Neural Networks for Data Science Applications

Master's Degree in Data Science

Lecture 5: Tips and Tricks

Lecturer: S. Scardapane

SAPIENZA
UNIVERSITÀ DI ROMA

Introduction

Scaling laws and overfitting

Now that we have a few basic components (convolutional layers, pooling, ...), we need some recipes to combine them together effectively.

The Oxford's **Visual Geometry Group** (VGG) in 2014 proposed an effective recipe, made by stacking *block*s composed as follows:

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*.

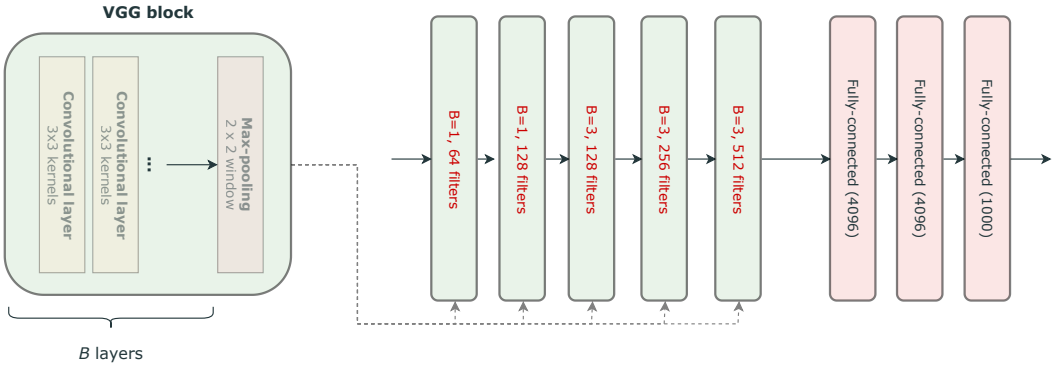


Figure 1: Original VGG-11. By varying the number and configuration of blocks, we go from VGG-11 to VGG-19.

Consider the **CIFAR-10**[1] 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.

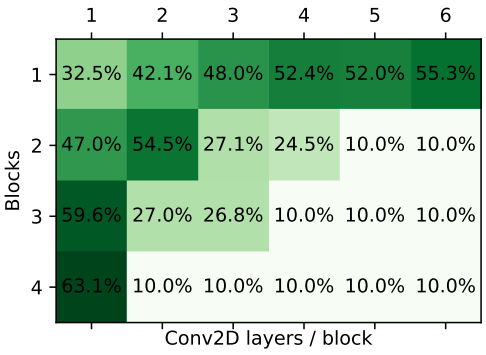


Figure 2: Test accuracy

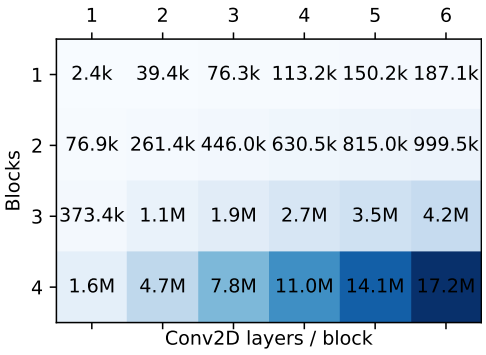


Figure 3: Parameters

- Some combinations do not even train (they get stuck at initialization). Other combinations appear much slower.
- Increasing blocks looks good, but the original image is destroyed after a few max pooling operations.
- Increasing 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!

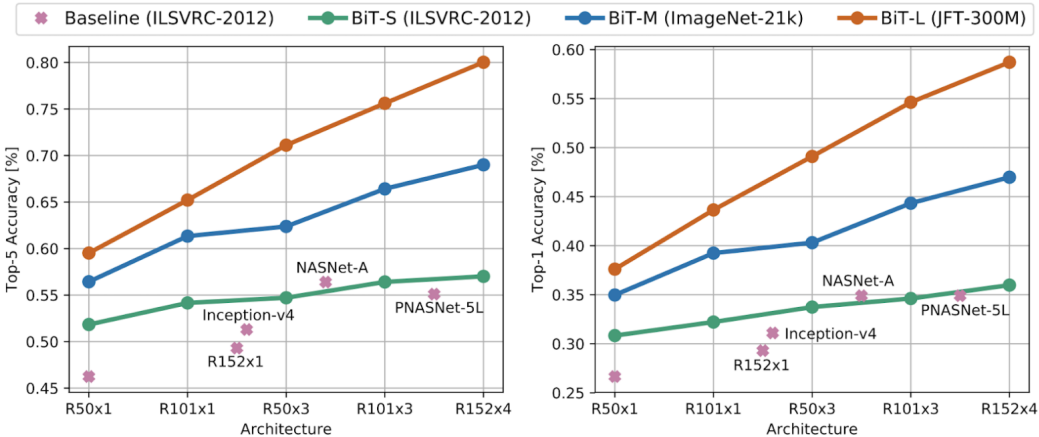
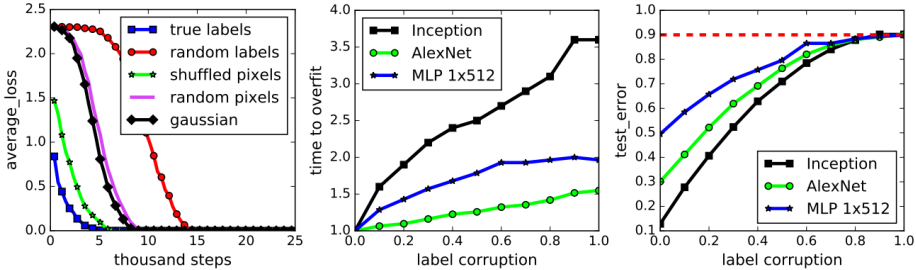


Figure 4: *Open-Sourcing BiT: Exploring Large-Scale Pre-training for Computer Vision* (Google AI Blog). **Note**: this uses vision transformers and pretraining, which we will look into soon.

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**).



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

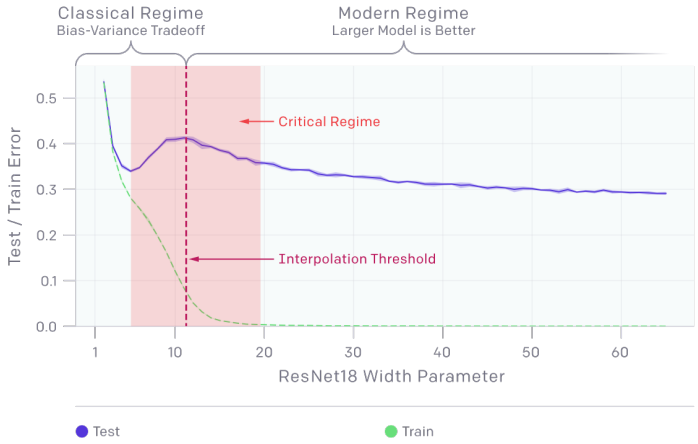


Figure 6: Deep Double Descent (OpenAI Blog).

- 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.

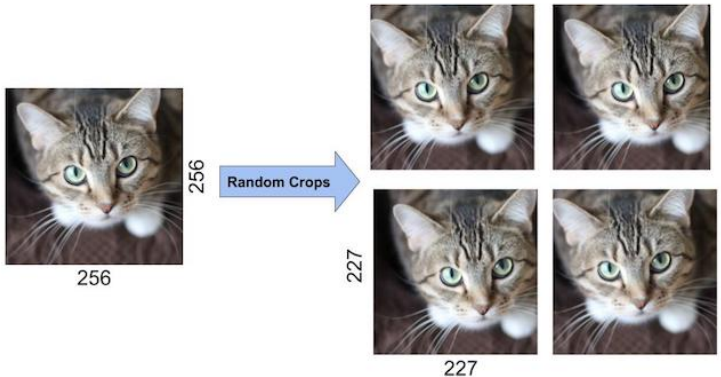


Figure 7: Devising data augmentation strategies is especially easy with images, e.g., cropping, shearing, shifting (image source).

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

For example, **mixup**[2] combines two examples (x_1, y_1) and (x_2, y_2) by taking convex combinations with a random λ:

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

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

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

Optimization strategies

Early stopping

Early stopping is a procedure to find the optimal number of iterations (supposing a single descent curve):

1. Keep a portion of the dataset as the **validation set**.
2. For each epoch, check the validation loss (or accuracy).
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 neural networks; 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:

$$\mathbf{w}^* = \arg\min \left\{ \sum_i l(f(x_i), y_i) + \lambda \cdot \|\mathbf{w}\|^2 \right\}, \tag{3}$$

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:

$$-\text{Gradient of loss} = -\nabla\left[\sum_i l(f(x_i), y_i)\right] - 2C\mathbf{w}\,. \tag{4}$$

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:

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

can lead to sparser solutions.[3]

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

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.[4]

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.

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

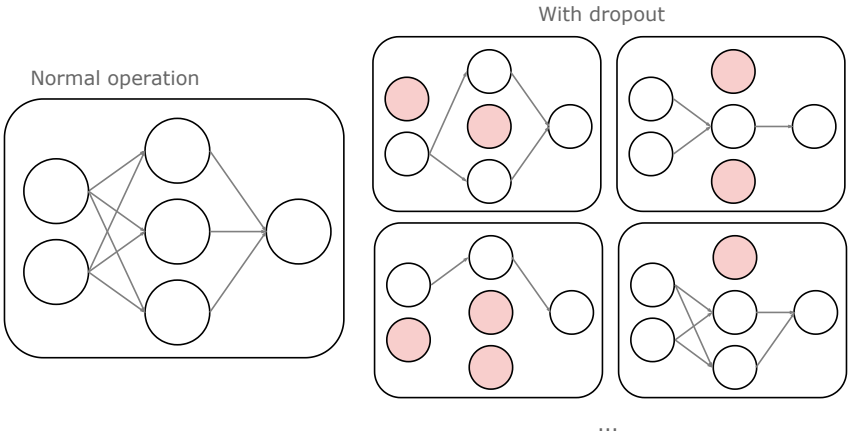


Figure 8: With dropout, the network can be seen as being drawn from a (very large) collection of sub-networks. Dropout can also be applied to the input of the network.

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:

$$\widetilde{\mathbf{H}} = \mathbf{H} \odot \mathbf{M}\,, \tag{6}$$

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.

Dropout is different from other layers we introduced up to now, because it is *removed* when not training (to avoid having a stochastic output).

To do this, when not training we replace the output of the layer with its *expected value* during training:

$$\mathbb{E}[\widetilde{\mathbf{H}}] = p\mathbf{0} + (1-p)\mathbf{H} = (1-p)\mathbf{H}\,. \tag{7}$$

Failing to do so can introduce an undesired **bias** in the network's behaviour.

Many layers in TF have separate training/test behaviours. To select the correct one, we can use a specific flag when calling it:

```
drop = layers.Dropout(0.5)
drop(x, training=True) # Use the training version
```

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

A common variant of dropout is **inverted dropout**:

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

This is useful because its inference behaviour requires to simply remove the layer:

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

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.
- **DropConnect** drops single weights instead of entire neurons:

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

New layers

Batch normalization

Batch normalization (BN), introduced in 2015, is a simple heuristic that allowed to train deep networks *significantly* better.[5]

The idea is to *learn* an optimal value for the means and variances of the units in a layer.

Along with dropout and **residual connections** (introduced next), BN was instrumental in consolidating deep learning as the state-of-the-art in many fields.

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

Consider again 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 $\mathrm{BN}(\mathbf{H})$ sets a new mean and variance for each column:

$$[\mathrm{BN}(\mathbf{H})]_{i,j} = \alpha_j H'_{i,j} + \beta_j \,. \tag{8}$$

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:

$$\mathbf{Z} = \phi\left(\mathrm{BN}(\mathbf{X}\mathbf{W} + \mathbf{b})\right) \,. \tag{9}$$

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*:

$$\tilde{\mu}_z = \frac{1}{bhw} \sum_{i,j,k} [\mathbf{H}]_{i,j,k,z} , \quad \tilde{\sigma}_z^2 = \frac{1}{bhw} \sum_{i,j,k} ([\mathbf{H}]_{i,j,k,z} - \tilde{\mu}_z)^2 . \tag{10}$$

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*.

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 today is **layer normalization**, where we compute a mean and variance for each input *across the mini-batch*:

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

For convolutive layers, this is also computed across spatial dimensions. LN can work also with very small batch sizes or even a batch size of 1.

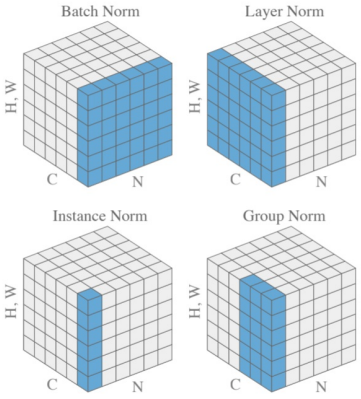


Figure 9: Qiao, S., Wang, H., Liu, C., Shen, W. and Yuille, A., 2019. **Micro-batch training with batch-channel normalization and weight standardization**. *arXiv preprint arXiv:1903.10520*.

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!

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:[6]

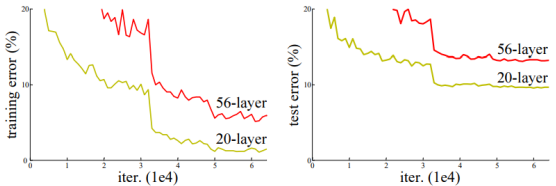


Figure 1. Training error (left) and test error (right) on CIFAR-10 with 20-layer and 56-layer "plain" networks. The deeper network has higher training error, and thus test error. Similar phenomena on ImageNet is presented in Fig. 4.

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

The idea is to allow for *skip* connections, in order to model only deviations from the identity function:

$$r(x) = f(x) + x\,, \tag{11}$$

where $f(x)$ is a standard network block (i.e., fully connected, VGG, ...). This simple idea allows to scale networks up to 100 and more layers.

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

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:

$$\partial\left[r(g(x))\right] = \partial\left[f(g(x))\right] + \partial g(x)\,.$$

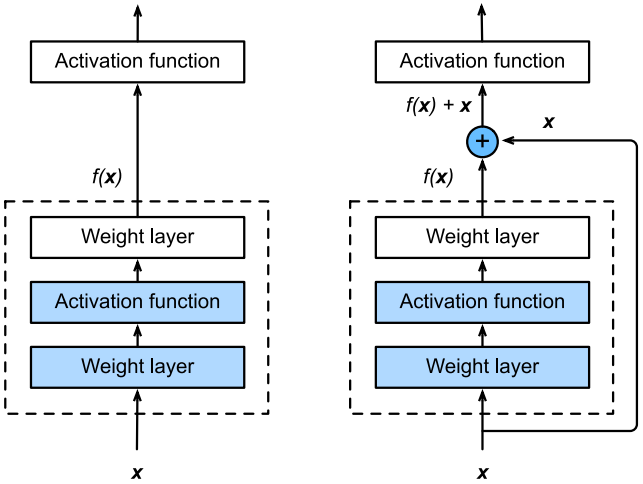


Figure 10: Source: Dive into Deep Learning, Chapter 7.6.

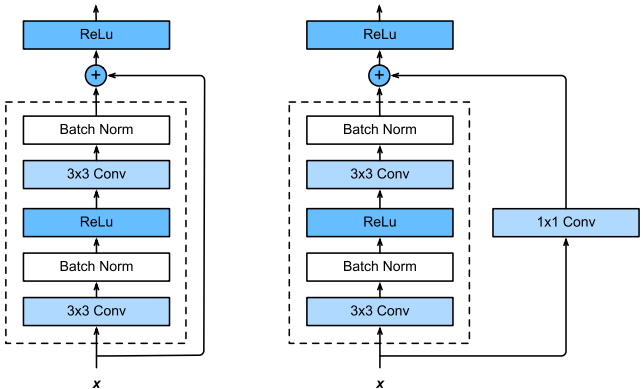


Figure 11: Residual block with rescaling of the skip connection. Source: Dive into Deep Learning, Chapter 7.6.

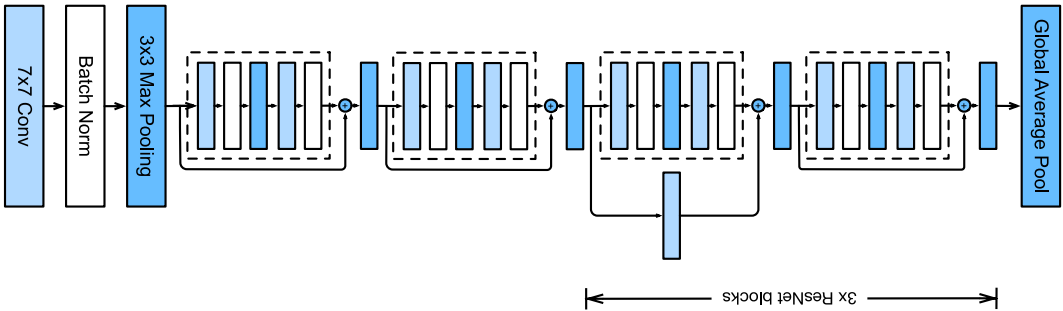


Figure 12: Concatenating *many* residual blocks, we obtain a residual network (ResNet). Source: Dive into Deep Learning, Chapter 7.6.

VGGNet and ResNet are only two of many families of neural networks, designed to provide pre-made models (and pretrained weights) to users.

See Chapter 7 in the book and the TensorFlow Hub for a much wider exposure to them. Some are built by automatically looking for the best architecture (e.g., neural architecture search), sometimes with a particular constraint (e.g., inference on embedded devices).

---

https://tfhub.dev/s?module-type=image-classification,image-classification-logits,image-classifier,image-feature-vector

- (**Mandatory**) Chapter 7 in the book.
- An example of a library devoted exclusively to image augmentations: `https://github.com/mdbloice/Augmentor`.
- Recipes for putting everything together: `https://karpathy.github.io/2019/04/25/recipe/`.
- An example of even more modern recipes: `https://arxiv.org/abs/2110.00476`.