Technische Universität München

Summary of the lecture I2DL

Introduction to Deep Learning

according to the Lecture of Prof. Niessner and Prof. Leal-Taixé

Compiled on December 8, 2022 at 13:18:37

Contents

Preface

Hey guys, Eva created this great summary and I added some of my notes, and we would like to share it with you, so everyone can add their notes. As we share the editable link, please make sure not to delete it ;) and you are very welcome to add your notes / pictures / improve the latex file (I am not a pro, just used a good template). If you add your note just make sure not to write anything that is already in the document — it is better to improve and expand the correct paragraphs. Have fun!!

Munich 2021-2023

I Introduction

Computer Vision: We're trying to make sure that machines are learning to see in a similar way that humans are doing. That is why we need Machine Learning methods, to get there. CV is the center for robotics so that you understand the environment and what it does for you. There's a lot of images/videos processing done by CV as well.

I.1 The history of Computer VisionThe history of Computer Vision

Hubel and Wiesel Experiment

Hubel and Wiesel (neurobiologists) experimented on cats' brains by putting electrodes in it and recording them while the cat was being shown stimuli through a screen (mostly edges). They were able to find out that visual cortex cells are sensitive to the orientation of edges, yet they were insensitive to the position of the edges. Something that we will see later in convolutional networks.

I.2 The summer vision project 1966

They tried to construct a significant part of a visual system, and it was the time when pattern recognition was coined.

CV is a very core element to other areas as well such as robotics, NLP, optics and image processing, algo optimization, neuroscience, AI and ML.

I.3 Image classification

Previously DL was not used for Image classification yet it became popular later. Earlier they did preprocessing (i.e. normalizing the colors of images), then came the feature descriptor which funcioned kind of the same as the Hubel Wiesel Experiment in the sense that certain properties were not important such as position of edges.

Different types of feature descriptor are HAAR, HOG, SIFT, SURF. In order to get to these feature descriptor they had to hand engineer it, since most are gradient based. After that you have aggregators such as svm rf, ann etc which would aggreagate the features and give the label.

Instead of feature extraction+accumulation we have a magic box that does that for us. That magic box is deep learning. We do not have the hand engineer the feature descriptors. We are letting a data set decide what the best possible descriptor might be that will give us the best results.

Image Classification Issues:

- Occlussion.
- Background cluttering: Background and foreground (object) similar colors
- Representation: Ex: cat drawing vs cat photo

I.4 History of Deep Learning

Started in 1940 with the electronic brain. Each cell has a certain pattern in them. They accumulated weights/impulses and eventually made a decision.

1960 we saw the perceptron. Instead of fixed weights, we could learn the weights. We showed the system a couple of example and we hope to essentially learn certain parameters of these perceptrons. We learn the feature extraction (weights) and the threshold of learning. This was all hardwired.

Then we had Adaline (the golden age of deep learning). There was a lot of hype and progress being made.

Then in 1969, people realized the problems with perceptrons, specifically the xor problem. The problem was that a linear model (a single perceptron) cannot separate the two classes. This era was called the AI winter.

In 1986, the multi-layer perceptron came to light. We have several layers that can be trained (optimized for the weights of the multi-layer perceptron). This is called backpropagation. Gradient based method.

In 1995 there was the SVM. Since it was successful, it put a halt to deep learning.

In 2006, Hinton and Ruslan developed Deep Belief Networks. The idea of pretraining came around. So you train an nn and then you train it again for a specific task. The idea of pretraining is still one of the most relevant today (for example transfer learning with ImageNet weights). Despite of this, neural networks were still not a mainstream method.

In 2012 : the AlexNet architecture (see Section X.2) was the first neural network based architecture that won the ImageNet competition based on the lowest top 5-error.

Definition of top 5 error: Give me an image, ask the method what class it is and see if the top five predictions include the correct class.

I.5 What made this [Deep Learning] possible?

- Big Data: When we have big data, models learn where to learn from and we have so much more data today then we did back then. The datasets are also online.
- Better Hardware: Not only has the data changed, the hardware has changes as well (i.e. GPU). Hardware was developed for the rendering of images in games, and it is now used as well for deep learning, to train models faster.
- Models are more complex

I.6 Different Tasks in DL

- Object Detection
- Self-Driving Cars
- Gaming (i.e. AlphaGO, AlphaStar)
- Machine Translation
- Automated Text Generation (ChatBots)
- Healthcare, cancer Detection

II Machine Learning Basics

There are a large variety of tasks, such as image classification, which is sometimes (depending on the number of classes) called binary classification. (key words: background clutter, semantic differences). The task of image classification should be done by using data, we want to train a model and make it learn from the data. From here we can say that we distinguish ML methods into two types: unsupervised learning methods and supervised learning methods.

Unsupervised Learning: doesn't have any labels or target classes. We just want to find out the properties of the structure of the data. An example of this can be clustering, kmeans, pca.
Supervised Learning: in supervised data we have the respective labels or target classes(done by annotators).

But how do we actually learn image classification, and how do we know we are learning and not memorizing things? First, we need to separate the data into training data and test data. Take the train data and train a model. Then you test on the test data to see how well the model is doing. We measure the performance, to see how good the model is doing. A metric would be accuracy. An underlying assumption is that train and test data come from the same distribution.

Nearest neighbor Model: Unsupervised learning method[ii], labels the sample based on the majority label of its neighboring samples. The hyper-parameters to be tweaked in KNN are: k, L1 or L2 distances. These hyperparameters are found by using the validation data. (Train-validation-test: 60-20-20 or 80-10-10). The test set is used only once at the end to display the model's performance. It is not used to 'fix' the accuracy. The latter is done in the validation set.

Cross Validation: Split the data in K folds and iterate through the permutations; test on one of the folds and train on the rest.

Decision Boundaries are boundaries where the data is separated into classes.

The pros and cons of using linear decision boundaries:

+ It's very easy to implement and derive
+ It's easy to find the hyperparameters
- The distribution must be clearly separated
- Harder to use for multi classes (?)

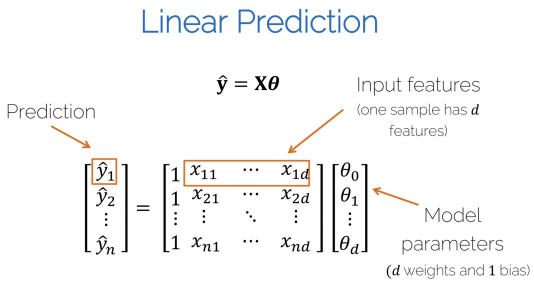
II.1 Linear Regression

Linear Regression is a supervised learning method that finds a linear model that explains a target y given inputs x with weights θ:

[ii] Well, I disagree with that. If you just want to list the neighbors, it is unsupervised. But as soon as you want to make predictions (e.g. average the neighbors), you need the true labels and it is supervised.

target, input, weights

$$\hat{y}_i = \sum_{j=1}^{d} x_{ij}\theta_j$$



and the prediction looks like:

bias

$$\hat{y}_i = \theta_0 + \sum_{j=1}^{d} x_{ij}\theta_j = \theta_0 + x_{i1}\theta_1 + x_{i2}\theta_2 + ... + x_{id}\theta_d \quad \Rightarrow \quad \hat{y} = \mathsf{X}\theta$$

x_{ij} are the features; θ are the weights (model parameters). θ_0 is a bias. In linear data it is where the decision line intersects with y.

Loss Function measures the goodness of the estimation and tells the optimization method how to make it better.

Optimization changes the model to improve the estimates and minimize the loss. The goal is to reduce the loss function.

Linear least Squares

$$J(\theta) = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2$$

$$\min_{\theta} J(\theta) = \frac{1}{n}\sum_{i=1}^{n}(x_i\theta - y_i)^2$$

J is the function that we want to minimize. Basically we want the find θs that minimize J.

To find the optimum we need to find $\frac{\partial J}{\partial \theta} = 0$

This loss function is convex which means it has a minimum which means there exists a solution i.e an optimal theta.

$(A \cdot B)^T = B^T A^T$ || $(A+B)^T = A^T + B^T$

Matrix notation: $\min_{\theta} J(\theta) = (\mathsf{X}\theta - y)^T(\mathsf{X}\theta - y)$ → $\theta^T X^T X\theta - \theta^T X^T y - y^T X\theta + y^T y$?

There exists a closed form solution for this loss function: $\theta = (X^T X)^{-1} X^T y = X^{\dagger} y$

Is least squares estimate the best estimate? If linear: Yes.

$\theta^T X^T X\theta - 2\theta^T X^T y + y^T y$? derivada

$X^T X\theta - X^T y = 0$;

$\theta = (X^T X)^{-1} X^T y$ ← $X^T X\theta - X^T y$

II.2 Maximum Likelihood

MLE is a method of estimating the parameters of a statistical model given observations. This is done by finding the parameter values that maximize the likelihood of making the observations given by the parameters.

$$\theta_{ML} = \arg\max_{\theta} p_{\text{model}}(Y|X,\theta)$$
$$= \prod_{i=1}^{n} p_{\text{model}}(y_i|x_i,\theta)$$
$$= \sum_{i=1}^{n} \log p_{\text{model}}(y_i|x_i,\theta)$$

$\log(a \cdot b) = \log(a) + \log(b)$

MLE assumes that the training samples are independent and generated by the same distribution.

What shape does our probability distribution have? Assuming Gaussian distribution: $y_i = \mathcal{N}((x_i)\theta, \sigma^2) = x_i\theta + \mathcal{N}(0, \sigma^2)$

$$p(y_i) = \frac{1}{\sqrt{(2\pi\sigma^2)}} e^{-\frac{1}{2\sigma^2}(y_i-\mu)^2}$$
$$p(y_i|x_i, \theta) = (2\pi\sigma^2)^{-1/2} e^{-\frac{1}{2\sigma^2}(y_i - x_i\theta)^2}$$

$N(\mu, \sigma^2)$

$\mu = (x_i)\theta$

then after more matrix calculations we get:

$$\theta_{ML} = -\frac{n}{2}\log(2\pi\sigma^2) - \frac{1}{2\sigma^2}(y - X\theta)^T(y - X\theta)$$
$$\theta = (X^T X)^{-1} X^T y$$

So the MLE is the same as the least squares estimate we found previously.

II.3 Logistic Regression

Sigmoid function maps the values into 0 and 1 and its formula is below:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

Regression: predict a continuous output value (e.g. temperature of a room)
Classification: predict a discrete value, Binary classification (output either 0 or 1) and Multi-class classification (set of N classes)

Probability of a binary output:

$$\hat{y} = p(y = 1|X, \theta) = \prod_{i=1}^{n} p(y_i = 1|x_i, \theta)$$

$$p(y|X, \theta) = \hat{y} = \prod_{i=1}^{n} \hat{y}_i^{y_i}(1 - \hat{y}_i)^{(1-y_i)}$$

$$\hat{y}_i = \sigma(x_i\theta)$$

$p(y_i = 1 | x_i, \theta)$

↓ bernoulli

$\hat{y}_i^{y_i} (1 - \hat{y}_i)^{(1-y_i)}$

$\hat{y}_i = \sigma(x_i\theta)$

Maximum Likelihood Estimate: $\boldsymbol{\theta}_{ML} = \arg\max_{\boldsymbol{\theta}} \log p(\mathrm{y} \mid \mathbf{X}, \boldsymbol{\theta})$

We are interested in maximizing the likelihood quantity $\theta_{ML} = argmax_\theta p(y = 1|X, \theta)$. Since log simplifies and maintains our maximum, we get $\theta_{ML} = argmax_\theta \log p(y = 1|X, \theta)$. Taking the logarithm, we had to add a negative, which ended up with a minimization. Here are the steps:

1. let $\hat{y} = p(y = 1|X, \theta)$
2. Since Binary Problem, we use Bernoulli, which can be defined as either :
   a) $P(k) = p^k(1 - p)^{(1-k)}, k = \{0, 1\}$ [we stick to this]
   b) $P(k) = pk + (1 - p)(1 - k), k = \{0, 1\}$

   As Both will produce :

a) $P(k=1)=p$

b) $P(k=0)=(1-p)$

3. Our prob is represented by sigmoid function. $\sigma(x)=\frac{1}{1+e^{-(x_i\theta)}}=\hat{y}$ (prediction)
   Note: The fraction in the sigmoid function with log will cause a negative later. Since $\log(\frac{a}{b})=\log(a)-\log(b)$

4. $\hat{y}_i=p(y=1|X,\theta)=\prod_{i=1}^{N}\hat{y}_i^{y_i}(1-\hat{y}_i)^{1-y_i}$. (going $\theta\rightarrow\hat{y}_i$)

5. Goal is to $\max_\theta p(y=1|X,\theta)=\max\hat{y}_i=\max\prod_{i=1}^{N}\hat{y}_i^{y_i}(1-\hat{y}_i)^{1-y_i}$

   This maximization boils down to 2 cases in Binary settings:

   a) $y_i=1, \max\prod_{i=1}^{N}\hat{y}_i^{y_i}(1-\hat{y}_i)^{1-y_i}$ reduces to $\max\prod_{i=1}^{N}\hat{y}_i\rightarrow$ *Goal* $\max\hat{y}_i$

   b) $y_i=0, \max\prod_{i=1}^{N}\hat{y}_i^{y_i}(1-\hat{y}_i)^{1-y_i}$ reduces to $\max\prod_{i=1}^{N}1-\hat{y}_i\rightarrow$ *Goal* $\min\hat{y}_i$

6. Taking $\log(p(y=1|X,\theta))$:

   Taking Log is tricky. As to Maintain Same Goals and to make sense :

   a) $log(p)$ Always -ve $\iff -log(p)$ Always +ve

      As log(p) where p is a probability, $p\in[0,1]$.Therefore, the logarithm function produces negative values

   b) We can't accept a -ve likelihood prob. , so we must Mult. by -1 with every log(prob).

   Therefore:

   $\log(p(y=1|X,\theta))=\log(\prod_{i=1}^{N}\hat{y}_i^{y_i}(1-\hat{y}_i)^{1-y_i})\Rightarrow -1\sum_{i=1}^{N}y_i\log(\hat{y}_i)+(1-y_i)\log(1-\hat{y}_i)=-\sum_{i=1}^{N}L(\hat{y}_i,y_i)$

   Goal is to:

   $\max_\theta\log(p(y=1|X,\theta))\Rightarrow -\sum_{i=1}^{N}y_i\log(\hat{y}_i)+(1-y_i)\log(1-\hat{y}_i)$

   This maximization boils down to 2 cases in Binary settings:

   a) $y_i=1, \max -1\sum_{i=1}^{N}\log(\hat{y}_i)=\min\sum_{i=1}^{N}\log(\hat{y}_i)\rightarrow$ Same Goal $\max\hat{y}_i$ ✓

   b) $y_i=0, \max -1\sum_{i=1}^{N}\log(1-\hat{y}_i)=\min\sum_{i=1}^{N}\log(1-\hat{y}_i)\rightarrow$ Same Goal $\min\hat{y}_i$ ✓

   c) These Goals also match from the perspective of NN weights. when you expand $\hat{y}=\sigma(x_i\theta)$:

      i. $y_i=1:\max\sum_{i=1}^{N}\log(\hat{y}_i)\Rightarrow\max\sum_{i=1}^{N}(x_i\theta)\rightarrow$ Goal $\max(x_i\theta)$

      ii. $y_i=0:\max\sum_{i=1}^{N}\log(1-\hat{y}_i)\Rightarrow\max\sum_{i=1}^{N}(1-x_i\theta)\rightarrow$ Goal $\max(1-x_i\theta)=\min x_i\theta$

7. Taking log transformed $\max_\theta\log(p(y=1|X,\theta))\rightarrow\min\sum_i^N L(\hat{y}_i,y_i)$

   Going from max $\rightarrow$ min feels more like a loss function that we hope to minimize .

8. Finally Cost func:

   $L(\hat{y}_i,y_i)=y_i\log(\hat{y}_i)+(1-y_i)\log(1-\hat{y}_i)$

   $C(\theta)=\frac{-1}{N}\sum_i^N L(\hat{y}_i,y_i)$ which we hope to min.

This is called binary cross-entropy loss or BCE.

In the more general case (number of classes >2), the cross entropy loss can be written as :

$$\mathcal{L}(\hat{y}_i,y_i)=\sum_{i=1}^{N}\sum_{j=1}^{C}y_{i,j}\log\hat{y_{i,j}}$$

Cost function: $C(\theta) = -\frac{1}{n}\sum_{i=1}^{n}\mathcal{L}(\hat{y}_i, y_i)$. We want to minimize the cost function (minimizing the negative log-likelihood is equivalent to maximizing the log-likelihood).

To optimize there is no closed form solution so we need to use iteration to solve it. This is done by using gradient descent. Gradient Descent based optimizers are explained in detail in section V.

L1 norm: $||v||_1 = \sum_{i=1}^{n}|v_i|$

L2 norm: $||v||_2 = \sqrt{\sum_{i=1}^{n}(v_i)^2}$

III Neural Networks

Linear score functions are defined as: 1 where W is the weights and x is our inputs.

The weights of the linear score would essentially be the mean of the loss function that's L2. Yet when the data has a lot of variety, the linear score function does not work anymore.

Adding more weight matrices does not work because the function is still linear. We need to add some non-linearity and we do that like the following:

$$
\begin{aligned}
\text{Lineare score function: } & f = Wx \\
\text{2 layers: } & f = W_2 \max(0, W_1 x) \\
\text{3 layers } & f = W_3 \max(0, W_2 \max(0, W_1 x)) \\
\vdots \quad & f = W_4 \tanh(0 W_3 \max(0, W_2 \max(0, W_1 x))) \\
& f = W_5 \sigma(W_4 \max(0 W_3 \max(0, W_2 \max(0, W_1 x))))
\end{aligned}
$$

Basically we have layers of non-linearity stacked on top of each other. These non linear functions are called activation functions.

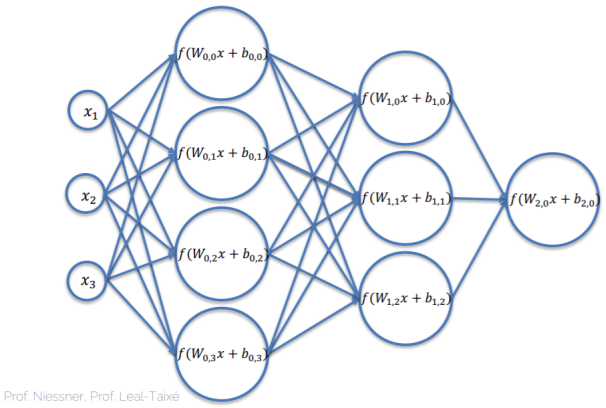


Figure III.1: Layers of a neural network: 3-dimensional input, first layer: 12 weights in total (3 inputs x 4 neurons) and 4 bias

As seen in the image above, a neural network is made of the input layer, the hidden layers and the output layer. The output layer size is the number of classes. The graph is fully connected, meaning that the previous layer's nodes are connected to all of the next layer's node.

Activation Functions

- Sigmoid: $\sigma(x) = \frac{1}{1+e^{-x}}$
- tanH: $\tanh(x)$
- ReLU: $\max(0, x)$
- Leaky ReLU: $\max(0.1x, x)$
- Parametric ReLU: $\max(ax, x)$

- Maxout: $\max(w_1^T x + b_1, w_2^T x + b_2)$
- ELU: $f(x) = \begin{cases} x \text{ if } x > 0 \\ \alpha\,(e^x - 1) \text{ if } x \leq 0 \end{cases}$

Why use activation functions? Without activation functions network would collapse and would have a linear model. Non-linearity → more capacity and better learning
Why organize a neural network into layers? Good to use with matrix calculations → easy to abstract math (e.g. gradient descent), every input layer corresponds to a picture.

Neural networks are inspired by biological neurons. Yet we are still at the beginning and not even close to what the human brain can do.

Summary

Given a dataset with ground truth training pairs $[x_i; y_i]$
Find Ws using stochastic gradient descent, that minimize the loss function. We need to compute gradients with backpropagation. Iterate many times over training set.

III.1 Computational Graphs

Computational graphs are directional graphs where matrix operations are represented as compute nodes; vertex nodes are variable or operators like $+ \ - \ * \ /$; directional edges show the flow of input to vertices.

Why? NN have complicated architectures, represent as computational graph because it has compute nodes (operations), edges that connect nodes (data flow), is directional (input from left to right, without loops), can be organized into layers.

Further meanings

- Multiplication of W_i and x: encode input information
- activation function: select key features
- convolutional layers: extract useful features with shared weights

$$BCE \to \frac{1}{n}\sum_{i=1}^{h} y_i \cdot \log(\hat{y}_i) + (1-y_i)\cdot \log(1-\hat{y}_i)$$

III.2 Loss Functions

The loss functions shows how close the predictions are to the targets. It's a measure of the goodness of predictions. A large loss function means bad predictions and the way we choose the loss functions is dependent on the problem at hand or the distribution of the target variable.

L1 Loss (Mean-absolute error Loss, MAE)

$$L(y, \hat{y}, \theta) = \frac{1}{n}\sum_{i=1}^{n} ||y_i - \hat{y}_i||_1$$

Mean-squared error (MSE) Loss

$$L(y, \hat{y}, \theta) = \frac{1}{n}\sum_{i=1}^{n} ||y_i - \hat{y}_i||_2^2$$

Binary Cross Entropy (BCE) for yes/no classification

$$L(y, \hat{y}, \theta) = -\frac{1}{n}\sum_{i=1}^{n}(y_i \log \hat{y}_i + (1 - y_i)\log[1 - \hat{y}_i])$$

Cross Entropy for multi-class classification

$$L(y, \hat{y}, \theta) = -\frac{1}{n}\sum_{i=1}^{n}\sum_{k=1}^{k}(y_{ik} * \log \hat{y}_{ik})$$

Notes:

- The minus sign is added thanks to the fact that we use $\log(p)$ where p is a probability, a value in the range of $[0, 1]$. Therefore, the logarithm function produces negative values. To overcome this, we add the minus.
- **Note** That for the **multi-class** classification we use the **one-hot-encoding** form of the ground-truth y_{ik}

Notations

- Ground truth: y
- Prediction: $\hat{y}$
- Loss function: $L(y, \hat{y}) = L(y, f_\theta(x))$
- Neural Network: $f_\theta(x)$ $\Rightarrow$ Goal: miminze the loss wrt $\theta \rightarrow$ Gradient Descent

The loss curve starts high and ideally goes lower and lower. The loss function can also be plotted against θ. And then we use gradient descent to find the optimal θ.

Gradient Descent

The updated θ where α is the step size or the learning rate (how fast we are going down the loss function).

$$\theta = \theta - \alpha\nabla_\theta L(y, f_\theta(x))$$

$$\theta^* = \arg\min L(y, f_\theta(x))$$

In order to compute the gradient for multi layers, we start from the nth layer and backpropagate to the first one using the chain rule.

Given the MSE Loss, we can derive the closed form formula for the derivative:

$$\nabla_\theta L(y, f_\theta(x)) = \frac{1}{n}\sum_{i}^{n}(W \cdot x_i - y_i) \cdot x_i^T$$

$$MSE = \frac{1}{n}\sum_{i=1}^{n} \|y_i - \hat{y}_i\|_2^2 \downarrow W \cdot x_i$$

Why gradient descent? Not always the best option, but easy to compute using compute graphs. Other methods: Newtons method, Adaptive moments, Conjugate gradient.

$$(y_i - W \cdot x_i)^2$$

$$y_i^2 - 2y_i \cdot W \cdot x_i + (W \cdot x_i)^2$$

Summary

- NN are computational graphs, Goal: for a given train set, find optimal weights
- Optimization is done using gradient-based solvers: many options
- Gradients are computed via backpropagation: Nice because can easily modularize complex functions

IV Optimization and backpropagation

Optimization schemes are based on computing gradients $\nabla_\theta L(\boldsymbol{\theta})$. Compute gradients analytically, but what if function is too complex → break down gradient computation: Backpropagation

IV.1 Backpropagation

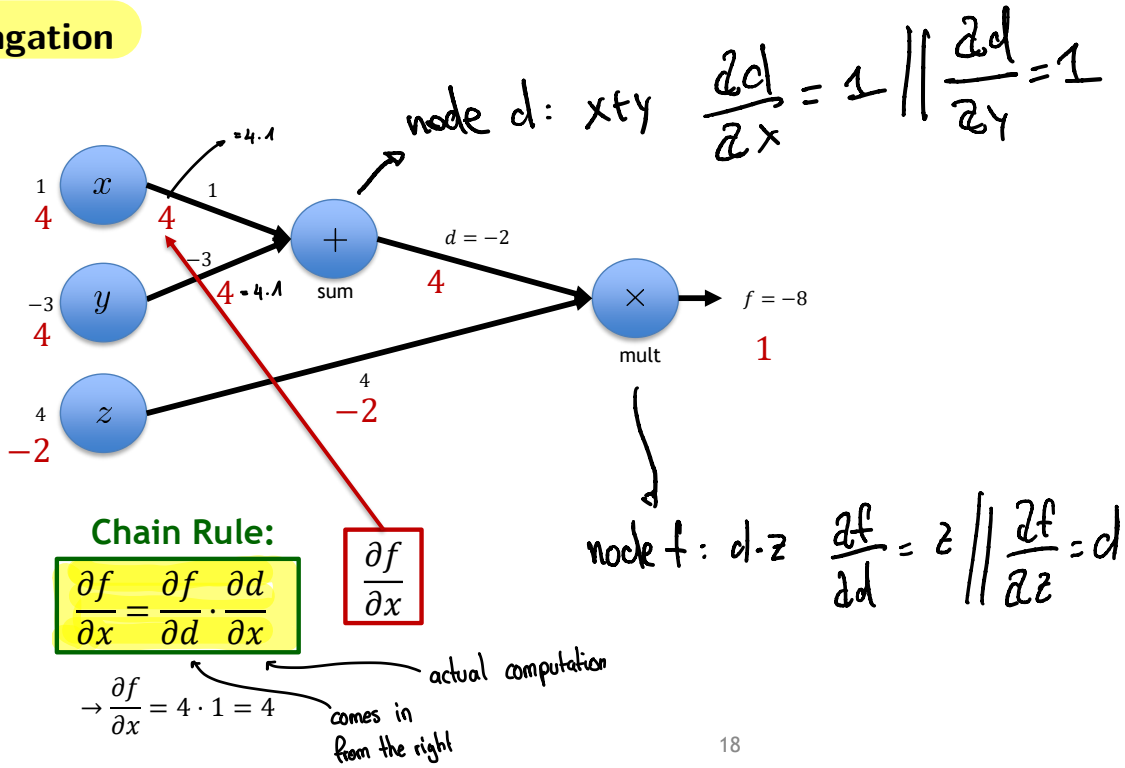


Figure IV.1: Exemplary computational graph for $f(x, y, z) = (x + y) \cdot z$
Forward pass in black and Backward pass in red

Forward pass Just calculate f. takes output from previous layer, performs operation, returns result. caches values needed for gradient computation during backprop

Backward pass Start from the end and compute the partial derivative for every node: $d = x + y$ and $f = d \cdot z$. takes upstream gradient, returns all partial derivatives.

$$\frac{\partial d}{\partial x} = 1, \frac{\partial d}{\partial y} = 1, \frac{\partial f}{\partial d} = z, \frac{\partial f}{\partial z} = d$$

Compute values with the chain rule.

Notations

- x_k input variables
- $w_{l,m,n}$ network weight: l which layer, m which neuron in layer, n which weight in neuron
- $\hat{y}_i$ computed output (i output dimension: n_{out})
- y_i ground truth targets

- L loss function

The bias is given as b_i in $\hat{y}_i = A(b_i + \sum^k x_k W_{i,j})$. If we want to compute the gradient of the loss function L wrt all weights W, we use the chain rule:

$$\text{Loss function: } L = \sum_i L_i$$

$$\text{L2 Loss: } L_i = (\hat{y}_i - y_i)^2$$

$$\text{Chain rule to compute partials: } \frac{\partial L}{\partial w_{i,k}} = \frac{\partial L}{\partial \hat{y}_i} \frac{\partial \hat{y}_i}{\partial w_{i,k}}$$

$\Rightarrow$ We compute the gradients wrt all the weights and all the biases.

IV.2 Gradient Descent

For a given training pair $\{x, y\}$, we want to update all weights, i.e. we need to compute the derivatives wrt all weights:

$$\nabla_W f_{\{x,y\}}(W) = \begin{bmatrix} \frac{\partial f}{\partial w_{0,0,0}} \\ \cdots \\ \cdots \\ \frac{\partial f}{\partial w_{l,m,n}} \end{bmatrix}$$

Gradient steps in direction of negative gradient:

$$W' = W - \alpha \nabla_W f_{\{x,y\}}(W),$$

where α is the learning rate and ∇ is the gradient w.r.t the weights.

Gradients can be computed analytically yet it is computationally expensive therefore it is easier to break down the gradient computation and utilize backpropagation.

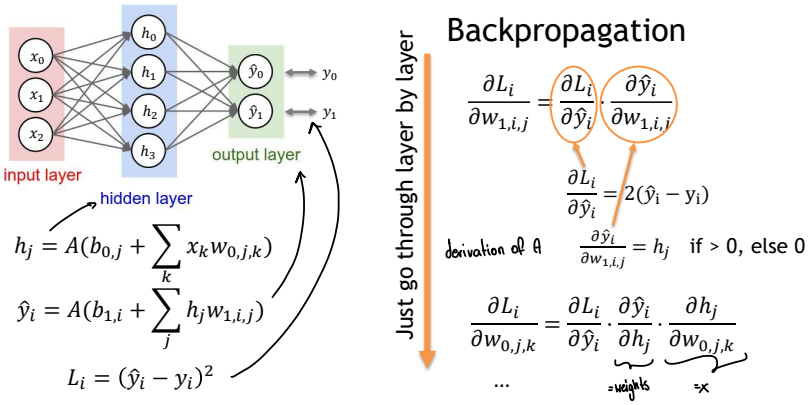


Figure IV.2: Gradient Descent for NN

How many unknown weights are there in input layer (3) hidden layer (4) and output (2)?
The output layer has: 2*4+2 weights → #neurons * # inputchannels + #biases
So the hidden layer will have: 4*3+4 weights

Derivatives of Cross Entropy Loss
Loss: $L = -\sum_{i=1}^{n_{out}} y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)$
Output: $\hat{y}_i = \frac{1}{1+e^{-s_i}}$, Scores: $s_i = \sum_j h_j w_{ji}$

Gradients of weights of last layer: $\frac{\partial L_i}{\partial w_{ji}} = \frac{\partial L_i}{\partial \hat{y}_i}\frac{\partial \hat{y}_i}{\partial s_i}\frac{\partial s_i}{\partial w_{ji}}$ with $\frac{\partial L_i}{\partial \hat{y}_i} = \frac{\hat{y}_i - y_i}{\hat{y}_i(1-\hat{y}_i)}$, $\frac{\partial \hat{y}_i}{\partial s_i} = \hat{y}_i(1-\hat{y}_i)$, $\frac{\partial s_i}{\partial w_{ji}} = h_j$
$\Rightarrow \frac{\partial L_i}{\partial w_{ji}} = (\hat{y}_i - y_i)h_j$, $\frac{\partial L_i}{\partial s_i} = \hat{y}_i - y_i$

Gradients of the first layer are:

$\frac{\partial L}{\partial h_j} = \sum_{i=1}^{n_{out}} \frac{\partial L}{\partial \hat{y}_i}\frac{\partial \hat{y}_i}{\partial s_j}\frac{\partial s_j}{\partial h_j} = \sum_{i=1}^{n_{out}} (\hat{y}_i - y_i)w_{ji}$

$\frac{\partial L}{\partial s_j^1} = \sum_{i=1}^{n_{out}} \frac{\partial L}{\partial s_i}\frac{\partial s_i}{\partial h_j}\frac{\partial h_j}{\partial s_j^1} = \sum_{i=1}^{n_{out}} (\hat{y}_i - y_i)w_{ji}(h_j(1-h_j))$

$\frac{\partial L}{\partial w_{kj}^1} = \sum_{i=1}^{n_{out}} \frac{\partial L}{\partial s_j^1}\frac{\partial s_j^1}{\partial w_{kj}^1} = \sum_{i=1}^{n_{out}} (\hat{y}_i - y_i)w_{ji}(h_j(1-h_j))x_k$

Example for Two-layer NN for regression with ReLU activation on Lecture 4, slide 44.

How to pick a good learning rate? If the learning rate is too high, then the model may not be able to converge to an optimal solution or it may converge too fast to a suboptimal solution and if the learning rate is too low, then the model will converge very slowly. We should find some sort of sweet spot so that the function converges fast and is not locked in saddle points.

How to compute the gradient for large training set? We can compute the gradient for individual training pairs and or training batches and then we take the average.

IV.3 Regularization

To close the generalization gap we can add regularization terms to the loss function such as the L2 reg or L1 reg → prevention from overfitting, make training harder, less likely to remember samples (Training error goes down, Validation error goes down and up, Difference = Generalization gap) (We can also use early stopping, dropout or max norm reg.)

$$\text{Loss function: } L(y, \hat{y}, \theta) = \sum_{i=1}^{n}(x_i\theta_{ji} - y_i)^2 + \lambda R(\theta) \quad \text{Regularization term}$$

$$\text{L2 Reg: } R(\theta) = \sum_{i=1}^{n}\theta_i^2$$

$$\text{L1 Reg: } R(\theta) = \sum_{i=1}^{n}|\theta_i|$$

L1: enforces sparsity (= Seltenheit), focus the attention to a few key features (probably more overfitting)
L2: enforces that the weights have similar values, will take all information into account to make decisions

The goal of regularization is to not overfit the data and it will make the training error higher, basically to generalize. It wants to make the training harder so that the network has to learn better features. The reg term should also not be too strong. We should also find a good balance for this term.
Regularization is any strategy that aims to lower validation error and increase training error.

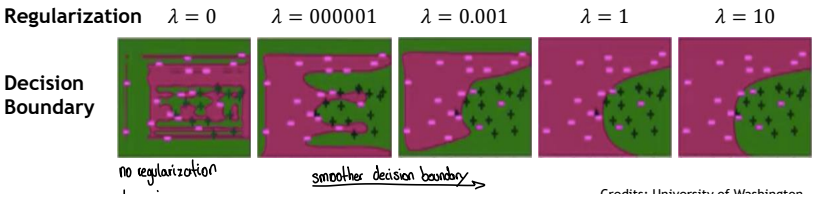
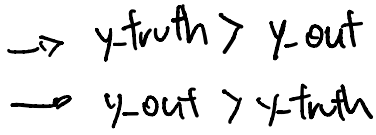


Figure IV.3: Different lambdas for Regularization

Forward and Backward passes of Loss function

```python
# L1: Forward
result = np.abs(y_out - y_truth)

# L1: Backward
gradient = y_out - y_truth

zero_loc = np.where(gradient == 0)
negative_loc = np.where(gradient < 0)
positive_loc = np.where(gradient > 0)

gradient[zero_loc] = 0
gradient[positive_loc] = 1
gradient[negative_loc] = -1

# MSE: Forward
result = (y_out - y_truth)**2

# MSE: Backward
gradient = 2 * (y_out - y_truth)

# BCE: Forward
result = -y_truth*np.log(y_out)-(1-y_truth)*np.log(1-y_out)

# BCE: Backward
gradient = -(y_truth/y_out) + (1-y_truth)/(1-y_out)
```



In the backward pass of the models always multiply with `d_out` or `self.cache`.

V Scaling Optimization and Stochastic Gradient Descent

V.1 Gradient Descent

As explained in the previous lecture, having w to optimize over, first we initialize it and then find the slope of the derivative of the function (in this case: the loss function) and we take gradient steps in the direction of the negative gradient. These steps that we take are dependent on the learning rate (α). When α is too large the step is bigger and vice versa. That is why α should be appropriately set.

Convergence

For a convex function: local minimum = global minimum BUT NN are non-convex (many different local minima, no practical way to say which is globally optimal).
Plateau is when the learning rate is significantly small and are are the gradient steps which leads to slow convergence or no convergence at all.

Given a loss function L and a single training sample $\{x_i, y_i\}$. Find the best model parameters $\theta = \{W, b\}$ such that the cost function $L_i\left(\boldsymbol{\theta}, \boldsymbol{x}_i, \boldsymbol{y}_i\right)$ is minimised. The gradient descent steps for single training sample:

1. Initialize θ^1 with random values
2. $\theta^{k+1} = \theta^k - \alpha \nabla_\theta L_i(\theta^k, x_i, y_i)$ (Computed via backpropagation)
3. Iterate until convergence $|\theta^{k+1} - \theta^k| < \epsilon$

Multiple training samples

1. Take the average cost function $L = \frac{1}{n}\sum_{i=1}^{n} L_i(\theta, x_i, y_i)$
2. Update step for multiple samples: $\theta^{k+1} = \theta^k - \alpha \nabla_\theta L(\theta^k, x_{1..n}, y_{1..n})$ with $\nabla_{\boldsymbol{\theta}} L\left(\boldsymbol{\theta}^k, \boldsymbol{x}_{\{1.n\}}, \boldsymbol{y}_{\{1.n\}}\right) = \frac{1}{n}\sum_{i=1}^{n} \nabla_{\boldsymbol{\theta}} L_i\left(\boldsymbol{\theta}^k, \boldsymbol{x}_i, \boldsymbol{y}_{\boldsymbol{i}}\right)$
3. Often written as $\nabla L = \sum_{i=1}^{n} \nabla_\theta L_i$ (omitting 1/n not wrong, but means rescaling the learning rate)

Optimal Learning Rate with Line search

1. Compute gradient $\nabla_\theta L = \frac{1}{n}\sum_{i=1}^{n} \nabla_\theta L_i$
2. Optimize for optimal step α: $\arg\min_\alpha L \underbrace{\left(\boldsymbol{\theta}^k - \alpha \nabla_{\boldsymbol{\theta}} L\right)}_{\boldsymbol{\theta}^{k+1}}$
3. $\boldsymbol{\theta}^{k+1} = \boldsymbol{\theta}^k - \alpha \nabla_{\boldsymbol{\theta}} L$

Problem: not practical for DL since need to solve huge system every step.

Stochastic Gradient Descent (SGD)

Since the gradient descent is very computationally complex, we can also use stochastic gradient descent which takes only a random portion of the data to train on.

We can consider the problem as empirical risk minimization where the loss over the training data is expressed as the expectation of all samples: $\frac{1}{n}\sum_{i=1}^{n} L_i(\theta, x_i, y_i) = E_{i\ [1..n]}[L_i(\theta, x_i, y_i)]$ where the expectation can be approximated with a small subset of the data:

$$E_{i\ [1..n]}[L_i(\theta, x_i, y_i)] = \frac{1}{|S|}\sum_{j\in S}(L_i(\theta, x_i, y_i)) \quad \text{with} \quad S \subseteq \{1...n\}$$

A minibatch is a smaller subset of the data where $m << n$ and the minibatch size is also a hyperparameter typically of a power of 2:

$$B_i = \{\{\boldsymbol{x_1}, \boldsymbol{y_1}\}, \{\boldsymbol{x_2}, \boldsymbol{y_2}\}, \ldots, \{\boldsymbol{x_m}, \boldsymbol{y_m}\}\}$$
$$\left\{B_1, B_2, \ldots, B_{n/m}\right\} \tag{V.1}$$

n: number of total samples, m: number of samples in batch, n/m: number of batches.

Smaller batch size means greater variance in the gradients (noisy updates).

An epoch is a complete pass through the train set. So after an epoch, we have seen all the samples from the training set.

- one epoch = one forward pass and one backward pass of all the training examples.
- number of iterations = number of passes, each pass using [batch size] number of examples. To be clear, one pass = one forward pass + one backward pass (we do not count the forward pass and backward pass as two different passes).

Example: if you have 1000 training examples, and your batch size is 500, then it will take 2 iterations to complete 1 epoch.

Every time we do an iteration the mini batch is different since it takes random data from the training set on each run. Compute the gradient for one minibatch, then update the weights.

When want to minimize the function $F(\theta)$ with the stochastic approximation:

$$\theta^{k+1} = \theta^k - \alpha_k H(\theta^k, X))$$

where $\alpha_1, \alpha_2 \ldots \alpha_n$ is a sequence of positive step-sizes and $H\left(\theta^k, X\right)$ is the unbiased estimate of $\nabla \mathrm{F}\left(\theta^k\right)$, i.e. $\mathbb{E}\left[H\left(\theta^k, X\right)\right] = \nabla \mathrm{F}\left(\theta^k\right)$.
It will converge to a local (global) minimum if the following conditions are met:

- $a_n \geq 0, \forall n \geq 0$
- $\sum_{n=1}^{\infty} \alpha_n = \infty$
- $\sum_{n=1}^{\infty} (\alpha_n)^2 < \infty$
- $F(\theta)$ is strictly convex

Robbins and Monro sequence is: $a_n \propto \frac{a}{n}$, for $n > 0$

Problems SGD has two main problems:

- The gradient is scaled equally across all dimensions. Cannot independently scale directions, need to have conservative min learning rate to avoid divergence, slower than necessary
- Finding a good learning rate is quite difficult

Gradient Descent with Momentum: With momentums we want to accumulate gradients over time, we want it go slower on the vertical jumps and faster on the horizontal ones (see Fig. V.1). So we want to take the history of gradients. Compute velocity

$$v^{k+1} = \beta v^k - \alpha \nabla_\theta L(\theta^k) \quad \theta^{k+1} = \theta^k + v^{k+1}$$

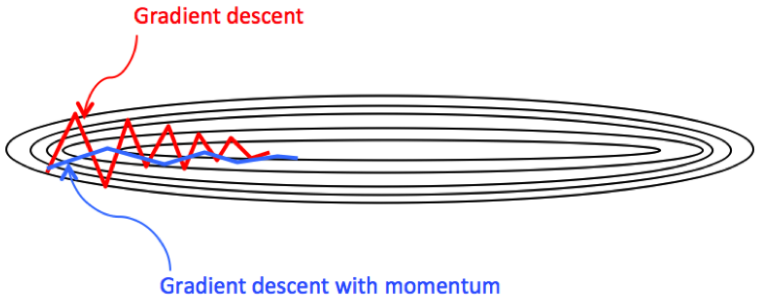


Figure V.1: Gradient Descent with Momentum

where β is the accumulation rate/momentum often set to 0.9 and α is the learning rate. This velocity is an exponential weights average of the gradients and it is vector valued. Steps will be largest when a sequence of gradients all point to the same direction. If $\beta = 0$ no momentum, if β is high only about previous gradient descents → Overshooting.

how quick

Important note: Momentum speeds up movement along directions of strong improvement (loss decrease) and also helps the network avoid local minima.

Nesterov momentum is when we use look ahead momentum, which means update theta over v then update the next v over the current updated theta and update the theta again → overcome local minima (not always good)

$$\tilde{\theta}^{k+1} = \theta^k + \beta v^k$$
$$v^{k+1} = \beta v^k - \alpha \nabla_\theta L(\tilde{\theta}^{k+1})$$
$$\theta^{k+1} = \theta^k + v^{k+1}$$

First you make a big jump in the direction of the previous accumulated gradient. Then measure the gradient where you end up and make a correction ⇒ Faster if the gradient always in the same direction, but risk of overshooting

RMSProp (Root Mean Squared Prop) divides the learning rate by an exponentially-decaying average of squared gradients. Which in other terms means: if we're having fluctuations, don't go so far in that direction.

element wise multiplication

$$s^{k+1} = \beta s^k + (1-\beta)[\nabla_\theta L \circ \nabla_\theta L]$$
$$\theta^{k+1} = \theta^k - \alpha \frac{\nabla_\theta L}{\sqrt{s^{k+1}} + \epsilon}$$

(Uncentered) variance of gradients: s_{k+1} is a second moment. Division by square gradients, if you divide in y-direction it will be large and if you divide in x-direction it will be small for Fig. V.1.

RMS-prop does not use momentum (the mechanism).

High variance → damp

⇒ Damping the oscillations for high-variance directions. RMSProp can use faster learning rate since it is less likely to diverge (speed up learning rate, second moment)

Adam Moment Estimation (ADAM) uses both the momentum and RMSProp.

$m^{k+1} = \beta_1 \cdot m^k + (1-\beta_1)\nabla_\theta L(\theta^k)$

$v^{k+1} = \beta_2 \cdot v^k + (1-\beta_2)[\nabla_\theta L(\theta^k) \circ \nabla_\theta L(\theta^k)]$

$\theta^{k+1} = \theta^k - \alpha \cdot \frac{m^{k+1}}{\sqrt{v^{k+1}}+\epsilon}$

However if k=0, the first estimates are very bad therefore we need a bias correction since $m^0 = 0$ and $v^0 = 0$.

Final update rules:

$m^{k+1} = \beta_1 \cdot m^k + (1-\beta_1)\nabla_\theta L(\theta^k)$

$v^{k+1} = \beta_2 \cdot v^k + (1-\beta_2)[\nabla_\theta L(\theta^k) \circ \nabla_\theta L(\theta^k)]$

$\hat{m}^{k+1} = \frac{m^{k+1}}{1-\beta_1^{k+1}}$ } mean centered bias correction

$\hat{v}^{k+1} = \frac{v^{k+1}}{1-\beta_2^{k+1}}$

$\theta^{k+1} = \theta^k - \alpha \cdot \frac{\hat{m}^{k+1}}{\sqrt{\hat{v}^{k+1}}+\epsilon}$

The m^{k+1} is the **first order momentum**, because it utilizes the **first moment**, which is the mean. It accumulates a "running mean" of the gradients, not because it averages over each individual batch. Hence, the notation m - mean.

The v^{k+1} is the **second order momentum**, because it utilizes the **second moment**, which is the variance. More accurately, the "**uncentered** variance".

Recall that the vairance is

$$Var(X) = E[(X-\mu)^2]$$

Here, we drop the mean μ, or just assume it is zero. Therefore, it is **uncentered**. So

$$\nabla_\theta L\left(\theta^k\right)^2 = \left[\nabla_\theta L\left(\theta^k\right) \circ \nabla_\theta L\left(\theta^k\right)\right]$$

$$Var(\nabla_\theta L\left(\theta^k\right)) \sim E[\nabla_\theta L\left(\theta^k\right)^2] \sim \beta_2 \cdot v^k + (1-\beta_2)\left[\nabla_\theta L\left(\theta^k\right) \circ \nabla_\theta L\left(\theta^k\right)\right]$$

Which is accumulated iteratively.

It accumulates a "running variance" of the gradients. Hence, the notation v - variance.

Notes:

- Momentum is a technique to utilize previous gradient knowledge, to boost the training steps in the correct direction, while **moment** is a statistical term, represnting the mean, variance and more.
- The moment term refers to a mean over some random-variable, therefore the momentum mechanism doesn't use the first order moment.

V.2 The Newton Method

We can approximate the loss function by a second order Taylor series expansion.

$L(\theta) \approx L(\theta_0) + (\theta-\theta_0)^T \nabla_\theta L(\theta_0) + \frac{1}{2}(\theta-\theta_0)^T H(\theta-\theta_0)$ (gradient, Hessian)

Differentiate and equate to 0: Update step: $\theta^* = \theta_0 - H^{-1}\nabla_\theta L(\theta)$

The computational complexity is $\mathcal{O}(k^3)$.

The Newton's method exploits the curvature to take a more direct route. We also got rid of the learning rate.

potential downsides: Faster convergence in terms of number of iterations "mathematical view" Approximating the inverse Hessian is highly computationally costly, not feasible for high-dimensional datasets

V.3 Other Optimization methods

- Broyden-Fletcher-Goldfarb-Shanno algorithm: family of quasi-Newton methods, approximation of the inverse of the Hessian: $\boldsymbol{\theta}^* = \boldsymbol{\theta}_0 - \mathbf{H}^{-1}\nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta})$
- Gauss-Newton: $x_{k+1} = x_k - H_f\left(x_k\right)^{-1} \nabla f\left(x_k\right)$ BUT second derivatives are often hard to obtain
- Levenberg: damped version of Gauss-Newton $(J_F(x_k)^T J_F(x_k) + \lambda I)(x_k - x_{k+1}) = \nabla f(x_k)$
- Levenberg-Marquardt: $\left(J_F\left(x_k\right)^T J_F\left(x_k\right) + \lambda \cdot \operatorname{diag}\left(J_F\left(x_k\right)^T J_F\left(x_k\right)\right)\right) \cdot \left(x_k - x_{k+1}\right) = \nabla f\left(x_k\right)$ Instead of a plain Gradient Descent for large λ, scale each component of the gradient according to the curvature: Avoids slow convergence in components with a small gradient

Standard: Adam
Fallback option: SGD with momentum
Newton, L-BFGS, GN, LM only if you can do full batch updates (doesn't work well for minibatches!!)
SGD is specifically designed for minibatch
When you can, use 2nd order method (it's just faster)
GD or SGD is not a way to solve a linear system!

```python
#ReLU: Forward
        outputs = np.maximum(x, 0)
        cache = x

#ReLU: Backward
        x = cache
        dx = dout
        dx[x < 0] = 0

#Affine layer: Forward
        x_reshaped = np.reshape(x, (x.shape[0], -1))
        out = x_reshaped.dot(w) + b

#Affine layer: Backward
        n = x.shape[0]
        dw = (np.reshape(x, (x.shape[0], -1)).T).dot(dout) / n
        dw = np.reshape(dw, w.shape)

        db = np.mean(dout, axis=0, keepdims=False)

        dx = dout.dot(w.T)
        dx = np.reshape(dx, x.shape)
```

VI Training Neural Networks I

VI.1 Learning rate

Learning rate: If too high, the model will overshoot during training and will perform poorly on any sample (both train and test data). If it is too low, the model will underfit the data. Underfitting also happens if you try to fit a linear model to non-linear data.

When you're far away from the optimum, the learning rate should be high and then decrease with time, that's why the learning rate decay is introduced.

Possible variants:

- **Fractional Decay:** $\alpha = \frac{1}{1+ \text{ decay-rate*epoch}} \cdot \alpha_0$
- **Step decay:** only every n steps, t is decay rate (often 0.5) $\alpha = \alpha - t \cdot \alpha$
- **Exponential Decay:** t is decay rate (t $<$ 1.0) $\alpha = t^{\text{epoch}} \cdot \alpha_0$

VI.2 Training

Definitions

- The training error comes from average minibatch error.
- Bias and Variance: Bias is the difference between the average prediction of our model and the correct value which we are trying to predict. A model with high bias pays very little attention to the training data and oversimplifies the model. It always leads to high error on the train and test data.
- Bias: Underfitting, error caused by model being too simple
- Variance is the variability of model prediction for a given data point or a value which tells us spread of our data. A model with high variance pays a lot of attention to the training data and does not generalize (Overfitting). These kinds of model perform very well on the training set, yet very poorly on the test set.
- Learning means generalization to unknown dataset (i.e., train on known dataset $\rightarrow$ test with optimized parameters on unknown dataset). Basically, we hope that based on the train set, the optimized parameters will give similar results on different data (i.e., test data).

Training schedule: Manually specify learning rate for entire training process
Manually set learning rate every n-epochs
How? Trial and error (the hard way) or Some experience (only generalizes to some degree)
Consider: #epochs, training set size, network size, etc.

Basic Recipe for Training
Given ground dataset with ground labels
- $\{x_i, y_i\}$

- x_i is the i^{th} training image, with labely_i
- Often dim(x) » dim(y) (e.g., for classification)
- i is often in the 100-thousands or millions

- Take network f and its parameters w, b
- Use SGD (or variation) to find optimal parameters w, b (Gradients from back propagation)

- **Training set ('train'):**
  - Use for training your neural network
  - During training: Train error comes from average minibatch error, Typically take subset of validation every n iterations (backward pass dominates forwas pass regarding costs)
- **Validation set ('val'):**
  - Hyperparameter optimization – Check generalization progres
- **Test set ('test'):**
  - Only for the very end
  - NEVER TOUCH DURING DEVELOPMENT OR TRAINING
- **Typical splits**
  - Train (60%), Val (20%), Test (20%)
  - Train (80%), Val (10%), Test (10%)

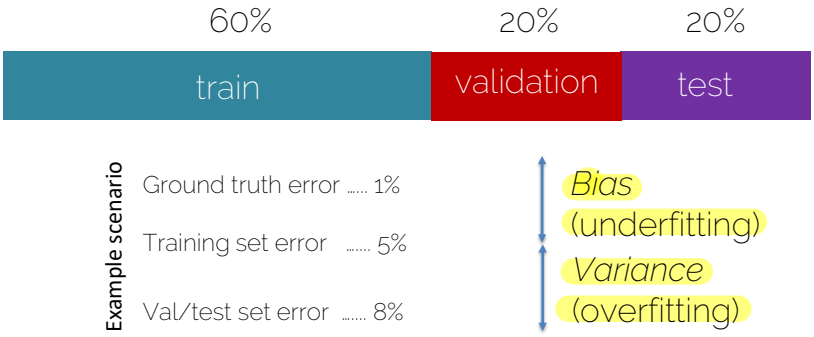


Figure VI.1: Ground truth error: errors in the labels (annotated manually), Training set error: underfitting issue, Val/test error: overfitting issue

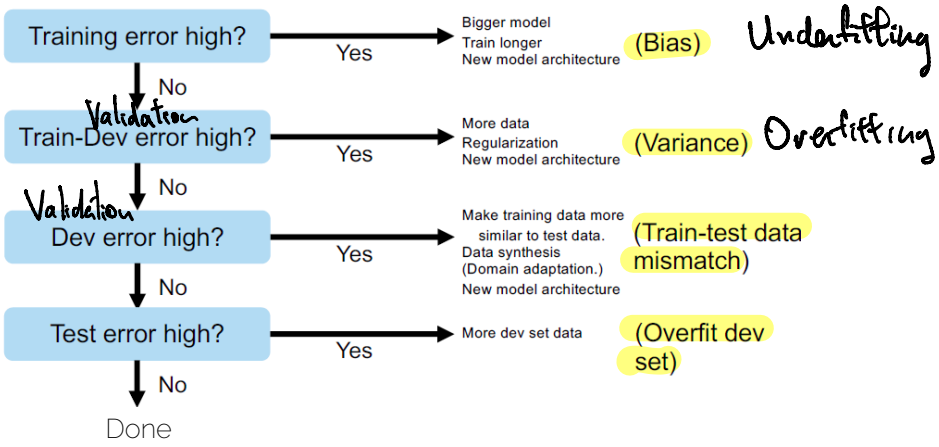


Figure VI.2: Decision tree for finding the error

Training error will go down during training, but validation error can go up because model will memorize data → stop in the middle and other methods

What does a good and bad training curve does look like?

Source: https://machinelearningmastery.com/learning-curves-for-diagnosing-machine-learning-model-performance/

- Train Learning Curve: Learning curve calculated from the training dataset that gives an idea of how well the model is learning. (loss VS epochs)

- Validation Learning Curve: Learning curve calculated from a hold-out validation dataset that gives an idea of how well the model is generalizing.

Accuracy and Loss curve will fluctuate: Apply smoothing function

Accurcay should go up / Loss should go down in the beginning very quickly → know quickly if model is working
gets slower at the end → change learning rate (smaller)

Summary

- Underfitting: Training and validation losses decrease even at the end of training
- Overfitting: Training loss decreases and validation loss increases
- Ideal Training: Small gap between training and validation loss, and both go down at same rate (stable without fluctuations)
- Bad Signs:
  - Training error not going down
  - Validation error not going down
  - Performance on validation better than on training set
  - Tests on train set different than during training
- Bad Practice:
  - Training set contains test data
  - Debug algorithm on test data
  - Never touch during development or training

VI.3 Hyperparameter tuning

(num layers, # weights, number iterations, learning rates, regularization, batch size ...)

Methods:

- Manually (most common)
- Grid search (define ranges for the hyperparameters)
- Random search (like grid search, but the points are picked at random and not sequentially)

Advice:

- If an iteration exceeds 500ms, things get dicey
- look for bottlenecks and estimate total time
- Start small, you can start with #layers / 5
- Check the loss and accuracy curves
- Only make one change at a time

Find a good Learning Rate

- Use all training data with small weight decay
- Perform initial loss sanity check e.g., log(C) for softmax with C classes
- Find a learning rate that makes the loss drop significantly (exponentially) within 100 iterations
- Good learning rates to try: 1e-1, 1e-2, 1e-3, 1e-4

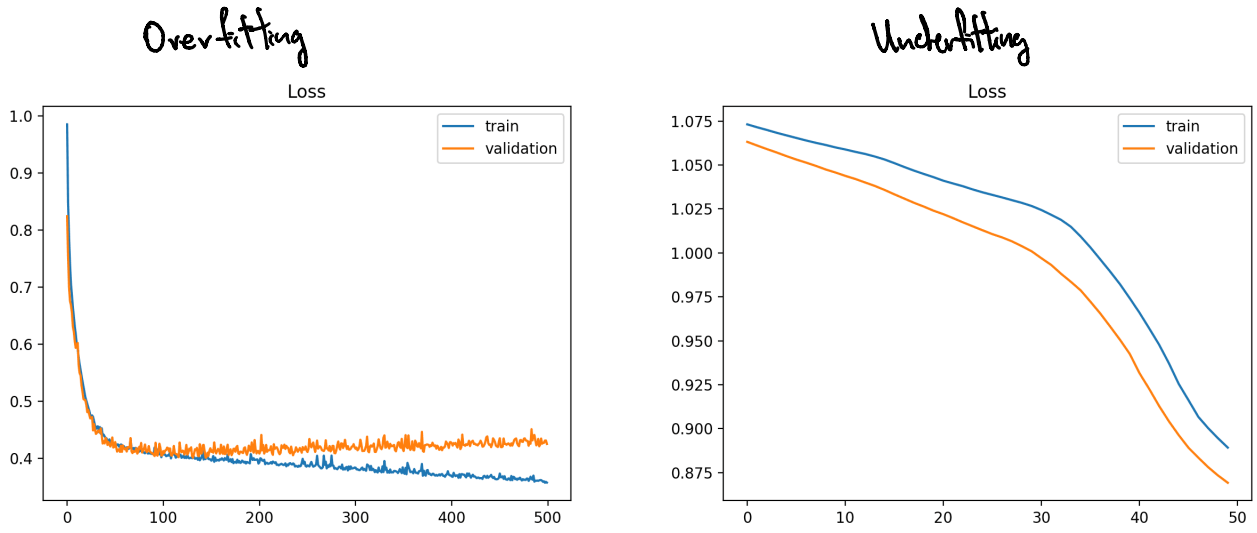


Figure VI.3: **Overfitting** (left): The plot of validation loss decreases to a point and begins increasing again. **Underfitting** (right): An underfit model may also be identified by a training loss that is decreasing and continues to decrease at the end of the plot. This indicates that the model is capable of further learning and possible further improvements and that the training process was halted prematurely.

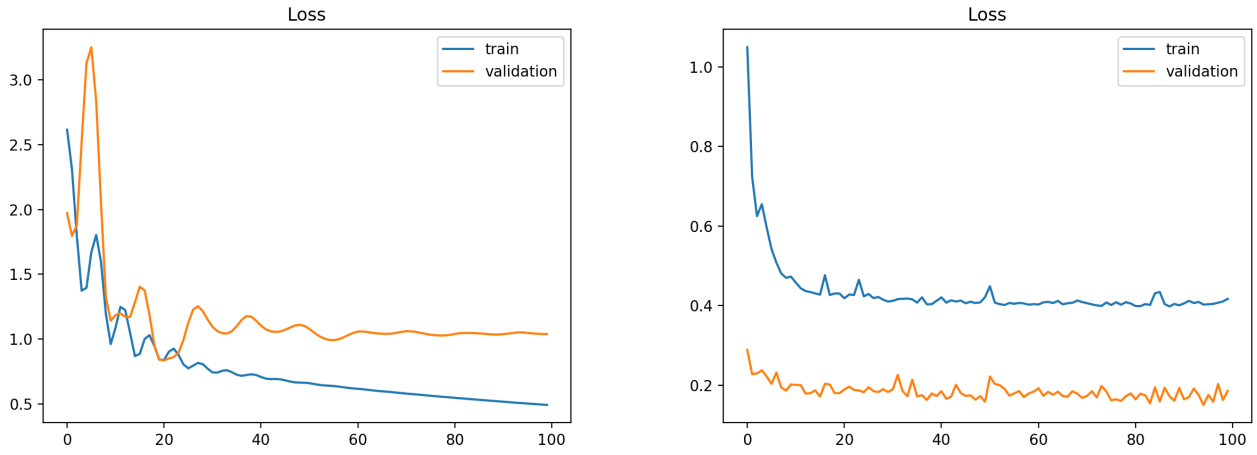
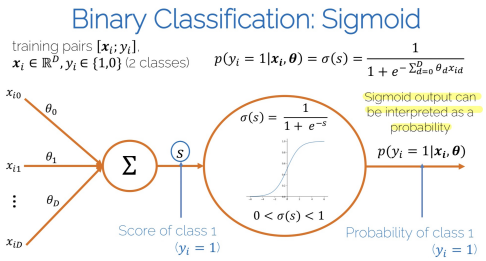
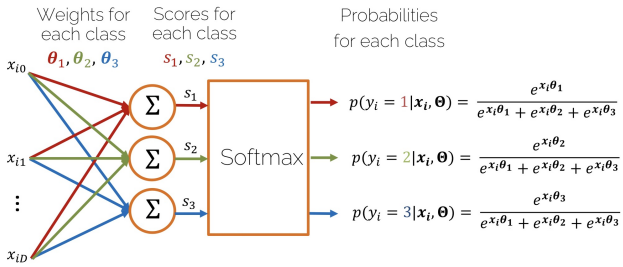


Figure VI.4: **Unrepresentative Train Dataset** (left): This situation can be identified by a learning curve for training loss that shows improvement and similarly a learning curve for validation loss that shows improvement, but a large gap remains between both curves.
Unrepresentative Validation Dataset (right): It may be identified by a validation loss that is lower than the training loss. In this case, it indicates that the validation dataset may be easier for the model to predict than the training dataset.





VII Training Neural Networks II

VII.1 Output and Loss Function

Loss Function is calculated on Output Layer (=Prediction) → What shape should the loss function have?

Naïve Losses

$$L_1 = \sum_{i=1}^{n} |y_i - f(x_i)| \quad \text{L1 Loss}$$

$$L_2 = \sum_{i=1}^{n} (y_i - f(x_i))^2 \quad \text{L2 Loss}$$

likely to suffer

L2 Loss	L1 Loss
Squared distances	Absolute differences
Prone to outliers	Robust (cost of outliers is linear)
Compute-efficient optimization	Costly to optimize
Optimum is the mean	Optimum is the median

does not suffer

Sigmoid: two classes, turns score into a probability $y_i \in \{1,0\}$

$$\sigma(s) = \frac{1}{1+e^{-s}}$$

$$p(y_i = 1|x_i, \theta) = \sigma(s) = \frac{1}{1 + e^{-\sum_{d=0}^{D} \theta_d x_{id}}}$$

$$s = \sum_{d=0}^{D} \theta_d x_{id}$$

Softmax: C classes, what is the score of a certain class k? $y_i \in \{1,2,\dots C\}$

$$p(y_i|x_i, \theta) = \frac{e^{s_{yi}}}{\sum_{k=1}^{C} e^{s_k}} = \frac{e^{x_i\theta_{yi}}}{\sum_{k=1}^{C} e^{x_i\theta_k}}$$

y_i: label (true class), C: number of classes, s: score of the class, s_{max}: maximal score

For numerical stability it's better to use:

probability of the true class

probability > 0

$$p(y_i|x_i, \theta) = \frac{e^{s_{yi}}}{\sum_{k=1}^{C} e^{s_k}} = \frac{e^{s_{yi} - s_{max}}}{\sum_{k=1}^{C} e^{s_k - s_{max}}}$$

sum to 1

$$\sigma(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

Cross Entropy Loss (Maximum Likelihood Estimate): Create a loss that pushes the probability into the right direction.

How wrong is my neural network? Loss function is low if prediction is correct.

$$L_i = -\log(p(y_i|x_i, \theta)) = -\log\left(\frac{e^{s_{yi}}}{\sum_k e^{s_k}}\right)$$

Total loss is averaged sum of losses.

Hinge Loss (Multiclass SVM Loss): Subtract score of the ground truth label from score of false predicted labels. Hinge Loss will be 0 if label is predicted correctly.

$$L_i = \sum_{k \neq y_i} \max(0, s_k - s_{y_i} + 1)$$

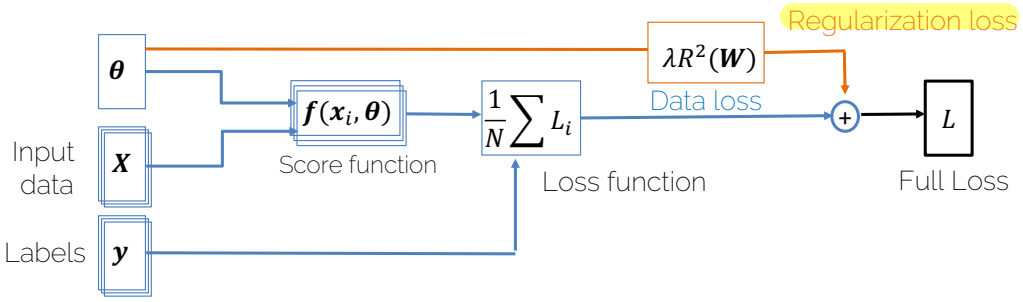
score del ground truth

score de los scores que no son ground truth

Multiclass Losses: Hinge vs Cross-Entropy

Both Hinge Losses could be the same, even if one model is better. Cross-Entropy shows the difference. The CE loss always wants to improve (it is never 0, due to the exp() effect) whereas the Hinge loss saturates.

Loss in Compute Graph



Want to find optimal $\boldsymbol{\theta}$. (weights are unknowns of optimization problem)

- Compute gradient w.r.t. $\boldsymbol{\theta}$.
- Gradient $\nabla_{\boldsymbol{\theta}} L$ is computed via backpropagation

I2DL: Prof. Niessner, Prof. Leal-Taixé 36

Summary

- Score Function: $\boldsymbol{s} = \boldsymbol{f}(\boldsymbol{x}_i, \boldsymbol{\theta})$
- Data Loss:
  - Cross Entropy: $L_i = -\log\left(\frac{e^{sy_i}}{\sum_k e^{s_k}}\right)$
  - SVM: $L_i = \sum_{k \neq y_i} \max(0, s_k - s_{y_i} + 1)$
- Regularization Loss: e.g. $L2$ -Reg: $R^2(\boldsymbol{W}) = \sum \boldsymbol{w}_i^2$
- Full Loss: $L = \frac{1}{N}\sum_{i=1}^{N} L_i + \lambda R^2(\boldsymbol{W})$ (λ: how much weight do I give to my regularization)
- Full Loss = Data Loss + Reg Loss

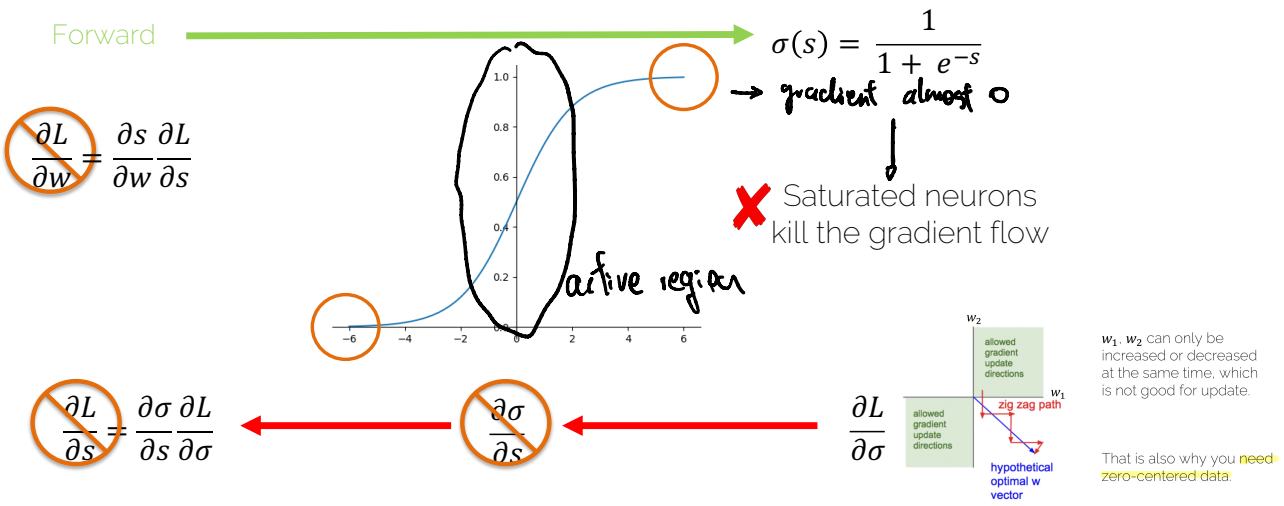
VII.2 Activation Functions

Bring non-linear motion in hidden units

Sigmoid Activation

$$\sigma(s) = \frac{1}{1 + e^{-s}}$$

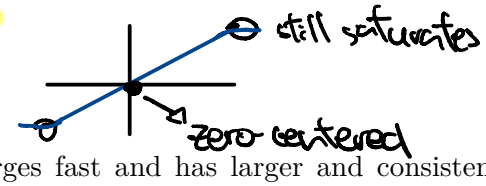
This type of activation function is not advised to use since it has a really narrow active region. If value is really high, gradient is very low and value will be 'killed' during backpropagation. The sigmoid output is



not zero centered. All the weight updates will be either both positive or both negative. w_1 and w_2 can only be increased or decreased at the same, which would mean that the path would follow a zig-zag manner. We need zero-centered data. The different weights needs to can have different sign

Tanh Activation: This function is zero-centered yet it saturates.

$$\sigma(s) = \tanh(s)$$

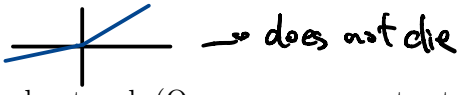


ReLU: Rectified Linear Unit: It does not saturate, it converges fast and has larger and consistent gradients.

$$\sigma(s) = \max(0, x)$$

Dead ReLU: Output = 0 all the time, we can't get out of that problem. That's why it's advised to add the slightly positive biases term (makes it likely that they stay active for most inputs: $f\left(\sum_i w_i x_i + b)\right)$, or use Leaky ReLU instead:

$$\sigma(s) = \max(0.01x, x)$$



Parametric ReLU: Parameter α instead of 0.01, that is trained of neural network (One more parameter to backprop into)

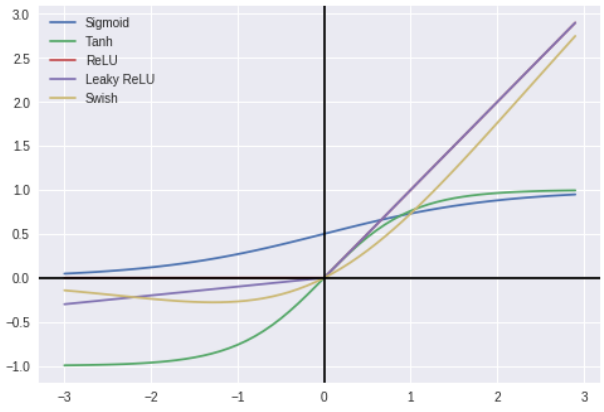
$$\sigma(s) = \max(\alpha x, x)$$

Maxout units: Train parameters with other parameters of the network. Piecewise linear approximation of a convex function with N pieces → Generalization of ReLUs, Linear Regimes, Does not die, does not saturate BUT Increases number of parameters

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

Summary

- Sigmoid is not really used.
- ReLU is the standard choice.
- Second choice are the variants of ReLU or Maxout
- Recurrent nets will require TanH or similar.



ACTIVATION FUNCTION	EQUATION	RANGE
Linear Function	$f(x) = x$	$(-\infty, \infty)$
Step Function	$f(x) = \begin{cases} 0 \text{ for } x < 0 \\ 1 \text{ for } x \geq 0 \end{cases}$	$\{0, 1\}$
Sigmoid Function	$f(x) = \sigma(x) = \frac{1}{1+e^{-x}}$	$(0, 1)$
Hyperbolic Tanjant Function	$f(x) = \tanh(x) = \frac{(e^x - e^{-x})}{(e^x + e^{-x})}$	$(-1, 1)$
ReLU	$f(x) = \begin{cases} 0 \text{ for } x < 0 \\ x \text{ for } x \geq 0 \end{cases}$	$[0, \infty)$
Leaky ReLU	$f(x) = \begin{cases} 0.01 \text{ for } x < 0 \\ x \text{ for } x \geq 0 \end{cases}$	$(-\infty, \infty)$
Swish Function	$f(x) = 2x\sigma(\beta x) = \begin{cases} \beta = 0 \text{ for } f(x) = x \\ \beta \to \infty \text{ for } f(x) = 2\max(0, x) \end{cases}$	$(-\infty, \infty)$

Figure VII.1: https://towardsdatascience.com/comparison-of-activation-functions-for-deep-neural-net

VII.3 Weight Initialization

issues { Small random Numbers → outputs go to zero (small output)
Big random Numbers → output saturated to -1 - 1 (saturated by activation function)

Initialization is extremely important because is not guaranteed to reach the optimum for different starting points.

If w=0 at the start then all the hidden units are all going to compute the same function (grads will be the same) so it would be as if you'd have just one neuron (No symmetry breaking).

So, initialize weights randomly (Gaussian with zero mean and standard deviation 0.01). Yet if they are small numbers then we would face vanishing gradients (from small outputs of layer l) and if the weights would be big numbers we would also face vanishing gradients caused by saturated activation function. (-1, 1)

Use **Xavier Initialization** which ensures the variance of the output is the same as the input → Gaussian with

$E[X^2] = Var[X] + E[X]^2$ if X, Y are independent

$Var[X,Y] = E[X^2Y^2] - E[XY]^2$

$E[XY] = E[X]E[Y]$

- Mean: $\mu = 0$
- Variance:

$$\text{Var}(s) = \text{Var}\left(\sum_i^n w_i x_i\right) = \sum_i^n \text{Var}\left(w_i x_i\right)$$
$$= \sum_i^n \left[E\left(w_i\right)\right]^2 \text{Var}\left(x_i\right) + E\left[\left(x_i\right)\right]^2 \text{Var}\left(w_i\right) + \text{Var}\left(x_i\right)\text{Var}\left(w_i\right)$$
$$= \sum_i^n \text{Var}\left(x_i\right)\text{Var}\left(w_i\right) = n(\text{Var}(w)\,\text{Var}(x))$$

(0 mean, 0 mean)

- Ensure that the variance of the output is the same as the input: $\text{Var}(s) = \text{Var}(x) \to n \cdot \text{Var}(w)\,\text{Var}(x) = \text{Var}(x) \to$
  - Tanh: $\text{Var}(w) = \frac{1}{n}$
  - ReLU (Xavier/2 Initialization): $\text{Var}(w) = \frac{1}{n/2} = \frac{2}{n}$

Xavier init. with ReLU (Kaiming Init) kills half of the data. Solution →

- Notice: n is the number of input neurons for the layer of weights you want to initialized. This n is not the number N of input data $X \in R^{N \times D}$ For the first layer $n = D$
- Side note: $E[X^2] = \text{Var}[X] + E[X]^2$ and if X,Y are independent: $\text{Var}[XY] = E[X^2Y^2] - E[XY]^2$, $E[XY] = E[X]E[Y]$

Image Classification	Output Layer	Loss function
Binary Classification	Sigmoid	Binary Cross entropy
Multiclass Classification	Softmax	Cross entropy

Other Losses:
SVM Loss (Hinge Loss), L1/L2-Loss

Initialization of optimization
- How to set weights at beginning

Use ReLU and Xavier/2 initialization

VIII Training Neural Networks III

VIII.1 Data Augmentation

A classifier has to be invariant to a wide variety of transformations. So to help it, we can synthesize data simulating plausible transformations.

Examples:

- Brightness and Contrast: More robust to illumination changes
- Random Crops (and Resizing): Sometimes image only half on the picture → not showing the full image helps network to handle cropped images
  - Training random crops: Pick a random L, resize training image: short side L, randomly sample crops
  - Testing fixed set of crops: Resize image at N scales, 10 fixed crops
- Flips
- Rotations
- Combinations of the above

data augmentation is a part of the network design

Use the same data augmentation when comparing two networks (part of your network design).
We have online and offline data augmentation. The former is done as a pre-processing step to increase the size of the dataset. The latter happens when we apply transformations in mini-batches and then feed it to the model.

VIII.2 Advanced Regularization

$$\theta_{k+1} = \theta_k - \underbrace{\xi}_{\text{learning rate}} \underbrace{\nabla_\theta(\theta_k, x, y)}_{\text{gradient}} - \underbrace{\lambda \theta_k}_{\text{gradient of L-2 regularization}}$$

Early Stopping
Stop at the point where the validation error stops decreasing. You can have an impatience parameter set. For example if impatience is 3, then the iterative training will stop when there's 3 consecutive losses that are increasing (overfitting).

Bagging and Ensemble Methods
Ensemble: Train multiple models average their results (e.g. different algorithm for optimization or change the objective function/loss function). If errors are uncorrelated, the expected combined error will decrease linearly with the ensemble size.

Bagging: Uses k different datasets (can be overlapping, drawn from the bigger dataset). Models specialize on a specific thing what is common in the dataset (e.g. white and black dogs). Problem: need to train 3 networks.

Dropout

Dropout

- Forward: Disable a random set of neurons (typically p=0.5). For each neuron there's a p possibility that it's dropped (nodes are not used in the forward pass) → only p of the capacity, only half of the features are used, decision needs to be made with fewer information.
- Backward: reprogram some of the active neurons to detect some of the ignored features → other neurons are responsible of detecting the features, one than more way to detect features → redundant representations, base your scores on more features → considerable as a model ensemble (model of inactive neurons and model of active neurons, trained on a different set of data (mini-batch) and with SHARED parameters)
  ⇒ Reducing the co-adaption between neurons (neurons depending on other neurons doing their job)
- Testing: All neurons are turned on, no dropout → Conditions at train and test time are not the same
  say $z = (\theta_1 x_1 + \theta_2 x_2) \cdot p$ (dropout probability)
  Expectation value for z: $E[z] = \frac{1}{4}(\theta_1 0 + \theta_2 0 + \theta_1 x_1 + \theta_2 0 + \theta_1 0 + \theta_2 x_2 + \theta_1 x_1 + \theta_2 x_2) = \frac{1}{2}(\theta_1 x_1 + \theta_2 x_2)$
  1/2 corresponds to the Dropout probability $p = 0.5$ → Weight scaling inference rule
- Efficient bagging method with parameter sharing, dropout reduces the effective capacity of a a model → larger models, more training time

VIII.3 Batch Normalization

Normally used as Regularization technique
Goal: Activation do not die out
Wish: Unit Gaussian activations (in our example) → Unit gaussian as Input and unit Gaussian as Output
Mean of your mini-batch examples over feature k → force unit Gaussian in each dimension of the features (Gaussian as input and as output): $\hat{x}^{(k)} = \frac{x^{(k)} - E[x^{(k)}]}{\sqrt{\text{Var}[x^{(k)}]}}$

For NN in general: BN normalizes the mean and the variance of the inputs to your activation functions
A layer to be applied after Fully Connected (or Convolutional) layers and before non-linear activation functions.

- A layer to be applied after Fully Connected (or Convolutional) layers and before non-linear activation functions

Implementation

1. Normalize $\hat{x}^{(k)} = \frac{x^{(k)} - E[x^{(k)}]}{\sqrt{\text{Var}[x^{(k)}]}}$ → fix values
2. Allow network to change the range: $y^{(k)} = \gamma^{(k)}\hat{x}^{(k)} + \beta^{(k)}$ where γ and β get optimized during backprop → learned as any other hyperparameter, the network can learn to undo the normalization: $\gamma^{(k)} = \sqrt{\text{Var}[x^{(k)}]}$ and $\beta^{(k)} = E[x^{(k)}]$

Is it okay to treat dimensions separately? (Computed for each k)

- Shown empirically that even if features are not correlated, convergence is still faster with this method
- All the biases of the layers before BN can be set to zero since they will be canceled out by BN anyway. BN shifts the activation by their mean values and any constant will get canceled.

Train-time: mean and variance is taken over the mini-batch
Test-time: compute the mean and variance by running an exponentially weighted averaged across training mini-batches. σ^2_{test} and μ_{test}:

$$Var_{running} = \beta_m * Var_{running} + (1 - \beta_m) * Var_{minibatch}$$

$$\mu_{running} = \beta_m * \mu_{running} + (1 - \beta_m) * \mu_{minibatch}$$

- Sometimes, β_m is called as "momentum", which I find falsely so.
- By default, but could be manually changed as a hyperparameter $\beta_m = 0.99$. Therefore, the new calculated mean and variance over the current batch have but little effect on the running variables.

⇒ Very deep nets are much easier to train: more stable gradients, a much larger range of hyperparameters works similarly when using BN

Drawbacks: As we reduce the batch size (< 8), the statistics for mean and variance get less and less accurate and BN starts to deliver not so good results → use Layer Norm, Instance Norm, Group Norm (the error stays quite constant despite the batch size)

The non-linearity of the Batchnorm during training is due to the fact that each batch out of the whole dataset is normalized by different values (mean and std). During the inference time (testing), it is considered as a linear function, since those values are a constant (running-mean and running-var), that do not depend on the test input. Improved Error (lower error)

VIII.4 Other Normalizations

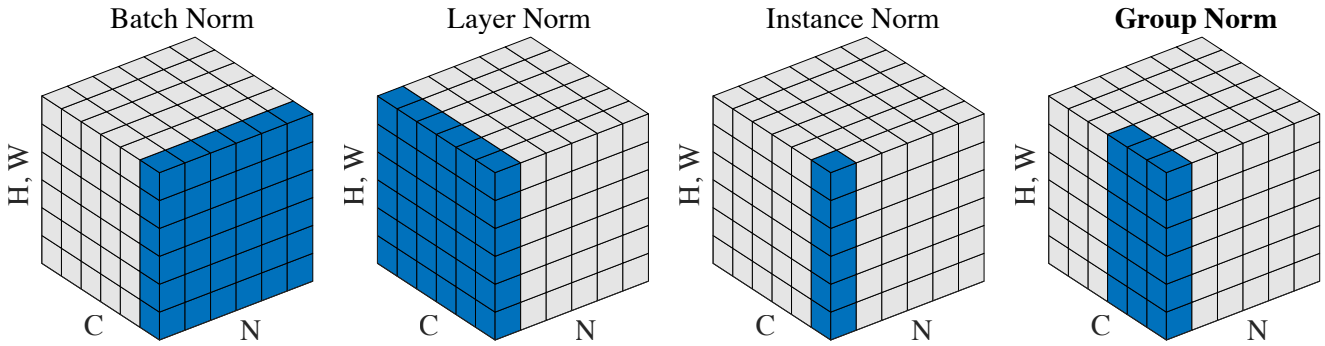


Figure VIII.1: Feature map: spatial sizes H and W, Number of channels C, Number of training samples N. The pixels in blue are normalized by the same mean and variance, computed by aggregating the values of these pixels.

VIII.5 Recap

Why not simply more Layers?

- We cannot make networks arbitrarily complex
- Why not just go deeper and get better? – No structure, It is just brute force, Optimization becomes hard, Performance plateaus/drops

IX Introduction to CNNs

Using only FC layers to build an architecture is impractical since it is close to brute-force and there's a lot of weights that need to be trained:

- $[3, 5, 5]$ image and 3 neuron in FC 75x3 weights need to be trained
- image sizes are 1k x 1k then we have 3 billion weights

Also there's no structure, optimization becomes hard and the performance drops.

Instead use a different structure: layer with structure, weight sharing (share the same weights for different parts of the image)

IX.1 Convolution

A convolution is an application of a filter to a function: $f * g = \int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau$ (The 'smaller' function is typically called the filter kernel)

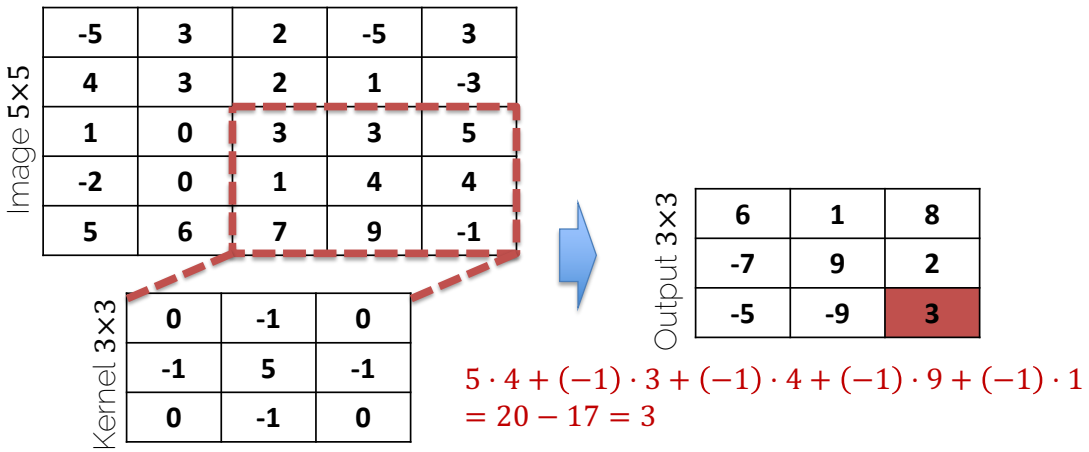


Figure IX.1: The filter kernel (g) is slid throughout the image (f) and computes the single values in the output data ($f * g$)

In the discrete case, there's unassigned values at the boundaries and for those we can: shrink (smaller f * g) or pad (often with 0s).
Each kernel gives us a different image filter → learn these filters

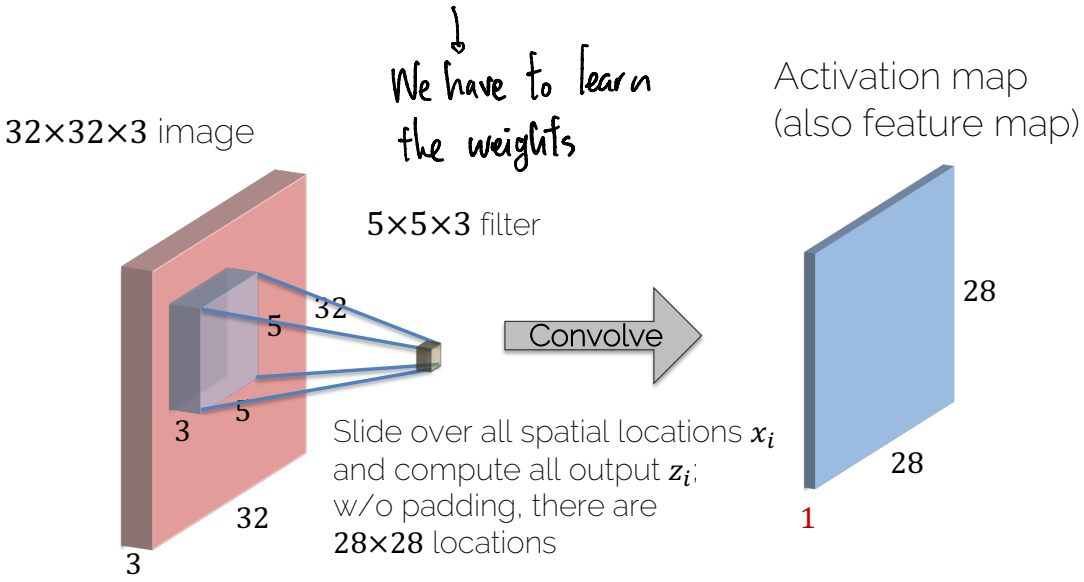
$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$ Edge Detection; $\frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$ Box ~~Mean~~ Blur; $\frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$ Gaussian blur; $\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$ Sharpen

$\begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}$ Sobel Filter, vertical edge detection; $\begin{bmatrix} 3 & 0 & -3 \\ 10 & 0 & -10 \\ 3 & 0 & -3 \end{bmatrix}$ Scharr filter, vertical edge detection

Convolutions on RGB Images

The depth of the image must match the depth of the filter (3 for a RGB image)

Create one output for every convolution: $z_i = \boldsymbol{w}^T \boldsymbol{x}_i + b$, where $\boldsymbol{w}^T$ and $\boldsymbol{x}_i$ are of size [width * height * depth] x 1



IX.2 Convolution Layer

In a convolution layer we can apply many filters with different weights, so we can have many activation maps → form a Convolution Layer. So a basic layer would be {filter width, filter height, number of different filter banks}.

Each filter captures a different characteristic such as: edges, circles, squares, etc.

IX.3 Dimensions of a Convolution Layer

N: Input width/height
F: Filter/Kernel size
S: Stride
P: Padding

Stride

- is the jump the filter does while sliding
- Output is: $(\frac{N-F}{S}+1) \times (\frac{N-F}{S}+1)$
- Fractions are illegal (Filter must not be outside the image partly). So check the stride.

Padding

- Since the sizes get too small too quickly, we use padding. Also like this we can use the corner pixels more than once.
- Pad the whole picture with numbers (mostly zeros) around
- Output: $(\left\lfloor \frac{N+2 \cdot P-F}{S} \right\rfloor + 1) \times (\left\lfloor \frac{N+2 \cdot P-F}{S} \right\rfloor + 1)$
- Different paddings:
  - Valid: no padding
  - Same: output size = input size, set $P = \frac{F-1}{2}$

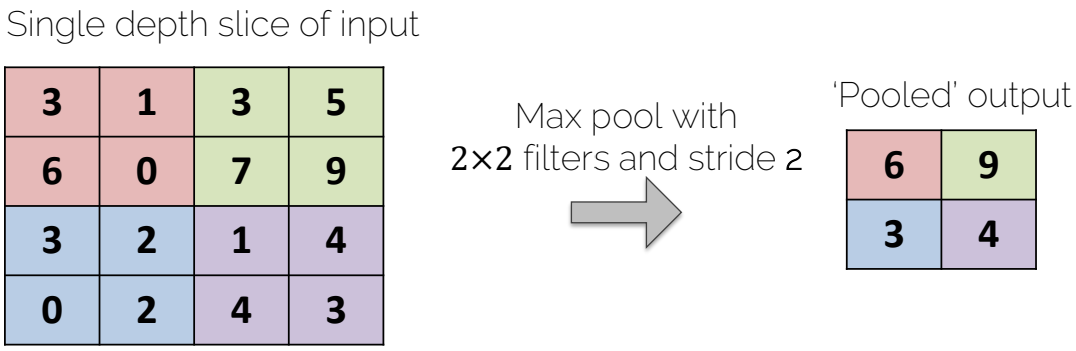
Number of parameters $= F \cdot F \cdot \text{Depth} \cdot \text{Number of Filters}$
Weight tensor: $(\text{Depth}, \text{Number of Filters}, F, F)$

Note: Each kernel filter gets 1 bias.

IX.4 Convolutional Neural Network (CNN): Pooling

Used as a downsampling layer. It picks the strongest activation in a region. Pooling layer goes directly after convolution layer:

- Conv Layer = 'Feature Extraction': Computes a feature in a given region
- Pooling Layer = 'Feature Selection': Picks the strongest activation in a region



Input: $W_{in} \times H_{in} \times D_{in}$ Two hyperparameters: Spatial filter extend F and Stride S

Output: $W_{out} \times H_{out} \times D_{out}$ where $W_{out} = \frac{W_{in}-F}{S} + 1$ and $H_{out} = \frac{H_{in}-F}{S} + 1$ and $D_{out} = D_i n$.
This is a fixed function, so it does not contain parameters.

Various pooling types:

- Max Pooling
- Average Pooling
- Global Average Pooling

To finalize the CNN we add a FC layer (One or two FC layers typically)

IX.5 Receptive Field

Receptive field is the size of the region in the input space that a pixel in the output space is affected by; the spatial extent of the connectivity of a convolutional filter. What is the relationship between the input and the output? For example: 3x3 receptive field: 1 output pixel is connected to 9 input pixels.

$$r_k = r_{k-1} + (\prod_{i=1}^{k-1} s_i) \cdot (f_k - 1) \; for \; k \geq 2$$

Where s is the stride and f is the kernel size. Also, r_1 is the kernel size of the first layer. **Note** that we start from the input and deeper, and not from the requested layer back!

For FC layers, the receptive field is the ENTIRE image. Since each output neuron of that layer is connected to each neuron of the previous one, where each layer represents the entire output, whether it is downscaled or upscaled.

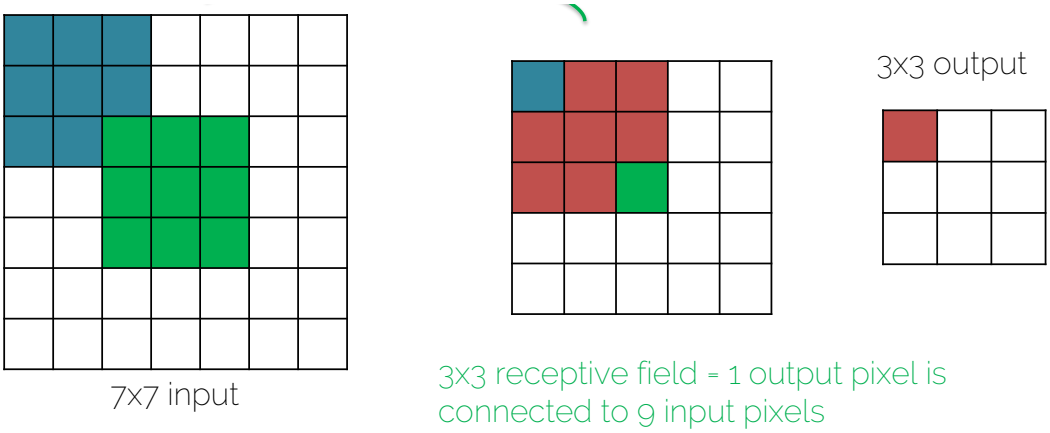


Figure IX.2: Finally: 5x5 receptive field on the original input: one output value is connected to 25 input pixels

Batch Norm for CNN:

$\mu_j = \frac{1}{N}\sum_{i=1}^{N} x_{i,j}$

$\sigma_j^2 = \frac{1}{N}\sum_{i=1}^{N}(x_{i,j} - \mu_j)^2$

$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}$

$y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j$

We compute minibatch mean and variance for each channel C.

Spatial batch norm: Compute mean and variance of each channel.

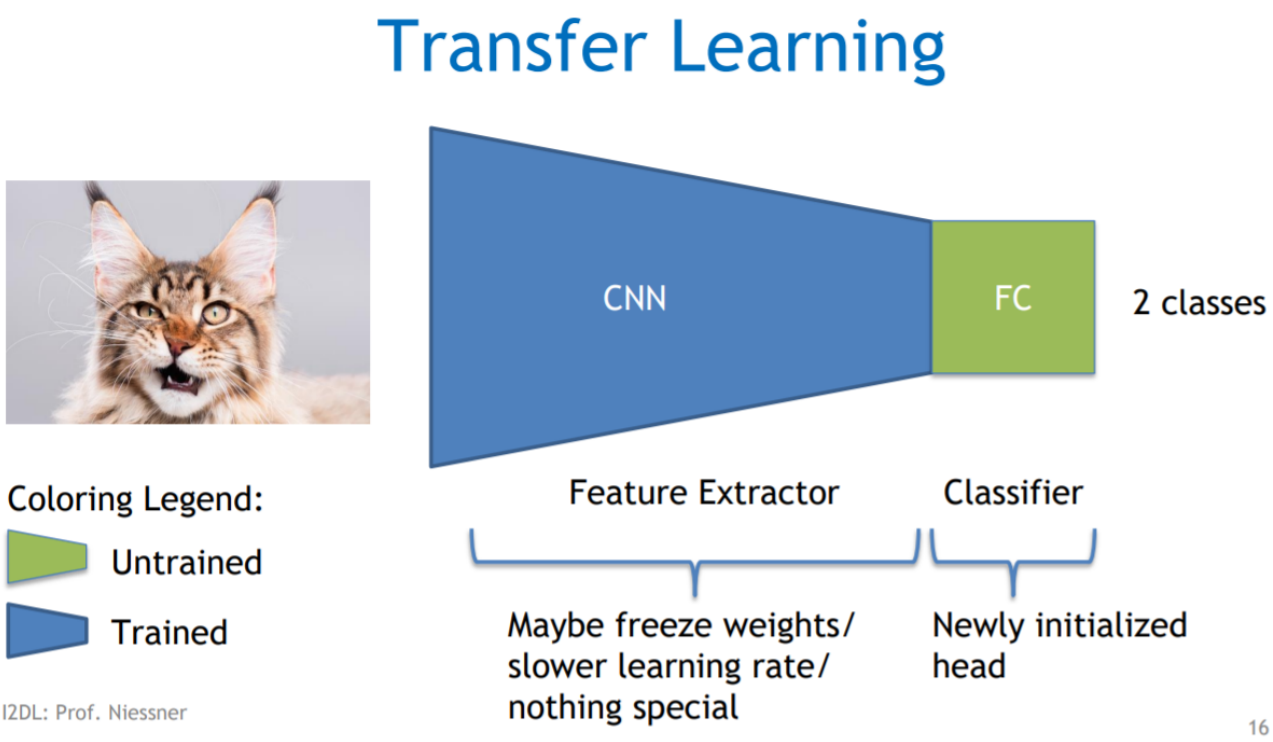
IX.6 Dropout for CNN

Standard: randomly drop a unit with a certain probability. This does not improve performance in CNN.
Spatial: randomly sets entire feature maps to zero.

IX.7 Transfer Learning

- The main idea is to **train a new classifier for a new problem, with an existing model.**
- The problems we try to overcome:
  - Training a Deep Neural Network requires a HUGE amount of data.
  - Much data might not be available, or it is expensive.
- What are the basic guidelines for such thing?
  - The new problem must be similar to the one the current model already aims to solve. (i.e - taking an already trained model that recognizes dogs, and make it also recognize cats).
  - Can we transfer knowledge from the previous problem to the new one?
  - can we re-use parts of the already trained network?

- The last bull point is the key idea of this method. A good modern classifier is constructed from two parts - CNN and FC, where the first one is referred as a **Features Extractor** and the latter as **The Classifier**.
- In order to use the existing knowledge:
  - Let us drop the FC part, and replace it with a new fresh untrained one.
  - For the CNN part we might want to:
    * Freeze the CNN weights completely during the new training session.
    * Slower the learning rate of the CNN only, so it learns a bit more from the new data, but doesn't make a significant impact, that might hurt the efficiency of the model.
    * let it train as usual.



X Popular CNN Architectures

Performance Metrics in ImageNet:

- **Top-1 Score**: check if a sample's top class is the same as its target label
- **Top-5 Score**: check if the label is in the first 5 predictions.
- **Top-5 Error**: percentage of test samples for which the correct class was not in the top 5 predicted classes.

X.1 LeNet

Firstly used for handwritten digits. Has the following architecture:

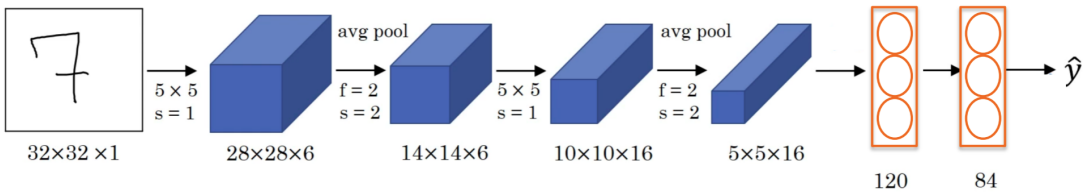


Figure X.1: LeNet Architecture: Conv -> Pool -> Conv -> Pool -> Conv -> FC

- Apply valid convolution: size shrinks (reduced by two pixels on each side)
- 6 5x5 convolution filters used in first layer (6 filters as the depth of the convolution obtained is 6)
- Average pooling is used (now: Max pooling much more common)
- Reduce first to 120, then to 84
- Tanh/sigmoid activation is used (not common now)
- Has 60k parameters
- As we go deeper: width, height go down and number of filters go up

X.2 AlexNet

- Same convolutions and paddings
- Similar to LeNet but ~1000 times bigger
- ReLU instead of tanh/sigmoid
- 60m parameters

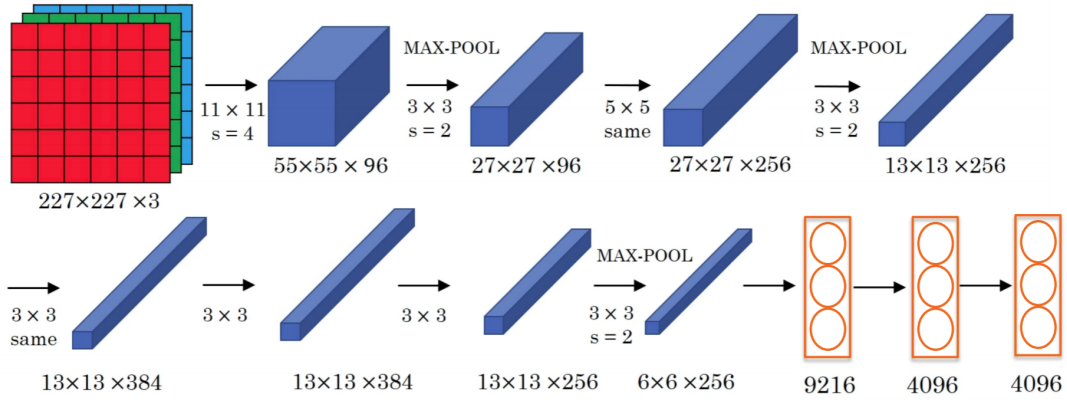


Figure X.2: AlexNet Architecture

X.3 VGGNet

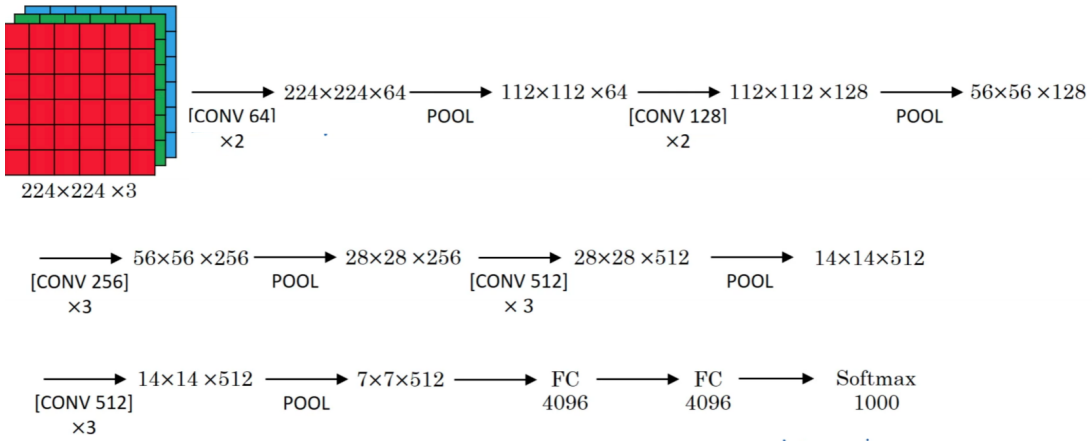


Figure X.3: VGGNet Architecture: Conv -> Pool -> Conv -> Pool -> Conv -> FC

- Same padding
- CONV = 3x3 filters with stride 1, same convolutions
- MAXPOOL = 2x2 filters with stride 2
- 138m parameters
- It's large but simplicity makes it appealing
- As we go deeper, width and height go down and number of filters up

X.4 Skip Connections: Residual Block

Since we add more and more layers, training becomes harder and we face vanishing or exploding gradients. Here skip connections are useful to train very deep models. Usually same padding is used since we need same dimensions (you can convert the dims with a matrix of learned weights or zero padding).

Note, that since we add the input of x^{L-1} into x^{L+1}, this addition **can not** make the neural-network go worse, since it could just learn to skip the next layers, if they do not contribute. That way, as it appeared in **ResNet**, we can harness the power of depth, while usually a deeper network doesn't generalize well.

- Two layers: $x^{L-1} \to x^{L} \to x^{L+1}$
- Main path: $x^{L+1} = f(W^{L+1}x^{L} + b^{L+1})$
- Add skip connection: $x^{L+1} = f\left(W^{L+1}x^{L} + b^{L+1} + x^{L-1}\right)$

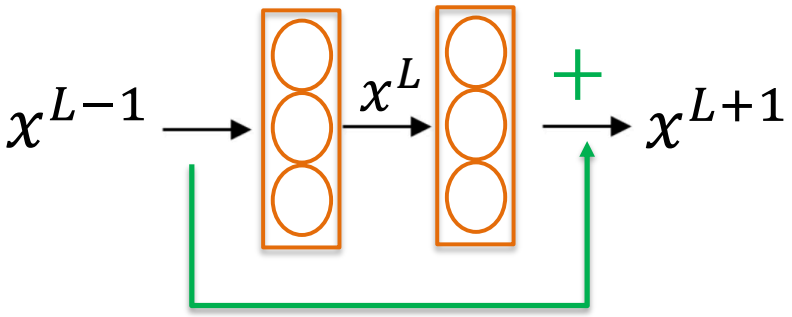


Figure X.4: Skip connection: Usually use a same convolution since we need same dimensions (Otherwise we need to convert the dimensions with a matrix of learned weights or zero padding)

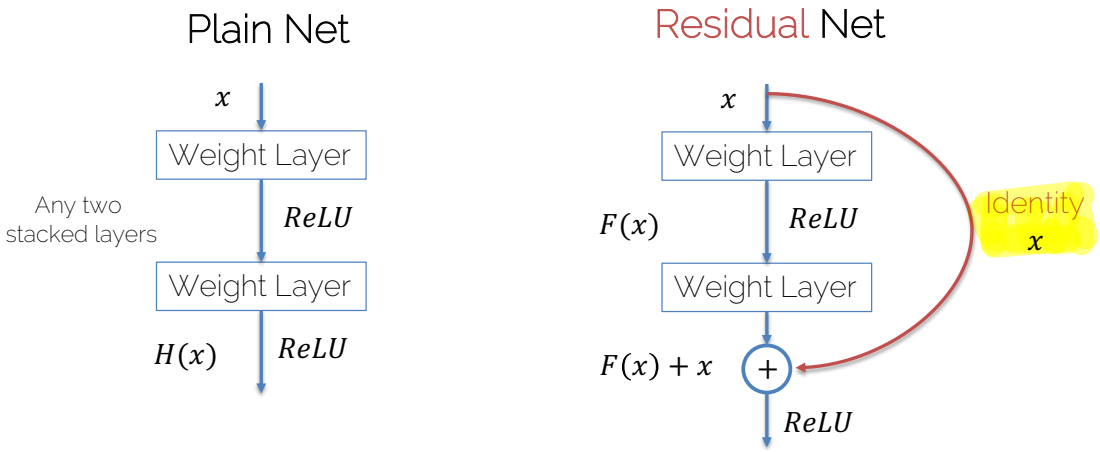


Figure X.5: Residual Net

We wouldn't call the skip connection itself as nonlinear then, because its not introducing new non-linearity.

X.5 ResNet (Residual Networks)

- Xavier/2 initialization
- SGD + Momentum(0.9)
- lr=0.1 (lr/10 when plateau)
- Minibatch: 256
- Weight decay: 1e-5
- No dropout
- 60m parameters

- If we go deeper, the performance starts to degrade at some point. (Deeper ResNets perform better than not so deep ResNets, not the case with PlainNets)

Why do ResNets (skip connection) work?

- The same values are kept and a non-linearity is added (good if previous values are 0)
- The identity is easy for the residual block to learn
- Guaranteed it will not hurt the performance, can only improve

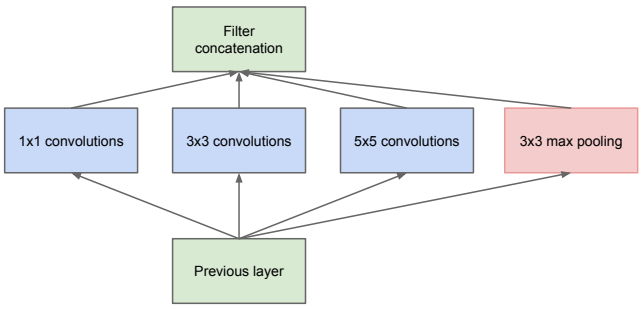
X.6 Inception Layer: 1x1 Convolution

1x1 Convolution: No padding needed, same output size. Just scales the input. (Same as having a 3 neuron fully connected layer)

Use more convolutional layers: Use it to shrink the number of channels and adds a non-linearity → one can learn more complex functions

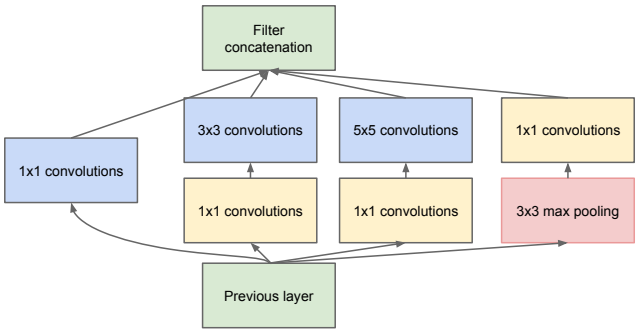
Inception layer:

- You can use all filter sizes instead of one at the same time → Extreme computational
- Implementation: All same convolutions → output spatial size is the same. For 3x3 max pooling stride of 1. Then concatenate at the end. This is not sustainable.
- That's why we use 1x1 convolutions. First do 1x1 convolution then the filter. It will reduce the number of multiplications needed
- Computational cost: Reduction of multiplications by 1/10 with 1x1 convolution
- Using the Inception Layer creates the GoogLeNet



(a) Inception module, naïve version

Figure X.6: Even a modest number of 5x5 convolutions can be prohibitively expensive on top of a convolutional layer with a large number of filters.



(b) Inception module with dimensionality reduction

Figure X.7: 1x1 convolutions are used to compute reductions before the expensive 3x3 and 5x5 convolutions. No blow-up in computational complexity.

X.7 XceptionNet

- „Extreme version of Inception": applying (modified) Depthwise Separable Convolutions instead of normal convolutions

- 36 conv layers, structured into several modules with skip connections
- Outperforms Inception Net V3
- Normal convolutions act on all channels

Depth-wise separable convolutions (DSC): Filters are applied only at certain depths of the features. Normal convolutions have groups set to 1 (act on all depths), the convolutions used in this image have groups set to 3

```
class torch.nn.Conv2d(in_channels, out_channels, kernel_size,
    stride=1, padding=0, groups=3)
class torch.nn.ConvTranspose2d(in_channels, out_channels, kernel_size,
    stride=1, padding=0, groups=3)
```

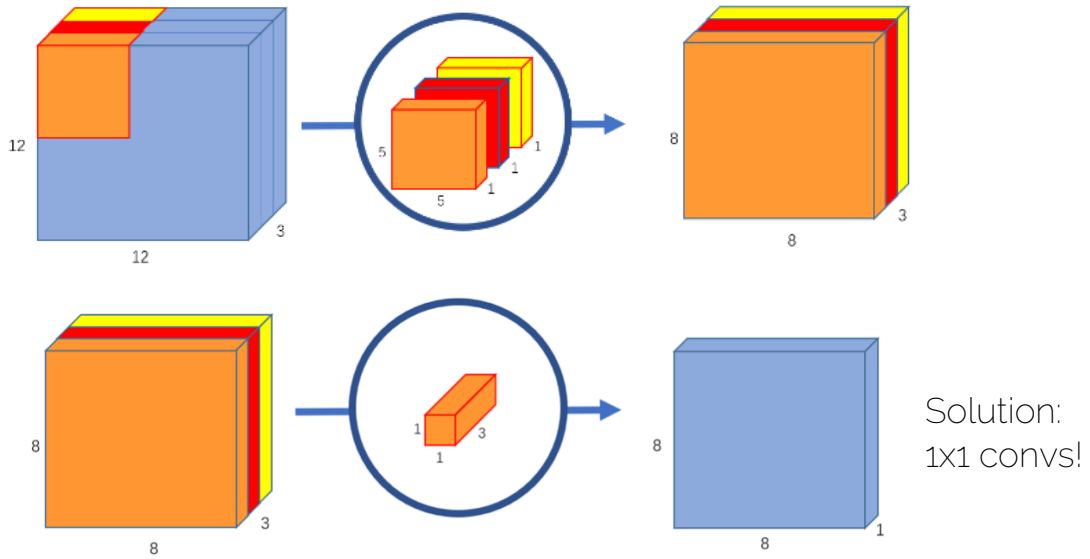


Figure X.8: Above: The depth size will always be the same.
Below: First DSC, then 1x1x3 convolution filter to change the output depth to 1

Why? DSC plus 1x1 Convolutions reduces the number of computations

X.8 Fully Convolutional Network

Convert fully connected layers to convolutional layers! (1x1 convolutions = FC layers) → Has a size of H/32 x W/32 instead of 1x1
Want output of the same size as input image → go back to original size with upsampling

Types of Upsampling

1. Interpolation: Few artifacts
   - Nearest neighbor interpolation: average of the pixels intensity around the pixel (with padding).
   - Bilinear interpolation: weighted average - give the surrounding pixels weights, and then sum. Could be of different kernel sizes. Results with a bigger, but blurry new image.
   - Bicubic interpolation: accomplished using either Lagrange polynomials, cubic splines, or cubic convolution algorithm

2. Transposed Convolution (Up-convolution): Unpooling (Blow image up: spread pixels and fill values with 0) + Convolution → efficient (If one does a cascade of unpooling + conv operations, we get to the encoder-decoder architecture)

U-Net: Even more refined: Autoencoders with skip connections (aka U-Net)

- Fully convolutional networks, that utilizes skip connections between the encoder and decoder units, for fine-grained details, boosting performance.
- Usually used for **segmentation** of the image to its objects, by classifying each single pixel among a veraity of classes.
- For every layer of the decoder part we concatate the output of the corresponding layer from the decoder (Which means that both encoder and decoder parts of the module have the same number of levels). Thus, we use the learnt features from the encoder as an input, a guidline of a sort, to the decoder on it's way to the target.
- **Left side:** Contraction Path (Encoder)
  - Captures context of the image
  - Follows typical architecture of a CNN: Repeated application of 2 unpadded 3x3 convolutions, Each followed by ReLU activation, 2x2 maxpooling operation with stride 2 for downsampling – At each downsampling step, # of channels is doubled
  - As before: Height and Width go down, Depth goes up
- **Right side:** Expansion Path (Decoder)
  - Upsampling to recover spatial locations for assigning class labels to each pixel: 2x2up-convolution that halves number of input channels, Skip Connections: outputs of up-convolutions are concatenated with feature maps from encoder, Followed by 2 ordinary 3x3 convs, final layer :1x1 conv to map 64 channels to # classes
  - As before: Heigh and Width go up, Depth goes down
- **Applications**:
  - Labeling pixels in the original images - useful for solving segmentation problems.
- **Note**: On the upscaling-stage we concatenate the corresponding layers from the down-scaling stage to the the new created layers by the **up-conv** layers. Then they are processed together.

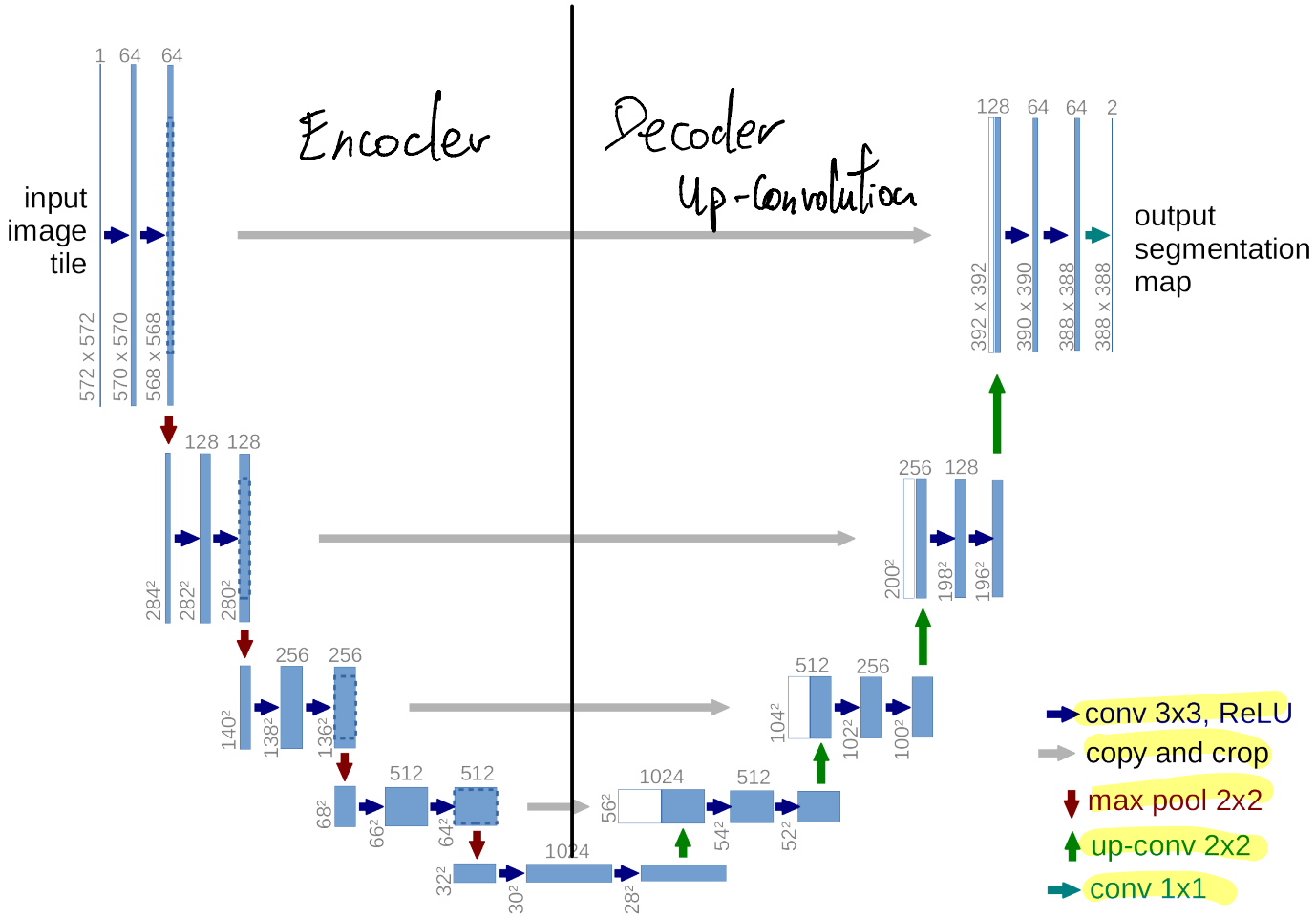


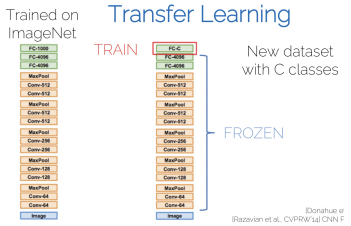
Figure X.9: U-Net architecture: Each blue box is a multichannel feature map. Number of channels denoted at the top of the box . Dimensions at the top of the box. White boxes are the copied feature maps.

XI Recurrent Neural Networks

XI.1 Transfer Learning

Training can be difficult with limited data and other resources (laborious task to manually annotate) → use pretrained models and add your own classifier on top of it
Two Distributions:

- Distribution P1: Large dataset
- Distribution P2: Small dataset



Slightly different task, but use what as been learned for another setting. Only change and the last decision class (fully connected layer). Reuse all the previous layer. Alternative: If the own dataset is big enough train more layers with a low learning rate.

When does it makes sense? It make sense when task T1 and T2 have the same input (e.g. an RGB image), when there's more data for T1 than for T2 or when the low level features for T1 can be useful to learn T2.

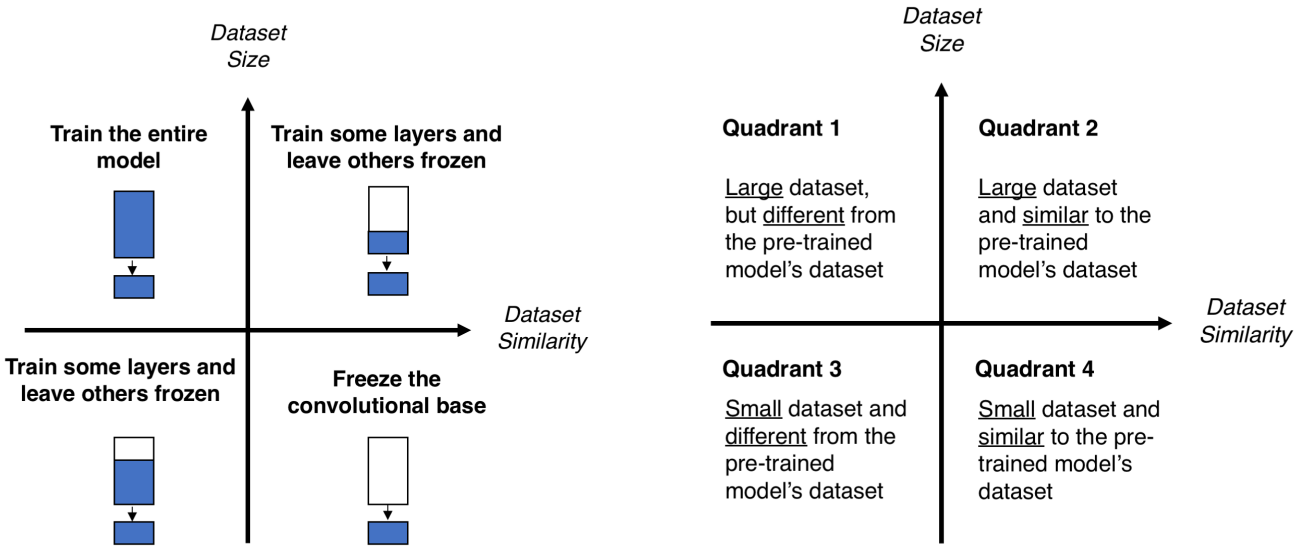


Figure XI.1: Size-similarity matrix and decision map for fine tuning pre-trained models

XI.2 Recurrent Neural Networks

RNN processes sequence data. The input and/or output can be sequences, whose components are related. They are flexible, can be one-one, one-many (Image captioning: Sentence for a single image), many-one (Language recognition), many-many (Machine translation (time shift), event classification (check if frames

show a dog)).
⇒ Learning process is not independent: remember things from processing training data, and remember things learnt from prior inputs, prior inputs influence decision

Basic Structure
Multi-layer RNN: Inputs, Hidden States, Outputs
Hidden states will have its own internal dynamics → more expressive model

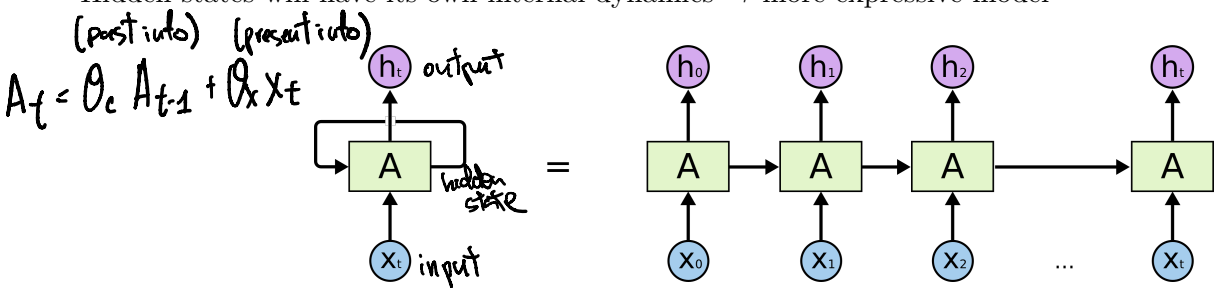


Figure XI.2: Basic structure of an RNN: From the current state, the output h is predicted. x_1, x_1 ... are processed with the same parameters. Output h_2 depends on $x_2, x_1$and x_0

We want to have notion of time or sequence. Take current input, weight it and compute the next state: $A_t = \theta_C A_{t-1} + \theta_x x_t$ where the former part is the previous hidden state and the latter is the input. The θs are parameters to be learned. The following formulas are used (ignoring activation functions):

$$A_t = \theta_C A_{t-1} + \theta_x x_t$$
$$h_t = \theta_h A_t$$

The parameters θ are the same for each time step. The connection between x^t and x^{t+1} is the same as between x^{t+1} and x^{t+2} (same weights). Backpropagation with chain rule all the way back t=0. Add the derivatives at different times for each weight. Not so straightforward as in other networks because weight appears more often.

Problem: Long-term dependencies (e.g. in sentences) → Important to keep enough information
Ignoring the input, same weight are are multiplied over and over again: $\boldsymbol{A_t = \theta_c{}^t A_0}$

- small weights → vanishing gradients (nearly 0)
- large weights → exploding gradients (very big)

θ is a matrix, we can write it by its eigendecompostion:

$$\theta = Q\Lambda Q^T \xrightarrow{\text{Orthogonal theta}} A_t = Q\Lambda^t Q^T A_0$$

- eigenvalues with magnitude < 1 → vanishing gradients
- eigenvalues with magnitude > 1 → exploding gradients (Gradient clipping: cut gradients larger than a threshold)

So in order to fix the exploding and vanishing gradients caused by θ, we enforce that the matrix has eigenvalues = 1. (Allows the cell to maintain its state).
But the vanishing gradients could also come from the activation function being used, tanh typically in this case.

→ Let us just make a matrix with eigenvalues = 1
(Allow the cell to maintain its "state")

XI.3 Long Short Term Memory

LSTMs try to solve those problems once in forall by setting the eigenvalues to 1 and ensuring that tanh does not cause vanishing gradients. Basically, they are an iteration of an simple RNN with tanh as non-linearity. It's a special kind of RNN capable of learning long-term dependencies.

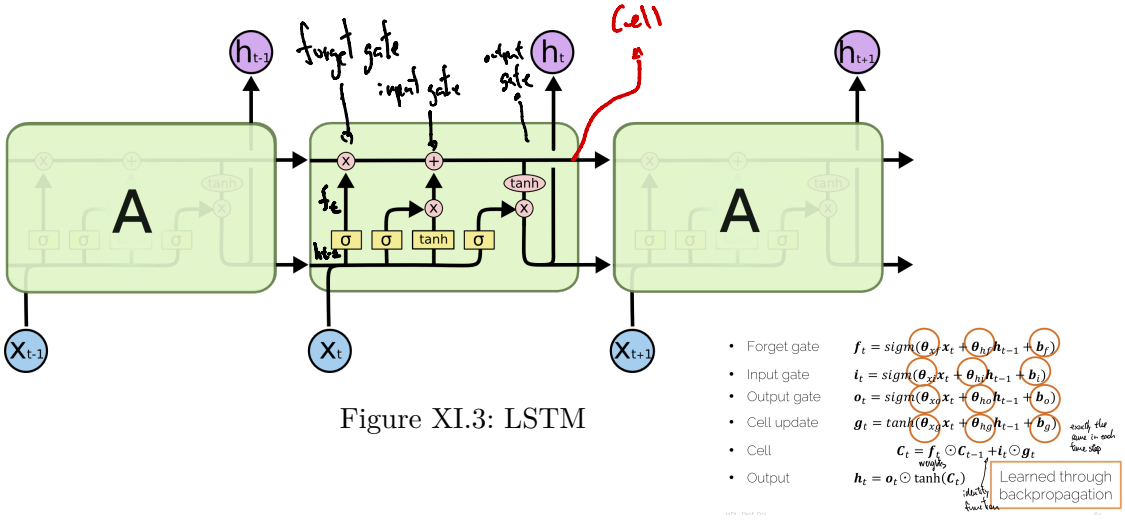
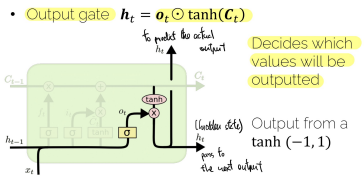


Figure XI.3: LSTM

Key ingredients:

- The cell transports the information through the unit (highway for the information) Add information through multiplication and sums. Just like the skip connections in the ResNet, this ensures that there is always a path we can take (i.e. longer sequence cannot cause problems).
- The gate removes or adds information to the cell state.
  - Forget gate: $f_t = \sigma(\theta_{xf}x_t + \theta_{hf}h_{t-1} + b_f)$. Decides when to erase the cell state. Sigmoid = output between 0 (forget) and 1 (keep)
  - Input gate: $i_t = \sigma(\theta_{xi}x_t + \theta_{hi}h_{t-1} + b_i)$. Decides which values will be updated. New cell state from $\tanh(-1, 1)$.
  - Interaction between current and previous time step: element-wise operations: $C_t = f_t \odot C_{t-1} + i_t \odot g_t$
  - Output gate: $o_t = \tanh(\theta_{xo}x_t + \theta_{ho}h_{t-1} + b_o)$ Decides which values will be outputted. Output from a $\tanh(-1, 1)$
- Actual output: $h_t = o_t \odot \tanh(C_t)$
- Cell update: $g_t = \tanh(\theta_{xg}x_t + \theta_{hg}h_{t-1} + b_g)$

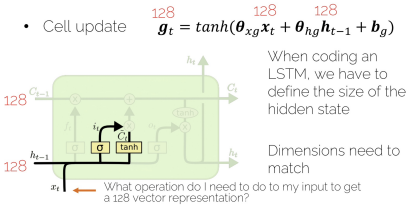


LSTM solved the vanishing gradient problem because...
... f_t are outputs of an sigmoid and therefore 1 for important information.
... because activation functions act through a summation, therefore vanishing gradient are not propagated through the whole cell state.

Dimensions: When coding an LSTM, we have to define the size of the hidden state. Dimensions need to match, but are not limited to vectors.

General LSTM Units: Input, states, and gates are not limited to 1st-order tensors. Gate functions can consist of FC and CNN layers
E.g.: ConvLSTM for video sequences: Input, hidden, and cell states are higher order tensors (i.e. images). Gates have CNN instead of FC layers

RNNs in Computer Vision: Caption generation and Instance segmentation (separate different instances of a class)



What is missing from self-attention?

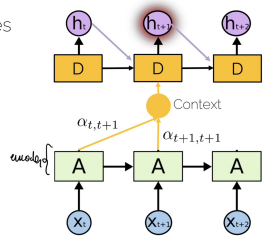
- Convolution: a different linear transformation for each relative position. Allows you to distinguish what information came from where.
- Self-attention: a weighted average.

XII Advanced Deep Learning Topics

XII.1 Attention

Attention: Architecture

- A decoder processes the information
- Decoders take as input:
  - Previous decoder hidden state
  - Previous output
  - Attention



Same goal as LSTM
Attention is the intuition between the hidden state and the output. We want to give the unit a context, and what it does is that it gives the current time stamps and all the other one a relationship. We have α which are variables (attention variables) which are weights that represent how important the time stamp t+1 is to predict the output at t+1. All these variables are processed together to what it's called a context.

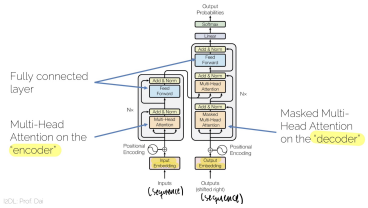
A decoder processes the information
Decoders take as input: Previous decoder hidden state, Previous output, Attention

Transformers: it gets rid of the recurrent architecture (memory problems of RNNs disappear). There are direct connections. It's not RNN or CNN. They just use attention.

Difference: in RNN the words are processed sequentially; transformers have connections all over the place and the alphas say how strongly the words are connected to each other.

Transformers have a lot of parameters too and a very related to Graph Neural Networks.

Transformers



XII.2 Graph Neural Networks

Node: a concept (e.g. word, image)
Edge: a connection between concepts

Generalizations of neural networks that can operate on graph-structured domains
Key challenges: The number of nodes and edges changes as we move from one graph to the other. We need invariance to node permutations.

There can be node and edge feature vectors, which go to a series of hidden layers where the propagation of information takes place. Propagate information over several iterations. The operation here is not a convolution. At the end the feature vectors have been changed. It represents the nodes but also the neighbors ⇒ graph with updated context-aware node and (possibly) edge feature vectors

Message passing networks

We can divide the propagation process in two steps: 'node to edge' and 'edge to node' updates (alternating). See XII.1.

Multi-object tracking with graphs

Same object in different frames of a video

Step 1: Object detection. Each node can represent a detection and the edges can represent the trajectory.
Step 2: Use the graph to perform learning. Not solve the problem directly, but perform neural message passing steps (sharing of information between node and edges). Classify edges: feature vector as input and decides if active or not. Extract trajectories.

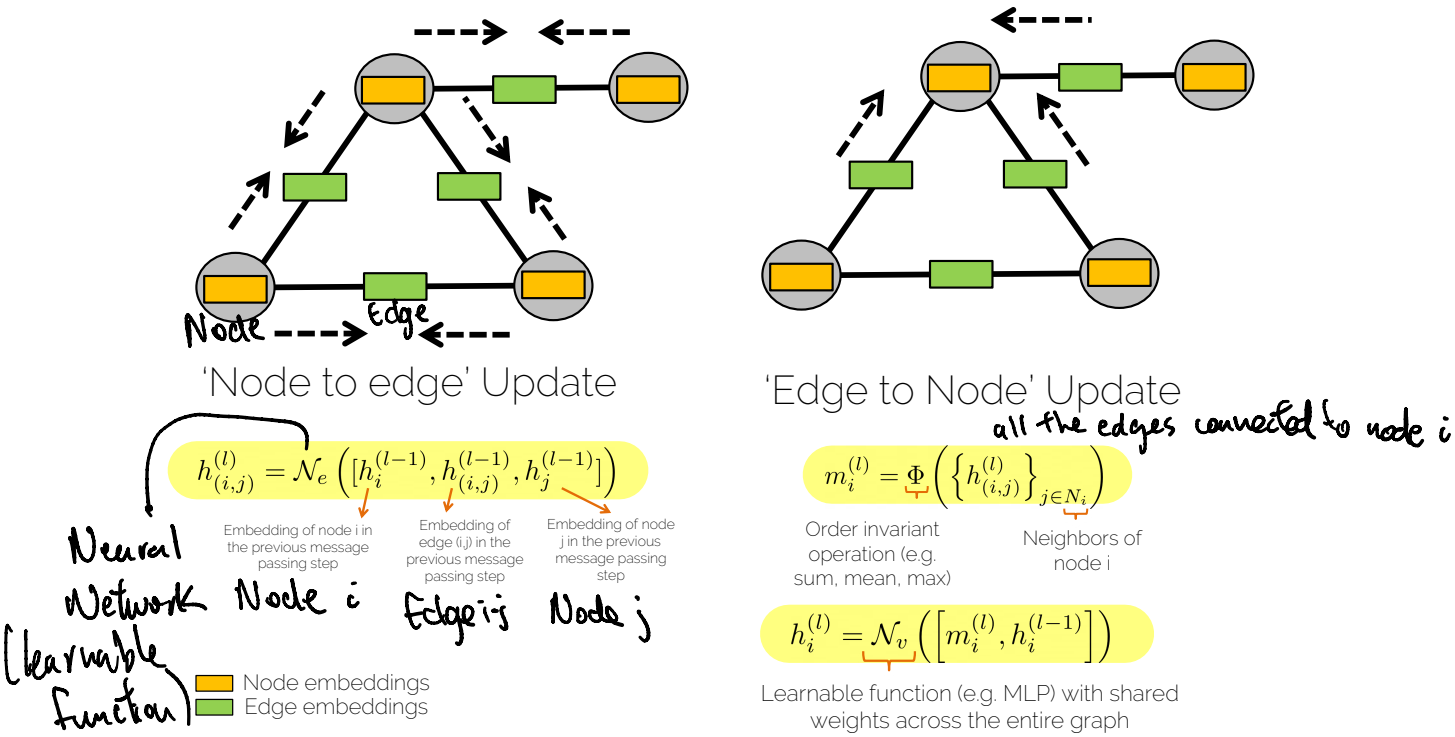


Figure XII.1: Node to edge: At every message passing step l, $\mathcal{N}_e$ is learnable (small) neural net with shared weight across the entire graph. Every edge is updated by applying the neural net to the old value, and the value of the two nodes of the edge.
Edge to node: This update is a bit trickier, as we don't know the number of edges a node has (compared to the previous case where an edge always has two nodes). To solve this, we need an order invariant function Φ that aggregates all edge values of a node to a single value. Then we again have a shared neural net that process this single combined edge information and the current node value to produce the new node value. The aggregation provides each node embedding with contextual information about its neighbors.
Repeating those steps allows the the nodes to communicate over the edges and the neural networks decide how important an information is.

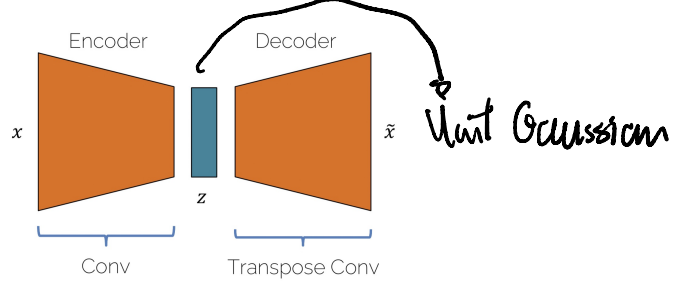
XII.3 Generative Models

Outputs of the NN with same width and height as input (e.g. Semantic Segmentation (FCN) with upsampling) but there are better ways as the SegNet (Convolutional Encoder-Decoder)
Given training data, how to generate new samples from the same distribution: Various methods

XII.4 Variational Autoencoder

Autoencoder: Encoding the input into a representation (bottleneck) and reconstruct it with the decoder ⇒ Reconstruct input, Unsupervised learning
Variational Autoencoder:

- Goal: Sample from the latent distribution to generate the new outputs
- Training: The loss makes sure the latent space is close to a unit Gaussian and the reconstructed output is close to the input
- Test: Sample from the latent space



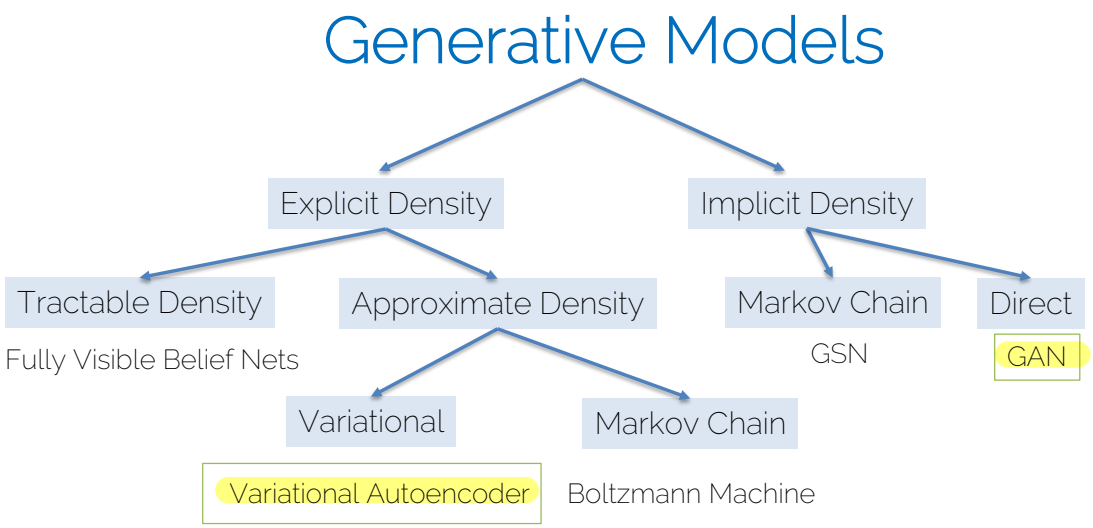


Figure XII.2: Explicit: Model explicitly the density. Plug in the model density function to likelihood. Then maximize the likelihood
Implicit: No explicit probability density function (pdf) needed. Instead, a sampling mechanism to draw samples from the pdf without knowing the pdf

- $\Rightarrow$ Probability distribution in latent space (e.g., Gaussian), Interpretable latent space (head pose, smile), Sample from model to generate output

XII.5 Generative Adversarial Networks (GAN)

Huge increase in the usage
Reconstruction with Loss (often L2) gives blurry output because distributes error equally, mean is the optimum. $\rightarrow$ **Learn the loss function**

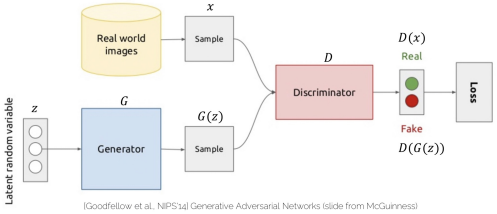
We have generators G and discriminators D(these tell you if the images are real or fake). G and D are neural networks so the improve each other with backpropagation. Train with real world images. Generator is trained to better generate images and discriminator is trained to detect real and fake images: G $\Leftrightarrow$ D

Real data: Sample from the data, pass it through D (tries to be near 1)
Fake data: Input noise passes through G which generates a image. D tries to identify the image $\rightarrow$ D tries to make D(G(z)) near 0, G tries to make D(G(z)) near 1.

Discriminator Loss: $J^{(D)} = -\frac{1}{2}E_{x\ p_{data}} \log D(x) - \frac{1}{2}E_z \log(1 - D(G(z)))$ (binary cross entropy)
Generator Loss: $J^{(G)} = -J^{(D)}$

Minimax Game: G minimizes probability that D is correct. Equilibrium is saddle point of discriminator loss (D provides supervision (i.e., gradients) for G)

Generative Adversarial Networks (GANs)

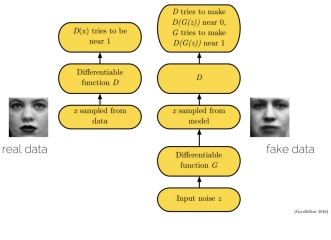


[Goodfellow et al., NIPS'14] Generative Adversarial Networks (slide from McGuinness)

GAN Applications

- Generate high quality (definition) images
- Generate new faces from scratch
- Conversions: a horse to a zebra (zebrify), day to night
- GAN based image editing: From drawing to real photo, change faces in videos

Generative Adversarial Networks (GANs)



XII.6 Reinforcement Learning

Good behavior → Reward
Bad behavior → Punishment

Learning by interacting with the environment. Basically we get the labels from the environment.

- Environment sends an Observation o_t to the Agent
- The agent makes decisions and performs an action a_t based on the history h of observations, actions and rewards up to time-step t
- Based on the action, Agent gets a reward r_t (positive or negative)

Characteristics: Sequential, non i.i.d. data (time matters), Actions have an effect on the environment → Change future input. No supervisor, target is approximated by the reward signal.

Problem: Your actions are changing the future, so you need to take into account and keep in a state all history. Therefore there needs to be a mapping between the history and the state s. But this is really hard to maintain. History grows linearly over time. So use a Markov assumption. A state t is Markov if it only depends on the past state. Reward and next state are functions of current observation o_t and action at only

Components of an RL agent: Policy (Behavior of the agent → Mapping from state to action) and value, q-function (How good is a state or (state, action) pair → Expected future reward)

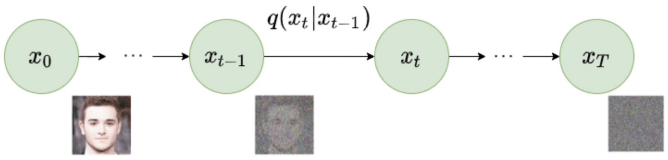
Diffusion

- Class of generative models

Diffusion Process

- Gradually add noise to input image x_0 in a series of T time steps
- Neural network trained to recover original data

objective: denoise X_T and recover the original data



2 steps:

1) Forward Diffusion: (add noise)

Markov chain of T steps (each step depends only on previous)
Adds noise to X_0 sampled from real distribution $q(x)$

$q(x_t|x_{t-1}) = N(x_t; \mu_t = \sqrt{1-\beta_t}\, x_{t-1}, \Sigma_t = \beta_t I)$
(mean) (variance)
(not efficient)

Reparameterization

- Define $\alpha_t = 1 - \beta_t$, $\bar{\alpha}_t = \prod_{s=0}^{t} \alpha_s$, $\epsilon_0, \ldots, \epsilon_{t-1} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$

$$x_t = \sqrt{1-\beta_t} x_{t-1} + \sqrt{\beta_t}\epsilon_{t-1}$$
$$= \sqrt{\alpha_t} x_{t-2} + \sqrt{1-\alpha_t}\epsilon_{t-2}$$
$$= \cdots$$
$$= \sqrt{\bar{\alpha}_t} x_0 + \sqrt{1-\bar{\alpha}_t}\epsilon_0$$

$$x_t \sim q(x_t|x_0) = \mathcal{N}(x_t; \sqrt{\bar{\alpha}_t} x_0, (1-\bar{\alpha}_t)\mathbf{I})$$

Approximate Reverse Process

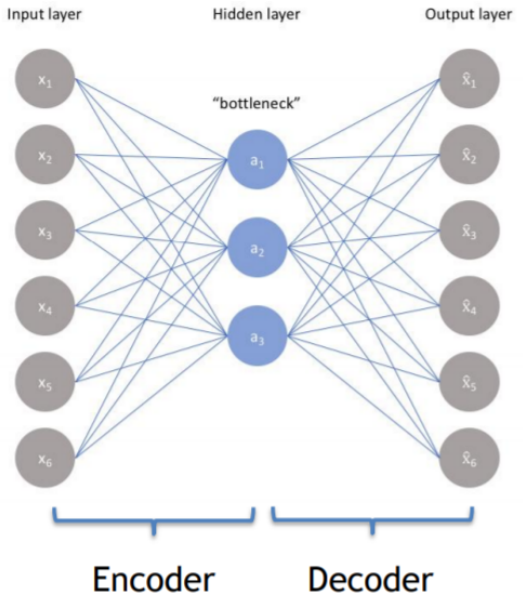
- Approximate $q(x_{t-1}|x_t)$ with parameterized model p_θ (e.g., deep network)

$$p_\theta(x_{t-1}|x_t) = \mathcal{N}(x_{t-1}; \mu_\theta(x_t, t), \Sigma_\theta(x_t, t))$$

$$p_\theta(x_{0:T}) = p_\theta(x_T) \prod_{t=1}^{T} p_\theta(x_{t-1}|x_t)$$

XII.7 Autoencoder

One of the most interesting network architectures. An autoencoder is such a powerful tool, that later on was perfected, and it is being used intensively in the industry and research, for the tasks of segmentation, 3D reconstruction, generative models, style transfers, etc. Basically, endless applications.



1. The basic architecture is still based on fully connected layers (Of course with activations layers).
2. It consists of two main parts:
   - Encoder: Reducing the dimensionality towards the "latent" space forces the network to focus on collecting the most meaningful features from the input data, so the loss of information would be as small as possible. As stated in the tutorial session, in similar notion to PCA. Therefore, we could say that the Encoder is used as a **features extractor**, much alike the upcoming convolutional layers (But yet, not as strong, and still much more expansive).
   - Decoder: Receives the dimensionality-reduced latent space, and aims to **reconstruct** the input of the Encoder. The output has the same shape as the input. A crucial note, for more advanced usage, of semantic segmentation.
3. For the loss function, we could use the
   $$L_1$$
   loss or the MSE, between each pixel in the input and its corresponding pixel in the output.
4. Latent space:
   - If the size of is too small, not much information could eventually pass through to the decoder, and the reconstruction would be very hard. The result would be very blurry.
   - If too big, the network could basically learn to copy the image, without learning any meaningful features.
5. Without non-linearities, it is very similar to PCA.

But why do we even want to use that? Auto encoders, as used in exercise 08, is an excellent solution to a state where our dataset is very big, but only just a small part of it is actually labeled, like medical a CT dataset, for example. So, we will have 2 steps:

1. Autoencoder

   $\rightarrow$

   reconstruct the input. Let the Encoder learn the relevant features about the **unlabeled** data. This part can be referred to as **unsupervised learning**.

2. After the training has converged, remove the decoder and discard it. Then, plug in instead, just after the latent-space, a very simple fully-connected classifier, and train on the **labeled** data, given the fact that the remaining Encoder is already trained as a good features extractor. This part can be referred to as **supervised learning**.