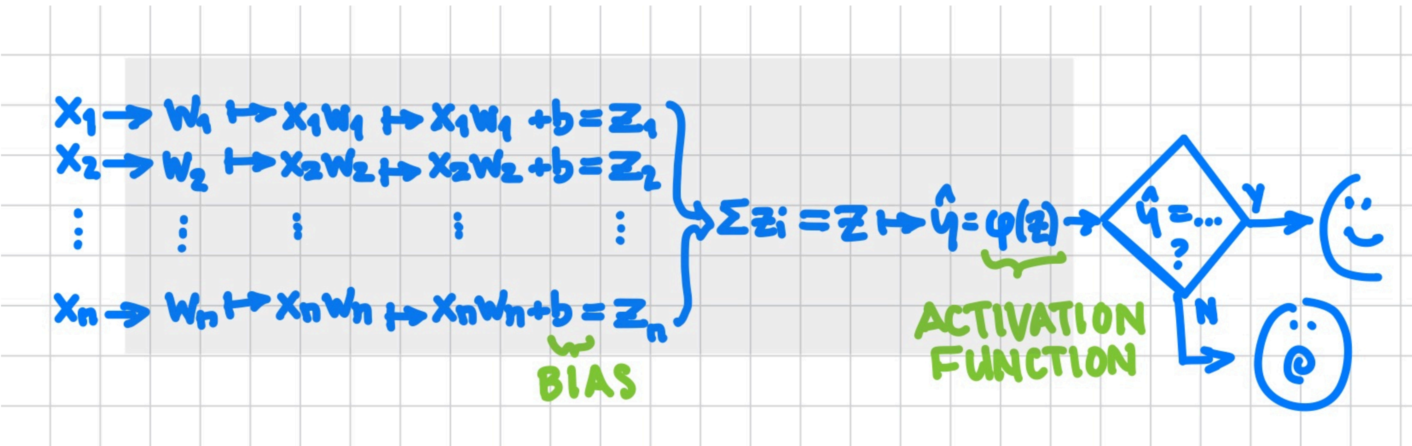
nmi | spring 2024

special lecture 04 : neural networks, briefly

1 perceptron

1.1 review



code, perceptron

```
import numpy as np

class Perceptron:

  def __init__(self,learning_rate=0.01,epochs=1000,weights=None,bias=None, \
             tol=1e-8,activation_func="step",delta_func=None,leaky=0.1):
    self.learning_rate = learning_rate
    self.epochs = epochs
    self.weights = weights
    self.bias = bias
    self.tol = tol
    self.delta_func = delta_func
    self.lr_leaky = leaky

    if activation_func is None:…
    if isinstance(activation_func,str):…
    else:…

  def copy(self):…

  def fit(self,X,y,display=False):
    n_samples,n_features = X.shape

    # init parms
    if self.weights is None or self.bias is None:…

    # learn
    rmse_old = self.tol*2
    for i in range(self.epochs):
      errors = 0
```

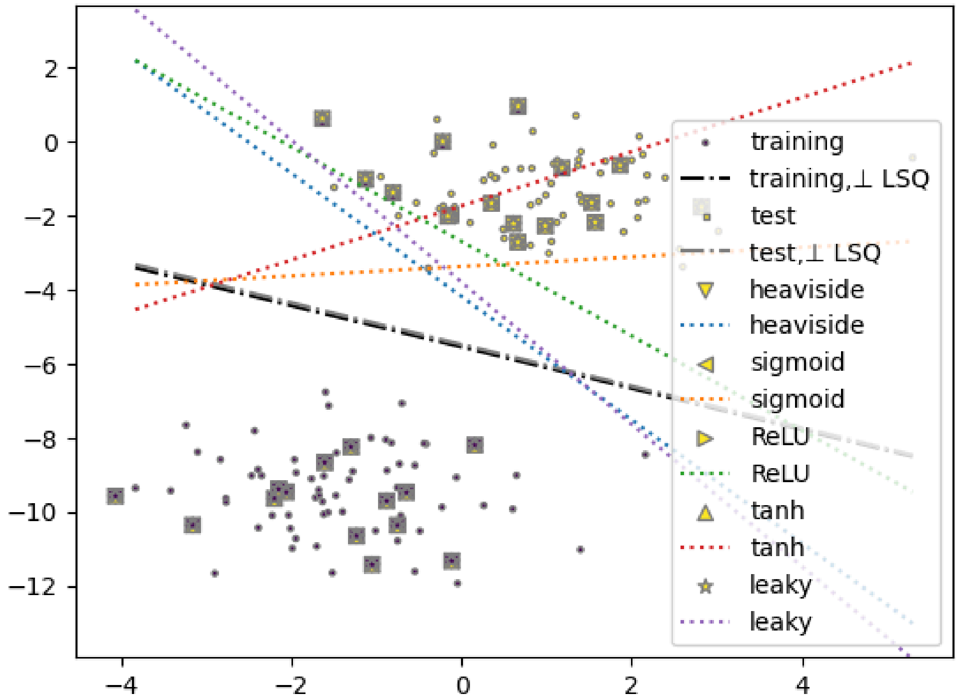
```
      for x,target in zip(X,y):
        z = np.dot(x,self.weights) + self.bias # linear_output
        y_predicted = self.activation_func(z)

        # update weights, bias
        error = y_predicted - target
        self.bias -= self.learning_rate*error
        delta = self.delta_func(x,z)
        self.weights -= self.learning_rate*error*delta*x

        errors += pow(error,2)
      rmse = np.sqrt(errors/len(y))
      if (rmse < self.tol) or (rmse == rmse_old) or (i % 100 == 0):⋯
      rmse_old = rmse

  def predict(self,X):
    linear_output = np.dot(X,self.weights) + self.bias
    y_predicted = self.activation_func(linear_output)
    return y_predicted

  # functions: activation, delta
  def leaky(self,x,alpha=None):⋯
  def dleaky(self,x,z,alpha=None):⋯
  def ReLU(self,x):⋯
  def dReLU(self,x,z):⋯
  def sigmoid(self,x):⋯
  def dsigmoid(self,x,z):⋯
  def step(self,x):⋯
  def dstep(self,x,z):⋯
  def tanh(self,x):⋯
  def dtanh(self,x,z):⋯

  # misc
  def sign(self,x):⋯

```

```
if __name__ == "__main__":

  # imports
  import matplotlib.pyplot as plt
  import numpy as np
  import scipy as sp
  from sklearn.model_selection import train_test_split
  from sklearn import datasets
  import statistics as st
  from tabulate import tabulate

  def accuracy(y_pred,y_true,tol=0):
    if tol == 0:⋯
    else:
      accuracy = np.sum(abs(y_pred-y_true)<=tol) / len(y_true)
    return accuracy

  # init runtime
  debug = False

  # datasets from sklearn
  X,y = datasets.make_blobs(n_samples=150,n_features=2,centers=2,cluster_std=1.05,random_state=2)
  X_train,X_test,y_train,y_test = train_test_split(X,y,test_size=0.2,random_state=123)
  x0 = np.array([np.amin(X_train[:,0]),np.amax(X_train[:,0])]) # interval, training data

  # plot, init
  fig = plt.figure()
  ax = fig.add_subplot(1,1,1)

  if True: # plot data, training⋯

  # weights, bias for everyone
  n_features = X.shape[1]
  if False: # suggested weights when random⋯
  else: # suggested for ReLU⋯
  weights = np.random.uniform(urng_from,urng_thru,n_features)
  bias = np.random.uniform(urng_from,urng_thru)
```

```
  # init nn
  ss_afname = ["heaviside","sigmoid","ReLU","tanh","leaky"]
  ss_marker = ["v","<",">","^","*"] # markers for plot
  data = [] # array for tabulate

  # train and test per activation function
  for s_afname,s_marker in zip(ss_afname,ss_marker):
    p = Perceptron(learning_rate=0.005,epochs=1000,weights=weights.copy(),bias=bias,activation_func=s_afname)
    if debug: # spacer⋯
    p.fit(X_train,y_train,display=debug)

    # prediction, classification
    predictions = p.predict(X_test)
    if debug: # accuracy wrt test data⋯
    if debug: # accuracy wrt training data⋯

    # plot, predictions
    plt.scatter(X_test[:,0],X_test[:,1],label=s_afname,marker=s_marker,c=predictions,edgecolor="grey")

    # plot, linear separation
    x1 = -(p.weights[0]*x0+p.bias)/p.weights[1]
    ax.plot(x0,x1,label=s_afname,ls=":") # hyperplane ~ decision boundary

    # tabulate weights, bias
    s_weights = ",".join(f"{weight:.8f}" for weight in p.weights)
    s_accuracy = f"{accuracy(predictions,y_test,p.tol)*100:3.0f}"
    if debug:⋯
    else:⋯

  # tabulate, show
  print()
  if debug:⋯
  else:⋯

  # plot, show
  ymin = np.amin(X_train[:,1])
  ymax = np.amax(X_train[:,1])
  ax.set_ylim([ymin-2,ymax+3])
  plt.title("\ndont look too close\n")
  plt.legend(loc="lower right")
  plt.show()
```

af	w	b	test(%)
heaviside	0.04536806,0.02725385	0.11406480	100
sigmoid	-0.01231378,0.09611858	0.32406480	100
ReLU	0.12265729,0.09611858	0.25906480	100
tanh	0.19444766,-0.26616686	-0.46093520	33
leaky	0.09304901,0.04847492	0.18406480	100

dont look too close



1.2 predict()

forward propagation:

1. application of weights;
2. summation;
3. activation.

1.2.1 activation functions, revisited

common activation functions

- linear. yes, thats right; it doesnt add a thing. think of it as a placeholder.

- heaviside: binary step. for beginners = my first activation function.

$$f(x) = \begin{cases} 0 & x < 0 \\ 1 & x \geq 0 \end{cases}$$

- sigmoid: input scaled $x \mapsto [0, 1]$. used for binary classification where output can be interpreted as a probability. a popular flavor.

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

- hyperbolic tangent, tanh: input scaled $x \mapsto [-1, 1]$ centered around zero. used in hidden layers to balance input signal.

$$f(x) = tanh(x) = \frac{2}{1 + e^{-2x}} - 1$$

- rectified linear unit (ReLU): blocks negative input. simple trick as an option when training deep neural networks. most popular for hidden layers, so they say.

$$f(x) = max(0, x)$$

- leaky ReLU: allows very small, non-zero gradient when negative input. keeps neurons alive when not actively firing.

$$f(x) = max(\alpha x, x), \quad \alpha \ll 1$$

- exponential linear unit (ELU): smooths negative inputs. may help maintain mean activation close to zero; an alternative for ReLU.

$$f(x) = \begin{cases} x & x > 0 \\ \alpha(e^x - 1) & x \leq 0 \end{cases}$$

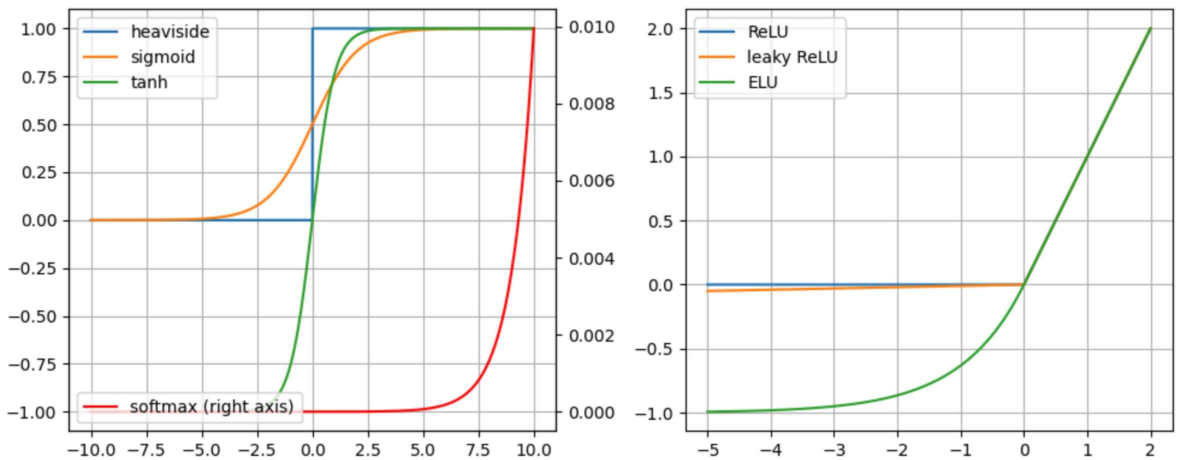
- normalized exponential (softmax): turns logits into probabilities. popular for multi-class classification, picking the most likely one.

$$P(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

visual, comparative

```
import matplotlib.pyplot as plt
import numpy as np

if __name__ == "__main__":

```

some activation functions



choice per location

- output layer: sigmoid, softmax
- hidden layers: ReLU, leaky ReLU, tanh.

1.2.2 wrt bias, threshold

consider the heaviside activation function,

$$\hat{y} = \begin{cases} 1 & \text{if } \sum_1^n x_i\,w_i \geq \theta \\ 0 & \text{if } \sum_1^n x_i\,w_i < \theta \end{cases} = \begin{cases} 1 & \text{if } \sum_1^n x_i\,w_i - \theta \geq 0 \\ 0 & \text{if } \sum_1^n x_i\,w_i - \theta < 0 \end{cases} = \begin{cases} 1 & \text{if } \sum_0^n x_i\,w_i \geq \theta \\ 0 & \text{if } \sum_0^n x_i\,w_i < \theta \end{cases} \quad \text{where } x_0 = 1, w$$

that makes threshold vs bias look somewhat ambiguous; however, threshold is the decision boundary for the activation function and bias is an offset to the weighted sum of inputs.

1.3 fit()

backpropagation ~ "backward propagation of errors":

1. forward pass to generate output;
2. backward pass where error informs changes in weights and bias.

the gradient of the loss function expressed with the chain rule of calculus reveals how much error each weight contributes to the error, providing a path for its adjustment.

gradient descent of the loss function wrt weights:

$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial \phi} \cdot \frac{\partial \phi}{\partial z} \cdot \frac{\partial z}{\partial w}$$

where L is the loss function, ϕ is the activation function, z is the weighted, biased input and w, b are the weights and bias.

weights are moved in the direction of steepest decrease in loss and with magnitude adjusted by learning rate hyperparameter η. ie,

$$w_{new} = w_{old} - \eta \cdot \frac{\partial L}{\partial w}.$$

similarly for bias,

$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial \phi} \cdot \frac{\partial \phi}{\partial z} \cdot \frac{\partial z}{\partial b}$$

$$b_{new} = b_{old} - \eta \cdot \frac{\partial L}{\partial b}.$$

this repeats until model converges to a state where the loss is minimized and/or some other criteria are met.

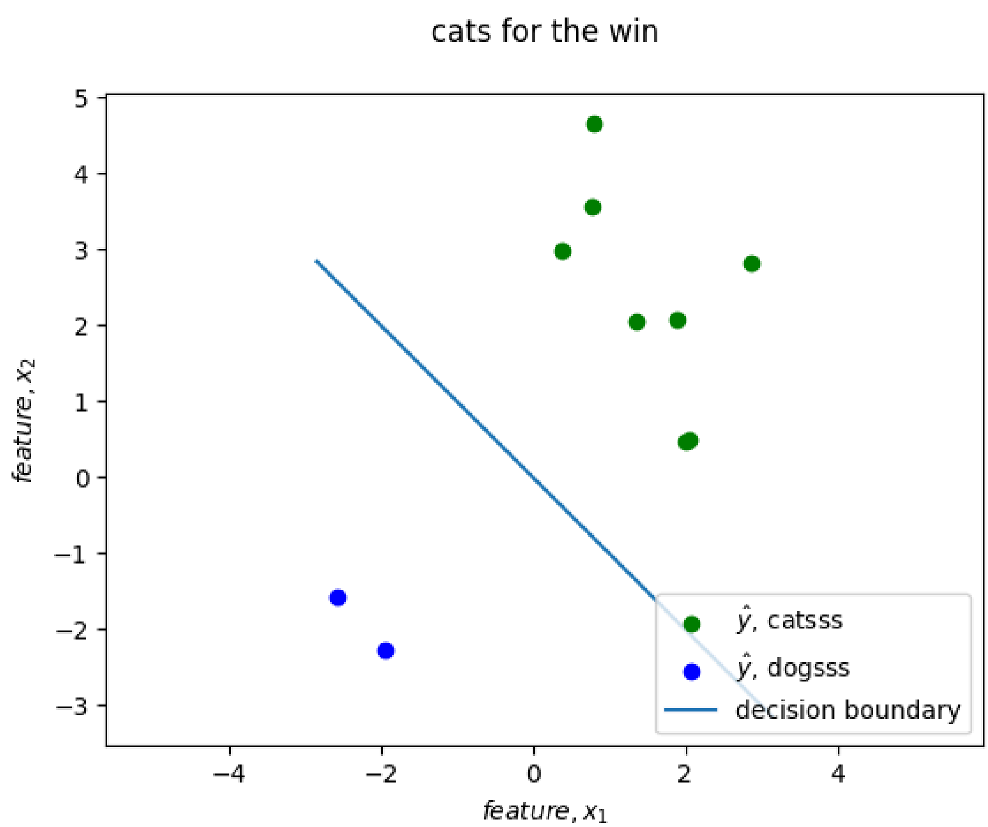
1.3 linear separability

to reiterate, data sets are linearly separable if you can put a hyperplane between them.

hyperplane, visual

code, python

```
if __name__ == "__main__":⋯
```



perceptron convergence theorem

for any set of **linearly separable** labeled examples, the perceptron learning algorithm will halt after a **finite number of iterations**. ie, after a finite number of iterations, the algorithm **finds (not optimizes)** a vector that classifies perfectly all the examples. (mostly [mit])

und so weiter

- **pure math** thats allowed near neural nets: linear algebra, basic abstract algebra, set theory, functional analysis. so they say.

theorems

- perceptron convergence theorem: rosenblatt [@princeton]; novikoff [@oregonstate]
- perceptron capacity: cover counting theorem [@mit]

bookshelf

- mit press, deep learning (textbook) [online]