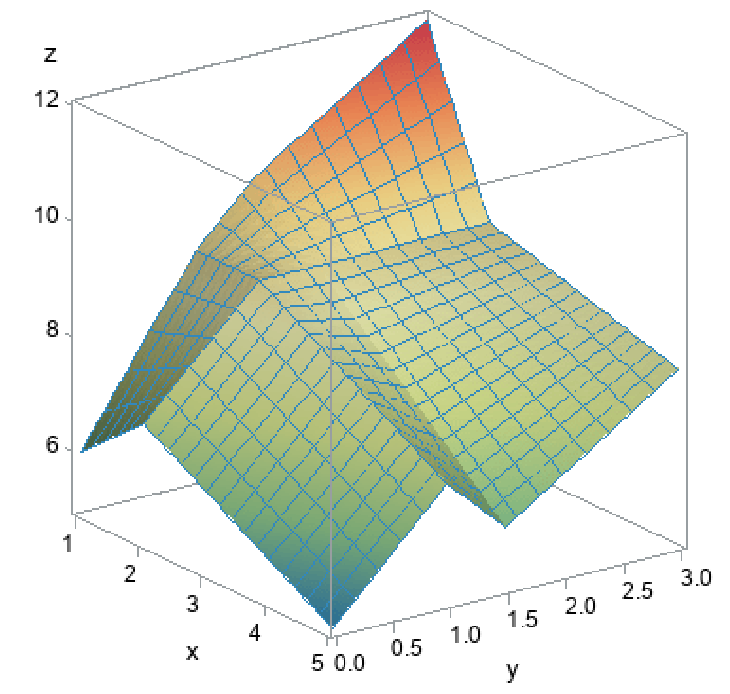
Lattice Models

Lattice Models

Lattice models are a form of piecewise linear interpolated model



- They are defined via a grid over their input variables
- Their parameters are the output values at each grid point
- The output values for input vectors not corresponding to a point of the grid...
- ...Is the linear interpolation of neighboring grid points

They are available in tensorflow via [the tensorflow-lattice module]

Lattice Models

Lattice models:

- Can represent non-linear multivariate functions
- Can be trained by (e.g.) gradient descent

The grid is defined by splitting each input domain into intervals

- The domain of variable x_i is split by choosing a fixed set of n_i "knots"
- ...Of course this leads to scalability issues: we will discuss them later

The lattice parameters are interpretable

They simply represent output values for certain input vectors

- They can be changed with predictable effects
- They can be constrained so that the model behaves in a desired fashion
- If we use hard constraints, we get a guaranteed behavior

Lattice Models and Interpretability

Interpretability is a major open issue in modern ML

It is often a key requirement in industrial applications

- Customers have trouble accepting models that they do not understand
- Sometimes you are legally bound to provide motivations

There are two main ways to achieve interpretability

The first is using a model that is inherently interpretable

- There are a few examples of this: linear regression, DTs, (some) SVMs, rules...
- Lattice models fall into this class

The second approach is computing a posteriori explanations

- E.g. approximate linear explanations
- ...Such as in the LIME or SHAP approaches

Implementing a Lattice Model

The first step for implementing a lattice model is choosing the lattice size

```
In [2]: lattice_sizes = [4] * 2 + [2] * 4
```

- We are using 4 knots for numeric inputs and 2 knots for the boolean inputs

Next, we need to split the individual input columns

```
In [3]: tr_ls = [tr_sc[c] * (s-1) for c, s in zip(dt_in_c, lattice_sizes)]
        val_ls = [val_sc[c] * (s-1) for c, s in zip(dt_in_c, lattice_sizes)]
        ts_ls = [ts_sc[c] * (s-1) for c, s in zip(dt_in_c, lattice_sizes)]
```

- This step is required by the tensorflow-lattice API
- We also scale the input to the range $[0, n_{knots} - 1]$
- ...Since this is the expected convention for the considered API

Implementing a Lattice Model

The we build the symbolic tensors for the model input

In [4]:
```python
mdl_inputs = []
for cname in dt_in_c:
    cname_in = layers.Input(shape=[1], name=cname)
    mdl_inputs.append(cname_in)
```

- We have one tensor per input column

Finally we can build our lattice model

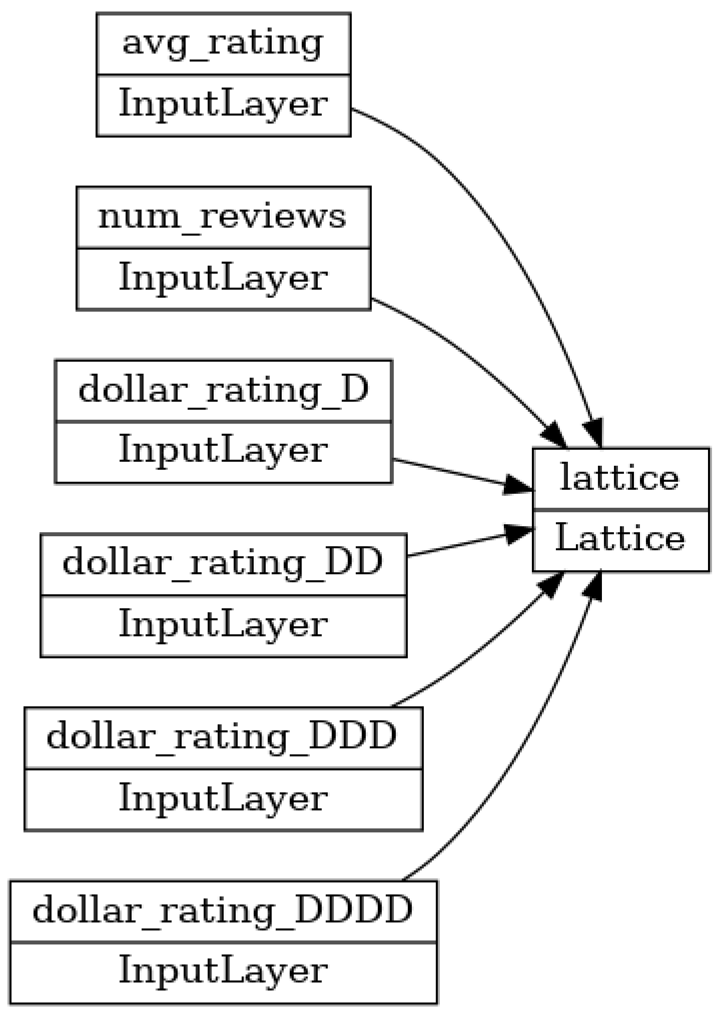
In [5]:
```python
import tensorflow_lattice as tfl

mdl_out = tfl.layers.Lattice(lattice_sizes=lattice_sizes,
    output_min=0, output_max=1, name='lattice',
)(mdl_inputs)

lm = keras.Model(mdl_inputs, mdl_out)
```

Implementing a Lattice Model

We can plot the model structure

In [6]:
```
util.plot_nn_model(lm, show_layer_activations=True, show_layer_names=True, show_shapes=False
```

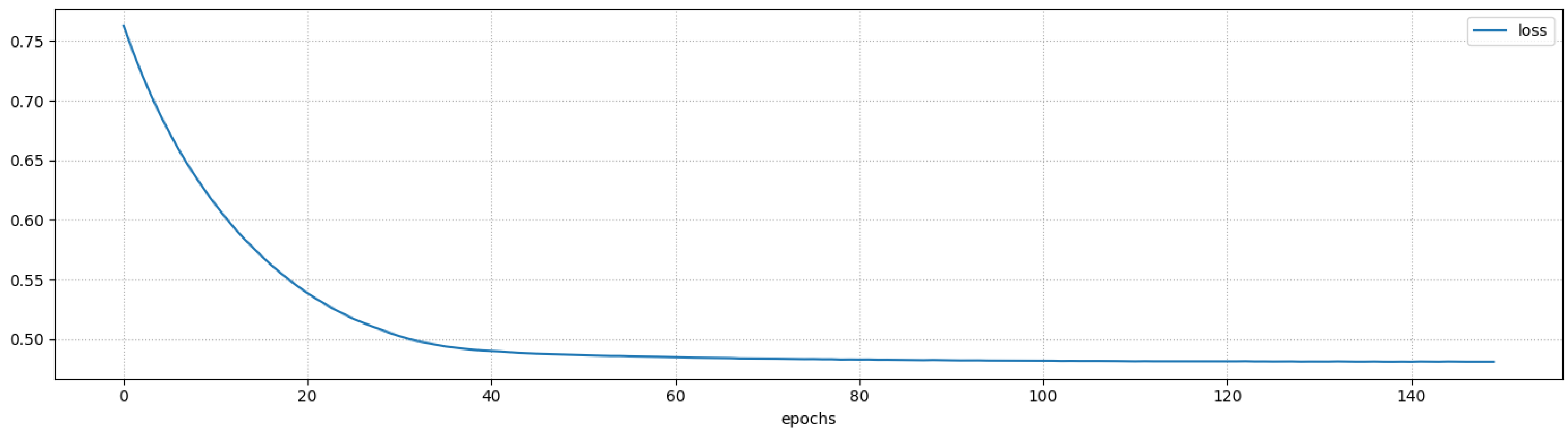
Out[6]:



Implementing a Lattice Model

We can train the model as usual

```
In [7]: history = util.train_nn_model(lm, tr_ls, tr_sc['clicked'], loss='binary_crossentropy', batch
        util.plot_training_history(history, figsize=figsize)
```




```
Final loss: 0.4809 (training)
```

Lattice Model Evaluation

A large enough lattice model can peform as well as a Deep Network

Let's see the performance in terms of AUC

In [8]:
```
pred_tr2 = lm.predict(tr_ls, verbose=0)
pred_val2 = lm.predict(val_ls, verbose=0)
pred_ts2 = lm.predict(ts_ls, verbose=0)
auc_tr2 = roc_auc_score(tr_sc['clicked'], pred_tr2)
auc_val2 = roc_auc_score(val_sc['clicked'], pred_val2)
auc_ts2 = roc_auc_score(ts_sc['clicked'], pred_ts2)
print(f'AUC score: {auc_tr2:.2f} (training), {auc_val2:.2f} (validation), {auc_ts2:.2f} (tes
```

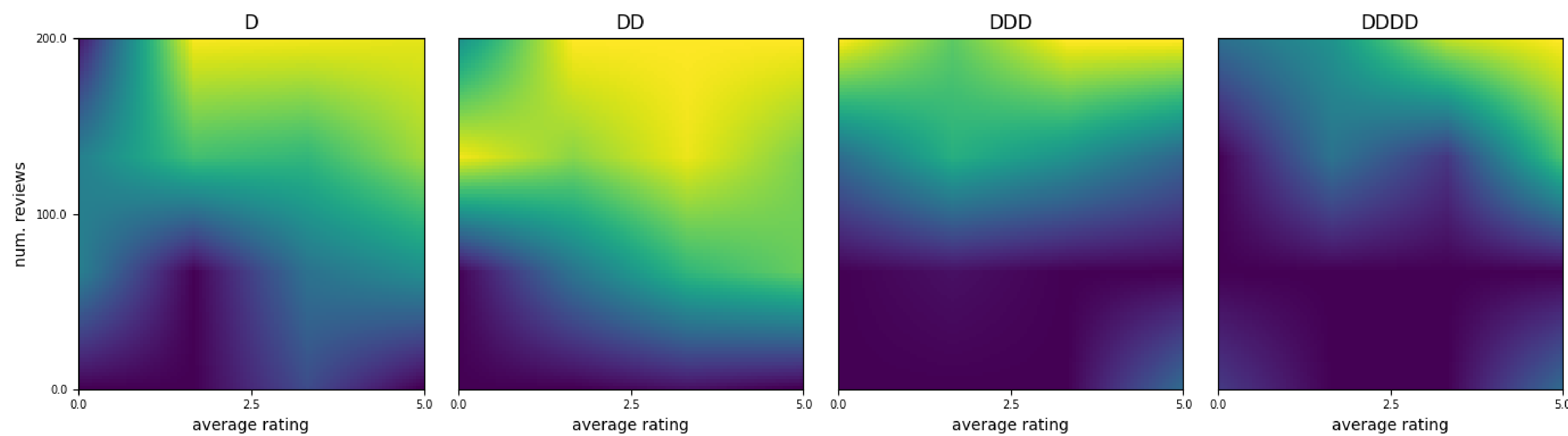
```
AUC score: 0.82 (training), 0.80 (validation), 0.76 (test)
```

- It is indeed comparable to that of the deep MLP
- ...Also in the fact that it works poorly on the test distribution

Lattice Model Evaluation

...And in fact the behavior is just as bad as the MLP (or worse)

In [9]:
```
lscale = scale / [s-1 for s in lattice_sizes[:2]]
util.plot_ctr_estimation(lm, lscale, split_input=True, figsize=figsize)
```



- The expected monotonicity constraints are still violated
- There are still many mistakes for less represented areas of the input space

Calibration

Calibration

Let's start fixing some of the outstanding issues

In a lattice model, the number of grid points is given by:

$$n = \prod_{i=1}^{m} n_i$$

- ...Hence the parameter number scales exponentially with the number of inputs
- So that modeling complex non-linear function seems to come at a steep cost

Scalability issues can be mitigated via two approaches:

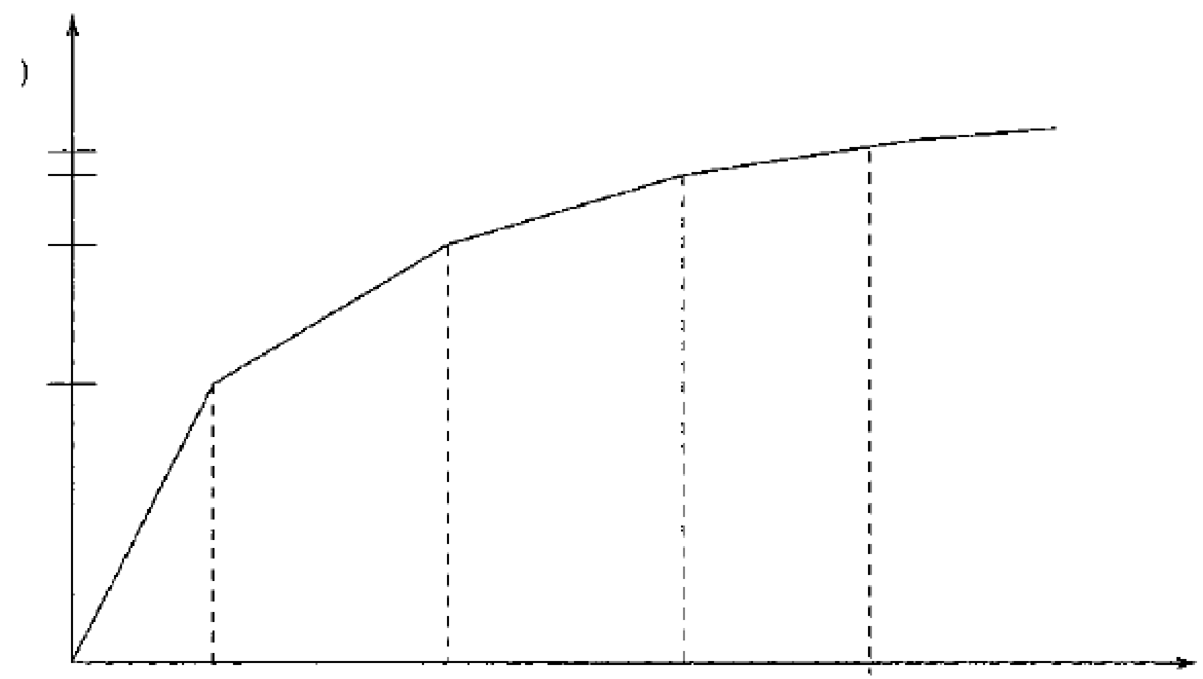
- Ensembles of small lattices (we will not cover this one)
- Applying a calibration step to each input variable individually

We will focus on this latter approach

Calibration for Numeric Inputs

Calibration for numeric attributes...

...Consists in applying a piecewise linear transformation to each input

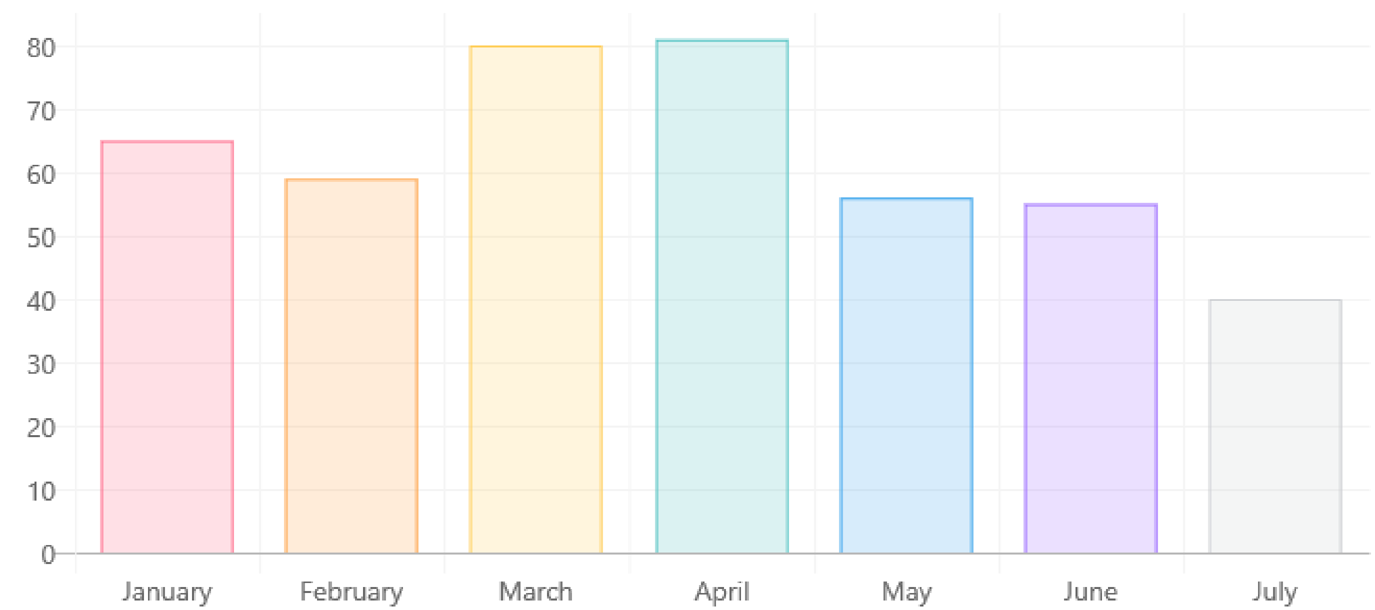


- This is essentially a 1-D lattice
- Calibration parameters are the function values at all knots

Calibration for Categorical Inputs

Calibration for categorical inputs...

...Consists in applying a map:



- Categorical inputs must be encoded as integers
- Each input value is mapped to an output value
- The parameters are the map output values

How is this related to scalability?

About Calibration

With this approach:

- We make each input more complicated
- ...Which allows us to make the lattice model simple

Calibration enables the use of fewer knots in the lattice

E.g. say we are aiming for 5 grid values per attribute, with 2 attributes

- With 5 knots per layer an single lattice: $5 \times 5 = 25$ parameters
- With 5 calibration knots + 2 lattice knots: $5 \times 2 + 2 \times 2 = 14$

We do not get the same level of flexibility, but we get close

- Additionally, we tend to get more regular results
- ...Since we have more bias and less variance

This might be an advantage for out-of-distribution generalization

About Calibration

Calibration enables using categorical inputs without a one-hot encoding

- The calibration map is almost equivalent
- ...Since it enables mapping each category to an arbitrary numeric value

Once again, we gain in terms of lattice parameters:

- E.g. 5 categories, no calibration: $2 \times 5 = 10$ parameters
- Whereas with calibration: $5 + 2 = 7$ parameters

To use calibration, we start by adjusting our lattice size

- We will use just two knots per dimension
- ...And we replace the 4 one-hot variable with a single one

In [10]:
```
lattice_sizes2 = [2] * 3
```

Preparing the Input

Then, we need to encode our categorical input using integers

We start by converting our string data input pandas categories

In [11]:
```
tr_sc2 = tr_s.copy()
tr_sc2['dollar_rating'] = tr_sc2['dollar_rating'].astype('category')
tr_sc2[:3]
```

Out[11]:

	avg_rating	num_reviews	dollar_rating	clicked
0	0.785773	0.61	DDDD	1
1	0.785773	0.61	DDDD	0
2	0.785773	0.61	DDDD	0

We can check how the categories are mapped into integer codes:

In [12]:
```
tr_sc2['dollar_rating'].cat.categories
```

Out[12]:
```
Index(['D', 'DD', 'DDD', 'DDDD'], dtype='object')
```

- The codes are are the positional indexes of the strings

Preparing the Input

Now we replace the category data with the codes themselves

```
In [13]: tr_sc2['dollar_rating'] = tr_sc2['dollar_rating'].cat.codes
         tr_sc2['dollar_rating'][:3]

Out[13]: 0    3
         1    3
         2    3
         Name: dollar_rating, dtype: int8
```

...And we apply the same treatment to the validation and test set:

```
In [14]: val_sc2 = val_s.copy()
         val_sc2['dollar_rating'] = val_sc2['dollar_rating'].astype('category').cat.codes

         ts_sc2 = ts_s.copy()
         ts_sc2['dollar_rating'] = ts_sc2['dollar_rating'].astype('category').cat.codes
```

Piecewise Linear Calibration

We use `PWLCalibration` objects for all numeric inputs

```
In [15]: avg_rating = layers.Input(shape=[1], name='avg_rating')
         avg_rating_cal = tfl.layers.PWLCalibration(
             input_keypoints=np.quantile(tr_sc2['avg_rating'], np.linspace(0, 1, num=20)),
             output_min=0.0, output_max=lattice_sizes2[0] - 1.0, name='avg_rating_cal'
         )(avg_rating)

         num_reviews = layers.Input(shape=[1], name='num_reviews')
         num_reviews_cal = tfl.layers.PWLCalibration(
             input_keypoints=np.quantile(tr_sc['num_reviews'], np.linspace(0, 1, num=20)),
             output_min=0.0, output_max=lattice_sizes2[1] - 1.0, name='num_reviews_cal'
         )(num_reviews)
```

- The knot values are learnable parameters
- ...But their positions are fixed

A good choice consist in using distribution quantiles

E.g. for five knots: the 0-th, 25-th, 50-th, 75-th, 100-th percentile

Categorical Calibration

We use `CategoricalCalibration` objects for the categorical input

In [16]:
```
dollar_rating = layers.Input(shape=[1], name='dollar_rating')
dollar_rating_cal = tfl.layers.CategoricalCalibration(
    num_buckets=4,
    output_min=0.0, output_max=lattice_sizes2[2] - 1.0,
    name='dollar_rating_cal'
)(dollar_rating)
```

- We use one "bucket" for each possible category

Building the Calibrated Lattice Model

We can now build the lattice model

...Using distinct input tensors for each input (as we did before)

In [17]:
```
lt_inputs2 = [avg_rating_cal, num_reviews_cal, dollar_rating_cal]

mdl_out2 = tfl.layers.Lattice(
    lattice_sizes=lattice_sizes2,
    output_min=0, output_max=1, name='lattice',
)(lt_inputs2)

mdl_inputs2 = [avg_rating, num_reviews, dollar_rating]
lm2 = keras.Model(mdl_inputs2, mdl_out2)
```

We can compare the number of parameters

In [18]:
```
print(f'#Parameters in the original lattice: {sum(len(w) for w in lm.get_weights())}')
print(f'#Parameters in the new lattice: {sum(len(w) for w in lm2.get_weights())}')
```

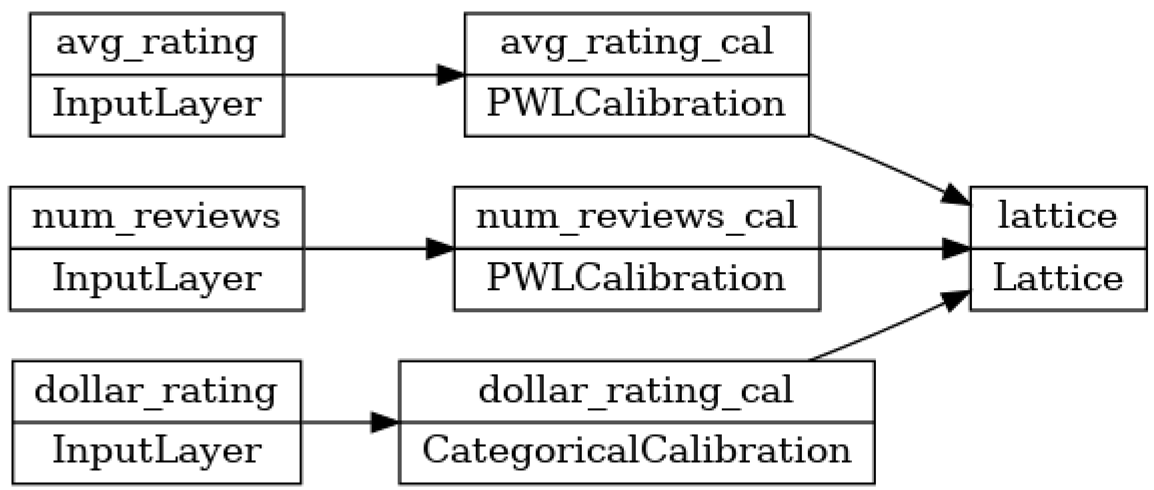
```
#Parameters in the original lattice: 256
#Parameters in the new lattice: 52
```

Building the Calibrated Lattice Model

Let's see which kind of architecture we have now:

In [19]:
```
util.plot_nn_model(lm2, show_layer_activations=True, show_layer_names=True, show_shapes=Fals
```

Out[19]:



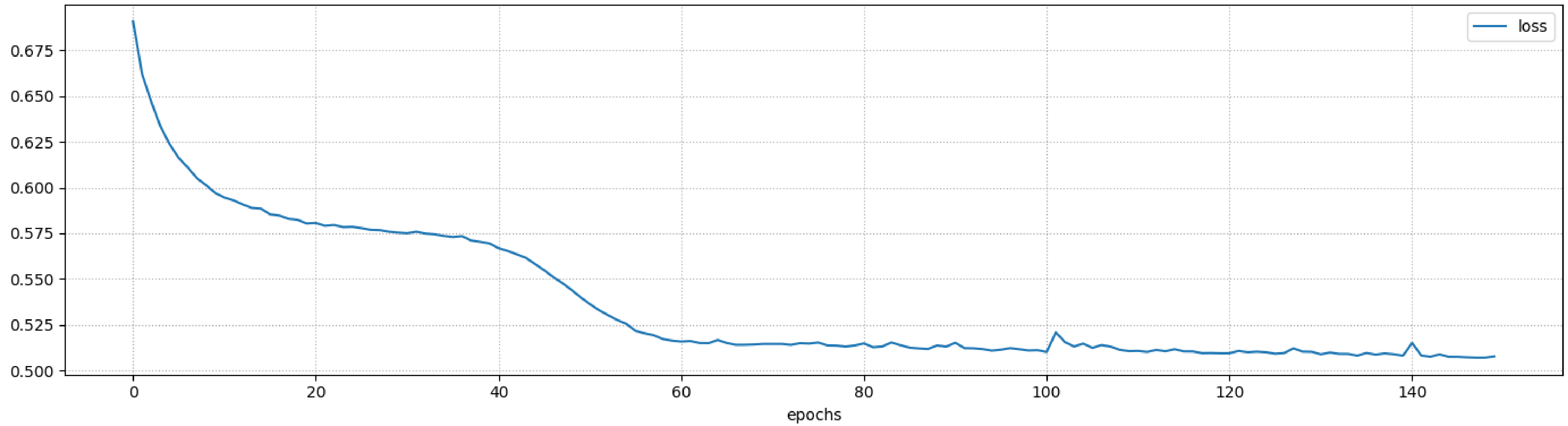
Before we can train it, we need to split our dataset columns

In [20]:
```
tr_ls2 = [tr_sc2[c] for c in dt_in]
val_ls2 = [val_sc2[c] for c in dt_in]
ts_ls2 = [ts_sc2[c] for c in dt_in]
```

Training the Calibrated Lattice

We can train the new model as usual

```
In [21]: history = util.train_nn_model(lm2, tr_ls2, tr_sc['clicked'], loss='binary_crossentropy', bat
         util.plot_training_history(history, figsize=figsize)
```




```
Final loss: 0.5078 (training)
```

Evaluating the Calibrated Lattice

...And finally we can evaluate the results

In [22]:
```
pred_tr3 = lm2.predict(tr_ls2, verbose=0)
pred_val3 = lm2.predict(val_ls2, verbose=0)
pred_ts3 = lm2.predict(ts_ls2, verbose=0)
auc_tr3 = roc_auc_score(tr_s['clicked'], pred_tr3)
auc_val3 = roc_auc_score(val_s['clicked'], pred_val3)
auc_ts3 = roc_auc_score(ts_s['clicked'], pred_ts3)
print(f'AUC score: {auc_tr3:.2f} (training), {auc_val3:.2f} (validation), {auc_ts3:.2f} (tes
```

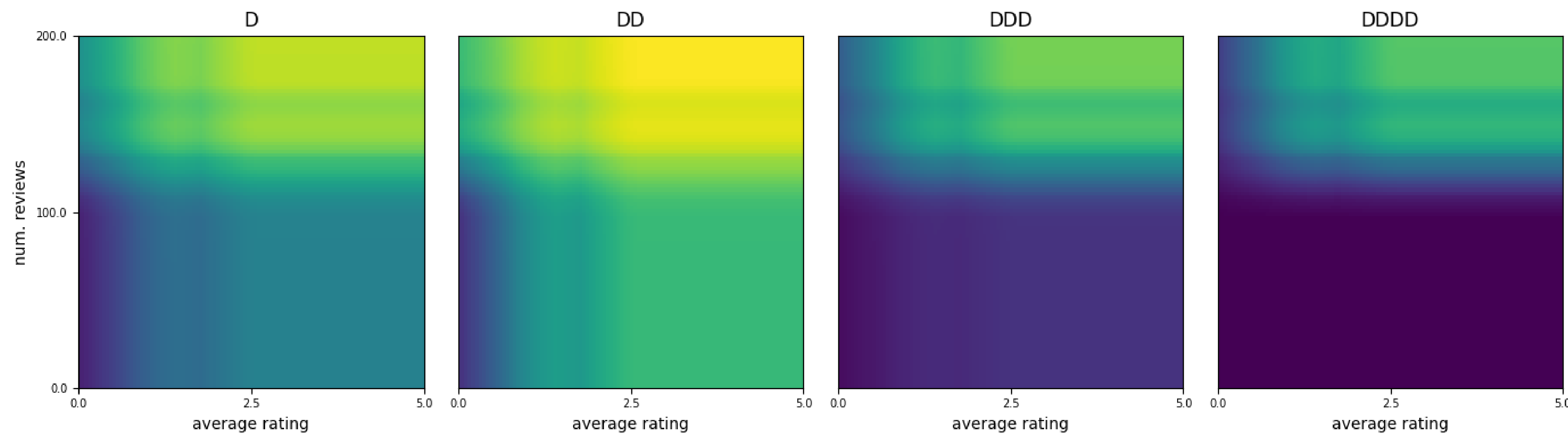
```
AUC score: 0.80 (training), 0.80 (validation), 0.80 (test)
```

- The performance is on par with the original one
- ...Except on the test set, where it works much better

Inspecting the Calibrated Lattice

We can inspect the learned function visually to get a better insight

```
In [23]: util.plot_ctr_estimation(lm2, scale, split_input=True, one_hot_categorical=False, figsize=fi
```

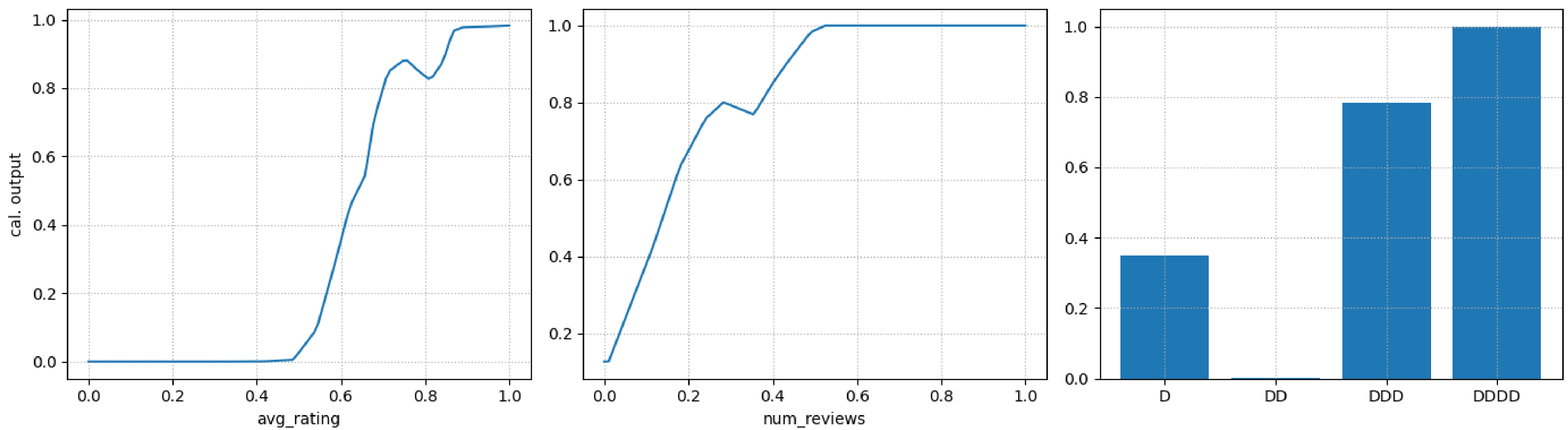


- The structure follows a (piecewise linear) "tartan pattern"
- This is particularly evident now, since we use just two knots per dimension

Inspecting the Calibrated Lattice

It is useful to inspect the calibration layers

In [24]:
```
calibrators = [keras.Model(mdl_inputs2[i], lt_inputs2[i]) for i in range(3)]
util.plot_ctr_calibration(calibrators, scale, figsize=figsize)
```



- The learned calibration functions violate the expected monotonicities
- ...Meaning that we still have one problem to solve

Shape Constraints

Shape Constraints

Lattice models are well suited to deal with shape constraints

Shape constraints are restrictions on the input-output function, such as:

- Monotonicity (e.g. "the output should grow when an input grows")
- Convexity/concavity (e.g. "the output should be convex w.r.t. an input")

Shape constraints are very common in industrial applications

Some examples:

- Reducing the price will raise the sales volume (monotonicity)
- Massive price reductions will be less effective (diminishing returns)
- Too low/high temperatures will lead to worse bakery products (convexity)

We can use them to fix our calibration issues

Shape Constraints

Shape constraints translate into constraints on the lattice parameters

- Let $\theta_{i,k,\bar{i},\bar{k}}$ be the parameter for the k-th knot of input i...
- ...While all the remaining attributes and knots (i.e. $\bar{i}$ and $\bar{k}$) are fixed

Then (increasing) monotonicity translates to:

$$\theta_{i,k,\bar{i},\bar{k}} \leq \theta_{i,k+1,\bar{i},\bar{k}}$$

- I.e. all else being equal, the lattice value at the grid points must be increasing
- Decreasing monotonicity is just the inverse

Then convexity translates to:

$$\left(\theta_{i,k+1,\bar{i},\bar{k}} - \theta_{i,k,\bar{i},\bar{k}}\right) \leq \left(\theta_{i,k+2,\bar{i},\bar{k}} - \theta_{i,k+1,\bar{i},\bar{k}}\right)$$

- I.e. all else being equal, the adjacent parameter differences should increase

Monotonicity and Smoothness

We can expect a monotonic effect of the average rating

I.e. Restaurants with a high rating will be clicked more often

In [25]:
```
avg_rating2 = layers.Input(shape=[1], name='avg_rating')
avg_rating_cal2 = tfl.layers.PWLCalibration(
    input_keypoints=np.quantile(tr_s['avg_rating'], np.linspace(0, 1, num=20)),
    output_min=0.0, output_max=lattice_sizes2[0] - 1.0,
    monotonicity='increasing',
    kernel_regularizer=('hessian', 0, 1),
    name='avg_rating_cal'
)(avg_rating2)
```

In addition to monotonicity, we use a Hessian reguralizer:

- This is a regularization term that penalizes the second derivative
- ...Thus making the calibrator more linear
- The two parameters are an L1 weight and L2 weights

Diminishing Returns

We can expect a diminishing returns from the number of reviews

- I.e. 200 reviews will be linked to much more clicks than 10 reviews
- ...But only a little more than 150 reviews

In [26]:
```
num_reviews2 = layers.Input(shape=[1], name='num_reviews')
num_reviews_cal2 = tfl.layers.PWLCalibration(
    input_keypoints=np.quantile(tr_s['num_reviews'], np.linspace(0, 1, num=20)),
    output_min=0.0, output_max=lattice_sizes2[1] - 1.0,
    monotonicity='increasing',
    convexity='concave',
    kernel_regularizer=('wrinkle', 0, 1),
    name='num_reviews_cal'
)(num_reviews2)
```

By coupling monotonicity with concavity we enforce diminishing returns

- We also use the "wrinkle" reguralizer, which penalizes the third derivative
- ...Thus making the calibration function smoother

Partial Orders on Categories

We can expect more clicks for reasonably priced restaurants...

...At least compared to very cheap and very expensive ones

In [27]:
```
dollar_rating2 = layers.Input(shape=[1], name='dollar_rating')
dollar_rating_cal2 = tfl.layers.CategoricalCalibration(
    num_buckets=4,
    output_min=0.0, output_max=lattice_sizes2[2] - 1.0,
    monotonicities=[(0, 1), (3, 1)],
    name='dollar_rating_cal'
)(dollar_rating2)
```

On categorical attributes, we can enforce partial order constraints

- Each (i, j) pair translates into an inequality $\theta_i \leq \theta_j$
- Here we specify that "D" and "DDDD" will tend to have fewer clicks than "DD"

Lattice Model with Shape Constraints

Then we can build the actual lattice model

In [28]:
```
lt_inputs3 = [avg_rating_cal2, num_reviews_cal2, dollar_rating_cal2]

mdl_out3 = tfl.layers.Lattice(
    lattice_sizes=lattice_sizes2,
    output_min=0, output_max=1,
    monotonicities=['increasing'] * 3, name='lattice',
)(lt_inputs3)

mdl_inputs3 = [avg_rating2, num_reviews2, dollar_rating2]
lm3 = keras.Model(mdl_inputs3, mdl_out3)
```

If we specify monotonicities in the calibration layers

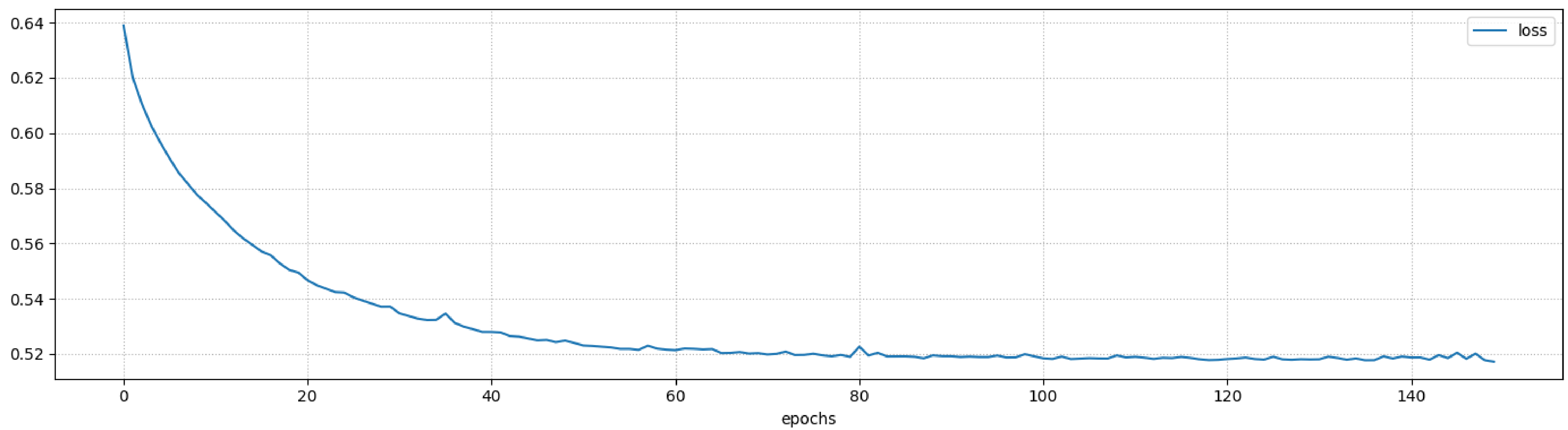
...Then the lattice must be monotone, too

- Otherwise, we risk loosing all our benefits
- Lattice monotonicities are always set to "increasing"
- ...Since we just want to preserve monotonicities from the calibration layers

Lattice Model with Shape Constraints

Let's train the constrained model

In [29]:
```
history = util.train_nn_model(lm3, tr_ls2, tr_sc['clicked'], loss='binary_crossentropy', bat
util.plot_training_history(history, figsize=figsize)
```




```
Final loss: 0.5172 (training)
```

Evaluating the Lattice Model with Shape Constraints

In [30]:

```
pred_tr4 = lm3.predict(tr_ls2, verbose=0)
pred_val4 = lm3.predict(val_ls2, verbose=0)
pred_ts4 = lm3.predict(ts_ls2, verbose=0)
auc_tr4 = roc_auc_score(tr_s['clicked'], pred_tr3)
auc_val4 = roc_auc_score(val_s['clicked'], pred_val3)
auc_ts4 = roc_auc_score(ts_s['clicked'], pred_ts3)
print(f'AUC score: {auc_tr4:.2f} (training), {auc_val4:.2f} (validation), {auc_ts4:.2f} (tes
```

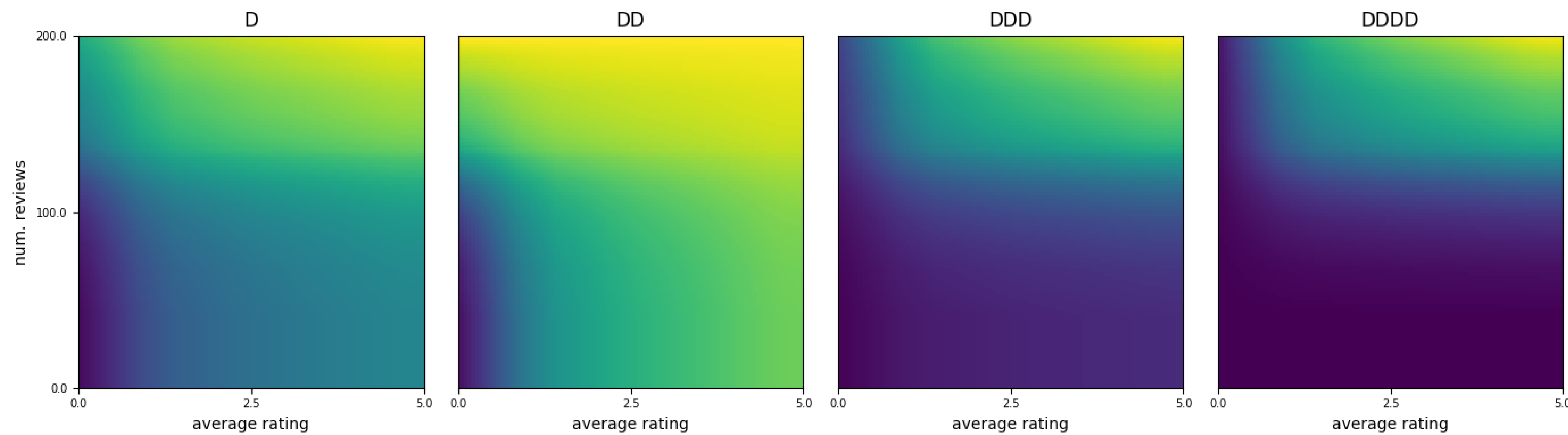
```
WARNING:tensorflow:5 out of the last 1262 calls to <function Model.make_predict_function.<l
ocals>.predict_function at 0x7f665421f1a0> triggered tf.function retracing. Tracing is expe
nsive and the excessive number of tracings could be due to (1) creating @tf.function repeat
edly in a loop, (2) passing tensors with different shapes, (3) passing Python objects inste
ad of tensors. For (1), please define your @tf.function outside of the loop. For (2), @tf.f
unction has reduce_retracing=True option that can avoid unnecessary retracing. For (3), ple
ase refer to https://www.tensorflow.org/guide/function#controlling_retracing and https://ww
w.tensorflow.org/api_docs/python/tf/function for  more details.
AUC score: 0.80 (training), 0.80 (validation), 0.80 (test)
```

The results are on par with the previous ones

Inspecting the Calibrated Lattice

Let's inspect the learned function

```
In [31]: util.plot_ctr_estimation(lm3, scale, split_input=True, one_hot_categorical=False, figsize=fi
```

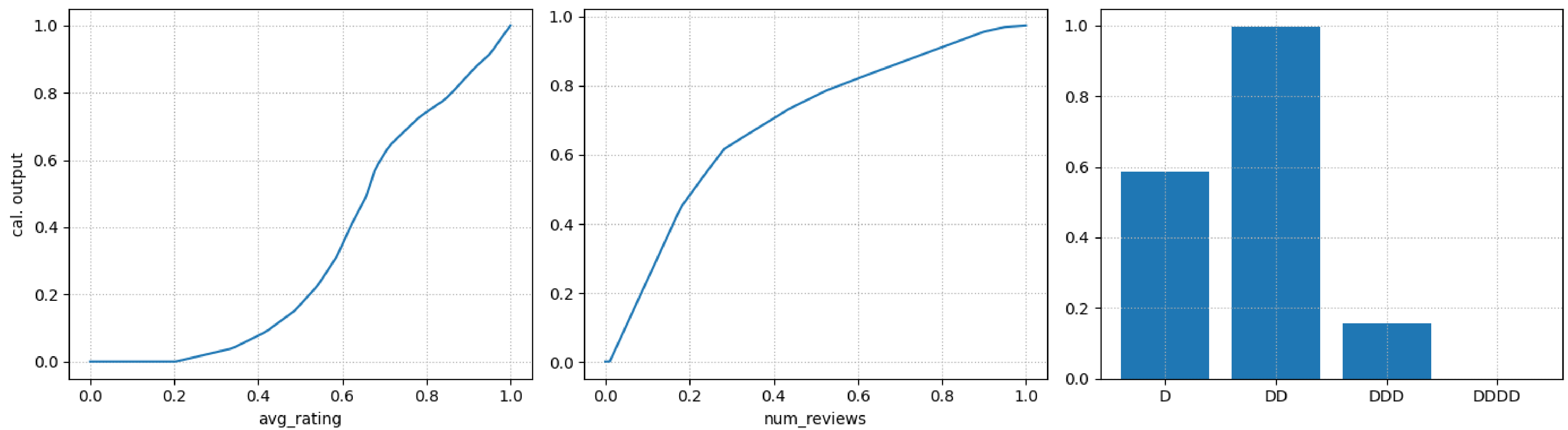


- All monotonicities are respected, the functions are much more regular
- Tartan-pattern apart, they closely match our ground truth

Inspecting the Calibrated Lattice

The most interesting changes will be in the calibration functions

In [32]:
```
calibrators = [keras.Model(mdl_inputs3[i], lt_inputs3[i]) for i in range(3)]
util.plot_ctr_calibration(calibrators, scale, figsize=figsize)
```



- Indeed, all monotonicities are respected
- The `avg_rating` regularizer is more linear
- The `num_reviews` one is convex and smooth

Considerations

Lattice models are little known, but they can be very useful

- They are interpretable
- Customer react (very) poorly to violation of known properties

In general, shape constraints are related to the topic of reliability

- I.e. the ability of a ML model to respect basic properties
- ...Especially in areas of the input space not well covered by the training set

Reliability is a very important topic for many applications of AI methods

Calibration is not restricted to the lattice input

- Indeed, we can add a calibration layer on the output as well
- ...So that we gain flexibility at a cost of a few more parameters

Considerations

Shape constraints are not unique to lattice models

It is possible to enforce monotonicity in linear models:

- We just need make the corresponding weight is non-negative/non-positive
- E.g. we can just clip (project) the weights after each update
- In Tensorflow/Keras, this is implemented by weight constraints

It is possible to enforce monotonicity in decision trees:

- E.g. we can discard all splits that violate monotonicity
- This is implemented in XGBoost

Convexity shape constraints are still supported onlyby lattices