A Diminishing Returns for Fine-grained Lattices

We limit our experiments to $2 \times 2 \times 2 \ldots$ lattices, which are single-celled lattices. Multi-cell lattices require additional linear inequality constraints on the lattice parameters. Even with multilinear interpolation, to ensure decreasing slopes as the input $x[d]$ moves between the cells of the lattice requires $(M-1) \times 2^{S-1}$ linear inequality constraints for a M^S lattice.

B Diminishing Returns for Deep Lattice Networks

To constrain a deep lattice network (DLN) to be convex and increasing for a feature, one must make each layer convex and increasing for any values that are influenced by inputs constrained by those shape constraints. DLN layers are of three types: linear embeddings, 1-d calibrators, or ensembles of lattices. All three layers can be constrained for shape constraints by training with the noted extra linear inequality constraints in this paper. However, if a linear embedding is used before a lattice, for example the W_l matrix in (5), then one must constrain W_l so that each feature only appears in one input to the lattice, because if a feature is part of two inputs to the lattice, then the multilinear interpolation is no longer linear with respect to the feature, which may affect the model's overall convexity or concavity.

C Concave/Convex Lattices with Simplex Interpolation

Simplex interpolation (also known as the Lovász extension [29]), is a more efficient way to linearly interpolate a lattice than multilinear interpolation [24], but unlike multilinear interpolation, the produced function is piecewise linear with $S!$ pieces. And as one varies an input $x[d]$, one crosses to different planar pieces, each which have different slopes defined by the lattice parameters. One must constrain the lattice parameters so that all $S!$ planar pieces one crosses have decreasing slope as $x[d]$ goes from $0 \to 1$ to guarantee a concave $f(x)$. The resulting constraints are illustrated in Fig. 4 for $S = 2$ and $S = 3$. In general one must satisfy $(S-1)2^{(S-2)}$ linear inequality constraints to guarantee concavity for a given input.

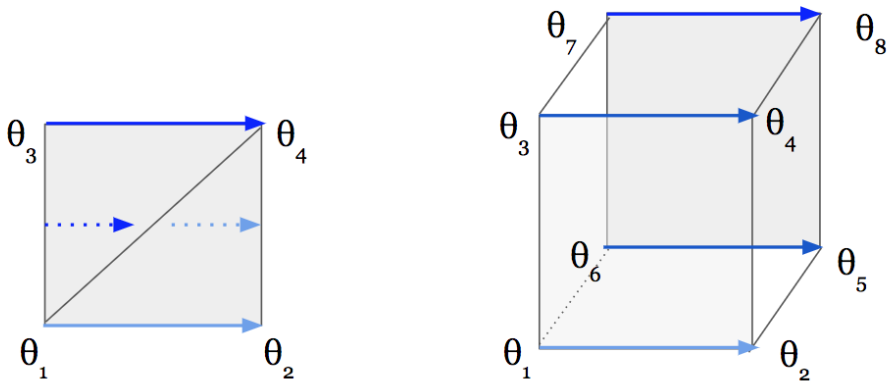

Figure 4: **Left:** A $S = 2$ dimensional lattice. Simplex interpolation interpolates the $S + 1$ lattice parameters for the upper triangle and bottom triangle, fitting a plane to each of the two simplices. Thus the upper triangle has slope $\theta_4 - \theta_3$, and the lower triangle has slope $\theta_2 - \theta_1$. To guarantee convexity from left-to-right, one must guarantee decreasing slopes by satisyfing the linear inequality constraint $\theta_4 - \theta_3 > \theta_2 - \theta_1$. **Right:** For a $S = 3$ dimensional lattice, simplex interpolation is piecewise linear on six simplices, and concavity from left-to-right requires decreasing slopes across adjacent simplices, so four linear inequalities must be satisfied: $\theta_8 - \theta_7 > \theta_4 - \theta_3$, $\theta_8 - \theta_7 > \theta_6 - \theta_5$, $\theta_4 - \theta_3 > \theta_2 - \theta_1$ and $\theta_6 - \theta_5 > \theta_2 - \theta_1$.

Table 4: Simulation

Model	Validation MSE	Test MSE
DNN	1.7×10^{-4}	2.15×10^{-4}
SCNN concave	9.9×10^{-4}	1.19×10^{-3}
SCNN dim. ret.	5.7×10^{-4}	6.25×10^{-4}
RTL dim. ret.	6.0×10^{-4}	5.79×10^{-4}
Cal. Lin. dim. ret.	1.98×10^{-1}	1.93×10^{-1}

D Simulation

To verify that all methods are working as intended, we learned the following smooth function $f : \mathbb{R}^6 \to \mathbb{R}$ that is ceterus paribus concave and increasing in variables $x[1]$, $x[2]$, $x[3]$:

$$f(x) = \frac{x[1]^{0.3} x[2]^{0.6} x[3]^{0.9}}{e^{x[4]+2x[5]+3x[6]}} + \cos\left(x[4]x[5]x[6]\right) \log\left(\sum_{i=1}^{6} x[i]\right). \tag{6}$$

Examples were sampled on a uniform grid on $[0.5, 1.5]^6$, 5 points per dimension, for a total of 5^6 points. They were then randomly shuffled and split into training, validation and test sets and used to evaluate adding concavity and diminishing returns constraints on $x[1], x[2], x[3]$ for various function classes. Results in Table 4 verify that all methods are working as expected, and that the methods with the additional shape constraints show less overfitting as measured by the difference in validation and test MSE.

E Churn Data for Domain Name Pricing Results

We ran follow-up experiments to better characterize the re-training churn [30] of the test results for the DNN, SCNN, and RTL models for the Domain Pricing experiment with respect to re-training. We fixed the hyperparameters to be the ones chosen by the validation process. Then we re-trained each model type 100 times. We sorted the resulting 100 test MSE's for each model type, and then plotted the sorted test MSE's in Figure 5.

The 100 re-trainings differ because for both model types, each re-training experienced a different random shuffle of the training data, and randomized mini-batching of the stochastic gradients. For DNN, there is also randomness from the initialization: the models were initialized using Glorot initialization, that is, the weights are uniform random over some interval, and the biases are initialized to 0. For SCNN, the models were initialized deterministically using the identity matrix and zero for the biases. The RTL models are deterministically initialized based on the training data quantiles, this leads to very stable results across re-trainings.

The DNN results show the most re-training variability. Over the 100 re-trainings, the DNN test MSE ranges from 0.00143 to 2.002, with a mean of 0.27, and median 0.05. For the SCNN, the deterministic regularization and added regularization from the shape constraints does seem to upper-bound the test error, but it still has a large range of .00122 to 0.3161, with better mean of 0.072 and worse median of 0.069.

F TensorFlow Code for Shape-Constrained Neural Network

```
import tensorflow as tf

def scnn_model_fn(features, labels, mode, params):
  inputs = {}
  for input_name in ['xu', 'xc', 'xs']:
    if input_name not in features:
      continue
    input_dim = features[input_name].shape[1].value
```

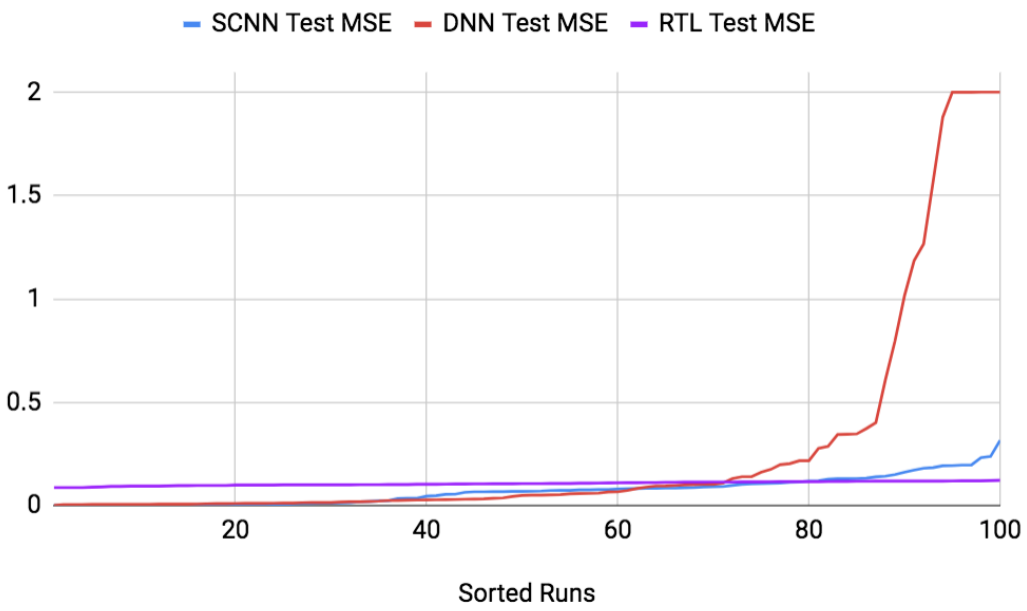

Figure 5: Plots show the 100 sorted test MSE values for 100 different DNN, SCNN and RTL models, where each of the models was re-trained with the (same) hyperparameters chosen on the validation set.

```
    if input_dim > 0:
      inputs[input_name] = tf.feature_column.input_layer(
        {input_name: features[input_name]},
        [tf.feature_column.numeric_column(
          input_name, shape=(input_dim)
        )])

# for simplicity, assume at least one feature is unconstrained
assert 'xu' in inputs
fc = tf.layers.dense

# if `is_convex`, constrain xc and xs to be convex, else concave
g = (tf.nn.relu if params['is_convex']
     else lambda x: -1*tf.nn.relu(-1*x))
h = tf.nn.relu

u_dims = params['u_dims']
z_dims = params['z_dims']
n_layers = len(z_dims)
nonneg = lambda x: tf.maximum(x, 0.0)
bias_init = tf.constant_initializer(0.0)
kernel_init = tf.initializers.identity()

prev_u = inputs['xu']
for i in range(n_layers):
  z_dim = z_dims[i]

  pre = fc(prev_u, z_dim, kernel_initializer=kernel_init,
           bias_initializer=bias_init)

  if 'xs' in inputs:
    inner = fc(prev_u, inputs['xs'].shape[1],
               kernel_initializer=kernel_init,
               bias_initializer=bias_init,
               activation=tf.nn.relu)
```

```
      pre += fc(tf.multiply(inputs['xs'], inner),
                z_dim, kernel_constraint=nonneg,
                kernel_initializer=kernel_init,
                use_bias=False)

    if 'xc' in inputs:
      inner = fc(prev_u, inputs['xc'].shape[1],
                 kernel_initializer=kernel_init,
                 bias_initializer=bias_init)
      pre += fc(tf.multiply(inputs['xc'], inner),
                z_dim, kernel_initializer=kernel_init,
                use_bias=False)

    if i > 0:
      prev_z_dim = z_dims[i-1]
      inner = fc(prev_u, prev_z_dim, kernel_initializer=kernel_init,
                 bias_initializer=bias_init,
                 activation=tf.nn.relu)
      pre += fc(tf.multiply(prev_z, inner), z_dim,
                kernel_constraint=nonneg,
                kernel_initializer=kernel_init,
                use_bias=False)

    if i == n_layers-1:
      z = pre
    else:
      z = g(pre)

    prev_z = z
    if i != n_layers-1:
      prev_u = fc(prev_u, u_dims[i],
                  kernel_initializer=kernel_init,
                  bias_initializer=bias_init,
                  activation=h)

    if mode == tf.estimator.ModeKeys.TRAIN:
      optimizer = tf.train.AdamOptimizer(
        learning_rate=params['learning_rate'])
      loss = tf.losses.mean_squared_error(z, labels)
      train_op = optimizer.minimize(
        loss, global_step=tf.train.get_global_step())
      return tf.estimator.EstimatorSpec(
        mode, loss=loss, train_op=train_op)
    if mode == tf.estimator.ModeKeys.PREDICT:
      predictions = {'predictions': z}
      return tf.estimator.EstimatorSpec(mode, predictions=predictions)

# example usage
scnn_estimator = tf.estimator.Estimator(
  model_fn=scnn_model_fn, params={'u_dims': [50], 'z_dims':[50, 1],
  'learning_rate': 0.1, 'is_convex': False})
```