Supplementary Slides

Md Tamjidul Hoque

Data Processing (2)

- Examples of data augmentation [1]:

Original

Channel Shuffle

Add (/ Multiply)

Gaussian Noise

Dropout

Dropout/channel

Coarse Dropout

Salt & Pepper Noise

Invert (per channel)

Jpeg compression

Blur

Multiply Hue

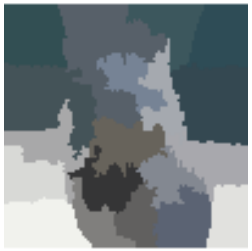
Multi. Saturation

Contrast

Histogram Equalization

Sharpen

Emboss



Superpixel (segmentation)

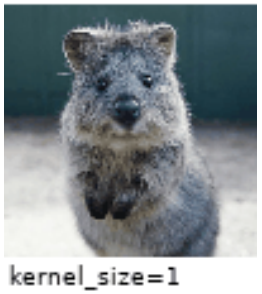
Fog

Snowflakes

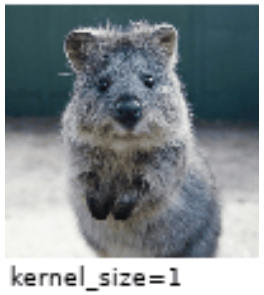
Data Processing (3)

- Examples of data augmentation:

Original

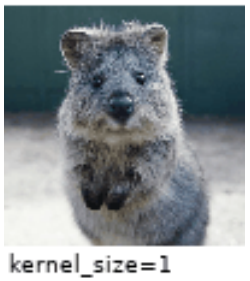


Average pooling

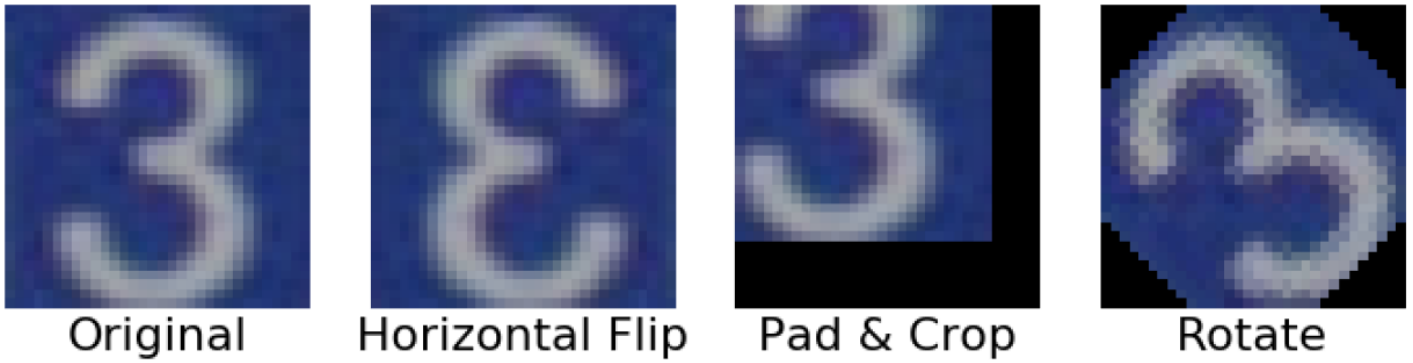


Max pooling

Min pooling



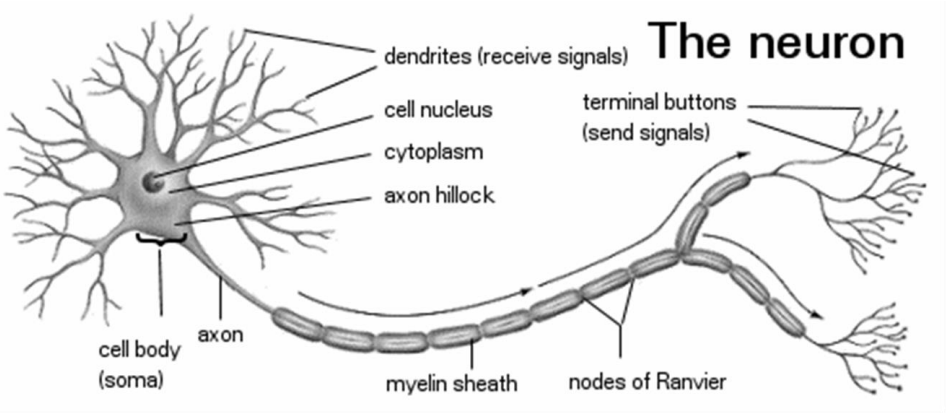
Median pooling



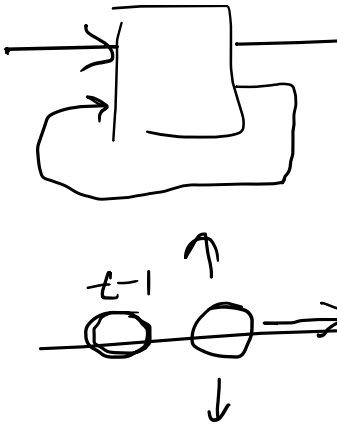
many more ….

We can even combine these strategies to create many more samples ...

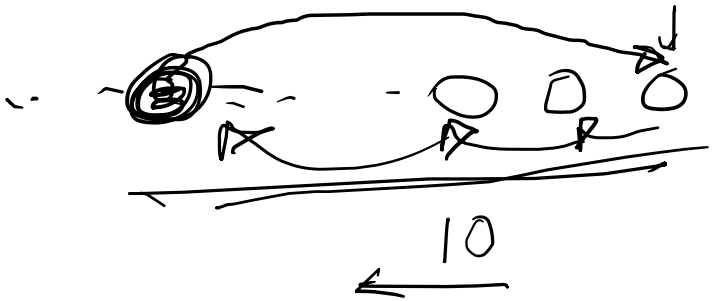
Modeling Neuron

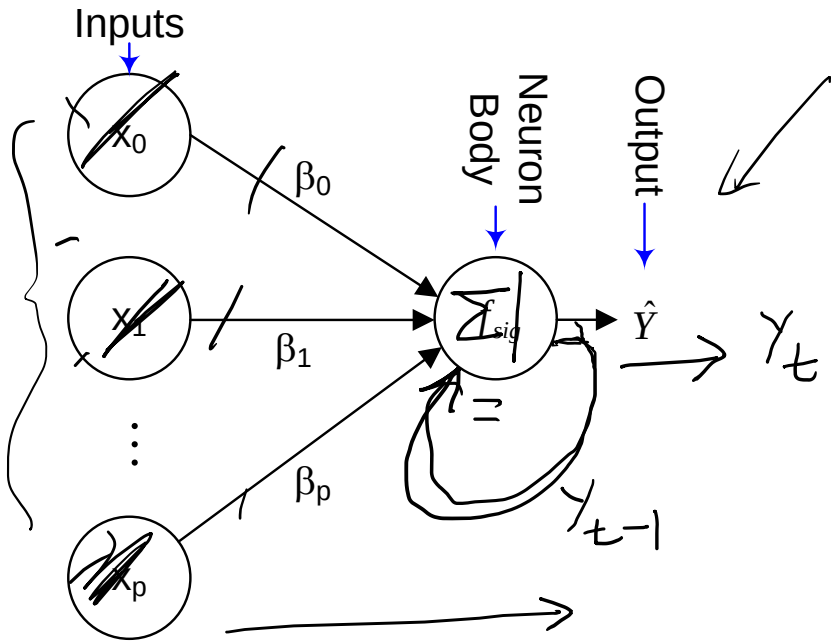


Neuron



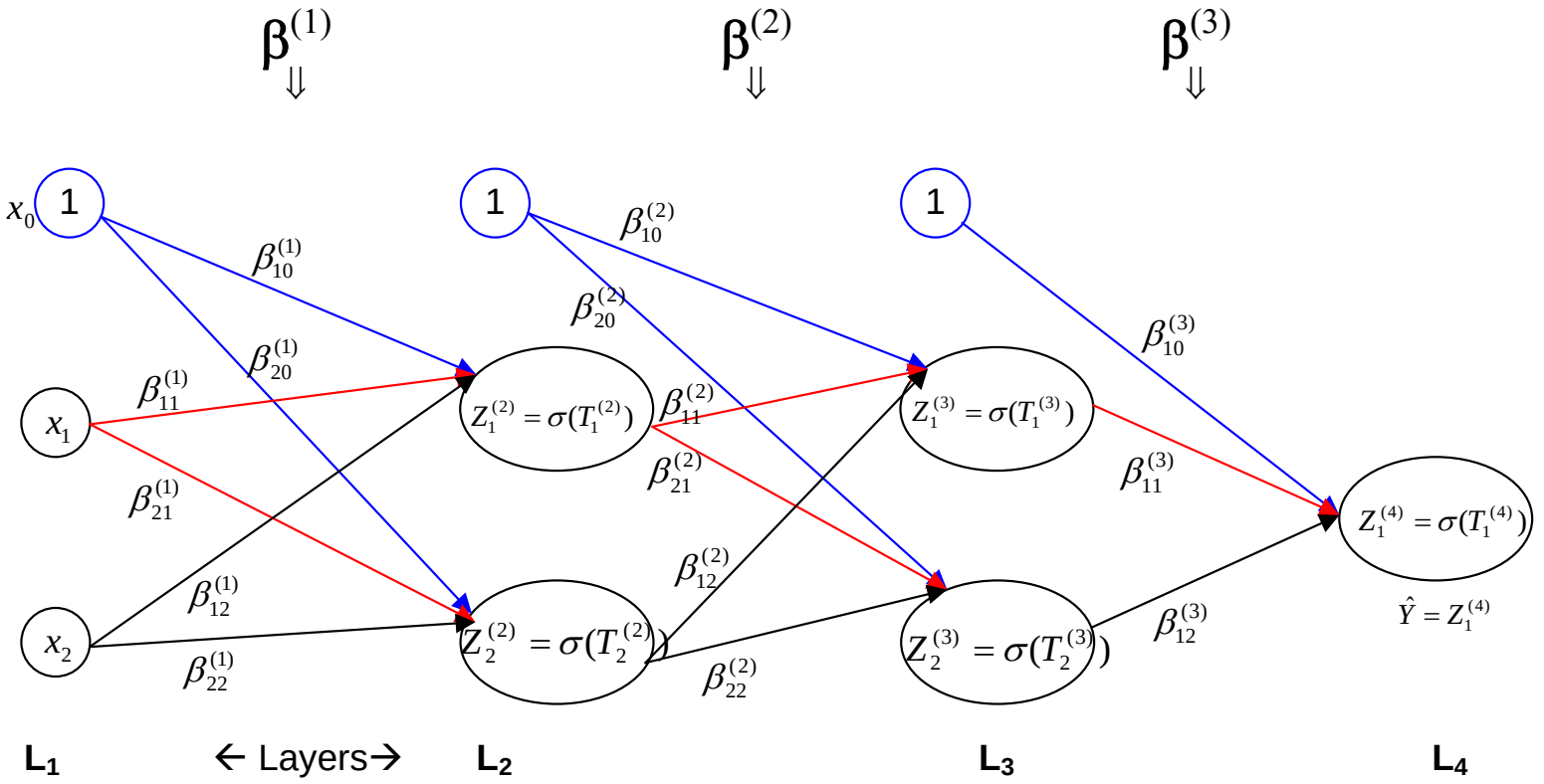
Modeling single neuron as logistic unit





$$\hat{Y} = f_{sig}(X_0\beta_0 + X_1\beta_1 + \cdots + X_p\beta_p) = f_{sig}(X^T\beta)$$

Notation for NN



Term (T): $T_1^{(2)} = X_0\beta_{10}^{(1)} + X_1\beta_{11}^{(1)} + X_2\beta_{12}^{(1)}$

$Z_1^{(2)} = \sigma(T_1^{(2)})$ σ = activation function,
Z = output of the activation function

$Z_1^{(2)} = f_{sig}(T_1^{(2)})$ When sigmoid is the activation fn..

$Z_1^{(3)} = \sigma(Z_0^{(2)}\beta_{10}^{(2)} + Z_1^{(2)}\beta_{11}^{(2)} + Z_2^{(2)}\beta_{12}^{(2)} + \ldots + Z_p^{(2)}\beta_{1p}^{(2)})$

… Classification MLPs

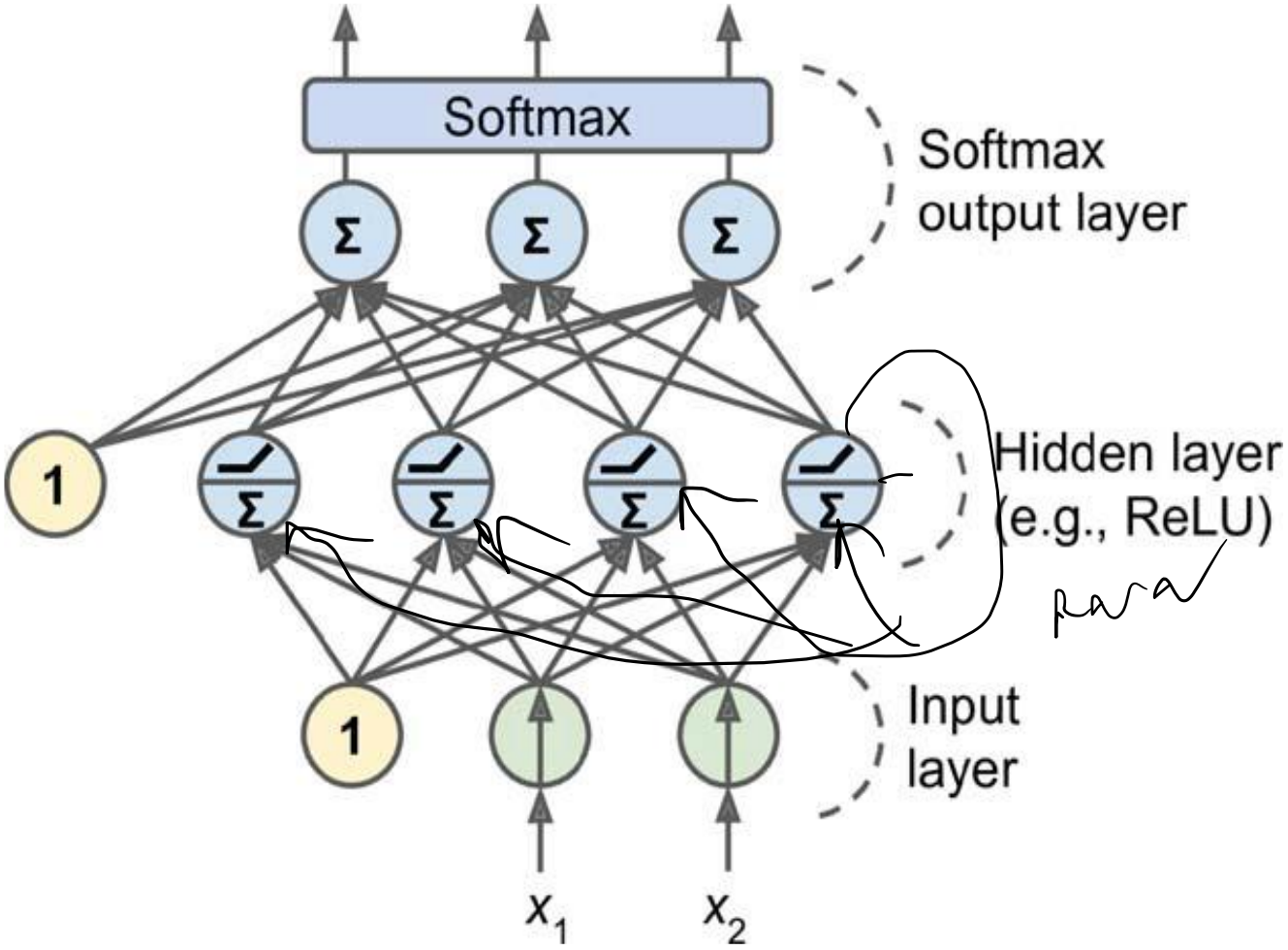
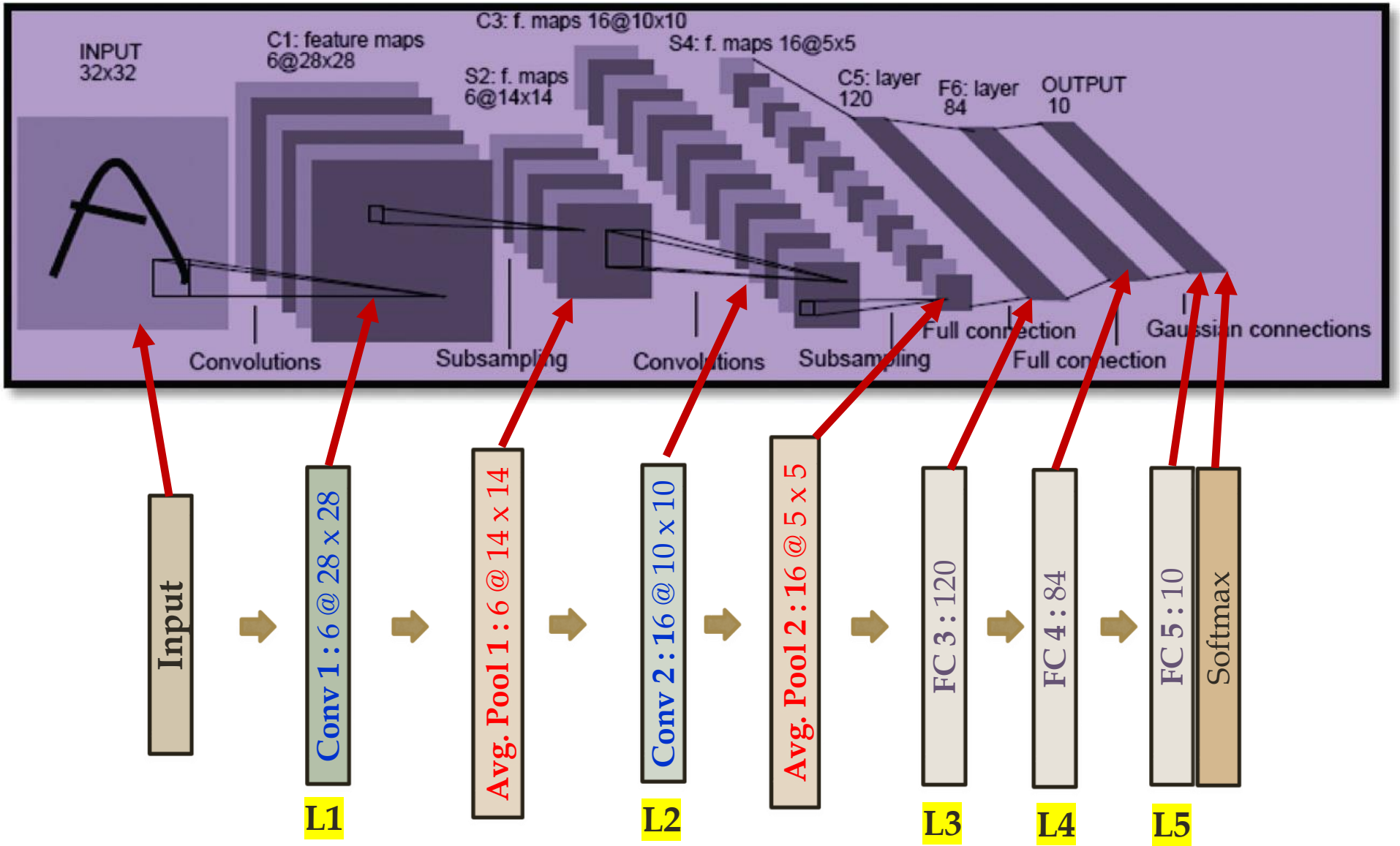


Figure 4: A modern MLP (including ReLU and softmax) for classification.

LeNet 5: A Classic CNN (4)

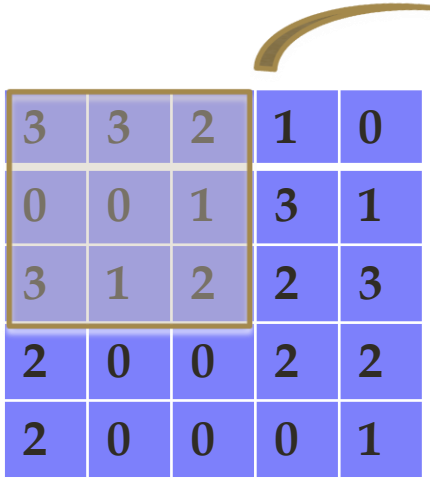
Renaming the layers w.r.t the current literature:



By layer, we will refer to the places where network parameters are involved.

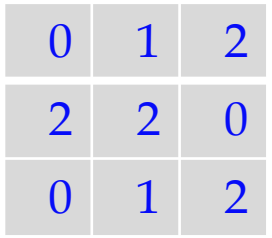
Convolutional Layer in CNN (1)

Example: Convolution operation (⊗) without padding and with stride of 1 [1].



Input 5 x 5 x 1

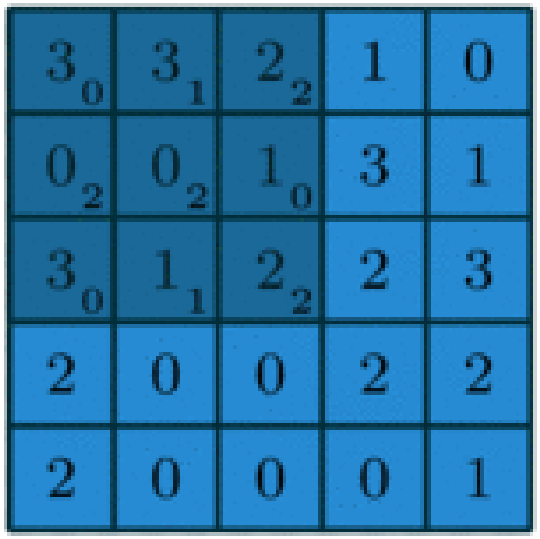
⊗



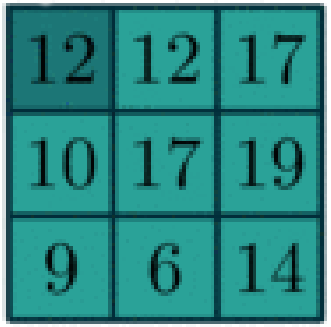
Filter 3 x 3 x 1

Computation of 1 output

= 3 x 0 + 3 x 1 + 2 x 2
+ 0 x 2 + 0 x 2 + 1 x 0
+ 3 x 0 + 1 x 1 + 2 x 2
= 7 + 0 + 5
= 12

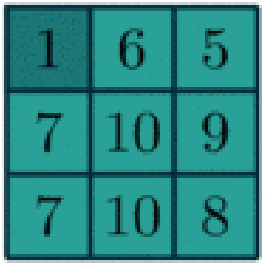


All Outputs:
3 x 3 x 1



Convolutional Layer in CNN (2)

Example: Convolution operation with zero-padding with 1 pixel and stride of 2 [1].



Q 1: Why padding?

Ans: Otherwise, with convolution operation, the output-size shrinks w.r.t. the input-size.

For example, padding with 1 pixel and the stride of 1, will keep the input and output sizes same.

Q 2: How many parameters are there if this above operation is occurring in a layer?

Ans: The filter contains 3 x 3 x 1 = 9 parameters. Besides, there is one bias value. Therefore, there are 10 parameters involved.

Convolutional Layer in CNN (3)

Output size Computation [1]:

- **Output size =** O x O **=** $[(I-F) / S + 1]^2$
  - where,
    - Input size = I x I
    - Filter size = F x F
    - Stride Size = S x S (in both dimensions, i.e., S per move)

  - For padding size P, O = [I+2P-F /S +1].

Example: In the previous slide, due to padding the input changed from 5 x 5 to 7 x 7. So, now I=> (I+2)=7. From filter size (3 x 3) we get F= 3. Stride, S =2.

Therefore, the output size: O = (I+2-F)/S + 1 = (7-3)/2 +1 = 3 => 3 x 3.

In this case, if we use S = 1, we get: O = (7-3) /1 + 1 = 5 => 5 x 5, then, the original input and the corresponding output sizes will remain same.

Convolutional Layer in CNN (4)

Output size Computation for Color Image[1]:

- Color Image includes depth. Therefore, both input and filter will have depth (=3, for R, G, and B).

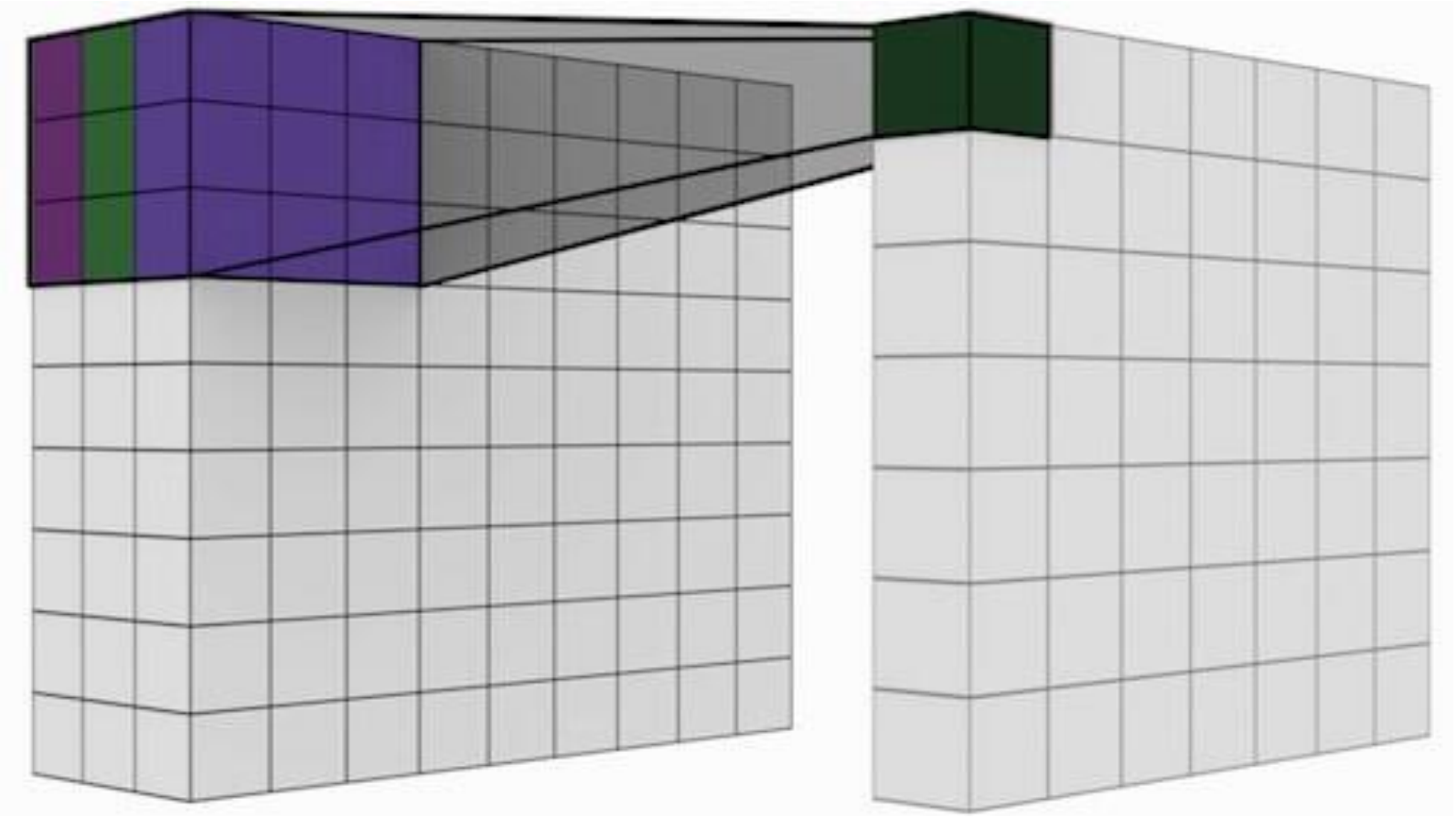


Figure: Regular Convolution - all input channels (e.g., RBG) are combined into 1 channel.

Convolutional Layer in CNN (5)

Output size Computation for Color Image[1]:

- Color Image includes depth. Therefore, both input and filter will have depth (=3, for R, G, and B).

Q 1: For an input color image of size = 9 x 9 (x 3), and 6 filters of 7 x 7 (x3) with stride 1 and pad 3 – what would be the output size (volume)?

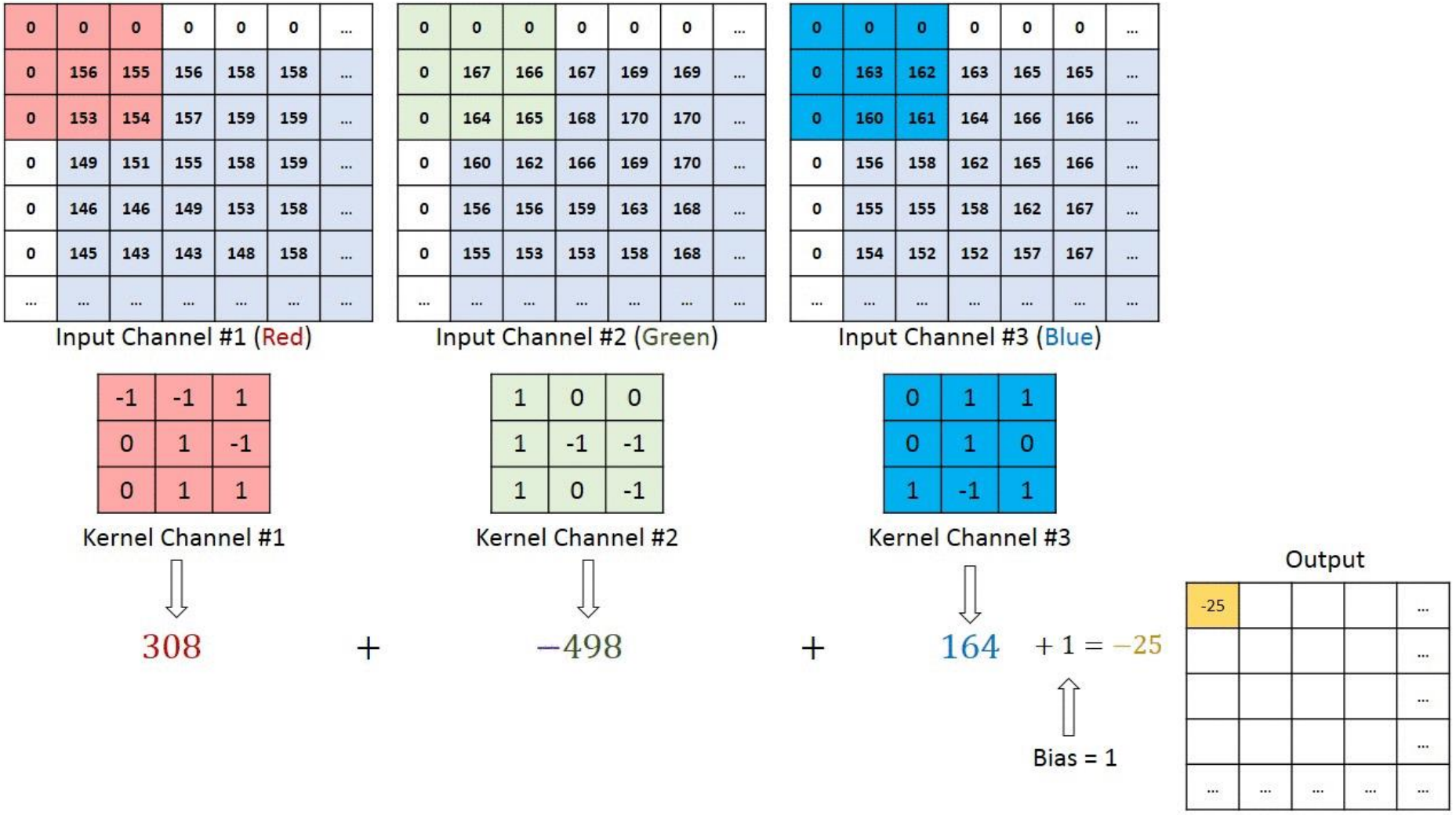
Ans: Output for one filter, O = (I+2P-F)/S+1 = (9+2x3-7)/1+1 = 9.
Therefore, the output volume for 6 filters = (9 x 9) x 6 = 486.

Q 2: How many parameters are there in layer **Q 1**?

Ans: The filters contain almost all the parameters and layer also will have 1 bias parameter. Each filter will have 7 x 7 x 3 = 147. Thus, 6 filters will have 6 x 147 = 882 parameters. Including bias, there will be a total of (882+1) = 883 parameters.

Convolutional Layer in CNN (6)

Output value (& size) Computation for Color Image [1]:



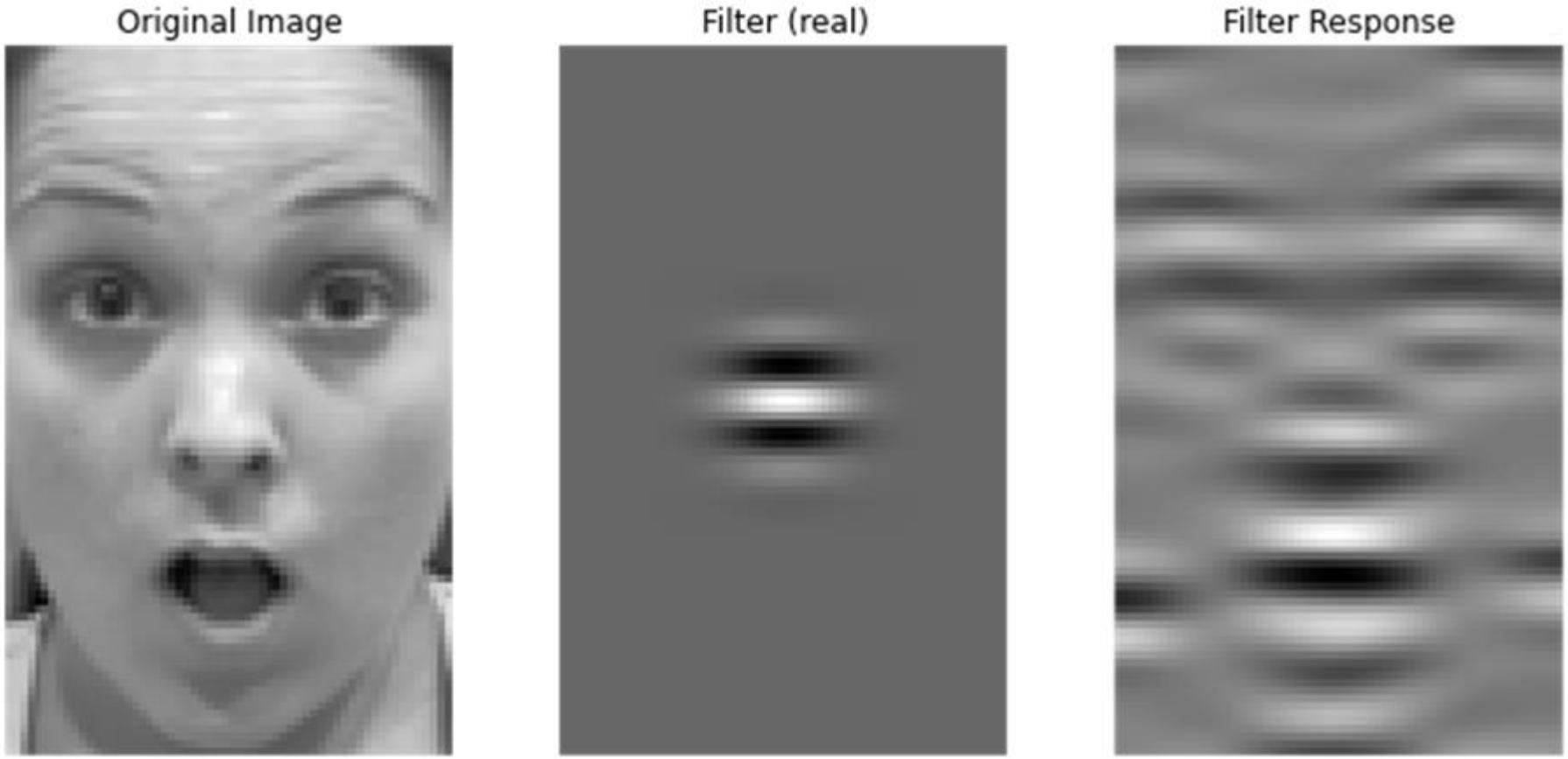
Convolutional Layer in CNN (7)

Example: **Sobel** Filter for edge detection [1-2] => Expressed in G_x and G_y

$$G_x = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} \quad G_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

Convolutional Layer in CNN (8)

Example: Application of Gabor Filter



Pooling Layer (1)

Pooling [1]:

- The purpose of the pooling layer is to reduce or **downsize** the spatial resolution of the feature maps.

- Previously, it was a common practice to use average pooling layer (see LeNet 5). However, in more recent models, max pooling layer is applied.

  - Why? **Ans**: In 2007, backpropagation was applied for the first time to a DCNN like architecture that used max pooling. It was shown empirically that the max pooling operation was vastly superior.
  - Also, we want to catch the max. firing in the neuron than the average.

Pooling Layer (2)

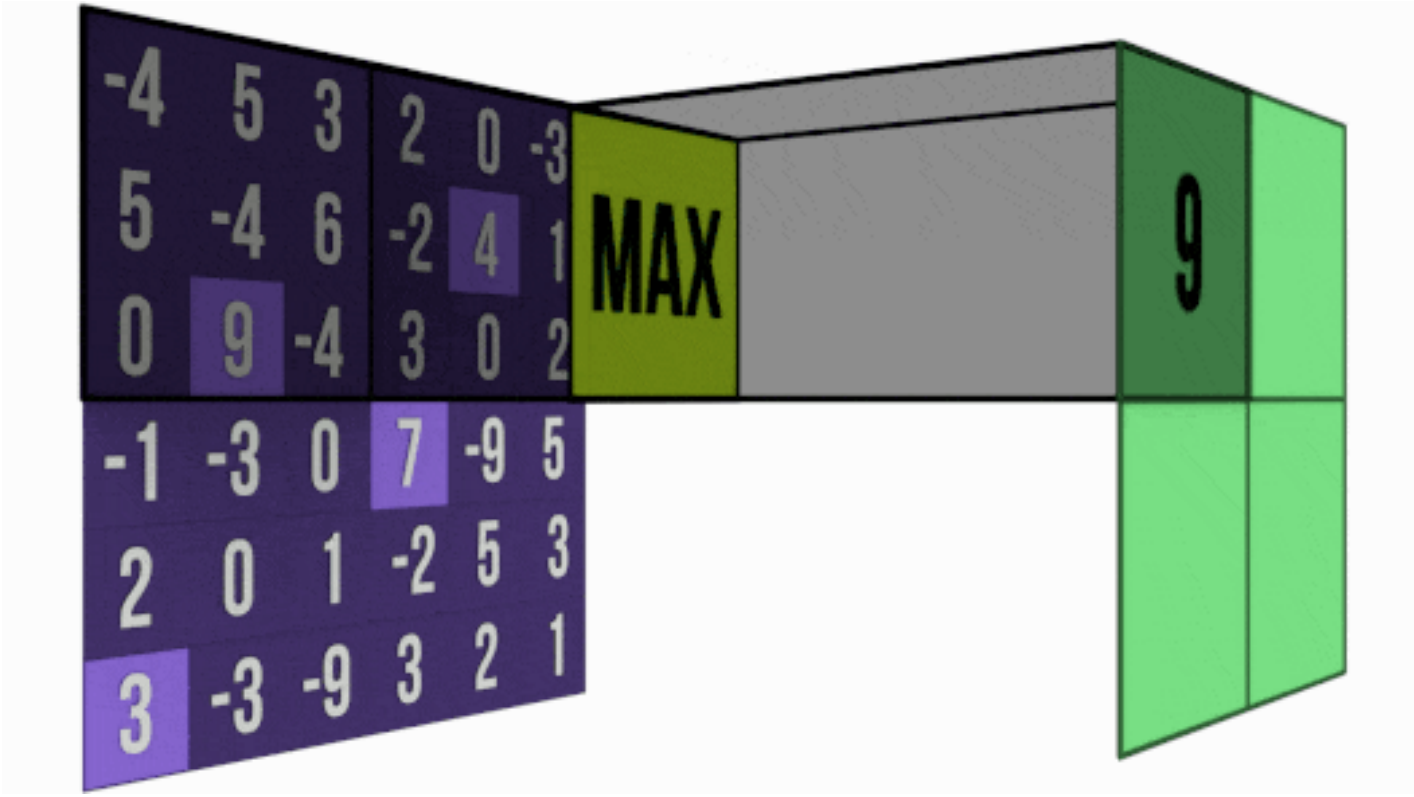


Figure: Max Pooling Layer [1].

Note:

- Usually there is no overlapping of the filter position when pooling. Afterall, the target is to down-scale the input.
- Instead of pooling – we sometimes apply more stride to down-sample the input in the convolution layer.

Pooling Layer (3)

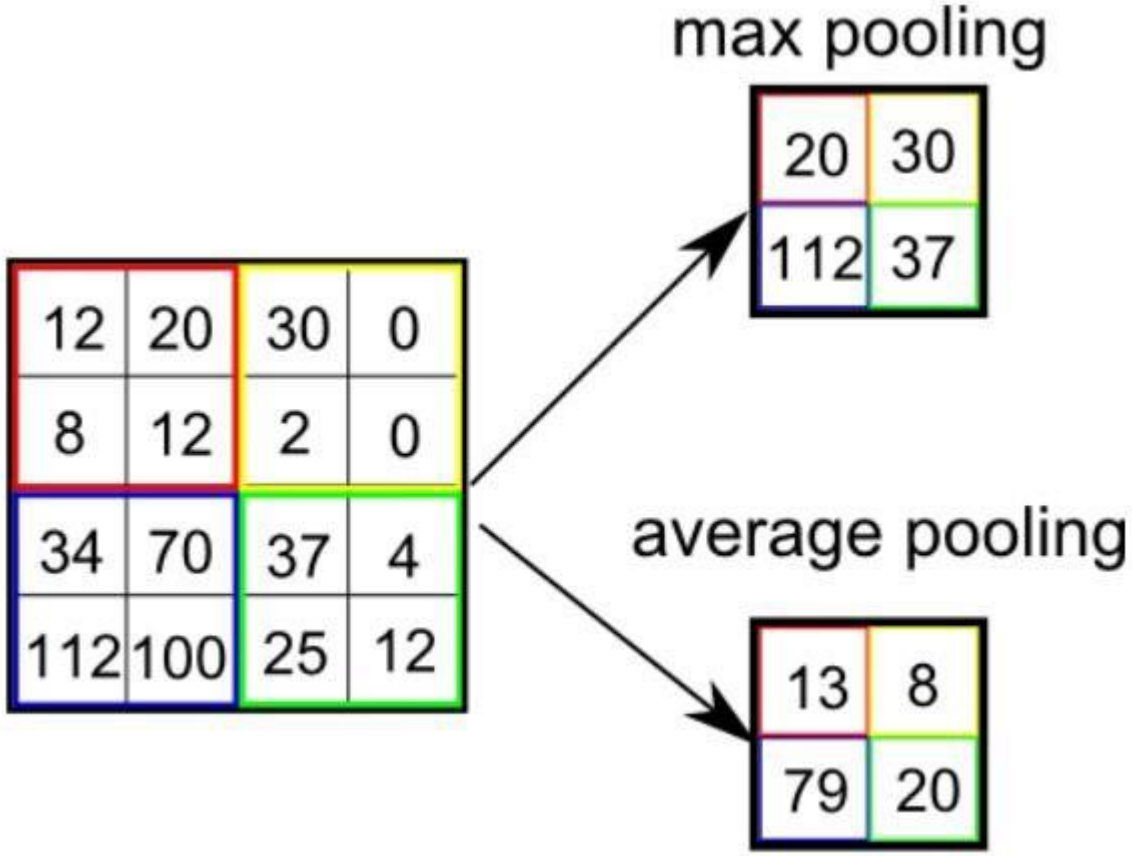


Figure: Max Pooling versus Average Pooling Layer [1].

AlexNet: Wins ILSVRC in 2012 (1)

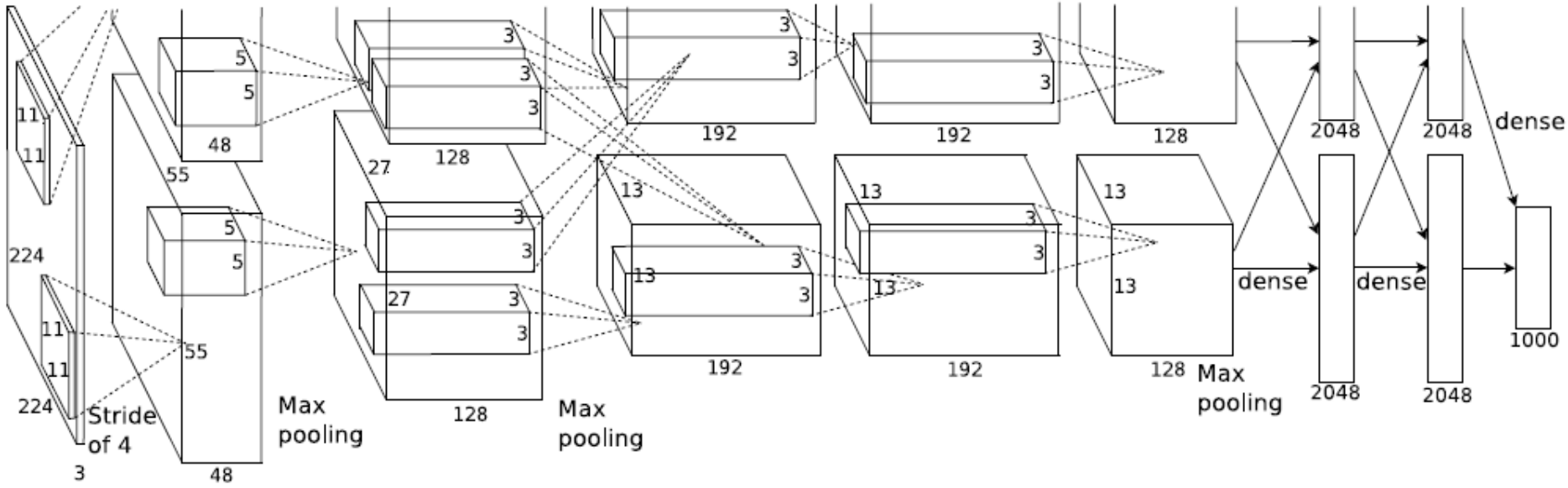


Figure: AlexNet [1-2], which applied DCNN architecture, split over two GPUs.

- AlexNet takes in input a color (RGB) image of dimension 22**7** X 22**7** x 3.
- First, it applies a Convolution Layer (CL) of **96** filters: 11 X 11, stride =4 and pad =0.
- Next, a Max-Pooling Layer (M-PL) of filter: 3 X 3, & stride=2 are applied.
- Normalization is applied (not shown).
- Again, a CL of **256** filters of size 5 X 5, stride = 1 & pad =2.
- Then, a M-PL of filter size 3 X 3 and stride = 2.
- Normalization is applied (not shown).
- Again, a CL of **384** filters of size 3 X 3, stride = 1 & pad = 1.
- Again, a CL of **384** filters of size 3 X 3, stride = 1 & pad = 1. [… continue]

… Implementing MLP with Keras

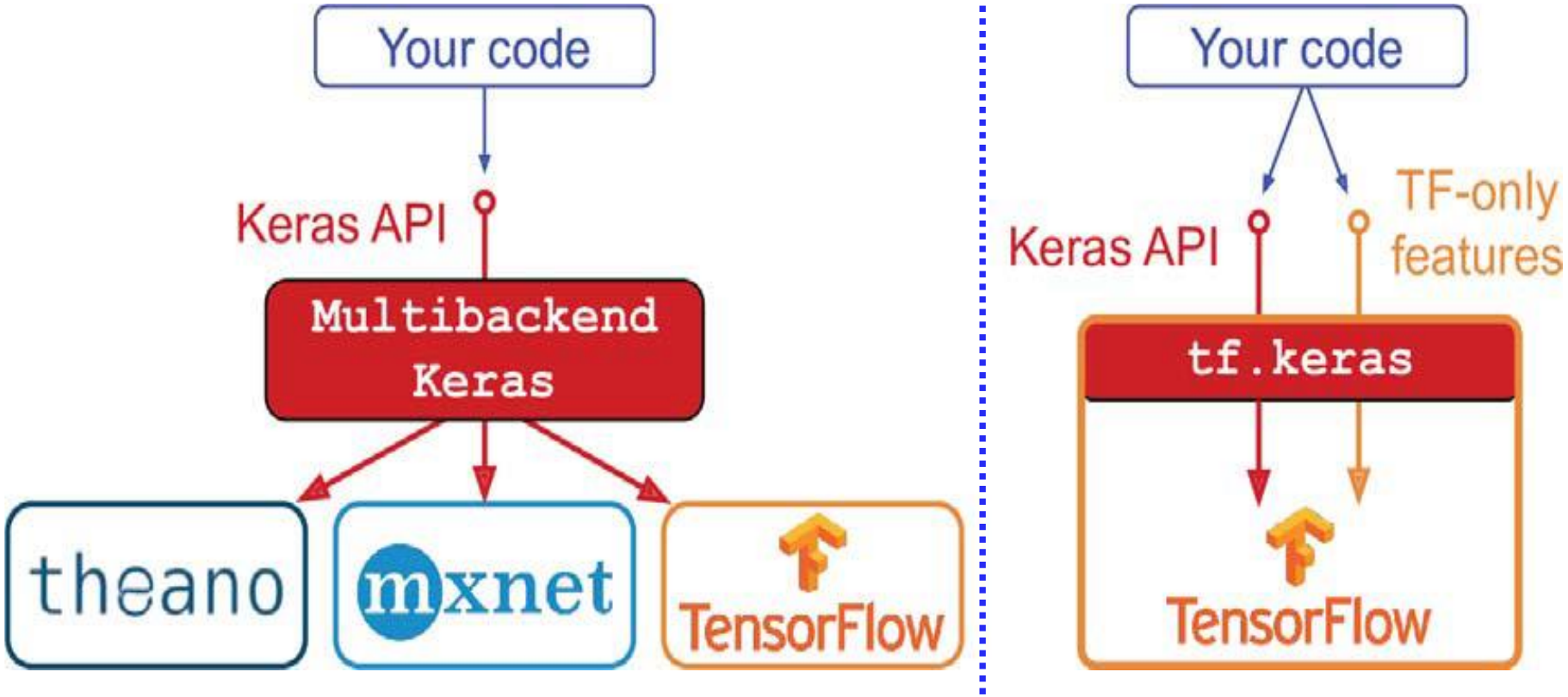
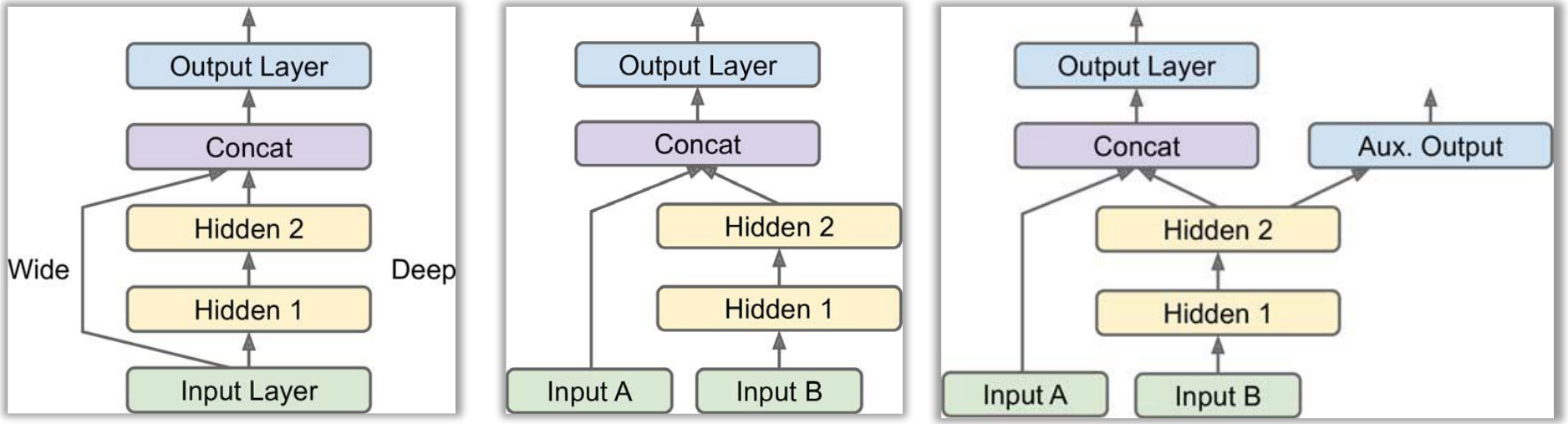


Figure 5: Two implementations of the Keras API: multibackend Keras (left) and tf.keras (right).

- As tf.keras is bundled with TensorFlow, we can start by installing TensorFlow to obtain Keras at the same go.

Programming Aspect Using Keras/TF

- Using the Sequential API
- Using the Functional API (to Build Complex Models) such as Fig. 7:



(a) (b) (c)

Figure 7: Wide & Deep neural network (a) simple example (b) with multiple inputs, and (c) with multiple inputs and multiple outputs.

- Build Dynamic Models using the Subclassing API

… Programming Aspect Using Keras/TF

- Save and Restore a Model using:
  - CallBacks
  - Early Stopping
- Use TensorBoard for Visualization

Nonsaturating Activation Functions

- The ReLU activation function behaves much better in deep neural networks because it does not saturate for positive values and because it is fast to compute.

- Unfortunately, the ReLU activation function is not perfect. It suffers from a problem known as the *dying ReLUs*:
  - during training, some neurons effectively "die," meaning they stop outputting anything other than 0.
  - A neuron dies when its weights get tweaked in such a way that the weighted sum of its inputs are negative for all instances in the training set.
    - When this happens, it just keeps outputting zeros, and Gradient Descent does not affect it anymore because the gradient of the ReLU function is zero when its input is negative.
  - Unless it is part of the first hidden layer, a dead neuron may sometimes come back to life: Gradient Descent may indeed tweak neurons in the layers below in such a way that the weighted sum of the dead neuron's inputs is positive again.

- In some cases, you may find that half of your network's neurons are dead, especially if you used a large learning rate.

… Nonsaturating Activation Functions

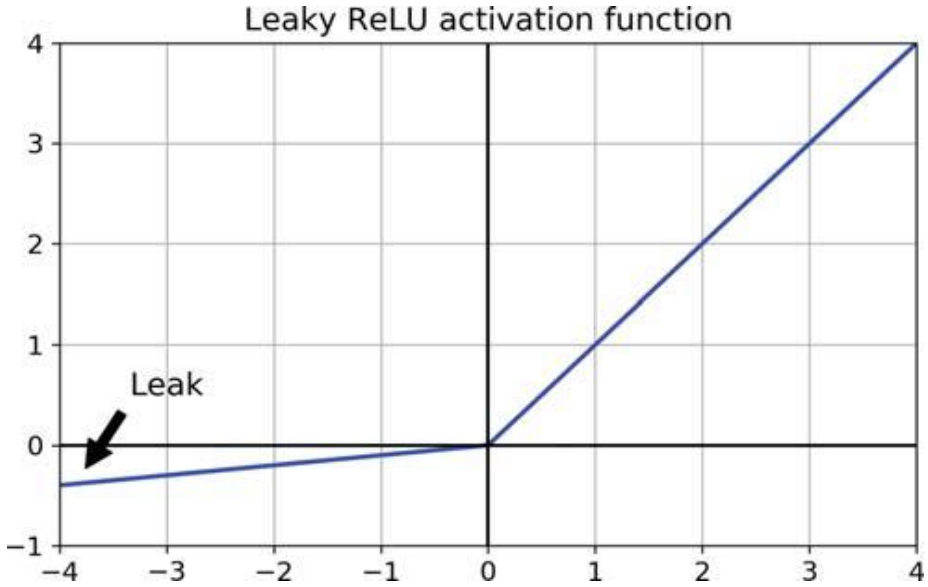


Figure 2: Leaky ReLU: like ReLU, but with a small slope for negative values.

- To solve, a variant of the ReLU function, such as the *leaky ReLU,* defined as $\text{LeakyReLU}_\alpha(z) = \max(\alpha z, z)$ (see Figure 2), can be used.
- The hyperparameter α defines how much the function "leaks": it is the slope of the function for $z < 0$ and is typically set to 0.01.
- This small slope ensures that leaky ReLUs never die; they can go into a long coma, but they have a chance to eventually wake up.
- Leaky variants always outperformed the strict ReLU.
- In fact, setting $\alpha = 0.2$ (a huge leak) seemed to result in better performance than $\alpha = 0.01$ (a small leak).

… Nonsaturating Activation Functions

Randomized leaky ReLU (RReLU):

- performed fairly well and seemed to act as a regularizer (reducing the risk of overfitting the training set).
- here, α is picked randomly in a given range during training and is fixed to an average value during testing.

Parametric leaky ReLU (PReLU):

- α is learned during training (instead of being a hyperparameter, it becomes a parameter that can be modified by backpropagation like any other parameter).
- PReLU was reported to strongly outperform ReLU on large image datasets, but on smaller datasets it runs the risk of overfitting the training set.

… Nonsaturating Activation Functions

$$\mathrm{ELU}_\alpha(z) = \begin{cases} \alpha(exp(z) - 1) & if\, z < 0 \\ z & if\, z \geq 0 \end{cases} \quad (2)$$

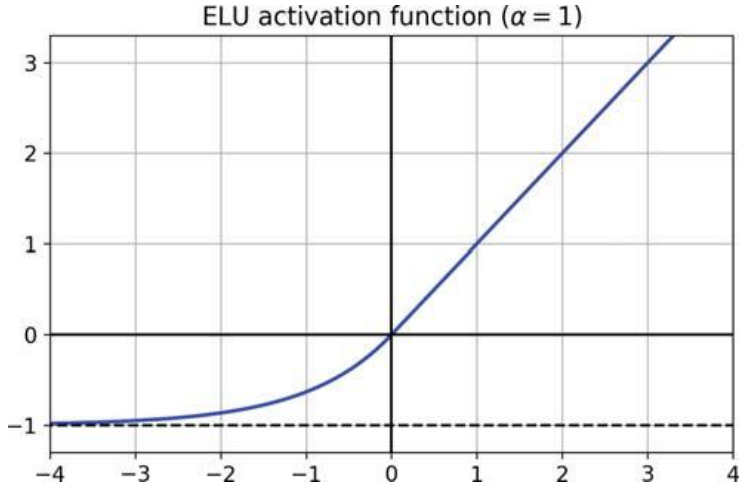


Figure 3: ELU activation function.

Exponential linear unit (ELU):

- Equation 2 defines the ELU and Figure 3 graphs it.
- ELU outperformed all the ReLU variants in the authors' experiments: training time was reduced, and the neural network performed better on the test set.
- The main drawback of the ELU activation function is that it is slower to compute than the ReLU function and its variants (due to the use of the exponential function).
- Its faster convergence rate during training compensates for that slow computation, but still, at test time an ELU network will be slower than a ReLU network.

… Nonsaturating Activation Functions

Scaled ELU (SELU): a *scaled* variant of the ELU

- If you build a neural network composed exclusively of a stack of dense layers, and if all hidden layers use the SELU activation function, then the network will self-normalize:
  - the output of each layer will tend to preserve a mean of 0 and standard deviation of 1 during training, which solves the vanishing/exploding gradients problem.
- As a result, the SELU activation function often significantly outperforms other activation functions for such neural nets, especially deep ones.

… Nonsaturating Activation Functions

There are, however, a few conditions for self-normalization to happen when using SELU:

- The input features must be standardized (mean 0 and standard deviation 1).
- Every hidden layer's weights must be initialized with LeCun normal initialization. In Keras, this means setting `kernel_initializer="lecun_normal"`.
- The network's architecture must be sequential. Unfortunately, if you try to use SELU in nonsequential architectures, such as recurrent networks or networks with *skip connections* (i.e., connections that skip layers, such as in Wide & Deep nets), self-normalization will not be guaranteed, so SELU will not necessarily outperform other activation functions.
- The paper only guarantees self-normalization if all layers are dense, but some researchers have noted that the SELU activation function can improve performance in convolutional neural nets as well.

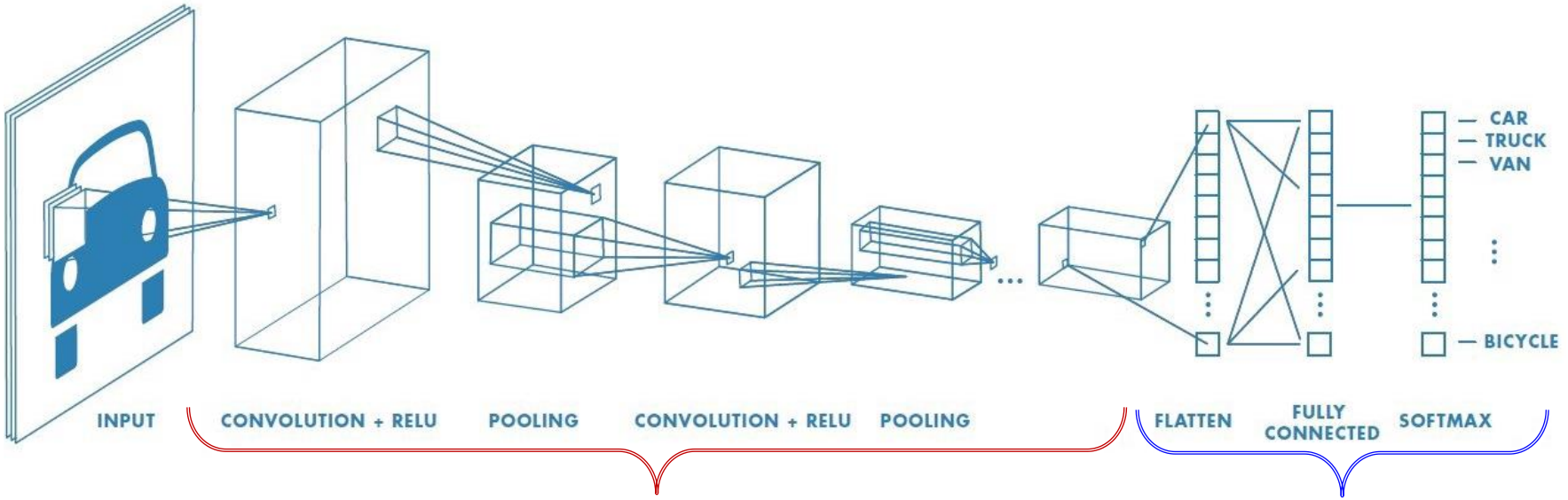
… Nonsaturating Activation Functions

So, which activation function should you use for the hidden layers of your deep neural networks?

In general SELU > ELU > leaky ReLU (& variants) > ReLU > tanh > logistic.

- If the network's architecture prevents it from self-normalizing, then ELU may perform better than SELU (since SELU is not smooth at z = 0).
- If you care a lot about runtime latency, then you may prefer leaky ReLU.
  - If you don't want to tweak yet another hyperparameter, you may use the default α values used by Keras (e.g., 0.3 for leaky ReLU).
- If you have spare time and computing power, you can use cross-validation (CV) to evaluate other activation functions, such as RReLU if your network is overfitting, or PReLU if you have a huge training set.
- That said, because ReLU is the most used activation function (by far), many libraries and hardware accelerators provide ReLU-specific optimizations; therefore, if speed is your priority, ReLU might still be the best choice.

Deep/Convolutional Neural Network



Convolution Layers: Extracts Features.

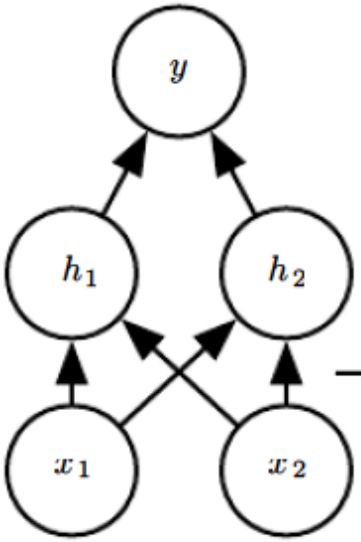
Fully Connected (FC) Layers: Performs Classification.

Figure: Convolutional Neural Network [1] has 2 different pipelines.

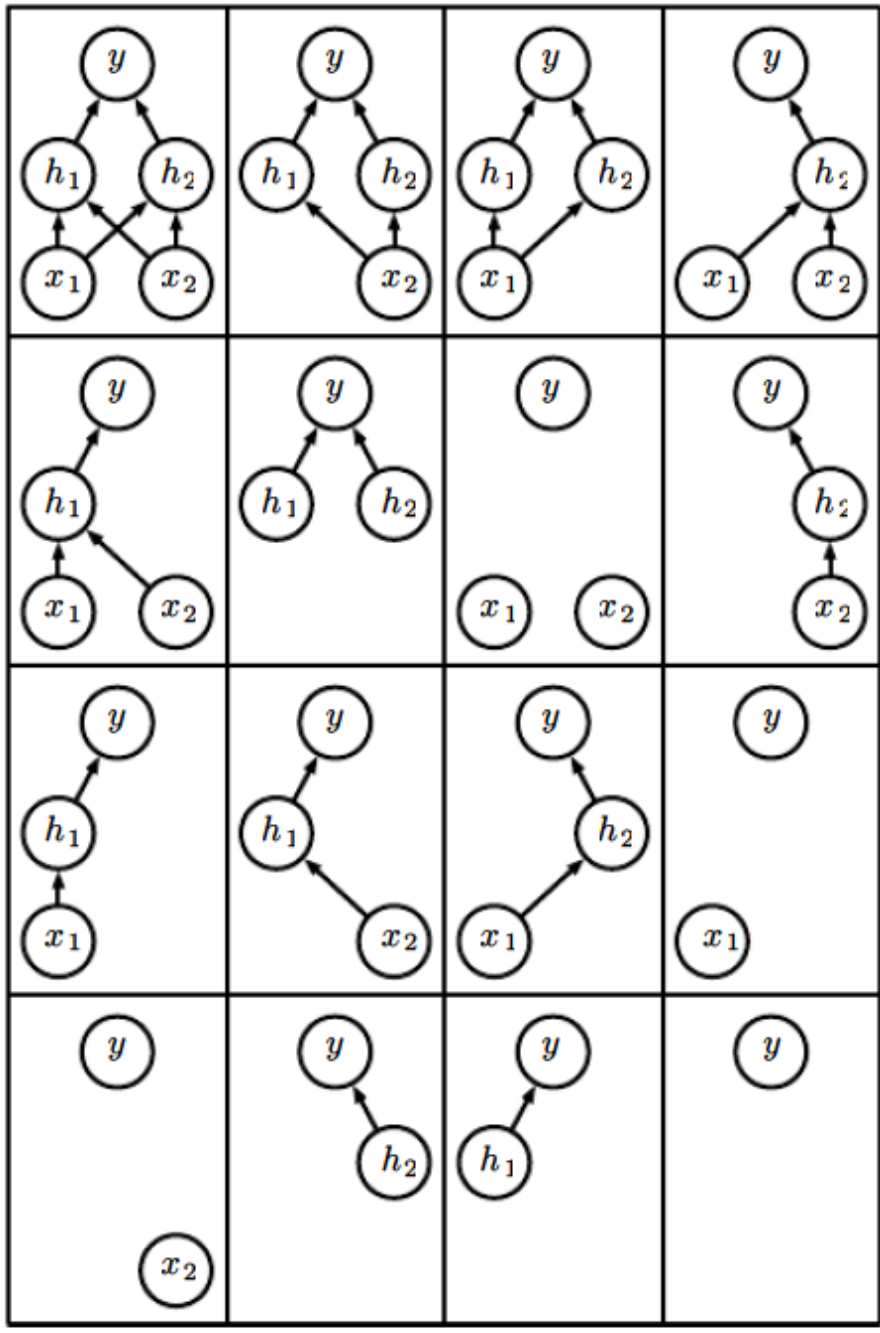
DropOut=> Regularization (1)

- In dropout, we can effectively remove a **unit** from a network by multiplying its output value by zero.
- Dropout is a simple way to prevent DNN from overfitting & improve model accuracy [1-2].
- Dropout can be thought of as a method of making bagging practical for ensembles of very many large neural networks.
- Dropout trains the ensemble consisting of all sub-networks that can be formed by removing **non-output** units from an underlying base network (see figure next).

DropOut=> Regularization (2)



Base network



Ensemble of subnetworks

- **Figure** 1: There are sixteen possible subsets of these four units.
- Here, a large proportion of the resulting networks have no input units or no path connecting the input to the output.
- This problem becomes insignificant for networks with wider layers, where the probability of dropping all possible paths from inputs to outputs becomes smaller.

DropOut=> Regularization (2)

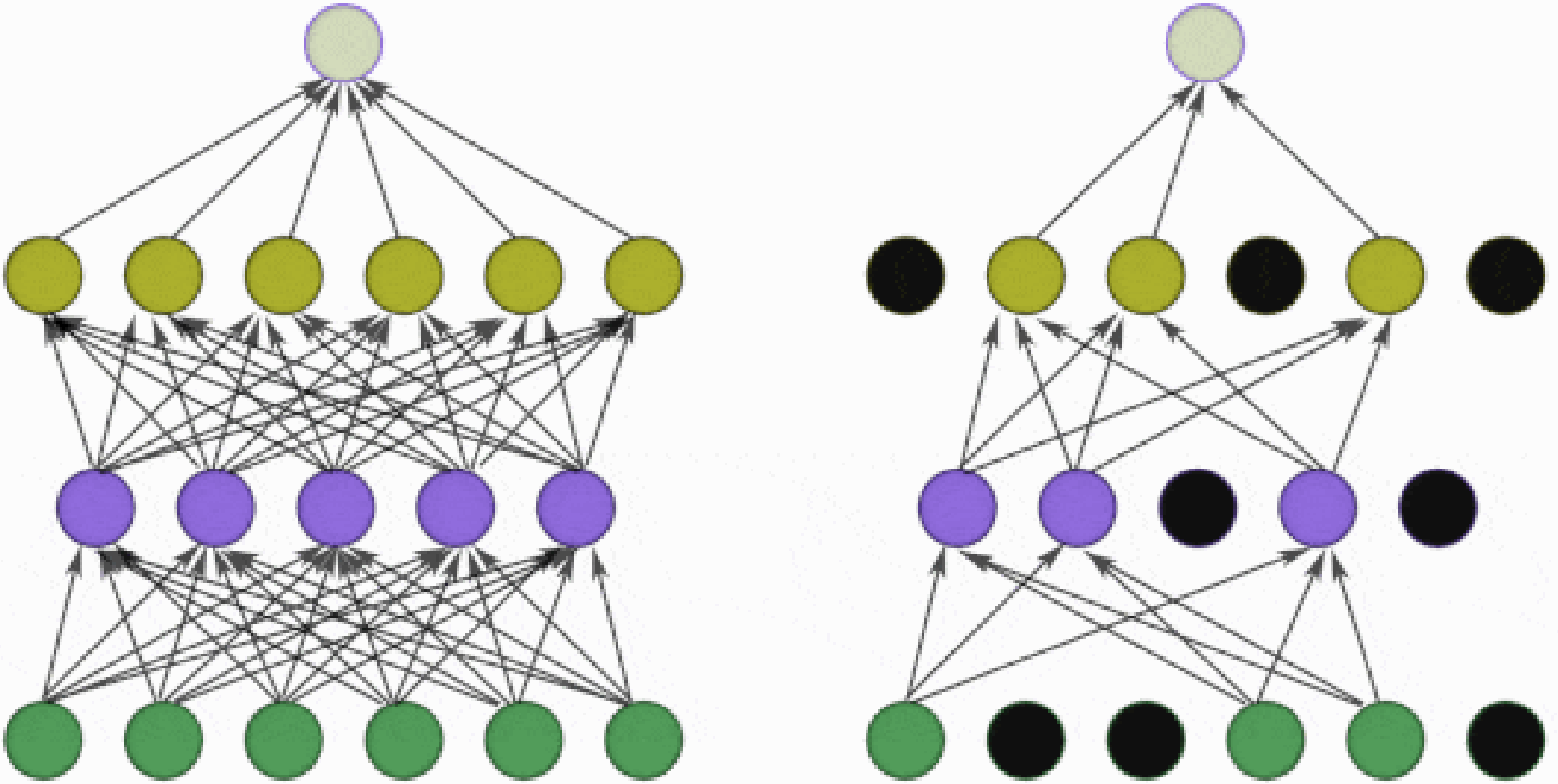
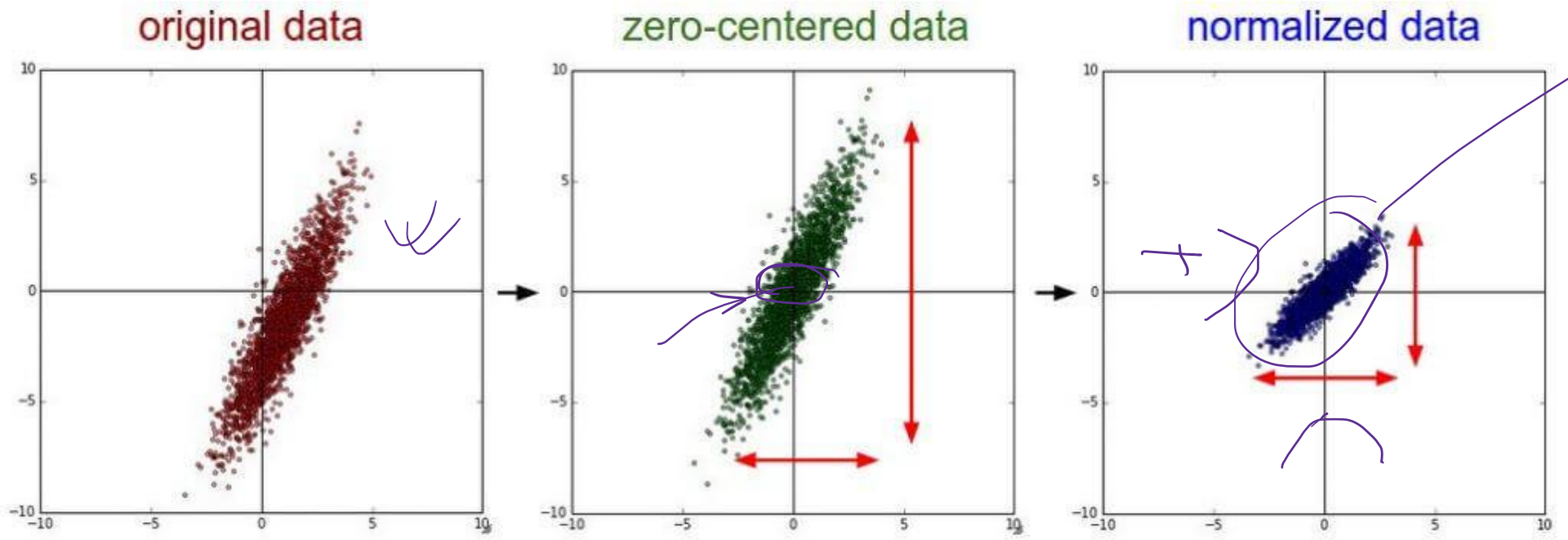


Figure 2: **Left**: A standard neural net with 2 hidden layers.
Right: An example of a thinned net produced by applying dropout to the network on the left. Black units have been dropped.

Data Processing (3)



- Zero-centered & Normalized data [1]:
  - To make the data zero-centered, subtract mean feature from its feature values (For RGB we can subtract channel wise).
  - Then, divide the zero-centered feature data by its corresponding standard deviation.

Note: For image data, normalization may not be necessary as the raw features (pixels) typically have similar range (0 to 255).

Data Processing (7)

- **Algorithm** #1: BN Transform, applied to activation *x* over a mini-batch:

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

- Note: γ and β are the parameters to learn from the training (why γ, β?).
- Ans: These parameters recover the identity mapping to provide output in actual scale.
- These parameters could also be approximated as:

$$\gamma^{(k)} = \sqrt{\mathrm{Var}[x^{(k)}]}$$
$$\beta^{(k)} = \mathrm{E}[x^{(k)}]$$

Data Processing (8)

- Advantages of (BN) [1-2]: The benefits of Batch Normalization are:
  - Networks train faster,
  - allows higher learning rates,
  - makes weights easier to initialize,
  - makes more activation functions viable,
  - simplifies the creation of deeper networks, and
  - provides some regularization (as it adds a little noise to the network).

- Disadvantages of BN: BN -
  - adds about 30% computational overhead, and
  - requires additional 25% of parameters per iteration.

Optimization (1)

1. Basic Algorithms for Optimization [1]: From chapter 4, we have learnt that for each epoch, during training, we optimize the NN.

- In the previous chapters, we learnt one of the basic algorithms for optimization, the *Gradient Descent* (GD) equation for optimizing MSE (or, RSS) as:

$$\beta_j(t+1) = \beta_j(t) + \frac{2\alpha}{N}\sum_{i=1}^{N}\left(y(i) - x^T(i)\ \beta\right)\ .\ x(i)_j \quad (1)$$

- Further, with this regularization goal $\min RSS_\lambda(\beta) = \sum_{i=1}^{N}\left[\left(\hat{y}(x_i,\beta) - y_i\right)^2 + \lambda\sum_{j=1}^{p}\beta_j^2\right]$,

  we obtained the GD equation with regularization as:

$$\beta_j(t+1) = \beta_j(t) + \frac{2\alpha}{N}\left[\sum_{i=1}^{N}\left(y(i) - x^T(i)\ \beta\right)\ .\ x(i)_j - \lambda\beta(t)_j\right] \quad (2)$$

- Or,

$$\beta_j(t+1) = \beta_j(t)\left(1 - \frac{2\alpha\lambda}{N}\right) + \frac{2\alpha}{N}\sum_{i=1}^{N}\left(y(i) - x^T(i)\ \beta\right)\ .\ x(i)_j \quad (3)$$

Optimization (2)

1.1 Stochastic Gradient Descent (SGD): *Stochastic Gradient Descent* (SGD) and its variants are probably the most used optimization algorithms for deep learning [1].

- For a massive dataset for training in Machine Learning or BigData, computation of next β in the above equations involve, all the training instances (also called *Batch Gradient Descent*), which becomes computationally too expensive.
- Optimization algorithms that use only a single example at a time are sometimes called stochastic or sometimes online methods.
- However, the usage of a single instance or example to compute next β would be too small to be a representative or unbiased approach for the whole dataset.
- For deep learning, most algorithms fall somewhere in between, using more than one but less than all the training examples.

Optimization (3)

- These algorithms were traditionally called minibatch or minibatch stochastic methods and it is now common to simply call them stochastic methods or stochastic gradient descent (SGD).
- Algorithm 1.1 describes the SGD method:

Algorithm 1.1: Stochastic gradient descent (SGD)

Require: Learning rate α; initial parameter $\beta(0)$; iteration $t = 0$.

WHILE stopping criterion not met DO

Sample a minibatch of m examples, $\{(x, y)_1, \ldots, (x, y)_m\}$, from N training points

Compute $\beta(t+1)$ for $\forall_j$ as:

$$\beta_j(t+1) = \beta_j(t) + \frac{2\alpha}{m}\sum_{i=1}^{m}(y(i) - x^T(i)\ \beta)\ .\ x(i)_j$$

$$\text{Or, } \beta_j(t+1) = \beta_j(t)\left(1 - \frac{2\alpha\lambda}{m}\right) + \frac{2\alpha}{m}\sum_{i=1}^{m}(y(i) - x^T(i)\ \beta)\ .\ x(i)_j$$

END WHILE

The 'or' part is the regularized version of the SGD algorithm.

Optimization (4)

- For SGD, it is possible to obtain a fair estimate of the gradient by taking the average gradient on a minibatch or, a small number of *m* instances.

- Of course, the better the *m* samples represent the training set, the better the outcome is expected to be.

- Methods that compute updates based only on the gradient of the first derivates of the loss/error function are usually able to handle smaller batch sizes like 100 instances.

- Second-order methods, which also use the Hessian matrix H, typically require much larger batch sizes like 10,000 instances.

Optimization (5)

- **Learning rate**: A crucial parameter for the GD/SGD algorithm is the learning rate α.
- In practice, it is necessary to decrease the learning rate over time (why?). Thus, we rather, denote the learning rate at iteration t as $\alpha(t)$.
- In practice, it is common to decay the learning rate linearly until iteration τ, as:

$$\alpha(t+1) = \begin{cases} (1-\gamma)\,\alpha(t), & \text{where, } t < \tau \\ \alpha(\tau), & \text{where, } t \geq \tau \end{cases} \qquad (4)$$

- Learning rate can be chosen by trial and error, but it is usually best to determine it by monitoring learning curves that plot the objective function as a function of time.
- When using the linear schedule, the parameters to choose are $\alpha(0)$ and $\alpha(\tau)$ or τ. Usually, τ may be set to the number of iterations required to make a few hundred passes through the training set. Typically $\alpha(\tau)$ should be set to roughly 1% the value of $\alpha(0)$.

Optimization (6)

- **Initial learning rate, $\alpha(0)$**: The main question is how to set the initial learning rate $\alpha(0)$?
- If we assign a very large value, overshooting can occur, which will eventually increase the value of the error/cost/loss functions significantly.
- On the other hand, if the learning rate is too low, learning proceeds exceptionally slowly.
- Typically, the optimal initial learning rate, in terms of total training time and the final cost value, is higher than the learning rate that yields the best performance after the first 100 iterations or so.
- Therefore, it is usually best to monitor the first several iterations and use a learning rate that is higher than the best-performing learning rate at this time, but not so high that it causes overshooting.

Optimization (7)

1.2 Momentum: While SGD remains a popular optimization strategy being easy to implement, learning with it can sometimes be slow.

- The method of **momentum** is designed to accelerate learning, especially in the face of high curvature, small but consistent gradients, or noisy gradients.
- The momentum algorithm accumulates an **exponentially decaying moving average** of the past gradients and continues to move in their direction.
- Formally, the momentum algorithm introduces a variable ***v*** that plays the role of velocity—it is the direction and speed at which the parameters move through parameter space.
- The velocity is set to an exponentially decaying average of the negative gradient.

Optimization (8)

- A hyperparameter $\delta \in [0, 1)$, called the friction parameter, determines how quickly the contributions of previous gradients exponentially decay. Algorithm 1.2 describes the momentum algorithm in SGD:

Algorithm 1.2: Stochastic gradient descent (SGD) with momentum

Require: Learning rate α; initial parameter $\beta(0)$; momentum parameter δ; velocity $v(-1) = 0$; iteration $t = 0$.

WHILE stopping criterion not met DO

Sample a minibatch of m examples, $\{(x, y)_1, \ldots, (x, y)_m\}$, from N training points

Compute $\beta(t+1)$ for $\forall_j$ as:

$$v(t) = v(t-1)\,\delta + \frac{2\alpha}{m}\sum_{i=1}^{m}(y(i) - x^T(i)\,\beta)\,.\,x(i)_j$$

$$\beta_j(t+1) = \beta_j(t) + v(t)$$

$$\text{Or, } \beta_j(t+1) = \beta_j(t)\left(1 - \frac{2\alpha\lambda}{m}\right) + v(t)$$

END WHILE

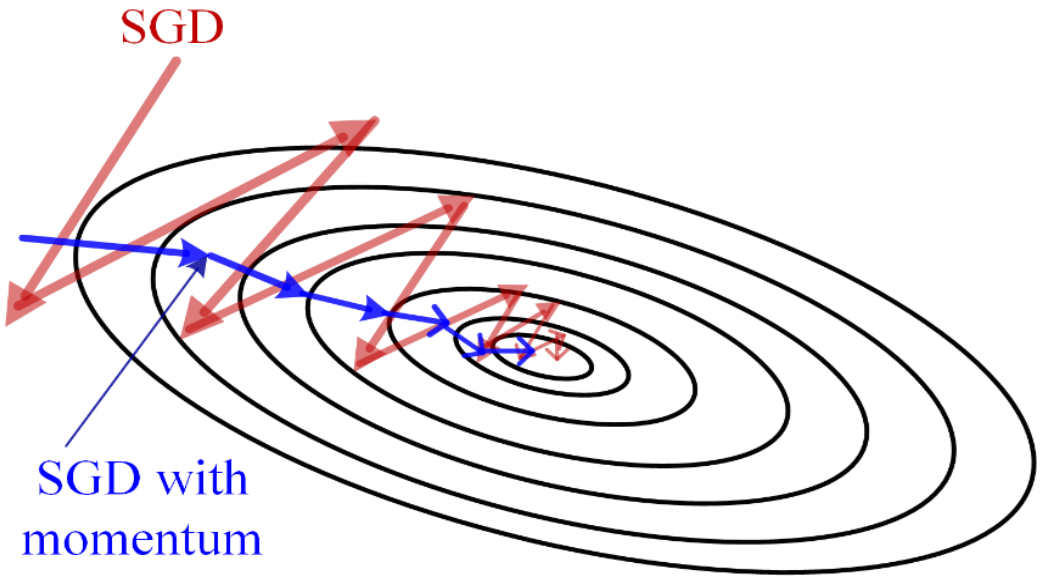
- As shown in Algorithm 1.2, the velocity v accumulates the gradient elements $\frac{2\alpha}{m}\sum_{i=1}^{m}\left(y(i) - x^T(i)\,\beta\right)\,.\,x(i)_j$.

Optimization (9)

- The larger the δ is relative to α, the more previous gradients affect the current direction.
- Here, the size of the step depends on how large and how aligned a sequence of gradients are. The step size is largest when many successive gradients point in exactly the same direction.
- Common values of δ used in practice include 0.5 and **0.9 to 0.99**.
- Like the learning rate, δ may also be adapted over time. Typically it begins with a small value and is later raised.
- It is less important to adapt δ over time than to shrink α over time.

Optimization (10)

- The effect of momentum is illustrated in the **Figure** below:



- **Figure 1.1:** Momentum aims primarily to solve two problems:
  - poor conditioning of the Hessian matrix and
  - variance in the stochastic gradient.
- Here, the contour lines depict a quadratic loss function with a poorly conditioned Hessian matrix.
- The heavily zig-zak **red path** indicates the path of GD/SGD without the momentum.
- The **blue path** cutting across the contours shows the path followed by the momentum learning rule, which is a less zig-zak path that helps faster convergence.

Optimization (11)

1.3 Nesterov Momentum: ~ is a variant of the momentum algorithm.

- Previously, for momentum, instead of applying, $\boldsymbol{\beta(t+1) = \beta(t) - \frac{\alpha}{m}\sum_{i=1}^{m}\frac{\partial E}{\partial \beta(t)}}$
- We applied,
  - $\boldsymbol{v(t) = v(t-1)\delta - \frac{\alpha}{m}\sum_{i=1}^{m}\frac{\partial E}{\partial \beta(t)}}$, where, velocity $v(0) = 0$, when iteration $t = 1$
  - $\boldsymbol{\beta(t+1) = \beta(t) + v(t)}$
- Now, if we rewrite (error or, loss term) **E** as $\boldsymbol{L\left(f\left(x^{(i)};\theta\right), y^{(i)}\right)}$, we can write the Nesterov momentum equations as:
  - $\boldsymbol{v(t) = v(t-1)\delta - \frac{\alpha}{m}\sum_{i=1}^{m}\frac{\partial}{\partial \beta(t)} L\left(f\left(x^{(i)};\theta + v(t-1)\delta\right), y^{(i)}\right)}$, and
  - $\boldsymbol{\beta(t+1) = \beta(t) + v(t)}$
- With Nesterov momentum the gradient is evaluated after the current velocity is applied.
- It is shown to improve the error rate in theory, but the improvement may be very marginal or none for some cases in practice.

Optimization (12)

- **Adaptive Gradient Algorithm, AdaGrad:** ~ individually adapts the learning rates of all model parameters by scaling them inversely proportional to the square root of the sum of all their historical squared values.

Algorithm 1.3: The AdaGrad algorithm

Require: Learning rate α; initial parameter $\beta(0)$; small-constant, ϵ = 1e-8; grad-sqr, gs=0; iteration $t = 0$.

WHILE stopping criterion not met DO

Sample a minibatch of m examples $\{(x, y)_1, \ldots, (x, y)_m\}$, from N training points

Compute $\beta(t+1)$ as:

$$gs = gs + \left(\frac{1}{m}\sum_{i=1}^{m}\frac{\partial E}{\partial \beta(t)}\right)^2;$$ $( )^2$ applied element-wise

$$\Delta\beta(t) = -\alpha\frac{\left(\frac{1}{m}\sum_{i=1}^{m}\frac{\partial E}{\partial \beta(t)}\right)}{\sqrt{gs+\epsilon}};$$ Division and sqrt. applied element-wise

$$\beta(t+1) = \beta(t) + \Delta\beta(t)$$

END WHILE

Optimization (13)

- In AdaGrad [1], the parameters with the largest partial derivative of the loss have a correspondingly rapid decrease in their learning rate, while parameters with small partial derivatives have a relatively small decrease in their learning rate.
- The net effect is greater progress in the more gently sloped directions of parameter space.
- In the context of convex optimization, the AdaGrad algorithm enjoys some desirable theoretical properties.
- However, for training deep NN models — the accumulation of squared gradients from the beginning of training can result in a premature and excessive decrease in the effective learning rate.
- AdaGrad performs well for some but not all deep learning models.

Optimization (14)

- **Root Mean Square Propagation**, or **RMSprop**: ~ modifies AdaGrad to perform better in the non-convex setting by changing the gradient accumulation into an exponentially weighted moving average.

Algorithm 1.4: The RMSProp algorithm

Require: Learning rate α; initial parameter $\beta(0)$; small-constant, ϵ = 1e-6; grad, g=0; decay rate ρ; accumulated squared gradient r =0; iteration $t = 0$.

WHILE stopping criterion not met DO

Sample a minibatch of m examples $\{(x, y)_1, \ldots, (x, y)_m\}$, from N training points

Compute $\beta(t+1)$ as:

$$g = \frac{1}{m}\sum_{i=1}^{m}\frac{\partial E}{\partial \beta(t)};$$

$$r = r\rho + (1-\rho)(g)^2; \qquad (\ )^2 \text{ applied element-wise.}$$

$$\Delta\beta(t) = -\alpha\frac{(g)}{\sqrt{\epsilon+r}}; \qquad \frac{1}{\sqrt{\epsilon+r}} \text{ applied element-wise.}$$

$$\beta(t+1) = \beta(t) + \Delta\beta(t)$$

END WHILE

Optimization (15)

- RMSProp uses an exponentially decaying average to discard history from the extreme past so that it can converge rapidly after finding a convex bowl, as if it were an instance of the AdaGrad algorithm initialized within that bowl.
- To apply, RMSProp is combined with Nesterov momentum (combined algorithm is not shown).
- Compared to AdaGrad, the use of the moving average introduces a new hyperparameter, ρ, that controls the length scale of the moving average.
- Empirically, RMSProp has been shown to be an effective and practical optimization algorithm for deep neural networks. It is currently one of the go-to optimization methods being employed routinely by deep learning practitioners.

Optimization (16)

- **Adaptive Moments, Adam** [1-2]: Adam can be best seen as a variant on the combination of RMSProp and momentum.

Algorithm 1.4: The Adam algorithm

Require: Learning rate $\alpha = 0.001$ **(suggested)**; initial parameter $\beta(0)$; small-constant, ϵ = 1e-8 (**suggested**); grad, g=0; decay rates: $\rho 1$ and $\rho 2$ in [0,1) (0.9 and 0.999 **suggested**); Initialize 1st and 2nd moment variables, $S = 0$, $r = 0$; iteration $t = 0$.

WHILE stopping criterion not met DO

Sample a minibatch of m examples $\{(x, y)_1, \ldots, (x, y)_m\}$, from N training points

Compute $\beta(t+1)$ as:

$$g = \frac{1}{m}\sum_{i=1}^{m}\frac{\partial E}{\partial \beta(t)};$$

$$t = t + 1$$

1st moment update: $s = \rho_1 s + (1 - \rho_1)g$

2nd moment update: $r = \rho_2 r + (1 - \rho_2)(g)^2$; $( )^2$ applied element-wise.

Correct bias in 1st moment: $\hat{s} = \frac{s}{1-\rho_1^t}$

Correct bias in 2nd moment: $\hat{r} = \frac{r}{1-\rho_2^t}$

$$\Delta\beta(t) = -\alpha\frac{\hat{s}}{\sqrt{\hat{r}}+\epsilon};$$ Operations applied element-wise.

$$\beta(t+1) = \beta(t) + \Delta\beta(t)$$

END WHILE

Optimization (17)

- First, in Adam, momentum is incorporated directly as an estimate of the first order moment (with exponential weighting) of the gradient.

- Second, Adam bias corrections to the estimates of both the first-order moments (the momentum term) and the (uncentered) second-order moments to account for their initialization at the origin.

- Adam is generally regarded as being robust to the choice of hyperparameters, though the learning rate sometimes needs to be changed from the suggested default.