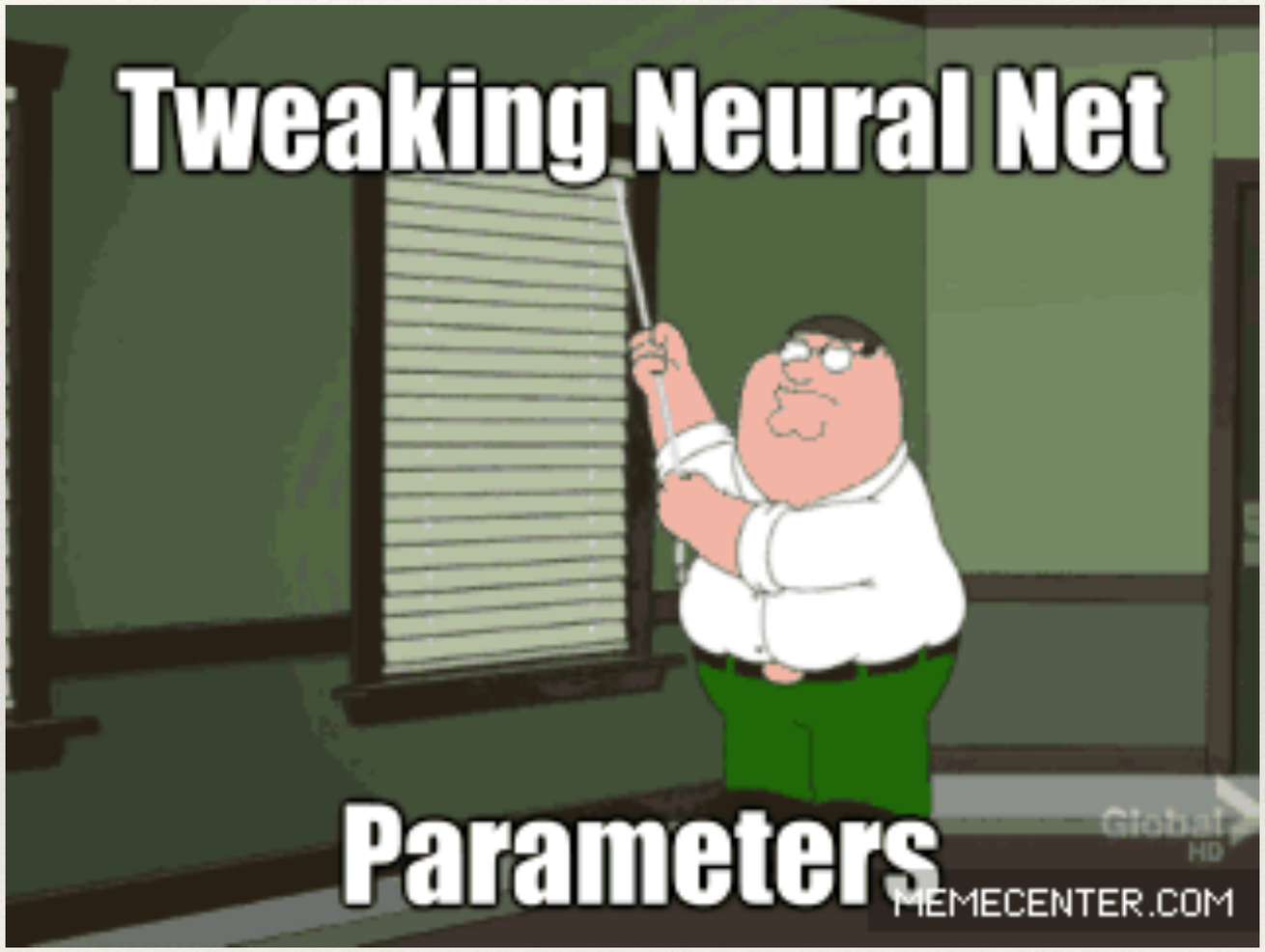
Reza Katebi

Generative Adversarial Networks (GAN)

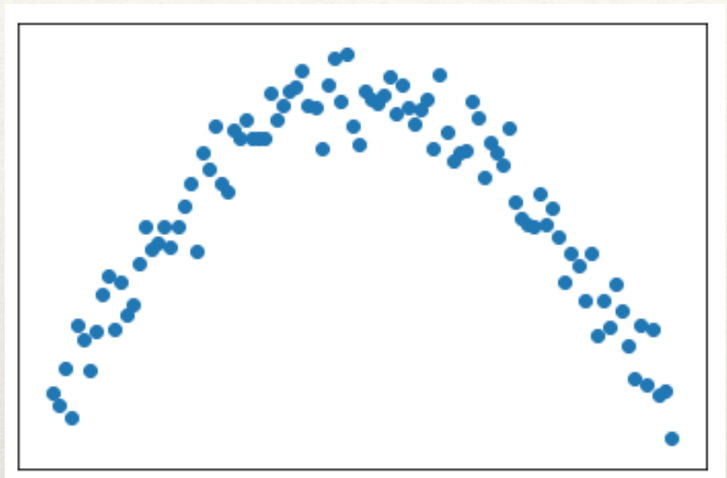
- How Neural Network works and can it be generative?
- Prisoner Dilema
- What is GAN?
- Implementing it together

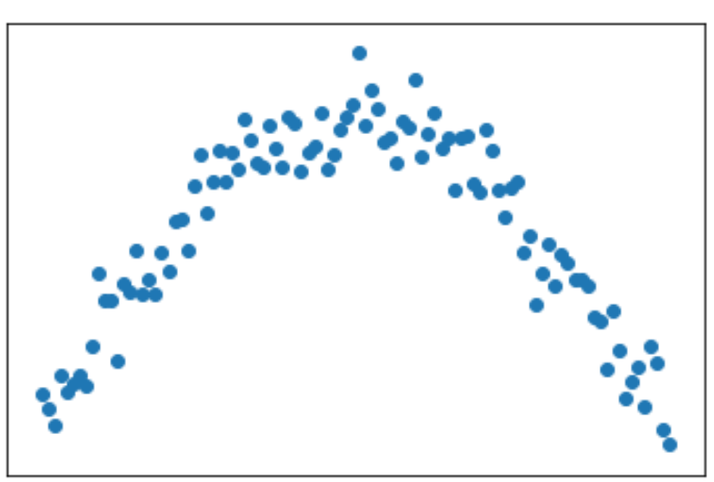
Neural Networks

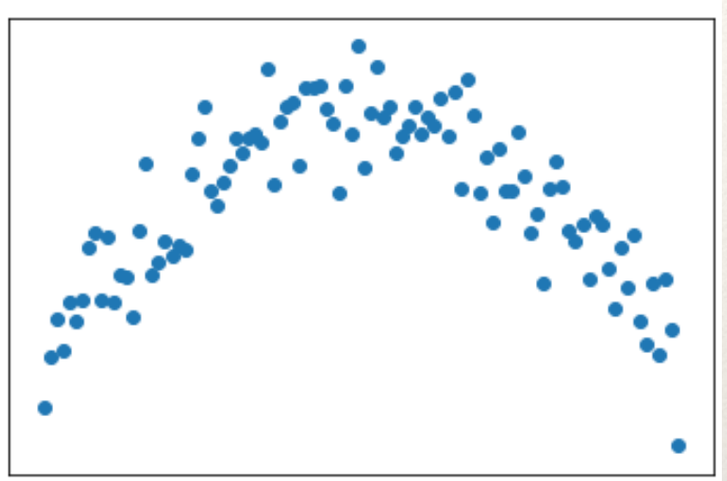
- Trained for classification or regression (Cat and Dog, House Pricing)
- Somewhere there there is a model of what is cat or dog
- It only can be used for classification
- It does not know how to generate new cat since it did not learn how

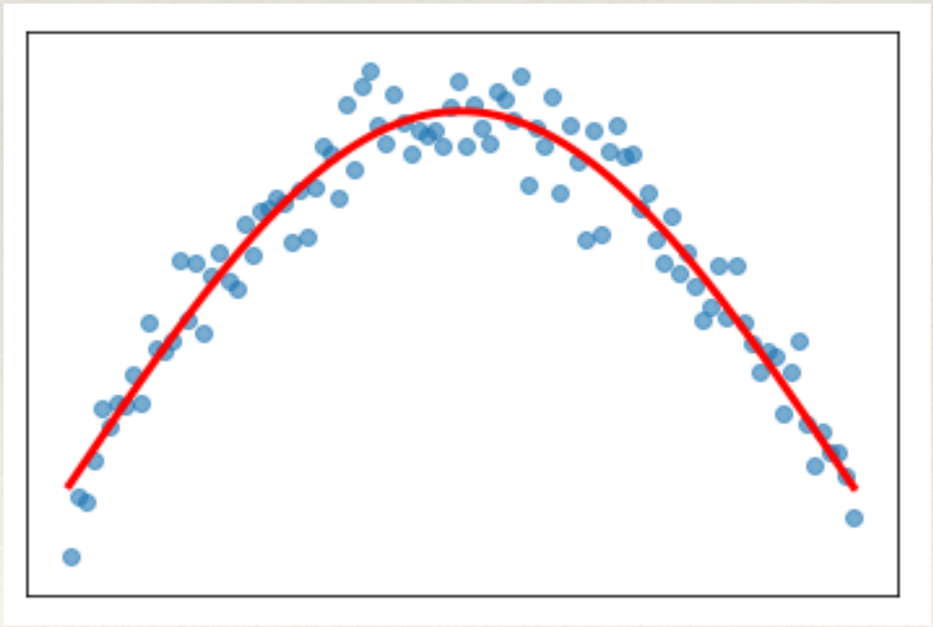


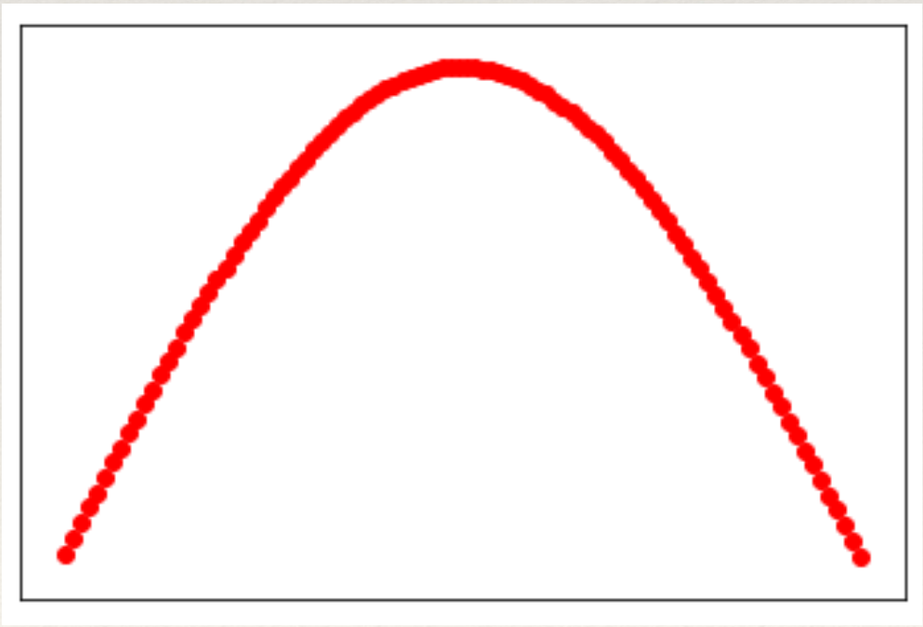
Neural Network





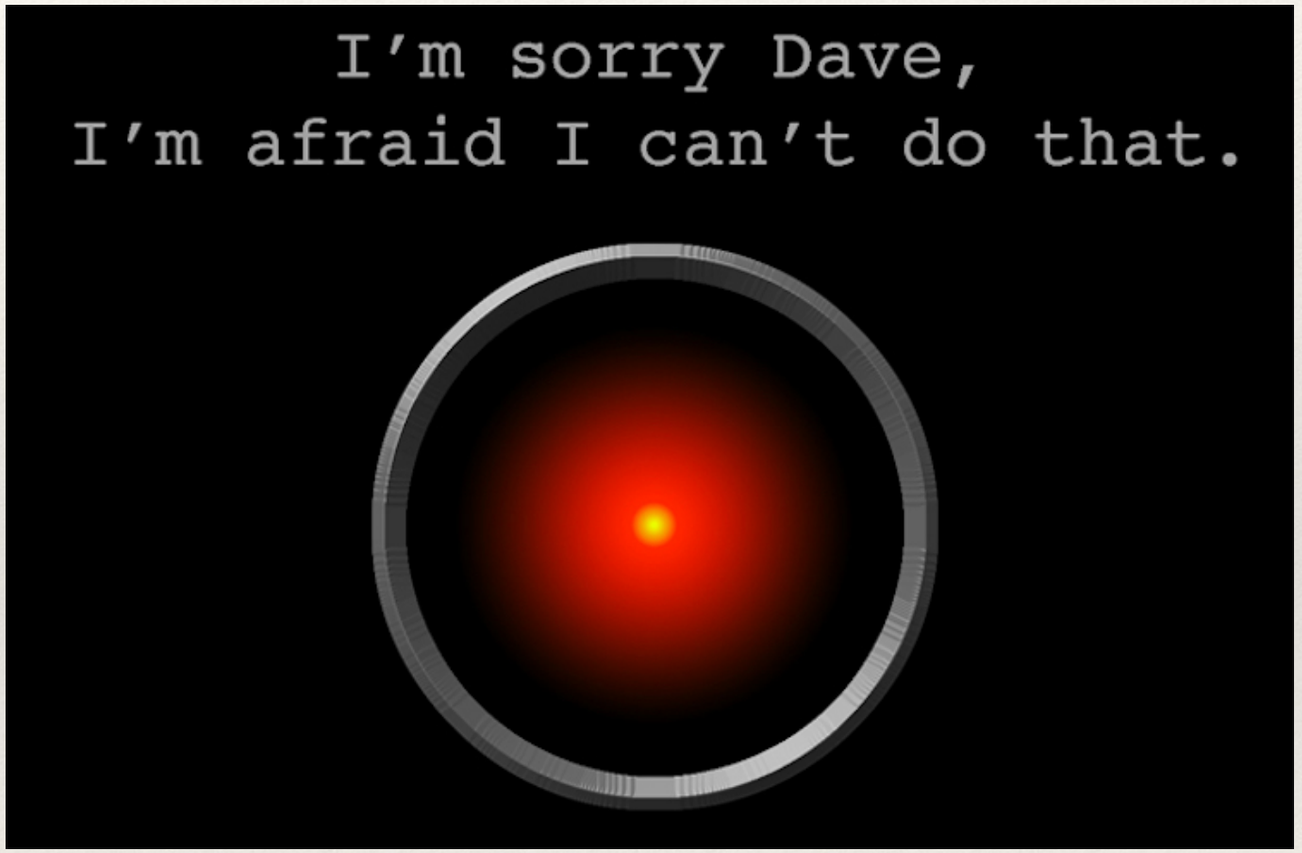






Neural Network

- Training on MNIST
- If misclassified 1 and 7
  - We repeat training for all numbers
- Humans (children)
  - If misclassified 1 and 7 and guessed
  - We focus on 1, 7 only
- Training where student is failing!
- You cannot go forever because humans will get tired and quit!
- You can hammer AI, No feeling yet!

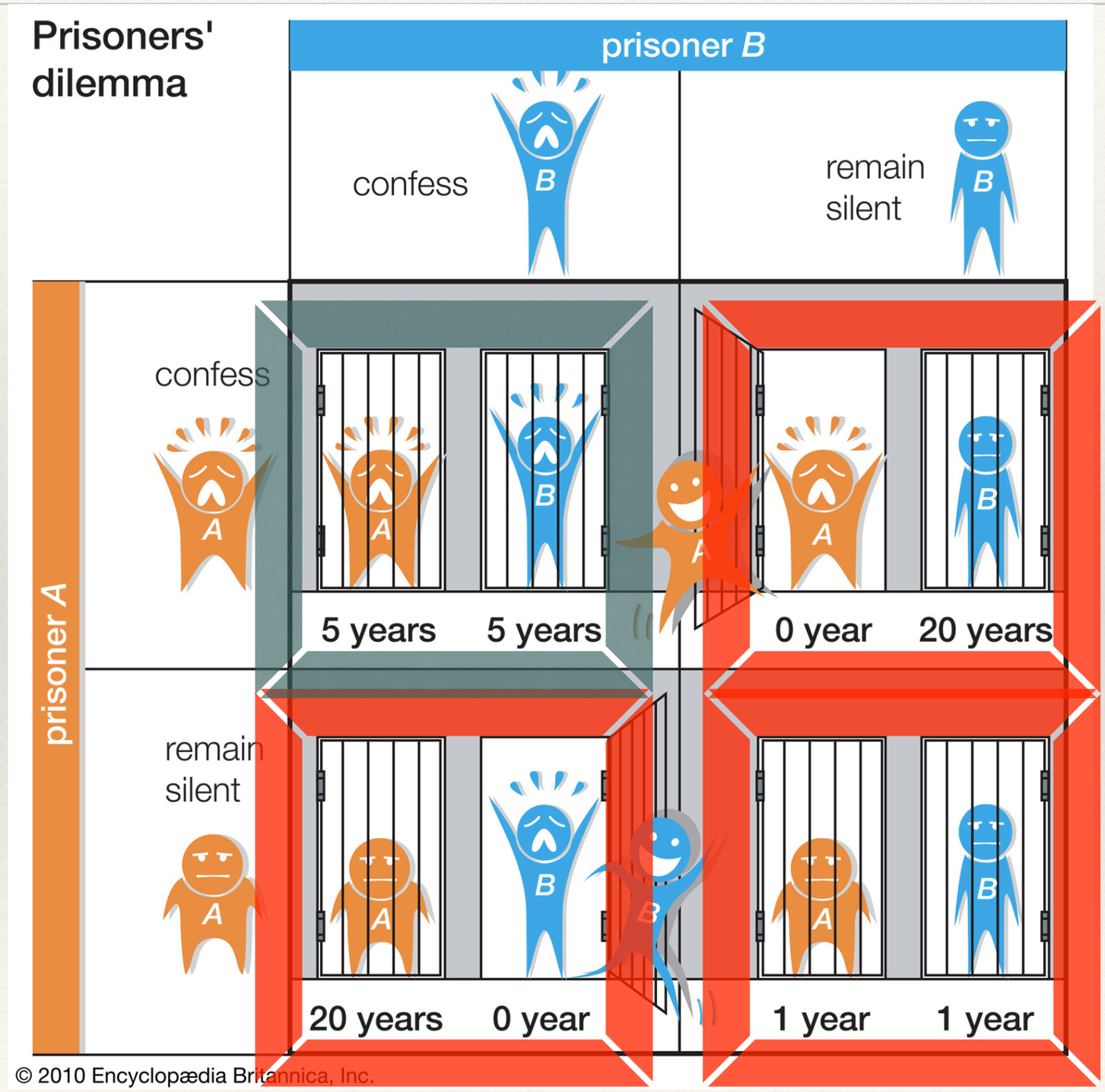


Neural Network

- Some process that will make the network to output as high as possible error
  - Spots weakness and forces network to focus on that
    - Network will resolve that weakness by learning
- Example: Game playing AI such as Alpha Zero
  - **19.6 million self played games in FOUR HOURS!!**
- GAN is something like this.

Zero-Sum Game

Nash Equilibrium



MiniMax Theorem

- Maximin: Largest value player A can gain not knowing the actions of player B:

$$\bar{V}_A = \max_{a_A} \min_{a_B} v_{a_A}(a_A, a_B)$$

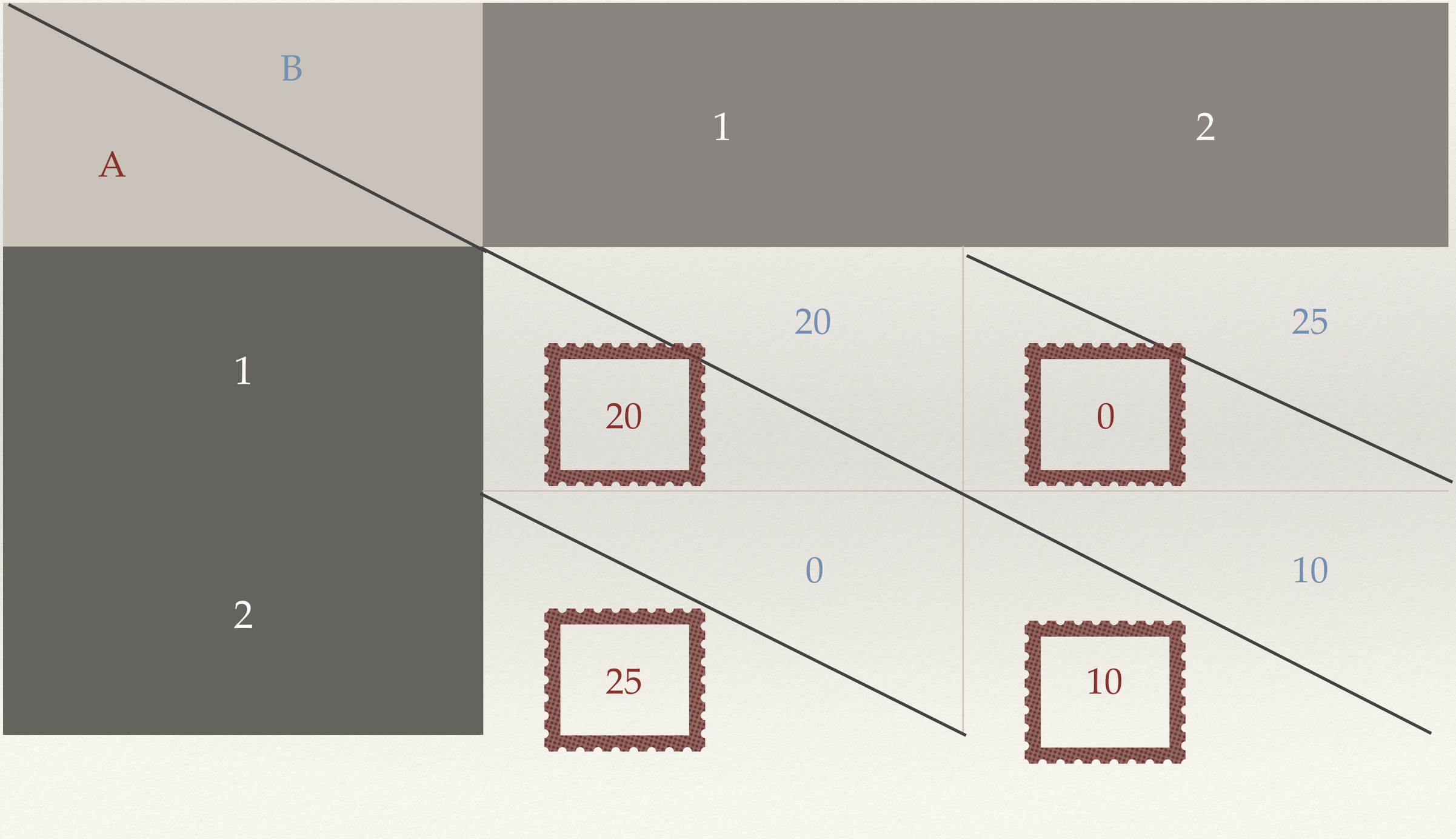
- Minimax: Smallest value player B can force player A to receive without knowing his actions:

$$\bar{V}_A = \min_{a_B} \max_{a_A} v_{a_A}(a_A, a_B)$$

max(0,10) = 10
A chooses 2

MinMax Example

min(25,10) = 10
B chooses 2



GANs

Algorithm 1 Minibatch stochastic gradient descent training of generative adversarial nets. The number of steps to apply to the discriminator, k, is a hyperparameter. We used $k = 1$, the least expensive option, in our experiments.

for number of training iterations **do**

  for k steps **do**

- Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
- Sample minibatch of m examples $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\boldsymbol{x})$.
- Update the discriminator by ascending its stochastic gradient:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D\left(\boldsymbol{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right) \right].$$

  end for

- Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
- Update the generator by descending its stochastic gradient:

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right).$$

end for

The gradient-based updates can use any standard gradient-based learning rule. We used momentum in our experiments.

Forger and Investigation Office

D: Investigator
(Discriminator)

G: Forger
(Generator)

X: Original
Data

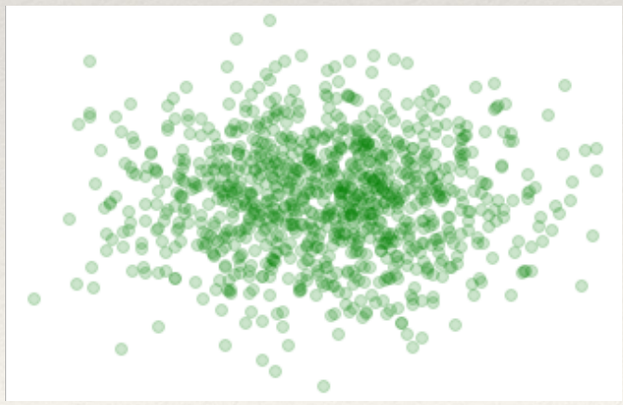
Z: Generator
Input

Forger and Investigation Office

D: Investigator (Discriminator)

G: Forger (Generator)

X: Original Data

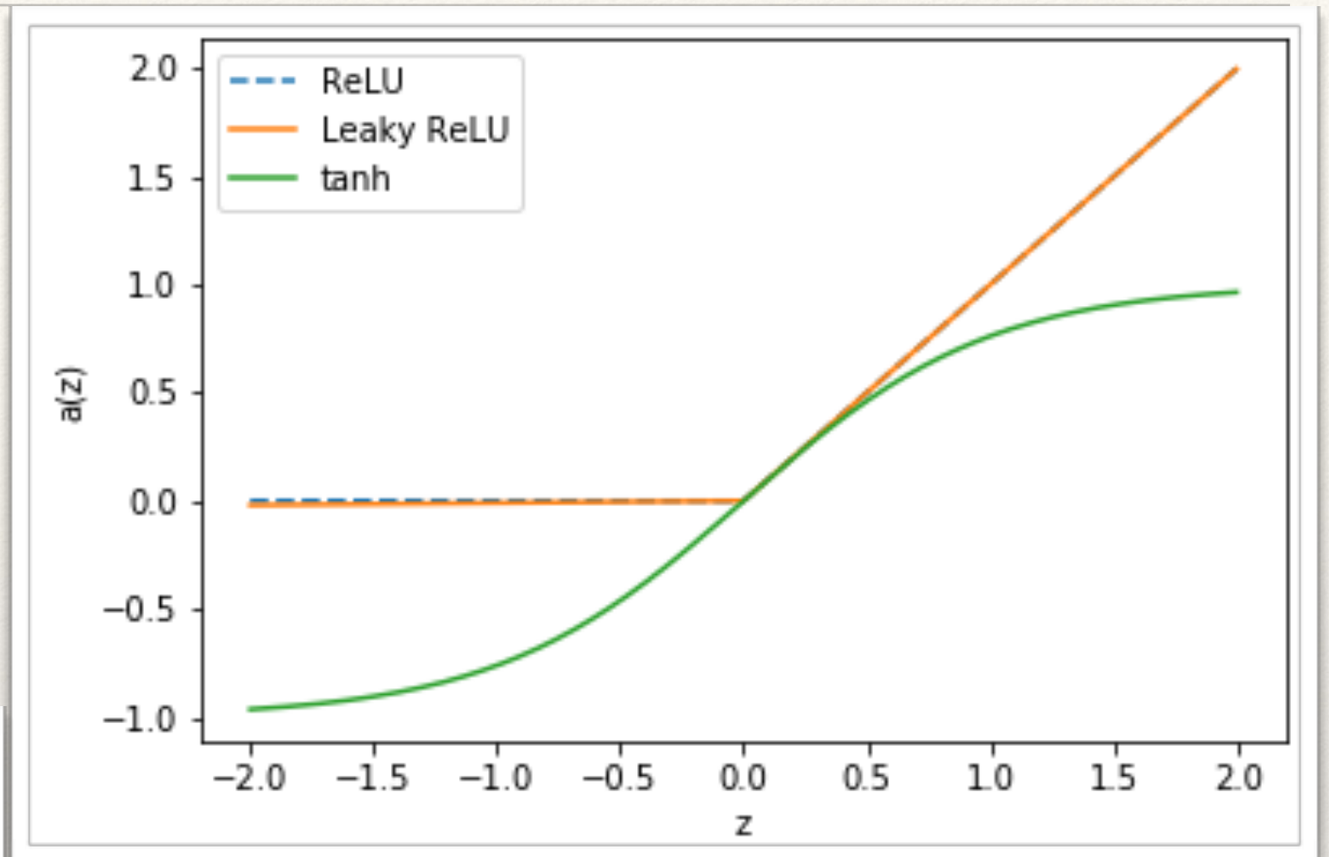


Z: Generator Input

Questions?

Why Rectified Linear Unit (ReLU)

- Computationally less expensive
- Reduces the likelihood of vanishing gradients
  - In many layers:
    - product of many smaller than one gradients gets closer to zero
- Increases the sparsity




```
def ReLU(x):
    return np.maximum(x, 0)

def leaky_ReLU(x):
    return np.maximum(x, 0.01*x)

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

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

arr = np.random.uniform(-2, 2, 1000)
arr.sort()
plt.plot(arr, ReLU(arr), label = 'ReLU', linestyle = '--')
plt.plot(arr, leaky_ReLU(arr), label = 'Leaky ReLU')
plt.plot(arr, tanh(arr), label = 'tanh')
plt.xlabel('z')
plt.ylabel('a(z)')
plt.legend()
plt.savefig('Activation_Functions.png')
plt.show()
```

Xavier initialization

- Initialize weights with values that are “not too small” or “not too large”
- Decreases the likelihood of explosion or vanishing gradients

```
def xavier_init(size):
    in_dim = size[0]
    # xavier variance is 1./in_dim
    # For ReLU it is 2./in_dim
    xavier_stddev = np.sqrt(2./in_dim)
    return Variable(torch.randn(*size) * xavier_stddev)
```

Adaptive Moment Estimation (Adam)

Algorithm 1: *Adam*, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation. g_t^2 indicates the elementwise square $g_t \odot g_t$. Good default settings for the tested machine learning problems are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. All operations on vectors are element-wise. With β_1^t and β_2^t we denote β_1 and β_2 to the power t.

Require: α: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters θ
Require: θ_0: Initial parameter vector

$m_0 \leftarrow 0$ (Initialize 1st moment vector)
$v_0 \leftarrow 0$ (Initialize 2nd moment vector)
$t \leftarrow 0$ (Initialize timestep)
while θ_t not converged **do**
  $t \leftarrow t + 1$
  $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)
  $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
  $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
  $\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
  $\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
  $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
end while
return θ_t (Resulting parameters)

Kingma, Diederik P., and Jimmy Ba.
"Adam: A method for stochastic optimization."
arXiv preprint arXiv:1412.6980 (2014).

Let's code