Preventing overfitting

- Approach 1: Get more data!
  - Always best if possible!
  - If no natural ones, use data augmentation
- Approach 2: Use a model that has the right capacity:
  - enough to fit the true regularities.
  - not enough to also fit spurious regularities (if they are weaker).
  - Parameter tuning
- Approach 3: Average many different models.
  - Models with different forms.
  - Train on different subsets
- Approach 4: Use specific regularizing structures

Regularization: Preventing Overfitting

- To **prevent overfitting**, a large number of different methods have been developed.
  - Data Augmentation (talked about)
  - Weight-sharing structures (talked about, e.g. CNN, RNN)
  - Weight-decay (talked about)
  - Early stopping (talked about)
  - **Model averaging**
  - **Dropout**
  - **Batch normalization**
  - **Weight regularization structures**
  - Bayesian fitting of neural nets
  - Generative pre-training (will talk later)
  - Sparsity in hidden units (will talk later)

Making models differ by changing their training data

- Bagging: Train different models on different subsets of the data.
  - Sample data with replacement
    a,b,c,d,e $\rightarrow$ a c c d b
  - Random forests use lots of different decision trees trained using bagging. They work well.
- We could use bagging with neural nets.

- Boosting: Train a sequence of low capacity models. Weight the training cases differently for each model in the sequence.
  - Boosting up-weights cases that previous models got wrong.
  - An early use of boosting was with neural nets for MNIST.
  - It focused the computational resources on modeling the tricky cases.

Bagging in Deep Neural Networks

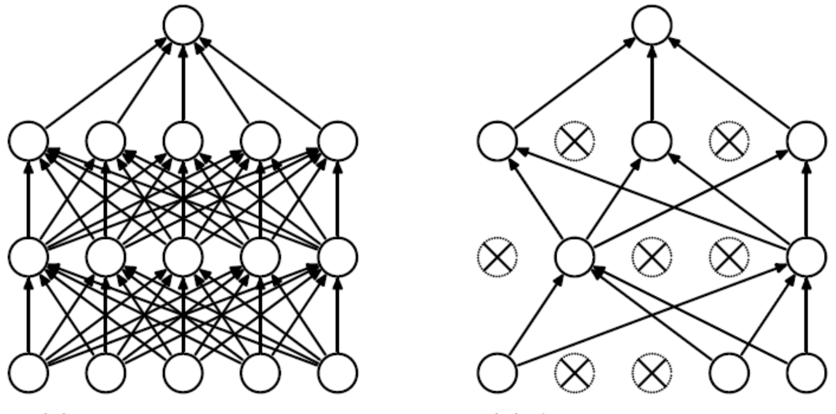
- Deep networks are inherent local optimization algorithms
- Different starting points give very different result networks!
- Directly averaging networks with different initializations
  - No bootstrapping!

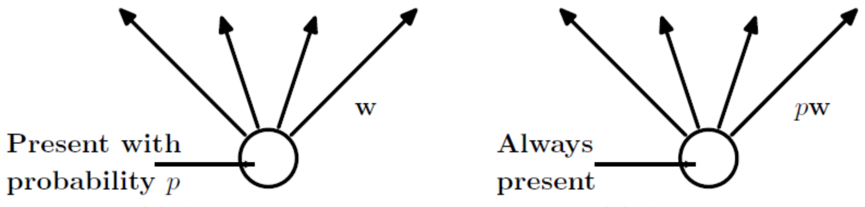
Multiple examples from one test data: Test time Cropping

- e.g. Resize the image into different sizes/aspect ratios, crop squares at different places of the image
  - Similar to object proposals, but squared
  - Reduce the error significantly with 144/150 crops (proposals)

Dropout: An efficient way to average many large neural nets

- Consider a neural net with one hidden layer.
- Each time we present a training example, we randomly omit each hidden unit with probability 0.5.
- So we are randomly sampling from 2^H different architectures.
  - All architectures share weights.


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



(a) At training time (b) At test time

Dropout as a form of model averaging

- We sample from 2^H models. So only a few of the models ever get trained, and they only get one training example.

- The sharing of the weights means that every model is very strongly regularized.
    - It's a much better regularizer than L2 or L1 penalties that pull the weights towards zero.

But what do we do at test time?

- We could sample many different architectures and take the geometric mean of their output distributions.
- It better to use all of the hidden units, but to halve their outgoing weights.
  - This exactly computes the geometric mean of the predictions of all 2^H models.

What if we have more hidden layers?

- Use dropout of 0.5 in every layer.
- At test time, use the “mean net” that has all the outgoing weights halved.
    - This is not exactly the same as averaging all the separate dropped out models, but it’s a pretty good approximation, and its fast.
- Alternatively, run the stochastic model several times on the same input.
    - This gives us an idea of the uncertainty in the answer.

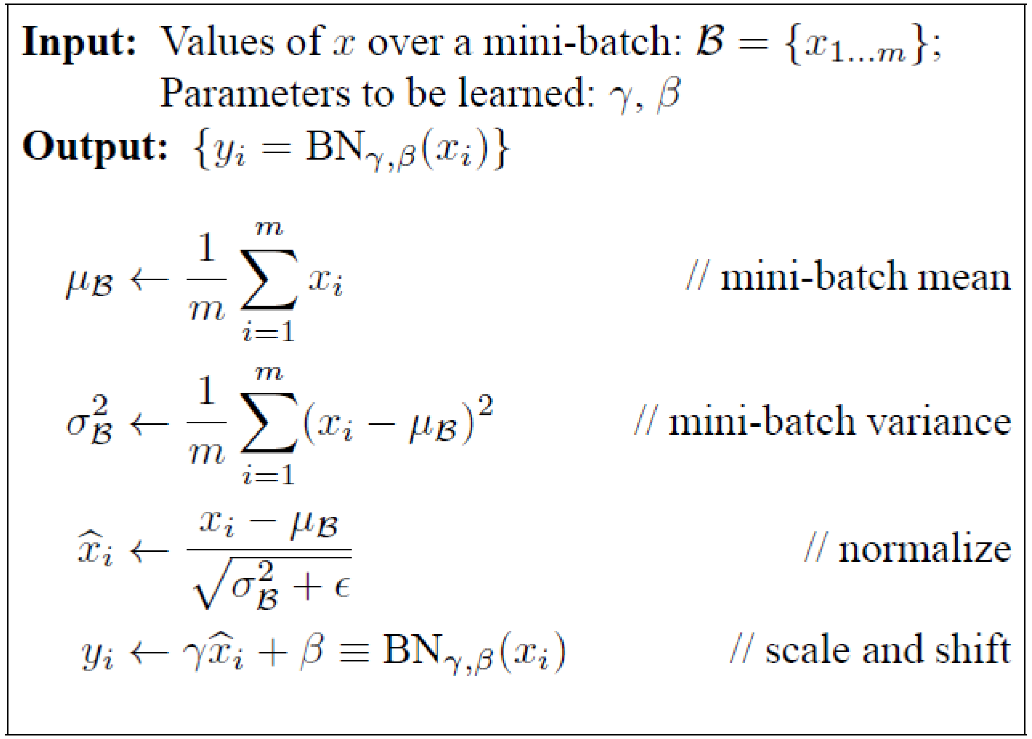
What about the input layer?

- It may help to use dropout there too, but with a higher probability of keeping an input unit.
  - Averaging out the noise in the input if it’s noisy (don’t use it if it’s not noisy)
  - This trick is already used by the “denoising autoencoders” developed by Pascal Vincent, Hugo Larochelle and Yoshua Bengio.

Batch Normalization Layer

- This is done for each hidden dimension separately

- How many parameters?
- Gradient w.r.t. parameters?
- Gradient w.r.t. input?



```
Input:  Values of $x$ over a mini-batch: $\mathcal{B} = \{x_{1\ldots m}\}$;
        Parameters to be learned: $\gamma$, $\beta$
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

    $\mu_\mathcal{B} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i$                              // mini-batch mean
    $\sigma_\mathcal{B}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_\mathcal{B})^2$    // mini-batch variance
    $\widehat{x}_i \leftarrow \frac{x_i - \mu_\mathcal{B}}{\sqrt{\sigma_\mathcal{B}^2 + \epsilon}}$   // normalize
    $y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i)$      // scale and shift
```

Other stuff

$$y^{(k)} = \gamma^{(k)} \widehat{x}^{(k)} + \beta^{(k)}.$$

- If $\gamma = 0$, equiv. to dropout
- No additional bias term needed in the conventional network (BN provides the bias term)