CSE517A Machine Learning **Spring 2025**

Lecture 10: Strategies for Deep Learning

Instructor: Marion Neumann

Reading: LFD eCH 7.4 (Regularization), 7.5 (Beefing Up GD), 7.6 (DL), ESL 11.5 (Training NNs)

1 Introduction

Deep neural networks (NNs) have a huge number of parameters, e.g. 60 000 000 for ALEXNET, almost 15 000 000 for VGG16 (cf. Figure 2), and 11.2 billion for STANFORD ANN. Hence, it should be fairly obvious that the key to successfully train deep NNs is to have huge amounts of training data. However, having enough training data is not the only issue we face. The two other main challenges for training NNs are:

- *overfitting* (which is again a pretty obvious problem, since we typically have way more parameters than training data)
- *difficult optimization* (non-convex, non-continuous objective function, batch/stochastic gradient updates, etc.)

Demo: go to http://playground.tensorflow.org, pick the spiral dataset and try to learn a *nice* decision boundary to experience the difficulty of training a NN.

In the following, we will discuss some methods, namely *regularization*, *optimization strategies*, and *parameter initialization* to tackle these challenges.

2 Regularization

2.1 Weight Decay

Use l_1 or l_2 regularization on weights (not biases!) in every layer.

Define $a_i = \mathbf{w}^\top \mathbf{x} + b_i^{(l)} = W_{i:}^{(l)} \mathbf{x} + b_i^{(l)}$ where $W_{i:}^{(l)}$ is the i-th row in $W^{(l)}$ and use, for example, l_2 regularization:

$$\frac{\lambda}{2}\|\mathbf{w}\|_2^2 = \frac{\lambda}{2}\mathbf{w}^\top\mathbf{w} \quad \Rightarrow \quad \frac{\partial \frac{\lambda}{2}\mathbf{w}^\top\mathbf{w}}{\partial \mathbf{w}} = \lambda\mathbf{w}$$

New gradient update:

$$W_{i:}^{(l)} = \mathbf{w} = \mathbf{w} - \alpha(\Delta\mathbf{w} + \lambda\mathbf{w}) = (1 - \alpha\lambda)\mathbf{w} - \alpha\Delta\mathbf{w} \tag{1}$$

The coefficient $(1 - \alpha\lambda)$ will shrink the weight vector (*weight decay*) before usual gradient update.

2.2 Constrained Optimization

Use explicit constraints and reproject weights if constraints are violated.

Let $\Omega(\mathbf{w})$ be the norm penalty (e.g. $\Omega(\mathbf{w}) = \frac{1}{2}\|\mathbf{w}\|_2^2$). We have the constraint $\Omega(\mathbf{w}) \leq k$ where k is some constant. Then we have the new objective: $\ell(\mathbf{w}, \mathbf{x}, y) + \lambda(\Omega(\mathbf{w}) - k)$. The new objective does gradient update and project gradient back to closest point that satisfies $\Omega(\mathbf{w}) \leq k$.

2.3 Early Stopping

Stop GD before convergence.

E.g. use a *validation set* and stop when validation error increases (cross-validation does not work), cf. Fig. 1. This is very popular if we have enough data.

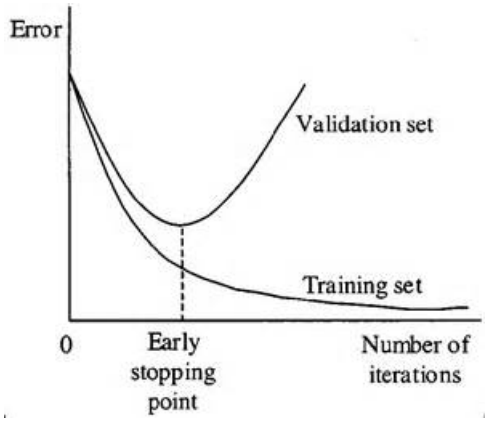


Figure 1: Stop iteration when validation error increases

2.4 Transfer Learning

Transfer learning is a general concept in machine learning, where you train a model on one dataset and then adapt it to a different dataset in the same domain. If you want to use NNs for problems where you do **not have enough training data**, then your best bet is to start with a pre-trained NN or train a NN on an available huge training dataset from the same domain. The pre-trained network is then optimized (**re-trained**) with the training data of your application. Instead of starting with random parameter initialization, the parameters of the pre-trained network are used for training. Note that obviously the architecture of your network will need to be the same as the architecture of the pre-trained network. This is a very common strategy in computer vision or natural language processing used for tasks like image or text classification, where pre-trained networks on huge datasets like IMAGENET or WIKIPEDIA exist. Figure 2 shows the architecture of VGG16, which is pre-trained on 15 million labeled images from IMAGENET.

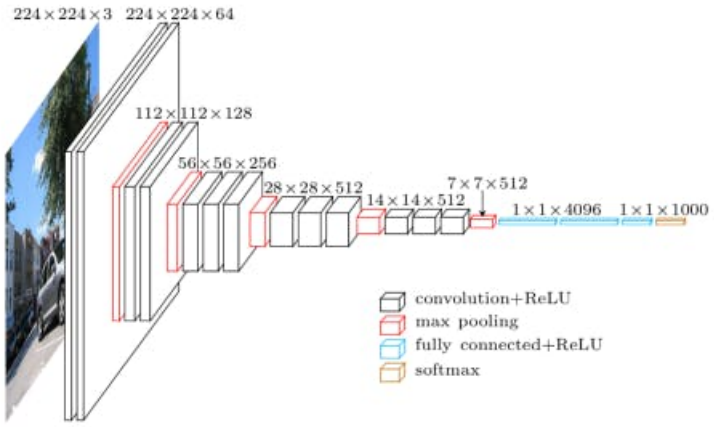


Figure 2: VGG16 was trained on 15 million annotated images from IMAGENT and has almost 15 million parameters. It took over **two weeks** to train the network using **four** GPUs.

Quiz question: Does the prediction task have to be the same? Yes or no?

2.5 Ensemble Learning

Train the NN with different initial weights and then use the average results from the ensemble models → each model is independent. This is the classical **bagging** approach, which is seldomly used to train NNs in practice due to its high computational cost.

An approximation to this actual bagging approach is **dropout**, where we randomly remove each hidden unit with a probability α (e.g. $\frac{1}{2}$) for each training point or mini-batch during training. Note that now the ensemble models are not independent as they share parameters. In fact, we only get one model that represents the entire ensemble.

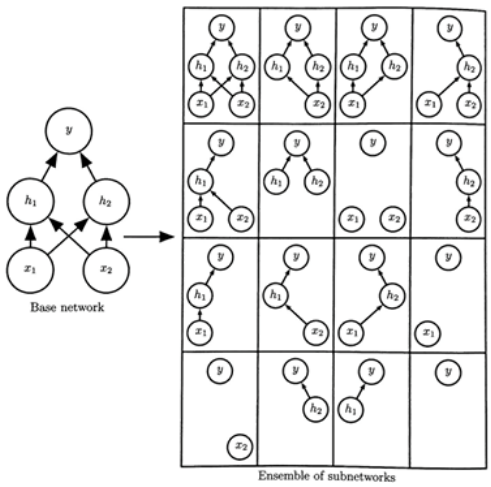


Figure 3: Illustration of dropout (illustration from [Goodfellow, Bengio, Courville, Deep Learning, 2016, MIT Press]).

2.6 Batch Normalization

The idea of batch normalization is to re-scale the output of every linear transformation during NN training to **prevent** *covariance shift*. Recall that input *data normalization* is obtained by subtracting the sample mean and dividing by the sample standard deviation (cf. Lecture 13).
So, to normalize the $\mathbf{a}$'s we need to estimate their mean and variance. This can be achieved in a stochastic gradient descent (SGD) by using the $\mathbf{a}_i$ of each mini-batch. Now, the input to the activation function is $\tilde{a} + b$, where b is the bias term and

$$\tilde{a} = \frac{a - \hat{\mu}_a}{\hat{\sigma}_a}$$

with $a = \mathbf{w}^\top \mathbf{z}$ (no bias) and $\hat{\mu}_a$ and $\hat{\sigma}_a$ being the mini-batch estimated sample mean and sample standard deviation.
Batch normalization has two major benefits. With batch normalization we can use higher learning rates and less dropout, which speeds up the training. Further, it acts as a regularizer as it adds some noise to the hidden layer activations.

2.7 Parameter Tying and Sharing

Introduce dependencies between parameters (cf. convolutional neural networks as introduced in Section 5.1).

2.8 Dataset Augmentation

Create new data points by transforming existing ones in ways that the label/target variable stays the same, e.g. add some small noise to the features. This can be done pretty easily for image data in form

of image rotation, translation, scaling, or by adding noise, or changing the color saturation/hue, etc. Data augmentation for an example handwritten digit image is illustrated in Fig. 4.

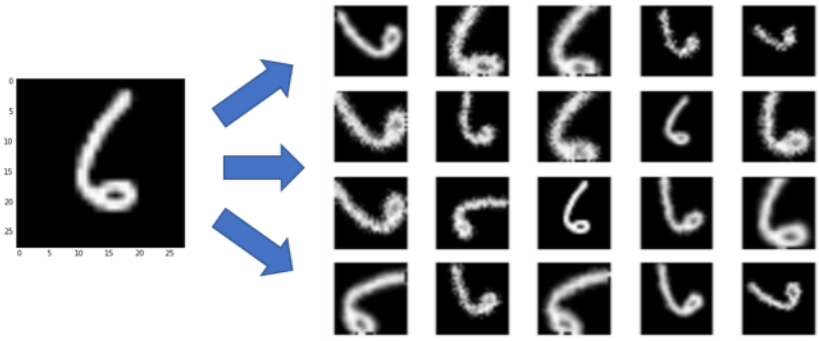
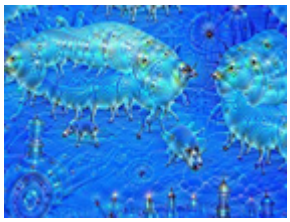


Figure 4: Example of data augmentation: add/remove noise, rotate, rescale, and/or shift input data.

In a similar vein, DeepDream perturbed images have also been used to enhance the training data with the goal of making the trained NN less sensitive to noise:

original (*left*)



perturbed image (*right*)

2.9 Adversarial Training

Adversarial Examples Construct adverserially perturbed versions of input points that the model would classify wrong ⇒ add those to the training set. To create such adverserial examples we may use a generative model such as a *generative adversarial network* (GAN), which is a type of autoencoder.

Min-Max Optimization Use a min-max optimization approach where the model tries to minimize the loss, while an *adversary* (often a separate model or a process) tries to maximize the loss by generating adversarial examples or simply adding noise to the input.

3 Optimization

Challenges in optimization:

- local minima
- saddle points, plateaus, flat regions
- inexact gradients
- too many parameters

Some commonly used methods:

(1) SGD: use **mini-batches** instead of single training examples

(2) use **adaptive learning rate** α (e.g. use line search $\alpha^* = \arg\min_\alpha \sum_{i=1}^n \ell_i$, or RMSprop, etc.)

(3) **momentum method** (best: combine with adaptive learning rate, cf. ADAM)

(4) conjugate gradients or BFGS (2nd order approximations that **avoid computing the Hessian**)

For details on those methods refer to Lecture 2 and this blog post which gives a good overview on GD optimization algorithms (http://ruder.io/optimizing-gradient-descent/).

4 Parameter Initialization

4.1 Random Initialization

Parameter initialization is very crucial for NN learning. Typically, we initialize W's and b's drawn randomly from a uniform or Gaussian distribution.
Note that if

- all weights are the same = all units are the same
- the weights are too small we get redundant units
- the weights are too large the parameter values explode during training resulting in overflow errors or saturated activation function gradients (*vanishing gradients problem*)

So, what scale to use?

$U(-\frac{1}{\sqrt{d}}, \frac{1}{\sqrt{d}})$ where d = number of inputs to first layer or $U(-\sqrt{\frac{6}{d+m_1}}, \sqrt{\frac{6}{d+m_1}})$ where m_1 = number of hidden units in the first layer. Typically the uniform distribution or a truncated Gaussian are preferred.

4.2 Greedy (or Stacked) Pretraining

In general, it is beneficial to initialize the networked with weights from a pretrained NN to leverage the benefits of transfer learning (cf. Section 2.4). If we do not have access to a pretrained network or have a good reason to use a differnt architecture, then we can use the following *greedy stratey* to pretrain the NN:

1. learn $W^{(1)}$ by only using on hidden layer
2. fix weights $W^{(1)}$
3. add another hidden layer and learn $W^{(2)}$
4. fix weights $W^{(1)}$ and $W^{(2)}$
5. continue layer-by-layer

$\Rightarrow$ fast as in every step we only train a NN with one hidden layer and then stack the network together step by step (*stacked learning*). However, how can we learn $W^{(1)}$ without $W^{(l)}$ $(\forall\, l > 1)$?

Supervised Pretraining

Train the supervised NN layer-by-layer and in each step omit $W^{(l)}$ for all later layers. That is we connect the current layer directly with the output layer in every step.

Unsupervised Pretraining

We can also pretrain the model using an unspuervised NN. Instead of connecting the current layer to the output layer, we connect it to a reconstruction layer that aims to reconstruct the original input. Typically we use an autoencoder as illustrated in Fig. 5.

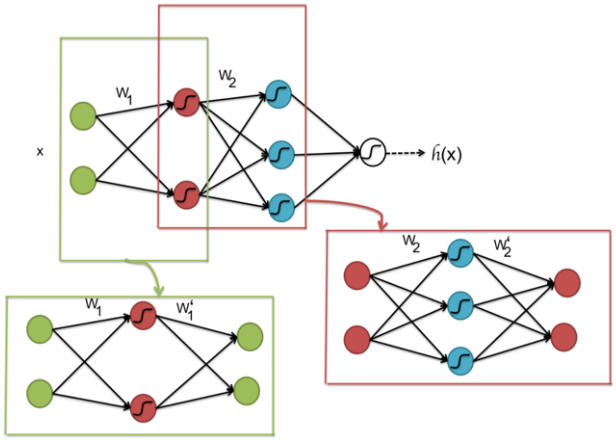


Figure 5: Illustration of stacked unsupervised pretraining (https://cs.stanford.edu/~quocle/tutorial2.pdf). First we learn the weights in the network in the green box, then fix those weights and learn the weights in the network in the red box.

Other options are to use latent variable models (LVMs) or restricted Boltzmann Machines. The use of these models is motivated by the insight that unsupervised pretraining is related to *representation learning* as the hidden layers can be viewd as modeling latent representations of the input data.

Insight: Pretraining is a form of **parameter initialization** and it also acts as a **regularizer**.

4.3 Transfer Learning

Instead of pretraining you own NN you can also use the parameters of a NN that was trained on data of the same domain to initialize your weights. Then you start training the NN with your own data on your specific learning task. Note that you will have to keep the network architecture the same as the pretrained network. Using a pretrained model (cf. transfer learning above) is typically preferred to greedy pretraining for domains where good pretrained networks exist (computer vision and natural language processing).

5 [Optional] Example Neural Network Models

NNs are very flexible ⇒ we can design all kinds of architectures for supervised and unsupervised ML tasks!

In the following, we will investigate **convolutional neural networks** (CNNs), which are used for *structured input domains* such as images (or graphs) and **recurrent neural networks** (RNNs), which are used for *sequential data* (speech, time series, etc.).
The most recent development are **transformers** that are based on all of those basic architectures and leverage the recent advances in being able to train NNs on wast amounts of training data in both unsupervised and supervised manners.

5.1 Convolutional Neural Networks (CNNs)

Goal: Create NNs for tensor data with spacial/temporal dependencies. Note that inputs may have different sizes (images for instance).

Recap: Discrete convolution (one dimensional)

$$s(t) = (x * w)(t) = \sum_{a=-\infty}^{\infty} x(a)w(t-a) \tag{2}$$

where $x(a)$ is the measurement and $w(t-a)$ is the weight dependent of distance to t. Hence, $s(t)$ can be seen as a weighted average estimating $x(t)$ using the surrounding measurements. Typically, $w(x) = 0$ for $|x| > thres$, this way the sum over a is actually a finite sum over a fixed size window as illustrated in Fig. 6.

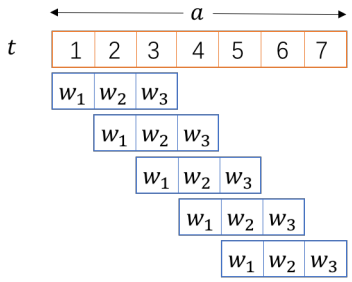


Figure 6: Illustration of a 1D convolution as a sliding window (*filter*) of size 3 over a sequence of length 7.

Extending this to two input dimesnions as we have for images leads to 2D convolution:

$$\begin{aligned} s(i,j) &= (I * K)(i,j) \\ &= \sum_m \sum_n I(m,n)K(i-m, j-n) \\ &= \sum_m \sum_n I(i-m, j-n)K(m,n), \end{aligned} \tag{3}$$

where K is the filter matrix.
Commonly used in NNs: $s(i,j) = \sum_m \sum_n I(i+m, j+n)K(m,n)$ (cross-correlation)

Main idea in CNNs: replace linear transformation with **multiple** convolutions (or cross-correlations)

$$W\mathbf{x} + b \rightarrow \begin{cases} s_1(i,j) \\ s_2(i,j) \\ s_3(i,j) \\ \vdots \end{cases}$$

where each $s_k(i,j)$ has a **different** filter matrix and the output of the convolution is a (3D) tensor with dimensions:

$$\frac{\text{input widths}}{stride} \times \frac{\text{input height}}{stride} \times \text{number of convolutions.}$$

The entries in the filter matrices correspond to the weights in a traditional FF-NN. These weights are illustrated for input inform aof a one dimensional array in Fig. 7. They need to be learned when training the CNN.

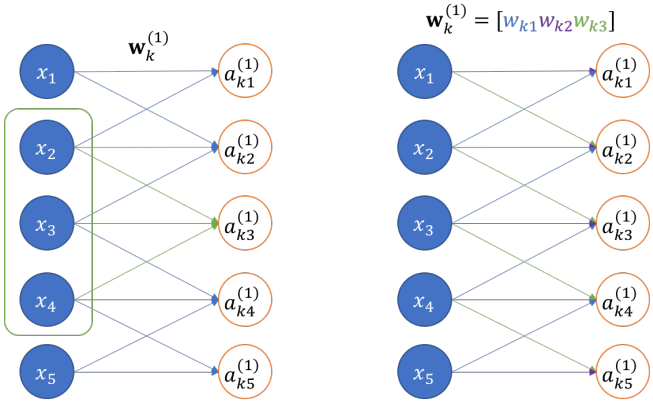


Figure 7: Illustration of **one** feature map $s_k(i)$ for a one dimensional array with filter weights $\mathbf{w}_k$, *receptive field* (*left*) and *parameter sharing* (*right*).

Note: each convolution is also a linear (affine) transformation.
Also, note that we do not have to move the convolution pixel-by-pixel. To reduce the dimensions in the next layer we can skip some number of pixels. The number of positions the filter is moved is called *stride*. After the convolution the a_{kl}'s get passed into the activation function.

Effects of convolution:

(1) **sparse connectivity**: each input dimension contributes to only a subset of the units in the next layer (cf. Fig. 7 (*left*))

(2) **parameter sharing**: same weights are used for all inputs (cf. Fig. 7 (*right*))

This gives us *translation invariance* w.r.t. **location** in the input.

How does a convolutional layer look like?

Vlayer input $\mathbf{x}$ → V $\underbrace{\text{convolution}}_{\text{affine transformation}}$ → V $\underbrace{\text{activation function}}_{\text{non-linear e.g. } relu}$ → V $\underbrace{\text{pooling}}_{\text{summary statistic}}$ → Vlayer output $\mathbf{z}$

Pooling gives us translation invariance to small changes in the **input values**. to prevent overfitting.

Common pooling functions:

- max-pooling: $z_j = \max_{z_{ji} \in \mathcal{N}_j} \{z_j, z_{j1}, z_{j2}, z_{j3}, ...\}$
- mean/average-pooling
- l_2-norm-pooling

CNNs can also be applied to multi-channel input such as RGB images. Now the input is a (3D) tensor as well.

5.2 Recurrent Neural Networks (RNNs)

Recurrent Neural Networks (RNNs) are used to model time-series and sequence data. One very prominent application is language modeling. See the **lecture slides** and http://colah.github.io/posts/2015-08-Understanding-LSTMs/ for more details.