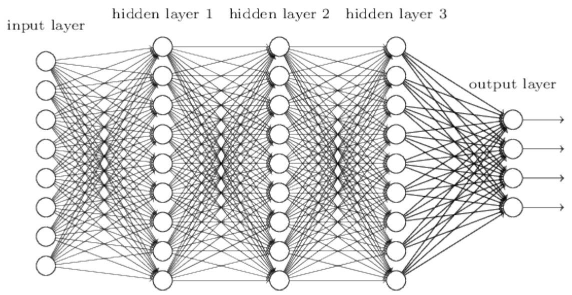
12 PQHS 471 Notes Week 12

12.1 Week 12 Day 1

12.1.1 Backpropagation to compute gradients

Consider a FFNN such as the one in this figure. The layers are numbered from 1 to L. In this figure, $L = 5$.



Let a^l be the vector of outputs from layer l. Then $a^1 = (x_1, \ldots, x_p)^T$ are the inputs of the network, and a^L are the outputs of the network. For $l = 2, \ldots, L$, let z^l be the vector of inputs into layer l ($l = 2, \ldots, L$). Then $z^l = g^l(a^{l-1}) = w^l a^{l-1} + b^l$ and $a^l = h^l(z^l)$, where h^l is an activation function. When the activation is node specific, an element-wise representation is $a^l_j = h^l(\sum_k w^l_{jk} a^{l-1}_k + b^l_j)$, where superscripts denote the layer and subscripts denote the node within the layer.

Let θ^l denote all the parameters (w^l, b^l) between layers $l-1$ and l. Then $z^l = g^l(a^{l-1})$ can be written as $z^l = g_{\theta^l}(a^{l-1})$. The NN model is a "well-specified" function

$$a^L = f(x) = h^L(g_{\theta^L}(h^{L-1}(g_{\theta^{L-1}}(\cdots(h^2(g_{\theta^2}(x)))\cdots)))),$$

with unknown parameters $(\theta^L, \ldots, \theta^2)$. The cost for an observation (x, y) is $C(y, f(x))$.

The **backpropagation algorithm** has two main steps: (1) forward propagation to calculate all the z^l and a^l; (2) backward propagation to calculate all the gradients $\frac{\partial C}{\partial \theta^l}$. We briefly describe the second step below. Note that $\frac{\partial C}{\partial \theta^l} = \frac{\partial C}{\partial f(x)} \frac{\partial f(x)}{\partial \theta^l}$.

For the gradient of $f(x)$ with respect to θ^L, we only need to consider $f(x) = h^L(z^L) = h^L(g_{\theta^L}(a^{L-1}))$, as if the input of the function is a^{L-1}. Then by the chain rule, $\frac{\partial f}{\partial \theta^L} = \frac{\partial f}{\partial z^L}\frac{\partial z^L}{\partial \theta^L} = \frac{\partial h^L(z^L)}{\partial z^L}\frac{\partial g(a^{L-1})}{\partial \theta^L}$. The first term, $\delta^L = \frac{\partial h^L(z^L)}{\partial z^L}$ does not depend on θ^L. For the second term, $\frac{\partial g(a^{L-1})}{\partial w^L_{jk}} = a_k^{(L-1)}$ and $\frac{\partial g(a^{L-1})}{\partial b^L_j} = 1$.

Similarly, for the gradient of $f(x)$ with respect to θ^{L-1}, we only need to consider $f(x) = u(z^{L-1}) = u(g_{\theta^{L-1}}(a^{L-2}))$, where $u() = h^L(g_{\theta^L}(h^{L-1}()))$. Then we use the chain rule again to obtain, $\frac{\partial f}{\partial \theta^{L-1}} = \frac{\partial f}{\partial z^{L-1}}\frac{\partial z^{L-1}}{\partial \theta^{L-1}} = \frac{\partial u(z^{L-1})}{\partial z^{L-1}}\frac{\partial g(a^{L-2})}{\partial \theta^{L-1}}$. The first term is $\delta^{L-1} = \frac{\partial u(z^{L-1})}{\partial z^{L-1}}$ can be computed with the chain rule. The second term is similar as above, $\frac{\partial g(a^{L-2})}{\partial w^{L-1}_{jk}} = a_k^{(L-2)}$ and $\frac{\partial g(a^{L-2})}{\partial b^{L-1}_j} = 1$. We continue doing this until the first layer.

At the end, all the partial derivatives can be written in a matrix/vector format:

$$\begin{cases} \delta^L = \frac{\partial C}{\partial z^L} & = (\Sigma^L)^T \cdot \nabla_a C = \nabla_a C \circ h^L(z^L), & & \text{(BP1)} \\ \delta^l = \frac{\partial C}{\partial z^l} & = (\Sigma^l)^T (w^{l+1})^T \delta^{l+1} = ((w^{l+1})^T \delta^{l+1}) \circ h^l(z^l), & 2 \le l \le L-1, & \text{(BP2)} \\ \frac{\partial C}{\partial b^l} & = \delta^l, & & \text{(BP3)} \\ \frac{\partial C}{\partial w^l} & = \delta^l (a^{l-1})^T. & & \text{(BP4)} \end{cases}$$

Here, $\Sigma^l = \frac{\partial a^l}{\partial z^l} = \{\frac{\partial a^l_i}{\partial z^l_j}\}$ is the Jacobian matrix at layer l.

The implementation of the whole algorithm is straightforward and surprisingly short, as shown in the Python code written by Michael Nielsen (the author of NNDL).

The above algorithm is for a single observation. Over a mini-batch of m observations, one could loop through the algorithm m times, or do them simultaneously in arrays. Nielsen implemented the latter, which takes half time to run compared to the former.

There are several **challenges** in fitting NN models. (1) How do we avoid vanishing gradients? (2) How do we ensure the algorithm converges to the minimum, does not stop prematurely and is not too slow? (3) How do we avoid overfitting given there are so many parameters in a model? (4) How do we select the many hyperparameters to obtain an optimal model? These will be addressed in the next several sections.

12.1.2 Vanishing gradient problem

Vanishing gradient problem: Some gradients can become very small to the point the corresponding parameters stop learning. This can be due to very small δ^l or very small a^{l-1}. In a multi-layer NN, the first few layers can have very small δ^l.

To address this problem, people has used the following approaches: (1) carefully choose the cost function and activation function; (2) initialize the weights and biases; (3) Use residual layers (Microsoft's ResNet 2015).

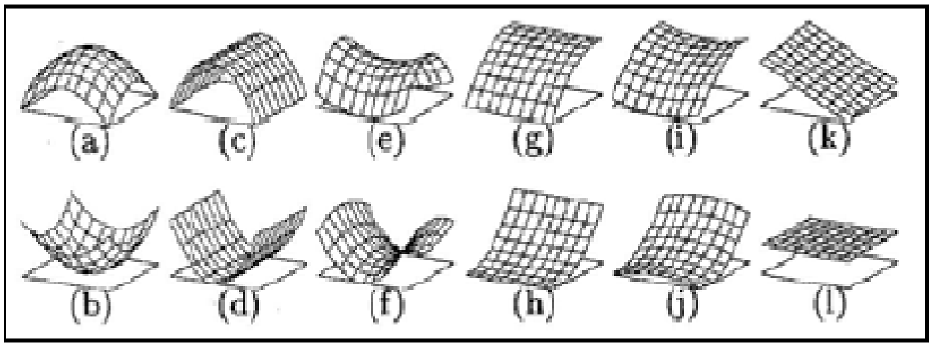
Initiation of parameters: For a sigmoid neuron with n_{in} inputs, where n_{in} is large (say, 1000), initiating $w, b \sim N(0,1)$ will have a high chance to have a saturated $\sigma(z)$ (i.e., $\sigma(z)$ is very close to 0 or 1, and $\sigma'(z) \approx 0$) and thus slow learning. Initiating $w \sim N(0, 1/n_{\text{in}})$ will be much better; this is the LeCun normal initializer.

A popular choice for initializing the weights is the Glorot uniform initializer (or Xavier uniform initializer) (Xavier Glorot and Yoshua Bengio, 2010). It draws values from a uniform distribution $U(-a_G, a_G)$, where $a_G = \sqrt{6/(n_{\text{in}} + n_{\text{out}})}$, n_{in} and n_{out} are the numbers of input and output units for the weight matrix. Another choice is the Glorot normal initializer, which draws values from a truncated normal distribution with mean 0 and SD $= \sqrt{2/(n_{\text{in}} + n_{\text{out}})}$.

Another option is the He uniform variance scaling initializer (He et al., 2015), which draws values from a uniform distribution $U(-a_H, a_H)$, where $a_H = \sqrt{6/n_{\text{in}}}$. The He normal initializer draws values from a truncated normal distribution with mean 0 and SD $= \sqrt{2/n_{\text{in}}}$. The LeCun uniform initializer draws values from a uniform distribution $U(-a_L, a_L)$, where $a_L = \sqrt{3/n_{\text{in}}}$; the LeCun normal initializer draws values from a truncated normal distribution with mean 0 and SD $= \sqrt{1/n_{\text{in}}}$.

12.1.3 Improving gradient descent

The high dimensional cost function can have "pits", "ravines", and "saddles".



(a) peak, (b) pit, (c) ridge, (d) ravine, (e) ridge saddle, (f) ravine saddle, (g) convex hill, (h) concave hill, (i) convex saddle hill, (j) concave saddle hill, (k) slop hill, (l) flat (figure from Ferdowsi and Ahmadyfard, 2008)

Standard GD: At iteration k, we update the parameter estimate $\theta_k = \theta_{k-1} - \eta_k g_k$, where η_k is the learning rate and $g_k = \nabla C(\theta_{k-1})$ is the gradient at θ_{k-1}. The amount of update is $-\eta_k g_k$.

To improve the performance of GD, people have tried many treaks. They fall into 4 types: (1) decreasing the learning rate, (2) adding a momentum, (3) computing the learning rate adaptively and allowing them to differ across parameters, and (4) computing both learning rate and gradient adaptively. Below are a few popular ones. Alec Radford generated a few GIFs to show a comparison of these approaches (Adam not included).

1. A **learning rate schedule** is a predetermined scheme to gradually decrease the learning rate instead of using a constant learning rate. (This is similar to simulated annealing.)

In `optimizer_sgd(lr=0.01, momentum=0.0, decay=0.0, nesterov=False)`, `lr` is the learning rate η, `decay` is a value d such that the learning rate at iteration k is $\eta_k = \eta/(1 + dk)$. By default, `decay=0` and $\eta_k = \eta$ for all k.

2. **Momentum**-based GD: Define a *velocity* vector v_k (initialize with $v_0 = 0$). At every iteration, we compute

$$v_k = \mu v_{k-1} - \eta_k g_k, \text{ and then } \theta_k = \theta_{k-1} + v_k.$$

Here, v_k is the sum of the amount of standard GD and the amount of momentum from the previous iteration, μv_{k-1}. The hyper-parameter $0 \le \mu \le 1$ is called the *momentum* and $1 - \mu$ is the *friction*. When $\mu = 0$, we have no momentum buildup and this reduces to the standard GD. When μ is close to 1, the momentum can build up quickly.

- **Nesterov's momentum** (Nesterov's accelerated gradient, or NAG): Same as above, but with the gradient calculated at the "lookahead" position $\theta_{k-1} + \mu v_{k-1}$ instead of θ_{k-1} to obtain a better assessment of the gradient. That is, $v_k = \mu v_{k-1} - \eta_k \nabla C(\theta_{k-1} + \mu v_{k-1})$. It significantly improved the performance of some RNN tasks. Bengio et al. (https://arxiv.org/abs/1212.0901) showed an alternative but equivalent formulation, which is implemented in `keras.optimizers.SGD()`:

(1) $v_0 = 0$
(2) In iteration k, compute η_k, $g_k = \nabla C(\theta_{k-1})$, and $v_k = \mu v_{k-1} - \eta_k g_k$. Then
  (a) If `nesterov = False`, $\theta_k = \theta_{k-1} + v_k$.
  (b) If `nesterov = True`, $\theta_k = \theta_{k-1} + \mu v_k - \eta_k g_k$.

3. **Adagrad** is to adaptively compute the learning rate for each parameter instead of using the same value for all parameters. The j-th parameter at iteration k is

$$\theta_{k,j} = \theta_{k-1,j} - \eta_{k,j} g_{k,j}, \text{ where } \eta_{k,j} = \eta/\sqrt{G_{k,j} + \epsilon}.$$

Here, $G_{k,j} = \sum_{l \le k} g_{l,j}^2$ is the cumulative squared gradients for the j-th parameter, and ϵ is a very small number to avoid division by zero. As we train the model, $G_{k,j}$ will grow, the learning rates may shrink to zero. In this case, the model may prematurely stop learning on the parameter.

- **RMSprop**: In the denominator of the formula for $\eta_{k,j}$, the sum of squares $G_{k,j}$ is replaced by exponentially weighted average (EWA) of the squared gradients,

$$\text{EWA}_{k,j} = (1 - \rho)\text{EWA}_{k-1,j} + \rho g_{k,j}^2,$$

where $0 < \rho < 1$. EWA is approximately equivalent to giving exponentially reduced weights to earlier iterations: $w_k = 1$ for $g_{k,j}^2$, $w_{k-1} = \rho$ for $g_{k-1,j}^2$, $w_{k-2} = \rho^2$ for $g_{k-2,j}^2$, etc.. Then $\text{EWA}_{k,j} \approx \frac{w_k g_{k,j}^2 + w_{k-1} g_{k-1,j}^2 + \cdots}{w_k + w_{k-1} + \cdots}$. ("RMS" in the name: $\sqrt{\text{EWA}_{k,j}}$ is the exponentially weighted RMS (root mean squares) of the gradients.)

- **Adadelta**: In the formula for $\eta_{k,j}$, the numerator η is replaced by an adaptive formula, while the denominator is similarly defined as in RMSprop. With Adadelta, we do not even need to tune the learning rate.

In `optimizer_rmsprop(lr=0.001, rho=0.9, epsilon=None, decay=0.0)`, the default $\rho = 0.9$. It is recommended to leave the parameters of this optimizer at their default values (except the learning rate).

In `optimizer_adadelta(lr=1.0, rho=0.95, epsilon=None, decay=0.0)`, it is recommended to leave the parameters of this optimizer at their default values. Here `lr` can be viewed as the initial value of learning rate.

4. **Adam**: Instead of $g_{k,j}$, an EWA of $g_{k,j}$ is used. That is, $m_{k,j} = \beta_1 m_{k-1,j} + (1 - \beta_1) g_{k,j}$, and

$$\theta_{k,j} = \theta_{k-1,j} - \eta_{k,j} m_{k,j},$$

where $\eta_{k,j}$ is similarly defined as in RMSprop (using notation β_2 instead of ρ). Both β_1 and β_2 should be < 1 but close to 1.

In Keras, `optimizer_adam(lr=0.001, beta_1=0.9, beta_2=0.999, epsilon=None, decay=0.0, amsgrad=False)`

"Adam is currently recommended as the default algorithm to use, and often works slightly better than RMSProp. However, it is often also worth trying SGD+Nesterov Momentum as an alternative." — CS231n (http://cs231n.github.io/neural-networks-3/)

12.1.4 Overfitting

If an iterative fitting algorithm shows no improvement on validation data after a certain number of epochs but continues to improve on training data, it is a sign of **overfitting**. How "improvement" is quantified can give drastically different impressions on when overfitting begins.

Below are a few common choices for overcoming overfitting in NNs.

Early stopping: Stop once the classification accuracy on the validation data has plateaued (e.g., not improving over the last 10 epochs). But it is not uncommon that the performance plateau for a while and then improve again. Early stopping was a popular choice in the past. Nowadays regularization, dropout, and batch normalization are often used.

Regularization: In L_2 regularization (also called weight decay), the cost function is $C = C_0 + \frac{\lambda}{2n}\sum_w w^2$, where C_0 is the original cost. In L_1 regularization, $C = C_0 + \frac{\lambda}{n}\sum_w |w|$. Here, the sums are over the weights, not the biases. No standardization is performed and all the weights are inherently assumed to be on comparable scales. These types of regularization slow the rate of overfitting.

In L_2 regularization, $C = C_0 + \frac{\lambda}{2n}\sum_w w^2$, then $\frac{\partial C}{\partial b} = \frac{\partial C_0}{\partial b}$ and $\frac{\partial C}{\partial w} = \frac{\partial C_0}{\partial w} + \frac{\lambda}{n}w$. The update of weights is $w \leftarrow w - \eta\frac{\partial C_0}{\partial w} - \eta\frac{\lambda}{n}w = (1 - \frac{\eta\lambda}{n})w - \eta\frac{\partial C_0}{\partial w}$. Here the current estimate is shrinked by a factor of $(1 - \frac{\eta\lambda}{n})$ before adding the gradient. Thus this approach is also called **weight decay**. If $\eta\lambda \ll n$, $(1 - \frac{\eta\lambda}{n}) \approx \exp(-\frac{\eta\lambda}{n})$.

In L_1 regularization, $C = C_0 + \frac{\lambda}{n}\sum_w |w|$, the update of weights is $w \leftarrow w - \frac{\eta\lambda}{n}\text{sign}(w) - \eta\frac{\partial C_0}{\partial w}$ (with the convention $\text{sign}(0) = 0$).

Dropout: In every iteration, randomly remove a fraction ϕ $(0 < \phi < 1)$ of the hidden neurons and their connections from the model, inflate the remaining neurons by a factor $1 = 1/\phi$, and update the parameters in the modified network. The dropout rate ϕ is a tuning parameter. One can use different dropout rates for different layers. From Krizhevsky (2012): "This technique reduces complex co-adaptations of neurons, since a neuron cannot rely on the presence of particular other neurons. It is, therefore, forced to learn more robust features that are useful in conjunction with many different random subsets of the other neurons."

It can be shown that dropout is approximately a form of generalized ridge regularization. In Keras, it is specified as a model layer using `layer_dropout(rate=)`.

Artificial expansion of training data: Apply operations to data that reflect real-world variation. For image data, rotation/translation/skewing can be used. For MNIST data, "elastic distortions" was used to emulate the random oscillations found in hand muscles (Simard et al. 2003). Input distortion enriches the training data and prevents overfitting to the original image. It forces the model not to focus on isolated values and individual pixels.

It can be shown that input distortion is approximately a form of ridge regularization. One can apply distortions to a test image, and then "poll" the results to produce a final classification. This process is called denoising.

Batch normalization: From Ioffe and Szegedy (2015): "Our method draws its strength from making normalization a part of the model architecture and performing the normalization for each training mini-batch. Batch Normalization allows us to use much higher learning rates and be less careful about initialization. It also acts as a regularizer, in some cases eliminating the need for Dropout."

12.1.5 The MNIST example revisited

I have tried a few options described above for improving GD and overcoming overfitting. Given a fixed model structure (i.e., fixed numbers of layers and nodes), having dropout layers and using the Adam optimizer seem to perform the best among the limited scenarios I have tried on this dataset.

Prepare the data:

```
library(keras)
c(c(x_train, y_train), c(x_test, y_test)) %<-% dataset_mnist()
dim(x_train) = c(nrow(x_train), 784)  ## reshape/flattening X to have 784 columns
dim(x_test) = c(nrow(x_test), 784)
x_train = x_train / 255  ## raw data has range 0-255
x_test = x_test / 255
y_train10 = to_categorical(y_train, 10)  ## make dummy (one-hot) variables for the outcome
y_test10 = to_categorical(y_test, 10)
```

Define and fit a FFNN model. Try the optimizers

```
use_session_with_seed(2018)
model = keras_model_sequential() %>%
  layer_dense(units = 100, activation = "relu", input_shape = 784) %>%
  layer_dropout(rate = 0.3) %>%
  #layer_dense(units = 100, activation = "relu") %>%
  #layer_dropout(rate = 0.3) %>%
  layer_dense(units = 10, activation = "softmax")

summary(model)

model %>% compile(
  loss = "categorical_crossentropy",
  #optimizer = optimizer_sgd(lr=3, momentum=0.5, decay=0, nesterov=T),
  #optimizer = optimizer_rmsprop(),
  #optimizer = optimizer_adadelta(),
  optimizer = optimizer_adam(),
  metrics = c("accuracy")
)

history = model %>% fit(
  x_train, y_train10,
  epochs = 20, batch_size = 100, verbose = 1,
  #validation_data = list(x_test, y_test10)
  validation_split = 0.2
)
```

One can plot the training history and apply the model to new data to make predictions.

```
plot(history)
history$metrics$val_acc
 [1] 0.9345833 0.9547500 0.9599167 0.9658333 0.9681667 0.9700000 0.9709167
 [8] 0.9728333 0.9726667 0.9730000 0.9730833 0.9745833 0.9748333 0.9756667
[15] 0.9762500 0.9755833 0.9768333 0.9760000 0.9769167 0.9766667

model %>% evaluate(x_test, y_test10, verbose = 0)  ## 0.9771 on test set

y_pred = model %>% predict_classes(x_test)  ## prediction on test set
table(y_test, y_pred)  ## test set confusion matrix

y_predprob = model %>% predict_proba(x_test)  ## softmax probabilities, 10000 x 10 matrix
all.equal(apply(y_predprob, 1, which.max)-1, as.numeric(y_pred))  ## True
```

To output the weights and biases for the whole model or for a layer, use `get_weights()`. The result is a list. For layers, `index=0` is the input layer, `index=1` is the first hidden layer, etc. In the model above, there is a "dropout layer" (`index=2`) after the first hidden layer, thus `index=3` is the real layer after the first hidden layer.

```
## get weights and biases for the whole model
wts = model %>% get_weights
#wts = get_weights(model)
str(wts)

## get weights and biases for a specific layer
wts1 = model %>% get_layer(index=1) %>% get_weights
#wts1 = get_weights(get_layer(model, index=1))
#wts1 = get_layer(model, index=1) %>% get_weights
str(wts1)

wts2 = model %>% get_layer(index=3) %>% get_weights
str(wts2)
```

Sanity check: Apply the weights to an observation to see if the results are as expected.

```
testsample = x_test[1000, , drop=F]  ## take an observation
model %>% predict_classes(testsample)
model %>% predict_proba(testsample)

## For the NN with a single hidden relu layer
z2 = testsample %*% wts[[1]] + array_reshape(wts[[2]], c(1, length(wts[[2]])))
a2 = pmax(z2, 0)  ## relu
z3 = exp(a2 %*% wts[[3]] + array_reshape(wts[[4]], c(1, length(wts[[4]]))))
a3 = z3 / sum(z3)  ## softmax
all.equal(a3, model %>% predict_proba(testsample))  ## True (effectively)
```

One can also obtain intermediate output values. Below is an example to obtain the values after the first hidden layer, and after the final layer.

```
layer1_model = keras_model(inputs = model$input,
                           outputs = get_layer(model,index=1)$output)
testsample_layer1 = predict(layer1_model, testsample)
all.equal(a2, testsample_layer1)  ## True (effectively)

layer3_model = keras_model(inputs = model$input,
                           outputs = get_layer(model,index=3)$output)
testsample_layer3 = predict(layer3_model, testsample)
all.equal(model %>% predict_proba(testsample), testsample_layer3)  ## True
```

Regularization: To apply regularization, use the `kernel_regularizer=` argument in `layer_dense()` to specify the "regularizer". Below is an example of using L_2 regularization on the weights in the first hidden layer with $lambda = 0.01$. Other options for the argument are `regularizer_l1()` and `regularizer_l1_l2()`. To regularize the biases, use the `bias_regularizer=` argument.

```
model = keras_model_sequential() %>%
  layer_dense(units = 100, activation = "relu", input_shape = 784,
              kernel_regularizer = regularizer_l2(l = 0.01)) %>%
  layer_dropout(rate = 0.3) %>%
  layer_dense(units = 10, activation = "softmax")
summary(model)
```

Initialization of weights and biases: By default, in `layer_dense()`, `kernel_initializer = "glorot_uniform"` for the weights and `bias_initializer = "zeros"` for the biases. The "zeros" initializer sets values to be zero. One can specify the seed for the random generator, say, `kernel_initializer = glorot_uniform(seed = 2018)`. There are more than a dozen "initializers" in Keras.

Batch normalization: To add batch normalization to a hidden layer before its activation function, one can do it as in the following, where the first hidden layer has batch normalization before applying the activation function.

```r
model = keras_model_sequential() %>%
  layer_dense(units = 100, input_shape = 784) %>%
  layer_batch_normalization() %>%
  layer_activation("relu") %>%
  layer_dropout(rate = 0.3) %>%
  layer_dense(units = 100, activation = "relu") %>%
  layer_dropout(rate = 0.3) %>%
  layer_dense(units = 10, activation = "softmax")
summary(model)
```

12.1.6 Hyperparameters

Hyperparameters in FFNN fall in three categories:

(1) Those defining the model: number of layers, number of neurons in a layer, activation functions, feature map size and pooling function in CNN.
(2) Those in model fitting criteria: cost function C, λ in regularization (λ can also be viewed as part of a model (as prior of model preference)).
(3) Those used during model fitting: weight initialization, learning rate η, momentum coefficient μ, dropout rate ϕ, mini-batch size, number of epochs.

NNDL Chapter 3 covers hyperparameter tuning in great details. Below is a very brief summary:

- First goal in model building is to get a model better than noise.
- Use a fraction of training and validation data to speed up feedback.
- For a multinomial outcome, use only two outcome categories to simplify the problem.
- Vary hyperparameters one at a time to see if they would improve the performance.

12.1.7 Example: Prediction of Hi-C intensity

Background: Chromatin conformation capture (3C) is a technique developed in 2001 to evaluate physical interactions between two chromosomal regions. It allowed us to study regulation of gene expression from the physical perspective (in addition to the molecular and statistical perspectives). The technique has been improved to have 4C, 5C, and Hi-C, etc. Hi-C (high-throughput CCC) allows us to obtain data on chromatin conformation agnostically over the whole genome. Hi-C data are expensive to generate. Researchers typically generate data that are good for obtaining confident intensity calling at the 40kb resolution.

Goal: Build a prediction model to make confident Hi-C intensity calling at a more refined resolution.

12.1.8 Customized evaluation metrics in Keras

In Keras, you can define new evaluation metrics to be computed at the end of every epoch. Here is an example we wrote in Python. The metric is to calculate Pearson's correlation between the observed and the predicted values.

```python
## Define a metric
def pearson_in_epoch(y_true, y_pred):
    fsp = y_pred - K.mean(y_pred, axis=0)
    fst = y_true - K.mean(y_true, axis=0)
    devP = K.std(y_pred, axis=0)
    devT = K.std(y_true, axis=0)
    return K.mean(fsp*fst, axis=0) / (devP*devT + K.epsilon())

## To use the metric
model.compile(loss='mean_squared_error', optimizer='rmsprop',
              metrics=['mae', pearson_in_epoch])
```

12.1.9 Assignment

1. Reading for next lecture: NNDL 6

12.2 Week 12 Day 2

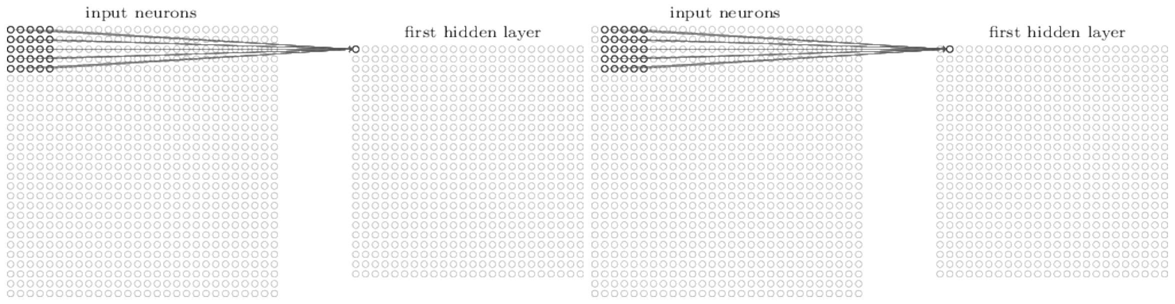
12.2.1 Convolutional neural networks (CNNs)

CNNs are designed to capture patterns in data that have local (e.g., spatial or temporal) dependencies. They are often used for image processing (2D source) and video processing (3D source).

In CNNs, one key concept is the **feature map** (also called *kernel* or *filter*). In a 2D CNN, it is a $k \times k \to 1$ mapping. For example, we can "shrink" a 28×28 image to a 24×24 image by applying a function

$$f^{(1)}(\{t_{i,j} : 1 \le i,j \le 5\}) = h^{(1)}\left(\sum_{i=1}^{5}\sum_{j=1}^{5} w_{i,j}^{(1)} t_{i,j} + b^{(1)}\right)$$

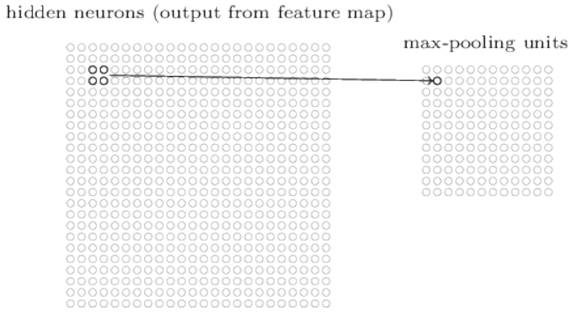
to every square of 5×5 pixels, as shown in the figures below. Here, $h^{(1)}$ is an activation function. This feature map has 26 parameters. The double summation inside the parentheses is a convolution operation in math; thus the name of the method. The w's are called *shared weights* and the b is called *shared bias*. For every neuron in the resulting 24×24 image, its input, a 5×5 region, is called the **local receptive field** for the hidden neuron. In this feature map, the **stride length** is 1 because we apply the function to a square and then to the next square by moving horizontally or vertically by 1 pixel. If we apply the function to a square and then to the next square by moving 2 pixels, the stride will be 2. For example, we may use a $4 \times 4 \to 1$ feature map with stride 2 to map a 28×28 image to a 13×13 image.



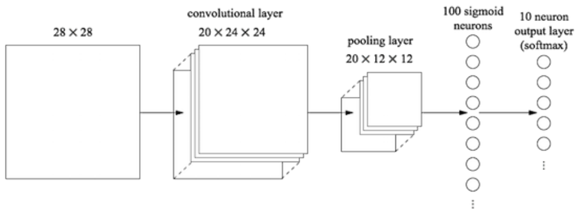
Sometimes the images are **padded** with zeros to make the result of a convolutional layer having the same resolution as its source image. For example, if we pad a 28×28 image with 2 pixels of zeros at all sides to make it a 32×32 image, then a $5 \times 5 \to 1$ feature map will result in a 28×28 image.

A feature map is sometimes followed by a **pooling** operation. For example, the figure below shows a $2 \times 2 \to 1$ "max-pooling" operation, which takes every 2×2 square and extract the maximum of the 4 pixel values. The squares are non-overlapping. Thus a $2 \times 2 \to 1$ "max-pooling" on a 24×24 image results in a 12×12 image.

An alternative to "max-pooling" is "average-pooling". Another option is the l_2-pooling, which is equivalent to the RMS, but often defined as the square root of the sum of squares of the input region. One can create a customized pooling layer by defining a Lambda layer with `layer_lambda()`. A pooling layer does not have any parameters.



For a feature map, the same function is used for all squares. Thus, a feature map captures a certain local pattern. Often there are many feature maps in a hidden layer so that we can capture different local patterns. For example, in the figure below, the first convolutional layer contains 20 feature maps of $5 \times 5 \to 1$ with stride 1. Thus there are 20 functions, $f^{(1)}, \ldots, f^{(20)}$, for this layer. It is followed by a "max-pooling" layer of $2 \times 2 \to 1$, a full-connected layer with 100 sigmoid neurons, and a full-connected softmax output layer with 10 neurons. This model has 289,630 parameters!



The layers and nodes of the whole model may be represented as

$$28 \times 28 \times 1 \overset{5\times5\to1}{\Longrightarrow} 24 \times 24 \times 20 \overset{2\times2\to1}{\Longrightarrow} 12 \times 12 \times 20 \Longrightarrow 2880 \Longrightarrow 100 \Longrightarrow 10$$

Another way to represent this model is "conv5-20, maxpool2, FC-100, softmax-10".

Note that both feature map and pooling can be done over rectangular regions with different width and height. The stride length can also be different horizontally and vertically.

The shared weights of a feature map can be drawn as a image to indicate what pattern is captured by the feature map. The weights, which can take any value, need to be transformed to $[0, 1]$ before drawing (e.g., using the sigmoid function). An example is in the NNDL book, showing the shared weights for 20 feature maps as 5×5 images, with whiter blocks for smaller (typically, more negative) weights and darker blocks for larger (more positive) weights. We will draw the shared weights for our model below.

12.2.2 The MNIST example with CNN

MNIST data preparation: Note that X is reshaped to be a 4D array for the CNN model to use.

```r
library(keras)

c(c(x_train, y_train), c(x_test, y_test)) %<-% dataset_mnist()
x_train <- array_reshape(x_train, c(nrow(x_train), 28, 28, 1))
#dim(x_train) <- c(nrow(x_train), 28, 28, 1)  ## same result
x_test <- array_reshape(x_test, c(nrow(x_test), 28, 28, 1))
x_train = x_train / 255  ## raw data has range 0-255
x_test = x_test / 255
y_train10 = to_categorical(y_train, 10)  ## make dummy (one-hot) variables for the outcome
y_test10 = to_categorical(y_test, 10)
```

Set up the model: We use ReLU activations here.

```r
use_session_with_seed(2018)
model = keras_model_sequential() %>%
  layer_conv_2d(filters = 20, kernel_size = c(5,5), activation = 'relu',
                input_shape = c(28,28,1)) %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_flatten() %>%
  layer_dense(units = 100, activation = 'relu') %>%
  layer_dense(units = 10, activation = 'softmax')
summary(model)
```

Number of parameters: (1) between the input and the first convolutional layer, $26 \times 20 = 520$; (2) between the max-pooling layer (which has $20 \times 12 \times 12 = 2880$ pixels) and the layer with 100 neurons, $2881 \times 100 = 288100$; (3)

between the last two layers, $101 \times 10 = 1010$. The total is 289,630. The max-pooling layer and the "flatten" layer do not have any parameters.

Using Adadelta as the optimizer, 12 epochs and batch size 128, this model achieves 98.78% test set accuracy.

```
model %>% compile(
  loss = loss_categorical_crossentropy,
  optimizer = optimizer_adadelta(),
  #optimizer = optimizer_rmsprop(),
  metrics = c('accuracy')
)

history = model %>% fit(
  x_train, y_train10,
  epochs = 12, batch_size = 128, verbose = 1,
  validation_data = list(x_test, y_test10)
  #validation_split = 0.2
)
```

To save/load the Keras mode.

```
save_model_hdf5(model, "MNIST-CNNmodel1.hdf5")
save(history, file="MNIST-CNNmodel1-history.RData")
model = load_model_hdf5("MNIST-CNNmodel1.hdf5", compile=F)
load("MNIST-CNNmodel1-history.RData")
```

```
history$metrics$val_acc
 [1] 0.9606 0.9715 0.9801 0.9808 0.9814 0.9834 0.9860 0.9856 0.9871 0.9874 0.9881 0.9878
y_pred = model %>% predict_classes(x_test)
table(y_test, y_pred)
      y_pred
y_test    0    1    2    3    4    5    6    7    8    9
     0  974    0    1    0    0    0    1    1    3    0
     1    0 1131    0    0    0    2    2    0    0    0
     2    3    4 1004    5    1    0    0    8    7    0
     3    0    0    0 1004    0    3    0    0    3    0
     4    0    1    0    0  975    0    2    0    1    3
     5    2    0    0    5    0  881    3    0    1    0
     6    5    2    0    0    1    3  947    0    0    0
     7    0    3    4    3    0    0    0 1014    1    3
     8    4    0    0    0    0    0    1    2  963    4
     9    2    3    0    3    7    1    0    6    2  985
```

In NNDL Chapter 6, the author used the same network structure with sigmoid activation, standard SGD with learning rate 0.1, 60 epochs and batch size 10. That model also achieved 98.78% test set accuracy.

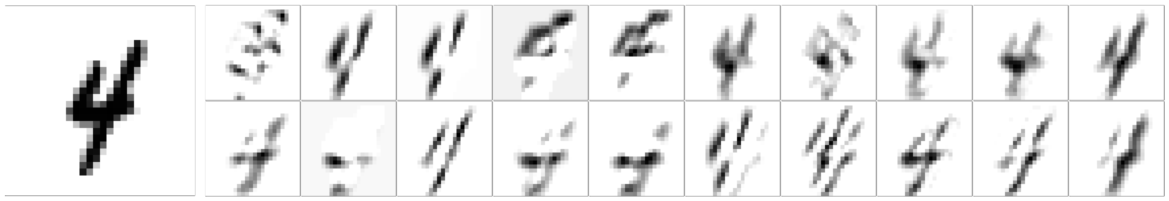
```
## get weights and biases for the whole model
wts = model %>% get_weights
str(wts)

## get weights and biases for a specific layer
wts1 = model %>% get_layer(index=1) %>% get_weights
str(wts1)
wts2 = model %>% get_layer(index=4) %>% get_weights
str(wts2)
wts3 = model %>% get_layer(index=5) %>% get_weights
str(wts3)
```

Sanity check: Apply the weights to an observation to see if the results are as expected.

```
testsample = x_test[7000,,,, drop=F]  ## take an observation
dim(testsample)  ## (1, 28, 28, 1)
range(testsample)
model %>% predict_classes(testsample)
model %>% predict_proba(testsample)
```

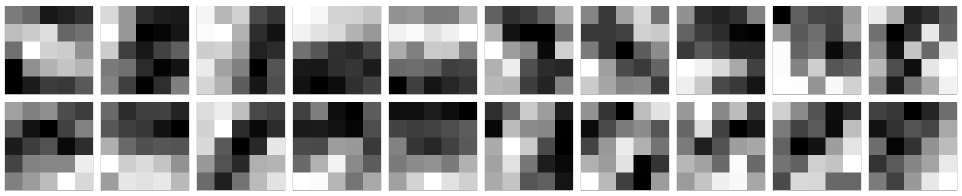
We can obtain intermediate output values after the first CNN layer and plot them.

```
testsample = x_test[7000,,,, drop=F]  ## take an observation
layer1_model = keras_model(inputs = model$input,
                           outputs = get_layer(model,index=1)$output)
testsample_layer1 = predict(layer1_model, testsample)
dim(testsample_layer1)  ## (1, 24, 24, 20)

im1 = t(drop(testsample))[,28:1]
image(1:28, 1:28, im1, col=gray(1-(0:255)/255), xaxt='n', yaxt='n')

#pdf("a.pdf", width=15, height=3)
#par(mfrow=c(2,10), mar=rep(.1,4))
for(i in 1:20) {
  aa = testsample_layer1[1,,,i]
  im1 = t(aa)[,24:1]
  image(1:24, 1:24, im1, col=gray(1-(0:255)/255), xaxt='n', yaxt='n')
}
#dev.off()
```



To plot the shared weights of the 20 feature maps in the first hidden layer. The weights are transformed to [0, 1] using the sigmoid function. The whiter blocks are for smaller (typically, more negative) weights and darker blocks for larger (more positive) weights.

```
wts1 = model %>% get_layer(index=1) %>% get_weights
fmaps = wts1[[1]]

for(i in 1:20) {
  aa = 1/(1+exp(-fmaps[,,1,i]))
  im1 = t(aa)[,5:1]
  image(1:5, 1:5, im1, col=gray(1-(0:255)/255), xaxt='n', yaxt='n')
}
```



In Keras, the default stride length is 1 both horizontally and vertically. To specify stride lengths, use the argument `strides=` in `layer_conv_2d()`. For example, `strides=2` means stride length 2 both horizontally and vertically; `strides=c(1,2)` means stride length 1 horizontally and 2 vertically.

12.2.3 Multiple input images

Note that in `layer_conv_2d()`, the argument `input_shape=` requires 3 numbers: the first two specify the resolution of the input image(s) and the third is the number of images. For example, a color image contains 3 images (e.g., for the red, greed, and blue components). Another example is the CNN below, which has a second convolutional layer with 40 feature maps of $5 \times 5 \to 1$. The input for that layer are 20 12×12 images after the first max-pooling layer. A feature map in the second convolutional layer will have a function like this:

$$f^{(1)}(\{t_{i,j;k} : 1 \le i, j \le 5, 1 \le k \le 20\}) = h^{(1)}\left(\sum_{k=1}^{20}\sum_{i=1}^{5}\sum_{j=1}^{5} w_{i,j;k}^{(1)} t_{i,j;k} + b^{(1)}\right),$$

with $5 \times 5 \times 20 + 1 = 501$ parameters. The total number of parameters between the first max-pooling layer and the second convolutional layer is $501 \times 40 = 20040$.

The layers and nodes of the whole model may be represented as

$$28 \times 28 \times 1 \overset{5\times5\to1}{\Longrightarrow} 24 \times 24 \times 20 \overset{2\times2\to1}{\Longrightarrow} 12 \times 12 \times 20 \overset{5\times5\to1}{\Longrightarrow} 8 \times 8 \times 40 \overset{2\times2\to1}{\Longrightarrow} 4 \times 4 \times 40 \Longrightarrow 640 \Longrightarrow 100 \Longrightarrow 10$$

Another way to represent this model is "conv5-20, maxpool2, conv5-40, maxpool2, FC-100, softmax-10". This model has 85,670 parameters. Note that adding the second convolutional layer greatly reduces the number of parameters from 289,630 to 85,670!

12.2.4 The MNIST example with CNN (cont'd)

The above model can be specified below.

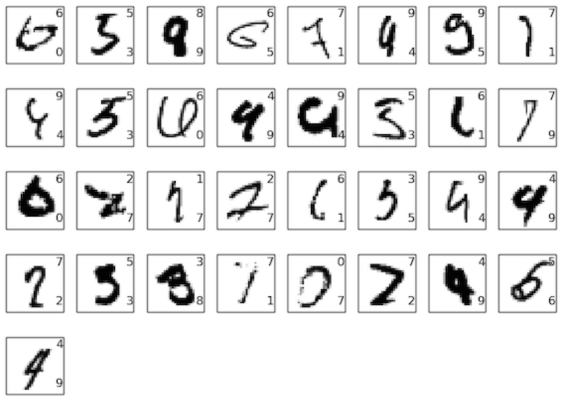
```r
model1 = keras_model_sequential() %>%
  layer_conv_2d(filters = 20, kernel_size = c(5,5), activation = 'sigmoid',
                input_shape = c(28,28,1)) %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_conv_2d(filters = 40, kernel_size = c(5,5), activation = 'sigmoid') %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_flatten() %>%
  layer_dense(units = 100, activation = 'sigmoid') %>%
  layer_dense(units = 10, activation = 'softmax')
summary(model1)
```

In NNDL Chapter 6, the author used the above network structure with standard SGD with learning rate 0.1, 60 epochs and batch size 10. That model achieved 99.06% test set accuracy. Changing all the activation functions to ReLU improved the performance to 99.23%. Expanding the training data improved the performance to 99.37%. Adding another fully connected ReLU layer with 100 neurons before the output layer improved the performance to 99.43%.

He finally tried the following model with 1000 neurons on the last two hidden layers, which has 1,672,570 parameters, to achieve 99.60% test set accuracy (with dropout and 40 epochs). He also applied dropout on the softmax layer, which I do not know how to do in Keras.

```r
model2 = keras_model_sequential() %>%
  layer_conv_2d(filters = 20, kernel_size = c(5,5), activation = 'relu',
                input_shape = c(28,28,1)) %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_conv_2d(filters = 40, kernel_size = c(5,5), activation = 'relu') %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_flatten() %>%
  layer_dense(units = 1000, activation = 'relu') %>%
  layer_dropout(rate = 0.5) %>%
  layer_dense(units = 1000, activation = 'relu') %>%
  layer_dropout(rate = 0.5) %>%
  layer_dense(units = 10, activation = 'softmax')
summary(model2)
```

He re-trained this model 5 times and applied ensemble on the 5 models (i.e., majority vote, as in random forests), to achieve 99.67%. The 33 misclassified images are below:



12.2.5 Example: The `lime` package

https://tensorflow.rstudio.com/blog/lime-v0.4-the-kitten-picture-edition.html

The R `lime` package implements the method by Ribeiro, Singh, Guestrin, 2016 to explain the predictions of a classifier.

The demonstration uses the VGG16 model, a winner of the 2014 ILSVRC, which has 16 hidden layers (excluding pooling and flatten layers). The layers and nodes of the whole model can be represented as

$$
\begin{aligned}
224\times224\times3 &\overset{3\times3\to1}{\Longrightarrow} 224\times224\times64 \overset{3\times3\to1}{\Longrightarrow} 224\times224\times64 \overset{2\times2\to1}{\Longrightarrow} 112\times112\times64\\
&\overset{3\times3\to1}{\Longrightarrow} 112\times112\times128 \overset{3\times3\to1}{\Longrightarrow} 112\times112\times128 \overset{2\times2\to1}{\Longrightarrow} 56\times56\times128\\
&\overset{3\times3\to1}{\Longrightarrow} 56\times56\times256 \overset{3\times3\to1}{\Longrightarrow} 56\times56\times256 \overset{3\times3\to1}{\Longrightarrow} 56\times56\times256 \overset{2\times2\to1}{\Longrightarrow} 28\times28\times256\\
&\overset{3\times3\to1}{\Longrightarrow} 28\times28\times512 \overset{3\times3\to1}{\Longrightarrow} 28\times28\times512 \overset{3\times3\to1}{\Longrightarrow} 28\times28\times512 \overset{2\times2\to1}{\Longrightarrow} 14\times14\times512\\
&\overset{3\times3\to1}{\Longrightarrow} 14\times14\times512 \overset{3\times3\to1}{\Longrightarrow} 14\times14\times512 \overset{3\times3\to1}{\Longrightarrow} 14\times14\times512 \overset{2\times2\to1}{\Longrightarrow} 7\times7\times512\\
&\Longrightarrow 25088 \Longrightarrow 4096 \Longrightarrow 4096 \Longrightarrow 1000 \overset{softmax}{\Longrightarrow} 1000
\end{aligned}
$$

or as "conv3-64, conv3-64, maxpool2, conv3-128, conv3-128, maxpool2, conv3-256, conv3-256, conv3-256, maxpool2, conv3-512, conv3-512, conv3-512, maxpool2, conv3-512, conv3-512, conv3-512, maxpool2, FC-4096, FC-4096, FC-1000, softmax-1000". This model has 138,357,544 parameters! The first convolutional layer has $(3\times3\times3+1)\times64 = 1792$ parameters.

In Keras, `application_vgg16()` will download the model and put it in `$HOME/.keras/models/`. The model file itself requires 550MB. The downloaded model does not have the last hidden layer (with 1000 neurons).

12.2.6 Example: prediction of 9-mer peptide binding in cancer immunotherapy

https://tensorflow.rstudio.com/blog/dl-for-cancer-immunotherapy.html

SB: strong binder; WB: weak binder; NB: non-binder

Milestones of ANNs:

- Backpropagation in NN (Rumelhart et al. 1986)
- MNIST dataset (1998): 70,000 28×28 images of handwritten digits, 10 classes

- **LeNet**: CNN was developed and applied to MNIST (LeCun et al. 1998)
- LSTM, a type of RNN (Hochreiter and Schmidhuber, 1997)
- NN in natural language processing (Bengio et al., 2003)
- Netflix Prize (2006–2009)
- CIFAR-10/CIFAR-100 datasets: 60,000 32×32 images, 10/100 classes.
- ImageNet dataset (Fei-Fei Li): Initially 3.2 million images in 5,247 "synsets". Now 13 million images.
- ILSVRC (ImageNet Large Scale Visual Recognition Challenge) (2010–2017): The catalyst for the **AI boom**.
  - AlexNet **brought DL into the mainstream**. It won the 2012 ILSVRC (top-5 error rate 15.4%, same below; 11x11 filters). They used ReLU rather than the conventional tanh function, and introduced dropout layers to overcome overfitting (Krizhevsky, Sutskever, Hinton, 2012).
  - VGG Net won the "classification+localization" category of the 2014 ILSVRC (error rate 7.3%; 3x3 filters). Showed that simple deep structures work for hierarchical feature extraction (Simonyan and Zisserman, 2014).
  - GoogLeNet/inception won 2014 ILSVRC (error rate 6.7%). Introduced the inception module, with CNN layers not stacked up sequentially (Szegedy et al., 2014). Later improvement: BN-inception-2 (error rate 4.8% with batch normalization) (Ioffe, Szegedy, 2015), inception-3 (error rate 3.6%) (Szegedy et al., 2015).
  - Microsoft ResNet won 2015 ILSVRC (error rate 3.6%). Introduced residual block to reduce overfitting. (He et al. 2015)
- Generative adversarial nets (GANs) (Goodfellow et al. 2014)
- AlphaGo (2016) and AlphaGo Zero (2017)

Pioneers in NNs: Geoffrey Hinton, Yann LeCun, Yoshua Bengio, etc.

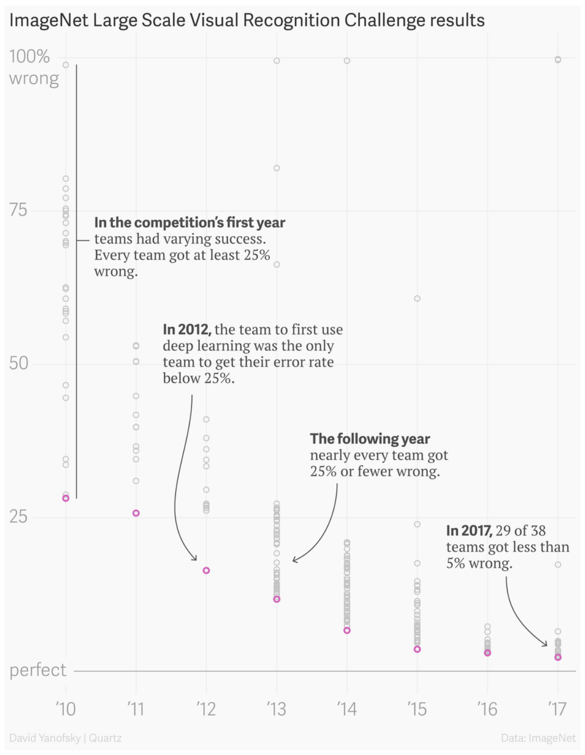


Figure from The data that transformed AI research—and possibly the world

12.2.7 Assignment

1. Reading for next lecture: NNDL 6.