PPGEEC2318

Machine Learning

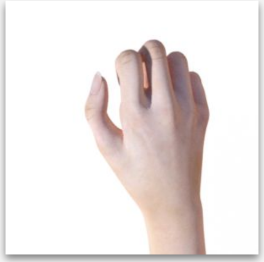
Rock, Paper, Scissors

Ivanovitch Silva
ivanovitch.silva@ufrn.br

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*
4. *learn how to* ***find a learning rate*** *to train the model*
5. *understand how the* ***Adam optimizer*** *uses adaptive learning rates*
6. ***capture gradients*** *and* ***parameters*** *to visualize their evolution during training*
7. *understand how* ***momentum*** *and* ***Nesterov*** *momentum work*
8. *use* ***schedulers*** *to implement l****earning rate changes*** *during training*

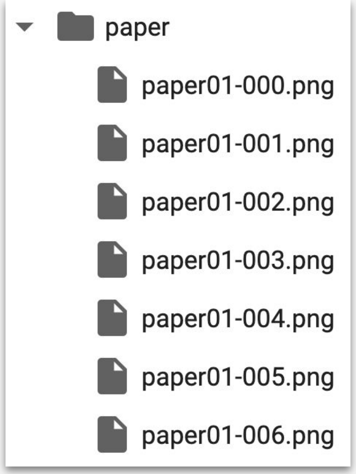
The dataset contains 2,892 images (2,520 train, 372 test) of diverse hands in the typical rock, paper, and scissors poses against a white background. This is a synthetic dataset as well since the images were generated using CGI techniques. Each image is 300x300 pixels in size and has four channels (RGBA).

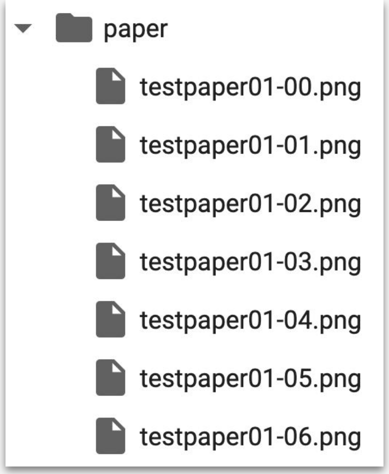


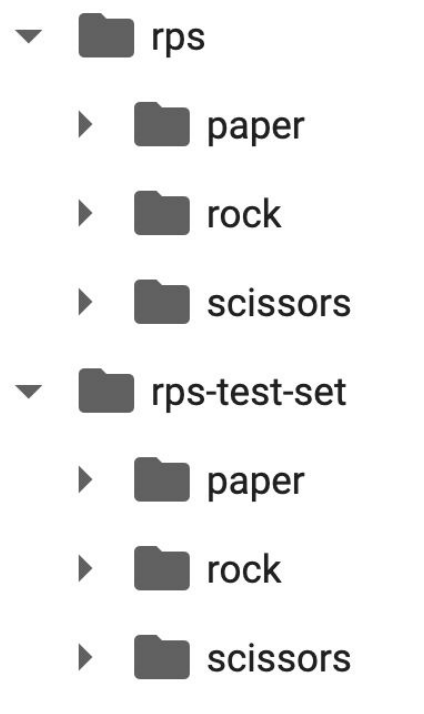
Rock

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

```
from torchvision.datasets import ImageFolder

# images are resized to 28x28 pixels
# automatically transformed to the RGB color model by the PIL loader,
# thus losing the alpha channel
temp_transform = Compose([Resize(28), ToTensor()])
temp_dataset = ImageFolder(root='rps', transform=temp_transform)


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

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

# you have 2520 images
temp_dataset[2519][0].shape
torch.Size([3, 28, 28])
```

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

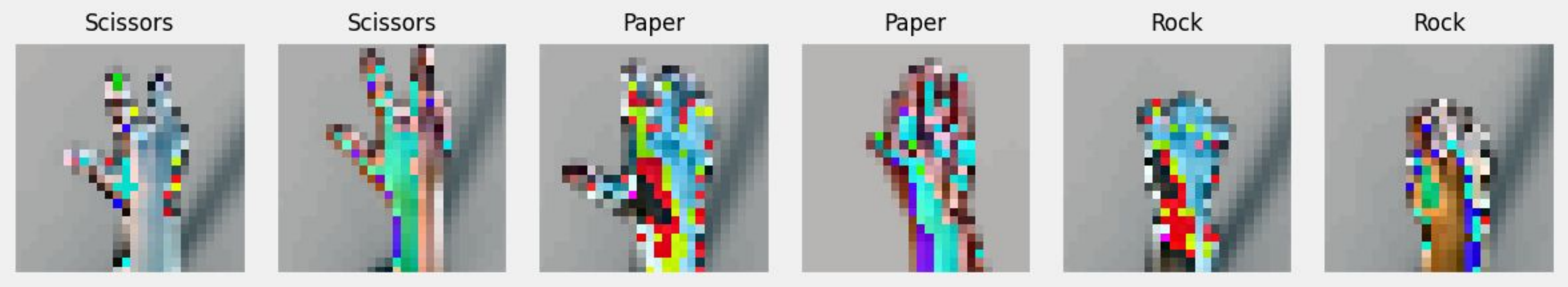
Standardization

```
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]))
```

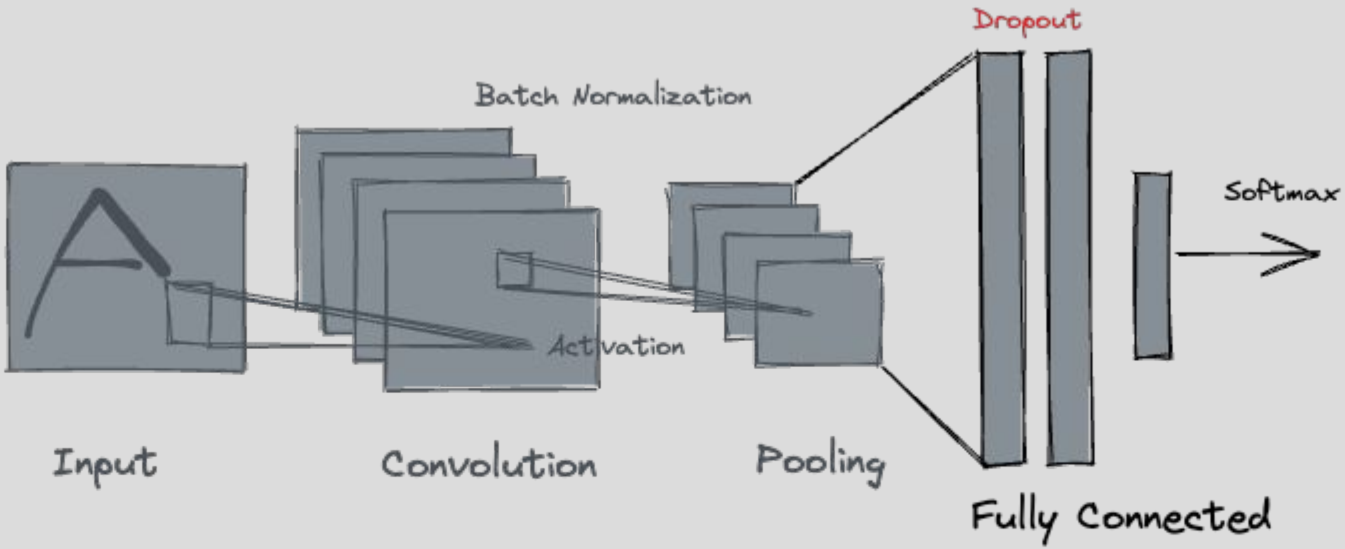
Data Preparation

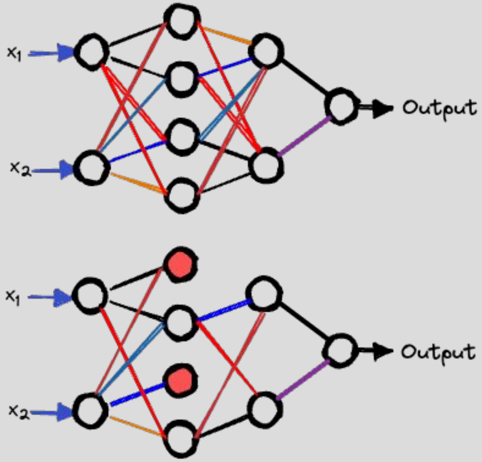
The real dataset

```
composer = Compose([Resize(28),
                    ToTensor(),
                    normalizer])

train_data = ImageFolder(root='rps', transform=composer)
val_data = ImageFolder(root='rps-test-set', transform=composer)

# Builds a loader of each set
train_loader = DataLoader(train_data, batch_size=16, shuffle=True)
val_loader = DataLoader(val_data, batch_size=16)
```



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?)

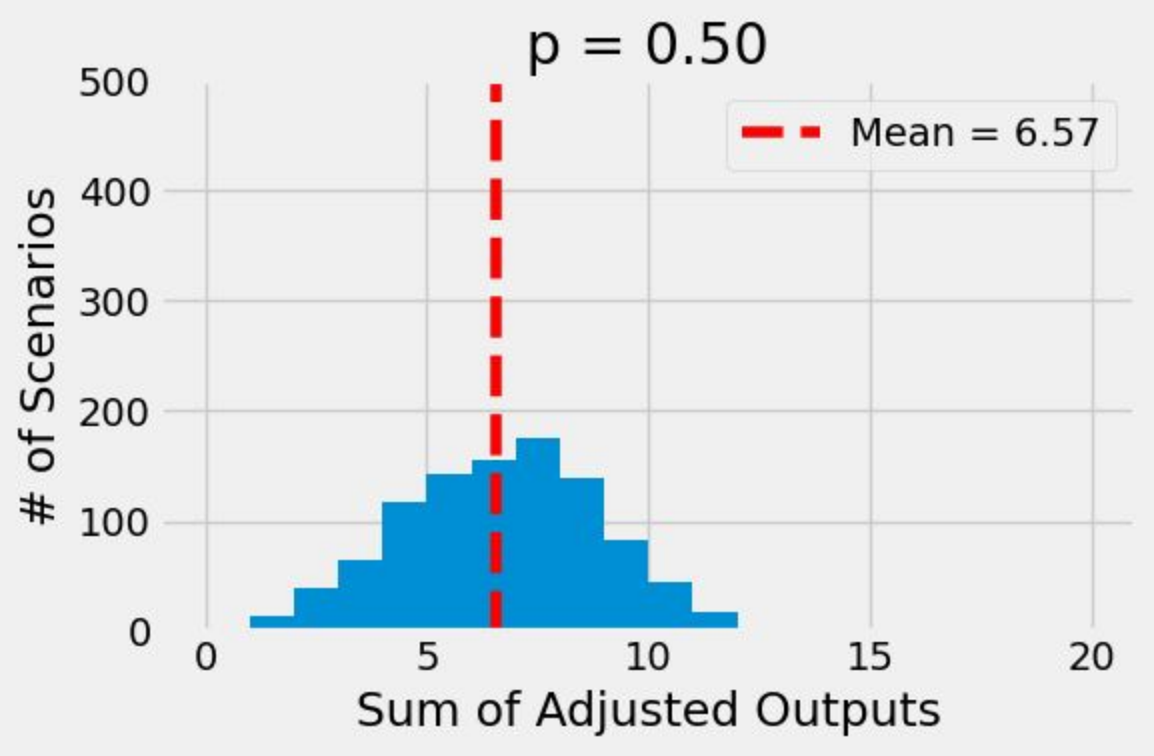
```
dropping_model = nn.Sequential(nn.Dropout(p=0.5))
spaced_points = torch.linspace(.1, 1.1, 11)
spaced_points

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)
```

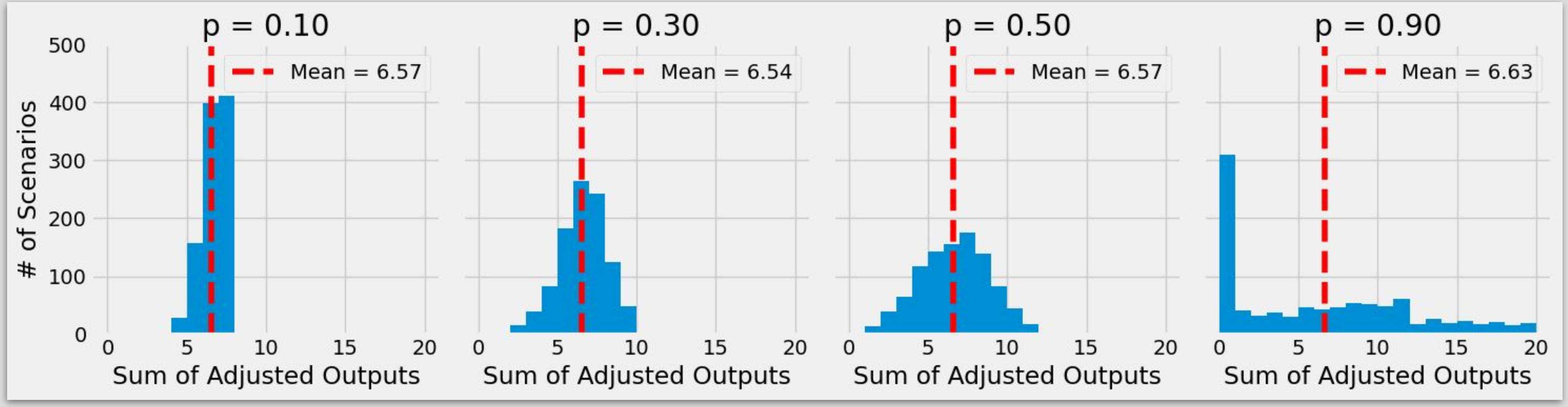
Dropout

Distribution of 1000 outputs



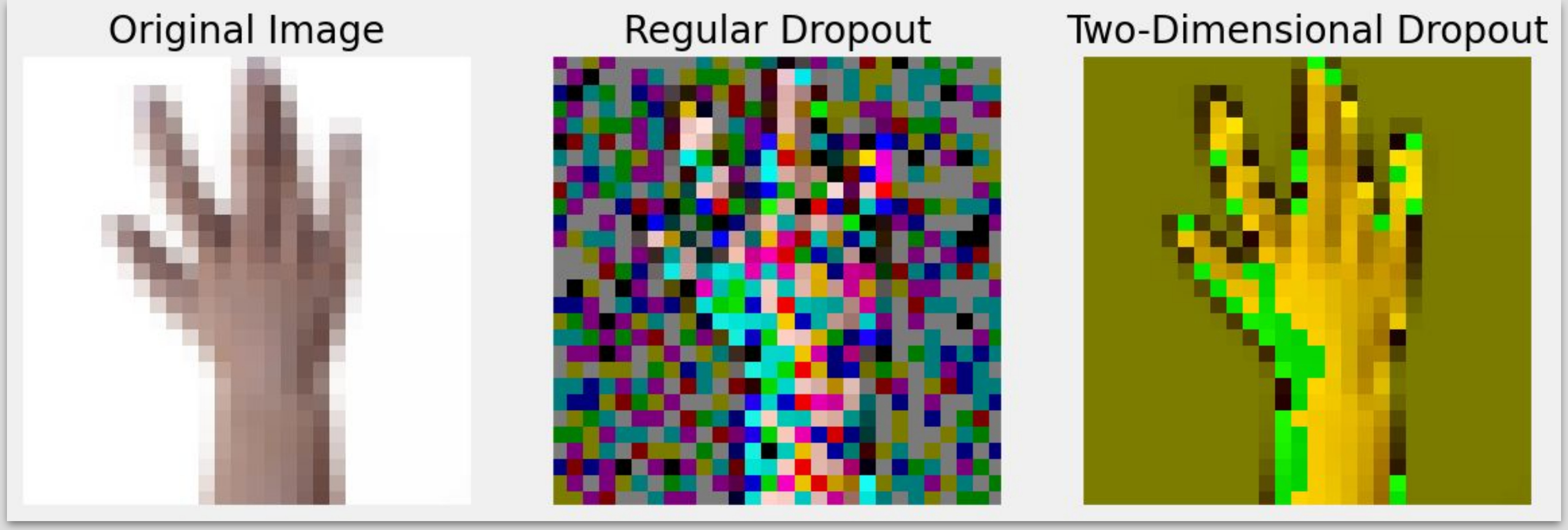
Dropout

Distribution of 1000 outputs



- For more typical dropout probabilities (like 30% or 50%), the distribution may take some more extreme values
- 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

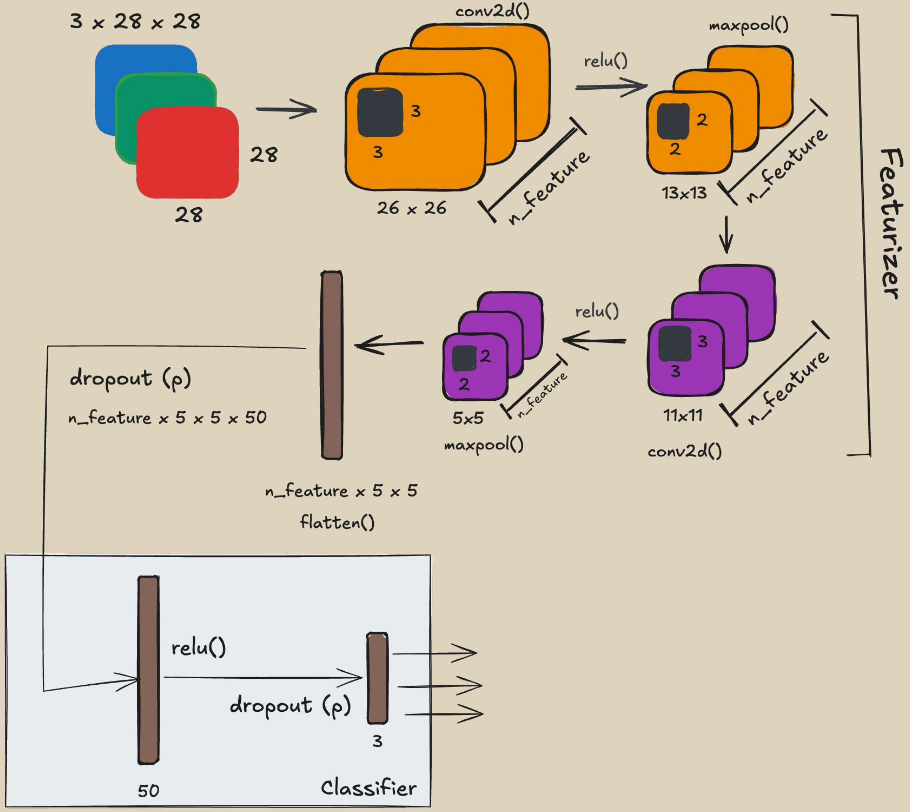
Two dimensional dropout



- 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.

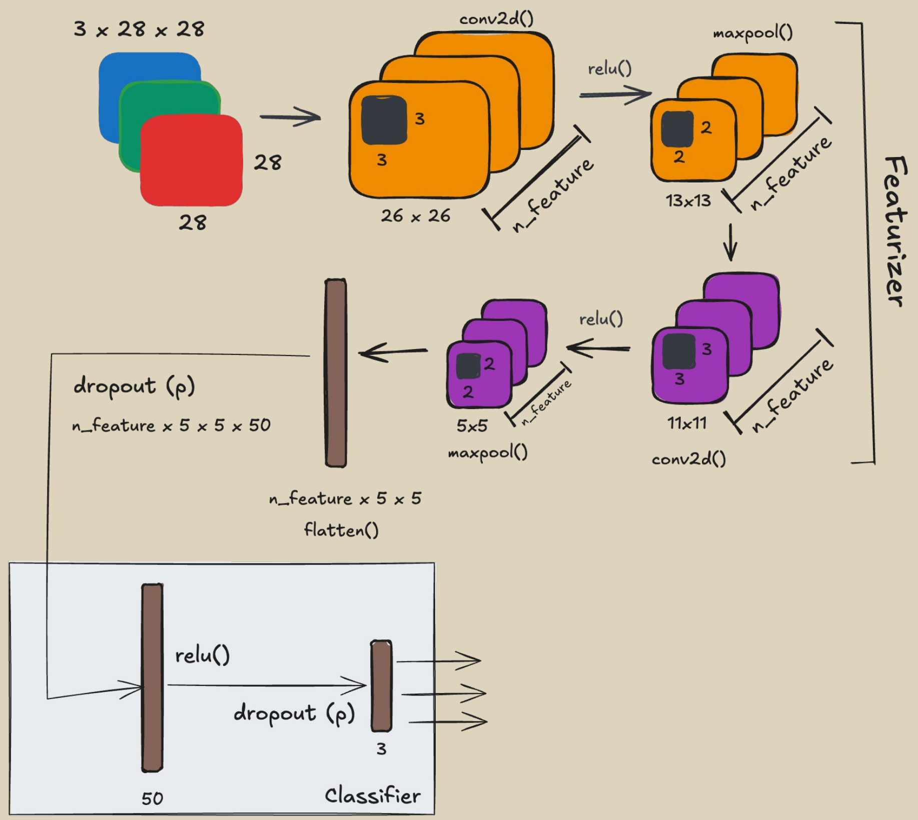
Fancier Model

```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)
```



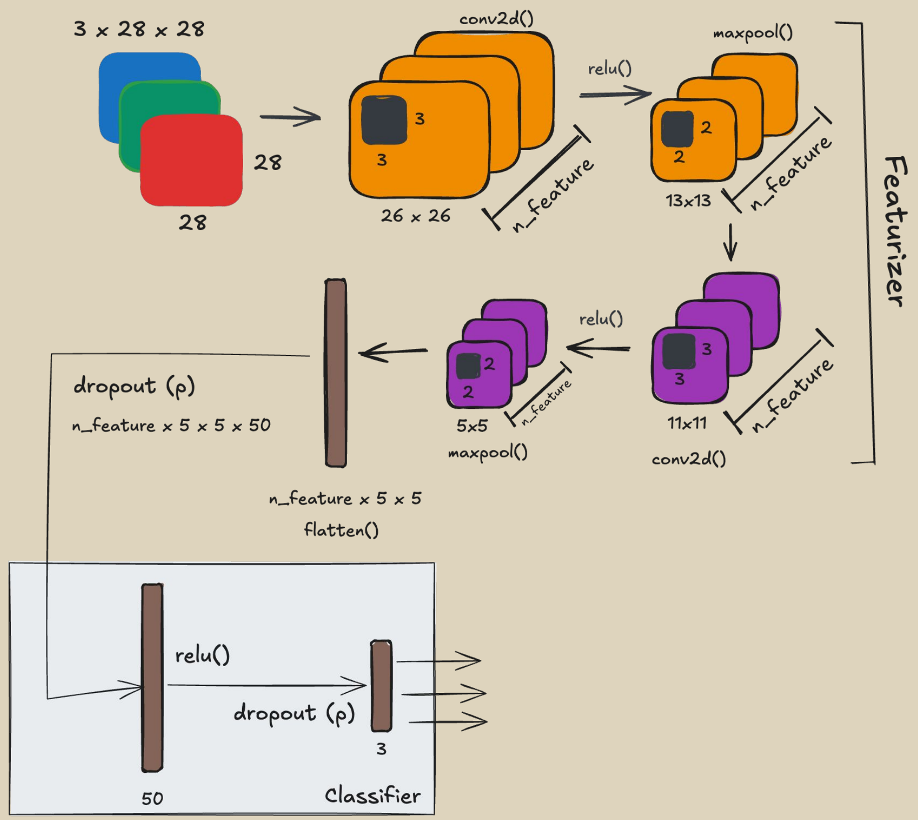
Fancier Model

```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
```



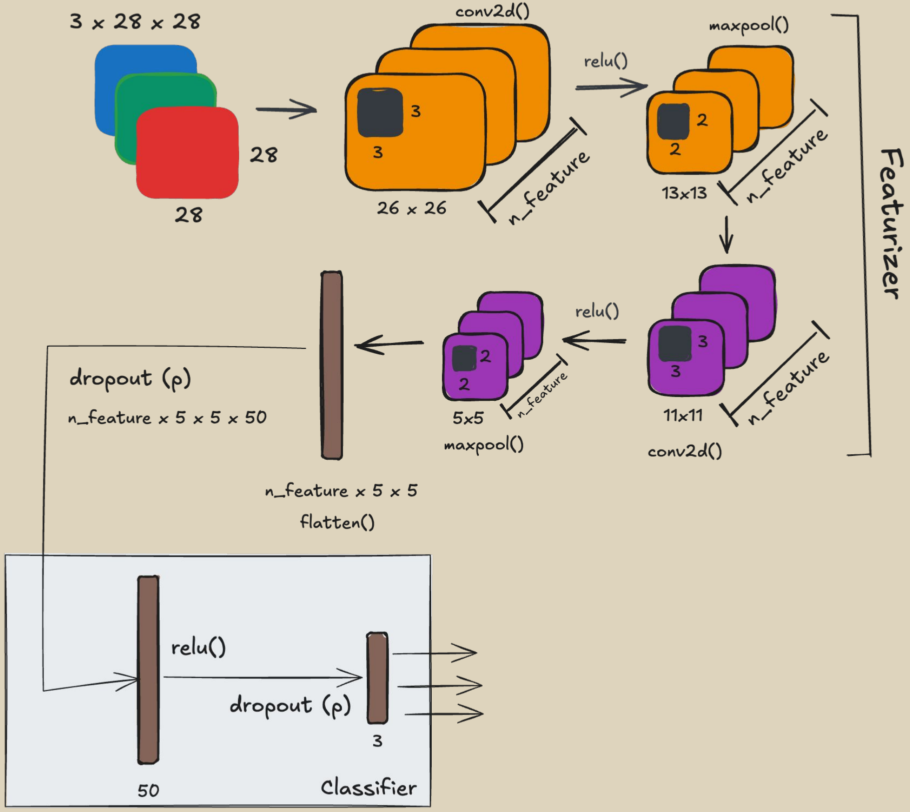
Fancier Model

```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
```



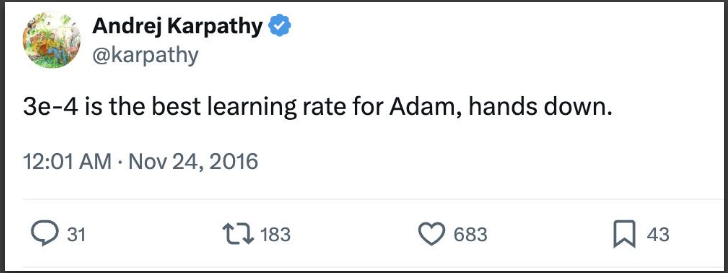
Case Study

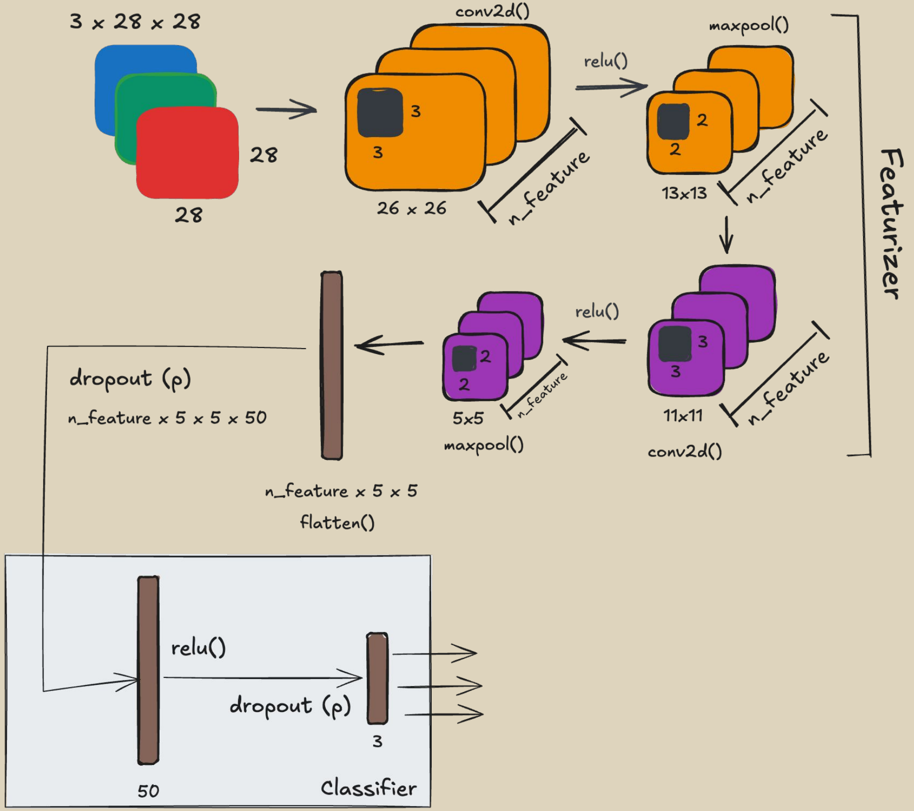
Model Configuration
Model Training
Accuracy
Regularizing Effect
Visualizing Filters



Model Config.

```
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)
```



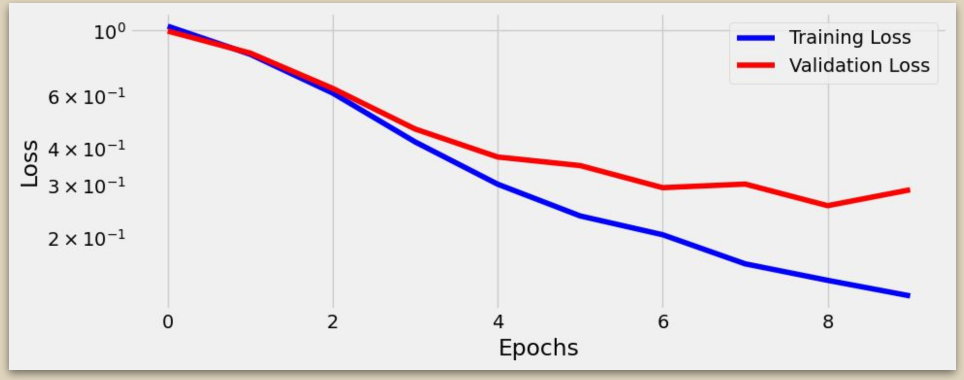


Model Train

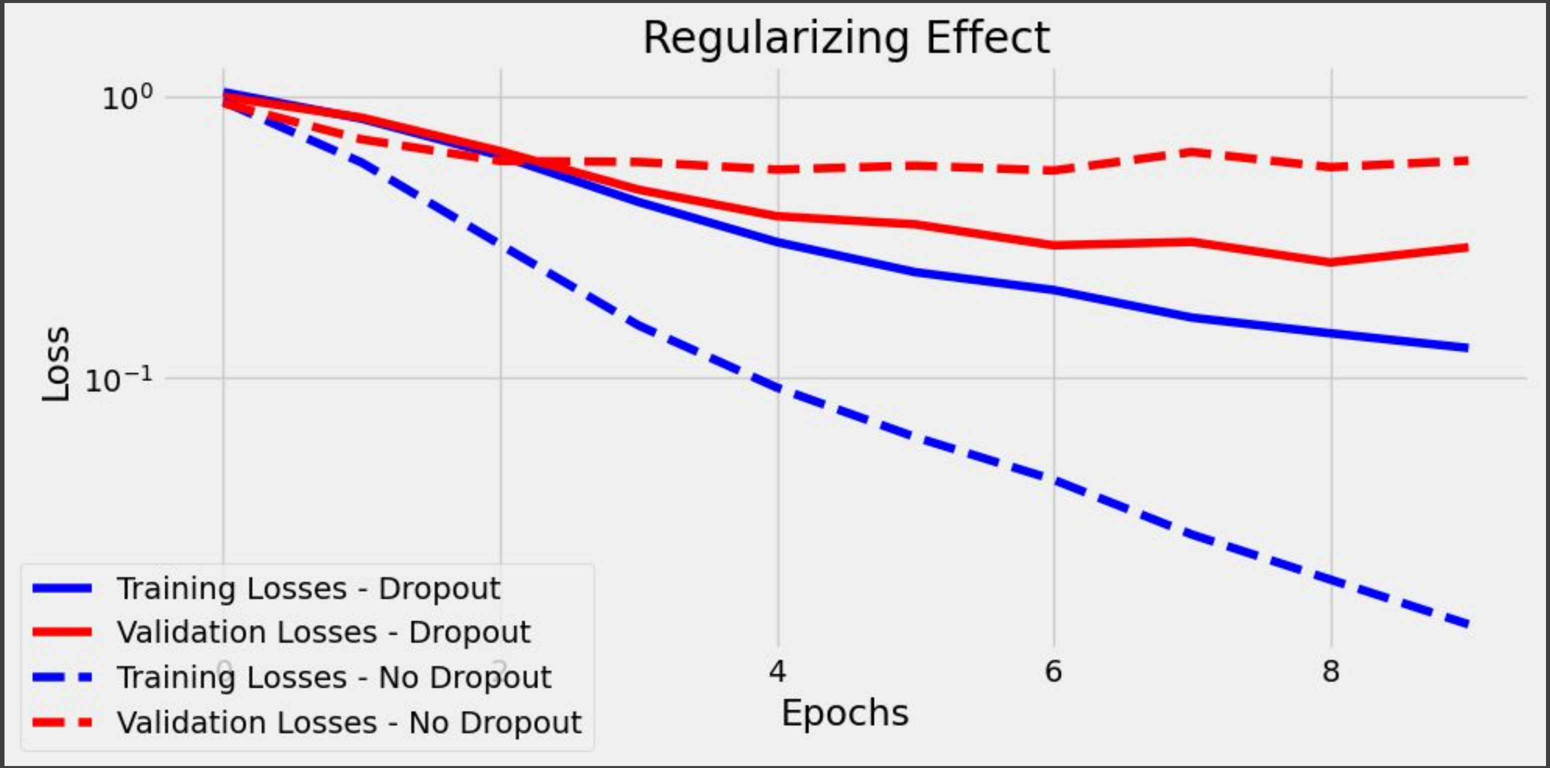
```
arch_cnn2 = Architecture(model_cnn2,
                         multi_loss_fn,
                         optimizer_cnn2)
arch_cnn2.set_loaders(train_loader, val_loader)
arch_cnn2.train(10)

arch_cnn2.count_parameters()
6823
```

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

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

87%



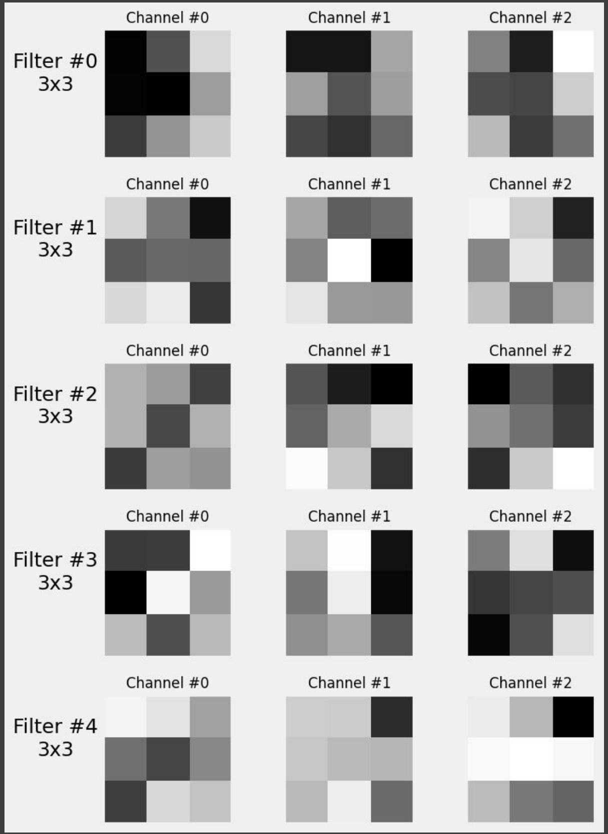
Regularizing Effect

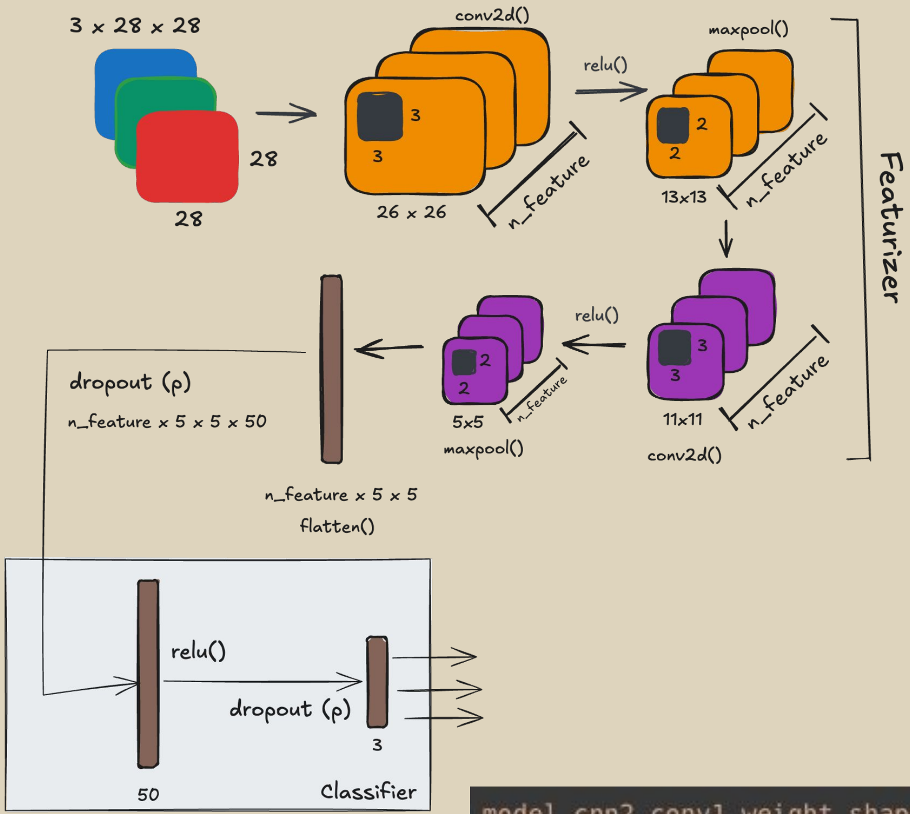


Regularizing Effect

```
# 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
```

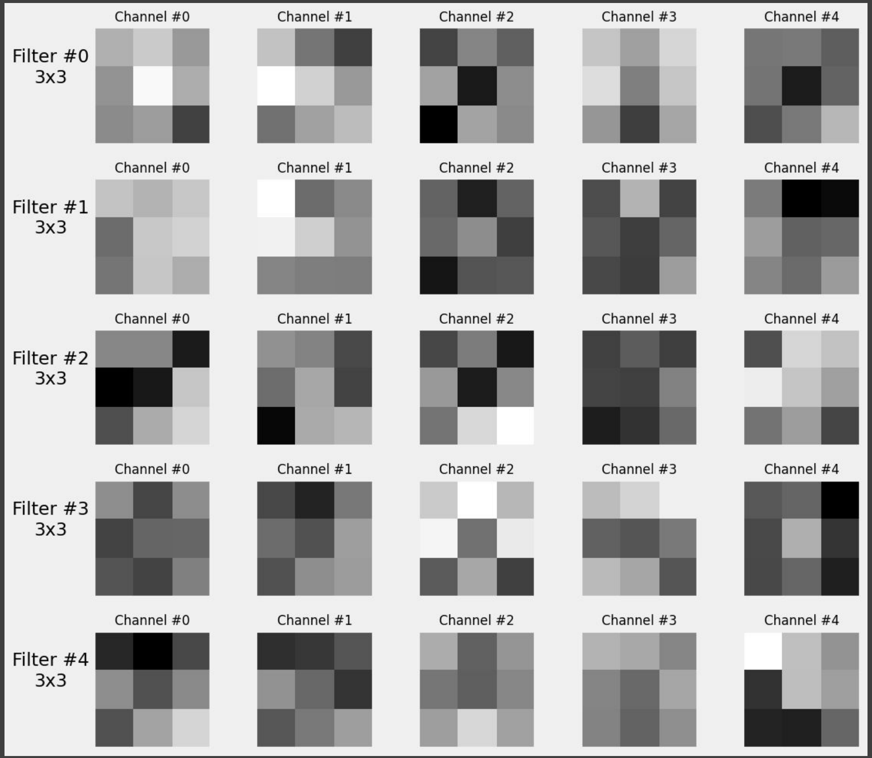
Visual. Filters

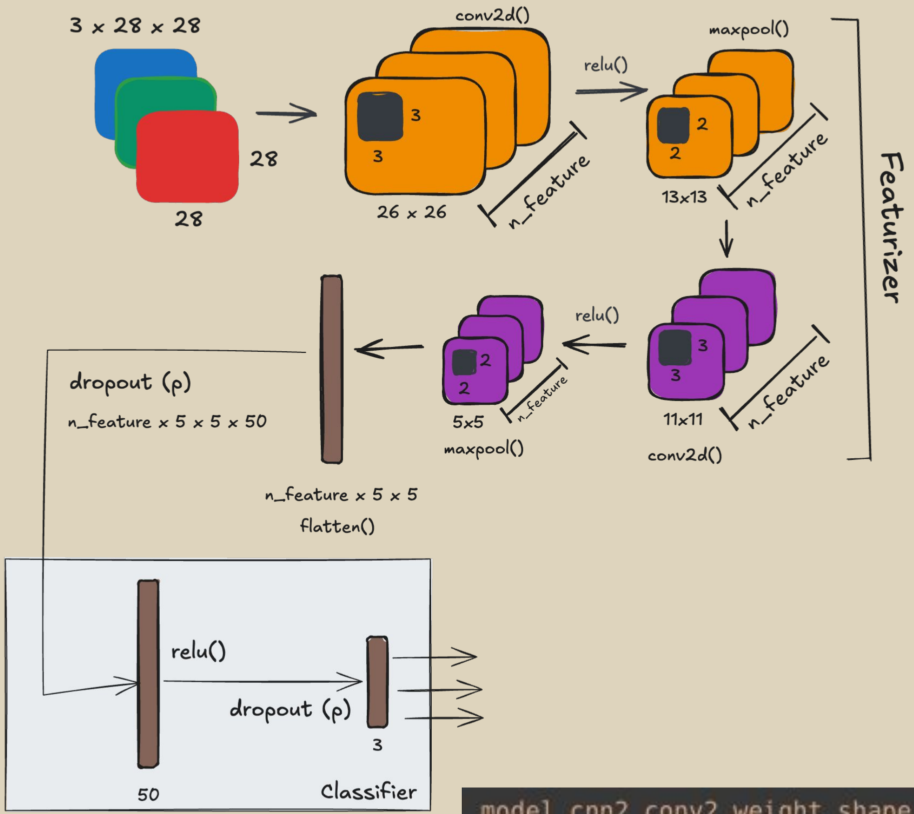




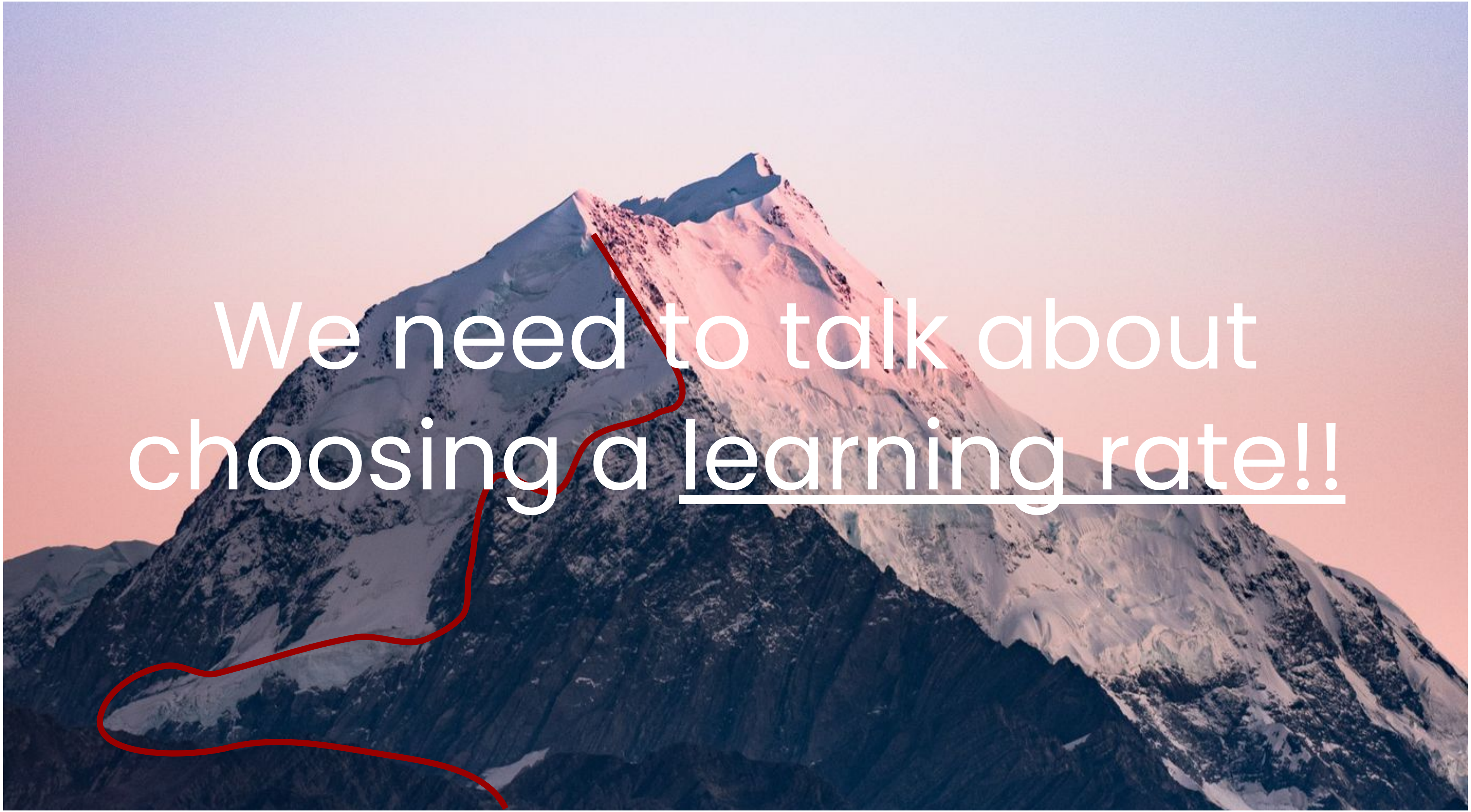

```
model_cnn2.conv1.weight.shape

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

Visual. Filters






```
model_cnn2.conv2.weight.shape

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



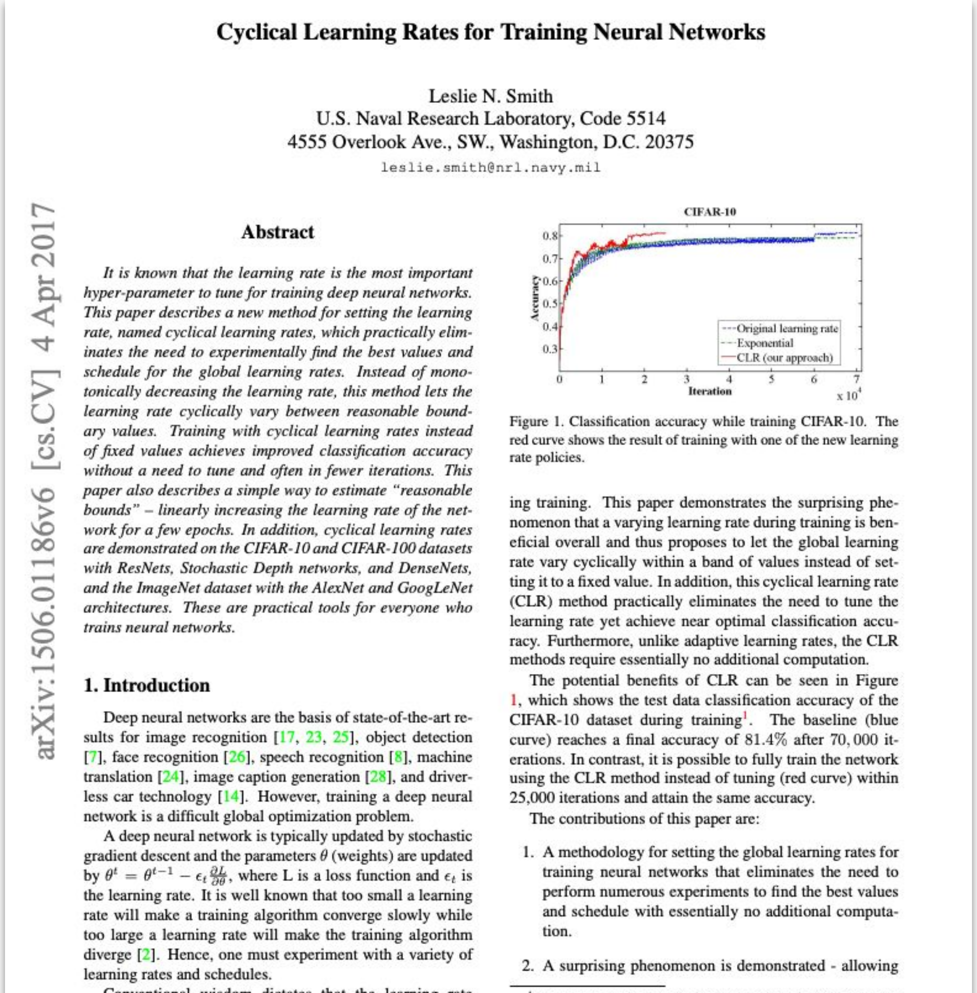
We need to talk about choosing a learning rate!!

All you need is a grid-search?

Trying multiple learning rates over a few epochs each and the evolution of the losses

```
# it is common to reduce the LR by a factor
# of 3 or a factor of 10

# using factor of 3
[0.1, 0.03, 0.01, 3e-3, 1e-3, 3e-4, 1e-4]

# using a factor of 10
[0.1, 0.01, 1e-3, 1e-4, 1e-5]
```

1) If the learning rate is too low
    a) The model doesn't learn much and the loss remains high.
2) If the learning rate is too high
    a) The model doesn't converge to a solution and the loss gets higher.



Cyclical Learning Rates for Training Neural Networks

Leslie N. Smith
U.S. Naval Research Laboratory, Code 5514
4555 Overlook Ave., SW., Washington, D.C. 20375
leslie.smith@nrl.navy.mil

arXiv:1506.01186v6 [cs.CV] 4 Apr 2017

Abstract

It is known that the learning rate is the most important hyper-parameter to tune for training deep neural networks. This paper describes a new method for setting the learning rate, named cyclical learning rates, which practically eliminates the need to experimentally find the best values and schedule for the global learning rates. Instead of monotonically decreasing the learning rate, this method lets the learning rate cyclically vary between reasonable boundary values. Training with cyclical learning rates instead of fixed values achieves improved classification accuracy without a need to tune and often in fewer iterations. This paper also describes a simple way to estimate "reasonable bounds" – linearly increasing the learning rate of the network for a few epochs. In addition, cyclical learning rates are demonstrated on the CIFAR-10 and CIFAR-100 datasets with ResNets, Stochastic Depth networks, and DenseNets, and the ImageNet dataset with the AlexNet and GoogLeNet architectures. These are practical tools for everyone who trains neural networks.

1. Introduction

Deep neural networks are the basis of state-of-the-art results for image recognition [17, 23, 25], object detection [7], face recognition [26], speech recognition [8], machine translation [24], image caption generation [28], and driverless car technology [14]. However, training a deep neural network is a difficult global optimization problem.

A deep neural network is typically updated by stochastic gradient descent and the parameters θ (weights) are updated by $\theta^t = \theta^{t-1} - \epsilon_t \frac{\partial L}{\partial \theta}$, where L is a loss function and ϵ_t is the learning rate. It is well known that too small a learning rate will make a training algorithm converge slowly while too large a learning rate will make the training algorithm diverge [2]. Hence, one must experiment with a variety of learning rates and schedules.

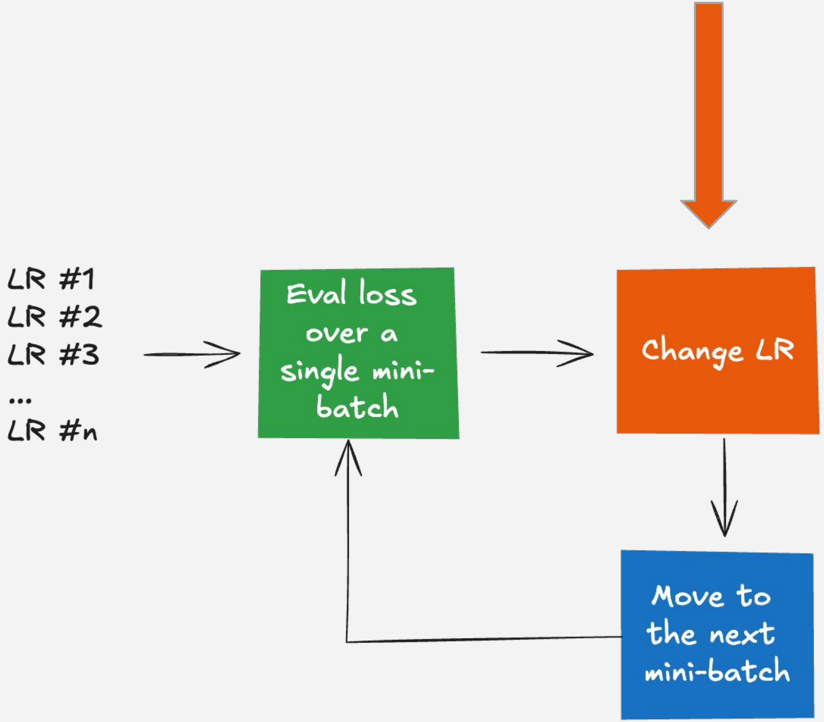
Figure 1. Classification accuracy while training CIFAR-10. The red curve shows the result of training with one of the new learning rate policies.

ing training. This paper demonstrates the surprising phenomenon that a varying learning rate during training is beneficial overall and thus proposes to let the global learning rate vary cyclically within a band of values instead of setting it to a fixed value. In addition, this cyclical learning rate (CLR) method practically eliminates the need to tune the learning rate yet achieve near optimal classification accuracy. Furthermore, unlike adaptive learning rates, the CLR methods require essentially no additional computation.

The potential benefits of CLR can be seen in Figure 1, which shows the test data classification accuracy of the CIFAR-10 dataset during training[1]. The baseline (blue curve) reaches a final accuracy of 81.4% after 70,000 iterations. In contrast, it is possible to fully train the network using the CLR method instead of tuning (red curve) within 25,000 iterations and attain the same accuracy.

The contributions of this paper are:

1. A methodology for setting the global learning rates for training neural networks that eliminates the need to perform numerous experiments to find the best values and schedule with essentially no additional computation.
2. A surprising phenomenon is demonstrated - allowing



LR Range Test

Higher-Order Learning Rate Function Builder

```
def make_lr_fn(start_lr, end_lr, num_iter, step_mode='exp'):
    if step_mode == 'linear':
        factor = (end_lr / start_lr - 1) / num_iter
        def lr_fn(iteration):
            return 1 + iteration * factor
    else:
        factor = (np.log(end_lr) - np.log(start_lr)) / num_iter
        def lr_fn(iteration):
            return np.exp(factor)**iteration
    return lr_fn
```

```
start_lr = 0.01
end_lr = 0.1
num_iter = 10
lr_fn = make_lr_fn(start_lr, end_lr,
                   num_iter, step_mode='exp')

lr_fn(np.arange(num_iter + 1))
array([ 1.        ,  1.25892541,  1.58489319,  1.99526231,
 2.51188643,3.16227766,  3.98107171,  5.01187234,
 6.30957344,  7.94328235,10.])

start_lr * lr_fn(np.arange(num_iter + 1))
array([0.01      , 0.01258925, 0.01584893, 0.01995262,
0.02511886, 0.03162278, 0.03981072, 0.05011872, 0.06309573,
0.07943282, 0.1])
```

```
start_lr = 0.01
end_lr = 0.1
num_iter = 10
lr_fn = make_lr_fn(start_lr, end_lr,
                   num_iter, step_mode='linear')

lr_fn(np.arange(num_iter + 1))
array([ 1. ,  1.9,  2.8,  3.7,  4.6,  5.5,  6.4,  7.3,  8.2,  9.1, 10.
])

start_lr * lr_fn(np.arange(num_iter + 1))
array([0.01 , 0.019, 0.028, 0.037, 0.046, 0.055, 0.064, 0.073, 0.082,
       0.091, 0.1  ])
```

How do I change the learning rate of an optimizer?

```
start_lr = 0.01
end_lr = 0.1
num_iter = 10
lr_fn = make_lr_fn(start_lr, end_lr,
                   num_iter, step_mode='exp')

lr_fn(np.arange(num_iter + 1))
array([ 1.        ,  1.25892541,  1.58489319,  1.99526231,
 2.51188643,3.16227766,  3.98107171,  5.01187234,
 6.30957344,  7.94328235,10.])

start_lr * lr_fn(np.arange(num_iter + 1))
array([0.01      , 0.01258925, 0.01584893, 0.01995262,
0.02511886, 0.03162278, 0.03981072, 0.05011872, 0.06309573,
0.07943282, 0.1])
```

```
dummy_model = CNN2(n_feature=5, p=0.3)
dummy_optimizer = optim.Adam(dummy_model.parameters(), lr=start_lr)
```

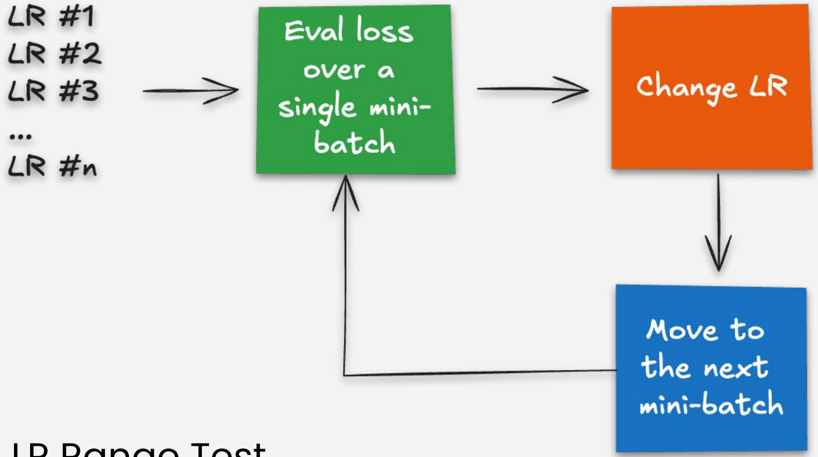
All you need is a scheduler!!!

The LambdaLR scheduler takes an **optimizer** and a **custom function** as arguments and modifies the learning rate of that optimizer accordingly

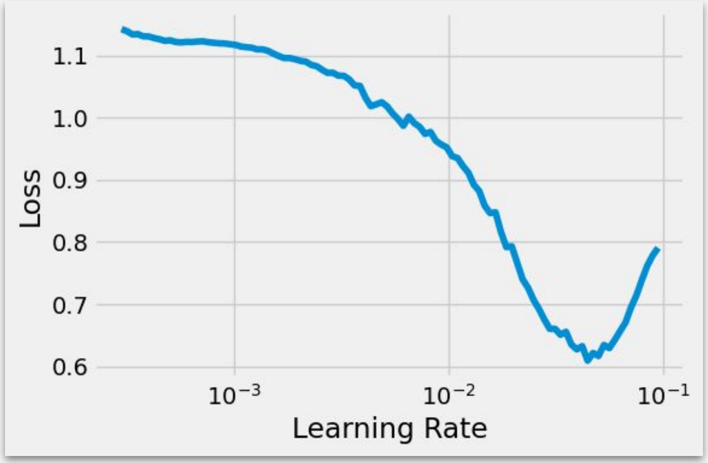
```
from torch.optim.lr_scheduler import StepLR, ReduceLROnPlateau, MultiStepLR, CyclicLR, LambdaLR

dummy_model = CNN2(n_feature=5, p=0.3)
dummy_optimizer = optim.Adam(dummy_model.parameters(), lr=start_lr)
dummy_scheduler = LambdaLR(dummy_optimizer, lr_lambda=lr_fn)
```

```
dummy_optimizer.step()
dummy_scheduler.step()

dummy_scheduler.get_last_lr()[0]
0.012589254117941673
```

```
start_lr * lr_fn(np.arange(num_iter + 1))
array([0.01      , 0.01258925, 0.01584893, 0.01995262, 0.02511886,
       0.03162278, 0.03981072, 0.05011872, 0.06309573, 0.07943282,
       0.1       ])
```



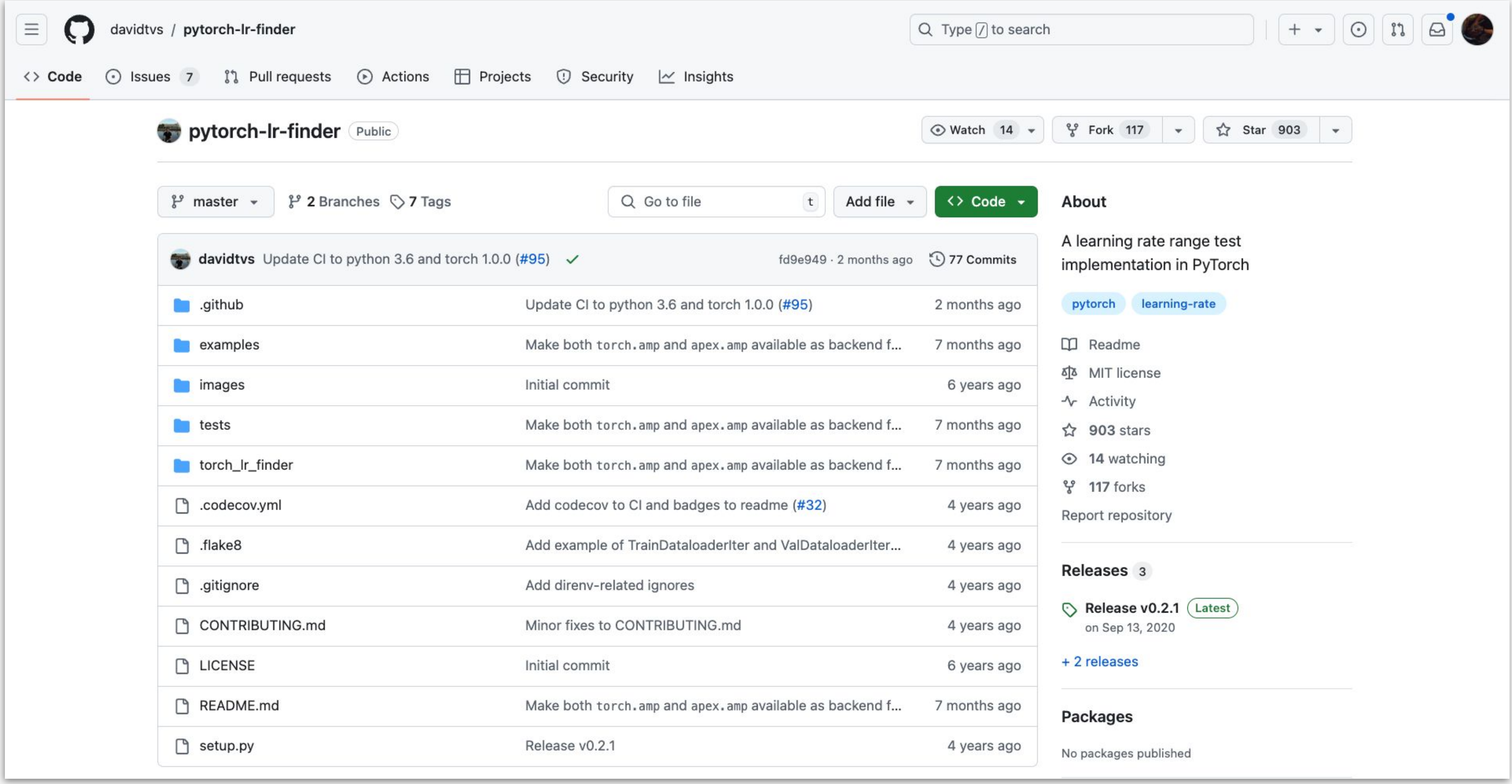
LR Range Test



(0.6091022460474141, 0.04434013138652989)

```
# model
new_model = CNN2(n_feature=5, p=0.3)

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

# optimizer
new_optimizer = optim.Adam(new_model.parameters(), lr=3e-4)

# architecture
arch_new = Architecture(new_model, multi_loss_fn, new_optimizer)

# lr range test
tracking, fig = arch_new.lr_range_test(train_loader, end_lr=1e-1, num_iter=100)
```

"U-Shape curve"

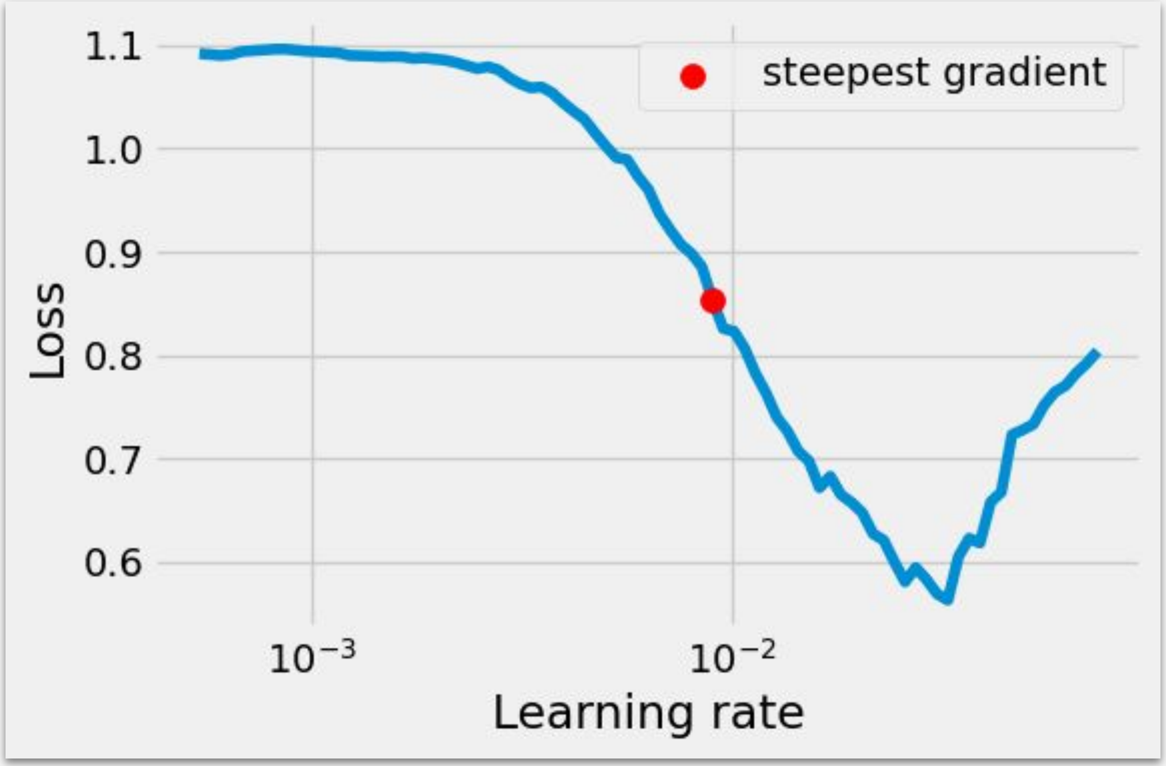

davidtvs / pytorch-lr-finder
Type / to search
Code
Issues 7
Pull requests
Actions
Projects
Security
Insights
pytorch-lr-finder
Public
Watch 14
Fork 117
Star 903
master
2 Branches
7 Tags
Go to file
t
Add file
Code
davidtvs Update CI to python 3.6 and torch 1.0.0 (#95)
fd9e949 · 2 months ago
77 Commits
.github
Update CI to python 3.6 and torch 1.0.0 (#95)
2 months ago
examples
Make both torch.amp and apex.amp available as backend f...
7 months ago
images
Initial commit
6 years ago
tests
Make both torch.amp and apex.amp available as backend f...
7 months ago
torch_lr_finder
Make both torch.amp and apex.amp available as backend f...
7 months ago
.codecov.yml
Add codecov to CI and badges to readme (#32)
4 years ago
.flake8
Add example of TrainDataloaderIter and ValDataloaderIter...
4 years ago
.gitignore
Add direnv-related ignores
4 years ago
CONTRIBUTING.md
Minor fixes to CONTRIBUTING.md
4 years ago
LICENSE
Initial commit
6 years ago
README.md
Make both torch.amp and apex.amp available as backend f...
7 months ago
setup.py
Release v0.2.1
4 years ago
About
A learning rate range test implementation in PyTorch
pytorch
learning-rate
Readme
MIT license
Activity
903 stars
14 watching
117 forks
Report repository
Releases 3
Release v0.2.1
Latest
on Sep 13, 2020
+ 2 releases
Packages
No packages published

```
!pip install --quiet torch-lr-finder
from torch_lr_finder import LRFinder
```

```
fig, ax = plt.subplots(1, 1, figsize=(6, 4))

# model, loss function, optimizer and device
new_model = CNN2(n_feature=5, p=0.3)
multi_loss_fn = nn.CrossEntropyLoss(reduction='mean')
new_optimizer = optim.Adam(new_model.parameters(), lr=3e-4)
device = 'cuda' if torch.cuda.is_available() else 'cpu'

# instantiate LR Finder object
lr_finder = LRFinder(new_model, new_optimizer, multi_loss_fn, device=device)

# LR range test
lr_finder.range_test(train_loader, end_lr=1e-1, num_iter=100)

# plot
lr_finder.plot(ax=ax, log_lr=True)
fig.tight_layout()

# reset model
lr_finder.reset()
```

The concept of the "steepest gradient" refers to the point where the loss decreases most rapidly.

The idea is to find the highest learning rate that still results in a decrease in loss before the loss starts to increase due to an excessively high learning rate. This zone is typically at the steepest part of the loss curve.



LR suggestion: **steepest gradient**
Suggested LR: **9.02E-03**

"OK, if I manage to choose a good learning rate from the start, am I done with it?"