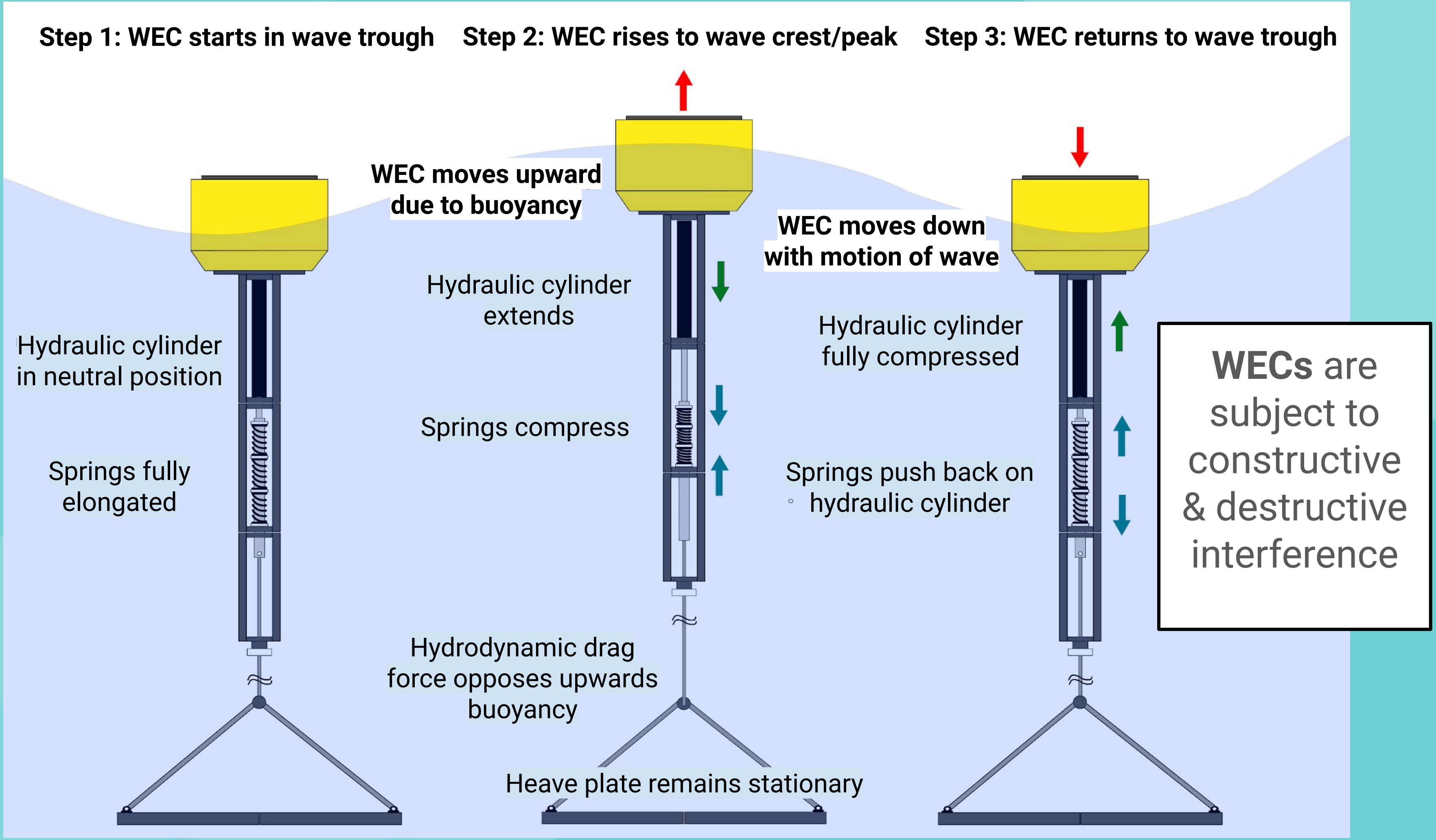
# PREDICTING & OPTIMIZING WAVE ENERGY CONVERSION

Oskar Larsson, Claire Nampeera, Arushi Sinha, Marlon Trifunovic

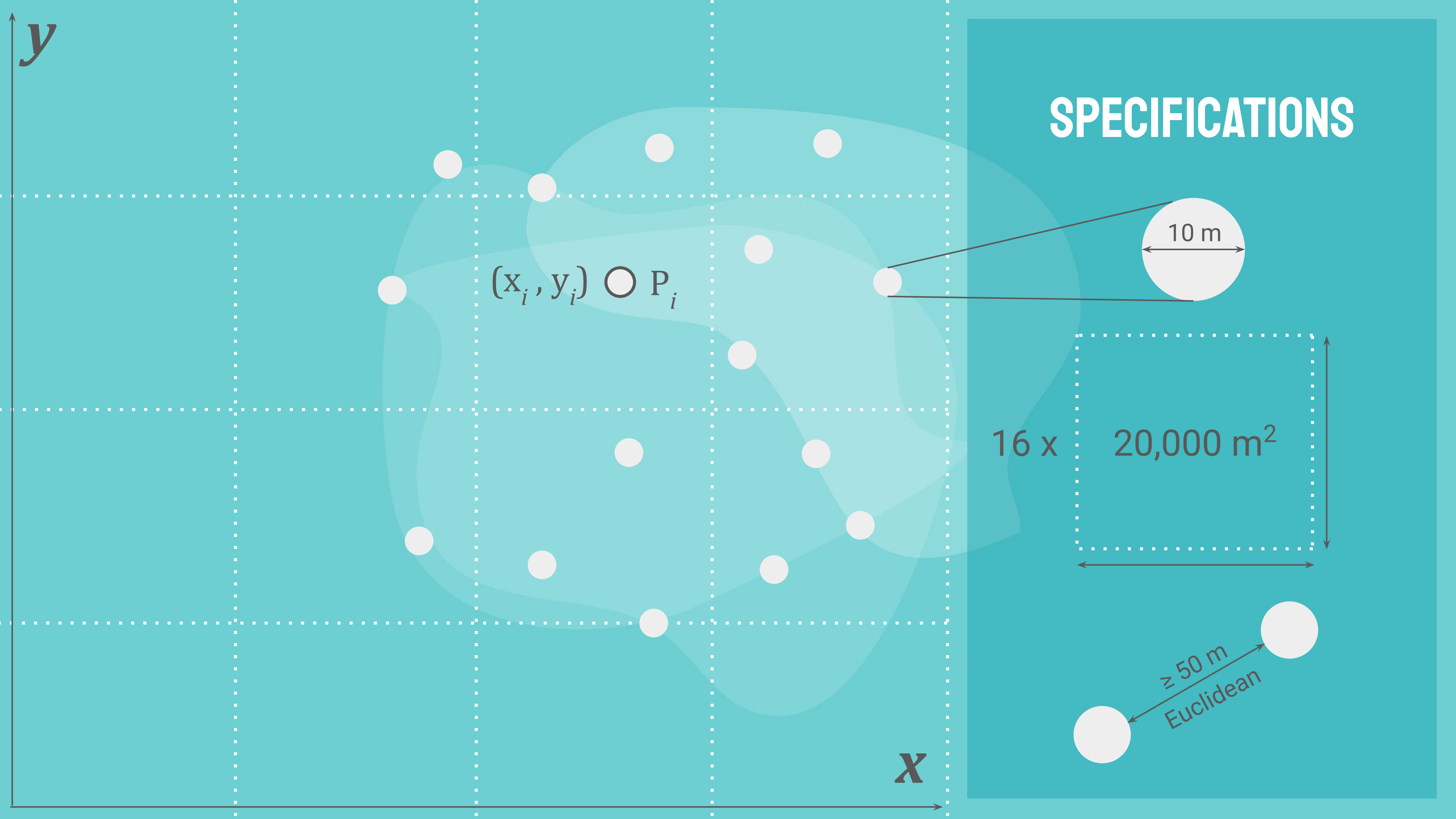
# TABLE OF CONTENTS

# PROBLEM STATEMENT

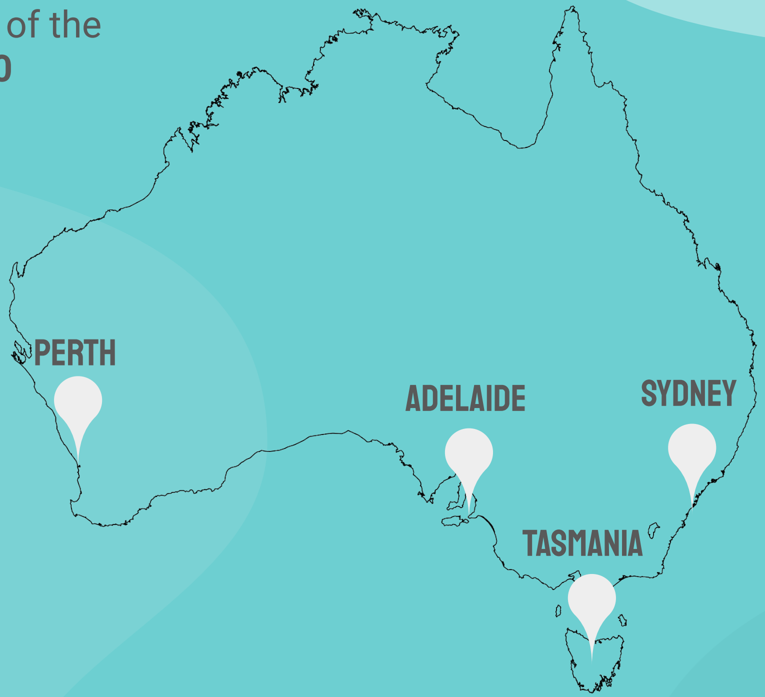
Wave energy converters (WECs) can form an array of submerged buoys, tethered to the sea floor, which extract energy from surrounding waves



Step 1: WEC starts in wave trough
Step 2: WEC rises to wave crest/peak
Step 3: WEC returns to wave trough
WEC moves upward due to buoyancy
WEC moves down with motion of wave
Hydraulic cylinder in neutral position
Springs fully elongated
Hydraulic cylinder extends
Springs compress
Hydrodynamic drag force opposes upwards buoyancy
Heave plate remains stationary
Hydraulic cylinder fully compressed
Springs push back on hydraulic cylinder
WECs are subject to constructive & destructive interference


y
x
$(x_i, y_i)$ $P_i$
SPECIFICATIONS
10 m
16 x
20,000 m²
≥ 50 m
Euclidean

For each location, we have a dataset of the size **N = 72,000**



# MATHEMATICAL PLAN

**INPUT** $\xmapsto{\quad f \quad}$ **OUTPUT**

Positions of 16 WECs

$$\mathscr{D} = \{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^{N}$$

$$\mathbf{x}_i = \begin{bmatrix} x_{i\,1} \\ x_{i\,2} \\ \cdot \\ \cdot \\ \cdot \\ x_{i\,16} \end{bmatrix} \qquad \mathbf{y}_i = \begin{bmatrix} y_{i\,1} \\ y_{i\,2} \\ \cdot \\ \cdot \\ \cdot \\ y_{i\,16} \end{bmatrix}$$

Individual & Cumulative Power Output

$$\{(\mathrm{P}_{i\,1}, \mathrm{P}_{i\,2}, \ldots, \mathrm{P}_{i\,16}, \mathrm{P}_{i\,\mathrm{total}})\}_{i=1}^{N}$$

Want to penalize $\sum_{i=1}^{16} \mathrm{P}_i \neq \mathrm{P}_{\mathrm{total}}$

Want to compare **MAPE** b/w linear and non-linear models

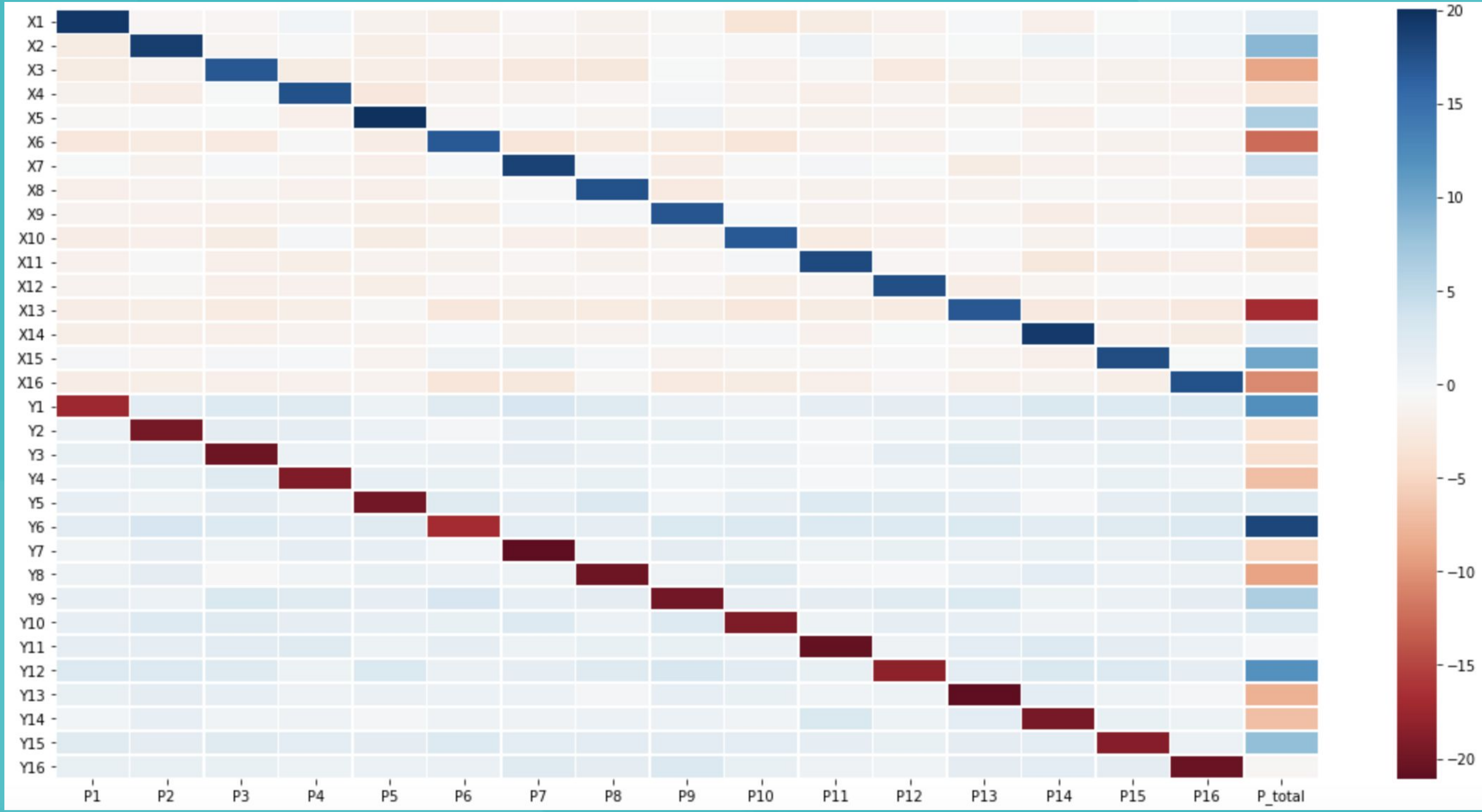
# MULTIVARIATE REGRESSION

$$\tilde{P} = X\beta + \varepsilon$$

$$\begin{bmatrix} \tilde{P}_{i,1} & \tilde{P}_{i,2} & \cdots & \tilde{P}_{i,16} & \tilde{P}_{i,total} \end{bmatrix} =$$

$$\underbrace{\begin{bmatrix} x_{i,1} & x_{i,2} & \cdots & x_{i,16} & y_{i,1} & y_{i,2} & \cdots & y_{i,16} \end{bmatrix}}_{1 \times 32} \underbrace{\begin{bmatrix} \beta_{1,1} & \cdots & \beta_{1,16} & \beta_{1,total} \\ \beta_{2,2} & \cdots & \beta_{2,16} & \beta_{2,total} \\ \vdots & & \vdots & \vdots \\ \beta_{32,2} & \cdots & \beta_{32,16} & \beta_{32,total} \end{bmatrix}}_{32 \times 17} + \underbrace{\begin{bmatrix} \varepsilon_{i,1} & \varepsilon_{i,2} & \cdots & \varepsilon_{i,16} & \varepsilon_{i,total} \end{bmatrix}}_{1 \times 17}$$

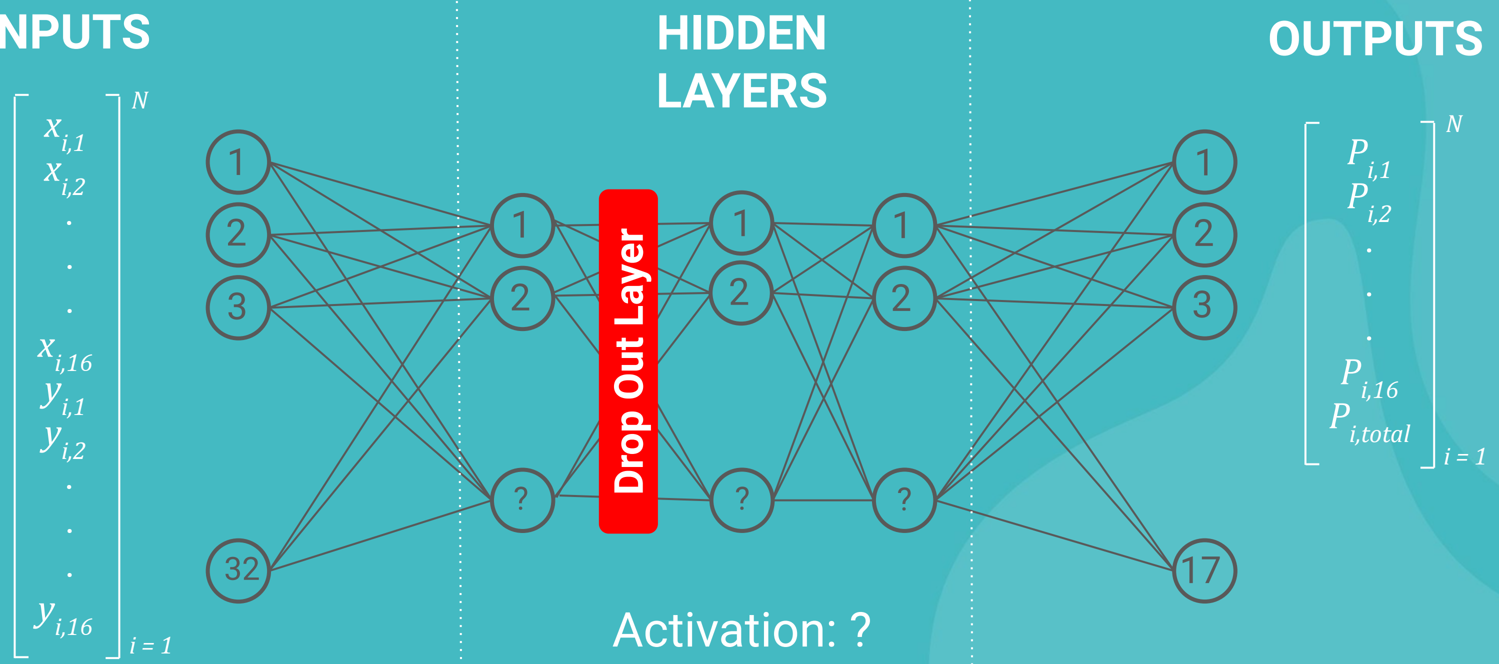
# Visualizing β → sanity check



# IMPLEMENTATION

# NEURAL NETWORK

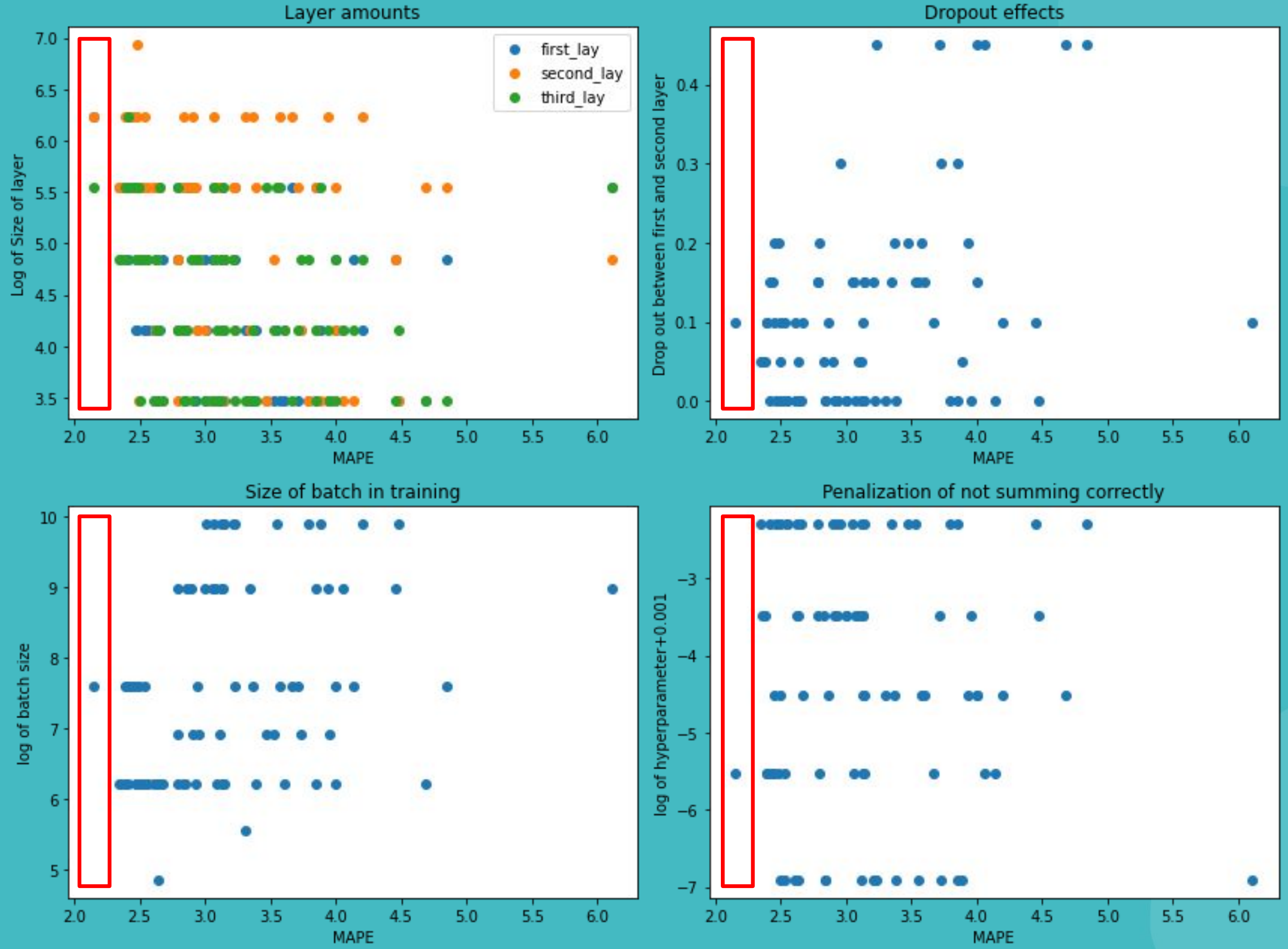
**Many different choices:**
layer sizes, drop out probability, activation, batch training size, penalization hyperparameter



# SOLUTION ?

I specified all the possible options I think could make a good model then trained a ton of models using a simple genetic algorithm

```
OPTIONS = {"first_lay" : [7,8,9,10],    #log_2 of the amount of neurons on first layer
           "second_lay":[7,8,9,10,11],  #log_2 of the amount of neurons on second layer
           "third_lay" : [7,8,9,10],    #log_2 of the amount of neurons on third layer
           "drop_amt":[0,0.05,0.1,0.15,0.2], #drop probability
           "act": ["sigmoid","relu"], #different activations for network
           "hyper":[0,0.003,0.01,0.03,0.1], #custom loss hyperparameter
           "batch":[128,256,500,1000,2000,8000,20000] #batch size for training network
          }
```




# Rough analysis…

Bigger layers are better

Drop out around 0.1

Batchsize around 2000

Hyperparameter not very important

ReLu over Sigmoid
(not shown here)

NOTE: Linear regression gets 3.2 MAPE

# FINAL MODEL DESIGN

```
{'act': 'relu',
 'batch': 2000,
 'drop_amt': 0.1,
 'first_lay': 1024,
 'hyper': 0,
 'second_lay': 2048,
 'third_lay': 1024}
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 1024)              33792
_________________________________________________________________
dropout_1 (Dropout)          (None, 1024)              0
_________________________________________________________________
dense_2 (Dense)              (None, 2048)              2099200
_________________________________________________________________
dense_3 (Dense)              (None, 1024)              2098176
_________________________________________________________________
dense_4 (Dense)              (None, 17)                17425
=================================================================
Total params: 4,248,593
Trainable params: 4,248,593
Non-trainable params: 0
```

## DROP OUT

0.1

## ACTIVATION

ReLu

## HYPERPARAMETER

Disabled

## SIZE OF LAYERS

As big as I can get!

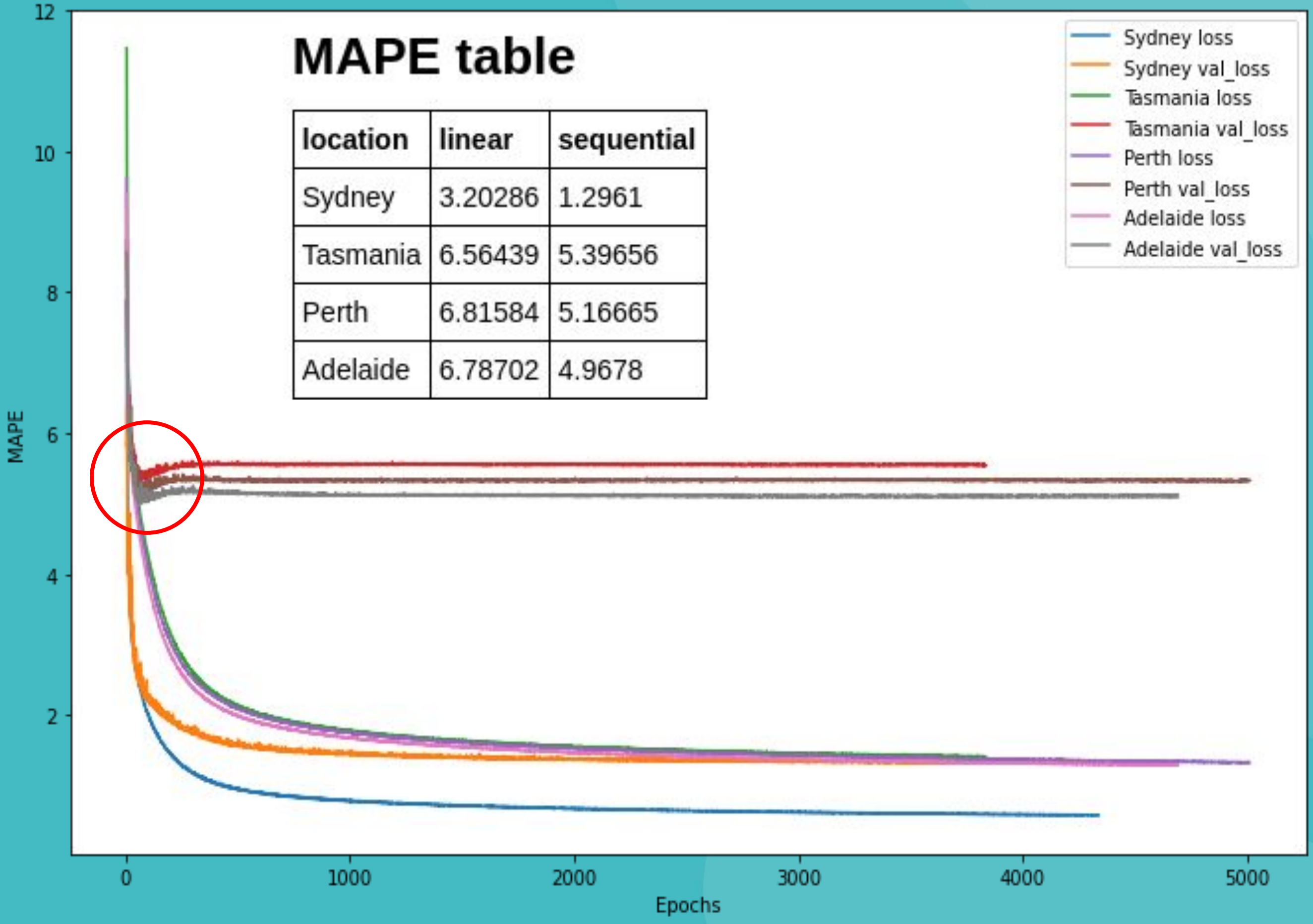
# TRAINING

With this final model architecture, I trained 4 different models, one for each location.

I trained until there was no improvement in loss. This training was overnight on GPU

**Issue**: **Overfitting**

**Solution**: Retrain network and save model when **val_loss is at minimum**




**MAPE table**

| location | linear | sequential |
|---|---|---|
| Sydney | 3.20286 | 1.2961 |
| Tasmania | 6.56439 | 5.39656 |
| Perth | 6.81584 | 5.16665 |
| Adelaide | 6.78702 | 4.9678 |

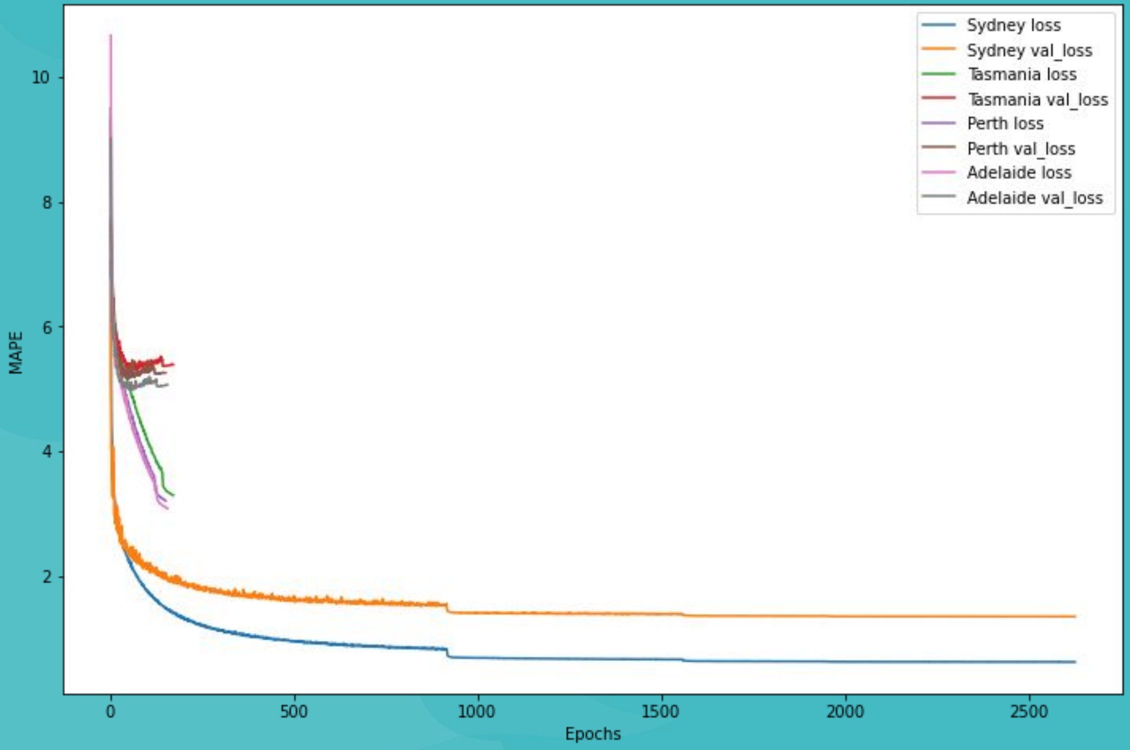
# SAVE THE MODEL WHEN BEST VAL_LOSS IS ACHIEVED

```
def train_model(file,patience=100,verbose=0):
    model = models.Sequential()
    model.add(layers.Dense(1024, input_dim=32, activation = "relu"))
    model.add(layers.Dropout(0.1))
    model.add(layers.Dense(2048, activation = "relu"))
    model.add(layers.Dense(1024, activation = "relu"))
    model.add(layers.Dense(17, activation = 'linear'))

    earlyStopping = EarlyStopping(monitor='val_loss', patience=patience, verbose=0, mode='min')
    mcp_save = ModelCheckpoint('tmp.h5', save_best_only=True, monitor='val_loss', mode='min')
    reduce_lr_loss = ReduceLROnPlateau(monitor='val_loss', factor=0.2, patience=70,
                                       verbose=1, min_delta=1e-5, mode='min')
    logger = TqdmCallback(verbose=verbose)

    model.compile("adam",loss="mean_absolute_percentage_error")
    loss_hist = model.fit(xs_trains[file], ys_trains[file], epochs = 15000, shuffle=True,
                          verbose=0, validation_data = (xs_tests[file], ys_tests[file]),
                          batch_size=2048, callbacks=[earlyStopping, mcp_save, reduce_lr_loss,logger])
    model.load_weights('tmp.h5')
    os.remove('tmp.h5')
    return model,loss_hist
```

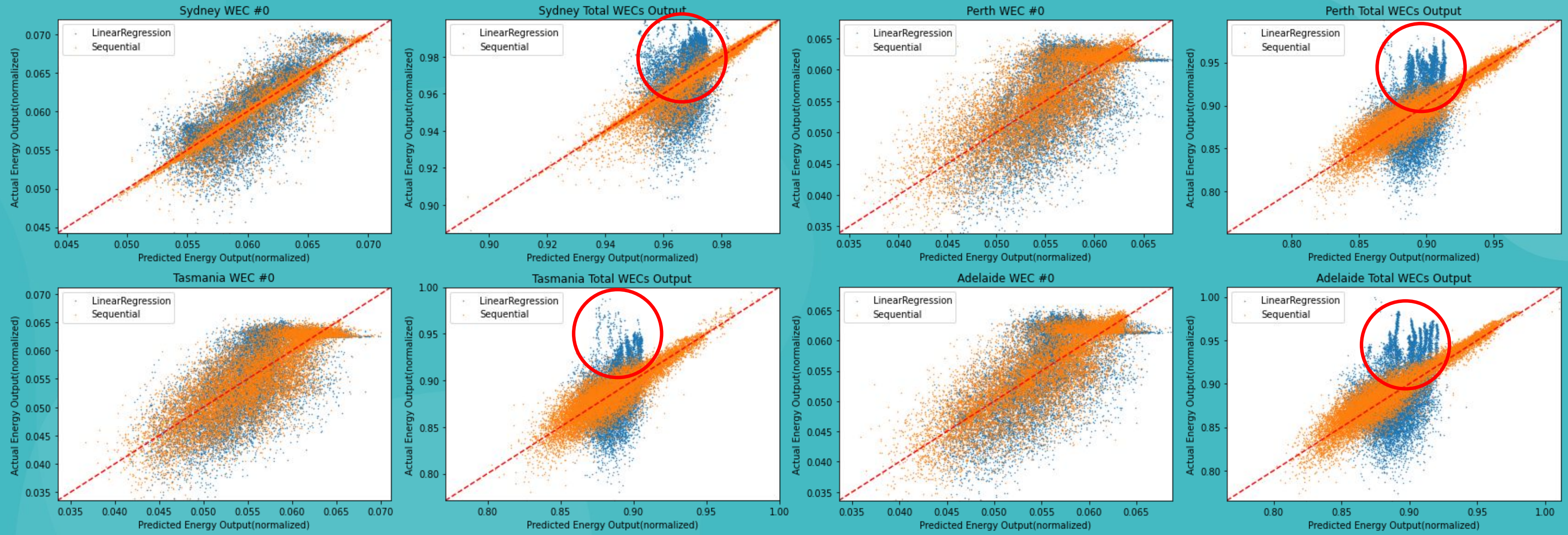
# TRAINING MODELS TOOK A FRACTION OF THE TIME



RESULTS

| MAPE | LINEAR | NON-LINEAR OLD | NON-LINEAR NEW |
|---|---|---|---|
| SYDNEY | 3.20% | 1.30% | 1.33% |
| TASMANIA | 6.56% | 5.40% | 5.18% |
| PERTH | 6.82% | 5.17% | 5.07% |
| ADELAIDE | 6.79% | 4.97% | 4.92% |

# GRAPHING THE ERROR




Linear Regression clearly is not picking up on some structure

# CROSS-TESTING

| Trained On \ Tested On | SYDNEY | TASMANIA | PERTH | ADELAIDE |
|---|---|---|---|---|
| SYDNEY | 1.33% | 12.41% | 11.49% | 11.00% |
| TASMANIA | 15.54% | 5.18% | 6.57% | 6.78% |
| PERTH | 14.64% | 6.41% | 5.07% | 6.36% |
| ADELAIDE | 14.68% | 6.75% | 6.35% | 4.92% |

# CONCLUSION

**MULTIVARIATE REGRESSION** was better than expected!
→ it could easily identify how a WEC's power output is correlated with its own coordinates
→ but it's linear so $P_{total}$ was basically guesswork (*frequency driven*) and structure (*constructive / destructive interference*) was not being captured
→ almost always fails when **power output is high**, possibly since it was too simple to pick up on **constructive interference**

**NEURAL NETWORK** made it better, especially in the case of predicting $P_{total}$
→ but with many, MANY testing iterations
→ turns out we didn't actually have to penalize $\sum_{i=1}^{16} P_i \neq P_{total}$