22 The Pseudoinverse; Better Generalization for Neural Nets

THE PSEUDOINVERSE AND THE SVD

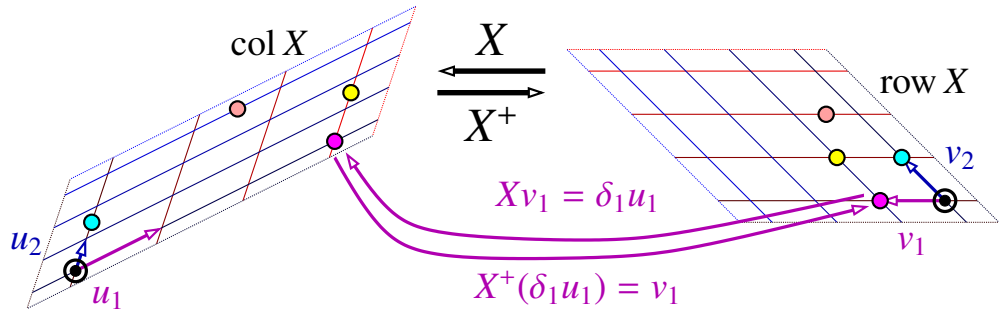
[We're done with unsupervised learning. For the rest of the semester, we go back to supervised learning.]

[The singular value decomposition can give us insight into the pseudoinverse and its use in least-squares linear regression. If you attended Discussion Section 6, you worked through an explanation of this, but now that I've introduced the compact SVD in Lecture 21, I'd like to summarize it.]

Let X be any $n \times d$ matrix. Let $X = UDV^\top$ be its *compact* SVD. Let $r = \operatorname{rank} X$.
Recall that $U \in \mathbb{R}^{n\times r}$, $D \in \mathbb{R}^{r\times r}$ is diagonal & invertible, $V \in \mathbb{R}^{d\times r}$, $U^\top U = I$, $V^\top V = I$.
The Moore–Penrose pseudoinverse of X is $X^+ = VD^{-1}U^\top$. It's $d \times n$.
[This is a better pseudoinverse than the one I defined in Lecture 10, not least because it's always defined.]

Observe:
(1) $XX^+ = UU^\top$, which is symmetric. Proof: $XX^+ = UDV^\top VD^{-1}U^\top = UDD^{-1}U^\top = UU^\top$.
(2) $X^+X = VV^\top$. [The proof is analogous to (1).]
(3) If $r = n$, then $XX^+ = I_{n\times n}$ and X^+ is a right inverse. Proof: U is square, $U^\top U = I \to UU^\top = I$; use (1).
(4) If $r = d$, then $X^+X = I_{d\times d}$ and X^+ is a left inverse. [The proof is analogous to (3) and uses (2).]
(5) By (3), **if X is invertible ($r = n = d$), $X^+ = X^{-1}$. [The pseudoinverse is the inverse when one exists.]**
(6) These are compact SVDs: $X^+ = VD^{-1}U^\top$, $X^\top = VDU^\top$, $(X^+)^\top = UD^{-1}V^\top$.
[If a factorization has the form of a compact SVD, it *is* a compact SVD.]
X^+ is like $X^\top$ with the nonzero singular values inverted.
(7) Given a compact SVD $X = UDV^\top$, $\operatorname{null} X = \operatorname{null} V^\top$.
Proof: $V^\top w = 0 \to Xw = UDV^\top w = 0 \to D^{-1}U^\top UDV^\top w = 0 \to V^\top w = 0$.
(8) By (6) & (7), $\operatorname{null} X^+ = \operatorname{null} U^\top = \operatorname{null} X^\top$ and $\operatorname{null}(X^+)^\top = \operatorname{null} V^\top = \operatorname{null} X$.
So row X^+ = col X and col X^+ = row X. **X^+ has the same four fundamental subspaces as $X^\top$.**
(9) (1) & (2) give eigendecompositions: $XX^+ = \begin{bmatrix} U & U_{\text{null}} \end{bmatrix} \begin{bmatrix} I_{r\times r} & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} U^\top \\ U_{\text{null}}^\top \end{bmatrix}$, $X^+X = \begin{bmatrix} V & V_{\text{null}} \end{bmatrix} \begin{bmatrix} I_{r\times r} & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} V^\top \\ V_{\text{null}}^\top \end{bmatrix}$.
[U_{null} and V_{null} have orthonormal column vectors spanning the null spaces of XX^+ and X^+X.]
(10) By (8) & (9), all have rank r: X, U, V, X^+, XX^+, X^+X.
(11) By (9), every $w \in \operatorname{col} U$ is an eigenvector of XX^+ with eigenvalue 1; all other eigenvalues are 0.
As $\operatorname{col} U = \operatorname{col} X$, **$XX^+$ is identity map on col X.** [Symmetrically,] **X^+X is identity map on row X.**

[In summary, the psuedoinverse is as close to an inverse of X as anything can be. Let's visualize what the pseudoinverse does. When you apply X to a vector in $\operatorname{row} X$, you get a vector in $\operatorname{col} X$; then when you apply X^+ to the result, you get the original vector back.]



rowcol.pdf [The singular vectors are perpendicular, but we are viewing the planes from oblique angles.]

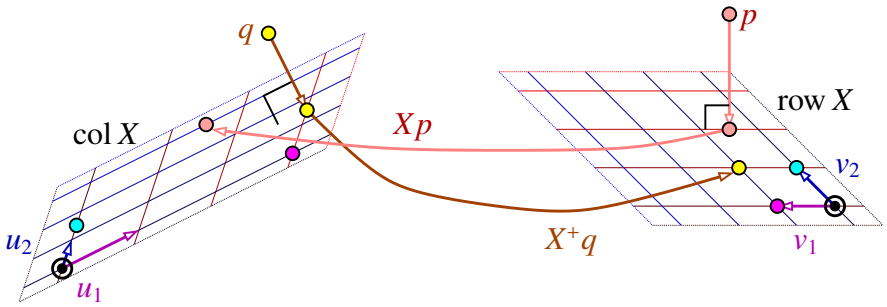
[If we think of X as a linear function that maps row X to col X, and we ignore the other dimensions of $\mathbb{R}^d$ and $\mathbb{R}^n$, then that linear function is a bijection. The inverse of that bijection is the pseudoinverse X^+.]

Linear function $f : \text{row}\, X \to \text{col}\, X, p \mapsto Xp$ is a bijection.
Its inverse is $f^{-1} : \text{col}\, X \to \text{row}\, X, q \mapsto X^+q$.

The r right singular vectors v_i are an orthonormal basis for row X.
The r left singular vectors u_i are an orthonormal basis for col X.
$Xv_i = \delta_i u_i$. $\quad X^+u_i = \frac{1}{\delta_i} v_i$.
[X maps each right singular vector to some scalar multiple of the corresponding left singular vector. The corresponding singular value tells us how much longer the vector gets when we map it. X^+ maps each left singular vector to some scalar multiple of a right singular vector.]

[Usually we don't think of X as a function from row space to column space. Usually we think of X as a function from some bigger space $\mathbb{R}^d$ to a bigger space $\mathbb{R}^n$. In our figure above, X might be a 4×3 matrix, but its rank is only two. Then X isn't a bijection any more, and neither is its pseudoinverse X^+. So X maps every point in $\mathbb{R}^3$ to a point on the plane col X. When you map a three-dimensional space down to two dimensions, it can't be a bijection, so X doesn't have an inverse. Just a pseudoinverse.]

[You can think of X as a function that orthogonally projects a three-dimensional point down onto the row space of X, then uses the bijection above to finish the mapping. Symmetrically, you can think of X^+ as a function that orthogonally projects a four-dimensional point down onto the column space of X, then uses the inverse bijection. Here's an illustration of mapping p to Xp and q to X^+q.]



rowcolpr.pdf

[With the compact SVD, we can show that the pseudoinverse always gives a solution to least-squares linear regression, even when $X^\top X$ is singular.]

Theorem: A solution to the normal equations $X^\top Xw = X^\top y$ is $w = X^+y$.

Proof: $X^\top Xw = X^\top XX^+y = VDU^\top UDV^\top VD^{-1}U^\top y = VD^2D^{-1}U^\top y = VDU^\top y = X^\top y$.

If the normal eq'ns have multiple solutions, $w = X^+y$ is the least-norm solution; i.e., it minimizes $\|w\|$ among all solutions. [If you attended Discussion Section 6, you might have proven this yourself.]

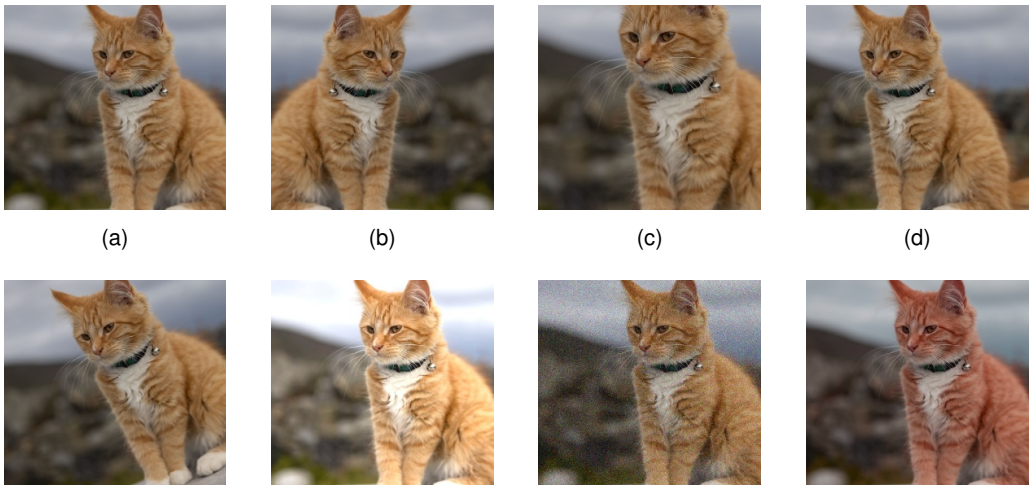
[This way of solving the normal equations is very helpful when $X^\top X$ is singular because $n < d$ or the sample points lie on a subspace of the feature space. But observe that if X has a very small singular value, the reciprocal of that singular value will be very large and have a very large effect on w; but when that singular value is exactly zero, it has no effect on w! So when we have a really tiny singular value, should we pretend it is zero? Ridge regression implements this policy to some degree; review Discussion Worksheet 12 for details.]

BETTER GENERALIZATION FOR NEURAL NETWORKS

[Classic methods for preventing overfitting, such as subset selection, ℓ_2 regularization, and ensembles of learners, sometimes help neural networks to generalize better to points they haven't been trained on.]

(1) Get more data. [This is the best method. Andrej Karpathy writes that "It is a very common mistake to spend a lot of engineering cycles trying to squeeze juice out of a small dataset when you could instead be collecting more data."]

(2) Data augmentation. Augment data set with modified versions of training points.



augmentation.pdf, (Bishop, Figure 9.1) [Examples of data augmentation applied to an original image (a). (b) Reflection. (c) Scaling. (d) Translation. (e) Rotation. (f) Changing brightness and contrast. (g) Added noise. (h) Color shift.]

[You can see that these augmentations do not change the fact that the image should be classified as a cat.]

pixmix.pdf, (Hendrycks et al., "PixMix", 2022) [More varieties of data augmentation.]

[Hendrycks et al. note that "For state-of-the-art models, data augmentation can improve clean accuracy [on the test set] comparably to a 10× increase in model size. Further, data augmentation can improve out-of-distribution robustness [on images from a distribution different than the training set] comparably to a 1,000× increase in labeled data."]

[One point they make is that while adding Gaussian noise is one augmentation that helps improve generalization to new images, it's even more effective to add artifacts that stimulate hidden units, such as the hidden units that detect edges in an image. So their augmentation methods mix images with other images that introduce spurious structure, not just Gaussian noise.]

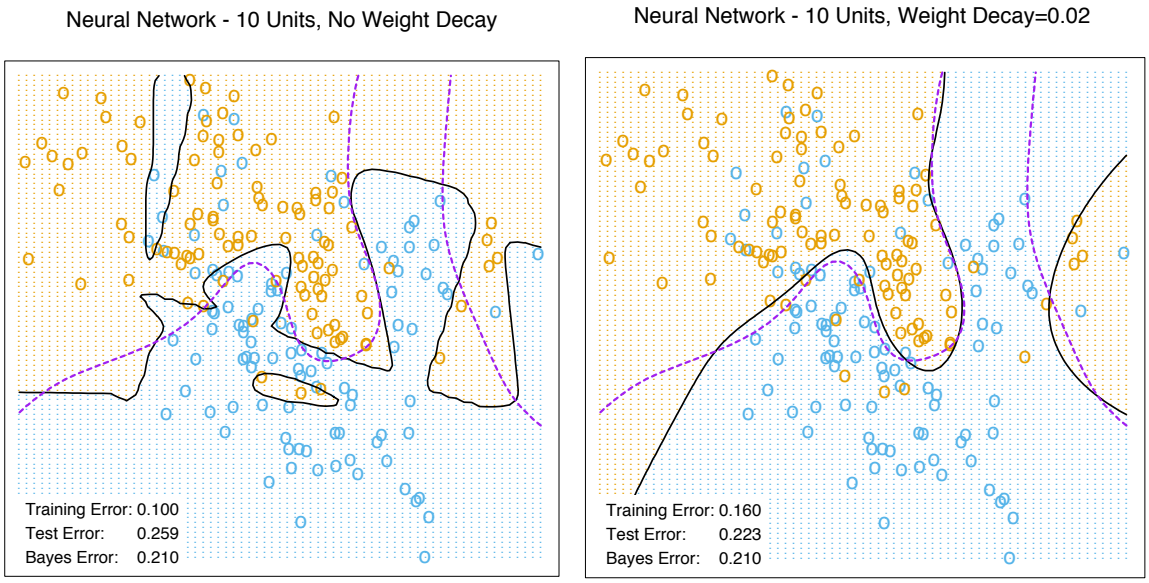
(3) Subset selection. [Recall Lecture 13.]

(4) ℓ_2 regularization, aka weight decay.
Add $\lambda \|w\|^2$ to the cost/loss fn, where w is vector of all weights in network.
[w includes all the weights in all the weight matrices, rewritten as a vector.]

[We regularize for the same reason we do it in ridge regression: we suspect that overly large weights are spurious.]

[With a neural network, it's not clear whether penalizing the bias terms is bad or good. Penalizing the bias terms has the effect, potentially positive, of drawing each ReLU or sigmoid unit closer to the center of its nonlinear operating region. I would suggest to try both ways and use validation to decide whether you should penalize the bias terms or not. Also, you could try using a different hyperparameter for the bias terms than the λ you use for the other weights.]

Effect: step $\Delta w_i = -\epsilon \dfrac{\partial J}{\partial w_i}$ has extra term $-2\epsilon\lambda\, w_i$
Weight w_i decays by factor $1 - 2\epsilon\lambda$ if not reinforced by training.



weightdecayoff.pdf, weightdecayon.pdf (ESL, Figure 11.4) Write "10 hidden units + softmax + cross-entropy loss." [Examples of 2D classification without (left) and with (right) weight decay. Observe that in the second example, the decision boundary (black) better approximates the Bayes optimal boundary (dashed purple curve).]

[AlexNet is a famous example of a network that used both ℓ_2 regularization and momentum. Just add the ℓ_2 penalty to the cost function J and plug that cost function into the momentum algorithm from Lecture 18. AlexNet set $\lambda = 0.0005$ and the momentum decay term to $\beta = 0.9$. They adjusted ϵ manually throughout training.]

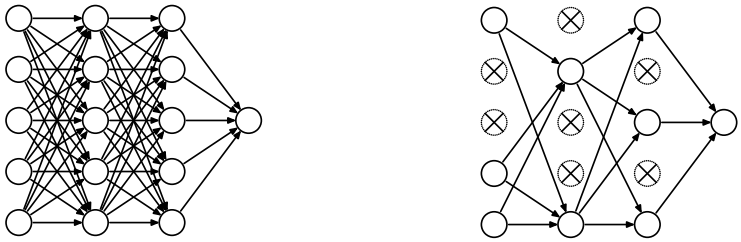
(5) Train for a very long time. [Andrej Karpathy: "I've often seen people tempted to stop the model training when the validation loss seems to be leveling off. In my experience networks keep training for an unintuitively long time. One time I accidentally left a model training during the winter break and when I got back in January it was SOTA ("state of the art").[9]]

[9] http://karpathy.github.io/2019/04/25/recipe/

(6) Ensemble of neural nets. Random initial weights + SGD + (optionally) bagging.
[Ensembles work well for neural nets, reportedly improving test accuracy by 2–3%. Random initial weights and random minibatches ensure that each neural net finds a different local minimum. If you have finite training data, validate to see if bagging helps or not. Obviously, ensembles of neural nets are very slow.]

For speed, sometimes the ensembles share the same early layers.
[Then only the last layers of each neural network are trained separately.]

(7) Dropout emulates an ensemble in one network.



dropout1.pdf, dropout2.pdf

During training, temporarily disable a random subset of the units, along with all edges in and out.
- No forward signal, no weight updates for edges in or out of disabled unit.
- Disable each hidden unit with probability (typically) 0.5.
- Disable each input unit with probability (typically) 0.2.
- Disable a **different** random subset for each SGD minibatch.

After training, before testing: enable all units. If units in a layer were disabled with probability p, multiply all edge weights out of that layer by p.
[When we disabled units, the edge weights had to grow large to make up for their disabled neighbors. At test time, all units and edges are enabled, so we have to reduce the weights to compensate.]

[Dropout gives an effect similar to averaging over multiple neural networks, but it's faster to train. Dropout usually gives better generalization than ℓ_2 regularization. Geoff Hinton and his co-authors give an example where they trained a network to classify MNIST digits. Their network without dropout had a 1.6% test error; it improved to 1.3% with dropout on the hidden units only; and further improved to 1.1% with dropout on the input units too.]

[Recall Karl Lashley's rat experiments, where he tried to make rats forget how to run a maze by introducing lesions in their cerebral cortexes, and it didn't work. He concluded that the knowledge is distributed throughout their brains, not localized in one place. Dropout is a way to force neural networks to distribute knowledge throughout the weights.]

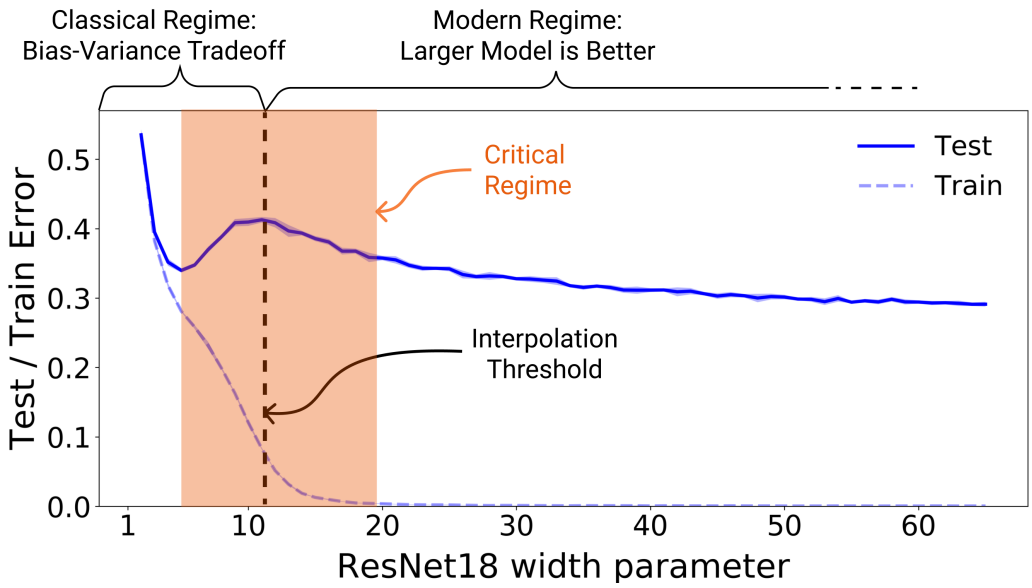
Double Descent

[Early neural network researchers sometimes struggled with their networks falling into bad local minima and failing to achieve low training errors. But with experience and greater computational power, we've discovered that these problems can usually be solved simply by adding more units to every hidden layer. We call this "making the network wider." Sometimes you also have to add more layers. But if your layers are wide enough, and there are enough of them, a well-designed neural network can typically output exactly the correct label for every training point, which implies that you're at a *global* minimum of the cost function.]

Hidden layers are wide enough + numerous enough ⇒ network output can interpolate the label ($\hat{y} = y$) for every training pt ⇒ find global minimum of cost fn.

[I have pointed out that if you use sigmoid or softmax output units, you can't set the labels to exactly 1 or 0 and achieve interpolation, as sigmoid and softmax outputs are strictly between 0 and 1; but with labels like 0.1 and 0.9, interpolating the labels is a realistic goal! And the linear output units used for regression can interpolate arbitrary numbers. Bottom line: if you fall into a bad local minimum, your network is too small.]

[One reason it took so long to make this discovery is that researchers believed that having too many weights in a neural network would cause overfitting. It turns out that's only half true. Empirically, we sometimes observe a phenomenon called "double descent," illustrated below.]



doubledescent.pdf (Nakkiran et al., "Deep Double Descent") [A classic double descent curve (solid blue) for test error. The horizontal axis indicates the number of units in each hidden layer of a residual neural network used for image recognition, and the vertical axis measures the the test error (solid curve) and training error (dashed curve).]

[Consider the solid blue curve, showing the test error as the width of a network increases. The horizontal axis is the number of units per hidden layer. As that number increases, at left the test error exhibits the classic U-shaped bias-variance "tradeoff." But when we pass the point where the network is interpolating the labels and continue to add more weights, we sometimes see a second "descent," where the test error starts to decrease again and ultimately gets even lower than before! The peak in the middle of the curve tends to be larger when there is more noise in the labels. Observe that the test error continues to fall even after the training error is zero. The takeaway is, "**bigger models are often better.**"]

[The currently accepted explanation for double descent, per Nakkiran et al., is that "at the interpolation threshold . . . the model is just barely able to fit the training data; forcing it to fit even slightly-noisy or mis-specified labels will destroy its global structure, and result in high test error. However for over-parameterized models, there are many interpolating models that fit the training set, and SGD is able to find one that 'absorbs' the noise while still performing well on the distribution."]

[Double descent has also been observed in decision trees and even in linear regression where we add random features to the training points (thereby adding more weights to the linear regression model).]