Neural Networks: Part III

Training Practices

Daniel Yukimura

yukimura@impa.br

September 12, 2018

Training Practices

- Training Neural Network is not always easy
  Recall our goal of minimizing the risk function

$$R(\theta) = \mathbb{E}[\ell(y, f(x, \theta))]$$

- While we have only access to an empirical version of it

$$R_n(\theta) = \sum_{i=1}^{n} \ell\left(f\left(x_i, \theta\right), y_i\right)$$

Training Practices

- Training Neural Network is not always easy
  Recall our goal of minimizing the risk function

$$R(\theta) = \mathbb{E}[\ell(y, f(x, \theta))]$$

- While we have only access to an empirical version of it

$$R_n(\theta) = \sum_{i=1}^{n} \ell\left(f\left(x_i, \theta\right), y_i\right)$$

Training Practices

Questions:

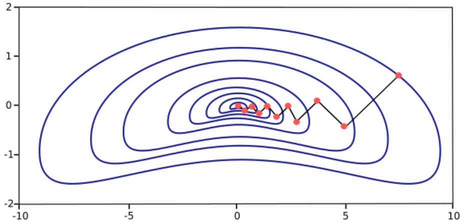
1. How do we minimize $R_n(\theta)$? since it's a high dimensional non-convex functional.
2. A minimum of $R_n(\theta)$ is good enough?
   It is also a good minimum for the theoretical risk $R(\theta)$?
   Remark: A global minimum for $R_n(\theta)$ would almost surely overfit the data.

Optimization

Gradient Descent

Recall the classical Gradient Descent (GD), where we adjust the parameters according to the gradient at the present point:

$$\theta_{t+1} = \theta_t - \eta \nabla L(\theta_t)$$



Optimization

In PyTorch, we could implement such operation as:

```
for e in range(nb_epochs):
    output = model(train_input)
    loss = criterion(output, train_target)
    model.zero_grad()
    loss.backward()
    with torch.no_grad():
        for p in model.parameters(): p -= eta * p.grad
```

However, it has a memory footprint proportional to the full set size.

Optimization

Stochastic Gradient Descent

Instead, we compute the gradient on random mini-batches:

$$\hat{L}_S(\theta) = \frac{1}{S} \sum_{i \in \mathcal{S}} \ell(\theta, Z_i) \qquad \theta_{t+1} = \theta_t - \eta \nabla \hat{L}_S(\theta_t)$$

```
optimizer = torch.optim.SGD(model.parameters(), lr = eta)
for e in range(nb_epochs):
    for b in range(0, train_input.size(0), mini_batch_size):
        output = model(train_input.narrow(0, b, mini_batch_size))
        loss = criterion(output, train_target.narrow(0, b, mini_bat
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

Optimization

Adaptive Learning Rates

- It's crucial to control the learning rates for the SGD case, since the iterations introduce a source of noise, that doesn't vanish with time. One common rule is to take it to decay linearly, or at least satisfying

$$\sum_{k=1}^{\infty} \eta_k = \infty, \quad \text{and} \quad \sum_{k=1}^{\infty} \eta_k^2 < \infty$$

- We can treat the learning rate, or the form it evolves as a **hyperparameter** (parameter whose value is set before the learning process begins).

Optimization

Momentum

Classic gradient descent can often become really slow, this is usually the case since there is a strong assumption behind the method that assumes a behaved gradient and local curvature. One way of addressing this problem is by using moment.

$$\begin{aligned} u_t &= \gamma u_{t-1} + \eta g_t \\ \theta_{t+1} &= \theta_t - u_t \end{aligned}$$

- In some sense it brings inertia, by adding the influence of the previous direction, and allowing acceleration.

Optimization

Adam (Adaptive Moments)

$$
\begin{aligned}
m_t &= \gamma\beta_1 m_{t-1} + (1-\beta_1)g_t \\
\hat{m}_t &= \frac{m_t}{1-\beta_1} \\
v_t &= \beta_2 v_{t-1} + (1-\beta_2)g_t \cdot g_t \\
\hat{v}_t &= \frac{v_t}{1-\beta_2} \\
\theta_{t+1} &= \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \varepsilon}\hat{m}_t
\end{aligned}
$$

Optimization

In PyTorch we can use the package `torch.optim` to apply several known optimization strategies

- `torch.optim.SGD` (includes momentum)
- `torch.optim.Adam`
- `torch.optim.Adagrad`
- `torch.optim.RMSProp`
- ...

Training Practices

Let's return to the question of whether a local minimum of $R_n(\theta)$ is good enough.

- Designing better optimization algorithms surely help us reaching better minimums.
- Another way of approaching generalization is to use **regularization**. Which usually comprises in controlling the model capacity by adding a criterion on the loss/risk to be minimized

$$\hat{\theta}_n = \operatorname*{argmin}_{\theta \in \Theta} R_n(\theta) + \lambda \Omega(\theta)$$

Regularization

L^2 and L^1 Regularization:

- The most classical regularization is the L^2 kind, which uses the euclidean norm as penalty

$$\hat{\theta}_n = \operatorname*{argmin}_{\theta \in \Theta} R_n(\theta) + \lambda \|\theta\|_2.$$

This kind of regularization, known as **Tikhonov** regularization, deals with the typical problem of a ill-posed Hessian. A ill-posed Hessian loss makes the problem unstable, sensitive to noise.

Regularization

L^2 and L^1 Regularization:

- The next most common kind is the L^1 regularization

$$\hat{\theta}_n = \operatorname*{argmin}_{\theta \in \Theta} R_n(\theta) + \lambda \|\theta\|_1.$$

In this case we control the capacity of the model by promoting sparsity. The **Lasso** method is a modification of such regularization, and is very popular when working with high dimensional statistics.

Regularization

Dropout:

- Is an example of a “deep” regularization, since it’s made specifically for neural networks.
- It can be seen as a method for training an ensemble of models, simpler models.
  Remark: One can train several alternative models in a same task, and use a decision criterion over their predictions, to obtain a better one, this is called **ensemble learning**.
- Dropout trains the ensemble of all sub-networks that can be formed by turning off non-output units.

Regularization

Dropout:

- Is an example of a “deep” regularization, since it’s made specifically for neural networks.
- It can be seen as a method for training an ensemble of models, simpler models.
  Remark: One can train several alternative models in a same task, and use a decision criterion over their predictions, to obtain a better one, this is called **ensemble learning**.
- Dropout trains the ensemble of all sub-networks that can be formed by turning off non-output units.

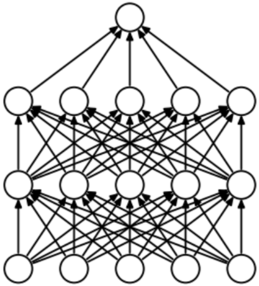
Regularization

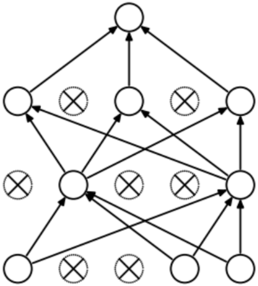
Dropout:

- Is an example of a “deep” regularization, since it’s made specifically for neural networks.
- It can be seen as a method for training an ensemble of models, simpler models.
  Remark: One can train several alternative models in a same task, and use a decision criterion over their predictions, to obtain a better one, this is called **ensemble learning**.
- Dropout trains the ensemble of all sub-networks that can be formed by turning off non-output units.

Regularization

Dropout:


(a) Standard Neural Net


(b) After applying dropout.

Regularization

Dropout:

- The sub-network is taken randomly at each step, making a statistically dependent set of ensembles. Therefore, the loss function must be weighted appropriately, which in this case uses the probability that such sub-model is chosen.

$$\sum_{\mathcal{S} \subseteq \mathcal{N}} p(\mathcal{S}) p_{\mathcal{S}}(y \mid x)$$

- Nevertheless, such sum has a exponential number of terms, making it intractable in practice. Instead, one would use an heuristic idea of simply scaling the weights by the probability of choosing such neuron at the end of training.

Regularization

Dropout:

- The sub-network is taken randomly at each step, making a statistically dependent set of ensembles. Therefore, the loss function must be weighted appropriately, which in this case uses the probability that such sub-model is chosen.

$$\sum_{\mathcal{S}\subseteq\mathcal{N}} p(\mathcal{S})p_{\mathcal{S}}(y \mid x)$$

- Nevertheless, such sum has a exponential number of terms, making it intractable in practice. Instead, one would use an heuristic idea of simply scaling the weights by the probability of choosing such neuron at the end of training.

Regularization

Dropout:

- In PyTorch dropout is a `torch.Module` implemented as `text.DropOut`.
- For example, to add dropout to a network like this one

```
model = nn.Sequential(nn.Linear(10, 100), nn.ReLU(),
                      nn.Linear(100, 50), nn.ReLU(),
                      nn.Linear(50, 2));
```

we can simply add DropOut layers

```
model = nn.Sequential(nn.Linear(10, 100), nn.ReLU(),
                      nn.Dropout(),
                      nn.Linear(100, 50), nn.ReLU(),
                      nn.Dropout(),
                      nn.Linear(50, 2));
```

Regularization

Dropout:

- When you are using dropout in your model is necessary to set it in to “train” or “test” mode.
- The module class `nn.Module.train(mode)` sets

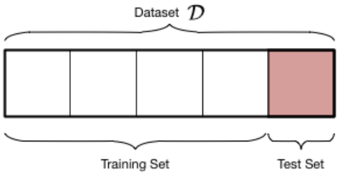
  ```
  dropout.training = True
  ```

  by default. To use it for prediction or testing, you must set this flag off.

  ```
  model.train(False)
  ```

Evaluation Protocols

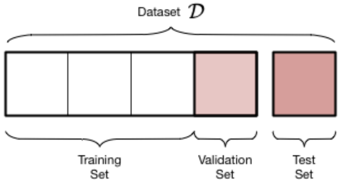
- The way we evaluate the generalization of a ML model usually demands a train test division of the available data



- Nevertheless, in many situations as we have seen, the tuning of hyperparameters might be necessary. A few examples are learning rates, the λ penalty for regularization, the exclusion probability on dropout, number of layers, number of neurons, etc...

Evaluation Protocols

- Using the training set for this search could lead to overfitting. Therefore, we instead use a **Validation set** to make such consideration



- We use something known as **structural risk minimization** principle

$$\hat{\lambda} = \operatorname*{argmin}_{\lambda} \hat{R}(\hat{\theta}_{\lambda})$$

Where $\hat{\theta}_{\lambda}$ is the estimator trained using the hyperparameter lambda, and $\hat{R}$ is an estimate of the risk (usually from the validation set)

Evaluation Protocols

Cross Validation

When data is scarce, where a new data subdivision could harm the performance of training, one can use the strategy of **cross validation (CV)**: *average through multiple random splits of the data in a train and a validation sets.*

Algorithm:

- Divide the train set of size N in K blocks of similar size, denote each block by $\mathcal{D}_k$, and the remaining data as $\mathcal{D}_{-k}$.
- Let $\mathcal{F}$ a learning algorithm that given a training set and hyperparameter vector λ returns an estimator
$$\hat{\theta}_\lambda = \mathcal{F}(\mathcal{D}, \lambda)$$
- The K-th fold CV of the risk is given by the average

$$\hat{R}(\lambda, \mathcal{D}, K) = \frac{1}{N} \sum_{k=1}^{K} \sum_{i \in D_k} \ell\left(y_i, f(x_i, \mathcal{F}(\mathcal{D}_{-\|}, \lambda))\right)$$