# Introduction to neural networks

## Machine learning & neural networks

Anne Helby Petersen

# Logistic regression with single predictors

Top-scores for single predictors using logistic regression:

```
##          variable  accuracy
## 1 CORTICOSTEROID 0.6952055
## 2             AST 0.6952055
## 3             ALP 0.6883562
## 4          ECOG_1 0.6712329
## 5  AGEGRP_75plus 0.6712329
```

# Logistic regression with single predictors

Top-scores for single predictors using logistic regression:

```
##          variable  accuracy
## 1 CORTICOSTEROID 0.6952055
## 2             AST 0.6952055
## 3             ALP 0.6883562
## 4         ECOG_1 0.6712329
## 5  AGEGRP_75plus 0.6712329
```

$\Rightarrow$ **We need to use more variables.**

# Logistic regression with single predictors

Top-scores for single predictors using logistic regression:

```
##          variable  accuracy
## 1 CORTICOSTEROID 0.6952055
## 2             AST 0.6952055
## 3             ALP 0.6883562
## 4          ECOG_1 0.6712329
## 5  AGEGRP_75plus 0.6712329
```

$\Rightarrow$ **We need to use more variables.**

But that opens up a lot of choices. And we don't want to *think*.

# Logistic regression with single predictors

Top-scores for single predictors using logistic regression:

```
##          variable  accuracy
## 1 CORTICOSTEROID 0.6952055
## 2             AST 0.6952055
## 3             ALP 0.6883562
## 4          ECOG_1 0.6712329
## 5  AGEGRP_75plus 0.6712329
```

$\Rightarrow$ **We need to use more variables.**

But that opens up a lot of choices. And we don't want to *think*.

$\Rightarrow$ **Automate it - use neural networks.**

# More automation, please

Logistic regression is helpful, but cumbersome. We would have to manually perform *feature engineering*:

- choose what variables to use
- choose their functional forms (squared, log-transformation, ...)
- decide if interaction effects should be added

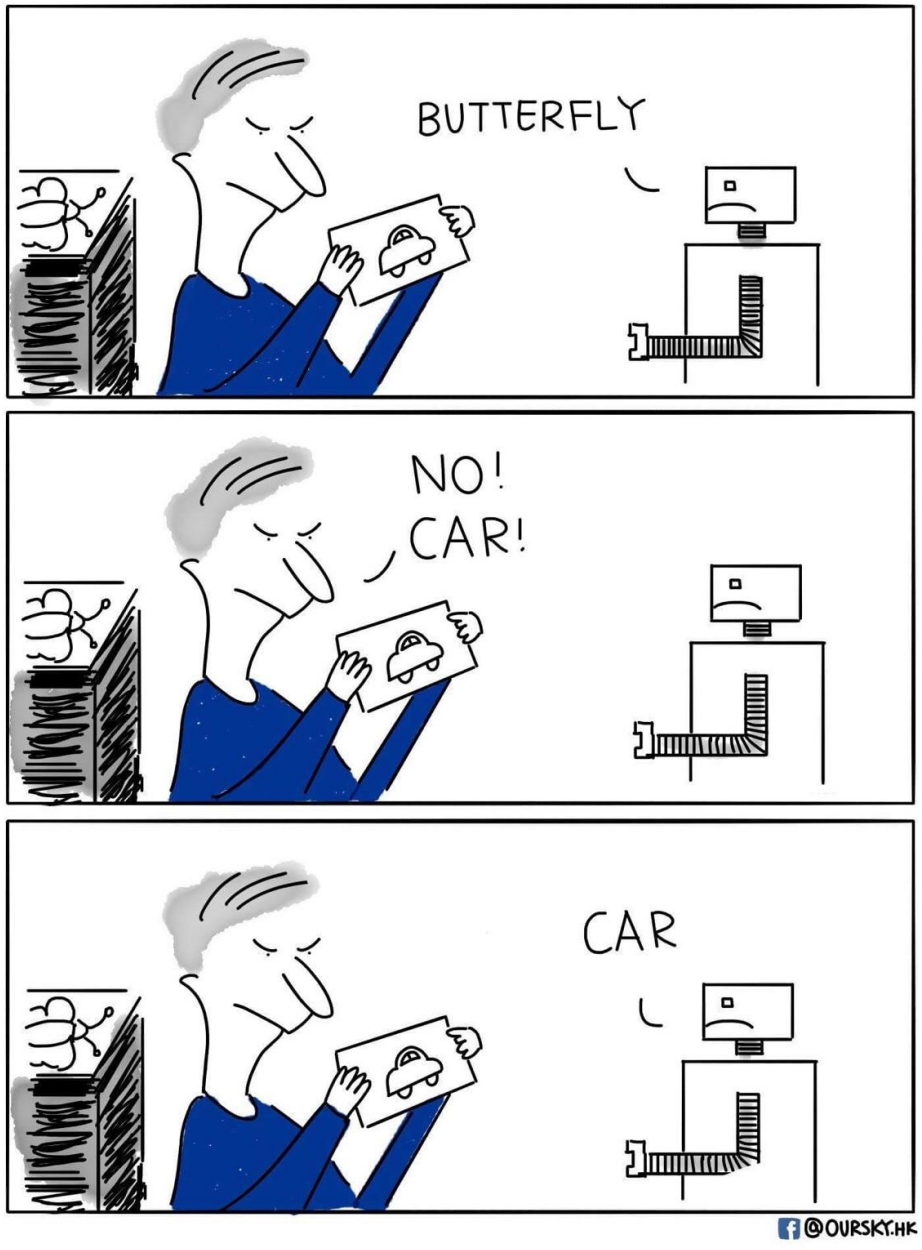
# More automation, please

Logistic regression is helpful, but cumbersome. We would have to manually perform *feature engineering*:

- choose what variables to use
- choose their functional forms (squared, log-transformation, ...)
- decide if interaction effects should be added

**Goal now: More automation, less thinking.**

# Teaching by examples



# Logistic regression revisited

Recall that the logistic regression model on $d$ pre-selected variables $x_1, ..., x_d$ computes label probabilities using the function:

$$f_{\text{logit}}(x_1, x_2, ..., x_d) = \frac{\exp(\hat{\alpha} + \hat{\beta}_1 x_1 + ... + \hat{\beta}_d x_d)}{1 + \exp(\hat{\alpha} + \hat{\beta}_1 x_1 + ... + \hat{\beta}_d x_d)}$$
$$= g(\hat{\alpha} + \hat{\beta}_1 x_1 + ... + \hat{\beta}_d x_d)$$

where $\hat{\alpha}, \hat{\beta}_1, ..., \hat{\beta}_d$ are learned from data and $g(x) = \frac{\exp(x)}{1+\exp(x)}$

# Logistic regression revisited

Recall that the logistic regression model on $d$ pre-selected variables $x_1, ..., x_d$ computes label probabilities using the function:

$$f_{\text{logit}}(x_1, x_2, ..., x_d) = \frac{\exp(\hat{\alpha} + \hat{\beta}_1 x_1 + ... + \hat{\beta}_d x_d)}{1 + \exp(\hat{\alpha} + \hat{\beta}_1 x_1 + ... + \hat{\beta}_d x_d)}$$
$$= g(\hat{\alpha} + \hat{\beta}_1 x_1 + ... + \hat{\beta}_d x_d)$$

where $\hat{\alpha}, \hat{\beta}_1, ..., \hat{\beta}_d$ are learned from data and $g(x) = \frac{\exp(x)}{1+\exp(x)}$

**Main ideas in neural networks:**

1. Do this repeatedly to allow for more flexible modeling of the $x$s
2. Use other (and varying) functions $g$.

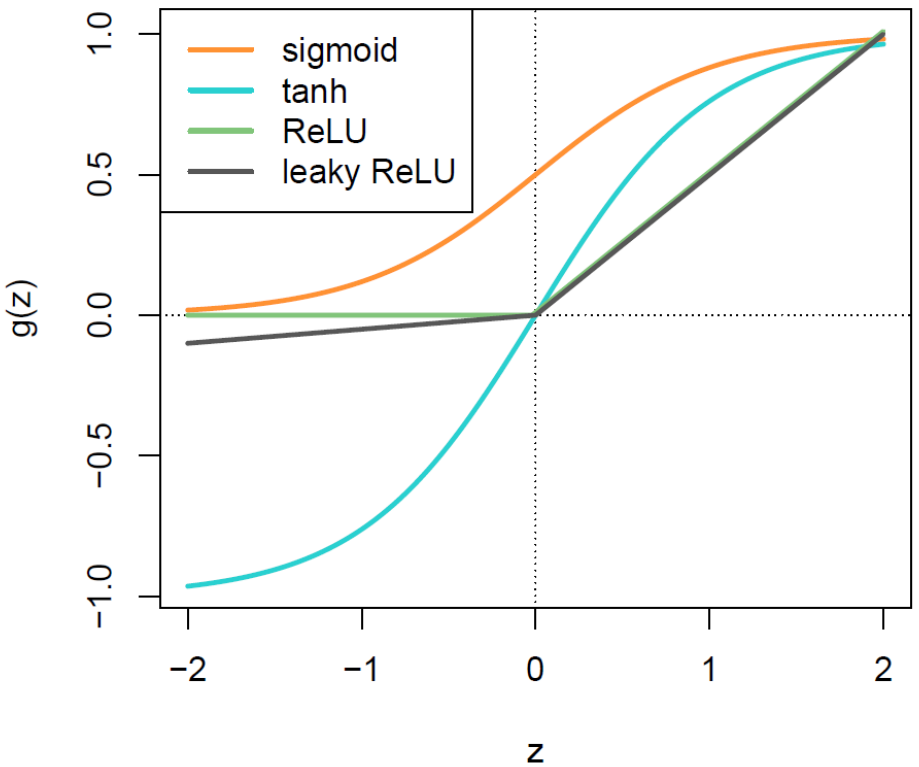
# A simple neural network

Let's look at a simple NN on the "board"...

# Activation functions: $g^{(1)}, g^{(2)}, \ldots$

Rules of thumb:

- 'relu' is often claimed to perform the best on hidden layers.
- 'softmax' (not shown) is go to for output layer when the task is classification. It gives weights that sum to 1 and which can then be interpreted as probabilities.



# Learning for NNs

- **Estimation task:** Find optimal values for the weights $w_{lj}^{(k)}$

# Learning for NNs

- **Estimation task:** Find optimal values for the weights $w_{lj}^{(k)}$
- **Strategy:** Minimize a *loss* function
  - For regression, this could be e.g. the *MSE* (mean squared error) loss

$$\frac{1}{n_{\text{train}}} \sum_{i=1}^{n_{\text{train}}} (Y_i - \hat{Y}_i)^2$$

  - For binary classification, we use the *binary cross entropy* loss

$$-\frac{1}{n_{\text{train}}} \sum_{i=1}^{n_{\text{train}}} Y_i \cdot \log(f(x_i \mid W)) + (1 - Y_i) \cdot (\log(1 - f(x_i \mid W)))$$

    where $f(x_i \mid W) = f(x_{i1}, ..., x_{ip} \mid W)$ and $W$ are all the weights used for $f$.

# Learning for NNs

- **Estimation task:** Find optimal values for the weights $w_{lj}^{(k)}$
- **Strategy:** Minimize a *loss* function
  - For regression, this could be e.g. the *MSE* (mean squared error) loss

$$\frac{1}{n_{\text{train}}} \sum_{i=1}^{n_{\text{train}}} (Y_i - \hat{Y}_i)^2$$

  - For binary classification, we use the *binary cross entropy* loss

$$-\frac{1}{n_{\text{train}}} \sum_{i=1}^{n_{\text{train}}} Y_i \cdot \log(f(x_i \mid W)) + (1 - Y_i) \cdot (\log(1 - f(x_i \mid W)))$$

    where $f(x_i \mid W) = f(x_{i1}, ..., x_{ip} \mid W)$ and $W$ are all the weights used for $f$.

- **Method:** Stochastic gradient descent

# Optimization algorithm

Algorithm outline:

1. Begin with random values for all the weights $w_{lj}^{(k)}$.
2. In each *epoch* (round) do the following:
   - 2.1 Split the training data into batches, each of size $n_{\text{batch}}$ (possibly equal to 1).
   - 2.2 Starting with a random batch $b$, do the following sequentially for each batch:
     - 2.2.1 Compute the loss function for the observations in $b$
     - 2.2.2 Change weights $W$ slightly such that the loss becomes smaller for the observations in $b$
3. Return $f$ with the final weights found.

# Optimization algorithm

Algorithm outline:

1. Begin with random values for all the weights $w_{lj}^{(k)}$.
2. In each *epoch* (round) do the following:
   - 2.1 Split the training data into batches, each of size $n_{\text{batch}}$ (possibly equal to 1).
   - 2.2 Starting with a random batch $b$, do the following sequentially for each batch:
     - 2.2.1 Compute the loss function for the observations in $b$
     - 2.2.2 Change weights $W$ slightly such that the loss becomes smaller for the observations in $b$
3. Return $f$ with the final weights found.

# Stochastic gradient descent

**Q**: How do we "change weights $W$ slightly such that the loss becomes smaller for the observations in $b$"?

**A**: Stochastic gradient descent.

# Stochastic gradient descent

**Q**: How do we “change weights $W$ slightly such that the loss becomes smaller for the observations in $b$”?

**A**: Stochastic gradient descent.

- Take small "steps" in the direction that decreases the loss the most (negative gradient)

# Stochastic gradient descent

**Q**: How do we “change weights $W$ slightly such that the loss becomes smaller for the observations in $b$”?

**A**: Stochastic gradient descent.

- Take small "steps" in the direction that decreases the loss the most (negative gradient)
  - Compute gradients using *backpropagation*

# Stochastic gradient descent

**Q**: How do we "change weights $W$ slightly such that the loss becomes smaller for the observations in $b$"?

**A**: Stochastic gradient descent.

- Take small "steps" in the direction that decreases the loss the most (negative gradient)
  - Compute gradients using *backpropagation*
- Let's see how it works if we only had one weight...

# Stochastic gradient descent

**Q**: How do we "change weights $W$ slightly such that the loss becomes smaller for the observations in $b$"?

**A**: Stochastic gradient descent.

- Take small "steps" in the direction that decreases the loss the most (negative gradient)
  - Compute gradients using *backpropagation*
- Let's see how it works if we only had one weight...
- Note: The algorithm is only guaranteed to find a *local* minimum

# Stochastic gradient descent tuning

Tunable parameters:

- **Epoch number**: How many times do we go through all the observations?
- **Batch size**: How many observations do we look at at a time when updating weights?
- **Step size** (a.k.a. *learning rate*): How big steps do we take when changing the weights?

# Stochastic gradient descent tuning

Tunable parameters:

- **Epoch number**: How many times do we go through all the observations?
- **Batch size**: How many observations do we look at at a time when updating weights?
- **Step size** (a.k.a. *learning rate*): How big steps do we take when changing the weights?
  - We will use the `RMSprop` variety of stochastic gradient descent which makes this choice for us adaptively.

# Prepping the data

We need to do two things before we get started:

1. Scale the data
   - For each variable in both test and training data, subtract the *training data mean* of that variable, divide by the *training data standard deviation* of that variable.
   - Variables in the scaled training data will then have mean 0, standard deviation 1.
2. Convert input data to `matrix`, convert labels to "one hot deck encoding" (two dummies).

# Prepping the data

We need to do two things before we get started:

1. Scale the data
   - For each variable in both test and training data, subtract the *training data mean* of that variable, divide by the *training data standard deviation* of that variable.
   - Variables in the scaled training data will then have mean 0, standard deviation 1.
2. Convert input data to `matrix`, convert labels to "one hot deck encoding" (two dummies).

All this is done for you if you run:

```
source("prepDataForNNs.R")
```

and now use `NN_traindata_x`, `NN_traindata_DEATH2YRS`, `NN_testdata_x`, `NN_testdata_DEATH2YRS`, ...

# Let's make a NN!

We will now fit a NN, Casper, in `R` with the structure:

L1: Input layer of 91 features

L2: Hidden layer with two nodes and `sigmoid` activation function

L3: Output layer with two nodes and `softmax` activation function

we will use

- 20 epochs
- batch size 10

# Let's make a NN!

We will now fit a NN, Casper, in `R` with the structure:

L1: Input layer of 91 features

L2: Hidden layer with two nodes and `sigmoid` activation function

L3: Output layer with two nodes and `softmax` activation function

we will use

- 20 epochs
- batch size 10

Let's do this in `R`...

# Time to try making NNs yourselves

**Go to the course website and find exercise session 3:**

## Exercise session 3

### Machine learning & neural networks

Anne Helby Petersen

May 9, 2019

## Overview

The goal of this exercise session is to:

- Try fitting a neural network
- Comparing performance on training data and test data

## 3.1: The first neural network

Below, we fit a "simple" neural network, Mindy, that uses all of the available features/x variables as predictors and contains a single hidden layer. More specifically, it will have the following structure:

1. An input layer that takes in the 91 variables in the training data.
2. A hidden layer that consists of 91 neurons and that uses a `sigmoid` activation function to pass its output on to the next layer.
3. An output layer that gives each observation two probabilities. The first probability is the estimated probability of that observation having label `0`, the second probability is the estimated probability of that observation having label `1`.