Deep Learning

Machine Learning for Economics and Finance
Bachelor in Economics

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Marcel Weschke

27.05.2025

Roadmap

Learning Goals and Literature

(1) Learn about the basic architectures and mechanisms of neural networks
(2) Be able to train feedforward neural networks to solve supervised learning problems
(3) Understand why and how numerical optimization routines and regularization are important for fitting neural nets
(4) Learn how we can use Keras to fit neural networks in R

Book Chapter: 10

Further Readings:

- `https://www.deeplearningbook.org/` (Chapters 6-8)
- Guide to Deep Learning in R: `https://www.manning.com/books/deep-learning-with-r` (Introduction chapters are available online)

Deep Learning in R

- We will use TensorFlow—a powerful deep learning library developed by google
- The Keras library provides a user friendly interface to TensorFlow in R
- Both require a Python installation
- Installation guide for Keras and TensorFlow:
  https://hastie.su.domains/ISLR2/keras-instructions.html

What is Deep Learning?

- Deep learning is part of a broader family of machine learning methods based on artificial neural networks
- The adjective 'deep' refers to the use of multiple layers in the network to detect linear and non-linear features in the data
- Deep learning can be used for both, supervised and unsupervised learning problems (we focus on supervised learning)
- It nests many other machine learning techniques such as linear regressions, lasso or ridge regression as special cases

Why Deep Learning?

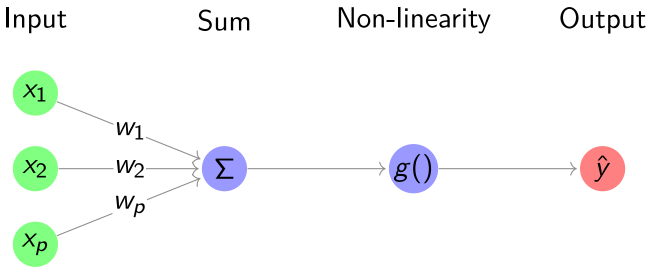
- First deep learning methods date back to the 1940s
- Usage of deep learning techniques has gone through the roof in the past years
- Why now?
  - Availability and storage capacities of big data
  - Increase in computing power
  - Advances in deep learning techniques for specific learning problems
  - New software makes application of deep learning very user friendly (TensorFlow, Keras,...)

Applications of Deep Learning?

Deep learning techniques have proven to be highly successful in various fields:

- **Image recognition and computer vision** (esp. convolutional neural networks)
  - Cancer detection
  - Self-driving cars
  - Face recognition
- **Speech recognition and language processing** (esp. recurrent neural networks)
  - Automatic translations
  - Text analysis
- **Finance**
  - Return predictions
  - Sentiment text analysis
  - Fraud detection

Key Building Block: The Perceptron



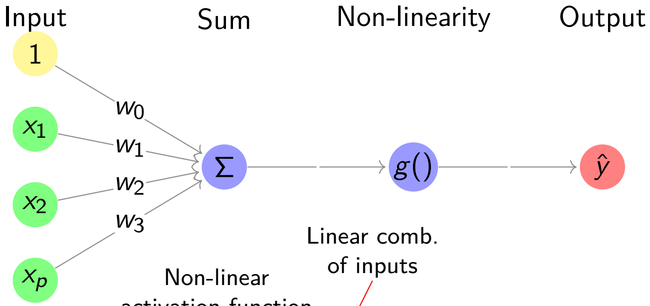
Formally:

$$\hat{y} = g\left(\sum_{j=1}^{p} w_j x_j\right)$$

Non-linear activation function

Linear comb. of inputs

Key Building Block: The Perceptron



Linear comb. of inputs

Non-linear activation function

Formally:

$$\hat{y} = g\left(w_0 + \sum_{j=1}^{p} w_j x_j\right) = g\left(w_0 + x^T w\right)$$

Bias

where $x = (x_1, x_2 \ldots x_p)^T, w = (w_1, w_2, \ldots w_p)^T$

The Activation Function

The activation function allows the network to learn non-linearities in the data

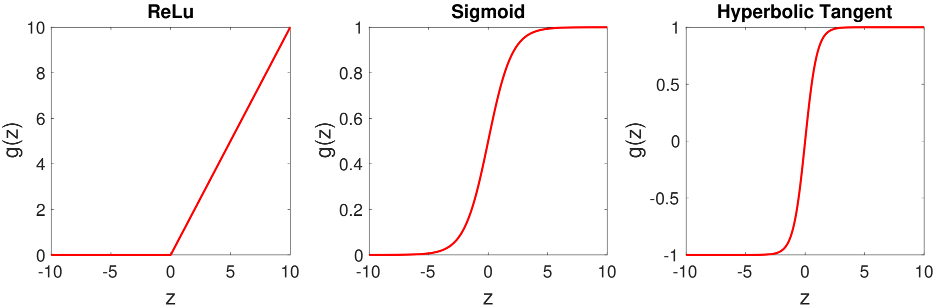
For $g(z) = z$, we are back in the linear regression case

Common choices for the activation function:

1. **Sigmoid**: $g(z) = \frac{1}{1+e^{-z}}$
2. **Hyperbolic tangent (tanh)**: $g(z) = \frac{e^{z}-e^{-z}}{e^{z}+e^{-z}}$
3. **Rectified Linear Unit (ReLu)**: $g(z) = \max(0, z)$

Choosing the *right* activation function is non-trivial and depends on the problem itself (more on that later)

The Activation Function



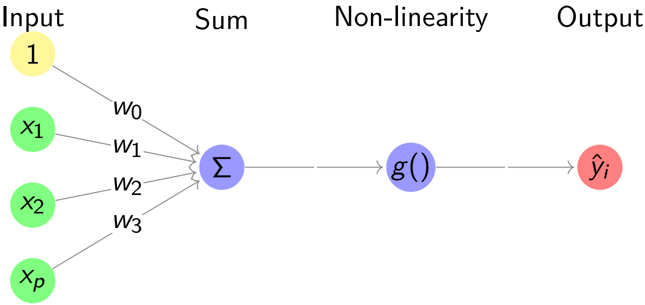
$$g(z) = \max(0, z)$$

$$g(z) = \frac{1}{1 + e^{-z}}$$

$$g(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$

```
activation = "relu" activation = "sigmoid" activation = "tanh"
```

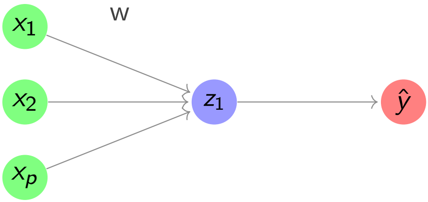
Perceptron: Simplified Representation



Formally:

$$\hat{y} = g\left(w_0 + x^T w\right)$$

Perceptron: Simplified Representation



Purple nodes combine two steps:

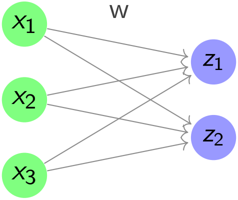
1. Compute neuron z:

$$z = \left(w_0 + x_i^T w\right)$$

2. Apply activation function $g()$:

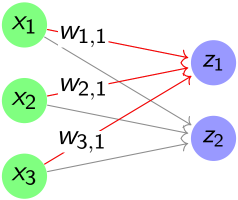
$$g(z)$$

Adding a Neuron to the Network



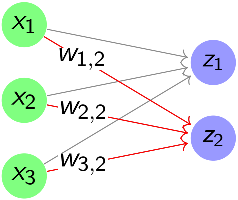
Now there are 6 links w between the three x_j and the two z_k

Adding a Neuron to the Network



$$z_1 = w_{0,1} + \sum_{j=1}^{p} w_{j,1} x_j = w_{0,1} + w_{1,1} x_1 + w_{2,1} x_2 + w_{3,1} x_3$$

Adding a Neuron to the Network



$$z_2 = w_{0,2} + \sum_{j=1}^{p} w_{j,2} x_j = w_{0,2} + w_{1,2} x_1 + w_{2,2} x_2 + w_{3,2} x_3$$

Total number of weights in w: $p \times d$ where d denotes the number of neurons z_k (plus d biases)

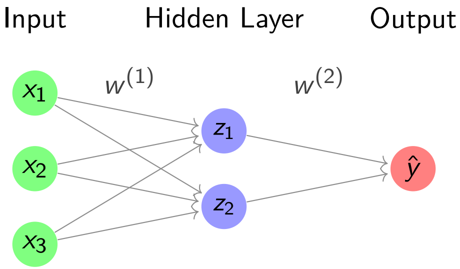
Adding a Neuron to the Network

Problem:

- Now we have two neurons z_1 and z_2.
- But how do we obtain a single output $\hat{y}$?

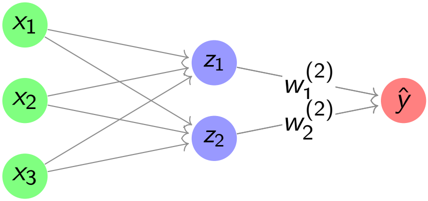
Solution: Use again a linear transformation (and an activation function $g()$)

Adding a Neuron to the Network



For each linear transformation, we obtain a vector of weights; $w^{(1)}$ and $w^{(2)}$ in this example

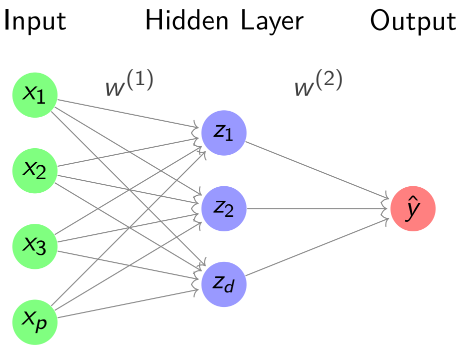
Compute Output $\hat{y}$ using Neurons z_k



1. Apply activation function to neurons: $g(z_k)$
2. Use linear transformation: $w_0^{(2)} + w_1^{(2)} g(z_1) + w_2^{(2)} g(z_2)$
3. Use activation function to compute final output:

$$\hat{y} = g(w_0^{(2)} + w_1^{(2)} g(z_1) + w_2^{(2)} g(z_2))$$

Single Layer Neural Network



$$z_k = w_{k,0}^{(1)} + \sum_{j=1}^{p} w_{k,j}^{(1)} x_j, \qquad \hat{y} = g(w_0^{(2)} + \sum_{j=1}^{d} w_j^{(2)} g(z_j))$$

Building a Single Layer NN in R using Keras

Step 1: Build network and add hidden layer

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

# Network architecture
model = Sequential([
  Dense(units=3, activation='relu', input_shape=(100,))
])
```

- units: number of neurons in the hidden layer
- input shape: number of features p

Q: How many weights does the layer of the neural network have?

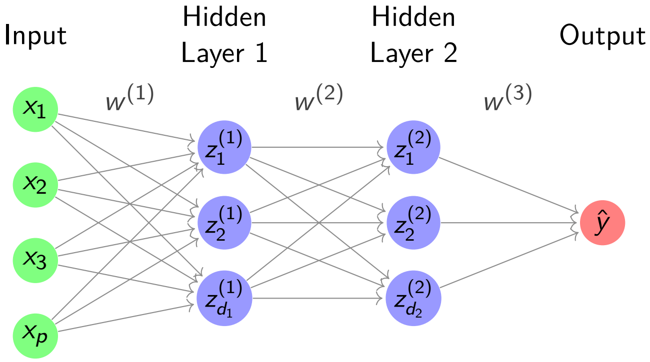
Building a Single Layer NN in R using Keras

Step 2: Going from hidden layer to output $\hat{y}$

```
# Network architecture
model = Sequential([
  Dense(units=3, activation='relu', input_shape=(100,)),
  Dense(units=1)
])
```

- `units` is set to 1 as we are predicting a single value $\hat{y}$
- A new layer uses the output size of the previous layer as the input size (we do not need to specify `input_shape`)
- As we do not specify an activation function, the output layer only computes the linear transformation (this is equivalent to setting $g(z) = z$)

Adding Another Hidden Layer



$$z_k^{(l)} = w_{k,0}^{(l)} + \sum_{j=1}^{d_{l-1}} w_{k,j}^{(l)} g(z_j^{(l-1)})$$

Note that the activation functions for each layer can differ but for the ease of notation we neglect the index here. Formally it must read $g^{(l)}()$

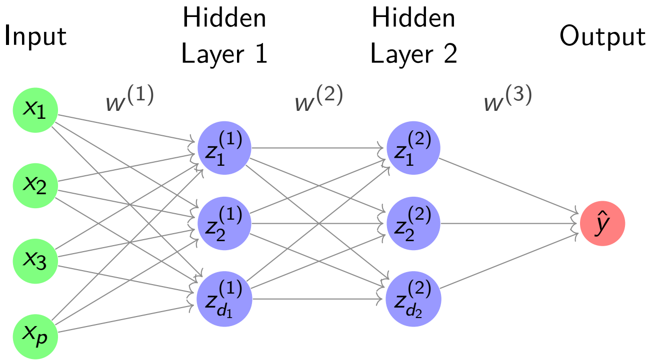
Adding another Hidden Layer in Python

```python
# Network architecture
model = Sequential([
  Dense(units=10, activation='relu', input_shape=(100,)),
  Dense(units=5,  activation='sigmoid'),
  Dense(units=1)
])
```

- First layer has 10 neurons and uses the relu activation function
- Second layer has 5 neurons and uses the sigmoid activation function
- Final output is a (continuous) scalar

Q: How many weights does the neural network have?

Dense Layers



- A layer where all neurons are connected which each other is called a *dense layer* (or fully connected layer)
- A dense layer allows for a lot of flexibility, but due to the many weights there is also the risk of overfitting

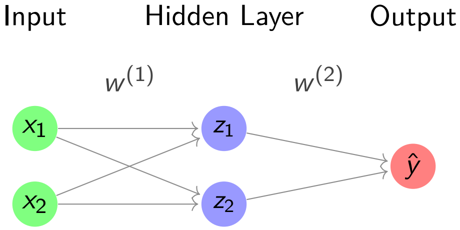
A Simple Example of a Feedforward NN

- Suppose we want to predict stock returns
- We start with a simple model with two features:
  - x_1: market return
  - x_2: size of the stock
  - y: return of the stock
- So we have data

$$\boldsymbol{X} = \begin{pmatrix} x_{11} & x_{12} \\ x_{21} & x_{12} \\ \vdots & \vdots \\ x_{n1} & x_{n2} \end{pmatrix}, \quad \boldsymbol{y} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix}$$

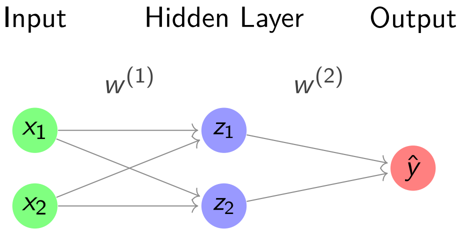
A Simple Example of a Feedforward NN

We use a feedforward neural network with one hidden layer with two neurons:



Suppose we have some initial guess for our weights $w^{(1)}$ and $w^{(2)}$ and we take a single data point $x_{i,1} = 0.05$, $x_{i,2} = 100$ and $y_i = 0.08$

A Simple Example of a Feedforward NN



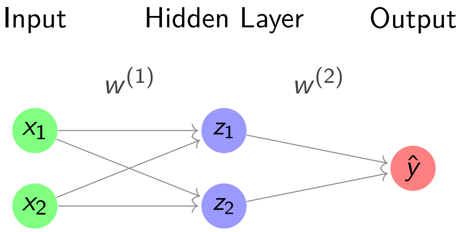
Step 1:

- Compute z_1 and z_2:

$$z_1 = w_{0,1}^{(1)} + w_{1,1}^{(1)} x_{i,1} + w_{2,1}^{(1)} x_{i,2} = w_{0,1}^{(1)} + w_{1,1}^{(1)} * 0.05 + w_{2,1}^{(1)} * 100$$
$$z_2 = w_{0,2}^{(1)} + w_{1,2}^{(1)} x_{i,1} + w_{2,2}^{(1)} x_{i,2} = w_{0,2}^{(1)} + w_{1,2}^{(1)} * 0.05 + w_{2,2}^{(1)} * 100$$

- Recall $w^{(1)}$ is known

A Simple Example of a Feedforward NN



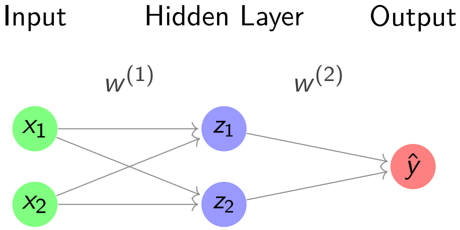
Step 2:

- Apply activation function to both neurons
- Here we choose to use the ReLu function:

$$
\begin{aligned}
g(z_1) &= \max(z_1, 0) \\
g(z_2) &= \max(z_2, 0)
\end{aligned}
$$

- Recall z_k is known from **Step 1**

A Simple Example of a Feedforward NN



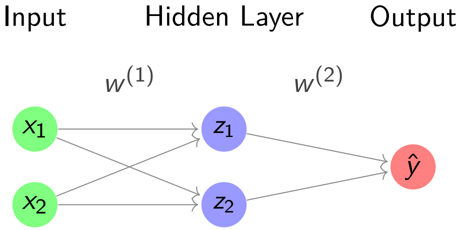
Step 3:

- Apply linear transformation to obtain $\hat{y}_i$

$$\hat{y}_i = w_0^{(2)} + w_1^{(2)} g(z_1) + w_2^{(2)} g(z_2)$$

- Note that we didn't use an activation function for $\hat{y}_i$ as y is a continuous variable
- This is equivalent to setting $g(z) = z$ for y

Python-Code: A Simple Example of a Feedforward NN




```python
# Network architecture
model = Sequential([
  Dense(units=2, activation='relu', input_shape=(2,)),
  Dense(units=1)
])
```

A Simple Example: Training the NN

- Now we can compare the prediction

$$\hat{y}_i = f(x_i, \boldsymbol{W})$$

to the realizations y_i

- We can do this by using some loss function

$$L_i(f(x_i, \boldsymbol{W}), y_i)$$

- For example we could use the squared prediction error:

$$L_i(f(x_i, \boldsymbol{W}), y_i) = (f(x_i, \boldsymbol{W}) - y_i)^2 = (y_i - \hat{y}_i)^2$$

A Simple Example: Training the NN

Step 4:

- We redo the exercise for all realization i to obtain the empirical loss function (sometimes also objective function or cost function)

$$
\begin{aligned}
J(\boldsymbol{W}) &= \frac{1}{n}\sum_{i=1}^{n} L_i(f(x_i, \boldsymbol{W}), y_i) \\
&= \frac{1}{n}\sum_{i=1}^{n}(f(x_i, \boldsymbol{W}) - y_i)^2
\end{aligned}
$$

Fitting the network: Find the weights that minimize $J(\boldsymbol{W})$

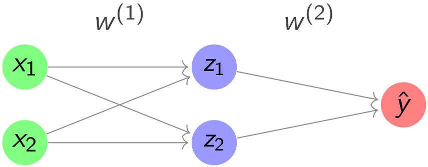
```
loss = "mse"
```

Example: Hitters Data

Let us fit our first neural network to predict the salary in the Hitters data

For this we use 09-Deep_learning-Hitters.R

Feedforward NN for Classification Problems

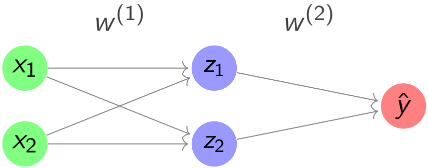


Suppose now that we want to predict whether returns are positive or negative (classification problem)

Hence, $y \in [0, 1]$ and $\hat{y}_i$ should represent a probability instead of a continuous outcome

Step 1 and **Step 2** remain exactly the same as before and all we need to adjust are **Step 3** and **Step 4**

Feedforward NN for Classification Problems



To ensure that $y \in [0, 1]$ we can use the sigmoid function as an activation function

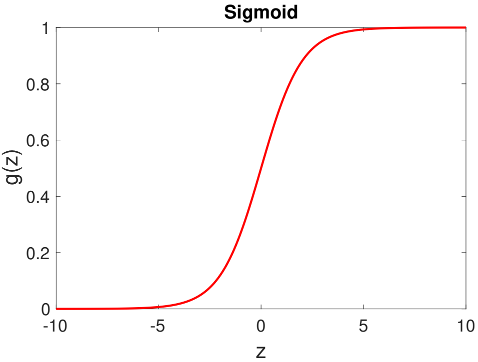
Step 3:

- Apply linear transformation and activation function to obtain $\hat{y}_i$

$$\hat{y}_i = g^{(2)}(w_0^{(2)} + w_1^{(2)} g^{(1)}(z_1) + w_2^{(2)} g^{(1)}(z_2))$$

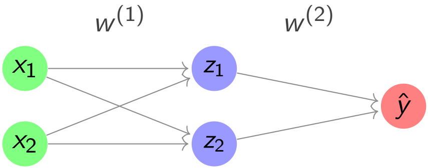
where $g^{(2)}(z) = \frac{1}{1+e^{-z}}$

Feedforward NN for Classification Problems



$$g(z) = \frac{1}{1 + e^{-z}}$$

Python-Code: Feedforward NN for Classification Problems




```
# Network architecture
model = Sequential([
  Dense(units=2, activation='relu', input_shape=(2,)),
  Dense(units=1, activation='sigmoid')
])
```

Feedforward NN for Classification Problems

Step 4:

- As a loss function for classification problems, we can use cross entropy loss:

$$J(\boldsymbol{W}) = \frac{1}{n}\sum_{i=1}^{n} L_i(f(x_i, \boldsymbol{W}), y_i)$$

$$= -\frac{1}{n}\sum_{i=1}^{n} \underbrace{y_i}_{\text{actual}} \log(\underbrace{f(x_i, \boldsymbol{W})}_{\text{predicted}}) + (1 - \underbrace{y_i}_{\text{actual}}) \log(1 - \underbrace{f(x_i, \boldsymbol{W})}_{\text{predicted}})$$

$$\underset{\text{f(x)}}{\begin{pmatrix} 0.8 \\ 0.1 \\ 0.9 \\ \vdots \end{pmatrix}} \begin{matrix} \color{red}{X} \\ \color{red}{X} \\ \checkmark \\ \end{matrix} \underset{y}{\begin{pmatrix} 0 \\ 1 \\ 1 \\ \vdots \end{pmatrix}}$$

```
1    loss = "binary_crossentropy"
```

Roadmap

Training neural networks: Loss minimization

Aim: We want to find the weights $\boldsymbol{W}$ that minimize our empirical loss function:

$$\begin{aligned} \boldsymbol{W}^* &= \arg\min_{\boldsymbol{W}} J(\boldsymbol{W}) \\ &= \arg\min_{\boldsymbol{W}} \frac{1}{n} \sum_{i=1}^{n} L_i(f(x_i, \boldsymbol{W}), y_i) \end{aligned}$$

where $\boldsymbol{W} = \{w^{(1)}, w^{(2)}, \ldots\}$ which can potentially be very large
$\Rightarrow$ As $J(\boldsymbol{W})$ is usually non-convex, finding $\boldsymbol{W}^*$ is non-trivial

Minimization of the Loss Function

- Loss function can be highly non-convex
- Finding global minimum can be difficult
- Fortunately, Tensorflow provides very good solver for this task:
  - Adam
  - Adadelta
  - RMSProp
  - Stochastic Gradien Descent (SGD)
  - ...

```
optimizer = "adam"
```

```
optimizer = "rmsprop"
```

```
optimizer = "adadelta"
```

```
optimizer = "sgd"
```

Training Neural Nets in Practice: Mini-Batches

- To find $\boldsymbol{W}^*$ we need to compute the gradient $\frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
- This can be very time consuming when the data set is large and the network is deep
- **Idea if Mini-Batches**: Only use a subset of the data to approximate the gradient
- The size of the mini-batch $b \ll n$ needs to be provided by the user

```
batch_size = 128
```

Epochs

- To train Neural Nets we need to define how long the solver is searching for a minimum
- For this we define the number of epochs (instead of the number of iterations)
- One epoch is a loop over the complete dataset (the solver 'sees' every datapoint exactly once)

```
epochs = 5
```

A Feedforward Neural Network in Python

```python
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.optimizers import SGD

# Build the neural network
model = Sequential([
  Dense(units=200, activation='relu', input_shape=(1000,)),
  Dense(units=50, activation='relu'),
  Dense(units=1)
])

# Compile the model (SGD optimizer and MSE loss)
model.compile(optimizer=SGD(), loss='mse')

# Train the model
history = model.fit(X_train, y_train, epochs=5, batch_size=128,
↪  validation_split=0.1, verbose=1)
```

Example: Hitters Data

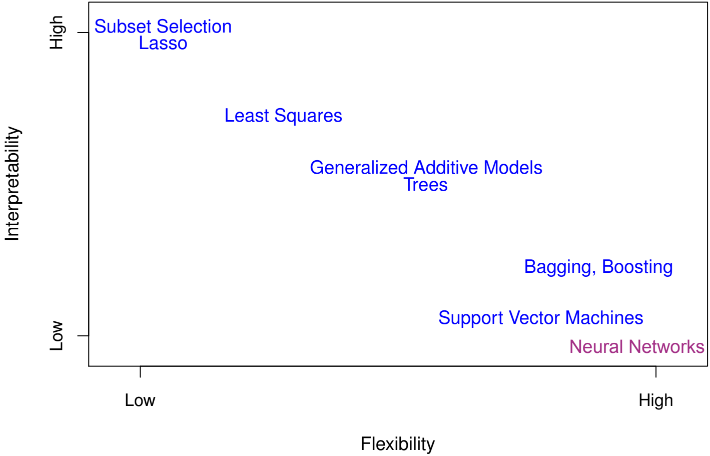
Let us again look in more detail at our first neural network to predict the salary in the Hitters data

File: 09-Deep_learning-Hitters.R

Overfitting and Regularization

- Neural networks are a **very flexible** method and hence can detect highly non-linear structures in the data
- But: this also poses the **risk of overfitting**
- So we need ways to handle the many parameters
- Note: best way to prevent overfitting is by adding more data

Flexibility vs Interpretability



Overfitting and Regularization

We will consider the following regularization approaches

- **Weight Regularization**
- **Dropout**
- **Early Stopping**

Weight Regularization

As in Lasso and Ridge regression, we can add a penalty term for large weights to the objective function

$$\boldsymbol{W}^* = \arg\min_{\boldsymbol{W}} J(\boldsymbol{W}) + \lambda\Omega(\boldsymbol{W})$$

- L^2 penalty (Ridge): $\Omega(\boldsymbol{W}) = \sum_j w_j^2$

```
from tensorflow.keras.regularizers import l2

Dense(units=16, activation='relu',
      kernel_regularizer=l2(0.01))  # L2 penalty λ = 0.01
```

Adds an L^2 penalty with $\lambda = 0.01$ to the weights of that layer

Weight Regularization

- L^1 penalty (Lasso): $\Omega(\boldsymbol{W}) = \sum_j |w_j|$

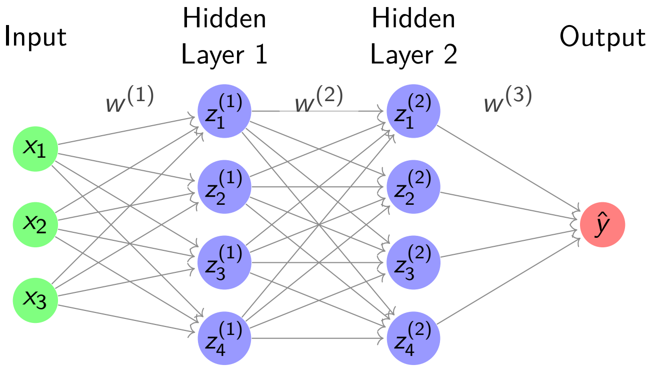
```
from tensorflow.keras.regularizers import l1

Dense(units=16, activation='relu',
      kernel_regularizer=l1(0.01))  # L1 penalty λ = 0.01
```

- Penalty for bias:

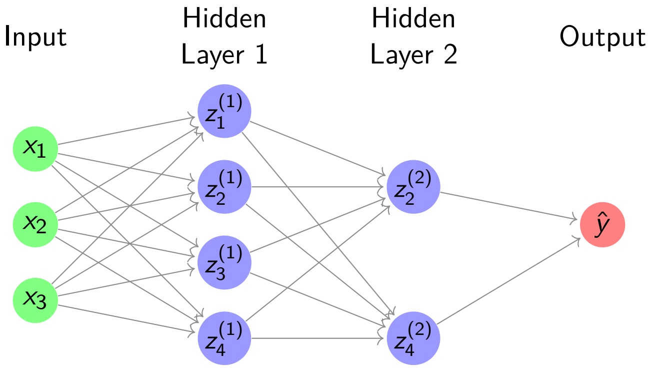
```
Dense(units=16, activation='relu',
      kernel_regularizer=l1(0.01),
      bias_regularizer=l1(0.01))    # L1 penalty on biases
```

Dropout



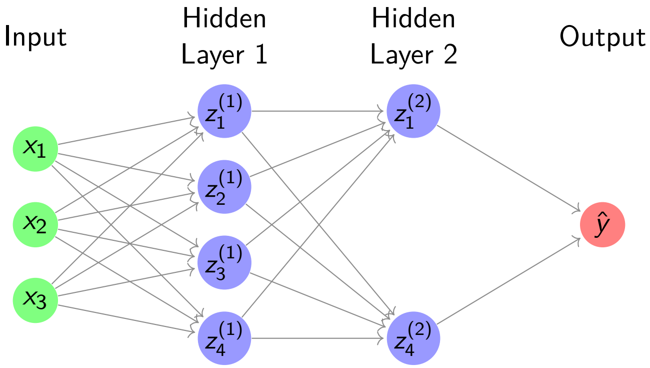
- Idea: randomly set some activations to zero during training

Dropout



- Dropout rate of 50%: in each iteration, randomly drop out 50% of the output features

Dropout



- Dropout rate of 50%: in each iteration, randomly drop out 50% of the output features

Dropout

- Significantly reduces the number of weights
- Forces network not to rely too heavily on specific nodes
- Typically dropout rate of 0.2 for input units and 0.5 for hidden nodes

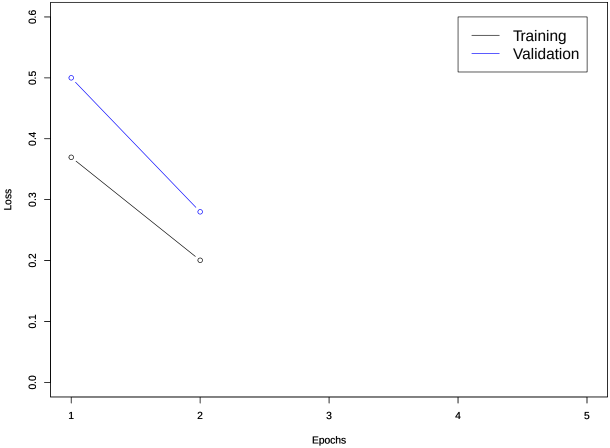
```
from tensorflow.keras.layers import Dense, Dropout

Dense(units=16, activation='relu'),
Dropout(rate=0.5) # Randomly sets 50% of the inputs to 0 during training.
```

Early Stopping

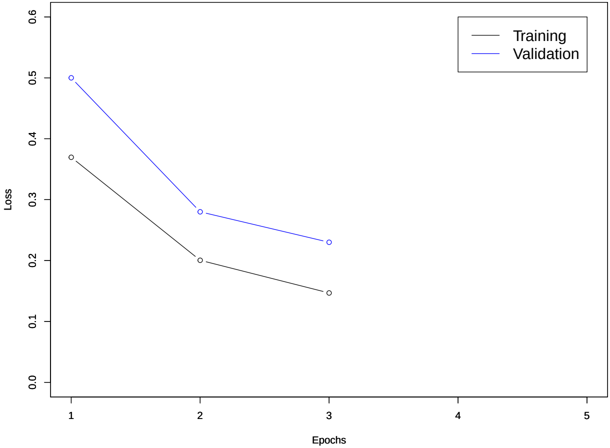
- Numerical optimization: Finding the (global) minimum of a function
- For deep learning, searching for the global minimum can lead to overfitting
- In practice, we are often not interested in finding the global minimum of $J(\boldsymbol{W})$ but it is sufficient to find a *small* $J(\boldsymbol{W})$
- Idea: Monitor the training and validation error and stop after a certain number of epochs (for example when the validation error is lowest)

Early Stopping



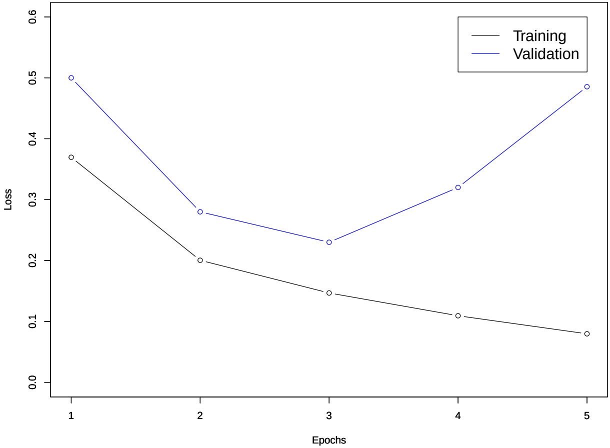
Idea: Stop solver before the optimizer overfits

Early Stopping



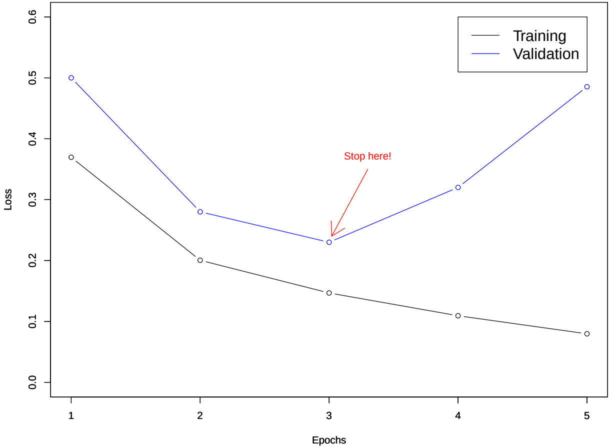
Idea: Stop solver before the optimizer overfits

Early Stopping



Idea: Stop solver before the optimizer overfits

Early Stopping



Idea: Stop solver before the optimizer overfits

Early Stopping

Stop training when valuation loss does not decrease any more:

```python
from tensorflow.keras.callbacks import EarlyStopping

# Early Stopping
es_callback = EarlyStopping(
  monitor='val_loss',        # Metric to monitor
  patience=0                 # Stop as soon as val_loss stops improving
)

# Train the model with early stopping
model.fit(
  X_train, y_train,
  validation_data=(X_val, y_val),  # Provide validation data
  epochs=20,
  batch_size=512,
  callbacks=[es_callback],
  verbose=1
)
```

Network Architecture in Practice

- Setting up neural networks and tuning the hyperparameters is a research field itself
- There are no general rules how to set up the networks
- However, with more practice you will learn what works and what doesn't
- Good starting point: set up a network with many layers that overfits the data; then try to put in more shape until the overfitting stops

Outlook

Convolutional Neural Networks

- Try to detect local features in the data
- Very successful for image and video recognition

Recurrent Neural Networks

- Keeps track of information in sequential or time series data
- Very successful for text and voice processing as well as time series predictions

Example: Hitters Data

Let us again look at the neural network to predict the salary in the Hitters data

File: 09-Deep_learning-Hitters.R

Example: Hitters Data

Task: Build a new network that predicts whether the salary is larger or smaller than the median salary in the training sample

Steps:

1. Define a new variable that is 1 if $y > median(y)$ and 0 otherwise (use ifelse() function)
2. Do this for training, validation and test data, always using the median of the training data
3. Build and fit a network that can handle classification problems (use sigmoid as the final layer activation function and binary_crossentropy for the loss function.)
4. Apply the regularization techniques we discussed in class and analyze how they affect the outcomes