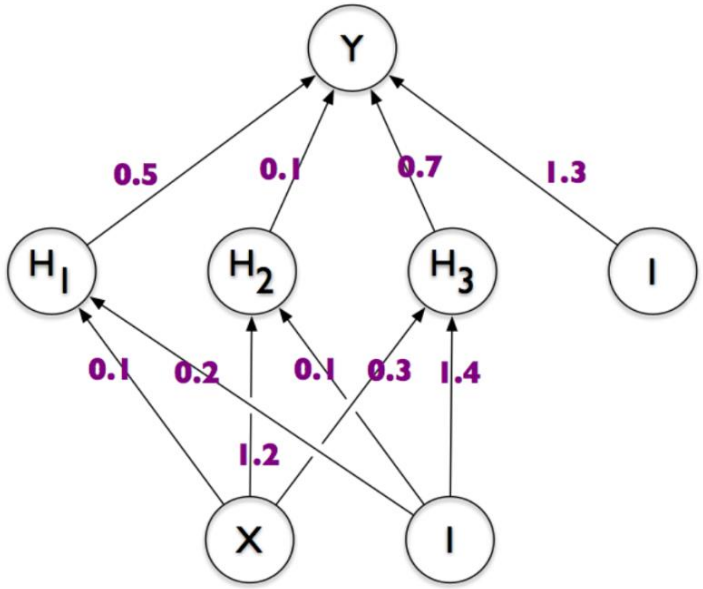
Neural Networks

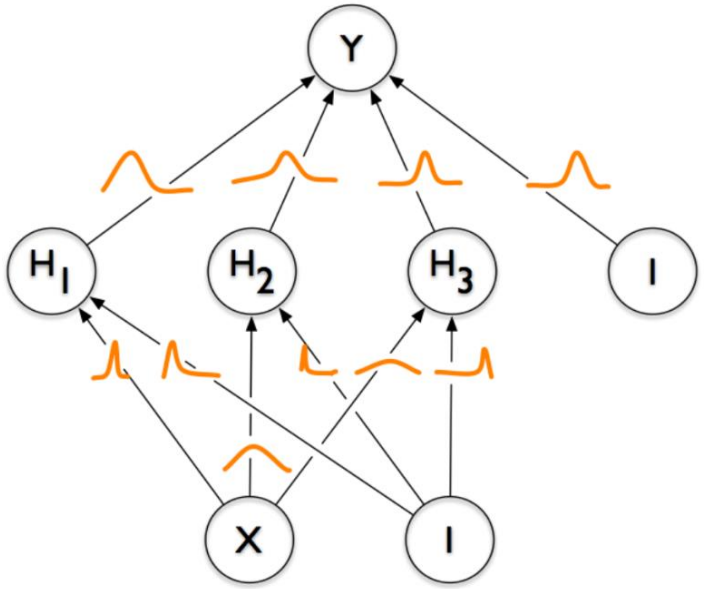
Semester 1

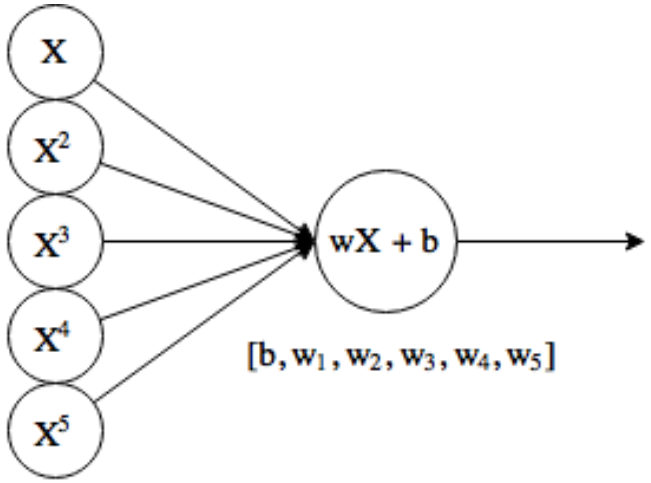
Practical works for GRIAT
RCSE master program

Teacher: Makhmutova Alisa Zufarovna
AZMakhmutova@kai.ru
@DarkAliceSophie

Bayesian statistics



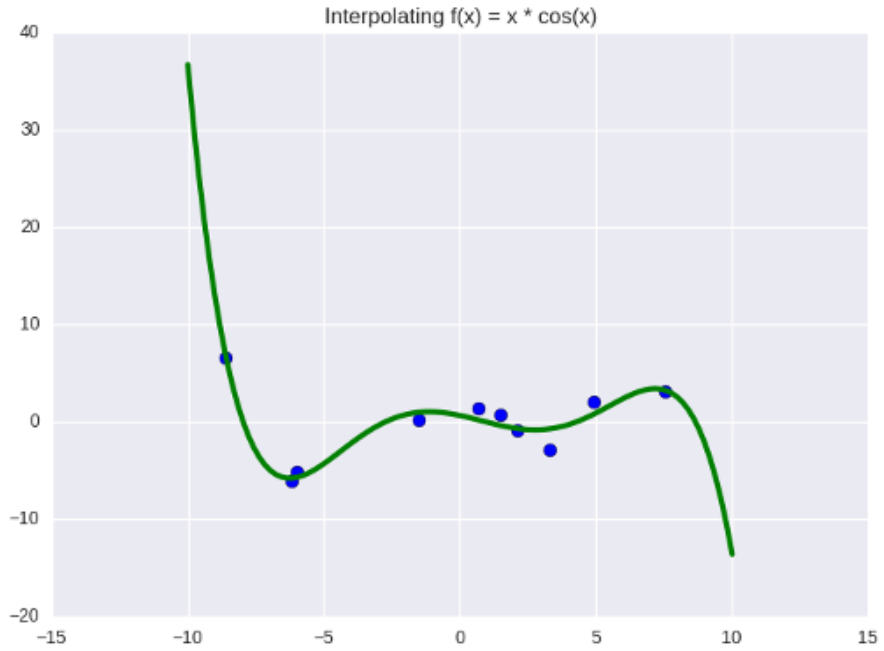




Bayesian statistics

fifth degree polynomial

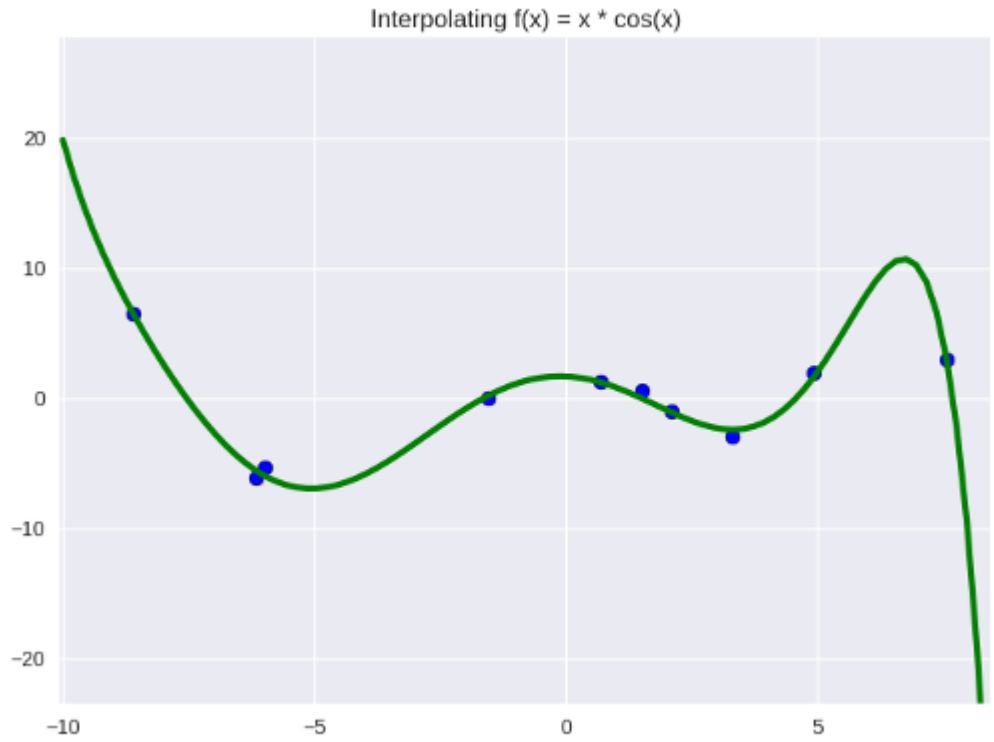
Property	Classification	Regression
Output type	Discrete (class labels)	Continuous (number)
What are trying to find?	Decision boundary	“Best fit line”
Evaluation	Accuracy	“Sum of squared error” or r^2



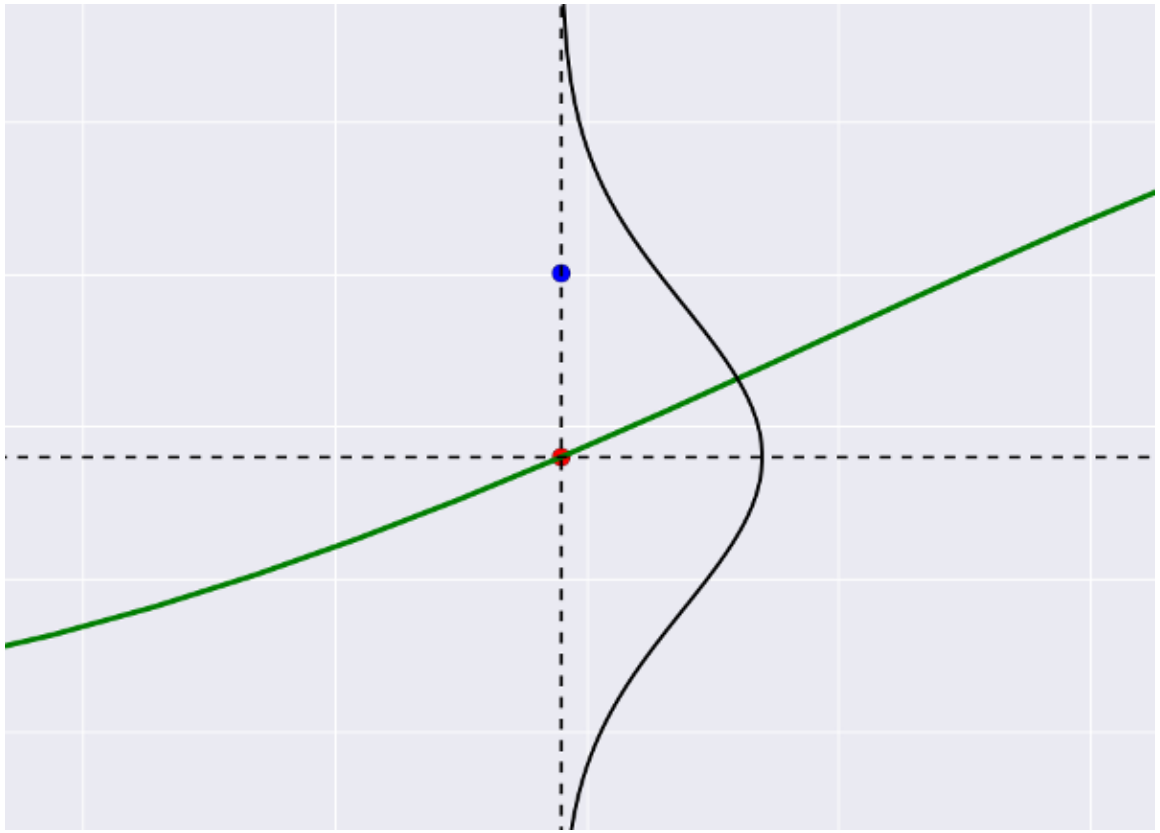
Coefficients =[0.5423, -3.2648, -16.5311, 43.3645, 25.6159, -51.2418]

$$f(x) = -51.2418x^5 + 25.6159x^4 + 43.3645x^3 - 16.5311x^2 - 3.2648x + 0.5423$$

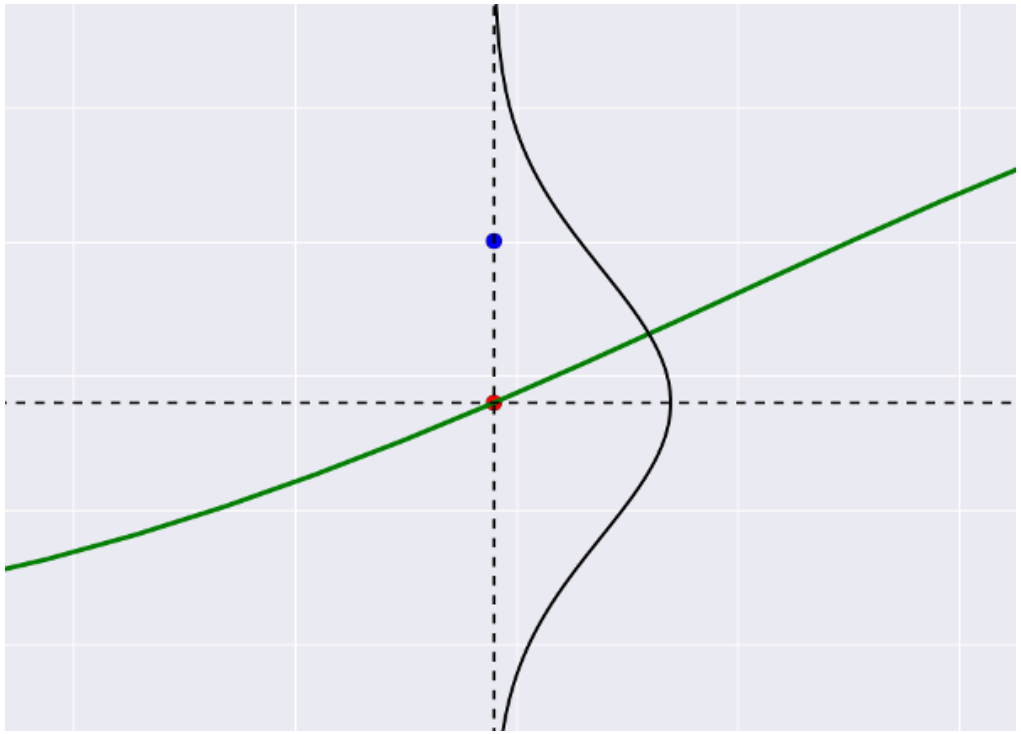
Bayesian statistics



n-1 degree polynomial



Bayesian statistics



Likelihood (правдоподобие) – $P(D \mid \mu, \sigma)$ *or* $P(D \mid \theta)$ - the probability that if the distribution of data is controlled by such parameters like θ, the result will be D.
Almost all tasks in machine learning have some models with parameters θ. The goal is according to data D to choose the parameters θ describing them in the best way.
In classical statistics it is the task of finding *maximum likelihood*

$$P(x \mid \mu_i, \sigma_i^2) = \prod_i \frac{1}{\sigma_i \sqrt{2\pi}} e^{-\frac{(x_i - \mu_i)^2}{2\sigma_i^2}} \qquad \sigma_i^2 = 1$$

$$argmax\ P(x \mid \mu_i) = argmax \prod_i e^{-(x_i - \mu_i)^2}$$

$$argmax\ P(x \mid \mu_i) = argmax \sum_i -(x_i - \mu_i)^2 \longrightarrow argmax\ P(x \mid \mu_i) = argmin \sum_i (x_i - \mu_i)^2$$

Bayesian statistics

$$p(\theta \mid D) = \frac{p(\theta)p(D \mid \theta)}{p(D)} = \frac{p(\theta)p(D \mid \theta)}{\int p(D \mid \theta)p(\theta)d\theta}$$

- $p(\theta)$ - prior probability – *априорная вероятность, распределение - reflects our initial assumption about how the distribution of parameters looks*
- $p(D \mid \theta)$ - likelihood – *правдоподобие - the value of the deviation of the dataset points from real regression curve*
- $p(\theta \mid D)$ - posterior probability – *апостериорная вероятность, распределение - "what we think about the distribution of the parameters after seeing the data"*
- $p(D) = \int p(D \mid \theta)p(\theta)d\theta$ - evidence – *вероятность данных - the general probability of obtaining data such as we have, for all possible values of the parameters* θ

Regularization

L1 and L2 regularization are two common regularization techniques that can reduce the effects of overfitting. Both of these algorithms can either work with an objective function or as a part of the backpropagation algorithm. In both cases the regularization algorithm is attached to the training algorithm by adding an additional objective.

Both of these algorithms work by adding a weight penalty to the neural network training. This penalty encourages the neural network to keep the weights to small values. Both L1 and L2 calculate this penalty differently. For gradient-descent-based algorithms, such as backpropagation, you can add this penalty calculation to the calculated gradients. For objective-function-based training, such as simulated annealing, the penalty is negatively combined with the objective score.

Ways of optimization

Regularization is a technique that reduces overfitting, which occurs when neural networks attempt to memorize training data, rather than learn from it.

L1 Regularization, also called LASSO (Least Absolute Shrinkage and Selection Operator) is should be used to create sparsity in the neural network. In other words, the L1 algorithm will push many weight connections to near 0. When a weight is near 0, the program drops it from the network. Dropping weighted connections will create a sparse neural network. Feature selection is a useful byproduct of sparse neural networks. Features are the values that the training set provides to the input neurons. Once all the weights of an input neuron reach 0, the neural network training determines that the feature is unnecessary. If your data set has a large number of input features that may not be needed, L1 regularization can help the neural network detect and ignore unnecessary features.

Regularization

- L1 is implemented by adding the following error to the objective to minimize:

$$E_1 = \lambda \sum_w |w|$$

- You should use Tikhonov/Ridge/L2 regularization when you are less concerned about creating a space network and are more concerned about low weight values. The lower weight values will typically lead to less overfitting:

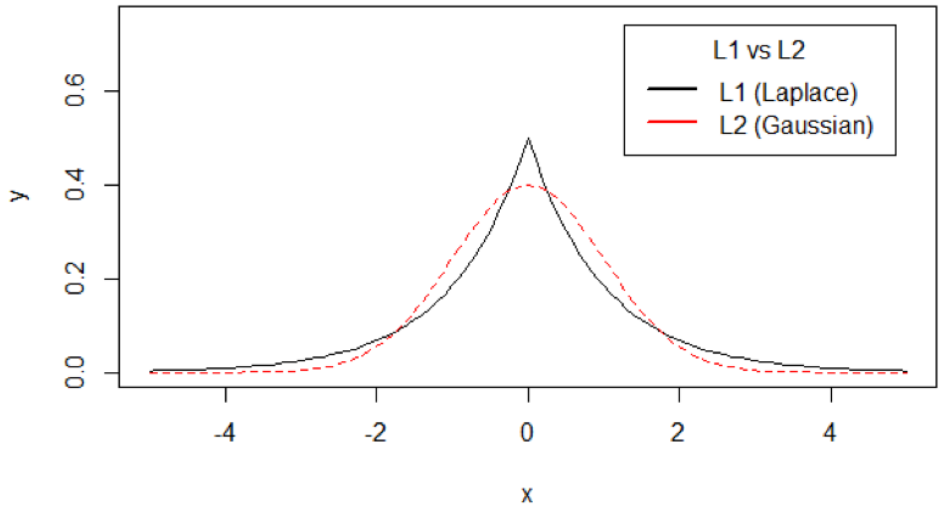
$$E_2 = \lambda \sum_w w^2$$

Like the L1 algorithm, the λ value determines how important the L2 objective is compared to the neural network's error. Typical L2 values are below 0.1 (10%). The main calculation performed by L2 is the summing of the squares of all of the weights. The bias values are not summed.

Regularization

As you can see, L1 algorithm is more tolerant of weights further from 0, whereas the L2 algorithm is less tolerant. You also need to note that both L1 and L2 count their penalties based only on weights; they do not count penalties on bias values.

Tensor flow allows L1/L2 to be directly added to your network.

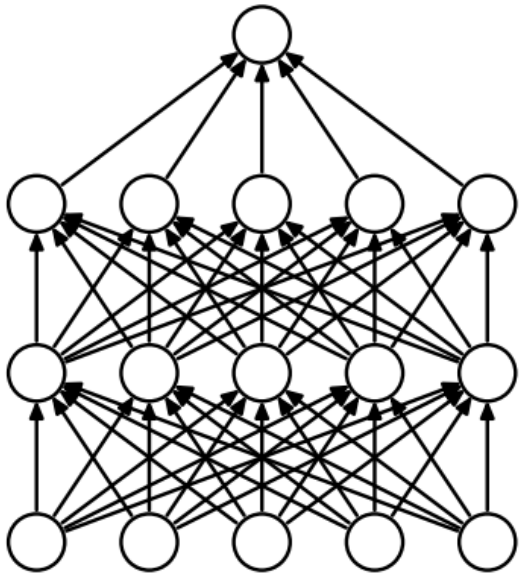


Regularization

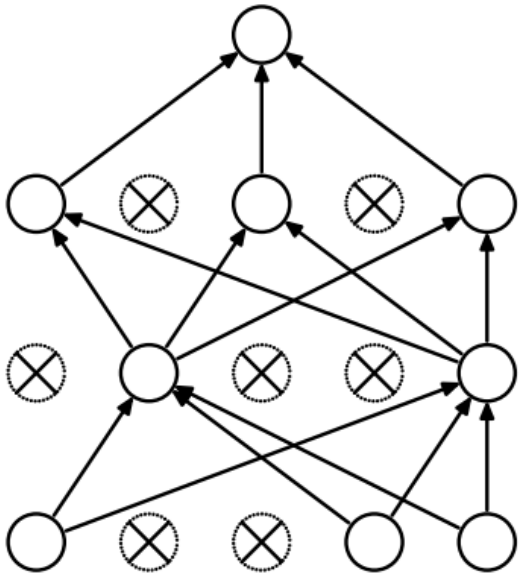
- kernel_regularizer — for matrix of weights for a layer;
- bias_regularizer — for bias vector;
- activity_regularizer — for outputs vector.

```
model.add(Dense(256, input_dim=32,
        kernel_regularizer=regularizers.l1(0.001),
        bias_regularizer=regularizers.l2(0.1),
        activity_regularizer=regularizers.l2(0.01)))
```

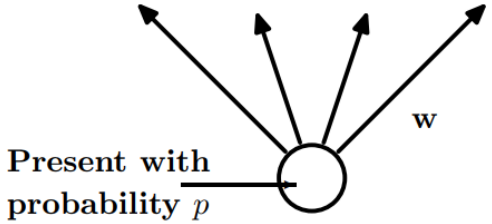
Dropout Regularization



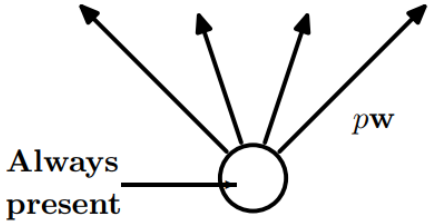
(a) Standard Neural Net



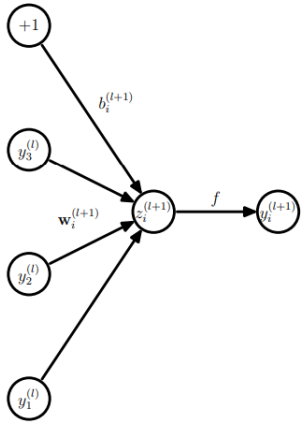
(b) After applying dropout.



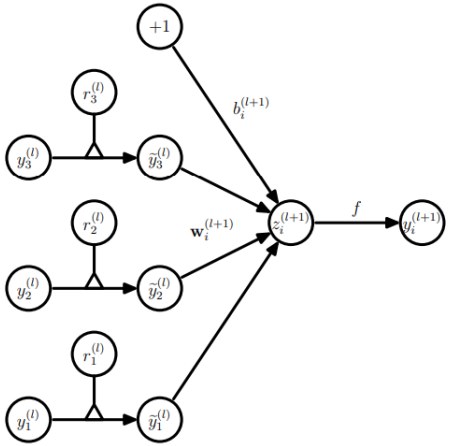
(a) At training time



(b) At test time



(a) Standard network



(b) Dropout network

Dropout Regularization

Most neural network frameworks implement dropout as a separate layer. Dropout layers function as a regular, densely connected neural network layer. The only difference is that the dropout layers will periodically drop some of their neurons during training. You can use dropout layers on regular feedforward neural networks.

The usual hyper-parameters for a dropout layer are the following:

- Neuron Count
- Activation Function
- Dropout Probability

The neuron count and activation function hyper-parameters work exactly the same way as their corresponding parameters in the dense layer type mentioned previously. The neuron count simply specifies the number of neurons in the dropout layer. The dropout probability indicates the likelihood of a neuron dropping out during the training iteration. Just as it does for a dense layer, the program specifies an activation function for the dropout layer. A certain percentage neurons we be masked during each training step. All neurons return after training is complete. To make use of dropout in Keras use the Dropout layer and specify a dropout probability. This is the percent of neurons to be dropped. Typically this is a low value, such as 0.1: `model.add(Dropout(0.1))`

Weight Initialization

Why is initialization so important? What is the difference between the initialization schemes?

- All zero initialization –**not** a good idea
- Random
- Xavier / Glorot - activation function is a Tanh
- He - activation function is a ReLU

Weight Initialization

We will use a model with **5** hidden layers with **100** units each, and a single unit output layer as in a typical binary classification task (that is, using a **sigmoid** as activation function and binary **cross-entropy** as loss) and will refer to this model as the BLOCK model, as it has consecutive layers of same size.

Layer (type)	Output Shape	Param #
h1 (Dense)	(None, 100)	1100
h2 (Dense)	(None, 100)	10100
h3 (Dense)	(None, 100)	10100
h4 (Dense)	(None, 100)	10100
h5 (Dense)	(None, 100)	10100
o (Dense)	(None, 1)	101

Total params: 41,601
Trainable params: 41,601
Non-trainable params: 0

Weight Initialization

Inputs

The inputs are **1,000 random points** drawn from a **10-dimensional ball** such that the samples have **zero mean** and **unit standard deviation**.

In this dataset, points situated *within half of the radius* of the ball are labeled as *negative cases* (0), while the remaining points are labeled *positive cases* (1).

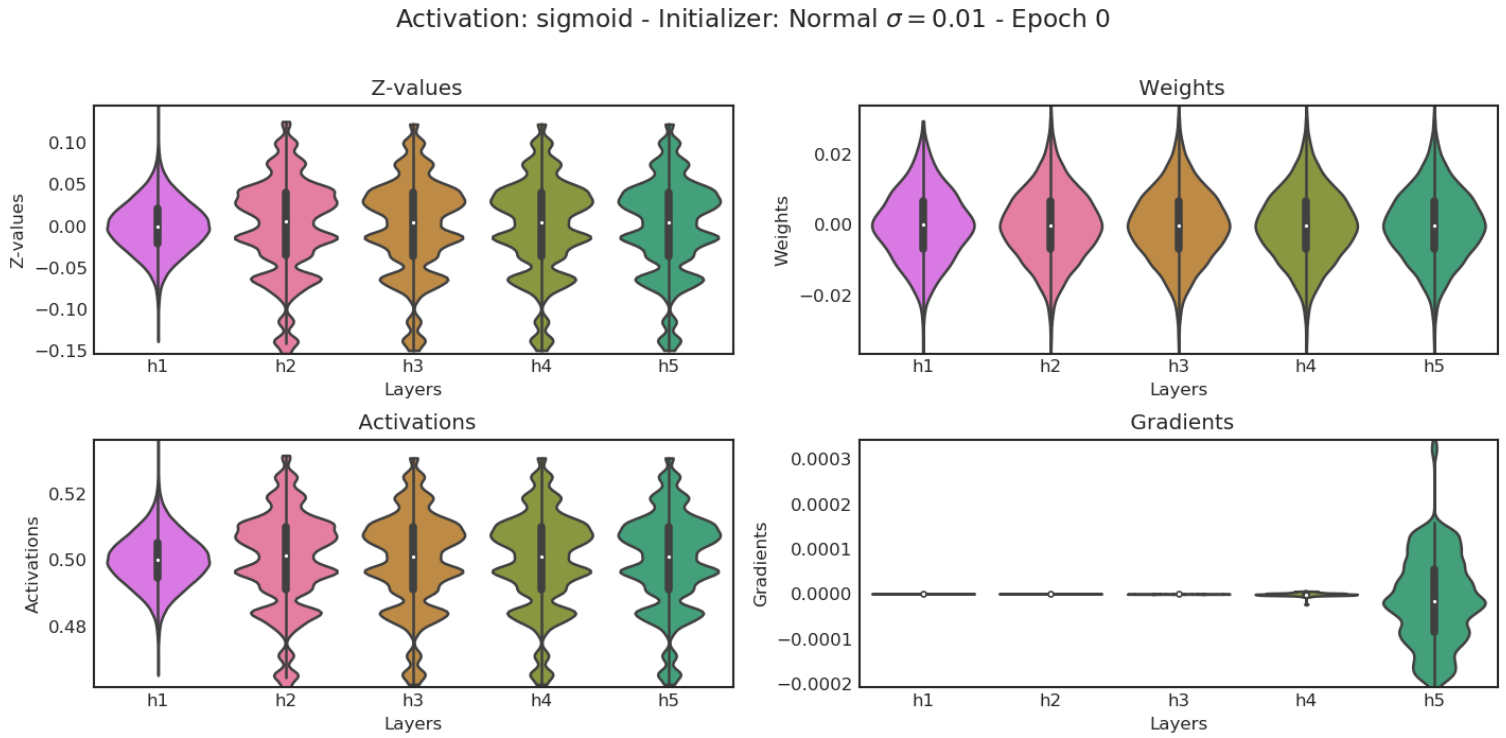
```
from deepreplay.datasets.ball import load_data

X, y = load_data(n_dims=10)
```

Weight Initialization

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Input, Dense, Activation

from deepreplay.datasets.ball import load_data

X, y = load_data(n_dims=10)

def build_model(n_layers, input_dim, units, activation, initializer):
    if isinstance(units, list):
        assert len(units) == n_layers
    else:
        units = [units] * n_layers

    model = Sequential()
    # Adds first hidden layer with input_dim parameter
    model.add(Dense(units=units[0],
                    input_dim=input_dim,
                    activation=activation,
                    kernel_initializer=initializer,
                    name='h1'))

    # Adds remaining hidden layers
    for i in range(2, n_layers + 1):
        model.add(Dense(units=units[i - 1],
                        activation=activation,
                        kernel_initializer=initializer,
                        name='h{}'.format(i)))

    # Adds output layer
    model.add(Dense(units=1, activation='sigmoid', kernel_initializer=initializer, name='o'))
    # Compiles the model
    model.compile(loss='binary_crossentropy', optimizer='sgd', metrics=['acc'])
    return model
```

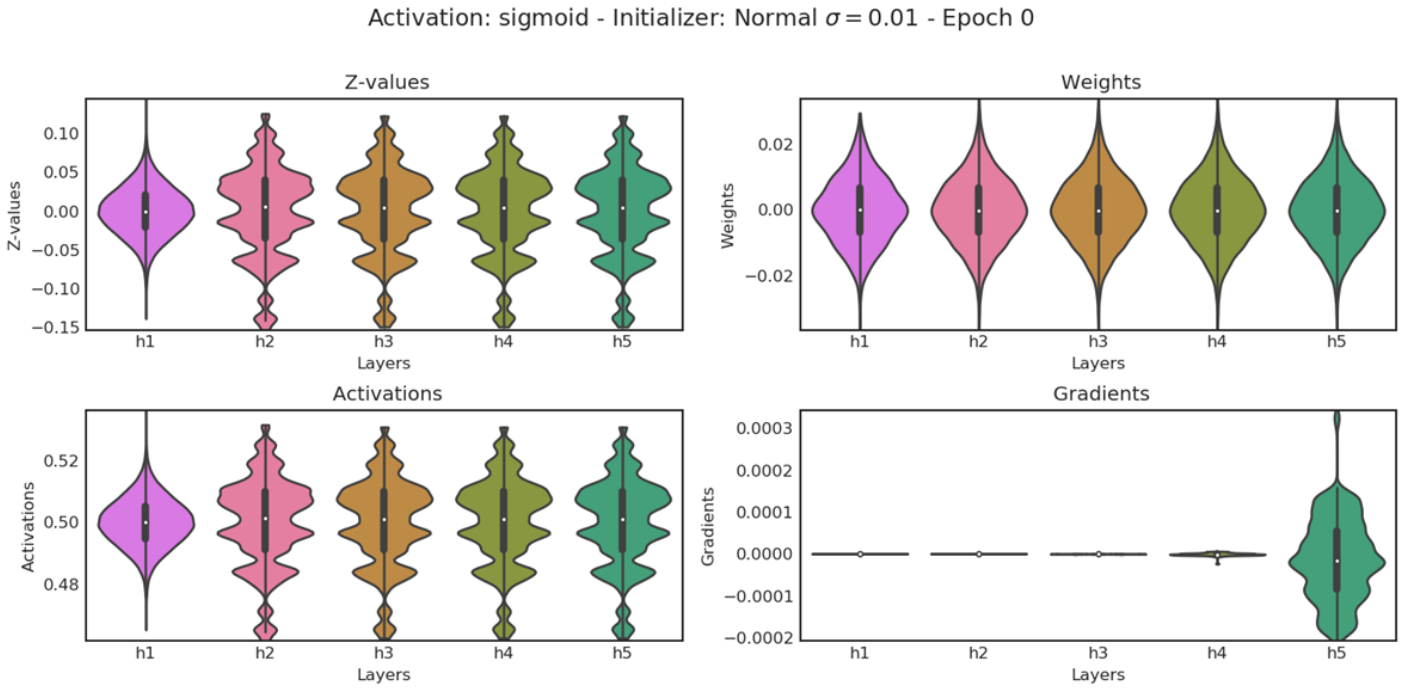
Weight Initialization

At the beginning, there was a sigmoid activation function and randomly initialized weights. And training was hard, convergence was slow, and results were not good. The usual procedure was to draw random values from a Normal distribution (zero mean, unit standard deviation) and multiply them by a small number, say 0.01. The result would be a set of weights with a standard deviation of approximately 0.01.



- Both **Z-values** (remember, these are the outputs *before* applying the activation function)
and **Activations** are within a **narrow range;**
- **Gradients** are pretty much **zero;**
- And what is the deal with the **weird distributions** on the left column

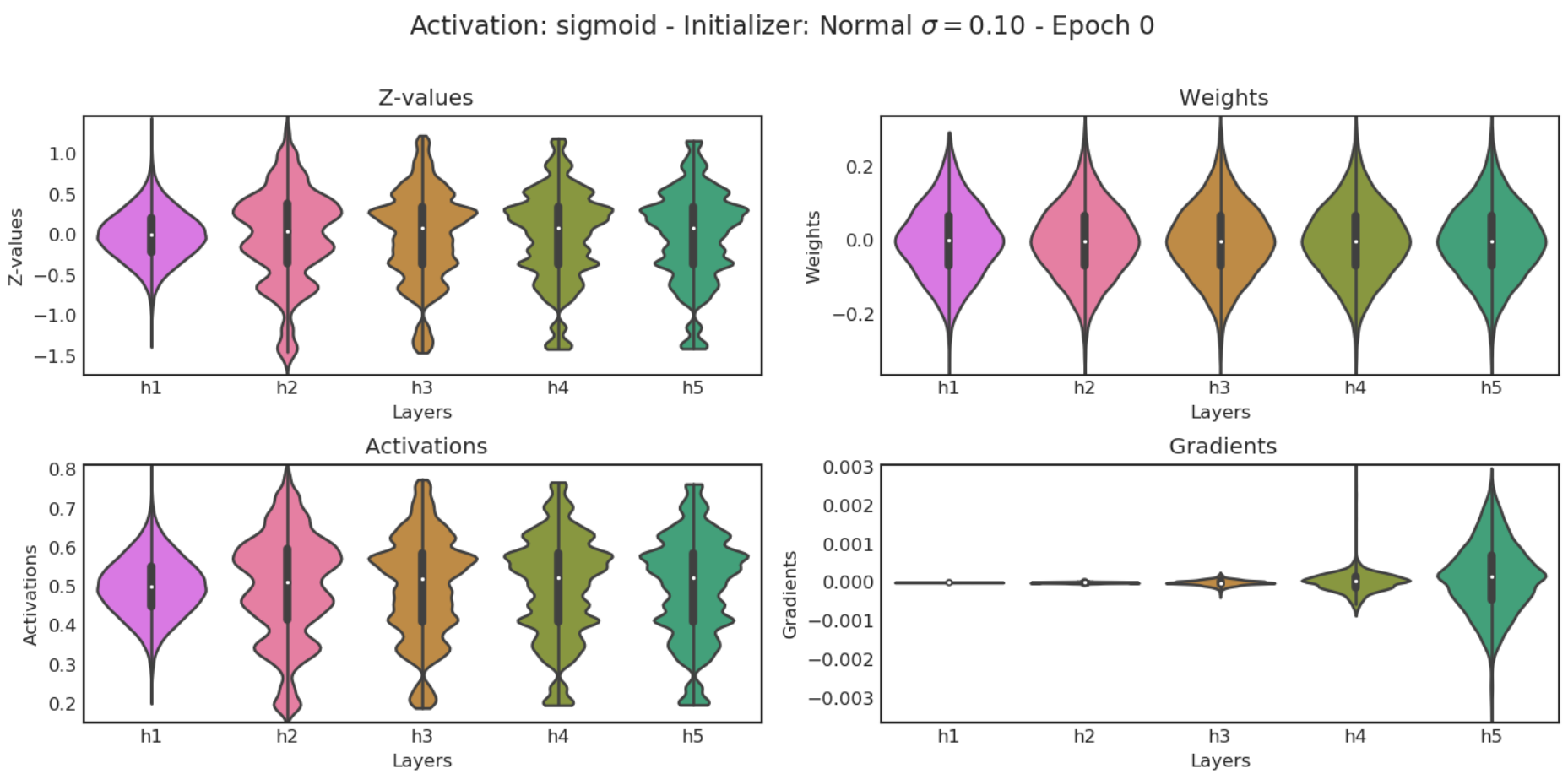
Weight Initialization



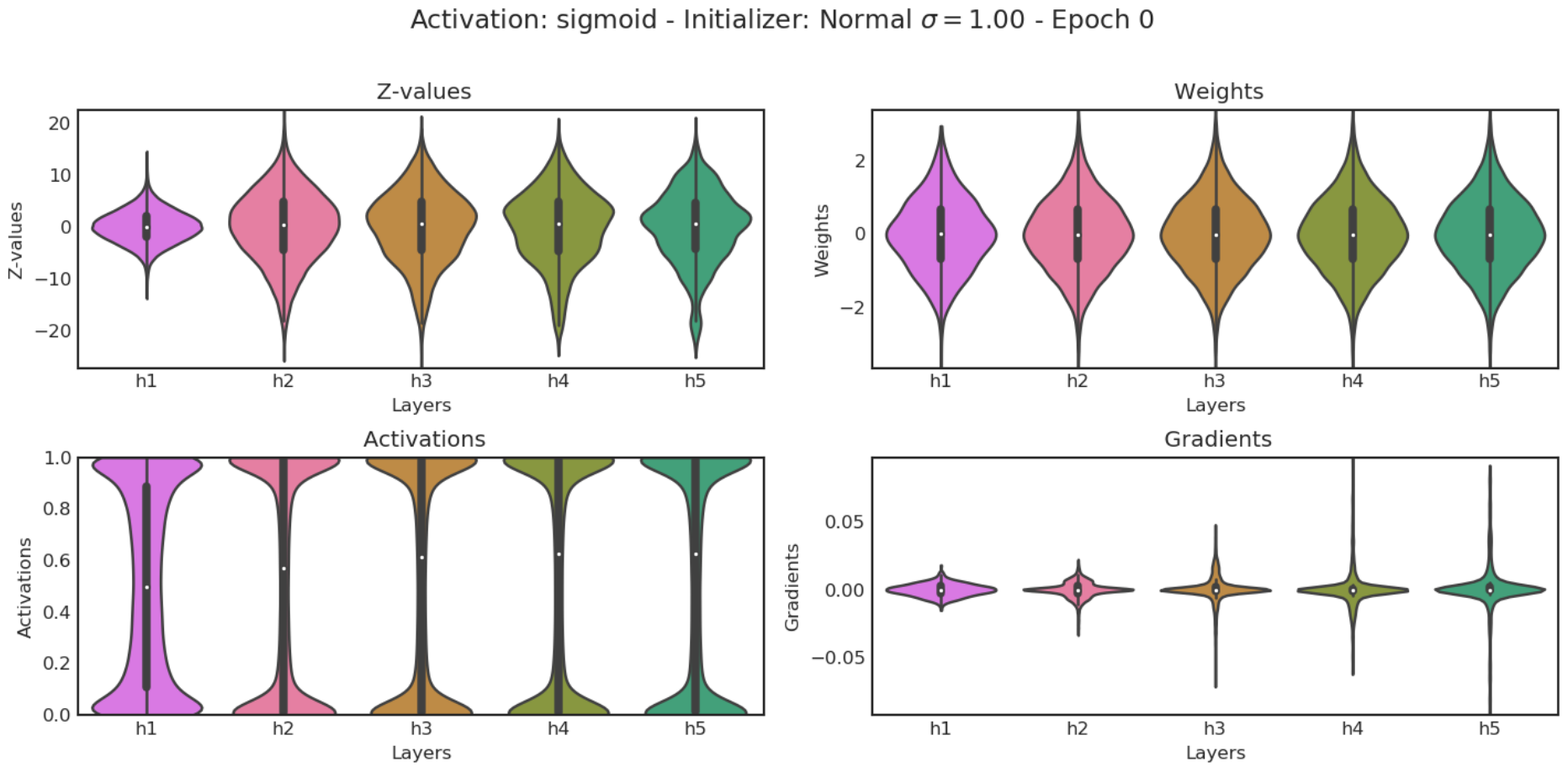
- Weights were initialized using the naive scheme (upper right subplot)
- 1,000 samples were used in a forward pass through the network and generated Z-values (upper left subplot) and Activations (lower left subplot) for all the layers (output layer was not included in the plot)
- The loss was computed against the true labels and backpropagated through the network, generating the gradients for all layers (lower right subplot)

What does it mean? It means that our network is borderline useless, as it cannot LEARN anything (that is, update its weights to perform the proposed classification task) in a reasonable amount of time.
This is an extreme case of vanishing gradients

Weight Initialization



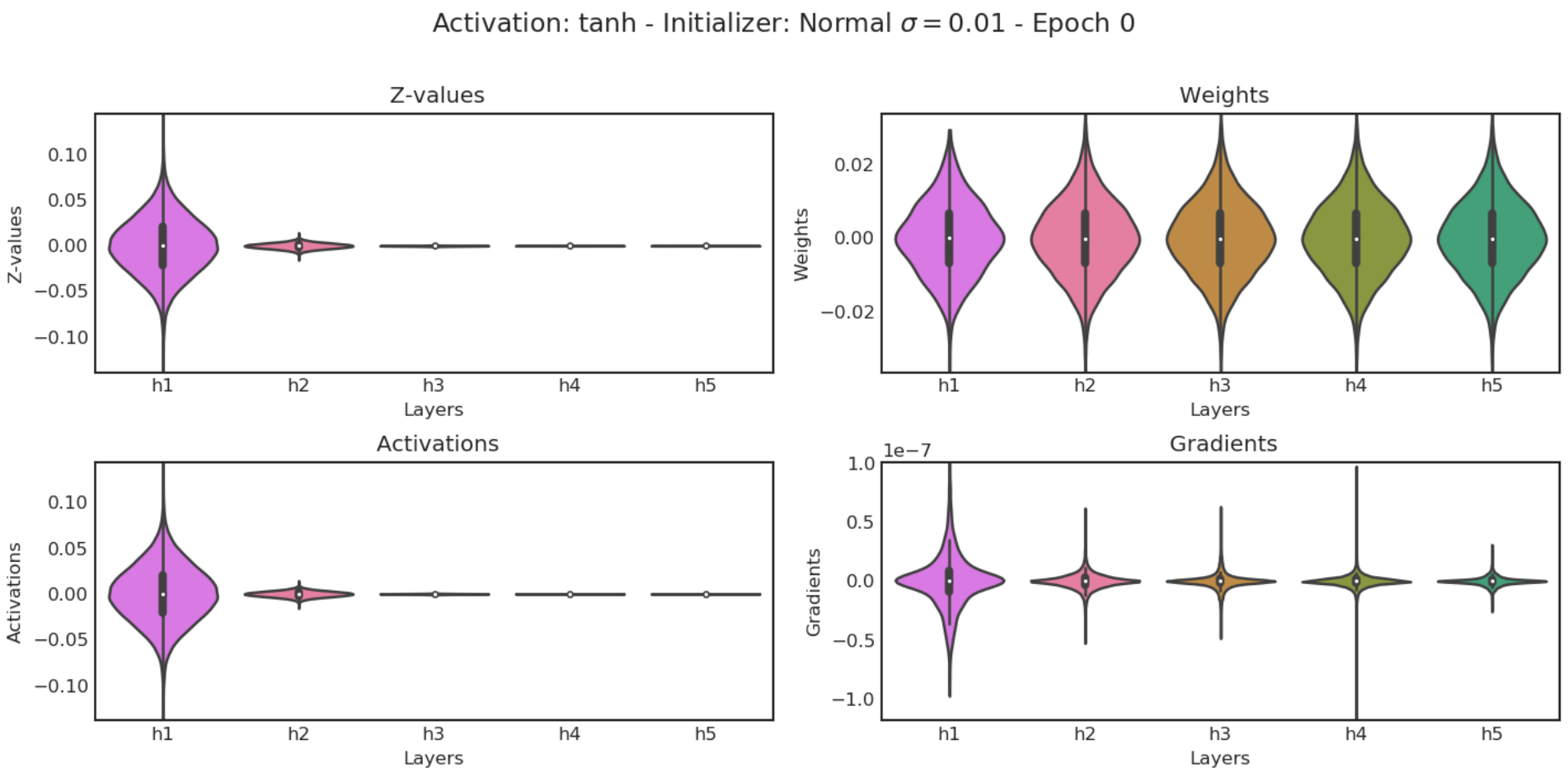
Weight Initialization



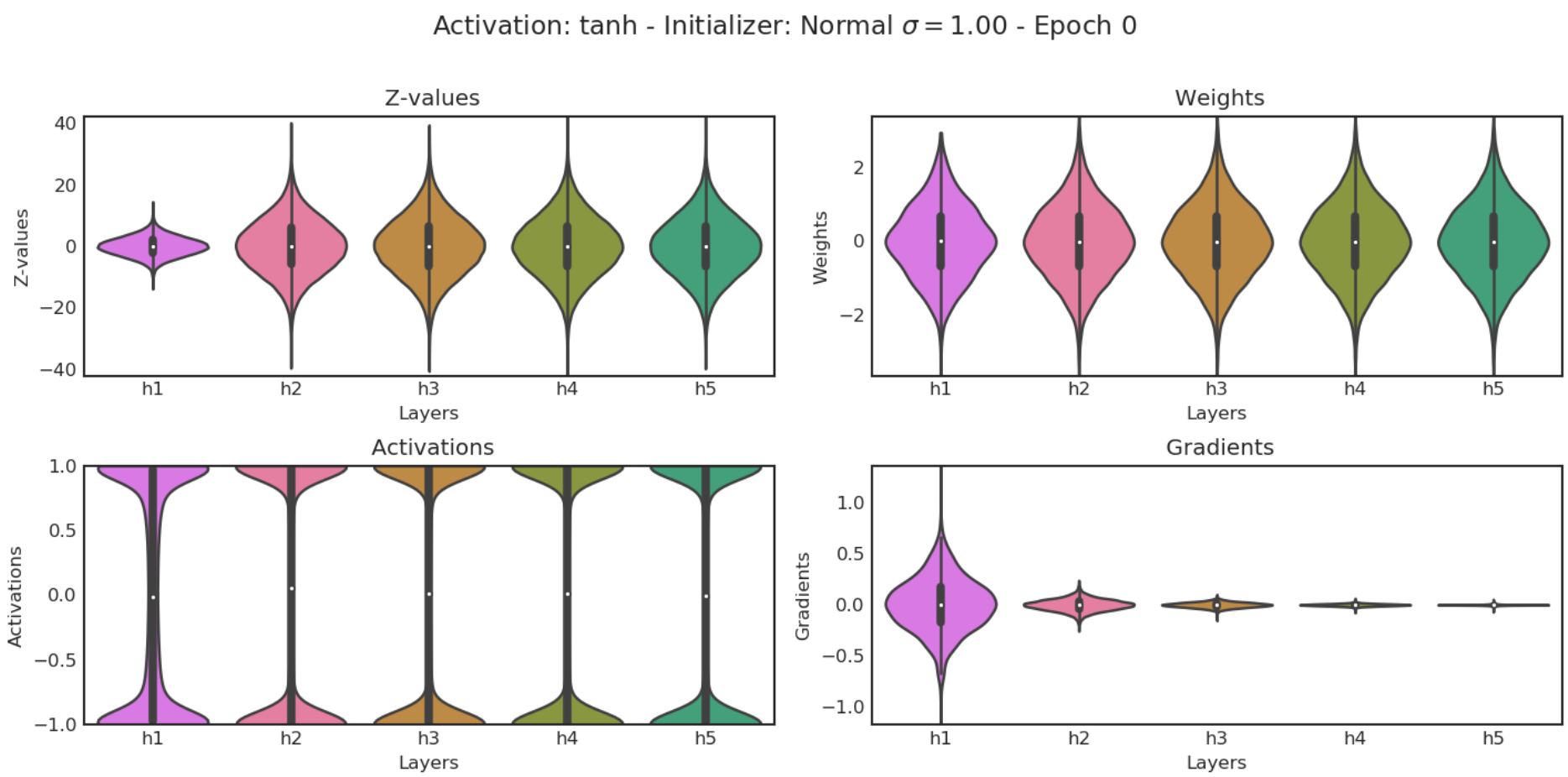
Z-values now exhibit a *too wide range*, forcing the **Activations** pretty much into *binary mode*

Sigmoid activation function has this fundamental issue of being centered around 0.5

Weight Initialization

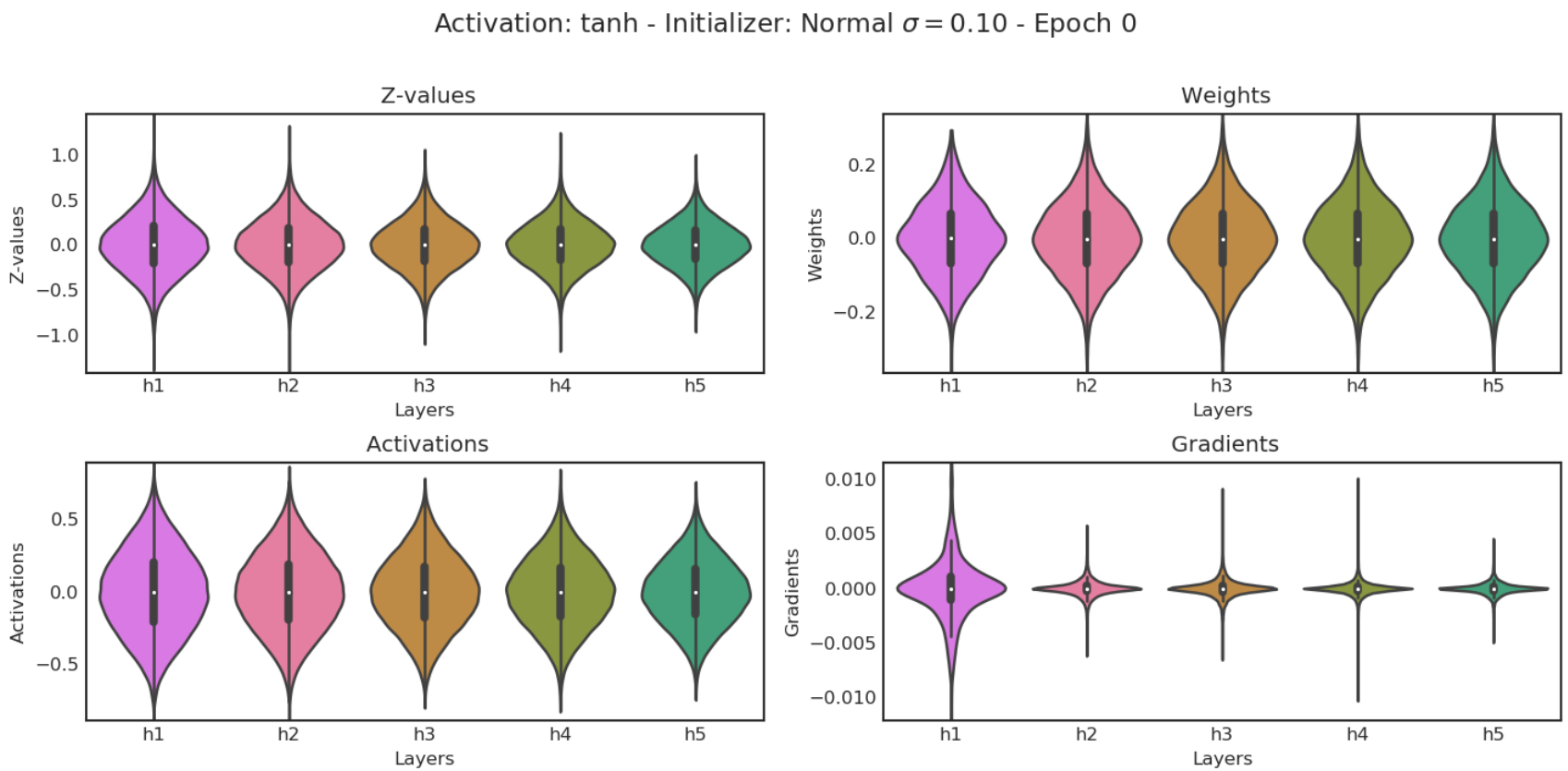


Weight Initialization



We can observe the **exploding** gradients problem, for a change. See how gradient values grow bigger and bigger as we backpropagate from the last to the first hidden layer? Besides, just like it happened when using a Sigmoid activation function, Z-values have a too wide range and Activations are collapsing into either zero or one for the most part.

Weight Initialization



•**Gradients** are reasonably **similar along all the layers** (and within a *decent scale*— about *20x smaller* than the weights)

• **Z-values** are within a *decent range* (-1, 1) and are reasonably **similar along all the layers** (though some shrinkage is noticeable)

•Third, **Activations** have *not collapsed into binary mode*, and are reasonably **similar along all the layers** (again, with *some* shrinkage)

Xavier / Glorot Initialization Scheme

Glorot and Bengio devised an initialization scheme that tries to keep all the winning features listed , that is, gradients, Z-values and Activations similar along all the layers. Another way of putting it: keeping variance similar along all the layers.

Let's say we have ***x*** values (either inputs or activation values from a previous layer) and ***W*** weights. The variance of the product of two independent variables is given by the formula below:

$$Var(w_i x_i) = \mathbb{E}\left[x_i^2 w_i^2\right] - (\mathbb{E}[x_i w_i])^2 = \mathbb{E}[x_i]^2 Var(w_i) + \mathbb{E}[w_i]^2 Var(x_i) + Var(w_i)Var(x_i)$$

Then, let's assume both ***x*** and ***W*** have zero mean. The expression above turns into a simple product of both variances of *x* and *W:*

$$Var(w_i x_i) = Var(w_i)Var(x_i)$$

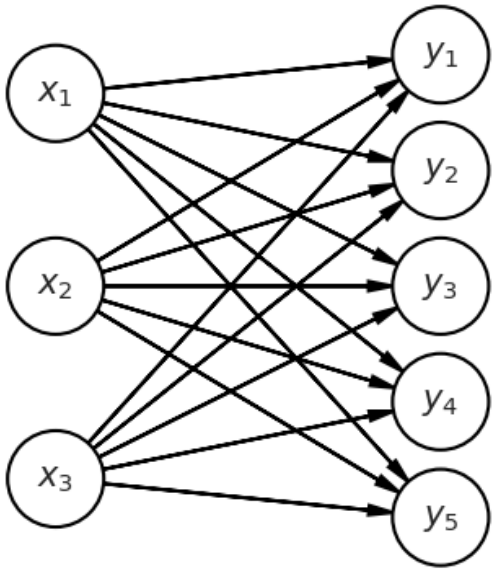
Xavier / Glorot Initialization Scheme

- The inputs should have zero mean for this to hold in the first layer, so always scale and center your inputs
- Sigmoid activation function poses a problem for this, as the activation values will have a mean of 0.5, NOT zero. This point only makes sense to stick with Tanh.

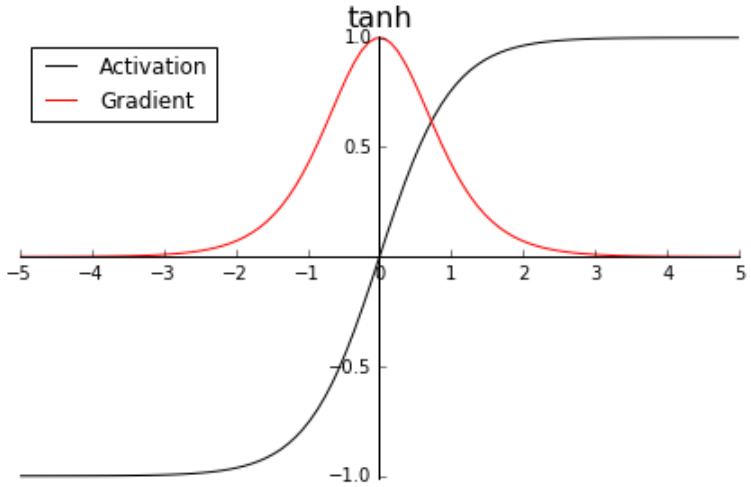
Xavier / Glorot Initialization Scheme

This example is made of two hidden layers, X and Y, fully connected. We only care about the weights connecting these two layers and, guess what, the variances of both activations and gradients.

For the activations, we need to go through a forward pass in the network. For the gradients, we need to backpropagate. And, for the sake of keeping the math simple, we will assume that the activation function is linear (instead of a Tanh) in the equations, meaning that the activation values are the same as Z-values.



Although this may seem a bit of a stretch, Figure shows us that a Tanh is roughly linear in the interval [-1, 1], so the results should hold, at least in this interval



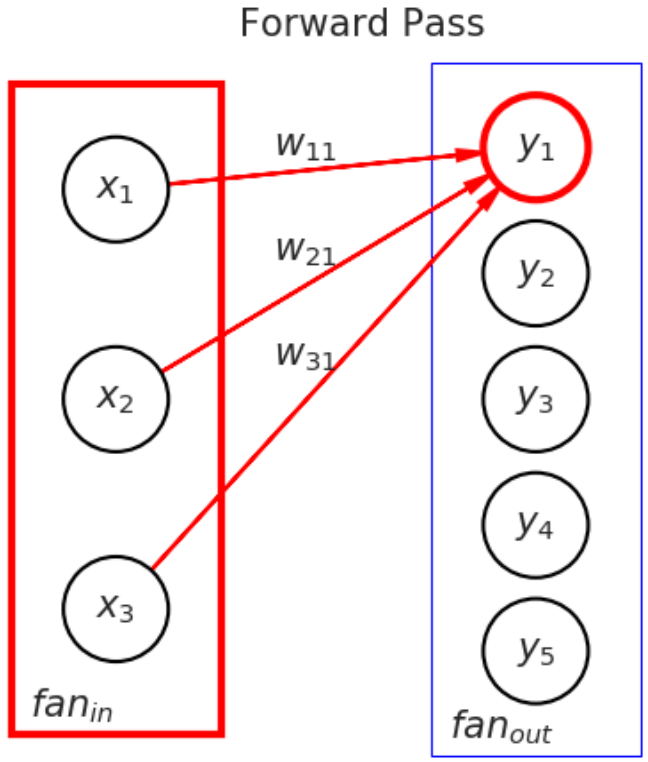
Xavier / Glorot Initialization Scheme

So, instead of using a vectorized approach, we'll be singling out **one** unit, *y1*, and work the math for it.

- three units in the previous layer (fan-in)
- weights (*w11, w21 and w31*)
- the unit we want to compute variance for, *y1*

Assuming that ***x*** and ***W*** are independent and identically distributed, we can work out some simple math for the variance of *y1*

1. $$y_1 = x_1 W_{11} + x_2 W_{21} + x_3 W_{31}$$
2. $$Var(y_1) = Var(x_1 W_{11} + x_2 W_{21} + x_3 + W_{31})$$
3. $$Var(y_1) = Var(x_1 W_{11}) + Var(x_2 W_{21}) + Var(x_3 W_{31})$$
4. $$Var(y_1) = \sum_{i=1}^{3} Var(x_i) Var(W_{i1})$$
5. $$Var(y_1) = n_i Var(x) Var(W_{i1})$$



Xavier / Glorot Initialization Scheme

Our goal is to keep **variance** similar along all the layers. In other words, we should aim for making the variance of ***x*** the same as the variance of ***y***.

For our single unit, **y1**, this can be accomplished by choosing the variance of its connecting weights to be

$$6. \qquad Var(y_1) = Var(x) \iff Var(W_{i1}) = \frac{1}{n_i} = \frac{1}{fan_{in}}$$

$$Var(W) = \frac{1}{fan_{in}}; \sigma(W) = \sqrt{\frac{1}{fan_{in}}}$$

By the way, this is the **variance** to be used if we're drawing random **weights** from a **Normal distribution** What if we want to use a **Uniform distribution**? We just have to compute the (symmetric) lower and upper **limits**, as shown below

$$Var(Uniform(-limit, limit)) = \frac{limit^2}{3}$$

$$Var(W) = \frac{1}{fan_{in}} \iff limit = \sqrt{\frac{3}{fan_{in}}}$$

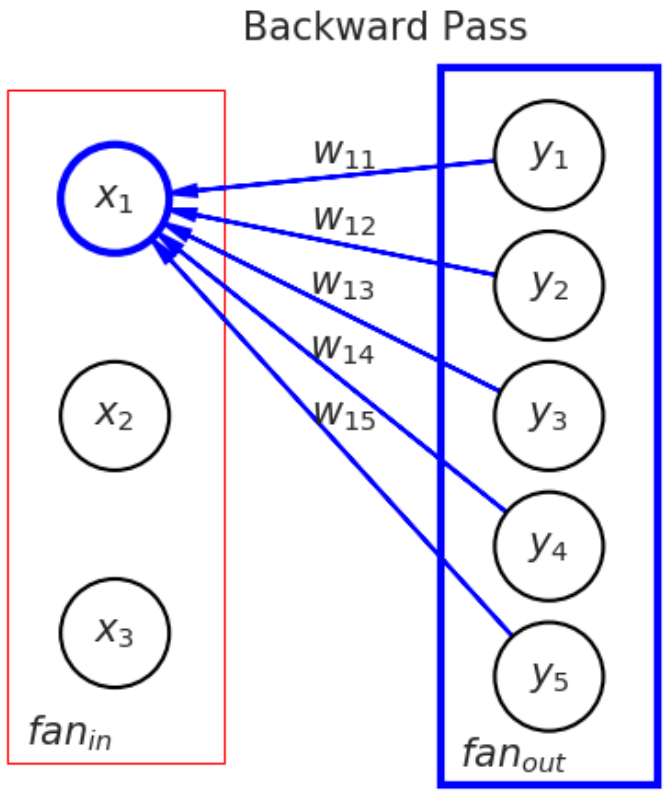
Xavier / Glorot Initialization Scheme

Again, let's single out **one** unit, ***x1***, for the backward pass.

- five units in the **following layer**(**fan-out**)
- **weights** (*w11*, *w12*, *w13*, *w14* and *w15*)
- the unit we want to compute the **variance** of the **gradients** with respect to it, ***x1***

We will make the same assumptions and follow the same steps as in the forward pass. For the variance of the gradients with respect to x1, we can work out the math the same way

1. $$\Delta_{x_1} = \Delta_{y_1} W_{11} + \Delta_{y_2} W_{12} + \Delta_{y_3} W_{13} + \Delta_{y_4} W_{14} + \Delta_{y_5} W_{15}$$
2. $$Var(\Delta_{x_1}) = Var(\Delta_{y_1} W_{11} + \Delta_{y_2} W_{12} + \Delta_{y_3} W_{13} + \Delta_{y_4} W_{14} + \Delta_{y_5} W_{15})$$
3. $$Var(\Delta_{x_1}) = Var(\Delta_{y_1} W_{11}) + Var(\Delta_{y_2} W_{12}) + Var(\Delta_{y_3} W_{13}) + Var(\Delta_{y_4} W_{14}) + Var(\Delta_{y_5} W_{15})$$
4. $$Var(\Delta_{x_1}) = \sum_{j=1}^{5} Var(\Delta_{y_j}) Var(W_{1j})$$
5. $$Var(\Delta_{x_1}) = n_j Var(\Delta_y) Var(W_{1j})$$



Xavier / Glorot Initialization Scheme

To keep **the variance of gradients similar along all the layers**, we find the needed **variance** of its connecting **weights** to be:

6. $$Var(\Delta_{x_1}) = Var(\Delta_y) \iff Var(W_{1j}) = \frac{1}{n_j} = \frac{1}{fan_{out}}$$

The inverse of the "**fan in**" gives us the desired **variance** of the **weights** for the **forward pass**, while the inverse of the "**fan out**" gives us the desired **variance** of the (***same***!) **weights** for the **backpropagation**.

What if "**fan in**" and "**fan out**" have **very** different values?

Xavier / Glorot Initialization Scheme

- We finally arrive at the expression for the variance of the weights, as in Glorot and Bengio, to be used with a **Normal distribution**

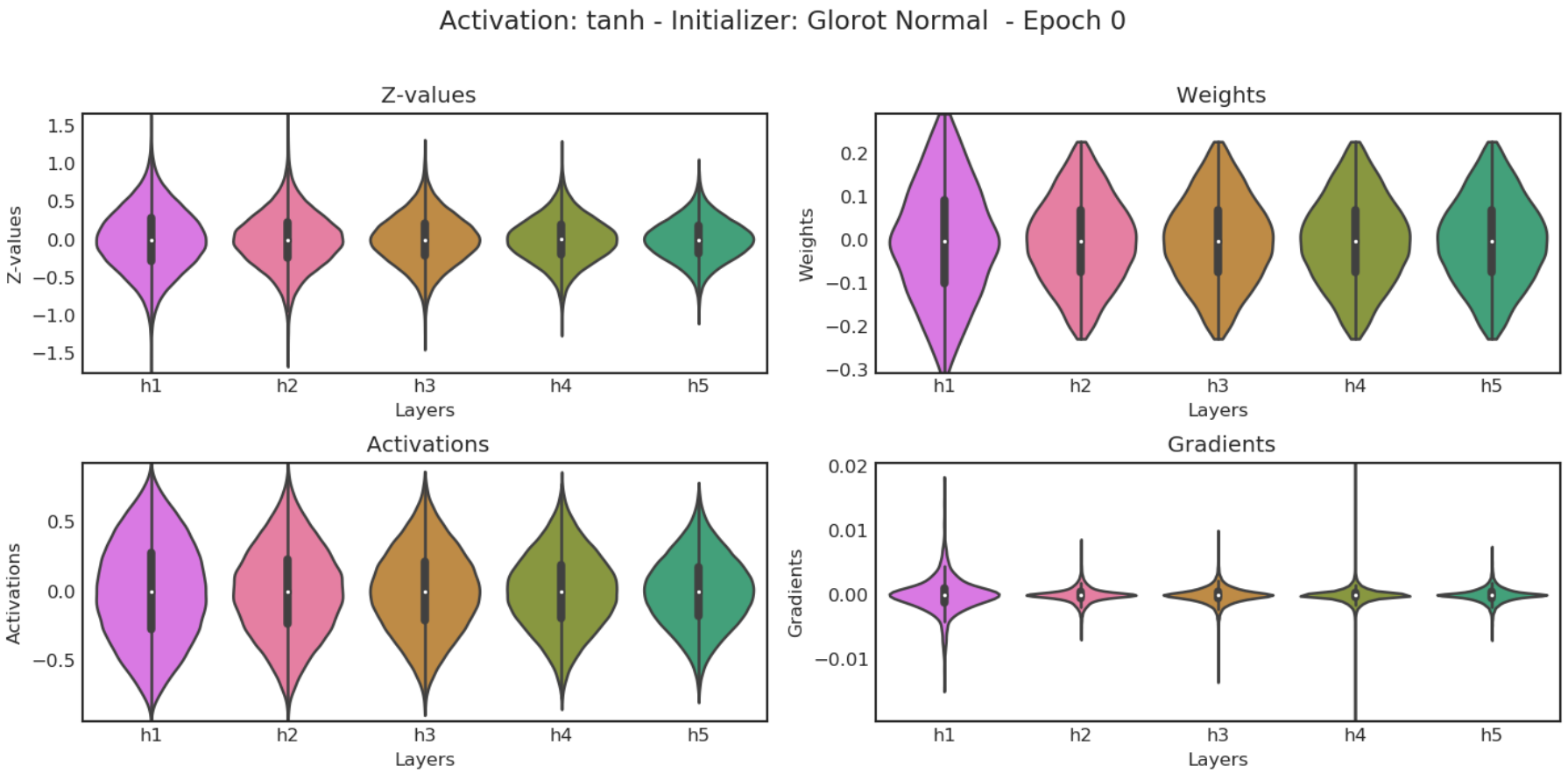
$$Var(W) = \frac{1}{\frac{fan_{in}+fan_{out}}{2}} = \frac{2}{fan_{in}+fan_{out}}; \sigma(W) = \sqrt{\frac{2}{fan_{in}+fan_{out}}}$$

- And, for the **Uniform distribution**, we compute the limits accordingly:

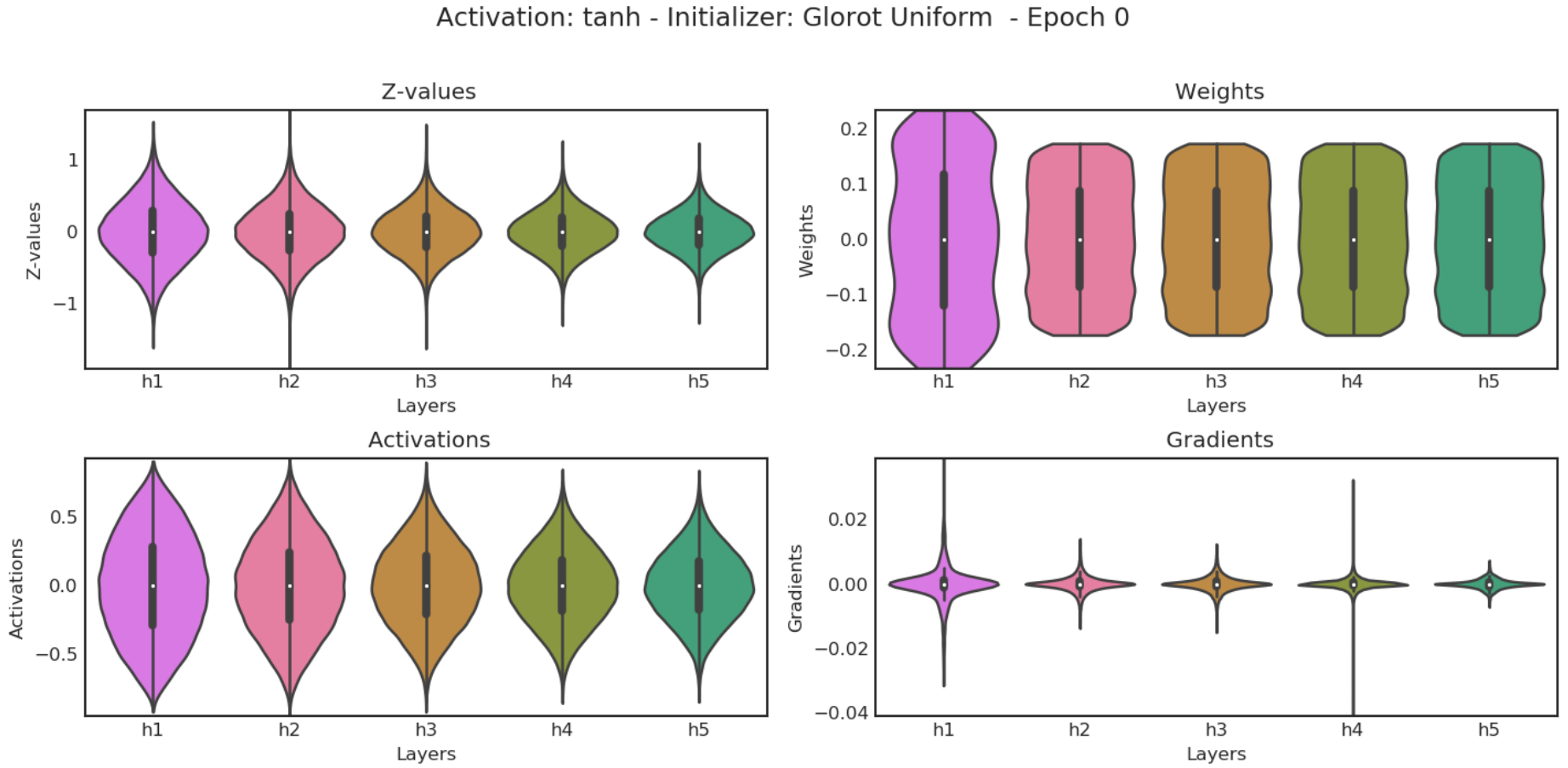
$$Var(Uniform(-limit, limit)) = \frac{limit^2}{3}$$

$$Var(W) = \frac{2}{fan_{in}+fan_{out}} \iff limit = \sqrt{\frac{6}{fan_{in}+fan_{out}}}$$

Xavier / Glorot Initialization Scheme



Xavier / Glorot Initialization Scheme

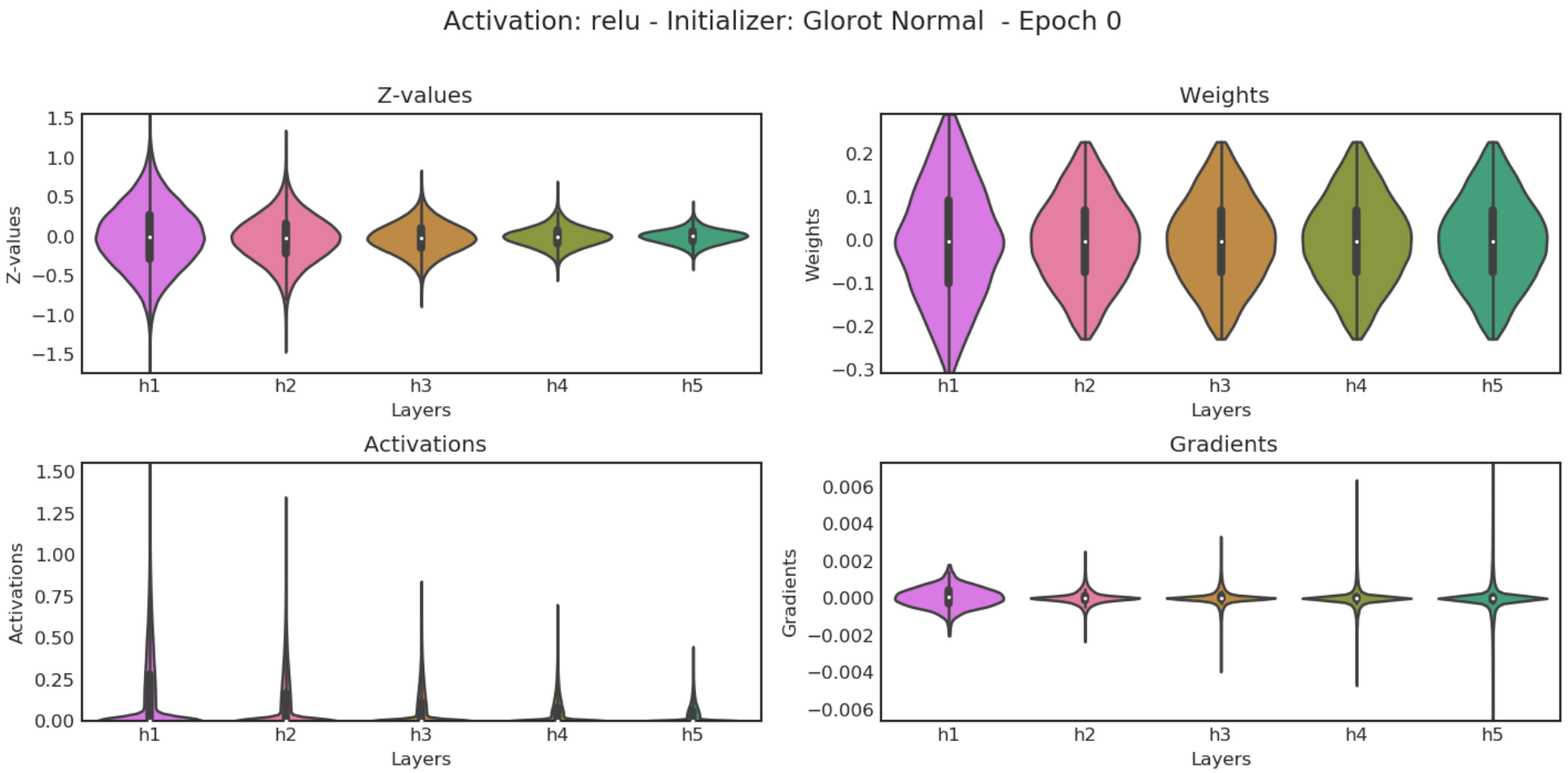


Xavier initialization

One very interesting and practically important conclusion was made in the work of Xavier [*Glorot X., Bengio Y. Understanding the Difficulty of Training Deep Feedforward Neural Networks*]. It turns out that logistic sigmoid σ is very unfortunate activation function for deep neural networks. Experiments with deep networks based on logistic sigmoid showed that the last level of the network is very quickly saturated, and getting out of this situation of saturation of a deep network is very difficult.

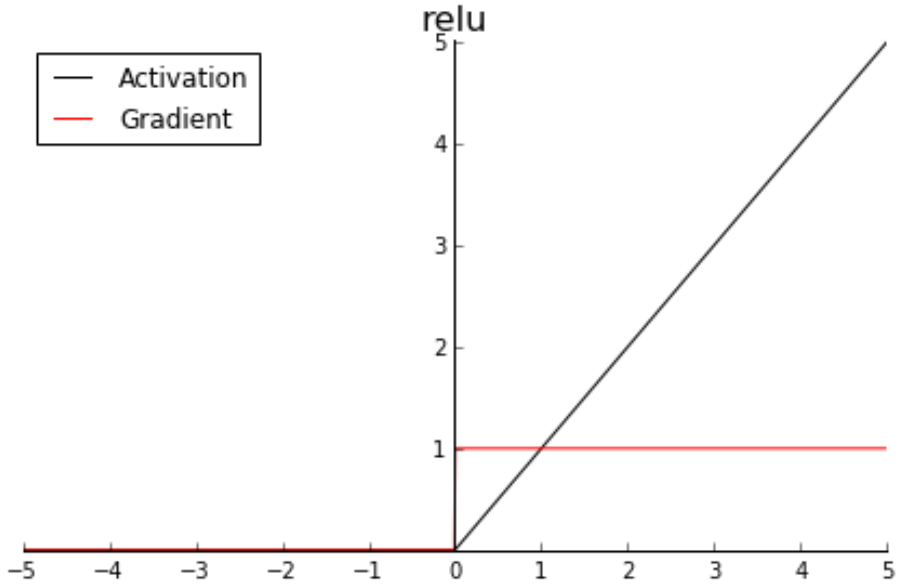
Although we did get a great way to initialize for the weights of such neurons as above, this is our story about the initialization of weights does not end there. The fact is that all of the above applies exclusively to symmetric activation functions! This means that Xavier initialization will work fine for logistic sigmoid or hyperbolic tangent, but in modern convolutional (and not only) architectures, asymmetric activation functions are widely used, especially often ReLU.

He Initialization Scheme



He Initialization Scheme

There is only one tiny adjustment we need to make… multiply the variance of the weights by 2. The reason is also pretty straightforward: the **ReLU** turns **half** of the **Z-values** (the negative ones) into **zeros**, effectively removing about **half**of the **variance**. So, we need to double the variance of the weights to compensate for it.



Since we know that the **Glorot initialization scheme** preserves **variance** (**1**), how to **compensate** for the **variance halving** effect of the **ReLU** (**2**)? The result (**3**), as expected, is **doubling the variance**.

1. $$Var(y) = Var(x)Var_{Glorot}(W)$$
2. $$Var(y) = ReLU(Var(x)Var_{He}(W)) = \frac{Var(x)Var_{He}(W)}{2}$$
3. $$\frac{Var(x)Var_{He}(W)}{2} = Var(x)Var_{Glorot}(W) \iff Var_{He}(W) = 2 * Var_{Glorot}(W)$$

So, the expression for the **variance** of the **weights**, as in *He* et al., to be used with a **Normal distribution** is

$$Var(W) = \frac{2}{fan_{in}}; \sigma(W) = \sqrt{\frac{2}{fan_{in}}}$$

He Initialization Scheme

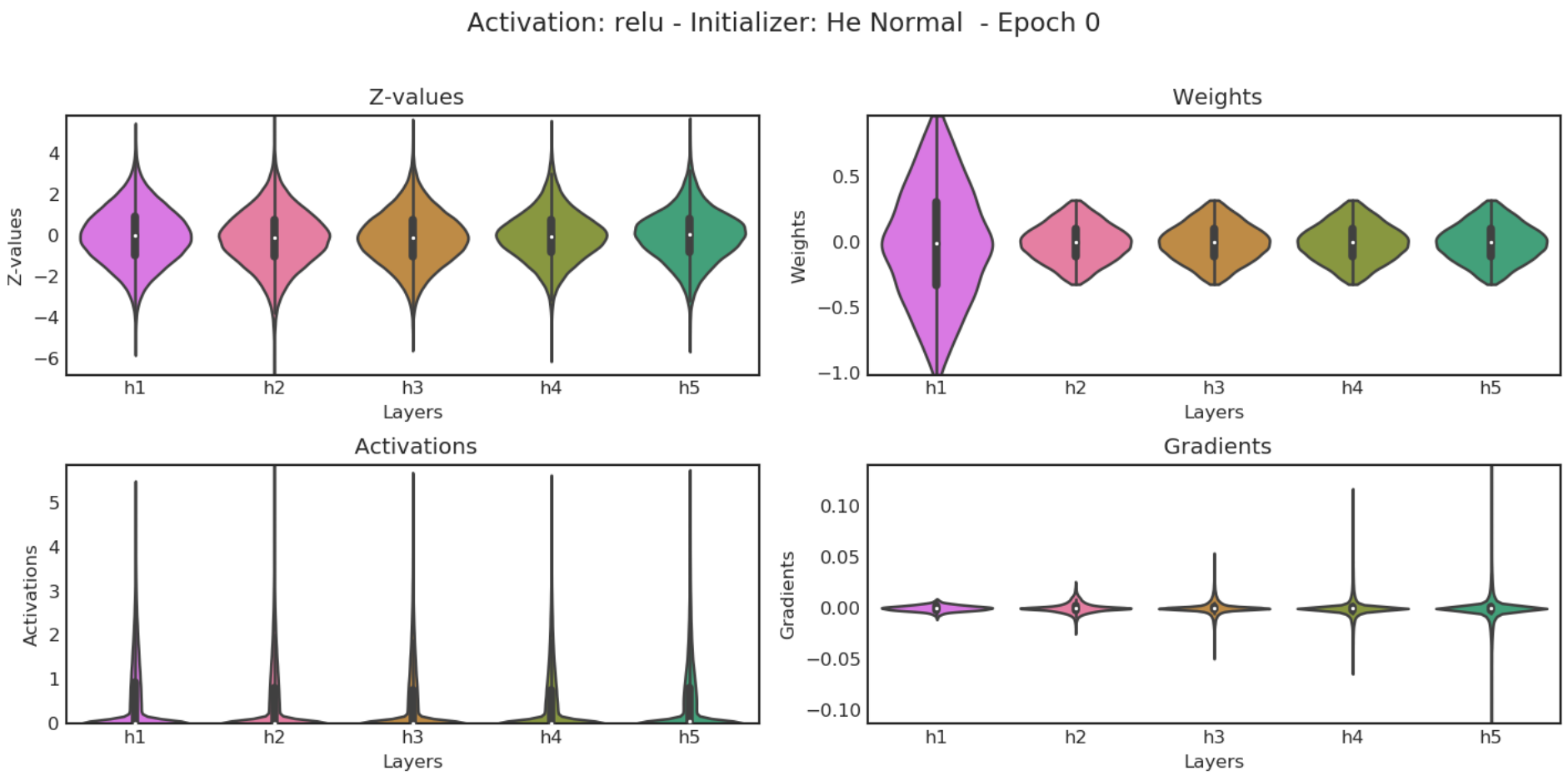
- For the **Uniform distribution**, we compute the limits accordingly

$$Var(Uniform(-limit, limit)) = \frac{limit^2}{3}$$

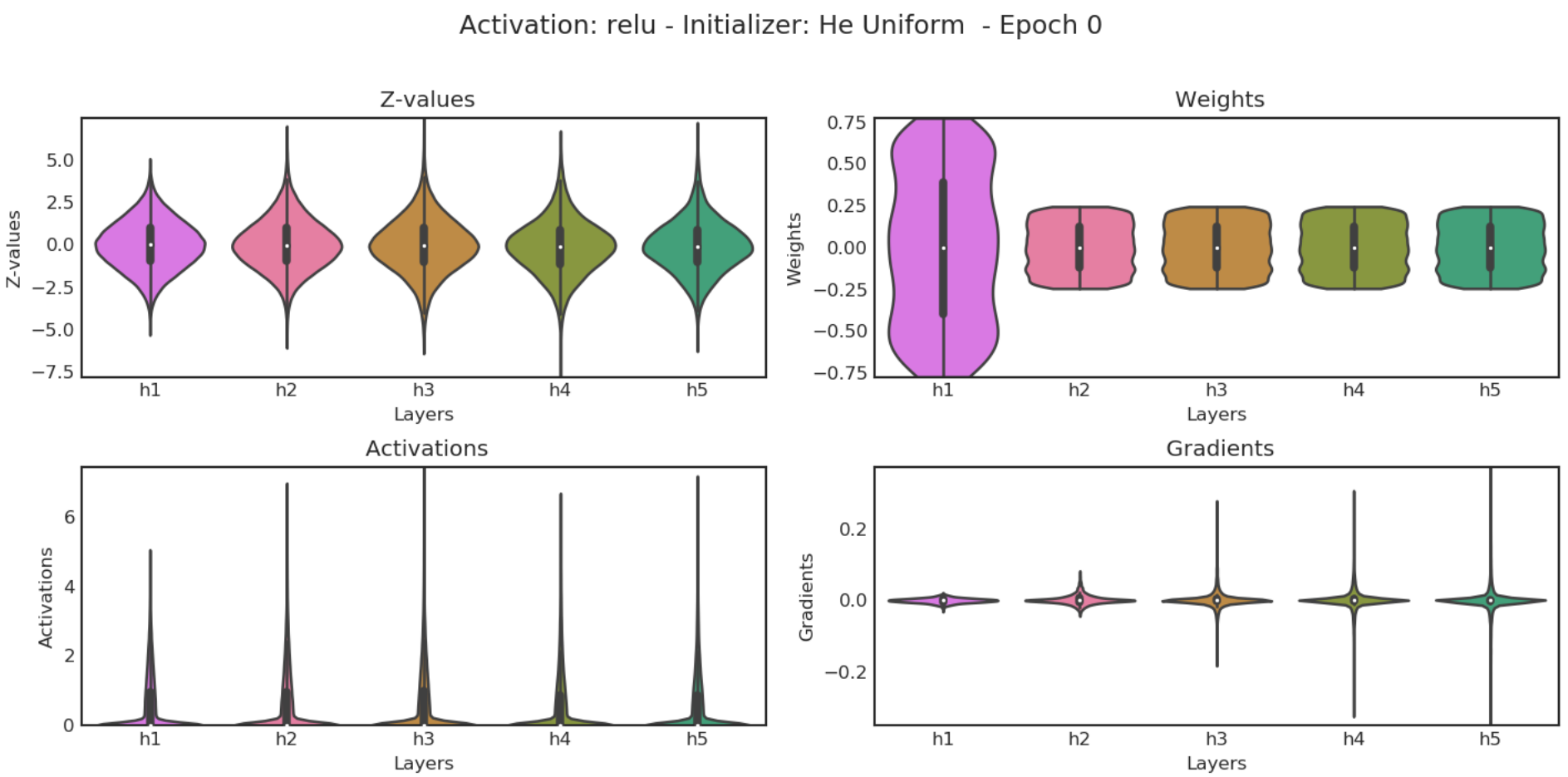
$$Var(W) = \frac{2}{fan_{in}} \iff limit = \sqrt{\frac{6}{fan_{in}}}$$

He et al. showed in their paper that, for common network designs, if the initialization scheme scales the activation values during the *forward pass*, it does the trick for the *backpropagation* **as well**. Moreover, it works **both** ways, so we could even use "**fan out**" instead of "**fan in**"

He Initialization Scheme



He Initialization Scheme



Weight Initialization

A brief summary of this section is very simple: for symmetric activation functions with zero mean (mainly tanh), use Xavier initialization, and for ReLU and the functions like it - Xe initialization.

Adaptive gradient descent

By discretizing time and moving forward step by step, we have θ_t on step *t*, so vector of updates will be $u_t = -\eta \nabla_\theta E(\theta_{t-1}), \quad \theta_t = \theta_{t-1} + u_t$

```
u = - learning_rate * grad
theta += u
```

It may well be that some weights are already close to their local minima, and these coordinates need to move slowly and carefully, and the other weights are still in the middle of the corresponding slope, and they can be changed much faster. In order to be able to adapt the learning rate for different parameters automatically adaptive optimization methods were created. Despite the fact that they still have their own metaparameters, these methods behave better on sparse data and are more stable than changing the unified learning rate in its pure form.

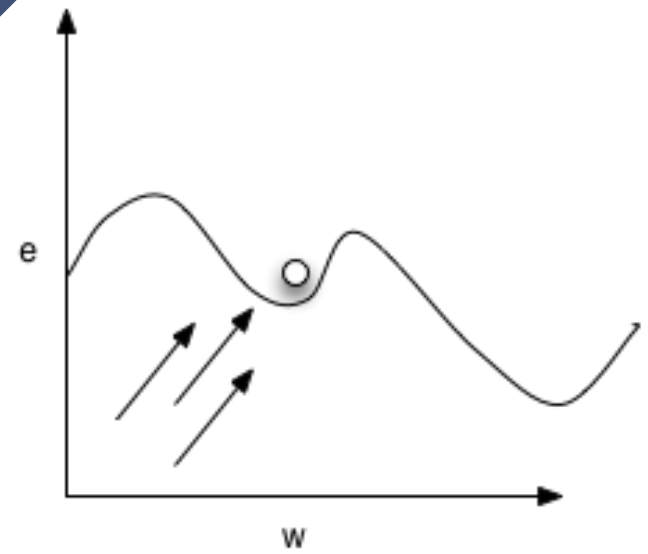
Momentum Backpropagation

Momentum adds another term to the calculation of vector of updates u_t:

$$u_t = -\eta \nabla_\theta E(\theta_{t-1}) + \lambda u_{t-1}, = \theta_{t-1} + u_t$$

u = lambda*u - learning_rate * grad

theta += u



Like the learning rate, momentum adds another training parameter that scales the effect of momentum. Momentum backpropagation has two training parameters: learning rate (η, eta) and momentum (λ, lambda). Momentum simply adds the scaled value of the previous weight change amount (u_{t-1}) to the current weight change amount(u_t).

This has the effect of adding additional force behind a direction a weight was moving. This might allow the weight to escape a local minima. A very common value for momentum is 0.9.

Adaptive gradient descent

One problem with simple backpropagation training algorithms is that they are highly sensative to learning rate and momentum. This is difficult because:

- Learning rate must be adjusted to a small enough level to train an accurate neural network.
- Momentum must be large enough to overcome local minima, yet small enough to not destabilize the training.
- A single learning rate/momentum is often not good enough for the entire training process. It is often useful to automatically decrease learning rate as the training progresses.
- All weights share a single learning rate/momentum.

Adaptive gradient descent

Other training techniques:

- Resilient Propagation - Use only the magnitude of the gradient and allow each neuron to learn at its own rate. No need for learning rate/momentum; however, only works in full batch mode.
- Nesterov accelerated gradient - Helps mitigate the risk of choosing a bad mini-batch.
- Adagrad - Allows an automatically decaying per-weight learning rate and momentum concept.
- Adadelta - Extension of Adagrad that seeks to reduce its aggressive, monotonically decreasing learning rate.
- Non-Gradient Methods - Non-gradient methods can sometimes be useful, though rarely outperform gradient-based backpropagation methods. These include: simulated annealing, genetic algorithms, particle swarm optimization, Nelder Mead, and many more

ADAM Update

Kingma and Ba (2014) introduced the Adam update rule that derives its name from the adaptive moment estimates that it uses. Adam estimates the first (mean) and second (variance) moments to determine the weight corrections. Adam begins with an exponentially decaying average of past gradients (m):

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

This average accomplishes a similar goal as classic momentum update; however, its value is calculated automatically based on the current gradient (g_t). The update rule then calculates the second moment (v_t):

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

ADAM Update

The values m_t and v_t are estimates of the first moment (the mean) and the second moment (the uncentered variance) of the gradients respectively. However, they will have a strong bias towards zero in the initial training cycles. The first moment's bias should be corrected.

The ultimate Adam update rule:

$$u_t = \frac{\eta}{\sqrt{v + \varepsilon}} m_t$$

```
m = beta1*m + (1-beta1)*dx
v = beta2*v + (1-beta2)*(dx**2)
u = - learning_rate * m / (np.sqrt(v) + eps)
theta += u
```

```
train_op = tf.train.AdamOptimizer().minimize(loss)
```

```
model.compile(loss='mean_squared_error', optimizer='adam')
```

Kingma and Ba (2014) propose default values of $\beta_1 = 0{,}9;\ \beta_2 = 0{,}999, \varepsilon = 10^{-8}$

Convolutional Neural Networks

...The visual system of the brain has the organization, computational profile, and architecture it has in order to facilitate the organism's thriving at the four Fs: feeding, fleeing, fighting, and reproduction.

P. S. Churchland, V. S. Ramachandran, T. J. Sejnowski. A Critique of Pure Vision

Convolutional neural networks, CNN - this is a very wide class of architectures whose main idea is to reuse the same parts of the neural network to work with different small, local sections of inputs. Like many other neural architectures, convolutional networks have been known for a long time, and today they have found a lot of diverse applications, but the main application for which people once invented convolutional networks remains ***image processing***.

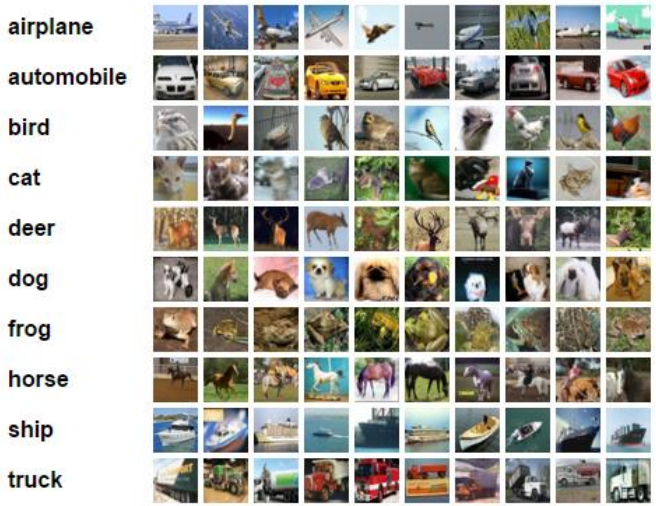
Convolutional Neural Networks

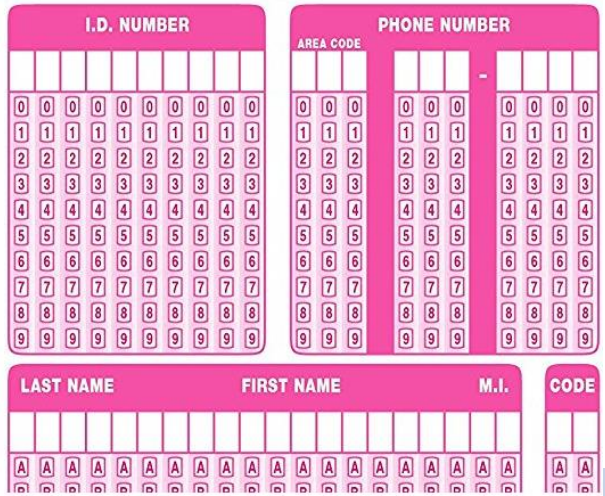
According to current representations:

- in the V1 zone, local features of small areas of image read from the retina stand out;
- V2 continues to highlight local features, slightly summarizing them and adding binocular vision (i.e., the stereo effect of two eyes);
- in the V3 zone, color, textures of objects are recognized, the first results of their segmentation and grouping appear;
- zone V4 is already beginning to recognize geometric shapes and outlines of objects that are still simple; besides, it is here that the modulation is strongest through our attention: the activation of neurons in V4 is not uniform across the entire field of view, but depends strongly on what we are paying attention to, consciously or unconsciously;
- the V5 zone is mainly engaged in the recognition of movements, trying to understand where and with what speed those objects move in sight, the outlines of which stood out in the V4 zone;
- in the V6 zone, data about the whole picture is summarized; it reacts to changes throughout the field of view (wide-field stimulation) and changes in the picture due to the fact that the person himself moves;
- sometimes also distinguish the V7 zone, where the recognition of complex objects, in particular human faces.

Computer Vision Data Sets:

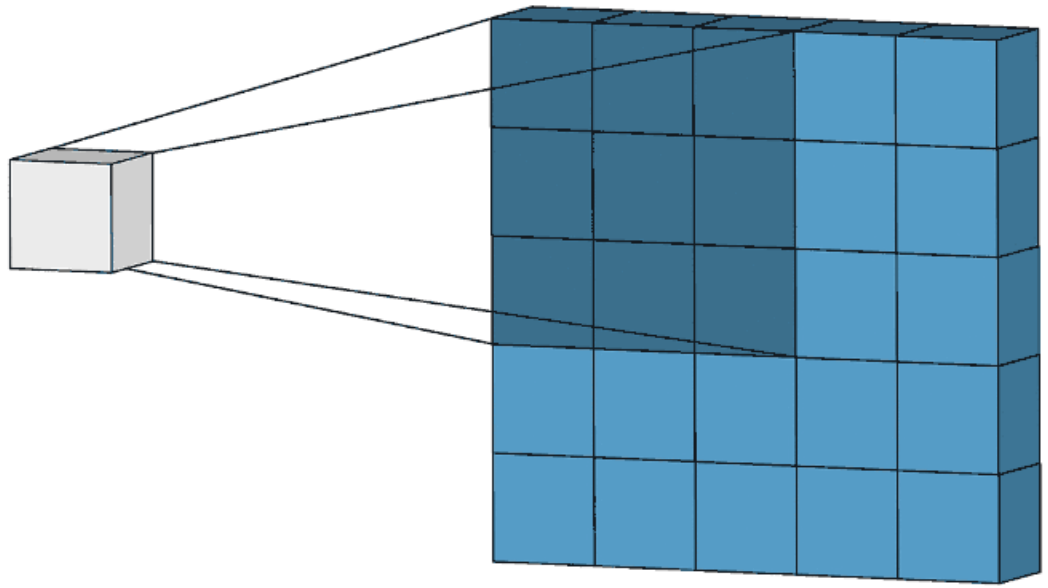
- MNIST Digits Data Set - is very popular in the neural network research community.
- CIFAR Data Set - the CIFAR-10 data set contains low-rez images that are divided into 10 classes. The CIFAR-100 data set contains 100 classes in a hierarchy.
- Imagenet:Large Scale Visual Recognition Challenge 2014





Convolutional Neural Networks

The convolutional neural network (CNN) is a neural network technology that has profoundly impacted the area of computer vision (CV). Fukushima (1980) introduced the original concept of a convolutional neural network, and LeCun, Bottou, Bengio & Haffner (1998) greatly improved this work. From this research, Yan LeCun introduced the famous LeNet-5 neural network architecture.



Convolutional Neural Networks

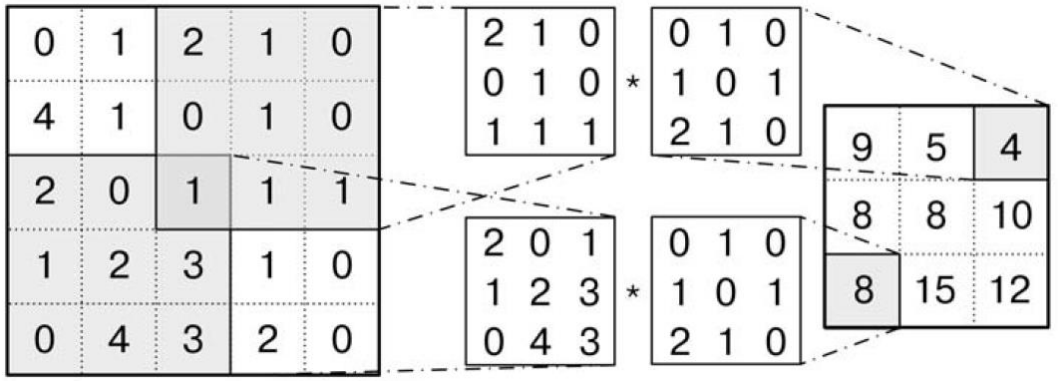
The 2D convolution is a fairly simple operation at heart: you start with a kernel, which is simply a small matrix of weights. This kernel “slides” over the 2D input data, performing an elementwise multiplication with the part of the input it is currently on, and then summing up the results into a single output pixel.

Convolutional Neural Networks

- The kernel repeats this process for every location it slides over, converting a 2D matrix of features into yet another 2D matrix of features. The output features are essentially, the weighted sums (with the weights being the values of the kernel itself) of the input features located roughly in the same location of the output pixel on the input layer.

$$\begin{pmatrix} 0 & 1 & 2 & 1 & 0 \\ 4 & 1 & 0 & 1 & 0 \\ 2 & 0 & 1 & 1 & 1 \\ 1 & 2 & 3 & 1 & 0 \\ 0 & 4 & 3 & 2 & 0 \end{pmatrix} * \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 2 & 1 & 0 \end{pmatrix} = \begin{pmatrix} 9 & 5 & 4 \\ 8 & 8 & 10 \\ 8 & 15 & 12 \end{pmatrix}$$

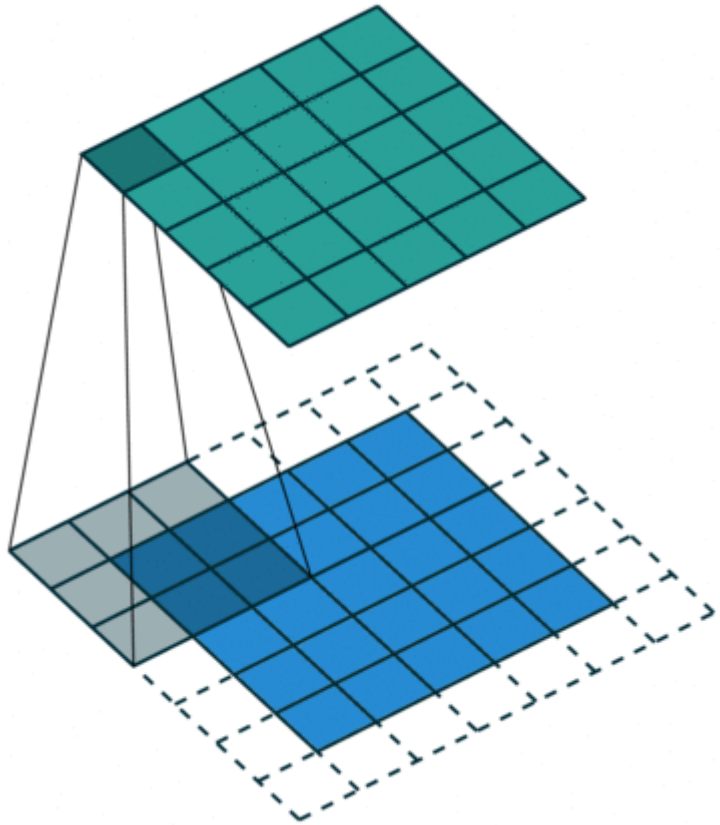
- Whether or not an input feature falls within this “roughly same location”, gets determined directly by whether it’s in the area of the kernel that produced the output or not. This means the size of the kernel directly determines how many (or few) input features get combined in the production of a new output feature.



- This is all in pretty stark contrast to a fully connected layer.

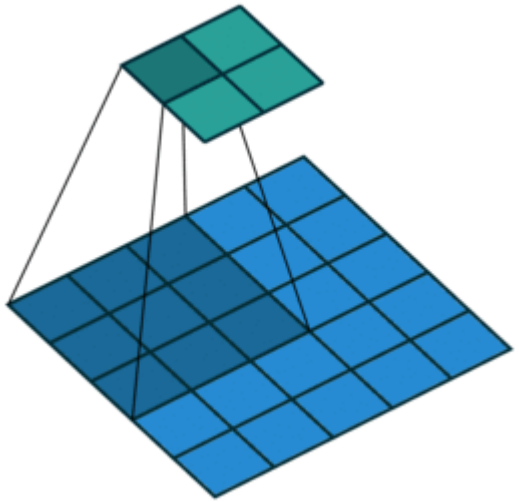
Convolutional Neural Networks - Padding and Strides

Padding: during the sliding process, the edges essentially get “trimmed off”, converting a 5×5 feature matrix to a 3×3 one. The pixels on the edge are never at the center of the kernel, because there is nothing for the kernel to extend to beyond the edge. This isn’t ideal, as often we’d like the size of the output to equal the input. Padding does something pretty clever to solve this: pad the edges with extra, “fake” pixels (usually of value 0, hence the oft-used term “zero padding”). This way, the kernel when sliding can allow the original edge pixels to be at its center, while extending into the fake pixels beyond the edge, producing an output the same size as the input.



Convolutional Neural Networks - Padding and Strides

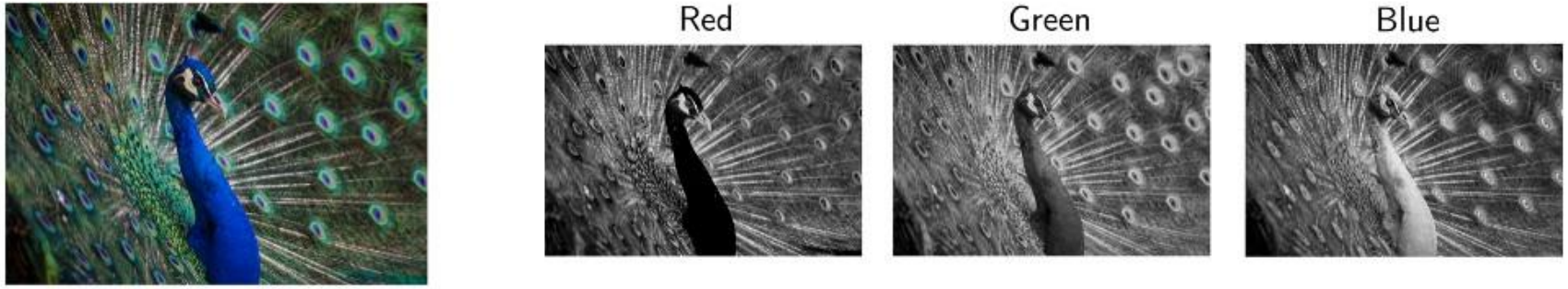
Striding: Often when running a convolution layer, you want an output with a lower size than the input. This is commonplace in convolutional neural networks, where the size of the spatial dimensions are reduced when increasing the number of channels. One way of accomplishing this is by using a pooling layer (eg. taking the average/max of every 2×2 grid to reduce each spatial dimensions in half). Yet another way to do is is to use a stride.



The idea of the stride is to skip some of the slide locations of the kernel. A stride of 1 means to pick slides a pixel apart, so basically every single slide, acting as a standard convolution. A stride of 2 means picking slides 2 pixels apart, skipping every other slide in the process, downsizing by roughly a factor of 2, a stride of 3 means skipping every 2 slides, downsizing roughly by factor 3, and so on.

Convolutional Neural Networks

Of course, the diagrams above only deals with the case where the image has a single input channel. In practicality, most input images have 3 channels, and that number only increases the deeper you go into a network. It's pretty easy to think of channels, in general, as being a "view" of the image as a whole, emphasizing some aspects, de-emphasizing others.

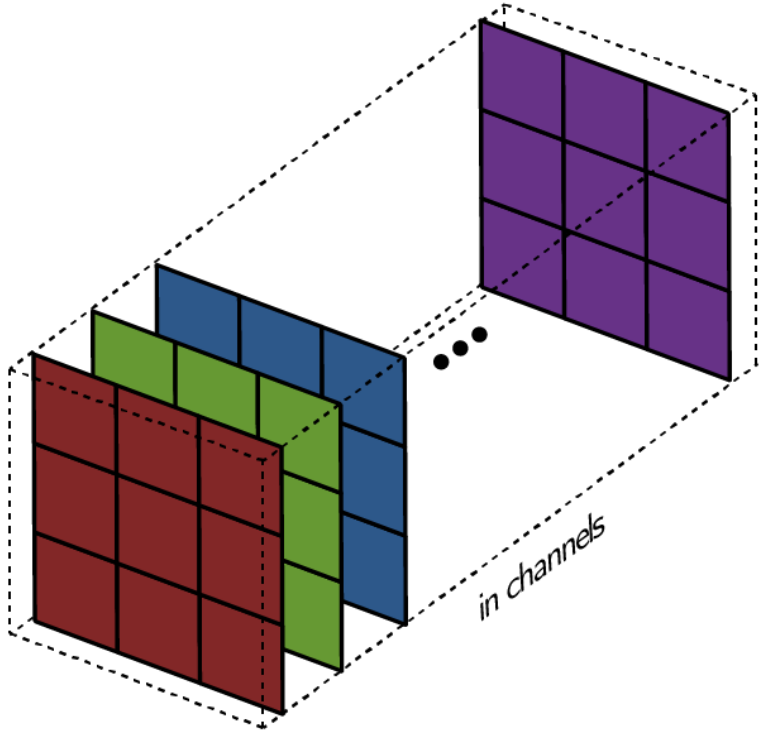


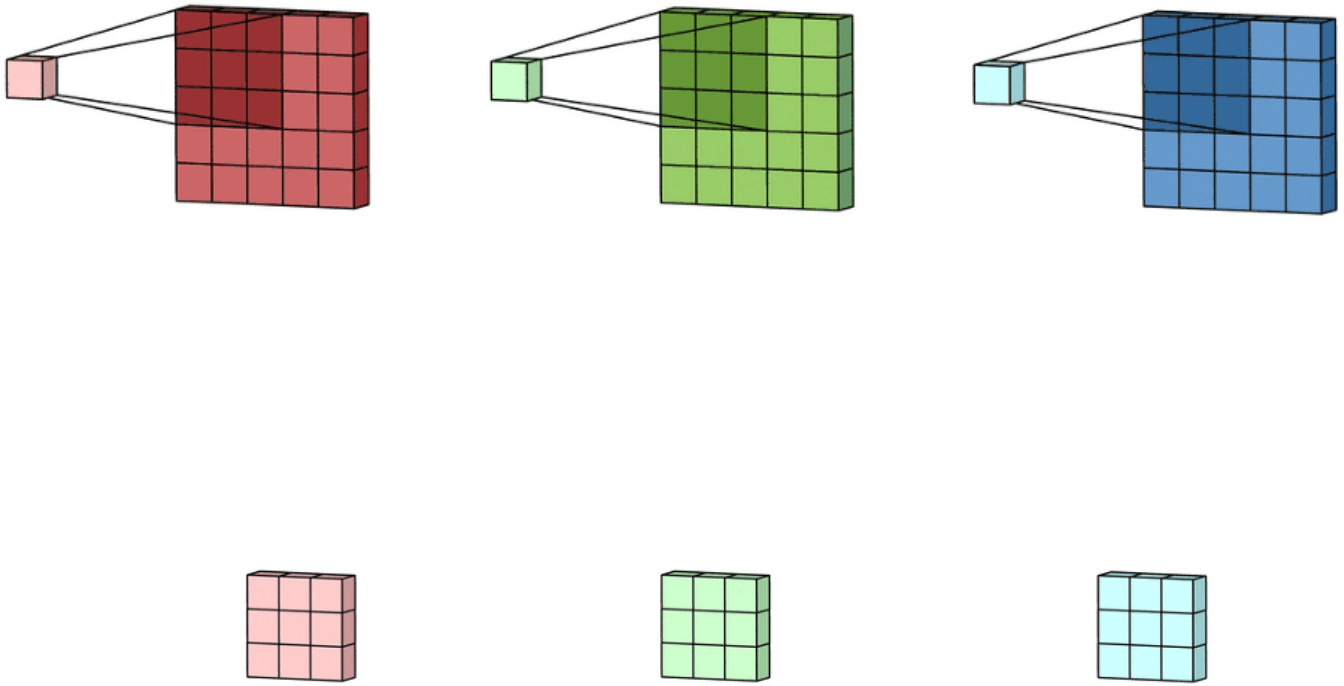
Convolutional Neural Networks

So this is where a key distinction between terms comes in handy: whereas in the 1 channel case, where the term filter and kernel are interchangeable, in the general case, they're actually pretty different. Each filter actually happens to be a collection of kernels, with there being one kernel for every single input channel to the layer, and each kernel being unique.

Each filter in a convolution layer produces one and only one output channel, and they do it like so:

Each of the kernels of the filter "slides" over their respective input channels, producing a processed version of each. Some kernels may have stronger weights than others, to give more emphasis to certain input channels than others (eg. a filter may have a red kernel channel with stronger weights than others, and hence, respond more to differences in the red channel features than the others).

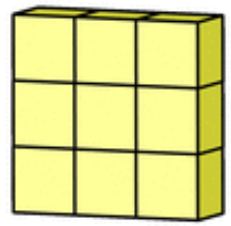




Each of the per-channel processed versions are then summed together to form one channel. The kernels of a filter each produce one version of each channel, and the filter as a whole produces one overall output channel.

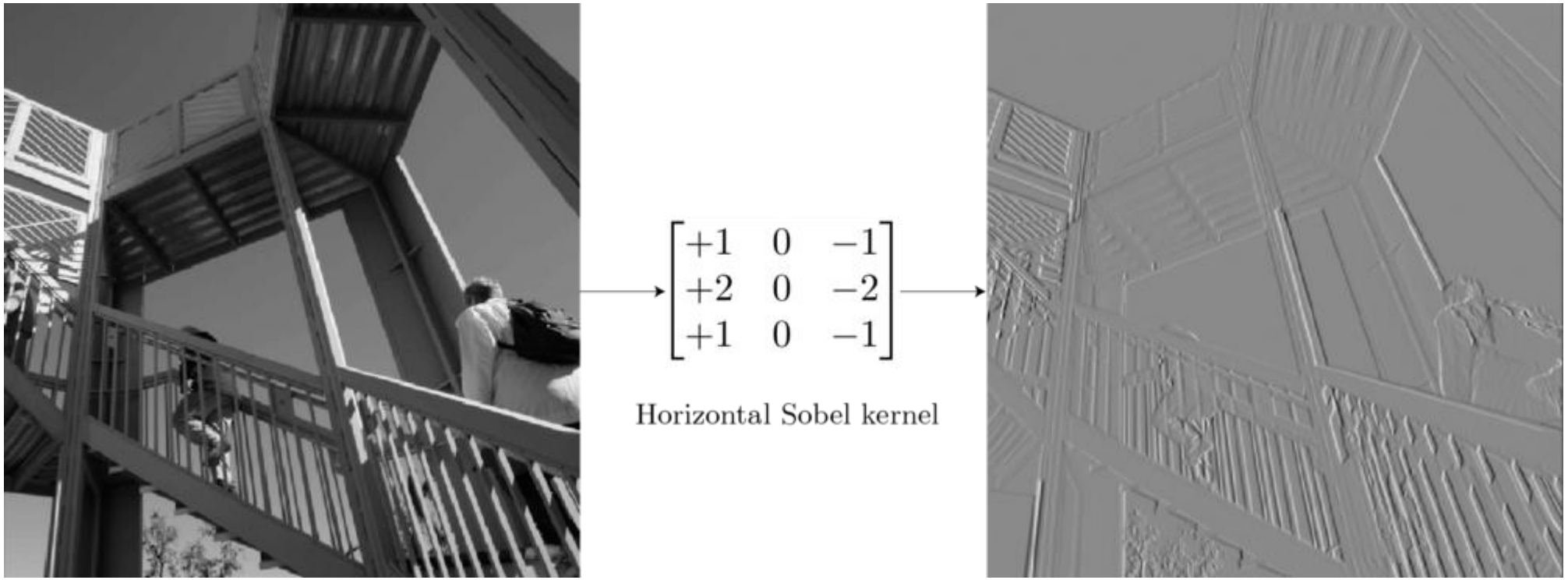
Convolutional Neural Networks

Finally, then there's the bias term. The way the bias term works here is that each output filter has one bias term. The bias gets added to the output channel so far to produce the final output channel.

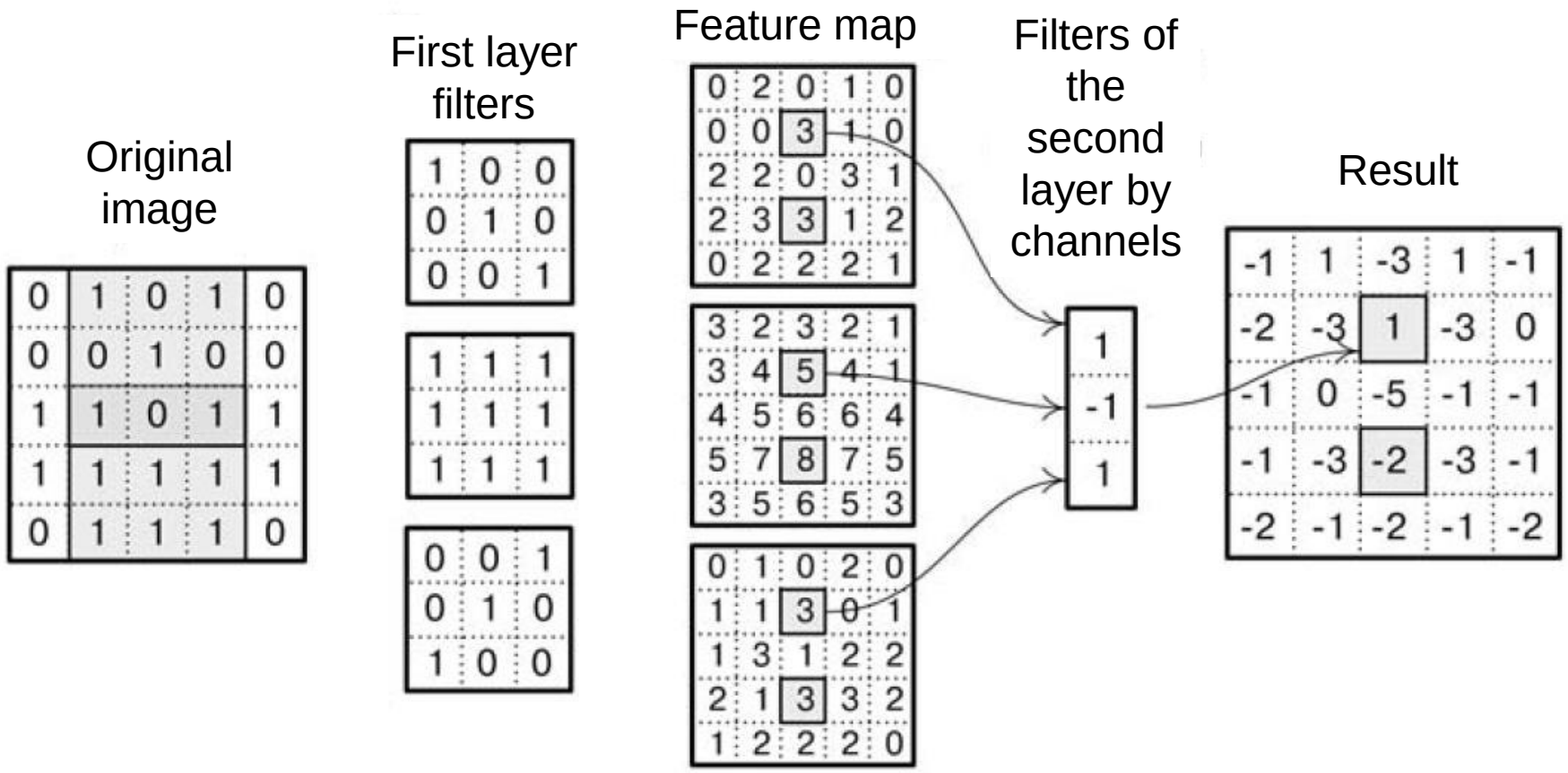


And with the single filter case down, the case for any number of filters is identical: Each filter processes the input with its own, different set of kernels and a scalar bias with the process described above, producing a single output channel. They are then concatenated together to produce the overall output, with the number of output channels being the number of filters. A nonlinearity is then usually applied before passing this as input to another convolution layer, which then repeats this process.

Convolutional Neural Networks



Convolutional Neural Networks



Convolutional Neural Networks in TensorFlow

- **Input data** for two-dimensional convolution in TensorFlow should have a four-dimensional structure that looks like this: [*batch size; height; width; channels*]
- For example, if we use mini-batches of 32 images in each and train the network on RGB images of faces with a size of 28x28 pixels, then the total dimension of the data tensor will be [32; 28; 28; 3]: if you multiply all the dimensions, it turns out that each mini-batch fed to the input of the network contains about 75 thousand numbers! Each image is presented with 28x28x3 = 2352 real numbers
- The dimension of the **convolutional weights tensor** is determined by the size of the convolution kernel and the number of channels both at the input and at the output: [*height; width; input channels; output channels*]

Convolutional Neural Networks in TensorFlow

- `"VALID"` = without padding:

```
inputs:         1  2  3  4  5  6  7  8  9  10 11 (12 13)
                |_____________|                 dropped
                      |_____________|
```

- `"SAME"` = with zero padding:

```
               pad|                                      |pad
inputs:         0 |1  2  3  4  5  6  7  8  9  10 11 12 13|0  0
               |_____________|
                     |_____________|
                           |_____________|
                                 |_____________|
```

In this example:

- Input width = 13
- Filter width = 6
- Stride = 5

- Valid or no padding:

The valid padding involves no zero padding, so it covers only the valid input, not including artificially generated zeros. The length of output is ((the length of input) - (k-1)) for the kernel size k if the stride s=1.

- Same or half padding:

The same padding makes the size of outputs be the same with that of inputs when s=1. If s=1, the number of zeros padded is (k-1).

- Full padding:

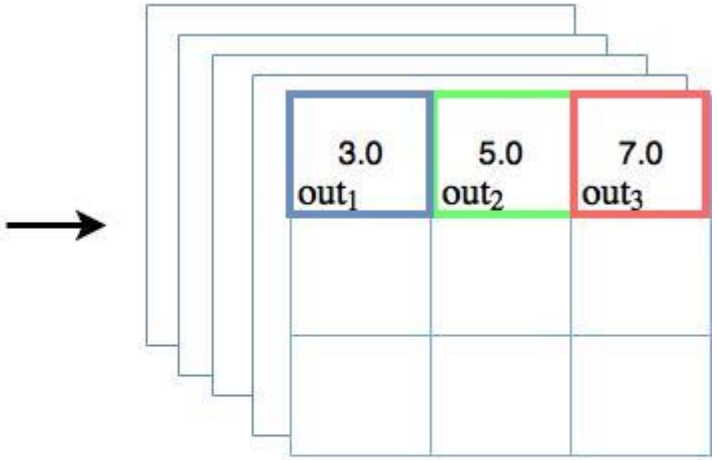
The full padding means that the kernel runs over the whole inputs, so at the ends, the kernel may meet the only one input and zeros else. The number of zeros padded is 2(k-1) if s=1. The length of output is ((the length of input) + (k-1)) if s=1.

Convolutional Neural Networks in TensorFlow

```
import tensorflow as tf
import numpy as np

x_inp = tf.placeholder(tf.float32, [5, 5])
w_inp = tf.placeholder(tf.float32, [3, 3])

x = tf.reshape(x_inp, [1, 5, 5, 1])
w = tf.reshape(w_inp, [3, 3, 1, 1])

x_valid = tf.nn.conv2d(x, w, strides=[1, 1, 1, 1], padding="VALID")
x_same = tf.nn.conv2d(x, w, strides=[1, 1, 1, 1], padding="SAME")
x_valid_half = tf.nn.conv2d(x, w, strides=[1, 2, 2, 1], padding="VALID")
x_same_half = tf.nn.conv2d(x, w, strides=[1, 2, 2, 1], padding="SAME")

x = np.array([[0, 1, 2, 1, 0],
              [4, 1, 0, 1, 0],
              [2, 0, 1, 1, 1],
              [1, 2, 3, 1, 0],
              [0, 4, 3, 2, 0]])
w = np.array([[0, 1, 0],
              [1, 0, 1],
              [2, 1, 0]])

sess = tf.Session()
y_valid, y_same, y_valid_half, y_same_half = sess.run([x_valid, x_same, x_valid_half, x_same_half], feed_dict={x_inp: x, w_inp: w})

print ("padding=VALID:\n", y_valid[0, :, :, 0])
print ("padding=SAME:\n", y_same[0, :, :, 0])
print ("padding=VALID, stride 2:\n", y_valid_half[0, :, :, 0])
print ("padding=SAME, stride 2:\n", y_same_half[0, :, :, 0])
```

Pooling

2.0 in_1	3.0 in_2	0.0 in_3	5.0 in_4	2.5 in_5	0.0 pad_1
2.0 in_6	1.5 in_7	0.5 in_8	0.0 in_9	7.0 in_{10}	0.0 pad_2
1.5	5.0	5.0	3.0	2.0	0.0
3.0	5.0	7.0	1.5	0.0	0.0
2.0	5.0	2.0	1.5	2.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0

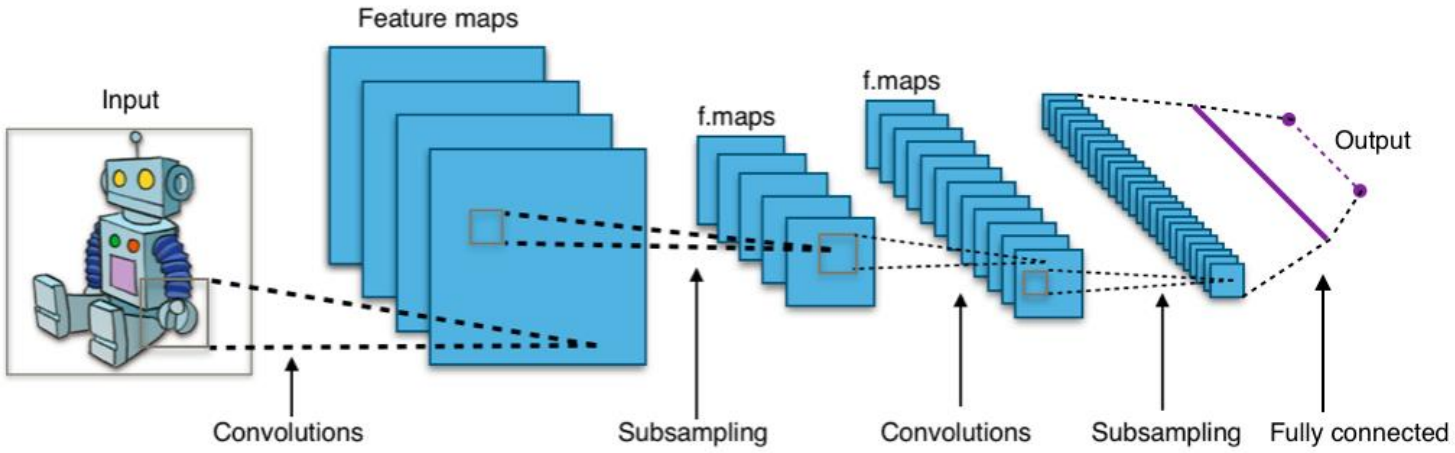


Pooling

```
import tensorflow as tf
import numpy as np

x_inp = tf.placeholder(tf.float32, [4, 4])
w_inp = tf.placeholder(tf.float32, [3, 3])

x = tf.reshape(x_inp, [1, 4, 4, 1])
w = tf.reshape(w_inp, [3, 3, 1, 1])

x_valid = tf.nn.max_pool(x, ksize=[1, 2, 2, 1], strides=[1, 1, 1, 1], padding="VALID")
x_valid_half = tf.nn.max_pool(x, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding="VALID")

x = np.array([[0, 1, 2, 1],
              [4, 1, 0, 1],
              [2, 0, 1, 1],
              [1, 2, 3, 1]])

sess = tf.Session()
y_valid, y_valid_half = sess.run([x_valid, x_valid_half], feed_dict={x_inp: x})


print ("padding=VALID:\n", y_valid[0, :, :, 0])
print ("padding=VALID, stride 2:\n", y_valid_half[0, :, :, 0])
```

The fully connected layer

First we have the input image of a robot. Then multiple convolutional filters (these would include rectified linear unit activations), followed by pooling / sub-sampling. Then we have another layer of convolution and pooling. Notice the number of channels (the stacked blue squares) and the reduction in the x, y sizes of each channel as the sub-sampling / down-sampling occurs in the pooling layers. Finally, we reach a fully connected layer before the output. This layer hasn't been mentioned yet, and deserves some discussion.



The fully connected layer

At the output of the convolutional-pooling layers we have moved from high resolution, low level data about the pixels to representations of objects within the image. The purpose of these final, fully connected layers is to make classifications regarding these objects – in other words, we bolt a standard neural network classifier onto the end of a trained object detector. As you can observe, the output of the final pooling layer is many channels of x x y matrices. To connect the output of the pooling layer to the fully connected layer, we need to flatten this output into a single (N x 1) tensor.

CNN for digits recognition

- x_image = tf.reshape(x, [-1,28,28,1])
- Kernel – 5x5, filters – 32

1) Layer of conv2d
2) ReLu
3) Pooling
4) + one more layer* with 64 filters
5) h_pool_2_flat = tf.reshape(h_pool_2, [-1, 7*7*64])
6) Use Adam