Neural Networks for Data Science Applications

Master's Degree in Data Science

Lecture 8: Building deep neural networks

Lecturer: S. Scardapane

SAPIENZA
UNIVERSITÀ DI ROMA

Introduction

A few bits of history

The basic building blocks we saw up to now are relatively old: in 1998, a team at Bell Labs already had a working CNN (5-7 layers) for handwritten digits recognition, termed LeNet-5.

However, from 2012 onwards, the combination of more computational power (especially GPUs), data, and a few algorithmic improvements quickly made deep CNNs the absolute state-of-the-art across several domains.

LeCun, Y., et al., 1998. **Gradient-based learning applied to document recognition**. *Proceedings of the IEEE*, 86(11), pp.2278-2324.

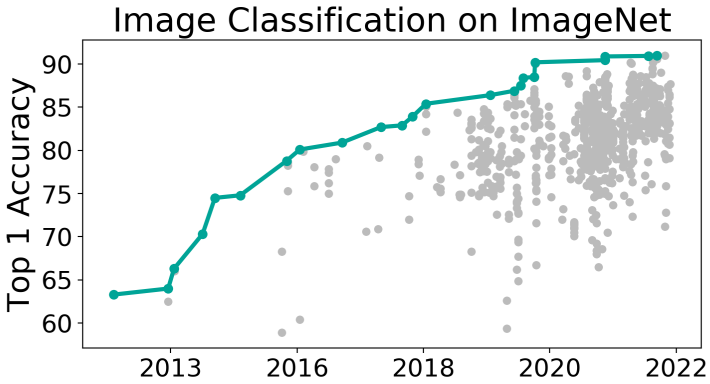


Figure 1: Evolution of accuracy on the ImageNet Large Scale Visual Recognition Challenge (ILSVRC).

AlexNet was the first CNN to win the ILSVRC image classification competition by a large margin.[1]

It had 8 adaptable layers (5 convolutional, 3 fully-connected). For training, it exploited several ideas, some of which relatively novel at the time:

- ReLU activation instead of sigmoid-like functions;
- Data augmentation and dropout to handle overfitting.

[1] Krizhevsky, A., Sutskever, I. and Hinton, G.E., 2012. **Imagenet classification with deep convolutional neural networks**. In *Advances in neural information processing systems* (pp. 1097-1105).

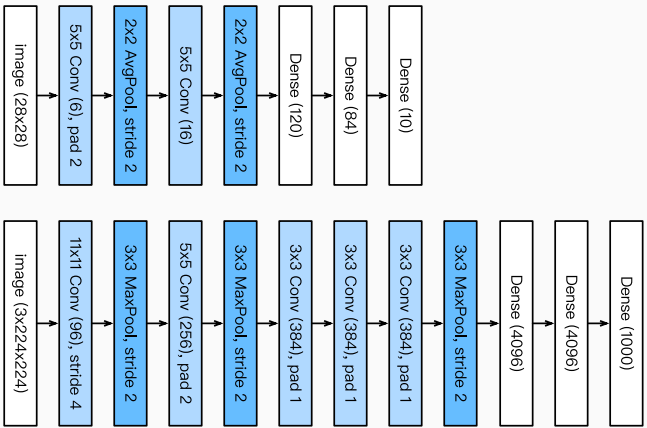


Figure 2: Top: LeNet (1998), bottom: simplified version of the original AlexNet (2012). Source: Dive Into Deep Learning, Chapter 8.1.

The Oxford's **Visual Geometry Group** (**VGG**) in 2014 popularized the idea of defining **blocks** composed of several layers, from which variants of a given architecture can be made according to a predefined **scaling recipe**.

Their proposed block was very simple:

1. Several convolutional layers with size 3×3 and the same number of filters;
2. A single max-pooling block with 2×2 windows at the end of the block.

In the VGG architecture, filters are generally doubled after one or two blocks.

Simonyan, K. and Zisserman, A., 2014. **Very deep convolutional networks for large-scale image recognition**. *arXiv preprint arXiv:1409.1556*.

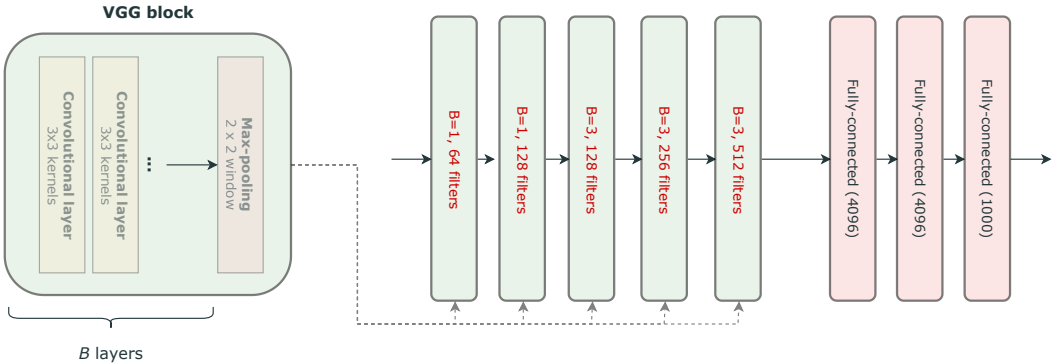


Figure 3: Original VGG-11. By varying the number and configuration of blocks, we go from VGG-11 to VGG-19.

Consider the **CIFAR-10**[2] dataset: 60000 32x32 colour images in 10 classes, with 6000 images per class.

To see what happens when varying the depth, we experiment with a VGG-like architecture, varying the number of blocks, and the number of convolutional layers inside each block. We use a global average pooling at the end followed by a linear layer, with cross-entropy loss. We run 3 epochs, 3 repetitions each, with the Adam optimization algorithm.

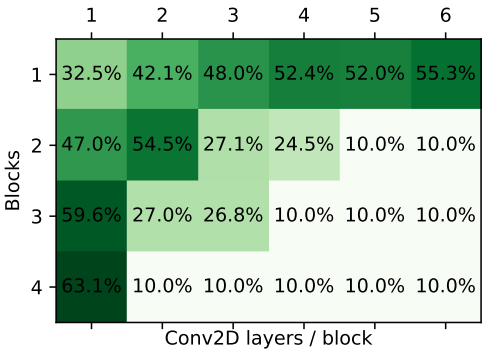


Figure 4: Test accuracy

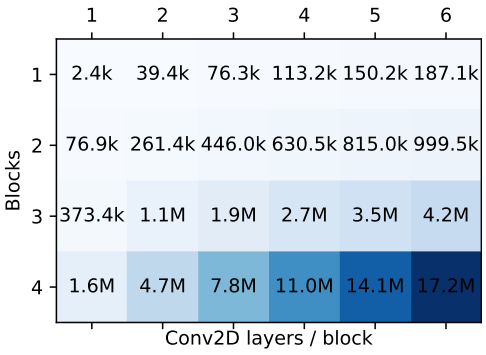


Figure 5: Parameters

- Some combinations do not even train (they get stuck at initialization). Other combinations appear much slower.
- Increasing the number of blocks looks good, but the original image is destroyed after a few max pooling operations.
- Increasing the number of layers in a block is also good, but the gains are more marginal.
- It is very easy to make the number of parameters go up.

This is harder than expected!

Subsequent advances in the period 2012-2017 came from other teams competing in the ILSVRC (ImageNet) challenge and led to an explosion of new families and methods, including:

- Parallel layers and later **Batch Normalization** in the **GoogLeNet** (Inception) family.
- **Residual connections** in the **ResNet** models.
- Depthwise convolutions for efficiency in the **MobileNet** family.
- Neural architecture search combined with simple scaling strategies, e.g., **EfficientNets** and **NASNets**.

When moving towards a *very deep* regime, many strange phenomena appear, including (but not limited to):

- **Scaling laws**: performance scales linearly in a power law of data and compute, in a Moore-like fashion.
- **Multiple descents**: periods of overfitting may lead (after a while) to periods of better generalization.
- **Emergent properties**: scaling sufficiently may lead to a phase transition quickly moving performance on certain tasks from 0 to state-of-the-art (depending on the metric).

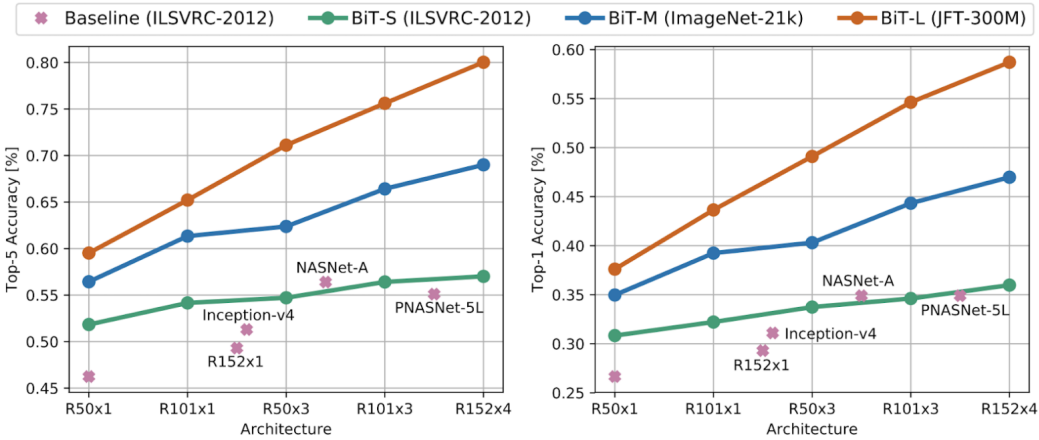


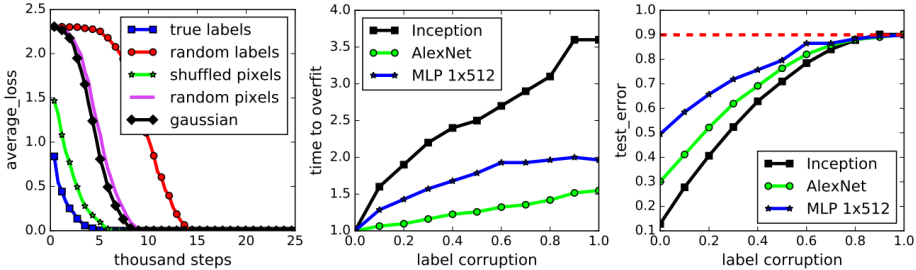
Figure 6: *Open-Sourcing BiT: Exploring Large-Scale Pre-training for Computer Vision* (Google AI Blog). **Note**: this uses vision transformers and pretraining, which we will look into soon.

Overfitting happens when the performance of a model on the training set is improving, while the performance on a separate validation set is worsening.

Deep learning models are strangely resilient to classical overfitting, but they have shown some peculiar characteristics, e.g.:

1. Models trained for long enough on random data can still memorize the entire dataset and achieve perfect accuracy.
2. Models can start improving after a period of apparent overfitting (**double descent**).

Visualization of overfitting in deep networks



(a) learning curves (b) convergence slowdown (c) generalization error growth

Figure 1: Fitting random labels and random pixels on CIFAR10. (a) shows the training loss of various experiment settings decaying with the training steps. (b) shows the relative convergence time with different label corruption ratio. (c) shows the test error (also the generalization error since training error is 0) under different label corruptions.

Figure 7: Taken from (Zhang et al., 2016). A large CNN can fit data perfectly even with random labels and/or random pixels.

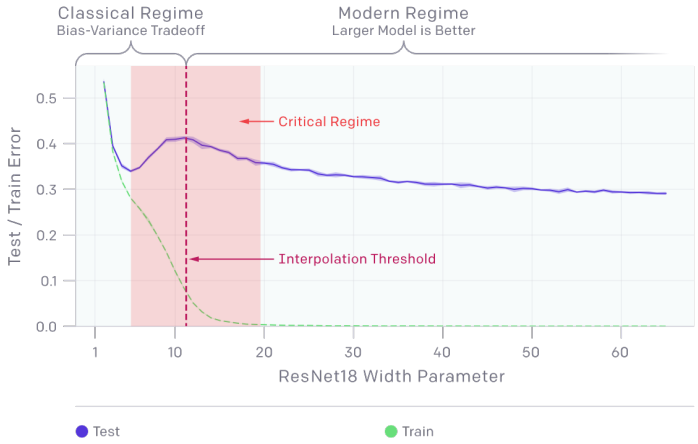


Figure 8: Deep Double Descent (OpenAI Blog).

- The performance of a neural network depends heavily on the amount/quality of data, its architecture, hyper-parameters, initialization, optimization, etc. *This requires a mix of good recipes, manual/automatic search, rules-of-thumbs, and experience.*
- The tools we have available up to now are not enough for truly deep networks. In this lecture, we cover a number of additional tricks and layers designed especially to simplify and improve the training of the models.

Data strategies

Data augmentation

Data augmentation is a technique to *virtually* increase the size of the dataset at training time:

1. Sample of mini-batch of examples;
2. For each example, apply one or more transformations randomly sampled (e.g., flipping, cropping, ...).
3. Train on the transformed mini-batch.

Data augmentation can be extremely helpful for overfitting, making the network more robust to small changes in the input data.

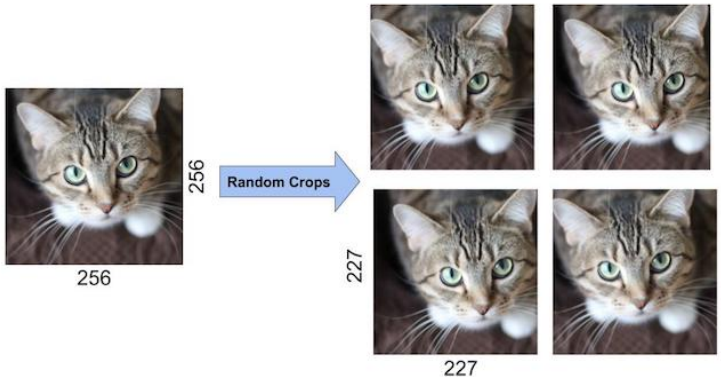


Figure 9: Devising data augmentation strategies is especially easy with images, e.g., cropping, shearing, shifting (image source).

Devising efficient forms of data augmentation is a popular research field.

For example, **MixUp**[3] combines two examples (x_1, y_1) and (x_2, y_2) by taking convex combinations with a random λ:

$$x = \lambda x_1 + (1 - \lambda) x_2 \,, \tag{1}$$

$$y = \lambda y_1 + (1 - \lambda) y_2 \,. \tag{2}$$

[3] Zhang, H., Cisse, M., Dauphin, Y.N. and Lopez-Paz, D., 2017. **mixup: Beyond empirical risk minimization**. *arXiv preprint arXiv:1710.09412*.

For images, MixUp can create samples that look highly unnatural. In **CutMix**[4], we first sample a random patch from the second image x_2. Define a mask $\mathbf{M}$ where $M_{ij} = 1$ if the pixel belongs to the patch or not, we overimpose the random patch on the first image:

$$x = (1 - \mathbf{M}) \odot x_1 + \mathbf{M} \odot x_2 \,, \tag{3}$$

$$y = \lambda y_1 + (1 - \lambda) y_2 \,. \tag{4}$$

where λ is again sampled from the uniform distribution in $[0, 1]$.

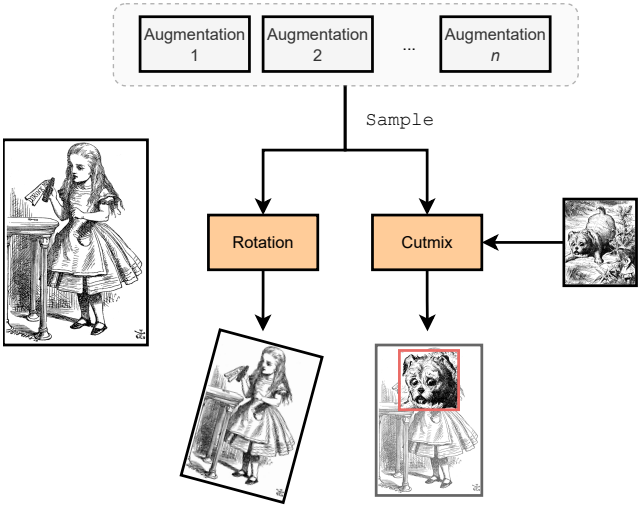
[4] Yun, S., et al., 2019. **Cutmix: Regularization strategy to train strong classifiers with localizable features**. In IEEE/CVF ICCV (pp. 6023-6032).

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

def randaugment(N, M):
"""Generate a set of distortions.

   Args:
    N: Number of augmentation transformations to
        apply sequentially.
    M: Magnitude for all the transformations.
"""

  sampled_ops = np.random.choice(transforms, N)
  return [(op, M) for op in sampled_ops]
```

Figure 2. Python code for RandAugment based on numpy.

Figure 10: RandAugment is a popular way of defining data augmentation, by selecting randomly *N* data augmentation steps, all of a predefined magnitude *M*.

Sampling data augmentation strategies



In code, data augmentation can be included as part of the data processing, or as part of the model itself:

```
# Data augmentation in a data pipeline
train_data = train_data.map(partial im:
    tf.image.random_brightness(im, 0.1))

# Add data augmentation before a model
model = Sequential(
  tf.keras.layers.RandomBrightness(0.1),
  CNN()
)
```

A layer like `RandomBrightness` has different behaviours depending on whether we are training (**training mode**) or not (**inference mode**). They can be triggered with the `training` parameter when calling it:

```
augm = tf.keras.layers.RandomBrightness(0.1)

# Add image augmentation
augm(im, training=True)

# Do nothing
augm(im, training=False)
```

Using the `predict` and `fit` functions of `tf.keras` automatically selects the correct version.

Optimization strategies

Early stopping

If we want to minimize $L(\mathbf{w})$, a valid stopping criterion could be:

$$\|L(\mathbf{w}_t) - L(\mathbf{w}_{t-1})\|^2 \leq \varepsilon$$

for some threshold ε. This is rarely done in practice: for large networks, compute time is the bottleneck, while for smaller models (or small datasets) overfitting is the major concern.

Early stopping is a procedure to find the optimal number of iterations to avoid overfitting:

1. Keep a portion of the dataset as the **validation set**.
2. For each epoch, check the validation loss (or accuracy, or some other metric of interest).
3. Whenever validation loss is not improving for a while (a certain number of epochs called **patience**), stop the optimization process.

Early stopping is extremely common in training with small datasets; it highlights the difference between pure optimization and learning.

Optimization strategies

Regularization

A warning sign of overfitting can be **large weights**: these networks tend to be less *smooth* and make sharper changes in their outputs.

Regularization forces the optimization to select a network with smaller weights by penalizing large norms:

$$\mathbf{w}^* = \arg\min \left\{ \sum_i l(f(x_i), y_i) + \lambda \cdot \|\mathbf{w}\|^2 \right\}, \tag{5}$$

λ is a hyper-parameter: with $\lambda = 0$ we have no regularization; with a λ too large, all weights would go to 0.

Consider the gradient update of a regularized loss:

$$-\text{Gradient of loss} = -\nabla\Big[\sum_i l(f(x_i), y_i)\Big] - 2\lambda \mathbf{w}\,. \tag{6}$$

In the absence of the first term, the weights would *decay* exponentially to zero. In pure SGD, this form of regularization is also called **weight decay**.

In other optimization algorithms, weight decay and regularization are different strategies and must be implemented differently.

Loshchilov, I. and Hutter, F., 2017. **Fixing weight decay regularization in Adam**. *arXiv preprint arXiv:1711.05101*.

Regularization allows us to steer the optimization problem towards favourable solutions.

Many other types of regularization exists! For example, replacing the Euclidean norm of the weights with the sum of absolute values:

$$\mathbf{w}^* = \arg\min \left\{ \sum_i l(f(x_i), y_i) + C \cdot \sum_j |w_j| \right\}, \tag{7}$$

can lead to sparser solutions.[5]

[5] Scardapane, S., Comminiello, D., Hussain, A. and Uncini, A., 2017. **Group sparse regularization for deep neural networks**. *Neurocomputing*, 241, pp.81-89.

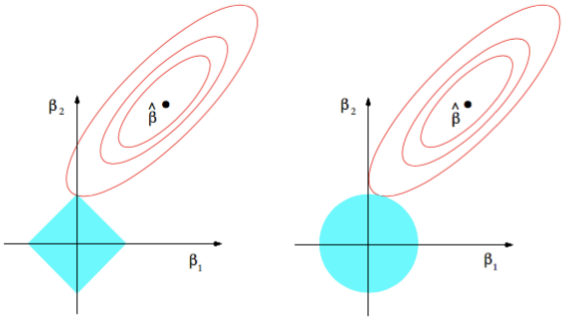


FIGURE 3.11. *Estimation picture for the lasso (left) and ridge regression (right). Shown are contours of the error and constraint functions. The solid blue areas are the constraint regions* $|\beta_1| + |\beta_2| \leq t$ *and* $\beta_1^2 + \beta_2^2 \leq t^2$*, respectively, while the red ellipses are the contours of the least squares error function.*

Figure 11: Graphical depiction of why the absolute value promotes sparsity in the linear case (reproduced from *The Elements of Statistical Learning*).

The ℓ_1 norm $\|\mathbf{w}\|_1 = \sum_i |w_i|$ is difficult to optimize with SGD, because it is not differentiable at 0. Interestingly, we can *reparameterize* $\mathbf{w}$ with two new vectors $\mathbf{a}$ and $\mathbf{b}$, $\mathbf{w} = \mathbf{a} \odot \mathbf{b}$, and rewrite:

$$f(\mathbf{w}) + C\|\mathbf{w}\|_1 \rightarrow f(\mathbf{a} \odot \mathbf{b}) + C\left[\|\mathbf{a}\|^2 + \|\mathbf{b}\|^2\right] . \tag{8}$$

This has a similar loss landscape but it is much easier to optimize. In general, how we **parameterize** the same vector can have a significant impact on training and optimization.

Ziyin, L. and Wang, Z., 2022. **Sparsity by Redundancy: Solving L_1 with a Simple Reparametrization**. *arXiv preprint arXiv:2210.01212*.

New layers

Dropout regularization

Why is data augmentation helpful?

The core idea is that we can make the network more robust by adding slight **perturbations** to the input. We can prove this to be a form of regularization.[6]

Dropout extends this idea to the network itself: instead of perturbing the images, we perturb the hidden layers by randomly *dropping* (removing) some of the connections.

[6] Bishop, C.M., 1995. **Training with noise is equivalent to Tikhonov regularization**. *Neural Computation*, 7(1), pp.108-116.

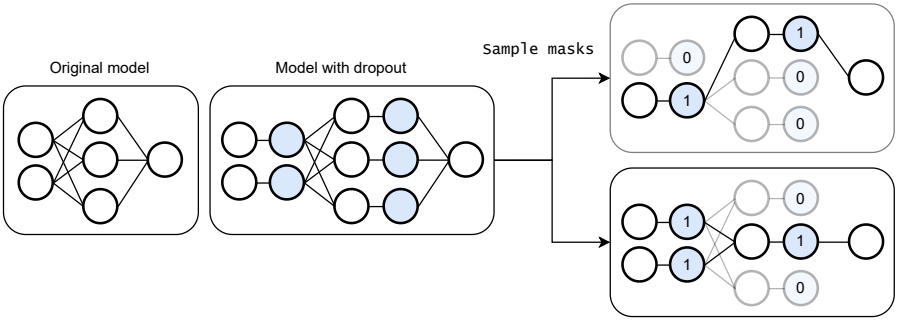


Figure 12: With dropout, the network can be seen as being drawn from a (very large) collection of sub-networks. Dropout can also be applied to the input of the network.

Define $\underset{(b,f)}{\mathbf{H}}$ as the output of a generic fully-connected layer having f units, being fed with b inputs (mini-batch size).

With dropout, during training we replace it with:

$$\widetilde{\mathbf{H}} = \mathbf{H} \odot \mathbf{M}\,, \tag{9}$$

where $\mathbf{M}$ is a binary matrix with entries drawn from a Bernoulli distribution with probability p (i.e., $M_{i,j}$ is 0 with probability p, 1 with probability $(1-p)$).

If $M_{i,j} = 0$, the value $H_{i,j}$ is replaced with 0.

Srivastava, N., et al., 2014. **Dropout: a simple way to prevent neural networks from overfitting**. *The Journal of Machine Learning Research*, 15(1), pp. 1929-1958.

Consider a NN $f(\mathbf{x}; \mathbf{M})$ with a single layer of dropout. The output is a random variable w.r.t. the distribution of the mask $\mathbf{M}$. At inference time, a sensible approach would be to take the expected value:

$$\hat{y} = \mathbb{E}_{p(\mathbf{M})}\left[f(\mathbf{x}; \mathbf{M})\right] \approx \frac{1}{N}\sum_{i=1}^{N} f(\mathbf{x}; \mathbf{M}_i)\,, \tag{10}$$

where $\mathbf{M}_i \sim p(\mathbf{M})$ are draws of the mask, and the second step is a **Montecarlo** approximation. We call this **Montecarlo dropout**, and we can use the different samples to gather an uncertainty estimate on the output.

Gal, Y. and Ghahramani, Z., 2016. **Dropout as a Bayesian approximation: Representing model uncertainty in deep learning**. In ICML (pp. 1050-1059). PMLR.

A simpler and more common choice is to take the expected value of the dropout *layer* (as opposed to the entire network), since we can compute it in closed form:

$$\mathbb{E}[\widetilde{\mathsf{H}}] = p\mathbf{0} + (1-p)\mathsf{H} = (1-p)\mathsf{H}\,. \tag{11}$$

To simplify the inference, a common variant is the so-called **inverted dropout**:

$$\widetilde{\mathsf{H}} = (\mathsf{H} \odot \mathsf{M})/(1-p)\,.$$

This is useful because $\mathbb{E}[\widetilde{\mathsf{H}}] = p\mathbf{0} + (1-p)\frac{\mathsf{H}}{1-p} = \mathsf{H}$, i.e., we can simply remove the layer. Some books consider this *the* dropout implementation.

- Dropout applied to a convolutive layer would drop *single channels of single pixels*. **Spatial dropout** drops entire channels at a time. **Cutout** drops patches of the tensor (for each channel).
- In general, dropout (even its variants) is less common for convolutive layers, but more common for other types of networks (transformers, recurrent networks).
- **DropConnect** drops single weights instead of entire neurons:

$$\widetilde{\mathsf{H}} = \phi\left((\mathsf{W} \odot \mathsf{M})\mathsf{x}\right) .$$

Dropout can also be switched on or off during training. **Early dropout** can be useful to counteract gradient variance in SGD during the initial stages of training; **late dropout**, instead, can be useful against overfitting.

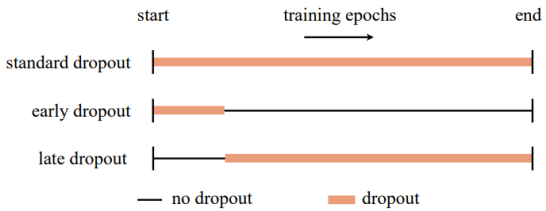


Figure 13: Reproduced from *Dropout Reduces Underfitting* (Liu et al., 2023).

New layers

Batch normalization

Given a dataset $\underset{(n,d)}{\mathsf{X}}$, it is common in machine learning to preprocess it so that each column has mean zero and unitary variance (z-scaling or standard scaling):

$$\mathsf{X}' = \frac{\mathsf{X} - \boldsymbol{\mu}}{\sqrt{\boldsymbol{\sigma}^2}}, \tag{12}$$

where $\boldsymbol{\mu}, \boldsymbol{\sigma}^2 \in \mathbb{R}^d$, $\mu_j = \frac{1}{n}\sum_i \mathsf{X}_i$ and $\sigma_j^2 = \frac{1}{n}\sum_i (\mathsf{X}_{ij} - \mu_j)^2$ are the empirical mean and variance. For example, we can do this in scikit-learn with `StandardScaler`. For many optimization algorithms, this can accelerate training significantly (**preconditioning**), because it makes the Hessian of X closer to the identity.

Batch normalization (BN), introduced in 2015, extends this idea by normalizing the outputs of each layer/block in a network,[7] and then learning an optimal mean and variance for each unit.

This is not trivial, because the mean and variance of the layer's output will vary during the optimization, and recomputing them on the entire dataset can be computationally expensive. BN works by approximating the estimates using the data in the mini-batch.

[7] Ioffe, S. and Szegedy, C., 2015. **Batch normalization: Accelerating deep network training by reducing internal covariate shift**. *Proc. ICML*.

Consider now a generic output $\underset{(b,f)}{\mathsf{H}}$ of a fully-connected layer (with batching).

First, compute the empirical mean and variance column-wise:

$$\tilde{\mu}_j = \frac{1}{b}\sum_i [\mathsf{H}]_{i,j}\,, \quad \tilde{\sigma}_j^2 = \frac{1}{b}\sum_i ([\mathsf{H}]_{i,j} - \tilde{\mu}_j)^2\,.$$

Second, we standardize the output so that each column has mean 0 and standard deviation 1:

$$\mathsf{H}' = \frac{\mathsf{H} - \tilde{\boldsymbol{\mu}}}{\sqrt{\tilde{\boldsymbol{\sigma}}^2 + \varepsilon}}\,,$$

where $\varepsilon > 0$ is a small coefficient to avoid division by 0.

The final output of the batch normalization layer $\text{BN}(\mathbf{H})$ sets a new mean and variance for each column:

$$[\text{BN}(\mathbf{H})]_{i,j} = \alpha_j H'_{i,j} + \beta_j \,. \tag{13}$$

The $2f$ values α_j, β_j are trained via gradient descent.

Commonly, BN is inserted between a fully-connected layer and the activation function:

$$\mathbf{Z} = \phi\left(\text{BN}(\mathbf{X}\mathbf{W} + \mathbf{b})\right) \,. \tag{14}$$

Similarly to dropout, BN requires a different behaviour outside of training, since it is undesirable that the output for an input depends on the mini-batch it is associated to.

Two common solutions are:

- After training, compute a mean and variance by running the trained model on the entire dataset, fixing the values μ_j, σ_j^2 to that value.
- Keep a moving average of all the estimated means and variances when training, using the final value during inference.

Wu, Y. and Johnson, J., 2021. **Rethinking "Batch" in BatchNorm**. *arXiv preprint arXiv:2105.07576*.

BN is very common in convolutive layers. Consider the output $H_{(b,h,w,c)}$ of a generic 2D convolutive layer (b, as before, is the size of the mini-batch).

BN works exactly as before, but the mean and the variance are computed *for each channel*:

$$\tilde{\mu}_z = \frac{1}{bhw} \sum_{i,j,k} [\mathbf{H}]_{i,j,k,z} , \quad \tilde{\sigma}_z^2 = \frac{1}{bhw} \sum_{i,j,k} ([\mathbf{H}]_{i,j,k,z} - \tilde{\mu}_z)^2 . \tag{15}$$

Despite its simplicity, batch normalization is extremely effective when training deep NNs.

Originally, its efficiency was believed to be consequence of a so-called *internal covariate shift* (i.e., distributions of activations changing layer-by-layer).

Nowadays, it is believed that BN works by making the optimization landscape *smoother* and, consequently, the gradients more predictive.

Santurkar, S. et al., 2018. **How does batch normalization help optimization?**. In NeurIPS (pp. 2483-2493).
Lipton, Z.C. and Steinhardt, J., 2018. **Troubling trends in machine learning scholarship**. *arXiv preprint arXiv:1807.03341*.

Batch normalization has multiple issues in practice:

- It introduces dependencies across the elements of the mini-batch, making it less suitable in, e.g, distributed optimization or contrastive self-supervised learning.
- The variance of the estimate can be very high when the batch size is small, which is a problem with very large models.
- There is a mismatch between its training and inference behaviours.

This has led to many variants and a significant research on developing **normalizer-free** models: `https://iclr-blog-track.github.io/2022/03/25/unnormalized-resnets/`.

It is very easy to define multiple variants of BN by varying the axes along which statistics are computed and controlled.

For example, a popular variant is **layer normalization**, where we mean and variances are computed for each row (each input) independently:

$$\tilde{\mu}_i = \frac{1}{f}\sum_j [\mathsf{H}]_{i,j}\,, \quad \tilde{\sigma}_i^2 = \frac{1}{f}\sum_j ([\mathsf{H}]_{i,j} - \tilde{\mu}_i)^2\,.$$

This works also with small batch sizes (even 1) and it does not add any inter-batch dependency. Importantly: in LN, α and β have the same shape as axis along which we normalize.

Most implementations accept an `axis` parameter:

```
ln = tf.keras.layers.LayerNormalization(axis=[1, 2, 3])
```

For BN, `axis` is the axis that is being normalized, but the statistics are computed across *the other* axes. For LN (and variants), `axis` selects the axes across which statistics are computed. In both cases, α and β have the same dimensionality as `axis`.

So, `ln` above will have $2hwc$ parameters!

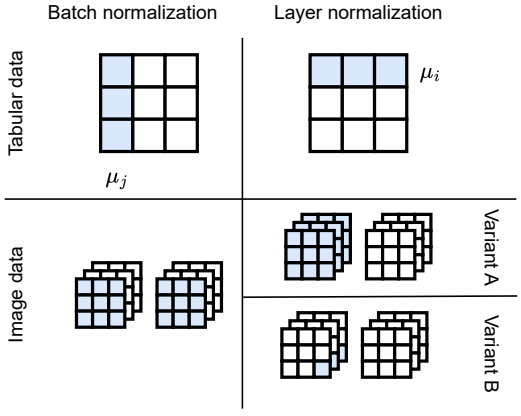


Figure 14: Variant of batch and layer normalization for tabular and image-like data.

New layers

Residual connections

Consider a neural network $h(x)$ which is performing relatively well. If we train a deeper network $f(x) = g(h(x))$, we would still expect a good accuracy, since at the very least we should have $f(x) \approx h(x)$ with $g(x) \approx x$.

However, this was not matched by practice:[8]

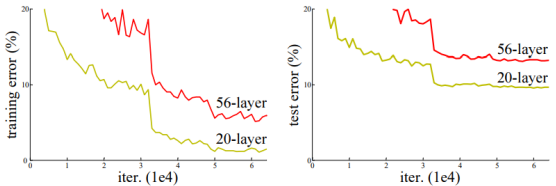


Figure 1. Training error (left) and test error (right) on CIFAR-10 with 20-layer and 56-layer "plain" networks. The deeper network has higher training error, and thus test error. Similar phenomena on ImageNet is presented in Fig. 4.

[8] He, K., Zhang, X., Ren, S. and Sun, J., 2016. **Deep residual learning for image recognition**. In IEEE CVPR (pp. 770-778).

In a **residual network**, we modify each block $f(x)$ (e.g., a VGG block) by adding a **skip connection**:

$$r(x) = f(x) + x\,, \tag{16}$$

If x and $f(x)$ have different dimensionality, we can rescale x with a matrix multiplication or a 1×1 convolutive block. $r(x)$ is called a **residual block**.

Residual blocks work very well with batch normalization on $f(x)$ (the **residual path**), because it tends to bias the network towards the skip path at initialization.[9]

[9] De, S. and Smith, S., 2020. **Batch normalization biases residual blocks towards the identity function in deep networks**. *NeurIPS*.

Note that:

$$\partial r(x) = \partial f(x) + \mathbf{I}\,.$$

When using a residual connection, during backpropagation the gradient always flows unhindered trough the residual path, reducing any vanishing or exploding effects:

$$v^\top [\partial r(x)] = \underbrace{v^\top [\partial f(x)]}_{\text{Original VJP}} + v\,.$$

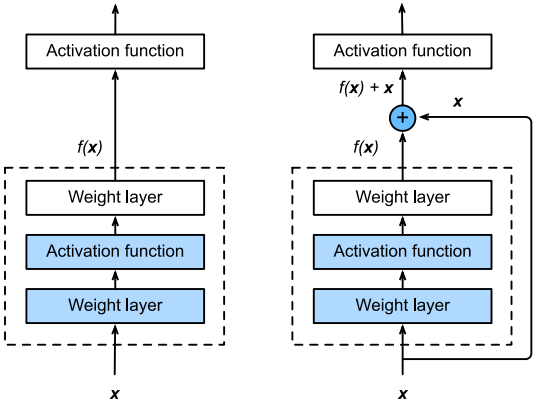


Figure 15: Source: Dive into Deep Learning, Chapter 7.6. We typically put the ReLU outside $f(x)$, otherwise each residual block would be positive and it would be impossible to have a negative change.

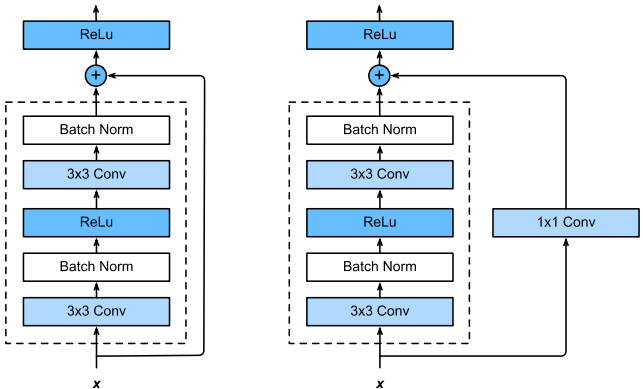


Figure 16: Residual block with rescaling of the skip connection. Source: Dive into Deep Learning, Chapter 7.6.

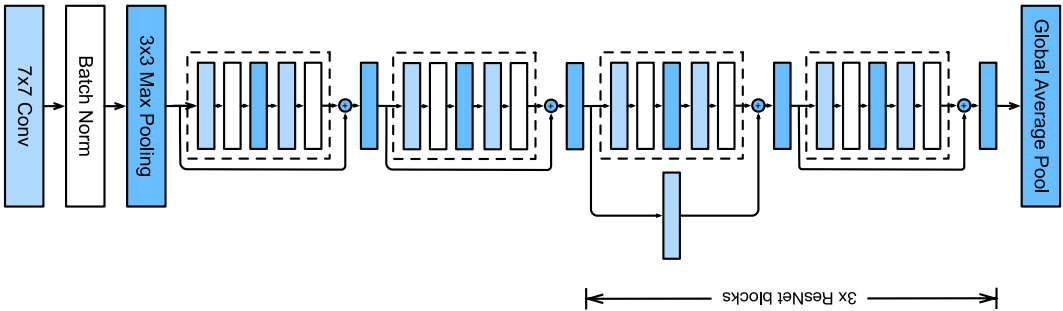


Figure 17: Concatenating *many* residual blocks, we obtain a residual network (ResNet). Source: Dive into Deep Learning, Chapter 7.6.

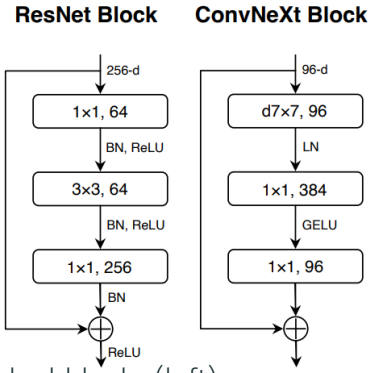


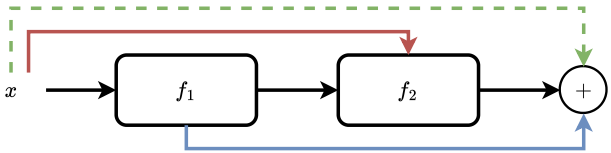
Figure 18: Bottleneck residual blocks (left) are very popular, as shown by their use in novel architectures (e.g., ResNeXt, on the right).

Liu, Z., et al., 2022. **A convnet for the 2020s**. In *IEEE/CVF CVPR* (pp. 11976-11986).

Note that a name like *ResNet* may refer to different things: a family of models, a single element of the family (e.g., ResNet-50), or a *pre-trained* version of the model itself.

Many repositories (TensorFlow Hub, PyTorch Hub, pytorch-image-models, HuggingFace Model) have appeared recently to categorize and provide easy, open access to them.

Two stacked residual blocks can be interpreted as creating four *paths* through the network, shown here with different colors:



The number of paths grow exponentially in the number of residual blocks, which can increase the robustness of the network and make it behave like an ensemble of shallow models (this interpretation will be very important in transformers).

Veit, A., Wilber, M.J. and Belongie, S., 2016. **Residual networks behave like ensembles of relatively shallow networks**. *NeurIPS*.

Consider a network with T residual blocks, the output can be written as:

$$h_t = f(h_{t-1}) + h_{t-1}\,, t = 1, \ldots, T\,, \tag{17}$$

with $h_0 = x$. In the limit $T \to \infty$ (infinite layers), each layer will only define an infinitesimal displacement. Although we cannot have infinite layers, we can handle this by conditioning a single residual block on t:

$$\frac{\partial h_t}{\partial t} = f(x, t)\,. \tag{18}$$

The previous equation can be solved by an ordinary differential equation (ODE) solver, and this class of models are called **neural ODEs**.

- **Chapter 9** in the book.