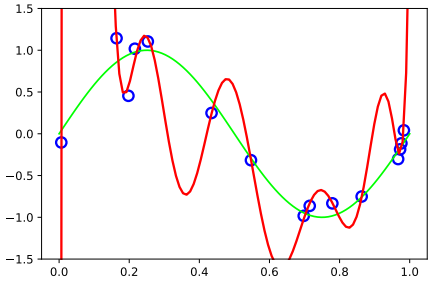
Aalto University

CS-E4890: Deep Learning
Regularization

Alexander Ilin

Overfitting

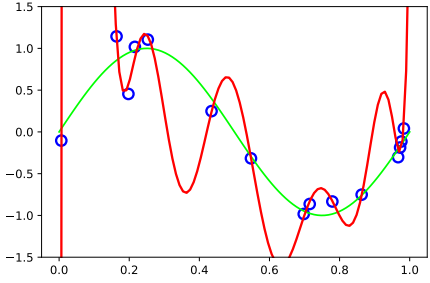
- Good performance on training data but bad performance on new, test data (poor generalization).



Regression with a polynomial function of order $M = 12$
green: correct model, red: fitted model

Why overfitting happens?

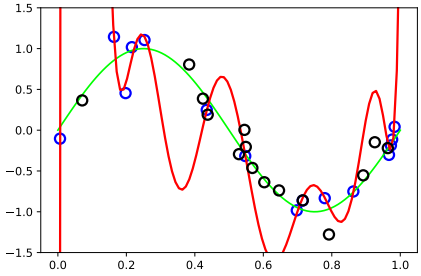
- Conventional wisdom: The model is too flexible for the amount of training data.
- Wikipedia: An overfitted model is a statistical model that contains more parameters than can be justified by the data.
- Rule of thumb (one in ten rule) for logistic regression: To keep the risk of overfitting low, the number of examples should be ten times larger than the number of parameters.



Regression with a polynomial function of order $M = 12$

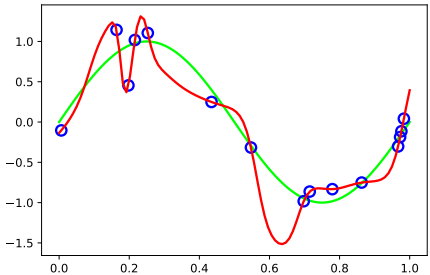
How to detect overfitting?

- Use validation set (black dots) to evaluate the performance.



Regularization

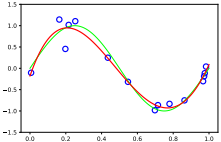
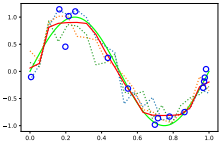
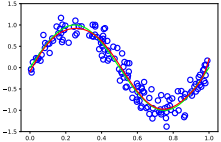
- **Regularization** is a technique used to solve the overfitting problem.
- Neural networks are very powerful (universal approximators), they can model very large and complex datasets.



Regression with an MLP network
green: correct model, red: fitted model

Regularization methods

1. Limit model capacity:
   - Reduce network size
   - Weight decay
   - Parameter sharing
2. Early stopping
3. Ensemble methods:
   - Dropout
   - Probabilistic treatment (e.g. Bayesian neural networks)
4. Data augmentation:
   - Noise injection
   - Transformations
   - Adversarial training

1. Limit model capacity

Reduce size of network

- Recall the conventional wisdom: Overfitting is likely to happen when a model contains more parameters than can be justified by the data.
- Solution: Reduce the number of parameters.
- We can vary the number of neurons/the number of layers to find the architecture that works best (on the validation data).
- Advantage: Conceptually easy.
- Disadvantage: Other regularization methods often give better accuracy.

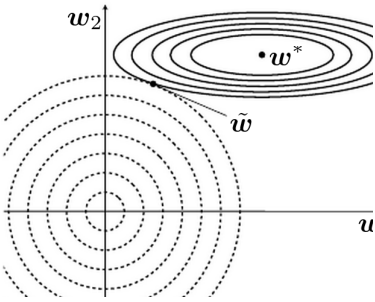
L_2 regularization (Tikhonov, 1943)

- Add a penalty term $\Omega(\mathbf{w})$ to the training cost:

$$\mathcal{L}_{\text{reg}} = \mathcal{L} + \Omega(\mathbf{w}) .$$
$$\Omega(\mathbf{w}) = \frac{\alpha}{2} \|\mathbf{w}\|^2 = \frac{\alpha}{2} \sum_i w_i^2$$

  The penalty term is a function of parameters $\mathbf{w}$, not data.

- L_2 regularization pushes the solution towards zero.



w^* unregularized solution
$\tilde{w}$ regularized solution

- L_2 regularization is often called *weight decay*. For example, `torch.optim.Adam(params, lr, betas, eps, weight_decay)` implements L_2 regularization with the hyperparameter α set by parameter `weight_decay`.

L_2 regularization vs weight decay

- Using term *weight decay* for L_2-regularization may cause confusion.
- Weight decay as described by Hanson and Pratt (1988):

$$\mathbf{w}_{t+1} = (1 - \lambda)\mathbf{w}_t - \eta \nabla \mathcal{L} \tag{1}$$

- For standard stochastic gradient descent, weight decay is equivalent to L_2 regularization:

$$\mathcal{L}_{\text{reg}} = \mathcal{L} + \frac{\alpha}{2} \|\mathbf{w}\|^2$$

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta \nabla \mathcal{L}_{\text{reg}} = \mathbf{w}_t - \eta \nabla \mathcal{L} - \eta \alpha \mathbf{w}_t = (1 - \eta\alpha)\mathbf{w}_t - \eta \nabla \mathcal{L}$$

- For algorithms like Adam, weight decay as given in (1) is not equivalent to L_2 regularization.
- Loshchilov and Hutter (2017) proposed a regularized version of Adam which tries to follow the early-days definition of weight decay. It is available in PyTorch as `torch.optim.AdamW`.

Why L_2 regularization reduces overfitting

- Intuition: Smaller weights usually produce smoother functions (smaller magnitudes of derivatives).
- Consider a linear regression problem (no bias term for simplicity):

$$\mathcal{L}(\mathbf{w}) = \frac{1}{2N}\sum_{n=1}^{N}\left(y_n - \mathbf{w}^\top \mathbf{x}_n\right)^2 + \frac{\alpha}{2}\mathbf{w}^\top\mathbf{w}$$

- Let us find the minimum by computing the gradient and equating it to zero:

$$\begin{aligned}\nabla_{\mathbf{w}}\mathcal{L} &= \frac{2}{2N}\sum_{n=1}^{N}\left(y_n - \mathbf{w}^\top\mathbf{x}_n\right)(-\mathbf{x}_n) + \frac{\alpha}{2}\mathbf{w} = \frac{1}{N}\sum_{n=1}^{N}\mathbf{x}_n\mathbf{x}_n^\top\mathbf{w} - \frac{1}{N}\sum_{n=1}^{N}y_n\mathbf{x}_n + \frac{\alpha}{2}\mathbf{w} \\ &= \left(\frac{1}{N}\sum_{n=1}^{N}\mathbf{x}_n\mathbf{x}_n^\top + \alpha\mathbf{I}\right)\mathbf{w} - \frac{1}{N}\sum_{n=1}^{N}y_n\mathbf{x}_n = 0\end{aligned}$$

which yeilds

$$\mathbf{w} = \left(\frac{1}{N}\sum_{n=1}^{N}\mathbf{x}_n\mathbf{x}_n^\top + \alpha\mathbf{I}\right)^{-1}\left(\frac{1}{N}\sum_{n=1}^{N}y_n\mathbf{x}_n\right)$$

Why weight decay reduces overfitting

- The solution of linear regression with weight decay:

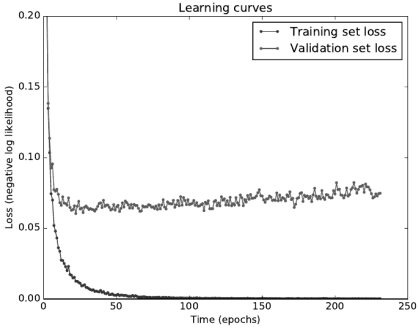
$$\mathbf{w} = \left(\frac{1}{N} \sum_{n=1}^{N} \mathbf{x}_n \mathbf{x}_n^\top + \alpha \mathbf{I} \right)^{-1} \left(\frac{1}{N} \sum_{n=1}^{N} y_n \mathbf{x}_n \right)$$

- L_2 regularization causes the learning algorithm to "perceive" the input as having higher variance. This makes the weights shrink.
- The regularization effect is larger for the weight values determined by the minor (opposite to principal) components of the data.

2. Early stopping

Early stopping

- Monitor validation performance during training.
- Stop when it starts to deteriorate (with other regularization techniques it might never start).
- Keeps solution close to the initialization.



Why early stopping reduces overfitting

- Consider a liner regression problem with a linear model (no bias term). The loss is a quadratic function with the minimum at $\mathbf{w}_*$:

$$\mathcal{L}(\mathbf{w}) = \mathcal{L}(\mathbf{w}_*) + \frac{1}{2}(\mathbf{w} - \mathbf{w}_*)^\top \mathbf{H}(\mathbf{w} - \mathbf{w}_*)$$

- The update with gradient descent:

$$\mathbf{w}_t = \mathbf{w}_{t-1} - \epsilon\mathbf{H}(\mathbf{w}_{t-1} - \mathbf{w}_*)$$

$$\mathbf{w}_t - \mathbf{w}_* = \mathbf{w}_{t-1} - \mathbf{w}_* - \epsilon\mathbf{H}(\mathbf{w}_{t-1} - \mathbf{w}_*) = (\mathbf{I} - \epsilon\mathbf{H})(\mathbf{w}_t - \mathbf{w}_*)$$

- Using the eigendecomposition $\mathbf{H} = \mathbf{Q}\mathbf{\Lambda}\mathbf{Q}^\top$ gives

$$\mathbf{Q}^\top(\mathbf{w}_t - \mathbf{w}_*) = \mathbf{Q}^\top(\mathbf{Q}\mathbf{Q}^\top - \epsilon\mathbf{Q}\mathbf{\Lambda}\mathbf{Q}^\top)(\mathbf{w}_t - \mathbf{w}_*) = (\mathbf{I} - \epsilon\mathbf{\Lambda})\underbrace{\mathbf{Q}^\top(\mathbf{w}_{t-1} - \mathbf{w}_*)}_{=(\mathbf{I}-\epsilon\mathbf{\Lambda})\mathbf{Q}^\top(\mathbf{w}_{t-2}-\mathbf{w}_*)}$$

$$= (\mathbf{I} - \epsilon\mathbf{\Lambda})^2\mathbf{Q}^\top(\mathbf{w}_{t-2} - \mathbf{w}_*) = (\mathbf{I} - \epsilon\mathbf{\Lambda})^t\mathbf{Q}^\top(\mathbf{w}_0 - \mathbf{w}_*)$$

- Assuming $\mathbf{w}_0 = 0$, this yields

$$\mathbf{Q}^\top\mathbf{w}_t = \mathbf{Q}^\top\mathbf{w}_* - (\mathbf{I} - \epsilon\mathbf{\Lambda})^t\mathbf{Q}^\top\mathbf{w}_* = [\mathbf{I} - (\mathbf{I} - \epsilon\mathbf{\Lambda})^t]\mathbf{Q}^\top\mathbf{w}_*$$

Optimal solution with weight decay

- Now consider minimizing the same loss with a weight decay penalty:

$$\mathcal{L}_\alpha(\mathbf{w}) = \mathcal{L}(\mathbf{w}_*) + \frac{1}{2}(\mathbf{w} - \mathbf{w}_*)^\top \mathbf{H}(\mathbf{w} - \mathbf{w}_*) + \frac{\alpha}{2}\mathbf{w}^\top \mathbf{w}$$

The optimal weights $\tilde{\mathbf{w}}$ can be found by equating the gradient to zero:

$$\nabla \mathcal{L}_\alpha = \mathbf{H}(\mathbf{w} - \mathbf{w}_*) + \alpha \mathbf{w} = 0$$
$$\tilde{\mathbf{w}} = (\mathbf{H} + \alpha \mathbf{I})^{-1} \mathbf{H} \mathbf{w}_*$$

In the rotated coordinate system, the solution is given by

$$\begin{aligned}
\mathbf{Q}^\top \tilde{\mathbf{w}} &= \mathbf{Q}^\top (\mathbf{Q}\mathbf{\Lambda}\mathbf{Q}^\top + \alpha \mathbf{I})^{-1} \mathbf{Q}\mathbf{\Lambda}\mathbf{Q}^\top \mathbf{w}_* \\
&= \mathbf{Q}^\top \left[\mathbf{Q}(\mathbf{\Lambda} + \alpha \mathbf{I})\mathbf{Q}^\top\right]^{-1} \mathbf{Q}\mathbf{\Lambda}\mathbf{Q}\mathbf{w}_* \\
&= (\mathbf{\Lambda} + \alpha \mathbf{I})^{-1} \mathbf{\Lambda}\mathbf{Q}^\top \mathbf{w}_*
\end{aligned}$$

Why early stopping reduces overfitting

- If we use L_2 regularization:

$$\mathbf{Q}^\top \tilde{\mathbf{w}} = (\mathbf{\Lambda} + \alpha \mathbf{I})^{-1} \mathbf{\Lambda} \mathbf{Q}^\top \mathbf{w}_*$$

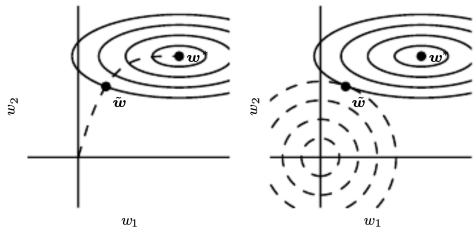
- If we use early stopping after iteration t:

$$\mathbf{Q}^\top \mathbf{w}_t = [\mathbf{I} - (\mathbf{I} - \epsilon \mathbf{\Lambda})^t] \mathbf{Q}^\top \mathbf{w}_*$$

- If the hyperparameters ϵ, α and t are chosen such that

$$(\mathbf{\Lambda} + \alpha \mathbf{I})^{-1} = [\mathbf{I} - (\mathbf{I} - \epsilon \mathbf{\Lambda})^t]$$

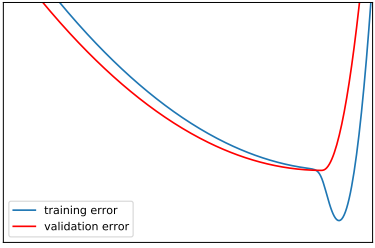
then L_2 regularization and early stopping can be seen as equivalent.



w^* unregularized solution
$\tilde{w}$ regularized solution

Early stopping

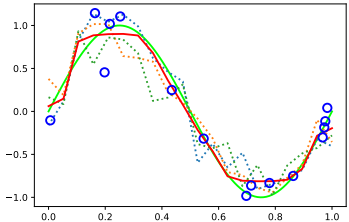
- Early stopping stops training before we go to a narrow hole in which the model may generalize poorly.



3. Ensemble methods

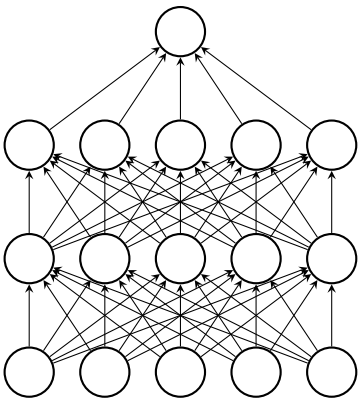
Ensemble methods

- Train several models and take average of their outputs.
- Also known as *bagging* or *model averaging*.
- It helps to make individual models different by
  - varying models or algorithms
  - varying hyperparameters
  - varying data (dropping examples or dimensions)
  - varying random seed



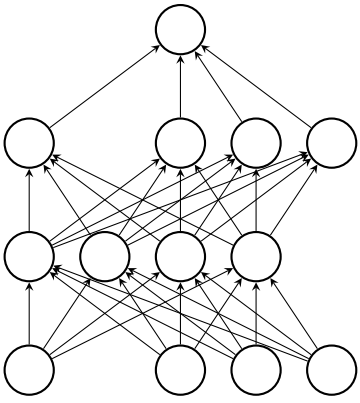
Dropout (Hinton et al., 2012)

- At training time: For each data example $\mathbf{x}$ (or mini-batch), randomly delete each hidden node with probability p.
- Can be seen as
  - injecting (multiplicative binary) noise
  - training an *ensemble* of models with shared weights.
- For a network with N neurons, our ensemble contains 2^N models.



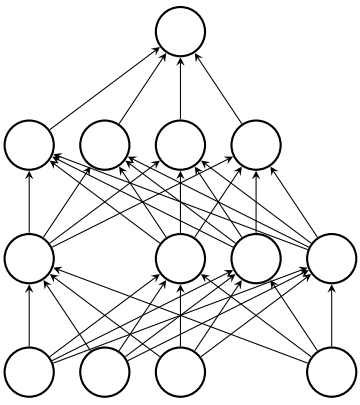
Dropout (Hinton et al., 2012)

- At training time: For each data example $\mathbf{x}$ (or mini-batch), randomly delete each hidden node with probability p.
- Can be seen as
  - injecting (multiplicative binary) noise
  - training an *ensemble* of models with shared weights.
- For a network with N neurons, our ensemble contains 2^N models.



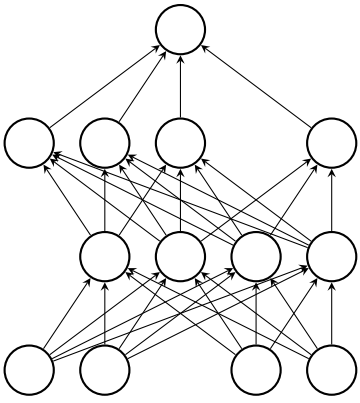
Dropout (Hinton et al., 2012)

- At training time: For each data example $\mathbf{x}$ (or mini-batch), randomly delete each hidden node with probability p.
- Can be seen as
  - injecting (multiplicative binary) noise
  - training an *ensemble* of models with shared weights.
- For a network with N neurons, our ensemble contains 2^N models.

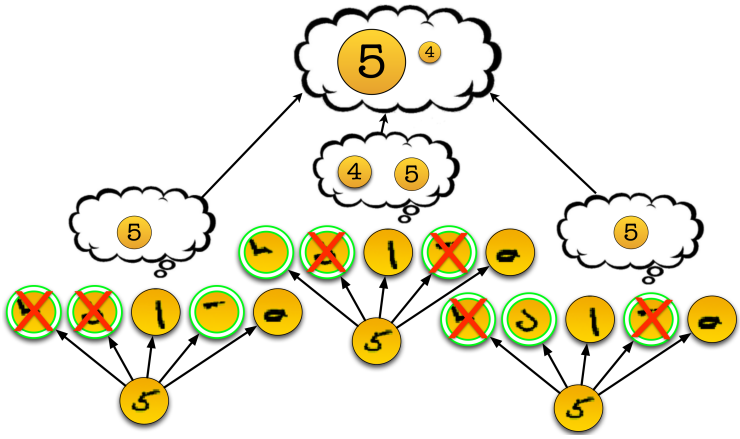


Dropout (Hinton et al., 2012)

- At training time: For each data example $\mathbf{x}$ (or mini-batch), randomly delete each hidden node with probability p.
- Can be seen as
  - injecting (multiplicative binary) noise
  - training an *ensemble* of models with shared weights.
- For a network with N neurons, our ensemble contains 2^N models.



Dropout as bagging



Dropout: Training and evaluation modes

- At test time, neurons are not dropped. Therefore, we need to adjust the activations to take into account the difference between training and evaluation modes.
- If a signal x is dropped with probability p, the expected value after the dropout is

$$E[x] = (1 - p)x$$

- PyTorch `nn.Dropout` layer:
  - Training mode: zero signals with probability p and scaled by factor $\frac{1}{1-p}$.
  - Evaluation mode: do nothing.

```
model = nn.Sequential(
    nn.Linear(1, 100),
    nn.Tanh(),
    nn.Dropout(0.02),
    ...
)

# Switch to training mode
model.train()
# training the model
...
# Switch to evaluation mode
model.eval()
# test the model
```

Probabilistic treatment: Bayesian neural networks

- Bayesian neural networks were proposed in late 80s, popularized by David MacKay (1992).
- Bayesian methodology: one should combine predictions $p(\mathbf{y} \mid \mathbf{x}, \mathcal{M}_i)$ given by all possible models:

$$p(\mathbf{y} \mid \mathbf{x}, D) = \sum_i p(\mathbf{y} \mid \mathbf{x}, \mathcal{M}_i) p(\mathcal{M}_i \mid D)$$

  weighting them by model evidence $p(\mathcal{M}_i \mid D)$.

Probabilistic treatment: Bayesian neural networks

- If we fix the architecture of a neural network, the set of possible models is defined by all possible parameter values $\mathbf{w}$:

$$p(\mathbf{y} \mid \mathbf{x}, D) = \int p(\mathbf{y} \mid \mathbf{x}, \mathbf{w}) p(\mathbf{w} \mid D) d\mathbf{w}$$

- We then need to evaluate the posterior distribution $p(\mathbf{w} \mid D)$ of the model parameters given the training data. We do that using Bayes rule:

$$p(\mathbf{w} \mid D) = \frac{p(D|\mathbf{w}) p(\mathbf{w})}{p(D)}$$

- We can use different strategies to approximate $p(\mathbf{w} \mid D)$:
  - maximum a posteriori estimation (point estimates of $\mathbf{w}$)
  - variational approximation of $p(\mathbf{w} \mid D)$
  - draw samples from $p(\mathbf{w} \mid D)$

Weight decay as Bayesian prior

- To find maximum a a posteriori estimate, we can maximize the logarithm of the posterior:

$$\mathcal{F}(\mathbf{w}) = \log p(\mathbf{w}|D) = \log p(D|\mathbf{w}) + \log p(\mathbf{w}) - \log p(D)$$

  or minimize the following loss function:

$$\mathcal{L}(\mathbf{w}) = -\log p(D|\mathbf{w}) - \log p(\mathbf{w})$$

- Recall that, for example, MSE can be viewed as $-\log p(D|\mathbf{w})$ for a Gaussian model:

$$\frac{1}{N}\sum_{n=1}^{N}\left\|\mathbf{y}^{(n)} - \mathbf{f}(\mathbf{x}^{(n)}, \mathbf{w})\right\|^2 = -\beta \log \prod_{n=1}^{N} \mathcal{N}(\mathbf{y}^{(n)} \mid \mathbf{f}(\mathbf{x}^{(n)}, \mathbf{w}), \sigma^2\mathbf{I}) + \text{const}$$

- If we assume Gaussian prior $p(\mathbf{w}) = \mathcal{N}(\mathbf{0}, \alpha^{-1}\mathbf{I})$ we get:

$$-\log p(\mathbf{w}) = -\log \exp\left(-\frac{\alpha}{2}\|\mathbf{w}\|^2\right) + \text{const} = \frac{\alpha}{2}\|\mathbf{w}\|^2 + \text{const}$$

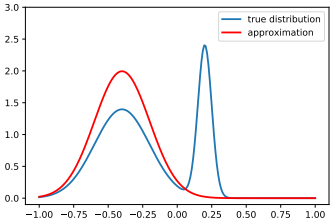
- Thus, L_2 regularization is equivalent to maximum a posteriori estimation in a probabilistic model with Gaussian prior (not really an ensemble method).

Variational approximation of the posterior distribution

- Variational approximations: Within a selected family of distributions $q(\mathbf{w})$, for example, Gaussian:

$$q(\mathbf{w} \mid \boldsymbol{\theta}) = \prod_i \mathcal{N}(w_i \mid \mu_i, \sigma_i^2),$$

  find the one that is closest to the true posterior distribution $p(\mathbf{w} \mid D)$.



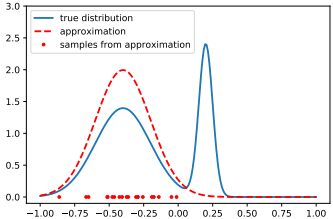
- $\boldsymbol{\theta} = \{\mu_i, \sigma_i\}$ are called variational parameters, we need to tune them.
- Blundell et al. (2015) tune $\boldsymbol{\theta} = \{\mu_i, \sigma_i\}$ by minimizing the Kullback-Leibler (KL) divergence between $q(\mathbf{w} \mid \boldsymbol{\theta})$ and the true posterior distribution:

$$\mathcal{L}(\boldsymbol{\theta}) = \mathrm{KL}[q(\mathbf{w} \mid \boldsymbol{\theta}) || p(\mathbf{w} \mid D)] = \int q(\mathbf{w} \mid \boldsymbol{\theta}) \log \frac{q(\mathbf{w} \mid \boldsymbol{\theta})}{p(\mathbf{w}) p(D \mid \mathbf{w})} dw$$

$$= \underbrace{\mathrm{KL}[q(\mathbf{w} \mid \boldsymbol{\theta}) || p(\mathbf{w})]}_{\text{regularization term}} - \underbrace{E_{q(\mathbf{w}|\boldsymbol{\theta})}[\log p(D \mid \mathbf{w})]}_{\text{fit to data}}$$

Sampling approach: Stein variational gradient descent

- Liu and Wang (2016) find samples $\mathbf{w}_k$ from the posterior approximation $q(\mathbf{w})$ that minimizes the KL divergence with the true posterior.
  - Each sample defines one neural network.
  - We create an ensemble of neural networks.
- We do not postulate the form of $q(\mathbf{w})$ explicitly.



- We still can compute the gradient of the KL divergence, which yields the update rule:

$$\mathbf{w}_k \leftarrow \mathbf{w}_k + \eta \frac{1}{K} \sum_{k'=1}^{K} \underbrace{k(\mathbf{w}_{k'}, \mathbf{w}_k) \nabla_{\mathbf{w}_{k'}} \left[\log p(\mathbf{w}_{k'}) + \log p(D \mid \mathbf{w}_{k'})\right]}_{\text{smoothed gradient}} + \underbrace{\nabla_{\mathbf{w}_j} k(\mathbf{w}_{k'}, \mathbf{w}_k)}_{\text{repulsive force}}$$

  $k(\mathbf{w}, \mathbf{w}')$ is some kernel, for example, $k(\mathbf{w}, \mathbf{w}') = \exp(-\frac{1}{h} \|\mathbf{w} - \mathbf{w}'\|^2)$.
- The repulsive force term prevents all $\mathbf{w}_k$ to collapse into the same values.

Bayesian neural networks

- By using an ensemble of models, Bayesian neural networks can reduce overfitting.
- BNNs can produces confidence intervals for their predictions.
- BNNs can miss some of the modes in the posterior distribution over the weights, thus the uncertainties can be easily underestimated.

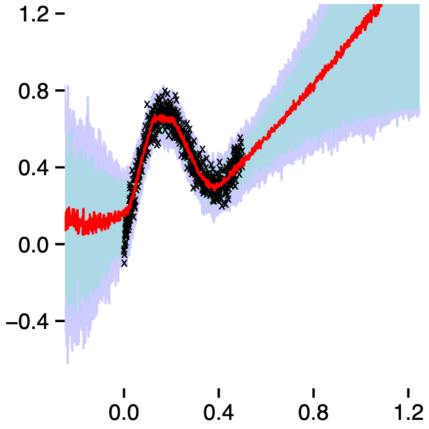


image from (Blundell et al., 2015)

4. Data augmentation

Injecting noise (Sietsma and Dow, 1991)

- Inject random noise during training (different noise instance in each epoch).
- Can be applied to input data, to hidden activations, or to weights.
- Can be seen as data augmentation.
- Simple end effective.

Image transformations

- In some domains it is easy to generate more labeled data by transformations.
- Transformation of images: random crop, translation, scaling, flip, rotation.
- The classification network learns to be invariant to such transformations.

Image from (Dosovitskiy et al., 2014)

mixup (Zhang et al., 2017)

- *mixup* constructs virtual training examples $\tilde{\mathbf{x}}$, $\tilde{\mathbf{y}}$ in the following way:

$$\tilde{\mathbf{x}} = \lambda \mathbf{x}_i + (1 - \lambda)\mathbf{x}_j$$
$$\tilde{\mathbf{y}} = \lambda \mathbf{y}_i + (1 - \lambda)\mathbf{y}_j$$

  where $\mathbf{x}_i$, $\mathbf{x}_j$ are raw input vectors and $\mathbf{y}_i$, $\mathbf{y}_j$ are one-hot label encodings. $(\mathbf{x}_i, \mathbf{y}_i)$ and $(\mathbf{x}_j, \mathbf{y}_j)$ are two examples drawn at random from the training set, $\lambda \in [0, 1]$.

- *mixup* extends the training distribution by incorporating the prior knowledge that linear interpolations of feature vectors should lead to linear interpolations of the associated targets.

- Note that for images, we take as training examples mixtures of two different images. Even though the mixtures do not look like real images, this data augmentation method works and improves generalization.

Adversarial examples

- Training of neural networks:

$$\frac{1}{N}\sum_{n=1}^{N}\mathcal{L}(\mathbf{x}^{(n)},\mathbf{y}^{(n)},\mathbf{w})\to\min_{\mathbf{w}}$$

  where $\mathcal{L}(\mathbf{x}^{(n)},\mathbf{y}^{(n)}\mathbf{w})$ is, for example, a cross-entropy loss.

- Szegedy et al. (2014) discovered that it is very easy to fool a trained neural network. One can modify a given input $\mathbf{x}$ such that the output of the network changes:

$$\mathcal{L}(\mathbf{x}+\mathbf{r},\mathbf{y},\mathbf{w})\to\max_{\mathbf{r}}$$

  keeping the adversarial perturbation $\mathbf{r}$ small, for example, $\|\mathbf{r}\|\leq\varepsilon$.

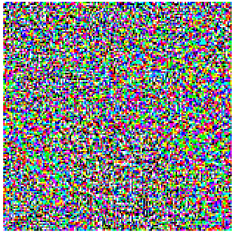
- Modified input $\mathbf{x}+\mathbf{r}$ is called an adversarial example.

FGSM attack (Goodfellow et al., 2014)

- Finding adversarial examples is surprisingly easy. For example, with the fast gradient sign method (FGSM):

$$\mathbf{x} + \mathbf{r} = \mathbf{x} + \varepsilon \, \text{sign}(\nabla_{\mathbf{x}} \mathcal{L}(\mathbf{w}, \mathbf{x}, \mathbf{y}))$$

$+ .007 \times$



$=$

$\mathbf{x}$, $f(\mathbf{x}) =$ "panda"

$\text{sign}(\nabla_{\mathbf{x}} \mathcal{L}(\mathbf{w}, \mathbf{x}, \mathbf{y}))$

$\mathbf{x} + \mathbf{r}$, $f(\mathbf{x}) =$ "gibbon"

Adversarial training

- Adversarial examples are difficult for neural networks, including them in the training set helps reduce the test error. This is called adversarial training.
- Adversarial training is data augmentation with adversarial examples.
- The existence of adversarial examples motivated a new subfield of deep learning in which they develop techniques to defend neural networks against adversarial attacks.

Madry's defense

- Madry's defense model (Madry et al., 2017) is one of the strongest defense models.
- Recall standard optimization:

$$E_{(\mathbf{x},\mathbf{y})\sim\mathcal{D}}\left[\mathcal{L}(\mathbf{w},\mathbf{x},\mathbf{y})\right] \rightarrow \min_{\mathbf{w}}$$

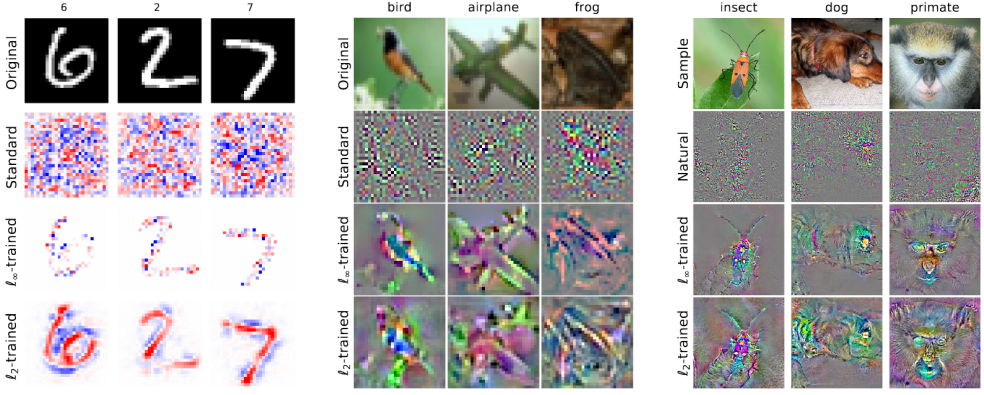
- Madry's defense model:

$$\min_{\mathbf{w}} E_{(\mathbf{x},\mathbf{y})\sim\mathcal{D}}\left[\max_{\delta\in\mathcal{S}} \mathcal{L}(\mathbf{w},\mathbf{x}+\delta,\mathbf{y})\right]$$

- Instead of feeding clean training samples $\mathbf{x}$, we feed the worst adversarial examples found with another optimization procedure.
- Saddle point problem: composition of an inner maximization problem and an outer minimization problem.

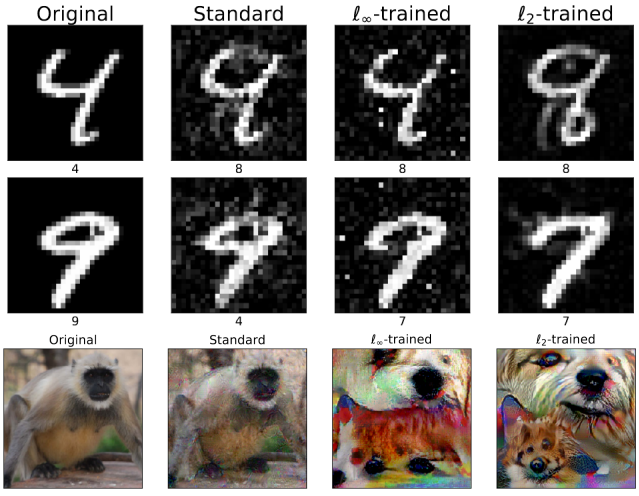
Adversarial training helps develop more meaningful representations

- Gradients wrt inputs look much more meaningful for an adversarily trained network.



images from (Madry et al., 2018)

Adversarial examples look more meaningful



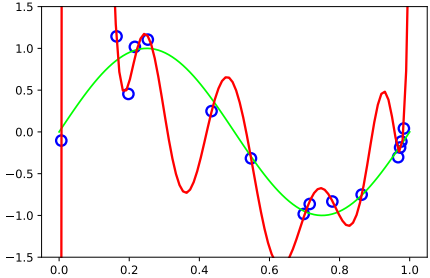
images from (Madry et al., 2018)

Rethinking generalization

(Zhang et al., 2016)

Conventional wisdom

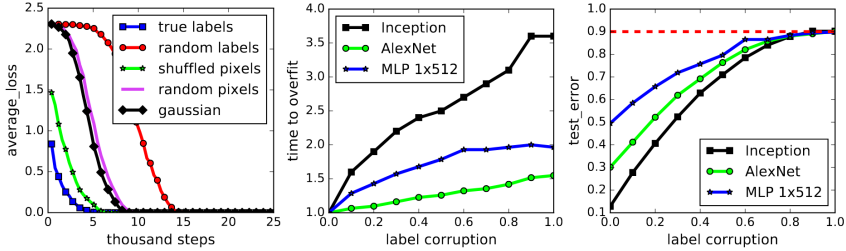
- The model is too flexible for the amount of training data.
- Wikipedia: An overfitted model is a statistical model that contains more parameters than can be justified by the data.



Regression with a polynomial function of order $M = 12$

Rethinking generalization (Zhang et al., 2016)

- Deep neural networks easily fit random labels:
  - The effective capacity of neural networks is sufficient for memorizing the entire data set.
  - Optimization on random labels remains easy.
- Same networks exhibit remarkably small difference between training and test performance.



(a) learning curves (b) convergence slowdown (c) generalization error growth

Fitting random labels and random pixels on CIFAR10.

The role of explicit regularization

- Explicit regularization may improve generalization performance, but is neither necessary nor by itself sufficient for controlling generalization error.

model	# params	random crop	weight decay	train accuracy	test accuracy
Inception	1,649,402	yes	yes	100.0	89.05
		yes	no	100.0	89.31
		no	yes	100.0	86.03
		no	no	100.0	85.75
(fitting random labels)		no	no	100.0	9.78

The training and test accuracy of various models on CIFAR10.

- Batch normalization is usually found to improve the generalization performance, even though it was not explicitly designed for regularization.

model	# params	random crop	weight decay	train accuracy	test accuracy
Inception	1,649,402	yes	yes	100.0	89.05
		yes	no	100.0	89.31
		no	yes	100.0	86.03
		no	no	100.0	85.75
(fitting random labels)		no	no	100.0	9.78
Inception w/o BatchNorm	1,649,402	no	yes	100.0	83.00
		no	no	100.0	82.00
(fitting random labels)		no	no	100.0	10.12

The training and test accuracy of various models on CIFAR10.

- Stochastic gradient descent (SGD) may acts as an implicit regularizer:
  - For linear models, SGD always converges to a solution with small norm.

Hyperparameter search

Selecting hyperparameters

- Hyperparameter search: use the performance on the validation set to select the optimal values of the hyperparameters.
- Hyperparameters that you may want to tune:
  - learning rate schedule
  - transformations used for data augmentation
  - weight decay coefficient
  - dropout rate
  - mini-batch size
  - number of layers
  - number of neurons
  - convolution kernel width
  - nonlinearity
- What works best in practice (this is not the case in the home assignment :-)):
  - A large model combined with strong regularization.
  - The training error is very low.

Hyperparameter search: Grid search

- Select a fixed set of possible values for each hyperparameter.
- Compute the validation loss for all hyperparameter combinations.
- Problems:
  - Many evaluations will be unnecessary if some hyperparameters are non-influential.
  - Computational cost increases exponentially with the number of hyperparameters.

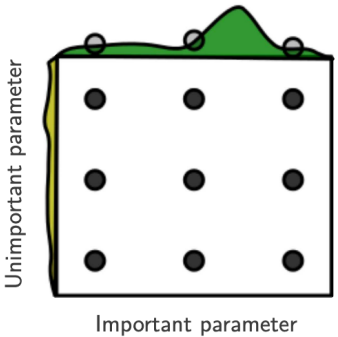


image from (Bergstra and Bengio, 2012)

Hyperparameter search: Random search

- Random combinations of the hyperparameters are formed and evaluated.
- Advantages:
  - Random search does not waste evaluations for non-influential hyperparameters.
  - More convenient and faster than grid search.

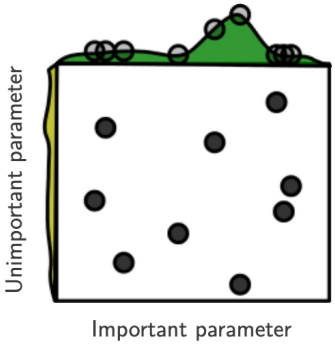


image from (Bergstra and Bengio, 2012)

Home assignment

Assignment `04_reg`

- First notebook: Experiment with different regularization methods on a toy regression problem.
- Second notebook: Implement and tune a recommender system.
- In order to achieve good performance on the test set, you will have to use regularization techniques and tune the (hyper)parameters.

How to represent users or items

- A simple representation is a one-hot vector. For example, user i can be represented with vector $\mathbf{z}_i$ such that $z_i = 1$, $z_{j \neq i} = 0$.
- Better representaion:
  - represent each user i as a vector $\mathbf{w}_i$
  - treat all vectors $\mathbf{w}_i$ as model parameters and tune them in the training procedure
  - this is equivalent to $\mathbf{W}\mathbf{z}_i$ where $\mathbf{W}$ is a matrix of "embeddings" (with vectors $\mathbf{w}_i$ in its columns).
- This is implemented in `torch.nn.Embedding(num_embeddings, embedding_dim)`
  - `num_embeddings` is the size of the dictionary
  - `embedding_dim` is the size of each embedding vector $\mathbf{w}_i$

Recommended reading

- Chapter 7 of Deep learning book
- References in the slides