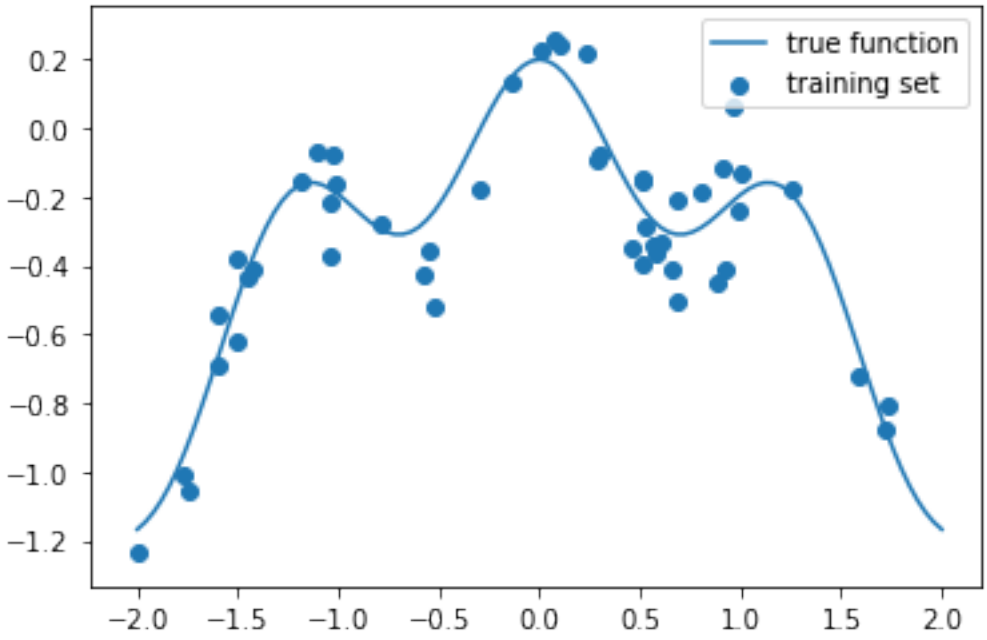
20200219_examples

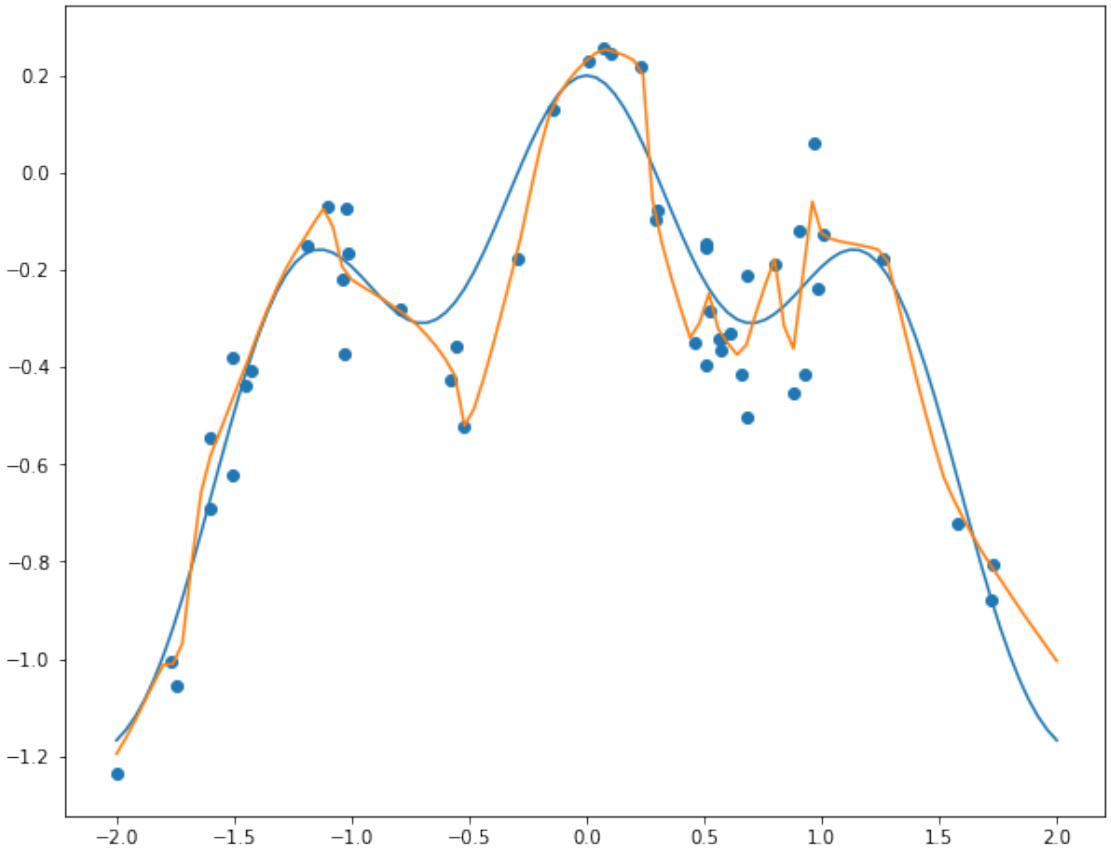
February 18, 2020

0.1 Data generation



0.2 Model with so many hidden layers (5) and units (512 per layer)

[11]:
```
hidden_units = [512, 512, 512, 512, 512]
np.random.seed(65392)
b_init_seeds = np.random.randint(1, 1e6, size = len(hidden_units)+1)
w_init_seeds = np.random.randint(1, 1e6, size = len(hidden_units)+1)
model_somany = fit_model_ex1(hidden_units, b_init_seeds, w_init_seeds,␣
 ↪return_history = False)
plot_layers(model_somany, hidden_legend=False, include_hidden = False)
```

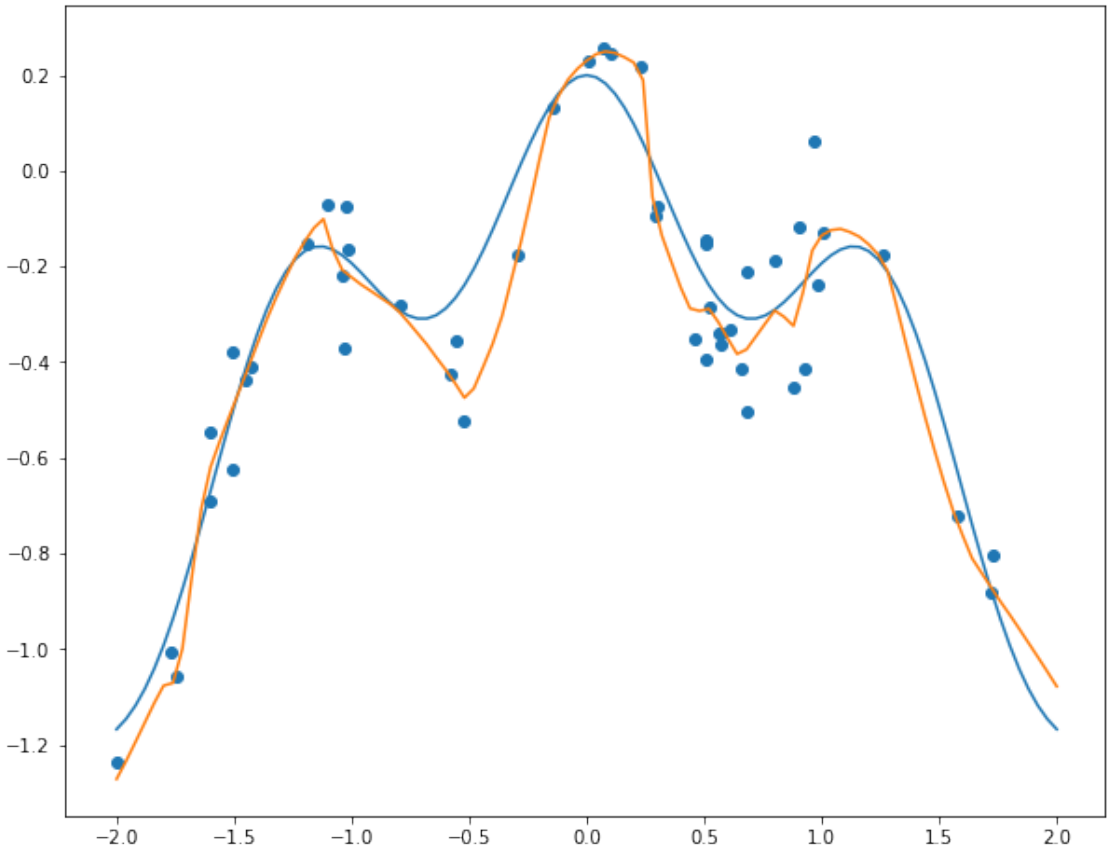

0.2
0.0
−0.2
−0.4
−0.6
−0.8
−1.0
−1.2
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0

0.3 Three ideas to fix:

0.3.1 Idea 1: smaller models – let's try fewer units per layer

Why? The bigger your model, the more flexibility/capacity to overfit.

[10]:
```
hidden_units = [256, 256, 256, 256, 256]
np.random.seed(65392)
b_init_seeds = np.random.randint(1, 1e6, size = len(hidden_units)+1)
w_init_seeds = np.random.randint(1, 1e6, size = len(hidden_units)+1)
(model_256, history_256) = fit_model_ex1(hidden_units, b_init_seeds,␣
 ↪w_init_seeds, num_epochs = 1000, return_history = True)
plot_layers(model_256, hidden_legend=False, include_hidden = False)
```

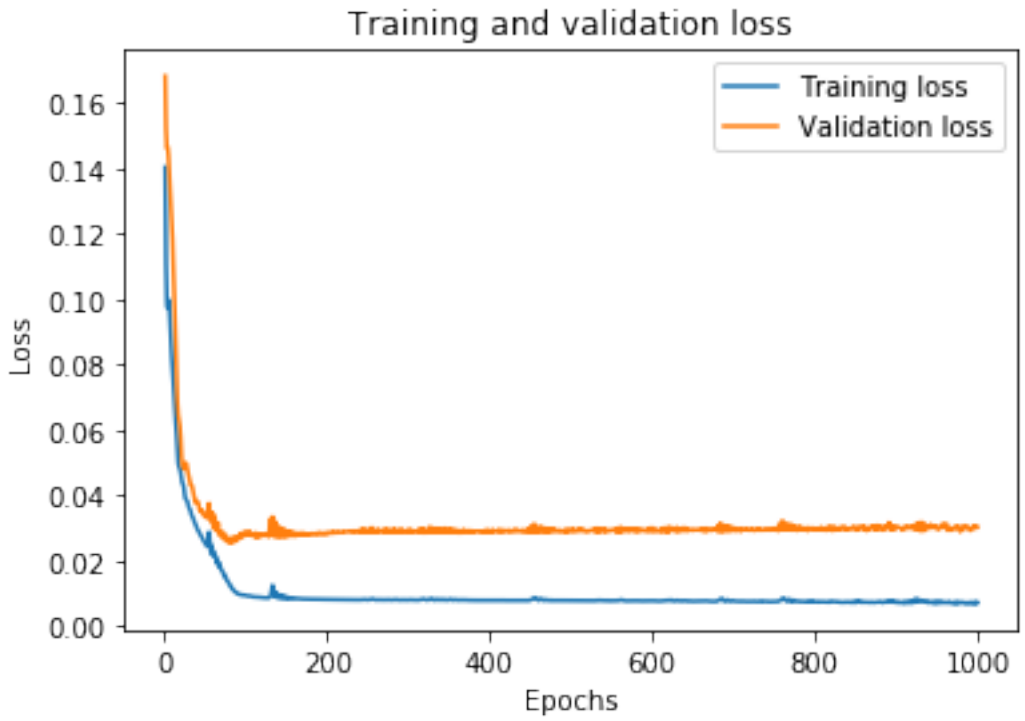


0.3.2 Idea 2: Don't train as long (early stopping)

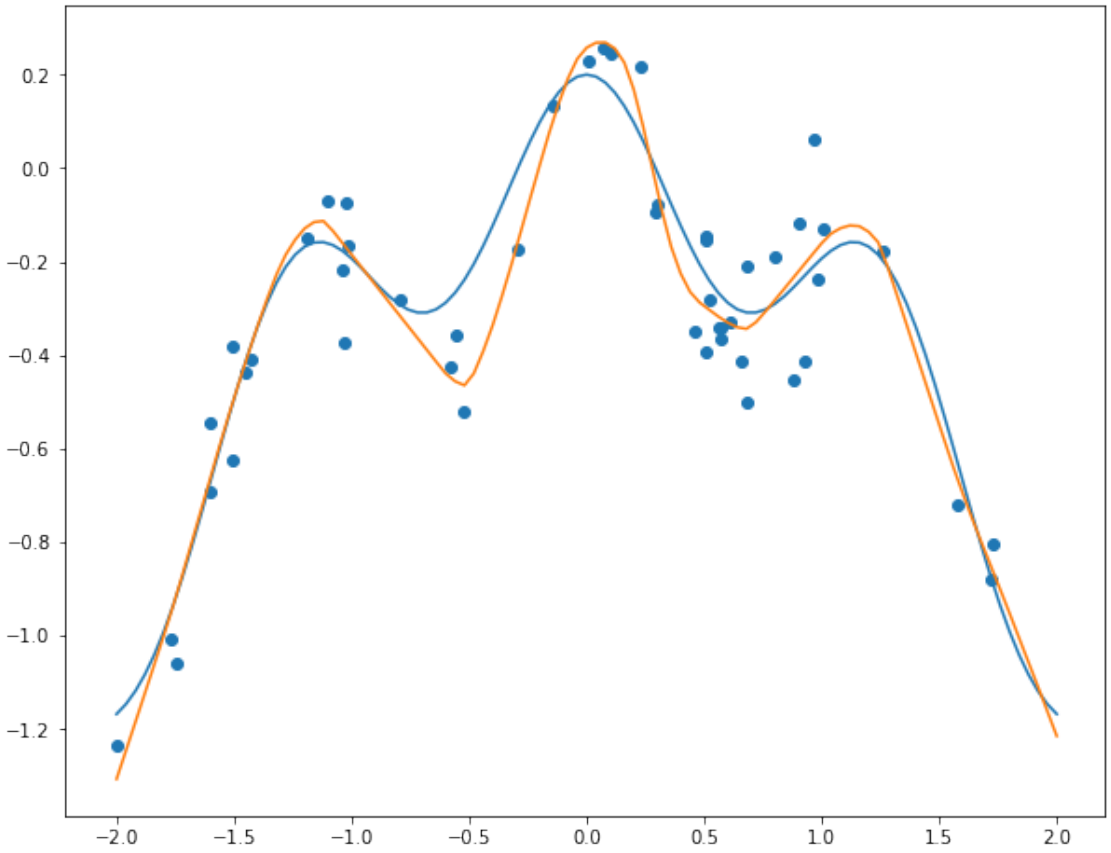
Why? The longer you train your model, the more tuned you are to your specific training set. At some point, validation set performance may start dropping off.

To figure out when we might reasonably stop, look at the plot of training and validation set performance:

[13]:
```
history_dict = history_256.history
history_dict.keys()
loss_values = history_dict['loss']
val_loss_values = history_dict['val_loss']

epochs = range(1, len(loss_values) + 1)

plt.plot(epochs, loss_values, label='Training loss')
plt.plot(epochs, val_loss_values, label='Validation loss')
plt.title('Training and validation loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

plt.show()
```



[14]:
```
hidden_units = [256, 256, 256, 256, 256]
np.random.seed(65392)
b_init_seeds = np.random.randint(1, 1e6, size = len(hidden_units)+1)
w_init_seeds = np.random.randint(1, 1e6, size = len(hidden_units)+1)
model_256_less_train = fit_model_ex1(hidden_units, b_init_seeds, w_init_seeds,␣
 ↪num_epochs = 100)
plot_layers(model_256_less_train, hidden_legend=False, include_hidden = False)
```



This works, but note that:

- early stopping is almost equivalent to L_2 regularization (not obvious - proved using multivariate Taylor series, maybe we'll do this later)
- early stopping can be a little risky; the right stopping point might be quite different with different initial parameter values. This affects L_2 regularization less.

For these reasons, in general I recommend training until convergence and using another idea for regularization – but this is good to know about and is also a partial motivation for dropout below.

0.3.3 Idea 3: Combine predictions from multiple models (build an ensemble)

Fitting a bunch of models and averaging their predictions is **always** a good idea:

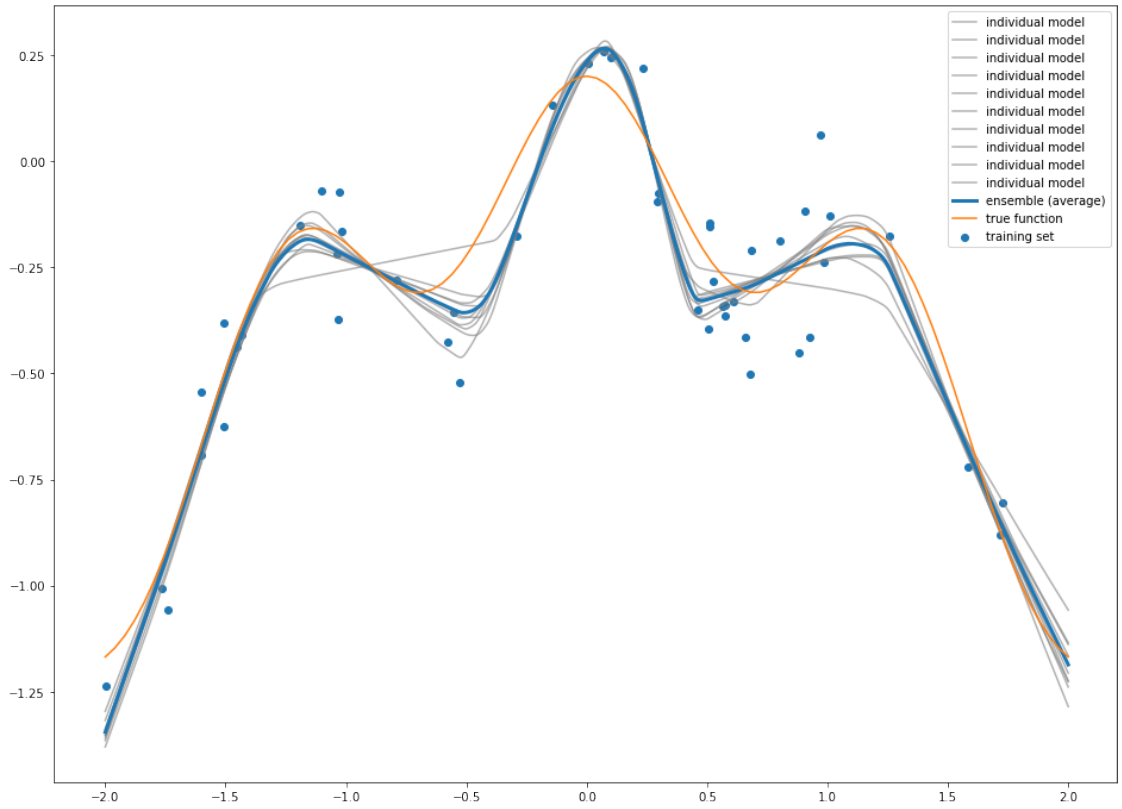
- This can reduce some of the variability from random initializations.
- If you use different model structures, can get some of the benefits from each model structure.
- Essentially always yields a fairly small improvement in overall performance. Ensembles win competitions where a small gain in performance matters, but is not always done in production models in companies.

See Stat 340 and/or machine learning in CS for more theoretical discussion in terms of variance reduction.

[0]:
```
hidden_units = [256, 256, 256, 256, 256]
np.random.seed(65392)

model_fits = []

for i in range(10):
  b_init_seeds = np.random.randint(1, 1e6, size = len(hidden_units)+1)
  w_init_seeds = np.random.randint(1, 1e6, size = len(hidden_units)+1)
  model_fits.append(fit_model_ex1(hidden_units, b_init_seeds, w_init_seeds,␣
 ↪num_epochs = 100))
```

[19]:
```
fig, ax = plt.subplots(1, 1, figsize=(16, 12))

plot_x_grid = np.linspace(-2, 2, 401)

preds = np.zeros((10, 401))
for i in range(10):
  preds[i, :] = model_fits[i].predict(plot_x_grid.T)[:, 0]
  ax.plot(plot_x_grid, preds[i, :], c="gray", alpha = 0.6, label = "individual␣
 ↪model")

ax.plot(plot_x_grid, np.mean(preds, axis = 0), linewidth = 3, label = "ensemble␣
 ↪(average)")
ax.plot(x_grid[0, :], true_f, label = "true function")
ax.scatter(train_x[:, 0], train_y[:, 0], label = "training set")
plt.legend()
```

[19]:
```
<matplotlib.legend.Legend at 0x7f05936fdf28>
```



0.3.4 Train and validation set performance of all four approaches

Here for the score labelled 'model_256_less_train' I have taken the average MSE across all the models we fit above with 256 units per layer and 500 epochs of training.

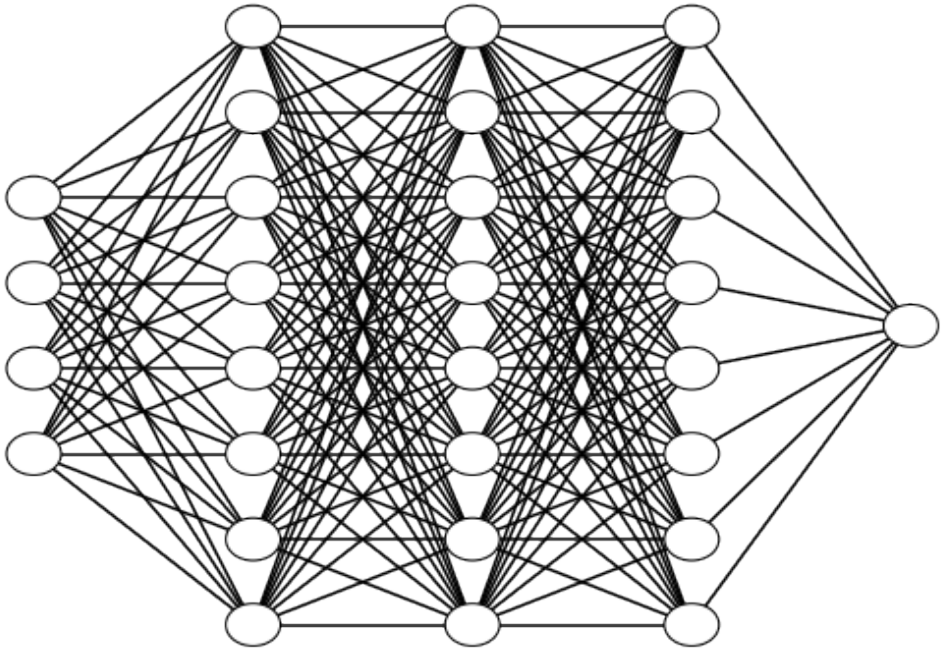
[20]:

	train_mse	val_mse
model_somany	0.00626838	0.0313612
model_256	0.00697594	0.0302323
model_256_less_train	0.0113222	0.0279638
ensemble	0.0105538	0.0270576

0.4 Dropout

Dropout can be viewed as combining the above ideas. Here's how it works:

- Specify your big model:

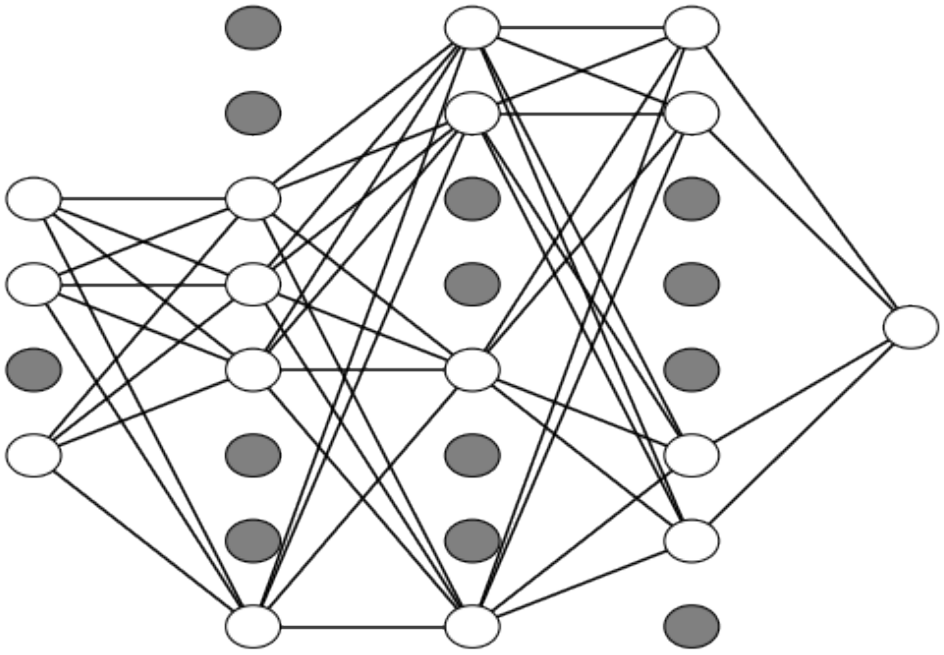


- On each step of gradient descent, **randomly** select a subset of units in each hidden or input layer to "drop out" of the network.
  - Effectively, the activations for these units are set to 0.
  - These units make no contribution to the prediction or to the gradients for this step of gradient descent.
  - This means we are effectively fitting a smaller neural network for this iteration.
- The **dropout rate** (between 0 and 1) is the probability that each unit is dropped out.
  - On average, this proportion of units will be dropped out in each layer – could be more or less on any given sample.

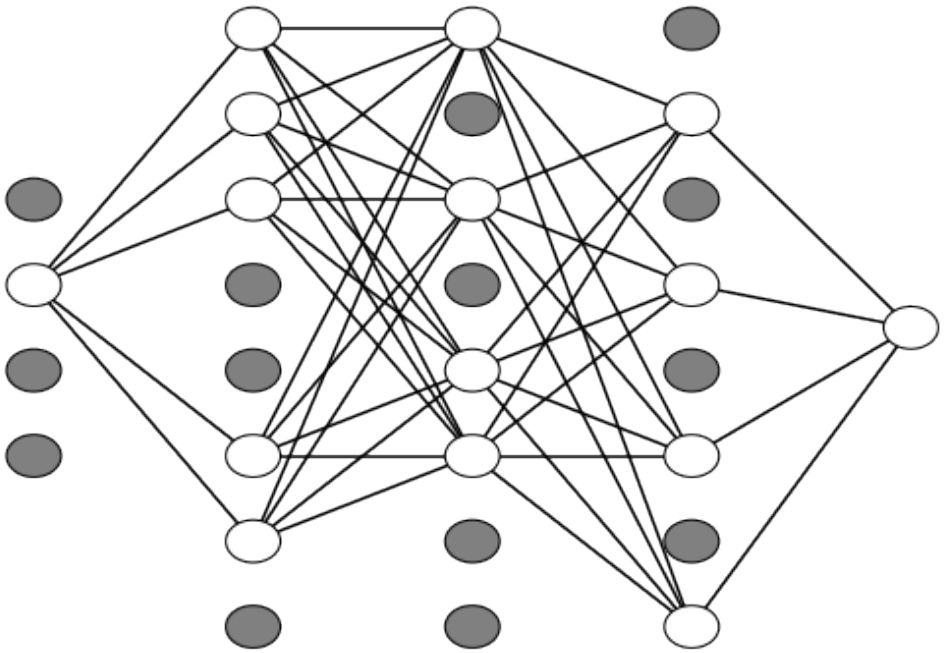
[95]:
```
dropout_rate = 0.5
np.random.seed(964)
print("sample 1: " + str(np.random.binomial(n = 1, p = dropout_rate, size = (4,␣
↪1))))
```

```
print("sample 2: " + str(np.random.binomial(n = 1, p = dropout_rate, size = (4,␣
 ↪1))))
print("sample 3: " + str(np.random.binomial(n = 1, p = dropout_rate, size = (4,␣
 ↪1))))
```

```
sample 1: [[1]
 [0]
 [1]
 [0]]
sample 2: [[0]
 [1]
 [0]
 [0]]
sample 3: [[1]
 [0]
 [0]
 [1]]
```



- On successive steps of gradient descent, we choose a different random subset of units to drop out.



- When making predictions for a validation or test set, scale down the activations from each unit by **multiplying by the dropout rate when making test set predictions**.
  - Suppose dropout rate is 0.5.
  - On average, each unit is present only half the time during training.
  - On average, its contribution during forward propagation is 0.5 times its activation output.
- Equivalently, make no adjustment during test phase and scale up activations by **dividing by the dropout rate during training**
  - With no adjustment, each unit would contribute only 0.5 times its activation output on average.
  - If we multiply by 2 (divide by 0.5) each unit contributes its activation output on average.

Why does dropout work?

- This is similar to building an **ensemble** because it's like we're fitting a bunch of different models (each gradient descent step is effectively working with a different model) and then combining them.
- This is similar to using a **reduced model size** because each gradient descent step only looks at/is able to use a subset of units. As far as optimization is concerned, on any given step there is less model flexibility available
- This is similar to using **early stopping** because no one of the smaller networks is trained fully to convergence. Each smaller model is only trained for one step of gradient descent, not 1000.

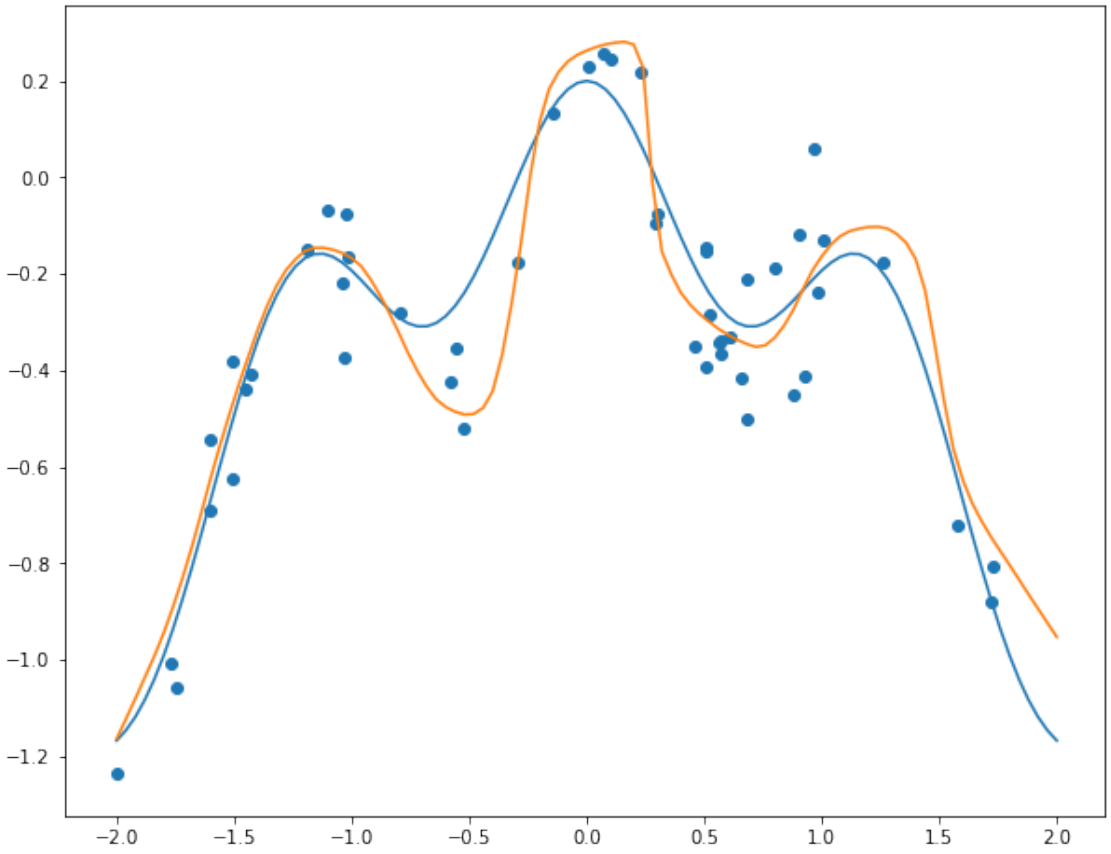
0.4.1 Dropout in Keras

[105]:
```
dropout_rate = 0.2

np.random.seed(8746)
b_init_seeds = np.random.randint(1, 1e6, size = 6)
w_init_seeds = np.random.randint(1, 1e6, size = 6)

dropout_model = models.Sequential()

# you could add dropout on inputs but that doesn't make sense with only 1 x
#dropout_model.add(layers.Dropout(rate = dropout_rate))

# add hidden layers
b_initializer = initializers.RandomNormal(seed=b_init_seeds[0])
w_initializer = initializers.RandomNormal(seed=w_init_seeds[0])
dropout_model.add(layers.Dense(1024,
    activation = 'relu',
    input_shape = (1,),
    bias_initializer = b_initializer,
    kernel_initializer = w_initializer))

dropout_model.add(layers.Dropout(rate = dropout_rate))

b_initializer = initializers.RandomNormal(seed=b_init_seeds[1])
w_initializer = initializers.RandomNormal(seed=w_init_seeds[1])
dropout_model.add(layers.Dense(1024,
    activation = 'relu',
    bias_initializer = b_initializer,
    kernel_initializer = w_initializer))

dropout_model.add(layers.Dropout(rate = dropout_rate))

b_initializer = initializers.RandomNormal(seed=b_init_seeds[2])
```

```
w_initializer = initializers.RandomNormal(seed=w_init_seeds[2])
dropout_model.add(layers.Dense(1024,
    activation = 'relu',
    bias_initializer = b_initializer,
    kernel_initializer = w_initializer))

dropout_model.add(layers.Dropout(rate = dropout_rate))

b_initializer = initializers.RandomNormal(seed=b_init_seeds[3])
w_initializer = initializers.RandomNormal(seed=w_init_seeds[3])
dropout_model.add(layers.Dense(1024,
    activation = 'relu',
    bias_initializer = b_initializer,
    kernel_initializer = w_initializer))

dropout_model.add(layers.Dropout(rate = dropout_rate))

b_initializer = initializers.RandomNormal(seed=b_init_seeds[4])
w_initializer = initializers.RandomNormal(seed=w_init_seeds[4])
dropout_model.add(layers.Dense(1024,
    activation = 'relu',
    bias_initializer = b_initializer,
    kernel_initializer = w_initializer))

# add output layer
b_initializer = initializers.RandomNormal(seed=b_init_seeds[5])
w_initializer = initializers.RandomNormal(seed=w_init_seeds[5])
dropout_model.add(layers.Dense(1,
    activation = 'linear',
    bias_initializer = b_initializer,
    kernel_initializer = w_initializer))

# compile and fit model
dropout_model.compile(optimizer = 'adam', loss = 'mean_squared_error', metrics =␣
 ↪['mean_squared_error'])

dropout_model.fit(train_x, train_y,
  epochs = 1000,
  batch_size = train_x.shape[0],
  verbose = 0)
```

[105]:
```
<keras.callbacks.History at 0x7f05928a8e10>
```

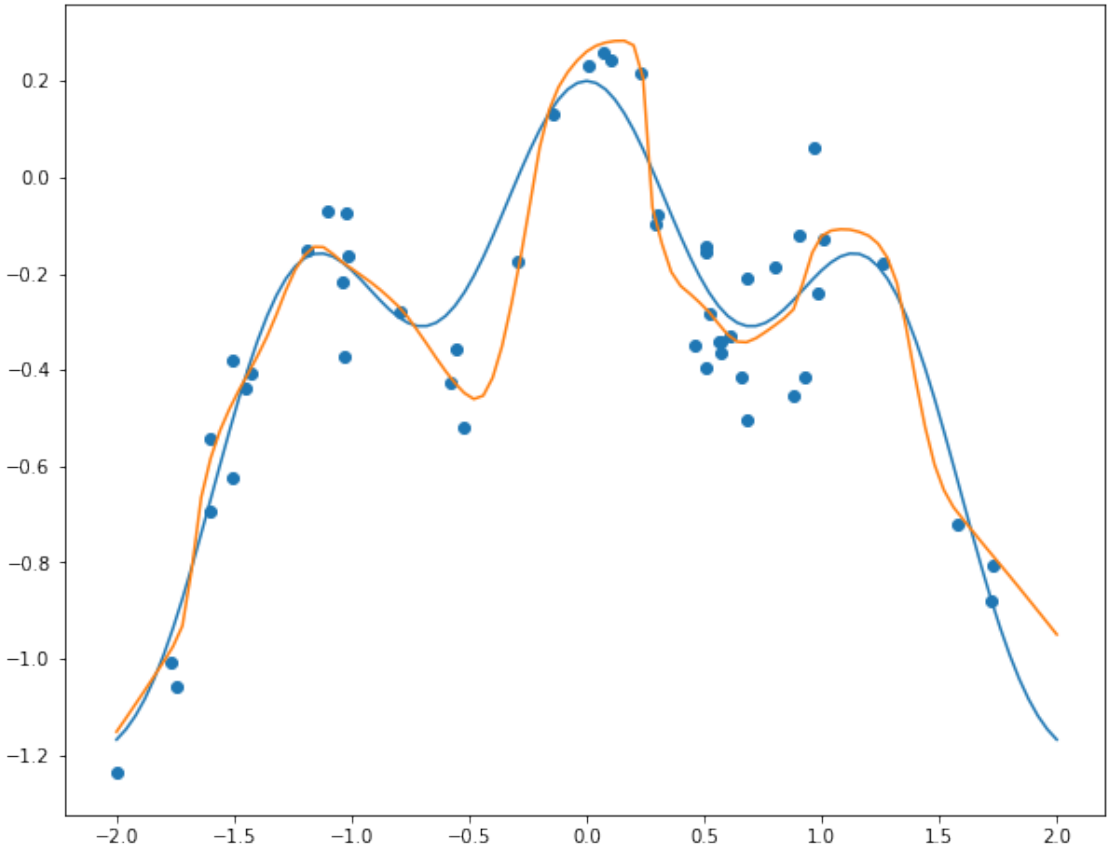
[106]:
```
plot_layers(dropout_model, hidden_legend=False, include_hidden = False)
print(dropout_model.evaluate(train_x, train_y))
print(dropout_model.evaluate(val_x, val_y))
```




```
50/50 [==============================] - 1s 15ms/step
[0.010358586870133878, 0.010358586870133878]
10000/10000 [==============================] - 1s 55us/step
[0.03549703559279442, 0.03549703559279442]
```

This looks ok but doesn't have the best training or validation set performance. The estimate is above the data in a few places; I suspect that this is because of the scaling adjustment.

Same model as above, but with dropout rate 0.05:



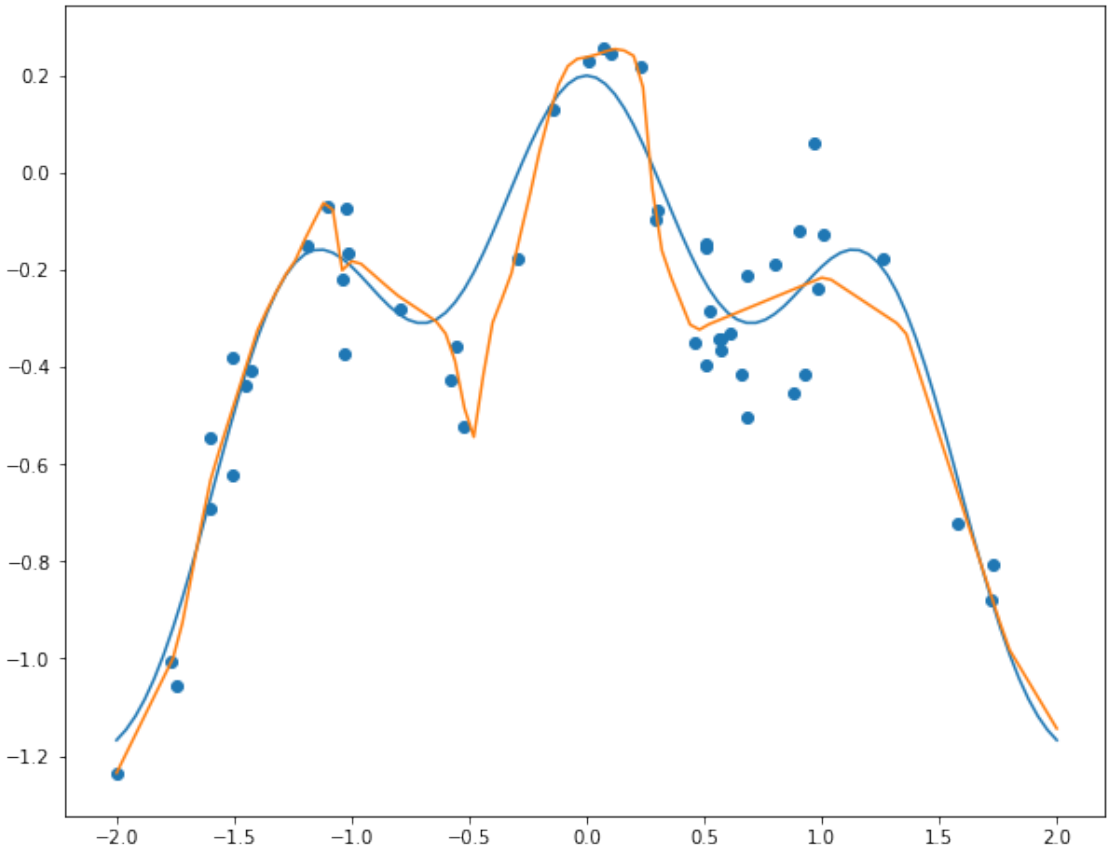

```
50/50 [==============================] - 1s 17ms/step
[0.008277178294956684, 0.008277178294956684]
10000/10000 [==============================] - 1s 54us/step
[0.032732396107912065, 0.032732396107912065]
```

1 Exploding and Vanishing Gradients

1.1 Observation: neural network performance can go down as you add more layers (for reasons other than overfitting)

1.1.1 13 layers of 32 units each

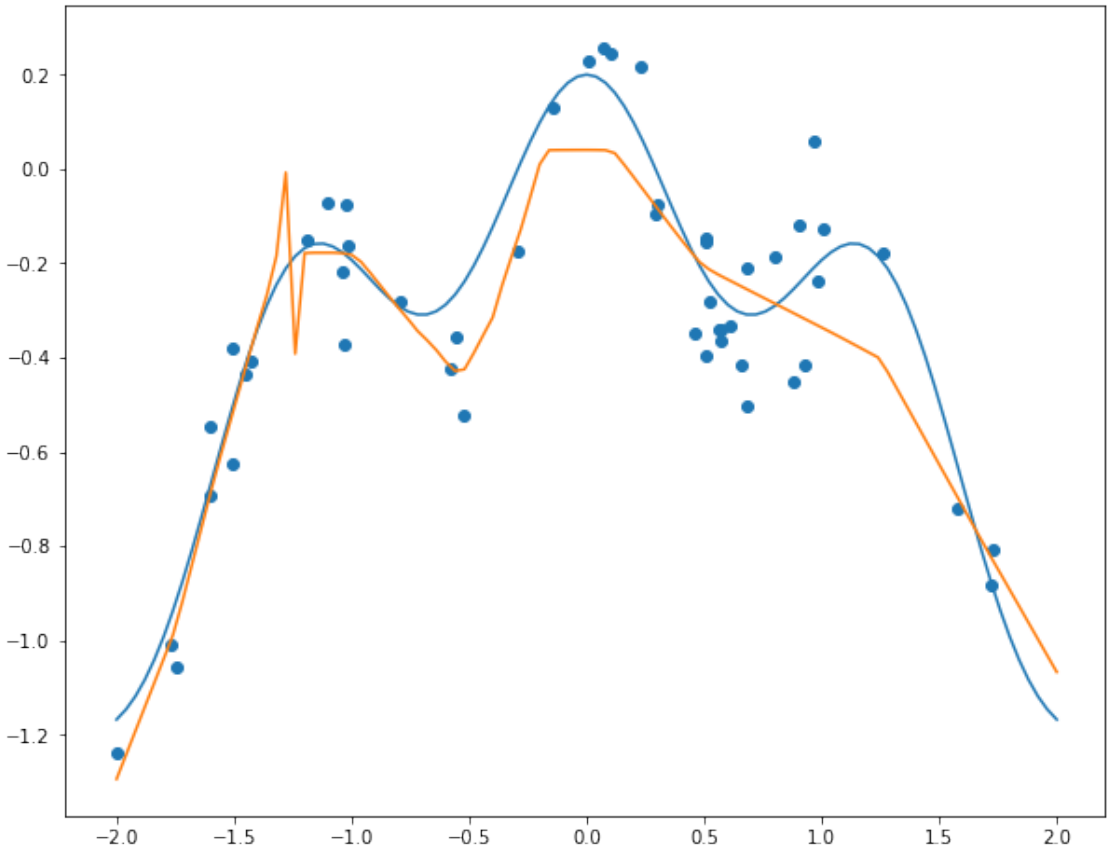
[24]:
```
hidden_units = [32, 32, 32, 32, 32, 32, 32, 32, 32, 32, 32, 32, 32]
np.random.seed(3575)
b_init_seeds = np.random.randint(1, 1e6, size = sum(hidden_units)+1)
w_init_seeds = np.random.randint(1, 1e6, size = sum(hidden_units)+1)
model_13layers = fit_model_ex1(hidden_units, b_init_seeds, w_init_seeds)
plot_layers(model_13layers, hidden_legend=False, include_hidden = False)
```



This model has slightly overfit the training data, but it's not too bad.

1.1.2 14 layers of 32 units each

[22]:
```
hidden_units = [32, 32, 32, 32, 32, 32, 32, 32, 32, 32, 32, 32, 32, 32]
np.random.seed(3575)
b_init_seeds = np.random.randint(1, 1e6, size = sum(hidden_units)+1)
w_init_seeds = np.random.randint(1, 1e6, size = sum(hidden_units)+1)
model_14layers = fit_model_ex1(hidden_units, b_init_seeds, w_init_seeds)
plot_layers(model_14layers, hidden_legend=False, include_hidden = False)
```



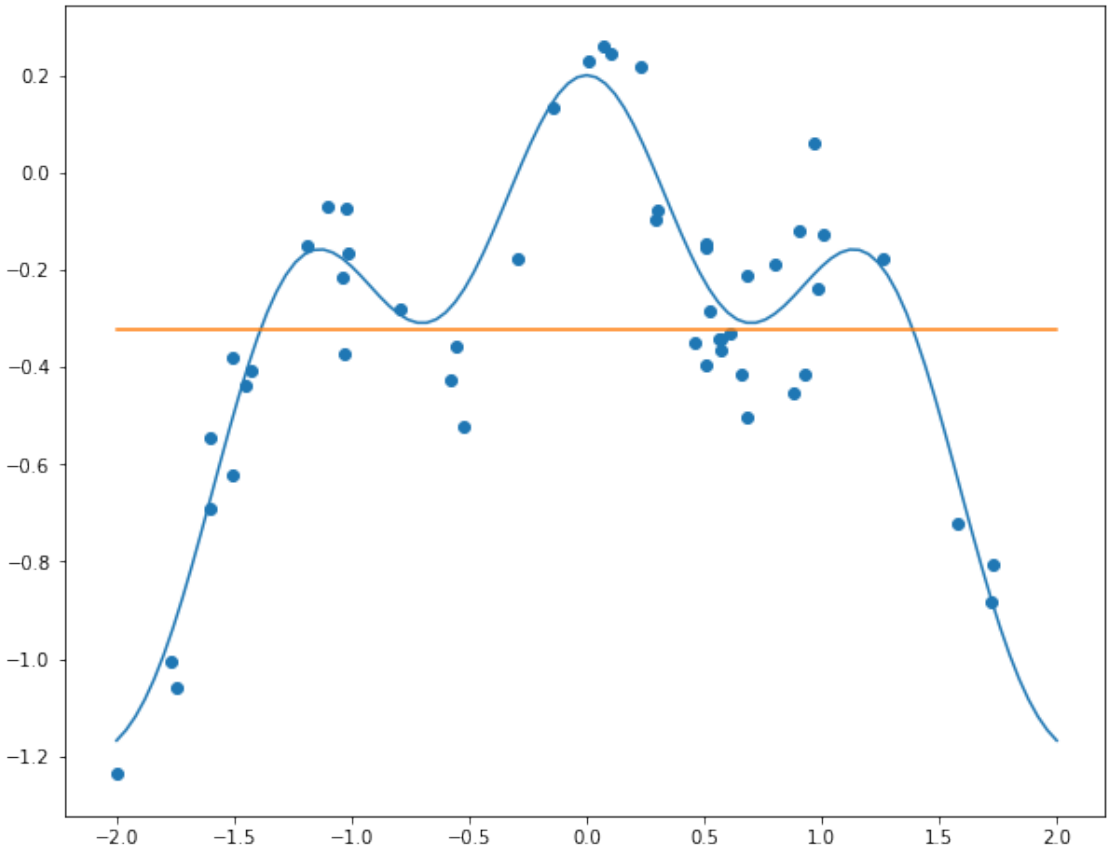
This model definitely looks worse than the model with 13 layers.

Note that the model does not look worse because it has overfit the data:

- There are spikes, but they don't correspond to training set data.
- Near 0 the estimated function is below the training data. This is not what we'd see if we were overfitting.

1.1.3 15 layers of 32 units each

[23]:
```
hidden_units = [32, 32, 32, 32, 32, 32, 32, 32, 32, 32, 32, 32, 32, 32, 32]
np.random.seed(3575)
b_init_seeds = np.random.randint(1, 1e6, size = sum(hidden_units)+1)
w_init_seeds = np.random.randint(1, 1e6, size = sum(hidden_units)+1)
model_15layers = fit_model_ex1(hidden_units, b_init_seeds, w_init_seeds)
plot_layers(model_15layers, hidden_legend=False, include_hidden = False)
```



Estimation has totally failed. Why?

Let's inspect the gradients for the weight parameters.

Here's a summary of our model:

[36]:
```
model_15layers.summary()
```

```
Model: "sequential_13"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_164 (Dense)            (None, 32)                64
_________________________________________________________________
dense_165 (Dense)            (None, 32)                1056
_________________________________________________________________
dense_166 (Dense)            (None, 32)                1056
_________________________________________________________________
dense_167 (Dense)            (None, 32)                1056
_________________________________________________________________
dense_168 (Dense)            (None, 32)                1056
_________________________________________________________________
dense_169 (Dense)            (None, 32)                1056
_________________________________________________________________
dense_170 (Dense)            (None, 32)                1056
_________________________________________________________________
dense_171 (Dense)            (None, 32)                1056
_________________________________________________________________
dense_172 (Dense)            (None, 32)                1056
_________________________________________________________________
dense_173 (Dense)            (None, 32)                1056
_________________________________________________________________
dense_174 (Dense)            (None, 32)                1056
_________________________________________________________________
dense_175 (Dense)            (None, 32)                1056
_________________________________________________________________
dense_176 (Dense)            (None, 32)                1056
_________________________________________________________________
dense_177 (Dense)            (None, 32)                1056
_________________________________________________________________
dense_178 (Dense)            (None, 32)                1056
_________________________________________________________________
dense_179 (Dense)            (None, 1)                 33
=================================================================
Total params: 14,881
Trainable params: 14,881
Non-trainable params: 0
_________________________________________________________________
```

Each of those layers has an associated array (tensor) of weight parameters:

```
weight_tensors:
[<tf.Variable 'dense_164/kernel:0' shape=(1, 32) dtype=float32_ref>,
<tf.Variable 'dense_165/kernel:0' shape=(32, 32) dtype=float32_ref>,
<tf.Variable 'dense_166/kernel:0' shape=(32, 32) dtype=float32_ref>,
<tf.Variable 'dense_167/kernel:0' shape=(32, 32) dtype=float32_ref>,
<tf.Variable 'dense_168/kernel:0' shape=(32, 32) dtype=float32_ref>,
<tf.Variable 'dense_169/kernel:0' shape=(32, 32) dtype=float32_ref>,
<tf.Variable 'dense_170/kernel:0' shape=(32, 32) dtype=float32_ref>,
<tf.Variable 'dense_171/kernel:0' shape=(32, 32) dtype=float32_ref>,
<tf.Variable 'dense_172/kernel:0' shape=(32, 32) dtype=float32_ref>,
<tf.Variable 'dense_173/kernel:0' shape=(32, 32) dtype=float32_ref>,
<tf.Variable 'dense_174/kernel:0' shape=(32, 32) dtype=float32_ref>,
<tf.Variable 'dense_175/kernel:0' shape=(32, 32) dtype=float32_ref>,
<tf.Variable 'dense_176/kernel:0' shape=(32, 32) dtype=float32_ref>,
<tf.Variable 'dense_177/kernel:0' shape=(32, 32) dtype=float32_ref>,
<tf.Variable 'dense_178/kernel:0' shape=(32, 32) dtype=float32_ref>,
<tf.Variable 'dense_179/kernel:0' shape=(32, 1) dtype=float32_ref>]
```

Here's a print out of the gradients $\frac{\partial}{\partial w^{[l]}} J(b, w)$ for each layer (I cut out some of the middle ones):

[45]:
```
w_gradients_by_layer
```

[45]:
```
[array([[-5.1128570e-11,  1.4558806e-11,  8.8109138e-12,  1.0374611e-11,
          0.0000000e+00, -4.1665487e-11,  2.3008637e-11,  1.4040640e-11,
          0.0000000e+00,  2.9587180e-11,  1.2040632e-11,  3.5399409e-11,
          8.6199811e-11,  6.9892765e-11,  3.9537221e-11, -3.5880274e-11,
          1.7834813e-11,  2.4683039e-11,  3.5862643e-11, -2.6082041e-11,
          0.0000000e+00,  0.0000000e+00,  1.9786824e-12, -3.0482141e-11,
          0.0000000e+00,  2.4237433e-11,  4.3163390e-11,  0.0000000e+00,
         -1.2263329e-11,  0.0000000e+00, -4.7673445e-12, -5.4395251e-12]],
       dtype=float32),
 array([[ 1.8212415e-13,  4.1658879e-14, -1.1628683e-12, ...,
          2.0910598e-13,  0.0000000e+00,  0.0000000e+00],
        [ 8.2808630e-12,  0.0000000e+00, -6.6680841e-11, ...,
          0.0000000e+00,  8.6835504e-16,  0.0000000e+00],
        [ 3.0822716e-13,  0.0000000e+00, -2.7909812e-12, ...,
          0.0000000e+00,  1.5795109e-16,  0.0000000e+00],
        ...,
        [ 0.0000000e+00,  0.0000000e+00,  0.0000000e+00, ...,
          0.0000000e+00,  0.0000000e+00,  0.0000000e+00],
        [ 1.9591384e-11,  1.9041973e-12, -1.3143256e-10, ...,
          9.5580835e-12,  3.6077512e-17,  0.0000000e+00],
        [ 2.9374606e-12,  0.0000000e+00, -2.5802249e-11, ...,
          0.0000000e+00,  6.4894921e-16,  0.0000000e+00]], dtype=float32),
 array([[ 2.0683519e-11,  1.9981850e-11, -3.2242306e-11, ...,
         -3.2364400e-10, -1.3208110e-10, -5.5204102e-11],
```

```
        [ 8.0050983e-14,  7.7335328e-14, -1.2478671e-13, ...,
         -1.2525924e-12, -5.1119059e-13, -2.1365525e-13],
        [ 1.4835239e-11,  1.4331966e-11, -2.3125772e-11, ...,
         -2.3213342e-10, -9.4735060e-11, -3.9595098e-11],
        ...,
        [ 2.7588951e-14,  2.6653018e-14, -4.3006770e-14, ...,
         -4.3169629e-13, -1.7617788e-13, -7.3634621e-14],
        [ 1.4896431e-13,  1.4391082e-13, -2.3221158e-13, ...,
         -2.3309093e-12, -9.5125828e-13, -3.9758420e-13],
        [ 0.0000000e+00,  0.0000000e+00,  0.0000000e+00, ...,
          0.0000000e+00,  0.0000000e+00,  0.0000000e+00]], dtype=float32),
 array([[ 1.10491796e-10,  0.00000000e+00,  0.00000000e+00, ...,
          0.00000000e+00,  4.33943881e-10, -2.73915390e-10],
        [ 2.77639300e-10,  0.00000000e+00,  0.00000000e+00, ...,
          0.00000000e+00,  1.09039644e-09, -6.88283430e-10],
        [ 1.24112609e-10,  0.00000000e+00,  0.00000000e+00, ...,
          0.00000000e+00,  4.87437979e-10, -3.07682102e-10],
        ...,
        [ 6.68120073e-11,  0.00000000e+00,  0.00000000e+00, ...,
          0.00000000e+00,  2.62396466e-10, -1.65630717e-10],
        [ 1.23228650e-10,  0.00000000e+00,  0.00000000e+00, ...,
          0.00000000e+00,  4.83966311e-10, -3.05490744e-10],
        [ 1.21087126e-10,  0.00000000e+00,  0.00000000e+00, ...,
          0.00000000e+00,  4.75555817e-10, -3.00181796e-10]], dtype=float32),


(...insert 3 more pages of gradients here...)

 array([[ 0.        ,  0.        ,  0.        , ...,  0.        ,
          0.        ,  0.        ],
        [ 0.        ,  0.        ,  0.00825372, ...,  0.00460549,
         -0.00259274,  0.        ],
        [ 0.        ,  0.        ,  0.05294726, ...,  0.029544  ,
         -0.01663234,  0.        ],
        ...,
        [ 0.        ,  0.        ,  0.        , ...,  0.        ,
          0.        ,  0.        ],
        [ 0.        ,  0.        ,  0.        , ...,  0.        ,
          0.        ,  0.        ],
        [ 0.        ,  0.        ,  0.08230466, ...,  0.0459251 ,
         -0.02585438,  0.        ]], dtype=float32),
 array([[0.0000000e+00, 0.0000000e+00, 0.0000000e+00, ..., 0.0000000e+00,
         0.0000000e+00, 0.0000000e+00],
        [0.0000000e+00, 0.0000000e+00, 0.0000000e+00, ..., 0.0000000e+00,
         0.0000000e+00, 0.0000000e+00],
        [8.6449215e-04, 2.0252685e-03, 5.4745768e-05, ..., 0.0000000e+00,
         0.0000000e+00, 3.2767624e-05],
```

```
       ...,
       [2.9915057e-02, 7.0082784e-02, 1.8944332e-03, ..., 0.0000000e+00,
        0.0000000e+00, 1.1338973e-03],
       [5.5880792e-02, 1.3091339e-01, 3.5387671e-03, ..., 0.0000000e+00,
        0.0000000e+00, 2.1180999e-03],
       [0.0000000e+00, 0.0000000e+00, 0.0000000e+00, ..., 0.0000000e+00,
        0.0000000e+00, 0.0000000e+00]], dtype=float32),
array([[0.93200916],
       [2.6990824 ],
       [3.7008305 ],
       [0.        ],
       [1.04841   ],
       [3.610091  ],
       [0.9690882 ],
       [0.        ],
       [3.0167778 ],
       [0.03568995],
       [1.3375859 ],
       [0.18449391],
       [0.        ],
       [0.2546359 ],
       [0.6377057 ],
       [1.9631177 ],
       [3.1632779 ],
       [2.7090552 ],
       [0.        ],
       [0.3374935 ],
       [3.2116172 ],
       [0.9469584 ],
       [3.6856844 ],
       [0.        ],
       [0.        ],
       [0.        ],
       [0.        ],
       [5.0867157 ],
       [0.        ],
       [0.        ],
       [0.        ],
       [1.8000932 ]], dtype=float32)]
```

That was a lot to take in. Let's summarize by looking at the **largest** gradient value in each layer.

[46]:
```
[np.max(w_grad_one_layer) for w_grad_one_layer in w_gradients_by_layer]
```

[46]:
```
[8.619981e-11,
 7.4668494e-11,
 5.555045e-10,
 1.1582649e-09,
 1.0108434e-08,
 7.581418e-08,
 3.8226204e-07,
 3.2883786e-06,
 2.7240822e-05,
 0.00021533686,
 0.0004904665,
 0.004181428,
 0.02815935,
 0.1306683,
 0.8350754,
 5.0867157]
```

By the time we get to the first layer, the gradients are essentially 0 and provide no information about how to update the weights to achieve a better model fit.

Why?