Introduction to Walsh-Hadamard Transform Autoencoders

Walsh-Hadamard Transform (WHT) autoencoders are a specialized type of linear autoencoder that leverage the orthogonal properties of the Walsh-Hadamard matrix for encoding and decoding. From first principles, this approach treats the WHT as a fixed, deterministic transformation basis, inspired by signal processing and information theory, where the goal is to project data into a sparse or compressed domain (via the transform) and reconstruct it faithfully. Unlike general neural autoencoders with learnable weights, here the "encoder" and "decoder" are predefined matrix multiplications using the Hadamard matrix, making it computationally efficient and interpretable. It's unsupervised, learning nothing—wait, actually, since the transform is fixed, "training" reduces to a single forward pass per sample, but we still evaluate reconstruction using a loss like RMSE to assess quality. This is akin to classical dimensionality reduction techniques like PCA, but with the binary, square-wave basis of Walsh functions, which excel at capturing high-frequency details in binary or piecewise-constant signals.

The core idea: The WHT decomposes a signal into Walsh basis functions (products of Rademacher functions), allowing perfect reconstruction for orthogonal projections. For autoencoding, we apply a bottleneck by selecting only the top-k coefficients in the transform domain, forcing compression while minimizing information loss.

Let's break this down step by step, starting from the basics.

1. Basic Components of a WHT Autoencoder

A WHT autoencoder consists of three main parts:

- **Encoder**: Takes the input data x (a vector of dimension d, e.g., a flattened image) and projects it onto the Walsh basis via matrix multiplication: $z = \frac{1}{\sqrt{d}} H x$, where H is the normalized Walsh-Hadamard matrix of size $d \times d$. Here, z is the full transform; for compression, we zero out small coefficients to get a sparse z of effective dimension $m < d$.
- **Latent Space (Bottleneck)**: The coefficients z in the Walsh domain. The bottleneck is enforced by thresholding or selecting only the largest m absolute values (keeping signs), which discards low-energy components, akin to lossy compression.
- **Decoder**: Reconstructs via the inverse transform: $\hat{x} = \frac{1}{\sqrt{d}} H z$, since H is symmetric and orthogonal ($HH = dI$, so inverse is H/d, but normalized version uses $1/\sqrt{d}$ for unitarity). This is exact if no bottleneck; with it, reconstruction is approximate.

The full autoencoder is $\hat{x} = \frac{1}{\sqrt{d}} H \left(\frac{1}{\sqrt{d}} H x \right) = x$ without bottleneck—perfect reconstruction. The bottleneck introduces the learning-like compression.

2. Mathematical Formulation

Assume a dataset $\{x^{(1)}, x^{(2)}, \dots, x^{(n)}\}$, each $x^{(i)} \in \mathbb{R}^d$ (or $\mathbb{R}^d$ normalized to [-1,1] for Hadamard).

- **Walsh-Hadamard Matrix**: For $d = 2^k$, H is recursively defined: $H_1 = [1]$, $H_2 = \begin{bmatrix} H_1 & H_1 \\ H_1 & -H_1 \end{bmatrix}$, and so on. Normalized: $H_{\text{norm}} = H / \sqrt{d}$.
- **Forward Transform (Encoder)**: $z = H_{\text{norm}} x$.
- **Bottleneck**: $z_{\text{bottleneck}} = \text{top-k}(z, m)$, where we keep the m largest |z| entries, zeroing others.
- **Inverse Transform (Decoder)**: $\hat{x} = H_{\text{norm}} z_{\text{bottleneck}}$.
- **Objective**: Minimize the Root Mean Squared Error (RMSE) over the dataset:

$$\text{RMSE}(x, \hat{x}) = \sqrt{\frac{1}{d} \|x - \hat{x}\|_2^2}$$

For the dataset:

$$J = \frac{1}{n} \sum_{i=1}^{n} \text{RMSE}(x^{(i)}, \hat{x}^{(i)})$$

Since no parameters are learned, J is just an evaluation metric—no optimization loop. We tune hyperparameters like m or threshold by minimizing J on validation data.

This formulation ensures orthogonality preserves energy: $\|x\|^2 = \|z\|^2$, minimizing distortion in the L2 sense.

3. Intuition and Why It Works

The Walsh basis consists of constant and square-wave functions at dyadic frequencies, ideal for signals with sharp transitions (e.g., edges in images). The encoder "compresses" by retaining dominant coefficients, discarding noise or redundancies. The decoder rebuilds using the basis expansion. Without bottleneck, it's lossless; with it, it approximates like JPEG's DCT but faster (O(d log d) via fast WHT algorithm).

- **Undercomplete**: $m < d$ forces sparsity, similar to PCA but with Walsh basis (better for binary data).
- **Overcomplete**: Not typical here, as H is square; extensions could use oversampled WHT.

It assumes data has sparse representation in Walsh domain, common for natural images or time series.

4. "Training" Process

Since fixed, it's evaluation-focused:

1. **Generate H**: Construct normalized Hadamard matrix for d (pad if needed).
2. **Forward Pass**: For each x, compute $z = H_{\text{norm}} x$, apply bottleneck to get z_b, then $\hat{x} = H_{\text{norm}} z_b$.
3. **Compute RMSE**: Average over dataset.
4. **Tune**: Vary m or threshold to minimize average RMSE.
5. **Evaluation**: Plot reconstructions or latent histograms. Use fast WHT for speed (butterfly algorithm, like FFT).

Hyperparameters: m (e.g., 10-50% of d), normalization (for signed data).

5. Variants of WHT Autoencoders

- **Denoising**: Apply transform to noisy input, threshold coefficients, inverse to clean.
- **Sparse WHT**: Use L1-like thresholding on |z| for automatic sparsity.
- **Seeded WHT (SWHT)**: Randomized permutations for privacy-preserving compression.
- **Multidimensional WHT**: Kronecker product for 2D images (row-column transforms).

6. Example: A WHT Autoencoder on MNIST

We'll implement a simple WHT autoencoder using NumPy and SciPy on MNIST (pad 784D to 1024D=2^{10}). Bottleneck keeps top-128 coefficients. RMSE as loss (evaluated, not optimized). Download MNIST via code if needed, but assume available.

```
Python
```

```
import numpy as np
from scipy.linalg import hadamard
from sklearn.datasets import fetch_openml
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler
import matplotlib.pyplot as plt

# Load and prepare MNIST (first 1000 samples for demo)
mnist = fetch_openml('mnist_784', version=1, as_frame=False, parser='auto')
X = mnist.data[:1000]  # 1000 x 784
X = X.astype(np.float32)

# Pad to 1024 dims (nearest power of 2)
d = 1024
X_padded = np.zeros((X.shape[0], d))
X_padded[:, :784] = X
X = X_padded  # Now 1000 x 1024

# Normalize to [-1, 1] (Hadamard works best here)
scaler = MinMaxScaler(feature_range=(-1, 1))
X = scaler.fit_transform(X)

# Generate normalized Hadamard matrix
H = hadamard(d).astype(np.float32)
H_norm = H / np.sqrt(d)  # Unitary

# Bottleneck function: keep top-k by absolute value
def bottleneck(z, m=128):
    z_b = np.zeros_like(z)
    indices = np.argsort(np.abs(z), axis=1)[:, -m:]  # Top m per sample
    for i, idx in enumerate(indices):
        z_b[i, idx] = z[i, idx]
    return z_b

# Autoencoder forward
def wht_autoencoder(x, H_norm, m=128):
    z = H_norm @ x.T  # d x n
    z_b = bottleneck(z.T, m).T  # n x d -> bottleneck -> d x n
    x_hat = H_norm @ z_b  # d x n
    return x_hat.T  # n x d

# "Train"/Evaluate RMSE on train/test split
```

```
X_train, X_test
= train_test_split(X, test_size=0.2, random_state=42)
x_recon = wht_autoencoder(X_train, H_norm)
rmse_train = np.sqrt(np.mean((X_train - x_recon)**2, axis=1)).mean()
print(f'Train RMSE: {rmse_train:.4f}')

# Example reconstruction visualization (first test sample)
x_test_sample = X_test[0:1]  # 1 x 1024
x_recon_sample = wht_autoencoder(x_test_sample, H_norm)
# Unpad for plot
def unpad(img):
    return img[:, :784].reshape(28, 28)

plt.figure(figsize=(4, 2))
plt.subplot(1, 2, 1)
plt.imshow(unpad(x_test_sample), cmap='gray')
plt.title('Original')
plt.subplot(1, 2, 2)
plt.imshow(unpad(x_recon_sample), cmap='gray')
plt.title('Reconstruction')
plt.show()
```

Explanation of the Code:

- **Padding/Normalization**: Extend to 1024D with zeros; scale to [-1,1] for Hadamard efficiency.
- **Hadamard Matrix**: SciPy generates unnormalized H; we normalize for unitarity.
- **Encoder/Bottleneck**: Matrix mult for transform, then select top-128 |z| coeffs (sparsity ~88% reduction).
- **Decoder**: Inverse mult reconstructs.
- **Evaluation**: Compute RMSE on train split. Expect ~0.05-0.15 depending on m (lower m = higher RMSE).
- **Visualization**: Plot original vs. recon (unpadded). Recon should preserve digit shape with some blurring.

Run this: Train RMSE might be ~0.08 for m=128, indicating decent compression. For denoising, add Gaussian noise to X_train before encoding.

Potential Challenges and Tips

- **Non-Power-of-2 Dimensions**: Pad or use sequency-ordered WHT approximations.
- **Sign Sensitivity**: Ensure data centering; Walsh is for zero-mean signals.
- **Scalability**: Use fast WHT (in-place butterfly) for large d.
- **Applications**: Image compression, watermarking, or anomaly detection (high RMSE = outlier).

Good Resources

- **Deep Learning Book by Ian Goodfellow et al.**: Chapter 14 (adapt linear AE sections to orthogonal transforms; free at deeplearningbook.org).
- **SciPy Documentation on Hadamard**: docs.scipy.org/doc/scipy/reference/generated/scipy.linalg.hadamard.html – For matrix generation.
- **"The Walsh-Hadamard Transform" Tutorial (DSP Guide)**: dspguide.com/ch20.pdf – First-principles signal processing view.
- **PyTorch/NumPy WHT Implementation Repo**: github.com/topics/walsh-hadamard – Code examples.
- **IEEE Paper: "Walsh-Hadamard Transform for Image Compression"**: Search arXiv for extensions to autoencoding.
- **Andrew Ng's Unsupervised Learning Notes**: Adapt PCA sections to WHT basis.