CS-568 Deep Learning

Nazar Khan

PUCIT

Regularization in Neural Networks

Before we start

1. Capabilities of polynomials (lines, quadratics, cubics, . . . , degree M).
2. Capability can be reduced by restricting coefficients.
3. Everything is noisy.
4. Zero *training* error is bad. Over-fitting vs generalisation.

Regularization in Neural Networks

- Over-fitting can be reduced via regularization.
  1. Penalise magnitudes of weights: $\tilde{E}(\mathbf{w}) = E(\mathbf{w}) + \frac{\lambda}{2}\|\mathbf{w}\|^2$.
  2. Separately penalise each layer: $\tilde{E}(\mathbf{w}) = E(\mathbf{w}) + \sum_{l=1}^{L} \frac{\lambda_l}{2}\|\mathbf{w}^{(l)}\|^2$.
  3. *Dropout*: *During training*, a randomly selected subset of activations are set to zero within each layer.
  4. *Batch Normalization*
  5. *Early stopping* by checking $E(\mathbf{w})$ on a validation set. Stop when error on validation set starts increasing.
  6. *Data Augmentation*: Training with augmented/transformed data.
  7. *Label Smoothing*
  8. Building invariance into the network structure (to be covered in the Convolutional Neural Networks lecture).

Droput

- One of the most used regularization techniques in neural nets.
- *During training*, a randomly selected subset of activations are set to zero within each layer.
- This makes the neural network less powerful.
- Dropout layer implementation is very simple.
  - For each neuron (including inputs), generate a uniform random number between 0 and 1.
  - If the number is greater than α, set the neuron's output to 0.
  - Otherwise, don't touch the neuron's output.
- Probably of dropping out is $1 - \alpha$.
- *Remember* which neurons were dropped so that gradients are also zeroed out during backpropagation.

Detour – Bagging

- Bagging is a popular ML meta-algorithm.
- Multiple ML models are trained separately to solve the same problem on *separate subsets* of the training data.
- Final answer is the average of all models.

$$F(x) = \frac{1}{M}\sum_{m=1}^{M} f_m(x)$$

- Bagging results are usually better than the best individual model.
- Dropout can be viewed as bagging.

Dropout as Bagging

- An architecture with n neurons can have 2^n sub-architectures depending on which neurons are switched off.
- Whenever a random subset of neurons is switched off, we are essentially training only one of the 2^n sub-architectures.
- At test time, use expected output of neuron, $E[y] = \alpha h(a)$, i.e., bagging.
- Alternatives:
  1. Push α into the next layer's weights after training and do testing as before.

$$
\begin{aligned}
z_k &= \sum w_{kj} y_j + b_k \\
&= \sum w_{kj} \alpha h(a_j) + b_k = \sum \underbrace{(\alpha w_{kj})}_{\tilde{w}_{kj}} h(a_j) + b_k
\end{aligned}
$$

  2. During training, multiply every output by $\frac{1}{\alpha}$ and do testing as before.

Dropout vs. DropConnect

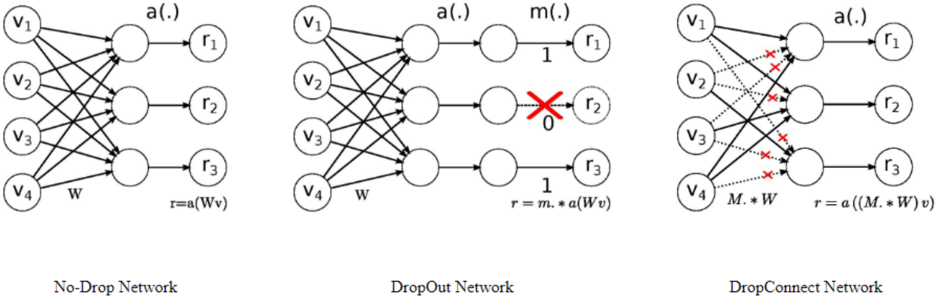
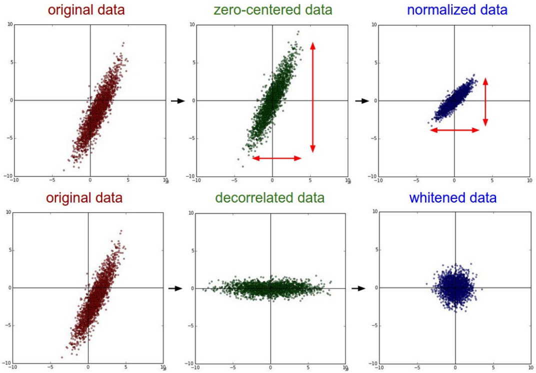


Figure: Dropout vs. DropConnect[2]. Image taken from
`https://cs.nyu.edu/~wanli/dropc/`

[2]Wan et al., 'Regularization of Neural Network using DropConnect'.

Normalisation

- The importance of normalising inputs is well-understood in ML.
- Improves numerical stability and reduces training time.
- Makes all features equally important before learning takes place.



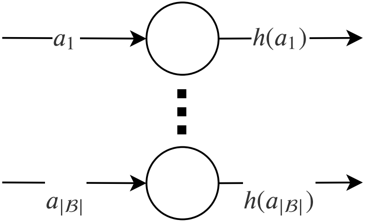
Normalisation of 2D data. Taken from
`http://cs231n.github.io/neural-networks-2/`

Batch Normalisation

- In neural networks, a neuron's input depends on previous neurons' outputs.
- Those outputs can vary wildly during training as the weights are adjusted.
- Normalising the input sample is not enough.
- Later neuron's input needs to be normalised as well.
- Inputs to every neuron in every layer must be normalised *in a differentiable manner*.
- Normalisation is useless for learning if gradient ignores it.

Batch Normalisation

- For the i-th input sample, a neuron passes its pre-activation a_i into its activation function $h(a_i)$.
- For a minibatch $\mathcal{B}$, the neuron will perform this step for each input sample in $\mathcal{B}$ *separately*.

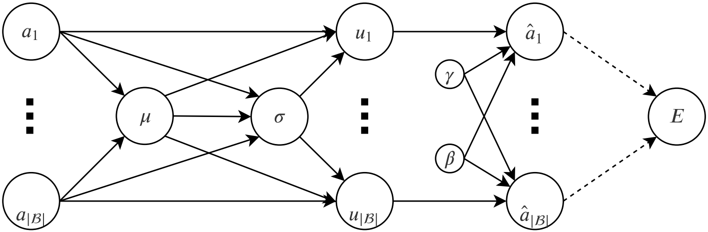


- Batchnorm takes place between this step.
- Each a_i is converted to $\hat{a}_i$ by looking at the other a_j values in the minibatch.
- Instead of a_i, the new $\hat{a}_i$ is passed into the acitvation function.

Batch Normalisation

Consider a neuron's pre-activations $a_1, a_2, \ldots, a_{|\mathcal{B}|}$ over a minibatch $\mathcal{B}$.

1. Compute mean $\mu = \frac{\sum a_i}{|\mathcal{B}|}$ and variance $\sigma^2 = \frac{\sum(a_i-\mu)^2}{|\mathcal{B}|-1}$.
2. Standardize the pre-activations as $u_i = \frac{a_i-\mu}{\sigma}$.
   This makes the set $u_1, u_2, \ldots, u_{|\mathcal{B}|}$ have zero-mean and unit-variance.
3. Recover expressive power by learnable transformation $\hat{a}_i = \gamma u_i + \beta$.



Batch Normalisation

The $\hat{a}_i$ values that are now passed into the activation function will have mean β and standard deviation γ, *irrespective of original moments μ and σ* for the minibatch.

The whole process is differentiable and therefore suitable for gradient descent.

Benefits of BatchNorm

- Avoids vanishing gradients for sigmoidal non-linearities.
- Allows much higher learning rates and therefore dramatically speeds up training.
- Reduces dependence on good weight initialisation.
- Regularizes the model and reduces the need for dropout.

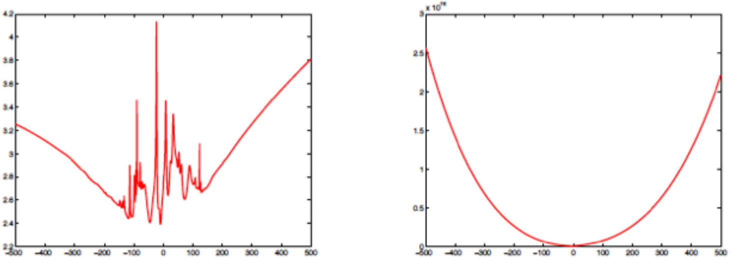
Why does Batchnorm work?

- The original paper[3] posited that Batchnorm succeeded by reducing *internal covariate shift (ICS)*.
  - ICS: Earlier neurons causing changes in distribution of inputs to subsequent neurons.
  - Causing later neurons to remain confused about which distribution to learn over.
  - Increases time to converge.
- Recent work[4] suggests that BatchNorm's might not even be reducing ICS.
  - Infact, ICS might not even be a problem.
  - Batchnorm succeeds because it has a regularization effect.
  - It reduces the values *and the gradients* of the loss function.

[3] Ioffe and Szegedy, 'Batch normalization: Accelerating deep network training by reducing internal covariate shift'

[4] Santurkar et al., *How Does Batch Normalization Help Optimization?*

Why does Batchnorm work?



Learning over smooth landscapes (right) is more stable and faster since we can increase learning rate without over-shooting. This figure is illustrative – effect of batchnorm is not as drastic.

Why does Batchnorm work?

- Another[5] suggestion is that it makes the learning problem easier.
- By decoupling the problems of estimation of direction and magnitude of the weight vector.
- Direction of the weight vector is learned separately from its size.

[5]Kohler et al., *Exponential convergence rates for Batch Normalization: The power of length-direction decoupling in non-convex optimization*

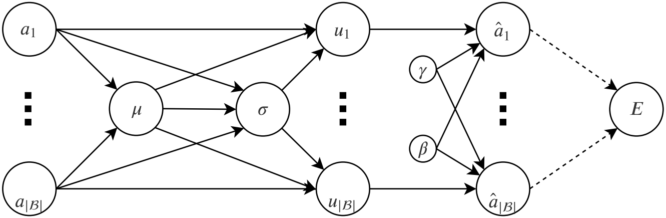
Derivatives

- Consider the j-th neuron in the l-th layer.
- Let $z_i = h(\hat{a}_i)$ be the neuron's output for the i-th sample in minibatch $\mathcal{B}$.

$$\hat{a}_i = \gamma u_i + \beta$$

$$u_i = \frac{a_i - \mu}{\sqrt{\sigma^2 + \epsilon}}$$

$$\mu = \frac{\sum a_j}{|\mathcal{B}|} \text{ and } \sigma^2 = \frac{\sum (a_j - \mu)^2}{|\mathcal{B}|}$$



Derivatives

- Recall that we can compute $\delta_i = \frac{\partial L}{\partial \hat{a}_i}$ via backpropagation as

$$\delta_i = h'(\hat{a}_i) \sum_{k=1}^{K} \delta_k w_{kj}^{(l+1)}$$

- So we will assume that we have already computed $\frac{\partial L}{\partial \hat{a}_i}$ via backpropagation.
- Then $\frac{\partial L}{\partial u_i} = \frac{\partial L}{\partial \hat{a}_i} \frac{\partial \hat{a}_i}{\partial u_i} = \delta_i \gamma$.
- *Goal*: Compute $\frac{\partial L}{\partial a_i}$ and proceed with backpropagation from there.
- *Direct* affectees of a_i are: u_i, μ and σ^2.
- So treat loss function as $L(u_i(a_i), \mu(a_i), \sigma(a_i))$.
- Using multivariate chain rule

$$\frac{\partial L}{\partial a_i} = \frac{\partial L}{\partial u_i} \frac{\partial u_i}{\partial a_i} + \frac{\partial L}{\partial \mu} \frac{\partial \mu}{\partial a_i} + \frac{\partial L}{\partial \sigma^2} \frac{\partial \sigma^2}{\partial a_i}$$

Derivatives

- Using multivariate chain rule

$$
\begin{aligned}
\frac{\partial L}{\partial a_i} &= \frac{\partial L}{\partial u_i}\frac{\partial u_i}{\partial a_i} + \frac{\partial L}{\partial \mu}\frac{\partial \mu}{\partial a_i} + \frac{\partial L}{\partial \sigma^2}\frac{\partial \sigma^2}{\partial a_i} \\
&= \frac{\partial L}{\partial u_i}\frac{\partial u_i}{\partial a_i} + \left(\sum_{\mathcal{B}} \frac{\partial L}{\partial u_j}\frac{\partial u_j}{\partial \mu}\right)\frac{\partial \mu}{\partial a_i} + \left(\sum_{\mathcal{B}} \frac{\partial L}{\partial u_j}\frac{\partial u_j}{\partial \sigma^2}\right)\frac{\partial \sigma^2}{\partial a_i} \\
&= \frac{\partial L}{\partial u_i}\frac{1}{\sqrt{\sigma^2+\epsilon}} + \sum_{\mathcal{B}} \frac{\partial L}{\partial u_j}\frac{-1}{\sqrt{\sigma^2+\epsilon}}\frac{1}{|\mathcal{B}|} + \\
&\qquad \sum_{\mathcal{B}} \frac{\partial L}{\partial u_j}\left(-\frac{1}{2}\frac{a_j-\mu}{(\sigma^2+\epsilon)^{\frac{3}{2}}}\right)\left(\frac{2(a_i-\mu)}{|\mathcal{B}|} + \underbrace{\sum_{\mathcal{B}} \frac{2(a_j-\mu)}{|\mathcal{B}|}\left(-\frac{1}{|\mathcal{B}|}\right)}_{=0}\right) \\
&= \frac{\partial L}{\partial u_i}\frac{1}{\sqrt{\sigma^2+\epsilon}} - \frac{1}{|\mathcal{B}|\sqrt{\sigma^2+\epsilon}}\sum_{\mathcal{B}}\frac{\partial L}{\partial u_j} - \frac{(a_i-\mu)}{|\mathcal{B}|(\sigma^2+\epsilon)^{\frac{3}{2}}}\sum_{\mathcal{B}}\frac{\partial L}{\partial u_j}(a_j-\mu)
\end{aligned}
$$

Batchnorm at testing time

- Testing is not done on minibatches.
- But each neuron trained itself on batchnormed pre-activations.
- It expects batchnormed pre-activations at testing time as well.
- *Solution*: Once the network is trained, for *each neuron*, compute the average μ, σ^2 over the set $\mathcal{S}$ of all training minibatches.

$$\mu_{\text{test}} = \frac{1}{|\mathcal{S}|} \sum_{\mathcal{B} \in \mathcal{S}} \mu(\mathcal{B})$$

$$\sigma^2_{\text{test}} = \frac{|\mathcal{B}|}{|\mathcal{B}| - 1} \frac{1}{|\mathcal{S}|} \sum_{\mathcal{B} \in \mathcal{S}} \sigma^2(\mathcal{B})$$

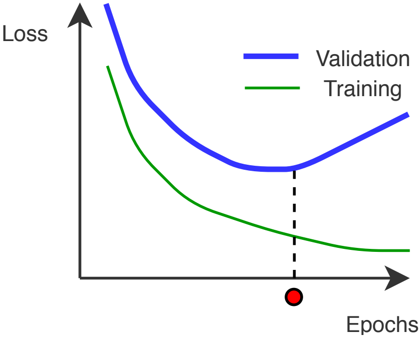
- $\frac{|\mathcal{B}|}{|\mathcal{B}|-1}$ for computing unbiased estimator of variance.
- Use $\mu_{\text{test}}, \sigma_{\text{test}}$ to normalize every testing sample.

So is Batchnorm legit?

- Around 2006, deep networks were successfully trained using greedy, unsupervised, layer-wise pretraining.
- The method worked but was unintuitive. Why should pretraining avoid vanishing gradients?
- We now know that greedy layer-wise pretraining is not necessary for deep networks.
- Batchnorm has the same feel to it.
- It works (extremely well) but what's the intuition behind making the i-th training sample's output *dependent* on other *randomly chosen* training samples?
- The jury is still out on Batchnorm.

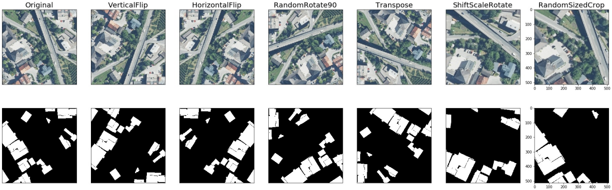
Early Stopping

- Split some part of the training set into a validation set that will not be used for training.
- During training, record loss on training as well as validation set.
- When validation loss starts increasing while training loss is still going down, the model has started overfitting.
- So stop training at that point.



Data Augmentation

- Augment training set with transformed versions of training samples.
- Domain specific data augmentations
  - Images: Color, Geometry
  - Text: Synonyms, Tense, Order
  - Speech: Speed, Sound effects



https://github.com/albumentations-team/albumentations

Data Augmentation

https://github.com/aleju/imgaug

Label Smoothing

- Training adjusts the model to make outputs as close as possible to the targets/labels.
- So if labels are smoothed a little, overfitting will be reduced.
- For example, if label 0 is mapped to 0.1 and 1 is mapped to 0.9, training will converge early.
- Training procedure will not try as hard as before to output as close as possible to 0 or 1.

Summary

- All data contains noise.
- Given enough power, a neural network will model noise as well.
- Restricting the network's power allows it to model the underlying behaviour of data instead of noise.
- This reduces over-fitting on training data and improves generalisation of the network on unseen data.