A Reproducibility

All code will be released under the URL `https://github.com/rainerkelz/ICBINB21`.

B Domain Transfer Function Architecture and Training

All audio was (re-)sampled at 44100[Hz], all spectrograms were computed using a 4096-point discrete Fourier transform, and a hop size of 512 samples. A mel-filterbank was used to reduce the linear spectrogram to a mel spectrogram with 256 logarithmically spaced bins. These mel spectrograms from the source and target domains were paired up and cut into temporally overlapping 256×256 square snippets (hop-size of 128 frames).

We tried multiple different objective functions, such as mean squared error, Huber loss and even to directly maximize a structural similarity measure [9], and some combinations thereof. In the end, we decided on using the mean squared error. Interestingly, minimizing the mean squared error also maximized the structural similarity measure *much better* than trying to include it directly (and yes, we made sure we had our signs right).

Because our domain pairs are actually somewhat close in appearance already, we did not try any adversarial loss, as in the `pix2pix` approach [6], or, even more relevant, the `TimbreTron` approach [5], but we consider these a definite next step, along with the possibility to extend the supervised to the semi-supervised setting.

The architecture of the domain transfer function is that of a UNet [8]. The exact architecture can be found below:

```
Unet(
  (inc): Sequential(
    (0): Conv2d(1, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): LeakyReLU(negative_slope=0.01, inplace=True)
    (3): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (4): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (5): LeakyReLU(negative_slope=0.01, inplace=True)
  )
  (down1): Sequential(
    (0): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (1): Sequential(
      (0): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): LeakyReLU(negative_slope=0.01, inplace=True)
      (3): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (4): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (5): LeakyReLU(negative_slope=0.01, inplace=True)
    )
  )
  (down2): Sequential(
    (0): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (1): Sequential(
      (0): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): LeakyReLU(negative_slope=0.01, inplace=True)
      (3): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (4): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (5): LeakyReLU(negative_slope=0.01, inplace=True)
    )
  )
  (down3): Sequential(
    (0): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (1): Sequential(
      (0): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): LeakyReLU(negative_slope=0.01, inplace=True)
      (3): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (4): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (5): LeakyReLU(negative_slope=0.01, inplace=True)
    )
  )
  (down4): Sequential(
    (0): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (1): Sequential(
      (0): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): LeakyReLU(negative_slope=0.01, inplace=True)
      (3): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (4): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (5): LeakyReLU(negative_slope=0.01, inplace=True)
    )
  )
  (up1): up(
    (up): Upsample(scale_factor=2.0, mode=bilinear)
    (conv): Sequential(
      (0): Conv2d(1024, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): LeakyReLU(negative_slope=0.01, inplace=True)
      (3): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (4): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (5): LeakyReLU(negative_slope=0.01, inplace=True)
    )
  )
  (up2): up(
    (up): Upsample(scale_factor=2.0, mode=bilinear)
    (conv): Sequential(
      (0): Conv2d(512, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): LeakyReLU(negative_slope=0.01, inplace=True)
      (3): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (4): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (5): LeakyReLU(negative_slope=0.01, inplace=True)
    )
  )
  (up3): up(
    (up): Upsample(scale_factor=2.0, mode=bilinear)
    (conv): Sequential(
      (0): Conv2d(256, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): LeakyReLU(negative_slope=0.01, inplace=True)
      (3): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (4): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (5): LeakyReLU(negative_slope=0.01, inplace=True)
    )
  )
  (up4): up(
    (up): Upsample(scale_factor=2.0, mode=bilinear)
    (conv): Sequential(
      (0): Conv2d(128, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): LeakyReLU(negative_slope=0.01, inplace=True)
      (3): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (4): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (5): LeakyReLU(negative_slope=0.01, inplace=True)
    )
  )
  (out): Conv2d(64, 1, kernel_size=(1, 1), stride=(1, 1))
)
```

C Linear Decomposition

The exact non-negative decomposition objective used, consists of three terms. A reconstruction error (cf. Equation (4)), a term that encourages “Hoyer sparsity” [4] (cf. Equation (5)), and one that encourages temporal continuity [1] (cf. Equation (6)). The matrix $\mathbf{A}$ has dimensions $n \times m$, $\mathbf{A}_j$ selects the j-th column vector, $\mathbf{A}_i$ selects the i-th row vector, and $\mathbf{A}_{ij}$ selects the element in the i-th row, and the j-th column. The trade-off weights were chosen as $\lambda_r = 1, \lambda_h = \lambda_t = 10^{-4}$. We run the Adam [7] optimizer for a fixed amount of steps for each piece, projecting the elements of $\mathbf{A}$ back onto the non-negative orthant after each update.

$$\hat{\mathbf{A}} = \arg\min_{\mathbf{A}} \Big[\lambda_r \mathrm{rec}(\mathbf{A}) + \lambda_h \mathrm{hoyer}(\mathbf{A}) + \lambda_t \mathrm{temp}(\mathbf{A})\Big], \mathbf{A}_{ij} \geq 0 \tag{3}$$

$$\mathrm{rec}(\mathbf{A}) = \|\mathbf{D}\mathbf{A} - \mathbf{V}\|_2^2 \tag{4}$$

$$\mathrm{hoyer}(\mathbf{A}) = \mathrm{mean}\left[\frac{\mathrm{norm}(\mathbf{A}_j, 1)}{\mathrm{norm}(\mathbf{A}_j, 2)}\right]_{j=1..m} \tag{5}$$

$$\mathrm{temp}(\mathbf{A}) = \mathrm{mean}\Big[\|\mathbf{A}_j - \mathbf{A}_{j+k}\|_2^2\Big]_{k=1..4, j=1..m-k} \tag{6}$$

D Dataset Details

For the train / validation / test split “All”, Figure 4 shows more details about the musical overlap. In both plots, the x-axis shows the 60 different pieces in test. In the upper line plot, the y-axis shows how many different pianos in the train and validation set play the same piece. In the lower scatter plot, we can observe in detail, which piano models in the train and validation sets play which piece in the test set.

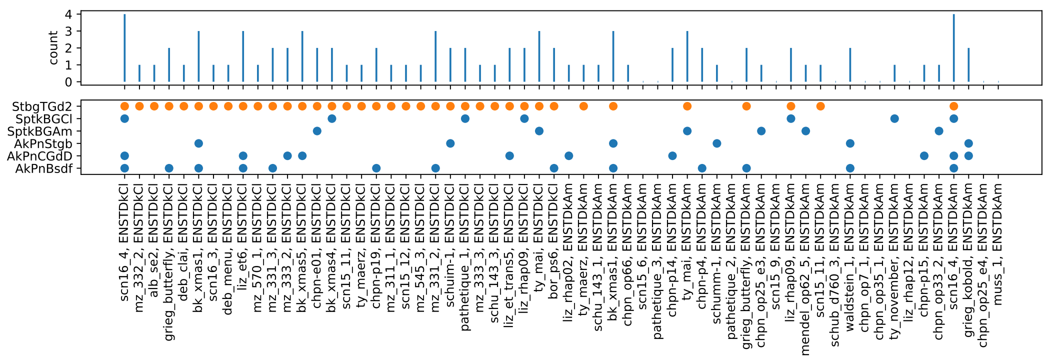

Figure 4: A more detailed view of the musical overlap in the train / validation / test split “All”. The piano model named “StbgTGd2” has the most musical overlap, which is why we chose all its 30 pieces to be the validation set. For easier visual distinction, it is depicted in orange color.

E Temporally Aligned Plots

For the sake of visual intuition building, we provide plots of temporally aligned spectrogram snippets $\mathbf{V}_S, \mathbf{V}_T, \mathbf{V}_{T'}$. The color map is two-tailed, with positive values shown in red, negative values shown in blue.

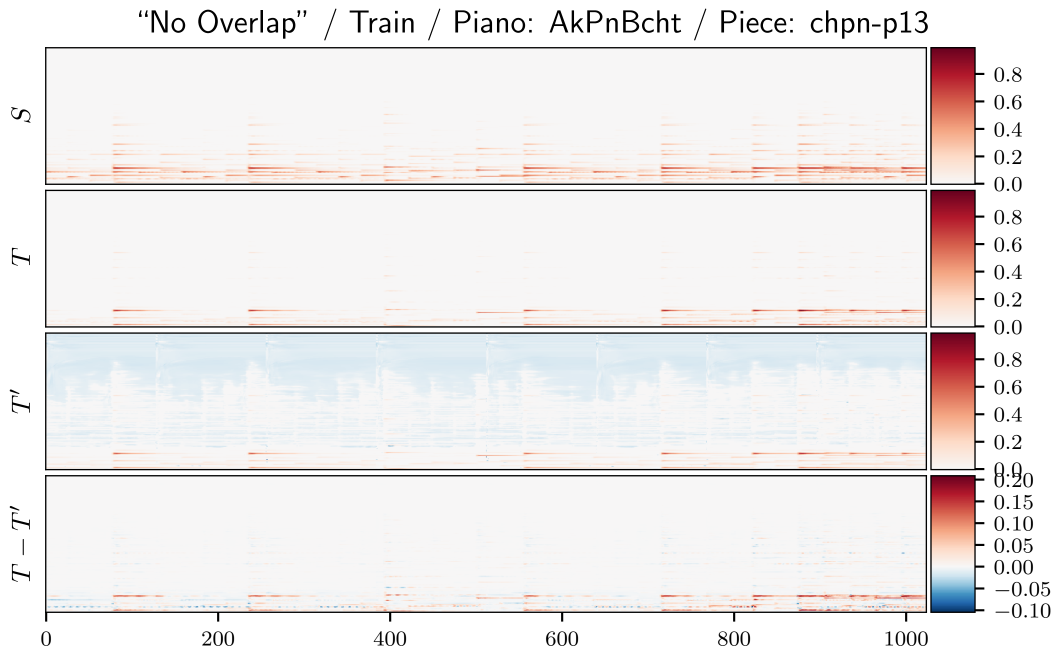

Figure 5: A snippet from the train set.

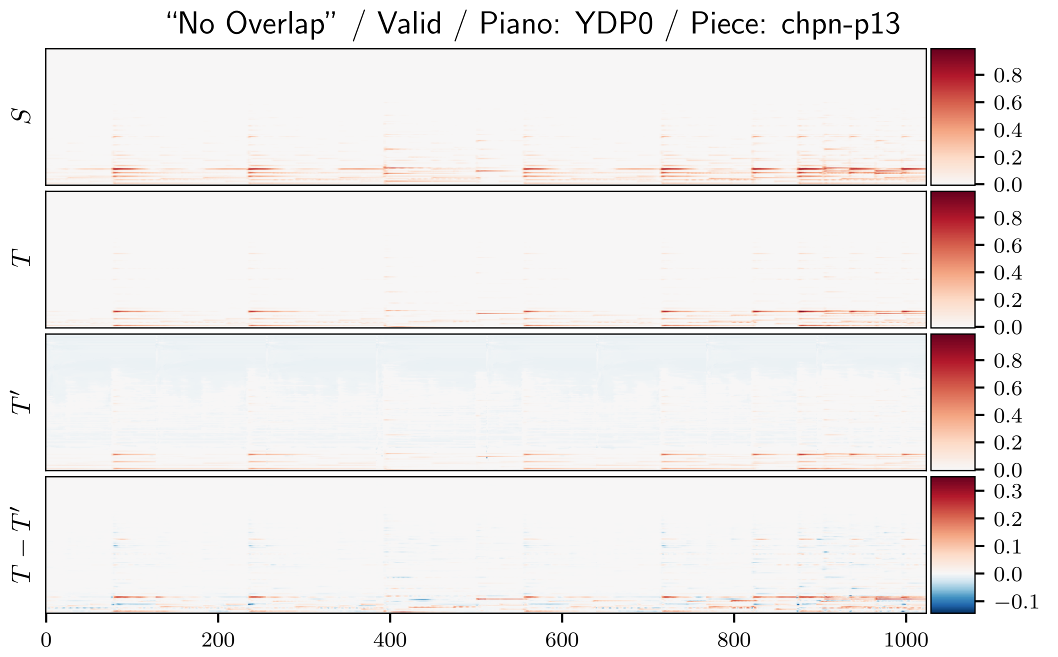

Figure 6: A snippet from the validation set.

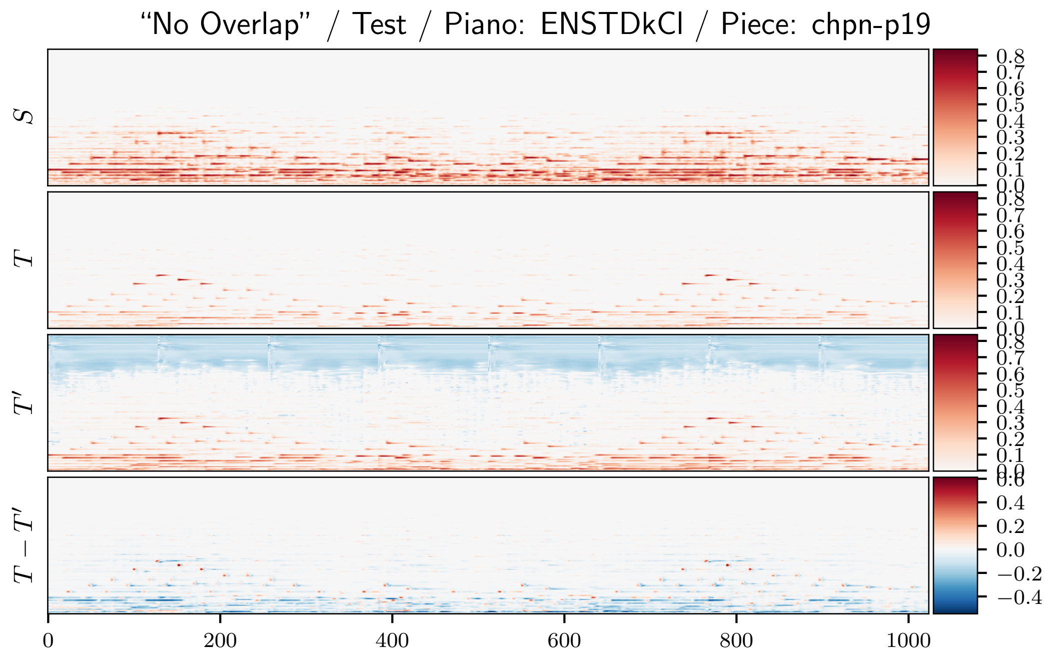

Figure 7: A snippet from the test set.

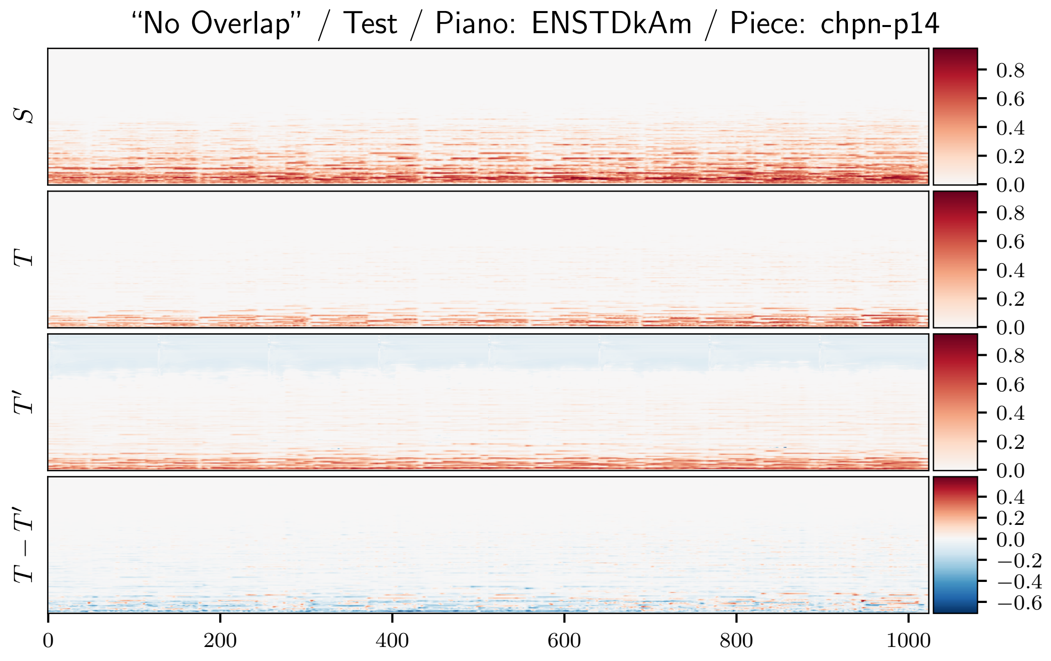

Figure 8: Another snippet from the test set.