

Background

Deep Learning

Mohammad Taher Pilehvar

<https://teias-courses.github.io/dl99>



Types of Machine Learning

- Four branches:
 - Supervised learning
 - Unsupervised learning
 - Self-supervised
 - Reinforcement learning

Supervised learning

- The most common case
- Learning to map input data to known targets (also called annotations), given a set of examples (often annotated by humans).
 - Optical character recognition
 - Speech recognition
 - Image classification
 - Language translation

Supervised learning

- Mostly classification and regression
- But more exotic variants:
 - Sequence generation - Given a picture, predict a caption describing it.
 - Syntax tree prediction - Given a sentence, predict its decomposition into a syntax tree.
 - Object detection - Given a picture, draw a bounding box around certain objects inside the picture.
 - Image segmentation - Given a picture, draw a pixel-level mask on a specific object.

Supervised learning

One-shot learning

You get only one or a few training examples in some categories.

Zero-shot learning

You are not presented with every class label in training. So in some categories, you get 0 training examples.

Unsupervised learning

Finds interesting transformations of the input data without the help of any targets, for the purposes of data visualization, data compression, or data denoising, or to better understand the correlations present in the data at hand.

- Dimensionality reduction
- Clustering

Self-supervised learning

Supervised learning without human-annotated labels.

There are still labels involved but they're generated from the input data, typically using a heuristic algorithm.

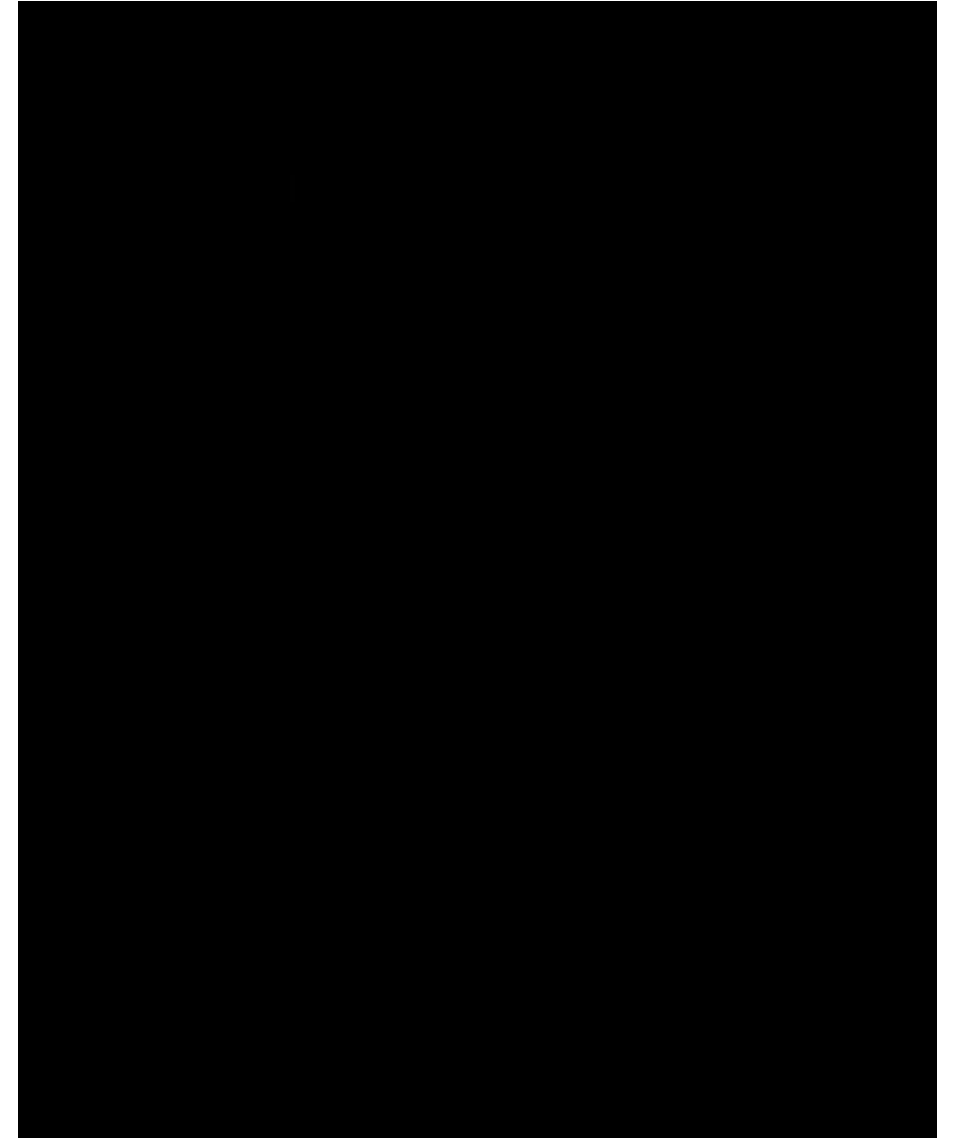
- Autoencoders - the generated targets are the input, unmodified.
- Predict the next frame in a video, given past frames, or the next word in a text, given previous words.
- Language modeling

Reinforcement learning

- At the moment mostly research
- Expected to soon take over large number of real-world applications: self-driving cars, robotics, resource management, education

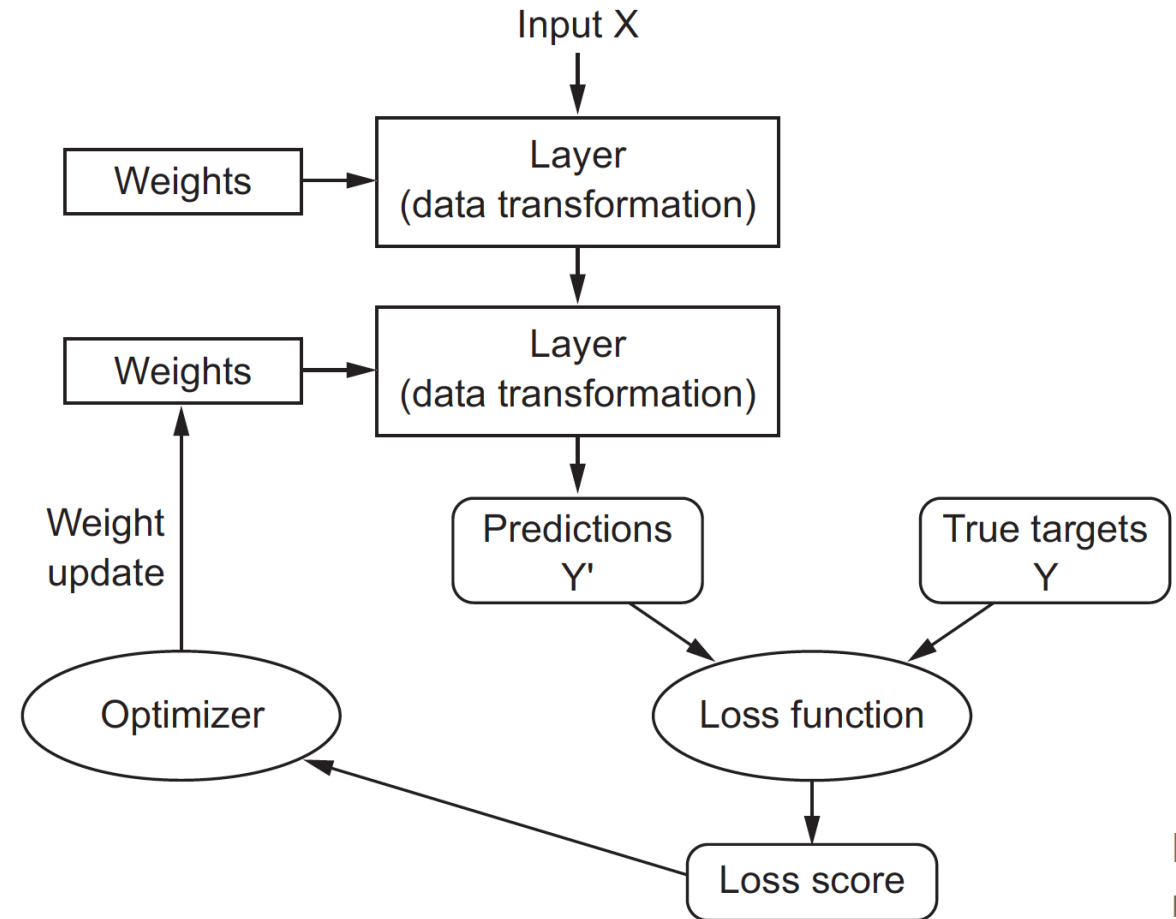
Reinforcement learning (II)

- Atari video



Anatomy of a neural network

- Model
 - Architecture + Parameters
- Activation function
- Optimizer
- Hyperparameters



Case study

- Day 'n' Night classification
- Face verification
- Neural style transfer (Art generation)
- Trigger-word detection

Case study I: Day 'n' Night classification

- Data?

10K images

Split? Bias?

- Input?



Resolution? (64,64,3)

- Output?

$y = 0$ or $y = 1$

sigmoid

- Architecture?

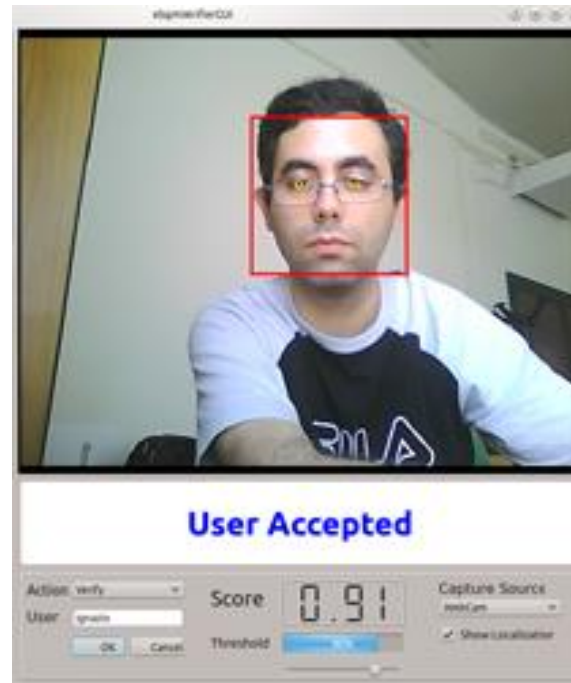
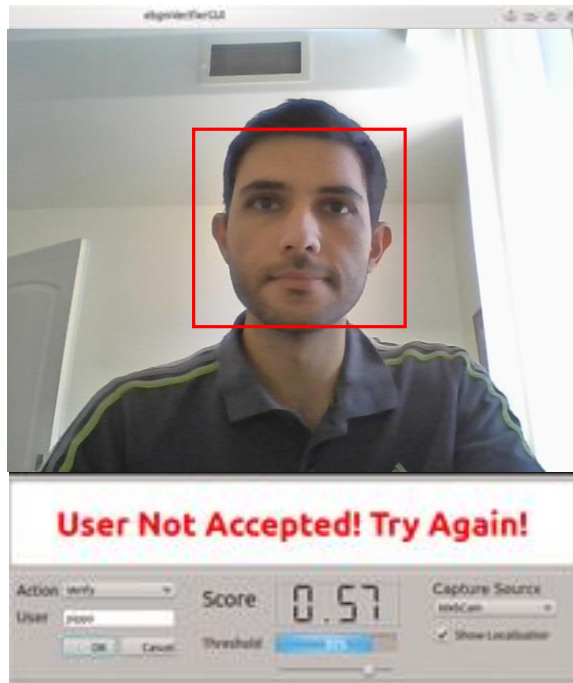
Shallow CNN

- Loss?

$$L = -[y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})]$$

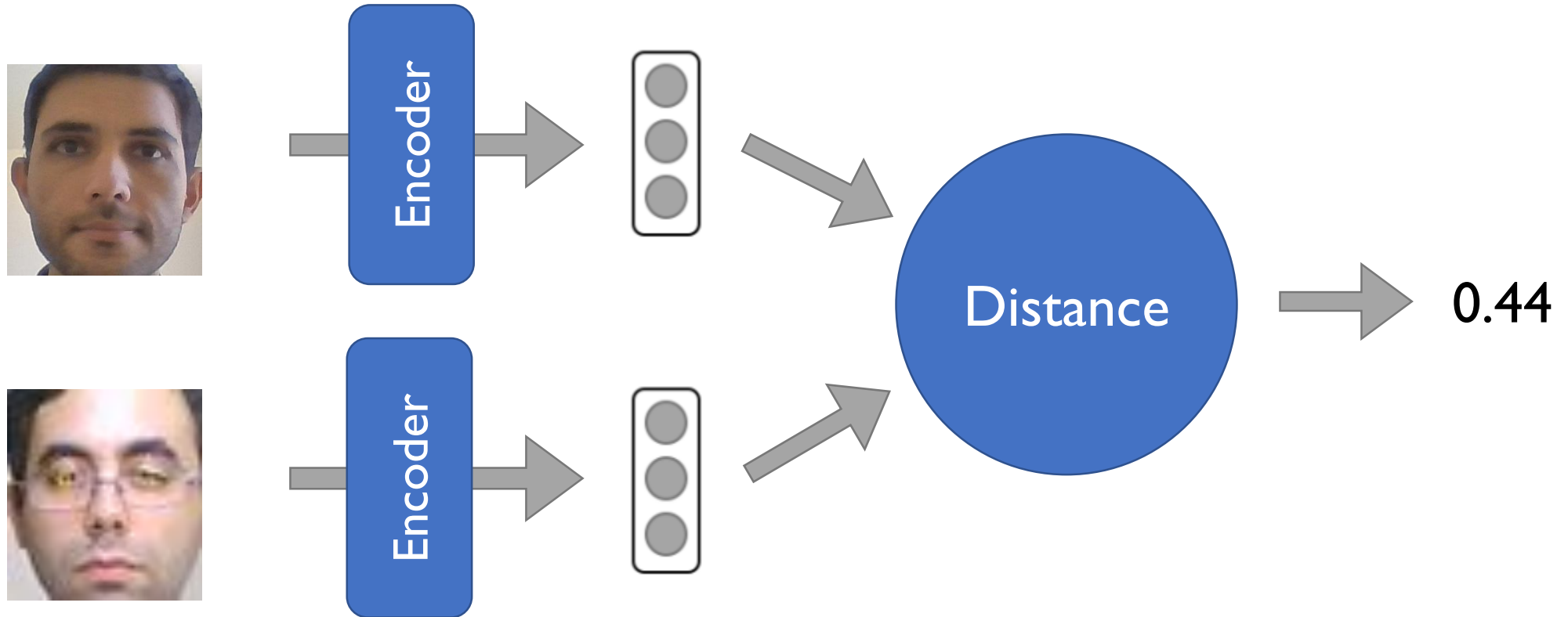
Case study 2: Face Verification

- Data? Labeled pictures
- Input? Resolution?
- Output? Binary classification



Case study 2: Face Verification

- Architecture?



Case study 2: Face Verification

- Loss? Training?

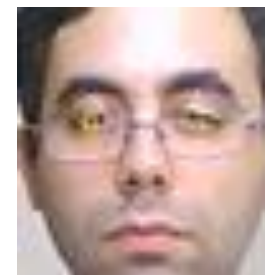
Triplets for training



0



1



positive anchor negative



Minimize
distance

Maximize
distance

Case study 2: Face Verification

- Which loss?

$$L = \left\| \text{Enc}(A) - \text{Enc}(P) \right\|_2^2$$

$$- \left\| \text{Enc}(A) - \text{Enc}(N) \right\|_2^2$$

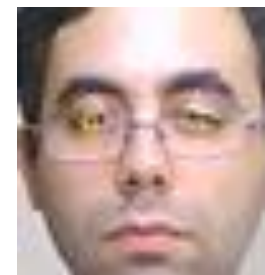
$$L = \left\| \text{Enc}(A) - \text{Enc}(N) \right\|_2^2$$

$$- \left\| \text{Enc}(A) - \text{Enc}(P) \right\|_2^2$$

$$L = \left\| \text{Enc}(P) - \text{Enc}(N) \right\|_2^2$$

$$- \left\| \text{Enc}(P) - \text{Enc}(A) \right\|_2^2$$

Triplets for training



positive anchor negative



Minimize
distance

Maximize
distance

Case study 2

Face Identification

Identify people in a photo

K-Nearest Neighbours?



Face Clustering

Group photos of people in your gallery

K-Means?



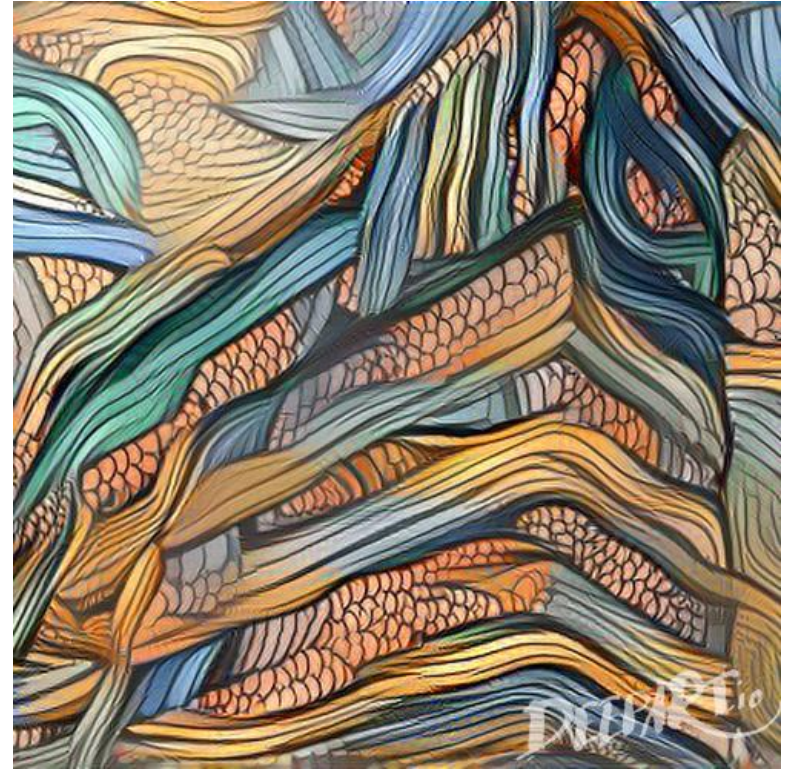
Case study 3: Art Generation



Content image



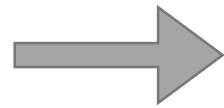
Style image



Generated image

Case study 3: Art Generation

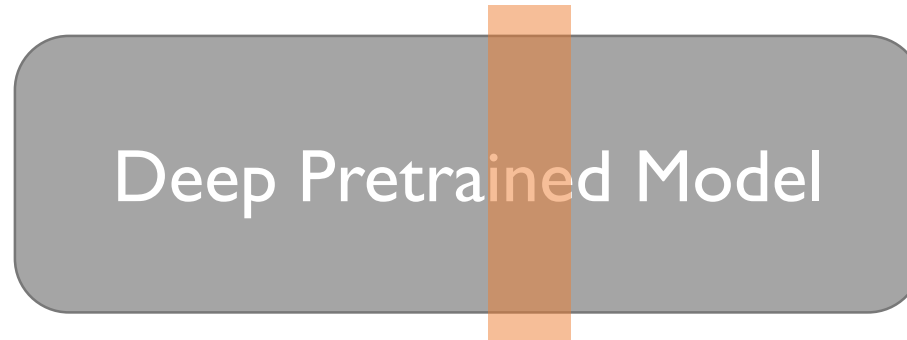
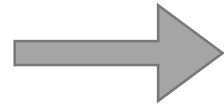
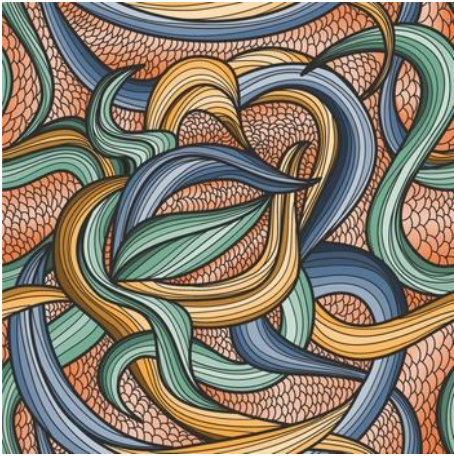
- Architecture?



Content

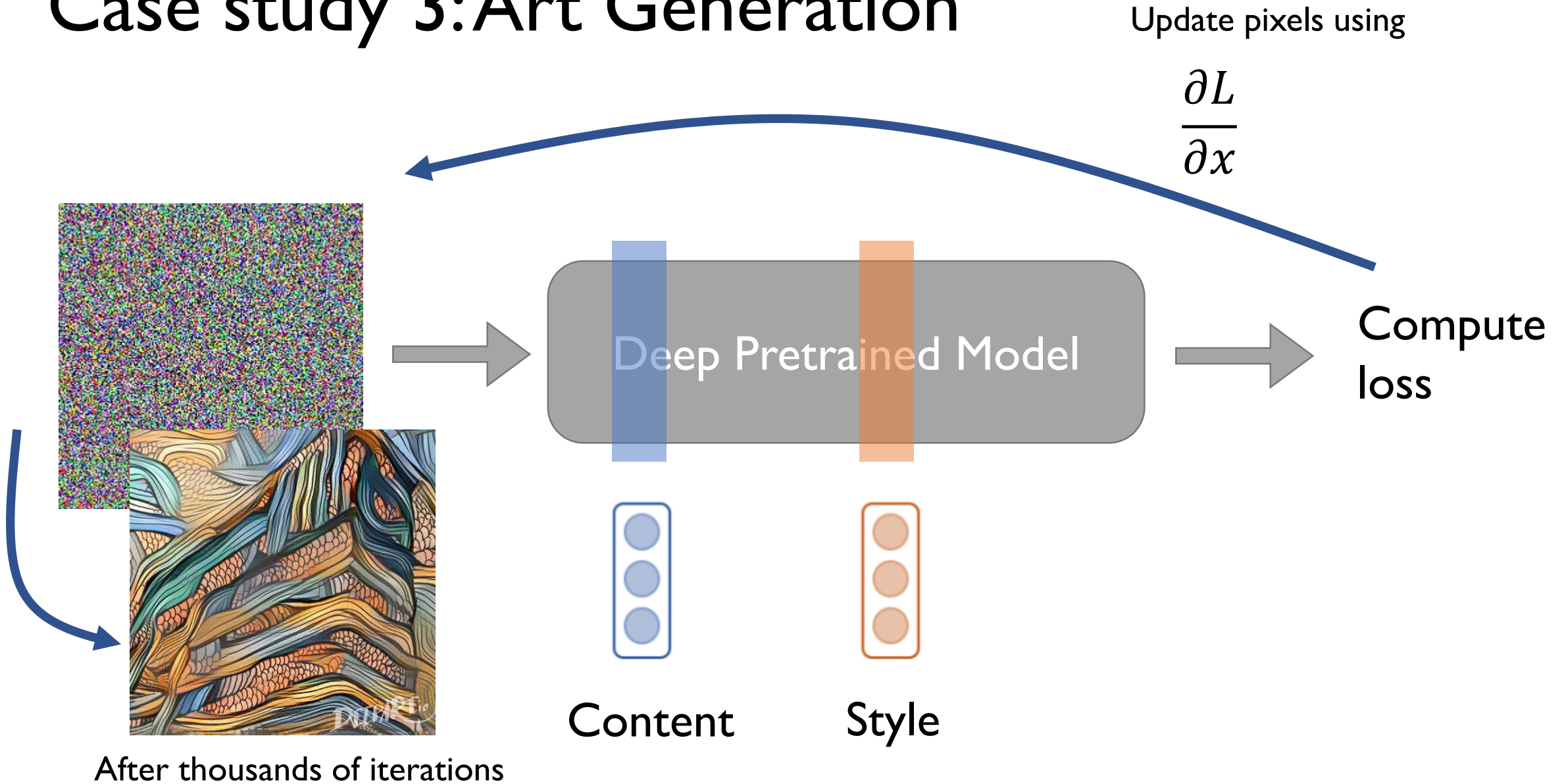
Case study 3: Art Generation

- Architecture?



Style

Case study 3: Art Generation



Case study 3: Art Generation

- Loss?

$$L = \left\| \text{Content}_C - \text{Content}_G \right\|_2^2$$

$$- \left\| \text{Style}_S - \text{Style}_G \right\|_2^2$$

$$L = \left\| \text{Style}_S - \text{Style}_G \right\|_2^2$$

$$+ \left\| \text{Content}_C - \text{Content}_G \right\|_2^2$$


$$L = \left\| \text{Style}_S - \text{Style}_G \right\|_2^2$$

$$- \left\| \text{Content}_C - \text{Content}_G \right\|_2^2$$

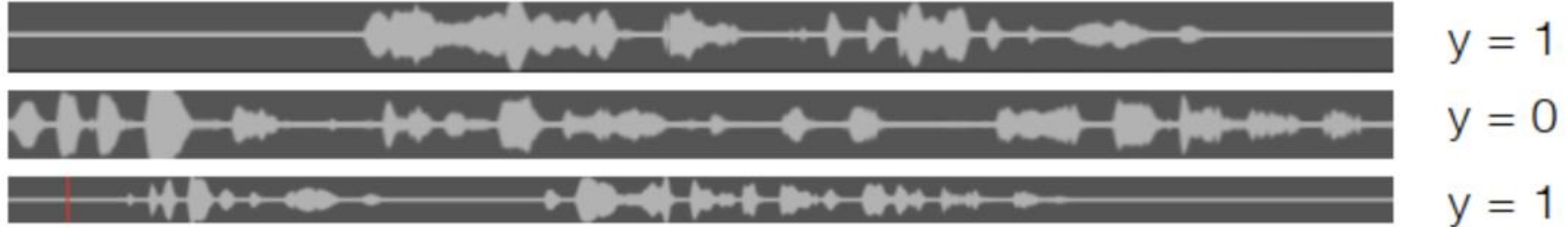
Case study 3: Art Generation




Case study 4: Trigger Word Detection

- Data? A bunch of 10sec audio clips, Distribution?
- Input? Resolution? 
- Output? Binary classification

Case study 4: Trigger Word Detection



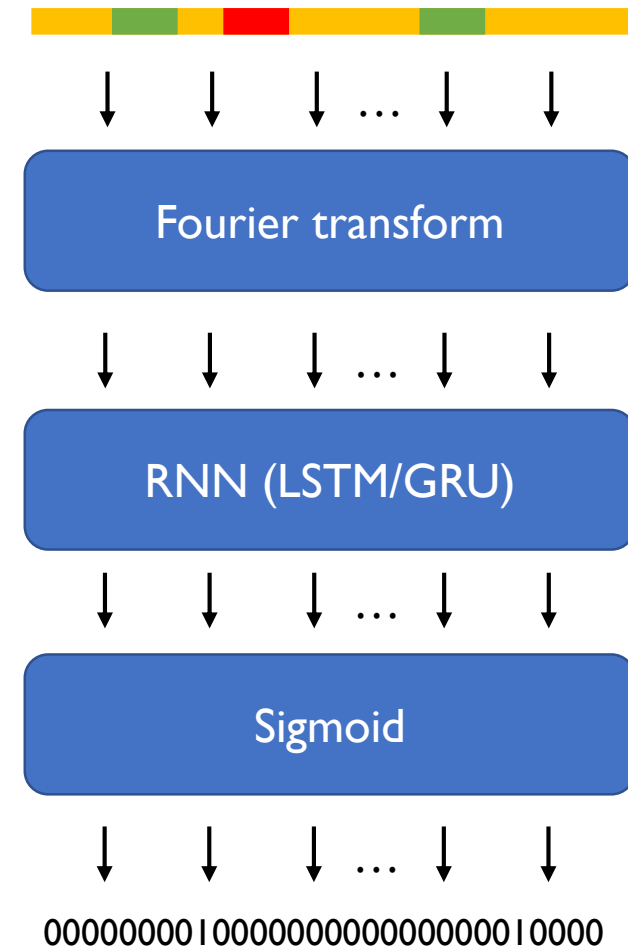
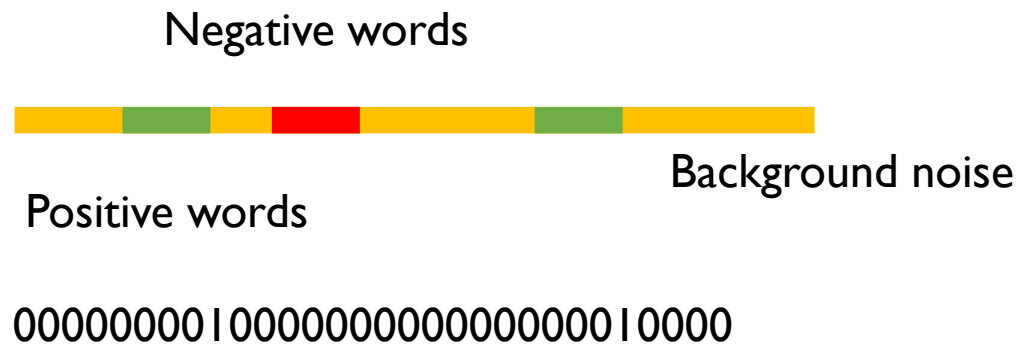
Case study 4: Trigger Word Detection

- Data? A bunch of 10sec audio clips, Distribution?
- Input? Resolution? 
- Output? $y = 000000010000$ Last activation?
Sigmoid (sequential)
- Architecture RNN?
- Loss
$$L = -[y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})]$$

(sequential)

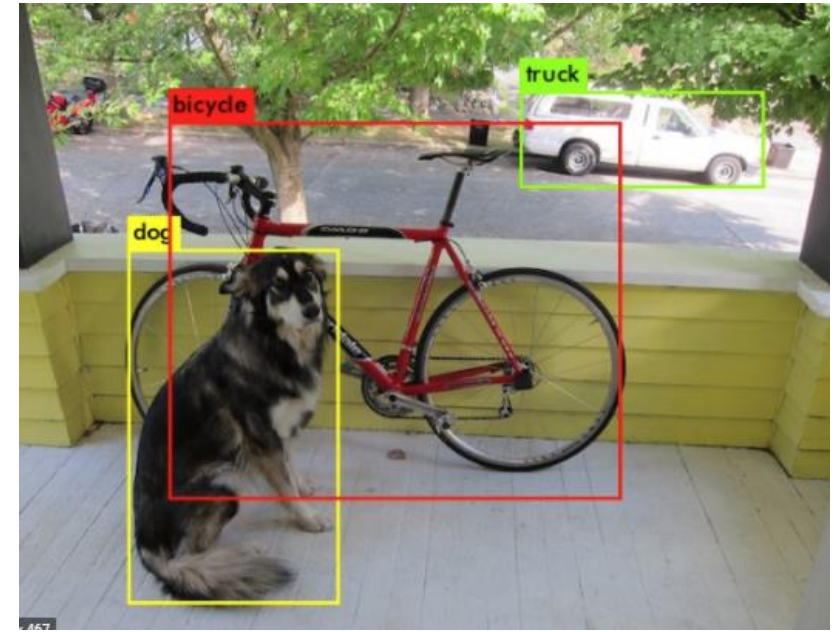
Case study 4: Trigger Word Detection

- How to collect data?
 - Generate!
- Architecture?



What loss function is this?

$$\begin{aligned} & \lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^B 1_{ij}^{obj} [(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2] \\ & + \lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^B 1_{ij}^{obj} [(\sqrt{w_i} - \sqrt{\hat{w}_i})^2 + (\sqrt{h_i} - \sqrt{\hat{h}_i})^2] \\ & + \sum_{i=0}^{S^2} \sum_{j=0}^B 1_{ij}^{obj} (C_i - \hat{C}_i)^2 + \lambda_{noobj} \sum_{i=0}^{S^2} \sum_{j=0}^B 1_{ij}^{noobj} (C_i - \hat{C}_i)^2 \\ & + \sum_{i=0}^{S^2} 1_i^{obj} \sum_{c \in classes} (p_i(c) - \hat{p}_i(c))^2 \end{aligned}$$



Evaluating machine learning models

Data is usually split into:

- Training set
- Validation (development) set
- Test set

In machine learning, the goal is to achieve models that generalize—that perform well on never-before-seen data—and overfitting is the central obstacle

Training, validation, and test sets

Train on training set and test on the test set!

Why validation set?

Developing a model always involves tuning its configuration: for example, choosing the number of layers or the size of the layers (called the hyperparameters)

Overfitting

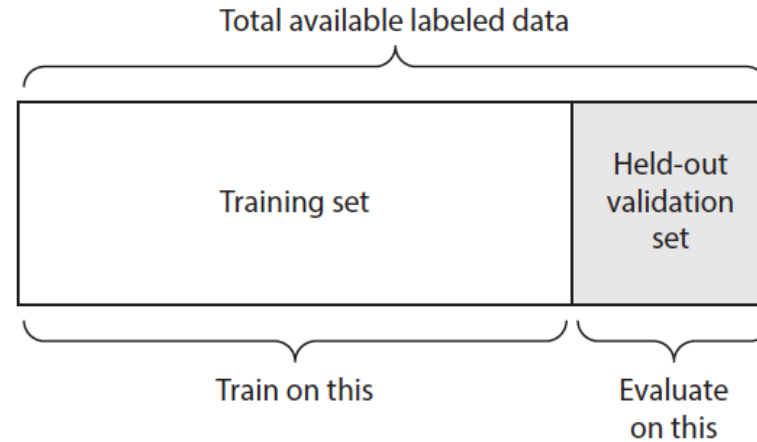
- Parameter tuning is a form of learning.
- If you do this tuning excessively, there is a risk for overfitting to the validation set.
- Model performs artificially well on the validation data but not necessarily well on the test (which is your final goal).

How to split data?

- Simple hold-out validation
- K-fold validation
- Iterated K-fold validation with shuffling

Splitting data: Simple hold-out validation

- Set apart some fraction of your data as your test set.
- Train on the remaining data, and evaluate on the test set.
- Remember we also need a validation set!



Splitting data: Simple hold-out validation (II)

Listing 4.1 Hold-out validation

```
num_validation_samples = 10000
np.random.shuffle(data)
validation_data = data[:num_validation_samples]
data = data[num_validation_samples:]
training_data = data[:]

model = get_model()
model.train(training_data)
validation_score = model.evaluate(validation_data)

# At this point you can tune your model,
# retrain it, evaluate it, tune it again...

model = get_model()
model.train(np.concatenate([training_data,
                           validation_data]))
test_score = model.evaluate(test_data)
```

Shuffling the data is usually appropriate.

Defines the validation set

Defines the training set

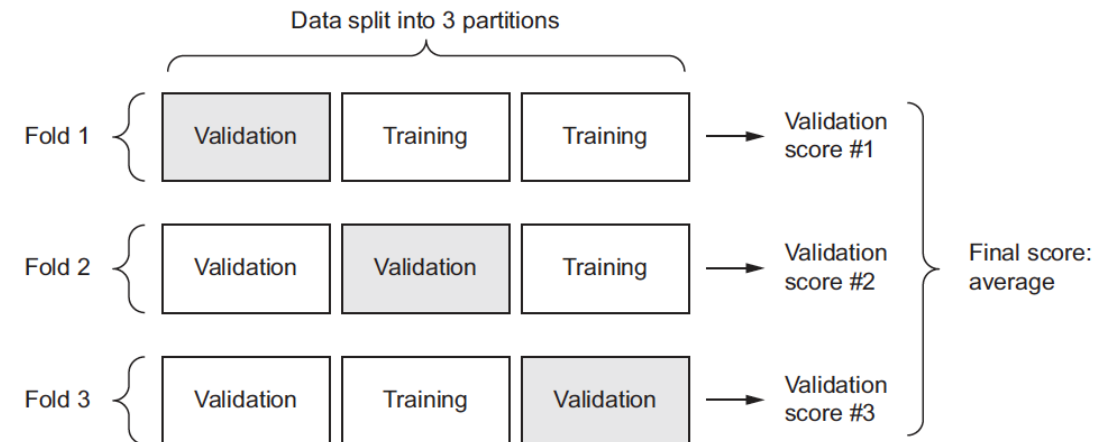
Trains a model on the training data, and evaluates it on the validation data

Once you've tuned your hyperparameters, it's common to train your final model from scratch on all non-test data available.

Splitting data: K-fold validation

If little data is available, then your validation and test sets may contain too few samples to be statistically representative of the data at hand

- K-fold validation: you split your data into K partitions of equal size. For each partition i , train a model on the remaining $K - 1$ partitions, and evaluate it on partition i .
- Your final score is then the averages of the K scores obtained



Splitting data: K-fold validation (II)

Listing 4.2 K-fold cross-validation

```
k = 4
num_validation_samples = len(data) // k

np.random.shuffle(data)

validation_scores = []
for fold in range(k):
    validation_data = data[num_validation_samples * fold:
                           num_validation_samples * (fold + 1)]
    training_data = data[:num_validation_samples * fold] +
                    data[num_validation_samples * (fold + 1):]

    model = get_model()
    model.train(training_data)
    validation_score = model.evaluate(validation_data)
    validation_scores.append(validation_score)

validation_score = np.average(validation_scores)

model = get_model()
model.train(data)
test_score = model.evaluate(test_data)
```

Selects the validation-data partition

Uses the remainder of the data as training data. Note that the + operator is list concatenation, not summation.

Creates a brand-new instance of the model (untrained)

Validation score: average of the validation scores of the k folds

Trains the final model on all non-test data available

Splitting data: Iterated K-fold validation with shuffling

- Applying K -fold validation multiple times, shuffling the data every time before splitting it K ways.
- The final score is the average of the scores obtained at each run of K -fold validation.
- Note that you end up training and evaluating $P \times K$ models.

Splitting data: Keep in minds!

- Data representativeness – You want both your training set and test set to be representative of the data at hand
 - Usually, you should randomly shuffle your data before splitting.
- The arrow of time – if order matters, don't shuffle!
 - Tomorrow's weather, stock movements, and so on.
- Redundancy in your data – there should be NO overlap between training and test sets.

Data pre-processing (for neural networks)

- Data vectorization
- Value normalization

Data pre-processing: Vectorization

- All inputs and targets in a neural network must be tensors of floating-point data (or, in specific cases, tensors of integers).
- Whatever data you need to process—sound, images, text—you must first turn into tensors, a step called data vectorization.

Data pre-processing: Vectorization



			165	187	209	58	7
		14	125	233	201	98	159
253	144	120	251	41	147	204	
67	100	32	241	23	165	30	
209	118	124	27	59	201	79	
210	236	105	169	19	218	156	
35	178	199	197	4	14	218	
115	104	34	111	19	196		
32	69	231	203	74			

Data pre-processing: Value normalization

- In general, it is not safe to:
 - Feed into a neural network data that takes relatively large values
 - Data that is heterogeneous
 - For example, data where one feature is in the range 0-1 and another is in the range 100-200.
- Doing so can trigger large gradient updates that will prevent the network from converging.

Data pre-processing: Value normalization

- Take small values: Typically, most values should be in the 0-1 range.
- Be homogenous: All features should take values in roughly the same range.
- Stricter normalization practice:
 - Normalize each feature independently to have a mean of 0.
 - Normalize each feature independently to have a standard deviation of 1.

Data pre-processing: Missing values

- If you are missing a feature for a specific instance, replace it with 0
 - Unless 0 is already a meaningful value
- The network will learn that 0 means missing data, it should be ignored.

Feature engineering

The process of using your own knowledge about the data and about the machine-learning algorithm at hand (in this case, a neural network)...

to make the algorithm work better...

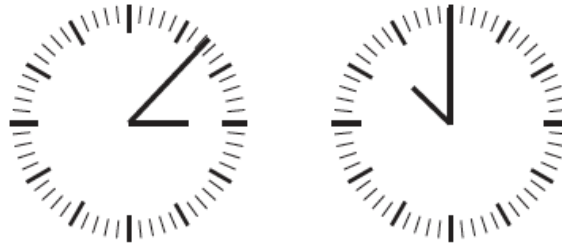
by applying hardcoded (non-learned) transformations to the data before it goes into the model.

- In many cases, it isn't reasonable to expect a machine learning model to be able to learn from completely arbitrary data.

Feature engineering: example

Problem: image of clock to time of the day

Raw data:
pixel grid



Better
features:
clock hands'
coordinates

$\{x1: 0.7,$
 $y1: 0.7\}$
 $\{x2: 0.5,$
 $y2: 0.0\}$

$\{x1: 0.0,$
 $y2: 1.0\}$
 $\{x2: -0.38,$
 $2: 0.32\}$

Even better
features:
angles of
clock hands

theta1: 45
theta2: 0

theta1: 90
theta2: 140

Feature engineering

- Critical for classical (shallow) algorithms
 - They didn't have hypothesis spaces rich enough to learn useful features by themselves.
- Modern deep learning removes the need for feature engineering
 - Neural networks are capable of automatically extracting useful features from raw data.
- But still:
 - Good features still allow you to solve problems more elegantly while using fewer resources.
 - Good features let you solve a problem with far less data.

Overfitting and underfitting

A fundamental issue in machine learning:
tension between optimization and generalization

- Underfit: at the beginning of training, the network hasn't yet modeled all relevant patterns in the training data
- Overfit: At some point, generalization stops improving, and validation metrics stall and then begin to degrade
 - It's beginning to learn patterns that are specific to the training data but that are misleading or irrelevant when it comes to new data

Prevent overfitting?

- More training data
- Not possible?
 - Limit the network
 - If a network can only afford to memorize a small number of patterns, the optimization process will force it to focus on the most prominent patterns, which have a better chance of generalizing well
 - This is called *regularization*

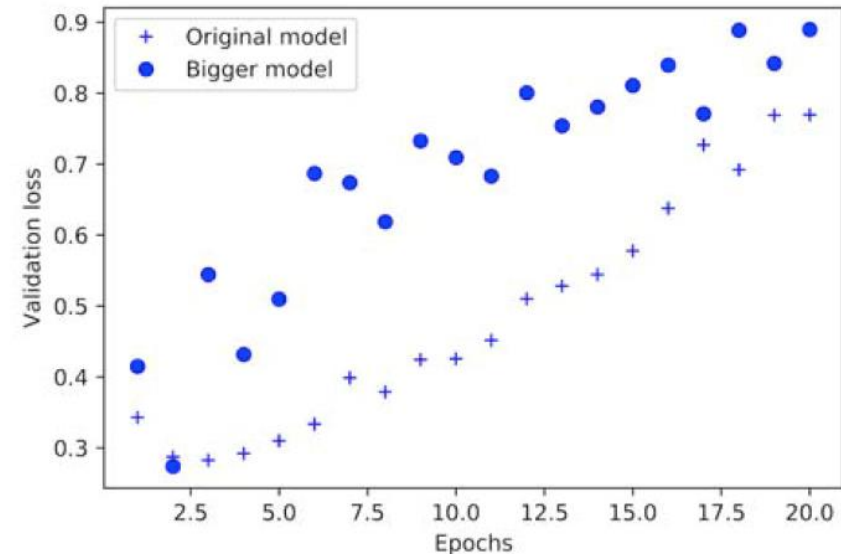
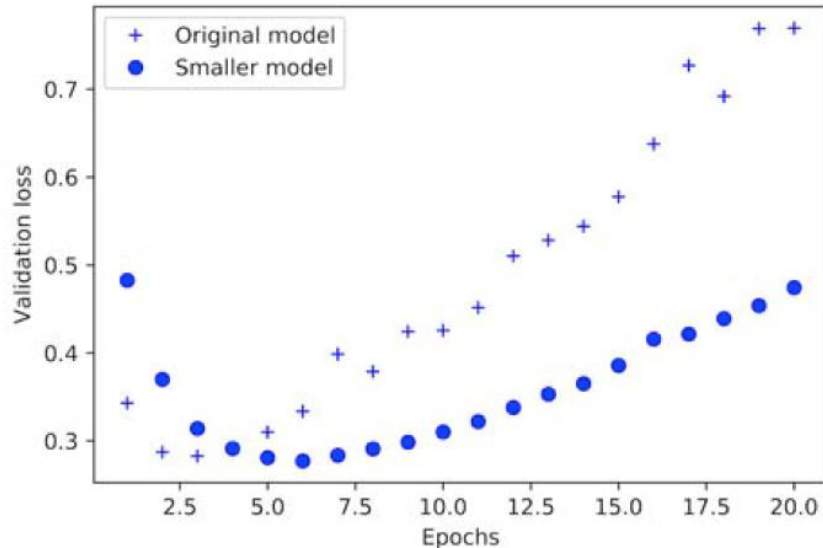
Reduce network size

Reduce the number of learnable parameters in the model (model's capacity)

- More parameters, higher *memorization capacity*
 - Models learns a dictionary-like mapping between training samples and their targets
- At the same time, have enough parameters not to underfit
 - Compromise between *too much capacity* and *not enough capacity*

How to decide on size then?!

- Unfortunately, there is no magical formula
- Start small, and increase the size until you see diminishing returns with regard to validation loss
- Movie review network:



Regularization

- Strategies or techniques which are used to **reduce the error** on *the test set* at an expense of *increased training error*
- An effective regularizer is said to be the one that makes a profitable trade by reducing variance significantly while not overly increasing the bias.

Weight regularization

- General rule:
 - Simpler models are less likely to overfit than complex ones.
- One way to make model *simple*:
 - Make the distribution of parameter values to have less entropy
 - By forcing weights to take small values
- Done by adding to the loss function, a cost associated with having large weights

Weight regularization

- L1

The cost added is proportional to the **absolute value of the weight coefficients**

- L2

- The cost added is proportional to the **square of the value of the weight coefficients**

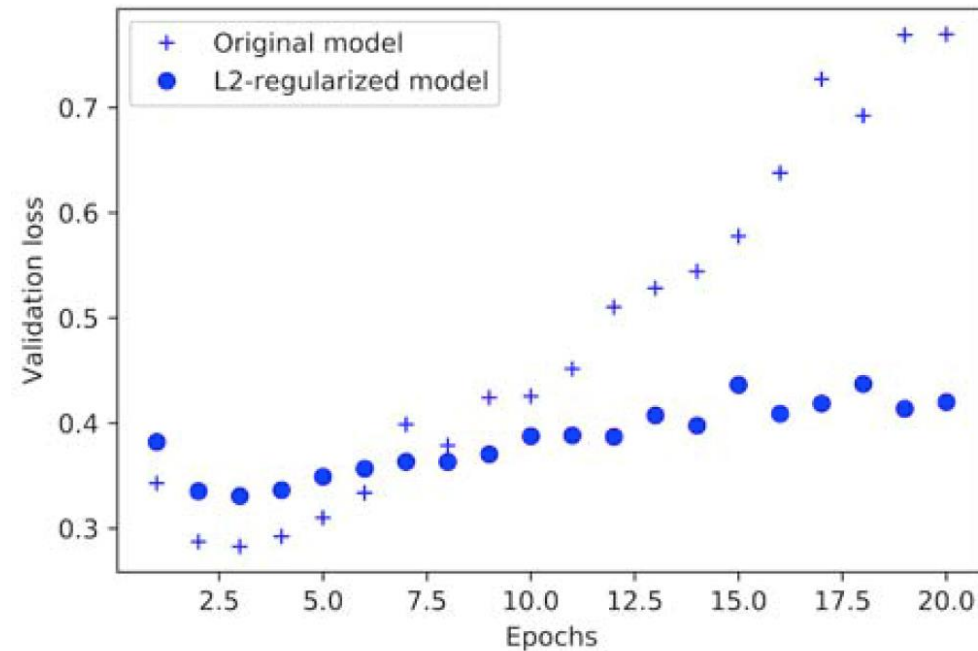
Movie review classification + L2 regularization

```
from keras import regularizers
model = models.Sequential()
model.add(layers.Dense(16, kernel_regularizer=regularizers.l2(0.001),
                        activation='relu', input_shape=(10000,)))
model.add(layers.Dense(16, kernel_regularizer=regularizers.l2(0.001),
                        activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
```

`l2(0.001)`

- Add `0.001 * weight_coefficient_value` to the total loss of the network

Movie review classification + L2 regularization



Dropout

- Developed by Geoff Hinton and his students at the University of Toronto
- One of the most effective and most commonly used regularization techniques for neural networks

Randomly drop out (setting to zero) a number of output features of the layer during training

Dropout

Randomly drop out (setting to zero) a number of output features of the layer during training

`[0.2, 0.5, 1.3, 0.8, 1.1]`



`[0, 0.5, 1.3, 0, 1.1]`

- *Dropout rate*: the fraction of the features that are zeroed out
 - Usually set between 0.2 and 0.5

Dropout

- At test time, no units are dropped out
- Instead, the layer's output values are scaled down by a factor equal to the dropout rate, to balance for the fact that more units are active than at training time

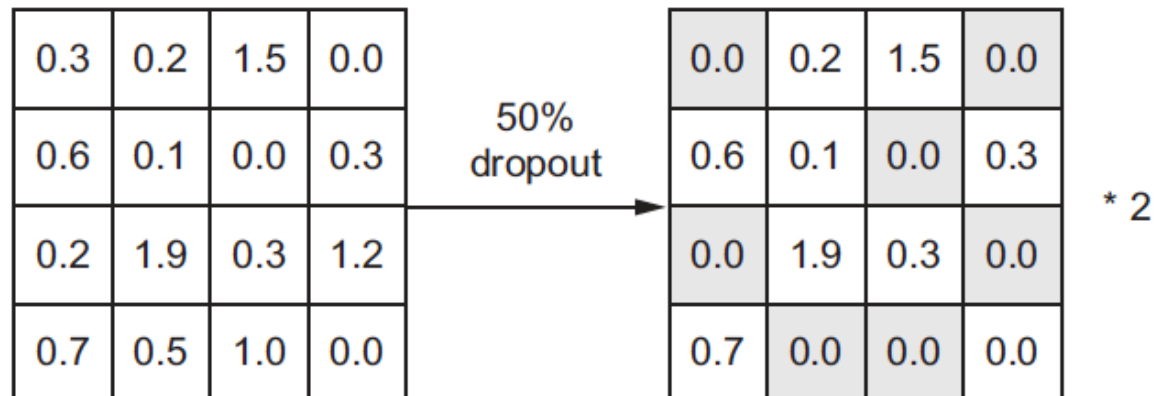


Figure 4.8 Dropout applied to an activation matrix at training time, with rescaling happening during training. At test time, the activation matrix is unchanged.

Dropout: why does it work?

“I went to my bank. The tellers kept changing and I asked one of them why. He said he didn’t know but they got moved around a lot. I figured it must be because it would require cooperation between employees to successfully defraud the bank. This made me realize that randomly removing a different subset of neurons on each example would prevent conspiracies and thus reduce overfitting.”

- Introducing noise in the output values of a layer can break up happenstance patterns that aren’t significant (“conspiracies”).



IMDB task: adding dropout

```
model = models.Sequential()  
model.add(layers.Dense(16, activation='relu', input_shape=(10000,)))  
model.add(layers.Dropout(0.5))  
model.add(layers.Dense(16, activation='relu'))  
model.add(layers.Dropout(0.5))  
model.add(layers.Dense(1, activation='sigmoid'))
```

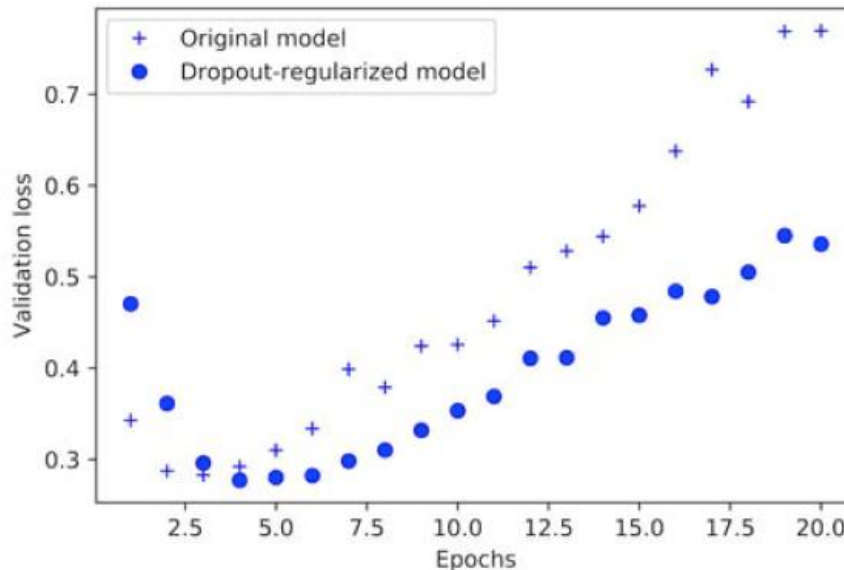


Figure 4.9 Effect of dropout on validation loss

Recap: prevent overfitting

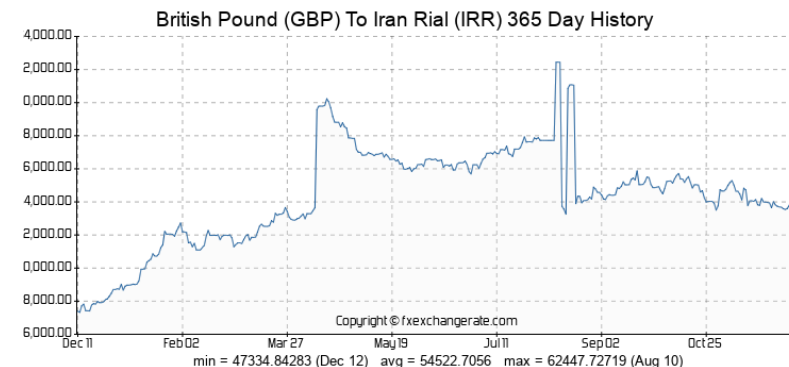
- Get more training data.
- Reduce the capacity of the network.
- Add weight regularization.
- Add dropout.

Workflow of machine learning

- Defining the problem and assembling a dataset
- Choosing a measure of success
- Deciding on an evaluation protocol
- Preparing your data
- Developing a model that does better than a baseline
- Scaling up: developing a model that overfits
- Regularizing your model and tuning your hyperparameters

Workflow: problem and dataset

- What will your input data be? What are you trying to predict?
 - Data availability is usually the limiting factor at this stage (unless you have the means to pay people to collect data for you).
- What type of problem are you facing? Is it binary classification? Multiclass classification? Regression? Reinforcement learning?
- Not all problems are solvable!



Workflow: measure of success

To achieve success, you must define what you mean by success!

- Accuracy?
- Precision and recall?
- Customer-retention rate?

Workflow: evaluation protocol

How you'll measure your current progress?

- Maintaining a hold-out validation set
 - The way to go when you have plenty of data
- Doing K-fold cross-validation
 - The right choice when you have too few samples for hold-out validation to be reliable
- Doing iterated K-fold validation
 - For performing highly accurate model evaluation when little data is available

Workflow: preparing your data

You should format your data in a way that can be fed into a machine-learning model

- As you saw previously, your data should be formatted as tensors.
- The values taken by these tensors should usually be scaled to small values: for example, in the $[-1, 1]$ range or $[0, 1]$ range.
- If different features take values in different ranges (heterogeneous data), then the data should be normalized.
- You may want to do some feature engineering, especially for small-data problems.

Workflow: do better than baseline

- Build a model that can beat a dumb baseline.
- We have two assumptions:
 - Your outputs can be predicted given your inputs.
 - The available data is sufficiently informative to learn the relationship between inputs and outputs.
- If things go well, we have three choices to make:
 - Last-layer activation
 - Loss function
 - Optimization configuration

Table 4.1 Choosing the right last-layer activation and loss function for your model

Problem type	Last-layer activation	Loss function
Binary classification	sigmoid	binary_crossentropy
Multiclass, single-label classification	softmax	categorical_crossentropy
Multiclass, multilabel classification	sigmoid	binary_crossentropy
Regression to arbitrary values	None	mse
Regression to values between 0 and 1	sigmoid	mse or binary_crossentropy

Workflow: overfit!

- Is your model sufficiently powerful?
- To figure out how big a model you'll need, you must develop a model that overfits. This is fairly easy:
 - Add layers.
 - Make the layers bigger.
 - Train for more epochs.

Workflow: regularize and tune

- You'll repeatedly modify your model, train it, evaluate on your validation data, ... until the model is as good as it can get
 - Add dropout.
 - Try different architectures: add or remove layers.
 - Add L1 and/or L2 regularization.
 - Try different hyperparameters (such as the number of units per layer or the learning rate of the optimizer) to find the optimal configuration.
 - Optionally, iterate on feature engineering: add new features, or remove features that don't seem to be informative.

A quick recap

A simple binary classification