Applied Machine Learning
Lecture 7: Neural networks

Selpi (selpi@chalmers.se)

The slides are further development of Richard Johansson's slides

February 14, 2020

Overview

Introduction

Getting rid of linear inseparability

neural networks, basic ideas

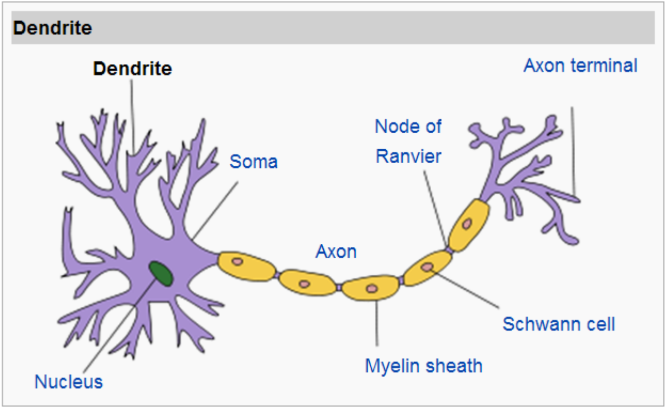
training feedforward neural networks

overview of neural network libraries

tricks of the trade

"Neurons" and Neural network

Structure of a typical neuron



- What is neural network?
- How does a neuron work in neural network?

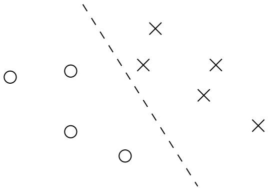
pros and cons of neural networks

- **pros**:
  - can express more complex relationships than e.g. linear models
  - they are excellent for "noisy" problems where it's hard to define features (case in point: images)
  - they have enabled new solutions to some difficult problems (case in point: translation)
- **cons**:
  - training is computationally demanding
  - more "bells and whistles" that require careful tweaking
  - training is mathematically less stable; finding a good model can require some luck
  - complex models $\Rightarrow$ may require a lot of training data to reach their full potential
- neural networks dominate in computer vision, but typically not for problems/datasets where there are "well-defined" features

Overview

recap: linear separability

- some datasets can't be modeled with a linear classifier!



- a dataset is **linearly separable** if there exists a $\boldsymbol{w}$ that gives us perfect classification

example: XOR dataset

```
X = numpy.array([[1, 1],
                 [1, 0],
                 [0, 1],
                 [0, 0]])
Y = ['no', 'yes', 'yes', 'no']

clf = LinearSVC()
clf.fit(X, Y)

# linear inseparability, so we get less than 100% accuracy
print(accuracy_score(Y, clf.predict(X)))
```

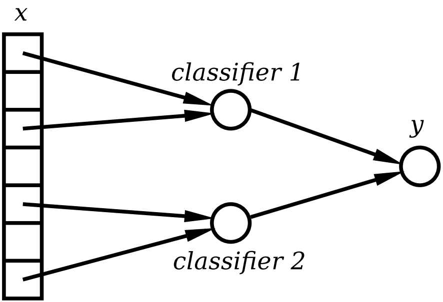
example: XOR dataset with a combination feature

```
# feature1, feature2, feature1&feature2
X = numpy.array([[1, 1, 1],
                 [1, 0, 0],
                 [0, 1, 0],
                 [0, 0, 0]])
Y = ['no', 'yes', 'yes', 'no']

clf = LinearSVC()
clf.fit(X, Y)

# now we have linear separability, so we get 100%
print(accuracy_score(Y, clf.predict(X)))
```

Overview

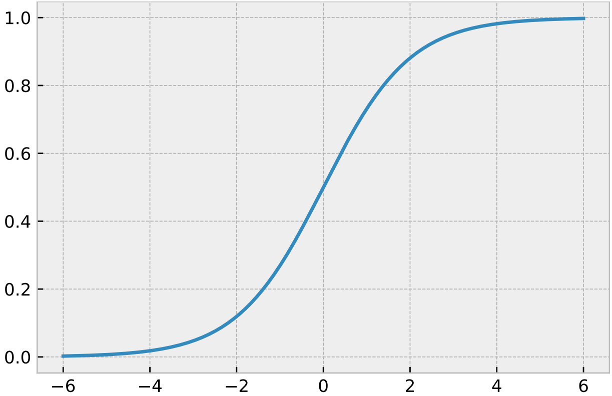
expressing feature combinations as “sub-classifiers”

- instead of hand-crafting additional features, such as $x_3 = x_1^2 + x_2^2$, we could imagine that the combination feature x_3 would be computed by a separate classifier, for instance LR
- we could train a classifier using the output of “sub-classifiers”



recap: the logistic or sigmoid function

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



a multilayered classifier

- a **feedforward neural network** or **multilayer perceptron** consists of connected layers of “classifiers”
  - the intermediate steps are called **hidden units**
  - the final classifier is called the **output unit**
- let's assume two layers for now
- each hidden unit h_i computes its output based on its own weight vector $\boldsymbol{w}_{h_i}$:

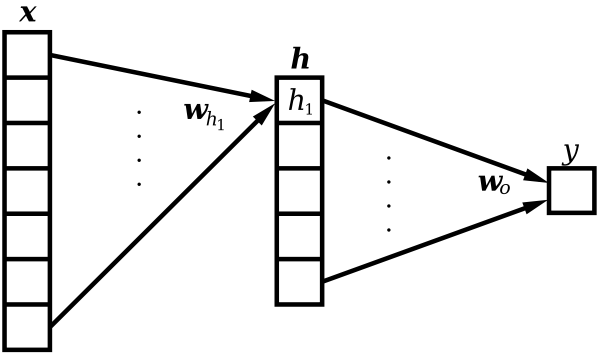
$$h_i = f(\boldsymbol{w}_{h_i} \cdot \boldsymbol{x})$$

- and then the output is computed from the hidden units:

$$y = f(\boldsymbol{w}_o \cdot \boldsymbol{h})$$

- the function f is called the **activation**
  - for now, let's assume that f is the sigmoid function, so the hidden units and output unit can be seen as LR classifiers

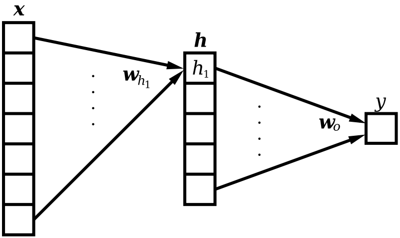
two-layered feedforward NN: illustration



implementation in NumPy

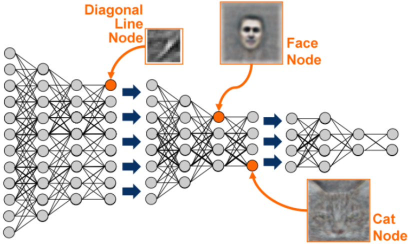
- recall that a sequence of dot products can be seen as a matrix multiplication
- in NumPy, the NN can be expressed compactly with matrix multiplication

```
h = sigmoid(Wh.dot(x))
y = sigmoid(Wo.dot(h))
```

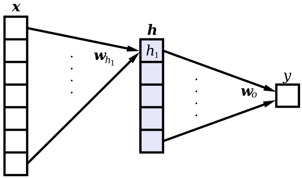


“deep learning”

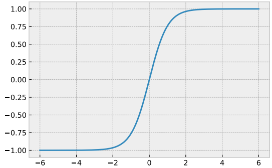
- why the “deep” in “deep learning”?
- although a single hidden layer is sufficient in theory, in practice it can be better to have several hidden layers



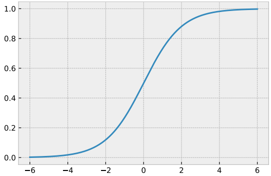
common activation functions: hidden layers



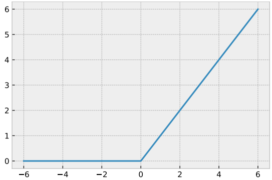
- **tanh** (hyperbolic tangent):



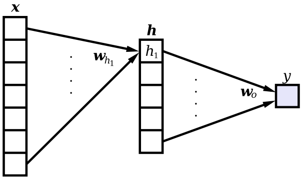
- **sigmoid** (logistic):



- **ReLU** (rectified linear unit):

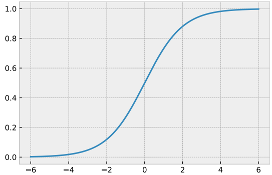


output layer: binary classification

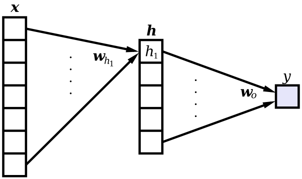


- **sigmoid** (logistic) for a **binary** classifier:

$$P(\text{positive}|\boldsymbol{x}) = \frac{1}{1 + e^{-\text{score}}}$$

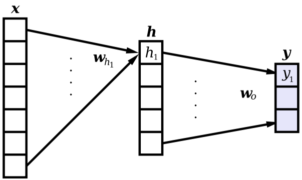


output layer: regression



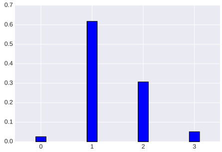
- what activation should we use for a **regression** problem?

output layer: multiclass classification



- **softmax** for a **multiclass** classifier (that is, more than 2 output classes):

$$P(\text{output} \; = \text{class } i|\boldsymbol{x}) = \frac{e^{\text{score}_i}}{\sum_k e^{\text{score}_k}}$$



in scikit-learn

- `sklearn.neural_network.MLPClassifier`
- `sklearn.neural_network.MLPRegressor`

Overview

training feedforward neural networks

- training a NN consists of finding the weights in the layers
- so how do we find those weights?

training feedforward neural networks

- training a NN consists of finding the weights in the layers
- so how do we find those weights?
- as we did for the SVC and LR!
  - state an **objective function** including a **loss** and possibly a **regularizer**
  - apply an **optimization algorithm** to find the weights that minimize the objective

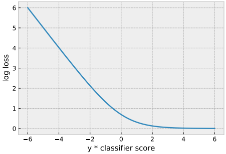
SGD: pseudocode

initialize $\boldsymbol{w}$
repeat ...
    pick a training instance $(\boldsymbol{x}_i, y_i)$
    compute gradient ∇f_i of the loss for current instance $(\boldsymbol{x}_i, y_i)$
    $\boldsymbol{w} = \boldsymbol{w} - \eta \cdot \nabla f_i(\boldsymbol{w})$
return $\boldsymbol{w}$

loss functions

- **log loss** for a binary classifier

$$\text{Loss} = -\log(P(\text{output} = y))$$

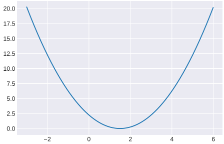


- **cross-entropy loss** for multiclass

$$\text{Loss} = -\log(P(\text{output} = y))$$

- **squared error loss** for regression

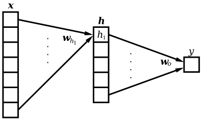
$$\text{Loss} = (y - o)^2$$



example

- let's use two layers with sigmoid units, and then the log loss

$$\boldsymbol{h} = \sigma(\boldsymbol{W}_h \cdot \boldsymbol{x})$$
$$y = \sigma(\boldsymbol{W}_o \cdot \boldsymbol{h})$$
$$\text{Loss} = -\log(y)$$



- so the whole thing becomes

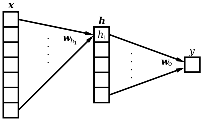
$$\text{Loss} = -\log\sigma(\boldsymbol{W}_o \cdot \sigma(\boldsymbol{W}_h \cdot \boldsymbol{x}))$$

- now, to do gradient descent, we need to compute gradients w.r.t. the weights $\boldsymbol{W_h}$ and $\boldsymbol{W_o}$

example

- let's use two layers with sigmoid units, and then the log loss

$$\boldsymbol{h} = \sigma(\boldsymbol{W}_h \cdot \boldsymbol{x})$$
$$y = \sigma(\boldsymbol{W}_o \cdot \boldsymbol{h})$$
$$\text{Loss} = -\log(y)$$



- so the whole thing becomes

$$\text{Loss} = -\log \sigma(\boldsymbol{W}_o \cdot \sigma(\boldsymbol{W}_h \cdot \boldsymbol{x}))$$

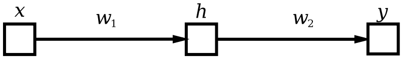
- now, to do gradient descent, we need to compute gradients w.r.t. the weights $\boldsymbol{W}_h$ and $\boldsymbol{W}_o$
- **ouch!** it looks completely unwieldy!

the chain rule of derivatives/gradients

- NNs consist of functions applied to the output of other functions
- the **chain rule** is a useful trick from calculus that can be used in such situations
  - assume that we apply the function f to the output of g
  - then the chain rule says how we can compute the gradient of the combination:

$$\text{gradient of } f(g(x)) = \text{gradient of } f(g) \cdot \text{gradient of } g(x)$$

chain rule example

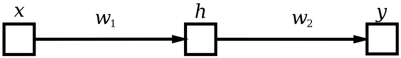


- let's say we have defined a simple neural network:

$$h = f_1(w_1 \cdot x)$$

$$\text{Loss} = f_2(w_2 \cdot h)$$

chain rule example



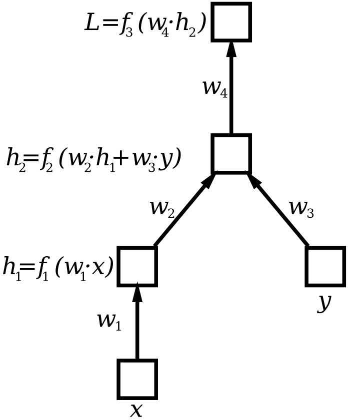
- let's say we have defined a simple neural network:

$$h = f_1(w_1 \cdot x)$$
$$\text{Loss} = f_2(w_2 \cdot h)$$

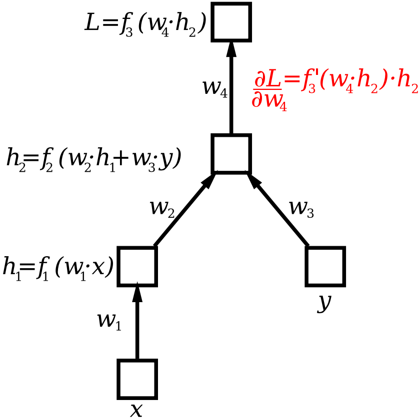
- then we can compute the gradients with respect to w_1 and w_2 using the chain rule:

$$\frac{\partial \text{Loss}}{\partial w_2} = f_2'(w_2 \cdot h) \cdot h$$
$$\frac{\partial \text{Loss}}{\partial w_1} = f_2'(w_2 \cdot h) \cdot w_2 \cdot f_1'(w_1 \cdot x) \cdot x$$

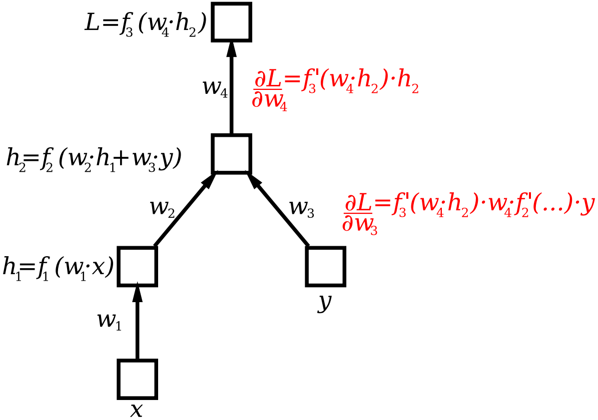
slightly more complicated computational graph



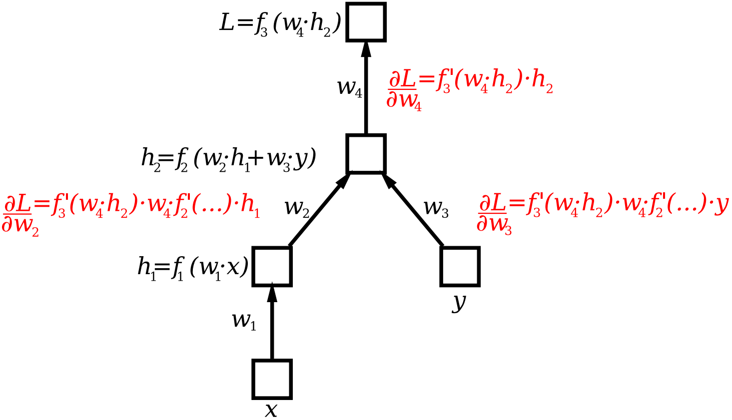
slightly more complicated computational graph



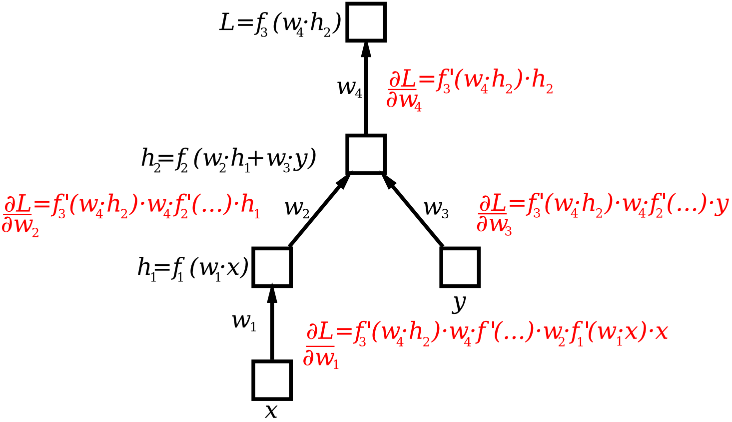
slightly more complicated computational graph



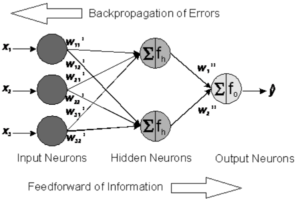
slightly more complicated computational graph



slightly more complicated computational graph



the general recipe: backpropagation



- using the chain rule, the gradients of the weights in each layer can be computed from the gradients of the layers after it
- this trick is called **backpropagation**
- it's not difficult, but involves a lot of book-keeping
- fortunately, there are computer programs that can do the algebra for us!
  - in NN software, we usually just declare the network and the loss, then the gradients are computed under the hood

training efficiency of NNs

- our previous classifiers took seconds or minutes to train
- NNs tend to take minutes, hours, days, weeks ...
  - depending on the complexity of the network and the amount of training data
- NNs use a lot of linear algebra (matrix multiplications) so it can be useful to work to speed up the math
  - parallelize as much as possible
  - use optimized math libraries
  - use a GPU
  - in short: you're better of using a specialized NN library

Overview

Introduction

Getting rid of linear inseparability

neural networks, basic ideas

training feedforward neural networks

overview of neural network libraries

tricks of the trade

neural network software: Python

- scikit-learn currently has quite limited support for NNs
- the main NN software in the Python world used to be **Theano**
  - developed by Yoshua Bengio's group in Montréal
  - `http://deeplearning.net/software/theano`
- the major players have released their own libraries in the last few years:
  - Google: TensorFlow
  - Facebook: PyTorch
  - Microsoft: CNTK
- these toolkits provide "building blocks" such as layers, activations, losses, regularizers, optimizers, ...

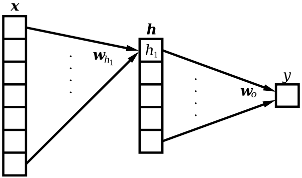
neural network software: Python (2)

- TensorFlow etc. do a lot of useful math stuff, and integrate nicely with the GPU, but they can be a bit low-level
- so there are libraries that create a more high-level interface, a bit similar to scikit-learn
  - **Keras**
  - `conda install keras`
  - required for Assignment 5

abstraction

low-level control

coding example with Keras




```
keras_model = Sequential()

n_hidden = 3
keras_model.add(Dense(input_dim=X.shape[1],
                      output_dim=n_hidden))
keras_model.add(Activation("sigmoid"))

keras_model.add(Dense(input_dim=n_hidden,
                      output_dim=1))
keras_model.add(Activation("sigmoid"))

keras_model.compile(loss='binary_crossentropy',
                    optimizer='rmsprop')

keras_model.fit(X, Y)
```

Overview

processing one instance at a time is inefficient

- as we have discussed, it is more efficient to carry out large-scale linear algebra operations
- but SGD works incrementally, one instance at a time!
- **minibatch** gradient descent: small subsets instead of single instances
- in Keras:

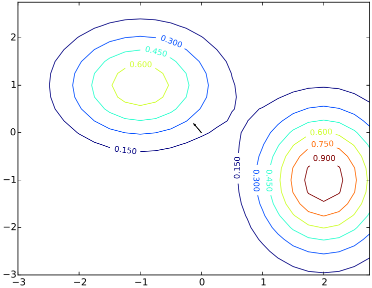
```
model.fit(X, Y, batch_size=400)
```

minibatch gradient descent: pseudocode

initialize $\boldsymbol{w}$
repeat ...
  select a small batch (subset) $\boldsymbol{X}_b$, Y_b from the $\boldsymbol{X}$, Y
  compute gradient ∇f_b of the loss for current batch $\boldsymbol{X}_b$, Y_b
  $\boldsymbol{w} = \boldsymbol{w} - \eta \cdot \nabla f_i(\boldsymbol{w})$
return $\boldsymbol{w}$

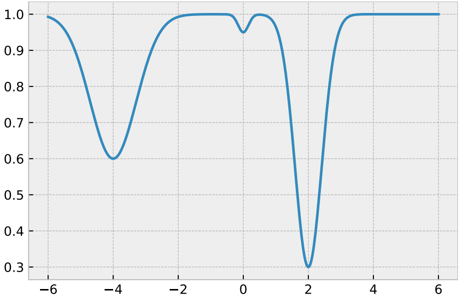
optimizing NNs

- unlike the linear classifiers we studied previously, NNs have non-convex objective functions with a lot of local minima
- so the end result depends on initialization



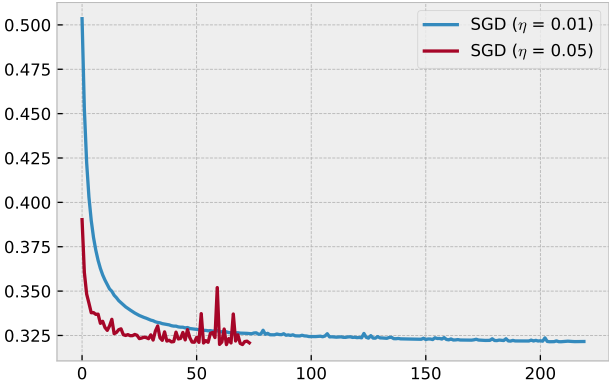
plateaus in the objective

- NN objectives tend to have plateaus:



- this irregular shape makes it hard to set the learning rate

example: two different learning rates



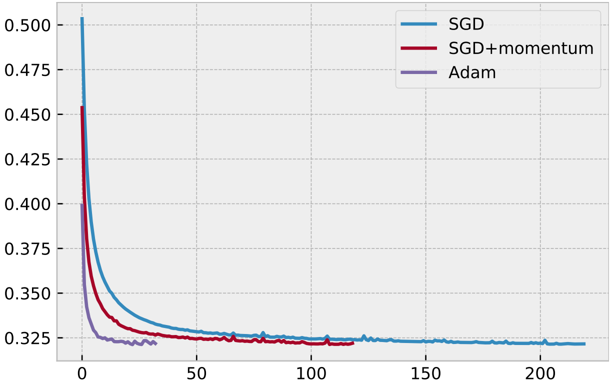
adaptive gradient updates

- **adaptive** gradient descent methods control η to accelerate and slow down when necessary
- popular adaptive methods: **Adam**, **Adagrad**, **RMSProp**, ...
- in Keras:

```
model.compile(..., optimizer='adam', ...)
```

- see Sebastian Ruder's report An overview of gradient descent optimization algorithms for an overview

example: comparison of optimizers



avoiding overfitting

- we've already seen how to apply a **regularizer** for logistic regression and SVC

$$\sum \mathrm{Loss}(\mathrm{x}_i, y_i, \boldsymbol{w}) + R(\boldsymbol{w})$$

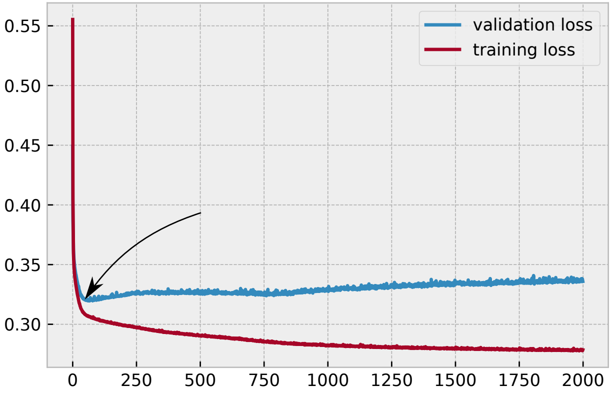
  where $R(\boldsymbol{w})$ can be $\|\boldsymbol{w}\|^2$, for instance
- in Keras:

```
from keras import regularizers
regularizer = regularizers.l2(0.001)
```

- in the neural network world, there are a few other methods that are also popular, such as **early stopping** and **dropout**

Early stopping

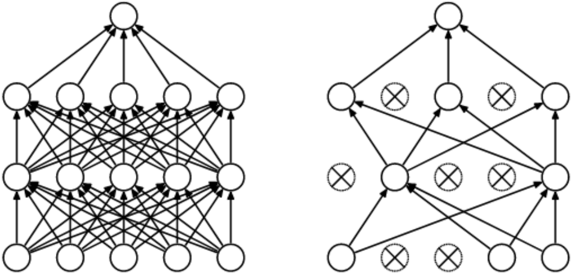
- reserve a held-out development set (validation set)
- terminate training when there is no improvement on the held-out data



early stopping: Keras

```
from keras import callbacks

cb = callbacks.EarlyStopping(monitor='val_loss', min_delta=0,
                             patience=10, verbose=0, mode='auto')

...

model.fit(X, Y, epochs=1000, batch_size=400, verbose=2,
          validation_split=0.1, callbacks=[cb])
```

dropout

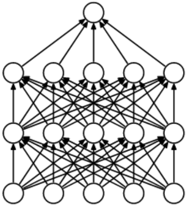
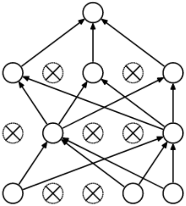


(a) Standard Neural Net (b) After applying dropout.

[source]

dropout in neural networks

- **dropout** is often used to reduce the risk of overfitting in neural networks

(a) Standard Neural Net (b) After applying dropout.

[source]

- why does it work?
- Srivastava et al. (2014) motivate dropout in terms of ensembles
  - if there are N connections in the model, we can see it as an ensemble of 2^N different models (subsets of connections)

dropout: Keras

```
from keras.layers import Dropout

model = Sequential()
model.add(Dense(... something ...))
model.add(Dropout(0.1))
model.add(Dense(... something ...))

model.compile(...)
modle.fit(...)
```

Review of neural network

- Could you explain the terms used in NN?
  - input layer, output layer, hidden layer(s), activation function, learning, cost function, parameters, feedforward, backpropagation
- Motivate different ways to overcome overfitting!

Next lecture

- Convolutional neural networks
- Application example: image analysis