Perceptrons (single layer)

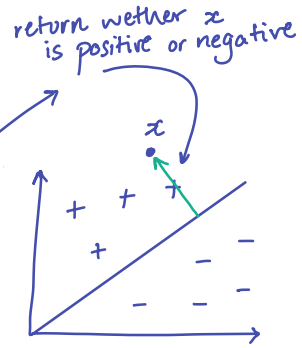
Perceptron is an algorithm that generates a linear classifier for supervised learning,
It, ideally, finds the optimal linear classifier for the data:

data = $\left\{\left(x^{(1)}, y^{(1)}\right), \ldots, \left(x^{(n)}, y^{(n)}\right)\right\}$

and $\theta \in \mathbb{R}^d, \theta_0 \in \mathbb{R}$
weight, offset/bias

Linear classifier: `h(x; theta, that0)`

$$h(x; \theta, \theta_0) = \text{sign}(\theta^T x + \theta_0) = \begin{cases} +1 & \text{if } \theta^T x + \theta_0 > 0 \\ -1 & \text{otherwise} \end{cases}$$



It may *not* find the optimal classifier if:
- the data is not linearly classifiable, eg ^^^
- there exists multiple linear classifiers (as soon as one is found, the weights stop adjusting)

The algorithm aims to find the hyperplane that separates the data by iteratively adjusting the weights of the hyperplane

The algorithm:

```
perceptron(D, T)          ( D={(x1,y1),...,(xn,yn)} - training data )
    θ = 0̄
    θ0 = 0
    for t = 1 to T
        for i = 1 to n        <- number of examples in data set
            if y(i)·(θᵀx(i) + θ0) ≤ 0        <- if this is positive, the prediction is correct → ignore
                θ = θ + y(i)x(i)             <- if it is negative adjust!
                θ0 = θ0 + y(i)
    return θ, θ0
```

(Loss function - only updates if values are wrongly classified)

Using numpy in Python:

```
import numpy as np
import math

###### useful functions #############################################
def positive(x, th, th0):
   return np.sign(np.dot(np.transpose(th), x) + th0)

def score(data, labels, ths, th0s):
   pos = np.sign(np.dot(np.transpose(ths), data) + np.transpose(th0s))
   return np.sum(pos == labels, axis = 1, keepdims = True)

###### actual perceptron ############################################

def perceptron(data, labels, params = {}):
    # if T not in params, default to 100
    T = params.get('T', 100)
    (d, n) = data.shape

    theta = np.zeros((d, 1)); theta_0 = np.zeros((1, 1))
    for t in range(T):
            for i in range(n):
                    x = data[:,i:i+1]
                    y = labels[:,i:i+1]
                    if y * positive(x, theta, theta_0) <= 0.0:
                            theta = theta + y * x
                            theta_0 = theta_0 + y
    return theta, theta_0

# Regular perceptron can be somewhat sensitive to the most recent examples
# that it sees. Instead, averaged perceptron produces a more stable output by
# outputting the average value of th and th0 across all iterations

def averaged_perceptron(data, labels, params = {}):
    # if T not in params, default to 100
    T = params.get('T', 100)
    (d, n) = data.shape

    theta = np.zeros((d, 1)); theta_0 = np.zeros((1, 1))
    thetas = np.zeros((d, 1)); theta_0s = np.zeros((1, 1))
    for t in range(T):
            for i in range(n):
                    x = data[:,i:i+1]
                    y = labels[:,i:i+1]
                    if y * positive(x, theta, theta_0) <= 0.0:
                            theta = theta + y * x
                            theta_0 = theta_0 + y
                    thetas = thetas + theta
                    theta_0s = theta_0s + theta_0
    return thetas/(n*T), theta_0s/(n*T)
```

```
###### useful functions #############################################
def positive(x, th, th0):
   return np.sign(np.dot(np.transpose(th), x) + th0)

def score(data, labels, ths, th0s):
   pos = np.sign(np.dot(np.transpose(ths), data) + np.transpose(th0s))
   return np.sum(pos == labels, axis = 1, keepdims = True)
```

(nicer colours)

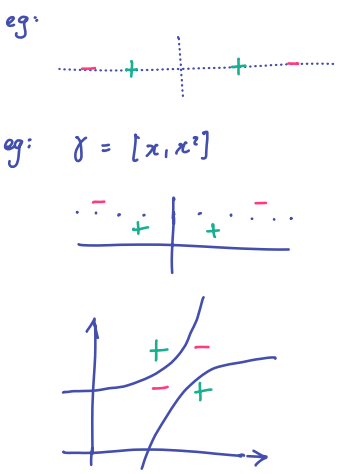
```python
###### actual perceptron ###################################################################################################
def perceptron(data, labels, params = {}):
    # if T not in params, default to 100
    T = params.get('T', 100)
    (d, n) = data.shape

    theta = np.zeros((d, 1)); theta_0 = np.zeros((1, 1))
    for t in range(T):
        for i in range(n):
            x = data[:,i:i+1]
            y = labels[:,i:i+1]
            if y * positive(x, theta, theta_0) <= 0.0:
                theta = theta + y * x
                theta_0 = theta_0 + y
    return theta, theta_0

# Regular perceptron can be somewhat sensitive to the most recent examples that it sees. Instead, averaged perceptron produces
# a more stable output by outputting the average value of th and th0 across all iterations
def averaged_perceptron(data, labels, params = {}):
    # if T not in params, default to 100
    T = params.get('T', 100)
    (d, n) = data.shape

    theta = np.zeros((d, 1)); theta_0 = np.zeros((1, 1))
    thetas = np.zeros((d, 1)); theta_0s = np.zeros((1, 1))
    for t in range(T):
        for i in range(n):
            x = data[:,i:i+1]
            y = labels[:,i:i+1]
            if y * positive(x, theta, theta_0) <= 0.0:
                theta = theta + y * x
                theta_0 = theta_0 + y
            thetas = thetas + theta
            theta_0s = theta_0s + theta_0
    return thetas/(n*T), theta_0s/(n*T)
```

Idea: if the data is not linearly classifiable, we can try to change basis and check if it is now classifiable.



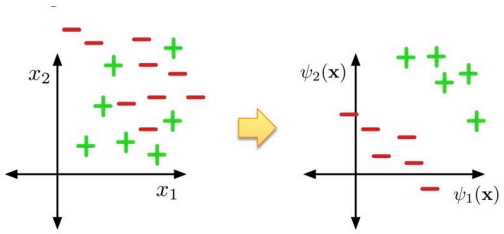
Polynomial basis:

Here is a table illustrating the kth order polynomial basis for different values of k.

Order	$d = 1$	in general
0	$[1]$	$[1]$
1	$[1, x]$	$[1, x_1, \ldots, x_d]$
2	$[1, x, x^2]$	$[1, x_1, \ldots, x_d, x_1^2, x_1x_2, \ldots]$
3	$[1, x, x^2, x^3]$	$[1, x_1, \ldots, x_1^2, x_1x_2, \ldots, x_1x_2x_3, \ldots]$

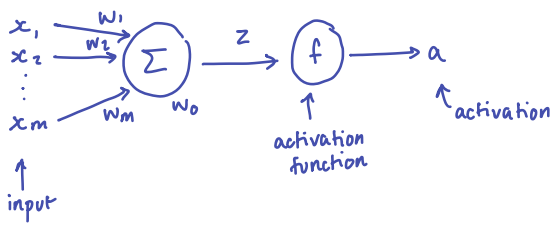
Multi-layer Perceptron

Idea: take the single-layer perceptron and add some 'hidden' layers that will make the data linearly classifiable



MLP from scratch:

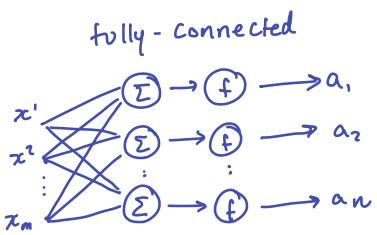
Single neuron:



$$X = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix} \in \mathbb{R}^m$$

$$a \in \mathbb{R}$$
$$a = f(z) = f\left(\left(\sum_{j=1}^{m} x_j w_j\right) + w_0\right)$$

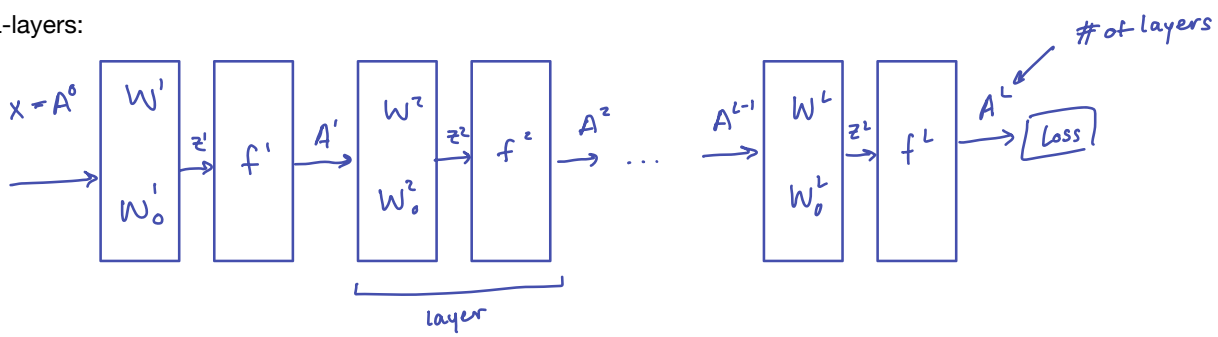
Single layer:



$W : m \times n$ matrix of vetors
$W_0 : n \times 1$ vector
$A = f(z) = f(W^T X + W_0)$

(f is applied element-wise)

L-layers:



$A^L = W^{L\,T} A^{L-1}$

$W^{L-1\,T} A^{L-2}$

$\vdots$

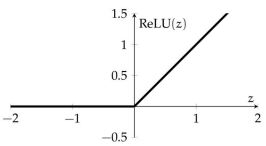
$W^{1\,T} X$

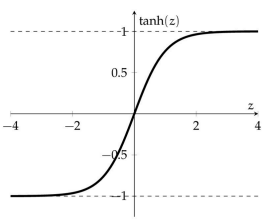
$$A^L = W^{L^T} W^{L-1^T} \ldots W^1 X$$

$$A^L = W^{total} X$$

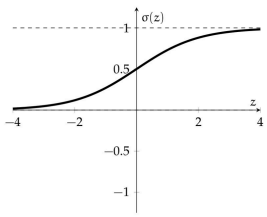
Note: the layers are only useful if f is non-linear
(if f is linear, Wtotal is simply a linear transformation)

Possible f functions:





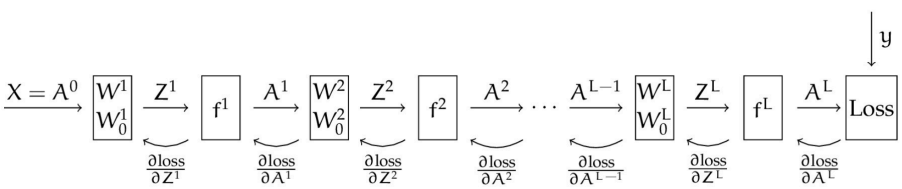
"Rule of thumb:
Start with ReLU activation. If necessary, try tanh."
(University of Toronto)



The idea now is to:

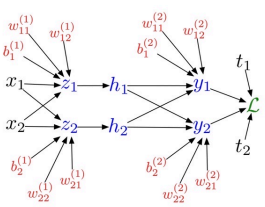
- forward pass, calculating the final output based on the original inputs and all the weights+activations
- compare to the actual output and calculate loss
- back-propagate through the layers, adjusting weights according to how much they are to "blame" for the loss

Error back-propagation:
Work backward and compute the gradient of the loss with respect to the weights in each layer



Another representation (notation is slightly different)

Backpropagation for a MLP (Vectorized)



Forward pass:

$$\mathbf{z} = W^{(1)}\mathbf{x} + \mathbf{b}^{(1)}$$
$$\mathbf{h} = \sigma(\mathbf{z})$$
$$\mathbf{y} = W^{(2)}\mathbf{h} + \mathbf{b}^{(2)}$$
$$\mathcal{L} = \frac{1}{2}||\mathbf{y} - \mathbf{t}||^2$$

Backward pass:

$$\overline{\mathcal{L}} = 1$$
$$\overline{\mathbf{y}} = \overline{\mathcal{L}}(\mathbf{y} - \mathbf{t})$$
$$\overline{W^{(2)}} = \overline{\mathbf{y}}\mathbf{h}^T$$
$$\overline{\mathbf{b}^{(2)}} = \overline{\mathbf{y}}$$
$$\overline{\mathbf{h}} = W^{(2)\,T}\overline{y}$$
$$\overline{\mathbf{z}} = \overline{\mathbf{h}} \circ \sigma'(\mathbf{z})$$
$$\overline{W^{(1)}} = \overline{\mathbf{z}}\mathbf{x}^T$$
$$\overline{\mathbf{b}^{(1)}} = \overline{\mathbf{z}}$$

Full algorithm:

SGD-NEURAL-NET($\mathcal{D}_n$, T, L, $(m^1, \ldots, m^L)$, $(f^1, \ldots, f^L)$)

for $l = 1$ **to** L
  $W^l_{ij} \sim$ Gaussian$(0, 1/m^l)$
  $W^l_{0j} \sim$ Gaussian$(0, 1)$
for $t = 1$ **to** T
  i = random sample from $\{1, \ldots, n\}$
  $A^0 = x^{(i)}$
  // forward pass to compute the output A^L
  for $l = 1$ **to** L
    $Z^l = W^{l\,T} A^{l-1} + W^l_0$
    $A^l = f^l(Z^l)$
  loss = Loss$(A^L, y^{(i)})$
  for $l = L$ **to** 1:
    // error back-propagation
    $\partial loss/\partial A^l$ = **if** $l < L$ **then** $\partial loss/\partial Z^{l+1} \cdot \partial Z^{l+1}/\partial A^l$ **else** $\partial loss/\partial A^L$
    $\partial loss/\partial Z^l = \partial loss/\partial A^l \cdot \partial A^l/\partial Z^l$
    // compute gradient with respect to weights
    $\partial loss/\partial W^l = \partial loss/\partial Z^l \cdot \partial Z^l/\partial W^l$
    $\partial loss/\partial W^l_0 = \partial loss/\partial Z^l \cdot \partial Z^l/\partial W^l_0$
    // stochastic gradient descent update
    $W^l = W^l - \eta(t) \cdot \partial loss/\partial W^l$
    $W^l_0 = W^l_0 - \eta(t) \cdot \partial loss/\partial W^l_0$

Note on initialisation:

- Weights should be chosen randomly -> if not, the symmetry between layers makes them less useful.
- Weights should not be too big -> the activation functions are only useful near zero, so gradient descent will not signal a useful direction to go if weights are too big.

Possible strategy
Choose each weight at random from a Gaussian (normal) distribution with mean 0 and standard deviation $(1/m)$ where m is the number of inputs to the unit.

Note on last-layer activation:
The activation function for the last layer of the network may be different from the one used in previous layers in order to make the data match the desired output type, eg:

last-layer function		other layers	purpose
$f(x) = x$	w/	squared loss	→ regression
sigmoid (interpret as probability)	w/	neg. log likelyhood	→ multi-class classification

Using PyTorch in Python:

```
from collections import OrderedDict
import torch
import torch.nn as nn
import torch
import torchvision
import torchvision.transforms as transforms
import ssl
ssl._create_default_https_context = ssl._create_unverified_context

# Load the MNIST dataset
transform = transforms.Compose(
    [transforms.ToTensor(),
    transforms.Lambda(lambda x: x.view(-1))])  # Flatten images

train_dataset = torchvision.datasets.MNIST(root='./data', train=True,
transform=transform, download=True)
test_dataset = torchvision.datasets.MNIST(root='./data', train=False,
transform=transform, download=True)

train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=64,
shuffle=True)
test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=64,
shuffle=False)
```

```
model = nn.Sequential(OrderedDict([
    ('layer1', nn.Linear(784, 100)),
    ('activation1', nn.ReLU()),
    ('layer2', nn.Linear(100, 50)),
    ('activation2', nn.ReLU()),
    ('output', nn.Linear(50, 10)),
    ('outActivation', nn.Sigmoid()),
]))

loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

X, y = next(iter(train_loader))


# Training loop
num_epochs = 3
for epoch in range(num_epochs):
    model.train()
    total_loss = 0
    correct = 0
    total = 0

    for X, y in train_loader:
        y_pred = model(X)
        loss = loss_fn(y_pred, y)

        optimizer.zero_grad()
        loss.backward()         # apply backpropegation
        optimizer.step()

        total_loss += loss.item()

        # Compute accuracy
        _, predicted = torch.max(y_pred, 1)  # Get the class with the highest score
        correct += (predicted == y).sum().item()
        total += y.size(0)

    train_accuracy = correct / total * 100
    avg_loss = total_loss / len(train_loader)
    print(f"Epoch [{epoch+1}/{num_epochs}], Loss: {avg_loss:.4f}, Accuracy:
{train_accuracy:.2f}%")

# Evaluate on test set
model.eval()
correct = 0
total = 0
with torch.no_grad():
    for X, y in test_loader:
        y_pred = model(X)
        _, predicted = torch.max(y_pred, 1)
        correct += (predicted == y).sum().item()
        total += y.size(0)

test_accuracy = correct / total * 100
print(f"Final Test Accuracy: {test_accuracy:.2f}%")
```

```python
# Load the MNIST dataset
transform = transforms.Compose([transforms.ToTensor(), transforms.Lambda(lambda x: x.view(-1))])  # Flatten images

train_dataset = torchvision.datasets.MNIST(root='./data', train=True, transform=transform, download=True)
test_dataset = torchvision.datasets.MNIST(root='./data', train=False, transform=transform, download=True)

train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=64, shuffle=True)
test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=64, shuffle=False)

model = nn.Sequential(OrderedDict([
    ('layer1', nn.Linear(784, 100)),
    ('activation1', nn.ReLU()),
    ('layer2', nn.Linear(100, 50)),
    ('activation2', nn.ReLU()),
    ('output', nn.Linear(50, 10)),
    ('outActivation', nn.Sigmoid()),
]))

loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

X, y = next(iter(train_loader))
# Training loop
num_epochs = 3
for epoch in range(num_epochs):
    model.train()
    total_loss = 0
    correct = 0
    total = 0

    for X, y in train_loader:
        y_pred = model(X)
        loss = loss_fn(y_pred, y)

        optimizer.zero_grad()
        loss.backward()         # apply backpropegation
        optimizer.step()

        total_loss += loss.item()

        # Compute accuracy
        _, predicted = torch.max(y_pred, 1)  # Get the class with the highest score
        correct += (predicted == y).sum().item()
        total += y.size(0)

    train_accuracy = correct / total * 100
    avg_loss = total_loss / len(train_loader)
    print(f"Epoch [{epoch+1}/{num_epochs}], Loss: {avg_loss:.4f}, Accuracy: {train_accuracy:.2f}%")
# Evaluate on test set
model.eval()
correct = 0
total = 0
with torch.no_grad():
    for X, y in test_loader:
        y_pred = model(X)
        _, predicted = torch.max(y_pred, 1)
        correct += (predicted == y).sum().item()
        total += y.size(0)

test_accuracy = correct / total * 100
print(f"Final Test Accuracy: {test_accuracy:.2f}%")

torch.save(model.state_dict(), "my_model.pickle")
```