What is the task definition of image classification? Explain at least 5 challenges and give examples. What is object detection and how does it differ from classification?

Task definition of Image Classification:

Given a finite set of class labels (e.g. {bird, cat, dog}).
Which class does the given image belong to?

Challenges of Image Classification:

- Pose and Viewpoint: How the object is oriented / located relative to the camera. (e.g. cat from behind, or from the side)
- Illumination: If we have poor illumination it is hard to distinguish objects from the background. Colors will change. Textures disappear. (e.g. cat in the dark with bright background).
- Deformation: Cats are very deformable. An object can take on different shapes if it is deformable. (e.g. cat is stretched out completely)
- Occlusion: Occlusion is when the object is only partly visible. A fraction of the object is hidden by another object. (e.g. cat hides behind curtain).
- Background: Contrast between object and background is poor. It is hard to distinguish object from background. (e.g. cat on carpet)
- Intraclass Variation: Variation of instances of a certain group/class. Objects of the same class that vary in color, shape. (e.g. cats of different color, shape, texture)

Task definition of Object Detection:

Given a finite set of class labels (e.g. {bird, cat, dog}).
We can have 0 or multiple relevant objects of these classes in an image.
We need to locate objects (of different classes) in an image.

Why do we need datasets? Explain the purpose of the three different subsets covered.

We need a labelled dataset, consisting of pairs of an image and a class label (e.g. cat, dog).
We need 3 disjoint datasets.

- Training set is used for training a model.
- Validation set is used for model validation, which is essentially hyperparameter optimization.
- Test set is for finally testing the model to estimate its effectivity on unseen data at the very end.
  If we would use the Test set during model training or validation, we have no data left to estimate a model's performance on unseen data.

Assume a company asks you to develop an application that is able to predict which kind of bird is depicted in a given image. Which kind of task is this? List and explain the individual steps you'd follow to solve this problem using deep learning.

This is the task of Image Classification, where these different kind of birds are your class labels and you want to know to which class a certain bird on an image belongs to.

For solving this task with Deep Learning:

1. We need a suitable large dataset of different kinds of birds.
2. I would use Transfer-Learning, because I assume that a dataset showing different kinds of birds will be quite small.
3. We use a pre-trained model that was trained on animal resp. bird images.
4. Cut-off part of the pre-trained network and replace with new Fully-Connected layers that you train.
5. The original pre-trained network is used as a feature extractor.
6. Additionally (e.g. if the dataset of the pre-trained model and the new dataset are not particularly similar), you can also fine-tune the whole network.

Generally for image classification in deep learning, we need these ingredients:

1. A suitable dataset

2. A network with a suitable final layer (dense/fully-connected layer).
3. A suitable loss function $L(\theta)$ (e.g. Cross-Entropy)
4. An algorithm for computing the gradient of the loss function $\nabla L(\theta)$ (e.g. backpropagation).
5. An algorithm for updating θ on this basis (e.g. ADAM / SGD).
6. A metric for estimating the model's performance (e.g. accuracy).

What is the motivation for solving vision tasks via machine learning? What is a machine learning algorithm and how are they used for solving image classification problems?

The question is, how to write an algorithm that detects features of an object in an image.
We cannot use traditional rule-based programming (if {} else {}) to capture the vast amount of features.

Machine Learning and especially Deep Learning can help us solve the image classification problem. Machine Learning algorithms are able to learn from data and in so its function improves with experience. ML algorithms learn relationships between object properties in an image and its corresponding class label.

General ML Pipeline is:

- Select a suitable ML algorithm
- Get a suitable dataset with images of object class instances you want to classify
- Split the dataset into training and validation set
- Extract discriminative features x for all class instances (features = properties of our samples)
- Determine the correct class label w of all class instances
- Show the ML algorithm for training (x, w) pairs (x = features, w = class label)
- The algorithm should learn a relationship between features and a class.
- Evaluate the algorithm's performance

Measuring or extracting good discriminative features is hard and this is where Deep Learning helps us by automating feature extraction.

What is a hyperparameter? Name at least 3 hyperparameters in the context of deep learning using convolutional neural networks. What is the purpose of hyperparameter selection, which search strategies exist, and how do they work?

A hyperparameter is set manually and controls the algorithm behavior. It is not a learned parameter. It is chosen experimentally or based on domain knowledge.

In convolutional neural networks hyperparameters can be the kernel dimensions (width, height), the learning rate, the batch size, the optimizer algorithm (e.g. ADAM, SGD), the number of hidden layers, the number of layer neurons, or activation functions (e.g. tanh, ReLu).

To find a good k for k-NN classifier, we

- Sample k from a reasonable range (min k=1, max k=#trainingSamples).
- Quantify k-NN algorithm performance on a validation set.
  It's important to use the validation set for hyperparameter selection, because we would not have any data left for estimating model performance on unseen data.
- Chose k with the highest validation accuracy (or per class scores).

Generally, we cannot test all hyperparameter combinations because testing hyperparameter combinations takes long, and increases with the number of hyperparameters.
Instead of doing an exhaustive search, the Grid Search and Random search strategies can be used.

How do these 2 hyperparameter search strategies work?

Given H hyperparameters with search intervals $I_1 \cdots I_H$

Grid search

- Sample uniformly from all I_h
- Test all combinations

Random search

- For j iterations, sample randomly from all I_h
- Test sample combination

Random search usually works better

- Wastes less time on unimportant hyperparameters

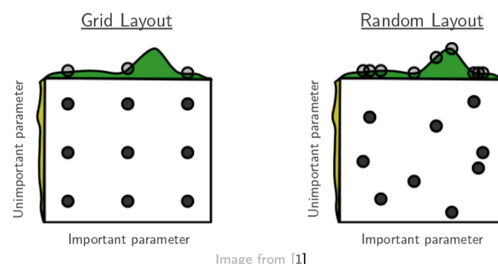
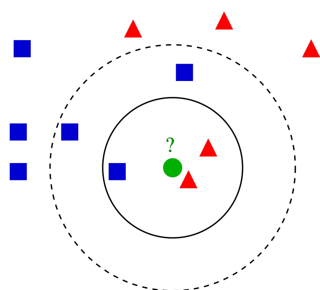


Image from [1]

How does the k nearest neighbor classifier work? Create a sketch for illustration, assuming a two-dimensional feature space and two different classes. Draw at least three training samples per class (must not lie on a line) as well as (roughly) the resulting decision boundaries. What are the limitations of this classifier?



The k-NN algorithm as classifier outputs a class membership. A new sample is classified by a majority vote of its k nearest neighbors. Usually k is odd, e.g. k=1, k=3, k=5.
In the illustration the green dot is a new observation that should be classified. If k=3 (solid line circle) it is assigned the red triangle because there are 2 red triangles and only 1 square inside the circle. If k=5 (dashed line circle) it is assigned to the blue squares.
k of k-NN is a hyperparameter and must be selected e.g. by hyperparameter optimization. k is not learned, but must be set manually.
Larger k values reduce effects of noise on the classification, but make boundaries between classes less distinct.
k-NN is still a poor image classifier because we used pixel values squeezed in a vector as features.
The k-NN algorithm is a lazy learner, because during training no model is built, but simply the training feature vectors and class labels are stored for the classification phase.
Prediction is slow but for small datasets.

What are class scores? What is the softmax function and why is it useful in this context?

We want to know how certain a classifier is, to react differently on this basis. (e.g. 91% banana, 9% child).
k-NN outputs the majority class for a new sample by looking at the k nearest neighbors.
We can adapt the classifier to predict class scores w as an int-valued vector with C dimensions (C = number of classes). An individual class score in w is then the frequency of a label among k closest samples. (e.g. w=(1,0,4) for classes red, green, blue with k=5).

These class scores are not optimal because these absolute frequencies depend on k.
We can normalize the class scores w to be a probability mass function. (w_c >= 0, sum of w_c = 1).

Most popular function for normalizing class scores is the softmax function.

$$\text{softmax}_c(\mathbf{w}) = \frac{\exp(w_c)}{\sum_c \exp(w_c)}$$

We obtain $\text{softmax}((1,0,4)) \approx (0.05, 0.02, 0.93)$

Negative values become positive.
Largest value is emphasized.
Small values are suppressed.
Softmax is not scale invariant.
Softmax maps from an output to something that can be considered a probability mass function.

The output of softmax encodes uncertainty, without having to also look on the other numbers.

Why do general machine learning algorithms (those expecting vector input) perform poorly on images? What is a feature, and what is the purpose of feature extraction? Explain the terms low-level feature and high-level feature.

Features are properties of our samples and ideally task-specific. (e.g. presence of whiskers for a cat and a nose for humans).
We use these features to make our algorithm learn a relationship to a corresponding class.
High-level features tell us something about the world depicted, that is the objects in our images (semantic information). We cannot reliable detect these high-level features manually, but Deep Learning can learn these features.
Low-level features are properties of the image itself and do not carry semantic information (e.g. an edge or a circle).

We can design manually general low-level features that are properties of the image itself.

For classical ML pixel values could be used as features by flattening the image to a vector, but then we would lose all positional, spatial data. A ML algorithm could only learn patterns of certain values in the vector. Pixel values are bad features.

Feature extractors extract properties of an image.

Other low-level features can be Brightness changes at certain scale and/or orientation.

Color histograms of the image can be used for measuring brightness.
Kernels can be used as feature extractors that applied (convolved) to an image respond to changes in the image. (e.g. Gabor, Sobel filters). These filters can be used for edge detection.

Other low-level features are Histogram of Oriented Gradients (HoG).

Adding those features obtained from the feature extractor to the image vector would lead to even more dimensions.
It is desirable to reduce the dimensionality D of feature vectors. Combat curse of dimensionality.
Goal is to preserve as much information as possible but limit dimensionality as much as possible.
Dimensionality reduction (e.g. PCA) can be used to reduce the dimensions.

These low-level features still do not improve image classification much, because they correspond to brightness changes in the image and do not carry semantic information.

List and explain the steps of the traditional image classification pipeline. What are the differences to a corresponding deep learning pipeline?

The traditional ML image classification pipeline is:

1. Extract low-level feature vectors.
   Use feature extractors to obtain low-level features, that is brightness changes at certain scale and/or orientation. (e.g. Kernels/Filters, HoG).
   Stack features in a vector.
2. Perform dimensionality reduction.
   By using dimensionality reduction vector dimensions are reduced. (e.g. use PCA)
3. Process using a generic ML algorithm like SVM.

In the DL pipeline, all the three steps are implicitly done by the DNN.

What is the definition of a parametric model? What do the parameters of such models control (what effect do they have?), and how are they set?

Let $\mathbf{x} \in \mathbb{R}^D$ be input and $\mathbf{w} \in \mathbb{R}^T$ be output

- Last time we used C instead of T

A model describes family of functions from $\mathbf{x}$ to $\mathbf{w}$

- Particular function $f : \mathbf{x} \mapsto \mathbf{w}$ learned during training

Model defines the hypothesis space

- Set of functions allowed as solution
- Extending family increases the model capacity (flexibility)

x is input vector, w is output vector.

A model f describes a family of functions that map from our input space D to the output space T. By defining a model we define the capabilities and boundaries of our classifier. But we do not say what is the actual mapping.

Training entails finding from this whole set of possible mappings a good mapping from the input to the output space based on the data that we have. The actual mapping of the model is then learned by training from the data.

The model defines the hypothesis space, that is the family of functions allowed and so the model capacity and flexibility.

In contrast to k-NN, with parametric models we can discard the training set after training, because all the information we have extracted from the training set is encoded in the parameters vector θ.

In parametric models f depends on parameters $\boldsymbol{\theta}$

- We write $\mathbf{w} = f(\mathbf{x}; \boldsymbol{\theta})$
- Training entails finding good parameters
- Training set can be discarded after training

DL models can have millions of parameters

What is a linear model, which types of parameters does it have, and what do they specify? Draw a sketch assuming two-dimensional feature space and three different classes. Draw a few samples per class so that the classes are linearly separable. Draw the decision boundaries a linear classifier might learn and explain how the individual boundaries are related to the classifier output (no need to calculate anything).

The most basic example are linear models

Hypothesis space comprises linear functions from $\mathbb{R}^D$ to $\mathbb{R}^T$

- Formally $\mathbf{w} = f(\mathbf{x}; \boldsymbol{\theta}) = \mathbf{W}\mathbf{x} + \mathbf{b}$ with $\boldsymbol{\theta} = (\mathbf{W}, \mathbf{b})$

Parameters

- $\mathbf{W} \in \mathbb{R}^{T \times D}$ is called weight matrix
- $\mathbf{b} \in \mathbb{R}^T$ is called bias vector

A liner model is a parametric model and its hypothesis space comprises linear functions from the input to the output space.

The parameters θ that we have to learn are (W, b). W is a weight matrix, b is a bias vector.

W specifies the slope (orientation) and b specifies the intercept (offset from 0) in f(x; θ) = Wx + b

Example of mapping an image to class scores:

Assume an image is a vector of 4 pixels (x_i). Multiply weight matrix W with input vector x_i. Sum with bias vector.

Result values can also be negativ. We can use softmax function to get a probability mass function.

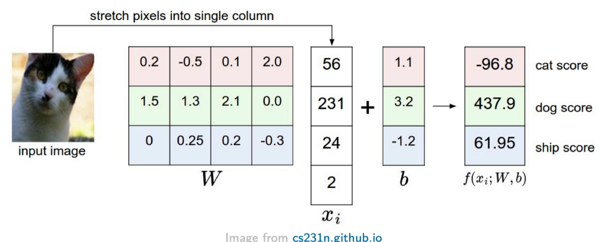
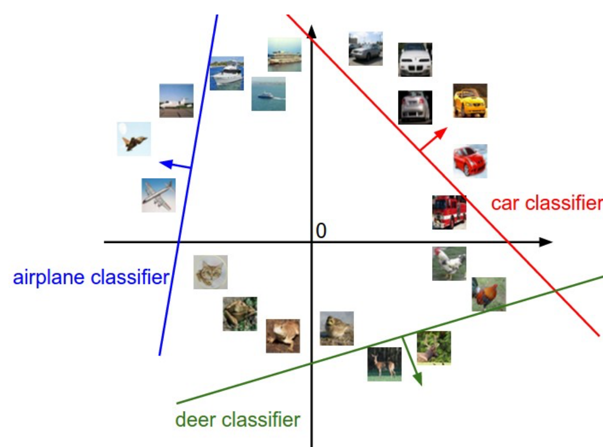


Image from cs231n.github.io

Individual class scores (e.g. cat score) only depends on one row in the weight matrix and bias vector. So, we can consider a linear classifier with T classes as T independent linear classifiers.

Geometric Interpretation of Linear Models for Classification:

Analogy of images as high-dimensional points. Since the images are stretched into high-dimensional column vectors, we can interpret each image as a single point in this space (e.g. each image in CIFAR-10 is a point in 3072-dimensional space of 32x32x3 pixels). Analogously, the entire dataset is a (labeled) set of points.
Since we defined the score of each class as a weighted sum of all image pixels, each class score is a linear function over this space. We cannot visualize 3072-dimensional spaces, but if we imagine squashing all those dimensions into only two dimensions, then we can try to visualize what the classifier might be doing:



Cartoon representation of the image space, where each image is a single point, and three classifiers are visualized. Using the example of the car classifier (in red), the red line shows all points in the space that get a score of zero for the car class. The red arrow shows the direction of increase, so all points to the right of the red line have positive (and linearly increasing) scores, and all points to the left have a negative (and linearly decreasing) scores. https://cs231n.github.io/linear-classify/

$\mathbf{x}$ is on positive side if $w_c = \mathbf{w}_c\mathbf{x} + b_c \geq 0$

- $\mathbf{w}_c$ is row c of $\mathbf{W}$

Class score increases with its distance to the hyperplane. High class scores indicate high certainty.

What is the purpose of a loss function? What does the cross-entropy measure? Which criteria must the ground-truth labels and predicted class-scores fulfill to support the cross-entropy loss, and how is this ensured?

For training any parametric model we need
- **A loss function (tells us how good classifier is on the training data)**
- An optimization algorithm (based on the output of the loss function and current parameters we adapt our parameters to improve the model performance.)

With our linear model we obtained class scores. In order to measure our unhappiness with outcomes we use a loss function (=cost function/objective).

A loss function $L(\boldsymbol{\theta})$ (or cost or objective function)
- Measures performance of $f(\cdot\,;\boldsymbol{\theta})$ (lower loss is better)
- On some (training) dataset $\mathcal{D} = \{(\mathbf{x}_s, \mathbf{w}_s)\}_{s=1}^{S}$
- With respect to parameters $\boldsymbol{\theta}$

Choice of L depends on task
- Most popular classification loss is cross-entropy

Loss function only depends on parameters θ, but not on our features x.

Entropy: Given random variable with mass function u.
u_t is between 0 and 1. So log(u_t) is negativ or 0. After summing and negating, entropy is 0 or positive.
The higher the entropy the more information is gained from sampling from this random variable.
Assume u_1 = 1, then u_1*log(u_1) = 1*0 = 0, all other u_t must be 0, then entropy is 0 (no information gain, no randomness).

Recall from information theory that
- Given a mass function $\mathbf{u} = (u_1, \dots, u_T)$
- The entropy of $\mathbf{u}$ is $H(\mathbf{u}) = -\sum_{t=1}^{T} u_t \log_b u_t$
- In information theory $b = 2$, here it does not matter

$H(\mathbf{u})$ is average information gain when sampling from $\mathbf{u}$

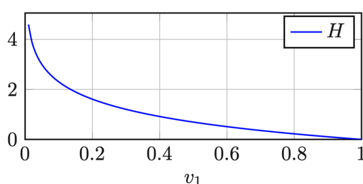
Given two probability mass functions $\mathbf{u}$ and $\mathbf{v}$ in $\mathbb{R}^T$
- $\mathbf{u} = (u_1, \dots, u_T)$ and $\mathbf{v} = (v_1, \dots, v_T)$

The cross-entropy between $\mathbf{u}$ and $\mathbf{v}$ is

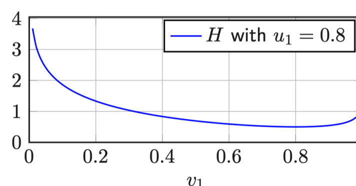
$$H(\mathbf{u}, \mathbf{v}) = -\sum_{t=1}^{T} u_t \ln v_t$$

Cross-Entropy: is defined the same as Entropy, but uses 2 probability mass functions u and v.
T is number of classes (e.g. cat, dog).
Plot of Cross-Entropy for T=2,   u_1=1, u_2=0
Plot shows how Cross-Entropy H changes when varying the 2nd probability mass function v (with v1).
Cross-Entropy H decreases to 0, if u and v are the same (i.e. u_1=v_1=1).
If u and v are the most dissimilar, u_1=1 but v_1=0, then the Cross-Entropy H is the largest.
The Cross-Entropy H measures the dissimilarity between u and v.
Large dissimilarity between u and v-> Large Cross-Entropy H
Similar u and v -> Cross-Entropy H is 0

Example with $T = 2$ and $u_1 = 1$
- The more different $\mathbf{u}$ and $\mathbf{v}$ the higher H
- H measures the dissimilarity between $\mathbf{u}$ and $\mathbf{v}$



Note that H can reach 0 only if any $u_t = 1$
- In general $H(\mathbf{u}, \mathbf{v}) =$ entropy of $\mathbf{u}$ if $\mathbf{u} = \mathbf{v}$



Cross-Entropy works good as loss function for our classifier. We want a function that is minimal if our classifier performs very well and we want our classifier to predict class scores that are probabilistic.

We can set our vector u based on the labels we have in our dataset.
Then vector v would be the prediction of our classifier after we apply a softmax.
So, our loss function is the cross entropy, and we get a 0 loss if our prediction matches the class label input.
But: **Cross-Entropy H can only reach 0, if any u_t = 1**!

In practice, we have to set *u* to 0 everywhere, except for the index of the **correct class** to **1**.
This is the necessary **criteria**, that **ensures** that the **cross-entropy can reach 0**.

To utilize the cross-entropy for classifier training we

- Let $\mathbf{u}$ encode the ground-truth label, $u_c = 1$
- Let $\mathbf{v}$ be the predicted softmax class scores

H measures how dissimilar true and predicted probabilities are

- How well the classifier performs on a single sample

On this basis we calculate the cross-entropy loss on $\mathcal{D}$ as

$$L(\boldsymbol{\theta}) = \frac{1}{S}\sum_{s=1}^{S} H(\mathbf{w}_s, \text{softmax}(f(\mathbf{x}_s; \boldsymbol{\theta})))$$

Average cross-entropy over some dataset $\mathcal{D}$

- We will use (subsets of) the training set as $\mathcal{D}$

Finally, to get a measure of the **loss over the whole training set** D, we use the **average Cross-Entropy Loss** over all training samples S.

w_s is the ground-truth vector from our training set
H is the Cross-Entropy
f(x_s; θ) is the output of our classifier
x_s is the input sample
softmax is used to get class scores as probability mass function (values between 0 and 1, sum to 1)

Models trained with this loss are called softmax classifiers

- Also called logistic regression if $T = 2$

Classifiers learn to predict probabilities per class label

What is the purpose of optimization in the context of machine learning? How does the gradient descent algorithm work? What is the gradient of a function? Explain the terms learning rate and step size.

We measure with the Loss function L(θ) how good our classifier is. (Cross-Entropy Loss)
We need a way to adapt our parameters on this basis so that it becomes better over time.

We need to minimize the loss function L(θ). This maximizes the training set classification performance. Even if our actual model mapping is linear, L(θ) is not linear in θ. So we need a non-linear optimization algorithm to minimize L(θ).

Gradient Descent is an iterative optimization algorithm. In every iteration we take a step.

Iterative Optimization algorithm

In every iteration we

- Compute gradient $\boldsymbol{\theta}' = \nabla L(\boldsymbol{\theta})$
- Update parameters $\boldsymbol{\theta} = \boldsymbol{\theta} - \alpha\boldsymbol{\theta}'$

Hyperparameter $\alpha > 0$ is called learning rate

- Final step size is $\alpha \|\boldsymbol{\theta}'\|$

The gradient points in the direction of the greatest increase of our loss function from where we are right now (θ).
The opposite direction of the gradient is the direction of the steepest decrease.

The step size, that is how far we go, depends on the magnitude of the gradient and the learning rate α.
That is if the gradient is steeper we go further or if the learning rate is higher.
The step size is the learning rate times the magnitude of the gradient.

Let $f(x_1, \ldots, x_n)$ be a differentiable, real-valued function

The partial derivative f_{x_i} of f with respect to x_i

- Is also a real-valued function $f_{x_i}(x_1, \ldots, x_n)$

$f_{x_i}(\mathbf{x})$ encodes

- How fast f changes with argument x_i
- At some location $\mathbf{x}$

Gradient ∇f is vector of all partial derivatives of f

- $\nabla f = (f_{x_1}, \ldots, f_{x_n})$
- Vector-valued function $\mathbb{R}^n \mapsto \mathbb{R}^n$

$\nabla f(\mathbf{x}) = (f_{x_1}(\mathbf{x}), \ldots, f_{x_n}(\mathbf{x}))$ encodes

- How fast f changes with all arguments $x_1 \cdots x_n$
- At some location $\mathbf{x}$

What is the Gradient of a function? Function f(x_1, …, x_n) is our loss function here and x is θ.
The partial derivative f_xi of f with respect to xi tells us how fast f changes if we change parameter xi at the current location x.

The Gradient ∇ f is the vector of all partial derivatives of f and tells us how fast f changes with all the arguments x_1 … x_n at the current location x.

$\nabla f(\mathbf{x})$ specifies how f changes locally at $\mathbf{x}$

- Points in direction of greatest increase
- Norm equals magnitude of increase

Exactly what we need to minimize L

- Compute direction of greatest increase $\nabla L(\boldsymbol{\theta})$
- Move in the opposite direction

We stop if $\nabla L(\boldsymbol{\theta}) \approx \mathbf{0}$ (if norm is close to 0)

- No information where to go next
- L is flat at current location
- The case if we are at $\hat{\boldsymbol{\theta}}$ (but not only then)

Simple and general algorithm

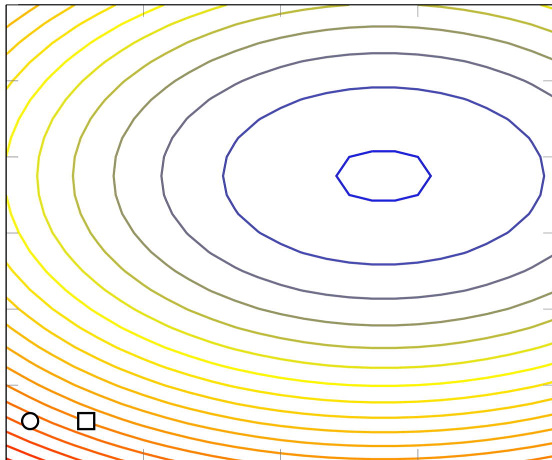
- Requires only that f is differentiable, real-valued
- Efficient (requires only first derivatives)

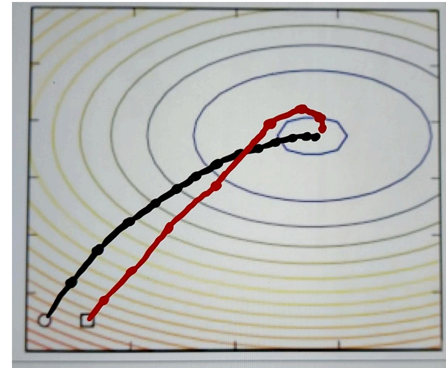
Several (possible) limitations

- Performs poorly for many f
- But works remarkable well with DL models

Performs well for convex functions, but poorly for many other functions. This limitation can be resolved with tweaks to work remarkable well.

Consider the following contour plot of a function with two parameters. How might gradient descent proceed in this case, assuming the circle in the bottom-left corner as the starting point? Mark the individual steps and connect them using lines. Give a brief explanation of momentum. How would momentum affect the training progress in the example case? Mark the individual steps gradient descent with momentum might take, assuming the bottom-left square as the starting point?





On a contour plot points that lie on a line have the same value.

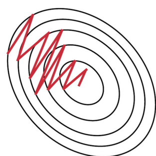
Without momentum: Gradient Descent oscillates more and takes more iterations.
With momentum: Gradient Descent increases the step size/velocity if successive gradients are similar and may overshoot due to its dynamic increase in step size.

Gradient Descent **Limitation**: *Poorly Conditioned Hessian* (Optimization surfaces that look similar like a canyon)
Gradient Descent wastes time jumping between canyon walls. It jump from one side of the wall to the other and back again. Only a very small part of the step size is in the correct direction to the center. This does not mean that Gradient Descent is stuck, but gets very slow.

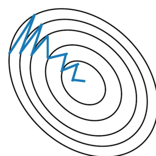
A solution for the Gradient Descent limitation of many oscillations is Momentum.

Momentum improves the speed of convergence by

- Dampening oscillations (reduces jumping)
- Increasing the step size dynamically (decreases how long optimization takes in regions of the optimization surface where the gradient does not change a lot)



Stochastic Gradient Descent **withhout** Momentum



Stochastic Gradient Descent **with** Momentum

Idea of increasing step size dynamically:
If the gradient does not change a lot over multiple iterations, the algorithm gains confidence that this direction is the right one and increases velocity resp. takes bigger steps.

Gradient Descent without Momentum forgets about steps taken in the last iterations, but with Momentum we integrate information of where we came from in the last couple iterations of the optimization process.

Remembering last taken steps can be implemented using the exponential moving average of gradients for direction v. The influence of older gradients decays exponentially.

Gradient Descent with Momentum:

Iteration of gradient descent with momentum

- Update velocity $\mathbf{v} = \beta\mathbf{v} - \alpha\nabla L(\boldsymbol{\theta})$
- Update parameters $\boldsymbol{\theta} = \boldsymbol{\theta} + \mathbf{v}$

Hyperparameter $\beta \in [0, 1)$ called momentum

- Defines decay speed and maximum step size

Compared to vanilla Gradient Descent the part *β v* is new.
v is the information of the previous **velocity** / step size / direction.
With the **Momentum hyperparameter *β*** we can define how much of the previous velocity v we want to reuse.
If *β* = 0 we end up with normal Gradient Descent.

$\mathbf{v}$ builds up momentum if successive gradients are similar

- Improves speed of convergence

Maximum step size is $\alpha\|\mathbf{g}\|/(1 - \beta)$

- Assuming the gradient is always $\mathbf{g}$
- At $\beta = 0.9$ maximum increase by factor of 10

If the Gradient is always quite the same we build up speed. The gradients will add up over time.

A slight adjustment of Gradient Descent with Momentum is **Nesterov Momentum**.

Evaluate gradient at $\boldsymbol{\theta} + \mathbf{v}$ instead of $\boldsymbol{\theta}$

Iteration of gradient descent with Nesterov momentum

- Update velocity $\mathbf{v} = \beta\mathbf{v} - \alpha\nabla L(\boldsymbol{\theta} + \mathbf{v})$
- Update parameters $\boldsymbol{\theta} = \boldsymbol{\theta} + \mathbf{v}$

Often works better than standard momentum

With momentum we already have the information where we came from. It's quite unlikely that the Gradient will change completely in the next iterations. What Nesterov Momentum does is, it **looks ahead** a bit. It computes the Gradient of the Loss function of where we will probably go. That is it **computes the Gradient of the Loss function at θ + v** instead of θ. Everything else is the same like in Gradient Descent with Momentum.
We jump to the position using the last velocity v and compute the Gradient there.

What is the difference between a local and a global optimum? Draw a sketch that illustrates the difference. Are local minima a problem in deep learning? Why (not)? What is momentum and why is it beneficial?

Algorithm stops if $\nabla L(\boldsymbol{\theta}) \approx \mathbf{0}$
- Applies to all critical points, not only minimum
- Should stop only at minimum

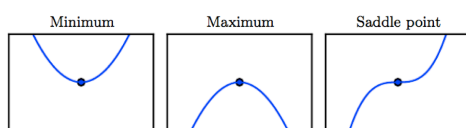


Image from [1]

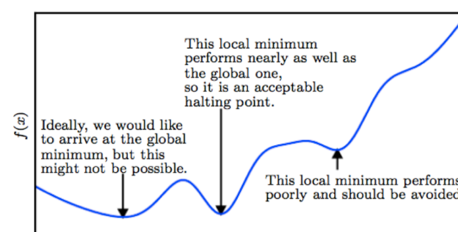


Image from [1]

Due to the nature of Gradient Descent, the algorithm stops if the Gradient of the loss is 0. There we can no longer subtract anything from the current parameters — we are basically stuck.

The gradient of the loss function is 0 at a minimum, maximum or saddle point.
The global minimum is where the loss function is the lowest and an optimum.
A local minimum is where the first derivative is 0, but does not have the lowest minimum value.

For example in a convex function, we have a single minimum which is the global minimum. But in practice the optimization surface has many local minima.

A problem with vanilla Gradient Descent is that we may jump over a local minimum if the learning rate is too high. But we may also move and get stuck in a local minimum which should be avoided.

Sometimes local minimum perform nearly as well as the global minimum.
Gradient Descent usually finds only a local minimum.

Algorithm stops at first minimum as $\nabla L(\boldsymbol{\theta}) \approx \mathbf{0}$
- But L generally has several local minima
- Algorithm usually finds only a local minimum

For loss functions of DL models evaluated on minibatches (below)
- Local minima are usually close to global minimum
- Optimization does not come close to critical points
- We can escape from critical points

Local minima are not a problem in deep learning. One reasons is that we evaluate the loss function not on the whole dataset, but on minibatches. Assume our loss is 0 for one minibatch, then our parameters stay the same, but if when the loss of the next minibatch is not 0, we make progress again. So basically we can escape local minima by evaluating our loss function on minibatches.
Another reason is, that in practice the loss is far away from a zero gradient anyway.
Decreasing the minibatch size S, also causes more noisy gradient estimates, and hence, gives Gradient Descent the ability to escape local minima.

Gradient Descent does not necessarily skip local minima and converge to global minima. Rather, it is very likely to skip small bumps / local minima and more likely to spend time around big bumps.

Momentum improves the speed of convergence by
- Dampening oscillations (reduces jumping)
- Increasing the step size dynamically (decreases how long optimization takes in regions of the optimization surface where the gradient does not change a lot)

Idea of increasing step size dynamically:
If the gradient does not change a lot over multiple iterations, the algorithm gains confidence that this direction is the right one and increases velocity resp. takes bigger steps.

The step size depends on the learning rate and also on the magnitude of the gradient. The learning rate is global and finding a good one is hard. Gradient Descent depends a lot on the learning rate selection.

Explain the differences between batch, minibatch, and stochastic gradient descent. Which version is most commonly used in deep learning and why? What effects has the minibatch size? Write pseudo-code that illustrates the overall structure of minibatch-based training and validation, continuing below the following line. A high-level overview is sufficient, no need to use math.

```
while epoch <= MAX_EPOCHS:
        shuffle training set
        for each minibatch b in training set:
                get predictions for b
                compute loss and gradient of loss
                update weights

for each minibatch b in validation set:
        get predictions for b
        compute loss
        add loss to validation metric calculator

print average validation loss
```

Batch Gradient Descent: Uses the **whole training set** at once to compute the gradient. We compute the average loss that is over all training samples. **Scales very poorly** with increasing number of samples. (assume we have millions of samples). The time complexity per iteration increases linearly with the sample size S and is problematic with large S.

Minibatch Gradient Descent: Processes the whole training set in **minibatches of size S** (one per iteration). This allows to estimate the loss of the whole training set with subsets of samples. One full run through the training set is called an **epoch**. It's important to sample minibatches randomly to break (possible) ordering in the dataset. For example shuffle training set once or before every epoch.

Stochastic Gradient Descent (SGD): Is Minibatch Gradient Descent with a minibatch size of **S=1** (i.e. 1 sample per minibatch). But in other literature **Minibatch Gradient Descent** is also often called SGD even if **S>1**.

In Deep Learning most commonly used is Minibatch Gradient Descent with a batch size of 32, 64, 128 or 256 (2^n for efficiency, data parallelism).

What effects has the minibatch size?

Decreasing the minibatch size S, also decreases

- the computation time per iteration
- the GPU memory required
- the accuracy of the gradient estimate

Decreasing the minibatch size S, also causes more noisy gradient estimates, and hence, gives Gradient Descent the ability to escape local minima.
The loss function is no longer only dependent on the parameters, but also on the data which changes over time.

How does the Adam optimizer differ from normal gradient descent with momentum? What are the advantages of the former over the latter?

Gradient Descent has as disadvantage the global learning rate which is very hard to set.

Adam stands for Adaptive Moment Estimation.
It is similar to gradient descent with momentum, but it has adaptive learning rates, that is it changes the learning rates itself, based on how the optimization surface looks like and it does that for every parameter independently.

In general optimization becomes faster, because Adam will speed up a lot if the optimization surface is kind of flat. Setting the learning rate becomes less critical because it is basically just one factor and Adam's adaptiveness.

What is the definition of a feedforward neural network? Which types of units do such networks have? Draw a graph of such a network. How can linear models be implemented using neural networks?

Definition of a feedforward neural network:
A neural network is a directed computational graph. Vertices (neurons or units) are scalar functions of the input. Edges define the data flow.
Neurons operate on (subsets of) x and/or neuron output.

Neurons at same level in hierarchy form a layer. Neurons in same layer usually perform same kind of operation.

Feedforward NN are acyclic. In contrast, NNs with cycles are called Recurrent NNs (RNN).

Graph of a feedforward neural network — on the left ->

A Neural Network (NN)

- Is a directed computational graph
- Vertices (neurons or units) are *scalar* functions of input
- Edges define data flow

And thus a function $f : \mathbf{x} \in \mathbb{R}^D \mapsto \mathbf{w} \in \mathbb{R}^T$

- That is composed of other functions (neurons)
- Neurons operate on (subsets of) $\mathbf{x}$ and/or neuron output

D input units (x) and T output units (f)

Flexible number of hidden units (h, g)

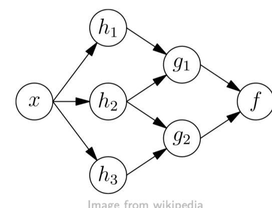


Image from wikipedia

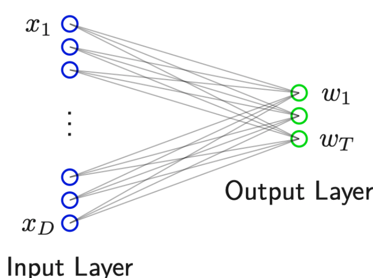
Linear models be implemented using neural networks like this:

Recall that in linear models $\mathbf{w} = \mathbf{W}\mathbf{x} + \mathbf{b}$

- $\mathbf{x} \in \mathbb{R}^D$, $\mathbf{w} \in \mathbb{R}^T$, $\mathbf{W} = [\mathbf{w}_1; \ldots; \mathbf{w}_T]$

To obtain the corresponding NN we define

- One input layer with D neurons
- One output layer with T neurons
- Each output neuron as $n_t(\mathbf{x}) = \mathbf{w}_t\mathbf{x} + b_t$



Each neuron in the output layer computes a linear function. Such neurons/layers are called linear.

Output neurons are connected to all neurons in the previous layer. Such neurons/layers are called fully-connected or dense.

What is the definition of a multilayer perceptron? What operation do (non-input) units perform? What is an activation function and which functions are common? Draw a sketch that shows the layers of such networks and how the individual units are connected. Why are multilayer perceptrons not suitable for deep learning for image analysis?

Definition of a multilayer perceptron (MLP):

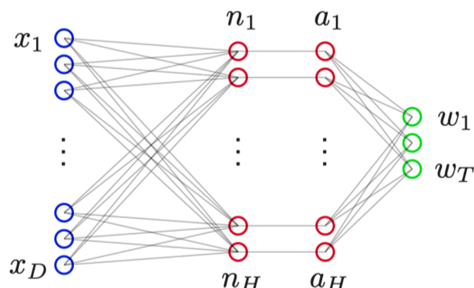
Linear NNs lack capacity (f is linear)

To increase the capacity we add

- A linear hidden layer with H neurons $v_h = n_h(\mathbf{x})$
- A layer with H non-linear activation functions $a_h(v_h)$

Common activation functions

- $a_h(v_h) = \max(0, v_h)$ (Rectified Linear Unit / ReLU)
- $a_h(v_h) = \tanh(v_h)$
- $a_h(v_h) = 1/(1 + \exp(-v_h))$ (logistic sigmoid)



Non-input hidden- units perform a linear scalar function, followed by an application of a non-linear activation function, like ReLu.
A non-linear activation function overcomes the limited capacity of linear NNs. If we introduce non-linear activation functions into a NN, our model mapping becomes non-linear.

The **representational capacity** of MLPs depends on these hyperparameters:

- Number of hidden units H
- Type of activation functions
- Number of hidden layers (depth)

The question is, can we make the number of hidden units and hidden layers large enough to solve our classification problem?
Assume image size of 32x32, flatten to 3072 input dimensions.
The parameter dimensions dim(θ) increases quickly with the number of hidden units H (e.g. H=500, dim(θ) ~ 1.5M) and network depth. In practice we even have larger images.

Having a flexible MLP model is not sufficient. Pixel values as features or low-level features perform poor. We want task-specific high-level features, but we cannot design such feature extractors.

What is the motivation for and purpose of representation learning? How is deep learning related to representation learning, and what is its definition?

We cannot design high-level feature extractors, but we can learn them. This is called Representation Learning.

Deep Learning is Representation Learning with DNNs.

Making the MLP deeper by adding hidden layers.
Gain ability to learn features in hierarchical way (input pixels, edges, corners/contours, object parts, object identity).
Later features build upon earlier (simpler) ones.

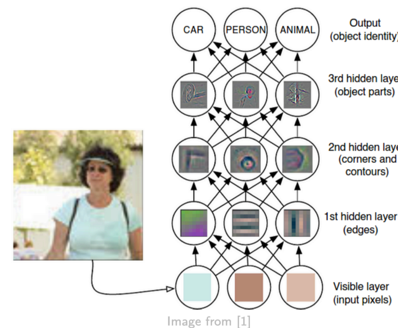


Image from [1]

What is the receptive field of a neuron? Assume a network consisting of two convolutional layers with 3 × 3 connectivity followed by a 2 × 2 max-pooling with stride 2 and again two convolutional layers with 3 × 3 connectivity. What is the receptive field of neurons in the final convolutional layer? How does the receptive field affect feature extraction?

The receptive field is the field of neurons of the input that one neurons "sees", so the input neurons that are indirectly connected and thus can effect the output of that neuron.
From right to left: 1 -> 3 -> 5 -> 10 -> 12 -> 14
How affect: the feature can only be from the receptive field so at the end of the network the output neurons always have to have the whole input as receptive field. The bigger the receptive field, the more higher-level the feature can be.

Neurons see small part of previous layer (sparse connectivity)

- But larger input region (receptive field) as depth increases

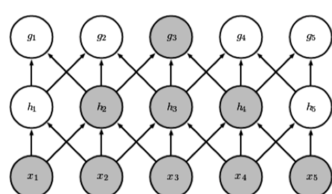


Image from [1]

What is the purpose of convolutional layers and which operation do they compute? What are the two key differences to linear layers, and what motivates these differences with respect to image analysis? What's the most popular activation function for these layers? Draw a graph of this function.

Purpose:
We want to make use of spatial structure of images instead of flattening images to vectors.
Spatially close pixels are highly correlated, others not. That is, nearby pixels correspond to the same object or part.
A convolutional layer computes features from spatially close pixels.
A conv layer connects spatially close neurons in a sparse way, instead of dense connectivity, like in MLP.
A convolution layer has a configurable connectivity c (e.g. 3x3). The connectivity along the channel dimension is usually the number of output channels of the previous layer (e.g. 3 for RGB for input neurons, leading to 3x3x3).

The transformation of a conv layer is a convolution of the input with kernel A_l, followed by an additive bias b_l.

Neurons in a convolutional layer detect features and respond to local structures similar to the kernel.

Instead of H hidden layer neurons in the linear layer, a conv layer has a neuron grid of W x H x C, with C feature maps of size W x H.
Only neurons in the same feature map (channel) share

Neurons in layer l compute $n_h(\mathbf{X}_h) = \mathbf{A}_l \cdot \mathbf{X}_h + b_l$

- $\mathbf{A}_l \cdot \mathbf{X}_h$ is a linear combination (like before)
- $\mathbf{A}_l$ is identical for all neurons in layer

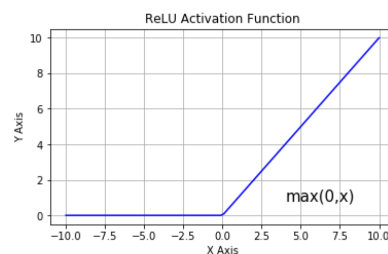
The overall transformation of the layer is thus

- A convolution of the input with kernel $\mathbf{A}_l$
- Followed by an additive bias b_l

Such layers are thus called convolutional (conv) layers

- Fundamental DL layer

parameters. A conv layer thus can learn C different features.



The most common **activation function** after a conv layer is **ReLu**.

2 key differences:
In a linear fully-connected layer we have to learn much much more **weights** (hidden units * dimensions of last layer), compared to a conv layer, where the weights are not dependent on the input size, but on the connectivity c, channel dimensions of last and current layer.

Further, by stacking conv layers, a layer l learns to combine features from layer l-1 to new ones, which desired for learning **hierarchical** representations (from local features to global features).

Number of weights $\mathbf{A}_l$ depends only on c, C_{l-1}, C_l
- $c = 3,\ C_{l-1} = 3,\ C_l = 32 \implies 864$ weights
- $c = 3,\ C_{l-1} = 32,\ C_l = 64 \implies 18.5$k weights

Way fewer parameters than with linear layers
- Can stack several conv layers
- Layer l learns to combine layer $l-1$ features to new ones
- This is exactly the hierarchical approach we desire

3 * 3 * 3 * 32 = 864
3 * 3 * 32 * 64 = 18,432

What are convolutional layers? Assuming an input shape of W × H × D, how many weight and bias parameters does a convolutional layer with a 3 × 3 kernel, f feature maps, stride 1, and padding have? What are feature maps and why are they needed?

Convolutional layers are the layers where filters are applied to the original image, or to other feature maps.

Weights: 3x3xDxF
Bias parameters: F

D = C_(l-1) … Channels last layer
F = Feature maps

Feature Maps are the 3rd dimension of the output of a layer (channels). It is needed so that one conv layer can detect more features than just one. Each feature map has its own kernel and therefore can detect something else. A conv layer has a neuron grid of W x H x C, with C feature maps of size W x H. Only neurons in the same feature map (channel) share parameters. A conv layer thus can learn C different features.

What is the purpose of pooling layers? Calculate the output of a 2×2 max-pooling layer with stride 2 assuming the following input. What is global average-pooling and what is it used for? Are there alternatives to pooling layers?

$$\begin{bmatrix} 1 & 1 & 2 & 4 \\ 5 & 6 & 7 & 8 \\ 3 & 2 & 1 & 0 \\ 1 & 2 & 3 & 4 \end{bmatrix} \Longrightarrow$$

6 8
3 4

Purpose: Reducing the dimensionality
Aggregation with pooling can be max or average pooling,
Pooling leaves the channel C unchanged.

Alternatives: a conv layer with a stride higher than 1 or without padding can also be used to reduce the dimensionality
Global average pooling: It is a usual pooling layer but with a kernel size that fits the whole input. So the output is only one number per feature map, the average of that map. It is used for the last layer of the conv layers. If the output is directly fed into the softmax it enforces a correspondence between feature map and class. If there is a fc layer after the global average pooling it is used so that the network can handle different input resolutions without changing anything. The fc always gets the same sized input vector.

2×2 max-pooling layer with stride 2

- $W_l = W_{l-1}/2$, $H_l = H_{l-1}/2$, and $c = 2$
- Output of neuron h is $\max(\mathbf{X}_h)$ with $\mathbf{X}_h \in \mathbb{R}^{2\times 2}$

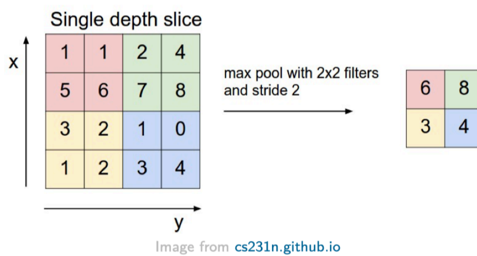
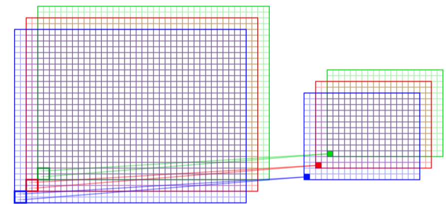


Image from cs231n.github.io

Number of neurons reduced by factor 4

- Corresponding efficiency increase
- At the cost of losing spatial resolution



Give a general overview of convolutional neural networks and their purpose, and draw a sketch that illustrates their overall structure (typical layer types and their arrangement). What are the two overall stages of such networks? What is needed to "combine" these stages / to make them compatible and how can this be achieved?

Overview of convolutional neural networks:

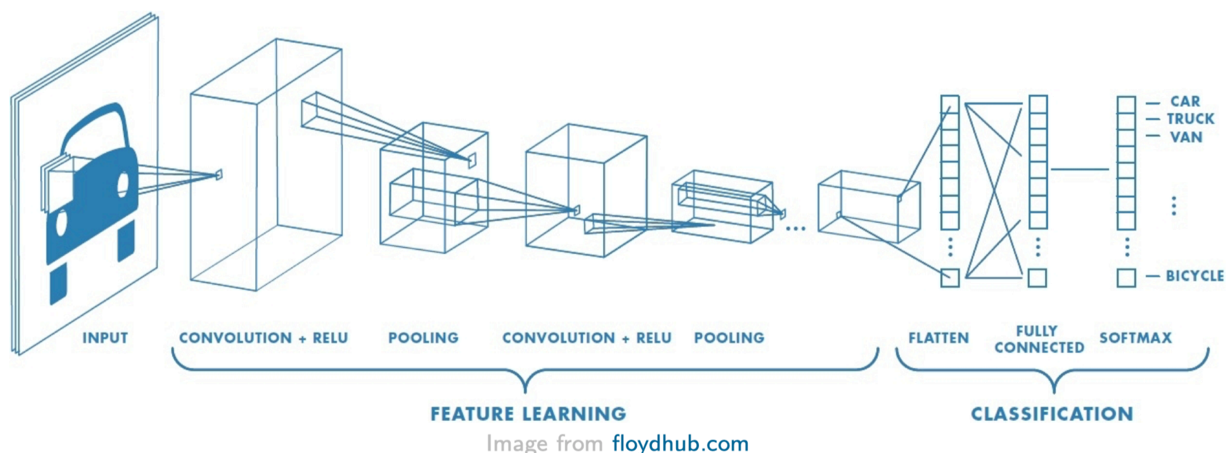


Image from floydhub.com

2 overall stages:
- Feature Learning
- Classification

Input images should be square images, i.e. H = W. If necessary, crop image to have a size of R. R should be divisible by 2 (to avoid problems during pooling).
With Convolution Layer and Relu we do hierarchical feature extraction from the input image. Pooling is used in between for dimensionality reduction (of W and H). We stack multiple blocks of conv and pool layers. The output of feature conv and pool layers are 3D tensors of shape WxHxC. The input tensor for the classification stage should have a W and H between 1 and 9. Modern architectures reduce W and H to 1 with global average pooling.

For classification we flatten the 3D tensor to a vector. We add a linear or non-linear (MLP) layer for classification. The number of neurons in the final fully-connected output layer should be the number of classes. We use a softmax activation function to obtain a likelihood-distribution of the class scores.

How is the depth of a CNN defined? What effect does increasing the depth have? Assuming an image classification problem with 10 classes and an image resolution of 64× 64 pixels, specify a suitable CNN architecture using the notation Cx (3 × 3 convolution with x feature maps), Lx (linear layer with x neurons), R (ReLU), P (2×2 max-pooling), B (batch normalization). for instance "C16 R P L20" would mean "convolutional layer with 16 feature maps followed by ReLU, max-pooling and a linear layer with 20 neurons". Explain why you selected this architecture.

Depth of a CNN is defined by the number of layers that network has (without considering the input layer).

Effects of increasing depth:
Theoretically, it improves performance of the network (at least with up to 15 layers of a standard architecture).

However, this is not always the case because of: Vanishing/Exploding gradients

How to solve it?

- Skip connections (ResNet)
- Batch normalization

We may design and experiment with a **neural network architecture** like this:
If we have 64x64 pixel (rgb), then we could do:

conv 3x3 with 64 channels (=C64),
bach normalization (B)
relu
C64
bach normalization (B)
relu
max pool 2x2 (=P)

conv 3x3 with 128 channels (=C128),
bach normalization (B)
relu
C128
bach normalization (B)
relu

global average pooling

flatten
linear layer 10 (10 because of 10 classes)
batch normalization
(relu or not)
softmax

Explain the two ways covered for computing the gradient of loss functions as well as their pros and cons. Math is not required but of course allowed.

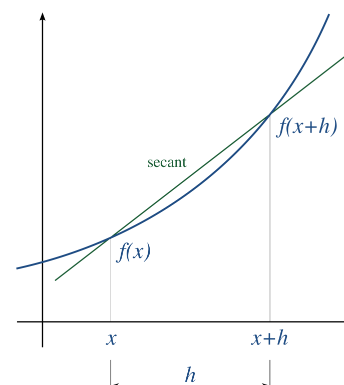
Two approaches for computing the Gradient are:
- Numerical differentiation
- Analytical using Calculus

Gradient computation with Numerical differentiation:
A simple two-point estimation is to compute the slope of a nearby secant line through the points (x, f(x)) and (x + h, f(x + h)). Choosing a small number h, h represents a small change in x, and it can be either positive or negative. The slope of this line is

$$\frac{f(x+h) - f(x)}{h}$$

One way to obtain $\nabla L(\boldsymbol{\theta})$ is numerical differentiation

- $\nabla L_p(\boldsymbol{\theta}) = (L(\boldsymbol{\theta} + \mathbf{1}_p \epsilon) - L(\boldsymbol{\theta}))/\epsilon$ for $p \in [1, \dim(\boldsymbol{\theta})]$
- Vector $\mathbf{1}_p$ is 1 at position p and 0 otherwise
- Follows directly from definition of the derivative



Numerical differentiation
- is easy to calculate
- it is slow

- it is only an approximation
- too inefficient in practice, because you have to calculate L dim(θ) + 1 times (at least) and complex models have millions of parameters

Analytic Gradient computation with Calculus:

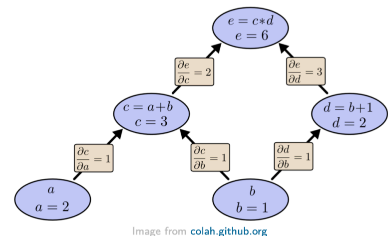
Recall that a NN is a computational graph — a function f mapping from x to w, composed of other functions. The loss function of a NN is again a graph.

Derivatives in such a graph can be computed iteratively by recursive application of the chain rule. Suppose we have a function F(x) = f(g(x)) then the derivative is F'(x) = f'(g(x)) * g'(x)

To compute gradients in such a graph we
- Evaluate the graph to store local results (forward pass)
- Aggregate local gradients (backward pass)

To calculate the Gradient in a computational graph we use the multivariate chain rule.
Example graph may be: e(a, b) = (a + b)(b + 1)

$$e_b(2,1) = c_b(2,1) \cdot e_c(2,1) + d_b(2,1) \cdot e_d(2,1) = 2 + 3 = 5$$



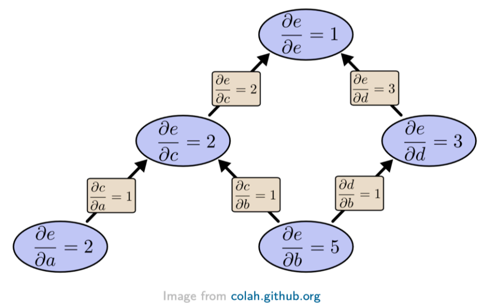
For every partial derivatives in the Gradient:
- Multiply local gradients along every path from a to e.
- Sum over all resulting values

A drawback of this algorithm (especially for large NN) is that we have to sum over many paths per partial derivative and visit edges multiple times.

An extension of that is to use **Backpropagation** (**Reverse-mode differentiation**) that solves this problem.

Backpropagation computes derivatives of output node wrt. all other nodes.
We do not have to visit a single edge more than once, so it saves us from doing redundant computations. It's more efficient by touching every edge only once.

Backpropagation achieved by:
- Start at the output (loss) node
- Propagate local gradients backwards to input nodes
- Storing intermediate results for efficiency



In detail:
Start at output node e and move towards inputs

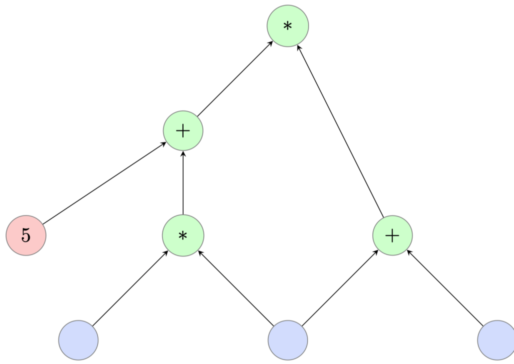
At every node n
- For every child c, compute local gradient l_c = ∂n/∂c
- For every child c, compute m_c = l_c · ∂e/∂n   (∂e/∂e = 1)
- Compute ∂e/∂c as sum over all m_c

https://colah.github.io/posts/2015-08-Backprop/

What is the purpose of the backpropagation algorithm, how does it differ from the "naive" algorithm for the same purpose, and what are its advantages? Explain the steps of the algorithm at a given node of a computational graph.

See answer for question above.

Assume the following computational graph. first insert digits from your Matrikelnummer into the empty input nodes, going from right to left both in terms of nodes and digits. (Assuming Matrikelnummer 0123456, the values of the rightmost node would be 6, that of the node left of it would be 5, and so on.) Then compute the partial derivative of the topmost node with respect to all input nodes via backpropagation. Write computation node values after the forward pass left of the nodes, local gradients left of the edge connecting the corresponding nodes, and "cached" partial derivatives of the topmost node right to the nodes.



2 Examples – so you get the idea:

Matr nr: ... 8 0 1

Gradient = $\begin{pmatrix}1\\0\\13\\5\end{pmatrix}$

$f = (ab+5)\cdot(b+c) = ab^2 + 5b + abc + 5c =$

$f_a' = b^2 + bc \rightarrow 0^2 + 0\cdot 1 = 0$

$f_b' = 2ba + 5 + ac \rightarrow 2\cdot 0\cdot 8 + 5 + 8\cdot 1 = 13$

$f_c' = ab + 5 \rightarrow 8\cdot 0 + 5 = 5$

$f_d' = b + c \rightarrow 0 + 1 = 1$

Mat nr: ... 136

Gradient = $\begin{pmatrix}9\\27\\17\\8\end{pmatrix}$

$f_a' = 3^2 + 3\cdot 6 = 27$

$f_b' = 2\cdot 3\cdot 1 + 5 + 1\cdot 6 = 17$

$f_c' = 1\cdot 3 + 5 = 8$

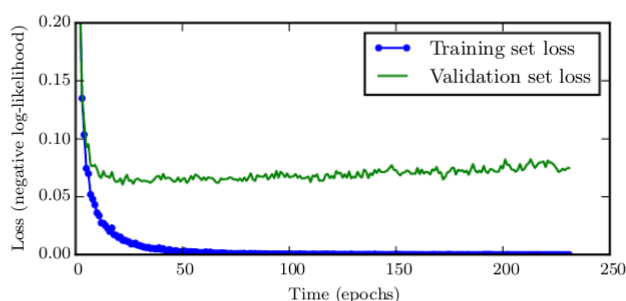
$f_d' = 3 + 6 = 9$

Analytic Gradient Computation, Back propagation

What are the goals of optimization and machine learning? Why do they differ? Create two sketches with each showing the training progress over time in terms of both training and test error; one that is good from an optimization perspective but bad from a machine learning perspective, and one that is worse from an optimization perspective but better from a machine learning perspective. Explain both sketches.

Training a model can be seen as an optimization perspective, because you want to minimize the loss to ~ 0. Training is finding parameters that minimize training loss (known as empirical risk minimization).

But, this is prone to overfitting to the training data and resulting in disappointing validation/test performance. Overfitted models are unable to generalize well to unseen data.



The image shows: Training loss decreases steadily. Validation loss begins to rise again at some point.

In ML our goals are actually a low training loss (avoid underfitting) and a small gap to the validation loss (avoid overfitting).

To achieve these goals we have to minimize the training loss, while also combatting overfitting via

- Data augmentation
- Regularization
- Early Stopping

We have to improve the validation/test performance at the possible expense of training performance.

What is overfitting and why is it undesirable? Explain three approaches to combat overfitting.

During learning, training loss is decreasing. Overfitting is a concept in data science, which occurs when a statistical model fits exactly against its training data. ... When the model memorizes the noise and fits too closely to the training set, the model becomes "overfitted," and it is unable to generalize well to new data. It is undesirable because our model is not good for any new datasets.

Three approaches are:

- early stopping
- data augmentation
- regularization (-> dropout, weight decay)
- train on more data (in practice limited)

What is early stopping and how does it work? What is the purpose of data augmentation? Assume that the task is to train a digit classifier. Think of and explain data transformations that are applicable in this case, and at least one that is not.

In **early stopping** we simply save the current model to disk as best_model if it has the highest validation performance seen so far (and overwrite that file in the process of training). As the epochs go by, the algorithm learns and its error on the training set naturally goes down, and so does its error on the validation set. However, after a while, the validation error stops decreasing and actually starts to rise.

Early stopping is probably the most commonly used form of regularization in deep learning. Its popularity is due both to its effectiveness and its simplicity.
One way to think of early stopping is as a very efficient hyperparameter selection algorithm. In this view, the number of training steps is just another hyperparameter.

Early stopping is a very unobtrusive form of regularization, in that it requires almost no change in the underlying training procedure, the objective function, or the set of allowable parameter values. This means that it is easy to use early stopping without damaging the learning dynamics

The purpose of **data augmentation** is to increase the size of a limited dataset by creating new artifical data, e.g. by transformations like a shift or rotation, but also adjustments in brightness and contrast and many more techniques are possible.

DIGIT CLASSIFIER:
https://www.researchgate.net/publication/338429324_Effectiveness_of_Data_Augmentation_on_Handwritten_Digit_Classification

For the data augmentation, we choosed:
Randomly turn around some training images by 10 degrees
Randomly go fast by 10% a few training images
Randomly transfer images horizontally by 10% of the width
Randomly shift images vertically by 10% of the height

We did not apply a vertical_flip nor horizontal_flip since it could have lead to misclassify symmetrical numbers.

What is the purpose of regularization? What is dropout, how does it work, and why is it effective? Where inside a network is dropout usually applied? What is weight decay and how does it differ from dropout?

Purpose of regularization is to reduce the test error, possibly at the expense of increased training error, because a central problem in machine learning is how to make an algorithm that will perform well not just on the training data, but also on new inputs. It is often done by decreasing the model's variance, that is sensitivity to small changes in the training set and thus combatting overfitting.

Dropout is a layer type whose neurons
- Output 0 with probability p
- Forward input with probability 1 – p
- During training

Usually placed before last (dense) layer
- Has effect of temporarily "discarding" neurons
- As neuron has no effect on output on next layer
- Thus net learns not to rely on certain neurons

Dropout layers were introduced by Hinton et al., in 2012. Since their
introduction, these layers have become a standard feature in most deep architectures. Instances of the layer can be positioned almost anywhere in the network. In a subset of the experiments in , a dropout layer
was placed with the input layer (in addition to their inclusion with the hidden
layers), and this arrangement resulted in further reduction in the error for the
recognition task.

The dropout layer is always in support of another layer, such
as a convolution or a fully connected layer. In this manner a dropout layer
always associates with a corresponding functional layer. The dropout layer has a regularizing influence on the deep network and tends to decrease the variance and overfitting in a network. Turning off a number of layer neurons is equivalent to modifying the structure of a network layer. For a dropout probability 0.5, this behavior is equivalent to sampling from a set of 2^n different networks where n is the number of neurons in the corresponding
functional layer.

Simply put, dropout refers to ignoring units (i.e. neurons) during the training phase of certain set of neurons which is chosen at random. By "ignoring", I mean these units are not considered during a particular forward or backward pass.
More technically, At each training stage, individual nodes are either dropped out of the net with probability 1-p or kept with probability p, so that a reduced network is left; incoming and outgoing edges to a dropped-out node are also removed.

Weight decay

Weight Decay (L2 weight regularization) is very common

- Almost always used in practice Penalizes large weights (not biases)
- Preventing certain inputs from dominating output
- Thus encourages model to use all inputs

Implemented by adding regularization term to loss function

$$L_{\text{reg}}(\boldsymbol{\theta}) = \frac{\delta}{2}\|\mathbf{w}\|^2 + L(\boldsymbol{\theta})$$

$\mathbf{w} \subset \boldsymbol{\theta}$ is vector of all weights

$\delta \in (0, 1)$ controls amount of regularization

- Often $\delta \in [0.0001, 0.01]$

Why should images be normalized and how does this work during training and testing? What is the purpose of batch normalization?

Normalize your input images

- As input variance affects the network
- E.g. normalizing from [0, 255] to [0, 1] reduces variance. So make sure to normalize input images
- Subtract per-channel mean of training set
- Then divide by per-channel standard deviation

When testing, you have to subtract the training set mean and divide by the per-channel training set std.

Purpose of **batch normalization** is to have more stable signal distributions over time. Problem with the current state is that although our input X was normalized with time the output will no longer be on the same scale. As the data go through multiple layers of the neural network and L activation functions are applied, it leads to an internal covariate shift in the data.

Advantages of batch normalization:

- Speed Up the Training
  (by Normalizing the hidden layer activation the Batch normalization speeds up the training process.
- Handles internal covariate shift
  (it solves the problem of internal covariate shift. Through this, we ensure that the input for every layer is distributed around the same mean and standard deviatio)

Where to add Batch Normalization layers?

- Add after every conv and hidden linear layer.
- Before the activation functions

Proper initialization preserves input variance only initially

- ▶ Parameters change during training
- ▶ Thus output distribution of layer changes over time

This complicates training

- ▶ Must account for changes in input distribution
- ▶ Layer input affected by parameters of all previous layers

Batch normalization reduces this problem

- ▶ Estimate input mean and variance
- ▶ Using current minibatch to approximate training set
- ▶ Normalize accordingly (per feature map for conv layers)
- ▶ Multiply by learned γ, add learned β

Results in more stable signal distributions over time

What are exploding and vanishing gradients and what causes them? How can they be avoided? Explain batch normalization.

Turns out deep networks can be hard to train

▶ Due to vanishing or exploding gradients

▶ Caused by aggregating small or large gradients. Recall that gradient computation entails chained multiplications

▶ If the local gradients are small (or large)

▶ The final gradient may be close to 0 (or very large)

Vanishing gradients make it difficult to know which direction the parameters should move to improve the cost function, while exploding gradients can make learning unstable. The cliff structures described earlier that motivate gradient clipping are an example of the exploding gradient phenomenon.

Vanishing gradients -> 0. Exploding gradients -> Inf.

Batch normalization.

Batch normalization reduces this problem
- ▶ Estimate input mean and variance of those inputs (image by image in whole batch)
- ▶ Using current minibatch to approximate training set
- ▶ Normalize accordingly (per feature map for conv layers)
- ▶ Multiply by learned γ, add learned β

If the previous layer is a convolutional layer then we calculate mean and invariance by feature map and use those for normalization. And now it has 0 mean and 1 variance. For example if we have conv layer with 64 features then we would also have 64 γ, and β

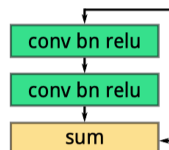
What are skip-connections and which problem do they solve? Explain two methods for combining signals. What is a residual network?

https://theaisummer.com/skip-connections/

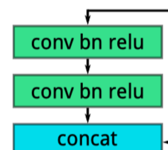
Skip connections facilitate signal propagation
- ▶ Connect layers that are not direct neighbors
- ▶ Allowing the signal to skip intermediate layers

To combine signals we have two options
- ▶ Sum them up
- ▶ Stack the tensors along channel dimension

When summing signals
- ▶ Input shape must not change in skipped layers
- ▶ Number of channels remains the same



When stacking signals
- ▶ Number of channels may change in skipped layers
- ▶ Number of channels will increase



As previously explained, using the chain rule, we must keep multiplying terms with the error gradient as we go backwards. However, in the long chain of multiplication, if we multiply many things together that are less than one, then the resulting gradient will be very small. Thus, the gradient becomes very small as we approach the earlier layers in a deep architecture. In some cases, the gradient becomes zero, meaning that we do not update the early layers at all.
In general, there are two fundamental ways that one could use skip connections through different non-sequential layers:
a) addition as in residual architectures,
b) concatenation as in densely connected architectures.
They are solving a problem of vanishing gradients. If you were trying to train a neural network back in 2014, you would definitely observe the so-called vanishing gradient problem. In simple terms: you are behind the screen checking the training process of your network and all you see is that the training loss stop decreasing but it is still far away from the desired value.
Apart from the vanishing gradients, there is another reason that we commonly use them. For a plethora of tasks (such as semantic segmentation, optical flow estimation , etc.) there is some information that was captured in the initial layers and we would like to allow the later layers to also learn from them. It has been observed that in earlier layers the learned features correspond to lower semantic information that is extracted from the input. If we had not used the skip connection that information would have turned too abstract.

Residual networks:

ResNet: skip connections via addition

The core idea is to backpropagate through the identity function, by just using a vector addition. Then the gradient would simply be multiplied by one and its value will be maintained in the earlier layers. This is the main idea behind Residual Networks (ResNets): they stack these skip residual blocks together. We use an identity function to **preserve the gradient**.

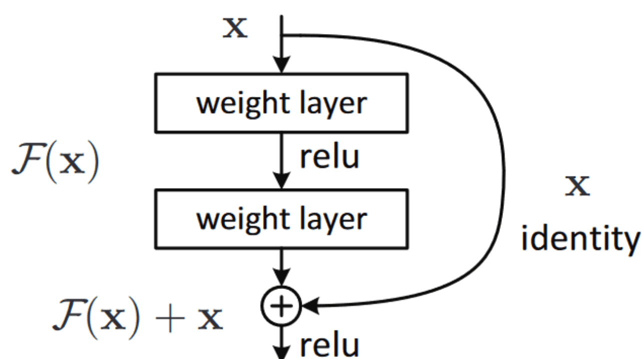
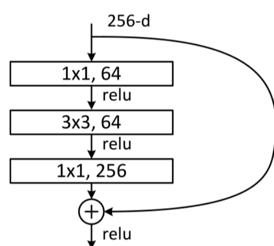


Image is taken from Res-Net original paper

Mathematically, we can represent the residual block, and calculate its partial derivative (gradient), given the loss function like this:

$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial H}\frac{\partial H}{\partial x} = \frac{\partial L}{\partial H}(\frac{\partial F}{\partial x} + 1) = \frac{\partial L}{\partial H}\frac{\partial F}{\partial x} + \frac{\partial L}{\partial H}$$

What are point-wise convolutions? Explain two applications covered in the lecture.



Initial 1 × 1 conv (pointwise conv)
- Reduces number of channels for efficiency
  (To decrease training time and resource consumption)
- Such layers are called bottleneck layers

1 × 1 convs are also used to support pooling
ResNet block
- Perform first conv with stride 2

Skip connection
- Add 1 × 1 conv with stride 2 before sum

Explain **two applications** covered in the lecture.
The 1×1 filter can be used to create a linear projection of a stack of feature maps.
The projection created by a 1×1 can act like channel-wise pooling and be used for dimensionality reduction.

The projection created by a 1×1 can also be used directly or be used to increase the number of feature maps in a model.

Two Applications: ResNet, MobileNet v2 -> channelwise conv followed by pointwise conv

---

What is learning rate decay and why is it useful?

Reduce the learning rate over time
▶ Allows for quick progress initially
▶ While providing stability later on Popular alternatives
▶ Reduce when validation loss no longer improves
▶ Reduce by factor of 10 after 50% and 75% of epochs
▶ Multiply by $\xi < 1$ after each epoch (exponential decay)

What are imbalanced data in classification, which negative effect do they have, and how can that effect be addressed?

So far we have assumed balanced data, i.e. Same number of samples per class

▶If this is not the case
▶ Classifier will pay more attention to majority classes
▶ Leading to bad accuracy for minority classes

This is usually not desired and must be addressed
▶ We will cover two popular options

Via class weights on the loss
- $L_s(\boldsymbol{\theta}) = \lambda_{c_s} H(\cdot)$ where c_s is class of sample s
- λ_{c_s} is weight based on frequency of class c_s
- Simple to implement, no computational overhead
- Does not work well for highly imbalanced data

Via oversampling of the training data
- Draw samples with replacement to balance class frequencies
- Works well with data augmentation
- Considerable computational overhead

Also the accuracy is not suitable anymore
▶ Does not consider per-class performance
▶ Use balanced accuracy or confusion matrix instead

What is the task definition of object detection? How can detection be implemented via classification and which downsides does this have?

Task definition of Object Detection:
Given a finite set of class labels (e.g. {bird, cat, dog}).

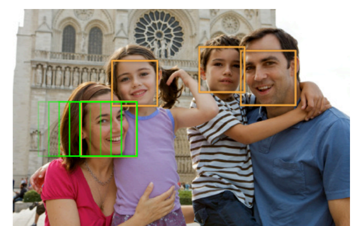


Image adapted from apple.com

We can have 0 or multiple relevant objects of these classes in an image.
We need to locate objects (of different classes) in an image.

Object detection using Sliding Window Approach:
We can implement Object Detection by Training a classifier for C+1 classes.
Usually an additional background class is used (class for unknown object).

For detection, we slide a fixed-size window over the image and predict class-scores for every window.
Then we perform non-maximum suppression.

Downside is that we must classify many windows.
A single fixed-size window does not suffice.
We must process the image at multiple scales (more inefficient).
Bounding boxes with a fixed aspect ratio is also a limitation.
We cannot handle multiple objects in the same window (softmax).
Instead of looking only once on the image, we have to look multiple times.

How does YOLO process images? Assuming two anchor boxes and five classes, what would YOLO predict? How is this information used for object detection / which post-processing steps are there?

In yolo the picture is divided into grid cells. For each grid cell we have anchor boxes. For each anchor box, the net predicts offset for the location and the scale (x, y, w, h), a confidence p and a class score vector C.
We use the boxes with the highest p value, apply the offset values and use the predicted class for that box. Then we remove all overlapping boxes with the same class prediction and repeat this step.

Our system divides the input image into an S × S grid. If the center of an object falls into a grid cell, that grid cell is responsible for detecting that object.
Each grid cell predicts B bounding boxes and confidence scores for those boxes. These confidence scores reflect how confident the model is that the box contains an object and also how accurate it thinks the box is that it predicts. Formally we define confidence as Pr(Object) ∗ IOU . If no object exists in that cell, the confidence scores should be zero. Otherwise we want the confidence score to equal the intersection over union (IOU) between the predicted box and the ground truth
Each bounding box consists of 5 predictions: x, y, w, h, and confidence. The (x, y) coordinates represent the center of the box relative to the bounds of the grid cell. The width and height are predicted relative to the whole image. Finally the confidence prediction represents the IOU between the predicted box and any ground truth box. Each grid cell also predicts C conditional class probabilities, Pr(Classi | Object). These probabilities are conditioned on the grid cell containing an object. We only predict one set of class probabilities per grid cell, regardless of the number of boxes B.

Assuming two anchor boxes and five classes, what would YOLO predict?
S*S *(B*(5 +C))
S*S*(2*(5+C))

S*S … Number of grid cells (width=height=S)
B … Number of Bounding boxes
5 … 5 prediction values, i.e. x, y, w, h and confidence p
C … C conditional class probabilities

How is this information used for object detection / which postprocessing steps are there?
(IS THIS CORRECT)??
We use the boxes with the highest p value, apply the offset values and use the predicted class for that box. Then we remove all overlapping boxes with the same class prediction and repeat this step.

We then pretrain the network for classification
- On a large dataset, usually ImageNet
- For the network to learn how the different objects look like
- Transfer learning as covered earlier

Weighted sum of three L2 losses per anchor box
- Confidence loss (loss on p)
- Classification loss (loss on class scores) *
- Localization loss (loss on x, y, w, h) *

(*) Only if IoU with ground-truth box is sufficient

What is semantic image segmentation? Draw a sketch of how CNNs for this purpose look like. What are the two overall stages of such networks? Which layers might they include that are not part of networks for other tasks?

Assigning a class value to each pixel of the input image. More specifically, the goal of semantic image segmentation is to label each pixel of an image with a corresponding class of what is being represented. Because we're predicting for every pixel in the image, this task is commonly referred to as dense prediction.

They look like a U with skip connection from layers in the downsampling to corresponding layer in the upsampling.

Down- and Upsampling are the stages.
For the upsampling they use special layers. E.g. upscaling, pixel shuffle, deconvolutions

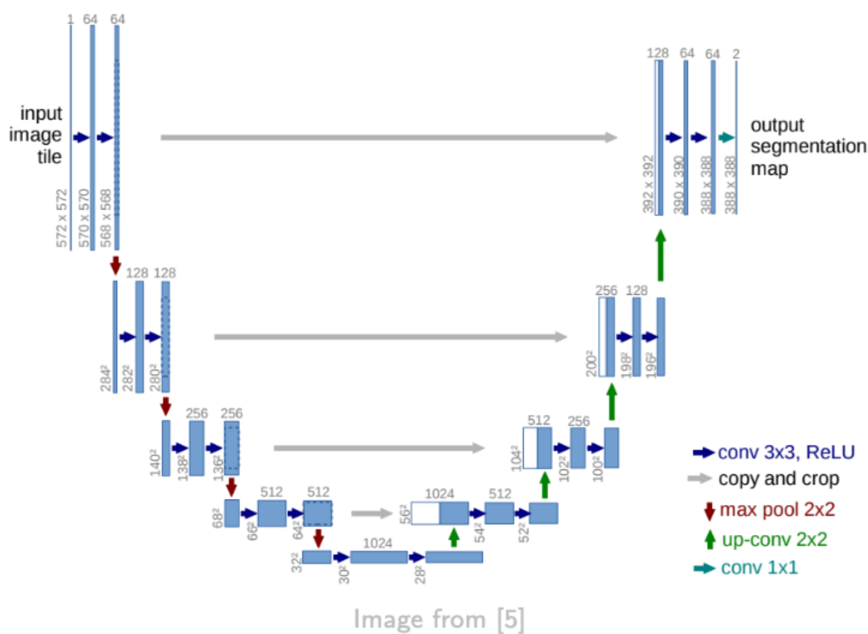


Image from [5]

What are transposed convolution layers? Assuming the following input I and kernel K, what would be the output of such a layer look like (stride 2, pad 1)?

$$I = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \quad , \quad K = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 2 & 1 \\ 1 & 1 & 1 \end{bmatrix} \quad \Longrightarrow$$

$K: \begin{bmatrix} 0 & 1 & 0 \\ 1 & 2 & 1 \\ 1 & 1 & 1 \end{bmatrix}$ $I: \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$

s=2, s=2, 5x5, K: 3x3, I: 2x2

$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}: \begin{bmatrix} 0 & 1 & 0 \\ 1 & 2 & 1 \\ 1 & 1 & 1 \end{bmatrix}$

$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}: \begin{bmatrix} 0 & 2 & 0 \\ 2 & 4 & 2 \\ 2 & 2 & 2 \end{bmatrix}$

$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}: \begin{bmatrix} 0 & 3 & 0 \\ 3 & 6 & 3 \\ 3 & 3 & 3 \end{bmatrix}$

$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}: \begin{bmatrix} 0 & 4 & 0 \\ 4 & 8 & 4 \\ 4 & 4 & 4 \end{bmatrix}$

Σ over overlapping cells

0	1	0	2	0
1	2	3	4	2
1	4	3	6	2
3	6	7	8	4
3	3	7	4	4

5x5

Result Matrix Size:

Apply Kernel 3x3 4 times (I has 2x2).

Stride = 2, padding = 1

Move along 1 dimension

$size(I) = \left\lfloor \frac{n + 2p - size(K)}{stride} \right\rfloor + 1$

$2 = \left\lfloor \frac{n + 2p - 3}{2} \right\rfloor + 1$

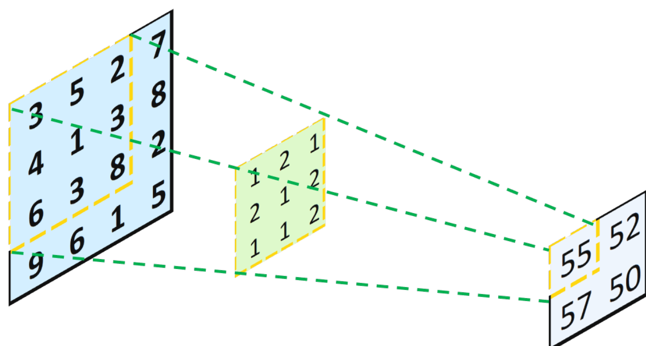
$3 + 2 \cdot (2-1) = n + 2p$

$n + 2p = 5$

Transposed Convolutions

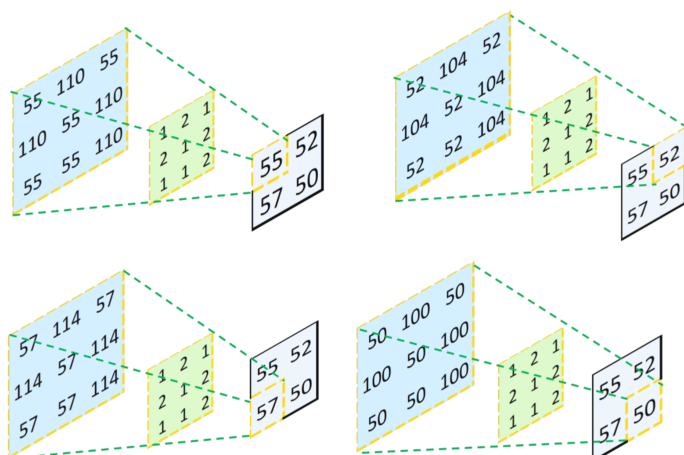
I understand the transposed convolution as the opposite of the convolution. In the convolutional layer, we use a special operation named cross-correlation (in machine learning, the operation is more often known as convolution, and thus the layers are named “Convolutional Layers”) to calculate the output values. This operation adds all the neighboring numbers in the input layer together, weighted by a convolution matrix (kernel). For example, in the image below, the output value 55 is calculated by the element-wise multiplication between the 3x3 part of the input layer and the 3x3 kernel, and sum all results together:

$$3 \times 1 + 5 \times 2 + 2 \times 1 + 4 \times 2 + 1 \times 1 + 3 \times 2 + 6 \times 1 + 3 \times 1 + 8 \times 2 = 55$$

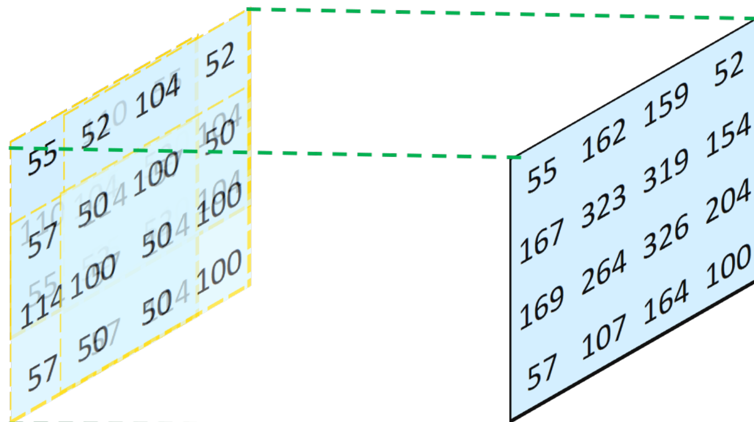


Without any padding, this operation transforms a 4x4 matrix into a 2x2 matrix. This looks like someone is casting the light from left to right, and projecting an object (the 4x4 matrix) through a hole (the 3x3 kernel), and yield a smaller object (the 2x2 matrix). Now, our question is: what if we want to go backward from a 2x2 matrix to a 4x4 matrix? Well, the intuitive way is, we just cast the light backward! Mathematically, instead of multiplying two 3x3 matrices, we can multiply each value in the input layer by the 3x3 kernel to yield a 3x3 matrix. Then, we just combine all of them together according to the initial positions in the input layer, and sum the overlapped values together:

Multiply Each Element in the Input Layer by Each Value in the Kernel



Combine All Four Resulting Layers Together And Sum the Overlapped Values (Image by Author)



In this way, it is always certain that the output of the transposed convolution operation can have exactly the same shape as the input of the previous convolution operation, because we just did exactly the reverse. However, you may notice that the numbers are not restored. Therefore, a totally different kernel has to be used to restore the initial input matrix, and this kernel can be determined through training.

https://towardsdatascience.com/understand-transposed-convolutions-and-build-your-own-transposed-convolution-layer-from-scratch-4f5d97b2967

https://medium.com/apache-mxnet/transposed-convolutions-explained-with-ms-excel-52d13030c7e8

Which methods are there for spatial upsampling in neural networks? Explain subpixel convolutions.

Different kinds of upsampling layers exist
▶ Upsampling
▶ Transposed convolutions
▶ Subpixel convolutions

Upsampling:

One way is to utilize standard 2D upsampling techniques
▶Nearest neighbor or bilinear interpolation
▶ Independently per channel

Fast and simple

Usually paired with a stride 1 conv layer

▶ To learn useful features / transformations

Example
- Upsampling by factor of 2
- Nearest neighbor interpolation

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \implies \begin{bmatrix} 1 & 1 & 2 & 2 \\ 1 & 1 & 2 & 2 \\ 3 & 3 & 4 & 4 \\ 3 & 3 & 4 & 4 \end{bmatrix}$$

Transposed Convolutions:

Transposed convolutions are convolutions with stride $1/s$
- Also called fractionally strided convolutions
- Or sometimes deconvolutions

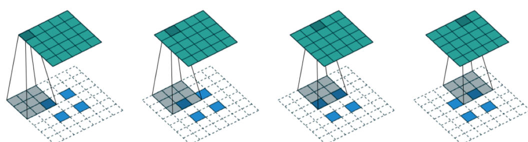


Image from [3]

Subpixel convolutions:

Subpixel convolutions are most recent upsampling method
- Usually outperforms other methods covered
- Also called pixel shuffle

Operation
- Given input of size $(C \cdot r^2) \times H \times W$
- Rearange (shuffle) to $C \times rH \times rW$
- Then do a stride 1 convolution

Example with $r = 3$ and $9 \times 7 \times 7$ input

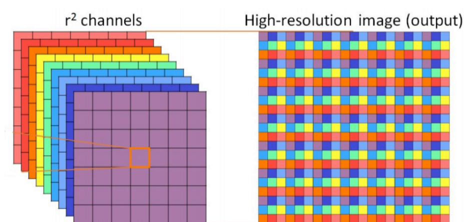


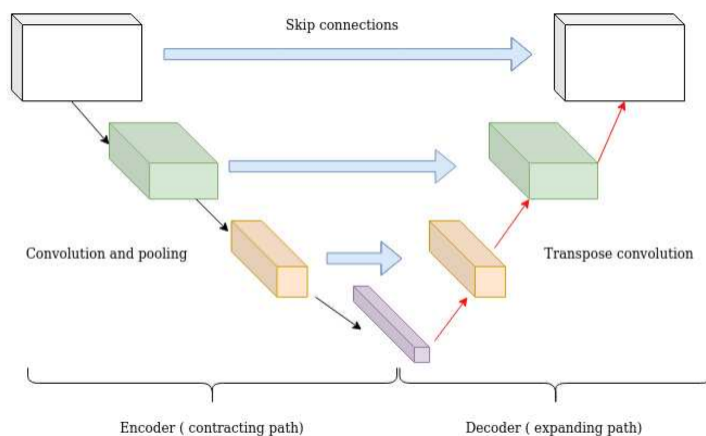
Image from [4]

What is a U-Net and what are such architectures used for? What are skip-connections, what is their purpose, and where are they located in the network? Draw a sketch.

UNet, evolved from the traditional convolutional neural network, was first designed and applied in 2015 to process biomedical images. As a general convolutional neural network focuses its task on image

classification, where input is an image and output is one label, but in biomedical cases, it requires us not only to distinguish whether there is a disease, but also to localise the area of abnormality.
UNet is dedicated to solving this problem. The reason it is able to localise and distinguish borders is by doing classification on every pixel, so the input and output share the same size.

By introducing skip connections in the encoder-decoded architecture, fine-grained details can be recovered in the prediction. Even though there is no theoretical justification, symmetrical long skip connections work incredibly effectively in dense prediction tasks (medical image segmentation).



A popular method for defining these connections is U-Net
▶Skip from before every downsampling layer
▶To after every corresponding upsampling layer
▶ Concatenate signals

What is a generative adversarial network? What are its main components and how do they interact? How does training work?

The main components are a generator and a discriminator. The generator generates new samples where the discriminator has to judge if this is a real sample or was only generated.
Train the generator to generate images that the discriminator does not recognize as fake till loss is below a threshold. Then train the discriminator to recognize the fake images till a threshold. Repeat that.

What is transfer learning and why is it useful?

Transfer learning is the process of using a model with already trained weights for your task which was trained on another task/dataset. For your own task you either just exchange the output layer and train the output layer accordingly for your task while freezing the weights of the previous layer, hence using the transfer model as a feature extractor (FAST).
Or you train the whole model, which is slower, since the weights of multiple layers are adjusted to fit your data.

How can deep learning approaches be used to obtain labeling data? Which approaches exist to simplify the annotation process of medical data?

Maybe Domain Adaptation?
You can use pretrained networks which were trained on a similar task to label your data. Moreover, you could label a small amount of your data and try to infer the rest (e.g. with transfer learning since then you do not need that massive amount of data). Data can also be created by using GANs.
Transfer learning can be used to perform automatic annotations from one image domain to another by mapping images from one medical domain, such as MRI, to another, e.g. CT scan.

Another approach is to simplify the annotation process by, for example, providing the labels of the image (which part are visible), instead of the segmentation masks, and in combination with the class activation maps, they are used to create image masks. As a final step, such masks are refined masks with other techniques to obtain more accurate masks.
A different option could be to train algorithms to be robust on weaker annotations, such as bounding boxes, squiggles, sparse dots or noisy labels, rather than using segmentation masks, in order to alleviate the amount of effort involved in the data annotation process.

What is the benefit/disadvantage of an unsupervised deep learning approach compared to a supervised approach in the medical domain? What is reinforcement learning and in which medical domain can it be used?

No annotation required; Overfitting and convergence as possible complication
Reinforcement: No predefined data, feedback by reward (Agent, Interpreter, Environment, Action, Reward, Punishment); Reward driven learning, Application: Robotic assisted surgery

---

Which network architecture can be used for anomaly detection in medical images and explain briefly how does it work?

train GAN on domain images
generate image that is as close as possible to a newly seen image
calculate difference between newly image and generated image = anomaly

---

What are the challenges in using medical imaging data? Can deep networks be used effectively for medical tasks (give 3 examples)? How can we use the training data most efficiently?

Datenschutz, small datasets, expensive experts
U-NET for tissue segmentation
GAN for retina anomaly detection
AE for cell classification
Most efficiently with a data augmentation, transfer learning, pre training, domain adaption, ???

What is the difference between 2D, 2.5D and 3D deep CNN networks?

2D takes 2D images as input and 3D takes 3D images as input. In 2.5D: 3 orthogonal 2D slices of a 3D space are taken as input. (NOT SURE: multiple randomly rotated views)

What approaches are key components to use deep CNNs in medical imaging applications, especially when data for training is not or sparsely available? Describe two approaches and the benefit of using them in medical imaging.

Data augmentation (Traditional, Mixing, Synthetic)
External labeled datasets (FineTuning, Domain Adaptation, Dataset Fusion)
Post segmentation refinement (CRF, cCRF, CRF as RNN) (WHAT IS THIS?)

Explain the challenges, benefits and drawbacks of using expert vs. non-experts to obtain labels of medical image data? Give an example of an alternative method to expert based manual labeling of medical data for deep learning approaches.

Experts: Cost intensive, time consuming
Non-Experts: noisy annotations, disagreement between users.
Alternative: Crowdsourcing

The following questions will not be part of the exam on 2020-06-23.

Name and describe the four steps of unfair algorithms according to US mathematician Cathy ONeil.

(Adapted from previous years answer-catalogue)

There are four levels of unfair algorithms:
- Level 1: unintentional stereotypes are reproduced (e.g. women are more often offered childcare positions, men are in leading positions) The data used for training contains these structures
- Level 2: bad algorithms due to carelessness: dark-skinned people are classified as gorillas; part-time jobs are arranged so that only one job is possible
- Level 3: legal, but "nasty": advertising industry is told when customers are particularly "vulnerable" and therefore compare less prices, accept higher prices, etc. Find the lowest possible wage someone would do a job for
- Level 4: illegal, programmed "malicious" on purpose: exhaust emissions from diesel vehicles, "sesame credit" chinese. private credit scoring and loyalty program system, "greyball" uber software to detect undesirable business practices

---

What is the meaning of XAI?

Explainable AI (XAI) is artificial intelligence (AI) in which the results of the solution can be understood by humans. It contrasts with the concept of the "black box" in machine learning where even its designers cannot explain why an AI arrived at a specific decision.
https://en.wikipedia.org/wiki/Explainable_artificial_intelligence

Transparency

- Entails the capability to describe, inspect and reproduce the mechanisms through which AI systems make decisions and learn to adapt to their environments
- Transparency is key to building and maintaining citizen's trust in the developers of AI systems

Explainability
AI's training methods and decision criteria may not be understood and may not be readily available for challenge and validation by a human operator

Design AI systems with explainability:

i. researching and attempting to use the simplest and most interpretable model possible for the application in question
ii. assessing whether it is possible to analyse, change and update training and testing data
iii. assessing whether interpretability can be examined after the model's training and development

Bias

Bias is a prejudice for or against something or somebody, that may result in unfair decisions.
BIAS leads to unfair and discriminatory outcomes