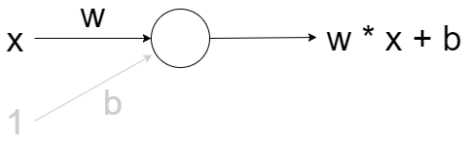
Самая простая нейросеть

Самой простой из возможных конфигураций нейросетей является один нейрон с одним входом и одним выходом без активации (или можно сказать с линейной активацией f(x) = x):



N.B. Как видите, на вход сети подаются два значения — x и единица. Последняя необходима для того, чтобы ввести смещение b. Во всех популярных фреймворках входная единица уже неявно присутствует и не задаётся пользователем отдельно. Поэтому здесь и далее будем считать, что на вход подаётся одно значение.

Несмотря на свою простоту эта архитектура уже позволяет делать линейную регрессию, т.е. приближать функцию прямой линией (часто с минимизацией среднеквадратического отклонения). Пример очень важный, поэтому предлагаю разобрать его максимально подробно.

In [1]:

```
import matplotlib.pyplot as plt
import numpy as np
```

In [2]:

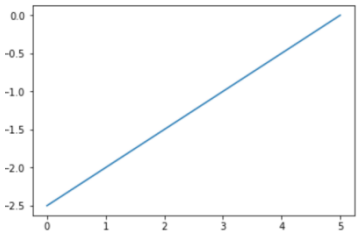
```
# накидываем тысячу точек от -3 до 3
x = np.linspace(-3, 3, 1000).reshape(-1, 1)
```

In [3]:

```
# задаём линейную функцию, которую попробуем приблизить нашей нейронной сетью
def f(x):
    return 2 * x + 5
```

In [4]:

```
def plot_start_line():
    plt.plot((0,5),(-2.5,0))
plot_start_line()
plt.show()
```



In [5]:

```
f = np.vectorize(f)
```

In [6]:

```
# вычисляем вектор значений функции
y = f(x)
```

In [8]:

```
from keras.models import Sequential
from keras.layers import Dense

def baseline_model():
    # создаём модель нейросети, используя Keras
    model = Sequential()
    model.add(Dense(1, input_dim=1, activation='linear'))
    model.compile(loss='mean_squared_error', optimizer='sgd')
    return model
```

In [9]:

```
def training_net(x,y,epochs,verbose):
    # тренируем сеть
    model = baseline_model()
    model.fit(x=x, y=y, epochs=epochs, verbose = verbose)
    return model

model_trained = training_net(x,y,100,0)
```

In [10]:

```
def log_out_weights(model_trained):
    # выводим веса на экран
    for layer in model_trained.layers:
        weights = layer.get_weights()
        print(weights)

log_out_weights(model_trained)
```

```
[array([[ 2.00000048]], dtype=float32), array([ 4.99998856], dtype=float32)]
```

In [11]:

```
def plot_linspace():
    plt.scatter(x, y, color='black', antialiased=True)
```

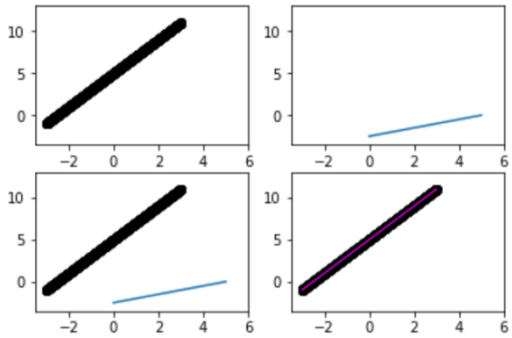
In [12]:

```
def approx(xlim_min,xlim_max,ylim_min,ylim_max):
    # отрисовываем результат приближения нейросетью поверх исходной функции
    fig = plt.figure()

    ax1 = fig.add_subplot(221)
    plot_linspace()
    plt.xlim(xlim_min,xlim_max)
    plt.ylim(ylim_min,ylim_max)

    ax2 = fig.add_subplot(222)
    plot_start_line()
    plt.xlim(xlim_min,xlim_max)
    plt.ylim(ylim_min,ylim_max)

    ax3 = fig.add_subplot(223)
    plot_start_line()
    plot_linspace()
    plt.xlim(xlim_min,xlim_max)
    plt.ylim(ylim_min,ylim_max)

    ax4 = fig.add_subplot(224)
    plt.scatter(x, y, color='black', antialiased=True)
    plt.plot(x, model_trained.predict(x), color='magenta', linewidth=1, antialiased=True)
    plt.xlim(xlim_min,xlim_max)
    plt.ylim(ylim_min,ylim_max)
```

In [13]:

```
approx(-3.5,6.0,-3.5,13.0)
plt.show()
```

Habr_NN_in_Pictures



Как видите, наша простейшая сеть справилась с задачей приближения линейной функции линейной же функцией на ура. Попробуем теперь усложнить задачу, взяв более сложную функцию:

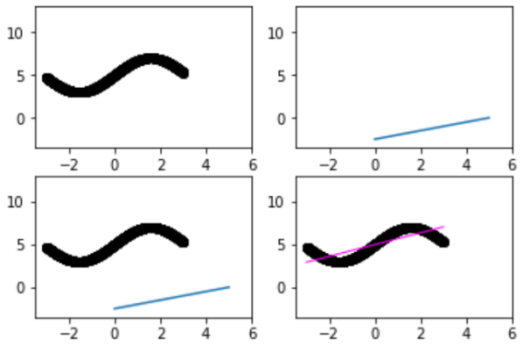
In [14]:

```
def f(x):
    return 2 * np.sin(x) + 5
```

In [15]:

```
y = f(x)
model_trained = training_net(x,y,200,0)
log_out_weights(model_trained)
approx(-3.5,6.0,-3.5,13.0)

plt.show()
```

```
[array([[ 0.6865344]], dtype=float32), array([ 4.98856497], dtype=float32)]
```



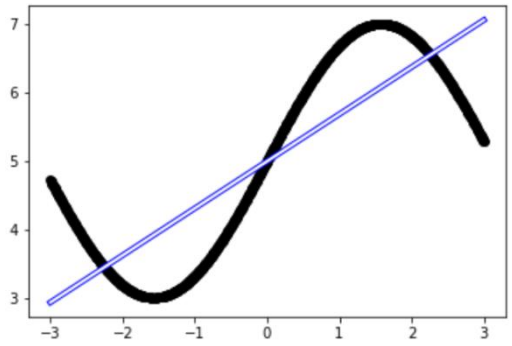
Первое число — это вес w, второе — смещение b. Чтобы убедиться в этом, давайте нарисуем прямую f(x) = w * x + b:

In [16]:

```
def line(x):
    w = model.layers[0].get_weights()[0][0][0]
    b = model.layers[0].get_weights()[1][0]

    return w * x + b
```

In [17]:

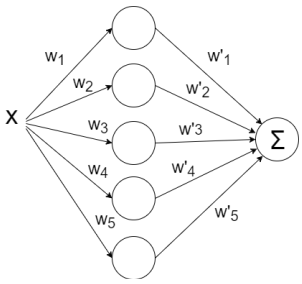
```
y = f(x)
# тренируем сеть
model = baseline_model()
model.fit(x, y, epochs=100, verbose = 0)

# отрисовываем результат приближения нейросетью поверх исходной функции
plt.scatter(x, y, color='black', antialiased=True)
plt.plot(x, model.predict(x), color='blue', linewidth=4, antialiased=True)
plt.plot(x, line(x), color='white', linewidth=2, antialiased=True)
plt.show()
```



Усложняем пример

Хорошо, с приближением прямой всё ясно. Но это и классическая линейная регрессия неплохо делала. Как же захватить нейросетью нелинейность аппроксимируемой функции?

Давайте попробуем накидать побольше нейронов, скажем пять штук. Т.к. на выходе ожидается одно значение, придётся добавить ещё один слой к сети, который просто будет суммировать все выходные значения с каждого из пяти нейронов:

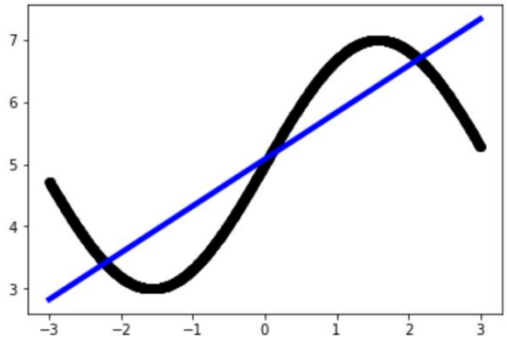


In [64]:

```
def baseline_model_multilayers():
    model = Sequential()
    model.add(Dense(5, input_dim=1, activation='linear'))
    model.add(Dense(1, input_dim=5, activation='linear'))
    model.compile(loss='mean_squared_error', optimizer='sgd')
    return model
```

In [65]:

```
y = f(x)
# тренируем сеть
model = baseline_model_multilayers()
model.fit(x, y, epochs=100, verbose = 0)

# отрисовываем результат приближения нейросетью поверх исходной функции
plt.scatter(x, y, color='black', antialiased=True)
plt.plot(x, model.predict(x), color='blue', linewidth=4, antialiased=True)

plt.show()
```



И… ничего не вышло. Всё та же прямая, хотя матрица весов немного разрослась. Всё дело в том, что архитектура нашей сети сводится к линейной комбинации линейных функций:

f(x) = w1' *(w1* x + b1) +… + w5' (w5 * x + b5) + b

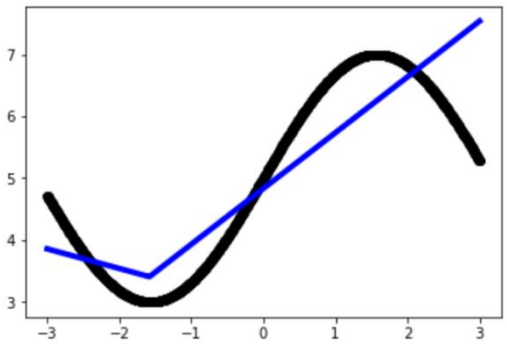
Т.е. опять же является линейной функцией. Чтобы сделать поведение нашей сети более интересным, добавим нейронам внутреннего слоя функцию активации ReLU (выпрямитель, f(x) = max(0, x)), которая позволяет сети ломать прямую на сегменты:

In [66]:

```
def baseline_model_multilayers_relu(nodes):
    model = Sequential()
    model.add(Dense(nodes, input_dim=1, activation='relu'))
    model.add(Dense(1, input_dim=nodes, activation='linear'))
    model.compile(loss='mean_squared_error', optimizer='sgd')
    return model
```

In [67]:

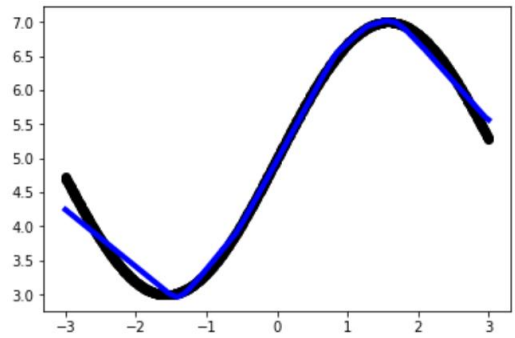
```
y = f(x)
# тренируем сеть
model = baseline_model_multilayers_relu(3)
model.fit(x, y, epochs=100, verbose = 0)

# отрисовываем результат приближения нейросетью поверх исходной функции
plt.scatter(x, y, color='black', antialiased=True)
plt.plot(x, model.predict(x), color='blue', linewidth=4, antialiased=True)

plt.show()
```



Максимальное количество сегментов совпадает с количеством нейронов на внутреннем слое. Добавив больше нейронов можно получить более точное приближение:

In [72]:

```
y = f(x)
# тренируем сеть
model = baseline_model_multilayers_relu(59)
model.fit(x, y, epochs=100, verbose = 0)

# отрисовываем результат приближения нейросетью поверх исходной функции
plt.scatter(x, y, color='black', antialiased=True)
plt.plot(x, model.predict(x), color='blue', linewidth=4, antialiased=True)

plt.show()
```



Дайте больше точности!

Уже лучше, но огрехи видны на глаз — на изгибах, где исходная функция наименее похожа на прямую линию, приближение отстаёт.

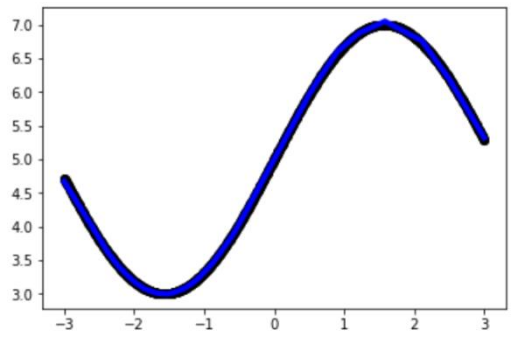
В качестве стратегии оптимизации мы взяли довольно популярный метод — SGD (стохастический градиентный спуск). На практике часто используется его улучшенная версия с инерцией (SGDm, m — momentum). Это позволяет более плавно поворачивать на резких изгибах и приближение становится лучше на глаз:

In [73]:

```
def baseline_model_sgdm():
    model = Sequential()
    model.add(Dense(100, input_dim=1, activation='relu'))
    model.add(Dense(1, input_dim=100, activation='linear'))

    sgd = SGD(lr=0.01, momentum=0.9, nesterov=True)
    model.compile(loss='mean_squared_error', optimizer=sgd)
    return model
```

In [76]:

```
from keras.models import Sequential
from keras.layers import Dense
from keras.optimizers import SGD

# тренируем сеть
model = baseline_model_sgdm()
model.fit(x, y, epochs=100, verbose = 0)

# отрисовываем результат приближения нейросетью поверх исходной функции
plt.scatter(x, y, color='black', antialiased=True)
plt.plot(x, model.predict(x), color='blue', linewidth=4, antialiased=True)

plt.show()
```



Усложняем дальше

Синус — довольно удачная функция для оптимизации. Главным образом потому, что у него нет широких плато — т.е. областей, где функция изменяется очень медленно. К тому же сама функция изменяется довольно равномерно. Чтобы проверить нашу конфигурацию на прочность, возьмём функцию посложнее:

In [110]:

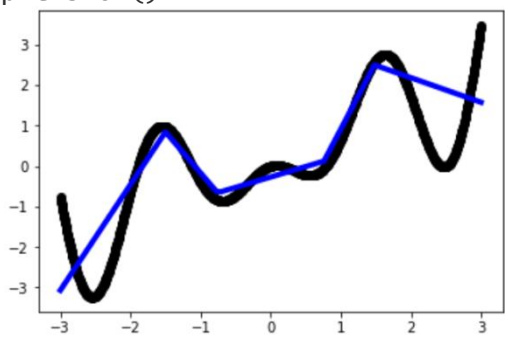
```
def f(x):
    if np.any(x) < 0:
        y = x * np.sin(x * 2 * np.pi)
    else:
        y = -x * np.sin(x * np.pi) + np.exp(x / 2) - np.exp(0)
    return y
```

In [111]:

```
y = f(x)
```

In [112]:

```
# тренируем сеть
model = baseline_model_sgdm()
model.fit(x, y, epochs=100, verbose = 0)

# отрисовываем результат приближения нейросетью поверх исходной функции
plt.scatter(x, y, color='black', antialiased=True)
plt.plot(x, model.predict(x), color='blue', linewidth=4, antialiased=True)

plt.show()
```



Увы и ах, здесь мы уже упираемся в потолок нашей архитектуры.

Дайте больше нелинейности!

Давайте попробуем заменить служивший нам в предыдущих примерах верой и правдой ReLU (выпрямитель) на более нелинейный гиперболический тангенс:

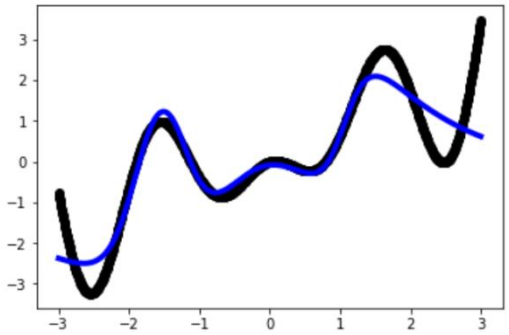
In [125]:

```
def baseline_model_sgdm_tanh():
    model = Sequential()
    model.add(Dense(20, input_dim=1, activation='tanh'))
    model.add(Dense(1, input_dim=20, activation='linear'))

    sgd = SGD(lr=0.01, momentum=0.9, nesterov=True)
    model.compile(loss='mean_squared_error', optimizer=sgd)
    return model
```

In [126]:

```
# тренируем сеть
model = baseline_model_sgdm_tanh()
model.fit(x, y, epochs=100, verbose = 0)
```

```
# отрисовываем результат приближения нейросетью поверх исходной функции
plt.scatter(x, y, color='black', antialiased=True)
plt.plot(x, model.predict(x), color='blue', linewidth=4, antialiased=True)

plt.show()
```



Инициализация весов — это важно!

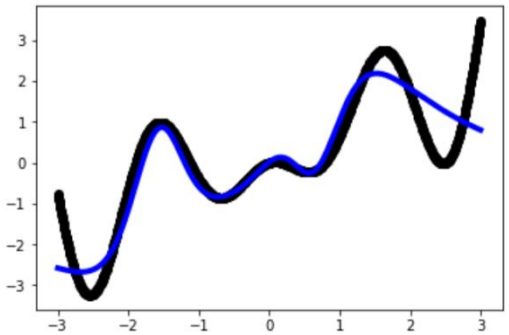
Приближение стало лучше на сгибах, но часть функции наша сеть не увидела. Давайте попробуем поиграться с ещё одним параметром — начальным распределением весов. Используем популярное на практике значение 'glorot_normal' (по имени исследователя Xavier Glorot, в некоторых фреймворках называется XAVIER):

In [135]:

```
def baseline_model_tanh_init_weghts():
    model = Sequential()
    model.add(Dense(20, input_dim=1, activation='tanh',
kernel_initializer="glorot_normal"))
    model.add(Dense(1, input_dim=20, activation='linear',
kernel_initializer="glorot_normal"))

    sgd = SGD(lr=0.01, momentum=0.9, nesterov=True)
    model.compile(loss='mean_squared_error', optimizer=sgd)
    return model
```

In [136]:

```
# тренируем сеть
model = baseline_model_tanh_init_weghts()
model.fit(x, y, epochs=100, verbose = 0)

# отрисовываем результат приближения нейросетью поверх исходной функции
plt.scatter(x, y, color='black', antialiased=True)
plt.plot(x, model.predict(x), color='blue', linewidth=4, antialiased=True)

plt.show()
```



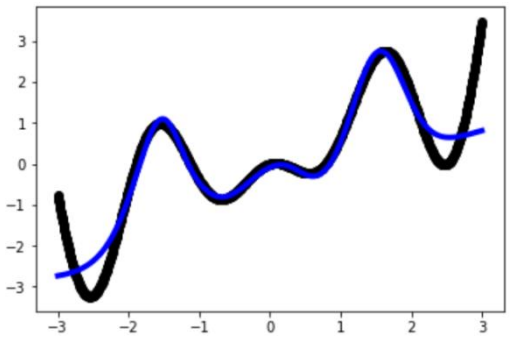
использование 'he_normal' (по имени исследователя Kaiming He) даёт ещё более приятный результат:

In [138]:

```
def baseline_model_tanh_init_weghts_he():
    model = Sequential()
    model.add(Dense(20, input_dim=1, activation='tanh', kernel_initializer=
'he_normal'))
    model.add(Dense(1, input_dim=20, activation='linear', kernel_initializer=
'he_normal'))

    sgd = SGD(lr=0.01, momentum=0.9, nesterov=True)
    model.compile(loss='mean_squared_error', optimizer=sgd)
    return model
```

In [153]:

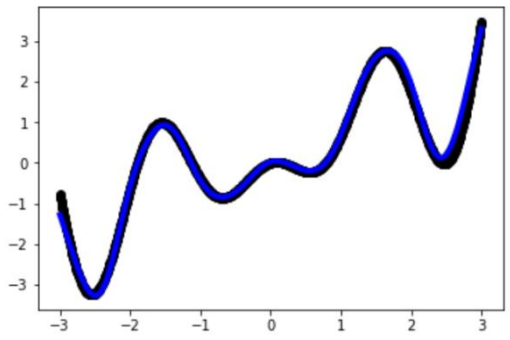
```
# тренируем сеть
model = baseline_model_tanh_init_weghts_he()
model.fit(x, y, epochs=100, verbose = 0)

# отрисовываем результат приближения нейросетью поверх исходной функции
plt.scatter(x, y, color='black', antialiased=True)
plt.plot(x, model.predict(x), color='blue', linewidth=4, antialiased=True)

plt.show()
```



увеличим число проходов до 400

In [148]:

```
# тренируем сеть
model = baseline_model_tanh_init_weghts_he()
model.fit(x, y, epochs=400, verbose = 0)

# отрисовываем результат приближения нейросетью поверх исходной функции
plt.scatter(x, y, color='black', antialiased=True)
plt.plot(x, model.predict(x), color='blue', linewidth=4, antialiased=True)

plt.show()
```



Как это работает?

Давайте сделаем небольшую паузу и разберёмся, каким образом работает наша текущая конфигурация. Сеть представляет из себя линейную комбинацию гиперболических тангенсов:

f(x) = w1' *tanh(w1* x + b1) +… + w5' *tanh(w5* x + b5) + b

Вернемся к варианту с 100 проходов:

In [178]:

```
# с помощью матрицы весов моделируем выход каждого отдельного нейрона перед суммацией
def tanh(x, i):
    w0 = model.layers[0].get_weights()
    w1 = model.layers[1].get_weights()

    return w1[0][i][0] * np.tanh(w0[0][0][i] * x + w0[1][i]) + w1[1][0]
```

In [179]:

```
# рисуем функцию и приближение
plt.scatter(x, y, color='black', antialiased=True)
plt.plot(x, model.predict(x), color='magenta', linewidth=2, antialiased=True)
```

Out[179]:

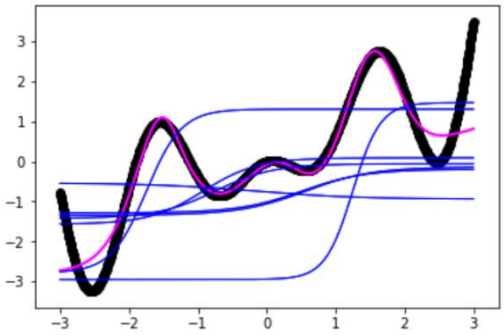
```
[<matplotlib.lines.Line2D at 0x1946d701b38>]
```

In [180]:

```
# рисуем разложение
for i in range(0, 7, 1):
    plt.plot(x, tanh(x, i), color='blue',
         linewidth=1)
```

In [176]:

```
plt.show()
```



На иллюстрации хорошо видно, что каждый гиперболический тангенс захватил небольшую зону ответственности и работает над приближением функции в своём небольшом диапазоне. За пределами своей области тангенс сваливается в ноль или единицу и просто даёт смещение по оси ординат.

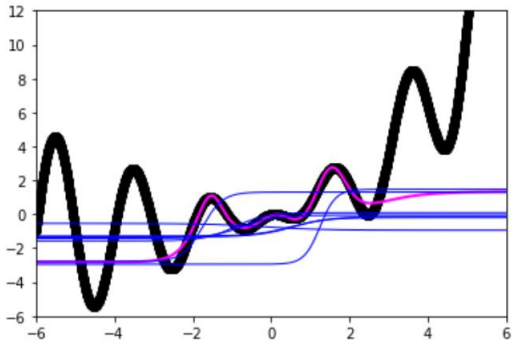
За границей области обучения

Давайте посмотрим, что происходит за границей области обучения сети, в нашем случае это [-3, 3]:

In [191]:

```
# накидываем тысячу точек от -3 до 3
x = np.linspace(-6, 6, 2000).reshape(-1, 1)
y = f(x)
plt.scatter(x, y, color='black', antialiased=True)
plt.plot(x, model.predict(x), color='magenta', linewidth=2, antialiased=True)
for i in range(0, 7, 1):
    plt.plot(x, tanh(x, i), color='blue',
```

```
        linewidth=1)
plt.xlim(-6,6)
plt.ylim(-6,12)
plt.show()
```



Как и было понятно из предыдущих примеров, за границами области обучения все гиперболические тангенсы превращаются в константы (строго говоря близкие к нулю или единице значения). Нейронная сеть не способна видеть за пределами области обучения: в зависимости от выбранных активаторов она будет очень грубо оценивать значение оптимизируемой функции. Об этом стоит помнить при конструировании признаков и входных данный для нейросети.

Идём в глубину

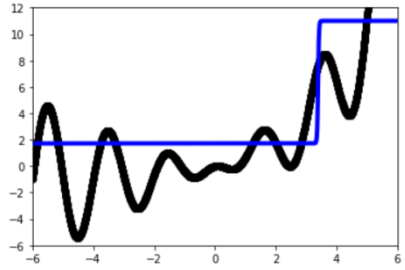
До сих пор наша конфигурация не являлась примером глубокой нейронной сети, т.к. в ней был всего один внутренний слой. Добавим ещё один:

In [235]:

```
def baseline_model_dl():
    model = Sequential()
    model.add(Dense(50, input_dim=1, activation='tanh', kernel_initializer= 'he_normal'))
    model.add(Dense(50, input_dim=50, activation='tanh', kernel_initializer= 'he_normal'))
    model.add(Dense(1, input_dim=50, activation='linear', kernel_initializer= 'he_normal'))
    sgd = SGD(lr=0.001, momentum=0.9, nesterov=True)
    model.compile(loss='mean_squared_error', optimizer=sgd)
    return model
```

In [238]:

```
model = baseline_model_dl()
model.fit(x, y, epochs=100, verbose = 0)

# отрисовываем результат приближения нейросетью поверх исходной функции
plt.scatter(x, y, color='black', antialiased=True)
plt.plot(x, model.predict(x), color='blue', linewidth=4, antialiased=True)
plt.xlim(-6,6)
plt.ylim(-6,12)
plt.show()
```



N.B. Слепое добавление слоёв не даёт автоматического улучшения, что называется из коробки. Для большинства практических применений двух внутренних слоёв вполне достаточно, при этом вам не придётся разбираться со спецэффектами слишком глубоких сетей, как например проблема исчезающего градиента. Если вы всё-таки решили идти в глубину, будьте готовы много экспериментировать с обучением сети.