In [1]:

```python
import jax
import flax.linen as nn
import tensorflow as tf
import tensorflow_datasets as tfds
import matplotlib.pyplot as plt
import numpy as np
import jax.numpy as jnp
from functools import partial
from typing import Sequence
import random
import time
```

Rapid intro to supervised learning with neural nets II: using JAX

This notebook gives a rapid introduction to supervised learning with neural networks. The example is based on Chapter 1 of Nielsen's online book "Neural Networks and Deep Learning" and it guides you to set up the neural network training using the JAX and Flax libraries.

For further reading I recommend also the review article "A high-bias, low-variance introduction to Machine Learning for physicists".

A few words on JAX

JAX is a Python library that provides useful functionality for machine learning applications (especially deep learning), namely automatic differentiation, just-in-time compilation, and vectorization. This is implemented in JAX through function transformations, i.e., functions that map functions to new functions.

Automatic differentiation

With automatic differentiation we can let the computer compute gradients of arbitrary functions. In JAX the function `jax.grad()` takes a function as argument and returns a function that is the gradient of the given function. Example:

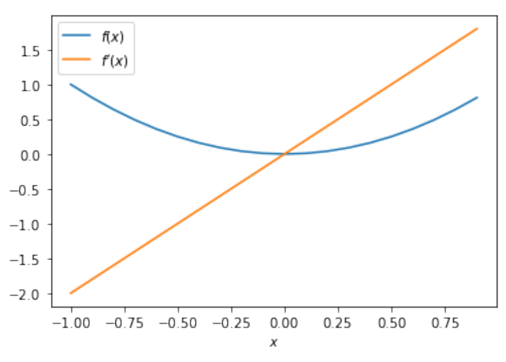
In [2]:

```python
# Define a function
def f(x):
    return x**2

# Get the gradient
f_prime = jax.grad(f)

# Evaluate function and gradient
x=np.arange(-1,1,.1)
y=np.array([f_prime(r) for r in x])
plt.plot(x,f(x),label=r"$f(x)$")
plt.plot(x,y,label=r"$f'(x)$")
plt.xlabel(r"$x$")
plt.legend()
```

```
WARNING:absl:No GPU/TPU found, falling back to CPU. (Set TF_CPP_MIN_LOG_LEVEL=0 and rerun for more info.)
```

Out[2]:

```
<matplotlib.legend.Legend at 0x7fc73f715150>
```



Vectorization

The `jax.vmap()` function allows us to apply a function that is defined for a single input to an array. For example, we can replace the line

```python
y=np.array([f_prime(r) for r in x])
```

from the previous cell as follows:

In [3]:

```python
# Get a vectorized version of the function
f_prime_vectorized = jax.vmap(f_prime)
# Apply the vectorized function to an input array
f_prime_vectorized(x)
```

Out[3]:

```
DeviceArray([-2.000000e+00, -1.800000e+00, -1.600000e+00, -1.400000e+00,
             -1.200000e+00, -1.000000e+00, -8.000000e-01, -6.000000e-01,
             -4.000000e-01, -2.000000e-01, -4.440892e-16,  2.000000e-01,
              4.000000e-01,  6.000000e-01,  8.000000e-01,  1.000000e+00,
              1.200000e+00,  1.400000e+00,  1.600000e+00,  1.800000e+00],            dtype=float32)
```

Just-in-time compilation

With `jax.jit()` we can ask JAX to just-in-time (JIT) compile our Python code such that it can be executed with the high efficiency of XLA.

A simple, yet not particularly spectacular, example is

In [4]:

```python
# Get a vectorized version of the function
f_prime_vectorized_compiled = jax.jit(f_prime_vectorized)
# Apply the vectorized function to an input array
f_prime_vectorized_compiled(x)
```

Out[4]:

```
DeviceArray([-2.000000e+00, -1.800000e+00, -1.600000e+00, -1.400000e+00,
             -1.200000e+00, -1.000000e+00, -8.000000e-01, -6.000000e-01,
             -4.000000e-01, -2.000000e-01, -4.440892e-16,  2.000000e-01,
              4.000000e-01,  6.000000e-01,  8.000000e-01,  1.000000e+00,
              1.200000e+00,  1.400000e+00,  1.600000e+00,  1.800000e+00],            dtype=float32)
```

Superficially, `f_prime_vectorized_compiled()` does not differ from `f_prime_vectorized()`. Under the hood, however, the compiled function executes its task (potentially) much more efficiently than the original one. In the code below you will find a number of `jax.jit` statements. By removing these statements and working only with the un-compiled versions of the respective functions you will observe a noticeable slow-down.

Notice: In order to use JAX function transformations (like vectorization) we have to replace the Numpy library with its JAX version, which has the same interface. For example, instead of `np.array` all our arrays will be `jnp.array` s. The same for functions, e.g. instead of `np.dot` we have to use `jnp.dot` for the dot-product.

The MNIST hand-written digits data set

Let's first get a simple exemplary data set - the MNIST hand-written digits. The following cell downloads both the test and training parts of the data set.

In [5]:

```python
trainData = jnp.array(
                tfds.as_numpy(
                    tfds.load('mnist', split='train', batch_size=-1, shuffle_files=False)
                )['image'].reshape(-1,28,28)
            )
trainLabels = jnp.array(
                    tfds.as_numpy(
                        tfds.load('mnist', split='train', batch_size=-1, shuffle_files=False)
                    )['label']
                )

testData = jnp.array(
                tfds.as_numpy(
                    tfds.load('mnist', split='test', batch_size=-1, shuffle_files=False)
                )['image'].reshape(-1,28,28)
            )
testLabels = jnp.array(
                tfds.as_numpy(
                    tfds.load('mnist', split='test', batch_size=-1, shuffle_files=False)
                )['label']
            )
```

`trainData` is now a `jax.numpy.array` of shape `(60000,28,28)`, meaning that we have 60k images of 28×28 pixels (grayscale), each showing one hand-written digit. `trainLabels` holds the corresponding *labels*, i.e. an integer for each image, stating which digit it shows.

Defining a neural network model using Flax

Flax is a library built on top of JAX, which allows you to easily compose complicated deep learning models. If you are familiar with Pytorch, the following syntax will be very intuitive for you.

In Flax a new model can be defined as a class that inherits from the `nn.Module` base class. Here, we introduce Flax's abbreviated model definition; notice that general model definitions can be more involved. In the short form, a model is defined by defining a `__call__` method that evaluates the network on the given input. The library provides implementations of typical linear transformations as well as typical activation functions (among other typical building blocks of neural networks).

In the cell below we use the provided `Dense` linear transformation and the `sigmoid` activation function to implement the same network architecture as the one that we coded from scratch in part I of this tutorial:

In [6]:

```python
class MyNet(nn.Module):
    layers: Sequence[int] # A tuple that contains the widths of all layers follwing the input layer

    @nn.compact
    def __call__(self, x):

        a = x.ravel() # flatten the input

        # Evaluate network layer by layer
        for width in self.layers:
            # Apply a the Dense layer with given width followed by the non-linearity
            a = nn.sigmoid(nn.Dense(width)(a))

        # Return activations of the output layer
        return a
```

Now we can again what the network thinks about our images of digits. For this purpose we define `initialize_network` and `neural_network` analogous to part I of the tutorial, but this time based on our `MyNet` class.

In [7]:

```python
def initialize_network(layers, seed=123):
    # Get random initial parameters. Notice: The `init` method needs an example input for this purpose.
    return MyNet(layers=layers).init(jax.random.PRNGKey(seed), trainData[0])


def neural_network(params, image, layers):
    # Evaluate the network with given parameters
    neural_network = MyNet(layers=layers)

    return jax.jit(jax.vmap(lambda x: neural_network.apply(params, x)))(image)

# Define the network size.
# Here we only need to include the width of layers *after* the input layer.
# The size of the input layer is determined automatically from the input data.
net_layers=(100,10)

# Get initial parameters
params = initialize_network(net_layers)

# Evaluate the network
neural_network(params, trainData[:3], net_layers)
```

Out[7]:

```
DeviceArray([[0.6283565 , 0.50599694, 0.6832025 , 0.50604314, 0.42828634,
              0.25097242, 0.6408896 , 0.5169621 , 0.38393703, 0.39720345],
             [0.6706722 , 0.57142377, 0.6822245 , 0.7108034 , 0.30151436,
              0.48183188, 0.35933733, 0.6973047 , 0.4983423 , 0.38091958],
             [0.48349693, 0.5223302 , 0.6942643 , 0.55765235, 0.46390426,
              0.4895871 , 0.51695365, 0.6439555 , 0.60555077, 0.6348824 ]],            dtype=float32)
```

Next, we need a cost function:

In [8]:

```python
@partial(jax.jit, static_argnums=3)
def cost_function(params, images, labels, layers):
    '''This function evaluates the cost function for given predictions and labels
    Args:
    * `params`: Network parameters.
    * `images`: A batch of input images.
    * `labels`: Correct labels for the given images.
    * `layers`: Size of the network (list of widths).
    Returns: Cost associated with the neural network predictions for the given data.
    '''

    labels = jax.nn.one_hot(labels, 10) # get one-hot encoding of labels
    predictions = neural_network(params,images,layers)
    cost = jnp.sum((predictions-labels)**2)

    return cost / labels.shape[0]
```

With this, we can check the performance of our randomly initialized network in classifying some of our images:

In [9]:

```python
batch = trainData[:128]    # select a batch of images
labels = trainLabels[:128] # and corresponding labels

cost_function(params,batch,labels,net_layers)
```

Out[9]:

```
DeviceArray(2.7373376, dtype=float32)
```

Now, what is missing is a function to compute the gradients of the cost function. This is easily solved using `jax.grad` for automatic differentiation:

In [10]:

```python
cost_function_gradients = jax.grad(cost_function)
```

Finally, we are ready to train the network:

In [11]:

```python
def evaluate_predictions(predictions, labels):
    '''This is a helper function that counts how many of the given predictions match the labels.
    Args:
    * `predictions`: Predictions from neural network (=activations on output layer)
    * `labels`: correct labels
    Returns: Number of correct predictions, i.e., number of cases, in which the index of the maximal
    activation matches the given label.
    '''

    pred_labels = jnp.argmax(predictions, axis=1)

    return jnp.where(pred_labels==labels)[0].shape[0]


# Get a key for the PRNG
prng_key = jax.random.PRNGKey(123)

# Here we define the hyperparamters
num_epochs = 10 # Number of epochs to loop over
learning_rate = 0.001 # Learning rate
batch_size = 128 # Size of mini-batches

# Compute the number of mini-batches that matches the chosen mini-batch size
batch_number = trainData.shape[0] // batch_size

# Evaluate network and assess performance
predictions = neural_network(params, testData, net_layers)
current_cost = cost_function(params, testData, testLabels, net_layers)
correct_predictions = evaluate_predictions(predictions, testLabels)
print("  Initial cost: %f" % (current_cost))
print("  Correctly predicted labels: %d / %d" % (correct_predictions, len(testLabels)))

# Training loop over epochs
for n in range(num_epochs):
    tic = time.perf_counter()

    print("* Episode %d" % (n))

    # Generate batches from randomly permuted data
    prng_key, tmp_key = jax.random.split(prng_key) # jax-style treatment of random numbers
    batches = (jax.random.permutation(tmp_key, trainData)[:batch_number*batch_size].reshape(-1,128,28,28),
               jax.random.permutation(tmp_key, trainLabels)[:batch_number*batch_size].reshape(-1,128))

    # Loop over mini-batches
    for samples, labels in zip(*batches):

        # compute gradients
        grads = jax.jit(cost_function_gradients, static_argnums=3)(params, samples, labels, net_layers)

        # Perform SGD parameter update step
        params = jax.tree_util.tree_multimap(lambda a,b: a-learning_rate*b, params, grads)

    # Evaluate network and assess performance
    predictions = neural_network(params, testData, net_layers)
    current_cost = cost_function(params, testData, testLabels, net_layers)
    correct_predictions = evaluate_predictions(predictions, testLabels)
    print("  Current cost: %f" % (current_cost))
    print("  Correctly predicted labels: %d / %d" % (correct_predictions, len(testLabels)))
    print("  -- Time for episode: %fs" % (time.perf_counter()-tic))
```

```
  Initial cost: 2.685170
  Correctly predicted labels: 781 / 10000
* Episode 0
  Current cost: 0.876519
  Correctly predicted labels: 3529 / 10000
  -- Time for episode: 2.247521s
* Episode 1
  Current cost: 0.805401
  Correctly predicted labels: 4743 / 10000
  -- Time for episode: 1.407948s
* Episode 2
  Current cost: 0.758254
  Correctly predicted labels: 5466 / 10000
  -- Time for episode: 1.470804s
* Episode 3
  Current cost: 0.717020
  Correctly predicted labels: 6019 / 10000
  -- Time for episode: 1.412190s
* Episode 4
  Current cost: 0.681147
  Correctly predicted labels: 6409 / 10000
  -- Time for episode: 1.379596s
* Episode 5
  Current cost: 0.647137
  Correctly predicted labels: 6812 / 10000
  -- Time for episode: 1.375014s
* Episode 6
  Current cost: 0.615984
  Correctly predicted labels: 7088 / 10000
  -- Time for episode: 1.481921s
* Episode 7
  Current cost: 0.588617
  Correctly predicted labels: 7338 / 10000
  -- Time for episode: 1.430292s
* Episode 8
  Current cost: 0.563799
  Correctly predicted labels: 7523 / 10000
  -- Time for episode: 1.499562s
* Episode 9
  Current cost: 0.541183
  Correctly predicted labels: 7650 / 10000
  -- Time for episode: 1.543345s
```