OOP,_Iterables,_Linear_Regression,_Stochastic_Gradient_Descent_moc

August 7, 2019

[1]:
```
%matplotlib inline
from fastai.basics import *
```

In this part of the lecture we explain Stochastic Gradient Descent (SGD) which is an **optimization** method commonly used in neural networks. We will illustrate the concepts with concrete examples.

1 Linear Regression with Stochastic Gradient Descent

1.1 Problem Setup

The goal of linear regression is to fit a line to a set of points.

Let's generate univariate data from Uniform Distribution between -30 and 30.

[2]:
```
data_size = 100
x = torch.ones(data_size,2)
x[:,0].uniform_(-30.,30)
x[:5]
```

[2]:
```
tensor([[-15.0058,   1.0000],
        [ 22.1868,   1.0000],
        [ 22.6643,   1.0000],
        [ -6.4448,   1.0000],
        [ 11.2798,   1.0000]])
```

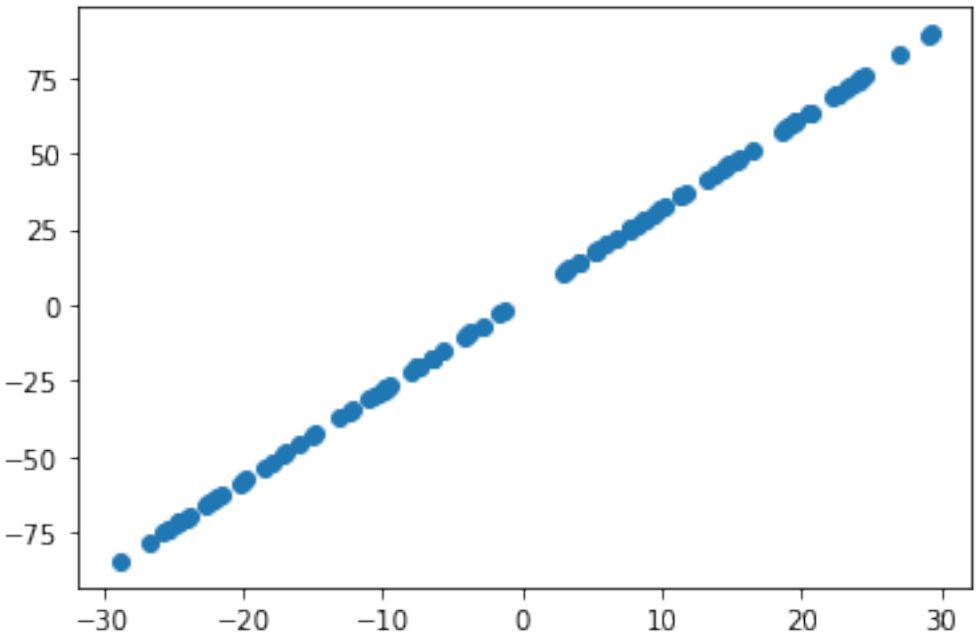
[3]:
```
a = tensor(3.,2); print(a.shape)
print(a)
```

```
torch.Size([2])
tensor([3., 2.])
```

1.1.1 How many parameters does this linear regression model have?

[4]:
```
y_generator = lambda x,a,data_size,alpha : x@a + alpha*torch.rand(data_size)
#y = x@a + torch.rand(n)
y = y_generator(x,a,data_size,0)
```

[5]:
```
plt.scatter(x[:,0], y);
```



You want to find **parameters** (weights) `a` such that you minimize the *error* between the points and the line `x@a`. Note that here `a` is unknown. For a regression problem the most common *error function* or *loss function* is the **mean squared error**.

[6]:
```
def mse(y_hat, y): return ((y_hat-y)**2).mean()
```

Suppose we believe `a = (3.0,2.0)` then we can compute `y_hat` which is our *prediction* and then compute our error.

[7]:
```
a = tensor(3.,2)
```

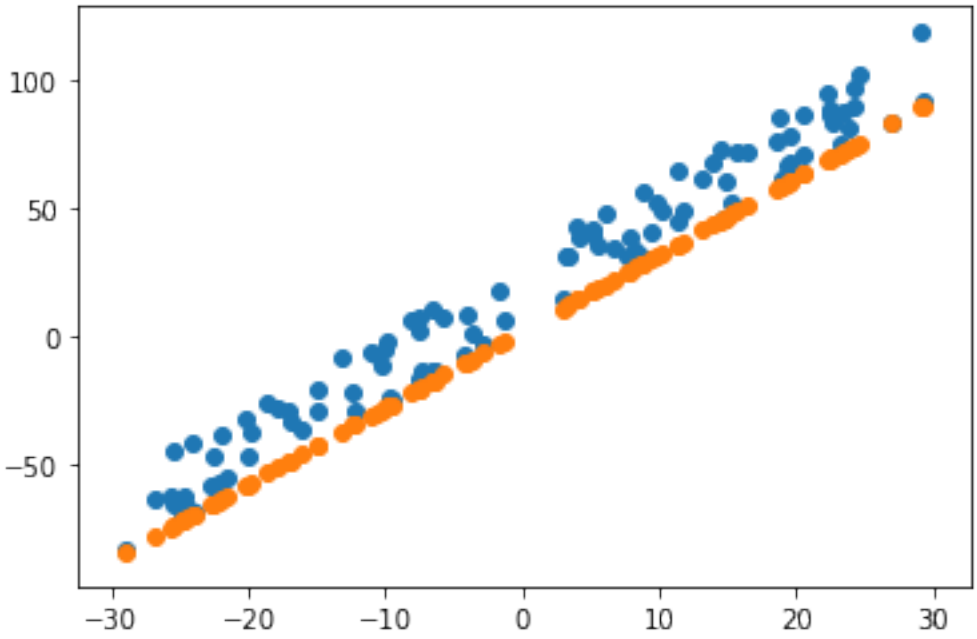
[8]:
```
simple_linear_regression = lambda x,a : x@a
y_hat = simple_linear_regression(x,a)
#y_hat = x@a
print(mse(y_hat, y))
```

```
tensor(0.)
```

Or maybe we wanted to generate some data that has random noise on a higher scale, with very high variance (noise variance)

[9]:
```
y = y_generator(x,a,data_size,30)
```

[10]:
```
plt.scatter(x[:,0],y)
plt.scatter(x[:,0],y_hat);
```



So far we have specified the *model* (linear regression) and the *evaluation criteria* (or *loss function*). Now we need to handle *optimization*; that is, how do we find the best values for `a`? How do we find the best *fitting* linear regression.

2 From Gradient to Stochastic Gradient Descent

We would like to find the values of `a` that minimize `mse_loss`.

Gradient descent is an algorithm that minimizes functions. Given a function defined by a set of parameters, gradient descent starts with an initial set of parameter values and iteratively moves toward a set of parameter values that minimize the function. This iterative minimization is achieved by taking steps in the negative direction of the function gradient.

Here is gradient descent implemented in PyTorch.

2.0.1 Putting Everything TOgether : Training SImple Linear Regression with List Comprehension

[11]:
```
import numpy
batch_size = 5

a = nn.Parameter(tensor(-1.,1))
losses = []

data_size=100
```

```
x = torch.ones(data_size,2)
x[:,0].uniform_(-30.,30)


y_generator = lambda x,a,data_size,alpha : x@a + alpha*torch.rand(data_size)
#y = x@a + torch.rand(n)
y = y_generator(x,a,data_size,5)

def mse(y_hat, y): return ((y_hat-y)**2).mean()
print(f"Data size is {data_size} and batch size is {batch_size}")

parameters = a

def update(parameters,batch_size,data_size,t):
    indices = numpy.random.randint(1, data_size, size=(batch_size,1)).flatten()
    y_hat = x[indices,:]@parameters
    loss = mse(y[indices], y_hat)
    if t % 10 == 0: print(loss)
    loss.backward(retain_graph=True)
    with torch.no_grad():
        parameters.sub_(lr * parameters.grad)
        parameters.grad.zero_()
    losses.append(loss)

lr = 1e-2
iterations_per_epoch = data_size//batch_size
epochs = 10
total_iterations = iterations_per_epoch * epochs
print(f"Total iterations is {total_iterations}")

#### USING LIST COMPREHENSION ####
[update(parameters,batch_size,data_size,t) for t in range(total_iterations)]

amnt_losses_recorded = total_iterations//10
print(f"Since every 10th iteration result is printed, in total we have␣
 ↪{amnt_losses_recorded} print statements")
```

```
Data size is 100 and batch size is 5
Total iterations is 200
tensor(14.4847, grad_fn=<MeanBackward0>)
tensor(19.4902, grad_fn=<MeanBackward0>)
tensor(10.9414, grad_fn=<MeanBackward0>)
tensor(16.0015, grad_fn=<MeanBackward0>)
tensor(8.2983, grad_fn=<MeanBackward0>)
tensor(2.2003, grad_fn=<MeanBackward0>)
tensor(8.9515, grad_fn=<MeanBackward0>)
tensor(5.9760, grad_fn=<MeanBackward0>)
```

```
tensor(4.8042, grad_fn=<MeanBackward0>)
tensor(16.1493, grad_fn=<MeanBackward0>)
tensor(5.8704, grad_fn=<MeanBackward0>)
tensor(11.4631, grad_fn=<MeanBackward0>)
tensor(5.6994, grad_fn=<MeanBackward0>)
tensor(3.9245, grad_fn=<MeanBackward0>)
tensor(7.8642, grad_fn=<MeanBackward0>)
tensor(8.5031, grad_fn=<MeanBackward0>)
tensor(9.1187, grad_fn=<MeanBackward0>)
tensor(10.8560, grad_fn=<MeanBackward0>)
tensor(10.7461, grad_fn=<MeanBackward0>)
tensor(7.4082, grad_fn=<MeanBackward0>)
Since every 10th iteration result is printed, in total we have 20 print
statements
```

[12]:
```
print(f"The losses look like this {losses[0:10]}")
```

```
The losses look like this [tensor(14.4847, grad_fn=<MeanBackward0>),
tensor(9.1633, grad_fn=<MeanBackward0>), tensor(9.1369,
grad_fn=<MeanBackward0>), tensor(9.3830, grad_fn=<MeanBackward0>),
tensor(12.7124, grad_fn=<MeanBackward0>), tensor(9.2520,
grad_fn=<MeanBackward0>), tensor(10.0163, grad_fn=<MeanBackward0>),
tensor(8.1113, grad_fn=<MeanBackward0>), tensor(9.5191,
grad_fn=<MeanBackward0>), tensor(12.2677, grad_fn=<MeanBackward0>)]
```

2.1 Interlude : Using Iterators to Improve Code Readability. Connection Between Functional and Object Oriented Programming

Read And this

[13]:
```
from itertools import count
class square_all:
    def __init__(self, numbers):
        self.numbers = iter(numbers)
    def __next__(self):
        return next(self.numbers) ** 2
    def __iter__(self):
        return self

numbers = count(5)
print(f"The numbers are {type(numbers)}")
squares = square_all(numbers)
print(f"The type of squares is are {type(squares)}")

#print(next(squares))
#print(next(squares))
```

```
The numbers are <class 'itertools.count'>
The type of squares is are <class '__main__.square_all'>
```

This iterator class works, but we don't usually make iterators this way. Usually when we want to make a custom iterator, we make a generator function:

[14]:
```
# This generator function is equivalent to the class we made above, and it␣
 ↪works essentially the same way.
def square_all(numbers):
    for n in numbers:
        yield n**2

# OR

def square_all(numbers):
    return (n**2 for n in numbers)
```

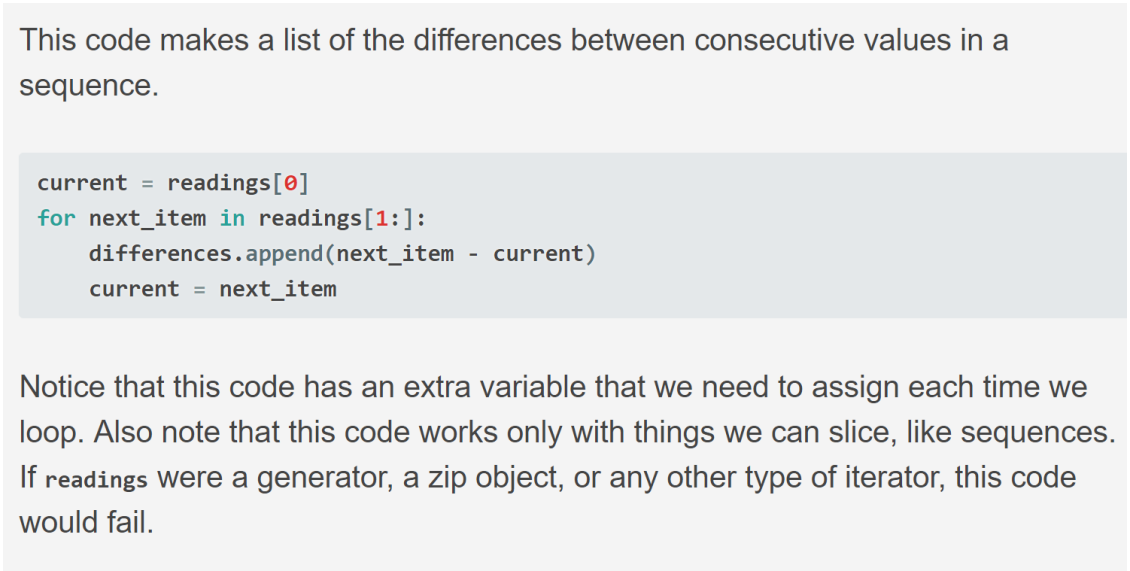
That yield statement probably seems magical, but it is very powerful:

`yield` allows us to put our generator function on pause between calls from the next function. The yield statement is the thing that separates generator functions from regular functions.

2.1.1 Mixing Generators and Iterables into the Learning Process

[15]:
```
from IPython.display import Image

Image("https://filedn.com/lK1VhM9GbBxVlERr9KFjD4B/img/initial_version.PNG")
```

[15]:



[16]:
```
Image("https://filedn.com/lK1VhM9GbBxVlERr9KFjD4B/img/improve_code_2.PNG")
```

[16]:

```
def with_next(iterable):
    """Yield (current, next_item) tuples for each item in iterable."""
    iterator = iter(iterable)
    current = next(iterator)
    for next_item in iterator:
        yield current, next_item
        current = next_item
```

We're manually getting an iterator from our iterable, calling `next` on it to grab the first item, then looping over our iterator to get all subsequent items, keeping track of our last item along the way. This function works not just with sequences, but with any type of iterable.

This is the same code as before, but we're using our helper function instead of manually keeping track of `next_item`:

```
differences = []
for current, next_item in with_next(readings):
    differences.append(next_item - current)
```

Notice that this code doesn't have awkward assignments to `next_item` hanging around our loop. The `with_next` generator function handles the work of keeping track of `next_item` for us.

Also note that this code has been compacted enough that we could even copy-paste our way into a list comprehension if we wanted to.

```
differences = [
    (next_item - current)
    for current, next_item in with_next(readings)
]
```

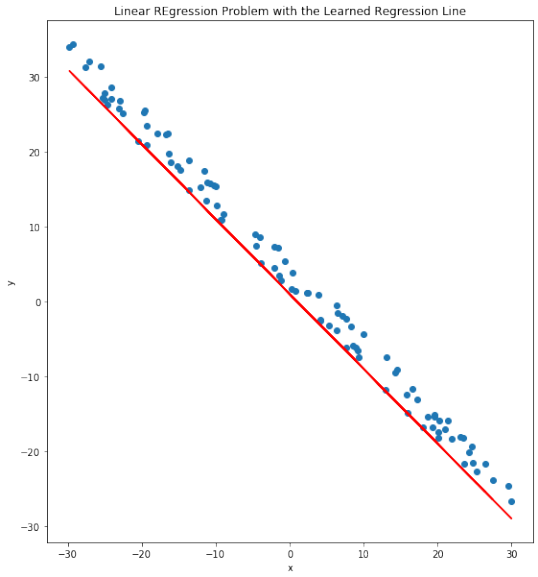
2.2 Using Generators to Find The Differences in Errors (Gradient of Errors)

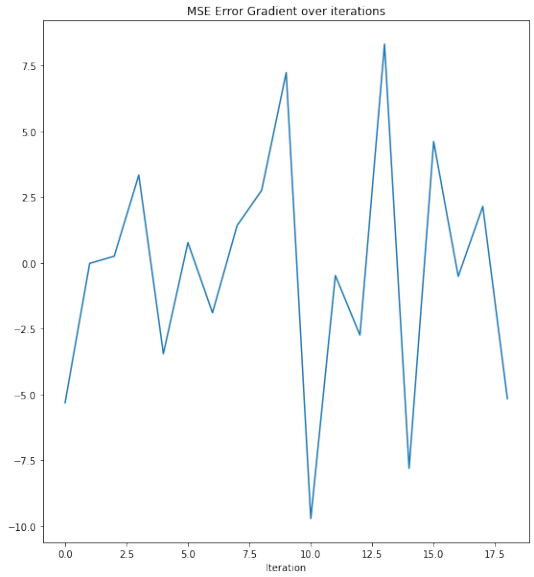
Say you want to expand your skills and not use Tensorboard to visualize the training procedure. Minitask :

- **Write a generator function that yields 2 subsequent values over the iterable at a time.**
- **Then use this function to get the gradient of the loss errors and visualize these errors using Matplotlib**

[17]:
```
import matplotlib.pyplot as plt

def testing_subtraction(tensor1,tensor2):
    try:
        diff = tensor1.data - tensor2.data
        #print(f"The difference of {tensor1.data} and {tensor2.data} is␣
 ↪{diff}")
    except:
        diff = 999
        print("Subtraction failed")

    return diff
def with_next(iterable):
    """Yield (current, next_item) tuples for each item in iterable."""
    iterator = iter(iterable)
    current = next(iterator)
    for next_item in iterator:
        yield current, next_item
        current = next_item

differences = []
for current, next_item in with_next(range(amnt_losses_recorded)):
    differences.append(testing_subtraction(losses[next_item],losses[current]))



fig, (ax1, ax2) = plt.subplots(1,2,figsize=(20, 10))
ax2.plot(differences)
ax2.set_title("MSE Error Gradient over iterations")
ax2.set_xlabel('Iteration')
ax1.scatter(x[:,0],y)
ax1.plot(x[:,0],x@a,'r');
ax1.set_xlabel('x')
ax1.set_ylabel('y')
ax1.set_title('Linear REgression Problem with the Learned Regression Line')
```

[17]:
```
Text(0.5, 1.0, 'Linear REgression Problem with the Learned Regression Line')
```





2.2.1 What can you say about the convergence of Stochastic Gradient Descent when looking at this plot?

2.3 Animate it!

Run the animation on the FastAI Tutorial

In practice, we don't calculate on the whole file at once, but we use *mini-batches*.

2.4 Vocabulary

[ ]:

- Learning rate
- Epoch
- Minibatch
- SGD
- Model / Architecture
- Parameters
- Loss function

For classification problems, we use *cross entropy loss*, also known as *negative log likelihood loss*. This penalizes incorrect confident predictions, and correct unconfident predictions.

[ ]:

[ ]:

[ ]:

[ ]:

[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:

[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:

[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:

[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:

[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:

[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:

[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:

[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:

[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:

[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:

[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:

[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:

[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:

[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:

[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:

[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:

[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:

[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:

[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:

[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:
[ ]:

[ ]:

[ ]:

[ ]:

[ ]:

[ ]:

[ ]:

[ ]:

[ ]:

[ ]:

[ ]:

[ ]:

[ ]: