Computational Cognitive Neuroscience

Week 1 – coursework intro

University of Helsinki, 21st January 2021

Coursework intro

- one programming (python) assignment per week

Coursework intro

- one programming (python) assignment per week
- given out Thursday before midnight; one week to complete

Coursework intro

- one programming (python) assignment per week
- given out Thursday before midnight; one week to complete
- course grade fully based on assignments

Coursework intro

- one programming (python) assignment per week
- given out Thursday before midnight; one week to complete
- course grade fully based on assignments
- individual - no group work

Coursework intro

- one programming (python) assignment per week
- given out Thursday before midnight; one week to complete
- course grade fully based on assignments
- individual - no group work
- more info and place to ask question is on github (see course moodle for link)

Coursework intro

- one programming (python) assignment per week
- given out Thursday before midnight; one week to complete
- course grade fully based on assignments
- individual - no group work
- more info and place to ask question is on github (see course moodle for link)
- self-study course so students expected to read course material to do questions

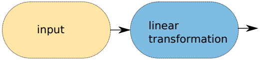
Coursework intro

- one programming (python) assignment per week
- given out Thursday before midnight; one week to complete
- course grade fully based on assignments
- individual - no group work
- more info and place to ask question is on github (see course moodle for link)
- self-study course so students expected to read course material to do questions
- Thursdays we cover some material useful for assignments - today neural networks

Neural networks intro

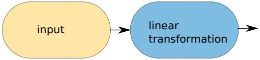
- "logistic regression in a trench coat"
- sequence of layers/modules/functions transform data and make a prediction
- layers mostly linear transformations followed by non-linearities
- parameters learned by minimizing a loss function wrt. data
- sequence of layers like composite functions $f(g(x, w))$; can use chain rule (backpropagation)

Linear layer



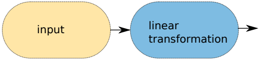
- Data point $\mathbf{x}_i \in \mathbb{R}^d$

Linear layer



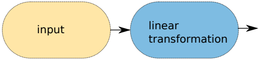
- Data point $\mathbf{x}_i \in \mathbb{R}^d$
- $z_i = \mathbf{w}^T \mathbf{x}_i + b$

Linear layer



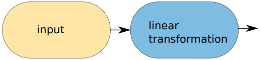
- Data point $\mathbf{x}_i \in \mathbb{R}^d$
- $z_i = \mathbf{w}^T \mathbf{x}_i + b$
    - $\mathbf{w}$ are weights, b bias, and z_i score

Linear layer



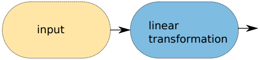
- Data point $\mathbf{x}_i \in \mathbb{R}^d$
- $z_i = \mathbf{w}^T \mathbf{x}_i + b$
  - $\mathbf{w}$ are weights, b bias, and z_i score
- usually output a vector rather than scalar, thus use weight matrix $\mathbf{W} = [\mathbf{w}_1, \ldots, \mathbf{w}_m]$

Linear layer



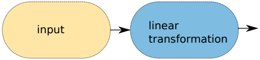
- Data point $\mathbf{x}_i \in \mathbb{R}^d$
- $z_i = \mathbf{w}^T \mathbf{x}_i + b$
  - $\mathbf{w}$ are weights, b bias, and z_i score
- usually output a vector rather than scalar, thus use weight matrix $\mathbf{W} = [\mathbf{w}_1, \ldots, \mathbf{w}_m]$
- m is the *width* or the number of *units* in the layer

Linear layer



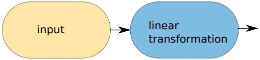
- Data point $\mathbf{x}_i \in \mathbb{R}^d$
- $z_i = \mathbf{w}^T \mathbf{x}_i + b$
  - $\mathbf{w}$ are weights, b bias, and z_i score
- usually output a vector rather than scalar, thus use weight matrix $\mathbf{W} = [\mathbf{w}_1, \ldots, \mathbf{w}_m]$
- m is the *width* or the number of *units* in the layer
- $\mathbf{z}_i = \mathbf{x}_i^T \mathbf{W} + \mathbf{b}$

Linear layer



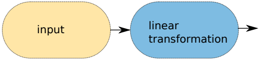
- Data point $\mathbf{x}_i \in \mathbb{R}^d$
- $z_i = \mathbf{w}^T \mathbf{x}_i + b$
  - $\mathbf{w}$ are weights, b bias, and z_i score
- usually output a vector rather than scalar, thus use weight matrix $\mathbf{W} = [\mathbf{w}_1, \ldots, \mathbf{w}_m]$
- m is the *width* or the number of *units* in the layer
- $\mathbf{z}_i = \mathbf{x}_i^T \mathbf{W} + \mathbf{b}$
- can do for many observations simultaneously $\mathbf{Z} = \mathbf{X}\mathbf{W} + \mathbf{B}$

Linear layer



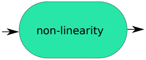
- Data point $\mathbf{x}_i \in \mathbb{R}^d$
- $z_i = \mathbf{w}^T \mathbf{x}_i + b$
  - $\mathbf{w}$ are weights, b bias, and z_i score
- usually output a vector rather than scalar, thus use weight matrix $\mathbf{W} = [\mathbf{w}_1, \ldots, \mathbf{w}_m]$
- m is the *width* or the number of *units* in the layer
- $\mathbf{z}_i = \mathbf{x}_i^T \mathbf{W} + \mathbf{b}$
- can do for many observations simultaneously $\mathbf{Z} = \mathbf{X}\mathbf{W} + \mathbf{B}$
- here $\mathbf{X} = [\mathbf{x}_1, \ldots, \mathbf{x}_n]^T$ and $\mathbf{B} = [\mathbf{b}, \ldots, \mathbf{b}]^T$

Linear layer



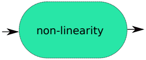
- Data point $\mathbf{x}_i \in \mathbb{R}^d$
- $z_i = \mathbf{w}^T \mathbf{x}_i + b$
  - $\mathbf{w}$ are weights, b bias, and z_i score
- usually output a vector rather than scalar, thus use weight matrix $\mathbf{W} = [\mathbf{w}_1, \ldots, \mathbf{w}_m]$
- m is the *width* or the number of *units* in the layer
- $\mathbf{z}_i = \mathbf{x}_i^T \mathbf{W} + \mathbf{b}$
- can do for many observations simultaneously $\mathbf{Z} = \mathbf{X}\mathbf{W} + \mathbf{B}$
- here $\mathbf{X} = [\mathbf{x}_1, \ldots, \mathbf{x}_n]^T$ and $\mathbf{B} = [\mathbf{b}, \ldots, \mathbf{b}]^T$
- or just $\mathbf{Z} = f_{\text{linear}}(\mathbf{X}, \mathbf{W}, \mathbf{b})$

non-linear / activation layer



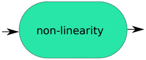
- some input from previous layer $\mathbf{Z}_{\text{in}}$, often after linear layer

non-linear / activation layer

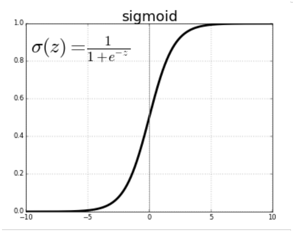


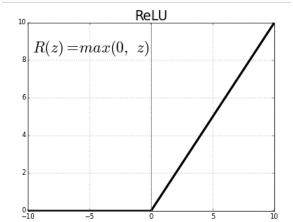
- some input from previous layer $\mathbf{Z}_{\text{in}}$, often after linear layer
- outputs a nonlinear transformation $\mathbf{Z}_{\text{out}} = f_{\text{nonlin}}(\mathbf{Z}_{\text{in}})$

non-linear / activation layer

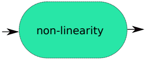


- some input from previous layer $\mathbf{Z}_{\text{in}}$, often after linear layer
- outputs a nonlinear transformation $\mathbf{Z}_{\text{out}} = f_{\text{nonlin}}(\mathbf{Z}_{\text{in}})$
- huge number of options e.g.:

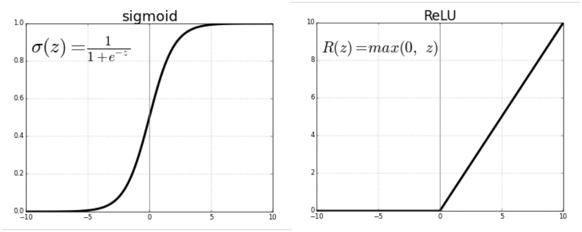




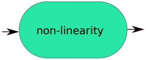
non-linear / activation layer



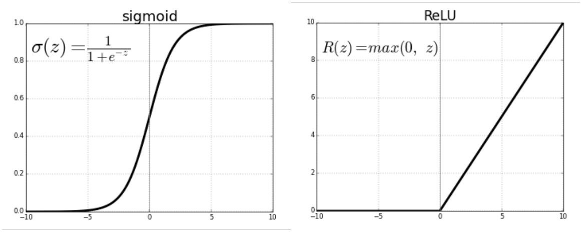
- some input from previous layer $\mathbf{Z}_{\text{in}}$, often after linear layer
- outputs a nonlinear transformation $\mathbf{Z}_{\text{out}} = f_{\text{nonlin}}(\mathbf{Z}_{\text{in}})$
- huge number of options e.g.:
  - ReLU: $z_{j,k}^{(out)} = \max\left(z_{j,k}^{(in)}, 0\right)$



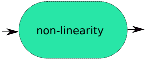
non-linear / activation layer



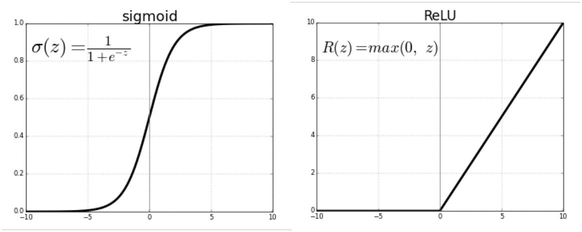
- some input from previous layer $\mathbf{Z}_{\text{in}}$, often after linear layer
- outputs a nonlinear transformation $\mathbf{Z}_{\text{out}} = f_{\text{nonlin}}(\mathbf{Z}_{\text{in}})$
- huge number of options e.g.:
  - ReLU: $z_{j,k}^{(out)} = \max\left(z_{j,k}^{(in)}, 0\right)$
  - sigmoid: $z_{j,k}^{(out)} = \frac{1}{1+e^{-z_{j,k}^{(in)}}}$



non-linear / activation layer

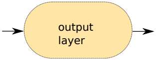


- some input from previous layer $\mathbf{Z}_{\text{in}}$, often after linear layer
- outputs a nonlinear transformation $\mathbf{Z}_{\text{out}} = f_{\text{nonlin}}(\mathbf{Z}_{\text{in}})$
- huge number of options e.g.:
  - ReLU: $z_{j,k}^{(out)} = \max\left(z_{j,k}^{(in)}, 0\right)$
  - sigmoid: $z_{j,k}^{(out)} = \frac{1}{1+e^{-z_{j,k}^{(in)}}}$



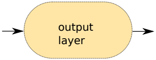
- usually no parameters

output layer



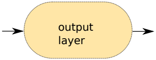
- the type of output layer depends on what we are predicting

output layer



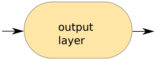
- the type of output layer depends on what we are predicting
- if target is continuous valued (regression problem) then output layer is usually just a linear layer

output layer



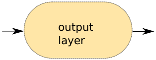
- the type of output layer depends on what we are predicting
- if target is continuous valued (regression problem) then output layer is usually just a linear layer
- if target is discrete or a classification problem, we need to somehow output class probabilities (next slide)

output layer



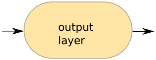
- the type of output layer depends on what we are predicting
- if target is continuous valued (regression problem) then output layer is usually just a linear layer
- if target is discrete or a classification problem, we need to somehow output class probabilities (next slide)
- in either case, the number of units in this layer must match the size of the output, for example:

output layer



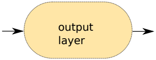
- the type of output layer depends on what we are predicting
- if target is continuous valued (regression problem) then output layer is usually just a linear layer
- if target is discrete or a classification problem, we need to somehow output class probabilities (next slide)
- in either case, the number of units in this layer must match the size of the output, for example:
  - assume this layer receives a matrix $\mathbf{Z}_{in}$ of dimension $n \times m$

output layer



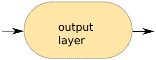
- the type of output layer depends on what we are predicting
- if target is continuous valued (regression problem) then output layer is usually just a linear layer
- if target is discrete or a classification problem, we need to somehow output class probabilities (next slide)
- in either case, the number of units in this layer must match the size of the output, for example:
  - assume this layer receives a matrix $\mathbf{Z}_{in}$ of dimension $n \times m$
  - assume that the target variable for each of the n observations is k-dimensional

output layer



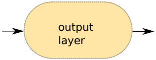
- the type of output layer depends on what we are predicting
- if target is continuous valued (regression problem) then output layer is usually just a linear layer
- if target is discrete or a classification problem, we need to somehow output class probabilities (next slide)
- in either case, the number of units in this layer must match the size of the output, for example:
    - assume this layer receives a matrix $\mathbf{Z}_{in}$ of dimension $n \times m$
    - assume that the target variable for each of the n observations is k-dimensional
    - then output layer is a linear layer with weight matrix of dimension $m \times k$

output layer



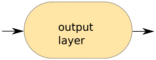
- the type of output layer depends on what we are predicting
- if target is continuous valued (regression problem) then output layer is usually just a linear layer
- if target is discrete or a classification problem, we need to somehow output class probabilities (next slide)
- in either case, the number of units in this layer must match the size of the output, for example:
  - assume this layer receives a matrix $\mathbf{Z}_{in}$ of dimension $n \times m$
  - assume that the target variable for each of the n observations is k-dimensional
  - then output layer is a linear layer with weight matrix of dimension $m \times k$
- for regression problems, no activation function on top of output layer

output layer for classification



- like for regression problem we have linear layer that matches output size,

output layer for classification

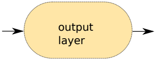


- like for regression problem we have linear layer that matches output size,
- eg. for a single observation i,

$$\mathbf{z}_i^{(out)} = \mathbf{w}^T \mathbf{z}_i^{(in)} + \mathbf{b}$$

classifying between C classes of data, $\mathbf{z}_i^{(out)}$ is C-dimensional

output layer for classification



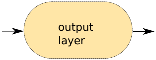
- like for regression problem we have linear layer that matches output size,
- eg. for a single observation i,

$$\mathbf{z}_i^{(out)} = \mathbf{w}^T \mathbf{z}_i^{(in)} + \mathbf{b}$$

  classifying between C classes of data, $\mathbf{z}_i^{(out)}$ is C-dimensional
- predict class probabilities with softmax function

$$\hat{\mathbf{y}}_i = \text{softmax}\left(\mathbf{z}_i^{(out)}\right), \text{ where } \hat{y}_{i,j} = \frac{e^{z_{i,j}}}{\sum_{l=1}^{C} e^{z_{i,l}}} = \hat{p}(c_i = j | \mathbf{z}_i)$$

output layer for classification



- like for regression problem we have linear layer that matches output size,
- eg. for a single observation i,

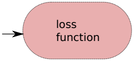
$$\mathbf{z}_i^{(out)} = \mathbf{w}^T \mathbf{z}_i^{(in)} + \mathbf{b}$$

  classifying between C classes of data, $\mathbf{z}_i^{(out)}$ is C-dimensional
- predict class probabilities with softmax function

$$\hat{\mathbf{y}}_i = \text{softmax}\left(\mathbf{z}_i^{(out)}\right), \text{ where } \hat{y}_{i,j} = \frac{e^{z_{i,j}}}{\sum_{l=1}^{C} e^{z_{i,l}}} = \hat{p}(c_i = j|\mathbf{z}_i)$$

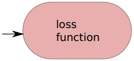
- hence $\hat{\mathbf{y}}_i = \left(\hat{p}(c_i = 1|\mathbf{z}_i), \ldots, \hat{p}(c_i = C|\mathbf{z}_i)\right)$

loss functions



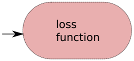
- measure how well current model fits data

loss functions



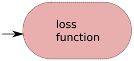
- measure how well current model fits data
- huge number of options, for different tasks and properties

loss functions



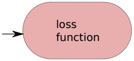
- measure how well current model fits data
- huge number of options, for different tasks and properties
- the most common ones are probably:

loss functions



- measure how well current model fits data
- huge number of options, for different tasks and properties
- the most common ones are probably:
  - mean squared error (regression tasks): $\frac{1}{n}\sum_{i=1}^{n}\|\mathbf{y}_i - \hat{\mathbf{y}}_i\|_2^2$

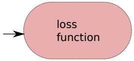
loss functions



- measure how well current model fits data
- huge number of options, for different tasks and properties
- the most common ones are probably:
  - mean squared error (regression tasks): $\frac{1}{n}\sum_{i=1}^{n}\|\mathbf{y}_i - \hat{\mathbf{y}}_i\|_2^2$
  - cross-entropy/negative log-likelihood (classification tasks):

$$\mathrm{XE}(\mathbf{y}_i, \hat{\mathbf{y}}_i) = -\sum_{j=1}^{C} y_{i,j} \log \hat{y}_{i,j}$$

loss functions



- measure how well current model fits data
- huge number of options, for different tasks and properties
- the most common ones are probably:
  - mean squared error (regression tasks): $\frac{1}{n}\sum_{i=1}^{n}\|\mathbf{y}_i - \hat{\mathbf{y}}_i\|_2^2$
  - cross-entropy/negative log-likelihood (classification tasks):

$$\mathrm{XE}(\mathbf{y}_i, \hat{\mathbf{y}}_i) = -\sum_{j=1}^{C} y_{i,j} \log \hat{y}_{i,j}$$

  - here $y_{i,j} = 1$ when j is the true class, 0 for others. $\hat{y}_{i,j}$ is the predicted probability for each class

Combining layers – softmax cross-entropy

- the output softmax layer, and the cross entropy layer are often combined

$$-\sum_{i=1}^{n} \log \underbrace{\left(\frac{\exp(z_{i,j}[y_i])}{\sum_{c=1}^{10} \exp(z_{i,j}[c_j])} \right)}_{\text{softmax output}}$$

Combining layers – softmax cross-entropy

- the output softmax layer, and the cross entropy layer are often combined
$$-\sum_{i=1}^{n} \log \underbrace{\left(\frac{\exp(z_{i,j}[y_i])}{\sum_{c=1}^{10} \exp(z_{i,j}[c_j])} \right)}_{\text{softmax output}}$$
- try to think why this form makes sense, and how it can be derived

Combining layers – softmax cross-entropy

- the output softmax layer, and the cross entropy layer are often combined

$$-\sum_{i=1}^{n} \log \underbrace{\left(\frac{\exp(z_{i,j}[y_i])}{\sum_{c=1}^{10} \exp(z_{i,j}[c_j])} \right)}_{\text{softmax output}}$$

- try to think why this form makes sense, and how it can be derived
- combining layers is easy as layers are essentially functions

Combining layers – softmax cross-entropy

- the output softmax layer, and the cross entropy layer are often combined

$$-\sum_{i=1}^{n} \log \underbrace{\left(\frac{\exp(z_{i,j}[y_i])}{\sum_{c=1}^{10} \exp(z_{i,j}[c_j])} \right)}_{\text{softmax output}}$$

- try to think why this form makes sense, and how it can be derived
- combining layers is easy as layers are essentially functions
- can e.g. combine linear layer + nonlinearity and call it a single layer

Combining layers – softmax cross-entropy

- the output softmax layer, and the cross entropy layer are often combined
$$-\sum_{i=1}^{n} \log \underbrace{\left(\frac{\exp(z_{i,j}[y_i])}{\sum_{c=1}^{10} \exp(z_{i,j}[c_j])} \right)}_{\text{softmax output}}$$
- try to think why this form makes sense, and how it can be derived
- combining layers is easy as layers are essentially functions
- can e.g. combine linear layer + nonlinearity and call it a single layer
- in softmax cross-entropy the computational properties are improved for combined layer, particularly in the backpropagation

Learning with backpropagation

- learning: adjust model parameters to minimize loss given data

Learning with backpropagation

- learning: adjust model parameters to minimize loss given data
- composite function nature of layers means we can use chain rule (in vector/matrix form) to compute derivatives

Learning with backpropagation

- learning: adjust model parameters to minimize loss given data
- composite function nature of layers means we can use chain rule (in vector/matrix form) to compute derivatives
- example. Assume layers
  $\mathbf{Z}_1 = f_1(\mathbf{X}, \mathbf{W}_1), \mathbf{Z}_2 = f_2(\mathbf{Z}_1, \mathbf{W}_2), l = f_3(\mathbf{Z}_2)$

Learning with backpropagation

- learning: adjust model parameters to minimize loss given data
- composite function nature of layers means we can use chain rule (in vector/matrix form) to compute derivatives
- example. Assume layers
  $\mathbf{Z}_1 = f_1(\mathbf{X}, \mathbf{W}_1), \mathbf{Z}_2 = f_2(\mathbf{Z}_1, \mathbf{W}_2), l = f_3(\mathbf{Z}_2)$
- then we can compute:

Learning with backpropagation

- learning: adjust model parameters to minimize loss given data
- composite function nature of layers means we can use chain rule (in vector/matrix form) to compute derivatives
- example. Assume layers $\mathbf{Z}_1 = f_1(\mathbf{X}, \mathbf{W}_1), \mathbf{Z}_2 = f_2(\mathbf{Z}_1, \mathbf{W}_2), l = f_3(\mathbf{Z}_2)$
- then we can compute:
  1. $\frac{\partial l}{\partial \mathbf{Z}_2} = \frac{\partial f_3(\mathbf{Z}_2)}{\partial \mathbf{Z}_2}$

Learning with backpropagation

- learning: adjust model parameters to minimize loss given data
- composite function nature of layers means we can use chain rule (in vector/matrix form) to compute derivatives
- example. Assume layers
  $\mathbf{Z}_1 = f_1(\mathbf{X}, \mathbf{W}_1), \mathbf{Z}_2 = f_2(\mathbf{Z}_1, \mathbf{W}_2), l = f_3(\mathbf{Z}_2)$
- then we can compute:
  1. $\frac{\partial l}{\partial \mathbf{Z}_2} = \frac{\partial f_3(\mathbf{Z}_2)}{\partial \mathbf{Z}_2}$
  2. $\frac{\partial \mathbf{Z}_2}{\partial \mathbf{Z}_1} = \frac{\partial f_2(\mathbf{Z}_1, \mathbf{W}_2)}{\partial \mathbf{Z}_1}, \rightarrow_{\text{chain rule}} \frac{\partial l}{\partial \mathbf{Z}_1} = \frac{\partial l}{\partial \mathbf{Z}_2} \frac{\partial \mathbf{Z}_2}{\partial \mathbf{Z}_1}$

Learning with backpropagation

- learning: adjust model parameters to minimize loss given data
- composite function nature of layers means we can use chain rule (in vector/matrix form) to compute derivatives
- example. Assume layers
  $\mathbf{Z}_1 = f_1(\mathbf{X}, \mathbf{W}_1), \mathbf{Z}_2 = f_2(\mathbf{Z}_1, \mathbf{W}_2), l = f_3(\mathbf{Z}_2)$
- then we can compute:
  1. $\frac{\partial l}{\partial \mathbf{Z}_2} = \frac{\partial f_3(\mathbf{Z}_2)}{\partial \mathbf{Z}_2}$
  2. $\frac{\partial \mathbf{Z}_2}{\partial \mathbf{Z}_1} = \frac{\partial f_2(\mathbf{Z}_1, \mathbf{W}_2)}{\partial \mathbf{Z}_1}, \rightarrow_{\text{chain rule}} \frac{\partial l}{\partial \mathbf{Z}_1} = \frac{\partial l}{\partial \mathbf{Z}_2} \frac{\partial \mathbf{Z}_2}{\partial \mathbf{Z}_1}$
  3. $\frac{\partial \mathbf{Z}_2}{\partial \mathbf{W}_2} = \frac{\partial f_2(\mathbf{Z}_1, \mathbf{W}_2)}{\partial \mathbf{W}_2} \rightarrow_{\text{chain rule}} \frac{\partial l}{\partial \mathbf{W}_2} = \frac{\partial l}{\partial \mathbf{Z}_2} \frac{\partial \mathbf{Z}_2}{\partial \mathbf{W}_2}$

Learning with backpropagation

- learning: adjust model parameters to minimize loss given data
- composite function nature of layers means we can use chain rule (in vector/matrix form) to compute derivatives
- example. Assume layers $\mathbf{Z}_1 = f_1(\mathbf{X}, \mathbf{W}_1), \mathbf{Z}_2 = f_2(\mathbf{Z}_1, \mathbf{W}_2), l = f_3(\mathbf{Z}_2)$
- then we can compute:
  1. $\frac{\partial l}{\partial \mathbf{Z}_2} = \frac{\partial f_3(\mathbf{Z}_2)}{\partial \mathbf{Z}_2}$
  2. $\frac{\partial \mathbf{Z}_2}{\partial \mathbf{Z}_1} = \frac{\partial f_2(\mathbf{Z}_1, \mathbf{W}_2)}{\partial \mathbf{Z}_1}, \rightarrow_{\text{chain rule}} \frac{\partial l}{\partial \mathbf{Z}_1} = \frac{\partial l}{\partial \mathbf{Z}_2} \frac{\partial \mathbf{Z}_2}{\partial \mathbf{Z}_1}$
  3. $\frac{\partial \mathbf{Z}_2}{\partial \mathbf{W}_2} = \frac{\partial f_2(\mathbf{Z}_1, \mathbf{W}_2)}{\partial \mathbf{W}_2} \rightarrow_{\text{chain rule}} \frac{\partial l}{\partial \mathbf{W}_2} = \frac{\partial l}{\partial \mathbf{Z}_2} \frac{\partial \mathbf{Z}_2}{\partial \mathbf{W}_2}$
  4. similarly for first layer

Learning with backpropagation

- learning: adjust model parameters to minimize loss given data
- composite function nature of layers means we can use chain rule (in vector/matrix form) to compute derivatives
- example. Assume layers
  $\mathbf{Z}_1 = f_1(\mathbf{X}, \mathbf{W}_1), \mathbf{Z}_2 = f_2(\mathbf{Z}_1, \mathbf{W}_2), l = f_3(\mathbf{Z}_2)$
- then we can compute:
  1. $\frac{\partial l}{\partial \mathbf{Z}_2} = \frac{\partial f_3(\mathbf{Z}_2)}{\partial \mathbf{Z}_2}$
  2. $\frac{\partial \mathbf{Z}_2}{\partial \mathbf{Z}_1} = \frac{\partial f_2(\mathbf{Z}_1, \mathbf{W}_2)}{\partial \mathbf{Z}_1}, \rightarrow_{\text{chain rule}} \frac{\partial l}{\partial \mathbf{Z}_1} = \frac{\partial l}{\partial \mathbf{Z}_2} \frac{\partial \mathbf{Z}_2}{\partial \mathbf{Z}_1}$
  3. $\frac{\partial \mathbf{Z}_2}{\partial \mathbf{W}_2} = \frac{\partial f_2(\mathbf{Z}_1, \mathbf{W}_2)}{\partial \mathbf{W}_2} \rightarrow_{\text{chain rule}} \frac{\partial l}{\partial \mathbf{W}_2} = \frac{\partial l}{\partial \mathbf{Z}_2} \frac{\partial \mathbf{Z}_2}{\partial \mathbf{W}_2}$
  4. similarly for first layer
- thus we find out how changing parameters in specific direction would influence loss

Parameter updates (optimization)

- we want to change each parameter into direction that minimizes loss (optimization)

Parameter updates (optimization)

- we want to change each parameter into direction that minimizes loss (optimization)
- the standard algorithm is called gradient descent

$$\mathbf{W}_{t+1} \leftarrow \mathbf{W}_t - \eta \frac{\partial l}{\partial \mathbf{W}_t}(\mathbf{X}, \mathbf{W}_t)$$

Parameter updates (optimization)

- we want to change each parameter into direction that minimizes loss (optimization)
- the standard algorithm is called gradient descent

$$\mathbf{W}_{t+1} \leftarrow \mathbf{W}_t - \eta \frac{\partial l}{\partial \mathbf{W}_t}(\mathbf{X}, \mathbf{W}_t)$$

- η is learning rate and controls the step size (how big updates)

Parameter updates (optimization)

- we want to change each parameter into direction that minimizes loss (optimization)
- the standard algorithm is called gradient descent

$$\mathbf{W}_{t+1} \leftarrow \mathbf{W}_t - \eta \frac{\partial l}{\partial \mathbf{W}_t}(\mathbf{X}, \mathbf{W}_t)$$

- η is learning rate and controls the step size (how big updates)
- too big learning rate can be miss minima, too small gets stuck in local minima

Parameter updates (optimization)

- we want to change each parameter into direction that minimizes loss (optimization)
- the standard algorithm is called gradient descent

$$\mathbf{W}_{t+1} \leftarrow \mathbf{W}_t - \eta \frac{\partial l}{\partial \mathbf{W}_t}(\mathbf{X}, \mathbf{W}_t)$$

- η is learning rate and controls the step size (how big updates)
- too big learning rate can be miss minima, too small gets stuck in local minima
- often $\mathbf{X}$ is split into some B subsets, prediction and backprop done for each subset in turn

Parameter updates (optimization)

- we want to change each parameter into direction that minimizes loss (optimization)
- the standard algorithm is called gradient descent

$$\mathbf{W}_{t+1} \leftarrow \mathbf{W}_t - \eta \frac{\partial l}{\partial \mathbf{W}_t}(\mathbf{X}, \mathbf{W}_t)$$

- η is learning rate and controls the step size (how big updates)
- too big learning rate can be miss minima, too small gets stuck in local minima
- often $\mathbf{X}$ is split into some B subsets, prediction and backprop done for each subset in turn
- leads to stochastic gradient descent

Parameter updates (optimization)

- we want to change each parameter into direction that minimizes loss (optimization)
- the standard algorithm is called gradient descent

$$\mathbf{W}_{t+1} \leftarrow \mathbf{W}_t - \eta \frac{\partial l}{\partial \mathbf{W}_t}(\mathbf{X}, \mathbf{W}_t)$$

- η is learning rate and controls the step size (how big updates)
- too big learning rate can be miss minima, too small gets stuck in local minima
- often $\mathbf{X}$ is split into some B subsets, prediction and backprop done for each subset in turn
- leads to stochastic gradient descent
- once we have done update for all subsets, we have completed a single training *epoch*

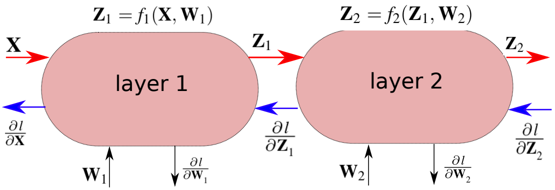
Parameter updates (optimization)

- we want to change each parameter into direction that minimizes loss (optimization)
- the standard algorithm is called gradient descent

$$\mathbf{W}_{t+1} \leftarrow \mathbf{W}_t - \eta \frac{\partial l}{\partial \mathbf{W}_t}(\mathbf{X}, \mathbf{W}_t)$$

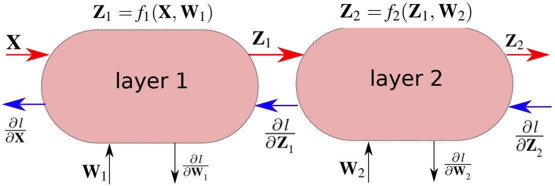
- η is learning rate and controls the step size (how big updates)
- too big learning rate can be miss minima, too small gets stuck in local minima
- often $\mathbf{X}$ is split into some B subsets, prediction and backprop done for each subset in turn
- leads to stochastic gradient descent
- once we have done update for all subsets, we have completed a single training *epoch*
- often multiple epochs of training needed

summary



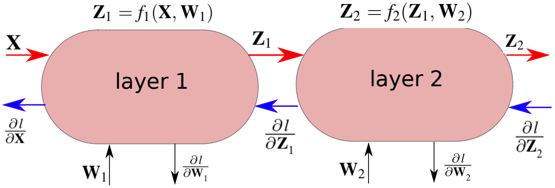
- forward pass (red) creates predictions and loss

summary



- forward pass (red) creates predictions and loss
- backward pass (blue) calculates gradients with respect to loss

summary



- forward pass (red) creates predictions and loss
- backward pass (blue) calculates gradients with respect to loss
- see github page for useful reference and reading