REGULARIZAÇÃO

DROPOUT

Zenilton K. G. Patrocínio Jr
zenilton@pucminas.br

Regularização: *Dropout*

[Srivastava et al., 2014]

"Defina aleatoriamente a saída de alguns neurônios para zero no *forward pass*"

isto é, multiplicar por variáveis aleatórias de Bernoulli com um probabilidade p.

Regularização: *Dropout*

[Srivastava et al., 2014]

“Defina aleatoriamente a saída de alguns neurônios para zero no *forward pass*”
isto é, multiplicar por variáveis aleatórias de Bernoulli com um probabilidade p.

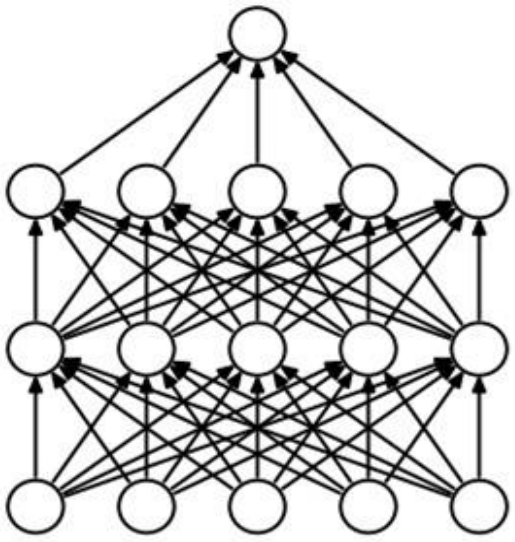
“A ideia principal é descartar aleatoriamente unidades (junto com suas conexões) da rede neural durante o treinamento.”

Regularização: *Dropout*

[Srivastava et al., 2014]

“Defina aleatoriamente a saída de alguns neurônios para zero no *forward pass*”
isto é, multiplicar por variáveis aleatórias de Bernoulli com um probabilidade p.

“A ideia principal é descartar aleatoriamente unidades (junto com suas conexões) da rede neural durante o treinamento.”



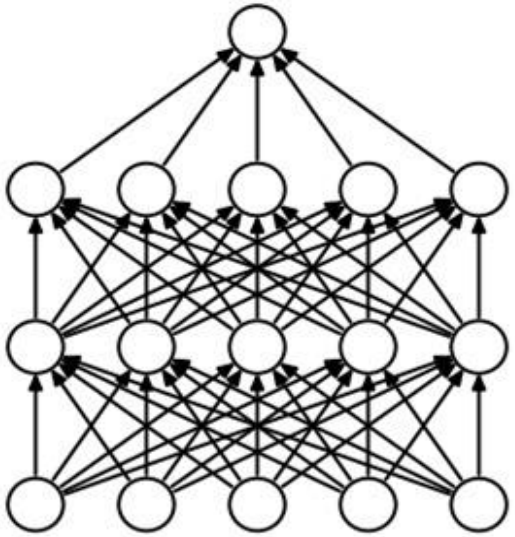
Rede Neural Original

Regularização: *Dropout*

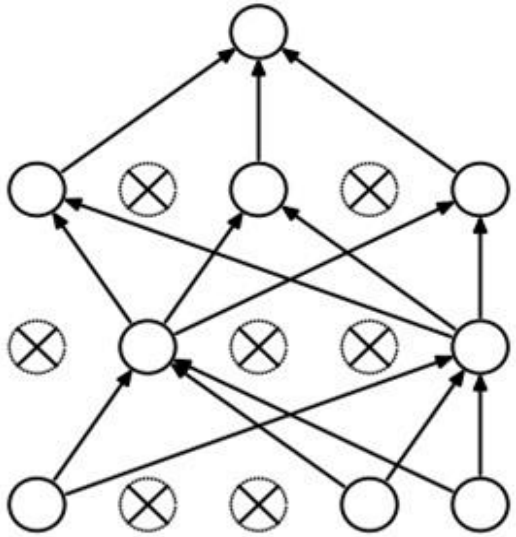
[Srivastava et al., 2014]

“Defina aleatoriamente a saída de alguns neurônios para zero no *forward pass*”
isto é, multiplicar por variáveis aleatórias de Bernoulli com um probabilidade p.

“A ideia principal é descartar aleatoriamente unidades (junto com suas conexões) da rede neural durante o treinamento.”



Rede Neural Original



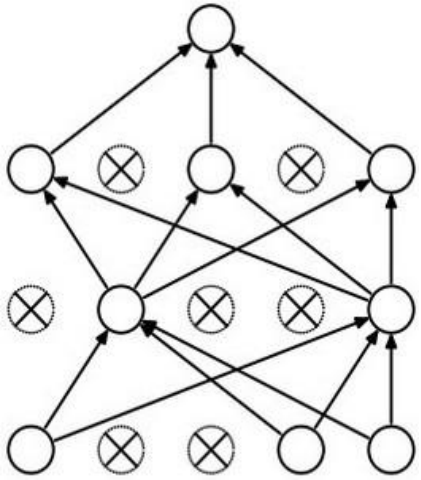
Rede após *Dropout*

Regularização: *Dropout*

```python
p = 0.5 # probability of keeping a unit active. higher = less dropout

def train_step(X):
  """ X contains the data """

  # forward pass for example 3-layer neural network
  H1 = np.maximum(0, np.dot(W1, X) + b1)
  U1 = np.random.rand(*H1.shape) < p # first dropout mask
  H1 *= U1 # drop!
  H2 = np.maximum(0, np.dot(W2, H1) + b2)
  U2 = np.random.rand(*H2.shape) < p # second dropout mask
  H2 *= U2 # drop!
  out = np.dot(W3, H2) + b3

  # backward pass: compute gradients... (not shown)
  # perform parameter update... (not shown)
```

Exemplo

Forward pass em uma rede de 3 camadas com *dropout*

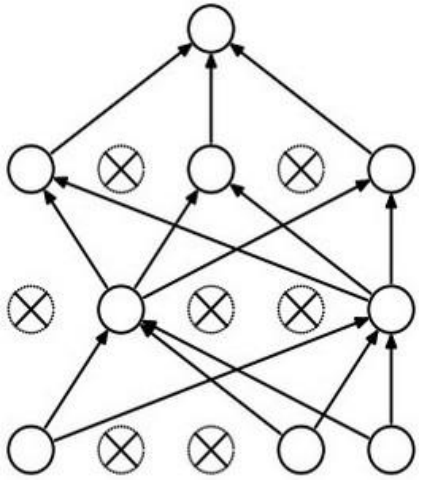


Regularização: *Dropout*

```python
p = 0.5 # probability of keeping a unit active. higher = less dropout

def train_step(X):
  """ X contains the data """

  # forward pass for example 3-layer neural network
  H1 = np.maximum(0, np.dot(W1, X) + b1)
  U1 = np.random.rand(*H1.shape) < p # first dropout mask
  H1 *= U1 # drop!
  H2 = np.maximum(0, np.dot(W2, H1) + b2)
  U2 = np.random.rand(*H2.shape) < p # second dropout mask
  H2 *= U2 # drop!
  out = np.dot(W3, H2) + b3

  # backward pass: compute gradients... (not shown)
  # perform parameter update... (not shown)
```

Exemplo

Forward pass em uma rede de 3 camadas com *dropout*

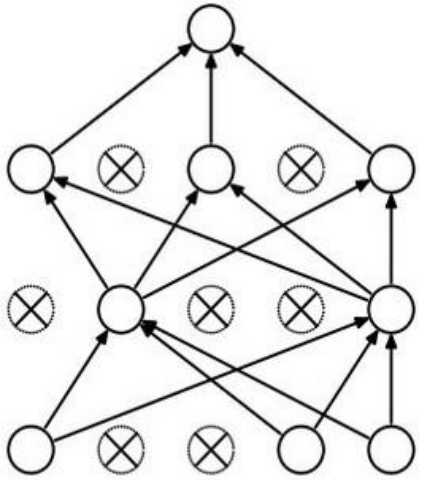


Regularização: *Dropout*

```
p = 0.5 # probability of keeping a unit active. higher = less dropout

def train_step(X):
  """ X contains the data """

  # forward pass for example 3-layer neural network
  H1 = np.maximum(0, np.dot(W1, X) + b1)
  U1 = np.random.rand(*H1.shape) < p # first dropout mask
  H1 *= U1 # drop!
  H2 = np.maximum(0, np.dot(W2, H1) + b2)
  U2 = np.random.rand(*H2.shape) < p # second dropout mask
  H2 *= U2 # drop!
  out = np.dot(W3, H2) + b3

  # backward pass: compute gradients... (not shown)
  # perform parameter update... (not shown)
```

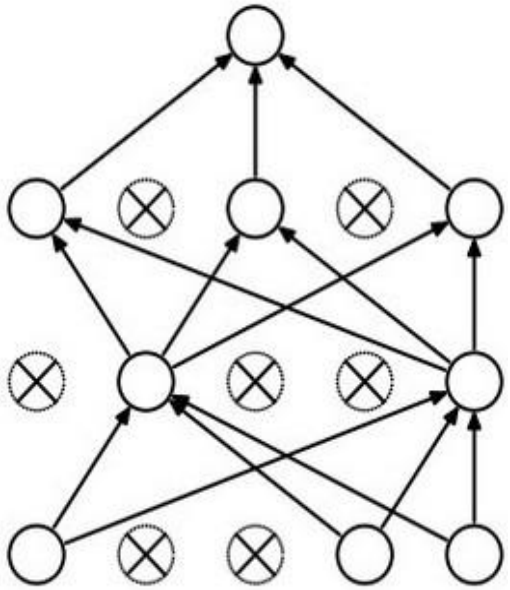
Exemplo

Forward pass em uma rede de 3 camadas com *dropout*



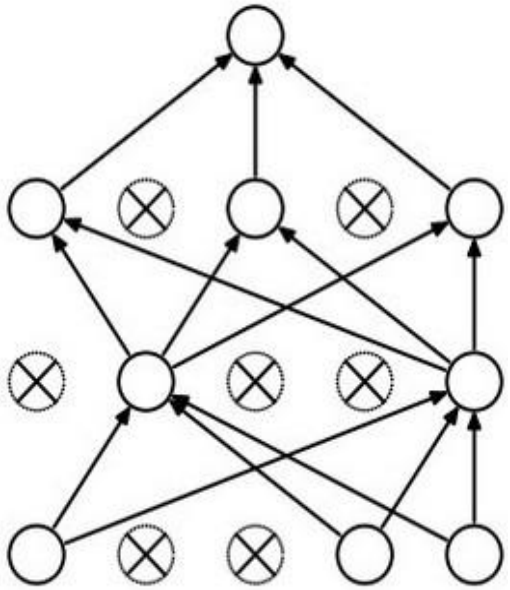
Regularização: *Dropout*

Qual a razão para isso ser uma “boa” ideia?

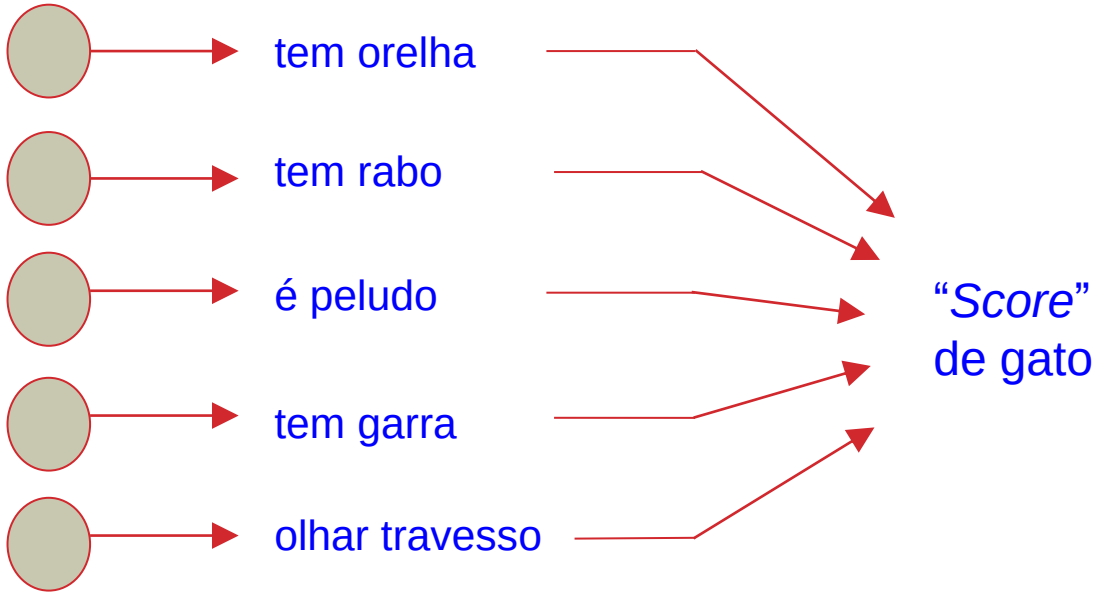


Regularização: *Dropout*

Qual a razão para isso ser uma “boa” ideia?

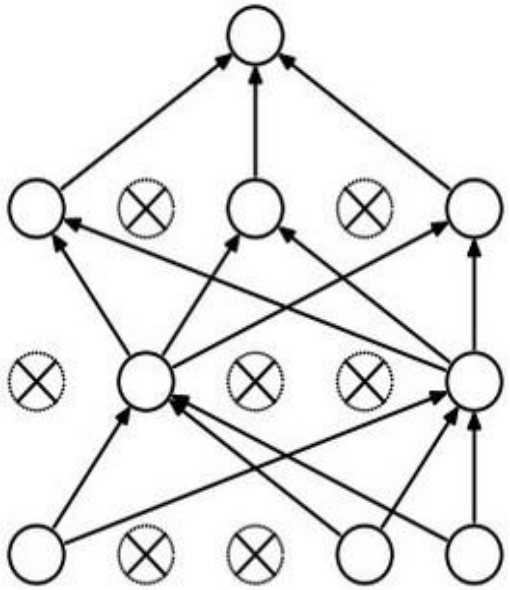


Força a rede a construir uma representação distribuida e redundante



Regularização: *Dropout*

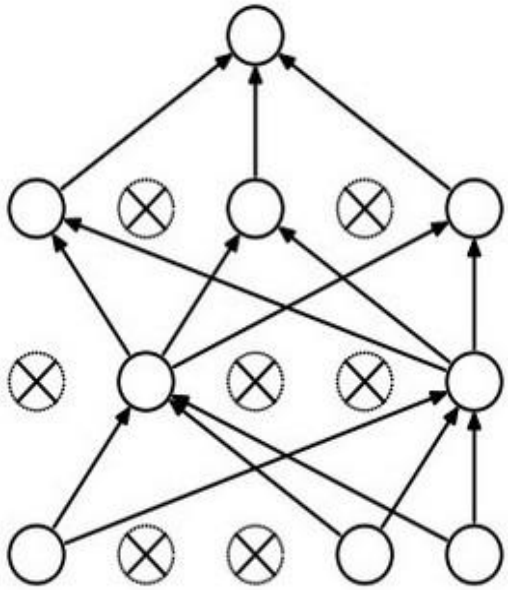
Qual a razão para isso ser uma “boa” ideia?



Força a rede a construir uma representação distribuida e redundante

Regularização: *Dropout*

Qual a razão para isso ser uma “boa” ideia?

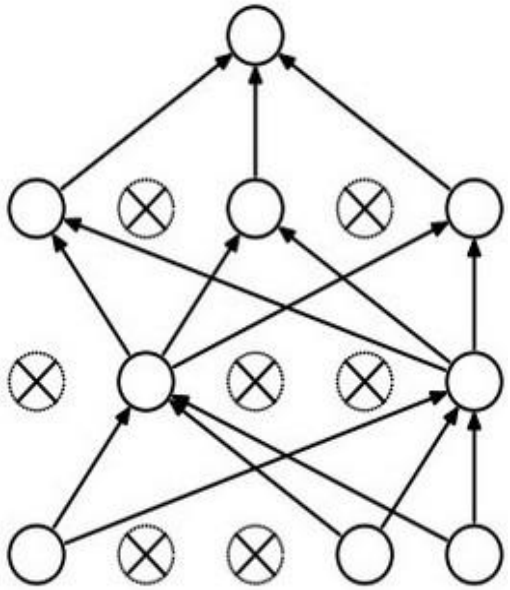


Outra interpretação:

Dropout equivale a treinar uma grande coleção de modelos que compartilham parâmetros – ***ensemble***

Regularização: *Dropout*

Qual a razão para isso ser uma “boa” ideia?



Outra interpretação:

Dropout equivale a treinar uma grande coleção de modelos que compartilham parâmetros – ***ensemble***

Cada máscara binária corresponde a um modelo, que é treinada em apenas uma parte dos dados (um *minibatch*)

Regularização: *Dropout*

Durante a predição…

Todos os neurônios são mantidos ativos

Regularização: *Dropout*

Durante a predição…

Todos os neurônios são mantidos ativos

$\Rightarrow$ Deve-se ajustar as ativações de forma que para cada neurônio:

$$\frac{\text{Saída na predição}}{} = \frac{\text{Expectativa de saída no treinamento}}{}$$

Regularização: *Dropout*

Durante a predição…

Todos os neurônios são mantidos ativos

$\Rightarrow$ Deve-se ajustar as ativações de forma que para cada neurônio:

Saída na predição = Expectativa de saída no treinamento $\rightarrow$ **Inviável !**

Regularização: *Dropout*

Durante a predição…

Todos os neurônios são mantidos ativos

⇒ Deve-se ajustar as ativações de forma que para cada neurônio:

Saída na predição = Expectativa de saída no treinamento ⟶ **Inviável !**

```
def predict(X):
  # ensembled forward pass
  H1 = np.maximum(0, np.dot(W1, X) + b1) * p # NOTE: scale the activations
  H2 = np.maximum(0, np.dot(W2, H1) + b2) * p # NOTE: scale the activations
  out = np.dot(W3, H2) + b3
```

Regularização: *Dropout*

Durante a predição…

Todos os neurônios são mantidos ativos

⇒ Deve-se ajustar as ativações de forma que para cada neurônio:

Saída na predição = Expectativa de saída no treinamento ⟶ **Inviável !**

```
def predict(X):
  # ensembled forward pass
  H1 = np.maximum(0, np.dot(W1, X) + b1) * p # NOTE: scale the activations
  H2 = np.maximum(0, np.dot(W2, H1) + b2) * p # NOTE: scale the activations
  out = np.dot(W3, H2) + b3
```

Solução heurística

Regularização: *Dropout* – Resumo

```
""" Vanilla Dropout: Not recommended implementation (see notes below) """

p = 0.5 # probability of keeping a unit active. higher = less dropout

def train_step(X):
  """ X contains the data """

  # forward pass for example 3-layer neural network
  H1 = np.maximum(0, np.dot(W1, X) + b1)
  U1 = np.random.rand(*H1.shape) < p # first dropout mask
  H1 *= U1 # drop!
  H2 = np.maximum(0, np.dot(W2, H1) + b2)
  U2 = np.random.rand(*H2.shape) < p # second dropout mask
  H2 *= U2 # drop!
  out = np.dot(W3, H2) + b3

  # backward pass: compute gradients... (not shown)
  # perform parameter update... (not shown)

def predict(X):
  # ensembled forward pass
  H1 = np.maximum(0, np.dot(W1, X) + b1) * p # NOTE: scale the activations
  H2 = np.maximum(0, np.dot(W2, H1) + b2) * p # NOTE: scale the activations
  out = np.dot(W3, H2) + b3
```

Regularização: *Dropout* – Resumo

```python
""" Vanilla Dropout: Not recommended implementation (see notes below) """

p = 0.5 # probability of keeping a unit active. higher = less dropout

def train_step(X):
  """ X contains the data """

  # forward pass for example 3-layer neural network
  H1 = np.maximum(0, np.dot(W1, X) + b1)
  U1 = np.random.rand(*H1.shape) < p # first dropout mask
  H1 *= U1 # drop!
  H2 = np.maximum(0, np.dot(W2, H1) + b2)
  U2 = np.random.rand(*H2.shape) < p # second dropout mask
  H2 *= U2 # drop!
  out = np.dot(W3, H2) + b3

  # backward pass: compute gradients... (not shown)
  # perform parameter update... (not shown)

def predict(X):
  # ensembled forward pass
  H1 = np.maximum(0, np.dot(W1, X) + b1) * p # NOTE: scale the activations
  H2 = np.maximum(0, np.dot(W2, H1) + b2) * p # NOTE: scale the activations
  out = np.dot(W3, H2) + b3
```

"Drop" no *forward pass*

Regularização: *Dropout* – Resumo

```
""" Vanilla Dropout: Not recommended implementation (see notes below) """

p = 0.5 # probability of keeping a unit active. higher = less dropout

def train_step(X):
  """ X contains the data """

  # forward pass for example 3-layer neural network
  H1 = np.maximum(0, np.dot(W1, X) + b1)
  U1 = np.random.rand(*H1.shape) < p # first dropout mask
  H1 *= U1 # drop!
  H2 = np.maximum(0, np.dot(W2, H1) + b2)
  U2 = np.random.rand(*H2.shape) < p # second dropout mask
  H2 *= U2 # drop!
  out = np.dot(W3, H2) + b3

  # backward pass: compute gradients... (not shown)
  # perform parameter update... (not shown)

def predict(X):
  # ensembled forward pass
  H1 = np.maximum(0, np.dot(W1, X) + b1) * p # NOTE: scale the activations
  H2 = np.maximum(0, np.dot(W2, H1) + b2) * p # NOTE: scale the activations
  out = np.dot(W3, H2) + b3
```

"Drop" no *forward pass*

Ajuste durante a predição

Regularização: *Inverted Dropout*

```python
p = 0.5 # probability of keeping a unit active. higher = less dropout

def train_step(X):
  # forward pass for example 3-layer neural network
  H1 = np.maximum(0, np.dot(W1, X) + b1)
  U1 = (np.random.rand(*H1.shape) < p) / p # first dropout mask. Notice /p!
  H1 *= U1 # drop!
  H2 = np.maximum(0, np.dot(W2, H1) + b2)
  U2 = (np.random.rand(*H2.shape) < p) / p # second dropout mask. Notice /p!
  H2 *= U2 # drop!
  out = np.dot(W3, H2) + b3

  # backward pass: compute gradients... (not shown)
  # perform parameter update... (not shown)

def predict(X):
  # ensembled forward pass
  H1 = np.maximum(0, np.dot(W1, X) + b1) # no scaling necessary
  H2 = np.maximum(0, np.dot(W2, H1) + b2)
  out = np.dot(W3, H2) + b3
```

Fazer uma divisão por **p**

Regularização: *Inverted Dropout*

```python
p = 0.5 # probability of keeping a unit active. higher = less dropout

def train_step(X):
  # forward pass for example 3-layer neural network
  H1 = np.maximum(0, np.dot(W1, X) + b1)
  U1 = (np.random.rand(*H1.shape) < p) / p # first dropout mask. Notice /p!
  H1 *= U1 # drop!
  H2 = np.maximum(0, np.dot(W2, H1) + b2)
  U2 = (np.random.rand(*H2.shape) < p) / p # second dropout mask. Notice /p!
  H2 *= U2 # drop!
  out = np.dot(W3, H2) + b3

  # backward pass: compute gradients... (not shown)
  # perform parameter update... (not shown)

def predict(X):
  # ensembled forward pass
  H1 = np.maximum(0, np.dot(W1, X) + b1) # no scaling necessary
  H2 = np.maximum(0, np.dot(W2, H1) + b2)
  out = np.dot(W3, H2) + b3
```

Fazer uma divisão por **p**

Predição fica inalterada!