

```

import autograd.numpy as np # We need to use this numpy wrapper to
make automatic differentiation work later
from sklearn import datasets
import matplotlib.pyplot as plt
from sklearn.metrics import accuracy_score

# Defining some activation functions
def ReLU(z):
    return np.where(z > 0, z, 0)

def sigmoid(z):
    return 1 / (1 + np.exp(-z))

def softmax(z):
    """Compute softmax values for each set of scores in the rows of
the matrix z.
Used with batched input data."""
    e_z = np.exp(z - np.max(z, axis=0))
    return e_z / np.sum(e_z, axis=1)[:, np.newaxis]

def softmax_vec(z):
    """Compute softmax values for each set of scores in the vector z.
Use this function when you use the activation function on one
vector at a time"""
    e_z = np.exp(z - np.max(z))
    return e_z / np.sum(e_z)

np.random.seed(2024)

x = np.random.randn(2) # network input. This is a single input with
two features
W1 = np.random.randn(4, 2) # first layer weights

```

Exercise 1a)

The shape of the input is 2, and the shape of the output of the first layer is 4, which we see from the properties of matrix multiplication.

```

# Exercise 1b)
b1 = np.random.randn(4)

# Exercise 1c)
z1 = W1 @ x + b1

# Exercise 1d)
a1 = ReLU(z1)

```

```
sol1 = np.array([0.60610368, 4.0076268, 0.0, 0.56469864])
print(np.allclose(a1, sol1))
True
```

Exercise 2a)

The input to the second layer is the output from the previous iteration a_1 , and its shape is 4.

```
# Exercise 2b)
W2 = np.random.randn(8, 4)
b2 = np.random.randn(8)

# Exercise 2c)
z2 = W2 @ a1 + b2
a2 = ReLU(z2)

print(
    np.allclose(np.exp(len(a2)), 2980.9579870417283)
) # This should evaluate to True if a2 has the correct shape :)
```

True

```
# Exercise 3a)
def create_layers(network_input_size, layer_output_sizes):
    layers = []

    i_size = network_input_size
    for layer_output_size in layer_output_sizes:
        W = np.random.randn(layer_output_size, i_size)
        b = np.random.randn(layer_output_size)
        layers.append((W, b))

        i_size = layer_output_size
    return layers

# Exercise 3b)
def feed_forward_all_relu(layers, input):
    a = input
    for W, b in layers:
        z = W @ a + b
        a = ReLU(z)
        print(a.shape)
    return a

# Exercise 3c)
input_size = 8
layer_output_sizes = [10, 16, 6, 2]

x = np.random.rand(input_size)
```

```

layers = create_layers(input_size, layer_output_sizes)
predict = feed_forward_all_relu(layers, x)
print(predict)

(10,)
(16,)
(6,)
(2,)
[5.36337158  0.          ]

```

Exercise 3d)

A neural network with no activation function is always mathematically equivalent to a neural network with only one layer because the weight matrices and bias vectors can all be multiplied and added together such that it is possible to take the input vector x and multiply it by a single matrix A and add a single vector d that will give us the same result as doing the process iteratively with more matrices and bias vectors. Introducing an activation function between each layer changes this property, because we can no longer guarantee that we can "extract" the original x and multiply it by a single matrix and add it to a single vector.

```

# Exercise 4a)
def feed_forward(input, layers, activation_funcs):
    a = input
    for (W, b), activation_func in zip(layers, activation_funcs):
        z = W @ a + b
        a = activation_func(z)
    return a

# Exercise 4b)
network_input_size = 4
layer_output_sizes = [12, 6, 8]
activation_funcs = [ReLU, ReLU, sigmoid]
layers = create_layers(network_input_size, layer_output_sizes)

x = np.random.randn(network_input_size)
feed_forward(x, layers, activation_funcs)

array([0.99848945, 0.99782321, 0.88940841, 0.98204008, 0.87767592,
        0.77041007, 0.99949516, 0.96909165])

# Exercise 4c)
network_input_size = 4
layer_output_sizes = [12, 6, 8]
activation_funcs = [sigmoid, sigmoid, ReLU]
layers = create_layers(network_input_size, layer_output_sizes)

x = np.random.randn(network_input_size)
feed_forward(x, layers, activation_funcs)

```

```
array([0.37452327, 1.20327283, 0.8628437 , 0.          , 0.20421673,
       0.          , 1.21546228, 0.          ])
```

The output of the network with sigmoid in the hidden layers and ReLU in the output layer changes in the sense that the output will reflect the behavior of the last function utilized. ReLU sets certain indices to 0 when certain conditions are met, whereas sigmoid will never do that, which is why this effect is visible only with the altered order of activation functions.

```
# Exercise 5a)
def create_layers_batch(network_input_size, layer_output_sizes):
    layers = []

    i_size = network_input_size
    for layer_output_size in layer_output_sizes:
        W = np.random.randn(layer_output_size, i_size).T
        b = np.random.randn(layer_output_size)
        layers.append((W, b))

        i_size = layer_output_size
    return layers

# Exercise 5b)
inputs = np.random.rand(1, 4)

def feed_forward_batch(inputs, layers, activation_funcs):
    a = inputs
    for (W, b), activation_func in zip(layers, activation_funcs):
        z = a @ W + b
        a = activation_func(z)
    return a

# Exercise 5c)
network_input_size = 4
layer_output_sizes = [2, 3, 1]
activation_funcs = [ReLU, ReLU, softmax]
layers = create_layers_batch(network_input_size, layer_output_sizes)

feed_forward_batch(inputs, layers, activation_funcs)

array([[1.]])
```

We tested for equivalent inputs for our single-input functions, and it works! We couldn't include it here because the testing involves a lot kernel restarting and commenting out sections of code so the random functions work correctly.

Exercise 6a)

The input size for this network should be a matrix with shape (150, 4) for the 150 data points and the four features. The output size of the last layer should be (150, 3), since we have three different categories.

```

iris = datasets.load_iris()

_, ax = plt.subplots()
scatter = ax.scatter(iris.data[:, 0], iris.data[:, 1], c=iris.target)
ax.set(xlabel=iris.feature_names[0], ylabel=iris.feature_names[1])
_ = ax.legend(
    scatter.legend_elements()[0], iris.target_names, loc="lower
right", title="Classes"
)

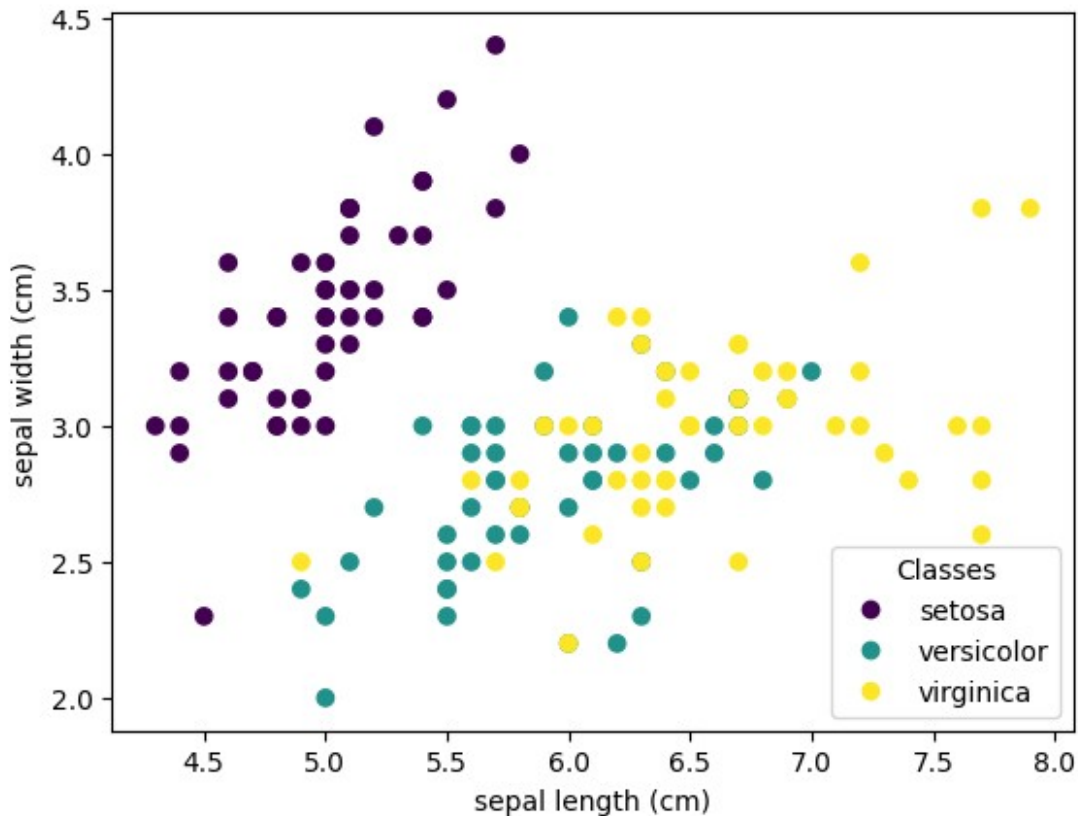
inputs = iris.data

# Since each prediction is a vector with a score for each of the three
types of flowers,
# we need to make each target a vector with a 1 for the correct flower
and a 0 for the others.
targets = np.zeros((len(iris.data), 3))
for i, t in enumerate(iris.target):
    targets[i, t] = 1

def accuracy(predictions, targets):
    one_hot_predictions = np.zeros(predictions.shape)

    for i, prediction in enumerate(predictions):
        one_hot_predictions[i, np.argmax(prediction)] = 1
    return accuracy_score(one_hot_predictions, targets)

```



```
# Exercise 6b)
# Create a network with 2 hidden layers.
network_input_size = 4
layer_output_sizes = [8, 3]
activation_funcs = [sigmoid, softmax]
layers = create_layers_batch(network_input_size, layer_output_sizes)

# Exercise 6c)
predictions = feed_forward_batch(inputs, layers, activation_funcs)
print(predictions)

[[0.25763986 0.26127581 0.48108432]
 [0.26049451 0.27748468 0.46202081]
 [0.25747878 0.26615222 0.476369  ]
 [0.25856502 0.26995157 0.47148341]
 [0.25638148 0.2570102  0.48660832]
 [0.25531283 0.25132223 0.49336494]
 [0.25512398 0.25734226 0.48753377]
 [0.25826543 0.26400524 0.47772933]
 [0.25852756 0.27477892 0.46669352]
 [0.26117103 0.27603834 0.46279063]
 [0.25782422 0.25901232 0.48316345]
 [0.25761852 0.26218633 0.48019516]
 [0.26098048 0.27809181 0.46092771]]
```

[0.25683892 0.26914457 0.47401651]
[0.2564729 0.25322989 0.49029721]
[0.25261871 0.24181761 0.50556368]
[0.25430637 0.24955183 0.4961418]
[0.25680058 0.25964469 0.48355473]
[0.25823568 0.25888954 0.48287478]
[0.25491796 0.25147843 0.49360361]
[0.26112259 0.27060338 0.46827403]
[0.25489605 0.2527938 0.49231015]
[0.25338399 0.2511074 0.49550861]
[0.25763865 0.26478723 0.47757412]
[0.25874505 0.26399882 0.47725613]
[0.26202039 0.28054754 0.45743207]
[0.25689411 0.26125225 0.48185364]
[0.25847882 0.26306971 0.47845147]
[0.25896399 0.26593033 0.47510569]
[0.25872766 0.26807118 0.47320115]
[0.26006886 0.27331819 0.46661295]
[0.25857335 0.26550435 0.4759223]
[0.25450954 0.2475826 0.49790787]
[0.25415871 0.24629313 0.49954815]
[0.26016185 0.2740003 0.46583785]
[0.25868872 0.26937156 0.47193972]
[0.25940652 0.26570154 0.47489194]
[0.25669156 0.25744881 0.48585963]
[0.25731545 0.26998042 0.47270413]
[0.25878541 0.26526135 0.47595324]
[0.25598845 0.25799501 0.48601654]
[0.26064528 0.30404531 0.43530941]
[0.25598005 0.26263065 0.4813893]
[0.25456976 0.25518385 0.49024638]
[0.25530903 0.25197053 0.49272045]
[0.25888504 0.27406027 0.46705468]
[0.2558957 0.2533321 0.49077219]
[0.25737836 0.26555377 0.47706787]
[0.25733528 0.25786027 0.48480445]
[0.25869862 0.26693701 0.47436438]
[0.28005543 0.30847088 0.4114737]
[0.27913763 0.2952931 0.42556926]
[0.28709844 0.31548588 0.39741568]
[0.31798773 0.35505382 0.32695845]
[0.29495779 0.32726332 0.37777889]
[0.30849975 0.32597576 0.36552449]
[0.28182932 0.29111907 0.42705162]
[0.29149139 0.32403279 0.38447581]
[0.28930266 0.32465204 0.3860453]
[0.3003143 0.31415838 0.38552732]
[0.319338 0.37207288 0.30858913]
[0.28362743 0.29848435 0.41788822]

[0.30446991 0.37159896 0.32393113]
[0.30210755 0.32252921 0.37536324]
[0.27457766 0.29335337 0.43206898]
[0.27860941 0.30519463 0.41619597]
[0.30259226 0.30802665 0.3893811]
[0.29191471 0.32641617 0.38166912]
[0.33119519 0.382739 0.28606581]
[0.2970758 0.33440068 0.36852353]
[0.3009137 0.29945356 0.39963273]
[0.28246453 0.31255706 0.40497841]
[0.33294121 0.36726394 0.29979486]
[0.30527521 0.33443714 0.36028765]
[0.28395287 0.31528712 0.40076001]
[0.28163061 0.31046015 0.40790924]
[0.29620176 0.33793425 0.36586399]
[0.29807625 0.32033896 0.38158479]
[0.29690664 0.31450353 0.38858984]
[0.28044639 0.31855679 0.40099683]
[0.2996774 0.33994495 0.36037765]
[0.29502141 0.33841405 0.36656454]
[0.28623572 0.31700327 0.39676102]
[0.34937942 0.35670723 0.29391334]
[0.30964014 0.30978697 0.38057288]
[0.27683153 0.28007565 0.44309282]
[0.28420504 0.30879352 0.40700145]
[0.31285192 0.37102087 0.31612721]
[0.28437933 0.29788273 0.41773794]
[0.30616101 0.33502243 0.35881656]
[0.3212623 0.34270691 0.33603079]
[0.29311509 0.31134416 0.39554076]
[0.29321925 0.32786861 0.37891214]
[0.29353432 0.33301552 0.37345015]
[0.30218217 0.32400327 0.37381456]
[0.28560257 0.30218532 0.41221212]
[0.29011516 0.3078597 0.40202514]
[0.28581692 0.31320365 0.40097943]
[0.27550196 0.30737502 0.41712302]
[0.29079923 0.31256084 0.39663993]
[0.38440712 0.33747621 0.27811668]
[0.37113391 0.35777487 0.27109122]
[0.34148247 0.34961041 0.30890712]
[0.36154159 0.35563336 0.28282505]
[0.37125642 0.3511574 0.27758619]
[0.36784756 0.37846298 0.25368946]
[0.38560634 0.36319982 0.25119385]
[0.36002237 0.38071378 0.25926385]
[0.38838184 0.40110001 0.21051815]
[0.31286887 0.30239221 0.38473892]
[0.30019841 0.30449604 0.39530555]


```
[0.35229228 0.35985877 0.28784895]
[0.32764795 0.33423674 0.33811531]
[0.38983413 0.37323538 0.2369305 ]
[0.37987986 0.34914024 0.27097991]
[0.32161556 0.31206696 0.36631748]
[0.33414274 0.33929511 0.32656215]
[0.3084605  0.30777523 0.38376427]
[0.42214514 0.42214514 0.15570972]
[0.37866449 0.40541665 0.21591886]
[0.32591941 0.32269056 0.35139004]
[0.36018684 0.34247456 0.2973386 ]
[0.3828584  0.40427367 0.21286792]
[0.32531802 0.34328769 0.33139429]
[0.32221764 0.31725513 0.36052724]
[0.32113652 0.33792471 0.34093877]
[0.31607977 0.33023585 0.35368438]
[0.31230143 0.31737226 0.37032631]
[0.37522955 0.36244406 0.26232639]
[0.31971884 0.35215175 0.32812941]
[0.35118721 0.38020083 0.26861197]
[0.28981537 0.30165057 0.40853406]
[0.37885561 0.36199063 0.25915376]
[0.32455412 0.34536204 0.33008383]
[0.38596391 0.39006101 0.22397508]
[0.33212175 0.35254341 0.31533484]
[0.33513832 0.31085009 0.35401159]
[0.33082532 0.33099265 0.33818203]
[0.31019841 0.31431056 0.37549103]
[0.31045429 0.32043452 0.3691112 ]
[0.33990006 0.32956604 0.3305339 ]
[0.29822527 0.30771111 0.39406363]
[0.37113391 0.35777487 0.27109122]
[0.34613825 0.33305633 0.32080543]
[0.33306963 0.31666747 0.3502629 ]
[0.31680978 0.32113841 0.3620518 ]
[0.3507111  0.3676207  0.2816682 ]
[0.31876383 0.32479814 0.35643803]
[0.32219436 0.30350821 0.37429743]
[0.33489629 0.3283136  0.33679011]]
```

```
# Exercise 6d)
```

```
# Compute accuracy of model using the accuracy function defined above
# with a few different recreations of the model. See how they change.
```

```
print(accuracy(predictions, targets))
```

```
0.19333333333333333
```

```
# Trying the same model again with different random numbers
```

```
# and comparing the accuracy score:
```

```
network_input_size = 4
```

```

layer_output_sizes = [8, 3]
activation_funcs = [sigmoid, softmax]
layers = create_layers_batch(network_input_size, layer_output_sizes)

predictions = feed_forward_batch(inputs, layers, activation_funcs)
print(accuracy(predictions, targets))

0.2933333333333333

# Third attempt with new random numbers:
network_input_size = 4
layer_output_sizes = [8, 3]
activation_funcs = [sigmoid, softmax]
layers = create_layers_batch(network_input_size, layer_output_sizes)

predictions = feed_forward_batch(inputs, layers, activation_funcs)
print(accuracy(predictions, targets))

0.12666666666666668

def cross_entropy(predict, target):
    return np.sum(-target * np.log(predict))

def cost(input, layers, activation_funcs, target):
    predict = feed_forward_batch(input, layers, activation_funcs)
    return cross_entropy(predict, target)

from autograd import grad

gradient_func = grad(
    cost, 1
) # Taking the gradient wrt. the second input to the cost function,
i.e. the layers

```

Exercise 7a)

The shape of the gradient wrt weights and biases should be the same as the matrices containing weights and biases. Later on we do addition with these matrices, indicating they must be the same size.

```

# Exercise 7b)
layers_grad = gradient_func(
    inputs, layers, activation_funcs, targets
) # Don't change this

# Exercise 7c)
def train_network(
    inputs, layers, activation_funcs, targets, learning_rate=0.001,
    epochs=100
):
    for i in range(epochs):

```

```

        layers_grad = gradient_func(inputs, layers, activation_funcs,
targets)
        for (W, b), (W_g, b_g) in zip(layers, layers_grad):
            W -= learning_rate * W_g
            b -= learning_rate * b_g

```

Exercise 7d)

The gradient method used above is plain fixed gradient descent.

```

# Exercise 7e)
# Training the network and seeing how the accuracy changes.
# Maybe with a plot?
inputs = iris.data
network_input_size = 4
layer_output_sizes = [8, 10, 6, 3]
layers = create_layers_batch(network_input_size, layer_output_sizes)
activation_funcs = [sigmoid, sigmoid, sigmoid, softmax]
targets = np.zeros((len(iris.data), 3))
for i, t in enumerate(iris.target):
    targets[i, t] = 1

train_network(inputs, layers, activation_funcs, targets, epochs=10000)

predictions = feed_forward_batch(inputs, layers, activation_funcs)
print(accuracy(predictions, targets))

0.9933333333333333

```

Exercise 7f)

How high of an accuracy is it possible to achieve with a neural network on this dataset, if we use the whole thing as training data?

It seems like we can get high accuracy since the dataset is small, but it will be overfit to the data and not generalize well. We asked ChatGPT for guidance, with the following prompt:

"How high of an accuracy is it possible to achieve with a neural network on the iris dataset of sklearn, if we use the whole thing as training data?"

GPT's answer:

Using the entire Iris dataset as training data can indeed provide a high accuracy with a neural network. The Iris dataset is relatively small and well-structured, and many machine learning models, including neural networks, can perform exceptionally well on this dataset.

Given that the Iris dataset consists of 150 samples with 4 features each, and it is a well-studied dataset, using all samples for training typically results in very high accuracy, close to 100%. This is because:

Small Dataset: The size of the dataset is small, which makes it easier for a model to memorize the data, especially if there is no separate test or validation set.

Clean and Well-Structured Data: The features in the Iris dataset are well-defined and do not have much noise, which allows algorithms to distinguish the classes easily.

To elaborate, many neural network architectures, even simple ones, can achieve near-perfect accuracy on the Iris dataset given enough training epochs and proper configurations (such as learning rate, activation functions, etc.). It is common to see accuracies of 99-100% when evaluating on the entire dataset used for training.

However, it's crucial to note that using the entire dataset for training and evaluation does not give a realistic measure of a model's generalization capability. In practice, you would typically split the dataset into a training set and a test set (or use cross-validation) to gauge how well your model is likely to perform on unseen data.

So, while it is possible to achieve around 100% accuracy on the Iris dataset when using the whole dataset for training, this is not indicative of the model's performance in a real-world scenario where it has to generalize to new data.