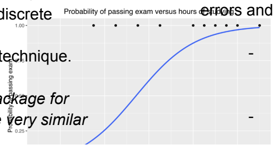
Ch 13 - Neural Networks

Regression attempts to predict a continuous variable, whereas **classification** attempts to predict a categorical / discrete variable.

Exception: Logistic Regression is a classification technique.



Regression Analysis with tidymodels

One of the advantages of using the tidymodels package for "traditional" machine learning is that the steps are very similar for different types of learning.

Step 0: Clean the data.

Identify and remove invalid columns and or data values, as needed.

Count missing values and decide whether you want to exclude or impute.

If excluding, remove entire row (before machine learning).

If imputing, wait until later.

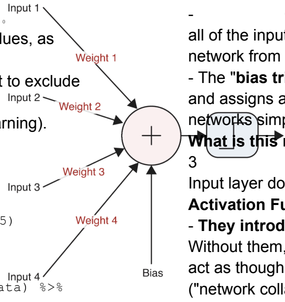
The target (predicted) variable must be numeric.

Step 1: Separate into training and test sets.

```
set.seed(33)
data_split = initial_split(data, prop = 0.75)
training_data = training(data_split)
test_data = testing(data_split)
```

Step 2: Create the recipe.

```
recp = recipe(Target ~ ., data = training_data) %>%
  step_X(...) %>%
  prep(training = training_data)
```



Step 3: "Bake" the training data.

```
training_data = bake(recp, new_data = NULL)
```

Step 4: "Bake" the test data.

```
test_data = bake(recp, new_data = test_data)
```

Step 5: Configure and fit the model.

```
mod = rand_forest(trees = 10, mode = "regression") %>%
  set_engine("ranger") %>%
  fit(Target ~ ., data = training_data)
```

Many algorithms used for **classification** have been adapted for **regression**.

Step 6: Make predictions.

```
predictions = predict(mod_rf, new_data =
test_data_baked)
```

Step 7: Combine predictions with actual outputs for the test set.

Select actual outputs from the test-set tibble.

Combine those with the model's predictions.

Convert to a tibble (if necessary).

Step 8: Calculate metrics

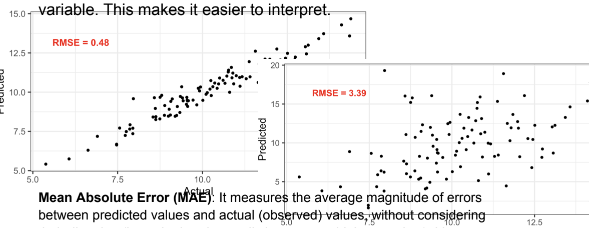
```
actual_pred %>%
  metrics(truth = Target, estimate = .pred) %>%
  print()
Output:
# A tibble: 3 × 3
  .metric .estimator .estimate
  <chr>   <chr>          <dbl>
1 rmse    standard       0.897
2 rsq     standard       0.786
3 mae     standard       0.522
```

Regression Metrics

Root Mean Squared Error (RMSE): a commonly used metric to evaluate the performance of a regression model by measuring the difference between predicted and actual values.

```
sqrt(mean((actual - predicted)^2))
```

- The larger the RMSE, the larger the average difference between predicted and actual values.
- The squaring step makes RMSE more sensitive to larger errors because squaring amplifies larger differences. Models with fewer large deviations between predicted and actual values will tend to have a lower RMSE.
- After squaring the differences and taking their mean, the square root step returns RMSE to the same units as the target variable. This makes it easier to interpret.



Mean Absolute Error (MAE): It measures the average magnitude of errors between predicted values and actual (observed) values, without considering their direction (i.e., whether the prediction was too high or too low). It's a way to quantify how far off predictions are, on average, in real-world units.

```
mean(abs(predicted - actual))
```

MAE Characteristics

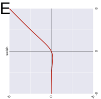
Both are easy to interpret. Both are in the same units as the target variable.

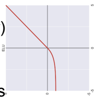
MAE is less sensitive to outliers. Unlike RMSE, which gives more weight to larger errors (because errors are squared), MAE treats all errors equally. This makes it useful when you want to focus on the overall accuracy of the model without overly penalizing large deviations.

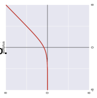
R-squared (R^2 or coefficient of determination): It represents the proportion of the variance in the dependent (target) variable that is explained by the independent (predictor) variables in the model.

ex) Suppose your R^2 value is 0.65. That would tell you that, according to the data you have collected, these three predictors (temperature, humidity, and nest elevation) explain 65% of the variation that we see in egg hatchability for this species. We estimate that other (unmeasured) factors explain the other 35%. These might include factors such as predation, parental behavior, or genetic differences between the birds.

Mean Squared Error (MSE). Identical to RMSE other than the square-root step at the end. Commonly used as a loss function in neural networks (rather than RMSE) because it is simpler and is more easily differentiable. But it is not as easily interpretable.



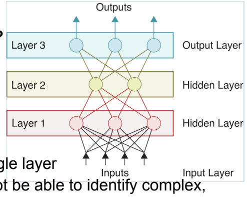




Huber Loss. A combination of MSE and MAE, providing robustness to outliers by applying squared error for smaller errors and absolute error for larger errors.

Bias term:

- Provides a way for a neural network to adapt its behavior, adjusting the network in complement to the input values.
- It increases the flexibility of a neural network, allowing it to learn a wider variety of patterns.
- One particular scenario where it is useful is when all of the inputs to a neuron (unit) are zero. The bias keeps the network from getting "stuck."
- The "**bias trick**" treats the bias as another input, sets it to 1, and assigns a weight to it. This makes the math behind neural networks simpler.



What is this network's "depth"?

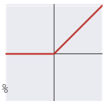
3

Input layer doesn't count

Activation Functions:

- **They introduce non-linearity.** Without them, all networks would act as though they had only a single layer ("network collapse") and would not be able to identify complex, real-world patterns.
- **They control whether a neuron "fires" (becomes active) based on its input.** This enables a model to focus on important patterns while ignoring irrelevant details.
- **Squash outputs to a specific range**. For example, the **sigmoid** function converts numeric outputs to probabilities for binary classification analyses.

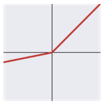
Kinds:



ReLU -

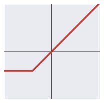
- Loosely inspired by biological neurons, which either "fire" (activate) when they receive a strong enough signal or remain inactive when the signal is too weak.
- **Supports computational efficiency**. In many layers of deep neural networks, a large portion of the neurons may output 0, which reduces the number of active computations and speeds up the model.
- **Computationally simple:**

```
y = max(0, x)
```



Leaky ReLU -

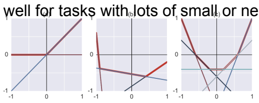
- Addresses one of ReLU's major limitations: "dead neurons" can become permanently inactive.
- The network can still learn something from negative signals.



Shifted ReLU -

- Also designed to avoid "dead neurons."
- Often works well for tasks with lots of small or near-zero inputs.

Maxout →



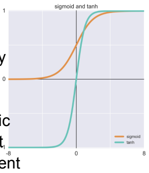
- Can account for more intricate relationships because it does not rely on a single line.
- Avoids dead neurons because it doesn't get "stuck" as easily.
- More computationally intensive and prone to overfitting.

Smooth Variations of ReLU -

(softplus - ELU - Swish)

- Similar to regular ReLU functions, these avoid the vanishing gradient problem—at the cost of a bit more complexity.
- They help prevent the learning process from getting "stuck." They are often suitable for very deep networks.

Activation Functions common for output layer:



Sigmoid converts outputs to probabilities. Used for binary classification. (Softmax is typically used for multiclass classification.)

Hyperbolic tangent (tanh) "squashes" inputs to a specific range. For example, you might want to zero-center output values and limit the range to [-1, 1]. Often used in Recurrent Neural Networks.

Ch 14 - Backpropagation

Regression Analysis with keras

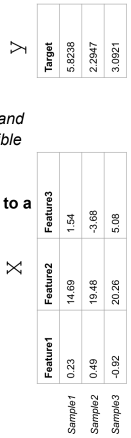
Step 0.1: Set the random seed.

```
tensorflow::set_random_seed(0)
```

This function sets various seeds for us behind the scenes and ensures that Keras and TensorFlow execute in a reproducible manner.

Steps 1-4 from previous chapter

Step 5: Separate features and labels and convert them to a numeric matrix (X) and a vector (y).



y

	Target
Sample1	5.8238
Sample2	2.2947
Sample3	3.0921

X

	Feature1	Feature2	Feature3
Sample1	0.23	14.69	1.54
Sample2	0.49	19.48	-3.68
Sample3	-0.92	20.26	5.08

Step 6: Calculate the input and output dimensions.

Input dim = # of columns in X.

Output dim = 1 for a typical regression analysis

Step 7: Design the model.

Step 8: Implement the model.

```
model = keras_model_sequential() %>%
  layer_dense(units = 64,
              activation = "relu",
              input_shape = input_dim) %>%
  layer_dense(units = 64,
              activation = "relu") %>%
  layer_dense(units = output_dim)
```

We don't specify an activation function on output layer because we want to keep the outputs as they are.

Step 9: Compile the model.

```
compile(model, optimizer = "rmsprop",
  loss = "mean_squared_error",
  metrics = "mean_absolute_error")
```

Step 10: Fit the model.

```
history = model %>% fit(
  x = X_train,
  y = y_train,
  epochs = 50,
  batch_size = 32,
  validation_split = 0.2)
```

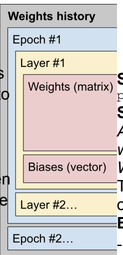
Step 11: Make predictions for test set.

```
predictions = predict(model, X_test)
```

Step 12: Evaluate the model.

After a Keras model has been fit (trained), we can retrieve the weights.

We can also get the weights and biases for all epochs.



The weights and biases for all epochs are packaged ← as a list of lists of lists.

Backpropagation

- Backpropagation helps neural networks learn from their mistakes. When a neural network makes a prediction, that prediction is compared to the correct answer. The extent to which it was wrong is known as the **loss**. Backpropagation sends this information backward through the network, layer by layer, to adjust the weights between the neurons. These adjustments often make the network's future predictions more accurate.
- The more influence a neuron had in creating the loss, the more its weights are adjusted. This process repeats many times, gradually improving the network's ability to make better predictions.

Weights

```
Growth Rate = ATP Concentration * β1
            + Nutrient Availability * β2
            + mTOR Expression * β3
            + β0
```

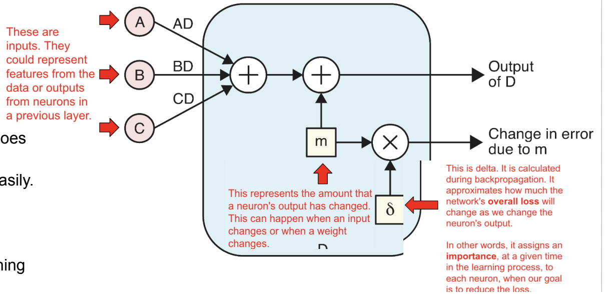
You would use the training data to find the coefficients / weights: β1, β2, β3, β0. These approximate how much each factor (ATP, nutrients, mTOR) affects a cell's growth rate.

β0 is the intercept or bias term. It represents the baseline growth rate when all factors are zero.

- In neural networks, weights perform a similar function, but there are more of them, and they are tied to neurons, which are structured in layers. They are adjusted during training to minimize the loss, helping the network learn patterns. More extreme weights (positive or negative) have a stronger influence on the outputs, whereas weights closer to zero have a weaker impact on the output.
- The end goal is to create a model that has weights that will be effective for performing a particular task (classification, regression, etc.) with new data.

Weights: They are multiplied by the inputs

A * AD + B * BD + C * CD

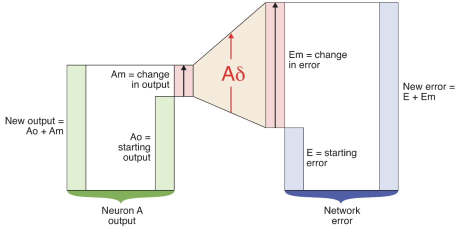


These are inputs. They could represent features from the data or outputs from neurons in a previous layer.

This represents the amount that a neuron's output has changed. This can happen when an input changes or when a weight changes.

This is delta. It is calculated during backpropagation. It approximates how much the network's **overall loss** will change as we change the neuron's output.

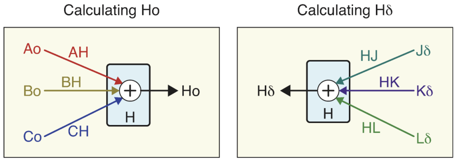
In other words, it assigns an **importance**, at a given time in the learning process, to each neuron, when our goal is to reduce the loss.



δ = How much does the network's error (loss) change in proportion to a change in the output of a neuron?

For neurons in the output layer, delta is directly related to the difference between the predicted output and the true label.

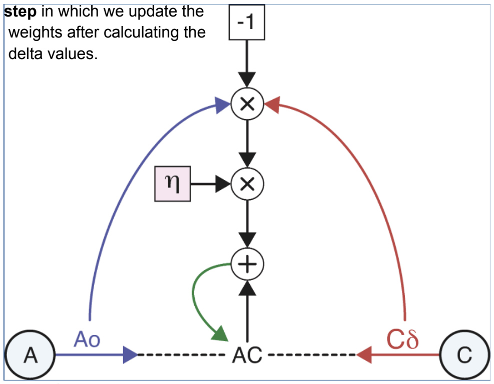
For neurons in hidden layers, delta is calculated based on the deltas of the neurons in the following layer (closer to the output layer), adjusted by the weights between them.



Forward pass. Calculate outputs for a layer based on its inputs and the weights connecting the inputs to the layer's neurons.

Backpropagation. Calculate the delta for each layer based on the deltas for the following layer and the weights connecting the two layers.

This is the **optimization step** in which we update the weights after calculating the delta values.



```
new_weight =
  current_weight -
  learning_rate * delta * input
```

The **learning rate (η)** controls how large the updates are, independently of the neurons. The user configures this. It may or may not be adjusted (automatically) throughout the training process

The **delta** values influence the size of weight updates and are specific to each neuron. They are calculated during backpropagation and change over time as the model learns.

The "**input**" is the output of a neuron in the preceding layer (closer to the input layer).

Ch 15 - Optimizers

The **learning rate** controls how fast or slow a neural network adjusts its weights during training. It balances the speed of convergence against the risk of overshooting or instability.

Finding the right learning rate is critical for effective training.

Downside of using a constant learning rate?

It often results in a failure to "converge" (settle into a minimum).

This typically gets worse as the learning rate increases.

It can also cause us to miss the global minimum.

What does the **decay parameter** control?

It indicates how much to drop the learning rate after each epoch.

Batch gradient descent changes the learning rate after each epoch (not after processing each sample). It typically produces a smooth error curve but is rarely used in practice.

Stochastic gradient descent (SGD) changes the learning rate after processing each sample. It gets its name because it processes each sample in a random (stochastic order). The error typically "jumps around" a lot.

Mini-batch gradient descent changes the learning rate after each batch of samples (a compromise between batch and sample processing). Each mini-batch can be processed in parallel. It helps smooth out the error curve.

Specifying a batch size ensures that any optimizer will process samples in batches.

```
history = model %>% fit(
  x = X_train,
  y = y_train,
  epochs = 50,
  batch_size = 32,
  validation_split = 0.2)
```

Make sure the batch size is smaller than the number of training samples.

Incorporating **momentum** into gradient descent?

When calculating a weight at a particular step, it accounts for how much the weight was updated in the previous step. This can help the model move in the direction of consistent gradients.

- The model moves more quickly when the gradients consistently point in the same direction.
- It can help push through plateaus and escape local minima.
- It can help the model settle into the global minimum, without overshooting it.

Nestorov momentum predicts what the gradient will be in the next step. This way it can avoid moving too far when close to a minimum.

a. Take a step. Move in the current direction with momentum from previous updates.

b. Instead of calculating the gradient at position B, predict the next position (P) using the current momentum.

c. Find gradient of the loss at the predicted location.

d. Make an adjustment based on this gradient. Use the gradient to correct the direction, refining our update. This new location should be better than if we had just used the regular momentum.

Optimizers

Adagrad (Adaptive Gradient) adapts the learning rate for individual weights based on how frequently each gets updated.

It keeps track of the cumulative sum of squared gradients for each weight.

The more a weight is updated, the more its corresponding sum increases, which reduces the learning rate.

This process can slow the learning process too much, causing the model to stop learning before reaching the optimal solution.

Instead of using the cumulative sum of all past squared gradients, **Adadelta** computes a **moving average** of the squared gradients, making the learning rate more stable over time.

RMSprop is conceptually similar to Adadelta.

Adam (Adaptive Moment Estimation) is one of the most popular optimizers. It combines ideas from Adagrad and Adadelta, adding improvements for faster and more reliable learning.

Adam maintains separate moving averages of both the gradients and the squared gradients. This enables it to keep track of the direction of the *gradients*, as well as their *magnitudes*.

```
model %>% compile(
  optimizer = "rmsprop",
  loss = "mean_squared_error",
  metrics = "mean_absolute_error")
```

AdaMax. It's like Adam but uses the maximum gradient instead of averaging squared gradients, making it more stable when gradients are very large.

FTRL (Follow-The-Regularized-Leader). Best for sparse data (most values are zero or missing), it adjusts learning rates aggressively and uses regularization to maintain sparsity.

NAdam. Combines Adam with Nesterov momentum, giving it a faster and more stable convergence.

Keras does not have an option to automatically select the optimizer. But this can be done with other packages.

How does **dropout** work in a neural network?

- **Dropout** randomly disables neurons during training, setting their outputs to zero.

- This prevents the network from becoming overly reliant on specific neurons and encourages all neurons to learn useful features (to avoid overfitting).

- During testing or prediction, all neurons are back in, making the network more robust.

```
layer_dropout(rate = 0.5) %>%
```

Dropout layers are not really layers, meaning that they do not add new neurons or weights to a network.

Batch normalization standardizes the outputs of neurons within each mini-batch of training, making sure they have a consistent range.

- It adjusts and scales neuron activations to ensure they're not too high or too low.

- This helps the model train faster and more reliably, preventing extreme variations in neuron outputs that could slow down learning or cause instability.

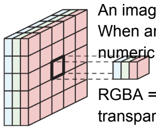
```
layer_batch_normalization() %>%
```

Batch normalization layers are **not layers either**.

Ch 16 - Convolutional Neural Networks

A **pixel** (short for picture element) is a single point in an image.

An image has one or more **channels**.



When an image has multiple channels, a pixel consists of numeric values across all channels at that location.

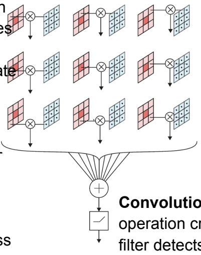
Grayscale = 1 ; Color (RGB) = 3

RGBA = 4 - The "alpha" channel indicates the level of transparency.

Multispectral / hyperspectral images = 4+

Often used in satellite images, medical images, and microscopy.

Apply a 3x3 **filter** to an image pixel?



- The filter is centered on the target pixel.
- Each of the 9 filter values is multiplied by the corresponding pixel values.
- These values are summed.
- The sum replaces the original pixel value in the output.
- An activation function may perform an additional transformation.
- Repeat process for each channel

Convolution. Sweep filter across an entire image. This operation creates a feature map, which represents how well the filter detects certain features (like edges or textures) across the entire image.

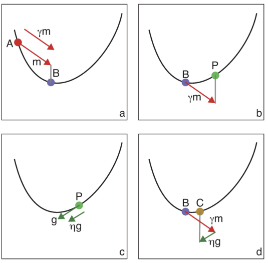
We use **multiple filters** within a single convolutional layer because Each filter acts like a feature detector that identifies a specific pattern in the input.

Having many filters makes a model more versatile and capable of identifying complex patterns in images.

Why use **multiple filters** within a single convolutional layer?

Each filter acts like a ***feature detector*** that identifies a specific pattern in the input.

Having many filters makes a model more versatile and capable of identifying complex patterns in Images.



The filters within a convolution layer are processed separately. Thus, they can be executed in parallel.

If a convolutional layer has 10 filters, it will produce 10 feature maps (one for each filter). The feature maps are then stacked together together to form a multi-dimensional output tensor.

The numbers ("kernel values") in a filter can be defined manually, but typically they are learned during the training.

$$\begin{bmatrix} -1 & 1 & -1 \\ -1 & 1 & -1 \\ -1 & 1 & -1 \end{bmatrix} \times \begin{bmatrix} 1 & 1 & 0 \\ 0 & 0 & 0 \\ 1 & 0 & 1 \end{bmatrix} = \begin{bmatrix} -1 & 1 & 0 \\ 0 & 0 & 0 \\ -1 & 0 & -1 \end{bmatrix} \Rightarrow -3 + 1 = -2$$

Preparing Images for Keras

- All images in a collection might not be the same size and/or they might not have the dimensions we think would be best to use as input to a neural network.

- Keras can resize them for us.

Step 1: Define the target image size and batch size.

```
img_width = 256
img_height = 256
batch_size = 32
```

The metadata can use any tabular format (such as .csv, .tsv, Excel).

It must include a column that indicates the name of each file.

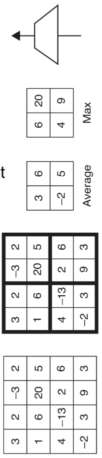
It must include a column that indicates the class of each sample (image).

Step 2: Indicate where files are stored.

Step 3: Read each annotation file into a tibble.

Step 4: Indicate how to scale the images between 0 and 1.

```
image_gen = image_data_generator(
  rescale = 1/255 )
```



Step 5: Define how to retrieve and process training images.

```
training_generator = flow_images_from_dataframe(
  generator = image_gen,
  dataframe = training_annotations,
  directory = training_dir,
  x_col = "filename",
  y_col = "class",
  batch_size = batch_size,
  target_size = c(img_height, img_width),
  class_mode = "categorical"
  color_mode = "grayscale" )
```

When images are resized, their dimensions may not scale directly to the target size (for example 128x128 to 64x64).

Interpolation adjusts the pixel values so that the resized images retain as much of the original information as possible.

"**nearest**" is the default in Keras. For pixels that need to be interpolated, it assigns the value of the nearest pixel.

"**bilinear**" takes a weighted average of the four nearest pixels to calculate the new pixel value. It provides a smoother result.

"**bicubic**" uses a 4x4 grid of pixels around the target location to figure out the new pixel value.

Steps 6 & 7: Define how to retrieve and process the validation and test images.

Step 8: Calculate the output dimensions (the model will use one-hot encoding based on the number of classes).

```
output_dim=length(unique(train_generator$class_indices
))
```

Step 9: Define the model - First layer

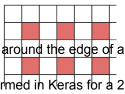
← **Pooling (downsampling)** shrinks the output size of an input tensor. This has a "blurring" effect, which allows for subtle differences in images and thus more flexibility in identifying patterns.

It is typical to use **either** pooling or striding, but not both.

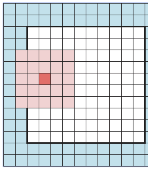
A benefit of striding is that it is part of the convolution step; therefore, it is more efficient (but not necessary more effective) than pooling.

Striding (alternative to pooling)

`strides = c(2L, 2L)` →



← **Padding** places zero values around the edge of an image so that filters can be applied to all pixels.



By default, padding is ***not*** performed in Keras for a 2D convolutional layer. Without padding, convolution cannot extend beyond the edges of input images.

This causes the output feature map to be smaller than the input. For example, applying a 3x3 filter without padding reduces both width and height by 2 pixels.

One time this is useful is when you want the model to focus more on the central part of an image.

There is no standard way to design a convolution neural network. But a common way is to **increase the number of filters across multiple layers** before performing regularization.

Early layers with fewer filters typically capture basic features like edges, lines, and corners.

Deeper layers, with more filters, capture more complex features like textures, shapes, or specific parts of objects.

Flattening takes a multi-dimensional feature map and converts it to a 1D vector. This makes it possible to use a dense (fully connected) layer with trainable weights.

The dense layer is necessary to make a prediction for each class. Dropout is often performed, too.

Feature Reduction

1x1 convolutions collapse information from multiple filters into fewer filters. The goal is to reduce redundancy while retaining as much useful information as possible.

```
kernel_size = c(1, 1)
```

This means that the filters operate on each pixel individually (not spatially).

Transposed convolution increases the size of an image or feature map. During training, the network learns weights for the upsampling process from all images in the training set. These weights can then be used to increase the resolution of images as they are scaled to larger sizes.

- One way to increase an image's size is to replicate each pixel multiple times.

- Another way is to pad the edges of an image with zeroes. This helps preserve the spatial structure of the image while increasing its dimensions. Essentially, the pixels around the edges of the image provide information for filling in the empty pixels.

- Another form of upsampling is to insert zeroes around and between pixels. This is known as atrous ("with holes") or dilated convolution.

Image Augmentation

- Increases the diversity of the training set.

- Mimics distortions that can naturally occur in images.

- Often helps to make models more robust to minor differences (resulting in less overfitting).

Examples:

- Horiz & Vetr flip, Rotations, Horiz & Vert shift, Zoom in & out, Shear

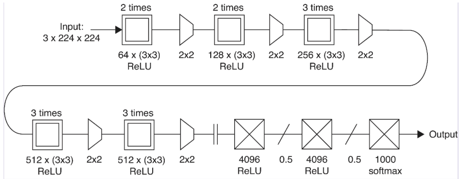
Hierarchy of Filters

To minimize the network's loss, it starts by identifying basic features that are common across the images. It iteratively identifies more advanced structures that are common across images.

By starting with basic building blocks, it can identify patterns that are more generalizable, rather than subtle differences peculiar to one or a few images.

- We use an individual filter to identify a particular (basic) pattern in the image.
- We then pool the feature map to summarize where those basic patterns are found.
- We combine (stack) the pooled feature maps to indicate where all of the basic patterns are found.
- We continue this process for the next layer in the network, building on the patterns identified in the preceding layer.
- This process continues for subsequent layers in the network, as it works toward identifying more complex structures.

Typically, the number of filters per layer increases (often by a factor of two). Layers with the same number of filters may be repeated. This is true for **ImageNet**.


Input:
3 x 224 x 224
2 times
64 x (3x3)
ReLU
2x2
2 times
128 x (3x3)
ReLU
2x2
3 times
256 x (3x3)
ReLU
2x2
3 times
512 x (3x3)
ReLU
2x2
3 times
512 x (3x3)
ReLU
2x2
4096
ReLU
0.5
4096
ReLU
0.5
1000
softmax
Output