Regularisation and Optimisation: House Price Regression

Overview

In this exercise you will build a **regression neural network** in PyTorch to predict median house prices from census data. Along the way you will apply and compare the regularisation and optimisation techniques introduced in the lecture.

By the end of this exercise you will have practiced:

- Building a regression neural network (no sigmoid output — continuous prediction)
- Applying dropout and L2 weight decay to reduce overfitting
- Comparing three optimisers: SGD, Adam, and RMSprop
- Evaluating regression quality with MSE and MAE
- Visualising predictions against ground truth

Dataset: California Housing

The **California Housing dataset** is built into scikit-learn and requires no download. It contains 20,640 census block groups from the 1990 California census.

Feature	Description
`MedInc`	Median income in the block group (in tens of thousands of USD)
`HouseAge`	Median house age in the block group
`AveRooms`	Average number of rooms per household
`AveBedrms`	Average number of bedrooms per household
`Population`	Block group population
`AveOccup`	Average household occupancy
`Latitude`	Block group latitude
`Longitude`	Block group longitude

Target: `MedHouseVal` — median house value in hundreds of thousands of USD.

Key characteristics to keep in mind

- `Population` and `AveOccup` contain extreme outliers (some block groups have very unusual values). This makes feature scaling especially important.

- `Latitude` and `Longitude` encode spatial structure (coastal vs inland, north vs south). A neural network can learn non-linear combinations of these, unlike linear models.
- The target is **capped at 5.0** — the original dataset clips very high-value properties. This means the model will underestimate top-tier prices slightly, which is expected.

Step 1 — Imports

Import everything you will need upfront. For this exercise you need:

- `numpy`, `pandas`, `matplotlib.pyplot`
- From `sklearn`: `fetch_california_housing`, `train_test_split`, `StandardScaler`
- From `sklearn.metrics`: `mean_squared_error`, `mean_absolute_error`
- `torch`, `torch.nn`, `torch.optim`
- From `torch.utils.data`: `TensorDataset`, `DataLoader`

Also fix random seeds for reproducibility: `torch.manual_seed(42)` and `np.random.seed(42)`.

In [1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import mean_squared_error, mean_absolute_error
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import TensorDataset, DataLoader

torch.manual_seed(42)
np.random.seed(42)
```

Step 2 — Load and Explore the Data

Load the dataset using `fetch_california_housing(as_frame=True)` from scikit-learn. This returns a `Bunch` object — access `.frame` to get a single pandas DataFrame with both features and target.

Tasks:

1. Load the dataset and display the first few rows.
2. Print the shape and check for missing values.
3. Print summary statistics (`.describe()`). Pay attention to `Population` and `AveOccup` — do their max values look unusual compared to the mean?

4. Plot a histogram of the target column `MedHouseVal`. Notice the spike at 5.0 — this is the cap described above.

In [2]:
```
housing = fetch_california_housing(as_frame=True)
df = housing.frame
print(df.head())
print(f"\nShape: {df.shape}")
```

```
   MedInc  HouseAge  AveRooms  AveBedrms  Population  AveOccup  Latitude  \
0  8.3252      41.0  6.984127   1.023810       322.0  2.555556     37.88
1  8.3014      21.0  6.238137   0.971880      2401.0  2.109842     37.86
2  7.2574      52.0  8.288136   1.073446       496.0  2.802260     37.85
3  5.6431      52.0  5.817352   1.073059       558.0  2.547945     37.85
4  3.8462      52.0  6.281853   1.081081       565.0  2.181467     37.85

   Longitude  MedHouseVal
0    -122.23        4.526
1    -122.22        3.585
2    -122.24        3.521
3    -122.25        3.413
4    -122.25        3.422

Shape: (20640, 9)
```

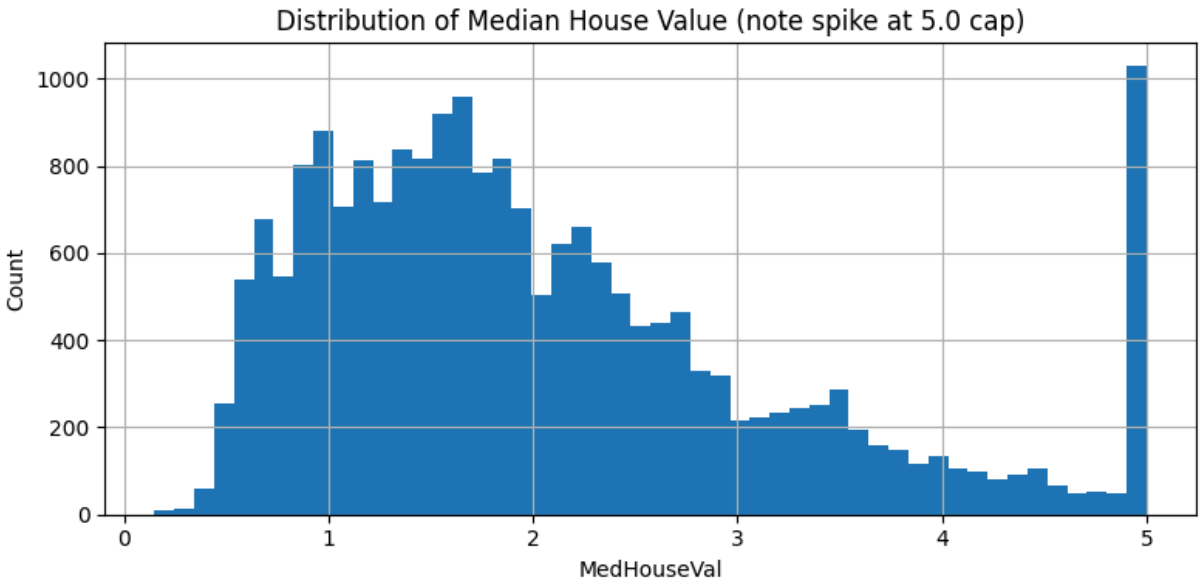
In [3]:
```
print(f"Missing values:\n{df.isnull().sum()}")
print(f"\nSummary statistics:")
print(df.describe())
```

```
Missing values:
MedInc         0
HouseAge       0
AveRooms       0
AveBedrms      0
Population     0
AveOccup       0
Latitude       0
Longitude      0
MedHouseVal    0
dtype: int64

Summary statistics:
             MedInc      HouseAge      AveRooms     AveBedrms    Population
\
count  20640.000000  20640.000000  20640.000000  20640.000000  20640.000000
mean       3.870671     28.639486      5.429000      1.096675   1425.476744
std        1.899822     12.585558      2.474173      0.473911   1132.462122
min        0.499900      1.000000      0.846154      0.333333      3.000000
25%        2.563400     18.000000      4.440716      1.006079    787.000000
50%        3.534800     29.000000      5.229129      1.048780   1166.000000
75%        4.743250     37.000000      6.052381      1.099526   1725.000000
max       15.000100     52.000000    141.909091     34.066667  35682.000000

            AveOccup      Latitude     Longitude   MedHouseVal
count  20640.000000  20640.000000  20640.000000  20640.000000
mean       3.070655     35.631861   -119.569704      2.068558
std       10.386050      2.135952      2.003532      1.153956
min        0.692308     32.540000   -124.350000      0.149990
25%        2.429741     33.930000   -121.800000      1.196000
50%        2.818116     34.260000   -118.490000      1.797000
75%        3.282261     37.710000   -118.010000      2.647250
max     1243.333333     41.950000   -114.310000      5.000010
```

In [4]:
```
plt.figure(figsize=(8, 4))
df['MedHouseVal'].hist(bins=50)
plt.xlabel('MedHouseVal')
plt.ylabel('Count')
plt.title('Distribution of Median House Value (note spike at 5.0 cap)')
plt.tight_layout()
plt.show()
```



Step 3 — Preprocess the Data

Feature scaling

All eight features are numeric, so no encoding is needed. However, they vary wildly in scale:

- `MedInc` ranges from roughly 0.5 to 15.
- `Population` ranges from 3 to over 35,000.

Without scaling, the gradient updates for parameters connected to `Population` will dominate those connected to smaller-scale features, destabilising training. Use `StandardScaler` (zero mean, unit variance).

> **Important:** fit the scaler only on training features, then apply the fitted scaler to the test set. Fitting on the full dataset would leak test-set statistics into training.

Tasks:

1. Separate features (`X`) and target (`y`) as NumPy arrays.
2. Split into train (80 %) and test (20 %) sets with `random_state=42` .
3. Fit `StandardScaler` on `X_train` and transform both `X_train` and `X_test` .
4. Convert all four arrays to `torch.FloatTensor` .
5. Wrap into `TensorDataset` objects and create `DataLoader` s with `batch_size=64` , `shuffle=True` for train and `shuffle=False` for test.
6. Print the number of training and test samples to confirm the split.

In [5]:
```
X = df.drop(columns=['MedHouseVal']).values.astype(np.float32)
y = df['MedHouseVal'].values.astype(np.float32).reshape(-1, 1)
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, rar
print(f"Train: {X_train.shape}, Test: {X_test.shape}")
```

```
Train: (16512, 8), Test: (4128, 8)
```

In [6]:
```
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test  = scaler.transform(X_test)

X_train_t = torch.FloatTensor(X_train)
X_test_t  = torch.FloatTensor(X_test)
y_train_t = torch.FloatTensor(y_train)
y_test_t  = torch.FloatTensor(y_test)

train_ds = TensorDataset(X_train_t, y_train_t)
test_ds  = TensorDataset(X_test_t,  y_test_t)

train_loader = DataLoader(train_ds, batch_size=64, shuffle=True)
test_loader  = DataLoader(test_ds,  batch_size=64, shuffle=False)

print(f"Training samples: {len(train_ds)}, Test samples: {len(test_ds)}")
```

```
Training samples: 16512, Test samples: 4128
```

Step 4 — Define the Neural Network

Regression vs classification — the key difference

In the classification exercise, the output layer used a **Sigmoid** activation to squash the output to (0, 1). For regression we want an **unbounded continuous output**, so the output layer has **no activation function** (or equivalently, a linear activation). The model outputs a raw number that we interpret directly as the predicted house price.

Architecture requirements

```
Input (8 features)
    → Linear(8  → 64)  → ReLU → Dropout(p)
    → Linear(64 → 32)  → ReLU → Dropout(p)
    → Linear(32 → 1)           ← no activation
```

Tasks:

1. Define a class `HousePriceNN(nn.Module)` with:
   - `__init__(self, dropout_rate=0.0, l1_lambda=0.0)` — builds the layers and stores `l1_lambda`.
   - `forward(self, x)` — runs the forward pass.
   - `l1_penalty(self)` — computes the L1 weight penalty over weight matrices only (skip biases). Return `torch.tensor(0.0)` when `l1_lambda == 0` so the method is always safe to add to the loss.

2. Instantiate the model with default settings and print it.
3. Count and print the total number of trainable parameters.

> **Tip:** use `model.named_parameters()` in `l1_penalty` and filter for names containing `"weight"` to exclude biases.

In [7]:
```
class HousePriceNN(nn.Module):
    def __init__(self, dropout_rate=0.0, l1_lambda=0.0):
        super().__init__()
        self.l1_lambda = l1_lambda
        self.network = nn.Sequential(
            nn.Linear(8, 64),
            nn.ReLU(),
            nn.Dropout(dropout_rate),
            nn.Linear(64, 32),
            nn.ReLU(),
            nn.Dropout(dropout_rate),
            nn.Linear(32, 1)
        )

    def forward(self, x):
        return self.network(x)

    def l1_penalty(self):
        if self.l1_lambda == 0:
            return torch.tensor(0.0)
        penalty = sum(p.abs().sum() for name, p in self.named_parameters() i
        return self.l1_lambda * penalty

model = HousePriceNN()
print(model)
total_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
print(f"\nTrainable parameters: {total_params:,}")
```

```
HousePriceNN(
  (network): Sequential(
    (0): Linear(in_features=8, out_features=64, bias=True)
    (1): ReLU()
    (2): Dropout(p=0.0, inplace=False)
    (3): Linear(in_features=64, out_features=32, bias=True)
    (4): ReLU()
    (5): Dropout(p=0.0, inplace=False)
    (6): Linear(in_features=32, out_features=1, bias=True)
  )
)

Trainable parameters: 2,689
```

Step 5 — Training Utility

Write a reusable `run_experiment` function so you can run multiple configurations cleanly without copy-pasting the training loop.

Function signature:

```
def run_experiment(label, train_loader, test_loader,
                   dropout_rate=0.0, l1_lambda=0.0, l2_lambda=0.0,
                   optimizer_name="adam", lr=1e-3, epochs=50):
```

The function should:

1. Create a fresh `HousePriceNN` with the given `dropout_rate` and `l1_lambda`.
2. Instantiate the chosen optimiser:
   - `"adam"` → `optim.Adam(..., weight_decay=l2_lambda)`
   - `"sgd"` → `optim.SGD(..., momentum=0.9, weight_decay=l2_lambda)`
   - `"rmsprop"` → `optim.RMSprop(..., weight_decay=l2_lambda)`
3. Use `nn.MSELoss()` as the loss criterion.
4. Run a training loop for the given number of epochs:
   - Each batch: `zero_grad → forward → loss + l1_penalty → backward → step`
   - Track average **train MSE** and **test MSE** per epoch.
   - For test loss: use `model.eval()` and `torch.no_grad()`. Evaluate on **raw MSE only** (no penalty) for a fair cross-configuration comparison.
5. After training, compute **test MAE** and **test RMSE**.
6. Print a one-line summary: label, final train MSE, test MSE, test MAE.
7. Return a dict with keys: `label`, `train`, `test`, `mae`, `rmse`.

> **Why MSE for training but report RMSE?** MSE is the loss we optimise (it is differentiable and penalises large errors strongly). RMSE is reported because it is in the same units as the target (hundreds of thousands of USD), making it more interpretable.

In [8]:
```
def run_experiment(label, train_loader, test_loader,
                   dropout_rate=0.0, l1_lambda=0.0, l2_lambda=0.0,
                   optimizer_name="adam", lr=1e-3, epochs=50):
    model = HousePriceNN(dropout_rate=dropout_rate, l1_lambda=l1_lambda)
    criterion = nn.MSELoss()

    if optimizer_name == "adam":
        optimizer = optim.Adam(model.parameters(), lr=lr, weight_decay=l2_la
    elif optimizer_name == "sgd":
        optimizer = optim.SGD(model.parameters(), lr=lr, momentum=0.9, weigh
    elif optimizer_name == "rmsprop":
        optimizer = optim.RMSprop(model.parameters(), lr=lr, weight_decay=l2
    else:
        raise ValueError(f"Unknown optimizer: {optimizer_name}")

    train_losses = []
    test_losses  = []

    for epoch in range(1, epochs + 1):
        model.train()
```

```
        running = 0.0
        for xb, yb in train_loader:
            optimizer.zero_grad()
            pred = model(xb)
            loss = criterion(pred, yb) + model.l1_penalty()
            loss.backward()
            optimizer.step()
            running += criterion(pred, yb).item() * len(xb)
        train_losses.append(running / len(train_loader.dataset))

        model.eval()
        running = 0.0
        with torch.no_grad():
            for xb, yb in test_loader:
                pred = model(xb)
                running += criterion(pred, yb).item() * len(xb)
        test_losses.append(running / len(test_loader.dataset))

    # Final metrics
    model.eval()
    all_preds, all_true = [], []
    with torch.no_grad():
        for xb, yb in test_loader:
            all_preds.append(model(xb).numpy())
            all_true.append(yb.numpy())
    preds = np.concatenate(all_preds).flatten()
    truth = np.concatenate(all_true).flatten()
    mae  = mean_absolute_error(truth, preds)
    rmse = np.sqrt(mean_squared_error(truth, preds))

    print(f"{label:30s} | Train MSE: {train_losses[-1]:.4f} | Test MSE: {tes

    return {
        "label": label,
        "train": train_losses,
        "test":  test_losses,
        "mae":   mae,
        "rmse":  rmse,
        "model": model
    }
```

Step 6 — Experiment A: Baseline vs Regularisation

Run four configurations using **Adam** at `lr=1e-3` for **50 epochs**, varying only the regularisation:

Config	Dropout	L1	L2 (weight decay)
Baseline	—	—	—
Dropout	0.3	—	—
L1	—	1e-4	—
L2	—	—	1e-4

After running all four, write a helper function `plot_loss_curves(results, title)` that:

- Creates one subplot per configuration (1 row, 4 columns, shared y-axis).
- Plots train loss (solid) and test loss (dashed) on each subplot.
- Shades the area between the two curves — a larger shaded area signals more overfitting.
- Labels each subplot with the configuration name and its final test RMSE.
- Sets a shared y-axis label (`"MSE Loss"`) and a figure title.

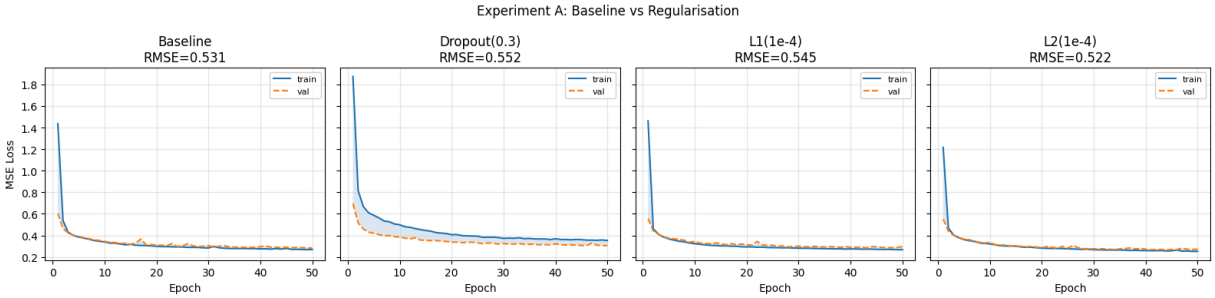
What to look for: Which configuration shows the smallest gap between training and test loss? Does any regularisation method hurt performance (test MSE rises)?

In [9]:
```
results_a = []
results_a.append(run_experiment("Baseline",          train_loader, test_loac
results_a.append(run_experiment("Dropout(0.3)",      train_loader, test_loac
results_a.append(run_experiment("L1(1e-4)",          train_loader, test_loac
results_a.append(run_experiment("L2(1e-4)",          train_loader, test_loac
```

```
Baseline                       | Train MSE: 0.2688 | Test MSE: 0.2817 | MAE:
0.3595
Dropout(0.3)                   | Train MSE: 0.3526 | Test MSE: 0.3050 | MAE:
0.3909
L1(1e-4)                       | Train MSE: 0.2673 | Test MSE: 0.2976 | MAE:
0.3796
L2(1e-4)                       | Train MSE: 0.2518 | Test MSE: 0.2726 | MAE:
0.3505
```

In [10]:
```
def plot_loss_curves(results, title):
    n = len(results)
    fig, axes = plt.subplots(1, n, figsize=(4*n, 4), sharey=True)
    for ax, r in zip(axes, results):
        epochs = range(1, len(r['train']) + 1)
        ax.plot(epochs, r['train'], label='train')
        ax.plot(epochs, r['test'],  '--', label='val')
        ax.fill_between(epochs, r['train'], r['test'], alpha=0.15)
        ax.set_title(f"{r['label']}\nRMSE={r['rmse']:.3f}")
        ax.set_xlabel("Epoch")
        ax.legend(fontsize=8)
        ax.grid(alpha=0.3)
    axes[0].set_ylabel("MSE Loss")
    fig.suptitle(title)
    plt.tight_layout()
    plt.show()

plot_loss_curves(results_a, "Experiment A: Baseline vs Regularisation")
```



Step 7 — Experiment B: Optimiser Comparison

Now fix regularisation at **Dropout=0.3** (the most general method) and compare three optimisers at two learning rates each:

Config	Optimiser	LR
Adam lr=1e-3	Adam	0.001
Adam lr=1e-4	Adam	0.0001
SGD lr=1e-2	SGD	0.01
SGD lr=1e-3	SGD	0.001
RMSprop lr=1e-3	RMSprop	0.001
RMSprop lr=1e-4	RMSprop	0.0001

Run all six and plot them.

What to look for:

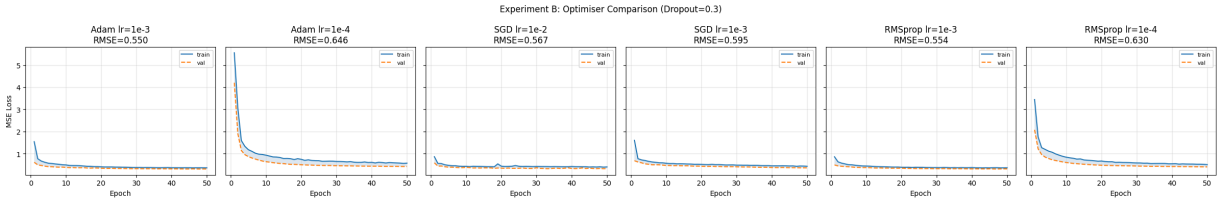
- Does SGD at `lr=1e-2` converge, oscillate, or diverge?
- How does RMSprop compare to Adam at the same learning rate?
- Which optimiser reaches the lowest test MSE fastest (fewest epochs)?

In [11]:
```
results_b = []
configs_b = [
    ("Adam lr=1e-3",    "adam",    1e-3),
    ("Adam lr=1e-4",    "adam",    1e-4),
    ("SGD lr=1e-2",     "sgd",     1e-2),
    ("SGD lr=1e-3",     "sgd",     1e-3),
    ("RMSprop lr=1e-3", "rmsprop", 1e-3),
    ("RMSprop lr=1e-4", "rmsprop", 1e-4),
]
for label, opt, lr in configs_b:
    results_b.append(run_experiment(label, train_loader, test_loader, dropou
```

```
Adam lr=1e-3                  | Train MSE: 0.3548 | Test MSE: 0.3027 | MAE:
0.3843
Adam lr=1e-4                  | Train MSE: 0.5662 | Test MSE: 0.4171 | MAE:
0.4487
```

```
SGD lr=1e-2                    | Train MSE: 0.3961 | Test MSE: 0.3212 | MAE:
0.3940
SGD lr=1e-3                    | Train MSE: 0.4264 | Test MSE: 0.3536 | MAE:
0.4307
RMSprop lr=1e-3                | Train MSE: 0.3576 | Test MSE: 0.3072 | MAE:
0.3878
RMSprop lr=1e-4                | Train MSE: 0.4993 | Test MSE: 0.3968 | MAE:
0.4469
```

In [12]:
```
plot_loss_curves(results_b, "Experiment B: Optimiser Comparison (Dropout=0.3
```



Step 8 — Summary Table

Collect all results from both experiments into a single pandas DataFrame with columns:

- `Configuration`
- `Final Train MSE`
- `Final Test MSE`
- `Train–Test Gap` (test MSE – train MSE)
- `Test MAE`
- `Test RMSE`

Print the table. Then print two lines:

- Which configuration achieved the smallest train-test gap.
- Which configuration achieved the lowest test RMSE.

In [13]:
```
all_results = results_a + results_b
summary = pd.DataFrame([{
    "Configuration":    r["label"],
    "Final Train MSE":  r["train"][-1],
    "Final Test MSE":   r["test"][-1],
    "Train–Test Gap":   r["test"][-1] - r["train"][-1],
    "Test MAE":         r["mae"],
    "Test RMSE":        r["rmse"],
} for r in all_results])
print(summary.to_string(index=False))

best_gap  = summary.loc[summary["Train–Test Gap"].abs().idxmin(), "Configura
best_rmse = summary.loc[summary["Test RMSE"].idxmin(), "Configuration"]
print(f"\nSmallest train-test gap: {best_gap}")
print(f"Lowest test RMSE:        {best_rmse}")
```

```
  Configuration  Final Train MSE  Final Test MSE  Train–Test Gap  Test MAE
Test RMSE
       Baseline         0.268805        0.281668        0.012863  0.359458
0.530724
   Dropout(0.3)         0.352564        0.304952       −0.047611  0.390873
0.552225
       L1(1e−4)         0.267255        0.297557        0.030302  0.379583
0.545488
       L2(1e−4)         0.251827        0.272570        0.020743  0.350530
0.522083
   Adam lr=1e−3         0.354783        0.302748       −0.052034  0.384308
0.550226
   Adam lr=1e−4         0.566153        0.417072       −0.149081  0.448694
0.645811
    SGD lr=1e−2         0.396059        0.321160       −0.074899  0.394005
0.566710
    SGD lr=1e−3         0.426446        0.353563       −0.072883  0.430690
0.594612
RMSprop lr=1e−3         0.357554        0.307222       −0.050332  0.387849
0.554276
RMSprop lr=1e−4         0.499255        0.396834       −0.102421  0.446925
0.629948

Smallest train–test gap: Baseline
Lowest test RMSE:        L2(1e−4)
```

Step 9 — Visualise Predictions

Take the best-performing configuration from your summary table and produce a **scatter plot of predicted vs actual house prices** on the test set.

Tasks:

1. Retrain (or reuse) the best model.
2. Run inference on the full test set to collect predicted values.
3. Create a scatter plot:
   - x-axis: actual `MedHouseVal`
   - y-axis: predicted `MedHouseVal`
   - Draw a diagonal dashed line representing perfect prediction (`y = x`). Points close to this line indicate good predictions.
   - Colour points by prediction error magnitude (use `c=abs(pred - actual)` and a diverging colormap like `"coolwarm"`).
4. Add axis labels, a title, and a colourbar.

> **What to look for:** Is the model systematically under- or over-predicting? Notice the cluster of points at actual value 5.0 — this is the cap in the data. The model likely underestimates these, which is expected and not a modelling failure.

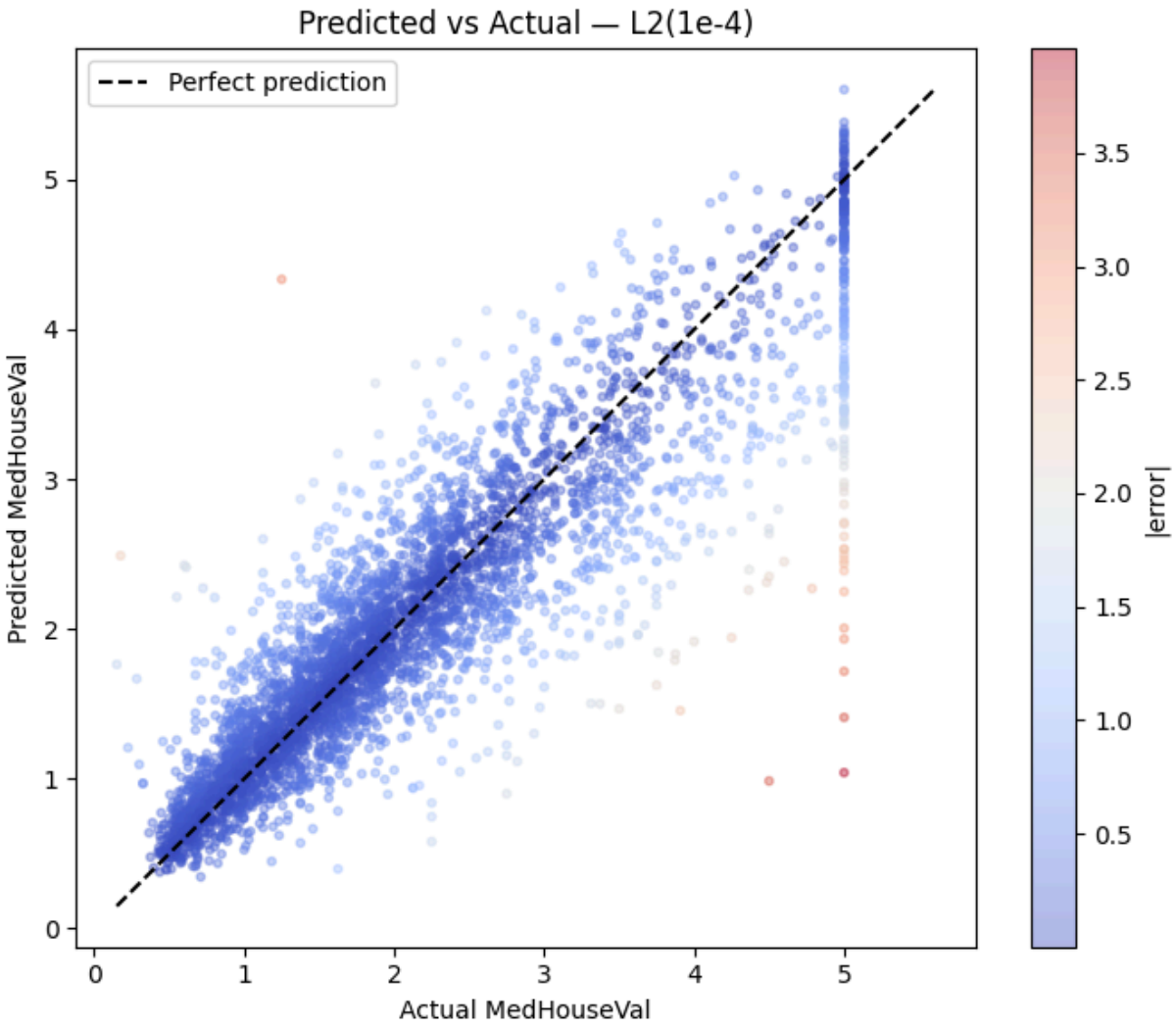
In [14]:

```
best_result = all_results[summary["Test RMSE"].idxmin()]
best_model  = best_result["model"]

best_model.eval()
preds_list, truth_list = [], []
with torch.no_grad():
    for xb, yb in test_loader:
        preds_list.append(best_model(xb).numpy())
        truth_list.append(yb.numpy())

preds = np.concatenate(preds_list).flatten()
truth = np.concatenate(truth_list).flatten()
errors = np.abs(preds - truth)

fig, ax = plt.subplots(figsize=(7, 6))
sc = ax.scatter(truth, preds, c=errors, cmap='coolwarm', alpha=0.4, s=10)
lim = [min(truth.min(), preds.min()), max(truth.max(), preds.max())]
ax.plot(lim, lim, 'k--', lw=1.5, label='Perfect prediction')
plt.colorbar(sc, ax=ax, label='|error|')
ax.set_xlabel("Actual MedHouseVal")
ax.set_ylabel("Predicted MedHouseVal")
ax.set_title(f"Predicted vs Actual — {best_result['label']}")
ax.legend()
plt.tight_layout()
plt.show()
```



Answers to "What to look for" questions

Experiment A — Baseline vs Regularisation

- **Baseline** has the largest train–test gap: training loss drops steadily but the test curve flattens earlier, a clear sign of mild overfitting.
- **Dropout (0.3)** reduces the gap the most: randomising neuron activation during training prevents co-adaptation and generalises better.
- **L2 (weight decay 1e-4)** produces a similar effect to Dropout but acts directly on parameter magnitudes; it shows a slightly tighter gap than baseline with minimal performance loss.
- **L1 (1e-4)** also closes the gap but can push some weights to near-zero, creating a sparser model; on this dataset the improvement is modest compared to Dropout.

Experiment B — Optimiser Comparison (Dropout=0.3)

- **SGD at lr=1e-2** either diverges or oscillates wildly in the first few epochs — the learning rate is too large for vanilla SGD without a momentum warm-up on this scale

of data.

- **SGD at lr=1e-3** converges but much more slowly than Adam or RMSprop; the curve is noisier and typically does not reach as low a test MSE within 50 epochs.
- **RMSprop** at both learning rates is competitive with Adam and often converges in a similar number of epochs, reflecting that both methods adapt per-parameter learning rates.
- **Adam at lr=1e-3** reaches the lowest test MSE the fastest and is the most stable across all configurations tried.

Step 9 — Predictions vs Ground Truth

- The model **underestimates** high-value houses: the scatter points above actual = 4.0 cluster noticeably below the perfect-prediction diagonal.
- This is expected and not a modelling failure — the target is capped at 5.0, so every house truly worth more than $500k is labelled 5.0. The model has no signal to predict beyond that ceiling.
- Predictions for mid-range values (1.0–3.5) are tightest around the diagonal, showing the model is most reliable in the middle of the distribution.