Optimization and Generalization

This lecture covers

- Gradient descent
- Backpropagation
- Improved optimization: momentum, RMS-Prop, Adam
- Initialization
- Pragmatics of training neural networks
- Hyperparameter tuning

Gradient Descent

- If the domains are continuous, Gradient descent movies each each variable downhill; proportional to the gradient of the heuristic function in that direction.
  The value of variable X_i goes from v_i to

Gradient Descent

- If the domains are continuous, Gradient descent movies each each variable downhill; proportional to the gradient of the heuristic function in that direction.
  The value of variable X_i goes from v_i to $v_i - \eta \frac{\partial h}{\partial X_i}$.
  η is the step size.

Gradient Descent

- If the domains are continuous, Gradient descent movies each each variable downhill; proportional to the gradient of the heuristic function in that direction.
  The value of variable X_i goes from v_i to $v_i - \eta \frac{\partial h}{\partial X_i}$.
  η is the step size.
- Neural networks do gradient descent with many parameters (variables) to minimize an error on a dataset. Some large language models have over 10^{12} parameters.

Differentiation

Two properties of differentiation are used in backpropagation:

- Linear rule: the derivative of a linear function, $aw + b$, is given by:

$$\frac{\partial}{\partial w}(aw + b) = a$$

- Chain rule: if g is a function of w and function f, which does not depend on w, is applied to $g(w)$, then

$$\frac{\partial}{\partial w} f(g(w)) = f'(g(w)) * \frac{\partial}{\partial w} g(w)$$

where f' is the derivative of f.

Use of chain rule

A network represents $f(e) = f_n(f_{n-1}(\ldots f_2(f_1(x_e))))$, where example e has features x_e. Suppose $v_i = f_i(v_{i-1})$ and $v_0 = x_e$. Consider weight w used in the definition of f_j:

$$\begin{aligned}
&\frac{\partial}{\partial w} error(f(e)) \\
&\quad = error'(v_n) * \frac{\partial}{\partial w} f_n(v_{n-1}) \\
&\quad = error'(v_n) * \frac{\partial}{\partial w} f_n(f_{n-1}(v_{n-2})) \\
&\quad = error'(v_n) * f_n'(v_{n-1}) * \frac{\partial}{\partial w}(f_{n-1}(v_{n-2})) \\
&\quad = error'(v_n) * f_n'(v_{n-1}) * f_{n-1}'(v_{n-2}) * \cdots * \frac{\partial}{\partial w}(f_j(v_{j-1}))
\end{aligned}$$

where f_i' is the derivative of f_i with respect to its inputs.

Backpropagation

- Backpropagation implements (stochastic) gradient descent for all weights.
- Two passes:
  - Prediction: given inputs compute outputs of each layer
  - Back propagate: Going backwards,

$$error'(v_n) * \prod_{i=0}^{k} f'_{n-i}(v_{n-i-1})$$

    for k starting from 0 are computed and passed to the lower layers. Weights in each layer are updated.

Neural-network learner

functions is the list of functions that compose the neural network.

repeat
    batch := random sample of *batch_size* examples
    for each example *e* in *batch* **do**
        for each input unit *i* **do** $values[i] := X_i(e)$
        for each *fun* in *functions* from lowest to highest **do**
            $values := fun.output(values)$
        for each output unit *j* **do**
$error[j] := \phi_o(values[j]) - Ys[j]$
        for each *fun* in *functions* from highest to lowest **do**
            $error := fun.Backprop(error)$
    for each *fun* in *functions* that contains weights **do**
        $fun.update()$
until termination

Neural-network learner

functions is the list of functions that compose the neural network.

repeat
    batch := random sample of *batch_size* examples
    for each example *e* in *batch* **do**
        for each input unit *i* **do** $values[i] := X_i(e)$
        for each *fun* in *functions* from lowest to highest **do**
            $values := fun.output(values)$
        for each output unit *j* **do**
$error[j] := \phi_o(values[j]) - Ys[j]$
        for each *fun* in *functions* from highest to lowest **do**
            $error := fun.Backprop(error)$
    for each *fun* in *functions* that contains weights **do**
        $fun.update()$
until termination

Neural-network learner

functions is the list of functions that compose the neural network.

```
repeat
    batch := random sample of batch_size examples
    for each example e in batch do
        for each input unit i do values[i] := X_i(e)
        for each fun in functions from lowest to highest do
            values := fun.output(values)
        for each output unit j do
error[j] := φ_o(values[j]) − Ys[j]
        for each fun in functions from highest to lowest do
            error := fun.Backprop(error)
    for each fun in functions that contains weights do
        fun.update()
until termination
```

Neural-network learner

functions is the list of functions that compose the neural network.

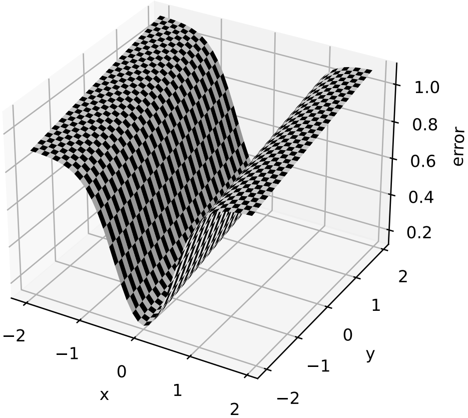
repeat
    batch := random sample of *batch_size* examples
    for each example *e* in *batch* **do**
        for each input unit *i* **do** $values[i] := X_i(e)$
        for each *fun* in *functions* from lowest to highest **do**
            $values := fun.output(values)$
        for each output unit *j* **do**
$error[j] := \phi_o(values[j]) - Ys[j]$
        for each *fun* in *functions* from highest to lowest **do**
            $error := fun.Backprop(error)$
    for each *fun* in *functions* that contains weights **do**
        $fun.update()$
until termination

Dense linear function

```
class Dense(n_i, n_o)                         ▷ n_i is # inputs, n_o is #outputs
    for each 0 ≤ i ≤ n_i and each 0 ≤ j < n_o do
        d[i, j] := 0; w[i, j] := a random value
    def output(in)                            ▷ in is array with length n_i
        for each j do out[j] := w[n_i, j] + ∑_i in[i] ∗ w[i, j]
        return out
    def Backprop(error)                       ▷ error is array with length n_o
        for each i, j do d[i, j] := d[i, j] + in[i] ∗ error[j]
        for each i do ierror[i] := ∑_j w[i, j] ∗ error[j]
        return ierror
    def update()                              ▷ update weights. η is learning rate.
        for each i, j do
            w[i, j] := w[i, j] − η/batch_size ∗ d[i, j]
            d[i, j] := 0
```

Problems for (stochastic) gradient descent

Error as a function of parameters x and y:



Want different step sizes for x and y.
With many parameters, treat each parameter independently.

Momentum

- The momentum for a parameter acts like a velocity of the step size. The standard stochastic gradient descent update acts like an acceleration.

Momentum

- The momentum for a parameter acts like a velocity of the step size. The standard stochastic gradient descent update acts like an acceleration.
- For updates are in the same direction, the step size

Momentum

- The momentum for a parameter acts like a velocity of the step size. The standard stochastic gradient descent update acts like an acceleration.
- For updates are in the same direction, the step size increases.

Momentum

- The momentum for a parameter acts like a velocity of the step size. The standard stochastic gradient descent update acts like an acceleration.
- For updates are in the same direction, the step size increases.
- For updates are in opposite direction, the step size

Momentum

- The momentum for a parameter acts like a velocity of the step size. The standard stochastic gradient descent update acts like an acceleration.
- For updates are in the same direction, the step size increases.
- For updates are in opposite direction, the step size decreases.

Momentum

- The momentum for a parameter acts like a velocity of the step size. The standard stochastic gradient descent update acts like an acceleration.
- For updates are in the same direction, the step size increases.
- For updates are in opposite direction, the step size decreases.
- For a dense layer, the update becomes:

```
def update()                                        ▷ update all weights
    for each i, j do
        v[i,j] := α ∗ v[i,j] − η/batch_size ∗ d[i,j]
        w[i,j] := w[i,j] + v[i,j]
        d[i,j] := 0.
```

Hyperparameter α, with $0 \leq \alpha < 1$, specifies how much of the momentum should be used.

Momentum

- The momentum for a parameter acts like a velocity of the step size. The standard stochastic gradient descent update acts like an acceleration.
- For updates are in the same direction, the step size increases.
- For updates are in opposite direction, the step size decreases.
- For a dense layer, the update becomes:

  def *update*() ▷ update all weights
      for each i, j **do**
          $v[i,j] := \alpha * v[i,j] - \eta / batch\_size * d[i,j]$
          $w[i,j] := w[i,j] + v[i,j]$
          $d[i,j] := 0.$

  Hyperparameter α, with $0 \leq \alpha < 1$, specifies how much of the momentum should be used.
- The SDG update method is equivalent to this with

Momentum

- The momentum for a parameter acts like a velocity of the step size. The standard stochastic gradient descent update acts like an acceleration.
- For updates are in the same direction, the step size increases.
- For updates are in opposite direction, the step size decreases.
- For a dense layer, the update becomes:

```
def update()                                        ▷ update all weights
    for each i, j do
        v[i,j] := α ∗ v[i,j] − η/batch_size ∗ d[i,j]
        w[i,j] := w[i,j] + v[i,j]
        d[i,j] := 0.
```

  Hyperparameter α, with $0 \leq \alpha < 1$, specifies how much of the momentum should be used.
- The SDG update method is equivalent to this with $\alpha = 0$.

Momentum

- The momentum for a parameter acts like a velocity of the step size. The standard stochastic gradient descent update acts like an acceleration.
- For updates are in the same direction, the step size increases.
- For updates are in opposite direction, the step size decreases.
- For a dense layer, the update becomes:

```
def update()                                        ▷ update all weights
    for each i, j do
        v[i,j] := α * v[i,j] − η/batch_size * d[i,j]
        w[i,j] := w[i,j] + v[i,j]
        d[i,j] := 0.
```

  Hyperparameter α, with $0 \leq \alpha < 1$, specifies how much of the momentum should be used.
- The SDG update method is equivalent to this with $\alpha = 0$.
- What happens in the canyon?

RMS-Prop

- In RMS-Prop the magnitude of the change in a weight depends on how its gradient compares to its historic value.
- It maintains r, a rolling average of the square of the gradient.

RMS-Prop

- In RMS-Prop the magnitude of the change in a weight depends on how its gradient compares to its historic value.
- It maintains r, a rolling average of the square of the gradient.
- For a dense layer, the update becomes:

1: **def** *update*() ▷ update weights
2: **for each** i, j **do**
3: $g := d[i,j]/batch\_size$
4: $r[i,j] := \rho * r[i,j] + (1-\rho) * g^2$
5: $w[i,j] := w[i,j] - \frac{\eta * g}{\sqrt{r[i,j] + \epsilon}}$
6: $d[i,j] := 0$.

- ϵ ($\approx 10^{-7}$) is used to ensure numerical stability.

RMS-Prop

- In RMS-Prop the magnitude of the change in a weight depends on how its gradient compares to its historic value.
- It maintains r, a rolling average of the square of the gradient.
- For a dense layer, the update becomes:

```
def update()                                   ▷ update weights
    for each i, j do
        g := d[i,j]/batch_size
        r[i,j] := ρ * r[i,j] + (1 − ρ) * g^2
        w[i,j] := w[i,j] − (η * g) / sqrt(r[i,j] + ϵ)
        d[i,j] := 0 .
```

$$g := d[i,j]/batch\_size$$
$$r[i,j] := \rho * r[i,j] + (1-\rho) * g^2$$
$$w[i,j] := w[i,j] - \frac{\eta * g}{\sqrt{r[i,j] + \epsilon}}$$
$$d[i,j] := 0$$

- ϵ ($\approx 10^{-7}$) is used to ensure numerical stability.

RMS-Prop

- In RMS-Prop the magnitude of the change in a weight depends on how its gradient compares to its historic value.
- It maintains r, a rolling average of the square of the gradient.
- For a dense layer, the update becomes:

```
def update()                                        ▷ update weights
    for each i, j do
        g := d[i,j]/batch_size
        r[i,j] := ρ * r[i,j] + (1 − ρ) * g^2
        w[i,j] := w[i,j] − (η * g) / sqrt(r[i,j] + ϵ)
        d[i,j] := 0 .
```

- ϵ ($\approx 10^{-7}$) is used to ensure numerical stability.
- When $r[i,j] \approx g^2 \gg \epsilon$, the ratio $g/\sqrt{r[i,j]+\epsilon}$ is

RMS-Prop

- In RMS-Prop the magnitude of the change in a weight depends on how its gradient compares to its historic value.
- It maintains r, a rolling average of the square of the gradient.
- For a dense layer, the update becomes:

```
def update()                                   ▷ update weights
    for each i, j do
        g := d[i,j]/batch_size
        r[i,j] := ρ * r[i,j] + (1 − ρ) * g^2
        w[i,j] := w[i,j] − (η * g) / sqrt(r[i,j] + ϵ)
        d[i,j] := 0 .
```

$$r[i,j] := \rho * r[i,j] + (1-\rho) * g^2$$

$$w[i,j] := w[i,j] - \frac{\eta * g}{\sqrt{r[i,j] + \epsilon}}$$

- ϵ ($\approx 10^{-7}$) is used to ensure numerical stability.
- When $r[i,j] \approx g^2 \gg \epsilon$, the ratio $g/\sqrt{r[i,j]+\epsilon}$ is approximately 1 or -1, depending on the sign of g.
- When $g^2 \ll r[i,j]$,

RMS-Prop

- In RMS-Prop the magnitude of the change in a weight depends on how its gradient compares to its historic value.
- It maintains r, a rolling average of the square of the gradient.
- For a dense layer, the update becomes:

```
1: def update()                                   ▷ update weights
2:     for each i, j do
3:         g := d[i,j]/batch_size
4:         r[i,j] := ρ * r[i,j] + (1 − ρ) * g^2
5:         w[i,j] := w[i,j] − (η * g) / sqrt(r[i,j] + ε)
6:         d[i,j] := 0 .
```

- ϵ ($\approx 10^{-7}$) is used to ensure numerical stability.
- When $r[i,j] \approx g^2 \gg \epsilon$, the ratio $g/\sqrt{r[i,j]+\epsilon}$ is approximately 1 or -1, depending on the sign of g.
- When $g^2 \ll r[i,j]$, the step size is smaller than η.

Adam

- Adam, for "adaptive moments", uses both momentum and the square of the gradient.

Adam

- Adam, for "adaptive moments", uses both momentum and the square of the gradient.
- It corrects to account for the parameters being initialized to 0.

Adam

- Adam, for "adaptive moments", uses both momentum and the square of the gradient.
- It corrects to account for the parameters being initialized to 0.

def *update*() ▷ update weights
    $t := t + 1$ ▷ t is initially 0
    for each i, j **do**
        $g := d[i,j]/batch\_size$
        $s[i,j] := \beta_1 * s[i,j] + (1 - \beta_1) * g$
        $r[i,j] := \beta_2 * r[i,j] + (1 - \beta_2) * g^2$
        $w[i,j] := w[i,j] - \dfrac{\eta * s[i,j]/(1 - \beta_1^t)}{\sqrt{r[i,j]/(1 - \beta_2^t)} + \epsilon}$
        $d[i,j] := 0.$

- s acts as momentum (initially 0)
- r is the rolling average of the square (initially 0).

Adam

- Adam, for "adaptive moments", uses both momentum and the square of the gradient.
- It corrects to account for the parameters being initialized to 0.

def *update*() ▷ update weights

  $t := t + 1$ ▷ t is initially 0

  for each i, j **do**

    $g := d[i,j]/batch\_size$

    $s[i,j] := \beta_1 * s[i,j] + (1 - \beta_1) * g$

    $r[i,j] := \beta_2 * r[i,j] + (1 - \beta_2) * g^2$

    $$w[i,j] := w[i,j] - \frac{\eta * s[i,j]/(1 - \beta_1^t)}{\sqrt{r[i,j]/(1 - \beta_2^t)} + \epsilon}$$

    $d[i,j] := 0.$

- s acts as momentum (initially 0)
- r is the rolling average of the square (initially 0).
- What happens at first step (when t becomes 1)?

Initialization

- Real-valued variables are normalized by subtracting the mean, and dividing by the standard deviation.

Initialization

- Real-valued variables are normalized by subtracting the mean, and dividing by the standard deviation.

Initialization

- Real-valued variables are normalized by subtracting the mean, and dividing by the standard deviation.
- In a one-hot encoding, categorical input variable X with domain $\{v_1, \ldots, v_k\}$ is represented as k input indicator variables, $X_1, \ldots, X_k$. An input example with $X = v_j$ is represented with $X_j = 1$ and every other $X_{j'} = 0$.

Initialization

- Real-valued variables are normalized by subtracting the mean, and dividing by the standard deviation.
- In a one-hot encoding, categorical input variable X with domain $\{v_1, \ldots, v_k\}$ is represented as k input indicator variables, $X_1, \ldots, X_k$. An input example with $X = v_j$ is represented with $X_j = 1$ and every other $X_{j'} = 0$.
- What happens if the weights in the hidden layers are all set to the same value?

Initialization

- Real-valued variables are normalized by subtracting the mean, and dividing by the standard deviation.
- In a one-hot encoding, categorical input variable X with domain $\{v_1, \ldots, v_k\}$ is represented as k input indicator variables, $X_1, \ldots, X_k$. An input example with $X = v_j$ is represented with $X_j = 1$ and every other $X_{j'} = 0$.
- What happens if the weights in the hidden layers are all set to the same value?
- For the output units, non-bias weights can be set to zero and the bias weights to the mean for regression or inverse-sigmoid of the empirical probability for classification. (Why?)

Pragmatics of Training Neural Networks

- Make sure it is learning something: The error on the training set should beat a naive baseline corresponding to the loss being evaluated.

Pragmatics of Training Neural Networks

- Make sure it is learning something: The error on the training set should beat a naive baseline corresponding to the loss being evaluated.
- If the performance on the training set is poor, change the model.

Pragmatics of Training Neural Networks

- Make sure it is learning something: The error on the training set should beat a naive baseline corresponding to the loss being evaluated.
- If the performance on the training set is poor, change the model.
  (Poor performance on the training set indicates

Pragmatics of Training Neural Networks

- Make sure it is learning something: The error on the training set should beat a naive baseline corresponding to the loss being evaluated.
- If the performance on the training set is poor, change the model.
  (Poor performance on the training set indicates under-fitting.)

Pragmatics of Training Neural Networks

- Make sure it is learning something: The error on the training set should beat a naive baseline corresponding to the loss being evaluated.
- If the performance on the training set is poor, change the model.
  (Poor performance on the training set indicates under-fitting.)
- Test the error on the validation set. If the validation error does not improve as the algorithm proceeds, it means the learning is not generalizing, and it is fitting to noise.

Pragmatics of Training Neural Networks

- Make sure it is learning something: The error on the training set should beat a naive baseline corresponding to the loss being evaluated.
- If the performance on the training set is poor, change the model.
  (Poor performance on the training set indicates under-fitting.)
- Test the error on the validation set. If the validation error does not improve as the algorithm proceeds, it means the learning is not generalizing, and it is fitting to noise.
  (Poor performance on the validation set indicates

Pragmatics of Training Neural Networks

- Make sure it is learning something: The error on the training set should beat a naive baseline corresponding to the loss being evaluated.
- If the performance on the training set is poor, change the model.
  (Poor performance on the training set indicates under-fitting.)
- Test the error on the validation set. If the validation error does not improve as the algorithm proceeds, it means the learning is not generalizing, and it is fitting to noise.
  (Poor performance on the validation set indicates overfitting.)

Pragmatics of Training Neural Networks

- Make sure it is learning something: The error on the training set should beat a naive baseline corresponding to the loss being evaluated.
- If the performance on the training set is poor, change the model.
  (Poor performance on the training set indicates under-fitting.)
- Test the error on the validation set. If the validation error does not improve as the algorithm proceeds, it means the learning is not generalizing, and it is fitting to noise.
  (Poor performance on the validation set indicates overfitting.)
  In this case you should

Pragmatics of Training Neural Networks

- Make sure it is learning something: The error on the training set should beat a naive baseline corresponding to the loss being evaluated.
- If the performance on the training set is poor, change the model.
  (Poor performance on the training set indicates under-fitting.)
- Test the error on the validation set. If the validation error does not improve as the algorithm proceeds, it means the learning is not generalizing, and it is fitting to noise.
  (Poor performance on the validation set indicates overfitting.)
  In this case you should simplify the model.

Pragmatics of Training Neural Networks

- Make sure it is learning something: The error on the training set should beat a naive baseline corresponding to the loss being evaluated.
- If the performance on the training set is poor, change the model.
  (Poor performance on the training set indicates under-fitting.)
- Test the error on the validation set. If the validation error does not improve as the algorithm proceeds, it means the learning is not generalizing, and it is fitting to noise.
  (Poor performance on the validation set indicates overfitting.)
  In this case you should simplify the model.
- Then carry out hyperparameter tuning.

Pragmatics of Training Neural Networks

- Make sure it is learning something: The error on the training set should beat a naive baseline corresponding to the loss being evaluated.
- If the performance on the training set is poor, change the model.
  (Poor performance on the training set indicates under-fitting.)
- Test the error on the validation set. If the validation error does not improve as the algorithm proceeds, it means the learning is not generalizing, and it is fitting to noise.
  (Poor performance on the validation set indicates overfitting.)
  In this case you should simplify the model.
- Then carry out hyperparameter tuning.
- If the performance isn't adequate, try to collect more data!

Pragmatics of Training Neural Networks

- Make sure it is learning something: The error on the training set should beat a naive baseline corresponding to the loss being evaluated.
- If the performance on the training set is poor, change the model.
  (Poor performance on the training set indicates under-fitting.)
- Test the error on the validation set. If the validation error does not improve as the algorithm proceeds, it means the learning is not generalizing, and it is fitting to noise.
  (Poor performance on the validation set indicates overfitting.)
  In this case you should simplify the model.
- Then carry out hyperparameter tuning.
- If the performance isn't adequate, try to collect more data!
- Data augmentation can be a way to get more data, e.g., adding noise, scaling, translating or rotating images. (What can go wrong?)

Hyperparameter Tuning

The hyperparameters that can be tuned include:

Hyperparameter Tuning

The hyperparameters that can be tuned include:

- the algorithm (a decision tree or gradient-boosted trees may be more appropriate than a neural network)

Hyperparameter Tuning

The hyperparameters that can be tuned include:

- the algorithm (a decision tree or gradient-boosted trees may be more appropriate than a neural network)
- number of layers

Hyperparameter Tuning

The hyperparameters that can be tuned include:

- the algorithm (a decision tree or gradient-boosted trees may be more appropriate than a neural network)
- number of layers
- width of each layer

Hyperparameter Tuning

The hyperparameters that can be tuned include:

- the algorithm (a decision tree or gradient-boosted trees may be more appropriate than a neural network)
- number of layers
- width of each layer
- number of epochs, to allow for early stopping

Hyperparameter Tuning

The hyperparameters that can be tuned include:

- the algorithm (a decision tree or gradient-boosted trees may be more appropriate than a neural network)
- number of layers
- width of each layer
- number of epochs, to allow for early stopping
- learning rate

Hyperparameter Tuning

The hyperparameters that can be tuned include:

- the algorithm (a decision tree or gradient-boosted trees may be more appropriate than a neural network)
- number of layers
- width of each layer
- number of epochs, to allow for early stopping
- learning rate
- batch size

Hyperparameter Tuning

The hyperparameters that can be tuned include:

- the algorithm (a decision tree or gradient-boosted trees may be more appropriate than a neural network)
- number of layers
- width of each layer
- number of epochs, to allow for early stopping
- learning rate
- batch size
- $L1$ and $L2$ regularization parameters.

Dropout

- dropout involves randomly dropping some units during training.

Dropout

- dropout involves randomly dropping some units during training.
- Ignoring a unit is equivalent to temporarily setting its output to zero.

Dropout

- dropout involves randomly dropping some units during training.
- Ignoring a unit is equivalent to temporarily setting its output to zero.
- Dropout is controlled by a parameter *rate*, which specifies the proportion of values that are zeroed.

Dropout

- dropout involves randomly dropping some units during training.
- Ignoring a unit is equivalent to temporarily setting its output to zero.
- Dropout is controlled by a parameter *rate*, which specifies the proportion of values that are zeroed.
- During evaluation, dropout is not used.

Dropout

- dropout involves randomly dropping some units during training.
- Ignoring a unit is equivalent to temporarily setting its output to zero.
- Dropout is controlled by a parameter *rate*, which specifies the proportion of values that are zeroed.
- During evaluation, dropout is not used.
- To account for missing units, the prediction needs to be scaled by

Dropout

- dropout involves randomly dropping some units during training.
- Ignoring a unit is equivalent to temporarily setting its output to zero.
- Dropout is controlled by a parameter *rate*, which specifies the proportion of values that are zeroed.
- During evaluation, dropout is not used.
- To account for missing units, the prediction needs to be scaled by $1/(1 - rate)$.