.plot(t10, q_learn10, 'r--', lw=3, label=r'$q_\theta(t)$')
ax.plot(t10, p_exact10, 'g-', lw=3, label=r'$p_{exact}(t)$')
ax.plot(t10, p_learn10, '--', color="orange", lw=3, label=r'$p_\theta(t)$')
plt.xlabel(r'$t$')
plt.ylabel(r'$p, q$')
plt.xlim(t10.min(), t10.max())
plt.ylim(-1.5,1.5)
plt.legend(frameon=False)
plt.tick_params(axis="both", which="both", direction="in")
plt.savefig("learned_pq_phys.pdf", dpi=1080)
plt.show()

fig, ax = plt.subplots(figsize=(15, 6))
ax.plot(t, q_learn - q_exact, 'b-', lw=3, label=r'$q_\theta(t) - q_{exact}(t)$ (w/
    o physics)')
ax.plot(t, q_learn10 - q_exact10, 'r--', lw=3, label=r'$q_\theta(t) - q_{exact}(t)
    $ (w/ physics)')
ax.plot(t, p_learn - p_exact, 'g-', lw=3, label=r'$p_\theta(t) - p_{exact}(t)$ (w/
    o physics)')
ax.plot(t, p_learn10 - p_exact10, '--', color="orange", lw=3, label=r'$p_\theta(t)
     - p_{exact}(t)$ (w/ physics)')
plt.xlabel(r'$t$')
plt.ylabel(r'$e(t)$')
plt.xlim(t10.min(), t10.max())
plt.legend(ncol=2, frameon=False)
plt.tick_params(axis="both", which="both", direction="in")
plt.savefig("learned_pq_err_phys.pdf", dpi=1080)
plt.show()

# extrapolation
t_long100, q_learn_long100, p_learn_long100 = integrate_learned(A_test, T=100)
q_exact_long100, p_exact_long100 = sho_analytic(t_long100, A=A_test)

fig, ax = plt.subplots(figsize=(15, 6))
ax.plot(t_long100, q_exact_long100, 'b-', lw=3, label=r'$q_{exact}(t)$')
ax.plot(t_long100, q_learn_long100, 'r--', lw=3, label=r'$q_\theta(t)$')
ax.plot(t_long100, p_exact_long100, 'g-', lw=3, label=r'$p_{exact}(t)$')
ax.plot(t_long100, p_learn_long100, '--', lw=3, color="orange", label=r'$p_\theta(
    t)$')
plt.xlabel(r'$t$');
plt.ylabel(r'$p, q$')
plt.xlim(t_long100.min(),t_long100.max())
plt.ylim(-1.75,1.75)
plt.legend(ncol=4, frameon=False)
plt.tick_params(axis="both", which="both", direction="in")
plt.savefig("learned_pq_ex_phys.pdf", dpi=1080)
plt.show()

fig, ax = plt.subplots(figsize=(15, 6))
ax.plot(t_long, q_learn_long - q_exact_long, 'b-', lw=3, label=r'$q_\theta(t) - q_
    {exact}(t)$ (w/o physics)')
ax.plot(t_long100, q_learn_long100 - q_exact_long100, 'r--', lw=3, label=r'$q_\
    theta(t) - q_{exact}(t)$ (w/ physics)')
ax.plot(t_long, p_learn_long - p_exact_long, 'g-', lw=3, label=r'$p_\theta(t) - p_
    {exact}(t)$ (w/o physics)')
ax.plot(t_long100, p_learn_long100 - p_exact_long100, '--', color="orange", lw=3,
    label=r'$p_\theta(t) - p_{exact}(t)$ (w/ physics)')
plt.xlabel(r'$t$');
plt.ylabel(r'$e(t)$')
plt.xlim(t_long100.min(),t_long100.max())
plt.legend(ncol=2, frameon=False)
plt.tick_params(axis="both", which="both", direction="in")
plt.savefig("learned_pq_ex_err_phys.pdf.pdf", dpi=1080)
plt.show()
```

Listing 3: `neural_hamiltonian.py`

```
===== B. Running the starter code & dataset generation =====

1. Compute exact and numerical p, q:

Maximum absolute error = 6.604e-10

2. Data generation:

X shape: (4004, 2)
Y shape: (4004, 2)
Train size: 3203
Val size  : 801

===== C. Learning dynamics with a neural Hamiltonian =====

Epoch 1: train_loss=9.587591e-01, val_loss=9.098519e-01
Epoch 50: train_loss=8.340965e-05, val_loss=7.641729e-05
Epoch 100: train_loss=2.237719e-05, val_loss=2.110202e-05
Epoch 150: train_loss=7.755750e-06, val_loss=7.434588e-06
Epoch 200: train_loss=3.566024e-06, val_loss=3.408729e-06
Epoch 250: train_loss=1.507338e-06, val_loss=1.473887e-06
Epoch 300: train_loss=1.030428e-06, val_loss=1.050491e-06
Epoch 350: train_loss=8.345358e-07, val_loss=8.522308e-07
Epoch 400: train_loss=6.878314e-07, val_loss=6.896047e-07
Epoch 450: train_loss=6.419417e-07, val_loss=6.636466e-07
Epoch 500: train_loss=6.012402e-07, val_loss=6.222261e-07

===== D. Using your new model =====

===== E. Going further =====

Epoch 1: train=1.435e+00, val=9.000e-01
Epoch 50: train=7.056e-05, val=6.243e-05
Epoch 100: train=2.509e-05, val=2.466e-05
Epoch 150: train=1.658e-05, val=1.643e-05
Epoch 200: train=1.371e-05, val=1.384e-05
Epoch 250: train=1.237e-05, val=1.231e-05
Epoch 300: train=1.151e-05, val=1.146e-05
Epoch 350: train=1.111e-05, val=1.099e-05
Epoch 400: train=1.086e-05, val=1.074e-05
Epoch 450: train=1.063e-05, val=1.054e-05
Epoch 500: train=1.051e-05, val=1.042e-05
```

Listing 4: Output terminal for `neural_hamiltonian.py`