Regularization

What is regularization?

- One definition in the book
  - Any modification to the learning that aims to reduce generalization error, but not training error
- The problem
  - With high-capacity models (many layers, many parameters), there is a high risk of overfitting
  - If we just naively reduce the model capacity, we risk underfitting
  - Hitting the “sweet spot” in the middle is difficult
- The solutions
  - Regularization allows you to use high-capacity models
  - Some constraint or clever strategy *regularizes* this model to behave more smoothly
  - We are thus able to trade-off between capacity and robustness

Examples of regularization

- Parameter norm penalties
  - Classical problem in optimization: unstable parameter solutions
- Data augmentation
  - Artificially expand the training dataset
  - Allows the model to see a larger variation of training examples
- Noise injection
  - Add noise to parts of the model
  - Input noise: a bit like dataset augmentation
  - Noise on the units: dropout

Recall maximum likelihood - again

- We set up a model to predict *y* given $\mathbf{x}$:
  $p(y|\mathbf{x})$
- Now, we change notation a bit, realizing that *y* is really also a function of our parameters, which we now call θ:
  $p(y|\mathbf{x}, \theta)$
- In ML estimation, what we really had was a likelihood function of the whole training data sample (*X,Y*):
  $\theta_{\mathrm{ML}} = \arg\max p(Y|X, \theta) = \arg\max \prod_i p(y_i|\mathbf{x}_i, \theta)$

$$\theta_{\mathrm{ML}} = \arg\max p(Y|X,\theta) = \arg\max \prod_i p(y_i|\mathbf{x}_i,\theta)$$

MAP inference

- Now let's think of the problem another way: $p(\theta|X,Y)$
- Whenever there's a conditional, we may think of good ol' Bayes
- If we use Bayes' theorem, this part is equal to (just pretend that *A* is θ and that *B* is *X,Y*):

$$p(\theta|X,Y) = \frac{p(X,Y|\theta)p(\theta)}{p(X,Y)}$$

- Since we are trying to maximize, we can neglect the denominator:

$$p(\theta|X,Y) \propto p(X,Y|\theta)p(\theta)$$

- What about $p(\theta)$?

$$p(\theta|X,Y) \propto p(X,Y|\theta)p(\theta)$$

MAP inference

- If we let $p(\theta)$ = const., we are back to something familiar:

$$p(X,Y|\theta) = \frac{p(X,Y,\theta)}{p(\theta)} = \frac{p(Y|X,\theta)p(X|\theta)p(\theta)}{p(\theta)} = p(Y|X,\theta)p(X|\theta)$$

$$\boxed{\theta_{\mathrm{ML}} = \arg\max p(Y|X,\theta)}$$

- If not, we have what is called a *prior* on our parameters:

$$p(\theta|X,Y) \propto p(X,Y|\theta)p(\theta)$$

  - The left part that models θ is the *posterior* distribution
- The solution to this is called a *MAP estimator*:

$$\theta\mathrm{MAP} = \arg\max p(Y|X,\theta)p(\theta)$$

MAP inference

- Let's look at a known case
  - Gaussian output distribution with a linear regression function: $y \sim \mathcal{N}(f(\mathbf{x}), \sigma^2)$
  - But now, let's also assume a standard Gaussian prior on θ: $\theta \sim \mathcal{N}(0, \mathbf{I})$
- We then get the following:

$$\begin{aligned}
\theta_{\text{MAP}} &= \arg\max p(Y|X, \theta)p(\theta) \\
&= \arg\max \left(\prod_{i=1}^{m} \mathcal{N}(y_i; f(\mathbf{x}_i), \sigma^2) \right) \mathcal{N}(\theta; 0, \mathbf{I}) \\
&= \arg\max \mathcal{N}(\theta; 0, \mathbf{I}) \prod_{i=1}^{m} \mathcal{N}(y_i; f(\mathbf{x}_i), \sigma^2)
\end{aligned}$$

$$\theta_{\text{MAP}} = \arg\max \mathcal{N}(\theta; 0, \mathbf{I}) \prod_{i=1}^{m} \mathcal{N}(y_i; f(\mathbf{x}_i), \sigma^2)$$

MAP inference

- If we plug in the Gaussian pdf and remove constants, we get:

  $$\theta_{\text{MAP}} = \arg\max e^{-\frac{1}{2}(\theta-0)^2} \prod_i e^{-\frac{1}{2}(y_i-\hat{y}_i)^2} \qquad \hat{y} = f(\mathbf{x})$$

- As usual, we take the negative logarithm to arrive at a loss to minimize:

  $$\theta_{\text{MAP}} = \arg\min \left(\frac{1}{2}\|\theta\|^2 + \frac{1}{2}\sum_i (y_i - \hat{y}_i)^2\right)$$

- What if the “likelihood part” (rightmost) is multinoulli?
  - No problem - the likelihood and the prior are separated in log-space:

    $$\theta_{\text{MAP}} = \arg\min \left(\frac{1}{2}\|\theta\|^2 - \sum_i y_i \log \hat{y}_i\right)$$

Weight decay

- All this leads to a modified loss function that takes into account the prior:
  $L(y, \hat{y}; \theta) = L(y, \hat{y}) + \alpha\Omega(\theta)$
- When using a Gaussian prior:
  $\Omega(\theta) = \frac{1}{2}\|\theta\|^2$
- we also call it
  - L_2 regularization/penalty
  - Tikhonov regularization
  - Ridge regression (mostly reserved for regression models)
- In deep learning: *weight decay*

New hyperparameter

$$L(y, \hat{y}; \theta) = L(y, \hat{y}) + \alpha\Omega(\theta)$$

Norm penalties

- Another common prior is the *Laplacian*:
  $\theta \sim \mathrm{Laplace}\,(0, 1) = \frac{1}{2} e^{-|\theta|}$
- which leads to the L_1 penalty:
  $\Omega(\theta) = \|\theta\|_1$

$$L(y, \hat{y}; \theta) = L(y, \hat{y}) + \alpha\Omega(\theta)$$

Norm penalties

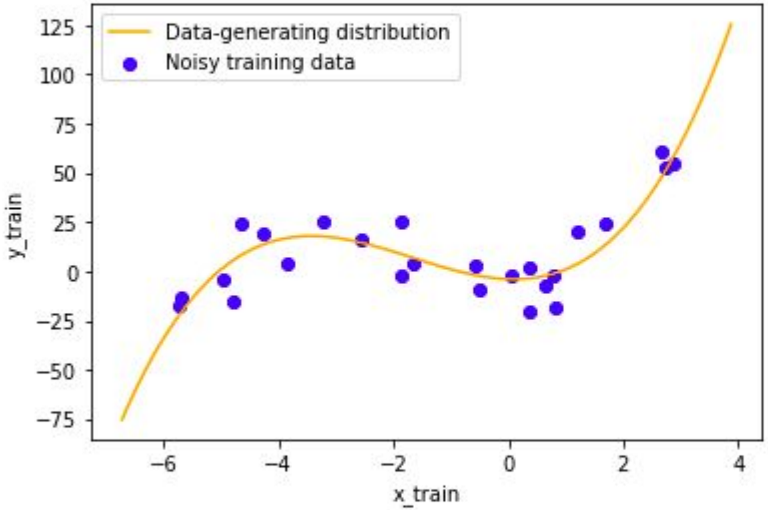
- Computing gradients is often very easy
- The first loss term we already solved with backprop
- The regularization term has a simple solution for L_1 and L_2
  - L_1: $\nabla_\theta \Omega = \frac{\theta}{|\theta|} = \operatorname{sign}(\theta)$
  - L_2: $\nabla_\theta \Omega = \theta$

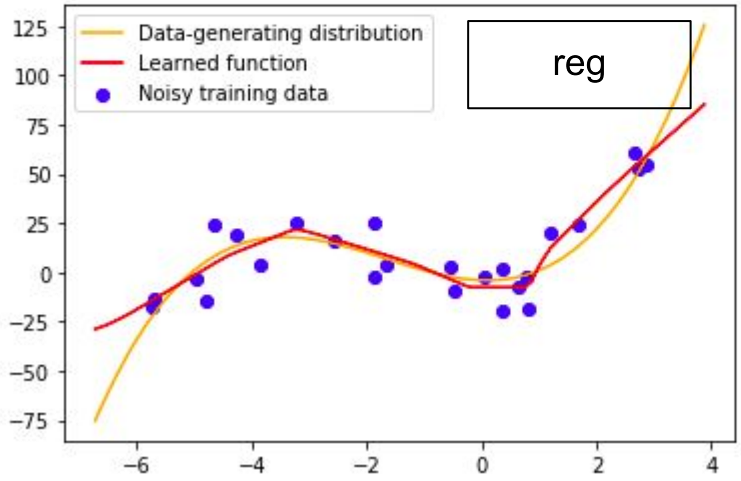
Weight decay

- Let's look at the L_2 case
  $L(y, \hat{y}; \theta) = L(y, \hat{y}) + \frac{1}{2}\alpha\|\theta\|^2$
- with the gradient
  $\nabla_\theta L(y, \hat{y}) + \alpha\theta$
- When we take small steps along the negative:
  $\theta \leftarrow \theta - \epsilon\left(\nabla_\theta L(y, \hat{y}) + \alpha\theta\right)$
- you should be able to see how this "shrinks" or "decays" the weights

Weight decay

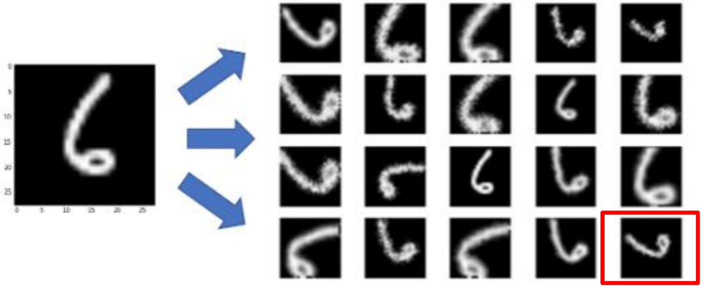
- Simple example





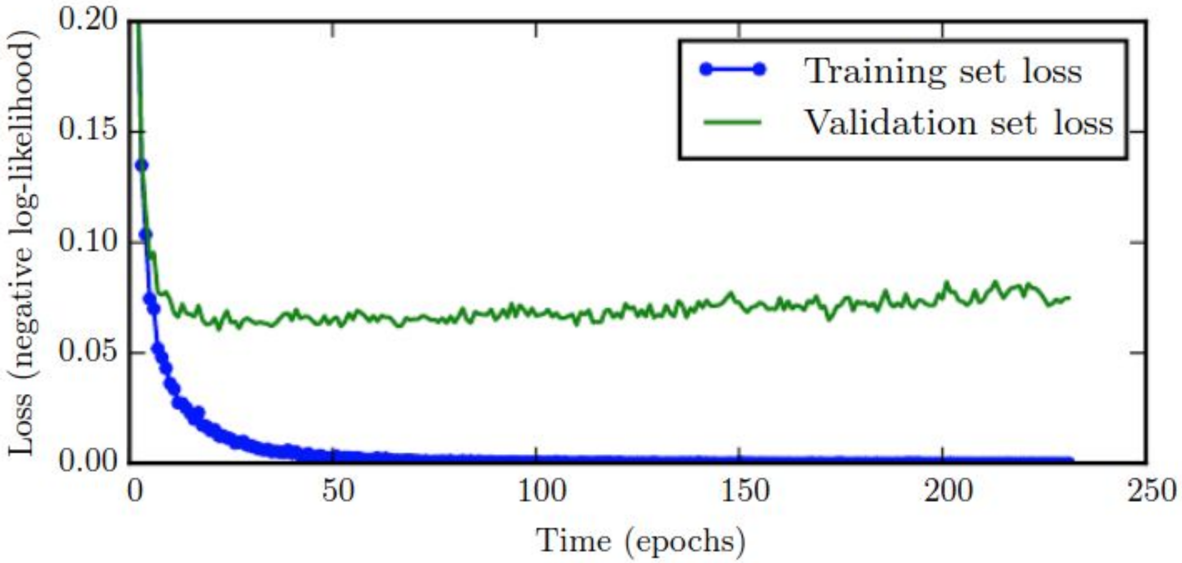
Augmentation

- When training data is scarce, we can artificially expand the training set
- Typical strategies
  - Noise
    - Additive, e.g. Gaussian
    - Multiplicative, e.g. Bernoulli (Dropout)
  - Affine transformation
    - Translations
    - Rotations
    - Scaling
    - Shearing
  - Truncation
    - Cropping
  - Non-linear operations
    - Brightness



https://hazyresearch.github.io/snorkel/blog/tanda.html

Early stopping



Early stopping

Number of updates per epoch

Best validation performance so far

New best performance achieved

Reset patience

Update parameters and performance

Algorithm 7.1 The early stopping meta-algorithm for determining the best amount of time to train. This meta-algorithm is a general strategy that works well with a variety of training algorithms and ways of quantifying error on the validation set.

Let n be the number of steps between evaluations.
Let p be the "patience," the number of times to observe worsening validation set error before giving up.
Let $\boldsymbol{\theta}_o$ be the initial parameters.
$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta}_o$
$i \leftarrow 0$
$j \leftarrow 0$
$v \leftarrow \infty$
$\boldsymbol{\theta}^* \leftarrow \boldsymbol{\theta}$
$i^* \leftarrow i$
while $j < p$ **do**
  Update $\boldsymbol{\theta}$ by running the training algorithm for n steps.
  $i \leftarrow i + n$
  $v' \leftarrow \text{ValidationSetError}(\boldsymbol{\theta})$
  if $v' < v$ **then**
    $j \leftarrow 0$
    $\boldsymbol{\theta}^* \leftarrow \boldsymbol{\theta}$
    $i^* \leftarrow i$
    $v \leftarrow v'$
  else
    $j \leftarrow j + 1$
  end if
end while
Best parameters are $\boldsymbol{\theta}^*$, best number of training steps is i^*.

Dropout

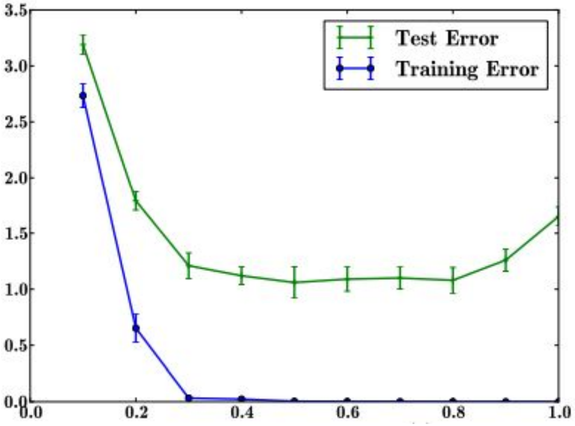
- A method for regularizing the network during training
- First the method:
  - During forward pass, randomly set neurons to zero with a probability $p < 1$
  - This “weakens” the network so that it only has a “power” of $1 - p$
  - To compensate, multiply all other neurons by $1 / (1 - p)$
  - The backward pass is trivial: some responses are just zero and provide no gradient
- Common values for p
  - Input units: between 0 and 0.2
  - Hidden units: 0.5
  - Output units (e.g. linear, softmax): **always 0!**

Dropout

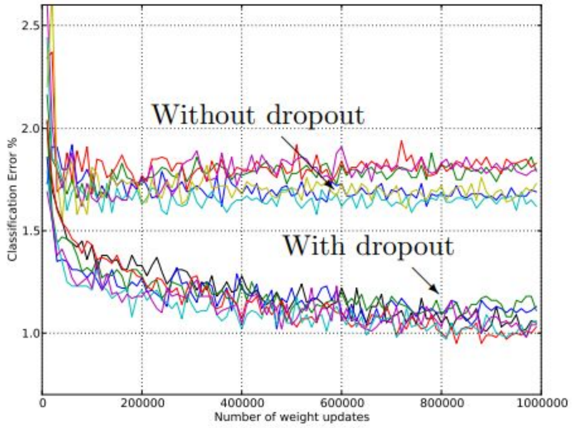
- Slightly more formal:
  - Multiply each neuron by a Bernoulli random variable d ~ Bernoulli(p)
  - The rescaling of 1 / (1 - p) ensures that the net input to any neuron stays the same
  - Each pass through a layer looks like this (no rescaling):
    $$\mathbf{h}_i = \phi(\mathbf{W}_i \cdot (\mathbf{h}_{i-1} \odot \mathbf{d}_{i-1}) + \mathbf{b}_i)$$
  - And with rescaling:
    $$\mathbf{h}_i = \phi\left(\mathbf{W}_i \cdot \left(\mathbf{h}_{i-1} \odot \frac{\mathbf{d}_{i-1}}{1-p}\right) + \mathbf{b}_i\right)$$

Dropout performance

- Paper: Srivastava, Nitish, et al. "Dropout: a simple way to prevent neural networks from overfitting." *JMLR*’14.

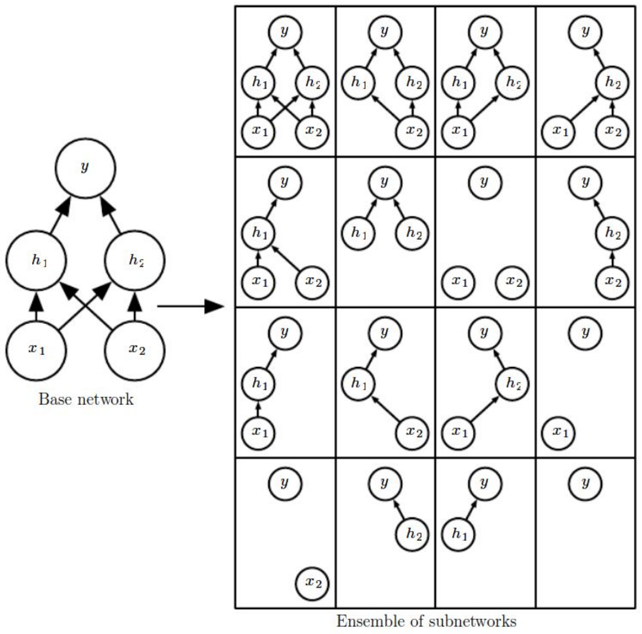


1-*p* (keep rate)



Dropout subnet interpretation

- Each time you drop a set of units, you effectively create a new “subnetwork”
- These subnetworks obviously share the same weights
- As such, repeated training like this, effectively corresponds to a variant of *bagging*
    - At each minibatch, dropout samples a new model
    - This model sees a small, random subset of the data
    - In the end, this “ensemble” of models is combined



Exercise

- You are given only 1000 MNIST training examples
- The baseline model overfits (100% train acc, ~89% valid acc)
- Can you improve this with regularization?

Bonus:

- What performance can you achieve using the whole dataset?