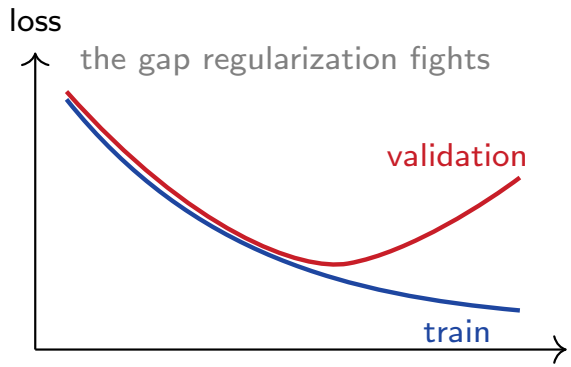
Regularizing Neural Networks

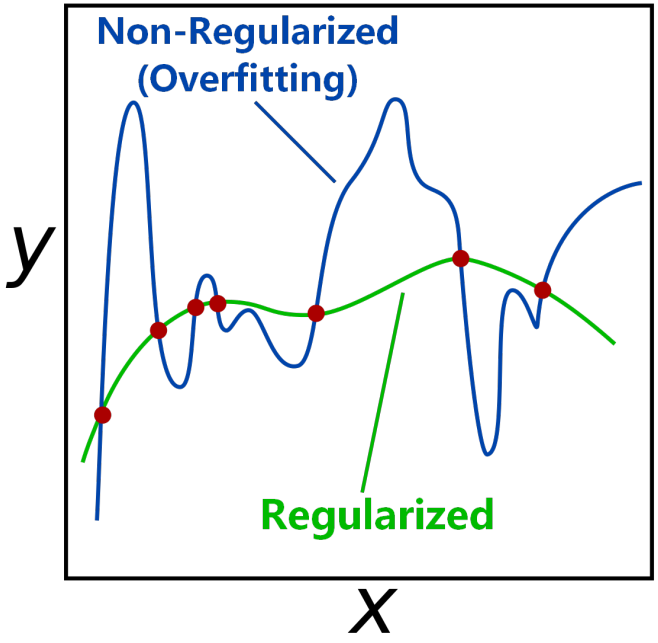
Weight decay, early stopping, dropout, and augmentation

This deck embeds the LMU *Introduction to Deep Learning* regularization chunks in full (basic regularization → early stopping → ensembles/dropout/augmentation), with our own notes sprinkled in.



Deep Learning

Basic Regularization



Learning goals

- Regularized cost functions
- Norm penalties
- Weight decay
- Equivalence with constrained optimization

REGULARIZATION

- Any technique that is designed to reduce the test error possibly at the expense of increased training error can be considered a form of regularization.
- Regularization is important in DL because NNs can have extremely high capacity (millions of parameters) and are thus prone to overfitting.

REVISION: REGULARIZED RISK MINIMIZATION

- The goal of regularized risk minimization is to penalize the complexity of the model to minimize the chances of overfitting.
- By adding a parameter norm penalty term $J(\boldsymbol{\theta})$ to the empirical risk $\mathcal{R}_{\text{emp}}(\boldsymbol{\theta})$ we obtain a regularized cost function:

$$\mathcal{R}_{\text{reg}}(\boldsymbol{\theta}) = \mathcal{R}_{\text{emp}}(\boldsymbol{\theta}) + \lambda J(\boldsymbol{\theta})$$

  with hyperparamater $\lambda \in [0, \infty)$, that weights the penalty term, relative to the unconstrained objective function $\mathcal{R}_{\text{emp}}(\boldsymbol{\theta})$.
- Therefore, instead of pure **empirical risk minimization**, we add a penalty for complex (read: large) parameters $\boldsymbol{\theta}$.
- Declaring $\lambda = 0$ obviously results in no penalization.
- We can choose between different parameter norm penalties $J(\boldsymbol{\theta})$.
- In general, we do not penalize the bias.

L2-REGULARIZATION / WEIGHT DECAY

Let us optimize the L2-regularized risk of a model $f(\mathbf{x} \mid \boldsymbol{\theta})$

$$\min_{\boldsymbol{\theta}} \mathcal{R}_{\text{reg}}(\boldsymbol{\theta}) = \min_{\boldsymbol{\theta}} \mathcal{R}_{\text{emp}}(\boldsymbol{\theta}) + \frac{\lambda}{2} \|\boldsymbol{\theta}\|_2^2$$

by gradient descent. The gradient is

$$\nabla \mathcal{R}_{\text{reg}}(\boldsymbol{\theta}) = \nabla \mathcal{R}_{\text{emp}}(\boldsymbol{\theta}) + \lambda \boldsymbol{\theta}.$$

We iteratively update $\boldsymbol{\theta}$ by step size α times the negative gradient

$$\boldsymbol{\theta}^{[t+1]} = \boldsymbol{\theta}^{[t]} - \alpha \left(\nabla \mathcal{R}_{\text{emp}}(\boldsymbol{\theta}) + \lambda \boldsymbol{\theta}^{[t]} \right) = \boldsymbol{\theta}^{[t]}(1 - \alpha\lambda) - \alpha \nabla \mathcal{R}_{\text{emp}}(\boldsymbol{\theta})$$

$\rightarrow$ The term $\lambda \boldsymbol{\theta}^{[t]}$ causes the parameter (**weight**) to **decay** in proportion to its size (which gives rise to the name).

EQUIVALENCE TO CONSTRAINED OPTIMIZATION

Norm penalties can be interpreted as imposing a constraint on the weights. One can show that

$$\arg\min_{\theta} \mathcal{R}_{\text{emp}}(\boldsymbol{\theta}) + \lambda J(\boldsymbol{\theta})$$

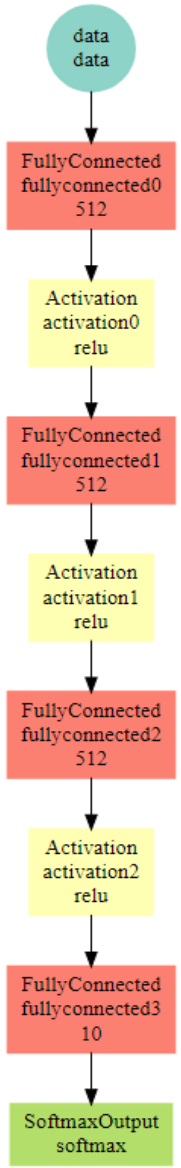
is equvilalent to

$$\arg\min_{\boldsymbol{\theta}} \mathcal{R}_{\text{emp}}(\boldsymbol{\theta})$$
$$\text{subject to} \quad J(\boldsymbol{\theta}) \leq k$$

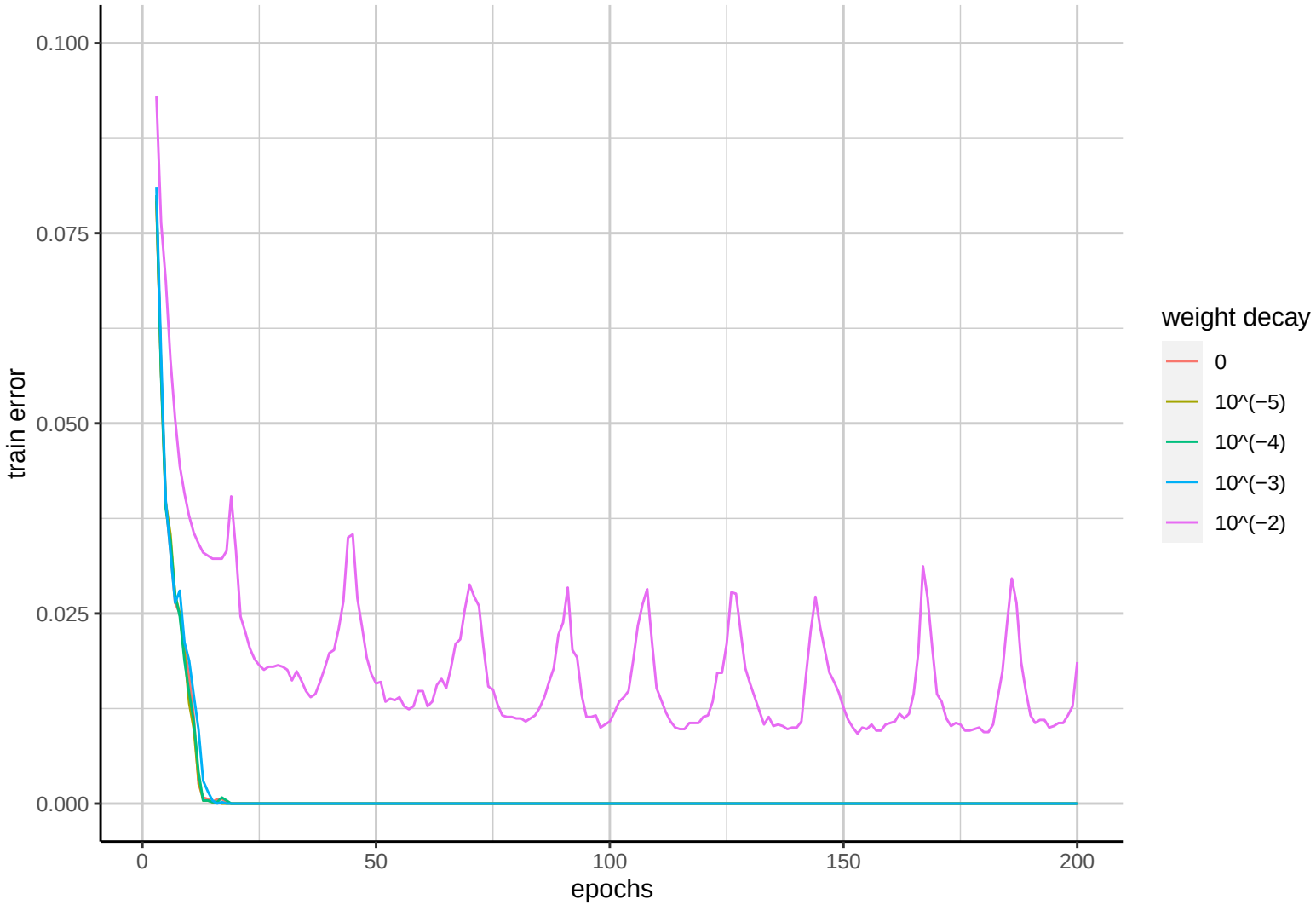
for some value k that depends on λ the nature of $\mathcal{R}_{\text{emp}}(\boldsymbol{\theta})$ (Goodfellow et al., 2016)

EXAMPLE: WEIGHT DECAY

- We fit the huge neural network on the right side on a smaller fraction of MNIST (5000 train and 1000 test observations)
- Weight decay: $\lambda \in (10^{-2}, 10^{-3}, 10^{-4}, 10^{-5}, 0)$

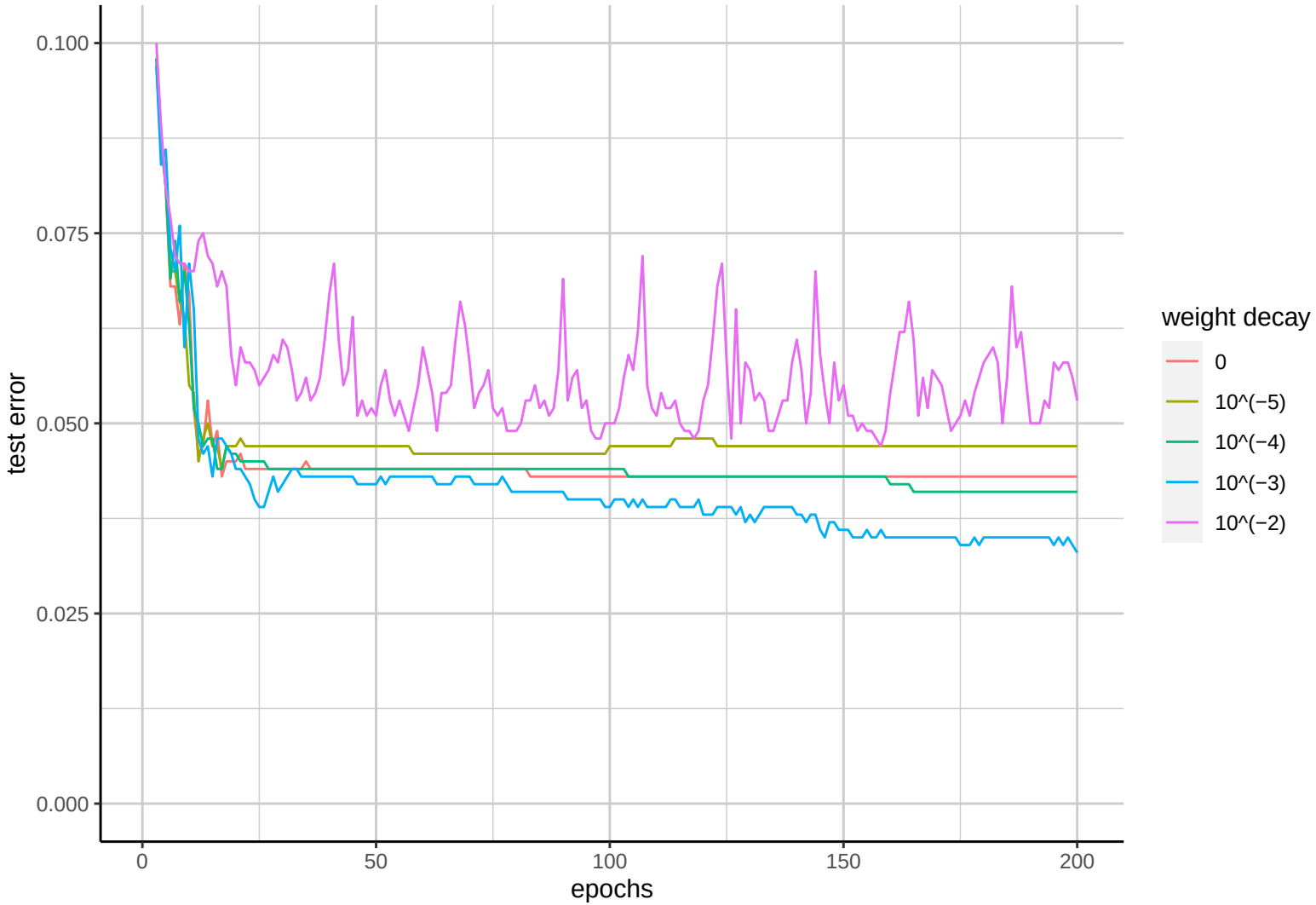


EXAMPLE: WEIGHT DECAY / 2



A high weight decay of 10^{-2} leads to a high error on the training data.

EXAMPLE: WEIGHT DECAY / 3



Second strongest weight decay leads to the best result on the test data.

TENSORFLOW PLAYGROUND

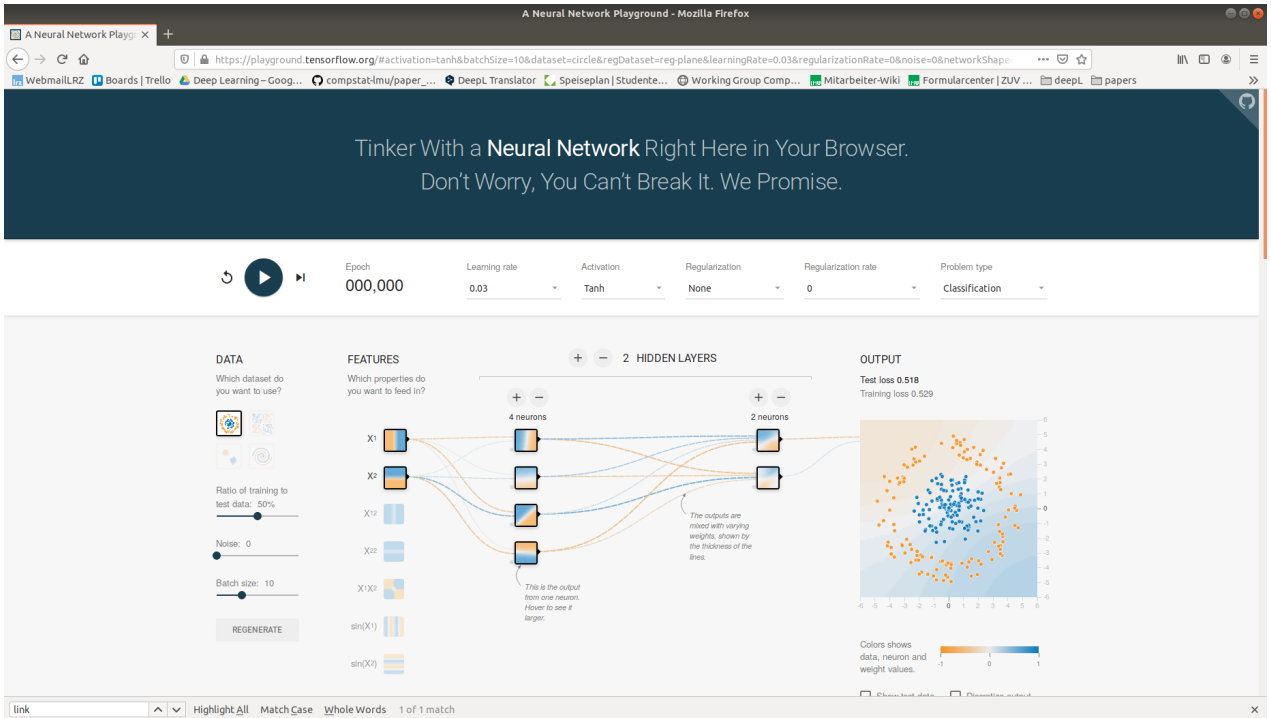


Figure: `https://playground.tensorflow.org/`, (Carter)

TENSORFLOW PLAYGROUND - EXERCISE

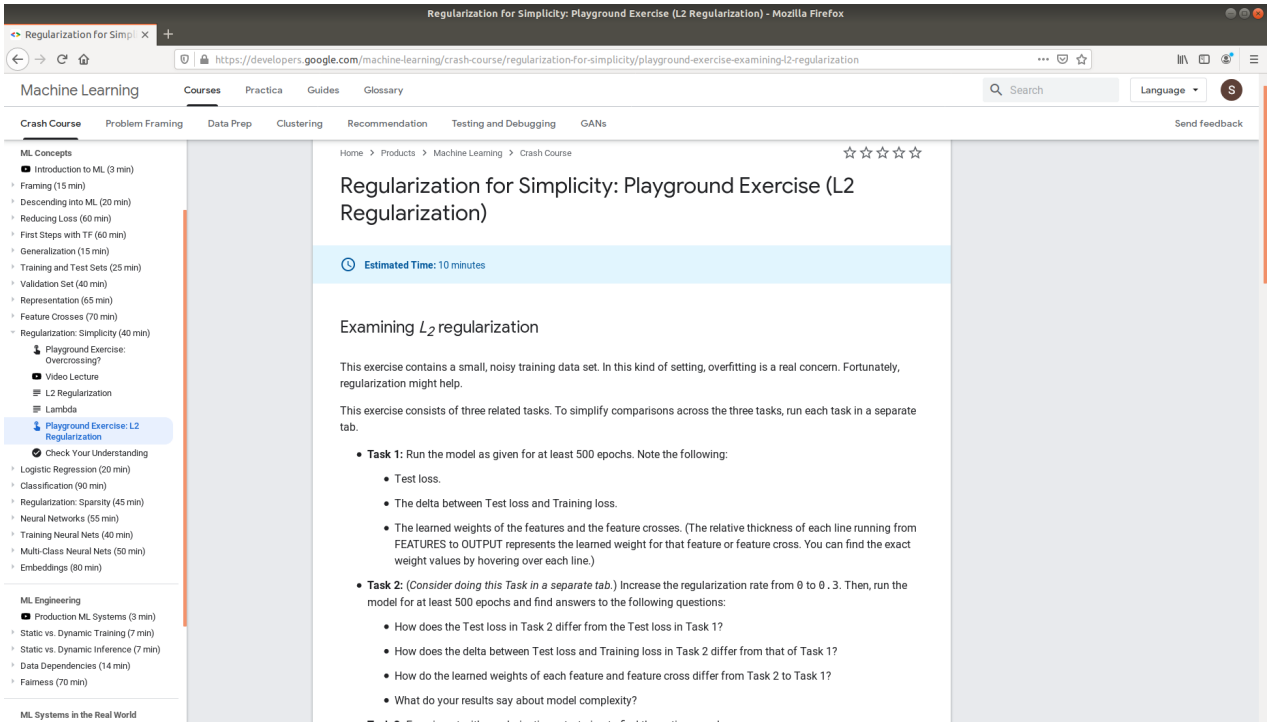


Figure: `https://developers.google.com/machine-learning/crash-course/regularization-for-simplicity/playground-exercise-examining-l2-regularization` (Google)

REFERENCES


Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning*. MIT Press.


MathWorks. (n.d.). *Regularization*. MATLAB & Simulink. `https://de.mathworks.com/discovery/regularization.html`


Carter, D. S. and S. (n.d.). *Tensorflow - Neural Network Playground. A Neural Network Playground*. `https://playground.tensorflow.org`


Google. (n.d.). *Regularization for Simplicity: Playground Exercise (L2 Regularization)*. Google. `https://developers.google.com/machine-learning/crash-course/regularization-for-simplicity/playground-exercise-examining-l2-regularization?hl=de`

Weight decay is Ridge – you've already met it

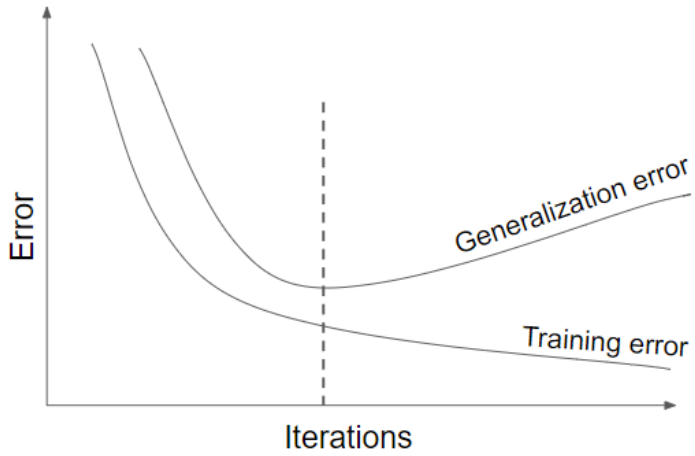
The penalty LMU just added, $R_{\text{reg}} = R_{\text{emp}} + \frac{\lambda}{2}\|\boldsymbol{\theta}\|_2^2$, is **exactly Ridge regression** from the regression chapter – now called *weight decay* because the update $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta}(1 - \alpha\lambda) - \alpha\nabla R_{\text{emp}}$ shrinks every weight a little each step.

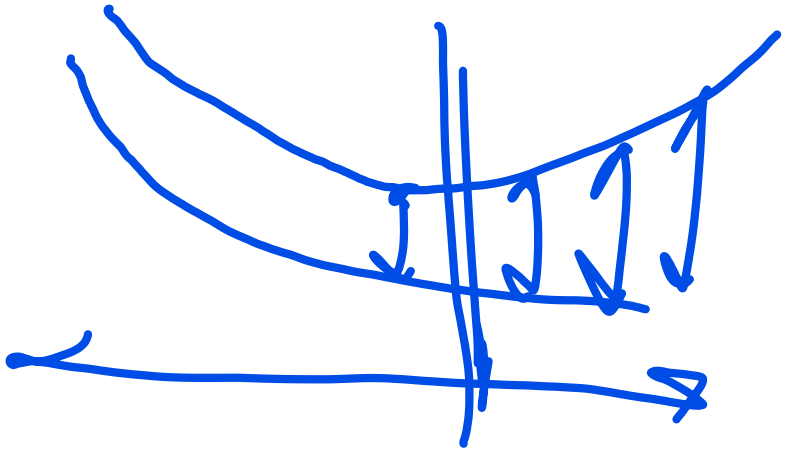
- Smaller weights $\Rightarrow$ a smoother function $\Rightarrow$ less overfitting – the same story as Ridge, now applied to a million weights.
- In PyTorch it is **one argument**: `SGD(..., weight_decay=1e-4)`. With Adam, use `AdamW` instead – the optimization deck shows why.

The network is still "a stack of regressions", so the *regressions'* regularizer still applies. Nothing here is NN-specific – only the scale.

Deep Learning

Early Stopping





Learning goals

- Know how early stopping works
- Understand how early stopping acts as a regularizer
- Know early stopping imitates *L2* regularization in some cases

EARLY STOPPING

- When training with an iterative optimizer such as SGD, it is commonly the case that, after a certain number of iterations, generalization error begins to increase even though training error continues to decrease.
- **Early stopping** refers to stopping the algorithm early before the generalization error increases.

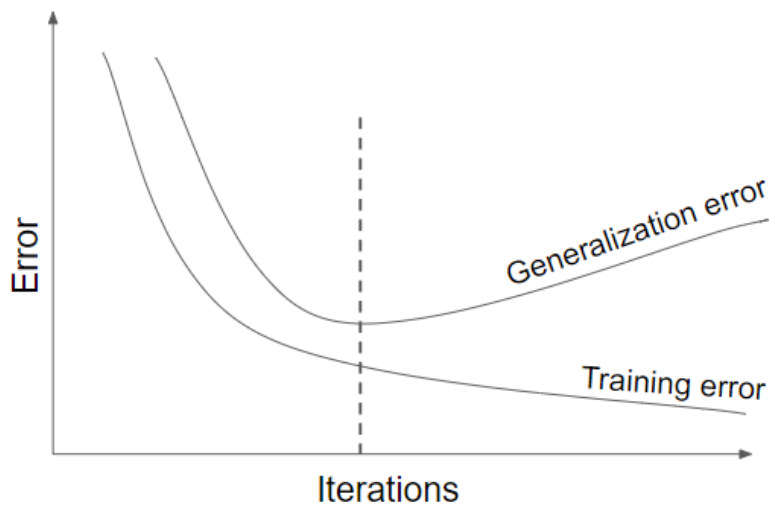


Figure: After a certain number of iterations, the algorithm begins to overfit.

EARLY STOPPING / 2

How early stopping works:

1. Split training data $\mathcal{D}_{\text{train}}$ into $\mathcal{D}_{\text{subtrain}}$ and $\mathcal{D}_{\text{val}}$ (e.g. with a ratio of 2:1).
2. Train on $\mathcal{D}_{\text{subtrain}}$ and evaluate model using the validation set $\mathcal{D}_{\text{val}}$.
3. Stop training when validation error stops decreasing (after a range of “patience” steps).
4. Use parameters of the previous step for the actual model.

More sophisticated forms also apply cross-validation.

EARLY STOPPING AND *L2*

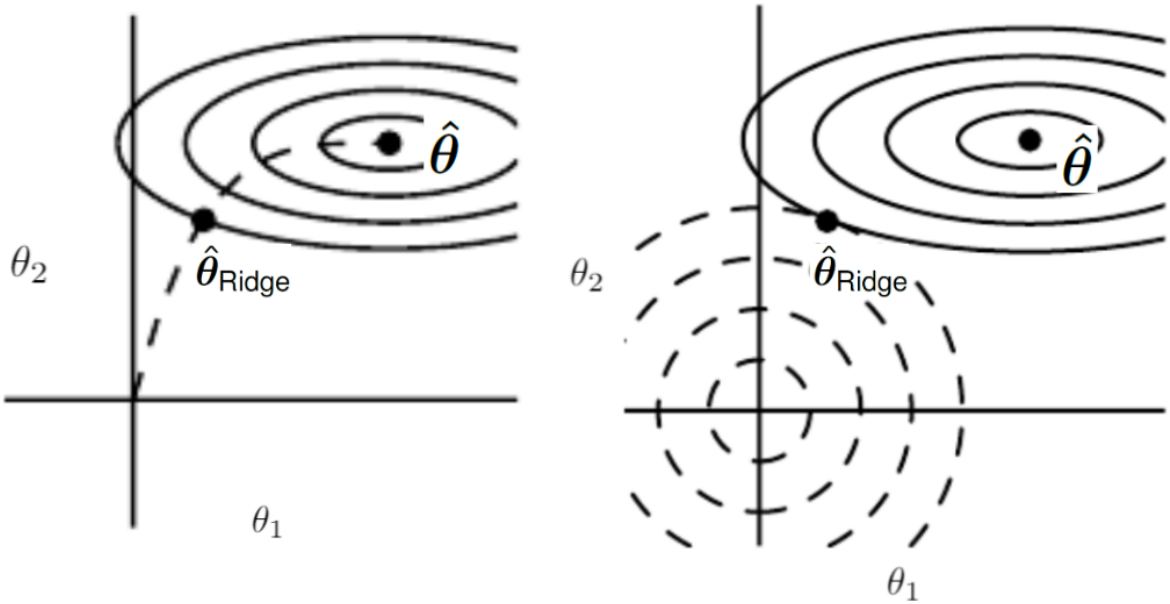
Strengths	Weaknesses
Effective and simple	Periodical evaluation of validation error
Applicable to almost any model without adjustment	Temporary copy of $\boldsymbol{\theta}$ (we have to save the whole model each time validation error improves)
Combinable with other regularization methods	Less data for training $\rightarrow$ include $\mathcal{D}_{\text{val}}$ afterwards

- For simple case of LM with squared loss and GD optim initialized at $\boldsymbol{\theta} = 0$: Early stopping has exact correspondence with *L2* regularization/WD: optimal early-stopping iter T_{stop} inversely proportional to λ scaled by step-size α

$$T_{\text{stop}} \approx \frac{1}{\alpha\lambda} \Leftrightarrow \lambda \approx \frac{1}{T_{\text{stop}}\alpha}$$

- Small λ (regu. $\downarrow$) $\Rightarrow$ large T_{stop} (complexity $\uparrow$) and vice versa

EARLY STOPPING AND *L2* / 2

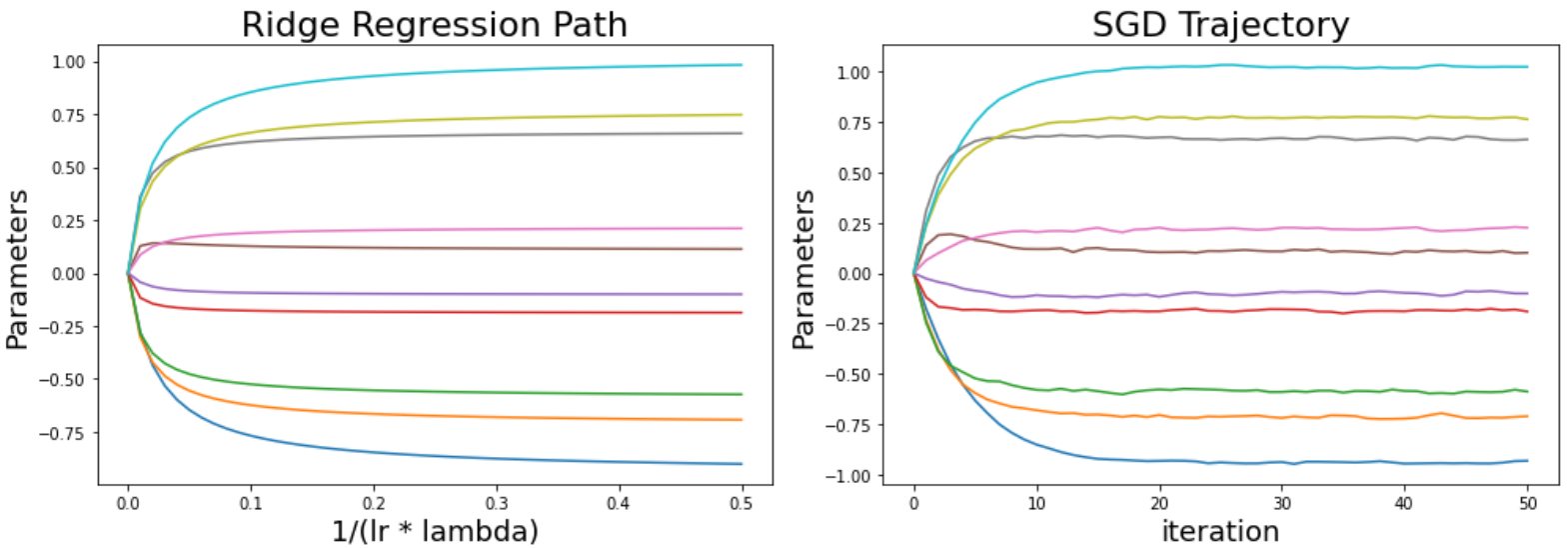


Goodfellow et al., 2016, p. 249 ff.

Figure: Effect of early stopping. *Left:* The solid lines indicate contours of the square loss objective. Dashed line indicates trajectory taken by GD initialized at origin. Instead of reaching minimizer $\hat{\theta}$, ES results in trajectory stopping earlier at $\hat{\boldsymbol{\theta}}_{\text{ridge}}$. *Right:* Effect of *L2* regularization. Dashed circles indicate contours of *L2* constraint which push minimizer of regularized cost closer to origin than minimizer of unregularized cost.

SGD TRAJECTORY AND *L2*

Solution paths for *L2* regularized linear model closely matches SGD trajectory of unregularized LM initialized at $\theta = 0$

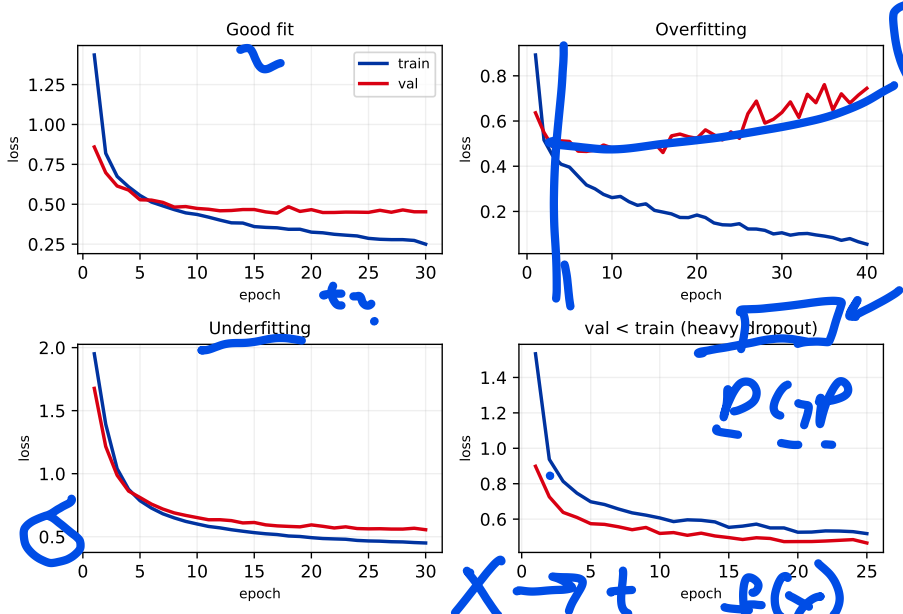


Ali et al., 2020

Caveat: Initialization at the origin is crucial for this equivalence to hold, which is almost never used in practice in ML/DL applications

Read your training curves

Early stopping needs a signal: plot **train and validation loss every epoch** – the single most useful diagnostic in all of training.



- **Overfitting:** train keeps falling, validation turns *up* – stop at the valley (that *is* early stopping).
- **Underfitting:** both stay high – too much regularization or too few epochs.
- **Good:** both fall, validation a little above train.

Diagnose *under- vs over-fitting from the curves first*, then reach for the matching knob – don't turn all of them at once.

Predict: can a network learn labels that mean nothing?

Take a standard image network. Replace **every** training label with a **random** class, then train as usual.

What happens to the training error?

And to the test error?

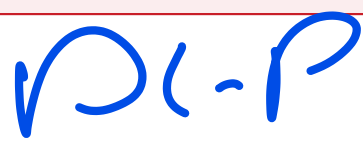
There is nothing to learn: the labels have no relation to the images. Commit to a guess before the next slide.

A network can memorize pure noise

Zhang et al. (ICLR 2017) took standard image networks and replaced every training label with a random one.

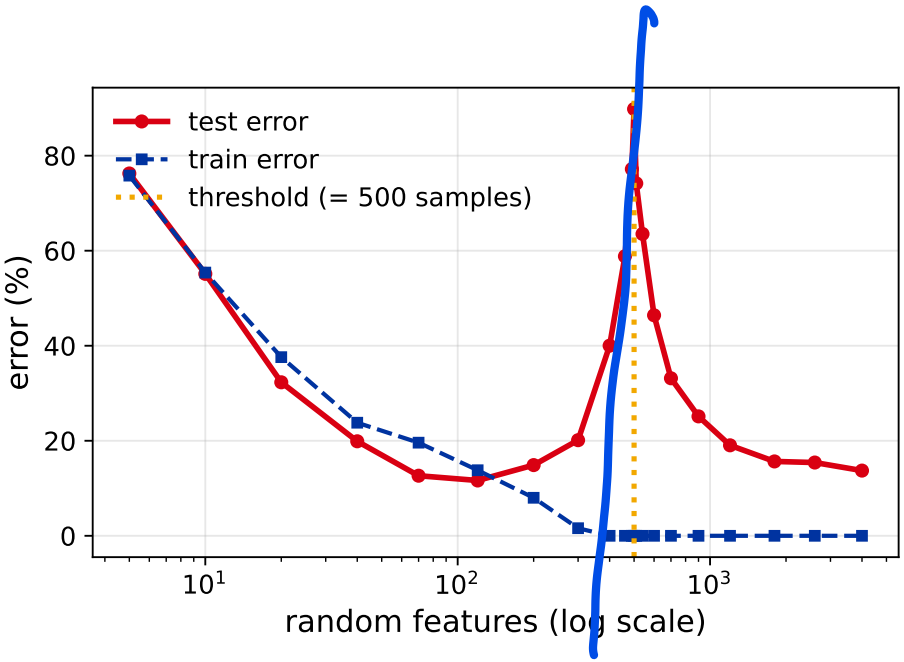
- The networks still reached **0 training error**. Test error was, of course, no better than chance.
- On ImageNet, one million random labels from 1000 classes: **95.2%** training accuracy.
- It still worked with **weight decay** switched on, and even with the images replaced by random pixels. Training took only a small constant factor longer.

Why it matters. Capacity alone cannot explain why these nets generalize on *real* labels: the same network, at the same size and with the same optimizer, memorizes noise just as happily. In the authors' words, explicit regularization "may improve generalization performance, but is neither necessary nor by itself sufficient".



The U-curve is not the whole story

Grow the model *past* the point where it already fits the training set perfectly. Does test error keep climbing?



No – it peaks, then falls. 500 digits, 15% corrupted labels; random ReLU features, least-squares readout.

- **11.6%** at 120 features;
- **89.8%** at 500 – the sample count;
- **13.7%** at 4000.

At the threshold essentially one fit exists, and it is bad. Past it many exist and the solver picks a smooth one.

Honest footnote. The second descent does not beat the first minimum (13.7% vs 11.6%) – capacity is not free. But "bigger always overfits" is not a law. *(Belkin et al., 2019.)*

Double descent in the wild

Our sweep was a toy. The phenomenon is general – and it is not only about model *size*:

- **Belkin et al. (2019, PNAS)** named the curve and showed it across model families. The random-feature setting we just ran is theirs.
- **Nakkiran et al. (ICLR 2020)**, *Deep Double Descent*, found it in real deep nets along three axes: **model-wise** (bigger), **epoch-wise** (train *longer* and test error dips, rises, then dips again), and **sample-wise** – there are regimes where **more training data makes it worse**.

What to take from it. Label noise makes the peak pronounced; clean, well-regularized, modestly sized models mostly stay on the classical U-curve, so this chapter's advice still holds. But "smaller, and stop early" is a *rule of thumb*, not a law – and this is part of why scaling very large models keeps paying off.

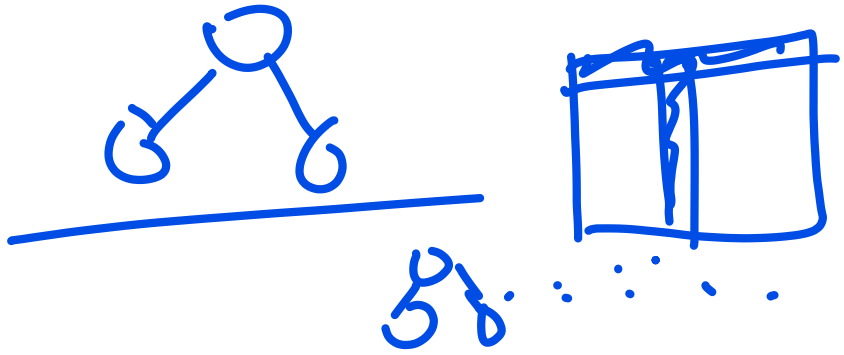
Bigger, predictably: scaling laws

Double descent showed bigger is not always worse. At the scale of language models the evidence goes further: bigger is **predictably better** – when data and compute grow with it.

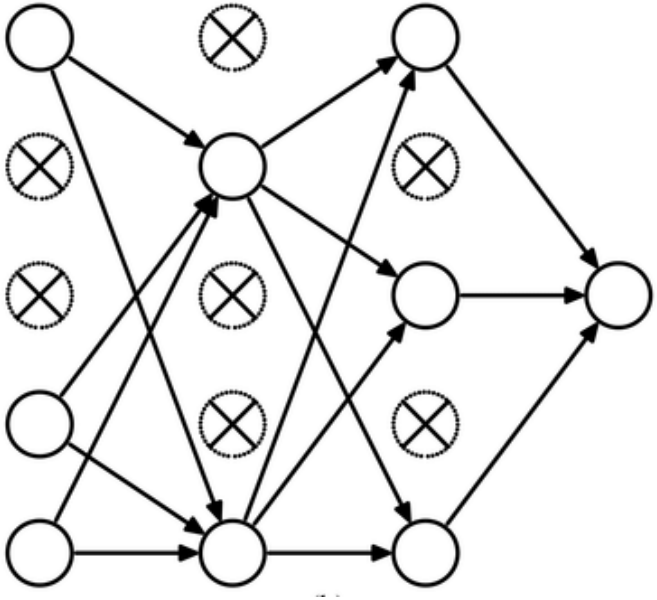
- **Kaplan et al. (2020)**: test loss falls as a smooth **power law** in model size, dataset size and training compute, with some trends spanning more than **seven orders of magnitude**.
- **Hoffmann et al. (2022)**, "Chinchilla": for a fixed compute budget, grow parameters and training tokens *together* – roughly **20 tokens per parameter**. Their 70B Chinchilla, trained on 1.4T tokens, beat the 280B Gopher.

Why it is here. With enough data, the overfitting this deck fights stops being the bottleneck, and model size becomes a knob you can plan with. Full treatment in the LLM chapter.

Deep Learning



Dropout and Augmentation



Learning goals

- Recap: Ensemble Methods
- Dropout
- Augmentation

Ensemble Methods

RECAP: ENSEMBLE METHODS

- Idea: Train **several models** separately, and **average their prediction** (i.e. perform **model averaging**).
- Intuition: This improves performance on test set, since different models will not make the same errors.
- Ensembles can be constructed in different ways, e.g.:
  - by combining completely different kind of models (using different learning algorithms and loss functions).
  - by **bagging**: train the same model on k datasets, constructed by sampling n samples from original dataset.
- Since training a neural network repeatedly on the same dataset results in different solutions (why?) it can even make sense to combine those.

RECAP: ENSEMBLE METHODS / 2

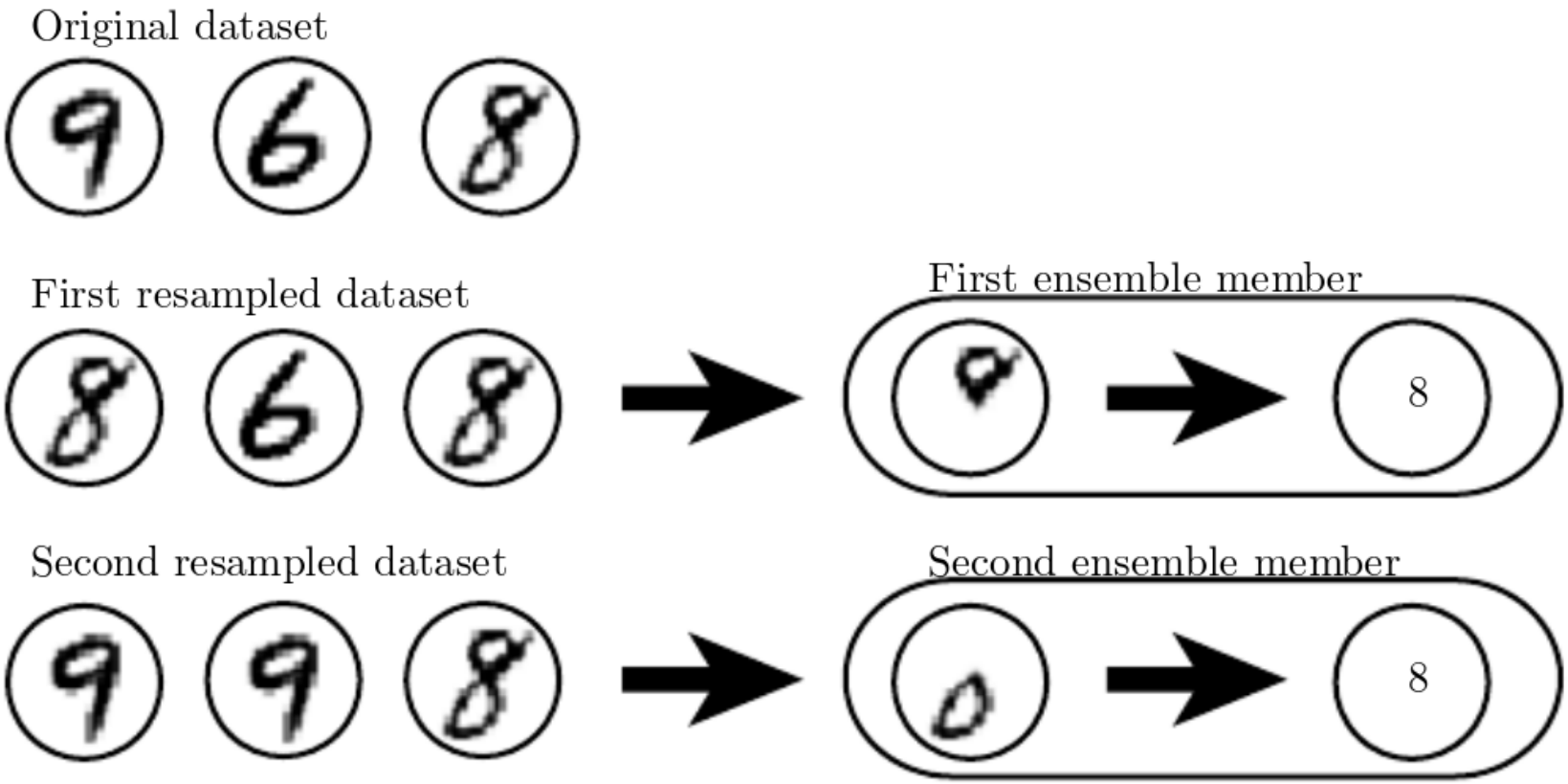
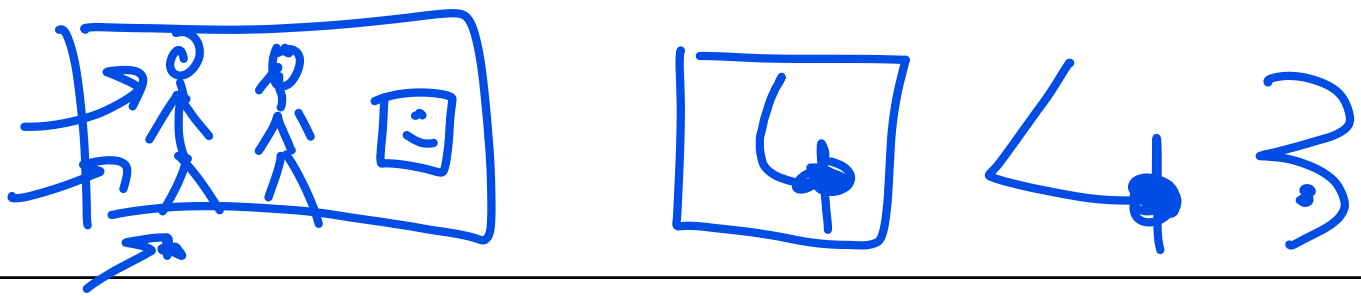


Figure: A cartoon description of bagging (Goodfellow et al., 2016)

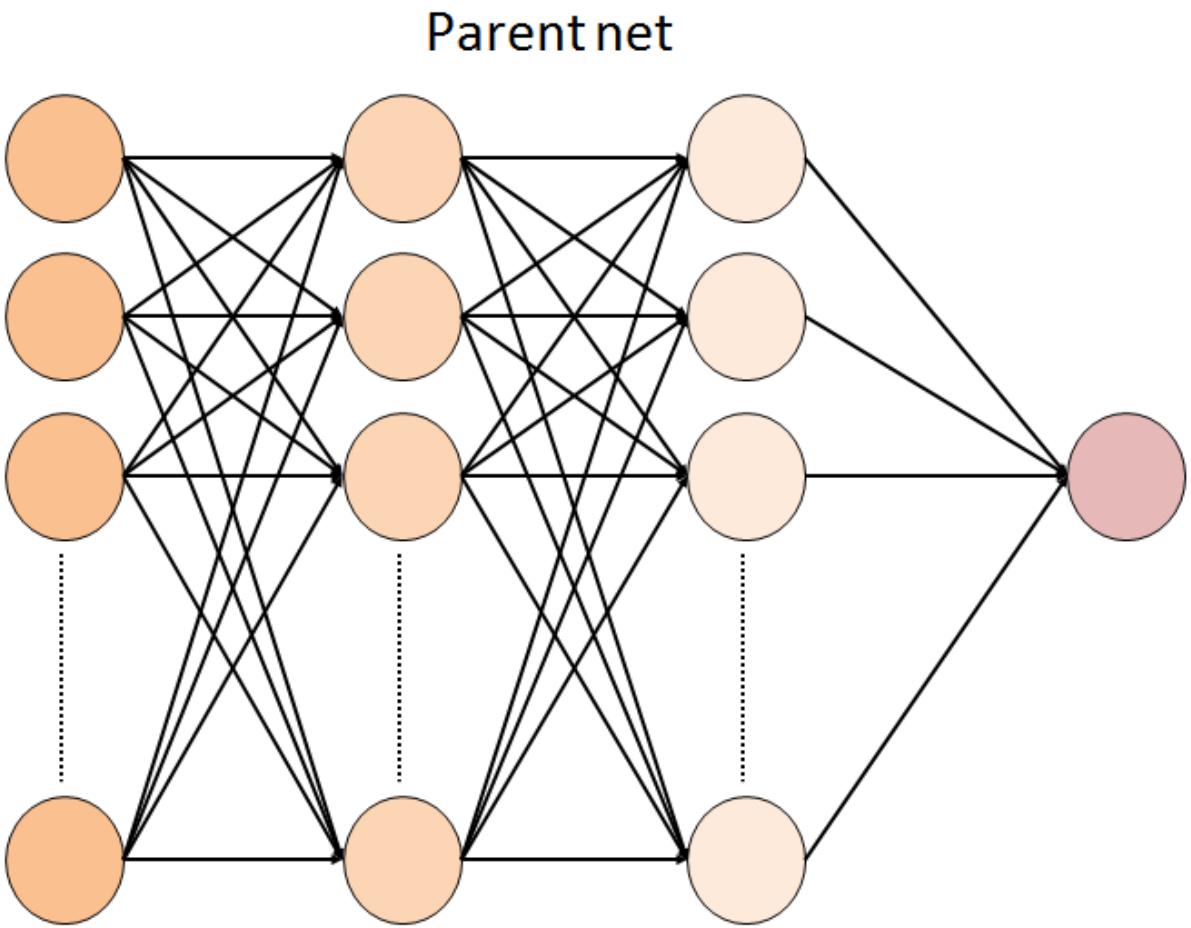
Dropout

DROPOUT

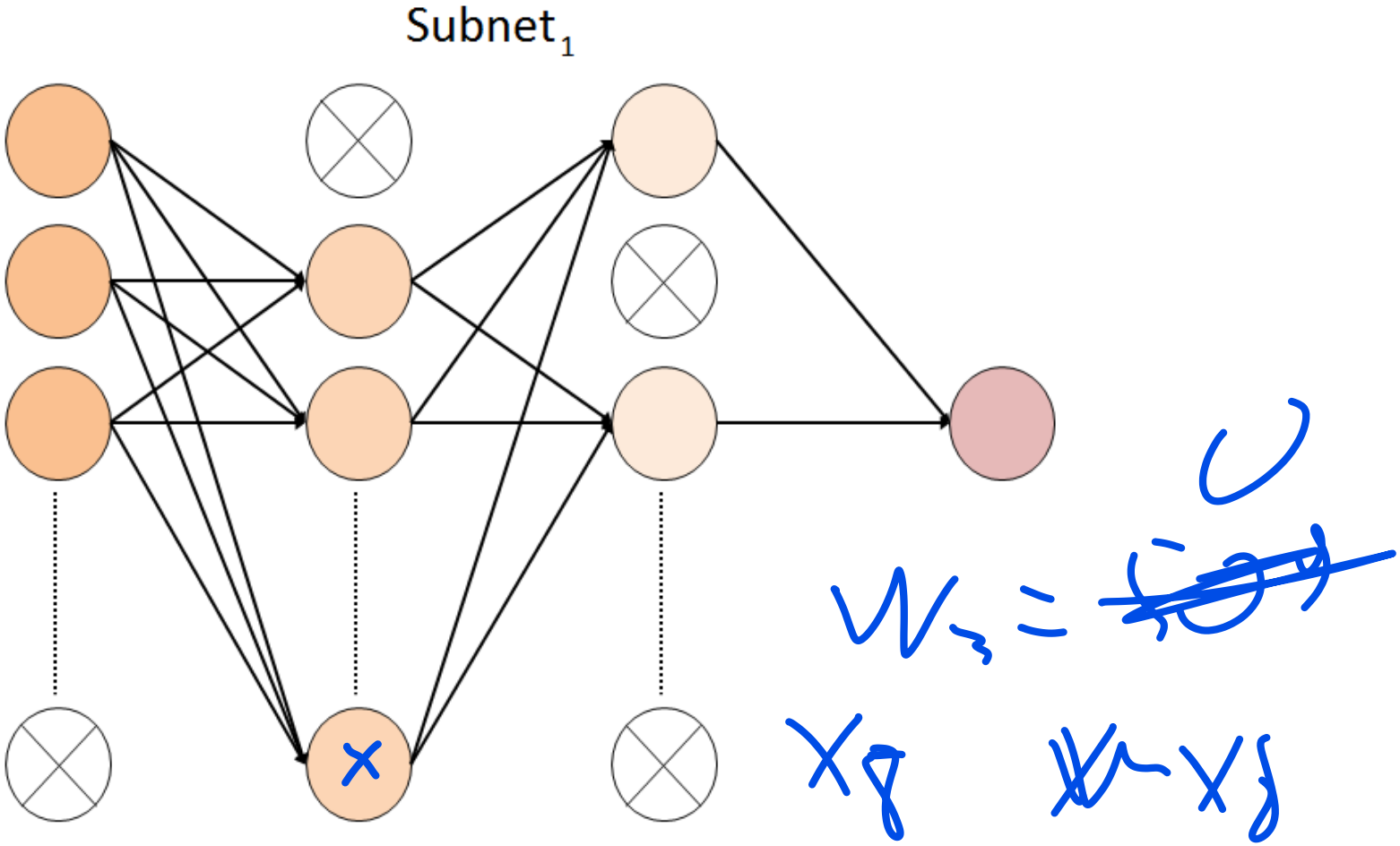
- Idea: reduce overfitting in neural networks by preventing complex co-adaptations of neurons.
- Method: during training, random subsets of the neurons are removed from the network (they are "dropped out").This is done by artificially setting the activations of those neurons to zero.
- Whether a given unit/neuron is dropped out or not is completely independent of the other units.
- If the network has N (input/hidden) units, applying dropout to these units can result in 2^N possible 'subnetworks'.
- Because these subnetworks are derived from the same 'parent' network, many of the weights are shared.
- Dropout can be seen as a form of "model averaging".



DROPOUT / 2

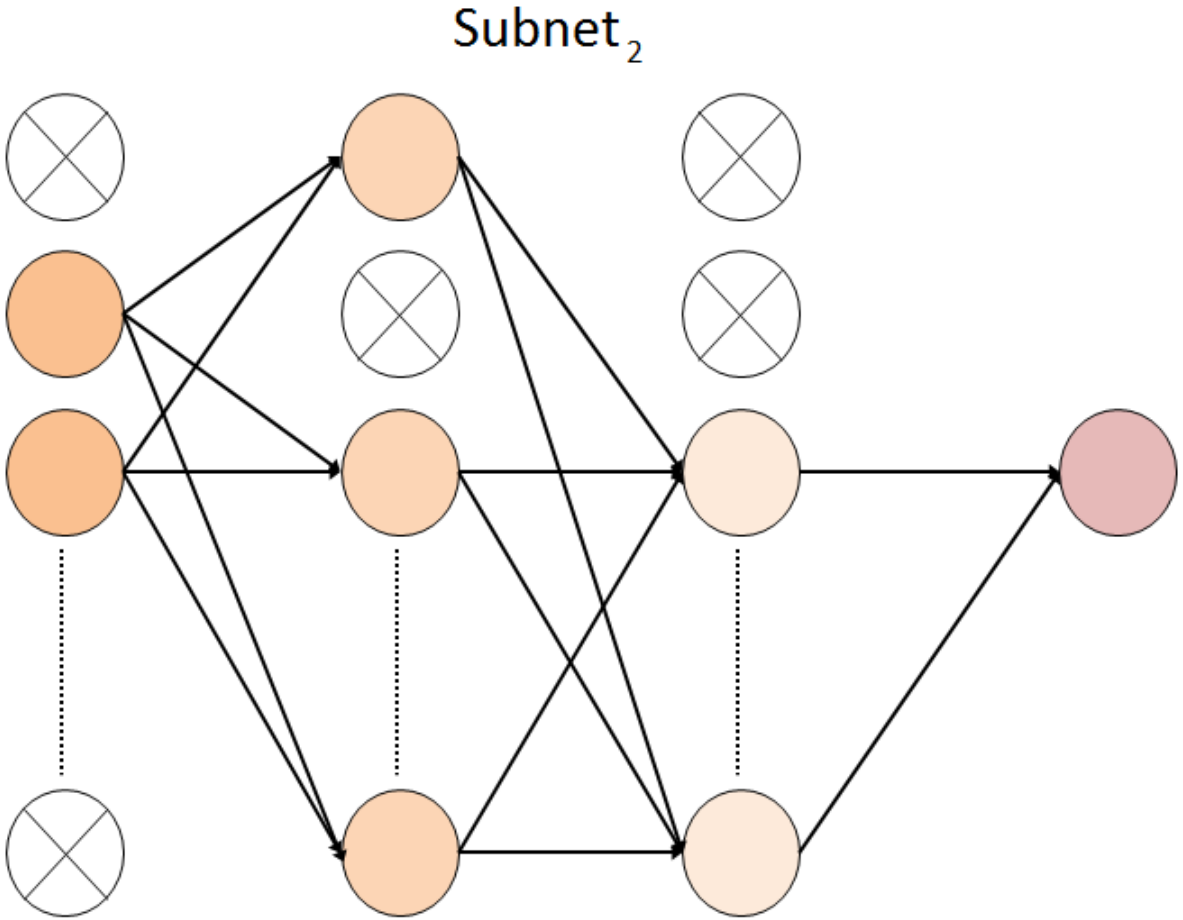


DROPOUT / 3



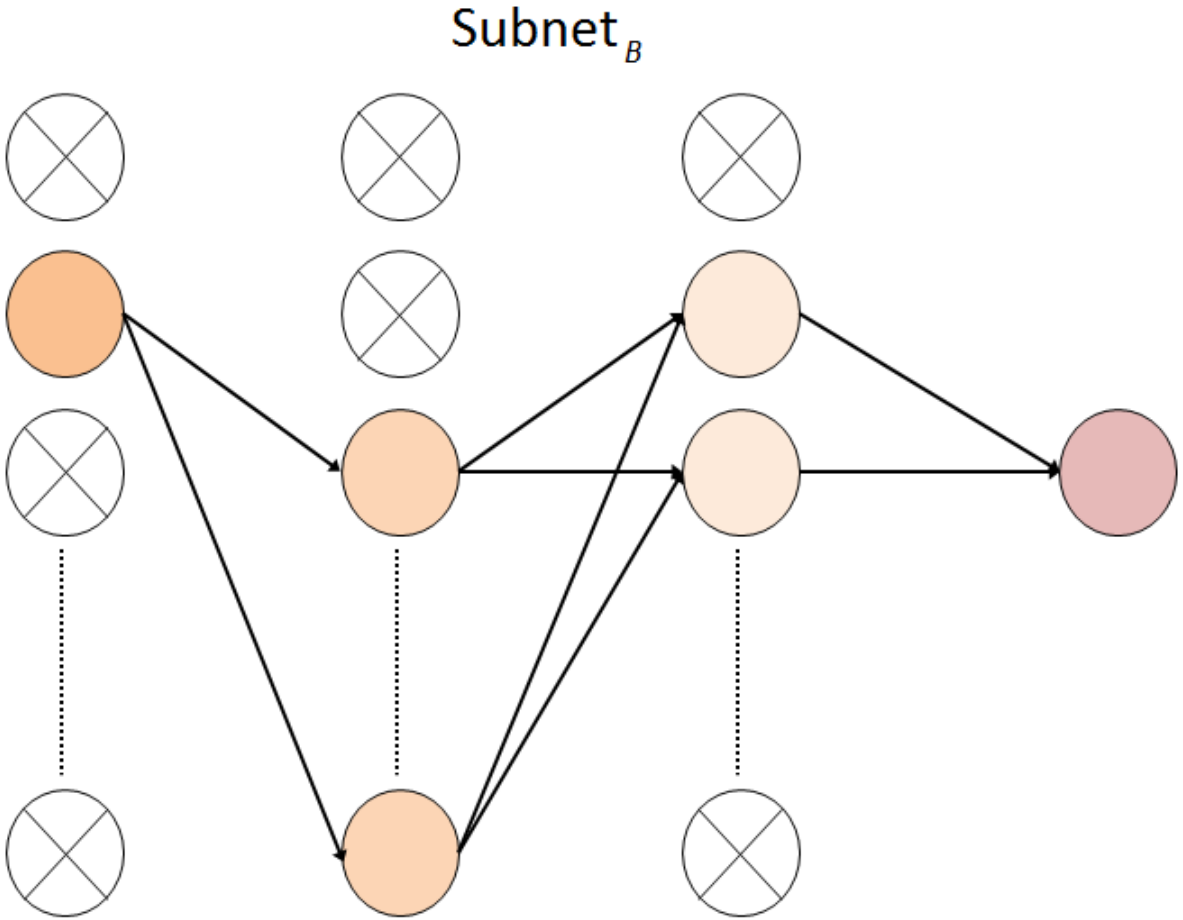
In each iteration, for each training example (in the forward pass), a different (random) subset of neurons are dropped out.

DROPOUT / 4



In each iteration, for each training example (in the forward pass), a different (random) subset of neurons are dropped out.

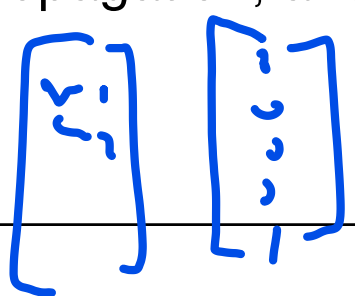
DROPOUT / 5



In each iteration, for each training example (in the forward pass), a different (random) subset of neurons are dropped out.

DROPOUT: ALGORITHM

- To train with dropout a minibatch-based learning algorithm such as stochastic gradient descent is used.
- For each training case in a minibatch, we randomly sample a binary vector/mask μ with one entry for each input or hidden unit in the network. The entries of μ are sampled independently from each other.
- The probability *p* of sampling a mask value of 0 (dropout) for one unit is a hyperparameter known as the 'dropout rate'.
- A typical value for the dropout rate is 0.2 for input units and 0.5 for hidden units.
- Each unit in the network is multiplied by the corresponding mask value resulting in a $subnet_{\mu}$.
- Forward propagation, backpropagation, and the learning update are run as usual.



DROPOUT: ALGORITHM / 2

Algorithm Training a (parent) neural network with dropout rate p

1: Define parent network and initialize weights
2: **for** each minibatch: **do**
3: **for** each training sample: **do**
4: Draw mask μ using p
5: Compute forward pass for $subnet_\mu$
6: **end for**
7: Update the weights of the (parent) network by performing a gradient descent step with weight decay
8: **end for**

$p = 0$

- The derivatives wrt. each parameter are averaged over the training cases in each mini-batch. Any training case which does not use a parameter contributes a gradient of zero for that parameter.

DROPOUT: WEIGHT SCALING

- The weights of the network will be larger than normal because of dropout. Therefore, to obtain a prediction at test time the weights must be first scaled by the chosen dropout rate.
- This means that if a unit (neuron) is retrained with probability p during training, the weight at test time of that unit is multiplied by p.

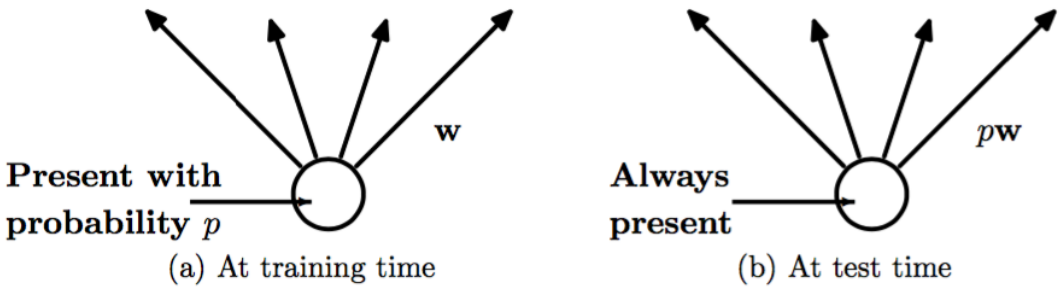


Figure: Training vs. Testing (Srivastava et. al., 2014)

DROPOUT: WEIGHT SCALING / 2

- Weight scaling ensures that the expected total input to a neuron/unit at test time is roughly the same as the expected total input to that unit at train time, even though many of the units at train time were missing on average
- Rescaling of the weights can also be performed at training time instead, after each weight update at the end of the mini-batch. This is sometimes called 'inverse dropout'. Keras and PyTorch deep learning libraries implement dropout in this way.

Dropout scaling, stated cleanly

LMU's dropout pages mix two meanings of p: on the algorithm pages p is the **drop** rate; on the weight-scaling pages (quoting Srivastava et al., 2014) p is the probability of **keeping** a unit. With p = drop rate, a unit is kept with probability $1 - p$, and:

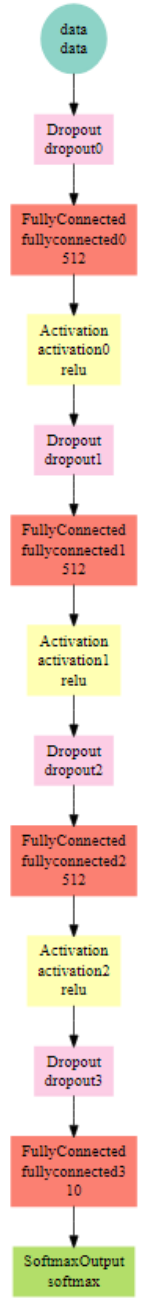
	training	**test time**
classic (Srivastava)	apply the mask	multiply outgoing weights by $1 - p$
inverted (PyTorch)	apply the mask, divide kept activations by $1 - p$	do nothing

`nn.Dropout(p=0.2)` on a vector of ones: the kept entries become $1/0.8 = 1.25$ in `train()` mode, and `eval()` returns the ones unchanged.

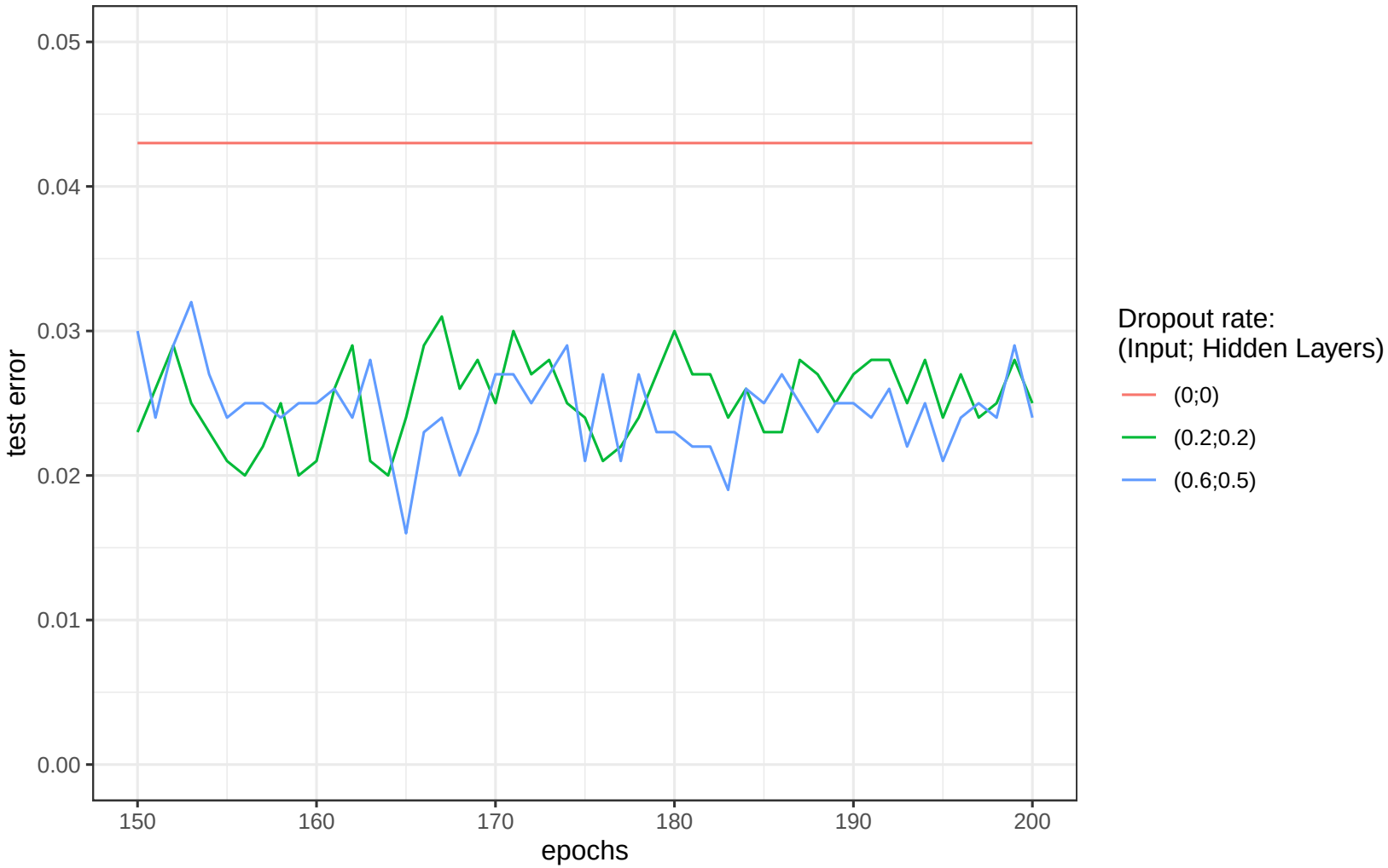
Both keep the *expected* input to each unit the same at train and test time. Inverted dropout scales the *activations* in the forward pass, not the weights after each update, so inference code needs no change at all.

DROPOUT: EXAMPLE

- To demonstrate how dropout can easily improve generalization we compute neural networks with the structure showed on the right.
- Each neural network we fit has different dropout probabilities, a tuple where one probability is for the input layer and one is for the hidden layers. We consider the tuples $(0; 0)$, $(0.2; 0.2)$ and $(0.6; 0.5)$.

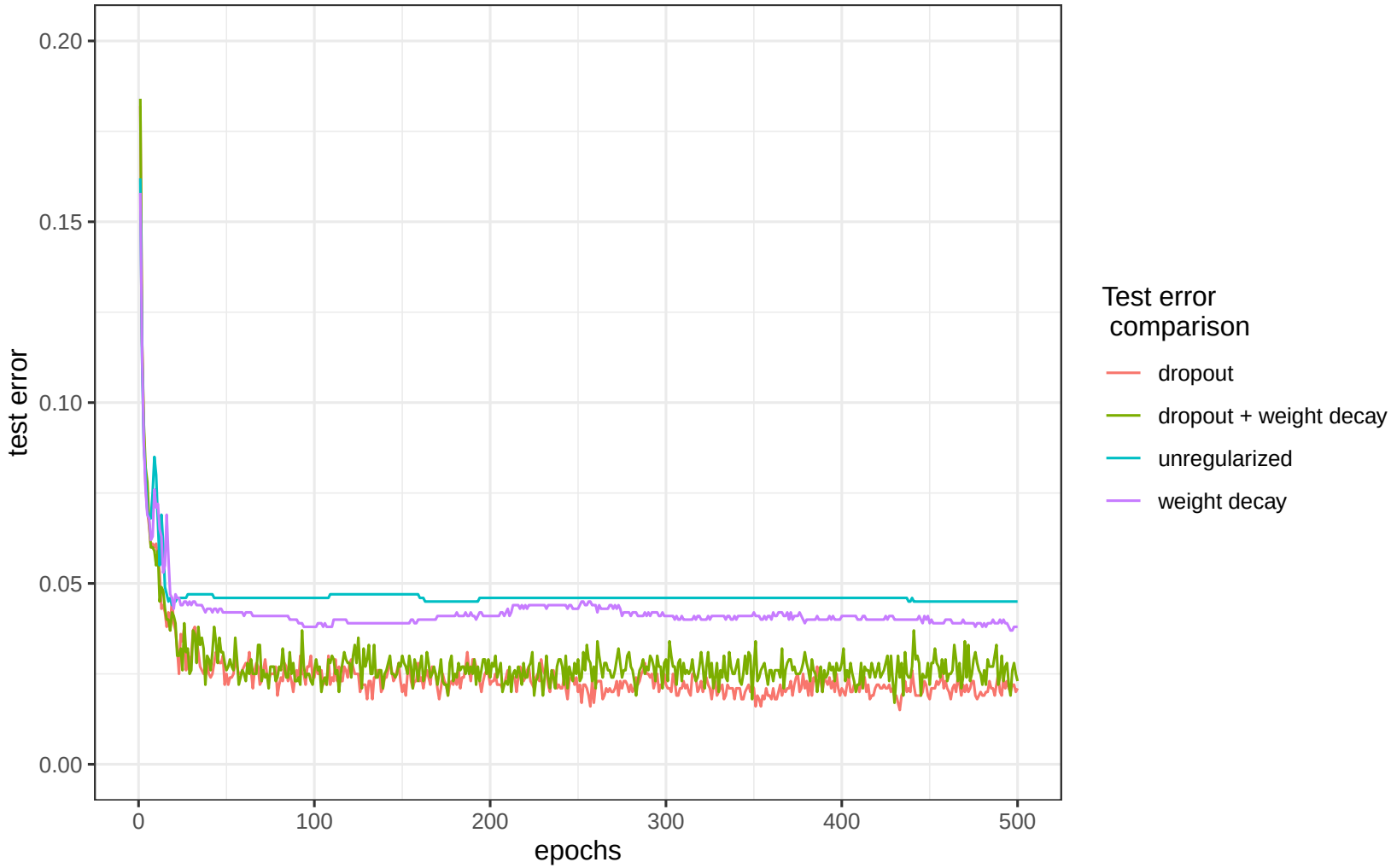


DROPOUT: EXAMPLE / 2



Dropout rate of 0 (no dropouts) leads to higher test error than dropping some units out.

DROPOUT, WEIGHT DECAY OR BOTH?



DROPOUT, WEIGHT DECAY OR BOTH? / 2

Here, dropout leads to a smaller test error than using no regularization or solely weight decay.

Dataset Augmentation

DATASET AUGMENTATION

- Problem: low generalization because high ratio of

$$\frac{\text{complexity of the model}}{\text{\#train data}}$$

- Idea: artificially increase the train data.
  - Limited data supply $\rightarrow$ create “fake data”!
- Increase variation in inputs **without** changing the labels.
- Application:
  - Image and Object recognition (rotation, scaling, pixel translation, flipping, noise injection, vignetting, color casting, lens distortion, injection of random negatives)
  - Speech recognition (speed augmentation, vocal tract perturbation)

DATASET AUGMENTATION / 2

Figure: Example for data set augmentation (Wu et al., 2015)

DATASET AUGMENTATION / 3

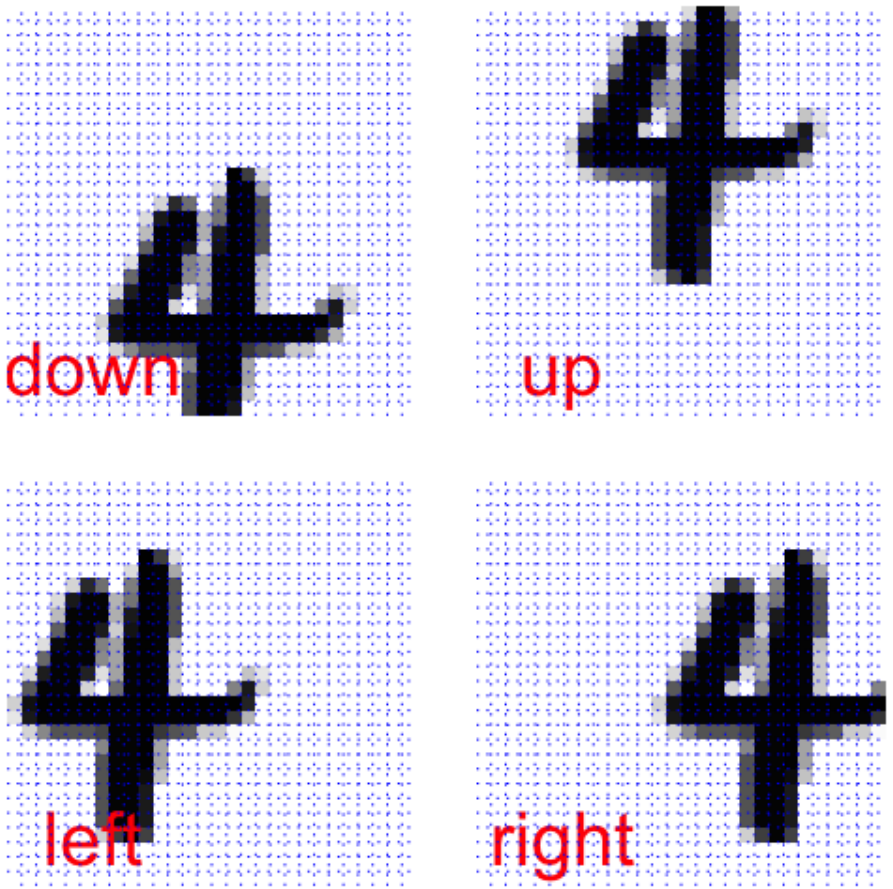


Figure: Example for data set augmentation (Wu et al., 2015)

$\Rightarrow$ careful when rotating digits (6 will become 9 and vice versa)!

REFERENCES


Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning*. MIT Press.


Hinton, G. E., Srivastava , N., Krizhevsky , A., Sutskever , I., & Salakhutdinov, R. R. (2012). Improving neural networks by preventing co-adaptation of feature detectors.


Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: A Simple Way to Prevent Neural Networks from Overfitting. Journal of Machine Learning Research, 15(56), 1929–1958. `http://jmlr.org/papers/v15/srivastava14a.html`


Wu, R., Yan, S., Shan, Y., Dang, Q., & Sun, G. (2015). Deep Image: Scaling up Image Recognition.

Dropout, then and now

Then. Introduced by **Hinton et al. (2012)** and written up in full by **Srivastava et al. (2014, JMLR)**. It helped AlexNet win ImageNet in 2012 and was soon seen as a breakthrough regularizer, used almost everywhere in deep learning.

Now. Large language models are typically pretrained **with dropout switched off**:

- Pretraining usually makes a *single pass* over an enormous dataset, so there is little to memorize and little overfitting to fight.
- **Liu, Bauer & Manning (Findings of ACL 2025)** tested it on BERT-style and GPT-style models (Pythia 160M and 1.4B): downstream performance *improved* without dropout.

The lesson is not "dropout is dead". Regularization answers *overfitting*. On a small dataset seen for many epochs it can still earn its keep – task 4 of the homework measures exactly that. Read the training curves first, then decide.

Putting it together

Five ways to hold back a model that would happily memorise its training set.

The regularization toolbox: which knob when

Technique	**What it does / when to reach for it**
Weight decay (L2)	shrink weights (= Ridge); raise λ if overfitting
Early stopping	stop at the validation valley; free, effectively always on
Dropout	train a crowd of subnetworks (= bagging inside one net); rate 0.2–0.5
Data augmentation	grow the data with *label-preserving* transforms
Good init + BatchNorm	(optimization decks) keep signal & gradients healthy – a mild regularizer too

All fight the same thing – a high-capacity model **memorising** the training set. **Diagnose from the curves, then turn one knob.** Dropout is the NN-specific one; the rest you have met before (Ridge, bagging, cross-validation).

Next: optimizing the training itself

We can now **build**, **train**, and **regularize** a network. The last piece is making training *faster and more reliable* than plain SGD:

- **Momentum** – roll through noise and small bumps instead of zig-zagging.
- **Adam** – a per-parameter adaptive step size; the common default.
- **Learning-rate schedules** – warmup, decay, cosine.

That is the next chunk (LMU's *optimization* week) – the tricks that make SGD actually converge well on huge models.