Domain Knowledge in Click-Through Rate Prediction

Introducing the Problem

BBS
MULTIPLYING OPPORTUNITIES

Click-Through Rate Prediction

Let's revisit our click-through rate prediction problem

The use case was about a Tripadvisor-like system

- Given information about restaurants
- and about where users clicked or not on the restaurant cards
- Our goal is to estimate the probability of clicking

```
In [6]: tr.iloc[:4]
```

Out[6]:

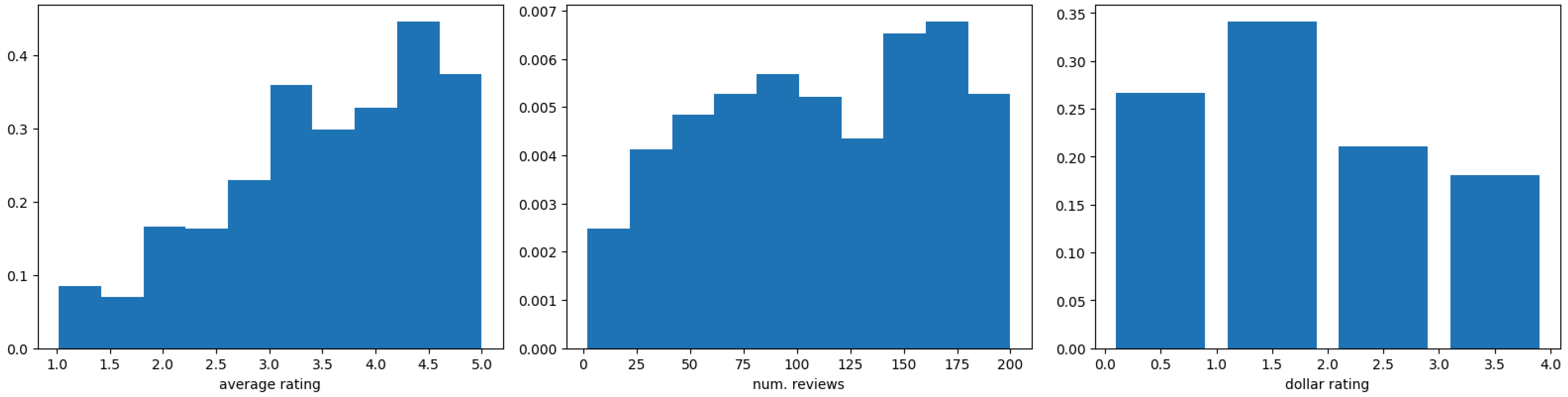
	avg_rating	num_reviews	dollar_rating	clicked
0	3.927976	122.0	DDDD	1
1	3.927976	122.0	DDDD	0
2	3.927976	122.0	DDDD	0
3	4.329771	122.0	DDDD	1

The intention is to replace the current recommendation algorithm with a better one

Data Distribution

We already inspected the distributon of the training data

In [7]:
```
util.plot_ctr_distribution(tr, figsize=figsize, nbins=10)
```

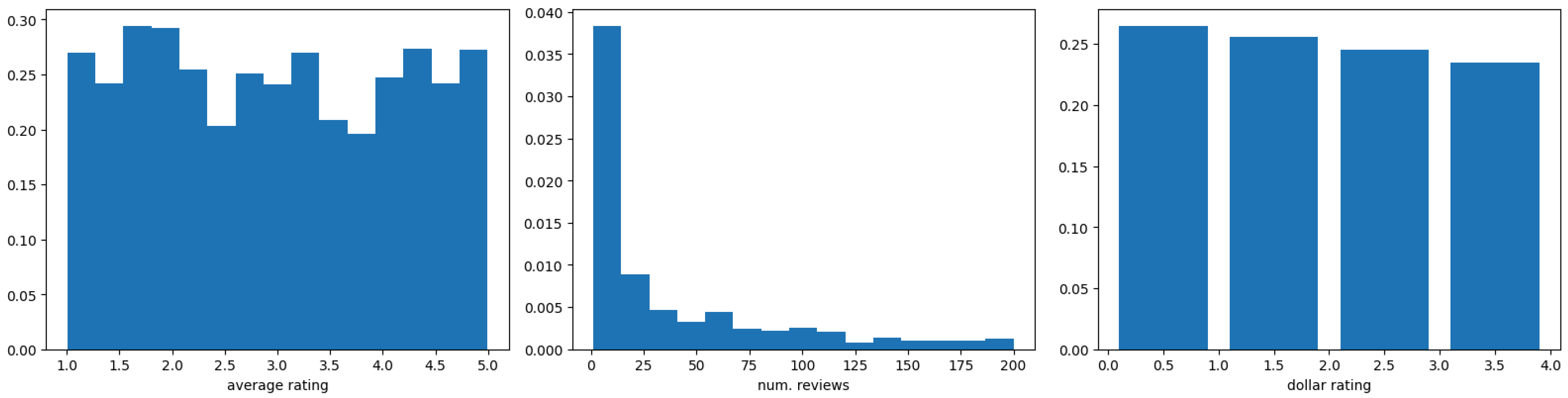


- ...But mentioned that we cheated a bit in our evaluation
- In particular, we used for testing data that was actually meant for validation

Data Distribution

We will now look at the actual test data

In [10]:
```
util.plot_ctr_distribution(ts, figsize=figsize)
```



The distribution is significantly different from the training one!

Why do you think is this the case?

Selection Bias

The reason is selection bias

- The training data comes from the current recommendation algorithm
- ...Which tends to strongly favor certain restaurants w.r.t. others

So, middle priced, well-rated restaurants appear to be over-represented

Selection bias is a very common issue

It tends to arise whenever there is a decision process in place:

- Recommendation systems
- Marketing
- Organ transplant programs
- Operation of industrial machines

Countering Selection Bias

Countering selection bias is not easy

- There are a few available approaches
- ...But they all assume (some) knowledge of the real world test distribution
- ...Which in many cases is not directly accessible

This often an underestimated issue in ML

- It's well know that the training data should be representative of the test data
- ...But's even more important that the test data is representative of the real world data

In our case study, we are in controlled conditions

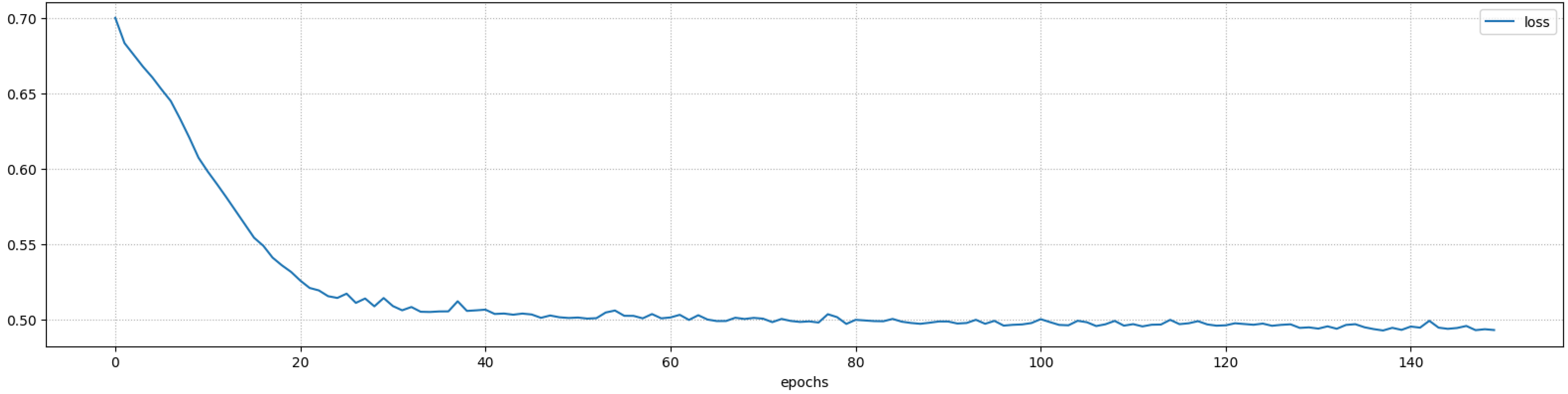
- This means we have access to the test data for the real world distribution
- ...And we'll be able to check how well one particular technique can work

Test Performance of Our Previous Solution

Let's check tbe "real world" performance of our previous solution

First, we tr-train our 3-layer neural network

In [11]:
```
nn = util.build_nn_model(input_shape=len(dt_in_c), output_shape=1, hidden=[16, 8, 8], output_activation='sigmoid')
history = util.train_nn_model(nn, tr_sc[dt_in_c], tr_sc['clicked'], loss='binary_crossentropy', batch_size=32, epochs=150)
util.plot_training_history(history, figsize=figsize, display_loss_curve=True)
```




```
Final loss: 0.4928 (training)
```

Test Performance of Our Previous Solution

Then we check again its performance in terms of ROC-AUC

In [12]:
```
pred_tr = nn.predict(tr_sc[dt_in_c], verbose=0)
pred_val = nn.predict(val_sc[dt_in_c], verbose=0)
pred_ts = nn.predict(ts_sc[dt_in_c], verbose=0)
auc_tr = roc_auc_score(tr_sc['clicked'], pred_tr)
auc_val = roc_auc_score(val_sc['clicked'], pred_val)
auc_ts = roc_auc_score(ts_sc['clicked'], pred_ts)
print(f'AUC score: {auc_tr:.2f} (training), {auc_val:.2f} (validation), {auc_ts:.2f} (test)')
```

```
AUC score: 0.81 (training), 0.80 (validation), 0.76 (test)
```

- The model is performing well on botht the training and validation data
- ...But there is a significant performance drop when moving to the test data

How can we mitigate this, given that the test data is not really accessible?

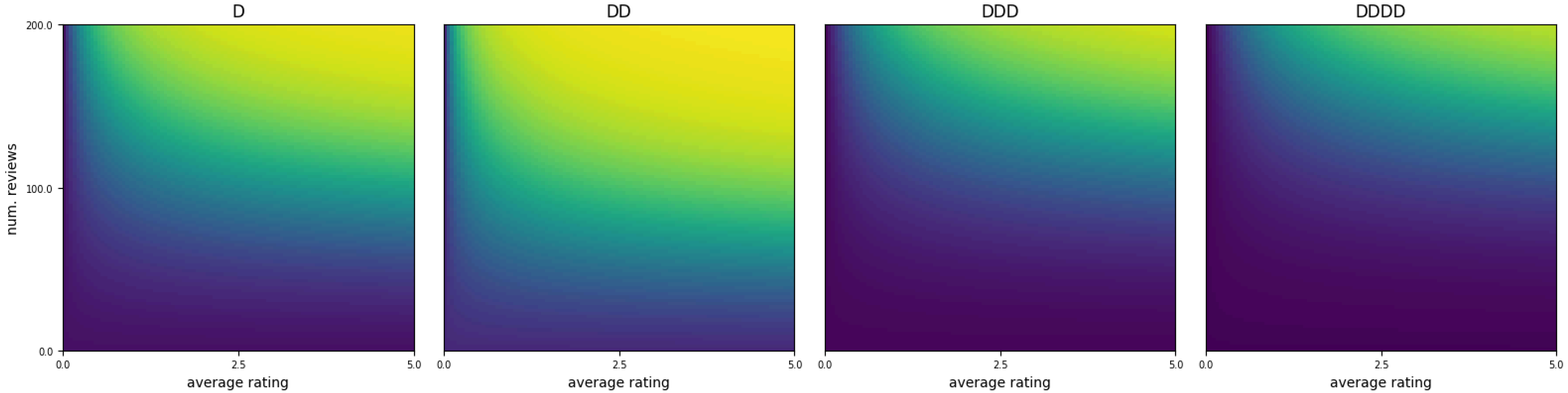
Domain Knowledge in Click-Through Rate Prediction

Mitigating Selection Bias via Domain Knowledge

Ground Truth Click-Through Rate

Let's check again the ground-truch click through rate

```
In [13]: util.plot_ctr_truth(figsize=figsize)
```



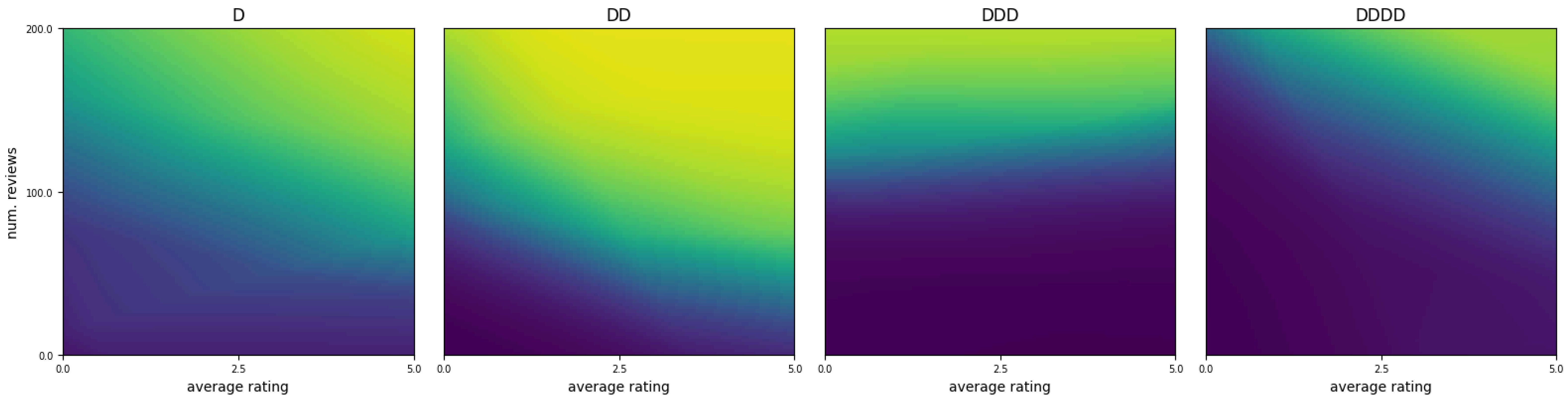
A domain expert may have several expectations on this function

- E.g. `average rating` and `num. reviews` cannot decrease the click-through rate

Checking the Learned Response Surfaces

There is not guarantee that the learned response surface satisfies these properties

In [14]:
```
util.plot_ctr_estimation(nn, scale, figsize=figsize)
```



- In fact, there is a good chance that the model we've just learned
- ...Is violating a little or a lot a few of the expected monotonicities

Domain Knowledge in Machine Learning

The monotonic relations we mentioned are a form of domain knowledge

...Which can be used for a variety of purposes

- We can use it to improve generalization and counter selection bias
- ...Or we can use it to compensate for a lack of data
- ...Or to accomplish more with smaller models
- ...Or to allow a human to nudge some control over the model behavior

The best is that domain knowledge is widespread in industrial settings

After all, people have been doing their jobs for years without data-driven AI

- It would be very naïve to discard all the knowledge accumulated in the process
- ...And even more naïve to discard human contribution to an activity

When addressing a problem, all available resources should be employed

Domain Knowledge and Constraints

There are multiple ways to take into account domain knowledge in ML

Here we will focus on one particular approach

- If we are quite sure about our assumptions on the test distributions
- ...We can view them as constraints on the model

In practice, we try to learn a model that satisfies certain restrictions

The ability of constraining ML models is not well known

...But it is fairly widespread! However:

- There are restrictions on which models and constraints can be used
- Adding constriant makes training more difficult
- ...And if the constraints are not well chosen it may hurt the model quality

Our Constraints

Our constraints can be though as monotonicities

- We know that the average rating should have a monotonic effect on the click-through rate
- ...And the same holds for the number of reviews

We will assume that our domain expert knows one more fact:

- Averaged-price restaurant ar typically preferred to expensive ones

Monotonicities are frequently encountered and seem simple

...But they are among the trickiest constraints to enforce

- They involve multiple examples, since they are based on comparisons
- They ideally should hold on all data, including unseen examples

As a result, only very few ML model types support them

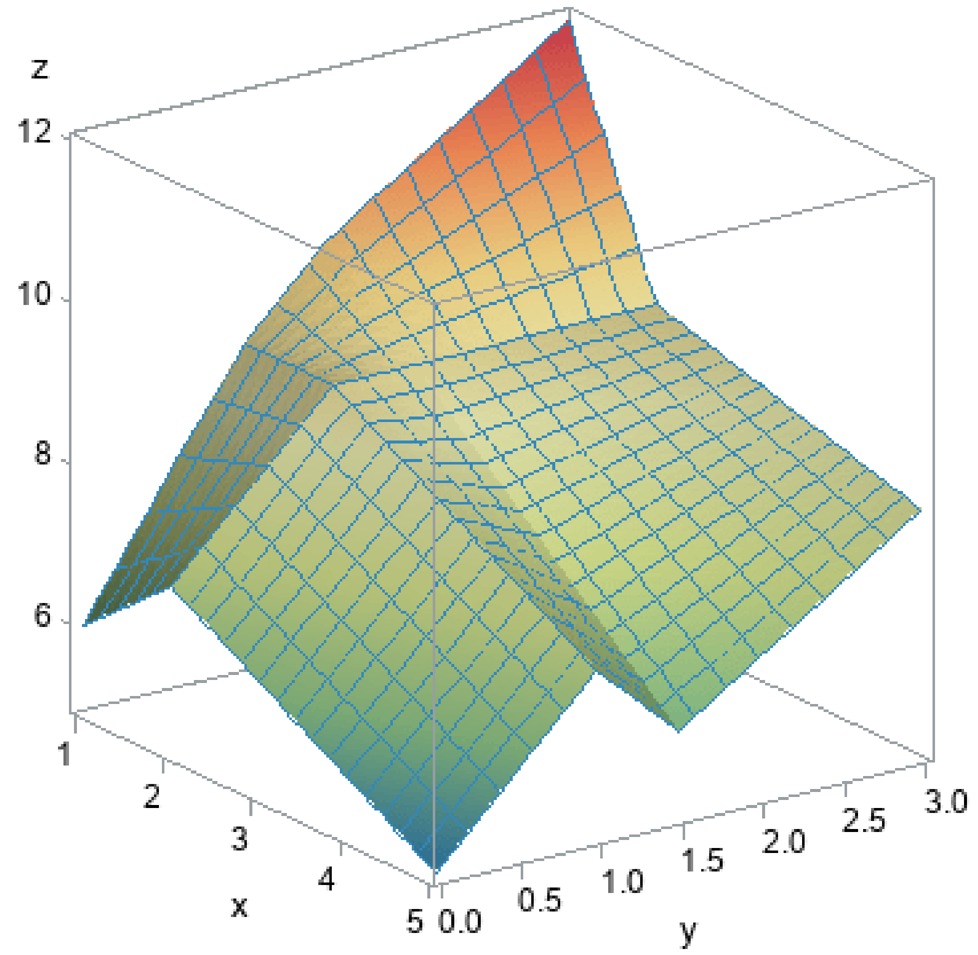
Domain Knowledge in Click-Through Rate Prediction

Lattice Models

MULTIPLYING OPPORTUNITIES

Lattice Models

Lattice models are among the few that can fully support monotonicity constraints



- 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

Lattice Model

Lattice models share some advantages with other ML models

- Like other ML models they can learn arbitrarily complex input/output relations
- They can be trained with the same algorithms used for Neural Networks

...But they are also easy to interpret

- Their parameters 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

As a drawback, lattice model have scalability issues

- They cannot be used directly for high-dimensional data
- ...Though they can appear as building block of more complex models

Constraints in Lattice Models

For sake of simplicity, let's consider a one-dimensional lattice

- The lattice will have one parameter θ_k for every grid point
- The parameter corresponds to the output value for the grid point

Then (increasing) monotonicity translates to:

$$\theta_k \leq \theta_{k+1}$$

- I.e. the output value at must be non-decreasing on the grid points
- The formulation can be extended to lattices with multiple inputs

Other constraints can be formulated in the same fashion

- E.g. convexity/concavity
- E.g. monotonicity between a subset of grid points

Defining the Constraints

Let's define the constraints on our model

In [15]:
```
calibration_args = {}
calibration_args['avg_rating'] = {'monotonicity': 'increasing','kernel_regularizer': ('hessian', 0, 1)}
```

- On `avg_rating`, we require monotonicity

In [16]:
```
calibration_args['num_reviews'] = {'monotonicity': 'increasing', 'convexity': 'concave', 'kernel_regularizer': ('wrinkle', 0, 1)}
```

- On `num_reviews`, we require monotonicity and concavity

In [17]:
```
calibration_args['dollar_rating'] = {'monotonicities': [(0, 1), (3, 1)]}
```

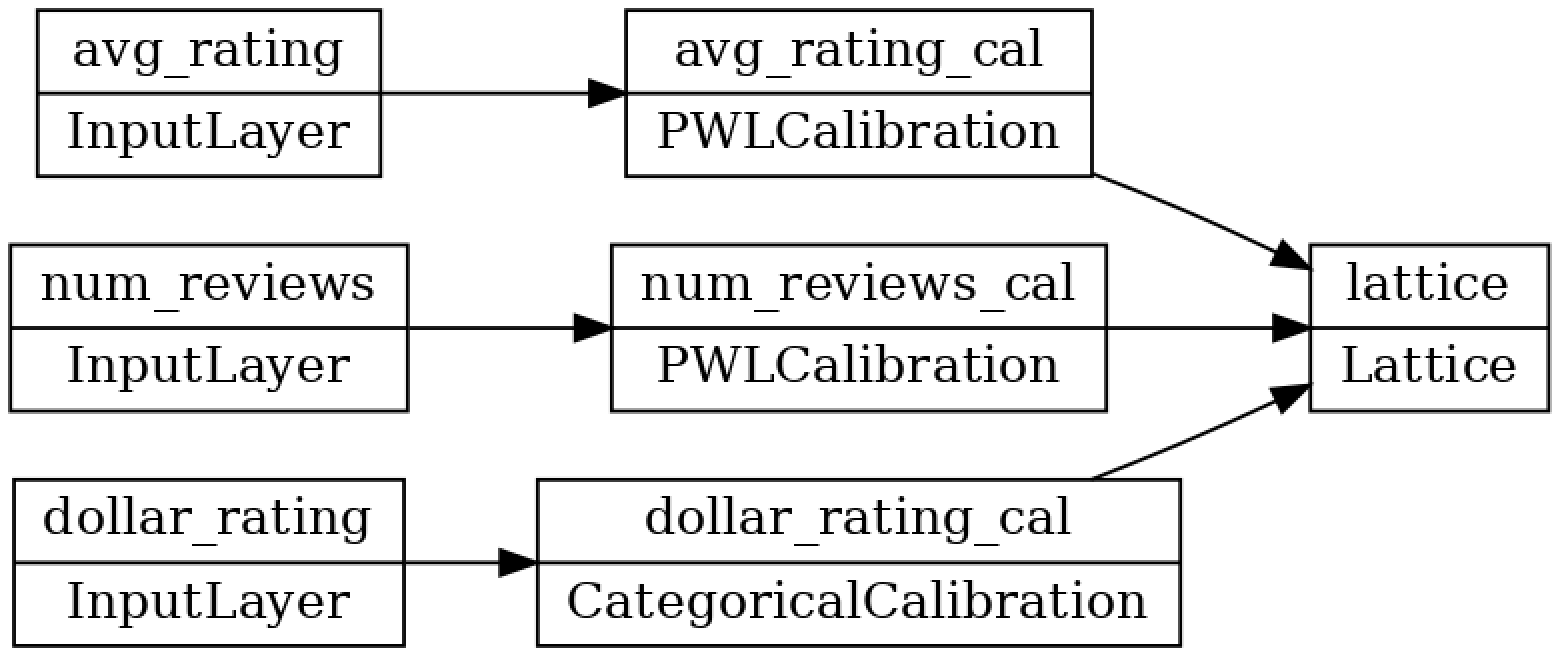
- On `dollar_rating`, we require monotonicity between specific categories
- In particularm "`D`" and "`DDDD`" should have lower click-through rate than "`DD`"

A Lattice Model for Our Problem

Now we can build a lattice model for our problem

In [18]:
```
lm, cal_models = util.build_calibrated_lattice_model(tr_sc2, lattice_sizes, attribute_names, calibration_types, calibration_sizes, ca
util.plot_nn_model(lm, show_layer_activations=True, show_layer_names=True, show_shapes=False, dpi=130)
```

Out[18]:



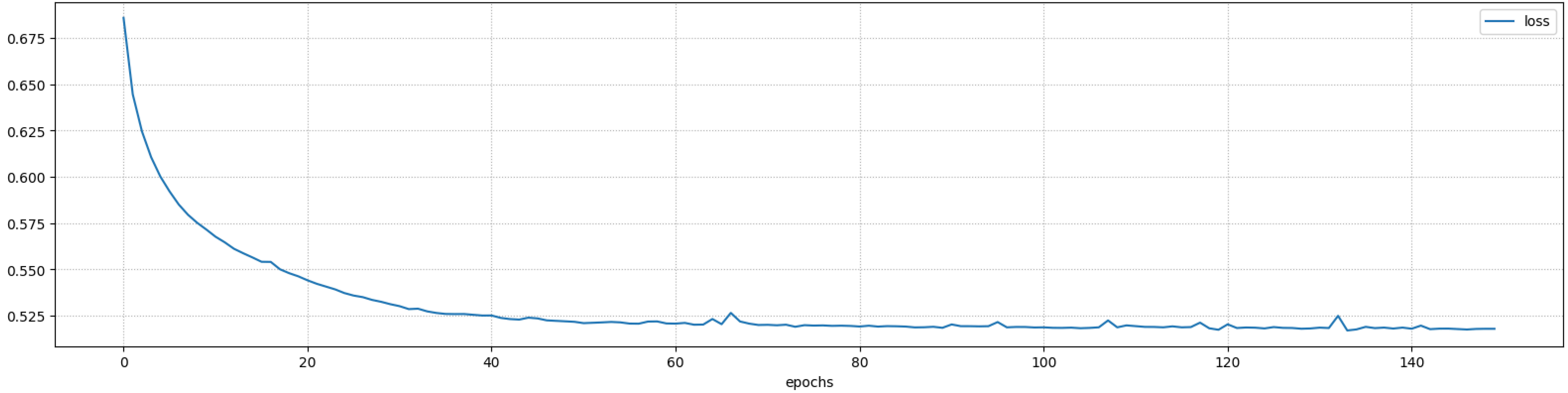
We are using a combination of lattices:

- We process every input through a one-dimensional lattice (piecewise linear interpolation)
- We combined the transformed input through a second lattice

Training the Lattice Model

The lattice model can be trained as usual

In [19]:
```
tr_ls, val_ls, ts_ls = [tr_sc2[c] for c in dt_in], [val_sc2[c] for c in dt_in], [ts_sc2[c] for c in dt_in]
history = util.train_nn_model(lm, tr_ls, tr_sc['clicked'], loss='binary_crossentropy', batch_size=32, epochs=150, verbose=0)
util.plot_training_history(history, figsize=figsize)
```




```
Final loss: 0.5179 (training)
```

Evaluting the Model

Let's evalute the performance of the model in terms of ROC-AUC

In [20]:
```
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_s['clicked'], pred_tr2)
auc_val2 = roc_auc_score(val_s['clicked'], pred_val2)
auc_ts2 = roc_auc_score(ts_s['clicked'], pred_ts2)
print(f'AUC score: {auc_tr2:.2f} (training), {auc_val2:.2f} (validation), {auc_ts2:.2f} (test)')
```

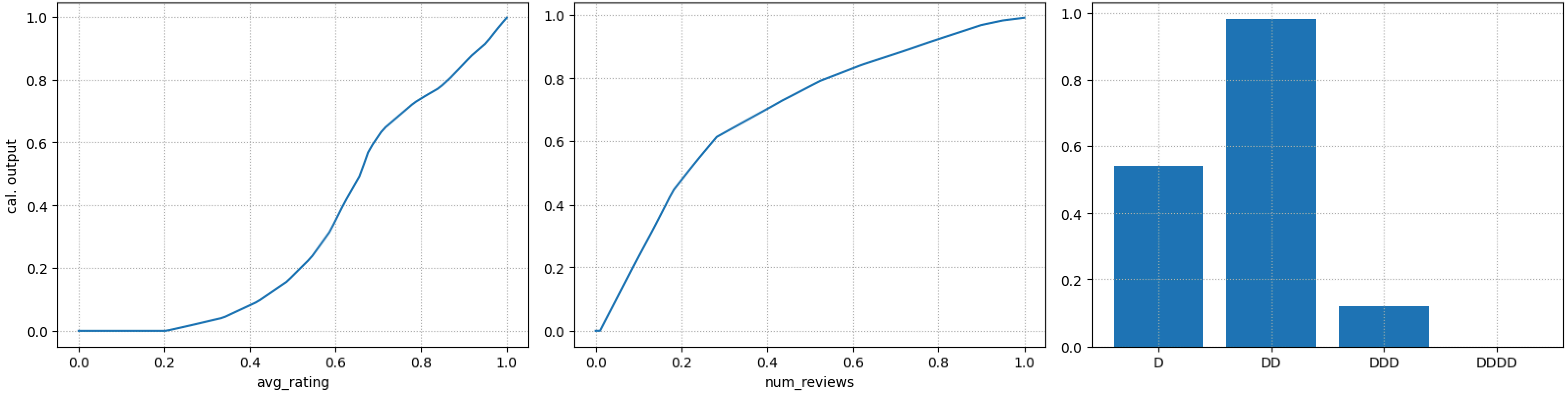
```
AUC score: 0.80 (training), 0.81 (validation), 0.81 (test)
```

- The performance on the test data is now on part with the training one
- This is due partially to the constrainta
- ...And partially to the fact that the model is simpler
- ...And therefore at a lower risk of overfitting

Inspecting the Model

We can inspec the firs-stage (one dimensional) lattices

In [21]:
```
util.plot_ctr_calibration(cal_models, scale, figsize=figsize)
```



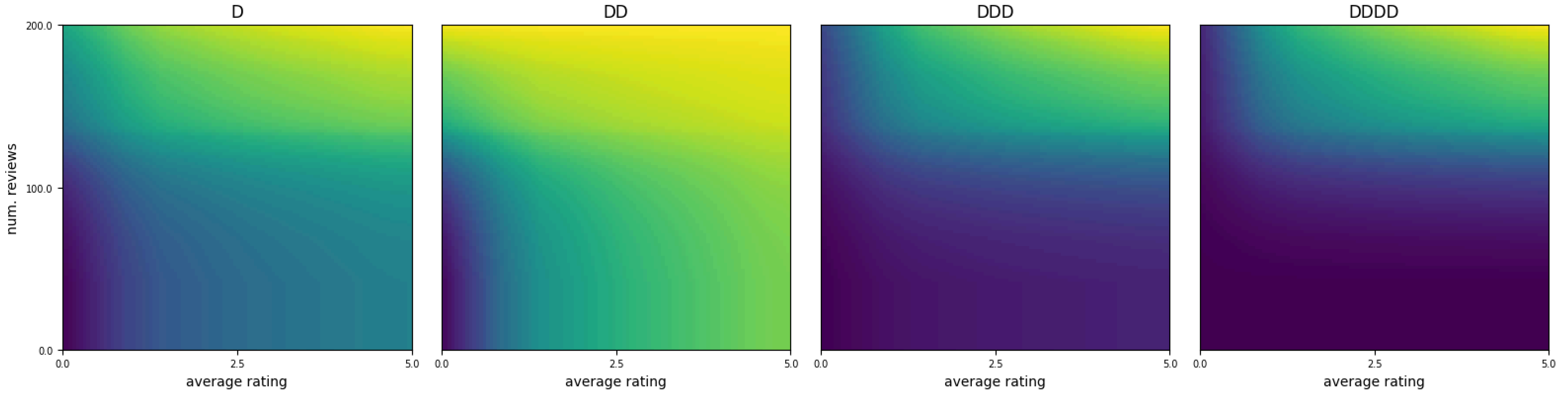
- As expected, all required monotonicities are respected

Inspecting the Model

The entire model also complies with the constraints

In [22]:
```
util.plot_ctr_estimation(lm, scale, split_input=True, one_hot_categorical=False, figsize=figsize)
```

```
WARNING:tensorflow:5 out of the last 148 calls to <function Model.make_predict_function.<locals>.predict_function at 0x7f5ca5009080>
triggered tf.function retracing. Tracing is expensive and the excessive number of tracings could be due to (1) creating @tf.function
repeatedly in a loop, (2) passing tensors with different shapes, (3) passing Python objects instead of tensors. For (1), please defi
ne your @tf.function outside of the loop. For (2), @tf.function has reduce_retracing=True option that can avoid unnecessary retracin
g. For (3), please refer to https://www.tensorflow.org/guide/function#controlling_retracing and https://www.tensorflow.org/api_docs/
python/tf/function for  more details.
```



- The tartan-like patter is caused by the fact that we first transform each input
- ...And the we use a relatively simple aggregation for the transformed inputs