Models from Linear Regression to Neural Networks

Johannes Härtel

Vrije Universiteit Brussel

February 20, 2024

Data Structures (Recap, Session 1)

pandas

TensorFlow

NumPy

Figure: Some Examples

What are the underlying data structures that we use in MSR? Why and where do we use them?

Optimization (Recap, Session 2)

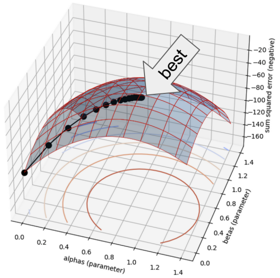

Figure: Finding the peak with Gradient Descent

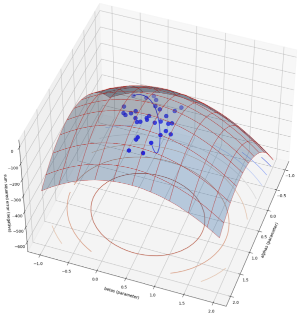

Figure: Approximation of the surface with Hamiltonian Monte Carlo

We defined a simple model, using such data structures. We examined how to optimize the model, finding parameters to fit data.

All models are wrong, but some are useful (George Box)

Blueprints vs. Flexibility when Defining Models

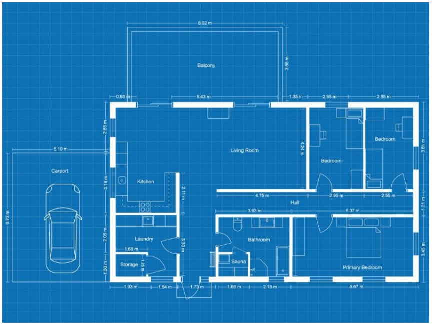

Figure: Model Blueprints

- How flexible are we in defining and using models, or do we need to stick to existing models?

Blueprints vs. Flexibility when Defining Models

- **Blueprints:** Established blueprints are well-supported by **high-level libraries** and **tools**. Examples are:
  - **sklearn** for **linear regression models**.
  - **lme4** for **Generalized Linear Mixed-Effects Models**.
  - **sklearn** for **basic feed-forward neuronal network**.

Blueprints vs. Flexibility when Defining Models

- **Blueprints:** Established blueprints are well-supported by **high-level libraries** and **tools**. Examples are:
  - **sklearn** for **linear regression models**.
  - **lme4** for **Generalized Linear Mixed-Effects Models**.
  - **sklearn** for **basic feed-forward neuronal network**.
- **Flexibility:** If you want to be flexible (or understand), you can use a **low-level library** to implement the models. Examples of low-level APIs are:
  - **Stan** for **Bayesian models**.
  - **PyMC3** for **Bayesian models**.
  - **TensorFlow** for **neural network (models)**.
  - **Keras** for **neural network (models)**.
  - **PyTorch** for **neural network (models)**.

In this session, we will see common how very common **blueprints or types** of models are implemented in different **low and high-level libraries**: that is in **sklearn**, **TensorFlow**, **Keras**, and **STAN**.

Linear Regression (Revisited)

Implementing a linear (regression) model using different libraries.

Simulating Data

```python
import numpy as np # We use numpy.
np.random.seed(0) # To make this reproducible.

# Generate fake—data (simulation).
n = 20 # number of observations.
alpha = 0.9 # Parameter (can be changed).
beta = 0.4 # Parameter (can be changed).
sigma = 0.1 # Parameter (can be changed).

xs = np.random.normal(size = n) # random values for x.
pred = alpha + beta * xs # mean values for y.
ys = np.random.normal(scale = sigma, size = n) + pred # final
     output y.
```

Simulating data that conforms to the definition of a linear regression model (recap session 2).

Simulating Data

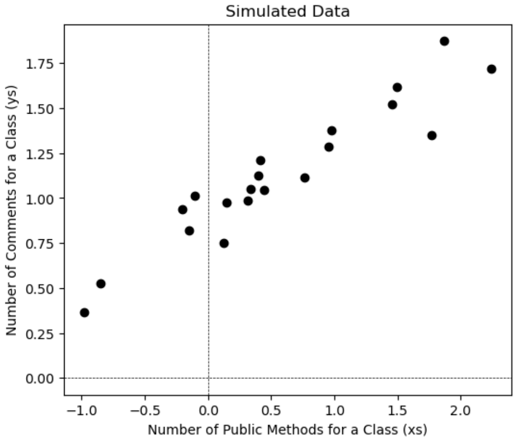

Figure: Simulated data

Simulating data that conforms to the definition of a linear regression model (recap session 2).

Implementing the model

- After having the simulation done ...
- We pretend not to know the parameters *alpha* and *beta* (and *sigma*) from the simulated data.
- We recover it from *xs* and *ys* using a model.
- We use different libraries to implement the model.
- Finally, to test our implementation, we compare if the recovered parameters correspond to the simulated ones.

sklearn

```python
# Implement the model using the scikit—learn library.
from sklearn.linear_model import LinearRegression

model = LinearRegression()
model.fit(xs.reshape(—1, 1), ys) # FYI: Ugly reshape tensor
    needed:—(, from (20) to (20, 1).

# Print model summary (recovered parameters):
print("Intercept (alpha): ", model.intercept_)
print("Coefficient (betas): ", model.coef_)
```

Intercept (alpha): 0.89862396434804 Coefficient (betas): [0.41220911]
Test: See that *alpha* ≈ 0.9 (intercept) and *beta* ≈ 0.4 (coefficient) are correct. 'Correct' means close to the values of the simulation.

TensorFlow (Core)

```
import tensorflow as tf # Using tensorflow core.
# Initialize the parameters (here 'variables').
alpha = tf.Variable(0.0); beta = tf.Variable(0.0)
xs_tf = tf.constant(xs, dtype=tf.float32) # to tensorflow ...
ys_tf = tf.constant(ys, dtype=tf.float32) # to tensorflow.
for i in range(1000): # Minimize 1000 steps.
    with tf.GradientTape() as tp: # Recorde gradients...
        # Define the loss (error) function on our own.
        pred_tf = alpha + beta * xs_tf # mean values for y.
        sum_squared_error = tf.reduce_sum(tf.pow(ys_tf -
            pred_tf, 2))
    # Minimize deriving error with respect to alpha and beta.
    gradients = tp.gradient(sum_squared_error, [alpha, beta])
    alpha.assign_sub(0.01 * gradients[0]) # gradient descent.
    beta.assign_sub(0.01 * gradients[1]) # gradient descent.
```

Intercept (alpha): 0.8986239 Coefficient (betas): 0.41220918

Keras

```
import keras # Using keras API (on top of tensorflow).
import keras.layers as layers

# Single layer and a single output (a linear regression).
# The layer that may realize a linear model is a 'Dense'.
model = keras.Sequential([
        layers.Dense(units=1, input_shape=[1], activation='
            linear')
])

# Optimize using (stochastic) gradient descent optimizer.
model.compile(optimizer='sgd', loss='mean_squared_error')
model.fit(xs, ys, epochs=1000, verbose=0)
```

Intercept (alpha): [0.89856374] Coefficient (betas): [[0.41226682]]
Many details are hidden behind the API.

STAN

Users specify log density functions in the Stan probabilistic programming language and then fit the models to data using:

- full Bayesian statistical inference with MCMC sampling (NUTS, HMC)
- approximate Bayesian inference with variational inference (ADVI)
- penalized maximum likelihood estimation with optimization (L-BFGS)

(Copied from ...)

- **Recap:** We have seen Hamiltonian Monte Carlo (HMC) in session 2.

STAN

Users specify log density functions in the Stan probabilistic programming language and then fit the models to data using:

- full Bayesian statistical inference with MCMC sampling (NUTS, HMC)
- approximate Bayesian inference with variational inference (ADVI)
- penalized maximum likelihood estimation with optimization (L-BFGS)

(Copied from ...)

- **Recap:** We have seen Hamiltonian Monte Carlo (HMC) in session 2.
- **The log (probability) density function** is very similar to our previous error function. Strongly simplified: It does not describe the error, but how ‘(un-)surprised we are’ of seeing data and parameters under a model.

STAN

Users specify log density functions in the Stan probabilistic programming language and then fit the models to data using:

- full Bayesian statistical inference with MCMC sampling (NUTS, HMC)
- approximate Bayesian inference with variational inference (ADVI)
- penalized maximum likelihood estimation with optimization (L-BFGS)

(Copied from ...)

- **Recap:** We have seen Hamiltonian Monte Carlo (HMC) in session 2.
- **The log (probability) density function** is very similar to our previous error function. Strongly simplified: It does not describe the error, but how '(un-)surprised we are' of seeing data and parameters under a model.
- **We are interested in the full shape of the function, not only the minimum/maximum.**

STAN

```
# Using a (c++ like) DSL for model specification.
stan_code = """
data {
    int<lower=0> N; // number of observations.
    vector[N] xs; // input values (array).
    vector[N] ys; // output values (array).
}
parameters { // Parameters we search for.
    real alpha; real beta;
    real<lower=0> sigma;
}
model { // Relationships between data and parameter.
    vector[N] pred;
    pred = alpha + beta * xs;
    ys ~ normal(pred, sigma);
}
"""
```

STAN

```python
import stan # Probabilistic programming language (STAN).

# Run the Hamiltonian Monte Carlo (HMC) sampler to aproximate
    our 'error' function. Here "error" is a bit different
    and called the 'posterior' or '(probability) density'.
# Confusing, I know...
data = {'N': n, 'xs': xs, 'ys': ys}
model = stan.build(stan_code, data=data)
posterior = model.sample(num_chains=1, num_samples=4000)
```

STAN

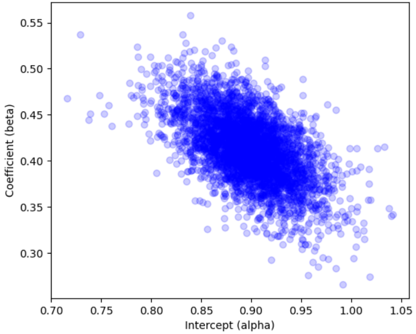

Figure: The results of the Hamiltonian Monte Carlo Sampler: Showing the log (probability) density function in terms of samples of parameter values. The **density of points** reflects our confidence.

The results are a table. All interpretation can be done based on **counting**.

Linear Regression (Extended)

Implementing a linear (regression) model with more than one variable using different libraries.

Simulating Data

```python
n = 70 # number of observations.
alpha = 0.9 # Parameter (can be changed).
betas = np.array([-0.3, 0.7]) # Parameters (can be changed).
sigma = 0.1 # Parameter (can be changed).
m = len(betas) # Length of betas (number of variables).

xs = np.random.normal(size = (n, m)) # random matrix n x m.
# We can generalize this using a matrix multiplication.
pred = alpha + np.matmul(xs, betas)
ys = np.random.normal(scale = sigma, size = n) + pred # final
    output y.
```

Simulating data conforming to a linear model with multiple variables. We can use a matrix multiplication (matrix multiplied by vector).

Simulating Data

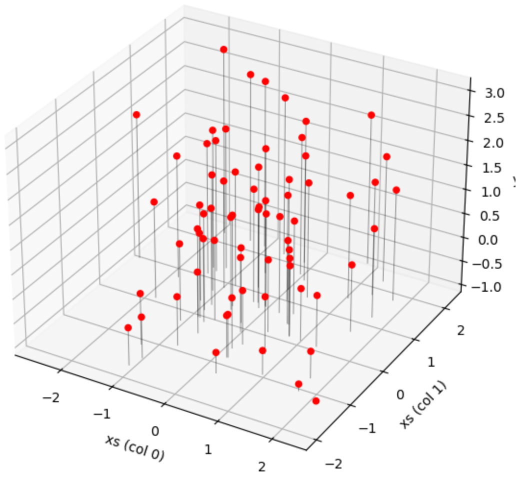

Figure: Simulated data (plot this in 3d when m = 2)

sklearn

```python
# Implement the model using the scikit—learn library.
from sklearn.linear_model import LinearRegression

model = LinearRegression()
model.fit(xs, ys) # FYI: Ugly reshape tensor removed! :—)

# Print model summary (recovered parameters):
print("Intercept (alpha): ", model.intercept_)
print("Coefficient (betas): ", model.coef_)
```

Intercept (alpha): 0.8927441591493339 Coefficient (betas): [-0.31105275 0.71347268] There are no major modifications.

TensorFlow (Core)

```python
# Initialize the parameters (here 'variables').
alpha = tf.Variable(0.0, dtype=tf.float32)
betas = tf.Variable(np.repeat(0.0, m), dtype=tf.float32)

for i in range(1000): # Minimize 1000 steps.
    with tf.GradientTape() as tp: # Recorde gradients...
        # Define the loss (error) function on our own.
        # We need to use matvec since tf.matmul only works
            with 2D tensors (missing correspondence to np,
            this is shit).
        pred_tf = alpha + tf.linalg.matvec(xs_tf, betas)

        sum_squared_error = tf.reduce_sum(tf.pow(ys_tf -
            pred_tf, 2))
```

Intercept (alpha): 0.8927441 Coefficient (betas): [-0.31105274
0.71347266]

Keras

```
model = keras.Sequential([
        layers.Dense(units=1, input_shape=[m], activation='
            linear')
])
```

Intercept (alpha): [0.8937436] Coefficient (betas): [[-0.31128523] [
0.7142051]]

STAN

```
stan_code = """
data {
    int<lower=0> N; // number of observations.
    int<lower=0> M; // number of columns.
    matrix[N, M] xs; // input values (matrix).
    vector[N] ys; // output values (array).
}
parameters { // Parameters we search for.
    real alpha;
    vector[M] beta;
    real<lower=0> sigma;
}
model { // Relationships between data and parameter.
    vector[N] pred;
    pred = alpha + xs * beta;
    ys ~ normal(pred, sigma);
}
```

STAN

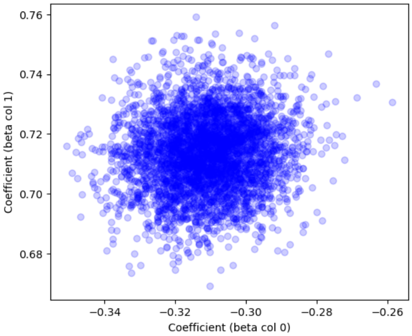

Figure: Here we just show the results for betas (intercept is avoided)

- **Test:** We see that the center is close to the parameters used in the simulation (-0.3 and 0.7).
- If we increase n, this will get more precise.

Going ”Nonlinear”

If our output is binary (like defects), using a linear regression is no real option.

Simulating Data

```python
alpha = 0.9  # Parameter (can be changed).
betas = np.array([0.7]) # Limit to a single beta.
m = len(betas)
xs = np.random.normal(size=(n, m), scale=2)  # random n x m.

# Linear component (lc).
lc = alpha + np.matmul(xs, betas)  # lc (prev: pred_ys).
prev_ys = lc + np.random.normal(size=n)  # ls + uncertianty (
    prev: ys)

# Different types of non—linearities (last with uncertainty).
nl1 = np.maximum(0, lc)
nl2 = 1.0 / (1 + np.exp(—lc))
nl3 = np.minimum(1, np.maximum(0, lc))
ys = np.random.binomial(1,  1.0 / (1 + np.exp(—lc)))
```

This is a code reference. **The next slide is more visual.**

Simulating Data

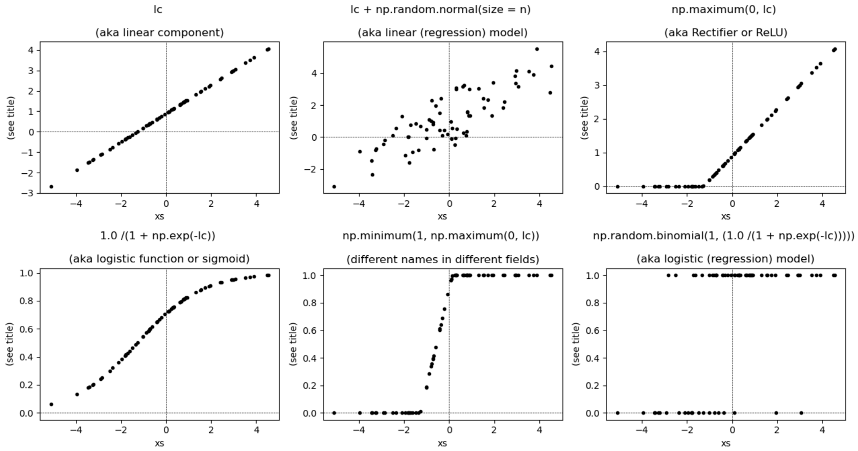

Figure: Simulated data with a linear and non-linear relationship (and optional uncertainty).

Implementing the model

- Comparable to the previous linear and extended linear part of this presentation, we pretend not to know parameters alpha and beta.
- We implement a model that computes both from xs and ys.
- We simulate ys conforming to the default of a logistic regression (same as in the previous plot, right-most, bottom cell).

```
ys = np.random.binomial(1, (1.0 /(1 + np.exp(-lc))))
```

sklearn

```python
# Go logistic using sklearn.
# Implement the model using the scikit—learn library.
from sklearn.linear_model import LogisticRegression

model = LogisticRegression()
model.fit(xs, ys)

# Print model summary (recovered parameters):
print("Intercept (alpha): ", model.intercept_)
print("Coefficient (betas): ", model.coef_)
```

- There are no major modifications, only a changed import.
- **Test:** Checking if the output: Intercept (alpha): [1.21233582] Coefficient (betas): [[0.82101543]] corresponds to the simulation.
- **The model is less accurate than the linear model for the same amount of data.** More data (changing n) fixes this.

TensorFlow (Core)

```
for i in range(1000): # Minimize 1000 steps.
    with tf.GradientTape() as tp: # Recorde gradients...
        # Define the loss (error) function on our own.
        lc_tf = alpha + tf.linalg.matvec(xs_tf, betas)
        pred_tf = tf.math.sigmoid(lc_tf) # sigmoid (or
            softmax, for multiple class).

        # The error is typically the binary cross—entropy.
        binary_cross_entropy = — tf.reduce_sum(ys_tf * tf.
            math.log(pred_tf) + (1 — ys_tf) * tf.math.log(1 —
             pred_tf))
```

- We need to apply sigmoid to the linear component and change to a binary cross-entropy loss.
- **Test:** Checking if the output: Intercept (alpha): 1.2445481 Coefficient (betas): [0.86146253] .

Keras

```
model = keras.Sequential([
        layers.Dense(units=1, input_shape=[m], activation='
            sigmoid')
])
# Optimize using (stochastic) gradient descent optimizer.
# Loss needs to be changed.
model.compile(optimizer='sgd', loss='binary_crossentropy')
model.fit(xs, ys, epochs=1000, verbose=0)
```

- Some things work out of the box here.
- We need to change the **loss and the activation function**.
- **Test:** Checking if the output: Intercept (alpha): [1.2355958] Coefficient (betas): [[0.84700495]] .

STAN

```
stan_code = """
data {
    int<lower=0> N; // number of observations.
    int<lower=0> M; // number of columns.
    matrix[N, M] xs; // input values (matrix).
    array[N] int<lower=0, upper=1> ys; // output.
}
parameters { // Parameters we search for.
    real alpha;
    vector[M] beta;
}
model { // Relationships between data and parameter.
    vector[N] pred;
    pred = alpha + xs * beta;
    ys ~ bernoulli_logit(pred);
}
"""
```

STAN

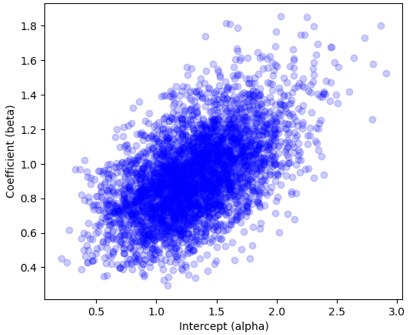

Figure: Samples for alpha and beta parameters

- **Test:** We see that the center is close to 0.9 (alpha) and 0.7 (beta).
- However, we also see that there is a lot of uncertainty.

Feed-forward Neural Network.

We now stack linear models followed by a non-linearity and produce the most basic form of a neural network.

Preface

- We finally **cross the border to true ML**.

Preface

- We finally **cross the border to true ML**.
- We are **not** interested in the parameters anymore. We cannot **interpret our ‘parameter stacks’ forming neural networks**.

Preface

- We finally **cross the border to true ML**.
- We are **not** interested in the parameters anymore. We cannot **interpret our ‘parameter stacks’ forming neural networks**.
- We are only interested in how good we can predict ys from xs.

Preface

- We finally **cross the border to true ML**.
- We are **not** interested in the parameters anymore. We cannot **interpret our ‘parameter stacks’ forming neural networks**.
- We are only interested in how good we can predict ys from xs.
- Again, we use different libraries to implement the model.

Preface

- We finally **cross the border to true ML**.
- We are **not** interested in the parameters anymore. We cannot **interpret our ‘parameter stacks’ forming neural networks**.
- We are only interested in how good we can predict ys from xs.
- Again, we use different libraries to implement the model.
- Again, we use simulated data to test our implementation.

Simulating Data

```python
n = 1500 # number of observations.
hidden = 16 # number of hidden units for one hidden layer.
m = 1 # number of input columns.

betas_1 = np.random.normal(size=(m, hidden))
bias_1 = np.random.normal(size=hidden)

betas_2 = np.random.normal(size= (hidden, 1))
bias_2 = np.random.normal(size=1)
```

Simulating Data

```python
# random input matrix n x m.
xs = np.random.normal(size = (n, m))

# We walk through the layers and produces our final ys
ys = xs # Input
ys = np.matmul(ys, betas_1) + bias_1 # linear model.
ys = np.maximum(0, ys) # Non—linearity.
ys = np.matmul(ys, betas_2) + bias_2 # linear model.
ys = ys[:, 0] # Output
```

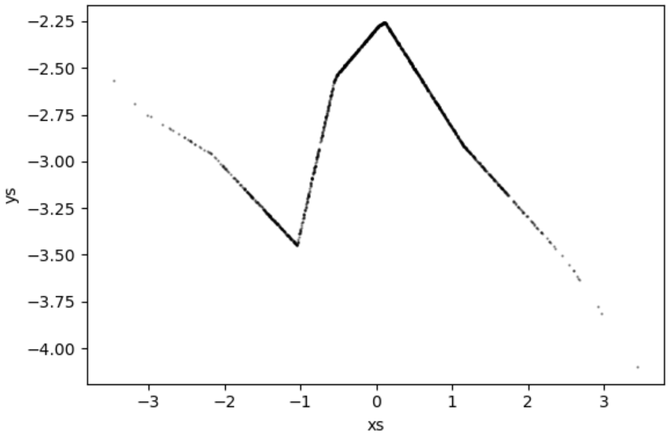

Figure: Simulated data with an arbitrary relationship.

When removing the **fixed seed** in the simulation, the plot will look much different on every run.

Implementing the model

- We now implement the model using different libraries.
- Occasionally, I explore some options on how to optimize and define the model.

sklean

```python
# We use the scikit-learn library to implement the model.
from sklearn.neural_network import MLPRegressor
xs_plot = np.linspace(min(xs), max(xs), 100)

# We explore two networks structures and two solvers.
for hidden_layer_sizes, solver in [((16,),'sgd'), ((16,),'adam'), ((16,16,16),'sgd'), ((16,16,16), "adam")]:

    model = MLPRegressor(random_state=1, hidden_layer_sizes=hidden_layer_sizes, activation='relu', solver=solver, max_iter = 1000).fit(xs, ys)

    ys_plot = model.predict(xs_plot)
```

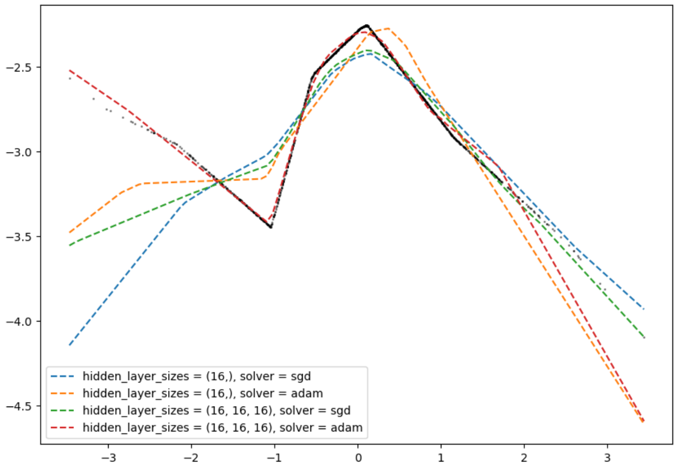

Figure: Showing how this model fits to the data.

Optimizing using stochastic gradient descent (sgd) is not good enough. We **change to adam**. We need to give the model **more layers** (compared to the simulation) so that it fits properly.

TensorFlow (Core)

```python
# Initialize the parameters (here 'variables').
bias_1 = tf.Variable(np.random.normal(size=(hidden,), scale
    =0.1), dtype=tf.float32)
betas_1 = tf.Variable(np.random.normal(size=(m, hidden),
    scale=0.1), dtype=tf.float32)
bias_2 = tf.Variable(np.random.normal(size=(1,), scale=0.1),
    dtype=tf.float32)
betas_2 = tf.Variable(np.random.normal(size=(hidden, 1),
    scale=0.1), dtype=tf.float32)

xs_tf = tf.constant(xs, dtype=tf.float32) # to tensorflow
ys_tf = tf.constant(ys, dtype=tf.float32) # to tensorflow
```

TensorFlow (Core)

```
for i in range(1000): # Minimize 1000 steps.
    with tf.GradientTape() as tp: # Recorde gradients...
        # Define the loss (error) function on our own.
        pred_ys_tf = xs_tf
        pred_ys_tf = tf.matmul(pred_ys_tf, betas_1) + bias_1
        pred_ys_tf = tf.nn.relu(pred_ys_tf)
        pred_ys_tf = tf.matmul(pred_ys_tf, betas_2) + bias_2
        pred_ys_tf = pred_ys_tf[:, 0]
        mean_square_error = tf.reduce_mean(tf.square(ys_tf -
            pred_ys_tf)) # Mean squared error.
    gradients = tp.gradient(mean_square_error, [bias_1,
        bias_2, betas_1, betas_2])
    bias_1.assign_sub(0.2 * gradients[0]) # learn rate ~ 0.2.
    bias_2.assign_sub(0.2 * gradients[1])
    betas_1.assign_sub(0.2 * gradients[2])
    betas_2.assign_sub(0.2 * gradients[3])
```

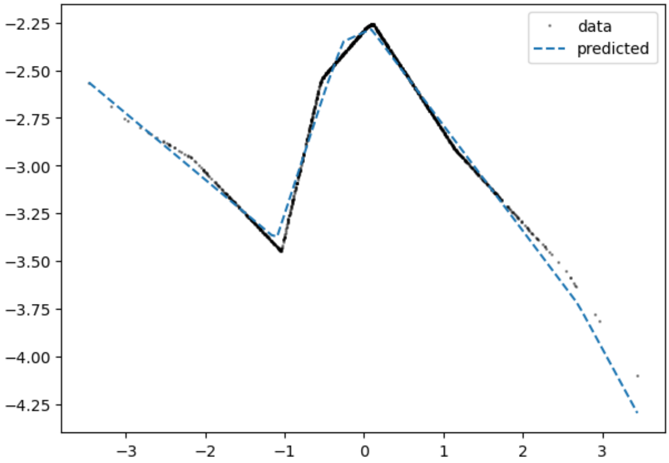

Figure: Showing how this model fits to the data.

Here I noticed that **gradient descent is good enough**. I just needed to adjust the learning rate and trick a bit the **initialization of the layers**.

Keras

```
model = keras.Sequential([
        layers.Dense(units=hidden, input_shape=[m],
            activation='relu'),
        layers.Dense(units=1, activation='linear')
])
# SGD.
optimizer = keras.optimizers.SGD(learning_rate=0.1)
model.compile(optimizer=optimizer, loss='mean_squared_error')
model.fit(xs, ys, epochs=100, verbose=0)
```

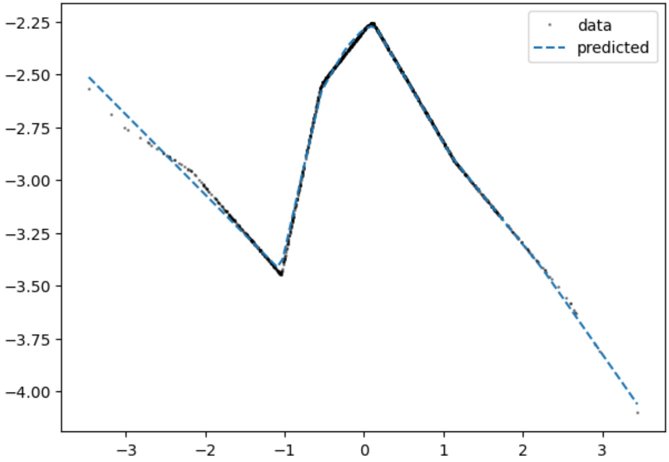

Figure: Showing how this model fits to the data.

STAN

Don’t do what follows this slide. *Stan is not made for the implementation of neural networks. I just want to show that it is possible, since the underlying technology is truly related.*

STAN

```
# This is highly forbidden. DON'T DO THIS AT HOME!
stan_code = """
data {
    int<lower=0> N; // number of observations.
    int<lower=0> M; // number of columns.
    int<lower=0> H; // number of hidden units.
    matrix[N, M] xs; // input values (matrix).
    vector[N] ys; // output values (array).
}
parameters { // Parameters we search for.
    vector[H] bias_1;
    matrix[M, H] betas_1;
    real bias_2;
    matrix[H,1] betas_2;
}
"""
```

STAN

```
# This is highly forbidden. DON'T DO THIS AT HOME!
stan_code += """
model { // Relationships between data and parameter.
    matrix[N, H] l1 = xs * betas_1;
    for (i in 1:N) l1[i] = to_row_vector(to_vector(l1[i]) +
        bias_1);
    matrix[N, H] l2 = inv_logit(l1); // Close to relu
    matrix[N, 1] l3 = l2 * betas_2;
    for(i in 1:N) l3[i] = l3[i] + bias_2;
    // Initialize the parameters (priors).
    to_vector(bias_1) ~ std_normal();
    to_vector(betas_1) ~ std_normal();
    to_vector(betas_2) ~ std_normal();
    bias_2 ~ normal(0, 1);

    ys ~ normal(to_vector(l3), 1); // Shit on sigma!
}
```

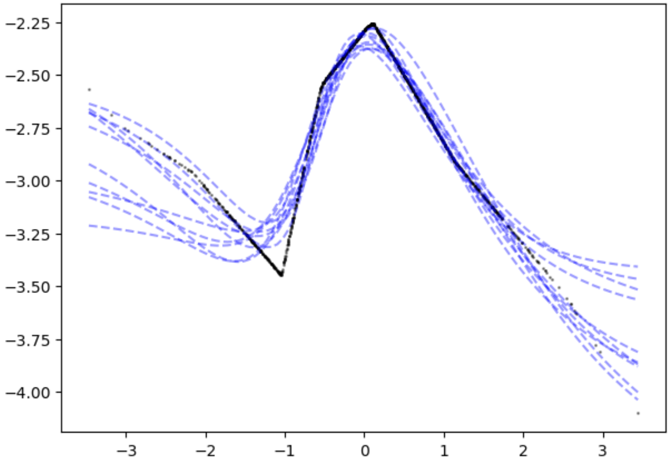

Figure: This is the neural network implemented in STAN (**Don't do this at home**).

The **Relu** *was replaced by* **sigmoid** *(also called inv(erted) logit). I did not find Relu in the Stan documentation.*

End