IntroML - Lecture Notes Week 7

Ruben Schenk, ruben.schenk@inf.ethz.ch

July 15, 2022

0.1 Stochastic Gradient Descent for ANNs

The **stochastic gradient descent** approach for ANNs looks as follows:

Algorithm: We want to solve

$$W^* = \arg\min_W \sum_{i=1}^{n} l(W;\, x_i,\, y_i)$$

1. We initialize the weight W
2. For $t = 1,\, 2, \ldots$:
   - Pick data point $(x,\, y) \in D$ uniformly at random
   - Take step in *negative gradient* direction

$$W \leftarrow W - \eta \Delta_W l(W;\, x,\, y)$$

Typically we use *minibatches* to reduce variance/exploit parallelization.

But how do we optimize over weights? We want to do **empirical risk minimization,** i.e. jointly optimize over all weights for all layers to minimize loss over the training data. This is in general a *non-convex* optimization problem! Nevertheless, we can still try to find a local optimum.

Remark: There are **weight-space symmetries.** Multiple distinct weights can compute the same predictions. In other words, multiple local minima can be equivalent in terms of input-output mapping.

Computing the gradient is done through backpropagation in matrix form:

Algorithm:

1. For the output layer
   - Compute "error": $\delta^{(L)} = \Delta_f l$
   - Gradient: $\Delta_{W^{(L)}} l = \delta^{(L)} v^{(L-1)T}$
2. For each hidden layer $l = L - 1 : -1 : 1$
   - Compute "error": $\delta^{(l)} = \phi'(z^{(l)}) \odot (W^{(l+1)T} \delta^{(l+1)})$
   - Gradient: $\Delta_{W^{(l)}} l = \delta^{(l)} v^{(l-1)T}$

0.2 Weight Initialization in Neural Networks

Onr problem we might encounter are **vanishing** and **exploding gradients.** Remember the gradient computation in backpropagation was defined as:

$$\Delta_{W^{(i)}} l = \delta^{(i)} v^{(i-1)T} \text{ where } \delta^{(i)} = \phi'(z^{(i)}) \odot (W^{(i+1)T} \delta^{(i+1)}) \text{ and } v^{(i)} = \phi(W^{(i)} v^{(i-1)})$$

The potential issue is exploding ($||\Delta_{W^{(i)}} l|| \to \infty$) or vanishing ($||\Delta_{W^{(i)}} l|| \to 0$) gradients can cause optimization to fail. Why can this happen? Potential reasons are $||\delta^{(i)}||$ going to 0 or ∞ (or analog. for

$||v^{(i)}||$).

- Using certain activation functions (e.g. ReLU) can help avoid $||\delta^{(i)}|| \to 0$.
- The error signal $\delta^{(i)}$ is scaled by $v^{(i)}$. This can help to reduce the vanishing/exploding gradient problem by keeping the magnitude of $v^{(i)}$ constant across the layers.

The general goal when initializing weights is to keep the variance of weights approximately constant across layers to avoid vanishing and exploding gradients and network activations. Usually, **random initialization** works well, e.g.

- Glorot (tanh): $w_{i,\,j} \sim \mathcal{N}(\frac{1}{n_{in}})$ or $w_{i,\,j} \sim \mathcal{N}(\frac{2}{n_{in}+n_{out}})$
- He (ReLU): $w_{i,\,j} \sim \mathcal{N}(\frac{2}{n_{in}})$

We need to ensure that at initialization, unit activations are approximately standardized.

0.3 Learning Rates

To implement the SGD update rule ($W \leftarrow W - \eta_t \Delta_W l(W;\, x,\, y)$), we need to choose the learning rate η_t. In practice, we often use a decaying learning rate schedule, e.g. piecewise constant.

We may want to monitor the ratio of weight change (gradient) to weight magnitude:

- If it's too small, we increase the learning rate
- If it's too large, we decrease the learning rate

Learning with momentum is a common extension to training with (stochastic) gradient descent. It can help to escape the local minima. The idea is as follows: We move not only into the direction of the gradient, but also in direction of the last weigh update. The updates then are:

- $d \leftarrow m \cdot d + \eta_t \Delta_W l(W;\, x,\, y)$
- $W \leftarrow W - d$

In some cases, learning with momentum cad *prevent oscillation.*

0.4 Reclarization in Neural Networks

Neural networks have many parameters, so there's a potential danger of overfitting. Countermeasures are:

- *Regularization (weight decay):* Add penatly term to keep the weights small

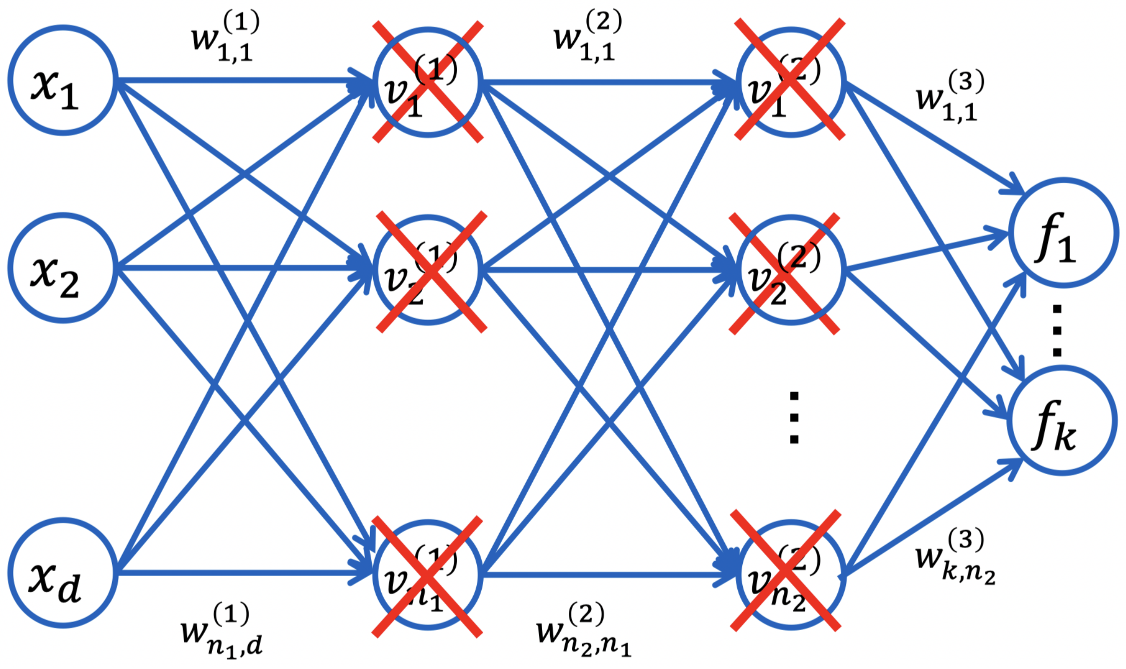
$$W^* = \arg\min_W \sum_{i=1}^{n} l(W;\, x_i,\, y_i) + \lambda ||W||_2^2$$

- *Early stopping:* Don't run SDG until convergence
- *"Dropout"*

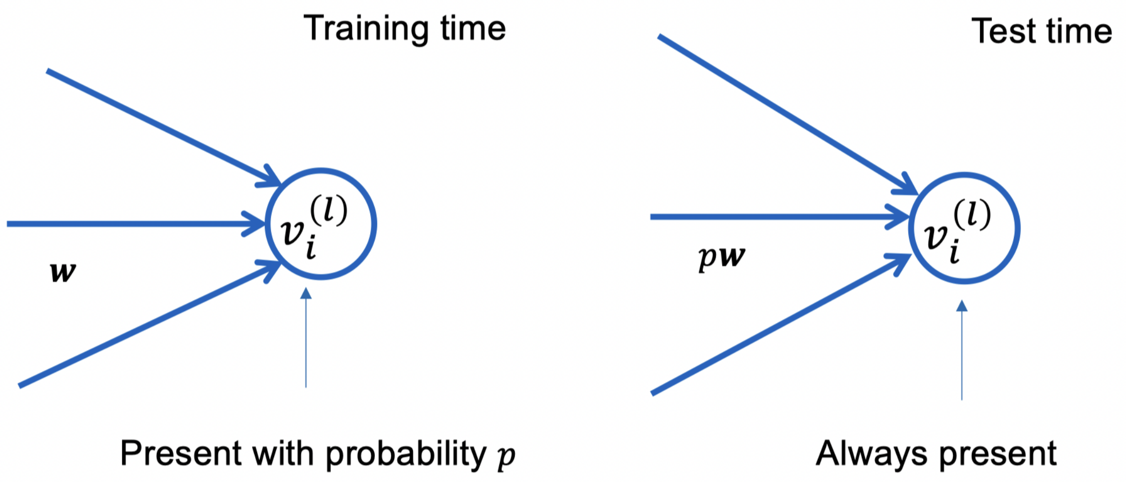
The idea behind **early stopping** is as follows: In general, we might not want to run the training until the weights convergence since this potentially leads to overfitting. One posssibility is:

1. Monitor the prediction perfomance on a validation set
2. Stop the training once the valdiation error stops to decrease

The idea behind the **dropout** technique is to randomly ignore hidden units during each iteration of SGD with probability p:



After the training, we use all units, but multiply the weights by p to compensate.



0.5 Batch Normalization

In deep learning, inputs are shifted and scaled through each layer. **Batch normalization** is a widely used technique that normalizes unit activations in each layer according to mini-batch statistics:

- Reduces internal covariate shift
- Enables larger learning rates
- Has regulatizing effect

This can be integrates as a layer into the neural network, with two learnable parameters β, γ (per hidden unit) that can "learn to undo" the normalization (and also serve as bias nodes).

This encourages standardization of unit activations not only for inputs, but also hidden units. Not only at initialization, but also during training.

Algorithm: Batch normalization layer

1. Input: Incoming activation values for all examples v_i for all $i \in S$ in the minibatch S, $\epsilon > 0$
2. Learnable parameters: β, γ
3. Batch normalization computes $\bar{v} = BN(v; \gamma, \beta)$ by:
   (a) Compute the mini-batch mean: $\mu_S = \frac{1}{|S|} \sum_{i \in S} v_i$
   (b) Compute the mini-batch variance: $\sigma_S^2 = \frac{1}{|S|} \sum_{i \in S} (v_i - \mu_S)^2$
   (c) Normalize each point: $\hat{v}_i = \frac{v_i - \mu_s}{\sqrt{\sigma_S^2 + \epsilon}}$
   (d) Scale and shift: $\bar{v}_i = \gamma \hat{v}_i + \beta$
4. Output: $\bar{v}_i$ for all $i \in S$ in the minibatch S

1 Convolutional Neural Networks

1.1 Introduction

So far, we have only discussed fully connected neural networks. There exist, however, specialized archtiectures designed for certain classes of applciations, such as:

- Convolutional neural networks
- Residual networks and skip connections
- Recurrent neural networks
- Memory (LSTMs, GRUs,...)
- etc.

Let us pose some invariances of predictions. Predictions should be unchanged under some transformations of the data, e.g.:¨

- Classifications of handwritten digits should be undchanged under translation, rotation, scale, etc.
- Speech recognition

How do we encourage a model to learn specific invariances?

- Augmentation of the training set
- Special regularization terms
- Invariance built into pre-processing
- Implement invariance into structure of ANN (e.g. using convolutional neural networks)

1.2 Convolutions

Convolutional neural networks are ANNs for *specialized applications* (e.g. image recognition). The hidden layers closest to the input layer *share parameters:* Each hidden unit only depends on all closeby inputs (e.g. pixels), and weights constrained to be identical across all units on the layer. This reduces the number of parameters, and ecnourages robustness against small amounts of translation.

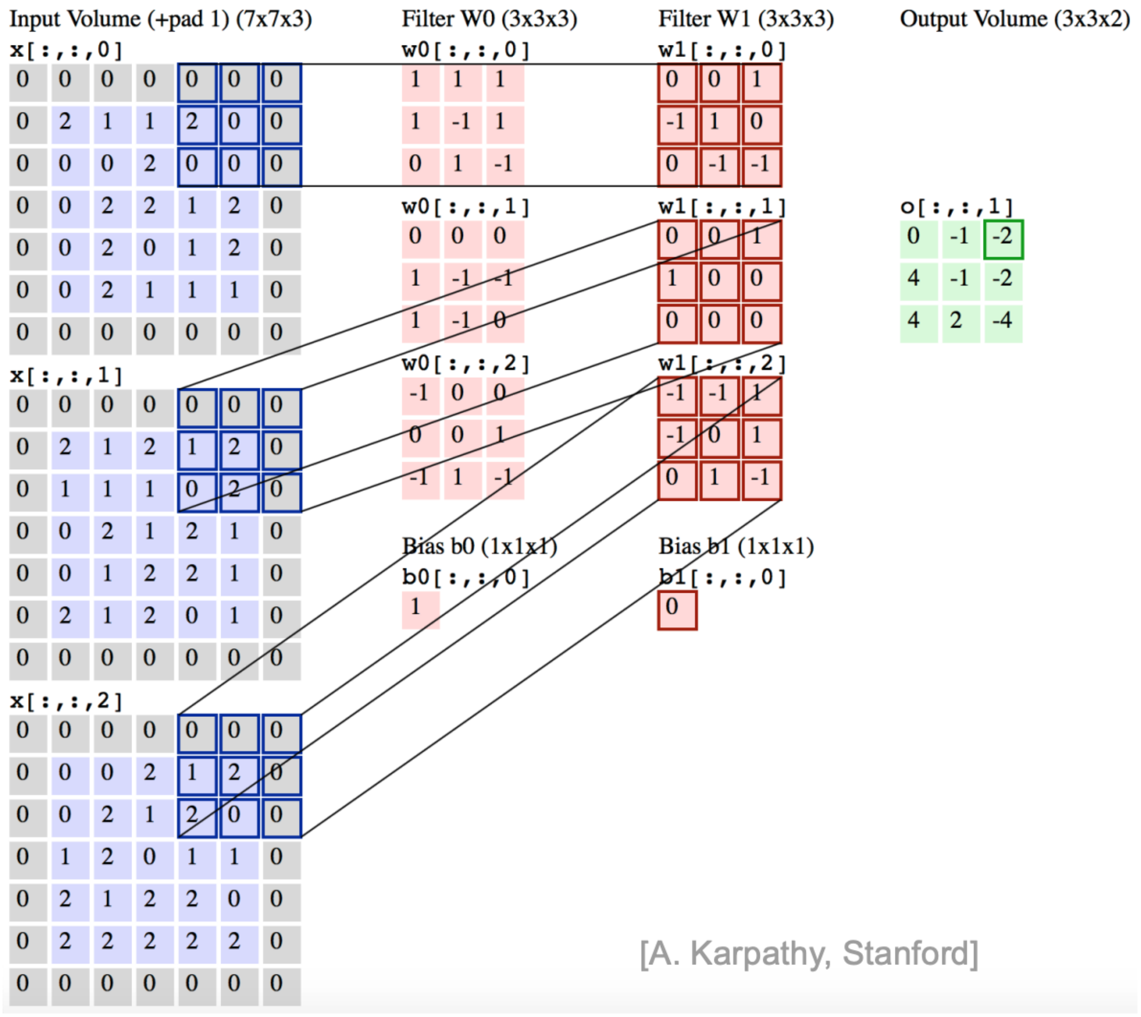
Remark: The weights can still be optimized via backpropagation.

Lets first consider convolutions in 1D. We can define this as follows: Given a row vector $w \in \mathbb{R}^k$ and a column $x \in \mathbb{R}^d$, then their **convolution** $z = w * x$ is defined by the vector $z \in \mathbb{R}^{d+k-1}$ via

$$z_i = \sum_{j=\max(1,\, i-d+1)}^{\min(i,\, k)} w_j x_{i-j+1}.$$

This naturally generalizes to matrices W to yield $z = W * x$.

Lets also consider convolutions with 2D data (such as images). We summarize the approach in the following figure:



If we compare those to fully connected layers, we see the following similarities:

- Fully connected layers: $v^{l+1} = \phi(Wv^l)$
- Convolutional layers: $v^{l+1} = \phi(W * v^l)$

We might also be interested in computing the *output dimensions:*

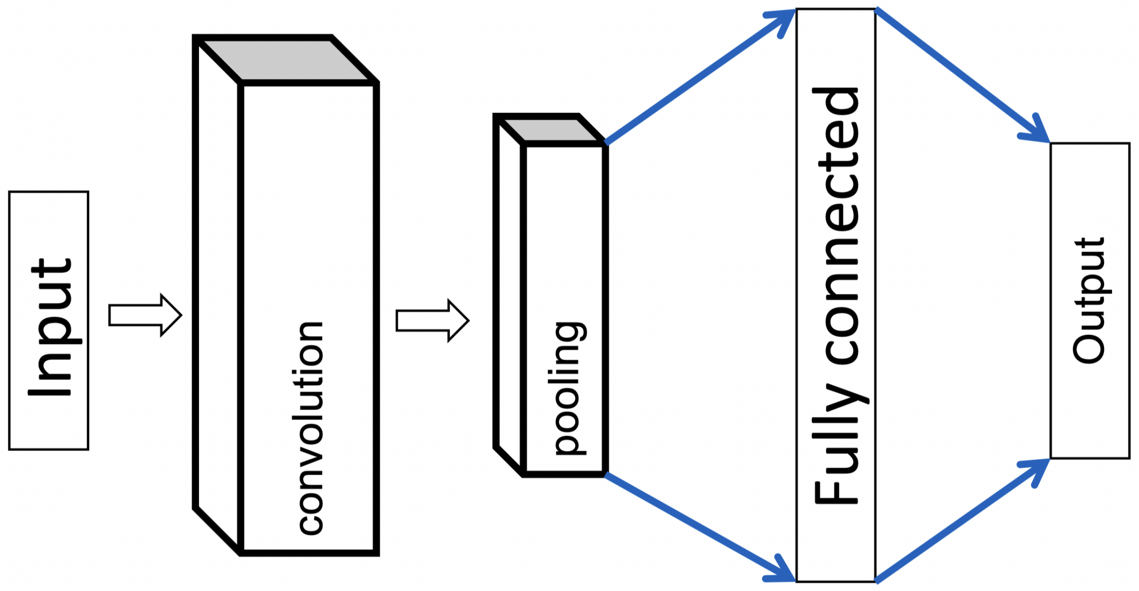
- Apply m different $f \times f$ filters
- To a $n \times n$ image
- With PAdding p
- And stride s

Then:

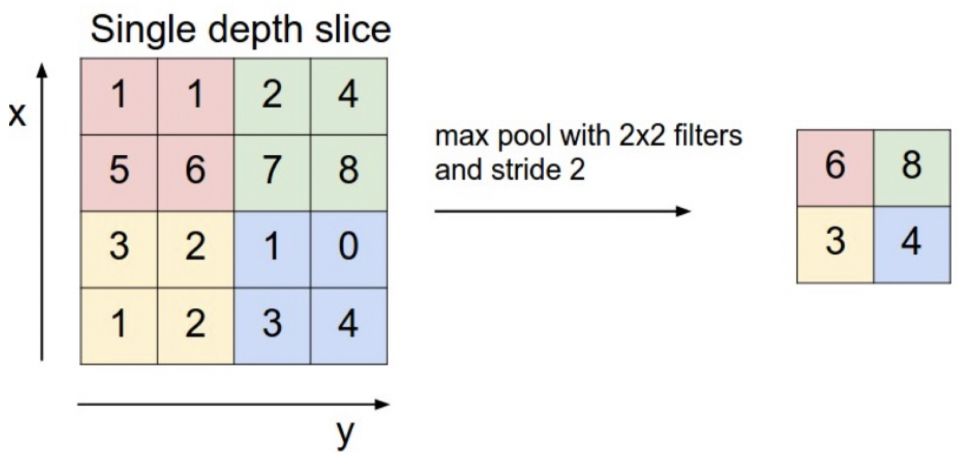
$$l = \frac{n + 2p - f}{s} + 1$$

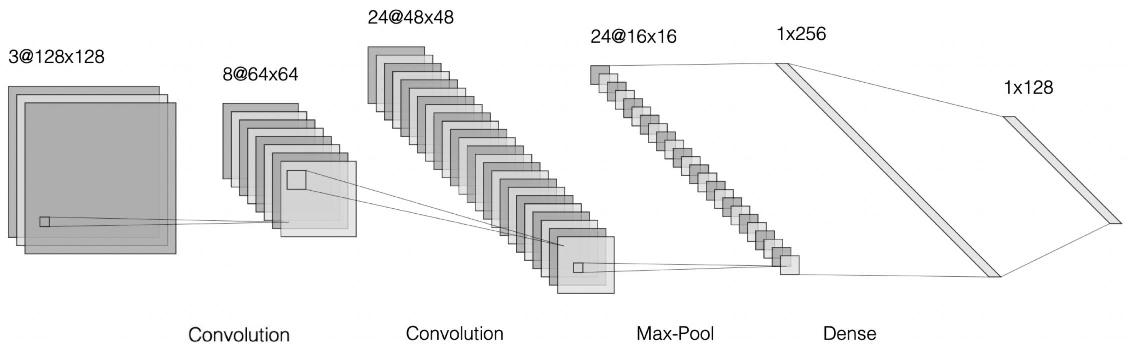
1.3 CNN Architecture

Finally, this leaves us with the following architecture for CNNs:

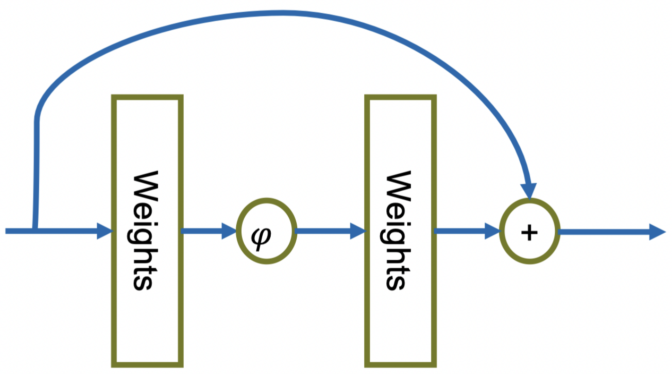


In some applications, it can make sense to **aggregate (pool)** several units to decrease the width of the network (and hence the number of parameters). Usually, one considers either the average or the maximum value. Example:





1.4 Residual Networks



The idea behind **residual connections** is as follows:

1. Introduce skip connections that can allow to effectively train deeper networks
2. "Identity shortcut connection" helps to avoid vanishing gradients
3. This allows to effectively train networks with 1000+ layers
4. It's also possible to skip more than one layer (DenseNets)