# IN5400 – Data Augmentation, Initialization + a bit more state of the art

Alex

February 22, 2022

[The following notes are compiled from various sources such as textbooks, lecture materials, Web resources and are shared for academic purposes only, intended for use by students registered for a specific course. In the interest of brevity, every source is not cited. The compiler of these notes gratefully acknowledges all such sources. ]

**Takeaway points**

at the end of this lecture you should be able to:

- be able to give suggestions how to use images of varying sizes in a neural network with a fixed input size
- be able to use a neural network with a different input size than the one used at training time
- explain some example data augmentations
- explain how to choose a data augmentation parameter.
- it is not a vision-only approach
- the ideas how to use RandAugment and AutoAugment
- how to use weight standardization
- how to use noisy student
- understand how teacher student training works
- be able to explain the loss function in contrastive learning
- be able to explain how similar and dissimilar samples are generated in contrastive learning and the role of data augmentation

**external reading material:**

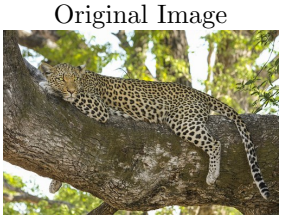
Chapter 13.1 in `https://d2l.ai/d2l-en.pdf`

The goal of this lecture is to give an introduction to data augmentation.

- **Problem 1: Input normalization** to match how the network was trained
- **Problem 2: The input dimensionality** of a neural network may not fit the dimensionality of the input data. We discuss ways to deal with it.
- **Problem 3: use neural network with a different than the specified input size** – yes one can!!
- **Problem 4: Data augmentation at Testing time**
  allow the neural net to see *multiple views* of the same sample. One creates multiple views of one sample. The final prediction will be an average of the prediction scores, averaged over all views. "Look at all aspects to understand a thing."

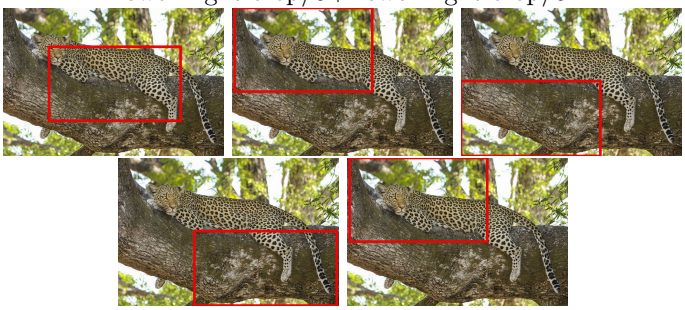
  Example in this class: Do not input an image as a whole, but input 5 crops: a cropped image from the center, and 4 corner crops.

  Original Image

  compute the prediction as an average over crops:

  prediction = Center crop/5+ upper left crop/5+ lower left crop/5+ lower right crop/5+ lower right crop/5

- **Problem 5: Data augmentation at Training time:** allow the neural net to be trained with more data samples. Data augmentation increases the data size by creating many similar samples from one sample.
- Example: training with slight photometric deformations of a sample (e.g. darker or brighter images) will allow the network to be able to recognize similarly brighter or darker images (relative to your original training images) when deployed at testing time.

This is very important, in particular for finetuning over small datasets.

There are two big important things which you are not going to learn when trying to get experience on MNIST and CIFAR, but which matter a lot in practice: data augmentation and transfer learning in a narrow sense.

- **Problem 6: Getting the scale of relevant objects right:** another example of a fixed, non-randomized augmentation for images (will see that in a later lecture):
  make the image size such that relevant structures (e.g. cats, cars) can be identified well.

  A neural network for images has convolutional kernels with a fixed size. During training their weights are learned and adapt to some structures.

  The sizes of input images at test time must be such that ... the structures to be identified at test time (e.g. cats, cars) – have similar size as – structures used at training time.

  If your training set contains only screen filling cat faces,



  then dont complain that you will not be able to predict cat correctly on this:



Your localized convolutional kernels were simply not trained to find structures on such small scales!
but data augmentation (or training a bounding box detector) can help with that – you can classify over many small windows.

- Above are not image-specific problems. The same problems (normalization, input dimensionality, multiple views, slightly augmented input data) arise when using a neural network e.g. for text processing, as the number of words is varying. Later we will see recurrent neural networks which can deal with inputs being sequences of arbitrary lengths (however even then it can make a big difference to deal with dimensionality properly).
- **Coding:** You will use a pretrained deep neural net to compute predictions for images.

# 1 Problem 1: Input normalization:

What inputs does a neural network expect ? As for images, common image readers produce subpixels in $[0, 255]$. the output of `PIL.Image` into numpy is usually of shape $(height, width, channels)$.

A neural network expects inputs often in $[0, 1]$ with shape $(batchsize, channels, height, width)$ with afterwards subtracted the mean of the training dataset.
For the conversion to $[0, 1]$ - in pytorch and mxnet `transforms.ToTensor()` takes care of that.

Often one has to do **further preprocessing**. The reason is: you have to do the same preprocessing steps at test time, as what was done at training time, otherwise train and test data have different distributions in input space. This usually results in heavily reduced performance.

Neural nets are usually trained with images, from which the training dataset mean is subtracted (that is: for every pixel). It was observed that training with samples which have on average over the dataset a zero mean results in faster convergence.
See `http://yann.lecun.com/exdb/publis/pdf/lecun-98b.pdf` for an explanation.
Furthermore, often input dimensions are also normalized by dividing them over a standard deviation (either channel-wise or pixel-wise) estimated over the training set.

$$subpixel[channel] = subpixel[channel] - trainset_mean[channel]$$
$$subpixel[channel] = \frac{subpixel[channel] - trainset_mean[channel]}{trainset_std[channel]}$$

In keras, that hosts many different modes, one can see at least three different modes:

`https://github.com/keras-team/keras-applications/blob/master/keras_applications/imagenet_utils.py`

One of them does not do division by standard deviation by default (`else:` – the caffe standard)

**The rule:**

Check what preprocessing was used at training time for the network you are going to use. Check what of that makes sense to be used at test time. Fail here = even prediction at test time with a pretrained net will fail (or training itself).

## 2 Problem 2: data input size vs nn input size without changing the network input

The default resnet deep neural net has an input size of $224 \times 224$ for images (inception $299 \times 299$). Your images usually are not of that size.
What can one do? One can resize the neural network for every image. We will talk later how to do that. But reinitializing the NN is ultra-slow, in particular when one wants to process a larger set of images.
Alternative is to adapt the image. One may resize the image to $224 \times 224$ with giving up the aspect ratio (height to width). Unusual – aspect ratio.more common:

...

- resize the image while preserving the aspect ratio such that the smaller size $s$ is $s \geq 224$ pixels, and then take a crop of size $224 \times 224$
- the crop can have multiple approaches: a **center crop** or a **random crop** of $224 \times 224$ with random position.

The same problem affects sequence analysis with 1d-cnns.

Recurrent neural networks can deal with sequences of arbitrary lengths, however training them with subsampled sequences of fixed length can be considered for improving performance.

## 3 Problem 3: use neural network with a different than the specified input size

Learning goal: you can most of the time use a pretrained neural network with a different input size than its original input size.

You want to classify over a larger input than $224 \times 224$, say $330 \times 360$ ? Yes you can.

The important thing to understand is: if you change the input size, then this does **NOT** affect the number of parameters in any convolutional layer, only its output sizes.
It can break at subsequent fully connected layers - but often it can be fixed. So if there is a fully connected layer, with a pooling or flatten layer before it, then one must change the parameters of the pooling or flatten layer to become a global pooling with a fixed output size.

when resizing input to a network

- Convolution layers: Resizing the input of a neural network changes the sizes of resulting feature maps, but for convolution layers it does **not change their number of parameters** – therefore loading pretrained convolution layers will still work.
- Dense Layers: Resizing the input of a neural network **changes the number of parameters** in fully connected/dense layers. In order to ensure the same number of parameters in fully connected layers with changed size as with original input size, one needs to ensure that the number of inputs to fully connected layers stays the same. If the fully connected layer is preceded by a pooling layer, then the pooling layer can be modified to solve this problem.
- Useful: Global pooling/adaptive pooling are pooling methods where the output size is fixed and instead the pooling kernel size is changing dependent on the input to match the desired output size.

Lets look at a resnet 18 to understand this: `https://github.com/pytorch/vision/blob/master/torchvision/models/resnet.py` in `class ResNet(nn.Module):` and `def resnet18(pretrained=False, **kwargs):`
Consider how the forward pass works `def forward(self,x)`:

$(224, 224) \rightarrow$ many conv layers $\rightarrow (512, 7, 7) \rightarrow$ Pooling by `self.avgpool`
$\rightarrow (512, 1, 1) \rightarrow x = x.view(x.size(0), -1) \rightarrow (512) \rightarrow self.fc- > (1000)$

```
self.avgpool = nn.AdaptiveAvgPool2d(...)
```

This use ensures that rescaling of the input will not affect the input to the dense layer

The same problem affects sequence analysis with 1d-cnns.

# 4 Problems 4 and 5: Data augmentation

*One does not simply predict on a raw sample. One does not simply train with a raw sample.*
**Using data augmentation is a fundamental standard in deep learning.**
Unfortunately, many tutorials skip this.

Examples of data augmentation at test time for images:

- random crops of a fixed size of the image
- corner and center crops (see above)
- mirroring along the vertical axis

**Test time data augmentation**

At test time data augmentation is often used in an averaging sense. Let $x$ be an input sample, $f(\cdot)$ a predictor, and $A_a$ the a-th augmentation (e.g. the a-th crop of an image, a randomized rotation), then the prediction is computed as an average over a set of chosen augmentations:

$$f_{avg}(x) = \frac{1}{n_A} \sum_{i=1}^{n_A} f(A_{a,i}(x)), \text{ param } a \text{ deterministic or } a \sim P_\theta(a)$$

Here the average is computed over augmentations of the same input sample $x$.

The idea of data augmentation at test time is to predict on an average of multiple views of the same input.

**in pytorch?**

`https://pytorch.org/vision/stable/transforms.html`

**Training time data augmentation**

The idea of data augmentation at training time is to extend the training data available by creating instances

- which can be **plausibly** derived from the training data set
- which are plausible to have the same ground truth label as their origin
- which are plausible to be seen in the test set.
- which sometimes are used to encode invariances that one wants to have in the learning process (e.g. recognition independent of whether object is rotated ... (!))

Thus to increase the training dataset.

$$f(x) = f(A_a(x)), \ a \sim P_\theta(a)$$

**Learning theory perspective: Data augmentation vs the iid assumption???**

Examples of data augmentation at training time for images:

- geometric distortions:

  – rotations - they make sense sometimes up to a certain degree, some imaging problems are suitable for arbitrary rotations

  – mirroring

  – distort an MNIST digit by e.g. shear. Not used often outside of MNIST or similar shape recognition tasks.

- photometric distortions (see Chapter 13.1 in `https://d2l.ai/d2l-en.pdf`): modify brightness, and contrast/gamma of an image. for some scenario (e.g. medical stainings) one can think also to play with hues (colors).

  Contrast change (e.g. `https://www.dfstudios.co.uk/articles/programming/image-programming-algorithms/image-processing-algorithms-part-5-contrast-adjustment/`) we assume that each subpixel lies in $[0, 255]$.

  $$sub = C \times (sub - 128) + 128, C > 0$$

  For randomized image augmentation purposes the value $C$ is drawn for every image and minibatch from a random distribution such as

  $$C \sim Uniform[a, b]$$

  The random distribution has usually parameters, here $a$, $b$ which need to be selected on a validation dataset, but not on the test set.

  Brightness change:

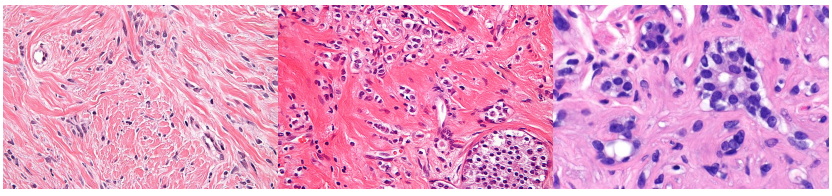
  $$sub = C + sub$$

  Gamma change:

  $$sub = 255 * \left(\frac{sub}{255}\right)^{C}, C > 0$$

  The hue is a value for the color in HSV colorspace.

  An example where playing with hues a little bit may make sense: Below is the same stain Haematoxylin and Eosin, of the same tissue. Different lab people create different color intensities!



For one quick intro on photometric distortions : Andrew Howard, Some Improvements on Deep Convolutional Neural Network Based Image Classification `https://arxiv.org/pdf/1312.5402.pdf`. See also the googlenet paper.

**General data augmentation**

is a transformation of an input $x$

$$A(x)$$

Often it has a parameter $a$, e.g. value of brightness to add

$$A_a(x)$$

Such parameters, are at training time often drawn from a random distribution

$$a \sim P_\theta(a)$$

The parameters of the random distribution are hyperparameters, and need to be validated on a validation set.

**Augment to what degree?**

Data augmentation methods depends typically on some parameter. One needs to

- train with multiple choices of the parameter,
- find the best parameter by looking at validation dataset errors.
- Once all parameters have been selected, then one does one final evaluation on the test data set – in order to check that one did not overfit by optimizing parameters by looking at the validation error.

## 4.1 Augmentation outside of vision

- generic option: add gaussian noise
- generic option: add gaussian noise in feature space (e.g. after fourier transform)
- out of exams: riskier options – generative methods like seq2seq learning and cycleGANs
- Examples of data augmentation at training time for text sentences?
- cycle-gan inspired (Jigsaw toxic comment classification):
  - text → other language → translate back

- out of exams: Noising in NLP: `https://arxiv.org/pdf/1703.02573.pdf`, ICLR 2017
- out of exams: use a language model to suggest augmentations: `http://aclweb.org/anthology/N18-2072`
- Audio: modify the speed mildly `https://openreview.net/pdf?id=HyaF53XYx`
- Vision (sorry): Neural style transfer – surely can be applied also to audio!

**Train time data augmentation**

Data augmentation is used at training time in a randomized fashion – because mild randomization often helps against overfitting.

- at training time: *randomized transforms* – The randomization depends on the transformation. Example: For image brightness, one choose a random add within some range, like $[-0.2, 0.2]$, every time an image is loaded, so for one minibatch one draws a vector of random numbers).
- at training time: often one keeps an image as it is with $p$ probability, and applies randomized transforms with $1 - p$ probability. Every time an image batch is loaded, one chooses a new parameter from a random distribution.
- **one does usually NOT apply these distortions on the validation or test set !! (except one wants averaging over multiple views)**

**important**:
The probability distribution of the distortion parameter has hyperparameters, that needs to be validated on the validation dataset.

$$\text{argmin}_w \frac{1}{n} \sum_{i \in \text{ minibatch}} L(f_w(A_i(x_i, \text{params})), y_i)$$

where $A_i(\cdot, \text{params})$ is a randomized augmentation, e.g. brightness modification with a randomly drawn value, which varies for every image during every training epoch.

Compare to the augmentation at test time – for which the average is computed over different augmentations of the same input sample $x$.

Augmentation can be also applied to the ground truth, adding some noise to the labels! Example: adding noise to regression ground truth, moving boundary boxes of detections, moving segmentation boundaries

**Dont forget this one**

`model.eval()` whenever you compute test/val scores or you will predict only nonsense, and you cant see why it fails

# 5 How to select Data Augmentation Parameters?

Standard approach: try out several settings. Choose best according to performance on validation data

Problem: many data augmentation choices. Each has parameters.
Systematic search on validation data is too expensive (see AutoAugment search space)!

## 5.1 RandAugment

Cubuk et al. 2019 `https://arxiv.org/abs/1909.13719`

Given 14 transforms

```
transforms = ['Identity', 'AutoContrast', 'Equalize','Rotate', 'Solarize',
 'Color', 'Posterize','Contrast', 'Brightness', 'Sharpness',
 'ShearX', 'ShearY', 'TranslateX', 'TranslateY']
```

Simple idea:

- given hyperparameters $N, M$:
  - $N$ number of transformations applied to one sample sequentially
  - $M$ intensity of distortion
- do not care about order of them or which ones
- randomly draw any $N$ with uniform probability*!
- define strength of each augmentation by an integer $0, \ldots, 10$ by linearly partitioning a maximal range (based on Table 6 in Cubuk et al.`https://arxiv.org/abs/1805.09501`)
  - apply each transformation with the same strength $M \in 0, \ldots, 10$
- optimize only over hyperparameters $M, N$

In result almost as good as searching for specific policies (e.g Table 4 in the paper, factor $10^{32}$ faster).

(* see section 4.7 in the paper – tried optimization with $\alpha_{ij}$ - the probability to use transformation $i$ in step $j$ for $N = 2$ – expensive, needs $14N$ transformations to be tried out on every image.)

## 5.2 UniformAugment

Chen et al. 2020 `https://arxiv.org/abs/2003.14348`

Idea: do not search anymore if you know a reasonable maximal range for each transformation ( if ... !).

- use $N = 2$. draw a sequence of two augmentations with uniform probability $p(T) = \frac{1}{|T|}]$
- draw randomly apply/drop probability from $U([0, 1])$, draw randomly augmentation strength from $U([0, 1])$, where value of one corresponds to the maximum strength $M$
- apply augmentation
- out of class: A bit bold motivation. assumption: for the right choice of space $M$ for augmentation strength, given one real image $x$, then applying transformations $T_\lambda(x), \lambda \sim U([0, 1])$ covers the space of all realistic images with a good match of probability of occurrence.

**A word of caution**

These 14 transformations were found to be useful on general ImageNet images.
This may not hold for other domains, eg. medical or satellite images. It may not hold when your domain has much fewer images than imagenet.

# 6 Neural Net initialization

`http://neuralnetworksanddeeplearning.com/chap3.html#weight_initialization`
`https://arxiv.org/pdf/1502.01852.pdf`
need to initialize parameters $w$ of a neural network. These are current guidelines:

**Initialization for ReLU networks**

- set biases to zero $b = 0$
- initialize weights as random values for symmetry breaking
- conv layers: draw weights from a normal with standard deviation equal to $\sigma = \sqrt{\frac{2}{n}}$
  $w_d \sim N(0, \sigma^2)$
- later: use transfer learning instead of training with random init
- older Xavier-Glorot style initialization is considered obsolete for deep networks while it still works for shallow networks

**Initialization for pReLU networks**

- set biases to zero $b = 0$
- initialize weights as random values for symmetry breaking
- conv layers: draw weights from a normal with standard deviation equal to $\sigma = \sqrt{\frac{2}{(1+a)^2 n}}$ where $a$ is the negative slope
  $w_d \sim N(0, \sigma^2)$
- later: use transfer learning instead of training with random init

## 6.1 Neural Net initialization: why random weights?

If neurons in one layer and their input neurons receive the same weight, then their forward pass values might be the same and gradient updates might be the same.

Then these neurons learns to encode the same thing.

## 6.2 Neural Net initialization: choice of variance

How to arrive at drawing weights from a zero mean normal distribution with variance equal to $\frac{2}{D_{in}}$?

**Motivation for the parameters**

You are not required to memorize the deductions below in this section. You should understand this:

The parameters are chosen so that

- variances of feature maps in the forward pass are approximately equal across layers.
- variances of gradient norms in the backward pass are approximately equal across layers.

Consider a linear neuron. $x$ are the inputs from a previous layer, which itself might be the outputs of a preceding relu on the layer $y^{(l-1)}$ which was computed before $y^{(l)}$:

$$y^{(l)} = \sum_{d=1}^{n^{(l)}} w_d^{(l)} x_d^{(l)} + b$$
$$x_d^{(l)} = \max(0, y^{(l-1)})$$

The basic idea is: initialize $w_d$ such that the variance of outputs at initialization is preserved through layers:

$$Var(y^{(1)}) = Var(y^{(2)}) = \ldots = Var(y^{(l-1)}) = Var(y^{(l)})$$

Furthermore we assume:

- $w_d$ will be drawn from a random distribution with zero mean $E[w_d] = 0$
- $b = 0$ at initialization
- $x_d$ is statistically independent of $w_d$, thus $E[f(x_d)g(w_d)] = E[f(x_d)]E[g(w_d)]$

How to go?

$$Var(y) = Var(\sum_{d=1}^{n^{(l)}} w_d x_d) = \sum_{d=1}^{n^{(l)}} Var(w_d x_d) = n^{(l)} Var(wx)$$

This gives one result: $n^{(l)}$ will be a parameter involved in the initialization of $w_d$. Furthermore using $E[w_d] = 0$ and that $x_d$ is statistically independent of $w_d$, we can obtain:

$$Var(wx) = E[(wx)^2] - (E[wx])^2 = Var(w)E[x^2]$$

note for the above: $E[w^2x^2] = E[w^2]E[x^2]$ due to the independence assumption.

The next step is to derive $E[x^2]$ from $Var(y^{(l-1)})$. The following argument:

- if $w_d^{(l-1)}$ are a symmetric distribution and have zero mean, then $y^{(l-1)} := w^{(l-1)}x^{(l-1)}$ are also symmetric with zero mean, no matter how $x^{(l-1)}$ was distributed – this is due to the assumption that $x$ is statistically independent from $w$, therefore $P(X \in A, W \in B) = P(X \in A)P(W \in B)$

- symmetric around a mean of zero means $\forall t > 0 : P(y^{(l-1)} < -t) = P(y^{(l-1)} > t)$ which is $= 1 - P(y^{(l-1)} < t)$
- densities are the differential of cdfs,

  $$f(t) = \frac{d}{dt} P(y^{(l-1)} < t)$$

  therefore we have for a symmetric probability:

  $$\begin{aligned} P(y^{(l-1)} < -t) &= 1 - P(y^{(l-1)} < t) \\ \Rightarrow \frac{d}{dt} P(y^{(l-1)} < -t) &= \frac{d}{dt}(1 - P(y^{(l-1)} < t)) \\ \Rightarrow f(-t) &= -f(t) \end{aligned}$$

- Now

  $$\begin{aligned} Var(y^{(l-1)}) &= \int_{z=-\infty}^{\infty} (z - E[z])^2 f(z) dz \\ &= \int_{z=-\infty}^{0} z^2 f(z) dz + \int_{z=0}^{\infty} z^2 f(z) dz \\ &= \int_{z=0}^{+\infty} (-z)^2 f(-z) \cdot (-1) dz + \int_{z=0}^{\infty} z^2 f(z) dz \\ &\overset{\text{symm!}}{=} 2 \int_{z=0}^{\infty} z^2 f(z) dz = 2 \int_{z=-\infty}^{\infty} \max(0, z)^2 f(z) dz = 2E[x^2] \end{aligned}$$

  this used the anti-symmetry of the cdf.

Putting that together results in

$$Var(y^{(l)}) = n^{(l)} Var(w^l x^l) = Var(w^l) E[x^2] = Var(w^l) \frac{1}{2} n^{(l)} Var(y^{(l-1)})$$

Iterating this over layers $l$ yields:

$$Var(y^{(l)}) = \prod_{r=1}^{l} \left( Var(w^r) \frac{1}{2} n^{(r)} \right) Var(y^{(1)})$$

From here, to keep the variance constant we need:

$$\begin{aligned} \forall l \; Var(w^l) \frac{1}{2} n^{(l)} &= 1 \\ \forall l \; Std(w^l) &= \sqrt{\frac{2}{n^{(l)}}} \end{aligned}$$

This is the derivation of the Kaiming initializer.

## 6.3 out of class: why people used Xavier initialization?

The old Xavier Glorot-style argument ignored the symmetry breaking by the relu, thus assumed $E[x^{(l)}] = E[y^{(l-1)}] = 0$ (which makes sense for antisymmetric sigmoids). Furthermore it assumed $var(x) = var(y^{(l-1)})$ Under such an assumption

$$Var(wx) = Var(w)E[x^2] = Var(w)var(x) = Var(w)var(y^{(l-1)})$$
$$Var(y^{(l)}) = \prod_l \left(Var(w^l)n^{(l)}\right) Var(y^{(1)})$$

This results in a heuristic $Std(w^l) = \sqrt{\frac{1}{n^{(l)}}}$ which however does not work well often.

Xavier initialization considered also the variance flow during backward pass

$$\frac{\partial L}{\partial z_k} = \sum_{d=1}^{n_{out}} \frac{\partial L}{\partial z_d}\frac{\partial z_d}{\partial z_k} = \sum_{d=1}^{D_{out}} \frac{\partial L}{\partial z_d} z_d^{'}(z) w_{kd}$$

In case of independence of terms we arrive at

$$Var(\frac{\partial L}{\partial z_k}) = n_{out} Var(\frac{\partial L}{\partial z_d} z_d^{'}(z) w_{kd})$$

This considers also $n_{out}$. The requirement for $n_{in}$ and $n_{out}$ can be heuristically fused by taking the harmonic mean of variances $\frac{1}{n_{out}}$ and $\frac{1}{n_{out}}$, resulting in a variance of $\frac{2}{n_{in}+n_{out}}$.

Aiming at a variance of $\frac{2}{n_{in}+n_{out}}$ and drawing from a uniform distribution $w_d \sim U(-a, a)$ which has a variance of $\frac{1}{12}(2a)^2 = \frac{a^2}{3}$, this requires to set $a = \frac{\sqrt{6}}{\sqrt{n_{in}+n_{out}}}$. Note that the variance $\frac{2}{n_{in}+n_{out}}$ is the harmonic mean of variances $\frac{1}{n_{out}}$ and $\frac{1}{n_{out}}$.

# 7 SOA 2019/2020/2021

today:

- Weight standardization
- Noisy student with Efficientnet `https://arxiv.org/abs/1911.04252`
- contrastive learning (maybe the next lecture)

## 7.1 Weight standardization

Group Norm + Weight standardization + Batch size 1 seems to outperform Batch normalization.

Qiao et al. `https://arxiv.org/abs/1903.10520`
Simple idea:

- vanilla convolution is locally $conv(x) = W \star x$ with some kernel $W$, $W.shape = (C_{out}, k, k, C_{in})$, $W \in \mathbb{R}^{C_{out} \times k \times k \times C_{in}}$
- motivation: batchnorm decouples gradient flow from scale of weight
- can be also achieved by normalizing the weights directly!
- reshape / regroup kernel into $W \in RR^{O \times I}$, $O = C_{out}$ output dimensions, $I = C_{in} * k * k$ input dimensions,
- replace $conv(x) = W \star x$ by a convolution with standardized weights

$$conv(x) = \widehat{W} \star x$$
$$\widehat{W} = WS(W)$$

where $WS(W) = (\widehat{W}_{o,i})_{i \in I, o \in O}$ is the weight standardization function given as:

$$\widehat{W}_{o,i} = \frac{W_{o,i} - \mu(W_{o,:})}{\sigma(W_{o,:})}$$
$$\mu(W_{o,:}) = \frac{1}{I} \sum_{i=1}^{I} W_{o,i}$$
$$\sigma(W_{o,:}) = \sqrt{\frac{1}{I} \sum_{i=1}^{I} W_{o,i}^2 - \mu^2(W_{o,:})} = \sqrt{\hat{E}_{i \sim I}[W_{o,i}^2] - \hat{E}_{i \sim I}[W_{o,i}]^2 + \epsilon}$$

Compare to standardization of a random variable:

$$\hat{X} = \frac{X - \mu}{\sqrt{\sigma^2}}$$

- compute predictions and the loss using convolutions with standardized weights

$$conv(x) = \widehat{W} \star x = WS(W) \star x$$

- optimize loss with respect to the raw weights $W$ . Means: gradient update is applied on the raw, unnormalized weights.

Out of class: Paper delivers on explanation in terms of reducing the Lipschitz constants

## 7.2 Noisy student

Xie et al. `https://arxiv.org/abs/1911.04252`

**Require:** Labeled images $\{(x_1, y_1), (x_2, y_2), ..., (x_n, y_n)\}$ and unlabeled images $\{\tilde{x}_1, \tilde{x}_2, ..., \tilde{x}_m\}$.

1: Learn teacher model $\theta_*^t$ which minimizes the cross entropy loss on labeled images

$$\frac{1}{n}\sum_{i=1}^{n} \ell(y_i, f^{noised}(x_i, \theta^t))$$

2: Use a normal (i.e., not noised) teacher model to generate soft or hard pseudo labels for clean (i.e., not distorted) unlabeled images

$$\tilde{y}_i = f(\tilde{x}_i, \theta_*^t), \forall i = 1, \cdots, m$$

3: Learn an **equal-or-larger** student model $\theta_*^s$ which minimizes the cross entropy loss on labeled images and unlabeled images with **noise** added to the student model

$$\frac{1}{n}\sum_{i=1}^{n} \ell(y_i, f^{noised}(x_i, \theta^s)) + \frac{1}{m}\sum_{i=1}^{m} \ell(\tilde{y}_i, f^{noised}(\tilde{x}_i, \theta^s))$$

4: Iterative training: Use the student as a teacher and go back to step 2.

**Algorithm 1:** Noisy Student Training.

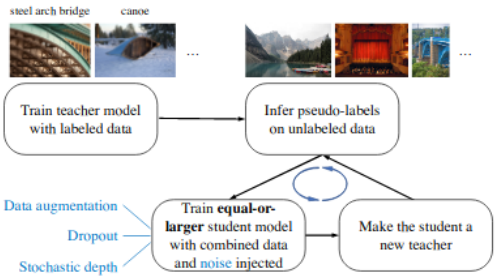



Figure 1: Illustration of the Noisy Student Training. (All shown images are from ImageNet.)

nothing more to say

- makes use of additional unlabeled data, labels it by the teacher to obtain probability distribution per image as label
- uses soft labels computed by the teacher on original images, but for training adds noise to the student model for the same images
- combines semi-supervised data, weakly supervision and model noising

A word of caution

probably needs a sufficient strong teacher to start with. likely wont work with poorly performing teachers...

## 7.3 Background: teacher-student training

- goal: want to train a network (the student)
- idea: do not train it using hard ground truth labels. Given a sample $x$, obtain softmax probability distribution $t(x)$ from the teacher first
- train student $s(\cdot)$ with the teacher probabilities $t(x)$ as soft labels for a loss capable to do so, e.g. using cross-entropy

  Reminder: Cross entropy of two distributions $p$, $s$:

  $$CE(p, q) = \sum_{c=1}^{C} -p_c \log s_c$$

  Reminder: if $p$ are one-hot ground truth labels, and $s = s(x)$ is the softmax prediction over an image, this reduces to:

  $$-\log s_c(x) \text{ where } c \text{ is the ground truth class of } x$$

  the neg log probability of the prediction for the ground truth class.

  The cross-entropy between teacher and student are the teacher-weighted neg log student probabilities:

  $$L(x, t(x)) = \sum_{c=1}^{C} -t_c(x) \log s_c(x)$$

  **important:** trained is only $s_c$, the gradients are computed only with respect to parameter of $s_c$!
- why does it work well ? The losses are lower for hard samples where $t_c(x)$ is close to the guessing threshold $\frac{1}{C}$ for most classes. Losses are larger for easy samples, where the teacher is very confident and predicts $t(x)$ close to a one-hot-label.

  Idea here: effectively prioritizes the learning of easy samples! (Out of class: see curriculum learning in machine learning). (out of class, opposite idea: hard negative mining for problems with too many negative samples)
- applications: usually used to train structurally smaller/lighter students from complex teacher models, e.g. to obtain a faster or more resource efficient model (Noisy student trains a model of the same complexity)

# 8 Contrastive Learning

Coarse Idea:

- Pretrain a network without millions of labeled samples such as imagenet. Use instead a **large set** of unlabeled samples.
- Use data augmentations to train a classifier over similarity outputs.
- two data augmentations from the same image define the same class, should have high similarity score

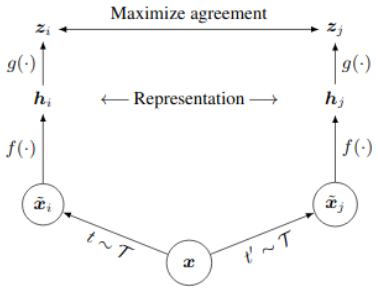



*Figure 2.* A simple framework for contrastive learning of visual representations. Two separate data augmentation operators are sampled from the same family of augmentations ($t \sim \mathcal{T}$ and $t' \sim \mathcal{T}$) and applied to each data example to obtain two correlated views. A base encoder network $f(\cdot)$ and a projection head $g(\cdot)$ are trained to maximize agreement using a contrastive loss. After training is completed, we throw away the projection head $g(\cdot)$ and use encoder $f(\cdot)$ and representation $\boldsymbol{h}$ for downstream tasks.

  - same image: train parameters of $f$ and $g$ such that the vectors $z_i, z_j$ have high similarity if $x_i$ and $x_j$ are created by data augmentations from the same base image $x$

- two data augmentations from two different images define two different classes and should have low similarity scores
- high level view: pretrains a network representation using an auxiliary task – see analogies to the learning of word embeddings (CBoW, skip-gram, Mikolovs paper).

A few Papers:

- MoCo He et al. `https://arxiv.org/abs/1911.05722`, MoCo v2 Chen et al. `https://arxiv.org/abs/2003.04297`
- SimCLR Chen et al. `https://arxiv.org/abs/2002.05709`, SimCLR v2 `https://arxiv.org/abs/2006.10029`

What for this can be used?

- if you want to predict on Imagenet, then currently pretraining with imagenet labels is still better but ...

- potential use case 1: when comparing for initialization for finetuning (1) weights from imagenet vs (2) contrastive, then contrastive learning can outperform imagenet-based initialization. see table 1 in the Moco v2 paper.
- potential use case 2: **apply contrastive learning** on target domain unlabeled data **after loading imagenet weights**
  - suppose you have a target domain with much unlabeled data, then: use after loading imagenet weights, apply contrastive learning. Combine then with finetuning for the few labeled samples.
  - concretely: SimCLR v2 shows that when using a strategy similar to noisy students, that contrastive learning with unlabeled data and 10% labels of imagenet comes close to the performance as training imagenet conventionally with 100% labels – but it requires and profits from deeper networks than supervised training.

  Promising direction: can finetune better on a dataset with few labels and a big amount of unlabeled data.

- in terms of messing up potential, plain finetuning is easier to handle.
- needs lots of tricks to work properly. Next: show some key architectural components to make it work.

**Steps used in contrastive learning:**

- loss function:

  SimCLR:

  - draw always a pair of samples (indexed by $(2i-1, 2i)$) by randomized data augmentation from the same image.
  - draw N pairs from $N$ different images, thus having $2N$ samples in total

$$
\begin{aligned}
sim(u, v) &= \frac{u^T v}{\|u\|\|v\|} \text{ (cosine angle)} \\
L_{2i-1,2i} &= -\log \operatorname{softmax}(\{\text{ all pairs } (i,j)\})[(2i-1, 2i)] \\
&\quad -\log \frac{\exp(sim(z_{2i-1}, z_{2i})/\tau)}{\sum_{k=1}^{2N} \mathbb{1}_{[k \neq i]} \exp(sim(z_i, z_k)/\tau)} \\
L &= \frac{1}{2N} \sum_{i=1}^{N} L_{2i-1,2i} + L_{2i,2i-1}
\end{aligned}
$$

  $L_{2i-1,2i}$ – neg log Softmax over 2N similarities as outputs – for the slot/ component where two images should be similar

  MoCo/ MoCo v2:

- draw $q$, $k^+$ by randomized data augmentation from the same image.
- in theory: draw a set $\{k^-\}$ by randomized data augmentation from other images than the one used for $k^+$ and $q$
- in practice: $\{k^-\}$ comes from a queue of features from augmented images from past training minibatches (faster by reusal)

$$\begin{aligned} L_{q,k^+,\{k^-\}} &= -\log \text{softmax}(((q,k^+),(q,k_1^-),(q,k_2^-),\ldots,(q,k_N^-)))[(q,k^+)] \\ &= -\log \frac{\exp(q^T k^+/\tau)}{\exp(q^T k^+/\tau) + \sum_{k^-} \exp(q^T k^-/\tau)} \end{aligned}$$

Softmax over $1 + |\{k^-\}|$ similarities as outputs

Both losses enforces data augmentations from the same image to be similar, and data augmentations from different images to be dissimilar.

- MLP head on top of representation to be trained (SimCLR, MoCo v2):

  take feature map $h$ from the representation layer (example for SimCLR: feature maps from variants of ResNet50 after the global pooling), then pass it through: linear-ReLu-linear:

  $$z = g(h) = W_2 \star R(W_1 \star h)$$

  Moco v2: same idea, output of $W_1$ is 2048-dim

  SimCLR v2: same idea, but three layers of weights, not only 2.

  It is important for performance to use more than one linear layer on top of $h$.

- representation below the MLP head:
  - SimCLR: identical model $f(\cdot)$ for all samples (variants of ResNet50 after the global pooling)

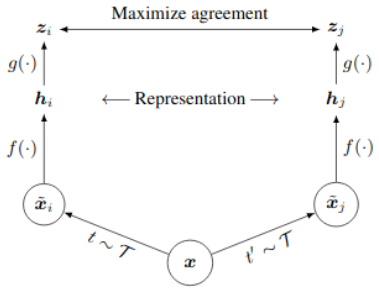



*Figure 2.* A simple framework for contrastive learning of visual representations. Two separate data augmentation operators are sampled from the same family of augmentations ($t \sim \mathcal{T}$ and $t' \sim \mathcal{T}$) and applied to each data example to obtain two correlated views. A base encoder network $f(\cdot)$ and a projection head $g(\cdot)$ are trained to maximize agreement using a contrastive loss. After training is completed, we throw away the projection head $g(\cdot)$ and use encoder $f(\cdot)$ and representation $\boldsymbol{h}$ for downstream tasks.

serious drawback: very large batchsizes necessary 4096 or 8192. Not feasible with high end GPUs as of end of 2020. Does Google want to promote its TPUs? ;-)

- MoCo v2: using two different models. One to encode query $q$ and one for similar and dissimilar keys $k^+, \{k^-\}$

  * idea: have a query encoder $f_{\theta_q}(\cdot)$ for encoding the positive query $q$.

  * $\theta_q$ (query encoder parameters) is updated by backpropagation of the loss.

  * have a key encoder $f_{\theta_k}(\cdot)$ for $k^+, \{k^-\}$ - which is **not** backpropagation trained .

  * update key encoder parameters $\theta_k$ by applying a slow momentum $\alpha = 0.999$ to incorporate the backprop-learned query encoder parameters:

$$\theta_{k,t+1} = 0.999\,\theta_{k,t} + 0.001\,\theta_{q,t+1}$$

  * advantage: GPU-feasible batchsizes (256) - but see the moco v2 paper, you still may need a rack of 4 or 8 gpus as office heating

- data augmentations:

  SimCLR: random crop (with resize and random horizontal flip $p = 0.5$ chance to apply the flip), random color distortion ($p = 0.5$ chance to apply the jitter), and random Gaussian blur ($p = 0.5$ chance to apply the blur, $\sigma \sim U(0.1, 2.0)$)

  MoCo v2: random crop (with resize and random horizontal flip $p = 0.5$ chance to apply the flip), random color distortion ($p = 0.5$ chance to apply the jitter), random horizontal flip, random greying, and random Gaussian blur

out of exams: SimCLR v2: `https://arxiv.org/abs/2006.10029`

- Fig 3: pretrain with a deeper head, keep first layer of added head (VGG is ...)!
- then finetune on labeled part of target domain
- then: eq 2 and eq 3 is comparable noisy student training (see before :) ).
- add momentum encoder from MoCo