Regularization - decrease generalization error, but not training error
Regularization trades increases bias for reduced variance, at a reasonable cost

Limit capacity of a network by adding a parameter norm penalty
$\tilde{J}(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) = J(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) + \alpha\Omega(\boldsymbol{\theta})$ Larger values of alpha = more regularization
We do not induce much variance by leaving the biases unregularized, since it only needs to observe 1 variable (whereas weights observe how 2 vars interact)
w indicates all the weights, which should be affected, while theta = all the params (including b)

Weight decay / Ridge regression / Tikhonov regularization / L^2 parameter norm adds a regularization term $\frac{1}{2}\|\boldsymbol{w}\|_2^2$ driving weights towards the origin

The gradient of the regularized objective function J˜ is $\tilde{J}(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) = \frac{\alpha}{2}\boldsymbol{w}^\top \boldsymbol{w} + J(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}),$

with the corresponding parameter gradient $\nabla_{\boldsymbol{w}}\tilde{J}(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) = \alpha\boldsymbol{w} + \nabla_{\boldsymbol{w}}J(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}).$
We take a gradient descent step with $\boldsymbol{w} \leftarrow (1 - \epsilon\alpha)\boldsymbol{w} - \epsilon\nabla_{\boldsymbol{w}}J(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}).$
In a single step, the weight vector is being multiplicatively shrunk

We simplify the objective function by making a quadratic approximation to the objective function
$\boldsymbol{w}^* = \arg\min_{\boldsymbol{w}} J(\boldsymbol{w}).$

The approximation J^ is given by $\hat{J}(\boldsymbol{\theta}) = J(\boldsymbol{w}^*) + \frac{1}{2}(\boldsymbol{w} - \boldsymbol{w}^*)^\top \boldsymbol{H}(\boldsymbol{w} - \boldsymbol{w}^*)$
H is the Hessian of J w/ respect to w evaluated at w*
Since w* is the minimum where the gradient vanishes, there is no first order term
H is positive semidefinite since w* is the location of a minnimum of J

The minimum of J occurs where its gradient $\nabla_{\boldsymbol{w}}\hat{J}(\boldsymbol{w}) = \boldsymbol{H}(\boldsymbol{w} - \boldsymbol{w}^*)$ is equal to 0
Modify the above equation by adding the weight decay gradient, and solve for the min

$$\begin{aligned}\alpha\tilde{\boldsymbol{w}} + \boldsymbol{H}(\tilde{\boldsymbol{w}} - \boldsymbol{w}^*) &= 0\\ (\boldsymbol{H} + \alpha\boldsymbol{I})\tilde{\boldsymbol{w}} &= \boldsymbol{H}\boldsymbol{w}^*\\ \tilde{\boldsymbol{w}} &= (\boldsymbol{H} + \alpha\boldsymbol{I})^{-1}\boldsymbol{H}\boldsymbol{w}^*.\end{aligned}$$

w-bar represents the location of the minimum
As alpha approaches 0, the regularized solution approaches w*

Decompose H into a diagonal matrix and an orthonormal basis of eigenvectors | $\boldsymbol{H} = \boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^\top.$

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

Weight decay rescales w* along the axis defined by eigenvectors of H => the component of w* aligned with the i-th eigenvector of H is rescaled by $\frac{\lambda_i}{\lambda_i+\alpha}$.
When the eigenvalues of H are very large, therefore, lambda$_i$ >> a, regularization does nothing

Only directions along which the parameters contribute significantly to reducing the objective function are preserved; barely significant directions are ignored

For linear regression, the cost function can be expressed in terms of the sum of squared errors
$(\boldsymbol{X}\boldsymbol{w}-\boldsymbol{y})^\top(\boldsymbol{X}\boldsymbol{w}-\boldsymbol{y})$.

With L2 reg, it becomes: $(\boldsymbol{X}\boldsymbol{w}-\boldsymbol{y})^\top(\boldsymbol{X}\boldsymbol{w}-\boldsymbol{y})+\frac{1}{2}\alpha\boldsymbol{w}^\top\boldsymbol{w}$.

The normal equation for the solution changes from $\boldsymbol{w}=(\boldsymbol{X}^\top\boldsymbol{X})^{-1}\boldsymbol{X}^\top\boldsymbol{y}$ to
$\boldsymbol{w}=(\boldsymbol{X}^\top\boldsymbol{X}+\alpha\boldsymbol{I})^{-1}\boldsymbol{X}^\top\boldsymbol{y}$.

Without L^2, the closed form solution has the covariance matrix $\frac{1}{m}\boldsymbol{X}^\top\boldsymbol{X}$, which is replaced with
$(\boldsymbol{X}^\top\boldsymbol{X}+\alpha\boldsymbol{I})^{-1}$
Diagonal entries of this new matrix correspond to variance; L^2 causes the learning algorithm to perceive X as having a higher variance, shrinking weights on features w/ low covariance to the outputs compared to the added variance

L^1 regularization is defined as $\Omega(\boldsymbol{\theta})=||\boldsymbol{w}||_1=\sum_i|w_i|,$
$\tilde{J}(\boldsymbol{w};\boldsymbol{X},\boldsymbol{y})=\alpha||\boldsymbol{w}||_1+J(\boldsymbol{w};\boldsymbol{X},\boldsymbol{y}),$ w/ the gradient
$\nabla_{\boldsymbol{w}}\tilde{J}(\boldsymbol{w};\boldsymbol{X},\boldsymbol{y})=\alpha\text{sign}(\boldsymbol{w})+\nabla_{\boldsymbol{w}}J(\boldsymbol{X},\boldsymbol{y};\boldsymbol{w})$ where sign(w) is the sign of w applied element wise
Regularization contribution does not scale linearly with each w$_i$ but it is a constant factor with a sign equal to sign(w$_i$) => algebraic solutions will not be clean
pg 1

Make a simplifying assumption that the Hessian is diagonal
$\boldsymbol{H}=\text{diag}([H_{1,1},\ldots,H_{n,n}])$, where H$_{i,i}$ > 0
This assumption is true if the linreg problem has been preprocessed to remove correlation between input features, which can be done through PCA
The quadratic approximation decomposes into a sum over the params
$$\hat{J}(\boldsymbol{w};\boldsymbol{X},\boldsymbol{y})=J(\boldsymbol{w}^*;\boldsymbol{X},\boldsymbol{y})+\sum_i\left[\frac{1}{2}H_{i,i}(\boldsymbol{w}_i-\boldsymbol{w}_i^*)^2+\alpha|w_i|\right].$$

The analytic solution is given by $w_i=\text{sign}(w_i^*)\max\left\{|w_i^*|-\frac{\alpha}{H_{i,i}},0\right\}.$

Consider when w_i* > 0 for all i

- If w_i* < alpha/$H_{i,i}$ => optimal w_i* is 0 since the contribution of J is overwhelmed by L_1 reg, pushing w_i to 0
- If w_i* > alpha/$H_{i,i}$ regularization shifts it to zero by alpha/$H_{i,i}$

If w_i* < 0, L^1 makes w_i less negative by the same shift

L^1 results in a more sparse solution
If we add the same constraints for a positive definite, diagonal Hessian to L^2 reg, we find that

$$\tilde{w}_i = \frac{H_{i,i}}{H_{i,i}+\alpha} w_i^*$$

If w_i* is nonzero, then so is the result $w_i^{\sim}$ <= L^2 reg does not sparsify

The sparsity propery of L^1 is used as a feature selection mechanism; LASSO integrates an L^1 penalty with a linear model and least squares cost function, causing a subset of params = 0

Many regularizatoin strategies can be interpreted as MAP Bayesian inference, w/ L^2 reg equivalent to MAP with a Gaussian prior on the weights
For L^1, the penalty alpha * omega(w) used to regularize a cost function is equivalent to the log prior term maximized by MAP when the prior is an isotropic Laplace distribution over R^n

$$\log p(\boldsymbol{w}) = \sum_i \log \text{Laplace}(w_i; 0, \frac{1}{\alpha}) = -\alpha||\boldsymbol{w}||_1 + n\log\alpha - n\log 2.$$

log alpha - log 2 terms do not depend on w and can be ignored

Norm Penalties as Constrained Optimization

$$\tilde{J}(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) = J(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) + \alpha\Omega(\boldsymbol{\theta}).$$

Recall we can minimize a function subject to constraints by creating a Generalized Lagrange function consisting of the original function plus the penalties, a product between a KKT coefficient, and a function representing whether the constraint is satisfied

To constrain omega(theta) to be less than k: $\mathcal{L}(\boldsymbol{\theta}, \alpha; \boldsymbol{X}, \boldsymbol{y}) = J(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) + \alpha(\Omega(\boldsymbol{\theta}) - k).$

The solution is given by $$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} \max_{\alpha, \alpha\geq 0} \mathcal{L}(\boldsymbol{\theta}, \alpha).$$

alpha increases for omega(theta) > k, and decreases if it is less than k; a*, the optimal value, encourages omega(theta) to shrink but not enough to make it less than k
We can fix a* and view the problem as a function of theta

$$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}, \alpha^*) = \arg\min_{\boldsymbol{\theta}} J(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) + \alpha^*\Omega(\boldsymbol{\theta}).$$

omega is the norm penalty

A larger alpha corresponds to a smaller constraint region; we do not know the size of the constraint region imposed by alpha since it does not tell us the value of k
To use explicit constraints, we can make grad. descent take a step downhill on J(theta) and project theta back to the nearest point satisfying omega(theta) < k => useful if we do not want to spend time searching for an alpha corresponding to k

Using explicit constraints can prevent units from becoming dead (which penalties do as they can cause the cost function to get stuck in a local minimum)
Explicit constraints also prevent feedback loops (large weights cause large gradients which cause large weights, and so on until numerical overflow)
Constraining the norm of each column of a weight matrix of a neural net layer, rather than the weight matrix's Frobenius norm, makes sure each neuron does not die

Regularization and Under-Constrained Problems
Many linear models depend on inverting the matrix X^TX which cannot be done if it is singular, whenever the data generating distribution has no variance in a certain direction or when no variance is observed because there are fewer examples than input features
Instead, regularization will focus on inverting $\boldsymbol{X}^\top \boldsymbol{X} + \alpha \boldsymbol{I}$, which is guaranteed invertible
It is possible for a problem with no closed form solution to be underdetermined

- If linearly separable, logistic regression works for w, 2w, … , nw and grad. descent will continually increase the weights until overflow <= this is known as underdetermined
- Weight decay causes grad. descent to quit increasing the magnitude of the weights when slope of the likelihood = alpha, the weight decay coefficient

We can solve underdetermined problems using the Moore-Penrose pseudoinverse

$$\boldsymbol{X}^+ = \lim_{\alpha \searrow 0} (\boldsymbol{X}^\top \boldsymbol{X} + \alpha \boldsymbol{I})^{-1} \boldsymbol{X}^\top$$

This performs linreg with weight decay => it is the limit as the regularization coefficient shrinks to 0, and can be interpreted as stabilizing underdetermined problems using regularization

(Skipped section about dataset augmentation)

Noise Robustness
Adding noise to the inputs is an effective regularization strategy and is equivalent to imposing a penalty on the norm of the weights, and even beyond that in general
Noise can also be added to weights, which is done in recurrent networks

- Bayesian considers model weights uncertain and representable via a probability distribution which incorporates this uncertainty, which can be done by adding noise

$J = \mathbb{E}_{p(x,y)} \left[(\hat{y}(\boldsymbol{x}) - y)^2 \right]$ Noise to weights can also be traditional regularization
m labeled training examples $\{(\boldsymbol{x}^{(1)}, y^{(1)}), \ldots, (\boldsymbol{x}^{(m)}, y^{(m)})\}$.
Each input includes a random perturbation $\epsilon_{\boldsymbol{W}} \sim \mathcal{N}(\boldsymbol{\epsilon}; \boldsymbol{0}, \eta \boldsymbol{I})$
The perturbed model is denoted as $\hat{y}_{\epsilon_{\boldsymbol{W}}}(\boldsymbol{x})$.

The objective function becomes
$$\begin{aligned}\tilde{J}_{\boldsymbol{W}} &= \mathbb{E}_{p(\boldsymbol{x},y,\boldsymbol{\epsilon_W})}\left[(\hat{y}_{\boldsymbol{\epsilon_W}}(\boldsymbol{x}) - y)^2\right] \\ &= \mathbb{E}_{p(\boldsymbol{x},y,\boldsymbol{\epsilon_W})}\left[\hat{y}^2_{\boldsymbol{\epsilon_W}}(\boldsymbol{x}) - 2y\hat{y}_{\boldsymbol{\epsilon_W}}(\boldsymbol{x}) + y^2\right].\end{aligned}$$
For a small learning rate, the minimization of J with added weight noise is equivalent to minimization of J with an additional regularization term: $\eta\mathbb{E}_{p(\boldsymbol{x},y)}\left[\|\nabla_{\boldsymbol{W}}\hat{y}(\boldsymbol{x})\|^2\right]$
This encourages parameters to go to regions where small weight perturbations have a small influence on the output => minima surrounded by flat regions

For linreg, this simplifies to $\eta\mathbb{E}_{p(\boldsymbol{x})}\left[\|\boldsymbol{x}\|^2\right]$, which is not a function of params and does not contribute to the gradient of J˜

Label smoothing regularizes a model based on softmax w/ k output values by replacing the 0 and 1 classification targets with $\frac{\epsilon}{k-1}$ and 1 - eps. since there is a small chance each label is mistaken
Label smoothing prevents the pursuit of hard probabilities, which are impossible to get

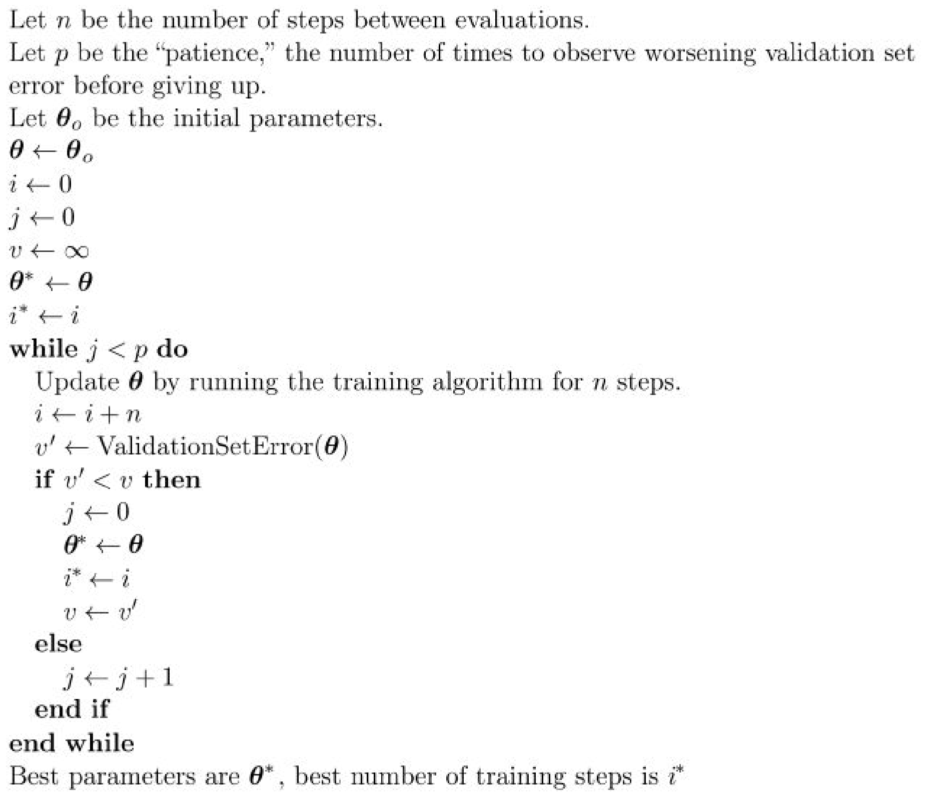
Semi-Supervised Learning
Semi-supervised learning uses unsupervised and supervised examples to predict P(y|x)
In deep learning it refers to learning a representation $\boldsymbol{h} = f(\boldsymbol{x})$.
Learns a representation such that examples in the same class have similar representations
Can construct generative models of P(x) or P(x, y) which shares parameters with a discriminative model of P(y|x)
The supervised criterion can be traded off w/ the unsupervised one

- Assumes that P(x) is connected to structure of P(y|x) by shared parameters

Multi-Task Learning
Multi-task learning pools examples arising out of similar tasks to constrain the parameters and generalize better
Different tasks y which take the same input x can have the same representation h
Task specific parameters are for a specific task, and are the upper layers of a neural net
Generic parameters, shared across similar tasks, are the lower layers
The statistical strength of the shared parameteers is due to extra training examples
The prior: among factors explaining variations observed in data associated with different tasks, some are shared across two or more tasks

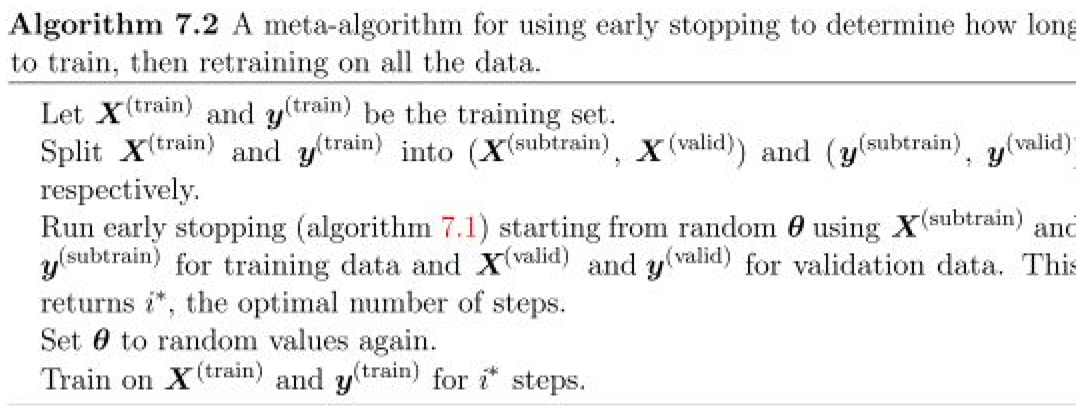
Early Stopping
Return to the point with the lowest validation error, rather than training error to prevent overfit

Save the parameters every time the model improves its validation error



```
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 $i^*$
```

The number of training steps can be considered another hyperparameter
Ideally, the validation set is being processed on a different GPU in parallel to improve efficiency
Once early stopping is used to determine the optimal # of steps, the validation data can be used for training as well



Algorithm 7.2 A meta-algorithm for using early stopping to determine how long to train, then retraining on all the data.

```
Let $\boldsymbol{X}^{(\text{train})}$ and $\boldsymbol{y}^{(\text{train})}$ be the training set.
Split $\boldsymbol{X}^{(\text{train})}$ and $\boldsymbol{y}^{(\text{train})}$ into $(\boldsymbol{X}^{(\text{subtrain})}, \boldsymbol{X}^{(\text{valid})})$ and $(\boldsymbol{y}^{(\text{subtrain})}, \boldsymbol{y}^{(\text{valid})}$
respectively.
Run early stopping (algorithm 7.1) starting from random $\boldsymbol{\theta}$ using $\boldsymbol{X}^{(\text{subtrain})}$ and
$\boldsymbol{y}^{(\text{subtrain})}$ for training data and $\boldsymbol{X}^{(\text{valid})}$ and $\boldsymbol{y}^{(\text{valid})}$ for validation data. This
returns $i^*$, the optimal number of steps.
Set $\boldsymbol{\theta}$ to random values again.
Train on $\boldsymbol{X}^{(\text{train})}$ and $\boldsymbol{y}^{(\text{train})}$ for $i^*$ steps.
```

Early stopping is a good regularizer as it does not require its gradient to be computed
Early stopping acts as a regularizer by restricting the optimization procedure to a small volume of parameter space around the initial weights $\boldsymbol{\Theta}_0$

- The effective capacity is given by eT, where e is the learning rate and T is the # of steps
- eT behaves as if it were the reciprocal of the weight decay coefficient

Consider a linear model, where the only params are weights (not biases)

The cost function J($\boldsymbol{\Theta}$) can be modeled w/ a quadratic approximation near w*, the optimal wts

$$\hat{J}(\boldsymbol{\theta}) = J(\boldsymbol{w}^*) + \frac{1}{2}(\boldsymbol{w} - \boldsymbol{w}^*)^\top \boldsymbol{H}(\boldsymbol{w} - \boldsymbol{w}^*),$$

If w* is a minimum of J(w), the hessian H is positive semidefinite
A taylor series approximation gives the gradient as:

$$\nabla_{\boldsymbol{w}} \hat{J}(\boldsymbol{w}) = \boldsymbol{H}(\boldsymbol{w} - \boldsymbol{w}^*).$$

Let the initial parameter vector be **0**
Gradient descent does the following
pg 2

$$\begin{aligned} \boldsymbol{w}^{(\tau)} &= \boldsymbol{w}^{(\tau-1)} - \epsilon \nabla_{\boldsymbol{w}} \hat{J}(\boldsymbol{w}^{(\tau-1)}) \\ &= \boldsymbol{w}^{(\tau-1)} - \epsilon \boldsymbol{H}(\boldsymbol{w}^{(\tau-1)} - \boldsymbol{w}^*) \\ \boldsymbol{w}^{(\tau)} - \boldsymbol{w}^* &= (\boldsymbol{I} - \epsilon \boldsymbol{H})(\boldsymbol{w}^{(\tau-1)} - \boldsymbol{w}^*). \end{aligned}$$

Rewrite this expression in the space of eigenvectors of H, using the fact $\boldsymbol{H} = \boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^\top$ where Q is an orthonormal basis of eigenvectors

$$\begin{aligned} \boldsymbol{w}^{(\tau)} - \boldsymbol{w}^* &= (\boldsymbol{I} - \epsilon \boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^\top)(\boldsymbol{w}^{(\tau-1)} - \boldsymbol{w}^*) \\ \boldsymbol{Q}^\top(\boldsymbol{w}^{(\tau)} - \boldsymbol{w}^*) &= (\boldsymbol{I} - \epsilon \boldsymbol{\Lambda})\boldsymbol{Q}^\top(\boldsymbol{w}^{(\tau-1)} - \boldsymbol{w}^*) \end{aligned}$$

Assume that w^0 is **0** and epsilon is small enough to guarantee $|1 - \epsilon\lambda_i| < 1$ (which allows us to say [I - [...])^T], the parameter trajectory during training is as follows

$$\boldsymbol{Q}^\top \boldsymbol{w}^{(\tau)} = [\boldsymbol{I} - (\boldsymbol{I} - \epsilon\boldsymbol{\Lambda})^\tau]\boldsymbol{Q}^\top \boldsymbol{w}^*.$$

Rearrange the equation for $Q^T w^\sim$ from L^2 regularization

$$\boldsymbol{Q}^\top \tilde{\boldsymbol{w}} = (\boldsymbol{\Lambda} + \alpha\boldsymbol{I})^{-1}\boldsymbol{\Lambda}\boldsymbol{Q}^\top \boldsymbol{w}^*$$
$$\boldsymbol{Q}^\top \tilde{\boldsymbol{w}} = [\boldsymbol{I} - (\boldsymbol{\Lambda} + \alpha\boldsymbol{I})^{-1}\alpha]\boldsymbol{Q}^\top \boldsymbol{w}^*$$

We can see that if the hyperparameters epsilon, T, and alpha are chosen such that
$(\boldsymbol{I} - \epsilon\boldsymbol{\Lambda})^\tau = (\boldsymbol{\Lambda} + \alpha\boldsymbol{I})^{-1}\alpha,$ then early stopping is equivalent to L^2 reg
If all lambda_i are small, $\epsilon\lambda_i \ll 1 \text{ and } \lambda_i/\alpha \ll 1$

Number of training iterations is inversely proportional to L^2 reg param, and the inverse of T(epsilon) plays the weight decay coefficient
Relating early stopping to L^2 reg, values corresponding to directions of significant curvature learn early relative to directions of less curvature

Parameter Tying and Parameter Sharing

We can assert a prior by imposing dependencies between model parameters
Suppose we have two models performing the same classification task, with different input distributions: model A w/ parameters w^A and model B w/ parameters w^B
The two models have related outputs as follows: $\hat{y}^{(A)} = f(\boldsymbol{w}^{(A)}, \boldsymbol{x})$; $\hat{y}^{(B)} = g(\boldsymbol{w}^{(B)}, \boldsymbol{x})$.
If the tasks are similar enough, we say that for all i, $w_i^A = w_i^B$
We use a parameter L^2 norm penalty: $\Omega(\boldsymbol{w}^{(A)}, \boldsymbol{w}^{(B)}) = \|\boldsymbol{w}^{(A)} - \boldsymbol{w}^{(B)}\|_2^2$.

Another way to regularize params to be close to one another is to use constraints: to force sets of parameters to be equal
This regularization method is known as parameter sharing, since various models share a unique set of parameters
Only a subset of parameters, the unique set, for each model needs to be stored in memory

CNNs share parameters across multiple image locations, since photos are invariant to translation => the feature (hidden unit w/ same weights) is computed over various locations
This significantly lowers # of unique model params and significantly increases network sizes w/o requiring an increase in training data

Sparse Representations
Placing a penalty on the activations of a network, encouraging them to be sparse imposes a complicated penalty on the model's parameters
L^1 reg induces a sparse parametrization, encouraging many of the params to be 0
Representational sparsity describes a representation where many of the elements are 0 or close to 0; the below is a sparsely parametrized linear model

$$\underset{\boldsymbol{y} \in \mathbb{R}^m}{\begin{bmatrix} 18 \\ 5 \\ 15 \\ -9 \\ -3 \end{bmatrix}} = \underset{\boldsymbol{A} \in \mathbb{R}^{m \times n}}{\begin{bmatrix} 4 & 0 & 0 & -2 & 0 & 0 \\ 0 & 0 & -1 & 0 & 3 & 0 \\ 0 & 5 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & -1 & 0 & -4 \\ 1 & 0 & 0 & 0 & -5 & 0 \end{bmatrix}} \underset{\boldsymbol{x} \in \mathbb{R}^n}{\begin{bmatrix} 2 \\ 3 \\ -2 \\ -5 \\ 1 \\ 4 \end{bmatrix}}$$

The one below is linear regression w/ a sparse representation h of the data x

$$\underset{\boldsymbol{y} \in \mathbb{R}^m}{\begin{bmatrix} -14 \\ 1 \\ 19 \\ 2 \\ 23 \end{bmatrix}} = \underset{\boldsymbol{B} \in \mathbb{R}^{m \times n}}{\begin{bmatrix} 3 & -1 & 2 & -5 & 4 & 1 \\ 4 & 2 & -3 & -1 & 1 & 3 \\ -1 & 5 & 4 & 2 & -3 & -2 \\ 3 & 1 & 2 & -3 & 0 & -3 \\ -5 & 4 & -2 & 2 & -5 & -1 \end{bmatrix}} \underset{\boldsymbol{h} \in \mathbb{R}^n}{\begin{bmatrix} 0 \\ 2 \\ 0 \\ 0 \\ -3 \\ 0 \end{bmatrix}}$$

h represents the info in x, but does so with a sparse vector

Adding to the loss function J a norm penalty on the representation can achieve representational sparsity: $\tilde{J}(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) = J(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) + \alpha\Omega(\boldsymbol{h})$

$\Omega(\boldsymbol{h}) = ||\boldsymbol{h}||_1 = \sum_i |h_i|$. L^1 is only one choice of penalty which can result in sparse rep.

Hard constraint on activations can achieve representational sparsity
Orthogonal matching pursuit encodes an input x with the representation h which solves the constrained optimization problem

$$\underset{\boldsymbol{h}, ||\boldsymbol{h}||_0 < k}{\arg\min} ||\boldsymbol{x} - \boldsymbol{W}\boldsymbol{h}||^2.$$

where $||h||_0$ is the # of nonzero entries of h
This problem can be efficiently solved when W is orthogonal; this method is called OMP-k w/ k indicating the # of nonzero features allowed

Bagging and Other Ensemble Methods
Bagging, or bootstrap aggregating, reduces generalization error by combining several methods which then vote on the output for the test set
Averaging works since different models won't make the same errors on the test set

Consider a set of k regression models, each making an error of ϵ_i with the errors drawn from a normal distribution with variance $\mathbb{E}[\epsilon_i^2] = v$ and covariance $\mathbb{E}[\epsilon_i \epsilon_j] =$
The error made by the average prediction of all ensemble methods is 1/k(**Σ**$_i$ epsilon$_i$)
The expected squared error is given by

$$\mathbb{E}\left[\left(\frac{1}{k}\sum_i \epsilon_i\right)^2\right] = \frac{1}{k^2}\mathbb{E}\left[\sum_i \left(\epsilon_i^2 + \sum_{j \neq i} \epsilon_i \epsilon_j\right)\right]$$
$$= \frac{1}{k}v + \frac{k-1}{k}c.$$

When the errors are perfectly correlated, and c = v, the expected squared error is v, so model averaging does not help at all
When the errors are perfectly uncorrelated and c = 0, the expected squared error is v/k; the ensemble will perform at least as well as any of its members, and much better if the errors made are independent

Bagging allows the same kind of model, training objective, and objective function to be used several times
Bagging involves constructing k different datasets, each with the same # of examples as the original dataset, sampled with replacement; each dataset is missing some examples from the original dataset, and contains multiple duplicates, causing variance between each model
Even networks trained on the same examples will have multiple solution points as a result of random initialization, causing different members to make partially independent errors
Benchmark comparisons are made using a single model, not ensembles

Boosting constructs an ensemble w/ a higher capacity than each individual model, building ensembles of neural nets by adding networks to the ensemble

Dropout

Dropout provides an inexpensive approximation to training and lets us evaluate a bagged ensemble of exponentially many networks

Dropout trains the ensemble consisting of all networks that can be formed by removing non-output units from an underlying base network

We can remove a unit from a network by multiplying its output by 0 (in most cases, but this may need to be trivially modified when dealing with things like RBF)

Dropout aims to replicate bagging; using mini-batch learning algorithm, we randomly sample a different binary mask to apply to the input and hidden units

The probablity of sampling a mask value of 1, indicating the unit should be included, is a hyperparameter; also the probability of whether an input is used or not can be adjusted

We then run forward prop, and backprop as usual

Suppose a mask vector $\boldsymbol{\mu}$ specifies which units to include, and J($\boldsymbol{\Theta}$, $\boldsymbol{\mu}$) defines the cost of the model defined by parameters $\boldsymbol{\Theta}$ and mask $\boldsymbol{\mu}$

Dropout training consists of minimizing $E_{\boldsymbol{\mu}}$ J($\boldsymbol{\Theta}$,$\boldsymbol{\mu}$); this expectation contains exponentially many terms (all possible masks) but we can obtain an unbiased estimate by sampling values of $\boldsymbol{\mu}$

In dropout, each parent neural network inherits a different subset of params from its parent neural network, making it possible to represent an exponential number of models

With dropout, a tiny fraction of the sub-networks are each trained for a single step, and parameter sharing causes the remaining sub-networks to arrive at good parameter values

Beyond these differences, dropout is essentially the same as bagging

A bagged ensemble must accumulate votes from all its models - this is known as inference

In the case of bagging, each model produces a probability distribution $p^i(y \mid x)$; the prediction of the ensemble is given by the arithmetic mean of these distributions

$$\frac{1}{k}\sum_{i=1}^{k} p^{(i)}(y \mid \boldsymbol{x}).$$

W/ dropout, each sub-model defined by a mask $\boldsymbol{\mu}$ defines the probability distribution p(y | x, $\boldsymbol{\mu}$); the arithmetic mean over all masks is given by $\sum_{\boldsymbol{\mu}} p(\boldsymbol{\mu}) p(y \mid \boldsymbol{x}, \boldsymbol{\mu})$ where p($\boldsymbol{\mu}$) is the probability distribution used to sample $\boldsymbol{\mu}$ at training time

We can approximate the inference with sampling, by averaging the output from 10-20 masks

However, it is possible to obtain a good approximation in one forward prop, by using the geometric mean rather than the arithmetic mean of the ensemble members' predictions

The geometric mean of multiple prob. distributions is not guaranteed to be a probability distribution, so we impose the requirement that the prediction of any event cannot = 0, and we renormalize the resulting product distribution
The unnormalized prob. distribution defined by the geometric mean is given by

$$\tilde{p}_{\text{ensemble}}(y \mid \boldsymbol{x}) = \sqrt[2^d]{\prod_{\boldsymbol{\mu}} p(y \mid \boldsymbol{x}, \boldsymbol{\mu})}$$

where d is the # of units that may have been dropped

TO make predictions, the ensemble must be renormalized

$$p_{\text{ensemble}}(y \mid \boldsymbol{x}) = \frac{\tilde{p}_{\text{ensemble}}(y \mid \boldsymbol{x})}{\sum_{y'} \tilde{p}_{\text{ensemble}}(y' \mid \boldsymbol{x})}.$$

We can approximate p_{ensemble} by evaluating p(y|x) in one model, which contains all units but with the weights going out unit i multiplied by the probability of including unit i
This captures the right expected value of the unit's output - this is known as the weight scaling inference rule, which performs well empirically
Since dropout is usually ½, the weight scaling rule results in dividing the weights by 2 at the end of training, which makes sure that the total expected input at test time (when all units are used) is roughly the same as train time

Consider a softmax regression w/ n inputs represented by vector v:

$$P(\mathrm{y} = y \mid \mathbf{v}) = \text{softmax}\left(\boldsymbol{W}^\top \mathbf{v} + \boldsymbol{b}\right)_y.$$

Multiply with a binary vector d (element wise) to index into the family of the sub-models

$$P(\mathrm{y} = y \mid \mathbf{v}; \boldsymbol{d}) = \text{softmax}\left(\boldsymbol{W}^\top (\boldsymbol{d} \odot \mathbf{v}) + \boldsymbol{b}\right)_y.$$

Ensemble predictor defined by renormalizing geometric mean over all ensemble predictions

$$P_{\text{ensemble}}(\mathrm{y} = y \mid \mathbf{v}) = \frac{\tilde{P}_{\text{ensemble}}(\mathrm{y} = y \mid \mathbf{v})}{\sum_{y'} \tilde{P}_{\text{ensemble}}(\mathrm{y} = y' \mid \mathbf{v})}$$

Denominator represents all predictions

where

$$\tilde{P}_{\text{ensemble}}(\mathrm{y} = y \mid \mathbf{v}) = \sqrt[2^n]{\prod_{\boldsymbol{d} \in \{0,1\}^n} P(\mathrm{y} = y \mid \mathbf{v}; \boldsymbol{d})}.$$

Simplify $\tilde{P}_{\text{ensemble}}$ to see the weight scaling rule is exact

$$\tilde{P}_{\text{ensemble}}(\mathrm{y} = y \mid \mathbf{v}) = \sqrt[2^n]{\prod_{\boldsymbol{d} \in \{0,1\}^n} P(\mathrm{y} = y \mid \mathbf{v}; \boldsymbol{d})}$$

$$= \sqrt[2^n]{\prod_{\boldsymbol{d} \in \{0,1\}^n} \text{softmax}\left(\boldsymbol{W}^\top (\boldsymbol{d} \odot \mathbf{v}) + \boldsymbol{b}\right)_y}$$

$$= \sqrt[2^n]{\prod_{\boldsymbol{d} \in \{0,1\}^n} \frac{\exp\left(\boldsymbol{W}_{y,:}^\top (\boldsymbol{d} \odot \mathbf{v}) + b_y\right)}{\sum_{y'} \exp\left(\boldsymbol{W}_{y',:}^\top (\boldsymbol{d} \odot \mathbf{v}) + b_{y'}\right)}}$$

$$= \frac{\sqrt[2^n]{\prod_{\boldsymbol{d} \in \{0,1\}^n} \exp\left(\boldsymbol{W}_{y,:}^\top (\boldsymbol{d} \odot \mathbf{v}) + b_y\right)}}{\sqrt[2^n]{\prod_{\boldsymbol{d} \in \{0,1\}^n} \sum_{y'} \exp\left(\boldsymbol{W}_{y',:}^\top (\boldsymbol{d} \odot \mathbf{v}) + b_{y'}\right)}}$$

Since P˜ will be renormalized, we can ignore multiplication by factors that are constant in y

$$\tilde{P}_{\text{ensemble}}(\mathrm{y} = y \mid \mathbf{v}) \propto \sqrt[2^n]{\prod_{\boldsymbol{d} \in \{0,1\}^n} \exp\left(\boldsymbol{W}_{y,:}^\top(\boldsymbol{d} \odot \mathbf{v}) + b_y\right)}$$

$$= \exp\left(\frac{1}{2^n} \sum_{\boldsymbol{d} \in \{0,1\}^n} \boldsymbol{W}_{y,:}^\top(\boldsymbol{d} \odot \mathbf{v}) + b_y\right)$$

$$= \exp\left(\frac{1}{2}\boldsymbol{W}_{y,:}^\top \mathbf{v} + b_y\right).$$

=> Discard the denominator

Substituting this into the initial renormalized equation, the softmax classifier has weights ½W

The weight scaling rule is precise in other settings like regression w/ conditionally normal outputs and deep networks without nonlinearities
The approximation is empirically effective for deep nets with nonlinearities
Weight scaling can work better than Monte Carlo approximations, with just 20 samples

Dropout is computationally cheap - O(n) to multiply the binary mask and O(n) to store it, if the implementation requires it; weights must be divided by 2 to start running inference
Dropout does not limit model choice, as it works with neural nets, Boltzmann machines, etc.
However, dropout requires a larger model in order to obtain marginal boosts in validation sets
Unsupervised learning and Bayesian networks can be better than dropout if only few samples

When applied to linreg, dropout is the same as weight decay, with alpha being determined by each input's variance
Stochasticity while training w/ dropout is not necessary (it just approximates the sum over all submodels); fast dropout, reducing stochasticity, is a more principled approximation which can match standard dropout for small datasets

Training the network as a single model with dropout is better than dropout boosting (which is analogous to boosting, with multiple models and little regularization effect)
Regularization effect of bagged ensemble is effective when stochastically sampled members are trained to perform well independently of one another

DropConnect - each product between a single weight and hidden unit can be dropped
Stochastic pooling - ensembles of CNNs w/ each one attending to different spatial locations
Training a network w/ stochastic behavior and averaging over multiple decisions implements bagging with parameter sharing

Dropout is bagging an ensemble of models formed by any random modification, which neural networks can learn to resist (and ideally, those which allow quick approximate inference)
Any modification parametrized by $\boldsymbol{\mu}$ is training an ensemble consisting of p(y | x, $\boldsymbol{\mu}$) for every $\boldsymbol{\mu}$

$\mu \sim N(1, I)$ can outperform dropout on binary masks, since $E(\mu) = 1$, approximate wt. scaling is automatically implemented

Furthermore, dropout trains an ensemble of models that share hidden units, meaning they must be able to perform well regardless of which other hidden units are used in the model, causing the hidden units to be features which are applicable in many contexts

Dropout is highly destructive to the information content of the inputs, rather than the raw inputs themselves; for example, if a model learns an activation which encodes a nose, dropping h_i corresponds to dropping a nose, forcing a model to identify a face using some other feature
This is different than just dropping some features of the nose, as traditional noise does

Dropout uses multiplicative noise, rather than additive, preventing the model from causing h_i to become very large in order to make the noise epsilon insignificant
Batch normalization reparametrizes the model w/ multiplicative and additive noise to improve optimization; however, the noise can have a regularizing effect making dropout unnecessary

Adversarial Training
Even neural nets performing at human accuracy have a nearly 100% error rate on examples intentionally constructed using an optimization procedure searching for an input x' near an arbitrary point x such that the model output is very different at x'
Many times, x' can be so similar to x that a human cannot tell the difference, but the network can distinguish between the original and adversarial network

One can reduce the error rate on the i.i.d test set through adversarial training, which trains on adversarially perturbed examples from the training set
One of the cuases of adversarial examples is excessive linearity
Neural nets are primarily linear, which makes them easy to optimize; however, the value of a linear function changes rapidly by epsilon*$||w||_1$ where epsilon is a small change in inputs
This can compound and cause the model to predict something entirely different
Adversarial training encourages the network to be locally constant in the area of training data

Purely linear models like logistic regression cannot resist adversarial examples as they are forced to be linear; the flexibility of neural nts to learn linear / nonlinear functions allows them to capture linear trends in training data while being able to resist local perturbation

Adversarial examples generated using a label provided by a trained model are called virtual adversarial examples; the classifier may then be trained to assign the same label to x and x'
Requires assumption that small changes in the manifold do not change the class to change

Tangent Distance, Tangent Prop, and Manifold Tangent Classifier
Recall manifold hypothesis - all data lies near a low dimensional manifold
Tangent distance - metric derived from knowledge of manifolds where probability concentrates

Nearest neighbor distance between x_1 and x_2 corresponds to distance between M_1 and M_2 which is their respective manifolds
A cheap alternative to this is to approximate M_i by the tangent plane at x_i and measure the distance between the two tangents or a tangent plane and a point, which can be achieved by solving a low dimensional (same as manifolds) linear system

Tangent prop trains a classifier with an extra penalty to make each output locally invariant to known factors of variation
These factors of variation correspond to movement along the manifold near which examples of the same class concentrate; local invariance requires $\nabla_{\boldsymbol{x}} f(\boldsymbol{x})$ to be orthogonal to the known manifold vectors v^i at x, or the directional derivative of f at x in the directions v^i be small, which can be accomplished by adding a small regularization penalty omega

$$\Omega(f) = \sum_i \left((\nabla_{\boldsymbol{x}} f(\boldsymbol{x}))^\top \boldsymbol{v}^{(i)} \right)^2$$

Tangent vectors are derived from the knowledge of effects of transformations such as rotations
Tangent prop is also useful in the context of supervised learning

In dataset augmentation, the network is explicitly trained to correctly classify distinct inputs created by applying more than an infinitesimal amount of these transformations
Tangent prop regularizes the model to resist perturbation in directions correspoding to the transformations, which only regularizes the model to resist infinitesimal perturbation
The infinitesimal approach can only shrink derivatives of ReLUs by turning units off or shrinking weights, but not by saturating at a high value like how tanh works
Dataset augmentation works well with ReLUs since different subsets of rectified units activate for different transformed versions of each input

Double backprop regularizes the Jacobian to be small
Double backprop and adversarial training require the model should be invariant to all directions of change, while dataset augmentation and tangent prop require model to be invariant to certain specified directions
Adversarial training is the non-infinitesimal version of double backprop

Manifold tangent classifier eliminates the need to know the tangent vectors before; it makes use of autoencoders, which can be used to estimate the manifold tangent vectors
These estimated tangent vectors go beyond classical invariants such as rotation
Manifold tangent classifier uses an autoencoder to learn the manifold structure w/ unsupervised learning, and uses the tangents to regularize a neural net as in tangent prop