Classification Based on Missing Features in Deep Convolutional Neural Networks

Nemanja Milošević

UNSPMF

EuroPython 2019

Hello!

- Nemanja Milošević
- PhD Student, Teaching Assistant at University of Novi Sad
- Research topic: Neural network robustness
- nmilosev@dmi.uns.ac.rs
- nmilosev.svbtle.com
- @nmilosev

So what is this all about?

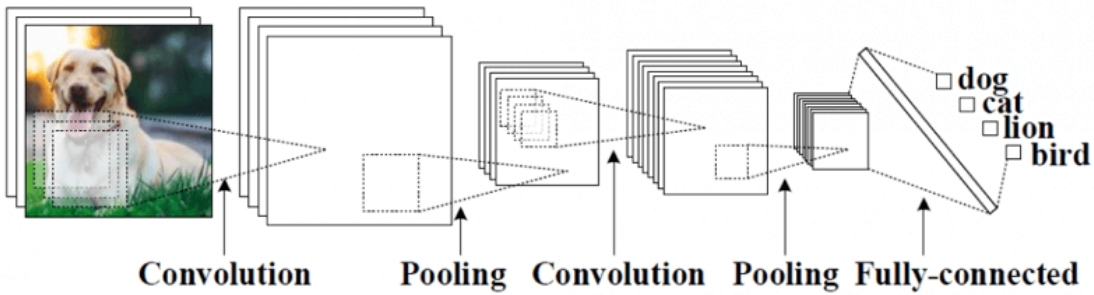
- This is a weird and quirky neural network model
- It tries to mimic deduction in classification
- It helps in certain scenarios
- Details and code snippets to follow!

A word of warning!

- This is all very hypothetical and untested
- Proof of concept paper sent to Neural Network World Journal (University of Prague)
- Deep neural network models are hard to interpret
- Question everything I say! :)

CNN's in a nutshell

- Convolutional layers are used to extact features from an image
- Modeled on animals visual cortex (neuron vicinity is important)
- More resistent to scaling, positioning issues
- FCL (Fully-Connected Layers) go after the convolutional layers
- Basically the same as MLP (Multi-Layer Perceptron – traditional neural network)
- *Features on an image are used to classify it!*



Motivation for missing feature classification

- What happens if we want to classify based on missing features?
- Feature set is finite – it is easy to get what features are missing
- It is possible to train a neural network to learn this way

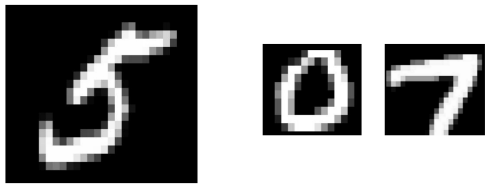


Figure 1: Digit "5" from the MNIST dataset and its missing features. Circle-like Feature 1 given here is present in digits 0, 6, 8, 9 while a sharp corner-line Feature 2 is present in digits 1, 2, 3, 4, and 7. Digit 5 does not have these features, therefore we can check the input image and see if these features are missing. If they are, we can safely assume that we are looking at digit 5.

Motivation for missing feature classification (cont'd)

- Why? $\rightarrow$ partial input recognition / occlusion

- Also, other adversary attacks
- Classification based on missing features
  - implemented with "inversion" of the output of the last convolutional layer
  - we only activate those neurons which are representing the missing features thus classification is done on the missing features

Classification on missing features

- First, we need the features somehow
- During training we let the sample through all conv and pool layers to get the features/positions vector
- Then, inversion of the last convolutional layer gives us the missing features
- Finally, we train the fully connected layers based on missing features

Step 1: Getting the features, Transfer learning

- We could handcraft the features but that is boring/difficult
- Instead we can train the network normally for a number of epochs and take the convolutional layers
- This is automatic, and much easier

Step 2: Activation functions

- So we got the feature vector, now what?
- Depending on the activation function in the last conv layer we modify the feature vector
- Simple example for the sigmoid function:
  - The output would be a number between 0 and 1
  - 1 means feature is present, 0 means it is not
  - To get the missing features apply:

$$f(x) = 1 - x$$

  - That's it!

Step 2: Activation functions (cont'd)

- That's cool but sigmoid is like 1976 and it is 2019
- ReLU is a much better choice, but beware
- ReLU is difficult to "negate" because it goes to infinity
- Solutions:
  - Use limited ReLU variant e.g. ReLU6
  - LeakyReLU or Swish could work (maybe)
  - Use *tanh*

Step 2: Activation functions – code

```
def forward(self, x):
    x = F.relu(F.max_pool2d(self.conv1(x), 2))
    x = F.relu(F.max_pool2d(self.conv2_drop(
                self.conv2(x)), 2))
    x = x.view(-1, 320)
    if self.net_type == 'negative':
        x = x.neg()
    if self.net_type == 'negative_relu':
        x = torch.ones_like(x).add(x.neg())
    x = F.relu(self.fc1(x))
    x = F.dropout(x, training=self.training)
    x = self.fc2(x)
    return F.log_softmax(x, dim=1)
```

Step 3: Layer freezing and resetting

- Our network is almost ready, but the pretrained part is not playing well with our "negation"
- If you train like this, the features will get corrupted due to the nature of backprop
- Solution: Freeze all the conv layers

- Optionally we can also reset the fully connected layers to "start fresh"
- While not necesarry it helps with the convergence

Step 3: Layer freezing and resetting – code

```
# reinitialize fully connected layers
model.fc1 = nn.Linear(320, HIDDEN).cuda()
model.fc2 = nn.Linear(HIDDEN, 10).cuda()
# freeze convolutional layers
model.conv1.weight.requires_grad = False
model.conv2.weight.requires_grad = False
# reinitialize the optimizer with new params
optimizer = \
    optim.SGD(filter(lambda p: p.requires_grad,
    model.parameters()), lr=LR, momentum=MOM)
```

Partial MNIST dataset

- To test our network we need a dataset
- For simplicity, we can use everyone's favorite: MNIST
- But wait, we need some partial inputs to validate our theory
- PMNIST has the same 60000 training samples but the validation set has been enhanced:
  - 10000 images with top 50% removed
  - 10000 images with left 50% removed
  - 10000 images with top-right 25% removed and bottom-left 25% removed
  - 10000 images with 33% removed in three randomly placed squares
- New 40000 images have been derived from the original 10000 validation set, not training set

Additional remarks

- It is easy to train on partial samples, but you should not do it
- On unmodified dataset we still have great accuracy
- PyTorch makes implementing "weird" models a treat!

Initial results on PMNIST

Dataset	Accuracy	Delta
Unmodified	98.55	0.31
Horizontal cut	67.00	4.45
Vertical cut	70.15	9.05
Diagonal cut	61.31	6.36
Triple cut	40.87	6.62

Table 1: The "Accuracy" column shows final, highest accuracy achieved on a given validation set while the "Delta" column shows accuracy gain over the standard unmodified network.

Future work

- Different datasets
- Different architectures
- Adversarial Networks
- Complete PyTorch reference implementation `git.io/fpArN`

Thank you so much for your attention!

Questions?

nmilosev@dmi.uns.ac.rs