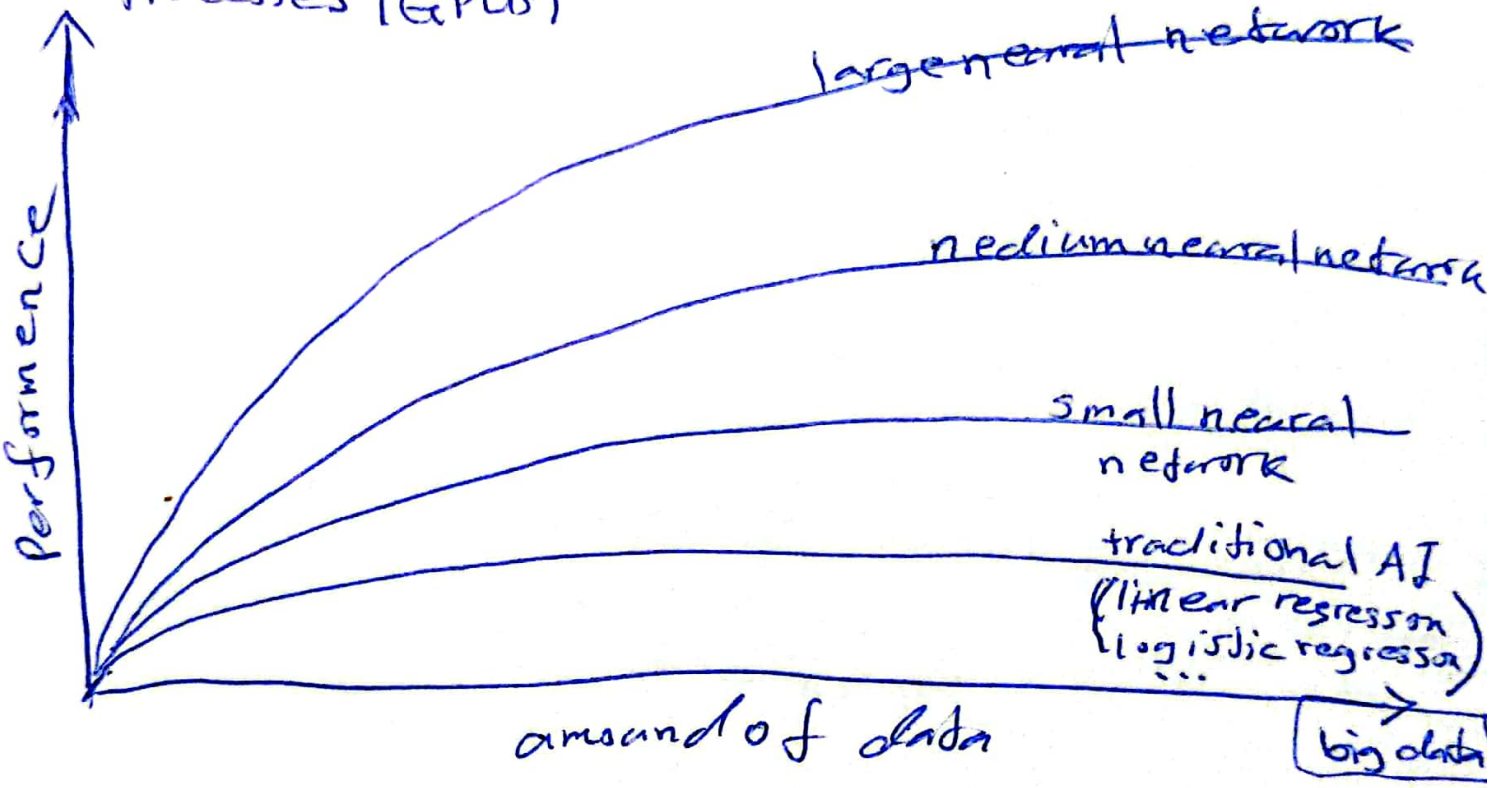
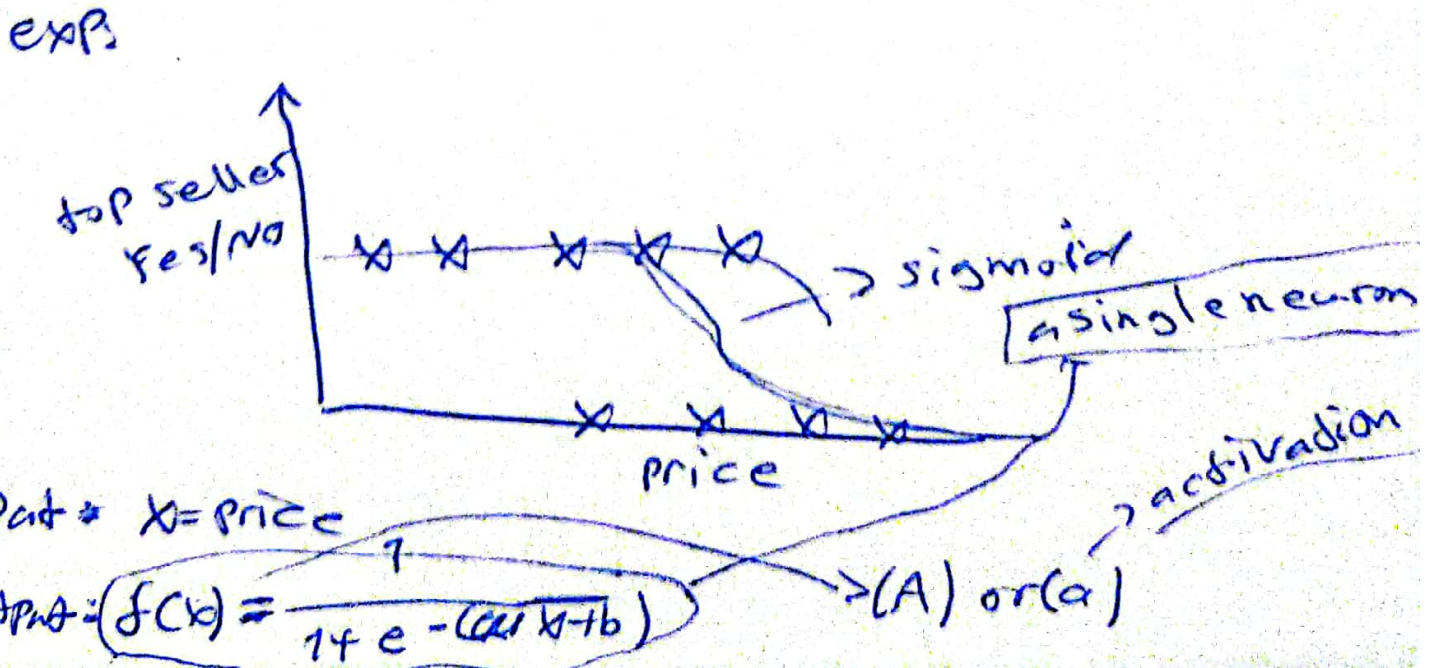
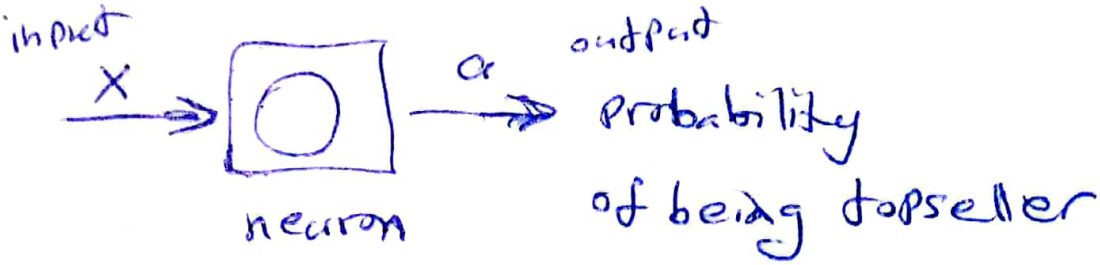
"Advanced learning Algorithms"

neural networks (deep learning algorithms)

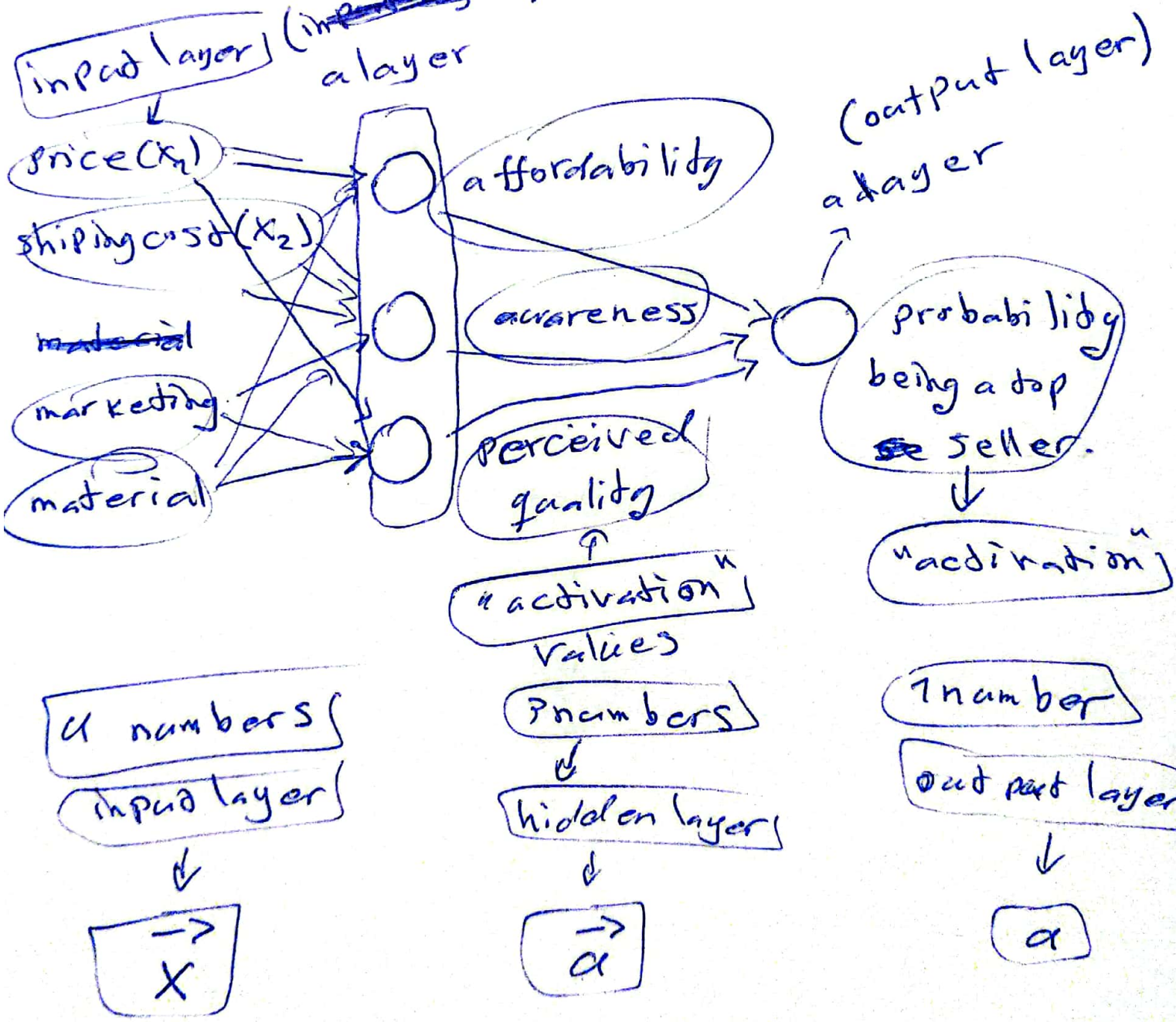


Demand Prediction

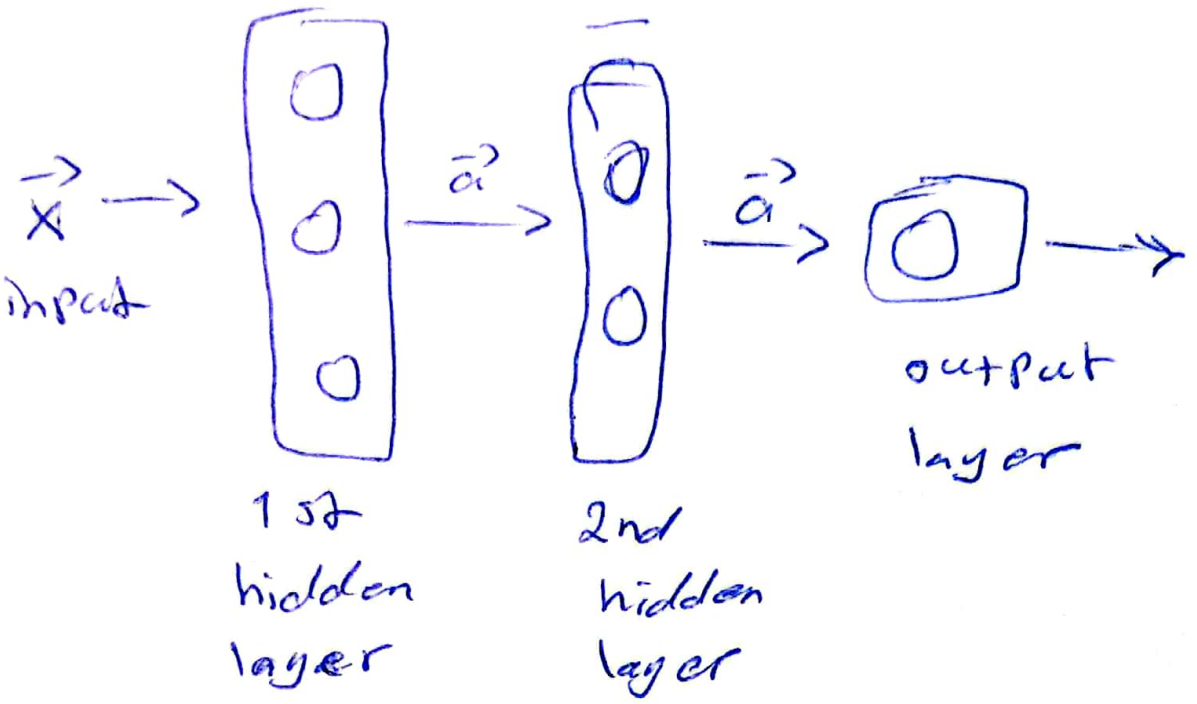




Exp: features (x_1, x_2, x_3, x_4)



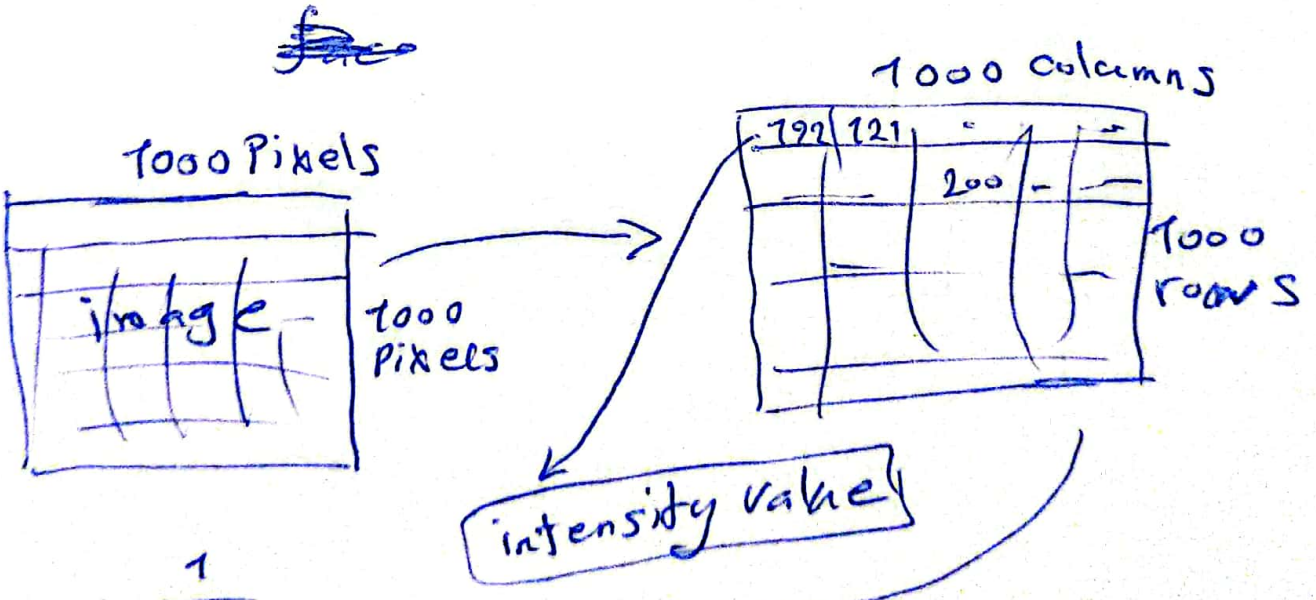
(3) a multi layer perceptron (neural network)

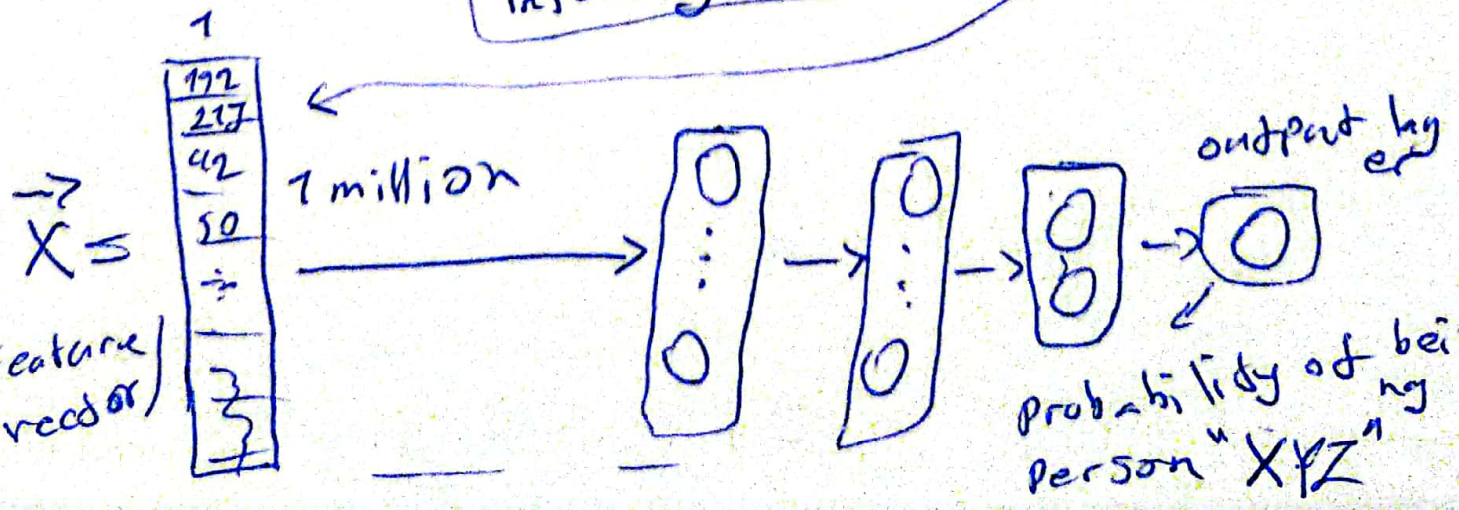


~~a multi~~

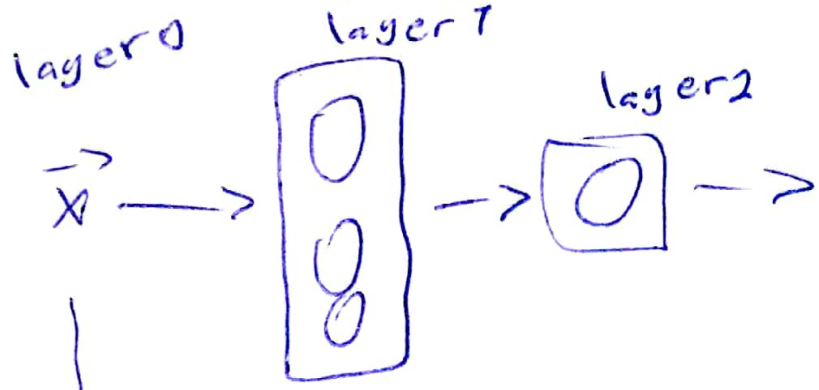
Recognizing images

~~face~~

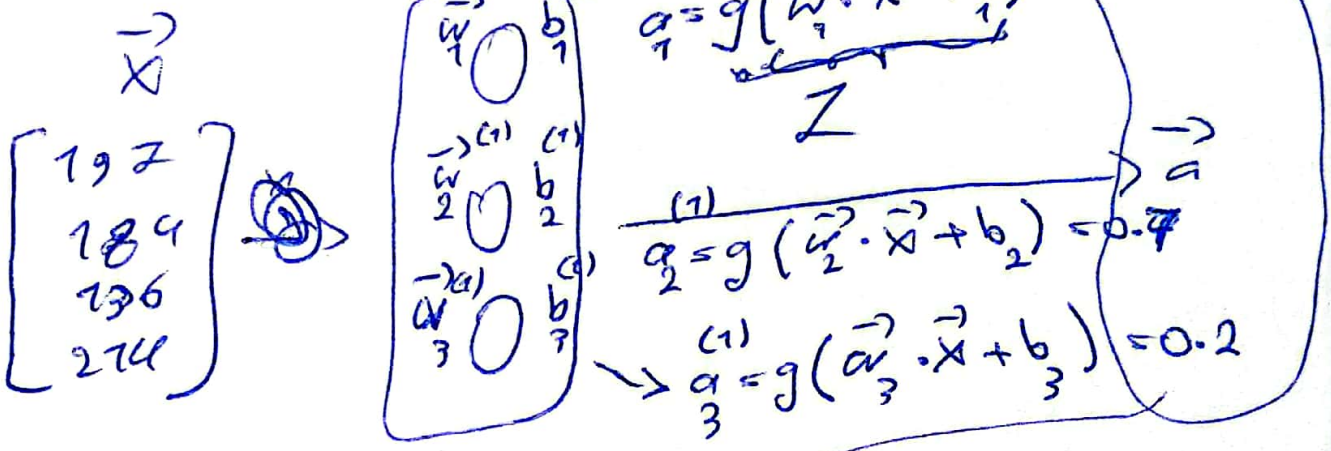




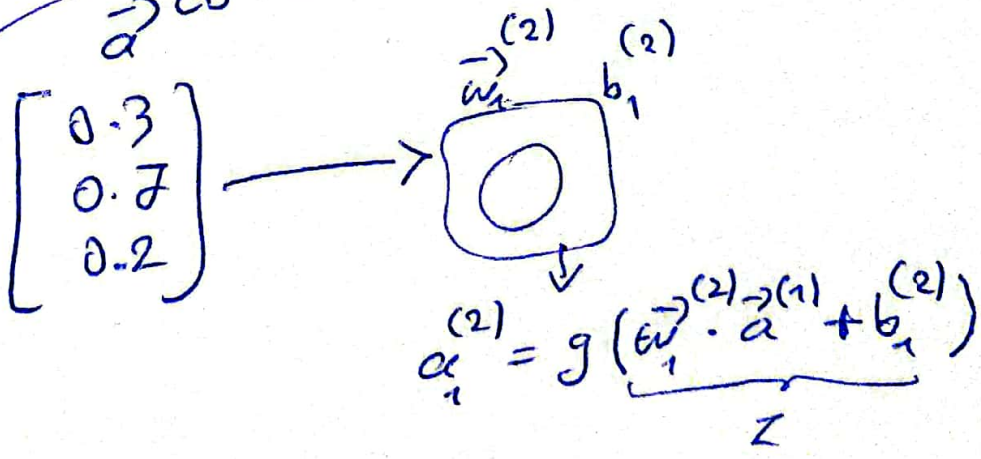
neural network layer



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



$\vec{a}^{(1)}$ → output of layer 1



$= 0.84$ output

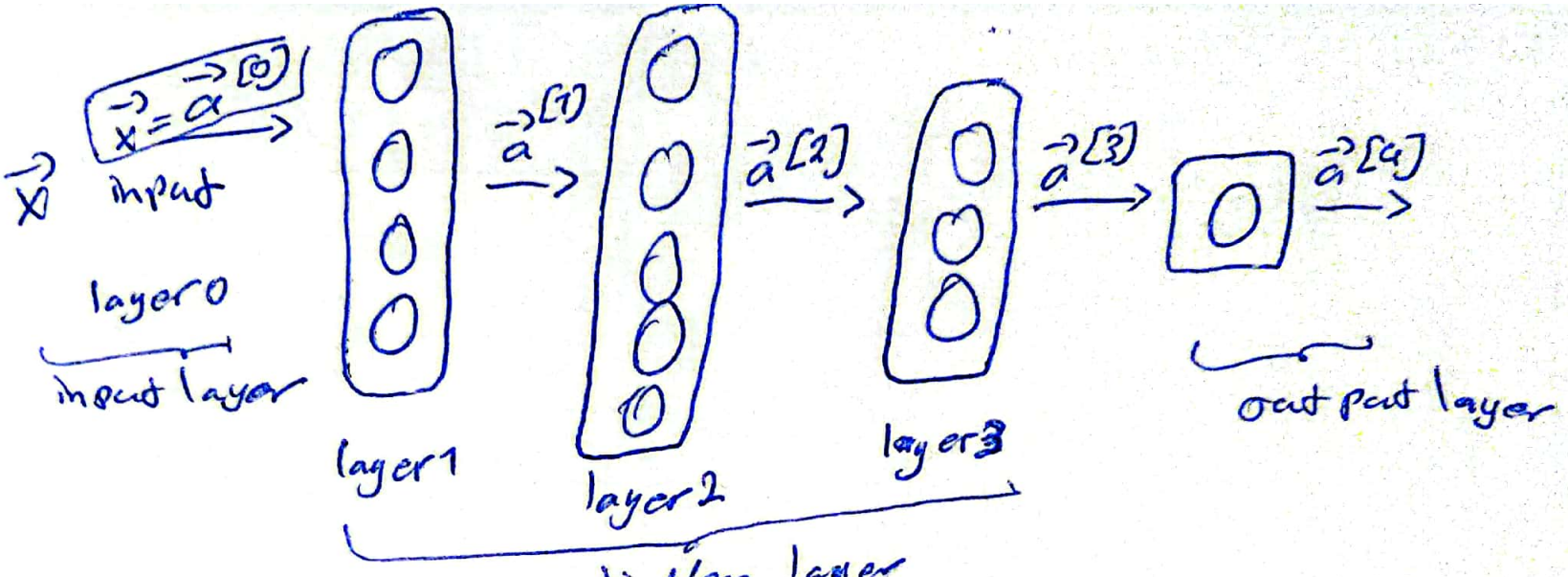
(optional)

is $a^{(2)} \geq 0.5$?

yes → $\hat{y} = 1$

no → $\hat{y} = 0$

(4) EXP



~~in~~ ..in counting the layer the input layer (layer 0) is not counted so this model has

4 layers

layer 3

$$\begin{bmatrix} a_1^{[3]} = g(\vec{w}_1^{[3]} \cdot \vec{a}^{[2]} + b_1^{[3]}) \\ a_2^{[3]} = g(\vec{w}_2^{[3]} \cdot \vec{a}^{[2]} + b_2^{[3]}) \\ a_3^{[3]} = g(\vec{w}_3^{[3]} \cdot \vec{a}^{[2]} + b_3^{[3]}) \end{bmatrix}$$

(sigmoid or activation function)

$$g(z) = \frac{1}{1 + e^{-(\vec{w} \cdot \vec{x} + b)}}$$

$$\vec{a}^{[3]} = \begin{bmatrix} a_1^{[3]} \\ a_2^{[3]} \\ a_3^{[3]} \end{bmatrix}$$

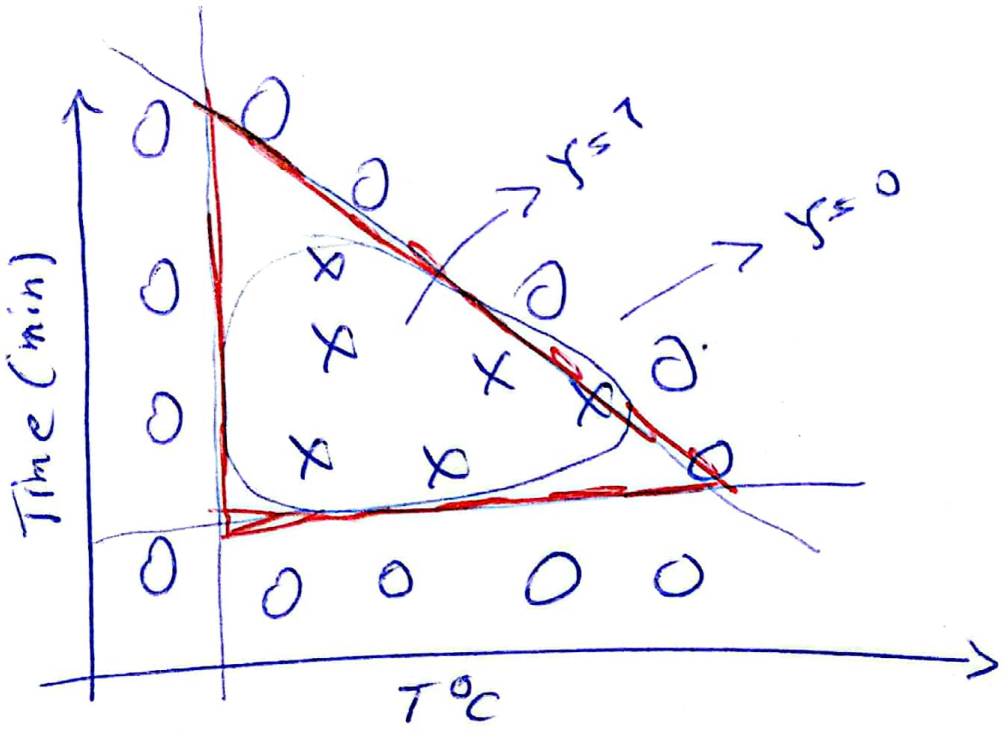
the generalized form

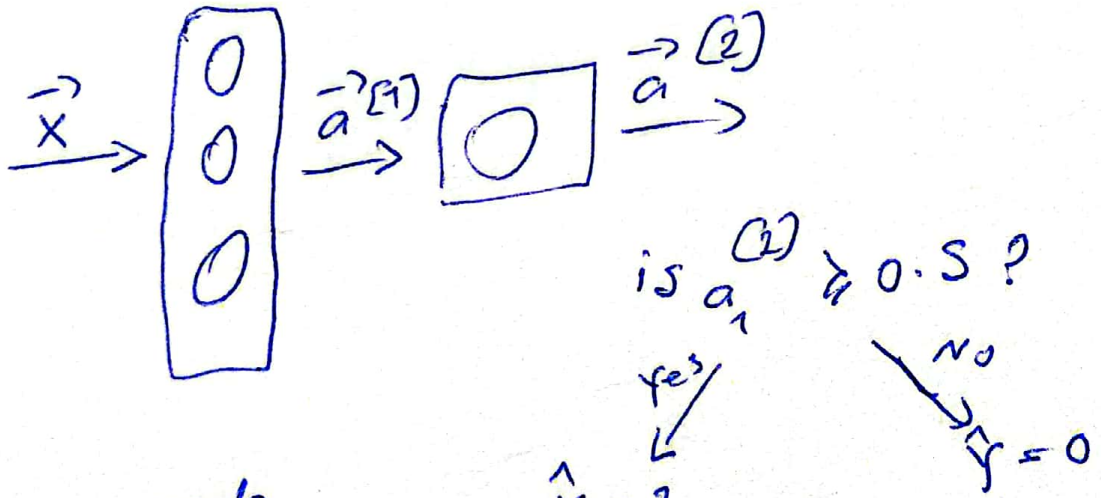
$$a_j^{[l]} = g(\vec{w}_j^{[l]} \cdot \vec{a}^{[l-1]} + b_j^{[l]})$$

"TensorFlow"

EXP

coffee roasting





Tensor code

```
x = np.array([[200, 17]])
layer_1 = Dense(units=3, activation="sigmoid")
a1 = layer_1(x)
```

↓ next Page

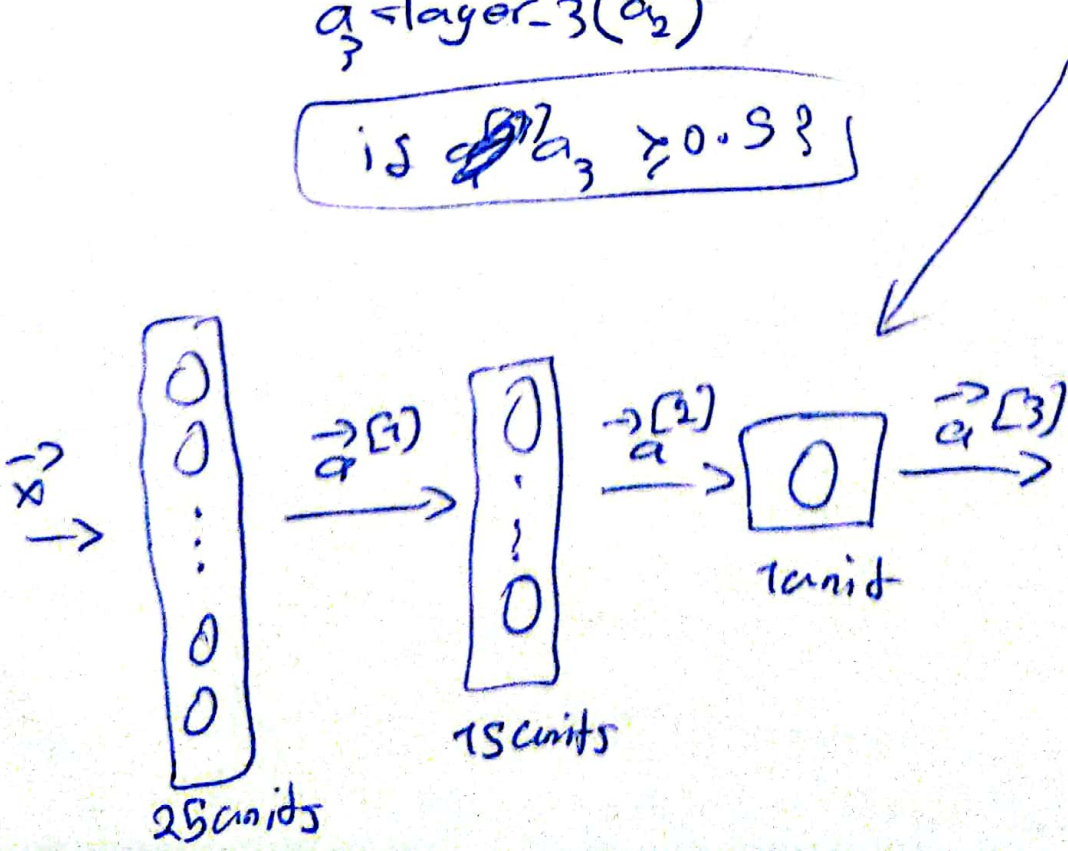
```
layer_2 = Dense(units=1, activation="sigmoid")
a2 = layer_2(a1)

if a2 >= 0.5:
    yhat = 1
else:
    yhat = 0
```

exp2

```
X = np.array([[0.0, ... 245, ... 240, ... 0]])
layer_1 = Dense(units=25, activation="sigmoid")
a1 = layer_1(X)
layer_2 = Dense(units=15, activation="sigmoid")
a2 = layer_2(a1)
layer_3 = Dense(units=1, activation="sigmoid")
a3 = layer_3(a2)
```

is $a_3 \geq 0.5$?



"data in Tensorflow"

- Tensorflow is created by Google brain Team.

→ a row vector

```
X = np.array([[7, 2]])
```

X.shape = (1×2)

→ a column vector

```
X = np.array([[1], [2]])   (2×1)
```

→ $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}_{(2\times3)}$ ⟹ np.array([[1, 2, 3], [4, 5, 6]])

→ $\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \\ 7 & 8 \end{bmatrix}_{(4\times2)}$ ⟹ np.array([[1,2], [3,4], [5,6], [7,8]])

```
x = np.array([1, 2])
```

→ a linear array

1D array

no row or columns

↓

Tensorflow doesn't accepts recognized 1D array

The input must be in 2D.

```
x = np.array([[0.7, 2]])
layer_1 = Dense(units=3, activation="sigmoid")
a1 = layer_1(x)
```

→ [[0.2, 0.7, 0.3]] (1×3) matrix

if we print a1:

output: tf.Tensor([[0.2, 0.7, 0.3]], shape=(1,3), dtype=float32)

a1.numpy() → converts "Tensor" to numpy "array"

array([[0.2, 0.7, 0.3]], dtype=float32)

"building a neural network" in Tensorflow

→ layer_1 = Dense(units=3, activation="sigmoid")
→ layer_2 = Dense(units=1, activation="sigmoid")
→ model = Sequential([layer_1, layer_2])

x = np.array([[1, 2], [2, 1], [3, 0], [4, 5]]) (4×2)

y = np.array([1, 0, 0, 1]) 1D array of length 4.

model.compile(...?...)
model.fit(x, y)

inference → model.predict(x_new)

simplify the creation of a model in Tensorflow (10)

```
model = Sequential([
    Dense(units=3, activation="sigmoid"),
    Dense(units=1, activation="sigmoid")])
```

```
model.compile(...?)
model.fit(X, Y)
model.Predict(X_new)
```

"forward propagation in a single layer"

$\vec{x} \rightarrow \begin{bmatrix} \vec{w}_1^{[1]}, b_1^{[1]} \\ \vec{w}_2^{[1]}, b_2^{[1]} \\ \vec{w}_3^{[1]}, b_3^{[1]} \end{bmatrix} \xrightarrow{\vec{a}^{[1]}} \begin{bmatrix} \vec{w}_1^{[2]}, b_1^{[2]} \end{bmatrix} \xrightarrow{\vec{a}^{[2]}}$

$a_1^{[2]} = a_{2-1}$

$\vec{x} = np.array([200, 17])$ 1D(array)

$a_1^{[1]} = g(\vec{w}_1^{[1]} \cdot \vec{x} + b_1^{[1]})$

$\vec{x}$ or $\vec{a}^{[0]}$

```
w1-1 = np.array([1,2])
b1-1 = np.array([-1])
```

```
Z1_1 = np.dot(w1_1, X) + b1_1
a1_1 = Sigmoid(Z1_1)
```

$$a_2^{[1]} = g(\vec{w}_2^{[1]} \cdot \vec{x} + b_2^{[1]})$$

```
w1_2 = np.array([-3, 4])
b1_2 = np.array([1])
Z1_2 = np.dot(w1_2, X) + b1_2
a1_2 = Sigmoid(Z1_2)
```

$$a_3^{[1]} = g(\vec{w}_3^{[1]} \cdot \vec{x} + b_3^{[1]})$$

```
w1_3 = np.array([5, -6])
b1_3 = np.array([2])
Z1_3 = np.dot(w1_3, X) + b1_3
a1_3 = Sigmoid(Z1_3)
```

```
a1 = np.array([a1_1, a2_2, a1_3])
```

$$a_1^{[2]} = g(\vec{w}_1^{[2]} \cdot \vec{a}^{[1]} + b_1^{[2]})$$

```
w2_1 = np.array([7, 8, 9])
b2_1 = np.array([3])
I2_1 = np.dot(w2_1, a1) + b2_1
a2_1 = sigmoid(I2_1)
```

"General implementation of forward propagation"

```
def dense(a_in, w, b):          # a_in: activation from the previous layer
    units = w.shape[1]
    a_out = np.zeros(units)
    for j in range(units):
        w = w[:, j]
        I = np.dot(w, a_in) + b[j]
        a_out[j] = g(Z)
    return a_out
```

```
w = np.array([
    [1, -3, 5],
    [2, 4, -6]])                # (2x3)
b = np.array([-1, 7, 2])        # 1D array
a_in = np.array([-2, 4])        # a[0] = x
```

```
def sequential(x):
    a1 = dense(x, w1, b1)
    a2 = dense(a1, w2, b2)
    a3 = dense(a2, w3, b3)
    a4 = dense(a3, w4, b4)
    f_x = a4                    ----------> return f_x
```

Speculations on Artificial General Intelligence (AGI)

AI

ANI (artificial narrow intelligence)
specific applications
E.g.; Smart speaker, self-driving car, web search, AI in farming and factories

AGI (artificial general intelligence)
do anything a human can do.

The "one learning algorithm" hypothesis.

"vectorization"

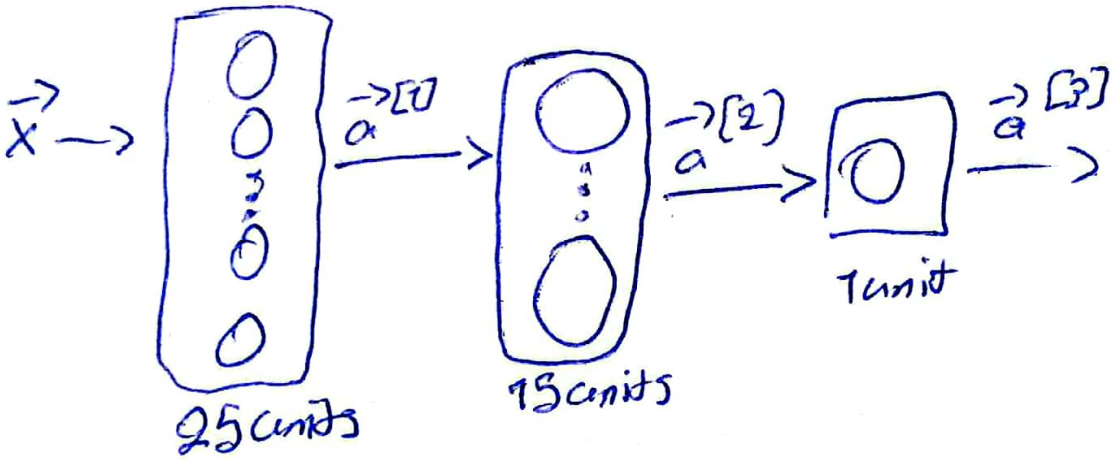
for loops vs. vectorization

```
X = np.array([[200, 17]])                    2D array
W = np.array([[1, -3, 5]
              [-2, 4, -5]])                  (2x3) 2D array
b = np.array([[-1, 1, 2]])                   (1x3) 2D array
```

```
def dense(A_in, W, B):
    Z = np.matmul(A_in, W) + B
    A_out = g(Z)
    return A_out  ---> [[1, 0, 1]]
```

Tensorflow implementation

Image recognition examples



given set of (X, y) examples

How to build and train this?

```
Import tensorflow as tf
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense
model = Sequential([
    Dense(units=25, activation="sigmoid"),
    Dense(units=15, activation="sigmoid"),
    Dense(units=1, activation="sigmoid"),
])
```

① builds the layers

```
from tensorflow.keras.losses import
    BinaryCrossentropy
model.compile(loss= BinaryCrossentropy())
model.fit(X, Y, epochs=100)
```

? — ② choose the loss function

③ train the model

Training details

logistic Regression

```
Z= np.dot(w, X) +b
f_x= 1/(1+np.exp(-Z))
```

① define model $f_{\vec{w},b}(\vec{x}) = ?$

```
loss= -y * np.log(f_x) - (1-y) * np.log(1-f_x)
```

② (logistic loss) loss function $L(f_{\vec{w},b}(\vec{x}), y) = ?$

```
w= w- alpha * dj_dw
b= b- alpha * dj_db
```

③ minimize $J(\vec{w}, b)$ (GD)

train a neural network in Tensorflow

```
model= Sequential([Dense(...), Dense(...), ..., Dense(...)])
model.compile(loss= BinaryCrossentropy())
model.fit(X, y, epochs=100)
```

① ② ③

logistic loss called as BinaryCrossentropy in Tensorflow.

$$L(f(\vec{x}), y) = -y \log(f(\vec{x})) - (1-y)\log(1-f(\vec{x}))$$

mean squared error called as MeanSquaredError() in Tensor flow (regresson models)

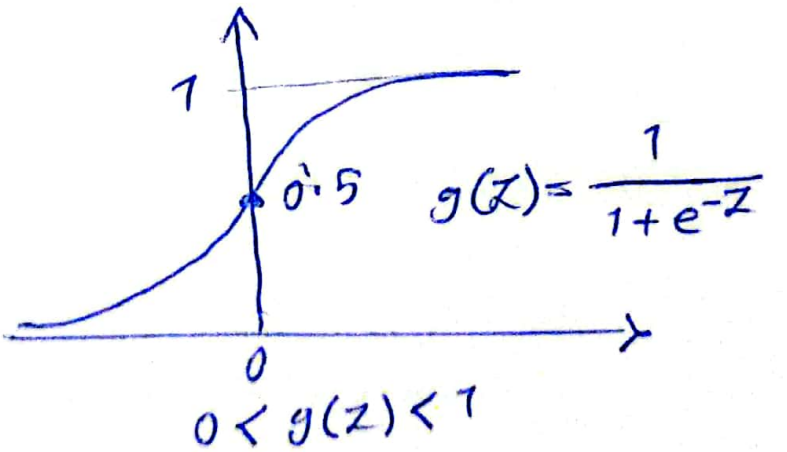
```
model.compile(loss= MeanSquaredError())
```

~~gradient descent called as~~

Activation functions

sigmoid activation: $a_2^{[1]} = g(\vec{w}_1^{[1]} \cdot \vec{x} + b_2^{[1]})$

(1)

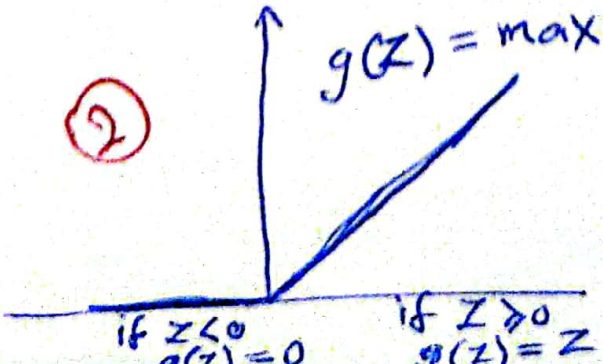


$0 < g(z) < 1$

a common activation function in neural networks: (ReLU)

rectified linear unit.

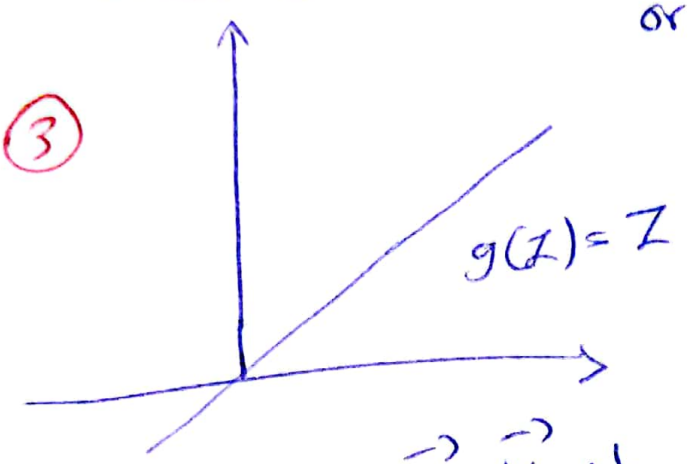
(2)



linear activation function

or no activation function

(3)



$$a = g(Z) = \underbrace{\vec{w} \cdot \vec{x} + b}_{Z}$$

(4) Softmax activation function (?)

choosing activation function

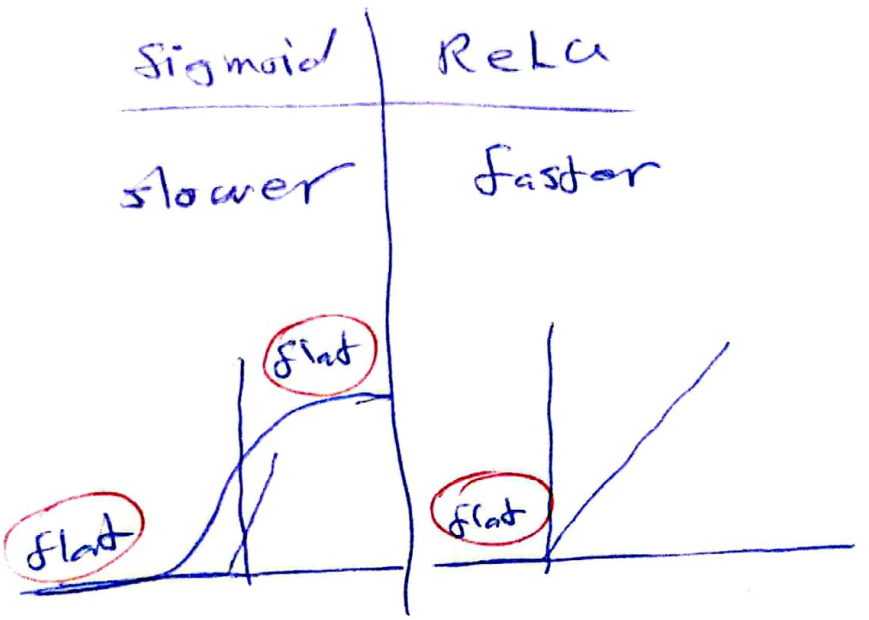
- one can choose different activation functions in different layers in a network.

output layer

- classification: (Binary) — sigmoid ($y = 0/1$) ($y = +/-$)
- regression:
  - linear activation function
  - ReLu ($y = 0$ or $+$)

for example the price of a product can never be negative, so ReLu is the best choice here.

Hidden Layer: ReLu is by far the most common choice.



exp

```
model=Sequential([
    Dense(units=25, activation="relu"),
    Dense(units=15, activation="relu"),
    Dense(units=1, activation="sigmoid")
])
```

if classification

else: relu or linear

"relu" or "linear"

"why activation function"

$$a^{[1]} = w_1^{[1]} \cdot \vec{x} + b_1^{[1]}$$

$$a^{[2]} = w_1^{[2]} \cdot a^{[1]} + b_1^{[2]}$$

$$= w_1^{[2]} \cdot \left(w_1^{[1]} \cdot \vec{x} + b_1^{[1]} \right) + b_1^{[2]}$$

$$a^{[2]} = \underbrace{\left(\vec{w}_1^{[2]} \cdot w_1^{[1]} \right)}_{w} \cdot \vec{x} + \underbrace{w_1^{[2]} \cdot b_1^{[1]} + b_1^{[2]}}_{b}$$

$$\vec{a}^{[2]} = wx + b \quad (?)$$

$f(x) = wx + b$ linear regression

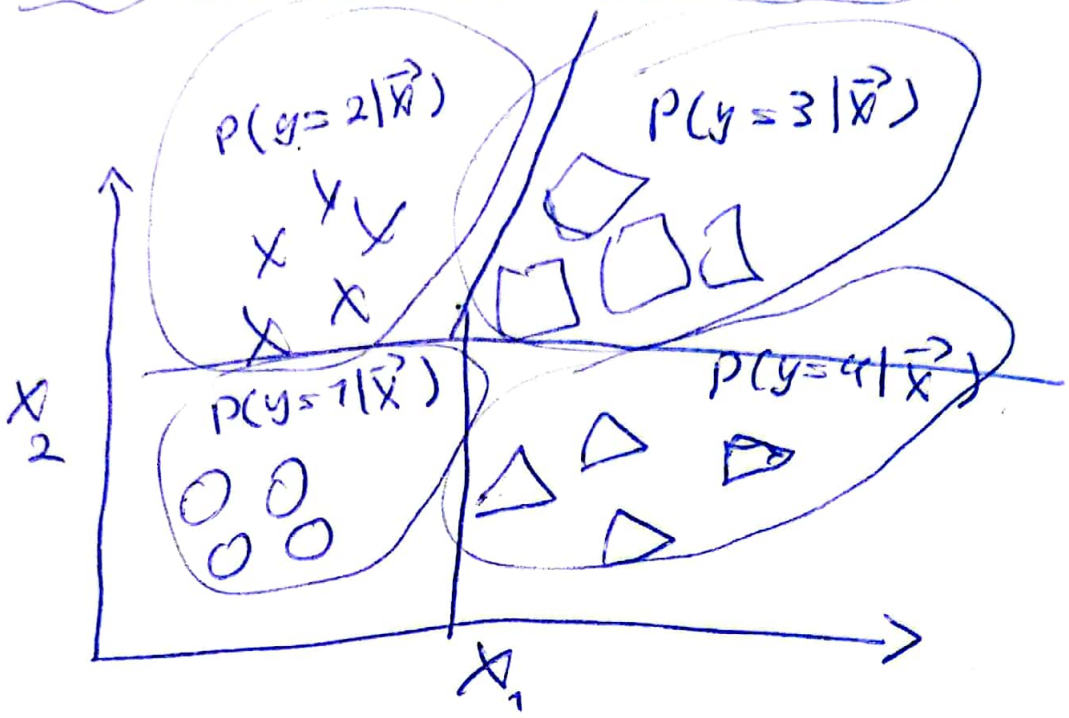
- if all layers of a neural network use the linear activation function the output is the same to the linear regression.

- if all hidden layers of a NN use the linear activation function and the output uses a sigmoid activation function, the output is equivalent to logistic regression.

- Do not use linear activation in hidden layers

(use ReLu)

multiclass classification



"Softmax regression algorithm"

- a generalization of logistic regression to the multiclass classification.

logistic regression (2 possible output values)

$$z = \vec{w} \cdot \vec{x} + b$$

$$a = g(z) = \frac{1}{1+e^{-z}} \begin{cases} P(y=1|\vec{x}) \\ P(y=0|\vec{x}) \end{cases}$$

Softmax regression (4 possible outputs)

- $\times$ $z_1 = \vec{w}_1 \cdot \vec{x} + b_1$
- $\square$ $z_3 = \vec{w}_3 \cdot \vec{x} + b_3$
- $\bigcirc$ $z_2 = \vec{w}_2 \cdot \vec{x} + b_2$
- $\triangle$ $z_4 = \vec{w}_4 \cdot \vec{x} + b_4$

$$a_1 = \frac{e^{z_1}}{e^{z_1}+e^{z_2}+e^{z_3}+e^{z_4}} = P(y=1|\vec{x})$$

$$a_2 = \frac{e^{z_2}}{e^{z_1}+e^{z_2}+e^{z_3}+e^{z_4}} = P(y=2|\vec{x})$$

$$a_3 = \frac{e^{z_3}}{e^{z_1}+e^{z_2}+e^{z_3}+e^{z_4}} = P(y=3|\vec{x})$$

$$a_4 = \frac{e^{z_4}}{e^{z_1}+e^{z_2}+e^{z_3}+e^{z_4}} = P(y=4|\vec{x})$$

---

softmax regresson (N possible outputs)

$$z_j = \vec{w_j} \cdot \vec{x} + b_j \qquad j = 1, 2, \ldots, N$$

$$\vec{w} = w_1, w_2, \ldots, w_N$$

$$\vec{b} = b_1, b_2, \ldots, b_N$$

$$a_j = \frac{e^{z_j}}{\sum_{k=1}^{N} e^{z_k}} = P(y=j|\vec{x})$$

note: $a_1 + a_2 + \cdots + a_N = 1$

---

cost for softmax function

logistic regresson

$$z = \vec{w} \cdot \vec{x} + b$$

$$a_1 = g(z) = \frac{1}{1+e^{-z}} = P(y=1|\vec{x})$$

$$= P(y=0|\vec{x})$$

$$a_2 = 1 - a_1$$

$$loss = -y \log a_1 - (1-y) \log (1-a_1)$$

$y \log a_1$: if $y=1$

$(1-y) \log (1-a_1)$: a_2, if $y=0$

---

softmax regression

$$a_1 = \frac{e^{Z_1}}{e^{Z_1}+e^{Z_2}+\dots+e^{Z_N}} = P(y=1 \mid \vec{X})$$

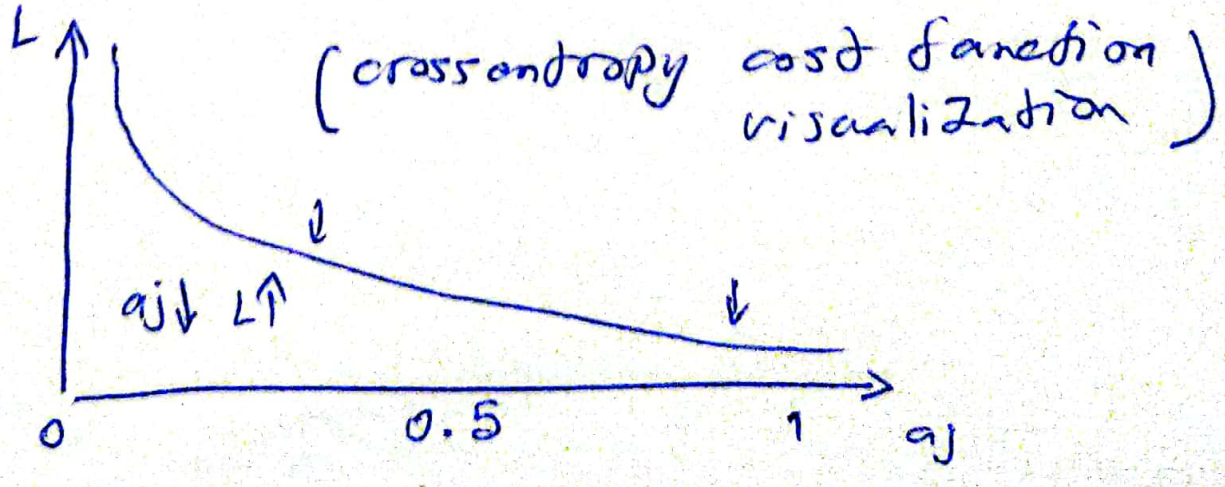
$$\vdots$$

$$a_N = \frac{e^{Z_N}}{e^{Z_1}+e^{Z_2}+\dots+e^{Z_N}} = P(y=N \mid \vec{X})$$

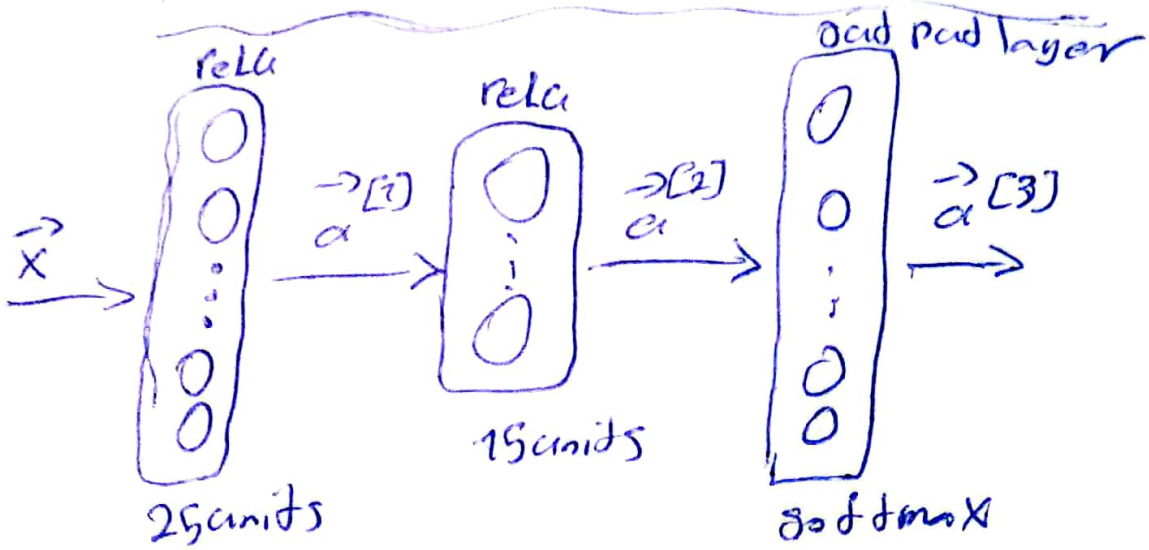
crossentropy

$$loss(a_1, \dots, a_N, Y) = \begin{cases} -\log a_1 & \text{if } y=1 \\ -\log a_2 & \text{if } y=2 \\ \vdots & \\ -\log a_N & \text{if } y=N \end{cases}$$

$$loss = -\log a_j \text{ if } y=j$$



Neural Network with softmax output



$$\vec{a}^{[1]} \to \vec{a}^{[2]} \to \vec{a}^{[3]}$$

$$I_1^{[3]} = w_1^{[3]} \cdot a^{[2]} + b_1^{[3]}$$

$$\vdots$$

$$I_{10}^{[3]} = w_{10}^{[3]} \cdot a^{[2]} + b_{10}^{[3]}$$

$$a_1^{[3]} = \frac{e^{I_1}}{e^{I_1} + \dots + e^{I_{10}}} = P(y=1 \mid \vec{x})$$

$$\vdots$$

$$a_{10}^{[3]} = \frac{e^{I_{10}}}{e^{I_1} + \dots + e^{I_{10}}} = P(y=10 \mid \vec{x})$$

$$\vec{a}^{[3]} = (a_1^{[3]}, \dots, a_{10}^{[3]}) = g(I_1^{[3]}, \dots, I_{10}^{[3]})$$

Softmax in TensorFlow

```
import tensorflow as tf
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense

model = Sequential([
    Dense(units=25, activation="relu"),
    Dense(units=15, activation="relu"),
```

```
Dense(units=10, activation="softmax")

])

from tensorflow.keras.losses import
                    SparseCategoricalCrossentropy

model.compile(loss=SparseCategoricalCrossentropy())

model.fit(X, y, epochs=100)
```

---

improved implementation of softmax

more numerically accurate implementation of logistic loss:

```
model = Sequential([
    Dense(units=25, activation="relu"),
    Dense(units=15, activation="relu"),
    Dense(units=10, activation="sigmoid")
                                    linear

model.compile(loss=BinaryCrossEntropy(from_logits=True)
model.fit(X, y, epochs=100)   , logit = model(X)
   train          softmax    f_x = tf.nn.sigmoid(logit)
                                    prediction

model = Sequential([
    Dense(units=25, activation="relu"),
    Dense(units=15, activation="relu"),          -> "linear"
    Dense(units=10, activation="softmax")])
```

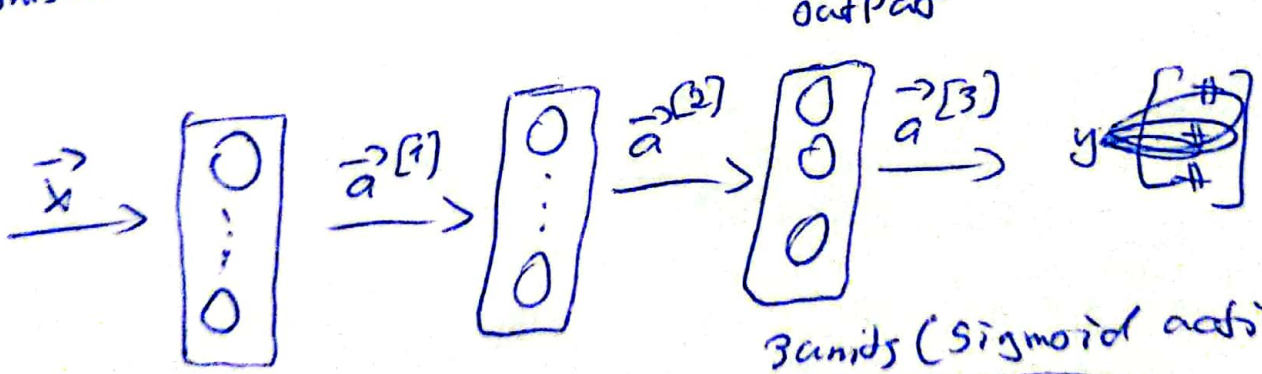
```
model.compile(loss=SparseCategoricalCrossEntropy(from_logits=True))
model.fit(X, Y, epochs=100) ---> train
logits = model(X) ---> predict [not a1, ..., a10
                                is Z1, ..., Z10]
f_x = tf.nn.softmax(logits)
```

"multilabel classification"

exp) in a picture

	image1	image2	image3
is there a car?	yes	no	yes
is there a bas?	no	no	yes
is there a Pedestrain?	yes	yes	yes
	$y = \begin{bmatrix}1\\0\\1\end{bmatrix}$	$y = \begin{bmatrix}0\\0\\1\end{bmatrix}$	$y = \begin{bmatrix}1\\1\\1\end{bmatrix}$

In this case train a model with three outputs:



3 units (sigmoid activation)

output → $\vec{a}^{[3]} = \begin{bmatrix} a_1^{[3]} \\ a_2^{[3]} \\ a_3^{[3]} \end{bmatrix}$ car, bas, Pedestration

Gradient descent

$$w_j = w_j - \alpha \frac{\partial}{\partial w_j} J(\vec{w}, b)$$

"Adam" algorithm : adjust the learning rate automatically.

Adam: Adaptive moment estimation

not just one alpha,

different alpha

exp

$$w_1 = w_1 - a_1 \frac{\partial}{\partial w_1} J(\vec{w}, b)$$

$$\vdots \qquad \vdots$$

$$w_{10} = w_{10} - a_{10} \frac{\partial}{\partial w_{10}} J(\vec{w}, b)$$

$$b = b - a_{11} \frac{\partial}{\partial b} J(\vec{w}, b)$$

if w_j (or b) keeps moving in same direction, increase a_j

if w_j (or b) keeps oscillating, reduce a_j

Implement ADAm algorithm in Tensorflow

```
model = Sequential([
    Dense(units=25, activation="sigmoid"),
    Dense(units=15, activation="sigmoid"),
    Dense(units=10, activation="linear"),
])

model.compile(optimizer=tf.keras.optimizers.
    Adam(learning_rate=1e-3), loss=
SparseCategoricalCrossentropy(fromlogits=True))

model.fit(X, Y, epochs=100)
```

Additional layer types

Dense Layer: each neuron output is a function of all the activation outputs of the previous layer.

convolutional layer: { faster computation
need less training data
(less prone to overfitting)

derivative

Cost function $J(w) = w^2$

Say $w = 3$ $\quad J(w) = 3^2 = 9$

if we increase w by a tiny amount $\varepsilon = 0.001$

How does $J(w)$ change?

$w = 3 + 0.001$ $\quad J(w) = w^2 = 9.006001$

if $w \uparrow 0.001$ $\quad \varepsilon$

$J(w) \uparrow 6 \times 0.001$ $\quad 6 \times \varepsilon$

$$\frac{d}{dw} J(w) = 6$$

informal definition of derivative

if $w \uparrow \varepsilon$ causes $J(w) \uparrow K \times \varepsilon$ then

$$\frac{d}{dw} J(w) = K$$

Gradient Descent

repeat {

$$w_j = w_j - \alpha \frac{d}{dw_j} J(\vec{w}, b)$$

}

if derivative is small, then this update step will make a small update to w_j and if it is large, then the update step makes a large update to w_j.

$w = 2 \rightarrow J(w) = w^2 = 4 \rightarrow w \uparrow 0.001$

$J(w) = J(2.001) = 4.004001$

$J(w) \uparrow 4 \times 0.001$

$\frac{d}{dw} J(w) = 4$

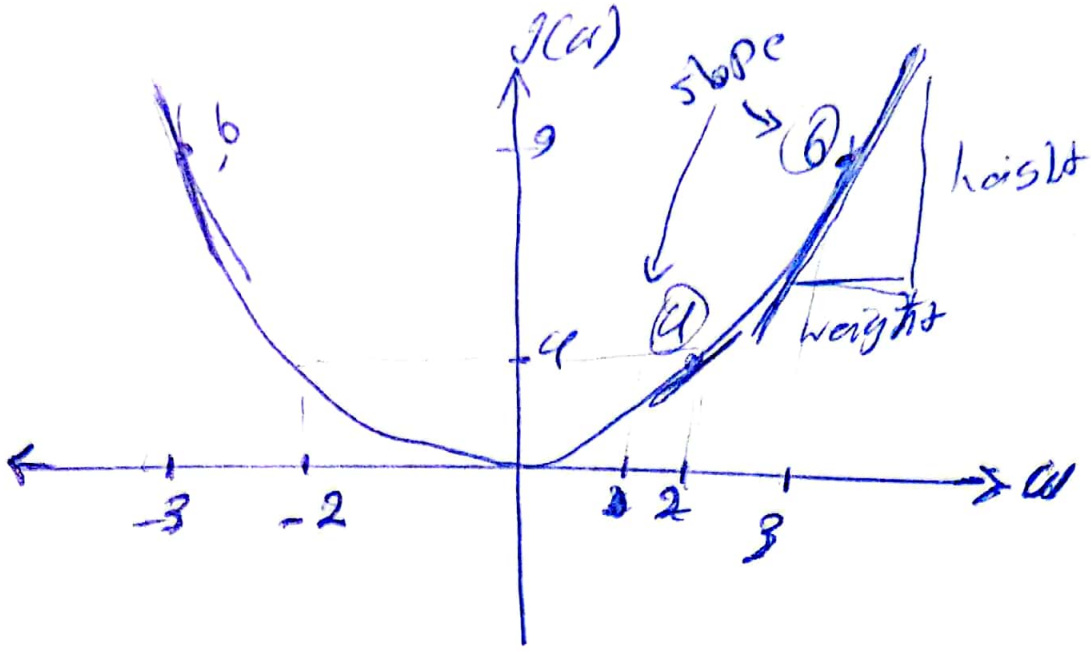
$w = -3 \quad J(w) = w^2 = 9 \quad w \uparrow 0.001$

$J(w) = J(-2.999) = 8.994001$

-0.006

$J(w) \downarrow 6 \times 0.006$

$\frac{d}{dw} J(w) = -6$



$$\frac{d}{dw} J(w) = 2w$$

```
import sympy

J, w = sympy.symbols("J, w")

J = w**2

dJ_dw = sympy.diff(J, w)   # = 2w
```

calculate derivative with sympy.

calculate the derivative of J with respect to w.

```
dJ_dw.subs([(w, 2)])   # = 4
```

plug a number in the derivative

if $J(w)$ is a function of one variable (w),

$\frac{d}{dw} J(w)$ → d → derivative notation

if $J(w_1, w_2, \ldots, w_n)$ is a function of more than one variable,

$\frac{\partial}{\partial w_i} J(w_1, w_2, \ldots, w_n)$ → ∂ → partial derivative notation

---

"computation graph"

linear activation $a = g(z) = z$

$a = wx + b$

$x \rightarrow \boxed{\;} \xrightarrow{a}$

$J(w,b) = \frac{1}{2}(a-y)^2$

if $w=2$, $b=8$, $x=-2$, $y=2$

$w \xrightarrow{2} \boxed{c = wx} \xrightarrow{-4} \boxed{a = c+b} \xrightarrow{4} \boxed{d = a-y}$

$b \xrightarrow{8}$ (into $a = c+b$)

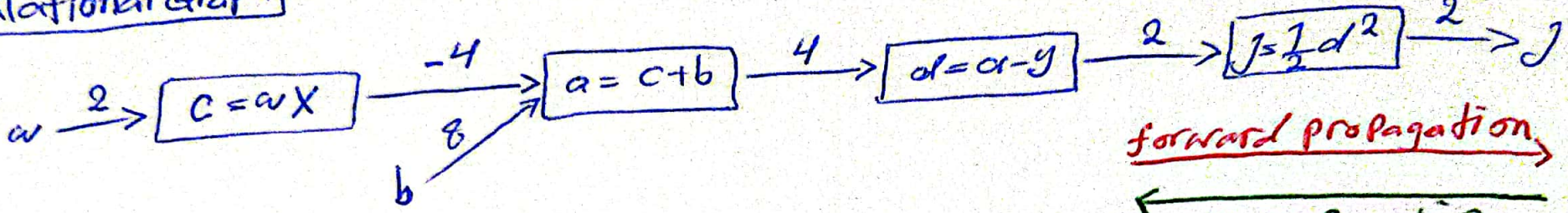
$\xrightarrow{2} \boxed{J = \frac{1}{2}d^2} \xrightarrow{2} J$

computation graph

(32)

$w=2 \quad b=8 \quad x=-2 \quad y=2 \quad a=wx+b \quad J=\frac{1}{2}(a-y)^2$

Computational Graph



Debugging a learning algorithm

if you implemented regularized linear regression on a dataset

$$J(\vec{w},b) = \frac{1}{2m}\sum_{j=1}^{m}\left(f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)}\right)^2 + \frac{1}{2m}\sum_{j=1}^{n} w_j^2$$

but it makes unacceptably large errors in predictions. what do you try next?

- Get more training example
- Try smaller set of features
- Try getting additional features
- Try adding Polynomial features (x_1^2, x_2^2, x_1x_2, etc)
- Try decreasing λ
- Try increasing λ

machine learning diagnostic

- A test which you can run to gain insight into what is/is not working with a learning algorithm, to gain guidance into improving its performance.

Evaluating a model

1) Train and test set : split data into train and test sets.

↙ compute the accuracy of the test set. Prediction error

2) model selection

1- $f_{\vec{w},b}(\vec{x}) = w_1 x + b$

2- $f_{\vec{w},b}(\vec{x}) = w_1 x + w_2 x^2 + b$

3- $f_{\vec{w},b}(\vec{x}) = w_1 x + w_2 x^2 + w_3 x^3 + b$

...

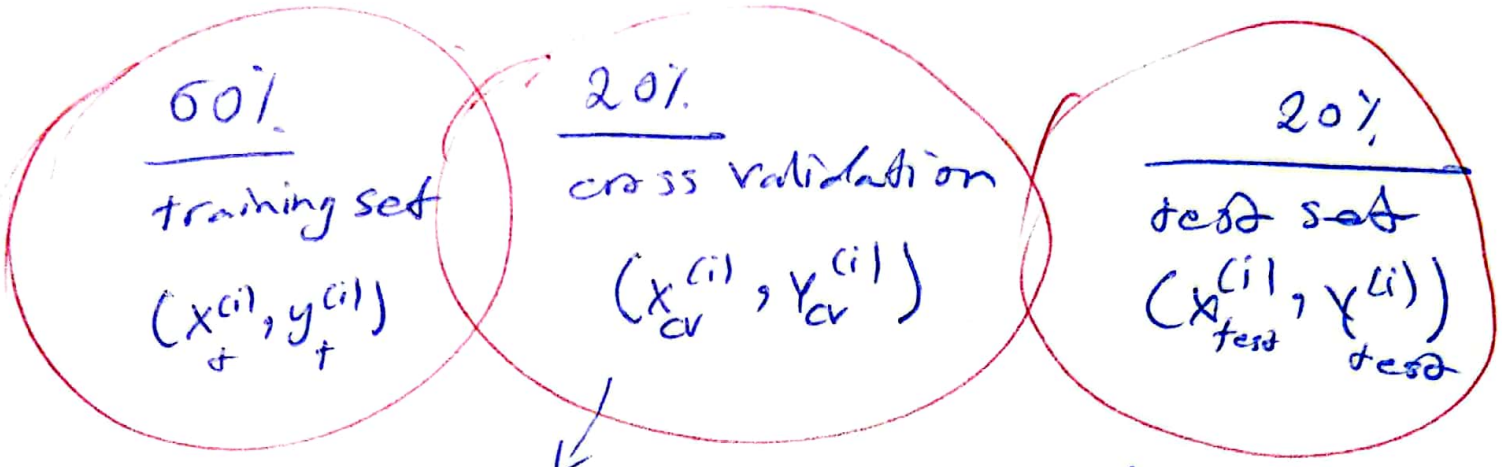
10- $f_{\vec{w},b}(\vec{x}) = w_1 x + w_2 x^2 + \dots + w_{10} x^{10} + b$

try different models and evaluate them and choose the one which gives you the best performance.

(validation set ; development set) → dev set

(3) Training / cross validation / test set.

exp

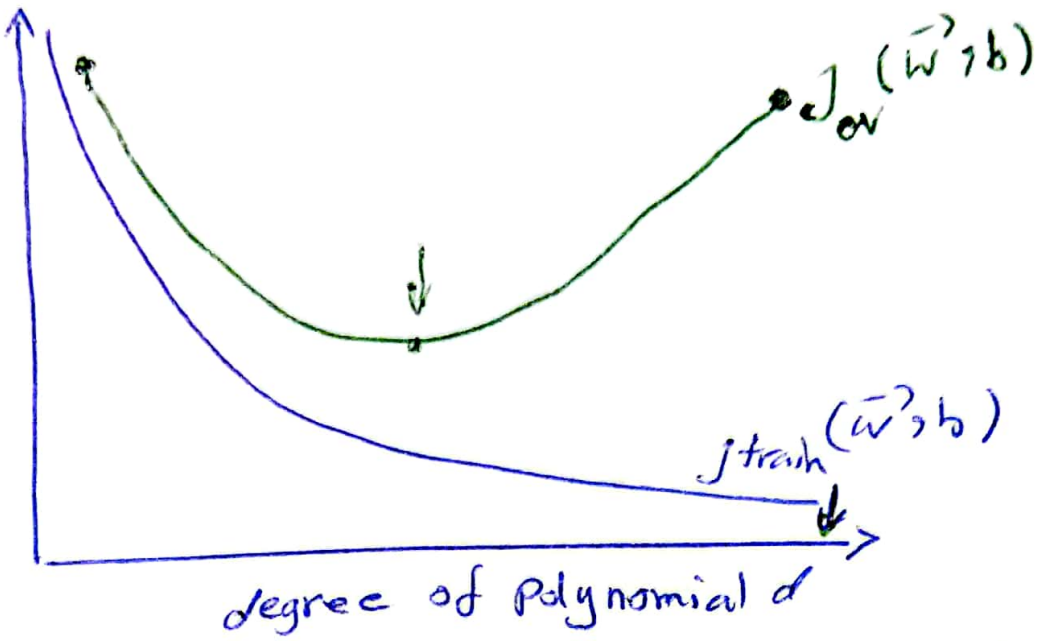


use this set to ~~evaluate~~ select the best model (model selection) and then use the test set to evaluate the performance of the model.

Bias / Variance

- High bias (underfit) → J_{train} is High, J_{cv} is High } underfitted
- High variance (overfit) → J_{train} is low, J_{cv} is high } & overfitted

right model : J_{train} is low, J_{cv} is low ✓



underfitted — overfitted

- How to detect High bias or High variance of model?

High bias (underfit)

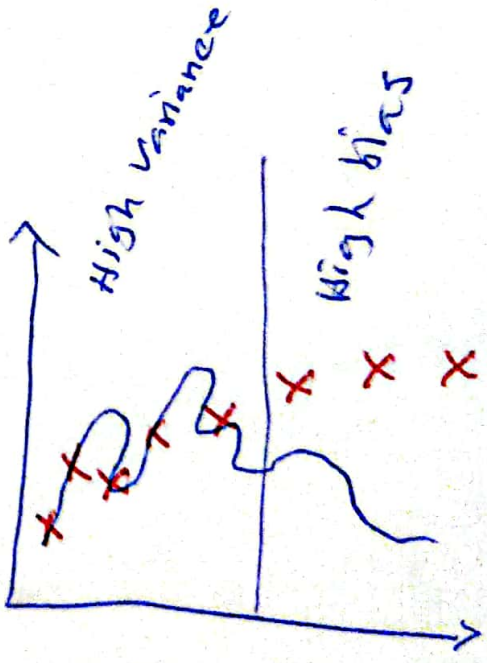
J_{train} will be high

($J_{train} \approx J_{cv}$)

High variance (overfit)

$J_{cv} \gg J_{train}$

(J_{train} may be low)



exp

High bias and High variance

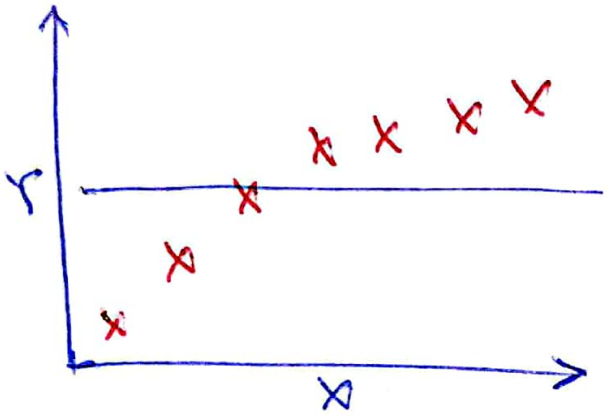
J_{train} will be high. and $J_{cv} \gg J_{train}$

regularization and bias/variance

exp

model: $f_{\vec{w},b}(x) = w_1 x + w_2 x^2 + w_3 x^3 + w_4 x^4 + b$

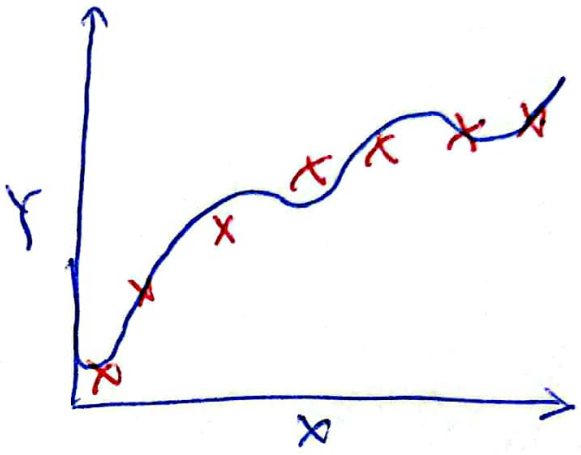
$$J(\vec{w},b) = \frac{1}{2m}\sum_{i=1}^{m}\left(f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)}\right)^2 + \frac{1}{2m}\sum_{j=1}^{n} w_j^2$$



λ = very high (10000)

(High bias)

$J_{train}(\vec{w},b)$ is large

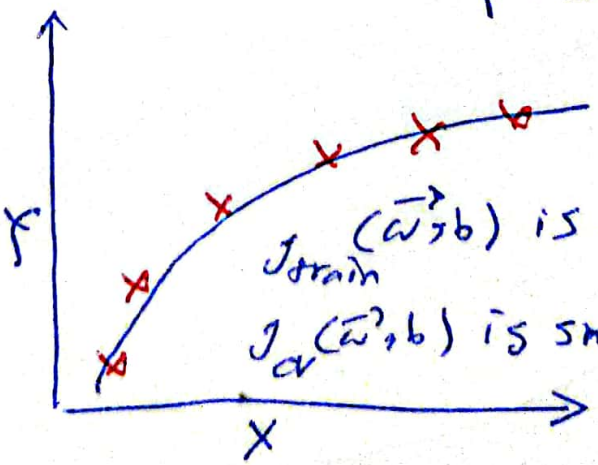


small λ ($\lambda = 0$)

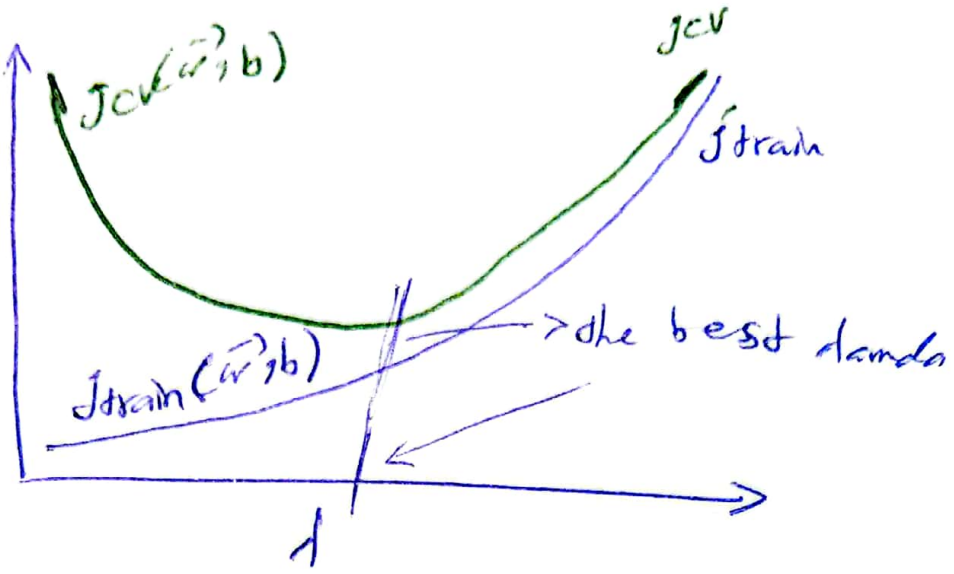
$J_{train}(\vec{w},b)$ is small.

$J_{cv}(\vec{w},b)$ is large.

(High variance)



Intermediate λ



Baseline performence

- what is the level of error you can reasonably hope to get to?

- Human level performance.
- Competing algorithms performance.
- Guess based on experience.

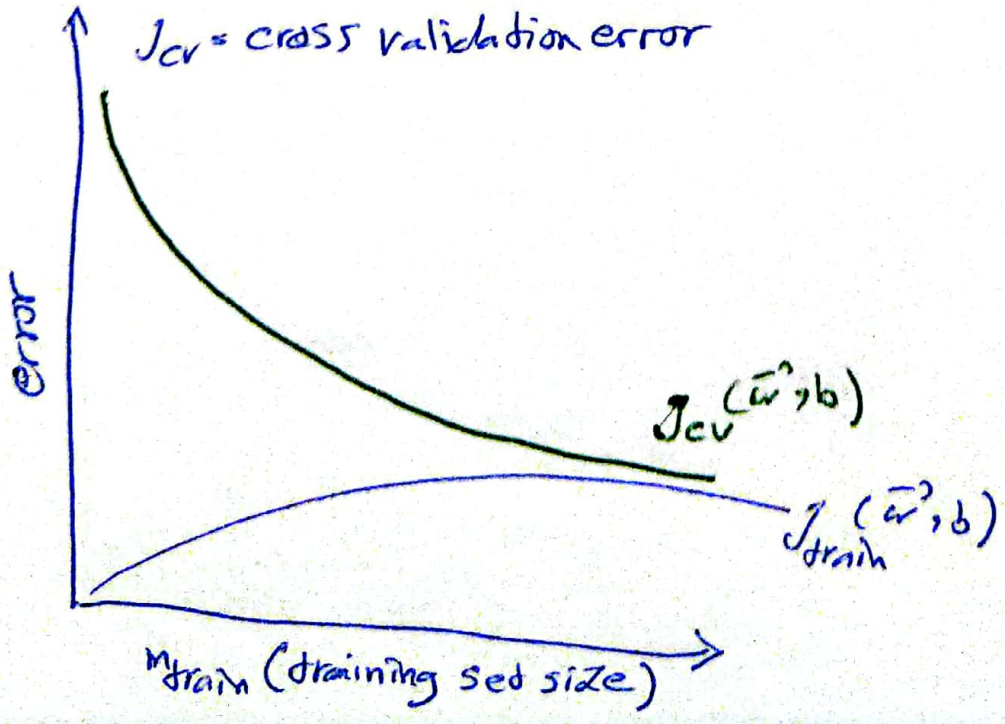
---

learning curves

J_{train} = training error

J_{cv} = cross validation error

model: $f_{\vec{w},b}(x) = w_1 x + w_2 x^2 + b$



- if a learning algorithm suffers from high variance (overfit), getting more training data is likely to help.

- if a learning algorithm suffers from high bias (under fit) getting more training data does not help.

---

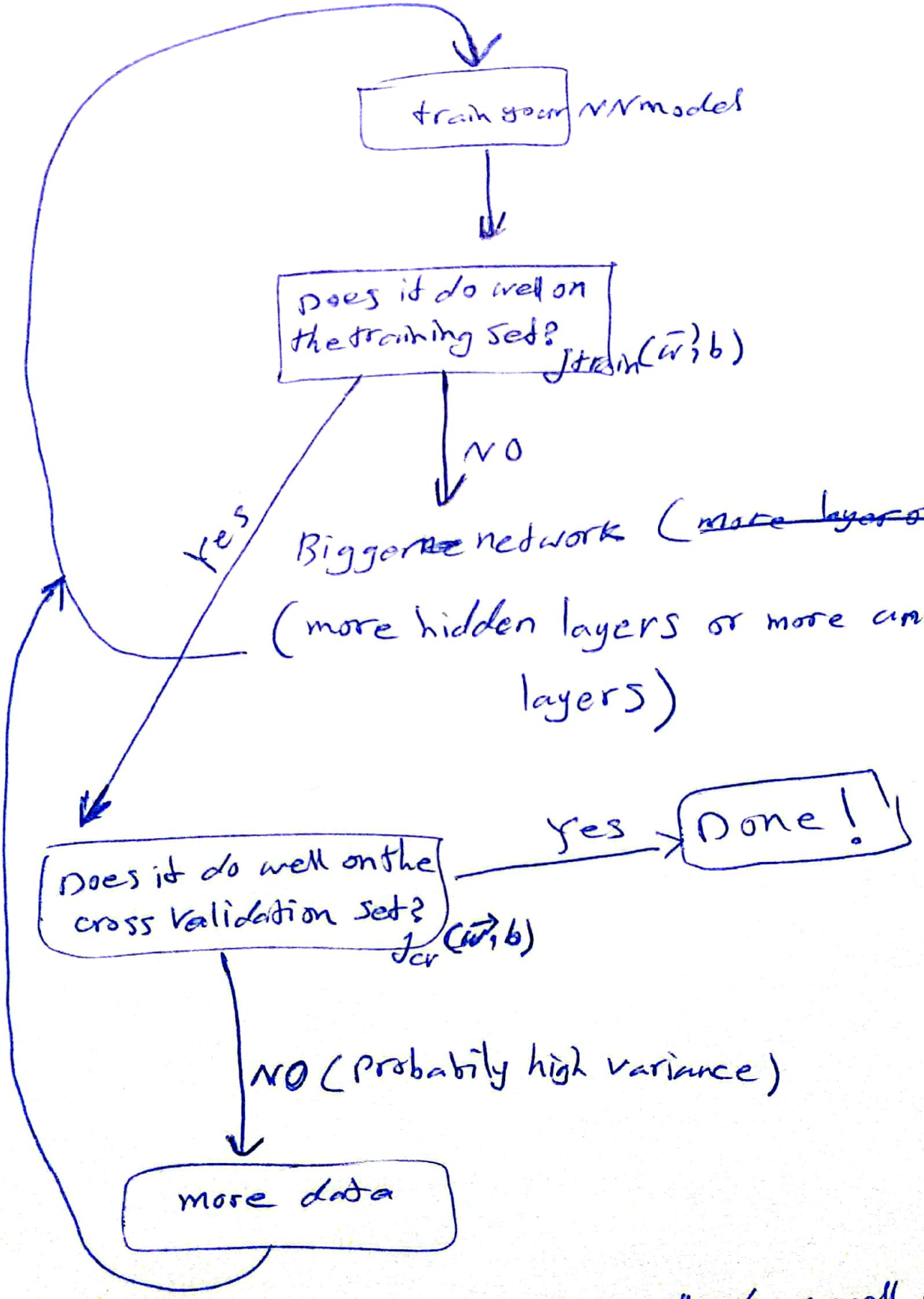
if a model get avery high error in predictions. what do you try next?

- Get more training example (fixes High variance)
- Try smaller sets of features (fixes High variance)
- Try getting additional features (fixes High bias)
- Try adding Polynomial features ($x_1^2, x_2^2, \ldots$) (fixes High bias)
- Try decreasing λ (fixes high bias)
- Try increasing λ (fixes high variance)

---

bias/variance and NNs.

- large neural networks are low bias machines, means if you make your model large enough, at the most cases you get a fitted model.

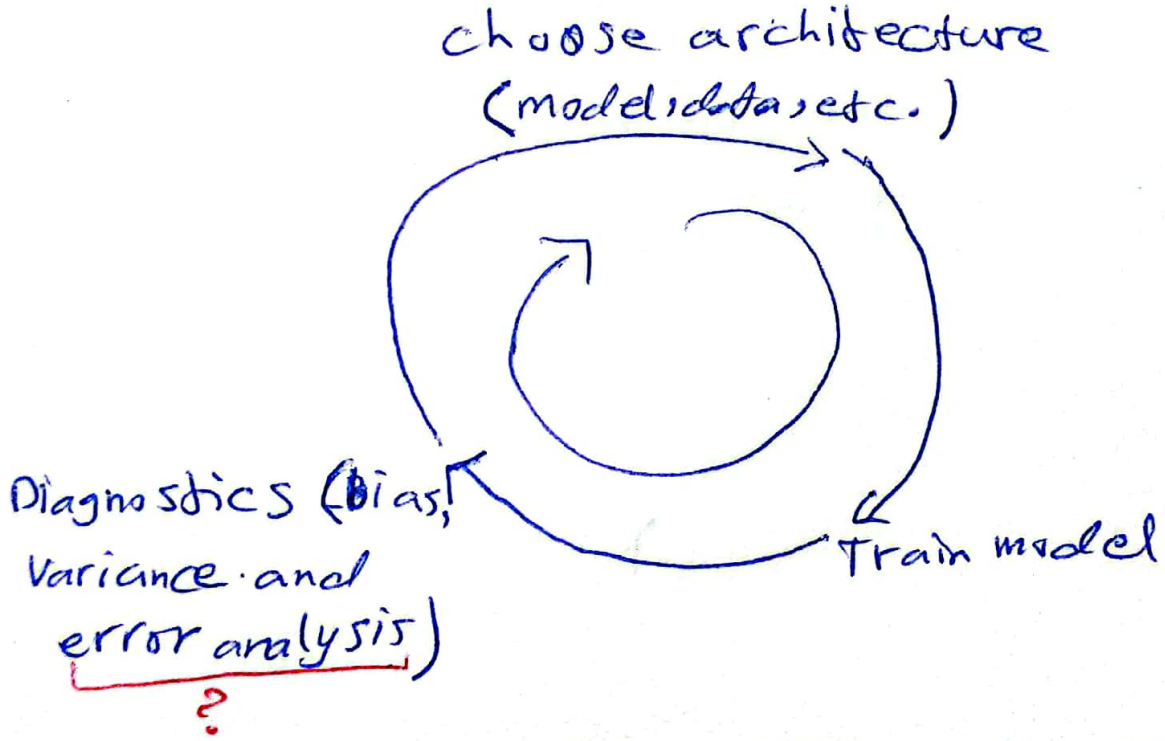


-A large neural network will usually do as well or better than a smaller one so long as regularization is chosen appropriately.

regularized model in tensorflow

```
layer = Dense(units=10, activation="relu",
              kernel_regularizer=L2(0.01))
```

Iterative loop of ML development



Error analysis

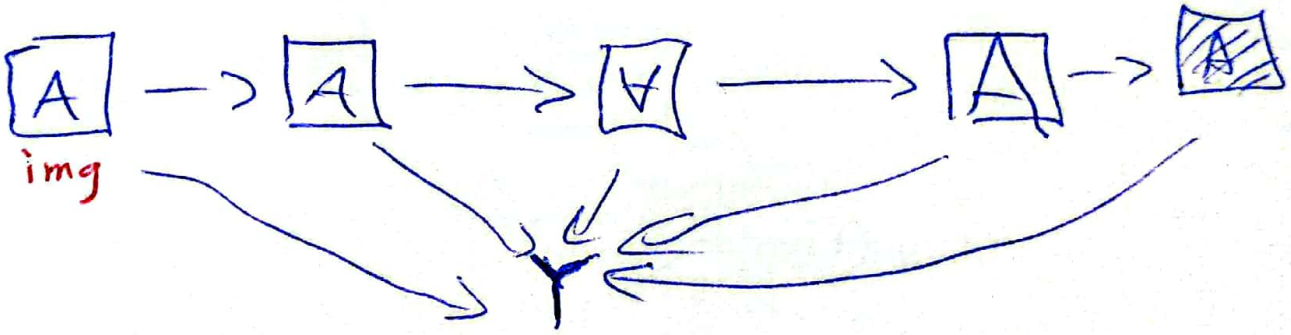
- Try to analyse misclassified examples (manualy collect them and analyse them and try to make decisions based on them to improve ~~your~~ the performance of your model.

Adding data

- Add more data of everything. E.g., "Honeypot" project.
- Add more data of things where Error analysis has indicated it might help.
- Beyond getting brand new training examples (x, y), another technique: Data augmentation

Augmentation: modifying an existing training example to create a new training example.

exp.



- Data synthesis: using artificial data inputs to create a new training example.

Engineering the data used by your system

Conventional model-centric approach:

AI = Code (model) + Data

work on this

Data-centric approach:

AI = code + Data

work on this

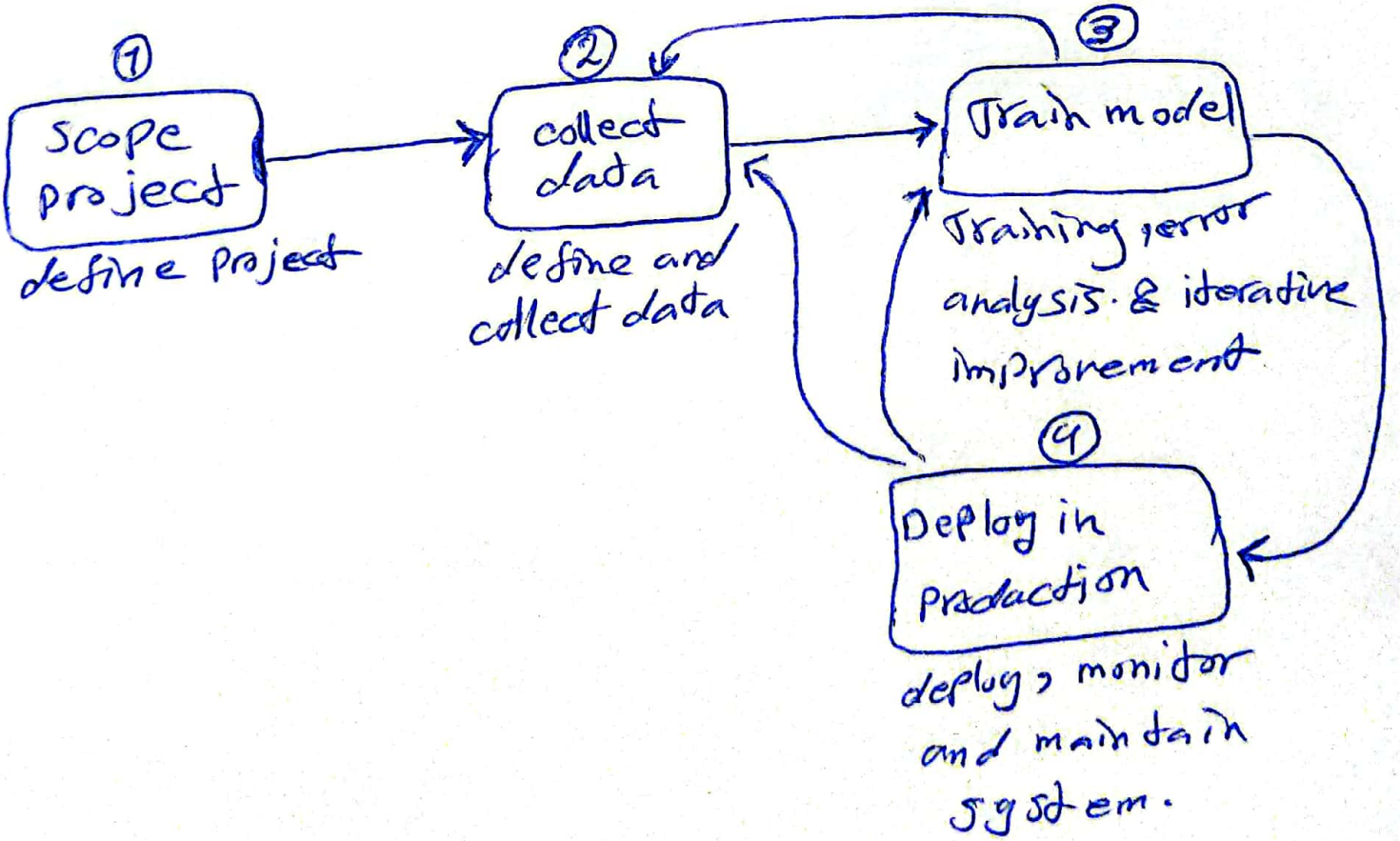
Transfer-learning

-using data from a different task.

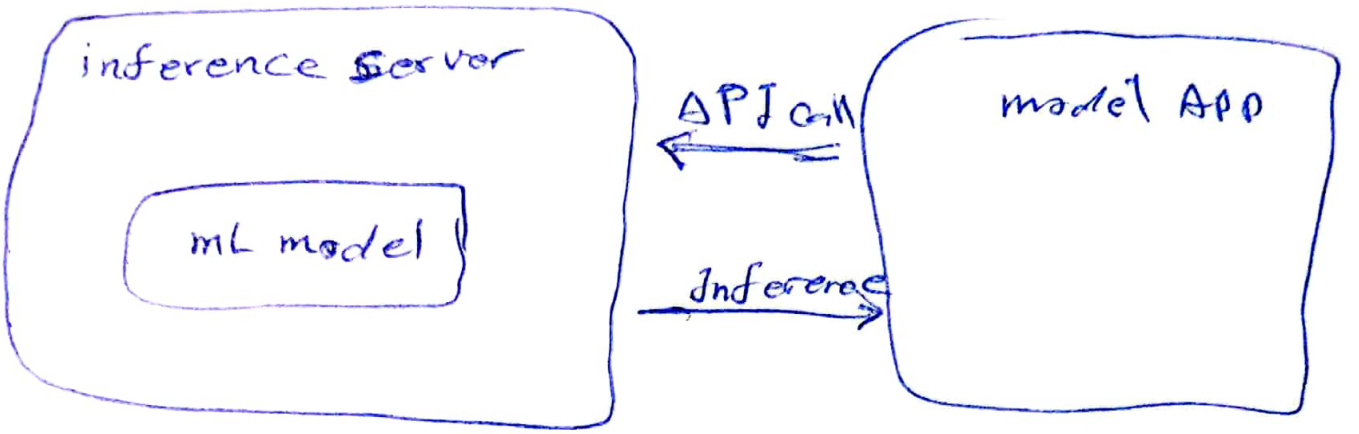
Steps

1. Download neural network Parameters pretrained on a large data set with same input type (e.g. images, audio, text) as your application (or train your own).
2. Further train (fine tune) the network on your own data.

full cycle of a ML project



Deployment



- Software engineering may be needed for:

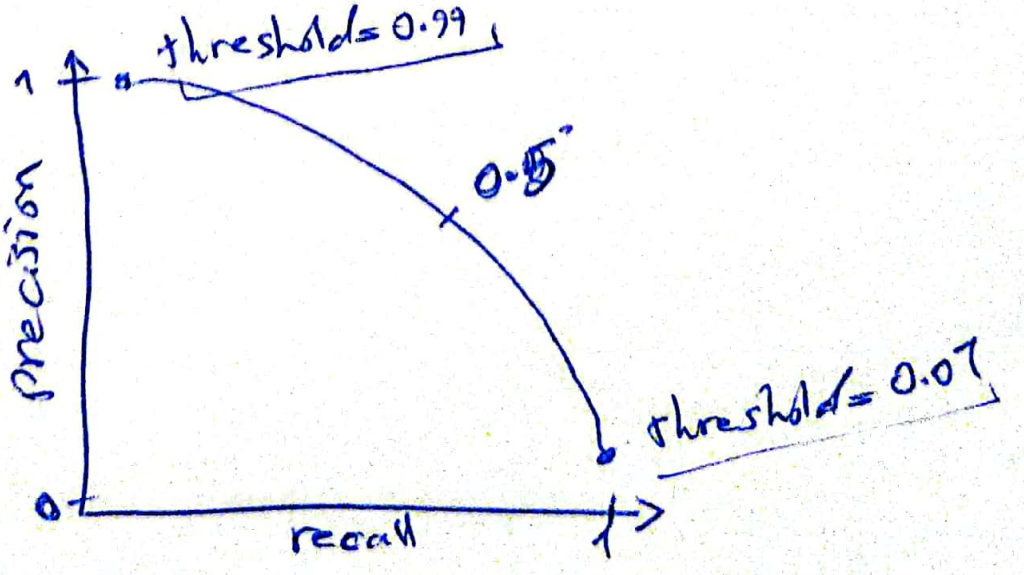
Precision/Recall

	1	0
1	True positive	false positive
0	false negative	true negative

$$\text{Precision} = \frac{\text{True Positive}}{\text{Predicted Positive}}$$

$$= \frac{\text{True Positive}}{\text{True-Pos} + \text{false-Pos}}$$

$$\text{recall} = \frac{\text{True\_Pos}}{\text{actual\_Pos}} = \frac{\text{True\_Pos}}{\text{True-Pos} + \text{false-neg}}$$



$$F1\text{-score} = \frac{1}{\frac{1}{2}\left(\frac{1}{P} + \frac{1}{R}\right)} = \frac{2PR}{P+R}$$

P = Precision

R = recall

Harmonic mean