In [ ]:
```
from ml import plots
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
import numpy as np
from matplotlib.colors import ListedColormap
from tqdm.auto import tqdm
```

In [ ]:
```
%matplotlib inline

plots.set_plot_style()
```

Neural Networks and Deep Learning

Most of this material was heavily inspired by http://cs231n.github.io/. Stanfords CNN for Image Recognition lecture.

Linear Classification II

Previously we talked about classification using a linear function $f(x) : \mathbb{R}^p \to \mathbb{R}$ and least-square regression.

Recall the linear function dependent on the parameter vector β

$$f(x) = \hat{y} = \hat{\beta}_0 + \sum_{j=1}^{p} x_j \hat{\beta}_j$$

$$f(x) = \hat{y} = x^T \beta$$

where $\beta = (\beta_0, \beta_1, \beta_2, \ldots, \beta_p)$ and $x = (1, x_1, x_2, \ldots, x_p)$.

We minimized the residual sum of squares using matrix multiplication.

$$RSS(\beta) = (\mathbf{y} - \mathbf{X}\beta)^T(\mathbf{y} - \mathbf{X}\beta) \Rightarrow \hat{\beta} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$$

In order to perform the classification we had to define the classification function

$$\hat{y} = \begin{cases} \text{Yes}, & \text{if } f(x) > 0.5 \\ \text{No}, & \text{if } f(x) \le 0.5 \end{cases}$$

Now we encode the class label directly into the function. Given a classification problem with K different classes we want $f(x) : \mathbb{R}^p \to \mathbb{R}^K$ to return a vector of length K. Instead of finding the parameters in a vector β we use a $p \times K$ matrix $\mathbf{W}$

$$f(x) = \hat{y} = x \cdot \mathbf{W}$$

At this point we could again minimize a modified version of the RSS or the SVM loss. This time however we want to be able to have a more *probabilistic* interpretation of the loss function.

We use a new loss function called the **cross-entropy loss**. Also known as logistic loss. Given two probability density functions p and q over the same probabilistic space, cross entropy is defined as

$$H_{\times}(p, q) = -\sum_x p(x) \, \log q(x).$$

The cross entropy gives smaller values for similar distributions q and p. In case of classification problems p is the true distribution of classes for a given x and $q(x)$ the estimated distribution as produced by our machine learner.

In the example below we look at p for the digits dataset. As one example in the dataset can be just one digit, the p vector has only a single non-zero entry.

If a classifier produces the same distribution for the dataset we have perfect classification.

In [ ]:
```
from sklearn.datasets import load_digits
from sklearn import preprocessing

X, y = load_digits(return_X_y=True)

y = preprocessing.LabelBinarizer().fit_transform(y)

print('True (discrete) probability distributions.')


fig, axs = plt.subplots(2, 5, figsize=(8, 4))
for ax, x_i, y_i in zip(axs.flat, X[10:20], y):
    img = x_i.reshape(8, 8)
    ax.imshow(img, cmap='gray_r', interpolation='nearest')
    ax.axis('off')

    print(y_i)
```

```
True (discrete) probability distributions.
[1 0 0 0 0 0 0 0 0 0]
[0 1 0 0 0 0 0 0 0 0]
[0 0 1 0 0 0 0 0 0 0]
[0 0 0 1 0 0 0 0 0 0]
[0 0 0 0 1 0 0 0 0 0]
[0 0 0 0 0 1 0 0 0 0]
[0 0 0 0 0 0 1 0 0 0]
[0 0 0 0 0 0 0 1 0 0]
[0 0 0 0 0 0 0 0 1 0]
[0 0 0 0 0 0 0 0 0 1]
```

We want our estimated probabilities for the distribitution of class labels to be as close to the true distribution as possible. The result of our function $f(x)$ does not produce probability estimates. We need to *transform the results to probabilities* using the so called **softmax** function

$$q_k(x) = \frac{e^{f_k(x)}}{\sum_j e^{f_j(x)}}$$

where the index $j \in \{1, \ldots, K\}$ loops over all classes and f_k is the prediction for class k.

In shorter words: A classifier using cross entropy loss with softmax produces probabilities for class memberships.

Below we implement the softmax function and the loss function which uses a weight matrix $\mathbf{W}$ to compute the loss of the entire dataset

In [ ]:
```
from sklearn.preprocessing import label_binarize as label_binarize_sklearn

def label_binarize(labels, n_classes):
    # sklearn does not expand binary labels but we need this for our
    # formulation of the training
    binary_labels = label_binarize_sklearn(labels, classes=range(max(n_classes, 3)))
    return binary_labels[:, :n_classes]
```

In [ ]:
```
label_binarize([0, 1, 1, 0], 2)
```

Out[ ]:
```
array([[1, 0],
       [0, 1],
       [0, 1],
       [1, 0]])
```

In [ ]:
```
from sklearn.datasets import load_digits

rng = np.random.default_rng(0)

X, y = load_digits(return_X_y=True)

ones = np.ones(shape=(len(X), 1))
X = np.hstack([ones, X])

# we have 10 classes and p attributes, digits from 0 to 9 and 64 pixels
K = 10
p = 64

# include the bias into the weight matrix
W = rng.normal(size=(p+1, K))

y = label_binarize(y, 10)
y
```

Out[ ]:
```
array([[1, 0, 0, ..., 0, 0, 0],
       [0, 1, 0, ..., 0, 0, 0],
       [0, 0, 1, ..., 0, 0, 0],
       ...,
       [0, 0, 0, ..., 0, 1, 0],
       [0, 0, 0, ..., 0, 0, 1],
       [0, 0, 0, ..., 0, 1, 0]])
```

In [ ]:
```
# softmax function shifted to zero so no numerical instabilities arise.
def softmax(f):
    f_shifted = f - f.max()
    p = np.exp(f_shifted).T / np.sum(np.exp(f_shifted), axis=1)
    return p.T
```

```
# loss for the entire training set
def loss_cross_ent(X, y, W):
    f = X @ W
    q = softmax(f)
    l = -np.sum(y * np.log2(q), axis=1)
    return l.mean() # the mean over all samples in the batch/dataset

loss_cross_ent(X, y, W)
```

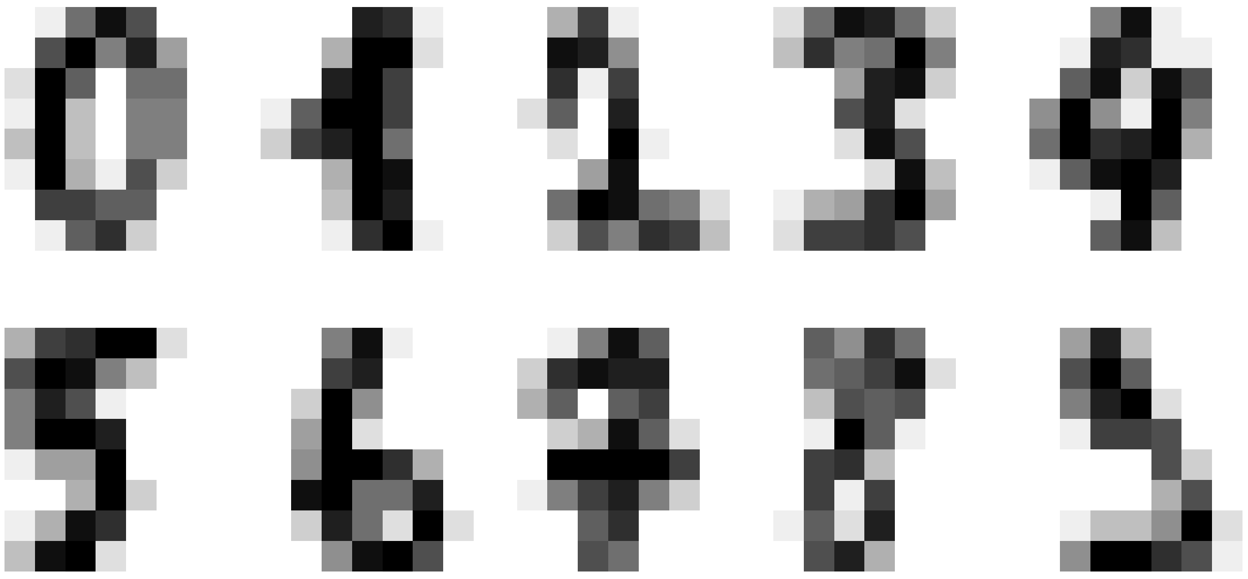
Out[ ]:
```
167.32104212686957
```

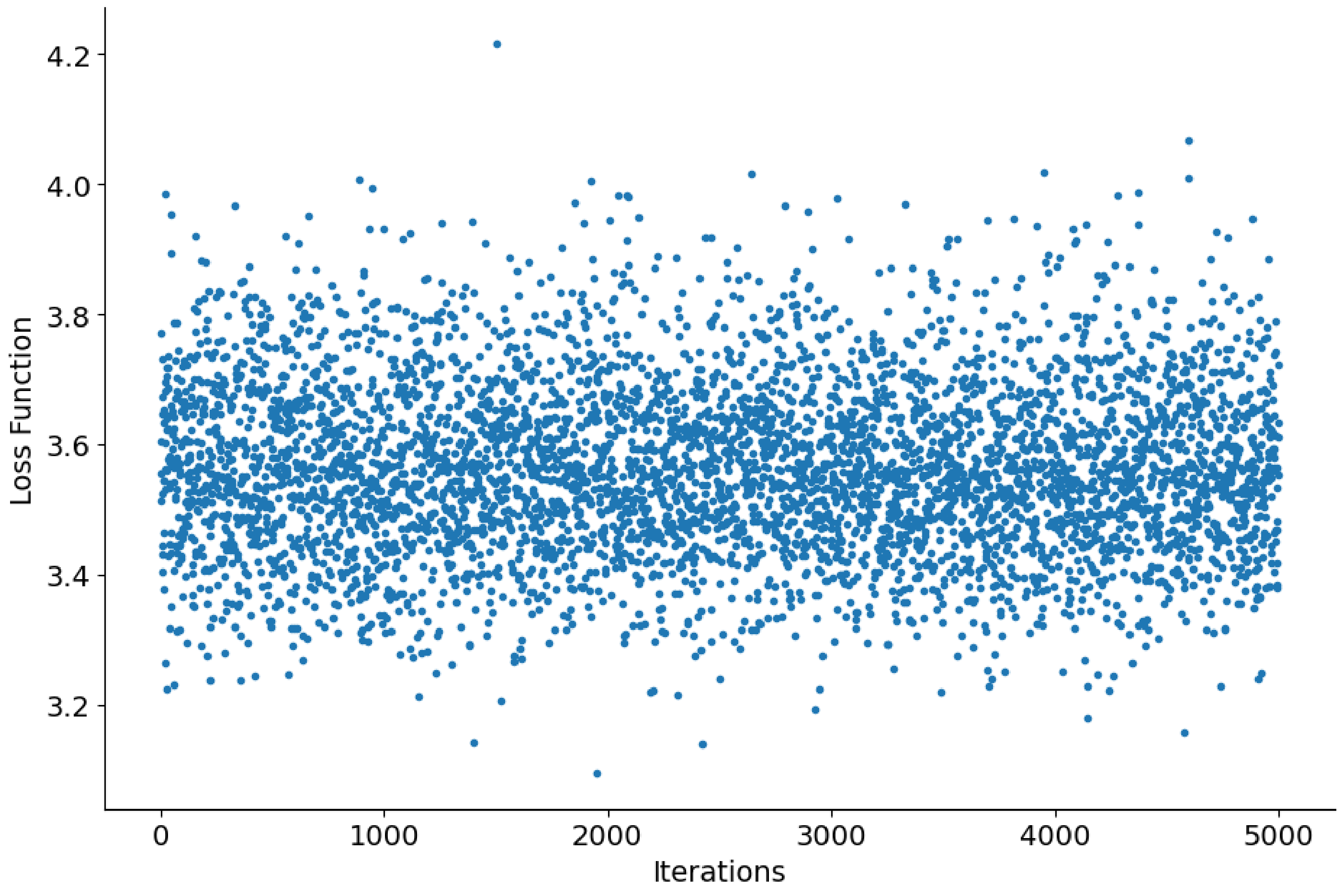
Now we need to find those weights in $\mathbf{W}$ which minimize the loss function. One way to minimize this is to randomly search for matrix entries until we get a better loss function. But thats obviously not very smart.

In [ ]:
```
def random_search(X, y, loss_function, max_iter=10000, rng=None):
    rng = rng or np.random.default_rng(0)

    # we have 10 classes and p attributes, digits from 0 to 9 and 64 pixels
    K = y.shape[1]
    p = X.shape[1]

    #save all losses
    losses = []

    #start with a random weight matrix
    W = rng.normal(0, 0.01, size=(p, K))
    bestloss = np.inf
    for i in tqdm(range(max_iter)):

        # choose a random direction
        W_new = rng.normal(0, 0.01, size=(p, K))
        loss = loss_function(X, y, W_new)
        losses.append(loss)
        if loss < bestloss:
            W = W_new
            bestloss = loss
    return losses

max_iter = 5000
losses_random = random_search(X, y, loss_cross_ent, max_iter=max_iter)
```

```
 0%|          | 0/5000 [00:00<?, ?it/s]
```

In [ ]:
```
plt.figure()
plt.plot(range(max_iter), losses_random, '.')
plt.xlabel('Iterations')
plt.ylabel('Loss Function')
plt.show()
```





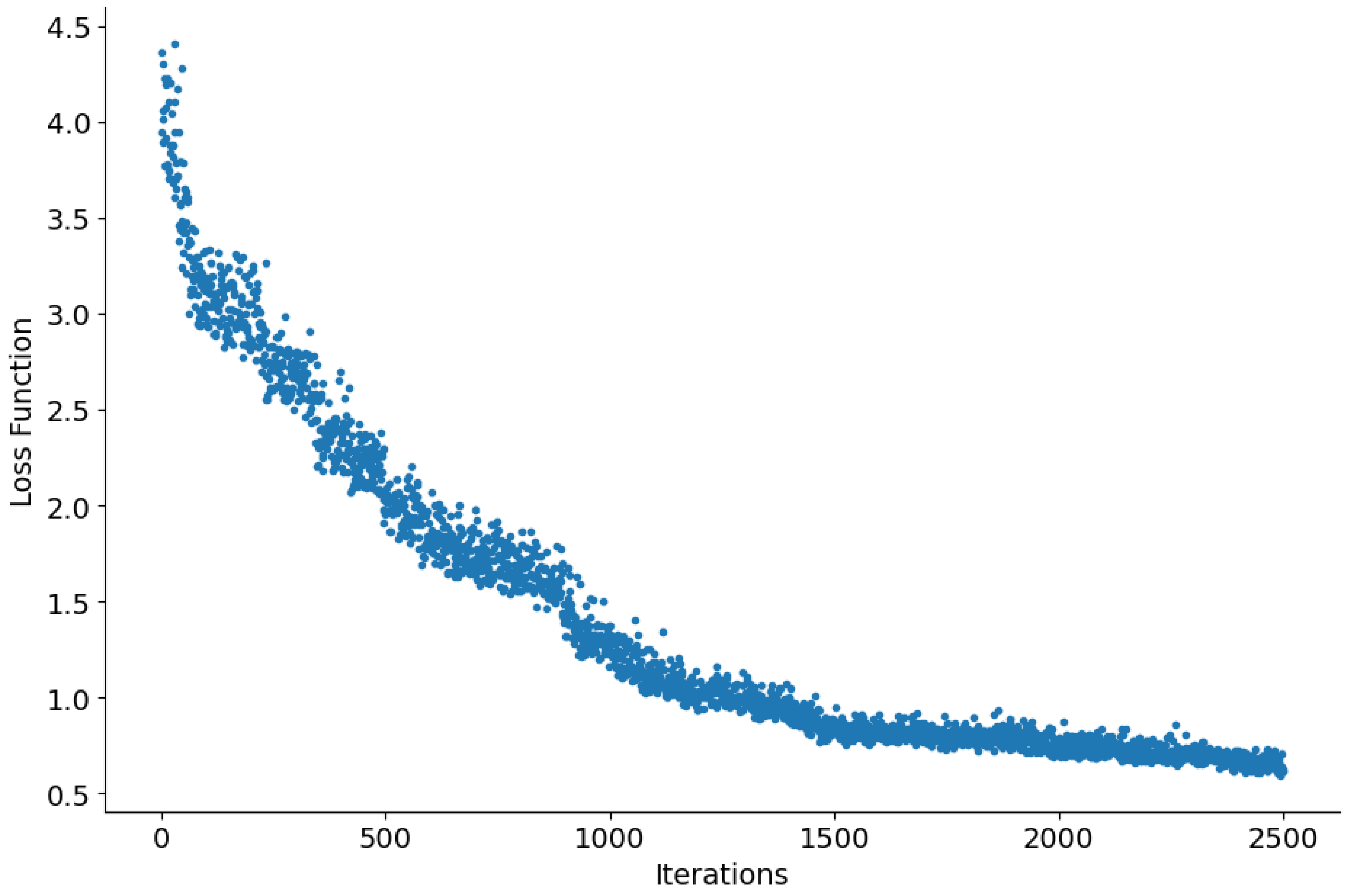
Like in all problems we discussed before, it is actually computationally infeaseble to find the global optimum. We can again try a local search to find a local optimum. Again this is not a show-stopper in real-life situation. Often finding a local optimum suffices. Sometimes the problem can be reformulated to convex in which case the local optimizer is guaranteed to find the global optimum.

Below we try an incremental approach. Which optimizes in a local neighbourhood of $\mathbf{W}$. We keep the best $\mathbf{W}$ we found in the previous iterations and only add a random direction to it.

```
In [ ]: def random_descent(X, y, loss_function, max_iter=10000, step_size=0.01, rng=None):
            rng = rng or np.random.default_rng(0)
            bestloss = np.inf

            # we have 10 classes and p attributes, digits from 0 to 9 and 64 pixels
            K = y.shape[1]
            p = X.shape[1]

            #save all losses
            losses = []

            #start with a random weight matrix
            W = np.random.normal(size=(p, K)) * step_size
            for i in tqdm(range(max_iter)):

                # choose a random direction
                W_new = rng.normal(W, step_size)

                loss = loss_function(X, y, W_new)
                losses.append(loss)
                if (loss < bestloss):
                    W = W_new
                    bestloss = loss
            return losses

        max_iter = 2500
        losses_random = random_descent(X, y, loss_cross_ent, max_iter=max_iter)
```

```
 0%|          | 0/2500 [00:00<?, ?it/s]
```

```
In [ ]: plt.figure()
        plt.plot(losses_random, '.')
        plt.xlabel('Iterations')
        plt.ylabel('Loss Function')
        plt.show()
```



In each step we shoose a random direction to optimize to. We can be smarter about it by following the gradient of the loss function in each iteration. The loss function will converge much faster.

The gradient has to be computed with respect to our weight matrix $\mathbf{W}$

$$L = H_\times(p, q) = -\sum_k p_k \log q_k = -\sum_k p_k \log\left(\frac{e^{f_k(x)}}{\sum_j e^{f_j(x)}}\right)$$

where

$$f = \mathbf{W} \cdot x \,.$$

We can build the derivative using the chain rule.

$$\frac{\partial \mathrm{L}}{\partial \mathrm{w}} = \frac{\partial \mathrm{L}}{\partial \mathrm{q}} \frac{\partial \mathrm{q}}{\partial \mathrm{f}} \frac{\partial \mathrm{f}}{\partial \mathrm{w}}$$

Note that derivatives of vectors always have to take the index into account. Or you can build the Jacobian.

$$\begin{align}
\frac{\partial \mathrm{L}}{\partial \mathrm{f_i}} &= -\sum_k y_k \frac{\partial \log \mathrm{q_k}}{\partial \mathrm{f_i}} \tag{1}\\
&= -\sum_k y_k \frac{1}{q_k} \frac{\partial \mathrm{q_k}}{\partial \mathrm{f_i}} \tag{2}\\
&\vdots \text{ Übungsaufgabe} \tag{3}\\
&= p_i - y_i \tag{4}
\end{align}$$

We use the computed gradient in the code to compute the update to the weight matrix $\mathbf{W}$

```
W = W - dW * step_size
```

The step size, or learning rate controls how far we go down the gradient in each step.

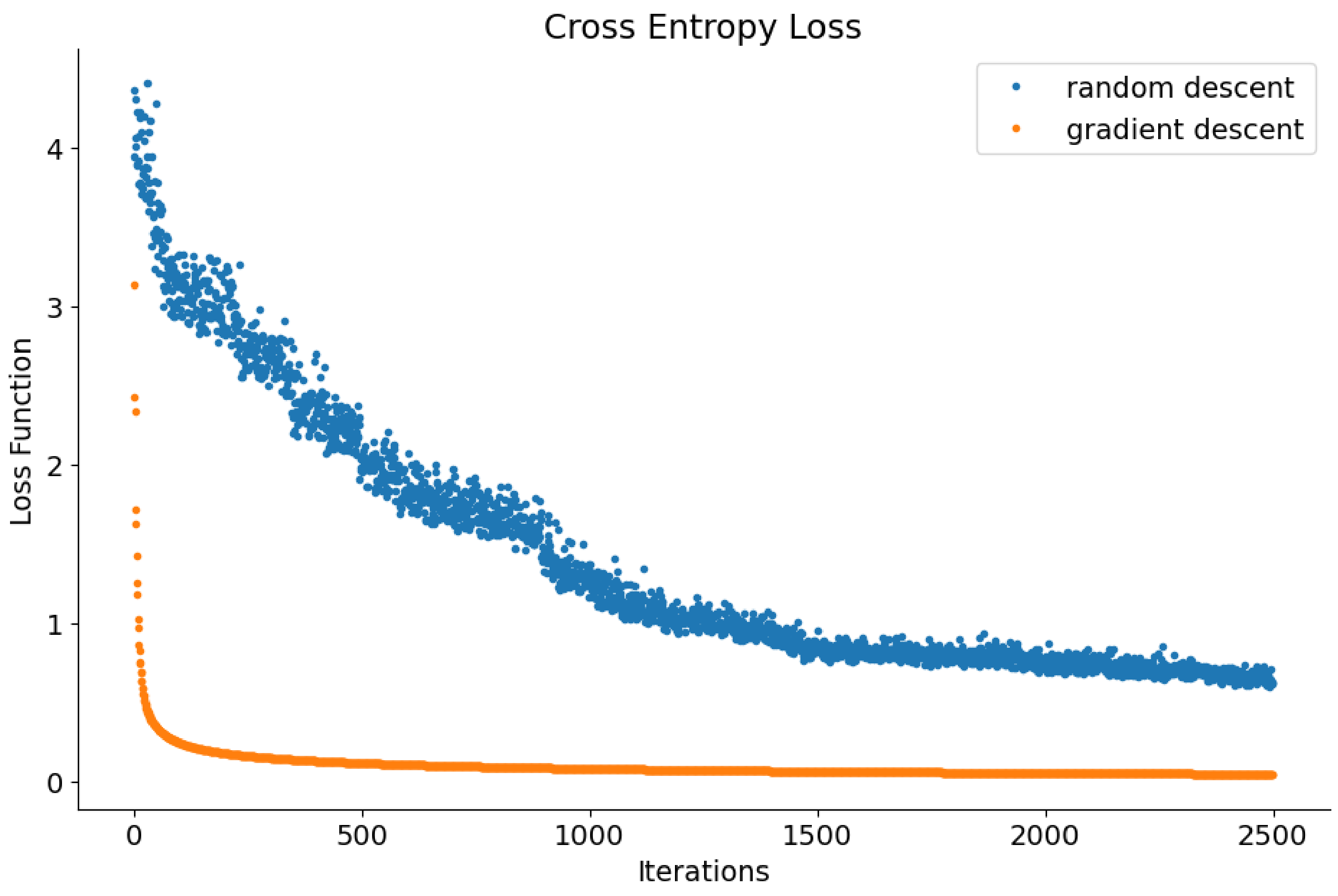
```
In [ ]: def gradient(W, X, y):
            f = X @ W
            p = softmax(f)
            dh = (p - y)
            dW = (X.T @ dh) / dh.shape[0]
            return dW
```

```
def gradient_descent(X, y, loss_function, gradient, max_iter=10000, step_size=0.01, rng=None):
    rng = rng or np.random.default_rng(0)

    bestloss = np.inf

    # we have 10 classes and p attributes, digits from 0 to 9 and 64 pixels
    K = y.shape[1]
    p = X.shape[1]

    #save all losses
    losses = []
    #start with a random weight matrix
    W = rng.normal(0, step_size, size=(p, K))
    for i in tqdm(range(max_iter)):
        W = W - step_size * gradient(W, X, y)
        loss = loss_function(X, y, W)
        losses.append(loss)

    return losses, W
```

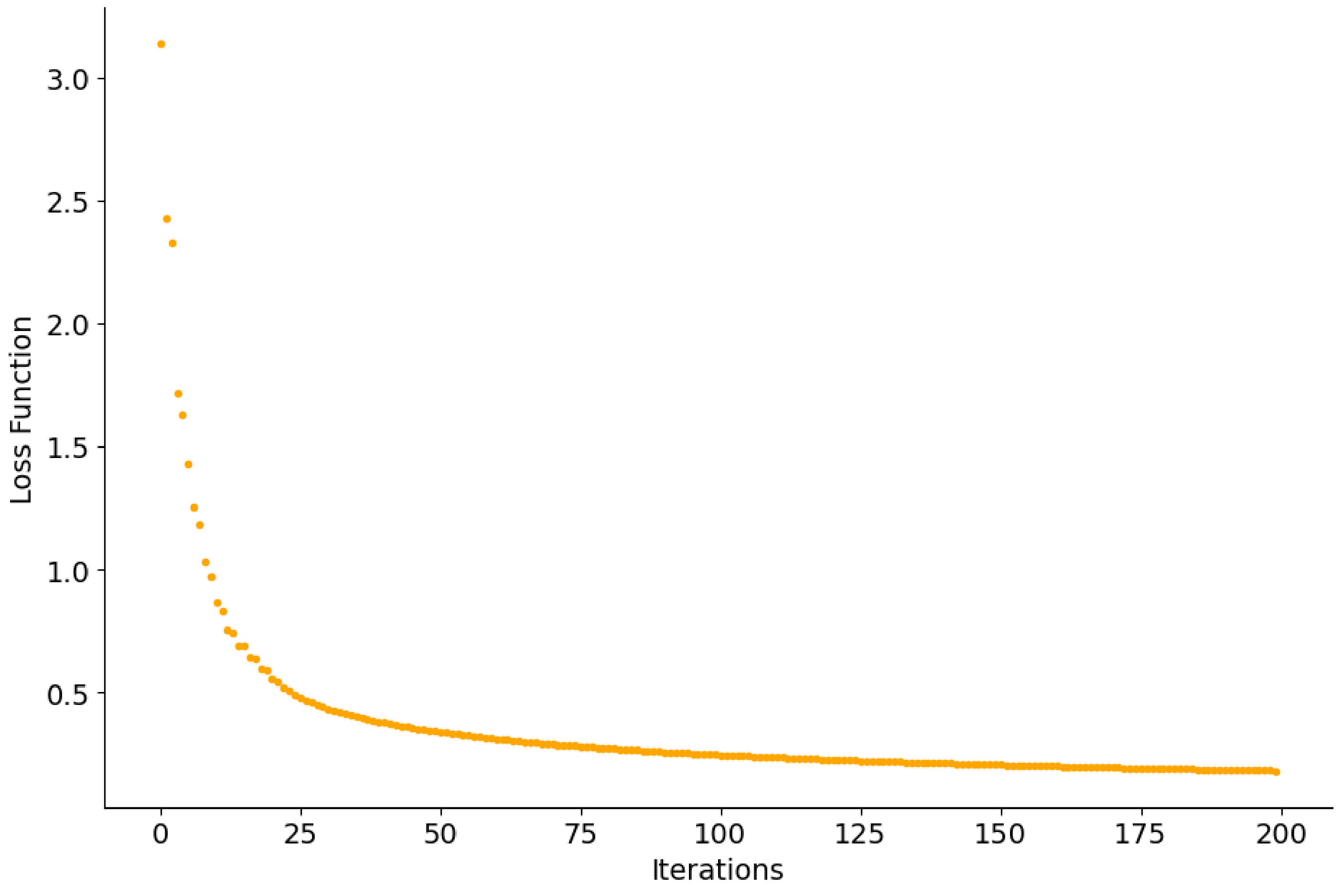
```
In [ ]: X, y = load_digits(return_X_y=True)

        X = np.hstack([ones, X])
        y = label_binarize(y, 10)

        losses, W = gradient_descent(X, y, loss_cross_ent, gradient, max_iter=max_iter)
```

```
 0%|          | 0/2500 [00:00<?, ?it/s]
```

```
In [ ]: plt.figure()
        plt.plot(losses_random, '.', label='random descent')
        plt.plot(losses, '.', label='gradient descent')
        plt.xlabel('Iterations')
        plt.ylabel('Loss Function')
        plt.legend()
        plt.title('Cross Entropy Loss')
        plt.show()
```



```
In [ ]: plt.figure()
        plt.plot(losses[:200], '.', label='gradient descent', color='orange')
        plt.xlabel('Iterations')
        plt.ylabel('Loss Function')
        plt.show()
```



We just implemented the so called SoftMax classifier. This classifier is non-linear. While the simple matrix multiplication $f = \boldsymbol{XW}$ is linear , the application of the softmax function certainly is not. Our prediction vector is given by applying a non-linear function to the output of f

$$p = q(f(x)) = q(\boldsymbol{x} \cdot \boldsymbol{W})$$

Below we apply this classifier to some test data. Clearly this on non-linear step does not suffice to classify non-linear data like the two spirals.

In [ ]:
```
from sklearn.datasets import make_blobs

n_classes = 3

X, y = make_blobs(n_samples=1500, n_features=2, center_box=(-5, 5), centers=n_classes, cluster_std=0.3, random_state=0)

# add biases to matrix
ones = np.ones(shape=(len(X), 1))
X_b = np.hstack([ones, X])
y_b = label_binarize(y, n_classes)

losses, W = gradient_descent(X_b, y_b, loss_cross_ent, gradient, max_iter=150)
```

```
0%|          | 0/150 [00:00<?, ?it/s]
```

In [ ]:
```
prediction = np.argmax(softmax(X_b @ W), axis=1)


cmap = plots.create_discrete_colormap(n_classes)
norm = plt.Normalize(-0.5, n_classes - 0.5)


plt.figure()
plt.axes(aspect='equal')
plt.scatter(X[:,0], X[:, 1], facecolor='none',s=40,  edgecolors=cmap(norm(prediction)))
plt.scatter(X[:,0], X[:, 1], c=y, s=10, cmap=cmap, norm=norm)
```

Out[ ]:
```
<matplotlib.collections.PathCollection at 0x7fa945e31d50>
```

In [ ]:
```
X, y = plots.twospirals(n_samples=300)
n_classes = len(np.unique(y))


#add biases to matrix
ones = np.ones(shape=(len(X), 1))
X_b = np.hstack([ones, X])
```

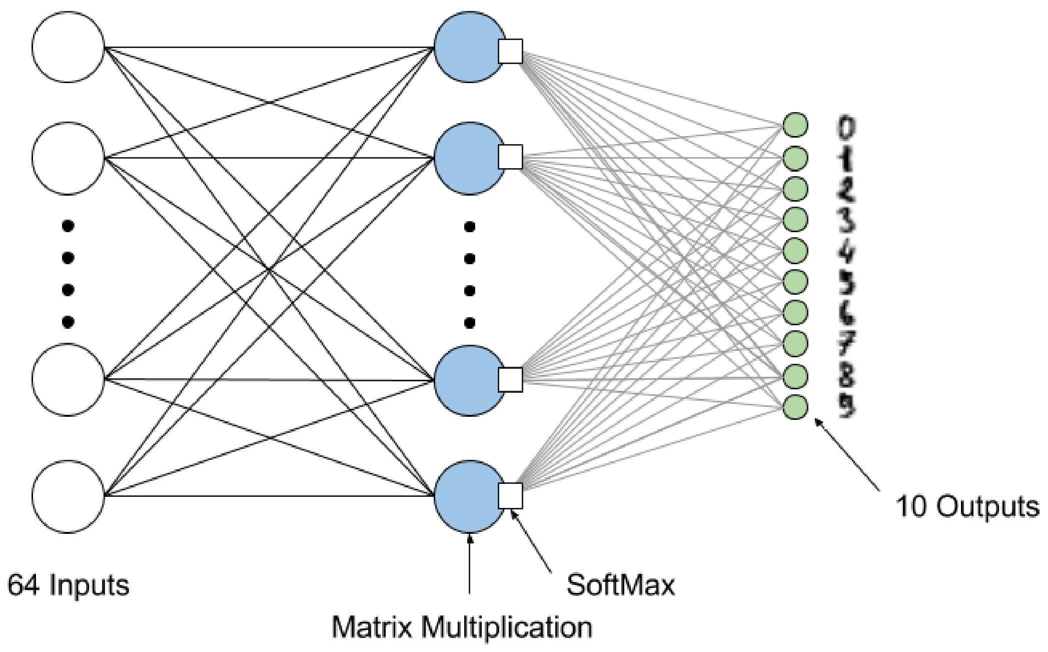
```
y_b = label_binarize(y, n_classes)

losses, W = gradient_descent(X_b, y_b, loss_cross_ent, gradient, max_iter=150)
```

```
0%|          | 0/150 [00:00<?, ?it/s]
```

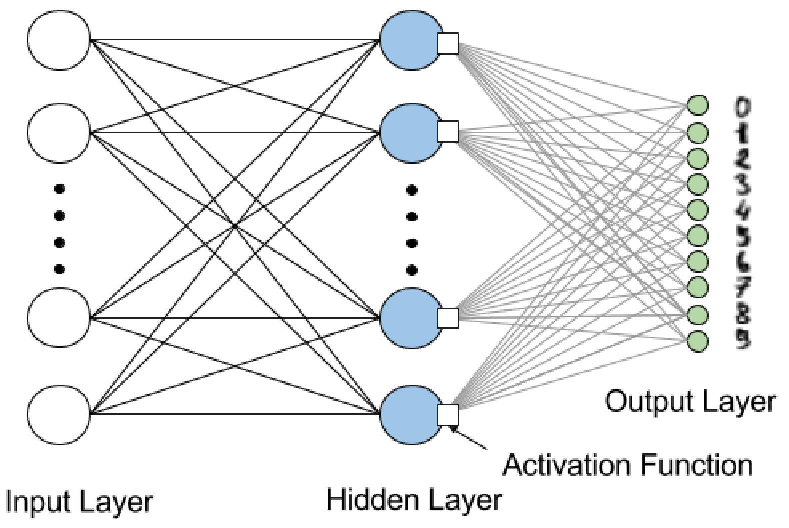
In [ ]:
```
X, y  = plots.twospirals(n_samples=300)

X_b = np.hstack([ones, X])
y_b = label_binarize(y, n_classes)

prediction = np.argmax(softmax(X_b @ W), axis=1)

cmap = plots.create_discrete_colormap(n_classes)
norm = plt.Normalize(-0.5, n_classes - 0.5)

plt.figure()
plt.axes(aspect=1)
plt.scatter(X[:,0], X[:, 1], c=y, cmap=cmap, s=10, label='truth')
plt.scatter(X[:,0], X[:, 1], facecolor='none', s=50,  edgecolors=cmap(norm(prediction)), label='prediction')
plt.legend()

None
```

Neural Networks (Multi Layer Perceptron)

We use another abstraction to describe the linear classification we used above. We started with an input image of 8×8 pixels. We flatten it to get a single vector x_i with 64 entries. In the next step we multiply the vector with a matrix W with dimensions 64×10 and get a result vector of length 10. The following image shows this process in a network like manner.



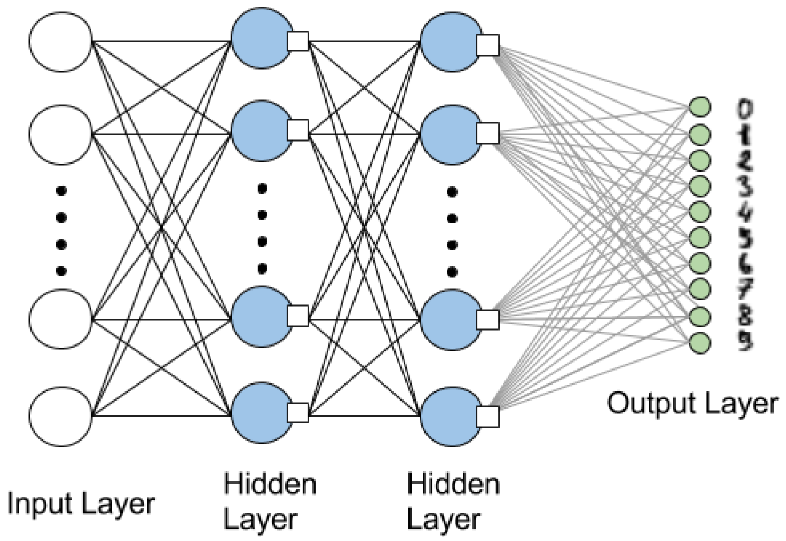
Neural networks use some special slang to describe their architecture. The column of blue dots, i.e. the matrix multiplication, is called a layer. Depending on context this will be called a *hidden layer* or *fully connected layer*. The input and outputs are called the *input* and *output layer* respectively. The white box labeled as Softmax is called *activation function*



We can extend the previous idea further by applying more than one layer of matrix multiplications.

$$p = q_2(q_1(x \cdot W_1), W_2)$$

Each entry in the weight matrices W_1 and W_2 are *learnable* parameters. The matrices have dimension $n_{\text{in}} \times n_{\text{out}}$ (neglecting biases). In the single layer network we trained above we had a single layer with 640 free parameters to optimize. In the case of two full layers we would have twice as many.



The method of finding the gradient for such a complex function is actually not different from what we did before. As long as the Loss function is a linear function with respect to the number of samples. This holds for loss functions like the mean square error, the cross entropy loss or the SVM loss.

$$L = \frac{1}{N} \sum_{n}^{N} L_i$$

As long as the activation functions and the loss function are differentiable the gradient can be calculated either numerically or analytically using **backpropagation**. Which essentially requires nothing more than recursive application of the chain rule of derivatives. In the example above we used backpropagation on a single layer to calculate the gradient.

Below we use the sklearn **multi-layer perceptron** to classify the spiral data.

In [ ]:
```python
from sklearn.neural_network import MLPClassifier

X, y  = plots.twospirals(n_samples=400)

clf = MLPClassifier(hidden_layer_sizes=(10, 10), max_iter=500)
clf.fit(X, y)

X, y  = plots.twospirals(n_samples=400)
prediction  = clf.predict(X)

n_classes = len(np.unique(y))
cmap = plots.create_discrete_colormap(n_classes)
norm = plt.Normalize(-0.5, n_classes - 0.5)


plt.figure()
plt.axes(aspect=1)
plt.scatter(X[:,0], X[:, 1], c=y, cmap=cmap, s=10, label='truth')
plt.scatter(X[:,0], X[:, 1], facecolor='none', s=50,  edgecolors=cmap(norm(prediction)), label='prediction')
plt.legend()
```

Out[ ]:
```
<matplotlib.legend.Legend at 0x7fa945e17670>
```

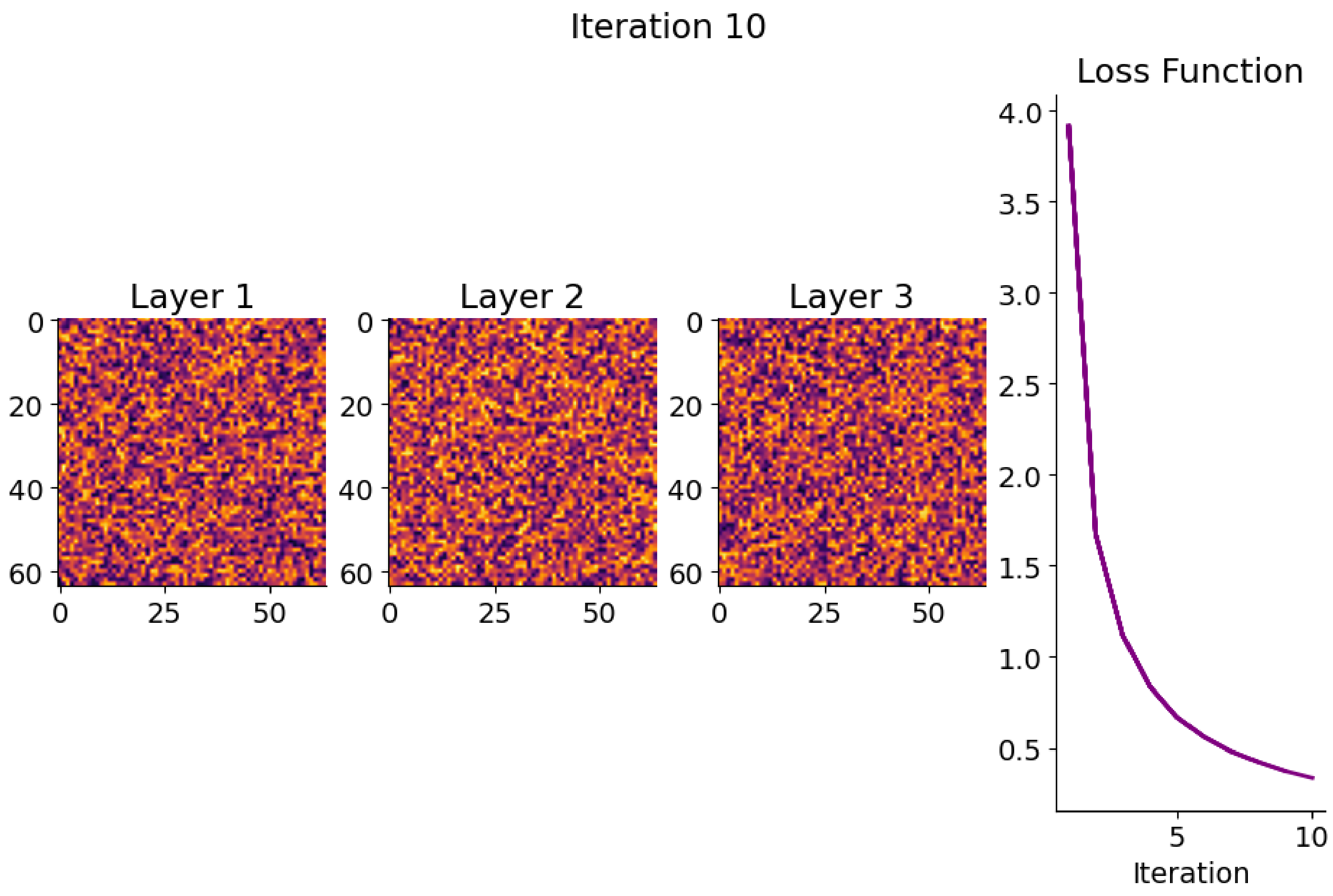
We initialized the MLP with two hidden layers of size 10. As our dataset is two dimensional, the input layer has size 2. The two layers are represented as matrices of size 2×10 which means we have 40 matrix entries (plus biases) to optimize in total.

The more layers and nodes a network has, the higher the representation power. In this case a simple 2 layer network does not suffice to represent the spiral data.

http://cs.stanford.edu/people/karpathy/convnetjs/demo/classify2d.html

In [ ]:
```python
from IPython import display
from sklearn.metrics import confusion_matrix
import time
import warnings
```

```
warnings.filterwarnings('ignore')

fig, [ax1, ax2, ax3, ax4] = plt.subplots(1, 4)

X, y  = load_digits(return_X_y=True)

last = 0
for i in range(10):
    mlp = MLPClassifier(hidden_layer_sizes=(64,64,64), max_iter=i + 1, random_state=43, solver='sgd',)
    mlp.fit(X, y)

    if i > 0 :
        plt.suptitle('Iteration {}'.format(i + 1))

        ax1.set_title('Layer 1')
        ax2.set_title('Layer 2')
        ax3.set_title('Layer 3')
        ax4.set_title('Loss Function')
        ax4.set_xlabel('Iteration')

        diff_0 = np.asarray(mlp.coefs_[0])
        diff_1 = np.asarray(mlp.coefs_[1])
        diff_2 = np.asarray(mlp.coefs_[2])
        diff_3 = np.asarray(mlp.coefs_[3])


        ax1.imshow(diff_0, )
        ax2.imshow(diff_1, )
        ax3.imshow(diff_2, )
        ax4.plot(range(1, i + 2), mlp.loss_curve_, color='purple')


        display.clear_output(wait=True)
        display.display(plt.gcf())

    last_0 = np.asarray(mlp.coefs_[0])
    last_1 = np.asarray(mlp.coefs_[1])
    last_2 = np.asarray(mlp.coefs_[2])
    last_3 = np.asarray(mlp.coefs_[3])

plt.close()
```



Activation Functions

There are a range of activation functions to choose from. The Softmax functions we used for linear classification is often used in the final/output layer of a network to produce probability estimates. An activation function has to be non-linear. Otherwise layers in the network could be merged together into one single matrix operation. Sometimes activation functions are also called non-linearities

Sigmoid

$$g(x) = \frac{1}{1 + e^{-x}}$$

The sigmoid function squashes values into range between 0 and 1. It has been very popular for neural networks in the past but has recently fallen out of favor because it does not behave well when using gradient descent.

$$\mathrm{dg(x)} = \frac{\mathrm{e^x}}{(1 + \mathrm{e^x})^2}$$

As visible in the plot below the gradient becomes zero for large values of x. Which leads to saturations in the gradient descent steps.

```
In [ ]: def sigmoid(x):
            return 1 / (1 + np.exp(-x))

        def sigmoid_gradient(x):
            return np.exp(x) / (1 + np.exp(x))**2


        x = np.linspace(-10, 10, 200)

        fig, ax = plt.subplots()
        ax.plot(x, sigmoid(x), color='purple')
        ax.plot(x, sigmoid_gradient(x), color='purple', ls='--')
        ax.spines['left'].set_position(('data', 0.0))
        ax.spines['bottom'].set_position(('data', 0.0))

        ax.set_title('Sigmoid')
```

```
Out[ ]: Text(0.5, 1.0, 'Sigmoid')
```

Tangens hyperbolicus

$$g(x) = \tanh x$$

The tangens hyperbolicus function behaves very similar to the sigmoid function. Its just a scaled version of it.

$$\mathrm{dg(x)} = 1 - \tanh^2 \mathrm{x}$$

The gradient also becomes zero for large values of x. However the values of the $\tanh$ function are centered around zero which has some advantagous properties when compared to the sigmoid function.

```
In [ ]: def tanh(x):
            return np.tanh(x)

        def tanh_gradient(x):
            return 1 - tanh(x)**2


        x = np.linspace(-10, 10, 200)

        fig, ax = plt.subplots()
        ax.plot(x, tanh(x), color='purple')
        ax.plot(x, tanh_gradient(x), color='purple', ls='--')
        ax.spines['left'].set_position(('data', 0.0))
        ax.spines['bottom'].set_position(('data', 0.0))

        ax.set_title('tanh')
```

```
Out[ ]: Text(0.5, 1.0, 'tanh')
```

Rectified Linear Unit (ReLU)

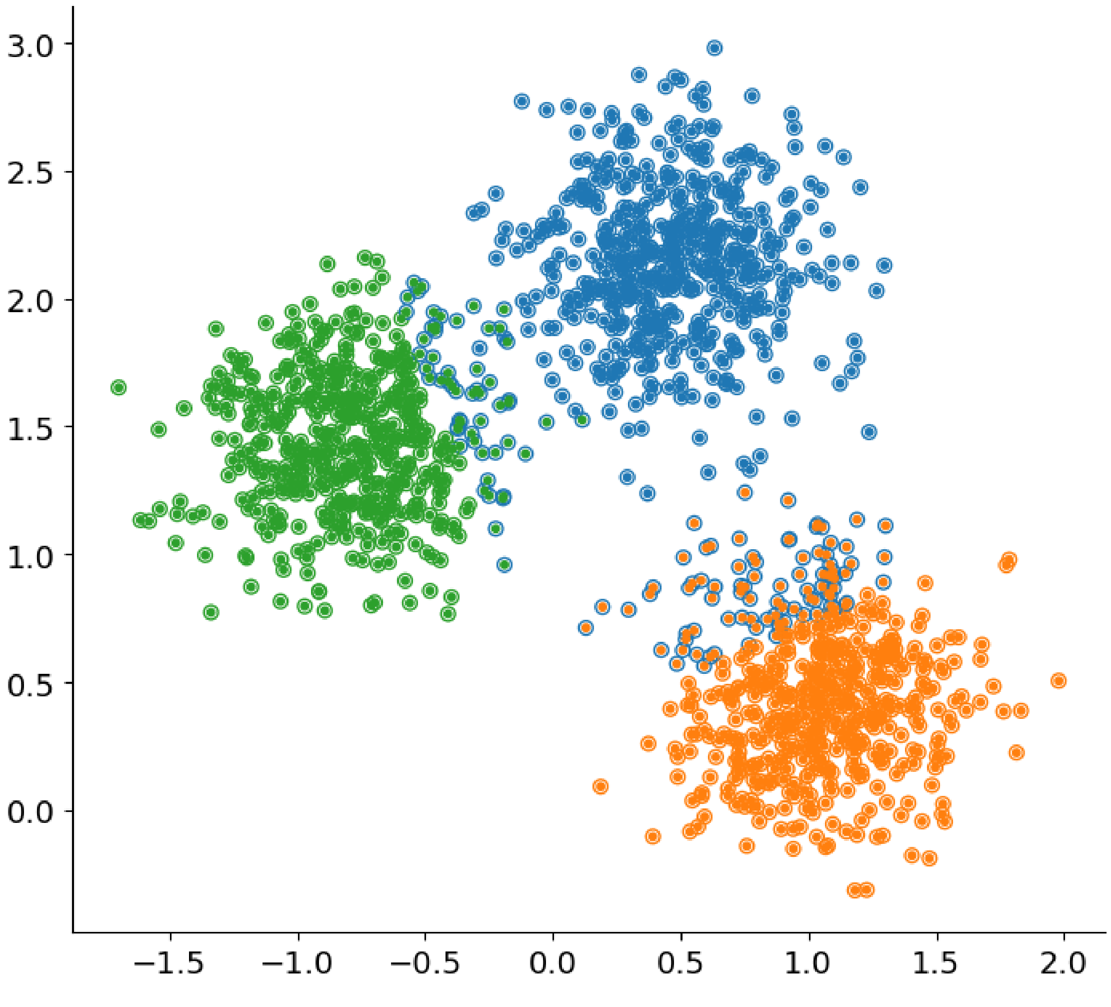
$$g(x) = \max 0, x$$

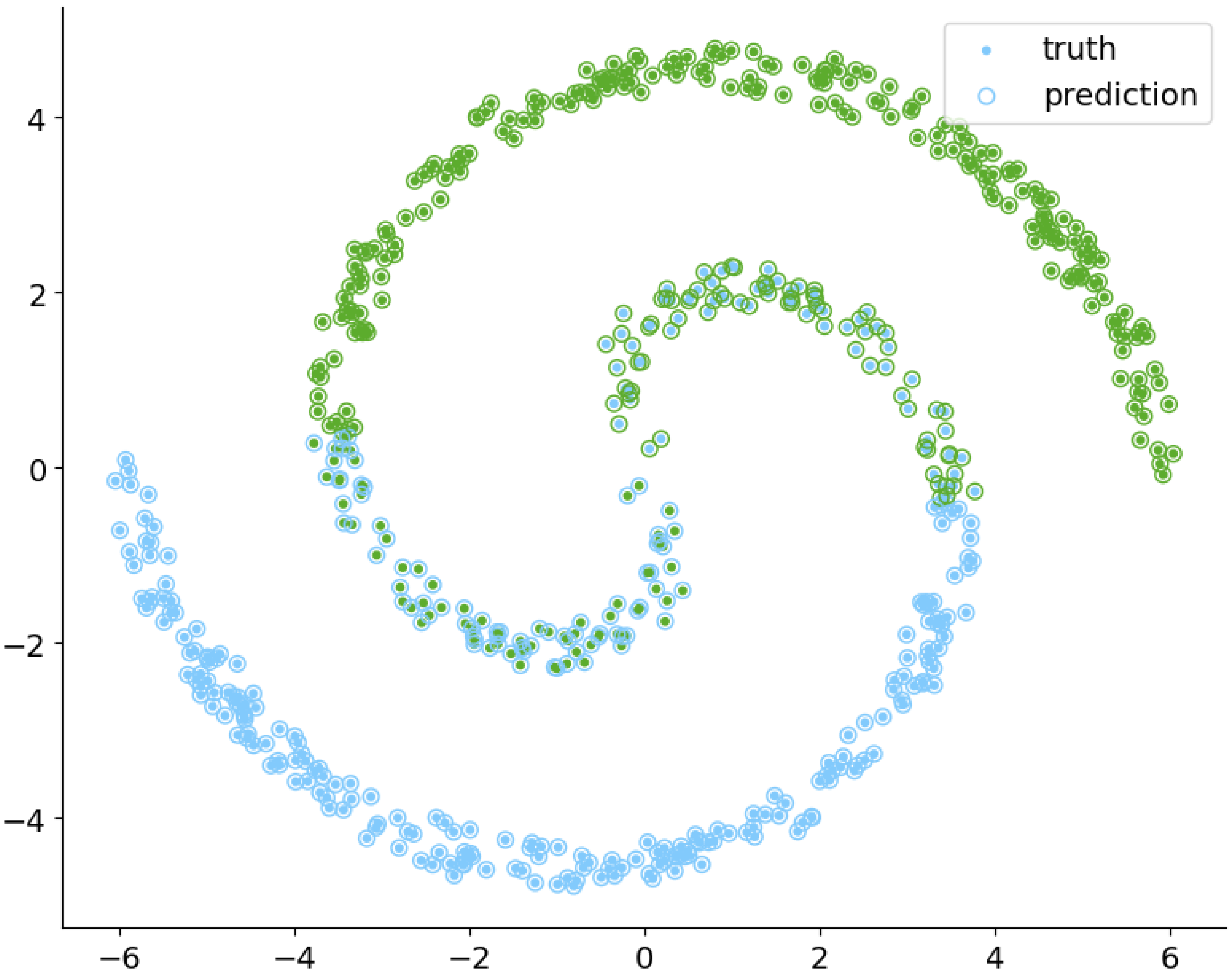
The ReLU function became very popular recently. It was used in some very image recognition networks. Evaluating the function is cheap and it often increases the speed of the gradient descent. See Krizhevsky et al.

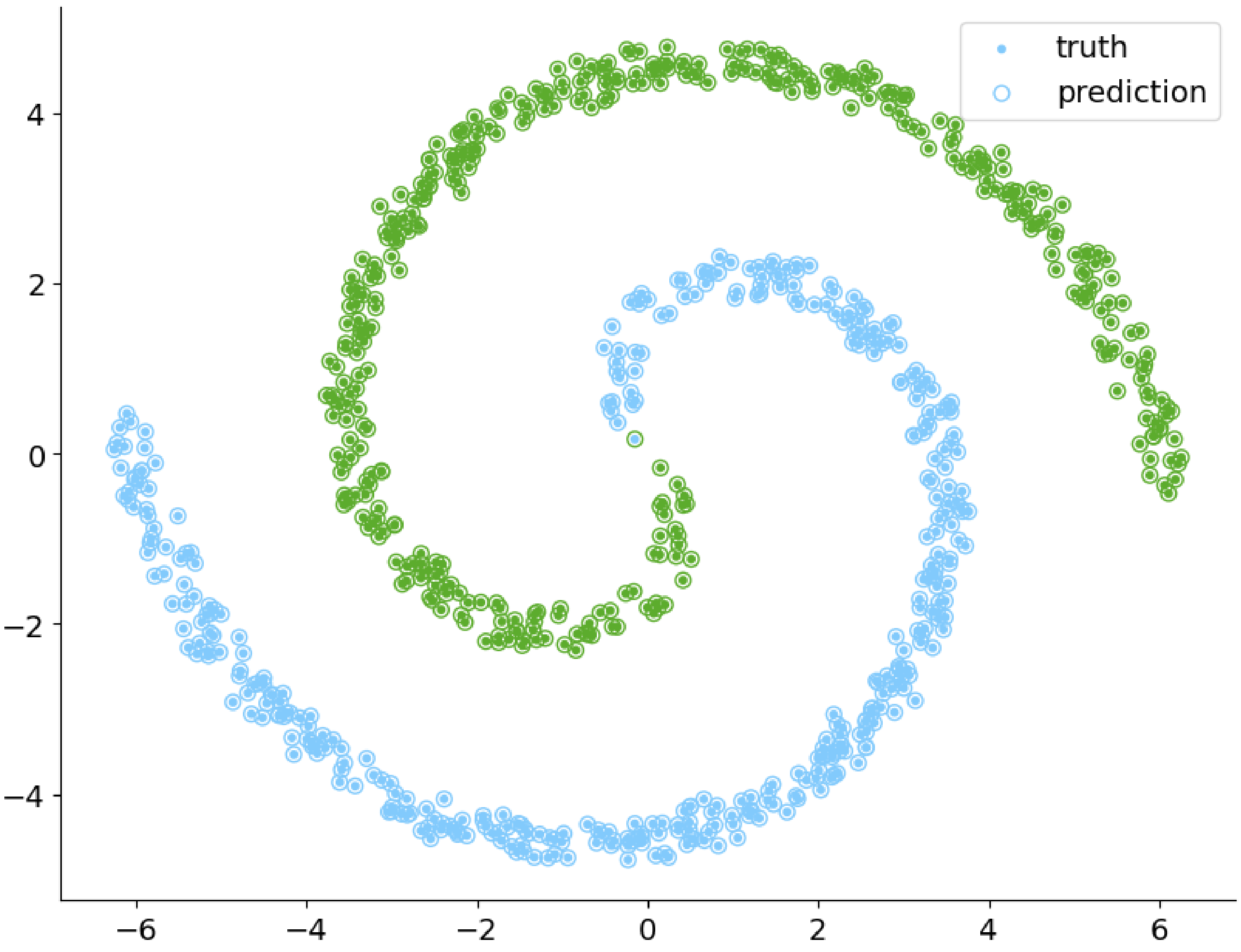
$$\mathrm{dg(x)} = \begin{cases} 1, & \text{if } x > 0 \\ \text{undefined}, & \text{if } x = 0 \\ 0, & \text{if } x < 0 \end{cases}$$

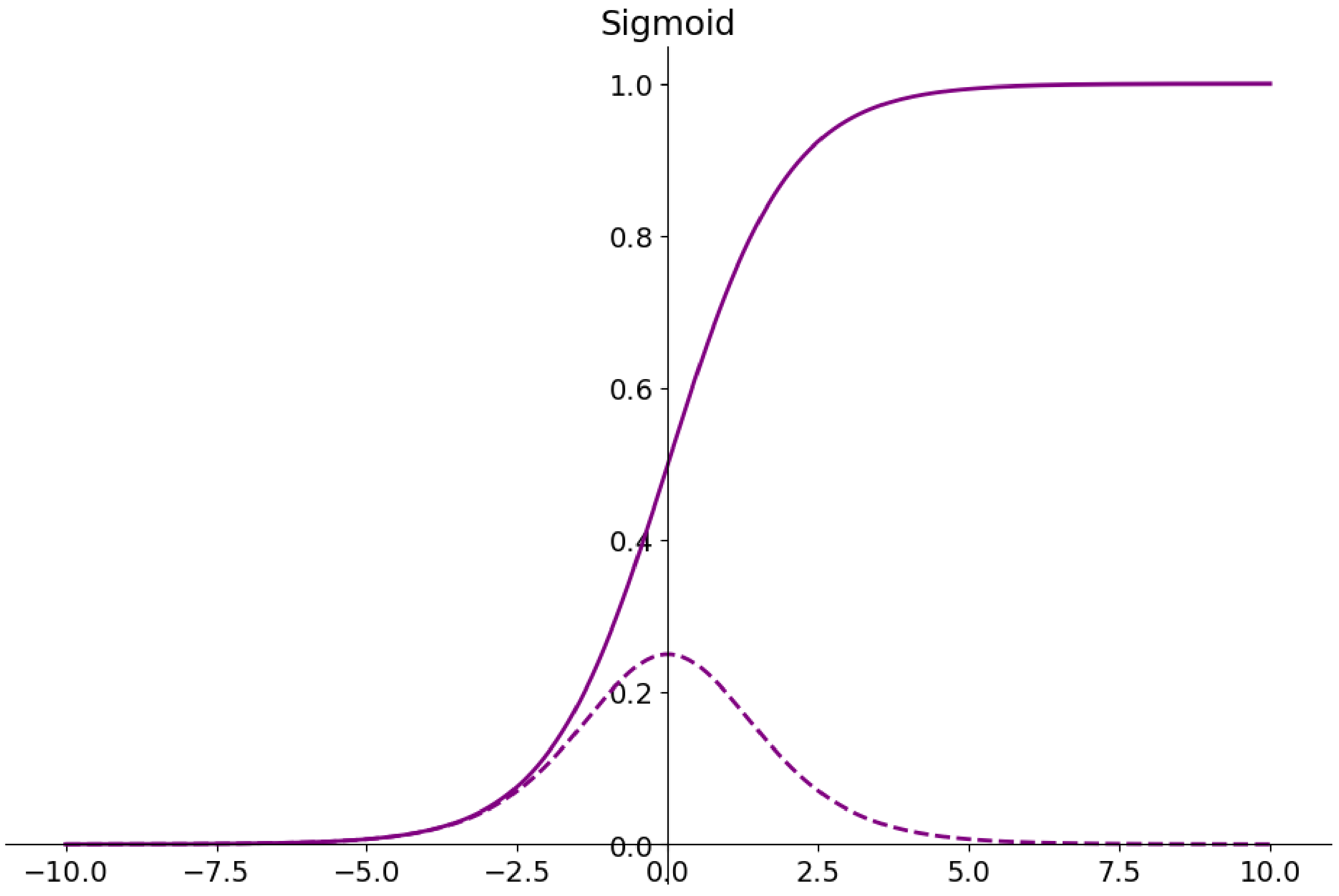
The gradient is zero for value smaller than 0 and is discontinous. This can lead to a problem called 'dying neurons'. The softplus function is a common alternative.

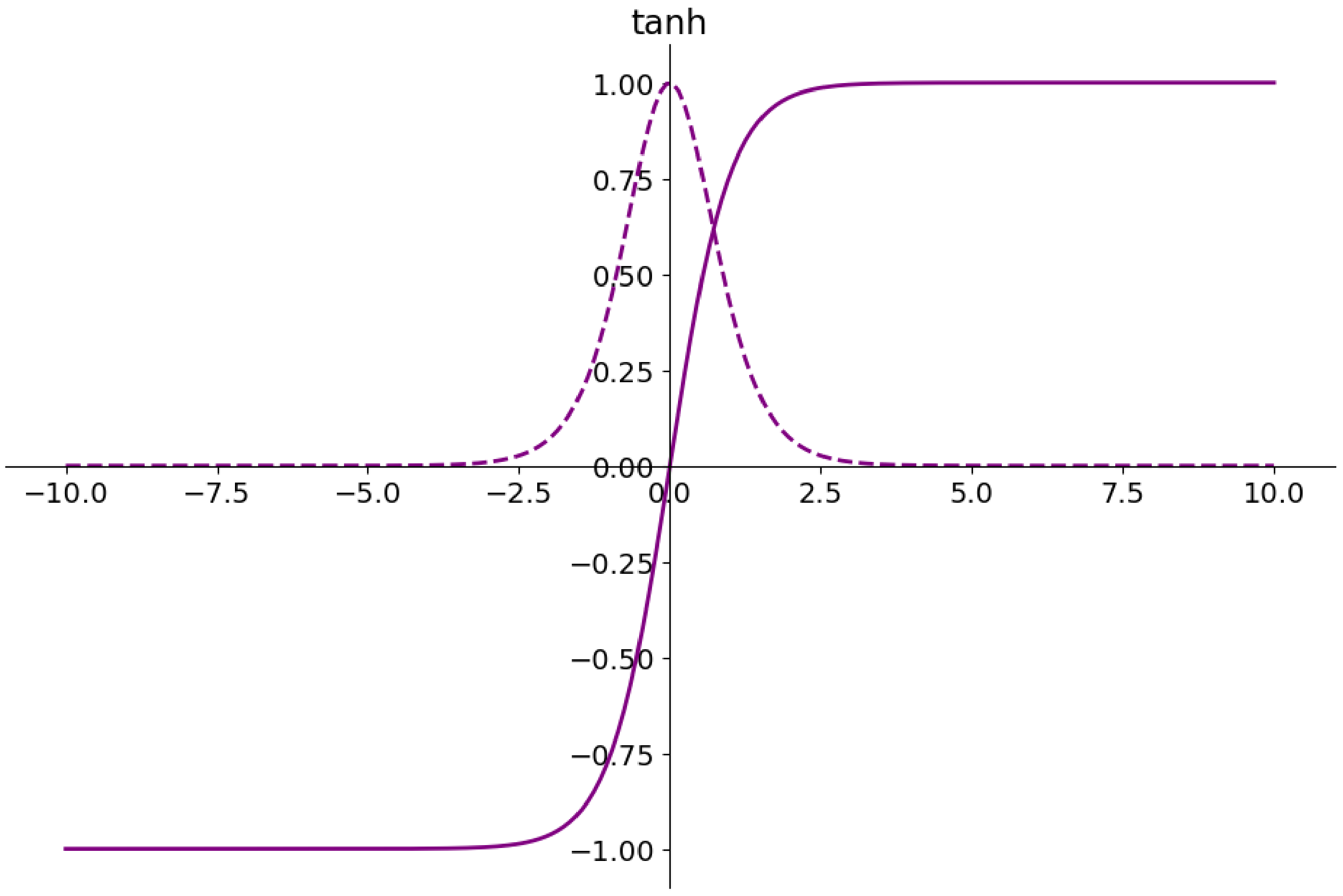
In [ ]:
```python
def relu(x):
    return np.maximum(0, x)

def relu_gradient(x):
    return np.clip(np.sign(x), 0, 1)

def softplus(x):
    return np.log(1 + np.exp(x))

def softplus_gradient(x):
    return 1 / (1 + np.exp(-x))

x = np.linspace(-3, 3, 1000)


fig, ax = plt.subplots()
ax.plot(x, relu(x), label='relu', color='purple')
ax.plot(x, relu_gradient(x), label='relu gradient', color='darkred', ls='--')
ax.plot(x, softplus(x), label='softplus', color='darkorange')
ax.plot(x, softplus_gradient(x), label='softplus gradient', color='darkorange', ls='--')
ax.spines['left'].set_position(('data', 0.0))
ax.spines['bottom'].set_position(('data', 0.0))
ax.legend()
plt.show()
```

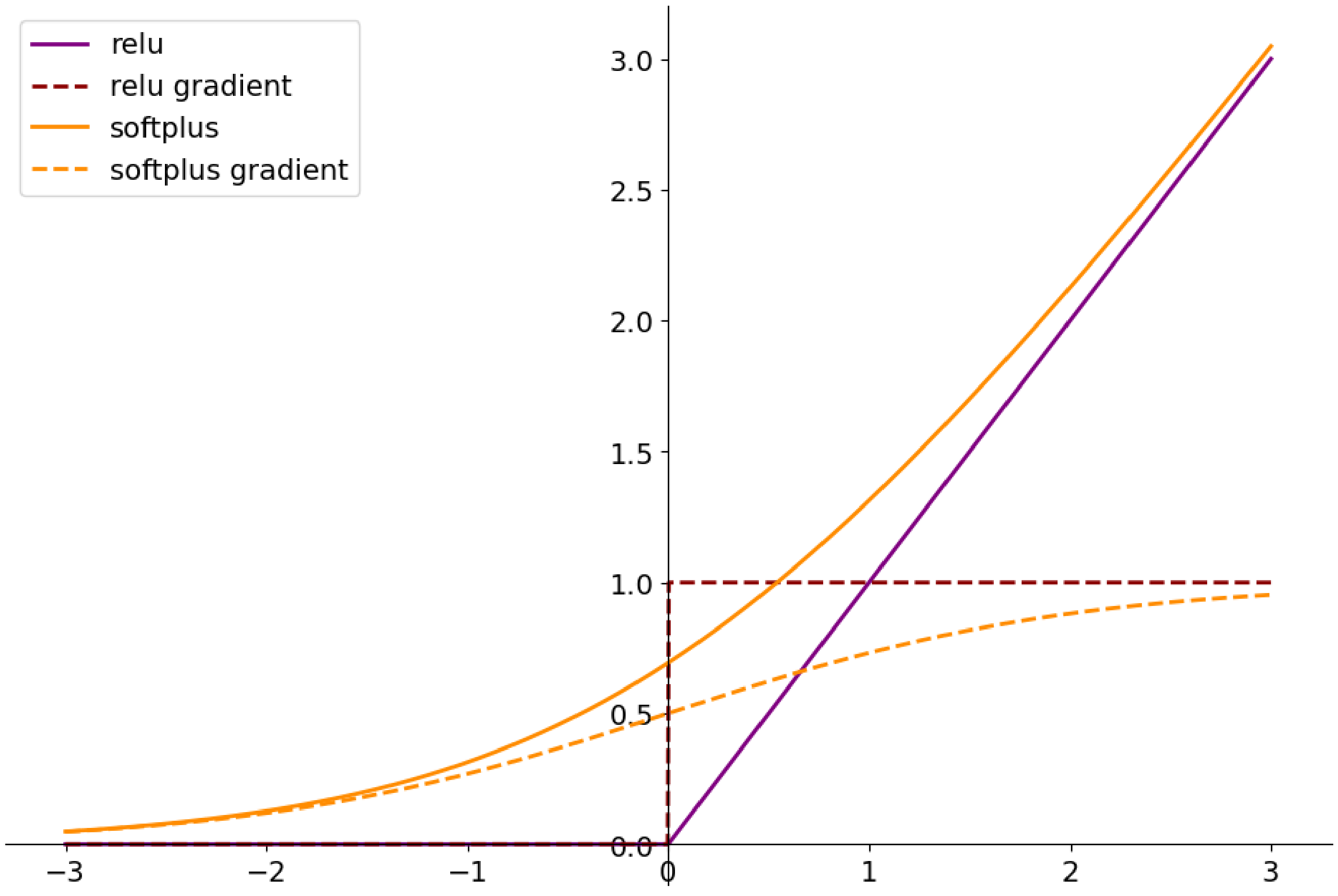



truth
prediction
4
2
0
−2
−4
−6
−4
−2
0
2
4
6


truth
prediction
4
2
0
−2
−4
−6
−4
−2
0
2
4
6


Sigmoid
1.0
0.8
0.6
0.4
0.2
0.0
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0


tanh
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0



Regularization

We've seen how easily neural networks can overfit. We can try to avoid that using two techniques.

- Add a penalty term to the loss function.
- Add special regularization layers.

Adapting the loss function is a common method to avoid overfitting not only for neural networks. We have seen a similar idea for the SVM loss function.

A common approach is the **L2** regularization.

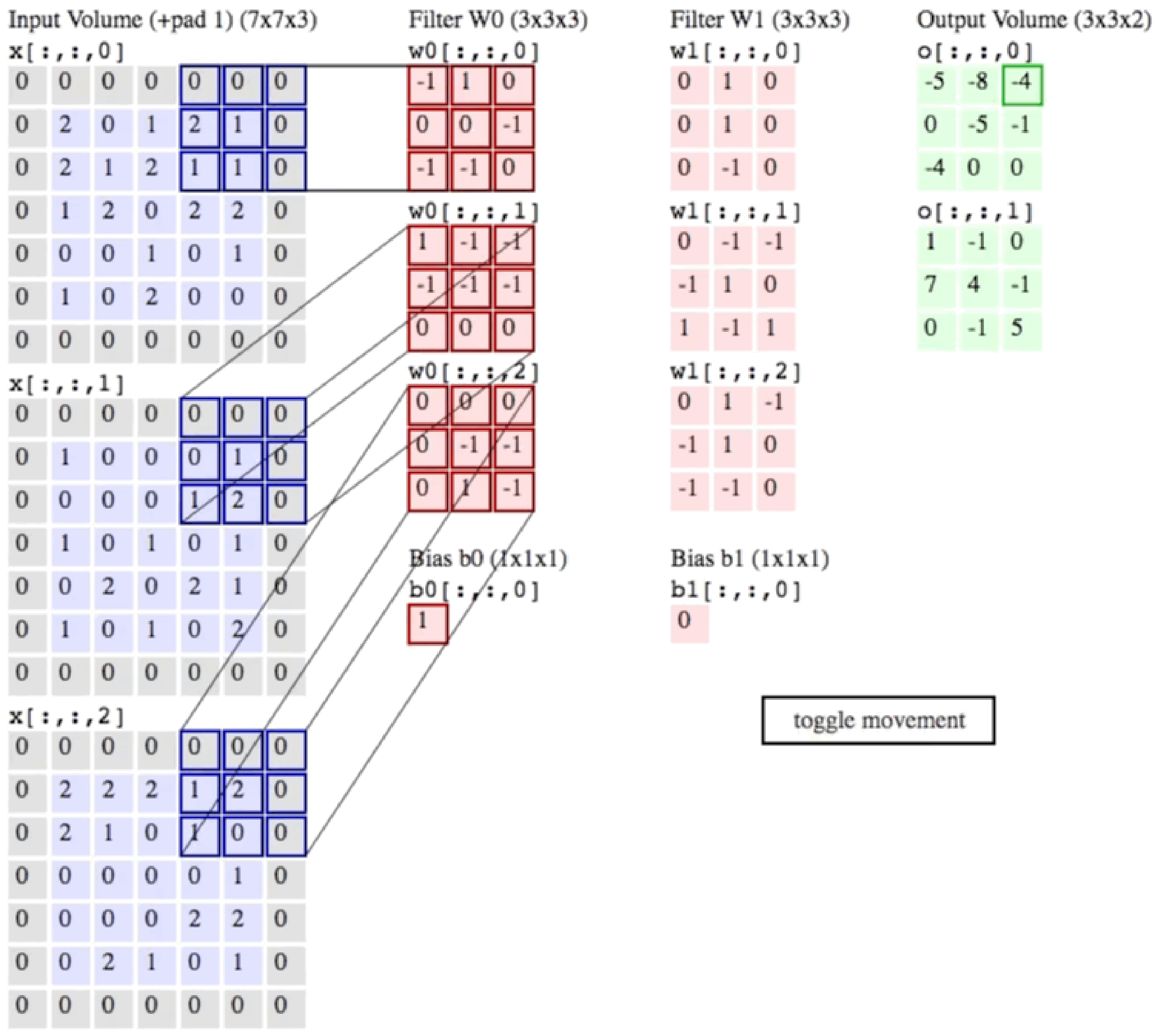
$$L_r = L + \frac{1}{2}\lambda w^2$$

It avoids strong changes in w and creates more diffuse weight matrices.

So called **dropout layers** randomly select neurons to activate during the training. Each iteration only optimizes a subset of neurons.

Convolutional Neural Nets for Image Recognition

The basic idea behind CNNs are simplifications made for handlin high dimensional inputs like high resolution images. Before the first fully connected layers are trained on the input data a so called convolution kernel is applied to the image. The parameters of this kernel are free to be optimized during training. The weights for each region in the image are shared however. This reduces the amount of parameters need to train a network and also leads to **translational invariance**.



Neural Networks in practice

A few key points to take away:

- The more layers the network the more powerful it becomes. However training might not converge or overfitting might occur.
- The term neural network is a misnomer. That's not how real brains work.
- You need regularization to avoid overfitting.
- The word *Deep* in *Deep Learning* refers to the number of hidden layers. No magic involved but a lot of hype.
- Do not use the sigmoid activation. Use ReLU if you are unsure.

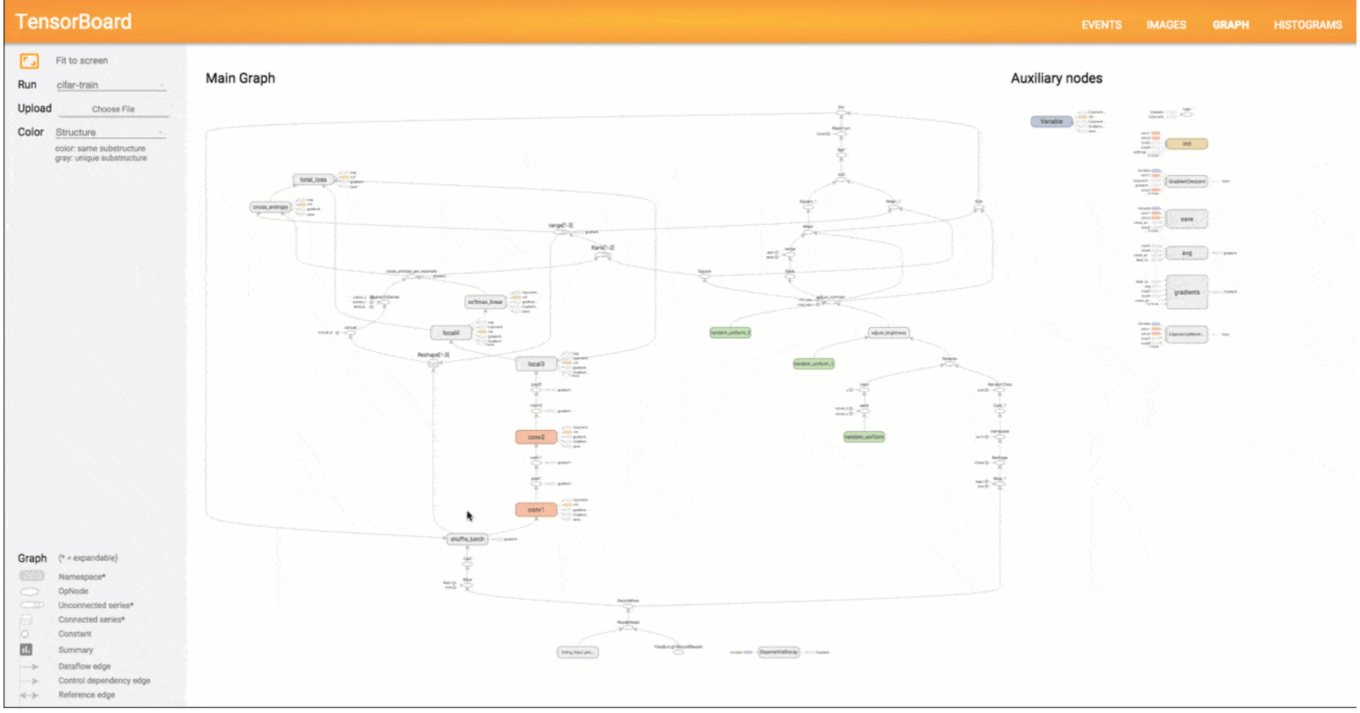
Software

Large networks require considerable computing power to optimize. Matrix computations are well suited for GPUs. The scikit-learn library does not support GPU computation out of the box. It also does not support convoluted neural networks.

There is a whole zoo of software libraries to train large neural networks. Here are some important ones.

Tensorflow

Developed by Google Inc. Maybe the most popular solution.



Keras

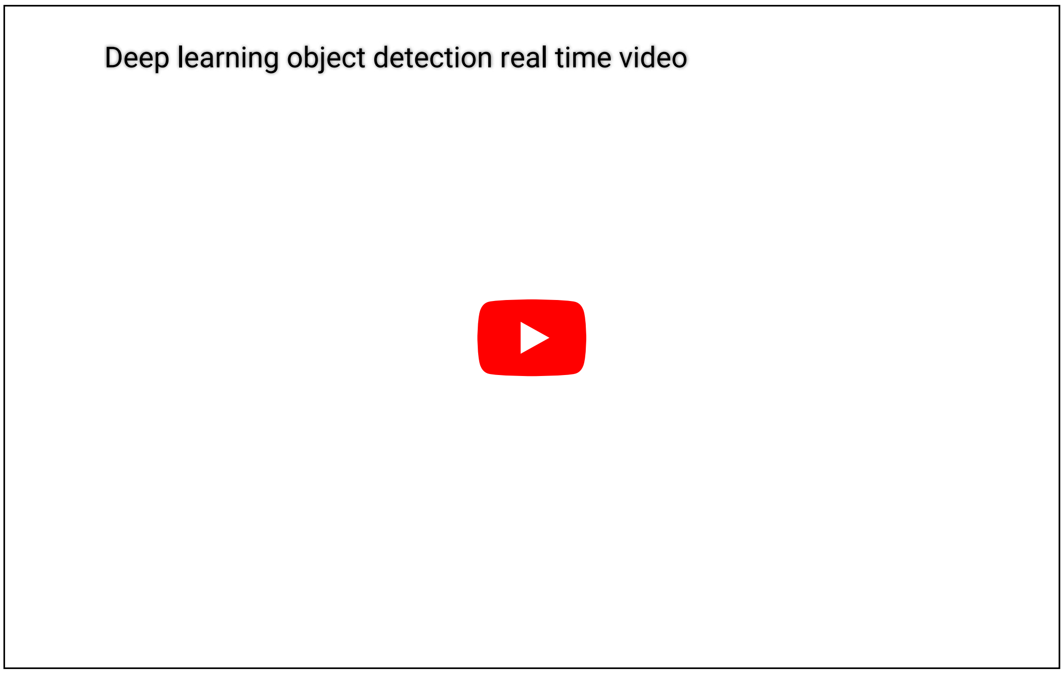
High-Level API build ontop of TF

pytorsch

Facebook's solution https://github.com/pytorch/pytorch

```
In [ ]: from IPython.display import HTML

        HTML('<iframe width="660" height="415" src="https://www.youtube.com/embed/cf0JG0791P4" frameborder="0" gesture="media" al
```

Out[ ]:



Deep learning object detection real time video

```
In [ ]:
```