Training Neural Networks – Part 1

Table of Contents

[Activation Functions 2](#_Toc138881931)

[Sigmoid 2](#_Toc138881932)

[Hyperbolic Tangent 4](#_Toc138881933)

[Rectified Linear Unit 4](#_Toc138881934)

[Data Preprocessing 7](#_Toc138881935)

[Weight Initialization 9](#_Toc138881936)

[Residual Networks 12](#_Toc138881937)

[Regularization 13](#_Toc138881938)

[Dropout 13](#_Toc138881939)

[Dropout During Testing 15](#_Toc138881940)

[Dropout Architectures 17](#_Toc138881941)

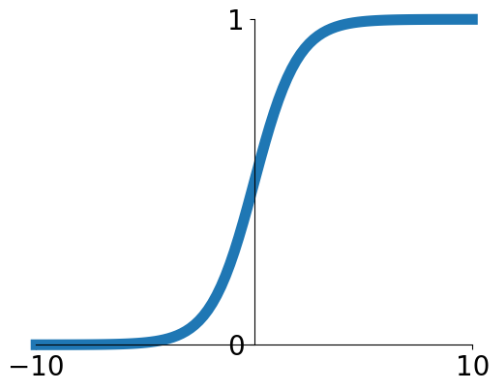
[Data Augmentation 18](#_Toc138881942)

[Other Regularization Techniques 19](#_Toc138881943)

## Activation Functions

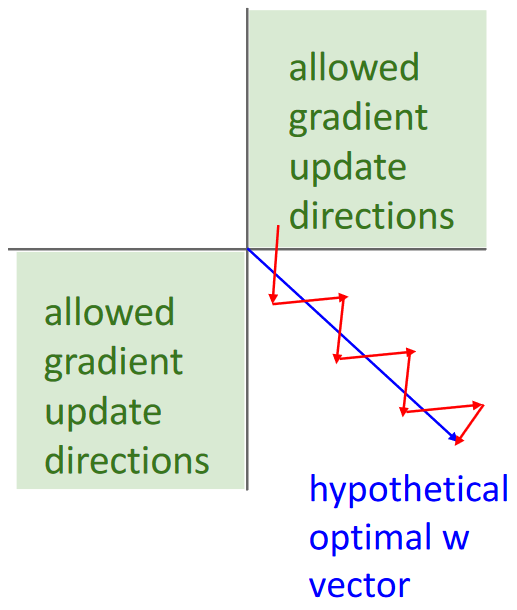
We have previously studied **activation functions** and how they are used to introduced non-linearity into a network. There are several types of activation functions, and we will now be studied the benefits and drawbacks of a few of the common ones.

### Sigmoid



The **sigmoid** activation function has been historically popular due to its interpretation as a probability, i.e., the probability of a neuron being ‘on’. However, it has three issues:

1. The first issue is the most major one. For very high or very low values of , the output becomes **saturated**. This ‘kills’ the gradient in that it becomes meaningless.
2. The outputs are **not zero-centered**. For the case of sigmoid, this means we can never have a **negative output**.



The graph above has a 2D vector, so we should be getting gradients in four possible directions. However, since the sigmoid activation does not allow for negative values, we become limited to just two directions: the case where both gradients are positive and the case where both gradients are negative. The only way for the gradient to move in the required direction is to use an awkward zig-zag motion. The diagonal direction would require one of the gradients to be positive while the other is negative, which it cannot achieve.

This issue can be somewhat mitigated by the use of **mini-batches**. Suppose we have four inputs, two of which has positive gradients and two of which have negative gradients. The gradient update would depend on the average values of each of the two gradients across the four inputs, which has the possibility of being a combination of a positive and a negative value. As such, the issue of sigmoid being unable to produce negative values is a relatively minor one.

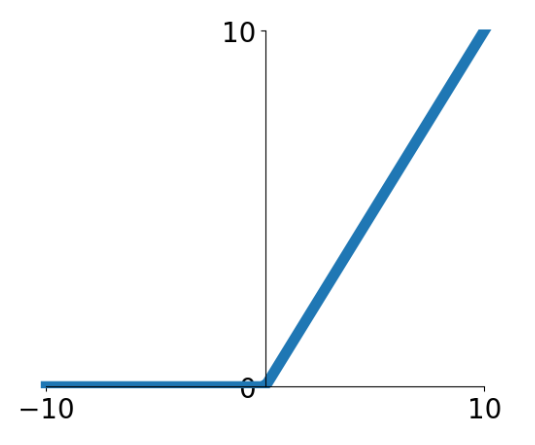
1. The third issue is that the **exp** function is computationally expensive to use.

### Hyperbolic Tangent



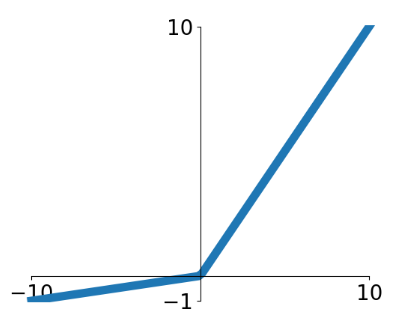
The **hyperbolic tangent** activation function is similar to the sigmoid function. It is **zero-centered**, so it solves the issues that sigmoid has with being unable to update gradients in any direction, but the issue of **saturated outputs** still remains. Additionally, it also uses the **exp** function, making it equally computationally expensive.

### Rectified Linear Unit

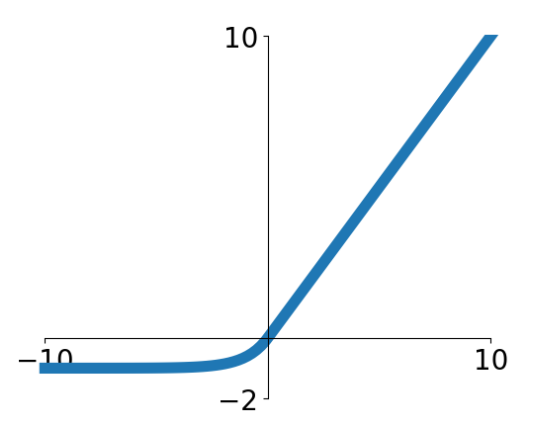


The Rectified Linear Unit, or **ReLU**, activation function is possibly the most commonly used activation function due to the various benefits it provides. It does not saturate the output (at least not in the positive region), is very computationally inexpensive and has been shown to converge faster than either sigmoid or tanh. It is **not zero-centered**, but the fact that it works well suggests that this is not a big issue.

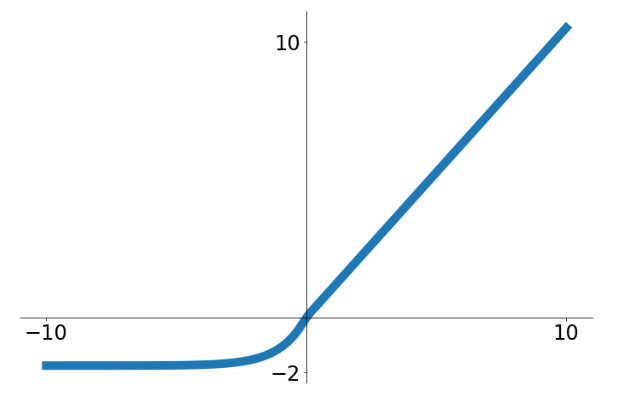
The problem with ReLU is that it does not produce any output for . If the entire dataset happens to have values of , then the network will never learn. This situation is called **dead ReLU**. One trick that people have used to get around this is to add a slightly **positive bias** to the ReLU, preventing the final output from being .



There have been various variations of the ReLU activation function which attempt to solve its issues. One of the more famous variations is called **Leaky ReLU**, where . This solves the problem of the gradient dying for . Another variation makes the multiplier a **learnable parameter**, , and is called **Parametric Rectified Linear Unit** (PReLU).

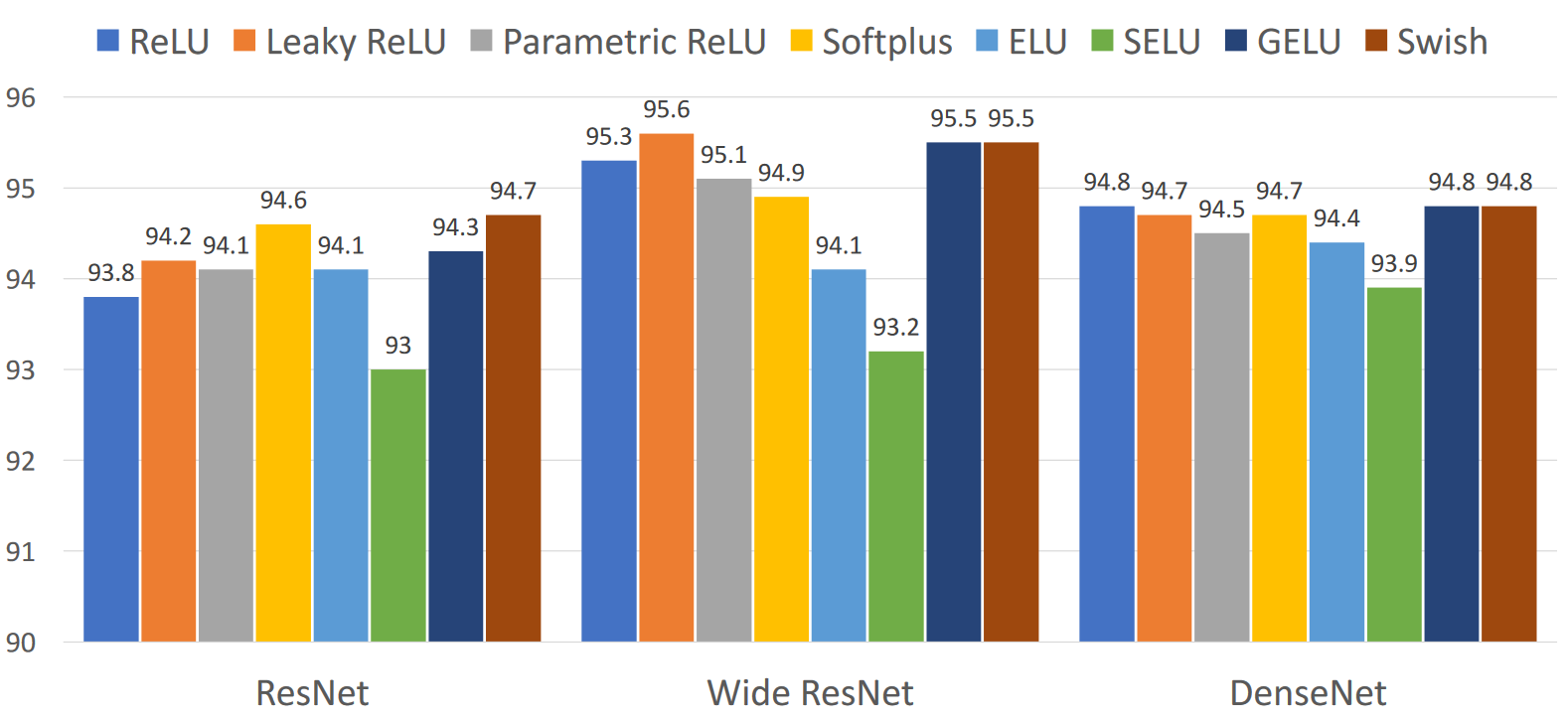


A different variation, the **Exponential Linear Unit** (ELU), which brings the output closer to being **zero-centered**. Here, . The negative regime for ELU is more comparable to the sigmoid function, creating a negative saturation regime which the authors claim adds robustness to noise. The issue with this is that it adds back the computationally expensive **exp** function.

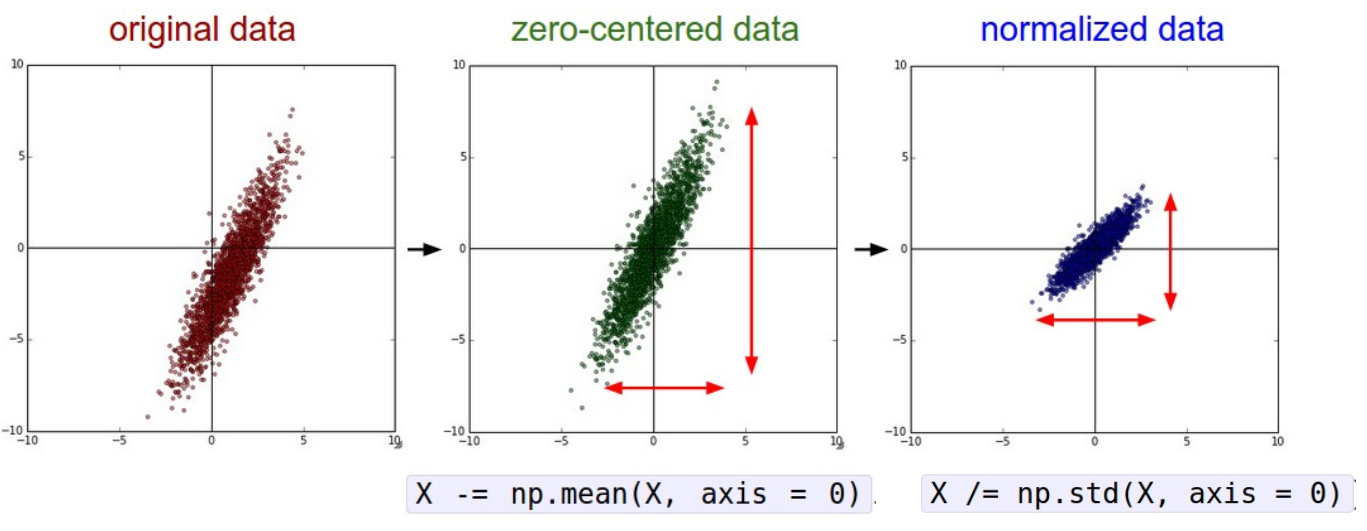


Yet another variation is the **Scaled Exponential Linear Unit** (SELU), which is a scaled version of ELU that works well with deeper networks. Here, . The authors proposed very specific values for and ( and ), which they showed allows for convergence without using batch normalization.

Despite the large number of variants of ReLU, there is not specific guideline about which one to use in which scenario. Different variants have been shown to work well on different networks. Even then, the variations are of a point of a percentage, as can be seen in the graph below for the CIFAR10 dataset. For general use cases, it is best to just use the original ReLU function. However, we should avoid using sigmoid or tanh.



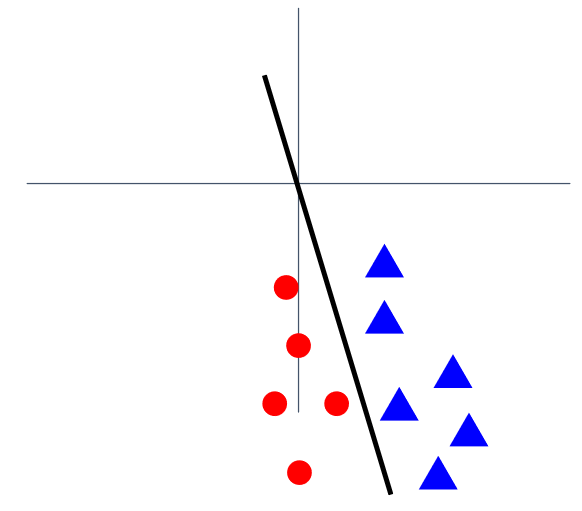
## Data Preprocessing



Two of the most common data preprocessing techniques used are to **zero-center** the data and to **normalize** the data.

Not zero-centering the data will result in all the gradients being positive or negative. This will make it difficult for gradient updates to take place in exactly the same way that the sigmoid activation function did.

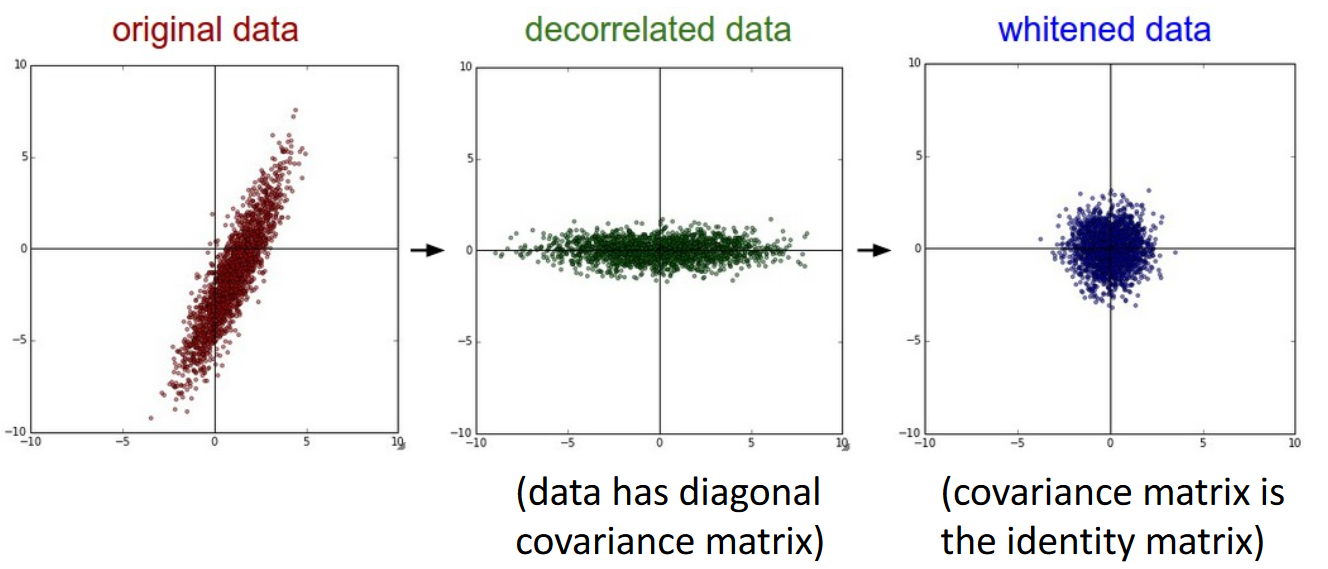
Normalization makes the data less sensitive to small changes in weight, in turn making it easier to optimize. For unnormalized data, such as in the graph below, a small variation in the decision boundary towards the center of the graph will result in a large variation in the area where the data actually lies.



These two techniques have been used in a variety of ways in different architectures:

* AlexNet – Subtracted the mean of the 32x32x3 images.
* VGGNet – Subtracted the per-channel mean, i.e., the mean for all images for each channel individually.
* ResNet – Subtracted the per-channel mean and divided by the per-channel standard deviation.

Other data pre-processing techniques include PCA and whitening, but these are less commonly used.



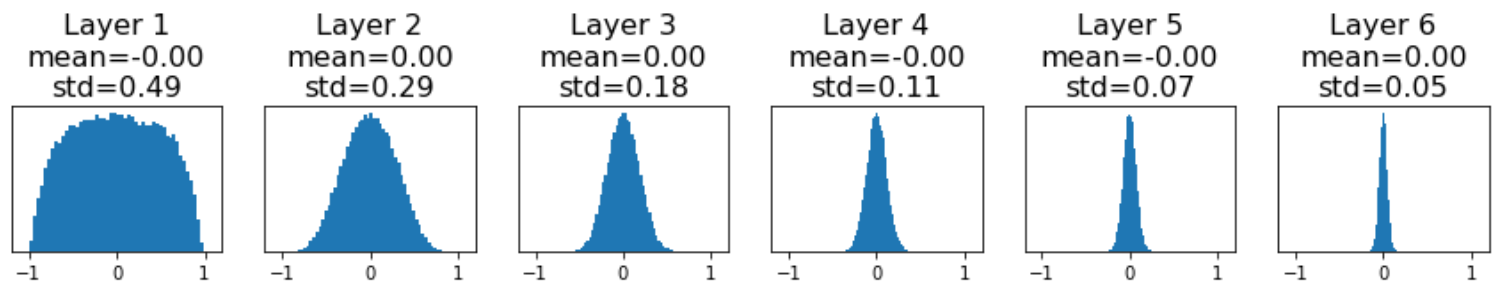
## Weight Initialization

We have previously learnt that we should not initialize the weights of a network to be all zeroes since that results in all of the outputs being zero, which in turn means all of the gradients are the same, thus preventing learning. To break this symmetry, we initialize weights to **small random numbers**.

W=0.01\*np.random.randn(Din,Dout)

PYTHON

This works well for smaller networks, but begins to cause issues for larger ones.



For deep networks, this process results in the deeper layers having their activations tend towards zero. This means the gradients will be zero, which means the network will fail to learn.

What if we change the weight initialization from using a 0.01 standard deviation to a 0.05 standard deviation? Perhaps the issue lies with the fact that we stayed too close to zero values.

W=0.05\*np.random.randn(Din,Dout)

PYTHON



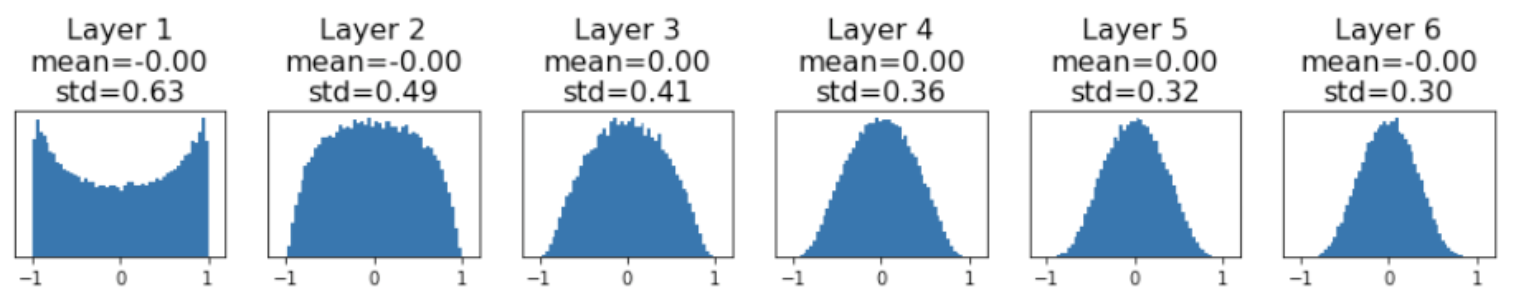
Now our activation values have become saturated (we used a tanh activation function here). This is also bad, because it means the weight initializations have become too large now.

The ‘just right’ initialization is achieved through **Xavier initialization**.

W = np.random.randn(Din,Dout) / np.sqrt(Din)

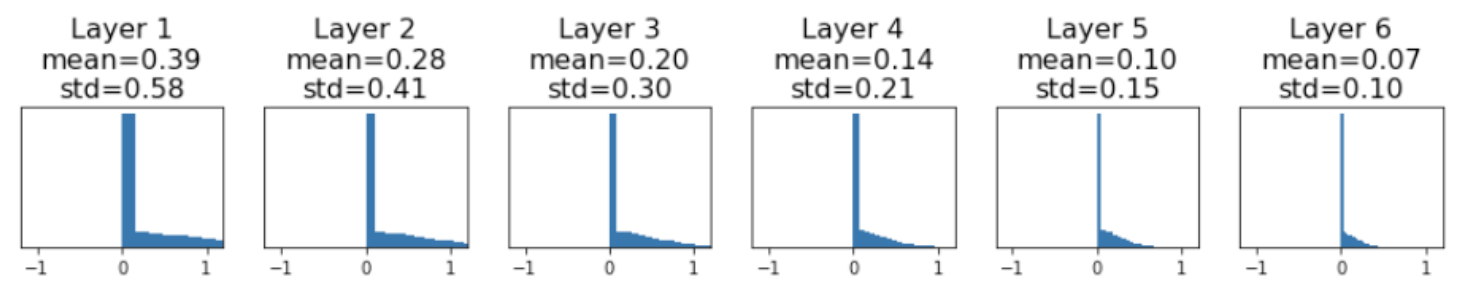
PYTHON

The equation above assumes that a fully connected layer is being used. For a convolution layer, .



The derivation of this is being skipped, but the main point is that the variance of the input should be equal to the variance of the output.

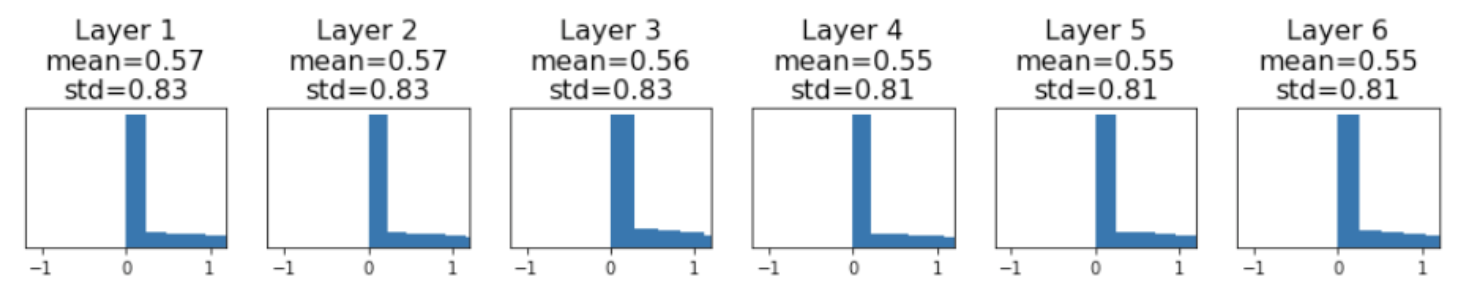
Unfortunately, if we switch out the tanh activation function for a ReLU one, the Xavier initialization results in the activations again collapsing to zero as the network gets deeper. This is because this initialization method expects a zero-centered activation function.



A correction to deal with this issue is to use **MSRA Initialization**, which uses a standard deviation of .

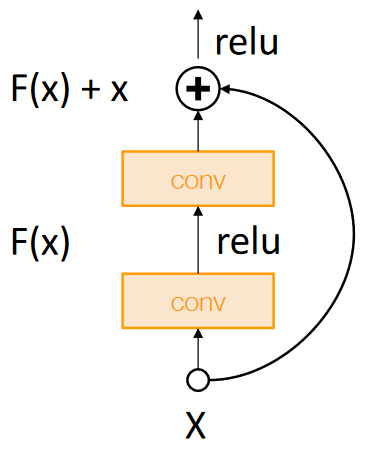
W = np.random.randn(Din,Dout) \* np.sqrt(2 / Din)

PYTHON



### Residual Networks

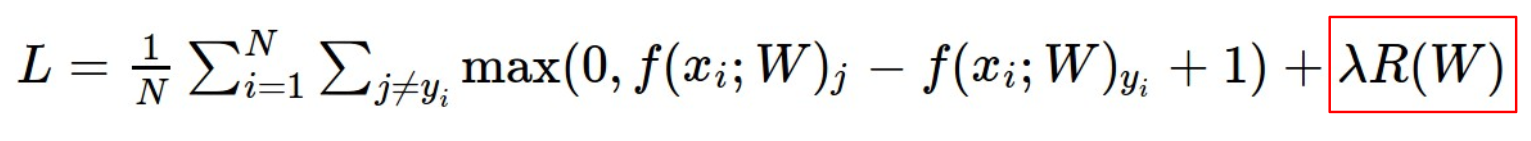
The basic idea behind using **MSRA Initialization** is that we want the variance of the output to be equal to the variance of the input, i.e., . This doesn’t work too well for **residual networks** which add the input to the output after the residual block. This means we will always have . This means the variance will keep growing as the number of layers increases, which will lead to terrible performance.



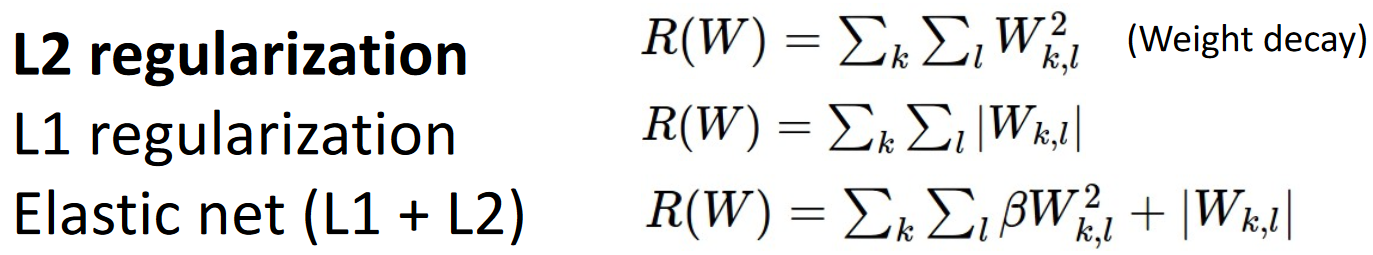
To resolve this, we modify the process a little and initialize just the first convolution block using MSRA. The second block is initialized to zero. Then .

## Regularization

We have already seen the most basic form of **regularization** which involves adding a term to the loss.



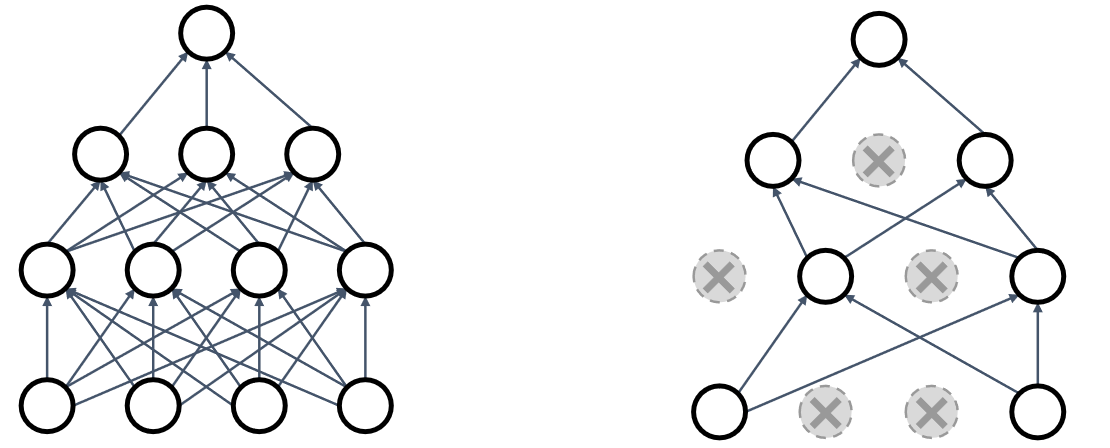
Commonly used regularization terms include L1 regularization, L2 regularization (weight decay) and Elastic net (the sum of L1 and L2).



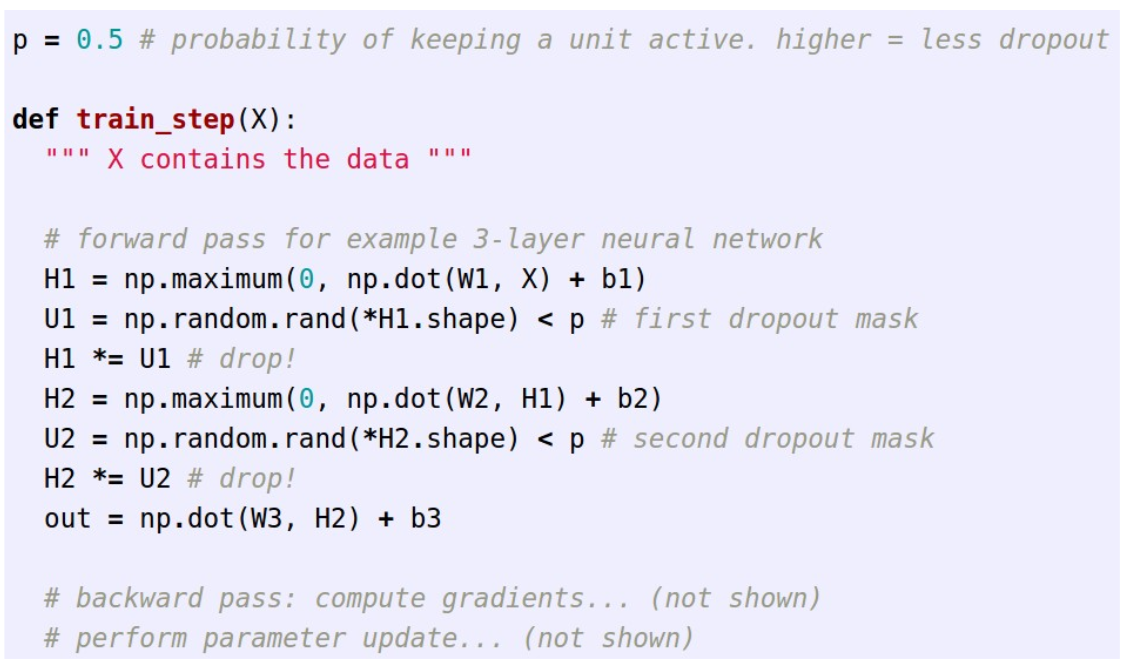
Other commonly used regularization techniques include dropout and data augmentation.

### Dropout

In **dropout**, we randomly set a portion of the neurons from the last layer to zero. The ratio of neurons discarded is a hyperparameter.



The implementation for this just uses a **binary mask**.



The reason dropout works is because it forces the network to learn **redundant representations** and prevents **co-adaptation of features**. What this means is that each neuron has a tendency to learn a different features. For example, if we are building a cat classifier and each neuron represents a different feature, the network will learn to classify the image as a cat even if not all of these features are present (since we are literally removing some of the features).



Another interpretation is that removing some neurons during one pass essentially results in a completely different network. This means that we are creating multiple networks in each pass and then creating an **ensemble** of the networks.

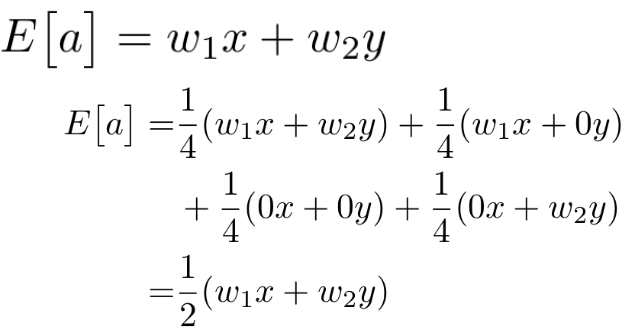
### Dropout During Testing

Then randomness of dropout creates a problem during the test phase. We cannot have neurons randomly being discarded during the test phase because that will mean we won’t always get the same result for the same input.

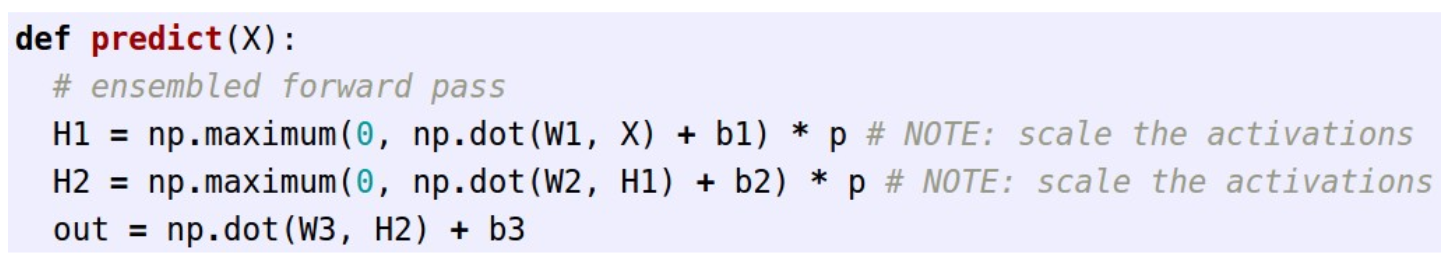
Instead, we use the expected value. The actual expected value is given by an integral that is difficult to compute for arbitrary networks.



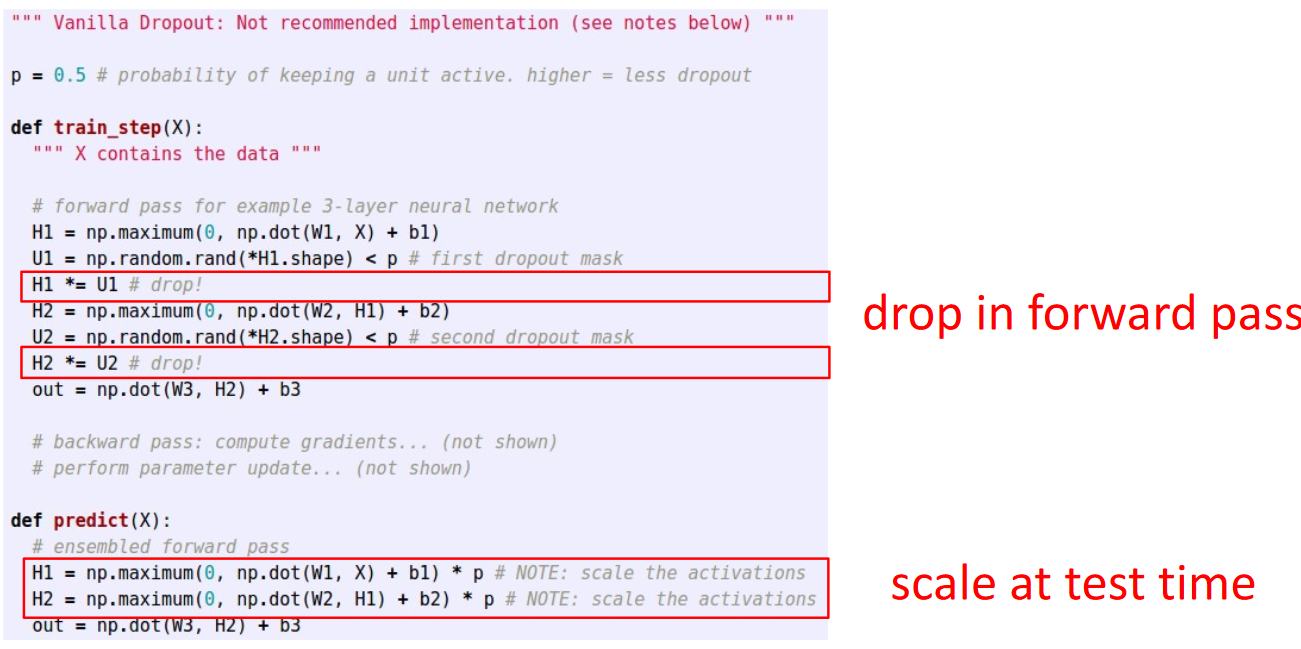
As a compromise, we use an approximation instead. If there are two inputs to a neuron, there are four possible cases: either both inputs are zeroed out, or neither are, or one is but the other isn’t. Thus,



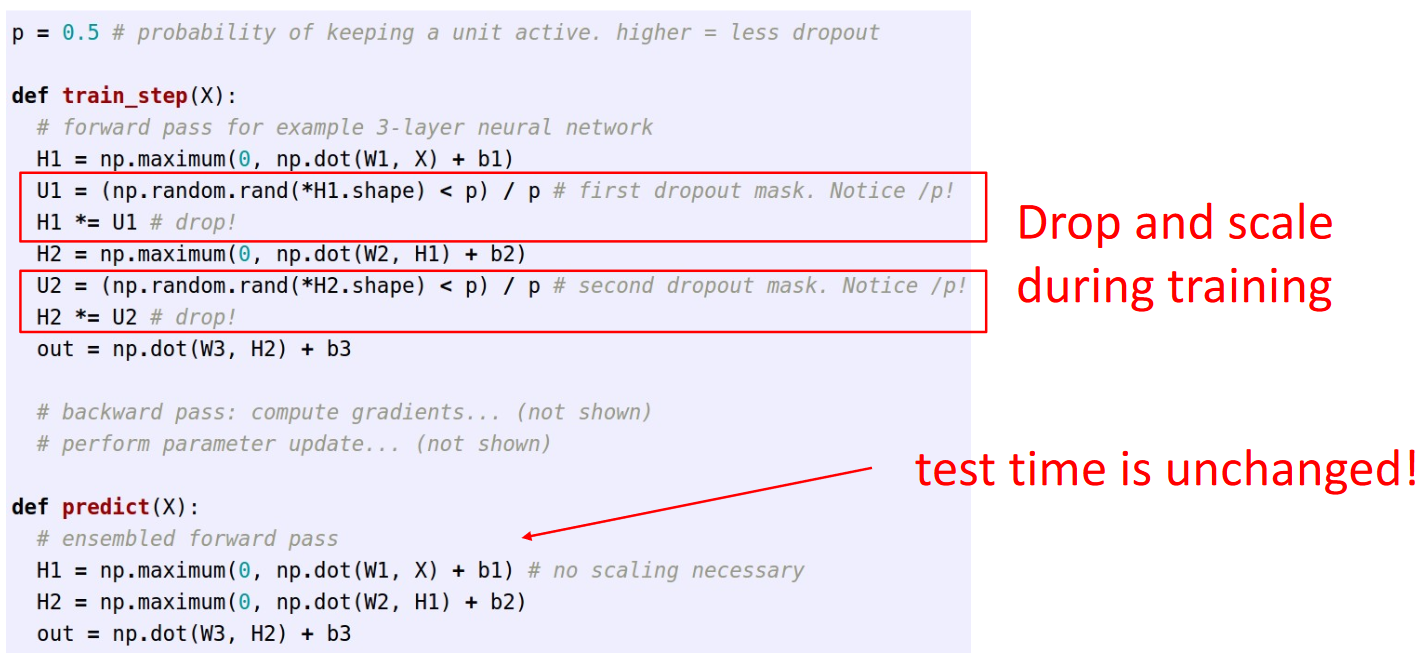
In practice, this means that during the test phase, we **do not drop anything** and **multiply** the output by the **dropout probability**.



In summary, we are dropping values during the training phase and scaling them during the test phase.



In practice however, **inverted dropout** is used. This involves dropping and scaling the output during the training phase and doing nothing during the test phase.



This pattern of adding randomness during training and averaging out the randomness during the test phase is also seen in other regularization techniques. For example, in batch normalization, the outputs from one batch depend on the outputs from all other batches. This adds randomness to the output since it depends on which random elements happen to get shuffled into that mini batch. This is removed during the test phase since the normalization process becomes fixed.

### Dropout Architectures

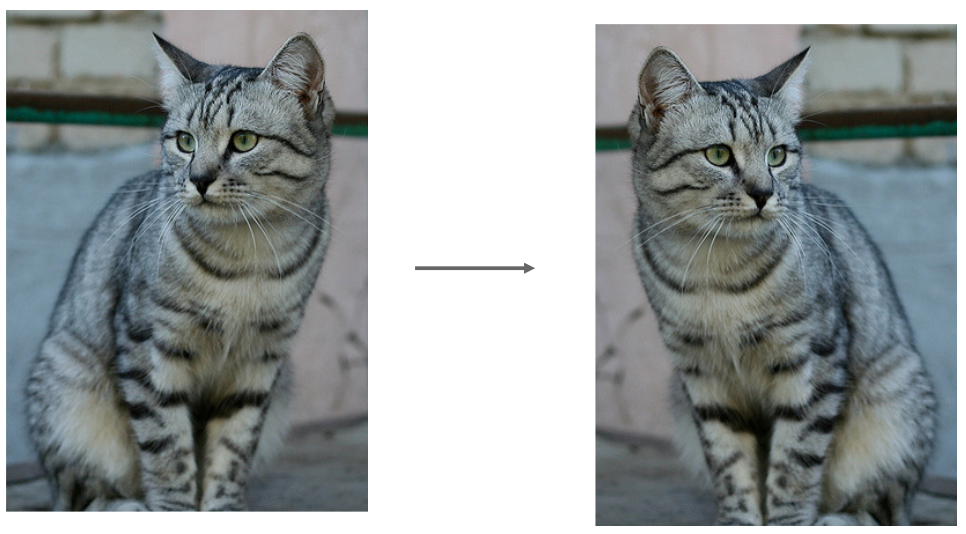
We have seen with AlexNet or VGG-16 that the vast majority of learnable parameters exist in the fully connected layers. This is where dropout is used. Later architectures like GoogLeNet and ResNet use global average pooling and do not use dropout at all.

### Data Augmentation

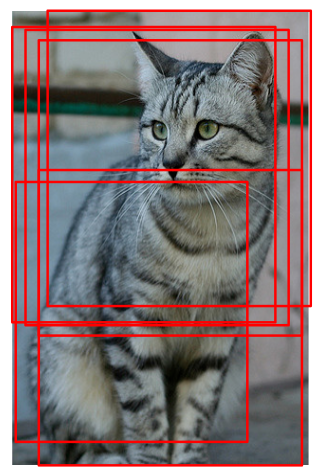
**Data Augmentation** is not generally considered a regularization technique, but since it involves randomly manipulating the data in order to increase the robustness of the model, it can still be classified as regularization.

There are several data augmentation techniques that are commonly used:

* Horizontal Flips



* Random crops and scales, which involves taking a random crop of the image of resizing the image to a random size



Under this technique, we need to use a fixed set of crops of an image and take to average output during the test phase so as to avoid random outputs.

* Color jitter, which involves randomizing the contrast and brightness.

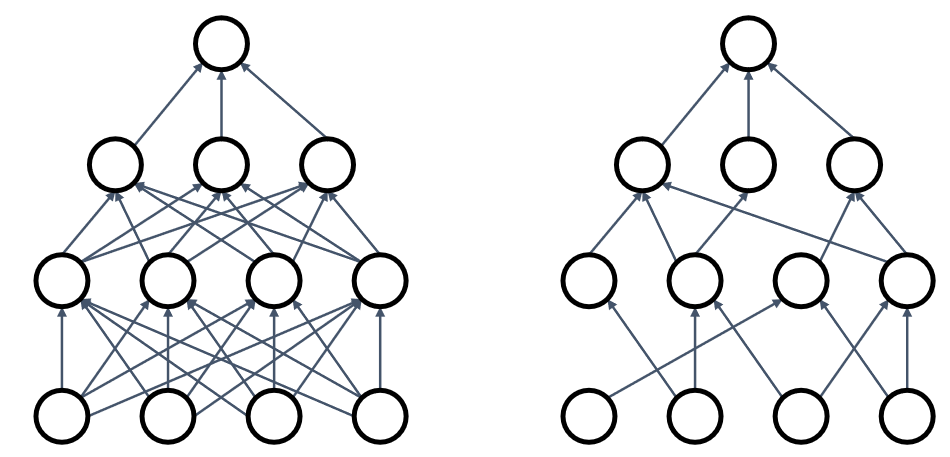


* Other techniques including translation, rotation, stretching, shearing, lens distortion, etc.

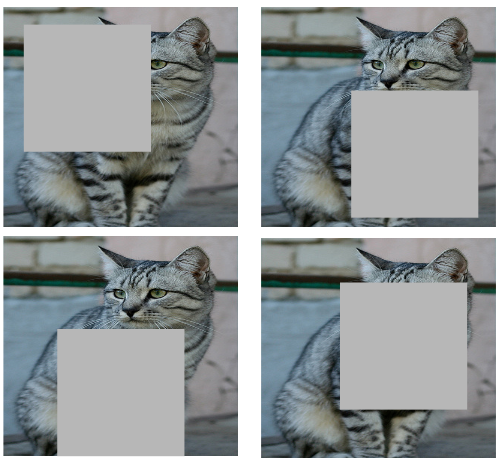
We do however need to be careful about whether the augmentation technique being used makes sense in our context. For example, a model being trained to differentiate between right and left hands would be harmed if we used horizontal flipping as an augmentation technique.

### Other Regularization Techniques

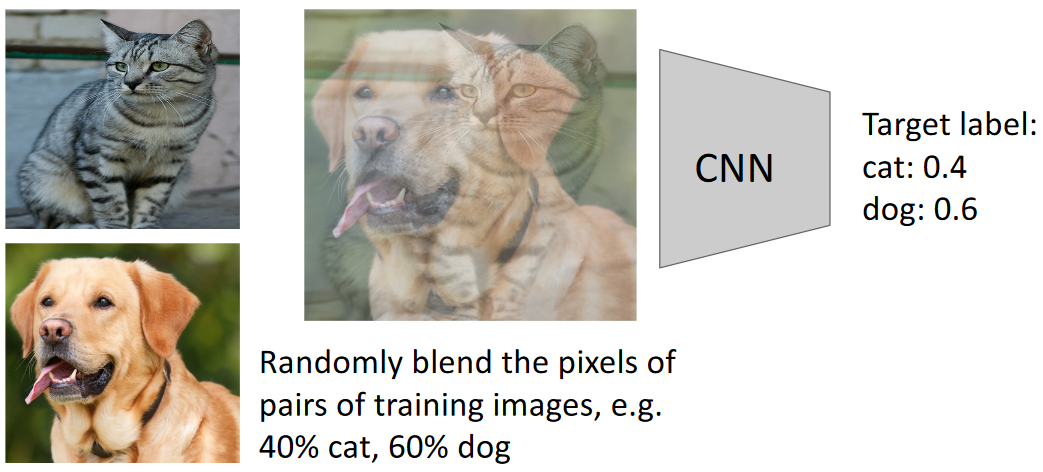
* **Dropconnect** – Dropout, but we remove connections instead of neurons.



* **Fractional Max-Pooling** – Use random sizes of pooling regions for pooling layers.
* **Stochastic Depth** – Skip some blocks in a residual network.
* **Cutout** – Remove a random region from the image.



* **Mixup** – Blend two images at a certain ratio.



Which of these techniques we use depends on our use case. Generally, dropout is good for large fully connected layers, batch normalization and data augmentation are almost always good and cutout and mixup are good for small classification datasets.