Week 04:
Training in Practice

Machine Learning 2

Dr. Hongping Cai

Topic 1:
Optimization
with Gradient Descent

Batch Gradient Descent

Update the weight

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

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

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

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

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

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

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

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

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

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

Randomly pick only one training sample. 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/

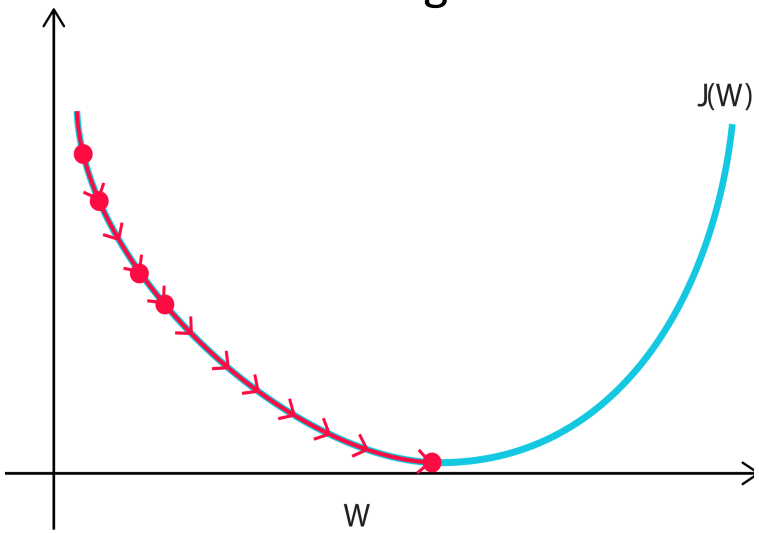
Topic 2:
Learning Rate

Learning Rate

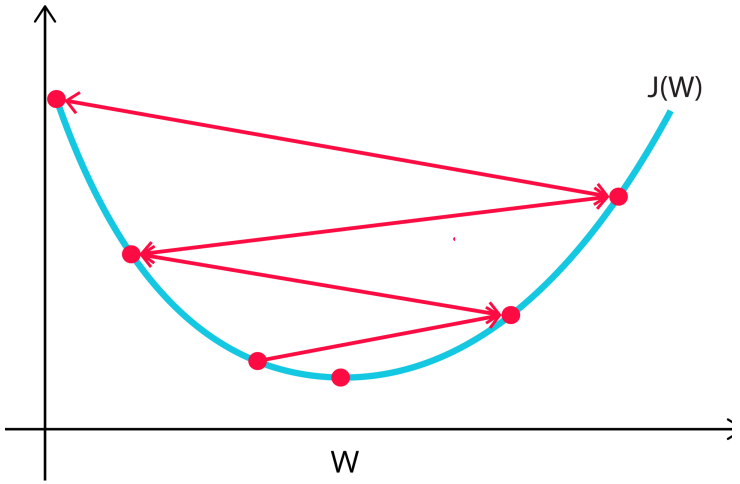
Update the weight

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

Too small: requires many updates before reaching the minimum.



Too large: overshoot and even diverge.

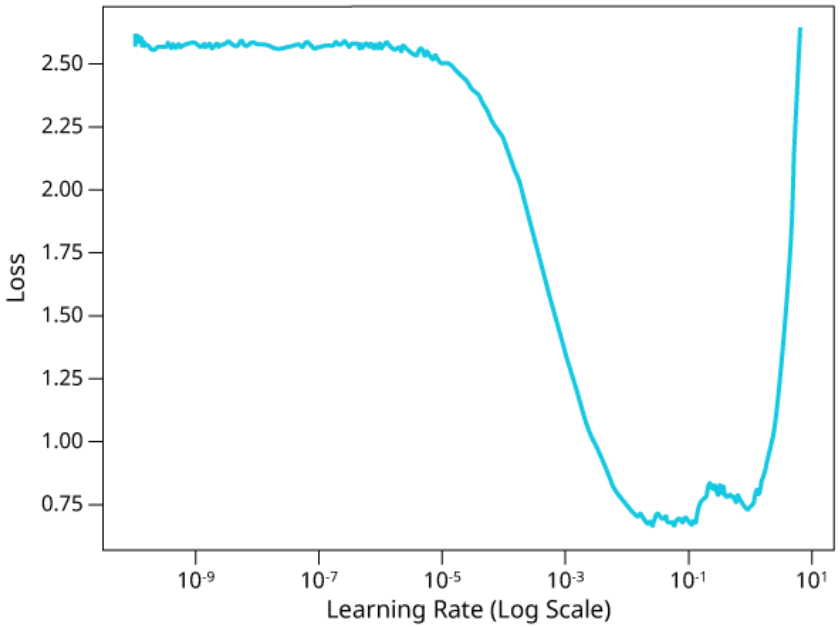


Learning Rate

Update the weight

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

Q: How to find the proper learning rate?



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

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

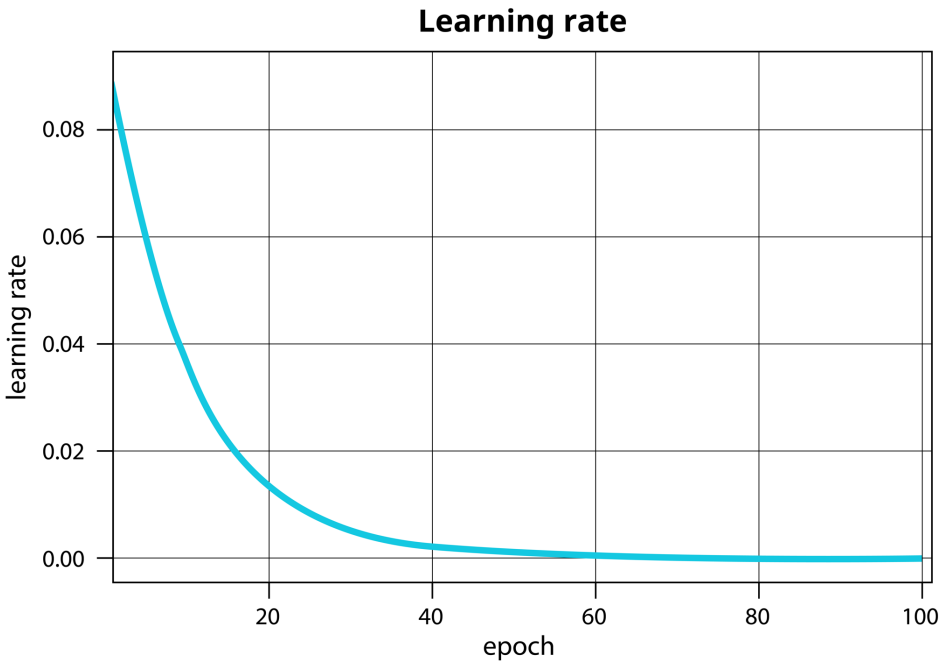
- Train the model for a few hundred iterations with each learning rate. Then plot the loss varied with learning rate.

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

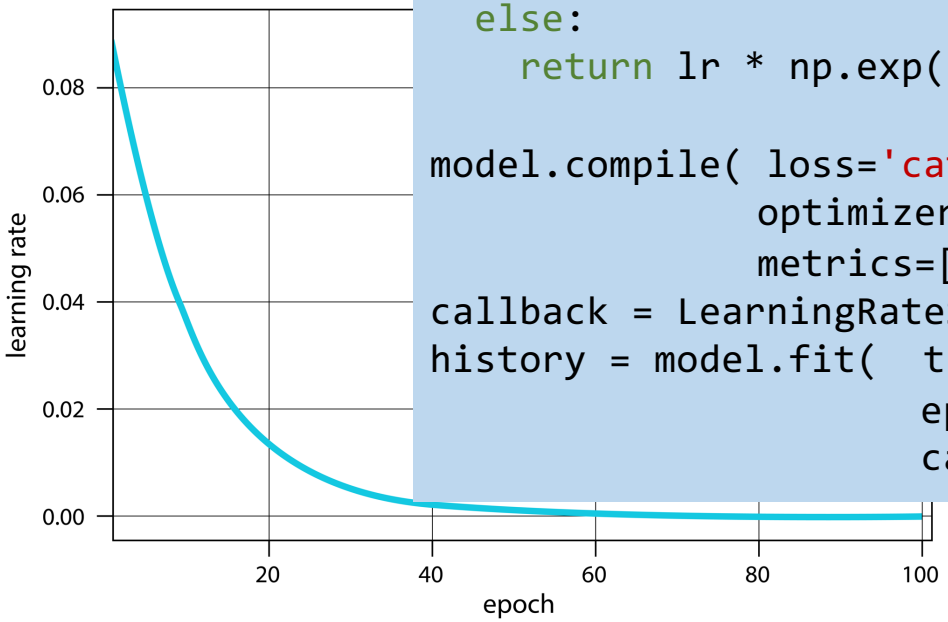
- Decrease the learning rate during training according to a pre-defined schedule.
  - Time-based decay
  - Step decay
  - Exponential decay

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

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])
```



o 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

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

Q: How to find the proper learning rate?

```
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'])
```

Option 3: Adaptive learning rate

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

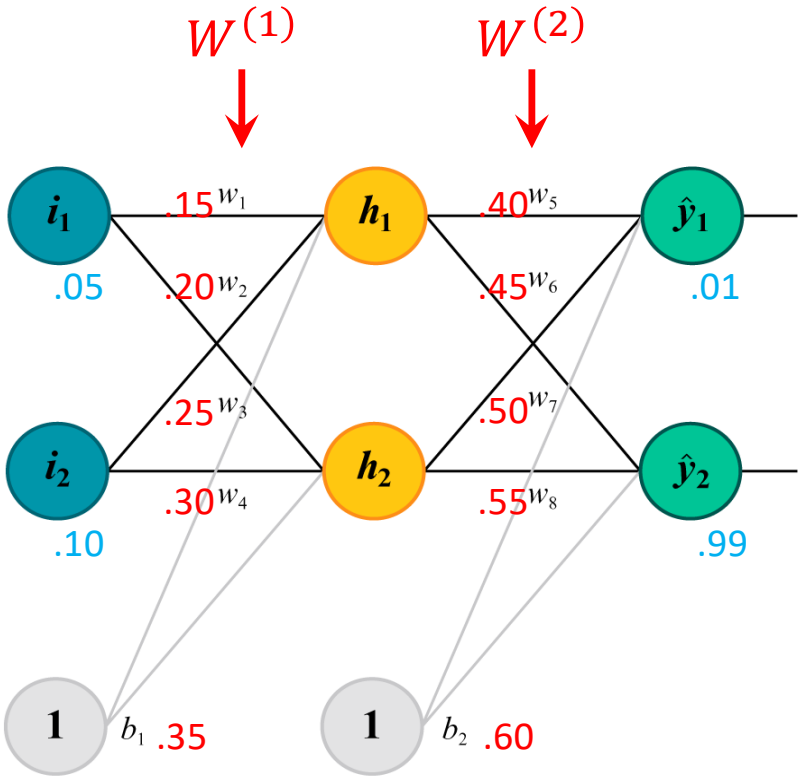
Usually a good choice

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

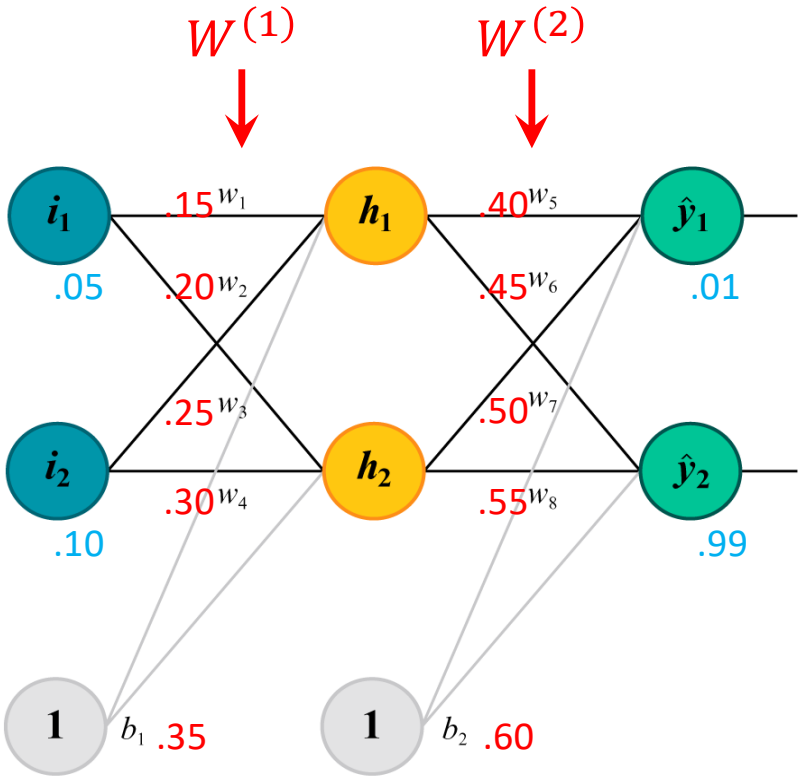
Vanishing gradient problem



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

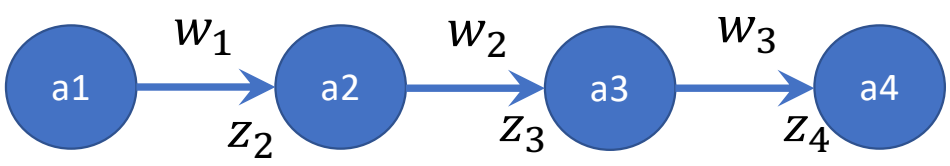
$$\approx 0$$

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

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

Vanishing gradient problem

- Why?



$$a_k = g(z_k)$$

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

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

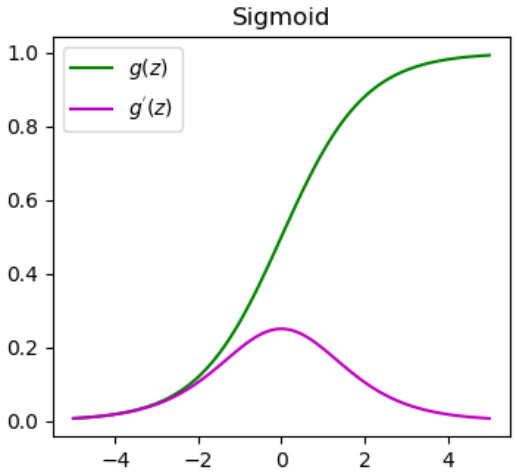
$$\frac{\partial J(W)}{\partial w_2} = \frac{\partial J(W)}{\partial a_4} \cdot \frac{\partial a_4}{\partial z_4} \cdot \frac{\partial z_4}{\partial a_3} \cdot \frac{\partial a_3}{\partial z_3} \cdot \frac{\partial z_3}{\partial w_2} = \frac{\partial J(W)}{\partial a_4} \cdot g'(z_4) \cdot w_3 \cdot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

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

Vanishing gradient problem

- Why?

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



$$g'(z_k) \leq 0.25$$

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

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

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

Vanishing gradient problem

- How to avoid it?

Activation function	Weight initialization
ReLU, instead of sigmoid or tanh	Glorot initialization (or Xavier initialization)
Layer restriction	**Network structure**
Batch normalization	Residual networks (ResNet, DenseNet, etc)

Deep Residual Learning (ResNet)

- Q: Will stacking more layers always give better performance?
- A: Not always.

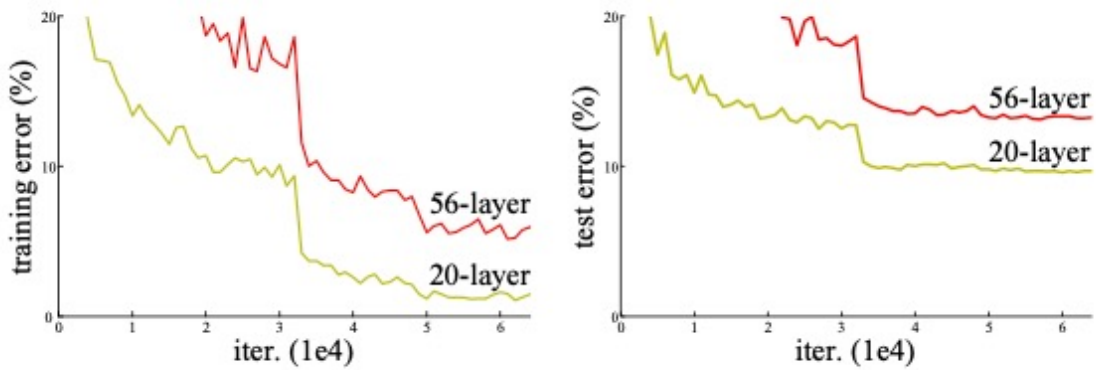
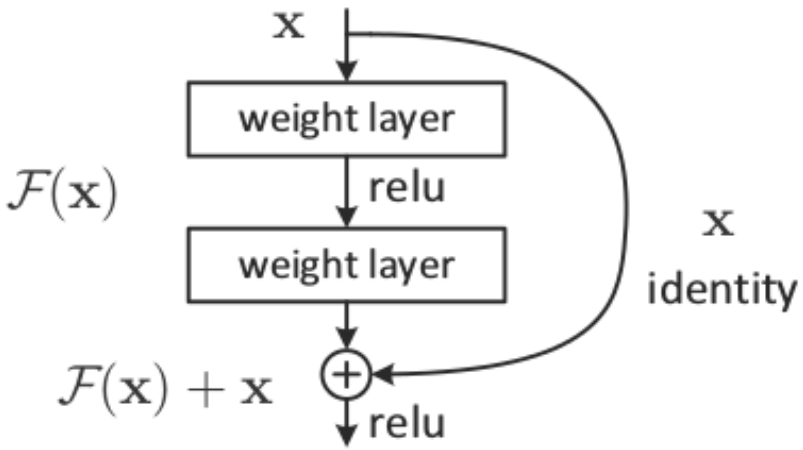


Figure 1. Training error (left) and test error (right) on CIFAR-10 with 20-layer and 56-layer "plain" networks. The deeper network has higher training error, and thus test error. Similar phenomena on ImageNet is presented in Fig. 4.

ResNet[1]

- Winner in the ILSVRC 2015 classification competition (3.57% top-5 error) with 152 layers
- Main technique: **skip connection**



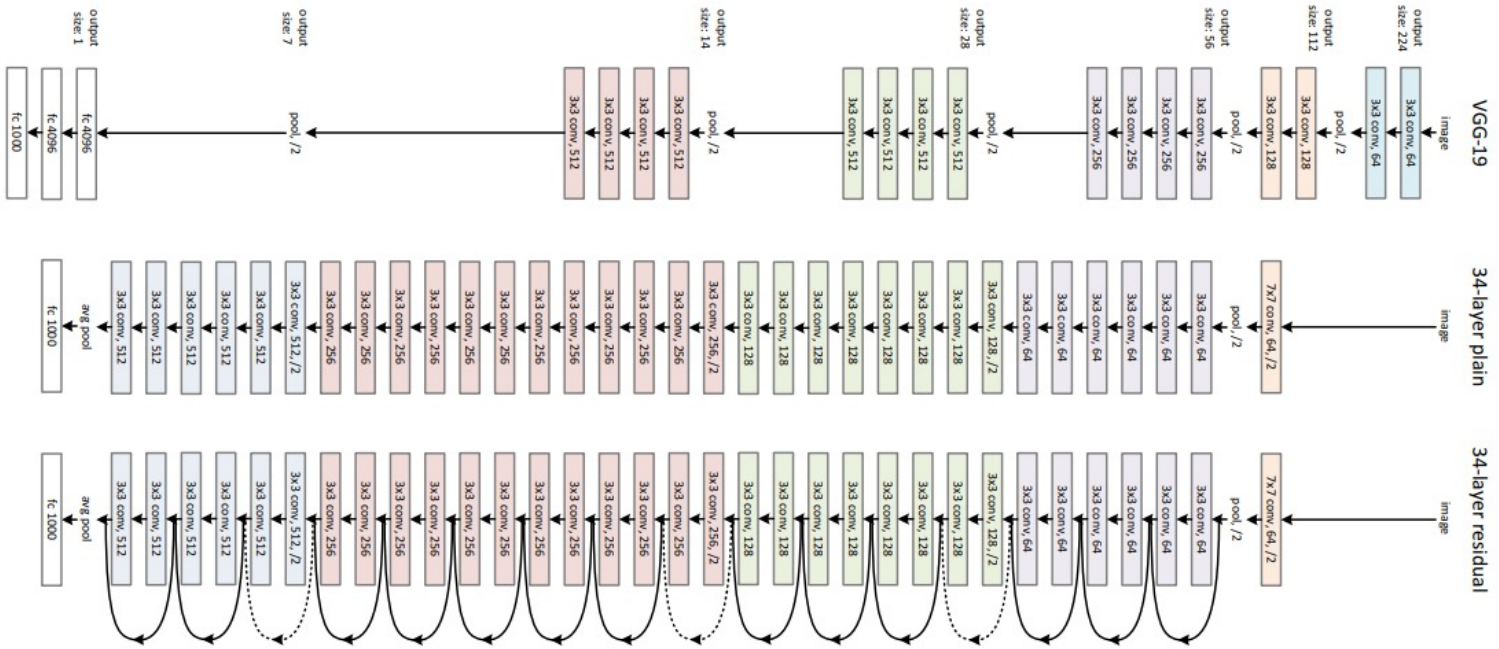
Desired underlying mapping: $\mathcal{H}(x)$

We fit the **residual mapping**: $\mathcal{F}(x) \coloneqq \mathcal{H}(x) - x$

That's why it was called **Deep Residual Learning**

[1] K. He, et al. Deep Residual Learning for Image Recognition. CVPR 2016

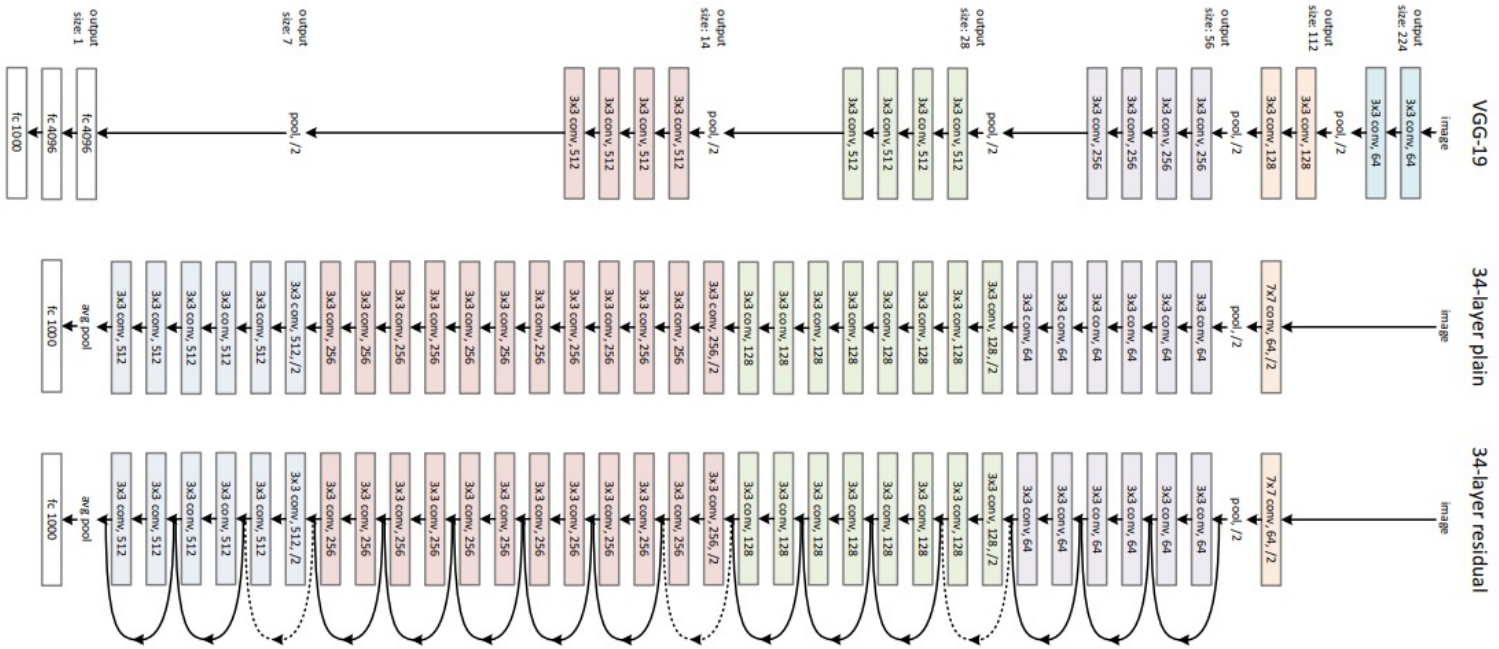
ResNet



- If any layer hurts the performance of architecture then it will be skipped. This results in training a very deep neural network.
- No extra parameter nor computational complexity

Img from: https://www.geeksforgeeks.org/residual-networks-resnet-deep-learning/

ResNet



	plain	ResNet
18 layers	27.94	27.88
34 layers	28.54	**25.03**

Table 2. Top-1 error (%, 10-crop testing) on ImageNet validation. Here the ResNets have no extra parameter compared to their plain counterparts. Fig. 4 shows the training procedures.

Img from: https://www.geeksforgeeks.org/residual-networks-resnet-deep-learning/

Reference for Topic 3

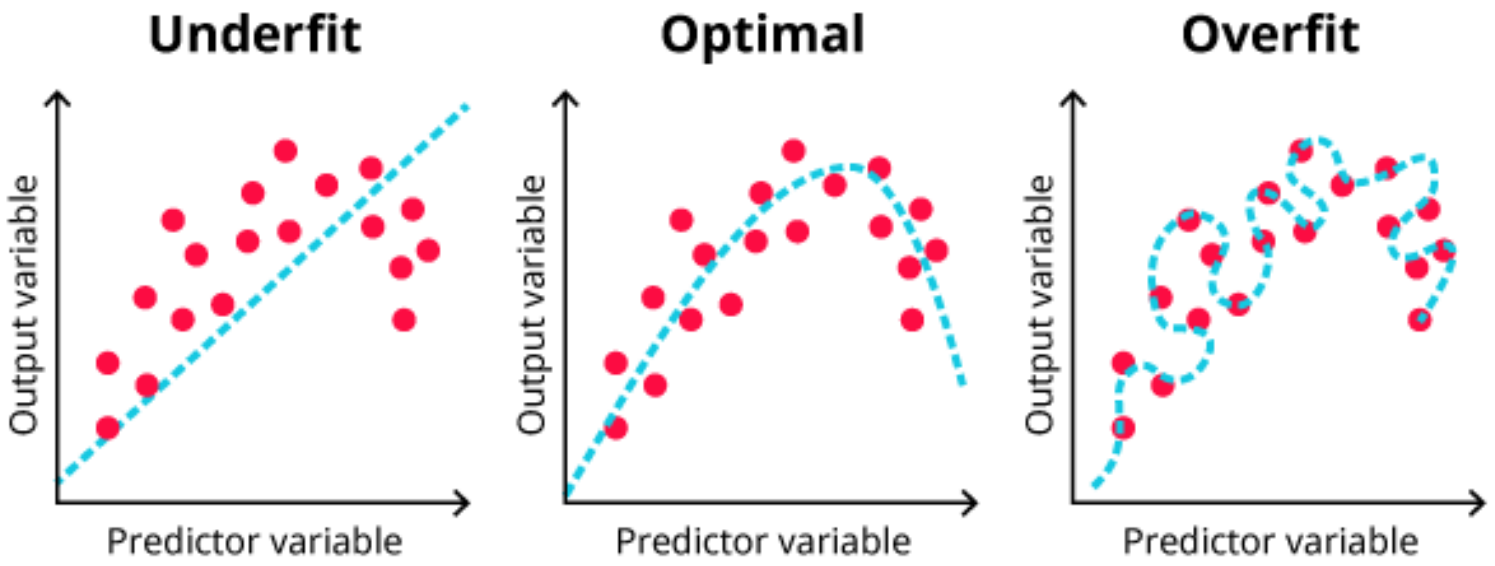
- Book: Aurelien Geron. Hands-On Machine Learning with Scikit-Learn and TensorFlow. O'Reilly. 2019.
- http://neuralnetworksanddeeplearning.com/chap5.html
- https://towardsdatascience.com/the-vanishing-gradient-problem-69bf08b15484

Topic 4:
Overfitting Problem

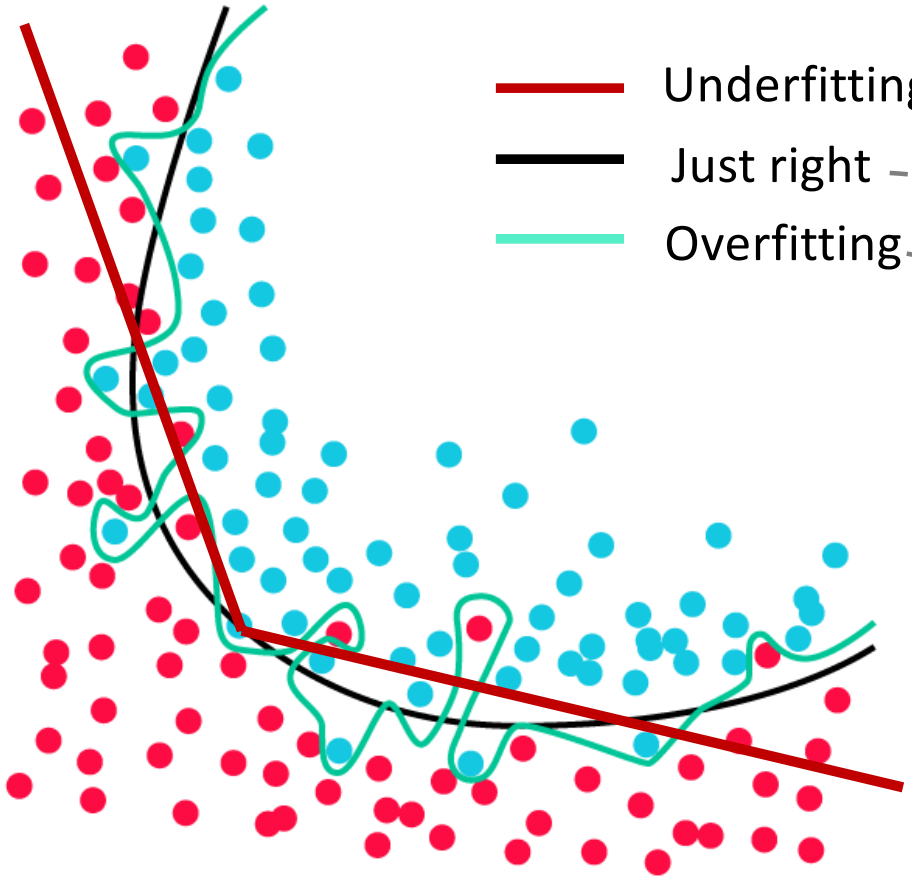
Underfitting and Overfitting

- Too simple
- Does not fully learn the data

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



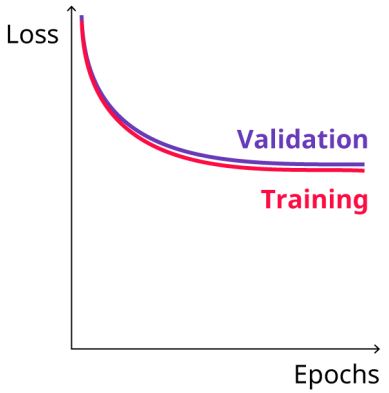
How to Identify Underfitting and Overfitting

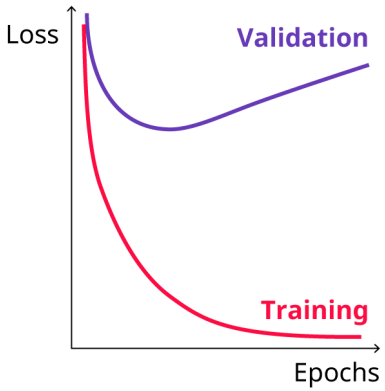


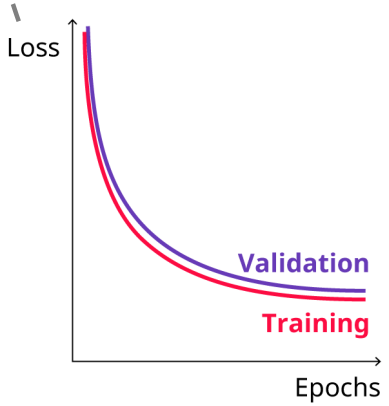
Underfitting

Just right

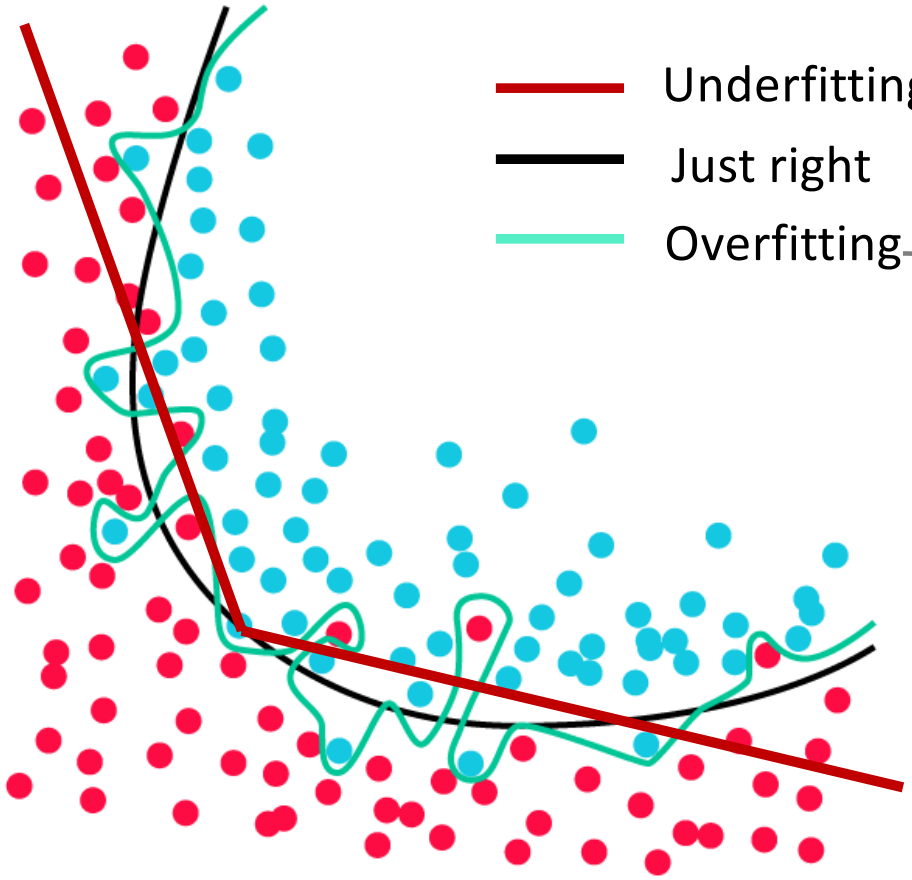
Overfitting







How to Prevent Underfitting and Overfitting

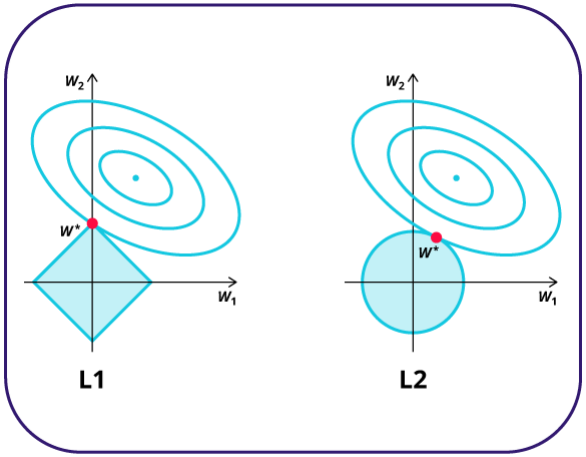


- Complexify model
- Add more nodes/layers
- Train longer

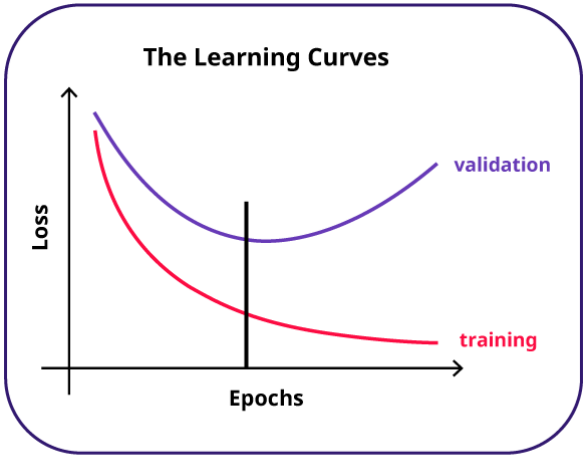
- More training data
  - Data augmentation
- Simplify model
- Regularization

Regularization

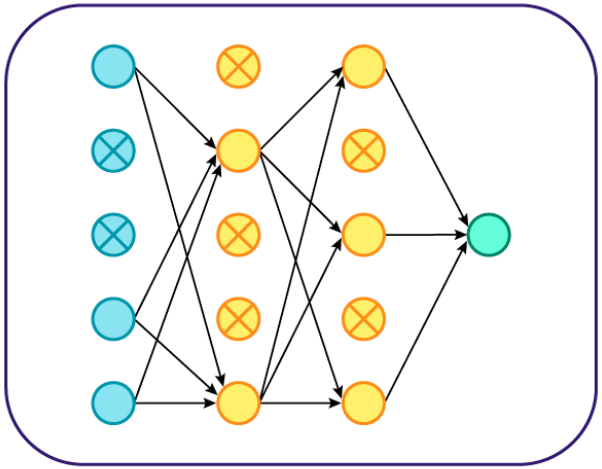
- **Regularization** is a technique which constraints the optimization problem to discourage complex model.
- Regularization helps the model generalize better on unseen data.



Weight regularization



Early stopping



Dropout

Weight Regularization

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

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

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

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'))
```

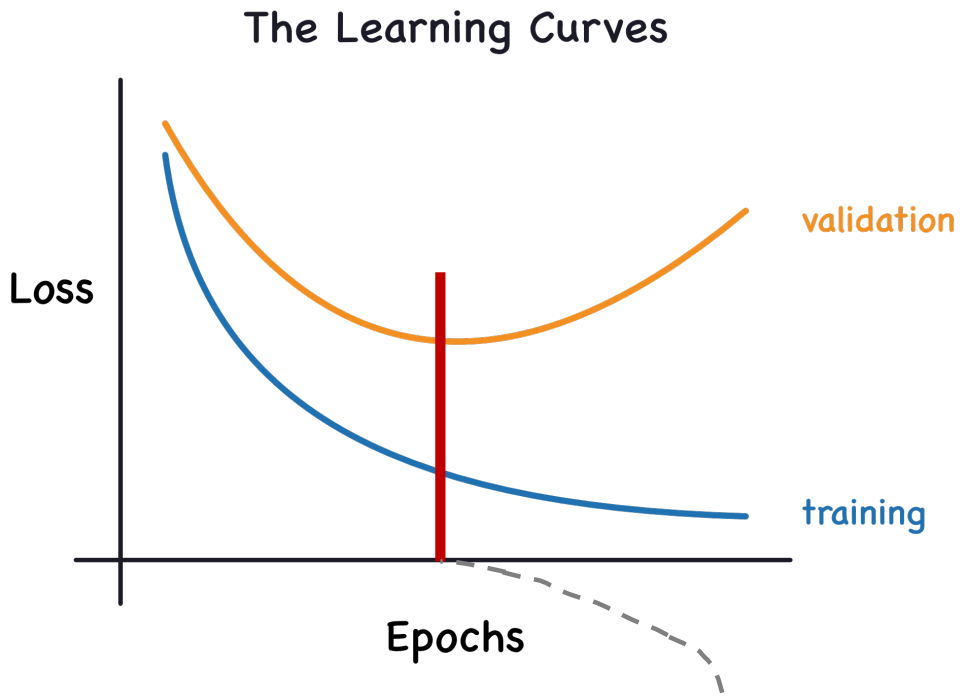
L2 Regularization:

- 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

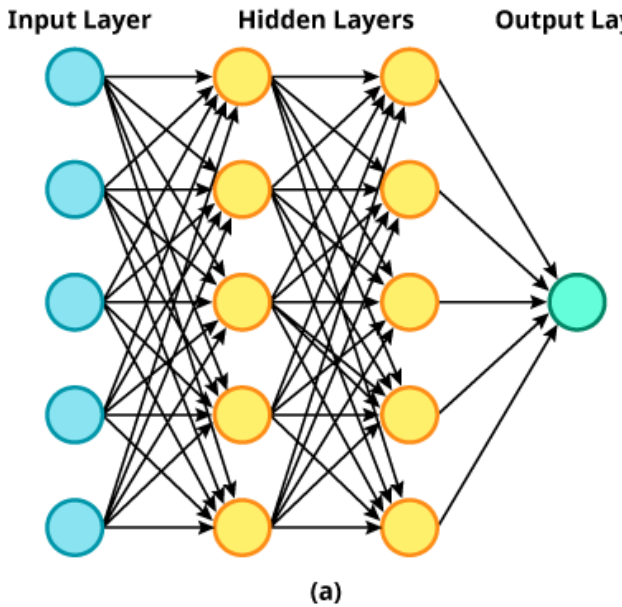
Early Stopping

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.

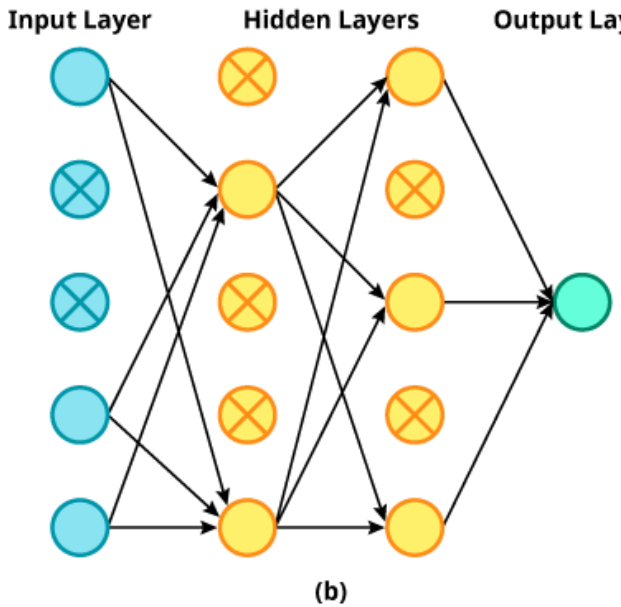
```
from keras.callbacks import EarlyStopping

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.



(a)

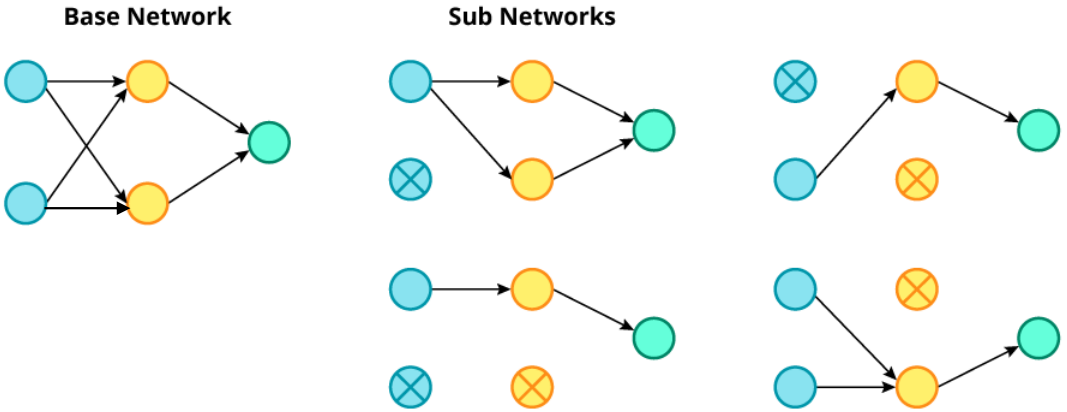


(b)

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

Dropout

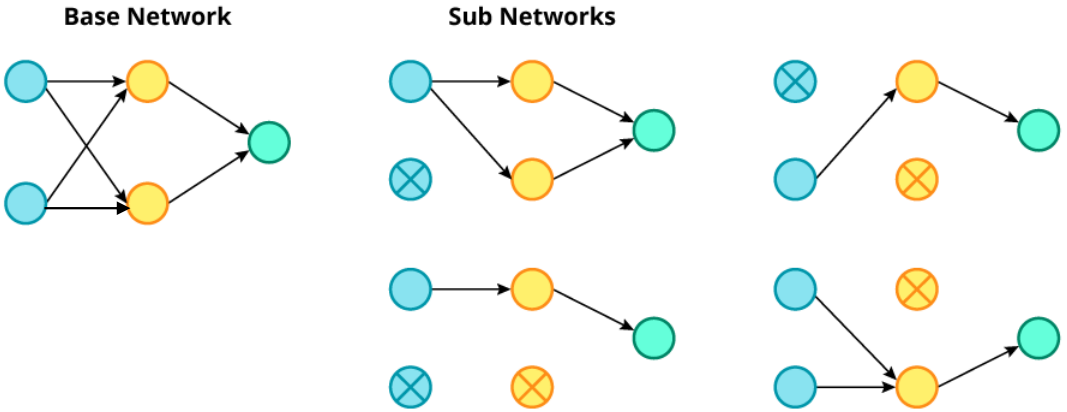
A: NO. Because the removed nodes are different in each training iteration.



- 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

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



Q: Should dropout process be also applied in the testing stage?

A: No.

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

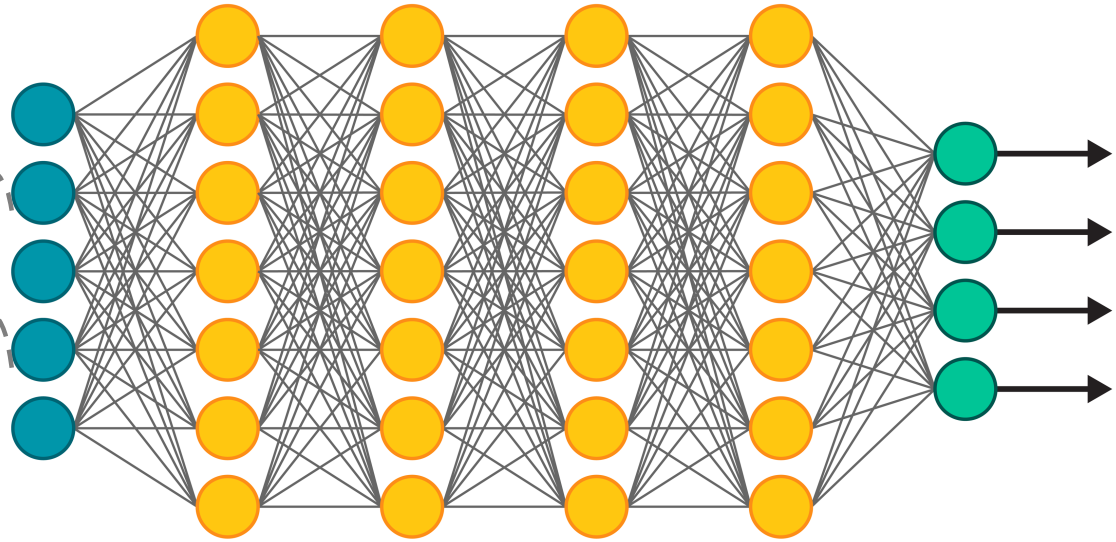
- Video lecture by Alexander Amini: MIT course on deep learning, https://www.youtube.com/watch?v=njKP3FqW3Sk
- https://www.kdnuggets.com/2019/12/5-techniques-prevent-overfitting-neural-networks.html
- https://medium.com/@jennifer.arty/regularization-methods-to-prevent-overfitting-in-neural-networks-1a79b5e3081f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

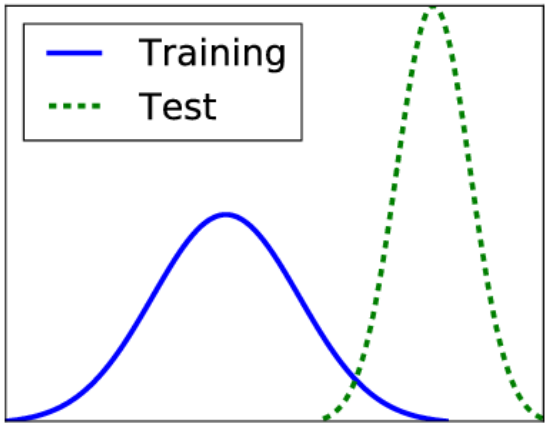
A: **Batch Normalization**.

Intuition for Batch Normalization

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

The change in the input distribution to a learning algorithm.

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: $\mathcal{B} = \{x_{1...m}\}$;
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)

It can be used with most network types, such as Multilayer Perceptrons (MLPs), Convolutional Neural Networks(CNNs) and Recurrent Neural Networks(RNNs).

Reference for Topic 5

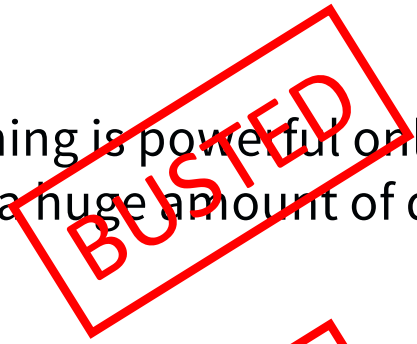
- Book: Aurelien Geron. Hands-On Machine Learning with Scikit-Learn and TensorFlow. O'Reilly. 2019.
- https://towardsdatascience.com/batch-normalization-in-neural-networks-1ac91516821c
- https://mlexplained.com/2018/01/10/an-intuitive-explanation-of-why-batch-normalization-really-works-normalization-in-deep-learning-part-1/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

Topic 6:
Data Augmentation

Training CNNs

- CNN is trained in the same way as MLP: gradient descent with backpropagation.
- In practice, it is relatively rare to have sufficient training data:
  - Data collection and annotation is expensive.
  - Sometimes near-impossible (e.g., medical images of a rare disease)

Training CNNs

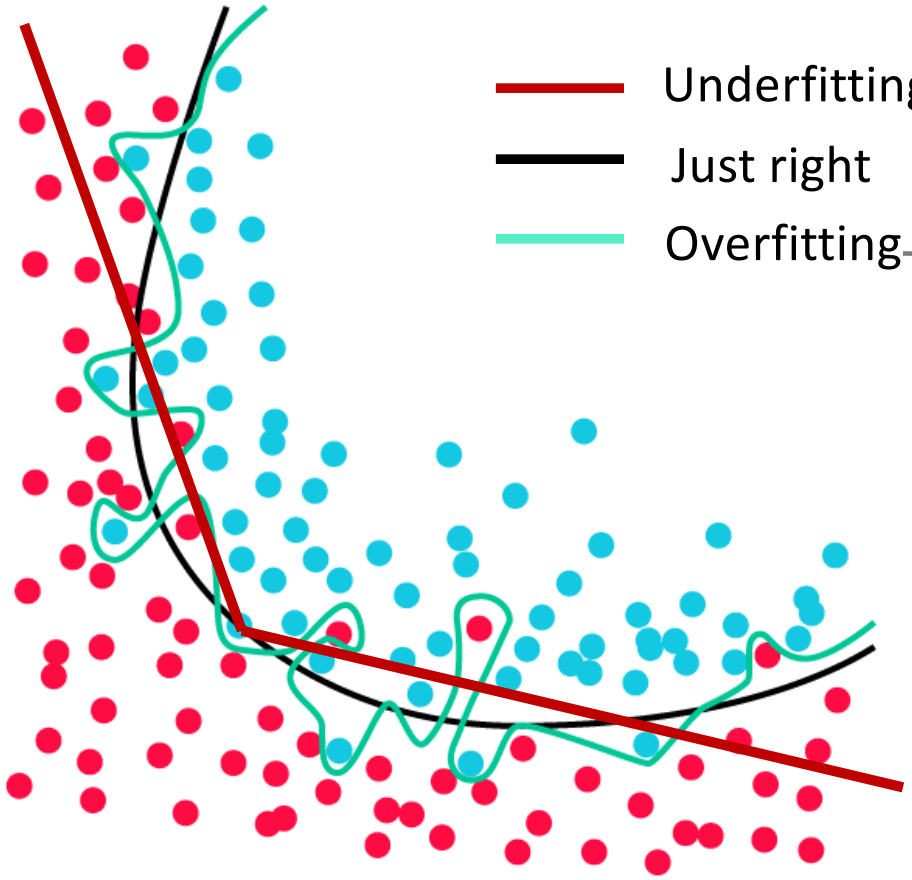


"Deep learning is powerful only when you have a huge amount of data."

"You can only use small CNNs if your dataset is small."

- ✓ **Data augmentation**: Expanding the training set
- ✓ **Transfer learning**: Making Less Data Cool Again

How to Prevent Underfitting and Overfitting



Underfitting

Just right

Overfitting

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

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

Data Augmentation

- **Data augmentation** is to generate synthetic data from existing training examples, by *augmenting* the samples via a number of random transformations.

```
from keras.preprocessing.image import ImageDataGenerator

train_datagen = ImageDataGenerator(
        rescale=1./255,
        rotation_range = 40,
        width_shift_range = 0.2,
        height_shift_range = 0.2,
        shear_range = 0.2,
        zoom_range = 0.2,
        horizontal_flip = True,
        fill_mode = 'nearest')

test_datagen = ImageDataGenerator(rescale=1./255)
```

The validation and test data shouldn't be augmented.

Augmented examples

Reference for Topic 6

- Book: Francois Chollet. Deep Learning with Python. Manning. 2018.
- Blog by Francois Chollet: https://blog.keras.io/building-powerful-image-classification-models-using-very-little-data.html

Topic 7:
Transfer Learning

Transfer learning

- **Transfer learning** is a machine learning technique where a model trained on one task is re-purposed on another related task.

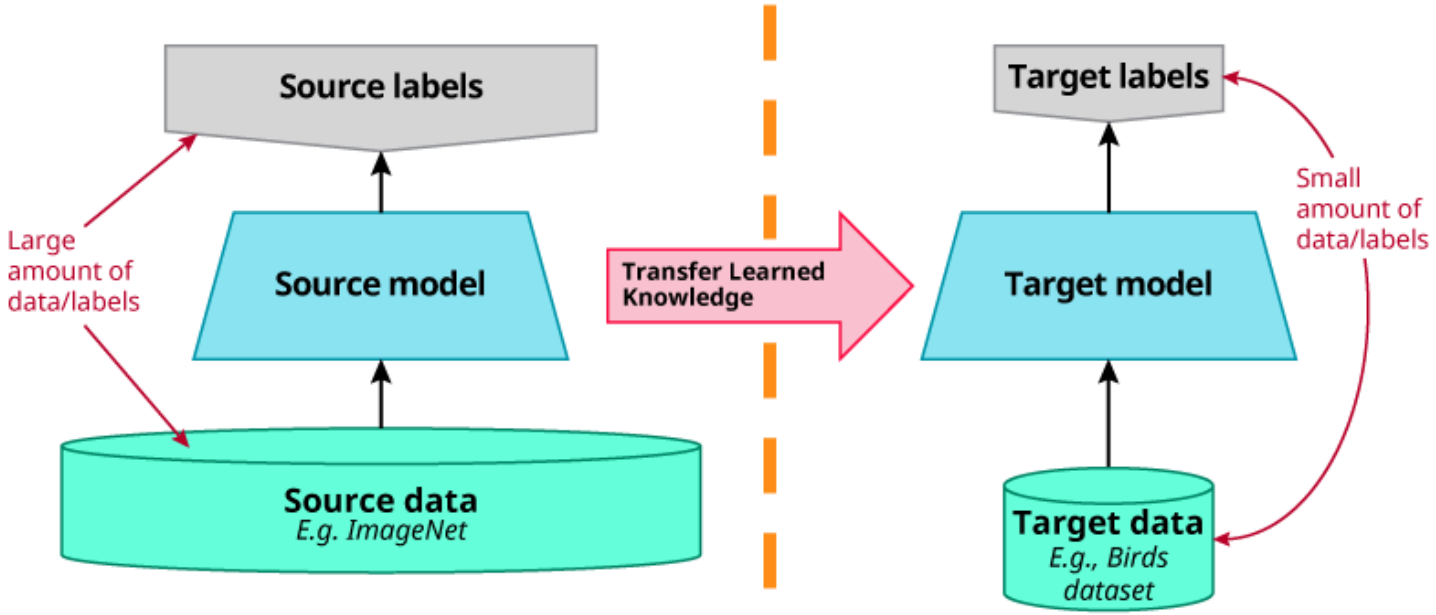
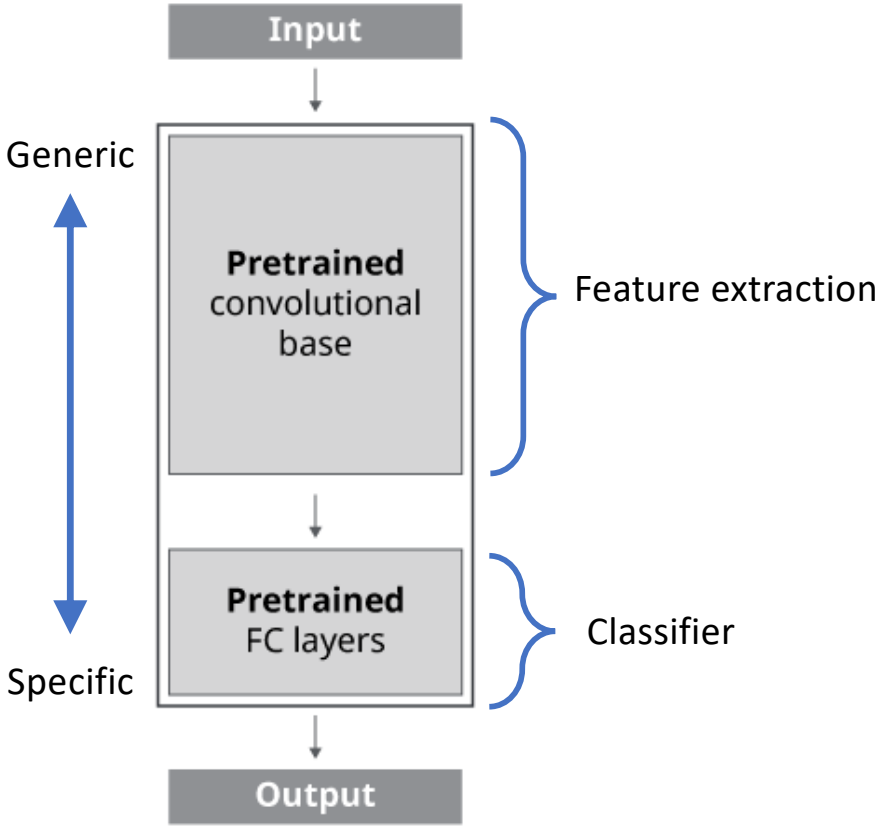


Image from: https://medium.com/the-official-integrate-ai-blog/transfer-learning-explained-7d275c1e34e2

The underlying assumption



- The **underlying assumption** of transfer learning is that generic features learned on a large enough dataset can be shared among seemingly disparate datasets.

Two ways for transfer learning with CNNs

1. Feature extraction

2. Fine-tuning

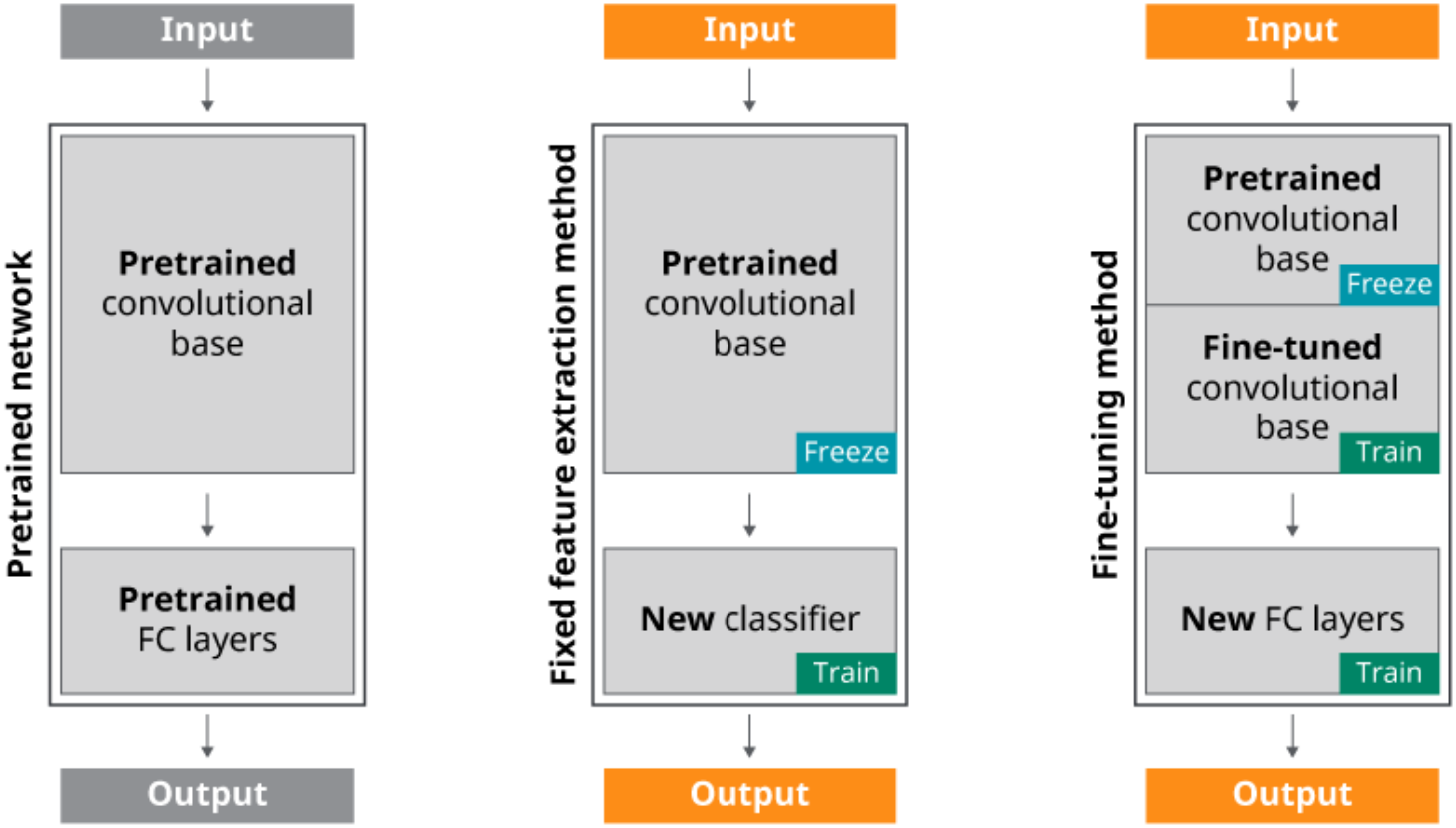
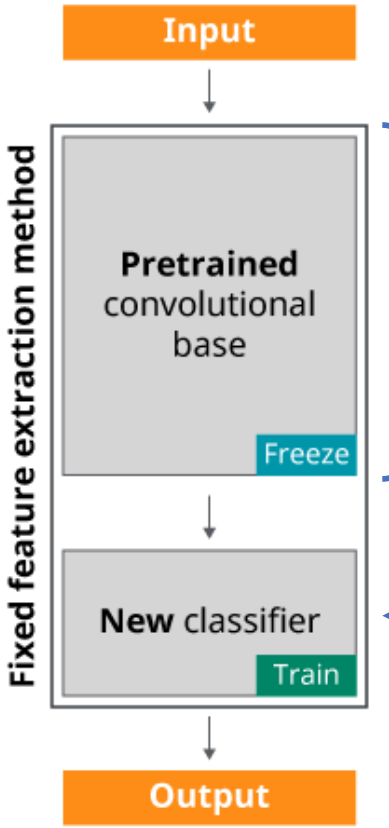


Image from: https://insightsimaging.springeropen.com/articles/10.1007/s13244-018-0639-9

1. Feature extraction



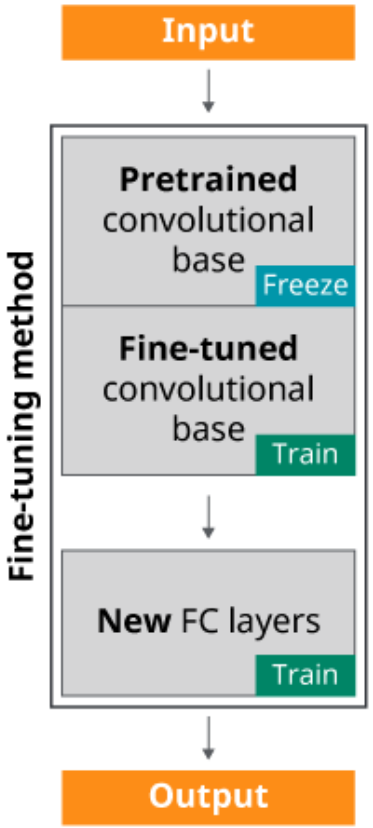
Step 1: Extract features with the pre-trained model

Step 2: train a classifier with the input features

- The new classifier could be any machine learning classifier, such as SVM, random forests and also the FC layers.

Image from: https://insightsimaging.springeropen.com/articles/10.1007/s13244-018-0639-9

2. Fine-tuning



- **Fine-tuning**: the learnt filters are not randomly initialized, but start from the pre-trained model. These filters will be slightly adjusted.
- It is common to fine-tune only the higher layers in the convolutional base, but still freeze the early layers, since early-features appear more generic.
- Use lower learning rate while fine-tuning, 1/10 of the original learning rate would be a good start point.

Image from: https://insightsimaging.springeropen.com/articles/10.1007/s13244-018-0639-9

Pre-trained models

- **Q**: Do I have to pre-train a model myself on a large dataset, then use it for feature extraction or fine-tuning on a small dataset?
- **A**: Not necessary. Many models pretrained on ImageNet are open to public:
  - AlexNet,
  - VGG
  - ResNet
  - Inception
  - Xception
  - DenseNet
  - MobileNet
  - ….

Pre-trained models

- In Keras, all pre-trained models are available in `keras.application`.

```
from keras.application import VGG16

conv_base = VGG16(weights='imagenet',
                  include_top = False,
                  input_shape=(150,150,3))
```

False means not including the fully connected layers .

Model	Size	Top-1 Accuracy	Top-5 Accuracy	Parameters	Depth
Xception	88 MB	0.790	0.945	22,910,480	126
VGG16	528 MB	0.713	0.901	138,357,544	23
VGG19	549 MB	0.713	0.900	143,667,240	26
ResNet50	98 MB	0.749	0.921	25,636,712	-
ResNet101	171 MB	0.764	0.928	44,707,176	-
ResNet152	232 MB	0.766	0.931	60,419,944	-
ResNet50V2	98 MB	0.760	0.930	25,613,800	-
ResNet101V2	171 MB	0.772	0.938	44,675,560	-
ResNet152V2	232 MB	0.780	0.942	60,380,648	-
InceptionV3	92 MB	0.779	0.937	23,851,784	159
InceptionResNetV2	215 MB	0.803	0.953	55,873,736	572
MobileNet	16 MB	0.704	0.895	4,253,864	88
MobileNetV2	14 MB	0.713	0.901	3,538,984	88
DenseNet121	33 MB	0.750	0.923	8,062,504	121
DenseNet169	57 MB	0.762	0.932	14,307,880	169
DenseNet201	80 MB	0.773	0.936	20,242,984	201

...

All available models in Keras: https://keras.io/api/applications/

Reference for Topic 7

- Book: Francois Chollet. Deep Learning with Python. Manning. 2018.
- Stanford University course: CNNs for Visual Recognition https://cs231n.github.io/transfer-learning/
- Blog by Francois Chollet: https://blog.keras.io/building-powerful-image-classification-models-using-very-little-data.html
- R. Yamashita, et al. Convolutioanl neural networks: an overview and applicatin in radiology. Insights into imaging. P611-629, 2018. https://insightsimaging.springeropen.com/articles/10.1007/s13244-018-0639-9