Week 3 - Lesson 2: Training In Practice

Dr. Hongping Cai

Department of Computer Science

University of Bath

Topic 1:
Optimization
with Gradient Descent

Batch Gradient Descent

Update the weight

$$w_{ji}^{(l)} \leftarrow w_{ji}^{(l)} - \eta \frac{\partial J(W)}{\partial w_{ji}^{(l)}}$$

$$J(W) = \frac{1}{N} \boxed{\sum_{n=1}^{N}} L\left(f\left(x^{(n)}; W\right), y^{(n)}\right)$$

Compute the gradients on the entire training set. This is called **Batch Gradient Descent**. It is very computationally extensive.

Mini-Batch Gradient Descent

Update the weight

$$w_{ji}^{(l)} \leftarrow w_{ji}^{(l)} - \eta \frac{\partial J(W)}{\partial w_{ji}^{(l)}}$$

$$J(W) = \frac{1}{N} \boxed{\sum_{n=1}^{N}} L\left(f\left(x^{(n)}; W\right), y^{(n)}\right)$$

$$\Downarrow$$

Randomly <u>pick a batch</u> of B training samples, compute the gradients over the batches. This is called **Mini-Batch Gradient Descent**.

$$J_B(W) = \frac{1}{B} \boxed{\sum_{n=1}^{B}} L\left(f\left(x^{(n)}; W\right), y^{(n)}\right)$$

Mini-Batch Gradient Descent

$$\frac{\partial J_B(W)}{\partial w_{ji}^{(l)}} \approx \frac{\partial J(W)}{\partial w_{ji}^{(l)}}$$

- It is a good approximation.
- Much faster convergence
- Can parallelize computation, achieve significant speed increases on GPU.

Stochastic Gradient Descent (SGD)

Update the weight

$$w_{ji}^{(l)} \leftarrow w_{ji}^{(l)} - \eta \frac{\partial J(W)}{\partial w_{ji}^{(l)}}$$

$$J(W) = \frac{1}{N} \boxed{\sum_{n=1}^{N}} L(f(x^{(n)}; W), y^{(n)})$$

Randomly pick <u>only one training sample</u>. This is called **Stochastic Gradient Descent (SGD)**. Easy to compute but very noisy (stochastic).

$$J_n(W) = L(f(x^{(n)}; W), y^{(n)})$$

Three Gradient Descent Variants

- Batch Gradient Descent
- Mini-Batch Gradient Descent
- Stochastic Gradient Descent (SGD)

Is typically the choice.

Note that people often use term “SGD” refers to the Mini-batch Gradient Descent.

```
model.compile(loss='categorical_crossentropy',
              optimizer= ‘SGD',
              metrics=['accuracy'])
history = model.fit(trainX, trainy, epochs=150, batch_size=64, validation_split=0.2)
```

One **epoch**: one learning cycle through the entire training data.

Reference for Topic 1

- Video lecture by Alexander Amini: MIT course on deep learning, https://www.youtube.com/watch?v=njKP3FqW3Sk
- https://cs231n.github.io/optimization-1/
- https://ruder.io/optimizing-gradient-descent/

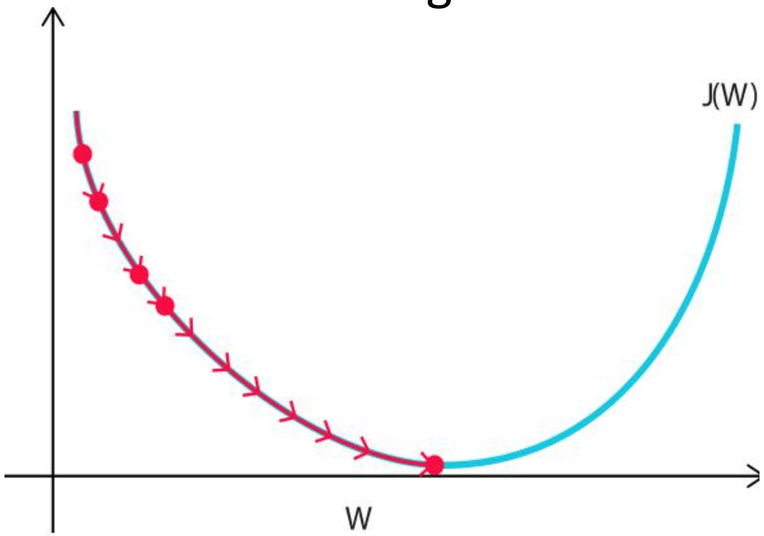
Topic 2:
Learning Rate

Learning Rate

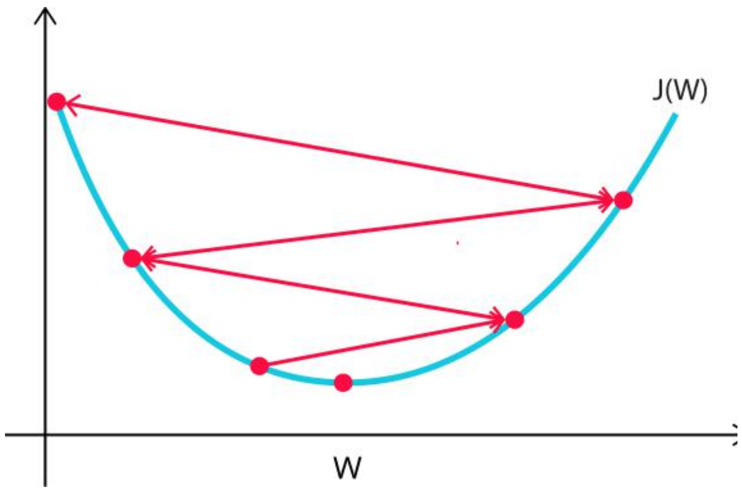
Update the weight

$$w_{ji}^{(l)} \leftarrow w_{ji}^{(l)} - \boxed{\eta} \frac{\partial J(W)}{\partial w_{ji}^{(l)}}$$

Too small: requires many updates before reaching the minimum.



Too large: overshoot and even diverge.

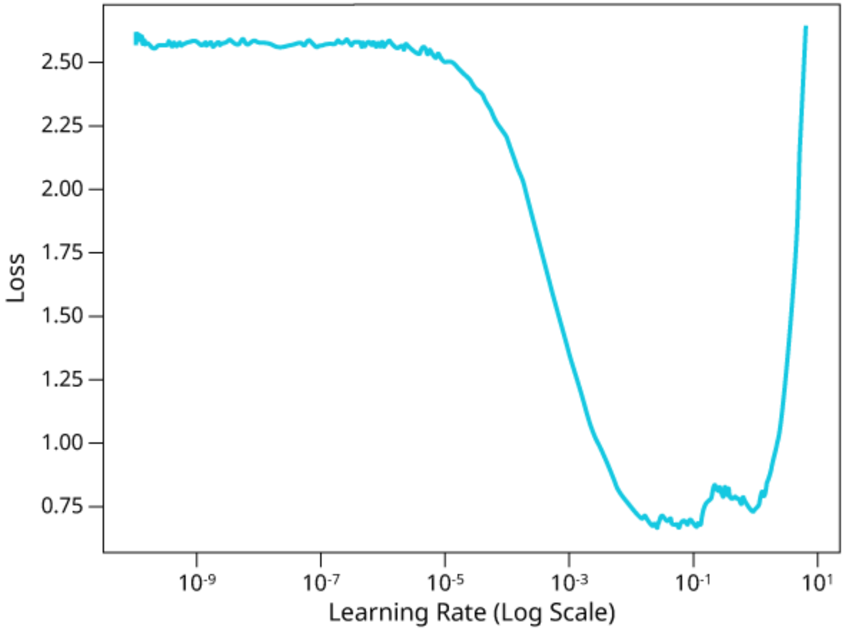


Learning Rate

Update the weight

$$w_{ji}^{(l)} \leftarrow w_{ji}^{(l)} - \boxed{\eta} \frac{\partial J(W)}{\partial w_{ji}^{(l)}}$$

Q: How to find the proper learning rate?



<u>Option 1</u>: Try different learning rate

$$\eta = 10, 1, 10^{-1}, 10^{-2}, 10^{-3}, \ldots$$

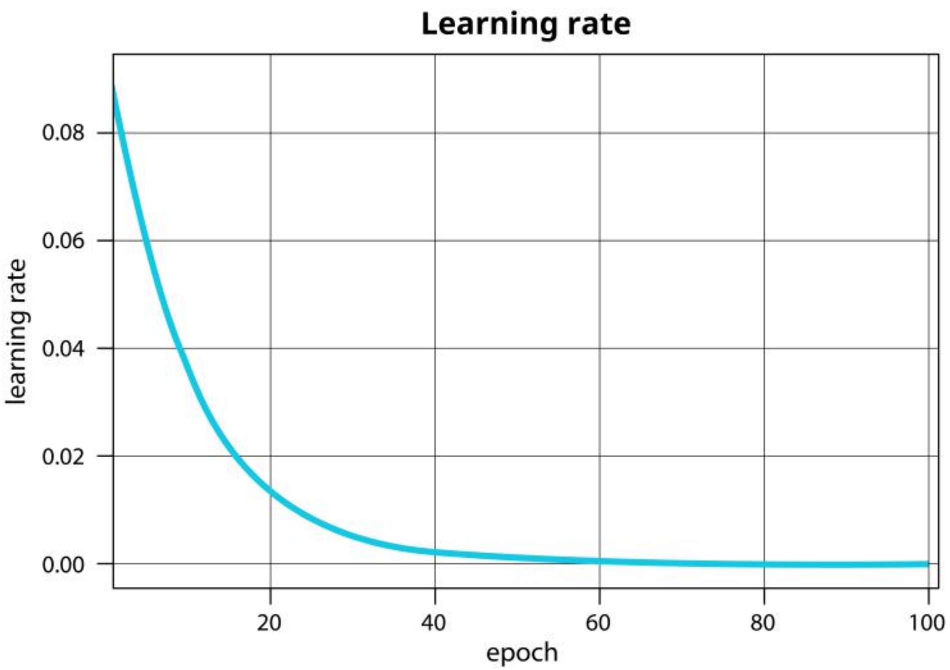
- Train the model for a few hundred iterations with each learning rate. Then plot the loss varied with learning rate.

Learning Rate

Update the weight

$$w_{ji}^{(l)} \leftarrow w_{ji}^{(l)} - \eta \frac{\partial J(W)}{\partial w_{ji}^{(l)}}$$

Q: How to find the proper learning rate?



Option 2: Learning rate schedule

- Decrease the learning rate during training according to a pre-defined schedule.
  - Time-based decay
  - Step decay
  - Exponential decay

$$\eta = \eta_0 \cdot e^{-kt}$$

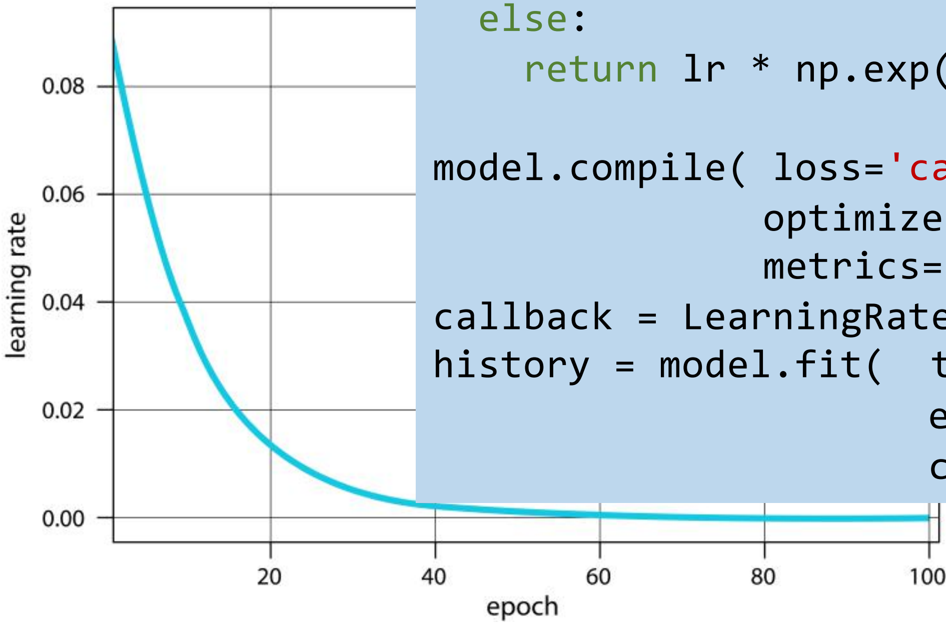
k is usually set to 0.1
t is the iteration number

Learning Rate

```
from keras.callbacks import LearningRateScheduler
import numpy as np

def scheduler(epoch, lr): # define a scheduler
  if epoch < 10:
    return lr
  else:
    return lr * np.exp(-0.1*epoch)

model.compile( loss='categorical_crossentropy',
               optimizer= 'SGD',
               metrics=['accuracy'])
callback = LearningRateScheduler(scheduler)
history = model.fit(  trainX, trainy,
                      epochs=150, batch_size=32,
                      callbacks=[callback])
```

› find the proper
ate?

schedule
rate during training
ned schedule.



k is usually set to 0.1
t is the iteration number

Learning Rate

Update the weight

$$w_{ji}^{(l)} \leftarrow w_{ji}^{(l)} - \boxed{\eta} \frac{\partial J(W)}{\partial w_{ji}^{(l)}}$$

Q: How to find the proper learning rate?

Option 3: Adaptive learning rate

- AdaGrad optimizer
- Adadelta optimizer
- RMSProp optimizer
- Adam optimizer
- …

Usually a good choice

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

OR:

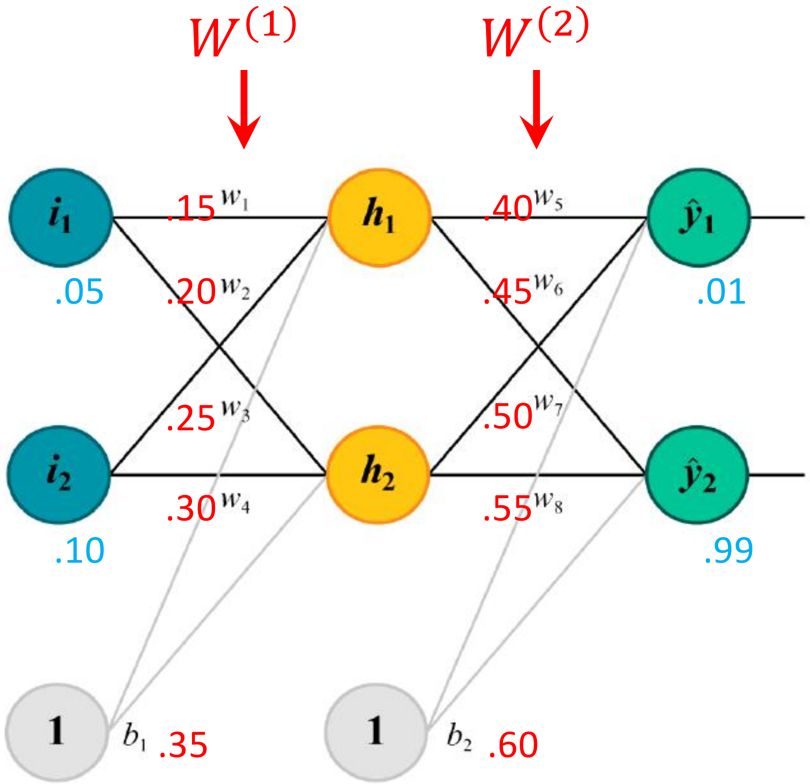
```
opt = keras.optimizers.RMSprop(lr=0.001, rho=0.9)
model.compile( loss 'categorical_crossentropy',
               optimizer=opt,
               metrics=['accuracy'])
```

Reference for Topic 2

- Book: Aurelien Geron. Hands-On Machine Learning with Scikit-Learn and TensorFlow. O'Reilly. 2019.
- https://www.allaboutcircuits.com/technical-articles/understanding-learning-rate-in-neural-networks/
- https://towardsdatascience.com/learning-rate-schedules-and-adaptive-learning-rate-methods-for-deep-learning-2c8f433990d1

Topic 3:
Vanishing Gradient Problem

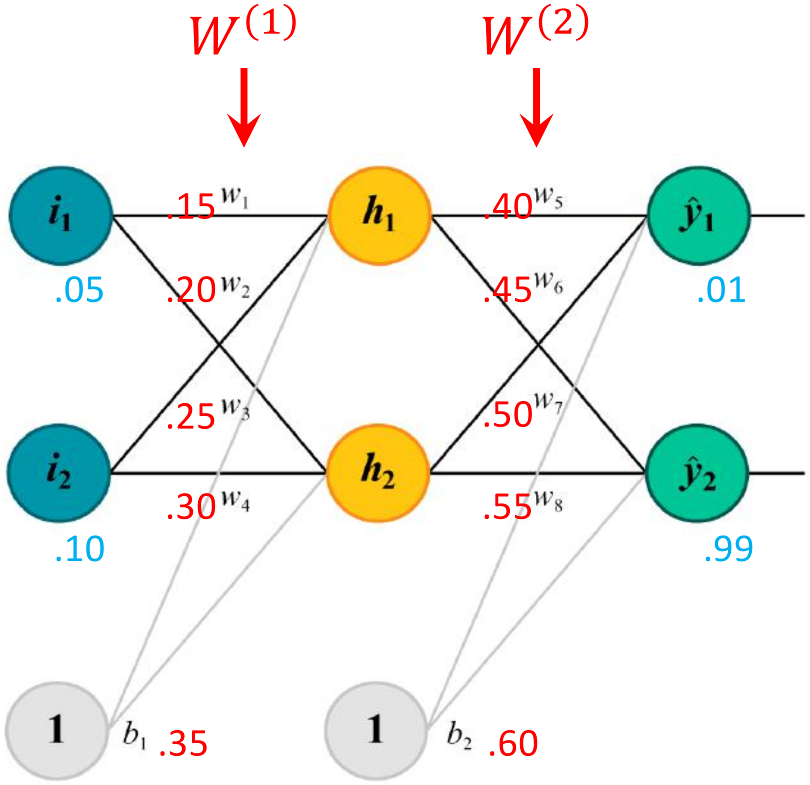
Vanishing gradient problem



	Iteration 0	Iteration 1	Iteration 2
$\frac{\partial J(W)}{\partial W^{(2)}}$	dJ_dw5 = +0.082167,	dJ_dw5 = +0.083706,	dJ_dw5 = +0.084819,
	dJ_dw6 = +0.082668,	dJ_dw6 = +0.084219,	dJ_dw6 = +0.085340,
	dJ_dw7 = −0.022603,	dJ_dw7 = −0.023732,	dJ_dw7 = −0.024896,
	dJ_dw8 = −0.022740,	dJ_dw8 = −0.023877,	dJ_dw8 = −0.025049,
$\frac{\partial J(W)}{\partial W^{(1)}}$	dJ_dw1 = +0.000439,	dJ_dw1 = +0.000366,	dJ_dw1 = +0.000285,
	dJ_dw2 = +0.000877,	dJ_dw2 = +0.000733,	dJ_dw2 = +0.000570,
	dJ_dw3 = +0.000498,	dJ_dw3 = +0.000426,	dJ_dw3 = +0.000345,
	dJ_dw4 = +0.000995,	dJ_dw4 = +0.000852,	dJ_dw4 = +0.000689,

Q: The gradients on the lower layer are much smaller than those on the higher layer. What effect?

Vanishing gradient problem



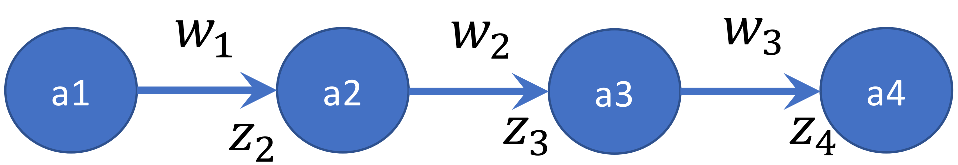
- This is called **vanishing gradient** problem: gradients get smaller and smaller as the backpropagation progresses down to the lower layers.

$$w_{ji}^{(l)} \leftarrow w_{ji}^{(l)} - \eta \underbrace{\boxed{\frac{\partial J(W)}{\partial w_{ji}^{(l)}}}}_{}\quad \overset{}{\approx \mathbf{0}}$$

- The lower layers' weights are updated very little. Therefore, the lower layers contribute very little to reduce the total loss.

Vanishing gradient problem

- Why?



$$a_k = g(z_k)$$

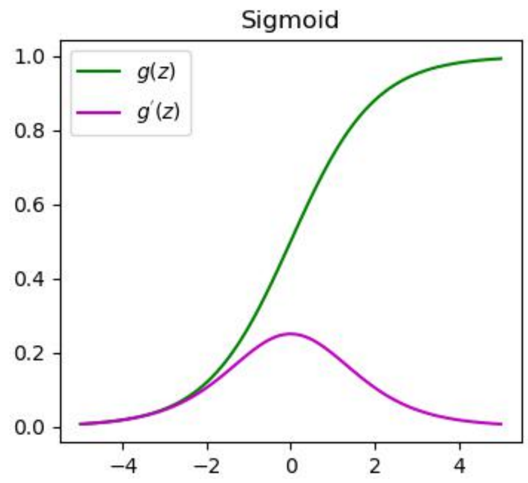
$$z_k = a_{k-1} * w_{k-1}$$

$$\frac{\partial J(W)}{\partial w_3} = \frac{\partial J(W)}{\partial a_4} \cdot \frac{\partial a_4}{\partial z_4} \cdot \frac{\partial z_4}{\partial w_3} = \frac{\partial J(W)}{\partial a_4} \cdot g'(z_4) \cdot a_3$$

$$\frac{\partial J(W)}{\partial w_2} = \frac{\partial J(W)}{\partial a_4} \cdot \frac{\partial a_4}{\partial z_4} \cdot \frac{\partial z_4}{\partial a_3} \cdot \frac{\partial a_3}{\partial z_3} \cdot \frac{\partial z_3}{\partial w_2} = \frac{\partial J(W)}{\partial a_4} \cdot g'(z_4) \cdot \boxed{w_3 \cdot g'(z_3)} \cdot a_2$$

$$\frac{\partial J(W)}{\partial w_1} = \frac{\partial J(W)}{\partial a_4} \cdot \frac{\partial a_4}{\partial z_4} \cdot \frac{\partial z_4}{\partial a_3} \cdot \frac{\partial a_3}{\partial z_3} \cdot \frac{\partial z_3}{\partial a_2} \cdot \frac{\partial a_2}{\partial z_2} \cdot \frac{\partial z_2}{\partial w_1} = \frac{\partial J(W)}{\partial a_4} \cdot g'(z_4) \cdot w_3 \cdot g'(z_3) \cdot \boxed{w_2 \cdot g'(z_2)} \cdot a_1$$

Vanishing gradient problem



$g'(z_k) \leq 0.25$

- Why?

Standard weight initialization approach is using Gaussian distribution $\mu = 0, \sigma = 1$. Therefore, $|w_k| \leq 1$ (mostly)

$$\frac{\partial J(W)}{\partial w_3} = \frac{\partial J(W)}{\partial a_4} \cdot \frac{\partial a_4}{\partial z_4} \cdot \frac{\partial z_4}{\partial w_3} = \frac{\partial J(W)}{\partial a_4} \cdot g'(z_4) \cdot a_3$$

$$\frac{\partial J(W)}{\partial w_2} = \frac{\partial J(W)}{\partial a_4} \cdot \frac{\partial a_4}{\partial z_4} \cdot \frac{\partial z_4}{\partial a_3} \cdot \frac{\partial a_3}{\partial z_3} \cdot \frac{\partial z_3}{\partial w_2} = \frac{\partial J(W)}{\partial a_4} \cdot g'(z_4) \cdot \overset{\leq \mathbf{0.25}}{\boxed{w_3 \cdot g'(z_3)}} \cdot a_2$$

$$\frac{\partial J(W)}{\partial w_1} = \frac{\partial J(W)}{\partial a_4} \cdot \frac{\partial a_4}{\partial z_4} \cdot \frac{\partial z_4}{\partial a_3} \cdot \frac{\partial a_3}{\partial z_3} \cdot \frac{\partial z_3}{\partial a_2} \cdot \frac{\partial a_2}{\partial z_2} \cdot \frac{\partial z_2}{\partial w_1} = \frac{\partial J(W)}{\partial a_4} \cdot \underbrace{g'(z_4)}_{\leq \mathbf{0.25}} \cdot \underbrace{w_3 \cdot g'(z_3)}_{\leq \mathbf{0.25}} \cdot \underbrace{\overset{\leq \mathbf{0.25}}{\boxed{w_2 \cdot g'(z_2)}}}_{\leq \mathbf{0.25}} \cdot a_1$$

Vanishing gradient problem

- How to avoid it?

Activation function

ReLU, instead of sigmoid or tanh

Weight initialization

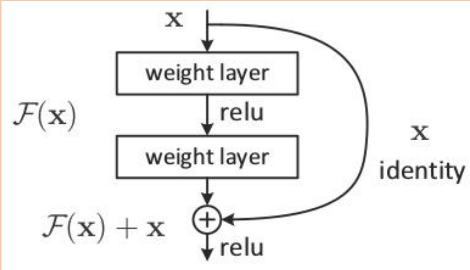
Glorot initialization (or Xavier initialization)

Layer restriction

Batch normalization

Network structure

Residual networks



Reference for Topic 3

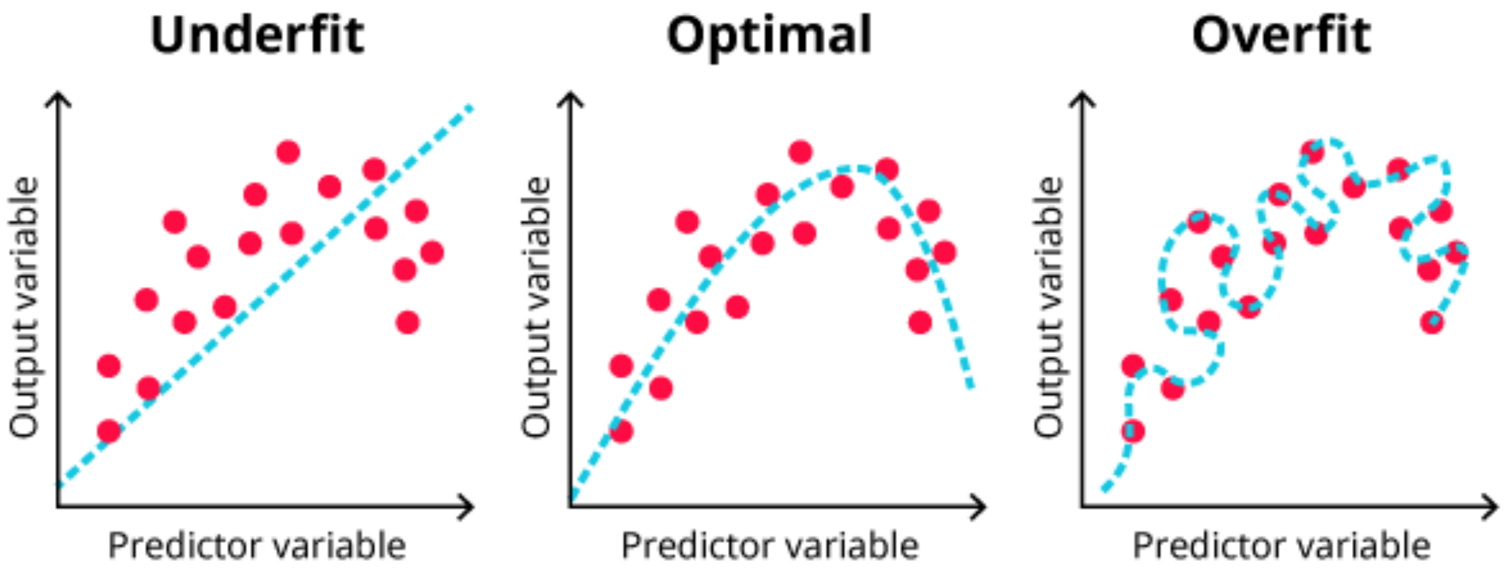
- Book: Aurelien Geron. Hands-On Machine Learning with Scikit-Learn and TensorFlow. O'Reilly. 2019.
- http://neuralnetworksanddeeplearning.com/chap5.html
- https://towardsdatascience.com/the-vanishing-gradient-problem-69bf08b15484

Topic 4a:
Overfitting Problem-1

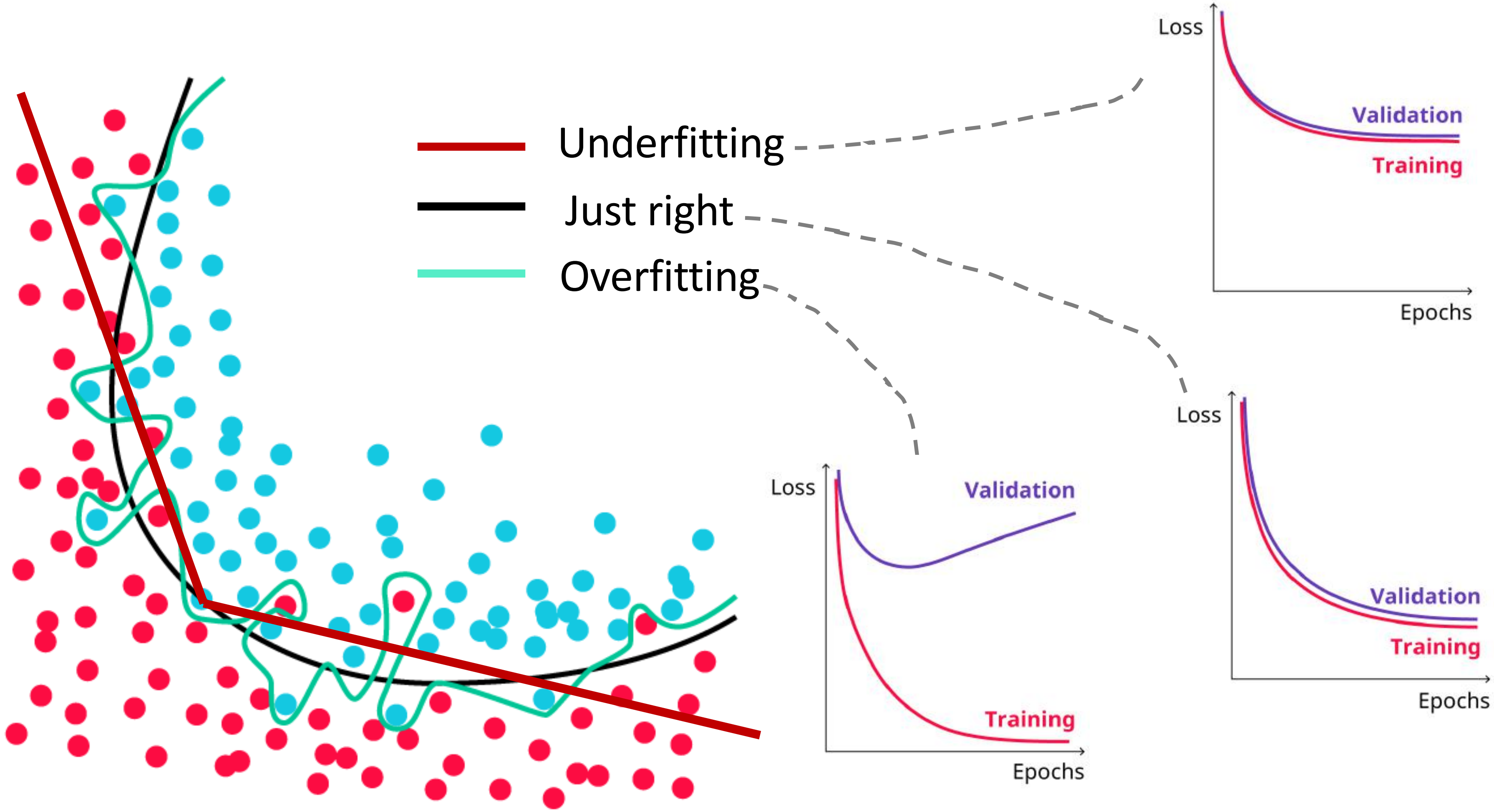
Underfitting and Overfitting

- Too simple
- Does not fully learn the data

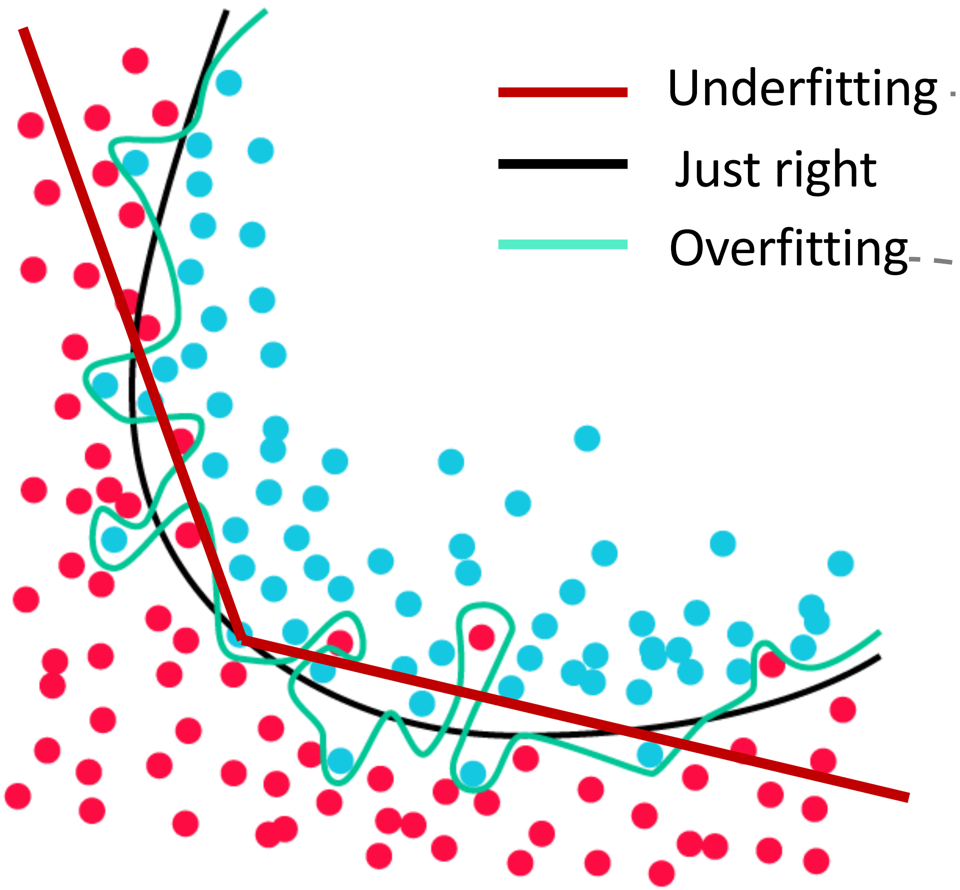
- Too complex
- Fits the noise in the training data
- Does not generalize well



How to Identify Underfitting and Overfitting



How to Prevent Underfitting and Overfitting

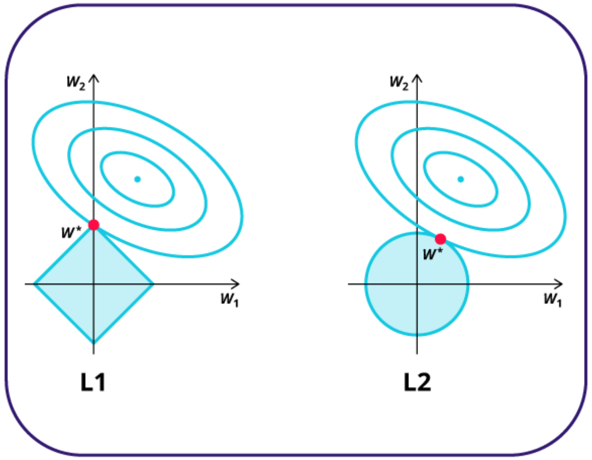


- Complexify model
- Add more nodes/layers
- Train longer

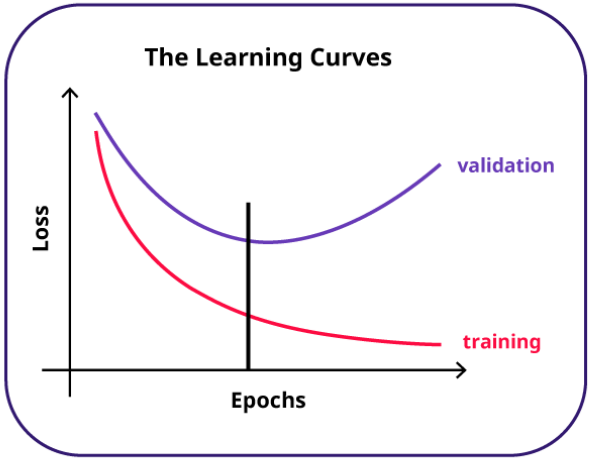
- More training data
  - Data augmentation
- Simplify model
- Regularization

Regularization

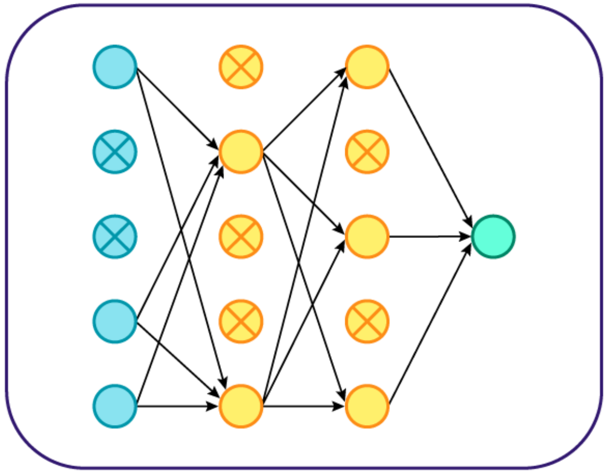
- **Regularization** is a technique which constraints the optimization problem to discourage complex model.
- Regularization helps the model generalize better on unseen data.



Weight regularization



Early stopping

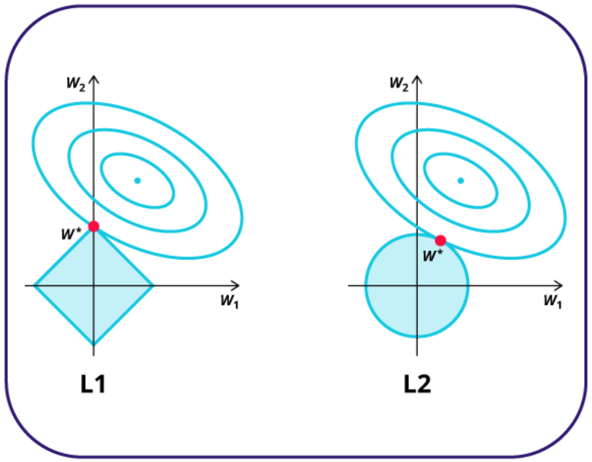


Dropout

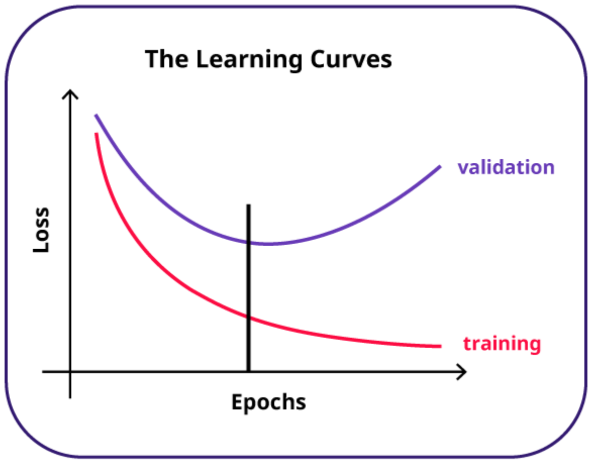
Topic 4b:
Overfitting Problem-2

Regularization

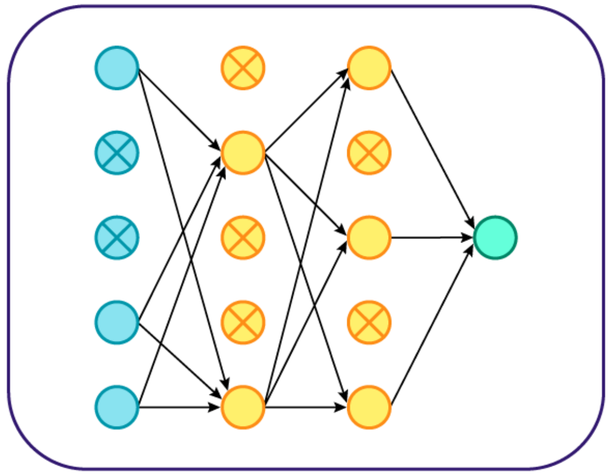
- Regularization helps the model generalize better on unseen data.



Weight regularization



Early stopping



Dropout

Weight Regularization

$$J(W) = \frac{1}{N}\sum_{n=1}^{N} L\left(f\left(x^{(n)}; W\right), y^{(n)}\right)$$

L2 Regularization: $J(W) = \frac{1}{N}\sum_{n=1}^{N} L\left(f\left(x^{(n)}; W\right), y^{(n)}\right) + \boxed{\lambda \sum_{k} w_k^2}$

L1 Regularization: $J(W) = \frac{1}{N}\sum_{n=1}^{N} L\left(f\left(x^{(n)}; W\right), y^{(n)}\right) + \boxed{\lambda \sum_{k} |w_k|}$

Weight Regularization

$\lambda = 0.01$

```
from keras.regularizers import l2

model = Sequential()
model.add(Dense(128, kernel_regularizer=l2(0.01), input_dim=8, activation='relu'))
model.add(Dense(64, kernel_regularizer=l2(0.01), activation='relu'))
model.add(Dense(8, kernel_regularizer=l2(0.01), activation='relu'))
model.add(Dense(1, activation='sigmoid'))
```

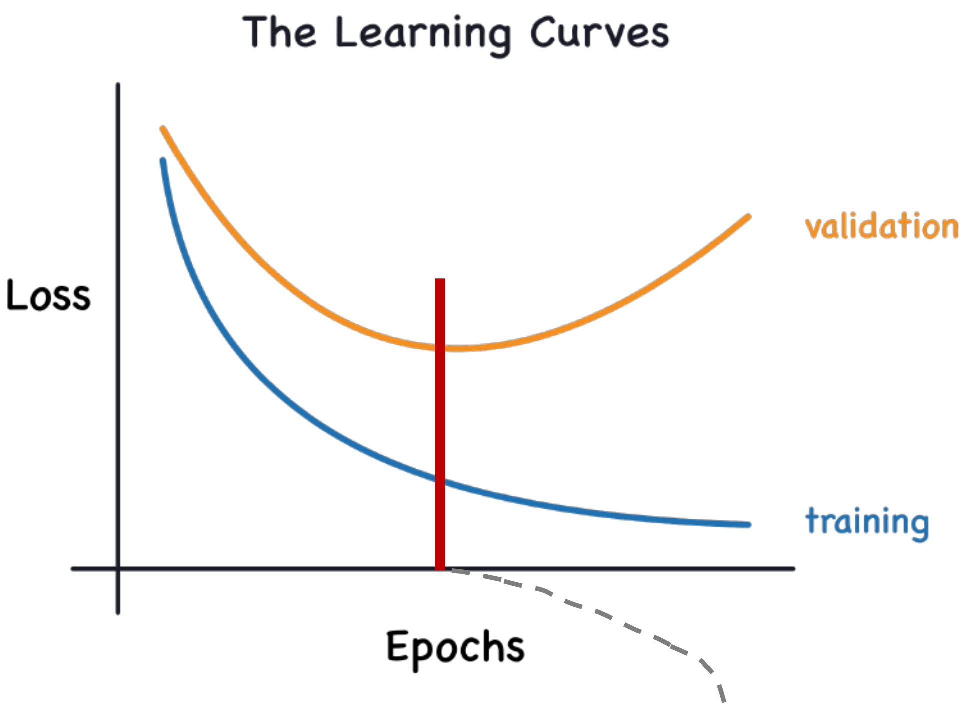
L2 Regularization:

- Is able to learn complex data patterns
- Is more commonly used
- Is not robust to outliers

L1 Regularization:

- Generates sparse models
- Is robust to outliers

Early Stopping



Stop training before it is likely to overfit

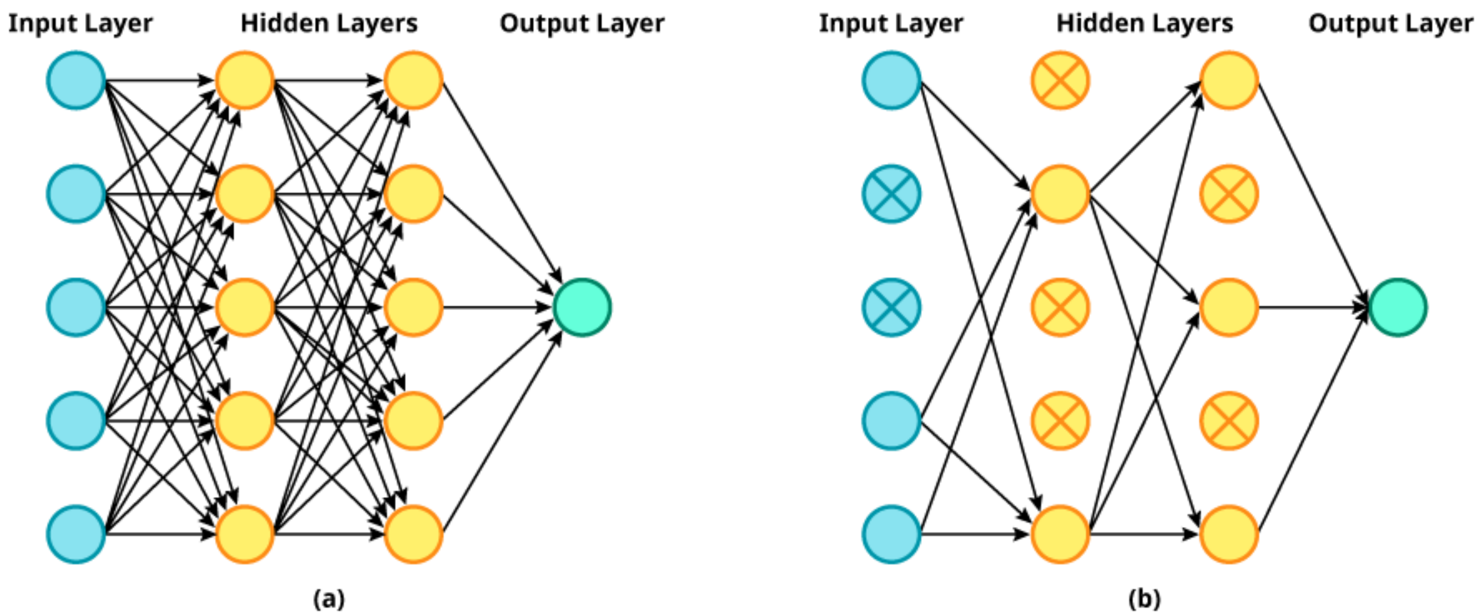
Early Stopping

If after 5 epochs there is no reduce of validation loss (with a tolerance of 0.001), the training will be stopped, the best weight for the lowest loss is kept.

```
from keras.callbacks import EarlyStopping

es_callback = EarlyStopping( monitor='val_loss',
                             min_delta = 0.001,
                             patience=5,
                             restore_best_weights=True)
model.fit(trainX, trainy, callbacks=[es_callback], epochs=1000, validation_split=0.3)
```

Dropout

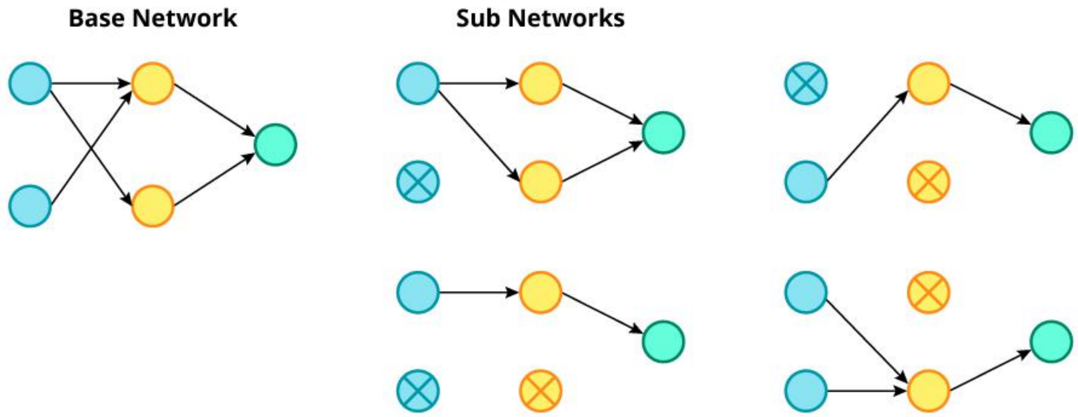
- **Dropout** is to randomly remove some hidden neurons along with their connections during training.



Q: Is it equivalent to using less nodes in each layer?

Dropout

A: NO. Because the removed nodes are different in each training iteration.



- Dropout is a kind of **ensemble of sub-networks** with shared parameters.
- Force the network not to rely on any particular connections of neurons.

Dropout

Randomly drop out 40% of the 128 nodes

```
from keras.layers import Dropout

model = Sequential()
model.add(Dense(128, input_dim=8, activation='relu'))
model.add(Dropout(0.4))
model.add(Dense(64, activation='relu'))
model.add(Dropout(0.4))
model.add(Dense(8, activation='relu'))
model.add(Dropout(0.4))
model.add(Dense(1, activation='sigmoid'))
```

In practice, you can usually apply dropout after all the dense layers excluding the output layer.

Reference for Topic 4

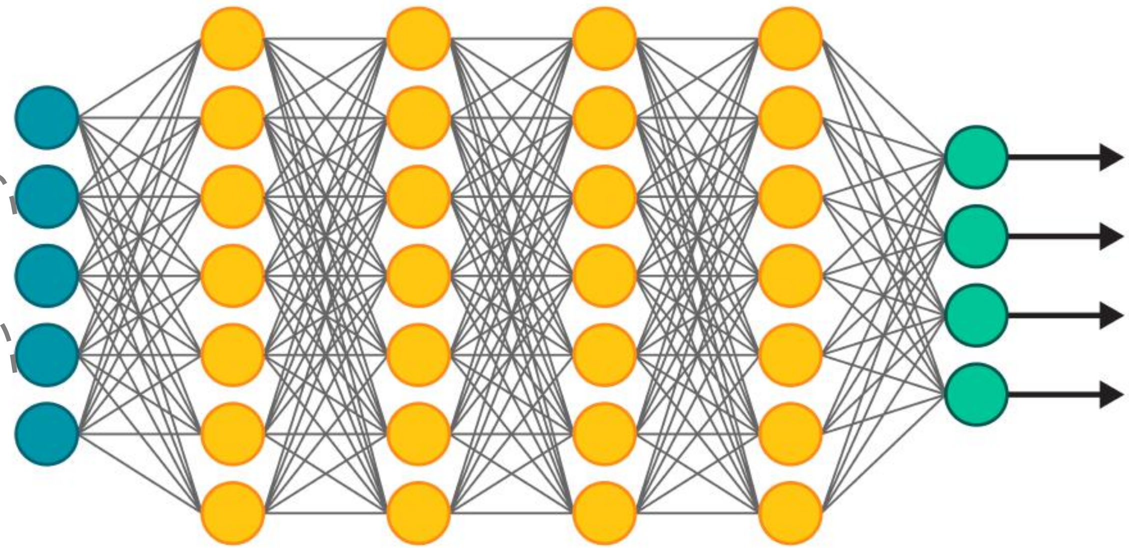
- Video lecture by Alexander Amini: MIT course on deep learning, https://www.youtube.com/watch?v=njKP3FqW3Sk
- https://www.kdnuggets.com/2019/12/5-techniques-prevent-overfitting-neural-networks.html
- https://medium.com/@jennifer.arty/regularization-methods-to-prevent-overfitting-in-neural-networks-1a79b5e3081f
- https://www.kaggle.com/ryanholbrook/dropout-and-batch-normalization

Topic 5:
Batch Normalization

Batch Normalization

x2 (house area): 78.5, 200.2, 12, 380.4, 60, …

x9 (No. of bedroom): 1, 3, 2, 7, 5, 2, …



Q: What should we do to train the network?

A: Standardize the input data: mean 0 and 1 std.

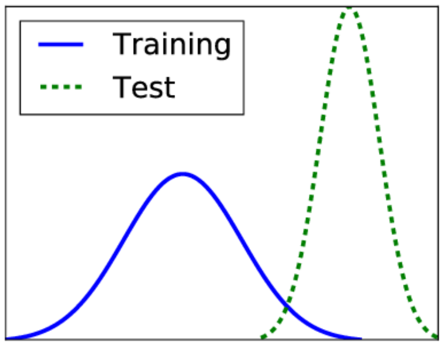
Q: Actually, all hidden layers have the same problem. How to solve it?

A: **Batch Normalization**.

Intuition for Batch Normalization

- Limit the **internal covariate shift**, allow more stable distribution of input for the internal layers.

Covariate Shift



Batch Normalization

- **Batch normalization** is a technique that standardizes the inputs to a layer for each mini-batch, then rescale and offsets them.

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1\ldots m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Before or After Activation Function?

*“The goal of Batch Normalization is to achieve a stable distribution of activation values throughout training, and in our experiments we apply it **before** the nonlinearity .” -- [S. Loffe, 2015]*

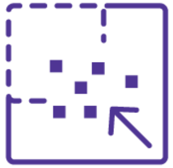
```
from keras.layers import BatchNormalization,Activation

model = Sequential()
model.add(Dense(128, input_dim=8))
model.add(BatchNormalization())
model.add(Activation('relu'))
model.add(Dense(64))
model.add(BatchNormalization())
model.add(Activation('relu'))
model.add(Dense(8))
model.add(BatchNormalization())
model.add(Activation('relu'))
model.add(Dense(1, activation='sigmoid'))
```

However, some others observed better performance with batch normalization **after** the activations.

Benefits

The networks are much less sensitive to the **weight initialization**.

Larger learning rates could be used, significantly speeding up the learning process



The **vanishing gradients** problem is strong reduced.

Act like a **regularizer**, reducing the need for other regularizations(such as dropout)

Reference for Topic 5

- Book: Aurelien Geron. Hands-On Machine Learning with Scikit-Learn and TensorFlow. O'Reilly. 2019.
- https://towardsdatascience.com/batch-normalization-in-neural-networks-1ac91516821c
- https://mlexplained.com/2018/01/10/an-intuitive-explanation-of-why-batch-normalization-really-works-normalization-in-deep-learning-part-1/
- https://paperswithcode.com/method/batch-normalization