A. Appendix

In the following we elaborate a number of technical details that had to be omitted from the main paper for lack of space.

A.1. Background on Tensor Decompositions

Low-rank tensor decompositions originated in chemometrics in the 1960ies [47], and have since the 1990ies attracted renewed interest in linear algebra and signal processing [29]. Contrary to matrices, there is no single, obvious and generally valid definition of a tensor's rank. Multiple useful definitions of tensor rank have been proposed, each of which gives rise to a different decomposition. One of the earliest was the *canonical decomposition* [22, 20, 10], with the key concept of expressing a tensor as the sum of a finite number of rank-one (separable) tensors. The decomposition is memory-efficient, but computing it may be numerically unstable; and it lacks a mechanism to control the trade-off between low rank and higher approximation error [6, 38]. The Tucker decomposition [46] factors a D-dimensional tensor into a smaller D-dimensional core tensor and D factor matrices. In contrast to canonical decomposition, in the Tucker model one can find the lowest possible rank for a given (approximation) error budget by truncation. On the downside, it is suitable only for low-dimensional tensors (typically, $D \leq 4$), as the number of parameters scales exponentially with the dimensionality D.

The more recent TT format [39] combines important advantages of the CP and Tucker models: like the former, its storage cost grows sub-exponentially with the dimension D, while it is numerically stable like the latter. We note that the recent *tensor chain* (TC) [12] is another related format that is more symmetric than TT and also has sub-exponential storage cost; however, the difficulty of rounding TC tensors makes them inadequate for the feature projection step required in our model to overcome the ambiguity of the decomposition.

A.2. Algorithm Details for Cross-Approximation

We use tensor train cross-approximation [39, 43] to learn an approximate TT decomposition of the $(D+1)$-dimensional latent encoding $\mathbf{E}$. We do this by collecting sets of samples (tensor elements) iteratively, one dimension at a time, in a sweep-like fashion: we traverse dimensions 1 through D, then $D+1$ through 2, and iterate. At each sweep k, sets of *left indices* L_d^k and *right indices* R_d^k are kept for every dimension d, in such a way that the set of selected fibers $\mathbf{E}[L_d^k, :, R_d^k]$ is a tensor of shape $r_{d-1} \times I_d \times r_d$. By organizing those fibers as row vectors, we obtain a matrix $\boldsymbol{E}_d$ of shape $(r_{d-1} \cdot I_d) \times r_d$. We then run *maxvol* [15] on $\boldsymbol{E}_d$ to select an $r_d \times r_d$ submatrix $\tilde{\boldsymbol{E}}_d$ containing r_d rows of $\boldsymbol{E}_d$, such that $|\det(\tilde{\boldsymbol{E}}_d)|$ is as large as possible. The newly selected row indices become L_{d+1}^k and are subsequently used to acquire the required samples for the next dimension $d+1$.

Once a full index selection sweep has concluded, we apply the cross interpolation formula: for each dimension d, we compute $\boldsymbol{E}_d \tilde{\boldsymbol{E}}_d^{-1}$ and reshape the result into a tensor of shape $r_{d-1} \times I_d \times r_d$ that becomes the d-th TT core $\mathbf{Q}_d$. We cast the matrix inversion product as a quadratic optimization problem and find a differentiable solution of it via the package *cvxpylayers* [1].

Forward propagation through C-PIC has complexity $\mathcal{O}\left(F_s D n r^2 + D n r^3\right)$, where s is the receptive field around a sample and F_s is the cost of a forward pass with a n_{dim}-channel tensor of size $s^3 \times n_{\text{dim}}$. We do not back-propagate through the index selection step, so the complexity of backprop for the sampled patches is $\mathcal{O}(B_s D n r^2)$, with B_s the cost of the backward pass for one patch.

A.3. Feature Projection Algorithm

Conceptually, our projection step is analogous to the task of computing the principal component analysis (PCA) from a given low-rank matrix factorisation $\boldsymbol{M} = \boldsymbol{U}\boldsymbol{V}^T$. To accomplish this without explicitly multiplying $\boldsymbol{U}$ with $\boldsymbol{V}^T$, one can:

1. Compute the QR factorisation of $\boldsymbol{V}$, i.e., $\boldsymbol{Q}\boldsymbol{R} = \boldsymbol{V}$
2. Set $\hat{\boldsymbol{U}} := \boldsymbol{U}\boldsymbol{R}^T, \hat{\boldsymbol{V}} := \boldsymbol{Q}$
3. Find the principal components of $\hat{\boldsymbol{U}}$, which are its r leading left singular vectors.

We perform this algorithm in the compressed TT domain, consisting of the following three steps:

i. stack the K TT tensors from one training batch along a new, leading dimension to form a new TT tensor of dimension $D+2$ (with dimension I_0 the number of samples in the batch);

ii. orthogonalise the stack w.r.t. that newly formed core (which can be treated as a matrix, since its shape is $1 \times K \times j_1$);

iii. find the r leading PCA components of that matrix.

For step ii, instead of a single QR factorisation, our version requires a sequence of QR factorisations and core rotations, see Fig. 6 and [23].

A.4. Network Architecture Details

Table 3 shows the exact sequence of layers in our (convolutional) encoder, as well as in our (fully connected) regression heads after the TT bottleneck.

The 3D input scans are mapped to a latent encoding with a convolutional encoder with four layers, *ReLU* activations

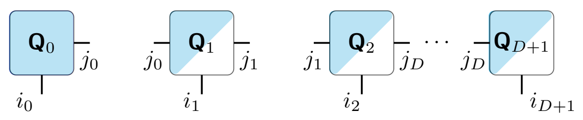


Figure 6: TT decomposition with left-orthogonalised TT-cores $\mathbf{Q}_d$, $d = 1, \ldots, D+1$, such that reshaping any $\mathbf{Q}_d$ into $r_{d-1} \times (I_d r_d)$ yields an orthonormal matrix [23]. Here, $\mathbf{Q}_0$ plays the role of $\hat{\boldsymbol{U}}$ from step 2, while the tensor reconstructed from the remaining cores (and reshaped into an $r_0 \times (I_1 \ldots I_{D+1})$ matrix) plays the role of $\hat{\boldsymbol{V}}^T$.

between layers, and a *sigmoid* after the last layer. All kernels are of size 3×3, resulting in a receptive field of 9^3 voxels since no padding is used.

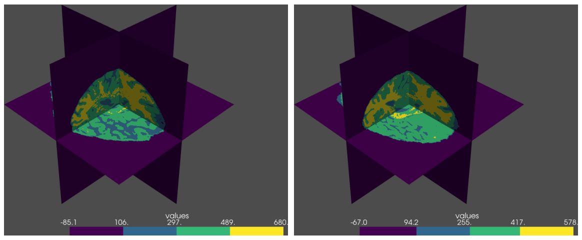
The regression heads are multi-layer perceptrons, with *ReLU* activations and batch normalisation (for OSIC dataset we found turning off batch normalization leads to better results and faster convergence). For BraTS survival time we employ standard regression, whereas for OSIC Pulmonary Fibrosis progression we use quantile regression to predict uncertainty, moreover we first transform the visual information with two layers without *batchnorm* (left side of center column in Table 3), then fuse it with the week number and decode into the final prediction with three more layers (right side of column).

Table 3: Network architectures used in our experiments.

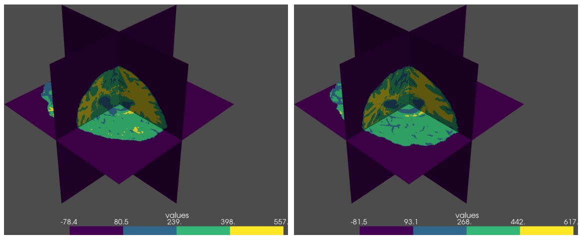
Encoder 3D	Quantile regressor		Regressor
conv(3^3, 5)	dense(500)		dense(100)
ReLU	ReLU		BN
conv(3^3, 15)	dense(100)		ReLU
ReLU	ReLU		dense(20)
conv(3^3, 25)		dense(100)	BN
ReLU		BN	ReLU
conv(3^3, n_{dims})		ReLU	dense(1)
Sigmoid		dense(20)	
		BN	
		ReLU	
		dense(3)	

A.5. Selected Visualizations

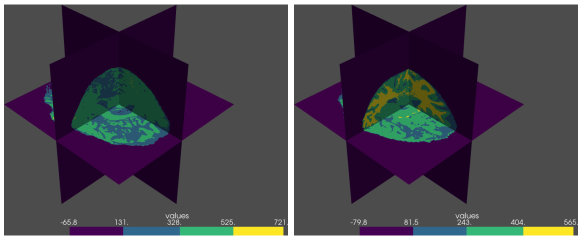
Fig. 7 demonstrates randomly selected predictions of survival time with the trained network on BraTS dataset.



(a) P: 538 days, GT: 616 days (b) P: 590 days, GT: 515 days



(c) P: 748 days, GT: 698 days (d) P: 400 days, GT: 359 days



(e) P: 460 days, GT: 439 days (f) P: 464 days, GT: 486 days

Figure 7: Example predictions for BraTS, at resolution $256 \times 256 \times 256$.