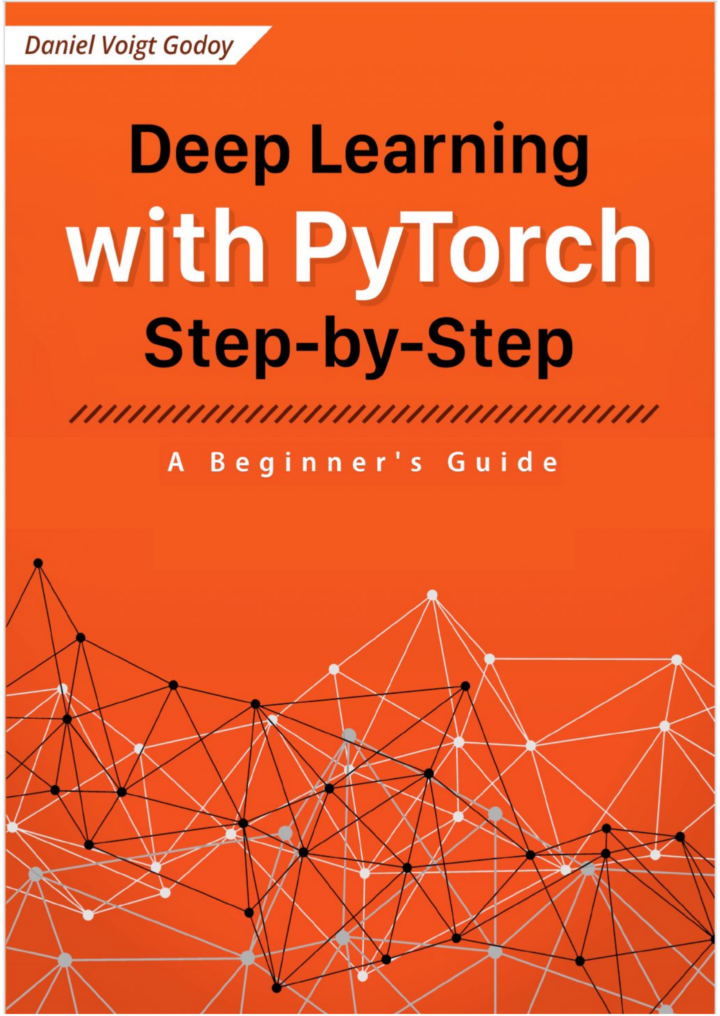
PPGEEC2318

# Machine Learning

*Rock, Paper, Scissors*

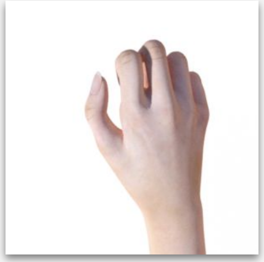
Ivanovitch Silva
ivanovitch.silva@ufrn.br


Daniel Voigt Godoy
Deep Learning
with PyTorch
Step-by-Step
A Beginner's Guide

# Agenda

1. ***Standardize*** *an image dataset*
2. ***train*** *a model to predict* ***rock, paper, scissors*** *poses from hand images*
3. *use* ***dropout*** *layers to* ***regularize*** *the model*

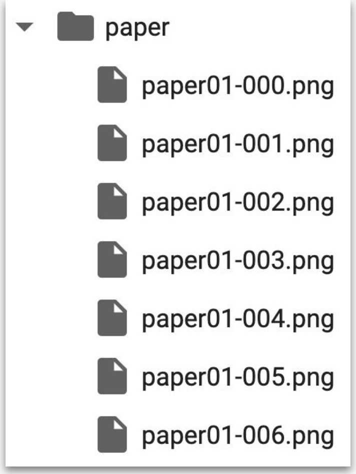
The dataset contains 2,892 images (2,520 train, 372 test) of diverse hands in the typical rock, paper, and scissors poses against a white background. This is a synthetic dataset as well since the images were generated using CGI techniques. Each image is 300x300 pixels in size and has four channels (RGBA).

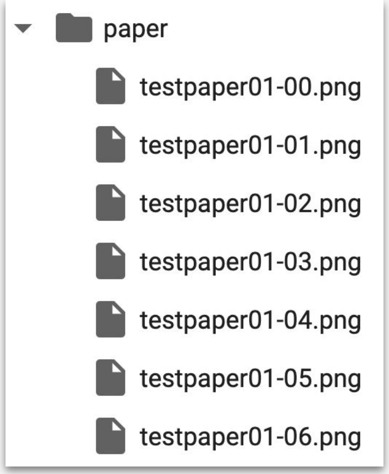


Rock

Paper

Scissors




# If the images are colored

We need to standardize the three channels (RGB)

Find the <mean,std> for each channel

and to limit them to <0,1>

Only for train dataset!!! Avoid data leakage!!!

# Data Preparation

## ImageFolder

```
# Compose a sequence of preprocessing transforms
# 1) Resize images to 28×28 pixels
# 2) Ensure output is a PIL/torchvision Image (dropping any alpha channel)
# 3) Convert pixel values to float32 and scale from [0–255] to [0.0–1.0]
temp_transform = Compose([
    Resize(28),                              # Resize each image to 28×28
    ToImage(),                               # Convert tensor back to PIL Image (enforces RGB)
    ToDtype(torch.float32, scale=True) # Cast to float32 and normalize pixel range
])

# Create an ImageFolder dataset from the 'rps' directory
# Images are grouped by subfolder name as class labels, and each image is transformed
temp_dataset = ImageFolder(
    root='rps',
    transform=temp_transform          # Apply the preprocessing pipeline to every image
)
```

# Data Preparation

## ImageFolder

```python
# Get total number of samples in the dataset
dataset_size = len(temp_dataset)
print(f"Dataset size: {dataset_size} images")

# Get number of classes
num_classes = len(temp_dataset.classes)
print(f"Number of classes: {num_classes}")

Dataset size: 2520 images
Number of classes: 3
```

```python
# the second element of this tuple is the label
temp_dataset[0][0].shape, temp_dataset[0][1]

(torch.Size([3, 28, 28]), 0)

temp_dataset[0][0]

Image([[[1.0000, 1.0000, 1.0000,  ..., 1.0000, 1.0000, 1.0000],
        [1.0000, 1.0000, 1.0000,  ..., 1.0000, 1.0000, 1.0000],
        [1.0000, 1.0000, 1.0000,  ..., 1.0000, 1.0000, 1.0000],
        ...,
        [1.0000, 1.0000, 1.0000,  ..., 0.9961, 0.9961, 0.9961],
        [1.0000, 1.0000, 1.0000,  ..., 0.9961, 0.9961, 0.9922],
        [1.0000, 1.0000, 1.0000,  ..., 0.9961, 0.9922, 0.9922]],

       [[1.0000, 1.0000, 1.0000,  ..., 1.0000, 1.0000, 1.0000],
        [1.0000, 1.0000, 1.0000,  ..., 1.0000, 1.0000, 1.0000],
        [1.0000, 1.0000, 1.0000,  ..., 1.0000, 1.0000, 1.0000],
        ...,
        [1.0000, 1.0000, 1.0000,  ..., 0.9961, 0.9961, 0.9961],
        [1.0000, 1.0000, 1.0000,  ..., 0.9961, 0.9961, 0.9922],
        [1.0000, 1.0000, 1.0000,  ..., 0.9961, 0.9922, 0.9922]],

       [[1.0000, 1.0000, 1.0000,  ..., 1.0000, 1.0000, 1.0000],
        [1.0000, 1.0000, 1.0000,  ..., 1.0000, 1.0000, 1.0000],
        [1.0000, 1.0000, 1.0000,  ..., 1.0000, 1.0000, 1.0000],
        ...,
        [1.0000, 1.0000, 1.0000,  ..., 0.9961, 0.9961, 0.9961],
        [1.0000, 1.0000, 1.0000,  ..., 0.9961, 0.9961, 0.9922],
        [1.0000, 1.0000, 1.0000,  ..., 0.9961, 0.9922, 0.9922]]], )
```

# Data Preparation

## Standardization

```
temp_loader = DataLoader(temp_dataset, batch_size=16)
# Each column represents a channel
# first row is the number of data points
# second row is the the sum of mean values
# third row is the sum of standard deviations
first_images, first_labels = next(iter(temp_loader))
Architecture.statistics_per_channel(first_images, first_labels)

tensor([[16.0000, 16.0000, 16.0000],
        [13.8748, 13.3048, 13.1962],
        [ 3.0507,  3.8268,  3.9754]])


# We can leverage the loader_apply() method to get the sums for the whole dataset:
results = Architecture.loader_apply(temp_loader,
                                    Architecture.statistics_per_channel)

tensor([[2520.0000, 2520.0000, 2520.0000],
        [2142.5356, 2070.0806, 2045.1444],
        [ 526.3025,  633.0677,  669.9556]])
```

# Data Preparation

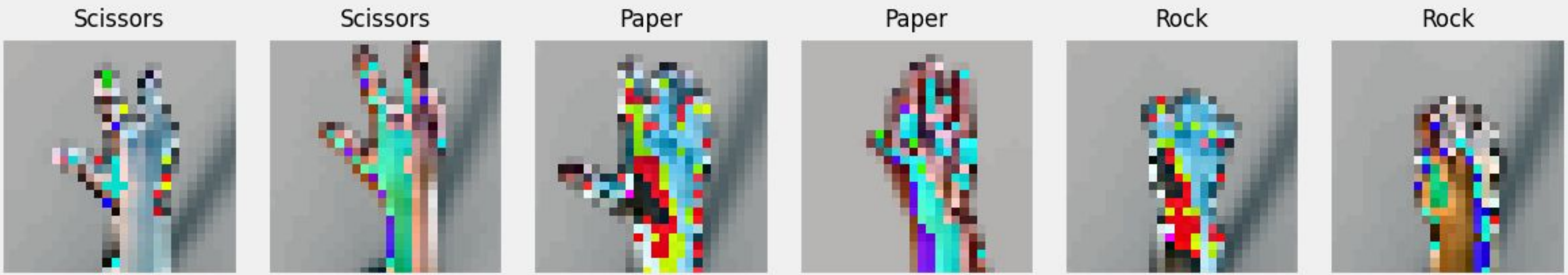
## Standardization

```
temp_loader = DataLoader(temp_dataset, batch_size=16)

# we can compute the average mean value and the average standard deviation, per channel.
# Better yet, let's make it a method that takes a data loader and
# returns an instance of the Normalize() transform
normalizer = Architecture.make_normalizer(temp_loader)
normalizer

Normalize(mean=tensor([0.8502, 0.8215, 0.8116]),
          std=tensor([0.2089, 0.2512, 0.2659]))
```

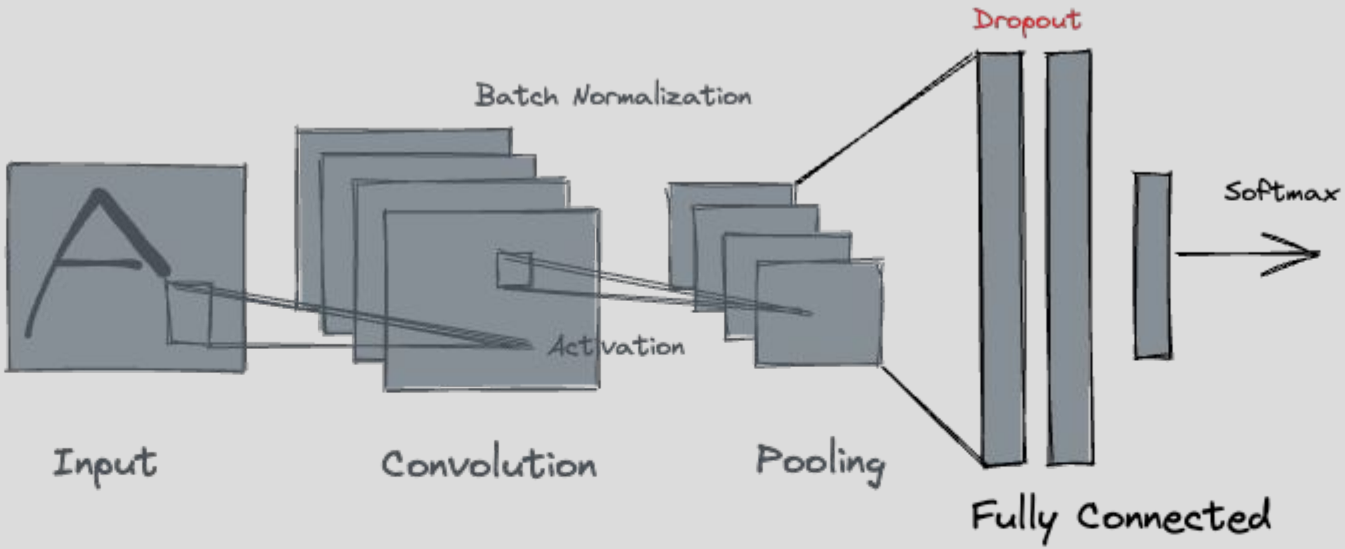
# Data Preparation

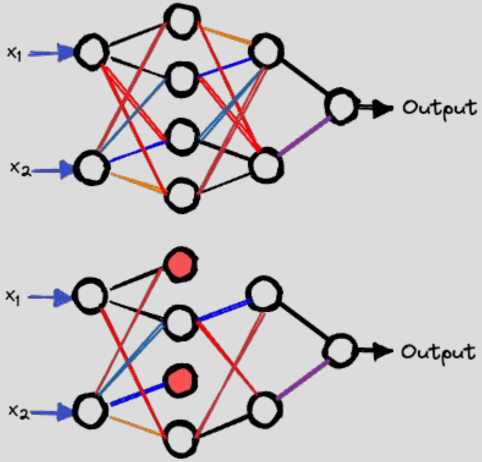
The real dataset

```python
composer = Compose([
    Resize(28),                                    # Resize to 28×28
    ToImage(),                                     # Convert to PIL Image in RGB
    ToDtype(torch.float32, scale=True), # Cast to float32 and normalize to [0,1]
    normalizer                                     # Apply custom normalization transform
])

# Instantiate training and validation datasets from folders:
# - 'rps' contains subfolders per class for training
# - 'rps-test-set' likewise for validation
train_data = ImageFolder(root='rps', transform=composer)
val_data   = ImageFolder(root='rps-test-set', transform=composer)

# Wrap datasets in DataLoaders for batching and shuffling:
# - batch_size=16 yields mini-batches of 16 images
# - shuffle=True randomizes training order each epoch
train_loader = DataLoader(train_data, batch_size=16, shuffle=True)
val_loader   = DataLoader(val_data,   batch_size=16)  # no shuffle for validation
```



# Dropout




It is a form of regularization
Reduces overfitting
Increases test/validation accuracy (sometimes at expense of training accuracy)
Randomly disconnects node from current layers to next layer with probability, p

# Dropout (what's going on here?)

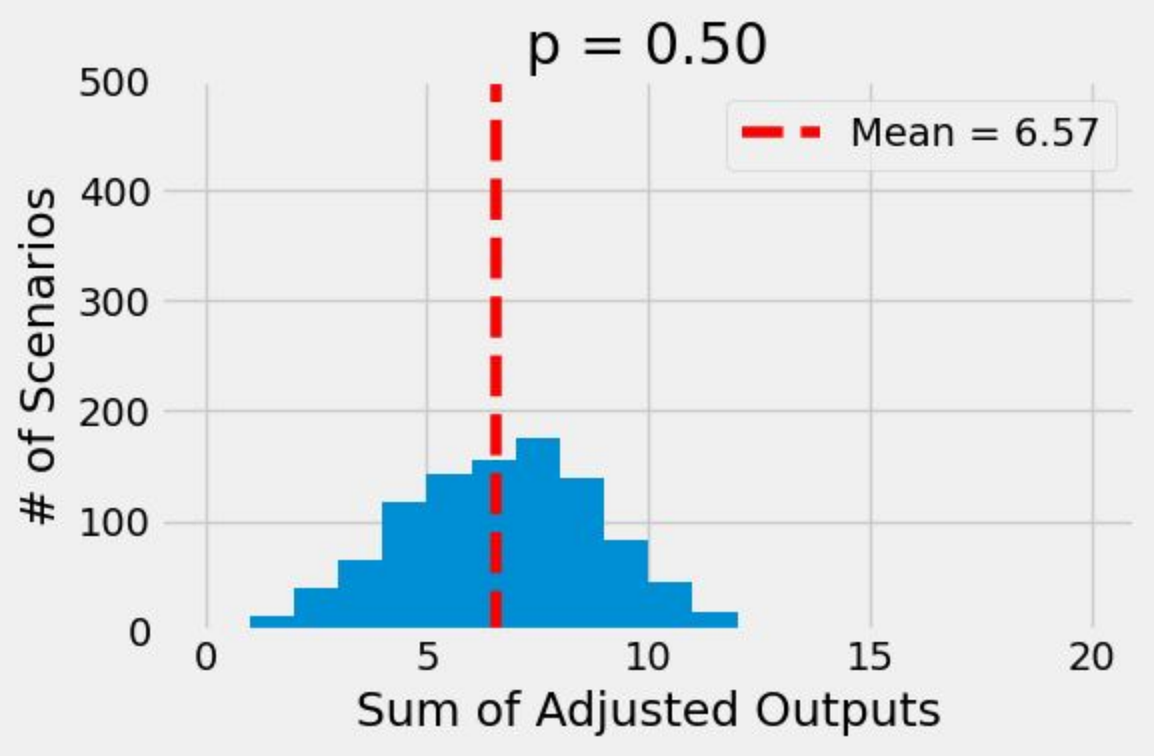
```
dropping_model = nn.Sequential(nn.Dropout(p=0.5))
spaced_points = torch.linspace(.1, 1.1, 11)
spaced_points

tensor([0.1000, 0.2000, 0.3000, 0.4000, 0.5000, 0.6000, 0.7000, 0.8000, 0.9000,
        1.0000, 1.1000])

dropping_model.train()
output_train = dropping_model(spaced_points)
output_train

tensor([0.0000, 0.4000, 0.0000, 0.8000, 0.0000, 1.2000, 1.4000, 1.6000, 1.8000,
        0.0000, 2.2000])
```

# Dropout (what's going on here?)

```
F.linear(output_train, weight=torch.ones(11), bias=torch.tensor(0))
tensor(9.4000)

dropping_model.eval()
output_eval = dropping_model(spaced_points)
output_eval

tensor([0.1000, 0.2000, 0.3000, 0.4000, 0.5000, 0.6000, 0.7000, 0.8000, 0.9000,
        1.0000, 1.1000])

F.linear(output_eval, weight=torch.ones(11), bias=torch.tensor(0))
tensor(6.6000)
```

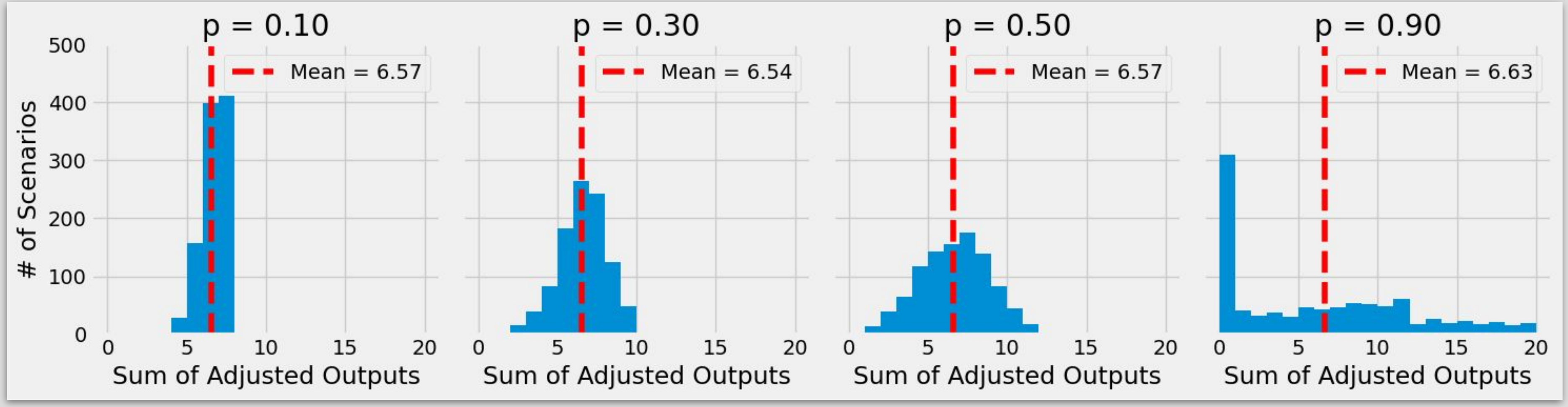
# Dropout

Distribution of 1000 outputs



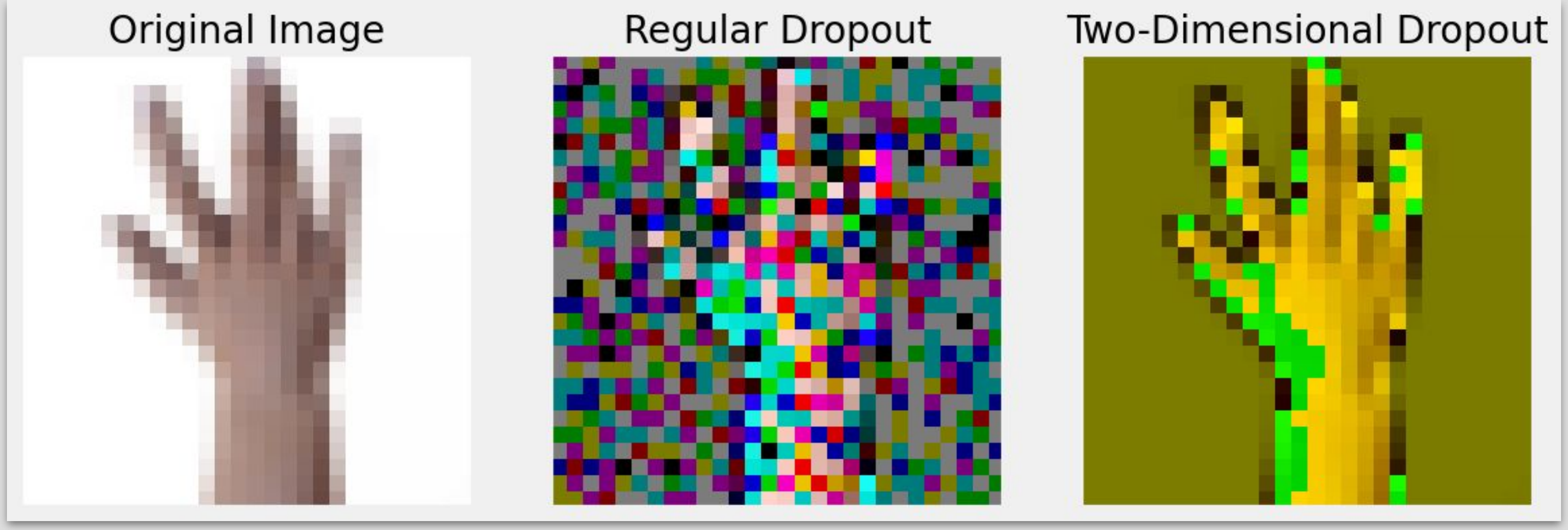
# Dropout

## Distribution of 1000 outputs




- For more typical dropout probabilities (like 30% or 50%), the distribution may take some more extreme values when compared to 10%
- The variance of the distribution of outputs grows with the dropout probability.
- A higher dropout probability makes it harder for your model to learn—that's what regularization does

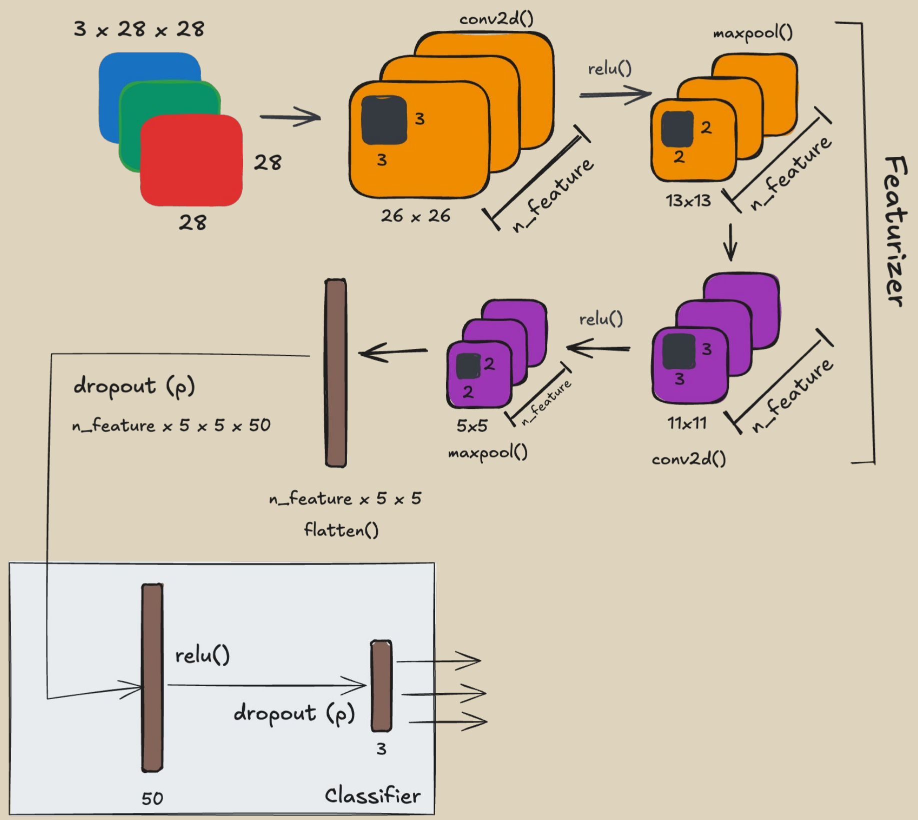
# Two dimensional dropout




- It drops entire channels / filters.
- If a convolutional layer produces ten filters, a two-dimensional dropout with a probability of 50% would drop five filters (on average)
- The remaining filters would have all their pixel values left untouched.

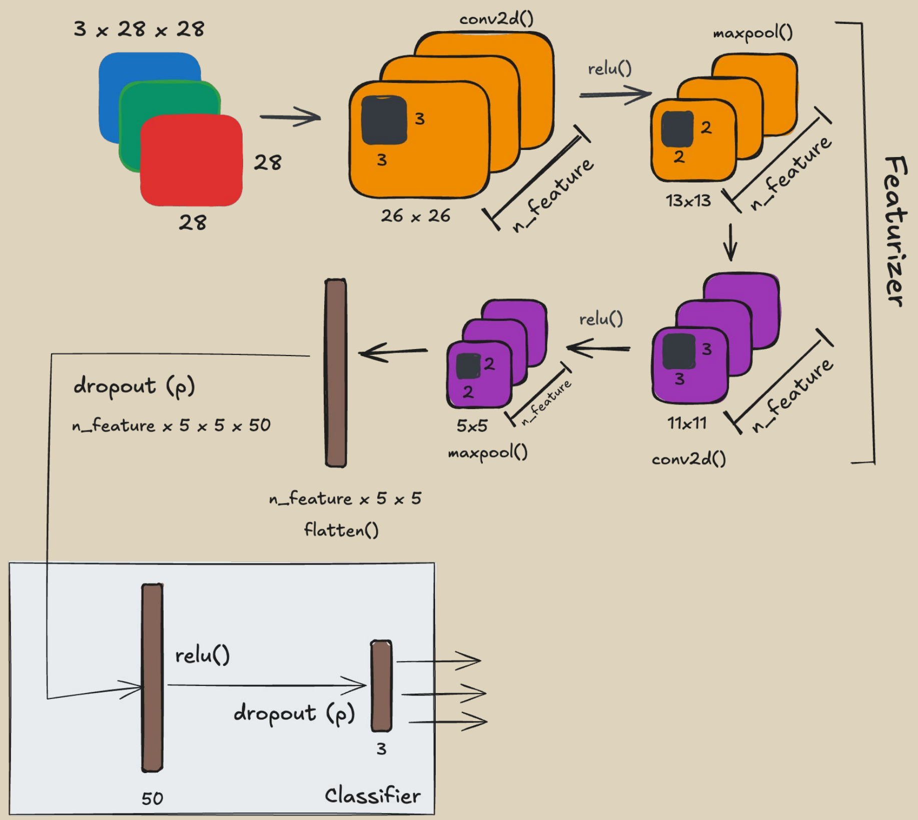
# Fancier Model

```python
class CNN2(nn.Module):
    def __init__(self, n_feature, p=0.0):
        super(CNN2, self).__init__()
        self.n_feature = n_feature
        self.p = p
        # Creates the convolution layers
        self.conv1 = nn.Conv2d(in_channels=3,
                               out_channels=n_feature,
                               kernel_size=3)
        self.conv2 = nn.Conv2d(in_channels=n_feature,
                               out_channels=n_feature,
                               kernel_size=3)
        # Creates the linear layers
        # Where do this 5 * 5 come from?! Check it below
        self.fc1 = nn.Linear(n_feature * 5 * 5, 50)
        self.fc2 = nn.Linear(50, 3)
        # Creates dropout layers
        self.drop = nn.Dropout(self.p)
```



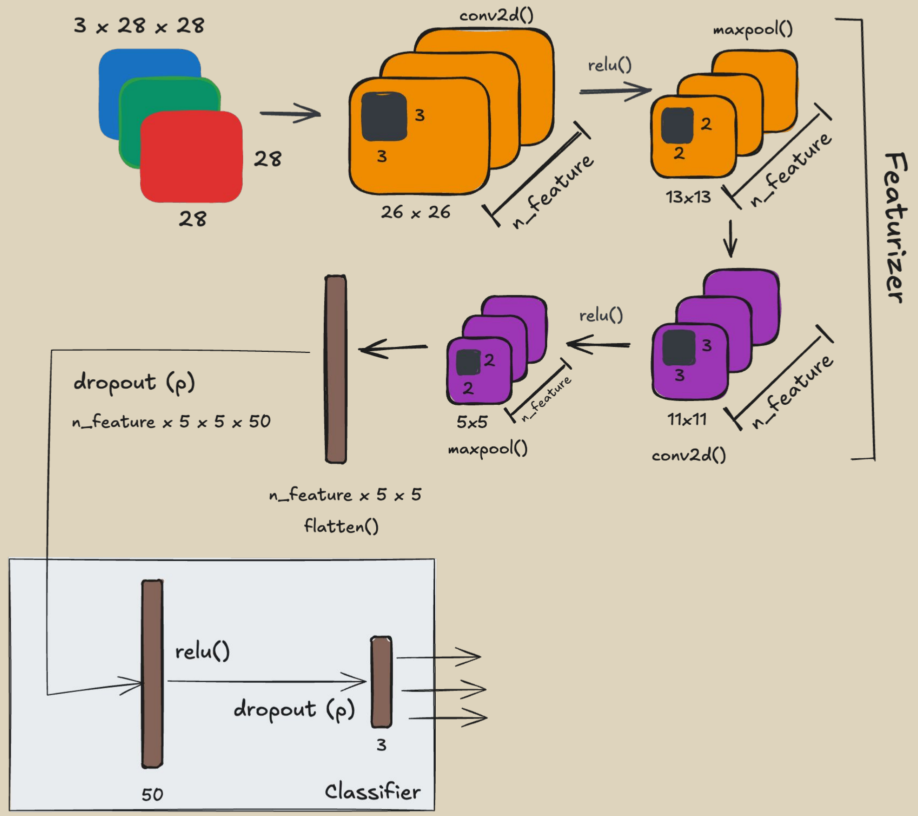
# Fancier Model

```python
def featurizer(self, x):
    # Featurizer
    # First convolutional block
    # 3@28x28 -> n_feature@26x26 -> n_feature@13x13
    x = self.conv1(x)
    x = F.relu(x)
    x = F.max_pool2d(x, kernel_size=2)
    # Second convolutional block
    # n_feature * @13x13 -> n_feature@11x11 -> n_feature@5x5
    x = self.conv2(x)
    x = F.relu(x)
    x = F.max_pool2d(x, kernel_size=2)
    # Input dimension (n_feature@5x5)
    # Output dimension (n_feature * 5 * 5)
    x = nn.Flatten()(x)
    return x
```



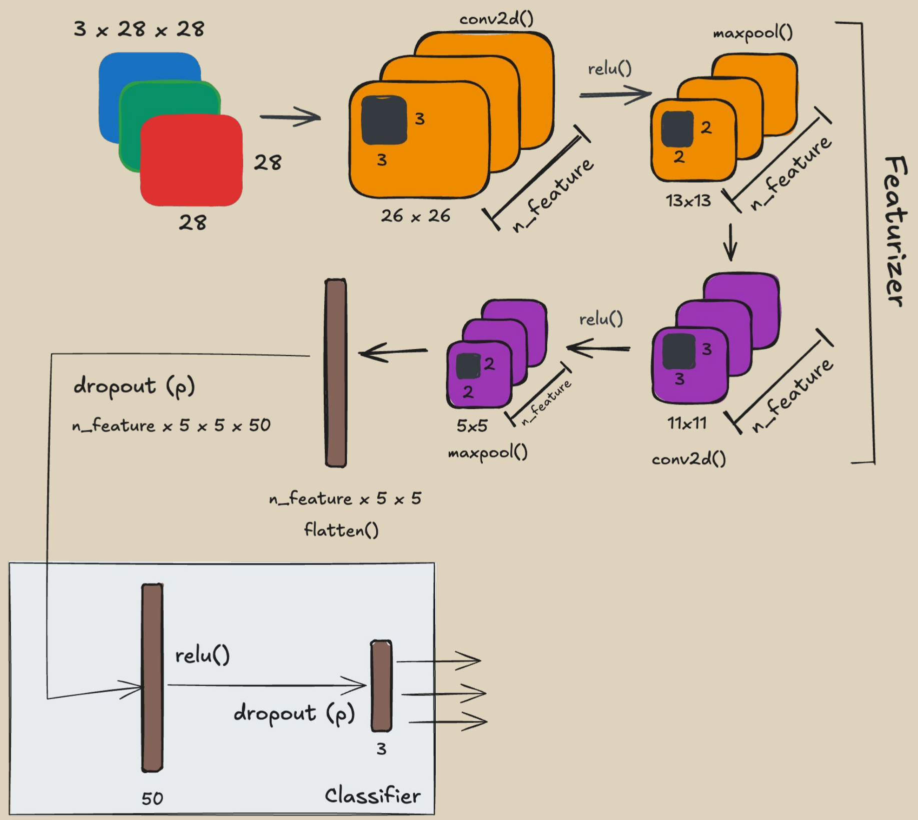
# Fancier Model

```python
def classifier(self, x):
    # Classifier
    # Hidden Layer
    # Input dimension (n_feature * 5 * 5)
    # Output dimension (50)
    if self.p > 0:
        x = self.drop(x)
    x = self.fc1(x)
    x = F.relu(x)
    # Output Layer
    # Input dimension (50)
    # Output dimension (3)
    if self.p > 0:
        x = self.drop(x)
    x = self.fc2(x)
    return x

def forward(self, x):
    x = self.featurizer(x)
    x = self.classifier(x)
    return x
```



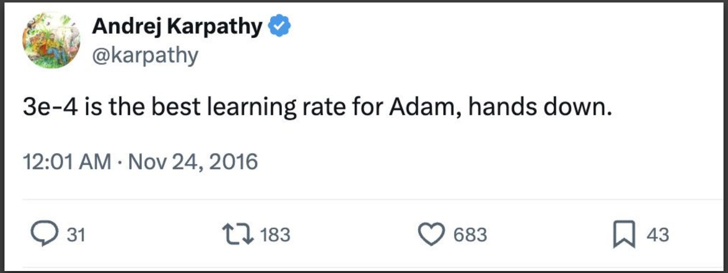
# Case Study

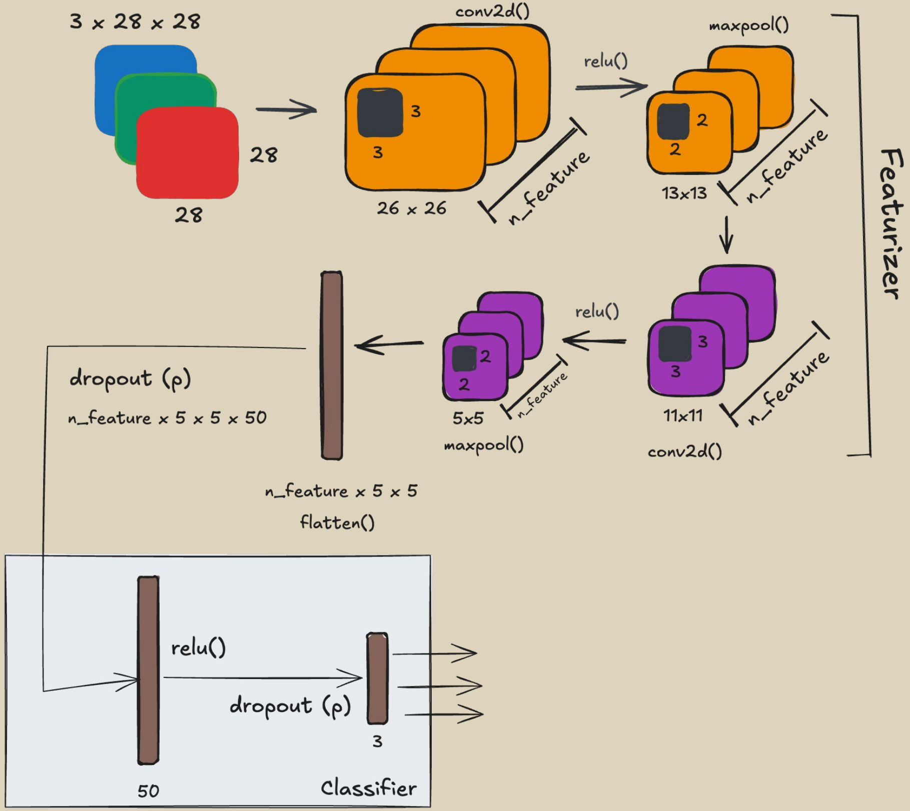
Model Configuration
Model Training
Accuracy
Regularizing Effect
Visualizing Filters



# Model Config.

```python
torch.manual_seed(13)

# Model/Architecture
model_cnn2 = CNN2(n_feature=5, p=0.3)

# Loss function
multi_loss_fn = nn.CrossEntropyLoss(reduction='mean')

# Optimizer
optimizer_cnn2 = optim.Adam(model_cnn2.parameters(), lr=3e-4)
```



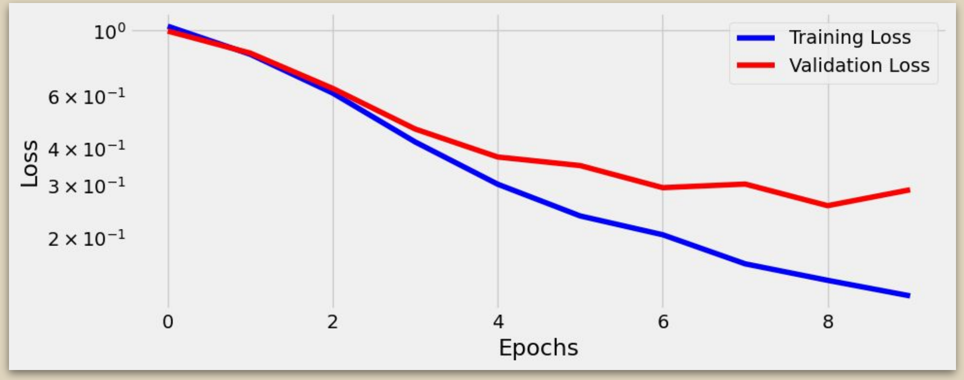



# Model Train

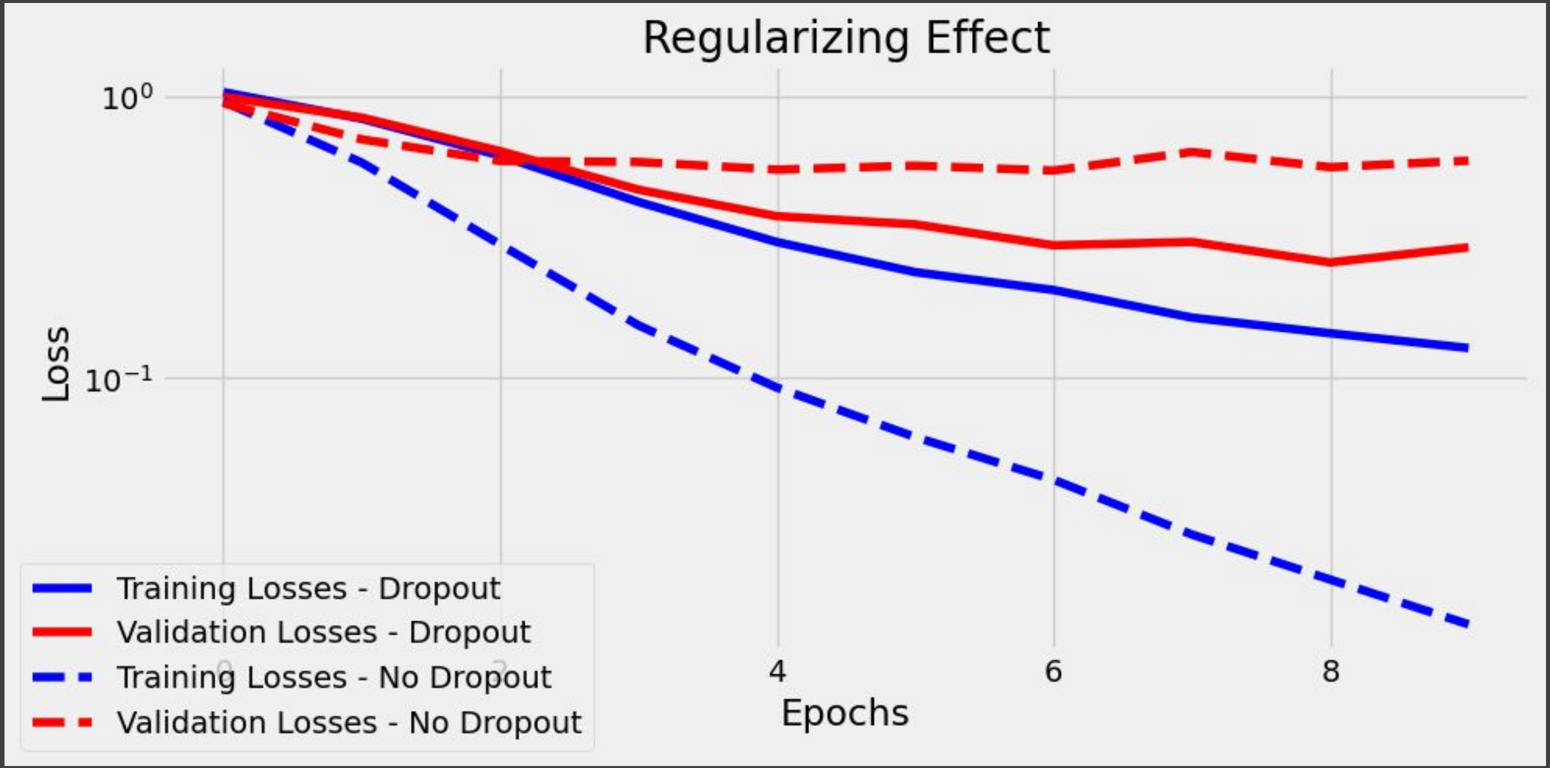
```
arch_cnn2 = Architecture(model_cnn2,
                         multi_loss_fn,
                         optimizer_cnn2)
arch_cnn2.set_loaders(train_loader, val_loader)
arch_cnn2.train(10)

arch_cnn2.count_parameters()
6823
```

```
Architecture.loader_apply(val_loader,
                          arch_cnn2.correct)

tensor([[ 89, 124],
        [118, 124],
        [117, 124]])
```

87%



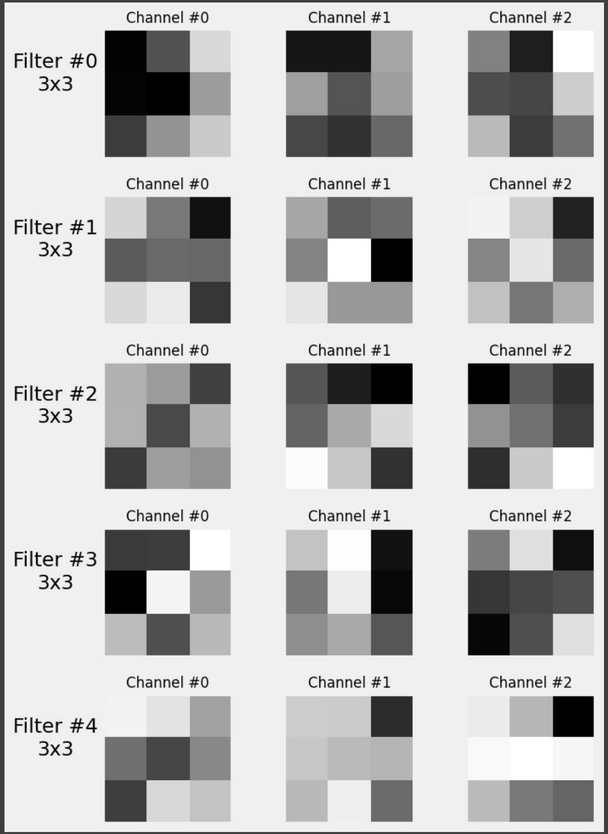
# Regularizing Effect

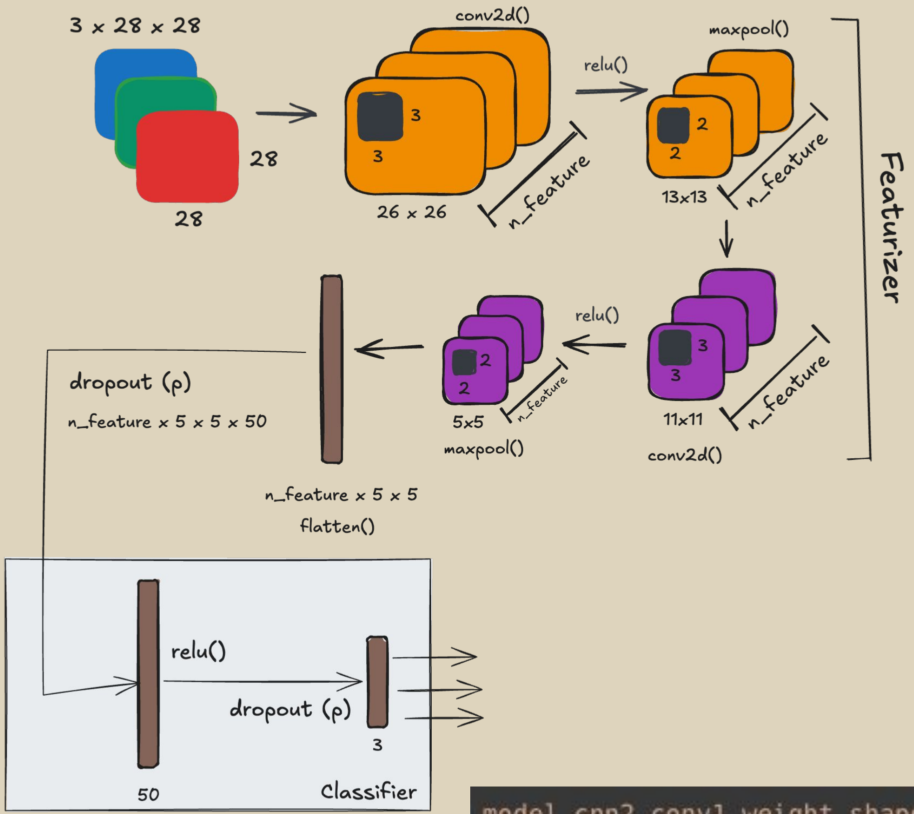


# Regularizing Effect

```
# no dropout
print(
    Architecture.loader_apply(train_loader, arch_cnn2_nodrop.correct).sum(axis=0),
    Architecture.loader_apply(val_loader, arch_cnn2_nodrop.correct).sum(axis=0)
)

tensor([2520, 2520]) tensor([292, 372])
1.0 0.7849462365591398

# with dropout
print(
    Architecture.loader_apply(train_loader, arch_cnn2.correct).sum(axis=0),
    Architecture.loader_apply(val_loader, arch_cnn2.correct).sum(axis=0)
)
tensor([2504, 2520]) tensor([326, 372])
0.9936507936507937 0.8763440860215054
```

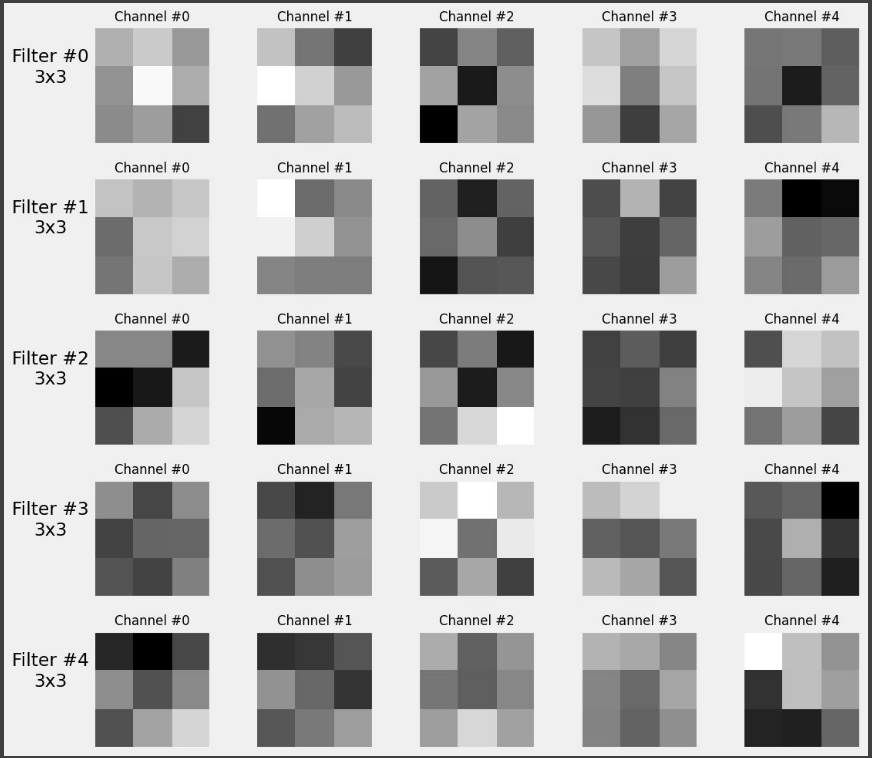
# Visual. Filters

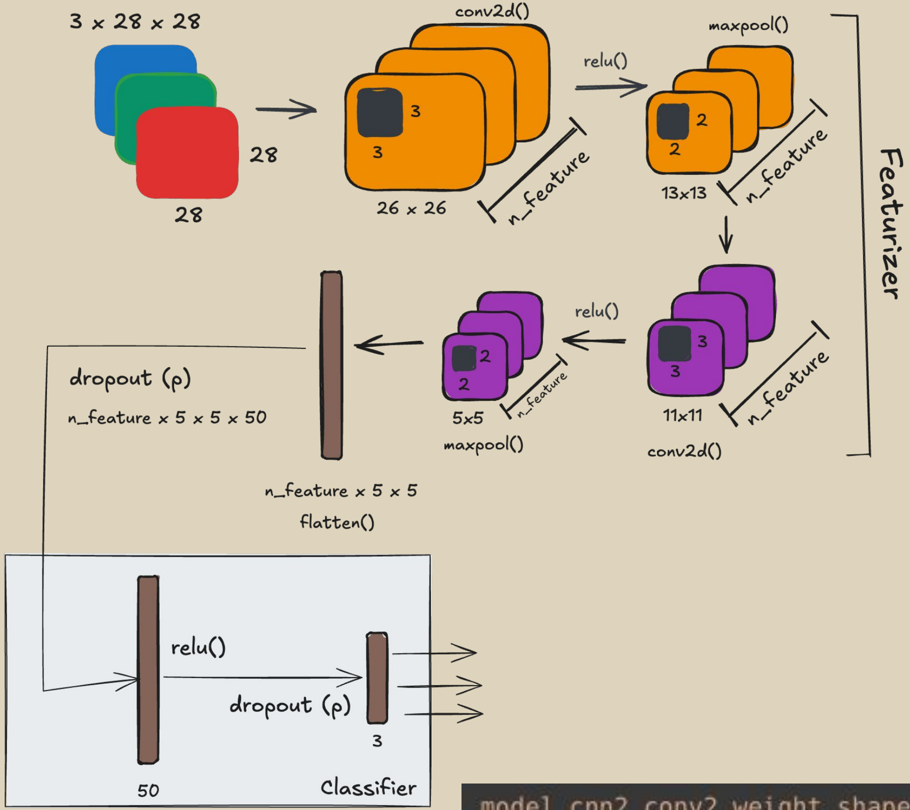



```
model_cnn2.conv1.weight.shape

torch.Size([5, 3, 3, 3])
```

# Visual. Filters




```
model_cnn2.conv2.weight.shape

torch.Size([5, 5, 3, 3])
```

Cont.