Training DNNs: Tricks

Ju Sun
Computer Science & Engineering
University of Minnesota, Twin Cities

March 5, 2020

Recap: last lecture

Training DNNs

$$\min_{\boldsymbol{W}} \frac{1}{m} \sum_{i=1}^{m} \ell\left(\boldsymbol{y}_i, \mathrm{DNN}_{\boldsymbol{W}}\left(\boldsymbol{x}_i\right)\right) + \Omega\left(\boldsymbol{W}\right)$$

- What methods? Mini-batch stochastic optimization due to large m
  * SGD (with momentum), Adagrad, RMSprop, Adam
  * diminishing LR (1/t, exp delay, staircase delay)
- Where to start?
  * Xavier init., Kaiming init., orthogonal init.
- When to stop?
  * early stopping: stop when validation error doesn't improve

This lecture: **additional tricks/heuristics that improve**

- **convergence speed**
- **task-specific (e.g., classification, regression, generation) performance**

Outline

Why scaling matters?

Consider a ML objective: $\min_{\boldsymbol{w}} \ f(\boldsymbol{w}) \doteq \frac{1}{m}\sum_{i=1}^{m} \ell(\boldsymbol{w}^\intercal \boldsymbol{x}_i; y_i)$, e.g.,

- Least-squares (LS): $\min_{\boldsymbol{w}} \ \frac{1}{m}\sum_{i=1}^{m} \|y_i - \boldsymbol{w}^\intercal \boldsymbol{x}_i\|_2^2$
- Logistic regression: $\min_{\boldsymbol{w}} \ -\frac{1}{m}\sum_{i=1}^{m} \left[y_i \boldsymbol{w}^\intercal \boldsymbol{x}_i - \log\left(1 + e^{\boldsymbol{w}^\intercal \boldsymbol{x}_i}\right)\right]$
- Shallow NN prediction: $\min_{\boldsymbol{w}} \ \frac{1}{m}\sum_{i=1}^{m} \|y_i - \sigma(\boldsymbol{w}^\intercal \boldsymbol{x}_i)\|_2^2$

Gradient: $\nabla_{\boldsymbol{w}} f = \frac{1}{m}\sum_{i=1}^{m} \ell'(\boldsymbol{w}^\intercal \boldsymbol{x}_i; y_i)\, \boldsymbol{x}_i$.

- What happens when coordinates (i.e., features) of $\boldsymbol{x}_i$ have different orders of magnitude? Partial derivatives have different orders of magnitudes $\Longrightarrow$ slow convergence of vanilla GD (recall why adaptive grad methods)

Hessian: $\nabla_{\boldsymbol{w}}^2 f = \frac{1}{m}\sum_{i=1}^{m} \ell''(\boldsymbol{w}^\intercal \boldsymbol{x}_i; y_i)\, \boldsymbol{x}_i \boldsymbol{x}_i^\intercal$.

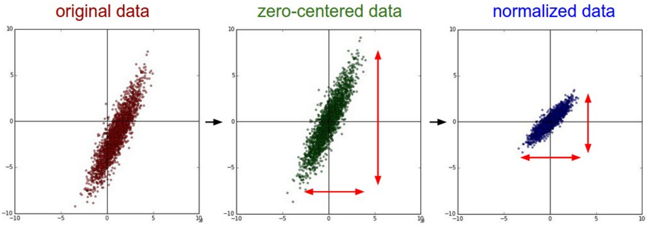
- Suppose the off-diagonal elements of $\boldsymbol{x}_i \boldsymbol{x}_i^\intercal$ are relatively small (e.g., when features are "independent").
- What happens when coordinates (i.e., features) of $\boldsymbol{x}_i$ have different orders of magnitude? Conditioning of $\nabla_{\boldsymbol{w}}^2 f$ is bad, i.e., f is elongated

Fix the scaling: first idea

Normalization: make each feature zero-mean and unit variance, i.e., make each row of $\boldsymbol{X} = [\boldsymbol{x}_1, \ldots, \boldsymbol{x}_m]$ zero-mean and unit variance, i.e.

$$\boldsymbol{X}' = \frac{\boldsymbol{X} - \boldsymbol{\mu}}{\boldsymbol{\sigma}} \quad (\boldsymbol{\mu}\text{—row means, } \boldsymbol{\sigma}\text{—row std, broadcasting applies})$$

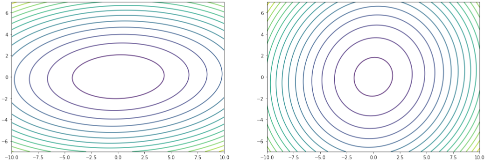
```
X = (X - X.mean(axis=1))/X.std(axis=1)
```



Credit: Stanford CS231N

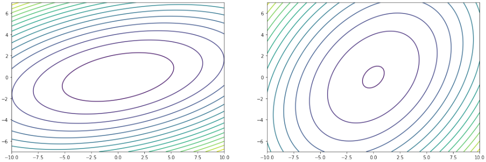
NB: for data matrices, we often assume each column is a data point, and each row is a feature. This convention is different from that assumed in Tensorflow and PyTorch.

For LS, works well when features are approximately independent



before vs. after the normalization

For LS, works not so well when features are highly dependent.



before vs. after the normalization

How to remove the feature dependency?

Fix the scaling: second idea

PCA and whitening

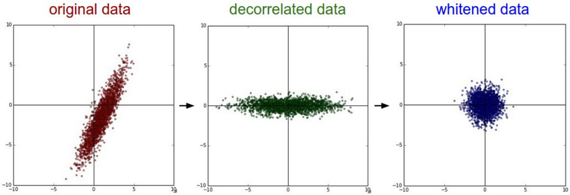
PCA, i.e., zero-center and rotate the data to align principal directions to coordinate directions

```
X -= X.mean(axis=1) #centering
U, S, VT = np.linalg.svd(X, full_matrices=False)
Xrot = U.T@X #rotate/decorrelate the data
```

(math: $\boldsymbol{X} = \boldsymbol{U}\boldsymbol{S}\boldsymbol{V}^\intercal$, then $\boldsymbol{U}^\intercal \boldsymbol{X} = \boldsymbol{S}\boldsymbol{V}$)

Whitening: PCA + normalize the coordinates by singular values

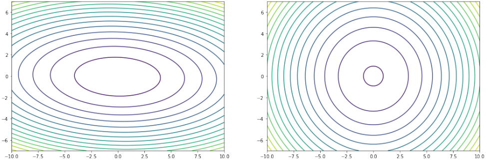
```
Xwhite =1/(S+eps)*Xrot
```
(math: $\boldsymbol{X}_{\text{white}} = \boldsymbol{V}$)



Credit: Stanford CS231N

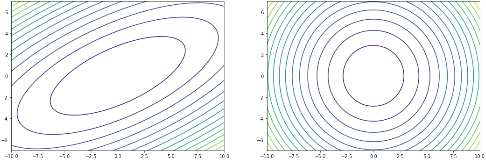
Fix the scaling: second idea

For LS, works well when features are approximately independent



before vs. after the whitening

For LS, also works well when features are highly dependent.



before vs. after the whitening

In DNNs practice

fixing the feature scaling makes the landscape "nicer"—derivatives and curvatures in all directions are roughly even in magnitudes. So for DNNs,

- Preprocess the input data
  - zero-center
  - normalization
  - PCA or whitening (less common)
- But recall our model objective $\min_{\boldsymbol{w}}\ f(\boldsymbol{w}) \doteq \frac{1}{m}\sum_{i=1}^{m} \ell(\boldsymbol{w}^\intercal \boldsymbol{x}_i; y_i)$ vs. DL objective

$$\min_{\boldsymbol{W}}\ \frac{1}{m}\sum_{i=1}^{m} \ell\left(\boldsymbol{y}_i, \sigma\left(\boldsymbol{W}_k \sigma\left(\boldsymbol{W}_{k-1} \ldots \sigma\left(\boldsymbol{W}_1 \boldsymbol{x}_i\right)\right)\right)\right) + \Omega(\boldsymbol{W})$$

  - DL objective is much more complex
  - But $\sigma\left(\boldsymbol{W}_k \sigma\left(\boldsymbol{W}_{k-1} \ldots \sigma\left(\boldsymbol{W}_1 \boldsymbol{x}_i\right)\right)\right)$ is a composite version of $\boldsymbol{w}^\intercal \boldsymbol{x}_i$:

$$\boldsymbol{W}_1 \boldsymbol{x}_i,\ \boldsymbol{W}_2 \sigma\left(\boldsymbol{W}_1 \boldsymbol{x}_i\right),\ \boldsymbol{W}_3 \sigma\left(\boldsymbol{W}_2 \sigma\left(\boldsymbol{W}_1 \boldsymbol{x}_i\right)\right),\ \ldots$$

- **Idea: also process the input data to some/all hidden layers**

Batch normalization

Apply normalization to the input data to some/all hidden layers

- $\sigma\left(\boldsymbol{W}_k \sigma\left(\boldsymbol{W}_{k-1} \ldots \sigma\left(\boldsymbol{W}_1 \boldsymbol{x}_i\right)\right)\right)$ is a composite version of $\boldsymbol{w}^\intercal \boldsymbol{x}_i$:

$$\boldsymbol{W}_1 \boldsymbol{x}_i,\ \boldsymbol{W}_2 \sigma\left(\boldsymbol{W}_1 \boldsymbol{x}_i\right),\ \boldsymbol{W}_3 \sigma\left(\boldsymbol{W}_2 \sigma\left(\boldsymbol{W}_1 \boldsymbol{x}_i\right)\right),\ \ldots$$

- Apply **normalization** to the outputs of the colored parts based on the statistics of a **mini-batch** of $\boldsymbol{x}_i$'s, e.g.,

$$\boldsymbol{W}_2 \underbrace{\sigma\left(\boldsymbol{W}_1 \boldsymbol{x}_i\right)}_{\doteq \boldsymbol{z}_i} \longrightarrow \boldsymbol{W}_2 \underbrace{\mathrm{BN}\left(\sigma\left(\boldsymbol{W}_1 \boldsymbol{x}_i\right)\right)}_{\mathrm{BN}(\boldsymbol{z}_i)}$$

- Let $\boldsymbol{z}_i$'s be generated from a mini-batch of $\boldsymbol{x}_i$'s and $\boldsymbol{Z} = \left[\boldsymbol{z}_1 \ldots \boldsymbol{z}_{|B|}\right]$,

$$\mathrm{BN}\left(\boldsymbol{z}^j\right) = \frac{\boldsymbol{z}^j - \mu_{\boldsymbol{z}^j}}{\sigma_{\boldsymbol{z}^j}} \quad \text{for each row } \boldsymbol{z}^j \text{ of } \boldsymbol{Z}.$$

Flexibity restored by optional scaling γ_j's and shifting β_j's:

$$\mathrm{BN}_{\gamma_j, \boldsymbol{\beta}_j}\left(\boldsymbol{z}^j\right) = \gamma_j \frac{\boldsymbol{z}^j - \mu_{\boldsymbol{z}^j}}{\sigma_{\boldsymbol{z}^j}} + \beta_j \quad \text{for each row } \boldsymbol{z}^j \text{ of } \boldsymbol{Z}.$$

Here, γ_j's and β's are trainable (optimization) variables!

Batch normalization: implementation details

$$\boldsymbol{W}_2 \underbrace{\sigma\left(\boldsymbol{W}_1 \boldsymbol{x}_i\right)}_{\doteq \boldsymbol{z}_i} \longrightarrow \boldsymbol{W}_2 \underbrace{\mathrm{BN}\left(\sigma\left(\boldsymbol{W}_1 \boldsymbol{x}_i\right)\right)}_{\mathrm{BN}(\boldsymbol{z}_i)} \qquad \mathrm{BN}_{\gamma_j, \boldsymbol{\beta}_j}\left(\boldsymbol{z}^j\right) = \gamma_j \frac{\boldsymbol{z}^j - \mu_{\boldsymbol{z}^j}}{\sigma_{\boldsymbol{z}^j}} + \beta_j \ \forall\ j$$

Question: how to perform training after plugging in the BN operations?

$$\min_{\boldsymbol{W}} \ \frac{1}{m} \sum_{i=1}^m \ell\left(\boldsymbol{y}_i, \sigma\left(\boldsymbol{W}_k \mathrm{BN}\left(\sigma\left(\boldsymbol{W}_{k-1} \dots \mathrm{BN}\left(\sigma\left(\boldsymbol{W}_1 \boldsymbol{x}_i\right)\right)\right)\right)\right)\right) + \Omega\left(\boldsymbol{W}\right)$$

Answer: for all j, $\mathrm{BN}_{\gamma_j, \boldsymbol{\beta}_j}\left(\boldsymbol{z}^j\right)$ is nothing but a differentiable function of $\boldsymbol{z}^j$, γ_j, and β_j — chain rule applies!

- $\mu_{\boldsymbol{z}^j}$ and $\sigma_{\boldsymbol{z}^j}$ are differentiable functions of $\boldsymbol{z}^j$, and $\left(\boldsymbol{z}^j, \gamma_j, \beta_j\right) \mapsto \mathrm{BN}_{\gamma_j, \boldsymbol{\beta}_j}\left(\boldsymbol{z}^j\right)$ is a vector-to-vector mapping
- Any row $\boldsymbol{z}^j$ depends on all $\boldsymbol{x}_k$'s in the current mini-batch B as $\boldsymbol{Z} = [\boldsymbol{z}_1 \dots \boldsymbol{z}_{|B|}] \longleftarrow [\boldsymbol{x}_1 \dots \boldsymbol{x}_{|B|}]$
- Without BN: $\nabla_{\boldsymbol{W}} \frac{1}{|B|} \sum_{k=1}^{|B|} \ell\left(\boldsymbol{W}; \boldsymbol{x}_k, \boldsymbol{y}_k\right) = \frac{1}{|B|} \sum_{k=1}^{|B|} \nabla_{\boldsymbol{W}} \ell\left(\boldsymbol{W}; \boldsymbol{x}_k, \boldsymbol{y}_k\right)$, the summands can be computed in parallel and then aggregated
  With BN: $\nabla_{\boldsymbol{W}} \frac{1}{|B|} \sum_{k=1}^{|B|} \ell\left(\boldsymbol{W}; \boldsymbol{x}_k, \boldsymbol{y}_k\right)$ has to be computed altogether, due to the inter-dependency across the summands

Batch normalization: implementation details

$$\mathrm{BN}_{\gamma_j, \boldsymbol{\beta}_j}\left(\boldsymbol{z}^j\right) = \gamma_j \frac{\boldsymbol{z}^j - \mu_{\boldsymbol{z}^j}}{\sigma_{\boldsymbol{z}^j}} + \beta_j \;\forall\; j$$

What about validation/test, where only a single sample is seen each time?

idea: use the average $\mu_{\boldsymbol{z}^j}$'s and $\sigma_{\boldsymbol{z}^j}$'s over the training data (γ_j's and β_j's are learned)

In practice, collect the momentum-weighted running averages: e.g., for each hidden node with BN,

$$\overline{\mu} = (1-\eta)\,\overline{\mu}_{old} + \eta\mu_{new}$$
$$\overline{\sigma} = (1-\eta)\,\overline{\sigma}_{old} + \eta\sigma_{new}$$

with e.g., $\eta = 0.1$. In PyTorch, `torch.nn.BatchNorm1d`, `torch.nn.BatchNorm2d`, `torch.nn.BatchNorm3d` depending on the input shapes

Batch normalization: implementation details

Question: BN before or after the activation?

$$\boldsymbol{W}_2\sigma\left(\boldsymbol{W}_1\boldsymbol{x}_i\right) \longrightarrow \boldsymbol{W}_2\mathrm{BN}\left(\sigma\left(\boldsymbol{W}_1\boldsymbol{x}_i\right)\right) \quad \text{(after)}$$
$$\boldsymbol{W}_2\sigma\left(\boldsymbol{W}_1\boldsymbol{x}_i\right) \longrightarrow \boldsymbol{W}_2\left(\sigma\left(\mathrm{BN}\left(\boldsymbol{W}_1\boldsymbol{x}_i\right)\right)\right) \quad \text{(before)}$$

- The original paper [Ioffe and Szegedy, 2015] proposed the "before" version (most of the original intuition has since proved wrong)
- But the "after" version is more intuitive as we have seen
- Both are used in practice and debatable which one is more effective
  * https://www.reddit.com/r/MachineLearning/comments/67gonq/d_batch_normalization_before_or_after_relu/
  * https://blog.paperspace.com/busting-the-myths-about-batch-normalization/
  * https://github.com/gcr/torch-residual-networks/issues/5
  * [Chen et al., 2019]

Why BN works?

Short answer: we don't know yet

Long answer:

- Originally proposed to deal with *internal covariate shift* [Ioffe and Szegedy, 2015]
- The original intuition later proved wrong and BN is shown to make the optimization problem "nicer" (or "smoother") [Santurkar et al., 2018, Lipton and Steinhardt, 2019]
- Yet another explanation from optimization perspective [Kohler et al., 2019]
- A good research topic

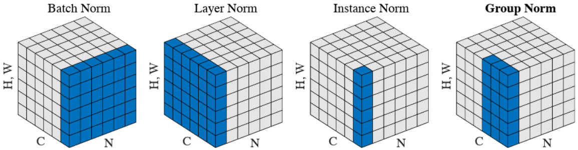
Batch PCA/whitening?

fixing the feature scaling makes the landscape "nicer"—derivatives and curvatures in all directions are roughly even in magnitudes. So for DNNs,

- Add (pre-)processing to input data
  * zero-center
  * normalization
  * PCA or whitening (less common)
- Add batch-processing steps to some/all hidden layers
  * Batch normalization
  * Batch PCA or whitening? Doable but requires a lot of work [Huangi et al., 2018, Huang et al., 2019, Wang et al., 2019]

normalization is most popular due to the simplicity

Zoo of normalization



Normalization methods. Each subplot shows a feature map tensor, with N as the batch axis, C as the channel axis, and (H, W) as the spatial axes. The pixels in blue are normalized by the same mean and variance, computed by aggregating the values of these pixels.

Credit: [Wu and He, 2018]

normalization in different directions/groups of the data tensors

weight normalization: decompose the weight as magnitude and direction $\boldsymbol{w} = g \frac{\boldsymbol{v}}{\|\boldsymbol{v}\|_2}$ and perform optimization in $(g, \boldsymbol{v})$ space

An Overview of Normalization Methods in Deep Learning https://mlexplained.com/2018/11/30/an-overview-of-normalization-methods-in-deep-learning/

Check out PyTorch normalization layers https://pytorch.org/docs/stable/nn.html#normalization-layers

Outline

Data Normalization

Regularization

Hyperparameter search, data augmentation

Suggested reading

Regularization to avoid overfitting

Training DNNs $\min_{\boldsymbol{W}} \frac{1}{m}\sum_{i=1}^{m} \ell\left(\boldsymbol{y}_i, \mathrm{DNN}_{\boldsymbol{W}}\left(\boldsymbol{x}_i\right)\right) + \lambda\Omega\left(\boldsymbol{W}\right)$ with **explicit regularization** Ω. But which Ω?

- $\Omega\left(\boldsymbol{W}\right) = \sum_k \left\|\boldsymbol{W}_k\right\|_F^2$ where k indexes the layers — penalizes large values in $\boldsymbol{W}$ and hence avoids steep changes (set `weight_decay` as λ in `torch.optim.xxxx`)
- $\Omega\left(\boldsymbol{W}\right) = \sum_k \left\|\boldsymbol{W}_k\right\|_1$ — promotes sparse $\boldsymbol{W}_k$'s (i.e., many entries in $\boldsymbol{W}_k$'s to be near zero; good for feature selection)

```
l1_reg = torch.zeros(1)
for W in model.parameters():
    l1_reg += W.norm(1)
```

- $\Omega\left(\boldsymbol{W}\right) = \left\|\boldsymbol{J}_{\mathrm{DNN}_{\boldsymbol{W}}}\left(\boldsymbol{x}\right)\right\|_F^2$ — promotes smoothness of the function represented by $\mathrm{DNN}_{\boldsymbol{W}}$ [Varga et al., 2017, Hoffman et al., 2019, Chan et al., 2019]
- Constraints, $\delta_C\left(\boldsymbol{W}\right) \doteq \begin{cases} 0 & \boldsymbol{W} \in C \\ \infty & \boldsymbol{W} \notin C \end{cases}$, e.g., binary, norm bound
- many others!

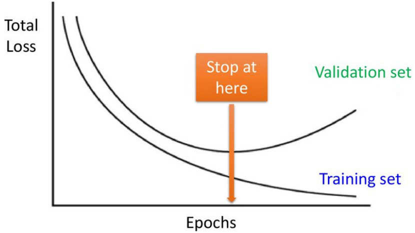
Implicit regularization

Training DNNs $\min_{\boldsymbol{W}} \ \frac{1}{m}\sum_{i=1}^{m} \ell\left(\boldsymbol{y}_i, \mathrm{DNN}_{\boldsymbol{W}}\left(\boldsymbol{x}_i\right)\right) + \lambda\Omega\left(\boldsymbol{W}\right)$ with **implicit regularization** — operation that is not built into the objective but avoids overfitting

- early stopping
- batch normalization
- dropout
- ...

Early stopping

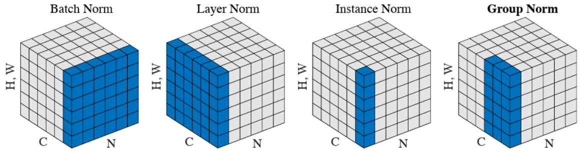
A practical/pragmatic stopping strategy: **early stopping**



... periodically check the validation error and stop when it doesn't improve

Intuition: avoid the model to be too specialized/perfect for the training data

More concrete math examples: [Bishop, 1995, Sjöberg and Ljung, 1995]



Normalization methods. Each subplot shows a feature map tensor, with N as the batch axis, C as the channel axis, and (H, W) as the spatial axes. The pixels in blue are normalized by the same mean and variance, computed by aggregating the values of these pixels.

Credit: [Wu and He, 2018]

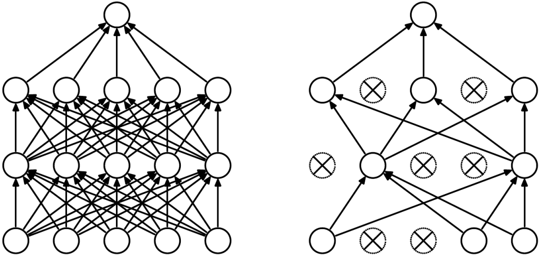
normalization in different directions/groups of the data tensors

weight normalization: decompose the weight as magnitude and direction $\boldsymbol{w} = g \frac{\boldsymbol{v}}{\|\boldsymbol{v}\|_2}$ and perform optimization in $(g, \boldsymbol{v})$ space

An Overview of Normalization Methods in Deep Learning

https://mlexplained.com/2018/11/30/an-overview-of-normalization-methods-in-deep-learning/

Dropout



(a) Standard Neural Net (b) After applying dropout.

Credit: [Srivastava et al., 2014]

Idea: kill each non-output neuron with probability $1 - p$, called **Dropout**

- perform Dropout independently for each training sample and each iteration
- for each neuron, if the original output is x, then the expected output with Dropout: px. So rescale the actual output by $1/p$
- no Dropout at test time!

Dropout: implementation details

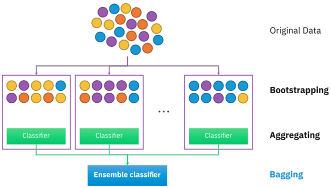
```
p = 0.5 # probability of keeping a unit active. higher = less dropout

def train_step(X):
  # forward pass for example 3-layer neural network
  H1 = np.maximum(0, np.dot(W1, X) + b1)
  U1 = (np.random.rand(*H1.shape) < p) / p # first dropout mask. Notice /p!
  H1 *= U1 # drop!
  H2 = np.maximum(0, np.dot(W2, H1) + b2)
  U2 = (np.random.rand(*H2.shape) < p) / p # second dropout mask. Notice /p!
  H2 *= U2 # drop!
  out = np.dot(W3, H2) + b3

  # backward pass: compute gradients... (not shown)
  # perform parameter update... (not shown)

def predict(X):
  # ensembled forward pass
  H1 = np.maximum(0, np.dot(W1, X) + b1) # no scaling necessary
  H2 = np.maximum(0, np.dot(W2, H1) + b2)
  out = np.dot(W3, H2) + b3
```

Credit: Stanford CS231N

What about derivatives? Back-propagation for each sample and then aggregate

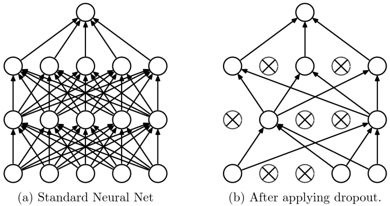
PyTorch: `torch.nn.Dropout`, `torch.nn.Dropout2d`, `torch.nn.Dropout3d`

Why Dropout?



Credit: Wikipedia

bagging can avoid overfitting



Credit: [Srivastava et al., 2014]

For an n-node network, 2^n possible sub-networks.

Consider the average/ensemble prediction $\mathbb{E}_{\mathrm{SN}}\left[\mathrm{SN}\left(\boldsymbol{x}\right)\right]$ over 2^n of sub-networks and the new objective

$$F\left(\boldsymbol{W}\right) \doteq \frac{1}{m}\sum_{i=1}^{m} \ell\left(\boldsymbol{y}_i, \mathbb{E}_{\mathrm{SN}}\left[\mathrm{SN}_{\boldsymbol{W}}\left(\boldsymbol{x}_i\right)\right]\right)$$

Mini-batch SGD with Dropout samples data point and model simultaneously (stochastic composite optimization [Wang et al., 2016, Wang et al., 2017])

Outline

Data Normalization

Regularization

Hyperparameter search, data augmentation

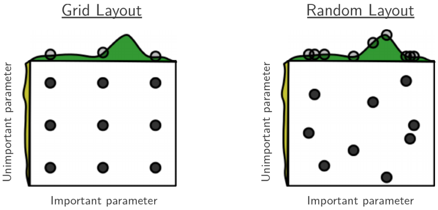
Suggested reading

Hyperparameter search

...tunable parameters (vs. learnable parameters, or optimization variables)

- Network architecture (depth, width, activation, loss, etc)
- Optimization methods
- Initialization schemes
- Initial LR and LR schedule/parameters
- regularization methods and parameters
- etc

https://cs231n.github.io/neural-networks-3/#hyper



Credit: [Bergstra and Bengio, 2012]

Data augmentation

- More relevant data always help!
- Fetch more external data
- Generate more internal data: generate based on whatever you want to be robust to
  - * vision: translation, rotation, background, noise, deformation, flipping, blurring, occlusion, etc

Credit: https://github.com/aleju/imgaug

See one example here https://pytorch.org/tutorials/beginner/transfer_learning_tutorial.html

Outline

Data Normalization

Regularization

Hyperparameter search, data augmentation

Suggested reading

Suggested reading

- Chap 7, Deep Learning (Goodfellow et al)
- Stanford CS231n course notes: Neural Networks Part 2: Setting up the Data and the Loss https://cs231n.github.io/neural-networks-2/
- Stanford CS231n course notes: Neural Networks Part 3: Learning and Evaluation https://cs231n.github.io/neural-networks-3/
- http://neuralnetworksanddeeplearning.com/chap3.html

References i

[Bergstra and Bengio, 2012] Bergstra, J. and Bengio, Y. (2012). **Random search for hyper-parameter optimization.** *Journal of machine learning research*, 13(Feb):281–305.

[Bishop, 1995] Bishop, C. M. (1995). **Regularization and complexity control in feed-forward networks.** In *International Conference on Artificial Neural Networks ICANN*.

[Chan et al., 2019] Chan, A., Tay, Y., Ong, Y. S., and Fu, J. (2019). **Jacobian adversarially regularized networks for robustness.** *arXiv:1912.10185*.

[Chen et al., 2019] Chen, G., Chen, P., Shi, Y., Hsieh, C.-Y., Liao, B., and Zhang, S. (2019). **Rethinking the usage of batch normalization and dropout in the training of deep neural networks.** *arXiv:1905.05928*.

[Hoffman et al., 2019] Hoffman, J., Roberts, D. A., and Yaida, S. (2019). **Robust learning with jacobian regularization.** *arXiv:1908.02729*.

[Huang et al., 2019] Huang, L., Zhou, Y., Zhu, F., Liu, L., and Shao, L. (2019). **Iterative normalization: Beyond standardization towards efficient whitening.** pages 4869–4878. IEEE.

[Huangi et al., 2018] Huangi, L., Huangi, L., Yang, D., Lang, B., and Deng, J. (2018). **Decorrelated batch normalization.** pages 791–800. IEEE.

[Ioffe and Szegedy, 2015] Ioffe, S. and Szegedy, C. (2015). **Batch normalization: Accelerating deep network training by reducing internal covariate shift.** In *The 32nd International Conference on Machine Learning*.

[Kohler et al., 2019] Kohler, J. M., Daneshmand, H., Lucchi, A., Hofmann, T., Zhou, M., and Neymeyr, K. (2019). **Exponential convergence rates for batch normalization: The power of length-direction decoupling in non-convex optimization.** In *The 22nd International Conference on Artificial Intelligence and Statistics*.

[Lipton and Steinhardt, 2019] Lipton, Z. C. and Steinhardt, J. (2019). **Troubling trends in machine learning scholarship.** *ACM Queue*, 17(1):80.

[Santurkar et al., 2018] Santurkar, S., Tsipras, D., Ilyas, A., and Madry, A. (2018). **How does batch normalization help optimization?** In *Advances in Neural Information Processing Systems*, pages 2483–2493.

[Sjöberg and Ljung, 1995] Sjöberg, J. and Ljung, L. (1995). **Overtraining, regularization and searching for a minimum, with application to neural networks.** *International Journal of Control*, 62(6):1391–1407.

[Srivastava et al., 2014] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). **Dropout: a simple way to prevent neural networks from overfitting.** *The journal of machine learning research*, 15(1):1929–1958.

[Varga et al., 2017] Varga, D., Csiszárik, A., and Zombori, Z. (2017). **Gradient regularization improves accuracy of discriminative models.** *arXiv:1712.09936*.

[Wang et al., 2016] Wang, M., Fang, E. X., and Liu, H. (2016). **Stochastic compositional gradient descent: algorithms for minimizing compositions of expected-value functions.** *Mathematical Programming*, 161(1-2):419–449.

[Wang et al., 2017] Wang, M., Liu, J., and Fang, E. X. (2017). **Accelerating stochastic composition optimization.** *The Journal of Machine Learning Research*, 18(1):3721–3743.

[Wang et al., 2019] Wang, W., Dang, Z., Hu, Y., Fua, P., and Salzmann, M. (2019). **Backpropagation-friendly eigendecomposition.** In *Advances in Neural Information Processing Systems*, pages 3156–3164.

[Wu and He, 2018] Wu, Y. and He, K. (2018). **Group normalization.** In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 3–19.