CSI5180. Machine Learning for Bioinformatics Applications

Deep learning — encoding and transfer learning

by

Marcel Turcotte

Version **November 12, 2019**

Preamble

Preamble

Deep learning — encoding and transfer learning

In this lecture, we further investigate deep learning. We review diverse methods to encode the data for these artificial neural networks. We present the concept of embeddings and specifically embeddings for biological sequences. Finally, we discuss the concept of transfer learning.

General objective :

- **Explain** the various ways to encode data for deep networks

Learning objectives

- **Explain** the concept of embeddings
- **Describe** how to implement transfer learning
- **Justify** the application of transfer learning

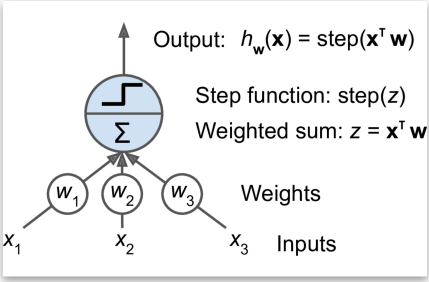
Reading:

- Ehsaneddin Asgari and Mohammad R K Mofrad, Continuous distributed representation of biological sequences for deep proteomics and genomics, *PLoS One* **10**:11, e0141287, 2015.
- Wang, S., Li, Z., Yu, Y., Xu, J. Folding Membrane Proteins by Deep Transfer Learning. *Cell Systems* **5**:3, 202, 2017.

Plan

Summary

Summary - threshold logic unit

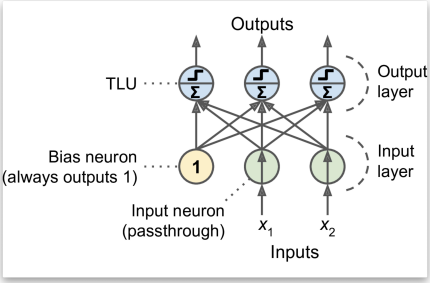


Source: [3] Figure 10.4

- **Model**

$$h_w(x) = \phi(x^T w)$$

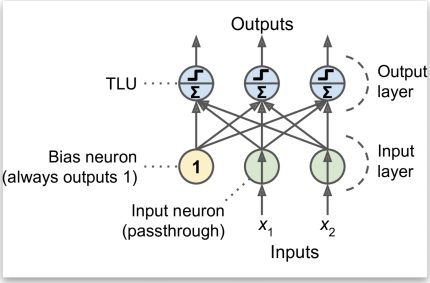
Summary - Perceptron



Source: [3] Figure 10.5

- A **Perceptron** consists of a single layer of threshold logic units.

Summary - Perceptron



Source: [3] Figure 10.5

- A **Perceptron** consists of a single layer of threshold logic units.
- It computes the following function:

$$h_{W,b}(X) = \phi(WX + b)$$

Summary - Definitions

- **Input neuron:** a special type of neuron that simply **returns the value of its input**.

Summary - Definitions

- **Input neuron:** a special type of neuron that simply **returns the value of its input**.
- **Bias neuron:** a neuron that **always return 1**.

Summary - Definitions

- **Input neuron:** a special type of neuron that simply **returns the value of its input**.
- **Bias neuron:** a neuron that **always return 1**.
- **Fully connected layer** or **dense layer: all** the neurons are connected to **all** the neurons of the previous layer.

Summary - Definitions

- **Input neuron:** a special type of neuron that simply **returns the value of its input**.
- **Bias neuron:** a neuron that **always return 1**.
- **Fully connected layer** or **dense layer: all** the neurons are connected to **all** the neurons of the previous layer.
- **X: input matrix** (rows are instances, columns are features).

Summary - Definitions

- **Input neuron:** a special type of neuron that simply **returns the value of its input**.
- **Bias neuron:** a neuron that **always return 1**.
- **Fully connected layer** or **dense layer: all** the neurons are connected to **all** the neurons of the previous layer.
- **X: input matrix** (rows are instances, columns are features).
- **W: weight matrix** (# rows corresponds to the number of inputs, # columns corresponds to the number of neurons in the output layer).

Summary - Definitions

- **Input neuron:** a special type of neuron that simply **returns the value of its input**.
- **Bias neuron:** a neuron that **always return 1**.
- **Fully connected layer** or **dense layer: all** the neurons are connected to **all** the neurons of the previous layer.
- **X: input matrix** (rows are instances, columns are features).
- **W: weight matrix** (# rows corresponds to the number of inputs, # columns corresponds to the number of neurons in the output layer).
- **b: bias vector** (same size as the number of neurons in the output layer).

Summary - Definitions

- **Input neuron:** a special type of neuron that simply **returns the value of its input**.
- **Bias neuron:** a neuron that **always return 1**.
- **Fully connected layer** or **dense layer: all** the neurons are connected to **all** the neurons of the previous layer.
- **X: input matrix** (rows are instances, columns are features).
- **W: weight matrix** (# rows corresponds to the number of inputs, # columns corresponds to the number of neurons in the output layer).
- **b: bias vector** (same size as the number of neurons in the output layer).
- **Activation function:** maps its input domain to a restricted set of values (heavyside and sign are commonly used with threshold logic unit perceptrons).

Summary - Multilayer Perceptron

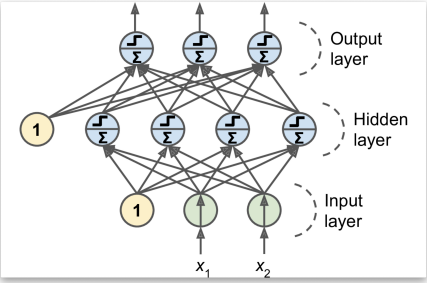
- A **two-layer** perceptron computes:

$$y = f_2(f_1(X))$$

where

$$f_l(Z) = \phi(W_l Z + b_l)$$

ϕ is an activation function, typically one of: **hyperbolic tangent**, **Rectified Linear Unit function**, **sigmoid**, etc. W is a weight matrix, X is an input matrix, and b is a bias vector. In the context of artificial neural networks, matrices are called **tensors**.



Source: [3] Figure 10.7

Summary - Multilayer Perceptron

- A k-**layer** perceptron computes the following function:

$$y = f_k(\ldots f_2(f_1(X))\ldots)$$

where

$$f_l(Z) = \phi(W_l Z + b_l)$$

Keras

Using Keras

- https://keras.io
  (François Chollet/Google/2015 1st release)

Using Keras

- https://keras.io
  (François Chollet/Google/2015 1st release)
- **Personally**, I find it easier to install and maintain Keras using a package manager, such as **Conda** (specifically, I use **Anaconda**).

Using Keras

- `https://keras.io`
  (François Chollet/Google/2015 1st release)
- **Personally**, I find it easier to install and maintain Keras using a package manager, such as **Conda** (specifically, I use **Anaconda**).
- Easy to use, yet **powerfull** and **efficient** (makes use of GPUs if available)

Using Keras

- https://keras.io
  (François Chollet/Google/2015 1st release)
- **Personally**, I find it easier to install and maintain Keras using a package manager, such as **Conda** (specifically, I use **Anaconda**).
- Easy to use, yet **powerfull** and **efficient** (makes use of GPUs if available)
- Two main API: **Sequential** and **Functional**

Sequential API

```
from keras.models import Sequential

model = Sequential()
```

Sequential API

```
from keras.models import Sequential

model = Sequential()
```

```
from keras.layers import Dense

model.add(Dense(units=64, activation='relu', input_dim=100))
model.add(Dense(units=10, activation='softmax'))
```

Sequential API

```
from keras.models import Sequential

model = Sequential()
```

```
from keras.layers import Dense

model.add(Dense(units=64, activation='relu', input_dim=100))
model.add(Dense(units=10, activation='softmax'))
```

```
model.compile(loss='categorical_crossentropy',
              optimizer='sgd',
              metrics=['accuracy'])
```

Sequential API

```
from keras.models import Sequential

model = Sequential()
```

```
from keras.layers import Dense

model.add(Dense(units=64, activation='relu', input_dim=100))
model.add(Dense(units=10, activation='softmax'))
```

```
model.compile(loss='categorical_crossentropy',
              optimizer='sgd',
              metrics=['accuracy'])
```

```
model.fit(x_train, y_train, epochs=5, batch_size=32)
```

Sequential API

```
from keras.models import Sequential

model = Sequential()
```

```
from keras.layers import Dense

model.add(Dense(units=64, activation='relu', input_dim=100))
model.add(Dense(units=10, activation='softmax'))
```

```
model.compile(loss='categorical_crossentropy',
              optimizer='sgd',
              metrics=['accuracy'])
```

```
model.fit(x_train, y_train, epochs=5, batch_size=32)
```

```
loss_and_metrics = model.evaluate(x_test, y_test)
```

Functional API

```
from keras.layers import Input, Dense
from keras.models import Model

# This returns a tensor
inputs = Input(shape=(784,))

# a layer instance is callable on a tensor, and returns a tensor
output_1 = Dense(64, activation='relu')(inputs)
output_2 = Dense(64, activation='relu')(output_1)
predictions = Dense(10, activation='softmax')(output_2)

# This creates a model that includes
# the Input layer and three Dense layers
model = Model(inputs=inputs, outputs=predictions)
model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
model.fit(data, labels)  # starts training
```

Preprocessing

Scaling

- As discussed at the begining of the term, it is almost always a good idea to **scale the input data**.
  - **Custom code**
  - **sklearn.preprocessing.StandardScaler**
  - **keras.layers.Lambda**
  - **Standardization layer**

keras.layers.Lambda

```
means = np.mean(X_train, axis=0, keepdims=True)
stds = np.std(X_train, axis=0, keepdims=True)

eps = keras.backend.epsilon()

model = keras.models.Sequential([

    keras.layers.Lambda(lambda inputs: (inputs - means) / (stds + eps)),

    [...] # other layers
])
```

Source: [3] §11

Standardization layer

```
class Standardization(keras.layers.Layer):
    def adapt(self, data_sample):
        self.means_ = np.mean(data_sample, axis=0, keepdims=True)
        self.stds_ = np.std(data_sample, axis=0, keepdims=True)
    def call(self, inputs):
        return (inputs-self.means_)/(self.stds_+keras.backend.epsilon())
```

Standardization layer

```
class Standardization(keras.layers.Layer):
    def adapt(self, data_sample):
        self.means_ = np.mean(data_sample, axis=0, keepdims=True)
        self.stds_ = np.std(data_sample, axis=0, keepdims=True)
    def call(self, inputs):
        return (inputs-self.means_)/(self.stds_+keras.backend.epsilon())
```

```
std_layer = Standardization()
std_layer.adapt(data_sample)
```

Standardization layer

```
class Standardization(keras.layers.Layer):
    def adapt(self, data_sample):
        self.means_ = np.mean(data_sample, axis=0, keepdims=True)
        self.stds_ = np.std(data_sample, axis=0, keepdims=True)
    def call(self, inputs):
        return (inputs-self.means_)/(self.stds_+keras.backend.epsilon())
```

```
std_layer = Standardization()
std_layer.adapt(data_sample)
```

```
model = keras.Sequential()
model.add(std_layer)
... # create the rest of the model
model.compile([...])
model.fit([...])
```

Standardization layer

```
class Standardization(keras.layers.Layer):
    def adapt(self, data_sample):
        self.means_ = np.mean(data_sample, axis=0, keepdims=True)
        self.stds_ = np.std(data_sample, axis=0, keepdims=True)
    def call(self, inputs):
        return (inputs-self.means_)/(self.stds_+keras.backend.epsilon())
```

```
std_layer = Standardization()
std_layer.adapt(data_sample)
```

```
model = keras.Sequential()
model.add(std_layer)
... # create the rest of the model
model.compile([...])
model.fit([...])
```

Source: [3] §11

Categorical data

```
from numpy import array
import numpy as np

from sklearn.preprocessing import LabelEncoder
from keras.utils import to_categorical

data = ['T','T','C','T','G','G','C','A','C','T','T','G']

values = array(data)

label_encoder = LabelEncoder ()

integer_encoded = label_encoder.fit_transform(values)
data_encoded = to_categorical(integer_encoded)
```

Categorical data

```python
print(data_encoded)
```

```
[[0. 0. 0. 1.]
 [0. 0. 0. 1.]
 [0. 1. 0. 0.]
 [0. 0. 0. 1.]
 [0. 0. 1. 0.]
 [0. 0. 1. 0.]
 [0. 1. 0. 0.]
 [1. 0. 0. 0.]
 [0. 1. 0. 0.]
 [0. 0. 0. 1.]
 [0. 0. 0. 1.]
 [0. 0. 1. 0.]]
```

Embeddings

- "An **embedding** is a **trainable** dense **vector** that represents a category." [3] §13

Embeddings

- "An **embedding** is a **trainable** dense **vector** that represents a category." [3] §13
- With the **one hot encoding**, we used a sparse encoding with **one dimension per category**, e.g. A = [1,0,0,0], to avoid creating false associations between categories.

Embeddings

- "An **embedding** is a **trainable** dense **vector** that represents a category." [3] §13
- With the **one hot encoding**, we used a sparse encoding with **one dimension per category**, e.g. A = [1,0,0,0], to avoid creating false associations between categories.
- With **embeddings**, the philosophy is the other way around, we want **categories** that are **similar** to have **similar vector representations**.

Embeddings

- "An **embedding** is a **trainable** dense **vector** that represents a category." [3] §13
- With the **one hot encoding**, we used a sparse encoding with **one dimension per category**, e.g. A = [1,0,0,0], to avoid creating false associations between categories.
- With **embeddings**, the philosophy is the other way around, we want **categories** that are **similar** to have **similar vector representations**.
  - The **representation** is **learnt from the data**!

Embeddings

- "An **embedding** is a **trainable** dense **vector** that represents a category." [3] §13
- With the **one hot encoding**, we used a sparse encoding with **one dimension per category**, e.g. A = [1,0,0,0], to avoid creating false associations between categories.
- With **embeddings**, the philosophy is the other way around, we want **categories** that are **similar** to have **similar vector representations**.
  - The **representation** is **learnt from the data**!
  - Initially, **each category** is assigned a **random vector**.

Embeddings

- "An **embedding** is a **trainable** dense **vector** that represents a category." [3] §13
- With the **one hot encoding**, we used a sparse encoding with **one dimension per category**, e.g. A = [1,0,0,0], to avoid creating false associations between categories.
- With **embeddings**, the philosophy is the other way around, we want **categories** that are **similar** to have **similar vector representations**.
  - The **representation** is **learnt from the data**!
  - Initially, **each category** is assigned a **random vector**.
  - **During learning**, gradient descent will make the vector representations of similar categories more similar one to another.

Embeddings

- "An **embedding** is a **trainable** dense **vector** that represents a category." [3] §13
- With the **one hot encoding**, we used a sparse encoding with **one dimension per category**, e.g. A = [1,0,0,0], to avoid creating false associations between categories.
- With **embeddings**, the philosophy is the other way around, we want **categories** that are **similar** to have **similar vector representations**.
  - The **representation** is **learnt from the data**!
  - Initially, **each category** is assigned a **random vector**.
  - **During learning**, gradient descent will make the vector representations of similar categories more similar one to another.
- **Why?**

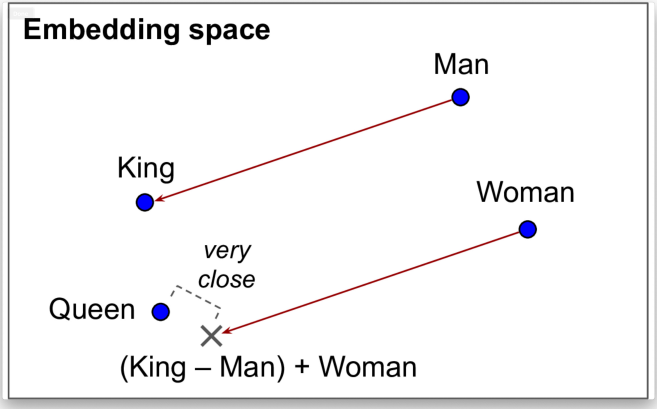
Embeddings

- "An **embedding** is a **trainable** dense **vector** that represents a category." [3] §13
- With the **one hot encoding**, we used a sparse encoding with **one dimension per category**, e.g. A = [1,0,0,0], to avoid creating false associations between categories.
- With **embeddings**, the philosophy is the other way around, we want **categories** that are **similar** to have **similar vector representations**.
  - The **representation** is **learnt from the data**!
  - Initially, **each category** is assigned a **random vector**.
  - **During learning**, gradient descent will make the vector representations of similar categories more similar one to another.
- **Why?**
  - A **better representation** can accelerate learning and make more accurate predictions.

Embeddings

- "An **embedding** is a **trainable** dense **vector** that represents a category." [3] §13
- With the **one hot encoding**, we used a sparse encoding with **one dimension per category**, e.g. A = [1,0,0,0], to avoid creating false associations between categories.
- With **embeddings**, the philosophy is the other way around, we want **categories** that are **similar** to have **similar vector representations**.
  - The **representation** is **learnt from the data**!
  - Initially, **each category** is assigned a **random vector**.
  - **During learning**, gradient descent will make the vector representations of similar categories more similar one to another.
- **Why?**
  - A **better representation** can accelerate learning and make more accurate predictions.
  - **Embeddings** can be **reused**! [A form of **transfer learning**]

Word embbedings



Source: [3] Figure 13.5

"**Man** is to **King** as **Woman** is to **Queen**"

2013

- **Distributed Representations of Words and Phrases and their Compositionality**
- Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, Jeffrey Dean
- https://arxiv.org/abs/1310.4546
  - "Somewhat surprisingly, **many of these patterns** can be represented as **linear translations**."
  - "For example, the result of a vector calculation vec("**Madrid**") - vec("**Spain**") + vec("**France**") is closer to vec("**Paris**") than to any other word vector."

Hypothetical example

- **Imagine** that a (coding) DNA sequence is divided into **3-letter words**.

Hypothetical example

- **Imagine** that a (coding) DNA sequence is divided into **3-letter words**.
- There would be **64** such words (64 categories).

Hypothetical example

- **Imagine** that a (coding) DNA sequence is divided into **3-letter words**.
- There would be **64** such words (64 categories).
- Initially, each category is assigned a **random vector**.

Hypothetical example

- **Imagine** that a (coding) DNA sequence is divided into **3-letter words**.
- There would be **64** such words (64 categories).
- Initially, each category is assigned a **random vector**.
- **During learning**, 3-letter words corresponding to **codons** encoding the same **amino acid** would see their vector representation be made more and more similar.

Embeddings in bioinformatics

- Bepler, T. & Berger, B. Learning protein sequence embeddings using information from structure. *arXiv.org* cs.LG, (2019). †
- Woloszynek, S., Zhao, Z., Chen, J. & Rosen, G. L. 16S rRNA sequence embeddings: Meaningful numeric feature representations of nucleotide sequences that are convenient for downstream analyses. *PLoS Comput Biol* **15**, (2019). †

Embeddings in bioinformatics

- Asgari, E. & Mofrad, M. R. K. Continuous Distributed Representation of Biological Sequences for Deep Proteomics and Genomics. *PLoS ONE 10*, (2015).
- Menegaux, R. & Vert, J.-P. Continuous Embeddings of DNA Sequencing Reads and Application to Metagenomics. *J Comput Biol* **26**, cmb.2018.0174518 (2019).
- Min, X., Zeng, W., Chen, N., Chen, T. & Jiang, R. Chromatin accessibility prediction via convolutional long short-term memory networks with k-mer embedding. *Bioinformatics* **33**, I92I101 (2017).
- Hamid, M.-N. & Friedberg, I. Identifying Antimicrobial Peptides using Word Embedding with Deep Recurrent Neural Networks. *Bioinformatics* **25**, 3389 (2018).
- Shen, Z., Bao, W. & Huang, D.-S. Recurrent Neural Network for Predicting Transcription Factor Binding Sites. *Sci Rep* **8**, 15270 (2018).

Transfer learning

Transfer learning

- **Transfer learning** is taking a **sizable portion** of a deep network **trained for one application**, and slightly modify it before using it in **another application**.

Transfer learning

- **Transfer learning** is taking a **sizable portion** of a deep network **trained for one application**, and slightly modify it before using it in **another application**.
  - **Why?**

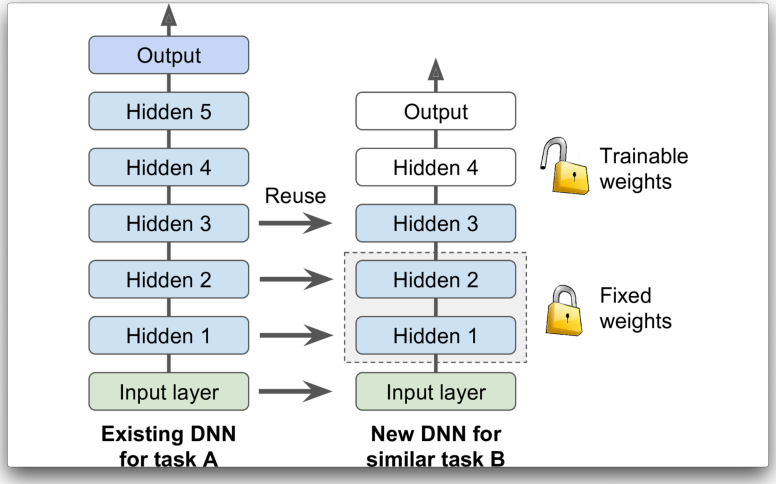
Transfer learning

- **Transfer learning** is taking a **sizable portion** of a deep network **trained for one application**, and slightly modify it before using it in **another application**.
  - **Why?**
    - An **obvious** reason would be to **speed up** the learning process.

Transfer learning

- **Transfer learning** is taking a **sizable portion** of a deep network **trained for one application**, and slightly modify it before using it in **another application**.
  - **Why?**
    - An **obvious** reason would be to **speed up** the learning process.
    - A **much more interesting** reason (IMHO) is to apply deep learning for applications where the number of examples **is low**.

Transfer learning



Source: [3] Figure 11.4

Transfer learning in bioinformatics

Computational ***elucidation of membrane protein (MP) structures is challenging*** *partially due to* ***lack of sufficient solved structures*** *for homology modeling.*

Transfer learning in bioinformatics

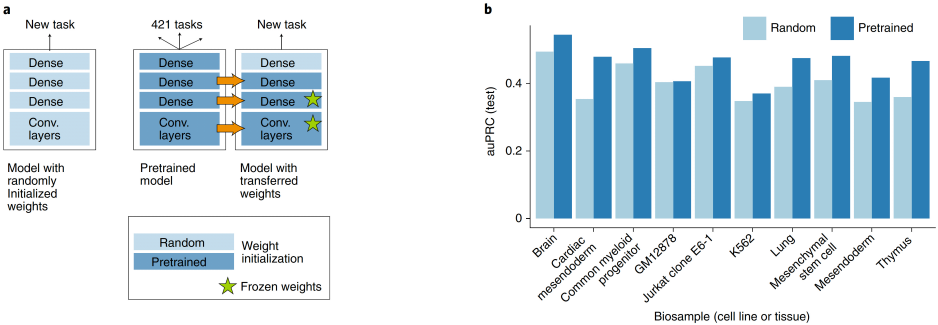
Computational ***elucidation of membrane protein (MP) structures is challenging*** *partially due to* ***lack of sufficient solved structures*** *for homology modeling. Here, we describe a high-throughput deep transfer learning method that first predicts MP contacts by* ***learning from non-MPs*** *and then predicts 3D structure models using the predicted contacts as distance restraints.*

Transfer learning in bioinformatics

Computational ***elucidation of membrane protein (MP) structures is challenging*** *partially due to* ***lack of sufficient solved structures*** *for homology modeling. Here, we describe a high-throughput deep transfer learning method that first predicts MP contacts by* ***learning from non-MPs*** *and then predicts 3D structure models using the predicted contacts as distance restraints.*

- Wang, S., Li, Z., Yu, Y., Xu, J. Folding Membrane Proteins by Deep Transfer Learning. *Cell Systems* **5**(3), 202, 2017.

Kipoi



- Ziga Avsec, Roman Kreuzhuber, Johnny Israeli, Nancy Xu, Jun Cheng, Avanti Shrikumar, Abhimanyu Banerjee, Daniel S Kim, Thorsten Beier, Lara Urban, Anshul Kundaje, Oliver Stegle, and Julien Gagneur. The Kipoi repository accelerates community exchange and reuse of predictive models for genomics. *Nat Biotechnol*, **37**(6):592600, Jun 2019.

Transfer learning

[4] §8.7:

1. You **build a deep model** on the **original big dataset** ([non-membrane proteins]).

Transfer learning

[4] §8.7:

1. You **build a deep model** on the **original big dataset** ([non-membrane proteins]).
2. You **compile a much smaller labelled dataset** for your **second model** ([membrane proteins]).

Transfer learning

[4] §8.7:

1. You **build a deep model** on the **original big dataset** ([non-membrane proteins]).
2. You **compile a much smaller labelled dataset** for your **second model** ([membrane proteins]).
3. You **remove the last one or several layers** from the first model. Usually, these are layers responsible for the classification or regression; they usually follow the embedding layer.

Transfer learning

[4] §8.7:

1. You **build a deep model** on the **original big dataset** ([non-membrane proteins]).
2. You **compile a much smaller labelled dataset** for your **second model** ([membrane proteins]).
3. You **remove the last one or several layers** from the first model. Usually, these are layers responsible for the classification or regression; they usually follow the embedding layer.
4. You **replace the removed layers with new layers** adapted for your new problem.

Transfer learning

[4] §8.7:

1. You **build a deep model** on the **original big dataset** ([non-membrane proteins]).
2. You **compile a much smaller labelled dataset** for your **second model** ([membrane proteins]).
3. You **remove the last one or several layers** from the first model. Usually, these are layers responsible for the classification or regression; they usually follow the embedding layer.
4. You **replace the removed layers with new layers** adapted for your new problem.
5. You **"freeze" the parameters** of the layers remaining from the **first model**.

Transfer learning

[4] §8.7:

1. You **build a deep model** on the **original big dataset** ([non-membrane proteins]).
2. You **compile a much smaller labelled dataset** for your **second model** ([membrane proteins]).
3. You **remove the last one or several layers** from the first model. Usually, these are layers responsible for the classification or regression; they usually follow the embedding layer.
4. You **replace the removed layers with new layers** adapted for your new problem.
5. You **"freeze" the parameters** of the layers remaining from the **first model**.
6. You **use your smaller labelled dataset** and **gradient descent** to train the parameters of only the new layers.

Transfer learning with Keras

[3] §11:

```
model_A = keras.models.load_model("my_model_A.h5")
model_B_on_A = keras.models.Sequential(model_A.layers[:-1])
model_B_on_A.add(keras.layers.Dense(1, activation="sigmoid"))
```

Transfer learning with Keras

[3] §11:

```
model_A = keras.models.load_model("my_model_A.h5")
model_B_on_A = keras.models.Sequential(model_A.layers[:-1])
model_B_on_A.add(keras.layers.Dense(1, activation="sigmoid"))
```

Alternatively:

```
model_A_clone = keras.models.clone_model(model_A)
model_A_clone.set_weights(model_A.get_weights())
```

Transfer learning with Keras

[3] §11:

```
for layer in model_B_on_A.layers[:-1]:
    layer.trainable = False

model_B_on_A.compile(loss="binary_crossentropy", optimizer="sgd",
                     metrics=["accuracy"])

history = model_B_on_A.fit(X_train_B, y_train_B, epochs=4,
                           validation_data=(X_valid_B, y_valid_B))
```

Transfer learning with Keras

[3] §11:

```
for layer in model_B_on_A.layers[:-1]:
    layer.trainable = True

optimizer = keras.optimizers.SGD(lr=1e-4) # the default lr is 1e-2

model_B_on_A.compile(loss="binary_crossentropy", optimizer=optimizer,
                     metrics=["accuracy"])

history = model_B_on_A.fit(X_train_B, y_train_B, epochs=16,
                           validation_data=(X_valid_B, y_valid_B))
```

Remarks

- **Transfer learning** is possibly **unique** to deep learning methods.

Remarks

- **Transfer learning** is possibly **unique** to deep learning methods.
- When the number of **training examples** available is **too small** to justify using deep learning, there might be a sufficiently similar problem for which a **lot of data** is available.

Prologue

Summary

- **Embeddings** are representations that are learnt from data.

Summary

- **Embeddings** are representations that are learnt from data.
- **Transfer learning** allows for the application of deep learning to problems for which the number of training data is low.

Next module

- **Deep learning** - architectures

References

Ehsaneddin Asgari and Mohammad R K Mofrad.
Continuous distributed representation of biological sequences for deep proteomics and genomics.
PLoS One, 10(11):e0141287, 2015.

François Chollet.
Deep learning with Python.
Manning Publications, 2017.

Aurélien Géron.
Hands-on Machine Learning with Scikit-Learn, Keras, and TensorFlow.
O'Reilly Media, 2nd edition, 2019.

Andriy Burkov.
The Hundred-Page Machine Learning Book.
Andriy Burkov, 2019.

Yann LeCun, Yoshua Bengio, and Geoffrey Hinton.
Deep learning.
Nature, 521(7553):436–44, May 2015.

References

Prabina Kumar Meher, Tanmaya Kumar Sahu, Shachi Gahoi, Subhrajit Satpathy, and Atmakuri Ramakrishna Rao.
Evaluating the performance of sequence encoding schemes and machine learning methods for splice sites recognition.
Gene, 705:113–126, Jul 2019.

Long Zhang, Guoxian Yu, Dawen Xia, and Jun Wang.
Protein-protein interactions prediction based on ensemble deep neural networks.
Neurocomputing, 324:10–19, 2019.

Ruiqing Zheng, Min Li, Xiang Chen, Fang-Xiang Wu, Yi Pan, and Jianxin Wang.
BiXGBoost: a scalable, flexible boosting-based method for reconstructing gene regulatory networks.
Bioinformatics, 35(11):1893–1900, Jun 2019.

Maria Colomé-Tatché and Fabian J Theis.
Statistical single cell multi-omics integration.
Current Opinion in Systems Biology, 7:54–59, 2018.

References

Yuming Ma, Yihui Liu, and Jinyong Cheng.
Protein secondary structure prediction based on data partition and semi-random subspace method.
Sci Rep, 8(1):9856, Jun 2018.

Xuan Zhang, Jun Wang, Jing Li, Wen Chen, and Changning Liu.
Crlncrc: a machine learning-based method for cancer-related long noncoding rna identification using integrated features.
BMC Med Genomics, 11(Suppl 6):120, Dec 2018.

Xiaoying Wang, Bin Yu, Anjun Ma, Cheng Chen, Bingqiang Liu, and Qin Ma.
Protein–protein interaction sites prediction by ensemble random forests with synthetic minority oversampling technique.
Bioinformatics, 35(14):2395–2402, 12 2018.

Zhen Cao, Xiaoyong Pan, Yang Yang, Yan Huang, and Hong-Bin Shen.
The lncLocator: a subcellular localization predictor for long non-coding rnas based on a stacked ensemble classifier.
Bioinformatics, 34(13):2185–2194, 07 2018.

References

Xing Chen, Chi-Chi Zhu, and Jun Yin.
Ensemble of decision tree reveals potential miRNA-disease associations.
PLoS Comput Biol, 15(7):e1007209, Jul 2019.

Jialin Yu, Shaoping Shi, Fang Zhang, Guodong Chen, and Man Cao.
PredGly: predicting lysine glycation sites for homo sapiens based on XGboost feature optimization.
Bioinformatics, 35(16):2749–2756, Aug 2019.

Hui Peng, Yi Zheng, Zhixun Zhao, Tao Liu, and Jinyan Li.
Recognition of CRISPR/Cas9 off-target sites through ensemble learning of uneven mismatch distributions.
Bioinformatics, 34(17):i757–i765, 09 2018.

Weijia Su, Xun Gu, and Thomas Peterson.
TIR-Learner, a new ensemble method for TIR transposable element annotation, provides evidence for abundant new transposable elements in the maize genome.
Mol Plant, 12(3):447–460, 03 2019.

References

Xiangxiang Zeng, Yue Zhong, Wei Lin, and Quan Zou.
Predicting disease-associated circular RNAs using deep forests combined with positive-unlabeled learning methods.
Brief Bioinform, Oct 2019.

Jaswinder Singh, Jack Hanson, Rhys Heffernan, Kuldip Paliwal, Yuedong Yang, and Yaoqi Zhou.
Detecting proline and non-proline cis isomers in protein structures from sequences using deep residual ensemble learning.
J Chem Inf Model, 58(9):2033–2042, 09 2018.

Anand Pratap Singh, Sarthak Mishra, and Suraiya Jabin.
Sequence based prediction of enhancer regions from DNA random walk.
Sci Rep, 8(1):15912, 10 2018.

Stephen Woloszynek, Zhengqiao Zhao, Jian Chen, and Gail L Rosen.
16S rRNA sequence embeddings: Meaningful numeric feature representations of nucleotide sequences that are convenient for downstream analyses.
PLoS Comput Biol, 15(2):e1006721, 02 2019.

References

John M Giorgi and Gary D Bader.
Transfer learning for biomedical named entity recognition with neural networks.
Bioinformatics, 34(23):4087–4094, Dec 2018.

Tongxin Wang, Travis S Johnson, Wei Shao, Zixiao Lu, Bryan R Helm, Jie Zhang, and Kun Huang.
BERMUDA: a novel deep transfer learning method for single-cell RNA sequencing batch correction reveals hidden high-resolution cellular subtypes.
Genome Biol, 20(1):165, 08 2019.

Sheng Wang, Zhen Li, Yizhou Yu, and Jinbo Xu.
Folding membrane proteins by deep transfer learning.
Cell Syst, 5(3):202–211.e3, 09 2017.

Žiga Avsec, Roman Kreuzhuber, Johnny Israeli, Nancy Xu, Jun Cheng, Avanti Shrikumar, Abhimanyu Banerjee, Daniel S Kim, Thorsten Beier, Lara Urban, Anshul Kundaje, Oliver Stegle, and Julien Gagneur.
The Kipoi repository accelerates community exchange and reuse of predictive models for genomics.
Nat Biotechnol, 37(6):592–600, Jun 2019.

Marcel **Turcotte**

Marcel.Turcotte@uOttawa.ca

School of Electrical Engineering and **Computer Science** (EECS)
University of Ottawa